Len Bass, Paul Clements, Rick Kazman

Software Architecture in Practice

Second Edition

Addison-Wesley

КЛАССИКА COMPUTER SCIENCE

Л. БАСС, П. КЛЕМЕНТС, Р. КАЦМАН

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ПРАКТИКЕ

2-Е ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2006

ББК 32.973-018
УДК 004.14
Б27

Басс Л., Клементс П., Кацман Р.
Б27 Архитектура программного обеспечения на практике. 2-е издание. — СПб.: Питер, 2006. — 575 с.: ил.

ISBN 5-469-00494-5

Основываясь на собственном, причем весьма обширном, опыте, авторы, с одной стороны, раскрывают основные технические вопросы проектирования, специфицирования и проверки правильности, а с другой — неизменно подчеркивают важность коммерческого контекста, в котором проводится проектирование крупных систем. Цель книги заключается в том, чтобы представить процесс разработки архитектуры программных систем как можно более реалистично, отразив как возможности, так и ограничения, с которыми сталкиваются компании. Приводимые в этой связи конкретные примеры успешных архитектурных решений демонстрируют основные технические и организационные моменты.

Всем, кто занимается проектированием, разработкой или координацией производства крупных программных систем (или планирует приступить к подобной деятельности), кто планирует заказать такую систему для своего предприятия или правительственного учреждения, книга поможет разобраться в современном состоянии программной архитектуры.

ББК 32.973-018
УДК 004.14

ISBN 0321154959 (англ.)
ISBN 5-469-00494-5

Краткое содержание

Предисловие 19

Благодарности 24

К читателю 26

Часть 1. Планирование архитектуры 33

Глава 1. Архитектурно-экономический цикл 35

Глава 2. Что такое «программная архитектура»? 51

Глава 3. Авиационная система A-7E: конкретный пример применения архитектурных структур 81

Часть 2. Создание архитектуры 104

Глава 4. Атрибуты качества 106

Глава 5. Реализация качества 136

Глава 6. Управление воздушным движением. Пример разработки, ориентированной на высокую готовность . 167

Глава 7. Проектирование архитектуры 194

Глава 8. Моделирование условий полета. Конкретный пример архитектуры, ориентированной на интегрируемость . . . 218

Глава 9. Документирование программной архитектуры 246

Глава 10. Реконструкция программной архитектуры 279

Часть 3. Анализ архитектуры 308

Глава 11. Метод анализа компромиссных архитектурных решений — комплексный подход к оценке архитектуры 316

Глава 12. Метод анализа стоимости и эффективности — количественный подход к принятию архитектурно-проектных решений 354

Глава 13. Всемирная паутина. Конкретный пример реализации способности к взаимодействию 374

Часть 4. От одной системы к множеству 400

Глава 14. Линейки программных продуктов. Повторное использование архитектурных средств 402

Глава 15. CelsiusTech. Конкретный пример разработки линейки продуктов 421

Глава 16. J2EE/EJB. Конкретный пример стандартной вычислительной инфраструктуры 455

Глава 17. Архитектура Luther. Конкретный пример мобильных приложений на основе архитектуры J2EE . . 483

Глава 18. Конструирование систем из коробочных компонентов 514

Глава 19. Будущее программной архитектуры 540

Сокращения 548

Библиография 554

Алфавитный указатель 561

Содержание

Предисловие **19**

Что нового во втором издании 21

Благодарности **24**

К читателю **26**

Целевая аудитория 26

Части и главы 26

Часть 1. Планирование архитектуры 27

Часть 2. Создание архитектуры 27

Часть 3. Анализ архитектуры 29

Часть 4. От одной системы к множеству 29

Систематика конкретных примеров 31

Основные темы книги 31

Примечания 32

От издательства 32

ЧАСТЬ 1. ПЛАНИРОВАНИЕ АРХИТЕКТУРЫ **33**

Глава 1. Архитектурно-экономический цикл **35**

1.1. Откуда берутся варианты архитектуры? 38

Влияние на архитектуру оказывают заинтересованные в системе лица 38

Влияние на архитектуру оказывает компания-разработчик 40

Влияние на архитектуру оказывают опыт и привычки архитекторов . . . 40

Влияние на архитектуру оказывает техническая база 41

Вариативность факторов влияния на архитектуру 41
Архитектура оказывает обратное воздействие на факторы влияния . . . 42
1.2. Программный процесс и архитектурно-экономический цикл 45
Этапы разработки архитектуры 45
1.3. Из чего складывается «качественная» архитектура? 47
1.4. Заключение 50
1.5. Дискуссионные вопросы 50

Глава 2. Что такое «программная архитектура»? 51

2.1. Чем является программная архитектура и чем она не является 51
2.2. Другие взгляды на архитектуру 55
2.3. Архитектурные образцы, эталонные модели и эталонные варианты архитектуры 57
2.4. Почему программная архитектура так важна? 59
Архитектура как средство организации общения между заинтересованными лицами 59
В архитектуре излагаются начальные проектные решения 61
Архитектура как переносимая модель многократного применения 65
2.5. Архитектурные структуры и представления 68
Программные структуры 70
Отношения между структурами 75
Какие структуры выбрать? 76
2.6. Заключение 77
2.7. Дополнительная литература 77
2.8. Дискуссионные вопросы 78

Глава 3. Авиационная система A-7E: конкретный пример применения архитектурных структур 81

3.1. Связь с архитектурно-экономическим циклом 82
3.2. Требования и атрибуты качества 83
3.3. Архитектура авиационной электронной системы A-7E 88
Структура декомпозиции 88
Структура использования 94
Структура процессов 98
3.4. Заключение 101
3.5. Дополнительная литература 103
3.6. Дискуссионные вопросы 103

ЧАСТЬ 2. СОЗДАНИЕ АРХИТЕКТУРЫ 104

Глава 4. Атрибуты качества 106

4.1. Функциональность и архитектура 107
4.2. Архитектура и атрибуты качества 108
4.3. Атрибуты качества системы 109
Сценарии атрибутов качества 110
Создание сценария атрибута качества 113
4.4. Практическое применение сценариев атрибутов качества 113
Готовность 114
Модифицируемость 116
Производительность 118
Безопасность 121
Контролепригодность 125
Практичность 127
Формулировка понятий в общих сценариях 130
4.5. Другие атрибуты качества системы 131
4.6. Коммерческие атрибуты качества 131
4.7. Атрибуты качества архитектуры 133
4.8. Заключение 134
4.9. Дополнительная литература 134
4.10. Дискуссионные вопросы 135

Глава 5. Реализация качества 136

5.1. Определение тактики 137
5.2. Тактики реализации готовности 138
Обнаружение неисправностей 139
Восстановление после неисправности 139
Предотвращение неисправностей 142
5.3. Тактики реализации модифицируемости 143
Локализация изменений 144
Предотвращение волнового эффекта 145
Откладывание связывания 149
5.4. Тактики реализации производительности 150
Потребление ресурсов 152
Управление ресурсами 153
Арбитраж ресурсов 153

5.5. Тактики реализации безопасности 155
Противодействие атакам 155
Обнаружение атак 157
Восстановление после атак 157
5.6. Тактики реализации контролепригодности 158
Входные/выходные данные 159
Внутренний мониторинг 159
5.7. Тактики реализации практичности 160
Тактики периода исполнения 161
Тактики периода проектирования 162
5.8. Взаимосвязь тактик и архитектурных образцов 162
5.9. Архитектурные образцы и стили 164
5.10. Заключение 165
5.11. Дополнительная литература 166
5.12. Дискуссионные вопросы 166

Глава 6. Управление воздушным движением. Пример разработки, ориентированной на высокую готовность 167

6.1. Связь с архитектурно-экономическим циклом 170
6.2. Требования и атрибуты качества 171
6.3. Архитектурное решение 174
Физическое представление системы ISSS 175
Представление декомпозиции модулей 177
Представление процессов 178
Клиент-серверное представление 181
Кодовое представление 181
Многоуровневое представление 182
Новое представление: отказоустойчивость 185
Взаимоотношения представлений 187
Адаптационные данные 188
Уточнение тактики «общие абстрактные службы»: кодовые шаблоны для приложений 189
6.4. Заключение 191
6.5. Дополнительная литература 192
6.6. Дискуссионные вопросы 193

Глава 7. Проектирование архитектуры 194

7.1. Архитектура в контексте жизненного цикла 195
Когда приступать к проектированию? 195
7.2. Проектирование архитектуры 196
Атрибутный метод проектирования 197
7.3. Формирование рабочих групп 210
7.4. Создание макета системы 213
7.5. Заключение 215
7.6. Дополнительная литература 215
7.7. Дискуссионные вопросы 216

Глава 8. Моделирование условий полета. Конкретный пример архитектуры, ориентированной на интегрируемость . . . 218

8.1. Связь с архитектурно-экономическим циклом 219
8.2. Требования и атрибуты качества 221
Применение моделей 222
Рабочие состояния 223
8.3. Архитектурное решение 226
Время в системе моделирования условий полета 227
Архитектурный образец «структурная модель» 229
Организующие модули модели воздушного судна 230
Прикладные модули модели воздушного судна 232
Макет системы 237
Распределение функциональности между дочерними модулями контроллера 237
Декомпозиция на группы 239
Декомпозиция групп на системы 240
8.4. Заключение 242
Производительность 243
Интегрируемость 243
Модифицируемость 244
8.5. Дополнительная литература 244
8.6. Дискуссионные вопросы 245

Глава 9. Документирование программной архитектуры . . . 246

9.1. Варианты применения архитектурной документации 247
9.2. Представления 250

9.3. Выбор значимых представлений 251
9.4. Документирование представления 253
Документирование поведения 257
Документирование интерфейсов 257
9.5. Перекрестная документация 262
Как документация адаптируется к задачам заинтересованных лиц 263
Что такое архитектура 264
Почему архитектура стала именно такой, какой стала 265
9.6. Унифицированный язык моделирования 265
Модульные представления 266
Представления из группы «компонент и соединитель» 269
Представления распределения 275
9.7. Заключение 277
9.8. Дополнительная литература 278
9.9. Дискуссионные вопросы 278

Глава 10. Реконструкция программной архитектуры 279

10.1. Введение 279
Принцип инструментария 281
Операции в ходе реконструкции 281
10.2. Извлечение информации 283
Практические рекомендации 285
10.3. Создание базы данных 286
Практические рекомендации 287
10.4. Объединение представлений 288
Импорт представления 288
Устранение неоднозначности вызовов функций 289
Практические рекомендации 290
10.5. Реконструкция 290
Практические рекомендации 294
10.6. Пример 296
Извлечение информации 296
Создание базы данных 297
Объединение представлений и реконструкция 297
10.7. Заключение 306
10.8. Дополнительная литература 306
10.9. Дискуссионные вопросы 307

ЧАСТЬ 3. АНАЛИЗ АРХИТЕКТУРЫ 308

Зачем? 308
Когда? 309
Затраты 309
Выгоды 310
Методики 312
Планировать или не планировать? 312
Предварительные условия 313
Результаты 314
Дополнительная литература 315

Глава 11. Метод анализа компромиссных архитектурных решений — комплексный подход к оценке архитектуры . . 316

11.1. Участники ATAM 317
11.2. Результаты проведения оценки по методу ATAM 319
11.3. Этапы ATAM 321
Операции на различных этапах оценки 322
Эффективное распоряжение ограниченными временными ресурсами 334
11.4. Система Nightingale: конкретный пример проведения оценки по методу ATAM 335
Нулевой этап: установление партнерских отношений и подготовка 335
Этап 1: оценка 337
Этап 2: оценка (продолжение) 347
Этап 3: доработка 351
11.5. Заключение 352
11.6. Дополнительная литература 353
11.7. Дискуссионные вопросы 353

Глава 12. Метод анализа стоимости и эффективности — количественный подход к принятию архитектурно-проектных решений 354

12.1. Контекст принятия решений 355
12.2. Основы CBAM 357
Полезность 357
Вычисление коэффициента ROI 361

12.3. Реализация CBAM 362
Этапы 362
12.4. Конкретный пример: проект ESC агентства NASA 364
Этап 1: критический анализ сценариев 365
Этап 2: уточнение сценариев 365
Этап 3: расстановка сценариев согласно приоритетам 367
Этап 4: установление полезности 367
Этап 5: разработка для сценариев архитектурных стратегий и установление их желаемых уровней реакции атрибута качества 368
Этап 6: определение полезности «ожидаемых» уровней реакции атрибута качества путем интерполяции 370
Этап 7: расчет общей выгоды, полученной от архитектурной стратегии 371
Этап 8: отбор архитектурных стратегий с учетом ROI, а также ограничений по стоимости и времени 371
12.5. Результаты оценки по методу CBAM 372
12.6. Заключение 372
12.7. Дополнительная литература 373
12.8. Дискуссионные вопросы 373

Глава 13. Всемирная паутина. Конкретный пример реализации способности к взаимодействию 374

13.1. Отношение к архитектурно-экономическому циклу 375
13.2. Требования и атрибуты качества 377
Первоначальные требования 378
Требования приходят и уходят 380
13.3. Архитектурное решение 382
Реализация первоначальных требований: libWWW 382
Выводы из опыта применения libWWW 384
Ранний вариант архитектуры «клиент–сервер», реализованный при помощи libWWW 385
Общий шлюзовой интерфейс (CGI) 387
Реализация первоначальных задач по качеству 389
13.4. Еще одна итерация архитектурно-экономического цикла: эволюция вариантов веб-архитектуры систем электронной коммерции 389
Браузеры ради модифицируемости 391

HTTPS ради безопасности 392
Прокси-серверы ради производительности 392
Маршрутизаторы и брандмауэры ради безопасности 392
Выравнивание нагрузки ради производительности, масштабируемости и готовности 393
Веб-серверы ради производительности 394
Серверы приложений ради модифицируемости, производительности и масштабируемости 394
Базы данных ради производительности, масштабируемости и готовности 395
13.5. Реализация задач по качеству 395
13.6. Архитектурно-экономический цикл сегодня 396
13.7. Заключение 397
13.8. Дополнительная литература 398
13.9. Дискуссионные вопросы 399

ЧАСТЬ 4. ОТ ОДНОЙ СИСТЕМЫ К МНОЖЕСТВУ 400

Глава 14. Линейки программных продуктов. Повторное использование архитектурных средств 402

14.1. Обзор 402
14.2. За счет чего работают линейки программных продуктов? 404
14.3. Определение области действия 407
14.4. Варианты архитектуры линеек продуктов 410
Установление изменяемых параметров 411
Обеспечение изменяемых параметров 411
Оценка архитектуры линейки продуктов 413
Что и как оценивать 413
Когда приступать к оценке 413
14.5. Факторы, усложняющие применение линеек программных продуктов 414
Стратегии принятия 415
Создание продуктов и развитие линейки продуктов 416
Организационная структура 417
14.6. Заключение 419
14.7. Дополнительная литература 419
14.8. Дискуссионный вопрос 420

Глава 15. CelsiusTech. Конкретный пример разработки линейки продуктов 421

15.1. Связь с архитектурно-экономическим циклом 422
Ship System 2000: линейка продуктов для BMC 423
Экономика линеек продуктов: обзор результатов, достигнутых CelsiusTech 425
Чем руководствовалась компания CelsiusTech 428
Все было внове 430
Анализ коммерческого контекста 430
Организационная структура CelsiusTech 433
15.2. Требования и атрибуты качества 441
Операционная среда и физическая архитектура 442
15.3. Архитектурное решение 444
Представление процессов: удовлетворение требований по распределению и средства расширения линейки продуктов . . 444
Многоуровневое представление 446
Представление декомпозиции на модули: системные функции и группы системных функций 447
Применение архитектуры SS2000 449
15.4. Заключение 453
15.5. Дополнительная литература 454
15.6. Дискуссионные вопросы 454

Глава 16. J2EE/EJB. Конкретный пример стандартной вычислительной инфраструктуры 455

16.1. Связь с архитектурно-экономическим циклом 456
16.2. Требования и атрибуты качества 457
Всемирная паутина и J2EE 458
16.3. Архитектурное решение 460
Архитектурная методика EJB 463
EJB-программирование 468
Дескрипторы размещения 472
16.4. Решения по размещению системы 475
Управление состоянием — старая проблема в новом контексте . . 475
Проблемы распределения и масштабирования 478
Организация пула ресурсов 479
Зависимость от производительности виртуальной машины Java . . . 480

16.5. Заключение 481
16.6. Дополнительная литература 481
16.7. Дискуссионные вопросы 482

Глава 17. Архитектура Luther. Конкретный пример мобильных приложений на основе архитектуры J2EE . . . 483

17.1. Связь с архитектурно-экономическим циклом 485
Факторы влияния на архитектуру 485
Влияние архитектуры на компанию 487
17.2. Требования и атрибуты качества 488
17.3. Архитектурное решение 492
Пользовательский интерфейс 494
Приложения 499
Компоненты 500
Пример повторно используемого компонента: компонент технологического управления 504
Следствия применения J2EE 510
17.4. Механизм реализации атрибутов качества в архитектуре Luther 511
17.5. Заключение 512
17.6. Дополнительная литература 512
17.7. Дискуссионные вопросы 513

Глава 18. Конструирование систем из коробочных компонентов 514

18.1. Воздействие компонентов на архитектуру 516
18.2. Архитектурное несоответствие 517
Методики исправления интерфейсных несоответствий 518
Методики обнаружения интерфейсных несоответствий 521
Методики предотвращения интерфейсных несоответствий 522
18.3. Компонентное проектирование как поиск 524
18.4. Пример приложения ASEILM 528
Ансамбль Miva Empressa 529
Ансамбль Java-сервлетов 534
18.5. Заключение 538
18.6. Дополнительная литература 539

Глава 19. Будущее программной архитектуры 540

19.1. Снова архитектурно-экономический цикл 542
19.2. Создание архитектуры 543
19.3. Архитектура в рамках жизненного цикла 544
19.4. Влияние коммерческих компонентов 546
19.5. Заключение 546

Сокращения 548

Библиография 554

Алфавитный указатель 561

Предисловие

Программная архитектура как область исследований весьма значительна; разговоров о ней с каждым днем ведется все больше. Тем не менее, насколько мы знаем, ощущается дефицит технических и административных руководств по управлению архитектурой программных систем в компаниях, занимающихся их разработкой. Своим появлением книга обязана нашему убеждению в недостаточной изученности программной архитектуры в коммерческом и организационном аспектах.

Накопленный опыт проектирования и анализа крупных и сложных, преимущественно программных, систем привел нас к выводу о том, что, в конечном итоге, конструкцию системы, ее успех или провал обусловливают коммерция и организация. Системы конструируются согласно реальным (или предполагаемым, как это происходит в случае с готовыми продуктами) требованиям компаний. Именно они определяют такие характеристики системы, как производительность, готовность, защита, совместимость с другими системами, а также способность приспосабливаться к разного рода изменениям в течение своей жизни. Желание удовлетворить эти цели за счет соответствующих свойств программного продукта навязывает программным архитекторам определенные конструктивные решения.

Переплетение программной архитектуры с коммерческим контекстом мы намерены проиллюстрировать конкретными примерами, взятыми из реальных систем. В частности, мы приведем следующие ситуации:

- Желание наладить внутри компании быстрый и беспрепятственный обмен документами, сведя централизованное управление к минимуму, привело к внедрению программной архитектуры Всемирной паутины.
- Высочайшие требования по безопасности при управлении воздушным движением привели одну из компаний к осознанию необходимости построения системы с архитектурой, ориентированной на достижение сверхготовности.
- Распределение подсистем пилотажного тренажера среди удаленных разработчиков обусловило создание архитектуры, ориентированной на постепенную интеграцию этих подсистем.
- Потребность в организации одновременных поставок многочисленных продуктов побудила (скорее даже заставила) одну из компаний внедрить

архитектуру, позволившую структурировать ряд сложных, связанных между собой программных систем в виде единой линейки продуктов.

- Необходимость в стандартизации архитектурных методик, применяемых в различных организациях, с одной стороны, и признанных сообществом в целом — с другой, определила появление таких инфраструктур, как J2EE и EJB.

Эти примеры, равно как и многие другие, заставляют сделать вывод о том, что конкретные программные архитектуры определяются требованиями организаций, их бизнес-моделями, опытом архитекторов и современными тенденциями в проектировании.

С другой стороны, все вышеперечисленные факторы иногда испытывают обратное влияние программных архитектур, и мы намерены показать, как это происходит. Если отдельный продукт или набор продуктов оказывается успешным, то другие продукты начинают строиться по его образу и подобию. Эта мысль прекрасно иллюстрируется конкретным примером программного обеспечения, на котором основывается Всемирная паутина. До ее появления сетевым технологиям и доступности данных уделялось значительно меньше внимания, а вопросы защиты информации беспокоили лишь отдельные организации — в основном финансовые институты и правительственные учреждения.

Читателями настоящей книги мы мыслим специалистов в области разработки программного обеспечения — людей, занимающихся проектированием и реализацией крупных, преимущественно программных, систем, и тех, кто управляет их деятельностью, а также студентов, которые в один прекрасный день надеются стать такими специалистами.

По нашему мнению, во всем, что касается качества, сроков и стоимости, программная архитектура предоставляет максимальную отдачу на вложенный капитал. Архитектура появляется на ранней стадии жизненного цикла продукта, и от ее качества зависит результат всех последующих этапов: разработки системы, интеграции, тестирования и модификации. Непродуманная архитектура дискредитирует сам остов системы; мелких исправлений в такой ситуации недостаточно — приходится все переделывать. В сравнении с другими операциями разработки анализ архитектуры довольно дешев. Итак, эффективность вложений в разработку архитектуры обусловливается, во-первых, существенными нисходящими последствиями принятия архитектурных решений, а во-вторых, относительной экономичностью проверки и наладки архитектуры.

Кроме того, нам кажется, что наилучшие возможности по многократному применению появляются именно в архитектурном контексте. Ведь артефакты, предполагающие повторное использование, не ограничиваются компонентами. Благодаря многократному применению архитектуры создаются семейства систем, а те, в свою очередь, обусловливают появление новых организационных структур и возможностей ведения бизнеса.

Значительная часть книги отведена на описание реальных вариантов архитектуры, ориентированных на решение реальных задач реальных организаций. Отобранные примеры иллюстрируют те альтернативы, среди которых архитекторы обычно выбирают наилучшие средства достижения качества; кроме того, они отражают влияние организационных целей на конечные системы.

Помимо конкретных примеров в книге предлагается ряд методик проектирования, производства и оценки программной архитектуры. Мы рассмотрим принципы интерпретации требований по качеству в контексте архитектуры, а также приемы построения вариантов архитектуры, отвечающих этим требованиям. Методы представления и реконструкции архитектуры мы намерены оценивать как средства описания архитектуры и проверки ее на правильность. Поговорим мы и о принципах анализа и оценки адекватности архитектуры относительно поставленных перед ней целей. Все эти принципы взяты из нашего собственного опыта работы с различными программными системами, а также из аналогичного опыта наших коллег по Институту программной инженерии. Некоторые из этих систем, насчитывающих миллионы строк кода, разрабатывались большими группами программистов на протяжении нескольких лет.

Обсуждаемые в книге экономические вопросы (в частности, влияние архитектуры на конкурентоспособность компаний и срок вывода семейства продуктов на рынок) представлены без особых изысков и специального жаргона. Какникак, мы разработчики программ, а не экономисты. Посему технические разделы раскрываются значительно глубже. В них мы стараемся отразить последние достижения в области программной архитектуры, акцентировать внимание на «точках» практической реализации исследовательских усилий. Экстраполяции теоретических основ на конкретные задачи служат многочисленные примеры. Для того чтобы осознать их ценность, требуется довольно серьезная подготовка в компьютерных науках, разработке программных средств или в смежных сферах. С другой стороны, мы постарались выстроить изложение конкретных примеров таким образом, чтобы избавить читателя от излишних тонкостей соответствующих прикладных областей. Так, для того чтобы сделать некоторые выводы из примеров, связанных с системой управления воздушным движением и моделированием условий полета, совершенно не обязательно быть летчиком.

Что нового во втором издании

Наши задачи с момента первого издания не изменились, однако с того времени в рассматриваемой области появились новые разработки и новое понимание основ программной архитектуры. Новшества из первой категории отражены через новые конкретные примеры, а из второй — через введение новых глав и уточнение старых. Определенное влияние на новый вариант текста оказали те книги, над которыми нам довелось работать в промежутке между появлением двух изданий: «Документирование программной архитектуры» (Documenting Software Architectures), «Оценка программной архитектуры: методы и примеры» (Evaluating Software Architectures: Methods and Case Studies) и «Линейки программных продуктов: теория и практика» (Software Product Lines: Principles and Practice). Эти работы, равно как и другие выполненные нами в последнее время технические и исследовательские задачи, наложили на содержание второго издания весьма ощутимый отпечаток. В нем, в частности, отражены все основные результаты разработок в сферах анализа, проектирования, реконструкции и документирования программной архитектуры, полученные с момента выхода первого издания.

Анализ архитектуры к настоящему моменту превратился в обширную область знаний с профессиональными методами; с учетом этого обстоятельства в третьей части книги мы ввели новую главу о методе анализа компромиссных архитектурных решений (Architecture Tradeoff Analysis Method, ATAM[SM]). Многие промышленные предприятия уже приняли ATAM в качестве стандартной методики оценки программной архитектуры.

Область архитектурного проектирования со времени выхода первого издания также подверглась значительным изменениям. В различных главах настоящей работы рассматриваются принципы фиксации требований по качеству, концепции их выполнения посредством малых и крупномасштабных архитектурных решений (тактик и образцов, соответственно) и метод проектирования, отражающий способы их выполнения. Требованиям по качеству, методам их удовлетворения и атрибутному методу проектирования (Attribute Driven Design Method, ADD) посвящены три новых главы.

Основной методикой фиксации недокументированной архитектуры является ее реконструкция, или обратная разработка. Она используется в рамках конструкторских и аналитических проектов и учитывается при принятии решения о выборе того или иного основания для реконструкции систем. В первом издании мы ограничились упоминанием о наборе инструментов Dali и кратким изложением вариантов его применения в контексте обратной разработки; теперь этой теме отведена отдельная глава.

В самое последнее время значительное развитие получила еще одна область — документирование программной архитектуры. В момент публикации первого издания унифицированный язык моделирования (Unified Modeling Language, UML) только начинал набирать обороты. Теперь, когда он полностью утвердился в своем качестве, в книге появилось множество новых диаграмм. Что еще важнее, решения о фиксации информации для архитектуры уже не ограничиваются простым выбором нотации. Документированию архитектуры во втором издании посвящена целая глава.

Сведения об эффективном производстве различных систем на основе единой архитектуры представлены в полностью переработанной главе о линейках программных продуктов. Особое внимание в ней уделяется связке архитектуры с производственными задачами предприятий — ведь линейки продуктов (основанные на программной архитектуре) способны на порядок снизить издержки, повысить качество и ускорить вывод продуктов на рынок.

В современных экономических условиях на первый план выходят технологии конструирования распределенных и веб-систем. Эта тенденция отражена в обновленной главе о Всемирной паутине; примеры, взятые из веб-систем, мы приводим в главах о методе ATAM и о построении систем на основе компонентов; старый конкретный пример с использованием обобщенной архитектуры построения брокеров объектных запросов (Common Object Request Broker Architecture, CORBA) заменен новым, построенным на основе системы корпоративных JavaBeans (Enterprise JavaBeans, EJB); кроме того, появился дополнительный конкретный пример беспроводной системы EJB — она предназначена для специалистов по обслуживанию, оснащенных переносными компьютерами.

Наконец, одна из новых глав посвящена относительно детальному рассмотрению финансовых аспектов программной архитектуры. В ней мы описываем новый метод анализа стоимости и эффективности (Cost Benefit Analysis Method, CBAM) — с его помощью при принятии архитектурных решений учитываются не только вышеозначенные технические, но и экономические критерии.

Аналогично первому изданию, объединяющей идеей в настоящей книге является архитектурно-экономический цикл (Architecture Business Cycle, ABC). Все конкретные примеры, таким образом, характеризуются с позиции задач обеспечения качества, определивших конструкции соответствующих систем, и реализованных архитектурой принципов их решения.

Работая над вторым изданием, мы, как и в прошлый раз, прекрасно отдавали себе отчет в том, что основную аудиторию настоящей книги составляют специалисты-практики. Исходя из этого, основное внимание было сосредоточено на материале, неоднократно нашедшем применение в промышленности, а также на перспективных, с нашей точки зрения, разработках.

Очень хочется надеяться, что от чтения второго издания вы получите не меньше удовольствия, чем мы получили от его написания!

Благодарности

Без первого издания этой книги не было бы второго, и мы не устаем благодарить всех тех, кто вместе с нами работал над ее первоначальной версией. В качестве соавторов отдельных глав выступили Грегори Абауд (Gregory Abowd), Лайза Браунсуорд (Lisa Brownsword), Джеромы Карье (Jeromy Carriere), Линда Нортроп (Linda Northrop), Патриция Оберндорф (Patricia Oberndorf), Мэри Шоу (Mary Shaw), Роб Велтр (Rob Veltre), Курт Валнау (Kurt Wallnau), Нельсон Вайдерман (Nelson Weiderman) и Эйми Мурман-Заремски (Amy Moormann Zaremski). Поддержкой и одобрением нас радовали многие сотрудники Института программной инженерии — в частности, Линда Нортроп, Шолом Коэн (Sholom Cohen), Лайза Лейн (Lisa Lane), Билл Поллак (Bill Pollak), Барбара Томчик (Barbara Tomchik) и Барбара Уайт (Barbara White).

Мы в большом долгу перед многочисленными редакторами: Феликсом Бахманом (Felix Bachmann), Джоном Беннеттом (John Bennett), Соуньей Бот (Sonia Bot), Лайзой Браунсуорд (Lisa Brownsword), Бобом Элисоном (Bob Ellison), Ларри Говардом (Larry Howard), Ричардом Джуреном (Richard Juren), Филиппом Крюхтеном (Philippe Kruchten), Чун-Хон Лун (Chung-Horng Lung), Хоакином Миллером (Joaquin Miller), Линдой Нортроп, Дэвидом Ноткином (David Notkin), Патрицией Оберндорф (Patricia Oberndorf), Яном Пахлем (Jan Pachl), Люи Ша (Lui Sha), Нельсоном Вайдерманом (Nelson Weiderman), Эйми Мурман-Заремски и еще несколькими сотрудниками издательства Addison-Wesley, имена которых нам неизвестны. Капитан военно-морского флота США Роб Мэдсон (Rob Madson) участвовал в составлении графических иллюстраций, а Питер Гордон (Peter Gordon) из Addison-Wesley не позволял нам отрываться от действительности.

Что касается второго издания, то здесь отдельных похвал также заслуживают соавторы глав: Линда Нортроп, Феликс Бахман, Марк Кляйн (Mark Klein), Билл Вуд (Bill Wood), Дэвид Гарлан (David Garlan), Джеймс Айверс (James Ivers), Рид Литтл (Reed Little), Роберт Норд (Robert Nord), Джудит Стеффорд (Judith Stafford), Джеромы Карье, Лайам О'Брайен (Liam O'Brien), Крис Вероеф (Chris Verhoef), Джей Асунди (Jai Asundi), Хон-Мей Чен (Hong-Mei Chen), Лайза Браунсуорд, Анна Лиу (Anna Liu), Таня Басс (Tanya Bass), Джеймс Бек (James Beck), Келли Долан (Kelly Dolan), Куйвей Ли (Cuiwei Li), Андреас Лор (Andreas Lohr), Ричард Мартин (Richard Martin), Уильям Росс (William Ross), Тобиас Вайсхаупль (Tobias Weishaupl), Грегори Железник (Gregory Zelesnik), Роберт Сикорд (Robert Seacord) и Мэтью Басс (Matthew Bass). А что бы мы делали без редакторов?

Большое спасибо Александеру Рэну (Alexander Ran), Пауло Мерсону (Paulo Merson), Мэтту Бассу (Matt Bass), Тони Латтанце (Tony Lattanze), Лайаму О'Брайену и Роберту Норду.

Над материалом, связанным с выявлением и обеспечением атрибутов качества программных продуктов, кроме нас, трудилось множество людей. В этой связи следует особо отметить Джона Макгрегора (John McGregor), Боба Элисона, Энди Мура (Andy Moore), Скотта Хиссама (Scott Hissam), Чака Вайнстока (Chuck Weinstock), Марио Барбаччи (Mario Barbacci), Хизер Оппенхаймер (Heather Oppenheimer), Феликса Бахмана, Стефана Ковалевски (Stefen Kowalewski) и Марко Ауэрсвальда (Marko Auerswald).

Отдельное спасибо Майку Муру (Mike Moore) из центра космических полетов им. Годара NASA; именно ему мы обязаны возможностью поработать с центральной системой наблюдения за поверхностью Земли (ECS), на примере которой в главе 12 разбирается метод анализа стоимости и эффективности (CBAM).

Из сотрудников SEI необходимо отметить Линду Нортроп, которая координировала нашу работу, поддерживала боевой дух и вносила ценные замечания, Боба Фантазьера (Bob Fantazier), нашего бессменного иллюстратора, Шейлу Розенталь (Sheila Rosenthal), внесшую значительный вклад в исследовательскую работу, а также Лору Новачиц (Laura Novacic), Кэролин Кернан (Carolyn Kernan) и Барбару Томчик — поддержку, оказанную этими людьми, мы очень ценим.

Питер Гордон из Addison-Wesley, наш руководитель, как всегда, действует методом кнута и пряника. Мы крайне признательны ему и всем остальным сотрудникам издательства, так или иначе причастным к работе над настоящей книгой.

Часть технологической работы, связанной с подготовкой этого издания, Лен Басс выполнил в период посещения Научно-промышленной исследовательской организации Содружества Наций (Commonwealth Scientific Industrial Research Organization), расположенной в Австралии. Он благодарен ее сотрудникам за оказанную помощь.

Наконец, спасибо всем нашим близким за то, что в период работы над книгой они нас не только терпели, но и подбадривали.

К читателю

Целевая аудитория

Материал, изложенный в настоящей книге, должен представлять интерес для специалистов в области разработки программного обеспечения, а также для студентов, обладающих определенными познаниями и опытом в сфере программной инженерии. По нашему мнению, аудитория этого издания делится на три категории:

- практикующие разработчики программного обеспечения, желающие освоить техническую базу архитектуры программных систем и разобраться в коммерческих и организационных факторах, влияющих на ее конкретные очертания;
- технические руководители, желающие понять, каким образом с помощью программной архитектуры можно повысить эффективность контроля над конструированием систем и качество их организации;
- студенты первых и вторых курсов отделений компьютерных наук и программной инженерии, для которых данная книга может выступить в качестве дополнительного руководства.

Части и главы

Материал настоящего издания подразделяется на четыре части, которые в общих чертах соответствуют жизненному циклу продукта, или, как мы его называем, *архитектурно-экономическому циклу* (Architecture Business Cycle, ABC) существования архитектуры в коммерческом контексте:

- планирование архитектуры (главы 1–3);
- создание архитектуры (главы 4–10);
- анализ архитектуры (главы 11–13);
- переход от одной системы к множеству (главы 14–19).

Конкретные примеры приводятся в главах 3, 6, 8, 13, 15, 16 и 17 и четко обозначаются в их названиях.

Рассмотрим содержание частей и глав несколько подробнее.

Часть 1. Планирование архитектуры

Глава 1. Архитектурно-экономический цикл. Основная мысль, которую мы продвигаем на протяжении всей книги, заключается в том, что все варианты архитектуры, не являясь самоценными, существуют в рамках цикла. Любая архитектура — это лишь средство достижения поставленной цели. Ее свойства определяются функциональными задачами, а также задачами качества как заказчика, так и компании-разработчика. Среди прочих факторов влияния следует отметить уровень подготовки и опыт архитектора, а также доступные технические средства. Архитектура, в свою очередь, оказывает воздействие на разрабатываемую систему, а в качестве одного из основных активов способна даже определить дальнейшее развитие компании-разработчика. Фактором влияния на компанию, архитектуру и (возможно) техническую базу является сама система. При этом вырабатываются перспективные задачи самой системы и компании-разработчика в целом. Из всех этих влияний, а также из сопутствующих архитектуре цепей обратной связи образуется архитектурно-экономический цикл.

Глава 2. Что такое программная архитектура? Архитектура представляет собой описание всех структур системы (а именно: структуры декомпозиции на модули, структуры процессов, структуры размещения и уровневой структуры). Архитектура — это первый артефакт, который можно проанализировать на предмет адекватности обеспечения качественных свойств системы; она же выступает в качестве детального плана проекта. Архитектура — это одновременно и средство коммуникации, и изложение первоначальных проектных решений, и абстракция, предполагающая возможность многократного применения и экстраполяции на последующие системы. Именно это мы имеем в виду, рассуждая об «архитектуре».

Глава 3. Авиационная система A-7E: конкретный пример применения архитектурных структур. В процессе создания авиационной электронной системы A-7E особое внимание уделялось конструированию и специфицированию трех четко выраженных архитектурных структур; тем самым предполагалось упростить разработку и обеспечить модифицируемость. В главе 3 мы объясняем, как (и зачем) эти структуры проектировались и документировались.

Часть 2. Создание архитектуры

Глава 4. Атрибуты качества. Основным фактором разработки любой архитектуры является намерение придать программному продукту определенное качество. В этой главе мы рассматриваем атрибуты качества программных продуктов и их содержание. Кроме того, в ней излагается метод интерпретации атрибутов качества в архитектурных категориях; в частности, речь идет об описании стимулов, которые в приложении к системе позволяют выявлять атрибуты ее качества, и о четком, измеримом формулировании реакций на них со стороны системы.

Глава 5. Реализация качества. Определившись с атрибутами качества, которыми должна обладать предполагаемая система, остается спроектировать архитектуру, в рамках которой их можно реализовать. Рассматриваемые в данной главе методики ориентированы на придание системе качества периодов разработки и прогона. В качестве основных механизмов достижения этой цели выступают

тактики (tactics) — проектные решения, определяющие управление атрибутами качества. Из нескольких тактик образуются архитектурные стратегии и архитектурные образцы.

Глава 6. Управление воздушным движением. Конкретный пример разработки, ориентированной на высокую готовность. Задача обеспечения качества, поставленная в период разработки рассматриваемой системы управления воздушным движением, заключалась в обеспечении сверхвысокой готовности. Именно этой целью объясняется принятие ряда оригинальных архитектурных решений, которые мы также намерены разобрать. Акцент в этом конкретном примере ставится на взаимодействие архитектурных структур и представлений, с одной стороны (см. главу 2), и архитектурных тактик — с другой (см. главу 5); здесь показано, каким образом их совместные действия помогают реализовывать атрибуты качества.

Глава 7. Создание архитектуры. Разобравшись с основными инструментами (архитектурными представлениями и структурами, выражением атрибутов качества, тактиками и образцами их реализации), мы можем, наконец, обратиться непосредственно к созданию архитектуры. Функции архитектуры в данной главе рассматриваются с точки зрения жизненного цикла системы в целом. В ней, в частности, представлен метод проектирования, при помощи которого очень удобно формулировать ранние варианты архитектуры, а впоследствии их можно уточнять и развивать. При наличии первоначальной, упрощенной схемы архитектуры уже можно приступать к формированию группы разработчиков проекта и созданию макета системы, на основе которого впоследствии будет проводиться пошаговая (инкрементная) разработка.

Глава 8. Моделирование условий полета. Конкретный пример архитектуры, ориентированной на интегрируемость. В этой главе рассматривается архитектура систем для моделирования условий полета. На основе тщательно продуманной программной архитектуры сложной предметной области разработчики смогли сконструировать ряд крупных систем. Эти системы были полностью приведены в соответствие со строжайшими требованиями по функциональности и точности; они понятны специалистам по разработке, легко поддаются интеграции и нисходящим модификациям.

Глава 9. Документирование программной архитектуры. Архитектура полезна лишь в том случае, если в ней могут разобраться представители заинтересованной группы. В данной главе излагается методика документирования программной архитектуры. Суть этого процесса в основном состоит в фиксации отдельных значимых представлений, а также тех сведений, которые актуальны для архитектуры в целом. Здесь же мы приводим шаблоны представлений, шаблоны информации о перекрестном представлении и программных интерфейсов.

Глава 10. Реконструкция программной архитектуры. Предположим, что мы столкнулись с некоей системой, архитектура которой нам неизвестна. Быть может, она не фиксировалась, или документация потеряна, или в процессе развития архитектура и система слишком сильно разошлись. Возможно ли сопровождение такой системы? Как направить ее развитие таким образом, чтобы реализовать предусмотренные архитектурой атрибуты качества? Процесс, в ходе которого на

основе существующей системы восстанавливается архитектура реализованной системы в своем изначальном состоянии («as-built»), называется реконструкцией архитектуры. Методика реконструкции в этой главе приводится вместе с примером ее применения.

Часть 3. Анализ архитектуры

Глава 11. Метод анализа компромиссных архитектурных решений — комплексный подход к оценке архитектуры. Метод анализа компромиссных архитектурных решений (Architecture Tradeoff Analysis Method, ATAM) позволяет оценить архитектурные решения в контексте требований к поведению и атрибутам качества. Наряду с описанием метода ATAM в этой главе приводится полноценный пример его применения.

Глава 12. Метод анализа стоимости и эффективности — количественный подход к принятию архитектурно-проектных решений. Программные архитекторы и руководители проектов всегда стремятся довести до максимума разницу между прибылью от системы и стоимостью ее реализации. Метод анализа стоимости и эффективности (Cost Benefit Analysis Method, CBAM) позволяет принимать экономические решения, исходя из анализа архитектуры. Основываясь на ATAM, метод CBAM обеспечивает возможность моделирования издержек и прибыли архитектурно-проектных решений и предусматривает средства их оптимизации. В этой главе мы не только представим метод CBAM, но и охарактеризуем конкретный пример его применения.

Глава 13. Всемирная паутина. Конкретный пример реализации способности к взаимодействию. Начало развитию Всемирной паутины положило намерение руководства отдельно взятой организации наладить обмен информацией между штатными исследователями; в контексте конечного результата эта первоначальная задача кажется незначительной. В данной главе описывается архитектура программного обеспечения, на основе которого функционирует Всемирная паутина, объясняется, каким образом оно определило дальнейший рост глобальной сети и какое влияние этот рост в свою очередь оказал на обращающиеся к ее услугам организации.

Часть 4. От одной системы к множеству

Глава 14. Линейки продуктов. Повторное использование архитектурных средств. Программная архитектура, применяемая в качестве основы для формирования линейки программных продуктов, оказывается крайне эффективной. В этой главе представлены элементарные принципы производства линеек программных продуктов, причем архитектура преподносится в роли основного фактора обеспечения подвижек по части продуктивности, сроков выхода на рынок, качества и затрат. Ряд действий в рамках разработки программных средств и управления этим процессом рассматриваются в этой главе достаточно подробно, поскольку в контексте формирования линеек продуктов они занимают особое место.

Глава 15. CelsiusTech. Конкретный пример разработки линейки продуктов. CelsiusTech — это название компании, которой удалось с успехом реализовать выстроенную на архитектуре линейку продуктов. Эта архитектура и является предметом рассмотрения в данной главе; здесь мы делаем попытку объяснить, почему именно архитектура оказалась основным условием достижений CelsiusTech. Выбери компания какую-либо другую методику, ей не удалось бы сконструировать намеченные системы — у нее просто не хватило бы сотрудников. Ориентация на линейки продуктов отразилась как на организационной структуре компании, так и на стиле аргументации и ведения переговоров с клиентами.

Глава 16. J2EE/EJB. Конкретный пример стандартной вычислительной инфраструктуры. В этой главе речь идет о спецификации корпоративной архитектуры Java 2 (Java 2 Enterprise Edition, J2EE) от компании Sun Microsystems, а также об одной из ее важнейших составляющих — архитектурной спецификации Enterprise JavaBeans (Enterprise JavaBeans, EJB). Спецификация J2EE содержит стандартное описание процессов проектирования и разработки распределенных объектно-ориентированных программ на языке Java. Мы анализируем коммерческие факторы, обусловившие создание стандартной архитектуры производства распределенных систем, а также рассматриваем ориентированные на удовлетворение соответствующих потребностей средства J2EE/EJB.

Глава 17. Архитектура Luther. Конкретный пример мобильных приложений на основе архитектуры J2EE. Архитектура Luther изначально мыслилась как универсальная структура, позволяющая внедрять специализированные решения в предметной области технического обслуживания и эксплуатации крупногабаритных транспортных средств и в рамках промышленной инфраструктуры. Поскольку в ее основе лежит архитектура J2EE, эту главу можно считать обзором одного из вариантов применения рассматриваемой в главе 16 универсальной структуры J2EE/EJB. Приведенный в ней конкретный пример ориентирован на такую среду, в которой конечный пользователь, располагая соединением по беспроводной сети, оперирует неким устройством с ограниченными возможностями ввода-вывода и/или ограниченными вычислительными возможностями.

Глава 18. Конструирование систем из коробочных компонентов. Чем дальше, тем больше в процессе конструирования систем используется готовых, «коробочных», компонентов. Поскольку они способны накладывать на архитектуру определенные ограничения, их использование некоторым образом видоизменяет процесс проектирования. Как правило, отбор компонентов диктуется намерением реализовать некий набор функциональных возможностей; с другой стороны, компоненты предполагают некие архитектурные допущения, а следовательно, и допущения в отношении качества. В этой главе рассматривается довольно простой процесс, при помощи которого любой архитектор сможет отобрать только те компоненты, которые способны к успешному взаимодействию. Иллюстрируется эта методика на примере недавно созданной системы.

Глава 19. Будущее программной архитектуры. Здесь мы еще раз пробежимся по архитектурно-экономическую циклу, попробуем сформулировать задачи программной архитектуры, которые еще предстоит решить, и приведем доводы в пользу проведения дальнейших исследований в этой области.

Систематика конкретных примеров

Для нас совершенно очевидно, что разным читателям требуются разные сведения, а для большинства из них важна возможность усвоения материала с разной степенью детализации. По этой причине все конкретные примеры делятся на несколько категорий:

- краткое описание примера и решаемых им задач плюс замечания о соответствующей программной архитектуре;
- описание (полной или частичной) реализации в данном примере архитектурно-экономического цикла;
- требования и атрибуты качества, обусловившие принятие конкретного проектного решения;
- подробное рассмотрение архитектурного решения — именно на нем основана бóльшая часть конкретных примеров;
- краткое изложение важнейших проблем, представленных в данной главе.

Архитектурные решения в конкретных примерах описываются наиболее подробно. Если сведения, которые вы считаете нужным извлечь, исчерпываются технической базой, коммерческим контекстом и высокоуровневым описанием архитектурной методики, то для того, чтобы усвоить суть примера, вам достаточно ознакомиться с его краткой характеристикой, требованиями, задачами по качеству и выводами. Более подробный анализ примеров содержится в разделах, посвященных архитектурным решениям.

Основные темы книги

Связующей нитью материала в этой книге, несомненно, является архитектурно-экономический цикл; тем не менее в нем есть и другие магистральные темы. Если ваши интересы связаны с каким-то одним или несколькими аспектами программной архитектуры, то с помощью нижеследующего указателя вы можете проследить их упоминания в различных главах.

- Откуда берутся варианты архитектуры? — главы 1, 2, 4, 7, 11 и 12.
- Экономические аспекты архитектуры — главы 1, 4, 7, 11, 12, 14, 15 и 18.
- Каким образом на основе архитектуры реализуются атрибуты качества? — главы 4, 5, 11, 12 и конкретные примеры;
- Конкретные примеры реализации атрибутов качества на основе архитектуры — главы 3, 6, 8, 13, 15, 16 и 17.
- Архитектура как средство многократного применения — главы 14, 15, 16, 17 и 18.
- Компонентные системы и коммерческие инфраструктуры — главы 13, 16, 17 и 18.
- Архитектура систем реального времени — главы 3, 5, 6, 8 и 15.
- Архитектура информационных систем — главы 13, 16, 17 и 18.

Примечания

На всем протяжении книги встречаются краткие, подписанные, отделенные от основного текста примечания — над написанием каждого такого примечания трудился кто-то один из нас. Нужны они для того, чтобы донести до читателя какие-то дополнительные сведения или мнения, которые в силу разных причин не вписываются в рамки основного повествования.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.

Часть 1
ПЛАНИРОВАНИЕ АРХИТЕКТУРЫ

Откуда берутся различные варианты архитектуры? Их, естественно, выдумывают архитекторы. Что должен знать архитектор, чтобы сформулировать архитектуру? Что же, наконец, называется архитектурой? Тождественно ли это понятие «проекту»? Если это одно и то же, тогда откуда вокруг «архитектуры» такой ажиотаж? Если это разные вещи, то в чем разница, и почему это так важно?

Часть 1 посвящена рассмотрению факторов и движущих сил, с которыми любой архитектор сталкивается на первом этапе своей работы — на этапе *планирования* (envisioning), то есть создания важнейшего артефакта системы, влияние которого распространяется за рамки ее жизненного цикла. Под процессом проектирования, как правило, понимаются некие действия, направленные на обеспечение предсказуемости системы по части предъявления ею верных ответов и наличия в ней ожидаемых функций; архитектура же затрагивает более долгосрочные проблемы. Архитектору приходится уравновешивать множество конкурирующих, а иногда даже конфликтующих факторов влияния и потребностей; что самое замечательное, лишь немногие из них ориентированы на корректную работу системы. Организационные и технические обстоятельства привносят в архитектуру ощутимое количество дополнительных, иногда неявных, потребностей, и, практически никогда не фиксируемые, они на деле оказываются не менее значимыми, чем явные требования к свойствам программных продуктов.

Не менее занимательно то, какими путями архитектура оказывает влияние, причем довольно серьезное, на порождающую ее компанию. Полагать, что компания создает архитектуру, а затем, привязав ее к разрабатываемой системе, забывает, наивно. На самом деле между вариантами архитектуры и соответствующими компаниями существуют сложные отношения прямых и обратных влияний, способствующие росту, развитию и расширению сфер действия обеих сторон.

Эти отношения мы называем архитектурно-экономическим циклом (Architecture Business Cycle, ABC); от этого понятия мы отталкиваемся на всем протяжении книги, а подробный его анализ приводится в главе 1. Глава 2 готовит почву для углубленного изучения программной архитектуры — в частности, раскрывает

само это понятие, ориентирует его в контексте программной инженерии и наводит на ряд понятийных средств. Главное, что вам предстоит усвоить, — это то, что любая архитектура состоит из ряда отдельных взаимосогласованных структур, которые в процессе разработки системы позволяют решать те или иные инженерные задачи.

Глава 3 содержит первый в книге конкретный пример. Он демонстрирует архитектуру, при помощи которой удалось не только выполнить четко определенный набор требований (речь в данном случае идет о встроенной авиационной системе реального времени, ориентированной на долгосрочную модифицируемость), но и развить вышеупомянутые концептуальные проблемы. В качестве архитектурного решения для этой системы были выбраны три отдельные архитектурные структуры: декомпозиция модулей, варианты применения и структуры процессов.

Итак, приступим к обзору архитектурно-экономического цикла.

Глава 1
Архитектурно-экономический цикл

> Проще говоря, для того чтобы добиться преимущества, компания должна установить единоличный архитектурный контроль над широким, непостоянным конкурентным полем.
>
> *К. Моррис & К. Фергюсон [Morris 93]*

Проектировщиков программного обеспечения десятилетиями учили конструировать системы, отталкиваясь исключительно от технических требований. Имелось в виду, что сводка требований должна висеть на стене в кабинете проектировщика, а тот должен думать, как их все соблюсти. Из требований складывалось проектное решение, а из проектного решения — система. Современные методы разработки программного обеспечения, конечно, отошли от этой весьма безыскусной модели, и теперь между всеми действующими лицами, от проектировщика до аналитика, существует обратная связь. И тем не менее — все они до сих пор предполагают, что проектное решение должно строиться на технических требованиях, и все тут.

Архитектура (architecture) — предмет настоящего исследования — является неотъемлемым этапом процесса проектирования. *Программная архитектура* (software architecture) содержит в себе структуры, из которых складываются крупные программные системы. Архитектурное представление системы абстрактно; не затрагивая детали реализации, алгоритмы и представление данных, оно ориентировано на поведение и взаимодействие «черных ящиков». Появление программной архитектуры — это первый шаг на пути к созданию системы с заданными свойствами. Подробно программная архитектура рассматривается в главе 2. Пока что, не вдаваясь в подробности, мы приведем ее определение.

> Программной архитектурой программы или вычислительной системы называется ее структура или структуры, заключающие в себе программные элементы, их внешние свойства и взаимосвязи.

Все рабочие определения и анализ различий между архитектурой и другими разновидностями проектных решений приводятся в главе 2. По некоторым причинам, которые мы намерены постепенно озвучивать, архитектура в контексте систем исполняет весьма значительную роль средства коммуникации, построения умозаключений, анализа и наращивания. До последнего времени архитектурное проектирование рассматривалось под одним углом — предполагалось, что для построения архитектуры системы необходимо знать только предъявленные к ней требования.

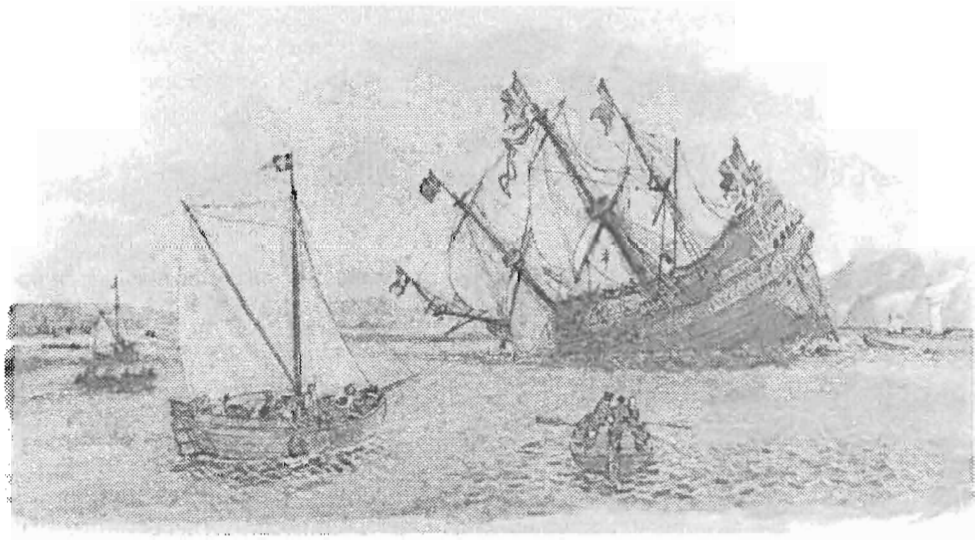


Рис. 1.1. Военный корабль «Ваза»
(приводится с разрешения Музея «Ваза» (Стокгольм, Швеция))

ШВЕДСКИЙ ВОЕННЫЙ КОРАБЛЬ «ВАЗА»

1620-е годы ознаменовались войной между Швецией и Польшей. Король Швеции Густав Адольф, предполагая завершить ее быстро и победоносно, распорядился спустить на воду новый военный корабль невиданной ранее мощи. «Ваза», изображенный на рис. 1.1, должен был стать самым грозным орудием войны во всем мире. Его длина составляла 70 метров, на нем помещалось 300 солдат, а на двух орудийных палубах было установлено 64 тяжелых орудия. Желая обеспечить своему флоту подавляющее преимущество и нанести противнику решающий удар на море, король потребовал загрузить «Ваза» орудиями до отказа. Создателем корабля был Хенрик Хибертсон (Henrik Hybertsson) — опытный голландский судостроитель с безупречной репутацией, — но и для него столь грандиозный проект был первым. Двухпалубные корабли в то время встречались крайне редко, и среди них не было ни одного, сравнимого с «Ваза» по размерам и вооружению.

Как и любой архитектор, вынужденный учитывать имеющийся опыт, Хибертсон оказался в ситуации, в которой ему пришлось уравновешивать множество разнородных интересов. Одинаково важно было обеспечить эффективность, функциональность, безопасность, надежность корабля, спустить его на воду как можно быстрее, и все это как можно дешевле. В качестве главного заказчика в данном случае выступал король, однако это не освобождало судостроителя от ответственности перед всеми членами экипажа. Привлекая к выполнению задачи весь накопленный опыт и действуя согласно современному уровню технической мысли, Хибертсон решил спроектировать «Ваза» по подобию однопалубных кораблей, а затем экстраполировать на результат свойства двухпалубного. В определенный момент Хибертсон осознал

безнадежность этого проекта, но ему повезло умереть примерно за год до окончания строительства.

Громадное судно, построенное согласно его инструкциям, спустили на воду 10 августа 1628 года. Сразу после пушечного салюта, произведенного по выходе в глубокую стокгольмскую гавань, корабль накренился на борт. Вода хлынула в трюм через орудийные порты, и «Ваза» опрокинулся. Через несколько минут его первое и единственное плавание закончилось на 30-метровой глубине. Десятки членов экипажа, составившего в общей сумме 150 человек, утонули.

По результатам проведенного расследования установили, что корабль был «скверно спроектирован». Другими словами, никчемной оказалась его архитектура. С высоты сегодняшнего уровня знаний мы можем заключить, что Хибертсон не справился с балансированием противоречивых ограничений. В частности, в его деятельности отсутствовали управление рисками и требованиями заказчика (хотя, вероятно, лучше с этой задачей не справился бы никто). Он молча согласился с изложенными требованиями, хотя те были невыполнимы.

История «Ваза», насчитывающая уже 375 лет, служит прекрасной иллюстрацией архитектурно-экономическому циклу: задачи компании определяют требования, требования определяют архитектуру, а архитектура — систему. Архитектура ограничена опытом архитектора и современным уровнем технической оснащенности. У Хибертсона не было ни опыта, ни необходимых технических средств.

Наша книга содержит три элемента, которые могли бы ему пригодиться:

1. Конкретные примеры удачной архитектуры, которые, с одной стороны, удовлетворяют выдвинутым требованиям, а с другой — расширяют современную техническую базу.
2. Методы оценки архитектуры, проводящейся до построения на ее основе систем и, таким образом, способствующей снижению рисков, связанных с реализацией беспрецедентных проектов.
3. Приемы инкрементной (пошаговой) архитектурно-ориентированной разработки, позволяющей своевременно исправлять изъяны проектов.

Мы хотим, чтобы современные архитекторы не попадали в такие же затруднительные ситуации, как несчастный голландский судостроитель, и с этой целью представляем на их рассмотрение различные варианты разрешения проектных дилемм. Умереть до развертывания системы в наши дни не слишком почетно.

— PCC

Эта позиция недальновидна (см. врезку «Шведский военный корабль “Ваза”») и неполна. Как вы думаете, что произойдет, если двум архитекторам, работающим в разных компаниях, представить одну и ту же спецификацию требований к системе? Будут ли созданные ими варианты архитектуры тождественны?

Как правило, в таких случаях варианты архитектуры получаются разными; следовательно, утверждение о том, что архитектуру определяют исключительно требования, неверно. Существуют и другие факторы влияния, и не обращать на них внимания — все равно что работать впотьмах.

Поставим уточняющий вопрос: какова связь между программной архитектурой системы, с одной стороны, и средой, в которой эта система конструируется и используется? Ответ на этот вопрос составляет лейтмотив настоящей книги. Программная архитектура появляется как результат взаимодействия *технических*, *экономических* и *социальных* факторов влияния. Впоследствии архитектура оказывает обратное, причем довольно существенное, воздействие на технические, экономические и социальные условия, а также косвенное влияние на последующие варианты архитектуры. Эту совокупность влияний, распространяющихся от среды на архитектуру, а затем обратно на среду, мы предпочитаем называть *архитектурно-экономическим циклом* (Architecture Business Cycle, ABC).

В настоящей главе мы проводим подробный анализ АВС и тем самым подготавливаем почву для подачи всего последующего материала. В четырех частях книги цикл рассматривается с различных точек зрения:

- каким образом задачи компании определяют требования и стратегию разработки;
- как на основе требований получается архитектура;
- как производится анализ вариантов архитектуры;
- каким образом на основе архитектуры создаются системы, задающие для компании новую, более высокую планку по части возможностей и требований.

1.1. Откуда берутся варианты архитектуры?

Любая архитектура являет собой результат принятия ряда экономических и технических решений. Эти факторы влияния играют свою роль в процессе проектирования, а их конкретная реализация обусловливается средой, в которой архитектура должна работать. Архитектор, которому для проектирования системы отводятся сжатые сроки в реальном времени, принимает одни проектные решения; тот же архитектор, не стесненный временными ограничениями, принимает уже другие решения. Еще один ряд решений принимается в ходе разработки системы вне реального времени. Даже если архитектору предъявляют те же требования, предоставляют то же оборудование, сопроводительное программное обеспечение и тот же штат, что и пять лет назад, сегодня он, скорее всего, будет принимать новые решения.

В форме требований формулируются некоторые, но далеко не все, желаемые свойства конечной системы. С другой стороны, не все требования напрямую затрагивают эти свойства; они, к примеру, могут регламентировать процесс разработки и применение тех или иных инструментальных средств. Спецификация требований — это только начало. Если не обращать внимания на другие ограничения, система может оказаться не менее скверной, чем если бы она плохо работала.

Анализ архитектурно-экономического цикла мы начнем с выявления различных факторов влияния — как тех, что воздействуют на архитектуру, так и тех, посредством которых она воздействует на среду.

Влияние на архитектуру оказывают заинтересованные в системе лица

В создании программной системы заинтересованы многие люди и организации. Эти *заинтересованные лица* (stakeholders) включают, среди прочих, заказчика, конечных пользователей, разработчиков, руководителя проекта, специалистов по сопровождению и маркетингу. У разных заинтересованных лиц есть свои представления о свойствах системы — в частности, они высказывают пожелания относительно поведения системы при прогоне, производительности на тех или иных аппаратных средствах, настраиваемости, быстрого выхода на рынок и низких за-

трат на разработку, высокой заработной платы программистов соответствующей специальности, широкой функциональности и т. д. На рис. 1.2 изображен архитектор, получающий от заинтересованных лиц «полезные советы».

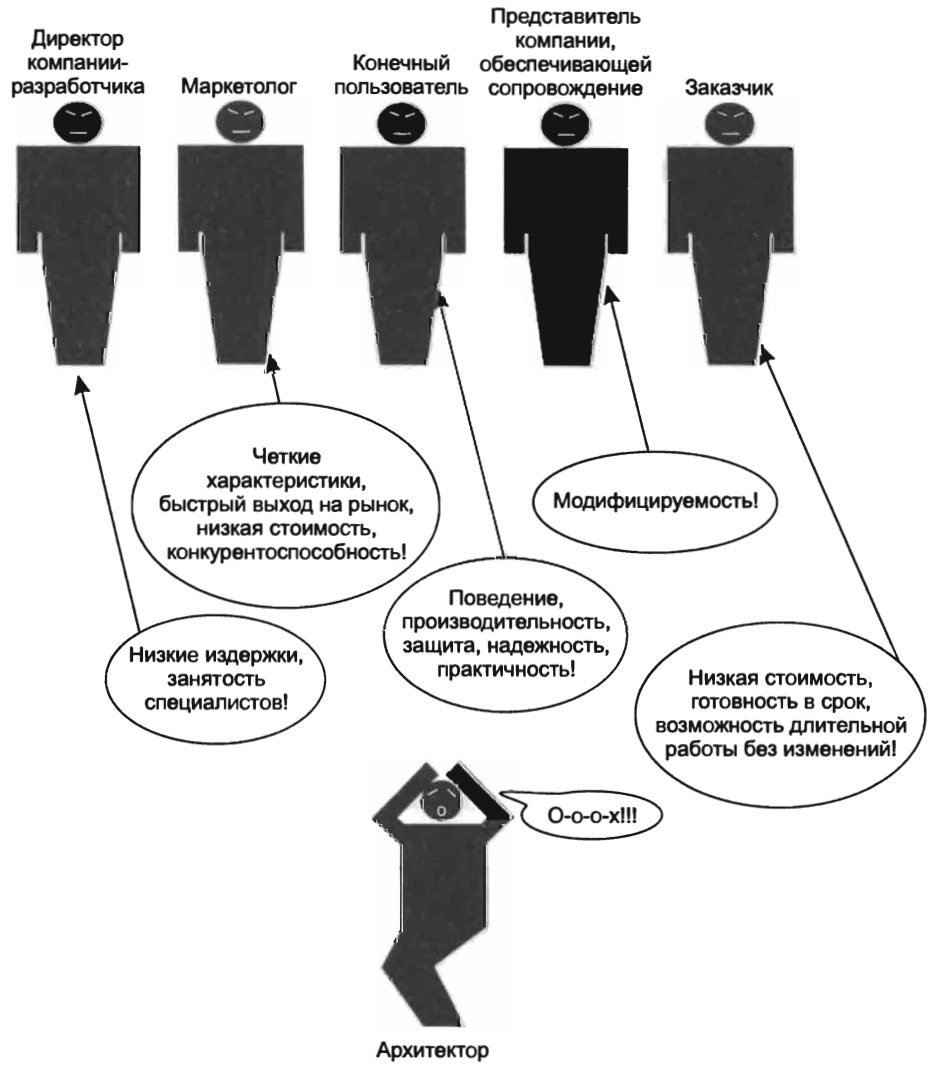


Рис. 1.2. Архитектор под воздействием заинтересованных лиц

Желательно, чтобы у системы были такие свойства, как высокая производительность, надежность, готовность, совместимость с различными платформами, умеренное потребление памяти, возможность использования в сети, безопасность, модифицируемость, практичность, способность взаимодействия с другими системами и приемлемое поведение. Именно из этих свойств, в чем мы сможем убедиться, складывается архитектура. От них зависит, понравится ли система заинтересованным лицам — тем, для кого она предназначена.

Трудность в том, что у каждого заинтересованного лица свои представления и задачи, причем иногда они противоречат друг другу. Желаемые свойства системы, конечно, можно изложить в таком артефакте, как сводка требований. Впрочем, подобного рода документы, в которых все требования к качеству сформулированы достаточно подробно, встречаются крайне редко. Как правило, архитектору приходится заполнять бреши и улаживать противоречия.

Влияние на архитектуру оказывает компания-разработчик

Воздействие компании-разработчика на архитектуру также не исчерпывается изложенными требованиями; следует также учитывать структуру и характер организации. К примеру, если в компании избыток программистов, специализирующихся на клиент-серверных технологиях, значит, вероятнее всего, менеджмент такой компании будет настаивать на разработке клиент-серверной архитектуры. В противном случае такую архитектуру могут и забраковать. Итак, одним из дополнительных факторов влияния является специализация сотрудников; кроме того, есть график работы над проектами и бюджет.

Факторы влияния, действующие в рамках компании-разработчика, делятся на три группы: краткосрочные экономические, долгосрочные экономические и организационные.

- Компании делают прямые инвестиции в различные активы — в частности, в существующие варианты архитектуры и основанные на них продукты. Каждый последующий проект в таком случае мыслится как продолжение ряда схожих систем, а в его смете заложено активное повторное использование имеющихся средств.
- Следуя своим стратегическим задачам, компании иногда делают долгосрочные инвестиции в инфраструктуру; в таком случае предполагаемая система мыслится как одно из средств финансирования и расширения этой инфраструктуры.
- Определенное влияние на программную архитектуру оказывает организационная структура компании-разработчика. В главе 8 приводится конкретный пример («Моделирование условий полета. Конкретный пример архитектуры, ориентированной на интегрируемость»); он демонстрирует, что некоторые подсистемы могут производиться компаниями-субподрядчиками, обладающими штатом из специалистов в данной области. Специализация разработки подсистем стала возможной благодаря реализованному в архитектуре разделению функций.

Влияние на архитектуру оказывают опыт и привычки архитекторов

Если у архитектора системы есть положительный опыт применения того или иного архитектурного решения (например, распределенных объектов или неявный вы-

зов методов), то, скорее всего, он будет пользоваться им в последующей работе. С другой стороны, если предыдущие попытки применения определенного решения закончились ничем, то архитектор вряд ли захочет задействовать его вновь. Решения в арсенале архитектора появляются по мере повышения образовательного уровня и накопления опыта, использования удачных архитектурных образцов (patterns) и обращения с системами, работавшими особенно плохо или особенно хорошо. Кроме того, архитекторы имеют обыкновение экспериментировать с новыми образцами и методиками, о которых они узнают из книг (подобных этой) и учебных курсов.

Влияние на архитектуру оказывает техническая база

Подготовка и опыт архитектора, в частности, проявляются в его работе с *технической базой* (technical environment). Определенное воздействие на архитектуру оказывают существующие в настоящее время технические средства. Речь здесь может идти как о методах работы, принятых в данной отрасли, так и о приемах программной инженерии, распространенных в профессиональном сообществе, в которое входит сам архитектор. В сегодняшних условиях нужно быть просто бесстрашным, чтобы при разработке архитектуры информационной системы напрочь отказаться от веб-технологий, объектов и решений, ориентированных на промежуточное (связующее, посредническое) программное обеспечение.

Вариативность факторов влияния на архитектуру

Источники влияния на архитектуру крайне многочисленны. Некоторые из них лишь подразумеваются, другие же находятся друг с другом в явном противоречии.

Ситуации, когда свойства системы, подразумеваемые экономическими и организационными задачами, выводятся на уровень сознания, а тем более — четко формулируются, — крайне редки. Даже требования заказчика в полной мере документируются отнюдь не всегда, а значит, неизбежный конфликт между задачами различных заинтересованных лиц разрешить не удается.

И тем не менее любой архитектор должен как можно раньше узнать и осознать характер, источники и степень важности различных накладываемых на проект ограничений. Следовательно, в его задачу входит *выявление заинтересованных лиц и их деятельное привлечение к фиксации требований и пожеланий.* В противном случае в какой-то момент представители заинтересованных лиц потребуют, чтобы архитектор объяснил, почему все предложенные ими варианты архитектуры неприемлемы; в результате срок сдачи проекта оттянется, а трудовые ресурсы будут простаивать. Привлекая заинтересованные лица к обсуждению проекта как можно раньше, архитектор уясняет ограничения задачи, классифицирует пожелания, договаривается о приоритетах и находит компромиссы. Рассматриваемые в части 3 обзоры архитектуры, наряду с итерационным макетированием, полностью обеспечивают достижение этой цели.

Совершенно очевидно, что владения архитектором технической стороной процесса недостаточно. То и дело тому или иному заинтересованному лицу придется объяснять, почему приоритеты реализации различных свойств расставлены именно так, а не иначе, и почему не все их пожелания можно будет воплотить в жизнь. Таким образом, хороший архитектор должен быть дипломатичным, уметь вести переговоры и улаживать разногласия.

Факторы влияния на архитектора, а через него и на архитектуру, показаны на рис. 1.3. Среди них — требования к продукту в представлении заинтересованных лиц, структура и задачи компании-разработчика, доступные технические средства, собственный образовательный уровень и опыт.

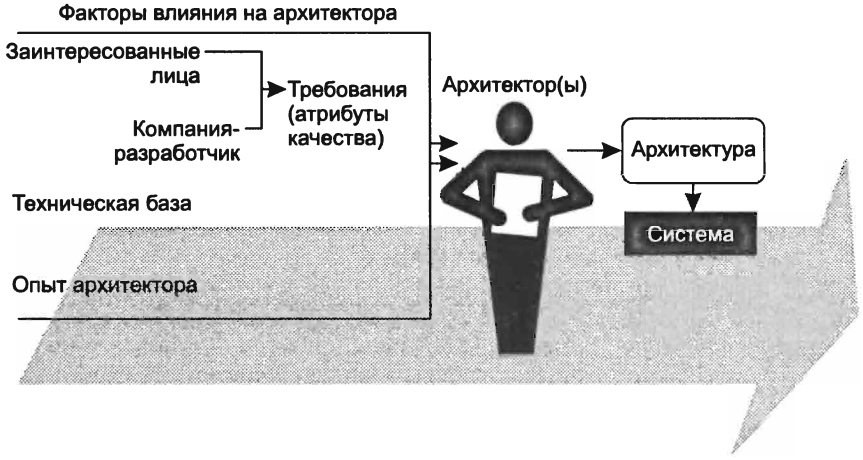


Рис. 1.3. Факторы влияния на архитектуру

Архитектура оказывает обратное воздействие на факторы влияния

Основная мысль, которую мы намерены донести в этой книге, заключается в том, что из взаимоотношений между производственными задачами, требованиями к продукту, опытом архитектора, архитектурами и созданными системами образуется цикл с цепями обратной связи; организация прохождения этого цикла входит в число задач компании. Успешно справляясь с ней, компания обеспечивает для себя возможность развития, расширения сферы коммерческой деятельности, а также реализации инвестиций в архитектуру и конструирование систем. Упомянутые цепи обратной связи изображены на рис. 1.4. Частично обратная связь поступает от самой архитектуры, частично — от построенной на ее основе системы.

Цикл этот выглядит следующим образом.

1. Архитектура влияет на структуру компании-разработчика. Архитектура обусловливает структуру системы; в частности (в этом мы сможем убедиться), она устанавливает набор блоков программного обеспечения, которое надлежит реализовать (или обеспечить их наличие другим путем), а затем

интегрировать в рамках системы. Эти блоки составляют основу разработки структуры проекта. Группы разработчиков укомплектовываются именно по блокам; операции в рамках процессов разработки, тестирования и интеграции также выполняются в отношении блоков. Согласно графикам и бюджетам, ресурсы выделяются частями в расчете на отдельные блоки. Если компания наработала опыт конструирования семейств сходных систем, она будет вкладывать средства в повышение профессионального уровня участников сформированных по блокам групп разработчиков. Следовательно, группы встраиваются в структуру организации. Такой представляется обратная связь от архитектуры к компании-разработчику.

Если мы обратимся к конкретному примеру линейки программных продуктов, рассматриваемому в главе 15, то увидим, что задачи по конструированию и сопровождению отдельных частей архитектуры семейства продуктов были делегированы отдельным группам разработчиков. За какой бы проект ни принималась компания, все эти группы исполняют существенную роль в декомпозиции систем, а попутно обеспечивают дальнейшее существование контролируемых ими блоков.

2. Архитектура способна оказывать воздействие на задачи компании-разработчика. Сконструированная на ее основе успешная система предоставляет компании возможность укрепиться в данном сегменте рынка. Такая архитектура предусматривает дальнейшее эффективное производство и размещение сходных систем, вследствие чего компания может откорректировать свои задачи и, воспользовавшись новым преимуществом, занять рыночную нишу. Так выглядит обратная связь от системы к компании-разработчику и конструируемым ею системам.

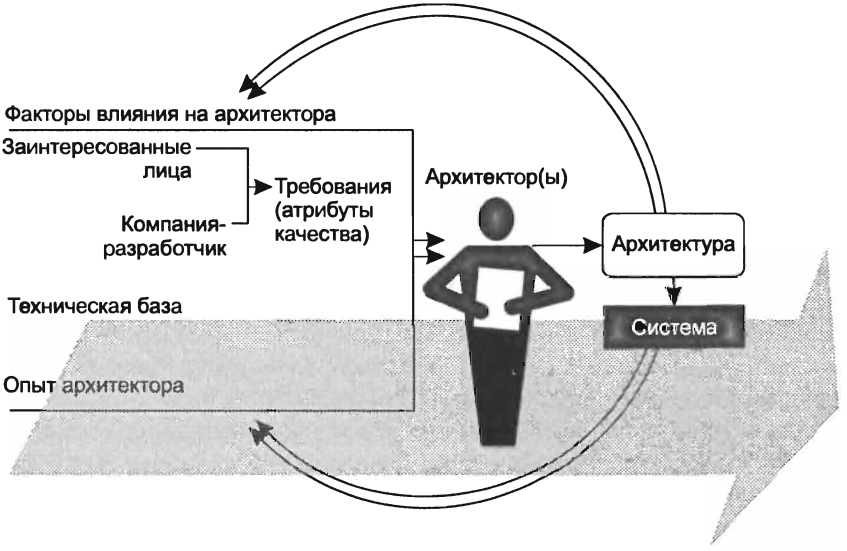


Рис. 1.4. Архитектурно-экономический цикл

3. Архитектура может оказывать воздействие на требования, выдвигаемые заказчиком относительно следующей системы, — ее (если она основана на той же архитектуре, что и предыдущая) он может получить в более надежном варианте, быстрее и экономичнее, чем в том случае, если бы она конструировалась «с чистого листа». Возможно, заказчик откажется от некоторых требований в пользу повышения экономичности. Готовые программные продукты несколько изменили требования, предъявляемые заказчиками, — не предназначенные для удовлетворения индивидуальных потребностей, они недороги и (как бы сказать...) отличаются высоким качеством. На заказчиков, не слишком гибких по части своих требований, аналогичное воздействие оказывают линейки продуктов. На материале главы 15 («CelsiusTech. Конкретный пример разработки линейки продуктов») мы намерены показать, каким образом архитектура линеек продуктов заставляет заказчиков без сожаления снимать ранее предъявленные требования, — дело в том, что взамен они быстро, надежно и задешево получают высококачественное программное обеспечение, удовлетворяющее их основные потребности.
4. Процесс конструирования систем пополняет опыт архитектора, который он может применить при работе над последующими архитектурами, и, соответственно, расширяет базу опыта компании. Успех системы, построенной на основе инструментальных магистралей, .NET или инкапсулированных конечных автоматов, стимулирует построение аналогичных систем в дальнейшем. С другой стороны, неудачные варианты архитектуры редко используются повторно.
5. Иногда оказать сильное воздействие, и даже внести изменения в культуру программной инженерии (техническую базу, в рамках которой обучаются и работают конструкторы), способны отдельные системы. Такой эффект на индустрию в 1960-х и начале 1970-х годов оказали первые реляционные базы данных, генераторы компиляторов и табличные операционные системы; в 1980-х — первые электронные таблицы и системы управления окнами. В 1990-х годах в качестве такого рода катализатора выступила Всемирная паутина — с ней, между прочим, связан конкретный пример, приведенный в главе 13. В главе 16 мы предполагаем, что в первом десятилетии XXI века аналогичный эффект окажет J2EE. Подобные инновации всегда находят отражение в последующих системах.

Из этих и некоторых других механизмов обратной связи и образуется архитектурно-экономический цикл, схема которого приводится на рис. 1.4. На этой иллюстрации изображены факторы влияния культуры и экономики компании-разработчика на программную архитектуру. В свою очередь архитектура оказывается основным определяющим фактором при задании свойств разрабатываемой системы или систем. В ABC отражено также и то обстоятельство, что любая компания (если, конечно, у нее хороший менеджмент) способна обратить организационные влияния и практические результаты разработки архитектуры в свою пользу и с их помощью выработать стратегию работы над последующими проектами.

1.2. Программный процесс и архитектурно-экономический цикл

Программным процессом (software process) называются действия по организации, нормированию и управлению разработкой программного обеспечения. Какие операции направлены на создание программной архитектуры, ее применение для реализации проектного решения, а впоследствии — на реализацию или управление развитием целевой системы или приложения? Вот их перечень:

- создание экономической модели системы;
- выявление требований;
- создание новой или выбор существующей архитектуры;
- документирование и распространение сведений об архитектуре;
- анализ или оценка архитектуры;
- реализация системы на основе архитектуры;
- проверка соответствия реализации архитектуре.

Этапы разработки архитектуры

Как следует из структуры ABC, между различными этапами разработки архитектуры существуют развернутые отношения обратной связи. Несколько нижеследующих подразделов отведены под краткий обзор этих этапов.

Создание экономической модели системы

Создание экономической модели не ограничивается оценкой потребности в системе на рынке. Этот этап исполняет существенную роль в контексте создания и сужения требований. Сколько должен стоить продукт? Каков целевой сегмент рынка? Насколько быстро продукт должен выйти на рынок? Должен ли он взаимодействовать с другими системами? Есть ли какие-нибудь системные ограничения, в рамках которых он должен существовать?

Все эти вопросы решаются с привлечением архитектора. Одного его, конечно, недостаточно, однако если при создании экономической модели консультации с архитектором не проводились, то возможность достижения коммерческих задач становится проблематичной.

Выявление требований

Способов узнать, чего же, наконец, хотят заинтересованные лица, множество. В частности, в рамках объектно-ориентированного анализа для фиксации требований используются сценарии, или элементы Use Case. Для системы с повышенными требованиями к безопасности применяются более строгие методики — например, модели конечных автоматов и языки формальных спецификаций. В главе 4 («Атрибуты качества») мы разберем ряд сценариев атрибутов качества, обеспечивающих фиксацию требований к качеству системы.

Применительно к конструируемой системе необходимо принять центральное, основополагающее решение — насколько в ней будут отражены другие, уже сконструированные системы. Поскольку в сегодняшних условиях найти систему, не имеющую сходств с другими системами, весьма непросто, методики выявления требований предполагают знание характеристик предшествующих систем. Архитектурное содержание линеек продуктов разбирается в главе 14 («Линейки продуктов. Повторное использование архитектурных средств»).

Еще один способ выявления требований подразумевает моделирование. Опытные системы помогают моделировать нужное поведение, проектировать пользовательские интерфейсы и проводить анализ потребления ресурсов. Таким образом, в глазах заинтересованных лиц система становится «реальной», а процесс принятия решений по проектированию системы и ее пользовательского интерфейса значительно ускоряется.

Вне зависимости от методики выявления требований желаемые атрибуты качества конструируемой системы обусловливают ее конечный вид. Для обеспечения отдельных атрибутов качества архитекторами уже давно применяются те или иные тактики. Многие из них рассматриваются в главе 5 («Реализация качества»). Архитектурные решения компромиссны, однако при специфицировании требований не все эти компромиссы очевидны. Со всей ясностью они проявляются только после создания архитектуры; тогда же принимаются решения относительно сортировки требований в соответствии с приоритетами.

Создание или выбор архитектуры

Фредерик Брукс (Fred Brooks) в своей знаменитой книге «Мифический человеко-месяц» красноречиво и убедительно доказывает, что основным условием стабильного проектирования системы является соблюдение концептуальной целостности, а она может проявиться лишь в узком кругу людей, совместно работающих над проектированием ее архитектуры. Принципы реализации в ходе создания архитектуры требований по поведению и качеству демонстрируются в главах 5 («Реализация качества») и 7 («Создание архитектуры»).

Распространение сведений об архитектуре

Для того чтобы архитектура действительно стала основой проекта, ее суть необходимо четко и недвусмысленно донести до всех заинтересованных лиц. Разработчики должны понимать, что от них требуется, тестировщики должны осознавать структуру своих задач, менеджмент должен знать график и т. д. Для того чтобы этой цели можно было добиться, документирование архитектуры должно быть информативным, ясным и понятным людям различных профессий. Документирование архитектуры рассматривается в главе 9 («Документирование программной архитектуры»).

Анализ или оценка архитектуры

В процессе проектирования всегда рассматривается множество вариантов проекта. Некоторые из них забраковываются сразу. Из числа остальных в конечном итоге отбирается наиболее подходящий. Одна из глобальных задач, стоящих пе-

ред любым архитектором, заключается именно в том, чтобы сделать этот выбор рационально. Методы принятия решений на этом этапе рассматриваются в ряде глав части 3 («Анализ архитектуры»).

Оценить архитектуру на предмет атрибутов качества, которые она обеспечивает, совершенно необходимо — без этого нельзя быть уверенным в том, что конечная система сможет удовлетворить все потребности заинтересованных лиц. Все большее распространение получают методики анализа, ориентированные на оценку сообщаемых системе архитектурой атрибутов качества. Сценарные методики обеспечивают наиболее универсальную и эффективную оценку архитектуры. Самая зрелая методическая база характерна для метода анализа компромиссных архитектурных решений (Architecture Tradeoff Analysis Method, ATAM), рассматриваемого в главе 11; метод анализа стоимости и эффективности (Cost Benefit Analysis Method, CBAM, см. главу 12), с другой стороны, предусматривает крайне ценную возможность увязки архитектурных решений с их экономическим содержанием.

Реализация на основе архитектуры

Этот процесс предусматривает согласованность действий разработчиков со структурами и протоколами взаимодействия, заданными архитектурой. Соответствие положениям архитектуры в первую очередь обеспечивается четкой и в полной мере озвученной документацией. Дополнительным преимуществом в контексте этой задачи будет среда, или инфраструктура, активно содействующая (в отличие от простого кода) созданию и сопровождению архитектуры.

Проверка соответствия архитектуре

Когда архитектура составлена и задействована, наступает этап сопровождения. Для того чтобы обеспечить на этом этапе согласованность архитектуры и ее представления, нужно все время быть начеку. Область эта довольно молода, однако в последние годы в ней ведутся довольно интенсивные исследования. Современное состояние методик восстановления архитектуры исходя из существующей системы и проверки ее согласованности первоначальной архитектуре представлено в главе 10 («Реконструкция программной архитектуры»).

1.3. Из чего складывается «качественная» архитектура?

Если утверждение о том, что, располагая одними и теми же техническими требованиями, два архитектора в двух компаниях создадут разные архитектуры, верно, то как эти архитектуры оценивать? Другими словами, какая из них *правильная*?

Не существует такой архитектуры, которую можно было бы признать однозначно хорошей или однозначно плохой. Варианты архитектуры могут более или менее соответствовать поставленной задаче. Скажем, распределенная трехзвенная клиент-серверная архитектура идеально подходит для системы управления финансами в крупной компании, но в то же время совершенно не годится для

авиационных приложений. Архитектуру, вполне отвечающую требованию по высокой модифицируемости, не имеет смысла использовать для создания одноразовой опытной системы. Один из главных постулатов этой книги заключается в том, что варианты архитектуры можно оценивать, и это один из факторов, обосновывающих повышенное к ним внимание, однако проводить такую оценку можно только в контексте конкретных задач.

Тем не менее в ходе проектирования архитектуры следует придерживаться ряда практических правил. Несоблюдение какого-то одного из них не означает, что архитектуру следует полностью забраковать; просто в причинах этого несоблюдения неплохо бы разобраться.

Все наши наблюдения делятся на две части: рекомендации по процессу и рекомендации по продукту (они же структурные рекомендации). Рекомендации по процессу таковы.

- Архитектура должна разрабатываться одним архитектором или небольшой командой архитекторов с очевидным руководителем.
- Архитектор (или команда архитекторов) должен знать функциональные требования к системе и располагать четким, отсортированным согласно приоритетам перечнем атрибутов качества (примерами которых могут быть безопасность и модифицируемость), которым предполагаемая архитектура должна соответствовать.
- Архитектура должна быть в полной мере документирована, причем в документации следует отразить как минимум по одному статическому и динамическому представлению (о том, что это такое, мы поговорим в главе 2), использующему согласованную, понятную всем заинтересованным лицам нотацию.
- Содержание архитектуры необходимо раскрыть для всех заинтересованных в системе лиц; последние должны принимать активное участие в ее обсуждении.
- Архитектуру следует проанализировать на предмет значимых количественных характеристик (например, максимальной производительности) и формально оценить на предмет атрибутов качества, причем сделать это нужно на том этапе, когда вносить поправки еще не поздно.
- Архитектура должна предусматривать возможность инкрементной (пошаговой) реализации; для этого следует создать «макет» (skeletal) системы и при минимальной функциональности испытать на нем все каналы связи. Макет системы можно использовать для инкрементного «наращивания» системы, тем самым уменьшив затраты на интеграцию и тестирование (см. главу 7, раздел 7.4).
- Все области состязаний за ресурсы в конечной архитектуре должны быть, во-первых, немногочисленными, а во-вторых, ясно определенными; порядок разрешения этих конфликтов необходимо четко обозначить, информацию о нем распространить и впоследствии поддерживать. В частности, если трудности возникают в связи с уровнем использования сети, архитектор должен составить (и провести в жизнь) нормативы, согласно которым все

группы разработчиков должны будут соблюдать минимальный уровень сетевого трафика. Если же требуется обеспечить высокую производительность, архитектор должен определить (и провести в жизнь) временные ресурсы для основных потоков.

Что касается структурных правил, то их мы представляем следующим образом.

- Архитектура должна состоять из строго очерченных модулей, а функциональные обязанности между ними должны распределяться согласно принципам сокрытия информации и разделения задач. На основе информационной закрытости должны строиться (прежде всего) те модули, в которых инкапсулируется гиперчувствительность вычислительной инфраструктуры, — при внесении изменений в инфраструктуру они уберегают от корректировки бóльшую часть программного обеспечения.
- Каждый модуль должен иметь четкий интерфейс, инкапсулирующий или «скрывающий» изменяемые аспекты (в частности, стратегии реализации и варианты структур данных) от стороннего, обращающегося к его средствам программного обеспечения. Наличие подобных интерфейсов позволяет добиться определенной независимости в действиях различных групп разработчиков.
- Атрибуты качества должны реализовываться с привлечением хорошо изученных архитектурных тактик и на индивидуальной основе (см. главу 5 «Реализация качества»).
- Зависимость архитектуры от конкретной версии коммерческого изделия или инструмента недопустима. Если она все же имеет место, архитектуру необходимо структурировать таким образом, чтобы адаптацию к другим продуктам можно было провести без существенных трудностей и издержек.
- Модули, производящие информацию, следует отделять от модулей, потребляющих информацию. Поскольку изменения во многих случаях ограничиваются исключительно производством или исключительно потреблением данных, такое разделение способствует повышению модифицируемости. Введение новых данных предполагает корректировку обеих сторон, обновление которых, в случае разделения, производится поэтапно.
- В состав архитектуры систем с параллельной обработкой должны входить четко заданные процессы или задачи, которые могут соответствовать, а могут и не соответствовать структуре декомпозиции на модули. Иначе говоря, любой такой процесс может распространяться на несколько модулей; с другой стороны, процедуры в некоторых модулях могут вызываться как элементы нескольких процессов (этот принцип реализован в системе A-7E — см. конкретный пример в главе 3).
- Все задачи и процессы следует писать так, чтобы обеспечить удобство их переназначения на другой процессор (даже в период прогона).
- В состав архитектуры должен входить ряд элементарных образцов взаимодействия (см. главу 5). Это нужно для того, чтобы система все время

выполняла свои задачи единообразно. В результате произойдет улучшение таких характеристик, как понятность, продолжительность разработки, надежность и модифицируемость. Кроме того, это придаст архитектуре концептуальную целостность — она, хоть и не поддается измерению, существенно облегчает разработку.

По мере того как вы будете знакомиться с представленными в этой книге конкретными примерами успешного решения той или иной архитектурной задачи, вспоминайте эти правила и проверяйте, насколько они соблюдались в каждом случае. Представленный набор правил не претендует ни на полноту, ни на безусловность; тем не менее он послужит ориентиром для любого архитектора, приступающего к решению архитектурно-проектной задачи.

1.4. Заключение

На материале настоящей главы мы пытались доказать, что факторы влияния на архитектуру отнюдь не исчерпываются функциональными требованиями к системе. Любая архитектура — это в равной степени результат применения навыков архитектора, технической базы, которой он располагает, и коммерческих задач компании, в которой он работает. В свою очередь архитектура, пополняя собой техническую базу и открывая новые возможности продвижения на рынки, оказывает обратное влияние на среду своего «обитания». Мы определили архитектурно-экономический цикл и представили его как лейтмотив книги, однако не стоит забывать, что в следующих главах это понятие будет раскрываться.

Наконец, мы изложили ряд практических правил, соблюдение которых обычно обеспечивает создание удачных вариантов архитектуры.

Далее мы углубимся в рассмотрение программной архитектуры как таковой.

1.5. Дискуссионные вопросы

1. В какой степени особенности вашей компании влияют на характер разрабатываемых в ней вариантов архитектур? Существует ли обратное влияние?
2. Какие коммерческие задачи вашей компании обусловливают (или обусловливали) создание вариантов программной архитектуры?
3. Какие заинтересованные лица в вашей компании в наибольшей степени влияют на содержание архитектуры систем? Каковы их задачи? Не бывает ли так, что эти задачи противоречат друг другу?

Глава 2
Что такое «программная архитектура»?

(в соавторстве с Линдой Нортроп[1])

> В отсутствие архитектуры системы и ее логического обоснования начинать масштабные работы по разработке проекта не следует. Спецификация архитектуры как поставляемого продукта обеспечивает ее применимость на любых этапах процессов разработки и сопровождения.
>
> *Барри Боэм [Boehm 95]*

В главе 1 мы говорили о том, насколько важна архитектура в контексте выполнения коммерческих задач компании. Обстоятельная разработка архитектуры обходится недешево, однако, позволяя компании решать системные задачи и расширять программные возможности, она обязательно окупается. Архитектура как один из активов компании-разработчика сохраняет значимость в течение длительного времени и переживает проект, для которого создавалась.

В данной главе мы рассмотрим архитектуру с точки зрения программной инженерии. Иначе говоря, мы проанализируем все преимущества наличия у проекта программной архитектуры, помимо представленных в главе 1.

2.1. Чем является программная архитектура и чем она не является

Рисунок 2.1 — иллюстрация к описанию системы гидроакустического моделирования — претендует на изображение «архитектуры высшего уровня»; как правило,

---

[1] Линда Нортроп (Linda Northrop) работает в Институте программной инженерии Университета Карнеги-Меллон руководителем программ.

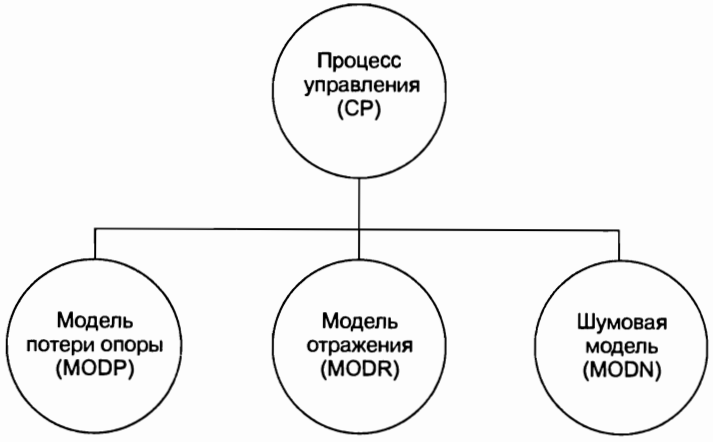


Рис. 2.1. Типичное, но неинформативное представление программной архитектуры

архитектура изображается именно на таких диаграммах. Какие выводы мы можем из нее сделать?

- Система состоит из четырех элементов.
- Поскольку три из четырех элементов — модель потери опоры (MODP), модель отражения (MODR) и шумовая модель (MODN) — расположены рядом друг с другом, между ними, вероятно, больше сходства, чем между каждым из них и четвертым элементом — процессом управления (CP).
- Поскольку данная диаграмма является вполне связной, между всеми ее элементами, по-видимому, присутствует некая взаимосвязь.

Можем ли мы заключить, что на рис. 2.1 приводится диаграмма архитектуры? Она, очевидно, соответствует весьма распространенному определению архитектуры как совокупности компонентов (в данном случае их четыре) и связей между ними (они налицо). Подойдем к вопросу с другой стороны. Предположив, что это наиболее элементарное определение верно, разберемся с тем, что представленная диаграмма *не в состоянии* нам сообщить.

- *Каков характер элементов?* В чем смысл их разделения? Может быть, они обрабатываются разными процессорами? Или запускаются в разные периоды времени? Из чего состоят эти элементы: из процессов, программ или из того и другого? Возможно, диаграмма демонстрирует схему разделения обязанностей между группами разработчиков проекта, или же она все-таки подразумевает разделение в период прогона? Являются ли представленные элементы объектами, задачами, функциями, процессами, распределенными программами или чем-то еще?
- *Каковы обязанности этих элементов?* Зачем они нужны? Какие функции они исполняют в рамках своей системы?
- *В чем значение связей между элементами?* Означают ли они, что элементы взаимодействуют друг с другом, управляют друг другом, отсылают друг другу какие-то данные, используют, запускают или синхронизируют друг друга, располагают некоей скрытой (от пользователя) информацией; воз-

можно, отношения между элементами строятся на основе сразу нескольких подобного рода связей? Каковы механизмы взаимодействия между элементами? Что за информация передается посредством этих механизмов?

- *Чем объясняется такое расположение элементов?* С какой стати процесс управления выведен на отдельный уровень? Может быть, он может вызывать все остальные элементы, а они его — нет? Возможно, он как блок реализации содержит три нижележащих элемента? Или все значительно проще — все четыре элемента не уместились на одной строке?

Задаваться этими вопросами *необходимо* — не зная, что представляют собой элементы и как путем их взаимодействия реализуются задачи системы, мы вряд ли сможем почерпнуть из такого рода диаграмм много полезной информации, а значит, относиться к ним следует скептически.

Итак, программная архитектура на представленной диаграмме не отражена — по крайней мере, ничего путного из нее извлечь мы не можем. Мягко говоря, подобные диаграммы — это только начало. Что же на самом деле заключает в себе понятие «программная архитектура»?

> Программная архитектура программы или вычислительной системы — это структура ее структур, то есть изложение ее программных элементов, их внешних свойств и установленных между ними отношений[1].

Внешними (externally visible) свойствами называются те предположения, которые сторонние элементы могут выдвигать в отношении данного элемента, — в частности, они касаются предоставляемых элементом услуг, рабочих характеристик, устранения неисправностей, совместного использования ресурсов и т. д. Проанализируем представленное определение более подробно.

Во-первых, *архитектура определяет программные элементы*. В составе архитектуры приводится информация о взаимоотношениях элементов. Собственно говоря, все, что архитектура сообщает нам об элементах, ограничивается информацией об их взаимодействии. Итак, архитектура — это, в первую очередь, абстракция системы, в которой отсутствует информация об элементах, не имеющая отношения к тому, как они используют, используются, соотносятся или взаимодействуют с другими элементами. Практически во всех современных системах взаимодействие между элементами осуществляется посредством интерфейсов, которые поддерживают деление информации об элементах на публичную и приватную части. Так вот, архитектура имеет дело только с публичной частью; приватные детали элементов — те, что относятся исключительно к их внутренней реализации, — в состав архитектуры не входят.

Во-вторых, из вышеприведенного определения ясно, что *в состав любой системы может входить и входит целый ряд структур*, а следовательно, ни одной отдельно взятой структуры, которую можно было бы уверенно назвать архитектурой, не существует. В частности, все нетривиальные проекты делятся на блоки

---

[1] В первом издании это определение было сформулировано несколько по-другому. Первичные стандартные блоки в нем назывались «компонентами». К настоящему моменту этот термин прочно ассоциируется с компонентной концепцией разработки программного обеспечения, а конкретнее — с периодом прогона. По этой причине, имея в виду большее обобщение, мы остановились на термине «элемент».

реализации; между этими блоками распределяются некоторые обязанности, и они же в большинстве случаев выступают в качестве базы для разделения задач между группами программистов. В таких элементах, наряду с программами и данными, доступными для вызова или обращения со стороны других программных средств и блоков реализации, содержатся приватные программы и данные. В крупных проектах эти элементы почти всегда разделяются на мелкие части, а обязанности по работе с ними распределяются между несколькими мелкими группами разработчиков. Довольно часто описания систем составляются при помощи таких структур. Ориентированные на разделение функций системы между различными группами исполнителей реализации, они отличаются чрезмерной статичностью.

Другие структуры в большей степени ориентируются на взаимодействие элементов в период прогона с целью исполнения функции системы. Представим, что систему предполагается сконструировать в виде ряда параллельных процессов. В этом случае часто применяется другая структура, включающая процессы периода прогона, программы из вышеописанных блоков реализации, совместно формирующие отдельные процессы, а также установленные между этими процессами отношения синхронизации.

Можем ли мы взять любую из этих структур в отдельности и назвать ее архитектурой? Нет, не можем — и это несмотря на то, что все они содержат архитектурные сведения. В состав любой архитектуры эти структуры входят наравне со многими другими. В этой связи очевидно, что, поскольку в составе архитектуры может содержаться несколько структур, в ней должны присутствовать несколько элементов (например, блок реализации и процессы), несколько вариантов взаимодействия между элементами (например, разбиение на составляющие и синхронизация) и даже несколько контекстов (например, время разработки и время прогона). Представленное определение не устанавливает сущность архитектурных элементов и отношений. Является ли программный элемент объектом? процессом? библиотекой? базой данных? коммерческим изделием? Он может быть чем угодно, причем возможности не ограничиваются вышеприведенными вариантами.

В-третьих, согласно нашему определению, *программная архитектура есть у любой вычислительной системы с программным обеспечением* — связано это с тем, что любую систему можно представить как совокупность ее элементов и установленных между ними отношений. В простейшем случае система сама по себе является элементом — не представляющим, вероятно, никакого интереса и бесполезным, однако, тем не менее, согласующимся с понятием «архитектура». То обстоятельство, что архитектура есть у каждой системы, совершенно не означает, что она является общеизвестной. Кто знает — быть может, специалистов, спроектировавших систему, уже не найти, документация тоже куда-то исчезла (или ее никогда не существовало), исходный код потерян (или не поставлялся), а остался лишь исполняемый двоичный код. Именно в этом случае различие между архитектурой системы и ее представлением становится очевидным. К сожалению, архитектура иногда существует самостоятельно, без описаний и спецификаций; в этой связи существенную важность приобретают *документирование* (architecture documentation, см. главу 9) и *реконструкция* (architecture reconstruction, см. главу 10) архитектуры.

В-четвертых, если поведение отдельно взятого элемента можно зафиксировать или выявить с точки зрения других элементов, то это *поведение входит в состав архитектуры*. Именно оно позволяет элементам взаимодействовать друг с другом, а взаимодействие, как известно, отражается в архитектуре в обязательном порядке. Этим, помимо прочего, объясняется, почему схемы из прямоугольников и линий, выдаваемые за архитектуры, таковыми не являются. Это не более чем рисунки; самое лучшее, что о них можно сказать, — это то, что они намекают на существование более четкой информации относительно фактических функций обозначенных элементов. Взглянув на наименования изображенных на подобной схеме прямоугольников (например, на них написано «база данных», «пользовательский интерфейс», «исполняемый файл» и т. д.), читатель хотя бы получит представление о функциональности и поведении соответствующих элементов. Сложившийся в голове читателя образ в чем-то напоминает архитектуру, однако, во-первых, он имеет ментальное происхождение, а во-вторых, отталкивается от отсутствующей на схеме информации. Мы совершенно не хотим сказать, что точное поведение и исполнение каждого элемента безусловно необходимо документировать; тем не менее в той степени, в которой поведение отдельного элемента влияет на характер взаимодействия с ним других элементов и на приемлемость системы в целом, это поведение входит в программную архитектуру.

Наконец, в представленном определении *не отражено качество архитектуры системы* — иначе говоря, никаких утверждений относительно перспектив соответствия системы установленным требованиям к поведению, производительности и жизненному циклу в ней нет. Поскольку метод проб и ошибок (предполагающий произвольный выбор архитектуры и последующее конструирование на ее основе системы под лозунгом «будь что будет») в качестве оптимального способа выбора архитектуры для системы нас совсем не устраивает, мы акцентируем внимание на вопросах *оценки архитектуры* (architecture evaluation, см. главы 11 и 12) и *архитектурного проектирования* (architecture design, см. главу 7).

2.2. Другие взгляды на архитектуру

Программная архитектура как дисциплина стремительно развивается, но все же она еще молода; поэтому какого-то универсального, повсеместно принятого ее определения не существует. Но и недостатка в вариантах определений не наблюдается. Большинство из них, как правило, сходятся в том, что любая архитектура состоит из структур, элементов и связей между ними; то, что они не взаимозаменяемы, объясняется расхождениями в деталях.

Программная архитектура изучается путем фиксации применяемых проектировщиками принципов конструирования и действий, которые они используют в процессе работы с реальными системами. Таким образом, предпринимается попытка выделить универсальные черты системного проектирования; в этом качестве программная архитектура имеет дело с самыми разными операциями, понятиями, методами, подходами и результатами. Именно поэтому в сообществе специалистов по программной инженерии распространены некоторые другие,

отличные от приведенного выше, определения программной архитектуры; поскольку какие-то из них вам наверняка встретятся, вы должны понимать, что они подразумевают, и знать, на основе каких аргументов в том или ином случае можно построить дискуссию. Отдельные, наиболее ходовые определения приводятся ниже.

- *Архитектура — это проект, поднятый на высокий уровень.* Действительно, лошадь относится к млекопитающим, однако они не равнозначны. В процессе проектирования решаются разные задачи, в том числе и неархитектурные — взять хоть вопрос об инкапсуляции значимых структур данных. Интерфейсы этих структур, без сомнения, относятся к архитектуре, но собственно их отбор — нет.
- *Архитектура — это общая структура системы.* Согласно распространенному (но неверному) мнению, у любой системы одна структура. Мы-то знаем, что это не так, а если кто-то вознамерится утверждать обратное, спросите его, какую конкретно структуру он имеет в виду. Эффект будет не только педагогическим. Как мы увидим впоследствии, наполнение системы атрибутами качества, которые в конечном итоге определяют ее успех или провал, происходит именно через разнообразные структуры. Множественность структур в рамках архитектуры составляет суть самого этого понятия.
- *Архитектура — это структура компонентов программы или системы, взаимосвязи, а также принципы и нормы их проектирования и развития во времени.* Это одно из ряда процессно-ориентированных определений, включающих дополнительные сведения о принципах и нормах. Многие считают, что в состав архитектуры входят декларация потребностей заинтересованных лиц и логическое обоснование, регламентирующее реализацию их требований. Действительно, сбор такой информации важен и необходим с точки зрения профессионализма. Однако мы не считаем, что эти документы являются частью архитектуры, — никто ведь не берется утверждать, что руководство пользователя автомобиля входит в состав автомобиля. Архитектуру, к какой бы системе она ни относилась, можно исследовать и проанализировать, не располагая знаниями о процессах ее проектирования и развития.
- *Архитектура — это компоненты и соединители.* Под соединителями имеется в виду механизм периода прогона, предназначенный для передачи в рамках системы сигналов управления и данных. Следовательно, данное определение ориентировано на архитектурные структуры периода прогона. К примеру, соединителем является UNIX-канал. Согласно этой логике, все архитектурные структуры, которые не относятся к периоду прогона (например, рассмотренное выше статическое разделение на ответственные блоки реализации), признаются второстепенными. В то же время в контексте решения системных задач они не менее важны, чем структуры периода прогона. Рассуждая об «отношениях» между элементами, мы имеем в виду все отношения — те, что реализуются в период прогона, и те, что к нему не относятся.

Все дискуссии по поводу программной архитектуры крутятся вокруг *структурности* систем. Иногда словом «архитектура» обозначают конкретный архитектурный образец (например, клиент–сервер), в других случаях — область исследований (например, книгу об архитектуре), однако в большинстве случаев под «архитектурой» имеют в виду структурные аспекты конкретной системы. Именно их мы пытались отразить в своем варианте определения.

2.3. Архитектурные образцы, эталонные модели и эталонные варианты архитектуры

В промежутке между схемами из прямоугольников и линий — простейшими «набросками» архитектур — и комплексными архитектурами, укомплектованными всей необходимой информацией о системах, существуют многочисленные переходные этапы. Каждый такой этап есть результат принятия ряда архитектурных решений, совокупность архитектурных альтернатив. Некоторые из них сами по себе имеют определенную ценность. Прежде чем переходить к анализу архитектурных структур, мы рассмотрим три промежуточных этапа.

1. *Архитектурный образец — это описание типов элементов и отношений и изложение ряда ограничений на их использование.* Образец имеет смысл рассматривать как совокупность ограничений, накладываемых на архитектуру, — конкретнее, на типы элементов и образцы их взаимодействия; на основе этих ограничений складывается ряд или семейство соответствующих им вариантов архитектуры. К примеру, одним из общеупотребительных архитектурных образцов является «клиент–сервер». Клиент и сервер — это два типа элементов; их взаимодействие описывается посредством протокола, при помощи которого сервер взаимодействует со всеми своими клиентами. Термин «клиент–сервер» по смыслу лишь предполагает множественность клиентов; конкретные клиенты не перечисляются, и речи о том, какая функциональность, помимо реализации протоколов, характерна для клиентов или для сервера, не идет. Согласно этому (неформальному) определению, образцу «клиент–сервер» соответствует бесчисленное количество различных вариантов архитектуры, причем все они чем-то отличаются друг от друга. Несмотря на то что архитектурный образец не является архитектурой, он все же содержит весьма полезный образ системы — он накладывает на архитектуру, а следовательно, и на систему, полезные ограничения.

   У образцов есть один крайне полезный аспект — дело в том, что они демонстрируют известные атрибуты качества. Именно поэтому архитекторы выбирают образцы не наугад, а исходя из определенных соображений. Некоторые образцы содержат известные решения проблем, связанных с производительностью, другие предназначаются для систем с высокими требованиями к безопасности, третьи успешно реализуются в системах с высокой готовностью. Во многих случаях выбор архитектурного образца оказывается первым существенным решением архитектора.

Синонимичным «архитектурному образцу» является общеупотребительный термин *архитектурный стиль* (architectural style).

2. *Эталонная модель — это разделение между отдельными блоками функциональных возможностей и потоков данных.* Эталонной моделью называется стандартная декомпозиция известной проблемы на части, которые, взаимодействуя, способны ее разрешить. Поскольку эталонные модели имеют происхождение в опыте, их наличие характерно только для сформировавшихся предметных областей. Вы можете назвать стандартные элементы компилятора или системы управления базами данных? А в общих словах объяснить, как эти элементы сообща решают свою общую задачу? Если можете, значит, вы знакомы с эталонной моделью этих приложений.
3. *Эталонная архитектура — это эталонная модель, отображенная на программные элементы (которые сообща реализуют функциональность, определенную в эталонной модели), и потоки данных между ними.* В то время как эталонная модель обеспечивает разделение функций, эталонная архитектура отображает эти функции на декомпозицию системы. Соответствие может быть как однозначным, так и не однозначным. В программном элементе может быть реализована как отдельная часть функции, так и несколько функций сразу.

Эталонные модели, архитектурные образцы и эталонные архитектуры не являются вариантами архитектуры; это не более чем полезные понятия, способствующие фиксации отдельных элементов архитектуры. Каждый из них появляется как результат проектных решений, принимаемых на самых ранних этапах. Отношение между этими проектными элементами представлено на рис. 2.2.

Рис. 2.2. Отношения между эталонными моделями, архитектурными образцами, вариантами эталонной и программной архитектуры. (С помощью стрелок мы указываем на то, что последующие понятия содержат большее количество проектных элементов.)

Аналогии архитектуры с другими значениями этого слова проводятся весьма часто. Как правило, архитектура ассоциируется с физической структурой (здания, улицы, аппаратура) и физическим расположением. Архитектор, разрабатывающий план здания, имеет целью обеспечить его доступность, эстетическую привлекательность, освещенность, эксплуатационную надежность и др. Программный архитектор в процессе проектирования системы должен стремиться к обеспечению таких характеристик, как параллелизм, модифицируемость, практичность, безопасность и т. д.; кроме того, он призван установить баланс между требованиями к этим характеристикам.

Аналогии между зданиями и программными системами не очень надежны, они слишком быстро оказываются несостоятельными. Хороши они тем, что по-

могают осознать важность позиции наблюдателя и прийти к выводу о множественности значений понятия «структура» в зависимости от мотивов ее изучения. Точное определение программной архитектуры значительно менее существенно, чем анализ сущности этого понятия.

2.4. Почему программная архитектура так важна?

В главе 1 мы в основном аргументировали значимость архитектуры для корпорации. В этой главе мы поговорим о том, почему архитектура так важна с технической точки зрения. В этом контексте следует привести три основных фактора.

1. *Взаимодействие между заинтересованными лицами.* Программная архитектура — это универсальная абстракция системы, на основе которой все или почти все заинтересованные в системе лица могут искать взаимопонимания, вести переговоры, находить компромиссы и просто общаться.
2. *Начальные проектные решения.* Программная архитектура содержит сведения о том, какие решения были приняты на ранних этапах разработки системы. Значимость подобного рода сводок отнюдь не ограничивается удельным весом этих этапов относительно оставшихся операций разработки, размещения и сопровождения. Именно в это время впервые появляется возможность анализа проектных решений, определяющих дальнейшую разработку системы.
3. *Переносимая абстракция системы.* Программная архитектура — это относительно небольшая, вполне доступная для человеческого восприятия модель структурирования системы и взаимодействия ее компонентов; помимо прочего, эта модель обладает свойством переносимости из системы в систему. Например, ее можно применить в отношении других систем, для которых характерны примерно те же требования к атрибутам качества и функциональным возможностям, и тем самым дать начало полноценному многократному применению.

Каждый из этих вопросов мы обсудим отдельно.

Архитектура как средство организации общения между заинтересованными лицами

Каждое заинтересованное в программной системе лицо — заказчик, пользователь, руководитель проекта, программист, испытатель и т. д. — требует внедрить в систему различные характеристики, основа для которых закладывается в архитектуре. К примеру, пользователю необходимо, чтобы система была надежной и доступной в любой момент; заказчика интересует, чтобы архитектура была реализована без расхождений с графиком и бюджетом; менеджер озабочен не только соблюдением временных и финансовых ограничений, но и возможностью при помощи архитектуры организации относительно независимой работы различных групп

разработчиков, их четкого и управляемого взаимодействия. Архитектор думает о том, какие стратегии позволят ему реализовать все эти задачи.

Архитектура — это общий язык, на котором можно сформулировать, обговорить и решить те или иные проблемы, причем сделать это на таком уровне, который, даже если речь идет о сложных системах, вполне доступен для комплексного изучения человеком (см. врезку «Что будет, если нажать эту кнопку?»). В отсутствие такого языка анализ крупных систем на ранних этапах сильно затрудняется; сложнее становится и принимать значимые решения, определяющие качество и полезность. Как мы увидим на материале части 3, архитектурный анализ одновременно отталкивается от этого уровня общения и вносит в него некоторые коррективы.

ЧТО БУДЕТ, ЕСЛИ НАЖАТЬ ЭТУ КНОПКУ?

Архитектура как средство организации общения между заинтересованными лицами

Заслушивался отчет о проекте. Его разработка, финансируемая из правительственного фонда, давно вышла за рамки графика и бюджета. Масштаб же его был столь серьезен, что упомянутые недочеты удостоились внимания конгрессменов, и теперь, пытаясь решить забытую было проблему, правительство организовало занудную отчетную сессию с обязательным посещением. Компанию-разработчика недавно перекупили, однако делу это не помогло. На второй день сессии была запланирована презентация программной архитектуры. Молодой архитектор — ученик главного архитектора системы — бодро объяснял, каким образом архитектура столь масштабной системы обеспечит соответствие высоким требованиям по работе в реальном времени, распределенности и высокой надежности. Основательная архитектура презентовалась не менее основательно. Анализ проводился тщательно и корректно. Тем не менее слушатели — около 30 представителей правительства с различными функциями в управлении и надзоре за этим нелегким проектом — утомились. Некоторые из них даже подумывали, что легче уйти в недвижимость, чем пытаться выдержать очередной отчет по принципу «ну давайте же наконец сделаем все как надо».

На схеме, составленной на полуформальной нотации из прямоугольников и линеек, были зафиксированы основные программные элементы представления системы периода прогона. Названия обозначались сокращениями, и если бы молодой архитектор не давал объяснений, в них вряд ли можно было бы разобраться. Линии указывали потоки данных, передачу сообщений и синхронизацию процессов. По словам архитектора, эти элементы внутренне избыточны. «В случае сбоя, — сказал он, наведя лазерную указку на одну из линий, — на этом пути сработает механизм перезапуска...»

«А что произойдет при нажатии кнопки выбора режима работы?» — внезапно прервал его один из слушателей. Им оказался представитель правительства — предполагаемого коллектива пользователей обсуждаемой системы.

«Простите?» — Архитектор не понял вопроса.

«Кнопка выбора режима работы, — повторил человек из правительства. — Что будет, если я ее нажму?»

«Ну-у, при этом запускается событие в драйвере устройства, вот здесь, — начал архитектор, манипулируя указкой. — Затем считываются данные в регистре, интерпретируется код события. Если речь идет о выборе режима, то... на доску объявлений будет подан сигнал, та, в свою очередь, сигнализирует объектам, которые на это событие подписались...»

«Нет, вы не поняли. Я имею в виду — что делает система? — Государственный муж вновь прервал докладчика. — Дисплей перезапускается? И что, если это произойдет в ходе реконфигурации системы?»

Несколько озадаченный архитектор убрал указку. Вообще-то вопрос не имел отношения к архитектуре, но, поскольку он все-таки архитектор, а положение это обязывает знать все требования, он ответил. «Если командная строка находится в режиме настройки, дисплеи перезапустятся, — изрек докладчик. — В противном случае на пульт управления выводится

сообщение об ошибке, а сигнал игнорируется». — Лазерная указка вновь материализовалась. — «Так вот, что касается механизма перезапуска...»

«Вы знаете, я вот почему интересуюсь, — продолжил будущий пользователь. — Исходя из вашей схемы, складывается впечатление, что дисплейный пульт подает сигналы целевому модулю.»

«А что, собственно, должно произойти? — В разговор включился еще один слушатель, адресуя свой вопрос любознательному джентльмену. — Вы что, хотите, чтобы во время перестройки режима пользователь получал какие-то данные об этом режиме?»

В течение последующих 45 минут архитектор внимательно наблюдал за тем, как слушатели тратят отведенное ему время на обсуждение корректного поведения системы в самых разных, одним им известных состояниях.

Дискуссия не имела отношения к архитектуре, однако именно она (и ее графическое представление) послужила побудительным мотивом. Архитектуру имеет смысл рассматривать как основу для взаимодействия заинтересованных лиц помимо архитекторов и разработчиков. К примеру, с ее помощью руководители проектов формируют рабочие группы и распределяют между ними ресурсы. Ну а пользователи? Для них архитектура незаметна, так с какой стати они должны вникать в сущность системы именно с ее помощью?

Но именно так они и делают. В представленном случае человек, который начал задавать вопросы, просидел два дня, уставившись в схемы функций, операций, пользовательского интерфейса и тестирования. Он устал, ему хотелось домой, но именно во время изложения отчета об архитектуре он понял, что что-то ему непонятно. Прослушав множество отчетов о вариантах архитектуры, я убедился в том, что рассмотрение системы под новым углом выводит на поверхность многие неясности. В такой роли для пользователей часто выступает именно архитектура, причем вопросы, которые они начинают задавать, оказываются по своему характеру поведенческими. Во врезке «Их решение не годится» (см. главу 11) мы рассмотрим пример процесса оценки архитектуры, в ходе которого представители пользователей больше интересовались не тем, как система будет работать, а тем, что она сможет делать, и это совершенно естественно. До представленного момента все их контакты с производителем осуществлялись только через посредство продавцов. Архитектор — настоящий специалист по интересующей их системе, и им довелось с ним пообщаться; вполне объяснимо, что они, нисколько не колеблясь, воспользовались моментом.

Очевидно, что рассматриваемую проблему можно решить при помощи тщательно составленных, всесторонних спецификаций требований; впрочем, в силу самых разных причин не во всех случаях они создаются или оказываются доступными. В отсутствие подобных документов именно спецификация архитектуры способна вывести недопонимание на уровень вопросов и таким образом внести в проблему ясность. Этот эффект разумнее принять к сведению, чем опровергать. В главе 11 мы будем рассматривать уточнение требований и их классификацию согласно приоритетам как один из основных аргументов за проведение оценки архитектуры.

Иногда подобная практика помогает выявить необоснованные требования, к обсуждению полезности которых впоследствии можно возвратиться. Проведение такого рода обзора, в ходе которого акцент ставится на взаимодействие требований и архитектуры, позволило бы вышеупомянутому молодому архитектору отвлечься от основной темы и уделить в рамках отчетной сессии некоторое время на разбор соответствующей информации. Тогда и представитель гвардии пользователей не почувствовал бы себя неуютно, задав вопрос в самый неподходящий момент. Хотя, в конце концов, он всегда может уйти в недвижимость.

— PCC

В архитектуре излагаются начальные проектные решения

Программная архитектура отражает самые ранние проектные решения относительно системы. Их труднее всего принять, а по мере разработки — скорректировать; при этом последствия наиболее существенны.

Архитектура определяет ограничения реализации

Реализация отражает архитектуру только в том случае, если она соответствует изложенным в ней структурным проектным решениям. Таким образом, реализацию следует разделить на ряд установленных архитектурой элементов; взаимодействие между этими элементами должно производиться в установленном архитектурой порядке; обязательства каждого отдельного элемента в отношении других элементов также должны исполняться согласно требованиям архитектуры.

Решения о распределении ресурсов тоже накладывают на реализацию определенные ограничения. Проблема в том, что конструкторы, работающие над отдельными элементами, иногда не имеют об этих решениях ни малейшего представления. Рассматриваемые ограничения подразумевают разделение задач, благодаря которому менеджерские решения в плане трудовых и вычислительных ресурсов реализуются наилучшим образом. От конструкторов отдельных элементов требуется знание их спецификаций, однако во всем, что касается компромиссных архитектурных решений, они могут пребывать в полном неведении. Архитекторы, напротив, обычно не являются экспертами в области разнообразных аспектов построения алгоритмов и тонкостей языков программирования, однако именно они ответственны за принятие архитектурных компромиссов.

Архитектура определяет организационную структуру

Мало того что архитектура устанавливает структуру разрабатываемой системы — эта последняя интегрируется в структуру проекта разработки[1] (а иногда, как видно на материале главы 1, даже в структуру компании-разработчика). Стандартный способ разделения труда по разработке крупной системы предполагает распределение отдельных частей системы между несколькими группами конструкторов. Это называется декомпозицией обязанностей по разработке системы. Поскольку в состав системной архитектуры входит высокоуровневая декомпозиция системы, именно она обычно служит основой для декомпозиции обязанностей; та, в свою очередь определяет структуру планирования, определения сроков и бюджета; каналы межгруппового взаимодействия; организацию управления конфигурациями и файловой системой; планы и процедуры интеграции и тестирования, и даже такие мелочи, как организация локальной сети проекта и количество групповых попоек. Процесс сопровождения также отражает программную структуру — для сопровождения отдельных структурных элементов создается ряд рабочих групп.

У декомпозиции обязанностей по разработке системы есть один побочный эффект — дело в том, что некоторые аспекты программной архитектуры замораживаются. Группа, ответственная за одну из подсистем, противится распространению своих обязанностей на другие группы. Если эти обязанности зафиксированы в договоре, их пересмотр оказывается довольно затратным предприятием. Отслеживание работы по совокупности распределенных задач также усложняется.

---

[1] Забавная ситуация: в английском языке для обозначения проекта как организационной единицы (project) и проекта как продукта разработки (design) используют два разных слова, в русском — одно. Это доводит муки творчества переводчиков (воздадим им должное!) до невообразимых пределов. — *Примеч. науч. ред.*

Изменить утвержденную архитектуру по управленческим и коммерческим соображениям становится практически невозможно. Это — дополнительный аргумент (наряду со многими другими) в пользу комплексной оценки архитектуры, которую мы настоятельно рекомендуем проводить перед замораживанием программной архитектуры крупной системы.

Архитектура сдерживает или способствует реализации атрибутов качества системы

Способность/неспособность системы реализовывать предполагаемые (или затребованные) атрибуты качества в значительной степени обусловливается архитектурой. Отношения между вариантами архитектуры и качеством подробно рассматриваются в главе 5; пока что вам следует усвоить следующие постулаты.

- Если система ориентирована на высокую производительность, необходимо следить за временным поведением ее элементов, а также за частотой и объемом межэлементного взаимодействия.
- Если приоритетом является модифицируемость, то обязанности между элементами следует распределять таким образом, чтобы вносимые в систему изменения не сопровождались далеко идущими последствиями.
- Если система должна стать серьезно защищенной, то особое внимание необходимо обратить на безопасность межэлементного взаимодействия и информацию, к которой те или иные элементы имеют право доступа. Кроме того, в архитектуру имеет смысл ввести ряд специализированных элементов (например, надежное ядро).
- Если упор в процессе разработки системы планируется сделать на масштабируемости, необходимо обеспечить строгую локализацию использования ресурсов, чтобы тем самым упростить введение элементов с более прогрессивными возможностями.
- Если, согласно требованиям проекта, систему следует поставлять поэтапно, в виде отдельных подмножеств, то особое внимание, опять же, необходимо обратить на межэлементное взаимодействие.
- Если для элементов данной системы планируется обеспечить возможность многократного применения в рамках других систем, то сцепление между элементами следует ограничить; иначе при извлечении из текущей среды отдельного элемента с ним придется переносить многочисленные дополнения.

Стратегии реализации этих и других атрибутов качества в высшей степени архитектурны. При этом важно понимать, что сама по себе архитектура не гарантирует ни функциональности, ни качества. Непродуманные нисходящие проектные и реализационные решения имеют обыкновение подрывать основы качественных архитектурных проектов. На качество системы влияют все решения, вне зависимости от того, на каком этапе ее жизненного цикла — высокоуровневого проектирования, кодирования или реализации — они принимаются. Следовательно, качество нельзя признать полностью находящимся в зоне ответственности архитектурного проектирования. В контексте обеспечения качества хорошая архитектура необходима, но не достаточна.

Прогнозирование системного качества путем изучения архитектуры

Можно ли, не дожидаясь окончания разработки и размещения системы, утверждать, что те или иные архитектурные решения приняты верно (иначе говоря, что система реализует требуемые атрибуты качества)? Будь ответ на этот вопрос отрицательным, задача выбора архитектуры потеряла бы всякий смысл — ее можно было бы выбирать совершенно произвольно. К счастью, прогнозы относительно качества системы можно делать, отталкиваясь исключительно от результатов оценки ее архитектуры. Методики оценки архитектур наподобие метода анализа компромиссных архитектурных решений (ATAM, см. главу 11) обеспечивают возможность нисходящего ознакомления с атрибутами качества программных продуктов; возможность эта появляется (но одновременно ограничивается) благодаря вариантам программной архитектуры.

Архитектура облегчает анализ изменений и их организацию

Сообщество разработчиков программного обеспечения пытается бороться с тем, что около 80 % издержек на производство типичной программной системы приходится на период, следующий за первоначальным развертыванием. Из этого обстоятельства мы можем сделать вывод, что большинство систем находятся именно на этом этапе. Многим программистам и проектировщикам, а возможно, даже большинству из них, никогда не приходилось участвовать в разработке «с чистого листа» — как правило, в своей работе они ограничены существующим программным кодом. В продолжение своего жизненного цикла программы претерпевают изменения; происходят они часто, а реализуются во многих случаях с некоторыми трудностями.

Любая архитектура предполагает разделение всех возможных изменений на три категории: локальные, нелокальные и архитектурные. Для того чтобы внести локальное изменение, достаточно откорректировать отдельный элемент. Нелокальное изменение требует корректировки ряда элементов, однако базовые архитектурные принципы остаются неизменными. Архитектурное изменение затрагивает сущность взаимодействия между элементами — образец архитектуры — и в большинстве случаев предполагает корректировку всей системы. Локальные изменения, естественно, предпочтительнее остальных; самой эффективной в этой связи следует считать ту архитектуру, в которой наиболее вероятные изменения внести легче всего.

Для принятия решений о необходимости внесения изменений, установления наименее опасных путей, оценки последствий предлагаемых изменений, определения последствий и расстановки приоритетов относительно требуемых изменений — для всего этого требуется подробный анализ взаимоотношений, производительности и поведения программных элементов системы. Все эти задачи значатся в должностной инструкции архитектора. Средством приобретения знаний, достаточных для принятия решений о предполагаемых изменениях, мы считаем анализ архитектуры.

Архитектура облегчает эволюционное макетирование

Любую определенную архитектуру можно проанализировать и смоделировать в качестве макета. Процесс разработки от этого выигрывает в двух отношениях.

1. Система становится исполняемой на раннем этапе жизненного цикла продукта. Ее точность повышается по мере замены элементов макета полноценными версиями программного обеспечения. В качестве элементов макета могут выступать как неточные версии конечной функции, так и суррогаты, которые потребляют и производят данные на нужных скоростях.
2. Из того, что система довольно рано становится исполняемой, следует, что потенциальные проблемы, связанные с производительностью, выявляются на ранних этапах ее жизненного цикла.

Каждый из этих факторов снижает риск разработки. Если архитектура входит в состав семейства родственных систем, то издержки создания структуры макетирования можно разнести на разработку других систем из этого семейства.

Архитектура позволяет более точно рассчитывать стоимость и сроки

Рассчитав стоимость и составив график, руководитель может, во-первых, получить необходимые ресурсы, а во-вторых, узнать, удовлетворительно ли состояние проекта. Расчеты стоимости на основе блоков системы оказываются точнее тех, которые базируются на общих знаниях о системе. Как мы уже говорили, организационная структура проекта определяется его архитектурой. Расчеты по блокам лучше поручить не руководителю проекта, а участникам соответствующей группы разработчиков — они, во-первых, справятся с этой задачей точнее, а во-вторых, будут ощущать ответственность за соблюдение установленных сроков. Далее — первоначальное определение архитектуры предполагает, что требования к системе уже рассмотрены и (возможно даже) утверждены. Чем больше нам известно о масштабе системы, тем точнее проводятся расчеты.

Архитектура как переносимая модель многократного применения

Чем раньше начинается повторное использование, тем оно полезнее. Повторное использование кода, несомненно, полезно, однако многократное применение на уровне архитектуры для систем с аналогичными требованиями подразумевает поистине громадные возможности. Повторное применение в этом случае распространяется не только на код, но, в первую очередь, и на требования, которые привели к созданию архитектуры, а также на знания по конструированию повторно использованной архитектуры. Естественно, что при многократном применении начальных архитектурных решений в рамках ряда систем на них распространяются все вышеупомянутые последствия.

Линейки программных продуктов строятся на основе общей архитектуры

Линейкой, или семейством, программных продуктов называется ряд преимущественно программных систем, для которых характерен общий, управляемый набор характеристик, реализующих потребности конкретного сегмента рынка или конкретные задачи и разработанных в установленном порядке на основе общего набора основных активов. Главным из этих активов является архитектура, спроектированная в расчете на удовлетворение потребностей всего семейства. Архитектор линейки продуктов старается выбрать для нее такую архитектуру (или семейство близкородственных вариантов архитектуры), которая сможет обслуживать всех ее возможных участников; при этом те проектные решения, которые касаются линейки продуктов в целом, принимаются сначала, а те, которые затрагивают только отдельных ее членов, — впоследствии. Архитектура определяет постоянные и переменные элементы линейки продуктов. Линейки программных продуктов — мощное средство мультисистемной разработки — на порядки улучшают такие характеристики, как срок выхода на рынок, стоимость, продуктивность и качество. Значимость архитектуры определяется самой сущностью этой парадигмы. Аналогично прочим капиталовложениям, архитектура линейки продуктов становится одним из основных активов компании-разработчика. Анализ линеек программных продуктов приводится в главе 14, а соответствующие конкретные примеры — в главах 15 и 17.

Системы можно конструировать на основе крупных элементов, созданных сторонними разработчиками

Раньше программные парадигмы были ориентированы на *программирование* (programming) как на основной вид деятельности, а проделанная работа измерялась в строках кода; архитектурно-ориентированная разработка во многих случаях направлена на *компоновку* (composing) и *сборку* (assembling) элементов, которые, вероятнее всего, разрабатывались отдельно, а может быть, даже независимо друг от друга. Возможность такой композиции появляется по той причине, что архитектура определяет те элементы, которые можно ввести в систему. В то же время она ограничивает проведение замен (или добавлений), исходя при этом из того, как они взаимодействуют со своей средой, как получают и передают управление, какие данные потребляют и производят, как обращаются к данным и при помощи каких протоколов они осуществляют обмен информацией и совместное использование ресурсов.

Одним из наиболее важных аспектов архитектуры является организация структуры ее элементов, интерфейсов и рабочих понятий. *Взаимозаменяемость* (interchangeability) считается самым важным принципом этой организации. Запущенное в 1793 году Эли Уитни (Eli Whitney) массовое производство мушкетов по принципу взаимозаменяемых частей положило начало эпохе промышленного производства. В отсутствие надежных физических измерений эта идея не внушала доверия. В контексте современного программного обеспечения, пока добиться надежного разграничения абстракций не удалось, принцип структурной взаимозаменяемости кажется не менее устрашающим, однако от этого он не теряет своей важности.

Готовые компоненты, подсистемы, совместимые интерфейсы передачи данных — все они основываются на принципе взаимозаменяемости. При этом нерешенными остаются многие проблемы, связанные с разработкой программного обеспечения методом композиции. Когда в качестве компонентов, претендующих на внесение и многократное применение, выступают индивидуальные подсистемы, сконструированные на основе противоречащих друг другу архитектурных допущений, интеграция их функций может столкнуться с непредвиденными сложностями. Вслед за Дэвидом Гарланом (David Garlan) и некоторыми его коллегами эту ситуацию стали называть *архитектурным несоответствием* (architectural mismatch).

Чем меньше, тем больше: ограничивать словарь проектных альтернатив выгодно

По мере накопления полезных архитектурных образцов и образцов проектирования становится совершенно очевидно, что, несмотря на более или менее безграничные возможности комбинирования компьютерных программ, добровольное сведение вариантов к относительно небольшому количеству альтернатив в контексте взаимодействия между программами и их сотрудничества может принести определенные выгоды. Таким образом, проектная сложность конструируемой системы сводится к минимуму. Среди преимуществ этого принципа следует упомянуть повышенные возможности многократного применения, более стандартизированные и простые решения, которые легче понять и распространить, углубление анализа, сокращение длительности отбора и совершенствование способности к взаимодействию.

Свойства программного проекта определяются выбранным архитектурным образцом. Те образцы, которые в наибольшей степени подходят для решения конкретной задачи, повышают качество реализации конечного проектного решения, а иногда и упрощают балансирование противоречащих проектных ограничений — этот эффект достигается путем углубления анализа плохо изученных проектных контекстов и/или формулирования противоречивости технических требований.

СИСТЕМНАЯ АРХИТЕКТУРА И ПРОГРАММНАЯ АРХИТЕКТУРА

На протяжении последних 5–10 лет мы часто выступали с сообщениями о программной архитектуре. Все это время среди слушателей находился кто-то, кто спрашивал: «Почему вы говорите о программной архитектуре? Ведь системная архитектура не менее важна!» или «В чем разница между программной и системной архитектурой?»

На самом деле, как выясняется, разница небольшая. То, что мы обычно говорим именно о программной архитектуре, объясняется желанием подчеркнуть важность принимаемых архитектором программных решений, которые распространяются на качество продукта в целом.

Случаи, когда в процессе создания программной архитектуры соображения по поводу системы не принимаются в расчет, крайне редки. К примеру, если вы хотите, чтобы архитектура была высокопроизводительной, у вас должно быть хотя бы некоторое представление о характеристиках той аппаратной платформы, на базе которой она будет работать (скорость процессора, емкость памяти, скорость обращения к диску); кроме того, вы должны знать характеристики всех устройств, с которыми системе предстоит взаимодействовать (традиционные устройства ввода-вывода, сенсоры, исполнительные механизмы), а в дополнение к этому желательно найти информацию о характеристиках сети (в первую очередь, о ее пропускной способности). Если архитектура должна быть сверхнадежной, вам, опять же, придется иметь дело с аппаратной частью — в этом случае с интенсивностью отказов, наличием

резервирования средств обработки и сетевых устройств. И так далее и тому подобное. Архитекторы редко обходят стороной вопросы аппаратного обеспечения.

Итак, проектируя программную архитектуру, следует представлять себе всю систему — как аппаратную, так и программную части. Отказываясь от этого принципа, вы напрашиваетесь на неприятности. Ни один инженер не возьмется прогнозировать характеристики системы, если специфицирована лишь одна из ее частей.

И тем не менее мы продолжаем настаивать на употреблении термина «программная архитектура». Почему же все-таки не «системная»? Дело в том, что большинство альтернатив архитектора связано не с аппаратурой, а с программным обеспечением. Речь не идет о том, что решений относительно аппаратной части принимать не надо, — просто она может быть неподконтрольна архитектору (именно такая ситуация складывается при разработке системы, которая должна работать на случайных клиентских машинах с подключением к Интернету); с другой стороны, все решения, касающиеся аппаратуры, могут приниматься другими лицами (по экономическим и правовым мотивам или согласно стандартам); кроме того, аппаратура имеет обыкновение со временем заменяться.

Все вышеприведенные факторы позволяют нам на вполне законных основаниях сосредоточиться на программной части архитектуры — именно здесь принимается большинство основополагающих решений, существует множество альтернатив и возможностей добиться успеха (или привести к полному провалу!).

— RK

Архитектура предполагает возможность разработки на основе шаблонов

В составе архитектуры присутствуют проектные решения о взаимодействии элементов; отраженные в реализациях всех элементов, они тем не менее локализуются и фиксируются лишь единожды. Для местной фиксации механизмов межэлементного взаимодействия служат шаблоны. К примеру, в шаблоне можно закодировать объявления публичной области элемента, в которой будут аккумулироваться результаты, или протоколы, применяемые элементом для взаимодействия с исполняемым файлом системы. Пример ряда четких архитектурных решений, предполагающих возможность шаблонной разработки, приводится в главе 8.

Архитектура помогает в процессе обучения

Архитектура, содержащая описание взаимодействия элементов в рамках требуемого поведения, может послужить в качестве своеобразной инструкции, вводящей новых участников проекта в курс дела. Этим дополнительно подтверждается наше утверждение о том, что одной из основных функций программной архитектуры является организация и содействие общению между представителями разношерстных заинтересованных групп. Архитектура — это хорошая опорная точка.

2.5. Архитектурные структуры и представления

Невропатологи, гематологи и дерматологи представляют структуру человеческого тела по-разному. Офтальмологи, кардиологи и ортопеды специализируются на подсистемах. Кинезиологи и психиатры занимаются различными аспектами по-

ведения организма в целом. Все эти представления фиксируются по-разному и обладают разными свойствами, однако, по сути, они связаны — все вместе они описывают архитектуру тела.

Та же ситуация с программным обеспечением. Современные программные системы настолько сложны, что разбирать их в комплексе крайне сложно. Приходится концентрировать внимание на одной или нескольких структурах программной системы. Для того чтобы рассуждать об архитектуре осмысленно, мы должны определиться с тем, какая структура или какие структуры в данный момент являются предметом обсуждения, — о каком *представлении* (view) архитектуры мы говорим.

Рассматривая представление архитектуры, мы будем употреблять связанные между собой понятия *структуры* (structure) и *представления* (view). Представление — это отображение ряда связанных архитектурных элементов в том виде, в котором ими оперируют заинтересованные в системе лица. В нем фиксируется отображения совокупности элементов и установленных между ними связей. Структура же — это собственно ряд элементов, существующих в рамках программного или аппаратного обеспечения. В частности, модульная структура представляет собой набор модулей системы с указанием их организации. Модульное представление есть отображение этой структуры, документированное и применяемое теми или иными заинтересованными лицами. Несмотря на то что эти термины иногда используются как синонимы, мы намерены придерживаться приведенных определений.

Архитектурные структуры подразделяются на три общие группы, в каждую из которых включается элементы определенного характера.

- *Модульные структуры.* Элементами таких структур являются модули — блоки реализации. Модули предполагают рассмотрение системы с точки зрения кода. Им как отдельным областям выделяются определенные функциональные обязанности. Особого внимания тому, как конечное программное обеспечение заявит себя в период прогона, в данном случае не уделяется. Модульные структуры позволяют отвечать на такие вопросы, как: «Какие основные функциональные обязанности несет данный модуль? К каким программным элементам он может обращаться? Какое программное обеспечение он фактически использует? Между какими модулями установлены отношения обобщения или специализации (например, наследования)?»
- *Структуры «компонент и соединитель».* В данном случае элементами являются компоненты (основные единицы вычислений) и соединители (инструменты взаимодействия между компонентами) периода прогона. Среди вопросов, на которые отвечают структуры «компонент и соединитель», — такие, например, как: «Каковы основные исполняемые компоненты и как происходит их взаимодействие? Каковы основные совместно используемые хранилища данных? Какие части системы воспроизводятся? Каким образом по системе проходят данные? Какие элементы системы способны исполняться параллельно? Какие структурные изменения происходят с системой во время ее исполнения?»
- *Структуры распределения.* Структуры распределения демонстрируют связь между программными элементами, с одной стороны, и элементами одной

или нескольких внешних сред, в которых данное программное обеспечение создается и исполняется, — с другой. Они отвечают на вопросы: «На каком процессоре исполняется данный программный элемент? В каких файлах каждый элемент хранится в ходе разработки, тестирования и конструирования системы? Каким образом программные элементы распределяются между группами разработчиков?»

Эти три структуры соответствуют трем универсальным типам решений, принимаемым в ходе архитектурного проектирования:

- Каким образом следует структурировать совокупность блоков кода (модулей) системы?
- Каким образом следует структурировать совокупность элементов системы, обладающих поведением (компоненты) и демонстрирующих взаимодействие (соединители) в период прогона?
- Каким образом следует установить связи между системой и непрограммными структурами среды (например, с процессорами, файловыми системами, сетями, группами разработчиков и т. д.)?

Программные структуры

Наиболее распространенные и полезные программные структуры изображены на рис. 2.3. Им посвящен ряд последующих разделов.

Модуль

Модульные структуры делятся на следующие разновидности.

- *Декомпозиция*. В качестве блоков выступают модули, между которыми установлены отношения «является подмодулем...»; таким образом, крупные модули в рекурсивном порядке разлагаются на меньшие, и процесс этот завершается только тогда, когда мелкие модули становятся вполне понятными. Модули в рамках этой структуры часто используются в качестве отправной точки для последующего проектирования — архитектор перечисляет блоки, с которыми ему предстоит работать, и в расчете на более подробное проектирование, а также, в конечном итоге, реализацию распределяет их между модулями. У модулей во многих случаях есть связанные продукты (например, спецификации интерфейсов, код, планы тестирования и т. д.). Структура декомпозиции в значительной степени обеспечивает модифицируемость системы — при этом складывается ситуация, когда наиболее вероятные изменения приходятся на долю нескольких небольших модулей. Довольно часто декомпозиция задействуется как основа для организации проекта, включающая структуру документации, планы интеграции и тестирования. Иногда блоки этой структуры получают оригинальные названия (от пользующихся ими организаций). К примеру, в ряде стандартов министерства обороны США определяются элементы конфигурации компьютерных программ (Computer Software Configuration Items, CSCI) и компоненты компьютерных программ (Computer Software Components,

CSC), которые представляют собой не что иное, как блоки модульной декомпозиции. В главе 15 мы рассмотрим системные функции и их группы в качестве блоков декомпозиции.

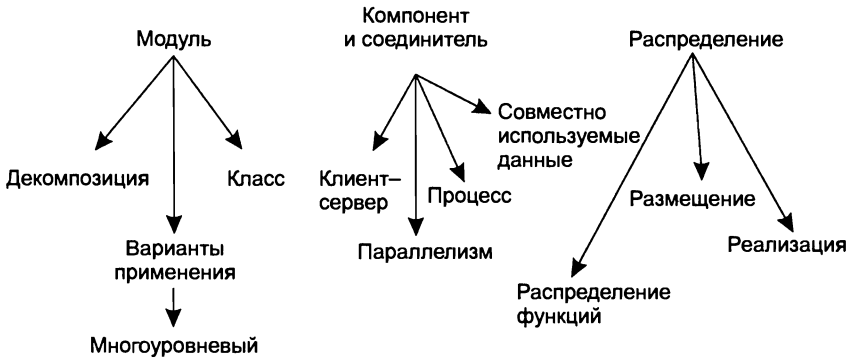


Рис. 2.3. Стандартные структуры программной архитектуры

- *Варианты использования*. Блоками этой важной, но не слишком распространенной структуры могут быть либо модули, либо (в случаях, когда требуется более мелкая структура) процедуры или ресурсы интерфейсов модулей. Между такими блоками устанавливаются *отношения использования* (uses relationship). Если для обеспечения правильности первого блока требуется наличие правильной версии (в отличие от заглушки) второго, то последний используется первым. Структура использования полезна при конструировании систем, которые либо легко расширяются дополнительными функциями, либо предполагают возможность быстрого извлечения полезных функциональных подмножеств. Способность без труда разбить рабочую систему на ряд подмножеств подразумевает возможность инкрементной разработки — многофункционального конструкторского приема, о котором мы поговорим несколько позже, в главе 7.
- *Многоуровневая*. Если отношения использования в рамках этой структуры находятся под строгим, особым образом осуществляемым контролем, возникает система уровней — внутренне связанных наборов родственных функций. В рамках строго многоуровневой структуры уровень n может обращаться к услугам только в том случае, если они предоставляются уровнем $n - 1$. На практике это правило существует в виде многочисленных вариантов (которые частично снимают представленное структурное ограничение). Уровни во многих случаях проектируются в виде абстракций (виртуальных машин), которые, стараясь обеспечить переносимость, скрывают детали реализации нижележащих уровней от вышележащих. Уровни участвуют в ряде приводимых в настоящем издании конкретных примеров — в главах 3, 13 и 15.
- *Класс*, или *обобщение*. Блоки модулей в рамках этой структуры называются классами. Отношения между ними строятся по образцам «наследует от...» и «является экземпляром...». Данное представление способствует анализу

коллекций сходного поведения или сходных возможностей (например, классов, которые наследуют от других классов) и параметрических различий, фиксация которых производится путем определения подклассов. Структура классов также позволяет анализировать вопросы повторного использования и инкрементного введения функциональности.

Компонент и соединитель

Среди структур данного вида выделяются следующие.

- *Процесс*, или *сообщающиеся процессы*. Подобно всем прочим структурам «компонент и соединитель», эта является ортогональной по отношению к модульным структурам, а связана она с динамическими аспектами исполняемой системы. В качестве блоков в данном случае выступают процессы или потоки, связь между которыми устанавливается путем передачи данных, синхронизации и/или операций исключения. Здесь, как и во всех остальных структурах «компонент и соединитель», действует отношение *прикрепления* (attachment), демонстрирующее связь компонентов друг с другом. Структура процессов существенно облегчает конструирование рабочей производительности и готовности системы.
- *Параллелизм*. Данная структура «компонент и соединитель» позволяет архитекторам выявлять перспективы параллелизма и локализовывать возможности состязаний за ресурсы. В качестве блоков выступают компоненты, а соединители играют роль «логических потоков». Логическим потоком называется такая последовательность вычислений, которую впоследствии, в ходе процесса проектирования, можно связать с отдельным физическим потоком. Структура параллелизма задействуется на ранних этапах процесса проектирования, способствуя выявлению требований к организации параллельного исполнения.
- *Совместно используемые данные*, или *репозитарий*. В состав данной структуры входят компоненты и соединители, обеспечивающие создание, хранение и обращение к данным постоянного хранения. Она наилучшим образом приспособлена к таким ситуациям, когда система структурирована на основе одного или нескольких репозитариев совместно используемых данных. Она отражает производство и потребление данных программными элементами периода прогона, а в деле обеспечения высокой производительности и целостности данных эти сведения представляют большую ценность.
- *Клиент–сервер*. Эта структура предназначена для систем, сконструированных в виде группы взаимодействующих клиентов и серверов. В качестве компонентов в данном случае выступают клиенты и серверы, а соединителями являются протоколы и сообщения, которыми они обмениваются в процессе обеспечения работоспособности системы. Такая структура требуется для разделения задач (обеспечивающего модифицируемость), физического распределения и выравнивания нагрузок (обеспечивающего эффективность исполнения).

Распределение

Среди структур распределения выделяются следующие.

- *Размещение.* Структура размещения отражает распределение программного обеспечения между элементами аппаратной обработки и передачи данных. В качестве распределяемых элементов могут выступать программные продукты (как правило, процессы из представления «компонент и соединитель»), аппаратные объекты (процессоры) и каналы передачи данных. Отношения устанавливаются по распределению и демонстрируют физические устройства, на которых размещаются программные элементы; возможны также отношения миграции, однако они устанавливаются только в случае динамического распределения. Настоящее представление позволяет инженерам анализировать производительность, целостность данных, готовность и безопасность. Все эти характеристики чрезвычайно важны в условиях распределенных и параллельных систем.
- *Реализация.* Данная структура демонстрирует отображение программных элементов (обычно — модулей) на файловую структуру (структуры) в условиях разработки системы, интеграции и управления конфигурациями. Это крайне важно в контексте организации разработки и процессов конструирования.
- *Распределение функций.* Данная структура обеспечивает разделение обязанностей по реализации и интеграции модулей между соответствующими группами разработчиков. Наличие в составе архитектуры структуры распределения функций делает очевидным, что при принятии соответствующих решений учитывались как архитектурные, так и организационные факторы. Архитектору должно быть известно, какие навыки требуются от разных групп разработчиков. Кроме того, если речь идет о масштабных распределенных проектах с несколькими источниками, на основе структуры распределения функций можно выявлять функциональные сходства и назначать их одной группе разработчиков, отказываясь, таким образом, от их стихийной многократной реализации.

Общая схема программных структур приводится в табл. 2.1. В ней рассматривается значения элементов, отношения, характерные для каждой из структур, и варианты их практического применения.

Таблица 2.1. Архитектурные структуры системы

Программная структура	Отношения	Варианты практического применения
Декомпозиция	«Является подмодулем...»; «пользуется скрытой информацией совместно с...»	Распределение ресурсов, структурирование и планирование проекта; информационная закрытость, инкапсуляция; управление конфигурациями

продолжение

Таблица 2.1 (*продолжение*)

Программная структура	Отношения	Варианты практического применения
Варианты использования	«Требует наличия...»	Конструирование подмножеств; инженерные расширения
Многоуровневая	«Требует наличия...», «обращается к услугам...», «обобщает...»	Инкрементная разработка; реализация систем на основе переносимости «виртуальных машин»
Классы	«Является экземпляром...», «использует метод доступа из...»	В объектно-ориентированных системах проектирования, производящих на основе универсального шаблона быстрые, почти идентичные реализации
Клиент–сервер	«Обменивается данными с...», «зависит от...»	Распределенное функционирование; разделение задач; анализ производительности; выравнивание нагрузки
Процесс	«Исполняется параллельно с...», «может исполняться параллельно с...», «исключает», «предшествует» и т. д.	Анализ сроков; анализ производительности
Параллелизм	«Исполняется в одном логическом потоке»	Выявление местоположений, в которых потоки могут разветвляться, объединяться, создаваться и уничтожаться
Совместно используемые данные	«Производит данные», «потребляет данные»	Производительность; целостность данных, модифицируемость
Размещение	Распределение, миграция	Анализ производительности, готовности и защиты
Реализация	«Хранится в...»	Управление конфигурациями, интеграция, тестирование
Распределение функций	«Назначается...»	Управление проектом, оптимальное использование интеллектуальных ресурсов, управление общностью

Как правило, структура системы анализируется с точки зрения ее функциональности; при этом мы забываем о других ее свойствах: физическом распределении, взаимодействии процессов и синхронизации; все эти свойства обязательно учитываются на уровне архитектуры. Каждая структура содержит метод анализа тех или иных значимых атрибутов качества. К примеру, для того чтобы создать легко расширяемую или сокращаемую систему, необходимо *сконструировать* (именно сконструировать, а не просто зафиксировать) структуру использования. Структура процессов *конструируется* с целью исключения взаимоблокировки

и расширения узких мест. Структура декомпозиции модулей *конструируется* в расчете на производство модифицируемых систем и т. д. Каждая подобная структура снабжает архитектора оригинальным представлением системы и дополнительной базовой точкой проектирования.

Отношения между структурами

Все вышеперечисленные структуры предполагают различные представления и подходы к проектированию системы; все они эффективны и полезны сами по себе. В то же время они не самостоятельны. Элементы одной структуры связаны с элементами прочих, и на эти отношения следует обратить внимание. В частности, модуль в структуре декомпозиции может декларироваться как отдельный компонент, как часть отдельного компонента или как несколько компонентов в рамках одной из структур «компонент и соединитель», отражая, таким образом, своего представителя периода прогона. Как правило, между структурами устанавливаются отношения «многие ко многим».

Иногда в рамках отдельных проектов одна структура считается основной, а остальные структуры, если это возможно, определяются в ее категориях. Часто, хотя и не всегда, в качестве доминантной структуры выступает декомпозиция модулей. Объясняется это вполне разумно — декомпозиция модулей порождает структуру проекта. В главе 4 представлены сценарии, которые помогают развивать отдельные структуры и их связи с другими структурами. К примеру, если разработчик программного обеспечения желает внести в клиент–серверную структуру системы какие-то изменения, он должен учитывать представления процессов и размещения: дело в том, что клиент–серверные механизмы, как правило, задействуют процессы и потоки, а физическое распределение реализует различные механизмы управления, которые оказываются необходимы в случае локализации упомянутых процессов на одной машине. При необходимости изменения механизмов управления оценить степень их корректировки помогают представление декомпозиции модулей и многоуровневое представление.

Не все системы гарантируют учет различных архитектурных структур. Чем крупнее система, тем существеннее различие между этими структурами; впрочем, если речь идет о небольшой системе, обойтись, как правило, можно минимальными средствами. Вместо нескольких структур «компонент и соединитель» хватит одной. Если процесс в системе всего один, то структуру процессов можно свести к единичному узлу, который по мере проектирования не нужно будет сопровождать. Если распределение не планируется (все операции будут производиться на одном процессоре), необходимость в структуре размещения отпадает.

Структуры — это основные базовые точки конструирования архитектуры. Отдельные структуры отвечают за те или иные атрибуты качества. Они выражают принцип разделения задач при создании архитектуры (а также при ее последующем анализе и изложении для заинтересованных лиц). На материале главы 9 мы покажем, что структуры, отобранные архитектором на роль отправных точек конструирования, одновременно обеспечивают основу для документирования архитектуры.

Какие структуры выбрать?

Мы привели краткий обзор ряда полезных архитектурных структур, однако на самом деле их значительно больше. С какими из них архитектору имеет смысл работать? Какие архитектор должен задокументировать? Ну конечно, не все.

Недостатка в советах не наблюдается. В 1995 году Филипп Крюхтен (Philippe Kruchten) [Kruchten 95] опубликовал очень важную статью, в которой понятие архитектуры описывается в категориях составляющих ее структур; в ней же он советует сосредоточиться на четырех структурах. Для того чтобы удостовериться, что структуры не конфликтуют друг с другом, а, вместе взятые, действительно описывают удовлетворяющую всем требованиям систему, Крюхтен посоветовал обратиться к основным элементам Use Case. Этот метод, получивший название «4+1», стал довольно популярен, а впоследствии выступил в роли концептуальной основы рационального унифицированного процесса (Rational Unified Process, RUP). О каких именно представлениях рассуждает Крюхтен?

- *Логическое представление.* Элементы — это «ключевые абстракции», которые в контексте объектно-ориентированной технологии декларируются в качестве объектов и классов объектов. Это — модульное представление.
- *Процесс.* Это представление связывается с параллелизмом и распределением функций. Оно тождественно представлению «компонент и соединитель».
- *Разработка.* Это представление отражает организацию программных модулей, библиотек, подсистем и единиц разработки. Речь идет о представлении распределения, которое отображает программное обеспечение на среду разработки.
- *Физическое представление.* Отображая прочие элементы на узлы обработки и связи, оно также относится к представлению распределения (которое иногда называют представлением размещения).

Примерно в то же время, когда свою работу выпустил Крюхтен, другие исследователи — Сони, Норд и Хофмайстер [Soni 95] — опубликовали весьма заметную статью, в которой описывали структуры, довольно популярные среди архитекторов их компании. Речь шла о концептуальном представлении, представлениях соединения модулей и исполнения, а также о кодовом представлении. Они, опять же, точно соответствуют моделям модулей, распределения и «компоненту и соединителю».

За ними последовали другие работы, и список доступных структур стал быстро расширяться. Использовать все эти структуры, конечно, не стоит, хотя, с другой стороны, они по большей части присутствуют в конструируемых системах. Не забывайте, что от архитектора, помимо прочего, требуется понимать, каким образом различные структуры обеспечивают реализацию атрибутов качества; именно из этого соответствия и следует исходить, составляя перечень подходящих структур. В главе 9, рассуждая об архитектурном представлении, мы намерены более полно раскрыть суть этого принципа.

2.6. Заключение

В настоящей главе мы привели определение программной архитектуры и некоторых связанных с ней понятий: эталонной модели, эталонной архитектуры и архитектурного образца. Мы объяснили, почему в контексте программной инженерии архитектура играет столь существенную роль, выявили ее функции как источника начального накопления знаний о системе, как катализатора взаимодействия между заинтересованными лицами и как повторно используемого средства. Все эти функции мы рассмотрим в последующих главах.

Из нашего определения архитектуры становится совершенно понятно, что все системы состоят из множества структур. Представив некоторые наиболее распространенные структуры, мы объяснили, почему каждая из них служит своеобразной отправной точкой для всего последующего процесса проектирования.

В следующей главе мы впервые в этой книге рассмотрим конкретный пример. Он демонстрирует применимость различных архитектурных структур в процессе проектирования сложной системы.

2.7. Дополнительная литература

Понятийные основы изучения программной архитектуры были в значительной степени заложены ранними работами Дэвида Парнаса (David Parnas) (см. врезку «Архитектурное дежа вю»). Читатели Парнаса, скорее всего, отметили бы его фундаментальную статью об информационной закрытости [Parnas 72], работы по семействам программ [Parnas 76], неотъемлемым структурам программных систем [Parnas 74] и введение в структуру использования, ориентированную на конструирование подмножеств и супермножеств систем [Parnas 79]. Все эти исследования включены в более распространенный сборник его основных работ [Hoffman 01].

Пространная документация по программно-архитектурным образцам перечислена в работах «Pattern-Oriented Software Architecture» [Buschmann 96, Schmidt 00].

Применению архитектурных представлений в промышленных проектах посвящены работы [Soni 95] и [Kruchten 95]. На основе первой из них впоследствии была выпущена книга [Hofmeister 00] со всесторонним анализом представлений и их использования в ходе разработки и анализа. На основе содержания статьи [Kruchten 95], как мы уже говорили, построен рациональный унифицированный процесс, которому посвящено огромное количество исследований. Из них мы рекомендуем [Kruchten 00].

Сведения об архитектурном несоответствии содержатся в работе авторского коллектива под руководством Гарлана [Garlan 95]. Барри Боэм (Barry Boehm) [Boehm 95] раскрывает особенности процессов, относящиеся к программной архитектуре.

На веб-сайте программной архитектуры Института программной инженерии [SEI ATA] приводятся многочисленные ресурсы и ссылки на ресурсы, связанные

с программной архитектурой; среди прочего на нем опубликована подборка самых разнообразных определений этого термина.

Полиш (Paulish) [Paulish 02] рассматривает влияние на архитектуру финансовых и временных ограничений.

2.8. Дискуссионные вопросы

1. Программная архитектура часто сравнивается с архитектурой зданий. Перечислите допустимые аспекты такого сравнения. Как здания ассоциируются со структурами и представлениями программной архитектуры? Как они соотносятся с образцами? В чем недостатки такого сопоставления? На каком этапе оно становится неприемлемым?
2. В чем разница между эталонной архитектурой и архитектурным образцом? Какие возможности в части планирования деятельности организации и архитектурного анализа предусматривает одно из этих понятий и не предусматривает другое?

АРХИТЕКТУРНОЕ ДЕЖА ВЮ

Архитектура — несомненно, существенный элемент разработки системы — как область исследований в настоящее время пользуется большой популярностью. При этом следует напомнить, что отдельные ее аспекты уже давно и тщательно изучены. Во многих отношениях устойчивый интерес к архитектуре, который мы наблюдаем сейчас, — это «второе открытие» фундаментальных принципов, ярко и убедительно изложенных четверть века назад такими исследователями, как Фред Брукс, Эдсгер Дийкстра (Edsger Dijkstra), Дэвид Парнас и др.

Термин «архитектура» в программировании изначально употреблялся для описания неединично реализованных вычислительных систем. Это значение сохраняется в силе до сих пор. В 1969 году Фред Брукс и Кен Айверсон (Ken Iverson) определили архитектуру как «концептуальную структуру вычислительной машины... с точки зрения программиста» [Brooks 69]. Несколько лет спустя Брукс, ссылаясь на Г. Блау (G. Blaauw), привел другое определение архитектуры — как «комплексной и подробной спецификации пользовательского интерфейса» [Brooks 75]. Между архитектурой и реализацией было проведено четкое разграничение. Цитируя Блау, Брукс писал: «архитектура сообщает о том, *что* происходит, а реализация — о том, *как* это должно происходить». Такое разграничение принято и сегодня — в эпоху объектно-ориентированного программирования оно как нельзя кстати.

Некоторые исследовательские сообщества до сих пор употребляют термин «архитектура» для обозначения пользовательского представления системы, однако под *программной* архитектурой мы имеем в виду другое. Структура(ы), из которых состоит программная архитектура, невидимы для конечного пользователя системы. Тем не менее разграничение между тем, *что* происходит, и тем, *как* это происходит, справедливо и здесь. Программная архитектура не имеет отношения к тому, как элементы исполняют свои функции, равно как конечному пользователю все равно, как система справляется со своими задачами. Сегодня ядром определения программной архитектуры является понятие об архитектуре как об общем описании класса систем (то есть об абстракции, в которой архитектура присутствует во всех вариантах).

Еще в 1969 году Эдсгер Дийкстра советовал обращать особое внимание на декомпозицию и структуру программных средств, утверждая, что программировать с единственной целью — достичь корректного результата — недостаточно [Dijkstra 68]. В работе, посвященной одной из операционных систем, он ввел принцип многоуровневой структуры, согласно которому программы следует группировать по отдельным уровням, причем те из них, что находятся на одном уровне, должны взаимодействовать только со смежными уровнями. Дийкстра указывал на концептуальную целостность подобной организации и, как следствие, простоту разработки и сопровождения.

Работу в этом направлении продолжил Дэвид Парнас — исследователь, который своими трудами начала 1970-х годов стимулировал быстрое развитие программной инженерии. Именно он изложил большинство фундаментальных правил и принципов построения программных вариантов архитектуры, из числа которых следует особо выделить следующие:

- Принцип конструирования, описывающий разбиение системы на элементы с целью повышения удобства сопровождения и (в чем мы убедимся в главе 5) обеспечения возможности повторного использования. Сложно найти более фундаментальный принцип построения архитектуры, нежели этот, названный Парнасом принципом информационной закрытости [Parnas 72].
- Принцип обращения к элементу исключительно через его интерфейс. Это вообще концептуальная основа всего объектного проектирования [Parnas 72].
- Наблюдение, согласно которому любая программная система состоит из множества отдельных структур, сопровождающееся предостережением о недопустимости их смешения. Кстати, сегодняшние «архитектурщики» часто забывают этот совет [Parnas 74].
- Введение структуры использования — принципа управления связями между элементами с целью повышения расширяемости системы, обеспечения оперативного и несложного получения подмножеств [Parnas 79].
- Принцип выявления и обработки ошибок (теперь их называют исключениями) в компонентных системах, ставший в большинстве современных языков программирования основополагающим [Parnas 72, 76].
- Тезисы о том, что (1) любую программу следует рассматривать как члена семейства программ, (2) общность таких членов можно использовать в своих интересах и (3) те проектные решения, которые легче всего пересмотреть, необходимо реализовывать в последнюю очередь. Первичное структурирование программы — один из этапов создания архитектуры — должно предусматривать принятие начальных, распространяющихся на все семейство, проектных решений [Parnas 76].
- Признание воздействия структуры системы на атрибуты ее качества (в частности, на надежность) [Parnas 76].

Даже сам Парнас не отрицал, что некоторые его принципы явились разработкой существующих принципов. К примеру, относительно информационной закрытости он говорил, что лишь фиксирует то, чем программисты-профессионалы (в особенности программисты операционных систем — составители драйверов устройств) занимаются уже долгое время. Тем не менее, если взять исследования Парнаса за основу, то из них можно почерпнуть последовательное изложение базисных для программной архитектуры *структурных вопросов*. Его работы превращают программную архитектуру в полноценную область исследований. Без обращения к его постулатам ни одна новая книга на заданную тему не может претендовать на полноту изложения.

Недавно мы с одним коллегой спорили насчет того, что конкретно следует называть интерфейсом программного элемента; для обоих было очевидно, что именами программ, к которым существует возможность обращения, и принимаемыми ими параметрами определение интерфейса не исчерпывается. Коллега выразил предположение, что на самом деле речь идет о ряде допущений об элементе, которые мы можем принять с достаточной степенью уверенности и которые варьируют в зависимости от контекста применения этого элемента. Я согласился и показал ему статью Парнаса [Parnas 71], в которой тот говорил абсолютно о том же. Мой приятель, изрядно упавший духом, изрек: «Теперь я знаю, что почувствовал Скотт, когда дошел до Южного полюса и увидел там флаг Амундсена. Он, наверное, подумал: "Черт! Что делать? Меня же теперь загрызут!"».

Флаг Парнаса стоит, не пошатнувшись, а мы с завидной регулярностью обнаруживаем его на своем поле. В следующей главе мы рассмотрим конкретный пример архитектуры, которую Парнас создал для того, чтобы применить свои разработки в реальном приложении с высочайшими требованиями. С тех пор прошло много времени, но лично мы не знаем ни одного другого проекта, в котором архитектурные принципы были так четко изложены и так добросовестно проведены в жизнь, — в частности, это касается конструирования и сопровождения нескольких структур в расчете на реализацию задач по качеству; жесткой информационной закрытости, позволившей создать элементы многократного применения и такую

же архитектуру; и наконец, досконального специфицирования этой архитектуры, ее элементов и установленных между ними отношений.

Вскоре после того как исследователи во главе с Парнасом заложили основы построения архитектуры, дисциплина встала на путь эволюционного развития. Известно, что опыт применения фундаментальных принципов постепенно приводит к их уточнению, экстраполяции на теорию практики и, в конечном итоге, к появлению совершенно новых концепций. Так, несколько десятилетий назад Парнас писал в своих работах о семействах программ (program families); теперь же достижения в области организации, процессов и управления наблюдаются по большей части в контексте их наиболее успешных концептуальных наследников — линеек продуктов (product lines), — и на материале главы 14 вы сможете в этом убедиться. О разделении задач Дийкстра писал четверть века назад, но давно ли объекты (опять же, концептуальное продолжение изложенного им принципа) получили широкое распространение как стандарт проектирования? Работы Брукса и Блау об архитектуре изданы еще раньше, и сейчас мы знаем, что разобраться в архитектуре невозможно без учета ее экономической составляющей; существуют даже способы проведения анализа архитектур до фактического конструирования систем (их мы рассмотрим позже).

Причины того, что сегодня архитектура как область исследований развивается быстрыми темпами и уже достаточно широка, мы видим в обширном наследии высококлассных исследователей, с работ которых и началось это динамичное развитие. После необходимого уточнения и наработки опыта их практического применения идеи этих людей приобретают статус магистральных приемов конструирования систем.

— PCC

Глава 3
Авиационная система A-7E: конкретный пример применения архитектурных структур

> Структура объектно-ориентированной программы в период прогона во многих случаях существенно отличается от структуры ее кода. Структура кода фиксируется в период компиляции; она состоит из классов, отношения наследования которых неизменны. Структура программы периода прогона, напротив, представляет собой быстро изменяющуюся сеть из взаимодействующих объектов. Получается, что две эти структуры в значительной степени независимы друг от друга. Разобраться в одной, отталкиваясь от свойств другой, вряд ли проще, чем осознать динамический характер живых экосистем исходя из жесткой классификации растений и животных, и наоборот.
>
> *Э. Гамма, Р. Хелмс, К. Джонсон, Дж. Виссидес [Gamma 95]*

На материале главы 2 мы доказали, что программная архитектура описывает отдельные элементы системы и взаимосвязи между ними. Мы обратили внимание на то, что элементы систем неоднородны и для составления комплексного представления об архитектуре любой системы бывает полезно, и даже необходимо, привлекать разные архитектурные структуры. Каждая такая структура ориентирована на отдельный аспект архитектуры.

В настоящей главе мы намерены разобрать конкретный пример архитектуры, созданной путем конструирования и специфицирования трех архитектурных структур: *декомпозиции модулей* (module decomposition), *вариантов использования* (uses) и *процессов* (process). Мы покажем, как эти структуры дополняют друг друга, как в конечном итоге формируется полная картина функционирования

системы и как проявляется влияние структур на отдельные атрибуты качества системы. Параметры трех структур приводятся в табл. 3.1.

Таблица 3.1. Архитектурные структуры системы A-7E

Структура	Элементы	Отношения между элементами	Оказывает воздействие на
Декомпозиция модулей	Модули (блоки реализации)	«Является подмодулем...»; «пользуется секретом совместно с...»	Простота замены
Варианты использования	Процедуры	«Требует наличия...»	Способность к получению подмножеств и инкрементной разработке
Процесс	Процессы; поток процедур	«Синхронизируется с...»; «использует центральный процессор совместно с...»; «исключает»	Возможность планирования; достижение нужной производительности путем параллелизма

3.1. Связь с архитектурно-экономическим циклом

Механизм действия архитектурно-экономического цикла (АВС) в отношении рассматриваемой в данной главе авиационной системы A-7E показан на рис. 3.1.

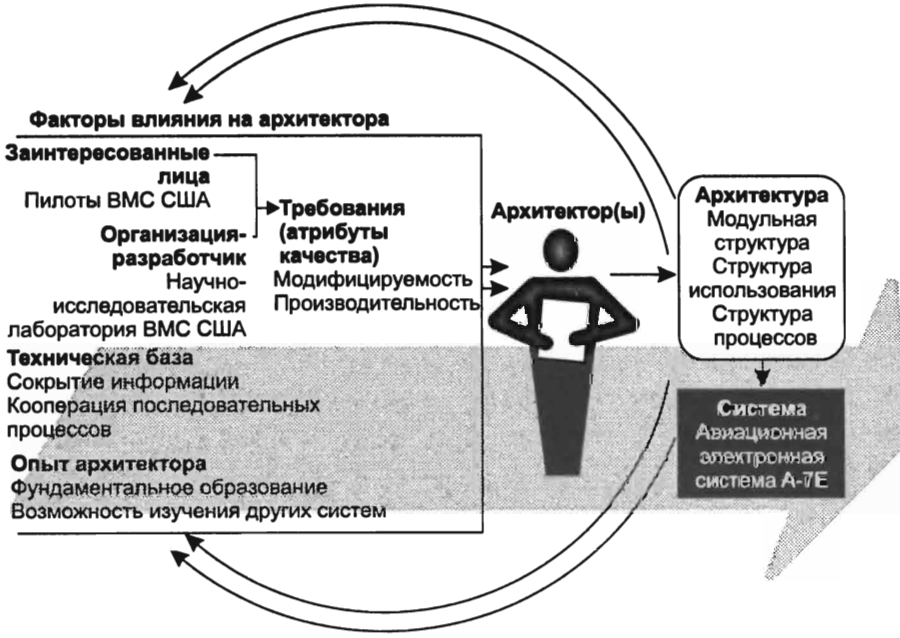


Рис. 3.1. Архитектурно-экономический цикл в применении к авиационным системам A-7E

Конструирование этой системы, предназначенной для тренировки пилотов самолета A-7E ВМС США, началось в 1977 году и финансировалось военно-морским министерством. В качестве организации-разработчика выступила рабочая группа программной инженерии при научно-исследовательской лаборатории ВМС США. Разработчики ставили целью проверить действенность отдельных стратегий программной разработки (речь в данном случае шла о сокрытии информации и кооперации последовательных процессов) в отношении высокопроизводительных встроенных систем реального времени.

Архитекторами системы оказались один из авторов настоящей книги и один из ведущих специалистов-теоретиков программной инженерии; отсутствие у них опыта работы в области авиационной электроники отчасти компенсировалось возможностью консультаций с экспертами и изучения существующих авиационных систем. Компилятора для целевой платформы в их распоряжении не было.

Начнем с описания прикладной области, предполагавшихся функций системы, атрибутов качества, без которых она не могла обойтись, и роли программного обеспечения в деле выполнения ее задач.

3.2. Требования и атрибуты качества

На рис. 3.2 изображен самолет A-7E Corsair II. Это одноместный, базирующийся на авианосце штурмовик, состоявший на вооружении ВМС США с 1960-х по 1980-е годы. Его предшественник — A-7C — был одним из первых в мире серийных самолетов, оборудованных бортовым компьютером, который помогал пилоту выполнять задачи, связанные с навигацией и «доставкой вооружения» (военный эвфемизм «атаки наземной цели»).

Рис. 3.2. A-7E Corsair II. Приводится с разрешения правообладателя — Squadron/Signal Publications, Inc.

Бортовой компьютер A-7E представляет собой компактную специализированную вычислительную машину производства компании IBM, для которой не существует ни одного компилятора; программировать, таким образом, можно только на языке ассемблера. Специальные регистры этой машины подключены к аналого-

цифровому и цифро-аналоговому преобразователям, при помощи которых данные принимаются и передаются примерно двум десяткам устройств авиационно-электронного комплекса.

Магистральной задачей программного обеспечения A-7E является считывание показаний датчиков и обновление данных на бортовых индикаторах, помогающих пилоту сбрасывать снаряды точно по выбранной цели. В отличие от многих современных авиационных систем программные средства A-7E не предназначены для непосредственного управления воздушным судном.

Ниже приводится перечень основных датчиков, с которых программное обеспечение считывает данные и работой которых оно управляет.

- Воздушный зонд замеряет атмосферное давление и воздушную скорость.
- РЛС переднего обзора, нацеливаемая под азимутом или углом возвышения и возвращающая прямую дальность указанной наземной точки.
- Доплеровская РЛС измеряет путевую скорость и угол сноса (разницу между направлением носа самолета и направлением его движения относительно земли).
- Инерциальная система измерений (inertial measurement set, IMS) сообщает ускорение по трем ортогональным осям. Программные средства должны оперативно считывать эти показания, путем их интегрирования по времени вычислять скорость, а затем, исходя из интегрирования скорости по времени, определять текущее положение воздушного судна в пространстве. Кроме того, программные средства обеспечивают совмещение и устраняют отклонения по осям, которые в целях точного соответствия системе координат самолета должны быть постоянно направлены на север, на восток и по вертикали соответственно.
- Интерфейс с инерциальной системой измерений авианосца, при помощи которой рассчитывается текущее положение самолета во время его нахождения на борту.
- Датчики, сообщающие о наполнении шести подкрыльевых бомбовых отсеков A-7E и типе содержащихся в нем орудий (всего их более 100 типов). В программной части хранятся крупные таблицы с параметрами всех типов вооружения; и на их основе в каждом конкретном случае определяется баллистическая траектория свободного падения.
- Радиолокационный высотомер, измеряющий расстояние до земли.

Бортовые индикаторные устройства, в отношении которых осуществляется программное управление, делятся на две группы: во-первых, это собственно индикаторы, а во-вторых, устройства, при помощи которых происходит взаимодействие пилота с программной частью. Содержание последней группы раскрывается в нижеследующем списке.

- Карта, на которой посредством пленки с задней подсветкой постоянно отображается текущее местоположение самолета. У пилота всегда есть выбор между двумя вариантами ориентации карты — ее верхняя оконечность может соответствовать либо текущему курсу, либо географическому северу.

- Проекционный бортовой индикатор — устройство, которое проецирует цифровые и иконографические данные на прозрачное стекло, расположенное между пилотом и лобовым стеклом. Поскольку положение головы пилота известно и неизменно, на этот дисплей выводится реальная информация — например, положение цели или линия, изображающая направление движения самолета.
- Клавишная панель и три небольших буквенно-цифровых дисплейных окна. С помощью клавишной панели пилот может получать цифровые данные примерно 100 различных видов. Группа переключателей на панели управления компьютером позволяет пилоту устанавливать режимы навигации и доставки вооружения.
- Ряд индикаторов и круговых шкал, а также звуковой сигнал.

Способов информирования программных средств о местонахождении наземной цели (или навигационной точки маршрута) в распоряжении пилота имеется несколько.

- Ввод ее широты и долготы с клавиатуры.
- Подводка координат цели под центральное перекрестье путем разворота карты рычагом управления; последующее «подтверждение» координат выполняется нажатием специальной кнопки на штурвале.
- Нацеливание на точку РЛС переднего обзора и подтверждение этой точки.
- Наложение специального символа с проекционного бортового индикатора на желаемую точку и ее подтверждение.

После выполнения одного из этих действий программное обеспечение выводит на проекционный бортовой индикатор навигационные данные (направление, расстояние, остаток времени полета) и направляющие подсказки, указывающие путь к намеченному местоположению.

Существует более двух десятков режимов навигации, отличия между которыми определяются степенью надежности тех или иных датчиков в текущих условиях. Прямых и косвенных программных способов определения текущей высоты полета по меньшей мере пять; помимо прочего, для этой цели используется тригонометрическая схема, в которой в качестве сторон треугольника выступают дальность и угол возвышения РЛС переднего обзора (рис. 3.3). Все режимы доставки вооружения, которых в общей сложности насчитывается более 20, предъявляют повышенные требования к проведению в реальном времени (25 раз в секунду) вычислений, обеспечивающих точность бомбометания A-7E.

В конце 1980-х годов модель A-7E Corsair была снята с вооружения. Впрочем, некоторые блоки электронных систем истребителей текущего поколения — в частности, проекционные бортовые индикаторы, режимы доставки вооружения и навигации — демонстрируют значительное сходство с ней.

Архитектура, которую мы представим в этой главе, относится не к первоначальной, а к модернизированной версии программного обеспечения, которую инженеры BMC создали на основе системы A-7E для демонстрации своих идей (см. врезку «О проекте A-7»). Среди атрибутов качества, которые было необходимо

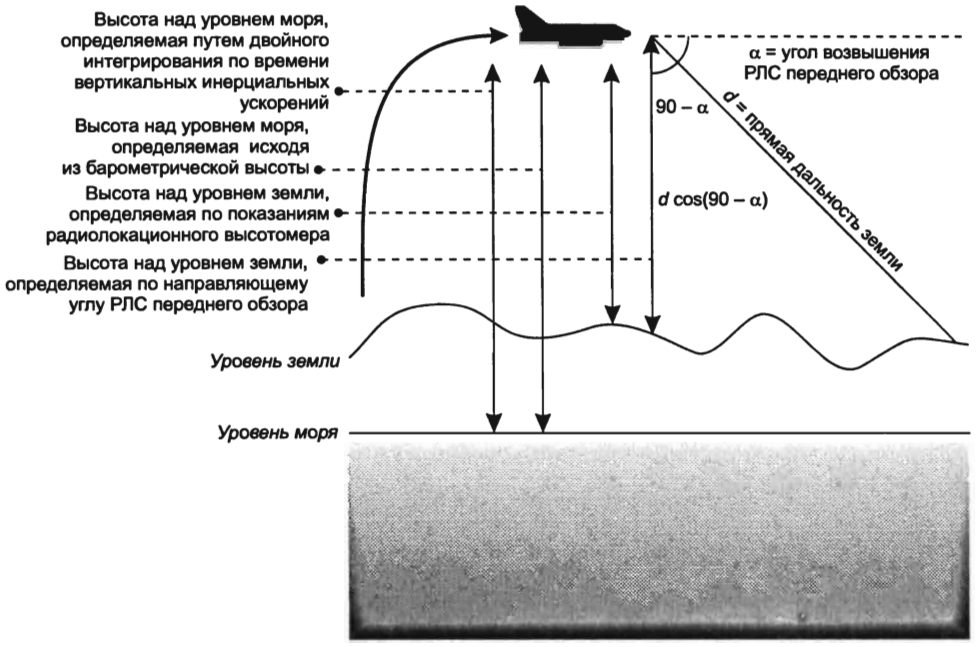


Рис. 3.3. Определение высоты полета A-7E

реализовать в этой новой системе, значились функционирование в реальном времени и модифицируемость в зависимости от вероятных изменений. Требования к производительности, в частности, касались количества обновлений изображений для дисплеев A-7E в секунду и скорости вычислений, связанных с доставкой вооружения. Под модифицируемостью подразумевалась замена вооружений, платформы и символики на дисплее, а также ввод новых данных с клавиатуры.

О ПРОЕКТЕ A-7

«К середине 1970-х годов специалистам научно-исследовательской лаборатории ВМС (Naval Research Laboratory, NRL) в Вашингтоне (Колумбия) стало ясно, что многие академические и лабораторные разработки в области компьютерных наук разработчиками программного обеспечения систем ВМС не используются». С этих слов начинается типичное описание Проекта по снижению стоимости программного обеспечения систем ВМС США (Navy's Software Cost Reduction, SCR), или, как его называют гораздо чаще, A-7. Далее в большинстве описаний говорилось, что специалисты лаборатории ВМС, поразмыслив, разработали перспективную высокоточную программу для ВМС (имеется в виду программное обеспечение для истребителей A-7E) с тем, чтобы впоследствии, обратившись к вышеупомянутой невостребованной технологии, провести ее модернизацию и повторную реализацию. Все эти декларации ставили целью доказать исключительную ценность данной технологии в контексте разработки реальных приложений.

Понять, о какой конкретно технологии идет речь, можно было только читая между строк, — в основном речь шла о проектной стратегии сокрытия информации. Ничего удивительного, если учесть, что идеологом проекта A-7 был не кто иной, как Дэвид Парнас (David Parnas) — первый исследователь, заговоривший о сокрытии информации как об одной из методик проектирования. Парнас хотел выяснить, насколько его идеи (как сокрытие информации, так и некоторые другие — скажем, кооперация последовательных процессов) применимы к системе с жесткими требованиями, серьезными ограничениями по потреблению памяти и ог-

раниченными временными ресурсами. Если же они для этого не годились, Парнасу нужно было знать, почему и что нужно сделать, чтобы исправить ситуацию. Демонстрации не совсем четко сформулированных методов на модельных задачах ему, конечно, было недостаточно. Для проекта A-7 планировалось составить законченную инженерную модель — документацию, проект, код, методологию, принципы, — которую, таким образом, можно было бы повторить; все эти данные предполагалось публиковать в общедоступных изданиях.

В 1977 году силами нескольких сотрудников-совместителей работа над проектом началась. Вскоре выбрали и демонстрационное приложение — программные средства A-7E. Руководствовались тем, что это система реального времени (то есть она должна полностью соответствовать требованиям по времени), она относится к категории встроенных (предусматривает взаимодействие с разного рода сложными аппаратными устройствами), она абсолютно надежна и к тому же ограничена необычайно малой емкостью памяти компьютера — 32 000 шестнадцатибитных слов. Итак, если бы новые методики выдержали испытание *этой* программой, они справились бы с чем угодно.

Первой появилась спецификация требований к программному обеспечению. Она не входила в первоначальные планы, но, когда Парнас попросил специалистов ВМС предоставить ему сводку требований к A-7, оказалось, что таковой не существует. Осознав, что ввести стандарт для тестирования и оценки результатов все-таки придется, программные инженеры из лаборатории ВМС, пусть и с неохотой, но принялись за документирование требований к программной части. На выходе получилась не только сама сводка требований, но и, что более важно, метод ее составления. Теперь по образцу SCR документируются многие встроенные программные системы реального времени.

Затем разработчики занялись проектированием интерфейсов для всех модулей. Таким вот образом несколько специалистов чуть ли не впервые реализовали метод объектного проектирования. Мало того — желая учесть любые последующие изменения, они сконструировали то, что сегодня называют доменной моделью. Намереваясь создать типовой проект, предусматривающий возможность многократного применения, они сконструировали эталонную архитектуру (см. главу 12). Так, в промежутках между изобретением новых методов программной инженерии, изучением предметной области авиационной электроники и написанием новаторских статей они занимались производством программных средств.

Реализация проекта началась с размещения сокращенного варианта программы, на примере которого планировалось доказать ее способность к генерации исполняемого кода; далее предполагалось разместить два более крупных варианта и, наконец, всю систему. Обратившись к структуре использования — одной из трех архитектурных структур, на которые мы обращаем внимание в нижеследующем конкретном примере, — они смогли быстро и без труда выделить эти подмножества исходя из своих потребностей. К тому моменту, когда второе из трех подмножеств было подготовлено, руководители проекта поняли, что все поставленные задачи выполнены, а учитывая малочисленность штата, ограниченность бюджета и недостаточность знаний в области авиационной электроники, доводить повторную реализацию до конца не имеет смысла. В 1987 году, после того как второе подмножество было успешно введено в строй, проект свернули. В рамках этого второго подмножества присутствовали элементы всех модулей второго уровня, а также полезная и довольно сложная функция навигации.

По итогам работы был сделан вывод о том, что стратегия сокрытия информации не просто подходит для встроенных систем реального времени, но во многих отношениях идеальна для применения в этом контексте. Особое внимание к интерфейсам модулей и их спецификациям исключило необходимость в проведении одного из этапов проекта — интеграции; ошибки, которые обычно ассоциируются с этим этапом, просто не допускались. Конечный программный продукт укладывался во временные рамки, однако в том, что касалось эффективности потребления памяти, заметно уступал ручному коду ассемблера, на разработку которого ушли годы. Надо надеяться, что чем дальше, тем меньшее значение будет иметь фактор эффективности использования памяти — по крайней мере, в современных условиях он не так существен, как это было в 1977 году.

Архитектура, которую мы приводим в качестве конкретного примера, относится к готовому проекту, из которого в 1987 году было получено последнее подмножество. У нас нет причин сомневаться в том, что на основе этой архитектуры в ее неизменном виде можно было бы провести полную реализацию системы. Как бы то ни было, она демонстрирует повышенное

внимание к различным архитектурным структурам, или представлениям, обеспечивающим достижение определенных задач; именно в этом ключе мы и намерены ее рассмотреть.

Почему же, если с момента работы над проектом A-7E прошло так много времени, он до сих пор представляет интерес? Дело в том, что в нем реализовано два важных принципа. Первый утверждает жизнеспособность и практичность сокрытия информации как стратегии проектирования — этот принцип сообществом соблюдается. Согласно второму, тщательная разработка разных архитектурных структур приносит свои плоды в виде реализации атрибутов качества. Этот принцип усвоен не слишком хорошо, и поэтому, в контексте наблюдаемого на сегодняшний день интереса к программной архитектуре и руководствуясь правилом «повторение — мать учения», мы возьмемся изложить его еще раз.

— PCC

3.3. Архитектура авиационной электронной системы A-7E

Архитектура авиационной системы A-7E основывается на трех архитектурных структурах, которые мы рассматривали в главе 2:

- декомпозиция (структура модулей);
- варианты использования (структура модулей);
- процесс (структура компонентов и соединителей).

Каждый из них мы обсудим в отдельности.

Структура декомпозиции

Во всех случаях, кроме тех, когда масштаб программы предполагает возможность ее производства силами одного программиста, задача разделяется на несколько блоков, каждый из которых можно реализовывать отдельно от других; в связи с этим решаются вопросы взаимодействия этих модулей. В качестве блока структуры декомпозиции, естественно, выступает модуль. Модуль определяет группу процедур — как публичных, так и приватных, а также ряд приватных структур данных. В качестве отношений между модулями из структуры декомпозиции применяются: «является подмодулем...» или «пользуется секретом совместно с...».

Вплоть до 1977 года основное требование, предъявлявшееся к встроенным (равно как и к большинству других) системам, касалось их производительности. Проектировщики системы A-7E поставили целью уравновесить производительность и модифицируемость и доказать, что одно другому не мешает.

Информационная закрытость

Декомпозиция модулей A-7E базируется на информационной закрытости. Информационная закрытость — архитектурная тактика (к ее обсуждению мы вернемся в главе 5) — обеспечивает инкапсуляцию тех составляющих системы, которые с наибольшей степенью вероятности будут изменяться независимо в каждом конкретном модуле. Интерфейс модуля, с другой стороны, раскрывает только те элементы системы, вероятность изменения которых невелика; элементы, скрываемые интерфейсом модуля, называются его секретами.

Приведем пример. Если датчик высоты, установленный в самолете, предполагается заменить до завершения жизненного цикла авиационной программы, тогда, согласно принципу информационной закрытости, все детали, касающиеся взаимодействия с этим устройством, становятся секретом отдельного модуля. Интерфейс этого модуля содержит абстракцию датчика — скажем, программу, возвращающую последние показания датчика (дело в том, что эта возможность характеризует все без исключения датчики, которые впоследствии могут быть установлены взамен текущего). В случае замены датчика корректируются только внутренние элементы соответствующего модуля; все остальные составляющие программного обеспечения остаются без изменений.

В соответствии с принципом информационной закрытости взаимодействие модулей осуществляется исключительно через предопределенный набор публичных средств — через их *интерфейсы* (interfaces). В составе любого модуля есть ряд *процедур доступа* (access procedures), к которым могут обращаться все остальные модули. Процедурами доступа средства взаимодействия с содержащейся в конкретном модуле информацией исчерпываются.

Основное отличие этой стратегии от объектного проектирования заключается в следующем. Прообразами программных объектов являются физические объекты приложения, а также эмпирические знания о системе. Модули же сокрытия информации формируются по результатам каталогизации изменений, которым в течение жизненного цикла системы может с определенной степенью вероятности подвергнуться ее программное обеспечение.

Модуль либо делится на ряд подмодулей, либо является единым блоком реализации. В первом случае предусматривается описание по его субструктуре. Декомпозиция на подмодули и их проектирование продолжаются до тех пор, пока подмодули не станут достаточно малыми — такими, чтобы к последующей работе над ними можно было привлекать новых программистов.

Цели, для достижения которых проводится декомпозиция модулей, можно сформулировать следующим образом.

- Структура любого модуля должна быть простой и полностью понятной.
- Для изменения реализации отдельного модуля не обязательно иметь представление о реализациях других модулей; кроме того, это действие не должно влиять на их поведение.
- Простота внесения изменений должна коррелировать с их вероятностью; наиболее вероятные изменения должны вноситься без корректировки интерфейсов модулей; менее вероятные изменения могут предусматривать корректировку интерфейсов, но только в отношении небольших и не слишком востребованных модулей. Перестройкой интерфейсов активно применяемых модулей могут сопровождаться лишь наименее вероятные изменения.
- Крупные изменения должны вноситься в программное обеспечение в виде независимых изменений отдельных модулей (другими словами, если речь не идет о переработке интерфейсов, необходимости во взаимодействии между программистами, отвечающими за замену отдельных модулей, быть не должно). Если интерфейсы модулей не модифицировались, этого должно быть достаточно для запуска и тестирования любых комбинаций старых и новых версий модулей.

Документацию по структуре декомпозиции иногда называют *руководством по модулям*. Изложение в нем проектных решений, инкапсулированных в отдельных модулях, определяет их обязанности. Документация нужна для того, чтобы исключить дублирование и пропуски, провести в жизнь принцип разделения задач, а самое главное — помочь специалистам по сопровождению в деле выявления модулей, на которые следует обратить внимание при появлении того или иного сообщения о неисправности или запроса на внесение изменений.

Руководство по модулям, во-первых, содержит сведения о критериях, согласно которым на модули возлагаются обязанности, а во-вторых, определенным образом комментируя организацию модулей в рамках системы, облегчает поиск информации по определенной проблеме (устраняет избыточность поиска). Такая систематизация, согласно древовидному характеру структуры декомпозиции, способствует разделению системы на немногочисленные модули, разбиение которых (в целях уменьшения размера) ведется единообразно. Каждый узел, не являющийся листом дерева, отображает узел, состоящий из модулей, представляемых его потомками. Руководство по модулям не содержит информации об отношениях между модулями в период прогона — оно ничего не сообщает о том, как происходит взаимодействие модулей исполняемой системы; с другой стороны, оно дает представление об отношениях между блоками реализации в период проектирования проекта.

Проводить этот принцип в жизнь не так уж просто. Суть его заключается в том, что за счет предвидения вероятных изменений снижается ожидаемая стоимость программного обеспечения. Основой для проведения подобного анализа является опыт, экспертиза в прикладной области, а также понимание технологий производства аппаратного и программного обеспечения. Поскольку наличие подобного опыта у проектировщика сомнительно, формальные процедуры оценки строятся таким образом, чтобы в ходе их проведения можно было задействовать опыт сторонних специалистов. Роль модульной структуры в архитектуре системы A-7E сформулирована в табл. 3.2.

Таблица 3.2. Механизм выполнения задач по качеству в рамках структуры декомпозиции модулей A-7E

Задача	Как выполняется
Простота замены вооружения, платформы, символики, входных данных	Сокрытие информации
Представление об ожидаемых изменениях	Формальная процедура оценки с привлечением опыта специалистов в данной предметной области
Распределение задач между рабочими группами с расчетом на сведение их взаимодействия к минимуму	Модули систематизируются в рамках иерархии; перед каждой рабочей группой ставится задача по производству одного модуля второго уровня и всех его потомков

Структура декомпозиции на модули A-7E

В качестве описания структуры декомпозиции модулей A-7E и примера документации модульной структуры мы намерены привести ряд отрывков из руко-

водства по модулям программной части A-7E. Описание декомпозиционного дерева начинается с трех модулей высшего уровня. Связано это с тем, что, по некоторым наблюдением, изменения в системах, подобных A-7E, обычно происходят из трех источников: из области аппаратного обеспечения, с которым взаимодействует программная часть, из области предполагаемого внешнего поведения системы и, наконец, из той области, в которой решения принимает программный проектировщик проекта.

Модуль сокрытия аппаратного обеспечения. Модуль сокрытия аппаратного обеспечения (Hardware-Hiding Module) содержит процедуры, которые модифицируются в случае замены старых аппаратных блоков новыми, функционально аналогичными, но обладающими оригинальным аппаратно-программным интерфейсом. Этот модуль реализует виртуальное оборудование (virtual hardware) — набор абстрактных устройств, к которым обращаются все остальные программные элементы. Первичными секретами здесь являются аппаратно-программные интерфейсы. В качестве вторичных секретов выступают структуры данных и алгоритмов, участвующие в реализации виртуального оборудования. Одним из подмодулей модуля сокрытия аппаратного обеспечения является модуль расширения компьютера (Extended Computer Module), скрывающий элементы процессора.

Модуль сокрытия поведения. Процедуры модуля сокрытия поведения (Behavior-Hiding Module) модифицируются в тех случаях, когда изменение требований влечет за собой корректировку предполагаемого поведения. Эти требования являются первичным секретом данного модуля. Его процедуры вычисляют значения, которые впоследствии отправляются виртуальным устройствам вывода, реализуемым модулем сокрытия аппаратного обеспечения.

Модуль программных решений. Модуль программных решений (Software Decision Module) скрывает те программно-проектные решения, которые основываются на математических теоремах, физических фактах и факторах программирования наподобие алгоритмической эффективности и точности. Описание секретов этого модуля в сводке требований отсутствует. Основное его отличие от других модулей заключается в том, что в данном случае решения о секретах и интерфейсах принимаются программными проектировщиками. Следовательно, внесение изменений здесь обычно обусловливается не внешними факторами, а желанием повысить производительность и точность.

Далее в руководстве по модулям приводится описание арбитража конфликтов между этими категориями (например: к чему относится требуемый алгоритм — к поведению или к области программных решений?) с помощью комплексной и однозначной спецификации требований. Затем излагается декомпозиция второго уровня. Нижеследующий отрывок взят из описания декомпозиции модуля программных решений.

Модуль прикладных типов данных (Application Data Type Module). Этот модуль дополняет типы данных расширенного модуля компьютера типами данных, характерными для авиационных приложений и не требующими машинозависимой реализации. Среди них — расстояние (тип данных, применяемый для определения высоты), временные интервалы и углы (используются для вычисления широты и долготы). Реализуются они через элементарные числовые типы данных модуля расширения компьютера, а их переменные применяются так, как будто соответствующие типы встроены в упомянутый модуль.

В качестве секретов модуля прикладных типов данных выступают варианты представления данных переменными и процедурами, реализующими операции с этими переменными. Единицы измерения (футы, секунды, радианы и др.) входят в состав представлений и, соответственно, скрываются. Для отдельных случаев в модулях предусмотрены операции преобразования, которые подготавливают или принимают фактические значения, переводя их в указанные единицы измерения.

Модуль банка данных (Data Banker Module). В большинстве случаев данные производятся одним модулем, а потребляются другим. При этом потребители обычно требуют предоставить

им самые свежие значения. Период времени, по прошествии которого данное значение следует пересчитывать, определяется свойствами его потребителя (требованиями по точности) и производителя (стоимостью вычислений и частотой изменения значений). В этих условиях модуль банка данных исполняет роль посредника — именно он определяет, когда следует проводить подсчет новых значений.

Модуль банка данных получает значения от процедур-производителей; процедуры-потребители, в свою очередь, получают данные от процедур доступа модуля банка данных. Знать время обновления сохраненного значения для регистрации производителя и потребителя конкретного элемента данных не обязательно. В большинстве случаев изменения политики обновлений не приводят к модификации производителей или потребителей.

Модуль банка данных содержит все значения, которые могут что-либо сообщить о внутреннем состоянии модуля или о состоянии воздушного судна. Кроме того, он сигнализирует обо всех событиях, которые предполагают внесение изменений в сохраненные значения. Модуль банка данных необходим во всех ситуациях, при которых в ролях потребителя и производителя выступают разные модули, — пусть они даже оба входят в более крупный модуль на правах подмодулей. Модуль банка не используется только в двух случаях: во-первых, если потребителям требуются конкретные члены последовательности вычисленных производителем значений и, во-вторых, если вычисленное значение представляет собой не что иное, как функцию значений входных параметров, предоставленных процедуре–производителю, — например, $\sin(x)$[1].

При выборе методик обновления следует учитывать предъявляемые потребителями требования по точности, частоте повторного вычисления значения, максимальному периоду ожидания, который они могут себе позволить, частоте изменения значения, а также стоимости вычисления нового значения. Эти сведения, наряду с прочими, входят в спецификацию, которая предоставляется конструктору модуля банка данных.

Модуль фильтрации поведения (Filter Behavior Module) содержит цифровые модели физических фильтров. С их помощью сторонние процедуры фильтруют потенциально зашумленные данные. Первичными секретами этого модуля являются модели вычисления значений, которые строятся на основе выборочных значений и оценки погрешности. Вторичные секреты — это вычислительные алгоритмы и структуры данных, с помощью которых реализуются упомянутые модели.

Модуль физических моделей (Physical Models Module). Некоторые количественные величины, которые программе требуется определить, вычисляются не напрямую, а посредством математических моделей на основе наблюдаемых значений. Пример — время, за которое баллистический снаряд долетает до земли. В качестве первичных секретов модуля физических моделей выступают математические модели; вторичные секреты — это их вычислительные реализации.

Модуль обслуживающих программ (Software Utility Module) содержит служебные программы, которые в противном случае пришлось бы писать сразу нескольким программистам. Среди них — математические функции, диспетчер ресурсов, а также процедуры, которые сигнализируют о завершении модулями инициализации при включении питания. Секретами этого модуля являются структуры данных и алгоритмы реализации упомянутых процедур.

Модуль генерации системы (System Generation Module). Первичными секретами модуля генерации системы считаются решения, которые откладываются вплоть до периода генерации системы. Это, во-первых, значения параметров генерации системы, а во-вторых, альтернативные реализации модуля. В качестве вторичных секретов модуля генерации системы выступают метод генерации машинно-исполняемого кода и представление отложенных решений. Процедуры этого модуля запускаются не на бортовом компьютере, а на машине, применяемой для генерации его кода.

Декомпозиция в руководстве по модулям доводится до третьего (а иногда даже до четвертого) уровня, но здесь мы в эти детали углубляться не намерены. Тре-

[1] Модуль банка данных демонстрирует пример использования архитектурного образца классной доски (см. главу 5 «Реализация качества»).

Рис. 3.4. Представление декомпозиции модулей программной архитектуры системы A-7E

тий уровень структуры декомпозиции для архитектуры A-7E приводится лишь на рис. 3.4. Обратите внимание на идентичность имен некоторых модулей интерфейсов устройств (Device Interface) и управления функциями (Function Driver).

Различие между ними заключается в следующем. Модули интерфейсов устройств программируются с учетом знаний о взаимодействии с соответствующими устройствами программной части; при программировании модулей управления функциями уже известны значения, которые должны быть вычислены и отправлены этим устройствам. Здесь действует новое архитектурное отношение, о котором мы вскоре поговорим, — связано оно с тем, как в рамках этих модулей происходит взаимодействие программных средств, позволяющее им выполнять поставленные задачи.

И все же представление декомпозиции модулей этим не исчерпывается. Как вы помните, в главе 2, излагая определение архитектуры, мы говорили, что в ее состав входят поведенческие спецификации всех элементов. Переносимость и способность к взаимодействию — это те характеристики, которых можно добиться только посредством тщательно спроектированных, независимых от конкретного языка интерфейсов. Каждому модулю должен быть назначен собственный интерфейс. Документация, связанная с программными интерфейсами, рассматривается в главе 9.

В предыдущей главе мы обращали ваше внимание на функцию архитектуры как детального плана проекта разработки и собственно программной системы. В случае с архитектурой системы A-7E структуре декомпозиции модулей второго уровня была уготована особая роль — на нее как на первоисточник ссылались проектная документация, оперативные конфигурационные документы, планы тестирования, группы программистов; на ее основе строились процедуры оценки, расписывался график проекта и рассчитывались его контрольные точки.

Структура использования

Вторая структура архитектуры A-7E — структура использования — интересует нас никак не меньше структуры декомпозиции. Последняя не содержит никаких данных об исполнении программного продукта для периода прогона; о том, как в этот период проходит взаимодействие любых двух модулей, можно делать лишь более или менее обоснованные предположения; авторитетные сведения на эту тему в декомпозиции модулей отсутствуют. Подобного рода сведения о взаимодействии в рамках программного изделия содержатся в структуре использования.

Отношение использования

Концептуально структура использования основывается на отношении использования. Словосочетание «процедура A *использует* процедуру B» расшифровывается так: для того чтобы процедура A соответствовала предъявленным к ней требованиям, необходимо наличие правильно функционирующей процедуры B. На практике это очень напоминает отношение вызова (хотя об их тождестве речи не идет). Как правило, если речь идет об использовании процедурой A процедуры B, то первая вызывает вторую. Впрочем, в двух нижеприведенных случаях между вызовами и использованием наблюдаются существенные различия.

1. Согласно спецификации процедуры A, она должна вызывать процедуру B, однако последующие вычисления, произведенные процедурой A, не зависят от операций B. Для обеспечения работоспособности процедуры A требу-

ется наличие процедуры B, но ее правильность не является обязательной. При этом A вызывает B, но не использует ее. Процедура B, к примеру, может быть обработчиком ошибок.

2. Процедура B выполняет свою функцию, не сталкиваясь с вызовом со стороны процедуры A; впрочем, A использует ее результаты — в качестве таковых может выступать, например, обновленное хранилище данных, формируемое процедурой B. Кроме того, B может быть обработчиком прерываний, существование и корректное функционирование которого A принимает в виде допущения. Таким образом, A использует, но не вызывает B.

Отношение использования позволяет быстро идентифицировать функциональные подмножества. Из того, что процедура A находится в определенном подмножестве, можно сделать вывод, что все используемые ею процедуры следует искать там же. Такое подмножество определяется транзитивным замыканием отношения использования. Снимая требование о наполнении каждого отдельного подмножества целостной системой, оно способствует конструированию и организации данной структуры. В результате программисты получают структуру разрешения использования (allowed-to-use structure). По завершении реализации фактические варианты использования можно каталогизировать.

Первичной единицей структуры использования (или, иначе, структуры разрешения использования) является процедура доступа. Определяется она перечислением всех процедур, которыми могут пользоваться все остальные процедуры (из этого можно сделать вывод о том, в каких случаях использование одной процедурой другой запрещено).

На практике, несмотря на роль процедуры как единицы структуры использования, ограничения по использованию могут распространяться на все процедуры модуля. Следовательно, в структуре использования может фигурировать имя модуля, и в таком случае его следует рассматривать как сокращенное обозначение всех присутствующих в этом модуле процедур доступа.

Концептуальное описание структуры (разрешения) использования выглядит как бинарная матрица; в ее строках и столбцах фиксируются все имеющиеся в системе процедуры. Так, элемент (m,n) означает, что процедура m использует процедуру n (или имеет разрешение на ее использование). Поскольку на деле такая запись оказывается слишком громоздкой, правила принято не разбивать на отдельные процедуры, а сокращать до целых модулей.

Роль структуры использования в программной архитектуре A-7E иллюстрируется табл. 3.3.

Структура использования A-7E

Следует иметь в виду, что при документировании структуры использования в спецификации фиксируются лишь отношения разрешения использования; фактические варианты использования выводятся по результатам реализации. Спецификация разрешения использования для архитектуры A-7E предстает в виде семистраничной таблицы, небольшой отрывок которой приводится в табл. 3.4. Сокращения из двух символов в начале каждой новой строки обозначают модули второго уровня. Справа от точки размещаются имена подмодулей, о которых мы в этой главе практически не упоминаем.

Таблица 3.3. Роль структуры использования в контексте реализации задач по качеству системы A-7E

Задача	Как выполняется
Инкрементное конструирование и тестирование системных функций	Программистам предоставляется структура «разрешения использования», которая регламентирует использование одних процедур другими
Проектирование в расчете на межплатформенную переносимость	Ограничение количества процедур, обращающихся непосредственно к платформе
Создание руководства по использованию приемлемого размера	Установление (там, где это имеет смысл) отношений использования между модулями

Таблица 3.4. Отрывок спецификации разрешения использования A-7E

Использование процедур: процедура в составе модуля...	... разрешается пользоваться любой процедурой в составе модуля...
EC: модуль расширения компьютера	–
DI: модуль интерфейса устройства	EC.DATA, EC.PGM, EC.IO, EC.PAR, AT.NUM, AT.STE, SU
ADC: вычислитель воздушных сигналов	PM.ECM
IMS: инерциальная система измерений	PM.ACM
FD: модуль управления функциями	EC.DATA, EC.PAR, EC.PGM, AT.NUM, AT.STE, SU, DB.SS.MODE, DB.SS.PNL.INPUT, DB.SS.SYSVAL, DB.DI
ADC: функции вычислителя воздушных сигналов	DB.DI.ADC, DI.ADC, FB
IMS: функции инерциальной системы измерений	DB.DI.IMS, DI.IMS
PNL: функции панели (управления)	EC.IO, DB.SS.PNL.CONFIG, SS.PNL.FORMAT, DI.ADC, DI.IMS, DI.PMDS, DI.PNL
SS: модуль совместно используемых служб	EC.DATA, EC.PGM, EC.PAR, AT.NUM, AT.STE, SU
PNL: вспомогательный модуль ввода-вывода на панели	DB.SS.MODE, DB.DI.PNL, DB.DI.SWB, SS.PNL.CONFIG, DI.PNL
AT: модуль прикладных типов данных	EC.DATA, EC.PGM
NUM: числовые типы данных	Дополнительных подмодулей нет
STE: события смены состояний	EC.PAR

Посмотрим, какой в результате получается образец:

- Процедурам модуля расширения компьютера не разрешается обращаться к процедурам других модулей; с другой стороны, все остальные модули могут использовать некоторые его элементы.
- Процедуры модуля прикладных типов данных могут обращаться только к процедурам расширенного модуля компьютера.
- Процедуры модуля интерфейса устройства (по крайней мере, те из них, которые представлены в табл. 3.4) могут обращаться к процедурам модуля расширения компьютера, модуля прикладных типов данных и модуля физических моделей.

- Процедуры модулей управления функциями и совместно используемых служб могут обращаться к процедурам модуля банка данных, модуля расширения компьютера, модулей прикладных типов данных и интерфейса устройства.
- Обращение к процедурам модуля управления функциями со стороны любых других процедур недопустимо.
- Обращаться к процедурам модуля совместно используемых служб имеют право только процедуры модуля управления функциями.

Итак, перед нами *многоуровневая* система. На нижнем уровне находится модуль расширения компьютера, чуть выше — модуль прикладных типов данных. Совместно они образуют виртуальную машину, разрешающую любой процедуре, расположенной на любом данном уровне, использование процедур, находящихся на том же и на всех нижележащих уровнях.

На верхних уровнях находятся модули управления функциями и совместно используемых служб; для исполнения своих функций они могут обращаться к любым средствам системы. В средней части иерархии уровней расположены модули физических моделей и фильтрации поведения, а также модуль банка данных. Модуль обслуживающих программ, занимающий особое, параллельное по отношению к этой структуре, положение, для выполнения своих задач имеет право обращаться к любым модулям, за исключением модуля управления функциями.

Многоуровневый архитектурный образец, имеющий широкое распространение, будет довольно часто встречаться в наших конкретных примерах. Наличие иерархии уровней является следствием применения структуры использования; в то же время уровни, не содержащие сведений о возможных подмножествах, не способны заменить эту структуру. Именно в этом суть структуры использования — *конкретный* модуль управления функциями может обращаться к *конкретному* набору операций, содержащихся в банке данных, модулях совместно используемых служб, интерфейсов устройств, прикладных типов данных и в модуле расширения компьютера. Совместно используемые службы, в свою очередь, имеют право обращаться к заданному набору процедур более низкого уровня, и так далее по нисходящей. Произведенный таким способом набор процедур в целом образует подмножество.

Структура разрешения использования дает представление о способе взаимодействия процедур модулей в период прогона — взаимодействия, посредством которого они выполняют свои задачи. Любая процедура модуля управления функциями управляет выходным значением определенного выходного устройства — например, положением отображаемого символа. Обобщенно процедуры управления функциями можно охарактеризовать так: они (при помощи процедур банка данных) забирают у производителей данные, применяют правила подсчета корректного выходного значения, а затем путем вызова процедуры интерфейса соответствующего устройства отправляют этому устройству конечный результат. Данные поступают из нескольких источников.

- Процедуры интерфейсов устройств сообщают данные о состоянии окружения, с которым взаимодействует данное программное средство.

- Процедуры физических моделей, ответственные за вычисление прогнозирующих показателей, сообщают сведения о внешнем мире (например, на основе текущего положения и скорости самолета рассчитывают координаты цели, которую мог бы поразить снаряд, будь он выпущен в данный момент).
- Процедуры совместно используемых служб сообщают о текущем режиме, достоверности текущих показаний датчиков и инициированных пилотом операций на панели управления.

По результатам проектирования структуры разрешения использования конструкторам становится известно, о каких интерфейсах им следует собрать информацию, чтобы выполнить свои задачи. Документирование структуры фактического использования и получение подмножеств начинается после завершения реализации. Способность к развертыванию отдельных подмножеств системы является одним из основных элементов эволюционного жизненного цикла поставки (Evolutionary Delivery Life Cycle — см. главу 7 «Создание архитектуры»). Как правило, лучший способ сделать хорошую мину при плохой игре, когда бюджет сокращается (или перерасходуется), а сроки горят, — это подготовить подмножество. В таких ситуациях правомерно утверждать, что при условии тщательного проектирования структуры использования можно было бы поставить больше подмножеств (все же лучше, чем ничего!).

Структура процессов

Третьей из ряда важнейших для системы A-7E архитектурных структур является структура процессов. Модуль расширения компьютера содержит виртуальный интерфейс программирования с поддержкой мультипроцессорной обработки, хотя собственно бортовой компьютер A-7E такой возможности не имеет. Предполагалось, что в определенный момент его заменят мультипроцессорным компьютером. Исходя из этого предположения, программное обеспечение было реализовано в виде набора сотрудничающих последовательных процессов, которые путем синхронизации (для взаимодействия друг с другом) получали возможность обращаться к совместно используемым ресурсам. При помощи неоперативного (предпрогонного) планирования этот набор удалось сформировать как одиночный исполняемый поток, который впоследствии должен был загружаться на базовый компьютер.

Процессом называется набор программных шагов, повторяющихся в ответ на запускающее событие или временное ограничение. У него собственный поток управления, а в ожидании события он сам себя приостанавливает (обычно для этого вызывается одна из программ сигнализации о событиях интерфейса данного модуля).

Процессы в системе A-7E создаются с двумя целями. Во-первых, с их помощью драйверы функций вычисляют для авиационной программной системы выходные значения. Исполняться они должны либо с определенной периодичностью (например, для беспрерывного обновления позиции символов на проекционном бортовом индикаторе), либо в ответ на какое-то запускающее событие (например, после нажатия пилотом кнопки сброса снаряда). Реализовывать эти дей-

ствия в виде процессов вполне естественно. Обобщенно структуру процессов управления функциями можно представить следующим образом:

- Периодический процесс: запускается каждые 40 мс.
  - ◊ В целях сбора всех значимых входных значений вызывает процедуры доступа других модулей.
  - ◊ Вычисляет конечное выходное значение.
  - ◊ Для отправки выходного значения окружению вызывает процедуру соответствующего модуля интерфейса устройства.
- Периодический процесс завершается.
- Процесс по требованию.
  - ◊ Ожидание запускающего события.
  - ◊ Процесс вычисляет конечное выходное значение.
  - ◊ С целью запуска действия в окружении процесс запускает соответствующую процедуру модуля интерфейса устройства.
- Процесс по требованию завершается.

Несколько реже процессы запускаются в целях реализации отдельных процедур доступа. Если издержки вычисления значения, возвращаемого процедурой доступа, оказываются непозволительно высокими, то, для того чтобы выдержать временные рамки, программист может организовать непрерывное фоновое вычисление значений и при вызове процедуры доступа возвращать последнее из них. К примеру, это может выглядеть так:

- Процесс: запускается каждые 100 мс.
  - ◊ Для подсчета значения собирает входные данные.
  - ◊ Подсчитывает значение.
  - ◊ Сохраняет его в переменной `most_recent`.
- Процесс завершается.
- Процедура `get_value(pl)`

```
pl := most_recent.
return
```

- Процедура завершается.

Итак, структура процессов состоит из набора программных процессов. Характерное для нее отношение «синхронизируется с...» основывается на событиях, о которых одни процессы сигнализируют, а другие этого ожидают. На этом отношении, в свою очередь, базируются операции планирования, и в том числе — предотвращение взаимоблокировки.

Методики неоперативного планирования, задействованные в программном обеспечении A-7E, мы обсуждать не будем; сказать о них нужно лишь следующее: во-первых, они позволяют избежать непроизводительных издержек, свойственных оперативному планированию; во-вторых, они не могут существовать без информации, содержащейся в структуре процессов. Последняя также предусматривает оригинальный прием оптимизации — путем слияния двух, по большому счету несвязанных, процессов планирование во многих ситуациях упрощается,

а издержки контекстного переключения, которые имеют место при приостановке одного процесса и возобновлении другого, сводятся на нет. Эта методика, применяемая автоматически в период конструирования системы, скрыта от программистов. Роль структуры процессов в архитектуре A-7E сформулирована в табл. 3.5.

Таблица 3.5. Роль структуры процессов в контексте реализации задач по качеству системы A-7E

Задача	Как выполняется
Отображение входных данных на выходные	Любой процесс реализуется в виде цикла, который отсчитывает, вводит, вычисляет и представляет выходные данные
Соответствие ограничениям реального времени	Процесс идентифицируется через свою структуру, после чего выполняется неоперативное планирование
Безотлагательное предоставление результатов продолжительных вычислений	Вычисления проводятся в фоновом режиме, и по запросу возвращается самое свежее из них

Структура процессов появилась последней — уже после проектирования остальных структур. Процедуры модулей управления функциями были реализованы в виде процессов. Другие процессы в фоновом режиме выполняли продолжительные вычисления, обеспечивая готовность соответствующих значений.

В структуре процессов зафиксирована информация двух видов. Первый вид — это документация по процедурам, включенным в тела процессов. Она дает представление о проходящих через систему потоках и сообщает конструкторам перечень процедур, требующих реентерабельного кодирования (кодирования в расчете на способность одновременного ведения нескольких потоков управления), — для этого применяются защищенные хранилища данных или взаимное исключение. Кроме того, эта информация позволила проектировщикам понять, какие процедуры впоследствии будут запускаться чаще других, и, соответственно, определить области, перспективные в плане проведения оптимизации.

Второй вид информации в рамках структуры процессов предполагает документирование процессов (или последовательных сегментов потоков процесса), которые не могут исполняться одновременно. На самом деле области взаимного исключения окончательно фиксируются только после завершения кодирования процессов; и все же, исходя из выявленных на раннем этапе отношений «исключения» между объектами, участники группы планирования могут установить отдельные количественные требования к неоперативному планировщику и приступить к планированию в областях, в которых автоматизация должна была принести наибольший эффект.

УСПЕХ ИЛИ ПРОВАЛ?

В своей колонке редактора The Journal of Systems and Software за ноябрь 1998 года [Glass 98] Боб Гласс (Bob Glass) высказывает мнение, что проект A-7E в конечном итоге провалился, поскольку рассмотренная в этой главе программная система так никогда и не заработала. Я очень уважаю Боба как личность и как специалиста, но в данном случае, уверен, он ошибается, пытаясь оценивать исследовательскую систему по коммерческим стандартам.

Что я имею в виду? У научного и предпринимательского сообществ разные культуры и разные критерии успеха. Взять хотя бы то, как представители этих сообществ «подают» себя клиентам. Предприниматели делают упор на надежность поставок, соблюдение временных

и финансовых ограничений. Вполне законными были бы претензии покупателя, если бы заказанный им автомобиль доставили в автосалон позже оговоренного срока, за более высокую цену и с нежелательными характеристиками.

Исследователи «выставляют на продажу» свои концепции. Так, в любом научно-исследовательском проекте описывается, каким образом в случае открытия финансирования этот проект изменит окружающий мир. Если окажется, что аналог результата исследований можно купить в соседнем супермаркете, у спонсора будут все основания выказывать недовольство. Обычно если спонсор видит, что выработанные в результате проведенных исследований идеи действительно способны в каком-то отношении изменить мир, он остается доволен.

Пусть эти характеристики немного шаблонны, но они, по большому счету, довольно точны. Потребителям коммерческих изделий часто нужны инновации. Заказчики научных исследований почти всегда хотят получения результатов. Представители обоих сообществ, стремясь обеспечить заключение контрактов, иногда прибегают к заведомо невыполнимым обещаниям. И тем не менее, по своей сути, представленные характеристики верны.

Назначение описанного в этой главе проекта A-7E состояло в том, чтобы продемонстрировать всем скептикам жизнеспособность «объектно-ориентированных методик» (правда, назывались они в то время по-другому) применительно к конструированию высокопроизводительных программных систем реального времени. Это — задача, поставленная перед исследовательским проектом. Требовалось изменить тогдашнее видение мира. Успех программы по снижению стоимости программного обеспечения систем ВМС США (разработка системы A-7E была его составной частью) с научной точки зрения не подлежит сомнению — доказательством тому сотни отзывов в профильных изданиях. Такую реакцию можно воспринимать как одобрение революционных по тем временам принципов инкапсуляции и сокрытия информации.

Итак, с коммерческой позиции система A-7E потерпела неудачу, но по исследовательским меркам — это успех. Возвращаясь к утверждениям Боба Гласса, вопрос нужно поставить следующим образом: получило ли военно-морское ведомство тот результат, на который рассчитывало? Ответ зависит от того, о чем изначально шла речь: о производственной системе или о научной работе? Поскольку разработки проводились в научно-исследовательской лаборатории ВМС США, надо думать, что проект A-7E следует оценивать по стандартам научного сообщества.

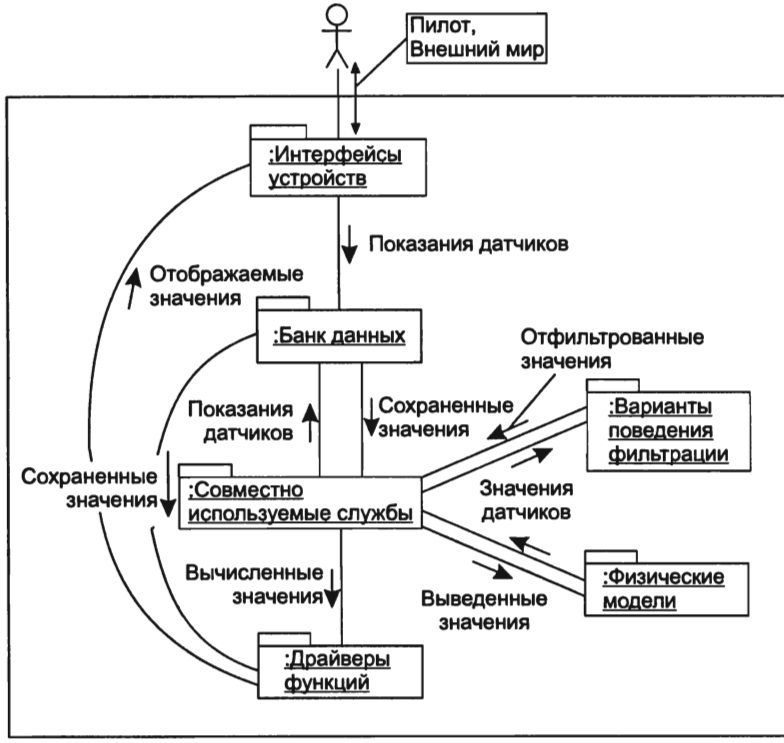
— LJB

3.4. Заключение

Мы рассмотрели архитектуру высокопроизводительной авиационной системы с точки зрения трех связанных, но различающихся структур. Структура декомпозиции модулей описывает отношения между компонентами — блоками реализации, которые распределяются между группами разработчиков, — в период проектирования. Структура использования описывает отношения использования между компонентами — процедурами и модулями — в период прогона. На основе этих сведений формируются очертания многоуровневой архитектуры. Структура процессов описывает параллелизм системы и является основой для их распределения между элементами аппаратной части.

Каждую из этих структур важно спроектировать как можно тщательнее — дело в том, что они закладывают основу для реализации трех атрибутов качества: 1) простоты изменения, 2) простоты извлечения подмножеств и 3) повышенных параллелизма и производительности. Не менее существенное значение имеет составление для каждой из этих структур комплексной документации, поскольку в других документах сведений о них не содержится.

Несмотря на ортогональность упомянутых структур, они связаны — в модулях содержатся процедуры, которые, с одной стороны, обращаются друг к другу, а с другой — совместно работают в рамках процессов. Для системы A-7E можно было задать и другие структуры — в частности, представление потока данных (представление «компонент и соединитель», вспомогательное по отношению к тем, что рассматривались в главе 2) могло бы выглядеть так, как показано на рис. 3.5. Все данные через модули интерфейсов устройств приходят из внешнего мира; затем, минуя этапы вычисления и модули хранения, они добираются до модулей управления функциями; те вычисляют выходные значения и отправляют их обратно соответствующим устройством. Проектировщики системы A-7E, впрочем, считали представления потока данных ненужными. «Какой атрибут качества из тех, что не реализуют другие представления, помогает реализовать это?» — так они аргументировали свою позицию. Но мнения могут быть разными, и иные проектировщики, возможно, удостоят своего внимания представление потока данных. По сути своей, архитектурные представления призваны углублять знания о системе и ее свойствах, усиливать интеллектуальный контроль над ней. Если то или иное представление соответствует этим требованиям, значит, оно вам подойдет.



Составлено на языке UML

Рис. 3.5. Набросок представления потока данных в программной системе A-7E

Помимо прочего, мы рассмотрели архитектуру в контексте атрибутов качества, которые проектировщики стремились реализовать, — заменяемости и понятности. В этой связи уместно привести тезис, аргументацией которого мы займемся в двух последующих главах: «варианты архитектуры отражают ряд желаемых атрибутов качества».

3.5. Дополнительная литература

Документация по проекту авиационной электронной системы A-7E содержится в работе [Parnas 85a]. Анализ данных о внесенных в систему изменениях и описание этих изменений приводятся в исследованиях [Hager 91] и [Hager 89]. Значительную часть материала о модульной структуре мы позаимствовали из руководства по модулям A-7E, написанного Кэтрин Бриттон (Kathryn Britton) и Дэвидом Парнасом [Britton 81].

3.6. Дискуссионные вопросы

1. Предположим, что одна из версий программной системы A-7E установлена на летном тренажере. На нем нет вооружения, а его функция заключается в обучении пилотов способам навигации при помощи бортовой электроники. Какие архитектурные структуры следовало модифицировать в процессе разработки такой системы и каким образом эти изменения нужно было внести?
2. В главе 7 мы обсудим архитектуру как основу для инкрементной разработки; этот способ предполагает постепенное наращивание системы и регулярное развертывание ее готовых подмножеств. Предположим, что даже самое небольшое подмножество программной системы A-7E (корректно, согласно требованиям) выполняет некую функцию, результаты которой видны пилоту. (Скажем, отображает некоторое значение — например, выводит на бортовой индикатор текущий курс.) Какие модули требуются этому подмножеству и без каких можно обойтись? Предложите три инкрементных дополнения к нему и наметьте для них план разработки (иначе говоря, составьте перечень необходимых модулей).
3. Предположим, что для проверки правильности значений, хранящихся в банке данных и вычисляемых драйверами функций, в систему введен ряд контрольных блоков. В случае обнаружения несоответствия между хранимыми или вычисленными значениями, с одной стороны, и корректными значениями, которые определяются этими блоками, — с другой, они должны подавать сигнал об ошибке. Расскажите, какие изменения следует внести в каждую из архитектурных структур системы A-7E в целях адаптации к этому решению? Если вы считаете необходимым введение дополнительных модулей, перечислите критерии сокрытия информации, по которым эти модули будут размещены в иерархической системе.

Часть 2
СОЗДАНИЕ АРХИТЕКТУРЫ

В части 1 книги мы сформулировали определение архитектурно-экономического цикла (Architecture Business Cycle, ABC) и разобрались с основными понятиями, необходимыми для дальнейшего изучения программной архитектуры. В частности, мы установили факторы влияния на архитектора, проявляющиеся на начальных стадиях производства систем, и выяснили, что требования к тем или иным атрибутам качества — будь то производительность или модифицируемость — во многих случаях обусловливаются коммерческими задачами компании-разработчика. Так что же собой представляет процесс создания архитектуры архитектором? Об этом речь пойдет в части 2. Поскольку успех системы в значительной степени определяется реализацией атрибутов качества, мы начнем с рассмотрения качества и тех средств, при помощи которых архитектор способен его обеспечить.

Перефразируя Бута Таркингтона (Booth Tarkington), скажем, что качество — в глазах смотрящего. Заказчики не обязаны принимать решения архитектора с бурным восторгом — у них, в конце концов, могут быть свои представления о качестве. Инструментом объективной оценки качества являются сценарии атрибутов качества. В главе 4 мы рассмотрим различные составляющие качества, которые в тех или иных обстоятельствах оказываются значимыми для архитектуры. Для всех шести важнейших атрибутов (готовность, модифицируемость, производительность, безопасность, контролепригодность и практичность) мы представим методики составления сценариев, отражающих требования по качеству. Эти сценарии определяют степень значимости конкретного атрибута качества в контексте данной системы, и именно исходя из этой его оценки архитекторы и заказчики должны выносить суждения о проекте.

Впрочем, анализ требований по качеству для архитектора — не более чем средство постановки задачи. В главе 5 речь пойдет о тех имеющихся в распоряжении любого архитектора инструментах (тактиках и образцах), при помощи которых он должен реализовывать атрибуты качества. Для достижения высокой готовности, к примеру, необходимо в той или иной форме организовать резервирование данных или кода. Резервирование, в свою очередь, заставляет архитектора решать новые проблемы — в частности, обеспечивать синхронизацию точных копий.

Глава 6 отводится под второй в книге конкретный пример — систему, разработанную по заказу Федерального авиационного агентства США и предназначенную для реализации функций управления воздушным движением. Эта система, к которой в процессе проектирования предъявлялись очень высокие требования по готовности (ограничивавшие простой пятью минутами в год), иллюстрирует применение тактик, перечисленных в главе 5.

Сценарии атрибутов качества и архитектурные тактики — это лишь некоторые из инструментов архитектора. В главе 7 мы обсудим, как эти инструменты применяются при проектировании архитектуры и создании макета системы; кроме того, мы проанализируем влияние архитектуры на структуру компании-разработчика.

В главе 8 рассматривается третий конкретный пример — система моделирования условий полета. От подобного рода систем требуется производительность в реальном времени и удобство модифицирования. Мы раскроем механизмы реализации этих задач.

Спроектированную архитектуру необходимо задокументировать. В первую очередь документируются значимые представления и только после этого — материал, выходящий за рамки представлений. Обзор методик документирования архитектуры содержится в главе 9.

Проблема отсутствия материалов по архитектуре встречается сплошь и рядом — иногда ее забывают задокументировать, в других случаях документация теряется; наконец, зачастую фактический вариант системы обнаруживает значительные отличия от проектного варианта. Процесс восстановления архитектуры существующей системы рассматривается в главе 10.

Глава 4
Атрибуты качества

(в соавторстве с Феликсом Бахманом и Марком Кляйном)[1]

— Котик! Чешик! — робко начала Алиса. — Скажите, пожалуйста, куда мне отсюда идти?
— А куда ты хочешь попасть? — ответил Кот.
— Мне все равно... — сказала Алиса.
— Тогда все равно, куда и идти, — заметил Кот.
— ...только бы попасть куда-нибудь, — пояснила Алиса.
— Куда-нибудь ты обязательно попадешь, — сказал Кот. — Нужно только достаточно долго идти.

Льюис Кэрролл. «Алиса в стране чудес»

Архитектурно-экономический цикл убеждает нас в том, что требования по качеству к системной архитектуре определяются коммерческими мотивами. Подобного рода атрибуты качества сопровождают основные требования по функциональности — формулировки возможностей и поведения системы, а также предоставляемых ею услуг. Несмотря на явное сходство функциональности с другими атрибутами качества, она занимает в схеме разработки центральное, а иногда и единоличное, положение, и в этом у вас еще будет возможность убедиться. Мы считаем, что такой подход недальновиден. Причины переработки систем часто кроются не в недостатке функциональности — по этому показателю новые элементы как раз ничем не отличаются от старых, а в трудностях сопровождения, переноса, масштабирования, низкой производительности или противостояния сетевым атакам. В главе 2 мы говорили о том, что формирование архитектуры — это первый этап процесса создания программного продукта, на котором основное внимание следует уделять требованиям по качеству. Функциональность системы здесь отображается на программные структуры, от которых и зависит, сможет ли архитектура обеспечить наличие в системе необходимых атрибутов качества. В главе 5 мы обсудим влияние на реализацию качества архитектурно-проектных решений,

[1] Феликс Бахман (Felix Bachmann) и Марк Кляйн (Mark Klein) — старшие научные сотрудники Института программной инженерии.

а в главе 7 — стратегию компромиссных решений, с необходимостью принятия которых постоянно сталкиваются архитекторы.

Пока что мы намерены рассмотреть вопрос выражения атрибутов качества, наличие которых в предполагаемой системе или системах должна обеспечить разрабатываемая архитектура. Анализ отношений между программной архитектурой и атрибутами качества начнем с подробного описания последних. Что имеется в виду, когда систему называют, скажем, модифицируемой, надежной или защищенной? Характеристики подобных свойств и методика формулирования требований по качеству к системе — вот о чем мы будем говорить ниже.

4.1. Функциональность и архитектура

Функциональность и атрибуты качества ортогональны. Это утверждение, которое поначалу может показаться излишне смелым, на самом деле совершенно справедливо — по-другому просто не может быть. Если бы функциональность и атрибуты качества не были ортогональны, уровень защиты, производительности, готовности или практичности определялся бы выбранной функцией. Очевидно, что устанавливать нужный уровень этих характеристик можно произвольно. Мы не имеем в виду, что любая функция дает возможность реализовать любой атрибут качества на любом уровне. Операции обработки сложных графических данных или сортировки огромной базы данных сложны по определению и, следовательно, не позволяют достичь молниеносной производительности. Сомнению не подлежит лишь то, что в отношении любой из этих функций относительный уровень качества определяется решениями, которые принимает архитектор. Некоторые архитектурные решения способствуют повышению производительности, иные же приводят к обратному эффекту. В этом контексте целью настоящей главы, как и любой хорошей архитектуры, является разделение задач. Мы разберем все важные атрибуты качества по порядку и покажем, как их структурировать.

Что такое функциональность? Это способность системы выполнять те задачи, которые на нее возложены. Для выполнения любой задачи элементы системы — все или некоторые — должны работать согласованно; аналогичным образом в ходе строительства дома взаимодействуют монтажники, электрики, кровельщики, штукатуры, маляры и отделочники. Таким образом, если обязанности между элементами распределены неправильно или если у них нет средств координации с другими элементами (они требуются для того, чтобы элементы могли своевременно приступать к исполнению своих обязанностей), система не сможет обеспечить надлежащую функциональность.

Для обеспечения функциональности существует множество разных структур. Если бы функциональность была единственным требованием, систему можно было бы организовать в виде единичного монолитного модуля безо всякой внутренней структуры. В действительности системы разлагаются на модули, которые делают ее более понятной и поддерживают решение ряда других задач. Функциональность, таким образом, не сильно зависит от структуры. Когда существенную роль начинают играть другие атрибуты качества, программная архитектура ограничивает

распространение функциональности по структурам. К примеру, во многих случаях декомпозиция структур проводится таким образом, чтобы их конструированием могли заниматься сразу несколько специалистов (помимо прочего, такое решение оптимизирует срок выхода продукта на рынок, хотя в этом ключе оно рассматривается весьма редко). Очень важно, как функциональность пересекается с другими атрибутами качества и одновременно ограничивает их.

4.2. Архитектура и атрибуты качества

Вопросы воплощения атрибутов качества решаются в периоды проектирования, реализации и развертывания. Не существует ни одного атрибута качества, который зависел бы исключительно от какого-то отдельного этапа. Для достижения оптимальных результатов требуется правильность в общем (в архитектуре) и в частностях (в реализации).

- Практичность имеет как архитектурные, так и не архитектурные аспекты. Среди последних — обеспечение ясности и простоты применения пользовательского интерфейса. Что следует предпочесть: переключатель или флажок? Какая схема размещения информации на экране наиболее интуитивна? Какая гарнитура шрифта отображается четче других? Все эти вещи имеют существенное значение для конечного пользователя и влияют на практичность, но при этом, являя собой локальные дизайнерские решения, они не имеют отношения к архитектуре. Архитектура в данном случае — это возможность отменять операции, возвращаться к предыдущему состоянию и повторно обращаться к ранее введенным данным. Для реализации этих требований необходимо взаимодействие множества элементов.
- Модифицируемость определяется тем, каким образом происходит разделение функциональности (архитектурный аспект) и методик кодирования в рамках отдельного модуля (неархитектурный аспект). Так, назвать систему модифицируемой можно лишь в том случае, если для внесения изменений требуется наименьшее количество отдельных элементов. Именно на этом строится структура декомпозиции модулей системы A-7E, описанной в главе 3. Чем менее структурирован код, тем хуже система поддается модифицированию.
- Производительность также зависит от архитектурных и неархитектурных факторов. Отчасти она зависит от объема информации, передающейся между компонентами (это архитектурный аспект), от распределения функциональности между компонентами (архитектурный аспект), распределения совместно используемых ресурсов (архитектурный аспект), выбранных для реализации функциональности алгоритмов (неархитектурный аспект) и кодирования этих алгоритмов (также неархитектурный аспект).

В этом разделе мы хотим сделать упор на двух моментах:

1. Архитектура определяет возможность реализации предполагаемых атрибутов качества системы; проектирование и оценку этих атрибутов следует проводить на архитектурном уровне.

2. Сама по себе архитектура не реализует никаких атрибутов качества, но она образует основу для достижения качества — впрочем, если не уделять должного внимания деталям, пользы от этой основы не будет никакой.

Реализовать ряд атрибутов качества по отдельности в сложной системе невозможно. Реализация одного такого атрибута всегда каким-то образом — иногда положительно, а иногда и отрицательно — влияет на реализацию других. К примеру, такие атрибуты качества, как безопасность и надежность, обычно конфликтуют между собой. У самой защищенной системы наименьшее количество точек отказа — как правило, они концентрируются в ядре безопасности. У самой надежной системы их, напротив, наибольшее количество — как правило, это ряд резервируемых процессов или процессоров, причем отказ одного из них не приводит к отказу системы в целом. Другой пример конфликта между атрибутами качества очевиден — практически любой атрибут качества негативно отражается на производительности. Взять хотя бы переносимость. Лучший способ обеспечить переносимость программного продукта — это изолировать системные зависимости. Такое решение увеличивает непроизводительные издержки в ходе исполнения системы (обычно на границах процессов или процедур), а следовательно, снижает производительность.

Теперь приступим к обзору атрибутов качества. Они делятся на три класса:

1. Атрибуты качества системы. Из них мы рассмотрим готовность, модифицируемость, производительность, безопасность, контролепригодность и практичность.
2. Коммерческие атрибуты качества (например, срок вывода продукта на рынок), реализация которых обусловливается архитектурой.
3. Атрибуты качества самой архитектуры (например, концептуальная целостность), которые косвенно влияют на другие качества — например, модифицируемость.

4.3. Атрибуты качества системы

Атрибуты качества систем изучаются специалистами-проектировщиками программных средств по крайней мере с 1970-х годов. Существует множество различных классификаций и определений, нашедших своих приверженцев среди теоретиков и практиков. С точки зрения архитектора, у проведенных к настоящему моменту исследований атрибутов качества систем есть три основных недостатка.

- Все предложенные определения атрибутов непрактичны. Сказать, что система будет модифицируемой, — значит ничего не сказать. Любая система модифицируема в отношении одного ряда изменений и немодифицируема в отношении другого такого ряда. Другие атрибуты страдают тем же недугом.
- Очень часто обсуждения сосредоточиваются на том, к какому атрибуту качества следует отнести тот или иной аспект. К примеру, аспектом чего

является отказ системы: готовности, безопасности или практичности? В зависимости от того, кто чем занимается, специалисты тянут одеяло на себя.

- У каждого сообщества «качества» свой словарь. Одно и то же явление специалисты по производительности называют «событием», специалисты по безопасности — «атакой», специалисты по готовности — «сбоем», а специалисты по практичности — «данными, вводимыми пользователем». Разнобой в терминологии сбивает с толку.

Первые две проблемы (непрактичные определения и несогласованность в классификации аспектов) решаются путем применения для составления характеристик *сценариев атрибутов качества* (quality attribute scenarios). Решение третьей проблемы представляется нам так: каждый атрибут следует обсуждать кратко, с упором на фундаментальные задачи, и тем самым прояснять понятия, которые играют первостепенную роль для той или иной специализации.

Сценарии атрибутов качества

Сценарием атрибута качества называется требование, путем выполнения которого этот атрибут реализуется. Сценарий состоит из шести элементов.

- *Источник стимула.* Это некий субъект (человек, вычислительная система или любой другой), который порождает стимул.
- *Стимул.* Стимулом называется наблюдаемое в системе явление, требующее к себе внимания.
- *Условия.* Стимул возникает в определенных условиях. К примеру, система может находиться в состоянии перегрузки или исполняться в обычном режиме.
- *Артефакт.* Объектом воздействия стимула является некий артефакт. В этом качестве может выступать как система в целом, так и ее отдельные элементы.
- *Реакция.* Реакция — это действие, предпринятое в ответ на появление стимула.
- *Количественная мера реакции.* Предпринимаемые в ответ на стимул действия должны быть измеримы — только в этом случае соответствие требованию можно проверить.

Сценарии атрибутов качества, по нашему мнению, можно разделить на две группы: общие — те, что не зависят от конкретный системы и потенциально применимы к любой системе, и конкретные — те, что характерны только для рассматриваемой системы. Характеристики атрибутов здесь представлены как совокупности общих сценариев; следует, впрочем, иметь в виду, что для составления на основе одной из таких характеристик требований к конкретной системе соответствующие общие сценарии следует перевести в конкретные.

Элементы сценария атрибута качества показаны на рис. 4.1.

Сценарий готовности

Общий сценарий атрибута качества «готовность» представлен на рис. 4.2. Для всех его шести элементов указаны диапазоны допустимых значений. На основе

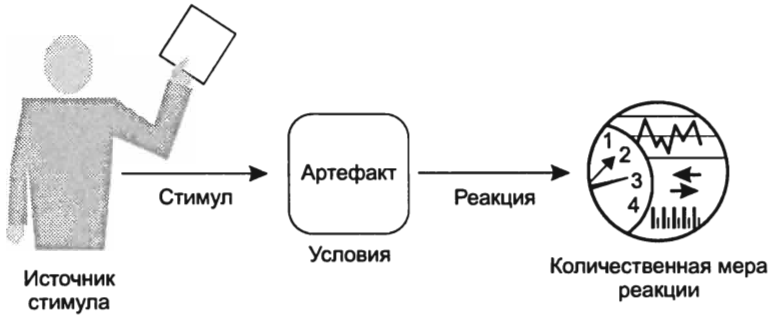


Рис. 4.1. Элементы атрибута качества

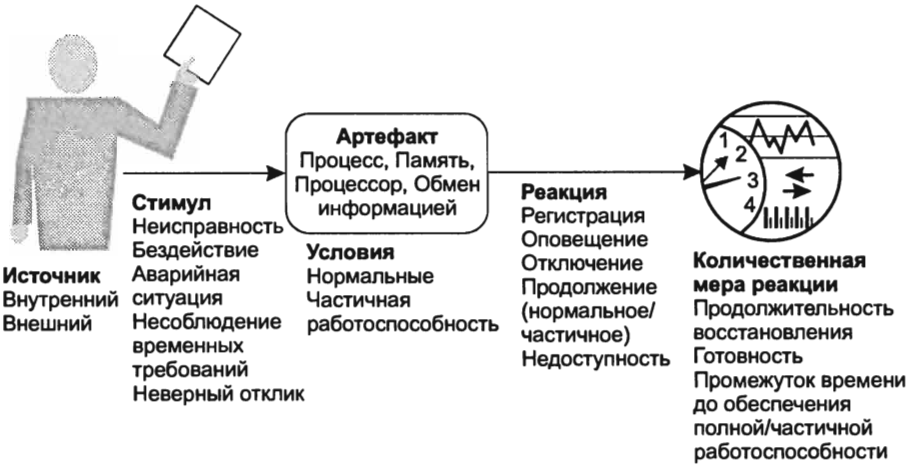


Рис. 4.2. Общие сценарии готовности

этого общего сценария можно создавать конкретные, системно-ориентированные сценарии. Впрочем, не во всех таких сценариях будет шесть элементов. Необходимые элементы выявляются по результатам применения сценария и согласно типам тестирования, при помощи которых планируется проводить проверку его реализации.

К примеру, на основе общего сценария, показанного на рис. 4.2, путем конкретизации каждого из его элементов можно получить такой конкретный сценарий: «Процесс, находясь в нормальном режиме работы, получает извне непредвиденное сообщение. Проинформировав о приеме сообщения соответствующую операцию, система, не переходя в состояние простоя, продолжает работать». Частично этот сценарий представлен на рис. 4.3.

От того, что выступает в качестве источника стимула, зависит характер реакции. К примеру, в сценарии безопасности могут предусматриваться разные реакции на запросы от доверенных и ненадежных источников. Кроме того, влияние на реакцию оказывают условия — если, скажем, система перегружена, реакция на событие может быть иной, чем в том случае, если бы система работала в нормальном режиме. Артефакт в этом контексте не столь важен. В этом качестве

почти всегда выступает система, причем по двум нижеизложенным причинам ее вызов производится явно.

Во-первых, во многих требованиях принимаются допущения относительно составляющих системы (например, «при отказе веб-сервера системы»). Во-вторых, в случае применения сценариев в рамках методов оценки или проектирования сведения о стимулированном элементе системы в артефакте делаются более явными. Наконец, наличие явных сведений о значении реакции способствует прояснению требований по атрибуту качества. По этим причинам мы оставили в составе сценария элемент «количественная мера реакции».

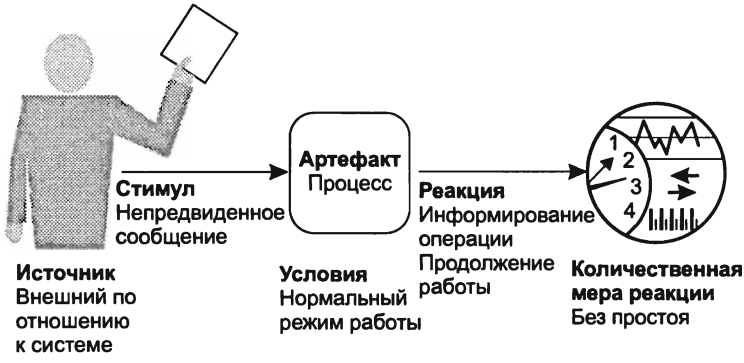


Рис. 4.3. Пример сценария готовности

Сценарий модифицируемости

Пример сценария модифицируемости: «В период проектирования разработчику требуется внести изменения в код пользовательского интерфейса, чтобы сделать фоновый цвет синим. Соответствующие операции и тестирование проводились в течение трех часов и не привели к появлению побочных эффектов в поведении». За исключением ряда деталей, этот сценарий приводится в графическом виде на рис. 4.4.

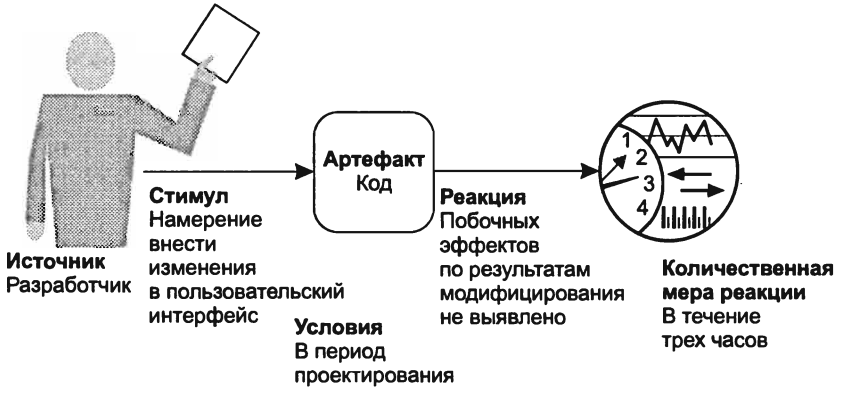


Рис. 4.4. Пример сценария модифицируемости

Требования по атрибутам качества свойств системы можно представить в виде ряда конкретных сценариев. Каждый из них должен быть понятен архитектору, а подробности реакции должны быть сформулированы так, чтобы возможным стало проведение проверки на предмет ее реализации в системе. В процессе выявления требований общие сценарии имеет смысл рассматривать в категориях атрибутов качества; если оказывается, что два различных атрибута порождают один и тот же сценарий, один из этих атрибутов исключается.

Для каждого атрибута строится отдельная таблица со всеми возможными системно-независимыми значениями шести элементов сценария качества. При создании общего сценария качества для каждого из элементов отбирается по одному значению. Конкретный сценарий создается в ходе выявления требований путем выбора в каждом столбце таблицы одной или нескольких записей и последующего обеспечения читаемости. К примеру, сценарий, показанный на рис. 4.4, выведен из сценария модифицируемости (см. табл. 4.2); при этом отдельные части были незначительно изменены для обеспечения совместимости с форматом сценария.

Роль конкретных сценариев при специфицировании требований по атрибутам качества близка к функции элементов Use Case при специфицировании функциональных требований.

Создание сценария атрибута качества

На материале этой главы мы пытаемся показать, как архитекторы формулируют содержательные атрибуты качества системы. Теоретически, эта задача должна выполняться в процессе выявления требований к проекту, однако на практике так происходит не часто. Как мы говорили в главе 1, случаи систематического выявления и фиксации требований к системе по качеству встречаются не так уж часто. Исправить эту ситуацию можно путем составления конкретных сценариев атрибутов качества. Для этого используются конкретные таблицы атрибутов качества, на основе которых строятся общие сценарии, а из них, в свою очередь, выводятся сценарии системно-ориентированные. Как правило, создаются не все возможные общие сценарии. Таблицы составляются скорее для того, чтобы обеспечить учет всех возможных вариантов; они не выступают в качестве явного механизма генерации. Нас не интересуют сценарии, которые не согласуются с точным определением свойства, — избыточность, которая возникает, если одно и то же требование по качеству можно сформулировать на основе двух разных атрибутов, легко устранить. С другой стороны, если одно из существенных требований по качеству не учтено, последствия могут быть весьма серьезными.

4.4. Практическое применение сценариев атрибутов качества

Общие сценарии — это основа для производства многочисленных родовых, системно-независимых, конкретных сценариев атрибутов качества. Каждый из них

потенциально (хотя и не обязательно) имеет существенное значение для конкретной системы. Для того чтобы адаптировать общие сценарии к такой системе, им необходимо придать системно-ориентированный характер.

Под «преобразованием общего сценария в системно-ориентированный» имеется в виду его формулировка в конкретных терминах для конкретной системы. Предположим, что общий сценарий звучит так: «Согласно поступившему запросу на изменение функциональности, все необходимые коррективы следует внести в определенный момент периода разработки, придерживаясь при этом установленных сроков». А вот его системно-ориентированная версия: «Согласно поступившему запросу на обеспечение поддержки веб-системой нового браузера, все необходимые коррективы следует внести в течение двух недель». Необходимо отметить, что одному общему сценарию может соответствовать несколько системно-ориентированных версий — к примеру, если в одной и той же системе требуется реализовать поддержку нового браузера и нового носителя.

Ниже мы рассмотрим шесть наиболее универсальных и существенных атрибутов качества системы. Этим мы решим две задачи: во-первых, обозначим понятия, которые применяются в тех или иных сообществах специалистов, и, во-вторых, изложим способы составления общих сценариев для отдельных атрибутов.

Готовность

Готовность отражает ситуацию отказа системы и его возможных последствий. Отказом системы называется положение, при котором система теряет способность предоставления услуг, заявленных в ее спецификации. Отказ заметен для пользователей системы, в качестве которых могут выступать люди или другие системы. Пример общего сценария готовности приведен на рис. 4.3.

Внимание здесь обращено на способ обнаружения отказов системы, частоту их появления, действия, следующие за ними, временной период, в продолжение которого система может быть нефункциональна, ситуации, при которых отказы могут проходить без последствий, способы их предотвращения, а также типы оповещений, следующих после их обнаружения.

Отказ (failure) и *неисправность* (fault) — это не одно и то же. Если неисправность не устранена или не замаскирована, она может перейти в состояние отказа. Пользователь системы может наблюдать ситуацию отказа, однако о неисправности он не имеет никакого представления. Когда неисправность становится обозримой для пользователя, она становится отказом. К примеру, неисправность может возникнуть из-за неверного выбора алгоритма вычислений; из-за этого произойдет ошибка в расчетах, которая в свою очередь приведет систему в состояние отказа.

После отказа системы на первый план выходит вопрос о том, сколько времени потребуется на его устранение. Из того, что отказ системы обозрим для пользователей, можно сделать вывод, что период устранения отказа эквивалентен периоду, по истечении которого отказ перестает быть обозримым. Вариантность здесь огромна — от небольшой задержки отклика до перелета в перуанское высокогорье для починки блока горного оборудования (последний пример привел чело-

век, занимающийся обслуживанием программного обеспечения двигателя горных машин).

Различие между неисправностями и отказами приводит к мысли о стратегиях автоматического восстановления работоспособности. Ведь если код, в котором наблюдается неисправность, исполняется, а затем система устраняет последствия этой неисправности, не выводя ее на внешний уровень, говорить об отказе не приходится.

Готовность системы — это вероятность функционирования системы, когда в этом есть необходимость. Как правило, она выражается формулой:

$$\alpha = \frac{\textit{среднее время до появления отказа}}{\textit{среднее время до появления отказа} + \textit{средняя продолжительность восстановления}}$$

Вычисления по этой формуле дают результат наподобие 99,9 % готовности — или, другими словами, 0,1 % вероятности того, что, когда в использовании системы возникнет необходимость, она окажется нефункциональна.

Плановый простой при вычислении готовности обычно не учитывают — предполагается, что в такие периоды система «не нужна» по определению. В результате иногда случается так, что система выходит из строя, пользователи ждут, пока ее починят, но, поскольку простой оказывается запланированным, он никак не влияет на показатели готовности.

Общие сценарии готовности

Ниже описываются элементы сценария готовности, которые, в частности, показаны на рис. 4.2.

- *Источник стимула*. Признаки отказов и неисправностей делятся на внутренние и внешние — в зависимости от их характера система может демонстрировать разные реакции. В нашем примере непредвиденное сообщение приходит извне.
- *Стимул*. Неисправности подразделяются на несколько типов.
  - ◊ *Бездействие*. Компонент не реагирует на входные данные.
  - ◊ *Аварийная ситуация*. Компонент регулярно демонстрирует бездействие.
  - ◊ *Несоблюдение временных требований*. Компонент реагирует, но отклик происходит слишком рано или слишком поздно.
  - ◊ *Неверная реакция*. Компонент реагирует, но выдает неверное значение.

  В сценарии, изображенном на рис. 4.3, стимулом является поступление непредвиденного сообщения. Это — пример отклонения от временных требований. Компонент сгенерировал сообщение не вовремя.
- *Артефакт*. Под артефактом имеется в виду ресурс, к которому предъявлены требования по готовности, — например, процессор, канал обмена данными, процесс или память.
- *Условия*. Характер предполагаемой реакции зависит, помимо прочего, от состояния системы в момент неисправности или отказа. К примеру, если система находится под воздействием предшествующих неисправностей

и работает вне нормального режима, при появлении новых неисправностей ее желательно выключить. С другой стороны, если речь идет о первой зафиксированной неисправности, вполне достаточно замедления отклика или исполнения функции. В нашем примере система работает в нормальном режиме.

- *Реакция.* Система может по-разному реагировать на отказы. Среди возможных вариантов — регистрация отказа, оповещение о нем отдельных пользователей или сторонних систем, переключение в безопасный режим с меньшими возможностями или функциональностью, включение внешних систем или выключение на период восстановления. В нашем примере система должна оповестить оператора о непредвиденном сообщении, а затем продолжить работу в нормальном режиме.
- *Количественная мера реакции.* Этот показатель выражает процент готовности, продолжительность восстановления, периоды, в течение которых система должна пребывать в состоянии готовности, и продолжительность готовности. Согласно схеме на рис. 4.3, простоя в результате поступления непредвиденного сообщения не происходит.

Возможные значения всех элементов сценария готовности представлены в табл. 4.1.

Модифицируемость

Качество модифицируемости выражает стоимость внесения изменений. Оно решает два вопроса.

1. *Под воздействием чего артефакт может быть изменен?* Изменения можно внести в любой аспект системы — как правило, это ее функции, платформа (аппаратная часть, операционная система, промежуточное программное обеспечение и т. д.), условия, в которых функционирует система (системы, с которыми она взаимодействует, протоколы, при помощи которых обменивается информацией с внешним миром, и т. д.), ее атрибуты качества (производительность, надежность системы или даже предполагаемые в будущем изменения) и возможности (количество пользователей, одновременных операций и т. д.). Некоторые элементы системы — например, пользовательский интерфейс и платформа — в этом отношении настолько выделяются из числа других, что их мы рассмотрим отдельно. Категория изменения платформы имеет также другое название — переносимость. Изменения могут заключаться в добавлении, удалении и модификации этих аспектов.
2. *Когда производится модификация и кто ее проводит (условия)?* В прошлом изменения чаще всего вносились в исходный код. При этом разработчик проводил модификацию, после чего она тестировалась и размещалась в новой версии. Сегодня проблема времени модификации все больше переплетается с проблемой ее исполнителя. Конечный пользователь, меняя заставку, тем самым вносит изменения в один из аспектов системы. Не менее очевидно, что модификация системы, проведенная с расчетом на то, чтобы ею можно было пользоваться удаленно, — это несколько другая категория

изменений. Изменения можно вносить в реализацию (путем редактирования исходного кода), в периоды компиляции (при помощи переключателей периода компиляции), построения (путем выбора тех или иных библиотек), настройки конфигурации (разными методами, в частности — путем установки параметров) или исполнения (также посредством установки параметров). В качестве исполнителей модификации могут выступать разработчики, конечные пользователи или администраторы.

Таблица 4.1. Составление общего сценария готовности

Элемент сценария	Возможные значения
Источник	Внутренний по отношению к системе; внешний по отношению к системе
Стимул	Неисправность: бездействие, аварийная ситуация, несоблюдение временных ограничений, неверная реакция
Артефакт	Процессоры, информационные каналы, постоянная память, процессы системы
Условия	Нормальный режим работы; режим с ухудшенными характеристиками (за счет уменьшения возможностей происходит нейтрализация неисправностей)
Реакция	Система должна обнаружить событие и выполнить одно из нижеследующих действий: ✦ зарегистрировать его; ✦ оповестить заинтересованные стороны — в частности, пользователя и другие системы; ✦ руководствуясь установленными правилами, отключить источники событий, которые приводят к неисправности или сбою; ✦ перейти в недоступное состояние и оставаться в нем в течение предварительно установленного периода времени, продолжительность которого должна зависеть от критичности системы; ✦ продолжить работу в нормальном режиме или режиме с ухудшенными характеристиками
Количественная мера реакции	Период времени, в течение которого система должна демонстрировать готовность; продолжительность готовности; период времени, в течение которого система может находиться в режиме с ухудшенными характеристиками; продолжительность восстановления

После того как изменение сформулировано, следует проектирование новой реализации, собственно реализация, тестирование и развертывание. На все это уходит время и расходуются деньги — впрочем, и то и другое можно подсчитать.

Общие сценарии модифицируемости

Принимая во внимание все вышеприведенные факторы, приступим к рассмотрению отдельных элементов общих сценариев модифицируемости. На рис. 4.4 приводится пример: «В период проектирования разработчику требуется внести изменения в код пользовательского интерфейса, с тем чтобы сделать фоновый цвет синим. Соответствующие операции и тестирование проводились в течение трех часов и не привели к появлению побочных эффектов в поведении».

- *Источник стимула.* В этой части определяется исполнитель изменений — разработчик, администратор системы или конечный пользователь. Естественно,

и тот, и другой, и третий выполняют действия с помощью неких механизмов, но это обстоятельство во внимание не принимается. На рис. 4.4 изменения вносятся разработчиком.

- *Стимул.* Здесь указываются конкретные изменения, которые предполагается внести. Среди возможных вариантов — введение новой или изменение/удаление существующей функции. Кроме того, изменения могут касаться атрибутов качества системы — в частности, повышения реактивности, готовности и т. д. Нельзя исключать и изменение мощности системы. Во многих случаях требуется увеличить количество пользователей, которые могут работать в системе одновременно. В нашем примере стимулом является абстрактная потребность в модификации — речь с одинаковой вероятностью может идти о функции, качестве или мощности.

  Для линеек программных продуктов (см. главу 14) характерен такой показатель, как изменчивость. Фактором в этом контексте называется количество определений данной изменчивости. Регулярная изменчивость налагает на количественную меру реакции более жесткие требования, чем изменчивость случайная.
- *Артефакт.* Этот элемент указывает объект модификации — функциональность системы, ее платформа, пользовательский интерфейс, окружение или сторонняя система, с которой взаимодействует рассматриваемая. Согласно рис. 4.4, модификации подвергается пользовательский интерфейс.
- *Условия.* Здесь определяется время модификации — это могут быть периоды проектирования, компиляции, построения, инициирования или прогона. В нашем случае модификация проводится в период проектирования.
- *Реакция.* Кто бы ни выступал в роли исполнителя модификации, он должен знать, какая задача перед ним стоит, выполнить эту задачу, провести тестирование и развертывание. В нашем примере модификация не приводит к каким-либо побочным эффектам.
- *Количественная мера реакции.* Любые реакции сопряжены с временными и финансовыми затратами, поэтому в качестве единиц измерения желательно принять время и деньги. Поскольку предсказать временные затраты не всегда возможно, во многих случаях применяются более конкретные единицы — например, объем изменений (выражает количество подверженных модификации модулей). В нашем примере содержится требование о том, что по своей продолжительности модификация не должна превышать трех часов.

Возможные значения всех элементов сценария модифицируемости представлены в табл. 4.2.

Производительность

Производительность — это временная характеристика. Когда фиксируются те или иные события (прерывания, сообщения, пользовательские запросы, временные промежутки), система должна на них реагировать. Характеристик поступления

событий и реакций на них великое множество, но, по сути, вопрос заключается в том, за какое время система справляется с реакцией на то или иное событие.

Таблица 4.2. Составление общего сценария модифицируемости

Элемент сценария	Возможные значения
Источник	Конечный пользователь, разработчик, администратор системы
Стимул	Намерение ввести/удалить/изменить/видоизменить функциональность, атрибут качества, мощность
Артефакт	Пользовательский интерфейс системы, платформа, окружение; сторонняя система, взаимодействующая с целевой
Условия	В период прогона, компиляции, построения, проектирования
Реакция	Локализация изменений в рамках архитектуры; выполнение модификации без воздействия на остальную функциональность; тестирование изменений; развертывание изменений
Количественная мера реакции	Стоимость в категориях количества модифицированных элементов, трудовых и денежных затрат; степень, в которой произведенные изменения затрагивают другие функции или атрибуты качества

Среди факторов, заметно осложняющих производительность, следует выделить значительное количество источников событий и образцов поступления. Источниками событий могут выступать запросы пользователей, которые, в свою очередь, делятся на поступающие из сторонних систем и из рассматриваемой системы. Сетевая система финансовых услуг имеет источником поступающих событий своих пользователей (иногда в количестве от десятков до сотен тысяч). Система управления двигателем, регулярно получающая разного рода запросы, должна одновременно следить за положением цилиндра в момент зажигания и характеристиками топливо-воздушной смеси, которая должна обеспечивать максимальную мощность при минимальном загрязнении воздуха.

Что касается сетевой финансовой системы, то здесь реакцией может быть показатель количества транзакций, обрабатываемых за одну минуту. В системе управления двигателем в этой роли может выступать изменчивость момента зажигания. Как бы то ни было, образцы поступающих событий и реакций поддаются систематизации, в результате которой появляется язык конструирования общих сценариев производительности.

Сценарий производительности начинается с поступающего в систему запроса на обслуживание. Для выполнения этого запроса требуется затратить некие ресурсы. Одновременно система может обслуживать другие запросы.

Образцы поступления событий делятся на периодические и непериодические. Периодическое событие, к примеру, может происходить каждые 10 мс. Такого рода события чаще всего встречаются в системах реального времени. Непериодические события поступают согласно некоему вероятностному распределению. Кроме того, события иногда происходят случайно — по образцу, не поддающемуся периодической или непериодической систематизации.

Путем варьирования образца поступления событий можно моделировать ситуацию множества пользователей и прочие факторы нагрузки. Другими словами, с точки зрения производительности системы, подаст ли один пользователь за

установленный период времени 20 запросов или за то же время по 10 запросов подадут два пользователя, — не имеет никакого значения. Существен лишь образец поступления на сервер и зависимости в рамках запросов.

Реакция системы на стимул может характеризоваться задержкой (временным промежутком между поступлением стимула и реакцией на него со стороны системы), предельным сроком обработки (к примеру, в системе управления двигателем топливо должно воспламениться в тот момент, когда цилиндр находится в определенном положении), пропускной способностью системы (например, количество транзакций, которое система способна обработать за одну секунду), неустойчивостью реакции (диапазоном задержек), количеством событий, не обработанных вследствие перегруженности системы, а также количеством потерянных по этой причине данных.

Обратите внимание — то обстоятельство, является ли система сетевой или автономной, здесь не учитывается. То же самое (пока что) можно сказать о конфигурации системы и потреблении ею ресурсов. Эти вопросы находятся в зависимости от ряда архитектурных решений, которые мы рассмотрим в главе 5.

Общие сценарии производительности

Принимая во внимание все вышеприведенные соображения, мы можем приступить к разбору элементов общего сценария производительности, пример которого показан на рис. 4.5. «В нормальном режиме пользователи инициируют по 1000 непериодических транзакций в минуту; задержка при их обработке занимает в среднем две секунды».

- *Источник стимула.* Стимулы могут поступать либо из внешних (иногда многочисленных), либо из внутренних источников. В нашем примере источником стимула являются пользователи.
- *Стимул.* В качестве стимулов выступают поступающие события. Образец поступления может быть периодическим, непериодическим или случайным. В нашем примере стимулом является непериодическое инициирование 1000 транзакций в минуту.
- *Артефакт.* В роли артефакта всегда выступают предоставляемые системой услуги — так происходит и в нашем примере.

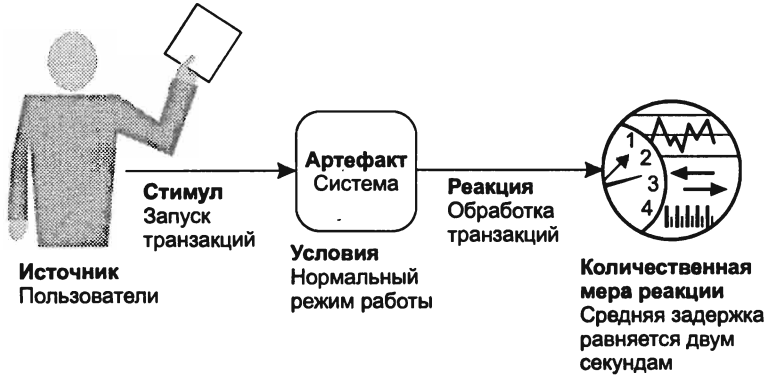


Рис. 4.5. Пример сценария производительности

Таблица 4.3. Общий сценарий производительности

Элемент сценария	Возможные значения
Источник	Ряд независимых источников, часть из которых может находиться внутри системы
Стимул	Периодическое, непериодическое или случайное поступление событий
Артефакт	Система
Условия	Нормальный режим; перегруженный режим
Реакция	Обработка стимулов; изменение уровня обслуживания
Количественная мера реакции	Задержка, предельный срок, пропускная способность, неустойчивость, коэффициент неудач, потеря данных

- *Условия.* Система может находиться в разных рабочих режимах — в частности, нормальном, аварийном или перегруженном. В нашем примере система находится в нормальном режиме.
- *Реакция.* Система должна обрабатывать поступающие события. В результате этих действий условия могут поменяться (например, система может перейти из нормального в перегруженный режим). В нашем примере выполняется обработка транзакций.
- *Количественная мера реакции.* Среди возможных единиц измерения — продолжительность обработки поступающих событий (задержка или предельный срок обработки события), разброс вариантов продолжительности (неустойчивость), количество событий, которые система способна обработать за определенный период времени (пропускная способность), а также характеристика событий, обработка которых невозможна (коэффициент неудач, потеря данных). В нашем примере предполагается обработка транзакций со средней задержкой в две секунды.

Элементы общих сценариев производительности представлены в табл. 4.3.

На протяжении большей части истории программной инженерии производительность оставалась ведущим фактором создания архитектуры. В таком качестве она зачастую препятствовала реализации остальных атрибутов качества. По мере стремительного падения соотношения цены и производительности аппаратного оборудования и параллельного повышения издержек на разработку программного обеспечения серьезную конкуренцию производительности стали составлять другие атрибуты качества.

Безопасность

Безопасность отражает способность системы противостоять попыткам несанкционированного доступа при одновременном обслуживании легальных пользователей. Попытка нарушения системы защиты называется атакой[1], а существует она в нескольких разновидностях. Иногда эти попытки направлены на получение

[1] Некоторые специалисты по безопасности в качестве синонима «атаки» употребляют термин «угроза».

доступа к данным и услугам или изменение данных, а иной раз — на воспрепятствование обслуживанию легальных пользователей.

Содержание атак, которые иногда удостаиваются широкого освещения в средствах массовой информации, варьируется от хищений путем электронных денежных переводов до модифицирования уязвимых данных, от получения номеров кредитных карт до уничтожения файлов в компьютерных системах и атак типа «отказ от обслуживания», проводимых с помощью червей и вирусов. Тем не менее в общем сценарии безопасности используются те же элементы, что и в других общих сценариях: стимул, источник стимула, условия, цель атаки, предполагаемая реакция системы и количественная мера реакции.

Безопасность можно определить как атрибут качества системы, предусматривающий строгое выполнение обязательств, конфиденциальность, целостность, гарантирование, готовность и аудит. Для каждой из этих характеристик мы приведем определение и пример.

1. Строгое выполнение обязательств — это свойство, согласно которому права на отмену транзакции (операции доступа или модификации данных или услуг) нет ни у одной участвующей стороны. Если, к примеру, вы заказали какой-то продукт в интернет-магазине, вы не можете этого отрицать.
1. Конфиденциальность — это свойство, предусматривающее защиту данных и услуг от несанкционированного доступа. Так, хакер не может прочитать вашу налоговую декларацию, хранящуюся на компьютере в соответствующем правительственном учреждении.
2. Целостность — это свойство данных и служб, в соответствии с которым они предоставляются в оговоренном виде. Если ваш преподаватель поставил вам какую-то оценку, впоследствии ее нельзя будет исправить.
3. Согласно гарантированию, участники транзакции не должны выдавать себя за сторонних лиц. Так, интернет-магазин, которому покупатели предоставляют номера своих кредитных карт, должен быть именно тем магазином, каким он себя называет, и никаким другим.
4. Готовность — это свойство системы, обеспечивающее ее открытость для законных пользователей. Покупателя не волнует, что его любимый интернет-магазин из последних сил сопротивляется DOS-атаке, — у него все равно должна быть возможность заказать ту книгу, которая ему нужна.
5. Аудит предполагает отслеживание системой всех операций на том уровне, на котором их впоследствии будет возможно реконструировать. Если, к примеру, вы переводите деньги с одного счета в швейцарском банке на другой, система зафиксирует эту операцию.

Каждой из перечисленных категорий соответствует ряд общих сценариев.

Общие сценарии безопасности

Ниже перечислены элементы общего сценария безопасности. Пример такого сценария приводится на рис. 4.6. Установленное лицо, работая на удаленном компьютере, пытается внести изменения в системные данные; при помощи данных из контрольного журнала система в течение одного дня проводит восстановление данных.

- *Источник стимула.* В роли исполнителя атаки может выступать либо человек, либо другая система. Иногда исполнитель идентифицируется (правильно или неправильно — другой разговор), в иных случаях остается неизвестным. Если исполнитель атаки имеет серьезные мотивы (например, политические) для совершения противоправных действий, то угрозы типа «мы знаем, кто ты такой, и мы тебя засудим» вряд ли приведут к желаемому результату; в таких случаях особое внимание следует уделять мотивации пользователя. Защитные мероприятия чрезвычайно осложняются, если исполнитель атаки сумел получить доступ к обширным (например, правительственным) ресурсам. Собственно атаки делятся на несколько типов: несанкционированный доступ, изменение данных и отказ от обслуживания.

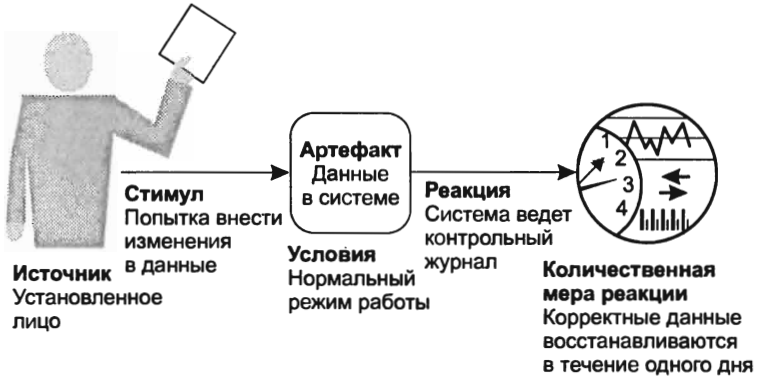


Рис. 4.6. Пример сценария безопасности

Вопрос заключается в том, как отличить нелегального пользователя от легального и как обеспечить всем тем, кто относится ко второй группе, полноценный доступ к системе? Если бы единственная цель заключалась в том, чтобы закрыть доступ к системе, эффективнее всего было бы закрыть его для всех пользователей без исключения.

- *Стимул.* Стимулом является атака или попытка подорвать надежность системы защиты. Другими словами, лицо или система без должных полномочий пытается вывести, изменить и/или удалить информацию, получить доступ к службам системы или понизить уровень готовности последних. На рис. 4.6 в роли стимула выступает намерение модифицировать данные.
- *Артефакт.* Целью атаки могут быть либо службы системы, либо хранящиеся в ней данные. В нашем примере иллюстрируется последняя ситуация.
- *Условия.* В период атаки система может находиться в оперативном или неоперативном режиме, может быть подключена или отключена от сети, защищена или не защищена брандмауэром.
- *Реакция.* В зависимости от своих задач злоумышленник может попытаться воспользоваться службами, не проходя авторизацию, не допустить до применения этих служб легальных пользователей, просмотреть уязвимые данные или изменить их. Таким образом, система должна, с одной стороны,

проводить авторизацию легальных пользователей, предоставлять им доступ к данным и службам, а с другой — отказывать неавторизованным пользователям в правах доступа и сообщать о попытках такого доступа. Система должна не только разрешать доступ легальным пользователям, но и — в некоторых случаях — предоставлять и отторгать соответствующие права. Одним из средств предотвращения атак является контрольный журнал — в нем отслеживаются все модификации и попытки доступа; с помощью этой информации можно не только восстановить измененные данные, но и привлечь злоумышленника к суду. Наличие контрольного журнала предполагается в примере на рис. 4.6.

- *Количественная мера реакции.* Реакция системы должна оцениваться по параметрам сложности подготовки разного рода атак и сложности восстановления и противодействия этим атакам. В нашем примере с помощью контрольного журнала счетá, с которых злоумышленник похитил денежные средства, восстанавливаются в своем исходном состоянии. Злоумышленника еще надо поймать, поскольку деньги остаются у него, однако эта задача выходит за рамки возможностей вычислительной системы.

Варианты составления общего сценария безопасности представлены в табл. 4.4.

Таблица 4.4. Составление общего сценария безопасности

Элемент сценария	Возможные значения
Источник	Лицо или система: ✦ правильно/неправильно идентифицированная, неидентифицированная; ✦ внутреннего/внешнего происхождения, авторизованная/неавторизованная; ✦ обладающая доступом к ограниченным/многочисленным ресурсам
Стимул	Попытка вывести данные, изменить/удалить данные, получить доступ к системным службам, снизить готовность системных служб
Артефакт	Системные службы; данные, хранящиеся в системе
Условия	Оперативный/неоперативный режим, наличие/отсутствие подключения к сети; наличие или отсутствие брандмауэрной защиты
Реакция	Аутентификация пользователя; сокрытие личных данных пользователя; блокирование доступа к данным и/или службам; разрешение доступа к данным и/или службам; предоставление или отзыв полномочий на доступ к данным и/или службам; регистрация доступа/изменений и попыток получить доступ/изменить данные/службы в соответствии с личными данными пользователя; хранение данных в нечитаемом формате; выявление необъяснимо высокой потребности в обслуживании, оповещение об этом пользователя или другой системы, ограничение готовности служб
Количественная мера реакции	Временные/трудовые/иные ресурсы, необходимые для того, чтобы успешно обойти систему защиты; вероятность обнаружения атаки; вероятность идентификации лица, выступившего исполнителем атаки или получившего доступ/модифицировавшего данные и/или службы; процент служб, готовность которых может быть сохранена в условиях атаки типа «отказ от обслуживания»; восстановление данных/служб; степень повреждения данных/служб и/или прецедентов отказа в правах доступа легальным пользователям

Контролепригодность

Контролепригодность (testability) программного продукта выражает его способность к демонстрации неисправностей путем тестирования (как правило, контрольного прогона). По меньшей мере 40% стоимости разработки качественно спроектированных систем связано с тестированием. Следовательно, попытки программных архитекторов снизить эту стоимость не лишены смысла.

В частности, контролепригодность основывается на предположении о том, что, если у программного продукта есть хотя бы одна неисправность, она проявится во время *следующего* контрольного прогона. Рассчитать такую вероятность весьма сложно, поэтому, добравшись до количественной меры реакции, мы приведем другие единицы измерения.

Для надежного тестирования системы требуется возможность: 1) *контроля* внутреннего состояния и входных данных каждого из ее компонентов и 2) *наблюдения* за ее выходными данными. Довольно часто для этих целей используются специализированные тестовые программы (test harness), проводящие испытания тестируемого программного обеспечения. К примеру, для данных, записанных посредством нескольких интерфейсов, может потребоваться проверить возможность считывания, а для двигателя — целый отсек.

Тестирование — заключительная операция ряда этапов жизненного цикла продуктов — проводится силами ряда разработчиков, тестировщиков, верификаторов и пользователей. Тестированию, в зависимости от ситуации, подвергаются участки кода, проектное решение или даже вся система. Количественная мера реакции в случае с контролепригодностью зависит от того, насколько эффективно тесты позволяют обнаруживать неисправности и как долго должно проводиться тестирование для того, чтобы достичь желаемого покрытия.

Общие сценарии контролепригодности

Пример сценария контролепригодности, а конкретнее тестирования отдельного блока, представлен на рис. 4.7. «Тестировщик блоков проводит тестирование готового компонента системы с интерфейсом, обеспечивающим контроль поведения и наблюдаемость выходных данных; 85-процентное покрытие путей достигается за три часа».

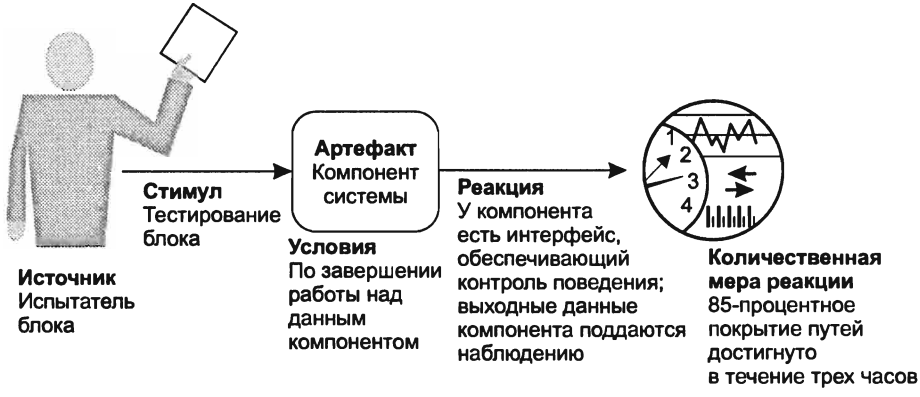


Рис. 4.7. Пример сценария контролепригодности

- *Источник стимула*. Тестирование может проводиться тестировщиком блоков, сборки или системы, а также клиентом. Проверка проектного решения иногда доверяется сторонним разработчикам. В нашем случае тестирование проводится тестировщиком.
- *Стимул*. Стимулом к тестированию выступает завершение одного из этапов процесса разработки — например, анализа или проектирования, кодирования класса, окончательной сборки подсистемы или даже разработки системы в целом. В нашем примере тестирование проводится после завершения работы над блоком кода.
- *Артефакт*. Артефактом может быть проектное решение, участок кода или вся система. В нашем примере тестируется блок кода.
- *Условия*. Тестирование может проводиться в периоды проектирования, разработки, компиляции или размещения. В примере на рис. 4.7 тест проводится во время разработки.
- *Реакция*. Поскольку контролепригодность тесно связана с наблюдаемостью и контролируемостью, реакция, в идеале, должна предусматривать возможности контроля над системой с целью проведения предполагаемых тестов и наблюдения реакций на каждый из них. В нашем примере и то и другое возможно.
- *Количественная мера реакции*. В качестве единиц измерения используются такие показатели, как процент исполнения исполняемых операторов, длина наибольшей цепочки зависимостей (дает представление о сложности тестирования) и вероятность обнаружения дополнительных неисправностей. На рис. 4.7 приводится процент покрытия исполняемых операторов.

Варианты составления общего сценария контролепригодности приводятся в табл. 4.5.

Таблица 4.5. Составление общего сценария контролепригодности

Элемент сценария	Возможные значения
Источник	Разработчик блока Сборщик элементов Верификатор системы Приемщик со стороны заказчика Пользователь системы
Стимул	Завершение этапов анализа, составления архитектуры, проектирования, кодирования класса, сборки подсистемы; сдача системы
Артефакт	Часть проектного решения, участок кода, целое приложение
Условия	В периоды проектирования, разработки, компиляции, размещения
Реакция	Предоставление доступа к значениям состояния; возврат вычисленных значений; подготовка среды тестирования
Количественная мера реакции	Процент исполнения исполняемых операторов Вероятность отказа в случае появления неисправности Продолжительность тестирования Длина наибольшей из всех выявленных по результатам тестирования цепочек зависимостей Продолжительность подготовки среды тестирования

Практичность

Практичность выражает степень сложности выполнения пользователем стоящей перед ним задачи и тип реализованного в системе механизма помощи пользователю. У этого атрибута качества несколько сторон.

- *Изучение возможностей системы.* Каким образом можно облегчить задачу пользователя, связанную с изучением незнакомой системы или одного из ее аспектов?
- *Эффективное использование системы.* Каким образом система способствует повышению эффективности работы с ней?
- *Минимизация последствий ошибок.* Что система может сделать для того, чтобы ошибка, допущенная пользователем, не привела к серьезным последствиям?
- *Адаптация системы к потребностям пользователя.* Что может сделать пользователь (или система), чтобы выполнить его задачу стало проще?
- *Доверие и удовлетворение пользователя.* Как система может убедить пользователя в том, что он предпринял правильное действие?

В последние пять лет активно изучались вопросы взаимосвязи между практичностью и программной архитектурой (см. врезку «Практичность: были не правы, каемся»). Как правило, проблемы, связанные с практичностью, в процессе разработки выявляются путем макетирования и пользовательского тестирования. Чем позже выявляется проблема и чем ниже в архитектурной иерархии находится ее решение, тем труднее, с точки зрения временных и финансовых ограничений, с ней разобраться. Излагая примеры сценариев, мы намерены сосредоточиться на тех аспектах практичности, которые в наибольшей степени влияют на характер архитектуры. Следовательно, в правильности составления этих сценариев необходимо убедиться еще до реализации архитектурного решения — в противном случае над этим придется работать в процессе пользовательского тестирования и макетирования.

Общие сценарии практичности

Пример сценария практичности приводится на рис. 4.8. «Пользователь, желающий свести к минимуму последствия допущенной ошибки, пытается отменить системную операцию в момент ее исполнения; отмена занимает менее одной секунды». Общие сценарии практичности состоят из следующих элементов.

- *Источник стимула.* В роли источника стимула всегда выступает конечный пользователь.
- *Стимул.* Стимул заключается в желании конечного пользователя добиться эффективного применения возможностей системы, изучить способы управления операциями системы, минимизировать последствия ошибок, адаптировать систему к выполнению стоящих перед ним задач или обеспечить удобство пользования системой. В нашем примере пользователь намерен отменить операцию — это один из вариантов минимизации последствий ошибок.
- *Артефакт.* Артефактом всегда является система.

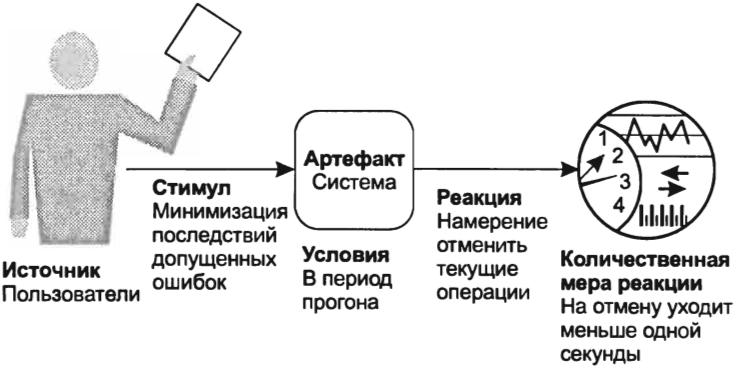


Рис. 4.8. Пример сценария практичности

ПРАКТИЧНОСТЬ: БЫЛИ НЕ ПРАВЫ, КАЕМСЯ (ИЛИ «ЭТО НЕ ПО-АРХИТЕКТУРНОМУ»)

Примерно пять лет назад ряд уважаемых специалистов в области программной инженерии сделали весьма смелое публичное заявление:

Для того чтобы сделать пользовательский интерфейс четким и простым в применении, требуется тщательно наладить механизм взаимодействия пользователя с системой... «впрочем, эти вещи не имеют отношения к архитектуре».

К нашему несчастью, этих специалистов звали Басс, Клеменц и Кацман, а книга называлась «Архитектура программных систем: практическое пособие, 1-е издание». За пять последующих лет мы узнали много нового о разных атрибутах качества, но больше всего мы поднаторели в вопросах практичности.

Мы всегда утверждали, что качество системы напрямую зависит от качества архитектуры; впрочем, наша аргументация по этому поводу в первом издании временами страдала. Несмотря ни на что, за прошедшие пять лет не случилось ничего такого, что смогло бы поколебать нашу уверенность в жесткой взаимосвязи качества архитектуры и системы. Все свидетельства говорят в пользу этого утверждения, и практичность в этом смысле не исключение. Действительно, многие вопросы практичности завязаны на архитектуру. Более того — те характеристики практичности, реализовать которые труднее всего (в частности, те, которые почти нереально воплотить в жизнь после того, как система уже сконструирована), наиочевиднейшим образом демонстрируют связь с архитектурой.

Если мы хотим, чтобы пользователь имел возможность прервать исполняемую операцию и, таким образом, вернуться к состоянию системы, зафиксированному до ее начала, мы должны прописать эту возможность в архитектуре. То же самое нужно сделать, чтобы позволить пользователю аннулировать результаты предыдущего действия или проинформировать его о состоянии текущей операции. Подобных примеров жуть как много.

К чему мы все это говорим? Утверждать, что тот или иной атрибут качества или ряд его аспектов носят неархитектурный характер, проще всего. Не все имеет отношение к архитектуре, это правда, однако довольно часто наши выводы по этому вопросу основываются на поверхностном анализе проблемы. Стоит копнуть чуть глубже, и свидетельства о связи с архитектурой не заставят себя долго ждать. И горе тому архитектору (равно как и авторам книг по архитектуре!), который их не заметит.

— RK

- *Условия.* Действия пользователя, имеющие отношение к практичности, всегда приходятся на периоды прогона или конфигурирования системы. Все действия, зафиксированные до наступления этих периодов, считаются произведенными разработчиками. Несмотря на то что пользователь и разра-

ботчик может оказаться одним и тем же лицом, мы всегда разделяем эти две роли. В примере на рис. 4.8 отмена операции производится в период прогона.

- *Реакция.* Система должна либо предоставлять пользователю все необходимые средства, либо уметь предвидеть его потребности. В нашем примере отмена операции производится по желанию пользователя, а система восстанавливается в предшествующем состоянии.
- *Количественная мера реакции.* Реакция в данном случае измеряется продолжительностью выполнения задачи, количеством ошибок, количеством разрешенных проблем, степенью удовлетворения пользователя, повышением уровня знаний пользователя, процентным отношением успешно проведенных операций к общему числу операций, а также количеством времени/данных, потерянных из-за ошибки. Согласно примеру с рис. 4.8, отмена операции занимает не более одной секунды.

Варианты составления общих сценариев практичности представлены в табл. 4.6.

Таблица 4.6. Составление общих сценариев практичности

Элемент сценария	Возможные значения
Источник	Конечный пользователь
Стимул	Намерение: изучить возможности системы; достичь эффективного применения этих возможностей; минимизировать последствия ошибок; адаптировать систему к своим потребностям; обеспечить удобство пользования системой
Артефакт	Система
Условия	В период прогона или конфигурирования
Реакция	Система демонстрирует один или несколько вариантов реакции из числа нижеследующих: ✦ для «изучения возможностей системы»: система справки контекстозависима; интерфейс системы оказывается знакомым пользователю; интерфейс оказывается удобным при работе в незнакомом контексте; ✦ для «эффективного использования системы»: агрегирование данных и/или команд; многократное применение ранее введенных данных и/или команд; эффективная навигация на экранном пространстве; ряд четких представлений с логичными операциями; комплексная система поиска; возможность выполнять несколько действий одновременно; ✦ для «минимизации последствий ошибок»: возврат к предыдущему состоянию (undo), отмена, восстановление после отказа системы, обнаружение допущенных пользователем ошибок и их исправление, извлечение забытых паролей, контроль системных ресурсов; ✦ для «адаптации системы»: настраиваемость по требованиям; локализация; ✦ для «обеспечения удобства пользования системой»: вывод информации о состоянии системы; проведение операций в соответствии с темпом работы пользователя

продолжение

Таблица 4.6 (*продолжение*)

Элемент сценария	Возможные значения
Количественная мера реакции	Длительность выполнения задачи, количество ошибок, количество разрешенных проблем, степень удовлетворения пользователя, повышение образовательного уровня пользователя, процентное отношение успешно завершенных операций к их общему числу, объем потерянных времени/данных

Формулировка понятий в общих сценариях

Одно из преимуществ общих сценариев заключается в том, что они позволяют налаживать общение с заинтересованными лицами. Мы уже обращали ваше внимание на то, что специалисты, в зависимости от того атрибута качества, на который они ориентируются, употребляют разные термины для обозначения одних и тех же базовых понятий и явлений. Это иногда приводит к непониманию. К примеру, в ходе обсуждения производительности предполагаемой системы заинтересованному лицу-пользователю может просто не прийти в голову, что задержка реакции на события непосредственным образом затрагивает его интересы. Четкое формулирование подобных вещей способствует принятию архитектурных решений, в особенности если речь идет о решениях компромиссного характера.

Таблица 4.7. Стимулы атрибутов качества

Атрибут качества	Стимул
Готовность	Непредвиденное событие, непоступление ожидаемого события
Модифицируемость	Требование по добавлению/удалению/изменению/видоизменению функциональности, платформы, атрибута качества, мощности
Производительность	Периодический, непериодический, случайный
Безопасность	Попытка просмотреть, откорректировать, изменить/удалить информацию, получить доступ к системным службам или понизить уровень их готовности
Контролепригодность	Завершение этапа разработки системы
Практичность	Намерение изучить возможности систем, добиться эффективного использования системы, минимизировать последствия допущенных ошибок, подстроить систему под свои потребности, обеспечить удобство пользования системой

В табл. 4.7 представлены стимулы, характерные для всех рассмотренных атрибутов, а также ряд понятий. Некоторые стимулы фиксируются в период прогона, иные поступают раньше. Задача архитектора состоит в том, чтобы понять, какие из этих стимулов на деле обозначают одно и то же явление, какие связаны с другими стимулами, а какие независимы. Когда в отношениях между ними наступит ясность, архитектор сможет донести эти стимулы до различных заинтересованных лиц, изложив их так, чтобы все всё поняли. Представить отношения между стимулами обобщенно невозможно — связано это с тем, что отчасти они зависят от конкретных условий. Событие, связанное с производительностью, может носить элементарный характер, а может быть связано с другими явлениями, от-

носящимися к более низкому уровню; отказ может быть причиной одного или нескольких событий производительности. В частности, к отказу может привести обмен рядом сообщений между клиентом и сервером (завершившийся непредвиденным сообщением), хотя каждое из этих событий, с точки зрения производительности, является элементарным.

4.5. Другие атрибуты качества системы

Итак, мы провели обобщенный анализ атрибутов качества. В классификациях атрибутов, исследовательской литературе и учебниках по программной инженерии упоминается ряд других атрибутов, которые частично отражены в наших сценариях. К примеру, во многих случаях существенное значение имеет масштабируемость (scalability); в нашем обзоре этот атрибут качества учитывается как модификация мощности системы — количества пользователей, которые могут работать в ней одновременно. Переносимость (portability) представлена как изменение платформы.

Если для вашей организации заметную роль играет какой-либо атрибут качества из числа неупомянутых — например, способность к взаимодействию, — для него также имеет смысл составить общий сценарий. Для этого нужно лишь наполнить содержанием шесть универсальных элементов сценария: источник стимула, стимул, условия, артефакт, реакция и количественная мера реакции. Если речь идет о способности к взаимодействию, в роли стимула можно представить потребность во взаимодействии с другой системой, в роли реакции — создание нового интерфейса или нескольких интерфейсов, а в роли единицы измерения (количественной меры) реакции — степень сложности в категориях времени, количества изменяемых интерфейсов и т. д.

4.6. Коммерческие атрибуты качества

Помимо атрибутов качества, экстраполируемых непосредственно на систему, существует ряд коммерческих (business) задач качества, которые также нередко оказывают существенное влияние на характер системной архитектуры. Эти задачи связаны со стоимостью, планированием, выходом на рынок и другими вопросами сбыта. Все они не менее абстрактны, чем вышеперечисленные атрибуты качества системы, и поэтому в целях влияния на процесс проектирования и обеспечения возможности тестирования их также следует конкретизировать при помощи сценариев. Впрочем, составление конкретных сценариев мы, пожалуй, доверим вам (см. соответствующий дискуссионный вопрос).

- *Срок выхода продукта на рынок.* При наличии серьезной конкуренции, а также при условии ограниченности времени, в течение которого система или продукт имеют шансы на успех, существенное значение приобретает продолжительность разработки. Это, в свою очередь, приводит к потребности в приобретении или повторном использовании существующих элементов.

Для сокращения сроков выхода продукта на рынок часто используются готовые элементы наподобие коммерческих коробочных продуктов (commercial off-the-shelf products, COTS) или элементы из предшествующих проектов. Возможность вставки или размещения в данной системе подмножества сторонней системы зависит от декомпозиции системы на элементы.

- *Стоимость и прибыль.* Совершенно естественно, что под разработку системы составляется бюджет, который желательно не превышать. Стоимость разработки напрямую зависит от конкретной архитектуры. К примеру, если архитектура основывается на технологиях (или специальных технологических знаниях), которыми компания-разработчик не располагает, на ее реализацию уйдет больше средств, чем если бы было принято решение об использовании освоенных технологий. Архитектура с повышенными требованиями к гибкости, как правило, оказывается дороже негибкой (хотя впоследствии ее будет дешевле сопровождать и модифицировать).
- *Предполагаемый срок службы системы.* Чем больше намеченный срок службы системы, тем выше требования к модифицируемости, масштабируемости и переносимости. С другой стороны, встраивание дополнительной инфраструктуры (например, дополнительного уровня, обеспечивающего возможность переносимости), как правило, увеличивает срок выхода продукта на рынок. Впрочем, у продукта с возможностью модифицирования и расширения больше шансов продержаться на рынке в течение длительного времени.
- *Целевой сегмент рынка.* Если речь идет об универсальном (массовом) программном обеспечении, потенциальный объем рынка определятся набором платформ и функций. Таким образом, чем больше внимания уделяется переносимости и функциональности, тем больше доля рынка. Определенную роль в этом контексте играют и другие атрибуты качества — в частности, производительность, надежность и практичность. В случаях, когда компания планирует выйти на масштабный рынок с рядом родственных продуктов, лучше всего подходит стратегия линейки продуктов с общим для всех систем ядром (которое, кстати, зачастую обеспечивает переносимость), вокруг которого строятся разные программные уровни — чем дальше, тем специфичнее. Методика построения линеек программных продуктов рассматривается в главе 14.
- *График развертывания.* Если продукт планируется сначала выпустить в базовом варианте, а затем дополнять новыми возможностями, на первый план выходят гибкость и настраиваемость архитектуры. В частности, систему следует конструировать с расчетом на удобство расширения и сокращения.
- *Интеграция с существующими системами.* Если новую систему планируется интегрировать с существующими системами, следует обратить особое внимание на фиксацию механизмов интеграции. Этот атрибут качества, несомненно важный с точки зрения маркетинга, имеет непосредственное отношение к архитектуре. В частности, в течение предыдущего десятиле-

тия многие корпорации стремились к тому, чтобы интегрировать существующие системы с HTTP-серверами и, таким образом, обеспечить доступ к ним через Интернет. Вряд ли нужно напоминать, что все архитектурные ограничения, связанные с интеграцией, требуют тщательного анализа.

4.7. Атрибуты качества архитектуры

Помимо атрибутов качества системы и атрибутов качества, связанных с экономической ситуацией, существуют атрибуты качества, имеющие непосредственное отношение к характеру самой архитектуры; их реализация не менее важна. Мы поговорим о трех таких атрибутах качества и, как и раньше, оставим составление конкретных сценариев на вашей совести — соответствующее задание сформулировано в разделе «Дискуссионные вопросы».

Концептуальная целостность — это фундаментальная идея, или представление, которое объединяет проектное решение системы на всех его уровнях. Одни и те же задачи архитектура должна выполнять одними и теми же способами. Особое внимание на концептуальную целостность системы обращает, в частности, Фред Брукс (Fred Brooks) — по его мнению, без этого система обречена на провал:

> Я продолжаю утверждать, что концептуальная целостность есть наиболее важный из всех факторов проектирования системы. Пусть лучше в системе не будет каких-то необычных функций и исправлений, но она должна выражать единый набор конструкторских решений — это значительно полезнее, чем нагромождать большое количество самостоятельных и не связанных друг с другом идей. [Brooks 75]

Брукс в основном имел в виду представление систем в глазах пользователей, однако его мысли в равной степени справедливы в отношении архитектурного плана. Ценность идей Брукса о концептуальной целостности для конечных пользователей аналогична ценности архитектурной целостности для других заинтересованных групп — в частности, для разработчиков и специалистов по сопровождению.

В части 3 настоящей книги мы будем говорить об оценке архитектуры; этот процесс предполагает наличие в составе разработчиков проекта архитектора. Отсутствие такового свидетельствует о том, что концептуальной целостности в проекте нет.

Правильность и *завершенность* — это те атрибуты качества, которые обеспечивают реализацию требований к системе и соответствие ресурсным ограничениям периода прогона. Формальная оценка, рассматриваемая в части 3, позволяет понять, насколько архитектура правильна и закончена.

Возможность построения — это атрибут качества, позволяющий завершить работу над системой, ограничившись доступными трудовыми ресурсами, уложившись во временные рамки и оставляя возможность для изменений в процессе разработки. Он выражает удобство конструирования заданной системы. Среди архитектурных методов, позволяющих его достичь, следует упомянуть тщательную декомпозицию на модули, разумное распределение этих модулей между группами разработчиков, а также ограничение зависимостей между модулями

(и, следовательно, между рабочими группами). Цель заключается в том, чтобы максимально задействовать параллелизм во время разработки.

Поскольку возможность построения, как правило, оценивается в категориях стоимости и временных ограничений, между этим атрибутом качества и различными моделями затрат существует устойчивая связь. Тем не менее возможность построения значительно сложнее типичных моделей затрат. Система строится из определенных материалов, которые создаются различными средствами. К примеру, пользовательский интерфейс строится на основе соответствующего набора инструментов (элементов управления), которыми оперирует разработчик интерфейса. Элементы управления в данном случае — это материалы, а разработчик — инструмент; таким образом, одним из аспектов возможности построения является соответствие между предполагаемыми материалами, с одной стороны, и инструментами, при помощи которых с ними производятся разного рода манипуляции, — с другой. Еще один аспект возможности построения — четкое представление о задаче, которую требуется решить. Логическое обоснование этого аспекта предусматривает сокращение сроков выхода продукта на рынок и поощрение вложений в изучение и конструирование нового понятия со стороны потенциальных поставщиков. Проект, предусматривающий решение на основе хорошо изученных понятий, таким образом, обладает большими возможностями построения, чем проект, внедряющий новые понятия.

4.8. Заключение

Программные архитекторы чаще всего стремятся реализовать те атрибуты качества, которые мы представили в этой главе. Поскольку в определениях существует некоторая неразбериха, мы решили описать их при помощи общих сценариев. Анализу подверглись атрибуты качества из числа системных, коммерческих и архитектурных атрибутов качества.

В следующей главе мы рассмотрим конкретные архитектурные методики реализации атрибутов качества.

4.9. Дополнительная литература

Обзор общих сценариев и отображения сценариев, выявленных в ходе оценки архитектуры, на общие сценарии содержится в издании [Bass 01b]. Детальные исследования готовности опубликованы в работах [Laprie 89] и [Cristian 93]. Вопросы безопасности рассматриваются в публикации [Ramachandran 02]. Взаимоотношениям между практичностью и программной архитектурой посвящены исследования [Gram 96] и [Bass 01a].

В издании [McGregor 01] приводится анализ контролепригодности. [Paulish 02] рассматривает процент стоимости разработки, связанной с тестированием.

Общепризнанные определения атрибутов качества содержатся в стандартах IEEE [ISO 91]. [Witt 94] обсуждает желательные атрибуты качества архитектуры (и архитекторов).

4.10. Дискуссионные вопросы

1. Каковы важнейшие атрибуты качества системы, над которой вы в данный момент работаете? Как выглядят системно-ориентированные сценарии, в которых эти атрибуты качества зафиксированы, и общие сценарии, из которых они произведены?
2. Брукс утверждает, что концептуальная целостность есть ключ к созданию успешных систем. Вы с ним согласны? Можете ли вы привести успешные системы, которые обходились без этого атрибута качества? Если таковые имеются, то за счет каких факторов они стали успешными? Как можно оценить систему на предмет ее соответствия заветам Брукса?
3. Составьте сценарии коммерческих и архитектурных атрибутов качества, приведенных в разделах 4.4 и 4.5. Учли ли вы в своих сценариях все эти атрибуты качества? Какие из них труднее всего зафиксировать при помощи сценариев?

Глава 5
Реализация качества

(в соавторстве с Феликсом Бахманом, Марком Кляйном и Биллом Вудом[1])

> Любой атрибут качества, пусть самый положительный, в изоляции от других атрибутов качества не приносит ничего, кроме вреда.
>
> — *Ралф Уолдо Эмерсон*

В главе 4 мы рассмотрели ряд атрибутов качества систем. Иллюстрировать материал мы предпочли с помощью сценариев. Представление о содержании атрибута качества позволяет выявить требования к качеству, однако абсолютно ничего не говорит о том, как их следует реализовывать. Сейчас мы поговорим именно об этом. Мы представим архитектурные методы реализации всех шести атрибутов качества, проанализированных в главе 4. Все возможные атрибуты качества нам охватить не удастся; впрочем, имейте в виду, что тактика обеспечения интегрируемости приводится в главе 8.

Основной упор мы сделаем на том, как архитекторы реализуют конкретные атрибуты качества. Требования к качеству регламентируют действия программных средств, позволяющие выполнить определенные коммерческие задачи. Нас в основном интересуют *тактики* (tactics), при помощи которых архитекторы вырабатывают проектные решения, обращаясь при этом к образцам проектирования, архитектурным образцам и архитектурным стратегиям. Предположим, к примеру, что коммерческая задача состоит в создании линейки продуктов. Для достижения этой цели в отдельных классах функций требуется реализовать изменчивость.

Прежде чем принимать решения относительно образцов, с помощью которых можно будет достичь изменчивости, архитектор должен выбрать для себя ряд

---

[1] Билл Вуд (Bill Wood) — старший научный сотрудник Института программной инженерии.

тактик модифицируемости — дело в том, что именно они обусловливают принятие архитектурных решений. Архитектурные образцы и стратегии реализуют совокупность тактик. Связь между требованиями по атрибутам качества (о них мы говорили в главе 4) и архитектурными решениями и является предметом настоящей главы.

5.1. Определение тактики

Что придает одному решению свойство переносимости, другому — высокую производительность, а третьему — интегрируемость? Реализация этих атрибутов качества связана с принятием фундаментальных проектных решений. Мы намерены в подробностях рассмотреть эти проектные решения, которые по-другому называются *тактиками* (tactics). Тактика — это проектное решение, которое влияет на управление реакцией по атрибуту качества. Совокупность тактик называется архитектурной стратегией (architectural strategy) — о ней речь пойдет в главе 12. Упаковка тактик в составе архитектурных образцов рассматривается в разделе 5.8.

Проект системы представляет собой совокупность решений. Некоторые из них помогают контролировать реакции по атрибутам качества; иные обеспечивают функциональность систем. В этом разделе речь пойдет о решениях по атрибутам качества, называемым тактиками. Связь тактики и реакции по атрибуту качества изображена в виде схемы на рис. 5.1. Тактики используются архитекторами многие годы; мы намерены выделить и описать их. Имейте в виду — мы ничего не изобретаем; мы просто фиксируем методы практической деятельности архитекторов.

Каждая тактика для архитектора — это проектная альтернатива. Предположим, к примеру, что одна из возможных тактик повышает готовность системы за счет реализации резервирования. Повышение готовности — это лишь одна из многочисленных альтернатив, среди которых архитектор волен выбирать. Как правило, повышение готовности через резервирование сопровождается синхронизацией (которая нужна для того, чтобы в случае повреждения оригинала можно было обратиться к резервной копии). Из этого примера можно сделать два вывода.

1. *Одна тактика может использоваться для уточнения другой.* Мы назвали резервирование тактикой. В свою очередь, можно различать «резервирование данных» (в системе баз данных) или «резервирование вычислений» (во встроенной системе управления). И то и другое — тактики. Проектировщик может пойти еще дальше и путем уточнения максимально конкретизировать резервирование. В отношении всех атрибутов качества, которые нам предстоит рассмотреть, мы намерены структурировать тактики в рамках иерархических систем.
2. *Упаковка тактик в образцы.* Любой образец (паттерн), призванный обеспечивать готовность, вероятнее всего, предусматривает совместное применение тактик резервирования и синхронизации. Кроме того, они в нем, наверное, конкретизированы. Ближе к завершению данного раздела мы представим пример образца, описанного в категориях тактик.

Рис. 5.1. Тактики предназначены для управления реакциями на стимулы

Тактики атрибутов качества систем мы предпочитаем систематизировать в рамках иерархических систем; необходимо при этом иметь в виду, что любая иерархия демонстрирует лишь часть тактик — завершенных перечней тактик просто не бывает. Мы приведем тактические методики реализации каждого из шести атрибутов, рассмотренных в главе 4 (готовность, модифицируемость, производительность, безопасность, контролепригодность и практичность). Для каждого мы представим вариант систематизации тактик и их краткое описание. Систематизация нужна для того, чтобы архитектор имел возможность поиска нужных тактик.

5.2. Тактики реализации готовности

Не забыли терминологию готовности, приведенную в главе 4? Отказом называется ситуация, при которой система теряет возможность обслуживания в соответствии со своей спецификацией; кроме того, признаки отказа заметны пользователям системы. Предвестником отказа является неисправность (или сочетание неисправностей). Мы также говорили о том, что одним из важнейших аспектов готовности является восстановление (устранение неисправности). Тактика, описание которой содержится в этом разделе, не позволяет неисправностям превращаться в отказы или, по меньшей мере, ограничивает последствия неисправности, обеспечивая возможность восстановления. Иллюстрация этой тактики приводится на рис. 5.2.

Рис. 5.2. Задача тактик готовности

Многие из перечисленных здесь тактик применяются в стандартных средах исполнения, каковыми являются операционные системы, серверы приложений и системы управления базами данных. Иметь достаточное представление о применяемых тактиках важно для того, чтобы последствия их применения можно было учитывать в ходе проектирования и оценки. Обо всех методиках поддержания готовности можно сказать, что они в том или ином виде предусматривают резервирование — одни посредством средств диагностики, позволяющих обнаруживать отказы, иные — через средства их восстановления. В некоторых случаях диагностика и восстановление проводятся автоматически, в других — вручную.

В первую очередь, мы рассмотрим обнаружение неисправностей. Затем перейдем к восстановлению после неисправностей и, наконец, к их предотвращению.

Обнаружение неисправностей

Из всех тактик обнаружения неисправностей наибольшим признанием пользуются три: ping/echo-пакеты, heartbeat-запросы и исключения.

- *Ping/echo-пакеты.* Проверяющий компонент отправляет проверяемому компоненту ping-запрос, ожидая через установленный период времени получить в ответ echo-пакет. Эта схема используется в группах компонентов, которые выполняют в отношении друг друга одну и ту же задачу. Кроме того, она может применяться клиентами, которые при помощи таких пакетов удостоверяются в приемлемой работоспособности объекта на сервере и канала связи. Детекторы неисправностей по схеме ping/echo можно расположить в рамках иерархии, согласно которой детектор нижнего уровня должен отправлять ping-запросы тем программным процессам, с которыми у него общий процессор, а детекторы верхних уровней обязаны запрашивать нижележащие уровни. В результате, по сравнению с ситуацией, при которой удаленный детектор неисправностей отправляет ping-запросы всем процессам, наблюдается экономия рабочей пропускной способности.
- *Heartbeat (таймер безопасности).* В таком случае один из двух компонентов периодически отправляет heartbeat-сообщение, а другой — ожидает его получения. Если отправки heartbeat-сообщения не происходит, компонент, который должен был его отправить, считается вышедшим из строя, о чем оповещается компонент устранения неисправностей. В составе heartbeat-сообщений, помимо прочего, можно передавать данные. К примеру, банкомат может периодически отправлять на сервер записи о последних транзакциях. Такой пакет одновременно выступает в качестве heartbeat и сообщает некоторую информацию.
- *Исключения.* Один из методов выявления неисправностей подразумевает использование исключений, которые порождаются при обнаружении любого из классов неисправностей, рассмотренных в главе 4. Обработчики исключений, как правило, исполняются в том процессе, в котором исключение появилось.

Тактики ping/echo-пакетов и heartbeat-сообщений применимы к ряду отдельных процессов, а тактика исключений, напротив, распространяется на единичный процесс. Обработчик исключений обычно выполняет семантическое преобразование неисправностей, тем самым переводя их в такую форму, в которой их можно обработать.

Восстановление после неисправности

Процесс восстановления после неисправности состоит из двух частей: подготовки к восстановлению и собственно восстановления. Ниже представлены некоторые тактики восстановления.

- *Голосование.* Процессы, исполняемые на резервированных процессорах, принимают равнозначные исходные данные, а затем, вычислив простое выходное значение, отправляют его голосующему. Если голосующий выявляет у одного из процессоров отклонение от нормы, тот отключается. Алгоритм голосования может действовать по «мажоритарному принципу», ориентироваться на «предпочтительный алгоритм» или руководствоваться какими-либо другими правилами. Этот метод исправляет неисправности в работе алгоритмов или отказы процессора и чаще всего встречается в системах управления. Если все процессоры используют одни и те же алгоритмы, выявляются только неисправности процессоров — неисправности алгоритмов не обнаруживаются. Таким образом, если последствия отказа чрезвычайно серьезны — например, не исключают возможность полного выхода из строя, — есть смысл использовать разнообразные резервируемые компоненты.

  Разнообразие иногда доходит до крайности — например, когда программное обеспечение каждого резервируемого компонента разрабатывается разными рабочими группами и исполняется на несходных платформах. Более обычной является ситуация, при которой на разных платформах разрабатывается один и тот же программный компонент. Разнообразие в разработке и сопровождении слишком дорого обходится, и поэтому применяется оно только в исключительных ситуациях — например, при отслеживании земной поверхности в авиационных системах. В системах управления разнообразие применяется в тех случаях, когда выходные данные, предоставляемые голосующему, просты и легко поддаются классификации по признаку эквивалентности или отклонения, вычисления проводятся циклически и все резервируемые компоненты получают от датчиков равнозначные входные данные. В случае отказа разнообразие исключает простой, поскольку голосующий продолжает работать. Среди вариантов этой методики следует упомянуть симплекс-метод, предполагающий применение результатов «предпочтительного» компонента во всех случаях, кроме тех, когда они отклоняются от результатов «доверенного» компонента, с показаниями которого голосующий сверяется. Если резервируемые компоненты проводят параллельные вычисления с одним и тем же набором входных значений, их (компонентов) синхронизация проводится автоматически.

- *Активное резервирование (горячий перезапуск)*[1]. Все резервируемые компоненты реагируют на события параллельно. Следовательно, все они находятся в одном и том же состоянии. В работу идет только один отклик (обычно берется отклик первого среагировавшего компонента), остальные отклоняются. В случае неисправности такая тактика не позволяет простою затянуться более чем на несколько миллисекунд — связано это с тем, что резервная копия постоянно находится в том же состоянии, что и действующий компонент, и поэтому время уходит только на переключение между

[1] Излагаемая в книге классификация методов резервирования отличается от предложенной в отечественном стандарте ГОСТ 27.002-89 и отражает американскую терминологию в теории надежности систем. — *Примеч. науч. ред.*

ними. Резервирование замещением часто применяется в схеме клиент–сервер — в частности, в системах управления базами данных, где оперативность отклика требуется даже в ситуации неисправности. В распределенных системах с повышенными требованиями к готовности резервируются даже каналы связи. Например, желательно, чтобы в локальной сети было несколько параллельных каналов и в каждом из них находилось по одному резервному компоненту. В таком случае, даже если произойдет отказ канала или моста, компоненты системы останутся в состоянии готовности.

Синхронизация предполагает, что любые сообщения, предназначенные для резервных компонентов, должны быть отправлены всем таким компонентам. В случае, если существует вероятность разрыва связи (например, вследствие шумов на линии передачи данных или ее перегрузки), в целях восстановления задействуется надежный протокол передачи. Такой протокол требует от получателей пакетов подтверждения их приема и предоставления неких значений, свидетельствующих о соблюдении целостности данных — например, контрольной суммы. Если отправителю не удается убедиться в том, что сообщение дошло до всех получателей, он отбирает те компоненты, которые не подтвердили прием, и отправляет им сообщение повторно. Повторная отправка непринятых сообщений (в некоторых случаях — по разным каналам связи) продолжается до тех пор, пока отправитель не приходит к выводу, что получатель вышел из строя.

- *Пассивное резервирование (теплый перезапуск/двойное резервирование/тройное резервирование).* Один из компонентов (первичный) реагирует на поступающие события и сообщает другим компонентам (запасным) о том, каким образом им требуется обновить состояние. В случае неисправности система в первую очередь проверяет актуальность состояния резервных копий и только после этого возобновляет обслуживание. Эта методика применяется в системах управления — как правило, в тех случаях, когда по каналам связи или от датчиков приходят входные данные, которые при обнаружении неисправности требуется перенаправить на резервный компонент. В частности, она используется в системе управления воздушным движением, рассматриваемой в главе 6. В этой системе решение о том, когда следует заменить первичный компонент, принимается вторичным компонентом; в других системах принятие таких решений происходит в рамках других компонентов. Настоящая тактика делает ставку на надежность резервных компонентов. Регулярное проведение необходимых переключений — например, раз в сутки или раз в неделю — способствует повышению готовности системы. В некоторых системах управления базами данных переключение производится с сохранением каждого нового элемента данных. Новый элемент сохраняется на теневой странице, а старая страница становится резервной. В таком случае простой, как правило, ограничивается секундами.

  Проведение синхронизации входит в обязанности первичного компонента, который для выполнения этой задачи может отправлять вторичным компонентам элементарные широковещательные пакеты.

- *Резерв.* Согласно этой тактике, путем настройки резервной вычислительной платформы, используемой для замещения, обеспечивается возможность замены сразу нескольких разнородных поврежденных компонентов. Ее следует перезагрузить в расчете на конкретную конфигурацию программных средств, а в случае отказа — инициализировать ее состояние. Для перевода резерва в необходимое состояние нужно периодически фиксировать контрольные точки системы и регистрировать все изменения состояния, сохраняя эти данные в устройстве, обеспечивающем достаточную устойчивость. В таком качестве часто используется вспомогательная рабочая станция-клиент, к которой пользователь может обратиться в случае отказа. Простой в случае применения этой тактики, как правило, исчисляется минутами.

Некоторые тактики восстановления предусматривают повторное введение компонентов. Так, после устранения отказа в резервном компоненте его можно вновь ввести в систему. Среди такого рода тактик следует упомянуть затенение, повторную синхронизацию состояния и откат.

- *Затенение.* Компонент, в котором недавно произошел отказ, может после восстановления некоторое время работать в «теневом режиме» — таким образом обеспечивается полное соответствие его поведения поведению работающих компонентов, после чего он снова вводится в действие.
- *Повторная синхронизация состояния.* При пассивном или активном резервировании требуется, чтобы восстанавливаемый компонент перед введением в действие обновлял свое состояние. Действенность этой методики зависит от допустимой продолжительности простоя, объема обновления и количества сообщений, необходимых для его проведения. Прежде всего предпочтение отдается обновлению, для выполнения которого требуется отправить одно сообщение. Инкрементные обновления состояния с периодами обслуживания между этапами способствуют усложнению программного продукта.
- *Контрольные точки/откат.* Контрольные точки, фиксирующие устойчивые состояния, создаются либо с определенной периодичностью, либо в качестве реакции на конкретные события. Иногда отказы системы происходят нестандартным образом и сопровождаются явно неустойчивыми состояниями. В таком случае систему следует восстановить при помощи предыдущей контрольной точки устойчивого состояния и журнала транзакций, происшедших с момента ее фиксации.

Предотвращение неисправностей

Ниже приводится ряд тактик предотвращения неисправностей.

- *Снятие с эксплуатации.* Эта тактика предполагает наложение запрета на функционирование компонента системы для проведения с ним мероприятий, направленных на предотвращение прогнозируемых отказов. В частности, в целях недопущения отказов из-за утечек памяти компонент можно перезагрузить. Если снятие с эксплуатации проводится автоматически, для

его проведения имеет смысл разработать архитектурную стратегию. Если снятие проводится вручную, средства, обеспечивающие такую возможность, должны присутствовать в системе.

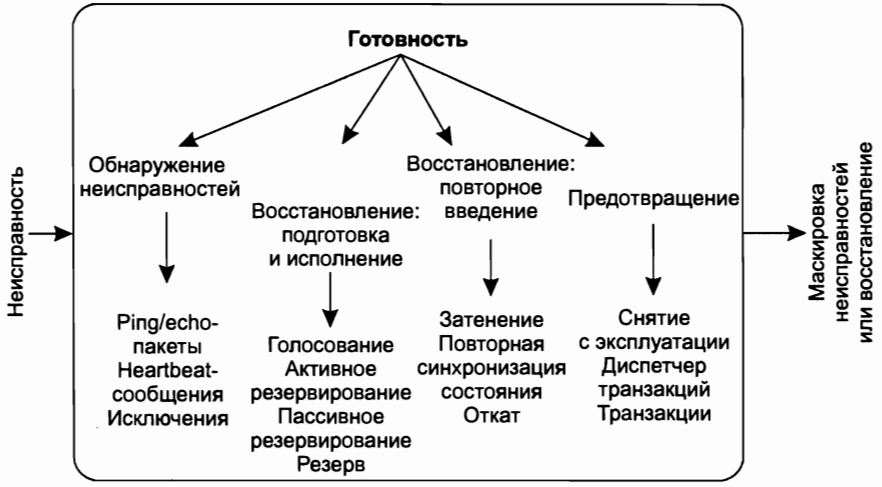


Рис. 5.3. Схема тактик готовности

- *Транзакции.* Транзакцией называется связка нескольких последовательных операций, которую можно аннулировать одним действием. Транзакции используются для сохранения данных в ситуациях, когда один из этапов процесса оборачивается отказом, а также для предотвращения конфликтов между несколькими одновременными потоками, обращающимися к одним и тем же данным.
- *Диспетчер процессов.* После обнаружения неисправного процесса диспетчерский процесс может удалить его и, согласно тактике резервирования, создать новый экземпляр, инициализированный с приемлемым состоянием.

Все рассмотренные тактики изображены на схеме (рис. 5.3).

5.3. Тактики реализации модифицируемости

Еще в главе 4 мы говорили о том, что основной задачей тактик управления модифицируемостью является контроль над временем и стоимостью реализации, тестирования и размещения изменений. На рис. 5.4 эта взаимосвязь изображена графически.

Тактики модифицируемости классифицируются по наборам, в соответствии с теми задачами, которые перед ними поставлены. Один из таких наборов ориентирован на уменьшение количества модулей, на которые изменения воздействуют напрямую. Мы называем этот набор «локализацией изменений». Перед другим набором ставится задача по ограничению изменений локализованных модулей.

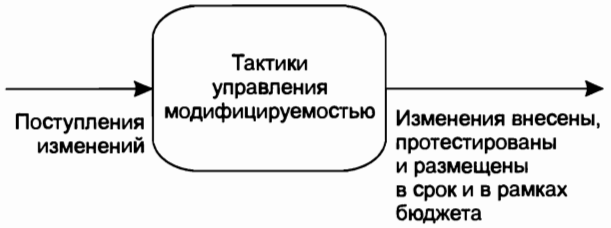


Рис. 5.4. Назначение тактик модифицируемости

Таким образом, он предотвращает наступление «волнового эффекта». Такое разделение подразумевает существование модулей, находящихся под непосредственным воздействием изменений (речь идет о тех модулях, чьи обязанности при внесении изменений корректируются), и модулей, подпадающих под это воздействие косвенно (это те модули, чьи обязанности остаются без изменений, но реализация меняется в соответствии с коррективами, внесенными в модули первой группы). Еще один набор тактик ориентирован на контроль продолжительности и стоимости размещения. Его мы называем «отложенным периодом связывания».

Локализация изменений

Наличие четкой взаимосвязи между количеством модулей, на которые воздействует ряд изменений, и стоимостью реализации этих изменений весьма сомнительно, однако, как правило, ограничение на модификацию ограниченного набора модулей приводит к снижению стоимости. Назначение тактик из этого набора состоит в том, чтобы в период проектирования распределить между модулями обязанности с расчетом на ограничение области распространения ожидаемых изменений. Мы знаем пять тактик подобного рода.

- *Обеспечение семантической связности.* Под семантической связностью имеются в виду отношения между обязанностями модуля. Цель заключается в том, чтобы обеспечить совместное исполнение всех обязанностей и одновременно свести зависимость от других модулей к минимуму. Для достижения этой цели следует выбрать обязанности, которые характеризуются семантической связностью. На ее измерение направлены метрики сцепления и связности, однако контекст внесения изменений они не учитывают. По этой причине семантическую связность следует оценивать исходя из набора ожидаемых изменений. В этой связи следует упомянуть такой подвид рассматриваемой тактики, как *общие абстрактные службы* (abstract common services). Предоставление общих служб посредством специализированных модулей, как правило, приравнивается к обеспечению возможности многократного применения. Это действительно так, однако не стоит забывать, что абстрагирование общих служб также обеспечивает модифицируемость. Если общие службы абстрагированы, то внесение в них изменений производится единовременно и не затрагивает те модули, которые к этим службам обращаются. Более того, модификация модулей посредством такого рода служб не оказывает никакого влияния на пользователей. Таким образом, рассматриваемая тактика не только предусматривает ло-

кализацию изменений, но и предотвращает волновой эффект. В качестве примеров абстрагирования общих служб можно упомянуть применение каркасов приложений и других промежуточных программных средств.

- *Прогнозирование ожидаемых изменений.* Сложно оценить конкретный вариант распределения обязанностей без учета совокупности ожидаемых изменений. Основной вопрос, на который в данном случае требуется ответить, можно сформулировать так: «Ограничивает ли предложенная декомпозиция выбор модулей, которые нужно будет откорректировать для выполнения данной модификации?» С ним связан другой вопрос: «Могут ли разнородные (в своей основе) изменения воздействовать на одни и те же модули?» В чем же здесь отличие от семантической связности? При распределении обязанностей на основе семантической связности предполагается, что ожидаемые изменения будут семантически связными. Тактика прогнозирования ожидаемых изменений ориентирована не на связность обязанностей модуля, а на минимизацию последствий изменений. Практическое применение этой тактики без связки с другими тактиками весьма затруднительно, поскольку спрогнозировать все изменения невозможно. По этой причине она, как правило, задействуется совместно с тактикой семантической связности.
- *Обобщение модуля.* Чем более общий характер носит модуль, тем шире диапазон функций, которые он может рассчитывать на основе входных данных. Правомерно утверждать, что входные данные задают язык модуля, сложность которого может варьировать от представления констант в качестве входных параметров до реализации модуля в виде интерпретатора и представления входных параметров в качестве программы на его языке. Чем более обобщен модуль, тем вероятнее, что предполагаемые изменения можно будет внести посредством настройки входного языка, не прибегая к модификации модуля.
- *Ограничение возможных альтернатив.* Модификации, особенно если они проводятся в линейке продуктов (см. главу 14), иногда приводят к далеко идущим последствиям и, следовательно, воздействуют на многочисленные модули. Таким образом, чем меньше возможных альтернатив, тем менее заметно влияние модификаций. К примеру, параметр изменчивости в линейке продуктов может предусматривать модификацию процессора. Ограничение изменений, вносимых в процессор членами его семейства, уменьшает количество возможных альтернатив.

Предотвращение волнового эффекта

Волновым эффектом модификации называется необходимость внесения изменений в модули, которые напрямую с ней не связаны. К примеру, если для проведения модификации какие-то коррективы вносятся в модуль A, то необходимость внесения изменений в модуль B обусловливается только тем, что они внесены в модуль A. Модуль B приходится модифицировать вследствие его зависимости от A.

Анализ волнового эффекта имеет смысл начать с обзора возможных типов зависимостей между модулями. Таких типов всего восемь:

1. Синтаксис
   - ◊ *данных*. Для того чтобы обеспечить правильность компиляции (или исполнения) B, тип (формат) данных, производимых A и потребляемых B, должен согласовываться с типом (или форматом) данных, принятым B.
   - ◊ *служб*. Для того чтобы обеспечить правильность компиляции или исполнения B, сигнатура служб, предоставляемых A и вызываемых B, должна согласовываться с допущениями, принятыми B.
2. Семантика
   - ◊ *данных*. Для того чтобы обеспечить правильность исполнения B, семантика данных, производимых A и потребляемых B, должна согласовываться с допущениями, принятыми B.
   - ◊ *служб*. Для того чтобы обеспечить правильность исполнения B, семантика служб, предоставляемых A и применяемых B, должна согласовываться с допущениями, принятыми B.
3. Последовательность
   - ◊ *данных*. Для того чтобы обеспечить правильность исполнения B, прием этим модулем данных, произведенных A, должен проводиться в установленной последовательности. К примеру, заголовок пакета данных должен приниматься раньше, чем тело пакета (в отличие от протоколов, которые встраивают порядковые номера в данные).
   - ◊ *управления*. Для того чтобы обеспечить правильность исполнения B, требуется предшествующее (в рамках определенных временных ограничений) исполнение A. Скажем, между исполнением A и исполнением B должно пройти не более 5 мс.
4. *Индивидуальность интерфейса A*. У A может быть несколько интерфейсов. Для того чтобы обеспечить правильность компиляции и исполнения B, индивидуальность (имя или дескриптор) соответствующего интерфейса A должна согласовываться с допущениями, принятыми B.
5. *Локализация A (в период прогона)*. Для того чтобы обеспечить правильность исполнения B, местоположение A в период прогона должно согласовываться с допущениями, принятыми B. К примеру, B может предположить, что A находится в другом процессе того же процессора.
6. *Качество услуг/данных, предоставляемых A*. Для того чтобы обеспечить правильность исполнения B, некое свойство, связанное с качеством предоставляемых A данных и услуг, должно согласовываться с допущениями, принятыми B. К примеру, для того чтобы алгоритмы B вычисляли правильные результаты, показания точности некоего датчика должны характеризоваться определенной степенью точности.
7. *Существование A*. Для того чтобы обеспечить правильность исполнения B, A должен существовать. К примеру, если B отправляет объекту A запрос на обслуживание, в то время как A не существует и его нельзя создать динамическим способом, B не сможет корректно функционировать.

8. *Ресурсное поведение A.* Для того чтобы обеспечить правильность исполнения B, ресурсное поведение A должно согласовываться с допущениями, принятыми B. Вариантов всего два — это либо использование ресурса (A использует ту же память, что и B), либо принадлежность ресурса (B резервирует ресурс, который принадлежит A).

Обрисовав отличительные черты различных типов зависимостей, мы можем смело переходить к обзору тактик предотвращения волнового эффекта для каждого из этих типов.

Следует иметь в виду, что ни одна из нижеприведенных тактик не гарантирует предотвращения волны семантических изменений. Начнем с перечисления тактик, ориентированных на интерфейсы конкретных модулей (информационная закрытость и обслуживание существующих интерфейсов), а затем поговорим о тактике, способной разрывать цепочки зависимостей, — применении посредника.

- *Информационная закрытость.* Информационной закрытостью называется декомпозиция обязанностей по отношению к сущности (системе или ее произвольной декомпозиции) на мелкие элементы и разделение информации на приватную и публичную. Публичные обязанности осуществляются через указанные интерфейсы. Задача состоит в том, чтобы изолировать изменения в рамках одного модуля и не допустить их распространения на другие модули. Это старейшая методика такого рода. Она напрямую связана с «прогнозированием ожидаемых изменений», поскольку эти изменения в ней закладываются в основу декомпозиции.
- *Обслуживание существующих интерфейсов.* Если B зависит от имени или сигнатуры интерфейса A, то за счет обслуживания этого интерфейса и его синтаксиса B остается без изменений. Результативность применения этой тактики при наличии семантической зависимости B от A гарантировать нельзя, поскольку замаскировать изменения данных и служб довольно сложно. Не менее сложно замаскировать зависимости от качества данных и услуг, использования и принадлежности ресурсов. Для стабилизации интерфейса имеет смысл отделить его от реализации. В результате появляется возможность создания абстрактных интерфейсов, маскирующих вариации. Вариации можно заключить в рамки существующих обязанностей или заменить одну реализацию модуля другой.

  Среди образцов, реализующих эту тактику, необходимо упомянуть следующие.

  - ◊ *Введение новых интерфейсов.* Большинство языков программирования позволяют создавать несколько интерфейсов. Новые видимые службы и данные можно открывать через новые интерфейсы — в таком случае существующие интерфейсы обходятся без изменений и сохраняют за собой старые сигнатуры.
  - ◊ *Введение нового адаптера.* При дополнении A новым адаптером он помещает A в оболочку, сохраняя для нее сигнатуру оригинала.
  - ◊ *Введение заглушки A.* Если для выполнения модификации A требуется удалить и при этом B зависит исключительно от сигнатуры A, тогда

путем создания заглушки А внесение изменений в В можно предотвратить.

- *Ограничение каналов связи.* Предполагает ограничение числа модулей, совместно с которыми данный модуль пользуется данными — другими словами, сокращение тех модулей, которые потребляют данные, производимые рассматриваемым модулем, а также тех, которые производят потребляемые им данные. Поскольку зависимости производства/потребления сопряжены с волной, действия такого характера способны уменьшить волновой эффект. Образец, в котором реализована эта тактика, представлен в главе 8 («Моделирование условий полета»).
- *Введение посредника.* Если модуль В характеризуется какой-либо зависимостью от модуля А, за исключением семантической, то между ними можно поместить посредника, ответственного за выполнение действий, связанных с этой зависимостью. Существует множество посредников с разными именами, и мы намерены разобрать их в соответствии с перечисленными типами зависимостей. Как и прежде, при наличии семантических связей посредник не может гарантировать результативность. Типы посредников таковы:
  - *Данные (синтаксис).* Репозитарии (как пассивные, так и «доски объявлений») играют роль посредников между производителем и потребителем данных. Они способны проводить преобразования синтаксиса производства А в форму, применение которой допускает В. Некоторые образцы публикации/подписки (предполагающие поток данных через центральный компонент) могут проводить аналогичные операции. Образцы «модель–представление–контроллер» (Model–View–Conroller, MVC) и «представление–абстракция–управление» (Presentation–Abstraction–Control, PAC) преобразуют данные из одного формализма (устройства ввода или вывода) в другой (применяемый моделью для MVC или абстракцией для PAC).
  - *Службы (синтаксис).* Образцы «фасад», «мост», «посредник», «стратегия», «агент» и «фабрика» — все они предлагают посредничество при выполнении задачи по преобразованию синтаксиса служб из одной формы в другую. Следовательно, они все предотвращают распространение изменений из модуля А в модуль В.
  - *Индивидуальность интерфейса А.* Для маскировки изменений индивидуальности интерфейса применяется образец «брокер». Если В зависит от индивидуальности интерфейса А и эта идентичность подвергается изменениям, ее следует передать брокеру — он установит связь с новой индивидуальностью А, вследствие чего В сможет избежать модификации.
  - *Местоположение А (в период прогона).* Сервер имен позволяет изменять местоположение А без корректировки В. При этом А обязан регистрировать на сервере имен свое текущее местоположение, а В — получать с этого сервера соответствующие сведения.

- ◊ *Ресурсное поведение A или ресурса, управляемого A*. Посредник, ответственный за распределение ресурсов, называется диспетчером ресурсов. Некоторые из них (например, диспетчеры, существующие в системах реального времени и основанные на монотонном анализе интенсивности) способны в пределах определенных рамок гарантировать выполнение любых запросов. При этом A, естественно, должен передать управление ресурсами диспетчеру.
- ◊ *Существование A*. Образец «фабрика» при необходимости может создавать экземпляры, и именно его действиями удовлетворяется зависимость B от существования A.

Откладывание связывания

Две категории тактик, которые мы на данный момент успели рассмотреть, ориентированы на сокращение количества модулей, требующих корректировки для реализации намеченных модификаций. Тем не менее два элемента наших сценариев модифицируемости — продолжительность размещения и предоставление возможности проведения модификаций лицам, не относящимся к группе разработчиков, — не получают своего разрешения за счет сокращения количества модулей. Тактика откладывания связывания решает эту задачу, правда, ценой введения дополнительной инфраструктуры.

Связывание решений с существующей системой можно проводить в разные периоды. Мы рассмотрим только те из них, которые оказывают влияние на размещение. Размещение системы обусловливается тем или иным процессом. Если разработчик выполняет модификацию, то временной промежуток между этим действием и моментом, когда результаты модификации становятся доступны конечному пользователю, как правило, определяется продолжительностью процессов тестирования и распространения. Связывание в период прогона предполагает подготовленность системы к связыванию и завершение этапов тестирования и распространения. Откладывание периода связывания, помимо всего прочего, позволяет конечному пользователю или администратору системы проводить настройку и предоставлять входные данные, влияющие на поведение.

Многие тактики — нижеприведенные в их числе — оказывают свое действие в периоды загрузки или исполнения.

- ✦ *Регистрация в период прогона* обеспечивает функционирование согласно стандарту Plug-and-Play, однако с осуществлением контроля регистрации связываются дополнительные издержки. В частности, регистрацию по образцам публикации/подписки можно реализовать при прогоне или при загрузке.
- ✦ *Конфигурационные файлы* предназначены для установки параметров при запуске.
- ✦ *Полиморфизм* предоставляет возможность отложенного связывания вызовов методов.
- ✦ *Замена компонентов* позволяет проводить связывание в период загрузки.

- *Применение предписанных протоколов* обеспечивает возможность связывания независимых процессов в период прогона.

Схема тактик модифицируемости приводится на рис. 5.5.

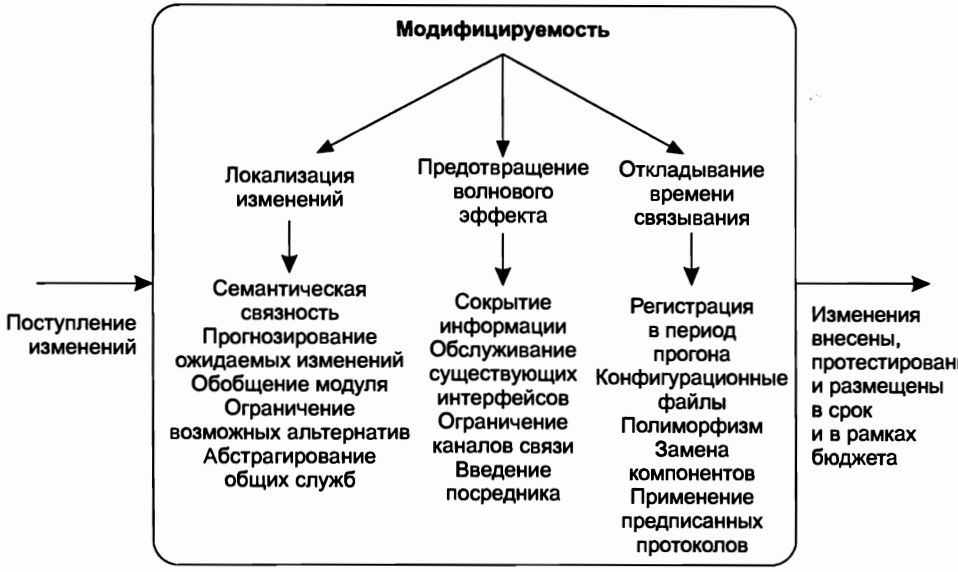


Рис. 5.5. Схема тактик модифицируемости

5.4. Тактики реализации производительности

Еще в главе 4 мы сформулировали назначение тактик реализации производительности — оно заключается в реагировании на поступающее в систему событие в течение определенного времени. Событие может быть единичным, а может быть частью потока, но оно в любом случае инициирует запрос на проведение вычислений. Примеры событий — поступление сообщения, истечение периода времени, обнаружение значимого изменения состояния в окружении системы и т. д. Система обрабатывает эти события и генерирует отклик (реакцию). Тактики производительности позволяют контролировать период времени, отводимый на реагирование (рис. 5.6). Задержкой называется временной промежуток между поступлением события и появлением реакции.

Рис. 5.6. Назначение тактик производительности

После поступления события система приступает к его обработке; по тем или иным причинам обработка может быть заблокирована. Отсюда делаем вывод о двух основных слагаемых времени отклика — это потребление ресурсов и продолжительность блокирования.

1. *Потребление ресурсов.* Ресурсы — это центральный процессор, хранилища данных, пропускная способность сетевых соединений и память; кроме того, в категорию ресурсов попадают некоторые сущности, задаваемые конкретной проектируемой системой. К примеру, речь может идти об управлении буферами и обеспечении последовательного доступа к критическим секциям. Каждому из перечисленных типов событий соответствует собственный цикл обработки. К примеру, сообщение генерируется одним компонентом, а затем через сетевое соединение поступает другому компоненту. Затем оно размещается в буфере, каким-то образом преобразуется (согласно терминологии, принятой рабочей группой OMG, это преобразование называется маршалингом), обрабатывается по определенному алгоритму, опять преобразуется в выходную форму, помещается в выходной буфер и, наконец, отправляется очередному компоненту, системе или пользователю. Общая задержка обработки события складывается из задержек всех этапов.
2. *Продолжительность блокирования.* В ходе проведения вычислений доступ к ресурсу может быть заблокирован; поводами к блокированию могут быть состязание за ресурс, его неготовность или зависимость данного вычисления от результатов других вычислений, которые еще не завершены.
   - ◊ *Состязание за ресурсы.* На рис. 5.6 приводится схема поступающих в систему событий. Поступать они могут одним или несколькими потоками. Если ряд потоков или несколько событий в составе одного потока соперничают за право обращения к одному и тому же ресурсу, происходит задержка. Как правило, чем серьезнее состязание за ресурс, тем больше вероятность возникновения задержки. С другой стороны, это зависит от арбитража и способа обработки механизмом арбитража отдельных запросов.
   - ◊ *Готовность ресурсов.* Если состязания не наблюдается, но требуемый ресурс недоступен, вычисление останавливается. Причинами неготовности ресурса могут быть, к примеру, его неоперативное состояние или сбой в компоненте. В любом случае, одна из задач архитектора заключается в том, чтобы выявить ситуации недоступности ресурсов, которые потенциально способны существенно увеличить общую задержку.
   - ◊ *Зависимость от результатов других вычислений.* Задержка вычисления иногда обусловливается необходимостью синхронизации с результатами других вычислений или ожиданием результатов инициированного вычисления. К примеру, если при получении информации из двух разных источников они считываются последовательно, задержка будет больше, чем если бы они считывались параллельно.

Принимая во внимание все вышеприведенные обстоятельства, мы переходим к рассмотрению трех новых категорий тактик: потреблению, управлению и арбитражу ресурсов.

Потребление ресурсов

Источниками потребления ресурсов являются потоки событий. Характеристиками потребления являются временной интервал между событиями в потоке ресурсов (регулярность запросов в рамках потока) и объем потребления ресурса при каждом запросе.

Согласно одной из тактик, для того чтобы сократить задержку, следует уменьшить количество ресурсов, необходимых для обработки потока событий. Достижению этой цели служат несколько методов.

- *Повышение вычислительной эффективности.* На одном из этапов обработки события или сообщения применяется тот или иной алгоритм. Чем эффективнее алгоритмы на важнейших участках, тем меньше задержка. В некоторых случаях события могут выбирать между несколькими ресурсами. К примеру, промежуточные данные, с одной стороны, хранятся в репозитарии, а с другой, в зависимости от временных и пространственных ресурсов, могут регенерироваться. Эта тактика чаще всего задействуется в отношении процессоров, однако в применении к другим ресурсам — например, к диску — она не менее эффективна.
- *Сокращение издержек вычислений.* В отсутствие запросов на ресурс потребность в обработке уменьшается. В главе 17 мы покажем, что по сравнению с удаленным вызовом методов (Remote Method Invocation, RMI) Java-классы предъявляют более мягкие требования относительно передачи информации. Применение посредников (играющих важную роль в контексте реализации модифицируемости) повышает уровень потребления ресурсов при обработке потоков событий; соответственно, отказ от посредничества способствует уменьшению задержки. Это классический пример компромисса между модифицируемостью и производительностью.

В соответствии с другой тактикой для сокращения задержки требуется уменьшить количество обрабатываемых событий. Достижению этой цели служат два метода.

- *Уменьшение частоты поступления событий.* Снижения потребления ресурсов, помимо прочего, можно добиться путем уменьшения частоты опроса переменных среды. Иногда такая возможность появляется после реконструкции системы. В других случаях неоправданно высокая частота опроса применяется для организации между несколькими потоками гармонических периодов. Другими словами, за счет учащенного опроса конкретного потока или потоков событий они синхронизируются.
- *Управление частотой опроса.* Если поступление внешних событий не поддается контролю, следствием уменьшения частоты опроса запросов может быть их сокращение.

Ниже приводится ряд других тактик сокращения и контроля ресурсопотребления, предусматривающих регулирование использования ресурсов.

- *Ограничение времени исполнения* для отклика на событие. В некоторых случаях это имеет смысл. К примеру, ограничение количества итераций в ите-

ративных информационно-зависимых алгоритмах приводит именно к такому результату.

- *Ограничение длины очереди.* Имеется в виду установление максимального количества поступающих событий, которые можно помещать в очередь, и, соответственно, ресурсов, задействованных в их обработке.

Управление ресурсами

Даже если потребление ресурсов не поддается контролю, определенное воздействие на время отклика оказывают тактики управления этими ресурсами.

- *Введение параллелизма.* Параллельная обработка запросов способствует сокращению продолжительности блокирования. Параллелизм можно организовать путем обработки потоков событий в разных потоках процессов или создания дополнительных потоков для обработки разных наборов операций. Для обеспечения эффективности параллелизма требуется обоснованное распределение потоков между ресурсами (выравнивание нагрузок).
- *Ведение нескольких копий данных или вычислений.* Согласно образцу «клиент–сервер», клиент представляет собой точную копию вычисления. Такие копии сокращают состязательность, характерную для ситуации выполнения всех вычислений на центральном сервере. Тактика кэширования предполагает дублирование данных в разноскоростных или отдельных репозиториях, что также способствует сокращению состязательности. Поскольку кэшированные данные, как правило, представляют собой копию существующих данных, система должна принять на себя обязанность по обеспечению согласованности и синхронизации таких копий.
- *Увеличение ресурсов.* Сокращению задержки способствует увеличение скорости процессоров, введение новых процессоров, расширение памяти и повышение скоростных характеристик сети. Одним из факторов выбора ресурсов обычно является их стоимость, но от этого ничего не меняется — увеличение ресурсов в любом случае способствует сокращению задержек. Анализ компромиссного решения по стоимости/производительности приводится в главе 12.

Арбитраж ресурсов

Если за ресурс наблюдается состязание, составляется график его использования. Такие графики, в частности, создаются для процессоров, буферов и сетей. Задача архитектора состоит в том, чтобы выявить характеристики использования каждого ресурса и выбрать оптимальную стратегию планирования (составления графика).

Любую политику планирования можно разделить на две части: назначение приоритетов и координацию. Назначение приоритетов производится во всех без исключения политиках. Иногда приоритеты расставляются по схеме «первым пришел — первым обслужен». В других случаях приоритеты привязываются

к временным требованиям запроса или его семантической значимости. Среди критериев планирования — оптимальное использование ресурса, важность запроса, минимизация обращений к ресурсам, сокращение задержки, повышение пропускной способности, предотвращение зависания и обеспечение равноправия и т. д. Архитектор должен обладать достаточными знаниями об этих критериях, иметь в виду, что иногда они конфликтуют, и четко представлять последствия выбора той или иной тактики в контексте их удовлетворения.

Координация потока событий с высоким приоритетом возможна лишь в том случае, если соответствующий ресурс доступен. Иногда приходится прерывать обслуживание текущих пользователей ресурсов. В том, что касается прерывания обслуживания, возможны три решения: 1) прерывание может происходить в любой момент или 2) только в определенные моменты, 3) прерывание исполняемых процессов не разрешается. Ниже приводятся наиболее распространенные политики планирования.

1. *Первым пришел — первым обслужен* (FIFO). Все запросы признаются равноправными и обслуживаются по очереди. При этом возникает возможность задержки одного запроса другим, на обслуживание которого уходит слишком много времени. Если все запросы действительно равноправны, в этом нет ничего страшного, однако при наличии высокоприоритетных запросов начинаются проблемы.
2. *Планирование с фиксированным приоритетом.* Каждый источник ресурсов запрашивает для себя определенный приоритет и назначает его своим ресурсом. Эта стратегия обеспечивает повышенное качество обслуживания запросов с высоким приоритетом, но в то же время допускает произвольную продолжительность ожидания обслуживания значимых запросов с низким приоритетом. Ниже приводятся наиболее распространенные стратегии назначения приоритетов.
   - ◊ *Семантическая значимость.* Приоритет каждому потоку назначается статически, согласно некоей предметной характеристике порождающей его задачи. Этот вариант применяется в универсальных вычислительных машинах, где в качестве предметной характеристики выступает время инициирования задачи.
   - ◊ *Монотонное назначение приоритетов согласно предельным срокам.* Чем более сжатые сроки обработки потока, тем выше ему назначается приоритет (статически). Применяется при планировании потоков с различными приоритетами и сроками обработки в реальном времени.
   - ◊ *Монотонное назначение приоритетов согласно частоте.* Чем чаще поступает поток, чем выше его приоритет, назначаемый статически. Это один из вариантов монотонного назначения согласно предельным срокам, но, с другой стороны, он более известен и лучше поддерживается операционными системами.
3. *Динамическое приоритетное планирование*:
   - ◊ *Циклическое обслуживание* — стратегия планирования, согласно которой запросы сначала упорядочиваются, а затем при первой же возможности ресурсы отдаются следующему в установленном порядке запросу.

Один из вариантов циклического обслуживания называется циклическим исполнением и предполагает назначение ресурсов через фиксированные промежутки времени.

- ◊ *Приоритет коротких предельных сроков.* Распределение приоритетов среди запросов проводится согласно предельным срокам их обработки.

4. *Статическое планирование.* Стратегия циклического планирования предполагает неоперативное определение моментов прерывания и порядка распределения ресурсов между запросами.

Библиография по теории планирования приводится в разделе «Дополнительная литература» в конце главы.

Схема тактик производительности приводится на рис. 5.7.

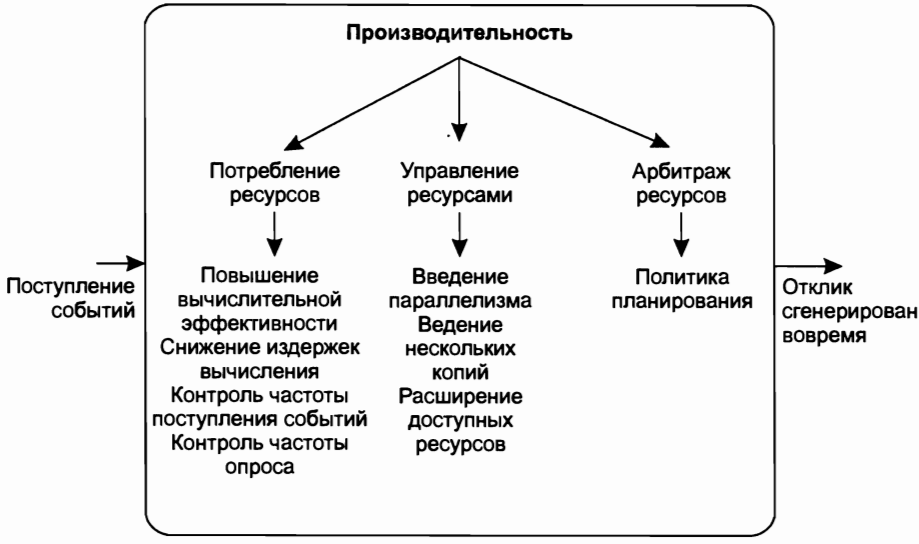


Рис. 5.7. Схема тактик производительности

5.5. Тактики реализации безопасности

Тактики реализации безопасности подразделяются на несколько подвидов: тактики противодействия атакам, тактики обнаружения атак и тактики восстановления после атак. Все эти категории в равной степени важны. Проведем бытовую аналогию: замок на двери есть средство противодействия атакам, датчик движения — средство обнаружения атаки, а наличие страховки — средство восстановления после атаки. Назначение тактик безопасности показано на рис. 5.8.

Противодействие атакам

В главе 4, характеризуя безопасность, мы обозначили в качестве целей строгое выполнение обязательств, конфиденциальность, целостность и гарантирование.

Достичь этих целей можно путем совместного применения нижеприведенных тактик.

- *Аутентификация пользователей.* Аутентификация проверяет, является ли пользователь или удаленный компьютер тем, за кого себя выдает. Средствами аутентификации являются пароли, одноразовые пароли, цифровые сертификаты и биометрические данные.

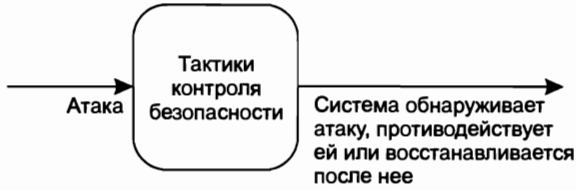


Рис. 5.8. Назначение тактик безопасности

- *Авторизация пользователей.* Авторизация проверяет наличие у аутентифицированного пользователя полномочий на получение доступа к данным или службам и их модификацию. Как правило, для проведения авторизации в системе используются образцы управления доступа. Субъектами управления доступом могут быть пользователи или классы пользователей. Последние определяются группами пользователей, ролями пользователей или списками отдельных лиц.
- *Обеспечение конфиденциальности данных.* Данные необходимо защищать от несанкционированного доступа. Как правило, для обеспечения конфиденциальности данные и каналы связи шифруются. Шифрование — это средство дополнительной защиты постоянно обслуживаемых данных, усиливающее результаты авторизации. Каналы связи же обычно вообще не контролируются средствами авторизации. Шифрование, таким образом, оказывается единственным способом защиты данных, передаваемых по открытым каналам связи. Такие каналы реализуются посредством виртуальной частной сети (virtual private network, VPN) или (в случае с веб-каналами) протокола защищенных сокетов (secure sockets layer, SSL). Шифрование бывает симметричным (обе стороны пользуются одним и тем же ключом) и ассиметричным (предусматриваются открытый и секретный ключи).
- *Обеспечение целостности.* Данные должны передаваться в неизменном виде. Содержащаяся в них служебная информация — например, контрольные суммы или результаты хэширования — может шифроваться совместно с исходными данными или независимо от них.
- *Минимизация подверженности внешним воздействиям.* В большинстве случаев для проведения атаки на все содержащиеся на хосте данные и службы злоумышленнику достаточно одной лазейки. Задача архитектора состоит в том, чтобы оптимально распределить службы между хостами.
- *Ограничение доступа.* Брандмауэры ограничивают доступ, проверяя порты отправки и назначения сообщений. Сообщения неизвестного происхожде-

ния иногда свидетельствуют об атаке. Ограничить доступ известными источниками не всегда возможно. К примеру, запросы на общедоступный вебсайт поступают из самых разных, в том числе неизвестных, источников. В таком случае имеет смысл организовать так называемую демилитаризованную зону (demilitarized zone, DMZ). DMZ открывает доступ к интернет-службам, одновременно защищая ресурсы частной сети. Она находится между общедоступной сетью и брандмауэром, рядом с внутренней сетью. В DMZ располагаются устройства, допускающие прием сообщений от произвольных источников — в частности, веб-службы, почтовые службы и службы доменных имен.

Обнаружение атак

Обычно для этой цели используются *системы обнаружения вторжений* (intrusion detection systems). Они сравнивают образцы сетевого трафика с записями своей базы данных. В случае выявления злоупотреблений данный образец трафика сравнивается с сохраненными образцами известных атак. При подозрении на аномалию образец трафика сравнивается с сохраненным образцом самого себя. Во многих случаях для сравнения пакетов их необходимо отфильтровать. Фильтрация выполняется на основе протокола, TCP-флагов, размеров полезной нагрузки, адреса источника, адреса назначения или номера порта.

Обязательными элементами систем обнаружения вторжений являются сенсоры выявления атак, диспетчеры, синтезирующие показания этих сенсоров, базы данных с сохраненными для последующего анализа событиями, средства внесетевой отчетности и анализа, а также консоль управления, при помощи которой аналитик может вносить в деятельность системы какие-то коррективы.

Восстановление после атак

Тактики восстановления после атак могут быть ориентированы на восстановление предшествующего состояния или на идентификацию исполнителя атаки (последние можно причислить как к превентивным, так и к карательным мерам).

Тактики восстановления нормального состояния системы или данных частично совпадают с тактиками готовности — и те и другие ориентированы на перевод системы из неустойчивого в устойчивое состояние. В данном случае упор делается на ведение резервных копий административных данных системы: паролей, списков контроля доступа, служб доменных имен и личных параметров пользователя.

Одна из тактик идентификации исполнителя атаки предполагает *ведение контрольного журнала* (audit trail). В контрольном журнале содержатся копии всех проведенных в системе транзакций с данными, а также идентифицирующие данные. Контрольная информация помогает проследить действия исполнителя атаки, обеспечивает строгое выполнение обязательств (в журнале содержатся свидетельства о подаче запросов) и восстановление системы. Иногда контрольные журналы сами становятся объектами атак, поэтому хранить их нужно со всеми мерами предосторожности.

Схема тактик безопасности представлена на рис. 5.9.

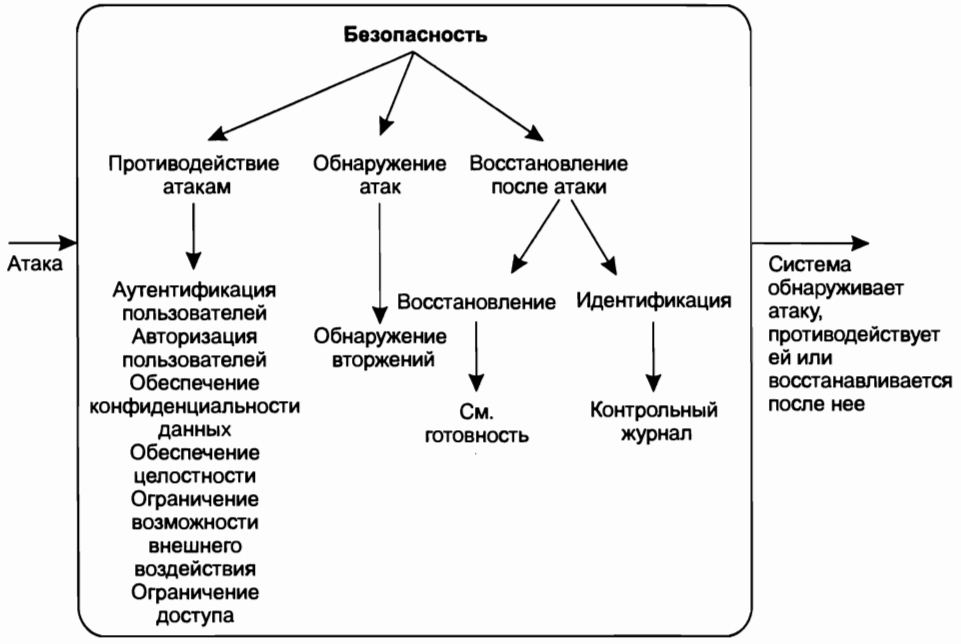


Рис. 5.9. Схема тактик безопасности

5.6. Тактики реализации контролепригодности

Тактики контролепригодности призваны упрощать тестирование при завершении того или иного этапа разработки программного продукта (рис. 5.10). Архитектурные методики повышения контролепригодности программного обеспечения известны не так широко, как их аналоги в более изученных областях модифицируемости, производительности и готовности. В пользу их развития свидетельствует то обстоятельство, что на тестирование уходит весьма значительная доля средств, выделяемых на разработку системы, а значит, любые действия архитектора, направленные на снижение этой стоимости, приобретают большую ценность.

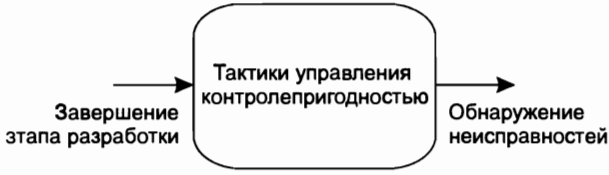


Рис. 5.10. Назначение тактик контролепригодности

В главе 4 мы представили оценку проекта как одну из методик тестирования, однако в данный момент нас интересуют только вопросы тестирования работающей системы. Любой режим тестирования направлен на обнаружение неисправ-

ностей. Для того чтобы это стало возможным, в тестируемый программный продукт следует сначала направить входные данные, а затем собрать выходные.

Отправка в тестируемый программный продукт входных и сбор выходных данных обычно осуществляются специальной тестирующей программой (test harness). Вопросы проектирования и генерации таких программ мы не рассматриваем, хотя во многих системах на это уходит много времени и средств.

Мы намерены обратить ваше внимание на две категории тактик тестирования: тактики подачи входных и сбора выходных данных и тактики внутреннего мониторинга.

Входные/выходные данные

Ниже перечислены тактики контроля входных и выходных данных.

- *Запись/считывание*. В данном случае предполагается сбор информации, проходящей через интерфейс, и ее применение тестирующей программой в качестве входных данных. Информация, проходящая через интерфейс во время нахождения системы в нормальном режиме, сохраняется в репозитарии и представляется одновременно как выходные данные одного компонента и входные данные другого. Запись этой информации позволяет генерировать для одного из них тестовые входные данные и сохранять тестовые выходные данные в расчете на последующий анализ.
- *Отделение интерфейса от реализации*. Разделение интерфейса и реализации обеспечивает возможность замены реализаций в зависимости от целей тестирования. Реализация-заглушка позволяет тестировать оставшиеся элементы системы в отсутствие компонента-оригинала. Оригинальный компонент можно заменить специализированным компонентом-тестировщиком оставшихся элементов системы.
- *Специализация путей/интерфейсов доступа*. Наличие специализированных интерфейсов тестирования позволяет собирать и специфицировать значения переменных компонента, причем проводятся эти операции либо с помощью тестирующей программы, либо независимо от нормальной работы компонента. К примеру, доступ к метаданным может осуществляться через специализированный интерфейс, через который тестирующая программа координирует свои операции. Специализированные пути/интерфейсы доступа следует отделять от путей/интерфейсов доступа, обеспечивающих обычную функциональность. Наличие в архитектуре иерархии тестовых интерфейсов обеспечивает возможность применения тестов на любом из ее уровней и наблюдения за реакцией посредством тестовых функций.

Внутренний мониторинг

Поддержка процесса тестирования со стороны отдельного компонента обеспечивается путем реализации тактик на основе его внутреннего состояния.

- *Встроенные мониторы*. Компонент может обслуживать информацию о состоянии, рабочей нагрузке, мощности, безопасности и любые другие данные,

доступ к которым можно получить через интерфейс. Интерфейс этот может быть постоянным или временным; для введения временных интерфейсов применяются, в частности, аспектно-ориентированное программирование и макрокоманды процессора. Во многих случаях события регистрируются при активизированных режимах мониторинга. Фактически, такие режимы увеличивают издержки тестирования, поскольку после их выключения снова приходится проводить тесты. С другой стороны, повышенная обозримость действий компонента, как правило, значительно перевешивает издержки, связанные с дополнительным тестированием.

Схема тактик контролепригодности приводится на рис. 5.11.

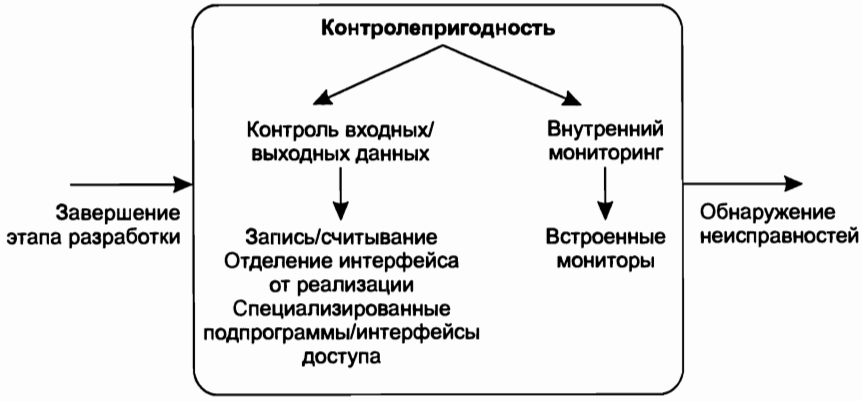


Рис. 5.11. Схема тактик контролепригодности

5.7. Тактики реализации практичности

Практичность, как мы уже говорили, выражает простоту выполнения пользователем желаемой задачи и наличие в системе вспомогательных средств, ориентированных на помощь пользователю. На реализацию практичности направлены тактики двух видов, соответствующих двум категориям «пользователей». Первая категория — тактики периода исполнения — поддерживают пользователя во время работы системы. Вторая категория основывается на итеративном характере решений, применяемых в пользовательских интерфейсах, и ориентируется на разработчиков интерфейсов, действующих в период проектирования. Она жестко связана с представленными выше тактиками реализации модифицируемости.

Назначение тактик периода исполнения показано на рис. 5.12.

Рис. 5.12. Назначение тактик реализации практичности периода исполнения

Тактики периода исполнения

Во время работы системы качество практичности можно повысить несколькими способами. Во-первых, нужно сделать так, чтобы пользователь знал, какие операции система выполняет в данный момент. Во-вторых, у пользователя должна быть возможность отдавать практичные команды из числа тех, что мы перечислили в главе 4. К примеру, команды отмены текущей и аннулирования предыдущей операции, группировки и одновременного вывода нескольких представлений помогают пользователю исправить допущенную ошибку и повысить эффективность своих действий.

При описании ролей участников (связки «человек–машина») в выполнении отдельных операций специалисты по человеко-машинному взаимодействию оперируют терминами «инициатива пользователя», «инициатива машины» и «смешанная инициатива». В тех сценариях практичности, которые мы приводили в главе 4, учитываются оба субъекта инициативы. К примеру, намереваясь отменить исполнение команды, пользователь отдает соответствующее распоряжение (проявляя «инициативу пользователя»), а система на нее реагирует. С другой стороны, во время отмены система может вывести на экран индикатор выполнения (это уже «инициатива системы»). Таким образом, операция отмены являет собой пример «смешанной инициативы». Тактики, при помощи которых архитектор составляет разного рода сценарии, можно разделить по тому же принципу — как относящиеся к инициативе пользователя и инициативе системы.

Реакция на инициативу пользователя проектируется архитектором так же, как любой другой функциональный элемент. Архитектор должен перечислить обязанности системы, связанные с реакцией на команду пользователя. Вернемся к примеру с отменой операции. В тот момент, когда пользователь отдает команду отмены, система должна находиться в состоянии ожидания ее поступления (отсюда — обязанность по содержанию постоянного приемника, устойчивого к блокированию вследствие отмены разного рода операций); затем отменяемую команду следует уничтожить, все ресурсы, задействованные при ее исполнении, — освободить; при этом компоненты, сотрудничавшие с отмененной командой, следует информировать о ее отмене, для того чтобы они смогли предпринять уместные в этом случае действия.

Если инициатива принадлежит системе, она должна располагать определенной информацией (моделью) о пользователе, задаче, которую он пытается выполнить, а также о собственном состоянии. Каждая модель предусматривает разные варианты входных данных, без которых претворить инициативу в жизнь невозможно. Тактики инициативы системы формулируют модели, с помощью которых система может прогнозировать собственное поведение или намерения пользователя. Инкапсулировав эту информацию, архитектор упрощает задачи составления и корректировки этих моделей. Составлять и корректировать модели можно либо динамическим способом — исходя из предшествующего поведения пользователя, либо непосредственно в ходе разработки.

- *Обслуживание модели задачи.* Модель задачи применяется для определения контекста, который дает системе представление о том, что пользователь намерен сделать, и возможность помочь ему. К примеру, если программе

известно, что предложения обычно начинаются с заглавных букв, она может автоматически менять регистр строчных букв после точки.

- *Обслуживание модели пользователя.* Модель пользователя содержит сведения об умении пользователя работать с системой, его представлениях о времени отклика и других аспектах, характеризующих конкретного пользователя или класс пользователей. К примеру, модель пользователя позволяет системе устанавливать определенный темп прокрутки страниц, соответствующий скорости чтения.
- *Обслуживание модели системы.* Модель системы определяет ожидаемое поведение системы с расчетом на предоставление пользователю информации о ее действиях. В частности, она прогнозирует период времени, в течение которого текущая операция должна быть завершена.

Тактики периода проектирования

В процессе тестирования пользовательские интерфейсы обычно подвергаются серьезному пересмотру. Происходит это так: специалист по практичности предоставляет разработчикам список поправок к проекту пользовательского интерфейса, а те их реализуют. В этой связи существенное значение получает уточненный вариант тактики семантической связности, относящейся к реализации модифицируемости.

- *Отделение пользовательского интерфейса от остальных элементов приложения.* Семантическая связность обосновывается необходимостью локализации ожидаемых изменений. Поскольку пользовательский интерфейс часто корректируется в процессе разработки и после ее завершения, отделение его кода помогает локализовать ожидаемые изменения. Для реализации этой тактики и обеспечения возможности модификации пользовательских интерфейсов разработаны специальные программно-архитектурные образцы:
  - ◊ Модель–представление–контроллер;
  - ◊ Представление–абстракция–управление;
  - ◊ «Seeheim»;
  - ◊ «Arch/Slinky».

Схема тактик реализации практичности периода прогона представлена на рис. 5.13.

5.8. Взаимосвязь тактик и архитектурных образцов

Мы представили вашему вниманию ряд тактик, при помощи которых архитекторы реализуют те или иные атрибуты качества. Как правило, для проведения в жизнь отдельной тактики или нескольких тактик архитекторы подбирают подходящие образцы. Каждый образец, хотим мы того или нет, реализует сразу несколько

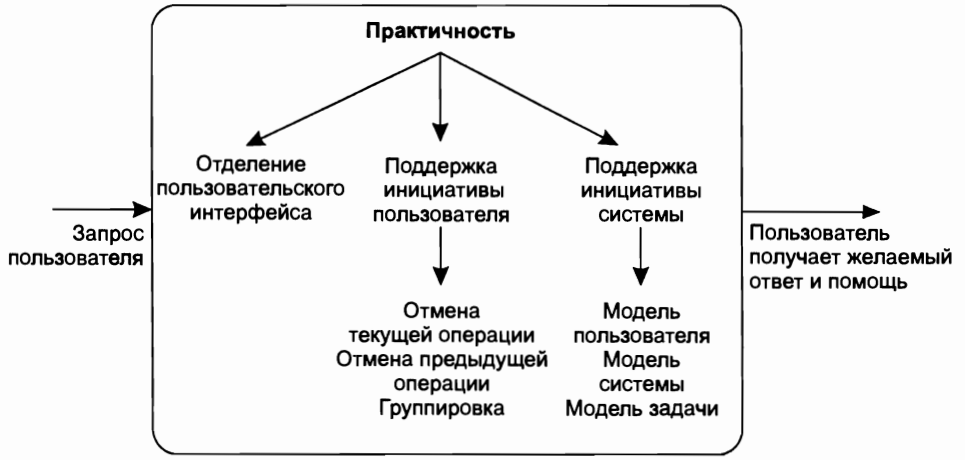


Рис. 5.13. Схема тактик практичности периода прогона

тактик. Не желая оставить это утверждение без должной аргументации, мы рассмотрим образец «активный объект» (Active Object), описание которого содержится в работе [Schmidt 00]:

> Расцепляя исполнение метода и его вызов, образец проектирования «активный объект» усиливает параллелизм и упрощает синхронизированный доступ к объектам, находящимся в собственных потоках управления.

У этого образца шесть элементов: *агент* — интерфейс, через который клиенты вызывают публично доступные методы активного объекта; *запрос метода*, определяющий интерфейс для исполнения методов активного объекта; *список активизации*, содержащий буфер ожидающих запросов методов; *планировщик*, принимающий решение о том, какие запросы методов следует исполнить далее; *слуга*, который определяет поведение и состояние, моделируемые в виде активного объекта; и *будущее*, посредством которого клиент получает результат вызова метода.

Задача этого образца состоит в усилении параллелизма; параллелизм, как известно, относится к производительности. Таким образом, можно утверждать, что он реализует тактику производительности «введение параллелизма». Но — обратите внимание, какие еще тактики принимают в нем участие.

- *Информационная закрытость (модифицируемость)*. Каждый элемент выбирает для себя обязанности и скрывает их выполнение за интерфейсом.
- *Посредник (модифицируемость)*. В роли посредника выступает агент, буферизующий изменения вызова метода.
- *Время связывания (модифицируемость)*. Образец активного объекта предполагает, что запросы на объект поступают к нему в период прогона. При этом точное время связывания клиента с агентом не установлено.
- *Политика планирования (производительность)*. Планировщик реализует политику планирования.

Итак, любой образец реализует сразу несколько тактик, которые зачастую даже относятся к разным атрибутам качества. Кроме того, при реализации этого образца

также принимаются решения о реализации тех или иных тактик. К примеру, реализация может предусматривать ведение журнала запросов к активному объекту в расчете на возможное восстановление, ведение контрольного журнала или обеспечивать контролепригодность.

Для проведения анализа архитектор должен иметь представление обо всех встроенных в реализацию тактиках, а на этапе проектирования от него требуется принятие решений о том, какие тактики смогут наилучшим образом выполнять поставленные перед системой задачи.

5.9. Архитектурные образцы и стили

Архитектурный образцы (стили) в программном обеспечении — это аналоги архитектурных стилей, применяемых в строительстве зданий (например, известны готический стиль, стиль эпохи Возрождения, античные стили). Любой архитектурный образец состоит из нескольких ключевых характеристик и правил, которые в сочетании обеспечивают архитектурную целостность. Архитектурный образец определяется:

- набором типов элементов (например, репозитария данных или компонента, вычисляющего математическую функцию);
- топологической схемой элементов, обозначающей взаимосвязь между ними;
- набором семантических ограничений (например, фильтры в составе стиля «каналы и фильтры» — это преобразователи данных; они постепенно преобразуют входящие потоки в выходящие, но при этом не располагают контролем над восходящими и нисходящими потоками элементов);
- набором механизмов взаимодействия (например, вызовов подпрограмм, событий-подписчиков, досок объявлений), регламентирующих координацию элементов в рамках установленной топологии.

Совсем недавно Мэри Шоу (Mary Shaw) и Дэвид Гарлан (David Garlan) попытались классифицировать набор архитектурных образцов, которые тогда назывались архитектурными стилями, или идиомами. Усилиями сообщества программных инженеров на основе результатов их исследований были сформулированы архитектурные образцы — по аналогии с образцами проектирования и образцами кодирования.

В своей работе [Shaw 96] Мэри Шоу выдвинула гипотезу о существовании высокоуровневых абстракций сложных систем, которые на тот момент не изучались и не классифицировались, — впрочем, аналогичная ситуация в тот момент складывалась во многих других инженерных дисциплинах.

Образцы регулярно проявляются в решениях систем, но иногда их трудно обнаружить, поскольку, в разных дисциплинах одни и те же архитектурные образцы называются по-разному. Ситуация требовала систематизации повторяющихся архитектурных образцов, формулирования их свойств и преимуществ. Один из вариантов такой классификации представлен на рис. 5.14.

Образцы на этой иллюстрации подразделяются на родственные группы в рамках иерархии наследования. К примеру, событийная система изображается как

вторичный стиль независимых элементов. У самих событийных систем два вторичных образца: неявный и явный вызов.

Какая связь между архитектурными образцами и тактиками? Как мы уже говорили, любую тактику следует рассматривать как фундаментальный «строительный блок» проектного решения, из которого, в свою очередь, выводятся архитектурные образцы и стратегии.

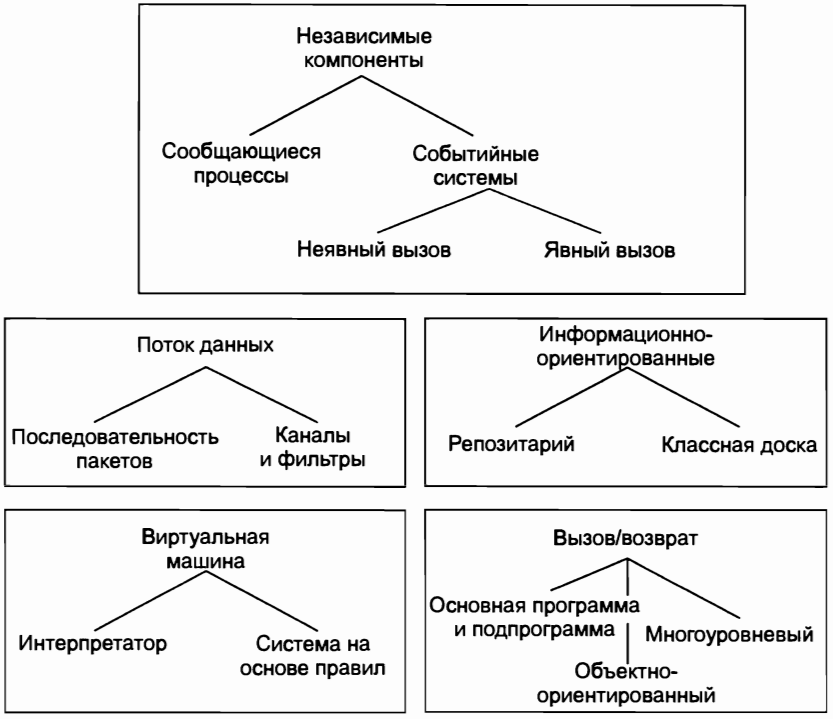


Рис. 5.14. Сокращенная классификация архитектурных образцов, между которыми установлены отношения «является»

5.10. Заключение

Мы рассмотрели способы выполнения архитектором требований по атрибутам качества. Эти требования, в свою очередь, позволяют системе выполнять поставленные перед ней коммерческие задачи. Объектом нашего внимания в настоящей главе стали тактики, которые, наравне с архитектурными образцами и стратегиями, позволяют архитектору вырабатывать проектное решение.

Мы представили ряд распространенных тактик реализации шести проанализированных в главе 4 атрибутов качества: готовности, модифицируемости, производительности, безопасности, контролепригодности и практичности. Все рассмотренные тактики открыты и пользуются широкой известностью.

Как мы уже говорили, выбрав тактики, архитектор лишь приступает к выполнению основной задачи, которая заключается в соотнесении тактик и образцов.

Не существует проектного решения, в котором была бы задействована всего одна тактика. Архитектор, таким образом, должен иметь представление о том, какие атрибуты качества реализуют различные тактики и к каким побочным эффектам они приводят, а также осознавать риски, сопряженные с отказом от других возможных альтернатив.

5.11. Дополнительная литература

Подробный анализ проблем безопасности содержится в работе [Ramachandran 02]; сведения о связи между практичностью и программно-архитектурными образцами есть в издании [Bass 01a], а о методиках обеспечения готовности в распределенных системах — в исследовании [Jalote 94]. [McGregor 01] — добротный источник информации о контролепригодности.

Двухтомный справочник по архитектурным образцам — [Buschmann 96] и [Schmidt 00] — содержит данные по образцам MVC и PAC (том 1) и программной архитектуре, ориентированной на образцы (том 2).

Дискуссии по поводу симплекс-архитектуры обеспечения готовности ведутся по адресу http://www.sei.cmu.edu/simplex/.

В работе [Bachmann 02] применение тактик рассматривается как основа для анализа модифицируемости и производительности; [Chretienne 95] содержит данные о концепциях теории планирования; в работе [Briand 99] предлагается обзор метрик сцепления.

Существует документация по различным образцам: «Модель–представление–контроллер» [Gamma 95], «Представление–абстракция–управление» [Buschmann 96], «Seeheim» [Plaff 85] и «Arch/Slinky» [UIMS 92].

5.12. Дискуссионные вопросы

1. Возьмем популярный веб-сайт — например, Amazon или eBay. Какие тактики имеет смысл рассмотреть при выборе архитектурных образцов и стратегий, реализующих требования к производительности, которые вы должны были перечислить в ответе на вопрос 3 в главе 4?
2. Какие компромиссные решения относительно других атрибутов качества (безопасности, готовности и модифицируемости), вероятнее всего, придется принять, если обратиться к тактикам, выбранным при ответе на предыдущий вопрос?
3. Внимание, которое уделяется практичности в ходе проектирования архитектур, не всегда достаточно — таким образом, реализовать цели, связанные с практичностью, становится значительно сложнее, поскольку до них дело доходит в последний момент. Возьмите любую систему, об архитектуре которой вы имеете представление, и постарайтесь вспомнить, какие тактики реализации практичности в ней задействованы.

Глава 6
Управление воздушным движением. Пример разработки, ориентированной на высокую готовность

> В течение целых десяти лет Федеральное авиационное агентство пыталось заменить стремительно старевшую систему управления воздушным движением и все это время сталкивалось с проблемой сложности выполнения поставленной задачи. Новый проект под названием «Комплексная система автоматизации» (AAS) полностью отвечает всем требованиям, которые предъявляются к вычислительным системам в 1990-е годы. Программа, в которой больше миллиона строк, распределяется между многими сотнями компьютеров и встраивается в новое, модернизированное оборудование, которое, в свою очередь, должно круглосуточно реагировать на непредсказуемые, поступающие в реальном времени события. Минимальный сбой в такой ситуации потенциально угрожает общественной безопасности.
>
> *В. Вейт Гиббс [Gibbs 94]*

К программным приложениям управления воздушным движением (Air Traffic Control, ATC) предъявляются невероятно высокие требования. Они должны работать в жестких условиях реального времени (hard real time) — иначе говоря, безусловно соответствовать временным ограничениям; выполнять особые требования к безопасности (safety critical) — из-за неправильной работы системы могут погибнуть люди; кроме того, они относятся к категории сверхраспределенных (highly distributed) — для ведения самолетов по авиалиниям требуется

сотрудничество десятков диспетчеров. Коммерческие, частные и военные воздушные суда пользуются воздушным пространством Соединенных Штатов интенсивнее, чем во всех остальных странах, — соответственно, на эту сферу обращено серьезное общественное внимание. Но безопасностью проблемы не исчерпываются — правительство тратит на построение и сопровождение хорошо защищенных, надежных систем управления воздушным движением огромные деньги. Стоимость разработки системы АТС исчисляется миллиардами и даже десятками миллиардов долларов.

В настоящей главе мы рассмотрим конкретный пример одного из элементов спроектированной недавно системы управления воздушным движением нового поколения, разработанной в США. Мы увидим, как за счет архитектуры — в частности, набора тщательно отобранных проекций (см. главу 2) и тактик (см. главу 5) — ее разработчикам удалось добиться выполнения серьезных разнородных требований. За недостатком финансирования систему так и не ввели в действие, однако ее реализация явственно продемонстрировала соответствие всем поставленным качественным задачам.

Управлением воздушным движением в Соединенных Штатах занимается Федеральное авиационное агентство (Federal Aviation Administration, FAA) — правительственное учреждение, отвечающее за общую безопасность полетов. Именно оно и выступило заказчиком описанной ниже системы. Безопасность прохождения самолета по авиалиниям и через наземные средства обслуживания обеспечивается во взаимодействии с различными элементами системы АТС. Координация передвижения самолета по территории аэропорта осуществляется *службой наземного движения* (ground control). Со специальных вышек координируется его перемещение в *узловом диспетчерском районе* (terminal control area) — цилиндрическом сегменте воздушного пространства с центром над аэропортом. Обязанности по обеспечению безопасности полетов в воздушном пространстве страны поделены между 22 *районными центрами* (en route centers).

Рассмотрим перелет из г. Ки-Уэст (Флорида) в аэропорт Далласа (Вашингтон, округ Колумбия). От стоянки до конца взлетно-посадочной полосы передвижение самолета координируется службой наземного движения Ки-Уэста. Во время взлета и набора высоты контроль переходит к диспетчерской вышке. С момента выхода самолета из узлового диспетчерского района Ки-Уэста он находится в зоне ответственности районного центра управления полетами Майами (он, помимо прочего, обязан контролировать полеты над Ки-Уэстом). Впоследствии управление полетом передается районным центрам Джексонвиля, Атланты и т. д. вплоть до того момента, как самолет входит в воздушное пространство, подконтрольное районному центру Вашингтона. Тот через какое-то время передает управление диспетчерской вышке аэропорта Далласа, которая берет на себя обязанности по координации захода рейса на посадку и приземления. Покинув взлетно-посадочную полосу, экипаж связывается со службой наземного движения, которая доводит самолет до стоянки. Эта схема работы систем управления воздушного движения в США сильно упрощена, однако для анализа нашего конкретного примера такого уровня детализации вполне достаточно. На рис. 6.1 приводится иллюстрация процесса передачи управления воздушным движением, а на рис. 6.2 — карта 22 американских районных центров управления полетами.

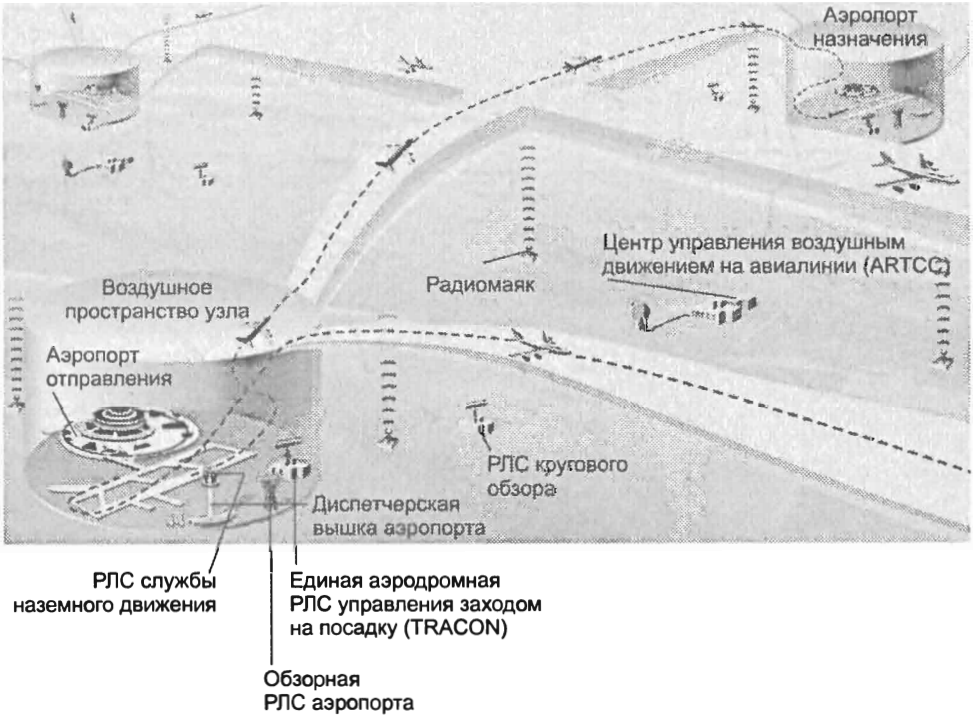


Рис. 6.1. Координация перелета из точки А в точку В средствами системы управления воздушным движением США (иллюстрация публикуется с разрешения Яна Уорпола/Scientific American, 1994)

Система, обзор которой мы собираемся представить вашему вниманию, называется Основной системой контроля секторов (Initial Sector Suite System, ISSS). Изначально она предназначалась для замены устаревшего оборудования и программного обеспечения 22 районных центров управления полетами в США. Это лишь один из элементов крупномасштабного правительственного проекта, в результате поэтапной реализации которого аналогичные системы нового поколения предполагалось установить на диспетчерских вышках, в службах наземного движения и трансокеанских центрах управления полетами.

То обстоятельство, что ISSS задумывалась лишь как один из элементов набора близкородственных систем, оказало на ее архитектуру глубочайшее влияние. В частности, везде, где это возможно, разработчики должны были внедрять решения и элементы, предусматривающие возможность повторного применения в последующих системах. Как-никак, у систем в районных центрах управления полетами, на диспетчерских вышках и в службах наземного движения много общего: наличие интерфейсов для обмена данными с радиосистемами, базами планирования полетов и друг с другом, необходимость в обработке показаний РЛС, повышенные требования к надежности и производительности и т. д. Таким образом, проектные решения, реализованные в ISSS, в значительной степени обусловливались требованиями ко всем обновлявшимся системам. Весь набор модернизированных систем предполагалось назвать комплексной системой автоматизации (Advanced Automation System, AAS).

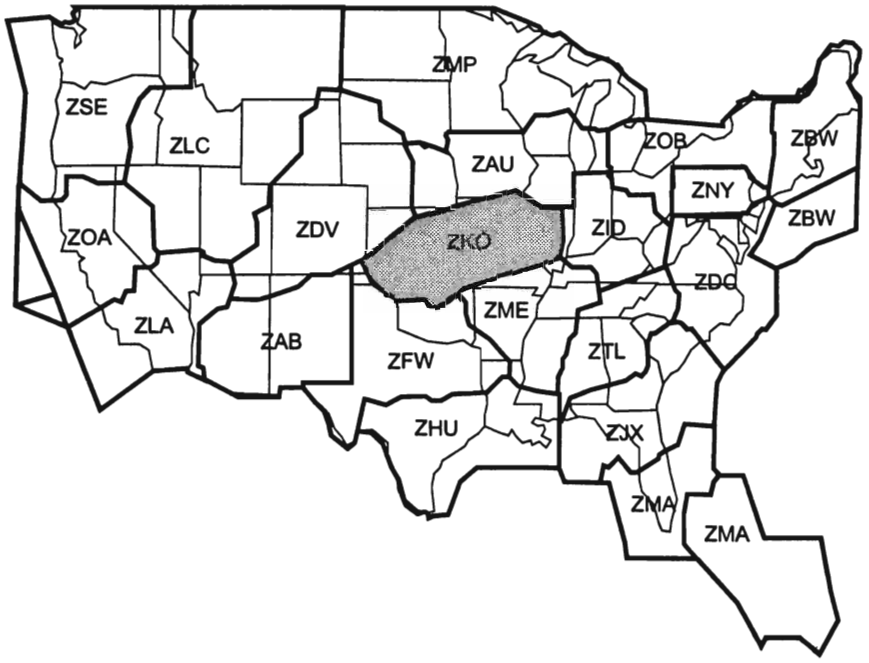


Рис. 6.2. Распределение воздушного пространства США между районными центрами управления полетами

В конце концов от программы AAS отказались в пользу менее амбициозного и дорогостоящего, зато постепенного плана модернизации. Тем не менее конкретный пример системы ISSS до сих пор сохраняет свою ценность — дело в том, что, когда пришло известие об отмене проекта, проект и львиная доля кода уже были подготовлены. Более того, независимая группа аудиторов, обследовавшая архитектуру системы (равно как и большинство других ее аспектов), пришла к выводу о ее соответствии сформулированным требованиям. Наконец, система, впоследствии размещенная вместо ISSS, заимствовала у архитектуры последней множество элементов. Все эти обстоятельства позволяют говорить об архитектуре ISSS как о примере решения в высшей степени трудной задачи.

6.1. Связь с архитектурно-экономическим циклом

На рис. 6.3 приводится схема архитектурно-экономического цикла (Architecture Business Cycle, ABC) применительно к системе управления воздушным движением. Конечными пользователями системы должны были стать специалисты по управлению полетами; заказчиком выступило Федеральное авиационное агентство США; на роль компании-разработчика выбрали крупную корпорацию, имевшую обширный опыт создания важных, преимущественно программных, систем

по заказу правительства США. Среди факторов формирования технической базы следует назвать требование о применении для реализации крупных правительственных программных систем языка Ada, а также появление распределенной обработки данных — стандартной методики конструирования систем и обеспечения отказоустойчивости.

Рис. 6.3. Архитектурно-экономической цикл применительно к системе управления воздушным движением

6.2. Требования и атрибуты качества

С учетом того, что управление воздушным движением представляет собой открытую сферу, в которой переплетаются правительственные, коммерческие и общественные интересы, а в случае возникновения неисправностей существует риск гибели людей, два основных атрибута качества можно сформулировать следующим образом:

1. Сверхвысокая готовность — другими словами, нарушение работоспособности системы с превышением заданного интервала времени абсолютно недопустимо. Количественные требования к готовности системы ISSS выражались показателем 0,99999, из которого следовало, что период ее неготовности необходимо было ограничить пятью минутами в год. (Правда, если система оказывалась способна восстановиться после отказа и возобновить работу в пределах 10 секунд, этот отказ при подсчете периода неготовности не учитывался.)

2. Высокая производительность. Системе предстояло обрабатывать данные о почти 2440 рейсах, не «теряя» из виду ни один из них. Сети должны были выдерживать серьезные нагрузки, а программное обеспечение — проводить вычисления быстро и предсказуемо.

Следует перечислить и некоторые другие требования, которые, не будучи столь значимыми в контексте обеспечения безопасности самолетов и пассажиров, все же в значительной степени обусловили характер и принципы построения архитектуры.

- Открытость. Система должна была предусматривать возможность ввода в свой состав коммерческих программных компонентов — в частности, функций управления воздушным движением и основных вычислительных служб наподобие пакетов графического отображения.
- Способность к выделению подмножеств системы — на случай, если проект стоимостью во многие миллиарды долларов падет жертвой сокращения бюджета (а значит, и функциональности), как, в конечном итоге, и случилось.
- Возможность дополнения функциональности, модернизации аппаратного и программного обеспечения (в частности, установки новых процессоров, устройств ввода-вывода и драйверов, новых версий компилятора языка Ada).
- Способность к взаимодействию и сопряжению с неопределенным набором внешних систем — как аппаратных, так и программных, почтенного возраста и до сих пор не реализованных.

Наконец, отличительной чертой этой системы является наличие огромного количества заинтересованных лиц — в частности, диспетчеров, выступавших в роли конечных пользователей. В этом, на первый взгляд, нет ничего удивительного, но диспетчеры могли отказаться от пользования непонравившейся системой, даже если бы она отвечала всем эксплуатационным требованиям. Из-за этого весьма существенного фактора процессы выявления требований и проектирования системы сильно растянулись.

Термином *блок управления секторами* (sector suite) обозначается комплект контроллеров (каждый из которых находится на консоли — см. рис. 6.4), предназначенных для управления воздушным движением в отдельном секторе воздушного пространства района. Итак, представленную выше упрощенную схему управления воздушным движением можно дополнить тем, что обязанности по координации передвижения воздушных судов передаются не только от района к району, но и от сектора к сектору. Принципы выделения секторов определяются индивидуально в каждом районе. В частности, основным фактором может быть стремление равномерно распределить нагрузку между диспетчерами района; если это так, то чем интенсивнее воздушное движение в том или ином секторе, тем он меньше.

Проект системы ISSS предусматривает гибкость в отношении количества диспетчерских пунктов в каждом секторе; допустимые значения находятся в диапазоне от одного до четырех, причем во время работы системы они могут корректироваться командным способом. В любом секторе должно быть не менее двух

диспетчеров. Один из них — диспетчер радиолокационного контроля — следит за обзорными показаниями РЛС, ведет переговоры с бортами и обеспечивает безопасное эшелонирование. На его плечи, таким образом, ложится тактическое управление ситуацией в секторе. Второй диспетчер ответствен за данные — он получает информацию (в частности, планы полетов) обо всех бортах, находящихся в секторе или приближающихся к нему. Диспетчер данных оповещает диспетчера РЛС о намерениях борта, а тот, в свою очередь, старается максимально безопасно и эффективно провести борт через воздушное пространство сектора.

ISSS — крупная система. Некоторое представление о ее масштабах, возможно, смогут дать следующие цифры.

- В каждом центре управления полетами ISSS способна обеспечивать работу до 210 консолей. В каждой консоли установлен процессор класса «рабочая станция» — IBM RS/6000.
- Согласно требованиям к ISSS, каждый центр должен одновременно управлять движением 400–2440 бортов.
- В каждом аппаратном блоке воздушным движением может быть установлено от 16 до 40 РЛС.
- В каждом районном центре может быть от 60 до 90 пунктов управления (и в каждом из них по одной или по несколько консолей).
- Реализация ISSS состоит примерно из одного миллиона строк кода на языке Ada.

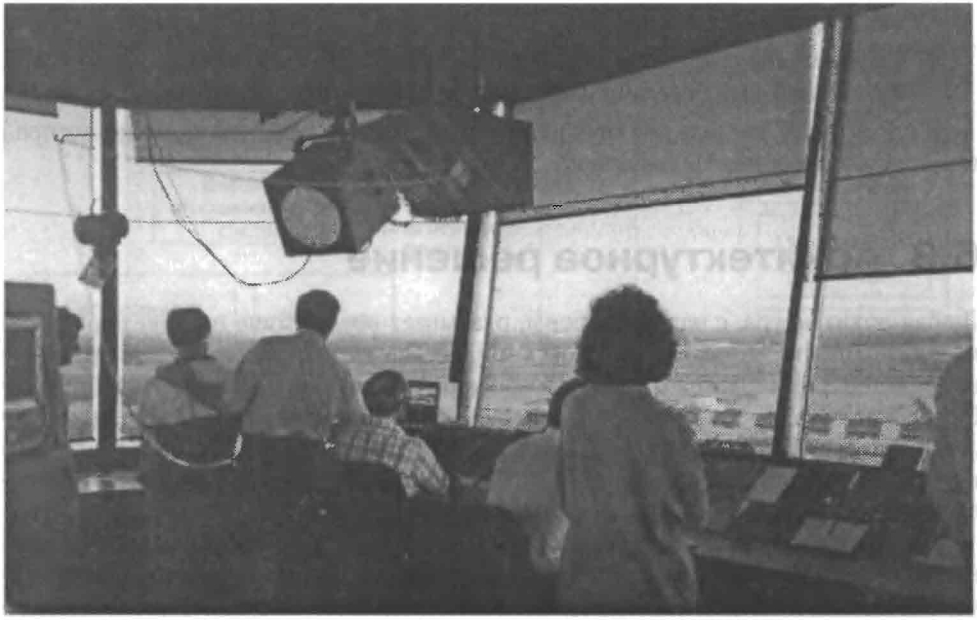


Рис. 6.4. Диспетчеры, работающие с блоком управления сектором (приводится с разрешения Технического центра Уильяма Дж. Хьюза; Федеральное авиационное агентство разрешает свободное распространение этого снимка)

Итак, перед системой ISSS были поставлены следующие основные задачи:

- Получение отчетов о радиолокационных целях, которые хранятся в существующей системе управления полетами — в так называемом хост-компьютере (Host Computer System).
- Преобразование радиолокационных отчетов в форму, пригодную для отображения на экране, и их пересылка на все консоли. Каждая консоль выбирает из них только те, которые ей требуются; при этом любая консоль может отобразить любую область.
- Обработка предупреждений о конфликтах (потенциально ведущих к столкновению воздушных судов) и всех прочих данных, передаваемых с хост-компьютера.
- Предоставление хост-компьютеру интерфейса для подачи входных данных и получения планов полетов.
- Предоставление подробных данных текущего контроля и управляющей информации (в частности, относящейся к сетевому управлению), при помощи которых администраторы узлов проводят реконфигурацию систем в реальном времени.
- Обеспечение возможности записи и последующего считывания.
- Размещение на консолях элементов графического пользовательского интерфейса — в частности, оконная организация экранного пространства. Кроме того, необходимы дополнительные меры предосторожности — например, если окна не будут прозрачными, диспетчеры могут упустить из виду критические данные.
- Обеспечение средств резервирования на случай сбоев на хост-компьютере, в основной сети передачи данных и в основных радиолокационных датчиках.

В следующем разделе мы проанализируем архитектуру, при помощи которой эти требования удалось выполнить.

6.3. Архитектурное решение

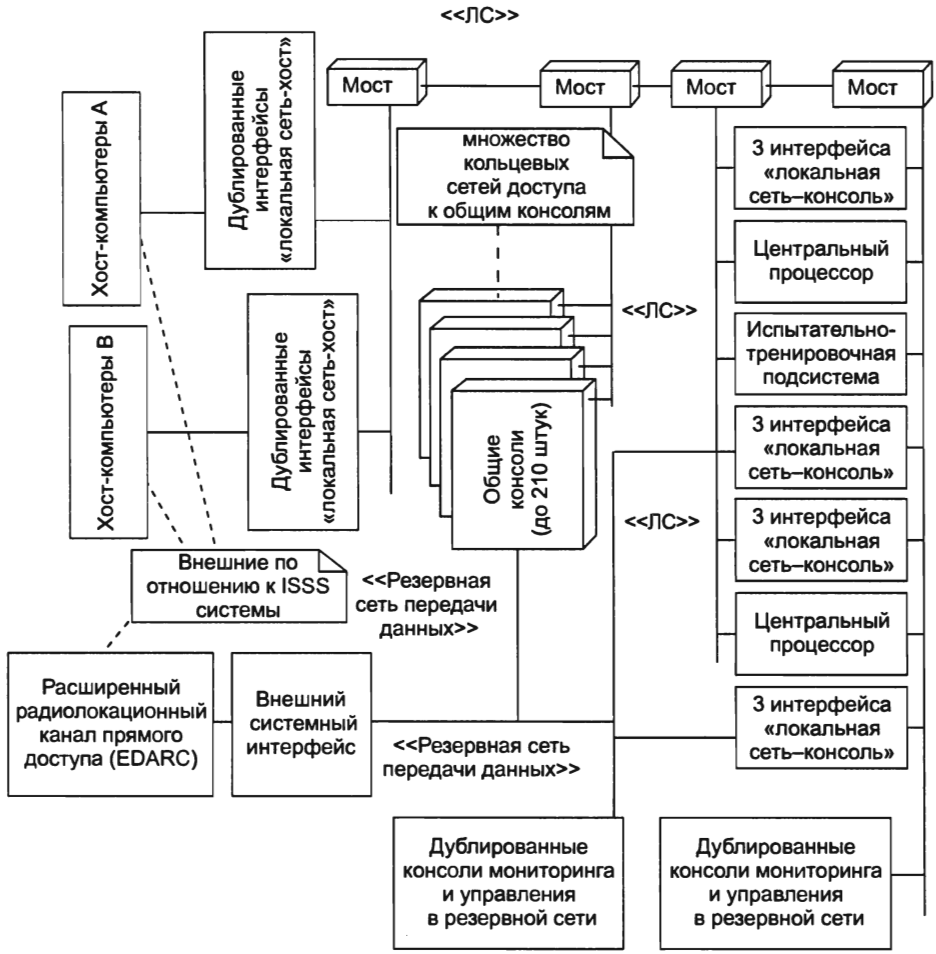
Любая архитектура, с одной стороны, оказывает воздействие на поведение, производительность, отказоустойчивость и удобство сопровождения, но с другой — ее характер формируется с учетом требований, выставляемых ко всем этим атрибутам качества. В случае с ISSS основным фактором влияния на архитектуру стало необычно высокое требование к готовности системы — не более пяти минут простоя в год! Именно оно в большей, чем многие другие, степени мотивировало принятие архитектурных решений.

Обзор архитектуры ISSS мы начнем с описания физической среды размещения программного продукта. Затем мы перечислим ряд программно-архитектурных представлений (как в главе 2) и обозначим тактики их реализации (как в главе 5). Параллельно мы рассмотрим еще одно представление, которое раньше не рассматривали — отказоустойчивость. Разобравшись с анализом отношений между представлениями, мы в довершение всего приведем детализированную версию

тактики «общие абстрактные службы», позволяющую реализовать модифицируемость и расширяемость, — называется она «кодовые шаблоны».

Физическое представление системы ISSS

ISSS — это распределенная система, многочисленные элементы которой соединяются посредством локальных сетей. Физическое представление системы ISSS изображено на рис. 6.5. Вспомогательные системы, равно как и их интерфейсы для взаимодействия с оборудованием ISSS, на иллюстрации не отражены. Нет на ней и указаний на структуру программного обеспечения. Ниже перечислены основные элементы физического представления и их назначение.



Составлено на языке UML

Рис. 6.5. Физическое представление системы ISSS

- Ядром районной системы автоматизации является хост-компьютер. В каждом таком центре установлено по два хост-компьютера — один из них основной, а другой должен быть постоянно готов к решению тех или иных задач вместо основного. Хост обрабатывает обзорные данные и сведения, относящиеся к планам полетов. Обзорные данные выводятся на районные дисплейные консоли, с которыми работают диспетчеры. При необходимости сведения о полетах распечатываются на специальных ленточных принтерах. Отдельные элементы этих данных выводятся посредством тегов данных, ассоциированных с обзорными данными РЛС.
- Общие консоли выступают в роли рабочих станций диспетчеров управления воздушным движением. На их дисплеях выводятся данные о местоположении воздушных судов и соответствующие теги данных в горизонтальной проекции (индикатор РЛС), данные о планах полетов в форме электронных полетных лент[1] и т. д. Они позволяют диспетчерам редактировать информацию о полетах, контролировать выводимые на дисплеи данные и формат их отображения. В каждом блоке управления, обслуживающем команду диспетчеров сектора управления воздушным пространством, находится от 1 до 4 консолей.
- Общие консоли соединяются с хост-компьютерами по локальной сети передачи данных (Local Communications Network, LCN) — основной сети ISSS. Взаимодействие хоста с сетью осуществляется через дублированные интерфейсные блоки (LIU-H) — отказоустойчивую резервированную пару.
- Локальная сеть передачи данных состоит из четырех параллельных кольцевых сетей с маркерным доступом, которые, во-первых, резервируют друг друга, а во-вторых, обеспечивают выравнивание общей нагрузки. В одной из сетей проводится широковещательная передача обзорных данных всем процессорам. Один процессор применяется для двухточечной передачи данных между парами процессоров; другой выделяет канал для передачи дисплейных данных от общих консолей записывающим блокам в расчете на последующее поуровневое считывание; третий процессор выступает в роли резерва. Соединение между сетями с маркерным доступом и магистральной сетью поддерживается при помощи мостов. Они же обеспечивают переход с поврежденной кольцевой сети на резервную и маршрутизацию по альтернативным путям.
- Расширенный радиолокационный канал прямого доступа (enhanced direct access radar channel, EDARC) предусматривает резервный вывод на районные дисплейные консоли информации о местоположении бортов и блочных данных о полетах. Он вводится в действие лишь в тех случаях, когда дисплейные данные перестают поступать с хоста, и представляет «сырые», необработанные радиолокационные данные и интерфейсы для процессора ESI (external system interface, внешний системный интерфейс).

[1] Полетная лента — это распечатываемая системой полоса бумаги с данными плана полета находящегося в текущем секторе или приближающегося к нему борта. До появления системы ISSS надписи на эти ленты наносились от руки, карандашом. В ISSS реализована возможность управления содержанием лент с экрана.

- В качестве резервной сети передачи данных (backup communications network, BCN) используется сеть Ethernet со стеком протоколов TCP/IP. Она выполняет все системные функции, за исключением тех, которые берет на себя интерфейс EDARC, и заменяет основную сеть передачи данных на время устранения ее отказов.
- У обеих сетей — основной (LCN) и резервной (BCN) — есть собственные консоли мониторинга и управления. С их помощью специалисты по сопровождению системы наблюдают за ее состоянием и регулируют ее операции. Это обычные консоли со специальным программным обеспечением функций M&C, а также высокоуровневыми (глобальными) функциями контроля готовности.
- Испытательно-тренировочная подсистема предназначена для тестирования нового аппаратного оборудования и программного обеспечения, а также для обучения пользователей; все эти действия проводятся параллельно с работой основной системы управления воздушным движением.
- Центральные процессоры относятся к классу универсальных ЭВМ. В одной из первых версий ISSS на них возлагались функции записи и считывания данных.

Каждая общая консоль подключена одновременно к основной и резервной сетям. Поскольку в одном центре управления воздушным движением может быть установлено до 210 общих консолей, для их подключения используется сразу несколько кольцевых сетей доступа. Так выглядит физическое представление системы ISSS — иначе говоря, аппаратура, на которой установлено ее программное обеспечение.

Представление декомпозиции модулей

Модульные элементы системного программного обеспечения ISSS называются элементами конфигурации компьютерных программ (Computer Software Configuration Items, CSCIs) — они утверждены в правительственном стандарте разработки программного обеспечения, на применении которого настоял заказчик. В основном элементы CSCI регламентируют распределение обязанностей между группами разработчиков, занимающихся их проектированием, конструированием и тестированием. Как правило, для каждого такого элемента составляется четкое логическое обоснование группировки содержащихся в нем мелких программных элементов (пакетов, процессов и т. д.).

В системе ISSS пять элементов CSCI.

1. Элемент управления выводом (Display Management), ответственный за производство и сопровождение дисплеев на общих консолях.
2. Общие системные службы (Common System Services), ответственные за обеспечение утилит, которые используются в управлении воздушным движением, — как вы помните, в планы разработчика входило построение в рамках программы AAS ряда других систем.

3. Модуль «Запись, анализ и считывание» (Recording, Analysis and Playback), ответственный за запись сеансов управления воздушным движением для последующего анализа.
4. Модификация национальной воздушно-космической системы (National Airspace System Modification), определяющая внесение изменений в программное обеспечение хост-компьютера (в силу того, что этот элемент не вписывается в предметную область настоящей главы, мы не будем его рассматривать).
5. Операционная система IBM AIX — базовое операционное окружение системного программного обеспечения.

Перечисленные элементы CSCI являются поставляемыми модулями документации и программ; на их основе планировались контрольные точки; на каждый из них возлагалась ответственность за логически связанный сегмент функциональности системы ISSS.

Представление декомпозиции модуля отражает ряд тактик реализации модифицируемости (см. главу 5). Основной тактикой распределения строго очерченных, непересекающихся обязанностей между элементами CSCI архитекторы выбрали «семантическую связность». Модуль «Общие системные службы» отражает тактику «общие абстрактные службы». CSCI «Запись, анализ и считывание» отражает тактику реализации контролепригодности «запись/считывание». Доступ к ресурсам всех элементов CSCI предоставляется через тщательно спроектированные программные интерфейсы, что соответствует тактикам «прогнозирование ожидаемых изменений», «обобщение модуля» и «поддержание стабильности интерфейсов».

Представление процессов

Параллелизм в ISSS основывается на элементах типа «приложение» (application). Приложение в данном случае примерно соответствует процессу (в том смысле, который вкладывал в это понятие Дийкстра, рассуждая о кооперации последовательных процессов) и лежит в центре методики, которую разработчики ISSS адаптировали для обеспечения отказоустойчивости. Каждое приложение реализуется в виде «основного» модуля языка Ada (процесса, планируемого операционной системой) и составляет один из элементов CSCI (что помогает выявить соответствие между представлениями декомпозиции модулей и процессов). Взаимодействие приложений проходит в форме передачи сообщений, которые в рамках представления «компонент и соединитель» выступают в роли соединителей.

В конструкции системы ISSS предусматривается возможность многопроцессорной обработки. Логическая комбинация процессоров называется *группой процессоров* (processor group — об этом мы говорили применительно к физическому представлению), а предназначена она для размещения ряда отдельных копий одного или нескольких приложений. На этом принципе основывается отказоустойчивость, а следовательно, и готовность. Одна исполняемая копия является первичной, а все остальные — вторичными; отсюда названия отдельных копий приложения: *основное адресное пространство* (primary address space, PAS) и *за-*

пасное адресное пространство (standby address space, SAS). Совокупность основного адресного пространства и сопровождающих его запасных адресных пространств называется *операционным блоком* (operational unit). Каждый отдельно взятый операционный блок полностью размещается в процессорах одной группы — напомним, что на одну группу может приходиться от одного до четырех процессоров. Те элементы системы ISSS, которые конструируются без расчета на отказоустойчивость (другими словами, не предполагают сосуществования основной и запасной версий), запускаются независимо на разных процессорах. Называемые *функциональными группами* (functional groups), они при необходимости могут быть размещены на любом процессоре; при этом каждая копия представляет собой отдельный экземпляр программы с собственным состоянием.

Итак, в роли приложения может выступать как операционный блок, так и функциональная группа. Различие между ними состоит в том, сколько вторичных копий обеспечивают резервирование функциональности приложения, — они (или она) полностью повторяют состояние и данные первичной копии и при необходимости подменяют ее. Конструкция операционных блоков отличается отказоустойчивостью; для функциональных групп это не характерно. Реализация приложения в виде операционных блоков проводится только при наличии соответствующих требований по готовности; во всех остальных случаях приложение реализуется в виде функциональной группы.

Приложения взаимодействуют по модели «клиент–сервер». Клиент транзакции отсылает серверу *сообщение-запрос на обслуживание* (service request message), на которое тот отвечает подтверждением. (Как и во всех прочих реализациях клиент-серверной схемы, любая сторона — в данном случае приложение — в одной транзакции может выступать в роли клиента, а в другой — в роли сервера.) Основное адресное пространство (PAS) внутри операционного блока отправляет всем своим резервным пространствам (SAS) уведомления о смене состояния; резервные пространства, готовые в случае отказа основного пространства или соответствующего процессора заменить его, постоянно пребывают в состоянии ожидания тайм-аутов и других стимулов к действию. Рисунок 6.6 иллюстрирует координацию основного и резервных адресных пространств приложения при резервировании и передаче результатов их взаимодействия группам процессоров.

При получении функциональной группой сообщения она отвечает на него и соответствующим образом обновляет свое состояние. Как правило, основное адресное пространство операционного блока принимает сообщения и отвечает на них, представляя весь операционный блок. После этого оно обновляет собственное состояние и состояние резервных адресных пространств — для этой цели тем отправляются дополнительные сообщения.

Переключение в случае отказа основного адресного пространства происходит следующим образом:

1. Одно из резервных адресных пространств становится основным.
2. Новое основное адресное пространство восстанавливает взаимодействие с клиентами своего операционного блока (в каждом таком блоке существует фиксированный список клиентов), отправляя им специальное сообщение. В нем оно сообщает об отказе операционного блока и спрашивает, кто

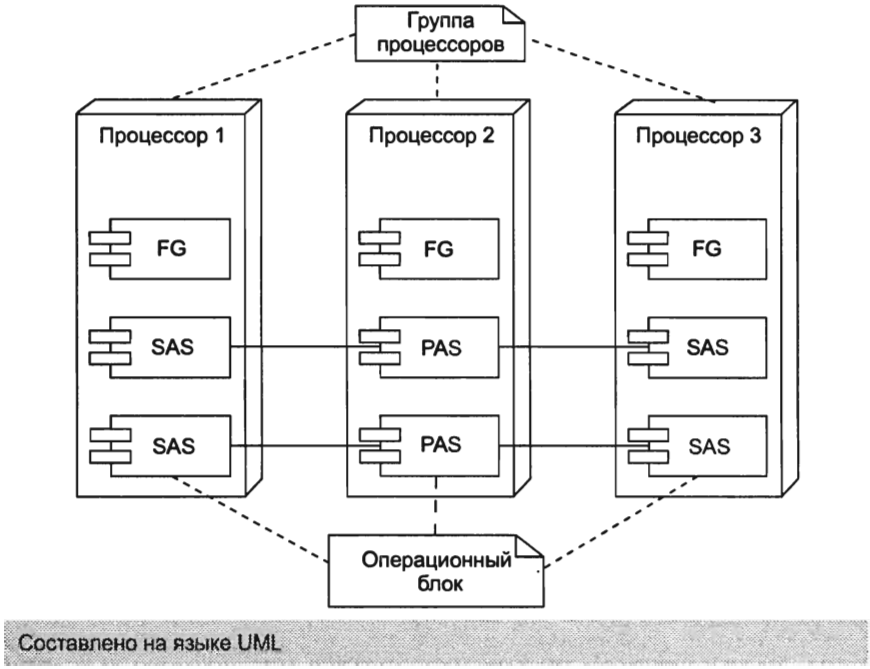


Рис. 6.6. Функциональные группы (FG), операционные блоки, группы процессоров и основное (PAS)/резервные (SAS) адресные пространства

и какую информацию ожидает от него получить. Все запросы, получаемые в ответ на это сообщение, обслуживаются.

3. Для замены предыдущего основного адресного пространства запускается новое резервное адресное пространство.
4. Новое резервное адресное пространство извещает новое основное пространство о своем существовании и с этого момента начинает получать от последнего сообщения с обновлениями.

Если отказ поражает резервное адресное пространство, для его замены на некотором другом процессоре запускается новое резервное пространство. Установив взаимодействие с основным пространством, оно начинает принимать данные о состоянии.

Введение нового операционного блока осуществляется следующим образом:

- Определяются и локализуются необходимые входные данные.
- Определяются операционные блоки, рассчитывающие на получение выходных данных от нового операционного блока.
- Образцы передачи данных нового операционного блока встраиваются в общесистемный ациклический граф, причем его ациклический характер во избежание взаимоблокировок должен сохраняться неизменным.
- Сообщения конструируются в расчете на достижение заданных потоков данных.

- Выявляются внутренние данные состояния для расстановки контрольных точек и участия в сообщениях обновления, которые основные адресные пространства высылают резервным.
- Данные состояния разбиваются на сообщения, соответствующие формату применяемой сети.
- Определяются необходимые типы сообщений.
- На случай отказов составляется план переключения, который в целях обеспечения полного состояния регулярно обновляется.
- Гарантируется непротиворечивость данных в случае переключения.
- Необходимо сделать так, чтобы отдельные этапы обработки проходили быстрее, чем продолжается один «такт» системы.
- Планируются протоколы совместного с другими операционными блоками использования данных и протоколы блокирования данных.

Выполнение этой последовательности действий под силу только группам, состоящим из опытных разработчиков. Тактика, которую мы рассмотрим в следующем разделе — «кодовые шаблоны», — призвана сделать этот процесс в большей степени повторяемым и снизить вероятность ошибок.

Представление процессов отражает сразу несколько тактик реализации готовности — в частности, «повторную синхронизацию состояния», «затенение», «активное резервирование» и «снятие с эксплуатации».

Клиент-серверное представление

Поскольку взаимодействие между приложениями в представлении процессов проходит по модели «клиент–сервер», имеет смысл отдельно рассмотреть соответствующее представление. Следует, впрочем, иметь в виду, что описываемое в нем поведение в значительной степени повторяет содержание показанного ранее представления процессов. Клиент-серверное представление системы показано на рис. 6.7.

Конструкция клиентов и серверов предполагает наличие в них совместимых (в отличие от специализированных) интерфейсов. Этому содействует практика применения простых протоколов передачи сообщений. Результат соответствует положениям ряда тактик реализации модифицируемости: «поддержания стабильности интерфейсов», «замены компонентов» и «применения предписанных протоколов».

Кодовое представление

Несмотря на то что в главе 2 мы не упоминали о кодовом представлении, в архитектурах крупных систем оно встречается довольно часто. Его задача — показать соответствие функциональности и блоков кода.

В состав любой (основной) программы в системе ISSS, написанной на языке Ada, может входить один или несколько исходных файлов; как правило, в ней

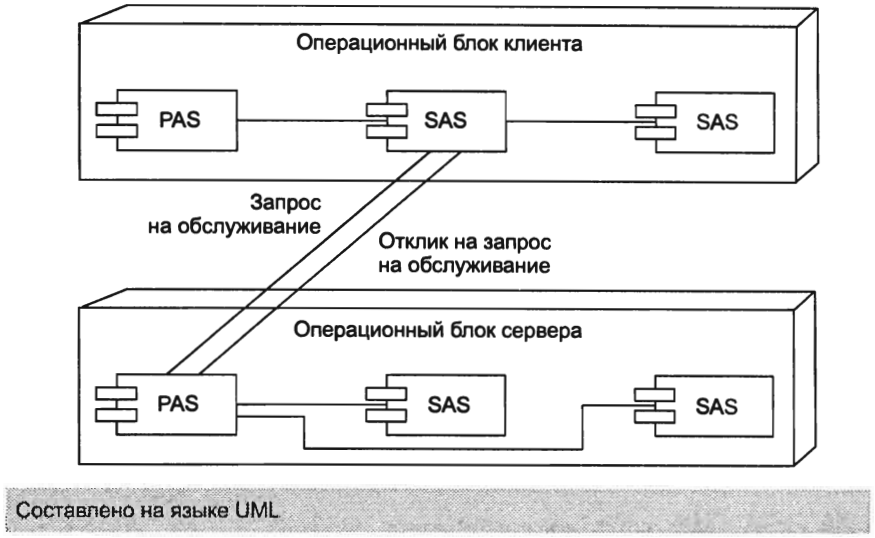


Рис. 6.7. Приложения в роли клиентов и серверов

содержится ряд *подпрограмм* (subprograms), некоторые из которых объединяются в раздельно компилируемые *пакеты* (packages). В ISSS есть несколько таких программ, причем часть из них функционирует согласно клиент-серверной модели.

Программа на языке Ada может состоять из одной или нескольких *задач* (tasks) — конструкций языка, предусматривающих возможность параллельного исполнения. В рамках кодового представления они отражают процессы, описанные в представлении процессов. Поскольку управление задачами Ada осуществляется системой поддержки исполнения этого языка, система ISSS отображает их на процессы UNIX (AIX) — таким образом, все потоки управления (целые программы или задачи в составе программы на языке Ada) представляют собой параллельно исполняемые, независимые процессы AIX.

Приложения (операционные блоки или функциональные группы) распадаются на пакеты Ada. Некоторые из этих пакетов не содержат ничего, кроме определений типов, иные предполагают возможность многократного использования в разных приложениях. *Пакетированием* (packaging) называется один из процессов проектирования, реализующий абстрагирование и сокрытие информации; ответственность за его проведение возлагается на главного проектировщика операционного блока.

Многоуровневое представление

Прикладные программы управления воздушным движением в системе процессоров ISSS исполняются в среде AIX — одной из коммерческих разновидностей операционной системы UNIX. Впрочем, для обеспечения функционирования отказоустойчивой распределенной системы, каковой является ISSS, набора служб UNIX недостаточно. Отсюда потребность в введении дополнительного обслужи-

вающего программного обеспечения. Иерархия уровней программной среды стандартного процессора ISSS представлена на рис. 6.8.

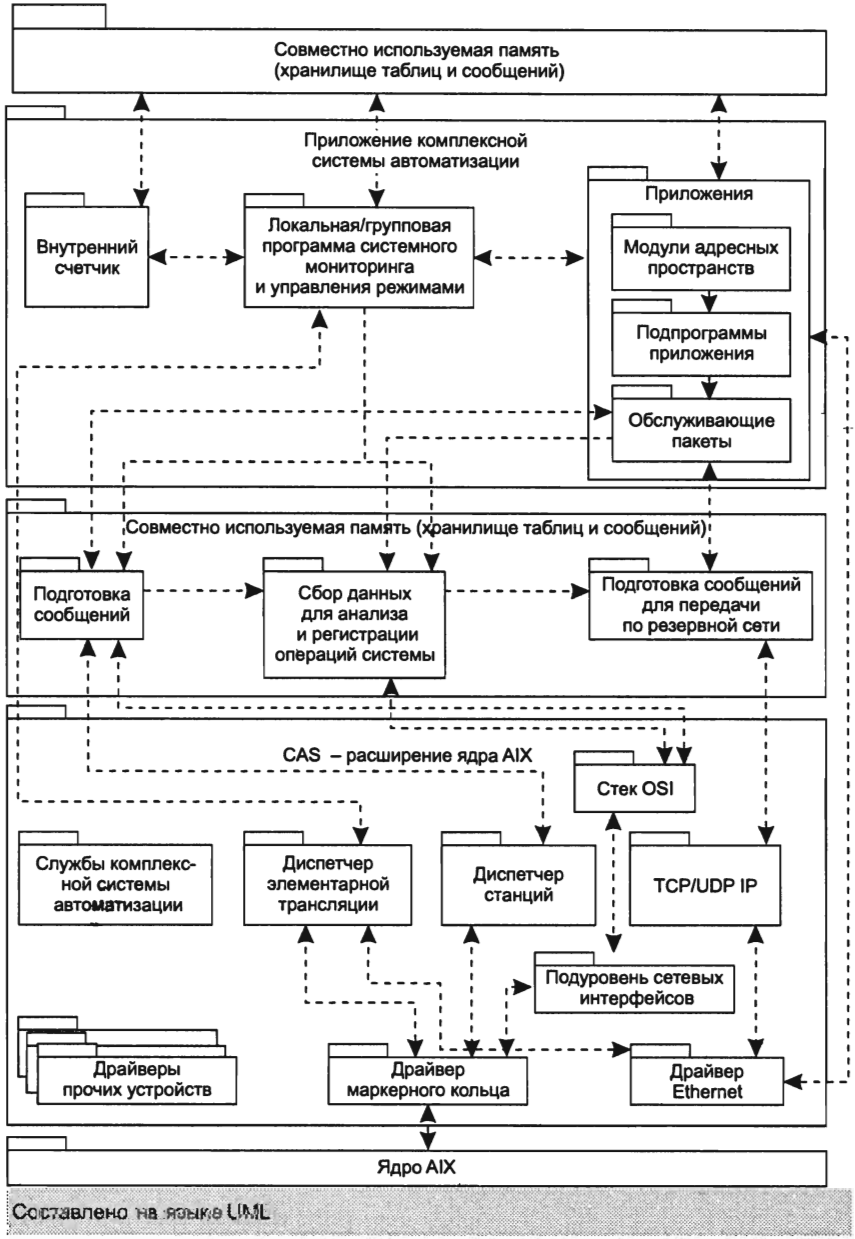


Рис. 6.8. Уровни программной архитектуры системы ISSS. Ассоциации на схеме демонстрируют потоки данных и/или управления — таким образом, получается накладка многоуровневого представления на представление «компонент и соединитель»

Строго говоря, на рис. 6.8 изображена *накладка* (overlay), совмещающая многоуровневое представление и представление «компонент и соединитель», — дело в том, что на этой схеме показаны характерные для периода прогона связи между подмодулями на разных уровнях. Правда, связи между службами комплексной системы автоматизации (AAS) и драйверами прочих устройств, с одной стороны, и остальными подмодулями с другой, здесь не показаны — они настолько многочисленны, что могли бы занять все пространство диаграммы. Большинство элементов многоуровневой системы могут обращаться к ней совершенно свободно. Фактически установленные связи мы перечислим в пояснительном тексте.

Два нижних ряда элементов, расположенные непосредственно над ядром AIX, заняты расширениями AIX, исполняемыми внутри адресного пространства этого ядра. Согласно требованиям к производительности и совместимости с операционной системой AIX, эти расширения в основном выполнены в виде небольших программ на языке C. Поскольку исполняются эти программы в адресном пространстве ядра, их неисправности потенциально способны нанести ущерб системе AIX; таким образом, это должны быть относительно небольшие и надежные программы, отражающие тактику ограничения внешних воздействий (см. главу 5). Эта тактика, изначально предназначенная для реализации безопасности (конкретнее, для предотвращения атак типа «отказ от обслуживания), в ISSS используется для повышения готовности. К счастью, некоторые тактики с одинаковым успехом реализуют сразу несколько атрибутов качества.

Основную роль во взаимодействии между модулями диспетчера локальной готовности в рамках блока управления секторами исполняет диспетчер элементарной трансляции (atomic broadcast manager, ABM) — он обеспечивает готовность функций этого блока. Диспетчер станций, с одной стороны, представляет в основной сети передачи данных службы дейтаграмм, а с другой — выступает в роли локального представителя управляющих служб этой сети. Аналогичные функции для двухточечной передачи сообщений предусматриваются на подуровне сетевых интерфейсов, который пользуется сетевой информацией совместно с диспетчером станций.

На двух следующих уровнях представлены расширения операционной системы; поскольку они исполняются вне адресного пространства ядра AIX, их неисправности не способны напрямую повредить AIX. Как правило, эти программы составляются на языке Ada.

Программа подготовки сообщений (Prepare Messages) оперирует сообщениями, которые передаются по основной локальной сети прикладным программам. Аналогичные действия в отношении сообщений, передаваемых по резервной локальной сети, выполняются программой подготовки сообщений резервной сети. В числе прочего, эти программы призваны устанавливать, какая из многочисленных резервных копий прикладной программы в рамках блока управления сектором является первичной и, следовательно, должна принимать сообщения. Необходимую для этого управляющую информацию поставляет диспетчер локальной готовности.

На верхнем уровне находятся приложения. Диспетчер локальной готовности и программа внутренней временной синхронизации представляют собой системные службы прикладного уровня. Диспетчер локальной готовности осуществля-

ет управление запуском, завершением и готовностью прикладных программ. Путем взаимодействия со всеми адресными пространствами на своем процессоре он контролирует их работу и проверяет состояние. Кроме того, он обменивается информацией с диспетчерами локальной готовности на других процессорах в рамках данного блока управления сектором и, таким образом, координирует готовность его функций — в частности, функции переключения с первичной на резервную копию прикладной программы. Наконец, диспетчер локальной готовности взаимодействует с глобальным приложением контроля готовности, находящимся на консоли мониторинга и управления, — в ходе этого он составляет для последнего отчеты о состоянии и принимает от него команды управления. Программа внутренней временной синхронизации синхронизирует такт процессора с тактами других процессоров системы ISSS, обеспечивая, таким образом, нормальную работу функций контроля готовности. (См. представление отказоустойчивости на рис. 6.9.)

Новое представление: отказоустойчивость

Как мы уже говорили, перечень представлений, приведенный в главе 2, не является исчерпывающим. Перечня, учитывающего все представления программной архитектуры для всех возможных систем или для отдельно взятой системы, не может быть в принципе. Радостно, что в последнее время все больше внимания уделяется вопросам влияния программной архитектуры на реализацию атрибутов качества; соответственно, признается важность ясного формулирования ожидаемых от архитектуры атрибутов качества. Пытаясь достичь этой цели, архитекторы имеют обыкновение составлять такие представления, которые демонстрируют реализацию архитектурой конкретного атрибута качества, — например, представление безопасности. Что касается атрибутов качества периода прогона, то здесь представления составляются в категориях «компонент и соединитель» и демонстрируют взаимодействие между элементами в этот период. Для атрибутов качества, не относящихся к периоду прогона, представления составляются в категориях модулей и показывают проектные характеристики блоков реализации, направленные на реализацию того или иного атрибута качества — например, модифицируемости.

Из-за повышенных требований к готовности системы ISSS отказоустойчивость стала неотъемлемой характеристикой ее проектного решения. Во-первых, возможность холодного перезапуска системы в случае отказа следовало полностью исключить. Лучшим выходом из ситуации признали немедленное (по меньшей мере, быстрое) переключение на резервный компонент. По мере проектирования этот принцип вырисовывался со все большей четкостью, и в конечном итоге появилась новая архитектурная структура — иерархия отказоустойчивости (рис. 6.9). Она описывает механизм обнаружения и устранения неисправностей, а также восстановления системы. В то время как схема «основное/резервные адресные пространства» отвечает за перехват ошибок и восстановление в рамках отдельного приложения, иерархия отказоустойчивости обеспечивает обнаружение и восстановление после ошибок, явившихся результатом взаимодействия между приложениями.

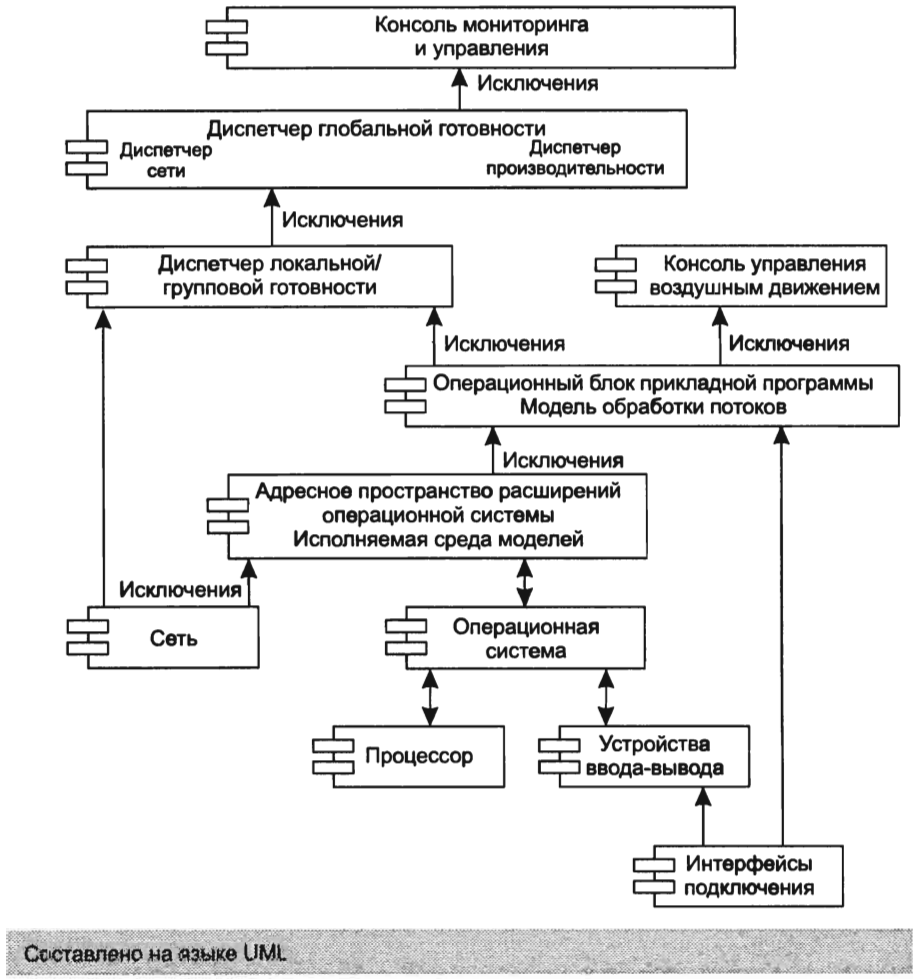


Рис. 6.9. Представление «компонент и соединитель», выражающее отказоустойчивость системы ISSS

Иерархия отказоустойчивости в системе ISSS предусматривает несколько уровней обнаружения неисправностей и восстановления. Каждый из них в асинхронном режиме выполняет следующие действия:

- проводит обнаружение ошибок в себе, в одноранговых элементах и на нижележащих уровнях;
- обрабатывает исключения, порождаемые на нижележащих уровнях;
- проводит диагностические мероприятия, восстановление, составляет отчетность и порождает исключения.

Каждый вышележащий уровень обеспечивает более высокую степень готовности системы, чем нижележащие. Иерархия отказоустойчивости состоит из следующих уровней:

- физический уровень (сеть, процессор и устройства ввода-вывода);

- уровень операционной системы;
- уровень исполняемой среды;
- прикладной уровень;
- уровень локальной готовности;
- уровень групповой готовности;
- уровень глобальной готовности;
- уровень системного мониторинга и управления.

Мероприятия, связанные с обнаружением и устранением неисправностей, проводятся на всех уровнях иерархии. Обнаружению неисправностей служат встроенные тесты, тайм-ауты событий, тесты сетевых схем, протоколы группового членства и, как последнее средство, — мероприятия, проводимые специалистами в ответ на аварийные сигналы и показания индикаторов.

Обнаружение ошибок также проводится на всех уровнях иерархии — вручную или автоматически. Диспетчеры локальной, групповой и глобальной доступности задействуют табличные методы восстановления. В основном адресном пространстве предусмотрено четыре типа восстановления после отказов. Решение о применении того или иного типа восстановления принимается диспетчером локальной готовности согласно таблицам решений в зависимости от текущего рабочего состояния:

- в случае переключения резервное адресное пространство почти без промедления принимает на себя обязанности бывшего основного адресного пространства;
- при теплом перезапуске задействуются контрольные точки (они сохраняются в энергонезависимой памяти);
- при холодном перезапуске теряется история состояний и применяются данные по умолчанию;
- для перехода на новую (или старую) логику или адаптационные данные проводится переключение (cutover).

За резервирование отвечают сетевое оборудование (основная и резервная сети передачи данных, а также все мосты), аппаратные процессоры (в группу процессоров входят от двух до четырех процессоров; резервная запись) и программное обеспечение (в операционный блок входит несколько адресных пространств).

В дополнение к тактикам реализации готовности, которые мы описали в представлении процессов, обнаружению сбоев служат ping/echo-пакеты и heartbeat-запросы; для оповещения элементов, ответственных за исправление ошибок, служат *исключения* (exceptions), а для проведения восстановления применяются *резервы* (spares).

Взаимоотношения представлений

То и дело встречаются ситуации, когда элементы одного представления «приглашаются» в другие представления. Представления сами по себе открывают путь к анализу структуры системы, однако для того чтобы разобраться в ней еще лучше,

во многих случаях требуется провести анализ отношений между представлениями и, в частности, отображений одного представления на другое. Такой подход позволяет рассмотреть архитектуру более целостно.

Единицами представления декомпозиции модулей в системе ISSS являются элементы конфигурации компьютерных программ (computer software configuration items, CSCIs). Они состоят из приложений, которые одновременно являются элементами представления процессов и клиент-серверного представления. Реализуются приложения в виде программ и пакетов на языке Ada, участвующих в кодовом представлении; в свою очередь, эти приложения и пакеты отображаются на потоки — единицы представления параллелизма (о нем мы не говорили). Многоуровневое представление описывает распределение функциональности между модулями в рамках представления декомпозиции и демонстрирует те элементы, к которым эти модули могут обращаться. Наконец, специализированное представление, направленное на реализацию конкретного атрибута качества периода прогона — представление отказоустойчивости, — предполагает участие элементов представления процессов, модульного и многоуровневого представлений.

В главе 9, посвященной документированию программных архитектур, мы, помимо прочего, покажем, где в комплекте документации следует фиксировать отношения между представлениями. В случае с системой ISSS отображение необходимо задокументировать в виде таблиц с перечислением элементов из разных представлений и указанием их взаимоотношений.

Адаптационные данные

В ISSS широко используется тактика реализации модифицируемости под названием «конфигурационные файлы», которая, согласно терминологии этой системы, имеет и другое название — адаптационные данные. Конкретно-узловые адаптационные данные присутствуют в 22 районных центрах управления полетами, в которых планировалось разместить систему ISSS. Так называемые предопределенные адаптационные данные приспосабливают программные средства к изменениям, которые происходят в период разработки и размещения, но при этом не отражают различия между узлами. Адаптационные данные — это очень удобное и важное средство модифицирования системы согласно специфическим требованиям узлов, предпочтениям пользователей и центров, изменениям конфигурации и требований, а также многим другим аспектам программных средств, которые допускают модификации с течением времени и в зависимости от узлов размещения. Проектное решение программного продукта допускает считывание рабочих параметров и поведенческих спецификаций с входных данных; таким образом, в отношении набора линий поведения, которые возможно представить в этих данных, программный продукт совершенно универсален (и отражает тактику «обобщение модуля»). К примеру, для того чтобы реализовать новые требования, согласно которым данные, ранее отображавшиеся в одном окне, следует разделить на два отдельных окна (во многих системах сделать это не так уж просто), достаточно внести изменения в адаптационные данные и отредактировать несколько строк кода.

К сожалению, сопровождение механизма адаптационных данных сопряжено с некоторыми трудностями. К примеру, с операционной точки зрения ввести в систему несколько новых команд или командный синтаксис довольно просто, но, с другой стороны, такой уровень гибкости достигается за счет создания нового, весьма сложного интерпретирующего языка. Кроме того, между различными элементами адаптационных данных устанавливаются отношения, повышенная сложность которых способна оказывать негативное воздействие на правильность; в то же время автоматических или хотя бы полуавтоматических механизмов защиты от противоречивости такого рода просто не существует. Наконец, адаптационные данные существенно увеличивают пространство состояний, в рамках которого требуется корректное функционирование системного программного обеспечения, что, в свою очередь, осложняет процесс системного тестирования.

Уточнение тактики «общие абстрактные службы»: кодовые шаблоны для приложений

Как вы помните, согласно схеме «первичное/вторичные адресные пространства», отказоустойчивость достигается за счет резервирования — отдельные копии программного продукта хранятся в разных процессорах. Во время исполнения первичная копия регулярно отсылает всем вторичным копиям информацию о своем состоянии — делается это для того, чтобы при необходимости они могли взять на себя функции первичной копии. План реализации этих копий основывается на точных копиях одного и того же *исходного кода* (source code). Несмотря на то что первичная и вторичные копии никогда не выполняют одни и те же задачи в одно и то же время (первичная копия работает над исполнением своих обязанностей и отправляет вторичным копиям пакеты с обновлениями состояния; те, помимо того что получают эти пакеты, ожидают ситуаций, в которых им придется взять на себя обязанности основной копии), обе программы происходят от идентичных копий одного исходного кода. В расчете на это для каждого из приложений разработаны стандартные кодовые шаблоны; один из них приводится в листинге 6.1.

Соответствующая структура представляет собой непрерывный цикл, обслуживающий входящие события. Если результатом поступления данного события должно стать выполнение приложением нормального (не связанного с обеспечением отказоустойчивости) действия, приложение выполняет это действие, а затем отправляет своим вторичным копиям информацию для обновления. Большинство приложений обрабатывают от 50 до 100 нормальных событий. Среди других возможных событий — передача (пересылка и прием) пакетов с информацией об обновлении состояния и данных. Наконец, еще один набор событий включает уведомления о принятии данным элементом обязанностей первичного адресного пространства и поступающие от клиентов запросы на обслуживание, не завершенное предыдущим (поврежденным в данный момент) первичным адресным пространством.

У любого такого шаблона есть архитектурные аспекты. Он упрощает введение в систему новых приложений, причем с минимальным влиянием на работу существующих механизмов обеспечения отказоустойчивости. От кодировщиков

и специалистов по сопровождению не требуется конкретных знаний о механизмах обработки сообщений, они не обязаны обеспечивать отказоустойчивость приложений — эти задачи должны решаться на более высоком (архитектурном) уровне проектирования.

Кодовые шаблоны — это не что иное, как уточненная версия тактики «общие абстрактные службы»; в шаблоне конкретизируются общие элементы каждого приложения. Эта тактика связана с рядом других тактик реализации модифицируемости. Относительно изменяемых элементов она отражает тактику «прогнозирование ожидаемых изменений», а процессам придает «семантическую связность» — дело в том, что все они при абстрактном рассмотрении выполняют одни и те же задачи. Шаблон позволяет программистам сконцентрироваться на деталях приложения, в результате чего реализуется тактика «обобщение модуля». Благодаря введению в шаблон интерфейсов и протоколов поддерживается стабильность интерфейсов и обеспечивается «применение предписанных протоколов».

Обзор методик и тактик, позволивших программной архитектуре системы ISSS реализовать задачи по качеству, приводится в табл. 6.1.

Листинг 6.1. Структурный кодовый шаблон для отказоустойчивых приложений ISSS

```
terminate:= false
инициализировать приложение/протокол взаимодействия приложений
запрос текущего состояния (запрос образа)
Loop
   Get_event
   Case Event_Type is

   -- «нормальные» (не связанные с обеспечением отказоустойчивости) запросы
   -- на выполнение действий;
   -- их поступление возможно только в том случае, если данный элемент
   -- в настоящее время исполняет роль основного адресного пространства

   when X=>Process X
      отправка пакетов обновления состояния другим адресным пространствам
   when Y=>Process Y
      отправка пакетов обновления состояния другим адресным пространствам
   ...
   when Terminate_Directive =>очистка ресурсов; terminate := true

   when State_Data_Update =>применяется к данным о состоянии
   -- возможно только в том случае, если данный элемент является вторичным
   -- адресным пространством, принимающим от первичного пространства,
   -- выполнившего «нормальное действие», пакет с обновлениями

   -- отправка, прием данных о состоянии
   when Image_Request =>отправка новому адресному пространству данных
                        о текущем состоянии
   when State_Data_Image =>инициализация данных о состоянии

   when Switch_Directive =>оповещение обслуживающих пакетов об изменении
                           ранга

   -- такие запросы поступают после перехода (переключения) вторичного
   -- адресного пространства в роль основного;
   -- они сообщают об обслуживании, запрошенном у старого (поврежденного)
```

```
   -- основного пространства, перешедшего в обязанности нового (текущего)
   -- основного пространства. Символами A,B и др. обозначаются имена
   -- клиентов.
   when Recon_from_A=>восстанавливает взаимодействие с A
   when Recon_from_B=>восстанавливает взаимодействие с B
   ...
   when others=>log error
   end case
exit when terminate
end loop
```

Таблица 6.1. Методы реализации задач по качеству для системы управления воздушным движением

Задача	Метод реализации	Тактика(и)
Высокая готовность	Аппаратное резервирование (процессоров и сети); программное резервирование (многоуровневая схема обнаружения неисправностей и восстановления)	Повторная синхронизация состояния; затенение; активное резервирование; снятие с эксплуатации; ограничение внешних воздействий; отправка ping/echo-сообщений; отправка heartbeat-запросов; исключения; резерв
Высокая производительность	Распределенная многопроцессорная схема; анализ возможности планирования предварительной обработки; сетевое моделирование	Введение параллелизма
Открытость	Оборачивание и многоуровневая организация интерфейсов	Общие абстрактные службы; поддержание стабильности интерфейсов
Модифицируемость	Табличные адаптационные данные; продуманное распределение обязанностей между модулями; жесткое разграничение интерфейсов	Общие абстрактные службы; семантическая связность; поддержание стабильности интерфейсов; прогнозирование ожидаемых изменений; обобщение модулей; замена компонентов; применение предписанных протоколов; конфигурационные файлы
Способность к выделению подмножеств	Тщательное разделение задач	Общие абстрактные службы
Способность к взаимодействию	Разделение функциональности между клиентом и сервером, обмен данными на основе сообщений	Применение предписанных протоколов; поддержание стабильности интерфейсов

6.4. Заключение

Подобно всем остальным конкретным примерам в настоящей книге, в системе ISSS архитектурные решения предстают в роли решающих факторов реализации требований к приложению. Основные методики, применявшиеся для достижения этой цели, приводятся в табл. 6.1. Вследствие длительного (как предполагалось) срока эксплуатации, высокой стоимости разработки, крупномасштабности, важности исполняемых функций и существенной обозримости в систему ISSS постоянно вносились какие-то изменения — и это не говоря о серьезнейших

рабочих требованиях. Человеко-машинные интерфейсы, новое аппаратное оборудование и коммерческие компоненты, обновления операционной системы и сети, увеличение вычислительных мощностей — все это было неизбежно. Благодаря введению разного рода механизмов реализации отказоустойчивости (и кодовых шаблонов), в частности, аппаратного и программного резервирования, а также многоуровневого обнаружения неисправностей и распределенных многопроцессорных вычислений с передачей сообщений от клиента к серверу архитектурное решение оказалось вполне адекватным беспрецедентно высоким рабочим требованиям.

Осталось лишь сказать несколько слов о тщательной проверке архитектуры ISSS, проведенной группой специалистов в связи с тем, что правительство США приняло решение отказаться от этой системы в пользу менее сложного и дорогостоящего решения. Они проверяли способность архитектуры обеспечить затребованную производительность, готовность и модифицируемость — в последнем случае для достижения своей цели эксперты задействовали ряд сценариев изменений:

- внесение серьезных изменений в человеко-машинный интерфейс консоли мониторинга и управления;
- импортирование в систему ISSS приложений для управления воздушным движением, разработанных сторонней компанией;
- введение в систему новых представлений управления воздушным движением;
- замена процессоров RS/6000 микросхемой с аналогичными характеристиками;
- исключение из списка требований электронной полетной ленты;
- 50-процентное увеличение максимальной нагрузки системы по обработке маршрутов полета.

В каждом из перечисленных случаев эксперты установили, что проводить модификации в условиях данной программной архитектуры просто, а в некоторых случаях чуть ли не элементарно. Это можно оценивать как признание тщательности проектирования архитектуры и явного учета атрибутов качества, а также обеспечивающих их реализацию архитектурных тактик.

6.5. Дополнительная литература

Мероприятия, предпринятые Федеральным авиационным агентством США с целью модернизировать программное обеспечение управления воздушным движением, освещены в литературе достаточно широко; взять, например, исследование [Gibb 94]. Отчет о проверке системы ISSS на предмет возможности дальнейшего применения содержится в работе [Brown 95]. Удобство сопровождения в этих статьях понимается двояко: во-первых, как атрибут качества системы, а во-вторых, как способность организации-подрядчика выполнять задачи, связанные с сопровождением. Такой существенный аспект удобства сопровождения, как соот-

ветствие между потребностями системы в сопровождении и возможностями компании по проведению соответствующих действий, исследователи обычно упускают из виду.

6.6. Дискуссионные вопросы

1. Высокая готовность — это качество, которое архитектура, представленная в настоящей главе, должна была реализовать в первую очередь. Каким образом это требование должно было повлиять на другие атрибуты качества — например, на производительность? Как изменится архитектура, если это требование снять?
2. Сколько архитектурных образцов вы смогли обнаружить в архитектуре системы ISSS?
3. Согласно требованиям, перечисленным в разделе 6.2, составьте ряд сценариев атрибутов качества (см. главу 4). В тех случаях, когда каких-то сведений недостает, пытайтесь делать логические выводы.

Глава 7
Проектирование архитектуры

(в соавторстве с Феликсом Бахманом)

> По нашим наблюдениям, практически у всех удачных объектно-ориентированных систем есть две отличительные особенности, которые, как по совпадению, в большинстве случаев не реализованы в системах неудачных: во-первых, это наличие стройной архитектурной концепции, а во-вторых, применение контролируемого, итеративного и инкрементного цикла разработки.
>
> *Грэди Буч (Grady Booch) [Stikeleather 96]*

Вплоть до настоящего момента мы говорили о тех вещах, которые должен знать любой архитектор. В частности, мы перечислили основные понятия и принципы, касающиеся по преимуществу коммерческих аспектов архитектуры (глава 1), архитектурных представлений и структур (глава 2), атрибутов качества (глава 4), а также архитектурных тактик и образцов их реализации (глава 5). В главах 3 и 6 представлены конкретные примеры, призванные укрепить полученные знания.

Теперь мы уделим внимание проектированию архитектуры как таковой, точнее — действиям, которые должны предприниматься на начальных этапах этого процесса. Таким образом, в настоящей главе мы намерены рассмотреть четыре основные темы:

- архитектура в контексте жизненного цикла;
- проектирование архитектуры;
- структура группы разработчиков и ее связь с архитектурой;
- создание макета системы.

7.1. Архитектура в контексте жизненного цикла

Любая организация, строящая процесс разработки программных продуктов на основе архитектуры, должна знать, какое место эта последняя занимает в жизненном цикле. В литературе имеют хождение несколько моделей жизненного цикла, но лишь в одной из них — изображенной на рис. 7.1 модели эволюционного жизненного цикла поставки (Evolutionary Delivery Life Cycle) — архитектура выводится на первый план. Назначение этой модели состоит в том, чтобы, во-первых, получить отзывы пользователей и заказчиков, а во-вторых, предварить конечный выпуск разработкой нескольких промежуточных. Кроме того, с каждой новой итерацией эта модель позволяет вводить в систему новые функции, а после завершения разработки соответствующего набора характеристик — поставлять ограниченные версии. (Дополнительные сведения об этой модели жизненного цикла содержатся в разделе «Дополнительная литература» далее в этой главе.)

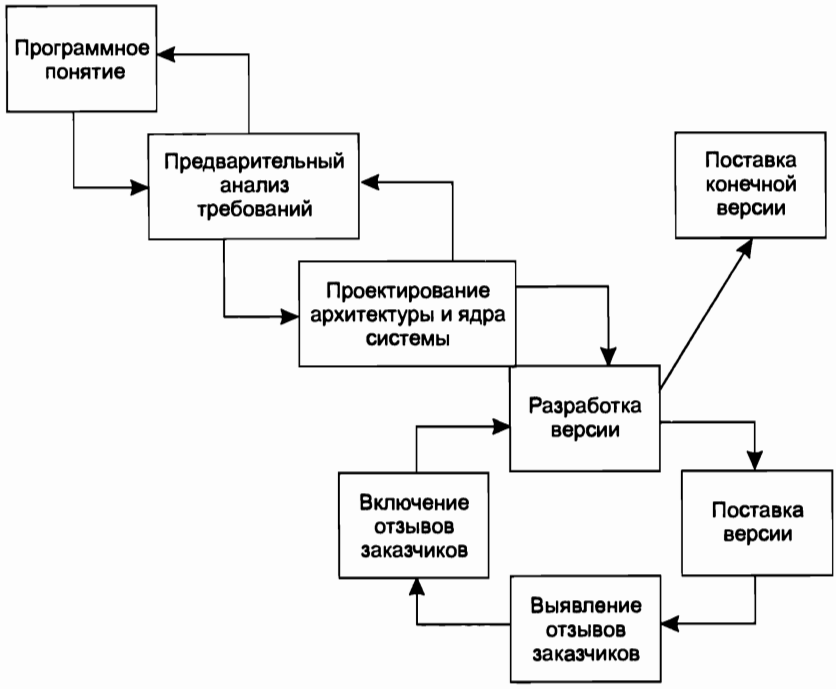


Рис. 7.1. Эволюционный жизненный цикл поставки

Когда приступать к проектированию?

Итеративный характер проектирования архитектуры, согласно вышеупомянутой модели жизненного цикла, обнаруживается через посредство предварительного

анализа требований. Действительно, как можно проектировать систему, не имея ни малейшего представления о выдвигаемых к ней требованиях? С другой стороны, для начала проектирования требуется знать не так уж много характеристик.

Любая архитектура «формируется» из некоей совокупности функциональных и коммерческих требований, а также требований по качеству. Эти требования, обнаруживаемые, в частности, по конкретным примерам, мы называем *архитектурными мотивами* (architectural drivers). Характеристики архитектуры системы A-7E, рассмотренной в главе 3, обусловливаются требованиям по модифицируемости и производительности. Архитектура системы управления воздушным движением из главы 6 находится в зависимости от требований по готовности. Что же касается программы моделирования условий полета, представленной в главе 8, то для ее архитектуры определяющими являются требования по производительности и модифицируемости, в чем вы еще сможете убедиться и т. д.

Для того чтобы установить архитектурные мотивы, проще всего разобраться с наиболее значимыми коммерческими задачами. Таковых не может быть слишком много. Представьте эти задачи в виде сценариев атрибутов качества или элементов Use Case. Затем выберите те из них, которые потенциально способны оказать на архитектуру наибольшее влияние. Именно они и являются архитектурными мотивами, и их вряд ли окажется больше десяти. Метод анализа компромиссных архитектурных решений (architecture tradeoff analysis method, ATAM), рассматриваемый в главе 11, предусматривает применение *вспомогательного дерева* (utility tree), которое помогает на основе разного рода коммерческих факторов выводить сценарии атрибутов качества.

Определившись с архитектурными мотивами, можете смело приступать к проектированию архитектуры. Процесс анализа требований, который вам предстоит провести впоследствии, во многом основывается на вопросах, сформулированных на этапе архитектурного проектирования (об этом свидетельствует одна из присутствующих на рис. 7.1 обратных стрелок).

7.2. Проектирование архитектуры

Ниже в этом разделе мы намерены рассмотреть метод проектирования архитектуры, позволяющий удовлетворить как требования к качеству, так и функциональные требования. Его мы называем атрибутным методом проектирования (Attribute-Driven Design, ADD). Исходными данными для ADD является набор сценариев атрибутов качества, а также знания об отношениях между реализацией атрибутов качества и архитектурой. Метод ADD правомерно рассматривать как расширение большинства других методов разработки — в частности, рационального унифицированного процесса (Rational Unified Process, RUP). RUP включает этапы, связанные с высокоуровневым проектированием архитектуры, за которыми следуют действия, направленные на детальное проектирование и реализацию. В результате встраивания ADD в RUP этапы высокоуровневого проектирования подвергаются некоторым изменениям, однако весь последующий процесс остается без изменений.

Атрибутный метод проектирования

ADD — это методика определения программной архитектуры, в которой процесс декомпозиции основывается на предполагаемых атрибута качества продукта. Это рекурсивный процесс декомпозиции, на каждом из этапов которого происходит отбор тактик и архитектурных образцов, удовлетворяющих тем или иным сценариям качества, а также распределение функциональности, направленное на конкретизацию типов модулей данного образца. В контексте жизненного цикла ADD располагается сразу после анализа требований, а начинается он, как мы уже говорили, в тот момент, когда об архитектурных мотивах можно говорить с достаточной степенью уверенности.

Результатом применения ADD являются первые несколько уровней представления декомпозиции модулей архитектуры, а также все другие связанные с ними представления. Не стоит, впрочем, думать, что после ADD становятся известны все детали представлений, — система описывается как набор контейнеров функциональности и существующих между ними взаимосвязей. Во всем процессе проектирования это первый случай сочленения архитектуры, результаты которого, естественно, весьма приблизительны. Тем не менее, в контексте реализации предполагаемых атрибутов качества это очень важный момент, поскольку именно здесь закладываются основы достижения функциональности. Различие между архитектурой, являющейся результатом выполнения ADD, и архитектурой, готовой к реализации, лежит в плоскости принятия подробных проектных решений. В частности, это может быть выбор между конкретными объектно-ориентированными образцами проектирования, с одной стороны, и промежуточным программным обеспечением, применение которого сопряжено с многочисленными архитектурными ограничениями, — с другой. Архитектура, спроектированная средствами ADD, предусматривает отсрочку принятия этого решения, благодаря чему достигается бóльшая гибкость.

С участием общих сценариев из главы 4, а также тактик и образцов из главы 5 можно создать ряд различных процессов проектирования. Отличаются они принятыми принципами деления проектных работ и содержанием процесса проектирования. Ниже мы приведем более подробное описание ADD — тем самым мы постараемся показать, как следует реализовывать общие сценарии и тактики, как делить проектные работы на отдельные участки и что, по нашему мнению, являет собой суть процесса проектирования.

Метод ADD мы продемонстрируем на примере архитектуры линейки продуктов для открывателя гаражной двери в рамках домашней информационной системы. Предположим, что в обязанности открывателя входит поднятие и опускание двери тремя способами: согласно положению переключателя, с пульта дистанционного управления (ПДУ), а также средствами домашней информационной системы. Будем также иметь в виду, что с помощью последней можно проводить диагностику неисправностей открывателя.

Пример входных данных

Предположим, что в качестве входных данных ADD принимает набор требований. Как и любые другие методы проектирования, ADD трактует как входные

данные функциональные требования (как правило, они выражаются в виде элементов Use Case) и ограничения. Отличительная особенность ADD кроется в трактовке требований к качеству. ADD жестко регламентирует представление требований к качеству — они могут быть выражены только в виде набора системно-ориентированных сценариев реализации качества. Рассматриваемые в главе 4 общие сценарии, во-первых, исполняют роль входных данных процесса выявления требований, а во-вторых, содержат ряд инструкций по разработке системно-ориентированных сценариев. Степень детализации последних зависит от конкретного приложения. Отдельные части полнокровных сценариев в наших примерах не учтены, поскольку они не оказывают никакого влияния на процесс проектирования.

В примере с гаражной дверью участвуют следующие сценарии атрибутов качества:

- Устройство и элементы управления открытием и закрытием двери, как уже упоминалось, обусловливаются конкретным продуктом в рамках линейки. В частности, управление может производиться из домашней информационной системы. Архитектура продукта для конкретного набора элементов управления должна напрямую выводиться из архитектуры линейки продуктов.
- Отличаются и процессоры, применяемые в разных продуктах. Архитектура продукта для каждого конкретного процессора должна напрямую выводиться из архитектуры линейки продуктов.
- Если в пространстве, занимаемой гаражной дверью, во время ее опускания обнаруживается препятствие (человек или любой другой объект), она должна остановиться (или, согласно другому варианту, полностью открыться) в течение 0,1 с.
- Открыватель гаражной двери должен предусматривать возможность диагностики и управления средствами домашней информационной системы, для чего предполагается применение ориентированного на конкретный продукт протокола диагностики. Кроме того, необходима возможность прямого производства отражающей этот протокол архитектуры.

Начало процесса ADD

Мы уже говорили об архитектурных мотивах. Метод ADD, предусматривающий предварительное выявление мотивов, может начаться лишь тогда, когда они станут известны в полном объеме. Вполне возможно, что в процессе проектирования приоритеты среды архитектурных мотивов придется реорганизовать — подобные явления вызываются, во-первых, новыми интерпретациями требований, а во-вторых, их изменением. В любом случае, процесс не начнется до тех пор, пока в отношении основных требований не появится хоть какая-то ясность.

В следующем разделе мы приступим к непосредственному рассмотрению метода ADD.

Этапы ADD

Начнем с краткого обзора этапов проектирования архитектуры методом ADD. Затем мы раскроем их содержание более подробно.

1. *Выбор модуля для декомпозиции.* Как правило, в качестве исходного модуля берется система в целом. Все необходимые входные данные (ограничения, функциональные требования и требования к качеству) для него должны быть известны.
2. *Уточнение модуля в несколько этапов*:
   a. выбор архитектурных мотивов из набора конкретных сценариев реализации качества и функциональных требований. На этом этапе определяются наиболее важные в контексте проведения декомпозиции моменты;
   b. выбор архитектурного образца, соответствующего архитектурным мотивам. Создание (или выбор) образца, тактики которого позволяют реализовать эти мотивы. Выявление дочерних модулей, необходимых для реализации этих тактик;
   c. конкретизация модулей, распределение функциональности из элементов Use Case, составление нескольких представлений;
   d. определение интерфейсов дочерних модулей. Декомпозиция имеет своим результатом новые модули, а также ограничения на типы взаимодействия между ними. Для каждого из модулей эти сведения следует зафиксировать в документации по интерфейсу;
   e. проверка и уточнение элементов Use Case в сценариях качества и наложение, исходя из них, ограничений на дочерние узлы. На этом этапе мы проверяем, все ли факторы учли, а также, в расчете на последующую декомпозицию или реализацию, подготавливаем дочерние модули.
3. *Вышеперечисленные этапы следует выполнить в отношении всех нуждающихся в дальнейшей декомпозиции модулей.*

1. Выбор модуля для декомпозиции

Система, подсистема и подмодуль — все это модули. Как правило, декомпозицию начинают с системы; она разбирается на подсистемы, а те в свою очередь — на подмодули.

В нашем примере в роли системы выступает открыватель гаражной двери. На этом уровне существует единственное ограничение — открыватель должен предусматривать возможность взаимодействия с домашней информационной системой.

2a. Выбор архитектурных мотивов

Как вы уже знаете, архитектурные мотивы представляют собой сочетание функциональных требований и требований к качеству, из которых «складывается» архитектура того или иного модуля. Мотивами считаются наиболее высокоприоритетные из всех предъявляемых к модулю требований.

В частности, архитектурными мотивами являются четыре представленных выше сценария. В системах, с оглядкой на которые мы составляли этот пример,

сценариев качества десятки. Ознакомившись с ними, мы сделали вывод о том, что магистральные требования касаются работы в реальном времени[1] и модифицируемости, необходимой при разработке линеек продуктов. Кроме того, налицо требование об оперативной диагностике. Все эти требования следует учитывать в ходе первоначальной декомпозиции системы.

Выявление архитектурных мотивов далеко не всегда проходит по нисходящей. Иногда для того, чтобы уяснить все ветвления тех или иных требований, не обойтись без детальных аналитических действий. К примеру, довольно трудно бывает определить роль производительности в контексте конкретной конфигурации системы в отсутствие ее опытной реализации. Не ознакомившись с механикой гаражной двери и скоростью предполагаемых процессоров, мы вряд ли смогли бы возвести требование по производительности в ранг архитектурного мотива.

Декомпозицию модуля мы намерены провести, исходя из архитектурных мотивов. Естественно, что к модулю предъявляются и другие требования, однако, ограничившись мотивами, мы беремся обеспечить соответствие наиболее важным из них. Рассматривать все требования как равнозначные неразумно — наименее важные из них должны удовлетворяться с учетом ограничений, присущих наиболее важным. Именно в этом принципе заключается основное различие между ADD и всеми прочими методами архитектурного проектирования.

2b. Выбор архитектурного образца

На материале главы 5 мы установили, что каждому атрибуту качества соответствует ряд очевидных тактик реализации (а также образцов, посредством которых эти тактики проводятся в жизнь). Каждая из этих тактик обеспечивает реализацию одного из нескольких атрибутов качества. Шаблоны, в которые они заключены, в свою очередь оказывают воздействие на другие атрибуты качества. Сочетание ряда тактик в архитектурном проектировании призвано сбалансировать все предполагаемые атрибуты качества. Анализ реализации требований к качеству и функциональных требований проводится на этапе уточнения.

Основная цель этапа 2b заключается в том, чтобы установить общий, состоящий из типов модулей архитектурный образец. Этот образец, включающий в себя избранные тактики, должен обеспечивать соответствие архитектурным мотивам. В процессе отбора тактик следует учитывать два фактора. Первый — это, собственно, мотивы. Второй — это побочные эффекты, оказываемые реализующим тактику шаблоном на прочие атрибуты качества.

Разберем для примера тактику, которая, как правило, используется для реализации модифицируемости, — интерпретатор. Введение в систему интерпретируемого языка спецификаций упрощает механизмы создания новых и корректировки имеющихся функций. Один из частных случаев применения интерпретатора — запись и исполнение макрокоманд. Интерпретируемым языком, регламентирующим структуру и оформление веб-страниц, является HTML. С одной стороны,

[1] В принципе, реакция на обнаружение препятствия в течение 0,1 с — это отнюдь не смертельное требование, однако следует учитывать, что речь идет о рынке товаров массового производства, где любое ограничение производительности процессора влечет за собой значительное снижение стоимости. Кроме того, нельзя забывать, что вследствие серьезной инерции остановить гаражную дверь довольно сложно.

интерпретатор прекрасно справляется с реализацией модифицируемости в период прогона, но с другой — оказывает сильное отрицательное воздействие на производительность. Таким образом, решение о применении интерпретатора следует принимать лишь в том случае, если модифицируемость имеет относительно бóльшую важность, чем производительность. Есть и компромиссное решение — задействовать интерпретатор лишь для отдельного элемента образца, а для всех остальных выбрать другие тактики.

Рассматривая изложенные в главе 5 тактики в свете установленных архитектурных мотивов, обнаруживаем, что важнейшими атрибутами качества являются производительность и модифицируемость. Среди тактик реализации модифицируемости нам известны «локализация изменений», «предотвращение волнового эффекта» и «откладывание времени связывания». Поскольку наши сценарии реализации модифицируемости в основном ориентированы на изменения в период проектирования системы, на первое место выходит тактика «локализация изменений». Кроме того, выберем «семантическую связность» и «сокрытие информации» и скомбинируем их для определения виртуальных машин соответствующих областей. В качестве тактик реализации производительности используем «ресурсопотребление» и «арбитраж ресурсов», конкретнее — «повышение вычислительной эффективности» и «выбор политики планирования». Итак, речь идет о следующих тактиках:

- *Семантическая связность* и *сокрытие информации*. Обязанности, связанные с пользовательским интерфейсом, передачей данных и датчиками, выводятся в отдельные модули. Их мы называем виртуальными машинами; поскольку на основе предполагаемой архитектуры требуется получить три разных продукта, эти модули, очевидно, будут различаться. Обязанности, связанные с диагностикой, также подвержены разделению.
- *Повышение вычислительной эффективности*. Вычисления, критичные с точки зрения производительности, следует сделать как можно более эффективными.
- *Разумное планирование*. Критичные с позиции производительности вычисления следует планировать в расчете на соблюдение требований по времени.

Архитектурный образец, полученный в результате сочетания упомянутых тактик, показан на рис. 7.2. Имейте в виду, что это лишь один из приемлемых вариантов, но отнюдь не единственно возможный.

2c. Конкретизация модулей и распределение функциональности посредством проекций

В предыдущем разделе мы говорили о том, как архитектурные мотивы качества через посредство тактик воздействуют на структуру декомпозиции модуля. Там же мы, по существу, определили типы модулей этапа декомпозиции. Теперь покажем, как эти типы модулей следует конкретизировать.

Конкретизация модулей. В схеме, приведенной на рис. 7.2, участвуют некритичные с точки зрения производительности вычисления; изображены они выше виртуальной машины, ответственной за передачу данных и взаимодействие с датчиками. Поверх виртуальной машины, как правило, исполняется приложение.

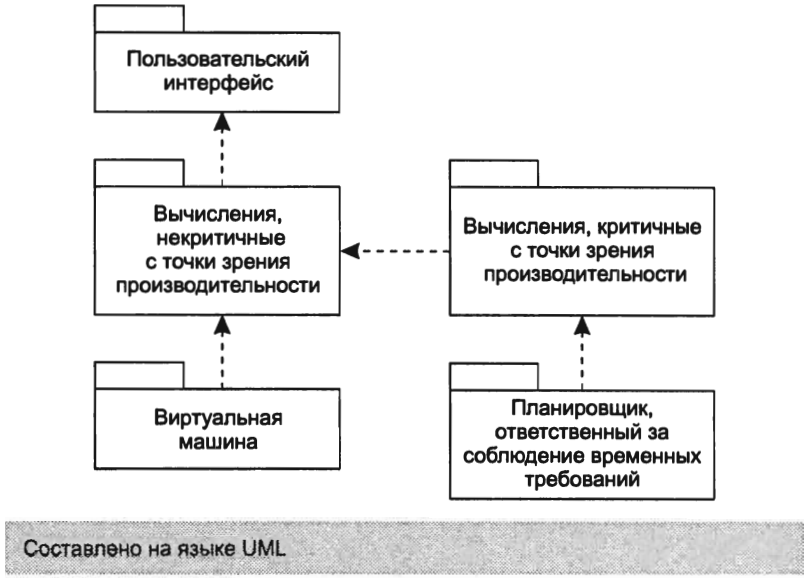


Рис. 7.2. Архитектурный образец, содержащий тактики реализации мотивов открывателя гаражной двери

Любая конкретная система в большинстве случаев состоит из нескольких модулей. Каждой функциональной «группе» соответствует один модуль, и все они будут отражены в образце. При распределении функциональности мы отталкиваемся от тех же критериев, которые характерны для функциональных методов проектирования, — в частности, для большинства объектно-ориентированных методов.

К примеру, обязанности, связанные с обнаружением препятствий и остановкой гаражной двери, вследствие наличия требований по времени относятся к категории критичных. Управление поднятием и опусканием двери в нормальном режиме не связано с какими-либо ограничениями по времени, поэтому эти действия, напротив, рассматриваются как некритичные. Туда же относим и диагностику. Итак, конкретизация модуля некритичных с точки зрения производительности действий с рис. 7.2 конкретизируется в виде модулей диагностики и поднятия/опускания двери, изображенных на рис. 7.3. Кроме того, ряд обязанностей — передача информации, считывание показаний датчиков и управление силовым приводом — отходит к виртуальной машине. Отсюда — два экземпляра виртуальной машины, которые также представлены на рис. 7.3.

Результатом выполнения данного этапа следует считать появление приемлемой декомпозиции модуля. Последующие этапы направлены на проверку адекватности декомпозиции требуемой функциональности.

Распределение функциональности. Применение принадлежащих к родительскому модулю элементов Use Case помогает архитектору лучше уяснить принципы распределения функциональности. Для покрытия всей предполагаемой функциональности иногда также требуется удалить имеющиеся дочерние модули или ввести новые. Наконец, любой элемент Use Case родительского модуля должен представляться последовательностью обязанностей в дочерних модулях.

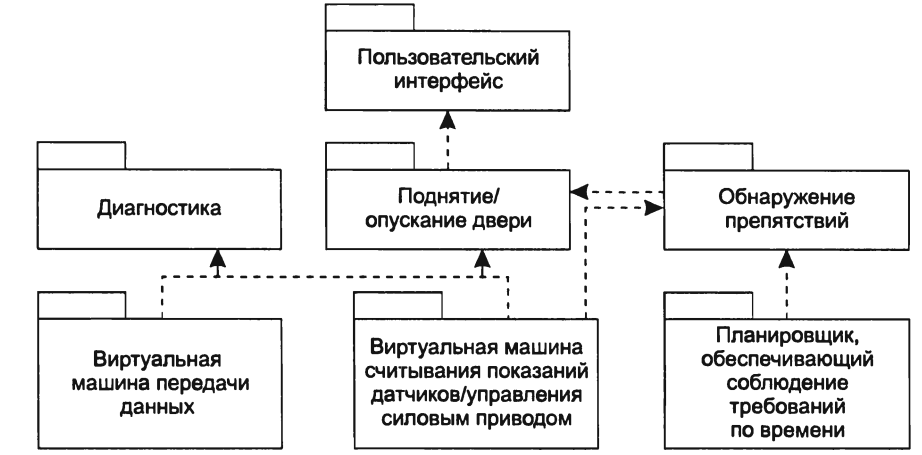


Рис. 7.3. Декомпозиция открывателя гаражной двери первого уровня

Распределение обязанностей между дочерними модулями, помимо прочего, сопряжено с выявлением путей обмена информацией. В результате между модулями устанавливаются отношения «производитель–потребитель», требующие записи. Определение механизмов обмена информацией на этом этапе проектирования не так уж важно. Проталкивается она или выталкивается? Передается в виде сообщений или вызываемых параметров? На все эти вопросы нужно будет отвечать позже. Сейчас нас должна интересовать лишь сама информация и распределение ролей «потребитель» и «производитель». ADD не предусматривает средств получения такого рода информации — для этого требуется детальное проектирование.

В некоторых тактиках содержатся оригинальные образцы взаимодействия между модулями различных типов. К примеру, тактика «посредник» типа «публикация–подписка» вводит образец «публикация» для одних модулей и «подписка» для других. Поскольку эти образцы взаимодействия формируют обязанности соответствующих модулей, их следует записывать.

Главное, чтобы по результатам выполнения вышеописанных этапов у вас появилась уверенность в том, что система сможет реализовать функциональные требования. С другой стороны, для того чтобы проверить соответствие атрибутам качества, распределенных обязанностей мало. Для анализа реализации таких атрибутов качества, как производительность, безопасность и надежность, требуется информация о динамическом размещении и размещении периода прогона. Следовательно, наряду с представлением декомпозиции модулей мы должны рассмотреть ряд других представлений.

Отображение архитектуры в представлениях. Несколько индивидуальных представлений мы рассмотрели в главе 2. Судя по нашему опыту работы с методикой ADD, для начала вполне достаточно взять по одному представлению из трех основных групп (декомпозиция модулей, параллелизм и размещение). Сам

по себе рассматриваемый метод не зависит от выбора представлений, и при необходимости демонстрации дополнительных аспектов — например, объектов периода прогона — можно ввести дополнительные представления. Ниже мы вкратце опишем механизм применения трех представлений общего характера в ADD.

- *Представление декомпозиции модулей.* Выше мы выяснили, каким образом представление декомпозиции модулей обеспечивает контейнеры для выявляемых обязанностей. Кроме того, при помощи этого представления выявляются основные отношения потоков данных между модулями.
- *Представление параллелизма.* В рамках представления параллелизма моделируются динамические аспекты системы, такие как параллельные операции. Это помогает выявить состязания за ресурсы, потенциальные случаи взаимоблокировки, проблемы с непротиворечивостью данных и т. д. Моделирование параллелизма в системе, как правило, сопряжено с установлением новых обязанностей модулей, которые фиксируются в модульном представлении. Кроме того, оно иногда приводит к обнаружению новых модулей — например, диспетчера ресурсов, который помогает разрешать проблемы параллельного доступа к дефицитным ресурсам.

  Проекция параллелизма относится к категории представлений «компонент и соединитель». Компоненты представляют собой экземпляры модулей в представлении декомпозиции модулей, а соединители — носители *виртуальных потоков* (virtual threads). «Виртуальный поток» описывает исполняемую ветвь, охватывающую всю систему или ее отдельные элементы. Его не следует путать с потоками (процессами) операционных систем, которые связаны с другими свойствами, — в частности, с распределением памяти/ресурсов процессора. На том уровне проектирования, о котором сейчас идет речь, эти свойства не представляют важности. С другой стороны, после принятия решений об операционной системе и размещении модулей на процессорах виртуальные потоки придется отобразить на потоки операционной системы. Эти действия связаны с детальным проектированием.

  В рамках представления параллелизма соединители представляют такие потоки, как «синхронизация с...», «запуск», «отмена» и «передача данных». Экземпляры модулей в представлении декомпозиции модулей показывают в совершенно определенном ракурсе — как средство, обеспечивающее понимание отображения между двумя представлениями. Следует иметь в виду, что точка синхронизации расположена в специальном модуле и, таким образом, ориентирована на корректное распределение обязанностей.

  Уяснить параллелизм в системе помогают нижеследующие элементы Use Case.

  - ◊ *Два пользователя выполняют одно и то же действие в одно и то же время.* Помогает выявить проблемы, связанные с состязанием за ресурсы и целостностью данных в целом. В нашем примере с гаражной дверью можно себе представить ситуацию, при которой один человек пыта-

ется закрыть дверь с пульта дистанционного управления, а другой открывает ее с помощью переключателя.

- ◊ *Один пользователь одновременно выполняет несколько действий.* Помогает выявить проблемы, связанные с обменом данных и управлением операциями. Вполне возможно, что пользователь решит открыть гаражную дверь в момент ее диагностики.
- ◊ *Запуск системы.* Здесь дается отличная перспектива постоянных операций системы и механизмов их инициализации. Данный элемент, кроме того, упрощает задачу принятия решения относительно стратегии инициализации — это может быть любая модель, включая «все параллельно» или «все последовательно». Если обратиться к нашему примеру, разумным представляется такой вопрос: зависит ли возможность запуска системы открывателя гаражной двери от готовности домашней информационной системы? Находится ли система открывателя гаражной двери в постоянной готовности? Может быть, она ожидает поступления сигнала или запускается и останавливается каждый раз при открытии и закрытии двери?
- ◊ *Выключение системы.* Помогает решить вопросы очистки — например, достижения и сохранения устойчивого состояния системы.

В нашем примере точка синхронизации присутствует в виртуальной машине считывания показаний датчиков/управления силовым приводом. Критическая секция производительности, равно как и секция поднятия/опускания двери, ответственна за считывание показаний с датчика. В момент выполнения операции, относящейся к секции поднятия/опускания двери, критическая система производительности имеет право прерывать операции виртуальной машины считывания показаний датчиков/управления силовым приводом. Для последней требуется механизм синхронизации. Это становится очевидным при рассмотрении виртуальных потоков критической секции производительности и секции поднятия/опускания двери — и та и другая обращаются к виртуальной машине считывания показаний датчиков/управления силовым приводом. Пересечение двух виртуальных потоков свидетельствует о необходимости внедрения механизма синхронизации.

Параллелизм также может быть изменяемым параметром; именно такой случай рассматривается в главе 14, посвященной линейкам программных продуктов. Для одних продуктов подходит последовательная инициализация, для других — параллельная. Если декомпозиция не предусматривает механизма изменения метода инициализации (например, путем обмена компонентов), значит, такую декомпозицию следует скорректировать.

- ✦ *Представление размещения.* Если в системе используется несколько процессоров или специализированное оборудование, его размещение иногда приводит к появлению дополнительных обязанностей. Представление размещения помогает обозначить и спроектировать размещение таким образом, чтобы гарантированно реализовать предполагаемые атрибуты качества.

Ее применение приводит к декомпозиции виртуальных потоков представления параллелизма на виртуальные потоки в рамках конкретного процессора, с одной стороны, и сообщения, которые, перемещаясь от процессора к процессору, инициируют каждое последующее действие, — с другой. Таким образом, представление размещения оказывается основой для анализа сетевого трафика и выявления потенциальной перегрузки.

Кроме того, представление размещения помогает принимать решения о необходимости введения дополнительных экземпляров модулей. К примеру, требования по надежности иногда предполагают дублирование критической функциональности на нескольких процессорах. Помимо прочего, представление размещения помогает формулировать аргументы «за» или «против» применения специализированного аппаратного обеспечения.

Создание представления размещения носит неслучайный характер. Как и в случае с представлениями декомпозиции модулей и параллелизма, провести распределение аппаратных компонентов помогают архитектурные мотивы. Тактики наподобие дублирования, размещая на нескольких процессорах ряд точных копий, обеспечивают достижение высокой производительности или надежности. Другие тактики — к примеру, механизм планирования в реальном времени — фактически запрещают размещение на нескольких процессорах. При размещении тех элементов, которые не предопределяются выбранными тактиками, на первый план выходят функциональные факторы.

Переход виртуального потока из одного процессора в другой налагает на различные модули дополнительные обязанности. Он выражает требование по обмену данными между процессорами. Обязанность по проведению этого обмена естественным образом ложится на тот или иной модуль, а значит, должна быть зафиксирована в представлении декомпозиции модулей.

Вопросы размещения в нашем примере связаны с разделением обязанностей между системой открывателя двери и домашней информационной системой. Какая из них будет ответственна за аутентификацию удаленных вызовов и какой протокол будет применяться для передачи данных между ними?

2d. Задание интерфейсов дочерних модулей

Согласно задачам ADD, интерфейс любого модуля должен демонстрировать предоставляемые и требуемые службы и свойства. В этом кроется его отличие от сигнатуры. Он документирует средства модуля, к которым можно обращаться со стороны.

Анализ и документирование декомпозиции в категориях структуры (представление декомпозиции модуля), динамизма (представление параллелизма) и периода прогона (представление размещения) — все это раскрывает допущения о взаимодействии для дочерних модулей, которые должны быть обязательно отражены в их интерфейсах. Модульное представление документирует:

- производителей/потребителей информации;
- образцы взаимодействия, требующие от модулей предоставления одних служб и использования других.

Представление параллелизма документирует:

- взаимодействие между потоками, ведущими к интерфейсу модуля, предоставляющего или использующего ту или иную службу;
- информацию об активности компонента — например, о наличии у него собственного исполняемого потока;
- информацию о действиях компонента, связанных с синхронизацией и упорядочиванием, а также (иногда) с блокированием вызовов.

Представление размещения документирует:

- требования по аппаратной части — например, касающиеся применения специализированного оборудования;
- требования по времени — например, о том, что скорость вычислений процессора должна быть не меньше 10 MIPS;
- требования по обмену данными — например, о том, что обновление информации должно проводиться не чаще, чем раз в секунду.

Все эти сведения должны быть отражены в документации интерфейсов модулей.

2е. Проверка и уточнение элементов Use Case, сценариев реализации качеств и ограничений, налагаемых на дочерние модули

Все рассмотренные до сих пор этапы ориентированы на составление плана декомпозиции модулей. Но, помимо прочего, эту декомпозицию следует проверить, а дочерние узлы — подготовить к собственной декомпозиции.

Функциональные требования. У каждого дочернего модуля есть некие обязанности, которые отчасти выводятся на основе анализа декомпозиции функциональных требований. Эти обязанности можно представить в качестве элементов Use Case данного модуля. Есть еще один вариант определения элементов Use Case — он предполагает дробление и уточнение Use Case более низкого уровня. К примеру, элемент Use Case, связанный с инициализацией системы в целом, можно разбить на ряд элементов инициализации подсистем. Поскольку аналитик получает возможность придерживаться уточнения, для этого подхода характерна отслеживаемость.

В нашем примере первоначальные обязанности открывателя гаражной двери выражаются в открывании и закрывании двери по требованию (локальным или удаленным способом); остановке закрытия двери при обнаружении препятствия в течение 0,1 с; взаимодействии с домашней информационной системой и обеспечении проведения удаленной диагностики. Декомпозиция этих обязанностей на соответствующие модулям функциональные группы выглядит следующим образом.

- *Пользовательский интерфейс.* Выявление запросов пользователей и их представление в форме, ожидаемой модулем поднятия/опускания двери.
- *Модуль поднятия/опускания двери.* Управление силовым приводом, обеспечивающим открытие и закрытие двери. Остановка движения двери при полном закрытии или открытии.

- *Обнаружение препятствий.* Фиксация момента обнаружения препятствия и последующая остановка двери в процессе опускания или перенаправление ее движения.
- *Виртуальная машина передачи данных.* Управление обменом данных с домашней информационной системой.
- *Виртуальная машина считывания показаний датчиков/управления силовым приводом.* Управление взаимодействием с датчиками и силовыми приводами.
- *Планировщик.* Обеспечение соблюдения детектором препятствий всех требований по времени.
- *Диагностика.* Управление обменом диагностической информацией с домашней информационной системой.

Ограничения. Существует несколько способов обеспечения соответствия ограничениям родительского модуля.

- *Декомпозиция соответствует ограничениям.* К примеру, ограничение, согласно которому возможно применение только одной операционной системы, можно удовлетворить путем задания этой операционной системы в качестве дочернего модуля. Этим все действия и ограничиваются.
- *Ограничение удовлетворяется одним-единственным дочерним модулем.* К примеру, ограничение, в соответствии с которым необходимо применение специального протокола, можно удовлетворить путем определения для этого протокола дочернего модуля инкапсуляции. Даже при условии назначения для ограничения дочернего модуля оно может быть как удовлетворено, так и не удовлетворено — в зависимости от декомпозиции этого модуля.
- *Ограничение удовлетворяется несколькими дочерними модулями.* К примеру, для реализации всех протоколов, необходимых для передачи данных в Интернете, требуется два модуля (клиент и сервер). Соответствие условию зависит от декомпозиции этих дочерних модулей и их координации.

Одно из ограничений в нашем примере выражается в необходимости взаимодействия с домашней информационной системой. Оно удовлетворяется одним дочерним модулем — виртуальной машиной передачи данных, которая способна выявлять ситуации отсутствия связи.

Сценарии реализации атрибутов качества. Сценарии качества также требуют уточнения и распределения между дочерними модулями.

- Любой сценарий реализации качества можно полностью удовлетворить путем декомпозиции, без каких-либо дополнительных воздействий. В таком случае он помечается как удовлетворенный.
- Сценарий реализации качества можно удовлетворить текущей декомпозицией, налагающей ограничения на дочерние модули. К примеру, возможна ситуация, когда введение уровней удовлетворяет сценарий модифицируемости, но он, в свою очередь, ограничивает образец использования дочерних модулей.
- Декомпозиция может быть нейтральной по отношению к сценарию качества — например, если сценарий практичности затрагивает те участки пользо-

вательского интерфейса, которые пока что не входят в декомпозицию. В таком случае для сценария следует выделить один из дочерних модулей.

- Возможна ситуация, при которой текущая декомпозиция оказывается не способной обеспечить соответствие сценарию качества. Если этот сценарий является значимым, декомпозицию стоит пересмотреть. В противном случае необходимо зафиксировать логическое обоснование декомпозиции, объясняющее, почему удовлетворение данного сценария не представляется возможным. Как правило, подобные ситуации возникают в результате принятия компромиссных решений, затрагивающих другие, возможно, более значимые сценарии.

Те сценарии реализации качества, которые в нашем примере признаны архитектурными мотивами, удовлетворяются и уточняются следующим образом.

- В разных продуктах из линейки продуктов реализованы разные устройства и элементы управления открытием и закрытием двери. В некоторых случаях среди них присутствует возможность управления с домашней информационной системы. Этот сценарий делегируется модулю пользовательского интерфейса.
- В разных продуктах используются разные процессоры. Необходимо сделать так, чтобы архитектуру конкретного продукта можно было вывести непосредственно из архитектуры линейки продуктов. Настоящий сценарий делегируется всем модулям. Каждый из них берет на себя обязательство не обращаться к тем специализированным возможностям процессора, которые не поддерживаются стандартными компиляторами.
- Если препятствие (человек или объект) обнаруживается во время опускания гаражной двери, она должна остановиться (или, в качестве альтернативы, открыться) в течение 0,1 с. Этот сценарий делегируется планировщику и модулю обнаружения препятствий.
- Открыватель гаражной двери должен быть готов к проведению диагностики и администрированию средствами домашней информационной системы, взаимодействующей с ним при помощи ориентированного на данный продукт протокола диагностики. Обязанности, связанные с этим сценарием, делят между собой модули диагностики и передачи данных. Модуль передачи данных в этом случае ответствен за протокол обмена данными с домашней информационной системой, а модуль диагностики — за координацию всех прочих механизмов взаимодействия, которые задействуются при диагностике.

Результатом этого этапа должна стать декомпозиция модулей на дочерние модули, каждый из которых принимает на себя определенные обязанности; кроме того, предполагается появление набора элементов Use Case, интерфейса, сценариев реализации качества и ряда ограничений. Этого вполне достаточно для того, чтобы перейти к следующему шагу декомпозиции.

Обратите внимание на то, как далеко (или, быть может, недалеко?) мы продвинулись за один шаг: в нашем распоряжении словарь модулей и их обязанностей; мы рассмотрели множество элементов Use Case и сценариев реализации качества и частично разобрались в их ветвлении. Мы выявили информационные

потребности модулей и характер их взаимодействия. Все эти сведения необходимо зафиксировать в логическом обосновании проекта, чем мы и займемся в главе 9 «Документирование программной архитектуры». С другой стороны, мы еще не затрагивали большинство деталей. Мы не знаем языков обмена данными между модулем пользовательского интерфейса и модулем поднятия/опускания двери. Остается неизвестным алгоритм, который будет ответствен за обнаружение препятствий. Мы не имеем даже самого общего представления о взаимодействии критичных и некритичных секций производительности.

Но! Если речь идет о проектировании масштабной системы, того, что мы выяснили, достаточно для распределения функций между рабочими группами и придания им начального ускорения (желательно ногой). В случае проектирования небольшой системы (какой, в частности, является открыватель гаражной двери) можно переходить непосредственно к следующему шагу и искать ответы на поставленные вопросы.

7.3. Формирование рабочих групп

Когда первые несколько уровней структуры модульной декомпозиции архитектуры приобретают относительную стабильность, можно переходить к распределению модулей между группами разработчиков. В результате должно появиться представление распределения рабочих обязанностей, которое мы рассматривали в главе 2. В нем модули либо распределяются между существующими рабочими единицами, либо назначаются новым.

Не так давно — а именно в 1968 году — в литературе проходила дискуссия относительно тесной взаимосвязи между архитектурой и компанией-разработчиком. В работе [Conway 68, 29] эта проблема оценивается следующим образом.

> Возьмем любые два узла системы — x и y. Либо они соединены ветвью, либо нет. (Другими словами, либо обмен данными между ними имеет какое-то значение в контексте функционирования системы, либо нет.) Если ветвь имеется, значит, две (не обязательно явно выраженные) рабочие группы X и Y, которые разрабатывали эти два узла, очевидно, как-то согласовывали спецификацию интерфейса, через который они должны были обмениваться информацией. Если же между x и y нет никакой ветви, значит, эти подсистемы не передают друг другу никаких данных, что со всей ясностью предполагает отсутствие предмета для обсуждения между разрабатывавшими их рабочими группами, — итак, связь между X и Y мы в этом случае установить не можем.

Согласно мысли Конвэя (Conway), структуру организации (по крайней мере, в том, что касается каналов взаимодействия между ее подразделениями) можно установить по структуре разработанной ею системы; при этом отношения между структурами организации и системы обязательно являются двунаправленными.

Воздействие архитектуры на формирование структуры организации очевидно. После согласования архитектуры проектируемой системы происходит распределение крупных модулей между рабочими группами и соответственно этому составляется декомпозиция обязанностей. Затем каждая группа вырабатывает индивидуальные правила работы (или принимается общесистемный набор таких

правил). Если речь идет о крупной системе, то рабочие группы могут относиться к разным субподрядчикам. Правила работы в разных случаях учитывают организацию досок объявлений и веб-страниц для общения между разработчиками, соглашения по именованию файлов и систему управления конфигурациями. Все эти тонкости могут отличаться от группы к группе, что, повторим, особенно характерно для крупных систем. Более того — для каждой группы устанавливаются процедуры контроля качества и тестирования, пути взаимодействия и координации действий с другими группами.

Таким образом, группы в рамках организации работают над модулями. Участники каждой отдельно взятой группы должны при взаимодействии друг с другом пользоваться средствами связи с высокой пропускной способностью — дело в том, что они постоянно обмениваются информацией, представленной в форме подробных проектных решений. Низкая пропускная способность вполне достаточна (на самом деле даже желательна) для средств связи между группами. (На этот счет у Фреда Брукса (Fred Brooks) свое мнение — по его убеждению, если не регламентировать обмен информацией между участниками одной группы, проект может провалиться.) Все эти принципы, естественно, справедливы лишь в том случае, если в системе проведено разделение задач.

Результатом несоблюдения этих проектных критериев становятся излишне сложные системы. Оказывается, что от структуры рабочих групп и качества их координации во многом зависит судьба любого крупного проекта. С чем может быть связана необходимость в сложном и активном взаимодействии между рабочими группами? Либо с тем, что излишне сложны механизмы взаимодействия между элементами, над которыми они трудятся, либо с тем, что требования к этим элементам не были предварительно в достаточной степени «ужесточены». В таком случае не обойтись без высокой пропускной способности средств связи *между* группами (не только внутри групп), сложных переговоров и согласований, а иногда — даже без переработки элементов и их интерфейсов. Подобно программным системам, рабочие группы должны стремиться к низкому сцеплению и высокой связности.

По какой причине структура рабочих групп отражает структуру декомпозиции модулей? Согласно принципу сокрытия информации — базовому принципу конструирования, регламентирующему структуру декомпозиции модулей систем, модули должны инкапсулировать, или скрывать, изменяемые детали. Для достижения этой цели интерфейсы строятся с расчетом на абстрагирование изменяемых аспектов и предоставление пользователям (в роли которых в данном случае выступают программные элементы других системных модулей) универсального, цельного набора служб. Каждый модуль при этом образует собственную небольшую *предметную область* (domain) — область специализированных знаний и опыта. Как показывают нижеследующие примеры, именно это естественным образом приводит в соответствие структуры рабочих групп и модулей декомпозиции.

- *Модуль, относящийся к уровню пользовательского интерфейса системы.* Интерфейс прикладного программирования, который он предоставляет другим модулям, никоим образом не связан с какими бы то ни было конкретными элементами пользовательского интерфейса (переключателями,

шкалами, диалоговыми окнами и т. д.), при помощи которых информация представляется человеку; связано это с тем, что последние подвержены изменениям.

- *Модуль, относящийся к планировщику процессов*, скрывает количество готовых процессов и алгоритм планирования. В качестве предметной области в данном случае выступает планирование процессов и список соответствующих алгоритмов.
- *Модуль физических моделей* архитектуры системы A-7E (см. главу 3). Он инкапсулирует уравнения, при помощи которых вычисляются показатели физической среды. Предметная область в данном случае включает в себя численный анализ (уравнения следует реализовать с расчетом на поддержание в числовом компьютере достаточной степени точности) и авиационную электронику.

Если мы рассматриваем модули как уменьшенные предметные области, абсолютно правомерно предположить, что распределение разработчиков по рабочим группам согласно их профессиональным знаниям было бы наиболее эффективным решением. Это возможно только в рамках модульной структуры. Некоторые организации иногда даже формируют специализированные группы, не связанные с какими-то определенными архитектурными структурами, — об этом речь пойдет во врезке «Организационные и архитектурные структуры».

Влияние организации (точнее, той группы, которая занимается конструированием описанной в архитектуре системы) на архитектуру не так очевидно, но от того не менее значительно, чем влияние архитектуры на организацию. Предположим, что вы — участник рабочей группы, специализирующейся на разработке системы управления базами данных. Вас распределили на разработку некоего приложения — неважно, какого именно. Как бы то ни было, любую стоящую перед вами задачу вы будете склонны рассматривать как связанную с базами данных, беспокоиться о том, к какой системе управления базами данных следует в данном случае обратиться и не стоит ли соорудить собственную, предполагать, что для извлечения данных используется механизм, аналогичный запросам, и т. д. Вы будете настаивать на том, чтобы архитектура предусматривала наличие явных подсистем для (скажем) хранения данных и управления ими, составления и реализации запросов. Человек из группы, специализирующейся на телекоммуникационных технологиях, напротив, будет рассматривать систему в категориях, характерных для его профессиональной ориентации, — по его мнению, базы данных достаточно отразить в одной (не слишком его интересующей) подсистеме.

ОРГАНИЗАЦИОННЫЕ И АРХИТЕКТУРНЫЕ СТРУКТУРЫ

Не успели мы написать раздел 7.3, в котором приводится пример связи между организационной и архитектурной структурой, как один знакомый специалист по телекоммуникациям предложил свой пример. Организация, о которой он рассказал, считает необходимым оперативно реагировать на жалобы клиентов и предложения о модификациях. Согласно этой схеме, исполнение каждого исходящего от клиента запроса на внесение изменений вменяется в обязанность определенному специалисту. Любое изменение предусматривает корректировку ряда архитектурных компонентов, а это значит, что участники группы реагирования на запросы заказчиков, вынужденные работать с системой в целом, должны стоять особняком от любых других групп, ответственных за те или иные компоненты. Таким образом, организа-

ционная структура, подстроенная исключительно под структуру архитектурную, оказывается неадекватной реалиям.

Такой расклад сначала сбил нас с толку. Но, приступив к подробному допросу информатора, мы выяснили, что каждая модификация в этой компании проводилась дважды: первый раз (оперативно) службой работы с заказчиками, а второй — теми группами разработчиков, которые были ответственны за соответствующие компоненты. Любые другие схемы способны в сжатые сроки угробить архитектуру — разве только каждый компонент в ней реализует для конечного пользователя одну, четко определенную функцию.

Теперь поподробнее о нашем последнем утверждении. Архитектура, как мы неоднократно заявляли, призвана удовлетворять многим, зачастую входящим друг с другом в противоречие, требованиям. Архитектура, в которой каждый компонент реализует для пользователя одну функцию, прекрасно удовлетворяет требованиям по модифицируемости — *если только модификация не затрагивает физические элементы, воздействующие на другие функции*. На этапе сопровождения, как в изложенном нашим коллегой контрпримере, такая архитектура локализует модификации любой конкретной функции в одном-единственном компоненте. С другой стороны, такая «функциональная» архитектура, естественно, не предусматривает возможностей повторного использования компонентов и совместного использования данных, что делает ее весьма неэффективной с точки зрения реализации.

На самом деле компания, о которой идет речь, стремилась довести повторное использование до максимума и, в расчете на достижение этой цели, организовала рабочие единицы в точном соответствии с компонентной структурой. Поскольку любая модификация (потенциально) касается нескольких организационных единиц и их действия необходимо согласовывать (читай: чем больше рабочих групп задействовано в процессе модификации, тем более замедляется реагирование), компании пришлось сформировать группу быстрого реагирования, но в результате каждую модификацию приходилось проводить дважды.

— LJB

В главе 1 мы говорили о том, каким образом на характер архитектуры влияют организационные факторы — прежде всего опыт, а также желание применить или развить те или иные навыки. Вышеприведенный сценарий являет собой конкретный пример этого явления. По мере того как компания продолжает работу в той или иной предметной области, она вырабатывает артефакты, применяемые для выполнения задач, а обязанность по их поддержанию ложится на ее организационные группы. Более подробно мы поговорим об этом в главах 14 и 15, посвященных линейкам программных продуктов.

7.4. Создание макета системы

После проектирования архитектуры и формирования рабочих групп можно приступать к конструированию макета системы. Цель этого этапа в том, чтобы обеспечить основополагающую возможность реализации функциональности системы в предпочтительном (с точки зрения задач проекта) порядке.

Согласно классической методике программной инженерии, следует «заглушать» секции кода — так, чтобы различные части системы можно было вводить в действие и тестировать по отдельности. О каких именно частях идет речь? Последовательность реализации всегда можно установить по характеристикам архитектуры.

В первую очередь следует реализовать те программы, которые связаны с исполнением и взаимодействием между архитектурными компонентами. В системе

реального времени это может быть планировщик, в системе на основе правил — процессор правил (с прототипом набора правил), управляющий их активизацией, в многопроцессной системе — механизмы синхронизации процессов, а в клиент-серверной системе — механизм координации действий клиента и сервера. Базовый механизм взаимодействия во многих случаях выражается в виде промежуточных программ стороннего производства; если это так, реализация заменяется установкой. Помимо базовой инфраструктуры передачи данных или взаимодействия имеет смысл ввести в действие ряд простейших функций, инициирующих механическое поведение. На этом этапе в вашем распоряжении уже появляется исполняемая система, которая, правда, только и делает, что мурлычет что-то себе под нос. Это та основа, на которую можно начинать насаживать дополнительную функциональность.

Теперь выбирайте — какие функциональные элементы следует ввести в эту систему. Решение в данном случае принимается под влиянием нескольких факторов: во-первых, из побуждения снизить риск, обратившись к наиболее трудным областям в первую очередь, во-вторых, исходя из возможностей и специализации имеющегося персонала, и, в-третьих, из соображений выбросить продукт на рынок как можно быстрее.

Приняв решение относительно введения в систему очередной порции функциональности, обратитесь к структуре использования (см. главу 2) — она сообщит о том, какие еще программные средства системы должны корректно исполняться (другими словами, развиться из состояния банальной заглушки в полноценные элементы), для того чтобы реализация выбранных функций стала возможной.

Таким вот образом в систему можно вводить все новые и новые элементы, пока они не закончатся. На любом из таких этапов действия по интеграции и тестированию не представляют большой сложности; равным образом легко обнаружить источник недавно появившихся неисправностей. Чем меньше приращение, тем более предсказуемы бюджет и график и тем больше диапазон решений по поставке у управленцев и специалистов по маркетингу.

Заглушенные элементы, несмотря ни на что, приближают момент завершения работы над системой. Поскольку заглушки относятся к тем же интерфейсам, которые будут присутствовать в окончательной версии системы, даже в отсутствие полноценной функциональности они помогают уяснить и протестировать взаимодействие между компонентами. Проверить это взаимодействие с помощью компонентов-заглушек можно двумя способами: путем генерирования жестко закодированных искусственных выходных данных или считывания таких данных из файла. Кроме того, они способны генерировать искусственную нагрузку, по результатам которой можно приблизительно определить, сколько времени уйдет на фактическую обработку данных в законченной рабочей версии системы. Это помогает на ранней стадии проектирования выявить требования по производительности системы, включая рабочие взаимодействия и узкие места.

Согласно Кусумано (Cusumano) и Селби (Selby), компания Microsoft выстраивает свою стратегию на основе эволюционного жизненного цикла поставки (Evolutionary Delivery Life Cycle). По той версии методики, которой пользуется Microsoft, «завершенный» макет системы создается на раннем этапе жизненного цикла продукта, а впоследствии с некоторой регулярностью (во многих случаях

ежедневно), строятся «рабочие» версии с сокращенной функциональностью. В результате получается рабочая система, о достаточности характеристик которой можно принять решение в любой момент и которую в любой же момент можно развернуть. Есть, впрочем, одна проблема, которую следует предусмотреть, — дело в том, что та рабочая группа, которая заканчивает свою часть системы первой, задает интерфейс, которому должны соответствовать все последующие системы. Это обстоятельство ставит в невыгодное положение наиболее сложные части системы — их разработка сопряжена со значительно большими объемами аналитических действий, вследствие чего вероятность первоочередного определения их интерфейсов сильно снижается. Таким образом, и без этого сложные подсистемы становятся еще более сложными. По нашему убеждению, все согласования по поводу интерфейсов следует проводить на этапе создания макета системы — при таком условии на последующих стадиях разработки можно будет обратить большее внимание на достижение эффективности.

7.5. Заключение

Процесс проектирования архитектуры должен проходить в согласии с анализом требований, однако дожидаться, пока анализ будет проведен во всей полноте, совершенно не следует. Приступать к проектированию архитектуры можно в момент окончательного выявления основных архитектурных мотивов. В некоторый момент, когда архитектура дойдет до определенного состояния (опять же, не обязательно будет полностью завершена), можно переходить к разработке макета системы. Она выступает в роли каркаса, на основе которого проводится итерационная разработка (одной из характеристик которой является возможность поставки в любой момент).

Трудно преувеличить важность для проектирования архитектуры сценариев реализации качества и тактик, описанных в главах 4 и 5. ADD — это нисходящий процесс проектирования, основывающийся на применении требований по качеству, исходя из которых принимается решение о привлечении того или иного архитектурного образца, и функциональных требований, которые обеспечивают конкретизацию типов модулей, соответствующих выбранному образцу.

Путем формирования каналов взаимодействия между группами разработчиков архитектура по-своему влияет на структуру разрабатывающей организации. Существующие организационные единицы с определенной специализацией и устоявшимися интересами в свою очередь оказывают обратное влияние на архитектуру.

7.6. Дополнительная литература

В работе [McConnell 96] эволюционный жизненный цикл поставки рассматривается как наиболее удачный представитель всей породы моделей жизненного цикла программного обеспечения. Эта модель, позволяющая осуществить выпуск на любой стадии разработки продукта, ориентирована на организации, ощущающие

давление по срокам вывода продуктов на рынок, установившие в качестве приоритетного направления достижение функциональных результатов. В сочетании с построением макета системы и вниманием к структуре использования характеристики каждой версии продукта можно реализовать таким образом, чтобы максимально упрочить положение на рынке.

Плодовитая и новаторская деятельность Кристофера Алегзандера (Christopher Alexander) на поприще создания образцов проектирования в архитектуре (имеется в виду постройка зданий) послужила основой для разработки образцов программного проектирования. Всем тем, кто стремится к пониманию сущности образцов проектирования, совершенно необходимо прочесть труд [Alexander 77]. (Кстати, эти знания пригодятся, если вам когда-нибудь придется строить дом.)

Из всех авторов, работающих в области образцов проектирования программ, чаще всего цитируются участники так называемой «великолепной четверки» [Gamma 95]. В исследовании [Buschmann 96] архитектурные стили рассматриваются как образцы проектирования — таким образом, эти две важнейшие понятийные области сводятся воедино.

«Мифический человеко-месяц» [Brooks 95] — это обязательное чтение для всех программных инженеров. В переработанной версии книги анализируются сильные стороны итерационной разработки на основе архитектуры, причем особый упор делается на то, как эти принципы применяются в компании Microsoft.

В издании [Bosch 00a] представлен оригинальный метод архитектурного проектирования, который отличается от ADD тем, что в нем сначала формируются предпосылки для реализации функциональности и уже потом на этой основе реализуются все остальные атрибуты качества.

Описание рационального унифицированного процесса содержится в работе [Kruchten 00]. Подробный анализ принципов разработки, принятых в компании Microsoft, дается в издании [Cusumano 95].

7.7. Дискуссионные вопросы

1. Исходя из архитектуры, составляются рабочие группы. Они занимаются конструированием модулей, из которых состоит архитектура. Итак, в большинстве случаев в структуре рабочих групп отражена модульная декомпозиция. Каковы преимущества и недостатки формирования рабочих групп исходя из компонентной структуры или на основе какой-то другой архитектурной структуры — например, структуры процессов?
2. ADD предусматривает метод «разбиения» требований. В первую очередь удовлетворяются архитектурные мотивы и только после этого в контексте подстроенного под них решения осуществляются попытки удовлетворить всем остальным требованиям. Какие еще существуют методы разбиения применительно к проектной стратегии декомпозиции? Почему ни одна декомпозиция не дает возможности удовлетворить всем требованиям?
3. Какие еще методики помогают построить первоначальную версию программной архитектуры или архитектуры системы? Как в рамках этих методик

решаются вопросы соответствия функциональным, коммерческим требованиям и требованиям по качеству?

4. Как ADD соотносится со специализированными методиками проектирования в категориях конечного результата, времени и ресурсов, необходимых для реализации сравниваемых методов? В каких случаях лучше обращаться к ADD и в каких — к специализированным методам?

ОТКУДА БЕРУТСЯ СТАНДАРТЫ?

Пару лет назад в Usenet проскользнула занимательная тема. Обсуждался следующий вопрос: почему в США принята весьма странная стандартная ширина железнодорожной колеи — 4 фута 8 $^1/_2$ дюймов. Выводы, которые были сделаны, одинаково справедливы для всех областей технологии, в которых применяются стандарты.

Оказалось, что причин принятия такого стандарта две: обратная совместимость с существующими системами и опыт железнодорожных архитекторов. Дело в том, что первые разработчики американской железнодорожной системы учились в Великобритании, работали с британским инструментом и равнялись на подвижной состав британского образца. Это обстоятельство лишь переводит вопрос на другой уровень: почему эта причудливая ширина колеи применялась в Британии?

Говорят, что британцы пользовались этим стандартом по точно тем же двум причинам, что и американцы, — из-за необходимости обеспечить совместимость со старыми образцами и исходя из опыта архитектора. Вагонетки (предшественники железных дорог) строились в расчете именно на эту ширину колеи. Строители железных дорог, переквалифицировавшись из строителей вагонеток, поначалу пользовались старым инструментом.

Но и у вагонеток есть своя история. Первые вагонетки конструировались по образцу повозок. Соответственно, первые строители вагонеток пользовались теми же инструментами и отталкивались от тех же представлений, которые ранее применялись в строительстве повозок. Как бы то ни было, наш вопрос остается без ответа — почему расстояние между колесами повозок должно было быть равным 4 футам и 8 $^1/_2$ дюймам?

Это решение объяснялось современными условиями, соответствующими существовавшей на тот момент технологии: любое другое расстояние неизбежно приводило бы к поломке колес, потому что колеи на дорогах Британии в то время были именно этой ширины.

Дороги тоже не появились из небытия. Сдерживавший нововведения технологический стандарт обусловливал эту немаловажную особенность повозок. На этот раз причины следует искать в Риме. Первые магистральные пути в Британии строились римлянами, и происходило это в течение первых четырех веков нашей эры. Зачем римляне приняли такое решение? Да затем, что по дорогам такой ширины могли проехать их колесницы. Следовательно, ширина колеи в современных Соединенных Штатах связана с конструкцией римских колесниц, применявшихся двумя тысячелетиями ранее.

Но это еще не все.

Расстояние между колесами римской колесницы обусловливалось шириной упряжи, через которую в колесницу запрягалась лошадь. Упряжь не могла быть другой ширины, ибо в противном случае лошадь разрушала бы колею. Таким образом, мы можем уверенно утверждать, что сегодняшняя ширина рельсов в Соединенных Штатах установилась в результате следования ряду стандартов, каждый из которых обусловливался сочетанием технических факторов, ограничениями, свойственными системам предыдущих поколений, и опытом архитекторов. Итак, по совокупности факторов получается, что причиной, по которой ширина американских рельсов сегодня составляет именно ту величину, которую составляет, является стандартный круп римского боевого коня.

Все эти доводы, быть может, не слишком убедительны, но — вдумайтесь! — какие могут быть последствия легкомысленного отношения к стандартам. Когда Наполеон напал на Россию, движение его армии в Восточной Европе, сильно замедлились — все потому, что колея на местных дорогах не соответствовала римскому стандарту. Нарушив временные рамки, они попали в условия русской зимы. Что случилось после этого, мы все прекрасно знаем.

— RK

Глава 8
Моделирование условий полета. Конкретный пример архитектуры, ориентированной на интегрируемость

> Удивительно, но факт: на протяжении последних тридцати лет вычислительные мощности [в области моделирования условий полета] увеличивались почти экспоненциально. Сегодня у нас нет оснований предполагать, что эта тенденция претерпит изменения.
>
> *Лоренс Фогэрти [Fogarty 67]*

Мы вряд ли погрешим против истины, если заявим, что современные системы моделирования условий полета — это самые сложные из всех существующих программных систем. Они отличаются сильной распределенностью и жесточайшими требованиями по времени; помимо этого, они должны быть подготовлены к регулярным обновлениям, обеспечивающим высочайшую точность и соответствие непрерывно совершенствующимся воздушным судам и среде, которую они имитируют. Создание и сопровождение столь масштабных систем сопряжено с существенной трудностью проектирования ряда элементов:

- функционирование в жестких условиях реального времени;
- модифицируемость, необходимая для отражения эволюции моделируемых воздушных судов и изменения их окружения;
- масштабируемость функций (одна из разновидностей модифицируемости) как средство расширения систем, с тем чтобы они могли как можно точнее моделировать реальные условия.

Как бы то ни было, даже по подзаголовку главы видно, что основной заботой архитекторов систем моделирования условий полета остается обеспечение *интегрируемости* (integrability). Несмотря на то что мы не рассматривали этот атрибут качества систем в главе 4, в контексте крупных систем — в особенности тех, которые разрабатываются распределенными группами или несколькими компаниями, — он часто оказывается определяющим. Интегрируемость выражает легкость сочетания элементов, разработанных не связанными друг с другом рабочими группами (в том числе и сторонними организациями), в целях удовлетворения предъявляемых к программному продукту требований. Аналогично всем остальным атрибутам качества, реализации интегрируемости служат архитектурные тактики (причем некоторые из них одновременно направлены на достижение модифицируемости). Эти тактики, в частности, предусматривают компактность, простоту и стабильность интерфейсов; применение предписанных протоколов; низкое сцепление и минимизацию зависимостей между элементами; использование компонентных каркасов; а также сопровождение ряда версий интерфейсов, которые, с одной стороны, допускают расширение, а с другой предусматривают функционирование существующих элементов в условиях исходных ограничений.

В этой главе мы обсудим трудности разработки систем моделирования условий полета, а также ориентированный на их преодоление архитектурный образец. Известный под названием «структурной модели» (Structural Model), он делает упор на следующие аспекты:

- простоту и общность подструктур системы;
- минимизацию сцепления вычислительных процессов, с одной стороны, и стратегий передачи данных и управления, — с другой;
- минимизацию типов модулей;
- минимизацию стратегий общесистемной координации;
- прозрачность решения.

Архитектурный образец, основывающийся на этом принципе, обеспечивает не только высокую интегрируемость, но и реализацию других атрибутов качества, необходимых для моделирования условий полета. Образец этот состоит из ряда более мелких элементарных образцов.

8.1. Связь с архитектурно-экономическим циклом

Наше внимание во время анализа нижеизложенного конкретного примера будет обращено на изучение того сегмента архитектурно-экономического цикла (architecture business cycle, ABC), который связывает предполагаемые атрибуты качества с архитектурой. ABC систем моделирования условий полета на основе образца «структурная модель» представлен на рис. 8.1. Рассматриваемые здесь системы моделирования принадлежат ВВС США. В роли конечных пользователей выступают пилоты и экипажи имитируемых воздушных судов. Системы

моделирования условий полета служат трем целям: во-первых, подготовке пилотов к управлению тем или иным воздушным судном; во-вторых, обучению экипажей операциям с размещенными на бортах системами вооружений; и, в-третьих, подготовке к выполнению конкретных боевых задач воздушных судов. Некоторые системы предназначены для индивидуального пользования, однако большинство все-таки ориентированы на одновременную подготовку нескольких экипажей к совместным боевым операциям.

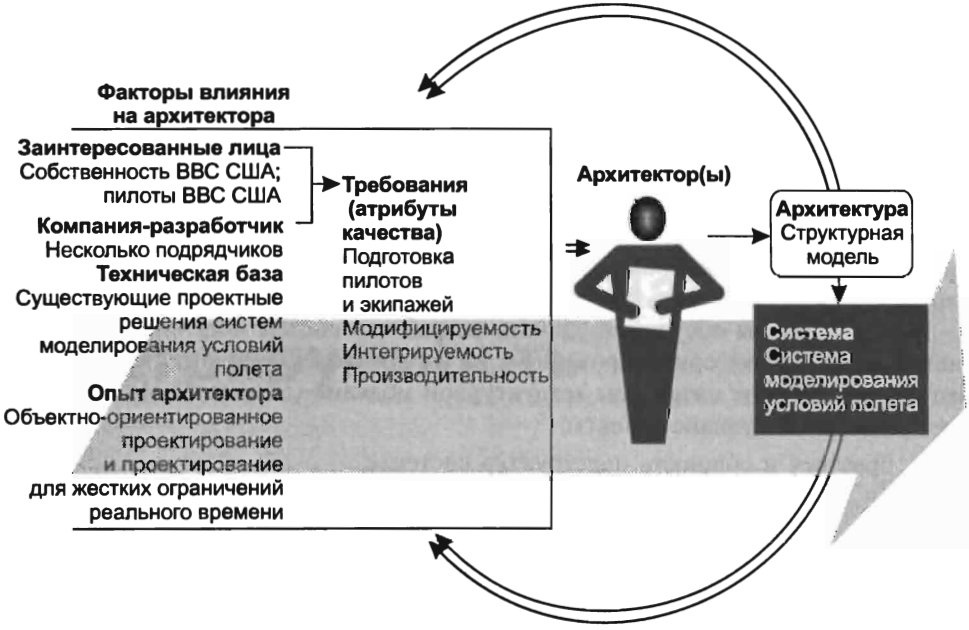


Рис. 8.1. Начальные этапы архитектурно-экономического цикла системы моделирования условий полета

Конструированием систем моделирования условий полета занимаются подрядчики, отбираемые на конкурентных торгах. Системы эти отличаются большим объемом (некоторые из них насчитывают до полутора миллионов строк кода), серьезной продолжительностью жизни (во многих случаях имитируемое воздушное судно состоит на вооружении в течение 40 лет или даже дольше) и строжайшими требованиями по функционированию в реальном времени и по точности (в таких условиях, как нормальный режим полета, аварийные маневры или отказ оборудования, модель должна в точности повторять поведение настоящего воздушного судна).

Появление образца «структурная модель» относят к 1987 году — к тому времени, когда ВВС начали анализировать возможности применения объектно-ориентированных методик проектирования. Необходимость в подобного рода изысканиях диктовались несовершенством традиционных решений, применявшихся при проектировании электронных систем моделирования условий полета, первые из которых появились еще в 1960-х годах. В частности, налицо были кон-

структивные проблемы (с разрастанием и усложнением систем экспоненциально разбухал этап интеграции) и трудности, связанные с жизненным циклом (стоимость проведения некоторых модификаций превышала стоимость исходных систем).

Как мы увидим, все эти проблемы удалось решить с помощью образца «структурная модель». В частности, на его основе был разработан тренажер-имитатор систем вооружений B-2, система тренировки экипажей C-17 и семейство тренажеров «Спецназ» (Special Operations Forces).

8.2. Требования и атрибуты качества

У пилотажного тренажера может быть три роли. Первая заключается в подготовке экипажа. Его члены размещаются на подвижной платформе в окружении инструментов, в точности повторяющих оборудование имитируемого воздушного судна, а перед собой наблюдают визуальные образы возможного окружения. Мы не намерены вдаваться в подробности устройства подвижной платформы и формирователя образов. Они находятся под управлением специализированных процессоров, выходящих за рамки рассматриваемой архитектуры. Пилотажный тренажер призван подготовить пилота и экипаж к управлению конкретным воздушным судном — в частности, к выполнению таких маневров, как дозаправка в воздухе и реагирование на атаки. Одной из важнейших составляющих тренировочного процесса является точность моделирования. К примеру, необходимо безупречно сымитировать особенности средств управления, проявляющиеся при выполнении тех или иных маневров. В противном случае пилот и экипаж могут получить неверное представление, что, в свою очередь, иногда приводит к катастрофическим последствиям.

Вторая роль пилотажного тренажера — моделирование окружающей среды. Как правило, она выражается в виде вычислительной модели, хотя в условиях совместной тренировки экипажей в моделировании могут принимать участие сторонние лица. Модель имитирует атмосферные условия, угрозы, вооружение и другие воздушные суда. К примеру, если экипаж обучается дозаправке, модель воздушного судна имитирует атмосферную турбулентность.

Третья роль, связанная с пилотажным тренажером, — это роль оператора моделирования. Любое учебное упражнение выполняет определенную задачу в определенных условиях. По мере выполнения упражнения оператор отслеживает действия пилота и экипажа и инициирует учебные ситуации. Иногда эти ситуации планируются заранее, но во всех остальных случаях их организует оператор. Существует ряд типичных ситуаций — в частности, неисправность оборудования (например, неточный выпуск шасси), вражеские атаки и турбулентность, вызванная грозой. Сидя за специальной консолью, оператор следит за действиями экипажа, инициирует неисправности и управляет моделирование окружающей среды. На рис. 8.2 приводится фотография зала с современными пилотажными тренажерами.

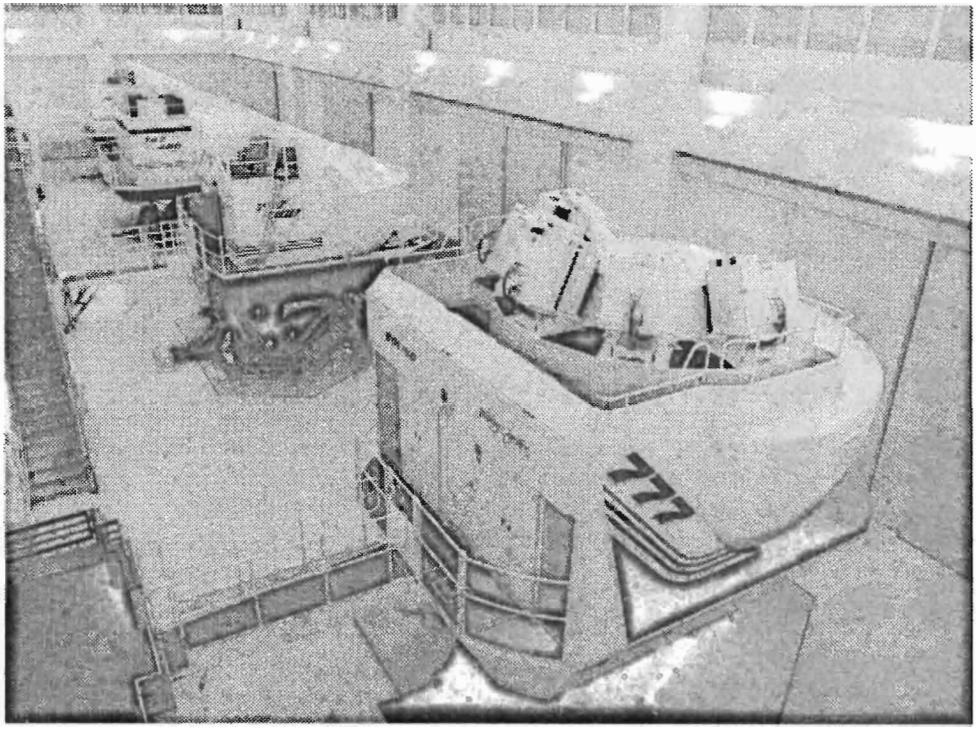


Рис. 8.2. Современные пилотажные тренажеры (приводится с разрешения компании Boeing)

Применение моделей

Модели воздушных судов и окружения предусматривают почти безграничный диапазон степеней точности. В качестве примера рассмотрим ситуацию моделирования воздушного давления на борт. Наиболее простая модель исходит из того, что воздушное давление зависит только от высоты над уровнем моря. Согласно более сложной модели, к фактору высоты прибавляются местные погодные условия. Моделирование образцов местных погодных условий требует дополнительных вычислительных мощностей, но зато учитывает нисходящие и восходящие потоки воздуха. Возможна и еще более сложная модель с повышенными вычислительными требованиями, в соответствии с которой воздушное давление обусловливается высотой, местными погодными условиями и поведением находящихся поблизости воздушных судов. Действительно, одной из причин возникновения турбулентности может быть недавнее прохождение другого борта в воздушном пространстве моделируемого воздушного судна.

Способность пилотажных тренажеров моделировать воздушные суда и окружающую среду с произвольной точностью достигается за счет максимального использования вычислительных возможностей (так было раньше и так же, судя по всему, будет и впредь). Поскольку обучение экипажей на тренажерах — это один из центральных элементов летной подготовки, даже незначительное повышение точности способно значительно улучшить результаты тренировки и, сле-

довательно, укрепить навыки экипажей. Таким образом, одно из основных требований, предъявляемых к системам моделирования условий полета, касается производительности.

Рабочие состояния

Система моделирования условий полета может находиться в нескольких состояниях.

- «Работа» (Operate) соответствует нормальному режиму функционирования системы моделирования в роли тренажера.
- «Настройка» (Configure) применяется при необходимости внесения в текущий учебный сеанс каких-либо изменений. К примеру, если инструктор собирается перейти от одиночного пилотажного упражнения на модель дозаправки в воздухе, он переводит систему в состояние настройки.
- «Остановка» (Halt) прекращает текущий сеанс моделирования.
- «Воспроизведение» (Replay) применяется для просмотра зафиксированных в журнале смоделированных ситуаций без взаимодействия с экипажем. Помимо прочего, этот режим используется для демонстрации экипажу записи выполненных им действий — в тех, к примеру, случаях, когда экипаж, столкнувшись с трудностями в управлении бортом, перестает реагировать на прочие факторы. В главе 5 («Реализация атрибутов качества») мы представили тактику тестирования «запись/воспроизведение». Здесь она используется в процессе тренировки.

Рассматриваемые в этой главе системы моделирования обладают следующими четырьмя свойствами:

1. *Ограничения по функционированию в реальном времени.* Частота смены кадров в системах моделирования условий полета должна соответствовать требованиям по точности. Объяснить, что такое «частота смены кадров», проще всего по аналогии с кинофильмами. Кадр — это снимок, фиксирующий тот или иной момент времени. Определенное количество кадров, сделанных последовательно в течение заданного периода времени, создает у наблюдателя впечатление непрерывного движения. В зависимости от условий восприятия устанавливаются разные значения частоты смены кадров. Системы моделирования условий полета, как правило, работают на частотах 30 или 60 Гц (при которых обновление производится 30 и 60 раз в секунду, соответственно). Все вычислительные операции при этом должны завершаться в пределах времени, отведенного на один кадр.

   Частота любого элемента системы моделирования должна быть кратна базовой частоте (причем коффициент кратности должен быть целым). Если, к примеру, базовая частота равняется 60 Гц, то более медленные элементы системы должны работать на частоте 30, 20, 15, 12, 10, 6, 5, 4, 3, 2 или 1 Гц. Нецелочисленных коэффициентов, приводящих к значениям наподобие 25 Гц, быть, таким образом, не должно. Причина этого ограничения кроется в необходимости жесткого согласования входных сигналов от датчиков.

Неприемлемой считается ситуация, при которой пилот, приступающий к выполнению поворота, ощущает результаты своих действий с задержкой — пусть даже минимальной (скажем, на одну десятую секунды). Задержки, происходящие из-за несогласованности, если даже они настолько малы, что не выводятся на уровень сознания, представляют серьезную проблему. В частности, они вызывают так называемую «тренажерную болезнь» (simulator sickness) — чисто физиологическую реакцию на недостаточно согласованные входные сигналы.

2. *Постоянная разработка и модификация.* Системы моделирования условий полета заменяют тренировки на настоящих воздушных судах, поскольку последние, во-первых, обходятся значительно дороже, а во-вторых, оказываются значительно опаснее. Для того чтобы максимально приблизить такую подготовку к реальным условиям, тренажер должен в точности повторять оригинальное судно. С другой стороны, как гражданские, так и военные воздушные суда постоянно модифицируются и совершенствуются. Следовательно, точного соответствия тренажера оригиналу можно добиться исключительно при условии его непрерывной модификации. Более того, чем дальше, тем больше в системы моделирования вводится новых ситуаций, охватывающих, с одной стороны, возможные неисправности, а с другой — внешние ситуации, например управление военным вертолетом в городской среде.
3. *Масштабность и сложность.* Для моделирования самых простых учебных ситуаций в пилотажных тренажерах, как правило, требуется ввод десятков тысяч строк кода. В сложных тренажерах совместной подготовки объем исчисляется миллионами строк. За последние 30 лет сложность систем моделирования условий полета увеличивалась экспоненциально.
4. *Географически распределенная разработка.* Существуют две причины, по которым разработка военных систем моделирования условий полета проводится распределенным методом, — одна техническая и одна политическая. С технической точки зрения отдельные элементы проекта требуют участия специалистов в разных областях, поэтому генеральные подрядчики имеют обыкновение заключать субдоговоры с компаниями разных специализаций. С другой стороны, высокотехнологичные области (к которым и относится разработка пилотажных тренажеров) — это всегда лакомый кусок для политиков, которые, естественно, прилагают все усилия к тому, чтобы занять в подобных проектах специалистов из своих округов. В любом случае, интегрируемость пилотажных тренажеров, которая и так проблематична вследствие объема и сложности кода, усложняется за счет протяженности каналов взаимодействия.

Еще две проблемы, связанные с пилотажными тренажерами, вынудили ВВС США искать новые решения.

1. *Крайне высокая стоимость отладки, тестирования и модифицирования.* Среди факторов, из-за которых стоимость тестирования, интеграции и модификации систем моделирования условий полета превышает стоимость разработки, — сложность этих программных продуктов, необходимость

функционирования в реальном времени и тенденция к регулярным модификациям. Все возрастающая сложность (и связанное с ней повышение стоимости) привела к тому, что при проектировании архитектуры подобного рода продуктов пристальное внимание стали уделять интегрируемости и модифицируемости.

Одним из последствий усложнения систем стало удорожание интеграции. К примеру, интеграция одной из крупных систем ВВС (объем кода в которой составил 1,7 млн строк) обошлась заказчику значительно дороже суммы, прописанной в бюджете. На очереди в тот момент были системы, объем которых должен был в 1,5 раза превысить этот показатель, и стоимость их изготовления казалась совершенно неподъемной. Так интегрируемость вышла на первый план среди всех архитектурных мотивов.

2. *Неясность соответствия между структурами программного продукта и воздушного судна.* Традиционно в качестве ведущей задачи по качеству при разработке систем моделирования условий полета выступала вычислительная эффективность в период прогона. В этом нет ничего удивительного, учитывая, с одной стороны, требования к производительности и точности, а с другой — то обстоятельство, что в первое время такие системы конструировались на платформах с крайне ограниченными ресурсами памяти и обработки данных. Традиционные решения пилотажных тренажеров основывались на циклическом следовании контурам управления. Те, в свою очередь, имели движущей силой задачи, запускавшие контур. Предположим, к примеру, что пилот выполняет поворот влево. Он передвигает руль направления и регулирует элероны; вследствие этих действий изменяется положение рулевых поверхностей, и за счет аэродинамических процессов борт начинает поворачивать. Соответствующая модель в тренажере должна выражать связь между элементами управления, поверхностями, аэродинамическими характеристиками и направлением движения воздушного судна. В рамках традиционной архитектуры систем моделирования эта модель заключалась в единый модуль поворота. Аналогичные модули отводились на горизонтальный полет, взлет, приземление и прочие ситуации. Таким образом, стратегия декомпозиции предусматривала анализ стоящих перед пилотом и экипажем задач, моделирование выполняющих эти задачи компонентов и максимально возможную локализацию вычислений.

   Такая архитектура способствует повышению производительности — каждая задача моделируется в рамках единичного модуля (или ограниченного набора модулей), вследствие чего продвижение данных, необходимое для проведения вычислений, сводится к минимуму. Недостаток этой архитектуры в том, что один и тот же физический компонент оказывается представлен сразу в нескольких моделях, а значит, и в нескольких модулях. Активное взаимодействие между модулями оказывает негативное влияние как на модифицируемость, так и на интегрируемость. Если предположить, что модуль, отвечающий за поворот, интегрирован с модулем, отвечающим за горизонтальный полет, и при этом в данных, предоставляемых модулю поворота, обнаруживаются ошибки, можно с достаточной степенью

уверенностью сделать вывод о том, что к тем же данным обращается модуль горизонтального полета. Таким образом, в ходе интеграции и сопровождения нужно было принимать во внимание многочисленные последствия сцепления.

На решение именно этих проблем ориентирован архитектурный образец под названием «структурная модель». Он рассматривается в оставшейся части этой главы. Вкратце его можно охарактеризовать так: это образец, предусматривающий применение объектно-ориентированных методик проектирования и направленный на моделирование подсистем и дочерних модулей контроллера воздушного судна. В целях контроля порядка работы подсистем тренажера и обеспечения точности он сочетает объектно-ориентированное проектирование с механизмом планирования в реальном времени.

8.3. Архитектурное решение

Эталонная модель системы моделирования условий полета представлена на рис. 8.3. Три рассмотренные выше роли (воздушное судно, окружающая среда и оператор) показаны во взаимодействии с экипажем и разного рода системами подсказок. Оператору, как правило, выделяется аппаратная платформа, отделенная от модели воздушного судна. Модель среды размещается либо на собственной аппаратной платформе, либо на операторской станции.

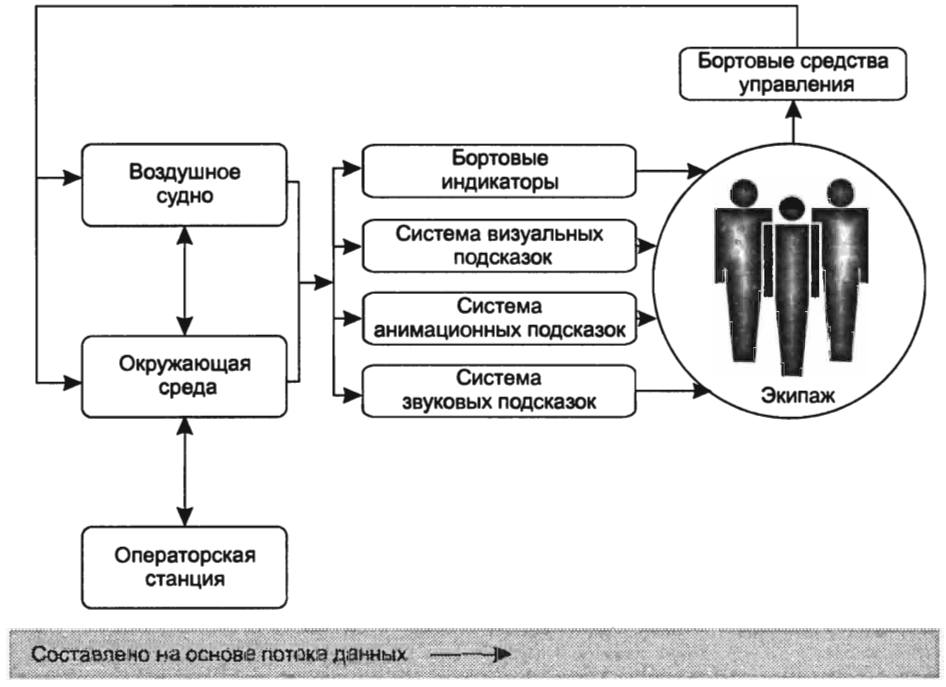


Рис. 8.3. Эталонная модель системы моделирования условий полета

Причины логического отделения операторской станции от остальных частей очевидны. Инструкторская станция предназначена для выполнения оператором функций управления и наблюдения за действиями тренируемого экипажа. Два оставшихся элемента ответственны непосредственно за моделирование. В дополнительных пояснениях нуждается разделение ролей моделирования полета и окружающей среды. Рассмотрим для наглядности ситуацию запуска снаряда. Первоначально снаряд логически относится к воздушному судну — вплоть до момента попадания в воздушную среду, когда он переходит в ее «компетенцию». В течение некоторого времени после запуска аэродинамические параметры снаряда в основном обусловливаются близостью воздушного судна. Следовательно, по крайней мере на начальном этапе, моделирование аэродинамики должно проходить в тесной связке с бортом. Если бы снаряд считался постоянно принадлежащим к внешней среде, для моделирования его запуска потребовалась бы четко скоординировать воздушное судно и внешнюю среду. Поскольку первоначально он относится к борту, а после запуска передается внешней среде, аналогичный путь должно пройти управление.

Время в системе моделирования условий полета

В главе 5 мы говорили об управлении ресурсами как об одной из категорий тактик реализации рабочих задач. Наиболее важным ресурсом в контексте любой системы моделирования в реальном времени является само время. Пилотажный тренажер призван имитировать реальные условия — для этого он воссоздает реальные линии поведения с временным критерием. Таким образом, в момент, когда пилот выполняет любые действия со средствами управления тренажера, реакция последнего должна поступить за тот же промежуток времени, за который на соответствующее действие реагирует система управления реального воздушного судна. «За тот же промежуток времени» означает «не раньше и не позже». Слишком быстрое реагирование в контексте качества моделирования не менее пагубно, чем слишком медленное.

В системах моделирования условий полета применяются два принципиально различных способа управления временем: периодический и событийный. Периодическое управление временем используется в тех элементах, которые обязаны обеспечивать производительность в реальном времени (к таковым относится элемент моделирования воздушного судна), а событийное — там, где требования по производительности в реальном времени не так важны (в частности, на операторской станции).

Периодическое управление временем

Схема периодического управления временем исходит из фиксированного (модельного) кванта времени, продолжительность которого устанавливается исходя из частоты смены кадров. На этом основывается проводящееся системой планирование. Такая схема, как правило, связана с дисциплиной циклического планирования без прерывания обслуживания. Реализуется она путем итерации нижеследующего цикла.

- Установление начального модельного времени.

- Итерация двух следующих шагов вплоть до завершения сеанса.
  - Каждый из двух процессов вызывается на фиксированный (реальный) квант времени. Процессы вычисляют внутреннее состояние исходя из текущего модельного времени, а сообщают о нем — исходя из следующего периода модельного времени. Так, все вычисления завершаются в пределах кванта реального времени.
  - Модельное время увеличивается на квант.

При том условии, что все процессы смогут переходить из одного периода времени в рамках кванта к другому, периодическое управление временем гарантирует синхронизацию модельного и реального времени.

Обычно для реализации этой схемы требуется изменить обязанности отдельных процессов — уменьшить их настолько, чтобы все вычисления укладывались в рамки, заданные квантом. Задача по введению такого количества процессоров, которого будет достаточно для получения всеми процессами квантов вычисления, ложится на плечи проектировщика.

Событийное управление временем

Схема событийного управления временем похожа на применяемую во многих операционных системах схему планирования по прерываниям. Реализуется она путем итерации нижеследующего цикла.

- Помещение в очередь событий модельного события.
- Если в очереди остаются события,
  - производится отбор события с минимальным (то есть наступающим в самое ближайшее время) модельным временем;
  - текущее модельное время приравнивается к времени выбранного события;
  - для выбранного события запускается процесс, получающий возможность вводить в очередь событий новые события.

Согласно этой схеме, изменения модельного времени происходят за счет помещения активизированными процессами новых событий в очередь и отбора планировщиком следующего события для его обработки. В условиях чистого событийного моделирования модельное время может проходить значительно быстрее (как в военной игре) или значительно медленнее (как при инженерном моделировании) реального.

Системы смешанного времени

Вернемся к планированию трех элементов пилотажного тренажера. На операторской станции планирование обычно выстраивается на основе событий, источником которых являются действия оператора. Модель воздушного судна, напротив, планируется на периодической основе. Модель окружающей среды допускает оба режима. Таким образом, для сцепления воздушного судна и среды в некоторых случаях требуется согласование режимов времени.

В системах моделирования условий полета периодическое моделирование времени (реализуемое в модели воздушного судна) сочетается с событийным

моделированием (которое в некоторых случаях реализуется в модели окружающей среды) и с другими событийными действиями непредсказуемого характера (например, взаимодействием с операторской станцией или с пилотом во время установки последним переключателя). С точки зрения любого из участвующих в этой связке процессов существует возможность применения сразу нескольких политик планирования.

Наиболее простая политика управления событиями в рамках процессора с периодическим планированием предполагает запуск периодической обработки сразу после синхронизации и ее завершение до начала какой бы то ни было непериодической обработки. А периодическая обработка в этом случае проводится за ограниченный период времени — до его истечения необходимо извлечь и обработать как можно больше сообщений. Необработанные сообщения откладываются до последующих интервалов непериодической обработки. При этом должно быть соблюдено требование о том, чтобы сообщения, полученные от каждого отдельно взятого источника, обрабатывались в порядке их поступления.

Передача данных из тех элементов системы, которые управляются на основе событий, элементам, управляемым периодически, считается непериодической и планируется вышеописанным образом. Данные, передаваемые из элементов с периодическим планированием в элементы с планированием событийным, рассматриваются как события. Разобравшись с принятыми в системах моделирования условий полета схемами управления временем, мы можем смело переходить к анализу соответствующего архитектурного образца. Образец этот ориентирован на модель воздушного судна, и именно с ее позиции в нем рассматриваются все вопросы управления временем.

Архитектурный образец «структурная модель»

Структурная модель (Structural Model) является архитектурным образцом; так мы определили ее еще в разделе 2.3. Этот образец, таким образом, содержит ряд типов элементов и конфигурацию для их координации в период прогона. В настоящем разделе мы намерены рассмотреть не только структурную модель как таковую, но и предпосылки ее создания. Как вы помните, модель воздушного судна может распределяться между несколькими процессорами. Следовательно, требуется обеспечить внутреннюю межпроцессорную координацию структурной модели воздушного судна, а также ее координацию с моделью внешней среды и операторскими участками модели, которые, теоретически, также могут быть размещены на разных процессорах.

Составляющие архитектурного образца «структурная модель» весьма обобщенно можно обозначить как *организующую* (executive) и *прикладную* (application).

- *Организующая* часть ответственна за все вопросы, связанные с координацией: планирование подсистем в реальном времени, синхронизацию процессоров, управление событиями с операторской станции, совместное использование и целостность данных.
- *Прикладная* часть отвечает за вычисления, связанные с моделированием условий полета, — конкретнее, за моделирование воздушного судна. Ее функции реализуются подсистемами и их дочерними модулями.

Сначала мы подробно рассмотрим организующие модули модели воздушного судна и только после этого перейдем к прикладным.

Организующие модули модели воздушного судна

На рис. 8.4 изображена структурная модель воздушного судна с детально раскрытым организующим образцом. Последний включает в себя следующие модули: временной синхронизатор (Timeline Synchronizer), генератор периодической последовательности (Periodic Sequencer), обработчик событий (Event Hander) и заменители (Surrogates), относящиеся к остальным элементам тренажера.

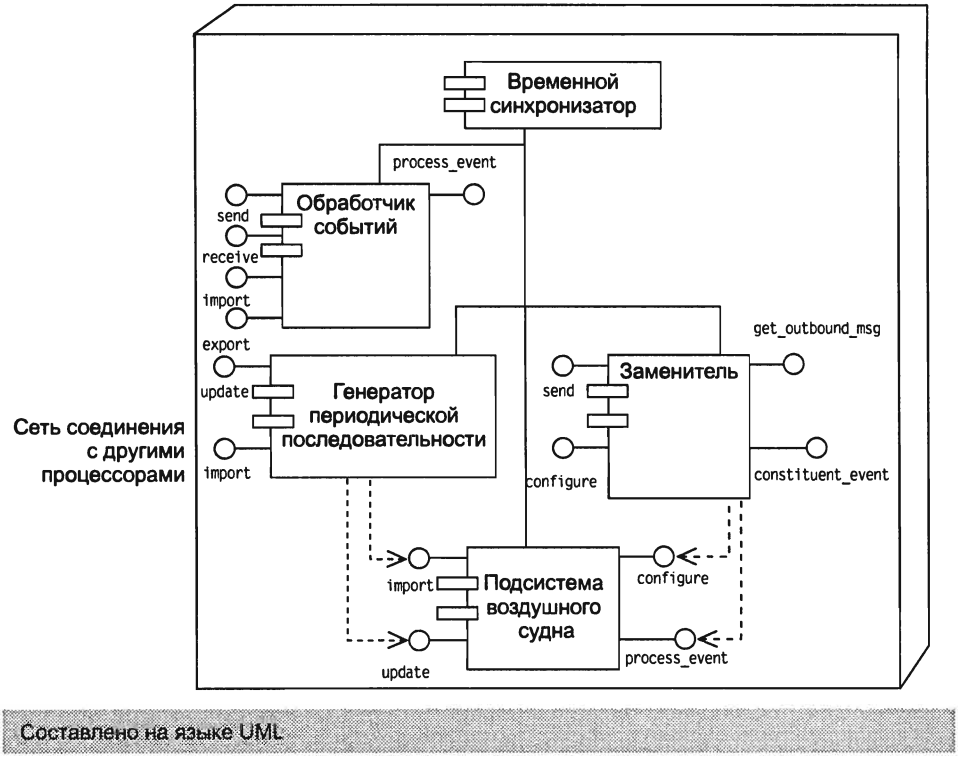


Рис. 8.4. Образец «структурная модель» процессора модели воздушного судна с детализацией организующей части

Временной синхронизатор

Временной синхронизатор — это основной механизм планирования модели воздушного судна. Кроме того, он ответствен за поддержание в модели воздушного судна внутреннего понятия о времени. Между тремя оставшимися элементами организующей части — генератором периодической последовательности, обработчиком событий и заменителями — необходимо распределить ресурсы процессора. Помимо прочего, временной синхронизатор поддерживает текущее состояние моделирования.

Временной синхронизатор передает и получает от трех оставшихся элементов данные и сигналы управления. Он же координирует время с другими частями тренажера — в частности, с процессорами, ответственными за тот или иной участок модели воздушного судна, у которых могут быть собственные временные синхронизаторы. Наконец, временной синхронизатор реализует политику планирования, направленную на согласование периодической и непериодической обработки. В целях обеспечения непрерывности приоритет отдается периодической обработке.

Генератор периодической последовательности

Генератор периодической последовательности отвечает за всю периодическую обработку, проводимую подсистемами системы моделирования. В частности, он активизирует выполнение подсистемами периодических операций согласно фиксированным графикам.

Генератор периодической последовательности предоставляет временному синхронизатору две операции. Операция import требует активизации генератором подсистем с целью исполнения ими собственных операций import. Операция update требует активизации генератором операций update подсистем.

Для выполнения действий по обработке генератору периодической последовательности требуются две вещи. Во-первых, у него должна быть возможность систематизировать знания, заключенные в графике. Под графиком (schedule) мы имеем в виду образцы вызовов составляющих, выражающие порядок и частоту распространения изменений среди реализуемых этими составляющими алгоритмов моделирования. Следование этим образцам, по существу, отражает течение времени в различных рабочих состояниях модели воздушного судна. Во-вторых, необходимо наличие некоего механизма диспетчеризации, при помощи которого генератор мог бы активизировать подсистемы через их периодические операции.

Обработчик событий

Модуль обработчика событий призван координировать всю проводящуюся подсистемами непериодическую обработку. Для этого он вызывает их непериодические операции.

Обработчик событий предоставляет временному синхронизатору четыре операции: configure (она, помимо прочего, применяется для запуска новых тренировочных заданий), constituent_event (применяется в тех случаях, когда событие нацелено на конкретный экземпляр модуля), get_outbound_msg (с помощью этой операции временной синхронизатор проводит непериодическую обработку в преимущественно периодических рабочих состояниях — например, в состоянии Operate) и send (она помогает контроллерам подсистем отправлять события другим контроллерам и сообщения другим системам).

Существуют два условия функционирования обработчика событий. Во-первых, используя знания о соответствии между идентификаторами событий и экземплярами подсистем, он должен уметь устанавливать получение события контроллером подсистемы. Во-вторых, у него должна быть возможность активизировать подсистемы и извлекать из событий необходимые параметры до их вызова.

Заменитель

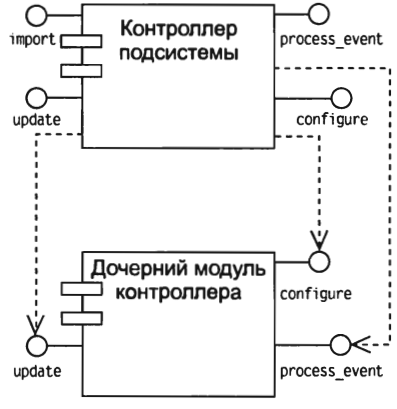
Заменителями называются приложения, реализующие тактику «введение посредника». Они ответственны за межсистемный обмен данными — конкретнее, между моделью воздушного судна, с одной стороны, и моделью окружающей среды или операторской станцией — с другой. Располагая информацией о физических характеристиках систем, с которыми они взаимодействуют, заменители обеспечивают представление, предоставляют протокол передачи данных и т. д.

Возьмем для примера ситуацию, в которой на операторской станции проводится мониторинг данных состояния модели воздушного судна и его результаты выводятся на дисплей для просмотра оператором. Взяв на себя управление процессором, заменитель собирает корректные данные, которые затем отправляет на операторскую станцию. Возможна и обратная ситуация, когда оператор в ходе тренировки экипажа устанавливает то или иное состояние модели воздушного судна. Заменитель, получив событие, передает его процессору событий, а тот оповещает соответствующие подсистемы.

В условиях применения заменителя периодический планировщик и обработчик событий могут не располагать детальной информацией об операторской станции или платформе, на которой функционирует модель окружающей среды. Все системно-ориентированные знания в таком случае встроены в заменитель. Распространение любых вносимых в эти платформы изменений в системе моделирования воздушного судна ограничивается заменителем.

Прикладные модули модели воздушного судна

Типы модулей прикладного субэлемента структурной модели воздушного судна показаны на рис. 8.5. Собственно, их всего два: *контроллер подсистемы* (subsystem controller) и *дочерний модуль контроллера* (controller child). Контроллеры подсистем передают данные от других и другим экземплярам контроллеров подсистем,



Составлено на языке UML

Рис. 8.5. Типы прикладных модулей

а также их дочерним модулям. Дочерние модули контроллеров передают данные только между собой и своими родителями; с другими дочерними модулями они не взаимодействуют. Управление они получают только от родителей и возвращают его им же. Эти ограничения запрещают дочерним модулям передавать данные и управление даже своим братьям. Запрет на сцепление дочерних экземпляров с кем-либо иным, кроме своих родительских элементов логически обосновывается задачами по достижению интегрируемости и модифицируемости.

В роли посредника при воздействии модификаций и интеграции всегда выступает родительский контроллер подсистемы. Все это согласуется с тактикой «ограничение обмена данными».

Контроллер подсистемы

Контроллеры подсистем взаимосвязывают ряд функционально-ориентированных дочерних модулей, решая тем самым две задачи:

- моделирование подсистемы в целом;
- посредничество в обмене данными управления и непериодическом обмене между системой и подсистемами.

Кроме того, они ориентируют дочерние модули на реализацию тренировочной функциональности — в частности, моделирование неисправностей и установку параметров.

Поскольку образец «структурная модель» ограничивает обмен данными между дочерними модулями контроллера, контроллеру подсистемы приходится устанавливать между ними самими, с одной стороны, и между ними и другими подсистемами — с другой, логические соединения. Входящие соединения удовлетворяют аналогичные потребности сторонних подсистем и заменителей. Эти соединения представляют собой наборы имен, посредством которых контроллеры подсистем осуществляют внутренние обращения к внешним данным. Допущения об установлении соединений принимаются при считывании или записи таких имен. Механизм фактического установления соединений определяется позже, на стадии детального проектирования, и выражается в изменяемом параметре образца (изменяемые параметры рассматриваются в главе 14 «Линейки продуктов»). Помимо установления соединений между собственными дочерними модулями и дочерними модулями других подсистем контроллер подсистемы исполняет роль посредника во взаимодействии между своими дочерними модулями — дело в том, что ограничение средств связи между ними предполагает запрет на прямой обмен данными.

Как мы уже говорили, система моделирования условий полета может находиться в нескольких состояниях. Посредством организующей части она переводится в конкретное состояние исполнения. После этого организующая часть сообщает контроллеру подсистемы о текущем состоянии. Для нас значимыми являются состояния «Работа» (operate) и «Стабилизация» (stabilize). В состоянии работы контроллер подсистемы выполняет нормальные вычисления, способствующие продвижению состояния моделирования. В состоянии стабилизации контроллер подсистемы управляемым образом завершает текущие вычисления

(при неконтролируемом завершении спорадические движения платформы могут повредить экипажу). Последовательность действий при этом выглядит так:

- Под прямым контролем организующей части происходит извлечение и локальное сохранение значений входящих соединений. Тем самым обеспечивается непротиворечивость данных и временная связность.
- Под контролем экземпляров организующей части проводится стабилизация алгоритмов моделирования дочерних модулей. После этого организующая часть признает подсистему в целом полностью стабильной.

Контроллеры подсистем *должны* выполнять следующие действия.

- В ответ на любое событие инициализировать себя и все свои дочерние модули набором начальных условий.
- Основываясь на знаниях возможностей дочерних модулей, передавать им запросы на неисправности и настройки параметров моделирования.

Наконец, контроллеры подсистем в некоторых случаях проводят реконфигурацию параметров задания, касающихся, в частности, вооружений, грузоподъемности и начального местоположения учебных заданий. Реализация этих возможностей проводится посредством периодических и непериодических операций, предоставляемых генератору периодической последовательности и обработчику событий соответственно.

Контроллеры подсистем должны предоставлять две периодические операции — update и import — и могут поддерживать две другие (непериодические): process_event и configure.

Update

Операция update заставляет контроллер подсистемы проводить периодическую обработку согласно текущему рабочему состоянию, которое предоставляется в качестве входного параметра. В состоянии operate операция update заставляет контроллер подсистемы:

1) устанавливать входные соединения и извлекать для дочерних модулей входные данные;
2) исполнять операции дочерних модулей в логической последовательности, с расчетом на распространение среди них изменений;
3) извлекать выходные данные дочерних модулей, удовлетворяя тем самым сторонние входящие соединения или исходящие соединения подсистемы.

Помимо исполнения функций генератора периодической последовательности этот алгоритм логически «склеивает» дочерние модули и придает моделированию связный, гармоничный характер. Этой цели служат вычисления, преобразования данных и изменения.

В состоянии stabilize операция update заставляет контроллер подсистемы провести одну итерацию алгоритма стабилизации и проверить, удовлетворены ли локально определенные критерии стабильности. Операция update предоставляет один выходной параметр, выражающий суждение контроллера относительно стабильности подсистемы. Предполагается, что такое заключение можно сделать локально, хотя, с другой стороны, такая возможность присутствует не всегда.

Контроллеры подсистем могут выполнять, а могут и не выполнять нижеследующие задачи.

- import. Операция import заставляет контроллер подсистемы завершать некоторые входные соединения — считывать их значения и проводить их локальное сохранение в расчете на последующее применение операцией update.

  Контроллеры подсистем предоставляют две непериодические операции: process_event и configure.

- process_event. Операция process_event применяется в преимущественно периодических рабочих состояниях (к числу которых, в частности, относится состояние operate) для вызова реакции контроллера подсистемы на событие, предоставляемое в виде входного параметра. Под эту категорию подпадает ряд событий операторской станции: process_malfunction, set_parameter и hold_parameter.

- configure. Операция configure применяется в таких преимущественно непериодических состояниях системы, как initialize. Она предназначена для установления именованного набора условий — например, конфигурации учебного устройства или тренировочного задания. Информация, необходимая контроллеру подсистемы для установления такого состояния, может предоставляться в виде входного параметра, ячейки памяти вторичного запоминающего устройства или извлекаться из базы данных. Контроллер подсистемы вызывает такие операции своих дочерних модулей, которые приводят к установлению требуемых состояний.

Дочерние модули контроллера

Дочерние модули контроллера модели воздушного судна во многих случаях имитируют его узлы — такие, как гидронасос, электрическое реле или топливный бак. С другой стороны, от них зависит обеспечение моделей, характерных для конкретного тренажера: сил и моментов, веса и равновесия, уравнений движения. Они локализуют отдельные узлы бортового оборудования: измерительные приборы, переключатели и дисплеи. Вне зависимости от того, какую функциональность они моделируют, дочерние модули контроллера считаются принадлежащими к одному типу модулей.

В общем, дочерние модули контроллера обеспечивают моделирование отдельных участков, или объектов, в рамках некоего функционального блока. Каждый из дочерних модулей предоставляет алгоритм моделирования, определяющий собственное состояние исходя из ряда факторов:

- предыдущего состояния;
- входных данных, представляющих соединения с логически смежными дочерними модулями;
- истекшего временного интервала.

Определение состояния проводится дочерним модулем по требованию контроллера подсистемы; он предоставляет все необходимые входные данные и извлекает данные выходные. Эта возможность называется *обновлением* (updating).

Дочерние модули способны порождать аварийные выходные данные, выражающие состояние неисправности. Во-первых, они моделируют в нормальном режиме работы такие изменения (например, износ), которые со временем приводят к возникновению неисправностей. Помимо этого, они могут инициировать состояние неисправности и выходить из него по требованию контроллера подсистемы.

Далее, дочерние модули контроллера имеют возможность устанавливать заданные значения параметров моделирования. Параметрами моделирования называются внешние имена рабочих параметров и критериев выбора, применяемых в алгоритмах моделирования дочерних модулей. Каждый дочерний модуль способен инициализировать себя некоторым известным состоянием. Подобно всем остальным действиям дочерних модулей, установка параметра и инициализация проводятся по требованию контроллера подсистемы.

Различия между обновлением, моделированием неисправностей, установкой параметров и инициализацией лежат в плоскости регулярности их применения контроллером подсистемы. Запросы на обновление, влияющие на течение времени в ходе моделирования, поступают дочерним модулям периодически. Запросы на все остальные возможности отправляются нерегулярно.

Реализуются все эти возможности через набор периодических и непериодических операций дочерних модулей, которые они предоставляют контроллеру подсистемы. update представляет собой единичную периодическую операцию, направленную на контроль периодического исполнения алгоритма моделирования. Получая внешние входные данные, дочерний модуль возвращает свои выходные данные через параметры операций. Все дочерние модули предоставляют две непериодические операции: process_event и configure.

Все логические взаимодействия между дочерними модулями проводятся через посредство контроллера подсистемы. В нем кодируются знания о том, как применять дочерние операции для удовлетворения требований моделирования, предъявляемых к подсистеме в целом. Вот некоторые из этих требований.

- Периодическое распространение среди дочерних модулей изменений состояния, осуществляемое через их операции update.
- Установление между дочерними модулями логических соединений при помощи входных и выходных параметров этих операций.
- Установление между логическим модулями, с одной стороны, и всеми остальными элементами модели — с другой, логических соединений с привлечением входящих и исходящих соединений подсистемы.

Неисправности дочерних модулей контроллера, согласно допущению, связываются с аварийными рабочими состояниями моделируемых реальных компонентов. Следовательно, решения о наличии и индивидуальностях этих неисправностей принимает проектировщик конкретного дочернего модуля; о своих решениях он сообщает проектировщику контроллера подсистемы, который на их основе реализует подаваемые подсистемой запросы о моделировании неисправностей. Неисправности, моделируемые подсистемой, могут напрямую не соответствовать тем, что относятся к дочерним модулям. Некоторые из них реализуются как группировки более примитивных неисправностей, моделируемых дочерними модуля-

ми. Обязанность по отображению неисправностей низкого уровня на неисправности уровня подсистемы ложится на контроллер.

Аналогичным образом, решения о наличии и индивидуальностях параметров моделирования принимаются проектировщиком дочернего модуля контроллера исходя из характеристик его алгоритма моделирования. Об этих решениях сообщается проектировщику контроллера подсистемы, который, принимая их во внимание, реализует запросы подсистемы и выполняет другие соответствующие их характеру задачи.

Макет системы

Все, что мы успели рассмотреть до настоящего момента, составляет макет системы (см. главу 7). В нашем распоряжении имеется структурный каркас системы без каких-либо деталей — другими словами, без фактической функциональности системы моделирования. На основе этого универсального каркаса можно смоделировать как вертолет, так и ядерный реактор. В процессе создания конкретной функциональной модели на макет «надеваются» те подсистемы и дочерние модули контроллеров, которые соответствуют поставленной задаче. Основой подобного рода конкретизации является процесс функционального разделения, к рассмотрению которого мы сейчас и обратимся.

Довольно странно, что всю систему моделирования условий полета, код которой исчисляется миллионами строк, можно полностью описать модулями шести типов: дочерними модулями контроллеров, контроллерами подсистем, временным синхронизатором, генератором периодической последовательности, обработчиком событий и заменителем. Такую архитектуру (сравнительно) легко сконструировать, понять, интегрировать, нарастить и модифицировать.

Не менее важным представляется то обстоятельство, что, располагая стандартным набором базовых образцов, можно создавать описывающие эти образцы формы спецификаций, кодовые шаблоны и экземпляры. Отсюда возможность проведения последовательного анализа. В условиях оперирования образцами архитектор может потребовать от проектировщика применения *исключительно* заданных стандартных блоков. Эта мысль, возможно, покажется слишком радикальной, но, тем не менее, чем меньше базовых стандартных блоков, тем больше внимания проектировщик может уделять функциональности — основной предпосылке разработки системы.

Распределение функциональности между дочерними модулями контроллера

Мы охарактеризовали архитектурный образец, на основе которого конструируется модель воздушного судна. Теперь следует рассмотреть принцип распределения между экземплярами модулей этого образца рабочей функциональности. Для этого необходимо обратиться к деталям моделируемого воздушного судна и, соответственно, определить экземпляры контроллеров подсистем. Конкретный механизм разделения зависит от характеристик систем воздушного судна, степени

его сложности, а также видов упражнений, в расчете на выполнение которых создается модель.

В этом разделе мы составим примерную схему разделения. Отталкиваться следует от намерения распределить функциональность между дочерними модулями контроллера согласно физическим характеристикам воздушного судна. Для достижения этой цели нам потребуется объектно-ориентированная методика декомпозиции. Она отличается рядом преимуществ.

- Благодаря относительно точному соответствию между составляющими воздушного судна и элементами системы моделирования в нашем распоряжении оказывается набор близких к реальности концептуальных моделей. Принцип взаимодействия узлов воздушного судна помогает понять механизм взаимодействия элементов его модели. Это обстоятельство помогает пользователям и наблюдателям получить более полное представление о системе моделирования — они могут экстраполировать свои знания об устройстве воздушного судна (о проблемной области) на его модель (область решения).
- По опыту работы с другими пилотажными тренажерами мы знаем, что любая модификация воздушного судна отождествляется с теми или иными его составляющими. Таким образом, место модификации модели соответствует аналогичной составляющей воздушного судна, а это значит, что любые вносимые в модель изменения прекрасно локализуются и определяются. В результате легче становится понять, как модификации воздушного судна повлияют на модель, и оценить финансовые и временные затраты на их реализацию.
- Уменьшается количество и размер интерфейсов модели. Причиной тому — жесткая семантическая связность внутри разделов, благодаря которой наиболее крупные интерфейсы размещаются внутри разделов, а не параллельно им.
- Локализация моделируемых неисправностей достигается за счет их связывания с конкретными узлами оборудования воздушного судна. Физическое соответствие упрощает анализ последствий неисправностей, а конечные реализации в этом случае отличаются достойной локализацией. Последствия неисправностей естественным образом распространяются данными, производящимися неисправным разделом. Последствия высокого порядка рассматриваются аналогично последствиям первого порядка. К примеру, протечка в гидравлическом соединении как последствие первого порядка моделируется непосредственно дочерним модулем контроллера. То обстоятельство, что эта утечка приводит к невозможности управления полетом, считается последствием более высокого порядка, но проявляется оно в результате естественного распространения модельных данных от дочернего модуля к контроллеру подсистемы и от одной подсистемы к другой.

Если разбить задачу моделирования воздушного судна на ряд блоков более приемлемого размера, на первый план выйдет конструкция воздушного судна. Конструкции как таковой, силам, которые она на себе испытывает, внешним по отношению к ней объектам, а также внутренним, подчиненным ей объектам со-

ответствует ряд групп. Как правило, распределение по группам выглядит следующим образом.

- *Кинетическая группа*. Элементы, представляющие прилагаемые к конструкции силы.
- *Системы воздушного судна*. Элементы, имеющие отношение к стандартным системам и являющиеся источниками разного рода мощностей или распределяющие энергию в пределах конструкции.
- *Авиационная электроника*. Узлы, предоставляющие вспомогательные функции, но не связанные напрямую с кинетическими характеристиками модели воздушного судна, системой ее управления или функционированием основных бортовых систем (например, системы радиосвязи).
- *Внешняя среда*. Элементы, отражающие среду, в которой функционирует модель воздушного судна.

Декомпозиция на группы

Декомпозиция на группы — это наиболее обобщенный вариант декомпозиции модели воздушного судна. Группы в свою очередь подлежат декомпозиции на системы, те — на подсистемы. Подсистемы содержат экземпляры контроллеров подсистем. Группы и системы не находят в архитектуре непосредственного отражения (никаких контроллеров групп в ней не представлено). Назначение их состоит в том, чтобы структурировать функциональность, распределяемую между различными экземплярами контроллеров подсистем. Такая декомпозиция проводится с участием *n-прямоугольных схем* (n-square charts).

n-прямоугольные схемы

n-прямоугольные схемы — это один из методов представления информации об интерфейсах системы. В нашем случае он помогает выразить связь между выбранными разделами. Ряд факторов, которые мы учитываем при принятии решений о разделении, связаны с интерфейсами разделов, а оценивать эти решения удобнее всего с помощью таких схем. Они, во-первых, помогают фиксировать входные и выходные данные модулей, а во-вторых, хорошо иллюстрируют применяемые в разных элементах проекта абстракции.

Пример n-прямоугольной схемы приводится на рис. 8.6. Прямоугольники, расположенные по главной диагонали, отражают разделы системы. Их входные данные приводятся в столбце раздела, а выходные — в соответствующей ему строке. Полный набор входных данных, таким образом, представлен объединенным содержимым всех ячеек столбца раздела. Полный набор данных выходных, соответственно, выражается объединением содержимого ячеек строки раздела. Поток данных из одного раздела в другой проходит в последовательности «вправо, вниз, влево и вверх».

n-прямоугольная схема, показанная на рис. 8.7, выражает интерфейсы между вышеперечисленными группами. Для краткости мы опустили все внешние по отношению к модели воздушного судна интерфейсы. Представление также упрощено — элементы данных на схеме выражают совокупные коллекции данных.

Рис. 8.6. n-прямоугольная схема

Кинетическая группа	Нагрузки	Вектор состояния воздушного судна	Местонахождение воздушного судна
Мощности	Группа систем воздушного судна	Мощности	
Инерциальное состояние	Нагрузки	Группа авиационной электроники	Данные о принадлежности
Атмосферные, топографические и метеорологические данные		Данные из окружающей среды	Группа окружающей среды

Рис. 8.7. n-прямоугольная схема декомпозиции на группы в предметной области моделирования воздушных судов

Интерфейсы не именуются и не типизируются. По мере анализа разделов и рассмотрения более ограниченных наборов элементов информацию можно будет детализировать. Инженеры систем, таким образом, могут дойти до представления всех примитивных объектов данных интерфейсов. В ходе детального проектирования можно определить типы и имена интерфейсов.

Не все модели воздушного судна соответствуют его реальной структуре. Аэродинамические модели выражают фундаментальные физические процессы, проходящие при взаимодействии судна с окружающей средой. Прямых аналогов с узлами судна в данном случае немного. Разделение этой области проводится на основе математических моделей и физических объектов, которые описывают динамику движения воздушного судна. Провести оптимальное разделение на основе математических моделей, которые оказывают воздействие на воздушное судно в целом, значительно труднее, чем на основе его физической структуры.

Декомпозиция групп на системы

Следующий этап — уточнение групп до систем. Как группа, так и система могут выступать в качестве единиц интеграции. Дело в том, что функциональность любой

системы представляет собой относительно автономное решение набора задач моделирования. На основе этих единиц удобно организовывать тестирование и проверку правильности. Раздел-группа — это, по существу, коллекция модулей кода, реализованных одним или несколькими инженерами. В рамках любой заданной группы можно выявить ряд систем. В качестве примера рассмотрим системы кинетической группы.

Системы в составе кинетической группы

Эти системы состоят из элементов, связанных с кинетическими характеристиками воздушного судна. Некоторые из них непосредственно участвуют в управлении движением судна и моделировании взаимодействия между судном и его рулевыми поверхностями, с одной стороны, и окружающей средой — с другой. В составе этой группы выделяются следующие системы:

- корпус;
- двигательная установка;
- шасси;
- инструменты управления полетом.

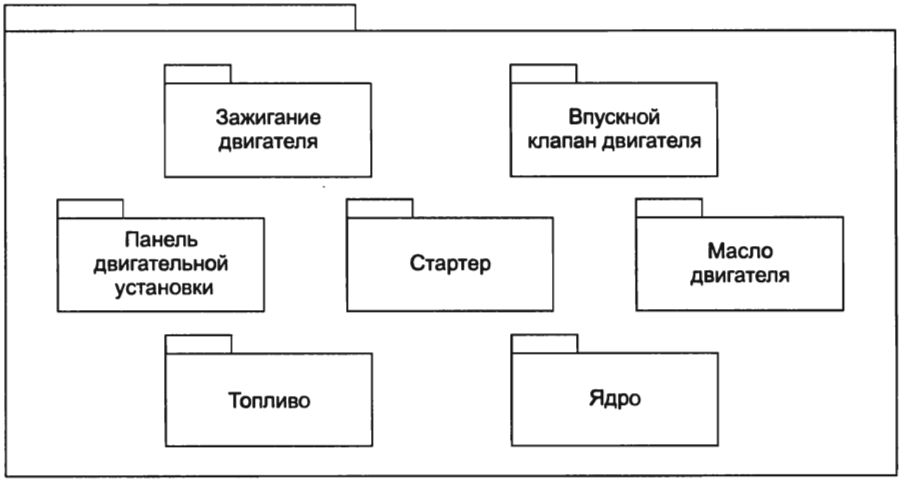


Рис. 8.8. Двигательная система воздушного судна

Все изображенные на рис. 8.8 подсистемы двигательной системы относятся к модели двигателей воздушного судна. Для представления нескольких двигателей создается соответствующее количество наборов переменных состояния, а также (там, где в этом есть необходимость) дублированные экземпляры объектов. Согласно своему основному назначению, эта система вычисляет тягу двигателя, моменты, вызываемые вращением узлов двигателя, а также силы и моменты, обусловливаемые распределением массы топлива.

Причина включения в группу топливной системы заключается в том, что основной интерфейс соединяет ее с двигателями. Эта система вычисляет силы, воздействующие на корпус воздушного судна вследствие движения топлива внутри баков, а также гравитационное воздействие массы топлива.

Мы определились с разделением функциональности, ее распределением по подсистемам и их контроллерам, а также с соединениями между подсистемами. Для того чтобы архитектуру можно было признать готовой, необходимо сделать еще две вещи:

- выявить экземпляры дочерних модулей контроллера двигательной подсистемы;
- аналогичным образом провести декомпозицию других групп, их систем и подсистем.

Итак, мы провели декомпозицию воздушного судна на четыре группы: кинетическую, систем воздушного судна, авиационной электроники и окружающей среды. Затем мы разбили кинетическую группу на четыре системы: корпуса, двигательную, шасси и инструментов управления полетом. Наконец, мы выполнили декомпозицию двигательной системы на ряд подсистем.

8.4. Заключение

В настоящей главе представлена архитектура систем моделирования условий полета, ориентированная в основном на реализацию трех атрибутов качества: производительность, интегрируемость и модифицируемость. В ходе реализации этих задач в конкретных проектах удалось добиться некоторой экономии. В частности, численность группы установки на месте сократилась до 50 % от предшествующего показателя. Связано это было с упрощением механизма обнаружения и устранения неисправностей. Реализация заданных атрибутов качества обеспечивается путем минимизации количества конфигураций типов модулей в архитектурном образце «структурная модель», ограничения взаимодействия между типами модулей и декомпозиции функциональности согласно предполагаемым модификациям реального воздушного судна.

Усовершенствования систем моделирования были достигнуты в основном за счет более глубокого понимания тщательно проанализированной и документированной программной архитектуры и более точного соблюдения ее постулатов. В работе Чэстек (Chastek) и Браунсуорд (Brownsword) [Chastek 96] описываются некоторые результаты, которые удалось получить благодаря применению образца «структурная модель»:

> Во время заводской приемки предыдущей информационной системы моделирования сопоставимого объема (B-52) было выявлено от 2000 до 3000 тестовых описаний (тестовых задач). В проекте на основе структурного моделирования этот показатель снизился до 600–700 описаний. Устранять проблемы стало легче, а причиной их возникновения во многих случаях оказывалось неверное толкование документации... В большинстве случаев специалистам удавалось локализовывать неисправности, не выезжая на место... Благодаря структурному моделированию процент брака по сравнению с предыдущими информационными системами моделирования снизился в два раза.

В начале главы мы обозначили три основные задачи по качеству, на реализацию которых ориентирован образец «структурная модель»: это производительность, интегрируемость и модифицируемость рабочих требований. Сейчас мы резюмируем механизмы реализации поставленных задач. В кратком виде эти сведения представлены в табл. 8.1.

Таблица 8.1. Механизмы реализации задач в образце «структурная модель»

Задача	Как реализуется	Применяемые тактики
Производительность	Стратегия периодического планирования	Статическое планирование согласно временному бюджету
Интегрируемость	Отделение вычислений от координирования Косвенные информационные и управляющие соединения	Ограничение взаимодействия Введение посредника
Модифицируемость	Минимизация типов модулей. Декомпозиция на основе физических элементов	Ограничение взаимодействия Семантическая связность Стабильность интерфейсов

Производительность

Основной задачей по качеству для образца «структурная модель» является обеспечение производительности в реальном времени. Достигается она в основном средствами организующей части и за счет применения стратегии периодического планирования. Для каждой подсистемы, которая активизируется организующей частью, устанавливается временной бюджет; с другой стороны, масштаб аппаратного обеспечения системы моделирования планируется в расчете на соответствие сумме всех временных бюджетов. Иногда для этой цели хватает одного процессора, в других случаях приходится вводить несколько процессоров. Для реализации производительности в реальном времени в рамках такой стратегии планирования требуется, чтобы суммарное время, отводимое участвующим в контурах управления подсистемам, соответствовало одному циклу системы моделирования. Таким образом, производительность в реальном времени обеспечивается за счет сочетания архитектурных образцов (конфигураций организующего модуля) и функциональной декомпозиции (механизма активизации экземпляров).

Интегрируемость

Согласно образцу «структурная модель» информационные и управляющие соединения между любыми двумя подсистемами намеренно сводятся к минимуму. В первую очередь дочерним модулям контроллера подсистемы запрещается передавать данные и управление своим братьям. Любой обмен данными или сигналами управления происходит только при посредничестве контроллера подсистемы. Таким образом, для интеграции в подсистему нового дочернего модуля контроллера требуется, во-первых, внутренняя непротиворечивость данных этого

контроллера и, во-вторых, правильность данных, передаваемых между контроллером и его дочерними модулями. Если бы каждый новый дочерний модуль имел возможность взаимодействия с другими дочерними модулями, все они оказались бы вовлечены в интеграцию; согласно принятой схеме, процесс значительно упрощается. Таким образом, интеграция сводится к линейной (а не экспоненциальной) задаче (с точки зрения количества дочерних модулей).

Интеграция любых двух подсистем также не сопряжена с непосредственным взаимодействием дочерних модулей; соответственно, задача, опять же, ограничивается обеспечением непротиворечивости данных, которые передаются этими двумя подсистемами. Возможность того, что введение новой подсистемы окажет воздействие на ряд существующих подсистем, существует, однако благодаря тому, что количество подсистем значительно меньше количества дочерних модулей, сложность этой задачи остается ограниченной.

Итак, интегрируемость в рамках структурной модели упрощается путем намеренного ограничения количества возможных соединений. У подобного ограничения есть обратная сторона — роль контроллеров подсистем зачастую сводится к пересылке данных дочерним модулям, в результате чего повышается сложность и снижается производительность. Впрочем, на практике преимущества значительно перевешивают издержки. Среди преимуществ следует также упомянуть возможность создания макета системы, предусматривающего инкрементную разработку и упрощенный механизм интеграции. Все проекты на основе структурного моделирования характеризуются простотой и беспрепятственностью интеграции.

Модифицируемость

Модифицируемость упрощается за счет минимизации конфигураций базового модуля (в которых проектировщику и специалистам по сопровождению, естественно, легче разобраться) и локализации функциональности, с тем чтобы в любой отдельно взятой модификации участвовало как можно меньше контроллеров подсистем и их дочерних модулей. Уменьшить количество соединений помогают n-прямоугольные схемы.

Более того, для всех подсистем, основанных на физических элементах, декомпозиция и модификации соответствуют физической структуре. Вероятность модификации тех подсистем, которые основаны не на физических элементах (например, уравнений движения), значительно ниже. Судя по отзывам специалистов, которым приходилось работать с образцом «структурная модель», побочные эффекты в ходе модификаций встречаются крайне редко.

8.5. Дополнительная литература

Исторические обзоры методик вычислений и инженерии, применявшихся при создании систем моделирования условий полета, содержатся в работах [Fogarty 67], [Marsman 85] и [Perry 66].

С 1987 года образец «структурная модель» претерпел значительные изменения. Ранние исследования по этой теме есть в изданиях [Lee 88], [Rissman 90] и [Abowd 93]. Отчет о результатах применения этого образца представлен в [Chastek 96].

Всем желающим подробно изучить функциональную декомпозицию систем моделирования условий полета мы рекомендуем ознакомиться с исследованием [ASCYW 94].

8.6. Дискуссионные вопросы

1. Жесткая взаимосвязь между структурой моделируемой системы, с одной стороны, и структурой системы моделирования — с другой, в числе прочего обеспечивает гибкость образца «структурная модель» во всем, что касается повторения характеристик моделируемой системы в случае изменений, расширения или сокращения. Предположим, что прикладная область — не моделирование. Насколько обоснованным в таком случае будет решение об использовании образца «структурная модель»? Почему? В каких условиях оно будет обоснованным и в каких — нет?
2. Ограничения, связанные с потоками данных и управлением между контроллерами подсистем и их дочерними модулями, отличаются жесткостью. Как вы оцениваете эти ограничения с позиции проектировщика и конструктора? Не кажется ли вам, что они слишком сильны?
3. Какие ограничения на решения проектировщика налагает макет системы? Как вы считаете — они полезны или вредны?

Глава 9
Документирование программной архитектуры

(в соавторстве с Феликсом Бахманом, Дэвидом Гарланом, Джеймсом Аймерсом, Ридом Литтлом, Робертом Нордом и Джудит Стеффорд)[1]

> Книги — как пчелы, разносящие живительную пыльцу мысли от одного человека к другому.
>
> *Джеймс Рассел Лоуэлл*

Уже неоднократно мы убеждались в том, что программная архитектура играет центральную роль в процессе разработки системы и в значительной степени определяет характер разрабатывающей ее компании. Она детально описывает систему и проект. На ее основе распределяются обязанности групп проектировщиков и исполнителей реализации, и именно она является основным носителем таких атрибутов качества системы, как производительность, модифицируемость и безопасность, — без объединяющей архитектурной концепции их не реализовать. Архитектура есть артефакт ранних стадий анализа, обеспечивающий способность применяемого проектного решения породить систему с приемлемыми характеристиками. Кроме того, на архитектуре в значительной степени основываются следующие за развертыванием операции изучения системы, ее сопровождения и минимизации вкладываемых ресурсов. Короче говоря, архитектура связывает воедино все этапы проекта и в таком виде представляет его перед многочисленными заинтересованными лицами.

Документирование архитектуры венчает процесс ее создания. Даже самая блестящая архитектура оказывается совершенно бесполезной, если никто не в состо-

---

[1] Феликс, Джеймс, Рид и Роберт — научные сотрудники Института программной инженерии; Дэвид — адъюнкт-профессор в Школе компьютерных наук при Университете Карнеги-Меллон; наконец, Джудит — доцент факультета компьютерных наук Университета Tufts.

нии в ней разобраться или (что еще хуже) если у заинтересованных лиц складывается о ней превратное представление. Если уж вы задались целью разработать достойную архитектуру, ее *необходимо* описать с достаточной степенью детализации, недвусмысленно и структурированно — так, чтобы любой желающий смог без затруднений найти в документации те сведения, которые ему требуются. В противном случае архитектура имеет шанс оказаться непрактичной, и затраченные на ее производство усилия полетят коту под хвост.

На материале настоящей главы вы узнаете, какую именно информацию об архитектуре следует вносить в ее документацию, и ознакомитесь с методами ее фиксации. Кроме того, мы обсудим существующие в настоящее время нотации, в том числе UML.

9.1. Варианты применения архитектурной документации

Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации архитектуры — другими словами, содержание документации зависит от предполагаемых вариантов ее применения. Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа. Между архитектурной документацией, предназначенной, скажем, для проведения анализа безопасности, и архитектурной документацией для изучения конструкторами должны существовать серьезные различия. С другой стороны, у этих вариантов будет мало общего с содержанием руководства для ознакомления, предназначенного новому сотруднику.

Архитектурная документация одновременно инструктивна и описательна. Ограничивая диапазон возможных решений, с одной стороны, и перечисляя принятые ранее проектные решения по системе — с другой, она обязывает соблюдать определенные принципы.

Из всего этого следует, что заинтересованные в составлении документации лица преследуют разные цели — от представленной в ней информации они требуют разной направленности, разных уровней детализации и разных трактовок. Предполагать, что один и тот же архитектурный документ будет одинаково воспринят всеми без исключения заказчиками, наивно. Стремиться следует к тому, чтобы любое заинтересованное лицо смогло быстро найти необходимую информацию, как можно меньше в процессе ее поиска натыкаясь на незначительные (с его точки зрения) сведения.

В некоторых случаях простейшим решением представляется создание нескольких документов, предназначенных для разных заинтересованных лиц. Впрочем, как правило, задача ограничивается созданием единого комплекта документации с дополнительными облегчающими поиск сведений инструкциями.

Согласно одному из фундаментальных правил технической документации в общем и документации программной архитектуры в частности, составитель

должен принимать позицию читателя. Без труда составленная, но трудная для восприятия (с точки зрения читателя — в данном случае заинтересованного лица) документация не найдет себе применения.

Для того чтобы понять, как структурировать документацию и повысить удобство ее использования, следует составить представление о заинтересованных лицах и разобраться в том, для чего она им может понадобиться. Как вы помните, еще в главе 2 мы выдвинули тезис о том, что архитектура — это в первую очередь средство коммуникации между заинтересованными лицами. Документация существенно упрощает их взаимодействие. Ряд примеров заинтересованных в архитектуре лиц и предположения о тех сведениях, которые они могут искать в документации, представлены в табл. 9.1.

Среди заинтересованных лиц встречаются как умудренные опытом люди, так и новички. Информация, которая им нужна, отличается не по содержанию, а по детализации — новичку она потребуется в «облегченном» виде. Архитектурная документация должна успешно вводить в курс дела всех, у кого есть такая потребность: новых разработчиков, инвесторов, временных участников проекта и т. д.

Одним из самых пристрастных потребителей архитектурной документации по прошествии некоторого времени становится сам архитектор — либо непосредственный проектировщик документированной архитектуры, либо его преемник. В обоих случаях это человек, проявляющий к рассматриваемой архитектуре недюжинный интерес. Новые архитекторы, естественно, стремятся узнать, как их предшественники справлялись с теми или иными задачами и почему они принимали определенные решения.

ДОКУМЕНТАЦИЯ КАК ВВЕДЕНИЕ В ПРОГРАММНУЮ АРХИТЕКТУРУ

Однажды мне довелось делать презентацию атрибутного метода проектирования (Attribute Driven Design, ADD; см. главу 7). Заказчики — представители одного из подразделений крупной производственной компании — видели в действии большинство наших разработок: метод анализа компромиссных архитектурных решений (ATAM, см. главу 11), реконструкцию (глава 10), а также наши предложения по линейкам продуктов (глава 14). Закончив вещать, я, весьма довольный собой, ждал реакции.

Оказалось, что не все так гладко. Представители заказчика, подтвердив свое намерение проводить разработку на основе архитектуры, признались, что численность группы разработчиков в их компании не позволяет применить на практике все, о чем мы им говорили, — на это уйдет несколько лет. А у них производственные планы и контракты, которые нужно выполнять. На то, чтобы воплотить в жизнь все наши рекомендации, у них просто не хватит ресурсов. Итак, наша задача состояла в том, чтобы грамотно направить их на путь архитектурной разработки, опустив все ненужные для начала подробности.

Дискуссия плавно перетекла в плоскость документации программной архитектуры и книги на эту тему, написанием которой мы в то время занимались. Заказчику хотелось узнать, каким образом документация помогает распространять среди персонала компании знания о принадлежащих ей программных продуктах. В конце концов мы договорились провести практикум по архитектурной реконструкции и документировать ее результаты согласно тем принципам, которые рассматриваются далее в этой главе.

Я всегда считал, что документацию следует создавать по окончании процессов проектирования и разработки. Необходимость создания архитектуры и документации обусловливается одними и теми же соображениями (коммуникативный, аналитический и образовательный факторы); разница лишь в ролях: архитектура — это побудительный мотив, а документация — производное.

Представители заказчика высказали иную точку зрения. Документирование программной архитектуры они рассматривали как идеальное упражнение для разработчиков. По их мысли, если уж разработчикам все равно приходится заниматься документацией, то почему

бы не составить для нее шаблон? В процессе заполнения им бы пришлось документировать разные представления (кстати, одной из наших задач было выбрать наиболее полезные для заказчика представления) и обсуждать пути удовлетворения проектируемым артефактом тех или иных задач по качеству. Таким образом, по мере составления документации они смогли бы углублять свои знания в области архитектуры.

Идея о применении документации в качестве средства обучения, честно говоря, раньше не приходила мне в голову. А ведь мысль-то сильная! Для тех, кому по роду деятельности приходится рыться в битах, представление об архитектуре и связанных с ней понятиях — большой прогресс. Изучение принципов программной архитектуры путем составления документации представляется мне весьма эффективным педагогическим приемом, который к тому же не требует от практикующей его компании серьезных расходов.

— LJB

Даже если архитектором остается тот же человек, документация прошлых архитектур служит для него репозитарием знаний, кладезем детальных проектных решений, помнить которые в силу их крайней многочисленности весьма проблематично.

Таблица 9.1. Заинтересованные лица и их коммуникативные потребности, удовлетворяемые архитектурой[1]

Заинтересованные лица	Способы применения
Архитектор и представляющие заказчика разработчики требований	Обсуждение конфликтующих требований и поиск компромиссов
Архитектор и проектировщики составляющих систему элементов	Разрешение состязаний за ресурсы, составление бюджета производительности и других бюджетов потребления ресурсов периода прогона
Конструкторы	Установление жестких ограничений применительно к нисходящим операциям разработки (и допустимых вольностей)
Тестировщики и сборщики	Регламентация корректного поведения совмещаемых элементов при тестировании методом «черного ящика»
Специалисты по сопровождению	Выявление областей влияния предполагаемых изменений
Проектировщики сторонних систем, взаимодействующих с рассматриваемой системой	Установление набора предоставляемых и требуемых операций, а также определение протокола их исполнения
Специалисты по атрибутам качества	Разработка модели, выступающей в качестве основы для функционирования аналитических инструментов: частотно-монотонного анализа возможности планирования в реальном времени, моделирования, имитационных генераторов, программ доказательства теорем, верификаторов и т. д. Подобные средства оперируют информацией о потреблении ресурсов, политиках планирования, зависимостях и т. п. Информации, содержащейся в документации архитектуры, должно быть достаточно для оценки различных атрибутов качества: безопасности, производительности, практичности, готовности и модифицируемости. Для анализа каждого из этих атрибутов характерны индивидуальные информационные потребности
Руководители	Формирование групп разработчиков согласно установленному распределению функций, планирование и распределение ресурсов проекта, контроль действий, выполняемых различными группами

продолжение

Таблица 9.1 (*продолжение*)

Заинтересованные лица	Способы применения
Руководители линеек продуктов	Установление принадлежности или отклонения потенциального члена семейства продуктов от их области действия; при необходимости — определение степени отклонения
Группа контроля качества	Подготовка к проверке соответствия, призванной подтвердить соблюдение конструкторами архитектурных директив

[1] Приводится по изданию [Clements 03] (адаптированная версия)

9.2. Представления

Вероятно, наиболее важным понятием документирования программной архитектуры является «*представление*» (view). В главе 2 мы определили программную архитектуру системы как «структуру структур системы, охватывающих элементы, их внешние свойства и отношения между ними». Кроме того, как мы установили, под представлением подразумевается отображение связного набора архитектурных элементов, составленного с позиции заинтересованных лиц и в расчете на использование заинтересованными лицами. Структурой называется сам набор таких элементов в том виде, в котором они существуют в программном или аппаратном обеспечении.

Также в главе 2 мы пришли к выводу о том, что программная архитектура как комплексный объект не поддается одномерному описанию. Для иллюстрации этого утверждения мы сравнили программную архитектуру с архитектурой зданий, заметив попутно, что подобного рода аналогиями не следует злоупотреблять. В архитектуре зданий применяется сразу несколько проекций: планы помещений, фасадные представления, электротехнические, водопроводные и вентиляционные схемы, схемы перемещения, проекции с прямым солнечным светом и солнечной радиацией, планы охранных систем и многие другие. Какое из этих представлений можно назвать архитектурой? Никакое. Какие из них являются *проводниками* архитектуры? Все.

С понятием представления, которое можно вкратце определить как способ фиксации структуры, связан основной принцип документирования программной архитектуры:

> Документирование архитектуры подразумевает документирование всех значимых представлений с последующей фиксацией сведений, относящихся одновременно к нескольким представлениям.

Принцип этот полезен тем, что он помогает разбить проблему документирования архитектуры на ряд менее обширных элементов. Этому разделению соответствует структура оставшейся части главы:

- выбор значимых представлений;
- документирование представления;
- документирование сведений, относящихся к нескольким представлениям.

9.3. Выбор значимых представлений

Как вы помните, в главе 2 рассматривался ряд структур и представлений. Какие представления следует считать значимыми? Для ответа на этот вопрос необходимо знать заинтересованных лиц и предпочтительные для них способы пользования документацией — лишь располагая этими сведениями, вы сможете составить удобный для них пакет документации. Любой вариант применения архитектуры — как средства постановки задач конструкторов, основы для изучения системы, восстановления ее свойств или планирования проекта — можно свести к отдельному заинтересованному лицу, предполагающему задействовать документацию архитектуры соответствующим образом. Не менее серьезное влияние на выбор представлений для дальнейшего документирования оказывают наиценнейшие для большинства заинтересованных в разработке системы лиц атрибуты качества. К примеру, *многоуровневое представление* (layered view) сообщает сведения о переносимости системы. По *представлению размещения* (deployment view) можно судить о производительности и надежности системы. И так далее. За отражение этих атрибутов качества в документации ратуют аналитики (а быть может, и сам архитектор), в задачу которых входит проверка архитектуры на предмет ее им соответствия.

Короче говоря, различные представления соответствуют отдельным задачам и вариантам использования. Именно по этой причине мы призываем к выбору того или иного представления или набора представлений. Их выбор зависит от задач потребителей документации. Представления ставят акценты на разные элементы системы и/или установленные между ними связи.

Таблица 9.2. Заинтересованные лица и виды архитектурной документации, представляющиеся для них наиболее полезными[1]

Заинтересованное лицо	Модульные представления				Представления «компонент и соединитель»	Представления распределения	
	Декомпозиции	Вариантов использования	Классов	Многоуровневое	Разные	Размещения	Реализации
Руководитель проекта	s	s		s		d	
Участник группы разработчиков	d	d	d	d	d	s	s
Тестировщики и сборщики		d	d		s	s	s
Специалисты по сопровождению	d	d	d	d	d	s	s
Разработчик приложений в рамках линейки продуктов		d	s	o	s	s	s

продолжение

Таблица 9.2 (*продолжение*)

Заинтересованное лицо	Модульные представления				Представления «компонент и соединитель»	Представления распределения	
	Декомпозиции	Вариантов использования	Классов	Многоуровневое	Разные	Размещения	Реализации
Заказчик					s	o	
Конечный пользователь					s	s	
Аналитик	d	d	s	d	s	d	
Специалист по инфраструктуре	s	s		s		s	d
Новое заинтересованное лицо	x	x	x	x	x	x	x
Архитектор и его преемники	d	d	d	d	d	d	s

[1] Приводится по изданию [Clements 03] (адаптированная версия)
Условные обозначения: d — подробная информация, s — отдельные детали, o — обзор, x — любая информация.

В табл. 9.2 приводится репрезентативная совокупность заинтересованных лиц, а также те представления, которые они, как правило, находят для себя полезными. Исходя из данных этой таблицы удобно делать выводы для конкретных случаев. Какие представления из тех, что подходят для заинтересованных в данной системе лиц, можно составить? Часть представлений, описанных в главе 2, отражены в табл. 9.2. В главе 2 представления разделены на три группы: модульные, распределения и «компонент и соединитель» (component-and-connector, C&C). Следовательно, архитекторы должны отвечать на три вопроса о любом проектируемом ими программном обеспечении:

1. Какова его структура как набора блоков реализации?
2. Какова его структура как набора элементов с заданным поведением и взаимодействием периода прогона?
3. Каковы его отношения с непрограммными структурами среды?

Существуют и другие представления. Представление отражает произвольный набор элементов системы и установленных между ними отношений. Следовательно, представлением может быть любое сочетание элементов и отношений, которое, по вашему мнению, имеет шансы оказаться интересным части заинтересованных лиц. Ниже приводится состоящая из трех этапов процедура подбора представлений для конкретного проекта.

1. *Составьте список возможных представлений.* Начните с составления для своего проекта таблицы заинтересованных лиц/представлений, отталкиваясь при этом от содержания табл. 9.2. Скорее всего, список заинтересованных лиц будет отличаться от отраженного в нашей таблице; в любом слу-

чае, вы должны учесть их всех. В столбцах выразите применимые к вашей системе представления. Некоторые из них (например, модульное представление и представление вариантов использования) носят универсальный характер; иные (многоуровневое представление, а также большинство представлений из группы «компонент и соединитель» — в частности, клиент-серверное представление и представление совместно используемых данных) подходят только для соответствующим образом спроектированных систем. Разобравшись с содержанием строк и столбцов, заполните все ячейки, отразив в них степень детализации сведений о каждом представлении, необходимую тем или иным заинтересованным лицам: здесь возможны произвольная детализация, обзор, средняя или высокая детализация.

2. *Комбинируйте представления.* Скорее всего, количество представлений, отраженных в составленном по результатам первого этапа списке, окажется недопустимым. Для того чтобы сократить список, найдите в таблице представления, требующие не более чем обзорной детализации или служащие ограниченному числу заинтересованных лиц. Проверьте, нельзя ли удовлетворить их запросы другим, более универсальным представлением. Затем найдите представления, которые можно комбинировать — выразить в одном сводном представлении информацию из нескольких исходных представлений. В небольших по масштабу проектах информативное содержание представления реализации, как правило, адекватно отражается в представлении декомпозиции на модули. Последнее, в свою очередь, хорошо сочетается с представлениями вариантов использования и многоуровневым представлением. Наконец, представление размещения можно скомбинировать с любым представителем группы «компонент и соединитель», отражающим распределение компонентов между аппаратными элементами, — например, с представлением процессов.
3. *Расставляйте приоритеты.* Представления, оставшиеся после выполнения второго этапа, должны соответствовать потребностям сообщества заинтересованных лиц. Теперь необходимо решить, какие из этих представлений имеют первостепенное значение. Конкретное решение зависит от деталей проекта; впрочем, вы в любом случае должны помнить, что полностью завершать работу над одним представлением, прежде чем приступать к следующему, совершенно не обязательно. Информация, детализированная на уровне обзора, представляет некоторую ценность, так что наши предпочтения — на стороне метода разработки материала «в ширину». Кроме того, интересы одних заинтересованных лиц в ряде случаев ставятся выше интересов других. Не забывайте о том, что руководитель проекта и менеджмент компаний-партнеров часто требуют внимания и предоставления той или иной информации.

9.4. Документирование представления

Стандартного шаблона документирования представлений не существует. Поэтому ниже речь пойдет о методике, доказавшей свою жизнеспособность в практической деятельности — *стандартной семичастной структуре* (seven-part standard

organization). Во-первых, вне зависимости от того, каким разделам вы отдадите предпочтение, ввести стандартную структуру совершенно необходимо. Распределение информации по отдельным разделам помогает составителю документации уверенно приступить к решению задачи и установить момент ее выполнения; что же касается читателя, то ему становится проще быстро отыскать интересующую информацию и пропустить все ненужное.

1. *Первичное отображение* (primary presentation) содержит перечень элементов представления и установленные между ними отношения. В первичном представлении должна содержаться (в форме словаря) та информация о системе, которую необходимо донести до читателя в первую очередь. Это, без сомнения, основные элементы и отношения представления, хотя в некоторых случаях они могут быть отражены не полностью. К примеру, в первичном отображении можно показать элементы и отношения, характерные для нормального режима работы, а сведения об обработке ошибок или исключений представить во вспомогательной документации.

   Как правило, первичное отображение составляется в графической форме. Дело в том, что большинство графических нотаций годятся только для первичного отображения — для всех остальных элементов документации они не приспособлены. Неизменным спутником графического представления должен быть перечень условных обозначений, содержащий объяснение использованной нотации и символики или хотя бы указывающий на источники получения этих объяснений.

   Иногда первичное отображение составляется в виде таблицы, которая по своему характеру прекрасно подходит для компактного представления больших объемов информации. Примером текстового первичного отображения является представление декомпозиции на модули системы A-7E, изложенное в главе 3. Требование о кратком выражении наиболее значимых сведений о представлении распространяется и на текст. В разделе 9.6 мы намерены обсудить методы составления первичного отображения средствами языка UML.

2. *Каталог элементов* (element catalog) предназначен для более детального описания элементов и отношений — как участвующих, так и не участвующих в первичном отображении. Архитекторы часто совершают одну и ту же ошибку — уделяя излишне серьезное внимание составлению первичного отображения, они забывают, что без дополнительной информации в нем мало толку[1]. К примеру, если на диаграмме показаны элементы A, B и C, необходимо составить относительно детальное описание сущности этих элементов, их назначения и ролей, причем разместить эту информацию лучше всего в словаре представления. Скажем, для представления декомпозиции на модули характерно наличие элементов-модулей, различных вариантов отношения «является частью», а также свойств, определяющих обязанности каждого модуля. В представлении процессов содержатся эле-

[1] Стараясь подчеркнуть роль первичного отображения как наброска документации представления, мы называем его архитектурным *эскизом* (architectural cartoon).

менты-процессы, отношения, определяющие синхронизацию и другие механизмы межпроцессного взаимодействия, а также свойства с временными параметрами.

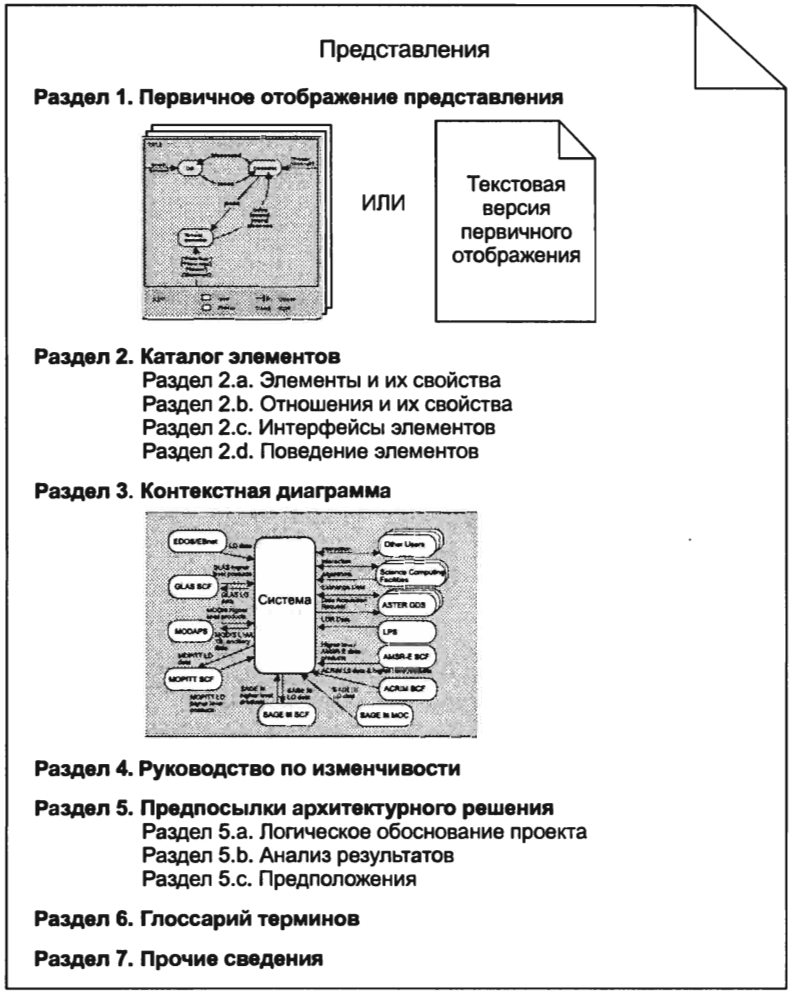
Кроме того, в каталоге обязательно должны разъясняться все элементы и отношения, значимые для данного представления, но по каким-то причинам не показанные в первичном отображении.

Чуть ниже мы обсудим еще два аспекта каталога элементов: поведение и интерфейсы элементов.

3. На *контекстной диаграмме* (context diagram) должно быть показано отношение изображенной в представлении системы к своему окружению из словаря представления. К примеру, представление «компонент и соединитель» подразумевает демонстрацию взаимодействия отдельных компонентов и соединителей с внешними компонентами и соединителями через посредство интерфейсов и протоколов.
4. *Руководство по изменчивости* (variability guide) демонстрирует способы применения изменяемых параметров, входящих в состав показанной в данном представлении архитектуры. В некоторых архитектурах принятие решений откладывается до более поздних стадий процесса разработки, что не отменяет необходимость в составлении документации. Примеры изменчивости обнаруживаются во всех линейках программных продуктов, архитектура которых предусматривает возможность создания множества конкретных систем (см. главу 14). В руководстве по изменчивости следует документировать все изменяемые параметры архитектуры, включая:
   - ◊ альтернативы, рассматриваемые при принятии того или иного решения. В модульном представлении такими альтернативами являются различные варианты параметризации модулей. В представлении «компонент и соединитель» могут быть отражены ограничения по дублированию, планированию или выбору протоколов. В представлении распределения это условия назначения конкретного программного элемента тому или иному процессору;
   - ◊ время связывания альтернатив. Одни решения принимаются в период проектирования, другие — в период производства, третьи — в период исполнения.
5. *Предпосылки архитектурного решения* (architecture background) — это раздел, в котором содержится обоснование отраженного в данном представлении проектного решения. Цель его — объяснить читателю, почему проект выглядит так, как он выглядит, и представить убедительные аргументы его превосходства. В состав этого раздела входят следующие элементы:
   - ◊ логическое обоснование, демонстрирующее предпосылки принятия проектных решений, отраженных в данном представлении, и причины отказа от альтернатив;
   - ◊ результаты анализа, оправдывающие проектное решение и объясняющие последствия модификаций;
   - ◊ отраженные в проектном решении допущения.
6. *Глоссарий терминов* (glossary of terms), применяемых в представлениях, и краткое описание каждого из них.

7. *Другая информация.* Содержание этого раздела зависит от методов работы конкретной организации. В частности, здесь можно разместить административные сведения: авторство, данные управления конфигурациями и история изменений. Кроме того, архитектор волен разместить здесь систему ссылок на отдельные разделы сводки требований. Таким образом, речь идет об информации, которая, строго говоря, не имеет прямого отношения к архитектуре, но которую удобнее всего привести вместе с архитектурными сведениями. Для этого и предназначен рассматриваемый раздел. В любом случае, в его вступительной части необходимо поместить содержание.

Вышеописанные элементы документации изображены в виде схемы на рис. 9.1.



Источник: [Clements 03] (адаптированная версия).

Рис. 9.1. Семь элементов документированного представления

Документирование поведения

Представления содержат структурную информацию о системе. Но для рассуждений о некоторых свойствах системы ее недостаточно. К примеру, для того чтобы уяснить вопросы взаимоблокировки, необходимо знать последовательность взаимодействий между элементами, которую структурная информация не отражает. Эти сведения, равно как возможности параллелизма и временные зависимости, характерные для взаимодействий (которые происходят в установленные моменты или по истечении установленных временных периодов), раскрывают поведенческие описания. Поведение документируется применительно к отдельному элементу или к множеству взаимодействующих элементов. Выбор в вопросах моделирования зависит от типа проектируемой системы. К примеру, если речь идет о встроенной системе реального времени, на первое место выходят свойства времени и время наступления событий. В системе банковского обслуживания значительно важнее фактического времени наступления событий оказывается их последовательность (например, последовательность элементарных транзакций и процедур отката). В зависимости от вида предполагаемых аналитических действий уместно обращаться к разным методикам моделирования и нотациям. Примерами поведенческих описаний в языке UML являются диаграммы последовательностей и схемы состояний. Подобные нотации весьма широко распространены.

Схемы состояний как формализм появились в 1980-х годах и изначально предназначались для описания реактивных систем. Ряд реализованных в них полезных расширений дополняет традиционные диаграммы состояний (такие, как вложенность состояний и состояния «и») и придает выразительность абстракции и параллелизму моделей. Схемы состояний позволяют рассуждать о системе в целом. Предполагается отображение всех ее состояний, а методики анализа в отношении системы приобретают универсальный характер. Становятся возможными ответы на вопросы типа: «Всегда ли время отклика на данный стимул будет меньше 0,5 секунды?»

Диаграмма последовательностей помогает документировать последовательность обменов стимулами. Она отражает кооперацию применительно к экземплярам компонентов и их взаимодействию, причем последнее представляется во временной последовательности. Вертикальное измерение при этом выражает время, а горизонтальное — различные компоненты. Диаграммы последовательностей позволяют строить умозаключения на основе конкретных сценариев использования. Они демонстрируют механизм реагирования системы на отдельные стимулы, иллюстрируют выбор путей в системе и отвечают на вопросы типа: «Какие параллельные операции проходят в момент реагирования системы на определенные стимулы в определенных условиях?»

Документирование интерфейсов

Интерфейс (interface) — это граница, на которой встречаются, взаимодействуют между собой или передают друг другу информацию две независимые сущности.

Согласно приведенному в главе 2 определению программной архитектуры, очевидно, что интерфейсы элементов — носителей их внешне видимых другим элементам свойств — являются понятием архитектурным. Поскольку без них невозможны ни анализ, ни проектирование систем, документировать интерфейсы совершенно необходимо.

Под документированием интерфейса подразумевается указание его имени, идентификация, а также отражение всех синтаксических и семантических сведений о нем. Первые два элемента — указание имени и идентификация — обобщенно называются «сигнатурой» интерфейса. Если в качестве ресурсов интерфейса выступают вызываемые программы, сигнатура обозначает их и определяет их параметры. Параметры определяются по порядку, типу данных и (иногда) по принципу возможности изменения их значений программой. Та информация о программе, которая содержится в сигнатуре, обычно приводится в заголовочных файлах C или C++ и в интерфейсах Java.

При всей полезности сигнатур (в частности, они делают возможной автоматическую проверку конструкций) ими все не исчерпывается. Соответствие сигнатур обеспечивает успешную компиляцию и/или компоновку системы, но совершенно не гарантирует достижение конечной цели — ее нормальное функционирование. Необходимая для этого информация относится к семантике интерфейса, которая сообщает, что происходит при активизации ресурсов.

Интерфейс документируется в форме спецификации — изложения свойств элемента, которые архитектор желает предать огласке. Это должна быть только та информация, которая необходима для организации взаимодействия с интерфейсом. Другими словами, архитектор должен решить, во-первых, какую информацию об элементе допустимо и уместно сообщить читателю и, во-вторых, какая информация, вероятнее всего, не будет подвержена изменениям. В процессе документирования интерфейса важно, с одной стороны, не раскрыть слишком много сведений, и с другой — не утаить необходимые данные. Разработчики не смогут наладить успешное взаимодействие с элементом, если о нем будет недостаточно информации. Избыток информации, в свою очередь, усложняет будущие изменения в системе и усиливает их влияние, делает интерфейс неудобным для восприятия. Для того чтобы достойно справиться с этой ситуацией, наибольшее внимание следует уделять не реализации элементов, а их взаимодействию с рабочими средами. Документированию подлежат только внешне видимые явления.

Элементы, присутствующие в виде модулей, часто напрямую соответствуют одному или нескольким элементам представления «компонент и соединитель». Как правило, интерфейсы элементов в представлении модулей и представлении «компонент и соединитель» схожи или идентичны и документировать их в обоих этих представлениях излишне. Поэтому спецификацию интерфейса следует привести в модульном представлении, а в «компоненте и соединителе» поместить ссылку на нее и изложить индивидуальную для данного представления информацию. Кроме того, один модуль может оказаться общим для нескольких модульных представлений — например, декомпозиции на модули и использования. В таком случае, как и в предыдущем, спецификацию интерфейса следует привести в одном из этих представлений, а во всех остальных поставить соответствующую ссылку.

Шаблон для документирования интерфейсов

Ниже изложен один из вариантов стандартной структуры документации интерфейса. Некоторые ее элементы, если они оказываются бесполезными в конкретном применении, можно выкинуть, другие, наоборот, ввести. Значительно более важными, чем сама стандартная структура, представляются нам способы ее применения. Ваша цель должна заключаться в том, чтобы в точности изложить все внешне видимые взаимодействия интерфейсов, предусмотренных в проекте.

1. *Индивидуальность интерфейса.* Если у элемента несколько интерфейсов, их нужно как-то отличать друг от друга. Как правило, для этого интерфейсам присваиваются имена, в некоторых случаях сопровождающиеся указанием номера версии.
2. *Предоставляемые ресурсы.* Основным предметом документирования любого интерфейса должны быть предоставляемые им ресурсы. Они определяются синтаксисом, семантикой (описывающей следствия их применения) и ограничениями по использованию. Существует ряд нотаций, предназначенных для документирования синтаксиса интерфейса. Одна из них — язык описания интерфейсов (interface definition language, IDL), разработанный рабочей группой OMG, — применяется в сообществе CORBA. С помощью специальных языковых конструкций этого языка описываются типы данных, операции, атрибуты и исключения. Семантическую информацию можно выразить только в комментариях. В большинстве программных языков есть встроенные средства спецификации сигнатур элементов, примером чему — заголовочные файлы (.h) в C и спецификации пакетов в Ada. Наконец, на языке UML синтаксическая информация об интерфейсе выражается через стереотип **<<interface>>** (см. рис. 9.4). Для интерфейса необходимо хотя бы ввести имя; кроме того, архитектор волен составить для него сигнатуру.
   - ◊ *Синтаксис ресурса.* Здесь указывается сигнатура ресурса. Содержащейся в сигнатуре информации должно быть достаточно для того, чтобы любая сторонняя программа смогла корректно с точки зрения синтаксиса обратиться к программе, использующей данный ресурс. Таким образом, в сигнатуру входят имя ресурса, имена и логические типы данных его аргументов (если таковые имеются) и т. д.
   - ◊ *Семантика ресурса.* Здесь приводится описание результатов вызова ресурса, в частности:
     1) присваивание значений данным, к которым может обращаться актер, вызывающий рассматриваемый ресурс, — от простого задания значения возвращаемого аргумента до обновления центральной базы данных;
     2) события и сообщения, сигнализируемые/отправляемые в результате обращения к ресурсу;
     3) предполагаемое поведение других ресурсов как результат обращения к рассматриваемому ресурсу (к примеру, если данный ресурс уничтожит некий объект, то последующие попытки обращения к этому объекту будут приводить к новому результату — ошибке);

4) результаты, наблюдаемые пользователем (встречающиеся в основном во встроенных системах: скажем, вызов программы, ответственной за включение бортового индикатора, приводит к наблюдаемому результату).

Кроме того, семантика должна прояснять вопросы, связанные с исполнением ресурса: будет ли оно носить элементарный характер, можно ли его приостановить или прервать. Как правило, семантическая информация выражается средствами естественного языка. Для фиксации предусловий и постусловий — сравнительно простого и эффективного метода выражения семантики — специалисты во многих случаях прибегают к булевой алгебре. Еще одним средством передачи семантической информации являются следы — записи последовательностей операций и взаимодействий, описывающих реакцию элемента на тот или иной вариант использования.

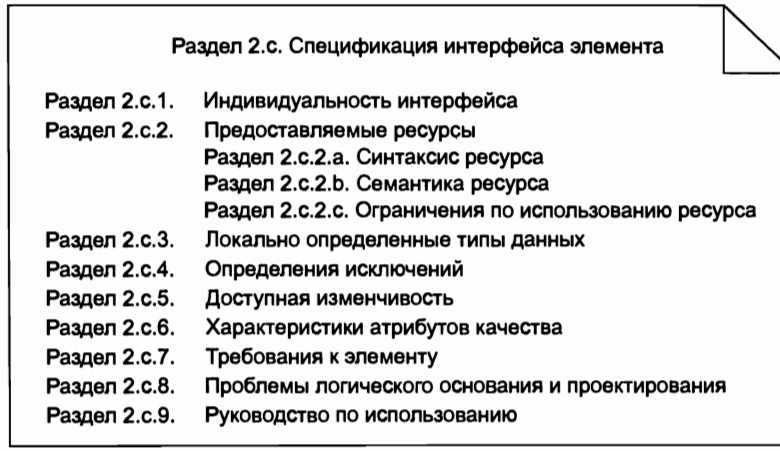
◊ *Ограничения на использование ресурсов.* В каких обстоятельствах возможно обращение к рассматриваемому ресурсу? Существует ли необходимость в предваряющей его считывание инициализации данных? Обусловливается ли вызов одного метода вызовом другого? Не установлены ли какие-то ограничения относительно количества актеров, имеющих возможность одномоментного взаимодействия с ресурсом? Возможно, в силу принадлежности элемента определенному актеру, он единственный, кто обладает полномочиями по модификации элемента, а все остальные актеры ограничиваются лишь считыванием. Не исключено также, что, согласно многоуровневой схеме безопасности, каждый актер может обращаться к строго определенным ресурсам или интерфейсам. Если рассматриваемый ресурс отталкивается от каких-либо допущений о своем окружении (например, требует наличия других ресурсов), их следует задокументировать.

3. *Определения типов данных.* Если какой-либо ресурс интерфейса задействует тип данных, отличный от предусмотренного соответствующим языком программирования, архитектор должен привести определение типа данных. Если он определен в другом элементе, достаточно установить ссылку на его документацию. Как бы то ни было, программисты, которые пишут элементы, обращаясь к такому ресурсу, должны знать: а) как объявлять переменные и константы типа данных; б) как в рамках этого типа данных записывать литеральные значения; в) какие операции и сравнения применимы к членам типа данных; и г) как преобразовывать значения этого типа данных в данные других типов.
4. *Определения исключений.* Здесь предполагаются описания исключений, которые могут порождать ресурсы на данном интерфейсе. Поскольку одни и те же исключения во многих случаях характерны для множества ресурсов, имеет смысл отделять список исключений каждого ресурса от их определений, причем последние размещать в специальном словаре. Настоящий раздел как раз исполняет роль словаря. Здесь же допустимо определять стандартное поведение обработки ошибок.

5. *Характерная для интерфейса изменчивость.* Предполагает ли данный интерфейс возможность конфигурирования элемента? Подобные *параметры конфигурации* (configuration parameters), а также их воздействие на семантику интерфейса подлежат документированию. В качестве примеров изменчивости можно привести емкость видимых структур данных и рабочие характеристики соответствующих алгоритмов. Для каждого параметра конфигурации архитектор должен зафиксировать имя, диапазон значений и время связывания его фактических значений.
6. *Характеристики атрибутов качества интерфейса.* Архитектор должен задокументировать характеристики атрибутов качества (например, производительность или надежность), предоставляемых интерфейсом конечным пользователям. Эти сведения можно выразить в форме ограничений на реализацию составляющих интерфейс элементов. Выбор атрибутов качества, на которых предполагается сконцентрироваться и принять обязательства, проводится в зависимости от контекста.
7. *Требования к элементам.* Среди требований элемента может значиться наличие конкретных именованных ресурсов других элементов. Документация в данном случае составляется так же, как и в отношении ресурсов, — указываются синтаксис, семантика и ограничения по использованию. В некоторых случаях подобного рода информацию удобнее документировать в форме ряда допущений о системе, принятых проектировщиком элемента. В таком случае требования можно представить на суд экспертов, которые уже на ранних стадиях процесса проектирования смогут подтвердить или опровергнуть принятые допущения.
8. *Логическое обоснование и соображения о проекте.* Как и в случае с логическим обоснованием архитектуры в целом (или архитектурных представлений), архитектор должен документировать факторы, обусловившие принятие тех или иных проектных решений относительно интерфейса. В логическом обосновании необходимо указать мотивацию принятия этих решений, ограничения и компромиссы, обрисовать рассмотренные, но впоследствии забракованные решения (не забыв попутно изложить причины отказа от них), и изложить соображения архитектора касательно дальнейших изменений интерфейса.
9. *Руководство по использованию.* Пункты 2 и 7 ориентированы на поресурсное изложение семантической информации об элементе. Иногда такой подход не оправдывает себя. В некоторых случаях требуется анализ семантики с позиции связей между множеством отдельных взаимодействий. Если это так, в дело вступает *протокол* (protocol), документировать который следует согласно последовательности взаимодействий. Протоколы отражают комплексное поведение при взаимодействии тех образцов использования, которые, по мысли архитектора, должны встречаться с определенной регулярностью. Сложность взаимодействия с элементом через его интерфейс следует компенсировать статической поведенческой моделью (например, схемой состояний) или примерами проведения отдельных видов взаимодействия (в форме диаграмм последовательностей). Механизм документирования

в данном случае аналогичен показанным в предыдущем разделе вариантам поведения уровня представления, с тем лишь различием, что руководство по использованию ориентируется на отдельный элемент.

Общая схема описанного шаблона приведена на рис. 9.2; фактически, это лишь более подробный вариант раздела 2.c на рис. 9.1.



Источник: [Clements 03] (адаптированная версия).

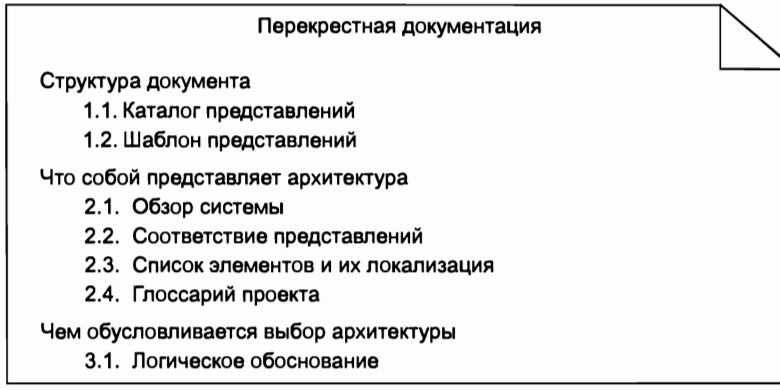
Рис. 9.2. Девять элементов документации интерфейса

9.5. Перекрестная документация

В этом разделе мы намерены рассмотреть приложение к документации представлений — раздел, в котором приводится информация, распространяющаяся на несколько представлений или на весь комплект документации. В рамках перекрестной документации выделяются три основных аспекта, которые мы обобщенно называем «как–что–почему»:

1. *Как* достигается такая структура документации, в рамках которой любое заинтересованное в архитектуре лицо получает возможность эффективного и надежного поиска необходимой информации? В этой части содержатся каталог и шаблон представлений.
2. *Что* представляет собой архитектура? Здесь информация, не вписывающаяся в рамки отдельных представлений, излагается в виде краткого обзора системы, призванного ориентировать читателя относительно поставленных перед ней задач; кроме того, здесь приводится объяснение взаимосвязи представлений, перечень элементов с указанием их локализации, а также общий для всей архитектуры глоссарий.
3. *Почему* архитектура стала такой, какая она есть? Подразумевается контекст системы, внешние ограничения, тем или иным образом повлиявшие на черты архитектуры, а также логическое обоснование принятия глобальных решений общего характера.

Все эти элементы изображены на рис. 9.3.



Источник: [Clements 03] (адаптированная версия).

Рис. 9.3. Схема перекрестной документации

Как документация адаптируется к задачам заинтересованных лиц

В каждом комплекте архитектурной документации необходимо предусматривать введение. Перед ним ставятся две задачи: во-первых, объяснить не знакомым с архитектурой заинтересованным лицам структуру документации, а во-вторых, помочь им найти необходимые сведения. Информация, отвечающая на вопрос «как?», выражается в двух формах:

- каталог представлений;
- шаблон представлений.

Каталог представлений

Каталог представлений должен знакомить читателя с содержанием представлений, включенных в комплект документации согласно решению архитектора.

Поскольку комплект документации, помимо прочего, является одним из средств коммуникации, всем новым читателям следует объяснять механизм поиска содержащихся в нем сведений. Именно для этого и предназначен каталог. Если комплект документации применяется как основа для проведения анализа, у аналитиков должна быть возможность выбрать нужные представления. К примеру, для анализа производительности требуются сведения о потреблении ресурсов. С помощью каталога аналитик может сориентироваться в представлениях, содержащих соответствующие данные.

Каждому представлению, отраженному в комплекте документации, в каталоге представлений соответствует отдельная запись. Любая запись состоит из нескольких элементов:

1) имя представления и обозначение стиля, который оно иллюстрирует;
2) описание типов элементов, отношений и свойств представления;

3) описание назначения представления;
4) управляющая информация о документации представления: указание его последней версии, местоположения и владельца.

Каталог представлений предназначен для описания комплекта документации; соответственно, он не имеет отношения к описанию системы. Характеристики системы должны быть показаны не здесь, а в отдельных представлениях. Каталог же, помимо прочего, содержит сведения об элементах, фактически отраженных в тех или иных представлениях.

Шаблон представления

Шаблон представления являет собой его стандартную структуру. Сведения, приведенные на рис. 9.1, наравне с сопутствующим материалом составляют основу шаблона представления — они определяют стандартные элементы документации представления, их содержимое и принятые в них правила. По своему назначению шаблон представления аналогичен любой другой стандартной структуре: читателю он помогает оперативно добраться до нужного раздела, а составителю — систематизировать информацию и установить критерии выявления задач, которые еще только предстоит выполнить.

Что такое архитектура

В этом разделе содержится информация о системе, которая строится на основе документируемой архитектуры, раскрываются взаимоотношения представлений и размещается указатель архитектурных элементов.

Обзор системы

Под обзором системы имеется в виду краткое описание ее функций и пользователей, с упоминанием наиболее важных дополнительных сведений или ограничений. Цель — помочь читателю составить стойкую умозрительную модель системы и ее задач. В отдельных случаях обзор системы составляется в масштабе всего проекта, а в рассматриваемом разделе документации помещается ссылка на него.

Соответствие между представлениями

Поскольку все представления архитектуры описывают одну и ту же систему, само собой разумеется, что каждый из них в чем-то похож на другие. Усвоение читателем взаимосвязей между представлениями способствует формированию комплексного восприятия архитектуры. Чем более очевидно эти связи выражаются в форме соответствий между представлениями, тем выше уровень владения принципами функционирования системы и тем меньше вопросов остается без ответов.

К примеру, возможно соответствие между отдельным модулем и множеством элементов периода прогона — именно так классы отображаются на объекты. Сложности возникают с неоднозначными соответствиями, а также в тех случаях, когда элементы периода прогона не представлены в виде элементов кода, — например, если они импортируются в период прогона или встраиваются в период

построения или загрузки. Впрочем, существует множество относительно простых соответствий типа «один ко многим» (или «ни одного ко многим»). Но все же более других распространены случаи соответствия отдельных *частей* элементов в одном представлении отдельным *частям* элементов в другом представлении.

Приводить соответствия между всеми парами представлений совершенно не обязательно — достаточно выбрать наиболее репрезентативные.

Список элементов

Списком элементов называется указатель всех без исключения элементов, вне зависимости от того, в каком представлении они встречаются. Сопровождаемые ссылками, эти записи помогают заинтересованным лицам не тратить на поиск необходимых элементов слишком много времени.

Глоссарий проекта

В глоссарии содержатся определения уникальных для системы терминов со специальным значением. Не помешает заинтересованным лицам и список сокращений с расшифровкой. В случае, если аналогичный глоссарий уже существует, на него достаточно лишь сослаться.

Почему архитектура стала именно такой, какой стала

Аналогичное по своему назначению логическим обоснованиям представления и проектного решения интерфейса, перекрестное логическое обоснование доказывает целесообразность архитектуры в целом (в контексте удовлетворения предъявляемых к ней требований). В частности, оно проясняет следующие моменты:

- последствия принятия общесистемных проектных решений относительно удовлетворения требований или обеспечения соответствия ограничениям;
- реакция архитектуры на введение нового, но спрогнозированного требования или корректировку существующего;
- ограничения, которые должен учитывать разработчик при реализации решения;
- отклоненные альтернативные решения.

Таким образом, на материале перекрестного логического обоснования можно сделать вывод о причинах, побудивших принять то или иное решение, и последствиях его пересмотра.

9.6. Унифицированный язык моделирования

До настоящего момента мы в основном обсуждали виды информации, включаемые в документацию архитектуры. Архитектура — это, в некотором смысле, выразитель важнейших знаний о программной системе, не зависящих от способов

фиксации, коими являются языки и нотации. Вместе с тем неофициальный стандарт архитектурной нотации сегодня существует — это унифицированный язык моделирования (Unified Modeling Language, UML). В первую очередь он применяется при составлении первичного отображения представления и лишь во вторую — выражает поведение элемента или группы элементов. Ответственность за дополнение диаграмм UML вспомогательной документации (каталогом элементов, логическим обоснованием и т. д.) полностью лежит на архитекторе. UML не предусматривает прямой поддержки компонентов, соединителей, уровней, семантики интерфейсов и многих других очевидно архитектурных аспектов системы.

Тем не менее в большинстве случаев конструкции языка UML оказываются достаточными для достижения удовлетворительных результатов — по крайней мере, в том, что касается первичных отображений архитектурных представлений. Итак, сначала мы рассмотрим модульные представления.

Модульные представления

Как вы помните, модулем называется код или блок реализации соответственно, модульное представление — это перечень всех модулей, их интерфейсов и отношений.

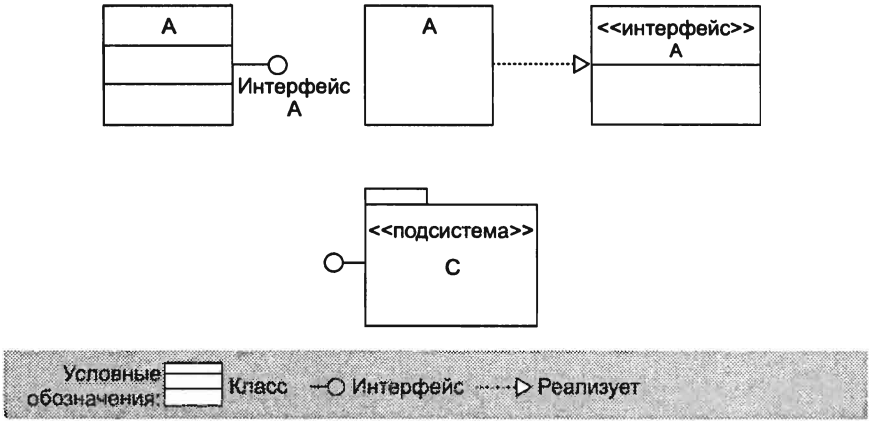


Рис. 9.4. Интерфейсы в UML

Интерфейсы

Рисунок 9.4 демонстрирует отображение интерфейсов модулей средствами UML. Интерфейсы на диаграммах UML обозначаются в виде «кружков на палочке». Присоединять их можно, помимо прочего, к классам и подсистемам.

Кроме того, UML позволяет стереотипировать в качестве интерфейсов символы классов (прямоугольники); пунктирная стрелка с треугольным наконечником проводится из элемента, реализующего данный интерфейс (на который указывает стрелка). В нижней части символа класса можно размещать сигнатуру интерфейса: имена его методов, аргументы, типы аргументов и т. д. Нотация «кружок-палочка» наилучшим образом показывает зависимости, связывающие элементы

и интерфейс. Прямоугольная нотация, в свою очередь, предусматривает более подробное описание синтаксиса интерфейса — в частности, предоставляемых им операций.

Модули

Конструкции, предназначенные для отображения разного рода модулей, представлены в UML весьма обширно. Некоторые примеры таковых приводятся на рис. 9.5. Конструкция класса в UML соответствует объектно-ориентированной специализации модуля. В случаях, когда значительную роль играет группировка функциональности — например, при отображении уровней и классов, — удобнее обращаться к пакетам. Конструкция подсистемы применяется при необходимости специфицирования интерфейса и поведения.

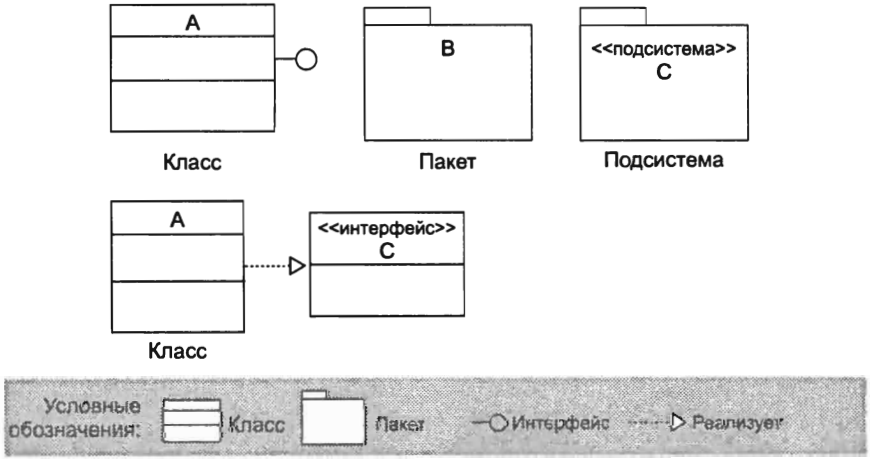


Рис. 9.5. Примеры нотаций модулей в UML

На рис. 9.6 приводится пример обозначения (средствами UML) отношений, присущих модульным представлениям. Декомпозиция на модули базируется на отношении «является частью». Для представления использования модулей применяют отношение зависимости, а, скажем, для представления классов модуля — обобщение или отношение «является» (оно же — «наследование»).

Агрегация

С помощью конструкции подсистемы в UML можно отображать модули, содержащие другие модули; прямоугольник класса, как правило, применяется в листьях декомпозиции. Подсистемы отображаются двояко — как пакеты и как классификаторы. В первом случае существует возможность декомпозиции подсистем, а значит, и агрегации (так здесь называют группировку) модулей. Подсистемы, изображаемые в качестве классификаторов, инкапсулируют свое содержимое и предоставляют явный интерфейс. UML предусматривает три способа обозначения агрегации:

- модули могут быть вложенными (рис. 9.7, слева);

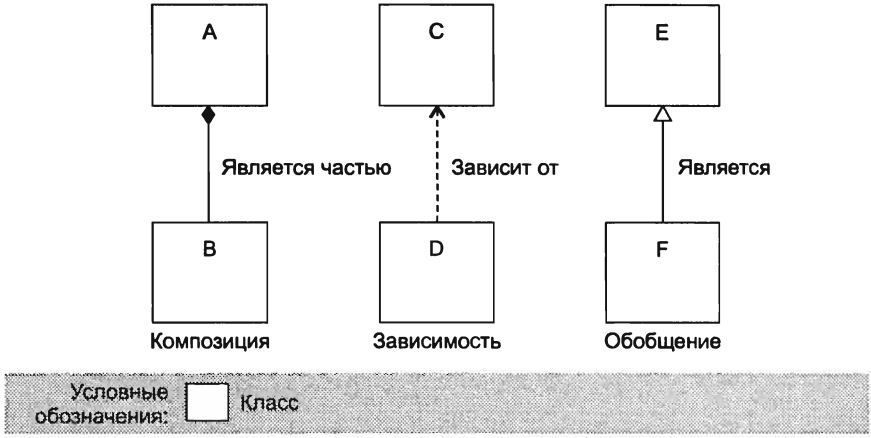


Рис. 9.6. Примеры нотаций отношений в UML. Модуль B здесь является частью модуля A, модуль D зависит от модуля C, а модуль F представляет собой разновидность модуля E

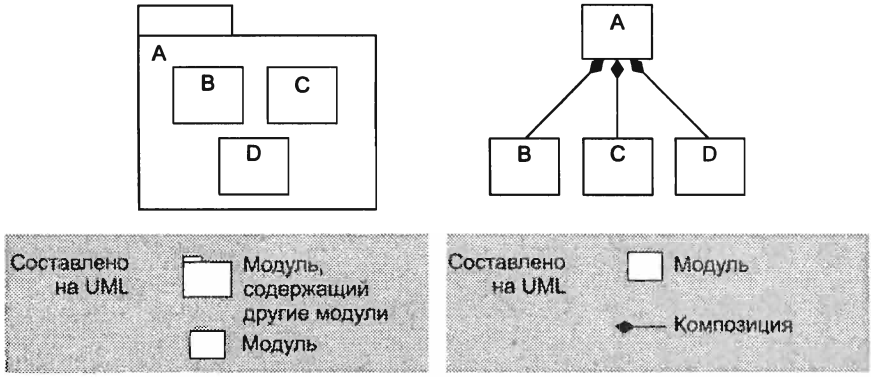


Рис. 9.7. Декомпозиция с вложением в UML. Модуль-агрегат, изображенный в виде пакета (слева); декомпозиция в UML изображается с помощью ребер (справа)

- можно изобразить последовательность двух (в том числе связанных) диаграмм, вторая из которых обозначает содержимое модуля, показанного в первой;
- ребро, обозначающее композицию (ромбик примыкает к контуру агрегата), соединяет родительский и дочерний модули (рис. 9.7, справа).

Композиция в UML понимается как разновидность агрегации с подразумеваемой жесткой принадлежностью — другими словами, ее части рождаются и умирают вместе с целым (агрегатом). Если, к примеру, модуль A состоит из модулей B и C, то последние не могут существовать без первого; соответственно, в случае уничтожения A в период прогона исчезают B и C. Таким образом, отношение композиции в UML выходит за рамки структурирования блоков реализации; оно также затрагивает элементы, принадлежащие к периоду прогона. Прежде чем принимать решение о введении отношения композиции UML, архитектор должен гарантировать соответствие этой принадлежности.

Обобщение

Поскольку модули в UML изображаются в виде классов (хотя в некоторых случаях — в виде подсистем), обобщение относится к числу основных приоритетов этого языка. Базовая нотация UML представлена на рис. 9.8.

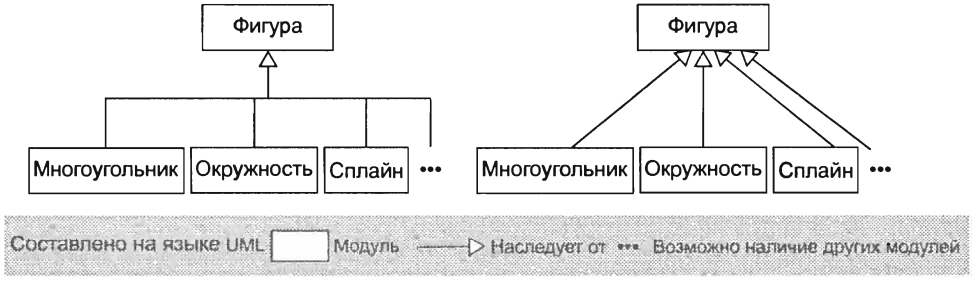


Рис. 9.8. Два стиля линий при документировании обобщения в UML

Показанные на рис. 9.8 диаграммы семантически идентичны. Троеточие (...) в UML заменяет подмодуль; этот элемент подразумевает возможность и вероятность наличия у данного модуля опущенных на диаграмме дочерних модулей. Модуль Фигура по отношению к Многоугольнику, Окружности и Сплайну (каждый из которых является его подклассом, дочерним модулем или потомком) является родительским модулем. По сравнению со своими специализированными версиями, представленными в виде дочерних модулей, модуль Фигура носит более общий характер.

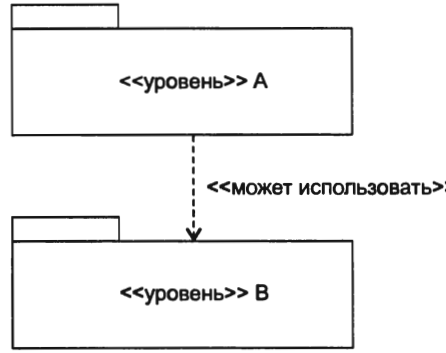
Зависимость

Базовая нотация зависимости изображена на рис. 9.6. Наиболее значимое с точки зрения архитектуры проявление зависимости встречается в уровнях. Как это ни прискорбно, встроенных примитивов, соответствующих уровню, в UML не существует. Впрочем, простые уровни можно изображать при помощи *пакетов* (packages; см. рис. 9.9). Пакеты — это универсальный механизм систематизации элементов по группам. В UML предусмотрены пакеты для систем и подсистем. Чтобы приспособить для представления уровней дополнительный пакет, его следует определить как пакет со стереотипом. Изображение уровней в виде пакетов UML, впрочем, сопряжено с двумя ограничениями: во-первых, пакет группирует модули, а во-вторых, между пакетами устанавливается отношение зависимости «может использовать» (allowed to use). Обратившись к нотации пакетов, уровень можно обозначить именем-стереотипом <<layer>>, поместить которое следует перед именем самого уровня; в качестве альтернативы можно ввести новое визуальное обозначение — например, затененный прямоугольник.

Представления из группы «компонент и соединитель»

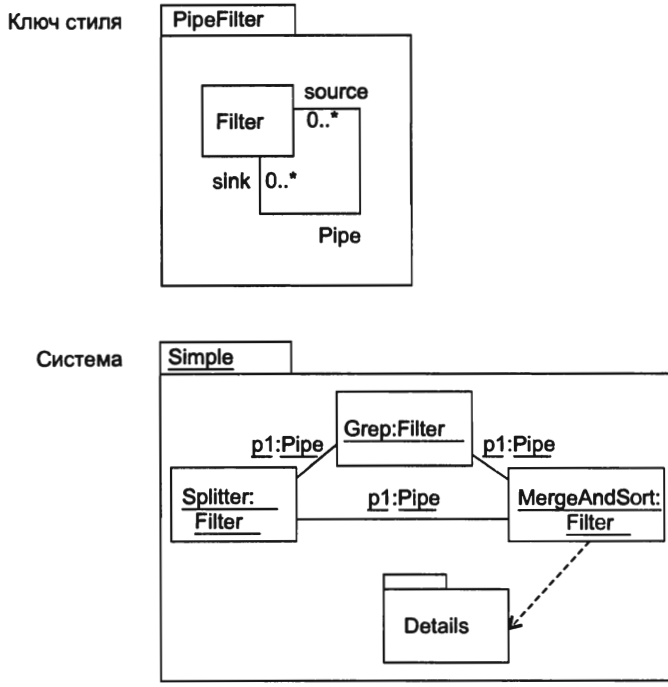
Однозначно выигрышной стратегии документирования средствами UML представлений из группы «компонент и соединитель» не существует — есть лишь ряд

возможных альтернатив. У каждой из них свои преимущества и недостатки. Перечень вариантов отображения типов «компонент и соединитель» мы начнем с понятия класса UML.



Составлено на языке UML

Рис. 9.9. Простейший способ представления уровней в UML



Составлено на языке UML

Рис. 9.10. Типы как классы и экземпляры как объекты в простой системе «каналы и фильтры»

Общий принцип действия системы, построенной по образцу «каналы и фильтры», изображен на рис. 9.10. Архитектурный тип «фильтр» здесь представлен в виде класса UML Filter. Экземпляры фильтров — в частности, Splitter — представлены на диаграмме объектов в виде соответствующих объектов. В целях определения границы пространства имен мы заключили описания в пакеты. Отображение MergeAndSort, обозначенное как Details, должно быть показано в каком-либо другом пакете.

Рассмотрим эту стратегию поподробнее.

Компоненты

Отношение «тип–экземпляр» в архитектурных описаниях близко соответствует отношению «класс–объект» в модели UML. Подобно типам компонентов в архитектурных описаниях, классы UML — это сущности первого класса, обогащенные структуры, предназначенные для фиксации программных абстракций. Для описания структуры, свойств и поведения класса пригодны все описательные механизмы UML, что обеспечивает возможность серьезной детализации и применения инструментов анализа на основе UML. Свойства архитектурных компонентов можно преподнести в виде свойств класса или посредством ассоциаций; поведение описывается при помощи поведенческих моделей UML; представлять отношения между типами компонентов удобно при помощи компонентов. Эффект уточнения семантики экземпляра или типа достигается путем добавления к нему одного из стандартных стереотипов; к примеру, для того чтобы указать на исполнение компонента в качества отдельного процесса, к компоненту можно добавить стереотип «process». Следует также иметь в виду, что отношение между MergeAndSort и его субструктурой обозначается отношением зависимости.

Интерфейсы

Способов изображения интерфейсов для компонентов (их иногда называют портами) всего пять (рис. 9.11). Их описания ниже приводятся в последовательности, соответствующей повышению выразительности. Впрочем, с увеличением выразительности повышается сложность; таким образом, остановиться следует на первой же стратегии, отвечающей сформулированным вами требованиям.

- *Вариант 1: Без явного отображения.* Отказ от изображения интерфейсов, с одной стороны, значительно упрощает диаграммы, но с другой — лишает возможности обозначить в рамках первичного отображения их (интерфейсов) имена и свойства. Таким образом, означенное решение следует считать приемлемым лишь в трех случаях: 1) если на каждый компонент приходится по одному интерфейсу; 2) если интерфейсы можно вывести из топологии системы; 3) если уточненную диаграмму предполагается разместить где-то в другом месте.
- *Вариант 2: Интерфейсы как аннотации.* Отображение интерфейсов в виде аннотаций решает задачу размещения связанной с ними информации, но, с другой стороны, в силу отсутствия у аннотаций UML семантического значения их применение в качестве основы для проведения анализа становится невозможным. Такое решение, опять же, допустимо лишь при том

условии, что описание детальных свойств интерфейса не является приоритетной задачей.

- *Вариант 3: Интерфейсы в виде атрибутов классов/объектов.* Рассматривая интерфейсы как атрибуты класса/объекта, мы инкорпорируем их в формальную структурную модель, однако в таких условиях их отображение на диаграмме классов ограничивается указанием имени и типа. Указанное ограничение снижает выразительность данного варианта.
- *Вариант 4: Интерфейсы как интерфейсы UML.* Нотация «кружок-палочка» в UML предусматривает возможность компактного описания интерфейса в рамках диаграммы классов, изображающей тип компонента. Что касается диаграммы экземпляров, то здесь ассоциативная роль UML, соответствующая экземпляру интерфейса и квалифицируемая именем типа интерфейса, обеспечивает компактность выражения взаимодействия экземпляра компонента через тот или иной экземпляр интерфейса. Такой подход предполагает визуальные различия между компонентами и интерфейсами, благодаря которым очевидной становится подчиненная роль интерфейсов.

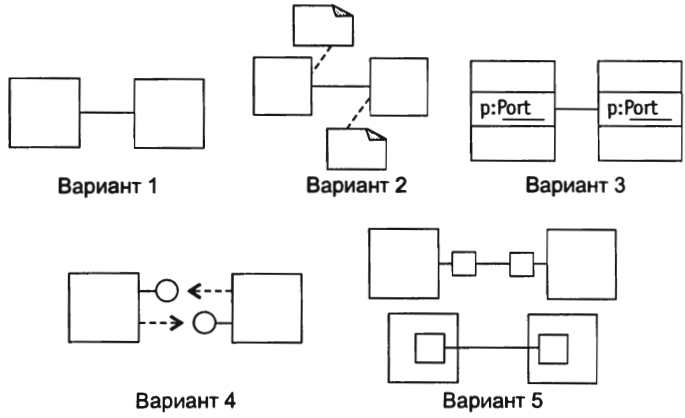


Рис. 9.11. Пять способов отображения интерфейсов компонентов (портов)

С другой стороны, эта стратегия не предусматривает изображения служб, требуемых от среды компонента, хотя таковые во многих случаях оказываются в качестве основных элементов интерфейса. Кроме того, у типа компонента может быть несколько экземпляров одного и того же типа интерфейса, однако реализация одним классом нескольких версий одного интерфейса UML бессмысленна. К примеру, в рамках этой методики крайне сложно определить тип-фильтр Splitter с двумя выходными портами одного и того же типа. Наконец, в отличие от классов, интерфейсы UML не имеют ни атрибутов, ни субструктуры.

- *Вариант 5: Интерфейсы как классы.* Описывая интерфейсы как классы в составе типа компонента, мы обеспечиваем выразительность, не свой-

ственную всем предыдущим альтернативам. Появляется возможность отображения субструктуры интерфейса и обозначения наличествующих у типа компонента нескольких однотипных интерфейсов. Экземпляр компонента моделируется как объект, содержащий ряд интерфейсных объектов. С другой стороны, отображая интерфейсы в виде классов, мы не только «засоряем» диаграмму, но и лишаем читателя возможности проводить зрительное различие между интерфейсами и компонентами. В нижней части варианта 5 на рис. 9.11 используется одна из разновидностей рассматриваемой нотации, согласно которой интерфейсы изображаются в виде вложенных классов. Впрочем, не следует забывать, что обозначение точек взаимодействия затрудняет восприятие, — связано это с тем, что вложенность обычно предполагает принадлежность одних классов другому классу; при этом вопрос о доступности экземпляров дочерних классов через экземпляры класса родительского не раскрывается.

Соединители

Вариантов отображения соединителей всего три. Решение о выборе одного из них, как и в предыдущем случае, следует принимать исходя из соотношения выразительности и семантического соответствия, с одной стороны, и сложности — с другой.

- *Вариант 1: типы соединителя как ассоциации, экземпляры соединителей как связи.* В рамках архитектурной блочно-линейной диаграммы системы линиями между компонентами изображаются соединители. Само собой напрашивается решение об изображении соединителей в виде ассоциаций между классами или связей между объектами. Этот метод отличается визуальным удобством, четким разграничением компонентов и соединителей, а также применением наиболее употребительного на диаграммах классов UML отношения — ассоциации. Кроме того, ассоциации в данном случае можно помечать, а направление соединителя обозначать стрелкой. К сожалению, у соединителей и ассоциаций разные значения. В рамках архитектурного описания система конструируется путем выбора компонентов, поведение которых раскрывается через интерфейсы, и их подключения к соединителям, ответственным за координацию вариантов их поведения. Поведением системы считается коллективное поведение набора компонентов, взаимодействие между которыми определяется и ограничивается установленными между ними соединениями.

  Несмотря на то что ассоциация, или связь, в UML обладает определенным потенциалом взаимодействия между соотносимыми элементами, механизм ассоциации в первую очередь ориентирован на описание концептуального отношения между двумя данными элементами. Кроме того, поскольку ассоциация представляет собой отношение между элементами UML, ее самостоятельность в рамках модели UML исключается. Следовательно, изолированное отображение типа соединителя невозможно. Выбор, таким образом, сужается до соглашений о присваивании имен и стереотипов, значения которых фиксируются описаниями объектного языка ограничений

UML. Вдобавок ко всему, рассматриваемая методика не позволяет специфицировать интерфейсы соединителя.

- *Вариант 2: типы соединителя как классы-ассоциации.* Один из способов компенсировать недостаток выразительности — квалифицировать ассоциацию классом – представителем типа соединителя. В этом случае тип и атрибуты соединителя можно зафиксировать как атрибуты класса или объекта. К сожалению, эта методика не предусматривает явного отображения интерфейсов соединителя.
- *Вариант 3: типы соединителя как классы, экземпляры соединителя как объекты.* Для того чтобы средствами UML придать соединителям первоклассный статус, типы соединителя следует отобразить как классы, а экземпляры соединителей — как объекты. Классы и объекты предусматривают те же четыре варианта отображения ролей, что характерны для интерфейсов: отказ от отображения, отображение в виде аннотаций, в виде интерфейсов, реализуемых классами, или в виде дочерних классов, вложенных в класс соединителя. Прикрепление интерфейсов компонента и соединителя в рамках схемы отображения интерфейсов можно выразить как ассоциацию или как зависимость.

Системы

Помимо отображения отдельных компонентов, соединителей и их типов, перед нами стоит еще одна задача — инкапсулировать графы компонентов и соединителей в системы. Здесь есть три варианта.

- *Вариант 1: системы как подсистемы UML.* Основным механизмом группирования связанных элементов в UML является пакет. UML даже определяет стандартный стереотип пакета — **<<subsystem>>**, предназначенный для группирования моделей UML, являющихся представителями логических частей системы. Предусматриваемые подсистемы способны обеспечить любое отображение компонентов и соединителей, причем в наибольшей степени они пригодны для группирования классов. Исходя из спецификации UML 1.4, одна из проблем, связанных с применением систем, заключается в следующем: несмотря на то что каждая из них — это одновременно и классификатор, и пакет, их значение не совсем ясно. Согласно одной из точек зрения, на определенных этапах процесса разработки подсистему следует рассматривать как сущность типа класса, а впоследствии — уточнять ее до более детальной субструктуры. Будь у нас такая возможность, конструкция подсистемы могла бы стать более подходящей для моделирования архитектурных компонентов.
- *Вариант 2: системы как вложенные объекты.* Одним из средств отображения систем является вложенность объектов. Компоненты в этом случае отображаются в виде экземпляров вложенных классов, а соединители моделируются по одному из вышеизложенных вариантов. Объекты предусматривают жесткие границы инкапсуляции и наличие у каждого класса связанной с ним «субструктуры». Как бы то ни было, трудности есть и у этой методики. Главная из них заключается в том, что ассоциации, приме-

няемые при моделировании соединителей между вложенными классами, не предполагают определения классом области действия. Соответственно, утверждать, что любая отдельно взятая пара классов взаимодействует через определенный соединитель (моделируемый с помощью ассоциации) исключительно в контексте данной системы, нельзя. Таким образом, указание на взаимодействие через ту или иную ассоциацию двух вложенных классов распространяется на любые экземпляры этих классов в масштабах всей модели.

- *Вариант 3: системы как кооперации.* Для описания средствами UML набора сообщающихся объектов с установленными связями применяются кооперации. Отображая компоненты в виде объектов, мы имеем полное право отображать системы в виде кооперации. Кооперация определяет набор участников и отношений, имеющих значение в контексте поставленной задачи, которая в данном случае заключается в описании структуры системы периода прогона. Участники определяют роли классификаторов, которые объекты исполняют или с которыми согласуются в ходе взаимодействия. Отношения, соответственно, устанавливают роли ассоциации, которым должны соответствовать связи.

  Диаграммы сотрудничества обеспечивают отображение кооперации на уровнях спецификации или экземпляра. На диаграмме сотрудничества уровня спецификации изображаются роли, определенные в рамках данной кооперации и систематизированные согласно образцу описания субструктуры системы. Диаграмма сотрудничества уровня экземпляра демонстрирует объекты и связи, соответствующие заданным на уровне спецификации ролям и взаимодействующие в целях реализации поставленной задачи. Следовательно, для отображения структуры системы периода прогона лучше подходит кооперации уровня экземпляра.

  На рис. 9.12 вышеописанный метод представлен в виде схемы. Архитектурный тип Filter отображается так же, как и раньше. Экземпляры фильтров и каналов отображаются в виде соответствующих ролей классификатора (к примеру, /Splitter обозначает роль Splitter) и ассоциации. Соответствующие этим ролям объекты и связи показаны на диаграмме сотрудничества уровня экземпляра, а их имена выделены подчеркиванием.

  Рассмотренный способ, совершенно естественный в контексте описания структур периода прогона, не предполагает явного отображения свойств системного уровня. Кроме того, налицо семантическое несоответствие: кооперация описывает репрезентативное взаимодействие между объектами и предусматривает частичное описание, в то время как архитектурная конфигурация ориентирована на фиксацию полного описания.

Представления распределения

Диаграмма размещения в UML представляет собой граф с узлами, соединенными посредством коммуникативных ассоциаций. Соответствующий пример приводится на рис. 9.13. Если узел содержит экземпляры того или иного компонента,

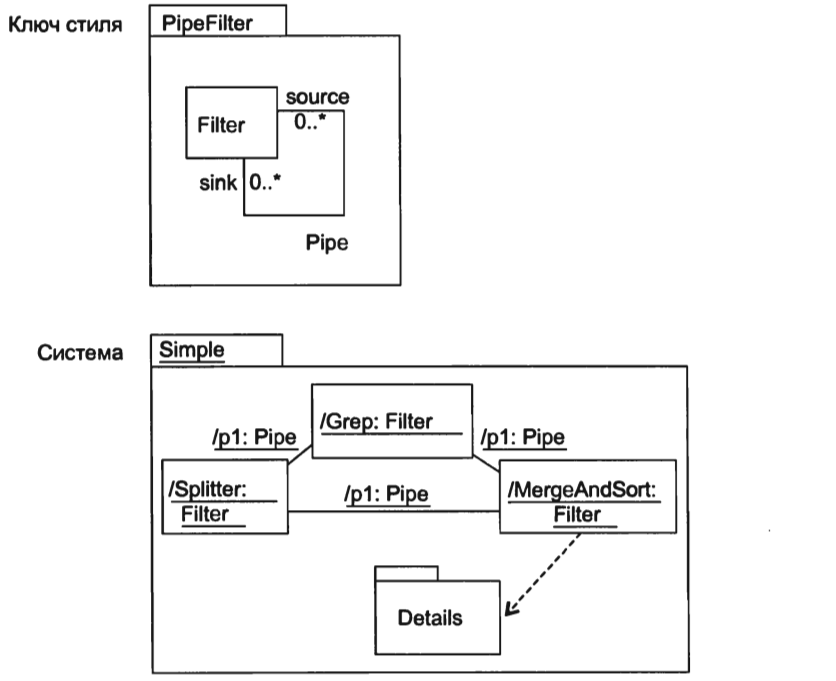


Рис. 9.12. Системы как кооперации

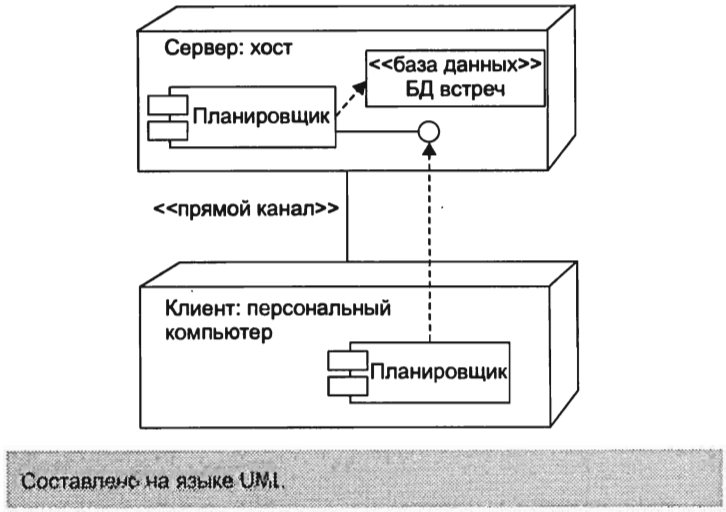


Рис. 9.13. Представление размещения в UML

значит, этот компонент существует или исполняется именно на нем. В составе компонентов могут находиться объекты, что, опять же, указывает на принадлеж-

ность данного объекта данному компоненту. Если соединение одних компонентов с другими обозначается пунктирной стрелкой зависимости (иногда — через интерфейсы), значит, один из этих компонентов обращается к службам других; при необходимости обозначения точной зависимости можно создать специальный стереотип. Помимо прочего, на диаграмме типа размещения можно показать, на каких узлах могут исполняться те или иные компоненты — для этого предназначены пунктирные стрелки со стереотипом <<supports>>.

Узел — это физический объект периода прогона, представляющий ресурс обработки. Последний характеризуется наличием памяти, а также (не всегда) вычислительных возможностей. Узлы соответствуют вычислительным устройствам, ресурсам ручной и механической обработки. Они также могут представлять типы экземпляров. В экземплярах узла могут размещаться вычислительные экземпляры периода прогона — как объекты, так и компоненты.

Связь между узлами устанавливается посредством ассоциаций. Ассоциация указывает на наличие между такими узлами канала передачи данных. Характер этого канала (например, тип физического канала или сети) обозначается стереотипами.

Вложенность символов в составе символа узла свидетельствует о композиционной ассоциации между узловым и внутренними классами или о композиционной связи между узловым и внутренними объектами.

9.7. Заключение

Грош цена той архитектуре, в которой никто не может разобраться. Этап документирования завершает процесс разработки архитектуры. На этом этапе архитектор ставит перед собой две задачи: во-первых, описать архитектуру в целях ознакомления с ней настоящих и будущих заинтересованных лиц, а во-вторых, избавить себя от необходимости отвечать на тысячи вопросов.

У вас должно сложиться четкое представление о составе заинтересованных в архитектуре лиц и их предпочтениях относительно применения документации. Документирование архитектуры в целом следует рассматривать как документирование набора наиболее значимых представлений с последующим введением сведений о перекрестной информации. При отборе представлений необходимо исходить из потребностей заинтересованных лиц.

Блочно-линейными диаграммами — будь они выражены средствами неформализованной нотации или, скажем, при помощи UML — документация не исчерпывается. Их следует дополнить вспомогательной документацией, разъясняющей суть элементов и отношений, представленных в первичном отображении. Сложно разобраться в архитектуре, не имея понятия о составляющих ее интерфейсах и поведении.

В этой главе представлена предписывающая структура документирования программной архитектуры. Почему же, спрашивается, мы не придерживались ее при описании конкретных примеров? Основной принцип составления технической документации в общем и документации программной архитектуры в частности гласит: излагайте материал, исходя из интересов предполагаемых читателей.

В нашем случае читателю нужен обзор системы, он хочет знать факторы, обусловившие формирование архитектуры в ее окончательном виде, и механизмы реализации задач по качеству — он не выступает в роли аналитика или, скажем, конструктора. Соответственно, представленные описания менее формальны и менее детальны, чем те, которые следует составлять в расчете на конструирование или анализ. Общие сведения мы излагаем при помощи первичных отображений (эскизов), но вместо формального каталога элементов, призванного детализировать содержание отображения, мы ограничиваемся повествовательным описанием.

9.8. Дополнительная литература

Значительная часть материала этой главы приводится в адаптированном виде по изданию [Clements 03]. В нем анализ архитектурной документации дается в более подробном виде. В документе [IEEE 00] изложен универсальный для всего сообщества стандарт документирования архитектуры, который, находясь в согласии с вышеприведенными данными, основывается на несколько отличной терминологии.

Добротных справочников по UML довольно много, однако наиболее удачным и полным до сей поры считается самый первый, хрестоматийный труд [Rumbaugh 99]. Рабочая группа OMG в настоящее время трудится над новой версией UML, ориентированной на более полное отображение программной архитектуры систем. Следить за ходом работ помогает сайт http://www.omg.org/uml/.

9.9. Дискуссионные вопросы

1. Какие из упомянутых в этой главе представлений имеют наибольшее значение для системы, над которой вы работаете в данный момент? Какие из них вы уже документировали? Почему этот вопрос так важен?
2. Предположим, что вы только что вошли в группу разработчиков проекта. В какой последовательности вы намерены знакомиться с документацией, для того чтобы войти в курс дела?
3. Какая документация требуется для проведения анализа производительности?

Глава 10
Реконструкция программной архитектуры

(в соавторстве с Джероми Карье, Лайамом О'Брайеном и Крисом Вероефом)[1]

> Прошлое, будущее и настоящее в равной степени покрыты завесой тайны, и задача историка — раскрыть суть явлений, происходящих сегодня.
>
> *Генри Дэвид Торо*

10.1. Введение

Вероятно, вы уже привыкли к тому, что архитектура рассматривается нами как нечто подконтрольное архитектору. Мы показали, как путем принятия архитектурных решений (а в части 3 речь пойдет об анализе этих решений) реализуются задачи и требования, предъявляемые к разрабатываемой системе. Но есть и альтернативный вариант развития событий. Предположим, что перед нами существующая система, архитектура которой нам неизвестна. Возможно, разработчики архитектуры решили не утруждаться ее документированием. Может быть, документация утеряна. Наконец, возможен и такой вариант, при котором после ряда модификаций исходная архитектура уже не соответствует фактическому состоянию системы. Что делать с сопровождением такой системы? Как контролировать ее развитие, как сохранить соответствие ее архитектуры (что бы она собой ни представляла) предъявленным к системе требованиям по качеству?

---

[1] Джероми Карье (Jeromy Carriere) — младший сотрудник Microsoft; Лайам О'Брайен (Liam O'Brien) — научный сотрудник Института программной инженерии; Крис Вероеф (Chris Verhoef) работает в Открытом университете Амстердама.

Ответы на перечисленные вопросы дает представленная в настоящей главе методика реконструкции архитектуры (architecture reconstruction), которая позволяет восстановить «фактическую» (as-built) архитектуру реализованной системы исходя из ее современного состояния. Для этого с помощью определенных инструментальных средств, которые собирают информацию о системе, последовательно строят и группируют уровни абстракции, проводится детальный анализ системы. Если инструменты успешно справляются со своей задачей, в конечном итоге мы получаем архитектурное отображение, дающее возможность строить о системе некие умозаключения. Иногда сгенерировать сколько-нибудь пригодное отображение не удается. В частности, это характерно для устаревших систем, в которых связного архитектурного решения не предполагалось изначально (впрочем, такой вывод полезен сам по себе).

Реконструкция архитектуры является интерпретирующим, интерактивным и итеративным процессом; она состоит из многочисленных операций и не является автоматической. В ее проведении должны принимать участие специалист по обратной разработке и, поскольку архитектурные конструкции не находят явного отражения в исходном коде, архитектор (или иной специалист, обладающий достаточными знаниями об архитектуре). В языках программирования отсутствуют конструкции, соответствующие «уровню», «соединителю» и прочим архитектурным элементам; следовательно, вычленить их из файла с исходным кодом непросто. Что касается архитектурных образцов, то они, если и используются, редко каким-либо образом обозначаются. Архитектурные конструкции отражаются в реализации через самые разнообразные механизмы — как правило, это коллекции функций, классов, файлов, объектов и т. д. Во время первоначальной разработки системы осуществляется отображение ее высокоуровневых проектных/архитектурных элементов на элементы реализации. Таким образом, при реконструкции этих элементов требуется провести обратное отображение, а для этого необходимы определенные познания в области архитектуры. Не менее важную роль играет владение конструктивными методиками компиляторов и утилитами наподобие grep, sed, awk, perl, python и lex/yacc.

Существует несколько вариантов применения результатов архитектурной реконструкции. Если документация отсутствует или устарела, то на базе восстановленного архитектурного отображения систему можно документировать заново (см. главу 9). Кроме того, после реконструкции фактической (as-built) архитектуры ее можно сравнить с проектной (as-designed) архитектурой. По итогам этой проверки мы можем убедиться в том, что специалисты по сопровождению (или, если уж на то пошло, разработчики) следуют «заповедям» архитектуры и не подрывают ее основ, разрушая абстракции, сводя воедино разные уровни, нарушая принцип информационной закрытости и т. д. По результатам реконструкции можно проводить архитектурный анализ (см. главы 11 и 12) или приводить систему в соответствие с новой версией архитектуры. Наконец, полученное отображение помогает выявлять элементы, допускающие повторное использование, и определять линейки продуктов на основе архитектуры (см. главу 14).

Методика архитектурной реконструкции нашла применение в самых разных проектах — от измерителей магнитного резонанса до автоматических телефон-

ных коммутаторов и от вертолетных навигационных систем до засекреченных систем NASA. Помимо прочего, она использовалась для:

- повторного документирования архитектур систем физического моделирования;
- выявления архитектурных зависимостей во встроенных системах управления горным оборудованием;
- оценки соответствия реализации наземной станции из системы спутниковой связи эталонной архитектуре;
- описания различных систем в автомобильной промышленности.

Принцип инструментария

Для проведения реконструкции архитектуры требуются специальные инструментальные средства; с другой стороны, ни один инструмент или набор инструментов не способен справиться с этой задачей самостоятельно. Во-первых, поскольку в исследуемых артефактах может встретиться множество различных языков, желательно, чтобы эти инструменты были ориентированы на конкретные языки. К примеру, программное обеспечение, находящееся в составе комплексного измерителя магнитного резонанса, может быть написано на 15 различных языках. Во-вторых, инструменты извлечения данных несовершенны — зачастую они возвращают неполные или ошибочные результаты; следовательно, имеет смысл задействовать несколько подобного рода инструментов, которые могли бы дополнять и проверять показания друг друга. Наконец, как мы уже говорили, перед реконструкцией ставятся разные задачи. От планов применения восстановленной документации зависит спектр извлекаемой информации, а он, в свою очередь, определяет набор инструментальных средств.

Из всего этого следует, что для проведения реконструкции архитектуры необходимо применять наборы инструментальных средств — так называемые *инструментарии* (workbenches). Инструментарий должен быть открытым (удобным в смысле введения новых инструментов) и предусматривать облегченный интеграционный каркас, исключающий излишнее воздействие новых инструментов на старые инструменты и данные.

Один из инструментариев разработан в Институте программной инженерии — он называется Dali, и для иллюстрирования материалов этой главы мы к нему еще обратимся. Несколько других инструментариев упоминается в разделе «Дополнительная литература» в заключительной части главы.

Операции в ходе реконструкции

В ходе реконструкции программной архитектуры выполняется ряд операций итерационного характера:

1. *Извлечение информации.* Цель — извлечь из разных источников нужную информацию.
2. *Составление базы данных.* Извлеченная информация стандартизируется — например, переводится в стандартную форму Rigi (формат данных на основе

кортежей вида отношение<объект1><объект2>) — и преобразуется в формат записей базы данных на основе SQL; по результатам этих преобразований создается база данных.

3. *Объединение представлений.* Информация, содержащаяся в базе данных, объединяется, формируя связное представление архитектуры.
4. *Реконструкция.* На этом этапе производятся важнейшие действия по построению абстракций и отображению данных, на основе которых генерируется отображение архитектуры в целом.

Как и следовало ожидать, все эти процессы итерационны. Схема операций по реконструкции архитектуры и проходящих между ними информационных потоков приводится на рис. 10.1.

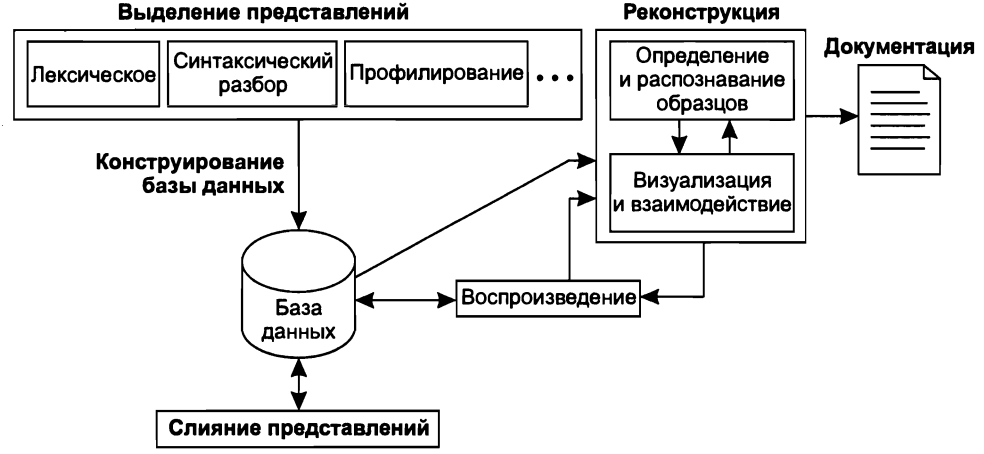


Рис. 10.1. Операции реконструкции архитектуры (стрелки обозначают информационные потоки между операциями)

Совершенно необходимо участие в процессе реконструкции нескольких человек: во-первых, специалиста, отвечающего непосредственно за реконструкцию, и, во-вторых, одного или нескольких людей, знакомых с реконструируемой системой (например, ее архитекторов и программных инженеров).

Извлекая из системы нужную информацию, специалист по реконструкции вручную или при помощи инструментальных средств строит абстракции и таким образом, восстанавливает черты архитектуры. При этом он выдвигает ряд гипотез о системе, выражающих обратные отображения исходных артефактов на проектные решения (в идеале, они должны быть противоположны вариантам проектного отображения). Гипотезы тестируются путем генерирования обратных отображений, применения их к извлеченной информации и проверки результата на правильность. Повышение эффективности при генерации и проверке гипотез достигается за счет привлечения знакомых с системой людей — в частности, ее архитектора или программных инженеров (тех, кто участвовал в первоначальной разработке системы, и тех, кто занимается ее сопровождением в настоящее время).

В нижеследующих разделах мы более подробно рассмотрим отдельные операции реконструкции архитектуры и изложим практические рекомендации по их

проведению. По большей части эти рекомендации не привязаны к конкретному инструментарию и поэтому потенциально применимы даже в тех случаях, когда реконструкция проводится вручную.

10.2. Извлечение информации

Этап извлечения информации подразумевает анализ фактического проектного решения и артефактов реализации системы с целью конструирования ее модели. Результатом проведения этого этапа должен стать набор информации, размещенный в базе данных, который в ходе объединения представлений обеспечивает конструирование представления системы.

У процесса извлечения информации две составляющие: теоретическая (какая информация об архитектуре способна максимально повысить эффективность реконструкции?) и практическая (какую информацию можно извлечь и представить с помощью имеющихся инструментов?). Имея в наличии исходные (код, заголовочные файлы, файлы конструкций и т. д.) и другие артефакты (например, кальки исполнения), можно выявить и зафиксировать интересующие элементы системы (файлы, функции, переменные) и установленные между ними отношения; этих сведений вполне достаточно для построения базовых представлений системы. Список элементов и отношений между ними, которые в большинстве случаев удается извлечь, представлен в табл. 10.1.

Таблица 10.1. Элементы и отношения, которые чаще всего удается извлечь

Исходный элемент	Отношение	Целевой элемент	Описание элемента
Файл	includes	Файл	Препроцессор #include в языке C, указывающий на включение одного файла в другой
Файл	contains	Функция	Определение функции в файле
Файл	defines_var	Переменная	Определение переменной в файле
Каталог	contains	Каталог	В каталоге присутствует подкаталог
Каталог	contains	Файл	В каталоге присутствует файл
Функция	calls	Функция	Вызов статической функции
Функция	access_read	Переменная	Доступ к переменной для чтения
Функция	access_write	Переменная	Доступ к переменной для записи

Все отношения между элементами по-своему содержательны. Отношение calls между функциями помогает построить граф вызываемых функций. Отношение includes между файлами сообщает о тех или иных зависимостях между системными файлами. Отношения access_read и access_write между функциями и переменными иллюстрируют использование данных — ведь одни функции записывают наборы данных, а другие их считывают. Исходя из этого можно делать выводы относительно передачи данных между различными частями системы — к примеру, установить наличие глобального хранилища данных или утвердиться во мнении о том, что основным механизмом передачи данных являются вызовы функций.

В ходе анализа крупной системы полезно обращать внимание на принцип расположения файлов в структуре каталогов — эти данные могут пригодиться при реконструкции. Поскольку элементы или подсистемы хранятся в определенных каталогах, фиксация отношений типа dir_contains_file и dir_contains_dir помогает идентифицировать элементы в дальнейшем.

Конкретный набор извлеченных элементов и отношений зависит, во-первых, от типа анализируемой системы и, во-вторых, от привлеченных инструментальных средств. Если система, которую предполагается реконструировать, относится к числу объектно-ориентированных, значит, помимо прочего, необходимо извлечь классы и методы (то есть элементы), а также отношения, например class_is_subclass_of_class и class_contains_method.

Полученная в результате анализа информация делится на статическую и динамическую. Для получения статической информации достаточно исследовать артефакты системы; динамическая информация, напротив, подразумевает анализ ее работы. В результате объединения этих двух видов можно достичь более высокой точности представления системы. (На тему объединения представлений мы поговорим в разделе 10.4.) Если архитектура системы предполагает изменения в период прогона (например, если при запуске системы она считывает конфигурационный файл, в результате чего выполняется загрузка новых элементов), при реконструкции не обойтись без фиксирования конфигурации для периода прогона.

Инструментов, применяемых для извлечения информации, великое множество; в частности, среди них есть следующие:

- синтаксические анализаторы (например, Imagix, SNiFF++, CIA, rigiparse);
- анализаторы абстрактно-синтаксических деревьев (AST; например, Gen++, Refine);
- лексические анализаторы (например, LSME);
- профайлеры (например, gprof);
- инструменты измерения кода;
- специализированные средства (например, grep, perl).

Синтаксические анализаторы выполняют анализ кода и генерируют на его основе внутренние отображения (которые, в свою очередь, предназначены для генерации машинного кода). Как правило, такое отображение можно преобразовать в представление. Анализаторы абстрактно-синтаксических деревьев (AST) выполняют примерно ту же функцию — с той лишь разницей, что на основе проанализированной информации они строят явное древовидное отображение. В наших силах создать такие инструменты анализа, которые по результатам своего прохода по AST смогут выводить в требуемом формате отдельные блоки информации, значимые с точки зрения архитектуры.

Что касается лексических анализаторов, то исходные артефакты анализируются ими исключительно как строки лексических элементов или символов. У пользователя лексического анализатора есть возможность задать набор образцов кода, случаи совпадения с которыми следует выводить. Аналогичным образом действуют специализированные инструменты наподобие grep и perl — зная, какая информация требуется пользователю, они ищут в коде соответствия установлен-

ным образцам. Все вышеупомянутые средства — синтаксические анализаторы с генерацией кода, анализаторы на основе абстрактно-синтаксических деревьев, синтаксические анализаторы и специализированные средства сопоставления с образцами — выводят статическую информацию.

Инструменты профилирования и анализа покрытия кода выводят информацию о коде во время его исполнения; кроме того, они в большинстве случаев не добавляют в систему новый код. Инструменты измерения кода, нашедшие широкое применение в области тестирования, напротив, предусматривают ввод в исполняемую систему нового кода с целью вывода информации определенного характера. Все эти средства генерируют динамические представления системы.

При необходимости из проектных моделей, файлов конструкций, сборочных и исполняемых файлов можно извлечь дополнительную информацию. К примеру, в файлах конструкций и сборочных файлах содержится информация о зависимостях между модулями или файлами системы, которая может быть не выражена в исходном коде.

Большое количество архитектурной информации можно статическим методом извлечь из исходного кода, артефактов периода компиляции и проектных артефактов. С другой стороны, учитывая возможность динамического связывания, наличие в исходных артефактах всего комплекса информации, значимой с точки зрения архитектуры, не гарантируется. Динамическое связывание, в частности, характерно для:

- полиморфизма;
- указателей функций;
- параметризации периода прогона.

Иногда топологию системы нельзя определить вплоть до прогона. К примеру, многопроцессные и многопроцессорные системы, задействующие промежуточное ПО наподобие J2EE, Jini или .NET, зачастую прибегают к динамическому построению топологии в зависимости от доступности системных ресурсов. Поскольку топология таких систем не отражена в исходных артефактах, ее обратная разработка при помощи статических инструментов извлечения невозможна.

Отсюда необходимость в инструментах, способных генерировать динамическую информацию о системе (к примеру, в инструментах профилирования). Само собой разумеется, что они должны работать на той же платформе, на которой исполняется система. Кроме того, следует иметь в виду трудности, связанные со сбором результатов работы инструментов измерения кода. Во встроенных системах зачастую отсутствуют средства вывода подобной информации.

Практические рекомендации

Ниже изложены некоторые наши соображения, связанные с выполнением рассматриваемого этапа методики реконструкции.

- *Старайтесь минимизировать издержки извлечения.* Тщательно подумайте над тем, какую именно информацию вам требуется извлечь из корпуса исходного кода. Является ли эта информация лексической по своему характеру?

Предполагает ли она исследование сложных синтаксических структур? Есть ли необходимость в проведении синтаксического анализа? Каждому из вариантов ответов соответствует тот или иной инструмент, которого оказывается достаточно для успешного выполнения задачи. Имейте в виду, что лексические методики, как правило, связаны с наименьшими издержками, и, если задачи по реконструкции не отличаются высокой сложностью, отдавать предпочтение следует именно им.

- *Проверяйте извлеченную информацию на правильность.* Прежде чем приступать к объединению полученных представлений и их обработке, проверьте правильность зафиксированной в рамках каждого представления информации. Корректность результатов работы инструментов, ответственных за анализ исходных артефактов, крайне важна. Для того чтобы убедиться в правильности зафиксированной информации, в первую очередь необходимо вручную исследовать и проверить подмножество элементов и отношений в сравнении с базовым исходным кодом. Решение об объеме информации, требующей проверки, вы должны принять самостоятельно. Сформируйте статистическую выборку, установите доверительный уровень и найдите подходящую выборочную стратегию.
- *При необходимости извлекайте динамическую информацию.* Такая необходимость возникает, если велика продолжительность периода прогона, значительная роль в системе отводится динамическому связыванию или архитектура является динамически настраиваемой.

10.3. Создание базы данных

На этапе создания базы данных извлеченная информация преобразуется в стандартный формат и сохраняется в базе данных. В первую очередь необходимо выбрать модель базы данных. При этом полезно руководствоваться следующими соображениями:

- Выбранная модель должна быть широко известной — это значительно упрощает задачу замены одной реализации базы данных другой.
- Учитывая потенциально большие объемы исходных моделей, следует максимально повысить эффективность подачи запросов.
- Необходима возможность удаленного доступа к базе данных посредством одного или нескольких географически распределенных пользовательских интерфейсов.
- Путем комбинирования информации из различных таблиц следует реализовать объединение представлений.
- Необходима поддержка языков запросов, способных выражать архитектурные образцы.
- Реализации должны предусматривать возможность введения контрольных точек и, следовательно, сохранения промежуточных результатов. В условиях интерактивного процесса она предоставляет пользователю возмож-

ность действовать свободно, не бояться вносить изменения, поскольку их в любой момент можно отменить.

К примеру, в инструментарии Dali применяется реляционная модель базы данных. Она предполагает преобразование извлеченных представлений (которые, в зависимости от инструментов извлечения, могут существовать в самых разных форматах) в стандартную форму Rigi. Затем данные считываются сценарием perl и выводятся в формате, предусматривающем наличие кода SQL, который позволяет конструировать реляционные таблицы и наполнять их извлеченной информацией. Схема этого процесса изображена на рис. 10.2.

Рис. 10.2. Преобразование извлеченной информации в формат SQL

Пример кода SQL, предназначенного для построения и наполнения реляционных таблиц, показан в листинге 10.1.

При вводе данных в базу генерируются две добавочные таблицы: elements и relationships. В них содержатся списки извлеченных элементов и отношений соответственно.

Для преобразования из формата (форматов), применяемого утилитой извлечения, можно задействовать любые новые инструментальные средства и методики. Такую возможность обеспечивают инструментарии. К примеру, если для обработки нового языка требуется новый инструмент, его можно сконструировать, а его выходные данные можно преобразовать в формат инструментария.

В текущей версии инструментария Dali функциональность реляционной базы данных POSTGRES формируется средствами SQL и perl, которые отвечают за генерирование и обработку архитектурных представлений (соответствующие примеры приводятся в разделе 10.5). Редактировать сценарии SQL в целях обеспечения их совместимости с другими реализациями SQL не представляет труда.

Листинг 10.1. Пример кода SQL, сгенерированного средствами инструментария Dali

```
create table calls( caller text, callee text );
create table access( func text, variable text );
create table defines_var( file text, variable text );
...
insert into calls values( 'main', 'control' );
insert into calls values( 'main', 'clock' );
...
insert into accesses values( 'main', 'stat 1' );
```

Практические рекомендации

На этапе создания базы данных имеет смысл учитывать следующие соображения.

- Обработка представлений данных (при объединении представлений) упрощается, если таблицы базы данных создаются на основе извлеченных отношений. К примеру, если сохранить в такой таблице результаты определенного

запроса, исполнять его повторно не придется. При необходимости получения аналогичных результатов можно будет обратиться к этой таблице.

- Перед конструированием любой базы данных необходимо тщательно продумать ее структуру. Каким будет первичный (а возможно, и вторичный) ключ? Какие соединения, охватывающие сразу несколько таблиц, будут сопряжены с наибольшими издержками? Применяемые в ходе реконструкции таблицы, как правило, не отличаются особой сложностью — порядка dir_contains_dir или function_calls_function, а в качестве первичного ключа выступает функция всей строки.
- Для преобразования формата данных, извлеченных специализированными инструментальными средствами, в формат, поддерживаемый инструментарием, полезно применять простые инструменты лексического анализа наподобие perl и awk.

10.4. Объединение представлений

На этапе объединения представлений производится определение и обработка извлеченной информации (которая теперь хранится в базе данных); эти операции направлены на согласование и расширение элементов, а также на установление между ними соединений. Для получения дополнительной информации задействуются разного рода методики извлечения. Уяснить принципы объединения помогают приведенные в последующих разделах примеры.

Импорт представления

Рассмотрим две выборки, показанные на рис. 10.3. Они заимствованы из наборов методов (каждому из которых предшествует соответствующий класс), извлеченных из реализованной на C++ системы. В таблицах представлена статическая и динамическая информация об объектно-ориентированном сегменте кода. К примеру, в таблице динамической информации показан вызов метода List::getnth. С другой стороны, пропущенный статическим инструментом извлечения, этот метод не вошел в состав результатов статического анализа. Кроме того, среди статической информации нет вызовов метода-конструктора и метода-деструктора InputValue и List; их придется добавить в таблицу классов/методов, которая выполняет согласование обоих источников информации.

Далее, согласно приведенным в настоящем примере результатам статического извлечения, у класса PrimitiveOp есть метод под именем Compute. Динамическое извлечение не обнаруживает такого класса, однако в то же время демонстрирует наличие таких классов, как ArithmeticOp, AttachOp и StringOp, у которых есть метод Compute и которые фактически являются подклассами PrimitiveOp. Итак, поскольку PrimitiveOp, очевидно, является суперклассом, в исполняемой программе он не вызывается. Однако статический инструмент извлечения, сканируя исходный код, обнаруживает вызов PrimitiveOp; полиморфный вызов одного из подклассов PrimitiveOp, напротив, происходит в период прогона.

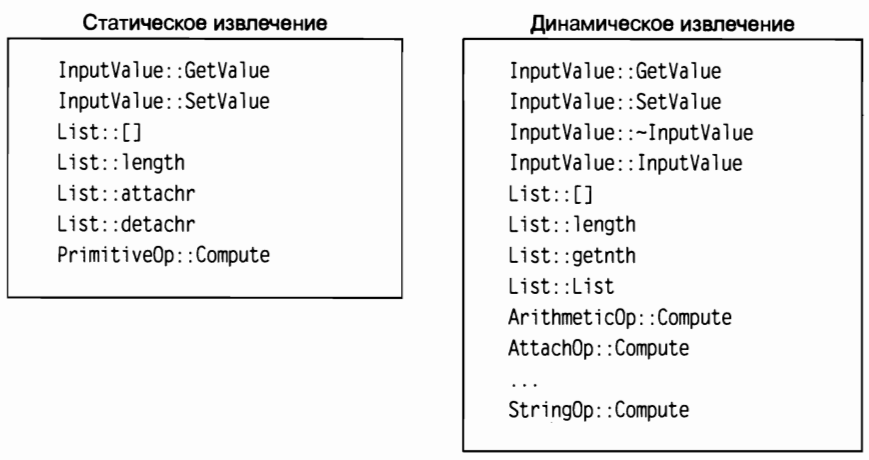


Рис. 10.3. Статическая и динамическая информация об отношении class_contains_method

Для того чтобы сформировать точное представление архитектуры, необходимо согласовать статическую и динамическую информацию о суперклассе PrimitiveOp. Соответственно, нужно провести объединение, для чего требуется сформировать SQL-запросы на извлеченные отношения calls, actually_calls и has_subclass. По результатам видно, что вызовы суперкласса PrimitiveOp::Compute (полученные на основе статической информации) и его многочисленных подклассов (полученные на основе динамической информации) — это фактически одно и то же.

В двух списках на рис. 10.4 показаны элементы, добавленные в объединенное представление (в дополнение к тем, которые присутствуют и в статической, и в динамической информации) и исключенные из него (несмотря на их наличие в статической или динамической информации).

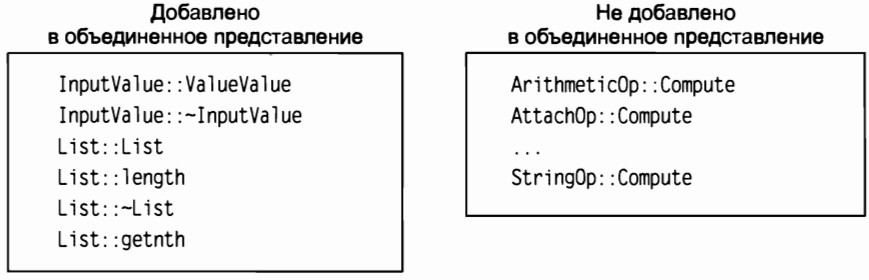


Рис. 10.4. Элементы, добавленные в объединенное представление и исключенные из него

Устранение неоднозначности вызовов функций

В многопроцессных приложениях велика вероятность возникновения конфликтов имен. Например, процедура под именем main может встречаться сразу в нескольких процессах. В рамках извлеченных представлений крайне важно устанавливать и устранять подобные конфликты. Стоит повторить, что исключению потенциальной неоднозначности способствует объединение легко извлекаемой

информации. В данном случае требуется объединить статическую таблицу calls с таблицей «включения функций в файлы» (выражающей соответствие определений функций и исходных файлов) и таблицей «зависимости между конструкциями» (устанавливающей соответствие между компилируемыми исходными и исполняемыми файлами). Объединение этих источников информации позволяет превратить потенциально неоднозначные имена процедур и методов в уникальные, допускающие возможность однозначного обращения в процессе архитектурной реконструкции.

Практические рекомендации

На этом этапе также следует учитывать ряд соображений.

- Если по отдельности извлеченные таблицы оказываются не в состоянии предоставить необходимую информацию, объедините их.
- Если внутри одной из таблиц существует неоднозначность и устранить ее без обращения к другим таблицам не представляется возможным, объедините таблицы.
- В зависимости от информации, которую предполагается извлечь, имеет смысл задействовать разные методики; к примеру, в вашем распоряжении динамический и статический методы извлечения. Кроме того, в случае появления подозрений на ошибочность или неполноту информации, предоставляемой тем или иным инструментом, можно ввести новые средства, относящиеся к той же методике, — например, воспользоваться несколькими синтаксическими анализаторами одного языка.

10.5. Реконструкция

Итак, к настоящему моменту информация, составляющая представление, извлечена, сохранена, уточнена или расширена. Представления в ходе реконструкции задействуются для выявления приблизительных, общего характера сведений об архитектуре. Реконструкция распадается на две основные операции: *визуализация и взаимодействие* и *определение и распознавание образцов*. Рассмотрим их по отдельности.

Визуализация и взаимодействие (visualization and interaction) — это механизм, позволяющий пользователю визуализировать, исследовать и управлять представлениями в интерактивном режиме. В Dali пользователь просматривает представления на графе с иерархической декомпозицией элементов и отношений, для чего существует инструмент Rigi. Пример архитектурного представления показан на рис. 10.5.

Определение и распознавание образцов (pattern definition and recognition) — это механизм архитектурной реконструкции: определения и распознавания проявления кода в архитектурных образцах. Средства реконструкции инструментария Dali, к примеру, позволяют пользователю выявлять группировки элементов и за счет этого конструировать из детальных представлений довольно абстракт-

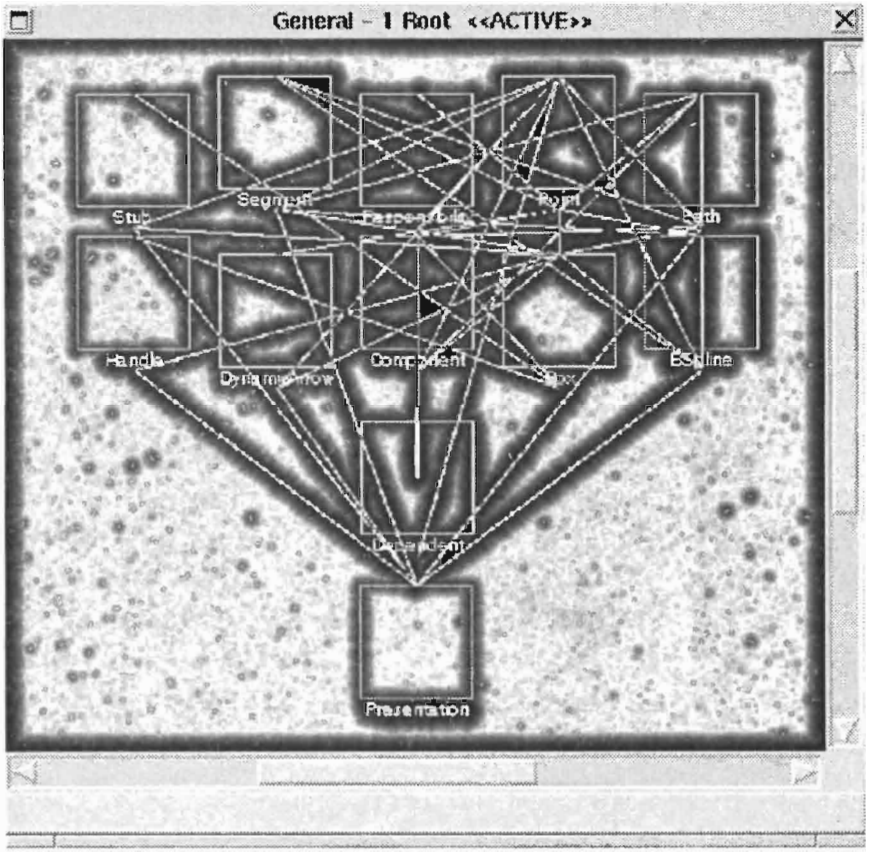


Рис. 10.5. Архитектурное представление, отображенное Dali

ные представления программной системы. Образцы, которые в Dali определяются путем сочетания средств SQL и perl, более известны нам под названием *сегментов кода* (code segments). С помощью запросов SQL в репозитарии Dali выявляются те элементы, которые участвуют в новой группировке; выражения perl, в свою очередь, применяются для преобразования имен и выполнения других действий по обработке результатов запросов. Сохраненные сегменты кода пользователи могут выборочно задействовать при выполнении текущей операции, а также обращаться к ним впоследствии.

Исходя из совокупности архитектурных образцов, которые, по мысли архитектора, должны присутствовать в системе, специалист по реконструкции составляет запросы. По результатам этих запросов формируются очередные группировки, демонстрирующие разного рода абстракции и кластеризацию низкоуровневых элементов (в качестве таковых выступают исходные артефакты или абстракции). Путем интерпретирования этих представлений и их активного анализа запросы и группировки уточняются, что, в свою очередь, обусловливает появление ряда гипотетических архитектурных представлений, предусматривающих возможность дальнейшей интерпретации, уточнения или отклонения. Универсальных критериев

завершения этого процесса не существует; считать его доведенным до конца, вероятно, следует на том этапе, когда полученное архитектурное представление оказывается достаточным для проведения анализа и документирования.

Предположим, что наша база данных состоит из подмножества элементов и отношений, представленного на рис. 10.6. Переменные a и b в данном примере определяются в функции f; таким образом, по отношению к f они являются локальными. Эту информацию можно представить графически — так, как это сделано на рис. 10.7.

Элемент	Отношение	Элемент
f	defines_var	a
f	defines_var	b
g	calls	f
f	calls	h

Рис. 10.6. Подмножество элементов и отношений

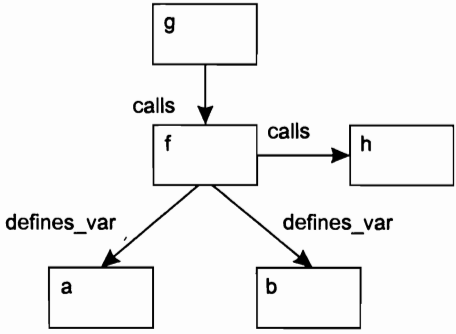


Рис. 10.7. Графическое отображение элементов и отношений

Локальные переменные не сообщают никаких существенных сведений об архитектуре системы в целом; следовательно, в контексте архитектурной реконструкции они не слишком интересны. Таким образом, экземпляры локальных переменных можно сгруппировать в соответствующие функции. Пример кода SQL и perl, выполняющего эту задачу, показан в листинге 10.2.

В первой части этого кода обновляется визуальное отображение — вслед за именем каждой функции ставится символ «+». Затем функция группируется с определенной в ней локальной переменной. Запрос SQL отбирает из таблицы элементов нужные функции, после чего в отношении каждой строки результата исполняется выражение perl. Массив $fields автоматически заполняется полями согласно результату запроса; в данном случае из таблицы извлечено единственное поле (tName), а это значит, что его значение в $fields[0] будет сохраняться для каждого отобранного кортежа. Выражение генерирует строки в следующей форме:

```
<function>+  <function>  Function
```

Итак, элемент <function> следует сгруппировать до <function>+ с типом Function.

Листинг 10.2. Группировка локальных переменных с определяющей их функцией средствами SQL и perl

```
# Агрегирование локальной переменной
SELECT tName
    FROM Elements
    WHERE tType='Function';
print ''$fields[0]+ $fields[0] Function\n'';

SELECT d1.func, d1.local_variable
    FROM defines_var d1;
print ''$fields[0] $fields[1] Function\n'';
```

Во второй части кода локальные переменные исключаются из визуализации. Запрос SQL идентифицирует локальные переменные во всех функциях, определенных путем отбора кортежей в таблице defines_var. Так, $fields[0] в выражении perl соответствует полю func, а $fields[1] — полю local_variable. Таким образом, выходные данные будут представлены в следующей форме:

```
<function>+  <variable>  Function
```

Другими словами, в агрегат <function>+ вводятся все ее локальные переменные. Последовательность исполнения двух вышеприведенных сегментов кода не принципиальна, так как конечные результаты подачи обоих запросов подвергаются сортировке.

Графическое отображение результатов исполнения представленных сегментов кода показано на рис. 10.8.

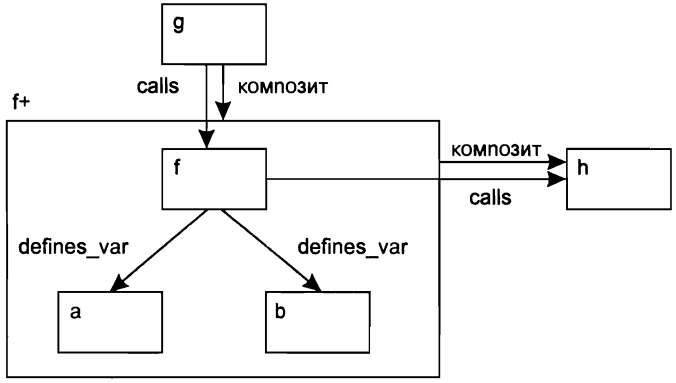


Рис. 10.8. Результат исполнения сегмента кода, показанного в листинге 10.2

Основным механизмом обработки извлеченной информации является обратное отображение, примерами которого, в частности, являются:

- идентификация типов;
- группировка локальных переменных в функции;
- группировка членов в классы;
- композиция элементов архитектурного уровня.

Пример запроса, идентифицирующего архитектурный элемент, показан в листинге 10.3. Выполняя идентификацию архитектурного элемента Logical_Interaction,

он попутно сообщает о том, что если именем класса является Presentation, Bspline или Color либо если этот класс является подклассом Presentation, то он принадлежит к элементу Logical_Interaction.

Причина, по которой сегменты кода составляются указанным образом, заключается в абстрагировании от низкоуровневой информации с целью генерации представлений архитектурного уровня. Составляя эти сегменты, специалист по реконструкции руководствуется целью проверить гипотезы о системе. Если тот или иной сегмент не приводит к получению полезных результатов, его можно отклонить. Специалист повторяет этот процесс вплоть до получения полезных архитектурных представлений.

Листинг 10.3. Запрос, направленный на идентификацию элемента Logical_Interaction

```
SELECT tSubclass
    FROM has_subclass
    WHERE tSuperclass='Presentation';
print ''Logical_Interaction $fields[0]'';

SELECT tName
    FROM element
    WHERE tName='Presentation'
    OR tName='BSpline'
    OR tName='Color';
print ''Logical_Interaction $fields[0]'';
```

Практические рекомендации

Ниже приводится ряд практических советов по осуществлению рассматриваемого этапа.

- Будьте готовы к тому, что вам придется активно взаимодействовать с архитектором и провести несколько итераций по созданным архитектурным абстракциям. Этот режим особенно актуален в отсутствие у системы явной, документированной архитектуры. (См. врезку «Игра «Поймай архитектуру»».) В таких случаях имеет смысл начинать с построения архитектурных абстракций в виде гипотез, а затем, формируя представления и демонстрируя их архитектору и другим заинтересованным лицам, проверять их правомерность. Исходя из выявленных ложноотрицательных и ложноположительных результатов специалист по реконструкции принимает решения о создании новых абстракций и, следовательно, исполнении очередных сегментов кода Dali (возможно, даже новых операций по извлечению информации).

- Разрабатывая сегменты кода, старайтесь соблюдать краткость и не перечислять все исходные элементы. Удачный в этом отношении пример сегмента кода показан в листинге 10.3, неудачный — в листинге 10.4. В последнем составляющие архитектурный элемент исходные элементы просто перечисляются, в результате чего он становится трудным для понимания и применения (в том числе повторного).

Листинг 10.4. Пример неудачного сегмента кода, зависящего от явного перечисления интересующих элементов

```
SELECT tName
    FROM element
    WHERE tName='vanish-xforms.cc'
    OR tName='PrimativeOp'
    OR tName='Mapping'
    OR tName='MappingEditor'
    OR tName='InputValue'
    OR tName='Point'
    OR tName='VEC'
    OR tName='MAT'
    OR ((tName ~ 'Dbg$' OR tName ~ 'Event$')
        AND tType='Class');
print ''Dialogue $fields[0]'';
```

- В качестве основы для построения сегментов кода можно использовать соглашения о присваивании имен, хотя в этом случае они должны последовательно соблюдаться в масштабе всей системы. Скажем, в соответствии с таким соглашением имена всех функций, данных и файлов, принадлежащих к элементу Interface, должны начинаться с префикса i_.
- Кроме того, основой для построения сегментов кода может быть структура каталогов, в которых хранятся файлы и функции. Исходя из этой структуры можно проводить группировку элементов.
- Архитектурная реконструкция, направленная на выявление архитектурных решений, исходит исключительно из результатов их принятия, выраженных в фактических артефактах (другими словами, из реализующего их кода). По мере проведения реконструкции требуется дополнительная информация, помогающая в процессе повторного принятия архитектурных решений; поскольку на этом этапе специалист, проводящий реконструкции, рискует проявить необъективность, к работе необходимо привлечь человека, обладающего серьезными знаниями о реконструируемой архитектуре.

ИГРА «ПОЙМАЙ АРХИТЕКТУРУ»

Перспектива восстановления «утерянной» архитектуры часто приводит в уныние. Оно и понятно — ведь команде обратной разработки предстоит с чистого листа восстановить архитектуру, которая должна, с одной стороны, адекватно отображать фактическое состояние системы, а с другой — предусматривать возможность построения умозаключений о системе, ее сопровождении, развитии и т. д.

Приступать к архитектурной реконструкции проекта имеет смысл лишь в тех случаях, когда архитектурная документация утеряна или с течением времени, в результате многочисленных пересмотров, выполненных разными специалистами, потеряла связь с реальностью. Так с чего же начать?

Когда нам впервые пришлось провести реконструкцию архитектуры нескольких систем, у нас были другие посылки. Закончив с разработкой инструментария Dali, мы должны были на чем-то его протестировать. Имея это в виду, мы отобрали несколько систем собственной разработки и с собственноручно спроектированной архитектурой. Создавая эти системы, мы стремились к построению явной архитектуры, вследствие чего их восстановление не представляло проблемы. И тем не менее без сюрпризов не обошлось. Нарушения в архитектуре обнаружились даже в относительно небольших системах, спроектированных и кодированных нашими же руками. Это обстоятельство привело нас в некоторое недоумение: если даже небольшие системы с добротной архитектурой оказались несовершенными, то чего же ожидать от крупных коммерческих систем приличной «выдержки»? Приободрившись от достигнутых успехов, мы вознамерились взяться за одну из таких систем.

Вскоре эти «прожекты» реализовались в виде крупной комплексной системы физического моделирования. Ее разработка длилась около шести лет. Написанная на двух языках, она не сопровождалась формальной архитектурной документацией; более того, ее проектировщики даже не планировали создавать таковую. При всем при этом главный архитектор намеревался, немного покопавшись, восстановить заложенную в системе архитектуру. Несмотря на то что в системе было всего 300 000 строк кода, ничего более сложного я в жизни не видел.

Перед тем как приступить к плотному взаимодействию с архитектором, мы достали копию кодовой базы и извлекли из нее ряд весьма полезных низкоуровневых отношений (в частности, function_calls_function и function_defines_global_variable). После этого мы заполнили соответствующими таблицами базу данных.

Затем пришел архитектор. Он сформулировал свое представление об архитектуре, мы выразили его догадки в виде ряда запросов SQL, подали их в базу данных и визуализировали результат. Получилось черт знает что с тысячами неклассифицированных элементов и не меньшим количеством вездесущих отношений. Посмотрев на все это, архитектор поразмыслил и предложил другое решение. Мы вновь засели за составление запросов SQL, соответствующим образом перестроили структуру базы данных и вывели результат. И опять ничего не вышло.

Занимались мы этим до позднего вечера и весь следующий день. В конце концов мы нашли архитектуру, которая, как здраво рассудил архитектор, оказалась вполне приемлемой, что, впрочем, не избавило ее от порядочной неразберихи.

К чему я это говорю? Во-первых, первоначальные предположения о строении архитектуры нередко оказываются ошибочными. Добраться до разумной структуры с первого раза удается редко. Во-вторых, если продукт изначально создавался без осознания его четкой архитектуры, скорее всего, ее не удастся создать и впоследствии. Бесполезно пытаться «поймать архитектуру», которой нет.

— RK

10.6. Пример

Процесс реконструкции мы проиллюстрируем типичным набором сегментов кода, составленных инструментарием Dali при восстановлении архитектуры UCMEdit — системы формирования и редактирования use case карт по Буру. Мы покажем, за счет чего специалисту, проводившему реконструкцию, удалось на основе необработанных данных, относящихся к нескольким извлеченным представлениям, составить простую и в то же время четкую картину программной архитектуры.

Извлечение информации

В табл. 10.2 представлены элементы и отношения, которые на первом этапе реконструкции удалось извлечь из исходного кода UCMEdit. Обращения к переменным — иными словами, отношения function_reads_variable и function_assigns_variable — здесь *не* упоминаются. С другой стороны, поскольку эти отношения потенциально важны в контексте установления архитектурного сцепления, для их фиксации была проведена вторая операция извлечения. Для извлечения отношений file depends_on file проведена обработка выходных данных утилиты GNU make, запущенной со сборочным файлом приложения.

После извлечения необходимых представлений фильтруются «неинтересные» функции — в частности, встроенные функции наподобие return и стандартные функции из библиотек C, такие как scanf и printf.

Создание базы данных

Теперь извлеченными отношениями наполняется база данных SQL. Как мы уже говорили в разделе 10.3, в целях каталогизации элементов и отношений в базе данных создаются две добавочные таблицы: в одной идентифицируются все определенные элементы, а в другой перечисляются все идентифицированные типы отношений. В одном из полей таблицы элементов (под именем type) хранится информация о типе элемента (файл, функция и т. д.).

Таблица 10.2. Элементы и отношения, извлеченные из UCMEdit

Отношение	Исходный элемент		Целевой элемент	
	Тип элемента	Имя элемента	Тип элемента	Имя элемента
calls	Функция	tCaller	Функция	tCallee
contains	Файл	tContainer	Функция	tContainee
defines	Файл	tFile	Класс	tClass
has_subclass	Класс	tSuperclass	Класс	tSubclass
has_friend	Класс	tClass	Класс	tFriend
defined_fn	Класс	tDefined_by	Функция	tDefines
has_member	Класс	tClass	Переменная-член	tMember
defines_var	Функция	tDefiner	Локальная переменная	tVariable
has_instance	Класс	tClass	Переменная	tVariable
defines_global	Файл	tDefiner	Глобальная переменная	tVariable

Объединение представлений и реконструкция

На рис. 10.9 показана необработанная извлеченная модель, состоящая из 830 узлов (элементов) и 2507 отношений. Теперь наша первоочередная задача — попытаться как-то структурировать весь этот хаос, то есть приступить к исполнению сегментов кода.

Надежнее всего сначала сгруппировать функцию и все определяемые в ней локальные переменные в новый составной элемент. После исполнения показанного в листинге 10.2 сегмента кода модель UCMEdit все еще являет собой странного вида паутину из узлов и ребер, но она уже заметно проще, чем извлеченные представления (см. рис. 10.9) до исполнения сегментов кода группировки функций. Теперь в модели UCMEdit 710 узлов и 2321 отношение.

Как мы знаем, UCMEdit является объектно-ориентированной системой; этим знанием следует воспользоваться при исполнении следующего низкоуровневого сегмента кода. Схожий по своему характеру с сегментом свернутых функций, этот сегмент кода сводит воедино классы, их переменные-члены и функции-члены, отображая их в виде единого узла класса. Модель, полученная по результатам его исполнения, показана на рис. 10.4; в ней всего 233 узла и 518 ребер — это значительное визуальное упрощение, однако обработка модели в такой форме все еще непрактична.

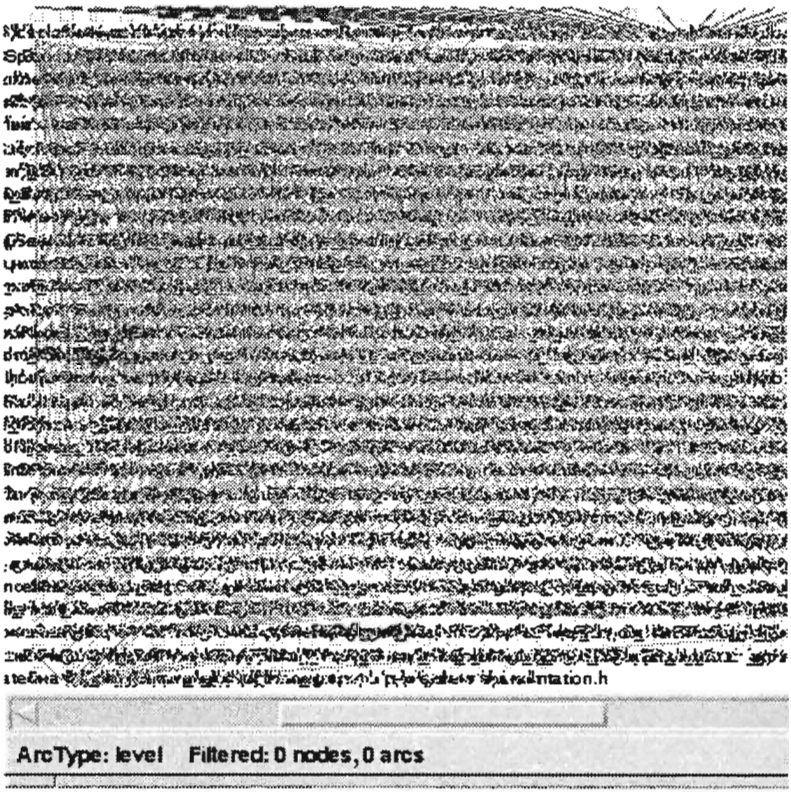


Рис. 10.9. Необработанный набор извлеченных элементов и отношений — белый шум

Элементов, не относящихся к какому бы то ни было извлеченному классу, до сих пор хватает. Причин для этого может быть две: дефект задействованных средств извлечения или отклонение решений от классического объектно-ориентированного проектирования. В данном случае справедливо и то и другое.

По результатам детального анализа выясняем, что ошибки исходят от сегментов кода извлечения, — вызовы функций-членов они выдают за вызовы глобальных функций. Кроме того, ряд функций, которые действительно являются глобальными, не обнаруживают принадлежности ни к одному из определенных в системе классов. Некоторые глобальные функции, выраженные в форме системных вызовов или примитивов системы управления окнами, естественно, необходимы. О том, как подобного рода «остаточные» явления отделить от остальной архитектуры, мы и поговорим далее.

На данном этапе модель UCMEdit представляет собой совокупность файлов, классов, остаточных функций и глобальных переменных. Локальные переменные сгруппированы с определяющими их функциями, а функции-члены и переменные-члены — с соответствующими классами. Глобальные переменные и функции теперь можно компоновать в файлы, в которых они определены, — делается это примерно так же, как компоновались функции и классы. В показанной на рис. 10.10 результирующей модели осталось три группы элементов: файлы, клас-

сы и остаточные функции. Прогресс в визуальном отображении опять налицо, но до истинного удобства манипулирования еще далеко.

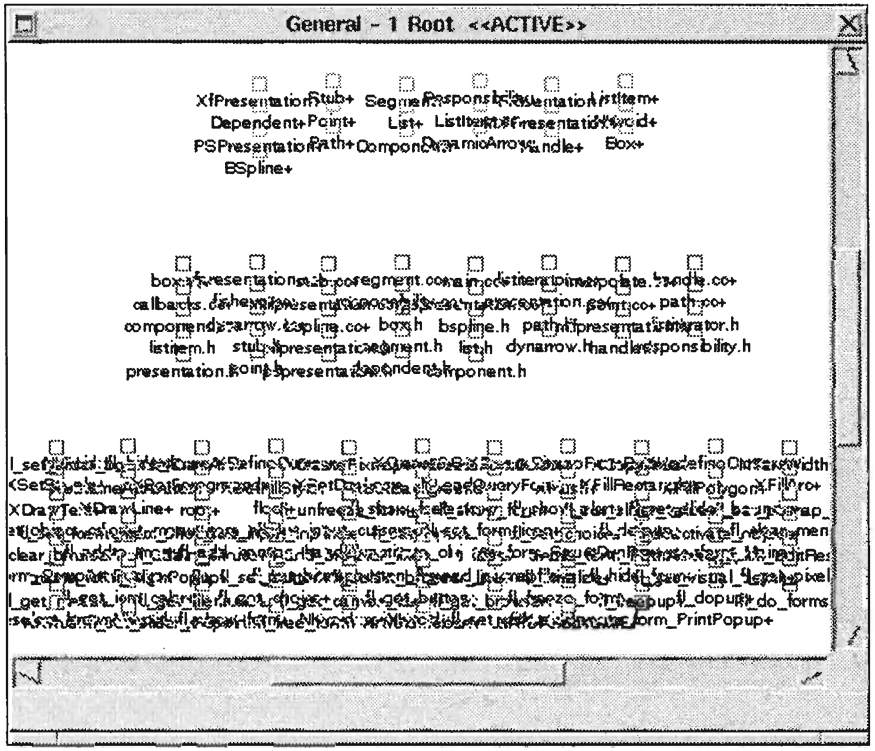


Рис. 10.10. Модель UCMEdit, состоящая из (сверху вниз) классов, файлов и «остаточных» функций (ребра не показаны)

Все исполненные до настоящего момента сегменты кода, с одной стороны, независимы от конкретного применения, но с другой — связаны со специализированными методиками извлечения и предметной областью программ C++. В следующих наборах сегментов кода, которые нам предстоит исполнить, задействуются экспертные знания, связанные с архитектурой UCMEdit. Таким образом, процесс реконструкции отходит от механического анализа, подразумевающего применение готовых сегментов кода, в пользу методики адаптивного распознавания и определения образцов, использующей информацию об архитектуре системы, которой должны обладать ее проектировщик и опытные программисты.

Специализированные знания, которые мы применяем к рассматриваемой системе в первую очередь, можно сформулировать следующим образом.

- Это интерактивное графическое приложение.
- В ней реализуется попытка инкапсулировать обращения к нижележащей подсистеме управления окнами и графикой в рамках одного уровня.
- Функции, входящие в задействованные графические библиотеки (Xlib, XForms и Mesa), придерживаются характерных соглашений по присваиванию имен.

Исходя из этих наблюдений разумно предположить существование архитектурных образцов — возможно, подсистем или отдельных образцов взаимодействия. По большому счету, эти предположения носят характер гипотез, для проверки которых мы должны убедиться в наличии или отсутствии образцов. Если результат упростит картину и окажется в согласии с нашими предположениями, значит, гипотеза верна и мы угадали ход мыслей архитектора. Даже если окажется, что мы заблуждались, в нашем распоряжении будет справедливая и полезная посылка к пониманию системы.

В представленных в листинге 10.5 сегментах кода, которые ориентированы на выявление графической подсистемы, внешние функции дополняют приложение функциональностью, связанной с отображением и взаимодействием. Рассмотрим первый сегмент кода — отфильтровывая все функции, являющиеся членами классов (тех, что выражены в виде поля tDefines в кортеже отношения defines_fn), он конструирует из таблицы elements новую таблицу. Затем он выбирает из новой таблицы все те функции, которые вызываются функциями, определенными в подклассах класса Presentation. Интересно, что этот сегмент кода ссылается на подклассы Presentation. Таким образом, он скрыто идентифицирует уровень, который, по мысли проектировщиков исходной системы, должен был инкапсулировать обращения к графической подсистеме. Эту информацию необходимо использовать в наших целях. Второй, третий и четвертый сегменты кода — именно в такой последовательности, — специфицируя сами себя именами функций, идентифицируют функции, определенные в библиотеках Mesa, XForms и Xlib соответственно.

Листинг 10.5. Сегменты кода графической подсистемы UCMEdit

```
# 1: Выявление вызовов с уровня доступа к графическим данным.
    DROP TABLE tmp;
    SELECT * INTO TABLE tmp
        FROM elements;
    DELETE FROM tmp
        WHERE tmp.tName=defines_fn.tDefines;
    SELECT t1.tName
        FROM tmp t1, calls c1, defines_fn d1,
            has_subclass s1, has_subclass s2
        WHERE t1.tName=c1.tCallee AND c1.tCaller=d1.tDefines
    AND d1.tDefined_by=s1.tSubclass
    AND s1.tSuperclass='Presentation';

print "Graphics $fields[0]+ null\n";

# 2: Выявление вызовов функций Mesa.
SELECT tName
        FROM elements
        WHERE tType='Function' AND tName LIKE 'gl%';

print "Graphics $fields[0]+ null\n";

# 3: Выявление вызовов функций XForms.
SELECT tName
        FROM elements
        WHERE tType='Function' AND tName LIKE 'fl_%';

print "Graphics $fields[0]+ null\n";
```

```
# 4: Выявление вызовов функций Xlib.
DROP TABLE tmp;
    SELECT * INTO TABLE tmp
        FROM elements;
    DELETE FROM tmp
        WHERE tmp.tName=defines_fn.tDefines;
    SELECT c1.tName
        FROM tmp c1
        WHERE tType='Function'
        AND tName LIKE 'X%';

print "Graphics $fields[0]+ null\n";
```

Взятые в целом, сегменты кода 2, 3 и 4 идентифицируют архитектурный элемент Graphics, который не прослеживается среди извлеченной информации, но в то же время присутствует в проектной архитектуре. Это хороший пример сопоставления реализованного и проектного вариантов архитектуры, проводящегося путем последовательного исполнения сегментов кода. Его результаты применительно к модели UCMEdit показаны на рис. 10.11.

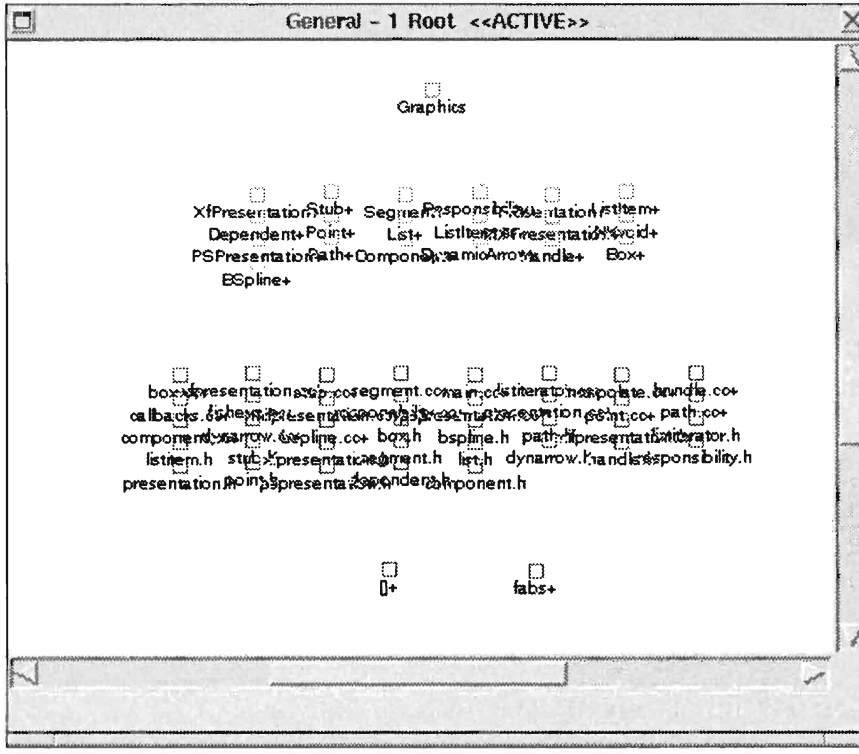


Рис. 10.11. Модель UCMEdit, состоящая из графической подсистемы, классов, файлов и оставшихся функций (ребра не показаны)

Обратите внимание: имена элементов, группируемых в рамках архитектурного элемента Graphics, содержат символ «+», присоединенный представленными на рисунке сегментами кода. Согласно этой методике, обращение к ранее

сконструированным составным элементам производится без подачи со стороны сегментов кода явного запроса в базу данных.

На рис. 10.11 всего две остаточные функции: fabs и []; последняя, очевидно, появилась в результате ошибки при извлечении, а вот первая — это функция математической библиотеки, которую следовало отфильтровать раньше, вместе со стандартными функциями библиотек C и встроенными функциями. Как бы то ни было, ни одна из них не представляет для нас интереса, а значит, их можно безболезненно удалить из модели.

Распределение функций по категориям «интересные» и «неинтересные» зависит от задач реконструкции. Если специалиста по реконструкции интересует конкретный аспект системы — например, зависимость подсистем от библиотек для конкретной платформы или операционной системы, вряд ли он будет удалять упомянутые функции из конкретной модели; более вероятно, что он сгруппирует их на отдельном уровне, с тем чтобы проанализировать их применение остальной частью приложения. Но наша задача заключается в том, чтобы построить архитектурное отображение прикладной части системы, и поэтому мы можем себе позволить от них избавиться.

Второй общий прикладной сегмент кода основывается на знаниях об отношении между классами и файлами в рассматриваемых приложениях. Во-первых, в исходном файле (.cc) не могут содержаться функции, относящиеся к нескольким классам; во-вторых, в заголовочном файле (.h) не может быть определений для нескольких классов. Отсюда возможность установить уникальное отношение вложенности: любой класс состоит из заголовочного файла, в котором содержится его определение, и исходного файла, определяющего его функции. Сегмент кода, генерирующий эти группировки, показан в листинге 10.6

Листинг 10.6. Сегменты кода, направленные на установление вложенности классов/файлов

```
SELECT DISTINCT tDefined_by
    FROM defines_fn;

print "$fields[0]+ $fields[0]+ Class $fields[0]++\n";

SELECT DISTINCT d1.tDefined_by, c1.tContainer
    FROM defines_fn d1, contains c1
        WHERE c1.tContainee=d1.tDefines;

print "$fields[0]+ $fields[1]+ Class\n";

SELECT d1.tClass, d1.tFile
    FROM defines d1;

print "$fields[0]+ $fields[1] Class\n";
```

Тот же пример иллюстрирует дополнительную характеристику подобных спецификаций — последнее поле в выражении perl, ассоциированном с первым сегментом кода ($fields[0]++), устанавливает переименование группируемого элемента. В этом сегменте кода происходит группировка классов в составные элементы (наличие в их именах замыкающих символов «+» связано с рассматриваемыми в разделе 10.4 сегментами кода для свертывания классов). Они получают имена

<class>+; исходные композиты классов при этом переименовываются в <class>++. Результат показан на рис. 10.12.

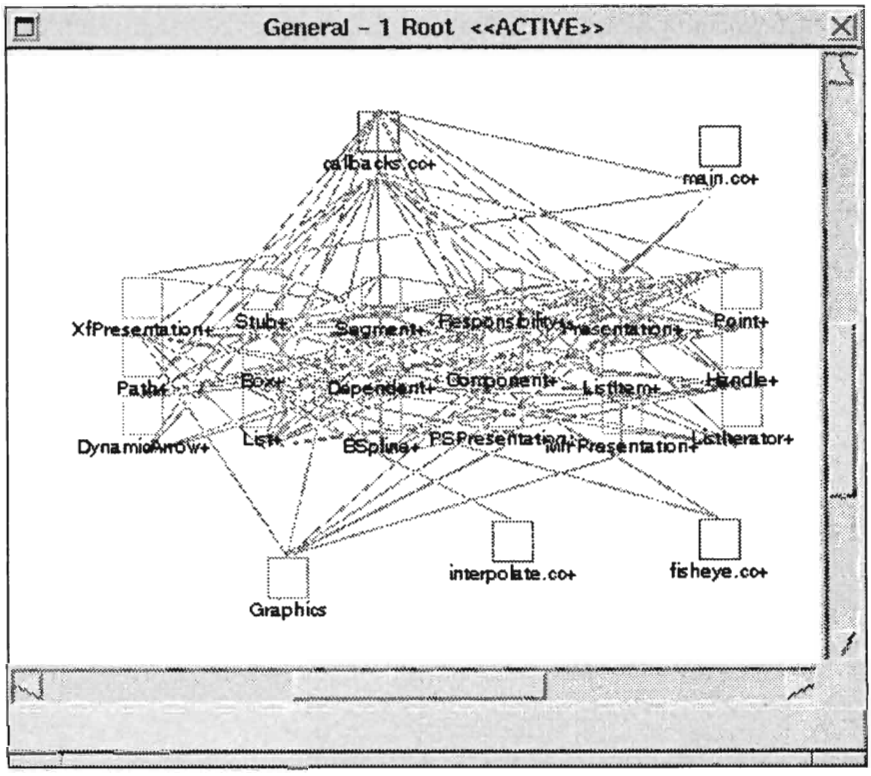


Рис. 10.12. Модель UCMEdit после применения общих сегментов кода

UCMEdit замышлялась как макетная система, демонстрирующая преимущества компьютерного редактирования use case карт. Поскольку «реберное» архитектурное решение этого приложения в начале разработки не рассматривалось, идентификацию архитектурных элементов из конкретной модели приходится проводить исходя из структуры приложения на момент завершения разработки. Методом непосредственного манипулирования, изложенным ниже, мы постараемся экстраполировать наше понимание приложения на его модель.

Во-первых, нам известно (эти знания подтверждаются по результатам наблюдения за моделью), что центральным элементом в структуре приложения является callbacks.cc, — в нем содержатся все обработчики событий системы и большая часть реализации пользовательского интерфейса. Во-вторых, наблюдаются явные взаимосвязи между двумя оставшимися файлами и классами, с которыми они соединены: interpolate.cc ассоциируется исключительно с BSpline, а к fisheye.cc обращаются только Box и Component. В-третьих, мы можем еще раз задействовать знания о структуре уровня инкапсуляции графики, или *представления* (presentation); он, как известно, выражается в классе Presentation и его подклассах. В-четвертых, по нашим наблюдениям, между классами List, ListItem и ListIterator существует

функциональная связь, и, кроме того, к ним обращаются почти все остальные классы.

Все эти наблюдения реализуются путем:

- идентификации файла callbacks.cc с архитектурным элементом Interaction;
- агрегирования файла interpolate.cc в элемент BSpline;
- агрегирования класса Presentation и его подклассов в элемент Presentation;
- агрегирования классов List, ListItem и ListIterator в элемент List, его сокрытия и трактовки в качестве «обслуживающего уровня».

Результаты внесения в модель всех этих изменений показаны на рис. 10.13.

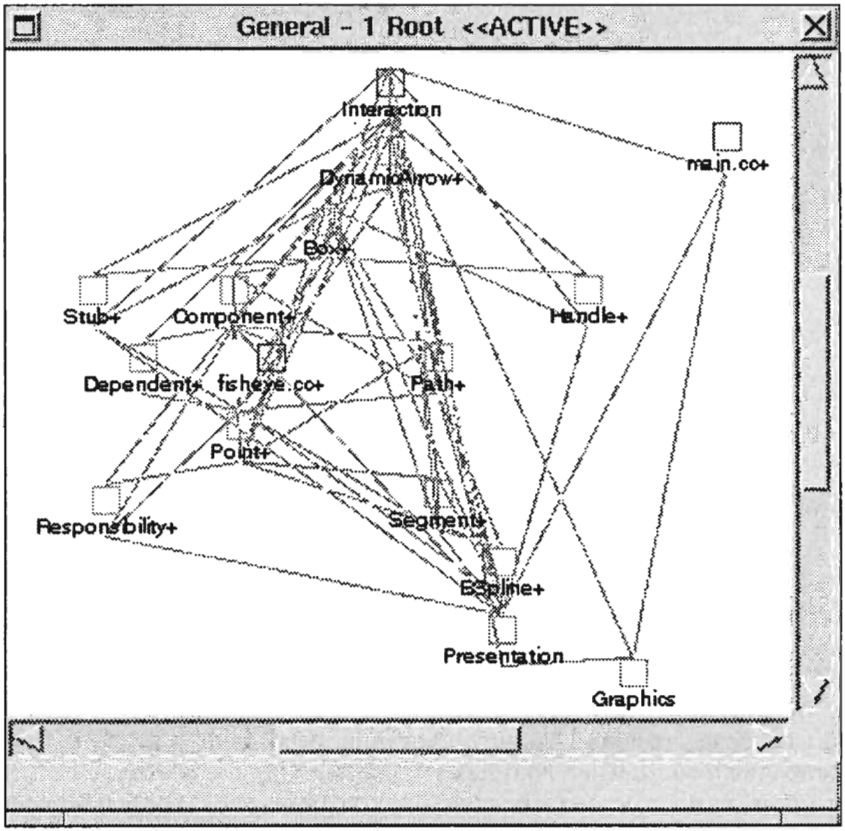


Рис. 10.13. Модель UCMEdit после применения прикладной методики непосредственного манипулирования

На данном этапе следует тщательно обдумать варианты дальнейшего упрощения модели. Автоматическая кластеризация на основе теоретико-графовых свойств наподобие силы связей в нашем случае бесполезна. Есть и другой вариант — попытаться построить уровни исходя из организации, сгенерированной алгоритмом компоновки графа (рис. 10.13), однако в этом случае функциональная согласованность в рамках уровней оставляет желать лучшего. Другими словами, обе

выдвинутые гипотезы не подтверждаются системой, и поэтому мы не настаиваем на их правомерности. Однако, с оглядкой на предметную область use case карт, мы возьмемся выдвинуть еще одну гипотезу.

Проанализировав понятия в use case картах, мы обнаружили, что элементы делятся на две широкие категории: одни связаны с компонентами, а другие — с путями, и именно эти две основные конструкции составляют use case карту. DynamicArrow, Path, Point, Responsibility, Segment, Stub и BSpline связаны с путями; Box, Component, Dependent, Handle и fisheye.cc — с компонентами. Результат кластеризации этих элементов на два архитектурных элемента — Path и Component — показан на рис. 10.14.

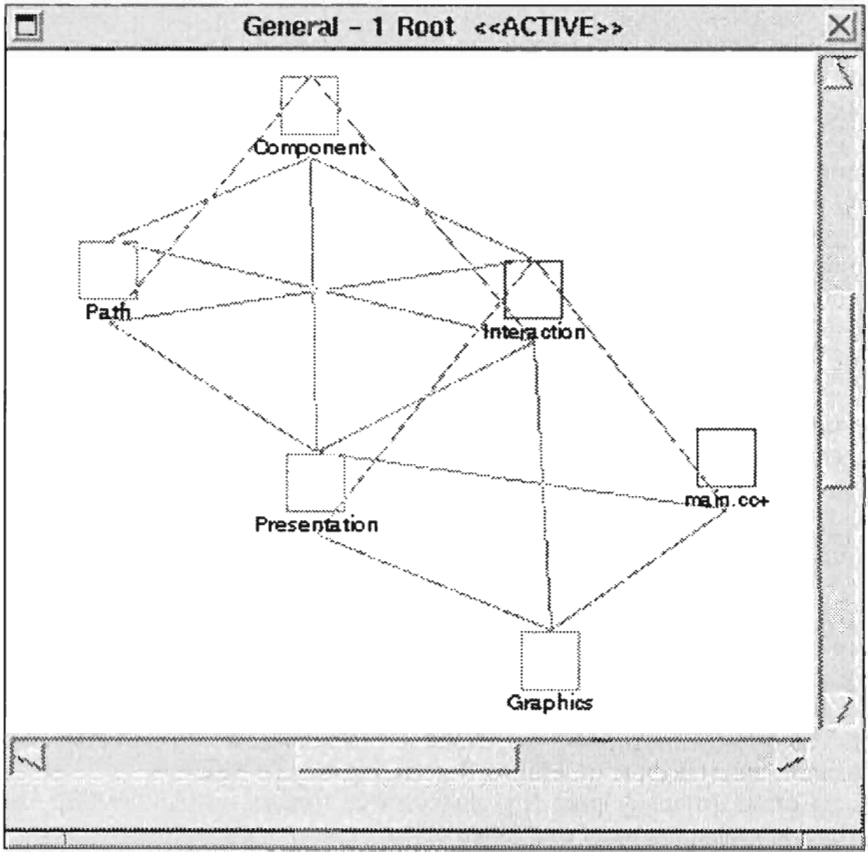


Рис. 10.14. Модель UCMEdit после кластеризации на основе прикладной области

По результатам проверки соединений между элементами все еще обнаруживается большое количество взаимосвязей. В принципе, ничего страшного, однако это обстоятельство наводит на мысль о том, что архитектуре UCMEdit не хватает функциональной согласованности в рамках элементов и их соединений.

К сожалению, никаких более значительных усовершенствований в модель UCMEdit нам внести не удастся. У этой системы есть такой недостаток, как сложность

отображения функциональности на программную структуру. Отсюда невозможность абстрагирования функционально связных элементов высокого уровня в рамках архитектуры.

С другой стороны, накопленные знания позволяют сделать ряд предложений по усовершенствованию архитектуры и составить документацию. Последняя возможность представляет особую ценность, поскольку, как мы выяснили, системе UCMEdit не хватает концептуальной целостности, а это зачастую выводит на первый план интуитивное понимание.

10.7. Заключение

Неосязаемость архитектуры во многих случаях приводит к тому, что на протяжении жизненного цикла системы она утрачивается или разрушается. Именно с этим связана потребность в методиках восстановления и извлечения вариантов архитектуры из унаследованных систем. Мы в настоящей главе представили обзор типичного процесса архитектурной реконструкции и весьма подробно изложили пример его проведения.

Нетривиальность отображения архитектуры на артефакты системы уровня исходного кода значительно усложняет архитектурную реконструкцию; для того чтобы облегчить ее проведение, желательно привлекать к процессу специалистов, обладающих солидными знаниями о рассматриваемой системе. Нисколько не умаляя важности инструментальных средств — особенно в тех случаях, когда они сочетаются в рамках компактного инструментария, — мы все же отметим, что в контексте реконструкции человеческие знания имеют поистине непреходящую ценность.

10.8. Дополнительная литература

В настоящее время существует несколько инструментариев реконструкции. В Институте программной инженерии (Software Engineering Institute, SEI) разработан инструментарий Dali [Kazman 99a]. Также следует упомянуть продукт разработки Sneed [Sneed 98], фабрики программного обновления, созданные Вероефом (Chris Verhoef) и его коллегами [Brand 97], а также инструментальный набор воссоздания архитектуры от Philips Research [Krikhaar 99].

Обзор стандартной формы Rigi содержится в издании [Muller 93]. Инструмент Rigi описывается в работе [Wong 94].

В труде [Bowman 99] изложен метод извлечения архитектурной документации из кода реализованной системы, сильно напоминающий Dali. В частности, там приводится пример реконструкции архитектуры системы Linux, в ходе которой путем анализа исходного кода программой cfx (это инструмент быстрого извлечения кода C) была получена символьная информация, на основе которой впоследствии провели генерацию набора отношений между символами. Затем последовала ручная древовидная декомпозиция системы Linux на подсистемы и назначение им исходных файлов. Далее при помощи утилиты обработки су-

щественных фактов были установлены отношения между идентифицированными подсистемами, а посредством инструмента визуализации Isedit проведена визуализация извлеченной структуры системы. Уточнить результирующую структуру удалось путем перемещения исходных файлов из одной подсистемы в другую.

Харрис (Harris) и ряд его коллег предлагают совместить в рамках каркаса архитектурной реконструкции восходящий и нисходящий подходы [Harris 95]. Каркас этот включает три части: механизм отображения архитектуры, машину распознавания исходного кода вместе со вспомогательной библиотекой запросов на распознавание, а также механизм «панорамного» обзора программы. При восходящем анализе панорамное представление позволяет отображать файловую структуру и исходные элементы системы, а также реорганизовывать информацию в рамках более содержательных кластеров. В нисходящем анализе для выявления тех элементов, которые планируется обнаружить в программной системе, используются отдельные архитектурные образцы. Запросы на распознавание помогают проводить проверку на предмет существования ожидаемых элементов.

В работе [Guo 99] приводится обзор полуавтоматического метода восстановления архитектуры под названием ARM, подходящего, правда, только для тех систем, которые проектировались и разрабатывались с помощью образцов. Состоит он из четырех основных этапов: 1) разработка конкретного плана распознавания конкретных образцов; 2) извлечение исходной модели; 3) обнаружение и оценка экземпляров образцов; 4) реконструкция и анализ архитектуры. Параллельно с описанием метода ARM в издании приводятся конкретные примеры его применения в целях реконструкции систем и проверки их соответствия документированным вариантам архитектуры.

10.9. Дискуссионные вопросы

1. Предположим, что, по вашему мнению, у реконструируемой системы многоуровневая архитектура. Какого рода информацией, извлеченной из исходного кода, эту гипотезу можно подтвердить или опровергнуть?
2. Предположим, что, по вашему мнению, архитектура реконструируемой системы соответствует схеме совместно используемого репозитария. Какой информацией, извлеченной из исходного кода, эту гипотезу можно подтвердить или опровергнуть?
3. Укажите архитектурные представления, которые следует реконструировать, для каждого из перечисленных в разделе 10.1 вариантов применения реконструкции.
4. В главе 6 описывается кодовый шаблон, который предусматривает последовательное обеспечение высокой готовности в масштабах всей системы управления воздушным движением ISSS. Предположим, что вы решили проверить разработчиков и специалистов по сопровождению на предмет следования этому шаблону в различные периоды жизненного цикла системы. Опишите процесс реконструкции, который вам в таком случае пришлось бы провести.

Часть 3
АНАЛИЗ АРХИТЕКТУРЫ

В результате анализа архитектурно-экономического цикла мы подошли к этапу, на котором в распоряжении архитектора имеется спроектированная и документированная архитектура. Логически это подводит нас к проблеме оценки и анализа архитектуры — тех действий, по результатам которых принимается решение об адекватности или неадекватности выработанного решения поставленной задаче. Именно об этом пойдет речь в третьей части книги, предварить которую, по нашему мнению, следует ответами на ряд вопросов об архитектурной оценке: зачем нужно ее проводить, когда это необходимо делать, каковы связанные с ней затраты (стоимость) и выгоды, следует ли ее планировать, какие требуются предварительные условия и что должны собой представлять ее результаты.

Зачем?

У любой архитектуры есть одно замечательное качество — она содержит сведения о наиважнейших свойствах системы, даже если эта система еще не существует. Принимая проектные решения, архитекторы отталкиваются от известных и предсказуемых нисходящих последствий, проявляющихся в ходе конструирования систем(ы). Без этого толку от процесса создания архитектуры было бы не больше, чем от кидания костей. Мы бы выбирали произвольную архитектуру, строили на ее основе систему, проверяли, реализовались ли в ней требуемые свойства, и, не встретив таковых, возвращались к началу. Готовых рецептов по проектированию архитектуры никто еще не придумал, но в том, что методология продвинулась гораздо дальше примитивного угадывания, можно не сомневаться.

В общем и целом, архитекторы представляют себе последствия принимаемых решений. На материале главы 5 мы можем сделать вывод о том, что архитектурные тактики и образцы наделяют системы известными свойствами. Следовательно, проектные альтернативы — другими словами, варианты архитектуры — поддаются анализу. Располагая архитектурой, можно делать выводы о системе — даже если до ее создания дело еще не дошло.

Так зачем же архитектуру нужно оценивать? Во-первых, от нее слишком многое зависит, а во-вторых, ничто не мешает вам это сделать. Эффективная методика оценки предполагаемой архитектуры, позволяющая сделать определенные

выводы *до* того, как эта архитектура будет утверждена в рамках детального плана проекта, представляет значительную ценность с экономической точки зрения. С появлением повторяемых, структурированных методов (таких, как, например, ATAM, рассматриваемый в главе 11) оценка архитектуры превратилась в относительно недорогой механизм смягчения рисков. В конце концов, убедиться в том, что архитектура соответствует поставленным задачам, требует здравый смысл. *Итак, оценка архитектуры должна стать стандартным компонентом любой методики архитектурно-ориентированной разработки.*

Когда?

Оценивать качество программного продукта разумно на самых ранних стадиях его жизненного цикла. Чем раньше обнаруживается проблема, тем легче ее устранить — для этого достаточно лишь внести соответствующие изменения в требование, спецификацию или проект. Приступать к обеспечению качества на поздних стадиях работы над проектом недопустимо — качество должно быть присуще проекту с момента его появления. Оценивать предполагаемые решения (и при необходимости отклонять их) следует еще на этапе проектирования, до перехода к долгосрочной институционализации.

С другой стороны, оценку архитектуры можно проводить многократно — на разных этапах жизненного цикла системы. Если архитектура находится в зачаточном состоянии, принятые решения следует оценивать наравне с теми, которые еще только обсуждаются. Выбирайте из множества архитектурных альтернатив наиболее подходящие варианты. Если архитектура готова или близка к состоянию готовности, проверьте ее, не дожидаясь, пока проект перейдет в стадию длительной и дорогостоящей разработки. Оценка будет весьма уместной, если вам придется иметь дело с существующей системой, представленной к модификации, переносу, интеграции с другими системами и любым прочим значимым мероприятиям по модернизации. Наконец, оценка архитектуры очень информативна: довольно часто при разработке новых проектов требуется разобраться в том, каким образом унаследованная система удовлетворяет (и удовлетворяет ли?) предъявленные к ней требования по атрибутам качества. Более того, принимая решение о *приобретении* крупной программной системы с предположительно длительным сроком службы, компания-покупатель обязательно должна разобраться в ее архитектуре. В результате ее оценки становится понятно, насколько система пригодна с точки зрения реализации ею важнейших атрибутов качества.

Наконец, оценка помогает выбрать один из двух вариантов архитектуры — тот, что в наибольшей степени соответствует критериям «добротности».

Затраты

Затраты (стоимость) оценки выражаются во времени, затраченном на эту процедуру специалистами. Имея опыт критического анализа приблизительно 300 полноценных вариантов архитектуры для проектов, на разработку которых уходит как минимум 700 человеко-дней, компания AT&T, по данным своих руководителей проектов, называет средние затраты на оценку — 70 человеко-дней. Проведе-

ние критического анализа по методике ATAM занимает примерно 36 человеко-дней[1]. Если в вашей компании на постоянной основе существует подразделение, занимающееся исключительно оценкой, при расчете затрат следует учитывать стоимость его содержания, а также временные затраты на обучение сотрудников.

Выгоды

Нам известны шесть видов преимуществ, связанных с обследованием архитектуры.

1. *Финансовые преимущества.* В AT&T руководитель каждого проекта обязан сделать приблизительный расчет средств, которые можно сэкономить путем оценки архитектуры. Согласно средним за восемь лет деятельности показателям, комплексная архитектурная оценка любого проекта уменьшает стоимость проекта на 10 %. Учитывая то, что на оценку уходит в среднем 70 человеко-дней, окупается оценка тех проектов, работа над которыми продолжается 700 человеко-дней и дольше.

   Мы не располагаем сопоставимыми количественными данными других компаний. Однако, по словам некоторых консультантов, более 80 % их работы — это повторение уже выполненных задач. Их клиенты довольно быстро осознали, что дополнительное проведение оценки просто выгодно.

   Случаев, когда в результате оценки заказчики смогли избавить себя от лишних трат, известно множество. Рассказывают, например, что одна крупная компания отказалась от многомиллионного контракта, узнав, что архитектура глобальной информационной системы, которую она планировала приобрести, не способна реализовать желаемые атрибуты качества. Проведенный на раннем этапе архитектурный анализ электронной системы перечисления денежных средств показал, что предоставляемая ею суточная емкость в $50 миллиардов в два раза меньше предполагаемой. Оценка системы розничной торговли выявила проблемы производительности самого высокого порядка, справиться с которыми не смогли бы даже самые совершенные аппаратные средства. Таким образом, заказчик сумел предотвратить собственное банкротство. И так далее.

   Рассказывают также и о том, к каким неприятностям приводит игнорирование процедуры оценки. В одном таком случае пересмотр системы учета клиентов планировалось провести за два года, однако по прошествии семи лет она претерпела целых три повторных реализации. Несмотря на то что вычислительная мощность в последней версии в 60 раз превосходила характеристики первоначального макета, удовлетворить требования по производительности так и не удалось. В другом случае проблемы с производительностью, обусловленные непродуманными проектными решениями, сделали невозможным компонентное тестирование. В результате проект — разработку крупной инженерной системы реляционных баз данных — свернули. Обошелся этот грандиозный успех в $20 миллионов.

[1] Этот показатель учитывает только работу группы оценщиков. Поскольку методика ATAM предполагает участие в оценке заинтересованных лиц и ответственных лиц, общие затраты увеличиваются.

2. *Вынужденная подготовка к критическому анализу.* Специалисты, на которых возлагается обязанность по оценке архитектуры, должны, во-первых, знать, на что следует обращать внимание, а во-вторых, предварительно составить некое описание архитектуры. Следовательно, им приходится документировать архитектуру. Варианты архитектуры, доступные для понимания всем разработчикам, встречаются не так уж часто. Составленные описания зачастую оказываются либо слишком краткими, либо (что случается чаще) слишком подробными и растягиваются на многие тысячи страниц. Вполне возможно недопонимание разработчиками тех или иных допущений, касающихся выделенных им элементов. Процесс подготовки к оценке способствует решению многих из этих проблем.
3. *Фиксация логического обоснования.* В процессе оценки архитектуры внимание всегда заостряется на нескольких областях; перед оценщиками ставятся вполне конкретные вопросы. Для того чтобы на эти вопросы можно было ответить, часто требуется получить представление о проектных решениях и соответствующих логических обоснованиях. Документация логического обоснования проекта оказывается востребованной на дальнейших стадиях жизненного цикла, помогая оценить эффект модификаций. Задокументировать логическое обоснование постфактум невероятно сложно. Его фиксация по результатам оценки архитектуры, таким образом, приносит немалую пользу впоследствии.
4. *Своевременное выявление недостатков существующей архитектуры.* Чем раньше проблемы обнаруживаются, тем дешевле их исправить. В частности, по результатам оценки архитектуры возможно выявление проблем, связанных с необоснованными (или слишком дорогостоящими в части реализации) требованиями, производительностью или потенциальными нисходящими модификациями. Оценка сценариев модификации системы способствует обнаружению трудности переносимости и расширяемости. Таким образом, оценка архитектуры помогает составить представление о возможностях и ограничениях продукта на ранней стадии его проектирования.
5. *Проверка требований.* Проверка соответствия архитектуры предъявляемым к ней требованиям перетекает в анализ требований как таковых. Это помогает лучше уяснить их суть и (как правило) расставить приоритеты. Процесс формулирования требований, проводящийся в отрыве от ранних стадий проектирования, обычно приводит к конфликту свойств системы. Высокая производительность, безопасность, отказоустойчивость, низкие затраты — всего этого легко потребовать, но крайне проблематично (а иногда и невозможно) достичь. Оценка архитектуры вскрывает конфликты, указывает на возможные компромиссные решения и создает условия для совместного обсуждения возникших проблем.
6. *Усовершенствование архитектуры.* Компании, встроившие оценку архитектуры в стандартный процесс разработки, отмечают повышение качества результата. Предугадывая возможные вопросы и предметы для обсуждения, зная, какая документация потребуется для оценки архитектуры, компании-разработчики естественным образом увеличивают эффективность

проведения этой процедуры. Результатом становится повышение качества архитектуры, наблюдаемое не только после оценки, но, по существу, и до ее проведения. Со временем компания вырабатывает культуру качественного архитектурного проектирования.

Итак, оценка архитектуры способствует повышению качества, снижению затрат и бюджетных рисков. Являясь основой для принятия любых технологических решений, архитектура оказывает огромное влияние на стоимость и качество продукта. Сама по себе оценка не гарантирует высокого качества или низких затрат, но она явственно обозначает области риска. Конечная стоимость и качество системы складывюется из совокупности факторов, включая тестирование, адекватность документации и кодирование.

Методики

Рассматриваемые в двух последующих главах методы ATAM и CBAM являют собой примеры *вопросных методик* (questioning techniques). Они предполагают формулирование испытательных вопросов согласно сценариям и проверку реакции оцениваемой архитектуры на различные ситуации. Среди других вопросных методик следует упомянуть контрольные списки и анкеты. Эффективными они оказываются в тех случаях, когда оценщикам приходится работать с однотипными системами, каждый раз организуя для них одни и те же испытания. Все вопросные методики, по сути, основываются на мысленных экспериментах, и с их помощью определяют степень соответствия той или иной архитектуры поставленным задачам.

Естественным дополнением вопросных методик являются *измерительные методики* (measuring techniques). Они опираются на количественные показатели. В качестве примера таковой можно привести архитектурные метрики. По результатам измерений сцепления в рамках архитектуры, связности ее модулей или глубины иерархии наследования можно сделать определенные выводы о модифицируемости результирующей системы. К измерительным методикам также относятся приемы, предполагающие проверку на предмет интересующих атрибутов качества (например, атрибутов периода прогона наподобие производительности или готовности) моделей и макетов.

Результаты приложения измерительных методик оказываются в некотором смысле более конкретными, чем ответы, получаемые по вопросным методикам, но, с другой стороны, они применимы только при наличии рабочего артефакта. Другими словами, измерительные методики предполагают существование объекта измерений. Вопросные методики, напротив, прекрасно приспособлены к обследованию гипотетических архитектур, а значит, могут применяться на гораздо более ранних стадиях жизненного цикла.

Планировать или не планировать?

Оценка может быть запланированной или незапланированной. Запланированная оценка считается нормальным элементом цикла разработки проекта. Время ее проведения определяется заранее, она учитывается при разработке рабочих пла-

нов и бюджета проекта и подлежит контролю по срокам. Незапланированная оценка, как правило, проводится спонтанно, являясь результатом возникновения разного род затруднений; ее организация, таким образом, призвана вытащить проект из тупика.

В лучшем случае запланированная оценка рассматривается как одно из средств реализации проекта, в худшем — как отвлечение от него. Воспринимать ее как инструмент проверки квалификации участников проекта не следует, скорее, это проверка соответствия изначально запланированному «курсу» разработки. Запланированная оценка носит предупреждающий характер и способствует сплочению коллектива.

Незапланированная проверка для участников проекта — это, напротив, настоящее испытание. Она требует выделения дополнительных временных ресурсов, которых и так не хватает. Решение о проведении незапланированной проверки принимается руководством в случае высокой вероятности неблагоприятного исхода и, таким образом, воспринимается как средство внесения промежуточных корректив. Незапланированные проверки носят реактивный характер и проводятся в довольно напряженной атмосфере. Руководитель группы оценщиков обязан не допустить выяснения отношений между своими подчиненными.

Не стоит и говорить, что запланированные оценки предпочтительнее.

Предварительные условия

Об успехе процедуры оценки можно судить по следующим критериям:

1. *Четко сформулированные задачи и требования к архитектуре.* Оценивать полезность архитектуры можно только в контексте определенных атрибутов качества. Система, отличающаяся сверхпроизводительностью, может оказаться совершенно непригодной для выполнения задач, требующих модифицируемости. Анализировать архитектуру, не разобравшись с критериями «добротности», — все равно что отправляться в путь куда глаза глядят. Иногда (хотя, судя по нашему опыту, крайне редко) эти критерии устанавливаются в форме спецификации требований. Более вероятно, что их определение придется на период перед оценкой или непосредственно на оценку. Задачи определяют цель оценки; их следует явно оговаривать в договоре об оценке и уточнять впоследствии.
2. *Поддающиеся контролю рамки.* Процедуру оценки следует ориентировать на решение определенных ясно сформулированных задач. Их количество должно быть сведено к минимуму (не более 3–5). Неспособность определить ограниченный круг высокоприоритетных задач часто свидетельствует о несбыточности ожиданий от результатов оценки (а возможно, и от системы в целом).
3. *Экономическая эффективность.* Организаторы оценки обязаны убедиться в том, что выгоды от проведения проверки (с высокой вероятностью) превысят затраты. Те виды оценки, о которых мы говорим, подходят для проектов среднего и крупного масштаба, однако их эффективность в отношении малых проектов весьма сомнительна.

4. *Наличие необходимых специалистов.* К проведению проверки совершенно необходимо привлечь архитектора или другого специалиста, способного авторитетно рассуждать об архитектуре и проектировании. Этот человек (или эти люди) должен уметь сжато и ясно изложить факты архитектуры и мотивацию принятия архитектурных решений. Если речь идет об очень крупной системе, необходимо участие проектировщиков ее основных компонентов — они должны проверить, насколько сложившееся у архитектора представление об архитектуре фактически отражено на более детальных уровнях. Проектировщики, кроме того, могут сообщить немало важного об атрибутах поведения и качества компонентов. В контексте методики ATAM следует обеспечить участие в оценке архитектуры заинтересованных лиц. Необходимо определить лиц, желающих получить отчет об оценке, установить их приоритеты и ожидания.
5. *Квалифицированность группы оценщиков.* В идеале, оценкой программной архитектуры должно заниматься специальное подразделение компании. Участники группы оценщиков должны быть беспристрастны, объективны и уважаемы. Группа должна создавать впечатление компетентности в своей области; результаты ее деятельности должны иметь определенное значение в глазах других специалистов — иначе они неизбежно будут воспринимать ее функции как бесполезные. В группе оценщиков должны работать люди, смыслящие в вопросах архитектуры, а во главе ее должен стоять специалист, имеющий серьезный опыт проектирования и оценки проектов на архитектурном уровне.
6. *Ясные ожидания.* Определяющее значение для успешного проведения оценки имеет четкое понимание ожиданий ее компании-организатора. Необходима ясность относительно задач оценки, ее предполагаемых результатов, анализируемых (и неанализируемых) областей, требующихся временных и трудовых ресурсов и заинтересованных в получении результата лиц.

Результаты

По результатам оценки следует составить отчет, охарактеризовать в нем все выявленные проблемы и подкрепить выводы фактами. Черновик отчета следует передавать поочередно каждому из оценщиков; совместно они должны устранить из него все недоразумения, любую необъективность, сопоставить различные элементы и только после этого составить окончательный вариант. В идеале, если те или иные проблемы не решаются, их следует ранжировать по принципу потенциального влияния на проект. Также следует собирать информацию о самом процессе оценки. На основе совокупных данных о ряде оценок можно выстроить специальные методики, выработать способы обучения, наконец, усовершенствовать процессы разработки системы в целом и оценки архитектуры в частности. Необходимо анализировать затраты и выгоды проведения оценки. Информацию о предшествующих процедурах оценки компания должна использовать, во-первых, для внесения корректив в последующие подобные процедуры и, во-вторых, для составления аналитических сводок по затратам и выгодам.

В настоящей части три главы. Метод ATAM (рассматриваемый в главе 11) представляет собой структурный метод оценки архитектуры. По результатам его применения составляется список рисков несоответствия архитектуры предъявленным к ней требованиям. CBAM (глава 12) — это метод ранжирования рисков. В крупных системах зачастую оказывается слишком много рисков. Решение о том, какой из них следует сгладить в первую очередь, необходимо принимать исходя из сопоставления затрат модифицирования архитектуры, без которых этот риск не сгладить, и выгод его сглаживания. В методе CBAM предусматривается структура, направленная на решение этой организационно-экономической задачи. В главе 13 приводится очередной конкретный пример систем, ориентированных на WWW. Их развитие рассматривается как прохождение нескольких архитектурно-экономических циклов.

Дополнительная литература

Изложенный в этом введении материал взят из исследования [Abowd 96] «Recommended Best Industrial Practice for Architecture Evaluation», составленного по результатам семинаров, проведенных его авторами и другими научными сотрудниками Института программной инженерии. На них побывали представители восьми промышленных и консультативных компаний.

Методика архитектурной оценки на основе контрольных списков и анкет — одна из разновидностей *активного анализа проектного решения* (active design review) — изложена в работе [Parnas 85b]. Активная оценка проектного решения предполагает участие нескольких лиц, которые, основываясь на документации, отвечают на заранее подготовленные вопросы. Этот метод заметно отличается от случайной и неорганизованной оценки, участники которой докладывают об отклонениях по мере их обнаружения.

В издании [Cusumano 95] метрики рассматриваются как средство выявления мест, в которых с наибольшей вероятностью могут произойти изменения.

Богатый опыт по части оценки вариантов архитектуры, накопленный компанией AT&T, обобщается в работе [AT&T 93].

Глава 11
Метод анализа компромиссных архитектурных решений — комплексный подход к оценке архитектуры

(в соавторстве с Марком Кляйном)

> Об услугах, которые нам оказывают другие люди, мы судим по тому, насколько существенными они кажутся им, а на действительную их ценность как-то не обращаем внимания.
>
> *Фридрих Ницше*

Настоящая глава представляет собой обзор метода анализа компромиссных архитектурных решений (Architecture Tradeoff Analysis Method, ATAM) — комплексной универсальной методики оценки программной архитектуры. В соответствии со своим названием этот метод обнаруживает степень реализации в архитектуре тех или иных задач по качеству, а также (исходя из допущения о том, что любое архитектурное решение влияет сразу на несколько задач по качеству) механизм их взаимодействия — другими словами, их взаимозаменяемость.

Оценить архитектуру крупной системы весьма не просто. Во-первых, чем больше система, тем масштабнее ее архитектура и тем протяженнее промежуток времени, за который о ней можно составить некое представление. Во-вторых, согласно Ницше (см. эпиграф) и архитектурно-экономическому циклу (Architecture Business Cycle, ABC), любая компьютерная система призвана решать коммерческие задачи; таким образом, в ходе оценки необходимо устанавливать связи между этими задачами и техническими решениями. Наконец, крупные системы, как правило, характеризуются многочисленностью заинтересованных лиц, а для того чтобы за ограниченный промежуток времени проанализировать их позиции, процесс оценки необходимо внимательно контролировать. Итак, из всего вышесказанного понят-

но, что основной проблемой в контексте оценки архитектуры является ограниченность по времени.

Основное назначение ATAM состоит в том, чтобы выявить коммерческие задачи, поставленные в контексте разработки системы и проектирования архитектуры. Вкупе с участием заинтересованных лиц это помогает специалистам по оценке сфокусироваться на тех элементах архитектуры, которые играют первостепенную роль для реализации упомянутых задач.

В этой главе мы перечислим этапы ATAM и рассмотрим их с функциональной точки зрения. Для иллюстрации теоретического материала мы также приведем конкретный пример употребления ATAM, основанный на нашем собственном опыте.

11.1. Участники ATAM

Ниже перечислены группы специалистов, участие и сотрудничество которых является необходимым условием проведения процесса ATAM.

1. *Группа оценки.* Специалисты из этой группы не должны участвовать в разработке оцениваемой архитектуры. Как правило, численность группы составляет 3–5 человек. Каждому из них назначается ряд ролей, которые он должен исполнять в период оценки. (Описание этих ролей, а также желательные характеристики каждой из них приводятся в табл. 11.1.) В некоторых случаях группа оценки являет собой постоянное подразделение, регулярно проводящее действия по оценке вариантов архитектуры; в иных ситуациях ее участники подбираются из числа специалистов с серьезными знаниями в области архитектуры для выполнения конкретной задачи. Иногда группы оценки и разработки набираются из сотрудников одной компании, однако не исключается вариант заказа оценки у третьей стороны. В любом случае это должны быть компетентные, объективные люди без предубеждений и корыстных целей.
2. *Лица, ответственные за проект.* Полномочия участников этой группы позволяют им выполнять для проекта представительские функции и принимать решения о внесении в него изменений. Как правило, в их число входят руководитель проекта, а также (при наличии такового) заказчик, финансирующий разработку. Кроме того, речь идет об архитекторе — ведь, согласно основному принципу оценки архитектуры, архитектор должен принимать в этом процессе самое деятельное участие. Наконец, одним из представителей проекта обычно является специалист, ответственный за проведение оценки; даже если он не наделен такими полномочиями, участие его в рассматриваемой группе обязательно.
3. *Заинтересованные в архитектуре лица.* Заинтересованным лицам, естественно, хочется, чтобы архитектура не отклонялась от проекта. Способность этих людей выполнять свои задачи напрямую зависит от того, в какой степени архитектура реализует модифицируемость, безопасность, высокую надежность и прочие атрибуты качества. В число заинтересованных лиц

входят разработчики, тестировщики, сборщики, специалисты по сопровождению, инженеры по эффективности, пользователи, конструкторы систем, взаимодействующих с рассматриваемой системой, и многие другие. В ходе оценки от них требуется сформулировать конкретные задачи по реализации атрибутов качества, при выполнении которых систему можно будет признать удачной. Нормальным следует считать участие в процессе оценки 12–15 заинтересованных лиц.

Таблица 11.1. Распределение ролей между участниками группы оценки[1]

Роль	Обязанности	Предпочтительные характеристики
Руководитель группы	Планирует оценку; общается с заказчиком и гарантирует выполнение его требований; утверждает договор о проведении оценки; набирает специалистов в группу оценки; координирует составление и предоставление заказчику сводного отчета (обязанности по его составлению можно делегировать)	Организованность, управленческие способности, навыки ведения переговоров с заказчиком, выполнение обязанностей в установленные сроки
Руководитель специалистов по оценке	Ответствен за проведение оценки; содействует выявлению сценариев; координирует отбор сценариев и их классификацию согласно приоритетам; содействует процессу оценки сценариев в контексте архитектуры; содействует проведению местного анализа	Способность выступать перед аудиторией; развитые навыки содействия; компетентность в вопросах архитектуры; опыт участия в оценке вариантов архитектуры; способность отличить беспредметные прения от дискуссии, потенциально ведущей к получению ценных выводов
Секретарь по сценариям	В процессе выявления сценариев пишет сценарии на лекционном плакате или белой доске; фиксирует согласованные формулировки сценариев; не позволяет остальным участникам углубляться в дискуссии до получения точных формулировок	Хороший почерк; способность сосредоточивать специалистов на задаче формулирования идеи (сценария); понимание технической терминологии и способность резюмировать соответствующие дискуссии
Секретарь по результатам	Работая на портативном компьютере или на рабочей станции, в электронной форме фиксирует результаты: непроверенные сценарии, соображения, влияющие на их составление (во многих случаях они не отражаются в формулировке самого сценария), резолюции сценариев применительно к архитектуре; в его обязанности также входит предоставление всем участникам печатной версии списка всех принятых сценариев	Крепкие навыки быстрого клавишного набора; способность оперативно восстанавливать в памяти нужную информацию; компетентность в архитектурных вопросах; навыки быстрого усвоения технических проблем; отсутствие комплексов, позволяющее (в подходящий момент) прервать дискуссию для прояснения вопроса и фиксации нужной информации

[1] *Источник*: приводится по изданию [Clements 02a] (адаптированная версия).

Роль	Обязанности	Предпочтительные характеристики
Хронометрист	Помогает руководителю специалистов по оценке придерживаться графика; ограничивает время, уделяемое на этапе оценки каждому из сценариев	Способность в нужный момент прервать дискуссию и напомнить ее участникам о временных ограничениях
Наблюдатель за процессом	Вырабатывает способы усовершенствования или видоизменения процесса оценки; в основном остается в тени, но изредка может высказывать руководителю разумные соображения относительно процесса оценки; отчитывается о проведенном процессе и формулирует предложения на будущее; кроме того, обязан детально изложить участникам группы оценки сведения, относящиеся к накопленному ранее опыту	Внимательность при наблюдении; серьезная компетентность в вопросах оценки; опыт осуществления методики архитектурной оценки
Координатор процесса	Помогает руководителю специалистов по оценке соблюдать все этапы метода оценки	Серьезные знания по части подробностей этапов метода; способность предоставления руководителю специалистов по оценке полезных консультаций
Дознаватель	Поднимает архитектурные вопросы, о которых не подумали заинтересованные лица	Серьезные знания в области архитектуры; понимание потребностей заинтересованных лиц; опыт работы с системами в сходных предметных областях; способность поднимать дискуссионные вопросы и добиваться их рассмотрения; хорошее знание оцениваемых атрибутов качества

11.2. Результаты проведения оценки по методу ATAM

Минимальный набор результатов процесса оценки на основе ATAM выглядит следующим образом.

- *Компактная презентация архитектуры.* Согласно распространенному мнению, документация архитектуры не может обойтись без объектной модели, списка интерфейсов и их сигнатур или какого-нибудь другого объемистого перечня. Одно из требований, предъявляемых к документации методом ATAM, напротив, подразумевает возможность ее составления за один час; получившаяся в результате архитектурная презентация должна быть, таким образом, краткой и понятной (хотя как раз понятной она бывает не всегда).

- *Формулировка коммерческих задач.* Часто бывает так, что отдельные участники группы разработчиков впервые узнают о коммерческих задачах именно из презентации ATAM.
- *Требования по качеству в форме совокупности сценариев.* Коммерческие задачи подразумевают формулирование требований по качеству. Некоторые из них фиксируются в форме сценариев.
- *Отображение архитектурных решений на требования по качеству.* Архитектурные решения можно интерпретировать исходя из тех атрибутов качества, реализации которых они способствуют или препятствуют. Для каждого анализируемого по методу ATAM сценария реализации атрибута качества определяются те архитектурные решения, которые помогают его реализовать.
- *Ряд установленных точек чувствительности и компромиссов.* Так называются архитектурные решения, оказывающие заметное воздействие на один или несколько атрибутов качества. К примеру, решение о введении резервной базы данных, очевидно, относится к числу архитектурных, — оказывая положительное воздействие на надежность, в отношении этого атрибута качества оно является точкой чувствительности. С другой стороны, на сопровождение резервной базы данных расходуются системные ресурсы, а значит, это решение отрицательно сказывается на производительности. Следовательно, оно одновременно является точкой компромисса между надежностью и производительностью. Признать его рискованным или, напротив, не связанным с рисками, можно только исходя из оценки стоимости производительности в контексте предъявляемых к данной архитектуре требований по атрибутам качества.
- *Набор рискованных и нерискованных решений.* Согласно ATAM, рискованное решение — это архитектурное решение, которое может привести к нежелательным последствиям, затрудняющим реализацию сформулированных требований по атрибутам качества. Нерискованным решением, соответственно, называется архитектурное решение, которое по результатам анализа признано безопасным. Установленные риски составляют основу плана смягчения архитектурных рисков.
- *Набор магистральных рисков.* После проведения анализа группа оценки должна проверить все установленные риски на наличие магистральных рисков, свидетельствующих о систематически встречающихся в архитектуре слабых местах, недостатках архитектурного процесса или некомпетентности группы архитекторов. Если не разобраться с такого рода рисками, они способны поставить под сомнение реализацию коммерческих задач проекта.

На основе перечисленных результатов специалисты составляют окончательный письменный отчет, в котором конспектируется сам метод, резюмируются сделанные выводы, фиксируются сценарии и материалы их анализа, систематизируются полученные данные.

Есть также ряд вторичных результатов оценки. Презентации архитектуры зачастую создаются конкретно для проведения оценки и в этом случае по качеству превосходят все составленные ранее материалы. Такого рода дополнительная документация, переживая этап оценки, становится одной из составляющих на-

следия проекта. Кроме того, сценарии, создаваемые участниками оценки, как выразители коммерческих задач и требований к архитектуре можно задействовать в качестве справочного руководства при развитии архитектуры. Наконец, результаты анализа, изложенные в сводном отчете, служат логическим обоснованием определенных архитектурных решений — как принятых, так и не принятых. Итак, вторичные результаты осязаемы и перечислимы.

Следует перечислить и неосязаемые результаты оценки на основе ATAM. Среди них — формирование в среде заинтересованных лиц чувства товарищества, открытие между ними и архитектором каналов взаимодействия, а также общее, распространяющееся на всех участников повышение компетентности в вопросах данной архитектуры, осознание ее преимуществ и недостатков. Не поддающиеся измерению, эти результаты не менее важны, чем все остальные, а их воздействие зачастую оказывается наиболее продолжительным.

11.3. Этапы ATAM

Операции оценки по методу ATAM распадаются на четыре этапа.

На нулевом этапе — «Установление партнерских отношений и подготовка» — руководители группы оценки проводят неофициальные совещания с основными ответственными за проект лицами и прорабатывают подробности предстоящей работы. Представители проекта посвящают специалистов по оценке в суть проекта, тем самым повышая квалификацию некоторых из них. Эти две группы принимают согласованные логистические решения: где и когда встречаться, кто предоставит лекционные плакаты и с кого пончики и кофе. Кроме того, они согласовывают предварительный перечень заинтересованных лиц (перечисляя их не по именам, а по ролям) и устанавливают сроки и получателей сводного отчета. Они также организуют снабжение специалистов по оценке архитектурной документацией — если, конечно, таковая существует и может оказаться полезной. Наконец, руководитель группы оценки объясняет руководителю проекта и архитектору, какую информацию им следует предоставить на первом этапе, и при необходимости помогает им составить соответствующие презентации.

Таблица 11.2. Этапы ATAM и их характеристики[1]

Этап	Операции	Участники	Средняя продолжительность
0	Установление партнерских отношений и подготовка	Руководство группы оценки и основные ответственные за проект лица	Проходит в неформальной обстановке, согласно конкретной ситуации; может длиться несколько недель
1	Оценка	Группа оценки и ответственные за проект лица	1 день с последующим перерывом продолжительностью от 2 до 3 недель
2	Оценка (продолжение)	Группа оценки, ответственные за проект лица и заинтересованные лица	2 дня
3	Доработка	Группа оценки и заказчик оценки	1 неделя

[1] *Источник*: приводится по изданию [Clements 02a] (адаптированная версия).

На первом и втором этапах проводится непосредственно оценка — все погружены в аналитические операции. К началу этих этапов все участники группы оценки должны ознакомиться с документацией по архитектуре, получить достаточное представление о системе, знать задействованные архитектурные методики и ориентироваться в первостепенных атрибутах качества. На первом этапе, намереваясь приступить к сбору и анализу информации, участники группы оценки встречаются с лицами, ответственными за проект (как правило, встреча длится весь день). На втором этапе к специалистам присоединяются заинтересованные в архитектуре лица, и в течение примерно двух дней они проводят аналитические мероприятия совместно. Первой и второй этапы подробно расписаны в следующем разделе.

Третий этап занимает доработка — группа оценки составляет в письменном виде и предоставляет получателям сводный отчет. По сути, участники занимаются самопроверкой и вносят в результаты своей работы разного рода коррективы. На заключительном совещании группы обсуждаются успехи и трудности. Участники изучают отчеты, выданные им на первом и втором этапах, и заслушивают выступление наблюдателя за процессом. По сути, они занимаются поиском путей усовершенствования по части исполнения своих ролей, с тем чтобы проводить последующие оценки с меньшими усилиями и с более высокой эффективностью. Действия, выполненные в период оценки участниками трех групп, тщательно регистрируются. По прошествии нескольких месяцев руководитель группы должен связаться с заказчиком оценки, для того чтобы оценить долгосрочные результаты ее проведения и сравнить издержки с выгодами.

Четыре этапа ATAM, их участники и приблизительный график представлены в табл. 11.2.

Операции на различных этапах оценки

Аналитические этапы ATAM (этап 1 и 2) состоят из девяти операций. Операции с первой по шестую проводятся на первом этапе. На втором этапе в присутствии всех заинтересованных лиц подводятся предварительные итоги и выполняются операции 7–9.

Аналитические операции, как правило, выполняются последовательно, в соответствии с установленным планом, однако для адаптации к доступным трудовым ресурсам или архитектурной информации возможно проведение динамических модификаций. Каждая оценка уникальна — иногда, если в том есть необходимость, группа на короткое время возвращается к предыдущим операциям, раньше времени переходит к последующим или выполняет операции повторно.

Операция 1: презентация ATAM

В ходе первой операции руководитель специалистов по оценке организует для представителей проекта презентацию метода ATAM. От него требуется растолковать всем участникам суть процесса, которому им придется следовать, ответить на все вопросы, а также определить контекст остальных операций и их ожидаемые результаты. Стандартная форма презентации позволяет руководителю кратко охарактеризовать все операции, предусмотренные в ATAM, и поставленные перед ними задачи.

Операция 2: презентация коммерческих факторов

Все участники оценки — как представители проекта, так и специалисты из группы оценки — должны знать контекст системы, а также основные коммерческие факторы, стимулирующие ее разработку. Ответственное за проект лицо (в идеале — руководитель проекта или заказчик системы) обязано представить обзор системы с коммерческой точки зрения. В числе прочего в составе этой презентации должны содержаться описания:

- важнейших функций системы;
- всех значимых технических, управленческих, экономических или политических ограничений;
- коммерческих задач и коммерческого контекста в его отношении к проекту;
- основных заинтересованных лиц;
- архитектурных мотивов (иначе говоря, основных задач по реализации атрибутов качества, оказывающих на архитектуру определяющее воздействие).

Операция 3: презентация архитектуры

На этом этапе главный архитектор (или группа архитекторов) организует презентацию архитектуры на адекватном ситуации уровне детализации. «Адекватность» детализации обусловливается несколькими факторами: степенью разработанности проекта и документации архитектуры, временными ограничениями, а также характером требований к поведению и качеству.

В ходе презентации архитектор должен изложить технические ограничения — в частности, назвать операционную систему, аппаратное и промежуточное программное обеспечение, а также другие системы, с которыми будет взаимодействовать рассматриваемая система. Что самое главное, от архитектора требуется описать архитектурные методики (или образцы, если архитектор свободно владеет соответствующим словарем), применяемые для реализации требований.

Максимально эффективное использование отведенного на презентацию времени обеспечивает высокий коэффициент «сигнал–помеха». Другими словами, не переходя к обсуждению вспомогательных областей и уклоняясь от детального рассмотрения отдельных аспектов, архитектор должен донести до слушателей суть архитектуры. Таким образом, нелишне заблаговременно рассказать архитектору, какую именно информацию от него ожидают получить участники группы оценки. При подготовке презентации применяются специальные шаблоны наподобие показанного в листинге 11.1. Некоторым архитекторам трудно обойтись без «генеральной репетиции» презентации, которая также считается одной из операций первого этапа.

Листинг 11.1. Пример шаблона презентации архитектуры[1]

```
Презентация архитектуры (~20 слайдов; 60 минут).

Важнейшие архитектурные требования, связанные с ними измеряемые величины
и все существующие стандарты/модели/методики, направленные на их
удовлетворение (2-3 слайда).
```

[1] *Источник*: приводится по изданию [Clements 02a] (адаптированная версия).

Значимые сведения об архитектуре (4-8 слайдов):

- Контекстная диаграмма, демонстрирующая систему в том контексте, в котором она должна будет существовать. Люди и другие системы, которые будут с ней взаимодействовать.
- Модульное или многоуровневое представление — модули (возможно, подсистемы или уровни), описывающие декомпозицию функциональности в системе, а также содержащиеся в них объекты, процедуры и функции и отношение между последними (например, вызовы процедур и методов, обратные вызовы, включение).
- Представление «компонент и соединитель» — процессы и потоки вместе с тем, что их соединяет: синхронизацией, потоками данных и событиями.
- Представление размещения — процессоры, память, внешние устройства/датчики, а также соединяющие их сети и устройства связи. Здесь же должны быть показаны процессы, исполняющиеся на разных процессорах.

Применяемые архитектурные методики, образцы и тактики с указанием атрибутов качества, на реализацию которых они направлены, и описанием механизмов этой реализации (3-6 слайдов).

- Применение коммерческих коробочных продуктов (COTS) вместе с описанием механизмов их отбора/интеграции (1-2 слайда).
- Кальки 1-3 наиболее важных сценариев Use Case. По возможности следует указывать потребление ресурсов периода прогона каждым из сценариев (1-3 слайда).
- Кальки 1-3 наиболее важных сценариев изменений. По возможности следует приводить описания воздействия изменений (расчетный диапазон/трудность проведения изменений) применительно к модифицируемым модулям или интерфейсам (1-3 слайда).
- Архитектурные проблемы/риски, связанные с удовлетворением важнейших архитектурных требований (2-3 слайда).
- Глоссарий (1 слайд).

По приведенному шаблону презентации видно, что основным инструментом изложения архитектуры являются архитектурные представления (см. главу 2). За редкими исключениями, в ходе проведения оценки весьма полезными оказываются контекстные диаграммы, «компонент и соединитель», представления декомпозиции уровней и размещения, а также многоуровневые представления; соответственно, архитектор должен быть готов продемонстрировать их. Если значимая в контексте рассматриваемой архитектуры информация, в особенности относящаяся к выполнению важнейших задач по реализации атрибутов качества, содержится в каких-то других представлениях, их также следует упомянуть. В общем, архитектору рекомендуется оперировать теми представлениями, которые в процессе создания архитектуры показались ему наиболее важными.

Во время презентации участники группы оценки могут попросить архитектора прояснить какие-то непонятные моменты; при этом они исходят из знаний, полученных в результате проводимого на нулевом этапе анализа архитектурной документации, и изучения в ходе предшествующей операции коммерческих факторов. Кроме того, они ожидают получить и зафиксировать сведения о применяемых архитектурных тактиках и образцах.

Операция 4: выявление архитектурных методик

Согласно ATAM, необходимым условием анализа архитектуры является компетентность в вопросах задействованных в ней архитектурных методик. На материале главы 5 мы выяснили, что архитектурные образцы, помимо прочего, вы-

годно отличаются тем, что их воздействие на различные атрибуты качества хорошо изучено. Так, многоуровневый образец повышает переносимость системы, причем иногда — за счет снижения производительности. Образец репозитария данных, как правило, отличается масштабируемостью по части количества производителей и потребителей данных. И так далее.

В рассматриваемый момент у всех участников группы уже должно сложиться четкое представление об образцах и методиках, задействованных архитектором при проектировании системы. Для этого есть все предпосылки — ведь они ознакомились с архитектурной документацией, а в ходе третьей операции прослушали проведенную архитектором презентацию. На том же этапе архитектор должен явно, по именам перечислить использованные образцы и методики, а от участников группы требуется умение распознавать те из них, которые архитектор не упомянул.

В период проведения настоящей операции задача группы оценки заключается в том, чтобы каталогизировать очевидные образцы и методики. Обязанность по составлению (при участии специалистов) соответствующего списка, а также по его последующему распространению возлагается на секретаря. В дальнейшем этот список становится основой для проведения анализа.

Операция 5: генерация дерева полезности атрибутов качества

В зависимости от того, способна ли архитектура реализовать в системе те или иные атрибуты качества, она признается пригодной или непригодной для конструирования этой самой системы. Архитектура, в которой достигается высочайшая производительность, никоим образом не сочетается с системой, в которой приоритет отдается, скажем, безопасности. В ходе операции 2, попутно с изложением коммерческих факторов, должны быть представлены задачи по реализации атрибутов качества, важнейшие в контексте данного варианта архитектуры; впрочем, их детализация на этом этапе не позволяет выполнять действия аналитического характера. Общие задачи наподобие «модифицируемости», «высокой пропускной способности» или «возможности перенесения на разные машины» обозначают контекст и направление, формируют основу для изложения последующей информации. С другой стороны, они не настолько конкретны, чтобы на их основе можно было делать выводы о пригодности архитектуры. О какой модифицируемости идет речь? Насколько высокой должна быть пропускная способность? На какие конкретно машины предполагается переносить систему?

Настоящая операция предполагает детальную фиксацию задач по реализации атрибутов качества; для достижения этой цели применяется механизм под названием «дерево полезности». Участники группы оценки совместно с ответственными за проект лицами устанавливают наиболее важные для системы задачи по реализации атрибутов качества, расставляют их согласно приоритетам, уточняют и выражают в виде сценариев. Дерево полезности способствует конкретизации требований по качеству и тем самым склоняет архитектора и представителей заказчика к их точному формулированию.

Корневым узлом дерева полезности является собственно *полезность* (utility), выражающая общую «добротность» системы. Атрибуты качества как компоненты полезности располагаются на втором уровне. Те из них, которые обозначены в презентации коммерческих факторов во время второй операции, составляют

исходный (начальный) набор второго уровня. Чаще всего дочерними элементами полезности оказываются производительность, модифицируемость, безопасность, практичность и готовность, однако оценщикам ничто не мешает вводить новые имена. Иногда разные группы заинтересованных лиц по-разному называют одни и те же концепты (в частности, некоторые заинтересованные лица любят термин «удобство сопровождения»). В отдельных случаях они употребляют такие понятия, которые носят явную смысловую нагрузку только в их собственной субкультуре, однако в других областях почти не используются, — например, что-нибудь вроде «гибкорасширяемости» (flextensibility). Имена, вводимые заинтересованными лицами, адекватны лишь до тех пор, пока по мере уточнения на последующих уровнях они могут объяснить свое значение (см. врезку «Что в имени твоем?»).

На более низких уровнях, дочерних по отношению к каждому из атрибутов качества, располагаются их уточнения. К примеру, производительность можно разложить на «задержку данных» и «пропускную способность для транзакций». Это — очередной шаг на пути к уточнению задач по атрибутам до сценариев атрибутов качества, предполагающих возможность расстановки приоритетов и проведения анализа. В частности, задержка данных уточняется до «Снизить задержку сохранения в базе данных по заказчикам до 20 мс» и «Довести частоту кадров видеоизображения в реальном времени до 20 кадров в секунду» — и тот и другой варианты задержки являются значимыми в контексте системы.

При помощи механизма сценариев (см. главу 4) общие (и неоднозначные) формулировки желаемых атрибутов качества конкретизируются и приводятся в форму, позволяющую проводить тестирование. Из них формируются и систематизируются по выражаемым атрибутам качества листья дерева полезности. Шестичастная форма, предложенная в главе 4, в данном случае адаптируется к задачам оценки. Сценарии ATAM состоят из трех частей: стимула (наблюдаемое в системе условие, породивший его источник и стимулируемый артефакт системы), среды (все происходящее в данный момент времени) и реакции (реакция системы на стимул, выраженный в измеримой форме).

Теперь в нашем распоряжении есть вполне осязаемые сведения, исходя из которых архитектуру можно оценивать. Собственно говоря, все аналитические операции в рамках ATAM предполагают единовременный выбор одного сценария и проверку степени реакции на него (реализации) со стороны архитектуры. Подробнее об этом мы поговорим применительно к следующей операции.

Некоторые сценарии выражают несколько атрибутов качества и, следовательно, размещаются на дереве сразу в нескольких местах. В этом, как правило, нет ничего страшного, однако, с другой стороны, руководитель группы оценки должен следить за теми сценариями, которые демонстрируют тенденцию к избыточному разрастанию, иначе их будет трудно анализировать. Пытайтесь разбивать сценарии на составляющие, выражающие относительно более мелкие проблемы.

Участники группы оценки должны не просто знать задачи, возложенные на архитектуру, но и осознавать их относительную значимость. На дереве полезности зачастую оказывается по 50 листьев-сценариев, на анализ которых во время совещания по оценке просто не хватает времени. Таким образом, составление дерева полезности одновременно способствует расстановке приоритетов. Решения о назначении сценариям приоритетов должны приниматься ответственными лицами согласованно. Иногда для этого вводится шкала приоритетов от 0 до 10,

в иных случаях вполне хватает относительных рангов: высокого, среднего и низкого. (Мы отдаем предпочтение второй методике: во-первых, она лучше приспособлена к разнородным группам, а во-вторых, по сравнению с присваиванием точных значений, занимает меньше времени.)

После этого в отношении сценариев проводится вторичная расстановка приоритетов — в этот раз на основе другого критерия. Архитектор ранжирует сценарии исходя из своих представлений о том, насколько проблематичным окажется реализация каждого из них в архитектуре. В данном случае, как и раньше, подходит схема «высокой/средней/низкой» сложности.

ЧТО В ИМЕНИ ТВОЕМ?

Изложенные в нашей книге методы оценки архитектуры предусматривают фиксацию атрибутов качества с помощью сценариев. Зачем это нужно? Дело в том, что сами по себе атрибуты качества слишком размыты, чтобы проводить на их основе аналитические действия. С другой стороны, дерево полезности ATAM систематизируется по именам атрибутов качества. Нет ли в этом противоречия? Нам не все равно, какие атрибуты качества заинтересованные лица сочтут наиболее важными. Они вольны называть атрибуты как угодно — если это способствует мыслительному процессу. К примеру, в табл. 11.5, приведенной ниже, разделены такие атрибуты, как «конфигурируемость» и «модульность»; у вас есть полное право возразить против такой формулировки — ведь оба указанных атрибута являются подвидами «модифицируемости», а значит, обозначить их следует как уточнения последнего. В принципе, мы с этим согласны. Но в данном случае, по мнению заинтересованных лиц, и тот и другой подвид в силу существенных различий заслуживали отдельных позиций на дереве полезности, и мы не стали с ними спорить. В конце концов, листы-сценарии на дереве полезности значительно важнее структуры ветвей.

Ситуация, при которой одни и те же имена атрибутов качества переходят из одной оценки в другую, встречается очень редко. То, что одна компания называет «удобством сопровождения», другая предпочитает именовать «взаимозаменяемостью». Иногда «переносимость» рассматривается как подвид модифицируемости, но во многих других случаях заинтересованные лица предпочитают трактовать эти атрибуты по отдельности. Надежность часто называют готовностью, и наоборот. Помимо всего прочего, нам приходилось сталкиваться с оригинальными именами атрибутов качества, смысл которых был понятен только сотрудникам конкретной организации, — примерами тому «развертываемость» и «продаваемость». Мы сначала не совсем поняли, что это значит, но, поскольку в ATAM нет необходимости тратить ценное время на терминологические споры, никаких проблем не возникло. Реальное значение атрибутов выражается в сценариях. Самое главное — упомянутые термины были понятны заинтересованным лицам; исходя из этого они могли составить сценарии и тем самым сформулировать задачи, которые требовалось решить.

В одном из случаев оценки по методу ATAM мы столкнулись с довольно оригинальными задачами компании-разработчика — они хотели привлечь новых способных сотрудников к работе в своем центральном офисе, который, на беду, находился в маленьком тихом городке на американском Среднем Западе. Исходя из этого коммерческого фактора они сформулировали архитектурную задачу — архитектура должна была опираться на современные программные разработки, которые, как предполагалось, заинтересуют новых специалистов.

Несмотря на то что атрибута качества под именем «Iowa-bility» нет ни в одном стандартном перечне IEEE, ISO или ANSI, в нашей практике был случай, когда он нашел свое отражение на дереве полезности ATAM и заставил специалистов основательно поразмыслить над тем, какие сценарии смогут выразить сформулированную в таком виде задачу.

— PCC

Теперь каждому сценарию соответствует упорядоченная пара наподобие (В,В[ысокий]), (В,С[редний]), (В,Н[изкий]) и т. д. Самые важные и сложные сценарии требуют выделения значительных временных ресурсов для проведения анализа, а оставшаяся часть будет просто фиксироваться. Сценариям, рассматриваемым

как несущественные (H,*) или легко реализуемые (*,H), вряд ли будет уделяться слишком много внимания.

Продуктом составления дерева полезности является перечень сценариев с расставленными приоритетами, выступающий в качестве плана проведения оставшейся части оценки по методу ATAM. Он сообщает участникам группы о том, на что имеет смысл потратить ограниченное время, — в частности, в каких случаях следует провести испытания архитектурных методик и рискованных решений. Дерево полезности помогает специалистам по оценке сориентироваться в архитектурных методиках, способных реализовать высокоприоритетные листы-сценарии.

К рассматриваемому периоду вся необходимая для проведения анализа информация должна быть выражена в табличной форме — в частности, речь идет об ожидаемых значимых атрибутах качества, реализуемых архитектурой и обусловливаемых коммерческими факторами (операция 2) и деревом полезности (операция 5), а также о самой архитектуре, зафиксированной в виде презентации (операция 3) и каталога применяемых методик (операция 4). Пример дерева полезности в табличной форме (без корневого узла) приводится в табл. 11.5.

Операция 6: анализ архитектурных методик

В рамках данной операции группа оценки по одному исследует сценарии с наивысшим приоритетом; архитектор при этом объясняет, каким образом они реализуются в архитектуре. Участники группы оценки — по большей части дознаватели — испытывают архитектурные методики, при помощи которых архитектор реализует каждый из сценариев. Параллельно группа документирует значимые архитектурные решения, устанавливает и каталогизирует рискованные и нерискованные решения, точки чувствительности и компромиссы. Если речь идет о хорошо изученной методике, участники группы расспрашивают архитектора о том, как ему удалось преодолеть ее известные недостатки и из чего он сделал вывод о ее адекватности. Тем самым они проверяют, подходит ли конкретная реализация методики для удовлетворения требований по конкретным атрибутам качества.

К примеру, на количество транзакций, которые база данных может обработать за одну секунду, влияет количество одновременных обращений к ней. Из этого следует, что по отношению к отклику, измеряемому транзакциями в секунду, распределение клиентов между серверами является точкой чувствительности. Если значение отклика при распределении становится неприемлемым, значит, мы имеем дело с рискованным решением. Если же архитектурное решение оказывается точкой чувствительности в отношении сразу нескольких атрибутов, его следует признать точкой компромисса.

В ходе критического анализа сценария следует обсуждать возможные риски, нерискованные решения, точки чувствительности и компромисса. Такие дискуссии, в свою очередь, иногда диктуют необходимость в проведении углубленного анализа. Определяющей здесь является позиция архитектора. Если, к примеру, архитектор оказывается не способен охарактеризовать количество клиентов или предложить выравнивание нагрузки путем распределения процессов между аппаратными устройствами, заниматься формированием сложных очередей или проводить частотно-монотонный анализ производительности не имеет смысла. Если же архитектор отвечает на эти вопросы, группа оценки должна провести

хотя бы фрагментарный, поверхностный анализ с целью выявления недостатков принятых архитектурных решений в контексте реализации ими требований по атрибутам качества. Комплексный анализ здесь не требуется. Цель заключается в том, чтобы при помощи выявленной значимой архитектурной информации установить связь между принятыми архитектурными решениями и требованиями по атрибутам качества, которые предполагается удовлетворить.

Сценарий #: A12	Сценарий: обнаружение аппаратного отказа главного переключателя и последующее восстановление
Атрибут(ы)	Готовность
Условия	Нормальный режим работы
Стимул	Отказ одного из процессоров
Реакция	Готовность переключателя 0,999999

Архитектурные решения	Чувствительность	Компромиссное решение	Риск	Без риска
Резервный(е) процессор(ы)	S2		R8	
Отсутствие резервного канала передачи данных	S3	T3	R9	
Сторожевой режим	S4			N12
Heartbeat-сообщение	S5			N13
Маршрутизация резервирования узла	S6			N14

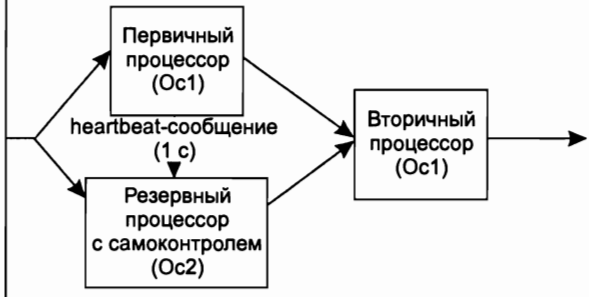
Обоснование	Защита от группового отказа обеспечивается путем применения разных аппаратных средств и операционной системы (см. риск 8) Продолжительность нажатия для наихудшего случая составляет 4 секунды — именно столько времени в наихудшем случае длится состояние вычисления Исходя из частоты отправки heartbeat-сообщений и проведения самоконтроля обнаружение отказа занимает не более 2 секунд Сторожевой режим отличается простотой и проверенной надежностью Из-за отсутствия резервного канала передачи данных риску подвергается требование о готовности (см. риск 9)
Диаграмма архитектуры	

Рис. 11.1. Пример анализа архитектурной методики[1]

[1] *Источник*: приводится по изданию [Clements 02a] (адаптированная версия).

На рис. 11.1 изображена стандартная форма, предназначенная для фиксации результатов анализа архитектурной методики применительно к конкретному сценарию. Из него следует, что по результатам рассматриваемой операции участники группы оценки устанавливают ряд точек чувствительности и компромиссов, рискованные и нерискованные решения. Любые точки чувствительности и компромиссы потенциально рискованны. К моменту завершения оценки по методу ATAM их необходимо занести в одну из двух категорий: рискованные и нерискованные. Рискованные и нерискованные решения, точки чувствительности и компромиссы — все они фиксируются в отдельных списках. Приведенные на рис. 11.1 обозначения R8, T3, S4, N12 и т. д. — это указатели на соответствующие позиции в таких списках.

К моменту завершения рассматриваемой операции у всех участников группы оценки должно сложиться устойчивое представление о важнейших аспектах архитектуры в целом и о логическом обосновании основных проектных решений; кроме того, в их распоряжении должны быть списки рискованных и нерискованных решений, точек чувствительности и компромиссов.

Перерыв и начало второго этапа

На этом завершается первый этап. Группа оценки делает перерыв на одну-две недели, во время которого ее участники резюмируют полученные данные и в неформальной манере (как правило, по телефону) переговариваются с архитектором. При желании в этот период можно провести анализ дополнительных сценариев или уточнить неясные вопросы.

Второй этап начинается лишь тогда, когда, во-первых, к его проведению подготовятся ответственные за проект руководители, а во-вторых, вместе соберутся все заинтересованные лица. Все работы на этом этапе проводятся в присутствии многочисленных участников и расширенной группы заинтересованных лиц. Сначала для того чтобы заинтересованные лица получили представление о применяемом методе и своей в нем роли, резюмируются результаты первого этапа. Затем руководитель группы оценки воспроизводит результат операций со второй по шестую, озвучивает текущее состояние списка рисков, нерискованных решений, точек чувствительности и компромиссов. После этого, когда заинтересованные лица «догонят» процесс оценки, можно приступать к трем оставшимся операциям.

Операция 7: мозговой штурм и расстановка сценариев согласно приоритетам

Составление дерева полезности главным образом преследует целью воспроизводство архитектурных мотивов по атрибутам качества с точки зрения архитектора и выбранных им путей реализации таковых. Мозговой штурм сценариев решает другую задачу — он выражает настроения, царящие среди заинтересованных лиц в условиях их наиболее широкого представительства. Наибольшая эффективность этой методики появляется в крупных группах, где она создает ситуацию стимулирования одних размышлений другими. Этот процесс благоприятствует взаимодействию заинтересованных лиц, поощряет творческую мысль и выражает коллективную позицию участников. Перечень сценариев, сформулированных методом

мозгового штурма, подлежит сравнению с перечнем, составленным согласно дереву полезности. Если эти два документа не противоречат друг другу, значит, намерения архитектора и ожидания заинтересованных лиц сходятся. Выявление новых значимых сценариев — это уже риск, свидетельствующий о рассогласовании целей заинтересованных лиц, с одной стороны, и архитектора — с другой.

Участники группы оценки на рассматриваемом этапе просят заинтересованных лиц озвучить сценарии, носящие операционно-смысловую нагрузку в контексте их индивидуальных ролей. Так, специалист по сопровождению, вероятно, сформулирует сценарий реализации модифицируемости, а пользователь в своем сценарии, скорее всего, сделает упор на ту или иную полезную функциональность или на удобство взаимодействия с системой.

Вполне допустимым на этом этапе следует считать возврат к тем сценариям дерева полезности, которые не были проанализированы ранее. Воссоздавая в ходе мозгового штурма те сценарии 5 и 6 операций, которым, по их мнению, не было уделено должного внимания, заинтересованные лица компенсируют недоработки.

После сбора сценариев следует операция по расстановке среди них приоритетов. Такая необходимость вызвана теми же факторами, которые обусловливают назначение приоритетов на дереве полезности, — участники группы оценки должны знать, чему следует уделить ограниченные временные ресурсы. Во-первых, заинтересованных лиц просят соединить воедино сценарии, которые, по их мнению, выражают один и тот же тип поведения или одну и ту же задачу по атрибуту качества. Затем путем голосования из числа получившихся сценариев выбираются наиболее важные. Каждому заинтересованному лицу предоставляется количество голосов, составляющее 30 % от общего числа сценариев[1], после чего полученное число округляется. Так, если сценариев всего двадцать, каждое заинтересованное лицо получает по шесть голосов. Распоряжаться ими заинтересованное лицо может произвольно — если потребуется, оно отдаст все шесть голосов за один сценарий, или по одному голосу за каждый из любых шести сценариев, или примет какое-либо промежуточное решение.

Голосование по сценариям проходит открыто; по нашему опыту, этот способ не только забавен — он способствует укреплению у участников чувства общности. После подсчета голосов руководитель группы оценки сортирует сценарии по количеству поданных голосов и выделяет из них те, за которые проголосовало заметно меньше заинтересованных лиц, чем за все остальные. Сценарии с наивысшими показателями утверждаются и впоследствии участвуют в дальнейших операциях. Так, к примеру, группа может взять на рассмотрение лишь пять сценариев, набравших наибольшее количество голосов.

Операция 8: анализ архитектурных методик

После выявления сценариев и их расстановки согласно приоритетам группа оценки инструктирует архитектора в процессе реализации наиболее ценных (см. операцию 7), с точки зрения заинтересованных лиц, сценариев. Архитектор должен

[1] Эта упрощенная методика довольно часто применяется в ходе мозгового штурма.

объяснить механизм воздействия значимых архитектурных решений на их реализацию. Лучше всего, если главным действующим лицом во время этой операции станет архитектор, объясняющий реализацию сценариев в контексте рассмотренных ранее архитектурных методик.

Действия, проводящиеся участниками группы оценки, схожи с теми, что требуется выполнить во время операции 6. Иначе говоря, они отображают недавно составленные сценарии с наивысшим приоритетом на выявленные к настоящему моменту архитектурные артефакты.

Операция 9: презентация результатов

Наконец, информацию, собранную согласно методике ATAM, необходимо резюмировать и еще раз изложить в присутствии заинтересованных лиц. Обычно подобного рода презентации организуются в форме устного отчета с показом слайдов, однако не исключается и вариант последующего предоставления заинтересованным лицам более подробного письменного отчета. Во время презентации руководитель группы оценки в очередной раз перечисляет этапы ATAM и излагает собранную по результатам их выполнения информацию — в частности, коммерческий контекст, важнейшие требования, ограничения и архитектуру. Помимо прочего, в отчете должны быть отражены следующие моменты:

- документированные архитектурные методики;
- набор сценариев, сформулированных методом мозгового шторма, и их приоритеты;
- дерево полезности;
- выявленные риски;
- установленные нерискованные решения;
- найденные точки чувствительности и компромиссы.

В ходе оценки все эти моменты должны быть обнаружены, публично оглашены и письменно зафиксированы. Также во время рассматриваемой операции участники группы оценки, исходя из какой-либо значимой задачи или регулярно встречающегося недостатка, формулируют (на основе выявленных рискованных решений) магистральные риски. К примеру, совокупность рисков, связанных с недостаточной или устаревшей документацией, можно сгруппировать в рамках магистрального риска, декларирующего нехватку внимания к документации. Совокупность рисков, связанных с неспособностью системы функционировать в условиях разного рода аппаратных и/или программных отказов, формирует магистральный риск недостаточного внимания к резервированию или готовности.

Для каждого из установленных магистральных рисков группа определяет находящиеся под его воздействием коммерческие факторы (они должны были быть выявлены в ходе операции 2). Определение магистральных рисков и их связей с конкретными факторами замыкает процесс оценки, поскольку его конечные результаты соотносятся с первоначальной презентацией. Не менее важно, что оно раскрывает перед ответственными лицами суть установленных рисков. То, что руководитель ранее рассматривал как далекий от практического контекста технический вопрос, теперь со всей очевидностью предстает как угроза, лежащая в зоне его ответственности.

В табл. 11.3 отражены все девять операций ATAM и показана степень их влияния на конечные продукты оценки по этой методике. «**» обозначает операции, которые напрямую воздействуют на эти результаты, а «*» — операции, влияющие на них лишь косвенно.

Таблица 11.3. Коррелированные операции и продукты ATAM[1]

	Продукты ATAM						
Операции	**Формулировка требований по атрибутам качества с расставленными приоритетами**	**Каталог применяемых архитектурных методик**	**Аналитические вопросы, касающиеся конкретных методик и атрибутов качества**	**Отображение архитектурных методик на атрибуты качества**	**Рискованные и нерискованные решения**	**Точки чувствительности и точки компромиссов**
1. Презентация ATAM						
2. Презентация коммерческих факторов	*[a]				*[b]	
3. Презентация архитектуры		**			*[c]	*[d]
4. Выявление архитектурных методик		**	**		*[e]	*[f]
5. Составление дерева полезности атрибутов качества	**					
6. Анализ архитектурных методик		*[g]	**	**	**	**
7. Мозговой штурм и распределение сценариев по приоритетам	**					
8. Анализ архитектурных методик		*	**	**	**	**
9. Презентация результатов						

[a] В ходе установления коммерческих факторов излагается первоначальное, наиболее общее описание атрибутов качества.

[b] Во время презентации коммерческих факторов допускается разглашение ранее выявленных или устойчивых рисков, которые в таком случае необходимо зафиксировать.

[c] В своей презентации архитектор может установить дополнительные риски.

[d] В своей презентации архитектор может выявить дополнительные точки чувствительности или компромиссы.

[e] Стандартные сопутствующие риски характерны для многих архитектурных методик.

[f] Многим архитектурным методикам свойственны стандартные сопутствующие варианты чувствительности и компромиссы между атрибутами качества.

[g] В ходе рассматриваемых аналитических операций возможно выявление новых архитектурных методик, не замеченных во время операции 4; в таком случае формулируются новые методико-ориентированные вопросы.

[1] *Источник*: приводится по изданию [Clements 02a] (адаптированная версия).

ИХ РЕШЕНИЕ НЕ ГОДИТСЯ

Возможно, у вас сложилось такое впечатление, что роль заинтересованных лиц в ходе проведения оценки по методу ATAM сводится к формулированию задач архитектуры и сценариев. В этом контексте следует заметить, что их присутствие во время презентации и оценки архитектуры не раз сыграло очень важную роль. Уровень знаний, которыми обладают заинтересованные лица, позволяет им заострять внимание на важных проблемах, которые архитектура (или специалист, выступающий на ее презентации) обходит стороной. В качестве примера приведем случай с оценкой системы управления финансами — прикладной области, в которой оценщики не слишком хорошо разбирались. Из-за этого во время оценки неоднократно разгорались дискуссии — в частности, такая.

Оценщик: Ладно, перейдем к следующему сценарию. Предоставляет ли ваша система такую-то возможность?

Производитель (мило улыбаясь): Конечно! От пользователя требуется лишь ввести номер счета, вызвать таблицу дебиторской задолженности и перенести результаты в файл уведомления спонсора.

Оценщик (одобрительно кивая, проверяя «нерискованность» сценария и радуясь тому, что оценка обещает пройти легче, чем он предполагал): Здорово! Так, теперь следующий сценарий.

Пользователь системы 1 (возмущенно): Секундочку! Вы хотите сказать, что перемещать данные автоматически нельзя? Это что ж получается — мне придется их вводить во *все* файлы уведомлений?

Производитель (с первыми признаками нервозности): Ну-у-у, вы знаете...

Пользователь системы 1 (в оскорбленных чувствах): Да вы знаете, сколько спонсоров у крупного университета вроде нашего?!

Производитель (теребит свой воротник): Что, много?

Пользователь системы 1 (теперь его черед мило улыбнуться): Да! Много.

Пользователь системы 2: А что, если я не знаю, какой номер счета вводить? Операция-то проводится именно по этой причине. В противном случае легче взять расписку об обновлении платежа.

Пользователь системы 1 (оценщику): Их решение не годится.

Оценщик (честно пытаясь вспомнить, что такое файл уведомления спонсора, почесывая голову по поводу этой загадочной расписки об обновлении платежа и, наконец, осторожно затирая поставленную было галочку): Так... кажется, у нас здесь риск. Как вы считаете, что следует изменить?

Вывод один. Компетентные заинтересованные лица способны выискать проблему, недоступную людям со стороны.

— РСС

Эффективное распоряжение ограниченными временными ресурсами

Как мы уже говорили во введении, одним из основных препятствий к проведению оценки архитектуры является нехватка времени. Очевидно, что методика ATAM решает эту проблему. Коммерческие цели в данном случае стимулируют сбор сценариев, составляющих дерево полезности. Определение приоритетов для остальных сценариев производится, по сути, путем восходящей проверки нисходящего метода составления сценариев на дереве полезности. В качестве руководства по оценке этих важных, но в то же время проблемных областей архитектуры выступают операции, составляющие методику. Именно в них сосредоточиваются наиболее значимые результаты.

11.4. Система Nightingale: конкретный пример проведения оценки по методу ATAM

Конкретный пример, который мы намерены привести в настоящем разделе, основан на нашем собственном опыте проведения оценки; он иллюстрирует процесс практического применения ATAM. Не считая возможным нарушать конфиденциальность компании-заказчика, мы изменили все названия, способные раскрыть информацию о ней.

Нулевой этап: установление партнерских отношений и подготовка

Заказчик оценки, который связался с нами, ознакомившись с опубликованными на нашем веб-сайте материалами по методу ATAM, оказался представителем крупного производителя программных систем для учреждений здравоохранения, сотрудничавшего с больницами, клиниками и HMO. Система, которую нам предстояло проверить, называлась Nightingale. Нам рассказали, что это крупная система, состоящая из нескольких миллионов строк кода, довольно давно перешедшая в стадию реализации. У нее уже был первый покупатель — сеть из сорока с лишним больниц, расположенных на юго-западе Соединенных Штатов.

Естественно, мы недоумевали, зачем заказчику проводить оценку архитектуры, если система почти готова к выпуску и распространению? Оказывается, он руководствовался двумя соображениями. Во-первых, если в архитектуре есть серьезные недостатки, в чем бы они ни проявлялись, об этом лучше узнать как можно раньше; во-вторых, компания, намеревавшаяся продавать систему другим своим клиентам, осознавала необходимость ее адаптации к потребностям, вариантам применения и регулятивному окружению каждого из них. Следовательно, пусть даже архитектура оказалась подходящей для первого, дебютного заказчика, компании нужно было убедиться в том, что она в достаточной степени выносliва и модифицируема а, значит, сможет послужить основой для создания семейства систем управления в области здравоохранения.

Предполагалось, что рассматриваемая система после ее установки в учреждениях здравоохранения будет исполнять роль информационной магистрали. В ее функции, помимо прочего, входило предоставление данных об истории болезни пациентов, отслеживание их страховых и прочих взносов. Являясь одновременно информационным хранилищам, она должна была выявлять разного рода тенденции (например, предшественников и рецидивы отдельных болезней). Система также должна была генерировать большое количество периодических отчетов и отчетов по требованию, причем каждый из них следовало адаптировать к потребностям конкретного учреждения. Для тех пациентов, которые вносили платежи самостоятельно, система должна была выполнять последовательность действий, связанных с инициализацией и обслуживанием ссуды на протяжении всего периода ее погашения. Более того, поскольку система должна была работать (или, по меньшей мере, быть доступной) во всех подразделениях данного учреждения здравоохранения, от нее требовалась настраиваемость на все возможные конфи-

гурации. В частности, в разных подразделениях используются разные конфигурации аппаратных средств и разные формы отчетности. Если пользователь переходит с одного рабочего места на другое, система должна узнавать его и, вне зависимости от местоположения, удовлетворять его информационные потребности.

На переговоры о смете ушло около месяца — вполне нормальный срок, учитывая то, что речь идет об урегулировании юридических формальностей между двумя крупными организациями. Когда смета, наконец, были подписана, мы собрали группу оценки из шести специалистов[1], распределение ролей между которыми показано в табл. 11.4.

Таблица 11.4. Распределение ролей между участниками группы оценки

Участник	Роль
1	Руководитель группы, руководитель специалистов по оценке, дознаватель
2	Руководитель специалистов по оценке, дознаватель
3	Хронометрист, дознаватель
4	Секретарь по сценариям, дознаватель, сборщик данных
5	Дознаватель, координатор процесса
6	Секретарь по результатам, наблюдатель за процессом

Мы назначили двух руководителей специалистов-оценщиков, которым предстояло координировать действия группы поочередно. По нашему опыту, эта схема значительно снижает утомление и нагрузки, а следовательно, приводит к более достойным результатам. Дознавателей мы выбирали исходя из их компетенции в вопросах производительности и модифицируемости. Кроме того, мы отдавали предпочтение специалистам с опытом интегрирования коммерческих коробочных продуктов — заказчик почти сразу предупредил нас, что в состав Nightingale входят несколько десятков коммерческих программных пакетов. К счастью, у одного из наших дознавателей также был опыт работы в области здравоохранения.

Мы провели однодневное вступительное совещание, на котором присутствовали участники группы оценки, руководитель проекта, главный архитектор, а также руководитель проекта первого покупателя Nightingale. Троих последних можно причислить к категории лиц, ответственных за проект. В результате мы узнали много нового о возможностях системы и требованиях к ней, получили каталог готовой архитектурной документации (из которого мы выбрали нужные для наших целей документы) и составили список заинтересованных лиц, которых предполагалось пригласить во время проведения второго этапа. Кроме того, мы согласовали график проведения совещаний на первом и втором этапах и установили срок предоставления сводного отчета. Наконец, мы обговорили подробности презентаций, которые во время операций 2 и 3 первого этапа должны были сделать руководитель проекта и архитектор соответственно, и убедились в том, что им понятно, что мы хотим от них услышать.

[1] Шесть оценщиков — это много. Как мы уже говорили, в группу оценки, как правило, входят от трех до пяти специалистов; в среднем — четыре. В данном случае группу расширили за счет двух новых сотрудников, которым предстояло ознакомиться с процессом ATAM более подробно и на практике наработать опыт.

Позже, прежде чем приступать к первому этапу, мы провели двухчасовое совещание группы. Ее руководитель еще раз обозначил распределение ролей и проверил, все ли понимают свои обязанности. Далее он изложил краткий обзор полученной архитектурной документации, обратив внимание участников на указанные в ней образцы и тактики. В результате этого предварительного совещания несколько повысился уровень знаний (а значит, и уверенность) оценщиков об архитектуре; кроме того, были заложены основы для выполнения четвертой операции — каталогизации образцов и методик.

Наконец, на этом совещании документация Nightingale была признана неполной и неясной. Не говоря об отсутствии целых разделов, архитектура была в ней представлена в виде ряда недостаточно определенных блочно-линейных диаграмм. Было очевидно, что, начни мы оценку сразу, концептуальная база для наших заключений оказалась бы неполной. Придя к такому выводу, мы позвонили архитектору и попросили его в устной форме устранить некоторые пробелы в наших знаниях. После этого, осознавая неполноту полученной информации, мы все же решили, что приступать к оценке можно. При этом мы запланировали каталогизацию риска, связанного с недостаточной документированностью.

Этап 1: оценка

Согласно схеме первого этапа, участники группы оценки встретились с ответственными за проект лицами. Помимо специалистов, присутствовавших на вступительном совещании (руководитель проекта, главный архитектор и еще один руководитель проекта, представляющий первого покупателя Nightingale), к нам присоединились двое главных проектировщиков.

Операция 1: презентация ATAM

В ходе презентации руководитель специалистов по оценке обратился к стандартному пакету схем нашей организации и с его помощью объяснил присутствующим суть метода. На протяжении часа он рассказывал им об операциях и этапах ATAM, описывал концептуальные основы метода (сценарии, архитектурные методики, точки чувствительности и т. п.) и перечислял продукты, которые предполагалось получить в результате его проведения.

Поскольку ответственные лица, присутствовавшие на совещании нулевого этапа, уже имели некоторое представление об ATAM, презентация прошла без каких-либо заминок.

Операция 2: презентация коммерческих факторов

Руководитель проекта от компании-заказчика изложил коммерческие задачи, поставленные перед системой Nightingale компанией-разработчиком и потенциальными заказчиками. Как выяснилось, компания-разработчик предъявляла к Nightingale следующие требования:

- поддержка различных вариантов использования системы дебютным заказчиком (в частности, отслеживание историй болезни и платежных историй, выявление тенденций и т. д.);

- создание новой версии системы (предназначенной для установки в кабинетах врачей), которую компания-разработчик могла бы предлагать другим заказчикам.

Последний коммерческий фактор свидетельствовал о том, что на основе рассматриваемой архитектуры планировалось не просто разработать отдельную систему, а учредить целую линейку программных продуктов (см. главу 14).

Первый заказчик Nightingale планировал развернуть ее взамен существующих систем, которые:

- устарели (одной из них было больше 25 лет);
- были основаны на старых языках и технологиях (в частности, COBOL и ассемблере IBM);
- оказались неудобными по части сопровождения;
- не отвечали текущим и перспективным коммерческим потребностям учреждений системы здравоохранения, в которых размещались.

Первый заказчик выдвигал следующие коммерческие требования:

- способность реагировать на культурные и региональные различия;
- поддержка нескольких языков (в основном английского и испанского) и валют (в особенности американского доллара и мексиканского песо);
- быстродействие новой системы должно было быть по меньшей мере не меньшим, чем аналогичный показатель у заменяемых систем;
- новая единая система должна была сочетать в себе разнородные унаследованные системы управления финансами.

Коммерческие ограничения были таковы:

- соблюдение принципа неувольнения сотрудников, осуществляемого путем их переподготовки;
- приверженность принципу разработки «лучше купить, чем сконструировать»;
- учет сокращения рыночного положения заказчика (за счет повышения конкуренции).

Технические ограничения были сформулированы следующим образом:

- при любой возможности предпочтение следовало отдавать коробочным программным компонентам;
- реализовать систему требовалось в течение двух лет, имея в виду, что замена аппаратного обеспечения проводится регулярно, каждые 26 недель.

Первостепенными были признаны следующие атрибуты качества:

- *Производительность*. Полезность систем здравоохранения измеряется скоростью реагирования. Пятисекундного времени отклика на транзакцию в старых системах было недостаточно; столь же неадекватным признавалось время отклика на оперативные запросы и на составление отчетов. Значимой проблемой в контексте производительности была также пропускная способность системы.

- *Практичность.* Из-за высокой текучести среди пользователей системы значительную важность представляла проблема их переподготовки. Таким образом, новая система должна была стать удобной в применении и легко осваиваемой.
- *Удобство сопровождения.* Система должна быть удобной в сопровождении, легко конфигурируемой и расширяемой. Только в этом случае она сможет адаптироваться к выходу на новые рынки (в частности, к размещению в кабинетах врачей), удовлетворять новым требованиям заказчиков, приспосабливаться к изменению законов и норм штатов и к культурным и региональным различиям.

Нижеследующие атрибуты качества руководитель проекта признал важными, но не в такой степени, как вышеперечисленные:

- *Безопасность.* Система должна была соответствовать коммерческому стандарту безопасности (другими словами, обеспечивать конфиденциальность и целостность данных) для систем управления финансами.
- *Готовность.* В рабочие часы от системы требовалось высокая готовность.
- *Расширяемость.* В перспективе расширять систему предполагалось для ее адаптации к потребностям крупных больниц, а сокращать — в расчете на самые миниатюрные клиники.
- *Модульность.* Компания-разработчик планировала продавать не только новые версии Nightingale в целом, но и ее отдельные компоненты. Для того чтобы осуществить эту затею, необходимо было реализовать атрибуты качества, близкородственные удобству сопровождения и масштабируемости.
- *Контролепригодность и удобство поддержки.* Система должна была быть доступной для понимания техническими специалистами в штате заказчика; такая необходимость обусловливалась перспективами обучения персонала и продолжительного использования.

Операция 3: презентация архитектуры

В ходе совместной работы участников группы оценки и архитектора — как до, так и во время оценки — было сформулировано несколько новых представлений архитектуры и архитектурных методик. Основные выводы состояли в следующем.

- В составе Nightingale было две крупные подсистемы: диспетчер оперативных транзакций (OnLine Transaction Manager, OLTM) и диспетчер принятия решений и составления отчетов (Decision Support and Report Generation Manager, DSRGM). OLTM должен был отвечать требованиям по интерактивной производительности, а DSRGM исполнял роль системы пакетной обработки, исполнявшей периодически инициируемые задания.
- При конструировании системы Nightingale в ней предусматривалась высокая конфигурируемость.
- Подсистема OLTM состояла из нескольких четко разделенных уровней.

- Nightingale была системой репозитарного типа; ее основу составляла крупная коммерческая база данных.
- Nightingale испытывала исключительную зависимость от коробочного программного обеспечения; в такой форме в ней были реализованы центральная база данных, процессор правил, механизм автоматизации документооборота, CORBA, блок веб-хостинга, инструменты распределения программных средств и многое другое.
- Для Nightingale было характерно строгое следование объектно-ориентированной технологии и реализация конфигурируемости по большей части за счет объектных каркасов.

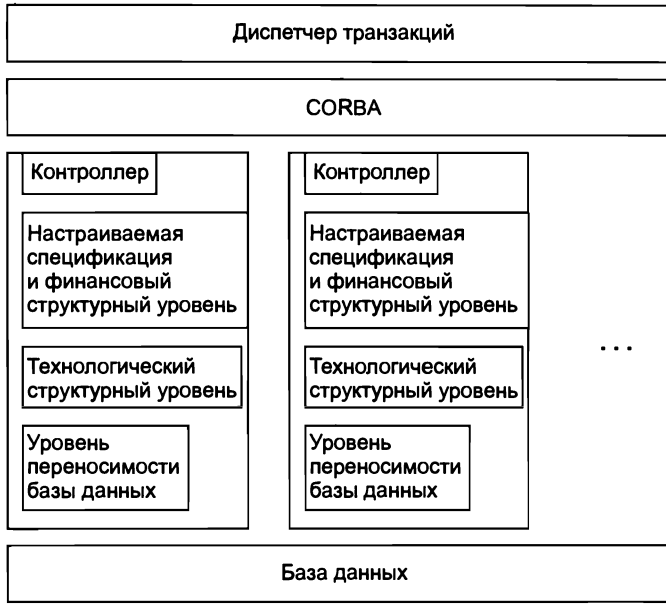


Рис. 11.2. Многоуровневое представление OLTM, изображенное архитектором в неформальной нотации

На рис. 11.2 показано многоуровневое представление OLTM, изображенное архитектором в неформальной нотации. На рис. 11.3 представлена схема функционирования OLTM в период исполнения — основные потоки информации и данных между частями системы, размещенными на различных аппаратных процессорах. Обе схемы мы приводим почти без изменений, для того чтобы наилучшим образом отразить реальные условия проведения оценки по методу ATAM. Обратите внимание на их неполное соответствие — на рис. 11.2 диспетчер транзакций и CORBA присутствуют, а на рис. 11.3 их нет. Подобного рода упущения во время оценки случаются сплошь и рядом, и в этом смысле значительную важность представляет один из шагов третьей операции, во время которого оценщики, пытаясь лучше понять архитектуру, задают вопросы о несоответствиях на диаграммах. Аналогичное представление периода прогона для OLTM, где любую транзакцию можно проследить в масштабе всей системы, показано на рис. 11.4; здесь,

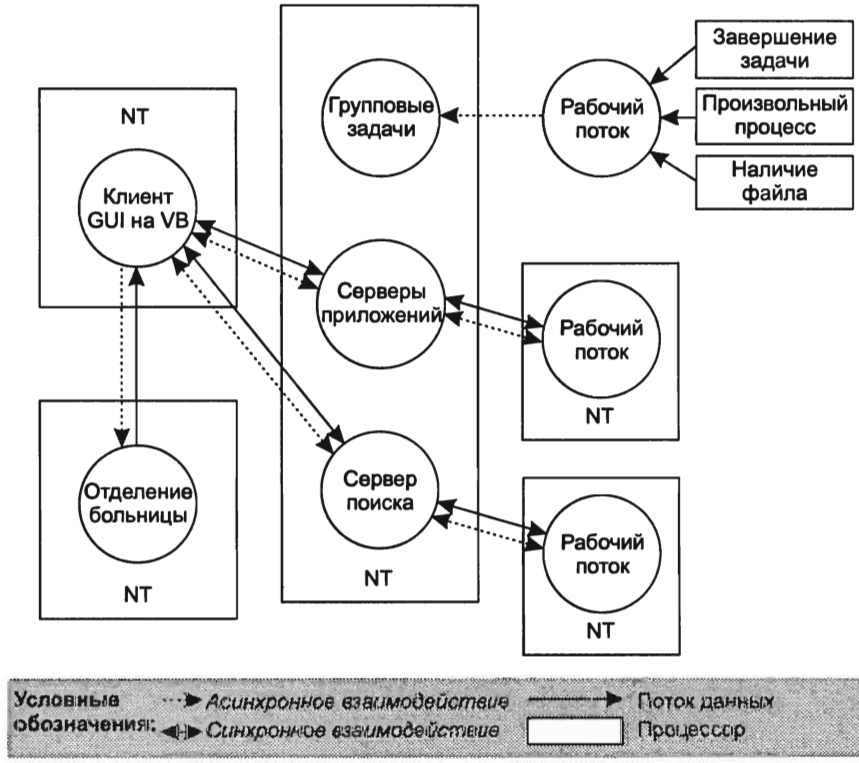


Рис. 11.3. Представление, изображающее передачу информации, потоки данных и процессоры OLTM

опять же, замечаем ряд несоответствий, которые на этот раз связаны с отсутствием расшифровки значения стрелок. По нашему заключению, эти стрелки также изображают потоки данных.

Все перечисленные представления системы Nightingale в равной степени разумны и несут важную смысловую нагрузку. На них изображены отдельные аспекты, значимые в контексте различных задач. Все они впоследствии были задействованы при проведении аналитических действий в рамках ATAM.

Операция 4: выявление архитектурных методик

Ознакомившись с презентацией архитектуры, участники группы оценки составили список всех упомянутых в ней архитектурных методик и дополнили его теми, о которых услышали в ходе обзора документации перед началом оценки. Получилось, что основные методики таковы:

- многоуровневая организация, в особенности в OLTM;
- объектно-ориентированная технология;
- реализация модифицируемости через конфигурационные классы без записи или перекомпиляции;
- обработка транзакций по схеме «клиент–сервер»;

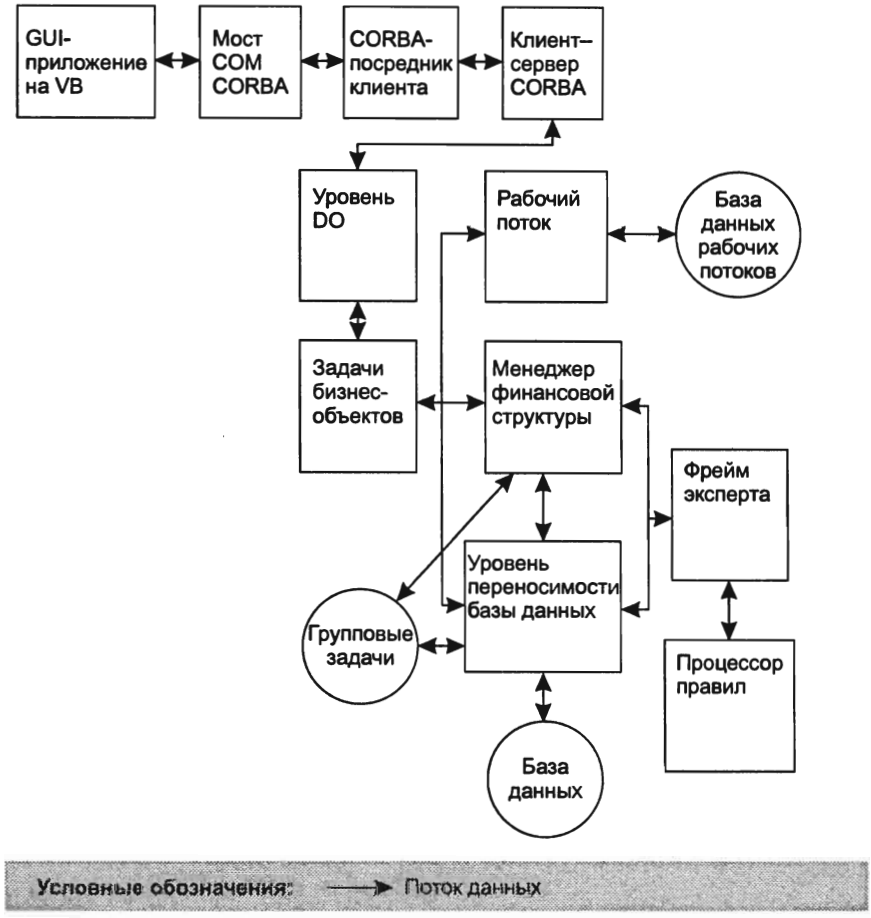


Рис. 11.4. Архитектурная проекция потока данных внутри OLTM

- информационно-ориентированный архитектурный образец с коммерческой базой данных в качестве основы.

Эти и другие методики составили концептуальную основу, исходя из которой, приступив к анализу сценариев, участники группы оценки задавали испытательные вопросы.

Операция 5: генерация дерева полезности атрибутов качества

Дерево полезности, составленное во время проверки системы Nightingale по методу ATAM, представлено в табл. 11.5. Обратите внимание — на нем присутствуют все атрибуты качества, установленные в ходе операции 2; более того, они уточнены до одного или нескольких конкретных значений.

Некоторые уточнения атрибутов качества остались без связанных сценариев. В этом нет ничего необычного и страшного. Иногда люди измышляют разумное с первого взгляда уточнение, но, столкнувшись с необходимостью его конк-

ретизации в контексте собственной системы, обнаруживают его несостоятельность.

Для того чтобы все участники группы оценки могли постоянно сверяться с деревом полезности, секретарь по результатам записывал имя каждого атрибута качества на отдельный лист лекционного плаката и наклеивал его на стену. Впоследствии, по мере уточнения каждого отдельного атрибута и его конкретизации сценариями, секретарь фиксировал всю необходимую информацию на этом и наклеенных ниже лекционных плакатах[1].

Таблица 11.5. Дерево полезности, составленное в ходе оценки системы Nightingale по методу ATAM, в табличном выражении

Атрибут качества	Уточнение атрибута	Сценарии
Производительность	Время отклика на транзакцию	В ответ на уведомление о смене адреса при пиковой нагрузке на систему пользователь обновляет учетную запись пациента; транзакция завершается менее чем за 0,75 секунды (B,C)
		В ответ на уведомление о смене адреса при нагрузке на систему, в два раза превышающей текущую пиковую, пользователь обновляет учетную запись пациента; транзакция завершается менее чем за 4 секунды (H,C)
	Пропускная способность	При пиковой нагрузке системе удается проводить 150 нормализованных транзакций менее чем за секунду (C,H)
	Составление отчетов	Предложений по сценариям не последовало
Практичность	Повышение квалификации	Новый сотрудник с одним или двумя годами опыта работы в отрасли осваивает базовые функции системы менее чем за неделю (C,H)
		Пользователь, находясь в определенном контексте, запрашивает справку и получает ее (B,H)
	Нормальный режим работы	Больничный кассир запускает для пациента, с которым одновременно общается, план выплат; процесс завершается без каких-либо задержек по вине системы (C,C)
Конфигурируемость		Больница повышает цену на отдельную услугу. Группа конфигураторов вносит в систему соответствующие изменения, затрачивая не более одного рабочего дня и не затрагивая исходный код (B,H)
Удобство сопровождения		Обнаружив недостатки, связанные с поиском и временем отклика, специалист по сопровождению исправляет ошибку и распространяет исправление (B,C)

продолжение

[1] Кроме того, мы пробовали составлять деревья полезности в оперативном режиме — заполняли таблицы, подобные табл. 11.5, и проецировали их непосредственно с компьютера. При таком подходе дерево легче составлять и корректировать, однако доступным для просмотра участниками остается содержание одного-единственного экрана. В то же время обзор дерева во всей его полноте стимулирует умственную деятельность и помогает находить бреши. Программные системы для совместной работы будто бы идеально подходят для подобных ситуаций, однако по простоте, надежности и экономичности им сложно тягаться с лекционными плакатами и клейкой лентой.

Таблица 11.5 (*продолжение*)

Атрибут качества	Уточнение атрибута	Сценарии
		Требование по отчетности подразумевает внесение изменений в метаданные составления отчетов (C,H)
		Производитель базы данных выпускает ее новую версию, на установление которой затрачивается минимальное количество времени (B,C)
Расширяемость	Введение нового продукта	Создается новый продукт, отслеживающий доноров банка крови (C,C)
Безопасность	Конфиденциальность	Полномочия физиотерапевта позволяют ему просматривать содержимое регистрационной карточки пациента, связанное с ортопедическим лечением; остальные элементы карточки, равно как и финансовая информация, для него закрыты (B,C)
	Целостность	Система успешно противостоит попыткам несанкционированных вторжений (B,C)
Готовность		Выпущенное производителем баз данных новое программное обеспечение вводится в систему методом горячей замены (B,H)
		Система обеспечивает возможность ежедневного круглосуточного доступа пациентов к своим учетным записям через Интернет (H,H)
Масштабируемость	Расширение системы	Первый заказчик системы приобретает компанию, масштаб которой превышает его собственные показатели в три раза; при этом требуется провести разбиение базы данных (H,B)
		Первый заказчик системы продает другой компании одно из своих подразделений (H,C)
		Первый заказчик системы проводит слияние двух своих подразделений (H,C)
		Компания-разработчик желает продать ряд компонентов системы Nightingale (C,H)
Модульность	Функциональные подмножества	Требуется сконструировать систему таким образом, чтобы она могла работать автономно и при этом предоставлять свою базовую функциональность (C,H)
	Гибкость в отношении замены коробочных продуктов	Замена коммерческой базы данных одного производителя коммерческой базой данных другого производителя (B,C)
		Замена операционной системы (B,C)
		Замена уровня переносимости базы данных (B,C)
		Замена диспетчера транзакций (B,C)
		Замена механизма автоматизации документооборота (B,C)
		Замена коммерческого бухгалтерского пакета (B,C)
		Замена базы данных Solaris, размещенной на платформах Sun (B,C)
		Замена процессора правил (B,C)

Атрибут качества	Уточнение атрибута	Сценарии
Способность к взаимодействию		Требуется реализовать в системе интерфейс с эпидемиологической базой данных, размещенной в нескольких национальных центрах контроля заболеваний (С,С)
Контролепригодность		
Удобство поддержки		

Сценарии в табл. 11.5 сопровождаются указанием приоритетов, распределенных участниками из числа ответственных лиц. В каждой упорядоченной паре символ до запятой обозначает значимость соответствующей возможности; символ после запятой отражает оценку архитектора относительно сложности ее реализации.

Обратите внимание — одни сценарии без труда формулируются исходя из ранее полученных данных, в других отсутствуют стимулы, в третьих — реакция. Неточности в спецификациях сценариев на данном этапе допустимы — главное, чтобы заинтересованные лица понимали, о чем идет речь. Если сценарий отбирается для проведения анализа, его, конечно, следует снабдить явными стимулом и реакцией.

Операция 6: анализ архитектурных методик

Сценарии с приоритетом (В,В) — иначе говоря, отличающиеся высокой значимостью и трудностью реализации и в связи с этим заслуживающие особого внимания в ходе анализа — на дереве полезности отсутствуют. По этой причине мы перешли к поиску сценариев с приоритетом (В,С). Оказалось, что они в изобилии представлены в группе атрибута «модульность» и связаны с заменой в составе системы разного рода коробочных продуктов. Несмотря на то что эти продукты были введены в систему целенаправленно, в соответствии со стратегией снижения рисков разработки для руководителей проекта это решение вылилось в нескончаемую головную боль — ведь все вело к тому, что система (а значит, и ее покупатели) подпадет под влияние многочисленных производителей. Следовательно, в их собственных интересах было реализовать гибкость архитектуры, которая позволила бы без труда заменять коробочные продукты.

Вместе с архитектором мы проработали несколько сценариев. На каждый ушло в среднем полчаса[1]. Поскольку все они оказались связаны с внесением изменений, мы резонно поинтересовались их диапазоном и воздействием. Выяснили мы следующее.

- Замена одной коммерческой базы данных другой, приобретенной у нового производителя, обещала стать трудным занятием. В масштабе всей системы использовался один из диалектов SQL (надмножество SQL по стандарту ANSI), специфичный для производителя текущей базы данных, а также ряд не менее специализированных инструментов и компонентов. По мнению архитектора, замена базы данных была бы крайне маловероятна, в связи

[1] На анализ первого сценария всегда уходит больше всего времени — в некоторых случаях в три раза больше, чем в среднем.

с чем он не брал в расчет высокую стоимость перехода на новую систему. Впрочем, руководитель проекта не выказал столь же твердой уверенности в невозможности развития событий по рассматриваемому сценарию. Таким образом, мы зафиксировали первый установленный на основе анализа архитектурный риск: «Поскольку в системе Nightingale используются специфичные для конкретного производителя инструменты и компоненты, а также диалект SQL, не поддерживаемый или несовместимый с базами данных других производителей, произвести замену базы данных будет очень сложно и отнюдь не дешево — весь процесс займет несколько человеко-лет». Архитектурное решение о связывании архитектуры с конкретной базой данных мы обозначили как точку чувствительности с отрицательным воздействием на модифицируемость.

- Заменить одну операционную систему другой было бы довольно просто. На стороне сервера операционная система была выделена на отдельный уровень, что способствовало локализации изменений. С другой стороны, подсистема OLTM напрямую зависела от средств аутентификации NT, а значит, от новой операционной системы требовались аналогичные свойства. Что касается DSRGM, то здесь все зависимости от операционной системы были исключены на уровне исходного кода — дело в том, что разработка этой подсистемы проводилась на платформе Windows NT, а размещена она была на платформе UNIX; отсюда очевидный вывод о ее полной независимости от операционной системы. В связи с этим мы зафиксировали первое нерискованное решение: «Поскольку зависимости от операционной системы в подсистемах OLTM и DSRGM локализованы или исключены, введение новой операционной системы вместо старой не предполагает сколько-нибудь серьезных модификаций». Инкапсуляцию зависимостей от операционной системы мы отметили как точку чувствительности с положительным воздействием на модифицируемость.
- Мы установили ряд проблем, связанных с внесением изменений в процессор правил. Считать такой сценарий надуманным нет никаких оснований — как мы выяснили, текущий процессор не отличался высокими характеристиками производительности и удобства сопровождения. Вероятнее всего, впоследствии его бы просто исключили, а правила реализовали бы средствами C++. Поскольку прямое построение цепочек правил не разрешалось (что было вполне разумно и объяснялось желанием отложить соответствующее решение на более поздний период), правила в силу их процедурного характера можно было скомпилировать. Подобная модификация, скорее всего, привела бы к нескольким серьезным последствиям.
  - ◊ Повышение производительности (хотя бесспорного решения эта проблема на тот момент еще не получила).
  - ◊ Устранение необходимости в обучении сотрудников языку правил и объяснению им принципа действия процессора.
  - ◊ Группе разработчиков не пришлось бы создавать полезные правила и среду моделирования.

- ◊ Правила, возможно, «закопались» бы в код C++ и сплелись с функциональным кодом, не имеющим прямого отношения к правилам; отсюда — затруднения по части их обнаружения и сопровождения.
- ◊ Возможность размещения в правилах ссылок на несуществующие объекты, вероятно, удалось бы исключить. В современном варианте эта возможность, обусловливавшая возникновение ошибок, существовала и имела серьезные шансы, просочившись через тестирование, попасть в производственную систему. Путем составления правил средствами C++ подобные ошибки можно было бы устранить уже в период компиляции.

В расчете на подобные изменения можно было написать генератор правил в виде кода C++. Против такого решения свидетельствовали существенный объем работ и их сложность, остававшаяся неизвестной. Итак, для рассмотренного сценария мы зафиксировали риск, связанный с трудностью исключения процессора правил. Сам факт применения этого процессора (в противоположность коду C++) мы обозначили как точку компромисса в архитектуре — ведь упрощение разработки и внесения изменений в базу правил достигалось за счет снижения производительности, необходимости в подготовке персонала и усложнения тестирования.

И так далее. Впоследствии, в рамках того же сценария, мы проанализировали замену коммерческого блока веб-хостинга, коммерческого бухгалтерского пакета, механизма автоматизации документооборота и операционной системы Solaris на платформах Sun.

На этом совещание, проводившееся в рамках первого этапа, завершилось. Нам удалось зафиксировать шесть точек чувствительности, одну точку компромисса, четыре рискованных и пять нерискованных решений.

Этап 2: оценка (продолжение)

На втором этапе, выдержав двухнедельный перерыв, мы вновь созвали совещание. Во время этого перерыва участники группы оценки занимались составлением сводного отчета — точнее, тех его частей, которые уже можно было закрывать. В частности, они описали коммерческие факторы и архитектуру согласно ее презентации, привели список методик, изобразили дерево полезности и изложили результаты анализа, проведенного на первом этапе. Кроме того, мы несколько раз созванивались с архитектором, пытаясь уточнить некоторые технические вопросы, и с руководителем проекта, который должен был обеспечить адекватное представительство заинтересованных лиц во время проведения второго этапа.

На втором этапе, помимо ответственных лиц, участвовавших в проведении предыдущего этапа, присутствовали десять новых заинтересованных лиц: разработчики, специалисты по сопровождению, представители первого заказчика и два конечных пользователя.

В первую очередь, специально для новых участников, мы повторили первую операцию (описание метода ATAM), а для того чтобы привести знания всех присутствующих к единому знаменателю, рассказали о результатах первого этапа. После этого мы приступили к выполнению операций 7, 8 и 9.

Операция 7: мозговой штурм и расстановка сценариев согласно приоритетам

Заинтересованные лица поработали на славу — всего в ходе этой операции они огласили 72 сценария. Более десятка из них были отражены на листьях дерева полезности, которое мы составили во время операции 5, но на первом этапе до их анализа дело не дошло. Это вполне нормально, даже полезно. Таким способом заинтересованные лица давали нам понять, что некоторые сценарии заслуживают большего внимания, чем то, которое они получили на первом этапе.

В табл. 11.6 приводятся наиболее интересные из всех сценариев, зафиксированных в ходе операции 7. Одни сформулированы на редкость удачно, иные, напротив, не слишком очевидны. Учитывая стихийный характер мозгового штурма, предполагающего активное участие всех присутствующих, в этом нет ничего страшного. Чем тратить ценное время на вылизывание каждого сценария, мы предпочитаем записывать высказываемые соображения сразу, как только они появляются. Если какой-то сценарий перед голосованием или анализом требуется откорректировать, мы с готовностью это сделаем (естественно, не без участия того человека, который его предложил).

Таблица 11.6. Сценарии, полученные методом мозгового шторма

Номер	Сценарий
1	Данные, ранее бывшие общедоступными, сделаны приватными; соответствующие изменения внесены в права доступа
2	Данные, взятые из информационного ядра, сдублированы в одном из отделений клиники, в результате чего снизилась производительность
3	При запуске правила в процессоре доступ к данным осуществляется слишком медленно
4	Пользователь регистрирует проведенный пациентом платеж в момент сильной загрузки системы, в результате чего замедляется реакция (происходит это в тестовой среде)
5	Пользователю, находящемуся в одном подразделении, требуется выполнить операции от имени других подразделений
6	Принято решение о поддержке немецкого языка
7	В систему вводится роль эпидемиолога и соответствующая функциональность
8	Система Nightingale устанавливается в кабинете с пятью врачами и адаптируется под условия заказчика
9	Для подачи асинхронных запросов пользователю требуется новое поле
10	Получив жалобу, сотрудники больницы обнаруживают, что на протяжении последних шести месяцев подкладываемые судна оценивались неверно
11	Больнице требуется централизовать процесс обслуживания регистрационных карточек в масштабе нескольких отделений; при этом все сопутствующие бизнес-процессы подвергаются реконструкции
12	Менеджер хочет составить отчет по истории просроченных платежей и пеням, выставленным пациентам, проходившим лечение от порезов и ран
13	«Условный» (what-if) сценарий: Приложение к учетной записи предполагаемой поправки в закон
14	Дефект, приведший к искажению данных, не удалось обнаружить до следующего цикла отчетности

Номер	Сценарий
15	При установке системы Nightingale в больнице необходимо преобразовать существующую в ней базу данных
16	В результате ошибки в процессе репликации база данных транзакций рассинхронизируется с резервной базой данных
17	Системная ошибка приводит к невозможности зачисления платежей на счета, находящиеся в Аризоне
18	Контрольный журнал транзакций не пополняется на протяжении трех дней (как исправить ошибку?)
19	Одно из отделений меняет продолжительность рабочего дня и месяца
20	Получение информации о зачислении платежа от системы управления базами данных страховой компании (при помощи определения метаданных)
21	Введение нового технологического процесса входной и выходной регистрации пациентов
22	Пакетные процессы инициируются на временной и событийной основах
23	Неисправность главной магистрали передачи данных от информационного ядра к филиалам клиники
24	Сервер базы данных одного из отделений клиники не загружается
25	При составлении отчета требуется получить информацию из двух больниц, в каждой из которых используется индивидуальная конфигурация
26	Центр денежных переводов дважды подает одну и ту же группу платежей, причем операции начинаются только после второй подачи
27	Терапевт, специалист по реабилитации, переводится в другую больницу; при этом ему требуется доступ к историям болезни бывших пациентов с полномочиями чтения
28	Согласованное распределение ряда изменений среди нескольких узлов в учреждениях здравоохранения (формы и конфигурации)
29	В результате пожара в информационном центре информационное ядро приходится перевести в другое место
30	Больница переуступает большое количество счетов, выставленных на другое подразделение
31	Изменение правил составления предупреждений о взаимоисключающих лекарственных препаратах
32	Пользователь в бухгалтерии больницы хочет перейти из режима распечатки выходных данных в режим их оперативного просмотра
33	Телефонная компания меняет междугородный телефонный код
34	Администратор счетов предумышленно перевел с них небольшие суммы на счета своих друзей. Как выявить нарушителя и определить масштаб злоупотреблений?

После того как мы провели слияние нескольких почти идентичных сценариев, заинтересованные лица проголосовали. Каждому из них мы выделили по 22 голоса (30 % от 72 сценариев плюс округление до ближайшего целого числа), которые следовало подавать в два захода. Подсчитав голоса, мы вместе с участниками группы оценки на протяжении получаса дополняли составленное во время операции 5 дерево полезности новыми высокоприоритетными сценариями, которых набралось всего около десятка. Все сценарии с высоким приоритетом, выявленные в ходе операции 7, мы без лишних раздумий поместили на дерево полезности в виде новых листьев существующих ветвей. Этим мы хотели показать, что

представления архитектора и заинтересованных лиц о важнейших атрибутах качества оказались сходными.

Разобравшись с согласованием отдельных сценариев на дереве полезности, мы приступили к анализу тех из них, которые получили наибольшее число голосов.

Операция 8: анализ архитектурных методик

В ходе операции 8 мы дополнительно провели анализ семи сценариев; следует заметить, что по меркам ATAM это довольно много. Учитывая ограниченность объема книги, мы рассмотрим результаты одного-единственного, 15-го сценария (см. соответствующую врезку).

Операция 9: презентация результатов

Девятая операция предполагает проведение одно- или двухчасовой презентации с изложением всех достигнутых результатов и полученных выводов. В начале презентации необходимо показать слушателям набор стандартных слайдов с основными тезисами метода; кроме того, следует подготовить несколько чистых слайдов-шаблонов, на которые впоследствии нужно будет нанести сводные данные по коммерческим факторам и архитектуре, перечень методик, дерево полезности, сведения об анализе сценариев и список результатов его проведения.

На втором этапе участники группы оценки должны по вечерам составлять сводки достигнутых за день результатов. Кроме того, при составлении графика второго этапа перед операцией 9 следует выделить дополнительное время, для того чтобы участники смогли встретиться и завершить работу над пакетом результатов.

Помимо рискованных и нерискованных решений, точек чувствительности и компромиссов, участники группы должны выявить магистральные риски — факторы, систематически оказывающие на архитектуру негативное воздействие. Это единственный вид результатов, с которыми участники еще не имели дела (и, соответственно, не помогали их устанавливать). Значение каждого из них мы стараемся излагать так, чтобы его понял заказчик, — в частности, мы указываем на те коммерческие факторы, реализацию которых те или иные магистральные риски ставят под сомнение.

СЦЕНАРИЙ 15: ПРИ УСТАНОВКЕ СИСТЕМЫ NIGHTINGALE В БОЛЬНИЦЕ НЕОБХОДИМО ПРЕОБРАЗОВАТЬ ЕЕ СУЩЕСТВУЮЩУЮ БАЗУ ДАННЫХ

Нет ничего удивительного в том, что проработке этого сценария архитектор уделил особое внимание — ведь от его реализации зависел успех системы в целом. Процедуру, ранее документированную, он изобразил для нас на белой доске.

Как часто бывает, анализ этого сценария помог нам лучше разобраться в архитектуре. По понятным причинам архитектор не стал останавливаться на проблеме преобразования базы данных во время презентации (операция 3), посчитав эту информацию служебной.

Результаты анализа процесса миграции убедили участников группы оценки в том, что процедура тщательно продумана, ее преимущества известны, а ограничения обоснованны. Тому, что архитектор не упомянул об этом процессе во время презентации (операция 3), мы нисколько не удивились. Неожиданность заключалась в другом: в пакете документации, который мы получили и изучили еще до начала первого этапа, о нем также ничего не было сказано. В ответ на наше недоумение архитектор признал, что процедура еще не документирована, тем самым предоставив нам повод зафиксировать риск. Впрочем, она компенсировалась нерискованным решением, которое мы сформулировали так: «Архитектура предусматривает простые и эффективные средства преобразования и миграции данных, значительно облегчающие размещение системы Nightingale».

Магистральных рисков для Nightingale мы выделили всего три.

1. *Избыточная зависимость от конкретных коробочных продуктов.* В связи с этим мы сообщили о трудностях замены базы данных и исключения процессора правил, а также о зависимости от старой и (вероятно) более не поддерживаемой версии уровня переносимости базы данных. Данный магистральный риск представляет угрозу для коммерческого фактора удобства сопровождения.
2. *Неполная определенность процесса восстановления после ошибок. Недостаточный уровень осведомленности заказчика о доступных инструментальных средствах.* Некоторые сценарии касались обнаружения и устранения ошибок в базе данных. Несмотря на то что соответствующие процедуры были предусмотрены в архитектуре, о некоторых из них архитекторы и проектировщики задумывались впервые. По словам представителей первого заказчика, процедур, направленных на исправление ошибок, у них не было (ни собственных, ни полученных от компании-разработчика). Этот магистральный риск ставил под сомнение реализацию коммерческого фактора практичности и обеспечения работы предприятия заказчика.
3. *Вопросы, связанные с документацией.* Документация проекта Nightingale оказалась в неудовлетворительном состоянии. К осознанию этого недостатка группа оценки пришла уже во время проводившегося перед первым этапом совещания, а сценарии, проанализированные на втором этапе, только утвердили нас в таком мнении. Несмотря на наличие ряда объемных и детальных элементов документации (построенной, в частности, средствами UML и модели Rose), ни вступительной части, ни обзора архитектуры составлено не было — а ведь без этого невозможно проводить подготовку персонала, внедрять в проект новых специалистов, сопровождать систему, координировать дальнейшую разработку и тестирование. Недокументированными также оставались обширная база правил, регламентировавшая поведение Nightingale, а также процедура преобразования и миграции данных. В отсутствие этой документации первый заказчик, уже готовый приобрести систему, не смог бы справиться с ее сопровождением; таким образом, риск распространялся на один из основных коммерческих факторов разработки Nightingale — обеспечение функционирования предприятия заказчика.

Этап 3: доработка

Осязаемым продуктом оценки по методу ATAM является сводный отчет с перечнем рискованных и нерискованных решений, точек чувствительности и точек компромиссов. Помимо этого, в него включается каталог применяемых архитектурных методик, дерево полезности, сценарии, сформулированные методом мозгового штурма, и описание анализа всех отобранных сценариев. Наконец, в сводном отчете указываются магистральные риски, которые удалось выявить участникам

группы оценки, и коммерческие факторы, реализацию которых они ставят под сомнение.

Как и при презентации результатов, при составлении отчета мы воспользовались стандартным шаблоном, многие из разделов которого уже были заполнены (в частности, раздел, посвященный описанию метода ATAM); остальные нам лишь предстояло заполнить. Отдельные части сводного отчета — в частности, дерево полезности и результаты анализа операции 6 — мы подготовили во время перерыва между первым и вторым этапами. Все эти подготовительные действия привели к положительному результату: если раньше на подготовку сводного отчета для заказчика оценки ATAM уходило около двух недель, то в рассматриваемом случае нам удалось составить качественный, комплексный отчет приблизительно за два дня.

11.5. Заключение

ATAM зарекомендовал себя как надежный метод оценки программной архитектуры. Он подразумевает формулирование (в форме сценариев) ответственными за проект и заинтересованными лицами точного перечня требований по атрибутам качества и исследование архитектурных решений, значимых в контексте реализации всех высокоприоритетных сценариев. Путем разделения такого рода решений на рискованные и нерискованные оценщикам удается выявить проблемные участки рассматриваемой архитектуры.

Одного лишь знания принципов, на которых основывается метод ATAM, недостаточно. Важно понимать, для каких целей он *не* предназначен.

- ATAM не предусматривает оценки требований. Другими словами, по результатам оценки, проведенной этим методом, нельзя судить о перспективах реализации *всех* предъявленных к системе требований. Он лишь помогает понять, удастся ли при данном проектном решении реализовать основные требования.
- ATAM не предоставляет возможности оценки кода. Поскольку оценка по этому методу проводится на ранних стадиях жизненного цикла, никаких допущений относительно существования кода не делается, а средств инспекции кода не предусматривается.
- ATAM не связан с фактическим тестированием системы. Так как, опять же, оценка ATAM проводится на раннем этапе жизненного цикла, допущения о существовании системы и средств фактического тестирования в рассматриваемый метод не заложены.
- Не являясь точным инструментом, ATAM выделяет в рамках архитектуры потенциальные области риска. Выражаются они в виде точек чувствительности и точек компромиссов. Поскольку ATAM основывается на знаниях архитектора, некоторые риски могут так и остаться неустановленными. Те же риски, которые удается выявить, в рамках ATAM не подлежат количественной оценке. Другими словами, оценщики не делают выводов об убытках, которые могут последовать по причине неустранения той или иной

точки чувствительности. Финансовые аспекты рассматриваются в главе 12 в связи с методом анализа стоимости и эффективности (cost benefit analysis method, CBAM).

Оценок, проведенных по методу ATAM, на нашем счету множество; нам также доводилось наблюдать за тем, как их выполняют другие специалисты, и даже проводить посвященные этому методу занятия. Практически в каждом из этих случаев мы сталкивались с одной и той же реакцией со стороны технических специалистов — они крайне удивлялись, узнав, что за относительно короткое время можно выявить немалое количество рисков. Ответственным лицам удавалось понять, почему реализации поставленных коммерческих задач препятствуют те или иные технические проблемы. Итак, ATAM оказался весьма полезной штукой.

11.6. Дополнительная литература

В то время, когда книгу, которую вы держите в руках, готовили к печати, мы занимались выверкой начального варианта учебного курса по ATAM. Подробности этого предприятия изложены на веб-сайте анализа компромиссных архитектурных решений на сервере Института программной инженерии — http://www.sei.cmu.edu/ata/ata-init.html. Более подробное исследование ATAM, сопровождаемое конкретным примером оценки спутниковой системы данных NASA, приводится в работе [Clements 02a].

Довольно необычно требования по атрибутам качества и их связь с проектными решениями трактуются в публикации [Chung 00]. В частности, в ней дается ссылка на издание [Boehm 76]. Описанное в нем дерево характеристик качества программных продуктов во многом схоже с деревьями полезности, применяемыми в рамках ATAM.

Если вам интересны исторические предпосылки возникновения ATAM и при этом вы не прочь ознакомиться со вторым (более простым) методом архитектурной оценки, прочитайте работу [Kazman 94], посвященную методу анализа программной архитектуры (software architecture analysis method, SAAM).

11.7. Дискуссионные вопросы

1. Проанализируйте одну из важных программных систем вашей компании. Помогает ли изложенный в этой главе шаблон по части формулирования коммерческих факторов и обсуждения архитектуры в целом? Если нет, какой информации недостает? Попробуйте набросать дерево полезности рассматриваемой системы.
2. Предположим, что вы решили провести оценку архитектуры этой системы. Кого вы привлечете к участию в процессе? Какие роли должны будут исполнять заинтересованные лица и как эти роли среди них лучше всего распределить?

Глава 12
Метод анализа стоимости и эффективности — количественный подход к принятию архитектурно-проектных решений

(в соавторстве с Джей Асунди[1] и Марком Кляйном)

> Тут миллиард, там миллиард — в конечном итоге получаются приличные деньги.
>
> *Эверетт Дирксен, сенатор США (1896–1969)*

Как вы знаете на материале главы 11, метод анализа компромиссных архитектурных решений (Architecture Tradeoff Analysis Method, ATAM) позволяет архитекторам программных систем оценивать технологические компромиссы, на которые они идут при проектировании и сопровождении. Основным предметом анализа в рамках ATAM является соотношение проектного решения архитектуры — существующей или нереализованной — и значимых с точки зрения заинтересованных лиц атрибутов качества. Кроме того, исследованию подвергаются архитектурные компромиссы — точки, в которых одно решение влияет на реализацию нескольких атрибутов качества.

При всем при этом ATAM не учитывает одного важного обстоятельства — как правило, наиболее значительные компромиссы в сложных системах оказываются завязанными на экономические факторы. Как компании следует распределить ресурсы, чтобы максимизировать доходы и минимизировать риски? Раньше этот вопрос решался в основном исходя из стоимости конструирования системы, при-

---

[1] Джей Асунди (Jai Asundi) занимается преподавательской работой в Техасском университете (Даллас).

чем долговременные издержки, связанные с прохождением циклов сопровождения и обновления, в расчет обычно не принимались. Не менее (а возможно, и более) важны *выгоды*, которые приносит компании то или иное архитектурное решение.

Учитывая ограниченность ресурсов, применяемых при конструировании и сопровождении системы, явственно ощущается потребность в некоем рациональном процессе, облегчающем процесс выбора архитектурных альтернатив на этапе первоначального проектирования и в последующие периоды обновления. Альтернативы эти различаются по издержкам, потреблению ресурсов и реализации характеристик (каждая из которых приносит компании те или иные выгоды); кроме того, выбор сам по себе — предприятие в некоторой степени рискованное и неясное. Для выявления всех этих аспектов необходимы *экономические* модели программных систем, учитывающие издержки, выгоды, риски и временные ограничения.

Имея в виду упростить принятие решений экономического характера, мы разработали метод экономического моделирования программных систем, ориентированный на анализ вариантов их архитектуры. Известный под названием метода анализа стоимости и эффективности (Cost Benefit Analysis Method, CBAM) и базирующийся на ATAM, он обеспечивает моделирование затрат и выгод, связанных с принятием архитектурно-проектных решений, и способствует их оптимизации. Методом CBAM оцениваются технологические и экономические факторы, а также сами архитектурные решения.

12.1. Контекст принятия решений

Все программные архитекторы и ответственные лица стремятся довести до максимума разницу между выгодами, полученными от системы, и стоимостью реализации ее проектного решения. Являясь логическим продолжением метода ATAM, CBAM основывается на его артефактах. Контекст CBAM изображен на рис. 12.1.

Поскольку архитектурные стратегии ограничиваются разнообразными техническими и экономическими факторами, стратегии, применяемые программными архитекторами и проектировщиками, должны быть поставлены в зависимость от коммерческих задач программной системы. Прямой экономический фактор — это стоимость реализации системы. Техническими факторами являются характеристики системы — другими словами, атрибуты качества. У атрибутов качества также есть экономический аспект — выгоды, получаемые от их реализации.

Как вы помните, по результатам оценки программной системы по методу ATAM в нашем распоряжении оказался ряд документированных артефактов.

- Описание коммерческих задач, определяющих успешность системы.
- Набор архитектурных представлений, документирующих существующую или предложенную архитектуру.
- Дерево полезности, выражающее декомпозицию задач, которые заинтересованные лица ставят перед архитектурой, — от обобщенных формулировок атрибутов качества до конкретных сценариев.

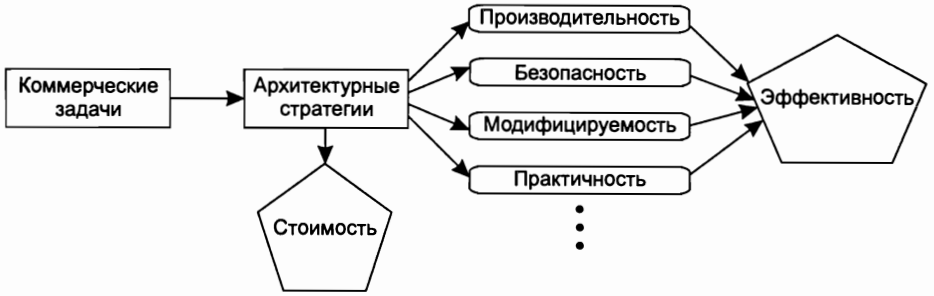


Рис. 12.1. Контекст метода анализа стоимости и эффективности (CBAM)

- Ряд выявленных рисков.
- Ряд точек чувствительности (архитектурных решений, которые оказывают влияние на отдельный показатель атрибута качества).
- Ряд точек компромиссов (архитектурных решений, которые воздействуют сразу на несколько показателей атрибута качества, причем на одни положительно, а на другие отрицательно).

ATAM помогает выявить ряд основных архитектурных решений, значимых в контексте сформулированных заинтересованными лицами сценариев атрибутов качества. Эти решения приводят к реакции со стороны атрибутов качества — точнее говоря, отдельных уровней готовности, производительности, безопасности, практичности, модифицируемости и т. д. С другой стороны, каждое архитектурное решение связано с определенными издержками (стоимостью). К примеру, достижение желаемого уровня готовности путем резервирования аппаратных средств подразумевает один вид издержек, а регистрация в файлах на диске контрольных точек — другой. Эти архитектурные решения приводят к (предположительно разным) измеримым уровням готовности, имеющим определенную ценность для компании – разработчика системы. Возможно, ее руководство полагает, что заинтересованные лица заплатят бóльшую сумму за систему с высокой готовностью (если, к примеру, это телефонный коммутатор или программное обеспечение для медицинского наблюдения), или боится погрязнуть в судебных разбирательствах в случае отказа системы (вполне разумно, если речь идет о программе управления антиблокировочной тормозной системой автомобиля).

ATAM обнаруживает архитектурные решения, принятые относительно рассматриваемой системы, и устанавливает их связь с коммерческими задачами и количественной мерой реакции атрибутов качества. Принимая эти данные на вооружение, CBAM помогает выявить связанные с такими решениями издержки и выгоды. Основываясь на этой информации, заинтересованные лица могут принять окончательные решения относительно резервирования аппаратной части, введения контрольных точек и всех прочих тактик, направленных на повышение готовности системы. Вполне возможно, что они предпочтут сконцентрировать ресурсы, которые, как известно, ограниченны, на реализацию какого-то другого атрибута качества — например, на улучшение соотношения выгод и издержек за счет повышения производительности. Из-за ограниченности бюджета разработки и обновления системы каждое архитектурное решение, по большому счету, соревнуется за право на существование со всеми остальными.

Подобно финансовому консультанту, который никогда напрямую не укажет, во что вкладывать деньги, CBAM не заменяет собой решений, принимаемых заинтересованными лицами. Он лишь помогает им установить и документировать издержки и выгоды архитектурных инвестиций, осознать неопределенность этого «портфеля»; на этой основе заинтересованные лица могут принимать рациональные решения, удовлетворяющие их потребности и сводящие к минимуму риски.

Короче говоря, метод CBAM исходит из предположения о том, что архитектурные стратегии (как совокупность архитектурных тактик) оказывают влияние на атрибуты качества системы, а те, в свою очередь, предоставляют заинтересованным лицам некоторые выгоды. Эти выгоды мы называем *полезностью* (utility). Любая архитектурная стратегия отличается той или иной полезностью для заинтересованных лиц. С другой стороны, есть издержки (стоимость) и время, которые необходимо потратить на реализацию этой стратегии. Отталкиваясь от этой информации, метод CBAM помогает заинтересованным лицам в процессе выбора архитектурных стратегий, характеризующихся максимальной *прибылью на инвестированный капитал* (return on investment, ROI), — другими словами, наиболее выгодных с точки зрения соотношения выгод и издержек.

12.2. Основы CBAM

Ниже мы рассмотрим принципы, составляющие основу метода CBAM. Их практическая реализация в виде последовательности этапов описывается в разделе 12.3. Предварительно вы должны разобраться с теоретической стороной расчета коэффициента ROI для различных архитектурных стратегий с учетом отобранных заинтересованными лицами сценариев.

Для начала рассмотрим ряд сценариев, сформулированных в рамках ATAM (или специально для оценки по методу CBAM). Их следует исследовать на предмет различий по ценности предполагаемых реакций, а затем классифицировать полученные результаты по критерию полезности. Полезность определяется значимостью каждого рассматриваемого сценария с учетом предполагаемого значения реакции. Далее рассматриваются архитектурные стратегии, приводящие к различным предполагаемым реакциям. Каждая стратегия характеризуется издержками и воздействием на несколько атрибутов качества. Иначе говоря, архитектурная стратегия, которая изначально реализуется с целью достижения желаемой реакции, попутно оказывает влияние на другие атрибуты качества. Полезность этих «побочных эффектов» необходимо учитывать при расчете общей полезности стратегии — ведь именно из общей полезности, прибавленной к проектной стоимости архитектурной стратегии, складывается окончательная величина ROI.

Полезность

При расчете полезности внимание обращается на проблемы, описанные в нижеследующих разделах.

Вариации сценариев

По аналогии с ATAM, сценарии в CBAM применяются как механизм конкретного выражения и представления отдельных атрибутов качества. Так же как и в ATAM, сценарии здесь разделяются на три части: стимул (взаимодействие с системой), условия (состояние системы в данный момент) и реакцию (результирующий атрибут качества, поддающийся количественной оценке). Впрочем, между упомянутыми методами есть и различия. В CBAM, к примеру, сценарии задействуются целыми наборами (которые составляются путем варьирования значений реакции), в то время как ATAM имеет дело с отдельными сценариями. Отсюда понятие кривой «реакция–полезность».

Кривые «реакция–полезность»

Каждая пара значений стимула–реакции в рамках сценария в какой-то степени полезна для заинтересованных лиц; более того, полезность возможных значений реакции можно сравнивать. К примеру, заинтересованные лица вряд ли оценят максимальную готовность в качестве реакции на отказ значительно выше, чем готовность умеренную. С другой стороны, низкая задержка, очевидно, имеет шансы на значительно более серьезную оценку по сравнению с умеренной задержкой. Любое отношение между набором величин полезности и соответствующим набором величин реакции можно выразить в виде графика, называемого кривой «реакция–полезность». Несколько примеров таких кривых приводятся на рис. 12.2. Точки с метками *a*, *b* и *c* на каждой из них выражают различные величины реакции. Таким образом, полезность на такой кривой изображается как функция от величины реакции.

Кривая «реакция–полезность» демонстрирует изменение величин полезности в зависимости от изменения величин реакции. Как видим на рис. 12.2, полезность может изменяться линейно, нелинейно и ступенчато. К примеру, график (c) демонстрирует значительное повышение полезности при ограниченном изменении уровня реакции атрибута качества, что соответствует вышеприведенному примеру с производительностью. Пример с готовностью лучше сочетается с графиком (a), где умеренное изменение уровня реакции приводит к едва заметному изменению полезности для пользователя.

Допытываться у заинтересованных лиц относительно характеристик полезности долго и утомительно. По этой причине для каждого сценария мы выбрали всего по пять значений реакции атрибута качества, посчитав, что приближенных величин будет вполне достаточно. Четыре из них универсальны для любых архитектурных стратегий, и о них мы поговорим прямо сейчас. Заметим лишь, что пятое значение, рассматриваемое далее по тексту, зависит от конкретной архитектурной стратегии.

Для того чтобы построить кривую «реакция–полезность», в первую очередь необходимо установить уровни атрибута качества в наилучшим и в наихудшем случаях. На уровне наилучшего случая атрибута качества заинтересованные лица усматривают максимальную полезность. К примеру, реакция системы на действия пользователя, происходящая за период времени в 0,1 с, воспринимается как мгновенная; таким образом, сокращение этого периода до 0,03 с бесполезно. Уровень

наихудшего случая атрибута качества — это нижний предел, на котором система *может* функционировать; если он не соблюдается, в глазах заинтересованных лиц система теряет смысл. Эти уровни — наилучшего и наихудшего случая — соответствуют значениям полезности 100 и 0 соответственно.

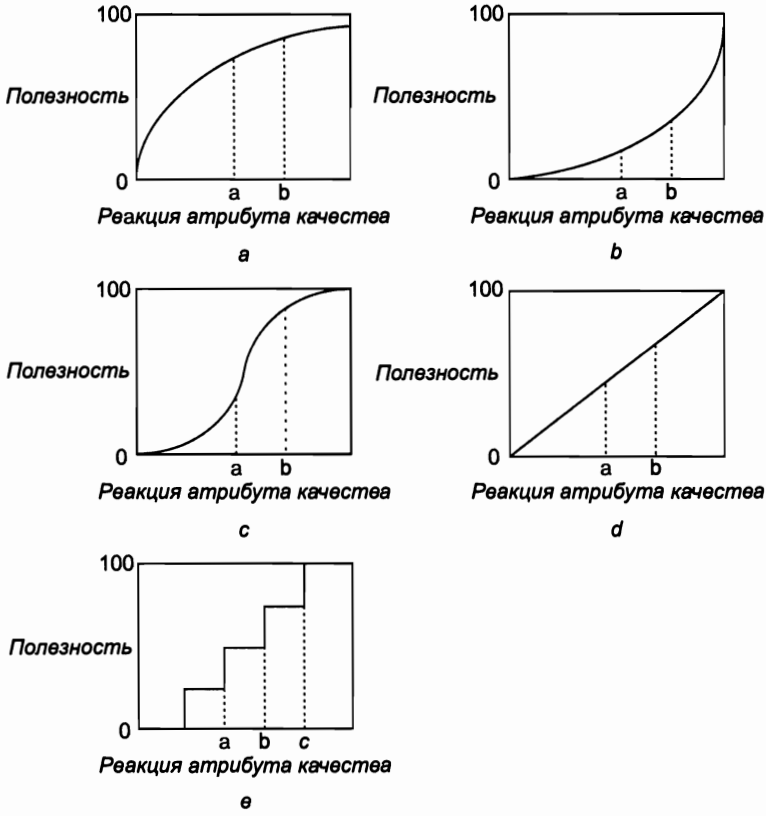


Рис. 12.2. Несколько примеров кривых «реакция–полезность»

Далее, для каждого сценария необходимо определить *текущий* (current) и *желаемый* (desired) уровни полезности. Значения полезности (находящиеся между 0 и 100) текущего и желаемого уровней определяются по показаниям заинтересованных лиц, причем значения наилучшего и наихудшего случаев используются как опорные точки (предположим, что в данный момент полезность считается 50-процентной, а желаемый уровень полезности атрибута качества находится на отметке 90 % максимально возможной полезности; соответственно, текущий уровень полезности приравнивается к 50, а желаемый — к 90). Таким способом кривые составляются для любых сценариев.

Расстановка приоритетов среди сценариев

Различные сценарии, сформулированные в рамках данной системы, имеют для заинтересованных лиц разную важность и, соответственно, характеризуются

разной полезностью. Относительная значимость каждого сценария выражается через его *вес* (weight), назначаемый по результатам двухэтапной процедуры голосования. На первом этапе заинтересованные лица путем подачи голосов упорядочивают сценарии. Исходят они при этом из «предполагаемого» значения реакции. После этого заинтересованные лица присваивают наиболее приоритетному сценарию вес 1, а всем остальным, в зависимости от их относительной значимости, дробные значения.

Если в какой-то момент в будущем появится потребность во введении новых сценариев, им также нужно будет присвоить значение веса. Совещательным путем заинтересованные лица приводят значения веса в согласие со своими представлениями.

Архитектурные стратегии

В обязанности архитектора (или архитекторов) входит выбор архитектурных стратегий, позволяющих перейти от *текущего* уровня реакции атрибута качества к *желаемому* или даже *наилучшему*. Специально для этой цели в рамках CBAM существует отдельный этап. Для каждой стратегии выводятся:

- ожидаемое значение реакции в каждом сценарии (полезность ожидаемого значения определяется путем интерполяции четырех значений, установленных при участии заинтересованных лиц);
- воздействие архитектурной стратегии на другие значимые атрибуты;
- стоимость реализации архитектурной стратегии.

Побочные эффекты

Как правило, архитектурные стратегии влияют на разные атрибуты качества — как относящиеся, так и не относящиеся к текущему сценарию (именно по этой причине существуют архитектурные компромиссы!). Определить полезность *побочных* реакций атрибута, появляющихся в результате применения данной архитектурной стратегии, совершенно необходимо. В крайнем случае следует создать новую версию сценария для побочного атрибута и построить кривую «реакция–полезность». На практике все значимые для заинтересованных лиц атрибуты качества обычно фигурируют сразу в нескольких сценариях, а значит, лишний раз строить кривые «реакция–полезность» не приходится. Единственное, что в таком случае нужно установить, — это ожидаемая полезность, связанная с данным атрибутом качества для данной архитектурной стратегии. Обратите внимание — если архитектурная стратегия задумана для того, чтобы подчеркнуть конфликт одного атрибута с другим — тем, над подсчетом полезности которого в данный момент идет работа, — то ожидаемая полезность может быть отрицательной.

При наличии вышерассмотренной дополнительной информации можно сложить выгоды от применения данной архитектурной стратегии для отдельных значимых атрибутов качества и тем самым установить общие выгоды.

Определение выгод и нормализация

Для того чтобы на основе кривых «реакция–полезность» вычислить общую полезность применения архитектурной стратегии для нескольких сценариев, доста-

точно сложить полезность для каждого из них в отдельности (вес сценариев при этом определяется его значимостью). Так, для любой архитектурной стратегии i выгода B_i вычисляется по формуле:

$$B_i = \sum_j (b_{i,j} \times W_j),$$

где $b_{i,j}$ — это выгода, извлеченная из стратегии i в силу ее воздействия на сценарий j, а W_j — вес сценария j. Каждое значение $b_{i,j}$ на рис. 12.2 выражает изменение полезности сценария, вызванное применением данной архитектурной стратегии: $b_{i,j} = U_{\text{ожидаемая}} - U_{\text{текущая}}$; иначе говоря, оно равняется полезности ожидаемого значения архитектурной стратегии в отношении данного сценария за вычетом текущей полезности системы. Как мы уже говорили, за счет умножения на вес (W_j) полученное значение полезности нормализуется относительной значимостью того или иного сценария.

Вычисление коэффициента ROI

Коэффициент ROI для любой архитектурной стратегии равняется частному от деления общей выгоды (B_j) на издержки (C_i) реализации. Издержки рассчитываются по модели, наиболее подходящей для разрабатываемой системы и ее окружения:

$$R_i = \frac{B_i}{C_i}$$

Исходя из полученного значения архитектурные стратегии подвергаются ранжированию; впоследствии ранги помогают определить оптимальный порядок реализации различных стратегий.

Рассмотрим изображенные на рис. 12.2 кривые (*a*) и (*b*). По мере возрастания реакции атрибута качества кривая (*a*) «расплющивается». В данном случае точка, после которой ROI начинает снижаться, несмотря на повышение реакции атрибута качества, скорее всего, уже достигнута. Другими словами, вкладывать более серьезные средства не имеет смысла, так как они не приведут к значительному повышению полезности. Теперь взглянем на кривую (*b*), на которой едва заметное повышение реакции атрибута качества приводит к «взлету» полезности. Таким образом, для того чтобы серьезно повысить ROI данной архитектурной стратегии, достаточно немного увеличить реакцию атрибута качества.

О ВАЖНОСТИ МОДЕЛИРОВАНИЯ ЗАТРАТ

Случайный посетитель: Мне говорили, вы разбираетесь в вопросах готовности...

Лен Басс: Да, я кое-что знаю, но, вообще-то, я не эксперт.

СП: Может быть, вы все-таки сможете мне помочь. Я не могу понять, какая готовность нужна моей системе. Мой начальник говорит, что в случае чего за свежими идеями стоит обратиться к веб-сайту одной из крупных инвестиционных компаний.

ЛБ: Ну, у них, наверное, миллионы клиентов, и поэтому требования по готовности, скорее всего, очень жесткие.

СП: В том-то все и дело. У системы, которую я разрабатываю, будет всего несколько сотен пользователей, которым вполне достаточно готовности по пять 10-часовых дней в неделю. Так вот, как мне убедить начальника в том, что он слишком много хочет?

Изложив множество методов реализации различных атрибутов качества, мы до сих пор не говорили о том, как держать в узде ожидания ответственных лиц. Мы все время предполагали, что разработкой системы движут некие коммерческие факторы. Эти факторы порождают определенные требования, и задача архитектора состоит в том, чтобы обеспечить их максимальное удовлетворение. Но что делать, если наше допущение оказывается несостоятельным и в свете коммерческих задач системы требования оказываются избыточными?

Поразмыслив над этой ситуацией, я пришел к выводу, что у архитектора-таки есть весомый аргумент, позволяющий ему бороться против сверхжестких требований, — это стоимость. По той же причине я не езжу на дорогом роскошном автомобиле — слишком накладно.

Высокая готовность подразумевает мощное резервирование и возможность отката. Для реализации этих характеристик требуются временные затраты и специалисты. Работу специалистов нужно оплачивать; еще одна статья расходов приходится на закупку программного обеспечения, отличающегося высокой готовностью, и его адаптацию к конкретным потребностям.

Издержки в программной инженерии высчитываются при помощи стоимостных моделей. Делая допущения о характере конструируемой системы, параметрах окружения и квалификации специалистов и исходя из предыстории, стоимостная модель помогает составлять сметы.

По многим причинам стоимостные модели (в особенности на ранних стадиях жизненного цикла) несовершенны, однако более удачного средства сдерживания требований пока что не существует. В таком качестве они бесценны для архитектора.

— LJB

12.3. Реализация CBAM

В ходе практической реализации теоретических основ CBAM следует стремиться к минимизации необходимых действий. В частности, полезно ограничить пространство решений.

Этапы

На рис. 12.3 изображена диаграмма процессов, составляющих основу CBAM. Первые четыре этапа на ней сопровождаются комментариями с указанием относительного числа рассматриваемых сценариев. Постепенно их число уменьшается — таким образом, заинтересованные лица сосредоточиваются на тех сценариях, которые, по их мнению, в контексте ROI возымеют наибольшее значение.

Этап 1: критический анализ сценариев. Критический анализ сценариев проводится в рамках ATAM; на этом же этапе заинтересованные лица могут формулировать новые сценарии. Приоритеты расставляются в соответствии с потенциалом сценариев в контексте выполнения коммерческих задач системы; по результатам этапа для дальнейшего рассмотрения отбирается треть от общего первоначального числа сценариев.

Этап 2: уточнение сценариев. Уточнению подвергаются сценарии, отобранные по результатам первого этапа; основное внимание при этом уделяется их значениям стимула–реакции. Для каждого сценария устанавливаются наихудший, текущий, желаемый и наилучший уровни реакции атрибута качества.

Этап 3: расстановка сценариев согласно приоритетам. Каждому заинтересованному лицу выделяется 100 голосов, которые он берется распределить между

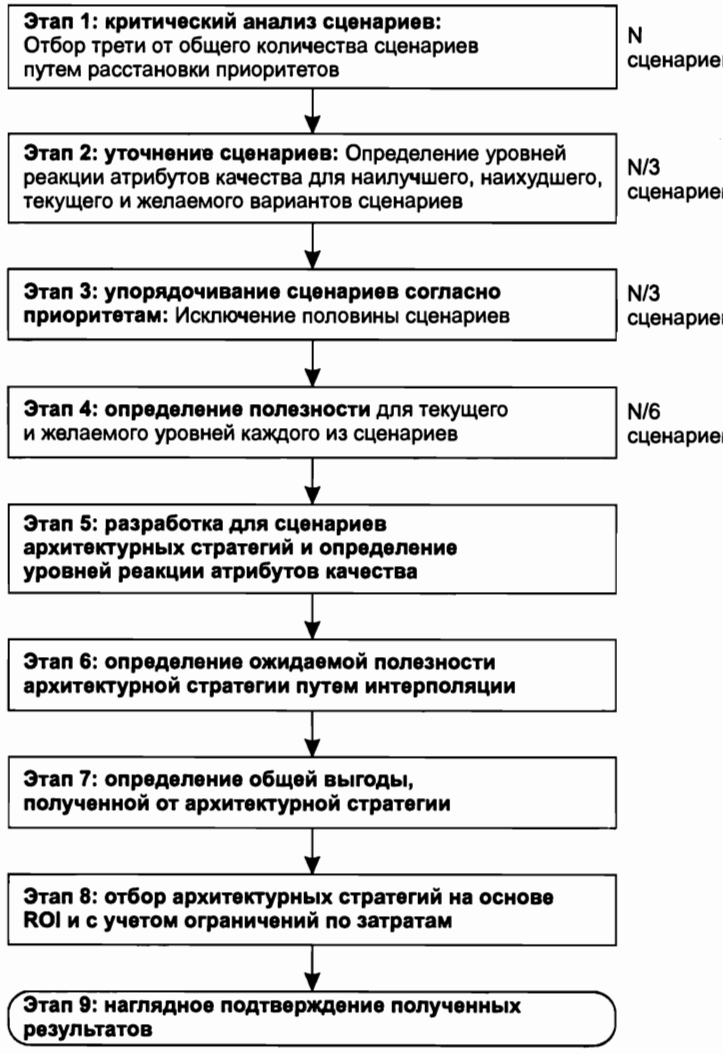


Рис. 12.3. Диаграмма процессов CBAM

сценариями, исходя из их желаемых значений реакции. После подсчета голосов для дальнейшего анализа остается только половина сценариев. Сценарию с наивысшим рангом присваивается вес 1.0, и, отталкиваясь от него, значения веса устанавливаются для всех остальных сценариев. Именно эти значения впоследствии задействуются при вычислении общей выгоды стратегии. Помимо прочего, на рассматриваемом этапе составляется список атрибутов качества, которые заинтересованные лица считают значимыми.

Этап 4: установление полезности. Для сценариев, оставшихся после проведения этапа 3, определяются значения полезности всех уровней реакции (наихудшего, текущего, желаемого, наилучшего) атрибута качества.

Этап 5: разработка для сценариев архитектурных стратегий и установление их желаемых уровней реакции атрибута качества. Разработка (или фиксация разработанных) архитектурных стратегий, ориентированных на реализацию выбранных сценариев, и определение «ожидаемых» уровней реакции атрибута качества. Учитывая то обстоятельство, что одна архитектурная стратегия иногда оказывает воздействие на несколько сценариев, расчеты необходимо провести для каждого из затронутых сценариев.

Этап 6: определение полезности «ожидаемых» реактивных уровней атрибута качества путем интерполяции. Исходя из установленных значений полезности (отраженных на кривой полезности) для рассматриваемой архитектурной стратегии определяется полезность желаемого уровня реакции атрибута качества. Эта операция проводится для каждого из перечисленных на этапе 3 значимых атрибутов качества.

Этап 7: расчет общей выгоды, полученной от архитектурной стратегии. Значение полезности «текущего» уровня вычитается из желаемого уровня и нормализуется исходя из поданных на третьем этапе голосов. Суммируются выгоды, полученные от конкретной архитектурной стратегии, по всем сценариям и для всех значимых атрибутов качества.

Этап 8: отбор архитектурных стратегий с учетом ROI, а также ограничений по стоимости и времени. Для каждой архитектурной стратегии определяются стоимостные и временные факторы. Значение ROI для стратегий определяется как отношение выгоды к издержкам. Архитектурные стратегии упорядочиваются по рангу согласно значениям ROI; впоследствии бюджет в первую очередь расходуется на высшие по рангу стратегии.

Этап 9: интуитивное подтверждение результатов. Проверяется соответствие выбранных архитектурных стратегий коммерческим задачам компании. Если наблюдаются противоречия, ищем недосмотры во время проведения анализа. В случае, если противоречия значительны, перечисленные этапы проводятся повторно.

12.4. Конкретный пример: проект ESC агентства NASA

Рассмотрим случай практического применения метода CBAM к реальной системе.

Система наблюдения за поверхностью Земли (Earth Observing System, EOS) — это группа спутников NASA, осуществляющих сбор информации для американской исследовательской программы глобальных изменений (US Global Change Research Program), а также ряда других научно-исследовательских организаций, базирующихся в различных странах мира. Центральная информационная система наблюдения за поверхностью Земли (Earth Observing System Data Information System Core System, ECS) собирает с ряда искусственных спутников с нисходящей связью данные, которые впоследствии подвергаются обработке. Задача ECS заключается в том, чтобы представить эти данные в форме более высокого уров-

ня, допускающий анализ и поиск. Необходимо одновременно предусмотреть универсальный способ хранения (и, следовательно, обработки) данных и общедоступный механизм введения новых форматов данных и алгоритмов обработки; в конечном итоге, информация должна стать доступной для всех желающих.

За один день ECS обрабатывает сотни гигабайт сырых данных об окружающей среде — все они поступают в систему в виде входного потока. По результатам вычисления 250 стандартных «продуктов» генерируется несколько тысяч гигабайт информации, архивируемой в восьми информационных центрах, расположенных на территории Соединенных Штатов. Важнейшие требования в системе относятся к производительности и готовности. Кроме того, долговременный характер подразумевает внимание к модифицируемости.

Руководитель проекта ECS, имея в своем распоряжении ограниченный годовой бюджет, должен был распределить его на нужды сопровождения текущей системы и ее модернизации. В ходе проведенных ранее аналитических действий (по методу ATAM) со слов заинтересованных лиц удалось зафиксировать множество желательных изменений и соответствующих им архитектурных стратегий. Поскольку из всего предложенного бюджета хватало лишь на 10–20 %, задача заключалась в том, чтобы выбрать для реализации относительно небольшой набор изменений. С помощью CBAM руководитель проекта установил коэффициенты прибыли на инвестированный капитал, и исходя из этого экономического критерия ему удалось принять рациональное решение.

Потенциал метода CBAM в описываемом случае мы направили на анализ одного из элементов ECS — рабочей группы по доступу к данным (Data Access Working Group, DAWG).

Этап 1: критический анализ сценариев

Для сверки сценариев, зафиксированных во время анализа по методу ATAM, мы еще раз собрали заинтересованных в системе ECS лиц и установили ряд новых сценариев. Поскольку у всех участников этой процедуры уже был опыт работы с ATAM, никаких сложностей у нас не возникло.

Сценарии, выбранные группой DAWG, перечислены в табл. 12.1. Имейте в виду, что они не слишком удачно сформулированы, а для некоторых даже не определены реакции. Вопросы эти решаются на этапе 2, когда количество сценариев существенно уменьшается[1].

Этап 2: уточнение сценариев

При уточнении сценариев мы уделили особое внимание точному определению количественных показателей стимула–реакции. Как показано в табл. 12.2, для каждого сценария мы установили и зафиксировали наихудший, текущий, желаемый и наилучший случаи.

[1] В рамках данного конкретного примера мы покажем только сокращенный набор сценариев.

Таблица 12.1. Сценарии после критического анализа, расположенные в порядке приоритета

Сценарий	Описание сценария
1	Сокращение отказов при распределении данных, приводящих к зависанию запросов на распределение и требующих ручного вмешательства
2	Сокращение отказов при распределении, приводящих к потере запросов на распределение
3	Сокращение количества заказов, отказавших в процессе подачи
4	Сокращение отказов при заказах, приводящих к зависанию и требующих ручного вмешательства
5	Сокращение отказов при заказах, приводящих к их потере
6	Отсутствие приемлемого метода отслеживания неудавшихся/отмененных заказов, поданных от имени ECSGuest, который предусматривал бы минимизацию ручного вмешательства (например, электронные таблицы)
7	Пользователю требуется дополнительная информация о причинах отказа при подаче заказа или получении данных
8	Из-за определенных ограничений приходится устанавливать искусственные пределы размера и количества заказов
9	В результате заказов малого объема пользователи получают слишком много оповещений
10	Система должна обеспечивать обработку пользовательских заказов за один день (если их объем не превышает 50 Гбайт) или за одну неделю (если их объем достигает 1 Тбайт)

Таблица 12.2. Задачи по реакции для уточненных сценариев

Сценарий	Задачи по реакции			
	Наихудший случай	**Текущий уровень**	**Желаемый уровень**	**Наилучший случай**
1	Зависание 10%	Зависание 5%	Зависание 1%	Зависание 0%
2	Потери > 5%	Потери < 1%	Потери 0%	Потери 0%
3	Отказы 10%	Отказы 5%	Отказы 1%	Отказы 0%
4	Зависание 10%	Зависание 5%	Зависание 1%	Зависание 0%
5	Потери 10%	Потери < 1%	Потери 0%	Потери 0%
6	Помощь требуется в 50% случаев	Помощь требуется в 25% случаев	Помощь требуется в 0% случаев	Помощь требуется в 0% случаев
7	Получение информации в 10% случаев	Получение информации в 50% случаев	Получение информации в 100% случаев	Получение информации в 100% случаев
8	Ограничения в 50 %случаев	Ограничения в 30% случаев	Ограничения в 0% случаев	Ограничения в 0% случаев
9	1 оповещение на 1 гранулу	1 оповещение на 1 гранулу	1 оповещение на 100 гранул	1 оповещение на 1000 гранул
10	Задача решается менее чем в 50% случаев	Задача решается в 60% случаев	Задача решается в 80% случаев	Задача решается более чем в 90% случаев

Этап 3: расстановка сценариев согласно приоритетам

В том, что касается голосования по вопросу об уточнении сценариев, группа специалистов по оценке сделала небольшое отступление от традиционной схемы метода. Вместо того чтобы организовать поименное голосование, они приняли решение обсудить каждый сценарий по отдельности и определиться с весом общими усилиями. На весь набор сценариев выделили 100 голосов (табл. 12.3). Изначально от заинтересованных лиц не требовалось подачи голосов, кратных пяти, однако они пришли к мнению, что более серьезная точность, во-первых, не нужна, а во-вторых, ничем не оправдана.

Таблица 12.3. Уточненные сценарии и количество поданных за них голосов

Сценарий	Голоса	Задачи по реакции			
		Наихудший случай	**Текущий уровень**	**Желаемый уровень**	**Наилучший случай**
1	10	Зависание 10%	Зависание 5%	Зависание 1%	Зависание 0%
2	15	Потери > 5%	Потери < 1%	Потери 0%	Потери 0%
3	15	Отказы 10%	Отказы 5%	Отказы 1%	Отказы 0%
4	10	Зависание 10%	Зависание 5%	Зависание 1%	Зависание 0%
5	15	Потери 10%	Потери < 1%	Потери 0%	Потери 0%
6	10	Помощь требуется в 50% случаев	Помощь требуется в 25% случаев	Помощь требуется в 0% случаев	Помощь требуется в 0% случаев
7	5	Получение информации в 10% случаев	Получение информации в 50% случаев	Получение информации в 100% случаев	Получение информации в 100% случаев
8	5	Ограничения в 50% случаев	Ограничения в 30% случаев	Ограничения в 0% случаев	Ограничения в 0% случаев
9	10	1 оповещение на 1 гранулу	1 оповещение на 1 гранулу	1 оповещение на 100 гранул	1 оповещение на 1000 гранул
10	5	Задача решается менее чем в 50% случаев	Задача решается в 60% случаев	Задача решается в 80% случаев	Задача решается более чем в 90% случаев

Этап 4: установление полезности

Полезность каждого сценария на этом этапе, опять же, определялась заинтересованными лицами согласованно. Нулевая сумма баллов полезности должна была обозначать отсутствие таковой; сумма балов, равная 100, напротив, указывала на максимально возможную полезность. Результаты обсуждения представлены в табл. 12.4.

Таблица 12.4. Голоса, поданные за сценарии, и суммы полученных ими баллов

Сценарий	Голоса	Задачи по реакции			
		Наихудший случай	Текущий уровень	Желаемый уровень	Наилучший случай
1	10	10	80	95	100
2	15	0	70	100	100
3	15	25	70	100	100
4	10	10	80	95	100
5	15	0	70	100	100
6	10	0	80	100	100
7	5	10	70	100	100
8	5	0	20	100	100
9	10	50	50	80	90
10	5	0	70	90	100

Этап 5: разработка для сценариев архитектурных стратегий и установление их желаемых уровней реакции атрибута качества

Исходя из требований, подразумеваемых вышеперечисленными сценариями, архитекторы ECS разработали 10 архитектурных стратегий. Как вы помните, любая отдельно взятая архитектурная стратегия может оказывать воздействие сразу на несколько сценариев. Учитывая такую сложность отношений, ожидаемый уровень реакции атрибута качества для каждой из стратегий пришлось устанавливать относительно всех значимых сценариев.

Набор архитектурных стратегий, а также определения сценариев, с которыми они связаны, представлены в табл. 12.5. Для каждой пары «архитектурная стратегия/сценарий» показаны ожидаемые уровни реакции относительно конкретного сценария (для сравнения также приводятся текущие уровни реакции).

Таблица 12.5. Архитектурные стратегии и сценарии, с которыми они связаны

Стратегия	Имя	Описание	Связанные сценарии	Текущий уровень реакции	Ожидаемый уровень реакции
1	Персистентность заказов при подаче	Сохранение заказа происходит сразу после его поступления в систему	3	Отказы 5%	Отказы 2%
			5	Потери < 1%	Потери 0%
			6	Помощь требуется в 25% случаев	Помощь требуется в 0% случаев
2	Разбиение заказов	Операторам разрешается разделять крупные заказы на несколько мелких	8	Ограничения в 30% случаев	Ограничения в 15% случаев
3	Группировка заказов	Ряд мелких заказов объединяется в один крупный	9	1 оповещение на 1 гранулу	1 оповещение на 100 гранул

Стратегия	Имя	Описание	Связанные сценарии	Текущий уровень реакции	Ожидаемый уровень реакции
			10	Задача решается в 60% случаев	Задача решается в 55%случаев
4	Сегментация заказов	Оператору разрешается пропускать элементы, извлечение которых из-за низкого качества данных или проблем с готовностью невозможно	4	Зависание 5%	Зависание 2%
5	Переназначение в заказах	Оператору разрешается переназначать для элементов заказа типы носителей	1	Зависание 5%	Зависание 2%
6	Повторное выполнение заказов	Заказы, выполнить которые с первого раза из-за временных сбоев в системе или проблем с данными не удалось, оператор может пытаться выполнить повторно	4	Зависание 5%	Зависание 3%
7	Силовое завершение заказов	Оператору разрешается подменять неготовность элемента, обусловленную ограничениями на качество данных	1	Зависание 5%	Зависание 3%
8	Оповещение о неудавшихся заказах	Пользователи должны получать оповещения лишь о (частично) неудавшихся заказах, а сопровождаться они должны подробной информацией о состоянии каждого элемента; отправку пользователям оповещений должен утверждать оператор; он же может их редактировать	6	Помощь требуется в 25% случаев	Помощь требуется в 20% случаев
			7	Получение информации в 50% случаев	Получение информации в 90% случаев
9	Отслеживание гранул согласно уровню	Оператор и пользователи могут в порядке очередности определять состояние элементов заказа	6	Помощь требуется в 25% случаев	Помощь требуется в 10% случаев
			7	Получение информации в 50% случаев	Получение информации в 95% случаев
10	Ссылки на информацию о пользователе	Оператор может быстро находить контактную информацию пользователя. Путем обращения сервера к информации SDSRV определяются возможные ограничения на данные.	7	Получение информации в 50% случаев	Получение информации в 60% случаев

продолжение

Таблица 12.5 (*продолжение*)

Стратегия	Имя	Описание	Связанные сценарии	Текущий уровень реакции	Ожидаемый уровень реакции
		Кроме того, сервер осуществляет маршрутизацию заказов/сегментов заказов соответствующим средствам распределения: DDIST, PDS, внешним механизмам подключения, инструментам обработки данных и т. д.			

Этап 6: определение полезности «ожидаемых» уровней реакции атрибута качества путем интерполяции

За определением относительного набора сценариев ожидаемого уровня реакции каждой архитектурной стратегии следует расчет их полезности. Для этого следует обратиться к суммам баллов полезности текущей и желаемой реакции для всех задействованных атрибутов. Исходя из этих значений путем интерполяции можно вычислить полезность ожидаемых уровней реакции атрибута качества для пар «архитектурная стратегия/сценарий», реализуемых с подсистемой DAWG системы ECS.

Таблица 12.6. Архитектурные стратегии и их ожидаемая полезность

Стратегия	Имя	Связанные сценарии	Текущая полезность	Ожидаемая полезность
1	Персистентность заказов при подаче	3	70	90
		5	70	100
		6	80	100
2	Разбиение заказов	8	20	60
3	Группировка заказов	9	50	80
		10	70	65
4	Сегментация заказов	4	80	90
5	Переназначение в заказах	1	80	92
6	Повторное выполнение заказов	4	80	85
7	Силовое завершение заказов	1	80	87
8	Оповещение о неудавшихся заказах	6	80	85
		7	70	90
9	Отслеживание гранул согласно уровню	6	80	90
		7	70	95
10	Ссылки на информацию о пользователе	7	70	75

Результаты расчета для пар «архитектурная стратегия/сценарий», представленных в табл. 12.5, приведены в табл. 12.6.

Этап 7: расчет общей выгоды, полученной от архитектурной стратегии

На основе представленной в табл. 12.6 информации можно рассчитать общую выгоду, полученную от применения каждой архитектурной стратегии; этой цели служит уравнение, которое мы привели в подразделе «Определение выгод и нормализация». В нем общая выгода рассчитывается как сумма значений выгоды от каждого сценария, а затем нормализуется относительным весом данного сценария. Баллы общей выгоды для каждой из рассмотренных архитектурных стратегий приводятся в табл. 12.7.

Таблица 12.7. Общая выгода, полученная от архитектурных стратегий

Стратегия	Связанный сценарий	Вес сценария	Ненормализованная выгода от архитектурной стратегии	Нормализованная выгода от архитектурной стратегии	Общая выгода от архитектурной стратегии
1	3	15	20	300	950
1	5	15	30	450	
1	6	10	20	200	
2	8	5	40	200	200
3	9	10	30	300	275
3	10	5	–5	–25	
4	4	10	10	100	100
5	1	10	12	120	120
6	4	10	5	50	50
7	1	10	7	70	70
8	6	10	5	50	150
8	7	5	20	100	
9	6	10	10	100	225
9	7	5	25	125	
10	7	5	5	25	25

Этап 8: отбор архитектурных стратегий с учетом ROI, а также ограничений по стоимости и времени

На завершающей стадии анализа участники группы оценки определили стоимость реализации всех архитектурных стратегий. Расчеты проводились исходя из опыта работы с системой, и в конечном итоге для каждой архитектурной стратегии удалось установить коэффициент прибыли на инвестированный капитал (ROI). Соответственно, у нас появилась возможность ранжировать стратегии (табл. 12.8). Неудивительно, что ранги примерно соответствуют порядку предложения стратегий: у первой стратегии высший ранг, а у третьей — второй после

высшего. Нижайший ранг у десятой стратегии, а второй с конца — у восьмой. Таким образом, проведенные расчеты подтверждают интуитивное понимание заинтересованными лицами выгод от различных архитектурных стратегий. И действительно, в случае с ECS наибольшие выгоды обещают принести первые две предложенные стратегии.

Таблица 12.8. Коэффициенты ROI различных архитектурных стратегий

Стратегия	Издержки	Общая выгода от применения стратегии	Коэффициент ROI стратегии	Ранг стратегии
1	1200	950	0,79	1
2	400	200	0,5	3
3	400	275	0,69	2
4	200	100	0,5	3
5	400	120	0,3	7
6	200	50	0,25	8
7	200	70	0,35	6
8	300	150	0,5	3
9	1000	225	0,22	10
10	100	25	0,25	8

12.5. Результаты оценки по методу CBAM

Наиболее очевидные результаты проведения оценки по методу CBAM — классификация архитектурных стратегий по критерию прогнозируемого коэффициента ROI — показаны в табл. 12.8. Впрочем, равно как и в отношении метода ATAM, преимущества CBAM не ограничиваются количественными показателями. У них есть социальный и культурный аспекты.

Дискуссии, сопровождающие процессы сбора информации и принятия решений, в рамках CBAM играют не менее значимую роль, чем ранжирование архитектурных стратегий. Процесс CBAM структурирует проблемы, которые обычно не выходят за рамки свободных дискуссий; он, таким образом, отделяет требования от архитектурных стратегий и способствует четкому формулированию стимулов и задач по реакции. Процесс CBAM заставляет заинтересованных лиц заранее и четко излагать свои сценарии, устанавливать для различных задач по реакции уровни полезности и, исходя из конечных результатов в определении полезности, расставлять сценарии согласно приоритету. Наконец, что само по себе неплохо, рассмотренный процесс конкретизирует сценарии и требования.

12.6. Заключение

CBAM — это одновременно итерационный процесс извлечения информации и каркас анализа решений. Он предназначен для обработки сценариев, выражающих различные атрибуты качества. Пространство решений заинтересованные лица

исследуют при помощи кривых «реакция—полезность», которые демонстрируют изменение полезности системы в зависимости от изменения ее атрибутов. Применяемый в рамках метода согласительный принцип стимулирует оживленные обсуждения среди заинтересованных лиц и решение между ними спорных вопросов. Прослеживаемость проектных решений предусматривает возможность модернизации и постоянного усовершенствования процесса проектирования.

Извлекать информацию из реальных проектов довольно сложно. В наши обязанности как исследователей входит разработка методов, доступных для реальных инженеров, работающих над реальными проектами. Методы эти призваны выдавать полезные результаты быстро и с разумными «издержками» в плане временных ресурсов заинтересованных лиц. Наш опыт оценки по методу CBAM свидетельствует о значительных расхождениях между теоретическим и практическим механизмами решения задач. Несколько раз применив этот метод в отношении системы ECS NASA, мы внесли в него весьма серьезные коррективы.

Несмотря на некоторые сложности практической реализации, мы не сомневаемся в превосходстве экономических методик над нерегулярными процедурами принятия решений, которые в контексте современных приложений (даже самых сложных) все еще занимают главенствующее положение. Предоставляя специалистам средства схематизации и структурирования дискуссий и процесса принятия решений, CBAM в значительной степени упорядочивает разработку сложных программных систем.

12.7. Дополнительная литература

Одними из первых исследований по методу CBAM явились [Kazman 01] и [Asundi 01]. Моделированию затрат посвящены работы [Boehm 81] и [Jones 99]. Об оценке архитектуры системы ECS по методике ATAM речь идет в издании [Clements 02a].

12.8. Дискуссионные вопросы

1. Одно из нововведений CBAM заключается в употреблении кривых «реакция—полезность». Ознакомьтесь со стилями кривых, изображенных на рис. 12.2. Каковы, по вашему мнению, были обстоятельства, в которых мы при участии заинтересованных лиц выяснили информацию для построения каждой из этих кривых? Какие ситуации они выражают?
2. Попытки определения издержек и выгод обычно сопряжены с неопределенностью. Каковы типичные источники неопределенности и как их следует характеризовать, измерять и минимизировать?

Глава 13
Всемирная паутина. Конкретный пример реализации способности к взаимодействию

(в соавторстве с Хон-Мей Чен[1])

> Нашей основной целью было достижение гибкости. Любые спецификации, призванные обеспечить способность к взаимодействию, на самом деле накладывали на реализацию Всемирной паутины ограничения. Таким образом, число спецификаций требовалось свести к минимуму... а самые необходимые следовало составить так, чтобы они не зависели друг от друга... Такой подход предоставляет возможность замены отдельных элементов проектного решения без корректировки архитектуры как таковой.
>
> *Тим Бернерс-Ли [Berners-Lee 96b]*

> Полагаю, что в будущем — не слишком отдаленном — у любого, не обзаведшегося собственной веб-страницей, будет полное право на этом основании требовать от правительства выделения субсидии.
>
> *Скотт Адамс, создатель Dilbert*

Наверное, самый выразительный пример практического действия архитектурно-экономического цикла (Architecture Business Cycle, ABC) — это изменения, которым подверглись задачи, бизнес-модель и архитектура Всемирной паутины с момента ее появления в 1990 году. Никто — ни заказчики, ни пользователи, ни сам архитектор (Тим Бернерс-Ли) — в тот момент не могли даже предположить, что

---

[1] Хон-Мей Чен (Hong-Mei Chen) — адъюнкт-профессор факультета менеджмента в области информационных технологий Университета шт. Гавайи.

Всемирной паутине уготовано пройти этап бурного развития и умопомрачительного роста. В настоящей главе мы намерены рассмотреть Всемирную паутину с точки зрения ABC и обратить ваше внимание на то, как изменяющиеся задачи и коммерческие потребности ее игроков отражаются на уровне архитектуры. Сначала мы обратимся к тем требованиям, которые предъявлялись к Сети первоначально, к ее первым игрокам, а затем проанализируем изменения, происшедшие в серверной архитектуре под влиянием ABC.

13.1. Отношение к архитектурно-экономическому циклу

Предложение об организации Всемирной паутины изначально сформулировал Тим Бернерс-Ли (Tim Berners-Lee) — научный сотрудник Европейской лаборатории ядерных исследований (European Laboratory for Particle Physics, CERN). Понаблюдав за другими сотрудниками CERN, он пришел к выводу, что они формируют постоянно развивающуюся социальную «паутину». Люди приходили и уходили, организовывали новые научные сообщества, распускали старые, пользовались одними и теми же документами, болтали в курилках и занимались кучей других дел. Идея Бернерса-Ли состояла в том, чтобы подкрепить эту неформальную сеть аналогичной паутиной, связывающей представленную в электронном виде информацию. В 1989 году он составил и распространил в пределах CERN документ, озаглавленный «Управление информацией: предложение» («Information Management: A Proposal»). К октябрю 1990 года пересмотренная версия предложения была утверждена руководством, и, после того как для нового проекта выбрали имя «Всемирная паутина» (World Wide Web), началась его разработка.

Элементы ABC, которые участвовали в первоначальном предложении, получившем поддержку руководства CERN, изображены на рис. 13.1. Исходная система была призвана активизировать взаимодействие между научными сотрудниками CERN (конечными пользователями) в рамках неоднородной вычислительной среды. В роли заказчика выступило руководство CERN, а разработчиком оказался один-единственный исследователь. Согласно предложенной Бернерсом-Ли экономической модели, проектируемая система должна была стимулировать обмен информацией между штатными сотрудниками CERN. Ограниченность самого предложения и поставленных задач (носивших, за невозможностью проверки реальности их достижения, умозрительный характер) компенсировалась незначительностью вложений со стороны CERN — для создания и тестирования системы требовалось всего-то несколько месяцев труда одного исследователя.

Тем исследователям, которые уже с начала 1970-х годов, когда появилась сеть Интернет, эксплуатировали ее возможности, техническая база новой системы не казалась чем-то из ряда вон выходящим. Центральное управление в существовавшей сети осуществлялось крайне фрагментарно (обязанности добровольческих комитетов сводились к определению протоколов, обеспечивавших передачу информации между различными узлами Интернета, и выделению полномочий для создания новых новостных групп); кроме того, для нее был характерен

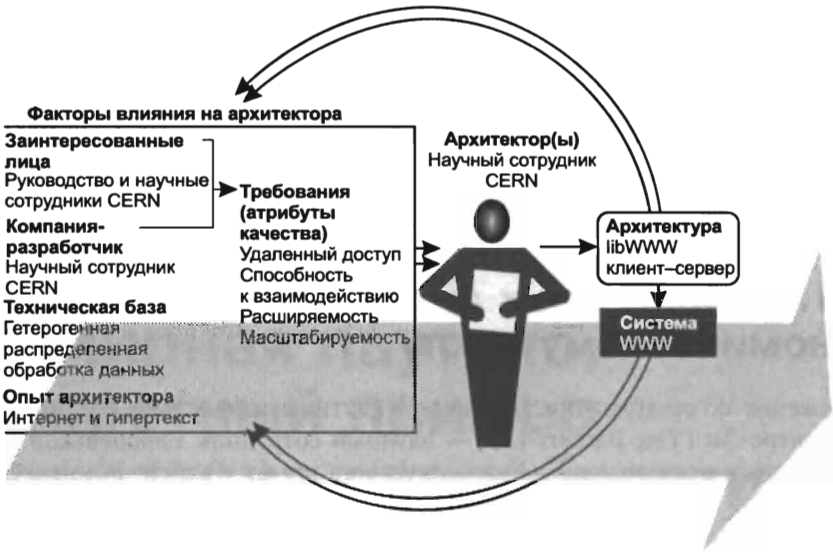


Рис. 13.1. Исходный архитектурно-экономический цикл Всемирной паутины

стихийный, «дикарский» стиль общения, основными каналами которого служили специализированные новостные группы.

Первые гипертекстовые системы появились еще раньше — все началось с концепции Ванневара Буша (Vannevar Bush), изложенной им в 1940-х годах. Исследования в рамках этой концепции проводились в 1960-е, 1970-е и даже в 1980-е годы; в частности, регулярно собирались конференции, посвященные гипертексту. Впрочем, крупномасштабной реализации концепции Буша к 1980-м годам так и не случилось — гипертекст в основном использовался в небольших системах управления документацией. Как оказалось, так будет не всегда.

В 1990 году руководство CERN одобрило предложение Бернерса-Ли. К ноябрю он разработал на платформе NeXT первую веб-программу; из этого со всей очевидностью следует, что к работе над реализацией он приступил, не дожидаясь ответа со стороны руководства. Рассогласованность позиции руководящих инстанций и действий исследователей вообще характерна для научно-исследовательских организаций, в которых начальные капиталовложения в проекты держатся на низком уровне. Неструктурированная разработка характерна для них в большей степени, чем для коммерческих компаний, что объясняется их ориентацией на свежие решения и творческие устремления своих сотрудников, а следовательно, большей, по сравнению со среднестатистической компанией-разработчиком, свободой действий.

Некоторые характеристики первоначальной реализации веб-системы до сих пор не реализованы в браузерах. К примеру, веб-система предусматривала для пользователей возможность создания ссылок, не выходя из программы просмотра; кроме того, комментировать информацию могли как авторы, так и читатели. Поначалу Бернерс-Ли был уверен в том, что пользователи не захотят возиться с языком разметки гипертекста (HyperText Markup Language, HTML) и унифи-

цированными указателями ресурсов (URLs). Как оказалось, он ошибался. Ради возможности создания веб-публикаций пользователи были готовы мириться с этими неудобствами.

13.2. Требования и атрибуты качества

Согласно представлениям сотрудников CERN и первоначальной реализации, Всемирная паутина (World Wide Web, WWW) характеризовалась рядом желаемых атрибутов качества. Она должна была воплотить переносимость, способность к взаимодействию с компьютерами разных типов (при условии, что на них установлено одно и то же программное обеспечение), масштабируемость и расширяемость. Коммерческие задачи по стимулированию взаимодействия между пользователями и созданию гетерогенной вычислительной среды диктовали реализацию таких атрибутов качества, как удаленный доступ, способность к взаимодействию, расширяемость и масштабируемость; все это в конечном итоге вылилось в создание libWWW — первой программной библиотеки, поддерживающей веб-разработку и распределенную архитектуру типа «клиент–сервер». Благодаря реализации в первоначальной программной архитектуре перечисленных свойств появилась инфраструктура, на основе которой Всемирная паутина впоследствии получила стремительнейшее развитие (табл. 13.1). Поскольку в libWWW предусматривается строгое разделение задач, библиотека может работать практически на любом аппаратном обеспечении, без труда приспосабливается к новым протоколам, форматам данных и приложениям. Отсутствие централизованного управления позволяет Паутине расти бесконечно.

Таблица 13.1. Статистика роста Всемирной паутины[1]

Дата	Количество веб-сайтов	Процентное отношение сайтов сайтов в домене .com	Количество хостов на веб-сервер
6/93	130	1,5	13 000
12/93	623	4,6	3475
6/94	2738	13,5	1095
12/94	10 022	18,3	451
6/95	23 500	31,3	270
1/96	100 000	50,0	94
6/96	252 000	68,0	41
1/97	646 162	62,6	40
1/98	1 834 710		16,2
1/99	4 062 280		10,6
1/00	9 950 491		7,3
1/01	27 585 719	54,68	4,0

[1] *Источник*: Приводится с разрешения Мэтью Грэя (Matthew Gray) из Технологического института Массачусетса.

Ниже мы намерены более подробно разобрать эти, а также некоторые другие базовые требования; к структуре библиотеки libWWW мы вернемся немного позже, в разделе 13.3. Стоит заметить, что требование об удобстве использования изначально сформулировано не было, а умопомрачительный взлет популярности Паутины начался лишь после возникновения «мышиных» браузеров. С другой стороны, требование к переносимости и поддержке гетерогенной вычислительной среды привело к появлению самого понятия «браузер» как отдельного элемента и тем самым подготовило почву для разработки более сложных программ просмотра.

Первоначальные требования

В первоначальных предложениях по проекту «Всемирная паутина» были озвучены следующие требования.

- *Удаленный доступ в пределах отдельных сетей.* Любая информация, хранящаяся на любой входящей в сеть CERN машине, должна быть доступна с любой другой машины в этой сети.
- *Гетерогенность.* Ограничения по работе системы на той или иной аппаратной или программной платформе недопустимы.
- *Децентрализация.* Согласно законам социальной паутины и Интернета, существование монопольного источника данных и услуг также не допускается. Данное требование ставилось в расчете на дальнейшее развитие WWW. В частности, следовало децентрализовать операцию установки ссылок на документы.
- *Доступ к существующим данным.* Существовавшие базы данных требовалось сделать доступными.
- *Возможность введения новых данных пользователями.* «Публиковать» в Сети свои собственные данные пользователи должны были через тот же интерфейс, с помощью которого они просматривали уже опубликованную информацию.
- *Личные ссылки.* Требовалась возможность частного комментирования ссылок и узлов.
- *Оформление.* Первоначально в качестве единственной формы отображения данных планировался вывод на терминал 24 × 80 символов ASCII. Графическое представление считалось необязательным.
- *Анализ данных.* Пользователям следовало предоставить возможность поиска по различным базам данных, выявления аномалий, закономерностей, девиаций и пр. В качестве примера Бернерс-Ли упоминал поиск недокументированного ПО и организаций без штата.
- *Активные ссылки.* Учитывая то, что информация постоянно изменяется, при ее просмотре пользователю необходимо было предоставить возможность обновления. Этого можно было добиться посредством извлечения информации при каждом обращении по соответствующей ссылке или (что

сложнее) путем оповещения пользователей об изменении информации, доступной по определенному адресу.

Помимо перечисленных требований было установлено несколько антитребований (nonrequirements). В частности, защита авторских прав и данных явно определялась как требования, которые в рамках данного проекта (в его исходной версии) реализованы *не* будут. Дело в том, что Всемирная паутина изначально планировалась как общедоступная среда. Кроме того, согласно оригинальному предложению, пользователи не обязаны были отдавать предпочтение какому-то одному формату разметки.

Ниже перечислены некоторые критерии и характеристики, которые в рассматриваемое время часто встречались в предложениях по системам гипертекста, но, несмотря на это, не были отражены в предложении по WWW:

- контроль над топологией;
- определение методик навигации и требований по пользовательскому интерфейсу, в том числе по ведению журнала визуальной истории;
- наличие нескольких типов ссылок, отражающих разные отношения между узлами.

Некоторые требования из числа продекларированных первоначально вошли в состав современной реализации Всемирной паутины; иные на сегодняшний день не реализованы или реализованы частично. Важность отдельных требований сильно недооценена — к примеру, возможности анализа данных, а также создания активных и личных ссылок до сих пор не получили должного внимания и по большей части не отражены на практике.

Для беспрецедентных систем в целом характерны адаптация и выборочная отсрочка. Мало того что под видом требований часто подаются списки желаемых характеристик, разобраться с компромиссными решениями, которые приходится принимать для реализации этих требований, в беспрецедентных системах зачастую удается лишь после создания проекта. Одни требования в процессе принятия компромиссных решений приобретают относительно большую важность, другие — относительно меньшую.

Про одно из вышеперечисленных требований можно со всей определенностью сказать, что его влияние сильно *недооценили*. Речь идет об «оформительской» функции графики, которая на сегодняшний день формирует львиную долю веб-трафика. Графическая информация возбуждает недюжинный интерес, и этим определяется ее доля в интернет-трафике. И тем не менее ни Бернерс-Ли, когда составлял первоначальное предложение, ни руководство CERN, когда его утверждало, не проявили должного внимания к графике, и первый браузер стал линейным. Аналогичным образом, в первоначальном предложении не было ни намека на намерения обеспечить поддержку звука и видеоизображения.

По мере прохождения Всемирной паутиной архитектурно-экономического цикла отдельные антитребования превратились в требования. В частности, существенный вес приобрела проблема безопасности; в особо крупных масштабах она обозначила себя после того, как на лидирующие позиции в составе веб-трафика стала выходить коммерческая информация. С учетом распределенности

и децентрализации сети Интернет проблема безопасности становится сложной и многокомпонентной. Трудно говорить о безопасности, если нельзя гарантировать защищенный доступ к приватным данным, — Всемирная паутина открывает возможность проникновения в систему клиента, которой при случае не преминут воспользоваться непрошеные гости.

За последние несколько лет вышеупомянутая проблема стала еще более актуальной — структура и направление развития WWW теперь определяются предприятиями электронной коммерции, безопасное функционирование которых обеспечивается многочисленными специализированными механизмами. Наиболее очевидным решением представляется простое шифрование уязвимых данных. В большинстве случаев оно осуществляется средствами протокола защищенных сокетов (secure sockets layer, SSL), представленного в веб-браузерах под именем протокола защищенной передачи гипертекста (hypertext transfer protocol secure, HTTPS). Впрочем, этот протокол лишь снижает вероятность выслеживания приватных данных во время их передачи по общедоступной сети. Другая категория решений, представителем которой является технология Microsoft Passport, ориентирована на проверку достоверности сведений, которые пользователь предоставляет сам о себе. (Различные аспекты безопасности рассматривались в главе 4, а в главе 5 изложен ряд тактик их реализации.)

Требования приходят и уходят

Предсказать невиданный рост Всемирной паутины (равно как и Интернета в целом), происшедший за последние несколько лет, было невозможно. Согласно статистическим данным 2001 года, каждые полгода Всемирная паутина удваивается в размере. Так, она прошла путь от 130 сайтов в середине 1993 года до 230 000 в середине 1996-го и, наконец, включала в себя 27 миллионов сайтов в начале 2001-го (см. табл. 13.1). На рис. 13.2 изображена схема интернет-магистралей, охватывающих всю территорию Соединенных Штатов. Соответственно возросла численность интернет-хостов (их статистика ведется согласно зарегистрированным IP-адресам) — от 1,3 миллиона в 1993 году до 9,5 миллиона в начале 1996-го.

Процессы роста охватывают как Всемирную паутину, так и Интернет в целом, однако по темпам первая явно опережает. Эта тенденция хорошо прослеживается по последнему столбцу табл. 13.1, демонстрирующему стабильное снижение отношения интернет-хостов к веб-серверам. Иначе говоря, все большая доля интернет-хостов становится веб-серверами.

Меняется не только масштаб Всемирной паутины, но и ее характер, о чем свидетельствует содержание третьего столбца табл. 13.1. Несмотря на истоки в научно-исследовательском сообществе, все большее место в ней занимает коммерческий трафик (подтверждение тому — количество интернет-хостов с именами, заканчивающимися на .com). В процентном отношении сайтов в домене .com около 55 %; стабилизация этой цифры никак не связана с каким бы то ни было спадом коммерческой деятельности — скорее ее следует относить к повышению популярности других доменов наподобие .net и .biz.

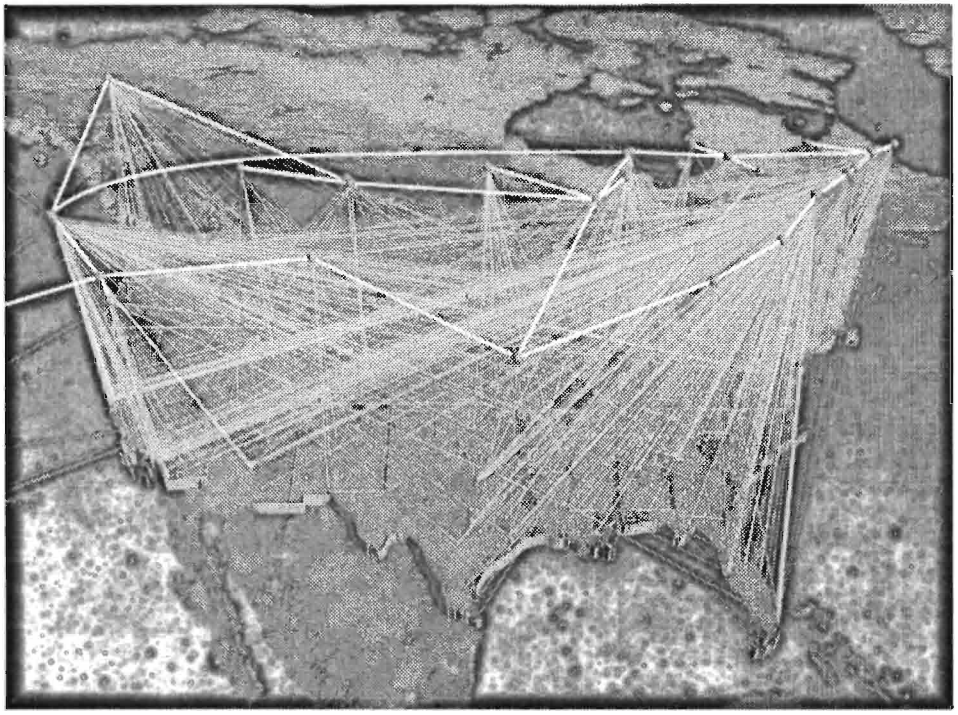


Рис. 13.2. Интернет-каналы в Соединенных Штатах[1]

Весьма занимателен побочный эффект упрощения и распространения доступа к Всемирной паутине. Удобный, практически неконтролируемый доступ к графике подтолкнул развитие «киберпорнографии», что, в свою очередь, определило появление нового требования о возможности маркирования контента и контролируемости доступа к нему. В результате появилась спецификация платформы отбора информации в Интернете (platform for Internet content selection, PICS) — общеотраслевого набора принципов и их реализаций, предусматривающего маркирование контента и гибкие критерии отбора. Таким образом, потребители информации получили возможность выбирать материал, который они хотят (а их клиенты могут) просматривать, руководствуясь лишь собственными вкусами и убеждениями; при этом ограничения свободы действий контент-провайдеров удалось избежать. Действенность PICS проявляется, когда, к примеру, родитель желает ограничить просмотр детьми видеоматериала фильмами с определенным рейтингом, а работодатель — запретить своим подчиненным посещать в рабочее время не относящиеся к делу сайты.

Чтобы уяснить степень различия современной Всемирной паутины от ее первоначального замысла, представим, что в своем предложении Бернерс-Ли

[1] © 1996, Donna Cox, Robert Patternson; производство Национального центра по применению суперкомпьютеров (National Center for Supercomputing Applications), Университет шт. Иллинойс, Урбана-Шампейн. Перепечатывается с разрешения.

сформулировал требование об ограничении информации, мотивируя его необходимостью уберечь детей от доступа к порнографическим материалам. Как бы на это отреагировало руководство CERN? Без разговоров отклонило бы. То, как меняются задачи заинтересованных лиц, мы обсудим в разделе 13.5 применительно к архитектурно-экономическому циклу Всемирной паутины.

13.3. Архитектурное решение

Архитектурной основой Всемирной паутины, принятой сначала CERN, а затем и консорциумом W3C, стало сочетание модели «клиент–сервер» и библиотеки (libWWW), скрывающей все зависимости по аппаратному обеспечению, операционным системам и протоколам. На рис. 13.3 приводится схема взаимодействия производителей и потребителей контента посредством соответствующих серверов и клиентов. Производитель размещает на машине, исполняющей роль сервера, описание информации на языке HTML. Обмен информацией с клиентом осуществляется сервером по протоколу передачи гипертекста (hypertext transfer protocol, HTTP). Программное обеспечение сервера и клиента основывается на libWWW, что позволяет маскировать детали протокола и зависимости от платформ. Одним из элементов, расположенных на стороне клиента, является браузер, который отвечает за понятное для потребителя информации отображение HTML.

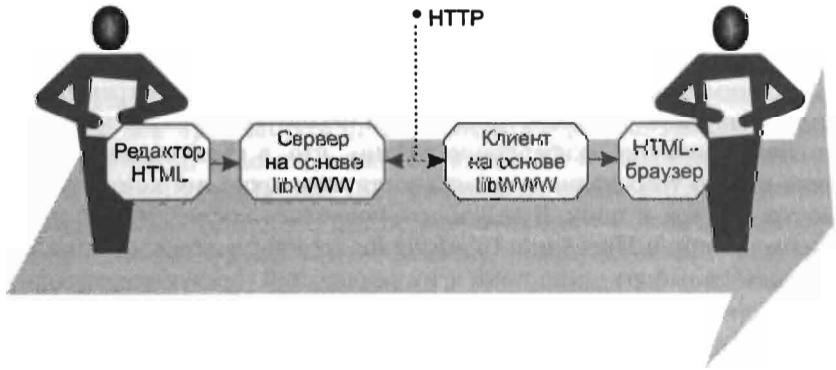


Рис. 13.3. Взаимодействие производителей и потребителей информации осуществляется посредством клиентов и серверов

Ниже мы намерены более подробно поговорить о библиотеке libWWW и архитектуре «клиент–сервер», которые, однажды составив основу первоначальной версии WWW, сохраняют это качество по сей день. В разделе 13.4 мы проанализируем изменения в архитектуре Всемирной паутины и в соответствующем программном обеспечении, происшедшие в ответ на революцию электронной коммерции.

Реализация первоначальных требований: libWWW

Как мы уже говорили, libWWW представляет собой библиотеку программного обеспечения для создания приложений, предназначенных для работы на сторо-

нах клиента и сервера. Реализованная в ней функциональность — подключение к удаленным хостам, интерпретация потоков данных HTML и пр. — универсальна для большинства таких приложений.

libWWW — это компактная, переносимая библиотека, которая встраивается в различные веб-приложения: клиенты, серверы, базы данных и веб-пауки. Она состоит из пяти уровней, изображенных на рис. 13.4.

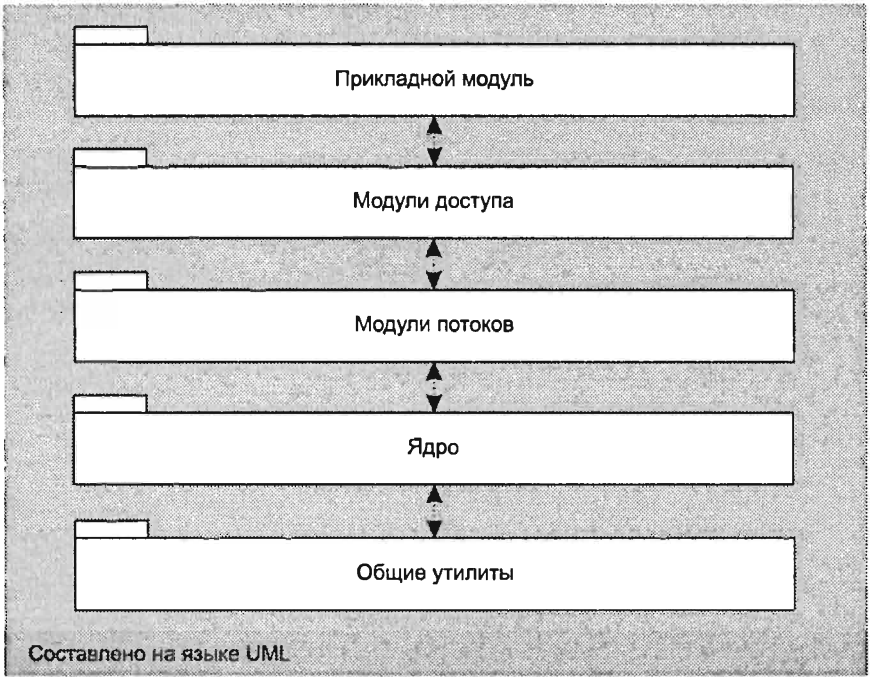


Рис. 13.4. Многоуровневое представление библиотеки libWWW

Из общих утилит формируется уровень переносимости, служащий основой для всех остальных элементов системы. На этом уровне находятся базовые стандартные блоки — в том числе средства сетевого управления и управления строками, а также типы данных (в частности, классы-контейнеры). Службы этого уровня обеспечивают независимость всех вышележащих уровней от конкретной платформы; таким образом, задача по перенесению на новую аппаратную или программную платформу практически полностью сводится к однократному (для каждой платформы) перенесению данного обслуживающего уровня.

На уровне ядра содержится «скелет» функциональности веб-приложения: средства доступа к сети, управления данными, синтаксического анализа, регистрации и т. д. Сам по себе этот уровень не выполняет никаких действий. Он лишь предоставляет стандартный интерфейс веб-приложения, при том что фактическая функциональность последнего обеспечивается сменными модулями и вызываемыми функциями, которые это приложение регистрирует. *Сменные модули* (plug-ins), регистрируемые в период прогона, реализуют на практике все функции уровня ядра — они ответственны за отправку и обработку данных. Как правило, они

обеспечивают поддержку протоколов, осуществляют транспортные функции низкого уровня и интерпретируют форматы данных. Сменные модули можно заменять в динамическом режиме, что упрощает задачу введения новой функциональности и даже позволяет вносить коррективы в характер веб-приложений.

Вызываемые функции (call-out functions) — это еще одно средство, при помощи которого приложения могут выходить за рамки функциональности, предоставляемой уровнем ядра. Они являют собой произвольные прикладные функции, которые вызываются перед или после подачи запросов модулям протоколов.

Каков характер отношений между общими утилитами и ядром? Общие утилиты, предоставляющие независимые от конкретной платформы функции, подходят для любых сетевых приложений. Те абстракции, которые предоставляет уровень ядра, напротив, специально предназначены для конструирования веб-приложений.

Уровень потоков предусматривает абстракцию потока данных, передаваемых между приложением и сетью.

На уровне доступа содержатся модули отдельных сетевых протоколов. Стандартный набор, который первоначально поддерживала библиотека libWWW, состоит из следующих протоколов: HTTP — базовый протокол WWW; NNTP (network news transport protocol, сетевой протокол передачи новостей) — протокол передачи сообщений в сети Usenet; WAIS (wide area information server, глобальный информационный сервер) — сетевая информационно-поисковая система; FTP (file transfer protocol, протокол передачи файлов), TELNET, rlogin, Gopher, локальная файловая система и TN3270. Многие из них в настоящее время утратили былое значение, однако есть и нововведения — к списку прибавился протокол HTTPS (HTTP secure, протокол защищенной передачи гипертекста). Введение новых модулей протоколов не представляет сложности, поскольку они строятся на абстракциях нижележащих уровней.

Верхний уровень, содержащий модули веб-приложения, фактически предоставляет функциональный набор для их написания: универсальные модули кэширования, регистрации информации, прокси-серверов (в целях преобразования протоколов) и шлюзов (для взаимодействия с защитными брандмауэрами), модули ведения журнала истории и т. д.

Выводы из опыта применения libWWW

На сегодняшний день, исходя из опыта конструирования самой библиотеки libWWW и множества основанных на ней приложений, уже можно делать некоторые выводы. При их формулировании учтены попытки разработчиков удовлетворить перечисленные в разделе 13.2 требования заинтересованных лиц: гетерогенность инструментальных средств для WWW, поддержка удаленного доступа в масштабе многочисленных сетей, децентрализация и т. д. Самой трудной задачей оказалось удовлетворение неожиданно возникшего требования о графическом оформлении. Таким образом, способность веб-приложений к росту стимулировала принятие в рамках libWWW новых решений и обусловила принятие ряда умозаключений.

- *Необходимы формализованные интерфейсы прикладного программирования* (application programming interfaces, APIs). Имеются в виду интерфейсы,

которые представляют функциональность библиотеки libWWW сконструированным на ее основе программам. Поскольку libWWW должна была содействовать разработке приложений на различных платформах и языках, зависимость спецификаций таких интерфейсов от конкретного языка не допускалась.

- *Функциональность и представляющие ее интерфейсы прикладного программирования должны быть многоуровневыми.* Разные приложения обращаются к разным ступеням абстракции обслуживания, организовать которые проще всего посредством уровней.
- *Библиотека должна поддерживать динамический, расширяемый набор характеристик.* Все эти характеристики должны быть заменяемыми, в том числе в период прогона.
- *Встраиваемые в программное обеспечение процессы должны быть многопоточными.* Веб-приложения должны предусматривать возможность одновременного выполнения нескольких функций. Связано это, в частности, с тем, что операции загрузки крупных файлов по медленному каналу связи занимают довольно много времени. Отсюда необходимость в одновременном поддержании нескольких потоков управления. Следовательно, функциональность, раскрываемая интерфейсами прикладного программирования, должна быть защищенной в такой степени, которая требуется для ее применения в многопоточной среде.

Как оказалось, поддержка библиотекой libWWW решения перечисленных задач реализована не так полно, как это можно было бы сделать. К примеру, ядро libWWW принимает ряд допущений о важнейших службах, что закрывает возможность динамической замены отдельных характеристик. Кроме того, поскольку библиотека libWWW не рассчитана на какую-либо конкретную платформу, опираться на однопоточную модель она не может. По этой причине в ней реализованы псевдопотоки, предоставляющие лишь часть необходимой функциональности. Наконец, большинство современных веб-приложений не поддерживают динамическое конфигурирование характеристик; для того чтобы зарегистрировать новые службы, их необходимо перезапустить.

Ранний вариант архитектуры «клиент–сервер», реализованный при помощи libWWW

На рис. 13.5 изображено представление размещения типичной клиент–серверной модели Всемирной паутины, построенной с участием libWWW. Там же показано представление декомпозиции на модули HTTP-клиента и серверных компонентов представления размещения. На этой схеме четко прослеживаются некоторые особенности libWWW. Во-первых, далеко не все элементы модели «клиент–сервер» строятся на ее основе. К примеру, от нее никак не зависит пользовательский интерфейс. Во-вторых, имена диспетчеров не обнаруживают точного соответствия с именами уровней. В то время как диспетчеры доступа, протоколов и потоков четко связаны с уровнями доступа и потоков, диспетчер кэширования обращается к службам прикладного уровня. Диспетчеры потоков в рамках пары «клиент–

сервер» управляют низкоуровневой передачей данных и тем самым обеспечивают для остальных элементов системы прозрачность информационного обмена.

Диспетчер пользовательского интерфейса ответствен за «облик и дух» пользовательского интерфейса клиента. С другой стороны, пользуясь расширяемостью набора ресурсов, доступных системе WWW, другой элемент — диспетчер представлений — может делегировать полномочия по отображению информации внешним программам (просмотра). Так происходит с ресурсами, которые известны системе, но которые диспетчер пользовательского интерфейса не поддерживает напрямую. В частности, большинство веб-браузеров делегируют внешней программе обязанности по отображению файлов PostScript и .pdf. Такое решение возникло как компромисс между стремлением к интеграции пользовательского интерфейса (способной стабилизировать его представление и, следовательно, повысить практичность), с одной стороны, и расширяемостью — с другой.

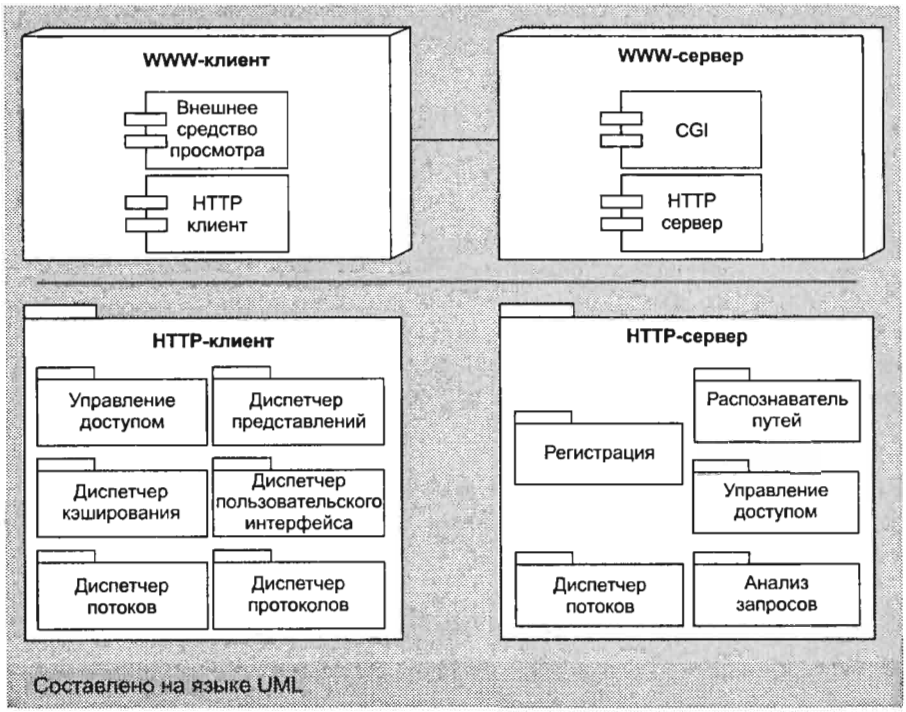


Рис. 13.5. Представление размещения клиент-серверной модели Всемирной паутины, а также представление декомпозиции на модули клиента HTTP и серверных компонентов

Диспетчер пользовательского интерфейса забирает запрос на поиск информации, поданный пользователем в виде URL, а затем передает эти данные диспетчеру доступа. Диспетчер доступа проводит проверку на наличие запрошенного URL в кэше и интерпретирует историческую навигацию (например, «Назад»). Если файл присутствует в кэше, он извлекается из диспетчера кэширования и передается диспетчеру представлений для отображения пользовательским интерфейсом или внешней программой просмотра. В случае отсутствия файла в кэше

диспетчер протоколов определяет тип запроса и запускает для его обслуживания соответствующий стек протоколов. С помощью последнего клиентский диспетчер протоколов передает запрос на сервер. Получив от сервера ответ, выраженный в форме документа, диспетчер потоков передает его диспетчеру представлений, который, в свою очередь, обеспечивает его отображение. При попытке установления соответствия типа документа внешним программам просмотра диспетчер представлений обращается за помощью к конфигурационному файлу управления статическим представлением (mimerc, mailcap и т. д.).

В обязанности HTTP-сервера входит обеспечение прозрачного доступа к файловой системе — благо именно она является источником документов, которые передаются в WWW. HTTP-сервер управляет доступом либо напрямую (для известных типов ресурсов), либо через посредника, называемого общим шлюзовым интерфейсом (common gateway interface, CGI). CGI обрабатывает те типы ресурсов, с которыми не может справиться сервер, а также рассматриваемые ниже расширения функциональности сервера. До появления этих расширений в WWW-серверах реализовывалось подмножество определенных HTTP-запросов, которые предусматривали поиск документов и связанной с ними метаинформации, а также исполнение программ, размещенных на стороне сервера, средствами CGI.

Получив запрос, диспетчер потоков на стороне сервера определяет его тип и при помощи распознавателя путей устанавливает путь к указанному URL. Для проверки полномочий доступа, которыми обладает запрашивающий клиент, HTTP-сервер обращается к таблице доступа. Если речь идет об обращении к защищенным данным, сервер может запустить сеанс парольной аутентификации клиента. Затем, приняв допущение об аутентификации, сервер обращается к файловой системе (расположенной за его границами) и записывает запрошенную информацию в выходной поток. Для исполнения программы средствами CGI ей предоставляется процесс (новый или освобожденный в результате опроса), после чего выходные данные исполняемой программы фиксируются диспетчером потоков сервера, и он, в свою очередь, передает их клиенту.

Как бы то ни было, CGI — это одно из основных средств, обеспечивающих расширяемость сервера; иначе говоря, оно удовлетворяет важнейшему требованию, определяющему развитие веб-приложений. Важность CGI как аспекта веб-приложений позволяет нам остановиться на этой теме чуть подробнее.

Общий шлюзовой интерфейс (CGI)

Информация, которую сервер возвращает клиенту, по большей части является статической; изменениям она подвержена только в рамках собственной файловой системы. Сценарии CGI, напротив, подразумевают возможность возврата динамической, индивидуальной для каждого запроса информации. Исторически сложившаяся роль CGI заключается в расширении функциональности сервера по части ввода информации, поиска и щелчков на изображениях. Впрочем, чаще всего CGI используется для создания *виртуальных документов* (virtual documents) — документов, которые синтезируются динамически в ответ на запросы пользователей. К примеру, если пользователь ищет в Интернете конкретную информацию, поисковая система генерирует ответ на каждый поисковый запрос;

сценарий CGI создает из ответа новый HTML-документ и возвращает его пользователю.

Сценарии CGI сохранили гибкость, характерную для ранних, основанных на библиотеке libWWW вариантов архитектуры. На рис. 13.5 CGI изображен как внешний по отношению к HTTP-серверу элемент. Сценарии CGI пишутся на разных языках — как компилируемых (C, C++, Fortran), так и интерпретируемых (Perl, VisualBasic, AppleScript и т. д.). Сценарии CGI позволяют разработчикам произвольным образом расширять функциональность сервера — в частности, производить информацию, возвращаемую сервером пользователю.

С другой стороны, поскольку в сценариях CGI может содержаться любая функциональность C, Perl и других языков, в системе защиты машины, на которой они устанавливаются, образуется серьезная брешь. К примеру, с помощью сценария (исполняемого как отдельный от сервера процесс) на хосте от имени удаленного пользователя можно запустить любую команду. Эта угроза, которую демонстрируют исполняемые на стороне сервера сценарии (в том числе CGI), заставила выставить к Всемирной паутине новое требование, касающееся повышения безопасности. Механизм реализации этого требования с помощью HTTPS описывается в следующем разделе.

Вероятно, наиболее важным результатом введения в архитектуру веб-технологии CGI явилась возможность «размещать» (put) в сети информацию — в дополнение к стандартной для сервера операции ее «получения» (get). Впрочем, это требование, выдвинутое еще к первоначальному проекту WWW, так и не удалось реализовать в полной мере. CGI позволяет пользователям размещать информацию только посредством специализированных механизмов — например, вносить записи в базу данных путем заполнения формы.

С помощью технологии CGI — в основном за счет обеспечения возможностей обработки сервером произвольных ресурсов и (ограниченного) размещения пользователями данных — удалось решить множество проблем, присущих первоначальному проекту libWWW. С другой стороны, у CGI есть несколько существенных

Таблица 13.2. Реализация задач по качеству, заданных для первоначальной версии WWW

Задача	Как реализована	Какие тактики применялись
Удаленный доступ	Организация WWW на основе сети Интернет	Применение предписанных протоколов
Способность к взаимодействию	Маскировка деталей конкретной платформы при помощи библиотеки libWWW	Общие абстрактные службы Информационная закрытость
Расширяемость программного обеспечения	Локализация расширений протоколов и типов данных в библиотеке libWWW; возможность введения подключаемых компонентов (апплетов и сервлетов)	Общие абстрактные службы Информационная закрытость Замена компонентов Конфигурационные файлы
Масштабируемость	Применение архитектуры «клиент–сервер», поддержание ссылок на данные, являющиеся локальными относительно местоположения ссылающихся данных	Введение параллелизма Снижение вычислительных издержек

недостатков. Один из них связан с безопасностью, другой — с переносимостью. Сценарии CGI, написанные на VisualBasic, AppleScript и C Shell, работают в средах Windows, Macintosh и UNIX соответственно. Переносить эти сценарии с одной платформы на другую довольно сложно.

Реализация первоначальных задач по качеству

Реализация первоначально сформулированных для Всемирной паутины задач по качеству — удаленному доступу, способности к взаимодействию, расширяемости и масштабируемости — расписана в табл. 13.2.

13.4. Еще одна итерация архитектурно-экономического цикла: эволюция вариантов веб-архитектуры систем электронной коммерции

Невероятный успех Всемирной паутины возбудил к ней серьезный интерес со стороны бизнес-сообщества, которое впоследствии через посредство архитектурно-экономического цикла оказало на ее архитектуру беспрецедентное воздействие. Коммерческие требования в архитектуре Всемирной паутины вскоре стали превалирующими. Большинство инновационных разработок в области веб-приложений было выполнено с подачи веб-сайтов B2B и B2C.

Согласно первоначальной концепции, Всемирная паутина должна была стать коллекцией документов, основанной на гипертексте. С точки зрения электронной коммерции WWW является коллекцией данных. Эти две концепции отчасти находятся в противоречии друг с другом. К примеру, сложности вызывает задача «проталкивания» информации пользователю. Общеупотребительная методика обновления данных предусматривает их регулярную перезагрузку; однако сами по себе изменения, происходящие с данными, не приводят к обновлению экрана. Другая проблема связана с кнопками «Назад» в различных браузерах — в определенных обстоятельствах их употребление приводит к выводу на экран устаревших данных.

Требования, которые в настоящее время выдвигают игроки электронной коммерции, отличаются от приведенных в разделе 13.2 первоначальных требований как по существу, так и по жесткости.

- *Высокая производительность.* Популярные веб-сайты ежедневно фиксируют десятки миллионов обращений, и пользователи рассчитывают на максимально возможное сокращение задержек. Покупатели не намерены терпеть отказы в ответ на свои запросы.
- *Высокая готовность.* Сайты электронной коммерции должны быть работоспособны 24 часа в сутки, 7 дней в неделю. Поскольку они никогда не закрываются, периоды простоя необходимо строго ограничивать — желательно, чтобы они не превышали нескольких минут в год.

- *Масштабируемость.* Вместе с ростом популярности веб-сайтов должна расти и их обрабатывающая способность — без этого невозможна обработка более серьезных объемов данных и поддержание приемлемого уровня обслуживания клиентов.
- *Безопасность.* Пользователи должны быть уверены в том, что секретную информацию, которую они отправляют через Сеть, никто не перехватит. Операторы веб-сайтов, в свою очередь, должны быть гарантированы от атак на свои системы (конкретнее — от похищения и модификации данных, отправки огромного количества запросов, приводящей к неготовности данных, порче данных и т. д.).
- *Модифицируемость.* Веб-сайты электронной коммерции постоянно (иногда — ежедневно) претерпевают обновления, в связи с чем возникает потребность в удобстве изменения размещаемой на них информации.

Архитектурное решение перечисленных требований лежит в плоскости *системной*, а не просто программной, архитектуры. Дело в том, что подобного рода системы по большей части состоят из коммерческих компонентов. В эту категорию, естественно, входят веб-серверы и веб-клиенты; кроме того, это базы данных, серверы безопасности, серверы приложений, прокси-серверы, серверы транзакций и т. д.

Эталонная архитектура современной системы электронной коммерции изображена на рис. 13.6. Функция взаимодействия программы просмотра с пользователем, как правило, осуществляется веб-браузером (кроме того, это может быть киоск, устаревшая система или какое-либо другое устройство с подключением к Сети). Функция бизнес-правил и бизнес-приложений обычно реализуется серверами приложений и транзакций. Уровень служб обработки данных в большинстве случаев воплощается в современной базе данных, хотя не исключаются и другие варианты — соединения с устаревшими системами и подключение к устаревшим базам данных. Изображенную схему часто называют n-звенной архитектурой (причем в данном случае $n = 3$). *Звено* (tier) представляет собой элемент функциональности, на который можно отвести отдельную физическую машину.

Рис. 13.6. Эталонная архитектура системы электронной коммерции

Типичная реализация архитектуры системы электронной коммерции включает некоторое количество звеньев (каждое из которых представляет собой связную группу программного обеспечения, как правило, состоящую из специализированных коммерческих компонентов), а также аппаратное обеспечение. Подобная конфигурация изображена на рис. 13.7, который, кроме того, иллюстрирует распределение программ между аппаратными блоками.

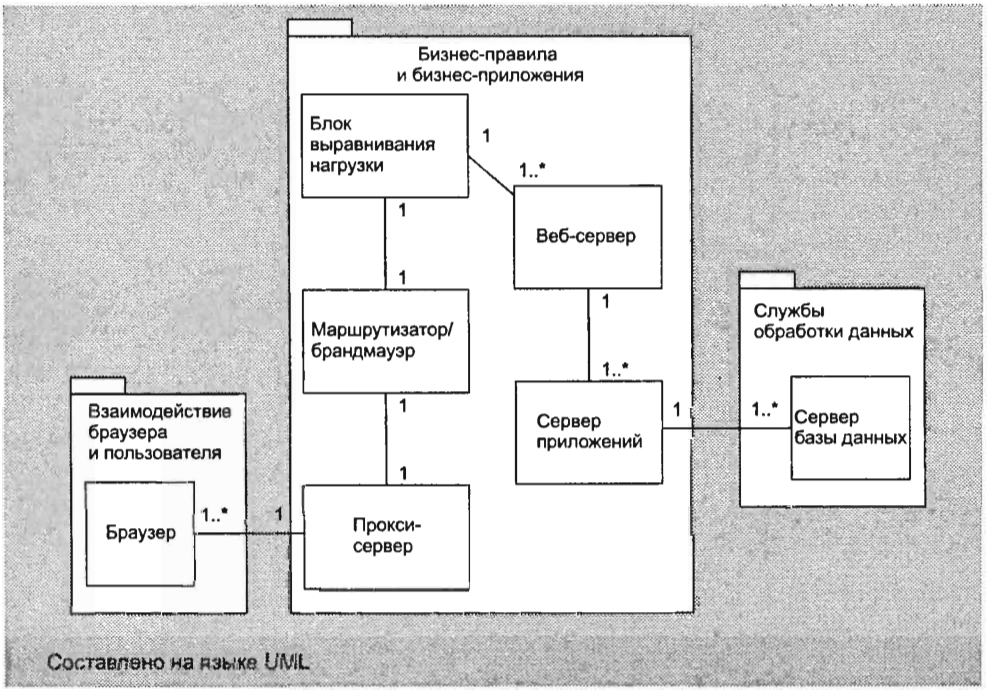


Рис. 13.7. Типичная система электронной коммерции

Надписи на рис. 13.7, повторяющие приведенные на рис. 13.6 функциональные элементы, в очередной раз свидетельствуют о том, что любая отдельная функция эталонной архитектуры может соответствовать нескольким звеньям архитектуры системы электронной коммерции. Веб-браузеры (клиенты) и веб-серверы с рис. 13.5 предстают здесь в виде элементарных компонентов. Таким образом, налицо тенденция перехода к компонентным системам, в которых снижается значимость внутренней компонентной структуры.

Изображенные на рис. 13.7 элементы мы намерены обсудить ниже в контексте атрибутов качества, реализации которых они способствуют.

Браузеры ради модифицируемости

Запросы на получение информации конечные пользователи, как правило, инициируют при помощи веб-браузера. Поддержка модифицируемости пользовательского интерфейса в современных веб-браузерах осуществляется весьма многообразно, причем наиболее очевидный способ не претерпел никаких изменений с момента появления Всемирной паутины, — пользовательский интерфейс, который отображает браузер, не «вшивается», а специфицируется средствами HTML. По крайней мере, так было раньше. Сегодня технологий создания сложных пользовательских интерфейсов стало гораздо больше. XML, Flash, ActiveX, Java-апплеты — это лишь некоторые средства, расширяющие стандартную палитру веб-инструментов (состоящую из графики и горячих точек) и способствующие

отображению средствами браузеров полностью программируемых интерактивных интерфейсов.

HTTPS ради безопасности

Поданный пользователем запрос необходимо передать на целевой веб-сайт. Передача обычно осуществляется средствами HTTP, однако если речь идет о секретной информации, вроде номеров кредитных карт и идентификационных номеров, применяется протокол HTTPS (HTTP Secure). В HTTPS в качестве подпротокола HTTP применяется протокол защищенных сокетов (Secure Sockets Layer, SSL) от компании Netscape. Для отправки зашифрованных запросов на службы TCP/IP он использует порт 443 (стандартный порт HTTP — 80). Шифрование данных в рамках SSL производится с помощью 128-битной пары открытого/секретного ключей — этот уровень шифрования при обмене небольшими пакетами коммерческой информации в рамках коротких транзакций считается достаточным.

Прокси-серверы ради производительности

Иногда запросы от браузеров проходят через прокси-сервер — элемент, ориентированный на повышение производительности веб-системы. Эти серверы кэшируют веб-страницы, к которым пользователи обращаются чаще всего, в результате чего необходимость в обращении к веб-сайту для их получения отпадает. (Кэши реализуют «многоэкземплярную» тактику.) За счет того, что они, как правило, максимально приближены к пользователям и зачастую даже находятся с ними в одной сети, вычислительные и коммуникационные издержки резко снижаются. В некоторых компаниях прокси-серверы ограничивают доступ сотрудников к тем или иным веб-сайтам. В таком случае прокси-сервер функционально сближается с брандмауэром.

Маршрутизаторы и брандмауэры ради безопасности

Запросы, отправляемые браузером (и в некоторых случаях обрабатываемые прокси-сервером), прибывают на маршрутизатор, расположенный в сети предприятия электронной коммерции. Иногда эта сеть защищается брандмауэром, а в некоторых случаях маршрутизатор перенаправляет HTTP-запросы на автономный брандмауэр. Маршрутизаторы часто проводят трансляцию сетевых адресов (network address translation, NAT) — иначе говоря, преобразуют IP-адреса из внешне видимых во внутренние. IP-адреса для любого возвращаемого с веб-сервера трафика, напротив, транслируются из внутренних во внешне видимые. NAT — это один из инструментов методики выравнивания нагрузки, к рассмотрению которой мы вернемся чуть позже.

Брандмауэр предотвращает несанкционированные информационные потоки и попытки доступа извне, осуществляя, таким образом, тактику «ограничения доступа». Брандмауэры подразделяются на несколько типов, из которых наи-

большим распространением пользуются *фильтры пакетов* (packet filters) и *прикладные посредники* (application proxies). Фильтры пакетов исследуют заголовки всех входящих пакетов TCP и IP, и в случае обнаружения злонамеренного поведения (например, попытки подключения через несанкционированный порт или отправки файлов неразрешенных типов) пакет отклоняется. В контексте передачи данных в WWW фильтры пакетов хороши тем, что исследуют каждый пакет в отдельности и не делают попыток фиксировать историю предыдущих сеансов связи.

Брандмауэры типа «прикладной посредник», как и предполагает их название, ориентированы на конкретное применение. Как правило, они интерпретируют прикладные протоколы и поэтому при фильтрации трафика опираются на известные модели поведения. К примеру, прикладной посредник может отказать в приеме HTTP-отклика, если перед этим на соответствующий узел не был отправлен HTTP-запрос. Брандмауэры такого типа работают значительно медленнее фильтров пакетов; связано это с тем, что, во-первых, они сохраняют историческую информацию значительного объема, а во-вторых, механизм обработки в них значительно сложнее.

Выравнивание нагрузки ради производительности, масштабируемости и готовности

Компоненты, обеспечивающие выравнивание нагрузки, а следовательно, производительность, масштабируемость и готовность, в обязательном порядке присутствуют на всех уважающих себя веб-сайтах электронной коммерции. Функция блока выравнивания нагрузки заключается в распределении «нагрузки» — входящих запросов HTTP и HTTPS — между участниками пула компьютеров с программным обеспечением веб-сервера. (Как мы говорили в главе 5, выравнивание нагрузки соответствует тактике «внедрения физического параллелизма»). Блок выравнивания нагрузки может перенаправить запрос на другой компьютер самостоятельно (и прозрачно) или отослать веб-клиенту ответ с соответствующей инструкцией. Несмотря на то что перенаправление проходит незаметно для конечного пользователя, фактически оно приводит к дополнительному круговому обращению.

При выборе компьютера для перенаправления трафика блок выравнивания нагрузки основывается на алгоритме кругового обслуживания или на знании вычислительных и нагрузочных характеристик подключенных компьютеров. Поскольку исполняемая блоком выравнивания нагрузки роль посредника распространяется на пул компьютеров, введение в этот пул дополнительных машин не требует модификации внешних интерфейсов. Таким образом, блок выравнивания нагрузки обеспечивает масштабируемость производительности — точнее говоря, тот тип этой характеристики, который называют горизонтальным масштабированием (он подразумевает введение дополнительных экземпляров данного ресурса).

Кроме того, блок выравнивания нагрузки способен отслеживать живучесть подключенных к нему компьютеров. Если один из них выходит из строя, его трафик передается другим участникам пула. Таким способом гарантируется готовность.

Веб-серверы ради производительности

Наконец, запрос HTTP или HTTPS прибывает на веб-сервер. Старые веб-серверы (один из таких изображается на рис. 13.5) в основном были однопоточными. Современные версии веб-серверов отличаются многопоточностью и применением пулов потоков, распределяемых для обработки входящих запросов. Располагая пулом с потоками, готовыми для обслуживания новых входящих запросов, многопоточный сервер снижает вероятность образования узких мест (а следовательно, и задержек) при обработке многочисленных «долгоиграющих» запросов HTTP или HTTPS (к таковым относятся операции проверки достоверности данных кредитных карт). Таким образом реализуется тактика «введение параллелизма».

Вертикальное масштабирование (введение более мощных экземпляров ресурса) достигается за счет замены существующих серверов более производительными машинами, способными параллельно запускать большее количество потоков.

Завершив анализ запроса, веб-сервер отправляет его серверу приложений. Тот обычно отвечает с помощью служб базы данных.

В главе 16 мы разберем систему корпоративных JavaBeans (Enterprise JavaBeans) — современную методику реализации веб-серверов.

Серверы приложений ради модифицируемости, производительности и масштабируемости

От веб-сервера запрос переходит к серверу приложений. «Сервер приложений» — это широкий (и, согласно распространенному мнению, не совсем ясный) термин, который обозначает класс приложений, действующих «посередине» *n*-звенной архитектуры. Речь идет о бизнес-правилах и бизнес-приложениях. Такие серверы реализуют бизнес-логику и возможность соединения, которые регламентируют взаимодействие серверов с клиентами. Тенденция к введению серверов приложений позволила разгрузить традиционные «жирные» клиенты и передать часть их функциональности среднему звену. Кроме того, они фокусируют базы данных на хранение, поиск и анализ данных, снимая с них обязанность по контролю над дальнейшим использованием этих данных.

Простые серверы приложений, как правило, состоят из интегрированной среды разработки (integrated development environment, IDE) и сервера исполнения. Среды IDE поддерживают программные модели наподобие COM (а в последнее время — .NET), CORBA или J2EE (последняя рассматривается в главе 16). Помимо этого, многие серверы приложений предусматривают наборы общеупотребительных служб для оперативного создания бизнес-приложений и приложений электронной коммерции — в частности, предназначенных для выписывания счетов, управления запасами, документооборотом и взаимодействием с заказчика-

ми. На более высоком уровне (в категориях стоимости, сложности и функциональности) стоят приложения обработки и отслеживания транзакций. Блоки отслеживания и процессоры транзакций взаимодействуют с базами данных и координируют такие задачи, как распределенные транзакции (в том числе объединение данных из нескольких источников), организация очередей, поддержание целостности транзакций и выравнивание рабочей нагрузки (в этом они схожи с упомянутыми выше блоками выравнивания нагрузки).

Базы данных ради производительности, масштабируемости и готовности

В конце концов запрос на обслуживание достигает базы данных, где преобразуется в команду добавления, модификации или поиска информации. Варианты архитектуры современных баз данных во многом повторяют атрибуты качества представленной на рис. 13.7 системы электронной коммерции. Во многих случаях для реализации производительности, масштабируемости и высокой готовности в них вводится внутренняя репликация. Кэширование также способствует повышению производительности.

13.5. Реализация задач по качеству

Взятые вместе, вышеописанные элементы позволяют реализовывать в веб-системах электронной коммерции самые жесткие задачи по безопасности, высокой готовности, модифицируемости, масштабируемости и высокой производительности. Как это происходит, показано в табл. 13.3.

Таблица 13.3. Реализация задач по качеству в веб-архитектуре систем электронной коммерции

Задача	Как реализуется	Тактики
Высокая производительность	Выравнивание нагрузки, трансляция сетевых адресов, многоэкземплярная тактика	Введение параллелизма; расширение ресурсов; прокси-серверы
Высокая готовность	Резервные процессоры, сети, базы данных и программы; выравнивание нагрузки	Активное резервирование; транзакции; введение параллелизма
Масштабируемость	Обеспечение горизонтального и вертикального масштабирования; выравнивание нагрузки	Общие абстрактные службы; применение предписанных протоколов; введение параллелизма
Безопасность	Брандмауэры; шифрование открытым/секретным ключами в общедоступных сетях	Ограничение доступа; целостность; ограничение внешних воздействий
Модифицируемость	Разделение функциональности браузера, проектного решения базы данных и бизнес-логики на отдельные звенья	Общие абстрактные службы; семантическая связность; посредник; стабильность интерфейса

13.6. Архитектурно-экономический цикл сегодня

Если проанализировать текущее состояние Всемирной паутины по прошествии нескольких итераций архитектурно-экономического цикла, в глаза бросится целый ряд явлений.

- Организации, формирующие техническую базу, подразделяются на несколько типов. Согласно наиболее общей классификации, их можно разделить на поставщиков услуг и контент-провайдеров. Поставщики услуг производят программные продукты, обеспечивающие функционирование Всемирной паутины: браузеры, серверы, базы данных, серверы приложений, технологии защиты (в частности, брандмауэры), серверы транзакций, сети и маршрутизаторы. Контент-провайдеры создают информацию, которая размещается в Сети. В обеих областях наблюдается жестокая конкуренция.
- Помимо W3C, значительное влияние на эволюцию Всемирной паутины оказывает ряд проектов с открытым кодом — в частности, проект Apache.
- CERN никоим образом не участвует в развитии WWW.
- Языки с поддержкой веб-технологий (в особенности это касается Java) вносят свои коррективы в способы разработки и доставки функциональности в этой среде. (Пример конструирования веб-приложений с помощью Enterprise JavaBeans описывается в главе 18.)
- Превращение Всемирной паутины в распределенную среду разработки привело к появлению ряда новых организаций и продуктов. К примеру, UDDI (Universal Description, Discovery and Integration, универсальная система предметного описания и интеграции) организует распределенные регистры веб-служб. Эти службы используются в качества стандартных блоков распределенных веб-приложений.

На рис. 13.8 изображен архитектурно-экономический цикл Всемирной паутины в ее современном виде.

В роли заказчиков выступают производители программных серверов и браузеров, а также поставщики услуг и контент-провайдеры. Конечные пользователи — это население планеты. Роль архитектора разделена между W3C и другими консорциумами (в частности, UDDI и Apache), с одной стороны, и несколькими влиятельными компаниями (Sun, Microsoft и AOL/Netscape) — с другой. По всем остальным показателям ABC практически не претерпел изменений — за исключением, пожалуй, того, что техническая база теперь включает саму Всемирную паутину, а следовательно, перечень атрибутов качества дополняется требованием о прямой совместимости.

Возвратный цикл ABC мы разобрали в разделе 1.1. Существование системы создает для компании-разработчика и ее заказчиков новые коммерческие возможности. В случае со Всемирной паутиной разработавшая ее компания — CERN — решила дистанцироваться от этого продукта и сосредоточиться на исследованиях в области ядерной физики; из-за этого коммерческими возможностями, созданными возвратным циклом ABC, воспользовались другие организации.

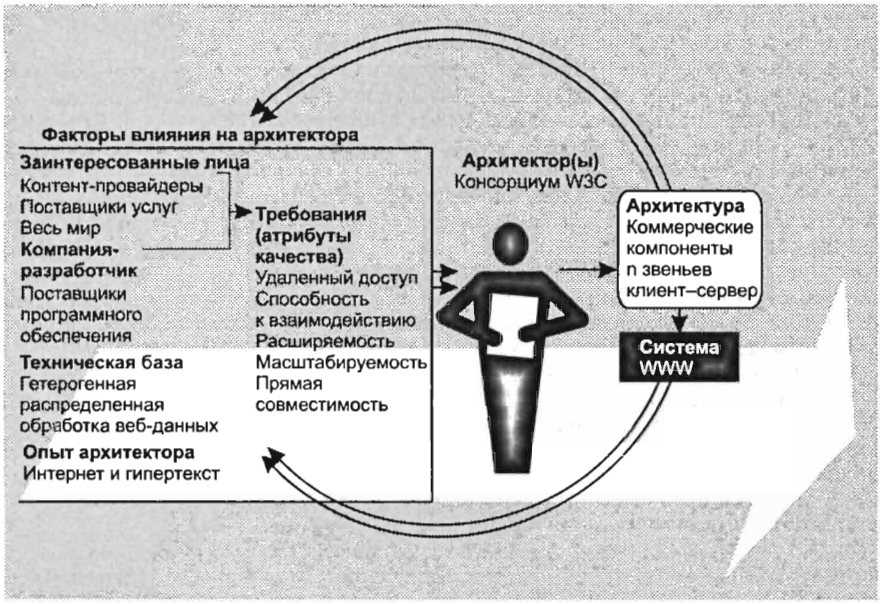


Рис. 13.8. Современный архитектурно-экономический цикл Всемирной паутины

13.7. Заключение

Своим успехом Всемирная паутина обязана способу реализации в ее архитектурных структурах желаемых атрибутов качества, а также преобразованиям этих структур в ответ на появление новых требований. Ее лавинообразное развитие свидетельствует о том, что всего за несколько лет архитектурно-экономический цикл был пройден неоднократно, создавая с каждой итерацией новые коммерческие возможности, требования и технические задачи.

О ТОМ, КАК ВСЕМИРНАЯ ПАУТИНА ИЗМЕНИЛА ДЕЛОВОЙ МИР: AMAZON.COM

К моменту открытия в 1995 году своего виртуального представительства Amazon.com, продававший более 1 миллиона изданий, по своему масштабу был уже на порядок крупнее среднестатистических книжных магазинов. От традиционных магазинов он отличался и по многим другим показателям. Причиной столь разительных отличий оказалась электронная ориентация Amazon — реализация задач и предложение продуктов через Сеть.

Будучи электронным магазином, Amazon изменил мир (по крайней мере, деловой). В частности, не затрачивая средства на издательскую деятельность, он смог себе позволить торговать книгами от независимых авторов и мелких издательств. Он изменил сам механизм покупки книг, чему поспособствовали издававшиеся в Сети отзывы читателей, обзоры, служба личных рекомендаций, доставляемые по электронной почте извещения о выходе новых книг любимых получателями авторов и т. д. Наконец, поскольку Amazon делегировал большинство операций сторонним организациям, а следовательно, избавился от значительной доли издержек, характерных для традиционной розничной книготорговли, магазин смог удержать цены на низком уровне.

Любой клиент Amazon.com может рассчитывать на индивидуальное обслуживание — в частности, на предложения книг, сходных с теми, которые он купил или просто просмотрел.

Такие возможности Amazon предоставляет исключительно за счет мощнейшей информационно-технологической инфраструктуры и возможностей анализа данных.

Amazon — это не просто книготорговая фирма. Скорее, это посредник и информационный брокер. Вдобавок к ассортименту книжной продукции предприятию удалось завербовать обширную и постоянно расширяющуюся сеть продавцов и покупателей. Это своеобразный «торговый центр», получающий процент с каждой совершенной сделки и комиссию за ссылки на сторонние веб-сайты.

По большому счету, информационно-технологическая инфраструктура Amazon имеет весьма отдаленное отношение к книжной продукции. Менеджмент Amazon пришел к этому заключению вполне своевременно и поэтому постепенно взял на себя функции розничной продажи игрушек, сотовых телефонов, лекарств, фотоаппаратов, программного обеспечения, автозапчастей, корма для домашних животных — то есть практически всех видов продуктов, которые можно продавать и доставлять в любые уголки земного шара. Всех этих задач никогда нельзя было бы добиться вне инфраструктуры Сети.

На сегодняшний день Amazon.com, обслуживающий клиентов более чем в 220 странах, претендует на статус крупнейшего сетевого магазина в мире. В его активе — пять международных веб-сайтов и примерно 20 миллионов зарегистрированных покупателей (немногим меньше, чем население Канады!). Процент повторных сделок достигает 70 % — традиционные предприятия розничной торговли могут лишь мечтать о таком показателе. На момент написания этих строк Amazon не удалось закрыть год с положительным балансом, но руководство магазина рассчитывает получать прибыль начиная с 2003 года.

Amazon — это лишь один из примеров воздействия Всемирной паутины на жизнь людей (по крайней мере, в сфере розничной торговли); впрочем, вероятно, он один из самых впечатляющих.

— RK

13.8. Дополнительная литература

Читателям, желающим поподробнее узнать о концепции гипертекста, мы рекомендуем ознакомиться с работой [Bush 45] и специализированным выпуском CACM [CACM 88].

Информации об истории и развитии Всемирной паутины больше всего в самой Всемирной паутине. Среди наших источников — [Berners-Lee 96a], [Gray (http://www.mit.edu/people/mkgray/net)] и [Zakon (http://www.zakon.com/robert/internet/timeline)].

Обширная информация о libWWW содержится в справочной библиотеке W3C, расположенной по адресу http://www.w3.org/pub/WWW/Library.

Добротное исследование по проблемам сетевой безопасности и криптографии, включая все аспекты защиты во Всемирной паутине, содержится в работе [Stallings 99]. Вопросы производительности в системах электронной коммерции хорошо освещены в издании [Menasce 00].

Архитектурному стилю, характерному для веб-приложений, посвящена работа [Fielding 96]. Сравнение образцов современных веб-серверов приводится в [Hassan 00]; именно оттуда мы заимствовали (и переработали) схему клиент-серверной архитектуры, показанную на рис. 13.5.

Результаты исследования по употреблению веб-серверов, проведенного в мае 2001 года компанией Netcraft, опубликованы по адресу http://www.netcraft.com/survey/.

13.9. Дискуссионные вопросы

1. Мы обозначили ряд атрибутов качества, реализация которых в WWW сделала возможным ее ошеломляющий успех: способность к взаимодействию, переносимость, удаленный доступ, расширяемость и масштабируемость. Какое из них, по вашему мнению, оказало на развитие Паутины решающее влияние? Если бы одним из этих качеств пришлось пожертвовать, смогла бы она стать настолько же успешной? Какие компромиссные решения в архитектуре приложений на основе libWWW приходится принимать вследствие сочетания названных задач по качеству?
2. Производительности не оказалось в списке первоначальных задач по качеству, которые следовало реализовать во Всемирной паутине. Для успешной системы это довольно необычно. Как вы считаете, почему успех ее все-таки настиг? Можно ли из этого сделать какие-то выводы относительно будущего развития вычислительной техники?
3. Какие образцы и тактики прослеживаются в элементах архитектуры, показанных на рис. 13.4, 13.5 и 13.7?

Часть 4
ОТ ОДНОЙ СИСТЕМЫ К МНОЖЕСТВУ

В четвертой части мы продолжаем разговор об архитектурно-экономическом цикле. На протяжении первой, второй и третьей частей мы плавно перемещались от характеристики архитектора к проверке архитектуры. Четвертая часть, посвященная вопросам конструирования на основе архитектуры множества систем, содержит также примеры линеек системных продуктов. Проблема подвергается анализу с позиций: 1) базовой технологии линейки продуктов; 2) отдельной компании-производителя систем управления огнем для военных кораблей; 3) общеотраслевой архитектуры; 4) компании, разрабатывающей продукты на основе общеотраслевой архитектуры, и 5) организации, которая при проектировании своих систем прибегает к использованию коммерческих компонентов.

Линейки программных продуктов подразумевают возможность повторного использования самых различных активов — от требований и планов тестирования до персонала. Такую способность им придает архитектура. В главе 14 речь пойдет об определении и разработке архитектуры линеек продуктов. В этом контексте мы будем часто обращаться к организационным вопросам, поскольку, как вы уже знаете, между архитектурой и компанией-разработчиком существует неразрывная связь.

В главе 15 приводится первый в этой части конкретный пример. Объектом нашего внимания станет шведская компания CelsiusTech, создавшая линейку систем управления огнем для военных судов. Помимо собственно архитектуры мы обсудим то, каким образом в результате ориентации на линейку продуктов претерпела изменения организационная структура и культура компании.

Впрочем, CelsiusTech практикует создание архитектуры в расчете на производство нескольких продуктов. Существуют также архитектуры, ориентированные на целые отрасли промышленности. К примеру, корпоративная архитектура Java 2/система корпоративных JavaBeans (Java 2 Enterprise Edition/Enterprise JavaBeans, J2EE/EJB) — архитектурная спецификация для информационных вебсистем — исполняет роль базовой архитектуры продуктов, разрабатываемых разными компаниями. Архитектурные решения, характерные для J2EE/EJB, и возможные в ее контексте компромиссы рассматриваются в главе 16.

Inmedius — одна из компаний, обращающихся к архитектуре J2EE/EJB, — специализируется на решениях для квалифицированных рабочих (в частности, для техников по обслуживанию), которые, не имея возможности пользоваться настольными компьютерами и лишь изредка добираясь до ноутбуков, плотно работают с разнообразными мобильными платформами. О том, как Inmedius удалось разработать решение, основанное на беспроводной технологии, переносимых и ручных (handheld) компьютерах, рассказывается в главе 17.

В главе 18 анализируется ситуация конструирования единичной системы на основе архитектуры и ряда коммерческих компонентов. Мы поговорим о том, что в этом случае следует доработать.

Наконец, мы отдадим дань своему любимому занятию — прогнозированию будущего развития программной архитектуры. Свои догадки (не более того, поверьте!) о том, что нас ждет через несколько лет, мы излагаем в главе 19.

Глава 14
Линейки программных продуктов. Повторное использование архитектурных средств

(в соавторстве с Линдой Нортроп)

> Первым на необходимость повсеместного введения практики многократного использования программных компонентов указал Макилрой. Было это в 1969 году. С тех пор сообщество разработчиков ПО беспрерывно бьется над осуществлением этой задачи. Отсюда закономерный вопрос: если преимущества повторно используемых программных компонентов настолько ошеломляющи, почему они еще не шествуют по компьютерным наукам победным маршем?
>
> *Грэди Буч* [Booch 94]

14.1. Обзор

На разработку программной архитектуры, занимаются которой далеко не последние специалисты, уходит много времени и усилий. Поэтому желание увеличить выгоду путем повторного использования архитектуры в нескольких системах представляется вполне естественным. Компании, обладающие серьезным опытом производства вариантов архитектуры, рассматривают их как ценную интеллектуальную собственность и постоянно ищут возможности получения от нее дополнительного дохода и снижения издержек. Повторное использование архитектуры позволяет достичь обеих этих целей.

Речь в настоящей главе пойдет о явном, планируемом повторном использовании программной архитектуры (равно как и других активов) в рамках семейства родственных систем. Разработка нескольких сходных систем на основе одной архитектуры (а также элементов, связанных с этой архитектурой) создает для компании значительные преимущества — конкретнее, снижает стоимость конструирования и сокращает время выхода на рынок. Именно этим объясняется привлекательность *линейки программных продуктов* (software product line), определяемой как:

> Набор преимущественно программных систем с общим контролируемым множеством характеристик, которые удовлетворяют конкретные потребности определенного сегмента рынка или выполняют определенную задачу и разрабатываются в установленном порядке на основе общего набора базовых средств. [Clements 02b, 5]

Итак, мы имеем дело с набором повторно используемых средств, в состав которого входят базовая архитектура и наполняющие ее общие (а иногда приспосабливаемые) элементы. Кроме того, здесь не обойтись без проектных решений и их документации, руководств пользователя, а также артефактов руководства проектом: бюджетов, графиков, планов тестирования программ и контрольных примеров. Вскоре мы покажем, что значительную роль в деле осуществления этой схемы играет правильное определение области действия линейки продуктов.

После успешного запуска линейки продуктов все повторно используемые средства — те, которые можно применить в нескольких системах и которые дешевле сохранить, чем разработать заново, — заносятся в *фонд базовых средств* (core asset base). В идеале, в базовых средствах следует предусматривать изменяемые параметры — точки, в которых возможно быстрое запланированное приспособление. Конструирование систем в рамках успешной линейки продуктов сводится к обращению к нужным средствам, их приспособлению согласно потребностям текущей системы и, наконец, ее сборке. Если даже для отдельных продуктов линейки и потребуется разработка дополнительного программного обеспечения, его удельный вес вряд ли превысит 20 %. Интеграция и тестирование в таком случае становятся основными операциями, вытесняя с этой позиции проектирование и кодирование.

Линейки продуктов в промышленном производстве не есть нововведение. Многие историки утверждают, что концепция эта появилась в самом начале XIX века, когда Эли Уитни (Eli Whitney) начал собирать винтовки из взаимозаменяемых частей; впрочем, есть и более ранние примеры. Сегодня линейки продуктов есть в компаниях Boeing, Ford, Dell и даже McDonald's. Каждый из этих производителей извлекает из общности свои выгоды. К примеру, модели Boeing 757 и 767 разрабатывались одновременно, и, несмотря на то, что эти два воздушных судна сильно отличаются друг от друга, их узлы совпадают примерно на 60 %.

Линейки *программных* продуктов, основанные на общности их участников, являют собой инновационную тенденцию в программной инженерии, которая к тому же неуклонно набирает популярность. Каждый заказчик выставляет к продукту собственные требования, для выполнения которых производитель должен проявлять гибкость. Так вот, линейки программных продуктов упрощают создание систем, ориентированных на удовлетворение потребностей конкретных заказчиков или групп.

Возможности повышения эффективности в категориях издержек, сроков выхода на рынок и продуктивности (при удачном построении линейки продуктов) захватывают дух. Примеров тому множество.

- Благодаря методике линеек продуктов Nokia производит от 25 до 30 моделей сотовых телефонов в год (хотя раньше за аналогичный период удавалось создать всего 4 модели).
- Компании Cummins, Inc. удалось сократить сроки производства программного обеспечения для дизельных двигателей с года до недели.
- Со своим семейством пейджеров Motorola добилась 400-процентного прироста продуктивности.
- По сведениям компании Hewlett-Packard, сроки выхода на рынок в рамках ее семейства печатных систем сократились в семь раз, а продуктивность увеличилась в шесть раз.
- На разработку первого продукта в рамках заказанного Национальным управлением воздушно-космической разведки США семейства наземных станций системы спутниковой связи потребовалось всего 10 % от запланированного числа разработчиков, а количество дефектов снизилось на 90 %.

Залогом успеха линейки продуктов является наличие согласованной стратегии, распространяющейся на программную инженерию, техническое руководство и управленческую структуру компании. Следуя основному предмету нашей книги, мы поговорим о тех аспектах программной инженерии, которые касаются программной архитектуры. Тем не менее не стоит забывать, что удачную линейку продуктов невозможно создать вне взаимодействия всех ее аспектов.

14.2. За счет чего работают линейки программных продуктов?

Смысл линейки программных продуктов заключается в стратегическом повторном использовании средств многократного применения для производства семейств продуктов. Почему линейки продуктов так привлекательны в глазах производителей и разработчиков? Все дело в том, что за счет общности продуктов — при условии грамотного к ней подхода и применения концепции повторного использования — можно добиться значительной производственной экономии. Потенциал повторного использования широк и обширен. В частности, он распространяется на следующие аспекты.

- *Требования.* Требования по большей части достаются в наследство от предыдущих систем, а потому допускают повторное использование. И тем не менее необходимость в проведении анализа требований остается.
- *Архитектурное проектирование.* На разработку архитектуры программной системы компании вынуждены направлять своих лучших сотрудников, которые прикладывают к достижению этой цели серьезные усилия. Как вы уже знаете, сформулированные для системы задачи по атрибутам каче-

ства — производительности, надежности, модифицируемости и т. д. — к моменту завершения работы над архитектурой в основном решаются или, напротив, отклоняются. Ошибки в архитектуре приводят к печальным для системы последствиям. Если же речь идет о новом продукте в рамках линейки, этот важнейший этап проектирования можно пропустить за счет введения готового решения.

- *Элементы.* Программные элементы допускают применение в разных продуктах. В отличие от банального повторного использования кода повторное использование элементов позволяет избежать операции первоначального проектирования, проведение которой зачастую связано со значительными трудностями. Удачные проектные решения, если они зафиксированы, подлежат многократному применению; ошибок при проектировании удается избежать. В частности, это касается проектирования интерфейса элемента, его документации, планов и процедур тестирования, а также любых моделей (например, моделей производительности), направленных на прогнозирование или измерение его поведения. Одним из повторно используемых наборов элементов является пользовательский интерфейс системы, воплощающий обширный и необходимых набор проектных решений.
- *Моделирование и анализ.* Модели производительности, анализ возможности планирования, проблемы распределенных систем (например, испытания на предмет взаимоблокировок), распределение процессов между процессорами, схемы отказоустойчивости, политики сетевой нагрузки — все эти элементы переходят от одного продукта к другому. По сведениям компании CelsiusTech (см. главу 15), ей почти полностью удалось устранить одну из основных проблем, характерных для распределенных систем реального времени, разработкой которых она занимается. Создавая в рамках линейки очередной продукт, ее сотрудники уверены, что все проблемы с соблюдением временных требований уже решены, а ошибки, связанные с распределенной обработкой данных — синхронизацией, загрузкой сети и взаимоблокировкой, — устранены.
- *Тестирование.* Необходимость в повторной разработке планов и процессов тестирования, контрольных примеров и данных, тестирующих программ и каналов связи, требуемых для оповещения о проблемах и их устранения, отпадает.
- *Планирование проекта.* Поскольку опыт есть самый лучший показатель будущей производительности, операции составления бюджета и планирования приобретают более предсказуемый характер. Декомпозицию обязанностей по разработке системы не приходится проводить для каждого продукта. Определить состав и размер групп разработчиков становится значительно проще.
- *Процессы, методы и инструменты.* Процедуры и средства управления конфигурациями, планы документирования и процессы утверждения, инструментальная среда, процедуры генерации и распределения системы, стандарты кодирования и множество других повседневных инженерных операций

вспомогательного характера — все они переносятся из одного продукта в другой. В наличии имеется и процесс программной разработки в целом.

- *Специалисты.* Общность практической деятельности позволяет без труда перебрасывать сотрудников из одного проекта в другой, согласно текущей ситуации. Знания, которыми обладают специалисты, применимы в масштабах всей линейки.
- *Примеры систем.* Размещенные продукты исполняют роль высококачественных демонстрационных прототипов, инженерных моделей производительности, безопасности, защиты и надежности.
- *Устранение дефектов.* Линейки продуктов способствуют повышению качества — в каждой последующей системе учитывается опыт устранения дефектов у ее предшественников. С каждой новой реализацией разработчик и клиенты обретают дополнительную степень уверенности в успехе. Чем сложнее система, тем выгоднее решать вечно досаждающие проблемы производительности, распределения и надежности в масштабе целого семейства.

Линейки программных продуктов основываются на повторном использовании. В то же время, как видно из эпиграфа к настоящей главе, попытки внедрить повторное использование в программной инженерии предпринимаются уже много лет, и успех этого предприятия пока что под вопросом — ожидания почти всегда оказываются радужнее реальных результатов. Причина неудач отчасти кроется в том, что до последнего времени технология повторного использования распространялась под лозунгом «сконструируй, и все будет!». Любая библиотека многократного применения состоит из элементов предыдущих проектов; от разработчиков, прежде чем переходить к кодированию новых элементов, требуется ознакомиться с ее содержанием. У этой схемы действий слишком много недостатков. Если библиотека невелика, разработчик, однажды не обнаружив нужного элемента, потеряет всякое желание продолжать поиски. Если же она слишком обширна, провести поиск будет труднее. Если искомые элементы невелики, их проще переписать, чем найти в библиотеке и модифицировать. Если они крупные, подробно разобраться в их назначении очень сложно, а вероятность того, что они в точности подойдут для нового приложения, крайне невелика. Происхождение элементов, хранящихся в библиотеках повторного использования, как правило, в лучшем случае, неочевидна. Разработчик, таким образом, не может быть полностью уверен в назначении и надежности элемента, а кроме того, он не знает условий, в которых проводилось его тестирование. В то же время точное соответствие между атрибутами качества, требуемыми в новом приложении, с одной стороны, и реализуемыми элементами библиотеки — с другой, практически никогда не встречается.

В большинстве случаев оказывается, что элементы библиотеки написаны для архитектурной модели, которой разработчик новой системы не пользуется. Даже если вам удастся найти нужный элемент, реализующий нужные атрибуты качества, совершенно не факт, что вам подойдет тип этого архитектурного элемента (например, искали объект, а нашли процесс) и его протокол взаимодействия, что он не будет противоречить принятым для нового приложения политикам обработки ошибок или восстановления после отказа, и т. д.

Линейки программных продуктов устанавливают жесткий контекст повторного использования: архитектура определена, функциональность задана, атрибуты качества известны. В библиотеку повторного использования (согласно терминологии линеек продуктов, она называется фондом базовых средств) попадают только те элементы, которые сконструированы в расчете на многократное применение в рамках данной линейки. Таким образом, повторное использование становится стратегическим, запланированным, утрачивает случайный характер.

14.3. Определение области действия

Область действия линейки продуктов регламентирует участие в ней систем. Иначе говоря, это перечень систем, которые компания (1) готова и (2) не готова конструировать в рамках линейки. В ходе разграничения области действия в пространстве всех возможных систем вычерчивается баранкообразная фигура (рис. 14.1). В центре этой фигуры изображаются системы, согласованные с линейкой продуктов, которые данная компания может и собирается сконструировать. Системы, оказавшиеся вне фигуры, находятся за пределами области действия — таким образом, они признаются неприспособленными для рассматриваемой линейки продуктов. Вхождение в линейку систем, перекрывающих границы фигуры, находится под вопросом; для этого требуются некоторые усилия и дифференцированное рассмотрение. Если, к примеру, взять линейку продуктов систем автоматизации конторских работ, то программа-планировщик мероприятий в конференц-зале в нее войдет, а система моделирования условий полета — нет. Специализированная система поиска в локальной сети имеет шансы попасть в линейку при соблюдении двух условий: во-первых, разумных сроков производства и, во-вторых, наличия серьезных стратегических мотивов для ее создания (например, вероятность спроса на аналогичный продукт со стороны будущих заказчиков).

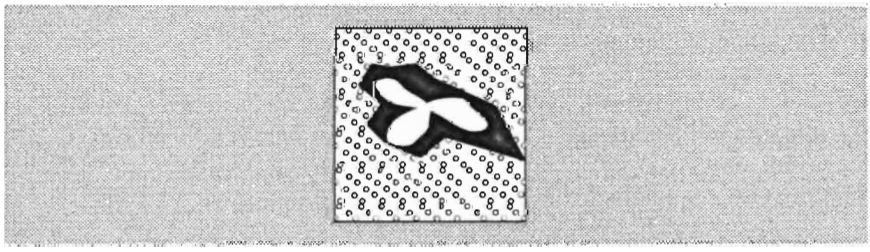


Рис. 14.1. Пространство всех возможных систем подразделяется на несколько участков: внутри области действия (белая), вне области действия (в крапинку) и предполагающие дифференцированное рассмотрение (черная)[1]

Область действия выражает наилучший прогноз компании относительно запросов на конструирование продуктов в обозримом будущем. Исходные данные в процессе определения области действия поставляют специалисты по стратегическому планированию, сотрудники отдела маркетинга компании, аналитики

[1] Приводится по изданию [Clements 02b] (адаптированная версия).

предметной области, способные систематизировать сходные системы (как существующие, так и проектируемые), а также эксперты по технологии.

Наличие разграниченной области действия во многом определяет дальнейшую судьбу линейки продуктов. Слишком узкая область (предполагающая изменчивость лишь в нескольких характеристиках) может не оправдать вложенных средств, поскольку вывести из нее достаточное количество продуктов не удастся. Не в меру широкое определение области (когда продукты различаются не только по характеристикам, но и по видам), вероятнее всего, приведет к высоким затратам на разработку отдельных продуктов на основе базовых средств, а потому достичь серьезной прибыли будет сложно. Уточнить область действия можно на этапе первоначального учреждения линейки продуктов, а также, исходя из стратегии принятия линейки продуктов (см. раздел «Стратегии принятия»), в любой последующий момент.

Задача обнаружения общности на этапе определении области действия не является основной — творчески мыслящий архитектор способен найти точки общности между любыми двумя системами. Значительно важнее найти такую общность, за счет которой можно будет существенно снизить затраты на конструирование последующих систем.

В контексте определения области действия следует учитывать не только конструируемые системы. Существенную помощь в этом процессе оказывают сведения о сегментации рынка и стиль взаимодействия с клиентами. К примеру, компания Philips — голландский производитель бытовой электроники — ведет отдельные линейки продуктов по домашней видеоаппаратуре и системам передачи цифровых видеосигналов. С одной стороны, обе линейки связаны с обработкой видеосигналов. С другой — первая ориентирована на рынок товаров массового потребления, в рамках которого покупатели обладают очень низким уровнем знаний, а вторая — на узкий (по количеству покупателей) сегмент рынка, состоящий из специалистов в соответствующей области. При разработке продуктов учитывается как опыт покупателей, так и сложность обслуживания продукта покупателем. Выявленных различий оказалось вполне достаточно для того, чтобы компания Philips отказалась от попыток создания единой для обоих рынков линейки продуктов.

Линейки продуктов с узкой областью действия позволяют конструировать специализированные инструменты для специфицирования новых продуктов.

- FAST — процесс, поддерживающий построение линеек продуктов путем разработки предметно-ориентированных языков и соответствующих компиляторов. Компилятор при этом входит в фонд базовых средств. При фиксации параметров изменчивости продукта (средствами предметно-ориентированного языка) в этот фонд также вносится оперативная библиотека кода, сгенерированная посредством компилятора.
- GM Powertrain, собирая продукты из базовых средств линейки, основывается на контрактах, хранящихся в базе данных. Каждый элемент снабжается четко определенными интерфейсами и возможными изменяемыми параметрами. Проводя в базе данных поиск желаемых характеристик, этот инструмент производит сборку продукта.

Линейки продуктов с широкой областью действия, как правило, разрабатываются по подобию каркасов или коллекций служб. Нижеследующие примеры — тому подтверждение.

- Линейка автомобильных навигационных систем зависит от требований производителей автомобилей, каждый из которых продвигает свои пользовательские интерфейсы и наборы характеристик. Исходя из этого поставщик навигационных систем спроектировал архитектуру как коллекцию каркасов. Соответственно, в ходе разработки любого продукта для него конструируется пользовательский интерфейс, а для заданных характеристик конкретизируются каркасы.
- Система Luther (см. главу 17) представляет собой линейку продуктов, сконструированную поверх J2EE (каркас). В ходе разработки каждого продукта конструируется пользовательский интерфейс и реализуются вспомогательные прикладные модули.

НЕ ВСЕ ТАК ПРОСТО

Парадигма линеек программных продуктов позволяет распределить средства, вложенные в разработку архитектуры (равно как и других базовых средств), на семейство родственных систем и тем самым на порядок сократить время выхода на рынок, повысить качество и продуктивность.

Реальность достижения таких результатов убедительно доказана крупными и мелкими компаниями, работающими в самых разных предметных областях. Результаты действительно стоят усилий. Более того, многочисленные источники (в том числе и в самих компаниях) сходятся в том, что для окупаемости затрат на учреждение линейки продуктов в ней достаточно выпустить около трех продуктов. По нашему мнению, это минимальное число продуктов в любой линейке.

Стоит, впрочем, отметить, что возможны и другие результаты. При определенных условиях попытки внедрить рассматриваемую методику не приводят ни к чему хорошему. Как и любая новая технология, методика построения линеек продуктов при внедрении требует тщательного планирования, учета истории компании, текущей ситуации и культурных стандартов.

К неудаче при попытке внедрения линейки продуктов способны привести следующие факторы:

- отсутствие явно выраженного лидера с достаточными управленческими и контрольными полномочиями;
- неспособность ответственных лиц постоянно и последовательно оказывать разработчикам поддержку;
- нежелание менеджеров среднего звена отказаться от единоличного контроля над проектами;
- нечеткое формулирование коммерческих задач, обусловливающих внедрение методики линеек продуктов;
- отказ от реализации методики при появлении первых же трудностей;
- недостаточно глубокое ознакомление персонала с методикой, неспособность должным образом объяснить или оправдать вносимые изменения.

К счастью, для борьбы с большинством перечисленных проблем существуют специальные стратегии. В частности, полезно проработать небольшой, но характерный пилотный проект, призванный показать количественные преимущества линеек программных продуктов. К работе над проектом следует привлечь тех специалистов, которые выразили готовность апробировать новую методику; скептиков лучше оставить в покое — пусть себе занимаются своим делом. Таким способом можно разобраться в процессе, уяснить роли и обязанности, устранить недоработки и уже после этого расширять масштаб применения методики.

Джо Гэймер (Joe Gahimer) из компании Cummins, Inc. (производителя крайне успешной линейки программных продуктов, рассматриваемой в работе [Clements 02b]) рассказывает о двух задействованных в продуктах компании элементах, владельцы которых были уверены в их неповторимости. По их мнению, схожесть регулятора скорости и регулятора гребного вала сводилась лишь к тому, что оба контролируют скорость. Тщательно зафиксировав детали обоих приложений, участники группы базовых средств в конце концов выяснили, что элементы эти не только схожи, но и функционально идентичны — разница лишь в значениях констант.

При внедрении методики линеек программных продуктов необходимо проявить упорство. На самом деле, оно решает практически все перечисленные выше проблемы. Наиболее эффективно с задачей, как правило, справляется лидер, который (по определению) настойчиво аргументирует преимущества линеек продуктов, преодолевает скептицизм и помогает преодолевать препятствия, возникающие на пути достижения поставленной цели.

— PCC

14.4. Варианты архитектуры линеек продуктов

Из всех элементов репозитария базовых средств наиболее важной является архитектура. Для того чтобы задуманная линейка программных продуктов оправдала возлагаемые на нее надежды, необходимо разделить ее элементы на две группы: 1) те, которые останутся неизменными у всех членов семейства, и 2) те, которые, как предполагается, будут варьировать. Выражается эта двойственность в программной архитектуре, которая по сути своей является абстракцией, допускающей множественность экземпляров; ее концептуальная ценность во многом обусловливается тем, что она позволяет сосредоточиваться на основах проектного решения, распространяющегося на ряд различных реализаций. Архитектура изначально ориентирована на разделение постоянных и изменяемых элементов. В рамках линейки программных продуктов архитектура выражает неизменяемые аспекты.

С другой стороны, архитектура линейки продуктов не ограничивается элементарной дихотомией — она устанавливает набор явно допустимых вариаций. Этим она отличается от традиционной архитектуры, в которой большинство экземпляров зависят от реализации задач (конкретной) системы по качеству и поведению. Таким образом, в обязанности архитектуры входит установление допустимых вариаций и обеспечение встроенных механизмов их реализации. Вариации эти иногда довольно существенны. Программные продукты, участвующие в линейке, существуют параллельно и потенциально различаются по поведению, атрибутам качества, платформе, сети, физической конфигурации, промежуточному программному обеспечению, масштабу и т. д.

Архитектор, работающий над линейкой программных продуктов, должен выполнить три задачи:

- выявить изменяемые параметры;
- обеспечить поддержку изменяемых параметров;
- оценить архитектуру на предмет пригодности к построению линейки продуктов.

Установление изменяемых параметров

Установление параметров изменчивости относится к рутинным операциям. Поскольку вариативность продукта проявляется очень во многих отношениях, возможность выявления точек изменчивости сохраняется на всех этапах процесса разработки. Одни определяются на этапе выявления требований к линейке продуктов, другие — на этапе архитектурного проектирования, третьи — во время реализации. Кроме того, изменяемые параметры иногда выявляются в ходе реализации второго продукта линейки (и всех последующих продуктов).

Среди изменяемых параметров, обнаруживаемых в процессе выявления требований, встречаются характеристики, платформы, пользовательские интерфейсы, атрибуты качества и даже целевые рынки. Некоторые из них взаимозависимы. К примеру, пользовательский интерфейс может быть привязан к предполагаемой платформе, которая, в свою очередь, может зависеть от конкретного целевого рынка.

Изменяемые параметры, обнаруженные в процессе архитектурного проектирования, можно рассматривать либо как альтернативы для реализации вариаций, установленных во время выявления требований, либо как нормальные проектные вариации (дело в том, что отдельные решения откладываются вплоть до поступления дополнительной информации). В любом случае, употребление термина «изменяемый параметр» вполне обоснованно — в архитектуре действительно есть точки, в которой вариации локализуются.

Обеспечение изменяемых параметров

Как правило, механизм реализации экземпляров в традиционных вариантах архитектуры сводится к модифицированию кода. В рамках линеек программных продуктов архитектурные средства обеспечения изменчивости отличаются гораздо большим разнообразием.

- *Включение или пропуск элементов.* Соответствующее решение отражается в процедурах конструирования различных продуктов. Кроме того, компиляцию реализации элемента можно проводить условно — в зависимости от некоего параметра, обозначающего его наличие или отсутствие.
- *Включение разного количества реплицированных элементов.* К примеру, для «мощных» вариантов разумно ввести дополнительные серверы — таким образом, их количество следует оставить неопределенным и обозначить как изменяемый параметр. Точное количество серверов в отношении конкретного продукта в таком случае будет устанавливаться файлом сборки.
- *Отбор версий элементов с единым интерфейсом, но разными характеристиками поведения и атрибутами качества.* Операция эта проводится в периоды компиляции, производства или (в некоторых случаях) прогона. В качестве механизмов отбора выступают статические библиотеки, в которых связывание внешних функций производится после периода компиляции, и динамически подключаемые библиотеки, которые, ничем не отличаясь от статических по критерию гибкости, откладывают принятие решений до

периода прогона, а при его наступлении исходят из контекста и условий исполнения. Сменить реализацию функций с известными именами и сигнатурами можно путем замены таких библиотек.

Посредством перечисленных механизмов на архитектурном уровне проводятся массовые изменения. Допускается введение и других механизмов, направленных на модификацию аспектов конкретных элементов. Под эту категорию, в частности, подпадает механизм редактирования исходного кода. Впрочем, в ней числятся и более изощренные методики.

- В объектно-ориентированных системах изменчивость достигается за счет специализации или обобщения конкретных классов. Учитывать возможность написания (по мере необходимости) специализаций для различных продуктов следует еще на этапе составления классов.
- Встраивание точек расширения в реализацию элемента. Они помогают безболезненно вводить новые варианты поведения и дополнительную функциональность.
- Реализовать изменчивость можно путем введения в элемент, подсистему или в совокупность подсистем параметров периода производства, которые через установку ряда значений позволяют конфигурировать продукт.
- Рефлексией называется способность программы управлять данными о себе, своей среде исполнения и состоянии. Рефлексивные программы могут корректировать собственное поведение в зависимости от контекста.
- Перегрузка способствует повторному использованию именованной функциональности с различными типами. С одной стороны, она увеличивает объем повторно используемого кода, но с другой — усложняет его и снижает понятность.

Само собой разумеется, что документацию (см. главу 9) в рамках линейки продуктов необходимо составлять для всех вариантов архитектуры: для того, что хранится в фонде базовых средств, и для тех, на основе которых выстраиваются конкретные продукты (в последнем случае акцент следует делать на степени изменения по сравнению с архитектурой линейки продуктов). В документации по архитектуре линейки продуктов нужно четко указывать и логически обосновывать (например, определением области действия) все изменяемые параметры. Кроме того, в ней следует описывать процесс конкретизации архитектуры — иначе говоря, расписывать применение изменяемых параметров. Теоретически, каждый параметр изменчивости лучше описывать отдельно, однако на практике далеко не все вариации относятся к числу допустимых. Некоторые из них остаются невостребованными или (что еще хуже) приводят к возникновению ошибок; таким образом, в документации должны быть отражены все варианты связывания вариаций — как правильные, так и неправильные.

Документацию архитектуры отдельного проекта можно описать в терминах приращения или связывания изменяемых параметров — например, указать, что архитектура продукта № 16 предполагает введение *трех* серверов, *шестидесяти четырех* клиентских рабочих станций, *двух* баз данных, *высокоскоростной* версии графического элемента *с низким разрешением* и нулевого шифрования в генераторе сообщений.

Оценка архитектуры линейки продуктов

Архитектуру линейки программных продуктов, равно как и любую другую архитектуру, необходимо оценить на предмет соответствия поставленным задачам. Более того, поскольку от нее зависит сразу несколько систем, оценка архитектуры линейки становится мероприятием еще более значимым, чем обычно.

Облегчает задачу то обстоятельство, что все методики оценки, описанные ранее, в полной мере распространяются на варианты архитектуры линейки продуктов. Архитектуру необходимо проверить на робастность и универсальность — атрибуты качества, без которых она не сможет служить основой систем в предполагаемой области действия линейки продуктов. Кроме того, проверку следует провести на предмет соответствия требованиям по поведению и атрибутам качества, предъявленным к конкретному продукту. Начнем мы с обсуждения содержательной стороны и методологической основы проведения оценки, после чего обратимся к ее временным характеристикам.

Что и как оценивать

Основным предметом оценки должны стать изменяемые параметры — целесообразность их введения, обеспечение ими гибкости, распространяющейся на всю предполагаемую область действия линейки продуктов, оперативность конструирования продуктов и отсутствие неприемлемых издержек производительности в период прогона. Если оценка проводится на основе сценариев, они должны быть ориентированы на конкретизацию архитектуры до отдельных продуктов семейства. Кроме того, вполне возможно, что требования по атрибутам качества в зависимости от конкретных продуктов варьируют, — отсюда необходимость в оценке способности архитектуры организовать все затребованные сочетания атрибутов. Следовательно, выявленные сценарии должны охватывать все требования по атрибутам качества, предъявляемые к членам семейства продуктов.

Довольно часто сведения об аппаратном обеспечении и ряде других факторов, влияющих на заложенную в архитектуре линейки продуктов производительность, в период оценки остаются неизвестными. В таких случаях о диапазоне аппаратных средств и прочих переменных параметрах принимаются некие допущения, на основе которых определяется диапазон реализуемой в архитектуре производительности. Иногда в ходе оценки удается выявить области потенциального состязания, для разрешения которых вводятся специальные политики и стратегии.

Когда приступать к оценке

Объектом оценки должен стать экземпляр или вариация архитектуры, предназначенная для конструирования одного или нескольких продуктов в рамках линейки. Специализация оценки зависит от степени различия вариантов архитектуры продуктов-участников линейки (в том, что касается атрибутов качества) от архитектуры линейки в целом. Если различия незначительные, оценку вариантов архитектуры отдельных продуктов можно сократить, благо многие вопросы будут решены в ходе оценки архитектуры линейки. Архитектура любого продук-

та есть вариация архитектуры линейки продуктов. Аналогичная ситуация складывается и с их оценкой. Таким образом, в зависимости от конкретного метода оценки у ее артефактов (сценариев, контрольных списков и т. д.) будет тот или иной потенциал повторного использования, и в ходе их создания это обстоятельство нужно иметь в виду. Результаты оценки вариантов архитектуры продуктов часто содержат ценную для архитекторов линейки ответную информацию и тем самым стимулируют архитектурные усовершенствования.

Если в линейку продуктов планируется включить новый продукт, выходящий за рамки ее первоначальной области действия (с учетом которой и проводилась оценка архитектуры линейки), для проверки соответствия этого продукта линейке в целом нелишне провести повторную оценку общей архитектуры. Если соответствие установлено, область действия линейки продуктов можно расширить в расчете на включение в нее нового продукта или даже вывести на его основе новую линейку. Если же установить соответствие не удается, оценка поможет определить изменения, которые необходимо внести в архитектуру для приспособления к новому продукту.

Оценка вариантов архитектуры отдельных продуктов и линейки в целом помогает, во-первых, выявить рискованные с архитектурной точки зрения решения и, во-вторых (если проводить оценку по методу CBAM — см. главу 12), определить продукты, сулящие наибольшие выгоды.

14.5. Факторы, усложняющие применение линеек программных продуктов

На создание удачной линейки продуктов компания-разработчик затрачивает довольно много ресурсов, и дело не только в технологическом барьере. Не меньшую важность в контексте получения выгоды от линеек программных продуктов представляют организационные, процессные и коммерческие факторы.

Согласно результатам исследований, проведенных в Институте программной инженерии, существует 29 проблем, или «практических областей» (practice areas), которые обусловливают успешность учреждения компаниями линеек программных продуктов. Многие из них в равной степени применимы при разработке единичных систем, в то же время в контексте линеек продуктов они приобретают новое измерение. Приведем два примера: определение архитектуры и управление конфигурациями.

Операция определения архитектуры важна для любого проекта, однако, как мы установили в предыдущем разделе, в рамках линейки программных продуктов она ориентируется на выявление изменяемых параметров. Управление конфигурациями, также актуальное для всех без исключения проектов, в контексте линеек программных продуктов демонстрирует тенденцию к усложнению — связано это с тем, что каждый продукт – участник линейки являет собой результат связывания многочисленных вариаций. Основная задача, которая ставится перед управлением конфигурациями в линейках продуктов, предусматривает воспроизведение всех версий всех продуктов, поставленных всем заказчикам; под «про-

дуктом» здесь имеется в виду код, а также вспомогательные артефакты в диапазоне от спецификаций требований и контрольных примеров до руководств пользователя и инструкций по инсталляции. Для решения этой задачи необходимо знать версии базовых средств, задействованные при конструировании конкретных продуктов, механизмы их приспособления, а также дополнительный специализированный код и документацию.

Обсуждать все аспекты производства линеек продуктов мы не намерены — в следующем разделе мы рассмотрим лишь несколько значимых областей, демонстрирующих количественные различия между процессами разработки линейки продуктов и единичной системы.

С нижеприведенными проблемами на этапе обсуждения перспективы внедрения линейки программных продуктов сталкиваются все без исключения организации.

Стратегии принятия

Внедрение в компании линеек программных продуктов мало чем отличается от внедрения любых других технологий — налицо одни и те же проблемы. Способ их разрешения зависит от организационной культуры и конкретного контекста.

Нисходящим принятием называется ситуация, в которой решение о введении линеек программных продуктов принимает руководитель компании. Задача в этом случае заключается в том, чтобы на практике заставить сотрудников изменить устоявшиеся привычки. При восходящем принятии проектировщики и разработчики, работающие на уровне продуктов, осознают бессмысленность дублирования результатов деятельности друг друга, переходят к совместному использованию ресурсов и наполняют массив общих базовых средств. Труднее всего в этом случае найти руководителя, готового поспособствовать принятию методики в масштабе всей компании. Оба способа принятия вполне жизнеспособны, и оба существенно облегчаются при наличии сильного *лидера* (champion) — человека, прекрасно разбирающегося в различных вопросах построения линеек продуктов и готового поделиться своими знаниями с другими.

Ортогональным относительно проблемы «как направить распространение технологии» является вопрос о развитии собственно линейки продуктов. Здесь главенствующую роль играют две основные модели[1].

В рамках *активной* (proactive) линейки продуктов разграничение семейства осуществляется путем всестороннего определения области действия. Сотрудники компании принимают соответствующее решение отчасти случайным образом, однако посылками для них служат опыт работы в данной прикладной области, знание рыночной ситуации и технологических тенденций, а также коммерческое чутье. Из двух моделей роста линеек продуктов активная модель предоставляет наиболее широкие возможности — в частности, она позволяет ответственным лицам в компании принимать долгосрочные стратегические решения. Явное определение

[1] Перечисленные ниже модели сформулированы Чарльзом Крюгером (Charles Krueger) в ходе последнего дагштульского семинара по линейкам программных продуктов (www.dagstuhl.de).

области действия линейки продуктов облегчает выявление ниш, недостаточно разработанных присутствующими на рынке продуктами, и за счет незначительно расширения линейки продуктов дает возможность их заполнения. Иначе говоря, активная область действия линейки продуктов позволяет разрабатывающей ее компании принимать независимые решения о своей будущности.

В некоторых случаях возможность прогнозирования потребностей рынка с той степенью определенности, которую предполагает активная модель, отсутствует. Так происходит, если линейка разрабатывает новую предметную область, рынок претерпевает колебания или если компания обнаруживает неспособность единовременно построить фонд базовых средств, охватывающий всю область действия. В таком случае ситуация, как правило, развивается по *реактивной* (reactive) модели. Решение о создании каждого последующего члена семейства продуктов принимается исходя из ассортимента существующих продуктов. С каждым новым продуктом архитектура и решения расширяются по мере необходимости, а фонд базовых средств составляется из тех элементов, которые *оказались* общими, — в противоположность активной модели, в которой общность подлежит предварительному планированию. Вообще, перспективному планированию и вопросам стратегического направления в рамках реактивной модели уделяется не слишком много внимания. Таким образом, компания оказывается в полной зависимости от рыночных тенденций.

Знакомство с различными моделями принятия позволяет ответственным лицам в компании делать осознанный выбор в пользу одной из них. Активная модель предполагает значительные начальные капиталовложения, однако объем дальнейших исправлений существенно уменьшается. Реактивная модель, напротив, основывается на исправлениях и дополнениях, однако практически исключает начальные расходы. Предпочтение той или другой следует отдавать в зависимости от экономической ситуации.

Создание продуктов и развитие линейки продуктов

В активе любой компании, учредившей линейку продуктов, должна быть ее архитектура и прочие связанные с ней элементы. Время от времени компании создают новых членов линейки, обладающих общностью по отдельным характеристикам с остальными членами, но отличающихся от них в других отношениях.

Одна из задач, связанных с линейками продуктов, заключается в контроле над ее развитием. С течением времени линейка продуктов — в особенности набор базовых средств, на основе которых конструируются продукты, — эволюционирует. Движущие силы этой эволюции делятся на внешние и внутренние.

1. *Внешние стимулы.*
   - ◊ Производители элементов, составляющих линейку, регулярно выпускают их новые версии, что, в свою очередь, оказывает влияние на конструкции последующих продуктов.
   - ◊ В линейку продуктов вводятся элементы, разработанные третьими лицами. К примеру, функции, ранее выполнявшиеся элементами внутрен-

ней разработки, через некоторое время могут перейти к элементам, приобретенным у сторонних производителей, и наоборот. Для внедрения в последующие продукты новых технологических разработок их зачастую приходится дополнять внешними элементами, в которых эти разработки реализованы.

- ◊ Поводами для введения в линейку продуктов новых характеристик служат желание соответствовать требованиям пользователей и поведение конкурентного окружения.

2. *Внутренние стимулы.*
   - ◊ Все новые функции, которые вводятся в продукт, следует проверять на предмет соответствия области действия линейки. В случае соответствия их можно конструировать на основе базовых средств. В противном случае необходимо принять одно из двух решений: 1) об отделении модернизированного продукта от существующей линейки продуктов и его самостоятельном развитии; 2) о расширении фонда базовых средств в расчете на новый продукт. Если новая функциональность имеет серьезные шансы быть задействованной в последующих продуктах, есть смысл обновить линейку продуктов; следует, впрочем, иметь в виду, что такое решение сопряжено с необходимостью выделения временных ресурсов на обновление базовых средств.
   - ◊ При внесении в линейку продуктов изменений возникает необходимость в замене устаревших продуктов новыми, построенными на основе новейшей версии фонда базовых средств; это не является непреодолимым препятствием, и отказываться от линейки продуктов в таких случаях не стоит. На поддержание совместимости продуктов с линейкой требуются дополнительные временные и трудовые ресурсы, однако эти вложения оправдываются за счет сокращения продолжительности последующих операций по модернизации. Для того чтобы отражать в новых продуктах свежие функции, введенные в линейку, их необходимо привести в соответствие друг другу.

Организационная структура

Относительно фонда базовых средств, на котором основываются продукты — участники линейки, но который в то же время развивается по собственному сценарию, руководство компании должно принять ряд решений, касающихся управления самим фондом и разработкой продуктов. Изучив организационные модели различных линеек продуктов, Ян Бош (Jan Bosch) в своей работе [Bosch 00b] предложил делить их на четыре типа.

1. *Выделение отдела разработки.* Все процессы, связанные с разработкой программного обеспечения, сосредоточиваются в одном подразделении. Предполагается, что все числящиеся в нем сотрудники обладают недюжинным опытом работы с линейками продуктов и, в зависимости от ситуации, могут переключаться от инженерии предметной области к прикладной

инженерии. Такая модель характерна для небольших компаний, а также организаций, предоставляющих консультационные услуги. Несмотря на удобство и краткость каналов взаимодействия, методика концентрации всех функций в рамках одного подразделения отличается рядом серьезных недостатков. По мнению Боша, действенным этот подход оказывается лишь в тех случаях, когда численность подразделения не превышает 30 человек (с нашей точки зрения, это довольно большая цифра). Таким образом, в мелких компаниях, занимающихся разработкой ограниченных по масштабу линеек продуктов, он весьма полезен.

2. *Выделение нескольких отделов разработки.* Каждый из них ответствен за то или иное подмножество систем в рамках семейства продуктов, выделяемое по признаку общности. Совместно используемые средства создаются теми подразделениями, которые испытывают потребность в их наличии, после чего предоставляются в пользование всем коллегам; не исключается также сотрудничество отделов для разработки новых средств. У этой модели несколько разновидностей, отражающих различные степени гибкости отделов по части разработки (или модификации) совместно используемых средств. В отсутствие каких-либо ограничений продукты обычно расходятся индивидуальными путями, ниспровергая таким образом саму методику линеек продуктов. Ответственность за конкретные средства распределяется между отделами, которые должны поддерживать собственные средства в согласии с линейкой продуктов в целом. Остальные отделы берут на себя обязательства по использованию этих средств. По оценкам Боша, рассматриваемая модель подходит для компаний, в которых штат насчитывает от 30 до 100 сотрудников. К сожалению, во многих подобных случаях отделы концентрируются исключительно на выделенных им продуктах, в результате чего задачи линейки отходят на второй план.
3. *Деятельность в рамках отдела инженерии предметных областей.* За разработку и сопровождение фонда базовых средств в данном случае отвечает специальное подразделение компании; результатами его работы при конструировании продуктов пользуются все остальные отделы. По мнению Боша, когда численность штата компании переваливает за 100 сотрудников, каналы взаимодействия между подразделениями теряют практичность и, таким образом, делают очевидной потребность во введении магистрального канала к фонду совместно используемых средств. Центральную роль в рамках этой модели играет четкий и ясный процесс, направленный, с одной стороны, на координацию обмена информацией, а с другой — на принуждение всех участников к первоочередному соблюдению интересов линейки.
4. *Иерархически выстроенные отделы инженерии предметных областей.* Очень крупные и/или очень сложные линейки продуктов имеет смысл структурировать согласно некоей иерархии. Если, к примеру, некоторые (входящие в линейку продуктов) подгруппы схожи друг с другом в большей степе-

ни, чем с остальными участниками линейки продуктов, имеет смысл организовать два подразделения: одно будет заниматься инженерией предметной области для совместно используемых средств линейки в целом, а другое — средствами, применяемыми в рамках специализированной подгруппы. Несмотря на то что в этом примере мы показали лишь два иерархических уровня, по большому счету, возможности расширения — если, к примеру, в рамках подгрупп выделяются специализированные подгруппы более низкого уровня и т. д. — ничем не ограничены. Иерархически организованные подразделения по предметным областям применимы лишь к самым крупным линейкам продуктов, разрабатываемым в больших компаниях. Их главный недостаток — это тенденция к разрастанию, при котором оперативность реагирования компании на вновь возникающие потребности снижается.

14.6. Заключение

Итак, в настоящей главе мы рассмотрели архитектурную парадигму разработки линеек программных продуктов. По мере того как все больше компаний убеждаются в том, что по части стоимости, времени и качества эта методика предоставляет самые что ни на есть солидные выгоды, она приобретает значительную популярность.

Впрочем, как и любая другая новая технология, методика линеек продуктов готовит для новичков некоторые сюрпризы. С точки зрения архитектуры основной упор делается на выявление общности и вариативности, а также управление ими; в то же время учитываются и некоторые нетехнические проблемы — механизм внедрения модели в компании, ее структура и способ поддержания внешних интерфейсов.

14.7. Дополнительная литература

В работе [Anastasopoulos 00] приводится вполне добротный перечень методик реализации изменчивости. Аналогичные перечни есть в публикациях [Jacobson 97] и [Svahnberg 00].

Комплексное исследование линеек программных продуктов проведено в издании [Clements 02a]. Анализ практических областей линеек продуктов в нем перемежается с конкретными примерами.

Организационные модели рассматриваются в работе [Bosch 00b].

Сведения по процессу FAST мы почерпнули из издания [Weiss 00], а по компании Philips — из [America 00]. Наконец, материалы для примера компании GM Powertrain взяты из труда [Bass 00].

14.8. Дискуссионный вопрос

Предположим, что, имея в своем распоряжении обширный набор общих средств (включая архитектуру), некая компания конструирует две схожие системы. Очевидно, что они составляют линейку продуктов. А было бы правомерно признать их линейкой, если бы общность исчерпывалась архитектурой? Допустим, что у них один общий элемент; что общность распространяется на операционную систему и оперативные библиотеки языка программирования; что за их разработку отвечает одна и та же группа разработчиков. Можно ли в трех указанных случаях включать эти продукты в одну линейку?

Глава 15
CelsiusTech. Конкретный пример разработки линейки продуктов

(в соавторстве с Лайзой Браунсуорд[1])

> Мы неустанно учились, но каждый раз, объединяясь в группы, сталкивались с необходимостью перестройки. Позже, исходя из собственного опыта, я осознал, что на каждую новую ситуацию мы отвечаем реорганизацией; эта система, создавая иллюзию прогресса, на деле приводит лишь к путанице, несостоятельности и падению духа.
>
> *Петроний Арбитр* [Petronius Arbiter], 210

В настоящей главе мы расскажем о наработках компании CelsiusTech AB — разработчика систем для шведских ВМС, сумевшего успешно внедрить методику построения линеек сложных, преимущественно программных, продуктов. Названная Ship System 2000 (SS2000), линейка продуктов этой компании включает в себя судовые системы для военных ведомств государств Скандинавии, Ближнего Востока и южнотихоокеанского региона.

Данный конкретный пример иллюстрирует архитектурно-экономический цикл (Architecture Business Cycle, ABC) в целом и выход CelsiusTech с методикой построения линеек продуктов на новые коммерческие рубежи в частности. Роли лиц, заинтересованных в прохождении архитектурно-экономического цикла, в контексте опыта Celsius изображены на рис. 15.1.

---

[1] Лайза Браунсуорд (Lisa Brownsword) — научный сотрудник Института программной инженерии при Университете Карнеги-Меллон.

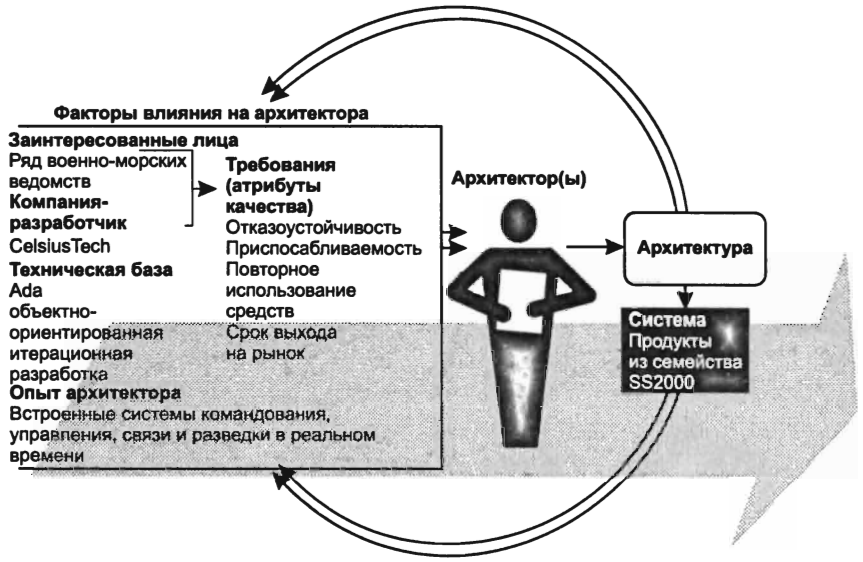


Рис. 15.1. Архитектурно-экономический цикл в CelsiusTech

15.1. Связь с архитектурно-экономическим циклом

Компания CelsiusTech уже довольно давно приобрела статус ведущего шведского поставщика систем командования и управления. Она входит в крупнейшую в Швеции (и одну из крупнейших в Европе) оборонно-промышленную группу, в которой, помимо нее, участвуют Bofors, Kuckums, FFV Aerotech и Telub. В период разработки рассматриваемых в настоящей главе систем в CelsiusTech входили три компании: CelsiusTech Systems (сложные программные системы), CelsiusTech Electronics (электронная техника для оборонной отрасли) и CelsiusTech IT (информационно-технологические системы). Их штат тогда насчитывал около 2000 специалистов, а годовой оборот — 300 миллионов американских долларов. Штаб-квартира компании находится в предместье Стокгольма, а дочерние компании — в Сингапуре, Новой Зеландии и Австралии.

Мы намерены рассмотреть лишь подразделение CelsiusTech Systems (для краткости будем называть его CelsiusTech), занимающееся изготовлением систем командования, управления и связи, систем управления огнем, систем радиолокационного подавления для флота, сухопутных и воздушных войск. Начиная с 1985 года эта организация сменила несколько владельцев и имен (рис. 15.2). Первоначально называвшаяся Philips Elektronikindustrier AB, в 1989 году она была куплена компанией Bofots Electronics AB, а в 1991 реорганизована под новым именем NobelTech AB. Наконец, в 1993 году ее приобрела компания CelsiusTech. В то время как все перечисленные сделки сопровождались сменой топ-менеджеров, бóльшая часть управленцев нижнего и среднего звена, равно как и технические

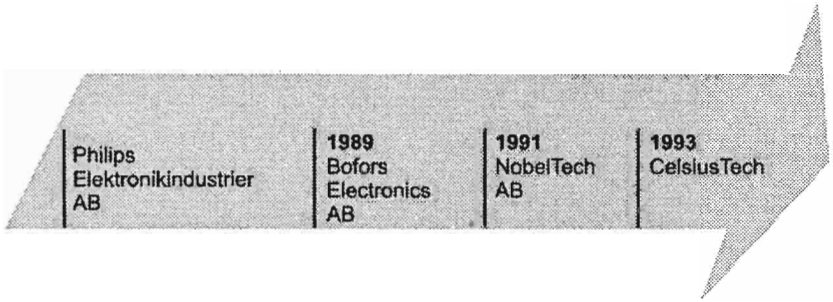


Рис. 15.2. Этапы развития компании CelsiusTech Systems

специалисты, оставались на своих местах, демонстрируя тем самым известную преемственность и стабильность.

Ship System 2000: линейка продуктов для ВМС

Линейка продуктов CelsiusTech для ВМС под названием Ship System 2000 (внутреннее название — Mk3) представляет собой интегрированную систему, объединяющую все устанавливаемые на военных судах системы вооружений, командования, управления и связи. Ее стандартные конфигурации состоят из 1–1,5 миллиона строк кода на языке Ada, распределяемых в локальной сети (local area network, LAN) с 30–70 микропроцессорами.

В рамках одной линейки продуктов сконструированы (и до сих пор конструируются) многочисленные системы для подводного и надводного флота ВМС. Среди них — системы вооружений, командования, управления и связи для следующих боевых единиц:

- шведские корветы береговой обороны (KKV) класса Göteborg (380 тонн);
- датские многоцелевые патрульные суда SF300 (300 тонн);
- финские ракетные катера (FAC) класса Rauma (200 тонн);
- австралийские/новозеландские фрегаты ANZAC (3225 тонн);
- датские океанские патрульные суда класса Thetis (2700 тонн);
- шведские подводные лодки класса Gotland A19 (1330 тонн);
- пакистанские фрегаты класса Type 21;
- оманские патрульные суда;
- датские корветы класса Niels Juel.

Подразделению военно-морских систем удалось продать более 50 своих продуктов в 7 странах.

На рис. 15.3 изображен многоцелевой корвет королевских ВМС Швеции класса Göteborg, зашедший в стокгольмскую гавань. Над ним возвышается антенна диапазона C/X обзорной РЛС обзора и индикации цели. Спереди и сзади от этой антенны, поверх надпалубных сооружений, расположены два разработанных в компании CelsiusTech, полностью укомплектованных устройства, состоящих из РЛС и оптико-электронного блока управления огнем.

Рис. 15.3. Шведский многоцелевой корвет класса Göteborg с системой управления и контроля от CelsiusTech[1]

Системы, конструируемые в рамках рассматриваемой линейки продуктов, сильно различаются по размеру, функциям и вооружению. В частности, в зависимости от страны-заказчика операторские дисплеи строятся на основе разных аппаратных средств и приспосабливаются к выводу информации на разных языках. Не меньше различий между датчиками, системами вооружений и их программ-

[1] Фотография заимствована из фондов Studio FJK; перепечатывается с разрешения правообладателя.

ными интерфейсами. Требования, предъявляемые к подводным лодкам, отличаются от требований, предъявляемых к надводным судам. Среди платформ, применяемых в данной линейке, — 68020, 68040, RS/6000 и DEC Alpha. Что касается операционных систем, то здесь возможны варианты в диапазоне от OS2000 (это собственная разработка CelsiusTech) до IBM AIX, POSIX, Digital Ultrix и некоторых других. Поддержка столь широкого круга систем обеспечивается в линейке продуктов SS2000 посредством единой архитектуры, единого фонда базовых средств и в рамках одной организации.

Экономика линеек продуктов: обзор результатов, достигнутых CelsiusTech

В этом разделе мы обсудим результаты, достигнутые компанией CelsiusTech по части конструирования преимущественно программных систем.

Сокращение графика

На рис. 15.4 приводится состояние и графики разработки позднейших систем из линейки продуктов CelsiusTech. Контракты на разработку систем для судов A и B были подписаны примерно в одно и то же время, что и подвигло CelsiusTech на переход к линейке продуктов. Основой для ее построения послужила система A. Разработка проекта A продлилась почти десять лет — несмотря даже на то, что уже к концу 1989 года на судне были установлены первые функциональные версии системы. Система B — второй из двух оригинальных продуктов, демонстрирующий заметное сходство с предыдущей системой Mk2.5, существовавшей вне линейки продуктов, — разрабатывалась около семи лет. Работа над ней велась параллельно с разработкой системы A, что поспособствовало утверждению новой линейки продуктов. Взятые по отдельности, эти системы не отличались особой продуктивностью, но, несмотря на это обстоятельство, CelsiusTech удалось

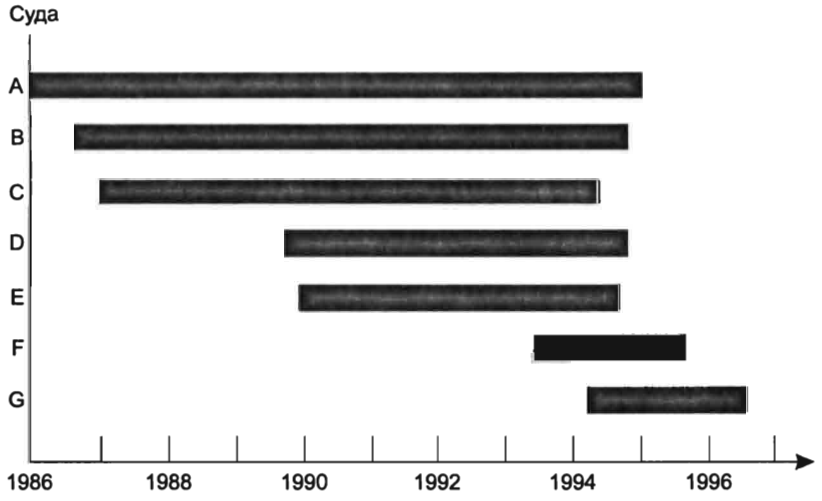


Рис. 15.4. Графики по продуктам

завершить обе (вместе с линейкой продуктов) силами специалистов, которых обычно хватает на один проект.

Когда на горизонте появились системы C и D, значительная часть линейки уже существовала; отсюда — заметное сокращение сроков завершения их разработки. Системы E и F, полностью опиравшиеся на средства линейки продуктов, демонстрируют еще более поразительное ускорение. По словам представителей CelsiusTech, три новейшие системы уверенно укладывались в график.

Повторное использование кода

Производственные графики, напрямую обусловливающие время выхода систем на рынок, ничего не сообщают о том, насколько эффективно в них используется фонд общих средств. Степень общности военно-морских систем CelsiusTech выражает график на рис. 15.5. В среднем, от 70 до 80 % в них занимают «дословные» элементы (взятые из библиотеки управления конфигурациями и задействованные повторно без модификации кода).

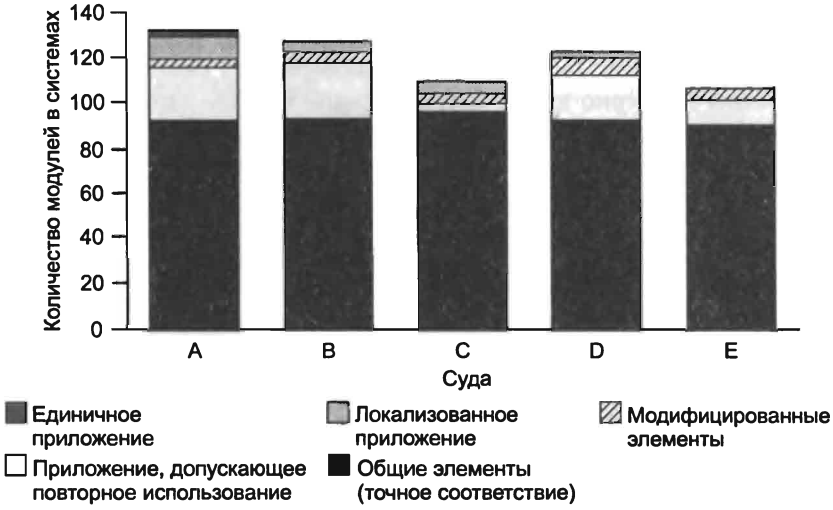


Рис. 15.5. Общность военно-морских систем CelsiusTech

Базовые средства как инструмент расширения области коммерческой деятельности компании

С помощью архитектуры и других базовых средств, изначально разработанных в расчете на военно-морские силы, компании CelsiusTech удалось пробиться на смежный рынок. STRIC, новая система противовоздушной обороны ВВС Швеции, основывается на абстракции, согласно которой наземная орудийная платформа — это судно с нулевыми показателями крена и килевой качки, местоположение которого меняется довольно редко. Благодаря гибкости (возможности внесения изменений) архитектуры и линейки продуктов SS2000, компании удалось сконструировать STRIC в кратчайшие сроки, причем 40 % элементов системы были заимствованы непосредственно из фонда базовых средств SS2000.

(См. врезку «Как в CelsiusTech выводили абстракцию»). Это обстоятельство наглядно демонстрирует один из каналов обратной связи архитектурно-экономического цикла — построение линейки продуктов и архитектуры SS2000 привело к появлению новых коммерческих возможностей.

КАК В CELSIUSTECH ВЫВОДИЛИ АБСТРАКЦИЮ

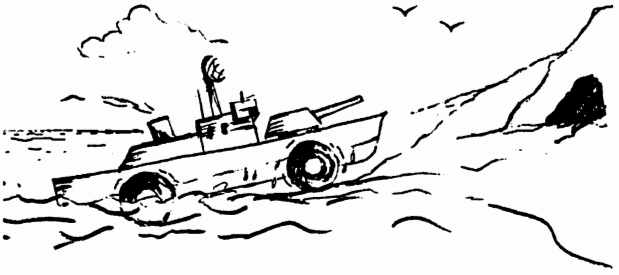
Лично для меня исследования линеек программных продуктов привлекательны по двум причинам. Во-первых, мне не терпится определить, что общего между компаниями, следующими принципу построения линеек продуктов. Когда мы только приступали к анализу линеек продуктов, список тех требований, соответствие которым я считал обязательным для всех успешных компаний, был весьма внушителен. Впоследствии, по мере изучения новых конкретных примеров, я обнаружил, что опыт у всех приверженцев рассматриваемой концепции совершенно разный, в связи с чем мой список требований неуклонно сокращался.

Впрочем, есть одна особенность, которая характерна для всех подобных компаний. Я имею в виду отношение к линейкам продуктов. Преуспевающая компания, построившая линейку продуктов, считает своим долгом заботиться, вскармливать и развивать ее, единственную и неповторимую, в особенности — фонд базовых средств. В этом их основное отличие от незрелых и неудачливых организаций, которые усматривают свою задачу лишь в том, чтобы наплодить кучу безликих продуктов, имеющих между собой некоторое сходство.

Подобные различия не слишком заметны, зато вполне красноречивы. Компания, делающая упор на отдельные продукты, подчиняет долгосрочные задачи по построению линеек сиюминутным задачам по соблюдению графика разработки продуктов. В частности, такие организации поощряют героические усилия своих сотрудников, направленные на закрытие проекта, — даже если они сводятся к банальному клонированию базовых средств в ночное время. Компания, исповедующая принцип построения линеек продуктов, напротив, рассуждает о своей линейке и ее благополучии с таким пиететом, как будто отдельные продукты в ее составе носят случайный, побочный характер. Такое исключительное отношение позволяет организациям, построившим свои линейки, совершать стратегические маневры быстро и без особых усилий.

Второй момент, кажущийся мне довольно занимательным, — это возможности, которые открываются перед выстроившей линейку компанией по части реализации предпринимательской способности. Обладая четким пониманием области действия своей линейки продуктов — иначе говоря, четко разграничив системы, которые можно и, наоборот, нельзя построить ее средствами, — предприятие получает возможность сознательного расширения в направлении смежных, не до конца занятых рынков.

Во время своего визита в CelsiusTech, посвященного сбору сведений для настоящей главы, я заметил на доске одного из разработчиков рисунок, который как нельзя более красноречиво иллюстрирует обе вышеупомянутые особенности. К сожалению, я не догадался его сфотографировать, но, как сейчас помню, картинка была примерно такого содержания:



Приблизительно в то же время, когда мы нанесли визит в CelsiusTech, компания объявила о намерении построить линейку систем противовоздушной обороны — другими словами, программного обеспечения для наземных зенитных установок. По оценке, проведенной

CelsiusTech, на момент объявления 40 % элементов предполагаемого семейства уже были готовы — благо они заложены еще в Ship System 2000.

Выполненный разработчиком рисунок выражает мысль, согласно которой система противовоздушной обороны есть не что иное, как упрощенное судно, базированное на суше, не накреняющееся, не подверженное килевой качке и по большей части находящееся в неподвижном состоянии. Из его содержания я сделал два вывода: во-первых, сотрудники CelsiusTech имеют четкое представление о том, что есть абстракция; во-вторых, у них сформировалось уважительное отношение к линейке продуктов. Суть рисунка не в том, что компании предстоит разработать систему противовоздушной обороны, скорее это предвкушение новой стадии, в которую вскоре обещает войти излюбленная сотрудниками линейка. На нем в лаконичной форме выражено стремление компании выйти на новый рынок, причем сделать это именно с помощью линейки продуктов. Итак, приверженность отдельно взятой компании принципу построения линеек в полной мере доказала свою состоятельность.

— PCC

Чем руководствовалась компания CelsiusTech

Для того чтобы разобраться в факторах, подтолкнувших руководство CelsiusTech к принятию решения о построении линейки продуктов, и действиях, которые нужно было предпринять для достижения этой цели, рассмотрим предысторию. Вплоть до 1986 года компания, специализировавшаяся в предметной области управления огнем, разработала более 100 систем в 25 конфигурациях размером от 30 000 до 700 000 строк исходного кода (SLOC).

В период с 1975 по 1980 год CelsiusTech занималась переводом своих технологических средств с аналоговой на 16-битную цифровую основу, в результате чего появились так называемые системы Mk2. Они оказалась компактными, работали в реальном времени и в основном были встроенными. В процессе конструирования и поставки 15 систем компании удавалось последовательно расширять их функциональность и углублять познания в области приложений реального времени.

С 1980 по 1985 год требования заказчиков изменились — теперь им нужна была интеграция функций управления огнем и вооружениями с функциями командования и управления; следовательно, поставляемые системы увеличились в размере и усложнились. В расширенном варианте архитектуры Mk2, получившем наименование Mk2.5, предусматривались многочисленные автономные узлы обработки, которые должны были размещаться на двухточечных каналах. Системы Mk2.5 оказались значительно более масштабными, причем как по объему поставляемого кода (вплоть до 700 000 SLOC), так и по численности разработчиков (300 инженерных лет требовалось ужать в 7 астрономических).

В рамках Mk2.5 использовались традиционные методики разработки. Для относительно небольших систем Mk2 они еще подходили, однако теперь, с появлением новой архитектуры, возникли трудности, связанные с предсказуемостью и временными рамками интеграции, перерасходом средств и отставанием по срокам. Подобные малоприятные эксперименты многому научили CelsiusTech. Сотрудники компании наработали ценный опыт элементарного распределения процессов реального времени по автономным каналам и применения высокоуровневого языка программирования операций в реальном времени (в данном случае таковым оказался Pascal-подобный RTL/2). На рис. 15.6 приводится характеристика по системам, разработанным CelsiusTech до 1985 года.

	1970–1980: системы Mk2	1980–1985: системы Mk2.5
Вид системы	Встроенная система управления огнем в реальном времени; язык ассемблера и RTL/2	Встроенная система командования, управления и связи в реальном времени; RTL/2
Объем	30–100 тысяч строк исходного кода	700 тысяч строк кода; 300 инженерных лет за 7 астрономических
Платформы	Аналоговые и 16-битные цифровые системы	Многопроцессорные; мини-компьютеры; двухточечные каналы связи

Рис. 15.6. Системы, сконструированные в CelsiusTech до 1985 года

В 1985-м произошло событие, определившее весь дальнейший ход развития компании (принадлежащей тогда концерну Philips). В этом году она одновременно получила два крупных подряда — один от шведских, другой от датских ВМС. После ознакомления с требованиями к двум системам стало ясно, что архитектура Mk2.5, не обеспечивающая соблюдения временных и финансовых ограничений, не годится для реализации столь крупных и сложных проектов. Руководители компании вместе с ее ведущими техническими специалистами приступили к обсуждению вариантов решения задачи — как разработать две сверхкрупные системы, да еще и одновременно? От технологий и методик разработки, применявшихся в рамках систем Mk2.5, очевидно, необходимо было отказаться, поскольку в случае их применения говорить о соблюдении графика, бюджета и реализации требуемой функциональности с какой бы то ни было степенью определенности не представлялось возможным. На это элементарно не хватило бы персонала.

В столь неприятной ситуации специалистам все-таки удалось принять решение о принятии новой бизнес-стратегии, которая, снимая акцент с конкретных продуктов, основывалась на потенциальной *коммерческой* возможности конструирования и сбыта целого ряда, или семейства, родственных систем. Так начиналась линейка продуктов SS2000. Немаловажную роль сыграл и другой коммерческий фактор — технологический ресурс военно-морских систем, равный 20–30 годам. За этот период времени накапливаются новые требования к противодействию угрозам и технологические достижения. Чем более гибкой и расширяемой становится линейка продуктов, тем шире коммерческие возможности. Так, из коммерческих факторов, или требований, выкристаллизовалась техническая стратегия.

Эта техническая стратегия предусматривала создание гибкого и надежного набора стандартных блоков, которые должны были составить содержательную основу линейки продуктов и из которых без особого труда можно было бы собирать системы. По мере формулирования новых требований к системам в линейку

продуктов предполагалось вводить дополнительные стандартные блоки, призванные поддерживать ее «коммерческую жизнеспособность».

В процессе определения технической стратегии была проведена оценка технологической инфраструктуры Mk2.5, выявившая серьезные ограничения. Исходя из ее результатов руководство компании приняло стратегическое решение о создании архитектуры нового поколения (Mk3), предусматривавшей новое аппаратное и программное обеспечение, а также новаторскую методику разработки. Предполагалось, что конструировать системы на ее основе можно будет в течение 10–20 лет.

Все было внове

Решение руководства CelsiusTech о переводе бизнес-стратегии на основу линейки продуктов было принято в период стремительного развития новых технологий. Таким образом, для реализации технической стратегии линейки продуктов SS2000 требовалось внести изменения практически во все аспекты аппаратного, программного и производственного обеспечения. Что касается аппаратной части, в связи с новыми задачами последовал переход с мини-компьютеров VAX/VMS на микрокомпьютеры серии Motorola 68000. Системы Mk2.5 ограничивались небольшим количеством процессоров, размещенных на двухточечных каналах. Продукты из линейки SS2000, напротив, предусматривали наличие многочисленных сверхраспределенных процессоров с требованиями по отказоустойчивости. В связи с новой концепцией жизненного цикла программных продуктов от структурного анализа/проектирования и водопадной разработки на основе RTL/2 компания перешла к языку Ada83, процессы разработки в котором отличаются более заметной ориентацией на объекты и итеративностью. Производственные средства также оказались подвержены изменениям — от создаваемых локально и соответствующим образом сопровождаемых инструментов разработки пришлось отказаться в пользу коммерческой среды разработки. Основные технические различия между двумя инфраструктурами показаны на рис. 15.7.

Анализ коммерческого контекста

В процессе учреждения компанией CelsiusTech линейки продуктов SS2000 важную роль играл ряд факторов, рассматриваемых ниже. Некоторые из них способствовали, другие препятствовали успешному решению поставленной задачи.

Неразбериха в отношениях собственности

Продажа, покупка и реструктуризация компаний — это, с одной стороны, обычное дело. С другой стороны, если компания пытается внедрить кардинально отличающиеся от предыдущих технические и бизнес-стратегии, упомянутые процессы оказывают на нее разрушительное влияние. Смена руководства, сопровождающая смену собственника компании, способна похоронить любые намерения, связанные с переходом на новые технологии или модернизацией (еще два тысячелетия назад это подметил Петроний Арбитр). То обстоятельство, что CelsiusTech счастливо избежала такой участи, можно отнести либо к ответственности и дально-

До 1986 года Старая техническая инфраструктура	1986 год Новая техническая инфраструктура
• Мини-компьютеры	• Микрокомпьютеры
• Ряд процессоров, соединенных двухточечными каналами связи	• Множество процессоров, соединенных коммерческой локальной сетью
• Без отказоустойчивости	• Отказоустойчивость, резервирование
• RTL/2	• Ada83
• Водопадный жизненный цикл, первые опыты в области инкрементной разработки	• Макетирование, итерационная инкрементная разработка
• Структурированный анализ и проектирование	• Анализ предметной области, объектно-ориентированный анализ и проектирование
• Локально разработанные вспомогательные инструментальные средства	• Среда разработки фирмы Rational

Рис. 15.7. Переход к новой технической инфраструктуре

видности руководства, либо к его перегруженности другими вопросами. Поскольку в переходный период CelsiusTech сменяла собственника неоднократно, более правдоподобным нам представляется второе объяснение. Очевидно, что руководители среднего звена, твердо убежденные в необходимости построения линейки продуктов, проводили свою линию, по большому счету, независимо от воли топ-менеджеров, которые, по-видимому, занимались в это время какими-то своими делами; если бы последние обратили внимание на происходящее, они вряд ли утвердили бы столь серьезные стратегические инвестиции. Как правило, реорганизация нарушает работу компании на всех уровнях. В данном случае менеджерам среднего звена удалось затормозить воздействие преобразований, явившихся естественным следствием смены собственника.

Потребность

Получение в 1986 году от военно-морских ведомств сразу двух крупных подрядов — казалось бы, повод для торжества — обернулось для CelsiusTech кризисом. Руководство признало, что имеющиеся у компании технические средства и трудовые ресурсы недостаточны для одновременной работы над двумя масштабными проектами, в которых к тому же требовалось реализовать новые технологии и прикладные области. Поскольку все контракты, заключавшиеся CelsiusTech, фиксировали стоимость работ, невозможность довести проекты до конца приравнивалась к вселенской катастрофе. И действительно, даже менее сложные системы

выходили за рамки бюджета и графика, трудно поддавались интеграции и совершенно не оставляли возможности прогнозирования.

К решению о построении линейки продуктов компания CelsiusTech, таким образом, пришла под давлением обстоятельств; осознавая, что на кон поставлено само ее существование, руководство было вынуждено рисковать. Поскольку в тот же период развитие новых технологий ускорилось, отделить издержки, связанные с построением линейки продуктов, от расходов на внедрение этих самых технологий оказалось очень трудно.

Развитие технологии

Все выбранные в 1986 году для разработки проектов технологии отличали незрелость и ограниченная применимость в промышленных масштабах. Все говорили о крупных, работающих в реальном времени распределенных системах, опирающихся на задачи и настраиваемые средствами языка Ada; на самом же деле такие системы только предстояло создать. Более того, объектно-ориентированная разработка в рамках языка Ada находилась на этапе теоретических прений. В то же время, в период с 1986 по 1989 год, сотрудники CelsiusTech стали свидетелями нижеследующих явлений, к которым нужно было еще приспособиться.

- Развитие технологий — в частности, Ada и объектной технологии.
- Развитие вспомогательной технологии — в том числе сетей и распределения.
- Развитие инфраструктуры — сред и инструментальных средств разработки, облегчающих процессы автоматизации разработки.
- Эффект обучения технических сотрудников и управленческого состава компании в области применения новых технологий и процессов построения линеек продуктов.
- Эффект обучения заказчиков, которые поначалу весьма слабо представляли себе договорные, технические и коммерческие аспекты построения линеек продуктов.
- Выработка общих требований представителями разных заказчиков.

Все эти факторы существенно затянули сроки построения линейки продуктов. Сегодня любая компания, решившаяся на аналогичное смещение парадигмы, находится в значительно более выгодном положении — ведь в ее распоряжении микрокомпьютеры, сети, переносимые операционные системы, стандарты открытых систем, объектно-ориентированные методы разработки, Ada (или другие языки программирования — в зависимости от конкретной платформы и предметной области), расчет производительности, технология распределенных систем, работающие в реальном времени операционные системы и аналитический инструментарий, среды разработки для крупных проектов и соответствующие консультанты. Все эти технологии либо уже сформировались, либо, по меньшей мере, готовы к употреблению. Кроме того, сегодня значительно более обширны знания в области конструирования и построения линеек программных продуктов (см. главу 14). По оценкам CelsiusTech, до трети начальных «технологических» капиталовложений компания освоила за счет конструирования средств, которые в настоящее время имеются на открытом рынке.

Организационная структура CelsiusTech

На протяжении десятилетнего периода деятельности CelsiusTech, на который мы обращаем основное внимание, организационная структура компании и принципы ее работы претерпели некоторые изменения — можно даже сказать, прошли несколько этапов. Соответственно изменились предъявляемые к персоналу требования, касающиеся знаний и навыков.

Организация работы над проектом до 1986 года

Во главе процесса разработки военно-морской системы командования и управления (Mk2.5) стоял руководитель проекта. Его задача состояла в том, чтобы перенести обязанности отдельных функциональных областей — например, вооружения или С3 (командования, управления и связи) — на крупные сегменты системных средств. Организационная структура проекта Mk2.5 изображена на рис. 15.8. Каждую функциональную область (командование и управление, сопровождение цели и т. д.) возглавлял руководитель проекта, располагавший определенным кругом сотрудников в прямом подчинении и ответственный за все операции разработки системы вплоть до выпуска и интеграции.

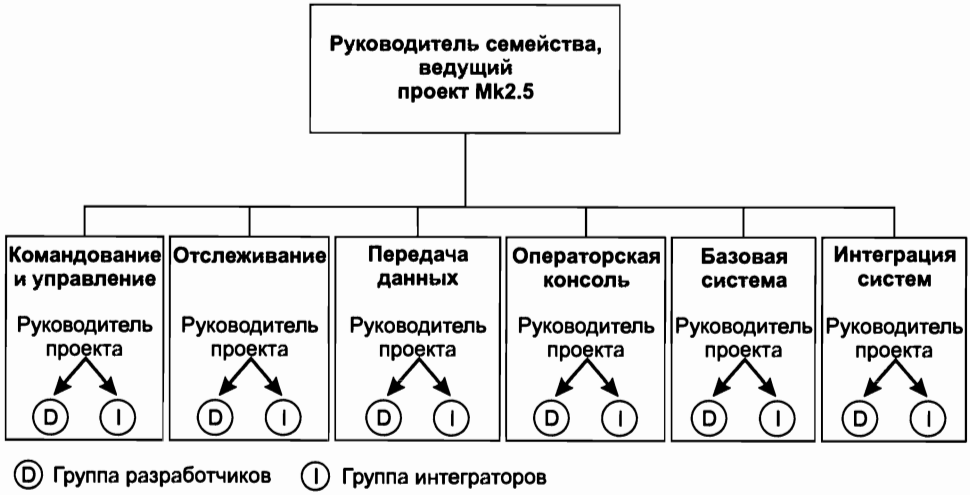


Рис. 15.8. Организационная структура проекта Mk2.5, 1980–1985

По наблюдениям CelsiusTech, столь дробная структура сформировала оригинальный режим разработки, обладавший следующими характеристиками.

- В процессе системного анализа устанавливалось соответствие основных сегментов системы и функциональных областей.
- Так как требования и интерфейсы распределялись и описывались документально, а взаимодействие между отдельными функциональными областями было ограниченно, в ходе проектирования, реализации и тестирования формировалось несколько вариантов интерпретации требований и интерфейсов.

- Факты несовместимости интерфейсов, как правило, удавалось обнаружить лишь на этапе интеграции системы; отсюда — временные затраты на распределение обязанностей, затягивание и усложнение процессов интеграции и установки.
- Знания руководителей функциональных областей в основном ограничивались сферой их обязанностей.
- Руководители функциональных областей не слишком стремились коллективно решать проблемы, возникавшие на программном уровне.

Рис. 15.9. Организационная структура линейки продуктов SS2000, 1987–1991

Организационная структура проекта SS2000 с конца 1986 по 1991 год

С появлением в конце 1986 года линейки продуктов SS2000 организационная структура утратила ряд характеристик, унаследованных от периода проекта Mk2.5. Структура CelsiusTech, просуществовавшая с конца 1986 по 1991 год, изображена на рис. 15.9. Теперь обязанности по созданию линейки продуктов и поставке на

ее основе клиентских систем перешли к генеральному руководителю программ. Осознавая недостатки имевшего место в прошлом дробления структуры, руководство CelsiusTech организовало сильную команду управленцев, которые понимали, что развитие линейки продуктов есть пополнение активов компании, ни в коем случае не сводящееся к построению громоздкой конструкции. В соответствии с этой новой концепцией все руководители проектов были подчинены генеральному руководителю. Разработчики распределялись согласно различным функциональным областям (например, вооружениям и С3), а также направлялись на создание человеко-машинного интерфейса (human-computer interface, HCI), общих служб (применяемых в рамках функциональных областей) и интерфейсов с различными аппаратными и операционными системами (которые обобщенно назывались Base System).

Далее, была сформирована компактная, высокопрофессиональная группа архитекторов, получившая комплексные полномочия и абсолютный контроль над разработкой; отчитывались ее участники напрямую перед генеральным руководителем программ. Руководство CelsiusTech пришло к выводу о том, что успешность линейки продуктов обусловливается наличием стабильной и гибкой архитектуры, известной всем в компании и наделенной серьезными полномочиями (эти полномочия делегировались высшими руководителями компании). Таким образом, компания затеяла реорганизацию, учитывающую преимущества ABC — архитектура становилась центром новой методики и в то же время начинала оказывать воздействие на организационную структуру самой компании.

В качестве основной задачи при создании линейки продуктов было задано скоординированное определение многочисленных выпусков и управление ими. Желая усилить управление выпусками, CelsiusTech объединила обязанности по интеграции программных систем и управлению конфигурациями в рамках новой группы, непосредственно подчинявшейся генеральному руководителю программ. Как группа архитекторов, так и группа управления интеграцией и конфигурациями вводились в CelsiusTech впервые; как оказалось впоследствии, они сильно поспособствовали созданию линейки продуктов SS2000.

Обязанности группы архитекторов распространялись на первоначальную разработку архитектуры линейки продуктов, а также на последующее владение этой архитектурой и контроль за ней. Тем самым в масштабах всех функциональных областей обеспечивалась непротиворечивость проектного решения и согласованность его интерпретации. В частности, в обязанности группы архитекторов входили следующие операции:

- Выработка понятий и принципов линейки продуктов.
- Выявление уровней и их экспортных интерфейсов.
- Описание интерфейсов, обеспечение их целостности и управляемости развития.
- Распределение функций системы между уровнями.
- Установление общих механизмов или служб.
- Определение, макетирование и координация общих механизмов наподобие обработки ошибок и протоколов межпроцессного взаимодействия.

- Консультирование сотрудников проекта относительно понятий и принципов линейки продуктов.

Первую итерацию архитектуры удалось создать за две недели силами двух старших инженеров с богатым опытом работы в данной предметной области. Оставаясь скелетом существующей линейки продуктов до настоящего времени, она формулирует основополагающие понятия, определяет уровни, идентифицирует примерно 125 системных функций (из 200 существующих на сегодняшний день), относит их к конкретным уровням, устанавливает принципиальные механизмы распределения и взаимодействия. По завершении первой итерации в группу архитекторов были привлечены главные проектировщики из всех функциональных областей. Эта обширная команда, состоявшая теперь из десяти старших инженеров, приступила к расширению и уточнению архитектуры. Новая система, таким образом, кардинально отличалась от старой, в которой руководители функциональных областей независимо друг от друга вырабатывали проектные решения и интерфейсы.

Обязанности объединенной группы управления интеграцией и конфигурациями сводились к следующему:

- Разработка стратегий и планов тестирования, а также контрольных примеров, которые не должны ограничиваться тестированием отдельных блоков.
- Координация всех тестовых прогонов.
- Разработка пошаговых графиков производства (совместно с группой архитекторов).
- Интеграция и выпуск правильных подсистем.
- Управление конфигурациями библиотек разработки и библиотек выпусков.
- Создание носителя для поставки программного обеспечения.

Организационная структура SS2000 с 1992 по 1998 год

В период с 1992 по 1994 год CelsiusTech сместила акцент с *разработки* архитектуры и элементов линейки продуктов в пользу компоновки на ее основе новых клиентских систем. Эта тенденция поспособствовала количественному усилению и расширению обязанностей группы управления клиентскими проектами. Корректировка организационной структуры CelsiusTech привела к распределению штата между двумя категориями проектов.

- *Компонентные проекты, направленные на разработку, интеграцию и координацию элементов линейки продуктов.* Производственные мощности распределялись между областями компонентных проектов, которые в свою очередь делились на функциональные области (вооружения, СЗ и HCI), общие службы, а также элементы операционной системы и сети. Регулярная ротация руководителей компонентных проектов способствовала расширению знаний о линейке продуктов среди менеджеров среднего звена. Производившиеся элементы предоставлялись для нужд клиентских проектов.
- *Клиентские проекты, направленные на решение всех финансовых вопросов, составление графиков и планирование, а также проведение анализа требо-*

ваний посредством интеграции/тестирования/поставки систем. Для каждой клиентской системы, конструируемой на основе средств линейки продуктов, назначался руководитель проекта, на которого возлагались обязанности по взаимодействию и ведению переговоров с заказчиком.

Когда работа над базовой частью линейки продуктов была завершена, а сотрудники CelsiusTech получили определенный опыт ее применения, встал вопрос о повышении эффективности производства систем и развития линейки с учетом новых технологий и динамики потребностей заказчиков. Так проявился эффект обратного воздействия ABC, побуждающий компанию к постоянному самообновлению; организационная структура в результате получила очертания, зафиксированные на рис. 15.10.

Рис. 15.10. Организационная структура SS2000, 1992–1998

Каждая крупная прикладная область (конкретнее, военно-морские системы и системы противовоздушной обороны) была выделена в отдельное подразделение с собственным руководителем. В состав обоих этих подразделений входили маркетинговая группа, группа планирования, группа клиентских проектов и группа описания систем. В сферу обязанностей каждого подразделения входили сопровождение соответствующих программных элементов и руководство клиентскими проектами. Все операции в рамках подразделения были прописаны в маркетинговом (Marketing Plan), производственном (Product Plan) и архитектурно-техническом (Technical–Architecture Plan) планах. Маркетинговая группа составляла маркетинговый план с анализом возможностей и ценности отдельных сегментов рынка. План производства, обязанности по составлению которого отводились группе планирования, устанавливал перечень продуктов, поставляемых на рынок данным подразделением. План производства, таким образом, являлся средством

реализации маркетингового плана. Сотрудники группы описания систем обязаны были составлять для своего подразделения архитектурно-технические паны. Архитектурно-технический план, в котором определялось направление развития архитектуры данного подразделения, в свою очередь, реализовывал план производства. При выдвижении новых предложений по проектам требовалось учитывать положения плана производства и архитектурно-технического плана. Таким образом, проекты удерживались в рамках линейки продуктов.

Поставщиком модулей выступила группа разработчиков (Development Group). Любая подгонка и разработка, ориентированная на конкретного заказчика, проводилась в рамках клиентского проекта подразделения силами группы разработчиков. За архитектуру отвечала группа описания систем подразделения. На правах принадлежности она контролировала развитие архитектуры, основные интерфейсы и механизмы. В отделе разработки систем для военно-морских сил (Naval Business Unit) группа описания систем была компактной (как правило, в нее входили шесть человек) и комплектовалась старшими инженерами, досконально изучившими свою линейку продуктов. В сферу ее ответственности входило общее разрешение конфликтов между требованиями заказчика и путями их воздействия на линейку.

Отдел разработки систем для военно-морских сил организовал пользовательскую группу по линейке продуктов SS2000, в рамках которой заказчики могли делиться опытом работы с линейкой и предлагать направление ее дальнейшего развития. В группе были представлены все без исключения заказчики SS2000.

Группа разработчиков предоставляла свои ресурсы всем подразделением. Интеграция, управление конфигурациями и контроль качества — все эти процессы входили в ее компетенцию и по мере необходимости выполнялись по заказу подразделений. В целях дальнейшей оптимизации создания на основе линейки продуктов новых систем была сформирована «базовая конфигурация SS2000» (Basic SS2000 Configuration Project) — фундаментальная, заранее интегрированная конфигурация, насчитывающая около 500 000 SLOC, включающая полный комплект документации и контрольных примеров; она должна была стать ядром всех последующих клиентских систем.

Группа технического управления (Technical Steering Group, TSG) была ответственна за выявление, оценку и испытание новых технологий, способных усилить присутствие CelsiusTech в освоенных ею областях коммерческой деятельности. Во главе ее стоял вице-президент по технологии, в подчинении которого находились ведущие технические специалисты из отделов разработки систем для военно-морских сил и систем противовоздушной обороны, группы разработчиков, а также группы исследований и разработки. Группа TSG контролировала создание и развитие архитектурно-технических планов всеми группами описания систем.

Персонал в 1986–1991-х годах

Как видно из содержания рис. 15.11, уровень кадрового обеспечения проектов за десять лет фиксируется в диапазоне от первоначального показателя в 20–30 человек до максимального в 200; средний уровень равен 150. На ранних стадиях развития программы, в процессе учреждения понятий и архитектуры линейки

продуктов, уровень кадрового обеспечения CelsiusTech был непомерно высок. Разработчики плохо понимали происходящее, поскольку понятия и методики постоянно подвергались корректировке.

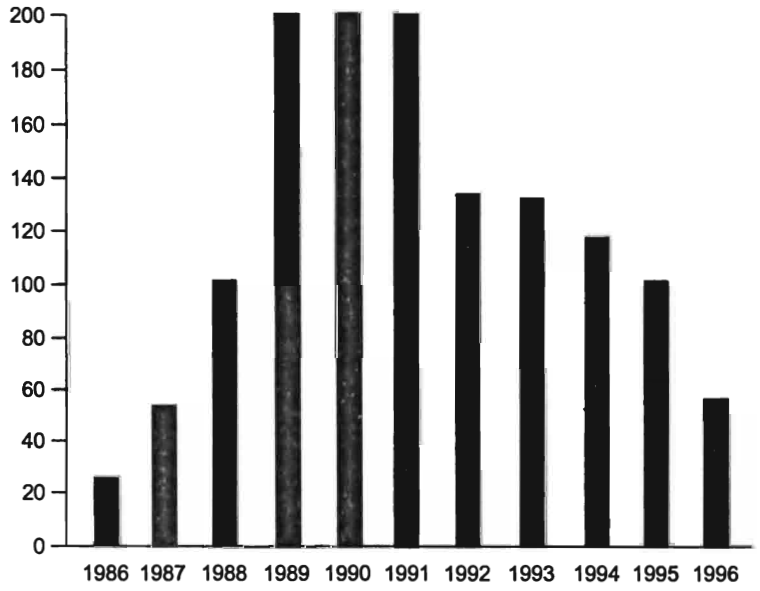


Рис. 15.11. Примерные профили кадрового обеспечения программ

На группу архитекторов ложилась ответственность за создание каркаса линейки продуктов. От ее участников требовалось глубокое знание предметной области и соответствующего рынка, инженерное мастерство и способность выявлять значимые общие механизмы и элементы линейки продуктов. Не менее важны были коммуникативные навыки и умение работать в команде. Разработчики должны были ориентироваться в каркасе и проектных нормах, увязывая с этими знаниями процесс конструирования соответствующих модулей. В продолжение периода формирования линейки продуктов от разработчиков требовалось знание языка Ada, объектного проектирования и среды разработки программных средств, в том числе — целевого испытательного стенда. Кроме того, они должны были хорошо разбираться в понятиях линейки продуктов, архитектуре и механизмах SS2000, создании повторно используемых модулей, методике пошаговой интеграции и по меньшей мере одной предметно-функциональной области.

Принимая во внимание неразвитость основных технологий, руководящей группе (равно как и ведущим техническим специалистам) приходилось опираться на собственную веру в достижение общими силами поставленной задачи. В основном они занимались тем, что разъясняли подчиненным коммерческую потребность внедрения новой методики и предполагаемый результат.

Все компании, предпринимающие попытки внедрить незрелые технологии, по мере появления неизбежных трудностей встречают сопротивление. Подавить такие настроения на ранних этапах способно только убедительное руководство, ориентированное на поиск новых решений. Генеральный руководитель программ

CelsiusTech смог отказаться от критики незрелых технологий, их поставщиков и собственных подчиненных и сконцентрировал свои усилия на проработке решений. Он стал поощрять эксперименты (при условии их подконтрольности) и технические инновации. Таким образом, он превратился в образец для подражания со стороны остальных менеджеров.

В период формирования линейки продуктов от менеджеров требовались серьезные знания связанных с ней понятий в сочетании с деловой мотивацией. Кроме того, они должны были разбираться в вопросах планирования, обладать коммуникативными навыками и уметь решать инновационные задачи.

Также менеджерам приходилось подавлять недовольство и сопротивление — явления, неизбежно сопровождающие переход к новой коммерческой парадигме и технологии. На то, чтобы уяснить сотрудникам суть новой бизнес-стратегии и изложить ее логическое обоснование, ушло довольно много сил. Те люди, которых эти разъяснения не убедили, а также те, кто их просто не понял, либо ушли из компании, либо были переведены на другие проекты и на сопровождение. В результате произошла существенная утечка знаний предметной области, для устранения которой потребовалось некоторое время.

Персонал в 1992–1998-х годах

К концу 1991 года полным ходом шла работа над четырьмя клиентскими системами. Значительное число созданных повторно используемых модулей уже нашли применение в двух первоначальных системах. Ядро линейки продуктов демонстрировало быстрое развитие, и теперь практика компоновки новых систем из существующих модулей была вполне распространена. По мере сокращения потребности в труде проектировщиков они переводились на другие выполнявшиеся компанией проекты. С другой стороны, прирост клиентских проектов, прорабатывавшихся одновременно, привел к повышению потребности в сборщиках — на каждую клиентскую систему их стабильно привлекалось от трех до пяти. Благодаря тому что в рассматриваемый период компания стала получать больше подрядов, численность руководящего состава не сократилась.

В период с 1994 по 1998 год профиль кадрового обеспечения претерпел значительные изменения. Благодаря развитию линейки продуктов и навыков ее применения к работе над двумя последними в этот период клиентскими системами компании удалось привлечь меньше, чем обычно, проектировщиков, разработчиков и сборщиков. Потребность в проектировщиках неизменно продолжала падать, что приводило к переходу этих специалистов из одного подразделения в другое. В наибольшей степени эта тенденция затронула сборщиков — что характерно, бюджеты CelsiusTech предусматривали участие в разработке каждой системы не более одного-двух специалистов этой направленности. Постоянные попытки оптимизировать композицию систем — в частности, проект «базовая конфигурация SS2000» — были направлены на дальнейшее снижение уровня кадрового обеспечения разработки. С другой стороны, учитывая неизменное распространение практики одновременного проведения нескольких клиентских проектов, которых, кстати, становилось все больше, уровень руководящего состава, как и раньше, оставался стабильным.

По мере того как основное внимание перемещалось в сторону *композиции* систем на основе элементов линейки продуктов, предъявляемые к разработчикам требования относительно знания предметной области и SS2000 значительно ужесточились (по сравнению с периодом формирования линейки продуктов). Применение языка Ada, объектной технологии и сред разработки — все эти вещи стали нормой. Группа управления интеграцией сосредоточилась на вопросах интеграции и управления выпусками многочисленных параллельных систем. Все более сильный акцент ставился на повторное использование планов тестирования и наборов данных во всех клиентских системах.

Группе архитекторов приходилось расширять знания о линейке продуктов, а также выполнять посредническую функцию в процессе формулирования текущих и прогнозируемых требований заказчиков. Предельную значимость сохранили задачи взаимодействия с руководителями клиентских проектов (в целях согласования многочисленных потребностей заказчиков) и разработчиками (желающими оптимизировать основные интерфейсы и механизмы). Сотрудники этой группы, постоянно развивавшие архитектуру, ее базовые интерфейсы и механизмы, должны были искусно уравновешивать задачи удовлетворения новых потребностей и сохранения общей архитектурной целостности. Они привлекались к технической оценке, разработке макетов новых интерфейсов (как для внешнего применения, так и для разработчиков приложений) и анализу воздействия на линейку продуктов новых технологий.

По мере прироста элементов линейки продуктов руководящий состав уделял развитию технологии и обучению все меньше внимания. Благодаря повышению числа подрядов, основным приоритетом для менеджмента стала координация меняющихся требований различных заказчиков. В процессе согласования требований участвовали заказчики, менеджеры разных клиентских проектов и участники группы архитекторов линейки продуктов. Руководители клиентских проектов были вынуждены совершенствовать навыки ведения переговоров и расширять знания текущих и предполагаемых в будущем тенденций в рамках линейки продуктов.

15.2. Требования и атрибуты качества

Для того чтобы из репозитария компании можно было вывести новые продукты, их структура должна быть ориентирована на совместное использование модулей. На материале главы 14 вы убедились в том, что такая структура предполагает наличие стандартного набора модулей и соглашений по поводу обязанностей, поведения, производительности, интерфейсов, локальности функционирования, механизмов передачи данных и координации, а также других свойств каждого из этих модулей. Эта структура, распространяющаяся на всех членов семейства, содержащиеся в ней модули, а также их свойства, общие для всей линейки, составляют *архитектуру линейки продуктов* (product line architecture).

На протяжении всей книги мы проводим мысль о том, что основной задачей архитектуры является получение системы, отвечающей требованиям к поведению и атрибутам качества. Архитектура линейки продуктов SS2000, охватывающая

всех ее участников, — не исключение. Ниже перечислены наиболее важные из предъявленных к системам этой линейки требований.

- *Производительность.* Системы командования и управления должны в реальном времени реагировать на постоянно поступающие входные показания датчиков и контролировать вооружения в предельно сжатые строки.
- *Модифицируемость.* Архитектура должна демонстрировать робастность во всем, что касается вычислительных платформ, операционных систем, введения или замены систем считывания и управления вооружением, требований к человеко-машинным интерфейсам, протоколов передачи данных и многого другого.
- *Безопасность, надежность и готовность.* Система должна находиться в состоянии постоянной готовности, предоставлять системам управления вооружениям правильные данные и команды и открывать огонь только в определенных условиях.
- *Контролепригодность.* Каждая система должна предусматривать возможность интеграции и тестирования, обеспечивающего оперативное обнаружение, локализацию и исправление появляющихся ошибок.

Помимо перечисленных требований, распространяющихся на отдельные системы, архитектура SS2000 обнаруживала применимость к целому классу систем. Таким образом, согласно одному из предъявлявшихся к ней требований, необходима была возможность замены каждого конкретного модуля на другой, подогнанный к данной системе, без нарушения архитектуры в целом.

Операционная среда и физическая архитектура

Требования, предъявляемые к современным судовым системам, оказывают сильное влияние на проектные решения. Системы считывания и управления вооружением устанавливаются на всех узлах судна; члены экипажа работают с ними посредством многочисленных раздельно размещенных рабочих станций. Человеко-машинный интерфейс (HCI) должен быть ориентирован, в первую очередь, на ускорение информационных потоков и исполнения команд; кроме того, он в обязательном порядке приспосабливается к конкретному боевому заданию судна и культурным особенностям членов экипажа. Вероятность отказов компонентов обусловливает отказоустойчивость проектных решений.

На рис. 15.12 изображено физическое представление типичной системы. В качестве магистрали передачи данных в ней выступает резервируемая локальная сеть, объединяющая от 30 до 70 разнородных взаимодействующих процессоров. Максимальное количество узлов такой сети достигает 30. *Узлом* (node) называется полюс потока передачи данных, который может принимать форму рабочей станции экипажа, орудийной платформы, считывающего блока; все эти устройства рассредоточены на различных узлах судна. На любом узле может быть установлено до шести процессоров. Локальная сеть организуется по стандарту «дублированная Ethernet». Модули интерфейсов устройств получают и отправляют

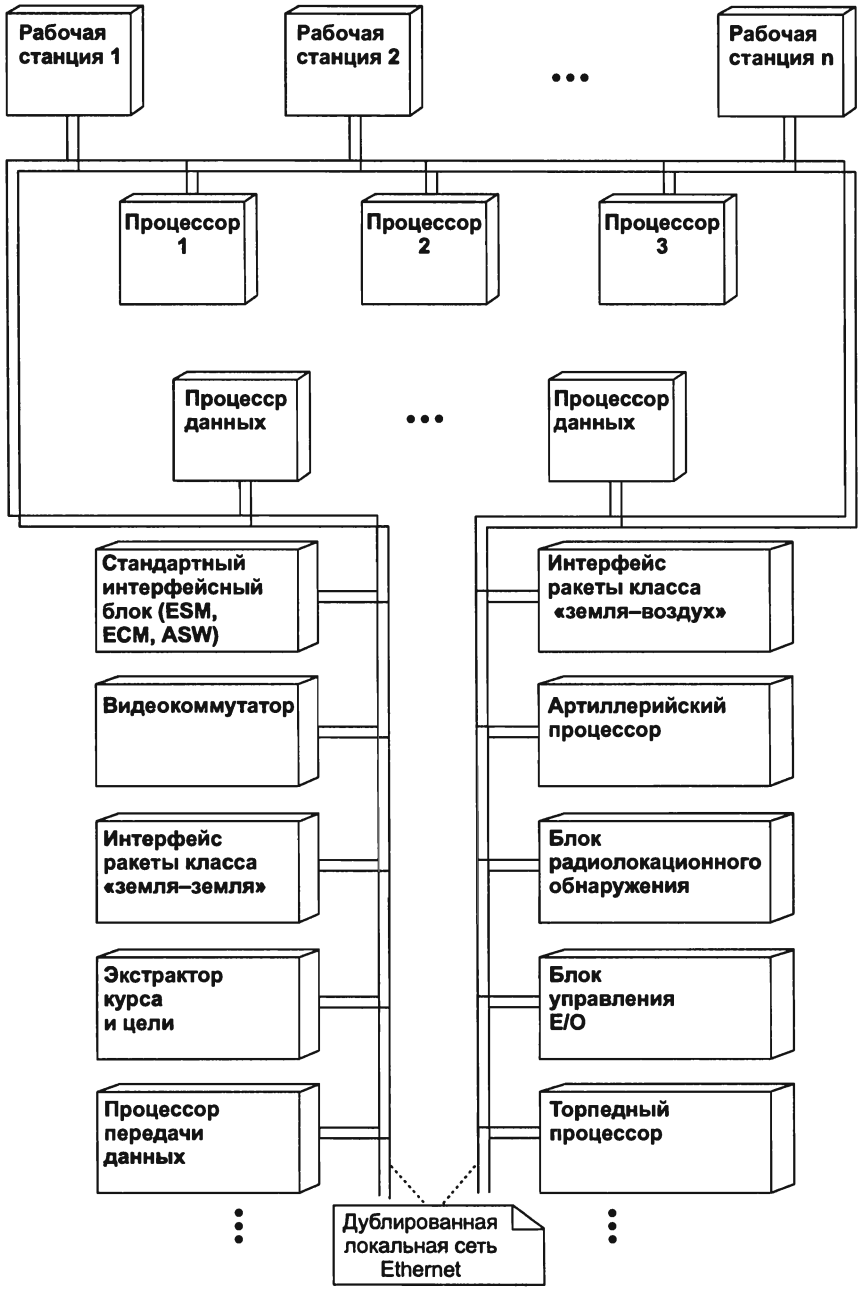


Рис. 15.12. Типичная физическая архитектура продукта из линейки SS2000

данные периферийным устройствам (в основном счетчикам) и управляемым системам вооружений.

15.3. Архитектурное решение

Данную архитектуру мы считаем нужным описать в трех представлениях. Представление процессов разъясняет реализацию распределения; многоуровневое представление помогает понять механизм разделения задач в Ship System 2000; наконец, представление декомпозиции на модули демонстрирует распределение обязанностей между несколькими крупными элементами системы: *системными функциями* (system functions) и *группами системных функций* (system function groups). Охарактеризовав архитектуру в категориях этих трех представлений, мы обсудим ряд проблем сопровождения и применения линейки продуктов, с которыми столкнулась компания CelsiusTech.

Представление процессов: удовлетворение требований по распределению и средства расширения линейки продуктов

На каждом процессоре исполняется набор Ada-программ; с другой стороны, Ada-программы исполняются в основном на одном процессоре. Каждая программа может состоять из нескольких Ada-задач. Системы в линейке продуктов SS2000 насчитывают до 300 Ada-программ.

Требование, касающееся исполнения на распределенной вычислительной платформе, оказывает на программную архитектуру серьезное влияние. Во-первых, оно подразумевает конструирование системы как набора взаимодействующих процессов и тем самым вводит в действие представление процессов. Само наличие представления процессов свидетельствует о применении тактики реализации производительности «введение параллелизма». Кроме того, в распределенных системах возникают задачи, связанные с предотвращением взаимоблокировки, протоколами передачи данных, отказоустойчивостью (на случай отказа процессора или канала связи), сетевым управлением, предотвращением насыщения и производительностью (обеспечивающей координацию задач). Для обеспечения распределенности применяется ряд соглашений. Они отвечают требованиям по распределенности архитектуры — в частности, тем ее аспектам, которые связаны с линейками продуктов. Среди задач и межкомпонентных соглашений нужно отметить следующие.

- Компоненты взаимодействуют путем передачи строго типизированных сообщений. Абстрактный тип данных и управляющие программы предоставляются компонентом, передающим сообщение. Строгая типизация позволяет устранять целые классы ошибок на этапе компиляции. Сообщение как основной механизм взаимодействия между компонентами обеспечивает возможность их написания вне зависимости от деталей (изменяемой) реализации, касающихся представления данных.

- В качестве механизма межпроцессного взаимодействия выступает протокол транспортировки данных между Ada-приложениями; он обеспечивает независимость от местоположения, а значит, и возможность передачи данных между приложениями на любых процессорах. Такая «анонимность распределения процессоров» позволяет переносить процессы с одного процессора на другой, проводить предпрогонную регулировку производительности и реконфигурацию в период прогона (обе эти операции относятся к средствам обеспечения отказоустойчивости) без внесения изменений в исходный код.
- Средства назначения Ada-задач участвуют в реализации модели поточной обработки.

Выполняя свои функции, производитель данных не знает, кто окажется их потребителем. Содержание и обновление данных концептуально отделены от их использования. Это наглядный пример применения тактики реализации модифицируемости под названием «введение посредника» посредством образца классной доски. Основным потребителем данных выступает компонент человеко-машинного интерфейса. Компонент, в котором содержится репозитарий, называется универсальным менеджером объектов (common object manager, COOB).

На рис. 15.13 иллюстрируется роль, которую COOB исполняет в период прогона. Содержание рисунка не ограничивается тем потоком данных, который обращается к COOB; на нем также изображены потоки, которые по соображениям производительности обходят этого менеджера стороной. Данные отслеживания (позиционная история цели), переносимые в крупной структуре данных, передаются напрямую от производителя к потребителю; то же самое в силу крайне высокой частоты обновления происходит при передаче информации от шарового манипулятора.

Ниже перечислены соглашения по производству данных.

- Отправка данных производится только в случае их изменения. Это предотвращает появление в сети избыточного трафика сообщений.
- В целях отделения программ от изменяющихся реализаций данные представляются в виде объектно-ориентированных абстракций. Строгая типизация позволяет обнаруживать ошибки, связанные с неверным использованием переменных, уже в период компиляции.
- С данными, которые они изменяют, компоненты связаны отношением принадлежности; они поставляют процедуры доступа, исполняющие роль диспетчеров. Поскольку к каждому блоку данных напрямую обращается только тот компонент, которому этот блок принадлежит, состязательность исключается.
- Возможность обращения к данным предоставляется всем заинтересованным сторонам на всех узлах системы. Привязка данных к определенному узлу не влияет на перечень компонентов, которые располагают доступом к ним.
- Благодаря распределению данных время отклика на запрос о поиске сокращается.

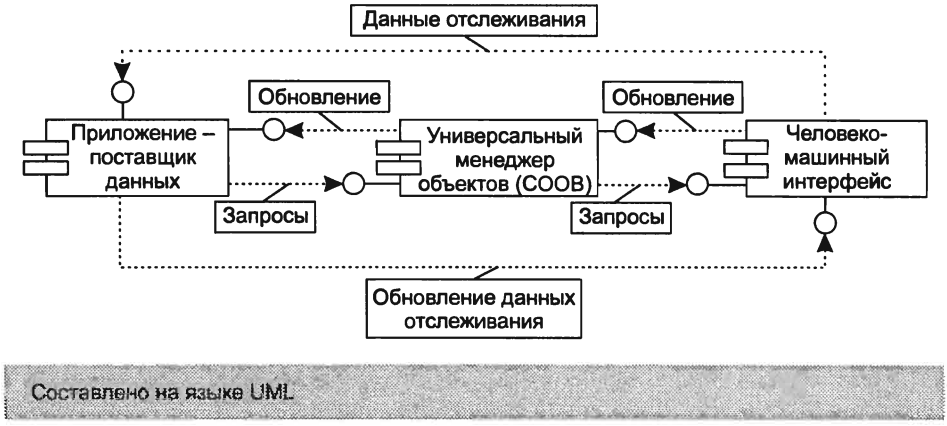


Рис. 15.13. Применение (и обход) менеджера СООВ

- В системе обеспечивается долгосрочная непротиворечивость данных. Кратковременная противоречивость допускается.

Сетевые соглашения заключаются в следующем:

- Сетевая нагрузка умышленно сокращается — иначе говоря, значительные усилия проектировщиков направляются на то, чтобы сделать поток данных в сети управляемым и обеспечить передачу по ней только важнейшей информации.
- Каналы передачи данных устойчивы к ошибкам. Приложения ориентируются на устранение ошибок по большей части за счет внутренних ресурсов.
- «Пропуск» приложением нерегулярных обновлений данных считается допустимым. К примеру, местоположение судна постоянно меняется, и пропущенное обновление информации о местоположении можно будет вывести на основе сопутствующих обновлений.

Существует также ряд других соглашений, не относящихся ни к одному из вышеперечисленных аспектов:

- Настраиваемость языка Ada широко задействуется как механизм повторного использования.
- Применяются стандартные протоколы исключений языка Ada.

Многие из этих соглашений (в частности, те, что касаются абстрактных типов данных, межпроцессорного взаимодействия, передачи сообщений и принадлежности данных) позволяют написать любой модуль вне зависимости от многочисленных изменяемых аспектов, которые он не контролирует. Другими словами, модули носят более общий характер и потому предполагают возможность непосредственного перенесения в сторонние системы.

Многоуровневое представление

Архитектура линейки продуктов SS2000 является многоуровневой.

- Модули группируются исходя из типа инкапсулированной в них информации. Модули, которые требуют модификации в случае изменения аппарат-

ной платформы, локальной сети или протоколов межузлового взаимодействия, составляют один уровень. Модули, которые реализуют общую для всех членов семейства функциональность, образуют другой уровень. Наконец, отдельный уровень отводится для индивидуальных модулей конкретного клиентского продукта.

- Уровни упорядочены — аппаратно-зависимые уровни, с одной стороны, прикладные — с другой.
- Деление на уровни является «строгим» — иначе говоря, взаимодействие уровней ограничивается. Модуль, находящийся на определенном уровне, может обращаться только к другим модулям своего уровня, а также к модулям следующего (по нисходящей) в иерархии уровня.

Нижний уровень в линейке SS2000 называется Base System 2000; он содержит интерфейс между операционной системой, аппаратным обеспечением и сетью, с одной стороны, и прикладными программами – с другой. Для прикладных программистов на уровне Base System 2000 предусматривается интерфейс программирования, при помощи которого они осуществляют межкомпонентное взаимодействие и передачу данных безотносительно к конкретным вычислительным платформам, сетевым топологиям, распределению функций между процессорами и т. д. Архитектурные уровни SS2000 изображены на рис. 15.14.

Представление декомпозиции на модули: системные функции и группы системных функций

В главе 2 мы упоминали о том, что модули, участвующие в представлении декомпозиции, в разных компаниях называются по-разному. Модули, применяемые CelsiusTech, называются системными функциями и группами системных функций.

Системная функция (system function) в SS2000 является первичным элементом декомпозиции на модули. Системная функция представляет собой совокупность программных средств, реализующих набор логически связанных требований. Состоит она из ряда блоков кода на языке Ada. *Группа системных функций* (system function group) содержит набор системных функций и является первичной единицей распределения обязанностей между группами разработчиков. В составе SS2000 примерно 30 групп системных функций, каждая из которых состоит из примерно 20 системных функций. Группируются они согласно основным функциональным областям — в частности, выделяются:

- функции командования, управления и связи;
- функции управления вооружением;
- фундаментальные функции — средства внутрисистемного взаимодействия и интерфейсы с вычислительной средой;
- человеко-машинный интерфейс.

Отношение между различными типами модулей изображено на рис. 15.15.

Группы системных функций могут состоять (и действительно состоят) из разноуровневых системных функций. Они соответствуют относительно крупным

Составлено на языке UML

Рис. 15.14. Многоуровневая программная архитектура SS2000

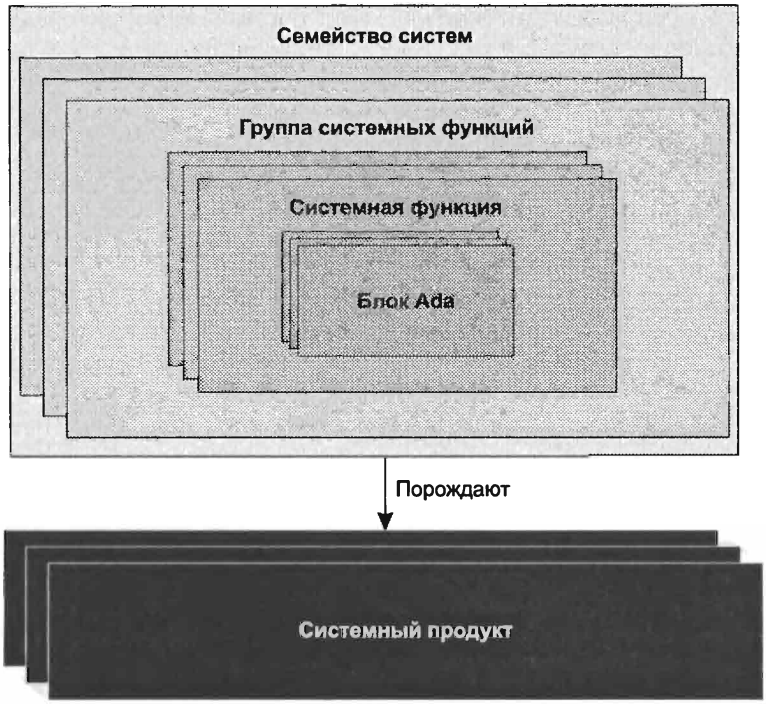


Рис. 15.15. Программные блоки в представлении декомпозиции на модули

функциональным блокам, которые обычно разрабатываются крупными командами разработчиков. В частности, для каждой группы системных функций составляется отдельная спецификация требований.

Именно системные функции и группы системных функций, а отнюдь не блоки кода Ada, являются базовыми единицами тестирования и интеграции в рамках линейки продуктов. Это довольно важно — любые новые члены линейки продуктов трактуются как сочетания нескольких десятков высококачественных, высоконадежных модулей, взаимодействие между которыми осуществляется контролируемым образом и предсказуемо; в этом их серьезное превосходство над тысячами мелких блоков, в отношении которых при каждом изменении приходится проводить регрессивное тестирование. Принцип повторного использования в CelsiusTech реализовывался именно за счет сборки крупных, заранее протестированных элементов.

Применение архитектуры SS2000

В табл. 15.1 приводится обзор архитектурных задач, предъявлявшихся к линейке SS2000, а также методик и тактик (см. главу 5) их реализации. В нижеследующем разделе, которым мы завершаем презентацию архитектуры SS2000, будут

рассмотрены четыре важных вопроса, возникших в процессе создания и сопровождения архитектуры, а также конструирования на ее основе семейства систем.

Таблица 15.1. Требования к SS2000 и архитектурные средства их реализации

Требование	Механизм реализации	Тактики
Производительность	Строгие протоколы сетевого трафика; в целях максимального усиления параллелизма программные средства писались как наборы процессов, независимые от местоположения; при транзакциях с высокими объемами данных менеджер COOB обходился; если же передача проходила через него, то данные отсылались только в случае изменения, а распространялись в расчете на минимизацию времени отклика	Введение параллелизма; сокращение потребления; многоэкземплярная тактика; расширение ресурсов
Надежность, готовность и безопасность	Резервированная локальная сеть; отказоустойчивые программные средства; стандартные протоколы исключений Ada; независимость программных средств от местоположения обеспечивала возможность миграции в случае отказа; строгая принадлежность данных исключала состязательность при записи	Исключения; активное резервирование; повторная синхронизация состояния; транзакции
Модифицируемость (в том числе способность к производству новых членов семейства SS2000)	Передача данных строго через сообщения обеспечила изолированность интерфейса от деталей реализации; программные средства писались независимыми от местоположения; разбиение на уровни способствовало переносимости между платформами, сетевыми топологиями, протоколами межпроцессорного взаимодействия и пр.; благодаря менеджеру COOB производители и потребители данных не знали о существовании друг друга; усиленная эксплуатация настраиваемости Ada и параметризации элементов; семантическая связность обеспечивалась через системные функции и группы системных функций	Семантическая связность; прогнозирование ожидаемых изменений; обобщение модулей; общие абстрактные службы; стабильность интерфейсов; посредник; конфигурационные файлы; замена компонентов; применение предписанных протоколов
Контролепригодность	Интерфейсы, применявшие строго типизированные сообщения, в период прогона исключают целый класс ошибок; строгая принадлежность данных; семантическая связность элементов и строгие описания интерфейсов облегчили задачу установления ответственности	Отделение интерфейса от реализации

Архитектура как основа для построения систем

При разборе данного конкретного примера мы активно продвигали тезис о том, что одних лишь технических решений при работе над линейкой продуктов недо-

статочно, — необходимо также принимать во внимание коммерческие и организационные соображения. И тем не менее именно архитектура послужила основным средством реализации линейки SS2000. В этом смысле огромную роль сыграли абстракции и многоуровневая организация. Благодаря абстрагированию удалось создать модули, инкапсулирующие изменяемые решения в границах интерфейсов. Когда такой модуль задействуется в том или ином продукте, в подходящий момент он конкретизируется путем параметризации. С течением времени, по мере удовлетворения очередных требований, модули меняются, однако содержащиеся в них изменяемые решения обеспечивают стабильность базы средств.

Размер и сложность рассматриваемой архитектуры и составляющих ее модулей обусловливают необходимость серьезных познаний в прикладной области — лишь в этом случае возможно разделение системы на модули для их независимой разработки. В линейках продуктов наподобие SS2000, характеризующихся высокой изменчивостью, такое решение, серьезно облегчающее развитие, полностью обосновано.

Сопровождение фонда базовых средств по мере производства новых систем

Как мы уже говорили, в качестве долгосрочного проекта в CelsiusTech рассматривается не отдельная судовая система, изготовленная по специальному заказу, и даже не совокупность размещенных к текущему моменту систем. Своей основной задачей компания видит ведение *линейки продуктов как таковой*. Ведение линейки продуктов подразумевает сопровождение фонда базовых средств, обеспечивающее возможность регенерации любого существующего члена линейки (в конце концов, по мере изменения требований они обнаруживают тенденцию к развитию и росту) и построения новых членов. В определенном смысле, сопровождение линейки продуктов означает поддержание *способности* к производству на основе базовых средств новых продуктов. Для достижения этой цели нужно сделать так, чтобы повторно используемые модули всегда были современными и универсальными. Ни один продукт не должен развиваться отдельно от линейки продуктов. В этом заключается решение задачи, сформулированной в главе 14 и связанной с синхронизацией развития линейки продуктов с развитием ее вариантов.

Далеко не каждый модуль используется во всех членах линейки продуктов. К примеру, требования по криптологии и пользовательским интерфейсам обусловливаются национальной спецификой — следовательно, чем пытаться выстроить универсальное решение, значительно разумнее сконструировать модули в расчете на конкретные системы. По этому принципу можно было бы даже вывести (в рамках общей линейки продуктов) своеобразные «подлинейки»: шведский набор продуктов, датский набор продуктов и т. д. Даже те модули, которые применяются в единичных случаях, проектируются и конструируются конфигурируемыми и гибкими, после чего вносятся в фонд базовых средств линейки продуктов — с расчетом на то, что их можно будет задействовать в последующих системах.

С точки зрения внешнего наблюдателя, CelsiusTech поставляет судовые системы. Сами же сотрудники компании трактуют свою деятельность как развитие и наращивание фонда общих средств, который, в свою очередь, обеспечивает

возможность изготовления судовых систем. Такое различие в осмыслении деятельности компании (а это именно осмысление), хоть и едва уловимо, все же проявляет себя в политиках управления конфигурациями, организации предприятия и методиках продвижения новых продуктов.

Сопровождение крупных, заранее интегрированных блоков

В классических трудах по программным репозитариям повторного использования в качестве единицы повторного использования, как правило, определяется либо мелкий модуль (например, пакет Ada, подпрограмма или объект), либо крупная, независимо исполняемая подсистема (например, инструментальный пакет или автономный коммерческий продукт). В первом случае модули нужно собрать, интегрировать, конфигурировать, отладить и тестировать; в последнем случае подсистемы обычно не обнаруживают высокой конфигурируемости и гибкости.

Специалисты из CelsiusTech избрали промежуточный вариант. В качестве единицы повторного использования применяется системная функция — цепочка родственной функциональности, содержащая модули из разных уровней архитектуры. Системные функции заранее интегрируются — иначе говоря, их модули собираются и компилируются совместно, а тестироваться могут как вместе, так и по отдельности. Извлекаемая из репозитария базовых средств системная функция полностью готова к употреблению. Таким образом, практика повторного использования в CelsiusTech распространяется не только на модули, но и на операции интеграции и тестирования, которые в иных случаях приходится проводить специально для каждого приложения.

Параметрические модули

В большинстве случаев при повторном использовании модулей никакие изменения в их код не вносятся; впрочем, утверждать, что изменения не требуются вообще, неправомерно. Во многих элементах абсолютные величины, варьирующиеся от системы к системе, заменяются символическими именами. Например, вычисления в модуле могут проводиться исходя из количества процессоров; знать их число при написании модуля, впрочем, необязательно. Таким образом, количество процессоров кодируется в виде символического значения — параметра, значение которому присваивается в ходе интеграции системы. Такой модуль, с одной стороны, корректно исполняется, а с другой — может быть задействован в новой версии системы с иным числом процессоров.

Со временем параметры зарекомендовали себя как удобные и эффективные инструменты реализации повторного использования модулей. Впрочем, на практике они слишком быстро размножаются. Путем параметризации любой модуль можно обобщить. В модулях линейки продуктов SS2000 содержится от 3000 до 5000 параметров, требующих индивидуальной подстройки под каждую конструируемую на основе этой линейки клиентскую систему. Специалисты CelsiusTech так и не нашли способа гарантировать, что при конкретизации в исполняемой системе те или иные сочетания значений параметров не приведут к вхождению в недопустимое рабочее состояние.

Многочисленность параметров в некоторой степени подорвала преимущества применения крупных системных функций и их групп в качестве базовых единиц

тестирования и интеграции. Новая версия системы, для которой подгоняются параметры, по сути, оказывается нетестированной. Более того, любое новое сочетание значений параметров теоретически способно ввести систему в неизвестное (и, естественно, непроверенное) рабочее состояние.

На практике же фиксируется лишь небольшая часть возможных сочетаний параметров. При этом нежелание испытывать новые сочетания препятствует проявлению присущей элементам гибкости (конфигурируемости).

Фактически, многочисленность параметров — это скорее проблема учета; мы не знаем ни одного случая, когда некорректное функционирование можно было бы отнести исключительно к недостаткам спецификаций параметров. Зачастую крупный модуль импортируется с тем же набором параметров, который применялся в предыдущем случае.

15.4. Заключение

В период с 1986 по 1998 год компания CelsiusTech прошла путь развития от оборонного подрядчика индивидуально конструируемых единичных решений до, по сути, производителя коммерческих коробочных военно-морских систем. Старую организационную структуру и руководство компания сочла непригодными для проведения в жизнь новаторской бизнес-модели. Кроме того, обнаружилось, что задача реализации и поддержания успешной линейки продуктов не ограничивается созданием нужных программных средств, грамотных архитектуры, среды разработки, аппаратных средств или сетей. Не меньшее значение на результат, как выяснилось, оказывают организационная структура, методы руководства и кадровое обеспечение.

Архитектура, впрочем, продемонстрировала себя основой всей методики — как с технической, так и с культурной точки зрения. В некотором смысле, она оказалась тем осязаемым объектом, создание и конкретизация которого заявлялись как конечная цель. В силу своей значимости архитектура оказалась в высшей степени видимой. Власть над ней, но, с другой стороны, и ответственность за ее развитие ложилась на участников компактной, высокопрофессиональной группы архитекторов. В результате была достигнута та «концептуальная целостность» архитектуры, которую Брукс [Brooks 95] считает основным условием успеха любого программного проекта.

Впрочем, с определения архитектуры процесс построения фундамента для долгосрочной разработки лишь начался. Серьезное значение придавалось проверке правильности, которую требовалось провести путем макетирования и с учетом начального опыта практического применения. По мере обнаружения недостатков архитектура подвергалась плавному, контролируемому развитию, которое, начавшись в период первоначальной разработки, продолжилось и в более поздние периоды. Для того чтобы поставить эту естественную эволюцию под контроль, группы сборщиков и архитекторов CelsiusTech объединили свои усилия — в результате любые изменения в важнейшие интерфейсы могли вноситься проектировщиками и группами проектировщиков исключительно при условии явного одобрения со стороны архитекторов.

Такой подход пользовался неограниченной поддержкой руководства проектом, и своей работоспособностью он во многом обязан именно авторитету группы архитекторов. При принятии проектных решений она выступала в качестве центральной, высшей инстанции, которую невозможно было обойти; таким образом, удалось добиться концептуальной целостности.

Созданию линейки продуктов, с одной стороны, и ее поддержанию и развитию — с другой, соответствуют различные организационные структуры. Руководство должно планировать изменения по части кадрового обеспечения, управления, обучения и потребностей компании. Построение жизнеспособной линейки продуктов требует участия архитекторов, обладающих комплексными знаниями в предметной области и высокой инженерной квалификацией. По мере планирования разработки новых продуктов и контроля над развитием линейки приходится обращаться к услугам экспертов в предметной области.

Переориентация CelsiusTech с единичных систем на линейку продуктов сопровождалась обучением и повышением квалификации руководства и технических специалистов. Это именно то, что мы называем возвратным циклом ABC.

15.5. Дополнительная литература

Есть два сообщения, повествующие о переходе CelsiusTech к методике построения линейки продуктов. Первое — составленное сотрудниками Института программной инженерии [Brownsword 96] — послужило основным источником материала для данной главы. Второе — это диссертация, защищенная в шведском университете г. Линкопинг [Cederling 92].

15.6. Дискуссионные вопросы

1. Можно ли на основе архитектуры CelsiusTech создать систему управления воздушным движением наподобие описанной в главе 6? Могла ли CelsiusTech, напротив, обратиться к архитектуре этой системы? В чем сущностные различия между двумя вариантами архитектуры?
2. В период разработки линейки продуктов SS2000 структура управления CelsiusTech несколько раз претерпевала изменения. Учитывая высказанный нами в главе 7 тезис о том, что структура продукта должна отражать структуру проекта, оцените воздействие этих изменений.

Глава 16
J2EE/EJB. Конкретный пример стандартной вычислительной инфраструктуры

(в соавторстве с Анной Лиу[1])

Пишется однажды, исполняется везде.
Девиз Java от Sun Microsystems

Пишется однажды, тестируется везде.
Присказка особо циничных программистов Java

В настоящей главе мы намерены представить вашему вниманию обзор спецификации корпоративной архитектуры Java 2 (Java 2 Enterprise Edition, J2EE) и поподробнее остановиться на одной из ее важнейших частей — системе корпоративных JavaBeans (Enterprise JavaBeans, EJB). J2EE — это стандартное описание методов проектирования и разработки распределенных объектно-ориентированных программ на языке Java, а также передачи данных и взаимодействия между различными компонентами Java. Спецификация EJB содержит описание компонентной модели программирования на стороне сервера. В целом, J2EE, помимо прочего, описывает разного рода корпоративные службы, в частности, именования, транзакций, жизненного цикла компонентов и устойчивости (persistence), а также методы единообразного обслуживания и обращения к службам. Наконец, эта спецификация регламентирует механизм инфраструктурного обслуживания разработчиков приложений производителями, направленный (при условии

[1] Анна Лиу (Anna Liu) — старший научный сотрудник в группе по программной архитектуре и технологиям (Software Architecture and Technologies Group) при научно-промышленной исследовательской организации Содружества Наций (Commonwealth Scientific Industrial Research Organization, CSIRO) (Сидней, Австралия) и одновременно адъюнкт-профессор Сиднейского университета.

соответствия стандарту) на переносимость конечного приложения в масштабах любых платформ J2EE.

J2EE/EJB — это лишь одна из многих методик конструирования распределенных объектно-ориентированных систем. В частности, в последнее десятилетие довольно широкое распространение получила обобщенная архитектура построения брокеров объектных запросов (Common Object Request Broker Architecture, CORBA) от рабочей группы по объектному менеджменту (Object Management Group, OMG). Согласно этой архитектуре, брокер объектных запросов (object request broker, ORB) позволяет объектам публиковать их интерфейсы, а клиентским программам (а иногда и другим объектам) — обнаруживать местонахождение удаленных объектов во всей компьютерной сети и запрашивать у них обслуживание. Компания Microsoft предлагает собственную технологию конструирования распределенных систем — она называется .NET. В архитектуре .NET аналогичные возможности построения распределенных объектных систем предоставляются для Windows-платформ.

В начале главы мы рассмотрим коммерческие факторы, обусловившие создание стандартной архитектуры распределенных систем; затем разберем реализацию соответствующих потребностей в архитектуре J2EE/EJB. Ознакомившись с типичными требованиями по качеству, предъявляемыми к веб-приложениям, мы постараемся уяснить механизм их удовлетворения средствами J2EE/EJB.

16.1. Связь с архитектурно-экономическим циклом

На протяжении 1980-х годов соотношение «цена/производительность» для персональных компьютеров постепенно приближалось к аналогичному показателю для профессиональных рабочих станций и «серверов». Этот приток вычислительных мощностей и ускорение сетевых технологий сделал возможным широкое распространение распределенной обработки данных.

В то же время конкурирующие производители компьютеров выпускали новые аппаратные средства, операционные системы и сетевые протоколы. С точки зрения компаний – конечных пользователей, все возрастающая дифференциация продуктов препятствовала внедрению распределенной обработки данных. Как правило, вложив средства в разные вычислительные платформы, в процессе построения распределенных систем в сильно гетерогенной среде компании испытывали значительные трудности.

Для решения этой проблемы в начале 1990-х годов силами рабочей группы по объектному менеджменту (Object Management Group, OMG) была разработана обобщенная архитектура построения брокеров объектных запросов (Common Object Request Broker Architecture, CORBA). Модель CORBA представляла собой стандартную программную платформу, на которой распределенные объекты могли обмениваться данными и взаимодействовать прозрачно и беспрепятственно. В таких условиях брокер объектных запросов (object request broker, ORB) позволяет объектам публиковать свои интерфейсы, и, в каком бы местоположении компьютерной сети они ни находились, клиентские программы могут обращаться к ним за обслуживанием.

Период, в течение которого CORBA оставалась единственной жизнеспособной технологией распределенных объектов, оказался непродолжительным. Вскоре компания Sun Microsystems выпустила язык программирования Java, предусматривавший поддержку удаленного вызова методов (remote method invocation, RMI) и, по сути, встраивавший специализированную для Java функциональность брокеров объектных запросов во все виртуальные машины Java (Java Virtual Machine, JVM). У Java есть такое преимущество, как переносимость. Код разработанного Java-приложения становится переносимым в масштабах всех JVM, которые реализованы на всех основных аппаратных платформах.

Sun Microsystems не стала останавливаться на Java. К концу 1990-х относится появление спецификации J2EE с Java RMI в качестве базовой инфраструктуры передачи данных. Для всего сообщества разработчиков программных средств она вскоре стала стандартом, упрощающим конструирование объектных систем на языке программирования Java. Позиции J2EE укрепились еще больше, когда производители программного обеспечения судорожно взялись за ее реализацию; с наступлением «эпохи Интернета» Java-программисты по всему миру активно осваивали каркас J2EE, разрабатывая приложения электронной коммерции. Таким образом, J2EE вступила в прямую конкуренцию одновременно с CORBA и со специализированными технологиями от Microsoft.

Архитектурно-экономический цикл J2EE/EJB изображен на рис. 16.1.

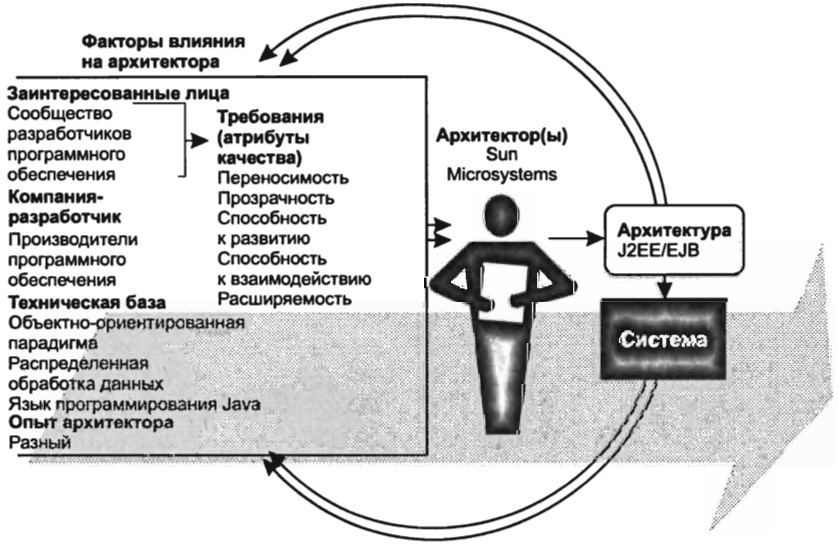


Рис. 16.1. Архитектурно-экономический цикл в связи с компанией Sun Microsystems и ее продуктами J2EE/EJB

16.2. Требования и атрибуты качества

Какие цели преследовала компания Sun Microsystems, взявшись за разработку спецификации J2EE/EJB? Как эти задачи отражены в атрибутах качества архитектуры J2EE/EJB?

Всемирная паутина и J2EE

В ответ на повышение требований, предъявляемых к коммерческим веб-системам, все больше количество корпоративных информационных систем конструируется на основе технологии распределенных объектов. Такие системы должны быть масштабируемыми, высокопроизводительными, переносимыми и безопасными. Обрабатывая огромное количество запросов от деятелей интернет-сообщества, они должны реагировать на них своевременно.

Самой сложной в настоящее время задачей для многих компаний электронной коммерции является достижение успешности сайтов. Чем успешнее сайт, тем больше на него обращений, однако, как мы говорили в главе 13, многочисленные обращения нагружают программное обеспечение. Очень многие интернет-сайты ежедневно регистрируют миллионы и даже десятки миллионов обращений. Снизить нагрузку можно за счет равномерного распределения пользователей между узлами в течение всего дня, однако на практике такие решения реализуются не слишком часто. Обычно запросы поступают крупными пакетами, в связи с чем к программному обеспечению сайтов выставляются более серьезные требования.

Истории о сайтах электронной коммерции, не выдержавших неожиданного притока посетителей, слышны чуть ли не на каждом углу. К примеру, в 1999 году во время Уимблдонского теннисного турнира на его сайт поступил почти 1 миллиард обращений, а во время одного из матчей число обращений достигло 420 000 в минуту (7000 в секунду)! Не стоит забывать, что Интернетом в настоящее время пользуется лишь незначительная часть населения Земли, — все только начинается.

В этом смысле правомерно утверждать, что Интернет навсегда изменил требования, предъявляемые к корпоративным программным системам. По самому своему характеру Интернет организует для приложений такие нагрузки, которые в традиционных сетевых информационных системах встречаются довольно редко. Когда приложения открываются для потенциально неограниченного числа одновременных обращений, значимость требований по таким атрибутам качества, как управляемость, масштабируемость, безопасность и готовность, резко возрастает. В табл. 16.1 перечислены требования по качеству, которым должны соответствовать все без исключения веб-приложения.

Разрабатывая спецификацию J2EE, компания Sun Microsystems стремилась создать технологическую базу, упрощающую конструирование подобного рода систем. В частности, спецификация EJB, входящая в состав J2EE, направлена на решение следующих задач.

- Создание компонентной архитектуры построения на языке Java распределенных объектно-ориентированных бизнес-приложений. Система EJB позволяет конструировать распределенные приложения за счет сочетания компонентов, разработанных инструментальными средствами разных производителей.
- Упрощение процесса написания приложений. Разработчикам приложений не приходится иметь дело с низкоуровневыми деталями транзакций и управления состоянием, равно как и с многопоточной обработкой и организацией пула ресурсов.

Таблица 16.1. Типичные требования по атрибутам качества, предъявляемые к веб-приложениям

Атрибут качества	Требование
Масштабируемость	Система должна предусматривать возможность колебаний нагрузки и реагировать на них без вмешательства человека
Готовность/надежность	Система должна находиться в состоянии готовности 24 часа в сутки, 7 дней в неделю, периоды простоя должны сводиться к минимуму
Безопасность	Система должна проводить аутентификацию пользователей и предотвращать несанкционированный доступ к данным
Практичность	У самых разных пользователей должна быть возможность обращаться к разному содержимому в разных формах
Производительность	Системы, с которыми взаимодействуют пользователи, должны демонстрировать высокую реактивность

Конкретнее, архитектура EJB выполняет следующие задачи:

- влияет на аспекты разработки, размещения и исполнения жизненного цикла корпоративного приложения;
- задает контракты, гарантирующие разработку и размещение компонентов, обладающих возможностью взаимодействия в период прогона, инструментальными средствами разных производителей;
- взаимодействует с другими интерфейсами прикладного программирования API;
- обеспечивает способность к взаимодействию между корпоративными beans и не-Java-приложениями;
- взаимодействует с CORBA.

J2EE допускает повторное использование Java-компонентов в рамках инфраструктуры на стороне сервера. При наличии подобающих инструментов сборки и размещения компонентов задача заключается в том, чтобы перенести удобство программирования в конструкторах с графическим пользовательским интерфейсом (типа Visual Basic) на процесс построения серверных приложений. Учитывая то обстоятельство, что стандартный каркас продуктов J2EE основывается на одном языке (Java), компонентные решения J2EE (по крайней мере, в теории) демонстрируют независимость от конкретных продуктов и переносимость в пределах платформ J2EE от разных производителей. Таким образом, в дополнение к представленным в табл. 16.1 базовым требованиям компания Sun вводит набор требований, касающихся действий группы программистов. Эти добавочные требования по атрибутам качества перечислены в табл. 16.2.

Таблица 16.2. Требования по атрибутам качества, предъявляемые компанией Sun применительно к J2EE

Атрибут качества	Требование
Переносимость	J2EE должна предусматривать реализацию на самых разных вычислительных платформах минимальными усилиями
Возможность построения	Разработчикам приложений необходимо предоставить средства управления общими службами — в частности, службами транзакций, именования и безопасности

продолжение

Таблица 16.2 (*продолжение*)

Атрибут качества	Требование
Умеренная специфичность	Детализация, предоставляющая разработчикам, производителям и сборщикам компонентов осмысленный стандарт, в сочетании с универсальностью, предусматривающей возможность создания отдельными производителями дополнительных характеристик и проведения оптимизации
Прозрачность реализации	Полная прозрачность деталей реализации, обеспечивающая независимость клиентских программ от деталей реализации объектов (местоположения компонента на стороне сервера, операционной системы, производителя и т. д.)
Способность к взаимодействию	Обеспечение возможности взаимодействия серверных компонентов, выполненных на основе реализаций от разных производителей; наличие мостов для взаимодействия платформы J2EE с другими технологиями наподобие CORBA и компонентной технологии Microsoft
Способность к развитию	Предоставление разработчикам возможности пошагового принятия различных технологий
Расширяемость	Обеспечение возможности внедрения значимых новых технологий по мере их разработки

16.3. Архитектурное решение

Рассматриваемая в предыдущем разделе методика удовлетворения требований по атрибутам качества, предложенная компанией Sun Microsystems, опирается на спецификации двух основных вариантов архитектуры: J2EE и EJB. J2EE описывает общую многозвенную архитектуру проектирования, разработки и размещения компонентных корпоративных приложений. Спецификация EJB как основной элемент технологии J2EE отражает более глубокие технические требования к возможности построения, расширяемости и способности к взаимодействию. Как J2EE, так и EJB выражают умеренную специфичность (balanced specificity) — иначе говоря, способность конкурирующих сторон дифференцировать свои предложения, сохраняя их в то же время на универсальной базе.

Основными характеристиками платформы J2EE являются:

- многозвенная распределенная прикладная модель;
- серверная компонентная модель;
- встроенное управление транзакциями.

Простое представление размещения многозвенной модели изображено на рис. 16.2. Элементы этой архитектуры расписаны более подробно в табл. 16.3.

Таблица 16.3. Обзор компонентов и служб в рамках технологии J2EE

Компонент/служба	Описание
Архитектура Enterprise JavaBeans (EJB)	В спецификации устанавливается интерфейс прикладного программирования, позволяющий разработчикам создавать, размещать и контролировать серверные компонентные приложения корпоративного уровня

Компонент/служба	Описание
JavaServer Pages (JSP)	Метод создания динамических веб-страниц
Java-сервлеты (Java Servlet)	Предоставляет в распоряжение разработчиков веб-приложений механизм расширения функциональности веб-сервера
Служба сообщений Java (Java Messaging Service, JMS)	Обеспечивает поддержку приложениями J2EE асинхронной передачи сообщений в двухточечном (один на один) режиме или в стиле взаимодействия «публикация–подписка» (многие ко многим); сообщения конфигурируются с расчетом на несколько вариантов качества обслуживания — от «без обязательств» до транзактного
Java-интерфейс именования и каталогов (Java Naming and Directory Interface, JNDI)	Служба каталогов J2EE позволяет Java-клиенту и сервлетам веб-звена извлекать ссылки на определяемые пользователем объекты наподобие EJB и элементов среды (например, местонахождение драйвера JDBC)
Служба транзакций Java (Java Transaction Service, JTS)	Обеспечивает элементам EJB и их клиентам возможность участия в транзакциях; ряд beans в приложении обновляется, а JTS гарантирует фиксацию или отмену изменений по окончании транзакции; драйверы JDBC-2 предоставляют поддержку протокола XA, а следовательно, и возможности осуществлять распределенные транзакции с участием одного или нескольких диспетчеров ресурсов
Соединительная архитектура J2EE (J2EE Connector Architecture, JCA)	Определяет стандартную архитектуру соединения платформы J2EE с разнородными корпоративными информационными системами — в частности, с пакетными приложениями корпоративного планирования ресурсов (Enterprise Resource Planning, ERP) и системами управления взаимодействием с заказчиками (Customer Relationship Management, CRM)
COM-мост обслуживания клиентского доступа (Client Access Services COM Bridge)	Предусматривает сетевую интеграцию приложений COM и J2EE; обеспечивает для клиентских приложений с поддержкой COM возможность доступа к серверным компонентам J2EE
Удаленный вызов методов через межброкерный протокол Интернет (RMI over IIOP)	При помощи стандартного межброкерного протокола Интернет (Internet Inter-ORB Protocol, IIOP) от OMG предоставляет разработчикам реализацию интерфейса прикладного программирования Java RMI; разработчики получают возможность написания удаленных интерфейсов между клиентами и серверами и их реализации посредством технологии Java и интерфейсов прикладного программирования Java RMI
Java-интерфейс связи с базами данных (Java Database Connectivity, JDBC)	Предоставляет программистам единообразный интерфейс с разного рода реляционными базами данных и общую базу для построения высокоуровневых инструментальных средств и интерфейсов

- *Звено клиента.* Клиентским звеном в веб-приложении является интернет-браузер, подающий HTTP-запросы и загружающий с веб-сервера страницы HTML. В приложениях, размещаемых в обход браузера, возможно участие автономных Java-клиентов или апплетов, которые напрямую взаимодействуют со звеном бизнес-компонентов. (Пример использования J2EE в обход браузера приводится в главе 17.)
- *Веб-звено.* Веб-звено сводится к веб-серверу, который обрабатывает клиентские запросы и отвечает на них путем запуска сервлетов J2EE или JavaServer Pages (JSPs). Сервлеты запускаются сервером в зависимости от типа

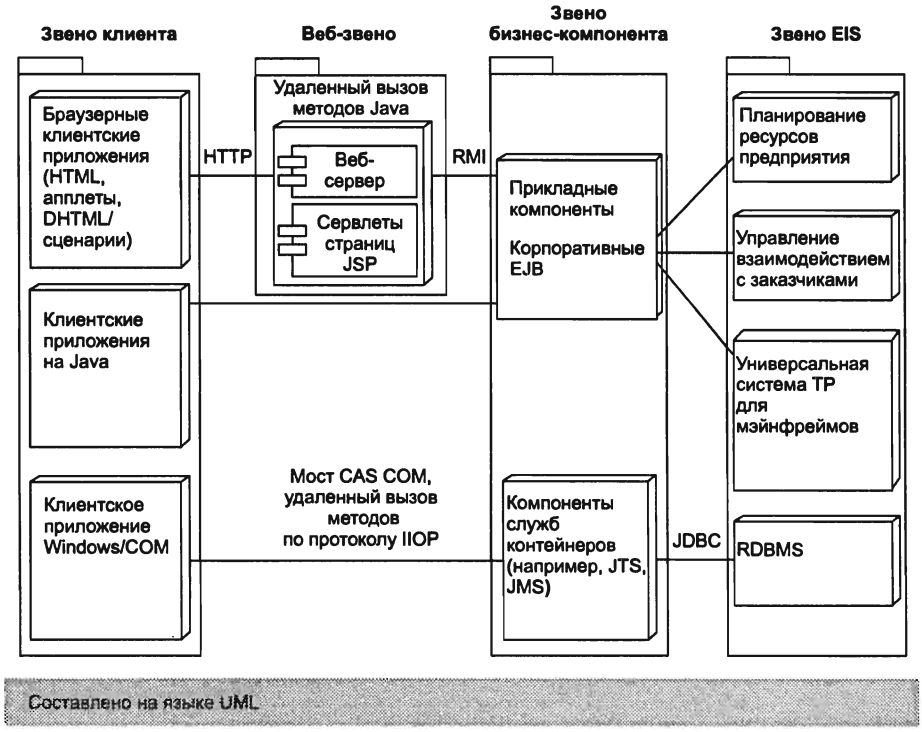


Рис. 16.2. Представление размещения многозвенной архитектуры J2EE. Ниже расписаны роли отдельных звеньев

пользовательского запроса. Они запрашивают на предмет требуемой информации звено бизнес-логики, а затем, перед тем как посредством сервера возвратить ее пользователю, выполняют форматирование. JSP являют собой статические страницы HTML, в которых содержатся фрагменты кода сервлета. Код, запускаемый механизмом JSP, принимает обязанность по форматированию динамической части страницы.

- *Звено бизнес-компонентов.* Бизнес-компоненты составляют основу бизнес-логики приложения. Реализуются они посредством EJB (это программно-компонентная модель, поддерживаемая J2EE). EJB принимают запросы от сервлетов, относящихся к веб-звену, затем в целях их удовлетворения, как правило, обращаются к источникам данных и, наконец, возвращают результаты сервлету. Размещаются компоненты EJB в среде J2EE внутри EJB-контейнера. Последний предоставляет своим EJB ряд служб — в частности, связанных с управлением транзакциями и жизненным циклом, управлением состоянием, безопасностью, многопоточной обработкой и организацией пула ресурсов. EJB лишь указывают тип поведения, которого контейнер должен придерживаться в период исполнения, а в остальном полностью полагаются на обслуживание со стороны контейнера. В результате прикладные программисты избавляются от необходимости забивать бизнес-логику кодом в целях решения системных задач и задач среды.

- *Звено корпоративных информационных систем.* Как правило, оно состоит из одной или нескольких баз данных и серверных приложений — например, мэйнфреймов или каких-либо других унаследованных систем, к которым при обработке запросов обращаются EJB. С базами данных — а в этом качестве чаще всего выступают системы управления реляционными базами данных (Relational Database Management Systems, RDBMS)) — обычно применяются драйверы JDBC.

Архитектурная методика EJB

Оставшаяся часть главы посвящена архитектуре корпоративных JavaBeans (Enterprise JavaBeans, EJB), которая определяет стандартную модель программирования, направленную на построение распределенных объектно-ориентированных серверных Java-приложений. В силу стандартного статуса этой модели существует (и эксплуатируется) возможность написания beans с заранее «пакетированной» полезной функциональностью. Задача программиста EJB, таким образом, заключается в том, чтобы связать эти пакеты с прикладной функциональностью и тем самым сформировать законченное приложение.

Подобно J2EE, архитектура EJB ориентирована на реализацию одного из основных принципов проектирования на Java — пресловутого девиза «пишется однажды, исполняется везде». JVM позволяет запускать Java-приложение на любой операционной системе. Впрочем, серверные компоненты выставляют ряд требований по дополнительному обслуживанию (например, по предоставлению служб транзакций и безопасности), предоставить которые непосредственно JVM не в состоянии. Как в J2EE, так и в EJB эти службы предоставляются посредством набора стандартных, независимых от конкретного производителя интерфейсов, обеспечивающих доступ к вспомогательной инфраструктуре, — совместными усилиями они организуют обслуживание, предусматриваемое любым сервером приложений.

В каждом J2EE-совместимом сервере приложений предусматривается EJB-контейнер (EJB container), координирующий исполнение компонентов приложения. Выражаясь более приземленно, контейнер предоставляет процесс операционной системы, в котором размещается один или (в большинстве случаев) несколько EJB-компонентов.

На рис. 16.3 изображено отношение между сервером приложений, контейнером и предоставляемыми службами. Вкратце, сразу после запуска клиентом серверного компонента контейнер автоматически назначает ему поток и запускает экземпляр вызванного компонента. От имени компонента контейнер управляет всеми ресурсами, а также взаимодействием компонента и внешних систем.

Компонентная модель EJB определяет базовую архитектуру EJB-компонента — она задает структуру его интерфейсов и механизмов взаимодействия с контейнером и другими компонентами. Кроме того, эта модель регламентирует разработку компонентов с возможностью совместной работы в крупном приложении.

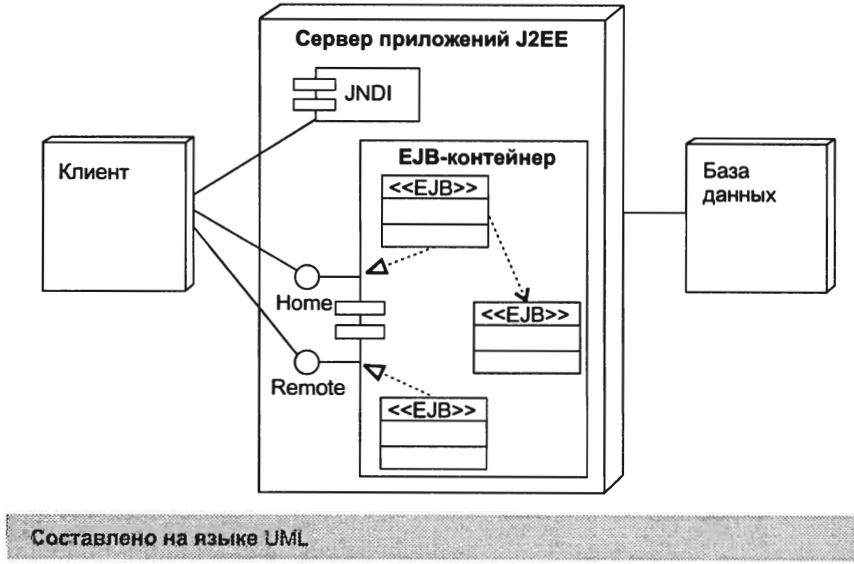


Рис. 16.3. Пример представления размещения архитектуры EJB

В спецификации EJB версии 1.1 выделяют два типа компонентов: *сеансовые beans* (session beans) и *beans-сущности* (entity beans).

- *Сеансовые beans* обычно содержат бизнес-логику и обслуживают клиентов. Сеансовый bean делится на два подтипа: *без состояния* (stateless) и *с запоминанием состояния* (stateful).
  - *Сеансовый bean без состояния* (stateless session bean) не вступает в диалог с вызывающим процессом. Таким образом, информация о состоянии от имени клиентов не сохраняется. Клиент, получающий ссылку на сеансовый bean без состояния в контейнере, неоднократно вызывает через него экземпляр bean. В период между последовательными вызовами службы никаких гарантий относительно связывания любого конкретного экземпляра сеансового bean без состояния клиенту не предоставляется. EJB-контейнер делегирует клиентские вызовы сеансовым beans без состояния лишь *по мере необходимости*; таким образом, у клиента нет сведений о том, с каким bean ему придется общаться. Отсюда следует, что хранить клиентское состояние в сеансовом bean без состояния бессмысленно.
  - *Сеансовый bean с запоминанием состояния* должен вступать в диалог с вызывающим процессом и сохранять информацию о состоянии этого диалога. С того момента как клиент получает ссылку на сеансовый bean с запоминанием состояния, все последующие вызовы по этой ссылке проходят через один и тот же экземпляр bean. Для каждого клиента, создающего экземпляр bean, контейнер формирует новый, специализированный сеансовый bean с запоминанием состояния. Следовательно, в таком bean клиенты вольны сохранять любую информацию о состоя-

нии, которая гарантированно сохраняется в нем до следующего обращения. Обязанность по управлению жизненным циклом сеансовых beans с запоминанием состояния берут на себя EJB-контейнеры. Если состояние bean в течение определенного периода времени остается без употребления, EJB-контейнер записывает его состояние на диск, а при последующем клиентском вызове bean осуществляет автоматическое восстановление. Этот механизм называется пассивацией/активацией (passivation and actication) bean с запоминанием состояния. Пассивацию мы чуть позже разберем более подробно.

- *Beans-сущности*, как правило, представляют бизнес-объекты данных. Члены данных bean-сущности напрямую отображаются на отдельные элементы данных, хранящиеся в связанной базе данных. Обращаются к beans-сущностям чаще всего сеансовые beans, предоставляющие клиентские службы бизнес-уровня. Beans-сущности подразделяются на два типа: *с контейнерным управлением устойчивостью* (container-managed persistence) и *с самоуправлением устойчивостью* (bean-managed persistence). Устойчивость в данном контексте означает способ записи и считывания данных bean (как правило, они хранятся в строках таблиц реляционных баз данных).
  - *Beans-сущности с контейнерным управлением устойчивостью* предполагают автоматическое отображение данных, представляемых bean, на связанное постоянное хранилище данных (например, на базу данных), осуществляемое средствами контейнера. Контейнер ответствен за загрузку данных в экземпляр bean и последующую (происходящую в определенные моменты — например, в начале и в конце транзакции) запись изменений в постоянное хранилище данных. Устойчивость с контейнерным управлением базируется на службах контейнера и не требует прикладного кода — благодаря тому, что контейнер генерирует код доступа к данным, реализация упрощается.
  - *Beans-сущности с самоуправлением устойчивостью* предполагают ответственность bean за обращения к представляемым постоянным данным, которые обычно осуществляются посредством ручных вызовов по JDBC. Устойчивость под управлением beans предоставляет разработчику дополнительную гибкость, необходимую для выполнения слишком сложных для контейнера операций, а также для обращения к не поддерживаемым контейнером источникам данных — в частности, к специальным или унаследованным базам данных. Реализация устойчивости под управлением beans требует от программиста более серьезных усилий, однако в определенных ситуациях труды компенсируются возможностью дополнительной оптимизации доступа к данным, а значит, и более высокой (по сравнению с устойчивостью с контейнерным управлением) производительностью.

В табл. 16.4 расписывается реализация архитектурой EJB основных требований по атрибутам качества, предъявляемых Sun к архитектуре J2EE в целом. Пример представления размещения архитектуры J2EE/EJB приводится на рис. 16.4.

Таблица 16.4. Реализация в EJB требований по атрибутам качества, предъявляемых компанией Sun к J2EE

Задача	Реализация	Тактики
Готовность/ надежность	J2EE-совместимые системы предоставляют готовые к употреблению службы транзакций, которые повышают готовность и надежность за счет встроенных механизмов восстановления после отказа	Heartbeat-запросы, транзакции, пассивное резервирование
Умеренная специфичность	Специфицируемые в категориях интерфейсов прикладного программирования Java, службы EJB отдают принятие реализационных решений конструкторам серверов приложений EJB; за счет уровня детализации разработчики, производители и сборщики компонентов получают содержательный стандарт; с другой стороны, уровень обобщения допускает введение дополнительных характеристик и проведение оптимизации «от производителя»	Прогнозирование ожидаемых изменений; общие абстрактные службы; сокрытие информации
Возможность построения	Серверы приложений EJB предлагают многочисленные готовые к употреблению службы конструирования серверных приложений Java — в частности, службы транзакций, устойчивости, многопоточной обработки и управления ресурсами; таким образом, разработчику не приходится иметь дело с низкоуровневыми деталями распределения; компанией Sun Microsystems создана эталонная реализация J2EE; производители серверов приложений также участвуют в процессе специфицирования J2EE	Абстрактные общие службы; обслуживание интерфейсов; информационная закрытость
Способность к развитию	Спецификация поделена на подкатегории, характеризующиеся возможностью независимого развития; координация запросов на спецификацию Java и соответствующих откликов осуществляется силами процесса сообщества Java (Java Community Process)	Семантическая связность; информационная закрытость
Расширяемость	Компонентный подход к спецификации EJB допускает последующее расширение; управляемые сообщениями beans введены в поздних версиях спецификации EJB, но при этом применимы в существующих системах EJB; спецификация J2EE содержит описание стабильных, востребованных большинством разработчиков компонентов базовых технологий наподобие EJB, JMS, JNDI, JTS и пр.; со временем происходит постепенное внедрение расширений (например, JCA)	Прогнозирование ожидаемых изменений

Задача	Реализация	Тактики
Прозрачность реализации	Спецификации интерфейсов Home и Remote способствуют минимизации сцепления спецификации и реализации интерфейсов. Таким образом, незаметные для клиента реализационные решения откладываются; обеспечивается полная прозрачность деталей реализации, благодаря которой клиентские программы становятся независимыми от деталей реализации объектов (местоположения серверных компонентов, операционной системы, производителя и т. д.)	Обслуживание существующих интерфейсов; семантическая связность
Способность к взаимодействию	Поддержка взаимодействия серверных компонентов, выполненных на основе реализаций от разных производителей; предоставление мостов, обеспечивающих возможность взаимодействия платформы J2EE с другими технологиями наподобие CORBA и компонентной технологии Microsoft	Применение предписанных протоколов
Производительность	Принцип распределенных компонентов применительно к J2EE/EJB позволяет регулировать производительность в масштабах нескольких систем	Конфигурационные файлы; выравнивание нагрузки; многоэкземплярная тактика
Переносимость	Контракты между элементами EJB и контейнерами гарантируют переносимость прикладных компонентов на различные контейнеры EJB; спецификация J2EE описывает роли производителей, сборщиков и специалистов по размещению прикладных компонентов, производителей EJB-серверов и EJB-контейнеров, системных администраторов; в ней же содержится точное описание контрактов между различными компонентами J2EE и прикладными компонентами; таким образом, теоретически, любой прикладной компонент характеризуется переносимостью в масштабах разных контейнеров J2EE; J2EE основывается на языке с собственной виртуальной машиной, реализованном на большинстве распространенных платформ	Обслуживание существующих интерфейсов; обобщение модулей; общие абстрактные службы
Масштабируемость	В многозвенной архитектуре J2EE и компонентной архитектуре EJB заложены встроенные механизмы расширения числа серверов в конфигурации и выравнивания нагрузки между серверами	Выравнивание нагрузки
Безопасность	J2EE-совместимые системы содержат готовые к употреблению декларативные, ролевые механизмы обеспечения безопасности (в том числе программируемой)	Аутентификация; авторизация; секретность данных

продолжение

Таблица 16.4 (*продолжение*)

Задача	Реализация	Тактики
Практичность	В J2EE-совместимых системах допускается возможность применения Java-технологий (в частности, JSP и сервлетов) визуализации контента согласно нуждам различных категорий пользователей	Отделение пользовательского интерфейса

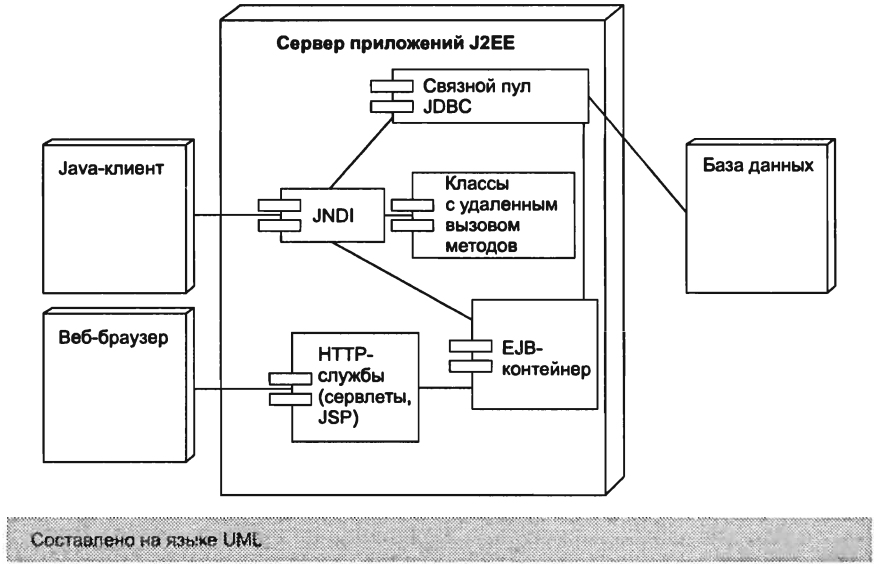


Рис. 16.4. Пример J2EE/EJB-совместимой реализации

EJB-программирование

Всю внешнюю информацию EJB получает от соответствующего контейнера. Если элементу EJB требуется получить доступ к JDBC-соединению или к другому bean, он обращается к службам контейнера. Доступ к индивидуальности вызывающего элемента, получение ссылки на самого себя, обращение к свойствам — все эти операции проводятся с помощью служб контейнера. Это наглядный пример тактики «введение посредника». Взаимодействие bean с контейнером обеспечивают три механизма: методы обратного вызова, интерфейс `EJBContext`, а также Java-интерфейс именования и каталогов (Java Naming and Directory Interface, JNDI).

Для создания серверного компонента EJB разработчику нужны два интерфейса, определяющие бизнес-методы bean, а также класс с его реализацией. Два упомянутых интерфейса — `remote` и `home` — показаны на рис. 16.5. С их помощью клиенты обращаются к bean внутри EJB-контейнера. Они показывают возможности bean и предоставляют все методы, необходимые для его создания, обновления, взаимодействия или удаления.

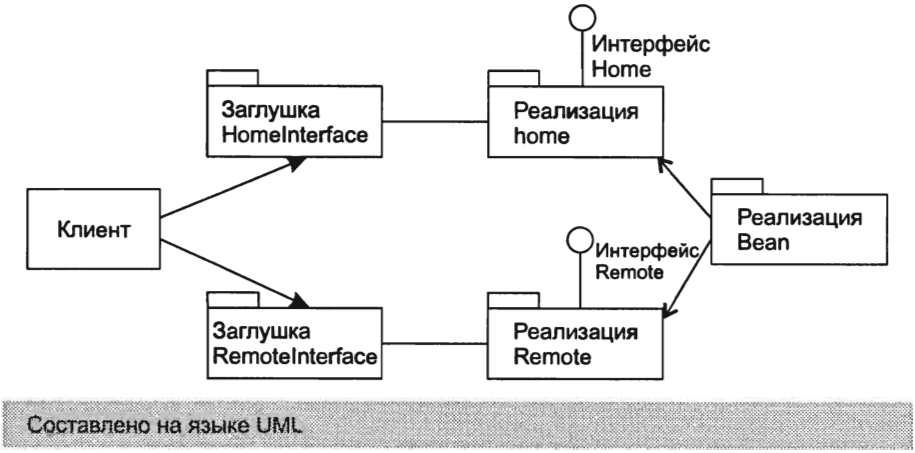


Рис. 16.5. Диаграмма пакетов EJB

Эти интерфейсы, по существу, принадлежат к двум разным категориям, выполняющим различные задачи. Интерфейс категории «home» (будем называть его home-интерфейсом) содержит методы жизненного цикла EJB, обслуживающие клиентов по части создания, уничтожения и поиска экземпляров bean. Интерфейс категории «remote» (будем называть его remote-интерфейсом), в свою очередь, содержит предоставляемые bean бизнес-методы. Все эти методы носят прикладной характер. Для того чтобы обращаться к ним через remote-интерфейс, клиенты должны с помощью home-интерфейса получить ссылку на него.

Элементарный home-интерфейс приводится в листинге 16.1. Наследующий от интерфейса **EJBHome**, в данном случае он содержит метод создания EJB типа **Broker**. Листинг 16.2 содержит код remote-интерфейса EJB типа **Broker**.

Remote-интерфейсы расширяют интерфейс **EJBObject**, который включает в себя ряд методов, применяемых контейнером для управления созданием и жизненным циклом элемента EJB. Программист волен делать выбор между принятием стандартного, унаследованного поведения и созданием для EJB индивидуального поведения. Впоследствии при помощи интерфейсов категории public клиент создает, управляет и удаляет beans с EJB-сервера. Класс реализации, называемый обычно bean-классом, становится доступным распределенным объектам после конкретизации в период прогона. Несколько упрощенный пример клиентского кода приводится в листинге 16.3.

Листинг 16.1. Простой home-интерфейс

```
public interface BrokerHome extends EJBHome
   {
   /*
   * Этот метод создает EJB Object.
   *
   * @return только что созданный EJB Object.
   */
   Broker create() throws RemoteException, CreateException;
   }
```

Листинг 16.2. Remote-интерфейс брокера

```
public interface Broker extends EJBObject
   {
   // Возврат только что созданного номера счета
   public int newAccount(String sub_name, String sub_address, int
      sub_credit) throws RemoteException, SQLException;
   public QueryResult queryStockValueByID(int stock_id)
      throws RemoteException, SQLException;
   public void buyStock(int sub_accno, int stock_id, int amount)
      throws RemoteException, SQLException, TransDenyException;
   public void sellStock(int sub_accno, int stock_id, int amount)
      throws RemoteException, SQLException, TransDenyException;
   publicvoid updateAccount(int sub_accno, int sub_credit)
      throws RemoteException, SQLException;
   public Vector getHoldingStatement(int sub_accno, int start_
      stock_id) throws RemoteException, SQLException;
   }
```

В качестве клиентов EJB могут выступать автономные приложения, сервлеты, апплеты и даже (в чем вы вскоре сможете убедиться) другие элементы EJB. Для получения ссылки на экземпляр серверного bean все клиенты обращаются к его home-интерфейсу. Ссылка эта ассоциирована с типом класса remote-интерфейса серверного bean; таким образом, взаимодействие клиента с серверным bean осуществляется исключительно теми методами, которые определены в его remote-интерфейсе.

В следующем примере участвует сеансовый bean без состояния **Broker**, обрабатывающий все клиентские запросы. Внутренне, для осуществления бизнес-логики он обращается к службам ряда beans-сущностей. Пример одного из методов компонента **Broker** — **updateAccount** — показан в листинге 16. 4.

Метод **updateAccount** использует bean-сущность под именем **Account**. Последняя инкапсулирует все детали обработки данных приложения — в данном случае процедуру обновления записи счета. Код метода **updateAccount** обращается к finder-методу с именем **findByPrimaryKey**, содержащемуся в home-интерфейсе bean-сущности **Account**. Принимая первичный ключ счета, этот метод обращается к соответствующей базе данных. Если первичного ключа оказывается достаточно для обнаружения в базе данных нужной записи счета, EJB-контейнер создает bean-сущность **Account**. Методы bean-сущности — в данном случае **update** — впоследствии обеспечивают доступ к данным в записи счета. Интерфейсы категорий home и remote bean-сущности **Account** показаны в листинге 16.5.

Листинг 16.3. Упрощенный пример клиентского кода EJB

```
Broker broker = null;
// Поиск интерфейса home
Object _h = ctx.lookup("EntityStock.BrokerHome");
BrokerHome home = (BrokerHome)
   javax.rmi.PortableRemoteObject.narrow(_h, BrokerHome.class);
// Создание Broker EJB Object с помощью интерфейса home
   broker = home.create();
// Исполнение запросов на брокере EJB
broker.updateAccount(accountNo, 200000);
broker.buyStock(accountNo, stockID, 5000);

// Готово....
broker.remove();
```

Листинг 16.4. Метод updateAccount bean-компонента Broker

```
public void updateAccount(int sub_accno, int sub_credit)
   throws RemoteException
   {
   try {
      Account account = accountHome.findByPrimaryKey
         (new AccountPK(sub_accno));
      account.update(sub_credit);
      }
   catch (Exception e) {
      throw new RemoteException(e.toString());
      }
   }
```

За реализацию удаленных (remote) методов отвечает bean-класс bean-сущности. Код метода update приводится в листинге 16.6. Он очень простой — фактически, он ограничивается одной-единственной строкой исполняемого кода Java. Такая простота достигается за счет применения данной bean-сущностью *устойчивости с контейнерным управлением* (container-managed persistence). EJB-контейнер «знает» (в этом вы скоро убедитесь) о соответствии между элементами данных в bean Account и полями в таблице счета из базы данных, к которой обращается приложение.

С помощью этой информации инструментальные средства контейнера генерируют SQL-запросы, необходимые для реализации finder-метода, а также другие запросы, обеспечивающие автоматизацию записи/считывания данных из/в bean-сущность в начале/конце транзакции. В нашем примере в конце метода updateAccount сеансового bean Broker элементы данных в bean-сущности Account записываются обратно в базу данных, обеспечивая тем самым устойчивый характер изменений в поле sub_credit. Все эти операции проводятся без явного контроля со стороны программиста, что свидетельствует о простоте построения систем на основе EJB.

Листинг 16.5. Интерфейсы категорий home и remote bean-сущности Account

```
public interface AccountHome extends EJBHome
   {
   /*
   * Этот метод создает EJB Object.
   *
   * @param sub_name Имя подписчика
   * @param sub_address Адрес подписчика
   * @param sub_credit Начальный взнос подписчика
   *
   * @return Только что созданный EJB Object.
   */
   public Account create(String sub_name, String sub_address,
      int sub_credit) throws CreateException, RemoteException;
   /*
   * Поиск Account по его первичному ключу (Account ID)
   */
   public Account findByPrimaryKey(AccountPK key)
      throws FinderException, RemoteException;
   }

   public interface Account extends EJBObject
   {
```

```
   public void update(int amount) throws RemoteException;
   public void deposit(int amount) throws RemoteException;
   public int withdraw(int amount) throws AccountException,
      RemoteException;
   // Методы «получатель» и «установщик» в полях Entity Bean
   public int getCredit() throws RemoteException;
   public String getSubName() throws RemoteException;
   public void setSubName(String name) throws RemoteException;
   }
```

Листинг 16.6. Метод update bean-сущности Account

```
public class AccountBean implements EntityBean
   {
   // Поля состояния под управлением контейнера
   public int     sub_accno;
   public String  sub_name;
   public String  sub_address;
   public int     sub_credit;

   // Тут много чего пропущено...
   public void update(int amount)
      {
      sub_credit = amount;
      }
   }
```

Дескрипторы размещения

Одним из наиболее заметных преимуществ модели EJB является заложенный в ней механизм разделения задач между бизнес-логикой и кодом инфраструктуры — пример применения тактики «семантическая связность». Такого рода разделение связано с тем, что элементы EJB в основном содержат бизнес-логику, в то время как EJB-контейнеры отвечают за вопросы среды и инфраструктуры — транзакции, управление жизненным циклом bean и безопасность. В результате bean-компоненты упрощаются — перечисленные сложности в их коде не отражены.

Для оповещения контейнера о службах, которые ему требуются, bean пользуется дескриптором размещения. Так называется связанный с EJB XML-документ. При размещении bean в контейнере последний считывает дескриптор размещения и из его содержания узнает задачи по обработке транзакций, устойчивости (для beans-сущностей) и управлению доступом. Таким образом, дескриптор есть декларативный механизм обработки перечисленных аспектов и в этом смысле являет собой пример применения тактики «откладывание времени связывания».

Прелесть этого механизма в том, что один и тот же EJB-компонент, в зависимости от прикладной среды, размещается с разными дескрипторами. Если существенное значение имеет безопасность, компонент обозначает свои потребности по части управления доступом. Если безопасность не имеет значения, управление доступом не специфицируется. В обоих случаях применяется одинаковый код EJB.

Дескриптор размещения составляется в предопределенном формате, которого все EJB-совместимые beans должны придерживаться, а EJB-совместимые серверы — уметь считывать. Формат этот задается в виде XML-шаблона DTD (Document

Type Definition, описание типа документа). Дескриптор размещения описывает тип bean (сеансовый или сущность), классы для обеспечения интерфейсов `remote` и `home`, а также bean-класс. Кроме того, он определяет транзактные свойства всех методов bean, роли безопасности, имеющие право доступа к тому или иному методу (управление доступом), а также устанавливает один из вариантов устойчивости — ее автоматическое обеспечение средствами контейнера или явное выполнение в коде bean.

Дескриптор размещения рассмотренного ранее bean **Broker** представлен в листинге 16.7. Помимо описания свойств, этот дескриптор идентифицирует данный bean как сеансовый без состояния и сообщает о том, что для исполнения каждого из его методов необходима транзакция с контейнерным управлением (на листинге эти свойства для простоты выделены полужирным начертанием). К примеру, стоит лишь ввести в поле **<session-type>** XML значение **stateful**, и контейнер будет управлять bean совершенно по-другому. В листинге 16.8 приводится дескриптор размещения bean-сущности **Account**. В дополнение к уже упомянутым свойствам размещения этот дескриптор сообщает контейнеру следующие сведения:

- для beans данного типа требуется управление устойчивостью;
- местонахождение источника данных JDBC для базы данных;
- отображение отдельных первичных ключей и элементов данных между базой данных и bean-сущностью.

Листинг 16.7. Описание размещения bean Broker

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>EntityStock.BrokerHome</ejb-name>
         <home>j2ee.entitystock.BrokerHome</home>
         <remote>j2ee.entitystock.Broker</remote>
         <ejb-class>j2ee.entitystock.BrokerBean</ejb-class>
          <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>EntityStock.BrokerHome</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

Листинг 16.8. Описание размещения bean-сущности Account

```
<ejb-jar>
   <enterprise-beans>
      <entity>
         <ejb-name>EntityStock.AccountHome</ejb-name>
         <home>j2ee.entitystock.AccountHome</home>
         <remote>j2ee.entitystock.Account</remote>
```

```
        <ejb-class>j2ee.entitystock.AccountBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>j2ee.entitystock.AccountPK</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-field>
           <field-name>sub_accno</field-name>
        </cmp-field>
        <cmp-field>
           <field-name>sub_name</field-name>
        </cmp-field>
        <cmp-field>
           <field-name>sub_address</field-name>
        </cmp-field>
        <cmp-field>
           <field-name>sub_credit</field-name>
        </cmp-field>
        <resource-ref>
           <res-ref-name>jdbc/sqlStock_nkPool</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
        </resource-ref>
     </entity>
  </enterprise-beans>
  <assembly-descriptor>
     <container-transaction>
        <method>
           <ejb-name>EntityStock.AccountHome</ejb-name>
           <method-intf>Remote</method-intf>
           <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
     </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

В табл. 16.2 мы перечислили требования по атрибутам качества, которые компания Sun предъявляет к J2EE. В табл. 16.5 мы обозначаем механизм реализации этих требований посредством дескрипторов размещения.

Таблица 16.5. Реализация требований, предъявляемых компанией Sun к атрибутам качества J2EE, при помощи дескрипторов размещения

Задача	Реализация	Тактики
Переносимость	Возможность разработки общей кодовой базы для разных целевых платформ; возможность настройки в период размещения нескольких версий дескриптора размещения, предназначенных для разных целевых платформ — благодаря этой возможности разрабатываемый прикладной компонент становится переносимым с одной целевой среды на другую	Семантическая связность, обобщение модулей, конфигурационные файлы
Возможность построения	Дескрипторы размещения обеспечивают разделение задач по разработке кода обобщения модуля	Семантическая связность, конфигурационные файлы и настройки размещения

Задача	Реализация	Тактики
Умеренная специфичность	Представление дескрипторов размещения в формате XML, с одной стороны, способствует содержательной стандартизации кодирования конфигурации, а с другой — предоставляет производителям возможность расширения дескрипторов размещения индивидуальными характеристиками	Конфигурационные файлы, обобщение модуля
Прозрачность	Детали дескриптора размещения, к которым обращаются серверные компоненты, незаметны для их клиентов	Введение посредника реализации

16.4. Решения по размещению системы

Все, о чем мы говорили выше, справедливо по отношению к спецификациям J2EE/EJB по версии Sun. Существует также ряд проблем реализации, которые архитектор в процессе размещения системы J2EE/EJB должен решить. Компонентная модель EJB — это мощный инструмент конструирования серверных приложений. Поначалу взаимосвязи между различными частями кода сбивают с толку, однако, если разобраться и наработать некоторый опыт, становится ясно, что конструировать EJB-приложения не так уж и сложно. Впрочем, несмотря на удобство кодирования, есть ряд сложностей, которые нужно уметь разрешать.

- Модель EJB позволяет комбинировать компоненты приложения в соответствии с разными архитектурными образцами. Какие из них лучше и что значит «лучше» в контексте конкретного приложения?
- Механизм взаимодействия beans с контейнером довольно сложен, а кроме того, он оказывает значительное воздействие на производительность приложения. Кроме того, серверные контейнеры EJB отличаются один от другого — таким образом, довольно значимыми становятся такие аспекты жизненного цикла разработки приложения, как отбор продуктов и конфигурирование конкретного продукта.

В завершающем разделе мы представим обзор основных проектных проблем, возникающих в процессе создания архитектуры и конструирования сверхмасштабируемых EJB-приложений.

Управление состоянием — старая проблема в новом контексте

При разработке серверного звена EJB встает вопрос о принятии одной из двух моделей обслуживания: без запоминания состояния (stateless) и с запоминанием состояния (stateful), реализуемых сеансовыми beans без состояния и сеансовыми beans с запоминанием состояния соответственно.

Для примера рассмотрим сетевой книжный магазин. В версии с запоминанием состояния EJB можно заставить сохранять детальные данные о покупателе

и управлять товарами, который он помещает в интерактивную корзину. Таким образом, EJB будет хранить состояние, связанное с посещением сайта данным покупателем. Поскольку bean сохраняет состояние диалога, клиент может за ним не следить. EJB отслеживает потенциальные покупки и при вызове метода подтверждения осуществляет их пакетную обработку.

В целях оптимизации использования ограниченных ресурсов системной памяти невостребованные клиентами сеансовые beans с запоминанием состояния пассивируются — иначе говоря, состояние диалога bean записывается на вспомогательное запоминающее устройство (как правило, на диск), а экземпляр, хранящийся в памяти, удаляется. Ссылка на bean у клиента при пассивации остается активной и готовой к употреблению. Когда клиент вызывает метод в пассивированном bean, контейнер запускает новый экземпляр bean и наполняет его состояние информацией, записанной на вспомогательное ЗУ.

Стратегия пассивации оказывает значительное влияние на масштабируемость. Если для обслуживания отдельных клиентов постоянно пассивируются и активируются многочисленные экземпляры сеансовых beans с запоминанием состояния, издержки производительности приложения могут очень сильно возрасти.

Сеансовый bean без состояния не принимает на себя обязательства по сохранению клиентского состояния диалога. При каждом запросе на обслуживание клиенту приходится информировать сервер о состоянии сеанса — в частности, сообщать свои данные и содержимое корзины; такая необходимость связана с тем, что на каждый запрос контейнер может выделять разные экземпляры сеансового bean без состояния. В рамках простой модели обслуживания без запоминания состояния это единственно возможный вариант. Механизм использования сеансовых beans с запоминанием и без запоминания состояния изображен на рис. 16.6.

Обобщенно преимущества сеансовых beans без состояния можно сформулировать следующим образом:

- Издержки производительности, связанные с пассивацией и активацией сеансовых beans, а следовательно, и с проведением дорогостоящих операций записи и считывания на диск, отсутствуют.
- Благодаря динамической маршрутизации запросов эти самые запросы перенаправляются на наименее загруженный сервер.
- В случае недоступности экземпляра сеанса запрос оперативно перенаправляется другому экземпляру.

Единственный недостаток методики незапоминания состояния состоит в том, что при каждом запросе между клиентом и EJB передается информации больше, чем обычно. Если предположить, что передаваемые данные не слишком объемны, то, скорее всего, степень масштабируемости системы в случае применения сеансового bean без запоминания состояния все-таки окажется выше.

Beans-сущности — за и против

Один из наиболее распространенных образцов проектирования EJB предполагает введение оболочки сеансового bean, которая, с одной стороны, выставляет для клиента службы, а с другой, стараясь выполнить клиентский запрос, обращается к инкапсулированным в bean-сущности бизнес-данным. Здесь налицо классиче-

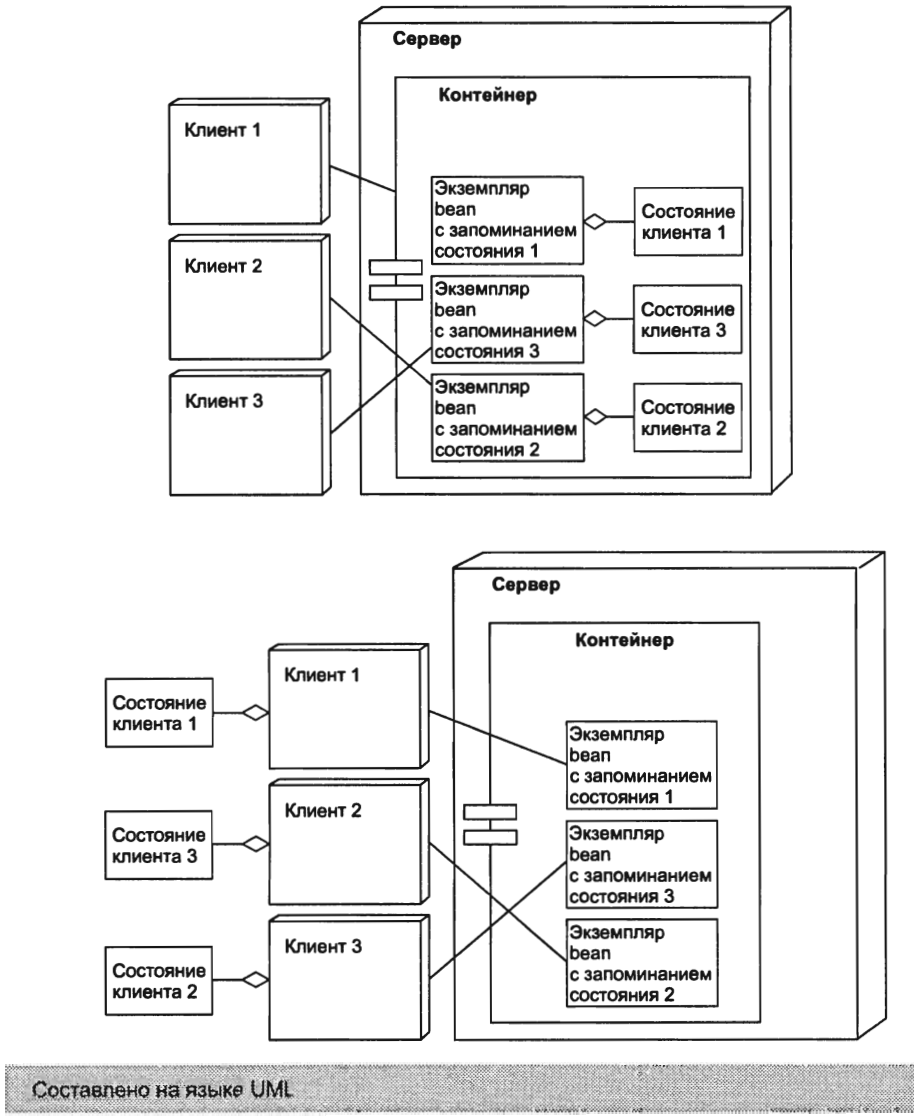


Рис. 16.6. Статическое связывание клиентов с экземплярами сеансового bean с запоминанием состояния и динамическое связывание с экземплярами сеансового bean без запоминания состояния

ская объектно-ориентированная модель программирования. Бизнес-данные, традиционно представленные в базе данных в реляционном формате, теперь инкапсулируются в объектно-ориентированном формате (beans-сущности). Определяемые в большом количестве для beans-сущностей методы get и set упрощают доступ к такого рода данным для сеансовых beans. Кроме того, если в beans-сущностях задействуется устойчивость с контейнерным управлением, необходимость в детальной разработке кода доступа к базе данных отпадает.

Существует, впрочем, риск значительного снижения производительности. По результатам тестирований, для типичной системы электронной коммерции с 85 % транзакций на чтение и 15 % на обновление архитектура приложений с beans-сущностями достигает лишь половины от уровня пропускной способности, демонстрируемого архитектурой исключительно с сеансовыми beans. Причины снижения производительности перечислены ниже.

- В то время как сеансовые beans напрямую обращаются к хранящемуся в базе данных бизнес-объекту, употребление beans-сущностей связано с введением дополнительного уровня косвенности. Автоматическая оптимизация вызовов beans-сущностей (сеансовыми beans) до локального вызова доступна отнюдь не во всех реализациях контейнеров. Если она не осуществляется, дополнительный RMI-вызов оказывается слишком дорогим.
- Дорогостоящим может оказаться и управление жизненным циклом beans-сущностей на дополнительном уровне. Активация подразумевает по меньшей мере одну операцию считывания из базы данных/с диска, а пассивация — одну операцию записи в базу данных/на диск.
- В транзакции участвует больше beans.

Как обычно, решение о том, компенсируют ли преимущества beans-сущностей вероятное снижение пропускной способности системы, остается за архитектором приложения.

Проблемы распределения и масштабирования

Чем популярнее становятся корпоративные веб-системы, тем чаще предприятия сталкиваются с неспособностью своих серверных систем обрабатывать возрастающие объемы входящего интернет-трафика. Существует всего два способа повышения вычислительной мощности серверного звена.

- Масштабирование «вверх» (*вертикальное масштабирование*) подразумевает введение новых вычислительных и системных ресурсов — например, расширение памяти на отдельной машине. Эта разновидность масштабирования применима в том случае, если во внутренней архитектуре сервера приложений нет «врожденных» узких мест. Если это так, то при введении дополнительных системных ресурсов и повышении мощности процессора программное обеспечение сервера приложений должно полностью освоить новые ресурсы и тем самым повысить пропускную способность системы.
- Масштабирование «вширь» (*горизонтальное масштабирование*) компенсирует неизменность производительности существующей машины распределением сервера приложений между несколькими машинами. При предоставлении приложению дополнительных машин суммарные системные ресурсы и производительность повышаются.

Считается, что из-за усложнения конфигурации и управления системой проводить горизонтальное масштабирование проблематично. Кроме того, при горизонтальном масштабировании на сервере приложений должны быть предусмотрены механизмы выравнивания нагрузки, обеспечивающие полное освоение клиентами дополнительных ресурсов на всех машинах.

С другой стороны, системы, рассредоточенные на нескольких машинах, отличаются рядом преимуществ по сравнению с системами, работающими на одной машине.

- *Дополнительное резервирование.* Если на одной машине произойдет отказ, ее функции возьмут на себя другие. Среди причин отказов — нарушение электроснабжения, сетевые сбои, аварийные отказы операционной системы, отказы сервера приложений и даже ошибки в прикладном коде.
- *Экономическая эффективность.* Соотношение «цена/производительность» для сети машин с относительно небольшой производительностью иногда превосходит соответствующий показатель для одной высокопроизводительной машины.

Во многих прикладных продуктах предусматриваются службы кластеризации, обеспечивающие горизонтальное масштабирование приложений. Здесь опять же следует подчеркнуть, что приложения кластеризации сильно отличаются друг от друга и архитектору следует обращать на эти различия пристальное внимание.

Распределенные транзакции

Многие EJB-серверы способны координировать транзакции с участием нескольких объектов из разных процессов в распределенной системе. При конструировании корпоративных систем возможность обработки распределенных транзакций посредством двухэтапного протокола фиксации зачастую играет очень важную роль.

Архитектор, проектирующий систему EJB, должен принять тщательно продуманное решение относительно потребности в распределенных транзакциях как таковых. Дело в том, что управление ими связано со значительными издержками, которые возрастают пропорционально количеству участников транзакций. Если координировать транзакции между несколькими диспетчерами ресурсов (или базами данных) не требуется, значит, не нужен и двухэтапный протокол фиксации.

Кроме того, при координации транзакций и исполнении процессов фиксации через сеть — от сервера или контейнера, с одной стороны, и внешним процессом управления транзакциями — с другой, — проходит ряд удаленных вызовов. Если реализация распределенных транзакций конкретного EJB-сервера предполагает координацию транзакций посредством дополнительных удаленных вызовов, весьма вероятно значительное замедление системы EJB и ухудшение общей масштабируемости системы.

Опыт применения разных реализаций J2EE и механизмов управления объектной технологией свидетельствует о высокой вариативности показателей производительности управления распределенными транзакциями. Таким образом, архитекторы приложений должны хорошо ориентироваться во всех вариантах конфигурации и размещения, возможных в рамках данной службы транзакций.

Организация пула ресурсов

В распределенной системе следует уделять особое внимание управлению ресурсами приложения — в частности, соединениями с базами данных и сокетами.

Методика организации пула ресурсов основывается на том обстоятельстве, что постоянный монопольный доступ к ресурсам требуется далеко не всем клиентам. В контексте EJB не каждому bean требуется монопольное соединение с базой данных. Значительно более эффективной представляется такая конфигурация системы, при которой соединения с базой данных организуются в рамках пула и многократно распределяются между различными клиентскими транзакциями.

В случае применения пула соединений с базой данных конечных соединений оказывается значительно меньше, чем в размещенной системе EJB-компонентов. Поскольку создание соединений с базой данных и управление ими — операции весьма дорогостоящие, такая архитектура повышает общую масштабируемость приложения. Более того, за счет того что необходимость постоянного восстановления соединений отсутствует, повышается производительность приложения.

Методика организации пула применима и к другим ресурсам — в частности, к сокетным соединениям и потокам. Организация пула компонентов предполагает, что выделять для каждого клиента специализированный ресурс не требуется. Среди стандартных настраиваемых параметров — контейнерные потоки, экземпляры сеансовых beans, емкость кэша bean-сущности и размер пула соединений с базой данных. Путем продуманной настройки этих параметров сокращается время отклика и повышается общая пропускная способность системы.

Зависимость от производительности виртуальной машины Java

Регулировка производительности любого Java-приложения в значительной степени зависит от JVM. Следовательно, если задача состоит в том, чтобы разработать и разместить высокопроизводительное серверное приложение EJB, не обойтись без планирования ряда операций, связанных с конфигурированием JVM и регулировкой производительности.

Одной из важнейших настроек является емкость кучи JVM. Кучей (heap) называется репозитарий объектов Java и свободной памяти. Когда в куче JVM заканчивается свободная память, исполнение в ней приостанавливается вплоть до того момента, когда алгоритм сбора мусора освободит невостребованные участки памяти. Блокирование прикладного кода в ходе сбора мусора — это очевидный удар по производительности. Все операции, проводящиеся в приложении EJB на стороне сервера, в эти периоды останавливаются.

Если емкость кучи высока, сбор мусора проводится довольно редко; впрочем, когда он все же случается, то занимает значительное время, в течение которого могут быть нарушены нормальные системные операции. Операция сбора мусора способна снизить скорость (а в отдельных случаях — полностью остановить) обработки данных на сервере, который в таком случае будет восприниматься как медленный и нереактивный.

Для оптимальной настройки емкости кучи JVM необходимо проследить за проводимыми на серверной машине операциями разбиения на страницы. Разбиение на страницы существенно снижает производительность, а для того чтобы

избежать проведения подобных операций на серверах приложений, следует увеличить емкость кучи JVM согласно потребностям конкретного приложения. Другое решение предполагает контроль над сборщиком мусора при помощи настройки компилятора `-gcverbose`. Если возможен добавочный сбор мусора, его, как правило, лучше включать.

16.5. Заключение

К созданию многозвенной архитектуры J2EE компанию Sun Microsystems подтолкнули ее собственные коммерческие потребности. Они, в свою очередь, формировались под влиянием опыта применения модели CORBA и конкурентного давления со стороны других специальных моделей распределенного программирования — в частности, COM+ от Microsoft. В состав J2EE входит серверный компонентный каркас для конструирования серверных Java-приложений корпоративного уровня, и называется он системой корпоративных JavaBeans (Enterprise JavaBeans).

Спецификация J2EE/EJB постоянно расширяется. В настоящее время в ней предусмотрены готовые к употреблению службы транзакций, безопасности, устойчивости и управления ресурсами. Все эти службы позволяют прикладным программистам, имеющим дело с J2EE/EJB, не заботясь о низкоуровневых деталях распределения, обращать более серьезное внимание на разработку бизнес-логики. Переносимость в J2EE/EJB реализуется за счет применения общеупотребительного, переносимого языка (Java) и точных контрактов между компонентами. Производительность и масштабируемость производительности достигаются за счет распределения приложений между многочисленными процессорами (горизонтальное масштабирование), сеансовых beans без запоминания состояния, организации пула ресурсов и ряда других механизмов.

Несмотря на кажущуюся простоту модели программирования J2EE/EJB, многие архитектурные решения прикладного уровня нужно тщательно продумывать. Для получения проектного решения, оптимального в контексте требований по атрибутам качества, необходимы анализ и сравнение различных компромиссных архитектурных решений.

16.6. Дополнительная литература

Сведения об архитектуре и спецификации J2EE/EJB крайне многочисленны. Стоит зайти на раздел сайта Sun Microsystems, посвященный этой технологии (http://java.sun.com/j2ee), — на нем предлагается удобное пособие для новичков, заинтересовавшихся J2EE, ряд официальных документов и собственно спецификация J2EE/EJB. Существует множество оживленных форумов, посвященных архитектуре и технологическому пространству J2EE, — в частности, форум, организованный компанией Middleware (http://www.theserverside.com).

16.7. Дискуссионные вопросы

1. В качестве дополнения к компонентной модели EJB версии 2.0 определены «управляемые сообщениями beans». Под этим именем кроются корпоративные beans, позволяющие приложениям J2EE проводить асинхронную обработку сообщений. Назовите несколько вариантов применения такого компонента. Какие возможности в контексте корпоративной архитектуры открывают управляемые сообщениями beans?
2. Многие методики в рамках спецификации J2EE/EJB, по существу, лишь реализуют тактику «введение посредника». Попробуйте найти как можно больше подобных случаев.
3. Обратимся к конкретному примеру компании CelsiusTech, изложенному в главе 15. Как вы считаете, подходит ли инфраструктура J2EE/EJB для реализации систем, выпускаемых этой компанией? Аргументируйте ваш ответ.

Глава 17
Архитектура Luther. Конкретный пример мобильных приложений на основе архитектуры J2EE

(в соавторстве с Таней Басс, Джеймсом Беком, Келли Долан, Куйвей Ли, Андреасом Леером, Ричардом Мартином, Уильямом Россом, Тобиасом Вайсхауплем и Грегори Железником)[1]

Бог — в деталях.

Людвиг Мис ван дер Роэ

Функции рабочих, управляющих или обслуживающих крупные транспортные средства (например, танки и самолеты) или объекты промышленной инфраструктуры (в частности, мосты и нефтяные вышки), крайне трудно компьютеризировать. Поскольку объекты, на которых они трудятся, отличаются большими размерами, работать с ними приходится непосредственно на месте их расположения, на открытом воздухе или в специальных помещениях — ни одна из этих категорий не предусматривает возможности применения настольных компьютеров. Таким образом, решения, как правило, сводятся к использованию беспроводной инфраструктуры и ручных/бесконтактных вычислительных устройств.

Компания Inmedius, занимающаяся техническим обеспечением квалифицированных операционистов и рабочих по обслуживанию, появилась в 1995 году как продолжение проекта по переносным компьютерам Университета Карнеги-Меллон (см. врезку «История переносных компьютеров»). Начав с разработки единичных продуктов, компания вскоре осознала необходимость в общих решениях с возможностью оперативной подгонки под потребности заказчиков.

---

[1] Все перечисленные специалисты работают в корпорации Inmedius (Питтсбург, США).

Функции, выполняемые квалифицированными рабочими, требуют серьезного конторского обеспечения. Помимо сбора отчетов о проблемах необходимо проводить планирование ремонта, учета и повторного заказа устанавливаемых запчастей, а также заказа журнала техобслуживания. Подобного рода технологическое управление предполагает взаимодействие квалифицированного рабочего с конторским сотрудником, сидящим за настольным компьютером.

Имея в виду создать общую инфраструктуру разработки специализированных решений задач по обслуживанию, компания Inmedius спроектировала архитектуру Luther. Основанная на корпоративной архитектуре Java 2 (Java 2 Enterprise Edition, J2EE), она являет собой приложение универсальной архитектуры J2EE/EJB (см. главу 16) в среде, в которой конечный пользователь, действующий в рамках беспроводной сети, работает с устройством ограниченных возможностей на входе-выходе и/или ограниченных вычислительных возможностей.

ИСТОРИЯ ПЕРЕНОСНЫХ КОМПЬЮТЕРОВ

Есть мнение, согласно которому первым переносным компьютером стали наручные часы. Изобретенные около 1900 года, они поначалу оказались не способны составить конкуренцию часам карманным. Действительно, зачем носить часы на запястье, если их можно положить в карман и легко оттуда достать? Зачем покупать новые часы, если старые исправно работают? Все изменилось во время Первой мировой войны, когда для того чтобы солдаты британской армии могли согласованно вступать в атаку, не выпуская из рук оружие, им стали выдавать наручные часы. После этого такие часы как символ поддержки «парней в окопах» стали необычайно популярны в самой Британии. Сегодня карманные часы встречаются все реже.

К началу 1990-х годов технология вышла на такой уровень, при котором стали реальностью переносные цифровые полнофункциональные вычислительные устройства. Исследования по применению такого рода устройств, помимо прочих учреждений, проводились в группе по переносным компьютерам (Wearable Group) при Университете Карнеги-Меллон (руководитель группы — Дэн Севёрек (Dan Siewiorek)). Переносные компьютеры в этой группе рассматривались как средство исполнения производственных функций в местах содержания крупных транспортных средств (например, самолетов) — иначе говоря, на открытом воздухе или в крупных помещениях наподобие ангаров и железнодорожных депо.

Поскольку акцент ставился на производственные функции, требовалось в первую очередь обеспечить удобство использования и сложность проектного решения, достаточную для достижения поставленных задач. Студенты проектировали и конструировали компьютеры, а сотрудники группы по переносным компьютерам проводили с ними эксперименты на реальных рабочих местах. Эксперименты оказались успешными, на переносные компьютеры сформировался спрос, и для его удовлетворения была организована компания Inmedius.

В то же самое время в информационной лаборатории (Media Laboratory) при Массачусетском технологическом институте действовала другая группа, сотрудники которой называли себя «borgs». Переносный компьютер они рассматривали как потребительский продукт, призванный изменить жизнь своего владельца, и старались доказать справедливость этого постулата на собственном примере. Они пребывали в постоянном поиске новых вариантов применения такого компьютера и приложений, ориентированных на запоминание данных. В частности, им принадлежат идеи использования проводимости человеческой кожи как средства поддержания сетевых соединений и обмена цифровыми визитными карточками в момент рукопожатия двух владельцев переносных компьютеров.

К концу 1990-х годов две группы объединили свои усилия в деле утверждения переносных компьютеров как академической дисциплины. Крупные коммерческие предприятия начали проявлять интерес к предложениям разных компаний по поставкам переносных компьютеров и головных дисплеев. Сегодня, по мере того как аппаратные средства стремятся к большей компактности, а программные средства наращивают сложность (об этом, в частности, пойдет речь в настоящей главе), переносные компьютеры имеют все шансы получить серьезное распространение.

— LJB

17.1. Связь с архитектурно-экономическим циклом

На рис. 17.1 изображен архитектурно-экономический цикл (Architecture Business Cycle, ABC) компании Inmedius и архитектуры Luther. Среди задач по качеству на рисунке обозначены возможность повторного использования, производительность, модифицируемость, гибкость потребительских устройств, а также возможность взаимодействия со стандартными коммерческими инфраструктурами; все они, как обычно, обусловлены коммерческими задачами заказчика и конечного пользователя.

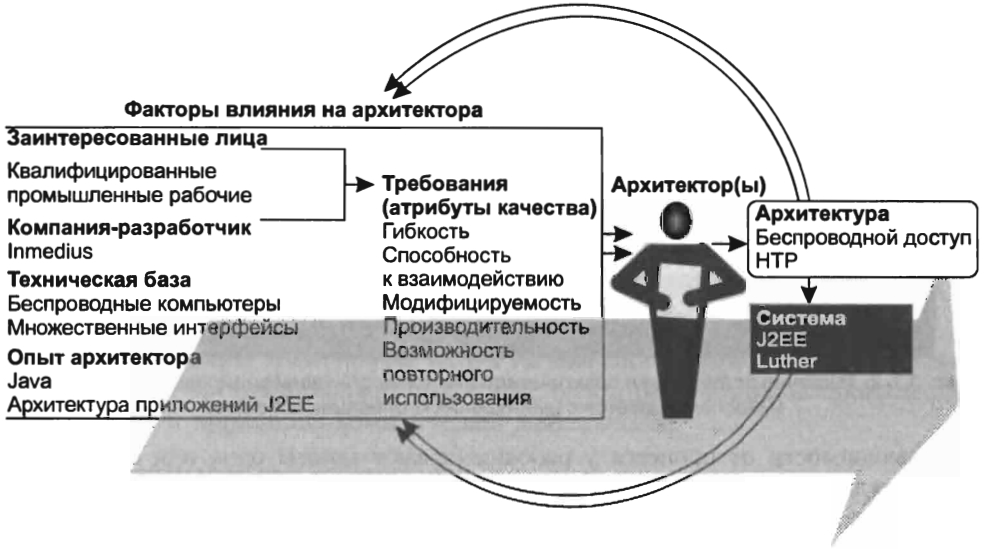


Рис. 17.1. Архитектурно-экономический цикл компании Inmedius и архитектуры Luther

Факторы влияния на архитектуру

В нижеследующих разделах мы разберем обстоятельства, оказавшие на архитектуру Luther то или иное воздействие.

Конечные пользователи

Компания Inmedius специализируется на компьютеризации функций квалифицированных рабочих. На рис. 17.2 изображен рабочий с одной из аппаратных конфигураций, поддерживаемых приложениями Luther. Рабочий осуществляет производственный процесс, этапы которого выводятся на головной дисплей. Компьютер, в котором в качестве основного входного устройства используется диск управления, закреплен на груди рабочего. Производственный процесс описан в инструкции, хранящейся на конторских компьютерах; по мере исполнения его этапов (иногда их насчитывается более 500) на переносной компьютер передаются соответствующие страницы инструкции. Посредством системы рабочий сообщает

в контору о выполнении отдельных компонентов процесса. К примеру, установив деталь, он указывает ее инвентарный номер; после этого конторские сотрудники вносят соответствующие изменения в опись и проводят аналитические действия, связанные с контролем качества.

Рис. 17.2. Решение от Inmedius в практической деятельности квалифицированного рабочего. (Перепечатывается с разрешения корпорации Inmedius.)

В зависимости от процесса у рабочих бывают заняты одна или обе руки — следовательно, вводить в компьютер данные они не могут. Более того, в некоторых случаях во время выполнения своих функций рабочие находятся в движении.

Поскольку требования по мобильности и количеству свободных (для ввода данных) рук, выставляемые разными заказчиками, непостоянны, каждому процессу потенциально соответствует индивидуальная аппаратная конфигурация.

Компания-разработчик

Если на основе архитектуры Luther можно будет разрабатывать сложные корпоративные решения, затрачивая на это меньше времени, чем на производство единичных систем, компания Inmedius добьется значительного конкурентного преимущества. Для этого компании необходимо кардинально повысить оперативность выхода корпоративных решений на рынок. Сохранить конкурентоспособность на целевых рынках Inmedius сможет лишь в том случае, если сократит циклы разработки таких решений до нескольких месяцев.

Разработка решений должна проводиться группами из нескольких десятков инженеров в кратчайшие сроки и с учетом экономических соображений. Необходимым условием удовлетворения потребностей заказчика является высокое качество продукции. Кроме того, артефакты поставляемого программного обеспечения должны предусматривать легкость проведения модификаций — любые

коррективы и улучшения должны вноситься сотрудниками Inmedius без особого труда и не должны подрывать целостность архитектуры первоначального решения.

Технологическая база

В архитектуре Luther отражены новейшие на момент ее создания разработки в области программного и аппаратного обеспечения. В главе 16 мы рассматривали архитектуру J2EE, на основе которой для коммерческих организаций разрабатываются корпоративные решения. Она успешно удовлетворяет выставленное к Luther требование относительно возможности взаимодействия с конторскими процессами. Кроме того, J2EE упрощает задачу группировки предметно-ориентированных прикладных функций в компоненты, допускающие повторное использование и разного рода сочетания.

Помимо совершенствования программного обеспечения на Luther оказали воздействие новые аппаратные технологии — в частности, это касается необходимости поддержки компактных беспроводных компьютеров с речевым вводом и головными дисплеями с высоким разрешением. С другой стороны, в разных условиях требуются разные типы устройств, для каждого из которых характерен индивидуальный набор возможностей. Отсюда — требование к гибкости Luther в отношении типов поддерживаемых пользовательских интерфейсов.

Влияние архитектуры на компанию

Результаты влияния архитектуры Luther на компанию-разработчика проявляются в областях организационной структуры, опыта разработчиков программного обеспечения и принципов коммерческой деятельности.

Организационная структура

Еще до создания архитектуры Luther компания Inmedius позиционировала себя как поставщика решений; каждое из них создавалось для конкретного заказчика в качестве единичного приложения. С организационной точки зрения крупнейшей инженерной группой в компании была группа решений (Solution Group). Во время разработки архитектуры Luther была создана группа продуктов (Product Group, с группой разработки компонентов (Component Development Group) в своем составе), отвечавшая за конструирование и сопровождение предметно-ориентированных компонентов, из которых сотрудники группы решений составляли клиентские продукты. Группа продуктов (Product Group) специализируется на развитии общих направлений работы на рынке, в то время как группа решений (Solution Group) занимается проектированием конкретных приложений для конкретных заказчиков. Это яркий пример двухэлементной организационной структуры, характерной для компаний, действующих в рамках линеек продуктов. Структура эта описана в главе 14 и проиллюстрирована конкретным примером компании CelsiusTech в главе 15.

Опыт разработчиков программного обеспечения

Еще до начала работы над архитектурой Luther в компании Inmedius работали опытные и квалифицированные разработчики программных продуктов. Тем не

менее, когда речь зашла о создании Luther, им пришлось приспособиться к ряду новых критериев, в частности:

- изучить язык программирования Java;
- получить сертификаты Sun Java Programmer;
- изучить архитектуру приложений J2EE;
- изучить механизмы пакетирования возможностей в рамках спецификаций J2EE и EJB;
- научиться пользоваться Java-сервлетами и JavaServer Pages;
- обучиться применению различных служб J2EE, предусмотренных в реализации этой спецификации.

Принципы коммерческой деятельности

Архитектура Luther оказала на ведение компанией Inmedius коммерческой деятельности очень заметное влияние. Как мы уже говорили (см. главу 14), для разработки единичных решений требуются обширные ресурсы. Неоптимальное расходование ресурсов и ограниченное восприятие, характерные для процесса разработки единичных систем, препятствуют складыванию глобального мышления. Переход к линейке продуктов на основе архитектуры Luther позволил компании Inmedius сосредоточиться на построении линеек, отказавшись от ориентации на индивидуальные системы. Более того, равно как и в случае с CelsiusTech, Inmedius удалось выйти на новые рынки, которые оказались своеобразным обобщением — как в коммерческом, так и в техническом смысле — тех сегментов рынка, на которых компания присутствовала ранее.

17.2. Требования и атрибуты качества

При проектировании архитектуры Luther перед сотрудниками компании была поставлена задача удовлетворить два набора взаимодополняющих требований. Первый из этих наборов регулирует вопросы, связанные с конструированием приложений — конкретнее, корпоративных приложений для рабочих «полевого обслуживания». Эти требования видимы для заказчиков, поскольку их неудовлетворение приводит к невозможности исполнения приложений согласно высказанным пожеланиям, — к примеру, в таком случае приложение, которое, в общем, функционирует нормально, может в беспроводной сети работать плохо. Второй набор требований связан с внедрением общей архитектуры для многочисленных продуктов. Соблюдение этих требований сокращает продолжительность интеграции, способствует более оперативному выходу новых продуктов на рынок, повышает качество продукции, упрощает внедрение новых технологий и обеспечивает непротиворечивость продуктов.

Обобщенно, требования можно разделить на шесть категорий:

- беспроводной доступ;
- пользовательский интерфейс;
- типы устройств;

- существующие процедуры, бизнес-процессы и системы;
- конструирование приложений;
- распределенные вычисления.

Беспроводной доступ

По мере выполнения своих функций рабочие постоянно находятся в движении среди разного рода механизмов, множества людей. Кроме того, их деятельность связана с определенным риском. Для взаимодействия с конторскими системами применяемые рабочими устройства должны предусматривать возможность обращения к удаленным серверам и источникам данных. При этом связь с локальной сетью по наземному каналу исключается. Поскольку заказчики Inmedius выставляют к продуктам разные требования, беспроводные сети могут характеризоваться разной пропускной способностью и готовностью.

Пользовательский интерфейс

Своими конкурентными преимуществами компания Inmedius, помимо прочего, обязана высокоточным пользовательским интерфейсам. Они позволяют рабочим сосредоточиться на выполняемых функциях, не уделяя слишком много внимания взаимодействию с устройством доступа. Устройства различаются по площади экрана, и на каждом из них архитектура Luther должна обеспечивать отображение осмысленной информации. Совершенно не обязательно для этого конструировать единый пользовательский интерфейс, а затем адаптировать его к каждому из типов устройств. В Luther применяется другое решение — архитектура обеспечивает оперативное конструирование интерфейсов, которые фильтруют, синтезируют и объединяют информацию так, что она адекватно отображается на конкретном устройстве и облегчает работу пользователя.

Разнообразие устройств

Рабочие, обслуживающие технику непосредственно в местах ее расположения, пользуются самыми разными устройствами. Среди них нет ни одного, которое подходило бы для всех рабочих функций в заданных условиях; зато у всех есть те или иные ограничения, которые архитектура Luther должна в обязательном порядке учитывать. От сотрудников Inmedius, таким образом, требовалось выработать решения, ориентированные на повышение производительности для всех возможных устройств — в частности, для следующих:

- персональных информационных устройств (personal data assistant, PDA): Palm Pilot, Handspring Visor, vTech Helio, IBM WorkPad, Apple's Newton и MessagePad 2000;
- карманных компьютеров (Pocket PC) наподобие Compaq iPAQ, Casio EM500, HP Jornada и Phillips Nino;
- ручных перьевых планшетов на основе операционной системы Windows CE: Fujitsu Stylistic, PenCentra и Siemens SIMpad SL4;
- ручных компьютеров на основе Windows CE с пером и клавиатурой: Vadem Clio, HP Jornada 700 series, NEC MobilePro, Intermec 6651 Pen Tablet Computer, Melard Sidearm и др.;

- переносных компьютеров наподобие Xybernaut MA-IV, семейства продуктов Via и устройства Spot от компании Pittsburgh Digital Greenhouse.

В зависимости от класса устройства характеризуются разной емкостью дискового пространства, скоростями процессора и набором устройств ввода. Все эти характеристики оказывают значительное влияние на стиль взаимодействия пользователя с устройством. К примеру, переносные компьютеры способны обеспечить рабочего, находящегося на месте выполнения своих функций, практически всеми возможностями настольного компьютера — исполняемые клиентские приложения на них не менее сложны. В таком случае пользователи могут свободно выбирать устройства ввода — в частности, в их распоряжении клавиатура, голосовой ввод, перо и некоторые специализированные устройства.

В устройствах класса PDA скорость процессора, необходимая емкость дискового пространства и набор устройств ввода в значительной степени ограничены — соответственно ограничены и механизмы взаимодействия пользователей с этими устройствами. Тем не менее в отдельных контекстах устройства класса PDA оказываются очень полезными и значительно упрощают выполнение рабочими их функций. Архитектура Luther учитывает разнообразие стилей пользовательского взаимодействия с устройствами, которые ограничиваются возможностями различных аппаратных средств.

Существующие процедуры, бизнес-процессы и системы

Функционирование большинства предприятий отнюдь не исчерпывается деятельностью рабочих на местах. Информацию, которые они собирают, необходимо сохранять на конторских машинах. Инструкции они в некоторых случаях получают извне; кроме того, существует множество приложений, ориентированных на выполнение традиционных бизнес-процессов.

Реагируя на перечисленные потребности, архитектура Luther предусматривает интеграцию своих функций с привычными для рабочих процедурами и процессами. Она позволяет размещать приложения на серверах и в базах данных различных производителей. Кроме того, она упрощает интеграцию приложений с унаследованными системами.

Конструирование приложений

Стремление к ускорению процесса конструирования приложений было одной из основных предпосылок к созданию архитектуры Luther. Эта задача состоит из нескольких аспектов, включая следующие:

- Поощрение повторного использования программных продуктов и упрощение взаимодействия приложений. Эти аспекты позволяют экономить ограниченные ресурсы и не изобретать колесо заново.
- Внедрение в корпоративные функции (например, в последовательность технологических операций) стратегии «лучше построить, чем покупать».
- Предоставление стабильной платформы для внедрения новых характеристик и инновационных технологий в масштабах приложений — в частности,

механизма определения местоположения (location sensing), автоматического обнаружения и идентификации близлежащих физических объектов и служб, усложненных характеристик пользовательского интерфейса наподобие синтетических опросов.

Распределенные вычисления

Архитектура Luther снабжает разработчиков корпоративных приложений каркасом и инфраструктурой, которые выравнивают различия в возможностях клиентских устройств и обогащают серверы приложений рядом характеристик, связанных с распределенными вычислениями.

- *Масштабируемость.* Серверный каркас Luther обеспечивает масштабируемость, не оказывая при этом негативного воздействия на производительность. Иначе говоря, введение предметно-ориентированных компонентов, вне зависимости от их количества, никоим образом не влияет на производительность прикладных приложений и не приводит к необходимости реконструкции клиентских приложений. Кроме того, от клиентских приложений требуется удобство реконфигурирования, направленного на освоение новых возможностей. Каркас позволяет приложениям обнаруживать новые возможности и путем динамической автореконфигурации вводить их в действие.
- *Выравнивание нагрузки.* В распределенной среде архитектура Luther обеспечивает выравнивание нагрузки. Бóльшая часть вычислительных операций в приложениях на ее основе выполняется на стороне сервера, после чего результаты отправляются клиентам. Если клиентских обращений к конкретному серверу становится слишком много, инфраструктура сервера приложений обнаруживает увеличение нагрузки и переводит операции обработки на компоненты сервера приложений, расположенные на других серверных узлах предприятия. Корпоративная среда приложений аналогичным образом обнаруживает отказы узлов и с целью возобновления исполнения переводит операции в другие приложения. Выравнивание нагрузки в обоих случаях должно проходить незаметно для пользователя. В первом случае также требуется прозрачность операции по отношению к клиентскому приложению.
- *Независимость от местоположения.* Для того чтобы выравнивание нагрузки стало возможным, требуется распределение предметно-ориентированных прикладных средств. Инструменты, направленные на реализацию этой задачи, заложены в архитектуре Luther. Для динамического изменения местоположения приложения должны быть независимыми от него.
- *Переносимость.* Среды корпоративных приложений, по определению, содержат в своем составе разнородные серверные аппаратные платформы. Каркас архитектуры Luther предусматривает исполнение программных средств на самых разных платформах, обеспечивая тем самым работоспособность корпоративных приложений.

17.3. Архитектурное решение

В ответ на предъявленные требования было принято принципиальное архитектурное решение, согласно которому архитектуру Luther следовало сконструировать поверх архитектуры J2EE. У этого решения есть ряд преимуществ.

- Архитектура J2EE существует во множестве коммерческих версий от разных производителей. Компоненты, которые потенциально могли оказаться полезными для архитектуры Luther, разрабатывались повсеместно (в частности, речь идет о компонентах технологического управления).
- Протокол HTTP, расположенный на верхнем уровне стека TCP/IP, становился основным средством передачи данных. Сам же стек TCP/IP поддерживался различными коммерческими беспроводными стандартами, в том числе IEEE 802.11b. В рамках инфраструктуры беспроводной локальной сети любого веб-клиента можно сделать мобильным. Кроме того, большинство устройств, которые архитектура Luther должна была поддерживать, в свою очередь поддерживают HTTP.
- Принятое решение предусматривает отделение пользовательского интерфейса и позволяет реализовать *парадигму пользовательского опыта* (user experience paradigm). Парадигма эта предполагает, что компьютер станет для рабочего одним из инструментов, причем инструмент этот будет носить ненавязчивый характер. Компьютер, согласно упомянутой парадигме, должен стать естественным расширением инструментов рабочего. При этом он приносит выгоды по части производительности — как самому рабочему, так и компании, в которой тот работает.

  Далее, эта парадигма декларирует необходимость разработки нескольких представлений корпоративного приложения — по одному для конкретной роли конкретного рабочего. Каждое представление подгоняется под роль, увеличивает производительность и удовлетворенность выполняемой работой и, наконец, фильтрует, объединяет, синтезирует и отображает соответствующую данной роли информацию. Каждое конкретное представление предусматривает использование наиболее подходящих к данной роли устройств ввода.

  К примеру, если клавиатурный ввод в конкретном случае не подходит, его можно заменить речевым вводом. Если условия работы слишком шумные, имеет смысл обратиться к специализированному устройству ввода — например, *диску управления* (dial). Поворачивая диск (который закрепляется на униформе рабочего — см. рис. 17.2), пользователь переходит по ссылкам, нажимает кнопки, выбирает положения переключателей и манипулирует другими средствами пользовательского интерфейса, предусмотренными в клиентском приложении, тем самым активизируя их. В средней части диска находится клавиша Enter, предназначенная для нажатия кнопок, переходу по ссылкам и тому подобных действий. Диск применяется и в самых трудных условиях — с ним можно работать даже в толстых перчатках.

  В главе 5 мы рассматривали тактику реализации практичности под названием «отделение пользовательского интерфейса». В рамках архитектуры

Luther эта тактика обеспечивает гибкость, позволяющую менять пользовательский интерфейс, адаптировать его к различным устройствам и потребностям. Здесь, между прочим, проявляется другой атрибут качества — модифицируемость. Это очередное подтверждение нашего постулата о том, что некоторые тактики позволяют реализовывать сразу несколько атрибутов качества.

- Рассматриваемое решение позволяет разделять и абстрагировать источники данных. В зависимости от конкретной ситуации использования устройства пользователю могут потребоваться фильтрация, объединение, синтез и отображение данных, исходящих из разнородных источников. Среди этих источников — системы управления базами данных, а также унаследованные приложения, сконструированные на основе корпоративных систем планирования ресурсов с инкапсуляцией корпоративных данных. Сотрудники Inmedius поняли, что при условии абстрагирования и отделения источников данных от приложений, которые к ним обращаются, а также при предоставлении для них четко очерченных стандартных интерфейсов приложения гарантированно придерживаются заданных абстракций и, таким образом, предусматривают возможность повторного использования. Кроме того, некоторые интерфейсы приняты в качестве стандартных (к таким интерфейсам, в частности, относится JDBC/ODBC). Они позволяют трактовать источники данных как абстрактные компоненты, которые при желании можно ввести или извлечь из корпоративного приложения.

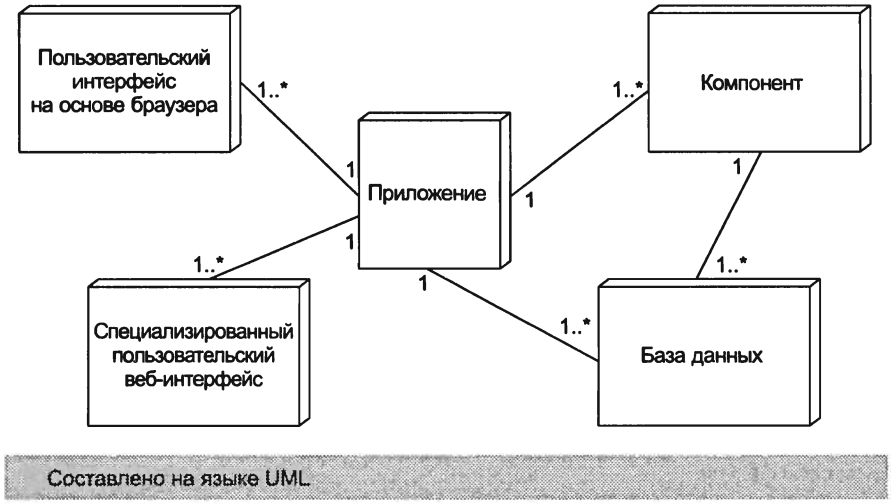


Рис. 17.3. Представление размещения приложения Luther

На рис. 17.3 изображена схема взаимодействия приложения Luther со средой (элементы J2EE на ней не показаны — отображение приложения на J2EE мы еще обсудим). Во-первых, обратите внимание на отношение (*n:1:m*) между пользовательскими интерфейсами, приложениями и тем, что сотрудники Inmedius называют компонентами, — иначе говоря, стандартными блоками конструирования

функциональности приложения. Приложение Luther является тонким; значительная часть его бизнес-логики собрана из существующих компонентов и не привязана к какому-либо конкретному пользовательскому интерфейсу. Обобщенно, прикладной код состоит из следующих элементов:

- определение состояния сеанса и управление этим состоянием;
- прикладная (то есть не предусматривающая повторного использования) бизнес-логика;
- логика, которая делегирует бизнес-запросы соответствующей последовательности вызовов компонентных методов.

В этом приложении нет основного метода — зато в нем есть интерфейс прикладного программирования (application programming interface, API), выражающий характеристики и функции, которые приложение представляет пользовательским интерфейсам. Пользовательский интерфейс независим от приложения. Он может включать любое подмножество характеристик, применимое к целевому интерфейсу устройства. К примеру, при создании интерфейса для устройства с микрофоном и динамиками, но без дисплея, интерфейс не предлагает средств графики.

Теперь попробуем провести углубленный анализ трех основных элементов, показанных на рис. 17.3: пользовательского интерфейса, приложения и компонентов.

Пользовательский интерфейс

Стратегию разработки пользовательских интерфейсов в рамках архитектуры Luther можно изобразить следующим образом. В первую очередь, специалисты в предметной области, специалисты по когнитивной психологии и дизайнеры ведут переговоры с клиентом с целью выявить задачи и роли разных рабочих, условия их труда, а также необходимые характеристики интерфейсов для предполагаемых устройств доступа. Далее, исходя из установленных ограничений, они пытаются воссоздать пользовательский опыт, выражая его в виде последовательности кадров, снимков экрана или макета. Все это делается для того, чтобы в процессе проектирования пользовательский опыт моделировался с должным качеством и точностью. Это очень важная задача, поскольку приложение призвано дополнять традиционные технологические операции и быть естественным расширением рабочей среды. Следовательно, воссоздание среды пользователей делегируется тем, кто знает ее лучше других: специалистам в предметной области, которые прекрасно представляют себе задачи и условия труда; специалистам по когнитивной психологии, разбирающимся в том, как люди думают, анализируют и осмысливают информацию; дизайнерам, умеющим представлять информацию в доступном и привлекательном виде.

На следующем этапе артефакты процесса проектирования — последовательность кадров, снимки экрана и макет — оперативно переносятся на работающие пользовательские интерфейсы реальных устройств. В этом контексте от архитектуры требуется обеспечение интеграции различных вариантов пользовательского опыта. Интеграция должна быть быстрой, ориентированной на максимизацию

повторного использования универсальных программных элементов и при всем при этом способствовать сохранению целостности и точности отражения пользовательского опыта.

Задача переноса пользовательского опыта на реальный пользовательский интерфейс осложняется рядом факторов. Во-первых, необходима поддержка многочисленных клиентских устройств. Среди них — самые разнообразные мобильные устройства с разными размерами экрана, операционными системами и устройствами ввода. Пользовательский интерфейс, который оптимально подходит для употребления на настольном компьютере, значительно ограничивается в возможностях при уменьшении экрана, сокращении ресурсов памяти и сужении функциональности — все эти явления наблюдаются на мобильных устройствах. В частности, некоторые подобного рода устройства не предусматривают поддержки клавиатуры и мыши; соответственно, пользовательские интерфейсы, которые требуют наличия этих средств ввода, становятся совершенно бесполезными. Второй фактор влияния — это технологические ограничения. К примеру, некоторые типы взаимодействия пользователя с системой и отображения информации довольно громоздки и потому при передаче по протоколу HTTP серьезно снижают производительность.

В конце концов, для каждого конкретного приложения могут существовать разные клиентские устройства и пользовательские интерфейсы. Таким образом, от программной архитектуры требуется значительная гибкость — иначе она просто не сможет справиться с многочисленными разнородными клиентами. На рис. 17.4 и 17.5 изображены два типа реализации пользовательского интерфейса, предусмотренные архитектурой Luther, а именно: клиент на основе браузера (рис. 17.4) и специализированный веб-клиент (рис. 17.5). На рис. 17.6 уточняется представление, приведенное на рис. 17.3; там же изображается структура каждого из упомянутых типов.

Клиенты на основе браузера

Клиенты пользовательского интерфейса на основе браузера напрямую соответствуют предусмотренным в архитектуре J2EE клиентам на основе браузера. При этом они не ограничены веб-браузерами. В равной степени они поддерживают другие формы разметки — в частности, язык разметки беспроводных систем (Wireless Markup Language, WML) для беспроводного прикладного протокола (Wireless Application Protocol, WAP), применяемого в мобильных телефонах. Несмотря на различия по части языка разметки, для предоставления контента используются все те же механизмы, а именно: сочетание сервлетов и JavaServer Pages (JSP).

Обмен информации осуществляется клиентом на основе браузера посредством стандартных методов (то есть коммерческих веб-браузеров на стороне клиента, HTTP посредством стека TCP/IP в качестве сетевого протокола и JSP и Java-сервлетов на стороне сервера) в универсальных форматах данных (гипертекстовые документы и таблицы стилей). Для того чтобы сделать клиента тонким, бóльшая часть логики представления реализуется на сервере. Таким образом, возможности создания интерфейса, обладающего переносимостью на браузеры разных производителей и версий, значительно увеличивается.

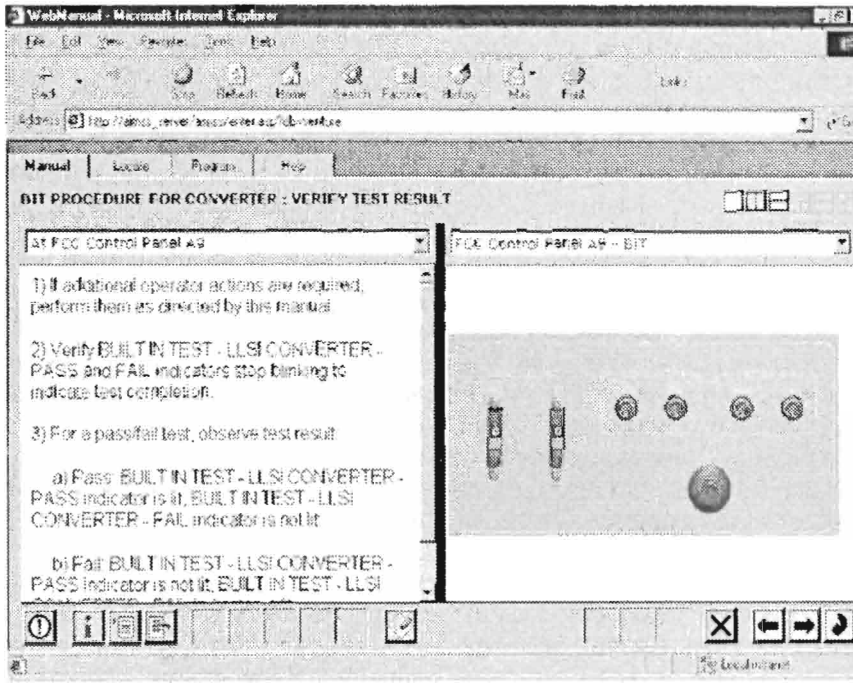


Рис. 17.4. Интерфейс, отображаемый в браузере и предназначенный для выполнения процедуры технического обслуживания

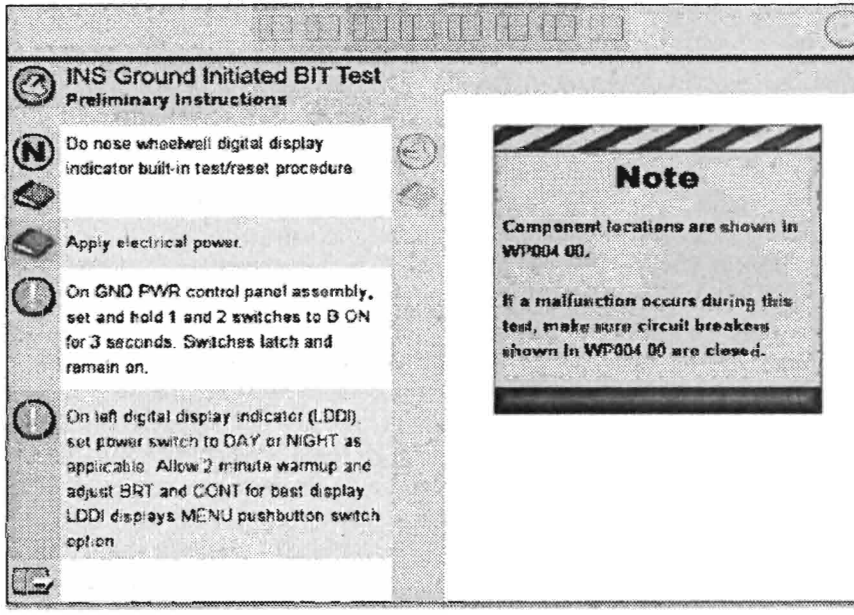


Рис. 17.5. Специализированный пользовательский веб-интерфейс

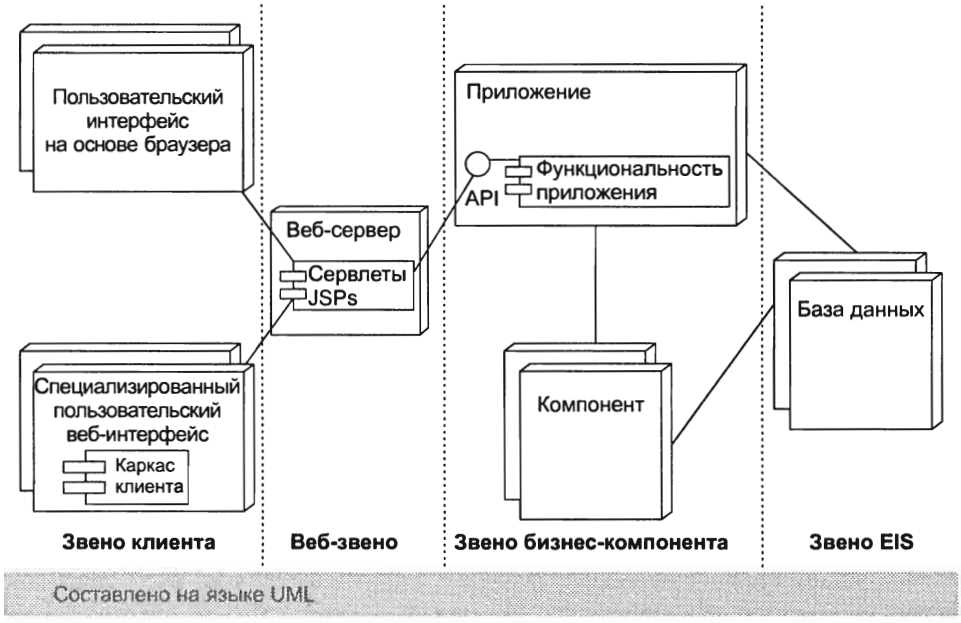


Рис. 17.6. Пользовательский интерфейс в представлении «компонент и соединитель», наложенном на представление размещения

Клиенты на основе браузера в основном ориентируются на:

- устройства с поддержкой браузеров и традиционных устройств ввода (перьев, клавиатуры и мыши);
- приложения, которые отображают информацию, легко представляемую на языках разметки, обеспечивают удобство вывода в браузере и расширения посредством сменных модулей.

Первоначально браузеры разрабатывались в расчете на настольные компьютеры, которые должны были стать оптимальным целевым устройством. На сегодняшний день поддержка браузеров присутствует в мобильных устройствах.

Применение интерфейсов на основе браузера характеризуется некоторыми ограничениями. К примеру, с точки зрения проектирования они не всегда способствуют оптимальному применению ограниченных ресурсов, таких как площадь экрана. Кроме того, типы взаимодействия с пользователем, поддерживаемые моделью браузера, весьма ограничены и построены на основе цикла «запрос/отклик» по протоколу HTTP. Кроме того, интерфейсы на основе браузера подходят не для всех мобильных устройств, поскольку для некоторых из них браузеров не существует вообще. Если же они и реализованы, то поддержка важнейших характеристик (таких, как фреймы, графика и JavaScript) в них может отсутствовать.

Специальные веб-клиенты

Специальные пользовательские веб-интерфейсы довольно сложны. Они не демонстрируют прямого соответствия специальному клиенту, который в рамках

архитектуры J2EE рассматривается как автономная программа, реализующая всю логику представления и взаимодействующая напрямую с бизнес-логикой (в частности, с элементами EJB) посредством удаленного вызова методов (remote method invocation, RMI) по межброкерному протоколу Интернета (Internet inter-ORB protocol, IIOP). Специальные веб-клиенты действительно представляют собой автономные программы, однако, в отличие от специальных клиентов по версии J2EE, для связи с сервером и взаимодействия с сущностями веб-звена (наподобие сервлетов и JSP) они используют протокол HTTP, демонстрируя тем самым схожесть с клиентами на основе браузера. Специальные веб-клиенты пишутся в собственной среде разработки в расчете на конкретное устройство или класс устройств. Являясь автономной программой, пользовательский интерфейс рассматриваемого типа предоставляет проектировщикам значительную свободу по части поддерживаемых моделей взаимодействия с пользователем. Отсюда — возможность оптимального освоения ресурсов наподобие пространства экрана. Возможность эта достигается за счет повышения стоимости разработки.

Как показано на рис. 17.6, в архитектуре Luther предпринята попытка минимизировать объем собственного кода при написании специального веб-клиента. Задача эта решается с помощью клиентского каркаса, поддерживающего интерфейсы данного типа. По сути, этот каркас стандартизирует элементы, задействованные в разных приложениях, — в частности, такие элементы, как управление сеансом, аутентификация, а также средства создания и упорядочивания логики представления со стороны клиента, веб-контейнера или с обеих этих сторон. Таким образом, клиент представляет собой тонкую автономную программу, которая создает или систематизирует свойственные пользовательским интерфейсам средства. Кроме того, она реализует некоторую часть логики приложения — в частности, проверку входных данных на правильность и сортировку информации в табличном представлении. Равно как и в случае с клиентами на основе браузера, большая часть логики представления реализуется в веб-звене, а конкретнее — в компонентах, управляемых клиентским каркасом.

Специальные веб-клиенты обладают множеством преимуществ над другими типами специализированных пользовательских интерфейсов. Во-первых, они тонкие — другими словами, в сравнении с толстым клиентом (то есть специальной программой, в которой вся логика представления реализуется в клиентском звене) они компактнее, удобнее в сопровождении, а кроме того, способствуют переносимости между разными устройствами. Во-вторых, для взаимодействия с веб-звеном они обращаются к протоколу HTTP и этим отличаются от специальных клиентов по версии J2EE, которые используют удаленный вызов методов (RMI) по протоколу IIOP. Таким образом, они лучше подходят для реализаций, написанных не на Java, и применения в беспроводных сетях.

Создавать специальные собственные пользовательские интерфейсы для каждого приложения на каждом устройстве — слишком дорого, даже если этих устройств не так уж и много. Повышения стоимости разработки удается избежать путем разделения устройств на классы, исходя из их характеристик. Для каждого из таких классов изложенными выше методами конструируется высокоточный интерфейс. Большую часть задач по реализации этого интерфейса в рамках данного класса устройств выполняется клиентским каркасом. Аналогичным обра-

зом, при реализации значительной части логики представления в веб-звене клиентские устройства, приписанные к одному классу, могут обращаться к клиентскому каркасу и совместно использовать определенную часть собственной реализации. Наконец, клиентский каркас предусматривает характеристики, позволяющие устройству объявлять характеристики своих интерфейсов. Эта информация доступна логике представления в веб-звене, и, таким образом, перед доставкой контента клиенту с ним можно проводить ряд действий по адаптации.

Приложения

Приложения в архитектуре Luther ответственны за объединение системы в единую функциональную сущность и предоставление интерфейсов прикладного программирования (APIs), ориентированных на взаимодействие с системой. Обращаясь к API, пользовательские интерфейсы предлагают их характеристики конечному пользователю.

Приложения могут находиться между произвольным числом пользовательских интерфейсов и компонентов. Таким образом, приложения связывают m компонентов и выставляют напоказ совокупную «прикладную» функциональность n пользовательских интерфейсов. Такие приложения «не распознают» пользовательские интерфейсы — иначе говоря, они способны выставлять напоказ такую функциональность, которую может использовать любой пользовательский интерфейс. В зависимости от ситуации любой пользовательский интерфейс выставляет напоказ всю функциональность или какое-то из ее подмножеств. К примеру, пользовательский интерфейс, исполняемый на мобильном клиенте (например, на устройстве с операционной системой Windows CE), не может предлагать возможности администрирования, которые были бы вполне приемлемы в рамках версии этого интерфейса для настольной системы. Идея в том, чтобы выставлять напоказ все функции, которые могут быть выполнены в данной системе. Решение о том, какие из них предоставить пользователю и как это сделать, каждый пользовательский интерфейс принимает самостоятельно.

Согласно требованию об оперативности разработки и размещения, приложение разрабатывается в как можно более тонком виде. Реализуется эта задача путем делегирования большей части бизнес-задач компонентам (об этом мы поговорим в следующем разделе). Критерий, исходя из которого прикладной код передается компонентам, очень прост. Сформулировать его можно так: «Предусматривает ли данная функциональность возможность повторного использования?». Если предусматривает, то ее необходимо обобщить, тем самым увеличив шансы на многократное применение, и, соответственно, реализовать как компонент. Если же вероятность повторного использования данного функционального блока низка, значит, его стоит встроить в приложение.

Ниже перечислены основные элементы приложений.

- *Интерфейс прикладного программирования.* Представляет собой фасад функций, которые система выставляет напоказ пользовательским интерфейсам. Обратите внимание на то, что данные, передаваемые через интерфейс прикладного программирования, не адаптированы к конкретному представлению (например, HTML) и носят универсальный характер (например, XML).

- *Состояние сеанса.* После инициализации, происходящей в результате аутентификации пользователя, информация о состоянии сеанса сохраняется вплоть до закрытия клиентской программы. В архитектуре J2EE управление сеансом упрощается за счет того, что контейнеры не только хранят и извлекают информацию о состоянии сеанса, но также предусматривают аутентификацию и авторизацию. Приложение определяет данные, которые необходимо сохранять между запросами, и для их хранения и извлечения делает соответствующие вызовы.
- *Прикладная бизнес-логика.* Это любая логика, которая является уникальной для приложения и не предусматривает повторного использования в других приложениях.
- *Делегирование компонентам.* Это код, предназначенный для делегирования задач компонентам. Как правило, реализуется посредством образца проектирования под названием «бизнес-делегат» (Business Delegate[1]).

Эти элементы появляются в результате применения тактики «прогнозирование ожидаемых изменений» и родственны тактике реализации модифицируемости под названием «отделение пользовательского интерфейса».

При создании нового пользовательского интерфейса совершенно не обязательно вносить изменения в прикладной уровень или в компонент. Интеграция новой реализации компонента в систему проходит независимо от прикладного уровня и пользовательских интерфейсов. При введении в систему новой функциональности она дополняется новым компонентом. В прикладной уровень вводятся соответствующие методы интерфейса прикладного программирования (API), а в пользовательский интерфейс вводятся (или не вводятся) новые характеристики, обеспечивающие выставление напоказ новых функций.

Компоненты

Компонент по определению выражает практику многократного применения. Стратегия, таким образом, состоит в том, чтобы сформировать из компонентов библиотеку, на основе которой можно будет быстро и без особого труда конструировать приложения, реализующие специализированные решения для конкретных заказчиков. В любой библиотеке содержатся *базовые компоненты* (core components), относящиеся к клиентскому и серверному каркасам; *предметно-ориентирован-*

[1] Бизнес-делегат выступает в роли фасада компонента — он обнаруживает его и предоставляет его функции остальным элементам приложения. Таким образом, бизнес-делегат должен знать, как обнаружить компонент, как к нему обратиться и как скрыть соответствующие детали от остального приложения. К примеру, если компонент реализован в виде EJB, бизнес-делегат выполняет предусмотренные Java-интерфейсом именования и каталогов (Java Naming Directory Interface, JNDI) операции поиска и тем самым сужает удаленный (remote) интерфейс EJB. При этом факт реализации данного компонента в виде EJB скрывается. Приложение *не* отвечает за управление жизненным циклом компонента — эту функцию выполняют контейнеры J2EE. С другой стороны, в связи с делегированием перед приложением стоит задача выбора компонента(ов). Кроме того, в приложении присутствует логика, регулирующая взаимодействие и взаимоотношения между компонентами. Она очевидным образом принадлежит приложению. При условии следования этому правилу реализация компонентов упрощается, а их взаимозависимость уменьшается.

ные компоненты (domain-specific components), отражающие специфику конкретной предметной области (например, сопровождения, восстановления или перестроения); и *компоненты общих возможностей* (generalized capability components; другое название — *вспомогательные компоненты* (utility components)), с помощью которых производится окончательная комплектация функциональности приложений (например, реализация авторизации и управление пользователями).

Стратегия Inmedius предусматривает развитие базовых, предметно-ориентированных компонентов и компонентов общих возможностей — как для каркаса архитектуры Luther, так и для конкретных предметных областей, в которых работают заказчики компании. Таким образом, разработка приложений превращается в процесс формирования бизнес-логики, которая обеспечивает объединение необходимого набора компонентов, выражающих разные возможности, в специализированные решения для заказчиков.

Отдельный и очень важный вопрос в конструировании линеек программных продуктов связан с созданием общих компонентов. Он выражает масштабное применение тактики реализации модифицируемости под названием «общие абстрактные службы» — в данном случае в целях производства новых решений.

Проектирование компонентов

Согласно стратегии проектирования компонентов, интерфейсы прикладного программирования и поведение компонентов должны при малейшей возможности разрабатываться на основе стандартов проектирования. К примеру, рассматриваемый чуть позже компонент технологического управления являет собой конкретизацию спецификации, составленной группой по технологическому управлению (Workflow Management Coalition), которая определяет функциональность и поведение при организации технологических процессов. Такая стратегия проектирования позволяет Inmedius заменять собственные компоненты компонентами любых других производителей и придерживаться тех же, что и они, спецификаций возможностей. Она способствует расширению библиотеки компонентов Inmedius.

Разделение возможностей

Иногда оказывается, что компонент возможности, необходимый для разработки конкретного приложения, в библиотеке отсутствует. В таком случае необходимо принять решение о том, как проектировать и реализовывать соответствующую возможность — как часть собственно приложения либо как новый компонент, допускающий повторное использование.

Основная проектная дилемма связана с тем, чем является данная возможность — частью бизнес-логики приложения, применимой исключительно к данному решению, или конкретизацией более общей возможности с перспективой повторного использования в других приложениях.

Пакетирование компонентов

Все без исключения приложения в рамках Luther обращаются к среде J2EE и ее службам. Учитывая это ограничение, компоненты в данной среде можно пакетировать в виде элементов EJB, компонентов Java beans, отдельных библиотек классов

Java, апплетов, сервлетов или произвольного сочетания перечисленных элементов. Другими словами, компонент, не являясь синонимом EJB, тем не менее допускает разные варианты пакетирования.

Выбор стратегии пакетирования конкретной возможности обусловливается набором применяемых служб J2EE и решениями, уравновешивающими ряд значимых факторов (частоты межобъектного взаимодействия, местонахождения объектных экземпляров, а также необходимости применения разного рода служб J2EE — в частности, службы транзакций и службы устойчивости состояния объекта в продолжение нескольких пользовательских сеансов). К примеру, обмен данными с элементами EJB осуществляется путем удаленного вызова методов (RMI) — довольно тяжеловесного механизма взаимодействия. Некоторые контейнеры J2EE допускают оптимизацию обмена данными с элементами EJB (который в таком случае производится путем локального вызова методов); однако делается это лишь в том случае, если обмен данными происходит в рамках одной виртуальной машины Java (Java Virtual Machine, JVM). Тем не менее, поскольку оптимизация не входит в число требований к контейнерам J2EE, обмен данными между элементами EJB всегда связан с риском серьезного увеличения издержек, в связи с чем к этому вопросу следует относиться с осторожностью — если, конечно, вас заботит производительность. В качестве альтернативного решения можно создать библиотеку классов Java, которая позволяет избежать удаленного вызова методов, а следовательно, и связанных с этим механизмом издержек. Впрочем, в таком случае компонентам приходится брать на себя дополнительные обязанности, которые обычно выполняются контейнерами, — в частности, связанные с созданием и удалением экземпляров компонентов.

Все связанные с компонентом объекты на протяжении сеанса должны быть доступны пользователю. В этот период они могут изменяться, однако в масштабах нескольких сеансов данные должны оставаться устойчивыми и непротиворечивыми — следовательно, компоненты часто прибегают к транзакциям. К одним и тем же объектам могут обращаться сразу несколько пользователей, причем зачастую они обращаются к ним с общей целью; такие ситуации должны быть корректно обработаны. Помимо прочего, поддержка транзакций делает возможным восстановление после отказа — благодаря сохраняющейся непротиворечивости базы данных эта процедура упрощается.

Как мы говорили в главе 16, модель EJB поддерживает несколько типов beans: beans-сущности (entity beans), сеансовые beans (session beans) и сеансовые beans без состояния (stateless session beans). Эти типы beans ориентированы на разные формы бизнес-логики и, соответственно, по-разному обрабатываются контейнерами. К примеру, bean-сущность допускает самостоятельное управление устойчивостью, которое осуществляется посредством поддерживаемых контейнером обратных вызовов (так называемая устойчивость под управлением beans, или самоуправляемая устойчивость); кроме того, эти задачи может выполнять сам контейнер (так называемое контейнерное управление устойчивостью). В любом случае, эти операции связаны со значительными непроизводительными издержками, которые ограничивают практическое применение beans-сущностей долговременными бизнес-объектами, характеризующимися крупноблочным доступом к данным.

Что делает контейнер J2EE?

Приложения выказывают потребность в самых разных возможностях — например, в поддержке транзакций, безопасности и выравнивании нагрузки. Эти возможности очень сложны (многие корпорации даже организуют специальные подразделения, предназначенные исключительно для их обеспечения) и находятся за рамками любого конкретного приложения или прикладной области. Один из основных мотивов, которым руководствовалась компания Inmedius, принимая решение о создании архитектуры Luther на основе J2EE, состоял в том, что подобные характеристики реализованы в коммерческих общедоступных совместимых с J2EE контейнерах, и, следовательно, Inmedius не пришлось реализовывать их самостоятельно.

Многие из этих возможностей можно конфигурировать для конкретного элемента EJB в период размещения приложения. Есть и другая альтернатива — они могут присутствовать в контейнерах J2EE и носить прозрачный характер. В любом случае, разработчикам EJB не приходится встраивать вызовы к ним непосредственно в код; следовательно, их можно без труда сконфигурировать с учетом потребностей любого конкретного заказчика. Тем самым упрощается процесс создания компонентов EJB, независимых от конкретного приложения, и гарантируется их исполнение в рамках любых совместимых с J2EE контейнеров.

- EJB-контейнер осуществляет поддержку транзакций как декларативно, так и программно. Программное взаимодействие с контейнером позволяет разработчику обеспечить низкоуровневую, жестко закодированную поддержку транзакций EJB. Кроме того, с помощью дескриптора размещения разработчик может декларативно специфицировать поведение методов в рамках транзакций. Отсюда — возможность дифференциации поведения транзакций в разных приложениях, причем для этого реализация и конфигурирование непосредственно в коде EJB не требуются.
- J2EE предусматривает интегрированную модель безопасности, распространяющуюся как на веб, так и на EJB-контейнеры. Подобно поддержке транзакций, характеристики безопасности применяются как декларативно, так и программно. В случае написания методов с определениями полномочий, необходимых для их исполнения, разработчик может задать в дескрипторе размещения перечень пользователей (или группу пользователей), которым разрешается доступ к методам. Кроме того, с помощью записей в дескрипторе размещения можно ассоциировать с методами права доступа. Это, опять же, позволяет приложению определять произвольные полномочия в рамках методов компонентов, причем сами компоненты при этом переписывать не требуется.
- Наконец, EJB-контейнер обеспечивает прозрачное выравнивание нагрузки. Создание и управление экземплярами EJB осуществляется контейнером в период исполнения — при необходимости они создаются, активируются, пассивируются и удаляются. Если обращений к данному элементу EJB давно не поступало, его можно пассивировать — при этом его данные сохраняются в постоянном устройстве хранения, а экземпляр удаляется из памяти. Таким способом контейнер фактически выравнивает нагрузку

в масштабе всех экземпляров в контейнере, регулирует потребление ресурсов и оптимизирует производительность системы.

Что делает разработчик компонентов?

Разработчик компонента создает для него клиентское представление, или интерфейс прикладного программирования, а также реализацию компонента. В отношении простых элементов EJB его действия ограничиваются написанием всего лишь трех классов: home-интерфейса, remote-интерфейса и класса реализации.

Кроме того, разработчик компонента составляет определение типов данных, выставляемых напоказ клиентам через интерфейс прикладного программирования. Они реализуются как дополнительные классы и во многих случаях принимают форму объектов значения (value objects), которые посредством интерфейса прикладного программирования передаются элементу EJB и извлекаются из него.

Пример повторно используемого компонента: компонент технологического управления

В этом разделе мы рассмотрим один из повторно используемых компонентов возможностей, разработанных для библиотеки компонентов Inmedius, связанные с ним проблемы и принятые решения. Компонент технологического управления — крупнейший из всех созданных на сегодняшний момент компонентов архитектуры — иллюстрирует процесс конструирования, пакетирования и введения в архитектуру Luther компонентов общих возможностей.

Логическое обоснование проекта

Основная обязанность компонента технологического управления — обеспечивать для клиента возможность моделирования технологического процесса и переноса через него цифровых артефактов. Кроме того, компонент позволяет клиентам определять ресурсы и распределять их между операциями технологического процесса. Естественно, от компонента требуется возможность повторного использования и расширяемость, а значит, он должен предоставлять общие возможности технологического процесса; предусматривать четкую и универсальную модель функционирования обращающихся к нему приложений; быть невосприимчивым к цифровым артефактам, проходящим через те или иные вариации технологического процесса. Для создания полнофункционального компонента технологического процесса требуются сложные идиомы — в частности, ветвление, слияние и зацикливание. Как правило, для реализации возможности технологического управления требуются недюжинные ресурсы.

Компания Inmedius неожиданно столкнулась с непредвиденной трудностью. Приложения, разрабатываемые в рамках Luther, явственно демонстрировали серьезную потребность в возможностях технологического управления, однако их комплексной реализации препятствовали следующие факторы:

- размер и сложность комплексного технологического управления выходили за рамки имевшихся у компании ресурсов;

- комплексная возможность технологического управления не относилась ни к базовым коммерческим задачам, ни к центральной компетенции компании;
- значительно более комплексные решения были уже созданы другими компаниями.

В связи с этим было принято стратегическое решение об образовании альянсов с компаниями, сумевшими создать средства технологического управления для приложений J2EE в виде компонентов. Однако прежде Inmedius предстояло реализовать подмножество этих средств, обеспечивающих решения.

Таким образом, стратегия заключалась в том, чтобы спроектировать компонент, который впоследствии можно было бы без труда заменить более комплексным компонентом от сторонней компании. Отсюда сформировалась потребность в стандартном интерфейсе компонента технологического управления. Обратите внимание на то, как в данном случае ведет себя архитектурно-экономический цикл. Проект архитектуры Luther способствовало открытию новой коммерческой перспективы (связанной с технологическим управлением), что побудило компанию принять явное решение о вхождении или, вернее, отказе от вхождения на данный сегмент рынка. Руководство Inmedius решило, что рассматриваемая область выходит за рамки базовой компетенции компании.

Группой по технологическому управлению (Workflow Management Coalition) был разработан ряд спецификаций функций и поведения, связанных с технологическим управлением, который впоследствии был признан сообществом. При построении компонентов архитекторы Inmedius исходили из этих спецификаций, однако реализовывали только ту функциональность, которая была необходима для разрабатываемых приложений.

Принятая стратегия сбалансировала знания и опыт сообщества разработчиков средств технологического управления и его деятельность в целом. Поскольку коммерческие задачи и отношения между объектами в этом сообществе уже определены, Inmedius не пришлось разрабатывать их с чистого листа. Кроме того, придерживаясь спецификаций группы по технологическому управлению, компания Inmedius получила возможность замены своего компонента технологического управления аналогичным компонентом от другого производителя. Операция эта, проводившаяся в случае, если конкретному заказчику требовалась функциональность, которую компонент Inmedius оказывался неспособным предоставить, не требовала особых усилий.

Две спецификации группы по технологическому управлению описывали два основополагающих элемента: модель технологического управления и представление ее экземпляров для периода прогона (рис. 17.7). Модель технологического управления состоит из одного или нескольких процессов, в каждом из которых определены действия, переходы между этими действиями и все ресурсы-участники. В каждом процессе задействован диспетчер, который управляет всеми экземплярами периода прогона для конкретного процесса; каждый экземпляр периода прогона хранит информацию о состоянии — в частности, о том, какие действия уже завершены, какие активны и кто их назначил, — а также контекстные данные, необходимые компоненту технологического управления для принятия решений по активному процессу.

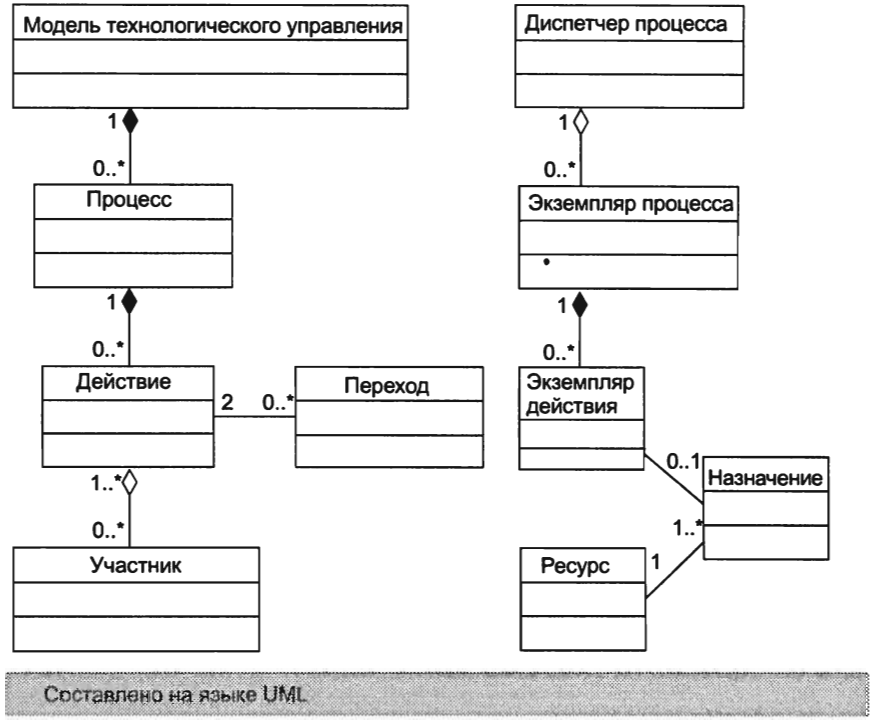


Рис. 17.7. Диаграмма классов компонента технологического управления

Одна из проблем, которую Inmedius предстояло решить, была связана с параллелизмом. Есть ли необходимость в том, чтобы модель технологического управления могли модифицировать несколько пользователей одновременно? Следует ли разрешать пользователю модифицировать модель технологического управления при наличии ее активных экземпляров для периода прогона? Может ли пользователь запускать новый технологический процесс, если его модель в данный момент модифицируется? Учитывая реализацию, а также отношения между моделью и ее экземплярами периода прогона, было бы весьма проблематично ответить утвердительно на любой из этих вопросов. Следовательно, в области решений перечисленные ситуации нужно запретить.

Поскольку все трудности в перечисленных ситуациях связаны с модифицированием модели технологического управления, решение предусматривало наличие связанного с ней ключа. Для того чтобы модифицировать модель, пользователь должен получить этот ключ. Каждой модели соответствует единственный ключ, причем при наличии активных экземпляров этой модели для периода прогона ключ не выдается. Кроме того, если модель технологического управления заблокирована, создание новых экземпляров периода прогона также блокируется.

Пакетирование

Компонент технологического управления пакетируется в двух элементах EJB: в сеансовом bean без состояния, предназначенном для управления экземплярами

модели технологического управления, и bean-сущности, предназначенной для управления самой моделью (рис. 17.8). Это решение о пакетировании компонента обусловливается исключительно характеристиками различных типов EJB.

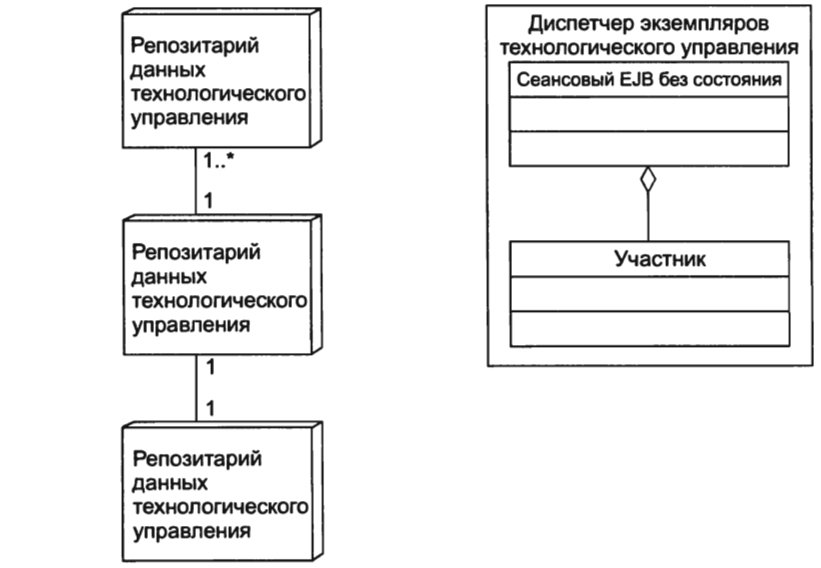


Рис. 17.8. Диаграмма пакетов компонента технологического управления

Абстракции, реализуемые в приложении сущностными EJB, выражают совместно используемые ресурсы, в рамках которых данные устойчивого объекта доступны многочисленным компонентам и пользователям. Модель технологического управления представляет единственный ресурс подобного рода, а именно процесс, допускающий многократную конкретизацию. Любой пользователь приложений Inmedius, независимо от своего местоположения, может запускать новые процессы, основанные на упомянутой модели технологического управления, и участвовать в их действиях.

Сеансовые элементы EJB моделируют состояние и поведение. В продолжение жизненного цикла экземпляра технологического процесса или сеанса пользователям предоставляются различные службы — в частности, определения новых моделей технологического управления, создания экземпляров технологического управления, создания действий, распределения между действиями ресурсов и завершения действий. Таким образом, реализация экземпляров технологического процесса в виде сеансовых EJB вполне естественна.

После принятия решения о представлении диспетчера экземпляра технологического процесса в виде сеансового элемента EJB необходимо было определиться с тем, к какому типу этот элемент будет принадлежать, — будет ли он сохранять состояние или нет. Решение этого вопроса зависело от характеристик сохраняемого

состояния. Как правило, сеансовый EJB с запоминанием состояния сохраняет информацию о состоянии для того клиента, с которым он в данный момент поддерживает диалог. В то же время состояние экземпляра технологического процесса в период прогона обновляется несколькими клиентами — в частности, теми, которые участвуют в процессе технологического управления, и диспетчерами, которые отслеживают процесс и анализируют его результаты. Исходя из этого диспетчер экземпляров технологического управления решили реализовать в виде сеансового EJB без запоминания состояния — во-первых, он более легковесен и масштабируем, чем сеансовый EJB с запоминанием состояния, а во-вторых, информацию о состоянии он сохраняет от имени конкретного клиента в базе данных, к которой могут обращаться все остальные клиенты.

С тем, как пакетировать отдельные объекты в рамках модели технологического управления, было связано еще одно компромиссное решение. Следует ли их пакетировать в виде сущностных EJB или в виде Java-классов, пакетированных в другой структуре (например, в библиотеке)? Поскольку эти объекты взаимодействуют и зависят друг от друга, при их пакетировании в виде сущностных EJB нужно было бы постоянно обнаруживать местонахождение и сохранять многочисленные EJB-дескрипторы приложения, а следовательно, вводить дополнительные издержки. Кроме того, как мы помним, вызов любого метода в EJB производится удаленно (в виде RMI), а значит, также связан с непроизводительными издержками. Большинство контейнеров J2EE способны определять, относится ли вызываемый метод к той же виртуальной машине Java, к которой принадлежит вызывающая сторона, и, таким образом, оптимизировать его до локального вызова, но все-таки это делают далеко не все. Исходя из этих соображений было принято следующее проектное решение. Крупные абстракции (такие, как модель технологического управления) в приложении предполагалось выразить в виде объектных EJB, а относительно мелкие — реализовать в сущностных EJB в виде библиотек Java-классов. Все эти операции были направлены на снижение дополнительных издержек, связанных с тяжеловесными отношениями между сущностными EJB.

Рассматриваемое проектное решение компонента технологического управления, в частности, выразилось в вопросе о расположении логики, ответственной за подачу запроса на ключ модификации модели технологического управления. Изначально эта логика располагалась внутри сущностного элемента EJB, реализовывавшего модель технологического управления. Запрос на выделение ключа должен был отправляться напрямую этому сущностному элементу, который в свою очередь должен был принимать решение о его предоставлении или непредоставлении (а в случае предоставления — о блокировке модели для других пользователей).

Эта проблема стала очевидной, когда пришло время корректировать бизнес-логику, — так, чтобы ключ выдавался только в отсутствие активных экземпляров технологического управления периода прогона. Методы, предоставлявшие информацию об экземплярах технологического управления в период прогона, определялись в сеансовом элементе EJB с запоминанием состояния — объекте, взаимодействовавшем с сущностным EJB. Решение о передаче ссылки на сеансовый EJB с запоминанием состояния сущностному EJB оказалось неоптимальным —

в первую очередь, по той причине, что в таком случае сущностный EJB был бы привязан к среде (а следовательно, возможность повторного использования исключалась); во-вторых, потому что сущностный EJB обращался бы к сеансовому EJB с запоминанием состояния только посредством удаленных вызовов методов.

В принципе, можно было бы напрямую обращаться к объектам доступа к данным, извлекая с их помощью из базы данных всю необходимую информацию. Однако в этом случае реализованная в сущностном EJB абстракция оказалась бы нарушенной, в связи с чем он получил бы дополнительные обязанности, относящиеся к другому объекту. Наконец, такое решение предполагает дублирование кода, затрудняющее сопровождение.

Было найдено следующее решение. Логика (регулирующая предоставление ключа для изменения модели технологического процесса) помещалась в сеансовый EJB без состояния. Сущностный EJB в таком случае знает, как извлекать ключи из базы данных и помещать их в нее. При получении запроса на предоставление ключа сеансовый EJB без состояния определяет возможность его предоставления и, если таковая существует, инструктирует сущностный EJB относительно блокировки модели технологического управления. Такое решение сохраняет целостность реализованных в объектах абстракций и устраняет излишние взаимосвязи между элементами EJB.

Распределенные и обособленные операции

С проектированием в компоненте распределенных и обособленных операций связан ряд нетривиальных проблем, которые, в основном, касаются поддержки распределенного параллелизма операций технологического управления. Рассмотрим сценарий, согласно которому модель технологического управления и ее экземпляры периода прогона рассредоточены между несколькими серверами. Средства поддержки транзакций в архитектуре J2EE не позволяют двум пользователям при одновременном обращении к одним данным в базе данных нарушать правила технологического процесса. С другой стороны, эти средства не гарантируют соблюдения правил при одновременном обращении двух пользователей к данным, относящимся к одному технологическому процессу, но продублированным в разных базах данных.

Согласно данному сценарию, в то время как один пользователь создает новый экземпляр модели для периода прогона в одном местоположении, другой пользователь может заблокировать модель технологического управления в другом местоположении. В ходе дублирования и синхронизации данных между распределенными серверами иногда возникают конфликты, которые в отсутствие мероприятий по их разрешению способны нарушить данные технологического управления в корпоративной среде. Для обеспечения соблюдения правил технологического управления в масштабах нескольких баз данных требуется дополнительная функциональность, предназначенная для разрешения такого рода конфликтов. Реализация функциональности на этом уровне не входила в планы Inmedius в отношении первоначальной версии архитектуры. Соответствие обозначенному требованию можно было обеспечить исключительно путем поддержки распределенных и детализированных рабочих сценариев.

В архитектуре и среде системы первоначально требовалось реализовать как минимум два подобных сценария. При распределенных операциях используется общий репозитарий, допускающий совместное использование и поддерживающий транзакции (к примеру, в такой роли может выступать база данных). Иначе говоря, в разных местоположениях могут существовать несколько экземпляров сервера приложений, однако все они должны обращаться к одному репозитарию данных, в котором содержатся модель технологического управления и ее экземпляры периода прогона. Связано это с тем, что в репозитарии хранится информация, с помощью которой сервер приложений устанавливает факт нарушения правил технологического управления. В рамках обособленных операций выделяется одна сборка (состоящая из сервера приложений и репозитария данных), исполняющая роль основной; все остальные сборки трактуются как ее подчиненные экземпляры. В обязанности основной сборки входит создание модели технологического управления, которая после этого дублируется во всех остальных сборках. После распространения модели изменять в ней можно только информацию об участниках заданных действий. При создании, а впоследствии и при закрытии экземпляров технологического управления периода прогона, хранящихся в подчиненных сборках, они передаются основной сборке в целях архивации.

Следствия применения J2EE

В этом разделе рассматривается логическое обоснование ряда решений, принятых в рамках архитектуры Luther относительно использования J2EE.

Индивидуальные проектные решения и решения, обусловленные спецификой J2EE

В ходе проектирования системы в исполняемой среде J2EE одни решения проектировщик принимает самостоятельно, а другие — исходя из правил и структуры J2EE. К примеру, J2EE устанавливает местоположение сервлетов JSP и EJB в составе контейнера: сервлеты и JSP находятся в веб-звене, а элементы EJB — в EJB-звене.

В то же время архитектура J2EE в некоторых случаях допускает принятие гибких проектных решений — к примеру, в том, что касается реализации безопасности (декларативной и программной), поддержки транзакций (декларативной и программной) и доступа к данным (с контейнерным управлением и с самоуправлением bean).

В процессе проектирования компонента проектировщик обладает неограниченными возможностями применения сервлетов, JSP и EJB. В такой ситуации следует избегать опрометчивых решений. К примеру, один из компонентов компании Inmedius допускает сотрудничество двух и более пользователей. Поскольку этот компонент выражает бизнес-логику, допускающую многократное применение, согласно правилам отбора компонентов его следует упаковывать в виде EJB. Углубленный анализ доказывает, что это проектное решение является неоптимальным. Как показано на рис. 16.2, при отображении проектного решения компонента на четыре логических звена, предусмотренные J2EE, необходимо принимать во внимание побочные факторы.

Проблемы многозвенности в J2EE

Одна из этих проблем касается производительности. В число основных факторов, способствующих снижению производительности, входит численность вызовов от одного объекта J2EE (от сервлета или EJB) к другому в рамках транзакции. С технической точки зрения любой вызов метода EJB соответствует удаленному вызову, привносящему дополнительные издержки. Способов решения этой проблемы и, следовательно, обеспечения приемлемой производительности компонента всего два: реализация крупномодульных элементов EJB и исключение отношений между сущностными EJB.

Еще одна проблема связана с транзакциями, которые управляются либо программно, либо декларативно. Очевидно, управлять транзакциями декларативно в некоторых отношениях проще — в коде отсутствуют операторы начала и завершения транзакции. С другой стороны, разработчики должны принимать во внимание предполагаемый механизм применения сущности J2EE. Согласно простейшему решению, транзакции требуются во всех методах. К сожалению, если насущная необходимость в применении транзакций отсутствует, появляются лишние издержки. Иногда дескриптор размещения предписывает поддержку транзакций даже в том случае, если методы сущности J2EE их не предусматривают. Если контейнер — второй участник транзакции — обратится к такой сущности J2EE, созданная им транзакция аварийно закроется. Декларирование дескриптором размещения поддержки транзакции в конкретном методе представляется более оптимальным решением. Особое внимание следует уделять анализу компонента на предмет выявления аспектов, корректность функционирования которых напрямую зависит от поддержки транзакций. Принятые решения необходимо привести в соответствие с декларативными и программными механизмами, предусмотренными в архитектуре J2EE.

17.4. Механизм реализации атрибутов качества в архитектуре Luther

Все, кроме одного, требования по качеству предъявлены к архитектуре Luther заказчиками — они касаются беспроводного доступа, гибкости пользовательских интерфейсов и устройств, поддержки существующих процедур, бизнес-процессов и систем и распределенных вычислений. Сотрудники Inmedius сформулировали единственное требование — об удобстве конструирования приложений.

Относительно реализации этих требований было принято фундаментальное решение о применении архитектуры J2EE. Впрочем, задействовать ее предполагалось строго определенным образом — четко и точно отделить пользовательский интерфейс от приложений, стараться придерживаться стандартов и по возможности сформировать библиотеку повторно используемых компонентов. Стратегии и тактики, применявшиеся для достижения поставленных задач, перечислены в табл. 17.1.

Таблица 17.1. Механизм реализации задач по качеству

Задача	Стратегия	Тактики
Беспроводной доступ	Применение стандартных беспроводных протоколов	Применение предписанных протоколов
Гибкий пользовательский интерфейс	Поддержка средствами HTTP интерфейсов на основе браузера и специальных интерфейсов	Семантическая связность; отделение пользовательского интерфейса; пользовательская модель
Поддержка разнородных устройств	Применение стандартных протоколов	Прогнозирование ожидаемых изменений
Интеграция с традиционными бизнес-процессами	Применение в качестве механизма интеграции спецификации J2EE	Общие абстрактные службы; замена компонента
Оперативное конструирование приложений	Базирование Luther на J2EE и конструирование повторно используемых компонентов	Общие абстрактные службы; обобщение модуля (в роли обобщенного модуля в данном случае выступает J2EE)
Распределенная инфраструктура	Применение J2EE и стандартных протоколов	Обобщение модуля; регистрация в период прогона

17.5. Заключение

Компания Inmedius специализируется на разработке решений для рабочих, обслуживающих технику на местах ее дислокации. Они выставляют к этим решениям требования по высокой мобильности и легкому доступу к компьютерам. Что касается собственно компьютеров, то они обычно обладают высокой переносимостью, а иногда даже предусматривают работу в бесконтактном режиме. Как бы то ни было, системы эти требуют интеграции с конторскими операциями.

Сконструированная в компании Inmedius архитектура Luther направлена на оперативное конструирование систем обслуживания заказчиков. Основывается она на спецификации J2EE. Значительное внимание разработчики уделили созданию повторно используемых компонентов и каркасов, которые упрощают введение новых элементов. Пользовательский интерфейс спроектирован в расчете на удовлетворение потребностей заказчиков и создание решений на основе браузера.

Зависимость от J2EE, с одной стороны, стимулировала развитие коммерческих задач Inmedius, а с другой — обусловила необходимость в принятии дополнительных проектных решений, связанных с пакетированием в виде тех или иных типов beans. В этом мы усматриваем пример обратного воздействия архитектурно-экономического цикла, акцентирующего переход от единичных решений к универсальным решениям.

17.6. Дополнительная литература

Читателям, желающим подробнее изучить переносные компьютеры, мы рекомендуем ознакомиться с изданием [Barfield 01] и с докладами на организуемом Ин-

ститутом программной инженерии (SEI) ежегодном Международном симпозиуме по переносным компьютерам (http://iswc.gatech.edu/).

Описание применяемого в архитектуре Luther образца «бизнес-делегат» содержится в работе [Alur 01]. О деятельности группы по технологическому управлению (Workflow Management Coalition) можно узнать на сайте этой организации по адресу http://www.wfmc.org.

17.7. Дискуссионные вопросы

1. В большинстве рассматриваемых в этой книге конкретных примеров архитектура предусматривает отделение производителей данных в системе от их потребителей. Почему это так важно? Что можно сказать об этой тактике? Составьте список тактик или методик проектирования, с помощью которых осуществляется такое разделение; начните с тех, что упоминаются в настоящей главе.
2. В архитектуре Luther и в других конкретных примерах серьезное внимание уделяется отделению пользовательского интерфейса от остальных элементов приложения. С чем, по вашему мнению, связана чрезвычайная распространенность этой тактики?

Глава 18
Конструирование систем из коробочных компонентов

(в соавторстве с Робертом С. Сикордом и Мэтью Бассом)[1]

На блюде все это смотрится так красиво —
чьи-то пальчики основательно потрудились!

Джулия Чайлд о новой французской кухне

Мы постоянно делаем упор на связь между требованиями по качеству и архитектурой. Утверждение это основывается на допущении о том, что контроль над проектным решением системы подразумевает контроль над реализуемыми ею атрибутами качества. Чем дальше, тем больше оно опровергается реальной практикой. Готовые компоненты для конструирования систем со временем получают широкое распространение — связано это с экономическими соображениями, а также с тем, что во многих технических областях для разработки приложений требуются крайне специализированные знания. Компоненты, с одной стороны, вносят в процесс проектирования существенные коррективы, но с другой — ограничивают архитектуру. Отбираемые для реализации определенного функционального набора, компоненты выражают те или иные архитектурные допущения (а значит, и допущения по качеству). Задача архитектора состоит в том, чтобы обеспечить корректность выбора этих допущений и их совместимость.

Начиная с 1960-х годов принятие многих проектных решений в значительной степени зависит от конкретной операционной системы. Системы управления базами данных сформировались как фактор влияния в начале 1970-х. Повсеместное распространение компьютеров привело к повышению возможностей использования для реализации системных задач компонентов, разработанных третьей стороной. Само по себе наличие компонентов не является принудительным сти-

[1] Роберт С. Сикорд (Robert C. Seacord) — старший научный сотрудник Института программной инженерии; Мэтью Басс (Matthew Bass) — младший научный сотрудник того же учреждения.

мулом к их использованию (в этом контексте см. врезку «Quack.com»), что, однако, не сокращает потребность в изучении механизмов их внедрения в систему.

Процесс отбора коробочных компонентов для системы предполагает поиск *сборок* (assemblies) совместимых компонентов, формулирование механизма реализации с их помощью атрибутов качества и принятие решений относительно возможности их интеграции в конструируемую систему.

QUACK.COM

Истоки

Компания Quack.com основана в конце 1998 года двумя бывшими сотрудниками Института программной инженерии (Джероми Карье (Jeromy Carriere) и Стивом Вудсом (Steve Woods)), а также профессором Гавайского университета Алексом Квилиси (Alex Quilici). Вооружившись идеей обеспечить доступ к коммерческой информации и данным иного характера по телефону, они сконструировали демонстрационную версию программы и к концу лета 1999-го добились финансирования со стороны ряда «благодетелей» и венчурных компаний. Осознавая важность голосовой архитектуры, они выстроили «реальную» систему в виде голосового портала поверх комплекта инструментов и платформы публикации голосовых приложений. В результате им удалось оперативно сконструировать и организовать сопровождение широкого ряда приложений. В принципе, спроектированная ими платформа имела все шансы на доминирование в зарождающейся индустрии. Через десять месяцев после начала финансирования они выпустили предварительную версию потребительского голосового веб-портала. Он предоставлял услуги доступа к информации о погоде, кинофильмах, курсах акций и прочим данным по телефону. 31 августа 2000-го компания вошла в состав America Online. Вскоре — 25 октября 2000 года — AOL выпустила приложение AOLbyPhone, сконструированное группой разработчиков Quack на основе их же платформы и инструментального набора.

История Quack.com наглядно выражает роль и ограничения, присущие коробочным компонентам. На основе вышеизложенных сведений мы можем сделать вывод о том, что компании требовалось вывести свой голосовой портал на рынок как можно скорее. Сказывалась деятельность в этой области других начинающих компаний, причем некоторые из них располагали более серьезной финансовой поддержкой. Сотрудники Quack пытались найти как можно больше готовых к употреблению компонентов, и результаты этого поиска оказали на конечную архитектуру серьезное влияние. Именно по этой причине компании удалось выйти на рынок всего через девять месяцев после начала внешнего финансирования.

Первый портал Quack, внесший весомый вклад в последующий успех проекта, оказался вполне полезным сам по себе, однако широкого внимания пользователей так и не получил. После приобретения проекта корпорацией AOL коммерческие приоритеты резко поменялись. Располагая базой из 34 000 000 подписчиков, AOL вывела на первый план такие коммерческие факторы, как готовность и производительность. Портал Quack.com предстояло подготовить к более интенсивному режиму использования и обеспечить соответствие более жестким требованиям по готовности.

Для подгонки проекта под новые требования было решено переписать компоненты. Вертикальное масштабирование и повышение готовности в рамках гибкой архитектуры не представляло особых проблем; в то же время предсказать реакцию компонентов на предполагаемые изменения оказалось категорически невозможно. Получить контроль над производительностью и готовностью системы в целом можно было лишь путем их переписывания (в порядке убывания критичности).

Многие другие системы прошли аналогичный этап. Недавно мы посетили небольшую начинающую компанию, занятую построением новой линейки программных продуктов. Понимая, что второго шанса произвести первое впечатление не будет, ее сотрудники поставили надежность и масштабируемость на вершину списка архитектурных задач. По словам их архитектора, «если функция не слишком важна, мы ограничиваемся коробочным компонентом. Если в отношении того или иного аспекта системы существует официальный или фактический стандарт, подходящий коробочный компонент можно выбрать — скорее всего, этот стандарт соблюдается разными производителями. Если же у нас есть какие-то сомнения и мы не

можем найти очевидных решений, значит, нужно конструировать компонент собственными силами». Прежде чем перейти к работе над новым проектом, этот архитектор участвовал в создании крупной поисковой системы и контент-провайдера. За четыре года число суточных посещений увеличилось с 45 000 до 45 000 000. Работая с системой многомиллионной посещаемости, он очень быстро научился действовать так, чтобы не просыпаться по ночам в связи с очередной проблемой, угрожающей коммерческой деятельности.

Как средство пакетирования функциональности и ее оперативного внедрения коробочные компоненты очень удобны. С другой стороны, они не позволяют архитектору получить полный контроль над атрибутами качества в системе. Подобно многим другим средствам программной инженерии, компоненты очень полезны, но решения всех проблем от них ждать не приходится.

— LJB и PCC

В настоящей главе мы рассмотрим упрощенный процесс отбора компонентов исходя из практических соображений. Сначала формулируются гипотезы о том, как должны «работать» выбранные компоненты. Затем для проверки этих гипотез строятся простые макеты. Наконец, после оценки функционирования макетов, на случай опровержения гипотез составляется запасной план. Основной принцип этой методики состоит в том, что отбирать единичные компоненты не имеет смысла. Отбирать и тестировать нужно сборки совместно функционирующих компонентов.

Кроме того, ниже мы опишем приложение данного процесса к недавно созданной системе.

18.1. Воздействие компонентов на архитектуру

Предположим, вы разрабатываете программное обеспечение системы управления химическим предприятием. Специальные дисплеи на этом предприятии информируют операторов о состоянии проводимых реакций. Одна из основных задач разрабатываемого программного обеспечения заключается в выводе информации на эти дисплеи. Вам известно, что среди продуктов, предлагаемых одним из производителей, числятся элементы управления пользовательским интерфейсом, способные решить эту задачу. Поскольку купить проще, чем сконструировать, вы собираетесь приобрести эти элементы управления, которые, кстати, написаны в единственной версии — на Visual Basic.

Какое влияние на архитектуру окажет принятое решение? Либо на Visual Basic с присущей ему ориентированностью на обратные вызовы придется написать всю систему, либо операторскую часть нужно каким-то образом изолировать от всех остальных частей системы. Это основополагающее структурное решение, необходимость в принятии которого обусловливается отбором единичного компонента для конструирования отдельной части системы.

Применение в разработке программных средств коробочных компонентов, при всем удобстве, способствует появлению дополнительных проблем. В частности, очень важным с точки зрения архитектуры ограничением являются возможности и обязанности компонентов.

Все компоненты, за исключением самых простых, основываются на допущении об использовании того или иного архитектурного образца, нарушать которое довольно сложно. К примеру, HTTP-сервер предполагает применение клиент-серверного архитектурного образца с определенными интерфейсами и механизмами интегрирования серверной функциональности. Если проектируемая архитектура не соответствует допущениям, заложенным в компоненте HTTP-сервера, задача интеграции очень сильно усложняется.

Поскольку любые компоненты направлены на соответствие определенному архитектурному образцу, решение относительно сборки компонентов, которые предполагается отобрать (или которые уже отобраны) для проектируемой системы, имеет смысл принимать загодя, еще до выбора архитектуры как таковой. Присущие компонентам архитектурные допущения и механизмы их успешной интеграции зачастую обусловливаются или, по меньшей мере, сильно зависят от конкретного набора компонентов. Следовательно, решения относительно отбора компонентов и механизма их взаимодействия необходимо принимать до завершения работы над архитектурой.

18.2. Архитектурное несоответствие

Далеко не все компоненты способны к взаимодействию — даже если это коммерческие продукты, совместимость которых утверждается производителем. Зачастую они «почти совместимы», причем «почти» в данном случае означает фактическую несовместимость. Бывают и более хитрые ситуации — компоненты внешне демонстрируют способность к совместной работе (собранный код компилируется и даже исполняется), и тем не менее из-за того что реальный механизм взаимодействия компонентов не отвечает ожиданиям, реакции оказываются некорректными. Подобного рода ошибки могут быть едва заметны — особенно в системах реального времени и в параллельных системах, в которых компоненты основываются на безобидных на первый взгляд допущениях о времени и относительной последовательности операций.

Короче говоря, компонент, если он не разработан специально для вашей системы, может не соответствовать всем вашим требованиям, более того, он может отказаться работать совместно с другими компонентами. Хуже всего то, что для проверки применимости компонента его, как правило, нужно сначала приобрести. Дело в том, что предоставляемой информации об атрибутах качества интерфейсов компонентов обычно оказывается недостаточно. Насколько безопасен ваш компилятор? Насколько надежна ваша почтовая система? Насколько точна математическая библиотека, к которой обращаются ваши приложения? И что делать, если окажется, что эти элементы «недостаточно» безопасны/надежны/точны?

Подобного рода препятствия к успешной интеграции компонентных систем Гарлан (Garlan), Аллен (Allen) и Окерблум (Ockerbloom) назвали *архитектурным несоответствием* (architectural mismatch). Эту проблему они рассматривают как несогласованность допущений, принимаемых в раздельно разработанных компонентах. Во многих случаях эта несогласованность обнаруживает себя в архитектуре — например, если два компонента не могут договориться о том, кто из

них кого запускает. Архитектурные несоответствия чаще всего обнаруживаются в период интеграции — система элементарно отказывается компилироваться, компоноваться или исполняться.

Архитектурное несоответствие есть частный случай *интерфейсного несоответствия* (interface mismatch). В данном случае имеется в виду определение интерфейса по Парнасу (Parnas) — допущения компонентов друг о друге. Это определение шире стандартного понятия интерфейса, которое, к сожалению, прочно прижилось в современной практике и соответствует интерфейсу прикладного программирования компонента (пример тому — спецификация интерфейса Java). Интерфейс прикладного программирования содержит имена программ и их параметров, а также (иногда) доносит отдельные сведения об их поведении; тем не менее, это лишь малая часть информации, необходимой для обеспечения корректного использования компонента. В полную спецификацию интерфейса должны входить сведения о побочных эффектах, потреблении глобальных ресурсов, требованиям по координации и пр. Подобно архитектурному несоответствию, интерфейсное несоответствие обнаруживается в период интеграции и может выступать причиной возникновения упомянутых выше хитроумных ошибок в период исполнения.

Допущения можно разделить на два подвида. *Допущения о предоставляемых услугах* (provides assumptions) характеризуют службы, которые компонент предоставляет пользователям или клиентам. *Допущения о требованиях* (requires assumptions) подробно описывают службы и ресурсы, необходимые для корректного функционирования компонента. Несоответствие между двумя компонентами происходит в случае, если принимаемые ими допущения двух обозначенных разновидностей не совпадают.

Как бороться с интерфейсным несоответствием? Помимо изменения требований — так, чтобы вчерашние дефекты стали сегодняшними характеристиками (да, иногда можно сделать и так), — есть еще три способа.

- *Специфицирование* и проверка компонентов системы позволяет предотвращать несоответствия.
- Обнаруживать те несоответствия, которые не удалось предотвратить, помогает тщательная *квалификация* компонентов.
- Исправлять выявленные несоответствия следует методом *адаптации* компонентов.

Методики исправления интерфейсных несоответствий

Проблема исправления несоответствий (или «исправления компонентов/интерфейсов») до сих пор не удостоена систематического анализа. Термины наподобие «связки компонентов» отражают характер интеграционного кода и соответствуют статусу «второй сорт», присваиваемому их разработке. Исправление интерфейсных несоответствий зачастую трактуется как задача хакеров и (иногда) неопытных программистов, у которых эстетическое ощущение не опустошено тысячами

«поделок», из которых состоит процесс интегрирования коробочных компонентов. По нашему мнению, это тот самый случай, когда надежность цепочки равна надежности ее слабого звена. Иначе говоря, от качества исправления компонента напрямую зависит успех (или неспособность) реализации общесистемных задач по качеству наподобие готовности и модифицируемости.

Первым шагом на пути систематизации процесса исправления интерфейса мы считаем классификацию основных методик, которыми эта задача решается, и их характеристик. Есть очевидный метод исправления. Заключается он в том, чтобы изменить код проблемного компонента. Это решение далеко не всегда осуществимо — коммерческие продукты довольно редко сопровождаются исходным кодом. Если же речь идет о старом компоненте, то найти его исходный код или человека, который способен в нем разобраться, обычно оказывается крайне сложно. Даже если такая возможность есть, во многих случаях корректировать код компонента нежелательно. Компоненты по определению используются во множестве систем. Таким образом, если изменения, направленные на обеспечение работоспособности компонента в новой системе, приведут к тому, что он перестанет работать в старых системах, сопровождать придется уже не одну, а несколько его версий.

В качестве альтернативы изменению кода одного или обоих несоответствующих компонентов возможно введение кода, согласующего их взаимодействие. Исправительный код делится на три класса: оболочки, мосты и посредники.

Оболочки

Термин *«оболочка»* (wrapper) предполагает инкапсуляцию, посредством которой компонент упаковывается в альтернативную абстракцию. Иначе говоря, клиент обращается к службам «завернутого» компонента исключительно через предоставляемый оболочкой альтернативный интерфейс. Можно сказать, что оборачивание — это создание альтернативного интерфейса компонента. Трансляция интерфейса подразумевает проведение трех операций:

- трансляцию элемента интерфейса компонента в альтернативный элемент;
- сокрытие элемента интерфейса компонента;
- недопущение модификации элемента базового интерфейса компонента.

Предположим, что в нашем распоряжении унаследованный компонент, предоставляющий программный доступ к службам визуализации графики. При этом программная служба существует в виде библиотек Fortran, а визуализация графики осуществляется в терминах специальных графических примитивов. Задача в том, чтобы, во-первых, сделать компоненты доступными клиентам через CORBA, а во-вторых, заменить специальные графические примитивы графическими элементами X Window System.

Для специфицирования нового интерфейса, обеспечивающего доступность служб компонентов клиентам CORBA, имеет смысл обратиться к языку описания интерфейсов (interface description language, IDL) — это решение удачнее связывания с библиотеками Fortran. В качестве кода, исправляющего допущения интерфейсов о предоставляемых службах, выступает «скелетный код» на C++, который генерируется компилятором IDL автоматически. Кроме того, в состав

исправительного кода входит рукописный код, привязывающий скелет к функциональности компонента.

Что касается оборачивания допущений о требованиях компонентного интерфейса, необходимого для окончательного перехода от специальных графических элементов к системе X Window, то здесь есть несколько вариантов. Одно из решений заключается в том, чтобы написать уровень для библиотеки трансляции, чей API соответствует API специальных графических примитивов; реализация этой библиотеки будет проводить трансляцию вызовов специальных графических элементов в вызовы X Window.

Мосты

Мост (bridge) осуществляет преобразование допущений о требованиях одного компонента в допущения о предоставляемых услугах другого компонента. Основное различие между мостом и оболочкой состоит в том, что код исправления, из которого состоит мост, не зависит ни от одного конкретного компонента. Кроме того, мост в обязательном порядке явно вызывает внешний агент — в этом качестве может выступать один из тех компонентов, которые соединяются мостом. Последнее обстоятельство свидетельствует о переходном характере мостов, а также о том, что конкретная трансляция определяется в период конструирования моста (например, во время его компиляции). Почему указанные различия так важны, станет ясно из обзора посредников.

Как правило, мосты осуществляют более узкий, чем оболочки, круг трансляций интерфейсов — связано это с тем, что они специализируются на конкретных допущениях. Чем больше допущений принято для моста, тем меньше компонентов, с которыми его можно применить.

Предположим, что в нашем распоряжении два унаследованных компонента. Один создает выходные макеты проектных документов на PostScript, а другой отображает документы в формате PDF (Portable Document Format). Задача — интегрировать эти компоненты, чтобы в отношении любого проектного документа можно было запустить компонент отображения.

Наиболее очевидным решением исправления интерфейса для данного сценария представляется простой мост, транслирующий PostScript в PDF. Писать его можно независимо от конкретных характеристик двух наших гипотетических компонентов — к примеру, имеет смысл создать механизм, извлекающий данные из одного компонента и наполняющий ими другой. В этом контексте на ум сразу приходят фильтры UNIX, однако это не единственный механизм, способный выполнить поставленную задачу.

Для исполнения такого моста можно написать сценарий. Он должен будет разбираться со спецификой интерфейсов интегрируемых компонентов. Следовательно, поскольку внешний агент/оболочка (shell) имеет дело с интерфейсами на обоих концах отношения интеграции, он не подпадает под наше определение оболочки (wrapper). С другой стороны, можно сделать так, чтобы каждый из двух компонентов мог самостоятельно запускать фильтр. В этом случае механизм исправления будет представлять собой гибрид оболочки и фильтра — в оболочке будет содержаться исправительный код, необходимый для определения потребности в запуске моста и для инициирования самого запуска.

Посредники

Посредники сочетают свойства мостов и оболочек. Основное различие между мостами и посредниками состоит в том, что последние, помимо прочего, осуществляют планирование, которое, в конечном счете, приводит к заданию трансляции в период прогона (как вы помните, мосты устанавливают трансляцию в период своего конструирования).

Общность посредника и оболочки выражается также в том, что и те и другие становятся более явными компонентами общей системной архитектуры. Другими словами, примитивные семантически, носящие зачастую переходный характер, мосты представляют собой вспомогательные механизмы исправления, выполняющие в рамках проектного решения неявную роль; что же касается посредников, то их семантическая сложность и автономия (устойчивость) в период прогона позволяют им исполнять в программной архитектуре более заметную роль. Для иллюстрации посредников обратимся к присущей им функции планирования в период прогона — благо именно в этом состоит разница между посредниками и мостами.

Возьмем, к примеру, сценарий интеллектуального слияния данных. Рассмотрим датчик, генерирующий высокоточные данные в больших объемах. В период исполнения действуют разные потребители информации с разными рабочими допущениями о точности данных. Одному потребителю, который, скажем, допускает низкую точность данных, нужно «выдрать» из потока данных определенную информацию. Другому потребителю с аналогичными требованиями по точности, но иными характеристиками по части пропускной способности, требуется временная буферизация данных. В каждом из этих случаев согласование датчика и его потребителей выполняет посредник.

Рассмотрим другой сценарий — сборка последовательности мостов в период прогона, направленная на интеграцию компонентов, при условии, что их требования по интеграции становятся известны именно в этот период. К примеру, один компонент производит данные в формате D^0, а другой потребляет данные в формате D^2. Прямого моста $D^0 \rightarrow D^2$ может и не быть, зато есть мосты $D^0 \rightarrow D^1$ и $D^1 \rightarrow D^2$, которые можно сцепить. Роль посредника, таким образом, будет заключаться в том, чтобы собрать эти мосты и тем самым провести преобразование $D^0 \rightarrow D^2$. Этот сценарий одновременно распространяется на традиционное понятие настольной интеграции и на более экзотичные адаптивные системы периода прогона.

Методики обнаружения интерфейсных несоответствий

Для того чтобы исправить несоответствие, его сначала нужно обнаружить. Представленный ниже процесс выявления несоответствий являет собой усовершенствованный вариант квалификации компонентов.

Термин *квалификация компонентов* (component qualification) обозначает процесс, в ходе которого проверяется соответствие коммерческого компонента ряду критериев «годности к использованию». Среди неотъемлемых процессов квалификации компонентов — макетная интеграция компонентов-кандидатов. На этом этапе интеграции выявляются малозаметные, трудно поддающиеся обнаружению

разновидности интерфейсных несоответствий — например, состязания за ресурсы. Фактически, потребность в проведении этого этапа обусловливается недостаточностью наших познаний в области интерфейсов компонентов.

Проверка эта исходит из наблюдения, согласно которому для предоставления каждой службы компонента необходимо удовлетворить ряд принимаемых им допущений о требованиях. Ведь служба — это не более чем удобный способ описания механизма пакетирования функциональности компонента в расчете на ее применение клиентами. Таким образом, квалификация предполагает:

- выявление для каждой службы, которую предполагается задействовать в системе, принимаемых компонентом допущений о требованиях;
- удовлетворение каждого допущения о требованиях соответствующим допущением о предоставляемых услугах, принимаемым системой.

Перейдем к конкретике и рассмотрим квалификацию компонента, предоставляющего многопоточным приложениям службы управления примитивными данными. В частности, этот компонент позволяет записывать значения данных в указанное местоположение (которое может задаваться с помощью ключа). В целях предоставления многопоточной службы хранения компоненту требуется от операционной системы ряд ресурсов — к примеру, файловая система и примитивы блокировки. Перечень принимаемых компонентом допущений о требованиях может быть (а может и не быть) составлен производителем компонента; если таковой отсутствует, значит, выявлением этих требований приходится заниматься оценщику компонента. В любом случае, конкретное отображение позволит сделать выводы о воздействии обновления операционной системы на данное отношение интеграции. Иными словами, меняется ли в ходе обновления операционной системы семантика операций fwrite и flock?

В упомянутый перечень можно внести дополнительные допущения — к примеру, возможно такое допущение о представляемых услугах, согласно которому у устройства хранения должен быть интерфейс CORBA. В зависимости от конкретной реализации брокера объектных запросов это допущение предполагает или не предполагает появления нового допущения о предоставляемых услугах, согласно которому на базовой машине, размещающей данное устройство хранения, должен исполняться процесс брокера объектных запросов. На основе перечня допущений иногда выявляются значительно более интригующие зависимости. К примеру, тот же самый гипотетический компонент может устанавливать заданное (но переменное) число клиентов, работающих с одним внешним процессом диспетчера данных, — соответственно, для всех клиентов сверх заданного количества нужно выделять дополнительные процессы. Подобные допущения позволяют точно определиться с тем, соответствует ли компонент всем ограничениям по ресурсам системы.

Методики предотвращения интерфейсных несоответствий

Согласно одному из методов предотвращения интерфейсных несоответствий, с самых ранних стадий проектирования следует систематически устанавливать как

можно больше допущений об интерфейсе компонента. Возможно ли специфицировать все допущения, принимаемые компонентом о своем окружении, а также те из них, которые ему дозволено принимать используемыми компонентами? Естественно, нет. Имеет ли смысл выделить важное подмножество допущений и есть ли у нас свидетельства, подтверждающие эффективность такого подхода? Да, есть. В проектном решении программного обеспечения A-7E, обзор которого содержится в главе 3, система подразделяется на иерархическое дерево модулей. На его высшем уровне содержится три модуля, а на листьях их 120. Для каждого листового модуля составлена спецификация интерфейса, в которой обозначаются программа доступа (в объектном проектировании эти программы называются методами), требуемые и возвращаемые параметры, видимые следствия вызова программы, параметры генерации системы, предусматривающие регулировку модуля в период компиляции, а также ряд допущений (примерно по дюжине на каждый модуль).

В допущениях выражались утверждения относительно *достаточности* (sufficiency) предоставляемых каждым модулем служб, а также *возможности реализации* (implementability) каждой службы в категориях требуемых модулям ресурсов. Среди конкретных тематических областей числились потребление совместно используемых ресурсов, следствия прохождения через средства модуля множества потоков управления, а также производительность. Эти допущения должны были оставаться неизменными на протяжении всего жизненного цикла системы, в которой модифицируемость позиционировалось как основная проектная задача. Обращаясь к допущениям, проектировщики модулей проверяли, в полной ли мере им удалось инкапсулировать в своих модулях области изменений. Специалисты в предметной и прикладной областях с помощью допущений проводили оценочные мероприятия, а пользователи модулей удостоверяли их пригодность. По убеждению участников проекта A-7, внимательное отношение к интерфейсам модулей, по сути, исключает необходимость проведения этапа интеграции (в рамках жизненного цикла программы). Почему? Средством предотвращения архитектурных несоответствий они избрали тщательное специфицирование — в частности, проверку достоверности явных перечней допущений силами специалистов в предметной и прикладной областях.

Определение интерфейса как набора допущений, выводящее его за рамки интерфейса прикладного программирования, расширяет понимание процесса специфицирования интерфейсов компонентов, взаимодействующих друг с другом в различных контекстах. *Приватные интерфейсы* (private interfaces) раскрывают только те принимаемые базовым интерфейсом компонента допущения о предоставляемых услугах и требованиях, которые являются значимыми в контексте требований по интеграции в конкретной системе или даже в контексте ее отдельных компонентов. Таким образом, информация о ненужных средствах, присутствие которых приводит к излишнему усложнению системы, скрывается.

Мы перечислили преимущества нескольких различных интерфейсов для одного компонента в противоположность единому многоаспектному базовому интерфейсу. Ужесточение контроля над межкомпонентными зависимостями увеличивает возможности манипулирования отдельными видами развития систем — в частности, предсказуемость воздействия модернизации коммерческого компонента

на новую версию. Оболочки как стратегия исправления ориентированы на обеспечение конфиденциальности. Архитектурные образцы, со своей стороны, предусматривают стандартные формы удовлетворения допущений интерфейсов о предоставляемых услугах и требованиях — в системе на основе архитектурного образца, определяющего компактный набор типов компонентов, количество производных базового интерфейса значительно сокращается.

Параметризованным (parameterized) называется интерфейс, предусматривающий возможность изменения допущений о предоставляемых службах и требованиях перед вызовом службы соответствующего компонента; эта возможность обеспечивается путем изменения значения переменной. В языках программирования давно сформировались методики параметризации с развитой семантикой, позволяющие регулировать интерфейс компонента с момента его проектирования и кодирования до момента вызова его служб (примеры таких методик — настраиваемость в Ada и полиморфизм в ML). Возможность настройки в коммерческих продуктах часто обеспечивается за счет параметризации (например, файлов ресурсов и переменных среды). Параметризованные интерфейсы предусматривают создание адаптационного кода, который по отношению к компоненту является одновременно внешним (в том, что касается настройки значений параметров) и внутренним (по части приспособления к значениям параметров).

Подобно посреднику, который определяется как мост с логикой планирования, *согласованный интерфейс* (negotiated interface) представляет собой параметризованный интерфейс с логикой самоисправления. Параметризация такого интерфейса производится либо его же средствами, либо внешним агентом. Можно даже утверждать, что согласованные интерфейсы характерны для программных средств с автоматическим конфигурированием — согласование при этом проходит в форме одностороннего диалога по принципу «решайте сами», проводящегося между программой конструирования компонентов и базовой платформой. Есть и другой вариант — некоторые продукты (в частности, модемы) в период прогона (вместо периода установки) определяют взаимоприемлемые параметры связи путем регулярного запуска специальных протоколов.

Подобно оболочкам, которые в качестве стратегий исправления устанавливают полупрозрачность, посредники выражают стратегию исправления, направленную на введение в несогласованные компоненты согласованных интерфейсов.

18.3. Компонентное проектирование как поиск

Поскольку возможности и обязанности компонентов в процессе разработки систем выступают одним из основных источников архитектурных ограничений, но в то же время в любой системе участвует множество компонентов, компонентное проектирование системы превращается в поиск совместимых *ансамблей* (ensembles) коробочных компонентов, которые в максимальной степени удовлетворяют задачам системы. Архитектор должен установить возможность или, наоборот, невозможность интеграции компонентов в каждом отдельно взятом ансамбле

и, в частности, оценить соответствие ансамбля архитектуре и выставленным к системе требованиям.

Если ансамбль способен ужиться в архитектуре, значит, он подлежит дальнейшему анализу. На первоначальном этапе необходимо оценить осуществимость анализа, убедиться в отсутствии существенных архитектурных несоответствий, не допускающих приемлемого приспособления. Необходимо также принять во внимание осуществимость исправления и остаточный риск после окончания этого процесса.

Одновременное проведение анализа в нескольких направлениях, очевидно, слишком затратно. Как мы покажем в примере, разумнее сосредоточиться на одном, основном, направлении, а все остальные рассматривать как дополнительные. Выбранные компоненты крайне важно рассматривать как ансамбли, и ни в коем случае не по отдельности. Кроме того, не следует забывать, что любое направление анализа — это лишь гипотеза, требующая проверки, и никак не окончательное решение.

«Можно ли реализовать атрибуты качества системы в рамках компонентно-ориентированных вариантов архитектуры?» Ответ поначалу кажется очевидным — нет. Возможность применения существующих коробочных пакетов, содержащих дополнительную функциональность, во многих случаях начинает быстро перевешивать производительность, безопасность и другие требования к системе. Коробочные компоненты иногда стирают границу между требованиями и проектным решением системы. Оценка компонентов часто приводит к изменению требований к системе — повышает ожидания относительно предоставляемых ими возможностей и заставляет пересматривать другие «требования».

Некоторая гибкость требований к системе полезна не только в контексте интеграции компонентных систем; она также помогает понять, какие требования действительно важны, и, соответственно, ориентироваться на их неукоснительное соблюдение. Как же обеспечить реализацию важнейших атрибутов качества в компонентной архитектуре?

В предыдущем разделе мы говорили о том, что интеграция компонентов — это одна из основных областей риска, и в задачи системного архитектора, помимо прочего, входит принятие решения об осуществимости интеграции ансамбля компонентов при соблюдении функциональной целостности системы и соответствии требованиям по атрибутам качества. Ансамбли необходимо оценивать не только на предмет возможности успешной интеграции компонентов, но и в контексте решения задач по атрибутам качества. В ходе оценки осуществимости ансамбля компонентов — в том числе его способности к реализации желаемых атрибутов качества — применяются модельные задачи.

Строго говоря, *модельная задача* (model problem) — это описание проектного контекста, определяющего ограничения реализации. К примеру, если в разрабатываемом программном обеспечении должен быть веб-интерфейс, исправно работающий в браузерах Netscape Navigator и Microsoft Internet Explorer, значит, эта часть проектного контекста ограничивает пространство решений. Все требуемые атрибуты качества также входят в проектный контекст.

Прототип, находящийся в конкретном проектном контексте, называется *модельным решением* (model solution). В зависимости от серьезности риска,

присущего проектному контексту, и успешности снижения этого риска модельными решениями, у модельной задачи может быть одно или несколько модельных решений.

Как правило, модельные задачи разрабатываются проектными группами. В идеале, проектная группа состоит из архитектора, выступающего техническим руководителем проекта и принимающего в рамках этого проекта основные решения, и некоторого количества проектировщиков/инженеров, реализующих модельное решение модельной задачи.

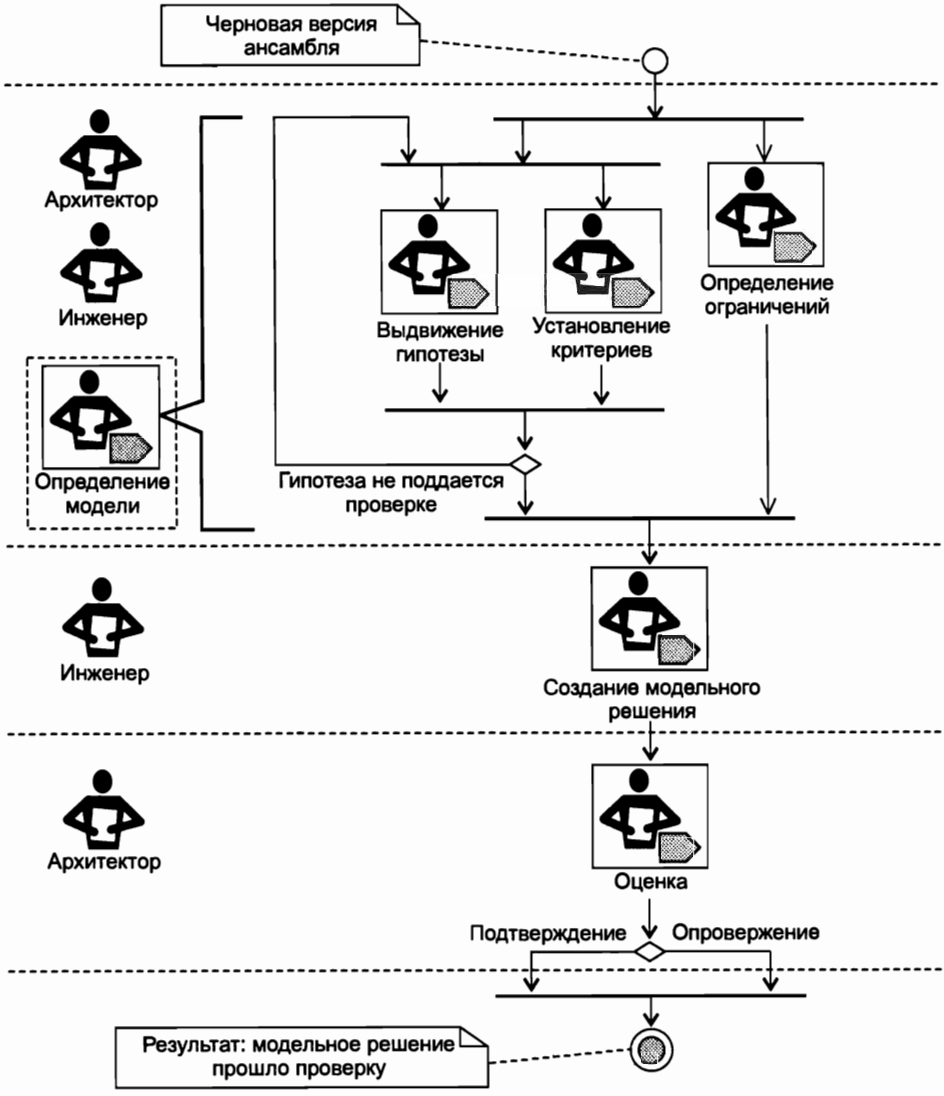


Рис. 18.1. Последовательность действий при решении модельной задачи

На рис. 18.1 изображена последовательность действий при решении модельной задачи. Процесс этот состоит из шести последовательно выполняемых этапов:

1. Архитектор вместе с инженерами формулируют *проектный вопрос* (design question). Ссылающийся на неизвестное, выраженное гипотезой, проектный вопрос инициирует модельную задачу.
2. Архитектор с инженерами устанавливают *исходные критерии оценки* (starting evaluation criteria). В них описывается механизм подтверждения/опровержения гипотезы в модельном решении.
3. Архитектор с инженерами определяют *ограничения реализации* (implementation constraints). Они устанавливают фиксированную (негибкую) часть проектного контекста, которая регулирует реализацию модельного решения. Среди такого рода ограничений фигурируют требования по платформе, версии компонентов и бизнес-правила.
4. В заданном проектном контексте инженеры вырабатывают *модельное решение* (model solution). Модельное решение представляет собой минимальное приложение, обращающееся только к тем характеристикам компонента (или компонентов), которые необходимы для подтверждения или опровержения гипотезы.
5. Инженеры устанавливают *конечные критерии оценки* (ending evaluation criteria). В их число включается начальный набор критериев, а также критерии, установленные по мере реализации модельного решения.
6. Исходя из конечных критериев, архитектор проводит *оценку* (evaluation) модельного решения. В результате проектное решение отвергается или одобряется. В связи с этим часто формулируются новые проектные вопросы, разрешаемые аналогичным образом.

Оставшаяся часть главы отведена под пример и иллюстрацию применения перечисленных этапов при разработке веб-приложения ASEILM.

О, ATAM, ГДЕ ТЫ?

Речь в этой главе идет о том, как определить соответствие отобранного ансамбля компонентов выставленным к системе требованиям по качеству и поведению. Очевидно, это архитектурный вопрос. Так почему же мы не решаем его методом архитектурной оценки — например, методом ATAM? В конце концов, задача ATAM состоит именно в том, чтобы оценить архитектурные решения (в том числе решение о применении компонентов, определенным образом «смонтированных» друг с другом) в свете предъявляемых к системе требований по качеству и поведению. Почему просто не «провести здесь оценку по методу ATAM», и все на этом?

Дело в том, что рассматриваемый в настоящей главе процесс, скорее, касается операций, которые обеспечивают первоочередное принятие наборов архитектурных решений, нежели собственно оценки результатов их принятия. Операции эти больше напоминают макетирование, чем аналитическую оценку.

На примере приложения ASEILM мы видим, насколько многочисленны проблемы совместимости, которые разработчикам приходится решать перед анализом соответствия ансамблей компонентов атрибутам качества. Сборка из компонентов ансамбля — это уже проблема. Если же один ансамбль окажется непригодным, приходится переходить к следующему. Рассматриваемый процесс, подразделяемый на ряд компактных и практичных этапов,

позволяет увереннее принимать обоснованные решения относительно применения того или иного ансамбля.

Каждый из возможных ансамблей строится на ряде гипотез, формулируемых исходя из понимания поставленной задачи. Процесс протекает полупараллельно — вы упорно стараетесь сочетать компоненты друг с другом и с остальными элементами системы, пока не обнаруживаете, что запутались. Затем вы либо собираете компоненты по-другому, либо сразу переходите к запасному плану (другими словами, к следующему ансамблю). Поскольку о том, насколько и каким образом выбранные ансамбли соответствуют атрибутам качества, как оказывается, вы не имеете ни малейшего понятия, на эти самые атрибуты приходится обращать особое внимание.

Для того чтобы провести оценку по методу ATAM, нужно обладать определенными сведениями о применяемых компонентах. Посылка процесса, который мы здесь описываем, заключается как раз в том, что четкая информация о них отсутствует.

Представив процесс в обличии метода, мы преследовали намерение лучше донести его суть и сделать акцент на его повторяемости; впрочем, по большей части, он основан на здравом смысле. Все начинается с обоснованного предположения о компонентах, которые предполагается использовать. Затем происходит конструирование макетов, которые подвергаются тестированию по отдельности и во взаимодействии. Функционирующие компоненты развиваются, и одновременно, на случай, если сформулированное предположение окажется ложным, составляется резервный план действий. Все эти операции проводятся не с отдельными компонентами, а с целым ансамблем.

После завершения процесса проверки ансамбля на правильность для него (равно как и для архитектуры системы в целом) можно смело проводить архитектурную оценку по методу ATAM или по любому другому методу.

— LJB и PCC

18.4. Пример приложения ASEILM

В этом примере мы рассмотрим информационную веб-систему, разработанную сотрудниками Института программной инженерии (Software Engineering Institute, SEI) в целях автоматизации его административных отношений с непостоянными партнерами. Создание системы автоматизированного управления лицензиатами SEI (Automated SEI Licensee Management system, ASEILM) обусловливалось следующими задачами:

- организация распространения лицензированных институтом SEI материалов (в частности, курсов и комплектов для проведения оценки) среди авторизованных лиц;
- сбор административной информации для проведения оценочных мероприятий;
- графическое представление показателей дохода, посещаемости и ряда других сведений о лицензированных материалах SEI;
- отслеживание посещаемости курсов и отчисляемых в пользу SEI гонораров.

В системе ASEILM предусматривается несколько типов пользователей с индивидуальными механизмами авторизации для запуска системных функций.

- Преподаватели курсов могут публиковать списки слушателей, сопровождать контактную информацию и загружать материалы по своим курсам.

- Ведущие специалисты по оценке имеют право открывать новые процедуры оценки, вводить относящуюся к ним информацию и загружать комплекты для проведения оценки.
- Администраторы SEI сопровождают списки авторизованных преподавателей и ведущих специалистов по оценке, а также имеют право просматривать и редактировать любую содержащуюся в системе информацию.

Основываясь на результатах первоначального анализа, разработчикам удалось составить список требований к системе, многие из которых обнаружили прямое соответствие атрибутам качества разрабатываемой системы (табл. 18.1).

Таблица 18.1. Требования по атрибутам качества

Атрибут качества	Требование
Функциональность	Обеспечение веб-связи с географически рассредоточенными заказчиками
Производительность	Производительность, достаточная для обслуживания заокеанских пользователей, пользующихся соединениями низкой пропускной способности (загрузка материалов на их машины должна выполняться не за часы, а за минуты)
Совместимость	Поддержка старых версий веб-браузеров, включая Netscape 3.0 и Internet Explorer 3.0
Безопасность	Поддержка нескольких классов пользователей и предоставление им услуг идентификации и авторизации
Безопасность	Организация безопасной передачи данных через Интернет на уровне коммерческого стандарта

Традиционная практика согласования требований методом взаимных уступок в случае с коробочными компонентами претерпевает некоторые изменения. От них правомерно ожидать и большего, и меньшего — большего в том смысле, что серьезная функциональность в компонентах предоставляется «за бесплатно», а меньшего — в смысле вероятной неточности соответствия этой функциональности потребностям организации, трудности или даже невозможности ее изменения.

Ансамбль Miva Empressa

Руководство обычно рассматривает конструирование систем на основе коробочных компонентов как упрощение процесса разработки, для проведения которого требуются менее опытные, чем обычно, программисты. В реальности в большинстве случаев все наоборот — разработка усложняется (по крайней мере, разработка с листа, то есть с участием неизученного набора компонентов). Для выявления компонентов, которые способны обеспечить реализацию проектного решения, для анализа совместимости рассматриваемых компонентов со всеми прочими компонентами, для определения компромисса между требованиями, применением конкретных компонентов и общей стоимостью разработки — для всего этого необходим недюжинный опыт. При отсутствии такого опыта процесс поиска и квалификации обещает растянуться на неопределенное время.

В нашем примере у группы разработчиков уже были некоторые познания относительно сервера приложений Miva Empressa, и потому они предпочли задействовать его при составлении исходной гипотезы. Miva Empressa — это расширение сервера Microsoft Internet Information Server (IIS), исполняющее сценарии Miva Script на основе XML. Приложения Miva Script в среде Miva Empressa исполняются в рамках IIS. Они способны проводить сложные вычисления — в частности, обеспечивать доступ к базам данных. Они заключены в показанный на рис. 18.2 «заказной компонент». Занимательно, что это *единственный* компонент, который в ASEILM разрабатывался с нуля.

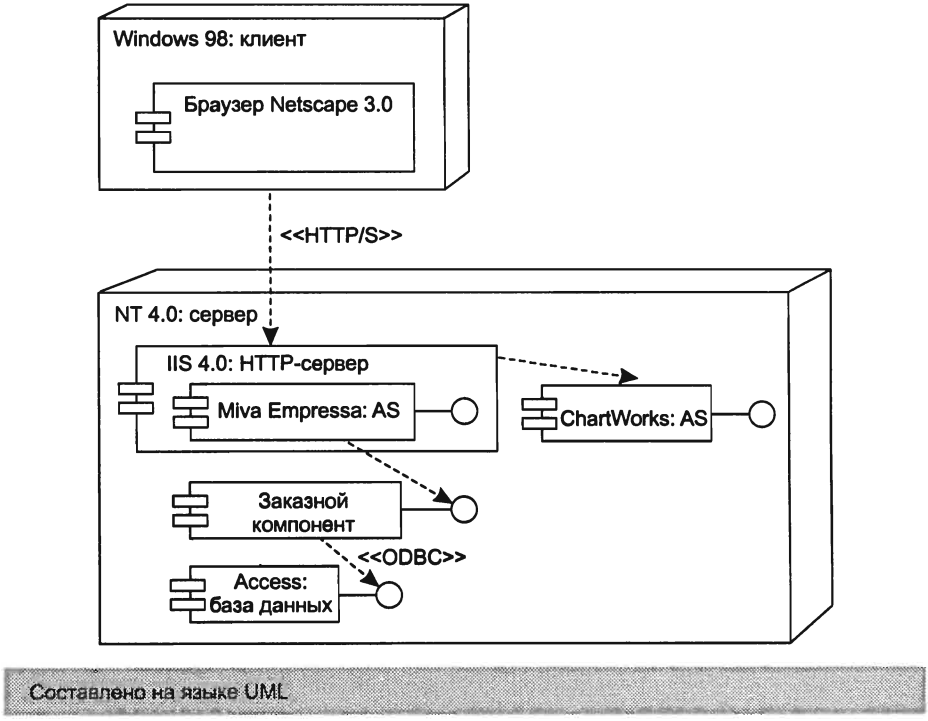


Рис. 18.2. Ансамбль Miva Empressa

В рассматриваемом ансамбле ASEILM, помимо сервера приложений Miva Empressa, задействуются некоторые другие коробочные компоненты:

- в качестве системы управления базами данных применяется Microsoft Access;
- графическое отображение дохода, посещаемости и прочей подобного рода информации обеспечивает приложение ChartWorks от компании Visual Mining;
- в качестве HTTP-сервера применяется Microsoft IIS;
- на серверной платформе устанавливается операционная система Windows NT 4.0.

Клиент может быть представлен любыми платформами и браузерами. В первоначальном ансамбле предусматривались браузер Netscape 3.0 и операционная

система Windows 98. Netscape 3.0 на тот момент уже считалась устаревшей версией с ограниченными возможностями, однако многие ведущие специалисты по оценке (один из видов пользователей ASEILM) продолжали ею пользоваться. Операционная система Windows 98 получила среди пользователей ASEILM серьезное распространение.

Ансамбль по определению является предусловием проведения операций процесса моделирования. Ансамбль, изображенный на рис. 18.2, рассматривался в качестве основы для выработки первоначального модельного решения. Описывая в последующих разделах процесс решения модельной задачи, мы отталкиваемся от исходной гипотезы, согласно которой применение ансамбля Miva Empressa вполне приемлемо.

Этап 1: формулирование проектного вопроса

На первом этапе процесса решения модельной задачи в виде элементов Use Case или сценариев формулируется одна или несколько гипотез, при помощи которых проектное решение проверяется на предмет осуществимости рассматриваемого ансамбля. Исходя из приведенного в табл. 18.1 перечня атрибутов качества системы, можно вывести две гипотезы.

1. *Гипотеза 1.* Ансамбль способен предоставить веб-доступ к данным, содержащимся в базе данных Access, и обеспечить их графическое представление при помощи столбиковых диаграмм и других видов управленческой графики.
2. *Гипотеза 2.* Данные, передаваемые между веб-браузером и HTTP-сервером, можно зашифровать средствами HTTPS.

Основная цель формулирования гипотезы 1 состояла в том, чтобы проверить функциональность системы и ее способность интегрировать требуемые компоненты. Гипотеза 2 позволяет подтвердить осуществимость решения одной из установленных задач по атрибуту безопасности, которая заключается в обеспечении системой ASEILM безопасной передачи данных через Интернет.

Подтверждение обозначенных гипотез в данном случае не доказывает осуществимость ансамбля в целом, однако за счет оценки дополнительных требуемых атрибутов качества значительно приближает эту перспективу. Кроме того, оценка гипотез позволяет провести более углубленный анализ компонентов и механизмов их взаимодействия с ансамблем.

Этап 2: установление исходных критериев оценки

Критерии оценки помогают определиться с соответствием/несоответствием исходных гипотез модельному решению.

- *Критерий 1.* Модельное решение предполагает вывод в браузере диаграммы с данными, хранящимися в базе данных Access.
- *Критерий 2.* Между HTTP-сервером и веб-браузером должна быть организована безопасная передача данных по соединению HTTPS.

Необходимо сделать так, чтобы соответствие критериям оценки можно было проверить. К примеру (касательно критерия 2), о безопасной передаче данных обычно свидетельствует присутствие в веб-браузере пиктограммы замка. Но этого

недостаточно — необходимо провести тщательное тестирование, способное подтвердить, что выводимые в веб-браузере данные действительно происходят из базы данных и не кэшируются где-то посреди маршрута.

Этап 3: определение ограничений реализации

Ограничения реализации позволяют выявить элементы, которые в данном проектном контексте демонстрируют негибкость. Они подтверждают обоснованность проектного решения применительно к разрабатываемой системе. В данном примере ограничения реализации, помимо вышеупомянутых, отсутствуют.

Этап 4: реализация модельного решения

Полностью определив модульную задачу, группа разработчиков приступила к реализации модельного решения — минимального приложения, способного подтвердить или, напротив, опровергнуть сформулированную гипотезу. В ходе реализации допустимо и даже полезно выявлять дополнительные критерии, без соответствия которым ансамбль нельзя признать осуществимым.

В рассматриваемом модельном решении за графическое представление дохода, посещаемости и сопутствующих данных отвечает приложение ChartWorks. Сначала разработчики опробовали очевидное решение, согласно которому браузер должен был посылать серверу IIS HTML-команду, которую тот перенаправлял бы ChartWorks. В состав таких команд предполагалось инкорпорировать запрос с указанием на извлекаемые и представляемые графически данные. При реализации этого решения разработчики столкнулись с двумя проблемами, связанными со сцеплением меток диаграммы с данными и поддержанием безопасного соединения.

Сцепление меток и данных

Приложение ChartWorks описывает диаграммы на CDL (chart description language — язык описания диаграмм); на нем же прописываются механизмы извлечения данных из базы (в этой роли в данном случае выступает Access) и их интеграции с ней. В контексте данного ансамбля требовалось извлечь из базы данных Access метки и данные о диаграмме, причем для этого задействовались два отдельных оператора CDL. К сожалению, CDL не предусматривает механизмов спаривания информации, сгенерированной в результате выполнения разных операторов. Из-за этого возможность непосредственного запроса базы данных средствами этого языка отпадает. Вместо этого запросы Access отправлялись при помощи Miva; в результате создавался текстовый файл, в котором сводились воедино данные и метка. Уже из этого файла при помощи CDL-оператора извлекались объединенные данные.

Этот подход, проявивший работоспособность, был тем не менее признан слишком сложным. К примеру, нужно было отслеживать и не путать многочисленные промежуточные файлы, соответствовавшие разным пользовательским сеансам.

Безопасная передача данных

После обработки на сервере IIS HTML-команда направляется приложению ChartWorks и заставляет его предоставить сгенерированное изображение. Таким образом, в том, что касается применения интерфейсов прикладного программирования ChartWorks, IIS испытывает серьезные ограничения. ChartWorks пре-

дусматривает API для HTTP — но не для HTTPS. Отсюда невозможность установления безопасного соединения между ChartWorks и браузером. Пытаясь решить эту проблему, группа разработчиков отказалась от соединения между IIS и ChartWorks по протоколу HTTPS. Действительно, поскольку оба приложения находятся на одном и том же процессоре, безопасность обеспечивается в отношении доступа к процессору, и протокол передачи к этому непричастен. К сожалению, ничего не получилось — поскольку на веб-страницах безопасные элементы соседствовали с небезопасными, браузер либо запрещал отображение страницы, либо оповещал пользователя о небезопасном элементе соединения. Ни того ни другого допустить было нельзя.

В целях устранения обозначенных проблем разработчики расположили между IIS и ChartWorks прокси-сервер, написанный на Perl. Таким образом, безопасное соединение устанавливалось между IIS и прокси-сервером, который, в свою очередь, обменивался данными с ChartWorks через простое HTTP-соединение. Схема этого решения изображена на рис. 18.3. Кроме того, в HTML-команду пришлось внести изменения, позволяющие ей запускать прокси-сервер Perl.

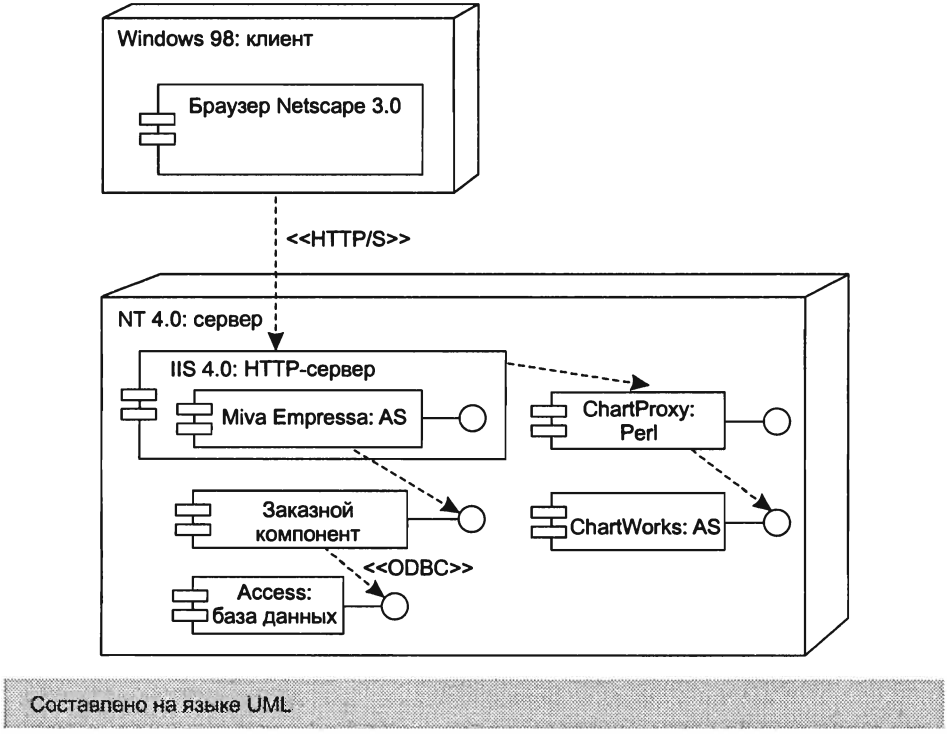


Рис. 18.3. Введение прокси-сервера

Этап 5: определение конечных критериев оценки

В ходе реализации модельного решения Miva Empressa разработчикам удалось выявить дополнительные критерии оценки; в частности, они установили новые

требования по атрибутам качества. По наблюдениям, проведенным в период реализации, элементы графического представления слишком сильно переплетались с серверной логикой. Это обстоятельство усложняло стоявшую перед дизайнерами, не разбиравшимися в универсальном программировании, задачу разработки пользовательского интерфейса системы. Таким образом, в модельную задачу был введен новый критерий оценки.

- *Критерий 3.* Логика представления должна быть четко отделена от серверной бизнес-логики и логики базы данных, а выражать ее следует через четко определенные интерфейсы.

Кроме того, разработчики выяснили, что база данных Access не поддерживает удаленные соединения. Обмен данными между базой данных и сервером приложений Miva через интерфейс ODBC был возможен лишь в случае размещения базы на одной платформе с сервером ISS. Поскольку ISS располагался за границами брандмауэра SEI и лишь в таком варианте мог быть доступен сообществу пользователей, базу данных также требовалось вывести за брандмауэр. Неприемлемость этого ограничения привела к формулированию четвертого критерия.

- *Критерий 4.* База данных должна размещаться в безопасном месте и защищаться брандмауэром.

Этап 6: Оценка модельного решения

Разобравшись с реализацией модельного решения и выявлением дополнительных критериев оценки, архитектор приступает к оценке первого, исходя из последних.

Применение механизмов исправления позволяет обеспечить соответствие обоим первоначальным критериям. В том, что ни один из новых критериев не удовлетворяется, нет ничего удивительного. Таким образом, в отсутствие очевидных способов исправления двух новых проблем, ансамбль был признан неосуществимым.

Ансамбль Java-сервлетов

Помимо ансамбля на основе Miva Empressa разработчики оформили альтернативный ансамбль, основанный на Java-сервлетах. Первоочередное проведение анализа ансамбля Miva Empressa обусловливалось наличием в группе разработчиков ASEILM серьезного опыта работы с компонентами; следовательно, на этом направлении были сосредоточены наиболее обширные ресурсы. Несколько меньше ресурсов пришлось на оценку ансамбля Java-сервлетов. Поскольку исследование, как и в первом случае, сводилось к операциям решения модельной задачи, три этапа удалось сохранить в неизменности.

- Этап 1 — проектный вопрос остался без изменений.
- Этап 2 — в наборе исходных критериев оценки остались все четыре критерия.
- Этап 3 — ограничения также не претерпели изменений.

Таким образом, новая процедура оценки началась с этапа 4, на котором (как изображено на рис. 18.4) конструируется модельное решение.

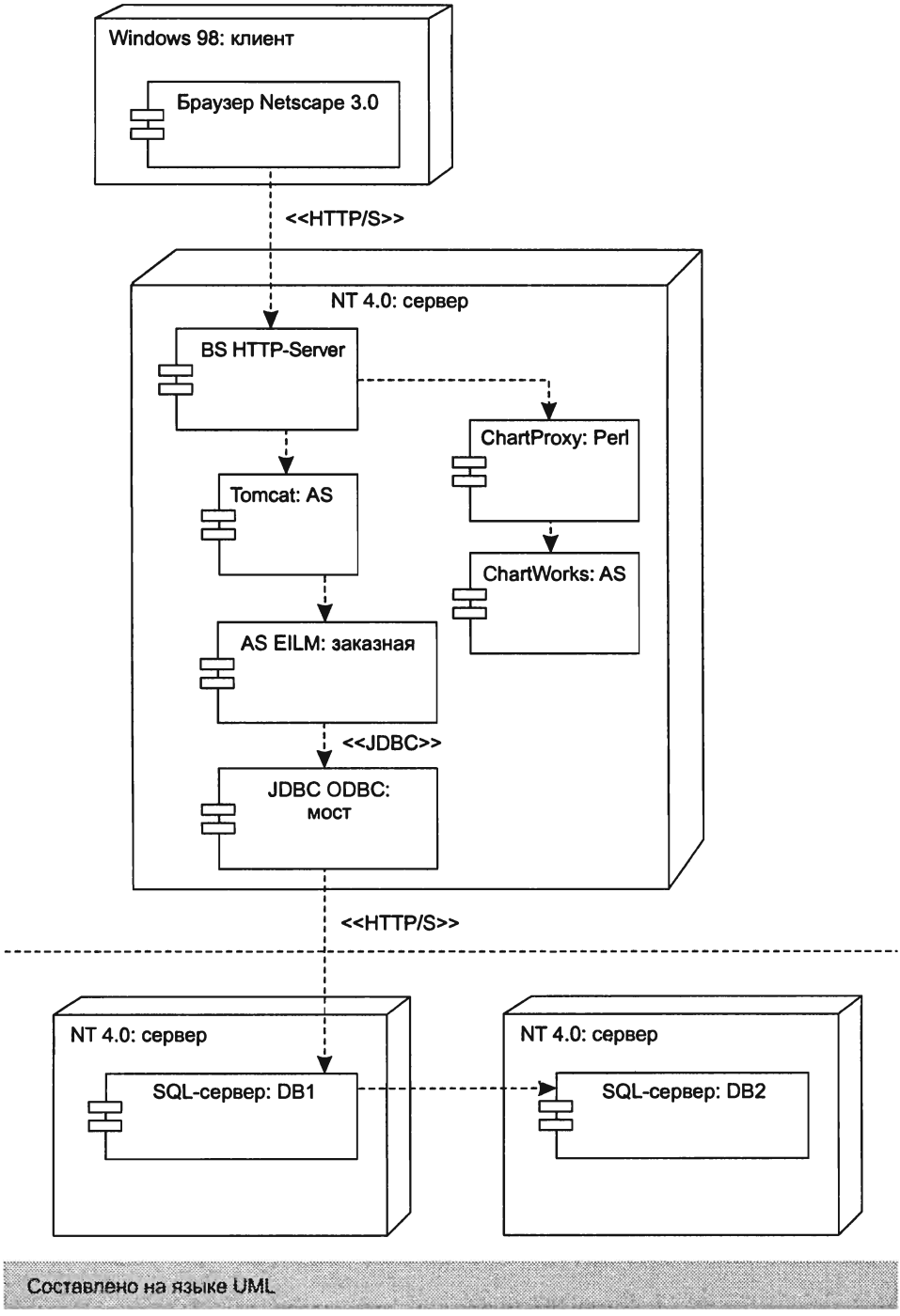


Рис. 18.4. Ансамбль JavaServer Pages

Два первых критерия в новом решении удовлетворялись теми же процессами, которые ранее были реализованы в ансамбле Miva Empressa. Поскольку приложение ChartWorks вошло в состав ансамбля Java, разработчики продолжили попытки исправить несоответствие протоколов HTTP/HTTPS путем введения адаптеров.

Java-сервлеты предусматривают отделение презентационных аспектов системы от бизнес-логики и логики базы данных. Логика представления была ограничена страницами HTML, в то время как бизнес-логику и логику базы данных разработчики перенесли в сервлеты и Java beans, исполнявшиеся на сервере приложений Tomcat, — таким способом они обеспечили соответствие критерию 3. Кроме того, заменив Access на SQL-Server, разработчики добились возможности применения удаленного соединения, не предполагавшего вывода базы данных за брандмауэр, — итак, с критерием 4 тоже разобрались.

В процессе разработки модельного решения нового ансамбля произошли следующие четыре события:

- Исходные критерии, как мы уже сказали, оказались недостаточными.
- Отдельные элементы проектного решения продемонстрировали неспособность удовлетворить исходные критерии. В частности, оказалось, что соблюдение критерия 2 («Между HTTP-сервером и веб-браузером должна быть организована безопасная передача данных по соединению HTTPS») не позволяет обеспечить безопасность системы (о том, с чем это связано, мы поговорим чуть позже).
- Заинтересованные лица сформулировали дополнительные требования.
- Новый Java-ансамбль поставил новые задачи.

Рассмотрим три последние проблемы.

Безопасность

Помимо защиты передачи данных по каналам, в пересмотре нуждалась модель аутентификации. Изначально при аутентификации пользователя на его машине в форме cookie записывался и привязывался к данному сеансу уникальный идентификатор. Разработчики понимали, что в случае компрометации системы безопасности на клиентской машине возможно получение доступа путем обмана с последующей компрометацией защиты серверной машины. Для защиты от таких явлений было принято решение о регистрации IP-адреса машины, его привязке к уникальному идентификатору и проверке при каждом последующем запросе.

Иногда хакеры обращаются к методике «перекрестных сценариев» (cross-side scripting). При этом они сохраняют на своей машине веб-форму и вносят в нее определенные изменения. Впоследствии, после подачи такой формы, происходит аварийный отказ сервера, и он выводит на клиентскую машину код или иную не предназначенную для вывода информацию. Стараясь предотвратить атаки подобного рода, разработчики ASEILM определили ряд исключений.

Дополнительные требования

В ходе разработки о системе ASEILM узнала другая группа заинтересованных лиц, пожелавшая объединить с ней свои данные. Никакой ясности по поводу

содержания, цели интеграции и структуры этих данных поначалу не было. По результатам анализа выяснилось, что у самых разных людей хранились данные, тем или иным образом связанные с данными, которые система ASEILM была призвана отслеживать. Для того чтобы максимально сократить влияние введения в ASEILM новых типов данных, разработчикам предстояло выделить в заказных компонентах уровень абстракции данных, изолировав его таким образом от бизнес-логики. В этом случае система могла бы функционировать, не зная источников и структуры хранилищ(а) данных. Уровни заказного компонента изображены на рис. 18.5.

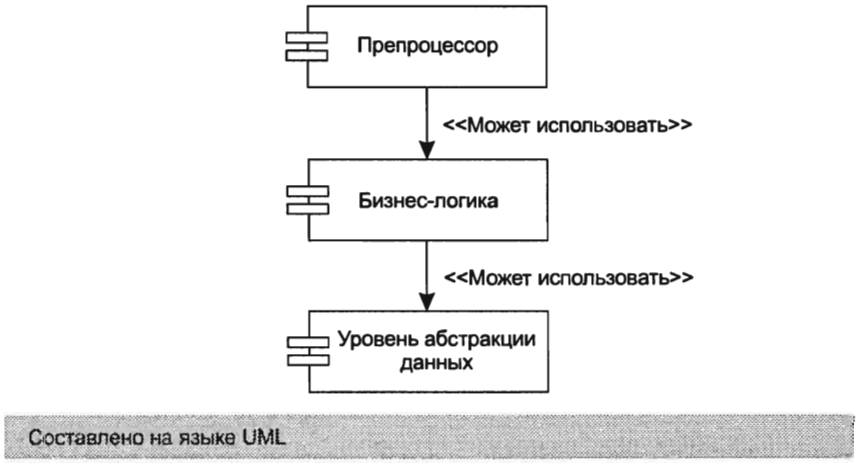


Рис. 18.5. Уровни заказного компонента

Параллелизм

Ансамбль Java-сервлетов, удовлетворявший критериям, которым не смог соответствовать ансамбль Miva, поставил новые задачи, связанные с управлением параллелизмом. Во время работы над модельным решением разработчики поняли, что, в отличие от ансамбля Miva, ансамбль Java-сервлетов не справляется с параллелизмом.

В документации по серверу приложений Tomcat параллелизм не упоминался вообще. Для того чтобы определить, насколько серьезна проблема, группе разработчиков пришлось проанализировать потоковую модель ансамбля. В частности, требовалось изучить отношения между IIS и Tomcat и их воздействие на систему. Выполнив анализ потоковой модели, разработчики предположили, что в каждом случае регистрации пользователя создается особый поток. На основе этой гипотезы они вывели три сценария.

- *Два пользователя одновременно обращаются к системе за разными данными.* После разделения заказного компонента на уровни бизнес-логики и абстракции данных было принято решение о кэшировании на последнем отдельных данных. Таким образом, при инициализации данные извлекались бизнес-логикой из базы данных, проходя при этом через уровень абстракции данных, и впоследствии обслуживались средствами бизнес-логики.

Разработчики не предпринимали никаких действий, направленных на организацию многопоточности бизнес-логики. В ситуации одновременного обращения двух пользователей к бизнес-логике она рассматривалась как критическая секция и пользовательский доступ к ней организовывался последовательно. Поскольку все значимые данные ОЗУ-резидентны, любой запрос выполняется быстро, и неприемлемая продолжительность ожидания фиксируется только при обработке многочисленных одновременных запросов. Учитывая условия использования, одновременные запросы обрабатываются в ограниченном количестве.

- *Два пользователя одновременно обращаются к системе за одними и теми же данными.* Один из аспектов этого сценария — обеспечение непротиворечивости данных в базе — является побочным продуктом решения сценария 1. Поскольку доступ к бизнес-логике осуществляется в последовательном режиме, любое обновление предполагает непротиворечивость данных. Второй аспект — вероятность просмотра и манипуляций с устаревшими данными — выражает проблему «проталкивания» данных пользователю средствами HTTP. Разработчики решили встроить в генерируемый HTML-код функцию периодической перезагрузку текущей веб-страницы — таким образом, в рамках заданного допуска гарантируется соответствие отображаемых и манипулируемых данных текущему состоянию. Это решение, которое, по большому счету, нельзя признать оптимальным, легко реализуется и, исходя из предполагавшейся пользовательской нагрузки, вполне допустимо.
- *Один и тот же пользователь одновременно запускает два сеанса.* Этот сценарий разработчики просто запретили.

Группа разработчиков сопоставила получившееся решение с конечными критериями оценки, которые с момента проведения первого эксперимента с Miva оставались неизменными. Убедившись в том, что ансамбль Java-сервлетов удовлетворяет их всех, группа продолжила работу над реализацией.

Решение на основе ансамбля Java-сервлетов обеспечило соответствие всем потребностям проекта, и к началу 2002 года система ASEILM была готова. Утверждать правомерность принятых допущений о моделях использования еще слишком рано, однако первые положительны. Стоит, впрочем, иметь в виду, что рассматриваемое решение не предполагает удобства масштабирования.

18.5. Заключение

Атрибуты качества можно реализовать даже в такой системе, которая по большей части состоит из коробочных компонентов, и, соответственно, заложенные в ней механизмы проектирования и взаимодействия не поддаются контролю со стороны архитектора. С другой стороны, принципы реализации атрибутов качества в подобного рода системах сильно отличаются от тех, что приняты в отношении специально разработанного кода. Процесс выявления требований в таком случае теряет привычную жесткость — компоненты, имеющиеся на рынке, за счет кор-

ректировки требований повышают общее качество коммерческого решения. С другой стороны, важнейшие требования в процессе оценки осуществимости ансамблей компонентов должны выступать в качестве критичных ограничений. Необходимо учитывать возможность возникновения непредвиденных обстоятельств, а по мере расширения и усложнения важнейших требований предусматривать резервное решение в виде специальной разработки.

18.6. Дополнительная литература

Материалы по методикам и процессам, рассмотренным в этой главе, содержатся в издании [Wallnau 02]. Вопросы внедрения коммерческих коробочных продуктов — их квалификация, связанные с ними риски и миграция — раскрываются на сайте http://www.sei.cmu.edu/cbs/.

Архитектурное несоответствие и методы его исправления подробно рассмотрены в работе [Garlan 95].

Глава 19
Будущее программной архитектуры

> Предсказывать трудно — в особенности предсказывать будущее.
>
> *Нильс Бор*

Историю программирования правомерно рассматривать как последовательное расширение средств выражения сложной функциональности. Вначале был язык ассемблера с узким набором элементарных абстракций, выражавших расположение в физической памяти (относительно адреса в некоем регистре базы), и машинный код исполнения примитивных арифметических операций и операций пересылки. Но даже в таких примитивных условиях у программ была архитектура. Ее элементами выступали блоки кода, связанные физической близостью друг к другу, ветвящимися операторами или подпрограммами с соединителями в виде конструкций ветвления и возврата. В первых языках программирования эти конструкции утвердились, и в роли соединителей уже выступали точки с запятой, операторы перехода и параметрические вызовы функций. 1960-е стали десятилетием подпрограмм.

В 1970-х годах, в связи с нежеланием разработчиков ограничиваться одним-единственным атрибутом качества — корректным функционированием, программы стали структурировать. На основе анализа потоков данных, диаграмм отношений «сущность-связь», информационной закрытости и ряда других принципов и методик формировались сотни проектных методологий. Каждая из них предполагала создание подпрограмм и коллекций подпрограмм с возможностью рационализации их функциональности в категориях атрибутов качества разработки. Эти элементы назывались модулями. В области соединителей изменений не намечалось, но в то же время некоторые новые модульные языки программирования расширили возможности их создания. Абстракции, встраивавшиеся в модули, со временем усложнялись и расширялись, что, в конечном итоге, привело к появлению первых повторно используемых модулей. Пакетировались они таким способом, который, теоретически, исключал необходимость изучения их внутреннего содержания. Таким образом, 1970-е стали десятилетием модулей.

В 1980-х модульные языки программирования, принцип информационной закрытости и некоторые родственные методологии оформились в концепцию

объектов. Объекты получили громадное распространение, а наследование обогатило ассортимент (не относящихся к периоду прогона) соединителей.

В 1990-х годах появились первые стандартные объектные архитектуры, выраженные поначалу в виде каркасов. В рамках объектной технологии сформировался стандартный словарь элементов, подтолкнувший разработку новых инфраструктур для связывания коллекций элементов. Возможности абстракций неуклонно увеличивались. Благодаря этому домашние вычислительные платформы сегодня позволяют обрабатывать сложные сущности — электронные таблицы, документы, графические изображения, звуковые ролики и базы данных — как «черные ящики», которые можно без труда включать в экземпляры друг друга.

Отодвигая отдельные элементы на второй план, архитектура имеет дело с их расположением и взаимосвязью. Необычайно широкие возможности взаимодействия возникают именно благодаря подобного рода абстракции, а индивидуальные элементы здесь играют подчиненную роль.

В текущем десятилетии фиксируется стремительное развитие промежуточного программного обеспечения и информационно-технологической архитектуры, превращающейся в стандартную платформу. Безопасность, надежность и службы обеспечения производительности, реализуемые сегодня в коммерческих элементах, десять лет назад казались достижимыми только лишь при индивидуальной разработке. Основные перечисленные выше этапы программирования изображены на рис. 19.1 в виде диаграммы.

Рис. 19.1. Хронологические этапы развития типов абстракции

Вот на таком этапе мы находимся в данный момент. У нас нет повода усомниться в том, что тенденция к созданию более крупных и функциональных абстракций продолжится. Сегодня уже существуют *генераторы* (generators) для таких сложных и ресурсоемких приложений, как системы управления базами данных и авиационные электронные системы. Наличие в предметной области генератора есть первый признак перехода средств языка программирования в этой области на новый, более высокий уровень. Выражение *«системы систем»* (systems of systems), акцентирующее способность систем к взаимодействию и свидетельствующее об очередном повышении возможностей абстракций, употребляется все чаще.

Далее мы хотели бы еще раз пробежаться по рассмотренным в этой книге проблемам. Полностью согласные с Нильсом Бором, мы не пророчествуем — мы

просто высказываем свои надежды. Перечисляя одну за другой тематические области программной архитектуры, мы поговорим о тех аспектах, которые на данный момент разработаны не так хорошо, как хотелось бы, и выделим проблемы, над которыми исследовательскому сообществу придется основательно потрудиться.

Начнем с обобщения сведений об архитектурно-экономическом цикле (Architecture Business Cycle, ABC), затем обсудим процесс создания архитектуры и ее отношения с жизненным циклом и, наконец, поговорим о том, как компоненты и компонентные каркасы видоизменяют задачи архитектора.

19.1. Снова архитектурно-экономический цикл

В главе 1 мы заявили, что архитектурно-экономический цикл будет объединяющей темой всей книги. По мере изложения мы конкретизировали и развивали ее, старались донести некоторые принципы создания, представления, оценки и разработки архитектуры. Для того чтобы превратить изучение программной архитектуры в окончательно сформировавшуюся область исследований, в ней должны существовать разработанные, фундаментальные направления с практическими результатами. В этом контексте имеет смысл выделить и обсудить четыре версии ABC, которые, по нашему мнению, заслуживают серьезной разработки в будущем.

- Простейший сценарий, согласно которому отдельно взятая компания создает единичную архитектуру для единичной системы.
- Сценарий, согласно которому компания создает на основе архитектуры не одну, а несколько систем, связанных фондом общих средств, и организует их в рамках линейки продуктов.
- Сценарий, согласно которому в результате совместных усилий большей части игроков сообщества создается стандартная или эталонная архитектура, на основе которой впоследствии создаются многочисленные системы.
- Сценарий, согласно которому архитектура, подобно Всемирной паутине, приобретает чуть ли не вселенское распространение, и, соответственно, круг ее разработчиков на порядки перерастает масштабы отдельной организации.

Каждая из перечисленных разновидностей архитектурно-экономического цикла состоит из тех же элементов, что и его первоначальная версия: заинтересованных лиц, технической базы, базы опыта, набора требований, которые необходимо реализовать, архитектора или архитекторов, одного или нескольких вариантов архитектуры, одной или нескольких систем. Вариантность архитектурно-экономического цикла обусловливается коммерческим контекстом, емкостью рынка и поставленных задач.

По нашему мнению, в будущих моделях определения стоимости и эффективности программного обеспечения (в этом смысле CBAM — это только начало) будут предусмотрены все эти версии архитектурно-экономического цикла. В частности, они должны учитывать стратегические инвестиции, в большинстве слу-

чаев связанные с процессом разработки продуктов на основе архитектуры, и прогнозировать количественный эффект создания архитектуры.

19.2. Создание архитектуры

Во всех конкретных примерах мы делали упор на требования по качеству, предъявляемые к конструируемой системе, применяемые архитектором тактики и их архитектурные проявления. При этом процесс перехода от формулировки требований по качеству к архитектурно-проектному решению не может обойтись без проведения подробнейших исследований. Как бы то ни было, а процесс проектирования в его сегодняшнем варианте — это скорее искусство, чем наука, а потому его академическое уточнение обещает дать полезные результаты.

Усовершенствовать процесс проектирования помогут ответы на ряд вопросов.

- *Можно ли считать перечни сценариев реализации атрибутов качества и тактик завершенными?* Мы представили подобные перечни для шести атрибутов качества. Практически во всех случаях их следует дополнять новыми тактиками и сценариями. Кроме того, есть и другие атрибуты качества, которым также соответствуют определенные сценарии и тактики. В частности, не менее важными, чем рассмотренные атрибуты, представляются способность к взаимодействию и легкость построения.
- *Каким образом выполнено сцепление сценариев и тактик?* Сцепление в изложенном материале производится на уровне атрибутов. Иначе говоря, сценарий генерируется согласно таблице генерации для конкретного атрибута — скажем, производительности. Затем проводится анализ тактик, и из них отбираются те, которые с наибольшей вероятностью обеспечат достижение желаемого результата. Несомненно, существуют более совершенные механизмы. Рассмотрим для примера сценарий реализации производительности из примера с открывателем гаражной двери из главы 7 — «При обнаружении препятствия остановка движения двери должна быть выполнена в пределах 0,1 с». Стоит лишь задать несколько вопросов, и проблему отбора тактик можно будет решить более углубленно. Возможны ли обнаружение препятствия и остановка двери за 0,1 с в случае, если остальные элементы системы бездействуют? Если окажется, что такой возможности нет, к алгоритму обнаружения препятствия следует применить тактику «повышение вычислительной эффективности». Если ответ будет положительным, имеет смысл сформулировать ряд вопросов о состязательности. Ответы на них помогут выбрать планировщика. Если в результате исследований удастся оформить системный метод сцепления сценариев и возможных тактик, это будет большой успех.
- *Можно ли прогнозировать результаты применения той или иной тактики?* В сообществе программных инженеров есть свой Святой Грааль, который они активно ищут, — речь идет о возможности прогнозирования атрибутов качества системы до ее фактического конструирования. Один из методов решения этой задачи предполагает прогнозирование воздействия тактик. Применение тактик обусловливается (формальными и неформальными)

аналитическими моделями различных атрибутов. У некоторых результаты вполне предсказуемы. К примеру, одна из тактик реализации модифицируемости предусматривает ведение управляемого конечным пользователем конфигурационного файла. С точки зрения модифицируемости результатом применения этой тактики является сокращение продолжительности изменения и размещения элемента конфигурации. Если первоначально она равна длительности размещения (в случае, если модификацию проводит разработчик), то в конце приближается к нулю (в худшем случае, становится равной продолжительности перезагрузки системы). Это — предсказуемый результат. Разработка подобных методик прогнозирования (равно как и углубление знаний о параметрах, для которых формулируются прогнозы) способна значительно приблизить перспективу создания систем с предсказуемыми атрибутами качества.

- *Каким образом тактики сочетаются в рамках образцов?* В примере с гаражной дверью мы выбрали тактики, а затем они чудесным образом соединились в образце. Здесь, опять же, требуется системный метод сочетания тактик, обеспечивающий прогнозируемость реакций атрибутов качества. Поскольку каждая тактика связана с предсказуемым изменением по части определенного атрибута качества, в рамках образцов можно подбирать компромиссные сочетания этих атрибутов. Вопрос о представлении и сочетании таких прогнозов при объединении тактик в образцы остается открытым.
- *Какие инструментальные средства упрощают проведение процесса проектирования?* По нашему мнению, стандартные блоки со временем будут увеличиваться, выражая более серьезную функциональность и более многочисленные атрибуты качества. Как эта тенденция повлияет на техническую базу? Можно ли будет, к примеру, встраивать тактики и их группировки в составе образцов в экспертные системы-помощники при автоматизации проектирования?
- *Можно ли «вплетать» тактики в системы?* Аспектно-ориентированная разработка программных средств направлена на разработку методов и инструментов реализации так называемых «пересекающихся» требований. Пересекающееся требование применимо сразу к нескольким объектам. К примеру, требование о диагностируемости автомобиля распространяется на все его узлы и, таким образом, пересекает требования, предъявляемые к ним по отдельности. Источниками пересекающихся требований выступают атрибуты качества, а методами реализации конкретных реакций — тактики. Правомерно ли причислять тактики к пересекающимся требованиям и подойдут ли для их удовлетворения методы и средства, разработанные сообществом аспектно-ориентированного программирования?

19.3. Архитектура в рамках жизненного цикла

Согласно нашим утверждениям, архитектура является *основным* артефактом жизненного цикла. В то же время жизненный цикл системы отнюдь не исчерпывается

разработкой архитектуры. Мы считаем, что в контексте взаимодействия архитектуры и жизненного цикла серьезного методического изучения требуют следующие вопросы.

- *Применение инструментальных средств для документирования.* В главе 9 мы говорили об архитектурной документации, однако вопросов ее составления не касались. В идеале, знания об архитектуре системы должны быть встроены в некий инструмент, при помощи которого документация будет генерироваться автоматически или полуавтоматически. Для инструментального составления документации требуется наличие у применяемого инструмента знаний об архитектурных конструкциях — мало того, в нем должен быть заложен метод перехода от одного представления к другому. Последняя задача, в свою очередь, подразумевает существование метода специфицирования соответствия между представлениями.

  Обеспечение соответствия между представлениями невозможно без решения ряда задач. Во-первых, представления должны быть непротиворечивыми — иначе говоря, изменение в одном представлении должно автоматически переходить в другие представления. Во-вторых, необходимо соблюдать индивидуальные и общие для всех представлений ограничения. К примеру, у нас должна быть возможность ограничивать процесс тремя потоками (это ограничение распространяется на отдельное представление) и привязывать определенные модули к одному процессу (ограничение, охватывающее несколько представлений).

- *Программная архитектура в рамках систем управления конфигурациями.* Одна из задач реконструкции программной архитектуры состоит в проверке соответствия фактической архитектуры проектной архитектуре. Предположим, что системе управления конфигурациями известна проектная архитектура, и при входной проверке нового или корректировке существующего кодового модуля архитектуру можно верифицировать на согласованность. В таком случае, поскольку соответствие обеспечивается системой управления конфигурациями, необходимость в отдельной процедуре проверки отпадает. Таким образом, один из мотивов проведения архитектурной реконструкции сходит на нет.

- *Переход от архитектуры к коду.* В случае применения нескольких представлений системы — будь то проектные модели, архитектура или код — их необходимо поддерживать в согласованном состоянии. Обновляемое представление остается корректным, остальные же, если их не согласовывать, со временем теряют всякую ценность. Если в определенной инструментальной среде жесткое сцепление между архитектурой и кодом отсутствует, появляются две дополнительные задачи. Первая заключается в переходе от архитектурной спецификации к коду — по той лишь причине, что этап архитектурного проектирования предшествует этапу кодирования. Вторая задача состоит в том, чтобы обеспечить отражение в архитектуре процесса развития системы, — как правило, именно код, а не архитектура, выступает в качестве обновляемого представления.

19.4. Влияние коммерческих компонентов

Как мы говорили в главе 18, возможности коммерческих компонентов неуклонно возрастают и они становятся все более доступными. Аналогичные тенденции наблюдаются в области предметно-ориентированных вариантов архитектуры и каркасов, облегчающих применение коммерческих компонентов. Справедливы они и в отношении спецификации J2EE, направленной на информационно-технологические варианты архитектуры. Через некоторое время предметно-ориентированные варианты архитектуры и каркасы появятся в большинстве общеупотребительных на сегодняшний день предметных област. Когда это случится, архитекторам придется озаботиться ограничениями, присущими выбранному каркасу, — вероятно, проектирование на основе каркасов будет распространено не меньше, чем индивидуальное проектирование.

Но даже наличие компонентов с расширенной функциональностью не освободит архитектора от решения проектных задач. В первую очередь, архитектор должен определиться со свойствами применяемых компонентов. Компоненты, выражающие архитектурные допущения, требуют идентификации и оценки воздействия на проектируемую систему — все это должен делать архитектор. Для этого нужен богатый выбор атрибутных моделей и серьезные лабораторные исследования, а иногда и то и другое. В поисках надежных прогнозов потребители компонентов обращаются к зарекомендовавшим себя аттестационным агентствам (одно из них — Underwriters Laboratories).

При проектировании систем с участием компонентов от сторонних разработчиков необходимо проводить оценку характеристик качества этих компонентов и каркаса, в рамках которого они существуют. В главе 16 мы рассмотрели ряд вариантов применения архитектуры J2EE/EJB и воздействие каждого из них на безопасность. Как архитектору узнать воздействие альтернативных решений в рамках каркаса и, что еще сложнее, атрибуты качества, реализуемые в безальтернативной ситуации? Необходим метод формулирования присущих компонентам архитектурных допущений и анализа последствий принятия тех или иных решений.

Наконец, требуются новые компоненты и соответствующие им каркасы; производиться они должны в расчете на реализацию желаемых атрибутов качества. Проектировщики должны ориентироваться не на конкретную компанию, а на совокупность заинтересованных лиц в масштабах всей отрасли. Более того, требования по атрибутам качества, сформулированные общими усилиями заинтересованных лиц в индустрии в целом, скорее всего, будут значительно разнообразнее, чем требования заинтересованных лиц в масштабах отдельной компании.

19.5. Заключение

В каком направлении движутся исследования программной архитектуры и их практические результаты? Мы не более проницательны, чем многие другие, что, впрочем, совершенно не означает, что мы воздержимся от прогнозов. Помимо расширения возможностей методик проектирования, развития инструментов жизненного цикла в сторону упрочения позиций архитектурной информации, а также усложнения стандартных блоков компонентов, мы возьмем на себя смелость предположить направление дальнейшего развития архитектуры в целом.

Когда Фреда Брукса однажды спросили, почему его книга «Мифический человеко-месяц» стала хрестоматийной, он ответил в том духе, что книга на самом деле не о компьютерах, а о людях. С программной инженерией — то же самое. Дэйв Парнас очень удачно сформулировал различие между программированием и программной инженерией. По его мнению, для продуктов, которые разрабатывает один человек в единственной версии, программирования вполне достаточно. Однако если вы предполагаете, что с продуктом будут работать сторонние пользователи (или хотя бы хотите впоследствии дать ему самостоятельную оценку), без программной инженерии не обойтись. Те же слова можно сказать и про архитектуру. Если бы наши заботы ограничивались вычислением правильного ответа, хватило бы банальной монолитной архитектуры. Архитектура нужна для удовлетворения потребностей тех людей, которые будут работать с проектируемой системой, — она обеспечивает достаточную производительность, позволяет укладываться в рамки бюджета, достигать желаемых выгод, консолидировать коллектив разработчиков, упрощать задачи специалистов по сопровождению и, наконец, растолковывать суть системы заинтересованным лицам.

Учитывая все эти обстоятельства, мы сделаем прогноз, в котором уверены на все сто. Позиции архитектуры будут сохраняться до тех пор, пока в проектировании и разработке программных средств участвуют живые люди.

ПРОГРАММНАЯ АРХИТЕКТУРА В ОБРАЗОВАНИИ

В этой главе мы обсудили техническую будущность программной архитектуры и изложили наши соображения по поводу ее дальнейшего развития. Но есть и другой вопрос — какое место займет изучение архитектуры в будущих образовательных программах в области программной инженерии? Наблюдательный читатель, вероятно, заметил, что в написании этой книги участвовали три члена семейства Бассов. Я получил степень бакалавра математики в 1964 году, Таня стала бакалавром компьютерных наук в 1991-м, а Мэтт — в 2000-м. Из моего опыта и опыта членов моей семьи можно сделать некоторые выводы.

К моменту получения диплома я видел компьютер один раз в жизни (нас водили на экскурсию лишь затем, чтобы его увидеть). Я абсолютно ничего не смыслил в программировании и принципах работы компьютеров. Естественно, меня сразу взяли на работу программистом. Мир тогда был совсем другим.

Знания, которыми нас пичкают в школе, быстро устаревают, и если мы собираемся заниматься своим делом профессионально на протяжении тридцати или даже сорока лет и при этом хотим оставаться на передовой, необходимо постоянно учиться.

Таня, получив диплом, уже знала несколько языков программирования, в том числе C; курс по C++ в ее время еще не читался, равно как и курсы по основам объектно-ориентированной технологии. Когда выпускником стал Мэтт, он тоже успел изучить несколько языков программирования, но уже других — в частности, C++ и Java. Кроме того, он имел представление об объектно-ориентированном проектировании.

Итак, за девять лет в учебный план вошли объектно-ориентированные языки и соответствующие методики. Мэтт не изучал архитектуру, однако к моменту его выпуска курсы по программной архитектуре успели войти в норму на старших курсах и начали появляться на младших.

Получив образование, Мэтт освоил значительно больше элементов абстракции и проектирования, чем Таня, и эта тенденция, конечно, будет продолжаться. Таким образом, по моему мнению, к 2010 году курсы программной архитектуры на младших курсах станут совершенно обычным явлением, а в некоторых университетах на этом уровне будет предусмотрено сразу несколько дисциплин. Выпускников-специалистов в области программной архитектуры будет уже в достатке.

Мы надеемся на то, что материал, изложенный в этой книге, в 2010 году будет читаться в университетах, и на то, что наш курс программной архитектуры не будет единственным.

— LJB

Сокращения

AAS	(Advanced Automation System) Комплексная система автоматизации — наименование запланированных мероприятий по сплошной реконструкции американской системы управления воздушным движением
ABC	(Architecture Business Cycle) Архитектурно-экономический цикл
ABM	(Atomic Broadcast Manager) Диспетчер элементарной трансляции
ADD	(Attribute Driven Design method) Атрибутный метод проектирования
API	(Application Programming Interface) Интерфейс прикладного программирования
ASEILM	(Automated SEI Licensee Management) Система автоматического управления лицензиатами SEI
AST	(Abstract Syntax Tree) Абстрактно-синтаксическое дерево
ATAM	(Architecture Tradeoff Analysis Method) Метод анализа компромиссных архитектурных решений
ATC	(Air Traffic Control) Управление воздушным движением
BCN	(Backup Communications Network) Резервная сеть передачи данных
CBAM	(Cost Benefit Analysis Method) Метод анализа стоимости и эффективности
C&C	(Component-and-Connector) Компонент и соединитель — класс проекций
CDL	(Chart Description Language) Язык описания схем
CERN	(European Laboratory for Particle Physics) Европейская лаборатория ядерных исследований
CGI	(Common Gateway Interface) Общий шлюзовой интерфейс
COCOMO	(Constructive Cost Modeling) Конструктивное стоимостное моделирование
COOB	(Common Object Manager) Универсальный менеджер объектов
CORBA	(Common Object Request Broker Architecture) Обобщенная архитектура построения брокеров объектных запросов
COSE	(Common Operating System Environment) Общее операционное окружение

COTS (Commercial Off-the-Shelf) Коммерческий коробочный продукт — обозначение программного обеспечения и компонентов, представленных на рынке в готовом виде

CPU (Central Processing Unit) Центральный процессор

CSC (Computer Software Components) Компоненты компьютерных программ

CSCI (Computer Software Configuration Item) Элемент конфигурации компьютерных программ — компонент программного обеспечения

CSCW (Computer Supported Cooperative Work) Совместная работа на базе ЭВМ

C3 (Command, Control, and Communications) Командование, управление и связь

DAWG (Data Access Working Group) Рабочая группа по доступу к данным

DBMS (Database Management Systems) Система управления базами данных

DMZ (Demilitarized Zone) Демилитаризованная зона

DSRGM (Decision Support and Report Generation Manager) Диспетчер принятия решений и составления отчетов

ECS (Earth Core System) Центральная информационная система наблюдения за поверхностью Земли

EDARC (Enhanced Direct Access Radar Channel) Расширенный радиолокационный канал прямого доступа в составе системы ISSS

EFC (EDARC Format Conversion) Программа преобразования форматов EDARC — приложение в составе элемента управления выводом данных на экран

EIS (EDARC Interface Software) Интерфейсная программа EDARC — приложение в составе элемента «Общие системные службы»

EJB (Enterprise JavaBeans) Система корпоративных JavaBeans, архитектурная спецификация Enterprise JavaBeans

EOS (Earth Observing System) Система наблюдения за поверхностью Земли

EOSDIS (Earth Observing System Data System Information System) Информационная система наблюдения за поверхностью Земли

ESI (External System Interface) Внешний системный интерфейс

ESIP (ESI Processor) Процессор внешнего системного интерфейса

FAA (Federal Aviation Administration) Федеральное авиационное агентство США, заказчик системы ISSS

FAR (Federal Acquisition Regulations) Федеральные правила приобретений

FG (Functional Group) Функциональная группа — приложение, которое в составе системы ISSS не проявляет отказоустойчивость (то есть не является операционным блоком)

FIFO (First-In/First-Out) Первым пришел — первым обслужен

FTP (File Transfer Protocol) Протокол передачи файлов

GIOP (General Inter-ORB Protocol) Универсальный межброкерный протокол

GUI (Graphical User Interface) Графический пользовательский интерфейс

HCI	(Human-Computer Interface) Человеко-машинный интерфейс
HCIS	(Host Computer Interface Software) Интерфейсная программа базового компьютера — приложение в составе элемента «Общие системные службы» системы ISSS
HCS	(Host Computer System) Базовый компьютер — центральная вычислительная машина системы управления воздушным движением
HTML	(HyperText Markup Language) Язык разметки гипертекста
HTTP	(HyperText Transfer Protocol) Протокол передачи гипертекста
HTTPS	(HyperText Transfer Protocol Secure) Протокол защищенной передачи гипертекста
I/O	(Input/Output) Ввод-вывод
IAPR	(Interactive Architecture Pattern Recognition) Система интерактивного распознавания архитектурных образцов
IDE	(Integrated Development Environment) Интегрированная среда разработки
IDL	(Interface Definition Language) Язык описания интерфейсов
IEEE	(Institute of Electrical and Electronics Engineers) Институт инженеров по электротехнике и электронике
IIOP	(Internet Inter-ORB Protocol) Межброкерный протокол Интернета
IMS	(Inertial Measurement System) Инерциальная система измерений
IP	(Internet Protocol) Интернет-протокол
ISO	(International Organization for Standardization) Международная организация по стандартизации
ISSS	(Initial Sector Suite System) Основная система контроля секторов — система, которая, согласно первоначальным планам, должна была устанавливаться во всех транзитных центрах управления воздушным движением; соответствующий конкретный пример рассматривается в главе 6
ISV	(Independent Software Vendor) Независимый поставщик программного обеспечения
IT	(Information Technology) Информационная технология
JDBC	(Java Database Connectivity) Java-интерфейс связи с базами данных
JMS	(Java Messaging Service) Служба сообщений Java
JNDI	(Java Naming and Directory Interface) Java-интерфейс именования и каталогов
JSP	JavaServer Pages
J2EE	(Java 2 Enterprise Edition) Корпоративная архитектура Java 2
JTS	(Java Transaction Service) Служба транзакций Java
JVM	(Java Virtual Machine) Виртуальная машина Java
KSLOC	(Kilos of Source Lines of Code) ...тысяч строк исходного кода — стандартная единица измерения статического размера компьютерной программы

KWIC (Keyword in Context) Ключевое слово в контексте

LAN (Local Area Network) Локальная сеть

LCN (Local Communications Network) (Основная) локальная сеть передачи данных

LGSM (Local/Group SMMM) локальная/групповая SMMM — приложение в составе элемента «Общие системные службы»

LIU (LCN Interface Unit) Интерфейсный блок локальной сети передачи данных

M&C (Monitor and Control) Мониторинг и управление — тип консоли в системе ISSS

MIPS (Million Instructions per Second) Миллион команд в секунду

MODN (Noise Model) Шумовая модель

MODP (Prop Loss Model) Модель потери опоры

MODR (Reverb Model) Модель отражения

MRI (Magnetic Resonance Imaging) Отображение магнитного резонанса, МР-отображение

MVC (Model-View-Controller) Образец «модель–представление–контроллер»

NASA (National Aeronautics and Space Administration) Национальный комитет по аэронавтике и исследованиям космического пространства, НАСА

NASM (National Airspace System Modification) Модификация национальной воздушно-космической системы — один из элементов конфигурации системы ISSS

NAT (Network Address Translation) Трансляция сетевых адресов

NISL (Network Interface Sublayer) Подуровень сетевых интерфейсов

NIST (National Institute of Standards and Technology) Национальный институт стандартов и технологии

NNTP (Network News Transport Protocol) Сетевой протокол передачи новостей

NRL (Naval Research Laboratory) Научно-исследовательская лаборатория ВМС США

OLE (Object Linking and Embedding) Связывание и внедрение объектов

OLTM (OnLine Transaction Manager) Диспетчер оперативных транзакций

OMA (Object Management Architecture) Архитектура управления объектами

OMG (Object Management Group) Рабочая группа по объектному менеджменту

ORB (Object Request Broker) Брокер объектных запросов

PAC (Presentation-Abstraction-Control) Образец «представление–абстракция–управление»

PAS (Primary Address Space) Основное адресное пространство — копия приложения, которое фактически исполняет все функции ISSS; *см. также* SAS

PCTE (Portable Common Tools Environment) Переносимые универсальные инструменты

PDF (Portable Document Format) Формат портативных документов

PICS (Platform for Internet Content Selection) Платформа отбора информации в Интернете

PMS (Prepare Messages) Подготовка сообщений — приложение в составе элемента «Общие системные службы» системы ISSS

RCS (Revision Control System) Система управления пересмотром проектных решений

RISC (Reduced Instruction Set Chip) Микросхема с сокращенным набором команд

RMI (Remote Method Invocation) Удаленный вызов методов

ROOM (Real-Time Object-Oriented Modeling) Объектно-ориентированное моделирование в реальном времени

RPC (Remote Procedure Call) Удаленный вызов процедуры

RUP (Rational Unified Process) Рациональный унифицированный процесс

SAAM (Software Architecture Analysis Method) Метод анализа программной архитектуры

SAR (System Analysis And Recording) Анализ и регистрация операций системы — функция системы ISSS, а также приложение в рамках функций записи, анализа и воспроизведения

SAS (Standby (Secondary) Address Space) Резервное (вторичное) адресное пространство — резервная копия приложения в системе ISSS, заменяющее PAS в случае его отказа

SCR (Software Cost Reduction) Проект по снижению издержек производства программного обеспечения систем ВМС США

SEI (Software Engineering Institute) Институт программной инженерии

SIMD (Single Instruction, Multiple Data) Архитектура с одним потоком команд и множеством потоков данных

SLOC (Source Lines of Code) ...строк исходного кода

SMMM (System Monitor And Mode Management) Программа системного мониторинга и управления режимами

SQL (Structured Query Language) Язык структурированных запросов

SSL (Secure Sockets Layer) Протокол защищенных сокетов

TAFIM (Technical Architecture for Information Management) Техническая архитектура управления информацией

TARGET (Theater-Level Analysis, Replanning and Graphical Execution Toolbox) Инструментальный комплекс анализа операций на театре военных действий, оперативного планирования и графического оформления

TCA (Terminal Control Area) Узловой диспетчерский район

TCP (Transmission Control Protocol) Протокол управления передачей

TCP/IP	(Transmission Control Protocol/Internet Protocol) Протокол управления передачей/Интернет-протокол
UDDI	(Universal Description, Discovery, and Integration) Универсальная система предметного описания и интеграции
UI	(User Interface) Пользовательский интерфейс
UML	(Unified Modeling Language) Унифицированный язык моделирования
URL	(Uniform Resource Locator) Унифицированный указатель ресурса
VPN	(Virtual Private Network) Виртуальная частная сеть
W3C	(World Wide Web Consortium) Консорциум Всемирной паутины
WAIS	(Wide Area Information Service) Глобальный информационный сервер
WAP	(Wireless Application Protocol) Беспроводной прикладной протокол
WIMP	(Window, Icon, Mouse, Pointer) Окна, пиктограммы, мышь, указатели (интерфейс)
WWW	(World Wide Web) Всемирная паутина
XML	(eXtensible Markup Language) Расширяемый язык разметки

Библиография

Abowd 93	Abowd, G., Bass, L., Howard, L., Northrop, L. "Structured Modeling: An O-O Framework and Development Process for Flight Simulators," CMU/SEI-1993-TR-14. Software Engineering Institute, Carnegie Mellon University, 1993.
Abowd 96	Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., Zaremski, A. "Recommended Best Industrial Practice for Software Architecture Evaluation," Technical Report CMU/SEI-96-TR-025. Software Engineering Institute, Carnegie Mellon University, 1996.
Alur 01	Alur, D., Crupi, J., Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies.* Sun Microsystems Press, 2001.
Alexander 77	Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S. *A Pattern Language.* Oxford University Press, 1977.
America 00	America, P., Obbink, H., van Ommering, R., van der Linden, F. "CoPAM: A Component-Oriented Platform Architecture Method Family for Product Family Engineering," in *Software Product Lines: Experience and Research Directions* (P. Donohoe, ed.). Kluwer, 2000.
Anastasopoulos 00	Anastasopoulos, M., Gacek, C. "Implementing Product Line Variability," IESE report 89.00/E, v. 1.0. Fraunhofer Institut Experimentelles Software Engineering, 2000.
ASCYW 94	ASCYW. *Structural Modeling Handbook.* Air Force Aeronautical Systems Command, 1994.
Asundi 01	Asundi, J., Kazman, R., Klein, M. "Using Economic Considerations to Choose amongst Architecture Design Alternatives, CMU/SEI-2001-TR 035. Software Engineering Institute, Carnegie Mellon University, 2001.
AT&T 93	AT&T. "Best Current Practices: Software Architecture Validation." Internal report, © 1991. AT&T, 1993.
Bachmann 02	Bachmann, F., Bass, L., Klein, M. "Illuminating the Fundamental Contributors to Software Architecture Quality," SEI/CMU-2002-TR-025. Software Engineering Institute, Carnegie Mellon University, 2002.

Barfield 01 — Barfield, W., Caudell, T. (eds.). *Fundamentals of Wearable Computers and Augmented Reality*. Lawrence Erlbaum Associates, 2001.

Bass 00 — Bass, L., Clements, P., Donohoe, P., McGregor, J., Northrop, L. "Fourth Product Line Practice Workshop Report," CMU/SEI-2000-TR-002. Software Engineering Institute, Carnegie Mellon University, 2000.

Bass 01a — Bass, L., John, B., Kates, J. "Achieving Usability through Software Architecture," CMU/SEI-2001-TR-005. Software Engineering Institute, Carnegie Mellon University, 2001.

Bass 01b — Bass, L., Klein, M., Moreno, G. "Applicability of General Scenarios to the Architecture Tradeoff Analysis Method," CMU/SEI-2001-TR-014. Software Engineering Institute, Carnegie Mellon University, 2001.

Berners-Lee 1996a — Berners-Lee, T. *WWW Journal* 3 (http://www.w3.org/pub/WWW/Journal), 1996.

Berners-Lee 1996b — Berners-Lee, T. "WWW: Past, Present, Future," *IEEE Computer*, October 1996.

Boehm 76 — Boehm, B., Brown, J., Lipow, M. "Quantitative Evaluation of Software Quality," *Proceedings of the Second International Conference on Software Engineering*. IEEE Computer Society, 1976.

Boehm 81 — Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

Boehm 95 — Boehm, B. "Engineering Context," *Proceedings of the First International Workshop on Architectures for Software Systems*. См. также CMU-CS-TR-95-151, School of Computer Science, Carnegie Mellon University, April 1995.

Booch 94 — Booch, G. *Object-Oriented Design with Applications, Second Edition*. Benjamin-Cummings, 1994.

Bosch 00a — Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

Bosch 00b — Bosch, J. "Organizing for Software Product Lines," *Proceedings of the Third International Workshop on Software Architectures for Product Families*. Springer LNCS, 2000.

Bowman 99 — Bowman, T., Holt, R., Brewster, N. "Linux as a Case Study: Its Extracted Software Architecture," *Proceedings of the Second International Conference on Software Engineering*. ACM Press, 1999.

Brand 97 — van den Brand, M., Sellink, M., Verhoef, C. "Generation of Components for Software Renovation Factories from Context-Free Grammars," *Proceedings of the Fourth Working Conference on Reverse Engineering*. ACM Press, 1997.

Briand 99 — Briand, L., Daly, J., Wust, J. "A Unified Framework for Coupling Measurements in Object-Oriented Systems," *IEEE Transactions of Software Engineering* 25(1), 1999.

Britton 81 — Britton, K., Parnas, D. "A-7E Software Module Guide," NRL Memorandum Report 4702, December 1981.

Brooks 69	Brooks, F., Iverson, K. *Automatic Data Processing (System 360 Edition)*. John Wiley, 1969.
Brooks 75	Brooks, F. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.
Brooks 95	Brooks, F. *The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition)*. Addison-Wesley, 1995.
Brown 95	Brown A., Carney, D., Clements, P. "A Case study in Assessing the Maintainability of a Large, Software-Intensive System," *Proceedings of the International Symposium on Software Engineering of Computer-Based Systems*. IEEE Computer Society, 1995.
Brownsword 96	Brownsword, L., Clements, P. "A Case Study in Successful Product Line Development," CMU/SEI-96-TR-016. Software Engineering Institute, Carnegie Mellon University, 1996.
Buschmann 96	Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley, 1996.
Bush 45	Bush, V. "As We May Think," *Atlantic Monthly*, July 1945.
CACM 88	Special Issue: HyperText Systems. *Communications of the ACM*, July 1988.
Cederling 92	Cederling, U. "Industrial Software Development: A Case Study," Ph. D. thesis. Linkoping University (Linkoping, Sweden), 1992.
Chastek 96	Chastek, G., Brownsword, L. "A Case Study in Structural Modeling," CMU/SEI-1996-TR-35, ESC-1996-TR-025. Software Engineering Institute, Carnegie Mellon University, 1996.
Chretienne 95	Chretienne, P., Lenstra, J., Coffman, E. (eds.). *Scheduling Theory and Its Applications*. John Wiley, 1995.
Chung 00	Chung, L., Nixon, B., Yu, E., Mylopoulos, J. *Non-Functional Requirements in Software Engineering*. Kluwer, 2000.
Clements 02a	Clements, P., Kazman, R., Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.
Clements 02b	Clements, P., Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
Clements 03	Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.
Conway 68	Conway, M. "How Do Committees Invent?" *Datamation* 14(4), 1968.
Cristian 93	Cristian, F. "Understanding Fault-Tolerant Distributed Systems" (ftp.cs.ucsd.edu/pub/tech-reports/understandingftsystems.ps.Z), 1993.
Cusumano 95	Cusumano, R., Selby, R. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995.

Dijkstra 68 — Dijkstra, E. "The Structure of the 'T.H.E.' Multiprogramming System," *Communications of the ACM* 18(8), 1968.

Fielding 96 — Fielding, R., Whitehead, E., Anderson, K., Bolcer, G., Oreizy, P., Taylor, R. "Software Engineering and the WWW: The Cobbler's Barefoot Children, Revisited," Technical Report 96-53. Department of Information and Computer Science, University of California, Irvine, November, 1996.

Fogarty 67 — Fogarty, L. "Survey of Flight Simulation Computation Methods," *Proceedings of the Third International Simulation and Training Conference*. Society for Computer Simulation, 1967.

Gamma 95 — Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Garlan 95 — Garlan, D., Allen, R., Ockerbloom, J. "Architectural Mismatch: Or Why It's Hard to Build Systems out of Existing Parts," *Proceedings of the Seventeenth International Conference on Software Engineering*. ACM Press, 1995.

Gibbs 94 — Gibbs, W. "Software's Chronic Crisis," *Scientific American*, September 1994.

Glass 98 — Glass, R, "Editorial," *Journal of Systems and Software* (Elsevier Science) 43(3): 161-163, 1998.

Gram 96 — Gram, C., Cockton, G. *Design Principles for Interactive Software*. Chapman & Hall, 1996.

Guo 99 — Guo, G., Atlee, J., Kazman, R. "A Software Architecture Reconstruction Method," Report No. WICSA1. *Proceedings of the First Working IFIP Conference on Software Architecture*. Kluwer, 1999.

Hager 89 — Hager, J. "Software Cost Reduction Methods in Practice," *IEEE Transaction on Software Engineering* 15, 1989.

Hager 91 — Hager, J. "Software Cost Reduction Methods in Practice: A Post-Mortem Analysis," *Journal of Systems Software* 14, 1991.

Harris 95 — Harris, D., Reubenstein, H., Yeh, A. "Reverse Engineering to the Architectural Level," *Proceedings of the Seventeenth International Conference on Software Engineering*. ACM Press, 1995.

Hassan 00 — Hassan, A., Holt, R. "A Reference Architecture for Web Servers," *Proceedings of the Working Conference on Reverse Engineering*. IEEE Computer Society, 2000.

Hoffman 01 — Hoffman, D., Weiss, D. (eds). *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

Hofmeister 00 — Hofmeister, C., Nord, R., Soni, D. *Applied Software Architecture*. Addison-Wesley, 2000.

IEEE 00 — The Institute of Electrical and Electronics Engineers Standards Board. *Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE-Std-1471-2000, September 2000.

ISO 91 — *International Standard ISO/IEC 9126. Information Technology: Software Product Evaluation — Quality Characteristics and Guidelines for Their Use.* International Organization for Standardization/International Electrotechnical Commission, Geneva, 1991.

Jacobson 97 — Jacobson, I., Griss, M., Jonsson, P. *Software Reuse: Architecture, Process, and Organization for Business Success.* Addison-Wesley, 1997.

Jalote 94 — Jalote, P. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

Jones 99 — Jones, T. Capers. *Estimating Software Costs.* McGraw-Hill, 1999.

Kazman 94 — Kazman, R., Bass, L., Abowd, G., Webb, M. "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proceedings of the Sixteenth International Conference on Software Engineering.* ACM Press, 1994.

Kazman 99a — Kazman, R., Carriere, S. "Playing Detective: Reconstructing Software Architecture from Available Evidence," *Journal of Automated Software Engineering* 6(2), April 1999.

Kazman 99b — Kazman, R., Barbacci, M., Klein, M., Carriere, S., Woods, S. "Experience with Performing Architecture Tradeoff Analysis," *Proceedings of the Twenty-First International Conference on Software Engineering.* ACM Press, 1999.

Kazman 01 — Kazman, R., Asundi, J., Klein, M. "Quantifying the Costs and Benefits of Architectural Decisions," *Proceedings of the Twenty-Third International Conference on Software Engineering.* IEEE Computer Society, 2001.

Krikhaar 99 — Krikhaar, R. *Software Architecture Reconstruction.* Ph.D. thesis, University of Amsterdam, 1999.

Kruchten 95 — Kruchten, P. "The 4+1 View Model of Architecture," *IEEE Software* 12(6), 1995.

Kruchten 00 — Kruchten, P. *The Rational Unified Process: An Introduction, Second Edition.* Addison-Wesley, 2000.

Laprie 89 — Laprie, J. *Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance* (T. Anderson, ed.). Blackwell Scientific, 1989.

Lee 88 — Lee, K., Rissman, M., D'Ippolito, R., Plinta, C., van Scoy, R. *An OOD Paradigm for Flight Simulators, Second Edition*, CMU/SEI-1988-TR-30. Software Engineering Institute, Carnegie Mellon University, 1988.

Marsman 85 — Marsman, A. "Flexible and High-Quality Software on a Multi-Processor Computer System Controlling a Research Flight Simulator," *AGARD Conference Proceedings No. 408: Flight Simulation* 9(1), 1985.

McConnell 96 — McConnell, S. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, 1996.

McGregor 01	McGregor, J., Sykes, D. *A Practical Guide for Testing Object-Oriented Software.* Addison-Wesley, 2001.
Menasce 00	Menasce, D., Almeida, V. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall, 2000.
Morris 93	Morris, C., Fergubor, C. "How Architecture Wins Technology Wars," *Harvard Business Review*, 71(March-April): 86-96, 1993.
Muller 93	Muller, H., Mehmet, O., Tilley, S., Uhl, J. "A Reverse Engineering Approach to System Identification," *Journal of Software Maintenance: Research and Practice* 5(4), 1993.
Parnas 71	Parnas, D. "Information Distribution Aspects of Design Methodology," *Proceedings of the 1971 IFIP Congress*, North Holland, 1971.
Parnas 72	Parnas, D. "On the Criteria for Decomposing Systems into Modules," *Communications of the ACM* 15(12), 1972.
Parnas 74	Parnas, D. "On a 'Buzzword': Hierarchical Structure," *Proceedings of the 1974 IFIP Congress.* Kluwer, 1974.
Parnas 76	Parnas, D. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, SE-2(1), 1976.
Parnas 79	Parnas, D. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering* SE-5(2), 1979.
Parnas 85a	Parnas, D., Clements, P., Weiss, D. "The Modular Structure of Complex Systems," *Proceedings of the Seventh International Conference on Software Engineering.* Reprinted in *IEEE Transactions on Software Engineering* SE-11, 1985.
Parnas 85b	Parnas D., Weiss, D. "Active Design Reviews: Principles and Practices," *Proceedings of the Eighth International Conference on Software Engineering*, 1985.
Paulish 02	Paulish, D. *Architecture-Centric Software Project Management.* Addison-Wesley, 2002.
Perry 66	Perry, D., Warton, L., Welbourn, C. "A Flight Simulator for Research into Aircraft Handling Characteristics," Report No. 3566. Aeronautical Research Council Reports and Memoranda, 1966.
Pfaff 85	Pfaff, G. (ed.). *User Interface Systems.* Eurographics Seminars, Springer-Verlag, 1985.
Ramachandran 02	Ramachandran, J. *Designing Security Architecture Solutions.* John Wiley, 2002.
Rissman 90	Rissman, M., D'Ippolito, R., Lee, K., Steward, J. "Definition of Engineering Requirements for AFECO: Lessons from Flight Simulators," CMU/SEI-1990-TR-25. Software Engineering Institute, Carnegie Mellon University, 1990.
Rumbaugh 99	Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

Schmidt 00	Schmidt, D., Stal, M., Rohnert, H., Buschmann, F. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley, 2000.
Seacord 99	Seacord, R., Wallnau, K., Robert, J., Comella-Dorda, S., Hissam, S. "Custom vs. Off-the-Shelf Architecture," *Proceedings of the Third International Enterprise Distributed Object Computing Conference*, 1999.
SEI ATA	См. http://www.sei.cmu.edu/ata/ata_init.html.
Shaw 96	Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
Sneed 98	Sneed, H. "Architecture and Functions of a Commercial Software Reengineering Workbench," *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering.* IEEE Computer Society, 1998.
Soni 95	Soni, D., Nord, R., Hofmeister, C. "Software Architecture in Industrial Applications," *Proceedings of the Seventeenth International Conference on Software Engineering.* ACM Press, 1995.
Stallings 99	Stallings, W. *Cryptography and Network Security: Principles and Practice, Third Edition.* Prentice Hall, 1999.
Stonebraker 90	Stonebraker, M., Rowe, L., Hirohama, M. "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering* 2(1), 1990.
Svahnberg 00	Svahnberg, M., Bosch, J. "Issues Concerning Variability in Software Product Lines," *Proceedings of the Third International Workshop on Software Architectures for Product Families.* Springer LNCS, 2000.
UIMS 92	UIMS Tool Developers Workshop. "A Metamodel for the Runtime Architecture of an Interactive System," *SIGCHI Bulletin* 24(1), 1992.
Wallnau 02	Wallnau, K., Hissam, S., Seacord, R. *Building Systems from Commercial Components.* Addison-Wesley, 2002.
Weiss 00	Weiss, D., Lai, C. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 2000.
Witt 94	Witt, B., Baker, F., Merritt, E. *Software Architecture and Design.* Van Nostrand Reinhold, 1994.
Wong 94	Wong, K., Tilley, S., Muller, H., Storey, M. "Programmable Reverse Engineering," *International Journal of Software Engineering and Knowledge Engineering* 4(4), December 1994.

Алфавитный указатель

A

A-7, проект, 86, 87, 88
A-7E, авиационная система, 81, 83, 84, 85, 86, 87, 88, 89, 90, 91, 95, 97, 98, 99, 100, 101
  архитектурно-экономический цикл, 82
  атрибуты качества
    модифицируемость, 86
    производительность, 86
    функционирование в реальном времени, 85
  библиография, 103
  бортовой компьютер, 83
  воздушный зонд, 84
  датчики, 84
  доплеровская РЛС, 84
  инерциальная система измерений (IMS), 84
  карта текущего местоположения, 84
  клавишная панель, 85
  многоуровневость, 97
  оценка успешности, 100
  проекционный бортовой индикатор, 85
  радиолокационный высотомер, 84
  РЛС переднего обзора, 84
  структура декомпозиции на модули, 88, 90, 92, 94
    интерфейсы, 89
    информационная закрытость, 89
    модуль генерации системы, 92
    модуль обслуживающих программ, 92
    модуль прикладных типов данных, 91
    модуль программных решений, 91
    модуль расширения компьютера, 91
    модуль сокрытия аппаратного обеспечения, 91
    модуль сокрытия поведения, 91
    модуль физических моделей, 92
    модуль фильтрации поведения, 92
A-7E, авиационная система (*продолжение*)
    модуль-банк данных, 91
    процедуры доступа, 89
    руководство по модулям, 90
  структура использования, 88, 94, 95, 97
    образец, 96
    отношение использования, 94, 95
    структура разрешения использования, 95, 97
  структура процессов, 88, 98, 99, 100
A-7E, авиационная электронная система, 523
Ada, язык программирования, 171, 172, 173, 178, 181, 182, 184, 188, 423, 430, 432, 439, 441, 444, 446, 447, 452, 524
Amazon.com, электронный магазин, 397
America Online (AOL), компания, 515
AOL, компания, 396
AOLbyPhone, приложение, 515
ASEILM, система управления лицензиатами, 527, 530, 531, 534, 536, 538
  Java-сервлетов ансамбль, 534, 536, 538
    безопасность, 536
    новые требования, 536
    параллелизм, 537
    решение модельной задачи, 534
  Miva Empressa, ансамбль, 529, 530, 532
    компоненты, 530
    оценка модельного решения, 534
    решение модельной задачи, 531, 532, 533
  задачи, 528
  типы пользователей, 528
  требования по атрибутам качества, 529
ATAM (метод анализа архитектурных решений), 196, 354, 356, 357, 362, 365, 372
  библиография, 373
ATAM (метод анализа компромиссных решений), 47, 316, 317, 319, 322, 323, 325, 326, 329, 334, 335, 337, 338, 343, 347, 350, 527
  библиография, 353

ATAM (метод анализа компромиссных решений) (*продолжение*)
вторичные результаты, 320
продукты
компактная презентация архитектуры, 319
магистральные риски, 320
нерискованные решения, 320
рискованные решения, 320
соответствие архитектурных решений требованиям, 320
составление сценариев, 320
точки компромиссов, 320
точки чувствительности, 320
формулировка коммерческих задач, 320
роли, 334
дознаватель, 319
координатор процесса, 319
наблюдатель за процессом, 319
руководитель группы, 318
руководитель специалистов по оценке, 318
секретарь по результатам, 318
секретарь по сценариям, 318
хронометрист, 319
участники
группа оценки, 317
заинтересованные лица, 317
ответственные лица, 317
этап 0 — подготовка, 321, 335, 336, 337
этап 1 — оценка
операция 1 — презентация ATAM, 322, 337
операция 2 — презентация коммерческих факторов, 323, 337
операция 3 — презентация архитектуры, 323, 324, 339, 340
операция 4 — выявление архитектурных методик, 324, 341
операция 5 — генерация дерева полезности, 325, 342
операция 6 — анализ архитектурных методик, 328, 345
этап 2 — оценка (продолжение), 329, 347, 348, 349
операция 8 — анализ архитектурных методик, 331, 350
операция 9 — презентация результатов, 332, 350, 351
этап 3— доработка, 322, 351

B

Boeing, компания, 403

C

C, язык программирования, 548
C++, язык программирования, 519, 548
CBAM
контекст, 356
метод анализа стоимости и эффективности, 543
полезность, 357
расчет общей выгоды от архитектурной стратегии, 364, 371
реактивно-полезностные кривые реакция-полезность, 358
CBAM (метод анализа стоимости и эффективности), 47, 353, 355, 357, 361, 362, 364, 365, 367, 370, 371, 372, 373, 414
библиография, 373
контекст, 355
полезность, 357
архитектурные стратегии, 360
вес сценария, 360
нормализация, 360
побочных реакций атрибута, 360
уровни, 359, 360
формула расчета выгоды, 360
результаты, 372
CDL, язык описания диаграмм, 532
CelsiusTech, компания, 405, 421, 423, 426, 429, 431, 432, 435, 436, 438, 440, 442, 444, 446, 447, 449, 450, 451, 453
архитектурно-экономический цикл (ABC), 422, 423, 426, 429, 430, 432, 435, 438
библиография, 454
конкретный пример построения линейки продуктов, 421, 423, 426, 429, 431, 432, 435, 436, 438, 440, 442, 444, 446, 447, 449, 450, 451, 453
мотивы построения линейки продуктов, 428, 429, 430
организационная структура, 433, 435, 438, 439, 441
CERN, Европейская лаборатория ядерных исследований, 375, 378, 379, 382, 396
CGI (общий шлюзовой интерфейс), 387
put, операция, 388
безопасность, 388

CGI (общий шлюзовой интерфейс) (*продолжение*)
виртуальные документы, 387
сценарии, 388
ChartWorks, система графического отображения, 530, 532, 533, 536
COM+
компонентная модель от Microsoft, 481
cookie, элементы, 536
CORBA, архитектура, 456, 481, 519, 522
COTS (коммерческие коробочные продукты), 132
CSC (компоненты компьютерных программ), 71
CSCI (элементы конфигурации компьютерных программ), 70
Cummins, компания, 404, 410

D

Dali, инструментарий, 281, 287, 290, 294, 296, 306
Dell, компания, 403
DMZ (демилитаризованная зона), 157

E

echo, пакет, 139
EDLC (эволюционный жизненный цикл поставки), 98
EJB, реализация требований по атрибутам качества, 465
EJB (система корпоративных JavaBeans), 394
EJB, спецификация, 455, 457, 458, 460, 462, 463, 469, 472, 475, 480, 481, 502, 506, 509
beans, 464, 465, 468, 469, 472, 474, 475, 480, 481
сеансовые, 464
состояние, 464
сущности, 465, 476, 478
устойчивость, 465
beans-сущности, 502
библиография, 481
дескрипторы размещения, 472, 473
задачи, 458, 459
компонентная модель, 463, 475
контейнеры, 463, 465, 469, 471, 472, 473, 475, 477, 479, 480
масштабирование, 478
вертикальное, 478
горизонтальное, 478
программирование, 468, 469, 470, 472
EJB, спецификация (*продолжение*)
пул ресурсов, 479
распределение, 478, 479
распределенные транзакции, 479
сеансовые beans без состояния, 502
сеансовые beans с запоминанием состояния, 502
состояние, 475

F

Ford, компания, 403
Fortran, язык программирования, 519
FTP (протокол передачи файлов), 384

G

Gopher, протокол, 384

H

heartbeat, пакет, 139
Hewlett-Packard, компания, 404
HTML (язык разметки гипертекста), 376, 382, 391
HTTP, протокол передачи гипертекста, 382, 533, 536, 538
HTTPS (протокол защищенной передачи гипертекста), 380, 384, 388, 392, 394, 531, 533, 536

I

IBM, компания, 83
IDL, язык описания интерфейсов, 519
Inmedius, компания, 483, 488, 489, 493, 494, 497, 501, 503, 510
история, 483, 484
организационная структура, 487
принципы коммерческой деятельности, 488
Internet Explorer, браузер, 525
Internet Information Server (IIS), веб-сервер, 530, 532, 537
ISSS (основная система контроля секторов), 169, 170, 172, 174, 176, 177, 178, 179, 181, 182, 184, 185, 187, 188, 189, 191, 192
BCN (резервная сеть передачи данных), 177
CSCI (элементы конфигурации компьютерных программ), 177
EDARC (канал прямого доступа), 176

ISSS (основная система контроля секторов) (*продолжение*)
Ethernet, стандарт, 177
LCN (локальная сеть передачи данных), 176, 177
адресные пространства, 179
архитектурно-экономический цикл (ABC), 170
архитектурное решение, 174, 176, 177, 178, 179, 180, 181, 184, 185, 187, 188, 189
атрибуты качества, 171
библиография, 192
блоки управления секторами, 172, 173
взаимоотношения представлений, 187, 188
группа процессоров, 178
диспетчер элементарной трансляции (ABM), 184
задачи, 174
заинтересованные лица, 172
клиент-серверное представление, 181
кодовое представление, 181, 182
CSCI (элементы конфигурации компьютерных программ), 188
задачи, 182
пакетирование, 182
пакеты, 182
подпрограммы, 182
многоуровневое представление, 182, 184
операционный блок, 179
представление декомпозиции модулей, 177, 178
представление отказоустойчивости, 185, 187
представление процессов, 178, 179, 180
программа подготовки сообщений, 184
тактики
адаптационные данные, 188, 189
кодовые шаблоны для приложений, 189, 190
общие абстрактные службы, 189, 190
требования
возможность дополнения функциональности, 172
высокая производительность, 172
открытость, 172
сверхвысокая готовность, 171
способность к взаимодействию и сопряжению, 172
способность к выделению подмножеств, 172
физическое представление, 175, 176, 177
функциональные группы, 179
хост-компьютер, 174

J

J2EE, архитектурная спецификация, 546
J2EE, спецификация, 455, 457, 458, 460, 462, 463, 469, 472, 475, 480, 481, 484
J2EE/EJB, архитектура, 455, 460, 463, 469, 472, 475, 480, 481
EJB, архитектурная методика, 463
архитектурное решение, 460, 462, 463, 470, 472
атрибуты качества, 457, 458
библиография, 481
звенья, 462
веб-звено, 462
звено бизнес-компонентов, 462
звено клиента, 462
звено корпоративных информационных систем, 463
и архитектурно-экономический цикл (ABC), 456
и Всемирная паутина, 458
компоненты и службы
JNDI, интерфейс, 468
требования, 457, 458
характеристики, 460
Java, язык программирования, 455, 457, 458, 460, 462, 463, 472, 475, 480, 481, 518, 534, 537, 548
ансамбль Java-сервлетов, 538
виртуальная машина Java (JVM), 457, 463, 480
удаленный вызов методов (RMI), 457, 478

L

Luther, архитектура, 483, 488, 489, 493, 494, 497, 501, 503, 510
архитектурно-экономический цикл (ABC), 485, 486, 488
опыт разработчиков, 488
организационная структура компании-разработчика, 487
принципы коммерческой деятельности, 488
технологическая база, 487
архитектурное решение, 492
HTTP, протокол, 492
зависимость от J2EE, 510, 511
коммерческие компоненты, 492
компонент технологического управления, 504, 505, 507, 509

Luther, архитектура (*продолжение*)
компоненты, 500, 501, 502
парадигма пользовательского опыта, 492
пользовательский интерфейс, 494, 495, 498
приложения, 499
специализированные устройства ввода, 492
атрибуты качества, 488, 489, 490
требования, 488, 489, 490
беспроводной доступ, 489
конструирование приложений, 490
пользовательский интерфейс, 489
разнородные устройства, 489
распределенные вычисления, 491
традиционные процедуры, бизнес-процессы и системы, 490
Luther, система, 409

M

McDonald's, компания, 403
Microsoft Access, СУБД, 530, 531, 532, 536
Microsoft Passport, технология, 380
Microsoft, компания, 396, 456, 457
Middleware, компания, 481
Motorola, компания, 404

N

NAT (трансляция сетевых адресов), 392
Netcraft, компания, 398
Netscape Navigator, браузер, 525, 531
Netscape, компания, 396
NNTP (сетевой протокол передачи новостей), 384
Nokia, компания, 404
NRL (научно-исследовательская лаборатория ВМС США), 86

O

ODBC, интерфейс, 534
OMG, рабочая группа, 278
OMG, рабочая группа по объектному менеджменту, 456

P

PDF, формат, 520
Perl, язык программирования, 533
Philips, компания, 408
PICS (платформа отбора информации в Интернете), 381
ping, пакет, 139
PostScript, язык описания страниц, 520

Q

Quack.com, компания
история, 515
проектирование на основе коробочных компонентов, 515

R

Rigi, стандартная форма, 281, 287, 290, 306
rlogin, протокол, 384
RMI (удаленный вызов методов), 152
ROI (коэффициент прибыли на инвестиции), 357, 361, 371

S

SAAM (метод анализа программной архитектуры), 353
SQL Server, СУБД, 536
SQL, язык, 282, 287, 289, 291, 293, 296
SS2000, линейка продуктов, 421, 423, 426, 429, 431, 432, 435, 436, 438, 440, 442, 444, 446, 447, 449, 450, 451, 453
архитектура, 441, 444, 445, 447, 449, 451, 452
группы системных функций, 444, 447
многоуровневое представление, 446
представление декомпозиции на модули, 447, 449
представление процессов, 444, 445, 446
применение, 449, 451, 452, 453
системные функции, 444, 447
библиография, 454
операционная среда и физическая архитектура, 442
организационная структура, 433, 435, 438, 439, 441
перечень систем, 423
требования и атрибуты качества, 441
безопасность, 442
готовность, 442
контролепригодность, 442
модифицируемость, 442
надежность, 442
производительность, 442
характеристика, 423

SSL (протокол защищенных сокетов), 156, 380, 392
Sun Microsystems, компания, 455, 457, 458, 460, 481
Sun, компания, 396

T

TELNET, протокол, 384
TN3270, протокол, 384
Tomcat, сервер приложений, 536, 537

U

UCMEdit, система редактирования карт, 296, 297, 301, 303, 305
UDDI (система предметного описания и интеграции), 396
UML (унифицированный язык моделирования), 265
  библиография, 278
  модульные представления, 266
    агрегация, 267
    зависимость, 269
    интерфейсы, 266
    модули, 267
    обобщение, 269
  нотации, 266
  представления «компонент и соединитель», 269
    интерфейсы, 271, 272
    компоненты, 271
    системы, 274
    соединители, 273
  представления распределения, 275
  стереотипы, 259, 266, 269, 273, 277
Underwriters Laboratories, компания по аттестации, 546
UNIX-фильтры, 520
URL (унифицированный указатель ресурса), 377
Use Case, элементы, 76, 196, 202, 204, 207, 531

V

Visual Basic, язык программирования, 516
Visual Mining, компания, 530
VPN (виртуальная частная сеть), 156

W

W3C, консорциум, 382, 396, 398
WAIS (глобальный информационный сервер), 384
WAP (беспроводной прикладной протокол), 495
Windows 98, операционная система, 531
Windows NT 4.0, операционная система, 530
WML (язык разметки для беспроводных систем), 495

X

X Window System, система графического отображения, 519
XML, язык разметки, 530
  шаблон DTD, 472

A

абстрактно-синтаксические деревья (AST), 284
Алегзандер, Кристофер (Christopher Alexander), 216
ансамбли компонентов, 524, 525, 527
  оценка осуществимости, 524
    модельная задача, 525, 527
    модельное решение, 526, 527
архитектура, 51, 52, 53, 54, 55, 56, 57, 58, 60, 62, 63, 64, 65, 66, 67, 68, 69, 70, 73, 74, 75, 76, 77, 78, 79
  анализ и оценка, 46
  аналогии с другими значениями слова, 58
  архитектурное несоответствие, 67
  архитектурные образцы, 57
  библиография, 77
  выявление требований, 45, 46
  диаграммы, 51, 52
  документирование, 247, 249, 250, 252, 254, 256, 258, 264, 265
  заинтересованные лица, 38
  как выразитель множественности структур системы, 53, 54, 56
  как выразитель начальных проектных решений, 59, 61, 62, 64, 65
  как выразитель поведения, 55
  как переносимая абстракция системы, 59, 65, 66, 67, 68
    взаимозаменяемость, 66
    облегчает анализ и организацию изменений, 64
    облегчает расчет стоимости и временных затрат, 65
    облегчает эволюционное макетирование, 65
    ограничение проектных альтернатив, 67

архитектура (*продолжение*)
определяет ограничения реализации, 62
определяет организационную структуру, 62
позволяет прогнозировать реализацию качества, 64
построение линеек продуктов, 66
разработка на основе шаблонов, 68
регулирует реализацию атрибутов качества, 63
способствует обучению, 68
как средство взаимодействия заинтересованных лиц, 59, 60
как средство определения программных элементов, 53
клиент–сервер, 377, 382, 385
критерии оценки качества, 47, 48, 50
линейки продуктов, 402, 403, 405, 407, 410, 411, 413, 414, 415, 417, 419
макет системы, 48
определение, 35, 53, 54, 55, 56
перекрестное логическое обоснование, 265
планирование, 33
повторное использование, 402, 403, 405, 407, 410, 411, 413, 414, 415, 417, 419
представления, 69, 76
проверка соответствия, 47
программная и системная, 67, 68
проектирование, 70
проектная, 280, 301
распространение сведений, 46
реализация, 47
создание или выбор, 46
создание экономической модели системы, 45
структуры, 53, 56, 69, 70, 71, 73, 74, 75
техническая база, 41
универсальность, 54
фактическая, 280
факторы влияния, 37, 41
заинтересованные лица, 38
компания-разработчик, 40
обратное воздействие, 42, 44
опыт архитектора, 40
техническая база, 41
эталонная архитектура, 58
эталонная модель, 58
архитектура Luther
реализация задач по качеству, 511
архитектура линейки продуктов
SS2000, 441, 442, 445, 447, 449, 451, 452
архитектурная документация
библиография, 278
варианты применения, 247, 249
и заинтересованные лица, 249, 251
как средство обучения, 248, 249
на языке UML, 265, 267, 269
перекрестная документация, 262
глоссарий проекта, 265
каталог представлений, 263
обзор системы, 264
соответствие между представлениями, 264
список элементов, 265
шаблон представления, 264
представления, 250, 252, 265
глоссарий терминов, 256
другая информация, 256
интерфейсы, 257, 259
каталог элементов, 254
компонент и соединитель, 269, 272, 274
контекстная диаграмма, 255
модульные, 266, 267
первичное отображение, 254
поведение, 257
предпосылки архитектурного решения, 255
распределения, 275
руководство по изменчивости, 255
стандартная семичастная структура, 254
архитектурно-экономический цикл, 219
и архитектура Luther, 485, 486, 488
архитектурно-экономический цикл (ABC), 33, 35, 37, 38, 40, 41, 42, 44, 45, 46, 47, 48, 49, 82, 316
веб-архитектуры, 375, 379, 382, 389, 390, 396
и программный процесс, 45, 46
перспективы, 542
архитектурное несоответствие, 517, 518, 519, 521, 522, 524
как частный случай интерфейсного несоответствия, 518
архитектурные мотивы, 196, 198, 199, 206, 209, 215
архитектурные образцы, 57, 197, 200, 201, 203
активный объект, 163
библиография, 166
и архитектурные стили, 164
и тактики, 163, 164
модель–представление–контроллер, 148
представление–абстракция–управление, 148
структурная модель, 219, 226, 229, 232, 243

архитектурные представления
  логическое, 76
  определение, 69
  процесс, 76
  разработка, 76
  физическое, 76
архитектурные стили
  и архитектурные образцы, 164
архитектурные стратегии
  и тактики, 137
архитектурные структуры, 53, 56, 69
  компонент и соединитель, 69, 72
  модульные, 69, 70, 71
  определение, 69
  отношения между ними, 75
  программные, 70
  распределения, 69, 73
атрибутный метод
  проектирования (ADD), 196, 197, 248
  конкретизация модулей, 201
  этапы, 199
    выбор архитектурного образца, 200
    выбор архитектурных мотивов, 199
    выбор модуля для декомпозиции, 199
    задание интерфейсов дочерних модулей, 206
    отображение архитектуры в представлениях, 203
    представление архитектуры в проекциях, 201
    проверка и уточнение, 207
    распределение функциональности, 201
атрибуты качества
  возможность реализации, 523
атрибуты качества, 106, 107, 108, 109, 110, 112, 113, 114, 115, 116, 117, 118, 119, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 221
  архитектуры, 109, 133
    возможность построения, 133, 134
    завершенность, 133
    концептуальная целостность, 133
    правильность, 133
  библиография, 134
  достаточность, 523
  и архитектура, 108
  имена, 327
  интегрируемость, 219, 224, 225, 233, 237, 242, 243
  коммерческие, 109, 131
атрибуты качества (*продолжение*)
  график развертывания, 132
  интеграция с существующими системами, 132
  предполагаемый срок службы системы, 132
  срок выхода продукта на рынок, 131
  стоимость и прибыль, 132
  целевой сегмент рынка, 132
  модифицируемость, 108, 218, 225, 233, 242, 244
  практичность, 108
  производительность, 108, 223, 225, 227, 242, 243
  реализация, 109, 136, 137, 138, 140, 141, 142, 144, 145, 146, 147, 152, 157, 159, 161, 162, 163, 164, 166
  системы, 109
    безопасность, 121, 122, 124, 155, 156
    готовность, 114, 115, 116, 138, 139, 140, 141, 142, 143
    контролепригодность, 125, 126, 158, 159
    масштабируемость, 131
    модифицируемость, 116, 117, 143, 144, 145, 146, 147
    недостатки, 109
    переносимость, 131
    практичность, 127, 128, 129, 130, 160, 161
    производительность, 118, 119, 121, 150, 152, 155
    сценарии атрибутов качества, 110, 112, 113, 114, 115, 116, 117, 118, 119, 121, 122, 124, 125, 126, 127, 128, 129, 130
  тактики, 136, 137, 138, 139, 141, 142, 144, 145, 147, 149, 151, 154, 157, 159, 161, 162, 163, 165, 166
    и архитектурные образцы, 163, 164
    определение, 137
    реализации безопасности, 155, 156
    реализации готовности, 138, 139, 140, 141, 142, 143
    реализации контролепригодности, 158, 159
    реализации модифицируемости, 143, 144, 145, 146, 147
    реализации практичности, 160, 161
    реализации производительности, 150, 152, 155
умеренная специфичность, 460

Б

Бернерс-Ли, Тим (Tim Berners-Lee), 375, 376, 378, 379, 381
Басс, Лен (Len Bass), 547
Басс, Мэтт (Matt Bass), 547
Басс, Таня (Tanya Bass), 547
библиотеки компонентов, 500
  базовые компоненты, 500
  вспомогательные компоненты, 501
  компоненты общих возможностей, 501
  предметно-ориентированные компоненты, 501
Бош, Ян (Jan Bosch), 417
Боэм, Барри (Barry Boehm), 77
брокер объектных запросов (ORB), 456
Брукс, Фредерик (Frederick Brooks), 46, 78, 80, 133, 211, 453, 547
Буш, Ванневар (Vannevar Bush), 376

В

веб-архитектура, 374, 376, 378, 379, 381, 383, 385, 389, 390, 392, 394, 395, 397, 399
  Amazon.com, 397
  HTTPS, 392
  libWWW, библиотека, 377, 378, 382, 385, 388, 398, 399
    вызываемые функции, 384
    интерфейсы прикладного программирования (API), 384
    протоколы, 384
    сменные модули, 383
  антитребования, 379
    безопасность, 379
    защита авторских прав, 379
    защита данных, 379
    трансформация в требования, 379
  архитектурно-экономический цикл, 375, 379, 382, 389, 390, 396
  атрибуты качества, 377, 378, 379, 380
    безопасность, 392
    готовность, 393, 395
    масштабируемость, 393, 394, 395
    модифицируемость, 391, 394
    производительность, 392, 393, 394, 395
  базы данных, 395
  беспрецедентный характер, 379
  библиография, 398
  брандмауэры, 392
    прикладные посредники, 393
    фильтры пакетов, 393
  веб-браузеры, 391
веб-архитектура (*продолжение*)
  веб-серверы, 394
  выравнивание нагрузки, 393
  гипертекст, 376
  задачи по качеству, 389, 395
  клиент–сервер, модель, 382, 385, 392
    CGI, 387
    диспетчер доступа, 386
    диспетчер кэширования, 386
    диспетчер пользовательского интерфейса, 386
    диспетчер представлений, 386
    конфигурационные файлы, 387
    таблица доступа, 387
  контент-провайдеры, 381, 396
  маршрутизаторы, 392
  поставщики услуг, 396
  прокси-серверы, 392
  серверы приложений, 394
  требования, 377, 378, 379, 380, 382
    активные ссылки, 378
    анализ данных, 378
    безопасность, 380, 388, 390
    введение данных пользователями, 378
    высокая готовность, 389
    высокая производительность, 389
    гетерогенность, 378
    децентрализация, 378
    динамика, 380
    доступ к существующим данным, 378
    личные ссылки, 378
    масштабируемость, 390
    модифицируемость, 390
    оформление, 378, 379
    удаленный доступ, 378
  электронная коммерция, 389, 390
    аппаратное обеспечение, 390
    звенья, 390, 394
Вероеф, Крис (Chris Verhoef), 306
виртуальная машина Java (JVM), 457, 463, 480
виртуальные потоки, 204
влияние архитектуры на организацию, 210
влияние организации на архитектуру, 212
вспомогательные деревья, 196
Вудс, Стив (Steve Woods), 515

Г

Гарлан, Дэвид (David Garlan), 67, 77, 164
генераторы, 541

Гласс, Боб (Bob Glass), 100
Группа по технологическому управлению, 501, 505, 513
Гэймер, Джо (Joe Gahimer), 410

Д

декомпозиция, n-прямоугольные схемы, 239
Дийкстра, Эдсгер (Edsger Dijkstra), 78, 80, 178
диск управления, 492

З

заинтересованные лица, 38, 41, 45, 46, 48, 50
Севёрек, Дэн (Dan Siewiorek), 484

И

изменчивость интерфейса, 261
индивидуальность интерфейса, 259
Институт программной инженерии (SEI), 281, 306, 353, 513, 528
инструментарии, 281, 283, 290, 296, 306
интерфейс
  логическое обоснование и соображения о проекте, 261
интерфейсное несоответствие, 518, 519, 521, 522, 524
  архитектурное несоответствие, 518
  методы исправления, 518
    мосты, 520
    оболочки, 519
    посредники, 521
  методы предотвращения, 523
интерфейсы
  допущения, 518, 523
    о предоставляемых услугах, 518, 519, 520, 522
    о требованиях, 518, 519, 520, 522
  определение, 518, 523
  параметризованные, 524
  приватные, 523
  согласованные, 524
информационной закрытости принцип, 79
исключения, 139

К

Карнеги-Меллон, университет, 483
Карье, Джероми (Jeromy Carriere), 515
квалификация, 518
квалификация компонентов, 521
Квилиси, Алекс (Alex Quilici), 515
комплексная система автоматизации (AAS), 167, 169, 177
компонент и соединитель, структура, 204
компонентное проектирование, 514, 515, 517, 518, 520, 521, 523, 524, 527, 529, 531, 534, 537, 538
  как поиск ансамблей, 524, 525, 527
  перспективы, 546
  сборки, 515
компонентные системы
  архитектурное несоответствие, 517
компоненты
  адаптация, 518
  ансамбли, 524, 525, 527
  архитектурное несоответствие, 517, 518, 519, 521, 522, 524
  в сборках, 515
  воздействие на архитектуру, 516, 517
  квалификация, 518
  коробочные, 514, 515, 517, 518, 520, 521, 523, 524, 527, 529, 531, 534, 537, 538
  перспективы, 546
  специфицирование, 518
компоненты и службы технологии J2EE
  обзор, 460
конкретизация модулей, 201
контрольный журнал, 157
коробочные компоненты, 514, 515, 517, 518, 520, 521, 523, 524, 527, 529, 531, 534, 537, 538
  ансамбли, 524, 525, 527
  перспективы, 546
Крюгер, Чарльз (Charles Krueger), 415
Крюхтен, Филипп (Philippe Kruchten), 76

Л

линейки продуктов, 80, 402, 403, 405, 407, 410, 411, 413, 414, 415, 417, 419
  архитектура, 441, 442, 445, 447, 449, 451, 452
  архитектурное проектирование, 404
  библиография, 419, 454
  в компании CelsiusTech, 405, 421, 423, 426, 429, 431, 432, 435, 436, 438, 440, 442, 444, 446, 447, 449, 450, 451, 453
  в компании Cummins, 404, 410
  в компании Hewlett-Packard, 404
  в компании Motorola, 404
  в компании Nokia, 404

линейки продуктов (*продолжение*)
в компании Philips, 408
в Управлении воздушно-космической разведки США, 404
варианты архитектуры, 410, 411, 412, 414
документация, 412
и повторное использование, 406
изменчивость и общность, 410
изменяемые параметры
выявление, 411
обеспечение поддержки, 411
оценка, 413
определение, 403
определение области действия, 407, 408, 409
графическое, 407
инструменты, 408
организационная структура, 417
выделение нескольких отделов разработки, 418
выделение отдела разработки, 417
иерархия отделов инженерии предметных областей, 418
отдел инженерии предметных областей, 418
оценка архитектуры, 413
практические области, 414
преимущества, 403
моделирование и анализ, 405
образцы систем, 406
планирование проекта, 405
процессы, методы и инструменты, 405
специалисты, 406
тестирование, 405
требования, 404
устранение дефектов, 406
элементы, 405
развитие, 416
под влиянием внешних стимулов, 416
под влиянием внутренних стимулов, 417
роль лидера, 415
сложности, 414, 417, 418
ошибки при внедрении, 409
стратегии принятия, 415
активная, 415
реактивная, 416
фонд базовых средств, 403, 407, 410, 412, 416, 417
экономические аспекты, 425, 426, 427

M

маршалинг, 151
масштабирование
вертикальное, 478
горизонтальное, 478
моделирование затрат, 361, 373
модельная задача, 525
процесс решения
определение ограничений реализации, 527, 532
оценка модельного решения, 527, 534
реализация модельного решения, 527, 532
установление исходных критериев оценки, 527, 531
установление конечных критериев оценки, 527, 533
формулирование проектного вопроса, 527, 531
модельное решение, 525
оценка, 527, 534
реализация, 527, 532
мосты, 520

H

неисправность и отказ, 114

O

оболочки, 519
общие абстрактные службы, 144
объектно-ориентированная технология, 220
ограничения на использование ресурсов, 260
определения исключений, 260
определения типов данных, 260
отказ и неисправность, 114
отношения
использования, 71
между структурами, 75
прикрепления, 72
оценка архитектуры, 308
библиография, 315
время проведения, 309
выгоды, 310, 311
запланированная, 312
затраты, 309
методики
ATAM, 315
CBAM, 315

оценка архитектуры (*продолжение*)
активный анализ проектного решения, 315
вопросные, 312
измерительные, 312
незапланированная, 313
результаты, 314
условия проведения, 313
цели, 308

П

парадигма пользовательского опыта, 492
параметризованные интерфейсы, 524
параметры конфигурации элемента, 261
Парнас, Дэвид (David Parnas), 77, 78, 79, 86, 87, 518, 547
перекрестные сценарии, 536
переносные компьютеры
библиография, 512
группа по переносным компьютерам (Wearable Group), 484
история, 484
периодическое управление временем, 227
посредники, 521, 524
предметные области, 211
предоставляемые ресурсы, 259
представление размещения, 205
представления, 250, 252
приватные интерфейсы, 523
программирование, этапы развития, 540, 542
1960-е годы как десятилетие подпрограмм, 540
1970-е годы как десятилетие модулей, 541
1980-е годы как десятилетие объектов, 541
1990-е годы как десятилетие развития абстракций, 541
2000-е годы как десятилетие развития IT-архитектуры, 541
программная архитектура, 51, 52, 53, 54, 55, 56, 57, 58, 60, 62, 63, 64, 65, 66, 67, 68, 69, 70, 73, 74, 75, 76, 77, 78, 79
атрибуты качества, 106, 107, 108, 109, 110, 112, 113, 114, 115, 116, 117, 118, 119, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134
реализация, 136, 137, 138, 140, 141, 142, 144, 145, 146, 147, 152, 157, 159, 161, 162, 163, 164, 166
библиография, 77
и системная архитектура, 67, 68
программная архитектура (*продолжение*)
и функциональность, 107
определение, 35, 53, 54, 55, 56
перспективы, 540, 542, 544, 545, 547, 548
автоматизация документирования, 545
архитектурно-экономический цикл (ABC), 542
атрибуты качества, 543
в образовательной программе, 547
и системы управления конфигурациями, 545
инструменты проектирования, 544
коробочные компоненты, 546
образцы проектирования, 544
роль в жизненном цикле, 545
сценарии, 543
тактики, 543
повторное использование, 402, 403, 405, 407, 410, 411, 413, 414, 415, 417, 419
с участием коробочных компонентов, 514, 515, 518, 520, 521, 523, 524, 527, 529, 531, 534, 537, 538
архитектурное несоответствие, 517, 518, 519, 521, 522, 524
воздействие на архитектуру, 516, 517
пример деятельности компании Quack.com, 515
сборки, 515
тактики, 136, 137, 138, 139, 141, 142, 144, 145, 147, 149, 151, 154, 157, 159, 161, 162, 163, 165, 166
и архитектурные образцы, 163, 164
определение, 137
реализации безопасности, 155, 156
реализации готовности, 138, 139, 140, 141, 142, 143
реализации контролепригодности, 158, 159
реализации модифицируемости, 143, 144, 145, 146, 147
реализации практичности, 160, 161
реализации производительности, 150, 152, 155
программы управления воздушным движением (ATC), 167, 169, 170, 172, 174, 176, 177, 178, 179, 181, 182, 184, 185, 187, 188, 189, 191, 192
районные центры, 168
служба наземного движения, 168
стоимость разработки, 168
требования
безопасность, 167

программы управления воздушным движением (ATC) (*продолжение*)
жесткие требования реального времени, 167
сверхраспределенность, 167
узловые диспетчерские районы, 168
проектирование архитектуры, 196, 197, 198, 201
библиография, 215
создание макета системы, 213
формирование рабочих групп, 210

P

Распределение функциональности, 202
рациональный унифицированный процесс (RUP), 76, 196
резервирование
активное, 140, 141
пассивное, 141
реконструкция архитектуры, 280, 282, 284, 288, 290, 293, 295, 296, 307
извлечение информации, 281, 283, 284, 296
инструменты, 284
инструментарии, 281, 283, 290, 296, 306
объединение представлений, 282, 288, 297, 300, 302, 305
перечень операций, 281
реконструкция, 282, 290, 292, 294, 297, 300, 302, 305
визуализация и взаимодействие, 290
определение и распознавание образцов, 290
создание базы данных, 286, 297
составление базы данных, 281
руководство по использованию, 261

C

сеансовые beans, пассивация, 476
сеансовые beans без состояния
преимущества, 476
семантика ресурса, 259
семейства программ, 80
синтаксис ресурса, 259
системная архитектура
для решения требований электронной коммерции, 390
системы моделирования условий полета, 218, 220, 228, 230, 234
архитектура, 226, 229, 233
архитектурно-экономический цикл, 219
системы моделирования условий полета (*продолжение*)
атрибуты качества, 221, 223, 225
библиография, 244
декомпозиция
на группы, 239, 240
на системы, 240
интегрируемость, 219, 224, 225, 233, 237, 242, 243
макет системы, 237
модели, 222
модифицируемость, 218, 225, 233, 242, 244
недостатки, 224
проектирование, 218
производительность, 223, 225, 227, 242, 243
рабочие состояния, 223, 231, 233, 235
распределение функциональности, 237
свойства, 223
структурная модель, 229
организующая часть, 229, 230
прикладная часть, 229, 232, 233
требования, 221, 223, 225
тренажерная болезнь, 224
управление временем, 227, 228
системы обнаружения вторжений, 157
системы систем, 542
системы управления реляционными БД (RDBMS), 463
смешанное управление временем, 228
событийное управление временем, 228
согласованные интерфейсы, 524
стоимостные модели, 362
сценарии
атрибутов качества, 110
артефакт, 110
безопасность, 121, 122, 124
готовность, 110, 111, 114, 115, 116
источник стимула, 110
количественная мера реакции, 110
конкретные, 110
контролепригодность, 125, 126
модифицируемость, 112, 113, 116, 117
общие, 110, 130
практичность, 127, 128, 129
применение, 113, 114
производительность, 118, 119, 121
реакция, 110
создание, 113
стимул, 110
условия, 110

Т

тактики, 136, 137, 138, 139, 141, 142, 144, 145, 147, 149, 151, 154, 157, 159, 161, 162, 163, 165, 166
  и архитектурные образцы, 163, 164
  и архитектурные стратегии, 137
  определение, 137
  реализации безопасности, 155, 156
    восстановление после атак, 157
    обнаружение атак, 157
    противодействие атакам, 155, 156
  реализации готовности, 138, 139, 140, 141, 142, 143
    восстановление после неисправности, 139, 140, 141, 142
    обнаружение неисправностей, 139
    предотвращение неисправностей, 142, 143
  реализации контролепригодности, 158, 159
    внутренний мониторинг, 159
    контроль входных и выходных данных, 159
  реализации модифицируемости, 143, 144, 145, 146, 147
    локализация изменений, 144, 145
    откладывание связывания, 149
    предотвращение волнового эффекта, 145, 146, 147, 148
  реализации практичности, 160, 161
    тактики периода исполнения, 161
    тактики периода проектирования, 162
  реализации производительности, 150, 152, 155
    арбитраж ресурсов, 153
    потребление ресурсов, 152
    управление ресурсами, 153
тестирующие программы, 159
тестовые программы, 125
требования к элементам, 261
требования по атрибутам качества
  к J2EE, 459
    реализация при помощи дескрипторов размещения, 474
  к веб-приложениям, 458

У

удаленный вызов методов (RMI), 457, 478
Уитни, Эли (Eli Whitney), 66, 403
Управление воздушно-космической разведки США, 404

Ф

Федеральное авиационное агентство США (FAA), 167, 168, 170, 192
Фогэрти, Лоренс (Laurence Fogarty), 218
функциональность
  и архитектура, 107
  и атрибуты качества, 107
  определение, 107
  структуры, 107
функциональные требования, 207

Х

характеристики атрибутов качества интерфейса, 261

Ш

Шоу, Мэри (Mary Shaw), 164

Э

эволюционный жизненный цикл поставки (EDLC), 195, 214, 215
электронная коммерция
  Amazon.com, 397
  требования, 389
  эталонная архитектура, 390
эталонная архитектура, 58
эталонная модель, 58

Лен Басс, Пол Клементс, Рик Кацман

Архитектура программного обеспечения на практике
2-е издание

Перевел с английского Ю. Гороховский

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*В. Шрага*
Научный редактор	*С. Орлов*
Художник	*Л. Адуевская*
Иллюстрации	*Л. Родионова, В. Шендерова, М. Шендерова*
Корректоры	*И. Хохлова, Н. Цибульникова*
Верстка	*Л. Харитонов*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 29.04.05. Формат 70×100/16. Усл. п. л. 46,44. Тираж 3000 экз. Заказ № 6018.
ООО «Питер Принт». 194044, Санкт-Петербург, пр. Б. Сампсониевский, 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Федерального агентства по печати и массовым коммуникациям.
197110, Санкт-Петербург, Чкаловский пр., 15.