

А.Якобсон, Г.Буч, Дж.Рамбо

ДЛЯ ПРОФЕССИОНАЛОВ

УНИФИЦИРОВАННЫЙ ПРОЦЕСС РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ



С Е Р И Я

ДЛЯ ПРОФЕССИОНАЛОВ



*Ivar Jacobson
Grady Booch, James Rumbaugh*

The Unified Software Development Process



Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid • Capetown • Sydney
Tokyo • Singapore • Mexico City

А. Якобсон, Г. Буч, Дж. Рамбо

ДЛЯ ПРОФЕССИОНАЛОВ

**УНИФИЦИРОВАННЫЙ
ПРОЦЕСС РАЗРАБОТКИ
ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ**



*Санкт-Петербург
Москва • Харьков • Минск
2002*

Айвар Якобсон, Грэди Буч, Джеймс Рамбо

Унифицированный процесс разработки программного обеспечения

Перевел с английского В. Горбунков

Главный редактор	<i>Е. Строганова</i>
Заведующий редакцией	<i>И. Корнеев</i>
Руководитель проекта	<i>А. Васильев</i>
Научный редактор	<i>Ф. Новиков</i>
Литературный редактор	<i>Е. Ваулина</i>
Художник	<i>Н. Биржаков</i>
Иллюстрации	<i>М. Жданова</i>
Корректоры	<i>Н. Тюрина, И. Хохлова</i>
Верстка	<i>А. Зайцев</i>

ББК 32.973-018

УДК 681.3.06

Якобсон А., Буч Г., Рамбо Дж.

Я46 Унифицированный процесс разработки программного обеспечения. — СПб.: Питер, 2002. — 496 с.: ил.

ISBN 5-318-00358-3

Книга, написанная признанными специалистами в области разработки программного обеспечения, описывает унифицированный процесс создания сложных программных систем, включающий в себя как использование средств унифицированного языка моделирования UML — стандартного способа визуализации, конструирования, документирования и пересылки артефактов программных систем, — так и все фазы подготовки и управления этим процессом. Эта книга будет полезна аналитикам, разработчикам приложений, программистам, тестерам и менеджерам проектов.

Original English language Edition Copyright © Addison-Wesley, 1999

© Перевод на русский язык, В. Горбунков, 2002

© Издательский дом «Питер», 2002

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственность за возможные ошибки, связанные с использованием книги.

ISBN 5-318-00358-3

ISBN 0-201-57169-2 (англ.)

ООО «Питер Принт». 196105, Санкт-Петербург, Благодатная ул., д. 67.

Лицензия ИД № 05784 от 07.09.01.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.

Подписано в печать 28.03.02. Формат 70×100^{1/16}. Усл. п. л. 39,99. Тираж 3000 экз. Заказ № 246.

Отпечатано с диапозитивов в ФГУП «Печатный двор» им. А. М. Горького

Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.

197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Предисловие	17
Часть 1. Унифицированный процесс разработки программного обеспечения	
Глава 1. Унифицированный процесс: управляемый вариантами использования, архитектуро-ориентированный, итеративный и инкрементный	32
Глава 2. Четыре «П» — персонал, проект, продукт и процесс — в разработке программного обеспечения	44
Глава 3. Процесс, направляемый вариантами использования	62
Глава 4. Архитектуро-центрированный процесс	88
Глава 5. Итеративный и инкрементный процесс	114
Часть 2. Основной рабочий процесс	
Глава 6. Определение требований — от концепции к требованиям	140
Глава 7. Определение требований в виде вариантов использования	158
Глава 8. Анализ	200
Глава 9. Проектирование	242
Глава 10. Реализация	294
Глава 11. Тестирование	320
Часть 3. Итеративная и инкрементная разработка	
Глава 12. Обобщенный рабочий процесс итерации	342
Глава 13. Анализ и планирование требований инициирует проект	368
Глава 14. Фаза проектирования создает базовый уровень архитектуры	388
Глава 15. Фаза построения приводит к появлению базовых функциональных возможностей	410
Глава 16. Внедрение завершается выпуском продукта	424
Глава 17. Заставим Универсальный процесс работать	438
Приложение А. Обзор языка UML	450
Приложение Б. Расширения UML, специфичные для Универсального процесса	462
Приложение В. Основной глоссарий	466
Литература	478
Алфавитный указатель	482

Содержание

Предисловие	17
Что такое процесс разработки программного обеспечения?	18
Зачем написана эта книга	19
Для кого предназначена эта книга	19
Подход, принятый в книге	20
История Унифицированного процесса	20
Подход компании Ericsson	21
Язык спецификации и описания	22
Objectory	23
Подход компании Rational	24
Rational Objectory Process: 1995-1997	24
Унифицированный язык моделирования	25
Rational Unified Process	26
Благодарности	26
За вклад в эту книгу	26
За многие годы	27
Особые благодарности	28
Процесс наступает	29
От издательства	29

Часть 1. Унифицированный процесс разработки программного обеспечения

Глава 1. Унифицированный процесс: управляемый вариантами использования, архитектуро- ориентированный, итеративный и инкрементный	32
Унифицированный процесс в двух словах	33
Унифицированный процесс управляет вариантами использования	34
Унифицированный процесс ориентирован на архитектуру	35
Унифицированный процесс является итеративным и инкрементным	36

Жизненный цикл Унифицированного процесса	38
Продукт	39
Разделение цикла разработки на фазы	40
Интегрированный процесс	43
Глава 2. Четыре «П» — персонал, проект, продукт и процесс — в разработке программного обеспечения	44
Персонал решает все	45
Процессы разработки влияют на персонал	45
Роли будут меняться	46
Размещение «Ресурсов» внутри «Сотрудников»	47
Проект порождает продукт	49
Продукт — это больше, чем код	49
Что такое программная система?	49
Артефакты	50
Система содержит набор моделей	50
Что такое модель?	51
Каждая модель — это самодостаточное представление системы	52
Модель изнутри	52
Связи между моделями	53
Процесс направляет проекты	53
Процесс как шаблон проекта	54
Связанные деятельности образуют рабочие процессы	54
Специализированные процессы	56
Похвала процессу	57
Средства и процесс — одно целое	57
Средства жестко привязаны к процессу	58
Процесс управляет средствами	58
Баланс между процессом и средствами его осуществления	59
Унифицированный язык моделирования поддерживает визуальное моделирование	60
Средства поддерживают весь жизненный цикл системы	61
Глава 3. Процесс, направляемый вариантами использования	62
Введение в разработку, управляемую вариантами использования	64
Зачем нужны варианты использования?	66
Определение требований, приносящих ощутимый и измеримый результат, важный для заказчика	67
Управление процессом	68
Задание архитектуры	69
Определение вариантов использования	70
Модель вариантов использования отражает функциональные требования	70

Актанты — это среда, в которой существует система	71
Варианты использования определяют систему	71
Анализ, проектирование и разработка при реализации варианта использования	72
Создание по вариантам использования аналитической модели	73
Каждый класс должен играть все свои роли в кооперациях	78
Создание модели проектирования из аналитической модели	78
Классы группируются в подсистемы	82
Создание модели реализации из проектной модели	83
Тестирование вариантов использования	85
Резюме	86
 Глава 4. Архитектуро-центрированный процесс	 88
Введение в архитектуру	89
Зачем нужна архитектура?	91
Понимание системы	91
Организация разработки	92
Повторное использование	92
Развитие системы	93
Варианты использования и архитектура	94
Шаги разработки архитектуры	98
Базовый уровень архитектуры	99
Использование образцов архитектуры	101
Описание архитектуры	103
Архитектор, создающий архитектуру	106
Описание архитектуры	107
Архитектурное представление модели вариантов использования	108
Архитектурное представление модели проектирования	108
Архитектурное представление модели развертывания	111
Архитектурное представление модели реализации	112
Три интересных понятия	113
Что такое архитектура?	113
Как ее получить?	113
Как ее описать?	113
 Глава 5. Итеративный и инкрементный процесс	 114
Введение в итеративность и инкрементность	115
Разрабатываем понемногу	116
Чем не является итерация	117
Почему мы используем итеративную и инкрементную разработку?	118
Снижение рисков	118
Получение устойчивой архитектуры	120
Поддержка изменяющихся требований	121
Доступность тактических изменений	121

Достижение постоянной целостности	122
Достижение легкой обучаемости	123
Итеративный подход — управляемый рисками	124
Итерации снижают технические риски	125
За нетехнические риски отвечает руководство	127
Работа с рисками	127
Обобщенная итерация	128
Что такое итерация?	128
Планирование итераций	130
Последовательность итераций	131
Результат итерации — приращение	132
Итерации в жизненном цикле программы	132
Модели в ходе итераций совершенствуются	135
Итерации проверяют организацию	136

Часть 2. Основной рабочий процесс

Глава 6. Определение требований — от концепции к требованиям

Почему трудно определять требования	140
Цели процесса определения требований	141
Обзор процесса определения требований	142
Роль требований в жизненном цикле разработки программного обеспечения	147
Понимание контекста системы с помощью модели предметной области	148
Что такое модель предметной области?	148
Разработка модели предметной области	149
Использование моделей предметной области	150
Понимание контекста системы с помощью бизнес-модели	151
Что такое бизнес-модель?	151
Как разработать бизнес-модель	153
Поиск вариантов использования по бизнес-модели	155
Дополнительные требования	156
Резюме	157

Глава 7. Определение требований в виде вариантов использования

Введение	158
Артефакты	160
Артефакт: Модель вариантов использования	160
Артефакт: Актант	161
Вариант использования	162

Артефакт: Описание архитектуры	166
Артефакт: Глоссарий	167
Артефакт: Прототип интерфейса пользователя.....	167
Сотрудники	167
Сотрудник: Системный аналитик	168
Сотрудник: Спецификатор вариантов использования.....	169
Сотрудник: Разработчик интерфейса пользователя	169
Сотрудник: Архитектор	170
Рабочий процесс	171
Деятельность: Нахождение актантов и вариантов использования	172
Деятельность: Определение приоритетности вариантов использования	181
Деятельность: Детализация вариантов использования	182
Деятельность: Создание прототипа интерфейса пользователя.....	188
Деятельность: Структурирование модели вариантов использования	194
Рабочий процесс определения требований: резюме	199
Глава 8. Анализ	200
Введение в анализ	200
Кратко об анализе	203
Почему анализ — это не проектирование и не реализация	204
Цели анализа: краткий обзор	205
Конкретные примеры случаев, в которых следует использовать анализ	205
Роль анализа в жизненном цикле программы	206
Артефакты	208
Артефакт: Модель анализа	208
Артефакт: Класс анализа	209
Артефакт: Анализ реализации варианта использования	213
Артефакт: Пакет анализа	217
Артефакт: Описание архитектуры (представление модели анализа)	220
Сотрудники	221
Сотрудник: Архитектор	221
Сотрудник: Инженер по вариантам использования	222
Сотрудник: Инженер по компонентам	222
Рабочий процесс	223
Деятельность: Анализ архитектуры	224
Деятельность: Анализ варианта использования	230
Деятельность: Анализ класса	234
Деятельность: Анализ пакетов	238
Рабочий процесс анализа — резюме	239

Глава 9. Проектирование	242
Введение	242
Роль проектирования в жизненном цикле разработки программного обеспечения	243
Артефакты	244
Артефакт: Модель проектирования	244
Артефакт: Класс проектирования	245
Артефакт: Проект реализации варианта использования	248
Артефакт: Подсистема проектирования	251
Артефакт: Интерфейс	253
Артефакт: Описание архитектуры (представление модели проектирования)	254
Артефакт: Модель развертывания	255
Артефакт: Описание архитектуры (представление модели развертывания)	256
Сотрудники	256
Сотрудник: Архитектор	256
Сотрудник: Инженер по вариантам использования	258
Сотрудник: Инженер по компонентам	258
Рабочий процесс	259
Деятельность: Проектирование архитектуры	259
Деятельность: Проектирование вариантов использования	276
Деятельность: Проектирование класса	282
Определение обобщений	286
Деятельность: Проектирование подсистемы	289
Рабочий процесс проектирования — резюме	291
Глава 10. Реализация	294
Введение	294
Роль реализации в жизненном цикле разработки программного обеспечения	295
Артефакты	296
Артефакт: Модель реализации	296
Артефакт: Компонент	296
Артефакт: Подсистема реализации	299
Артефакт: Интерфейс	301
Артефакт: Описание архитектуры (Представление модели реализации)	302
Артефакт: План сборки	302
Сотрудники	303
Сотрудник: Архитектор	303
Сотрудник: Инженер по компонентам	304
Сотрудник: Системный интегратор	305
Рабочий процесс	306

Деятельность: Реализация архитектуры	307
Деятельность: Сборка системы	309
Деятельность: Реализация подсистемы	311
Деятельность: Реализация класса	314
Деятельность: Выполнение тестирования модулей	315
Рабочий процесс реализации: резюме	319
Глава 11. Тестирование	320
Введение	320
Роль тестирования в жизненном цикле программы	321
Артефакты	322
Артефакт: Модель тестирования	322
Артефакт: Тестовый пример	322
Артефакт: Процедура тестирования	325
Артефакт: Тестовый компонент	327
Артефакт: План тестирования	328
Артефакт: Дефект	328
Артефакт: Оценка теста	328
Сотрудники	328
Сотрудник: Разработчик тестов	328
Сотрудник: Инженер по компонентам	328
Сотрудник: Тестер целостности	329
Сотрудник: Системный тестер	329
Рабочий процесс	330
Деятельность: Планирование тестирования	331
Деятельность: Разработка теста	332
Деятельность: Реализация теста	336
Деятельность: Проведение тестирования целостности	336
Деятельность: Проведение тестирования системы	337
Деятельность: Оценка результатов тестирования	337
Тестирование: резюме	339
Часть 3. Итеративная и инкрементная разработка	
Глава 12. Обобщенный рабочий процесс итерации	342
Необходимость баланса	343
Фазы, на которые предварительно разбивается работа	344
Фаза анализа и планирования требований определяет выполнимость	344
Фаза проектирования обеспечивает возможность выполнения	345
Фаза построения создает систему	346
Фаза внедрения переносит систему в среду пользователей	347
Еще раз об обобщенной итерации	347
Основные рабочие процессы повторяются на каждой итерации	348

Сотрудники участвуют в рабочих процессах	348
Планирование предваряет деятельность	350
Планирование четырех фаз цикла	350
Планирование итераций	351
Взгляд в будущее	352
Планирование критериев оценки	353
Риски влияют на планирование проекта	354
Управление списком рисков	354
Влияние рисков на план итераций	355
Выделение рискованных действий	355
Расстановка приоритетов вариантов использования	356
Риски, характерные для отдельных продуктов	357
Риск не создать правильную архитектуру	358
Риск неправильного определения требований	359
Требуемые ресурсы	360
Проекты сильно различаются	360
Как выглядит типичный проект	362
Сложный проект требует большего	362
Новая линия продуктов требует опыта	363
Цена за использование ресурсов	364
Оценка итераций и фаз	365
Критерии не достигнуты	366
Критерии сами по себе	366
Следующая итерация	366
Развитие набора моделей	367

Глава 13. Анализ и планирование требований

инициирует проект	368
Введение	368
В начале фазы анализа и планирования	369
Перед началом фазы анализа и планирования требований	369
Планирование фазы анализа и планирования требований	370
Расширение концепции системы	371
Задание критериев оценки	372
Типичный поток работ итерации на фазе анализа и планирования требований	374
Введение в пять основных потоков работ	374
Погружение проекта в среду разработки	376
Определение критических рисков	376
Выполнение основных рабочих процессов от определения требований до тестирования	376
Определение требований	378
Анализ	380
Проектирование	381

Реализация	382
Тестирование	382
Определение исходных деловых перспектив	383
Формулировка бизнес-предложения	383
Оценка доходности инвестиций	384
Определение итераций в фазе анализа и планирования требований	385
Планирование фазы проектирования	386
Результаты фазы анализа и планирования требований	387
Глава 14. Фаза проектирования создает базовый уровень архитектуры	388
Введение	388
В начале фазы проектирования	389
Планирование фазы проектирования	389
Построение команды	390
Модификация среды разработки	390
Задание критериев оценки	390
Типичный поток работ итерации на фазе проектирования	391
Определение и уточнение большей части требований	392
Разработка базового уровня архитектуры	393
Пока команда малочисленна — ищите путь	393
Выполнение основных рабочих процессов — от определения требований до тестирования	393
Определение требований	395
Анализ	397
Проектирование	401
Реализация	403
Тестирование	405
Определение деловых перспектив	406
Подготовка бизнес-предложения	407
Уточнение доходности инвестиций	407
Оценка результатов итераций и фазы проектирования	407
Планирование фазы построения	408
Основные результаты	409
Глава 15. Фаза построения приводит к появлению базовых функциональных возможностей	410
Введение	410
В начале фазы построения	411
Персонал для осуществления фазы	412
Задание критериев оценки	412
Типичный рабочий процесс итерации в фазе построения	413

Выполнение основных потоков работ — от определения требований до тестирования	415
Определение требований	416
Анализ	417
Проектирование	418
Реализация	419
Тестирование	421
Контроль бизнес-плана	422
Оценка результатов итераций и фазы построения	422
Планирование фазы внедрения	423
Основные результаты	423
Глава 16. Внедрение завершается выпуском продукта	424
Введение	424
В начале фазы внедрения	425
Планирование фазы внедрения	426
Персонал для осуществления фазы	427
Задание критериев оценки	428
Основные потоки работ на этой фазе не играют почти никакой роли	428
Что мы делаем на фазе внедрения	430
Выпуск бета-версии	430
Установка бета-версии	430
Реакция на результаты тестирования	431
Адаптация продукта к различным операционным средам	432
Завершение артефактов	433
Когда заканчивается проект	433
Завершение бизнес-плана	434
Контроль прогресса	434
Пересмотр бизнес-плана	434
Оценка фазы внедрения	435
Оценка итерации и фазы внедрения	435
Результаты экспертизы законченного проекта	436
Планирование следующего выпуска или версии	436
Основные результаты	436
Глава 17. Заставим Универсальный процесс работать	438
Универсальный процесс помогает справиться со сложностью	438
Цели жизненного цикла	439
Архитектура жизненного цикла	439
Базовые функциональные возможности	440
Выпуск продукта	440
Основные темы	440
Руководство управляет переходом на Универсальный процесс	441

Момент истины	442
Приказ о реинжиниринге убеждает в необходимости перехода	442
Осуществление внедрения	444
Специализация Унифицированного процесса	445
Привязка процесса	446
Заполнение каркаса процесса	446
Универсальный процесс для широкого круга лиц	447
Определение пользы от использования Универсального процесса	448
Приложение А. Обзор языка UML	450
Введение	450
Словарь	451
Механизмы расширения	452
Графическая нотация	452
Понятия, относящиеся к структуре	452
Понятия, относящиеся к поведению	453
Понятия, относящиеся к группировке	454
Понятия, относящиеся к примечаниям	454
Отношения зависимости	455
Отношения ассоциации	455
Отношения обобщения	455
Механизмы расширения	455
Глоссарий терминов	456
Приложение Б. Расширения UML, специфичные для Универсального процесса	462
Введение	462
Стереотипы	462
Именованные значения	464
Графическая нотация	465
Приложение В. Основной глоссарий	466
Введение	466
Понятия	466
Литература	478
Алфавитный указатель	482

Предисловие

Существует убеждение, что грамотные предприниматели должны ориентироваться на навыки и умения хорошо обученных одиночек. Профессионалы знают, что надо делать, и сделают это! В своем деле они не нуждаются в политическом и организационном руководстве.

Это мнение во многих случаях является заблуждением, а в случае разработки программ оно является опасным заблуждением. Разумеется, разработчики программ хорошо обучены, но это слишком молодая профессия. В результате разработчикам требуется организационное руководство, которое в этой книге носит название «Процесс разработки программного обеспечения». Кроме того, поскольку процесс, который мы излагаем в этой книге, представляет собой сочетание различных до недавнего времени методологий, мы считаем, что правильно будет называть его «Унифицированным процессом». Он объединяет не только работу трех авторов, в него входят многочисленные труды отдельных людей и компаний, создававших UML, а также значительного числа основных сотрудников Rational Software Corporation. В нем успешно использован опыт сотен организаций, применявших ранние версии процесса на площадках заказчиков.

Дирижер симфонического оркестра, например, всего-то и делает во время концерта, что подает музыкантам знак к началу и задает им единый темп. Ему не нужно больше ничего делать, потому что он руководил оркестром на репетициях и подготовил исполняемую партитуру. Кроме того, каждый музыкант прекрасно знает свой инструмент и играет на самом деле независимо от других членов оркестра. Что более важно для нашей цели, каждый музыкант следует «процессу», давно предписанному ему композитором. Это музыкальный темп, который является основой «политики и процедур», управляющих исполнением музыки. В противоположность музыкантам, разработчики программного обеспечения не могут играть, не глядя на окружающих. Они взаимодействуют друг с другом и с пользователем. У них нет темпа, которому они следуют, — зато у них есть *процесс*.

Необходимость такого процесса ощущается все сильнее, особенно в тех отраслях или организациях, где задачи, выполняемые программными системами, очень важны, — в финансах, управлении воздушным движением, обороне и системах связи. Поэтому мы считаем, что успешное ведение бизнеса или выполнение общественных миссий зависят от того, как работает поддерживающее их программное обеспечение. Программные системы становятся все сложнее и сложнее, время выхода на рынок требуется сокращать, и, соответственно, разработка все более затрудняется. По этим причинам индустрия программного обеспечения нуждается в процессе, который управлял бы разработкой, так же как оркестр нуждается в заданном композитором темпе, которому подчинено исполнение.

Что такое процесс разработки программного обеспечения?

Процесс определяет, *кто, когда и что* делает и *как* достичь определенной цели. В разработке программного обеспечения цель — разработать или улучшить существующий программный продукт. Хороший процесс предоставляет указания по эффективной разработке качественного программного продукта. Он определяет и представляет наилучшие для текущего развития отрасли способы разработки. В результате этот процесс уменьшает риск и повышает предсказуемость. Общий эффект — содействие взаимопониманию и культуре.

Нам нужен такой процесс, который управлял бы действиями всех его участников — заказчиков, пользователей, разработчиков и руководства. Старые процессы не подходят для этой цели, в данный момент нам необходим лучший в промышленности процесс. Наконец, этот процесс должен быть широко распространенным, чтобы все заинтересованные лица могли понять свои роли в обсуждаемой разработке.

Процесс разработки программного обеспечения также должен подразумевать возможность многолетнего развития. В ходе этого развития он будет постоянно адаптироваться к действительному положению дел, которое определяется доступными технологиями, утилитами, персоналом и организационными шаблонами.

- **Технологии.** Процесс должен строиться на основе технологий — языков программирования, операционных систем, компьютеров, пропускной способности сетей, сред разработки и т. п., — существующих во время использования процесса. Например, двадцать лет назад визуальное моделирование не было широко распространено. Оно было слишком дорогим. В настоящее время создатель процесса может только догадываться, как следовало использовать эти нарисованные от руки диаграммы. Согласно этим догадкам, качество моделирования, которое мог обеспечить инициатор процесса, было не слишком высоким.
- **Утилиты.** Процесс и утилиты могут разрабатываться параллельно. Утилиты и процесс неразделимы. Для того чтобы переход на новый путь был оправдан, правильно разработанный процесс должен поддерживать вложения на создание обеспечивающих его утилит.
- **Персонал.** Тот, кто создает процесс, должен быть в состоянии ограничить набор навыков, необходимых для работы с ним. Для работы с процессом должно хватать навыков разработчиков, имеющихся под рукой в настоящее время, или навыков, использованию которых этих разработчиков можно быстро обучить. Во многих областях сейчас можно использовать встроенные техники, например проверку модели на целостность можно производить посредством компьютерных утилит.
- **Организационные шаблоны.** Пока разработчики программ не являются независимыми экспертами, как музыканты симфонического оркестра, они далеки от тех работников-автоматов, на которых Фредерик В. Тейлор (Frederick W. Taylor) строил «научное управление» сто лет назад. Создатели процесса должны адаптировать процесс к реалиям сегодняшнего дня — фактам виртуальной организации, удаленной работе по высокоскоростным линиям, присутствию как частичных вла-

дельцев (в небольших фирмах, начинающих разработки), постоянных работников, так и контрактников и удаленных субконтракторов, и постоянному недостатку разработчиков.

Конструкторы процесса должны уравновесить эти четыре набора условий. Кроме того, баланс должен присутствовать не только в настоящий момент, но и в будущем. Создатель процесса должен проектировать процесс так, чтобы он был в состоянии развиваться, точно так же как разработчик программного обеспечения старается не создавать системы, которые устареют, отработав едва год, он хочет, чтобы его системы, развиваясь и совершенствуясь, существовали много лет. Для того чтобы достичь необходимого уровня стабильности и зрелости, процесс должен разрабатываться несколько лет, только после этого он достигает строгости, необходимой для коммерческой разработки, и рискованность его использования снижается до разумного уровня. Разработка нового продукта — само по себе довольно рискованное дело, даже если не добавлять к ней риск недостаточно проверенного на реальных примерах процесса. Исходя из этих соображений, процесс должен быть стабилен. Без этого баланса между технологиями, утилитами, персоналом и организацией использовать процесс будет чрезвычайно опасно.

Зачем написана эта книга

В этой книге представлен процесс разработки программного обеспечения, который постоянно был у нас в голове, когда мы создавали Унифицированный язык моделирования. Поскольку UML представляет собой стандартный способ визуализации, описания, конструирования, документирования и пересылки артефактов программных систем, мы считаем, что этот язык может быть использован в процессе разработки от его начала и до конца. UML — это средство, а не результат. Правильный результат — это прочная, гибкая и масштабируемая программа. Для ее создания используются как процесс, так и язык. Цель нашей книги — проиллюстрировать части этого процесса. Приведенное в книге приложение, посвященное языку UML, не предполагалось делать исчерпывающим или детализированным. Детальным руководством по UML можно считать *The Unified Modeling Language User Guide* [10]. Исчерпывающим справочником по UML является книга *The Unified Modeling Language Reference Manual* [11].

Для кого предназначена эта книга

Унифицированный процесс разработки программного обеспечения может использоваться любым человеком, вовлеченным в процесс разработки программ. Однако в первую очередь он предназначен для команд, которые осуществляют деятельность жизненного цикла — определение требований, анализ, проектирование, реализацию и тестирование, — то есть работают с моделями UML. Так, например, эта книга будет полезна для аналитиков и конечных пользователей, определяющих требуемую структуру и поведение системы, разработчиков приложений, проектирующих систему, которая удовлетворяет этим требованиям, программистам, пре-

вращающим проект в исполняемый код, тестерам, проверяющим и подтверждающим структуру и поведение системы, разработчикам компонентов, создающим и каталогизирующими компоненты, менеджерам проекта и менеджерам продукта.

Эта книга содержит основу объектно-ориентированных концепций. Опыт в разработке программ и объектно-ориентированном программировании полезен для понимания материала, но не является обязательным.

Подход, принятый в книге

Важнейшее место в этой книге отведено тем видам деятельности — определению требований, анализу и проектированию, — на которые делается основной упор в UML. Именно этим видам деятельности мы обязаны тому, что, выполняя процесс, в состоянии разработать *архитектуру* сложных программных систем. Мы будем, однако, рассматривать весь процесс, хотя и менее детально. Он оканчивается запуском исполняемой программы. Чтобы добиться этого, проекту необходимы усилия каждого члена команды, а также поддержка заинтересованных лиц. Как вы увидите, процесс опирается на огромное разнообразие видов деятельности. Создается и отслеживается множество артефактов. Эти работы не должны протекать бесконтрольно, ими следует управлять.

Полное, подробное описание процесса жизненного цикла находится за пределами возможностей любой отдельной книги. Эта книга должна была бы содержать рекомендации по проектированию, шаблоны артефактов, показатели качества, сведения по управлению проектом, управлению конфигурацией, единицы измерения и многое-многое другое! При разработке это «многое» доступно в онлайновой документации и изменяется в соответствии с требованиями новых проектов. Мы солгаемся на Rational Unified Process, новый программный продукт, предназначенный для работы в Web, который позволяет командам разработчиков программного обеспечения делать свою работу более эффективно. (Дополнительную информацию можно получить на <http://www.rational.com>.) Обеспечивая весь жизненный цикл программного обеспечения, Rational Unified Process дополняет Унифицированный процесс такими видами деятельности, которые в этой книге не затрагиваются или упоминаются вскользь, — бизнес-моделирование, управление проектом и управление конфигурациями.

История Унифицированного процесса

Унифицированный процесс является устоявшимся, потому что это результат трех десятилетий разработки и практического использования. Его разработка как продукта шла по пути, приведенному на рисунке П.1, от Objectory Process (впервые выпущенного в 1987 году) через Rational Objectory Process (выпущен в 1997) к Rational Unified Process (создан в 1998). На эту разработку оказывали воздействие многие факторы. Мы не будем даже пытаться указать их все (все мы просто не знаем) и оставляем это занятие для археологов программирования. Однако мы опишем вклад в этот продукт подходов Ericsson и Rational, а также некоторых других источников.

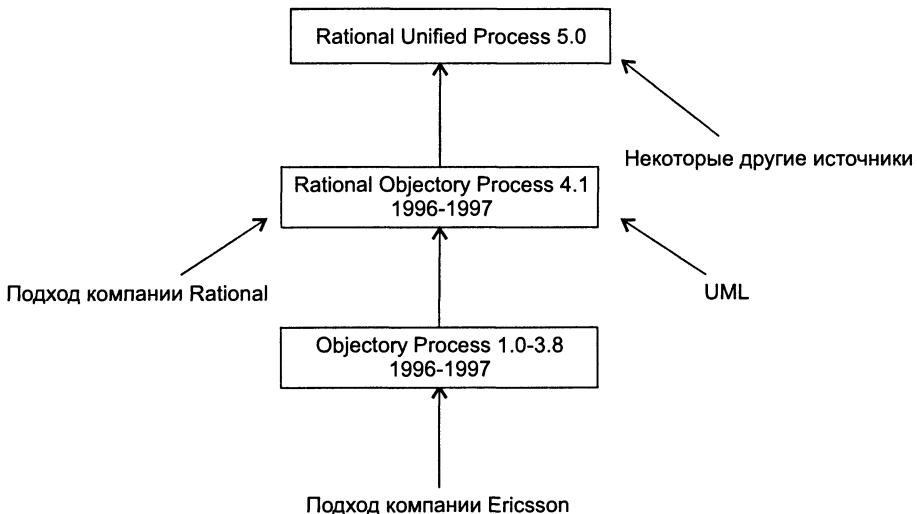


Рис. 1. Разработка Унифицированного процесса
(версии продукта указаны в серых прямоугольниках)

Подход компании Ericsson

Унифицированный процесс имеет глубокие корни. Используя термин, введенный Питером Р. Друкером (Peter R. Drucker), это «инновация, основанная на знаниях». «Между появлением новых знаний и их переплавкой в новую технологию проходит продолжительное время, — учит он. — Затем следует вторая продолжительная пауза, после которой новая технология появляется на рынке в виде продуктов, процессов и услуг» [19].

Одна из причин этого длительного периода затишья состоит в том, что инновации, основанные на знаниях, основаны на соединении множества различных знаний, что требует времени. Другая причина состоит в том, что люди, у которых возникли новые идеи, должны найти время обдумать их и поделиться ими с окружающими.

Чтобы рассмотреть первые шаги по созданию Унифицированного процесса, вернемся назад, в 1967 год, и внимательно присмотримся к достижениям компании Эрикссон [31, 33, 48]. В Эрикссон система моделировалась при помощи набора связанных между собой блоков (в UML это называется «подсистема» и реализуется при помощи «компонентов»). Они собирали блоки нижнего уровня в подсистемы верхнего уровня, чтобы лучше обеспечить управляемость системы. Они выделяли блоки из предварительно определенных вариантов трафика, которые сейчас называются «вариантами использования». Для каждого варианта использования определялись блоки, которые кооперировались, чтобы его реализовать. Зная, за что отвечает каждый блок, можно было создать его описание. Деятельность по проектированию давала в результате комплект статических диаграмм блоков с их интерфейсами и их группировку в подсистемы. Эти диаграм-

мы блоков прямо соответствовали упрощенной версии классов UML или диаграмм пакетов — упрощенной в том смысле, что для связи использовались только явные ассоциации.

Первым рабочим продуктом при проектировании было *описание архитектуры* программы. Оно основано на понимании большинства критических требований. В нем кратко описывается каждый блок и их группировка в подсистему. Набор диаграмм блоков описывает блоки и их связь друг с другом. По связям передаются сообщения особого рода — сигналы. Все сообщения описаны в библиотеке сообщений. Описание архитектуры программы и библиотека сообщений представляют собой основные документы, которые определяют разработку; кроме того, они используются для представления системы заказчикам. В то время (1968 год) заказчики еще не привыкли рассматривать программные продукты, представляемые им в виде, похожем на технические чертежи.

Для каждого варианта использования инженеры создавали диаграмму последовательностей или диаграмму коопераций (и сейчас создаваемые на языке UML). Эти диаграммы показывали, каким образом блоки динамически обмениваются информацией друг с другом с целью реализации варианта использования. Они создавали спецификацию в форме графа состояний (включающего только состояния и переходы) и графа состояний-переходов (упрощенную версию диаграмм деятельности UML). Этот подход, основанный на проектировании из блоков и хорошо описанных интерфейсов между ними, был ключом к успеху. Теперь новая конфигурация системы могла быть создана — например для нового заказчика — простой заменой одного блока на другой, имеющий аналогичный интерфейс.

Итак, блоки не были теперь просто подсистемами и компонентами исходного кода, они компилировались в исполняемые модули, без изменений устанавливались на нужную машину с последующей проверкой, выполняют ли они все, что от них требуется. Кроме того, появлялась возможность установки любого исполняемого блока в систему во время выполнения звонков для постоянно работающих телефонных систем. Для внесения изменений в систему не требовалось ее выключать. Это похоже на смену покрышек автомашины во время ее движения.

В сущности, этот подход использовал то, что мы называем сегодня *разработкой, основанной на компонентах*. Создателем этого метода был Ивар Джекобсон (Ivar Jacobson). Он руководил этим развитием процесса разработки программного обеспечения в течение многих лет — до периода Objectory.

Язык спецификации и описания

В тот период важнейшим делом стало издание в 1976 году CCITT, международным органом по стандартизации в области телекоммуникаций, Языка спецификации и описания (SDL) функционального поведения телекоммуникационных систем. Этот стандарт, оказавший важное влияние на подход фирмы Эрикссон, определяет систему в виде набора связанных блоков, которые общаются друг с другом исключительно путем обмена сообщениями (названных в стандарте «сигналами»). Каждый блок владеет набором «процессов» (это термин SDL для активных классов). Процесс имеет множество экземпляров, подобно классу в терминологии объектно-ориентированного подхода. Экземпляры процесса обмениваются

сообщениями. Рекомендованы диаграммы, специализации которых в UML называются диаграммами классов, диаграммами деятельности, диаграммами кооперации и диаграммами последовательностей.

Итак, SDL был специализированным стандартом объектного моделирования. Периодически обновляемый, он использовался более чем 10 000 разработчиками и поддерживался некоторыми поставщиками утилит. Изначально разработанный более 20 лет назад, он далеко опередил свое время. Однако он был разработан в те времена, когда объектное моделирование было еще не развито. SDL был вытеснен Унифицированным языком моделирования, Unified Modeling Language, который прошел стандартизацию в ноябре 1997 года.

Objectory

В 1987 году Айвар Якобсон (Ivar Jacobson) покинул Эрикссон и основал в Стокгольме компанию Objectory AB. В течение следующих восьми лет он и его сотрудники разрабатывали процесс, названный Objectory («Objectory» — это сокращение от «Object Factory», «Фабрика объектов»). Они распространяли его на другие отрасли промышленности, помимо телекоммуникаций, и на другие страны, помимо Швеции.

Хотя концепция *варианта использования* присутствовала еще в разработках Эрикссон, теперь у нее было имя (введенное на конференции OOPSLA в 1987 году) и разработанная техника изображения при помощи диаграмм. Идея была распространена на множество приложений. Стало яснее видно, что варианты использования управляют разработкой. На передний план вышло понимание того, что архитектура направляет разработчиков и информирует заинтересованных лиц.

Последовательные рабочие процессы были представлены в виде ряда моделей: требований, вариантов использования, анализа, проектирования, реализации и тестирования. Модель — это проекция системы. Отношения между моделями в этом ряду важны для разработчиков как путь переноса свойств от одного конца ряда моделей до другого. Фактически трассируемость стала предпосылкой к разработке, управляемой вариантами использования. Разработчики могли трассировать варианты использования по последовательности моделей до текста программ или, если возникали проблемы, обратно.

Разработка процесса Objectory происходила в виде серии выпусков, от Objectory 1.0 в 1988 году до первой онлайновой версии, Objectory 3.8, в 1995 году (обзор Objectory можно найти в [34]).

Важно отметить, что сам по себе продукт Objectory мог рассматриваться как система. Такой способ описания процесса — как системы-продукта — давал хороший способ разработки новой версии Objectory на основе предыдущей. Этот способ разработки Objectory делал ее удобнее для привязки к разработке различных продуктов и выполнения частных требований конкретной разработки. Сам процесс разработки программного продукта Objectory был сконструирован на основе этого процесса, что составляет его уникальную особенность.

Опыт разработки Objectory также привел к прояснению того, как работает инженер и как в основном работает бизнес. Эти же принципы были приложены к книге, их описывающей и выпущенной в 1995 году [39].

Подход компании Rational

Rational Software Corporation приобрела Objectory AB во время спада 1995 года, и задача унификации базовых принципов, лежащих в основе существующих способов разработки программного обеспечения, опять приобрела актуальность. Rational создала несколько способов разработки программного обеспечения, многие из которых были совместимы с включенной в их число Objectory.

Так, например, «в 1981 году Rational начала производить интерактивную среду, которая повышала продуктивность разработки больших программных систем», писали Джеймс Е. Арчер мл. (James E. Archer, Jr.) и Майкл Т. Девлин (Michael T. Devlin) в 1986 году [3]. «В этих разработках очень важны были объектно-ориентированное проектирование, абстракции, скрытие информации, многократное использование и создание прототипов», — продолжают они.

Разработки Rational с 1981 года детально описаны в книгах, докладах и внутренних документах, но, вероятно, двумя наиболее важными добавками к *процессу* были упор на архитектуру и итеративную разработку. Так, в 1990 году Майкл Девлин (Mike Devlin) написал концептуальную статью о направляемом архитектурой итеративном процессе разработки. Филипп Крухтен (Philippe Kruchten), заведовавший в Rational Архитектурной практикой, стал автором статей по итерациям и архитектуре.

Мы цитировали одну из них, статью об изображении архитектуры в виде четырех представлений: логическом представлении, представлении процесса, физическом представлении и представлении разработки, и дополнительном представлении, иллюстрирующем первые четыре представления при помощи вариантов использования или сценариев [52]. Идея о создании набора представлений вместо попыток впихнуть их все в один тип диаграмм была подсказана ему опытом работы в больших проектах. Множественные представления, дающие возможность взглянуть на систему под определенным углом, помогают заинтересованным лицам и разработчикам понять, какую цель они преследуют.

Кое-кто воспринимает итеративную разработку как нечто анархическое или беспорядочное. Подход четырех фаз (анализ и планирование требований, проектирование, построение и внедрение) приводит к лучшей структуре и контролю прогресса в ходе итераций. Фазы привносят в итерации порядок. Детальное планирование фаз и порядка итераций внутри них было совместной разработкой Уолкера Ройса (Walker Royce) и Рича Рейтмана (Rich Reitman), при постоянном участии Гради Буча (Grady Booch) и Филиппа Крухтена (Philippe Kruchten).

Буч участвовал во всем этом с самого начала существования Rational, а в 1996 году он сформулировал два «основных принципа», касающихся архитектуры и итераций:

- «Управляемая архитектурой разработка обычно является наилучшим способом создания очень сложных программных проектов».
- «Чтобы быть успешным, объектно-ориентированный проект должен применять инкрементный и итеративный процесс» [9].

Rational Objectory Process: 1995-1997

К моменту слияния Objectory 3.8 показал, как можно, подобно продукту, разработать и промоделировать процесс разработки программного обеспечения. Была раз-

работана оригинальная архитектура процесса разработки программ. Был определен набор моделей, в которые записывается результат процесса. Были детально проработаны такие части процесса, как моделирование вариантов использования, анализ и проектирование. Однако другие части — управление требованиями, не входящими в варианты использования, реализация и тестирование — не имели столь детальной проработки. Кроме того, в процессе содержались только наметки информации об управлении проектом, управлении конфигурациями, развертывании и создании среды разработки (обеспечении утилитами и процессами).

Теперь к Objectory были добавлены опыт и практика Rational. Результатом стал Rational Objectory Process 4.1. В частности, добавились фазы в сочетании с управляемым итеративным подходом. Архитектура была выделена явно, в виде описания архитектуры — «cateхизиса» организаций, занимающихся разработкой программ. Было разработано точное определение архитектуры. Архитектура стала восприниматься как существенная часть организации системы. Архитектура изображалась в виде набора архитектурных представлений моделей. Итеративная разработка из относительно общей концепции превратилась в управляемый рисками подход, при котором архитектура разрабатывается в первую очередь.

В это время UML находился в разработке и использовался в качестве языка моделирования для Rational Objectory Process (ROP). Первыми разработчиками UML были авторы этой книги. Команда, разрабатывавшая процесс, во главе с Филиппом Крухтеном (Philippe Kruchten), устранила некоторые из слабых мест ROP путем усиления управления проектом, например, на основе работ Ройса (Royce) [59].

Унифицированный язык моделирования

Необходимость в унифицированном и согласованном визуальном языке для точного выражения результатов довольно многочисленных объектно-ориентированных методологий, появившихся с начала 90-х годов XX века, уже некоторое время была очевидной.

В течение этого времени Гради Буч (Grady Booch), например, стал автором «метода Буч» [8]. Джеймс Рамбо (James Rumbaugh) был основным разработчиком Центра исследований и разработок Джениерал Электрик по ОМТ (Технологии объектного моделирования) [60]. Когда в октябре 1994 года они пришли в Rational, то при участии многих заказчиков Rational начали работу по унификации своих методов. Они выпустили версию 0.8 Унифицированного метода в октябре 1995 года, примерно в то же время, когда Ивар Джекобсон (Ivar Jacobson) пришел в Rational.

Эти три человека, работая совместно, выпустили версию 0.9 Унифицированного языка моделирования. При создании этой версии основные усилия были направлены на ассимиляцию работ других методологов и компаний, включая IBM, HP и Microsoft, каждая из которых способствовала созданию стандарта. В ноябре 1997 года, после прохождения процедуры стандартизации, Унифицированный язык моделирования в версии 1.1 был провозглашен стандартом Группы управления объектами. Как все это было, можно узнать в *User Guide [10] и Reference Manual [11]*.

UML использовался во всех моделях Objectory Process.

Rational Unified Process

В ходе этих работ Rational присоединяла другие компании, производящие утилиты для разработки программ, или сливалась с ними. Каждая из них добавляла свой опыт в области процессов в копилку, расширяя Rational Objectory Process:

- Requisite Inc. вложила свой опыт в управлении требованиями.
- SQA Inc. разработала процесс тестирования для работ по тестированию продуктов, который был добавлен к немалому опыту Rational в этой области.
- Pure-Atria добавила к опыту Rational в управлении конфигурациями свой.
- Performance Awareness добавила тестирование производительности и загрузки.
- Vigortech привнесла экспертизу структуры данных.

К процессу также были добавлены основанные на [39] новые рабочие процессы для бизнес-моделирования, используемые для порождения требований к бизнес-процессам, которые должны обслуживать программное обеспечение. Также в него вошло проектирование пользовательских интерфейсов на основе вариантов использования (базировавшееся на работах Objectory AB).

К середине 1998 года Rational Objectory Process превратился в полномасштабный процесс, способный поддерживать разработку программ в течение всего жизненного цикла. В ходе этого превращения он вобрал в себя большое количество технологий как привнесенных тремя авторами этой книги, так и почерпнутых Rational и UML из других источников. В июне Rational выпустила новую версию продукта, Rational Unified Process 5.0 [53]. Многие элементы этого внутрифирменного процесса были описаны в данной книге, и после ее выхода информация о нем стала общедоступной.

Изменение названия отражает тот факт, что произошла унификация процесса по многим параметрам: унификация подходов к разработке, использование Унифицированного языка моделирования, унификация работ многих методологов — не только в Rational, но и на сотнях площадок заказчиков, которые много лет использовали этот процесс.

Благодарности

Проект такого масштаба — это труд множества людей, и мы рады поблагодарить всех, кого сможем, поименно.

За вклад в эту книгу

Биргитта Ленвиг (Birgitte Lonvig) подготовила пример системы Интербанк и проработала все модели к нему. Это основной пример, проходящий через всю книгу.

Патрик Джонсон (Patrik Jonsson) извлекал материал из документации к Rational Objectory Process и распределял его в соответствии с представленными главами. Он также помогал в подготовке примеров. В ходе работы он предложил множество идей по лучшему представлению Унифицированного процесса.

Вейр Майерс (Ware Myers) участвовал в работе над этой книгой начиная с первых набросков. Он получил от главного автора первый вариант и превратил его во вполне читаемую английскую прозу.

Из рецензентов мы особенно благодарны Курту Биттнеру (Kurt Bittner), Крису Кобрину (Cris Kobryn) и Эрлу Эклунду мл. (Earl Ecklund, Jr.). Кроме того, мы высоко ценим отзывы Уолкера Ройса (Walker Royce), Филиппа Крухтена (Philippe Kruchten), Дена Лефингвелла (Dean Leffingwell), Мартина Гресса (Martin Griss), Марии Эриксон (Maria Ericsson) и Брюса Катца (Bruce Katz). В число рецензентов также вошли Пит Макбрин (Pete McBreen), Гленн Джонс (Glenn Jones), Иоган Джалле (Johan Galle), Н. Вену Гопал (N. Venu Gopal), Дэвид Райн (David Rine), Мэри Лумис (Mary Loomis), Мари Ленци (Marie Lenzi), Джанет Гарднер (Janet Gardner) и несколько человек, которые не пожелали назвать свое имя. Мы благодарны им всем.

Терри Куатрани (Terry Quatrani) из Rational улучшил английский язык в главах 1–5. Карен Тонгиш (Karen Tongish) прочла и отредактировала всю книгу. Мы благодарны им обоим.

Отдельно хотим поблагодарить Стефана Биланда (Stefan Bylund), который усиленно рецензировал первые варианты книги и предлагал улучшения, множество из которых мы включили в книгу.

За многие годы

Мы также благодарны многим людям, которые годами помогали нам «сделать процесс правильным» и всячески поддерживали нашу работу. В особенности мы благодарны следующим людям: Стефану Альквисту (Stefan Ahlquist), Али Али (Ali Ali), Гунилле Андерсон (Gunilla Andersson), Келл С. Андерсону (Kjell S. Andersson), Стен-Эрику Бергнеру (Sten-Erik Bergner), Дейву Бернстайну (Dave Bernstein), Курту Биттнеру (Kurt Bittner), Перу Быйорку (Per Bjork), Хансу Брандтбергу (Hans Brandtberg), Марку Бромсу (Mark Broms), Стефану Биланду (Stefan Bylund), Энн Карлбранд (Ann Carlbrand), Ингемару Карлссону (Ingemar Carlsson), Маргарет Чен (Margaret Chan), Магнусу Кристерсону (Magnus Christerson), Джиффи Клемму (Geoff Clemm), Кетрин Коннор (Catherine Connor), Хакану Далу (Hakan Dahl), Стефану Десджардинсу (Stephane Desjardins), Майку Девлину (Mike Devlin), Хакану Дираджу (Hakan Dyrhage), Сюзанне Дирадж (Susanne Dyrhage), Стефену Энебому (Staffan Ehnebom), Кристиану Энгеборгу (Christian Ehrenborg), Марии Эриксон (Maria Ericsson), Гуннару М. Эрикссону (Gunnar M. Eriksson), Ийену Гавину (Iain Gavin), Карло Готи (Carlo Goti), Сэмю Гукенхаймеру (Sam Guckenheimer), Бьорну Галлбранду (Bjorn Gullbrand), Санни Гупта (Sunny Gupta), Мартену Густафсону (Marten Gustafsson), Бьорну Густафсону (Bjorn Gustafsson), Ларсу Халлмаркену (Lars Hallmarken), Девиду Ханслипу (David Hanslip), Перу Хедфорсу (Per Hedfors), Барбаре Хедлунд (Barbara Hedlund), Йоргену Хеллбергу (Jorgen Hellberg), Иоахиму Герцогу (Joachim Herzog), Келли Хьюстон (Kelli Houston), Агнете Джекобсон (Agneta Jacobson), Стену Джекобсону (Sten Jacobson), Перу Дженсону (Paer Jansson), Хакану Дженсону (Hakan Jansson), Кристеру Йохансону (Christer Johansson), Ингемару Джонсону (Ingemar Johnsson), Патрику Джонсону (Patrik Jonsson), Дэну Джонсону (Dan Jonsson), Брюсу Катцу (Bruce Katz), Курту Катцеву (Kurt Katzeff), Кевину Келли (Kevin Kelly), Энтони Кестертону (Anthony

ny Kesterton), Перу Килгрену (Per Kilgren), Руди Костеру (Rudi Koster), Перу Кроллу (Per Kroll), Рону Крубеку (Ron Krubbeck), Микаэлю Ларсону (Mikael Larsson), Баду Лоусону (Bud Lawson), Дену Леффингвелю (Dean Leffingwell), Рольфу Лейдхаммару (Rolf Leidhammar), Хакану Лидстрому (Hakan Lidstrom), Ларсу Линдрусу (Lars Lindroos), Фредрику Линдстрому (Fredrik Lindstrom), Крису Литтлジョンсу (Chris Littlejohns), Эндрю Лионсу (Andrew Lyons), Джес Мадхур (Jas Madhur), Брюсу Маласки (Bruce Malasky), Крису Мак-Кленахану (Chris McCle-naghan), Кристиану Меку (Christian Meck), Сью Микель (Sue Mickel), Джорме Мобрин (Jorma Mobrin), Кристеру Нильсону (Christer Nilsson), Руну Нильсон (Rune Nilsson), Андерсу Нордину (Anders Nordin), Жан-Эрику Нордину (Jan-Erik Nordin), Роджеру Обергу (Roger Oberg), Бенни Одентегу (Benny Odenteg), Эрику Орнальфу (Erik Ornulf), Гуннару Овергаарду (Gunnar Overgaard), Карин Палмквист (Karin Palmkvist), Фабио Перузци (Fabio Peruzzi), Дженнни Петтерсон (Janne Pettersson), Гари Поллису (Gary Pollice), Тоне Принс (Tonya Prince), Лесли Пробаско (Leslee Probasco), Терри Куатрани (Terry Quatrani), Андерсу Роксторму (Anders Rock-strom), Уолкеру Ройсу (Walker Royce), Горану Шефте (Goran Schefte), Джейфу Шустеру (Jeff Schuster), Джону Смиту (John Smith), Джону Смиту (John Smith), Келл Сорме (Kjell Sorme), Иану Спенсе (Ian Spence), Биргитте Спиридон (Birgitta Spiridon), Фредрику Стромбергу (Fredrik Stromberg), Горану Санделоф (Goran Sundelof), Перу Сандквисту (Per Sundquist), Репу-Олафу Тусселиусу (Per-Olof Thysselius), Майку Тадболлу (Mike Tudball), Карин Виллерс (Karin Villers), Ктираду Врана (Ctirad Vrana), Стефану Уоллину (Stefan Wallin), Роланду Вестеру (Roland Wester), Ларсу Веттерборгу (Lars Wetterborg), Брайану Уайту (Brian White), Ларсу Викторину (Lars Wiktorin), Шарлотте Врейн (Charlotte Wranne) и Яну Вунше (Jan Wunsche).

Кроме того, следующие люди в течение многих лет оказывали главному автору этой книги персональную поддержку, за что он им крайне благодарен: Дайнс Бьорнер (Dines Bjorner), Тоур Бингефорс (Tore Bingefors), Дейв Бульман (Dave Bulman), Ларри Константин (Larry Constantine), Горан Хемдал (Goran Hemdal), Бо Хедфорс (Bo Hedfors), Том Лав (Tom Love), Нильс Леннмаркер (Nils Lennmarker), Ларс-Олаф Норен (Lars-Olof Noren), Дейв Томас (Dave Thomas) и Тарс-Эрик Торелли (Lars-Erik Thorelli).

Особые благодарности

Мы хотим особо поблагодарить Майка Девлина (Mike Devlin), президента Rational Software Corporation, за его веру в процесс Objectory как продукт, который поможет разрабатывать программы по всему свету, за его постоянную поддержку использования эффективных процессов разработки как двигателя в создании программного обеспечения.

И наконец, хотелось бы поблагодарить Филиппа Крухтена (Philippe Kruchten), руководителя Rational Unified Process, и всех членов команды Rational Process за объединение лучших идей Objectory с лучшей практикой Rational и UML с сохранением их ценности. Вдобавок, мы не достигли бы этой цели без личной обязательности и упорства Филиппа в решении задачи построения «всего лишь» лучшего в мире процесса разработки программ.

Процесс наступает

Эта книга и другие книги по этой теме, онлайновые версии и утилиты начинают эпоху *процесса* разработки программ. Унифицированный процесс черпал идеи из нескольких источников, и они всегда активно использовались. Он обеспечивает единый метод понимания процесса, которым могут пользоваться руководители, разработчики и инвесторы.

Все же еще должна быть сделана масса дел. Разработчики должны выучить унифицированные способы осуществления своей работы. Инвесторы и руководство должны поддержать их в этом. Для многих фирм-разработчиков программ прорыв — пока только мечта. Вы можете сделать его реальностью.

*Ivar Jacobson
Palo Alto, California
Декабрь 1998
ivar@rational.com*

От издательства

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

Мы будем рады узнать ваше мнение!

На web-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

ЧАСТЬ 1

Унифицированный процесс разработки программного обеспечения

В первой части мы рассматриваем основные идеи и понятия Унифицированного процесса.

Глава 1 в двух словах описывает Унифицированный процесс разработки программного обеспечения. Особое внимание обращается на то, что он управляется вариантами использования, является архитектурно-ориентированным, итеративным и инкрементным. Процесс использует Унифицированный язык моделирования, который позволяет создавать подобие чертежей, давно уже использующихся в других областях техники, для создания программ. Процесс делает удобным базирование многих разрабатываемых проектов на повторно используемых компонентах — частях программ с хорошо определенными интерфейсами.

Глава 2 вводит понятия четырех «П» — персонала, проекта, продукта и процесса — и описывает их взаимосвязи, важные для понимания последующего материала, излагаемого в книге. Также в этой главе описываются ключевые для понимания процесса понятия, такие, как *артефакт, модель, сотрудник и рабочий процесс*.

В главе 3 детально обсуждается концепция разработки, управляемой вариантами использования. Варианты использования — это средство для правильного выделения требований и применения их для управления процессом разработки.

Глава 4 описывает роль архитектуры в Унифицированном процессе. Архитектура определяет, что следует делать, она дает разбивку программы на основные уровни организации и создает каркас системы.

В главе 5 подчеркивается важность использования итеративного и инкрементного подхода к разработке программного обеспечения. На практике это означает

ускоренную разработку наиболее рискованных частей системы, раннее создание стабильной архитектуры, а затем заполнение ее в ходе последующих итераций множеством менее оригинальных частей. Каждая из этих итераций приводит к продвижению по дороге к финальному выпуску программного обеспечения.

В части 2 мы копнем глубже. Каждому из основных рабочих процессов — определению требований, анализу, проектированию, разработке и тестированию — мы посвятим по главе. Позже, в третьей части, мы используем эти рабочие процессы как независимые деятельности, протекающие в ходе различных итераций на протяжении четырех фаз, на которые мы делим цикл разработки.

В части 3 мы опишем конкретный ход работ на протяжении каждой фазы: фазы анализа и планирования требований — для определения деловых перспектив, фазы проектирования — для создания архитектуры и разработки плана, фазы построения — для превращения архитектуры в готовую к поставке систему и фазы внедрения — для обеспечения корректной работы системы в операционной среде. В этой части мы вновь вернемся к основным рабочим процессам и сгруппируем их применительно к каждой из фаз таким образом, чтобы иметь возможность получить желаемый результат.

Основные цели корпораций состоят, однако, не столько во владении хорошими программами, сколько в таком управлении своими бизнес-процессами или встроенными системами, чтобы у них была возможность, реагируя на запросы рынка, быстро производить товары и услуги высокого качества по умеренным ценам. Программное обеспечение — это стратегическое оружие, используя которое фирмы и правительства могут достичь гигантского снижения стоимости и времени создания товаров и услуг. Быстрое реагирование на изменения рынка невозможно без хороших организационных процессов. В глобальной экономике, которая функционирует двадцать четыре часа в сутки и семь дней в неделю, многие из этих процессов невозможно осуществлять без соответствующего программного обеспечения. Хороший процесс разработки программ, таким образом, становится важнейшим элементом успеха корпорации.

1

Унифицированный процесс: управляемый вариантами использования, архитектуро- ориентированный, итеративный и инкрементный

Сегодня развитие программного обеспечения происходит в сторону увеличения и усложнения систем. Это связано отчасти с тем, что компьютеры с каждым годом становятся все мощнее, что побуждает пользователей ожидать от них все большего. Эта тенденция также связана с возрастающим использованием Интернета для обмена всеми видами информации — от простого текста до форматированного текста, изображений, диаграмм и мультимедиа. Наши аппетиты в отношении более продвинутого программного обеспечения растут по мере того, как мы начинаем понимать с выходом каждого следующего выпуска, как еще можно улучшить этот продукт. Мы желаем иметь программное обеспечение, еще лучше приспособленное для наших нужд, а это, в свою очередь, приводит к усложнению программ. Короче говоря, мы желаем большего.

Мы также желаем получить это программное обеспечение быстрее. Время выхода на рынок — это другой важный стимул.

Сделать это, однако, нелегко. Наше желание получить мощные и сложные программы не сочетается с тем, как эти программы разрабатываются. Сегодня большинство людей разрабатывает программы, используя те же методы, что и 25 лет назад, что является серьезной проблемой. Если мы не улучшим наши методы, мы

не сможем выполнить свою задачу по разработке так необходимого сегодня сложного программного обеспечения.

Проблема программного обеспечения сводится к затруднениям разработчиков, вынужденных преодолевать в ходе разработки больших программ множество препятствий. Общество разработчиков программного обеспечения нуждается в управляемом методе работы. Ему нужен процесс, который объединил бы множество аспектов разработки программ. Ему нужен общий подход, который:

- обеспечивал бы руководство деятельностью команды;
- управлял бы задачами отдельного разработчика и команды в целом;
- указывал бы, какие артефакты следует разработать;
- предоставлял бы критерии для отслеживания и измерения продуктов и функционирования проекта.

Наличие хорошо определенного и хорошо управляемого процесса — в этом основное отличие сверхпродуктивных проектов от неудавшихся. (См. подраздел «Похвала процессу» главы 2, где описаны другие причины необходимости процесса.) Унифицированный процесс разработки программного обеспечения — результат более чем тридцатилетней работы — это решение проблемы программного обеспечения. Эта глава содержит обзор всего Унифицированного процесса. В последующих главах рассматриваются отдельные его элементы.

Унифицированный процесс в двух словах

Прежде всего Унифицированный процесс есть процесс разработки программного обеспечения. Процесс разработки программного обеспечения — это сумма различных видов деятельности, необходимых для преобразования требований пользователей в программную систему (рис. 1.1). Однако Унифицированный процесс — это больше чем единичный процесс, это обобщенный каркас процесса, который может быть специализирован для широкого круга программных систем, различных областей применения, уровней компетенции и размеров проекта.

Унифицированный процесс *компонентно-ориентирован*. Это означает, что создаваемая программная система строится на основе программных компонентов, связанных хорошо определенными интерфейсами (приложение А).

Для разработки чертежей программной системы Унифицированный процесс использует *Унифицированный язык моделирования*. Фактически Унифицированный язык моделирования является неотъемлемой частью Унифицированного процесса — они и разрабатывались совместно.

Однако действительно специфичные аспекты Унифицированного процесса заключаются в трех словосочетаниях — управляемый вариантами использования, архитектурно-ориентированный, итеративный и инкрементный. Это то, что делает Унифицированный процесс уникальным.

В следующих трех пунктах мы опишем эти ключевые слова. Далее в этой главе мы дадим краткий обзор процесса, его жизненного цикла, фаз, выпусков, итераций, рабочих процессов и артефактов. Основная цель этой главы состоит в том,

чтобы ввести важнейшие понятия и дать обзор процесса с высоты птичьего полета. После прочтения этой главы вы узнаете все об Унифицированном процессе, но не обязательно до конца поймете эти сведения. Остальная часть книги посвящена подробностям.

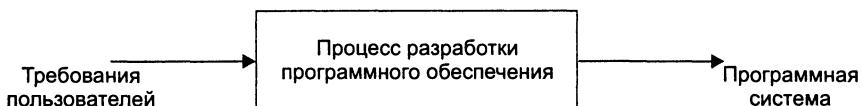


Рис. 1.1. Процесс разработки программного обеспечения

В главе 2 мы добавим к контекstu четыре «П» в разработке программ — персонал, проект, продукт и процесс. Затем мы посвятим по главе каждому из трех словосочетаний. Все это мы проделаем в первой части книги. В частях 2 и 3, которые образуют ядро книги, будут детально описаны различные рабочие процессы, входящие в процесс разработки.

Унифицированный процесс управляетяся вариантами использования

Программная система создается для обслуживания ее пользователей. Следовательно, для построения успешной системы мы должны знать, в чем нуждаются и чего хотят ее будущие пользователи.

Понятие *пользователь* относится не только к людям, работающим с системой, но и к другим системам. Таким образом, понятие *пользователь* относится к кому-либо или чему-либо (например к другой системе, внешней по отношению к рассматриваемой системе), что взаимодействует с системой, которую мы разрабатываем. Пример взаимодействия — человек, использующий банкомат (ATM). Он вставляет в прорезь пластиковую карту, отвечает на вопрос, высвечиваемый машиной на экране, и получает деньги. В ответ на вставленную карту и ответы пользователя система осуществляет последовательность действий (приложение А), которые обеспечивают пользователю ощущимый и значимый для него результат, а именно получение наличных.

Взаимодействие такого рода называется **вариантом использования** (приложение А; см. также главу 3). Вариант использования — это часть функциональности системы, необходимая для получения пользователем значимого для него, ощущимого и измеримого результата. Варианты использования обеспечивают функциональные требования. Сумма всех вариантов использования составляет **модель вариантов использования** (приложение Б, см. также главы 2, 3), которая описывает полную функциональность системы. Эта модель заменяет традиционное описание функций системы. Считается, что описание функций может ответить на вопрос: что система предположительно делает? Подход вариантов использования может быть охарактеризован добавкой трех слов в конец этого вопроса: *для каждого пользователя*? Эти три слова представляют собой очень важное дополнение. Они побуждают нас думать в понятиях результата, значимого для пользователя, а не

только в понятиях функций, которые хорошо бы иметь. Однако варианты использования — это не только средство описания требований к системе. Они также направляют ее проектирование, реализацию и тестирование, то есть они *направляют процесс разработки*. Основываясь на модели вариантов использования, разработчики создают серию моделей проектирования и реализации, которые осуществляют варианты использования. Тестеры тестируют реализацию для того, чтобы гарантировать, что компоненты модели реализации правильно выполняют варианты использования. Таким образом, варианты использования не только инициируют процесс разработки, но и служат для связи отдельных его частей. *Управляемый вариантами использования* означает, что процесс разработки проходит серии рабочих процессов, порожденных вариантами использования. Варианты использования определяются, варианты использования проектируются, и, в конце концов, варианты использования служат исходными данными, по которым тестеры создают тестовые примеры.

Поскольку варианты использования действительно управляют процессом, они не выделяются изолированно. Они разрабатываются в паре с архитектурой системы. Таким образом, варианты использования управляют архитектурой системы, а архитектура системы оказывает влияние на выбор вариантов использования. Соответственно и архитектура системы, и варианты использования развиваются по мере хода жизненного цикла.

Унифицированный процесс ориентирован на архитектуру

Роль архитектуры программы подобна роли архитектуры в строительстве зданий. Здание можно рассматривать с различных точек зрения — структура, службы, теплопроводность, водопровод, электричество и т. п. Строителям же необходимо видеть общую картину до начала строительства. Так и архитектура программной системы описывается различными представлениями будущей системы.

Понятие архитектуры программы включает в себя наиболее важные статические и динамические аспекты системы. Архитектура вырастает из требований к результату, в том виде, как их понимают пользователи и другие заинтересованные лица. Эти требования отражаются в вариантах использования. Однако они также зависят от множества других факторов, таких, как выбор платформы для работы программы (то есть компьютерной архитектуры, операционной системы, СУБД, сетевых протоколов), доступность готовых блоков многократного использования, (например, каркаса графического интерфейса пользователя, см. приложение B), соображения развертывания, унаследованные системы и нефункциональные требования (например, производительность и надежность). Архитектура — это представление всего проекта с выделением важных характеристик и затушевыванием деталей. Поскольку важность той или иной характеристики зависит, в частности, от правильности суждения, приходящей с опытом, результат построения архитектуры определяется людьми, которым поручена эта задача. Однако процесс помогает архитектору сконцентрироваться на правильных целях, таких, как понятность, легкость внесения изменений и возможность повторного использования.

Как связаны архитектура и варианты использования? Каждый продукт имеет функции и форму. Одно без другого не существует. В удачном продукте эти две стороны должны быть уравновешены. В этом примере функции соответствуют вариантам использования, а форма — архитектуре. Мы нуждаемся во взаимодействии между вариантами использования и архитектурой. Это вариант традиционной проблемы «курицы и яйца». С одной стороны, варианты использования должны, будучи реализованными, подойти к архитектуре. С другой стороны, архитектура должна предоставить возможности реализации любых понадобившихся сейчас и в будущем вариантов использования. Реально архитектура и варианты использования разрабатываются параллельно.

Таким образом, архитектор придает системе *форму*. Это означает, что форма, архитектура, должна быть спроектирована так, чтобы позволить системе развиваться не только в момент начальной разработки, но и в будущих поколениях. Чтобы найти такую форму, архитектор должен работать, полностью понимая ключевые функции, то есть ключевые варианты использования системы. Эти ключевые варианты использования составляют от 5 до 10% всех вариантов использования и крайне важны, поскольку содержат функции ядра системы. Проще говоря, архитектор совершает следующие шаги:

- Создает грубый набросок архитектуры, начиная с той части архитектуры, которая не связана с вариантами использования (так называемая платформа). Хотя эта часть архитектуры не зависит от вариантов использования, архитектор должен в общих чертах понять варианты использования до создания наброска архитектуры.
- Далее архитектор работает с подмножеством выделенных вариантов использования, каждый из которых соответствует одной из ключевых функций разрабатываемой системы. Каждый из выбранных вариантов использования детально описывается и реализуется в понятиях **подсистем** (приложение А; см. также подраздел «Классы группируются в подсистемы» главы 3), **классов** (приложение А) и компонентов (приложение А).
- После того как варианты использования описаны и полностью разработаны, большая часть архитектуры исследована. Созданная архитектура, в свою очередь, будет базой для полной разработки других вариантов использования.

Этот процесс продолжается до тех пор, пока архитектура не будет признана стабильной.

Унифицированный процесс является итеративным и инкрементным

Разработка коммерческих программных продуктов — это серьезное предприятие, которое может продолжаться от нескольких месяцев до года и более. Практически было бы разделить работу на небольшие куски или мини-проекты. Каждый мини-проект является итерацией, результатом которой будет приращение. Итерации относятся к шагам рабочих процессов, а приращение — к выполнению проекта. Для максимальной эффективности итерации должны быть *управляемыми*, то есть они должны выбираться и выполняться по плану. Поэтому их можно считать мини-проектами.

Разработчики выбирают задачи, которые должны быть решены в ходе итерации, под воздействием двух факторов. Во-первых, в ходе итерации следует работать с группой вариантов использования, которая повышает применимость продукта в ходе дальнейшей разработки. Во-вторых, в ходе итерации следует заниматься наиболее серьезными рисками. Последовательные итерации используют артефакты разработки в том виде, в котором они остались при окончании предыдущей итерации. Это мини-проекты, поскольку для выбранных вариантов использования проводится последовательная разработка — анализ, проектирование, реализация и тестирование, и в результате варианты использования, которые разрабатывались в ходе итерации, представляются в виде исполняемого кода. Разумеется, приращение не обязательно аддитивно. Разработчики могут заменять предварительный дизайн более детальным или зрелым, особенно на ранних фазах жизненного цикла. В течение более поздних фаз приращение обычно аддитивно.

На каждой итерации разработчики определяют и описывают уместные варианты использования, создают проект, использующий выбранную архитектуру в качестве направляющей, реализуют проект в компоненты и проверяют соответствие компонентов вариантам использования. Если итерация достигла своей цели — а так обычно и бывает — процесс разработки переходит на следующую итерацию. Если итерация не выполнила своей задачи, разработчики должны пересмотреть свои решения и попробовать другой подход.

Для получения максимальной экономии команда, работающая над проектом, должна выбирать только те итерации, которые требуются для выполнения цели проекта. Для этого следует расположить итерации в логической последовательности. Успешный проект будет выполняться в соответствии с правильным курсом, изначально запланированным разработчиками, лишь с небольшими отклонениями. Разумеется, до определенного предела, так, незамеченные ранее проблемы приведут к добавлению итераций или изменению их последовательности, и процесс разработки потребует больших усилий и времени. Минимизация незамеченных проблем является одной из целей снижения рисков.

Управляемый итеративный процесс имеет множество преимуществ.

- Управляемая итерация ограничивает финансовые риски затратами на одно приращение. Если разработчикам нужно повторить итерацию, организация тратит только усилия, затраченные на одну итерацию, а не стоимость всего продукта.
- Управляемая итерация снижает риск непоставки продукта на рынок в запланированные сроки. При раннем обнаружении соответствующего риска время, которое тратится на его нейтрализацию, вносится в план на ранних стадиях, когда сотрудники менее загружены, чем в конце планового периода. При традиционном подходе, когда серьезные проблемы впервые проявляются на этапе тестирования системы, время, требуемое для их устранения, обычно превышает время, оставшееся по плану для завершения всех работ, что почти всегда приводит к задержкам поставок.
- Управляемая итерация ускоряет темпы процесса разработки в целом, поскольку для более эффективной работы разработчиков и быстрого получения ими хороших результатов короткий и четкий план предпочтительнее длинного иично сдвигающегося.
- Управляемая итерация признает часто отвергаемый факт — что желания пользователей и связанные с ними требования не могут быть определены в начале

разработки. Они обычно уточняются в последовательных итерациях. Такой образ действий облегчает адаптацию к изменениям требований.

Эти концепции – управляемая вариантами использования, архитектурно-ориентированная и итеративная и инкрементная разработка – одинаково важны. Архитектура предоставляет нам структуру, направляющую нашу работу в итерациях, в каждой из которых варианты использования определяют цели и направляют нашу работу. Удаление одной из этих трех ключевых идей сильно уменьшит ценность Унифицированного процесса. Это как треножник. Без одной ножки треножник сразу же опрокинется.

Теперь, когда мы познакомились с основными ключевыми концепциями, настало время рассмотреть весь процесс, его жизненный цикл, артефакты, рабочие процессы, фазы и итерации.

Жизненный цикл Унифицированного процесса

Унифицированный процесс циклически повторяется. Эта последовательность повторений Унифицированного процесса, как показано на рис. 1.2, представляет собой жизненный цикл системы. Каждый цикл завершается поставкой выпуска продукта (приложение В, см. также главу 5) заказчикам.

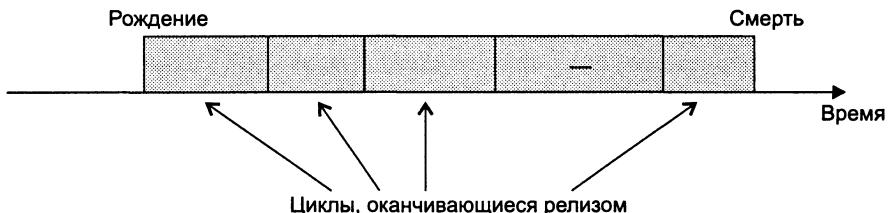


Рис. 1.2. Жизнь процесса, состоящая из циклов, от его рождения до смерти

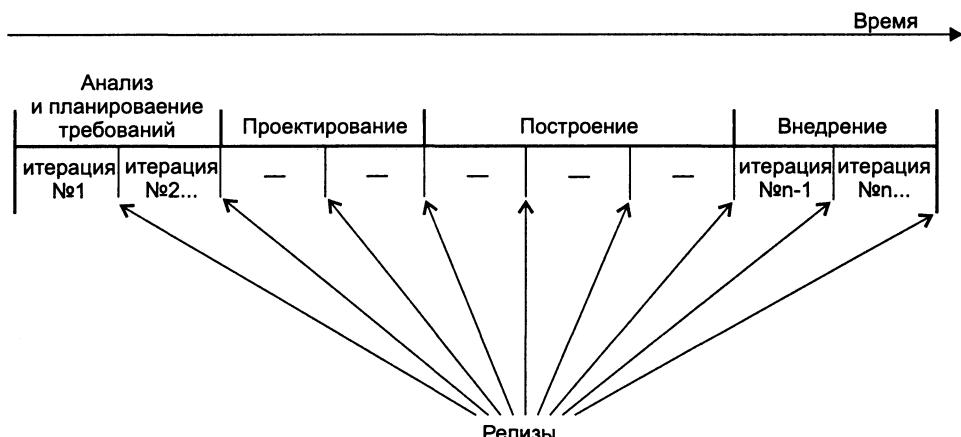


Рис. 1.3. Цикл с его фазами и итерациями

Каждый цикл состоит из четырех фаз — анализа и планирования требований, проектирования, построения и внедрения. Каждая **фаза** (приложение В), как будет рассмотрено ниже, далее подразделяется на итерации (рис. 1.3).

Продукт

Результатом каждого цикла является новый выпуск системы, а каждый выпуск — это продукт, готовый к поставке. Он включает в себя тело — исходный код, воплощенный в компоненты, которые могут быть откомпилированы и выполнены, плюс руководство и дополнительные компоненты поставки. Однако готовый продукт должен также быть приспособлен для нужд не только пользователей, а всех заинтересованных лиц, то есть всех людей, которые будут работать с продуктом. Программному продукту следовало бы представлять из себя нечто большее, чем исполняемый машинный код.

Окончательный продукт включает в себя требования, варианты использования, нефункциональные требования и варианты тестирования. Он включает архитектуру и визуальные модели — артефакты, смоделированные на Унифицированном языке моделирования. Фактически, он включает в себя все элементы, которые мы обсуждали в этой главе, поскольку они позволяют заинтересованным лицам — заказчикам, пользователям, аналитикам, проектировщикам, кодировщикам, тестерам и руководству — описывать, проектировать, реализовывать и использовать систему. Более того, именно эти средства позволяют заинтересованным лицам использовать систему и модифицировать ее от поколения к поколению.

Даже если исполняемые компоненты с точки зрения пользователей являются важнейшим артефактом, их одних недостаточно. Системное окружение меняется. Операционные системы, СУБД и сами компьютеры прогрессируют. По мере того, как цель понимается все лучше, изменяются и требования. Фактически, единственное, что постоянно в разработке программ, — это изменение требований. В конце концов разработчики вынуждены начинать новый цикл, а руководство вынуждено их финансировать. Для того чтобы эффективно выполнять новый цикл, разработчикам необходимо иметь полное представление о программном продукте (рис. 1.4):

- модель вариантов использования со всеми вариантами использования и их связями с пользователями;
- модель анализа, которая имеет две цели — уточнить детали вариантов использования и создать первичное распределение поведения системы по набору объектов, предоставляющих различные варианты поведения;
- модель проектирования, которая определяет (а) статическую структуру системы, такую, как подсистемы, классы и интерфейсы, и (б) варианты использования, реализованные в виде **коопераций** (приложение А; см. также раздел «Введение в разработку, управляемую вариантами использования» главы 3) между подсистемами, классами и интерфейсами;
- модель реализации, которая включает в себя компоненты (представленные исходным кодом) и раскладку классов по компонентам;
- модель развертывания, которая определяет физические компьютеры — узлы сети и раскладку компонентов по этим узлам;

- модель тестирования, которая описывает варианты тестов для проверки вариантов использования;
- и, разумеется, представление архитектуры.

Система также может включать в себя модель предметной области или бизнес-модель, описывающую контекст системы.

Все эти модели связаны. Вместе они полностью описывают систему. Элементы одной модели имеют **трассировочные** (приложение А; см. также подраздел «Связи между моделями» главы 2) зависимости вперед и назад, организуемые с помощью связей с другими моделями. Например, вариант использования (в модели вариантов использования) может быть **оттрасирован** на соответствующую реализацию варианта использования (в модели проектирования) и вариант тестирования (в модели тестирования). Трассировка облегчает понимание и внесение изменений.

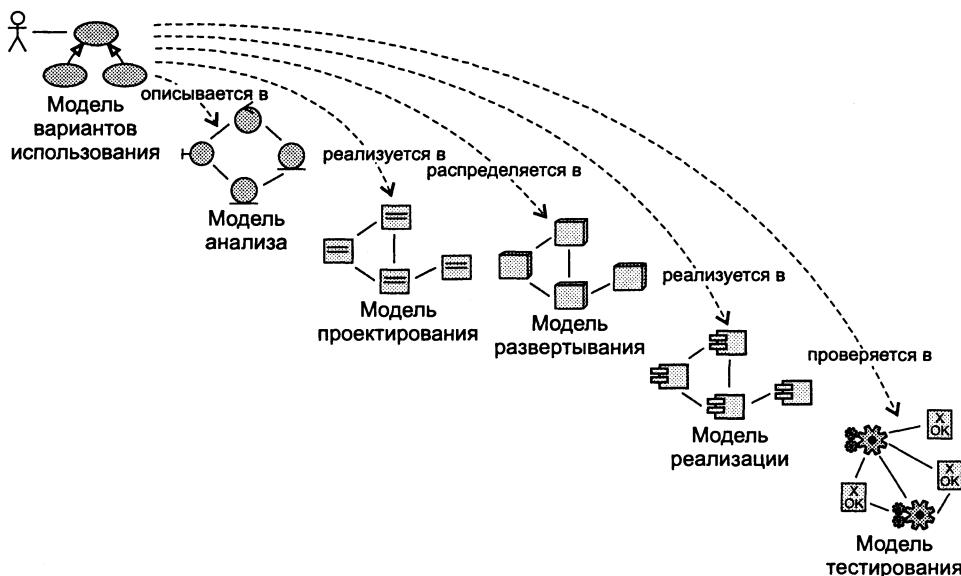


Рис. 1.4. Модели Унифицированного процесса.

Существует множество связей между моделями.

На рисунке показаны связи между моделью вариантов использования и другими моделями

Разделение цикла разработки на фазы

Каждый цикл осуществляется в течение некоторого времени. Это время, в свою очередь, делится на четыре фазы, показанные на рис. 1.5. В ходе смены последовательности моделей заинтересованные лица визуализируют происходящее на этих фазах. Внутри каждой фазы руководители или разработчики могут потерпеть неудачу в работе — но только на данной итерации и в связанном с ней приращении. Каждая фаза заканчивается **входом** — (приложение В; см. также главу 5). Мы опре-

деляем каждую веху по наличию определенного набора артефактов, например, некая модель документа должна быть приведена в предписанное состояние.

Вехи служат многим целям. Наиболее важная из них — дать руководителям возможность принять некоторые критические решения перед тем, как работа перейдет на следующую фазу. Вехи также дают руководству и самим разработчикам возможность отслеживать прогресс в работе при проходах этих четырех ключевых точек. Наконец, отслеживая время и усилия, затраченные на каждую фазу, мы создаем массив данных. Эти данные могут быть применены при оценке времени и персонала, нужных для выполнения других проектов, расчетах необходимости персонала в течение срока проектирования и контроля продвижения в проектировании.

На рисунке 1.5 в левой колонке перечислены рабочие процессы — определение требований, анализ, проектирование, разработка и тестирование. Кривые приближенно изображают (их не следует воспринимать слишком буквально) объемы рабочих процессов, выполняемых в каждой из фаз. Напомним, что каждая фаза обычно дополнительно подразделяется на итерации или мини-проекты. Типичная итерация захватывает все пять рабочих процессов, как это показано на рис. 1.5.

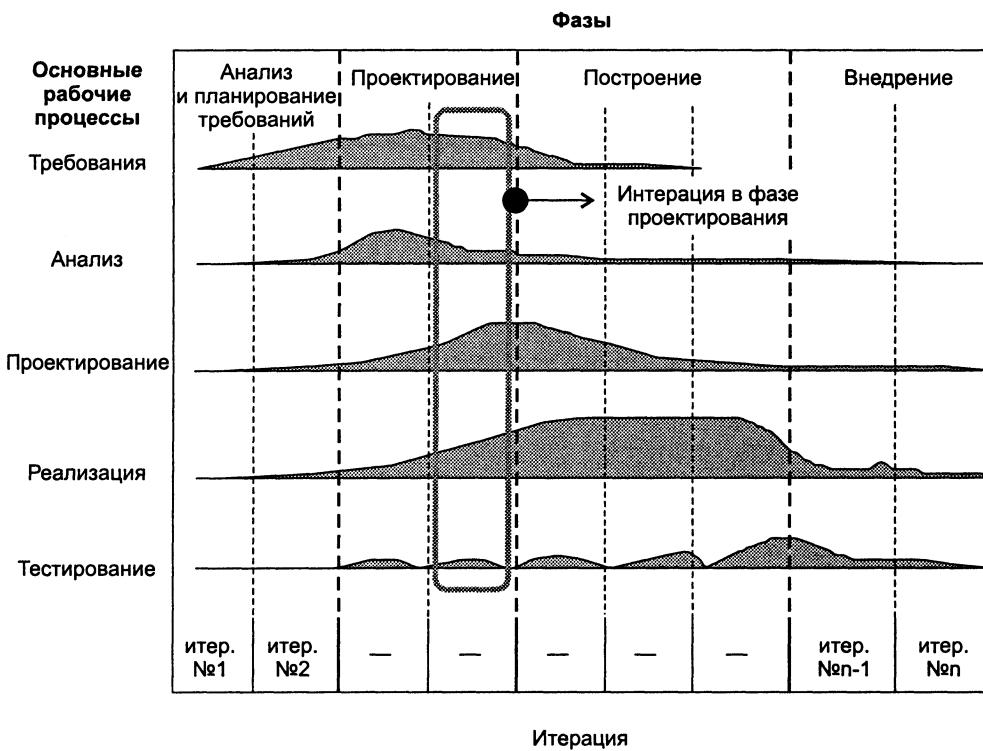


Рис. 1.5. Пять рабочих процессов — требования, анализ, проектирование, разработка и тестирование — происходят в течение четырех фаз — анализа и планирования требований, проектирования, разработки и внедрения

В ходе *фазы анализа и планирования требований* хорошая идея превращается в концепцию готового продукта, и создается бизнес-план разработки этого продукта. В частности, на этой фазе должны быть получены ответы на вопросы:

- Что система должна делать в первую очередь для ее основных пользователей?
- Как должна выглядеть архитектура системы?
- Каков план и во что обойдется разработка продукта?

Упрощенная модель вариантов использования, содержащая наиболее критичные варианты использования, дает ответ на первый вопрос. На этом этапе создается пробный вариант архитектуры. Обычно он представляет собой набросок, содержащий наиболее важные подсистемы. На этой фазе выявляются и расставляются по приоритетности наиболее важные риски, детально планируется фаза проектирования и грубо оценивается весь проект.

В ходе *фазы проектирования* детально описываются большинство вариантов использования и разрабатывается архитектура системы. Между архитектурой системы и системой имеется прямая и неразрывная связь. Поясним это на простейшем примере: архитектура подобна скелету, обтянутому кожей, но с минимальным количеством мышц (программ) между ними. Для того чтобы это создание могло совершать хотя бы простейшие движения, ему понадобится немало дополнительных мышц. Система, в отличие от архитектуры, — это полноценное тело, со скелетом, кожей и мышцами.

Поэтому архитектура определяется в виде представлений всех моделей системы, которые в сумме представляют систему целиком. Это означает, что существуют архитектурные представления модели вариантов использования, модели анализа, модели проектирования, модели реализации и модели развертывания. Представление модели реализации включает в себя компоненты для доказательства того, что архитектура выполнима. В ходе этой фазы определяются наиболее критичные варианты использования. Результатом выполнения этой фазы является **базовый уровень архитектуры** (приложение B, см. также подраздел «Шаги разработки архитектуры» главы 4).

В конце фазы проектирования менеджер проекта занимается планированием действий и подсчетом ресурсов, необходимым для завершения проекта. Ключевым вопросом в этот момент будет следующий: достаточно ли проработаны варианты использования, архитектура и план, и взяты ли риски под контроль настолько, чтобы можно было давать контрактные обязательства выполнить всю работу по разработке?

В ходе *фазы построения* происходит создание продукта — к скелету (архитектуре) добавляются мышцы (законченные программы). На этой фазе базовый уровень архитектуры разрастается до полной развитой системы. Концепции развиваются до продукта, готового к передаче пользователям. В ходе фазы разработки объем требуемых ресурсов вырастает. Архитектура системы стабильна, однако, поскольку разработчики могут найти лучшие способы структурирования системы, от них могут исходить предложения о внесении в архитектуру системы небольших изменений. В конце этой фазы продукт включает в себя все варианты использования, которые руководство и заказчик договорились включить в текущий выпуск. Правда, они могут содержать ошибки. Большинство дефектов будут обнаружены и исправлены в ходе фазы внедрения. Ключевой вопрос окончания фазы: удовлетворяет ли продукт требованиям пользователей настолько, что некоторым заказчикам можно делать предварительную поставку?

Фаза внедрения охватывает период, в ходе которого продукт существует в виде бета-выпуска или бета-версии. Небольшое число квалифицированных пользователей, работая с бета-выпуском продукта, сообщает об обнаруженных дефектах и недостатках. После этого разработчики исправляют обнаруженные ошибки и вносят некоторые из предложенных улучшений в главный выпуск, подготавливаемый для широкого распространения. Фаза внедрения включает в себя такие действия, как производство тиража, тренинг сотрудников заказчика, организацию поддержки по горячей линии и исправление дефектов, обнаруженных после поставки. Команда поддержки часто разделяет найденные дефекты на две категории: дефекты, оказывающие настолько значительный эффект на работу с программой, что это оправдывает немедленный выпуск дельта-версии, и дефекты, исправление которых можно отложить до следующего выпуска.

Интегрированный процесс

Унифицированный процесс основан на компонентах. Он использует новый стандарт визуального моделирования, Унифицированный язык моделирования и базируется на трех ключевых идеях – вариантах использования, архитектуре и итеративной и инкрементной разработке. Для того чтобы заставить эти идеи работать, необходим многоплановый процесс, поддерживающий циклы, фазы, рабочие процессы, снижение рисков, контроль качества, управление проектом и конфигурацией. Унифицированный процесс создает каркас, позволяющий объединить все эти различные аспекты. Этот каркас также работает, как зонтик, под которым поставщики утилит и разработчики могут создавать утилиты для автоматизации процесса, поддержки отдельных рабочих процессов, построения всего спектра моделей и интеграции работ в течении жизненного цикла и последовательности моделей.

Задача этой книги – описать Унифицированный процесс, делая упор на инженерные аспекты, на три ключевые идеи (варианты использования, архитектуру и итеративную и инкрементную разработку), на компоненто-ориентированную разработку и на использование Унифицированного языка моделирования. Мы опишем четыре фазы и различные рабочие процессы, но не будем детально разбирать проблемы управления, такие как планирование проекта, планирование использования ресурсов, снижение рисков, управление конфигурацией, определение размерности и контроль качества, и лишь кратко коснемся проблем автоматизации процесса.

2 Четыре «П» — персонал, проект, продукт и процесс — в разработке программного обеспечения

Итогом проекта (приложение Б) по разработке программного обеспечения является продукт, в создании которого принимает участие множество различных людей.

Процесс разработки программного обеспечения направляет усилия людей, вовлеченных в проект. Легче всего представить его в виде шаблона, предписывающего шаги, необходимые для выполнения проекта. Обычно процесс автоматизируется при помощи утилиты или набора утилит. Это показано на рис. 2.1.

В тексте этой книги мы будем использовать понятия *персонал, проект, продукт, процесс* (приложение Б) и *утилиты*, которые мы определим следующим образом:

- *Персонал*: архитекторы, разработчики, тестеры и их руководство, а также пользователи, заказчики и другие заинтересованные лица — являются исходной движущей силой программного проекта. Персонал — это реальные люди, в отличие от абстрактной конструкции *сотрудники*, которую мы введем позже.
- *Проект*: организационная сущность, при помощи которой происходит управление разработкой программного обеспечения. Результатом проекта является выпущенный продукт.
- *Продукт*: артефакты, создаваемые в течение жизни проекта, такие как *модели* (приложение А), тексты программ, исполняемые файлы и документация.
- *Процесс*: процесс создания программного обеспечения — это определение полного набора видов деятельности, необходимых для преобразования требований пользователя в продукт. Процесс служит шаблоном для создания проекта.
- *Утилиты*: программы, используемые для автоматизации определенных в процессе видов деятельности.

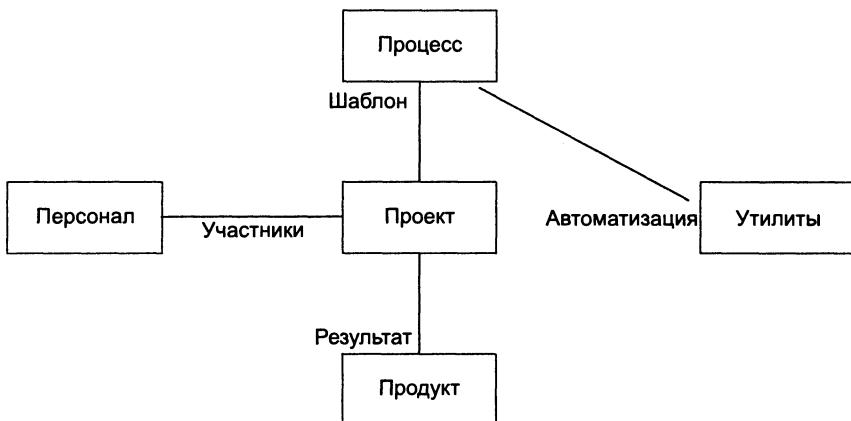


Рис. 2.1. Четыре «П» в разработке программного обеспечения

Персонал решает все

Персонал участвует в разработке программного обеспечения в течение всего жизненного цикла. Он финансирует продукт, планирует его разработку, разрабатывает, управляет, тестирует и извлекает из него выгоду. Из этого следует, что процесс, направляющий разработку, должен быть ориентирован на персонал. Иначе говоря, людям, работающим с этим процессом, должно быть удобно это делать.

Процессы разработки влияют на персонал

Способ организации и управления программным проектом сильно влияет на участвующий в проекте персонал. Важную роль играют такие концепции, как выполнимость, управление рисками, организация команды, планирование проекта и его понятность.

- *Выполнимость проекта.* Немногие люди будут рады работать над проектами, которые считаются невыполнимыми, — никто не хочет тонуть вместе с кораблем. Как мы упоминали в главе 1, итеративный подход к разработке позволяет оценить выполнимость проекта на ранних стадиях разработки. Работу над заранее невыполнимыми проектами можно прекращать на ранней стадии, что уменьшает моральные потери.
- *Управление рисками.* Если люди чувствуют, что риски не были проанализированы и, по возможности, уменьшены, это может их тревожить. Исследование значимых рисков на ранних стадиях проекта смягчает эти проблемы.
- *Структура команды.* С максимальной эффективностью люди работают в маленьких группах, от шести до восьми человек. Процесс, который предполагает выполнение малыми группами серьезной работы, такой, как оценка отдельного риска, разработка подсистемы (приложение А) или осуществление итерации, должен обеспечивать такую возможность. Удачная архитектура с хорошо определенными интерфейсами (приложение А, см. также главу 9) между подсис-

темами и компонентами (приложение А, см. также главу 10) допускает такое распределение усилий.

- *План проекта.* Если люди не верят в то, что план реален, мораль стремительно падает — люди не любят ходить на работу, зная, что как бы усердно они ни работали, им никогда не удастся получить ожидаемый от них результат. Технологии, используемые на фазах анализа и планирования требований и проектирования, позволяют разработчикам получать информацию о том, каким должен быть результат проекта, — то есть что должен делать продукт. Поскольку всегда полезно знать, что должен делать продукт, реалистичный план должен быть создан сразу же, как только найдется время для того, чтобы четко описать цели проекта. Это ослабит терзания персонала по поводу того, что «мы никогда не закончим».
- *Понятность проекта.* Персонал предпочитает знать, что он делает. Люди хотят понимать общую картину. Описание архитектуры предоставляет такую возможность каждому участнику проекта.
- *Ощущение законченности.* В итеративном жизненном цикле персонал часто получает отклики пользователей, что помогает персоналу закончить свою работу. Частые отклики и инициируемая ими доводка увеличивают темп работ. Быстрый темп работы в комбинации с частой реакцией на работу продукта усиливают ощущение законченности.

Роли будут меняться

Поскольку ключевые виды деятельности в процессе разработки выполняются персоналом, существует необходимость в таком унифицированном процессе разработки, поддерживаемом утилитами и Унифицированным языком моделирования (приложение В), который делал бы работу персонала более эффективной. То есть необходим процесс, позволяющий разработчикам успешнее (по таким параметрам, как время выхода на рынок, качество и цена) создавать программы. Этот процесс должен побуждать разработчиков определять требования, которые лучше описывают запросы пользователей. Этот процесс должен побуждать их выбирать архитектуру, которая позволяет создавать системы своевременно и с малыми затратами. Хороший процесс разработки имеет и другие преимущества. Он позволяет строить более сложные системы. Мы отмечали в главе 1, что реальный мир становится все более сложным, а заказчики требуют все более сложных систем. Бизнес-процессы и соответствующее им программное обеспечение будут жить долго. Поскольку окружающая среда будет продолжать изменяться в течение всего жизненного цикла программ, программные системы следует проектировать так, чтобы можно было в течение долгого времени вносить в них изменения.

Для того чтобы понимать и поддерживать очень сложные бизнес-процессы и воплощать их в программы, разработчики должны научиться работать совместно с множеством других разработчиков. Для эффективной работы все больших и больших команд необходим процесс, обеспечивающий правильное приложение их усилий. Правильное приложение усилий приведет к «умной работе» разработчиков, то есть их деятельность сосредотачивается исключительно на том, что приносит значимый для заказчика результат. Первым шагом в этом направлении яв-

ляется моделирование вариантов использования, ориентирующее все последующие усилия на тех проблемах, решение которых хочет получить пользователь. Следующий шаг — архитектура, которая позволит системе в будущем продолжать развиваться. Третий шаг — покупка или повторное использование максимально возможного количества программ. Это, в свою очередь, возможно только в том случае, если существует логичный способ объединения повторно используемых компонентов со свежеразработанными частями проекта.

В будущем многие программисты будут заниматься автоматизируемой деятельностью. Благодаря автоматизированным процессам и повторно используемым компонентам они смогут разрабатывать более сложные программы. В обозримом будущем персонал в разработке программ будет решать все. В конце концов, именно существование хороших людей принесло нам успех. Мы считаем, что следует повышать эффективность их работы и поручать им то, что могут делать только люди, — изобретать, находить новые возможности, использовать оценки и мнения, общаться с заказчиками и пользователями и понимать быстро изменяющийся мир.

Размещение «Ресурсов» внутри «Сотрудников»

В организации, производящей программное обеспечение, персонал заполняет множество различных позиций. Подготовка людей к занятию этих позиций включает соответствующее образование и специальную подготовку, сопровождаемую самостоятельной работой под присмотром преподавателей. Организация решает важную задачу, переводя человека из положения абстрактного «ресурса» на конкретную позицию «сотрудника».

Мы выбрали слово **сотрудник** (приложение В) для описания позиции, которая может быть поручена персоналу и которую может занимать работник [39]. *Сотрудник* — это роль, которую может играть человек в разработке программного обеспечения, например спецификатор вариантов использования, архитектор, инженер по компонентам или тестер интеграции. Мы не использовали вместо *сотрудника* термин *роль* по двум основным причинам: это слово имеет предопределенное значение с другим смыслом в Унифицированном языке моделирования, и концепция сотрудника должна быть очень конкретна. Мы хотим думать об отдельном сотруднике, как о позиции, занимаемой отдельным лицом. Мы также хотели бы использовать термин *роль* при обсуждении ролей сотрудников. Сотрудник может выполнять некоторые роли по отношению к другим сотрудникам в различных рабочих процессах. Например, сотрудник-инженер по компонентам может участвовать в нескольких рабочих процессах и в каждом из них исполнять отдельную роль.

Каждый сотрудник (то есть экземпляр сотрудника) соотносится с определенным набором видов деятельности, например, необходимых для разработки подсистемы. Для эффективной работы сотруднику необходима информация о том, как выполнять эти виды деятельности. Им необходимо понимание того, как их роли соотносятся с ролями других сотрудников. В то же время, если они выполняют свою работу, у них должны быть адекватные утилиты. Утилиты не только помогают сотруднику выполнять текущую деятельность, но и защищают его от не относящейся к делу информации. Для выполнения этих условий Унифицированный процесс формально определяет позиции — то есть сотрудников, которые могут участвовать в процессе.

Рисунок 2.2 иллюстрирует, как люди могут быть различными сотрудниками.

Сотрудник может также быть представлен совместно работающей группой лиц. Например, сотрудник «Архитектор» может быть реализован в виде архитектурного отдела.

Каждый сотрудник разработки программ имеет набор обязанностей и осуществляет набор видов деятельности.

При присвоении ресурсов сотрудникам менеджер проекта должен определить способности каждого человека и сравнить их с необходимой для данного сотрудника компетенцией.

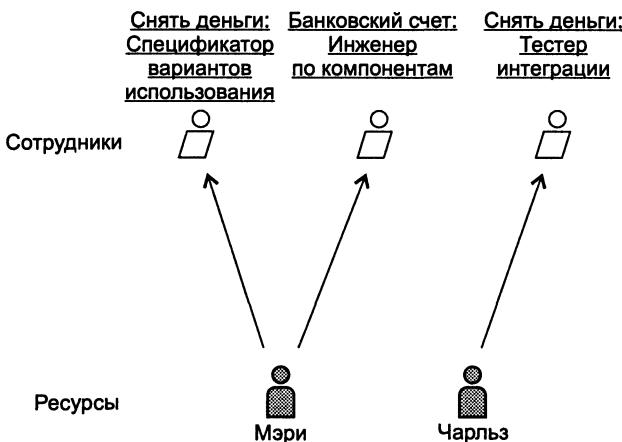


Рис. 2.2. Сотрудники и реализующие их ресурсы

Это нелегкая задача, особенно когда использование Унифицированного процесса только начато. Навыки ресурсов (то есть реальных людей) должны соответствовать способностям, требующимся для необходимых в проекте сотрудников. Некоторые способности, необходимые сотрудникам, могут быть приобретены путем обучения, в то время как для овладения другими необходим исключительно опыт работы. Так, например, можно обучить человека специфицировать варианты использования, но архитекторами обычно становятся, набирая соответствующий опыт.

За время жизни проекта один человек может сменить множество специальностей (то есть сотрудников). Мэри может начать работу в проекте спецификатором вариантов использования, а затем стать инженером по компонентам.

При выделении ресурсов менеджер проекта должен минимизировать перебрасывание артефактов от одного ресурса к другому, чтобы процесс протекал настолько гладко, насколько это возможно. Например, инженер по вариантам использования, разрабатывающий вариант использования *Получить наличные* (приложение А; см. также главу 7), хорошо разберется в задачах класса *Банковский счет* (приложение А), и будет вполне логично сделать его впоследствии инженером по компонентам, ответственным за класс *Банковский счет*. В противном случае для этой работы надо будет обучать нового человека. Это возможно, но менее эффективно — из-за потери информации, возможности непонимания и других причин.

Проект порождает продукт

Проект по разработке программного обеспечения заканчивается новым выпуском продукта. В первом проекте жизненного цикла (или первом цикле разработки, иногда называемом «целинным проектом») разрабатывается и выпускается исходная система или продукт. В последовательных циклах проектирования система проходит череду множества выпусков. Более полное представление об управлении проектом можно получить из [9] и [59].

В ходе жизненного цикла команда, работающая над проектом, сталкивается с изменениями, итерациями и организационными шаблонами, которыми будет определяться развитие проекта:

- **Последовательность изменений.** Результатом проекта по разработке программного обеспечения является продукт, но путь, которым идет разработка, есть серия изменений. Этот факт жизни проекта проявляется в умах разработчиков по мере продвижения по фазам и итерациям. Каждый цикл, каждая фаза и каждая итерация изменяют систему. Первый цикл разработки — особый случай, превращающий систему из ничего во что-то. Каждый цикл завершается *выпуском*. Изменения продолжают появляться и на уровне, более высоком, чем последовательность циклов, в течение *поколений*.
- **Серии итераций.** Внутри каждой фазы или цикла деятельность сотрудников протекает в виде серий итераций. В ходе каждой итерации происходит реализация набора взаимосвязанных вариантов использования или уменьшение определенных рисков. В ходе итерации разработчики проделывают серию рабочих процессов — требования, проектирование, реализация и тестирование. В любой итерации выполняются все эти рабочие процессы, мы рассматриваем итерацию как *мини-проект*.
- **Организационный шаблон.** Проект включает в себя команду разработчиков, которые должны получить результат, не выходя при этом за пределы бизнес-ограничений, таких, как время, деньги и качество. Люди принимают на себя функции различных сотрудников. Идея «процесса» — создать образец, внутри которого люди, исполняя функции сотрудников, выполняют проект. Этот шаблон или образец определяет типы сотрудников, необходимых для работы над проектом, и артефакты, с которыми они должны работать. Процесс также содержит множество подсказок, эвристик и примеров документирования, которые помогают назначенному персоналу делать свою работу.

Продукт — это больше, чем код

В контексте Унифицированного процесса разрабатываемый продукт — это программная система. Понятие *продукта* здесь не ограничивается поставляемым кодом, а относится ко всей системе целиком.

Что такое программная система?

Эквивалентна ли программная система машинному коду, исполняемым файлам? В общем да, но что такое машинный код? Это описание! Это описание в двоичном виде, которое может «прочесть» и «понять» компьютер.

Эквивалентна ли программная система исходным текстам программ? Является ли она также описанием, которое создано программистами и может быть прочитано и понято компилятором? Да, ответ может быть и таким.

Мы можем продолжать задавать вопросы в том же духе о виде программной системы в понятиях подсистем, классов, **диаграмм взаимодействий** (приложение А), **диаграмм состояний** (приложение А) и других артефактов. Будет ли это системой? Да, все это — ее части. Как насчет требований, тестирования, продаж, тиражирования, инсталляции и работы? Будет ли системой это? Да, все это также части системы.

Системой являются все артефакты, существующие в понятном компьютеру или человеку виде и представляемые компьютерам, сотрудникам или заинтересованным лицам. Под компьютерами мы понимаем утилиты, компиляторы или целевые компьютеры. В число сотрудников входят руководители, архитекторы, разработчики, тестеры, работники отдела продаж, администраторы и другой персонал. Заинтересованными лицами мы называем лиц, финансирующих разработку, пользователей, сотрудников отдела продаж, менеджеров проектов, менеджеров продуктов, производственников и других причастных к разработке людей. В этой книге мы будем использовать понятие *сотрудник* для всех этих трех категорий, если прямо не будет заявлено другое.

Артефакты

Артефакт (приложение В) — это общее название для любых видов информации, создаваемой, изменяемой или используемой сотрудниками при создании системы. Примерами артефактов могут служить диаграммы Унифицированного языка моделирования и связанный с ними текст, наброски и **прототипы** (приложение В, см. также главы 7 и 13) пользовательских интерфейсов, компоненты, планы тестирования и тестовые примеры (см. главу 11).

Можно выделить два основных типа артефактов — технические артефакты и артефакты управления. В этой книге основное внимание уделяется техническим артефактам, создаваемым в ходе различных фаз процесса (определение требований, анализ, проектирование, разработка и тестирование).

Однако в ходе разработки программ создаются также и артефакты управления. Некоторые артефакты управления существуют недолго — только в течение срока жизни проекта. В их число входят бизнес-план, план разработки (включая планы выпусков и итераций), план подбора сотрудников (то есть назначения людей на различные позиции в проекте) и разбивка видов деятельности сотрудника в соответствии с планом. Эти артефакты описываются посредством текста или диаграмм с использованием любых средств визуализации, необходимой для описания обязательств, даваемых командой разработчиков заинтересованным лицам. Артефакты управления также включают спецификации среды разработки — программного обеспечения для автоматизации процесса разработки и компьютерных платформ, необходимых для разработки и хранения технических артефактов.

Система содержит набор моделей

Наиболее интересный тип артефактов, с которыми имеет дело Унифицированный процесс, — это модели. Каждый сотрудник нуждается в своем уникальном видении системы (рис. 2.3). При разработке Унифицированного процесса мы иденти-

фицировали всех сотрудников и все варианты обозрения системы, которые могут им понадобиться. Отобранные варианты обозрения системы для всех сотрудников были структурированы в нечто большее — в модели, чтобы сотрудник мог получить из набора моделей любой конкретный вариант обозрения.

Построение системы, таким образом, это построение моделей и использование разных моделей для описания различных сторон системы. Выбор модели для системы — это одно из наиболее важных решений, принимаемых командой разработчиков.

В главе 1 мы получили представление об основных моделях Унифицированного процесса (рис. 2.4).

Унифицированный процесс предлагает для начала разработки тщательно подобранный набор моделей. Этот набор моделей дает варианты обозрения системы для всех сотрудников, включая сотрудников отдела продаж, пользователей и менеджеров проекта. Он подобран так, чтобы удовлетворить потребность сотрудников в информации.



Рис. 2.3. Сотрудники, занятые в разработке программного обеспечения (некоторые — в единственном числе, другие — по несколько человек)

Что такое модель?

Модель — это абстракция, описывающая моделируемую систему с определенной точки зрения и на определенном уровне абстрагирования [57]. Под точкой зрения мы понимаем, например, представление спецификации или представление проектирования.

Модели — это абстракции системы, которые создаются архитекторами и проектировщиками. Например, сотрудники, моделирующие функциональные требования к системе, представляют ее себе как нечто, имеющее снаружи пользователей, а внутри варианты использования. Они не обращают внимания на то, как выглядит система изнутри, их интересует только, как ее видят пользователи. Сотрудники, создающие проект, думают о системе в понятиях структурных элементов,

таких, как подсистемы или классы; их интересует, как эти элементы работают в данном контексте, как они сотрудничают для реализации вариантов использования. Они понимают, как работают эти абстрактные элементы, и держат в уме их интерпретацию.

Каждая модель — это самодостаточное представление системы

Модель — это семантически завершенная абстракция системы. Это самодостаточное представление в том смысле, что сотрудник, работающий с моделью, не нуждается в дополнительной информации (например, из других моделей) для ее понимания.

Идея самодостаточности означает также, что существует только одно толкование того, что произойдет с системой при наступлении события, описываемого в модели. Разработчики могут быть в этом уверены. Добавим к обсуждению системы, что модель также может описывать взаимодействие между системой и ее окружением. Так, кроме моделируемой системы, модель может содержать также элементы, описывающие соответствующие части ее окружения, называемые **актантами** (приложение А, см. также главу 7).



Рис. 2.4. Основной набор моделей Унифицированного процесса¹

Большинство технических моделей определены в тщательно подобранным подмножестве UML. Так, например, модель вариантов использования включает в себя варианты использования и актантов. Этого вполне достаточно человеку, использующему эту модель, чтобы в ней разобраться. Модель проектирования определяет подсистемы и классы системы и порядок их взаимодействия для реализации вариантов использования. Эти две модели — вариантов использования и проектирования — определяют два различных, но взаимно согласованных представления тех действий, которые совершил система, получив набор внешних сообщений от действующего лица. Они различны, потому что предназначены для использования различными сотрудниками в различных целях и для различных задач. Модель вариантов использования — это внешнее представление системы, а модель проектирования — внутреннее. Модель вариантов использования определяет применение системы, а модель проектирования описывает ее построение.

Модель изнутри

Модель всегда отождествляется с моделируемой системой. Этот элемент системы служит затем контейнером для других элементов. **Подсистема верхнего уровня**

¹ В понятиях UML эти «папки» отражают «бизнес-сущности» (или «единицы работы») Унифицированного процесса, но не элементы модели для конкретной системы. См. также разъяснение в разделе «Введение» главы 7.

(приложение Б) соответствует создаваемой системе. В модели вариантов использования система состоит из вариантов использования, в модели проектирования та же система состоит из подсистем, интерфейсов и классов. Также система содержит кооперации (приложение А), которые определяют все участвующие в них классы или подсистемы. Она также может содержать некоторую дополнительную информацию, например диаграммы состояний или взаимодействий. В модели проектирования каждая подсистема может сама служить контейнером для других подсистем. Это подразумевает существование в этой модели иерархии элементов.

Связи между моделями

Система содержит все **связи** (приложение А) между элементами модели и все ограничения, а также разграничения между элементами различных моделей. Итак, система — это не просто набор моделей, но и набор связей между ними.

Так, каждый вариант использования в модели вариантов использования имеет связь с соответствующим элементом модели анализа (и т. д.). Эта связь в UML называется связью трассировки или просто **трассировкой** (приложение А). На рисунке 2.5 эти связи изображены только в одну сторону.

Существуют также связи трассировки между, например, кооперациями в модели проектирования и кооперациями в модели анализа или компонентами в модели реализации и подсистемами в модели проектирования. Таким образом, при помощи трассировки мы можем связывать элементы одной модели с элементами другой.

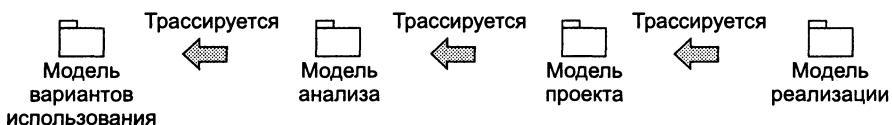


Рис. 2.5. Модели тесно связаны между собой трассировкой

Тот факт, что элементы двух моделей связаны между собой, не изменяет их поведения внутри тех моделей, к которым они принадлежат. Связи трассировки между элементами различных моделей не несут семантической информации, которая могла бы помочь понять взаимосвязанные модели; они просто соединяют модели. Существование трассировки имеет большое значение для разработки программного обеспечения. Это делает модели более понятными и облегчает перенос изменений из одной модели в другую.

Процесс направляет проекты

Понятие *процесс* трактуется по-разному. Оно употребляется в различном смысле для различных контекстов, таких как бизнес-процесс, процесс разработки, программный процесс. В контексте Унифицированного процесса мы считаем ключевым «бизнес»-процессом разработку программного обеспечения, то есть организацию разработки и сопровождения программ (или разработку бизнеса по произ-

водству программного обеспечения [49]. В этом бизнесе существуют и другие процессы, например процесс сопровождения, характеризуемый взаимодействием с пользователями продукта, и процесс продаж, который начинается с получения заказа и оканчивается поставкой продукта. Однако в этой книге мы сосредоточимся на процессе разработки [27].

Процесс как шаблон проекта

В Унифицированном процессе слово *процесс* относится к концепции, работающей подобно шаблону, который может быть многократно использован для создания его экземпляров. Это походит на идею класса, который может быть использован для создания объектов класса в объектно-ориентированной парадигме. *Экземпляр процесса* — это синоним *проекта*.

В этой книге *процесс разработки программного обеспечения* — это название для полного набора действий, необходимых для преобразования требований пользователей в согласованный набор артефактов, представляющих собой программный продукт, а позднее — для преобразования изменений в этих требованиях в новый согласованный набор артефактов.

Слово *требования* используется в традиционном смысле, означающем «то, что необходимо». В начале работы эти нужды могут пониматься не полностью. Для более полного определения требований мы должны полнее разобраться в процессах заказчиков и окружении, в котором работают пользователи.

Значимый результат процесса — это согласованный набор артефактов, базовый уровень, представляющий одну прикладную систему или семейство систем и являющийся программным продуктом.

Процесс — это определение набора действий, а не его выполнение.

И наконец, процесс покрывает не только первый цикл разработки (первый выпуск), но и большую часть последующих. В последующих выпусках в экземпляр процесса вносятся последовательные изменения требований. На выходе, соответственно, получаются измененные наборы артефактов.

Связанные деятельности образуют рабочие процессы

Мы будем описывать процесс разработки в понятиях рабочих процессов, где рабочий процесс представляет собой набор видов деятельности. Откуда возьмутся эти рабочие процессы? Мы не сможем получить их, разбивая процесс на множество маленьких взаимодействующих подпроцессов. Для определения путей разбивки процесса на небольшие куски нельзя использовать традиционные диаграммы. Эффективных способов определения структуры рабочего процесса не существует.

Вместо этого мы первым делом должны определить, какие типы сотрудников участвуют в процессе. Затем мы определяем артефакты, которые должны быть созданы в ходе процесса каждым типом сотрудников. Это определение, разумеется, не придет к вам, как озарение. Унифицированный процесс предполагает наличие большого опыта у людей, определяющих вероятные наборы артефактов и сотрудников. После того как мы определили эти наборы, мы можем определить, как для

различных сотрудников протекает процесс и как они создают артефакты и используют артефакты, созданные другими сотрудниками. На рисунке 2.6 мы видим **диаграмму деятельности** (приложение А), которая описывает рабочий процесс в моделировании вариантов использования. Обратите внимание на «плавательные дорожки» (приложение А) — по одной на каждого сотрудника — как работа перетекает от одного сотрудника к другому и как сотрудниками в этом потоке осуществляются их виды деятельности (изображенные в виде шестеренок).

Теперь мы можем легко отыскать виды деятельности, которые должны быть выполнены сотрудниками при их активации. Такая подборка «сотрудник — виды деятельности» будет нести важную информацию о каждом отдельном работнике. Кроме того, мы сразу заметим, если отдельного сотрудника потребуется использовать в ходе рабочего процесса более чем один раз.

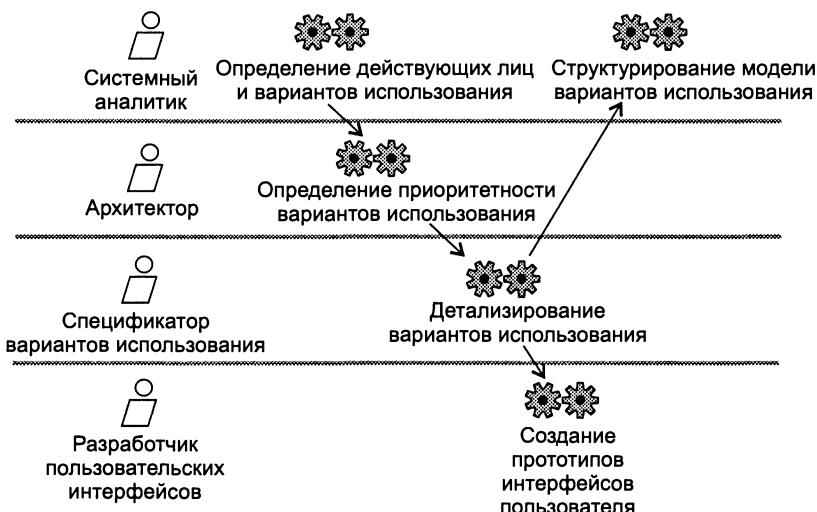


Рис. 2.6. Рабочий процесс с деятельностями и сотрудниками в плавательных дорожках

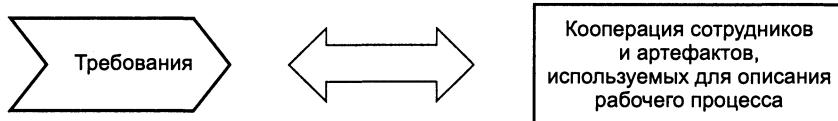


Рис. 2.7. «Рыба» — набросок рабочего процесса

Другими словами, мы определяем процесс разработки по кускам, которые называем **рабочими процессами** (приложение В). В терминах UML процесс — это **стереотип** (приложение А) кооперации, в которой участвуют сотрудники и артефакты. Точно так же, как сотрудники и артефакты участвуют в этом рабочем процессе, они могут участвовать (и участвуют) и в других рабочих процессах. Мы будем и дальше использовать нотацию рабочих процессов, показанную на рис. 2.7.

Примером рабочего процесса может быть рабочий процесс Определение требований. В нем участвуют следующие сотрудники: системный аналитик, архитектор, спецификатор вариантов использования и разработчик пользовательских интерфейсов. Он включает следующие артефакты: модель вариантов использования, варианты использования и еще несколько других. Другие примеры сотрудников — инженеры по компонентам и тестеры интеграции. Другие примеры артефактов — реализации вариантов использования (приложение Б, см. также главы 8 и 9), классы, подсистемы и интерфейсы.

Специализированные процессы

Не существует такого процесса разработки программного обеспечения, который мог бы применяться во всех случаях! Процессы варьируются, поскольку они создаются в различном контексте, предназначены для разработки различных типов систем и имеют различные типы бизнес-ограничений (планы, цена, качество и надежность). Соответственно, реальный процесс разработки программного обеспечения должен иметь возможность адаптироваться и конфигурироваться под текущие нужды конкретного проекта и/или организации. При разработке Унифицированного процесса в него была включена возможность специализации (о разработке процесса см. [42]). Это обобщенный процесс, каркас процесса. Любая организация, использующая Унифицированный процесс, со временем специализирует его, приспосабливая к своим запросам (типам приложений, платформам и т. д., о специализации процесса см. [44]).

Унифицированный процесс может быть специализирован под различные приложения и разные требования организаций. В то же время желательно, чтобы по крайней мере внутри организации процесс был более-менее единым. Это единство позволит легко обмениваться компонентами, перебрасывать персонал и руководителей с одного проекта на другой и иметь возможность сравнения показателей выполнения работ.

Основными факторами, вызывающими разницу в процессах, будут.

- *Организационные факторы.* Структура организации, культура организации, организация и управление проектом, имеющиеся в наличии способности и знания, предыдущий опыт и созданные программные системы.
- *Факторы предметной области.* Предметная область приложения, бизнес-процесс поддержки, сообщество пользователей и предложения конкурентов.
- *Факторы жизненного цикла.* Время до выхода на рынок, прошедшая часть жизненного цикла, проведенные при разработке программы экспертизы технологии и персонала и планируемые будущие выпуски.
- *Технические факторы.* Язык программирования, средства разработки, базы данных, каркасы и положенные в основу разработки «стандартные» архитектуры, протоколы связи и распространение.

Это и есть причины. Какой эффект они дадут? Мы можем решить убрать из Унифицированного процесса сотрудников и артефакты, чтобы он лучше подходил для малоопытных в разработке программ организаций. Можно также решить расширить процесс новыми, не определявшимися до сих пор сотрудниками или артефактами, поскольку эти расширения сделают процесс эффективнее в случае

вашего проекта. Вы можете также изменить способ задания некоторых артефактов, вы можете задать в своем определении другие структуры. Наш опыт говорит, что в первом проекте персонал активно использует стандартные средства Унифицированного процесса. Со временем люди набираются опыта и создают свои небольшие расширения.

Что в структуре Унифицированного процесса позволяет его специализировать [42]? Ответ прост, но поначалу не слишком понятен. Сам Унифицированный процесс в действительности построен с использованием объектов — вариантов использования, коопераций и классов. Эти классы, разумеется, не программные. Это бизнес-объекты, такие как сотрудники и артефакты. Они могут быть специализированы или заменены другими без изменения структуры процесса. Когда в последующих главах мы будем описывать рабочие процессы, вы увидите, что для описания мы используем объекты. Этими объектами будут сотрудники и артефакты.

Похвала процессу

Применение единого процесса как внутри команды разработчиков, так и при взаимодействии между командами очень выгодно, потому что:

- Каждый член команды разработчиков может понять, что он делает для разработки продукта.
- Разработчики лучше понимают, что делают другие разработчики — на предыдущих или последующих стадиях того же проекта, в похожих проектах той же фирмы, в другой географической точке и даже в проектах других компаний.
- Руководители, даже те, которые не могут читать тексты программ, могут, благодаря диаграммам и чертежам, понять, что делают разработчики.
- Разработчики и руководители могут переводиться из проекта в проект или из отдела в отдел, и при этом им не нужно изучать новый процесс.
- Обучение в компании можно стандартизировать. Обучение может происходить в колледже или на краткосрочных курсах.
- Курсы обучения разработке программного обеспечения повторяются, а это значит, что можно с достаточной точностью планировать и выделяемые на них деньги, и получаемый эффект.

Несмотря на эти преимущества, некоторые люди продолжают говорить, что стандартный процесс не решает «действительно сложных проблем». На это мы отвечаем просто: разумеется, не решает. Проблемы решают люди. Но хороший процесс помогает людям добиваться успеха коллективной работы. Сравните это с организацией военной операции. Бой всегда ведут отдельные люди, но результат боя определяется также и тем, насколько хорошо они организованы.

Средства и процесс — одно целое

Современный процесс разработки программного обеспечения предусматривает применение средств поддержки разработки. Сегодня невозможно представить себе

разработку программного обеспечения без использования этих средств. Процесс и средства его поддержки едины: средства поддержки интегрированы в процесс [41, 43].

Средства жестко привязаны к процессу

Процесс сильно зависит от средств поддержки. Они с успехом применяются для автоматизации повторяющихся задач, сохранения структурированности, управления большими массивами данных и проведения разработчика по конкретному пути разработки.

При слабом использовании средств поддержки процесс предполагает большой объем ручного труда, что делает его менее формализуемым. На практике большая часть формальной работы откладывается до реализации. Без средств, автоматически поддерживающих целостность проекта в течение жизненного цикла, сложно сохранять модели и реализацию в актуальном виде. Это затрудняет применение итеративной и инкрементной разработки. Или модели и реализация рано или поздно станут несовместимы, или для постоянной переработки документов с целью поддержания их актуальности потребуется масса ручной работы. В последнем случае продуктивность работы значительно снизится. Команда будет тратить все свое время на ручную проверку актуальности документации. Это очень трудно, если не невозможно, и в разработанных артефактах скорее всего появятся многочисленные дыры. И все равно такой путь потребует массы времени.

Средства поддержки создаются для полной или частичной автоматизации видов деятельности с целью повышения производительности и качества и сокращения затрачиваемого на разработку времени. Работая со средствами поддержки разработки, вы получаете другой процесс, более формальный. Вы можете описать новые виды деятельности, которые без средств поддержки не применялись. В течение всего жизненного цикла вы можете работать с большей точностью. Вы можете использовать формальные языки моделирования, например UML, для гарантии того, что каждая модель согласована как внутренне, так и с другими моделями. По одной модели вы можете генерировать части для другой (например, реализацию по проекту и наоборот).

Процесс управляет средствами

Процесс, ясно определенный или подразумеваемый, определяет функциональность средств поддержки разработки. Существование процесса — это, разумеется, единственная причина для применения каких-либо средств поддержки. Средства применяются для автоматизации как можно большей части процесса.

Возможность автоматизации процесса зависит от существования ясной картины того, в каких вариантах использования нуждается каждый из сотрудников и какие артефакты ему необходимо обрабатывать. Автоматизированный процесс обеспечивает эффективные средства для параллельной работы всех сотрудников одновременно, что позволяет организовать проверку целостности всех артефактов системы.

Средства, включаемые в автоматизированный процесс, должны быть *просты в использовании*. Чтобы их активно использовали, разработчикам средств нужно

вдумчиво разобраться в методах разработки программ. Например, как сотрудник будет решать некую задачу? Как, по его мнению, средства поддержки могут помочь ему в этой работе? Какие задачи будут повторяться и, следовательно, должны быть автоматизированы? Какие задачи редки и, вероятно, могут не автоматизироваться? Как средства поддержки могут направлять усилия сотрудника на решение сложных задач, с которыми может справиться только он, оставляя повторяющиеся задачи, которые средства поддержки сделают лучше, на откуп этим средствам? Чтобы ответить на подобные вопросы, сотрудники должны разбираться в средстве и быть в состоянии его использовать. Кроме того, чтобы окупить время, потраченное на его изучение, использование средства должно давать существенное увеличение производительности.

Для отстаивания простоты использования средств имеются особые причины. Сотрудники должны иметь возможность рассматривать различные альтернативы, и они должны быть в состоянии легко разбираться с альтернативными проектами-кандидатами. Они должны иметь возможность выбрать один из вариантов и проработать его. Если этот подход будет неверен, им надо дать возможность легко перейти к рассмотрению другого варианта. Средства поддержки должны давать сотрудникам возможность максимально использовать то, что уже сделано, чтобы работа не начиналась с нуля при переходе к новому варианту. Подведем итог. Важно, чтобы средства поддержки разработки осуществляли как автоматизацию повторяющихся действий, так и управление информацией, представленной в виде серий моделей и артефактов, и таким образом поощряли и поддерживали бы созидательную деятельность, которая является основой серьезной разработки.

Баланс между процессом и средствами его осуществления

Разработка процесса без размышлений о его автоматизации — это чистая теория. Разработка средств поддержки без знания процесса, который они должны поддерживать — бесплодное экспериментирование. Следует соблюдать баланс между процессом и средствами его поддержки.

С одной стороны, процесс управляет разработкой средств поддержки. С другой — средства направляют разработку процесса. Разработка процесса и разработка средств его поддержки должны идти параллельно. Каждый выпуск процесса должен сопровождаться выпусками средств поддержки. Этот баланс должен соблюдаться в каждом выпуске. Приближение баланса к идеалу потребует нескольких итераций, при этом последующие итерации будут направляться советами и рекомендациями пользователей, получаемыми в промежутках между итерациями.

Эти взаимоотношения процесса и средств сродни проблеме курицы и яйца. Что должно быть раньше? В недавнее время множество средств поддержки было выпущено раньше процесса. Процесс был еще не вполне разработан. В результате в типичных процессах «попал-или-промазал», к которым их пытались применять пользователи, средства поддержки не работали так хорошо, как предполагалось. Вера многих из нас в средства поддержки была поколеблена. Разработка программного обеспечения продолжала оставаться не слишком эффективным кустарным промыслом. Иначе говоря, процесс должен учиться у средств, а средства — поддерживать хорошо продуманный процесс.

Внесем полную ясность в этот момент: успешная разработка средств автоматизации процесса невозможна без параллельной разработки каркаса процесса, с которым должны работать эти средства. Это должно быть понятно всем. Если же тень сомнения все еще появляется в вашем мозгу, спросите себя, можно ли разрабатывать программы для поддержки бизнес-процессов банка, не имея представления о том, что это за процессы.

Унифицированный язык моделирования поддерживает визуальное моделирование

Мы установили важность средств поддержки разработок для осуществления целей процесса.

Рассмотрим соответствующий пример средства в контексте поддержки окружения Унифицированным процессом. Нашим примером будет средство для моделирования на основе UML.

UML — визуальный язык. Поэтому мы можем рассчитывать на наличие возможностей, характерных для программ инженерной графики, таких как редактирование по месту, форматирование, изменение масштаба, печать, цвета и автоматическое размещение. В придачу к этим возможностям в UML определены синтаксические правила, описывающие совместное использование элементов языка. Средства поддержки должны быть в состоянии обеспечить выполнение этих синтаксических правил. Эта функция лежит вне возможностей стандартных чертежных пакетов, где подобные правила отсутствуют.

К сожалению, строгое исполнение такого рода синтаксических правил делает использование средства поддержки невозможным. Так, при редактировании модели она часто бывает синтаксически некорректной, и в данном режиме средства поддержки должны допускать существование этой некорректности. Например, сообщение на диаграмме последовательности (приложение А, см. также главу 9) может быть определено до объявления в классе каких-либо операций.

UML включает в себя множество синтаксических правил, которые также следует поддерживать. Эти правила могут быть встроены в средство для поддержки моделирования в форме постоянно работающей проверки или вызываемой процедуры, которая сканирует модель, делая общую проверку на ошибки или проверяя семантическую или синтаксическую неполноту. Подведем итог. В средства моделирования должно быть включено не просто знание UML. Они должны включать возможность творческого использования языка моделирования.

При использовании UML в качестве стандартного языка, рынок получит гораздо лучшие средства поддержки разработок, чем при использовании любого другого существовавшего когда-либо языка моделирования. Эта возможность отчасти объясняется более точным определением UML. Также она обусловлена тем фактом, что UML сейчас — общепринятый промышленный стандарт. Вместо соревнования поставщиков средств поддержки, кто поддержит больше языков моделирования, игра идет на то, кто лучше поддерживает UML. Эти новые правила приносят больше пользы заказчикам и пользователям программ.

UML — это только язык моделирования. Он не определяет процесс использования UML для разработки программных систем. Средства моделирования не принуждают использовать процесс. Но если пользователи работают в соответствии с процессом, средства их в этом поддержат.

Средства поддерживают весь жизненный цикл системы

Существуют средства для поддержки всех фаз **жизненного цикла программы** (приложение В).

- **Управление требованиями.** Используются для хранения, просмотра, анализа, отслеживания и перебора различных требований проекта создания программного обеспечения. Требования могут иметь приписанный к ним статус. Средства поддержки могут позволять отслеживать требования в других артефактах жизненного цикла, таких как варианты использования или тестовые случаи (см. главу 11).
- **Визуальное моделирование.** Используются для автоматизации использования UML, то есть для визуального моделирования и построения приложений. Если эти средства будут встроены в нашу среду программирования, то мы можем гарантировать, что модель и реализация всегда будут согласованы друг с другом.
- **Средства программирования.** Используются для обеспечения разработчиков набором таких средств, как редакторы, компиляторы, отладчики, средства поиска ошибок и анализа производительности.
- **Контроль качества.** Используются для тестирования приложений и компонентов, то есть для записи и выполнения вариантов тестирования, графического интерфейса пользователя и интерфейса компонентов. В итеративном жизненном цикле регрессионное тестирование гораздо более важно, чем в традиционной разработке. Для получения максимальной производительности необходима автоматизация вариантов тестирования. Кроме того, многим приложениям необходимо тестирование загрузки и работы под большой нагрузкой. Как будет вести себя это приложение при одновременной работе с ним десяти тысяч пользователей? Вы будете знать ответ на этот вопрос до того, как к нему подключится десятитысячный пользователь.

В добавление к этим функционально-ориентированным средствам существуют другие средства, охватывающие несколько жизненных циклов. Эти средства включают в себя контроль версий, управление конфигурацией, отслеживание дефектов, документирование, управление проектом и автоматизацию процесса.

3

Процесс, направляемый вариантами использования

Задача Унифицированного процесса — предоставлять разработчикам план эффективного создания и развертывания систем, удовлетворяющих требованиям пользователя. Эффективность при этом измеряется в понятиях стоимости, качества и затраченного времени. Путь от получения требований заказчика до реализации не прост. Проблемы начинаются с определения требований заказчиков. Этот процесс предполагает, что мы способны определить требования пользователей и довести их без ошибок до всех участников проекта. После этого мы должны быть в состоянии спроектировать работающую реализацию, которая выполняла бы эти требования. И наконец, мы должны протестировать созданную систему, чтобы удостовериться в том, что требования пользователей выполнены. Сложность процесса заставляет нас описывать его в виде серии рабочих процессов, последовательно создающих работающую систему.

Как мы говорили в главе 1, Унифицированный процесс управляет вариантами использования, ориентирован на архитектуру, итеративен и инкрементен. В этой главе мы рассмотрим, каким образом Унифицированный процесс управляет вариантами использования. Эта концепция была впервые представлена в [32] и внимательно рассмотрена в [34]. Ориентированность на архитектуру будет подробно описана в главе 4, а итеративность и инкрементность — в главе 5. Разбивая обсуждение этих концепций на три главы, мы надеемся более точно и ясно передать идею разработки, управляемой вариантами использования. Преследуя ту же цель, мы не станем рассматривать в этой главе, как создавать модель развертывания, проектировать высокointегрированные системы, разрабатывать хорошие компоненты реализации (см. подраздел «Артефакт: компонент» главы 10) и проводить специальные виды тестирования. Эти темы не помогут нам лучше понять варианты использования и то, как они управляют процессом разработки, поэтому детальное изучение этих тем мы отложим до второй части книги.

На рис. 3.1 изображена последовательность рабочих процессов и моделей Унифицированного процесса. Разработчики начинают с определения требований за-

казчика в виде вариантов использования и формирования на их основе модели вариантов использования. Затем они анализируют и проектируют систему, осуществляющую эти варианты использования, строя сначала модель анализа, а затем модели проектирования и развертывания. После этого они реализуют систему, получая модель реализации, которая содержит весь код проекта в виде компонентов. В конце разработчики создают модель тестирования, которая позволяет им проверить, обеспечивает ли система функциональность, описанную в вариантах использования. Все модели связаны между собой зависимостями трассировки. Модель реализации наиболее формальна, а самая неформальная модель — это модель вариантов использования. Происходит это потому, что модель реализации создается на компьютерном языке, то есть части модели реализации после компиляции и компоновки превращаются в исполняемые файлы, тогда как в модели вариантов использования применяется в основном естественный язык.

Варианты использования были разработаны в качестве почти универсального средства для определения требований к любым программным системам, в том числе и собираемым из компонентов [40], но с их помощью можно не только определять требования. Они направляют весь процесс разработки. Варианты использования вносят значительный вклад в поиск и описание классов, подсистем и интерфейсов (приложение А), а также в поиск и описание вариантов тестирования, планирование итераций разработки и системной интеграции (см. главу 11).

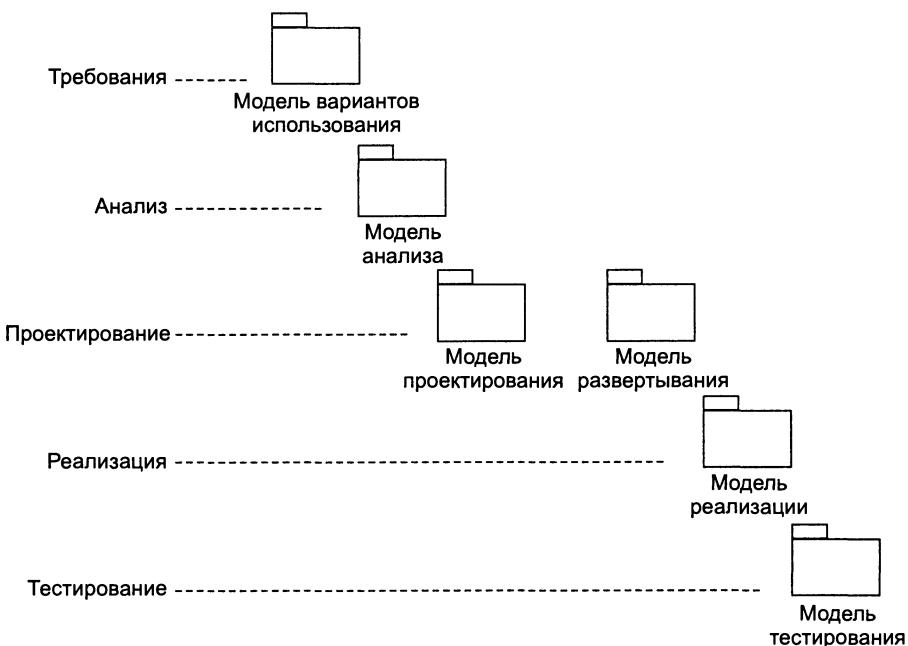


Рис. 3.1. Унифицированный процесс представляет собой последовательность рабочих процессов от определения требований до тестирования (слева, сверху вниз).

В ходе рабочих процессов разрабатываются модели,
от модели вариантов использования
до модели тестирования

На каждой итерации они направляют выполнение всех рабочих процессов, от определения требований через анализ, проектирование и реализацию до тестирования. Таким образом они связывают различные рабочие процессы между собой.

Введение в разработку, управляемую вариантами использования

Определение требований преследует две цели: определить существенные требования (см. главу 6) и представить их в форме, удобной для пользователей, заказчиков и разработчиков. Под «существенными требованиями» мы понимаем те требования, реализация которых принесет пользователям ощущимый и значимый для них результат. Под «представлением в форме, удобной для пользователей, заказчиков и разработчиков» мы понимаем главным образом то, что итоговое описание требований должны понимать пользователи и заказчики. Это один из главных результатов рабочего процесса определения требований.

Обычно у системы имеется несколько категорий пользователей. Каждая категория пользователей представляется в виде актанта. Актанты используют систему, участвуя в вариантах использования. Вариант использования — это последовательность действий, которые система предпринимает для получения актантом ценного для него результата. Полный набор актантов и вариантов использования системы образует модель вариантов использования [35, 36].

В ходе анализа и проектирования модель вариантов использования преобразуется в модель анализа, а затем в модель проектирования. Если говорить коротко, и модель анализа, и модель проектирования — это структуры, построенные на основе **классификаторов** (приложение А) и набора реализаций вариантов использования (см. главы 8 и 9), которые определяют, как эти структуры реализуют варианты использования. Классификаторы в основном представляют собой «классоподобные» сущности. Так, они содержат **атрибуты и операции** (приложение А), их можно описывать с помощью **диаграмм состояний** (приложение А), некоторые из них допускают создание экземпляров, они могут участвовать в кооперациях и т. п. В UML существует множество типов классификаторов. Примерами классификаторов для этих моделей могут быть подсистемы, классы и интерфейсы. Актанты, варианты использования, компоненты и **узлы** (приложение А, см. подраздел «Артефакт: модель развертывания» главы 9) также являются классификаторами.

Модель анализа — это детальное описание требований и предстоящих работ. Она рассматривается и как отдельная модель, и как первый шаг в создании модели проектирования. Разработчики используют ее для того, чтобы лучше понять варианты использования, описанные в рабочем процессе определения требований путем их уточнения при помощи коопераций между концептуальными классификаторами (в противоположность классификаторам проектирования, создаваемым позднее). Модель анализа также используется для построения гибкой и устойчивой системы (включая архитектуру), с как можно большим употреблением повторно используемых компонентов. Модель анализа отличается от модели проектирования тем, что первая — это концептуальная модель, а вторая — чертеж реализации. Модель анализа нередко существует лишь в течение нескольких первых итераций. Однако иног-

да, особенно в случае больших и сложных систем, аналитическая модель сопровождает систему в течение всего срока ее существования. В этом случае между соответствующими реализациями вариантов использования в модели анализа и модели проектирования существует прямая связь (через зависимость трассировки). Каждый элемент модели анализа трассирует реализующий его элемент из модели проектирования. (Модель анализа, ее свойства и связь с моделью проектирования подробно рассматриваются в подразделах «Введение в анализ», «Кратко об анализе» и «Роль анализа в жизненном цикле программы» главы 8.)

Модель проектирования имеет следующие свойства.

- Модель проектирования имеет иерархическую структуру. Она также содержит отношения, наложенные поверх иерархии. Отношения – ассоциации, обобщения и зависимости (приложение А) – часто используются в UML.
- Реализации вариантов использования в модели проектирования – это стереотипы кооперации. Кооперация описывает, как классификаторы используются в каком-либо виде деятельности, например реализаций варианта использования.
- Модель проектирования является чертежом реализации. Существует прямое отображение подсистем модели проектирования в элементы модели реализации.

Разработчики создают модель анализа, используя модель вариантов использования в качестве исходных данных [37]. Каждый вариант использования из модели вариантов использования превращается в реализацию варианта использования модели анализа. Дуализм вариантов использования/реализаций вариантов использования позволяет соблюсти полное подобие между определением требований и анализом. Прорабатывая варианты использования один за другим, разработчики идентифицируют классы, участвующие в реализации вариантов использования. Вариант использования *Получить наличные*, например, превращается в классы анализа *Получение*, *Банковский счет*, *Устройство выдачи* и еще несколько классов, которые мы пока идентифицировать не будем. Разработчики выделяют ответственности конкретных классов в **ответственности** (приложение А), определенных в вариантах использования. В нашем примере класс *Банковский счет* будет нести ответственность за выполнение операции «*Списать деньги со счета*». Таким образом, мы можем убедиться в том, что получили набор классов, совместно реализующих варианты использования и действительно необходимых.

Затем разработчики проектируют классы и реализации вариантов использования для того, чтобы лучше понять, какие продукты и технологии следует использовать для построения системы (брокеры объектных запросов, библиотеки для создания графического интерфейса пользователя, СУБД). Классы проектирования собираются в подсистемы, и между этими подсистемами определяются интерфейсы. Разработчики также создают модель развертывания, которая определяет физическое распределение частей системы по узлам – отдельным компьютерам сети – и проверяют, могут ли варианты использования быть реализованы в виде компонентов, выполняемых на этих узлах.

Далее, разработчики реализуют спроектированные классы в виде набора файлов с исходным текстом компонентов, из которых, если их скомпилировать и скомпоновать, можно получить исполняемый код – DLL, JavaBeans или ActiveX. Ва-

рианты использования помогают разработчикам определить порядок создания компонентов и включения их в систему.

И наконец, в ходе рабочего процесса тестирования тестеры проверяют, действительно ли система выполняет функции, определенные вариантами использования, и удовлетворяет требованиям к системе. Модель тестирования содержит тестовые примеры (и еще кое-что, подробнее об этом мы поговорим в главе 11). Тестовый пример определяет набор исходных данных, условий выполнения и результатов. Множество тестовых примеров могут быть порождены прямо из вариантов использования, таким образом, мы можем говорить об отношении трассировки между тестовыми примерами и соответствующими вариантами использования. Это означает, что тестеры будут проверять, делает ли система то, чего хотят от нее пользователи, то есть выполняет ли она варианты использования. Каждый, тестировавший в прошлом программное обеспечение, фактически тестировал варианты использования, — хотя тогда это и называлось иначе [64]. А вот что в Унифицированном процессе действительно изменилось, так это то, что теперь тестирование можно планировать гораздо раньше. Сразу же после определения вариантов использования можно начинать создание тестов вариантов использования (тесты для «черного ящика») и задавать порядок их выполнения, включения в систему и тестирования. Позднее, после реализации вариантов использования в ходе проектирования, можно детализировать тесты вариантов использования (тесты для «белого ящика»). Каждый способ выполнения варианта использования, а значит, каждый путь реализации варианта использования, является кандидатом на тестовый пример.

Пока мы описывали Унифицированный процесс в виде последовательности шагов, очень похожей на старый водопадный подход. Однако это делалось только для сохранения простоты рассуждений. В главе 5 мы увидим, как по-разному можно пройти эти шаги, используя итеративность и инкрементальность процесса разработки. На самом деле, все, что мы рассмотрели здесь — это одна итерация. Полный процесс разработки будет состоять из серии таких итераций, каждая из которых (а особенно первая) состоит из одного выполнения рабочих процессов определения требований, анализа, проектирования, реализации и тестирования.

Перед тем как начать детальное изучение различных рабочих процессов, рассмотрим положительные стороны применения вариантов использования.

Зачем нужны варианты использования?

Существует несколько объяснений удобства, популярности и повсеместного применения вариантов использования [40]. Две основные причины таковы:

- Варианты использования дают систематический и интуитивно понятный способ определения функциональных требований (приложение Б, см. также главы 6 и 7), ориентированный на получение необходимых пользователю результатов.
- Варианты использования управляют всем процессом разработки. Все основные виды деятельности — анализ, проектирование и тестирование — выполняются на основе определенных вариантов использования. Проектирование и тестирование к тому же планируются и координируются в понятиях вариантов использования. Это особенно хорошо видно после выполнения первых итераций, когда архитектура системы стабилизируется.

Определение требований, приносящих ощутимый и измеримый результат, важный для заказчика

По мнению Карла Вигера, «перспективы, открываемые вариантами использования, облегчают достижение основной цели разработки программ: создание продуктов, позволяющих заказчикам делать их работу» [20]. Существует несколько причин, благодаря которым дело обстоит именно так.

Во-первых, построение вариантов использования помогает создавать программы, соответствующие целям пользователей. Варианты использования — это функции системы, обеспечивающие получение пользователями ценного для них результата. Определяя поле зрения каждого типа пользователей, мы можем определить необходимые им варианты использования. С другой стороны, если мы не станем продумывать варианты использования, в которых участвуют отдельные пользователи, а начнем разработку с обдумывания набора функций системы, нам будет сложно понять, действительно ли эти функции так важны, и вообще годятся ли они в данном случае. Кому они должны помочь? Какие потребности бизнеса они удовлетворяют? Насколько значимый результат они должны принести бизнесу?

Лучшими вариантами использования считаются те, которые приносят наиболее ценные и значимые результаты для того бизнеса, в котором используется система. Варианты использования, приносящие результаты с отрицательной ценностью или заставляющие пользователя делать то, чего он делать не способен, не считаются вариантами использования. Мы будем называть их «вариантами запрещенного использования», поскольку они описывают те способы использования системы, которые должны быть предотвращены. Примером такого неправильного варианта использования могло бы быть разрешение *Клиенту банка* переводить деньги с чужих счетов на свой. Варианты использования, приносящие малоценные или не имеющие ценности результаты, используются редко, это скорее «варианты неиспользования».

Как мы отмечали в главе 1, множество людей приходило к выводу, что просто задаваться вопросом, чего они ждут от системы, неправильно. Вместо верного ответа они получали список функций системы, который на первый взгляд выглядел достаточно разумным, но при пристальном рассмотрении мог оказаться никак не связанным с истинными потребностями пользователей. Стратегия вариантов использования предлагает нам перефразировать этот вопрос, добавляя в его конец три слова — что должна делать система для каждого пользователя? Может показаться, что разница невелика, но результат получается совершенно другой. Это добавление помогает нам постоянно концентрироваться на понимании того, как система должна поддерживать работу своих пользователей. Это направляет нас на поиск функций, в которых нуждается каждый пользователь. Это также помогает нам воздержаться от предложения дополнительных функций, не нужных никому из пользователей.

Кроме того, варианты использования интуитивно понятны. Пользователи и заказчики не должны изучать сложную нотацию. Почти все время можно использовать английский (то есть естественный) язык¹. Это делает легким как чтение описаний вариантов использования, так и внесение в них изменений.

¹ Или русский. — Примеч. пер.

Определение вариантов использования проводится при участии пользователей, заказчиков и разработчиков. Пользователи и заказчики выступают в качестве определяющих требования экспертов. Роль разработчиков в том, чтобы облегчить обсуждение и помочь пользователям и заказчикам изложить свои пожелания.

Модель вариантов использования применяется для достижения соглашения с пользователями и заказчиками о функциях будущей системы. Мы можем считать модель вариантов использования полной спецификацией всех возможных путей использования системы (вариантов использования). Эта спецификация может войти и в договор, заключаемый с заказчиком. Модель вариантов использования помогает нам определить границы системы, описывая все, что она должна делать для пользователей. Интересный подход к структурированию вариантов использования содержится в [15]. А [57] содержит удачное общее введение в эту тему.

Управление процессом

Как мы говорили, управляемость посредством вариантов использования означает, что процесс разработки осуществляется в виде серий рабочих процессов, которые инициируются вариантами использования. Варианты использования помогают разработчикам в поиске классов. Классы выделяются из описаний вариантов использования. Разработчик просматривает их в поисках классов, пригодных для реализаций вариантов использования¹. Варианты использования также помогают нам разрабатывать пользовательские интерфейсы, которые облегчают выполнение работы пользователями. Затем реализации вариантов использования тестируются, чтобы проверить, что **экземпляры** (приложение A) классов правильно выполняют варианты использования [64].

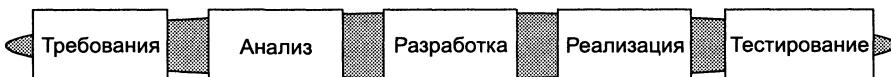


Рис. 3.2. Варианты использования связывают основные рабочие процессы.
Заштрихованный фоновый эллипс символизирует связывание
рабочих процессов вариантами использования

Варианты использования не просто инициируют процесс разработки, но и связывают его в единое целое, как показано на рис. 3.2.

Мы также должны быть уверены, что правильно определили варианты использования, то есть что пользователи получат те варианты использования, которые им действительно нужны. Лучший способ решить эту проблему, разумеется, это хорошо поработать, определяя требования. Но этого часто недостаточно. Работающая система требует от нас последующего сравнения вариантов использования с истинными задачами пользователей.

Варианты использования помогают менеджеру проекта планировать, распределять и отслеживать множество задач, которые выполняют разработчики. Менеджер проекта должен определить группу задач для каждого варианта использования. Описание каждого варианта использования — это задача, проектирование каждого варианта использования — это задача и тестирование каждого варианта

¹ Это упрощение. В действительности каждый вариант использования может использовать ранее разработанные классы (подсистемы), и нам придется подстраивать вариант использования под этот повторно используемый блок. Это рассматривается подробнее в подразделе «Варианты использования и архитектура» главы 4.

использования — это тоже задача. Затем менеджер проекта должен оценить силы и время, необходимые для выполнения этих задач. Задачи, основанные на вариантах использования, помогают менеджеру оценить размер проекта и требуемые для него ресурсы. Эти задачи затем могут быть поручены отдельным разработчикам, которые назначаются ответственными за них. Менеджер проекта может назначить одного разработчика ответственным за определение пяти вариантов использования при определении требований, другого — ответственным за проектирование трех вариантов использования, а третьего — за определение трех тестовых примеров для двух вариантов использования.

Варианты использования — это важный механизм обеспечения трассировки моделей. Вариант использования из рабочего процесса определения требований трассирует свои реализации из анализа и проектирования во все классы, участвующие в его реализации, компоненты (хотя и не напрямую) и, наконец, тестовые примеры для его проверки. Эта трассировка — важный аспект управления проектом. Когда вариант использования изменяется, соответствующие реализации, классы, компоненты и тестовые примеры должны быть просмотрены и приведены в соответствие с ним. Аналогично при изменении файла компонента (текста программы) соответствующие классы, варианты использования и тестовые примеры также должны быть просмотрены и т. д. (см. [15]). Трассировка между вариантами использования и другими элементами модели упрощает сохранение целостности системы и поддержание системы в актуальном состоянии при изменении требований.

Задание архитектуры

Варианты использования помогают вести итеративную разработку. Создание **приращения** (приложение B) в ходе каждой итерации, кроме, может быть, самой первой, направляется вариантами использования при проходе по всем рабочим процессам, от определения требований до проектирования и тестирования. Каждое приращение разработки, таким образом, возникает в результате работы по реализации набора вариантов использования. Другими словами, в каждой итерации определяется и реализовывается некоторое количество вариантов использования.

Варианты использования также помогают нам подобрать архитектуру. Мы выбираем набор вариантов использования — варианты использования, оказывающие влияние на архитектуру, и реализуем их в ходе первых итераций. Тем самым мы обеспечиваем систему стабильной архитектурой, которую будем использовать на протяжении множества последующих циклов. Мы вернемся к этому вопросу в главе 4.

Варианты использования служат отправной точкой для написания руководства пользователя. Поскольку каждый вариант использования определяет один из способов использования системы, они являются идеальным началом для описания взаимодействия пользователя с системой.

Оценивая, как часто выполняются различные варианты использования, можно определить, какой из них требует особой эффективности. Эта оценка может быть использована для определения требуемой производительности процессора используемого компьютера или оптимизации структуры базы данных под отдельные операции. Также подобные оценки могут быть применены для повышения удобства

работы с системой, становится возможным выделить наиболее важные варианты использования для того, чтобы разрабатывать их пользовательские интерфейсы особенно тщательно.

Определение вариантов использования

Теперь перейдем к обзору последовательности рабочих процессов. Как было сказано ранее, мы сосредоточимся на аспекте управления вариантами использования. В следующем пункте мы сконцентрируем свое внимание на определении функциональных требований в виде вариантов использования. Определение других типов требований, происходящее в то же самое время, будет интересовать нас значительно меньше.

В ходе рабочего процесса определения требований мы идентифицируем потребности пользователей и клиентов в виде требований. Функциональные требования описываются вариантами использования, входящими в модель вариантов использования. Прочие же требования могут присоединяться к тем вариантам использования, к которым они относятся, сохраняться в отдельном списке или фиксироваться каким-либо иным образом.

Модель вариантов использования отражает функциональные требования

Модель вариантов использования помогает клиентам, пользователям и разработчикам договориться о способах использования системы. Большинство систем имеет множество категорий пользователей. Каждая категория пользователей представлена отдельным *актантом*. Актанты используют систему, взаимодействуя с вариантами использования. Полный набор актантов и вариантов использования образует модель вариантов использования [34, 35]. *Диаграмма использования* (приложение А; см. также подраздел «Деятельность: нахождение актантов и вариантов использования» главы 7) описывает часть модели вариантов использования, показывая набор вариантов использования с актантами и с ассоциациями между каждой взаимодействующей парой актант — вариант использования.

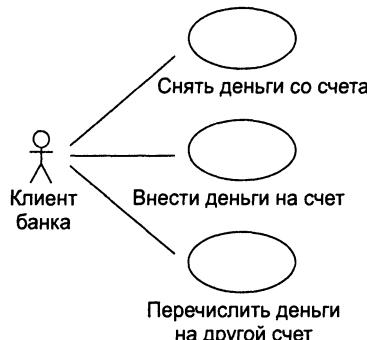


Рис. 3.3. Пример диаграммы использования с актантом и тремя вариантами использования

Пример. *Модель вариантов использования для банкомата (ATM).* Актант *Клиент банка* использует банкомат (ATM) для того, чтобы класть деньги на счет или снимать их со счета, а также производить перечисление денег со счета на счет. Это поведение представляется в виде трех вариантов использования, изображенных на рис. 3.3. Изображенные на этом рисунке ассоциации актанта *Клиент банка* с вариантами использования показывают, что он участвует в этих вариантах использования.

Актанты — это среда, в которой существует система

Актанты — это не обязательно люди. Актантами могут быть другие системы или внешнее оборудование компьютера, взаимодействующее с системой. Каждый актант, взаимодействуя с системой, исполняет согласованный набор ролей. Физический пользователь может играть роль одного или нескольких актандов, выполняя их функции по взаимодействию с системой. Поведение одного и того же актента может наблюдаться у нескольких физических пользователей. Например, поведение актента *Клиент банка* может быть присуще тысячам людей.

Актанты в ходе выполнения вариантов использования обмениваются информацией с системой, посыпая ей **сообщения** (приложение А) и получая ответные сообщения. Определяя, что делают актанды, а что — варианты использования, мы получаем четкое разграничение обязанностей между актантами и системой. Это разграничение помогает нам лучше определить сферу деятельности системы.

Мы можем найти и определить всех актандов, рассматривая, как будут работать с системой ее пользователи и какие внешние системы должны взаимодействовать с нашей системой. Каждая категория пользователей или внешних систем, взаимодействующих с нашей, будет представлена одним актантом.

Варианты использования определяют систему

Варианты использования создаются для того, чтобы пользователи, работая с системой, могли удовлетворить свои потребности. Модель вариантов использования определяет все функциональные требования системы. Мы даем следующее определение варианта использования:

Вариант использования определяет последовательность действий (включая ее варианты), которые может выполнить система, приносящих ощутимый и измеримый результат, значимый для некоторого актента.

Мы ищем варианты использования, рассматривая, как пользователи хотят использовать систему для выполнения своей работы (способ структурирования вариантов использования на основе целей см. в [15]). Каждый метод использования системы, приносящий пользователю ощутимый результат — кандидат в варианты использования. Эти кандидаты в дальнейшем превратятся в настоящие варианты использования, будут разбиты на более частные варианты использования или включены в глобальные варианты использования. Когда модель варианта использования определяет все функциональные требования правильно и таким образом, что их понимают клиенты, пользователи и разработчики, ее можно считать практи-

чески законченной. Последовательность действий, происходящих при выполнении варианта использования (то есть экземпляр варианта использования) — это определенный путь осуществления варианта использования. Возможно множество путей осуществления варианта использования, и многие из них очень похожи — ведь все они являются вариантами выполнения последовательности действий, описанных в варианте использования. Чтобы сделать модель вариантов использования понятной, мы должны собрать описания похожих вариантов выполнения в один вариант использования. Когда мы говорим, что идентифицируем и описываем вариант использования, мы на самом деле подразумеваем, что мы идентифицируем и описываем различные варианты выполнения, которые считаем полезным рассматривать как один вариант использования.

Пример. Вариант использования *Снять деньги со счета*. Последовательность действий пути осуществления варианта использования (очень упрощенно).

1. Кассир банка идентифицирует клиента.
2. Кассир банка выбирает счет, с которого будут сняты деньги, и определяет снимаемую сумму.
3. Система снимает соответствующую сумму со счета и выдает деньги.

Варианты использования также используются для хранения нефункциональных требований (приложение B; см. главу 6), определенных для этого варианта использования, таких как производительность, точность и безопасность. Например, к варианту использования *Снять деньги со счета* может быть приложено следующее требование: время отклика для *Клиента банка*, измеряемое от ввода снимаемой суммы до выдачи купюр, не должно превышать 30 секунд в 95% всех операций.

Подведем итог. К вариантам использования можно привязать все функциональные требования. К вариантам использования можно приложить многие нефункциональные требования. Фактически модель вариантов использования — это аппарат перевода требований в легкоуправляемую форму. Клиенты и пользователи, понимающие это, могут с помощью вариантов использования сообщать о своих потребностях разработчикам последовательным и лишенным избыточности способом. Разработчики могут разделить между собой работу по определению требований и использовать ее результаты (прежде всего варианты использования) в качестве исходных данных для анализа, проектирования, реализации и тестирования системы.

Анализ, проектирование и разработка при реализации варианта использования

В ходе анализа и проектирования мы преобразуем модель вариантов использования через модель анализа в модель проектирования, то есть в структуру, состоящую из классификаторов и реализаций вариантов использования. Цель состоит в том, чтобы реализовать варианты использования так, чтобы система имела в результате устраивающую нас эффективность и перспективы развития, не выходя при этом за рамки бюджета.

В этом пункте мы рассмотрим, как пройти через анализ к разработке проекта реализации вариантов использования. В главах 4 и 5 мы увидим, как опора на архитектуру и итеративная и инкрементная разработка помогают нам разрабатывать систему, которая может пополняться новыми требованиями, укладываясь при этом в рамки бюджетных ограничений.

Создание по вариантам использования аналитической модели

Модель анализа последовательно разрастается по мере анализа все новых и новых вариантов использования. На каждой итерации мы выбираем варианты использования, которые будем включать в модель анализа. Мы строим систему в виде структуры, состоящей из классификаторов (классов анализа) и отношений между ними. Мы также описываем кооперации, реализующие варианты использования, то есть реализации вариантов использования. На следующей итерации мы выбираем для проработки следующие варианты использования и добавляем их к результатам предыдущей итерации. В подразделах «Итеративный подход — управляемый рисками» главы 5 и «Расстановка приоритетов вариантов использования» главы 12 мы рассматриваем, как на первых итерациях идентифицировать и выбрать «важнейшие» варианты использования, чтобы построить стабильную архитектуру на ранних стадиях жизненного цикла системы.

Работа обычно выполняется так: сначала идентифицируются и описываются варианты использования текущей итерации, затем в соответствии с описанием для каждого варианта использования (как показано в подразделе «Варианты использования определяют систему» данной главы) предлагаются классификаторы и ассоциации, необходимые для реализации этого варианта использования. Мы проделываем эту работу для всех вариантов использования текущей итерации. В зависимости от того, в какой точке жизненного цикла мы находимся и с какой итерацией работаем, архитектура может уже быть определена, тогда она поможет нам обнаруживать новые классификаторы и повторно использовать существующие (см. подраздел «Варианты использования и архитектура» главы 4). Каждый классификатор играет в реализации варианта использования одну или несколько ролей. Каждая роль классификатора определяет обязанности, атрибуты и другие характеристики, которыми классификатор будет обладать в реализации варианта использования. Мы можем считать эти роли зародышами классификаторов. В UML роль является классификатором сама по себе. Например, мы можем думать о роли класса как о представлении этого класса. Таким образом, она включает содержимое этого класса, то есть обязанности, атрибуты и другие характеристики — но только те, которые имеют отношение к его роли в реализации варианта использования. Другой способ описать роль класса состоит в том, чтобы рассматривать ее как то, что остается от класса после наложения на него фильтра, отсекающего все то, что относится к другим ролям класса, за исключением совместно выполняемых обязанностей. Такой подход к роли кратко описан в этой главе. Для упрощения понимания материала во второй части этой книги, где классификаторы обсуждаются подробнее, этот подход не используется.

Пример. Реализация варианта использования в модели анализа. На рис. 3.4 мы показываем, как с помощью кооперации реализуется вариант использования *Снять*

деньги со счета (то есть реализацию варианта использования) с зависимостью «трассировка» (трассировка — стереотип зависимости, которая обозначена кавычками, « и ») между ними. В реализации варианта использования участвуют и играют свои роли четыре класса. Как можно понять из рисунка, нотацией реализации варианта использования или сотрудничества является эллипс, контур которого нарисован пунктиром.

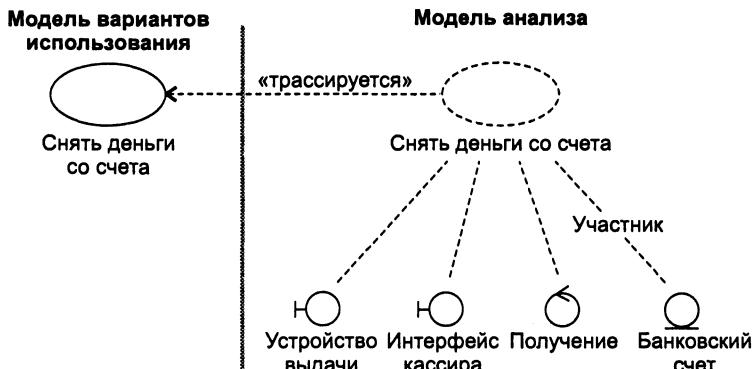


Рис. 3.4. Классы анализа, участвующие в реализации варианта использования
Снять деньги со счета. Устройство выдачи и Интерфейс кассира —
границевые классы, Получение — управляющий класс
и Счет — класс сущности

Обычно мы начинаем работу с изучения нескольких вариантов использования, создаем для них реализации вариантов использования и выделяем роли классификаторов. Затем мы делаем то же самое для других вариантов использования и выделяем новые роли классификаторов. Некоторые из этих новых ролей могут оказаться новыми или изменившимися ролями уже обнаруженных классификаторов, в то время как другие роли потребуют новых классификаторов. Тогда мы снова просматриваем первую порцию вариантов использования и так, чередуя варианты использования, постепенно строим устойчивую модель анализа. В результате каждый классификатор может участвовать и играть соответствующие роли в нескольких реализациях вариантов использования.

СТЕРЕОТИПЫ АНАЛИЗА

В модели анализа используется три различных стереотипа классов — граничный класс, управляющий класс и класс сущности. *Устройство выдачи* и *Интерфейс кассира* — это граничные классы; эти классы в основном используются для моделирования взаимодействия между системой и ее актантами (то есть пользователями и внешними системами). *Получение* — управляющий класс; эти классы в основном используются для представления координаты, последовательности, взаимодействия и управления другими объектами — и часто применяются для инкапсуляции управления, относящегося к определенному варианту использования (в данном случае — варианту использования *Снять деньги со счета*). *Банковский счет* — класс сущности; эти классы в основном используются для моделирования информации длительного хранения, часто постоянной. Таким образом, каждый из стереотипов классов поддерживает различные типы поведения (или, если хотите, функциональные

возможности). В результате стереотипы помогают нам построить надежную систему. Мы подробно рассмотрим эту тему в главе 8. Этот подход также помогает находить кандидатов на многократное использование и места, в которых возможно применение готовых классов, поскольку классы сущностей часто бывают одинаковыми для многих вариантов использования, а следовательно, и для многих различных приложений. Подразделение классов анализа на эти три стереотипа более подробно описано в [34]. Этот прием был проверен много летней практикой, широко используется сейчас и включен в UML [57].

Пример. Класс, участвующий в нескольких реализациях варианта использования модели анализа. В левой части рис. 3.5 – набор вариантов использования для банкомата (тот же самый, что и на рис. 3.3), в правой части – соответствующая структура системы, в данном случае – классы анализа, реализующие варианты использования. Структура системы моделируется **диаграммой классов** (приложение А). Диаграммы классов обычно используются для демонстрации классов и их отношений, но их также можно использовать для демонстрации подсистем и интерфейсов (мы рассмотрим этот вопрос при обсуждении проектирования в подразделе «Создание модели проектирования из аналитической модели» данной главы). Для того чтобы было проще понять, в какой реализации варианта использования участвует данный класс, мы использовали различную штриховку. Диаграмма показывает, как каждый из вариантов использования (слева) реализован в структуре классов анализа (справа). Так, например, классы *Интерфейс кассира*, *Получение*, *Банковский счет* и *Устройство выдачи* участвуют в реализации варианта использования *Снять деньги со счета*. Классы *Интерфейс кассира* и *Банковский счет* участвуют и исполняют роли во всех трех реализациях вариантов использования. Остальные классы участвуют в одной реализации варианта использования каждый, они исполняют только одну роль.

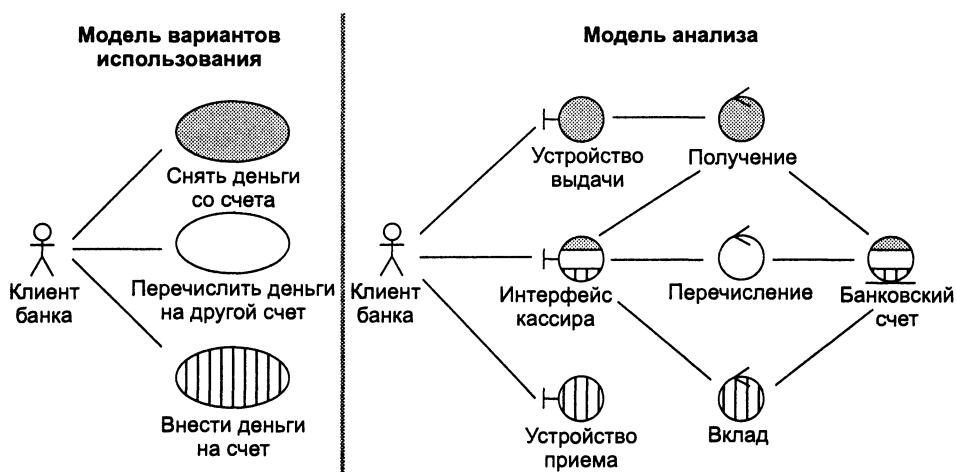


Рис. 3.5. Реализация каждого из вариантов использования (слева) в структуре классов анализа (справа)

Диаграмма классов банкомата (рис. 3.5) была получена в результате изучения описаний трех вариантов использования и поиска способов реализации каждого из них. Мы можем сделать это примерно так.

- Реализации трех вариантов использования *Снять деньги со счета*, *Перечислить деньги на другой счет* и *Внести деньги на счет* используют граничный класс *Интерфейс кассира* и класс сущности *Банковский счет*. Выполнение каждого варианта использования начинается с **объекта** (приложение А) *Интерфейс кассира*. Затем выполнение переходит к управляющему объекту, который координирует выполнение варианта использования. Класс этого объекта унаследован для каждого варианта использования. Класс *Получение*, таким образом, участвует в варианте использования *Снять деньги со счета* и т. д. Объект *Получение* предлагает *Устройству выдачи* выдать деньги, а объекту *Банковский счет* — уменьшить баланс счета.
- В реализации варианта использования *Перечислить деньги на другой счет* участвуют два объекта *Банковский счет*. Они оба нуждаются в запросе на изменение баланса счета от объекта *Перечисление*.
- Объект *Вклад* принимает деньги через *Устройство приема* и предлагает объекту *Банковский счет* увеличить баланс счета.

Пока целью нашей работы было определение структуры системы для текущей итерации. Мы идентифицировали обязанности участвующих классификаторов и определили отношения между классификаторами. Однако мы не старались подробно идентифицировать взаимодействия, необходимые для понимания вариантов использования. Мы нашли структуру и теперь должны наложить на нее шаблоны взаимодействия, требующиеся для каждой реализации варианта использования.

Как мы уже говорили, каждый вариант использования преобразуется в реализацию варианта использования и каждая реализация варианта использования содержит набор участвующих в ней классификаторов, исполняющих различные роли. Понимание шаблонов взаимодействия означает, что мы определяем, как выполняется реализация варианта использования или создается экземпляр. Например, что происходит, когда некий *Клиент банка* снимает деньги со счета, то есть когда он выполняет вариант использования *Снять деньги со счета*. Мы знаем, что в реализации варианта использования будут участвовать классы *Интерфейс кассира*, *Получение*, *Банковский счет* и *Устройство выдачи*. Мы также знаем, какие у них обязанности. Однако нам не известно, как они или, что более правильно, как объекты этих классов взаимодействуют между собой в реализации варианта использования. Мы должны это выяснить. Сначала мы используем диаграммы кооперации (приложение А) для того, чтобы смоделировать взаимодействия между объектами анализа (для моделирования взаимодействий в UML также имеются диаграммы последовательностей, их мы рассмотрим в подразделе «Создание модели проектирования из аналитической модели» данной главы, когда будем говорить о проектировании). **Диаграмма кооперации** напоминает диаграмму классов, но на ней изображаются не классы и ассоциации, а экземпляры и связи (приложение А). Она показывает, как объекты последовательно или параллельно взаимодействуют с нумерацией сообщений, которыми они обмениваются.

Пример. Использование диаграммы кооперации для описания реализаций вариантов использования в модели анализа. На рис. 3.6 мы используем диаграмму кооперации для описания того, как группой объектов анализа выполняется реализация варианта использования *Снять деньги со счета*.

Диаграмма показывает, как по мере выполнения варианта использования фокус перемещается от объекта к объекту и как между объектами пересылаются сообщения. Сообщение, посланное от одного объекта другому, переключает фокус на объект-приемник и побуждает его выполнить одну из обязанностей, порученных этому классу.

Название сообщения описывает предполагаемый смысл взаимодействия вызывающего объекта с вызываемым. Позже, в ходе проекта, эти сообщения будут преобразованы в одну или более операции, содержащиеся в соответствующих классах проектирования. Мы рассмотрим это в подразделе «Создание модели проектирования из аналитической модели» данной главы.

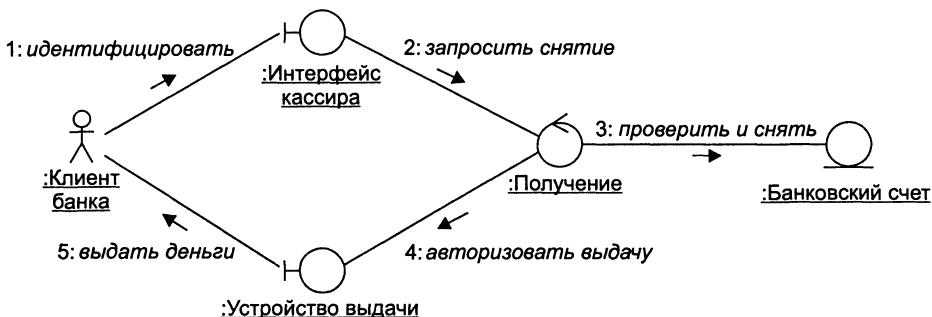


Рис. 3.6. Диаграмма кооперации для реализации варианта использования *Снять деньги со счета* модели анализа

Разработчики могут также добавить к диаграмме кооперации текст, поясняющий взаимодействие объектов при выполнении последовательности событий варианта использования. Существует также множество других способов описания реализации, например использование структурированного текста или псевдокода.

Пример. Описание последовательности событий реализации варианта использования. Здесь мы опишем реализацию варианта использования *Снять деньги со счета* в понятиях взаимодействующих объектов и актантов, как показано на рис. 3.6.

Клиент банка, желая снять деньги со счета, активирует объект *Интерфейс кассира*. Клиент банка идентифицирует себя и указывает, сколько денег снять и с какого счета (1). Интерфейс кассира проверяет идентификацию Клиента банка и предлагает объекту *Получение* выполнить операцию (2).

Если идентификация Клиента банка признана верной, объект *Получение* предлагает объекту *Банковский счет* подтвердить, что Клиент банка имеет право получить запрошенную сумму. Объект *Получение* делает это, предлагая объекту *Банковский счет* подтвердить правильность запроса и, если он верен, списать указанную сумму с баланса (3).

После этого объект *Получение* разрешает *Устройству выдачи* выдать сумму, запрошенную *Клиентом банка* (4). *Клиент банка* при этом получает запрошенную сумму денег (5).

Отметим, что этот простой пример описывает только одну последовательность действий из возможных при реализации варианта использования, а именно вариант без каких-либо осложнений. Осложнением мы называем, например, вариант, когда баланс *Банковского счета* меньше запрошенной к выдаче суммы.

Итак, мы проанализировали все варианты использования и идентифицировали роли классов, принимающих участие в каждой из реализаций вариантов использования. Переходим теперь к анализу классов.

Каждый класс должен играть все свои роли в кооперациях

Обязанности класса — это просто сумма всех ролей, которые он исполняет во всех реализациях вариантов использования. Сложив все роли и удалив перекрывающиеся части ролей, мы получим список всех обязанностей и атрибутов класса.

За определение ролей класса отвечают разработчики, выполняющие анализ и реализацию вариантов использования. Разработчик, отвечающий за класс, собирает все роли класса в полный набор обязанностей класса, а затем преобразует его в согласованный набор обязанностей.

Разработчики, отвечающие за анализ вариантов использования, должны гарантировать, что классы реализуют варианты использования с соответствующим качеством. Если класс изменен, разработчик класса должен убедиться, что класс по-прежнему способен выполнять свои роли в реализациях вариантов использования. Если изменилась роль в реализации варианта использования, разработчик варианта использования должен сообщить об изменении разработчику класса. Роли, таким образом, помогают разработчикам классов и разработчикам вариантов использования поддерживать целостность анализа.

Создание модели проектирования из аналитической модели

Модель проектирования создается с использованием модели анализа в качестве исходных данных. Она адаптируется к выбранной среде выполнения, например к брокеру объектных запросов, библиотекам для построения пользовательского интерфейса или СУБД. Она может также быть адаптирована для повторного использования **унаследованных систем** (приложение В) или других заготовок, разработанных для проекта. Таким образом, принимая во внимание, что модель анализа является первым шагом модели проектирования, модель проектирования будет чертежом для реализации.

Подобно модели анализа, модель проектирования также определяет классификаторы (классы, подсистемы и интерфейсы), отношения между ними и кооперации, которые реализуют варианты использования. Однако элементы, определенные в модели проектирования, являются «проектными аналогами» более концептуальных элементов, определенных в модели анализа, в том смысле, что новые эле-

менты (проекта) адаптированы к среде выполнения, а предыдущие элементы (анализа) — нет. Другими словами, модель проекта более «физическая» по природе, а модель анализа более «концептуальная».

Реализация вариантов использования модели анализа связана трассировкой с реализацией вариантов использования модели проектирования.

Пример. Реализация вариантов использования в моделях анализа и проектирования. На рис. 3.7 показано, как вариант использования *Снять деньги со счета* реализуется в моделях анализа и проектирования в виде реализации варианта использования.

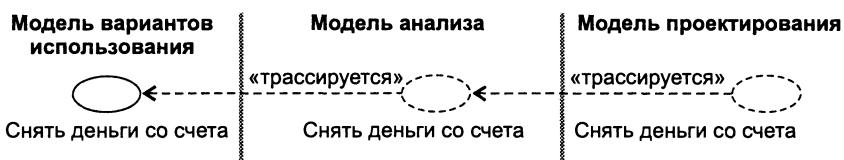


Рис. 3.7. Реализации варианта использования в разных моделях

Реализация варианта использования в различных моделях преследует различные цели. Вернемся к подразделу «Создание по вариантам использования аналитической модели» данной главы (см. рис. 3.4), где классы анализа *Интерфейс кассира*, *Получение*, *Банковский счет* и *Устройство выдачи* участвуют в реализации варианта использования *Снять деньги со счета* в модели анализа. После того как эти классы анализа разработаны, с их помощью определяются уточненные классы проектирования, которые адаптированы к среде выполнения. Это иллюстрируется рис. 3.8.

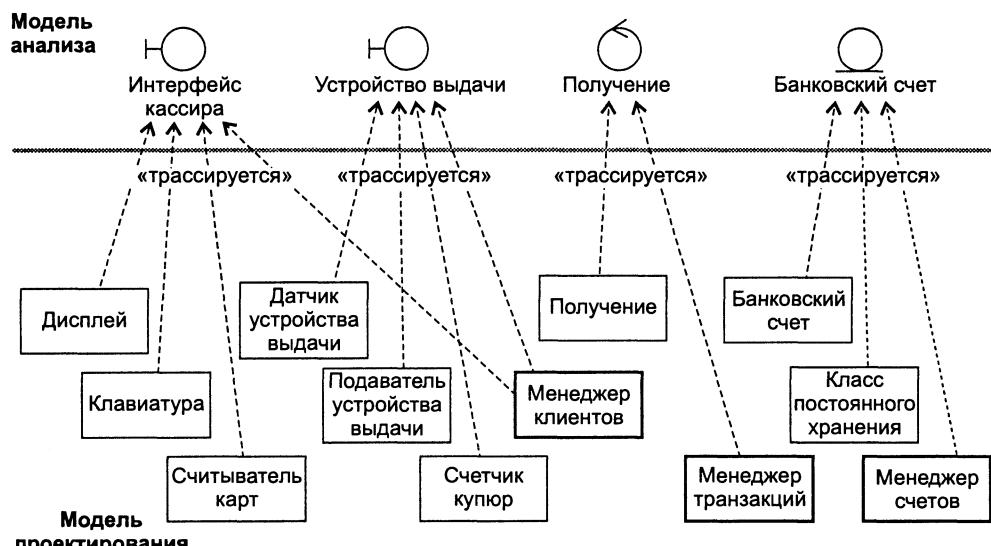


Рис. 3.8. Классы проектирования из модели проектирования трассируются от классов анализа из модели анализа

Так, например, класс анализа *Интерфейс кассира* преобразуется в четыре класса проектирования — *Дисплей*, *Клавиатура*, *Считыватель карт* и *Менеджер клиентов* (который является активным классом и поэтому изображен с толстой рамкой, см. приложение А).

Обратите внимание, что на рис. 3.8 множество классов проектирования трассируются от единственного класса анализа. Это нормально для классов проектирования, которые специфичны для конкретного приложения или разработаны для поддержки приложения или группы приложений. Поэтому структура системы, определяемая моделью анализа, естественным образом распространится и на проектирование, хотя, возможно, будут необходимы некоторые компромиссы (например, когда *Менеджер клиентов* участвует в классах проектирования *Интерфейс кассира* и *Устройство выдачи*). Кроме того, активные классы (*Менеджер клиентов*, *Менеджер транзакций* и *Менеджер счетов*) представляют собой специальные процессы, которые организуют работу других (неактивных) классов в случае распределенной системы (мы вернемся к этой проблеме в подразделе «Архитектурное представление модели развертывания» главы 4).

Как следствие, реализация варианта использования *Снять деньги со счета* в модели проектирования должна описывать, как вариант использования реализован в понятиях соответствующих классов проекта. Рисунок 3.9 изображает диаграмму классов, которая является частью реализации варианта использования.

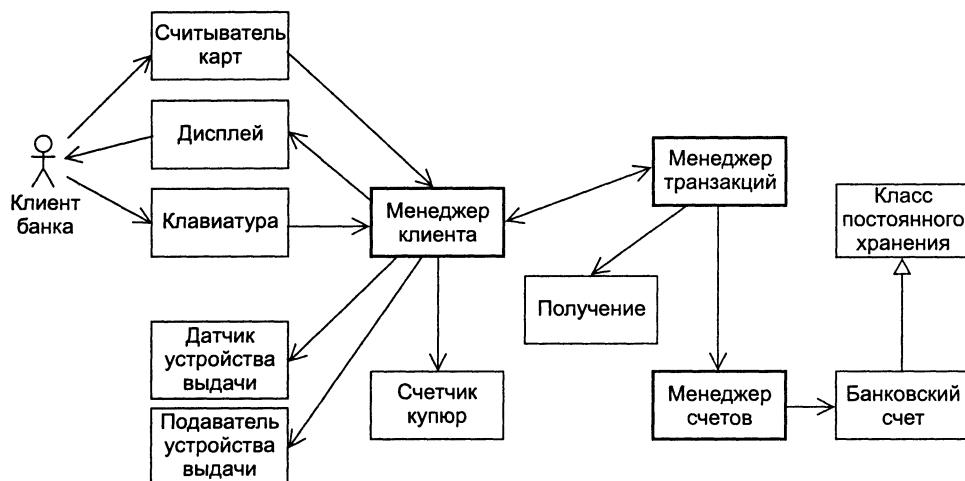


Рис. 3.9. Диаграмма классов, являющаяся частью реализации варианта использования Снять деньги со счета модели проектирования.
Каждый класс проектирования участвует в реализации варианта использования

Очевидно, что на этой диаграмме классов представлено большее количество деталей, чем на диаграмме классов модели анализа (рис. 3.5). Такая детализация необходима из-за адаптации модели проектирования к среде выполнения.

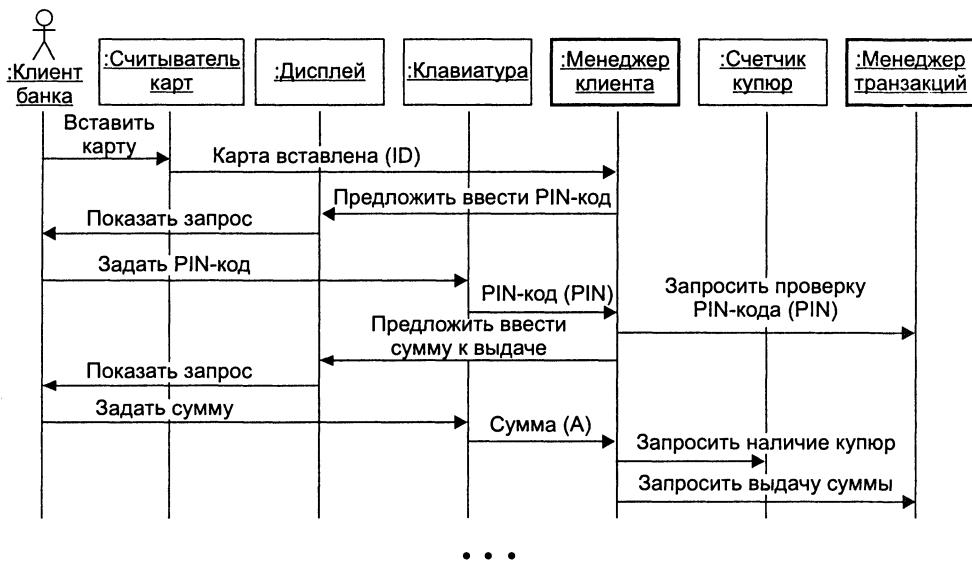


Рис. 3.10. Диаграмма последовательности, представляющая собой часть реализации варианта использования Снять деньги со счета в модели проектирования

Как и при анализе (см. рис. 3.6), мы должны идентифицировать взаимодействия между объектами проектирования, которые появились после того, как в модели проектирования были реализованы варианты использования. Прежде всего, воспользуемся диаграммой последовательности, чтобы промоделировать взаимодействия между объектами проектирования, как показано на рис. 3.10.

Диаграмма последовательности показывает, как фокус — исходно находящийся в верхнем левом углу — перемещается от объекта к объекту по мере выполнения варианта использования и как объекты обмениваются сообщениями. Сообщение, посланное одним объектом, побуждает объект-приемник этого сообщения получить фокус и выполнить одну из операций его класса.

Интересно будет сравнить эту диаграмму последовательности с ее «аналогом из анализа» — то есть диаграммой кооперации, см. рис. 3.6. Фактически первые два сообщения из диаграммы сотрудничества («1:идентифицировать» и «2:запросить снятие») превратились в весь набор сообщений диаграммы последовательности на рис. 3.10. Это дает нам понятие о сложности и уровне детализации модели проектирования по сравнению с моделью анализа.

Как и в диаграммах кооперации, разработчики могут дополнять диаграммы последовательностей текстом комментариев, поясняющим взаимодействие объектов проектирования при выполнении последовательности событий варианта использования.

Как можно заметить по этому примеру, модель проектирования, вероятно, будет содержать множество классов. Значит, нам потребуется способ организации классов. Это можно сделать при помощи подсистем, которые мы рассмотрим в следующем подразделе.

Классы группируются в подсистемы

Для большой системы, состоящей из сотен или тысяч классов, невозможно реализовать варианты использования, пользуясь только классами. Такая система будет слишком велика для того, чтобы в ней можно было обойтись без группировки высшего порядка. Классы группируются в подсистемы. Подсистема — это осмысленная группировка классов или других подсистем. Подсистема имеет набор интерфейсов, которые обеспечивают ее существование и использование. Эти интерфейсы определяют контекст подсистемы (актантов и другие подсистемы и классы).

Подсистемы нижнего уровня называются также **сервисными подсистемами** (приложение Б, см. также главу 9), потому что их классы предоставляют сервисы (более полное описание сервисов см. в подразделе «Артефакт: Описание архитектуры» главы 8). Сервисная подсистема представляет собой минимальную самостоятельную единицу дополнительных (или потенциально дополнительных) функциональных возможностей. Невозможно установить клиенту только часть подсистемы. Сервисные подсистемы также используются для моделирования совместно изменяемых групп классов.

Подсистемы могут разрабатываться как сверху вниз, так и снизу вверх. При разработке снизу вверх разработчики вводят подсистемы, отталкиваясь от уже существующих классов; вводимые подсистемы при этом собирают классы в наборы ясно определенных функций. Если же разработчики создают подсистемы сверху вниз, то первым делом, до начала выделения классов, архитектор идентифицирует подсистемы верхнего уровня и их интерфейсы. После этого разработчики исследуют отдельные подсистемы, разыскивая в них и проектируя классы этих подсистем. Понятие подсистем было введено в главе 1, подробнее они будут обсуждаться в главах 4 и 9.

Пример. *Подсистемы объединяют классы.* Разработчики собирают классы в три подсистемы, показанные на рис. 3.11. Эти подсистемы определены следующим образом: в одну подсистему помещены все классы, отвечающие за пользовательский интерфейс, вся работа с банковскими счетами была сосредоточена во второй подсистеме и классы, определяющие варианты использования, — в третьей. Преимущество размещения всех классов интерфейса пользователя в подсистеме *Интерфейс ATM* в том, что эту подсистему можно при необходимости заменить на любую другую подсистему, которая предоставляет подсистеме *Управление транзакциями* ту же самую функциональность. Альтернативная *Интерфейсу ATM* подсистема может иметь совершенно другую реализацию пользовательского интерфейса, например предназначенную для приема и выдачи монет, а не купюр.

Классы, связанные с вариантами использования, такие как класс *Получение* в подсистеме *Управления транзакциями*, заключены в свои собственные сервисные подсистемы. Каждая такая сервисная подсистема может в действительности содержать больше одного класса, но у нас упрощенный пример, и на нем этого не показать.

На рис. 3.11 также изображены интерфейсы между подсистемами. Круг представляет интерфейс. Сплошная линия между классом и интерфейсом означает, что класс предоставляет интерфейс. Стрелка с пунктирной линией от класса к интерфейсу означает, что класс пользуется интерфейсом. Для простоты мы не показы-

ваем интерфейсы, предоставляемые или используемые актантами, вместо них мы используем обычные ассоциации.

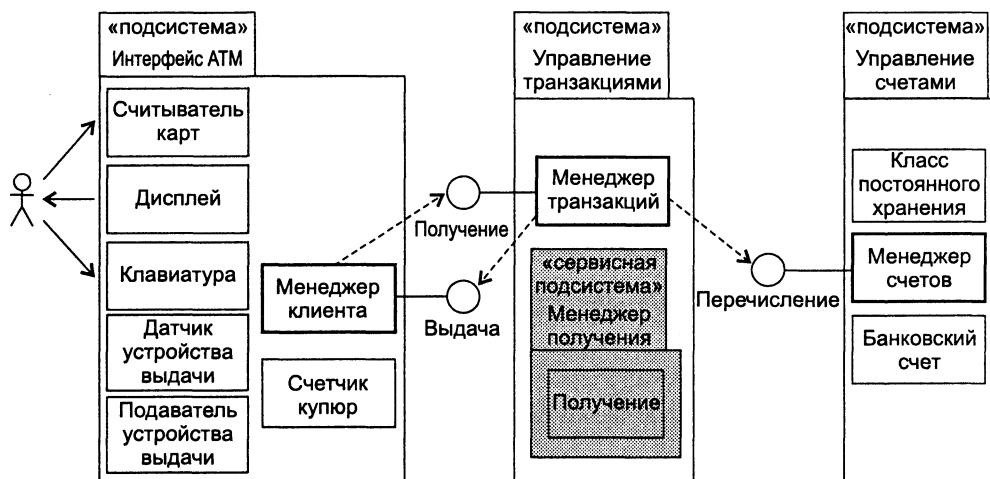


Рис. 3.11. Три подсистемы с сервисной подсистемой (изображена серым в подсистеме Менеджера транзакций)

Интерфейс *Перечисление* определяет действия при перечислении денег со счета на счет, снятия денег со счета и внесения их на счет. Интерфейс *Получение* определяет действия при запросе на снятие денег со счета. Интерфейс *Выдача* определяет действия, которые следует совершить другим подсистемам, например *Управлению транзакциями*, для выдачи денег клиенту банка.

Создание модели реализации из проектной модели

В ходе рабочего процесса реализации мы разрабатываем все необходимое для создания исполняемой системы: исполняемые компоненты, файлы компонентов (исходный код, сценарии для командного процессора и т. п.), табличные компоненты (элементы баз данных) и т. д. Компонентом мы называем физическую заменяемую часть системы, которая использует и обеспечивает реализацию некоторого набора интерфейсов. Модель реализации состоит из компонентов, в число которых входят все **исполняемые файлы** (приложение А, см. также главу 10), как-то: компоненты ActiveX, JavaBeans и другие виды компонентов.

Пример. Компоненты в модели реализации. На рис. 3.12 изображены компоненты, реализующие классы проектирования из модели проектирования (подраздел «Создание модели проектирования из аналитической модели» данной главы).

Так, файл компонента *dispenser.c* содержит исходный текст (и таким образом реализует) три класса, *Подаватель устройства выдачи*, *Датчик устройства выдачи* и *Счетчик купюр*. Этот файл компилируется и компонуется с файлом компонента *client.c* в компонент *client.exe*, который является исполняемым файлом.

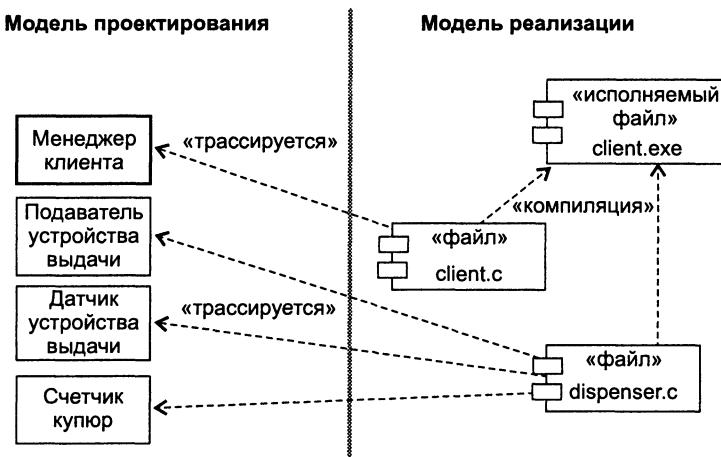


Рис. 3.12. Компоненты реализуют классы проектирования

Архитектурный контекст компонента определяется его интерфейсами. Компоненты взаимозаменяемы. Это означает, что разработчики могут заменить один компонент на другой, возможно, лучший, в том случае, если новый компонент предоставляет и использует те же самые интерфейсы, что и старый. Стандартным является способ реализации сервисных подсистем модели проектирования в компонентах, которые распределяются по узлам модели развертывания. В том случае, если, согласно модели развертывания, сервисная подсистема всегда размещается в узлах одного типа, она обычно реализуется одним компонентом. Устанавливаемая более чем на один тип узлов сервисная подсистема обычно разделяется — чаще всего путем разделения некоторых классов — на столько частей, сколько имеется типов узлов. Каждая из частей сервисной подсистемы в этом случае реализуется в виде отдельного компонента.

Пример. Реализация подсистемы компонентами. Предположим, что мы выбрали для нашего примера с банкоматом решение на основе архитектуры клиент/сервер (см. подраздел «Определение узлов и сетевых конфигураций» главы 9). Тогда мы можем разместить части сервисной подсистемы *Менеджер получения* (см. рис. 3.11), содержащие класс *Получение*, и на клиенте, и на сервере. В этом случае *Менеджер получения* необходимо было бы реализовать в виде двух компонентов — «*Получение на Клиенте*» и «*Получение на Сервере*».

Если компоненты реализуются с применением объектно-ориентированного языка программирования, реализация классов производится напрямую. Каждый класс проектирования соответствует классу реализации, типа классов C++ или Java. Каждый файл компонента может содержать один или несколько этих классов, что зависит от правил выбранного языка программирования.

Однако для создания рабочей системы недостаточно только разработать код. Разработчики, ответственные за реализацию компонентов, отвечают также и за то, чтобы перед передачей на сборку и тестирование системы созданные ими компоненты были протестированы.

Тестирование вариантов использования

В ходе тестирования мы проверяем правильность реализации системой первоначальной спецификации. Мы создаем модель тестирования, которая состоит из тестовых примеров и процедур тестирования (приложение В; см. также главу 11), а затем выполняем тестовые процедуры, чтобы удостовериться, что система работает так, как мы и ожидали. Тестовый пример – это набор тестовых исходных данных, условий выполнения и ожидаемых результатов, разработанный с определенной целью, например, реализовать некий путь выполнения варианта использования или проверить соблюдение определенного требования. Тестовая процедура – это описание подготовки, выполнения и оценки результатов конкретного тестового примера. Тестовые процедуры могут быть получены из вариантов использования. Для локализации проблемы обнаруженные дефекты следует проанализировать. Затем определяется приоритетность обнаруженных проблем, и они решаются в порядке их важности.

Мы начали разработку в подразделе «Определение вариантов использования» с определения вариантов использования, затем в подразделе «Анализ, проектирование и разработка при реализации варианта использования» мы проанализировали, спроектировали и реализовали систему, осуществляющую эти варианты использования. Теперь опишем, как проверить, что варианты использования были осуществлены правильно. В каком-то смысле в этом нет ничего нового. Разработчики всегда проверяли варианты использования, даже раньше, чем появился сам термин «вариант использования». Тестирование возможности использования системы осмысленным для пользователей способом было традиционным способом проверки функций системы. Но, с другой стороны, это и новый подход. Новым является то, что мы идентифицируем тестовые примеры (как варианты использования в рабочем процессе определения требований) до начала проектирования системы и что мы уверены, что наш проект действительно реализует варианты использования.

Пример. Определение тестового примера на основе варианта использования. На рисунке 3.13 изображен тестовый пример *Снять деньги со счета* — основной процесс, который описывает порядок проверки основного процесса варианта использования *Снять деньги со счета*.



Рис. 3.13. Тестовый пример из модели тестирования, определяющей, как тестировать вариант использования (Снять деньги со счета) из модели вариантов использования

Заметим, что сейчас мы ввели новый стереотип для вариантов использования, а именно крест. Это сделано для того, чтобы иметь возможность отображать эти варианты использования на диаграммах (см. главу 11).

Тестовый пример определяет исходные данные, ожидаемый результат и другие условия, необходимые для проверки основного процесса варианта использования *Снять деньги со счета*.

Исходные данные.

- Счет клиента банка имеет номер 12-121-1211 и на балансе счета \$350.
- Клиент банка идентифицирует себя верно.
- Клиент банка запрашивает к выдаче \$200 со счета 12-121-1211. В банкомате достаточно денег (минимум \$200).

Результат:

- Баланс счета 12-121-1211 Клиента банка уменьшается до \$150.
- Клиент банка получает из банкомата \$200.

Условия:

Никакие другие варианты использования в ходе выполнения тестового примера не имеют доступа к счету 12-121-1211.

Отметим, что этот тестовый пример основан на описании варианта использования *Снять деньги со счета*, описанного в подразделе «Варианты использования определяют систему». Рано определяя варианты использования, мы рано начинаем планировать тестирование и рано находим эффективные тестовые примеры. Эти тестовые примеры могут усложняться по ходу проекта, по мере того, как мы расширяем наше понимание осуществления системой вариантов использования. Некоторые утилиты вообще генерируют тестовые примеры по модели проектирования — вручную остается ввести только тестовые данные, необходимые для запуска теста.

Тестирование варианта использования может проводиться как с позиции актанта, для которого система является черным ящиком, так и с позиции проекта, когда тестовый пример строится для проверки того, что экземпляры классов, участвующих в варианте использования, делают то, что им положено. Тесты по методу черного ящика могут быть определены и запланированы сразу же, как только требования хоть в какой-то степени определяются.

Существуют также и другие виды тестов, например системные тесты, тесты приемки и тесты документации. Подробнее мы поговорим о тестировании в главе 11.

Резюме

Варианты использования управляют процессом. В течение рабочего процесса определения требований разработчики могут представлять требования в виде вариантов использования. Менеджеры проектов могут планировать проект в понятиях вариантов использования, с которыми работают разработчики. В ходе анализа и проектирования разработчики создают реализации вариантов использования в понятиях классов или подсистем. Затем разработчики реализуют компоненты. Компоненты объединяются в приращения, каждое из которых реализует свой набор вариантов использования. Наконец тестеры проверяют, осуществляют ли система нужные пользователям варианты использования. Другими словами, вариан-

ты использования связывают воедино все деятельности, входящие в разработку, и управляют процессом разработки. В этом, вероятно, заключается наиболее важное преимущество использования подобной технологии.

Варианты использования приносят проекту множество преимуществ. Однако это не все. В следующей главе мы поговорим о другом ключевом аспекте Унифицированного процесса — его ориентированности на архитектуру.

4

Архитектуро-центрированный процесс

В главе 3 мы упростили изложение, указав, что варианты использования укажут нам путь через требования, анализ, проектирование, реализацию и тестирование в ходе разработки системы. Однако есть и другие способы разрабатывать программы, кроме как вслепую пробиваться через рабочие процессы, ориентируясь исключительно на варианты использования.

Одних вариантов использования недостаточно. Чтобы получить работающую систему, необходимо кое-что еще. Это «кое-что» — архитектура. Мы можем думать об архитектуре как о единой концепции системы, с которой должны согласиться (или по крайней мере смириться) все сотрудники (то есть разработчики и другие заинтересованные лица). Архитектура дает нам ясное представление обо всей системе, необходимое для управления ее разработкой.

Мы нуждаемся в архитектуре, которая описывала бы наиболее важные для нас элементы моделей. Как мы определяем, какие элементы особо важны для нас? По тому, что они направляют нашу работу с системой как в текущем цикле разработки, так и в течение всего жизненного цикла. Эти важные для архитектуры элементы моделей включают в себя некоторые подсистемы, зависимости, интерфейсы, кооперации, узлы и активные классы (приложение А, см. также подраздел «Артефакт: класс проектирования» главы 9). Они описывают основу системы, которая служит нам базой для понимания, разработки и соблюдения при этом сметы.

Давайте сравним проект по производству программного обеспечения со строительством гаража на одну машину. Сначала строитель определяет, зачем пользователю гараж. Один из вариантов использования, разумеется, *Держать автомобиль*, то есть заезжать на автомобиле внутрь, оставлять его там, а позже забирать его оттуда. Нужно ли пользователю что-то еще? Допустим, он хочет устроить там мастерскую. Это наводит строителя на мысль о необходимости осветить гараж — при помощи нескольких окон и электролампочек. Множество инструментов приводятся в действие электричеством, так что строитель проектирует польюжины розеток и электрокабель соответствующей мощности. Вот так строитель разрабатывает простую архитектуру. Он может делать это в уме, потому что раньше уже видел гаражи. Он видел верстак. Он знает, что за верстаком будет работать человек. Строитель действует не вслепую, он уже знаком с типичной архитектурой не-

большого гаража. Он просто должен собрать воедино все части, чтобы выполнялись варианты использования гаража. Если бы строитель никогда не видел гараж и слепо сосредоточился на вариантах его использования, он мог бы построить в самом деле странное сооружение. Так что он должен принимать во внимание не только функции гаража, но и его форму.

Десятикомнатный дом, собор, торговый центр или небоскреб сильно отличаются от гаража. Существует множество способов постройки больших зданий. Чтобы спроектировать такое сооружение, требуется команда архитекторов. Члены команды должны будут обмениваться информацией об архитектурных решениях. Это означает, что они должны вести записи о своей работе в такой форме, которую понимают другие члены команды. Они также должны представлять свою разработку в форме, понятной неспециалистам — владельцу, пользователям и другим заинтересованным лицам. Наконец, они должны посредством чертежей довести архитектуру до строителей и поставщиков.

Точно так же развитие большинства программных систем — или програмно-аппаратных комплексов — нуждается в предварительном обдумывании. Результаты этих раздумий должны быть записаны так, чтобы их понимали не только те, кто впоследствии будет разрабатывать эту систему, но и другие заинтересованные лица. Кроме того, эти решения, эта архитектура, не появляются как по мановению волшебной палочки, архитекторы создают ее в течение нескольких итераций на фазах анализа и определения требований и проектирования. Фактически основная цель фазы проектирования состоит в создании надежной архитектуры в виде исполняемого **базового уровня архитектуры** (приложение В). В результате мы приступим к фазе построения, имея прочный фундамент для строительства системы.

Введение в архитектуру

Нам нужна архитектура. Прекрасно. Но что мы на самом деле понимаем под структурой программы? Каждому, кто просматривал литературу по архитектуре программ, вспоминалась притча о слепых и слоне. Слон для слепцов был похож на то, что ухватил каждый из них, — на большую змею (хобот), кусок каната (хвост) или дерево (ногу). Точно так же представление об архитектуре, если свести его к краткому определению в одно предложение, создавалось у различных авторов на основе того, что оказалось перед их взором, когда они задались вопросом о том, что же это такое — архитектура.

Давайте еще раз сравним архитектуру программ и зданий. Заказчик обычно смотрит на здание как на единую сущность. Поэтому архитектор может посчитать нужным сделать уменьшенную модель здания и общий чертеж здания с нескольких точек. Эти рисунки обычно не слишком детальны, но заказчик в состоянии их понять.

Однако в ходе строительства в него вовлекаются различные виды рабочих — плотники, каменщики, кровельщики, водопроводчики, электрики, подсобники и другие. Все они нуждаются в более детальных и специализированных строительных чертежах, и все эти чертежи должны быть согласованы друг с другом. Трубы вентиляции и водопровода, например, не должны проходить по одному и тому же

месту. Роль архитектора — определить наиболее существенные аспекты общего проекта здания. Таким образом, архитектор делает набор чертежей здания, которые описывают строительные блоки, например котлован под фундамент. Инженер, отвечающий за структуру, определяет размеры балок, на которые опирается сооружение. Фундамент поддерживает стены, полы и крышу. Он содержит гнезда для лифтов, водопровода, электропроводки, кондиционеров, сантехнических систем и т. д. Однако архитектурные чертежи не настолько детализированы, чтобы по ним могли работать строители. Проектировщики многих специальностей готовят чертежи и спецификации, детализирующие выбор материалов, систем вентиляции, электроснабжения, водопровода и т. д. Архитектор несет за проект полную ответственность, но детали разрабатывают другие типы проектировщиков. Вообще говоря, архитектор является экспертом в объединении всех систем здания, но не может быть экспертом по каждой системе. Когда будут сделаны все чертежи, архитектурные чертежи будут охватывать только наиболее существенные части здания. Архитектурные чертежи — это представления всех прочих типов чертежей, и они согласуются со всеми этими прочими чертежами.

В ходе строительства многие сотрудники используют архитектурные чертежи — представления детальных чертежей — для того, чтобы получить хорошее общее представление о здании, но для выполнения своей работы они применяют более детальные строительные чертежи.

Подобно зданию, программная система — это единая сущность, но архитектор и разработчики программного обеспечения считают полезным для лучшего понимания проекта рассматривать систему с разных точек зрения. Эти точки зрения описываются различными **представлениями** (приложения А и В) моделей системы. Все представления, вместе взятые, описывают архитектуру.

Архитектура программы включает в себя данные:

- об организации программной системы;
- о структурных элементах, входящих в систему, и их интерфейсах, а также их поведении, которое определяется кооперациями, в которых участвуют элементы;
- о составе структурных элементов и элементов поведения наиболее крупных подсистем;
- о стиле архитектуры (приложение В), принятом в данной организации, — элементах и их интерфейсах, их кооперации и композиции.

Однако архитектура программ имеет отношение не только к вопросам структуры и поведения, но и к удобству использования, функциональности, производительности, гибкости, возможности многократного использования, комплексности, экономическим и технологическим ограничениям, выгодности производства и вопросам эстетики.

В подразделе «Шаги разработки архитектуры» данной главы мы рассмотрим концепцию архитектуры программ в более конкретных понятиях и опишем, как она представляется при помощи Унифицированного процесса. Но о том, на что похоже **описание архитектуры** (приложение В), мы намекнем прямо здесь. Мы уже говорили, что архитектура является представлениями моделей: представлением модели вариантов использования, представлением модели анализа, представлением модели проектирования и т. д. Этот набор представлений успешно приводится к виду 4+1, обсуждаемому в [52]. Поскольку представление модели являет-

ся его частью, или долей, представление модели вариантов использования, например, похоже на саму модель вариантов использования. Оно содержит актанты и варианты использования, но только те, которые оказывают влияние на архитектуру. Точно так же архитектурное представление модели проектирования напоминает модель проектирования, но включает в себя только те модули, которые реализуют варианты использования, влияющие на архитектуру (см. подраздел «Риск не создать правильную архитектуру» главы 12).

В описании архитектуры нет ничего сверхъестественного. Оно похоже на полное описание системы со всеми моделями (есть небольшая разница, мы рассмотрим ее позже), только меньше. Насколько оно велико? Никаких абсолютных величин для описания архитектуры не существует, но по нашему опыту для большого спектра систем достаточно от 50 до 100 страниц. Это диапазон для отдельных **прикладных систем** (приложение B), описания архитектуры для **комплектов прикладных программ** (приложение B) будут больше.

Зачем нужна архитектура?

Для того чтобы у разработчиков складывалась единая концепция большой и сложной программной системы, необходим архитектор. Сложно представить себе программную систему, не существующую в нашем трехмерном мире. Программные системы часто бывают в какой-то степени беспрецедентны или уникальны. Нередко они используют прогрессивные технологии или новое сочетание технологий. Часто они используют существующие технологии на пределе их возможностей. Кроме этого, они должны быть построены так, чтобы адаптироваться к огромному количеству вероятных в будущем изменений. Поскольку системы стали более сложными, «проблема проектирования находится вне алгоритмов и структур данных: проектирование и определение общей структуры системы является проблемой иного типа» [2].

Кроме того, часто имеется существующая система, которая исполняет некоторые из функций проектируемой системы. Определение того, что делает эта система, при минимуме документации или вообще без нее, и решение вопроса о том, какую часть унаследованного кода разработчики могли бы использовать повторно, добавляет разработке сложности.

Мы нуждаемся в архитектуре для того, чтобы:

- понять систему;
- организовать разработку;
- способствовать повторному использованию кода;
- развивать систему в дальнейшем.

Понимание системы

В организации, разрабатывающей систему, эта система должна быть понятна всем, кто имеет к ней какое-то отношение. Сделать современные системы понятней не-легко по многим причинам.

- Они реализуют сложное поведение.

- Они работают в сложном окружении.
- Они сложны технологически.
- Они часто сочетают распределенные вычисления, коммерческие продукты и платформы (например, операционные системы и СУБД) и многократно используемые компоненты и структуры.
- Они должны удовлетворять запросам как отдельных людей, так и организаций.
- В некоторых случаях они настолько велики, что руководство вынуждено разбивать разработку на множество меньших проектов, выполняемых в разных местах. Это добавляет ко всему вышеперечисленному еще и трудности с координацией работ.

Кроме того, эти факторы постоянно изменяются. Все это вызывает постоянную боязнь того, что мы перестанем контролировать ситуацию.

Способ предотвратить подобную потерю понятности — сделать разработку архитектуро-центрированной (приложение В). Поэтому первое требование, предъявляемое к описанию архитектуры — оно должно позволить разработчикам, менеджерам, клиентам и другим заинтересованным лицам понять, что должно быть сделано, достаточно полно, чтобы это облегчило их работу. Нам помогут в этом модели и диаграммы, рассматривавшиеся в 3 главе. Они могут быть использованы и для описания архитектуры. По мере того как персонал знакомится с UML, использование его конструкций для моделирования архитектуры будет все более облегчать ее понимание.

Организация разработки

Чем больше организация, занимающаяся разработкой программного обеспечения, тем больше будут затраты на общение между разработчиками с целью координации усилий. Затраты на общение растут, если части проекта выполняются в разных местах. Разделяя систему на подсистемы с четко определенными интерфейсами и назначая группу или отдельного человека, ответственного за каждую подсистему, архитектор может снизить тяжесть проблемы коммуникаций между группами, разрабатывающими отдельные подсистемы, вне зависимости от того, находятся ли они в одном здании или на разных континентах. Хорошей считается та архитектура, которая четко определяет эти интерфейсы, делая возможным снижение затрат на общение. Хорошо определенный интерфейс успешно «сообщает» разработчикам ту информацию о работе других групп, которая им необходима.

Устойчивые интерфейсы позволяют частям программы, находящимся по разные стороны от интерфейса, развиваться независимо друг от друга. Правильная архитектура и образцы проектирования (приложение В) помогают нам определить правильные интерфейсы между подсистемами. Примером может послужить шаблон *Границный класс-Управляющий класс-Класс сущности* (см. подраздел «Создание по вариантам использования аналитической модели» главы 3), который помогает нам отделять поведение варианта использования от граничных классов и базовых данных.

Повторное использование

Для того чтобы объяснить важность архитектуры для повторного использования компонентов, приведем аналогию. Такое занятие, как прокладка труб, уже давно

стандартизовано. Слесари-водопроводчики давно уже извлекают пользу из стандартизации компонентов. Вместо того, чтобы биться над подгонкой размеров «творчески изготовленных» здесь и там компонентов, водопроводчики выбирают их из стандартного набора всегда совместимых деталей.

Подобно водопроводчикам, разработчики программ, желающие повторно использовать компоненты, должны разбираться в проблемной области (приложение В) и уметь подобрать подходящие для данной архитектуры компоненты. Разработчики прикидывают, как соединить эти компоненты, чтобы удовлетворить системные требования и реализовать модель вариантов использования. Если многократно используемые компоненты доступны, они используют их. Как и стандартная водопроводная арматура, программные компоненты, предназначенные для многократного использования, разрабатываются и тестируются в расчете на совместную работу, потому что в этом случае постройка системы занимает меньше времени и дешевле обходится. Результат предсказуем. Как и в сантехнике, где стандартизация заняла века, стандартизация компонентов программного обеспечения также требует опыта, но мы ожидаем, что массовая «компонентизация» произойдет всего за несколько лет. На самом деле она уже началась.

Производству программ еще предстоит достичь того уровня стандартизации, который присущ многим отраслям материального производства, но хорошая архитектура и четкие интерфейсы — это шаги в правильном направлении. Хорошая архитектура дает разработчикам прочный базис, опираясь на который они могут двигаться дальше. Работа архитектора — создать этот базис и определить, какие подсистемы многократного использования должны использовать разработчики. Многочленно используемые подсистемы должны быть специально разработаны так, чтобы они могли использоваться совместно [49]. Хорошая архитектура помогает разработчикам понять, где будет выгодно искать правильные элементы многократного использования, и находить их. UML ускорит процесс перехода на компоненты, поскольку стандартный язык моделирования — это предпосылка для построения проблемно-зависимых компонентов, которые можно будет многократно использовать.

Развитие системы

Если есть что-то, в чем мы можем быть уверены, так это то, что любая система более-менее существенного размера будет эволюционировать. Она эволюционирует уже во время разработки. Позже, когда она будет эксплуатироваться, изменения в окружающей среде приведут к ее дальнейшему развитию. В течение обоих этих периодов система должна с легкостью воспринимать изменения. Это означает, что разработчики должны иметь возможность изменять части проекта и реализации, не заботясь о том, что последствия этих изменений вдруг проявятся где-то в неожиданном месте системы. В большинстве случаев они должны иметь возможность добавить к системе новую функциональность (то есть варианты использования), не вызвав при этом разрушения существующего проекта и реализации. Другими словами, система должна быть гибкой, или устойчивой к изменениям. Другой способ обозначить эту цель — сказать, что система должна быть способна к аккуратному развитию. Плохо спланированные системы, напротив, с течением времени обычно деградируют, требуя постоянного латания дыр, пока наконец это не становится просто невыгодно.

Пример. Система AXE фирмы Ericsson. О важности архитектуры. Начало разработки системы телекоммуникационного оборудования AXE фирмы Ericsson приходится на 1970-е годы. Уже тогда в разработке использовалась ранняя версия наших архитектурных постулатов. Описание архитектуры программного обеспечения было важным артефактом, направляющим процессы разработки в течение всего срока жизни системы. Разработка архитектуры направлялась несколькими постулатами, которые теперь включены в Унифицированный процесс.

Один из этих постулатов — принцип модульности функций. Классы или эквивалентные им проектные элементы были сгруппированы в функциональные блоки, или сервисные подсистемы, с которыми клиенты могли работать как с дополнительными модулями (даже если блоки поставлялись всем клиентам). Сервисная подсистема имела большую внутреннюю связность. Изменения, вносимые в систему, обычно локализовались в пределах одной сервисной подсистемы и редко затрагивали большее число подсистем.

Другим принципом было разделение проектирования интерфейсов и проектирования подсистем. Целью было создать «подключаемые» проектные решения с тем, чтобы несколько сервисных подсистем могли использовать одинаковый интерфейс. Смена одной сервисной подсистемы на другую могла бы в этом случае быть проделана без изменения клиентам сервисной подсистемы, которым важны интерфейсы, а не код сервисных подсистем.

Третьим принципом было непосредственное сопоставление сервисных подсистем проектирования одному или нескольким компонентам реализации. Компоненты сервисных подсистем могли быть по-разному распределены между узлами. Для каждого узла, на котором выполнялась сервисная подсистема, существовал ровно один компонент. Таким образом, если сервисная подсистема запускалась на центральном компьютере (сервере), то существовал всего один компонент сервисной подсистемы. Если сервисная подсистема запускалась и на сервере, и на клиенте, компонентов было два. Применение этого принципа помогало управлять изменениями, вносимыми в программы в различных установках.

Еще один принцип предусматривал низкую связность между сервисными подсистемами. Единственным средством связи между сервисными подсистемами были сигналы. Так как сигналы асинхронны (семантика «посылка-без-ожидания», то есть система, послав сигнал, не ждет ответа), они поддерживают не только инкапсуляцию, но и распределенное выполнение.

Поскольку и начало ее разработки, и ее продолжение производилось на основе правильно разработанной архитектуры, система AXE продолжает использоваться и сегодня, имея более ста клиентов и несколько тысяч установок. Ожидается, что при условии регулярного обновления она проживет еще десятки лет.

Варианты использования и архитектура

Мы уже упоминали, что между вариантами использования и архитектурой существует некоторое взаимодействие. В главе 3 мы кратко рассмотрели, как разрабатывать систему, предоставляющую пользователям правильные варианты использования. Если система предоставляет правильные варианты использования — то есть варианты использования с высокой производительностью, качеством и удоб-

ством использования, — то пользователи могут использовать эту систему для выполнения своей работы. Но как нам получить такую систему? Ответ мы уже знаем: нужно построить такую архитектуру, которая позволит нам реализовать варианты использования как сейчас, так и в будущем и делать это рентабельно.

Уточним, как происходит этот процесс, рассмотрев сначала, что оказывает влияние на архитектуру (рис. 4.1), а затем — какие факторы влияют на варианты использования.



Рис. 4.1. На архитектуру оказывают влияние не только варианты использования, но и другие типы требований и продуктов. Разрабатывать архитектуру нам помогает опыт предыдущей работы и образцы архитектуры

Как мы уже упоминали, архитектура зависит от того, какие варианты использования мы желаем поддерживать в системе. Варианты использования — это направляющие для архитектуры. В конце концов, мы хотим иметь архитектуру, которая является подходящей для осуществления наших вариантов использования. На ранних итерациях мы выбираем несколько вариантов использования, которые, как мы полагаем, позволят нам наилучшим образом разработать архитектуру. Эти важные для создания архитектуры варианты использования будут включать те, которые необходимы клиентам в текущем выпуске и, возможно, в последующих выпусках.

Однако на архитектуру будут влиять не только важные для архитектуры варианты использования, но и другие факторы.

- Системное программное обеспечение, которым мы хотим пользоваться для построения системы (например, операционной системой или СУБД).
- Программное обеспечение **среднего уровня** (приложение B), которое мы хотим использовать. Например, мы должны выбрать брокер объектных запросов, предоставляющий механизм прозрачного маршалинга и доставки сообщений распределенным объектам в гетерогенных средах [56] или платформонезависимый каркас для создания пользовательского графического интерфейса.
- Унаследованные системы, которые войдут в нашу систему. Включая унаследованную систему, например существующую банковскую систему, в наш про-

граммный комплекс, мы можем повторно использовать множество существующих функций, но будем вынуждены приспосабливать нашу архитектуру к старому продукту.

- Стандарты и правила компании, которые мы должны соблюдать. Например, мы можем быть вынуждены использовать язык определения интерфейсов IDL [29], созданный OMG для определения интерфейсов к классам или стандарт телекоммуникаций TMN [30] – для определения объектов системы.
- Общие нефункциональные требования (то есть не привязанные к конкретному варианту использования), типа требований к доступности, времени восстановления или потребления памяти.
- Требования к распределению, определяющие, как система распределяется по компьютерам, например согласно архитектуре клиент/сервер.

Мы можем считать пункты, перечисленные на рис. 4.1 справа, ограничениями, побуждающими нас к созданию определенной архитектуры.

Разработка архитектуры происходит на итерациях фазы проектирования. Упрощенный, немного наивный подход – мысленная модель – осуществляется следующим образом. Мы начинаем с определения типа архитектуры, например по слойной архитектуре (приложение В). После этого в течение нескольких билдов (приложение В; см. также главу 10) первой итерации мы создаем архитектуру.

В первом билде мы работаем с частями **общего уровня приложений** (приложение В), которые являются общими для предметной области и не специфичны для системы, которую мы собираемся разрабатывать. Имеется в виду, что мы выбираем системное программное обеспечение (**уровень системного программного обеспечения**, приложение В, см. также подраздел «Определение подсистем среднего уровня и уровня системного программного обеспечения» главы 9), средний уровень, наследуемые системы, используемые стандарты и правила. Мы решаем, какие узлы будут присутствовать в нашей модели развертывания и как они должны взаимодействовать друг с другом. Мы решаем, как удовлетворить общие нефункциональные требования, например требования по доступности. На этом первом проходе достаточно будет добиться общего понимания приложения.

В втором билде мы работаем с архитектурой **специфического уровня приложения** (приложение В). Мы выбираем комплект архитектурно значимых вариантов использования, определяем требования, анализируем их, проводим проектирование, реализуем и тестируем их. Результатом будут новые подсистемы, реализованные в виде компонентов. Они создаются для поддержки выбранных нами вариантов использования. Кроме того, теперь, рассмотрев систему в понятиях вариантов использования, мы можем захотеть внести некоторые изменения в архитектурно значимые компоненты, выбранные во время первого билда. Для реализации вариантов использования создаются новые компоненты и изменяются старые. Таким образом мы адаптируем архитектуру для лучшего удовлетворения вариантам использования. Затем мы делаем следующий билд, затем еще и еще, пока не закончим итерацию. Когда фаза проектирования заканчивается, архитектура должна быть стабилизирована.

После того, как мы создали устойчивую архитектуру, мы можем вводить в систему полный список функциональных возможностей, реализуя в ходе фазы построения оставшиеся варианты использования. Варианты использования, реализу-

емые в течение фазы построения, обычно разрабатываются с использованием в качестве исходных данных требований пользователей и клиентов (рис. 4.2). Варианты использования также находятся под влиянием архитектуры, выбранной на фазе проектирования.

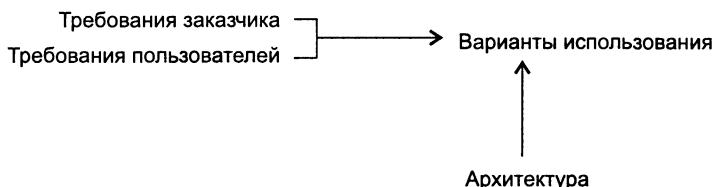


Рис. 4.2. Варианты использования могут быть разработаны на основе данных, полученных от заказчика и пользователей. Также на варианты использования влияет уже созданная архитектура

Чтобы лучше определять новые варианты использования, мы применяем наше знание архитектуры. Когда мы оцениваем ценность каждого предложенного варианта использования и затраты на него, мы делаем это применительно к существующей архитектуре. Некоторые варианты использования будут легко осуществимы, в то время как осуществить другие будет затруднительно.

Пример. Подгонка вариантов использования под существующую архитектуру. Клиент пожелал иметь функцию, контролирующую загрузку процессора. Это требование было определено как вариант использования, который измерял загрузку, используя высший уровень приоритета процессора. Осуществление этого варианта использования потребовало бы внесения изменений в используемую операционную систему реального времени. Вместо этого команда разработчиков предложила осуществлять требуемые функциональные возможности при помощи отдельного внешнего устройства, которое будет запрашивать систему и измерять время отклика. Клиент получил более надежные измерения, а разработчики ушли от необходимости изменять базовую архитектуру.

Мы проводим переговоры с клиентом и договариваемся об изменении вариантов использования путем подгонки вариантов использования и итогового проекта под используемую архитектуру для упрощения работы. Эта подгонка означает, что мы должны пересмотреть созданные ранее подсистемы, интерфейсы, варианты использования, реализации вариантов использования, классы и т. д. Подогнав варианты использования под архитектуру, мы сможем создавать новые варианты использования, подсистемы и классы с меньшими затратами, чем ранее.

Итак, с одной стороны, архитектура находится под влиянием вариантов использования, которые мы хотим реализовать в системе. Варианты использования управляют архитектурой. С другой стороны, мы используем построенную архитектуру, чтобы правильнее определять требования в виде вариантов использования. Архитектура управляет вариантами использования (рис. 4.3).

Что будет в начале, варианты использования или архитектура? Это – типичная проблема курицы и яйца. Такие проблемы лучше всего разрешаются итеративным подходом. Сначала, отталкиваясь от хорошего понимания предметной области (приложение B), мы строим предварительную архитектуру, не рассматривая детализированные варианты использования. Затем мы выбираем несколько

существенных вариантов использования и перерабатываем архитектуру, приспособливая ее под эти варианты использования. Затем мы выбираем еще варианты использования и снова перерабатываем архитектуру и т. д.



Рис. 4.3. Варианты использования управляют разработкой архитектуры, а архитектура направляет реализуемые варианты использования

На каждой итерации мы выбираем и реализуем набор вариантов использования, чтобы подтвердить, а при необходимости и улучшить архитектуру. На каждой итерации мы также содействуем реализации архитектуры специфического уровня приложения, которая основывается на выбранных нами вариантах использования. Варианты использования, таким образом, помогают нам по мере продвижения в деле разработки постепенно улучшать архитектуру. Это — одно из преимуществ использования методики разработки, управляемой вариантами использования. Мы вернемся к рассмотрению итеративного процесса в главе 5.

Подведем итоги. Добротная архитектура — это такая архитектура, которая позволяет нам эффективно поддерживать правильные варианты использования как сейчас, так и в будущем.

Шаги разработки архитектуры

Архитектура разрабатывается в ходе итераций, в основном в фазе проектирования. Каждая итерация выполняется так, как описано в главе 3: сначала определение требований, затем анализ, проектирование, реализация и тестирование, при этом основное внимание уделяется существенным для архитектуры вариантам использования и другим требованиям. Результатом, получаемым к концу фазы проектирования, будет базовый уровень архитектуры — скелет системы с минимумом программных «мыщ».

Какие варианты существенны для архитектуры? Мы тщательно рассмотрим этот вопрос в подразделе «Расстановка приоритетов вариантов использования» главы 12. Пока скажем лишь, что существенными для архитектуры вариантами использования являются те, которые помогают нам уменьшить самые серьезные риски, а так же наиболее важны для пользователей системы или помогают нам охватить все важные функции системы, не оставляя ничего незамеченным. Как показано в этом подразделе, реализация, интеграция и тестирование базового уровня архитектуры дают архитектору и другим сотрудникам уверенность в том, что архитектура действительно рабочая. В этом невозможно убедиться при помощи «бумажного» анализа и проектирования. Работающий базовый уровень архитектуры наглядно демонстрирует состояние разработки для любого сотрудника, желающего увидеть ее своими глазами.

Базовый уровень архитектуры

В конце фазы проектирования мы разработали модели системы, включающие в себя наиболее важные с точки зрения архитектуры варианты использования и их реализаций. Мы также остановили свой выбор, как рассматривалось в подразделе «Варианты использования и архитектура» данной главы, на некоторых стандартах, которых будем придерживаться в дальнейшем, используемом системном программном обеспечении и программном обеспечении среднего уровня, определившись с повторно используемыми унаследованными системами и необходимым распределением. Итак, мы получили предварительные версии модели вариантов использования, модели анализа, модели проектирования и других. Эта совокупность моделей (рис. 4.4) является базовым уровнем архитектуры¹. В него входят версии всех тех моделей, которые присутствуют в полномасштабной системе в конце фазы построения.

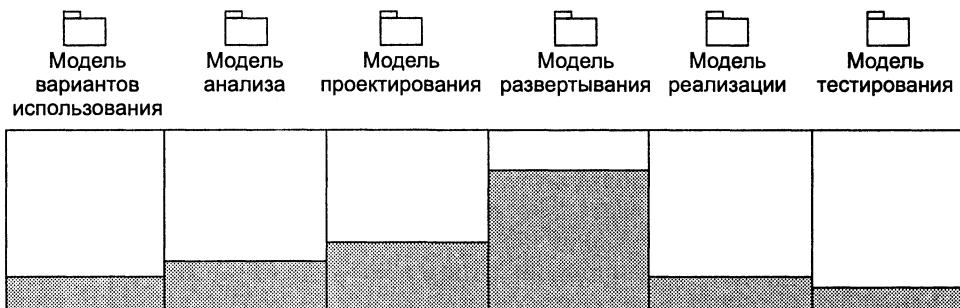


Рис. 4.4. Базовый уровень архитектуры — внутренний релиз системы, направленный на определение архитектуры

Базовый уровень содержит скелет тех же самых подсистем, компонентов и узлов, что и проектируемая система, но без большей части «мяса». Однако он обладает поведением и содержит исполняемый код. Базовая система, далее развивающаяся, превращается в полномасштабную, при этом возможны небольшие изменения в структуре и поведении. Изменения не могут быть значительны по определению, потому что в конце фазы проектирования архитектура стабилизована, в противном случае мы должны продолжать фазу проектирования, пока эта цель не будет достигнута.

На рис. 4.4 заштрихованная часть каждой модели представляет долю завершения модели к концу стадии проектирования, то есть версию модели, которая входит в базовый уровень архитектуры. Целые прямоугольники (сумма заштрихованных и незаштрихованных частей) представляют версию модели в конце фазы внедрения, то есть базовый уровень **внешнего выпуска** (приложение В) (не следует воспринимать слишком серьезно размеры заштрихованных областей, изображенных на рис. 4.4, они показаны в качестве иллюстрации). Между базовым уровнем

¹ Это не совсем так. В конце фазы проектирования у нас имеется версия модели вариантов использования, которая содержит как важные с точки зрения архитектуры, так и рядовые варианты использования (до 80 % всех включаемых в модель), описываемые нами в порядке осознания бизнес-вариантов. Таким образом, на базовом уровне архитектуры модель вариантов использования и модель анализа более полны, чем показанные на рис. 4.4. Однако в данном случае такое упрощение вполне допустимо.

архитектуры и базовым уровнем внешнего выпуска лежат несколько промежуточных уровней, представляющих собой **внутренние выпуски** новых версий моделей (приложение В). Мы можем рассматривать эти новые версии как приращения к предыдущим версиям, начиная с базового уровня архитектуры. Каждая новая версия модели является развитием предыдущей. Различные модели рис. 4.4, конечно, не развиваются независимо друг от друга. Каждый вариант использования в модели вариантов использования соответствует, например, реализации варианта использования в моделях анализа и проектирования и тестовому примеру из модели тестирования. Процесс и структура узла должны обеспечивать производительность, необходимую для вариантов использования. В противном случае следует изменить или варианты использования, или модель развертывания, возможно, следует изменить способ размещения активных классов по узлам, чтобы они лучше выполняли свою работу. Такие изменения в моделях развертывания или проектирования могут привести и к изменению модели варианта использования, если сделанные изменения этого требуют. Элементы различных моделей, как мы упоминали в подразделе «Связи между моделями» главы 2, связаны друг с другом за-висимостью трассировки.

Однако базовый уровень архитектуры, то есть внутренний выпуск системы в конце стадии разработки, содержит не только элементы модели. Он также включает в себя описание архитектуры. Это описание создается фактически одновременно с работой по созданию тех версий моделей, которые являются частью базового уровня архитектуры, часто даже до этого. Задача описания архитектуры — направлять команду разработчиков в течение всего срока жизни системы — не только итераций текущего жизненного цикла, но и всех будущих циклов. Это стандарт, которым разработчики должны руководствоваться как сейчас, так и в будущем. Поскольку архитектура должна быть стабильной, стандарт также не должен сильно изменяться.

Описание архитектуры может принимать различные формы. Это могут быть выдержки из описаний моделей, входящих в базовый уровень архитектуры, или обзор, сделанный на основе этих выдержек в удобном для чтения виде. Мы вернемся к этому в подразделе «Описание архитектуры». Однако описание включает выдержки, или представления, моделей, являющихся частью базового уровня архитектуры. По мере того как система развивается и модели на поздних стадиях разработки становятся больше, описание продолжает включать в себя представления моделей новых версий. Предполагая, что базовый уровень архитектуры стабилен — то есть существенные для архитектуры элементы моделей стабильны и не изменяются в ходе итераций, — мы можем считать, что описание архитектуры также будет стабильно в каждый момент времени, включая представления моделей системы.

Приятно увидеть, что можно создать стабильную архитектуру в ходе фазы проектирования первого жизненного цикла системы, когда существует еще только, скажем, 30% первого выпуска. Эта архитектура станет фундаментом, на который система будет опираться всю свою жизнь. Так как любое изменение фундамента будет очень недешевым, а в некоторых случаях — и весьма болезненным делом, важно получить устойчивую архитектуру как можно раньше. С одной стороны, разработка архитектуры для конкретной системы — это всегда нечто новое. С другой стороны, люди разрабатывают архитектуру не первый год. Накоплены опыт

и знания по разработке удачной архитектуры. Существует множество обобщенных решений — структур, коопераций и физических архитектур, — которые создавались многие годы, с которыми должен быть знаком каждый опытный архитектор. Такие решения обычно называют «образцами», как архитектурные образцы, описанные в [12] и образцы проектирования из [23]. Обобщенные образцы — это ресурсы, на которые архитектор может положиться.

Использование образцов архитектуры

Идеи архитектора Кристофера Александера о том, как использовать «языки образцов» для систематизации важных принципов и прагматик в проектировании зданий и общин, вдохновили нескольких членов объектно-ориентируемого сообщества на определение, сбор и проверку множества образцов программного обеспечения [2]. «Сообщество образцов» определяет образец как «решение часто встречающейся в проектировании проблемы». Многие из этих образцов проектирования изложены в литературе, где образцы описываются с использованием стандартных шаблонов документирования. Эти шаблоны дают образцу название и содержат краткое описание проблемы, вызывающих ее причин, решение в терминах коопераций с перечислением участвующих классов и взаимодействий объектов этих классов. Шаблоны также содержат текст образца и примеры его использования на нескольких языках программирования, краткое перечисление преимуществ и недостатков образца и ссылки на похожие образцы. Согласно Александеру, это предложено для того, чтобы программисты выучили названия и поведение многих стандартных образцов и применяли их, чтобы делать проекты лучше и более понятными. Такие образцы проектирования, как *Фасад*, *Декоратор*, *Обозреватель*, *Стратегия* и *Посетитель*, широко известны и используются.

Сообщество образцов также применило идею образцов с немного иным шаблоном документа для сбора стандартных решений часто встающих архитектурных проблем. В их число входят *Слои*, *Каналы и фильтры*, *Брокер*, *Грифельная доска*, *Горизонтально-вертикальные метаданные* и *MVC*. Другие разрабатывали образцы, которые следует использовать в ходе анализа («образцы анализа»), в ходе реализации («идиомы», которые соотносят типовые объектно-ориентируемые структуры со специфическими аспектами языков типа C++ и Smalltalk) и даже для повышения эффективности организационных структур («организационные образцы»). Как правило, образцы проектирования достаточно близки к объектно-ориентированным языкам программирования. Так, примеры к ним приводятся на C++, Java и Smalltalk. В то же время архитектурные образцы имеют дело, главным образом, с системами или подсистемами и интерфейсами, и примеры к ним обычно не содержат кода. Хорошая схема классификации приводится в [55].

С нашей управляемой моделями точки зрения мы определим образец как шаблон кооперации, который представляет из себя обобщенную кооперацию и может быть специализирован так, как определено **шаблоном** (приложение А). После специализации образцы проектирования становятся кооперациями классов и вариантов с поведением, описываемым диаграммами взаимодействия. Мы используем шаблон кооперации, потому что предполагаем получать обобщенные решения. Чтобы специализировать образец (например, определяя имена классов, указанных в шаблоне, их число и т. д.), используются наследование, расширение и другие механизмы. Во многих случаях шаблон кооперации при специализации разрас-

тается до конкретной кооперации, которая подобна вариантам использования. Для расширенного изучения образцов проектирования см. [23].

Архитектурные образцы используются таким же образом, но с упором на крупноблочные структуры и взаимодействия подсистем и даже систем. Существует множество образцов архитектуры. Коротко обсудим наиболее интересные из них.

Образец *Брокер* [12] – обобщенный механизм для управления распределенными объектами. Он позволяет объектам вызывать другие удаленные объекты через брокера, который передает запрос к узлу и процессу, содержащему запрашиваемый объект. Эта передача незаметна, что означает, что вызывающему объекту нет необходимости знать, является ли вызываемый объект удаленным. Образец *Брокер* часто использует образец проектирования *По доверенности*, который обеспечивает локальный прокси с такими же интерфейсами, как и у удаленного объекта, делая стиль и детали распределенной связи незаметными.

Существуют и другие образцы, которые помогают нам понять аппаратные средства создаваемых систем, и такие, которые помогают нам проектировать систему поверх этих аппаратных средств. Примерами таких образцов могут быть *Клиент-сервер*, *Трехуровневая система* и *Одноранговая система*. Эти образцы определяют структуру модели развертывания и рекомендуют, как следует размещать компоненты по узлам. В подразделе «Описание архитектуры» данной главы мы рассмотрим, как можно применить образец *Клиент-сервер* в АТМ-системе, описанной нами в главе 3. В нашем примере распределение по образцу *Клиент-сервер* предполагает наличие одного клиента, который выполняет весь код интерфейса пользователя и часть кода бизнес-логики (классы управления) для каждого реального банкомата. Узел – сервер – хранит фактические счета и бизнес-правила, по которым проверяется каждая операция.

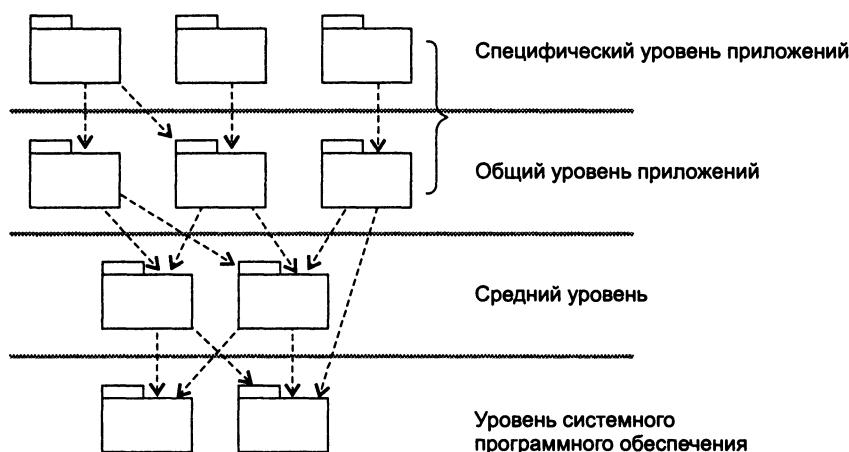


Рис. 4.5. Поуровневая архитектура организует систему в виде уровней подсистем

Образец *Уровни* применим к различным типам систем. Это – образец, который определяет, как организовать многоуровневую модель проектирования, то есть такую модель, в которой компоненты одного слоя могут быть связаны только с компонентами соседних слоев. Важность этого образца в том, что он упрощает

понимание и организацию разработки сложных систем. Образец *Уровни* уменьшает число связей в системе благодаря тому, что на нижних уровнях не известны никакие подробности или интерфейсы верхних уровней. Кроме того, он помогает нам определить фрагменты, пригодные для повторного использования, а также обеспечивает структуру, помогающую нам принять решение по вопросу «купить готовое или разработать самим».

Системы с многоуровневой архитектурой содержат на верхнем уровне специфические прикладные подсистемы. Они построены на базе подсистем более низких уровней, таких как каркасы и библиотеки классов. Посмотрим на рис. 4.5. Общий уровень приложений содержит подсистемы, которые не специфичны для единственного приложения и могут многократно использоваться для множества различных приложений в пределах той же предметной области. Архитектуру двух нижних уровней можно определить, не вдаваясь в детальное рассмотрение вариантов использования, потому что эти уровни не связаны с предметной спецификой приложения. Архитектура двух верхних уровней создается на базе существенных для архитектуры вариантов использования (эти уровни — предметно-зависимые).

Уровень (приложение В) — это набор подсистем, которые имеют одинаковую степень общности и одинаковый уровень стабильности интерфейса: нижние уровни будут общими для различных приложений и должны иметь более стабильные интерфейсы, в то время как верхние уровни более предметно-зависимы и могут иметь менее стабильные интерфейсы. Так как нижние уровни обеспечивают редко изменяющиеся интерфейсы, разработчики, создающие верхние слои, могут использовать стабильные нижние уровни в качестве фундамента. Подсистемы различных уровней могут повторно использовать варианты использования, другие подсистемы низшего уровня, классы, интерфейсы, кооперации и компоненты более низких уровней. В пределах одной системы может быть использовано множество образцов архитектуры. Образцы структурирования модели развертывания (например, *Клиент-сервер*, *Трехуровневая система* и *Одноранговая система*) можно комбинировать с образцом *Уровни*, который помогает структурировать модель проектирования. Образцы, относящиеся к структурам разных моделей, часто ортогональны друг другу. Даже образцы, относящиеся к одной и той же модели, нередко прекрасно сосуществуют. Например, образец *Брокер* хорошо дополняет образец *Уровни*, и оба они используются в модели проектирования. Образец *Брокер* указывает, как работать с распределением скрытых объектов, в то время как образец *Уровни* рекомендует метод организации всего проекта. Образец *Брокер*, по всей вероятности, будет реализован в виде подсистемы в среднем уровне.

Обратите внимание, что иногда один из образцов является основным. Например, в многоуровневой системе образец *Уровни* может определять общую архитектуру и распределение работ (слои создаются различными группами), в то время как *Каналы* и *Фильтры* входят в один или более уровней. С другой стороны, в системе, построенной на основе образца *Каналов* и *Фильтров*, общая архитектура представляла бы собой поток между фильтрами, а уровни использовались бы только для описания некоторых фильтров.

Описание архитектуры

Базовый уровень архитектуры, разработанный во время фазы проектирования, сохраняется, как мы отмечали в подразделе «Базовый уровень архитектуры»,

в форме описания архитектуры. Это описание порождается теми версиями различных моделей системы, которые были созданы в ходе фазы проектирования, как это изображено на рис. 4.6. Описание архитектуры — это выдержка или, как мы говорим, набор представлений — возможно, аккуратно приведенный в удобный для чтения вид — моделей базового уровня архитектуры. Представления моделей содержат существенные для архитектуры элементы. Множество элементов моделей, входящих в базовый уровень архитектуры, разумеется, войдет также и в описание архитектуры. Однако в описание архитектуры войдут не все элементы базового уровня, потому что для получения работающего базового уровня мы вынуждены были разработать некоторые элементы моделей, не существенные для архитектуры, но необходимые для создания исполняемого кода. Так как базовый уровень архитектуры используется не только для разработки архитектуры, но и для определения требований к системе на таком уровне, когда возможно создание детального плана, модель варианта использования этого базового уровня также может содержать не только те варианты использования, которые важны для построения архитектуры.

Описание архитектуры будет поддерживаться актуальным в течение всего времени жизни системы, отражая вносимые изменения и дополнения, которые существенны для архитектуры. Эти изменения обычно незначительны и могут включать в себя:

- обнаружение новых абстрактных классов и интерфейсов;
- добавление новых функциональных возможностей к существующим подсистемам;
- обновление компонентов многократного использования до новых версий;
- перестройку структуры процесса.

Само описание архитектуры может нуждаться в изменении, но увеличиваться в размерах не должно. Оно лишь обновляется, чтобы оставаться актуальным (рис. 4.6). На рис. 4.6 заштрихованные фигуры справа вверху показывают разрастание в ходе построения различных моделей по мере завершения. Описание архитектуры же почти не увеличивается (справа внизу), так как большая часть архитектуры была определена еще на фазе проектирования. В архитектуре происходят лишь незначительные изменения, обозначенные на рисунке различной штриховкой.

Как говорилось ранее, описание архитектуры содержит представления моделей. Они включают в себя варианты использования, подсистемы, интерфейсы, некоторые классы и компоненты, узлы и кооперации. Описание архитектуры также включает в себя важные для определения архитектуры требования, которые не описываются вариантами использования. Это обычно нефункциональные требования, определенные как дополнительные, например проблемы безопасности и важные ограничения, касающиеся **распределенного выполнения и параллельности** (приложение B, см. также подраздел «Артефакт: Класс проектирования» главы 9).

Описание архитектуры должно также содержать краткое описание платформы, наследуемых систем и используемого покупного программного обеспечения, например Java RMI, для распределенных объектов. Кроме того, важно описать кар-

касы, ответственные за обобщенные **механизмы** (приложение В; см. также подраздел «Определение обобщенных механизмов проектирования» главы 9), например сохранения и восстановления объектов в реляционной базе данных. Эти механизмы могут использоваться в нескольких реализациях вариантов использования, поскольку они были спроектированы для многократно используемых коопераций. В описании архитектуры, кроме того, должны быть задокументированы все использованные образцы архитектуры.

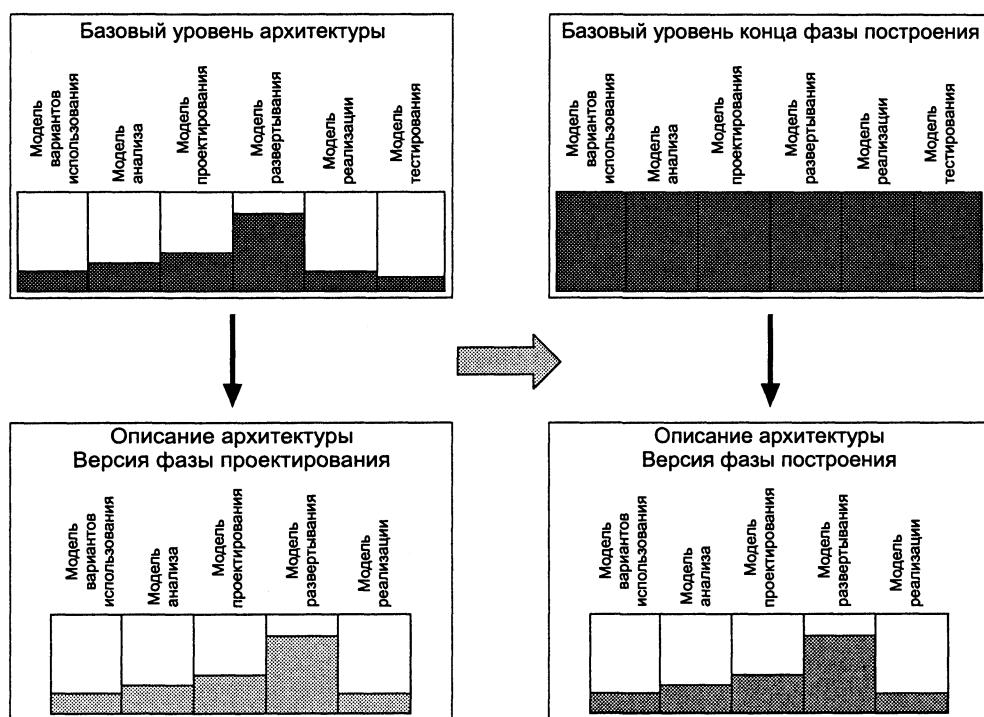


Рис. 4.6. Описание архитектуры почти не увеличивается и изменения в ней незначительны

Описание архитектуры выдвигает на передний план наиболее важные проблемы проектирования и открывает их для рассмотрения и получения откликов. Эти проблемы сразу же должны быть обсуждены, проанализированы и решены. Анализ может, например, включать оценку требуемой производительности, требований к памяти и прикидку возможных в будущем требований, которые могут разрушить архитектуру.

Несмотря на свою детализацию, описание архитектуры остается высокоуровневым представлением. С одной стороны, оно не предназначено для всеобъемлющего описания и не должно завалить участников разработки избытком деталей. Это дорожная карта, а не детальная спецификация системы. С другой стороны, оно должно содержать информацию для всех разработчиков, так что даже 100 страниц не будет чрезмерно много. Люди используют большой документ, если в нем есть то,

что им нужно, и в такой форме, что они могут это понять. В общем, в описании архитектуры должно быть написано, что разработчикам делать на работе.

При чтении описания архитектуры мы можем заметить, что некоторые подсистемы описаны в нем поверхностно, а интерфейсы и кооперации части подсистем — подробно. Причина этого отличия в том, что хорошо описанные подсистемы важны для построения архитектуры и должны находиться под контролем архитектора (см. подразделы «Планирование итераций» главы 12 и «Проектирование архитектуры» главы 14).

Это может помочь нам понять, что не относится к архитектуре. Большинство классов с операциями, интерфейсами и атрибутами, которые скрыты в подсистемах или сервисных подсистемах (закрыты для остальной части системы), для архитектуры не важны. Подсистемы, которые являются вариантами других подсистем, для архитектуры не важны. Опыт показывает, что менее 10% классов важны для архитектуры. Остальные 90% для архитектуры не важны, потому что большая часть системы и не подозревает об их существовании. Изменение в одном из таких классов не затрагивает чего-либо существенного вне сервисной подсистемы. Также и большинство реализаций вариантов использования не важны для архитектуры, поскольку они не налагаются на систему никаких дополнительных ограничений. Именно поэтому архитекторы могут планировать архитектуру, имея лишь часть вариантов использования и других требований. Большинство реализаций вариантов использования представляют собой просто дополнительное поведение, которое легко реализовать, даже при том, что оно составляет большую часть предоставляемой системой функциональности. Подведем итог: как только мы получаем архитектуру, реализовать большинство функциональных возможностей системы действительно легко.

Описание архитектуры не содержит информации, необходимой исключительно для проверки архитектуры. Не существует никаких тестовых примеров, тестовых процедур и архитектурного представления модели тестирования. Эти вопросы не относятся к архитектуре. Однако, как можно заметить из рис. 4.6, базовый уровень архитектуры содержит версии всех моделей, включая модель тестирования. Таким образом, базовый уровень, на котором основано описание архитектуры, как и прочие базовые уровни, подвергается тестированию.

Архитектор, создающий архитектуру

Архитектор создает архитектуру совместно с другими разработчиками. Они работают, создавая систему, которая будет иметь высокую производительность и высокое качество, будет высоко функциональной, тестируемой, дружественной к пользователю, надежной, постоянно доступной, точной, расширяемой, устойчивой к изменениям, разумной, легкой в обслуживании, переносимой, безопасной, защищенной и экономичной. Они знают, что должны существовать в пределах этих ограничений и как-то делать выбор между ними. Поэтому и существует архитектор. Архитектор имеет самую высокую степень технической ответственности в этих вопросах. Он выбирает между образцами архитектуры и существующими изделиями и назначает связи между подсистемами, чтобы разделить их интересы. Разделение интересов здесь означает создание такого проекта, в котором изменения в одной из подсистем не откликаются еще в нескольких.

Истинная задача состоит в том, чтобы исполнять требования приложения наилучшим из способов, возможных при текущем состоянии технологии, по доступной для приложения цене, другими словами, иметь возможность рентабельно осуществлять функциональные возможности приложения (то есть варианты использования) как в настоящее время, так и в будущем. В этом архитектору помогают UML и Унифицированный процесс. UML содержит мощные инструменты для формулирования архитектуры, а Унифицированный процесс дает детальное описание того, что входит в хорошую архитектуру. Но даже в этом случае выбор архитектуры осуществляется в итоге на основе оценки, основанной на квалификации и опыте. Архитектор является ответственным за эту оценку. Когда архитектор в конце фазы проектирования передает описание архитектуры менеджеру проекта, это означает: «Теперь я уверен, что мы сможем построить систему, не сталкиваясь с какими-либо серьезными техническими проблемами».

Квалифицированный архитектор должен быть знатоком в двух областях. Первая из них — это предметная область, в которой он работает, поскольку он должен общаться со всеми заинтересованными лицами, а не только разработчиками, и понимать их. Вторая — знание разработки программного обеспечения, вплоть до способности писать код, поскольку он должен выдать разработчикам архитектуру, координировать их усилия и получать от них отклик. Также полезно, когда архитектор имеет опыт разработки систем, подобных данной.

Труден хлеб архитектора в программной фирме. Он не должен быть менеджером проекта, поскольку эта позиция предполагает множество обязанностей и помимо архитектуры. Он должен иметь искреннюю поддержку руководства как при создании архитектуры, так и настаивая на ее соблюдении. Однако он должен быть достаточно уступчив, чтобы воспринимать то полезное, что говорят ему разработчики и другие заинтересованные лица. Это краткое описание качеств, необходимых архитектору. Для больших систем одного архитектора может не хватить. Вместо этого разумно было бы сделать так, чтобы архитектуру разрабатывала и поддерживала группа архитекторов.

Разработка архитектуры занимает значительное время. Отводимое на архитектуру время привлекает к себе внимание в планах разработки и может вызывать беспокойство у менеджеров, привыкших, что время разработки в основном уходит на реализацию и тестирование. Опыт показывает, однако, что если поздние фазы разработки проводятся на базе хорошей архитектуры, то полная продолжительность разработки заметно снижается. Об этом мы поговорим в главе 5.

Описание архитектуры

Мы довольно долго рассуждали об архитектуре, не приводя серьезного примера. Сейчас мы приведем такой конкретный пример описания архитектуры. Однако перед этим мы должны объяснить, почему он непрост.

Напомним, что описание архитектуры — это лишь соответствующие выдержки из моделей системы (то есть ничего нового при этом не добавляется). Первая версия описания архитектуры — это выдержки из тех версий моделей, которые мы получили к концу фазы проектирования первого цикла жизни системы. Учитывая, что мы не пытаемся перевести эти выдержки в более удобочитаемую форму, описание архитектуры очень похоже на обычные модели системы. Это означает,

что архитектурное представление модели варианта использования напоминает обычную модель варианта использования. Единственное различие состоит в том, что архитектурное представление содержит только важные для архитектуры варианты использования (см. подраздел «Риск не создать правильную архитектуру» главы 12), а в модель варианта использования входят все варианты использования. То же самое справедливо и для архитектурного представления модели проектирования. Оно похоже на модель проектирования, но содержит только те реализации вариантов использования, которые существенны для архитектуры.

Пример также трудно привести и потому, что интересно обсуждать архитектуру только реальных систем, а когда мы приводим в книге подробное описание системы, это по необходимости должна быть очень маленькая система. В этом подразделе для иллюстрации работы с архитектурными представлениями мы рассмотрим пример банкомата (ATM) из главы 3. При этом мы будем сравнивать представления и полные модели системы.

Описание архитектуры разбито на пять частей, по одной для каждой модели. Имеется представление модели вариантов использования, представление модели анализа (он присутствует не всегда), представление модели проектирования, представление модели развертывания и представление модели реализации. Представление модели тестирования отсутствует, потому что она не имеет значения для описания архитектуры и используется только для проверки базового уровня архитектуры.

Архитектурное представление модели вариантов использования

Архитектурное представление модели вариантов использования представляет наиболее важные актанты и варианты использования (или сценарии вариантов использования). О модели вариантов использования см. подраздел «Определение вариантов использования» главы 3.

Пример. Архитектурное представление модели вариантов использования банкомата (ATM). В примере с банкоматом *Снять деньги со счета* – наиболее важный вариант использования. Без него невозможно существование реальной ATM системы. *Внести деньги на счет* и *Перечислить деньги на другой счет* считаются менее важными для среднего клиента банка вариантами использования.

Поэтому для определения архитектуры архитектор предлагает полностью осуществить вариант использования *Снять деньги со счета* во время фазы проектирования, не считая никакие другие варианты использования (или их части) существенными для определения архитектуры. (На деле это решение было бы немного опрометчивым, но оно используется здесь только для пояснения.)

Таким образом, архитектурное представление модели вариантов использования будет содержать полное описание варианта использования *Снять деньги со счета*.

Архитектурное представление модели проектирования

Архитектурное представление модели проектирования содержит большинство важных для архитектуры классификаторов модели проектирования: наиболее важ-

ные подсистемы, интерфейсы, несколько особенно важных классов, прежде всего активные классы. Оно также показывает, как наиболее важные варианты использования реализованы в понятиях этих классификаторов, то есть реализаций вариантов использования. Активные классы (занятия) также обсуждаются в следующем подразделе данной главы при изучении модели развертывания (в которой активные классы распределяются по узлам).

Пример. *Архитектурное представление модели проектирования банкомата (ATM).* В подразделе «Создание модели проектирования из аналитической модели» главы 3 мы идентифицировали три активных класса: *Менеджер клиентов*, *Менеджер транзакций* и *Менеджер счетов* (рис. 4.7). Эти активные классы включаются в архитектурное представление модели проектирования.

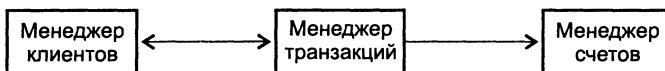


Рис. 4.7. Статическая структура архитектурного представления модели проектирования для системы ATM.
Диаграмма классов, содержащая активные классы

Кроме того, из подраздела «Классы группируются в подсистемы» главы 3 нам уже известно о трех подсистемах: *Интерфейс ATM*, *Управление Транзакциями* и *Управление Счетами*; рис. 4.8. Эти подсистемы необходимы для понимания варианта использования *Снять деньги со счета*, а значит, и для понимания архитектуры. Модель проектирования включает в себя также множество других подсистем, но здесь они рассматриваться не будут.

Подсистема *Интерфейс ATM* берет на себя всю работу с клиентом банка, включая прием от него информации и команд, предоставление информации и запросов клиенту и печать квитанций. Подсистема *Управление Счетами* занимается обслуживанием всей информации длительного хранения о банковских счетах и операциях со счетами. Подсистема *Управление Транзакциями* содержит классы для поведения, определяемого вариантами использования, например поведения, определяемого вариантом использования *Снять деньги со счета*. В примере из подраздела «Классы группируются в подсистемы» главы 3 мы упоминали, что классы, определяемые вариантом использования, часто собираются в сервисные подсистемы, такие как сервисные подсистемы для классов *Снять деньги со счета*, *Перечислить деньги на другой счет* и *Внести деньги на счет* подсистемы *Управление Транзакциями* (на рис. 4.8 не показаны). В реальности каждая сервисная подсистема обычно содержит несколько классов, но наш пример очень прост.

Подсистемы на рис. 4.8 обеспечивают поведение друг друга через интерфейсы, такие как интерфейс *Перечисления*, предоставляемый *Управлением Счетами*. Интерфейсы *Перечисления*, *Получение* и *Выдача* описаны в подразделе «Классы группируются в подсистемы» главы 3. Существуют также интерфейсы *Перечисленные*, *Вклады* и *История*, но они не вовлечены в вариант использования, который мы обсуждаем в этом примере, поэтому мы не будем о них говорить.

Однако статической структуры недостаточно. Мы также должны показать, как важные для архитектуры варианты использования реализуются в подсистемах модели проектирования. Поэтому мы еще раз опишем вариант использования

Снять деньги со счета, на этот раз в понятиях взаимодействующих подсистем и актантов, как показано на рис. 4.9, с использованием диаграммы кооперации (приложение А). Объекты классов, принадлежащих подсистемам, взаимодействуют друг с другом с целью исполнения экземпляра варианта использования. Объекты посылают друг другу сообщения, эта коммуникация изображена на диаграмме. Сообщения несут имена, определяющие операции, принадлежащие интерфейсам подсистем. Эта структура сообщений обозначается знаком :: (например, *Получение::совершить* (количество, счет), где *Получение* – интерфейс, предоставляемый классом подсистемы *Управление Транзакциями*).

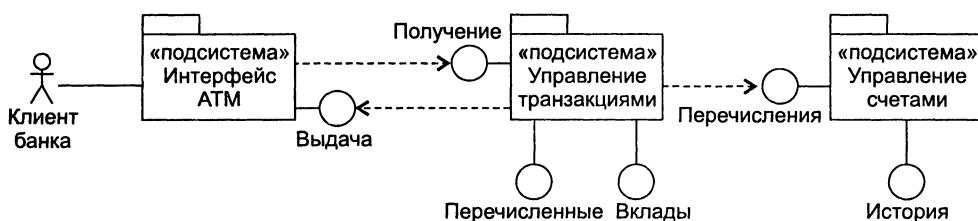


Рис. 4.8. Статическая структура архитектурного представления модели проектирования для системы ATM. Диаграмма классов, отображающая подсистемы и соединяющие их интерфейсы



Рис. 4.9. Подсистемы, кооперирующиеся для осуществления варианта использования *Снять деньги со счета*

Следующий список кратко описывает процессы в реализации варианта использования. Текст почти такой же, как в подразделе «Создание по вариантам использования аналитической модели» главы 3 (описание реализации варианта использования), но здесь она представлена в понятиях подсистем, а не классов.

Предварительное условие: клиент банка имеет банковский счет, доступный через ATM.

1. Актант *Клиент банка* желает снять деньги со счета и идентифицирует себя через *Интерфейс ATM*, возможно, используя магнитную карту с номером и PIN-кодом. *Клиент банка* также определяет, сколько он хочет снять и с какого счета. Мы предполагаем, что подсистема *Интерфейс ATM* в состоянии проверить идентификацию.
2. *Интерфейс ATM* запрашивает у подсистемы *Управление транзакциями* получение денег. Подсистема *Управление транзакциями* отвечает за выполнение всей последовательности получения денег в одной транзакции, так чтобы деньги одновременно были списаны со счета и выданы *Клиенту банка*.

3. Управление транзакциями запрашивает у Управления счетами выдачу денег. Подсистема Управление счетами определяет, возможна ли выдача денег, и, если это так, уменьшает счет на указанную сумму и возвращает сообщение, что выдачу можно совершить.
4. Управление транзакциями разрешает подсистеме Интерфейс ATM выдать деньги.
5. Интерфейс ATM выдает деньги Клиенту банка.

Архитектурное представление модели развертывания

Модель развертывания определяет физическую архитектуру системы в понятиях связанных узлов. Узлы — это аппаратные устройства, на которых могут выполняться компоненты программного обеспечения. Часто мы знаем, на что будет похожа физическая архитектура системы раньше, чем приступим к разработке. В этом случае узлы и связи между ними могут быть выстроены в модель развертывания уже во время рабочего процесса определения требований.

В ходе выполнения проекта мы решаем, какие классы являются активными, то есть нитями или процессами. Мы решаем, что должен делать каждый активный объект, каким должен быть цикл жизни активных объектов и как активные объекты должны передавать, синхронизировать и совместно использовать информацию. Активные объекты размещаются в узлах модели развертывания. Размещая активные объекты в узлах, мы рассматриваем потенциал узлов, как-то: мощность процессора, размер памяти и характеристики подсоединений, например пропускную способность и доступность.

Узлы и связи модели развертывания и распределение активных объектов по узлам могут быть показаны при помощи диаграмм развертывания (приложение А). На этих диаграммах можно также показать распределение по узлам выполняемых компонентов. Система ATM из нашего примера разнесена по трем различным узлам.

Пример. Архитектурное представление модели развертывания банкомата (ATM). Клиент банка получает доступ к системе через узел Клиента ATM, который обращается для выполнения транзакций к Серверу Приложений ATM (рис. 4.10). Сервер Приложений ATM, в свою очередь, использует Сервер Данных ATM для выполнения некоторых транзакций, например со счетами. Это верно не только для варианта использования Снять деньги со счета, который мы классифицировали как важный для архитектуры, но и для других вариантов использования, таких как Внести деньги на счет и Перечислить деньги на другой счет. В подразделе «Создание модели проектирования из аналитической модели» главы 3 мы описываем, какие классы выбрали для реализации варианта использования Снять деньги со счета.

Когда узлы определены, на них можно повесить функциональность. Для простоты мы сделаем это, загружая на каждый узел подсистему целиком (см. рис. 4.8). Подсистема Интерфейс ATM загружается на узел Клиент ATM, подсистема Управление Транзакциями — на Сервер Приложений ATM и подсистема Управление Счетами — на Сервер Данных ATM. В результате каждый активный класс этих подсистем (см. подраздел «Классы группируются в подсистемы» главы 3 и рис. 3.10)

разворачивается на соответствующем узле, превращаясь в работающий на узле процесс. Каждый из этих процессов обслуживает и держит в пространстве процесса объекты других (обычных, неактивных) классов подсистемы. Развёртывание активных объектов показано на рис. 4.11, где активные классы системы ATM распределены по узлам, а активные объекты показаны в виде прямоугольников с утолщенной рамкой.

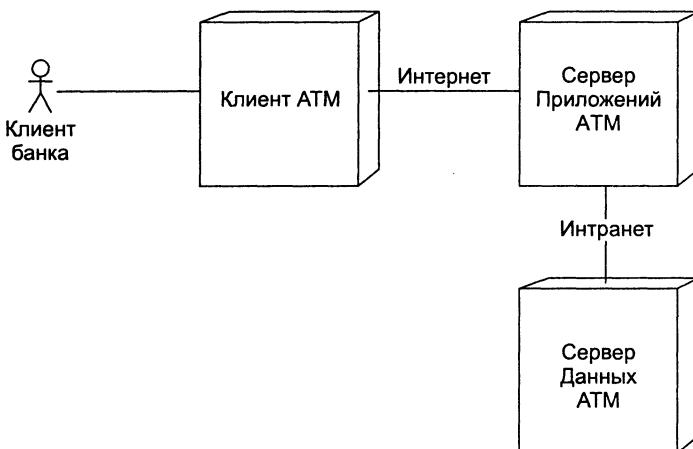


Рис. 4.10. Модель развертывания определяет три узла: Клиент ATM, Сервер Приложений ATM и Сервер Данных ATM

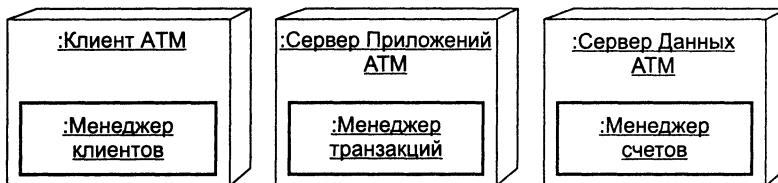


Рис. 4.11. Архитектурное представление модели развертывания

Это упрощенный пример распределения системы. В реальной системе распределение, конечно, сложнее. Одним из альтернативных решений проблемы распределения было бы использование для распределения объектов программного обеспечения среднего уровня, например брокера объектных запросов (ORB).

Архитектурное представление модели реализации

Модель реализации является прямым отображением моделей развертывания и проектирования. Каждая сервисная подсистема модели проектирования обычно превращается в один компонент для каждого типа узла, на который она устанавливается. Но так бывает не всегда. Иногда один и тот же компонент (например, компонент управления буфером) создан и выполняется на нескольких узлах. Некоторые языки обеспечивают создание компонентов — контейнеров, например

JavaBeans. В противном случае классы организуются в файлы программ, которые содержат выбранные наборы компонентов.

В подразделе «Создание модели реализации из проектной модели» главы 3 мы упомянули, что сервисную подсистему *Менеджер получения* можно рассматривать как два компонента, *Получение на сервере* и *Получение на клиенте*. Компонент *Получение на сервере* мог бы реализовывать класс *Получение*, а компонент *Получение на клиенте* – класс *Передача получения*. В нашем простом примере эти компоненты реализовывали бы по одному классу каждый. В реальной системе в каждой сервисной подсистеме было бы еще по несколько классов, так что компонент реализовывал бы несколько классов.

Три интересных понятия

Что такое архитектура?

Это то, что определяет архитектор в описании архитектуры. Описание архитектуры позволяет архитектору управлять разработкой системы с позиции «технаря». Архитектура программ концентрируется как на существенных структурных элементах системы – подсистемах, классах, компонентах и узлах, так и на кооперации этих элементов посредством интерфейсов.

Варианты использования побуждают архитектуру быть такой, чтобы система обеспечивала желаемые назначение и функциональные возможности при разумной производительности. Архитектура должна быть всеобъемлющей, но достаточно гибкой для добавления новых функций, и поддерживать повторное использование существующего программного обеспечения.

Как ее получить?

Архитектура разрабатывается итеративно в течение фазы проектирования, в ходе определения требований, анализа, проектирования, реализации и тестирования. Для построения базового уровня архитектуры, или «скелета» системы, используются существенные для архитектуры варианты использования и некоторые другие данные. В эти другие данные входят требования к системному программному обеспечению, программному обеспечению среднего уровня, использование унаследованных систем, нефункциональные требования и т. д.

Как ее описать?

Описание архитектуры – это представление моделей системы, представления моделей вариантов использования, анализа, проектирования, реализации и развертывания. Описание архитектуры представляет те части системы, которые необходимо понимать всем разработчикам и другим заинтересованным лицам.

5

Итеративный и инкрементный процесс

Чтобы быть эффективным, процесс разработки программного обеспечения должен иметь последовательность четко обозначенных вех (приложение В), которые обеспечивают менеджеров и остальную команду разработчиков критериями окончания текущей фазы. В соответствии с этими критериями они принимают решение о переходе к следующей фазе.

В ходе каждой фазы процесс, руководствуясь этими критериями, проходит ряд итераций и приращений (приложение В).

В фазе анализа и планирования требований главным критерием является жизнеспособность, достигаемая при помощи:

- определения и снижения рисков (приложение В; см. также раздел «Риски влияют на планирование проекта» главы 12), критичных для жизнеспособности системы;
- перехода от набора ключевых требований к потенциальной архитектуре при помощи моделирования вариантов использования;
- составления исходной сметы для широкого диапазона затрат, усилий, сроков и качества продукта;
- составления бизнес-плана (см. подробнее о бизнес-плане в главах 12–16), подтверждающего, что проект, вероятно, будет прибыльным при широком диапазоне условий.

В фазе проектирования главным критерием является возможность построить систему с учетом экономических требований, достигаемая при помощи:

- определения и снижения рисков, важных для разработки системы;
- определения большинства вариантов использования, описывающих предоставляемую разрабатываемой системой функциональность;
- перехода от потенциальной архитектуры к исполняемому базовому уровню архитектуры;
- подготовки плана проекта (приложение В), достаточно детального, чтобы проводить по нему фазу построения;

- создания сметы на суженном диапазоне условий для уточнения предварительной оценки;
- завершения бизнес-плана: проект стоит делать.

В фазе построения главным критерием является способность системы начать работать в среде пользователей, достигаемая при помощи последовательности итераций, направленной на периодический выпуск билдов и добавление к системе приращений, так, чтобы в ходе этой фазы жизнеспособность системы была всегда подтверждена имеющимися исполняемыми файлами.

В фазе внедрения главным критерием является система, доведенная до эксплуатационных кондиций, получаемая путем:

- модификации продукта, для решения проблем, не замеченных на более ранних фазах;
- исправления дефектов (приложение В; см. также подраздел «Артефакт: дефект» главы 11).

Одна из целей Унифицированного процесса — помочь архитекторам, разработчикам и другим заинтересованным лицам воспринять значение ранних фаз разработки. С этой точки зрения лучше всего процитировать совет Барри Боэма, данный им несколько лет назад [7]:

«Я не могу преувеличить важность вехи архитектуры жизненного цикла¹ для вашего проекта и вашей карьеры. Если проект не удовлетворяет критериям окончания фазы для вехи архитектуры жизненного цикла, не переходите к полномасштабной разработке. Возобновите работу заинтересованных лиц и разработайте новый план проекта, который успешно удовлетворял бы критериям окончания фазы для этой вехи».

Фазы и итерации внутри них более детально рассматриваются в третьей части книги.

Введение в итеративность и инкрементность

В главах 3 и 4 мы говорили, что процесс разработки программного обеспечения должен управляться вариантами использования и ориентироваться на архитектуру. Это два из трех ключевых положений Унифицированного процесса. Эти положения оказывают явственное техническое воздействие на результат процесса. «Быть управляемым вариантами использования» означает, что каждый шаг в создании продукта опирается на реальные действия пользователей. Это заставляет разработчиков постоянно контролировать, действительно ли система выполняет потребности пользователей. «Быть архитектуро-ориентированным» означает, что разработка основана на создании на ранних фазах образца архитектуры, которым будут управляться усилия по построению системы, создавая условия для плавного развития системы не только в ходе создания текущего выпуска, но и в течение всего времени жизни изделия.

¹ Она соответствует нашей вехе фазы планирования. — Примеч. авт.

Достижение правильного баланса между вариантами использования и архитектурой во многом подобно достижению правильного баланса между функциями и формой при создании любого другого изделия. Достижение баланса требует некоторого времени. То, что мы имеем вначале, соответствует проблеме курицы и яйца, мы упоминали об этом в подразделе «Варианты использования и архитектура» главы 4. Курица и яйцо, такие, какими мы их видим теперь, возникли в результате почти бесконечных итераций в ходе длительного процесса эволюции. Точно так же в ходе гораздо более короткого процесса разработки программного обеспечения разработчики сознательно создают этот баланс (между вариантами использования и архитектурой) в процессе ряда итераций. Таким образом, итеративный и инкрементный подход к разработке представляет собой третье ключевое положение Унифицированного процесса.

Разрабатываем понемногу

Третье ключевое положение предполагает стратегию разработки программ по маленьким управляемым кусочкам.

1. Мы планируем маленький кусочек.
2. Мы специфицируем, проектируем и реализуем маленький кусочек.
3. Мы собираем, тестируем и запускаем маленький кусочек на каждой итерации.

Если вы удовлетворены результатами шага, то беретесь за следующий шаг. После каждого шага вы получаете отклик, который позволяет вам перейти к работе над следующим шагом. Тогда вы переходите к следующему шагу, затем к следующему и т. д. Когда вы пройдете все запланированные вами шаги, вы создадите продукт, который можно передавать заказчикам и пользователям.

На итерациях ранних фаз главным образом происходит обозрение проекта, снижение опасных рисков и построение базового уровня архитектуры. Затем, по мере продвижения в ходе разработки, мы постепенно снижаем оставшиеся риски и реализуем компоненты. Форма итераций изменяется, и их результатом становятся приращения.

Проект развития программного обеспечения преобразует «дельту» (или изменения) пользовательских требований в дельту (или изменения) программного продукта (см. подраздел «Проект порождает продукт» главы 2). При итеративном и инкрементном подходе это преобразование изменений происходит небольшими шагами. Другими словами, мы разбиваем проект на множество мини-проектов, каждый из которых как раз укладывается в одну итерацию. Каждая итерация имеет все атрибуты проекта по разработке программного обеспечения: планирование, серию рабочих процессов (определение требований, анализ и проектирование, реализацию, тестирование) и подготовку выпуска.

Но итерация не является полностью независимой сущностью. Это стадия внутри проекта. Она создается как часть проекта. Мы говорим, что это — мини-проект, потому что в ходе него запросы заинтересованных лиц не выполняются целиком. Каждый из этих мини-проектов подобен старому водопадному процессу, поскольку он осуществляет все виды деятельности, присущие водопадному процессу. Мы могли бы именовать каждую итерацию «мини-водопадом».

Итеративный жизненный цикл приносит реальные результаты в форме внутренних (или предварительных) выпусков, каждый из которых добавляет некоторое приращение и снижает риски, вызывающие беспокойство. Эти выпуски можно передавать клиентам и пользователям и таким образом получать отклики, необходимые для проверки сделанного.

Планировщики должны делать так, чтобы итерации шли в прямой последовательности, и таким образом ранние итерации создавали бы базу знаний для более поздних итераций. Ранние итерации проекта разработки дают нам прирост понимания требований, проблем, рисков и **предметной области** (приложение В), в то время как более поздние итерации приносят дополнительные приращения, которые в конечном счете складываются в новый **внешний выпуск** (приложение В), то есть коммерческий продукт. Окончательный успех для планировщиков — это последовательность итераций, постоянно устремленная вперед. У нас не должно возникать необходимости возвращаться на две-три итерации назад, чтобы подправить модель из-за информации, которую мы получили на более поздней итерации. Мы не желаем ходить по осыпи, делая два шага вперед и один назад.

Резюмируем сказанное. Жизненный цикл состоит из последовательности итераций. Некоторые из них, особенно первые, помогают нам выделить риски, установить выполнимость работы, построить базовое ядро программы и написать бизнес-план. Другие итерации, особенно более поздние, добавляют к программе приращения, пока не будет создан внешний выпуск продукта.

Итерации помогают управлять планированием, организацией, мониторингом и контролем за проектом. Повторения организуются в рамках четырех фаз, каждая из которых имеет свои требования к подбору персонала, финансированию, планированию и критериям начала фазы и ее окончания. В начале каждой фазы руководство может принимать решения о том, как ее выполнять, какими должны быть ее результаты и какие риски следует уменьшить.

Чем не является итерация

Некоторые менеджеры думают, что «итеративный или инкрементный» — это просто ученые названия, применяемые вместо «гадательный», причудливые псевдонимы для «угадывания». Они боятся, что эти слова просто скрывают действительность, в которой разработчики не знают, что они делают. В фазе анализа и определения требований и даже в начале фазы проектирования в этом есть доля истины. Например, если разработчики не определили критические или опасные для проекта риски, то это утверждение истинно. Если они еще не приняли базовую концепцию или не создали базового уровня архитектуры, это утверждение истинно. Если они еще не понимают, как удовлетворить наиболее критичные требования, это утверждение истинно. Да, действительно, они не знают, что делают.

А если они притворятся, что знают, что делают, от этого будет какой-то прок? Если они будут планировать работу, основываясь на неполной информации, от этого будет какой-то прок? Если они попытаются следовать этому ненадежному плану, от этого будет какой-то прок? Ну разумеется, нет.

Подчеркнем еще раз, чем не является итеративный жизненный цикл.

- Это не случайное блуждание.
- Это не манежик для разработчиков.

- Это не вешь, интересная только разработчикам.
- Это не перепроектирование одного и того же куска снова и снова, пока разработчики наконец не наткнутся случайно на что-то работающее.
- Он не непредсказуем.
- Он не является оправданием промахов в планировании и управлении.

На самом деле управляемая итерация далека от случайности. Она запланирована. Это инструмент, который менеджеры могут использовать для управления процессом. Она уже в начале процесса снижает риски, которые могут угрожать дальнейшему развитию разработки. **Внутренние выпуски** (приложение В), выходящие после итераций, позволяют заинтересованным лицам получить обратную связь, приводящую, в свою очередь, к более ранней коррекции курса, которому следует разработка.

Почему мы используем итеративную и инкрементную разработку?

В трех словах: программы получаются лучше. Немного более развернуто: чтобы пройти главные и вспомогательные вехи, по которым мы контролируем ход разработки. И еще более развернуто.

- Чтобы как можно раньше получить описание критических и опасных рисков.
- Чтобы сформулировать архитектуру для создания программного обеспечения.
- Чтобы обеспечить каркас, который наилучшим образом поддерживает неизбежно появляющиеся в ходе разработки дополнительные требования и другие изменения.
- Чтобы создавать систему в несколько приемов путем приращений, а не всю сразу и до конца, что сильно поднимает стоимость изменений.
- Чтобы обеспечить процесс разработки, который повысит эффективность работы сотрудников.

Снижение рисков

Разработка программного обеспечения сопряжена с рисками, как и вообще любая инженерная деятельность. С точки зрения пророка менеджмента Питера Ф. Друкера, «риск свойственен переводу существующих ресурсов в будущие надежды» [18]. При разработке программного обеспечения мы имеем дело с этими реалиями жизни, стараясь идентифицировать риски настолько рано и устраниТЬ их настолько быстро, насколько это возможно. Риск — это воздействие, которое может привести к потерям или иному ущербу. Риск — это фактор, сущность, элемент или курс, представляющий опасность для разработки, величина которой не определена. В разработке программного обеспечения мы можем определить риск как нечто, с определенной вероятностью подвергающее опасности успешность проекта. Например.

- Брокер объектных запросов (приложение В), который мы первоначально рассматриваем, может быть не в состоянии обработать 1000 запросов на поиск объекта *счет-удаленного-клиента* в секунду.
- Система реального времени, вероятно, должна будет получать множество данных, в начале разработки не определенных. После этого ей, по-видимому, придется производить с этими данными вычисления значительного объема, подробно не оговоренные. Затем ей, скорее всего, нужно будет в течение короткого, но не указанного в начале разработки времени посыпать управляющий сигнал.
- Система телефонной коммутации должна, вероятно, будет давать отклик на различные входящие сигналы за считанные миллисекунды, как это будет указано заказчиком — компанией — оператором связи.

Как много лет назад писал Барри Боэм, для программного обеспечения необходима модель процесса, которая «создает управляемый рисками подход к процессу разработки программного обеспечения вместо применявшегося раньше управляемого документами или управляемого кодом процесса» [6]. Унифицированный процесс удовлетворяет этому критерию, поскольку опасные риски устраняются в нем на первых двух фазах, анализа и определения требований и проектирования, а устранение оставшихся происходит в порядке их серьезности в начале фазы построения. Унифицированный процесс идентифицирует риски, управляет ими и уменьшает их в ходе итераций на ранних фазах. В результате мы можем быть уверены, что какие-либо неопознанные или проигнорированные риски не проявятся впоследствии и не подвергнут проект опасности.

Итеративный подход к сокращению рисков имеет определенное сходство с водопадным подходом (приложение В). Модель водопада предполагает разработку, проходящую единожды через ряд шагов: определение требований, анализ, проектирование, реализацию и тестирование. При таком подходе, когда проект доходит до реализации, интеграции (приложение В) и тестирования, в нем должны участвовать все разработчики. В ходе интеграции и тестирования проблемы начинают превалировать над всем остальным. Менеджер проекта в этом случае бывает вынужден повторно поручить сотрудникам — часто более опытным разработчикам — решить возникшие проблемы для того, чтобы можно было продолжать работу. Однако все разработчики уже заняты, и менеджерам проектов очень нелегко «вырвать» незанятых сотрудников, которые обладали бы квалификацией, необходимой для решения возникших проблем. Для решения этой проблемы более опытные разработчики назначаются «дежурными уборщиками», в то время как менее опытные сидят и ждут. Сроки срываются, проект превышает смету. Если особенно не повезло, конкуренты выходят на рынок первыми.

Если мы вычертим график зависимости рисков от времени с начала разработки, как на рис. 5.1, обнаружится, что при итеративной разработке уменьшение опасных рисков начинается с самых ранних итераций. К тому времени, когда работа доходит до фазы построения, действительно опасных рисков остается немного, и деятельность разработчиков протекает без особых эксцессов. С другой стороны, при использовании водопадной модели серьезные риски никак не обрабатываются до наступления времени «большого взрыва» на этапе интеграции кода. На рис. 5.1 пунктирной линией показано, что при водопадной разработке наиболее опасные

риски сохраняются до момента интеграции и тестирования системы. Итерации, указанные в нижней части рисунка, имеют смысл только для итеративной и инкрементной разработки.



Рис. 5.1. Серьезные риски при итеративной разработке определяются и уменьшаются раньше, чем при водопадной

Согласно Каперсу Джонсу [51], примерно в двух третях серьезных проектов разработки программного обеспечения разработчики оказываются не в состоянии адекватно оценить риски. Вот место для улучшений! Но ранний удар по рискам — это лишь первый шаг.

Получение устойчивой архитектуры

Получение устойчивой архитектуры — отдельный результат итераций ранних фаз. В фазе анализа и определения требований, например, мы ищем базовую архитектуру, которая удовлетворяла бы ключевым требованиям, устранила критические риски и решала бы основные проблемы разработки. В фазе проектирования мы создаем базовый уровень архитектуры, который направляет дальнейшую разработку.

С одной стороны, капиталовложения во время этих фаз еще малы, и мы можем позволить себе проводить итерации до тех пор, пока не уверимся, что архитектура устойчива. После первой итерации фазы проектирования, например, мы можем сделать первоначальную оценку архитектуры. В этот момент мы еще можем позволить себе изменить архитектуру, если обнаружится необходимость в дополнительных вариантах использования или нефункциональных требованиях.

В случае водопадного подхода, к тому времени, когда мы обнаруживаем необходимость в изменении архитектуры, в разработку уже вложено так много средств, что внесение изменений в архитектуру приведет к значительным финансовым потерям. Кроме того, близится время сдачи системы. Попав в ловушку между ценой и графиком поставки, мы не имеем мотивации для внесения в архитектуру сколько-нибудь серьезных изменений. Сосредоточившись на архитектуре в ходе фазы

проектирования, мы уходим от этой проблемы. Мы стабилизируем архитектуру в виде базового уровня в начале жизненного цикла, когда затраты еще невысоки и времени впереди много.

Поддержка изменяющихся требований

Пользователи могут разобраться в работающей системе, даже если она еще несовершена, гораздо быстрее, чем в системе, существующей только в виде сотен страниц документации. Они также с трудом замечают прогресс в проекте, если все, что они видят, — это бумаги. Поэтому с точки зрения пользователей и других заинтересованных лиц более продуктивно создавать продукт в виде ряда исполняемых выпусков, или билдов, чем предлагать груды малопонятной документации. Билд — это рабочая версия части системы, которая демонстрирует подмножество функций системы. Каждая итерация может осуществляться в виде серии билдов, приближающих получение запланированного результата — приращения.

Наличие частично функционирующей системы уже на ранней фазе разработки позволяет пользователям и другим заинтересованным лицам вносить предложения и указывать требования, которые ранее, возможно, были пропущены. Бюджет и график еще не окаменели, так что разработчики могут относительно легко приспособиться к изменениям. В однопроходной водопадной модели пользователи не видят действующей системы до начала интеграции и тестирования. К этому моменту изменения, даже те, которые заслуживают внимания или кажутся небольшими, почти неизбежно выбивают проект из бюджета и графика. Таким образом, итеративный жизненный цикл облегчает заказчикам понимание потребности в добавлении или изменении требований, а разработчикам — учет потребностей заказчиков. В конце концов, они строят систему путем ряда итераций, и ответ на просьбы пользователей или включение в проект пересмотренных требований — это лишь изменение одной итерации.

Доступность тактических изменений

Используя итеративный и инкрементный подход, разработчики могут решать проблемы и вскрывать темные места на этапе ранних билдов и почти сразу вносить исправляющие их изменения. При таком подходе обнаруживаемые проблемы изливаются на разработчиков постоянной слабой струйкой, а это разработчикам выдержать легко. Поток сообщений о неисправностях, которые выплескивает на разработчиков «большой взрыв» при интеграции в ходе водопадной разработки, часто разрушает развитие проекта. Если разрушения значительны, проект может просто встать. Нагрузка сбивает разработчиков с ног, менеджеры проектов бегают по кругу, а остальные руководители паникуют. В отличие от такого варианта развития событий, серия билдов дает каждому ощущение выполнения работы.

Тестеры, технические писатели, создатели утилит, сотрудники, занимающиеся конфигурированием, и служба контроля качества могут согласовывать свои планы с выполнением графика проекта. Они узнают о существовании серьезных задержек еще в начале проекта, когда разработчики первыми сталкиваются с проблемами, которые дойдут и до этих вспомогательных служб. У них есть время скорректировать свои собственные планы. Если проблемы до начала тестирования ни-

как не проявляются, то в момент, когда их все же обнаружат, для аккуратного изменения планов уже не остается времени.

Когда служба контроля качества проверит итерацию, менеджеры проектов, архитекторы и другие заинтересованные лица могут оценить результат согласно предопределенным критериям. Они могут решить, привела ли итерация к верному приращению и были ли риски обработаны правильным образом. Эта оценка позволяет менеджерам определить успешность итерации. Если она признана успешной, руководство может разрешить начинать следующую итерацию. Если она была успешной лишь частично, они могут продолжить работу над этой итерацией или перенести нерешенные проблемы и необходимые переделки на следующую итерацию. В чрезвычайном случае, если оценка полностью отрицательна, они могут закрыть весь проект.

Достижение постоянной целостности

При завершении каждой итерации команда разработчиков демонстрирует уменьшение некоторых рисков. С каждой итерацией появляются новые функциональные возможности. Видно, что проект прогрессирует. Это понятно всем заинтересованным лицам.

Частые билды направляют разработчиков на регулярное подведение итогов — итогов в форме частей работающей программы. Опыт работы показывает, что поддержка программы в состоянии «закончено на 90%» — трудное дело для них или кого-либо еще. Это соотношение возникает, когда, судя по количеству кода или других **артефактов** (приложение B), продукт почти закончен. В отсутствие работающих билдов может, однако, оказаться, что наиболее трудная работа еще впереди. Проблемы могут и не обнаружиться до момента интеграции и тестирования системы. Напротив, последовательная серия итераций дает нам последовательность билдов, которые точно показывают состояние проекта.

Даже если разработчики будут не в состоянии достичь запланированного результата на ранних итерациях, у них еще будет время для новых попыток и улучшения модели в ходе последующих внутренних выпусков. Так как они занимаются сначала критическими вопросами, у них будет потом еще несколько возможностей улучшить свои решения.

Толстая линия на рис. 5.2 (первоначально представленном в [58]) иллюстрирует ход итеративной разработки. На первых же точках графика, сразу после написания от 2 до 4% кода, создается приращение (или билд). В этом месте проявляются небольшие проблемы, показанные небольшим понижением линии развития, но они быстро преодолеваются и процесс продолжает прогрессировать. И далее процесс разработки часто прерывается билдами. Каждый из них может привести к временному сбою в поступательном развитии. Но поскольку итерации относительно малы по сравнению с уровнем итоговой линии готового продукта (на нижней линии), восстановление происходит быстро. При водопадной разработке (тонкая линия) разработчики не начинают реализацию до окончания определения требований, анализа и проектирования. Они рапортуют об успешном продвижении реализации, потому что у них нет никаких промежуточных билдов, которые убедили бы их в обратном. Проблемы фактически скрыты, пока интеграция и испытание внезапно не выявляют их (поздно выявленные ошибки проектирования).



Рис. 5.2. В итеративной разработке частые билды не просто раньше выявляют проблемы, но также выявляют их малыми порциями, с которыми легче работать

Как показывает диаграмма, в водопадном подходе единственная интеграция незадолго до даты поставки выявляет множество проблем. Большой объем проблем и неизбежная в это время спешность подразумевают, что многие исправления не продумываются как следует. Уменьшение проблем и их исправление часто вызывают задержку поставки и срыв запланированных сроков. Следовательно, итеративная разработка заканчивается гораздо раньше, чем водопадная. Кроме того, продукт водопадной разработки может оказаться хрупким, то есть сложным в смысле его поддержки.

Достижение легкой обучаемости

После пары итераций каждый в команде разработчиков имеет хорошее представление о том, что входит в каждый из рабочих процессов. Они знают, что появляется в результате определения требований, а что — в результате анализа. Риск «аналитического паралича» (слишком много времени тратится на анализ) сильно уменьшается.

Кроме того, легче обучить новых сотрудников, потому что их можно учить прямо в ходе работы. Не нужно проектировать специальных пилотных версий только для того, чтобы помочь людям понять процесс. Они могут тренироваться на задачах, непосредственно связанных с работой. Учитывая, что они имеют соответствующее образование и работают с людьми, уже имеющими подобный опыт, они быстро набирают скорость. Если новые люди будут не в состоянии понять какой-то момент или сделают ошибку, эта ошибка не будет критичной для проекта, поскольку выявится во время построения первого же билда.

Итеративный подход также помогает проекту разобраться с рисками нетехнического характера, таких как организационные риски. Например, разработчики не могут достаточно быстро научиться тому, как:

- строить приложения на базе брокера объектных запросов;
- использовать инструменты для тестирования или **управления конфигурацией** (приложение В);
- работать в соответствии с процессом разработки программного обеспечения.

Пока происходит разработка, маленькая команда успевает познакомиться с этими новыми технологиями, инструментами и процессами. В ходе последующих итераций команда продолжает использовать все эти новинки, и к сотрудникам приходит мастерство. В то время как проект проходит итерации, команда постепенно растет. Начав работу группой из, возможно, 5–10 человек, команда разработчиков затем разрастается до 25, а к основной части разработки до примерно 100 человек. Команда растет шаг за шагом, и ядро команды в состоянии обучать новых ее членов по мере того, как они включаются в работу. Итеративный подход позволяет начальной команде наметить мелодию процесса и настроить инструменты до того, как к ним присоединится большинство разработчиков.

Проходя фазы и итерации, разработчики способны лучше выполнить требования реальных клиентов и уменьшить риски. По мере построения приращений все заинтересованные в этом лица могут наблюдать уровень прогресса. Снижая количество откладываемых на потом проблем, разработчики сокращают время выхода продукта на рынок. Кроме того, такой итеративный подход выгоден не только разработчикам и, в конечном счете, пользователям, но и руководству. Менеджеры могут ощущать реальный прогресс, отмечая законченные итерации.

Итеративный подход — управляемый рисками

Риск — это проектный фактор, который подвергает проект опасности. Это «вероятность того, что проект может столкнуться с нежелательной ситуацией, такой, как отставание от плана, перерасход средств или прекращение работ» (см. глоссарий в [51]).

Мы идентифицируем, расставляем по приоритетам и выполняем итерации, основываясь на рисках и их опасности. Это так, когда мы реализуем новые технологии. Когда мы выполняем требования клиентов, не важно, функциональные они или нет. Когда на ранних стадиях мы создаем устойчивую архитектуру, то имеем в виду такую, которая может приспосабливаться к изменениям с малым риском того, что появится необходимость что-либо перепроектировать. Да, мы организуем итерации для того, чтобы уменьшить риски. Другие серьезные риски связаны с вопросами производительности (скорость, мощность, точность), надежности, доступности, целостности системных интерфейсов, адаптируемости и переносимости (приложение В). Многие из этих рисков не могут быть обнаружены до реализации и начала тестирования программного обеспечения, которое осуществляет основные функции системы. Именно поэтому уже в фазах анализа и определения требований и проектирования следует предусмотреть итерации реализации и тестирования, в ходе которых изучались бы риски. Наша цель — определить риски на ранних итерациях.

Интересно отметить, что в принципе все технические риски можно спроектировать на вариант использования или сценарий варианта использования. В данном случае «спроектировать» означает, что после реализации варианта использования с его функциональными и нефункциональными требованиями риск будет снижен. Это верно не только для тех рисков, которые относятся к требованиям и архитектуре, но и для рисков, связанных с базовыми программными и аппаратными средствами. Тщательно выбирая варианты использования, мы можем осуществить все функции базовой архитектуры.

Снижение рисков — это основная задача итераций, которые мы делаем в фазах анализа и определения требований и проектирования. Позже, в фазе построения, риски в основном уже уменьшены до приемлемого уровня, а значит, на первый план выходят стандартные методы разработки. Мы стараемся планировать итерации так, чтобы каждая следующая итерация была основана на предыдущей. В этой фазе мы стараемся, в частности, уйти от опасности того, что мы не сможем правильно определить порядок итераций, и нам придется переделывать несколько предыдущих итераций.

Итерации снижают технические риски

Риски можно классифицировать различным образом [6, 51]. Однако нам достаточно будет описать варианты классификации лишь частично. Мы выделим четыре крупные категории рисков.

1. Риски, связанные с новыми технологиями.
 - Процессы могут быть распределены по многим узлам, что, вероятно, вызовет проблемы с синхронизацией.
 - Некоторые варианты использования могут зависеть от вычислительных методов, которые еще не до конца отработаны, например распознавание естественного языка или использование web-технологий.
2. Риски, связанные с архитектурой. Эти риски настолько важны, что мы разработали Унифицированный процесс, чтобы получить для работы с ними стандартный инструмент. Фаза проектирования и архитектурные итерации в ее пределах представляют собой то конкретное место в процессе разработки, в котором происходит работа с архитектурными рисками. Заранее создавая приспособленную к рискам архитектуру, мы устранием риска невозможности легкого приспособления к изменениям. Мы ликвидируем риск того, что впоследствии нам придется переделывать большую часть работы. Такая защищенная от рисков архитектура является устойчивой. Легкость внесения изменений вообще характерна для архитектурной **устойчивости** (приложение В). Другим преимуществом раннего получения устойчивой архитектуры является наглядность понимания того, в каких местах системы могут быть использованы компоненты многократного использования. Это позволяет нам уже на ранних этапах проекта задумываться о покупке части компонентов вместо того, чтобы создавать их своими силами. Также это уменьшает риск слишком поздно обнаружить, что построение системы обойдется чрезмерно дорого. Например.
 - Варианты использования, которые мы первоначально выберем, не смогут помочь нам определить структуру подсистемы, когда потребуется включать

в систему варианты использования, обнаруженные впоследствии. На ранних итерациях, например в ходе фазы проектирования, мы можем не обратить внимания на то, что один и тот же вариант использования через разные интерфейсы будут использовать несколько актантов. Пример такой ситуации -- несколько интерфейсов для снятия денег со счета. Один использует графический интерфейс пользователя и персональный компьютер; другой работает по коммуникационному протоколу по сети. Если мы намерены ограничиться только одним из этих вариантов использования, мы можем ограничиться соответствующей архитектурой, которая не имеет внутренних интерфейсов, позволяющих добавлять новые виды взаимодействий. Риск состоит в том, что такую систему будет трудно расширять.

- Некоторые каркасы (приложение В), которые планируется повторно использовать, на деле еще не использовались вне первоначального проекта, в ходе которого были построены. Риск состоит в том, что такой каркас не будет правильно работать с другими каркасами или что его будет нелегко использовать повторно.
 - Новая версия операционной системы, которую мы планируем использовать, возможно, еще не настолько отлажена, насколько нам хотелось бы. Риск состоит в том, что нам, вероятно, придется задержать выпуск нашей программы на некоторый срок, в течение которого мы будем ожидать выпуска производителем заплат и обновлений к операционной системе.
3. Риски, связанные с созданием правильной системы, такой, которая выполняла бы задачу и поддерживала работу пользователей. Эти риски подчеркивают важность определения требований как функциональных, так и нефункциональных, то есть, по существу, верного определения вариантов использования и построения пользовательских интерфейсов. Важно как можно раньше определить наиболее значимые функции и как можно раньше убедиться в том, что они осуществлены. Для снижения этих рисков мы реализуем варианты использования в порядке их важности для удовлетворения потребностей клиентов и вопросов производительности. Мы рассматриваем как поведение, так и свойства, такие, как производительность. Когда мы выбираем порядок реализации вариантов использования, то основываемся на возможности проявления рисков, например на возможности возникновения проблем с производительностью выполнения варианта использования. Между некоторыми требованиями (и вариантами использования, которые их отображают) и рисками, заложенными в них, существует корреляция. Особенно это верно для фаз анализа и определения требований и проектирования. Варианты использования, которые выбирает команда, действуют на разрабатываемую архитектуру. Например,
- Вариант использования *Следуй за мной* позволяет подписавшемуся владельцу телефона переадресовывать звонки на другой номер. Эта переадресация должна выполняться для всех звонков? Как насчет звонка будильника? В этот момент подписчик будет, вероятно, находиться у своего базового аппарата и не захочет, чтобы его будили в другом месте.
4. Некоторые риски связаны с производительностью. Например,
- Время отклика для варианта использования должно быть меньше 1 секунды.

- Число одновременно выполняемых экземпляров варианта использования больше 10000 в час.

Идентификация проблемных областей, таких как описанные выше, в значительной степени зависит от наличия людей с богатым опытом. Поскольку, по всей вероятности, ни один человек не будет иметь всего необходимого опыта, разрабатываемую систему станет изучать множество людей. Делайте списки возможных проблем и собирайтесь вместе на собрания по идентификации рисков. Эти собрания не предназначены для решения проблем, на них следует лишь определить их и расположить в порядке приоритета, в соответствии с которым они будут в дальнейшем изучаться в ходе итераций фаз анализа и определения требований и проектирования.

За нетехнические риски отвечает руководство

К нетехническим рискам относятся те, с которыми может столкнуться и которые вынуждено будет обходить руководство проекта. Примеры этой категории рисков.

- Организация в данный момент испытывает недостаток людей с опытом работы в некоторых специфических частях заявленного проекта.
- Организация собирается создавать части заявленной системы на новом для нее языке.
- При возникновении каких-либо проблем на любом из этапов разработки времени, отводимого клиентом на разработку, будет не хватать.
- Организация сможет выполнить заявленный график только в том случае, если субподрядчики, опыта работы с которыми у организации нет, поставят некоторые подсистемы в срок.
- Клиент может быть не в состоянии подписать акт о приемке в пределах срока, оставшегося до даты поставки.

Риски этого типа лежат за пределами круга вопросов, рассматриваемого в этой книге. Ограничимся следующей информацией: организация, занимающаяся разработкой программного обеспечения, должна идентифицировать их, предпринимать административные меры для отслеживания событий в каждой области риска и гарантировать, что в случае, если один из рисков осуществляется, ответственные за это руководители примут соответствующие меры.

Работа с рисками

Как только риски будут идентифицированы и расставлены в соответствии с приоритетом, команда решает, что делать с каждым из них. По существу у команды есть четыре возможности — уйти от риска, ограничить его, снизить или контролировать.

- От некоторых рисков можно и нужно уходить, возможно, перепланировав проект или изменив требования.
- Другие риски следует ограничить, сконцентрировав их таким образом, чтобы они затрагивали лишь небольшую часть проекта или системы.

- Некоторые риски можно смягчить, пробуя сделать рискованную часть проекта и наблюдая, оправдался риск или нет. Если риск привел к сбою, команда лучше узнает его и сможет получить возможность уйти от него, ограничить или контролировать этот риск.
- Некоторые риски, однако, невозможно смягчить. Команда может только контролировать их и наблюдать, не материализуются ли они. Если один из них материализовался, команда должна следовать имеющимся у нее **планам на случай неожиданностей** (приложение В). Если материализовался риск уровня «убийца проектов», мы должны глубоко вздохнуть и оценить ситуацию. Мы собираемся продолжать работу или должны отказаться от проекта? К этому моменту мы потратили лишь немного времени и денег. Мы знали о возможности наступления «убийцы проектов», и именно поэтому делали ранние итерации. Так что мы сделали полезную работу, обнаружив риск такой величины перед тем, как вовлекать в проект всех разработчиков.

Работа с рисками требует времени. Уход от риска или его ограничение требуют перепланирования или переделки. Смягчение риска может потребовать создания каких-то пробных элементов для проверки актуальности риска. Контроль риска включает в себя выбор механизма контроля, его проектирование и создание. Сам процесс смягчения или контроля рисков также требует усилий, то есть времени. А поскольку работа с рисками требует времени, организация редко может обработать все риски одновременно. Именно поэтому необходимо определение приоритетности итераций. Это именно то, что мы подразумеваем под управляемой рисками итеративной разработкой. Это и называется управлением рисками.

Обобщенная итерация

Как мы могли заметить, итерации на разных фазах цикла разработки сильно отличаются друг от друга. Это происходит потому, что различны задачи, с которыми разработчики сталкиваются на разных фазах разработки. В этом пункте мы намерены представить концепцию итерации на базовом уровне: что это такое, как ее планировать и каким должен быть результат итерации. В части 3 мы рассмотрим итерации каждой из этих четырех фаз в отдельных главах.

Что такое итерация?

Итерация — это мини-проект, более или менее полный проход по всем базовым рабочим процессам, заканчивающийся внутренним выпуском. Существует интуитивное понимание того, что такое итерация. Однако чтобы иметь возможность не поверхностно описывать работу, происходящую внутри итерации, мы расширим это определение.

Мы можем думать об итерации как о рабочем процессе, то есть рассматривать его как кооперацию между **сотрудниками** (приложение В), использующими и производящими артефакты. В Унифицированном процессе мы делаем различие между **основными рабочими процессами** и **рабочими процессами итераций** (приложение В). К настоящему времени нам известны пять основных рабочих процес-

сов: определение требований, анализ, проектирование, реализация и тестирование. Мы используем эти основные рабочие процессы только из педагогических соображений — они помогут нам описать рабочие процессы итераций. Таким образом, ничего волшебного в наборе основных рабочих процессов нет, и мы могли бы легко использовать другой их набор, например, объединить вместе анализ и проектирование¹. Мы используем именно такой набор, чтобы упростить описание более конкретных рабочих процессов, как абстрактный класс помогает нам описывать конкретные классы. Эти более конкретные рабочие процессы и есть рабочие процессы итераций. Мы подробно описываем основные рабочие процессы в главах 6–11, а рабочие процессы итераций с использованием основных рабочих процессов — в главах 12–16.

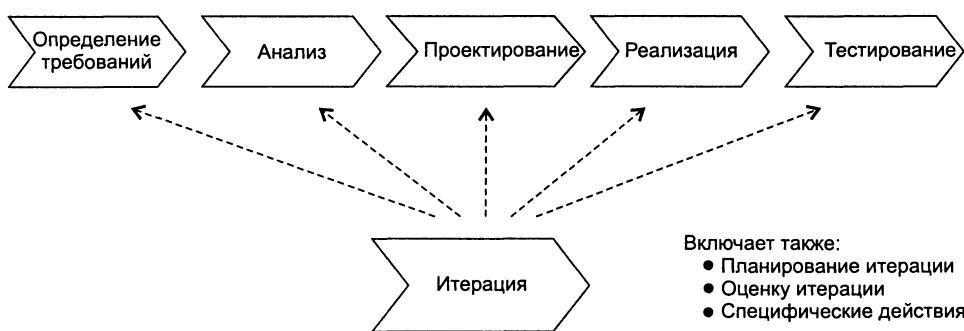


Рис. 5.3. Каждая итерация подразумевает осуществление пяти основных рабочих процессов. Она начинается с планирования и заканчивается оценкой результата

На рисунке 5.3 рассматриваются общие элементы каждого рабочего процесса итерации. Все они последовательно осуществляют пять основных рабочих процессов. Все начинаются с работ по планированию и заканчиваются оценкой. В части 3 мы описываем четыре типичных итерации, по одной для каждой фазы Унифицированного процесса. В каждой из них вновь происходит описание рабочих процессов и в каждой — по разному.

Чем это отличается от традиционной модели водопада? Каждый из основных рабочих процессов — это кооперация между группой сотрудников и набором артефактов. Однако между итерациями существует перекрытие. Сотрудники и артефакты могут участвовать более чем в одном основном рабочем процессе. Например, инженер по компонентам участвует в трех рабочих процессах — анализе, проектировании и выполнении. Наконец, рабочий процесс итерации создается путем наложения выбранного подмножества основных рабочих процессов друг на друга. Это дополнительное действие, вроде планирования и оценки.

В ходе ранних итераций разработчики сосредоточены на понимании проблемы и технологии. В фазе анализа и определения требований итерации сконцентрированы на создание бизнес-плана². В фазе проектирования итерации направлены на

¹ Рабочие процессы не следует путать с параллельными процессами. Рабочие процессы — это кооперации, используемые для построения описаний.

² В фазе анализа и определения требований в ходе изучения специфических проблем технологии итерация может следовать упрощенному варианту рабочих процессов.

создание базового уровня архитектуры. В фазе построения — посвящены созданию продукта в виде серии билдов, являющихся итогом каждой итерации. Эта деятельность завершается выпуском изделия, которое можно передавать сообществу пользователей. Однако каждая итерация происходит по тому самому шаблону, который показан на рис. 5.3.

После завершения каждая итерация оценивается. Одна из целей оценки состоит в том, чтобы определить, не появились ли новые или не изменились ли уже существующие требования так, что это затронет последующие итерации. При планировании тонкостей следующей итерации команда также смотрит, как повлияют на продолжающуюся работу оставшиеся риски.

На одну функцию следует в этом месте сделать особый упор. Это регрессионное тестирование (приложение В). Перед окончанием итерации мы должны гарантировать, что не разрушили ни одну из частей системы, которая после предыдущих итераций работала. Регрессионное тестирование особенно важно для итеративного и инкрементного жизненного цикла, поскольку каждая итерация вносит существенное дополнение к результатам предыдущей итерации и значительно изменяет состав и содержание артефактов проекта. Обратите внимание, что продлевать такое массированное регрессионное тестирование — после каждой итерации — без соответствующих тестовых утилит было бы непрактично.

Менеджеры проектов не должны соглашаться начинать следующую итерацию, если цели предыдущей итерации не достигнуты. Если же им все же приходится это сделать, план следует изменить, приспосабливаясь к новой ситуации.

Планирование итераций

Естественно, итеративный жизненный цикл требует более серьезного планирования и больших размышлений, чем водопадный подход. В водопадной модели все планирование происходит в начале работы, часто до уменьшения рисков и построения архитектуры. Полученный план страдает сильными неточностями. С другой стороны, при итеративном подходе мы не пытаемся спланировать подробно весь проект на фазе анализа и определения требований, мы планируем лишь первые шаги. Только когда в ходе фазы проектирования определяется фактическая база, команда проекта пытается спланировать фазы построения и внедрения. Конечно, в течение первых двух фаз существует рабочий план, но он не слишком детализирован.

Обычно (исключая самое начало проекта) в ходе планирования рассматриваются результаты предшествующих итераций, выбираются варианты использования, которые уместно будет рассмотреть в новой итерации, проверяется текущий статус рисков, которые будут важны для следующей итерации, и просматривается состояние последней версии набора моделей. Итерация заканчивается подготовкой внутреннего выпуска.

В конце фазы проектирования наконец создается база для планирования остальной части проекта и создания детального плана каждой итерации фазы построения. План первой итерации будет ясен. Более поздние итерации будут спланированы с меньшей детализацией и потребуют модификации, основанной на результатах и знании, полученных в ходе более ранних итераций. Точно то же самое произойдет и с планами на фазу внедрения, но эти планы, вероятно, придется изме-

нить в свете того, что команда узнает по результатам итераций фазы построения. Такое планирование позволяет управлять итеративной разработкой.

Последовательность итераций

В природе эволюция происходит без предварительного планирования. С итерациями при разработке программного обеспечения дело обстоит иначе. Варианты использования устанавливают цель. Архитектура устанавливает образец. Помня эту цель и образец, разработчики планируют последовательность своей работы по разработке продукта.

Планировщики стараются сделать так, чтобы итерации выстраивались по порядку, то есть чтобы ранние итерации обеспечивали базу для более поздних. Ранние итерации проекта приносят улучшенное понимание требований, проблем, рисков и предметной области, в то время как поздние итерации завершаются созданием приращений, которые в конечном счете складываются во внешний выпуск, то есть заказанный клиентом продукт. Для планировщиков полным успехом является создание такой последовательности итераций, которая никогда не требует возврата к пройденному для исправления модели в соответствии с результатами более позднего повторения.

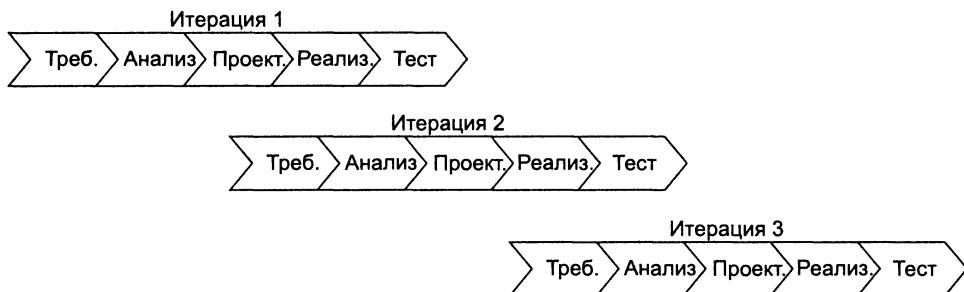


Рис. 5.4. Итерации включают в себя рабочие процессы от определения требований до тестирования

Итерации могут накладываться друг на друга в том смысле, что когда одна итерация заканчивается, следующая может уже начинаться, как показано на рис. 5.4. Планирование и другие ранние работы следующей итерации могут уже начаться, поскольку мы завершили предыдущую и готовим ее к выпуску. Однако мы не можем слишком сильно перехлестывать итерации, потому что предыдущая из них всегда является базой для следующей. Помните: конец итерации означает, что команда разработчиков закончила некий этап работы. Все программное обеспечение после итерации было собрано воедино и может быть выпущено в виде выпуска для внутреннего употребления.

Порядок, в котором мы располагаем итерации, в значительной степени зависит от технических факторов. Наиболее важная цель, однако, расположить итерации так, чтобы особенно трудные работы, те, которые требуют использования новых технологий, вариантов использования и архитектуры, могли быть сделаны как можно раньше.

Результат итерации — приращение

Приращение — это разница между внутренним выпуском предыдущей и следующей итераций.

К концу итерации набор представляющих систему моделей находится в специфическом состоянии. Это состояние, или статус, называется базовым уровнем. Каждая модель приходит к базовому уровню, каждый существенный элемент находится в состоянии базового уровня. Например, модель вариантов использования в конце каждой итерации содержит набор вариантов использования, демонстрирующий степень осуществления требований на этой итерации. Некоторые из вариантов использования в этом наборе завершены, в то время как другие завершены только частично. В то же время достигла базового уровня и модель проектирования, будучи согласованной с моделью вариантов использования¹. Подсистемы модели проектирования, интерфейсы и реализации вариантов использования также достигли базового уровня, будучи взаимно согласованными. Чтобы эффективно работать с множественными базовыми уровнями проекта, организация должна поддерживать согласованные и совместимые версии всех артефактов базового уровня. При итеративной разработке мы не можем преувеличить потребность в эффективных инструментах управления конфигураций.

В любой точке последовательности итераций некоторые подсистемы являются завершенными. Они содержат всю предписанную функциональность, были реализованы и оттестированы. Другие подсистемы завершены только частично, а третьи вообще пусты и вместо них для работы и интеграции с другими подсистемами используются заглушки. Таким образом, термин следует уточнить. Приращение — это разность между двумя последовательными базовыми уровнями.

В течение фазы проектирования, как мы уже отметили, мы создаем базовый уровень архитектуры. Мы определяем варианты использования, которые оказывают существенное воздействие на архитектуру. Мы рассматриваем эти варианты использования как кооперацию. Таким образом мы идентифицируем большинство подсистем и интерфейсов — или по крайней мере те из них, которые являются важными для архитектуры. Как только большинство подсистем и интерфейсов идентифицировано, мы конкретизируем наше знание, то есть пишем код, который их реализует. Частично эта работа будет сделана до выпуска базового уровня архитектуры и продолжится в ходе всех рабочих процессов. Однако большая часть детализации происходит в течение итераций фазы построения.

Когда мы подойдем к фазе внедрения, уровни согласованности между моделями и внутри моделей будут возрастать. Приращения итеративно детализируют модели, и интеграция в проект последнего приращения даст нам внешний выпуск системы.

Итерации в жизненном цикле программы

Как показано на рис. 5.5 [7], каждая из четырех фаз заканчивается главной вехой:

- анализ и определение требований: цели жизненного цикла;

¹ Не все варианты использования должны быть спроектированы, так что понятие «согласованы» относится только к вариантам использования, имеющим отражение в модели проектирования.

- проектирование: архитектура жизненного цикла;
- построение: возможность начать работу с продуктом;
- внедрение: внешний выпуск.

Цель каждой из главных вех — удостовериться, что модели различных рабочих процессов развиваются в ходе жизненного цикла сбалансированно. Под «сбалансированно» мы имеем в виду, что наиболее важные решения, включая те модели, которые имеют отношение к рискам, вариантам использования и архитектуре, создаются на ранних стадиях жизненного цикла. Затем работа должна направляться на увеличение детализации и на рост качества.

Основными целями фазы анализа и планирования требований являются определение задач создаваемого продукта, снижение наиболее опасных рисков и подготовка исходного бизнес-плана, показывающего, каких показателей может достичнуть проект по деловым параметрам. Другими словами, мы хотим установить цели жизненного цикла проекта.

Основными целями фазы проектирования являются создание базового уровня архитектуры, определение большинства требований и снижение рисков второго уровня, то есть определение архитектуры жизненного цикла. К концу этой стадии мы в состоянии оценить затраты и график работ и более-менее детально спланировать фазу построения. В этот момент мы должны быть в состоянии составить бизнес-предложение.

Вехи			
Веха целей жизненного цикла	Веха архитектуры жизненного цикла	Веха возможности начать работу	Веха внешнего выпуска
Анализ и определение требований	Проектирование	Построение	Внедрение
Итер.1	Итер.2	—	—
			Итер. №n-1 Итер. №n

Рис. 5.5. Фазы включают в себя итерации и оканчиваются главными вехами, в моменты которых руководство принимает важные решения (число итераций не является постоянным и различно для разных проектов)

Первичные цели фазы построения — создать законченную систему и гарантировать, что продукт можно начинать передавать заказчикам, то есть получить возможность начать работу с продуктом.

Первичные цели стадии внедрения — гарантировать, что мы имеем изделие, готовое к передаче сообществу пользователей. В ходе этой стадии развития пользователи обучаются использовать программное обеспечение.

В пределах каждой фазы существуют промежуточные вехи. Ими являются критерии завершения каждой из итераций. Каждая итерация производит результаты, артефакты моделей. Таким образом, в конце каждого из повторений создаются новые приращения модели варианта использования, модели анализа, модели про-

ектирования, модели развертывания, модели реализации и модели тестирования. Новые приращения будут объединены с результатами предыдущей итерации в новую версию набора моделей.

На промежуточных вехах руководство и разработчики, как мы говорили ранее, решают, как перейти к следующей итерации. На главных вехах в конце фаз руководство принимает критические решения типа продолжать/остановиться и определяет график, бюджет и требования.

Промежуточная веха (после внутреннего выпуска в конце итерации) — это запланированный шаг к главной вехе в конце фазы. Различие между главной вехой и промежуточной лежит в основном на уровне бизнеса. Разработчики последовательно разбираются с рисками и создают программные артефакты, пока не достигнут главной вехи. На каждой главной вехе руководство оценивает то, что сделали разработчики. Проход каждой главной вехи, таким образом, представляет собой важное деловое решение и обязывает руководство финансировать работу на (по крайней мере) следующую по плану фазу. Мы можем считать главные вехи точками, в которых решения руководства пересекаются с действиями исполнителей.

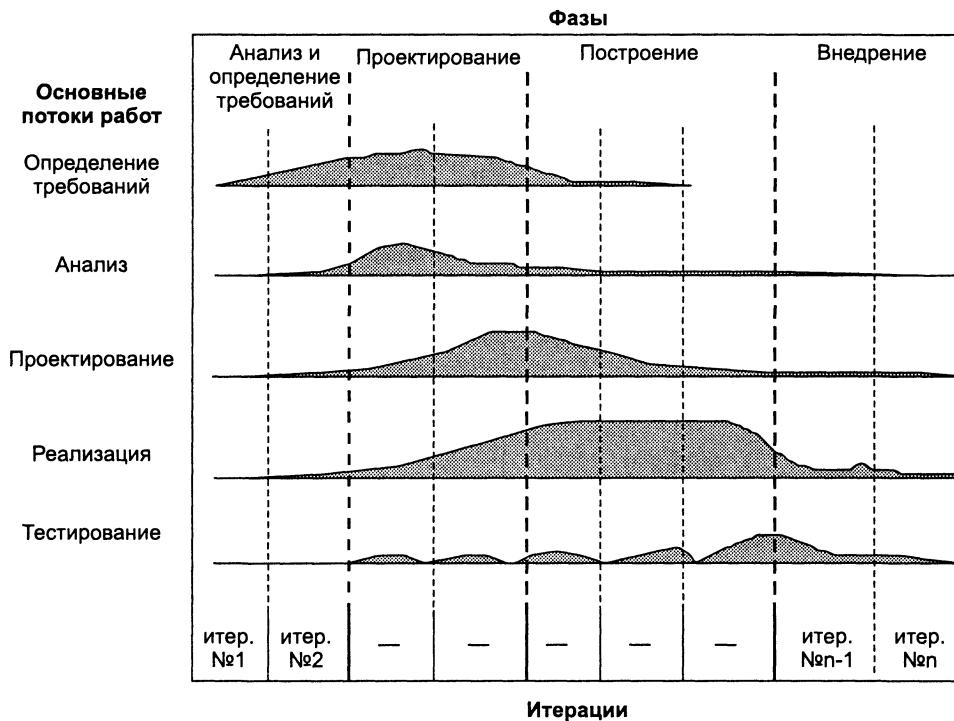


Рис. 5.6. Сдвиг важности в течение итераций с определения требований и анализа на проектирование, реализацию и тестирование

Такое разделение помогает руководству и другим вовлеченным в работу заинтересованным лицам сначала оценить то, что было сделано в ходе малобюджетных фаз анализа и определения требований и проектирования, а уже затем решать, начинать ли дорогостоящую фазу построения.

Проект разработки программного обеспечения можно грубо разделить на две части — фазы анализа и определения требований и проектирования, с одной стороны, и фазы построения и внедрения — с другой. В течение фаз анализа и определения требований и проектирования мы создаем бизнес-план, уменьшаем наиболее опасные риски, строим базовый уровень архитектуры и с высокой точностью планируем остальную часть проекта. Эту работу в состоянии сделать небольшая и не слишком дорогостоящая команда.

Затем проект переходит в фазу построения, где целью является создание продукта. Число людей, вовлеченных в проект, увеличивается. Они разрабатывают большую часть функциональности системы, основываясь на базовом уровне архитектуры, созданном в ходе фазы проектирования. Они в максимально возможной степени используют существующее программное обеспечение.

Совершаемые в ходе каждой итерации проходы по рабочим процессам определения требований, анализа, проектирования, реализации и тестирования имеют на разных фазах различную значимость, что и показано на рис. 5.6. В ходе фаз анализа и определения требований и проектирования большинство усилий направлено на определение требований и предварительный анализ и проектирование. В ходе построения важность смещается на детальное проектирование, реализацию и тестирование. Хотя это не показано на рис. 5.6, ранние фазы серьезно загружены управлением проектом и построением среды проектирования.

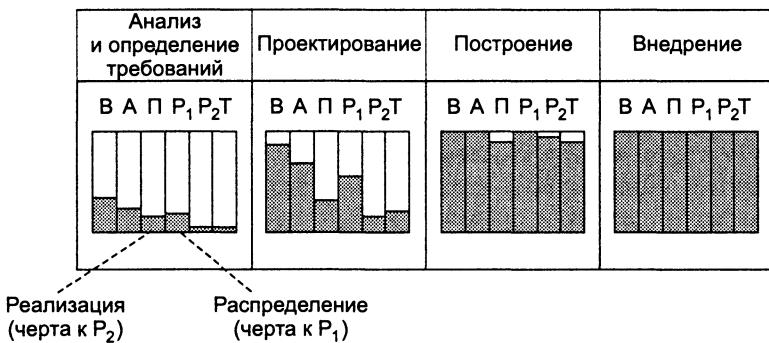


Рис. 5.7. Работа над всеми моделями продолжается в ходе всех фаз, что обозначено ростом заполнения соответствующих столбцов

Модели в ходе итераций совершенствуются

Итерации приращение за приращением строят итоговые модели. Каждая итерация добавляет понемногу к каждой модели, проходя через определение требований, анализ, проектирование, реализацию и тестирование. Некоторые из этих моделей, например модель вариантов использования, сильнее прорабатываются на ранних фазах. Другим же, например модели реализации, основное внимание уделяется в фазе построения. Это схематически показано на рис. 5.7. В фазе анализа и определения требований команда создает те части моделей, которые необходимы

мы для поддержки концептуального прототипа. В состав этого прототипа входят наиболее важные элементы модели вариантов использования (В), модели анализа (А) и модели проектирования (П), а также части моделей развертывания (P_1), реализации (P_2) и тестирования (Т). Большая часть исполняемого кода на этой фазе предварительна. Как показано на рис. 5.7, остается сделать еще много работы.

В фазе проектирования более темные области, обозначающие выполнение работ, весьма существенно увеличиваются. В конце этой фазы, однако, в то время как команда заканчивает около 80% модели вариантов использования (В) и модели развертывания (P_1), реализация (P_2) и тестирование (Т) функциональности закончены и включены в систему менее чем на 10%. Модели вариантов использования и развертывания после фазы разработки должны быть законченными. Иначе, не зная требований и предварительных условий (включая архитектуру) для реализации, мы не сможем точно спланировать фазу построения¹.

В фазе построения выполняется большая часть В, А, П, P_1 , P_2 и Т, чего, собственно, и следовало ожидать, так как критерий окончания фазы — завершение создания системы, которую можно начинать передавать пользователям. Позже, в ходе передачи системы в эксплуатацию в фазе внедрения, в нее будут внесены незначительные изменения и тонкая настройка.

Итерации проверяют организацию

Множество организаций, занимающихся разработкой программного обеспечения, имеют тенденцию переходить сразу к написанию кода, потому что строки кода — это то, что их менеджеры в состоянии сосчитать. Они имеют привычку сопротивляться изменениям, потому что изменения замедляют написание кода. Они не заинтересованы в многократном использовании анализа, проекта или кода, потому что их менеджеры получают премии за новый код.

Переход к итеративной разработке бросает вызов методам работы, принятым в этих организациях. Он требует изменения точки зрения. Внимание в организации должно передвинуться с подсчета строк кода к сокращению рисков и определению базового уровня архитектуры и функциональности. Менеджеры должны по-новому взглянуть на то, что они считают. Они должны будут показывать своими действиями, что они измеряют прогресс, — в понятиях адресованных рисков, проработанных вариантов использования и компонентов, реализующих эти варианты использования. Иначе разработчики скоро регрессируют к привычному образу действий — написанию строк кода.

Применение итеративного и инкрементного подхода имеет некоторые важные последствия.

- Чтобы создать бизнес-план в фазе анализа и определения требований, организация настойчиво снижает наиболее опасные риски и создает концептуальную версию системы.

¹ В главе 4 мы показывали, что модели вариантов использования и анализа в конце фазы проектирования имеют более низкий уровень завершенности, чем показано на рис. 5.7. Причина этого несоответствия в том, что в главе 4 мы сосредоточились исключительно на архитектуре и не рассматривали другую работу, которую тоже следовало делать (лучшее понимание вариантов использования, чтобы иметь возможность сделать бизнес-план).

- Чтобы сделать полноценное бизнес-предложение в конце фазы проектирования, организация должна знать, что, согласно контракту, предполагается построить (исходя из базового уровня архитектуры и требований) и обладать уверенностью, что в задании не осталось невыявленных рисков (недостаточно изученных моментов, способных увеличить стоимость или сорвать план).
- Чтобы минимизировать затраты, проблемы и время выхода на рынок, организация должна использовать многократно используемые компоненты (результат ранней проработки архитектуры, опирающийся на изучение предметной области, в которой должна работать предложенная система).
- Чтобы избежать задержки поставок, роста цены и создания низкокачественного продукта, организация должна «делать сначала сложные вещи».
- Чтобы избежать выпуска продукта, который опаздывает на рынок, организация не может больше упрямо говорить «нет» на все предложения о внесении изменений. Разбитый на фазы итеративный подход позволяет вносить изменения в течение всего времени разработки.

Итеративная и инкрементная разработка требует не только нового способа управлять проектами, но и утилит, которые помогут осуществлять этот новый способ. Фактически невозможно иметь дело со всеми артефактами системы, одновременно изменяемыми в каждом билде и каждом приращении, без поддержки утилит. Организация, выбирающая этот способ разработки, нуждается в различных утилитах для разных рабочих процессов наравне с утилитами для управления конфигурацией и контроля версий.

ЧАСТЬ 2

Основной рабочий процесс

Теперь, когда мы разобрались с базовыми понятиями, лежащими в основе ключевых действий Унифицированного процесса, мы подробно рассмотрим каждый из рабочих процессов. В части 3 будут описаны фазы и итерации процесса разработки.

В части 2 мы рассматриваем основные рабочие процессы — по одному в каждой главе. В главах 6 и 7 описана работа с требованиями, в главе 8 — анализ, в главе 9 — проектирование, в главе 10 — реализация и наконец в главе 11 — тестирование. Понятие «рабочий процесс» — это абстракция, подробно описанная в главе 5. В ходе итераций мы сосредоточиваемся на осуществлении рабочих процессов. Этой теме будет посвящена часть 3.

Описание каждого из основных рабочих процессов отдельно от других может ввести читателя в заблуждение. Во-первых, описывая основные рабочие процессы один за другим, мы создаем у читателя впечатление, что в процессе разработки программного обеспечения от начала проекта до его конца вся последовательность рабочих процессов проходит лишь единожды. Читатель может решить, что основные рабочие процессы достаточно проделать в ходе разработки по одному разу, как в старом водопадном процессе разработки. Кроме того, неподготовленный читатель может посчитать, что базовый рабочий процесс есть нечто монолитное и неделимое.

Ни одно из этих представлений не является соответствующим истине. Мы описываем базовые рабочие процессы в отдельных главах, исходя исключительно из методических соображений, для обеспечения полноты и четкости описания каждого рабочего процесса.

Первое замечание касается схожести с водопадной разработкой. Мы осуществляем последовательно пять рабочих процессов, но делаем это по одному разу на итерацию, а не единожды для процесса в целом. То есть, если у нас есть семь ите-

раций в четыре фазы каждая, мы можем проделать все рабочие процессы семь раз. Если вдуматься, мы не сможем использовать все пять рабочих процессов на ранних итерациях уже потому, что не дойдем до завершающих рабочих процессов, таких как реализация и тестирование, в ходе первых итераций. Принцип ясен: мы осуществляляем на каждой итерации те рабочие процессы, которые нужны нам на данной итерации.

Перейдем к замечанию о монолитности и неделимости рабочих процессов. Мы описываем каждый из основных рабочих процессов совершенно независимо от остальных. Они, однако, взаимодействуют между собой, создавая и используя артефакты друг друга. В этой части книги мы попытались немного упростить каждый из рабочих процессов, сосредоточившись на основных его действиях, вновь по причинам методического характера. Мы не стали описывать чередование действий основного рабочего процесса с действиями в других рабочих процессах. Эти чередования необходимы для итерационного процесса разработки программного обеспечения, и мы, несомненно, подробно их рассмотрим — но в третьей части книги.

В части 3 мы описываем, как рабочие процессы, описанные по отдельности, объединяются в рабочий проект. Например, фазе анализа и определения требований может соответствовать лишь часть рабочих процессов, в то время как в фазе конструирования будет необходим их полный набор.

6 Определение требований — от концепции к требованиям

Поколение за поколением некоторые племена американских индейцев строили каноэ из выдолбленных стволов деревьев. Строители каноэ начинали с поисков дерева диаметром в несколько футов, лежащего неподалеку от воды. Около него они разводили костер и клали горящие угли на бревно. Обугленную древесину было намного проще удалять теми каменными инструментами, которые были в их распоряжении. После того как через несколько дней выдалбливания каноэ, казалось бы, было полностью готово, строители выталкивали его на мелководье. Более чем вероятно, что первый вариант каноэ просто переворачивался. Каноэ не было сбалансировано. Следовало проделать еще много работы этими ужасными каменными инструментами, пока у строителей не получалась лодка, которая не опрокидывалась, когда кто-то наклонял ее, чтобы вытянуть рыбу из воды. Только после этого она считалась законченной. Это умение передавалось из поколения в поколение и врастало в кровь и плоть строителей.

Когда «племя» разработчиков программного обеспечения получает заказ на разработку новой системы, они оказываются в другом положении. Прежде всего, разработчики не являются будущими пользователями системы. Они не будут иметь немедленной и полной информации относительно того, как работает их «каноэ». Во вторых, они не занимались непрерывным изготовлением таких продуктов с раннего детства, и требования к системе и ограничения не вросли в их кровь и плоть. Вместо этого они должны понять, что от них требуется.

Мы называем этот акт постижения *определением требований*. Это процесс выяснения, что же должно быть создано. Этот процесс затруднителен. Фактически это настолько трудно, что для разработчиков все еще является обычным делом начать писать код (что довольно просто) раньше, чем они поймут, что этот код должен делать (что значительно сложнее).

Почему трудно определять требования

Профессиональные разработчики программного обеспечения обычно создают программы не для себя, а для других людей — пользователей программного обеспече-

ния. «Так, — говорят разработчики, — пользователи должны знать, что им нужно». Однако даже небольшой опыт получения требований от пользователей показывает, что они — не идеальный источник информации. Прежде всего, у любой системы обычно множество пользователей (или категорий пользователей), и в то время как каждый пользователь знает свою работу, никто не может увидеть картину в целом. Пользователи не знают, как работа в целом может быть сделана более эффективной. Большинство пользователей не знает, какие аспекты их работы могут быть переложены на программу. На самом деле пользователи часто не знают даже, что такое требования и как их точно изложить.

Традиционным подходом к этой проблеме было поручение выявления списка требований каждого пользователя посредникам — аналитикам — в надежде на то, что аналитик будет способен увидеть картину в целом и разработать полное, правильное и непротиворечивое техническое задание. Аналитики обычно записывают требования в документах размером в сотни, а иногда и тысячи страниц. Но абсурдно полагать, что человеческий мозг может создать полный, верный и непротиворечивый список из тысяч заявлений «система должна...». Что более важно, эти спецификации требований невозможно быстро преобразовать в проектные и рабочие спецификации.

Даже с помощью аналитиков пользователи не до конца понимали, что должна делать программная система, пока она не была почти закончена. Когда проект развивался, пользователи, посредники и сами разработчики начинали видеть, на что будет похожа система, и лучше понимать реальные потребности пользователей. Поступало множество предложений об изменениях. Многие из них действительно следовало бы сделать, но их выполнение обычно влекло за собой серьезное нарушение сроков и рост затрат.

Многие годы мы вбивали себе в голову, что пользователи знают, что такое требования, и что все, что мы должны сделать, — это расспросить их. Действительно, системы, которые мы создаем, должны быть удобны для пользователей, и мы можем изучать взаимодействие пользователя с программой, глядя на то, что делают пользователи. Однако значительно важнее то, что система должна выполнять ту задачу, для которой она создавалась. Например, система должна приносить результат, значимый для бизнеса, в котором она используется, и для клиентов системы, которые ее используют. Часто трудно понять, каким должен быть этот результат, а иногда невозможно создать систему, дающую такой результат. Более того, отражая меняющийся реальный мир, ценность этого результата сама может изменяться в ходе проекта: может измениться сам бизнес, могут изменяться технологии, при помощи которых формируется система, могут изменяться ресурсы (люди, деньги), при помощи которых формируется система и т. п.

Даже после этого озарения определение требований остается нелегким делом. В промышленности происходили длительные поиски хорошего, систематического процесса определения требований. Рассмотрением такого процесса мы и займемся в этой и следующей главах.

Цели процесса определения требований

Основная цель рабочего процесса определения требований состоит в том, чтобы направить процесс разработки на получение правильной системы. Это достигает-

ся описанием системных требований (то есть условий, которым должна соответствовать система). Описание требований должно быть достаточно хорошим для того, чтобы между клиентами (включая пользователей) и разработчиками могло быть достигнуто понимание по вопросу о том, что система должна делать и чего не должна.

Основное внимание следует обратить на то, что клиент, с которым мы работаем и для которого компьютеры не являются основной специальностью, должен быть способен прочесть и понять результаты определения требований. Чтобы выполнить это условие, мы должны использовать для описания результатов язык клиента. Как следствие, мы должны быть очень осторожны при включении в результаты определения требований формализмов, структуры и внутренних деталей работы программы.

Результаты рабочего процесса определения требований также помогают руководителю проекта планировать итерации и поставляемые клиентам выпуски (это обсуждается в части 3).

Обзор процесса определения требований

Каждый проект по созданию программного обеспечения уникален. Это следует из вариаций типов систем, клиентов, организаций, занимающихся разработкой, технологий и т. д. Также существуют и различные отправные точки для определения требований. В некоторых случаях мы начинаем с разработки бизнес-модели. Мы также можем получить готовую бизнес-модель, разработанную какой-нибудь другой организацией (см. подраздел «Что такое бизнес-модель?» данной главы). В других случаях программное обеспечение создается для встроенной системы, которая не имеет прямой связи с бизнесом. Здесь уместно было бы сначала построить простую модель объекта, например модель предметной области (см. подраздел «Что такое модель предметной области?»). В некоторых случаях клиент, возможно, уже разработал полное, детальное техническое задание, не основанное на объектной модели. В таком случае мы начинаем работу, используя в качестве исходного документа это техническое задание.

В других случаях клиенты могут иметь весьма неопределенное понятие о том, что должна делать их система, полученное, может быть, из концептуальных утверждений высшего руководства. Между этими крайними положениями лежит все многообразие вариантов. Мы рассмотрим в качестве отправной точки один из вариантов — «неопределенное понятие», для чего введем пример, который будем использовать в остальной части этой книги.

Пример. Консорциум *Interbank* обдумывает компьютерную систему. Консорциум InterBank, гипотетическое финансовое учреждение, стоит перед серьезными изменениями в связи с отменой государственного контроля, конкуренцией и возможностями, открываемыми WWW. Консорциум планирует создать новые приложения для работы на быстро меняющихся финансовых рынках. Это заставляет его филиал, занимающийся разработкой программного обеспечения, Interbank Software, приступить к разработке этих приложений.

Interbank Software решает разработать биллинговую и платежную систему в сотрудничестве с некоторыми из крупных клиентов банка. Для пересылки зака-

зов, счетов и платежей между покупателями и продавцами система будет использовать Интернет. Мотивацией для банка при разработке системы является привлечение новых клиентов. Новых клиентов должно привлекать предложение низких тарифов на проведение платежей. Банк будет также в состоянии уменьшить затраты на заработную плату, проводя платежные требования автоматически через Интернет вместо ручной обработки этих документов кассирами.

Мотивацией для покупателей и продавцов должно стать уменьшение затрат, документооборота и времени. Например, им больше не нужно будет посыпать заявки или выставлять счета обычной почтой. Оплата счетов будет происходить между компьютером покупателя и компьютером продавца. Покупателям и продавцам будет также удобнее просматривать информацию об их счетах и платежах.

Возможность существования таких различных отправных точек, как неопределенные концепции и детальные технические задания, предполагает, что аналитики должны быть способны приспособить свой подход к определению требований в любой ситуации.

Различные отправные точки влекут за собой различные типы рисков, так что аналитики должны выбрать подход, максимально снижающий риски. Уменьшение рисков подробно обсуждается в части 3.

Несмотря на различие отправных точек, существуют шаги, которые можно сделать в большинстве случаев. Это позволяет нам предложить типовой рабочий процесс. Такой рабочий процесс включает в себя следующие шаги, которые обычно не выполняются по отдельности

- Перечисление возможных требований.
- Осознание контекста системы.
- Определение функциональных требований.
- Определение нефункциональных требований.

Мы кратко опишем эти шаги в следующих пунктах.

Перечисление возможных требований. В ходе функционирования системы клиентам, пользователям, аналитикам и разработчикам приходит в голову множество хороших идей, которые могли бы превратиться в реальные требования. Мы храним список этих идей, рассматривая его как подборку кандидатов на реализацию в будущих версиях нашей системы. Этот список предложений растет, когда в него добавляются новые пункты, и сокращается, когда кандидаты становятся требованиями или преобразуются в другие артефакты, например, варианты использования. Список предложений используется исключительно для планирования работ.

Каждое предложение имеет короткое название и содержит краткое объяснение или определение, ровно столько информации, сколько необходимо, чтобы обеспечивалась возможность обсуждать его при планировании разработки продукта. Каждое предложение содержит также набор планируемых значений, в которые могут входить:

- состояние предложения (например, предложено, одобрено, включено в план, или утверждено);
- сметная себестоимость выполнения (в терминах типов ресурсов и человеко-часов);
- приоритет (например, критический, важный или вспомогательный);

- уровень риска, связанного с реализацией предложения (например, критический, значительный или обычный, см. главу 5).

Эти планируемые значения вместе с другими аспектами (см. главу 12) используются для того, чтобы оценить размер проекта и решить, как разбить проект на последовательные итерации. Так, например, приоритет и уровень рисков, связанные с предложением, используются, чтобы решить, во время какой итерации реализовывать данное предложение (это обсуждается в части 3). Кроме того, когда мы планируем реализацию предложения, этот факт должен быть отражен в одном или более вариантах использования или дополнительных требованиях (которые мы вскоре обсудим).

Осознание контекста системы. Множество людей, вовлеченных в развитие программного обеспечения, являются специалистами в вопросах, имеющих отношение к программному обеспечению. Однако чтобы верно определить требования и правильно сформировать систему, ключевые разработчики — в частности, архитектор и некоторые старшие аналитики — должны понимать контекст, в котором работает система.

Имеется по крайней мере два подхода к описанию контекста системы в форме, доступной для разработчиков программ: моделирование предметной области и бизнес-моделирование. Модель предметной области описывает важные понятия контекста как объекты предметной области. Предметная область при этом связывает эти объекты друг с другом. Идентификация и наименование этих объектов помогают нам разработать словарь терминов, который поможет каждому, кто работает над системой, лучше ее понимать. Впоследствии, когда мы будем проводить анализ и проектирование нашей системы, объекты предметной области помогут нам распознать некоторые классы. Как мы увидим далее, бизнес-модель может быть представлена как надмножество модели предметной области. Она содержит доменные объекты, и не только их.

Цель бизнес-моделирования состоит в описании процессов — существующих или воспринимаемых — для того, чтобы понять их. Бизнес-моделирование — единственная часть бизнес-инжиниринга, которую мы будем использовать в этой книге [39]. Удовлетворимся замечанием, что бизнес-инжиниринг очень похож на бизнес-моделирование, но имеет своей целью также улучшение бизнес-процессов организаций.

Когда аналитики моделируют бизнес, они получают обширную информацию о контексте программной системы и отражают ее в бизнес-модели. Бизнес-модель определяет, какие бизнес-процессы должна поддерживать система. Кроме идентификации вовлеченных в бизнес бизнес-объектов или объектов предметной области, бизнес-моделирование также устанавливает компетентности, необходимые для процессов: работников, их обязанности и действия, которые они должны выполнять. Это знание является определяющим при идентификации вариантов использования. Мы вскоре обсудим этот вопрос. Фактически подход бизнес-инжиниринга для определения требований при разработке бизнес-приложений является наиболее системным [39].

Архитектор и руководитель проекта совместно решают, ограничиться ли моделью предметной области или идти до конца и разрабатывать полную бизнес-модель, а может быть, не строить никакой модели вообще.

Определение функциональных требований. Прямой подход к выявлению системных требований основан на вариантах использования (варианты использования подробно рассматриваются в главе 7). Эти варианты использования охватывают как функциональные требования, так и те нефункциональные требования, которые специфичны для конкретных вариантов использования.

Опишем вкратце, как варианты использования помогают нам правильно определять требования. Каждый пользователь хочет, чтобы система выполняла для него какие-то действия, то есть осуществляла варианты использования. Для пользователя вариант использования — это способ, которым он взаимодействует с системой. Следовательно, если аналитики в состоянии описать все варианты использования, которые нужны пользователям, значит, они знают, что представляет из себя система с точки зрения функциональности.

Каждый вариант использования представляет собой один способ работы с системой (например, поддержку пользователя в ходе бизнес-процесса). Каждый пользователь нуждается в своих вариантах использования, каждый работает с системой по-своему. Чтобы определить варианты использования, которые необходимы в системе на самом деле, например, те, которые обслуживают бизнес или позволяют пользователям работать «удобным» для них способом, требуется, чтобы мы досконально знали потребности пользователя и клиента. Для этого мы должны разобраться в контексте системы, опрашивать пользователей, обсуждать предложения и т. д.

В дополнение к вариантам использования аналитики должны также определить, как должен выглядеть пользовательский интерфейс для реализации того или иного варианта использования. Самый лучший способ разрабатывать спецификацию пользовательского интерфейса — это набросать несколько версий представления информации, которую необходимо передать, обсудить эскизы с пользователями, сделать прототипы и отдать их пользователям на тестирование.

Определение нефункциональных требований. К нефункциональным требованиям относятся такие свойства системы, как ограничения среды и реализации, производительность, зависимость от платформы, ремонтопригодность, расширяемость, и надежность — все «-ости». Под надежностью понимаются такие характеристики, как пригодность, точность, средняя наработка на отказ, число ошибок на тысячу строк программы и число ошибок на класс. Требования по производительности налагают специфические условия на выполнение функций: скорость, пропускная способность, время отклика и используемая память. Большинство требований, связанных с производительностью, относятся лишь к нескольким вариантам использования и должны быть приписаны к этим вариантам использования как имеющие значения (приложение А). Практически это означает, что эти требования должны быть описаны в правильном контексте, то есть внутри вариантов использования (возможно, в отдельном разделе «Специальные требования»).

Пример. Специальные требования к варианту использования «Оплата счета».

Требования по производительности. Когда покупатель подает счет к оплате, система должна выдать результат проверки запроса не медленнее чем за 1.0 секунду в 90 % случаев. Время проверки никогда не должно превышать 10.0 секунд, если только связь не разорвана (в этом случае пользователь должен быть уведомлен о разрыве связи).

Некоторые нефункциональные требования относятся к реальным объектам, например банковским счетам в системе, предназначенной для банка. Эти требова-

ния могут первоначально определяться в соответствующем бизнес-объекте или объекте предметной области в модели системы.

Позднее, когда будут определены варианты использования и «понятия», на основе которых они функционируют, эти нефункциональные требования будут связаны с соответствующими понятиями. Под «понятием» мы понимаем или неформальную статью в гlosсарии, используемом для описания варианта использования (см. главу 7), или более формальный класс аналитической модели (см. главу 8). Для простоты в этой главе мы рассматриваем первый случай, то есть считаем, что требования связаны с понятиями из гlosсария.

В заключение заметим, что некоторые нефункциональные требования являются слишком обобщенными и не могут быть привязаны к конкретному варианту использования или конкретному реальному классу. Они должны быть вынесены в отдельный список **дополнительных требований** (приложение В).

Резюме. Чтобы эффективно определить требования, аналитики нуждаются в наборе методов и артефактов, которые помогали бы им получить удобное изображение системы, такое, чтобы оно способствовало осуществлению рабочих процессов. Мы будем называть все эти артефакты в целом *набором требований*. Традиционные технические задания прошлого сменил набор артефактов: модель вариантов использования и дополнительные требования. Эти артефакты требуют, чтобы контекст системы был представлен в виде бизнес-модели или модели предметной области (рис. 6.1). Отметим, что варианты использования также содержат нефункциональные требования, специфичные для этих вариантов использования.

Операция	Получаемый артефакт
Перечислить кандидатов в требования	Список предложений
Разобраться в контексте системы	Бизнес-модель или модель предметной области
Определить функциональные требования	Модель вариантов использования
Определить нефункциональные требования	Дополнительные требования или отдельные варианты использования (для требований, относящихся к отдельным вариантам использования)

Определяет традиционное техническое задание

Рис. 6.1. Набор требований состоит из различных артефактов, указанных в правом столбце. Каждая операция определения требований порождает один или более артефактов

Поскольку требования постоянно изменяются, мы нуждаемся в способе их обновления, позволяющем контролировать вносимые изменения. Это делается посредством итераций. Каждая итерация отражает некоторое изменение набора требований. Число вносимых изменений будет постепенно уменьшаться, поскольку мы постепенно войдем в фазу проектирования, и требования стабилизируются. Этот процесс описан в следующем подразделе. После этого мы сосредоточимся на

описании контекста системы моделью предметной области (подраздел «Понимание контекста системы с помощью модели предметной области») или бизнес-моделью (подраздел «Понимание контекста системы с помощью бизнес-модели»). В заключение мы рассмотрим дополнительные требования (раздел «Дополнительные требования»).

Определение требований в виде вариантов использования — более обширная тема, и мы вернемся к ней в главе 7.

Роль требований в жизненном цикле разработки программного обеспечения

Модель вариантов использования разрабатывается посредством нескольких приращений. Каждая итерация добавляет к модели новые варианты использования и добавляет новые подробности к описанию уже существующих.

Рисунок 6.2 дает понятие о том, как в ходе различных фаз проектирования и входящих в них итераций протекает рабочий процесс определения требований и получаются итоговые артефакты (см. главу 12).

- В течение фазы анализа и определения требований аналитики идентифицируют большую часть вариантов использования с целью разграничить систему и обозреть проект и детализируют наиболее критичные из них (менее 10 %).

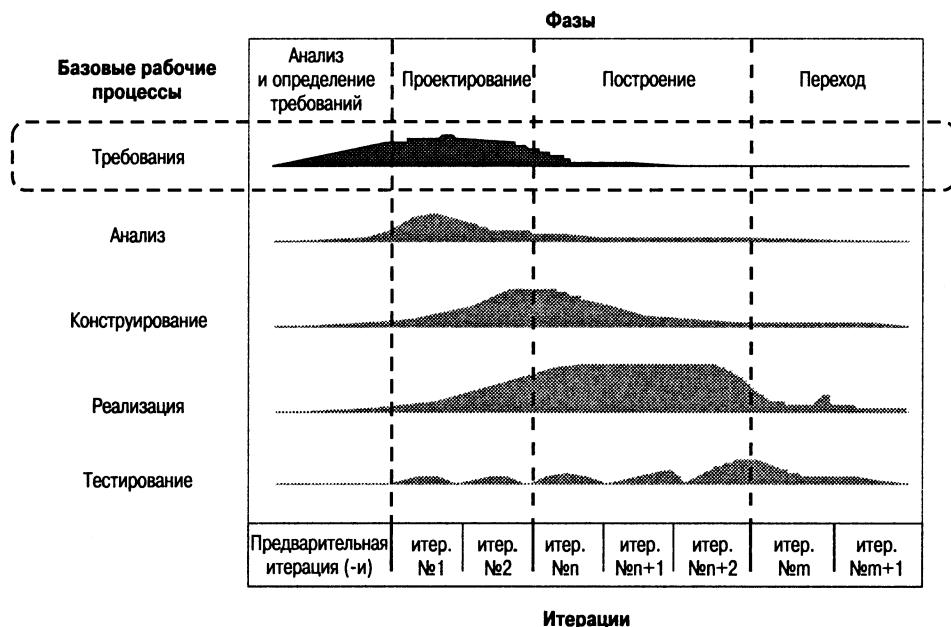


Рис. 6.2. Требования определяются в основном в ходе предварительной фазы и фазы детальной разработки

- В течение фазы проектирования аналитики определяют большую часть оставшихся требований, чтобы разработчики могли оценить объем работ. Цель — к концу фазы детальной разработки определить приблизительно 80% требований и описать большинство вариантов использования. (Отметим, что лишь 5–10% этих требований должны быть к этому моменту включены в архитектуру.)
- Оставшиеся требования фиксируются и выполняются в течение фазы построения.
- В течение фазы перехода определение требований практически не проводится. Исключением будут только случаи изменения требований.

Понимание контекста системы с помощью модели предметной области

Что такое модель предметной области?

Модель предметной области определяет наиболее важные типы объектов контекста системы. Объекты предметной области представляют собой «предметы», которые существуют, или события, которые происходят в той среде, в которой работает система [34, 60].

Многие из объектов предметной области, или, используя более точную терминологию, классов предметной области можно определить из технического задания или в ходе опроса специалистов по проблемной области. Классы предметной области можно разбить на три типовых категории:

- бизнес-объекты, которые описывают сущности, используемые в бизнесе, такие как заявки, счета и контракты;
- объекты и понятия реального мира, которые система должна отслеживать, такие как вражеские самолеты, ракеты и траектории;
- события, которые произойдут или произошли, например прибытие самолета, отлет самолета или перерыв на обед.

Модель предметной области описывается диаграммами UML, главным образом диаграммами классов.

Эти диаграммы дают клиентам, пользователям, рецензентам и другим разработчикам информацию о классах предметной области и их ассоциативных связях.

Пример. Классы предметной области *Заказ*, *Счет*, *Предмет* и *Банковский счет*. Система будет использовать Интернет для пересылки заказов, счетов и платежей между покупателями и продавцами. Система должна помогать покупателю готовить заказы, продавцу — рассчитывать стоимость заказов и рассыпать счета и снова покупателю — проверять правильность выписанных счетов и совершать платеж продавцу с банковского счета покупателя.

Заказ — это запрос покупателя продавцу на поставку изделий. Каждое изделие занимает «одну строку» в заказе. *Заказ* имеет такие атрибуты, как дата выписки и адрес поставки. Диаграмма класса изображена на рис. 6.3.

Счет — это запрос на оплату, посыпаемый продавцом покупателю в ответ на заказ товаров или услуг. *Счет* имеет такие атрибуты, как сумма к оплате, дата выписки и максимальная отсрочка оплаты. На несколько заказов может быть выслан один счет.

Счет считается оплаченным, когда деньги с банковского счета покупателя перешли на банковский счет продавца. *Банковский счет* имеет такие атрибуты, как баланс и владелец. Атрибут «Владелец» идентифицирует лицо, владеющее банковским счетом.



Рис. 6.3. Диаграмма классов в модели предметной области определяет наиболее важные понятия в контексте системы

НОТАЦИЯ UML

На рис. 6.3 представлены классы (прямоугольники), атрибуты (текст в нижней половине прямоугольников классов) и ассоциации (линии между прямоугольниками классов). Текст в конце линий ассоциации объясняет роль одного класса по отношению к другому, то есть роль ассоциации. Кратность — числа и звездочки в конце линии ассоциации — указывают, сколько объектов данного класса связаны с одним объектом класса с другого конца линии ассоциации. Так, ассоциация, соединяющая класс *Счет* и класс *Заказ* на рис. 6.3, имеет кратность $1..*$, проставленную со стороны класса *Заказ*. Это означает, что каждый объект класса *Счет* может быть требованием на оплату одного или более *Заказов*, что обозначено ролью ассоциации «Подлежащий оплате» (приложение А).

Разработка модели предметной области

Моделирование предметной области обычно проводится отделом аналитиков предметной области, применяющих для документирования результатов UML и другие языки моделирования. Чтобы команда аналитиков работала эффективно, в этих отделах должны присутствовать как специалисты в проблемной области, так и люди, разбирающиеся в моделировании.

Цель моделирования предметной области состоит в том, чтобы понять и описать наиболее важные классы контекста предметной области. Небольшие предметные области обычно содержат от 10 до 50 основных классов. В более обширной предметной области классов может быть гораздо больше.

Оставшиеся сотни кандидатов в классы, которые аналитики выявляют внутри предметной области, сохраняются в глоссарии в виде определений понятий. В противном случае модель предметной области станет слишком большой и потребует значительно больших усилий, чем запланировано для этой стадии процесса.

Иногда, например, для особенно малых предметных областей, относящихся к бизнесу, нет необходимости создавать объектную модель предметной области. Вполне достаточно будет глоссария понятий.

Глоссарий и модель предметной области помогают пользователям, клиентам, разработчикам и другим заинтересованным лицам использовать общий словарь. Для того чтобы обмениваться знаниями, необходима общая терминология. Там, где процветает беспорядок, разрабатывать программы трудно, если не невозможно. Поэтому для построения программной системы любого размера современные инженеры должны проделать процедуру, обратную той, что произошла при постройке Вавилонской башни, — «объединить» языки всех участников разработки так, чтобы они понимали друг друга.

В заключение — предостережение, касающееся места моделирования предметной области в процессе моделирования. Очень легко начать моделирование внутренней структуры системы, а не ее контекста [17]. Например, некоторые объекты предметной области могут быть непосредственно представлены в системе, и аналитики предметной области могут раз за разом попадать в ловушку описания деталей с учетом этого представления. В таких случаях очень важно иметь в виду, что цель моделирования предметной области состоит в том, чтобы разобраться в контексте системы и с помощью этого понять требования к системе — ведь они порождаются этим контекстом. Другими словами, моделирование предметной области должно способствовать пониманию задачи, которую система должна решать в контексте предметной области. С внутрисистемными методами решения этой задачи мы будем иметь дело в ходе рабочих процессов анализа, проектирования и реализации (см. главы 8, 9, и 10).

Использование моделей предметной области

Классы предметной области и глоссарий понятий применяются при разработке вариантов использования и аналитических моделей. Они используются:

- при описании вариантов использования и проектировании интерфейса пользователя, которые мы рассмотрим в главе 7;
- при определении внутренних классов разрабатываемой системы в ходе процедуры анализа, которую мы рассмотрим в главе 8.

Однако существует более регулярный способ определения вариантов использования и поиска классов в системе — разработка бизнес-модели. Как мы увидим, модель предметной области на самом деле является частным случаем более полной бизнес-модели. Таким образом, создание бизнес-модели — это серьезная альтернатива разработке модели предметной области.

Понимание контекста системы с помощью бизнес-модели

Бизнес-моделирование — это способ разобраться в бизнес-процессах организации. Но что если вы работаете с системой, которая не имеет никакого отношения к тому, что большинство людей понимает под словом «бизнес»? Например, что мы должны делать при разработке сердечного электростимулятора, антиблокировочной системы торможения для автомобиля, контроллера фотоаппарата или системы беспроводной связи? В этом случае мы по-прежнему можем создавать бизнес-модели этих систем, определяющие программную систему, которую мы собираемся разрабатывать. Эта система (часть человеческого органа, часть автомобиля, фотоаппарата, переключатель) — будет «бизнес-системой» для встроенного программного обеспечения. Это будет заметно по высокому уровню детализации использования системы, которые мы кратко рассмотрим. Наша цель — выделение вариантов использования программного обеспечения и бизнес-сущностей, которые будут поддерживаться программным обеспечением. Для того чтобы сделать это, мы должны углубиться в моделирование ровно настолько, насколько нужно, чтобы разобраться в контексте. Результатом этой деятельности будет модель предметной области, порожденная нашим пониманием функционирования изученной «бизнес-системы».

Технически бизнес-моделирование поддерживается двумя типами моделей UML: моделью вариантов использования и объектной моделью [57]. Обе они определены в бизнес-расширении UML.

Что такое бизнес-модель?

Бизнес-модель вариантов использования описывает бизнес-процессы компании в терминах бизнес-вариантов использования и бизнес-актантов для бизнес-процессов и клиентов, соответственно. Как и модель вариантов использования для программной системы, бизнес-модель вариантов использования представляет систему (в данном случае, бизнес-систему) в разрезе ее использования и объясняет, как она обеспечивает получение результата ее пользователями (здесь — клиентами и партнерами) [38, 39, 57].

Пример. Бизнес-варианты использования. Пример Консорциума Interbank предлагает бизнес-вариант использования, который включает в себя пересылку заказов, счетов и платежей между покупателем и продавцом — *Продажа: от Заказа до Поставки*. В этом бизнес-варианте использования покупатель знает, что покупать и где. В предлагаемой последовательности действий Interbank действует как посредник, соединяя покупателя и продавца друг с другом и обеспечивая безопасность процедур оплаты счетов следующим образом.

1. Покупатель заказывает товары или услуги.
2. Продавец поставляет товары или услуги.
3. Продавец выставляет счет покупателю.
4. Покупатель платит.

В этом контексте покупатель и продавец — бизнес-актанты Interbank, которые используют бизнес-вариант использования, предоставляемый им Interbank.

Бизнес обычно предполагает наличие множества бизнес-вариантов использования. Interbank не исключение. Для получения правильного контекста мы опишем здесь только два варианта использования, а другие процессы обсуждать не станем.

В бизнес-варианте использования *Получение Ссуды: от Заявления до Выплаты* клиент банка направляет заявление на получение ссуды в Interbank и получает от него деньги.

Клиент банка представляет собой обобщенного клиента банка. *Покупатель* и *Продавец* — более определенные категории клиентов.

В бизнес-вариантах использования *Снять деньги со счета, Положить деньги на счет и Перечислить деньги на другой счет* клиент банка снимает деньги со счета, вносит их или перемещает деньги со счета на счет. Этот бизнес-вариант использования в будущем также позволит клиенту банка осуществлять автоматическое перечисление средств.

Бизнес-модель вариантов использования описывается диаграммой использования (см. главы 4 и 7).

Модель бизнес-объектов — внутренняя модель бизнеса. Она описывает, как каждый бизнес-вариант использования реализуется сотрудниками, использующими бизнес-объекты и рабочие модули. Каждая реализация бизнес-варианта использования может быть описана диаграммами взаимодействия (см. главы 4 и 9) и диаграммами активности (такими, как диаграммы рабочих процессов в главах 7–11).

Бизнес-сущность представляет собой что-то вроде счета, к которому сотрудники имеют доступ и могут проверять, осуществлять операции, создавать или использовать в бизнес-варианте использования. Рабочий модуль — это набор таких сущностей, представляющийся конечному пользователю единым целым.

Бизнес-объекты и рабочие модули используются для представления тех же самых типов понятий, что и классы предметной области, — например, таких как *Заказ, Изделие, Счет и Банковский счет*. Мы можем, следовательно, создать диаграмму бизнес-объектов, очень похожую на рис. 6.3. Кроме того, можно создать и другие диаграммы для описания сотрудников, их взаимодействий и использования ими бизнес-объектов и рабочих модулей (рис. 6.4).

Каждый сотрудник, бизнес-сущность или рабочая модель могут участвовать в реализации нескольких бизнес-вариантов использования. Например, класс *Банковский счет* может быть использован в реализации всех трех бизнес-вариантов использования:

- В *Получении ссуды: От Заявления до Выплаты* деньги, полученные в виде ссуды, перечисляются на банковский счет.
- В *Снятии, Внесении и Перечислении денег со счета на счет* деньги снимаются или вносятся на банковский счет или перечисляются с одного счета на другой.
- *Продажа: От Заказа до Поставки* включает в себя перечисление денег со счета покупателя на счет продавца.

Пример. Бизнес-вариант использования Продажа: От Заказа до Поставки.

В бизнес-варианте использования *Продажа: От Заказа до Поставки* сотрудники должны последовательно выполнить следующие действия (рис. 6.4).

1. Покупатель заказывает товары или услуги, заключая контракт с продавцом.
2. Продавец посылает счет покупателю через проводящее платеж лицо.
3. Продавец предоставляет покупателю товары или услуги.
4. Покупатель платит через проводящее платеж лицо. Это действие включает в себя перечисление денег со счета покупателя на счет продавца.

Проводящее платеж лицо – это сотрудник банка, который участвует в выполнении шагов 2 и 4. Эти задачи и должны быть автоматизированы информационной системой.

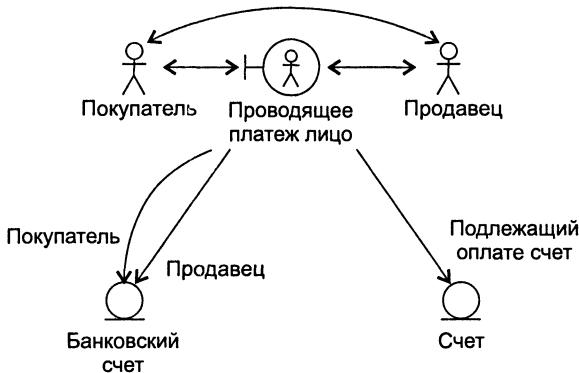


Рис. 6.4. В бизнес-вариант использования Продажа: От Заказа до Поставки включены Продавец, Покупатель и Лицо, проводящее платеж. Лицо, проводящее платеж, перечисляет деньги с одного счета на другой, как указано в счете

Покупатель и продавец используют (автоматизированного) посредника проведения платежей, потому что этот сотрудник приносит им пользу. Сотрудник, проводящий платежи, приносит пользу продавцу, посылая счета покупателям и отслеживая неоплаченные счета. Сотрудник, проводящий платежи, приносит пользу покупателю, упрощая ему процедуру платежа и обеспечивая удобную отчетность и возможность оплаты счетов.

Как разработать бизнес-модель

Бизнес-модель разрабатывается в два приема. Это происходит следующим образом.

1. Разработчики бизнес-модели должны создать бизнес-модель вариантов использования, идентифицирующую актантов, и бизнес-варианты использования, в которых участвуют эти актанты. Эта бизнес-модель вариантов использования позволит разработчикам модели лучше понять, какой результат приносит бизнес его участникам.
2. Разработчики модели должны разработать объектную бизнес-модель, состоящую из сотрудников, бизнес-сущностей и рабочих модулей, которые совместно реализуют бизнес-варианты использования¹. С этими объектами связываются бизнес-правила и другие нормы бизнеса. Цель этого шага состоит в том, чтобы создать сотрудников, бизнес-сущности и рабочие модули, которые реализуют бизнес-варианты использования настолько эффективно, насколько это возможно — то есть быстро, точно и недорого.

Бизнес-моделирование и моделирование предметной области похожи друг на друга. Фактически, мы можем считать моделирование предметной области упрощенным вариантом бизнес-моделирования, в котором мы сосредоточиваемся ис-

¹ Так как модель предметной области есть упрощенный вариант бизнес-модели, мы описываем в качестве исходной для дальнейших базовых рабочих процессов только последнюю (см. главы 7–11)

ключительно на «предметах», то есть классах предметной области или бизнес-объектах, с которыми работают сотрудники. Это означает, что классы предметной области и бизнес-объекты — очень близкие понятия, и мы будем использовать по-переменно то один из этих терминов, то другой.

Однако между бизнес-моделированием и моделированием предметной области имеются серьезные различия, которые говорят в пользу выполнения более формальной процедуры бизнес-моделирования:

- Классы предметной области возникают из базы знаний, составленной несколькими специалистами по проблемной области, или просто из общих соображений (например, из знания о других классах предметной области, спецификации требований и т. п.), относящихся к похожим на нашу системам. Бизнес-объекты же выделяют путем опроса клиентов бизнеса, вычленения бизнес-вариантов использования и последующего выбора объектов. При подходе, используемом в бизнес-моделировании, включение сущности в бизнес-модель должно оправдываться использованием этой сущности в бизнес-варианте использования. Эти различные подходы обычно приводят к разным наборам классов, ассоциаций, атрибутов и операций. При моделировании предметной области можно проследить путь от классов назад к опыту специалистов по проблемной области. При бизнес-моделировании можно проследить потребность в каждом элементе модели назад к клиентам.
- Классы предметной области содержат множество атрибутов, но обычно содержат мало операций или не содержат их вовсе. Для бизнес-объектов это не так. Бизнес-моделирование позволяет идентифицировать не только сущности, но и всех сотрудников, которые участвуют в реализации бизнес-варианта использования, используя эти сущности. Кроме того, при бизнес-моделировании мы определяем способы использования сотрудниками этих сущностей посредством операций, которые должна позволять выполнять каждая сущность. Эти операции также будут получены из требований и могут быть отслежены до клиентов.
- Список сотрудников, обнаруженных при бизнес-моделировании, используется как исходная точка для определения первоначального набора актантов и вариантов использования для информационной системы, которую мы создаем. Это позволяет нам отслеживать каждый вариант использования в информационной системе через сотрудников и бизнес-варианты использования назад, до клиентов. Мы займемся этим в главе 7. Кроме того, как было описано в главе 3, каждый вариант использования может быть прослежен до составляющих систему элементов. Итак, мы можем заключить, что комбинация бизнес-моделирования и подхода к разработке программного обеспечения, предлагаемого Унифицированным процессом, позволят нам непрерывно отслеживать потребности клиента — в бизнес-процессах, сотрудниках и вариантах использования и коде программы. При использовании же одной модели предметной области нет никаких очевидных способов отследить требования в промежутке между моделью предметной области и вариантами использования системы.

Применение бизнес-моделирования для описания Унифицированного процесса. Часть 1

Бизнес-моделирование применяется нами для описания процессов компании-клиента. Тот же самый подход может быть использован для описываемого в этой книге Унифицированного процесса, используемого для разработки программного обеспечения. Когда мы описываем Унифицированный процесс разработки программ-

ного обеспечения, мы демонстрируем использование бизнес-расширений UML (см. главу 2). Такой подход имеет важное теоретическое значение, он также очень практичен. Это своего рода раскрутка, применение метода к нему самому. Она показывает сильные и слабые стороны такого подхода. Итак, Унифицированный процесс — это бизнес-вариант использования, применяемый в промышленности создания программного обеспечения. Внутри промышленности создания программного обеспечения процесс организован, или, как мы говорим, «реализован» в виде последовательности взаимозависимых рабочих процессов: определение требований (обсуждаемое в этой главе и в главе 7), анализ (глава 8), проектирование (глава 9), реализация (глава 10) и тестирование (глава 11). Каждый рабочий процесс — это реализация части бизнес-варианта использования *Унифицированный процесс*, который описан в понятиях:

- сотрудников — например, системных аналитиков и определителей вариантов использования;
- бизнес-сущностей, или, как мы их называем, артефактов — например, вариантов использования и тестовых примеров;
- рабочих модулей — которые также являются артефактами — например, модели вариантов использования и описания архитектуры.

Сотрудники, бизнес-объекты и рабочие модули Унифицированного процесса также описываются диаграммами классов UML вместе с наиболее важными взаимосвязями.

Поиск вариантов использования по бизнес-модели

Используя в качестве исходных данных бизнес-модель, аналитик применяет для создания модели вариантов использования системный подход.

Сначала аналитик идентифицирует актантов для каждого сотрудника и участника бизнеса (то есть каждого потребителя), который будет пользоваться информационной системой.

Пример. Актант — Покупатель. Покупатель использует Биллинговую и Платежную систему для заказа товаров или услуг и оплаты счетов. Таким образом, покупатель является как потребителем, так и актантом, потому что использует платежную систему для заказа и платежей в двух вариантах использования — Заказ Товаров или Услуг и Оплата счетов.

В информационной системе должен поддерживаться каждый сотрудник (и бизнес-актант), который будет ее пользователем. Мы определим, какая необходима поддержка, перебирая всех сотрудников по одному. Для каждого сотрудника мы должны опознать все бизнес-варианты использования, в которых он участвует. Для каждого из вариантов использования, в котором сотрудник выполняет некоторую роль, будет существовать реализация варианта использования, в которой эту роль будет выполнять соответствующий класс.

После того как мы нашли все роли сотрудников или бизнес-актантов, по одной на каждый бизнес-вариант использования, в котором они участвуют, мы можем искать варианты использования для актантов информационной системы. Каждому сотруднику и участнику бизнеса соответствует актант информационной системы. Для каждой роли сотрудника или участника бизнеса необходим соответствующий вариант использования для актанта.

Таким образом, наиболее естественный способ определить пробный набор вариантов использования состоит в том, чтобы создать варианты использования соответствующего актанта для каждой роли каждого сотрудника или бизнес-актанта. В результате для любого бизнес-варианта использования будет существовать по одному варианту использования для каждого сотрудника и бизнес-актанта. Аналитики могут затем заниматься детализацией и приведением этих пробных вариантов использования в порядок.

Аналитики должны также решить, какие из задач сотрудников или участников бизнеса должны быть автоматизированы информационной системой (в виде вариантов использования), и реорганизовать варианты использования так, чтобы они лучше соответствовали потребностям этих лиц. Заметим, что для автоматизации пригодны не все задачи.

Пример. *Определение вариантов использования по бизнес-модели.* В продолжение предшествующего примера мы можем предложить пробный вариант использования *Покупка Товаров или Услуг*, которые будут основаны на действиях актанта-покупателя, выступающего в качестве бизнес-актанта в бизнес-варианте использования *Продажа: От Заказа до Поставки*. В ходе дальнейшего анализа становится очевидно, что *Закупку Товаров или Услуг* лучше реализовать в виде нескольких различных вариантов использования, например *Заказ Товаров или Услуг* или *Оплата счета*. Причина для дробления пробного варианта использования на отдельные меньшие варианты использования состоит в том, что покупатель не хочет выполнять вариант использования *Покупка Товаров или Услуг* в виде одной непрерывной последовательности действий. Вместо этого он желает дождаться поставки товаров или услуг, а потом уже оплачивать счет. Последовательность действий по оплате счета, следовательно, представляется в виде отдельного варианта использования *Оплата Счета*, который выполняется после того, как заказанное будет поставлено.

Итак, мы рассмотрели, как моделировать контекст системы, используя модель предметной области или бизнес-модель. Затем мы видели, как можно получить из бизнес-модели такую модель вариантов использования, которая вберет в себя все функциональные требования к системе и большую часть нефункциональных. Некоторые требования не могут быть связаны с каким-либо одним из вариантов использования и известны как *дополнительные требования*.

Дополнительные требования

Дополнительными чаще всего бывают нефункциональные требования, которые не могут быть связаны с каким-либо одним вариантом использования. Вместо этого каждое такое требование относится к нескольким вариантам использования или вообще ни к одному. Примерами таких требований могут быть производительность, требования к пользовательскому интерфейсу, физической структуре и архитектуре или ограничения реализации [28]. Дополнительные требования часто определяются так же, как требования вообще в традиционных технических заданиях, то есть в виде списка требований. Затем они используются в ходе анализа и проектирования вместе с моделью вариантов использования.

Требования к интерфейсу определяют интерфейс с внешними модулями, с которыми система должна взаимодействовать или в которые она будет встраиваться, — его формат, синхронизацию или другие факторы, важные для их взаимодействия.

Физические требования определяют физические характеристики, которые должны иметь система: ее материал, форму, размер или вес. Этот тип требований может использоваться для заявления требований к оборудованию, таких, как, например, необходимая физическая конфигурация сети.

Пример. Требования к компьютерной платформе.

Серверы. Sun Sparc 20 или PC Pentium

Клиенты. PC (процессор как минимум Intel 486) или Sun Sparc 5

Требования к проекту ограничивают структуру системы. К ним относятся требования расширяемости и ремонтопригодности или требования, описывающие повторное использование унаследованной системы и ее существенных частей.

Требования к реализации определяют или ограничивают кодирование или построение системы. Примеры таких требований — стандарты, которые необходимо соблюдать, руководящие принципы реализации, языки реализации, политика в отношении целостности базы данных, лимиты ресурсов и операционная система.

Пример. Требования к форматам файлов. Версия 1.2 Биллинговой и Платежной системы должна поддерживать работу с длинными именами файлов.

Пример. Требования к базовому программному обеспечению. Системное ПО. Операционная система на клиентских машинах: Windows NT 4.0, Windows 95 или Solaris 2.6. Серверная операционная система: Windows NT 4.0 или Solaris 2.6.

ПО для работы в Интернете. Netscape Communicator 4.0 или Microsoft Internet Explorer 4.0

Кроме того, часто имеются и другие требования, например юридические и регулирующие.

Пример. Другие требования.

Безопасность. Перечисление денег должно быть безопасно. Это означает, что к соответствующей информации могут иметь доступ только доверенные лица. Единственные доверенные лица — это клиенты банка, обладающие банковскими счетами, и актанты — администраторы системы.

Доступность. Биллинговая и Платежная система не должна находиться в нерабочем состоянии более 1 часа в месяц.

Легкость обучения. Для 90% покупателей время обучения (при помощи предоставленных пошаговых инструкций) отсылке простых заказов и оплате простых счетов не должно превышать 10 минут. Простым заказом считается заказ, состоящий из одного пункта. Простым счетом считается счет на оплату одного простого заказа.

Резюме

Итак, мы постарались подробно объяснить процесс определения требований. Мы увидели, как бизнес-модели и модели предметной области помогают определить контекст системы и как из бизнес-модели можно получить варианты использования. Мы увидели, что для определения требований используются варианты использования. К этому моменту мы вернемся в следующей главе. В последующих главах мы рассмотрим, как варианты использования и дополнительные требования помогают нам проводить анализ, разрабатывать архитектуру, проектировать, создавать и тестировать систему так, чтобы она гарантированно выполняла требования заказчиков.

7 Определение требований в виде вариантов использования

Введение

Главная задача процесса определения требований — это создание модели формируемой системы и разработка вариантов использования — удобный способ создать такую модель. Причина этого в том, что функциональные требования естественным образом собираются в варианты использования. Поскольку большинство остальных нефункциональных требований определяются для единственного варианта использования, работа с ними также происходит в контексте вариантов использования.

Прочие нефункциональные требования, общие для нескольких или всех вариантов использования, выделяются в отдельный документ и известны под названием дополнительных требований. Эти требования были описаны в главе 6, и мы не будем возвращаться к ним снова, пока не займемся изучением их использования в рабочих процессах анализа, проектирования, реализации и тестирования.

Варианты использования предлагают системный и интуитивно понятный способ определения функциональных требований с особым упором на значимость получаемых результатов для каждого индивидуального пользователя или внешней системы. Используя варианты использования, аналитики вынуждены думать о том, кто является пользователем системы и какие задачи бизнеса или другую работу они могут выполнять. Однако, как мы говорили в главе 4, варианты использования не употреблялись бы в разработке так активно, если бы этим ограничивалась вся польза, извлекаемая из их употребления. Их ключевая роль в направлении других процессов разработки стала важной причиной для использования вариантов использования во многих современных способах разработки программного обеспечения [41].

В этой главе мы детализируем наше понимание вариантов использования и актантов и представим более строгие определения, чем те, которыми мы пользовались в главе 3. Мы описываем рабочий процесс определения требований (подобным образом мы опишем в главах 8–11 и остальные рабочие процессы) в три этапа.

- Артефакты, создаваемые в ходе рабочего процесса определения требований.
- Сотрудники, участвующие в рабочем процессе определения требований.
- Рабочий процесс определения требований, включая подробное описание каждого вида деятельности.

Для начала мы рассмотрим сотрудников и артефакты, изображенные на рис. 7.1.

Применение бизнес-моделирования для описания Унифицированного процесса. Часть 2

Мы опознаем сотрудников и артефакты, используемые в каждом из рабочих процессов. Сотрудник — это обозначение занятия, которое может быть поручено одному лицу или группе лиц, оно определяет требуемые для этого занятия качества и способности (см. также подраздел «Размещение «Ресурсов» внутри «Сотрудников»» главы 2).

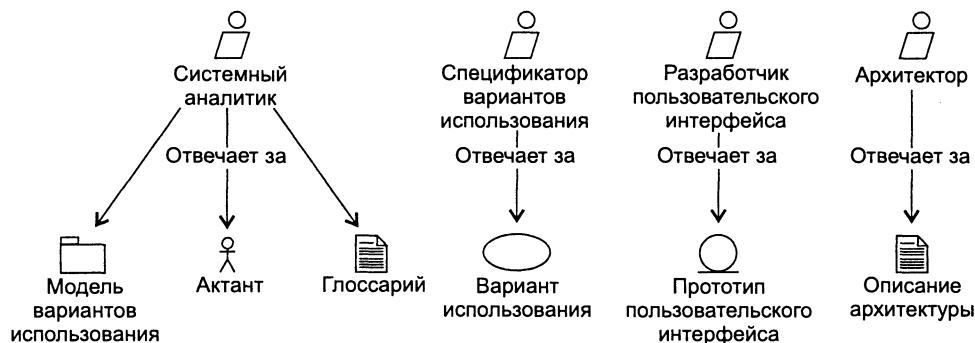


Рис. 7.1. Сотрудники и артефакты, входящие в определение требований в виде вариантов использования

Артефакт — это общий термин для любого вида описания или другой информации, создаваемой, изменяющейся или используемой сотрудниками при работе с системой. Артефактом может быть модель, элемент модели или документ. Например, в рабочем процессе определения требований артефактами являются модель вариантов использования и входящие в нее варианты использования. Заметим, что в бизнес-модели Унифицированного процесса артефактами будут бизнес-сущности или рабочие модули. Каждый сотрудник отвечает за создание и использование некоторого набора артефактов. Эта связь показана на диаграммах в виде ассоциации с именем «отвечает за» от сотрудника к соответствующим артефактам (см. рис. 7.1). Чтобы сделать эти диаграммы интуитивно понятными, мы используем для большинства артефактов специальные значки. Артефакты, представляющие собой документы, обозначаются значками документа. Артефакты, пред-

ставляющие собой модели, обозначаются соответствующими обозначениями из UML. Заметим, что для того, чтобы иметь право использовать эти специальные значки в бизнес-модели Унифицированного процесса, мы вводим стереотипы для документов, моделей и элементов моделей. Каждый такой стереотип является подтипов стереотипов *Бизнес-сущность* или *Рабочий модуль*. Мы покажем, как сотрудники совместно осуществляют рабочий процесс, рассмотрим, как центр внимания перемещается от одного сотрудника к другому и как сотрудники, осуществляя свою деятельность, работают с артефактами (см. также подраздел «Связанные деятельности образуют рабочие процессы» главы 2). В этом контексте мы также рассмотрим каждый из видов деятельности более подробно. Вид деятельности — это часть работы, которую сотрудник выполняет в ходе рабочего процесса, одна из операций, выполняемых сотрудником.

Артефакты

Главный артефакт, используемый при определении требований, — это модель вариантов использования, включающая в себя варианты использования и актантов. При этом могут также использоваться артефакты других видов, например прототипы пользовательских интерфейсов.

Эти артефакты были описаны в главе 3, но здесь мы дадим им более точные определения, совместимые с определениями из [57]. Затем мы отступим от формализма и покажем, как использовать эти конструкции на практике в Унифицированном процессе. Мы приводим определения для того, чтобы обеспечить себе надежное основание. Нет никакой необходимости применять этот формализм на практике. Мы все же включили определения в эту главу по следующим причинам.

- Определения могут быть важны для формального описания некоторых типов вариантов использования, таких как использование диаграмм деятельности или диаграмм состояний. Это особенно ценно для вариантов использования со множеством состояний и сложными переходами из состояния в состояние.
- Включение определений упрощает опознание правильных вариантов использования и их непротиворечивое описание. На самом деле, даже если вы отказываетесь от использования доступного формализма при описании, например, актантов или вариантов использования, неплохо держать их в уме. Это поможет вам сделать описание полным и непротиворечивым.
- Определения, что немаловажно, дают нам возможность пояснить положение структурных компонентов актанта и варианта использования относительно других структурных компонентов UML.

Артефакт: Модель вариантов использования

Модель вариантов использования позволяет разработчикам программного обеспечения и клиентам договориться о требованиях, которым должна удовлетворять система [36], то есть об условиях или возможностях, которым система должна соответствовать или которые она должна иметь. Модель вариантов использования

служит соглашением между клиентами и разработчиками и предоставляет необходимые исходные данные для анализа, проектирования и тестирования.

Модель вариантов использования — это модель системы, содержащая актанты, варианты использования и их связи (рис. 7.2).

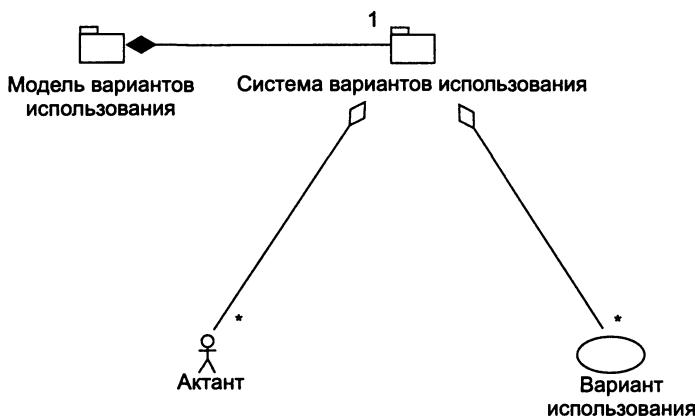


Рис. 7.2. Модель вариантов использования и ее содержимое.
Система вариантов использования обозначает верхний уровень пакетов модели

Модель вариантов использования может стать слишком большой и трудной для того, чтобы использовать ее единым блоком, а значит, нам нужен некий способ для работы с частями модели. UML позволяет нам оформить модель в виде диаграмм, которые представляют актантам и вариантам использования с различных точек зрения и в различных целях. Также заметим, что если модель вариантов использования велика, то есть содержит большое количество вариантов использования и/или актантов, то для управления ее размером может быть полезно ввести в модель пакеты. Это более или менее тривиальное расширение модели вариантов использования, и в этой книге оно не рассматривается.

Артефакт: Актант

Модель вариантов использования описывает действия системы для каждого типа пользователей. Каждая категория пользователей представлена одним или более актантами. Каждая внешняя система, с которой взаимодействует наша система, также представляется одним или более актантами. В число актантов входят, в частности, внешние устройства, например таймеры, которые считаются находящимися вне системы. Таким образом, актантам представляют собой внесистемные агенты, которые взаимодействуют с системой. Когда мы идентифицируем все актантам нашей системы, мы полностью определим внешнюю среду системы.

Актанты, как было показано в главе 6, часто соответствуют сотрудникам (или бизнес-актантам). Напомним, что роль сотрудника определяет его функции в рамках данного бизнес-процесса. Роли, выполняемые сотрудником, могут быть использованы для создания (или автоматической генерации при наличии соответствующих инструментальных средств) ролей, исполняемых соответствующими актантами.

тами системы. Затем, как это предлагалось в главе 6, мы приписываем каждому сотруднику по одному варианту использования системы на каждую из его ролей. Этот вариант использования предписывает роль сотрудника актанту, тем самым прибавляя актанту значимость.

Пример. *Актант.* Биллинговая и Платежная система взаимодействует с определенным типом пользователей, которые используют ее для заказа товаров, подтверждения заказов, оплаты счетов и т. д. Этот тип пользователей будет представлен актантом *Покупатель* (рис. 7.3).



Рис. 7.3. Актант Покупатель

Актант исполняет одну роль для каждого варианта использования, в котором участвует. Соответствующий экземпляр актанта играет эту роль каждый раз, когда соответствующий пользователь (человек или другая система) взаимодействует с системой. Таким образом, мы определили пользователя, взаимодействующего с системой, как экземпляр актанта. Всякая сущность, считающаяся актантом, может действовать как экземпляр актанта.

Вариант использования

Все способы использования системы актантами представляют собой варианты использования. Варианты использования — это «куски» функциональности, которые система предлагает актантам для получения значимого результата. Более формально, вариант использования определяет последовательность действий, включая альтернативный выбор, которую система может выполнять, взаимодействуя со своими актантами. Таким образом, вариант использования — это спецификация. Она определяет поведение динамических «вещей», например экземпляров варианта использования.

Пример. *Вариант использования Снять деньги со счета.* В главе 3 мы рассматривали вариант использования *Снять деньги со счета*, который поддерживал экземпляр актанта *Клиент Банка* для снятия денег через банкомат (рис. 7.4).



Снять деньги со счета

Рис. 7.4. Вариант использования Снять деньги со счета

Вариант использования *Снять деньги со счета* определяет возможные вариации этого варианта использования, то есть различные законные способы его выполнения системой, включая различные взаимодействия с участвующим в варианте использования экземпляром актанта. Так, частное лицо Джек сначала набрал ПИН-код 1234, указал к получению сумму \$220 и затем получил деньги. Система

выполнила один экземпляр варианта использования. Если Джек вместо этого наберет тот же самый ПИН-код, укажет к получению сумму \$240 и получит деньги, система выполнит другой экземпляр варианта использования. Третий экземпляр варианта использования будет выполнен, если Джек запросит \$480, а система откажет ему в этом из-за отсутствия денег у него на счете, неверного ПИН-кода или по другой причине.

В словаре UML вариант использования — это классификатор. Это означает, что он содержит операции и атрибуты. Описание варианта использования может, таким образом, включать диаграммы состояний, диаграммы активности и последовательностей (см. приложение А).

Диаграммы состояний определяют жизненный цикл экземпляров варианта использования в терминах состояний и переходов из одного состояния в другое. Каждый переход — это последовательность действий. Диаграммы деятельности описывают жизненный цикл более подробно, охватывая также последовательность действий, происходящих в случае каждого из переходов. Диаграммы кооперации и последовательностей используются, чтобы описать взаимодействие между, например, типичным экземпляром актанта и типичным экземпляром варианта использования. Как мы увидим в подразделе «Деятельность: Детализация вариантов использования», нам практически никогда не приходится использовать этот формализм при описании вариантов использования. Однако знание об уточненном понимании вариантов использования помогает при структурировании их описания.

Экземпляр варианта использования — это выполнение варианта использования. Иначе можно сказать, что экземпляр варианта использования выполняется системой, поскольку он «соответствует правилам» варианта использования. Выполняясь, экземпляр варианта использования взаимодействует с экземплярами актантов и выполняет последовательность действий, определенную данным вариантом использования. Эта последовательность задана диаграммой состояний или диаграммой деятельности, которые разными способами показывают один и тот же путь осуществления варианта использования. Возможно существование множества путей осуществления варианта использования, многие из которых очень похожи. Это альтернативные последовательности осуществления варианта использования. Например, путь прохождения варианта использования может выглядеть следующим образом.

- Экземпляр варианта использования инициируется и устанавливается в начальное состояние.
- Он активируется внешним сообщением, пришедшим от актанта.
- Он переходит в другое состояние, выполняя при этом некую последовательность действий. Эта последовательность содержит внутренние вычисления, выбор пути и посылку сообщений (каким-либо актантам).
- Он ждет в новом состоянии прихода от актанта другого внешнего сообщения.
- Он снова активируется пришедшим сообщением и т. д. Это может продолжаться много раз — до тех пор, пока экземпляр варианта использования не будет уничтожен.

Наиболее часто экземпляр варианта использования активируется экземпляром актанта так, как мы только что описали, но активация может быть также результа-

том системного события, например переключения триггера по таймеру (если таймер считается внутрисистемным объектом).

Варианты использования, как и все классификаторы, имеют атрибуты. Эти атрибуты представляют собой значения, которыми экземпляр варианта использования манипулирует в ходе выполнения своего варианта использования. Значения являются локальными для экземпляра варианта использования, то есть они не могут использоваться другими экземплярами варианта использования. Например, вариант использования *Снять деньги со счета* можно представить себе как имеющий атрибуты *ПИН*, *Банковский счет* и *Сумма к выдаче*.

Экземпляры вариантов использования не взаимодействуют с другими экземплярами вариантов использования. Единственный вид взаимодействий в модели вариантов использования происходит между экземплярами актанта и экземплярами варианта использования [40]. Причина этого в том, что мы хотим сделать модель вариантов использования простой и интуитивно понятной, чтобы иметь возможность успешно обсуждать ее с конечными пользователями и другими заинтересованными лицами, не запутываясь в деталях. Мы не хотим иметь дело с интерфейсами между вариантами использования, параллельным выполнением и другими конфликтами (например, совместным использованием других объектов) между различными экземплярами варианта использования. Мы рассматриваем экземпляры варианта использования в виде атомов, то есть считаем, что каждый из них выполняется целиком или не выполняется вообще, без какой-либо интерференции с другими экземплярами варианта использования. Поведение каждого варианта использования может, таким образом, интерпретироваться независимо, что сильно упрощает моделирование вариантов использования. Рассмотрение вариантов использования в виде атомарных сущностей не дает нам никаких сведений об обработчике транзакций, лежащем в основе вариантов использования, который предохраняет систему от внутренних конфликтов. Это делается исключительно для того, чтобы гарантировать, что мы сможем читать и понимать модель варианта использования.

Однако мы знаем, разумеется, что интерференция между различными использованием системы существует. Этот эффект не может быть устранен в модели вариантов использования. Мы полагаемся в решении этой проблемы на процессы анализа и проектирования (описанные в главах 8 и 9 соответственно), где мы реализуем варианты использования в форме кооперации классов и/или подсистем. В аналитической модели мы можем четко описать, как, например, какой-либо один класс может одновременно использоваться в нескольких реализациях варианта использования и как следует решать проблемы любых возможных интерференционных явлений между этими вариантами использования.

МОДЕЛИРОВАНИЕ БОЛЬШИХ СИСТЕМ

В этой книге мы сосредоточили свое внимание на моделировании отдельной системы. Однако во многих случаях нам нужно разрабатывать большие системы, системы, фактически сформированные из других систем. Мы называем их системами систем.

Пример. Одна из величайших систем в мире. Самая большая система, когда-либо созданная нами, людьми, система с почти миллиардом пользователей — это гло-

бальная сеть телекоммуникаций. Когда вы делаете телефонный звонок из, скажем, Сан-Франциско в Петербург, этот запрос пройдет, по всей вероятности, через приблизительно 20 систем, включая местные коммутаторы, международные коммутаторы, спутниковые системы, передающие системы и т. д. Объем разработки системы каждого из этих типов — приблизительно 1000 человеко-лет, и немалая их часть — это затраты на создание программного обеспечения. Просто потрясающе, что, когда мы совершаляем такие звонки, все обычно работает. При такой сложности и при том количестве людей, компаний и государств, которые вовлечены в сеть, как она ухитряется работать? Основная причина в том, что каждый интерфейс во всей сети (то есть архитектура сети) был стандартизирован отдельной организацией, ITU. ITU определил интерфейсы между всеми типами узлов сети и точную семантику этих интерфейсов. Построение системы систем основывается на методах, подобных тем, что использовались при формировании глобальной сети телекоммуникаций [46].

Сначала определяется главная система с ее вариантами использования. Они разрабатываются в терминах взаимодействующих подсистем. Варианты использования главной системы разделяются на варианты использования взаимодействующих подсистем, и подсистемы связываются друг с другом через интерфейсы. Эти интерфейсы детально определяются, после чего каждая отдельная подсистема может разрабатываться независимо (как система в себе) отдельной организацией. UML поддерживает этот вид архитектуры. Унифицированный процесс также может быть расширен для получения возможности разработки таких систем [50].

На самом деле варианты использования можно использовать для описания не только систем, но и меньших объектов, а именно подсистем или классов. Таким образом, подсистема или класс могут иметь описание из двух частей. Каждая из частей описывает одно представление — одна часть спецификацию, другая часть реализацию. Спецификация описывает, как подсистема или класс выглядят с точки зрения окружения в терминах вариантов использования. В реализации описывается, как подсистема или класс устроены изнутри для выполнения записанного в спецификации. Окружение обычно составляется из других подсистем и классов. Однако, если мы хотим трактовать окружение анонимно, мы можем также представить его при помощи актанта.

Это приближение используется, когда мы хотим трактовать подсистему как систему в ее области действия, например, в следующих случаях:

- Когда мы собираемся создать подсистему, используя другую технологию, нежели использованная при создании других подсистем. Мы можем делать это сколько угодно часто, лишь бы эта подсистема предоставляла нам правильные варианты использования и поддерживала и использовала описанные интерфейсы.
- Когда мы собираемся управлять подсистемой отдельно от других подсистем, например в географически удаленном месте.

Поток событий

Поток событий для каждого варианта использования может фиксироваться в отдельном текстовом описании последовательности действий варианта использова-

ния. Поток событий, таким образом, определяет, что делает система при выполнении указанного варианта использования.

Поток событий определяет также взаимодействие системы с актантами при выполнении варианта использования. С точки зрения руководства поток описания событий включает набор последовательностей действий, которые можно изменять, просматривать, проектировать, создавать, тестировать и описывать как пункт или подраздел в руководстве пользователя. Мы покажем примеры описания потока событий для варианта использования в подразделе «Детализация варианта использования».

Специальные требования

Специальные требования — это текстовое описание, в котором собраны все нефункциональные требования варианта использования. Это прежде всего нефункциональные требования, связанные с конкретным вариантом использования и требующиеся для последующих рабочих процессов — анализа, проектирования или реализации.

Мы покажем примеры описания специальных требований для варианта использования в подразделе «Детализация варианта использования».

Артефакт: Описание архитектуры

Описание архитектуры содержится в архитектурном представлении модели вариантов использования и касается вариантов использования, оказывающих влияние на архитектуру (рис. 7.5).



Рис. 7.5. Описание архитектуры

Архитектурное представление модели вариантов использования должно включать в себя варианты использования, которые описывают важные или критические функциональные возможности системы либо содержат некоторые важные требования, которые должны быть разработаны в начале жизненного цикла программного обеспечения. Это архитектурное представление используется в качестве исходных данных для расстановки приоритетов разработки (то есть анализа, проектирования и реализации) вариантов использования внутри итерации. Эта тема более подробно рассмотрена в частях 1 и 3 (см. подраздел «Варианты использо-

зования и архитектура» главы 4 и подраздел «Расстановка приоритетов вариантов использования» главы 12 соответственно).

Обычно соответствующие реализации вариантов использования могут быть найдены в архитектурных представлениях аналитической и проектной моделей (см. подраздел «Описание архитектуры» главы 4, подраздел «Артефакт: Описание архитектуры (представление модели анализа)» главы 8 и подраздел «Артефакт: Описание архитектуры (представление модели проектирования)» главы 9).

Артефакт: Глоссарий

Глоссарий может использоваться для определения важных или общих для проекта понятий, используемых аналитиками (и другими разработчиками) при описании системы. Глоссарий очень полезен в деле достижения согласия между разработчиками при определении различных концепций и понятий и снижает риск всеобщего непонимания.

Глоссарий нередко может быть получен из бизнес-модели или модели предметной области, но поскольку он менее формален (не включает в себя никаких классов или выявленных связей), он проще в работе и интуитивно понятнее при обсуждениях, проводимых с посторонними людьми — пользователями и клиентами. Кроме того, глоссарий обычно концентрируется на разрабатываемой системе, а не на контексте системы, что характерно для бизнес-модели или модели предметной области.

Артефакт: Прототип интерфейса пользователя

Прототипы пользовательского интерфейса помогают нам в ходе определения требований понять и определить взаимодействия между актантами-людьми и системой. Это помогает нам не столько разработать лучший пользовательский интерфейс, сколько лучше разобраться в вариантах использования. При определении интерфейса пользователя могут также использоваться другие артефакты — модели пользовательского интерфейса и эскизы экрана.

См. также [10, 11, 34, 35, 40, 57] по вопросу расширенного описания актантов и вариантов использования и [16] о проектировании пользовательского интерфейса.

Сотрудники

Ранее в этой главе мы обсуждали артефакты, производимые в ходе моделирования вариантов использования. Следующим шагом будет рассмотрение сотрудников, ответственных за порождение этих артефактов.

Как мы уже говорили в главе 2, сотрудник — это позиция, с которой должен быть сопоставлен «реальный» человек. Каждому сотруднику соответствует описание обязанностей и ожидаемого поведения этого сотрудника. Сотрудник не идентичен конкретному человеку, один человек может в течение проекта «пересаживаться со стула на стул» и рассматриваться каждый раз в качестве другого сотрудника. При этом сотрудник не соответствует определенной должности или позиции

в организации — это разные понятия. Скорее можно сказать, что сотрудник представляет собой абстракцию человека с некоторыми способностями, необходимыми в бизнес-варианте использования, в данном случае, разработке программного обеспечения методом Унифицированного процесса. Когда проект запущен, сотрудник предоставляет свои знания и способности кому-то, кому необходим для работы над проектом именно такой сотрудник. Мы можем указать трех сотрудников, которые участвуют в моделировании вариантов использования, каждого с его собственным набором действий и требуемых обязательств: системный аналитик, спецификатор вариантов использования и разработчик пользовательского интерфейса. Каждому из этих сотрудников будет посвящен пункт с анализом его функций. Существуют также и другие сотрудники, например рецензенты, но мы не станем рассматривать их в этой книге.

Сотрудник: Системный аналитик

Системный аналитик отвечает за весь набор требований, которые моделируются в виде вариантов использования, включая все функциональные и нефункциональные требования, которые относятся к конкретному варианту использования. Системный аналитик отвечает за ограничение системы, выявление актантов и вариантов использования и должен гарантировать, что модель вариантов использования полна и согласована. Для обеспечения непротиворечивости при определении требований системный аналитик может использовать глоссарий, чтобы обеспечить согласие по общим терминам, понятиям и концепциям. Обязанности системного аналитика иллюстрируются на рис. 7.6.

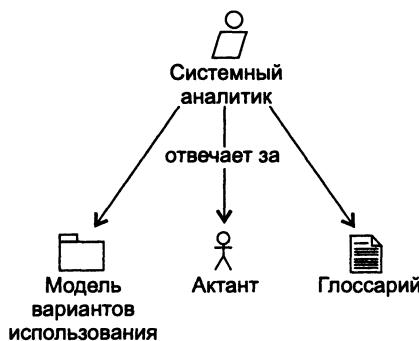


Рис. 7.6. Обязанности системного аналитика при определении требований в виде вариантов использования

Хотя системный аналитик и отвечает за модель вариантов использования и содержащиеся в ней актанты, он не отвечает за каждый конкретный вариант использования. Это отдельная работа и отдельная ответственность, поручаемая сотруднику — спецификатору вариантов использования (подраздел «Сотрудник: Спецификатор вариантов использования»). Системный аналитик также руководит работами по моделированию и координирует определение требований.

Для каждой системы существует один системный аналитик. Однако на практике функции этого сотрудника выполняются группой (в фирмах или подобных организациях) из множества людей, которые занимаются работой аналитиков.

Сотрудник: Спецификатор вариантов использования

Обычно работа по определению требований не может быть выполнена одним человеком. Системному аналитику помогают другие сотрудники, каждый из которых принимает на себя ответственность за детальное описание одного или более вариантов использования. Этих сотрудников мы назвали спецификаторами вариантов использования (рис. 7.7). Каждый спецификатор вариантов использования должен работать непосредственно с реальными потребителями его вариантов использования.



Рис. 7.7. Обязанности спецификатора вариантов использования при определении требований в виде вариантов использования

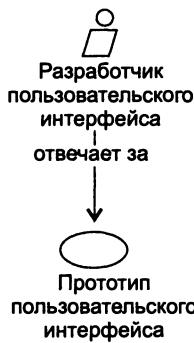


Рис. 7.8. Обязанности разработчика пользовательского интерфейса при определении требований в виде вариантов использования

Сотрудник: Разработчик интерфейса пользователя

Разработчики пользовательского интерфейса формируют вид интерфейсов пользователя. Эта работа нередко включает в себя разработку прототипов пользовательского интерфейса для некоторых вариантов использования, обычно по одному

прототипу для каждого актанта (рис. 7.8). Таким образом, разумно будет предположить, что каждый разработчик пользовательского интерфейса создает пользовательский интерфейс для одного или более актандов.

Сотрудник: Архитектор

Архитектор участвует в рабочем процессе определения требований для создания архитектурного представления модели вариантов использования (рис. 7.9).

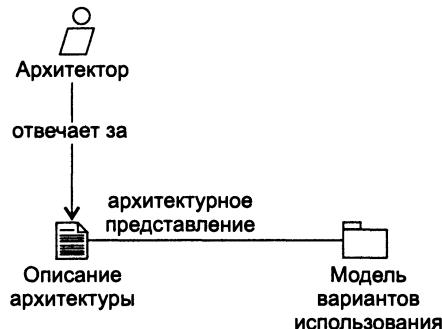


Рис. 7.9. Обязанности архитектора при определении требований в виде вариантов использования

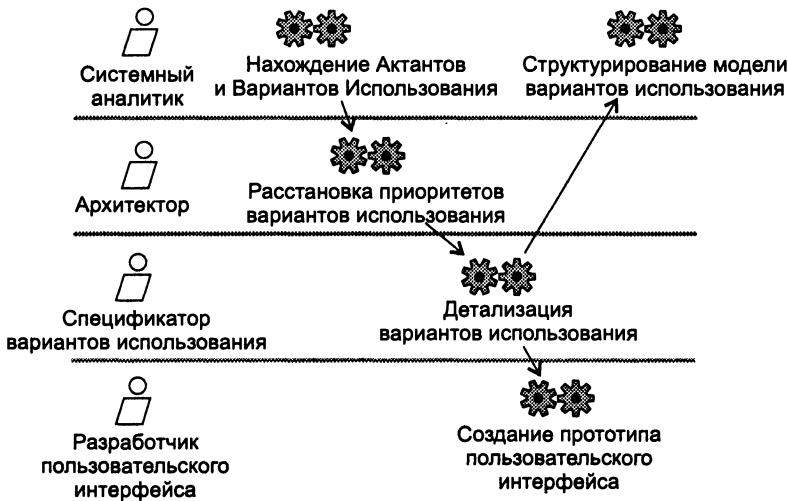


Рис. 7.10. Рабочий процесс определения требований как вариантов использования, включая участников в нем сотрудников и их действия

Архитектурное представление модели вариантов использования необходимо для планирования итераций (см. подраздел «Деятельность: Определение приоритетности вариантов использования»).

Рабочий процесс

В предыдущем пункте мы описали рабочий процесс определения требований в статическом представлении. Теперь мы используем диаграмму деятельности (рис. 7.10), чтобы описать его динамическое поведение.

Для пояснения, какой сотрудник какие действия выполняет, в диаграмме используется метафора «плавательных дорожек». Каждый из видов деятельности (изображенный в виде шестеренок) позиционирован в ту же дорожку, что и сотрудник, ее выполняющий. Сотрудники, выполняя действия, создают и изменяют артефакты. Мы описываем рабочий процесс как последовательность действий, которые упорядочены так, что результат одного вида деятельности является исходными данными для следующего. Однако диаграмма деятельности отражает только логические потоки. В реальной жизни нет необходимости совершать все действия в строгой последовательности. Мы можем делать работу любым другим путем, главное — получить в итоге «соответствующий» результат. Мы можем, например, начать с выделения некоторых вариантов использования (действие *Найти Актантов и Варианты использования*), затем создать интерфейс пользователя (действие *Прототипировать пользовательский интерфейс*). После этого мы понимаем, что следовало бы добавить еще один вариант использования, и переходим назад, к виду деятельности *Найти Актантов и Варианты использования*, нарушая таким образом строгую последовательность действий, и т. д.

Действие, таким образом, может быть неоднократно выполнено повторно, и за каждое выполнение может быть сделана только часть работы. Например, при очередном выполнении действия *Найти Актантов и Варианты использования* единственным результатом может быть идентификация одного дополнительного варианта использования. Пути от действия к действию лишь показывают логическую последовательность действий, использующих результат одного вида деятельности как исходные данные для следующего.

Сначала системный аналитик (лицо, действующее при поддержке группы аналитиков) выполняет действие *Найти Актантов и Варианты Использования*, чтобы подготовить первую версию модели вариантов использования с применением найденных актантов и вариантов использования. Системный аналитик должен гарантировать, что разрабатываемая модель вариантов использования охватывает все требования, полученные в качестве исходных данных рабочего процесса, то есть список пометок и модель предметной области или бизнес-модель. Затем архитектор(ы) вычленяют существенные для архитектуры варианты использования, чтобы обеспечить исходными данными расстановку приоритетов вариантов использования (и возможно, других требований), выполняемую в текущей итерации. Получив их, спецификаторы вариантов использования (несколько человек) описывают все варианты использования в порядке их приоритета. Более или менее параллельно с этим действием разработчики пользовательского интерфейса (несколько человек) предлагают для каждого актанта интерфейсы пользователя, основанные на вариантах использования. Затем системный аналитик реструктурирует модель вариантов использования, определяя общие точки вариантов использования и делая их настолько понятными, насколько возможно (общие точки будут кратко обсуждаться в описании действия *Структурировать модель вариантов использования*).

Результаты первой итерации этого рабочего процесса включают в себя первую версию модели вариантов использования, входящих в нее вариантов использования и связанных с ними прототипов пользовательского интерфейса. Результаты каждой следующей итерации содержат новые версии этих артефактов. Процесс заканчивается, когда закончено итеративное улучшение артефактов. Разумеется, все они к этому времени должны быть созданы.

Различные действия по разработке модели вариантов использования принимают различные формы на разных фазах проектирования (см. подраздел «Роль требований в жизненном цикле разработки программного обеспечения» главы 6). Например, когда системный аналитик выполняет действие *Найти Актантов и Варианты Использования* в фазе анализа и определения требований, он будет находить много новых актентов и вариантов использования. Но когда это же действие выполняется в ходе фазы проектирования, системный аналитик будет в основном вносить в набор актентов и вариантов использования незначительные изменения, например создание новых диаграмм использования, которые лучше описывают модель вариантов использования в некотором частном представлении. Далее мы опишем, как эти же действия проявляются в фазе построения.

Деятельность: Нахождение актентов и вариантов использования

Мы выделяем актентов и варианты использования для того, чтобы:

- отделить систему от ее окружения;
- определить, какие актенты и как взаимодействуют с системой и какие функциональные возможности (варианты использования) ожидаются от системы;
- определить и описать в гlosсарии общие понятия, которые необходимы для создания детальных описаний функциональных возможностей системы (например, варианты использования).

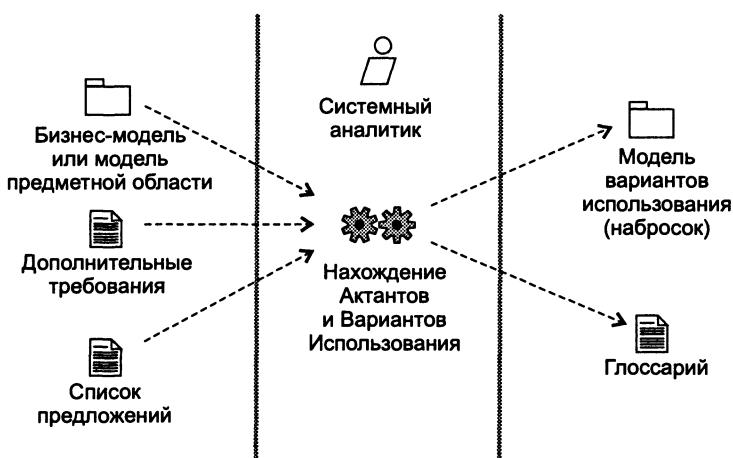


Рис. 7.11. Исходные данные и результаты определения актентов и вариантов использования

Идентификация актантов и вариантов использования — это определяющий вид деятельности для получения правильных требований, за которые отвечает системный аналитик (рис. 7.11). Но системный аналитик не может делать эту работу в одиночку. Аналитик получает данные от группы, в которую входят клиент, пользователи системы и другие аналитики, работающие в отделе анализа под руководством системного аналитика.

Иногда имеется готовая бизнес-модель, и можно начинать с нее. Если это так, группа может создать первый вариант модели вариантов использования более или менее «автоматически». В других случаях в качестве исходных данных предлагается модель предметной области, детальное техническое задание, включающее основные требуемые опции, или вообще краткая концепция. Также в качестве исходных данных могут быть выданы дополнительные требования, которые нельзя привязать к отдельным вариантам использования. Эти артефакты исходных данных подробно обсуждались в главе 6.

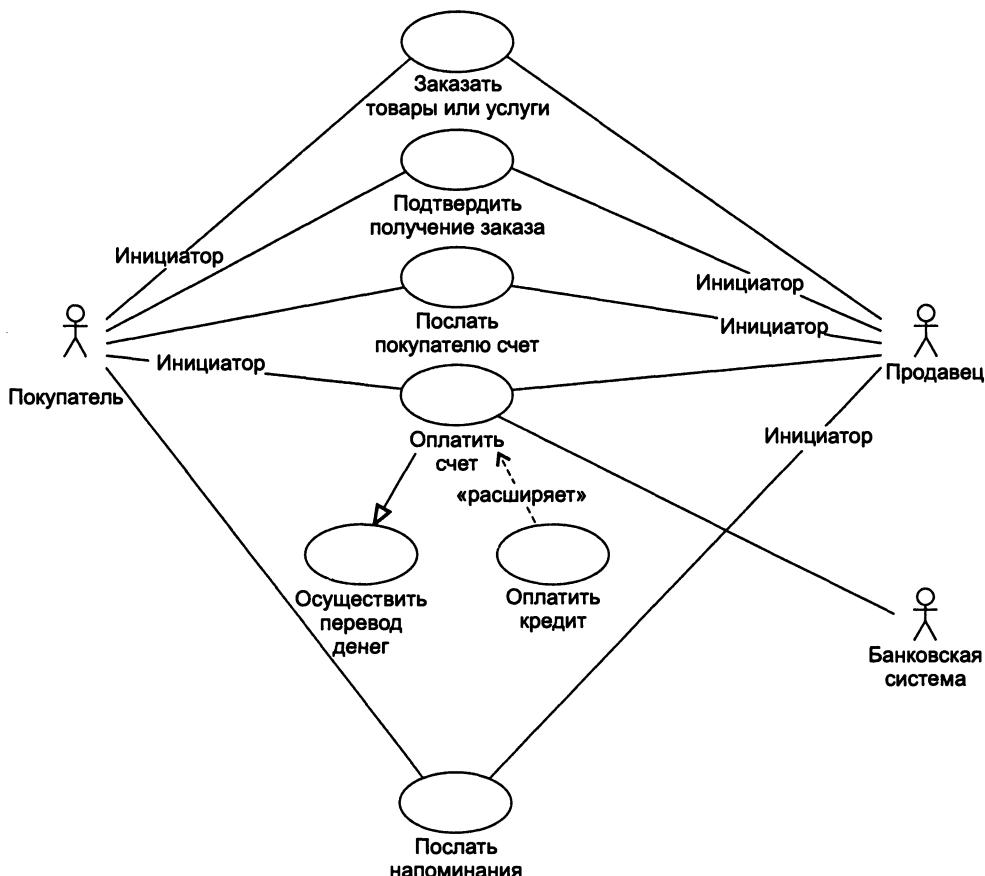


Рис. 7.12. Варианты использования Биллинговой и Платежной системы, поддерживающие бизнес-вариант использования Продажа:
От Заказа до Поставки

Этот вид деятельности выполняется в четыре этапа.

1. Идентификация актантов.
2. Выделение вариантов использования.
3. Идентификация описания каждого варианта использования.
4. Описание модели вариантов использования в целом (этот этап также включает в себя разработку глоссария).

Эти этапы не имеют какого-то определенного порядка выполнения. Часто их выполняют одновременно. Например, диаграммы вариантов использования модифицируются сразу же, как только удается выделить новый актант или вариант использования.

Результатом этой деятельности будет новая версия модели вариантов использования с новыми или измененными актантами и вариантами использования. Для того чтобы можно было подробно описать каждый вариант использования, должен быть схематически описан и изображен итоговый артефакт этой деятельности — модель вариантов использования. Это будет уже следующим действием — *Детализировать варианты использования*. Рисунок 7.12 иллюстрирует этот процесс диаграммой использования (реструктурированной в течение нескольких итераций). На рисунке роль *Инициатор*, присоединенная к ассоциации, указывает, какой из актантов инициирует вариант использования. Вскоре мы поговорим об этом подробнее.

Идентификация актантов

Решение задачи идентификации актантов зависит от исходной точки, с которой мы начинаем поиск. Когда мы начинаем работу с готовой бизнес-модели, поиск актантов прост. Системный аналитик системы может предполагать, что актантами будут каждый бизнес-сотрудник и каждый бизнес-актант (то есть клиент), использующий информационную систему (см. также подраздел «Поиск вариантов использования по бизнес-модели» главы 6).

В противном случае, независимо от того, имеем мы модель предметной области или нет, системный аналитик, работая вместе с клиентом, идентифицирует пользователей и пытается разделить их по категориям. Эти категории и будут актантами.

В любом случае должны быть идентифицированы как актанты, представляющие собой внешние системы, так и внутрисистемные актанты.

При выявлении кандидата в актанты полезны два приема. Во-первых, следует найти хотя бы одного пользователя, который в состоянии оценить кандидата в актанты. Это поможет нам выбирать только настоящие актанты и избегать тех, которые являются плодом нашего воображения. Во-вторых, перекрытие ролей, которые исполняют в системе экземпляры разных актантов, должно быть минимальным. Мы же не хотим, чтобы несколько актантов исполняли, по существу, одинаковые роли. Если это произошло, мы должны или попробовать объединить эти роли в единый набор ролей одного актанта, или попытаться найти другого, обобщенного актанта, которому мы назначим эти общие для перекрывающихся актантов роли. Этого нового актанта можно потом специализировать, используя обобщения. Например, *Покупатель* и *Продавец* будут специализациями актанта *Клиент банка*. Вообще говоря, в первый момент обычно возникает множество актан-

тов со значительным перекрытием. Чтобы получить правильные наборы актантов и обобщений, часто требуется несколько циклов обсуждения.

Системный аналитик идентифицирует актанты и кратко описывает роли каждого актанта и задачи, для которых актанты используют систему. Для передачи требуемой семантики важно найти для актантов осмысленные имена. Краткое описание актанта должно включать в себя обзор его задач и обязанностей.

Пример. Актанты – *Покупатель, Продавец и Банковская система*.

Покупатель. Покупатель представляет собой лицо, имеющее право покупать товары или услуги таким способом, который описан в бизнес-варианте использования *Продажа: От Заказа до Поставки*. Это лицо может быть физическим (то есть не сотрудником компании) или ответственным сотрудником компании. Покупатель товаров и услуг нуждается в Биллинговой и Платежной Системе для того, чтобы посыпать заказы и оплачивать счета.

Продавец. Продавец представляет собой лицо, продающее и поставляющее товары или услуги. Продавец использует систему для просмотра наличия новых заказов и посылки подтверждений получения заказа, счетов и напоминаний об оплате.

Банковская система. Биллинговая и Платежная Система посыпает подтверждения проведенных операций Банковской системе.

Результатом этого шага будет новая версия модели варианта использования с модифицированным набором актантов и кратким описанием каждого из них. Эти кратко описанные актанты теперь могут быть использованы в качестве исходных данных для идентификации вариантов использования.

Идентификация вариантов использования

Когда исходной точкой для нас служит бизнес-модель, мы идентифицируем актантов и варианты использования так, как описали в подразделе «Поиск вариантов использования по бизнес-модели» главы 6. Для каждой роли каждого сотрудника, участвующего в реализации бизнес-варианта использования и использующего информационную систему, выделяется по одному варианту использования. В противном случае системный аналитик идентифицирует варианты использования, применяя информацию, получаемую от клиентов и пользователей. Он перебирает актантов и предлагает для каждого из них кандидатов на варианты использования. Чтобы обнаружить нужные варианты использования, можно устраивать интервью или круглые столы (см. [13]). Актанту для создания, изменения, отслеживания, удаления или просмотра бизнес-объектов, например *Заказов и Счетов*, использующихся в бизнес-вариантах использования, необходимы варианты использования. Актант может также информировать систему о неких внешних событиях, то есть актант может нуждаться в системе для того, чтобы сообщать ей некоторую информацию, например о том, что счет просрочен. Могут также существовать дополнительные актанты, выполняющие запуск системы, ее остановку и техническое обслуживание.

Некоторые из кандидатов так и не станут вариантами использования, но из них могут получиться отличные куски для других вариантов использования. Напомним, что мы стараемся создавать варианты использования, которые можно изменять, рецензировать, тестировать и управлять ими как единым целым.

Мы выбираем для каждого варианта использования такие имена, которые побуждали бы нас думать о конкретной последовательности действий, приносящей ценный для актанта результат. Имя варианта использования часто начинается с глагола или отглагольного существительного, что отражает результат взаимодействия между актантом и системой. В нашем примере имеются варианты использования типа *Оплатить Счет* и *Заказать Товары или Услуги*.

Иногда трудно бывает определить область действия варианта использования. Последовательность взаимодействий системы с пользователем может быть определена или в одном варианте использования, или в нескольких вариантах использования, которые актант инициирует один за другим. Для того чтобы решить, является ли кандидат в варианты использования отдельным вариантом использования, мы должны понять, является ли он самодостаточным или всегда появляется в виде продолжения другого варианта использования. Помните, что варианты использования приносят своим актантам ценный для них результат (см. подраздел «Варианты использования»). Точнее говоря, вариант использования порождает конкретный результат, который является ценным для отдельных актантов. Эта прагматическая характеристика «хорошего» варианта использования может помочь определить присущую варианту использования область действия.

Отметим, что две ключевых фразы в этом руководстве — результирующая ценность и отдельный актант — представляют собой два полезных критерия для обнаружения вариантов использования.

Результирующая ценность. Каждый успешно выполненный вариант использования должен принести актанту некоторый ценный для него результат, так, чтобы задачи актанта были выполнены [15]. В некоторых случаях актант в ответ оплачивает полученный результат. Заметьте, что экземпляр варианта использования, например телефонный звонок, может происходить при участии более одного актанта. В этом случае критерий «ценности конкретного результата» должен применяться к актанту, инициировавшему вариант использования. Этот критерий «ценности конкретного результата» предупреждает нас от выбора незначительных вариантов использования.

Пример. Область действия варианта использования Оплатить счет. Биллинговая и Платежная система предлагает вариант использования *Оплатить счет*, который должен использоваться покупателем для указания системе оплатить счета за товары, которые он заказал и получил. После этого вариант использования *Оплатить счет* выполняет оплату в указанный срок платежа.

Вариант использования *Оплатить счет* включает в себя пометку и собственно выполнение оплаты. Если бы мы разбили вариант использования на два: один — *Пометить к оплате*, а второй — *Выполнить оплату*, то *Пометить к оплате* не имел бы отдельной ценности. Актант получает результат в момент оплаты счета (после этого нам не придется раздражаться, получая все эти напоминания о необходимости заплатить).

Отдельный актант. Выделяя те варианты использования, которые добавляют стоимость реальным людям — пользователям, мы удостоверяемся в том, что варианты использования не слишком разрослись.

Что касается актантов, то варианты использования, которые мы выбрали сначала, до того, как устоялась модель варианта использования, будут, по всей вероятности, неоднократно реструктурированы и пересмотрены.

Как мы обсуждали в главе 4, варианты использования и архитектура системы разрабатываются итеративно. Как только мы определимся с архитектурой, находимые нами новые варианты использования должны подходить под существующую архитектуру. Варианты использования, которые невозможно приспособить к выбранной архитектуре, следует подогнать под существующую архитектуру (может быть, нам придется также изменить архитектуру для удобства встраивания в нее новых вариантов использования). Например, в начале работы по определению варианта использования мы имели некоторое представление о взаимодействии пользователя с компьютером. Как только мы выберем конкретную среду графического интерфейса пользователя, нам, вероятно, придется менять варианты использования, хотя вносимые изменения в этом случае обычно минимальны.

Могут потребоваться и более серьезные изменения. Системный аналитик может предложить способ наблюдать за загрузкой системы, определив симулятор, который действует как главный актант в вариантах использования по опросу системы. Таким образом вы можете измерять время отклика и другие параметры производительности системы. Вы можете также измерить время, на которое вариант использования тормозится во внутренних очередях системы. Эти два подхода нередко дают близкие значения, но затраты на их выполнение могут быть, в зависимости от существующей архитектуры, весьма различны. Таким образом, системный аналитик может нуждаться в повторном обсуждении требований (вариантов использования и прочее) с клиентом, чтобы иметь возможность создать лучшую систему, более простую в разработке и обслуживании. Клиенту также будет выгодно заново обсудить требования и, может быть, получить необходимую функциональность быстрее, с более высоким качеством и по более низкой цене.

Краткое описание каждого варианта использования

Когда аналитики выделяют варианты использования, они иногда пишут пару слов в пояснение каждого варианта использования, а иногда только записывают их названия. Позже они кратко описывают каждый вариант использования. Сначала в описание входят несколько предложений, суммирующих действия варианта использования, позже, шаг за шагом, к нему добавляются описания действий системы при взаимодействии с ее актантами.

Пример. Первоначальное описание варианта использования Оплатить счет.

Краткое описание. Вариант использования *Оплатить счет* используется *Покупателем* для пометки счетов к оплате. Затем вариант использования *Оплатить счет* осуществляет платеж в назначенную дату.

Первоначальное пошаговое описание. Прежде чем этот вариант использования может быть инициирован, *Покупатель* должен получить счет (этот артефакт является результатом выполнения другого варианта использования, с именем *Выписать Счет Покупателю*) и получил заказанные товары или услуги:

- покупатель рассматривает счет, предъявленный к оплате, и находит его соответствующим первоначальному заказу;
- покупатель помечает счет к оплате для банка;
- когда настает день оплаты, система выясняет, достаточно ли денег на счете *Покупателя*. Если да, то транзакция совершается.

Мы кратко описали актанты и варианты использования. Однако мало описать и понять каждый вариант использования в отдельности. Мы также должны увидеть всю картину. Мы должны понять, как варианты использования и актанты связаны друг с другом и как вместе они образуют модель вариантов использования.

Описание Модели варианта использования в целом

Чтобы объяснить модель варианта использования в целом, и особенно влияние вариантов использования друг на друга и на актанты, мы создаем диаграммы и описания.

Нет строгих правил того, что включать в диаграмму. Мы сами выбираем набор диаграмм, которые будут наиболее ясно описывать систему. Например, могут быть нарисованы диаграммы для показа вариантов использования, которые участвуют в одном бизнес-варианте использования (рис. 7.12), или, возможно, иллюстрации вариантов использования, выполняемых одним актантом.

Чтобы гарантировать совместимость при параллельном описании отдельных вариантов использования, разумно будет разработать глоссарий понятий. Эти понятия можно получить из (и проследить по) классам модели предметной области или бизнес-модели, которые описаны в главе 6.

Модель вариантов использования не обязательно является плоской моделью, наподобие описанной здесь. Она может также быть организована в виде кластеров вариантов использования, называемых пакетами вариантов использования [57].

Этот шаг дает на выходе также обзорное описание модели вариантов использования. Это описание поясняет модель вариантов использования в целом. Оно описывает, как взаимодействуют между собой актанты и варианты использования и как варианты использования связаны друг с другом. В представлении модели вариантов использования по UML обзорное описание — это именованное значение самой модели (см. приложение А, раздел А.1.2).

Пример. *Обзорное описание.* Обзорное описание Биллинговой и Платежной системы (см. рис. 7.12) может выглядеть подобно приведенному ниже тексту. В это описание мы включили комментарии, записанные в конце примера.

Покупатель применяет вариант использования *Заказать Товары или Услуги*, чтобы просмотреть пункты заказа и цены, составить заказ и отправить его продавцу.

Сразу же или позднее покупателю поставляются товары или услуги вместе со счетом.

Покупатель активирует вариант использования *Оплатить Счет*, чтобы согласиться с полученным счетом и пометить его к оплате. В намеченный день вариант использования *Оплатить Счет* автоматически перечислит деньги со счета покупателя на счет продавца (комментарий 1).

Кроме того, вариант использования *Оплатить Счет* расширен вариантом использования *Оплатить Кредит*, если оплата по текущему счету превысила сумму, находящуюся на счете, и потребовалось кредитование (комментарий 2).

Теперь рассмотрим, как может использовать систему продавец. Продавец может рассматривать ее, предлагать внесение изменений и подтверждать полученные заказы, применяя вариант использования *Подтвердить Заказы*. Подтвержденный заказ будет сопровождаться поставкой заказанных товаров или услуг (этот вариант использования не описан в нашей модели, поскольку он происходит вне Биллинговой и Платежной системы).

После того как товары или услуги поставлены, продавец выставляет покупателю счет с помощью варианта использования *Выставить Счет Покупателю*. При выписке счета продавец может применить индивидуальные цены, скидки, а также объединить несколько счетов в один.

Если покупатель не оплатил платеж в срок, продавец получает соответствующую информацию и может использовать вариант использования *Послать Напоминание*. Система могла бы посыпать напоминания автоматически, но мы выбрали такое решение, в котором продавец имеет возможность просмотреть напоминания перед отсылкой, чтобы избежать неловкостей в обращении с клиентами (комментарий 3).

Комментарии.

1. Напомним, что модель вариантов использования — это нечто большее, чем просто список вариантов использования. Она также описывает обобщения, присутствующие в вариантах использования. Например, последовательность действий по оплате в варианте использования *Оплатить Счет* может применяться многими вариантами использования (даже если на рис. 7.12 показывается только одно обобщение). Это обобщение может быть представлено отдельным вариантом использования по имени *Осуществить перевод денег*, который повторно применяется такими отдельными вариантами использования, как *Оплатить Счет*. Обобщение означает, что последовательность действий, описанная в варианте использования *Осуществить перевод денег*, вставлена в последовательность, описанную в *Оплатить Счет*. Когда система выполняет экземпляр варианта использования *Оплатить Счет*, экземпляр будет также выполнять действия, описанные в варианте использования *Осуществить перевод денег*.
2. Когда система выполняет экземпляр варианта использования *Оплатить счет*, последовательность действий может быть расширена, чтобы включить в себя последовательность действий, описанную в дополнительном варианте использования *Оплатить Кредит*. Мы кратко рассмотрели отношения обобщения и расширения, чтобы показать, что модель варианта использования может быть структурирована для упрощения определения и понимания полного набора функциональных требований. Подробную информацию по этому вопросу можно найти в [36].
3. *Послать Напоминание* поясняет вариант использования, который упрощает коррекцию путей в бизнес-вариантах использования. Такие корректирующие пути помогают процессу «снова встать на след», они в состоянии предотвратить превращение маленького вопроса во взаимоотношениях с клиентом в реальную проблему. Таким образом, актанты также нуждаются в вариантах использования (или альтернативах для основных вариантов использования), которые помогали бы им корректировать отклоняющиеся пути процесса. Этим видом вариантов использования часто реализуется значительная часть требований в системах, имеющих сильно разветвленную логику работы.

Обсуждение.

Существует несколько способов создания модели вариантов использования, но этот пример поясняет только один из них. Обсудим некоторые вариации модели, которую мы построили. Что, если покупатель мог бы, просмотрев в Интернете ка-

талог доступных товаров или услуг, сделать интерактивный заказ и сразу же получить подтверждение? Был бы нам по-прежнему необходим отдельный вариант использования *Подтвердить Заказ*? Нет, так как автоматическое подтверждение заказа было бы включено в вариант использования *Заказать Товары или Услуги*.

Однако в нашем примере мы считали, что после того как заказ был просмотрен продавцом, он или подтверждается, или продавец предлагает некую альтернативу. Например, продавец может предложить альтернативный набор товаров, которые столь же полезны, но дешевле или могут быть быстрее поставлены. Фактическая последовательность при отправке заказа может в этом случае быть рядом предложений продавца сделать альтернативный заказ с последующим его подтверждением, как показано ниже.

- Покупатель отправляет первичный заказ;
- продавец отсылает покупателю предложение по альтернативному заказу;
- покупатель отсылает итоговый заказ;
- продавец отсылает покупателю подтверждение заказа.

Эти шаги охватывают два варианта использования: *Заказать Товары или Услуги* и *Подтвердить Заказ*. Почему не четыре отдельных варианта использования или один общий вариант использования? Рассмотрим сначала, почему эти шаги не составляются в один общий вариант использования. Мы не хотим вынуждать продавца и покупателя взаимодействовать в режиме реального времени ни в коем случае. Мы хотим, чтобы они могли пересыпать друг другу запросы, не дожидаясь друг друга. Мы можем предположить, что продавец хочет собрать несколько новых заказов от различных покупателей, а затем разом просмотреть и подтвердить их все. Это было бы невозможно, если бы все четыре шага представляли бы собой один вариант использования, потому что каждый вариант использования считается неделимым и, следовательно, должен быть закончен до того, как начнется другой вариант использования. В результате продавец не может начать обрабатывать пришедшие к нему новые заказы (после шага 1) в то время, когда он ожидает прихода итогового заказа в ответ на свои предложения (после шага 2).

Из четырех шагов можно, конечно, сделать четыре разных варианта использования, но начальный и итоговый заказ (шаги 1 и 3) настолько похожи, что они могут быть представлены как альтернативы для варианта использования *Заказать Товары и Услуги*. В конце концов, мы не желаем, чтобы набор вариантов использования разросся до размеров, превышающих пределы нашего понимания. Слишком много вариантов использования делают модель вариантов использования трудной для понимания и использования при анализе и проектировании.

Точно так же предложения по альтернативному заказу (шаг 2) включают в себя подтверждение части заказа и предложения альтернатив для остального, а подтверждение заказа (шаг 4) включает в себя подтверждение всех частей заказа. Оба эти шага могут быть выражены одним вариантом использования, *Подтвердить Заказ*, который позволяет продавцу подтверждать заказ частями и предлагать альтернативы. Итак, будет существовать вариант использования *Заказать Товары или Услуги*, который обеспечивает действия, соответствующие шагам 1 и 3, и вариант использования *Подтвердить Заказ*, который соответствует шагам 2 и 4.

Когда описание модели вариантов использования готово, мы привлекаем людей, не входящих в группу разработчиков (например, пользователей и клиентов),

чтобы они просмотрели модель вариантов использования и провели неформальное рецензирование, чтобы определить, что:

- Все необходимые функциональные требования определены как варианты использования.
- Последовательность действий для каждого варианта использования правильна, полна и понятна.
- Все варианты использования, приносящие незначительный или нулевой результат, были обнаружены и перестроены.

Деятельность: Определение приоритетности вариантов использования

Цель этого вида деятельности состоит в том, чтобы получить данные по приоритетности вариантов использования. Эти данные необходимы для определения того, какие из вариантов использования должны быть разработаны (проанализированы, сконструированы, созданы и т. д.) на ранних итерациях, а какие можно отложить и на более поздние (рис. 7.13).

Результаты фиксируются в архитектурном представлении модели вариантов использования. Это представление затем обсуждается с руководителем проекта и используется при планировании порядка разработки частей проекта по итерациям. Заметьте, что это планирование также должно учитывать и нетехнические факторы, такие как деловые или экономические аспекты разрабатываемой системы (см. подраздел «Планирование итераций» главы 12).

Архитектурное представление модели вариантов использования должно описывать варианты использования, критичные для архитектуры системы. Это подробно рассматривается в подразделе «Описание архитектуры (Представление модели вариантов использования)».

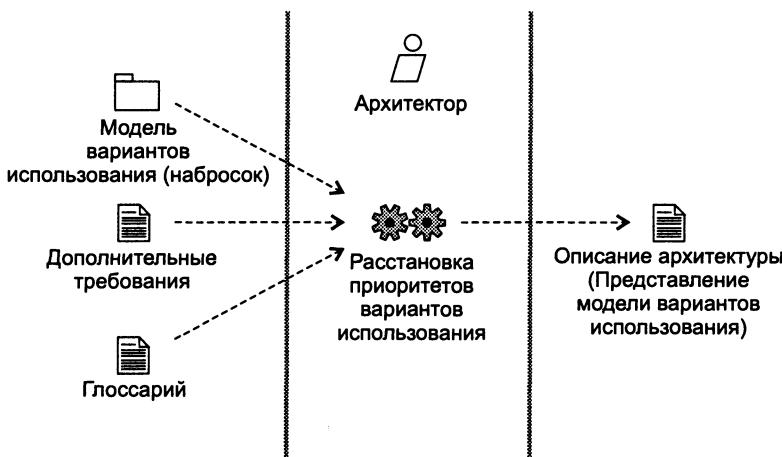


Рис. 7.13. Исходные данные и результат расстановки приоритетов вариантов использования

Деятельность: Детализация вариантов использования

Главная цель детализации каждого варианта использования состоит в том, чтобы подробно описать его поток событий, в том числе запуск, окончание и взаимодействие с актантами (рис. 7.14).

Используя в качестве отправной точки модель вариантов использования и связанные с ней диаграммы вариантов использования, спецификаторы вариантов использования могут описать варианты использования подробно. Спецификаторы вариантов использования превращают пошаговое описание каждого варианта использования в точную спецификацию последовательности действий. В этом пункте мы рассмотрим:

- как структурировать описание для нахождения всех альтернативных путей варианта использования;
- что включать в описание варианта использования;
- как при необходимости формализовать описание варианта использования.

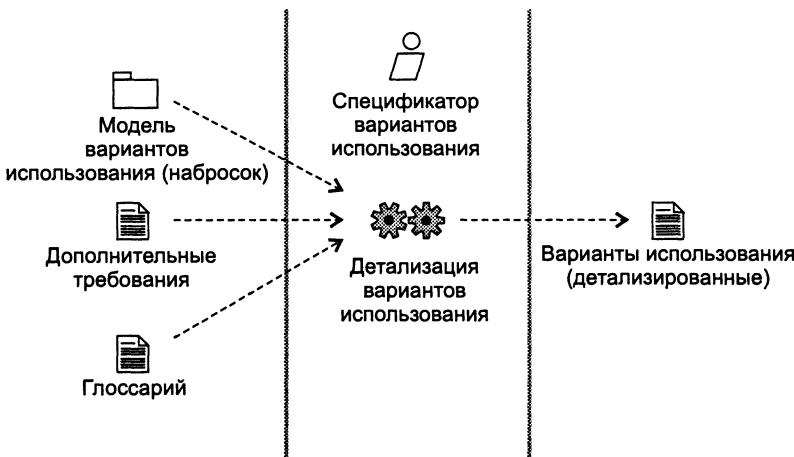


Рис. 7.14. Исходные данные и результат детализации вариантов использования

Каждый спецификатор вариантов использования должен работать в связке с теми людьми, которые непосредственно работают с этими вариантами использования. Спецификатор вариантов использования должен интервьюировать пользователей, записывать их понимание вариантов использования, обсуждать с ними предложения и просить их рецензировать описания вариантов использования.

Результатом этой деятельности будет детальное описание конкретных вариантов использования в текстовом виде и диаграммах.

Структурирование описания варианта использования

Мы уже упоминали, что вариант использования определяет состояния, в которых могут находиться экземпляры варианта использования и возможные переходы между этими состояниями (рис. 7.15). Каждый такой переход — это последователь-

ность действий, которая выполняется экземпляром варианта использования в случае некоторого события, например, получения сообщения.

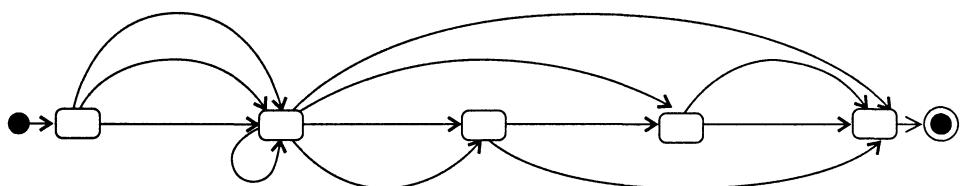


Рис. 7.15. Вариант использования в виде графа переходов между состояниями

В соответствии с рис. 7.15 вариант использования можно представлять имеющим начальное состояние (крайний левый прямоугольник), промежуточные состояния (средние прямоугольники), конечное состояние (крайний правый прямоугольник) и переходящим из одного состояния в другое (систему обозначений см. в приложении А). Прямые стрелки на рисунке указывают основной путь, загнутые — другие возможные пути. Граф переходов между состояниями, изображенный на рис. 7.15, может стать совершенно запутанным. Однако мы должны описать возможные переходы из состояния в состояние (последовательности действий) просто и точно. Правильным будет выбрать один полный основной путь (прямые стрелки на рис. 7.15) из начального состояния в конечное и описать в виде отдельного пункта. После этого можно описать остальные пути (загнутые стрелки) как альтернативы или отклонения от основного пути, каждый в отдельном пункте. Иногда, однако, альтернативы или отклонения достаточно невелики, чтобы их можно было встроить в описание основного пути. Здравый смысл укажет нам, что предпочтеть — встроенное описание альтернативы или отдельный пункт описания. Напомним, что наша цель — создать точное описание, которое достаточно просто читать. Любым выбранным методом мы должны описать все альтернативы, в противном случае вариант использования не считается описанным.

Альтернативы, отклонения от основного пути и исключительные ситуации могут порождаться многими причинами.

- Актант может выбирать различные пути осуществления варианта использования. Например, в варианте использования *Оплатить Счет* актант может решить оплатить счет или отказать в оплате.
- Если в варианте использования участвуют несколько актандов, действия одного из актандов могут влиять на деятельность других актандов.
- Система может обнаружить ошибки, сделанные актандом при вводе.
- Неисправность некоторых системных ресурсов может не позволить системе успешно выполнять ее работу.

В качестве основного должен выбираться «нормальный» путь, то есть тот, который пользователи считают наиболее типовым и приносящим актанду наиболее очевидный результат. Обычно такой основной путь должен включать в себя некоторое количество исключений и специальных случаев, редко встречающихся в работе.

Пример. *Пути варианта использования Оплатить Счет.* Заметьте, пожалуйста, как изменился этот текст по сравнению с предварительным вариантом, приведенным ранее в этой главе, когда у нас был только набросок описания варианта использования (см. подраздел «Краткое описание каждого варианта использования»). Это изменение показывает, как мы детализируем варианты использования посредством моделирования. Реальные полные описания вариантов использования имеют больший размер и содержат описание большего количества путей.

Предусловие: покупатель получил заказанные товары или услуги и по крайней мере один счет. Теперь он планирует пометить счет(а) к оплате.

Поток событий.

1. Основной путь.

1. Покупатель запускает вариант использования, начиная просматривать счета, полученные от системы. Система проверяет, что содержимое счетов соответствует подтверждениям заказов, полученным ранее (как часть варианта использования *Подтвердить Заказ*), и указывает на это покупателю. Подтверждение заказа описывает, что будет поставлено, когда, куда и по какой цене.
2. Покупатель решает пометить счет к оплате, и система генерирует запрос на платеж, чтобы банк мог перечислить деньги на счет продавца. Заметьте, что покупатель не может пометить один и тот же счет к оплате дважды.
3. Позже, а именно в намеченную дату, если на счете покупателя имеется достаточное количество денег, происходит оплата. При оплате деньги перечисляются со счета покупателя на счет продавца, как описано в абстрактном варианте использования *Совершить Перечисление* (который используется в *Оплатить Счет*). Покупатель и продавец уведомляются о результате операции. Банк получает плату за перечисление, которая снимается системой со счета покупателя.
4. Экземпляр варианта использования прекращает свое существование.

2. Альтернативные пути.

1. На шаге 2 покупатель может потребовать, чтобы система отослала продавцу сообщение о том, что счет отклонен.
2. На шаге 3, если на счете недостаточно денег, вариант использования отменит оплату и сообщит об этом покупателю.

Постусловие: образец варианта использования прекращает свое существование после того, как счет оплачен, или если оплата отменена и деньги не перечислены.

Так как варианты использования должны быть понятны и разработчикам, и клиентам, и пользователям, они должны всегда описываться на простом английском¹ языке, как показано в этом примере.

Что включать в описание варианта использования

Как показывает предшествующий пример, описание варианта использования должно включать в себя.

¹ Или другом естественном языке.

- Определение начального состояния (см. рис. 7.15) в виде предусловия.
- Описание, как и когда запускается вариант использования (то есть первое выполняемое действие, шаг 1).
- Требуемый порядок (если он существует), в котором должны выполняться действия. Здесь порядок определен последовательностью пронумерованных шагов (шаги 1–4).
- Определение, как и когда вариант использования оканчивается (шаг 4).
- Возможные конечные состояния (см. рис. 7.15) как постусловия.
- Запрещенные пути выполнения. Примечание к шагу 2 сообщает нам о пути, который невозможен — *Оплатить счет дважды*. Это путь, который пользователь выбрать не может.
- Описания альтернативных путей, которые встроены в описание основного пути. Весь шаг 3 — это действие, которое выполняется только при наличии на счете достаточной суммы денег.
- Описания альтернативных путей, которые были отделены от описания основного пути (шаг 5).
- Взаимодействие актантов с системой и сообщения, которыми они обмениваются (шаги 2 и 3). Типичный пример — *Покупатель* решает пометить счет к оплате на шаге 2, и *Покупатель* и *Продавец* получают сообщения о результатах операции на шаге 3. Иначе говоря, мы описываем последовательность действий варианта использования, как эти действия инициируются актантами и как их выполнение вызывает запросы к актантам.
- Использование системой объектов, значений и ресурсов (шаг 3). Другими словами, мы описали последовательность действий для варианта использования и приписали значения атрибутам варианта использования. Типичный пример — перечисление денег со счета покупателя на счет продавца на шаге 3. Другой пример — использование счетов и подтверждений заказов на шаге 1.
- Заметьте, что мы должны явно описать, что делает система (какие действия она выполняет) и что делает актант. Мы должны быть в состоянии отделить обязанности системы от обязанностей актантов, иначе описание варианта использования не будет достаточно точным для использования его в качестве спецификации на систему. Например, на шаге 1 мы пишем: «система проверяет, что содержимое счетов соответствует подтверждениям заказов, полученным ранее», а на шаге 3 — «плата за перечисление снимается системой со счета покупателя».

Атрибуты варианта использования могут использоваться как подсказка для того, чтобы позже, во время анализа и проектирования, выделять классы и атрибуты. Таким образом, на основе атрибута *Счет* варианта использования образуется класс проектирования *Счет*. В ходе анализа и проектирования мы также обнаружим, какие объекты в каких вариантах использования будут использоваться, но рассматривать этот вопрос в модели вариантов использования нет необходимости. Вместо этого (как обсуждалось в подразделе «Вариант использования») мы сохраним модель вариантов использования простой, запрещая взаимодействия между экземплярами вариантов использования и доступ экземпляров к атрибутам друг друга.

До сих пор мы говорили о функциональных требованиях и представлении их в виде вариантов использования, но мы должны также определить и нефункциональные требования. Большинство нефункциональных требований связаны с определенным вариантом использования, как-то: производительность, точность, время срабатывания, время восстановления или использование памяти, с которыми система должна выполнять данный вариант использования. Такие требования добавляются к рассматриваемому варианту использования в виде специальных требований (именованные значения в UML). Они могут быть зафиксированы в отдельном пункте описания варианта использования.

Пример. *Специальное требование (производительность).* Когда покупатель помечает счет к оплате, система должна выдать результат проверки менее чем за 1.0 секунду в 90% случаев. Время проверки никогда не должно превышать 10 секунд, если только связь с сетью не прервана (в этом случае пользователь должен быть извещен).

Если система взаимодействует с другой системой (актантом – не человеком), необходимо определить это взаимодействие (например, ссылкой на стандартный коммуникационный протокол). Это должно быть сделано на ранних итерациях фазы проектирования, так как способ реализации межсистемной связи обычно сильно влияет на архитектуру.

Описания варианта использования считаются законченными, когда они признаются понятными, правильными (то есть включающими правильные требования), полными (описывающими все возможные пути) и непротиворечивыми.

Описания оцениваются аналитиками, пользователями и клиентами на совещании по рецензированию, которое проходит в конце процесса определения требований. Только клиенты и пользователи могут определить, верны ли варианты использования.

Формализация описания варианта использования

Рисунок 7.15 поясняет, как экземпляры варианта использования в соответствии с переходами перемещаются из одного состояния в другое. Если в вариант использования входит всего несколько состояний, мы не всегда создаем явное описание этих процессов. Можно воспользоваться стилем примера *Оплатить Счет*. Однако всегда неплохо при описании варианта использования держать в голове конечный автомат, поскольку это гарантирует нам охват всех возможных случаев. Иногда, однако, например в комплексных системах реального времени, варианты использования могут быть настолько сложны, что необходимо использовать более структурированный метод описания. Взаимодействие между актантами и вариантами использования может, например, включать так много состояний и альтернативных переходов, что почти невозможно сохранять текстовое описание варианта использования непротиворечивым. Поэтому для описания вариантов использования полезно использовать методы визуального моделирования. Эти методы помогут системному аналитику лучше понять варианты использования.

- Для описания состояний варианта использования и переходов между этими состояниями могут использоваться диаграммы состояний UML (рис. 7.16).
- Для детального описания переходов между состояниями как последовательности действий могут использоваться диаграммы деятельности. Диаграммы

деятельности могут быть описаны в виде обобщенной формы диаграмм переходов [14]. Это хорошо известный метод описания, используемый в телекоммуникации.

- Для описания процесса взаимодействия экземпляра варианта использования с экземпляром актанта могут использоваться диаграммы взаимодействия. Диаграмма взаимодействия показывает вариант использования с участвующим в нем актантом (или актантами).

Диаграммы состояний, деятельности и взаимодействий пояснены в работах [10, 11, 57, 25].

Пример. Использование диаграмм состояний для описания вариантов использования. Рисунок 7.16 – диаграмма состояний варианта использования *Оплатить Счет*. Черная точка в верхней части графика обозначает начало варианта использования. Это место, из которого начинает выполняться конечный автомат, когда зарождается экземпляр варианта использования. Стрелка, выходящая из черной точки, показывает, в какое состояние немедленно после зарождения переходит конечный автомат, в данном случае – в первое состояние *Просмотр*. Состояния изображаются прямоугольниками со скругленными углами. Изменения состояний изображаются стрелками, направленными из одного состояния в другое.

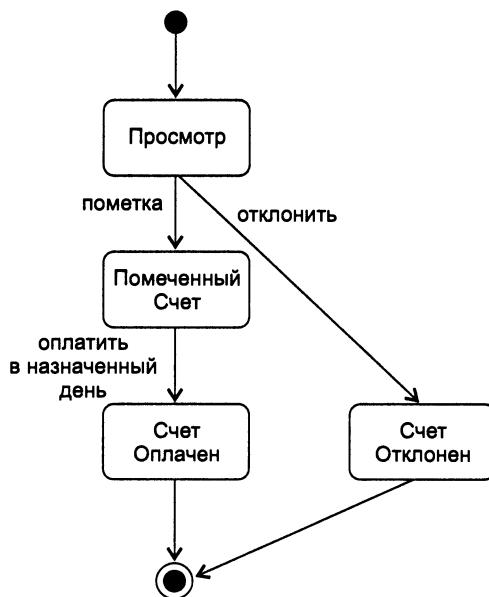


Рис. 7.16. Диаграмма состояний для варианта использования Оплатить Счет

Диаграмма состояний схематически показывает, как экземпляр варианта использования *Оплатить Счет* проходит отдельные состояния (скругленные прямоугольники) в ходе последовательных переходов (стрелки). Сначала пользователь просматривает счет (см. шаг 1 в предшествующем примере *Оплатить Счет*) и решает, пометить его к оплате (см. шаг 2) или отклонить (см. шаг 5). Вариант использования выходит из состояния *Счета, Помеченного к Оплате*, когда пome-

ченный счет оплачивается в срок, указанный для платежа (см. шаг 3). Вариант использования прекращает существовать (круг с черной точкой в нем) сразу же после того, как перейдет в состояния *Счет Оплачен* или *Счет Отклонен*.

Отметим, что использование этих диаграмм в контексте варианта использования может приводить к большим и сложным диаграммам, которые очень трудно читать и понимать. Например, единственный вариант использования может включать в себя множество состояний, которым нелегко дать значащие имена. Это особенно сложно, если диаграммы должны читать люди, которые не входят в состав команды разработчиков. Кроме того, разрабатывать детальные диаграммы и сохранять их потом совместимыми с другими моделями системы – дело недешевое.

Итак, наша основная рекомендация: эти виды диаграмм следует использовать осторожно, и нередко можно будет ограничиться исключительно текстовыми описаниями (описаниями потока событий) варианта использования. Кроме того, во многих случаях текстовые описания и диаграммы могут дополнять друг друга.

Деятельность: Создание прототипа интерфейса пользователя

Цель этого вида деятельности в том, чтобы сформировать прототип интерфейса пользователя (рис. 7.17).

К моменту начала этой деятельности системный аналитик уже разработал модель вариантов использования. В этой модели определено, какие пользователи имеются у системы и для чего данные пользователи будут использовать систему. Это было сделано при помощи диаграмм использования, обзорных описаний модели вариантов использования и детальных описаний для каждого варианта использования.

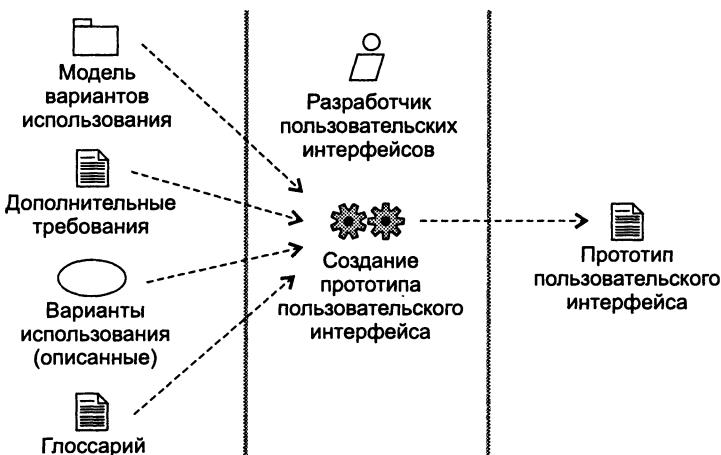


Рис. 7.17. Исходные данные и результат прототипирования пользовательских интерфейсов

Теперь мы должны спроектировать пользовательские интерфейсы, которые позволят пользователю эффективно выполнять варианты использования. Мы сде-

ляем это поэтапно. Начнем с вариантов использования и попробуем выделить информацию, необходимую для интерфейсов пользователя, которые предоставляют доступ к вариантам использования для каждого актанта. Иначе говоря, создадим логический проект пользовательского интерфейса. Затем мы создаем физический проект интерфейса пользователя, то есть проектируем и разрабатываем прототипы, поясняющие, как пользователи с помощью системы могут выполнить варианты использования [1]. Начнем реализацию интерфейса пользователя с определения необходимых действий. Мы должны понять потребности пользователей прежде, чем мы будем их реализовать [там же].

Итоговый результат этой деятельности представляет собой набор эскизов и прототипов интерфейса пользователя, которые определяют вид пользовательских интерфейсов для основных актантов.

Создание логического проекта пользовательского интерфейса

Когда актанты взаимодействуют с системой, они манипулируют элементами пользовательского интерфейса, которые представляют собой не что иное, как атрибуты вариантов использования. Часто эти элементы соответствуют понятиям, описанным в глоссарии (например, баланс счета, срок платежа, владелец счета). Актанты могут представлять себе эти элементы пользовательского интерфейса как иконки, пункты списка, папки или объекты на двумерной карте и манипулировать ими, выбирая их, перетаскивая с места на место или обращаясь к ним. Разработчик пользовательского интерфейса идентифицирует и определяет эти элементы для одного актанта за один раз, рассматривая все варианты использования, в которых может участвовать актант, и идентифицируя соответствующие элементы интерфейса пользователя для каждого варианта использования. Элементы интерфейса отдельного пользователя могут участвовать во многих вариантах использования, исполняя в каждом из них соответствующую роль. Таким образом, для обеспечения выполнения всех этих ролей должны быть разработаны соответствующие элементы интерфейса пользователя. Для каждого актанта нужно получить ответы на следующие вопросы.

- Какие элементы пользовательского интерфейса необходимы актанту для выполнения варианта использования?
- Как эти элементы должны быть связаны друг с другом?
- Как они будут применяться в различных вариантах использования?
- На что они должны походить?
- Какими следует манипулировать?

Для определения того, какие элементы интерфейса пользователя необходимы в каждом варианте использования, доступном актанту, мы можем задать следующие вопросы:

- Какие из классов предметной области, бизнес-объектов или рабочих модулей подходят для элементов пользовательского интерфейса для варианта использования?
- С какими элементами пользовательского интерфейса работает актант?
- Какие действия актант может производить и какие решения принимать?

- Какое побуждение и какую информацию получает актант перед выполнением каждого действия в варианте использования?
- Какую информацию актант должен обеспечить системе?
- Какую информацию система должна обеспечить актанту?
- Каковы средние значения для всех параметров ввода-вывода? Например, с каким количеством счетов актант будет обычно работать в течение сеанса, каков средний баланс счета? Мы нуждаемся в приблизительных оценках этих величин для того, чтобы оптимизировать графический интерфейс пользователя под эти данные (однако должны оставить возможность их изменения в достаточно широком диапазоне).

Пример. Элементы пользовательского интерфейса, входящие в вариант использования *Оплатить Счет*. На примере варианта использования *Оплатить Счет* мы покажем, как определить, с какими элементами пользовательского интерфейса необходимо работать актанту, а также в каких случаях актанту необходимо с ними работать.

Актанту, конечно, понадобится работать с элементами пользовательского интерфейса типа *Счета* (взятыми из класса предметной области или бизнес-сущности). Следовательно, как показано на рис. 7.18, *Счет* будет элементом пользовательского интерфейса. Заметьте, что счета имеют атрибуты *Срок платежа*, *Сумма к оплате* и *Банковский счет получателя*. Все эти атрибуты необходимы актанту, который должен решить, оплачивать ли ему данный счет.

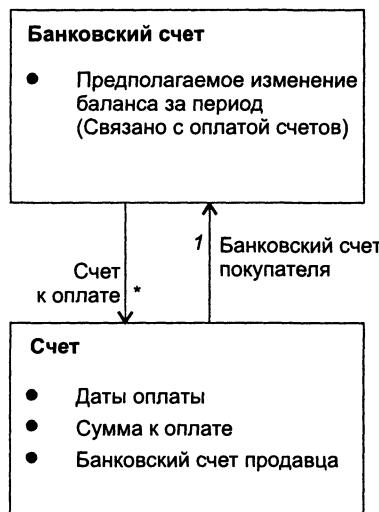


Рис. 7.18. Элементы пользовательского интерфейса Счет и Банковский счет с некоторыми из их атрибутов

Кроме того, когда актант решает, какой счет оплатить, он может захотеть отследить сумму денег на своем банковском счете, чтобы избежать перерасхода. Это пример действий и той информации, в которой нуждается актант. Интерфейс пользователя должен, следовательно, показывать счета, оплата которых намечена на бу-

дущее, и то, как намеченная оплата счетов будет воздействовать на баланс счета (что обозначено ассоциацией *Банковский счет покупателя*, полученной из ассоциации класса предметной области *Покупатель* в главе 6). *Банковский счет*, таким образом, будет еще одним аспектом интерфейса пользователя. Баланс по счету и то, какие в нем ожидаются изменения по причине оплаты счетов, обозначены на рис. 7.18 в виде атрибута счета и ассоциации *Счет к Оплате* между элементами пользовательского интерфейса *Банковский счет* и *Счет*.

Практически способ работы состоит в том, чтобы нарисовать элементы пользовательского интерфейса на бумажных наклейках (как показано на рис. 7.18) и на克莱ить их на доску. Это и будет внешний вид пользовательского интерфейса. Затем разработчики пользовательского интерфейса описывают использование этих элементов актантами при работе с вариантами использования. Преимущество использования наклеек в том, что они могут представлять необходимое количество данных. Кроме того, наклейки не выглядят неизменными, их легко передвинуть или вообще выбросить. В таких условиях пользователям проще вносить изменения.

Таким образом, разработчики пользовательского интерфейса удостоверяются, что каждый вариант использования доступен через элементы пользовательского интерфейса. Также они должны удостовериться и в том, что весь набор вариантов использования, доступных актанту, имеет хорошо интегрированный, легкий в использовании и логичный интерфейс пользователя.

Пока мы рассматривали, что получают актанты от пользовательского интерфейса. Теперь рассмотрим, как физический интерфейс пользователя обеспечивает им то, в чем они нуждаются.

Создание проекта и прототипа пользовательского интерфейса

Разработчики пользовательского интерфейса сначала создают наброски совокупностей элементов пользовательского интерфейса, из которых образуются полезные для актандов пользовательские интерфейсы. Затем они прикидывают, какие дополнительные элементы необходимы для объединения различных частей пользовательского интерфейса в единое целое. Этими дополнительными элементами могут быть контейнеры элементов интерфейса (например, папки), окна и управляющие элементы (рис. 7.19). Эти эскизы могут создаваться после, а иногда и одновременно с наклейками, разрабатываемыми при создании логического проекта пользовательского интерфейса.

Основы моделирования вариантов использования

Детальные описания вариантов использования — хорошая отправная точка для проектирования пользовательского интерфейса. Иногда даже чересчур хорошая. Проблема состоит в том, что описания часто содержат неявные решения по пользовательским интерфейсам. Когда проектировщики пользовательских интерфейсов впоследствии вносят предложения по интерфейсам вариантов использования, они могут незаметно для себя ограничить свой кругозор этими неявными решениями. Однако чтобы создать для вариантов использования наилучший пользовательский интерфейс, они должны постараться избежать этой ловушки. Например, вариант использования *Оплатить счет* начинается с «покупатель вызывает вариант использования, начиная работу с просмотром полученных счетов...». Такое описание может ввести разработчика пользовательского интерфейса в заблуждение, и он

разработает пользовательский интерфейс, включающий список полученных счетов, который будет просматривать актант. Но это может оказаться не лучшим способом изучать полученные счета. Покупатели могут считать, что проще просматривать счета менее очевидным способом, например через иконки, распределенные по горизонтали в соответствии с датой оплаты и по вертикали в соответствии с суммой.

Ларри Константин предлагает средство для разрешения проблемы неявных решений в пользовательском интерфейсе [16]. Он рекомендует, чтобы спецификаторы вариантов использования сначала создавали упрощенные варианты использования — базовые варианты использования, которые не содержат никаких неявных решений по пользовательским интерфейсам. Предыдущий пример может, например, быть записан, как «покупатель вызывает вариант использования для того, чтобы изучить полученные счета». Разработчики пользовательского интерфейса могут использовать эти базовые варианты использования как исходные данные для создания интерфейса пользователя, не обременяя себя никакими неявными решениями.

Этот пример — простая иллюстрация того, что нужно сделать с законченным описанием варианта использования, чтобы выделить его основной смысл. На практике приходится делать много больше, чем просто менять слова в описании. Важно избежать преждевременных решений по вопросам:

- метода, представления некоторого элемента пользовательского интерфейса, например, использовать ли список или текст;
- последовательности обхода интерфейса актантом, например ввода одного атрибута перед другим;
- устройств, необходимых для ввода и вывода, например использования мыши для ввода или монитора для вывода.

Затем, при необходимости, спецификаторы варианта использования могут сделать второй цикл описания вариантов использования, чтобы добавить детали, которые не были учтены в базовых описаниях.

Пример. Физический проект и прототип пользовательского интерфейса. Разработчик пользовательского интерфейса делает нижеприведенный набросок физического проекта визуализации баланса счета, на который с течением времени воздействуют намеченные к оплате счета. Счета показываются в виде белых трапеций, которые уменьшат баланс счета, когда, как предполагается, они будут оплачены. Разработчик пользовательского интерфейса выбрал форму трапеции, потому что она достаточно широка, чтобы ее можно было рассмотреть и, при необходимости, выбрать, и в то же время она имеет выделенную точку, которая используется, чтобы указать, когда точно произойдет оплата. Намеченные к оплате счета, например счет за аренду жилья, приведут к уменьшению баланса счета в момент платежа, что и показано на рис. 7.19. Эскиз также поясняет, как деньги добавляются на счет, например, когда его владелец получает зарплату. Также на эскизе показываются управляющие элементы пользовательского интерфейса — кнопки масштабирования и скроллинга. Кнопки прокрутки влево и вправо перемещают «временное окно», изменяя временной интервал просмотра. Актанту в данный момент будут доступны для просмотра изменения баланса только за этот интервал. Кнопки масштабирования Больше и Меньше влияют на масштаб графика.

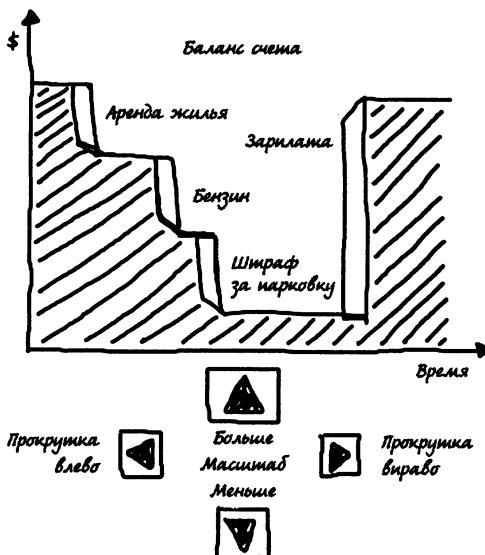


Рис. 7.19. Предлагаемый для связанных элементов Банковский счет и Счет пользовательский интерфейс

Теперь мы готовы сформировать исполняемые прототипы для наиболее важных совокупностей элементов пользовательского интерфейса. Эти прототипы можно создать при помощи какого-либо пакета быстрого макетирования.

Это могут быть несколько прототипов, по одному на актанта, для проверки того, что каждый актант может осуществлять те варианты использования, в которых нуждается. Усилия, затраченные на макетирование, должны соответствовать ожидаемому результату. Мы разрабатываем исполняемые прототипы интерфейса, когда удобство их использования приносит большую ценность (например, прототипы для самых важных актантов), в противном случае ограничиваясь эскизами на бумаге.

Ранняя проверка пользовательского интерфейса путем анализа эскизов и прототипов может предотвратить много ошибок, исправлять которые впоследствии будет недешево. Прототипы помогут также выявить не замеченные ранее дыры в описаниях вариантов использования и дадут возможность исправить их до того, как варианты использования будут применены в проектировании. Рецензенты должны удостовериться, что каждый пользовательский интерфейс:

- позволяет актанту должным образом перемещаться по элементам интерфейса;
- обеспечивает согласованное восприятие и согласованный способ работы с пользовательским интерфейсом, как-то: порядок переходов и акселераторы;
- соответствует общим стандартам, в частности, на цвета, размер кнопок и размещение панелей инструментов.

Отметим, что реализация реального пользовательского интерфейса (в противоположность прототипу, который мы сейчас разрабатывали) создается параллельно с остальными частями системы, то есть в ходе рабочих процессов анализа, про-

ектирования и реализации. Прототип интерфейса пользователя, который мы разработали, будет использован на этих этапах в качестве спецификации интерфейса пользователя. Эта спецификация будет затем реализована в понятиях компонентов качества реализации.

Деятельность: Структурирование модели вариантов использования

Модель вариантов использования структурируется следующим образом.

- Извлекаются общие и совместно используемые описания функциональности, которые могут использоваться в более конкретных описаниях (вариантах использования).
- Извлекаются дополнительные или необязательные описания функциональностей (варианты использования), которые могут расширять более конкретные описания (варианты использования).

До проведения этих действий системный аналитик выделил актанты и варианты использования, описал их диаграммами и объяснил модель вариантов использования в целом. Спецификаторы вариантов использования разработали детальное описание каждого варианта использования. Теперь системный аналитик может реструктурировать набор вариантов использования, чтобы сделать модель проще для понимания и работы с ней (рис. 7.20). Аналитик должен искать совместно используемые варианты поведения и расширения.

Поиск совместно используемых определений функциональности

После того как мы выделили и описали действия в каждом варианте использования, мы должны просмотреть их на предмет наличия действий или частей действий, одинаковых или совместно используемых несколькими вариантами использования.

Для уменьшения избыточности этот совместно используемый фрагмент следует извлечь и описать в отдельном варианте использования, который будет затем применен в первоначальных вариантах использования. Мы показываем это отношение совместного использования при помощи обобщения (в [36] это названо отношением использования). Обобщение вариантов использования — это своего рода наследование; экземпляры обобщенных вариантов использования сохраняют поведение, присущее обобщающему варианту использования. Другими словами, наличие в варианте использования А обобщенного варианта использования В говорит о том, что экземпляр варианта использования А включает в себя поведение варианта использования В.

Пример. Общность вариантов использования. Вернемся к рис. 7.12, приведенному ранее в этой главе, в котором из варианта использования *Оплатить Счет* был выделен многократно используемый вариант использования *Осуществить перевод денег*. Последовательность действий, описанных в варианте использования *Осуществить перевод денег*, таким образом, будет унаследована в последовательности, описанной в *Оплатить Счет* (рис. 7.21).

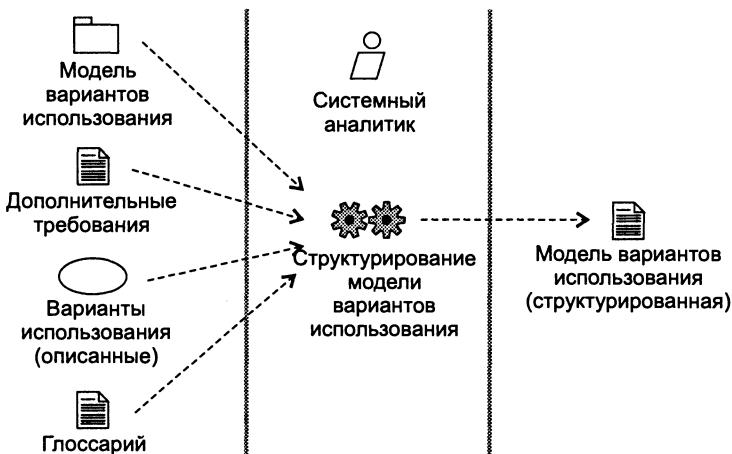


Рис. 7.20. Исходные данные и результат поиска обобщений в модели варианта использования

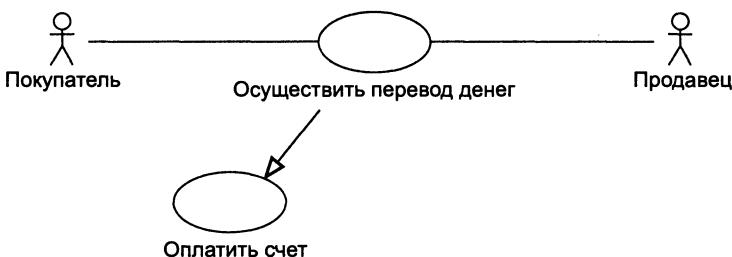


Рис. 7.21. Отношение обобщения между вариантами использования *Оплатить счет* и *Осуществить перевод денег*

Обобщения используются, чтобы упростить работу и понимание модели вариантов использования и многократно применять «заготовки» вариантов использования для создания полных вариантов использования, необходимых заказчику. Такие полные варианты использования называются конкретными вариантами использования. Они инициируются актантом, и их экземпляры включают в себя полную последовательность действий, выполняемых системой. «Заготовки» вариантов использования предназначены только для многократного использования в других вариантах использования и известны под названием абстрактных вариантов использования. Абстрактный вариант использования не существует сам по себе, но экземпляр конкретного варианта использования демонстрирует поведение, описываемое абстрактными вариантами использования, которые он (повторно) использует. Чтобы лучше запомнить, мы назовем этот экземпляр, который актант(ы) наблюдают при взаимодействии с системой, «реальным» вариантом использования.

Пример. «Реальный» вариант использования. Мы можем представить себе «реальный» вариант использования так, как показано на рис. 7.22, где *Оплатить Счет* включает в себя обобщенный вариант *Осуществить перевод денег*.



Рис. 7.22. Экземпляр «реального» варианта использования, собранный из вариантов использования Оплатить Счет и Осуществить перевод денег

Этот «реальный» вариант использования — результат, получаемый после применения обобщения к двум вариантам использования: одному конкретному, а другому абстрактному. Реальный вариант использования представляет поведение экземпляра варианта использования так, как его воспринимает взаимодействующий с системой актант. На рис. 7.22 экземпляр реального варианта использования изображен так, как он воспринимается актантами, Покупателем А и Продавцом В. Если бы модель содержала другие конкретные варианты использования, обобщенные вариантом использования *Осуществить перевод денег*, то имелись бы и другие реальные варианты использования. Эти реальные варианты использования имели бы перекрывающиеся спецификации, причем общей частью было бы то, что определено в варианте использования *Осуществить перевод денег*.

Заметьте, что этот пример не показывает полной картины происходящего. Существует ведь еще вариант использования *Оплатить кредит*, который расширяет вариант использования *Оплата счета*, порождая другие реальные варианты использования. Его мы рассмотрим в следующем пункте.

Поиск дополнительных и возможных описаний функциональности

Другое отношение между вариантами использования — это отношение расширения [36]. Расширение дополняет модель до последовательности действий варианта использования. Расширение ведет себя так, будто оно просто добавлено в первоначальное описание варианта использования. Другими словами, отношение расширения варианта использования А к варианту использования В указывает, что экземпляр варианта использования В может включать в себя (в определенных состояниях, описанных в расширении) поведение, описанное в А. Поведение, описываемое несколькими расширениями единственного основного варианта использования, может происходить внутри одного экземпляра варианта использования.

Отношение расширения включает в себя как условия осуществления расширенного поведения, так и указание на место расширения целевого варианта использования, то есть ту точку в варианте использования, куда подключается расширение. Когда экземпляр (целевого) варианта использования приходит в эту точку, происходит оценка условий расширения. Если условие выполняется, экземпляр варианта использования включает в себя последовательность расширяющего варианта использования.

Пример. Отношение расширения в вариантах использования. Вернемся к рис. 7.12, приведенному ранее в этой главе, и примеру, данному в подразделе «Описание модели варианта использования в целом», в котором вариант использования *Оплатить Счет* был расширен вариантом использования *Оплатить Кредит*. Последовательность действий, описанных в варианте использования *Оплатить Кредит* (рис. 7.23),

вставляется в последовательность, описанную в *Оплатить Счет*, если появляется необходимость в кредите (это и есть условие расширения).

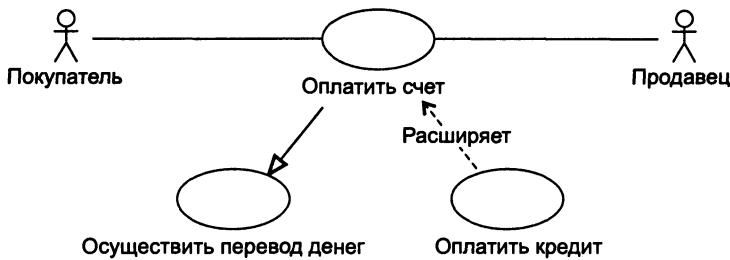


Рис. 7.23. Отношение расширения между вариантами использования Оплатить Счет и Оплатить кредит

Заметим, что мы можем продолжить обсуждение реальных вариантов использования, воспринимаемых актантами с учетом отношения расширения. Применяя отношения расширения варианта использования *Оплатить кредит* к целевому варианту использования (то есть *Оплатить Счет*, обобщенный с помощью *Осуществить перевод денег*), мы получаем новый реальный вариант использования, который является слиянием трех вариантов использования (рис. 7.24). На рисунке реальный вариант использования представлен так, как он воспринимается актантами, Покупателем А и Продавцом В.



Рис. 7.24. Реальный вариант использования, собранный из вариантов использования Оплатить Счет и Осуществить перевод денег и расширенный вариантом использования Оплатить кредит

Мы можем сказать, что реальные варианты использования — варианты использования как они есть — получаются путем применения отношений обобщения и расширения к модельным вариантам использования. Реальные варианты использования — это то, что приносит пользователям ценный для них результат. Таким образом, критерии для хороших вариантов использования, которые были упомянуты в подразделе «Нахождение актантов и вариантов использования» (варианты использования приносят конкретным актантам наблюдаемый и ценный для них результат) подходят только для реальных вариантов использования. Это означает, что мы должны выбрать отдельные критерии для хороших конкретных вариантов использования, абстрактных вариантов использования и расширений вариантов использования. Конкретные варианты использования не должны описывать (значимое) поведение, которое присутствует также и в других конкретных вариантах использования. Абстрактный вариант использования создается только для того, чтобы быть примененным в конкретных вариантах использования, описывая

разделяемое, многократно используемое поведение. Расширяющий вариант использования определяет дополнительное поведение других вариантов использования, независимо от того, является ли это поведение обязательным или нет.

Таким образом, чтобы понять отношения обобщения и расширения, мы ввели понятие реальных вариантов использования. Когда мы выделим конкретные, абстрактные и расширяющие варианты использования, мы можем объединять их, получая реальные варианты использования. Однако, когда мы начинаем моделировать новую систему, мы обычно идем по другому пути. Отталкиваясь от реальных вариантов использования, мы идентифицируем многократно используемое поведение, которое отделяет конкретные варианты использования от абстрактных, и дополнительное поведение, которое мы трактуем как расширение других вариантов использования.

В [36] более основательно рассмотрены отношения обобщения (которые также называют отношениями использования) и расширения, а также вопрос о том, в каком случае следует использовать каждое из них.

Поиск других отношений между вариантами использования

Существуют и другие варианты взаимосвязи между вариантами использования, например отношение включения [57]. Это отношение, которое можно для простоты считать обратным отношению расширения, обеспечивает явные и безусловные расширения варианта использования. Однако при включении варианта использования последовательность поведения и атрибуты включаемого варианта использования закрыты и не могут быть изменены или вызваны — использоваться может только результат (или функция) включенного варианта использования. В этом состоит его отличие от отношения обобщения. В этой книге мы не будем надолго задерживаться на обсуждении данного отношения, но высажем несколько предстережений.

- Структура вариантов использования и их отношений должна в максимально возможной степени отражать реальные варианты использования (как это обсуждалось ранее). Чем дальше эта структура уходит от реального варианта использования, тем труднее будет понять варианты использования и их задачи, и не только внешним группам — пользователям и клиентам, но и самим разработчикам.
- Каждый вариант использования должен рассматриваться как отдельный артефакт. Кто-либо, например спецификатор варианта использования, должен отвечать за его описание, и в последующих рабочих процессах — анализе и проектировании — вариант использования должен быть реализован в отдельной реализации варианта использования (как мы увидим в главах 8 и 9). По этой причине варианты использования не должны быть настолько малы или велики, чтобы для управления ими потребовались значительные усилия.
- Избегайте функциональной декомпозиции вариантов использования в модели вариантов использования. Это лучше делать при уточнении каждого из вариантов использования в аналитической модели. Как мы увидим в главе 8, причина этого в том, что функциональность, определенная вариантами использования, будет разобрана при анализе в объектно-ориентированных традициях

как кооперация концептуальных объектов анализа. Такая декомпозиция при необходимости даст нам более глубокое понимание требований.

Рабочий процесс определения требований: резюме

В этой и предыдущей главах мы рассмотрели, как определяются требования к системе в виде:

- Бизнес-модели или модели предметной области для определения контекста системы.
- Модели вариантов использования для определения функциональных требований и нефункциональных требований, относящихся к отдельным вариантам использования. Модель вариантов использования определяется общим описанием, набором диаграмм и детальным описанием каждого из вариантов использования.
- Набора эскизов и прототипов интерфейса пользователя для каждого актанта, создаваемых разработчиком пользовательских интерфейсов.
- Описания обобщенных дополнительных требований, которые для конкретных вариантов использования не определяются.

Этот результат – очень хорошая отправная точка для следующих рабочих процессов: анализа, проектирования, реализации и тестирования. Варианты использования будут направлять разработку в ходе этих рабочих процессов итерацию за итерацией. Для каждого варианта использования в модели вариантов использования мы будем находить соответствующую реализацию варианта использования в фазах анализа и проектирования и набор тестов в фазе тестирования. Таким образом, варианты использования свяжут различные рабочие процессы.

В главе 8 мы перейдем к следующему шагу в цепи рабочих процессов – анализу, где переформулируем варианты использования как объекты, чтобы лучше понять требования и подготовиться к проектированию и реализации системы.

8 Анализ

Введение в анализ

В ходе анализа мы подвергаем разбору требования, полученные на этапе определения требований, уточняя и структурируя их. Цель выполнения этого процесса состоит в том, чтобы добиться более точного понимания требований и получить простое в использовании описание требований, которое поможет нам представить структуру системы в целом, включая ее архитектуру.

Прежде чем мы объясним, что это значит, давайте вспомним некоторые результаты определения требований. Напомним, что правило номер один при определении требований – использовать язык клиента (см. раздел «Цели процесса определения требований» главы 6). Кроме того, как обсуждалось в главе 7, мы полагаем, что варианты использования создают хорошую основу для этого языка. Но даже если мы сумеем достичь договоренности с клиентом о том, что должна делать система, *вероятно, останутся нерешенные проблемы, касающиеся системных ресурсов*. Это – цена, которую мы платим за использование интуитивно понятного, но неопределенного языка клиента при определении требований. Чтобы пролить свет на то, какие же «нерешенные проблемы» могли остаться в системных требованиях, описанных в процессе определения требований, вернемся к вопросу эффективной передачи информации о функциях системы от заказчика:

1. *Варианты использования должны быть независимы друг от друга настолько, насколько это возможно.* Это необходимо для того, чтобы не запутаться в деталях, связанных с интерференцией, параллелизмом и конфликтами между вариантами использования, например при конкуренции за разделяемые системные ресурсы (см. подраздел «Вариант использования» главы 7). Так, варианты использования *Положить на счет* и *Снять со счета* обращаются к одному и тому же банковскому счету клиента. Также конфликт может произойти из-за объединения вариантов использования, которые в сумме приводят к нежелательному поведению, например, когда абонент телефонной сети использует вариант использования *Запросить Звонок-Будильник* вместе с вариантом использо-

вания *Переназначить Входящие Звонки*, заказывая в результате разбудить другого абонента. Проблема заключается в том, что интерференция, параллелизм и конфликты между вариантами использования в процессе определения требований неразрешимы.

2. *Варианты использования должны быть описаны на языке клиента.* Это делается прежде всего для сохранения в описаниях варианта использования естественного языка и осторожности в использовании более формальной нотации, такой как диаграммы состояний, деятельности и взаимодействия (см. подраздел «Формализация описания варианта использования» главы 7). Но, используя только естественный язык, мы теряем выразительную мощность. Множество деталей, которые мы могли бы точно описать посредством более формальных систем, в ходе определения требований остаются неопределенными или описанными поверхностью.
3. *Каждый вариант использования должен быть структурирован так, чтобы давать полное и интуитивно понятное описание функциональных возможностей.* Это достигается структурированием вариантов использования (а значит, и требований) так, чтобы они интуитивно отражали «реальные» варианты использования, осуществляемые системой. Нельзя структурировать их на мелкие, абстрактные, непонятные варианты использования, например, сокращая избыточность. Мы должны, насколько это возможно, достичь компромисса между понятностью и пригодностью к работе для описаний вариантов использования (см. подраздел «Поиск других отношений между вариантами использования» главы 7). Вопросы, требующие такой избыточности описания требований, в ходе определения требований не решаются.

Что касается этих нерешенных задач, основная цель анализа — решить их путем более глубокого изучения требований. Главное отличие, по сравнению с определением требований, состоит в том, что при описании результатов может использоваться язык разработчиков.

Как следствие, при анализе мы можем больше уделить внимания внутренней организации системы и вследствие этого решать задачи, связанные с интерференцией вариантов использования и т. п. (см. пункт 1). Кроме того, при анализе мы можем использовать более формальный язык, что позволит нам точнее описать детали, связанные с системными требованиями (см. пункт 2 чуть ранее). Мы будем называть этот процесс *уточнением требований*.

Заметим также, что при анализе мы можем структурировать требования так, чтобы облегчить их понимание, подготовку, изменение, многократное использование и вообще работу с ними (см. пункт 3). Предлагаемая структура (основанная на анализе классов и пакетов) ортогональна структуре, построенной в соответствии с требованиями (и основанной на вариантах использования). Однако эти различные структуры могут легко отображаться друг в друга, так что мы можем проектировать различные описания — на различном уровне детализации — для соответствующих требований и легко сохранять совместимость описаний. Фактически, это отображение определено между вариантами использования в модели варианта использования и реализациями вариантов использования в аналитической модели. Подробнее мы обсудим этот вопрос чуть позже (табл. 8.1).

Таблица 8.1. Краткое сравнение модели вариантов использования и аналитической модели

Модель вариантов использования	Аналитическая модель
Использует язык заказчика	Использует язык разработчиков
Внешний вид системы	Внутренний вид системы
Структурирована по вариантам использования, описывает структуру внешнего вида	Структурирована по стереотипным классам и пакетам, описывает структуру внутреннего вида
Используется в первую очередь как соглашение между заказчиком и разработчиками о том, что должна делать система, а чего не должна	Используется в первую очередь разработчиками для того, чтобы понять, как система должна быть оформлена, то есть спроектирована и разработана
Может содержать избыточность, несовместимые требования и т. п.	Не должна содержать избыточности, несовместимых требований и т. п.
Определяет функциональность системы, в том числе зависящую от архитектуры	Описывает, как функциональность реализуется в системе, включая функциональность, зависящую от архитектуры; дает наметки для проектирования
Определяет варианты использования, которые затем будут анализироваться в аналитической модели	Определяет реализации вариантов использования, каждая из которых представляет собой анализ варианта использования из модели вариантов использования

Ну и наконец, структурирование требований, осуществляемое в ходе анализа, также является существенным вкладом в формирование системы (включая ее архитектуру), потому что мы хотим сделать систему работоспособной в целом, а не только описать требования к ней.



Рис. 8.1. Сотрудники и артефакты, вовлеченные в анализ

В этой главе мы представим детальное объяснение того, что мы понимаем под анализом, а также под уточнением и структурированием требований. Мы начнем с краткого «позиционирования» анализа в последовательности рабочих процессов (раздел «Кратко об анализе»), затем опишем роль анализа в ходе различных фаз жизненного цикла программного обеспечения (раздел «Роль анализа в жизненном цикле программы»). После этого мы рассмотрим артефакты (раздел «Арте-

факты») и сотрудников (раздел «Сотрудники»), вовлеченных в процесс анализа (рис. 8.1). В заключение мы опишем рабочий процесс анализа в целом (раздел «Рабочий процесс»).

Кратко об анализе

Язык, которым мы пользуемся в анализе, основан на концептуальной модели объекта, называемой *аналитической моделью*. Аналитическая модель помогает нам уточнить требования по пунктам, упомянутым ранее (предыдущий раздел) и дает советы по внутренней организации системы, в том числе и по совместному использованию внутренних ресурсов. Фактически, внутренние ресурсы могут быть представлены как объекты аналитической модели, как банковский счет клиента, к которому обращаются варианты использования *Положить на счет* и *Снять со счета*. Кроме того, аналитическая модель предоставляет в наше распоряжение большую выразительную мощность и формализм таких средств, как, например, диаграммы взаимодействия, используемые для описания динамики системы.

Аналитическая модель также помогает нам структурировать требования, как рассматривалось в предыдущем разделе, и обеспечивает структуру, которая ориентирована на удобство использования, в частности — простоту отработки изменений, внесенных в требования, и возможности многократного использования. Позже в этой главе мы обсудим принципы построения аналитической модели, гибко отрабатывающей такие изменения и содержащей многократно используемые элементы. Эта структура полезна не только для работы с требованиями. Она также используется в качестве исходных данных для проектирования и реализации системы (как описано в главах 9 и 10). Мы создаем структуру для формирования системы и принятия решений по ее проектированию и реализации. В соответствии с этим аналитическая модель может рассматриваться как первый шаг к проектной модели, хотя она является моделью и сама по себе. Сохраняя структуру аналитической модели при проектировании, мы получим систему, которая, в целом, должна также быть удобна в работе. Она будет гибко отрабатывать вносимые в требования изменения и содержать элементы, которые можно будет многократно использовать при разработке похожих систем.

Однако важно отметить, что аналитическая модель создает абстракции и не лучшим образом решает некоторые проблемы и поддерживает некоторые требования. Мы полагаем, что лучше переложить их на процессы проектирования и реализации (см. приложение B, а также подраздел «Почему анализ — это не проектирование и не реализация»). В результате структура, создаваемая аналитической моделью, не может остаться неизменной. Она должна дополнительно обсуждаться и приводиться к удобному виду в ходе проектирования и реализации; мы поговорим об этом в главах 9 и 10. Причина, по которой эта «неизменная структура» не может быть использована на практике, состоит в том, что при проектировании мы должны определиться с платформой реализации: языком программирования, операционными системами, покупными модулями, системой версий и т. д. Лучшая по соотношению «цена-эффективность» архитектура может быть получена путем изменения структуры аналитической модели при переходе к проектной модели и формированию системы.

Почему анализ — это не проектирование и не реализация

Вы можете спросить, почему мы не анализируем требования одновременно с проектированием и реализацией системы. Наш ответ — потому, что *и проектирование и реализация намного больше по объему, чем анализ (уточнение и структурирование требований), так что разделить их просто необходимо*. При проектировании мы должны сформировать систему, найти ее форму, в том числе и архитектуру; форму которая существует поверх всех требований к системе, форму, которая включает в себя компоненты кода, скомпилированные и объединенные в исполняемые версии системы; и главное, форму, которой мы сможем пользоваться длительное время, — форму, которая может устоять под давлением времени, изменений и эволюции; форму, обладающую целостностью.

При проектировании мы, таким образом, должны принимать решения, учитывая то, как система должна выполнять, например, требования производительности и рассредоточенной работы, отвечая на вопросы типа «как оптимизировать эту процедуру, чтобы она выполнялась максимум за 5 миллисекунд?» и «как загрузить этот код на узел сети, чтобы при этом не перегрузить сеть?». Имеется еще множество других проблем такого рода, которые нужно решить в ходе проектирования: как эффективно эксплуатировать такие компоненты сторонних производителей, как системы управления базами данных и брокеров объектных запросов, как интегрировать их в архитектуру системы, как наилучшим образом использовать язык программирования и т. д. Мы не будем давать здесь полного списка всех дополнительных вопросов, возникающих в ходе проектирования и реализации, но вернемся к ним в главах 9 и 10. Мы надеемся, что объяснили нашу позицию, которая состоит в том, что *проектирование и реализация намного объемнее, чем просто анализ требований, их уточнение и структурирование. Проектирование и реализация — это фактически и есть весь процесс формирования системы в том виде, как она будет существовать, выполняя все эти требования, включая нефункциональные*. Чтобы дать некоторое понятие о том, как абстракции модели анализа соотносятся с богатством подробностей проектной модели, скажем лишь, что соотношение числа элементов этих моделей 1:5 — вполне обыкновенное дело.

Понимая все это, мы также считаем, что перед началом проектирования и реализации нужно иметь точное и детальное понимание требований на таком уровне, который клиента (в большинстве случаев) не волнует. Кроме того, очень полезно иметь структуру требований, которая может быть использована как исходные данные для формирования системы. Все это достигается посредством анализа.

Выполняя анализ, мы добиваемся разделения задач, которые готовим и упрощаем для последующей деятельности по проектированию и реализации. Мы разграничиваем проблемы, которые должны быть решены, и решения, которые должны быть приняты для того, чтобы справиться с ними. Кроме того, разделяя эти задачи, разработчики оказываются в состоянии «окинуть взглядом гору» в самом начале работы по созданию программы и вследствие этого не впасть в паралич при попытке решить слишком много проблем сразу — включая проблемы, которые, вероятно, не удастся решить вообще, потому что исходные требования были не определены или неправильно поняты.

Цели анализа: краткий обзор

Анализ требований в форме модели анализа важен по нескольким причинам, объясненным ранее.

- Модель анализа дает более точную спецификацию требований, чем та, что была получена нами в результате определения требований, включая модель вариантов использования.
- Модель анализа описывается с использованием языка разработчиков и вследствие этого позволяет вводить больше формализма и может использоваться для анализа внутренних механизмов системы.
- Модель анализа структурирует требования так, что это облегчает их понимание, подготовку, внесение изменений и вообще работу с ними.
- Модель анализа может рассматриваться как первый шаг к модели проектирования (хотя это отдельная модель), а значит, в качестве важных исходных данных для формирования системы в ходе проектирования и реализации. Это важно, поскольку удобной в работе должна быть вся система, а не только описание требований.

Конкретные примеры случаев, в которых следует использовать анализ

В дополнение к тому, что мы сказали ранее, мы дадим более конкретные пояснения, когда применять анализ и как использовать его результаты (то есть аналитическую модель).

- Выполняя анализ отдельной фазой, вместо того, чтобы выполнить его в ходе проектирования и реализации, мы можем недорого проанализировать большую часть системы. Мы можем затем использовать результат анализа для планирования последующих работ по проектированию и реализации; или в виде нескольких последовательных этапов, на каждом из которых проектируется и реализуется небольшая часть системы, или как несколько параллельных этапов, возможно, разрабатываемых и выполняемых группами разработчиков, разнесенных географически. Выделение и планирование этих этапов без результатов анализа может быть сложнее.
- Анализ обеспечивает получение краткого обзора системы, который может быть трудно получить, исследуя результаты проектирования или реализации, так как на этих этапах вводится слишком большая детализация (напомним отношение 1:5, обсуждавшееся в подразделе «Почему анализ — это не проектирование и не реализация»). Такой краткий обзор оказывается очень ценным для новых сотрудников, включившихся в работу над системой или разработчиков, обслуживающих систему в целом. Так, например, одна организация разработала большую систему (с тысячами сервисных подсистем), используя принципы, подобные описанным в главах 3 и 4. После этого мы создали аналитическую модель, чтобы добиться лучшего понимания уже разработанной системы. Исполнительный директор суммировал опыт заказчика так: «Спасибо за аналитическую модель, теперь мы в состоянии обучить системных архитекторов за два года

вместо пяти». Для меньшей системы период обучения измеряется в месяцах, а не в годах, но соотношение величин будет тем же самым.

- Некоторые части системы имеют альтернативные схемы и/или реализации. Например, системы, от работоспособности которых зависит жизнь людей, такие как системы управления самолетом или железной дорогой, могут состоять из нескольких различных программ, которые одновременно производят одни и те же действия, и важные маневры могут происходить только в том случае, если эти вычисления дают одинаковые результаты. Другой пример — когда клиент хочет иметь двух или больше продавцов или субподрядчиков, чтобы обеспечить продажи программ в разных местах; или хочет иметь предложения, основанные на одной и той же спецификации от двух конкурирующих фирм по разработке программного обеспечения. Вообще говоря, это случаи, когда часть системы реализуется несколько раз, при помощи различных технологий — языков программирования или компонентов, работающих на различных платформах. Модель анализа может предложить концептуальные, точные и унифицированные методы просмотра этих альтернативных реализаций. В этом случае аналитическая модель, очевидно, должна сохраняться все время жизни системы.
- Система построена с использованием сложной унаследованной системы. Эта унаследованная система или ее часть может быть затем повторно перепроектирована в понятиях аналитической модели так, чтобы разработчики могли понять унаследованную систему без необходимости копаться в деталях ее проектирования и реализации и сформировать новую систему, используя унаследованную как конструктивный блок многократного использования. Полная переработка проекта и новая реализация такой унаследованной системы может быть очень сложна, дорога и не слишком полезна — особенно если унаследованная система не должна быть изменена и была выполнена по устаревшим технологиям.

Роль анализа в жизненном цикле программы

Анализ — это главная стадия в течение начальных итераций фазы проектирования (рис. 8.2). Это способствует созданию надежной и устойчивой архитектуры и облегчает глубокое понимание требований. Позже, в конце проектирования и в построении, когда архитектура устойчива и требования поняты, центр внимания перейдет на процессы проектирования и реализации.

В каждом проекте цели анализа должны быть так или иначе достигнуты. Но конкретный способ представления и использования анализа может быть различным для разных проектов. Мы видим три основных варианта.

1. Проект использует модель анализа (как будет подробно рассмотрено позже в этой главе), чтобы описать результаты анализа, и сохраняет совместимость этой модели в течение всего жизненного цикла программного обеспечения. Этот процесс может быть, например, сделан непрерывным — на каждой итерации проекта мы будем добиваться каких-либо преимуществ, указанных в предыдущем подразделе.

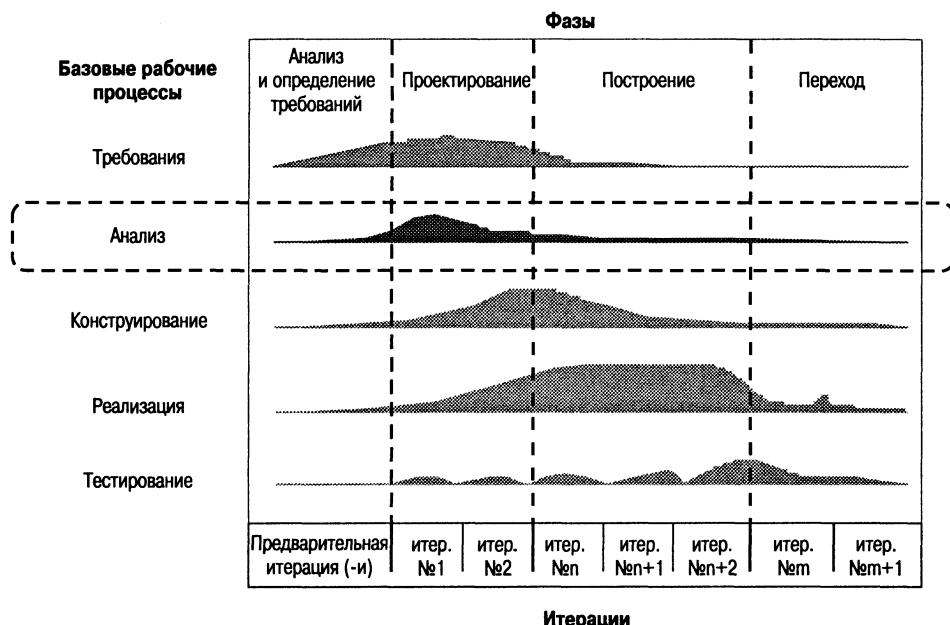


Рис. 8.2. Центр внимания при анализе

2. Проект использует модель анализа, чтобы описать результаты анализа, но рассматривает эту модель как промежуточное средство — возможно, с центром в фазе проектирования. Позже, в течение фазы построения, когда проектирование и реализация станут важнее, модель анализа больше сохраняться не будет. Вместо этого любой оставшийся «вопрос анализа» будет решаться в виде интегрированной части проектных работ в результирующей проектной модели (которую мы рассмотрим в главе 9).
3. Проект вообще не использует аналитическую модель для описания результатов анализа. Вместо этого анализ требований проходит в проекте как интегрированная часть определения требований или проектирования. Первый случай потребовал бы большей формальности в модели вариантов использования. Это может быть оправдано, если клиент способен понять результаты, хотя мы полагаем, что это имеет место нечасто. Второй случай усложнил бы проектирование, что мы объясняли в подразделе «Почему анализ — это не проектирование и не реализация». Однако это может быть оправдано, если, например, требования очень просты и/или известны, если форма системы (включая архитектуру) проста для нахождения или если разработчики имеют интуитивное, но правильное понимание требований и сразу способны к построению системы, которая выполняет эти требования. Мы полагаем, что это также редко случается.

При выборе между первыми двумя вариантами должны быть взвешены как преимущества поддержания аналитической модели, так и стоимость этого поддержания в течение нескольких итераций и версий. Мы, следовательно, должны сделать правильную оценку соотношения цена/прибыль и решить, прекратим мы хра-

нить модель анализа так быстро, как сможем, — вероятно, по окончании фазы проектирования, — или будем хранить и поддерживать ее до окончания срока жизни системы.

Принимая третий вариант, мы подтверждаем, что проект может обойтись не только без затрат на сохранение аналитической модели, но и без затрат на создание аналитической модели (как-то: затрат времени и ресурсов на обучение разработчиков и получение ими соответствующего опыта по использованию этой модели). Однако, как уже подчеркивалось, поскольку мы полагаем, что обычно выгоды от работы с аналитической моделью, по крайней мере на первых порах, превосходят затраты на ее создание, этот вариант должен быть использован только в редких случаях разработки необычайно простых систем.

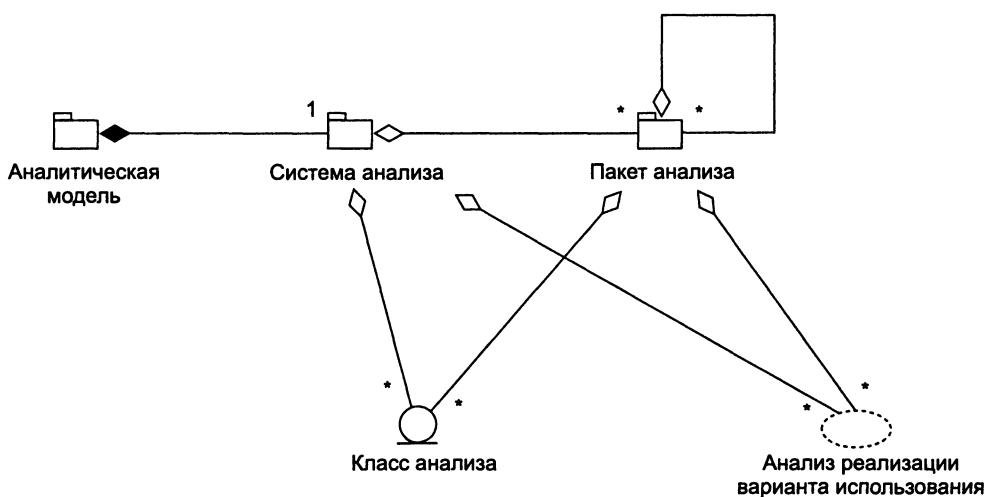


Рис. 8.3. Модель анализа — это иерархия пакетов анализа, содержащих классы анализа и анализы реализаций вариантов использования

Артефакты

Артефакт: Модель анализа

Мы ввели понятие модели анализа в начале раздела «Кратко об анализе». Структура, заложенная в аналитической модели, определяется иерархией, как показано на рис. 8.3.

Модель анализа представлена системой анализа, лежащей на верхнем уровне пакетов модели. Для организации модели анализа в виде более управляемых частей используются пакеты анализа, которые представляют собой абстракции подсистем и, возможно, уровни проекта системы. Классы анализа представляют собой абстракции классов и, возможно, подсистемы системы проектирования. Внутри модели анализа варианты использования реализованы с помощью классов ана-

лиза и их объектов. Это реализовано при помощи кооперации внутри аналитической модели и обозначается при помощи анализов реализаций вариантов использования. Артефакты аналитической модели подробно рассматриваются в подразделах «Артефакт: Класс анализа», «Артефакт: Анализ реализации варианта использования», «Артефакт: Пакет анализа» и «Артефакт: Описание архитектуры (представление модели анализа)».

Артефакт: Класс анализа

Класс анализа представляет собой абстракцию одного или более классов и/или подсистем в проекте системы. Эта абстракция имеет следующие характеристики.

- Класс анализа сосредоточен на представлении функциональных требований и откладывает нефункциональные требования на последующие стадии – проектирование и реализацию, обозначая их как специальные требования класса. Это делает класс анализа более очевидным в контексте проблемной области, более «концептуальным» и часто более детализированным, чем соответствующие проект и реализация.
- Класс анализа редко определяет или поддерживает какие-либо интерфейсы в понятиях операций и их сигнатур. Вместо этого его поведение определено в соответствии с ответственостями верхнего, менее формального уровня. Ответственность – это текстовое описание связанного поднабора поведения, определенного в классе.
- Класс анализа определяет атрибуты, хотя эти атрибуты – также довольно высокого уровня. Типы этих атрибутов часто концептуальны и берутся из прикладной области, в то время как типы атрибутов классов проектирования и реализации – это обычно типы языка программирования. Кроме того, атрибуты, обнаруженные в ходе анализа, обычно становятся классами проекта и реализации.
- Класс анализа вовлечен в отношения, хотя эти отношения более концептуальны, чем их аналоги в проектировании и реализации. Например, направленность ассоциаций не слишком важна в анализе, но существенна для проектирования. Также в анализе могут использоваться обобщения, но их невозможно использовать в проектировании, если они не поддерживаются языком программирования.
- Классы анализа всегда можно отнести к одному из типов: граничный, управляющий или сущности (рис. 8.4). Каждый стереотип подразумевает определенную семантику (мы ее вскоре опишем), которая приводит к мощному и логичному методу обнаружения, и описание классов анализа и способствует созданию надежной модели объекта и архитектуры. Однако воспользоваться этим ясным и интуитивным методом для стереотипизации классов проектирования и реализации гораздо сложнее. Поскольку они охватывают и нефункциональные требования, они «живут в контексте решений предметной области» и часто описываются с использованием синтаксиса языка программирования и других технологий низкого уровня.

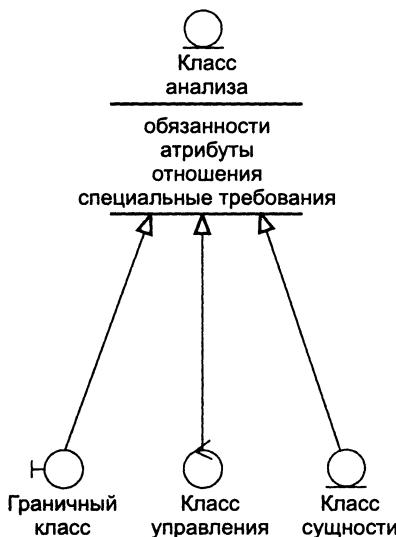


Рис. 8.4. Ключевые атрибуты и подтипы (то есть стереотипы) класса анализа

Вариант 1



Счет



Интерфейс
кассира



Снятие
денег

Вариант 2

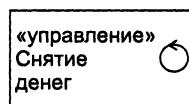
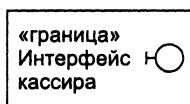
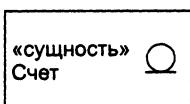


Рис. 8.5. В UML имеются три стандартных стереотипа классов, используемых при анализе

Эти три стереотипа стандартизированы в UML и используются, чтобы помочь разработчикам различать действия различных классов [57]. Каждый стереотип имеет свой собственный символ, что проиллюстрировано на рис. 8.5.

Границные классы

Границные классы используются для моделирования взаимодействия между системой и ее актантами (пользователями и внешними системами). Взаимодействие часто включает в себя получение (и передачу) информации и запросы пользователей и внешних систем к ним.

Границные классы моделируют части системы, которые зависят от ее актантов в том смысле, что они поясняют и собирают требования к границам системы. Таким образом, изменения в интерфейсе пользователя или коммуникационном интерфейсе обычно локализовано в одном или нескольких границных классах.

Границные классы часто представляют собой абстракции окон, форм, панелей, коммуникационных интерфейсов, интерфейсов принтера, датчиков, терминалов и (возможно, не объектно-ориентированных) API. Однако граничные классы находятся на довольно высоком концептуальном уровне и не должны описывать каждую мелочь интерфейса пользователя. Вполне достаточно, если граничные классы опишут только то, что происходит при взаимодействии с актантом (то есть запросы и информацию, передаваемую вперед-назад между системой и ее актантами). Граничные классы не должны описывать физическую реализацию, поскольку она рассматривается на последующих стадиях проектирования и реализации.

Каждый граничный класс должен быть связан как минимум с одним актантом, и наоборот.

Пример. Границный класс *Интерфейс запроса на оплату*. Данный граничный класс под названием *Интерфейс запроса на оплату* используется для поддержки взаимодействия между *Покупателем* и вариантом использования *Оплатить счет* (рис. 8.6).



Рис. 8.6. Граничный класс Интерфейс запроса на оплату

Интерфейс запроса на оплату позволяет пользователю просматривать счета, подлежащие оплате, и указывать системе оплатить счет (помечая его). *Интерфейс запроса на оплату* также позволяет пользователю отклонить счет, который покупатель не желает оплачивать.

Далее мы приведем примеры того, как этот граничный класс соотносится с «внутренними» классами — сущностей и управления.

Классы сущностей

Класс сущности используется для моделирования долгоживущей, нередко сохраняемой информации. Классы сущности моделируют данные об информационном наполнении и связях некоторых явлений или концепций — человека, объекта или события из реального мира.

В большинстве случаев классы сущностей получаются непосредственно из соответствующего бизнес-класса сущности (или доменного класса) и бизнес-модели объекта (или доменной модели). Однако главная разница между классами сущностей и бизнес-классами сущностей состоит в том, что первый описывает объекты, обрабатываемые рассматриваемой системой, в то время как второй — бизнес-объекты (или объекты прикладной области) вообще. В результате классы сущностей отражают информацию так, чтобы она была полезна разработчикам при проектировании и выполнении системы, включая поддержку персистентности. Для бизнес-классов сущностей (или классов сущностей предметной области) это не так. Они описывают контекст системы и вследствие этого могут содержать информацию, которая вообще не обрабатывается внутри системы.

Объект сущности не обязан быть пассивным и может иногда иметь сложное поведение, зависящее от содержащейся в нем информации. Объекты сущностей ограничивают изменения информацией, которую они представляют.

Классы сущностей часто показывают логическую структуру данных и способствуют пониманию того, как система от нее зависит.

Пример. Класс сущности *Счет*. Класс сущности под названием *Счет* используется для представления счетов. Класс сущности связан с граничным классом *Интерфейс запроса на оплату*, через который пользователь просматривает и обрабатывает счета (рис. 8.7).

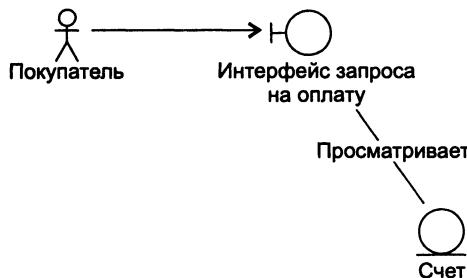


Рис. 8.7. Класс сущности Счет и его взаимосвязи с граничным классом Интерфейс запроса на оплату

Управляющие классы

Управляющие классы отвечают за координацию, порядок последовательности, взаимодействия и управление другими объектами и часто используются для хранения управления, относящегося к некоторому варианту использования. Классы управления также используются для представления сложных переходов и вычислений, например бизнес-логики, которая не может быть связана ни с какой определенной долгосрочной информацией, хранящейся в системе (то есть с определенным классом сущности).

Динамика системы моделируется именно управляющими классами, так как они обрабатывают и координируют основные действия и потоки управления и делегируют работу другим объектам (граничным и объектам сущностей).

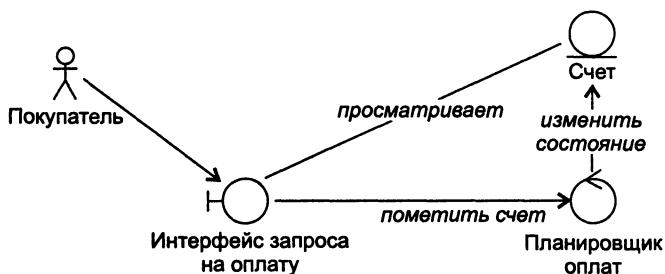


Рис. 8.8. Введение управляющего класса Планировщик оплат и его взаимосвязи с граничным классом и классом сущности

Заметьте, что управляющие классы не хранят ни детали взаимодействия с актантами, ни вопросы работы с информацией длительного хранения, существующей в системе; этим занимаются граничные классы и классы сущностей, соответственно. Но зато классы управления хранят и ограничивают доступ к управлению, координации, порядку последовательности, взаимодействиям, а иногда — и к сложной бизнес-логике, которых требуют другие объекты.

Пример. Управляющий класс *Планировщик оплат*. Дополняя предшествующий пример, мы вводим управляющий класс под названием *Планировщик Оплат*, который является ответственным за координацию между *Интерфейсом запроса на оплату* и *Счетом* (рис. 8.8).

Планировщик оплат принимает запрос на платеж, например, запрос на оплату счета и дату оплаты счета. Позднее, в день платежа *Планировщик оплат* осуществляет платеж, проводя перевод денег между соответствующими счетами.

Артефакт: Анализ реализации варианта использования

Анализ реализации варианта использования — это кооперация внутри аналитической модели, которая описывает, как реализован определенный вариант использования и как он выполняется в понятиях классов анализа и взаимодействующих объектов анализа. Реализация варианта использования, таким образом, представляет собой непосредственную проекцию соответствующего варианта использования из модели вариантов использования (рис. 8.9).

Реализация варианта использования содержит текстовое описание потока событий, диаграммы классов, описывающие участвующие в нем классы анализа, и диаграммы взаимодействия, которые описывают реализацию конкретного потока или сценария вариантов использования в понятиях взаимодействующих объектов анализа (рис. 8.10). Кроме того, поскольку реализация вариантов использования описывается в терминах классов анализа и их объектов, она, естественно, сосредоточивается на функциональных требованиях. Таким образом, и здесь, так же как в классах анализа, можно отложить рассмотрение нефункциональных требований до последующих действий — проектирования и реализации, обозначив их пока как специальные требования к реализации.

Диаграммы классов

Класс анализа и его объекты часто участвуют в нескольких вариантах использования, при том, что некоторые обязанности, атрибуты и ассоциации класса связаны лишь с одним вариантом использования. Поэтому так важно в ходе анализа скординировать все требования к классу и его объектам, которые имеют различные реализации вариантов использования. Чтобы сделать это, мы добавляем диаграммы классов к реализации варианта использования, дабы увидеть участвующие в ней классы и их связи (рис. 8.11).

Диаграммы взаимодействий

Последовательность действий варианта использования начинает выполняться, когда актант инициирует вариант использования, посыпая системе некое сообщение. Если теперь мы заглянем «внутрь» системы, то увидим, что это сообщение от

актанта получит граничный объект. Затем граничный объект перешлет это сообщение какому-то другому объекту, так что для того, чтобы реализовать этот вариант использования, вовлеченные в него объекты взаимодействуют между собой. В фазе анализа мы предпочтаем описывать этот процесс при помощи диаграмм кооперации, так как наша основная задача — поиск требований и обязательств объектов, а не поиск детализированных хронологических последовательностей взаимодействий (тогда мы бы мы использовали диаграммы последовательностей).



Рис. 8.9. Создание трассировки между анализом реализации варианта использования в аналитической модели и вариантом использования в модели вариантов использования



Рис. 8.10. Ключевые атрибуты и ассоциации анализа реализации варианта использования

Используя диаграммы кооперации, мы поясняем взаимодействие объекта, создавая связи между объектами и присоединяя к этим связям сообщения. Название сообщения должно обозначить намерение вызывающего объекта при взаимодействии с вызываемым.

Пример. Диаграмма сотрудничества, описывающая реализацию варианта использования. Рисунок 8.12 — это диаграмма кооперации, реализующая первую часть варианта использования *Оплатить счет*.

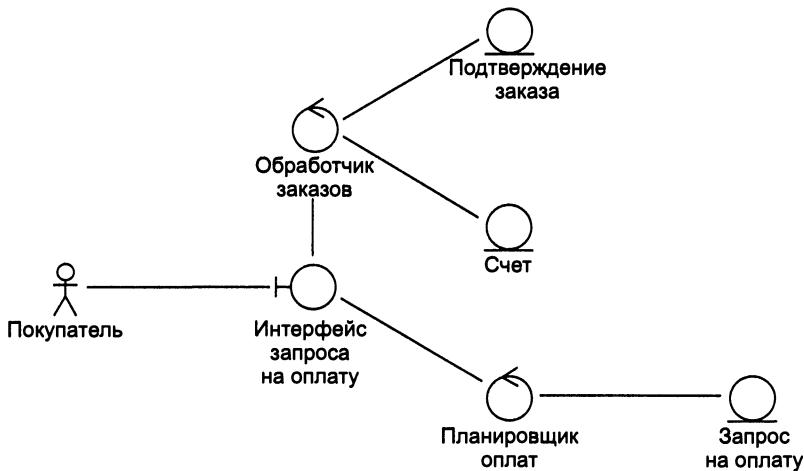


Рис. 8.11. Диаграмма классов реализации варианта использования Оплатить счет

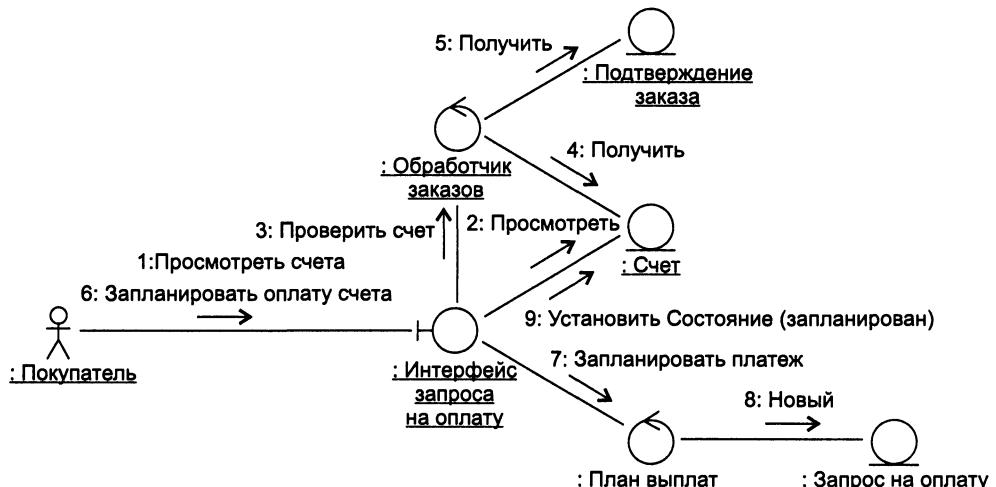


Рис. 8.12. Диаграмма кооперации реализации варианта использования Оплатить счет

Относительно создания и уничтожения объектов анализа внутри реализации варианта использования отметим, что различные объекты имеют различные жизненные циклы (приложение В).

- Границный объект не обязан быть привязанным к одной реализации варианта использования, например он может представлять собой окно и совместно использоваться двумя или более экземплярами варианта использования. Однако граничные объекты часто создаются, и уничтожаются внутри единственной реализации варианта использования.

- Объект сущности часто не привязан к одной реализации варианта использования. Напротив, объект сущности обычно является долгоживущим и до своего уничтожения успевает поработать в нескольких реализациях.
- Управляющие классы часто инкапсулируют управление определенного варианта использования, что подразумевает создание управляющего объекта в начале варианта использования и уничтожение управляющего объекта при его окончании. Однако существуют и исключения, когда управляющий объект существует более чем в одной реализации варианта использования, когда несколько управляющих объектов (различных управляющих классов) совместно работают в одной реализации варианта использования или когда реализация варианта использования не требует вообще никаких управляющих объектов.

Поток событий — Анализ

Диаграммы, особенно диаграммы кооперации реализации варианта использования, может быть нелегко прочесть, и приходится использовать дополнительный текст, чтобы от диаграмм была какая-то польза. Текст должен быть написан в терминах объектов, в особенности управляющих объектов, которые взаимодействуют между собой, чтобы выполнить вариант использования. Текст не должен, однако, содержать упоминаний об атрибутах объектов, обязанностях и ассоциациях, которые трудно поддерживать в актуальном состоянии, потому что они очень часто изменяются.

Пример. Поток событий — анализ, поясняющий диаграмму кооперации. Диаграмме кооперации, приведенной в предыдущем примере (см. рис. 8.12), соответствует помещенное ниже текстовое описание.

Покупатель просматривает счета, управляемые системой, чтобы найти свежеполученные (1, 2), через *Интерфейс запроса на оплату*. *Интерфейс запроса на оплату* использует *Обработчик Заказов*, чтобы проверить счета по соответствующим подтверждениям заказов (3, 4, 5) перед тем, как показать их покупателю. Как будет проводиться эта проверка, зависит от бизнес-правил, установленных фирмой покупателя; они могут включать в себя сравнение цен, даты поставки и комплекта поставки по счету и подтверждению заказа. Объект *Обработчика Заказа* использует эти бизнес-правила, чтобы решить, какие вопросы (на рисунке соответствуют сообщениям *Взять* 4, 5) следует задать, чтобы запросить *Подтверждение Заказа* или *Отклонить Счет*, и как анализировать ответы. Предполагается, что в результате счет будет как-то помечен *Интерфейсом запроса на оплату*, возможно, с использованием разноцветной подсветки.

Покупатель выбирает счет через *Интерфейс запроса на оплату* и помечает счет к оплате (6). *Интерфейс запроса на оплату* запрашивает *Планировщик Оплаты* наиметить оплату счета (7). *Планировщик Оплаты* создает требование на оплату (8). *Интерфейс запроса на оплату* затем изменяет состояние счета на «намечен к оплате» (9).

Планировщик Оплаты приступает к оплате в день оплаты (на диаграмме не показано).

Интересно сравнить это описание с потоком событий варианта использования, описанным в главе 7, подразделе «Поток событий». Описание из главы 7 — это

внешний вид наблюдаемого поведения варианта использования, в то время как описание, данное здесь, фокусируется на том, как вариант использования реализуется системой в понятиях совместно работающих (логических) объектов.

Специальные требования

Специальные требования — это текстовые описания, которые охватывают все нефункциональные требования реализаций вариантов использования. Некоторые из этих требований уже определялись в каком-то виде в течение рабочего процесса определения требований (как описано в главах 6 и 7), и лишь немного изменились в ходе анализа реализаций вариантов использования. Кроме того, среди них могут быть и новые или производные от старых требования, найденные в процессе анализа.

Пример. *Специальные требования к Реализации вариантов использования.* Примером специальных требований к реализации варианта использования *Оплатить счет* может служить следующий:

Если покупатель хочет просмотреть полученные счета, они должны появиться на экране не более чем через 0,5 секунд.

Счета должны оплачиваться с использованием стандарта SET.

Артефакт: Пакет анализа

Пакеты анализа обеспечивают средства для организации артефактов модели анализа в обозримые блоки. Пакет анализа может состоять из классов анализа, реализаций вариантов использования и других пакетов анализа (рекурсивно), рис. 8.13.

Пакеты анализа должны быть сильно внутренне связаны (то есть их содержимое должно быть жестко связано) и слабо связаны внешне (то есть их зависимость друг от друга должны быть минимальной).

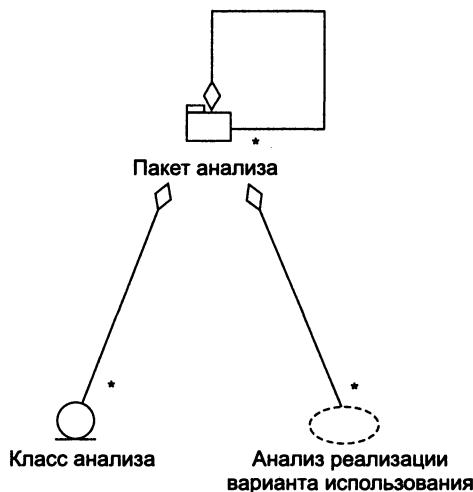


Рис. 8.13. Содержимое пакета анализа

Кроме того, пакеты анализа должны обладать следующими характеристиками.

- Пакеты анализа могут позволять проводить раздельный анализ. Например, в большой системе некоторые пакеты анализов могут быть проанализированы независимо, возможно, одновременно различными разработчиками с различным знанием предметной области.
- Пакеты анализа должны быть созданы на основе функциональных требований и области применения (то есть приложении или бизнесе) и должны распознаваться людьми со знанием предметной области. Пакеты анализов не должны быть основаны на нефункциональных требованиях и области решения.
- Пакеты анализа удобны для создания или распределения среди подсистем двух высших прикладных уровней модели проектирования. В некоторых случаях пакет анализа мог бы даже отражать целиком верхний уровень модели проектирования.

Сервисные пакеты

Кроме обеспечения актантов нужными им вариантами использования, каждая система также обеспечивает набор услуг своим клиентам. Клиент приобретает соответствующую смесь услуг, чтобы дать пользователям необходимые в их бизнесе варианты использования.

- Вариант использования определяет последовательность действий: задача запускается актантом, сопровождается взаимодействиями между актантом и системой и заканчивает свою работу, возвращая значение актанту. Обычно варианты использования существуют не в изоляции. Например, вариант использования *Послать Счет Покупателю* предполагает, что другой вариант использования создал счет и что известны адрес и иные данные покупателя.
- Сервис представляет собой когерентный набор функционально связанных действий — функциональный модуль — участвующий в нескольких вариантах использования. Клиент системы обычно покупает набор сервисов, чтобы предоставить своим пользователям все необходимые варианты использования. Сервис неделим в том смысле, что система предоставляет его полностью или никак.
- Варианты использования — для пользователей, а сервисы — для клиентов. Сервисы пересекаются с вариантами использования, то есть вариант использования требует действий нескольких сервисов.

В Унифицированном процессе понятие сервиса поддерживается сервисными пакетами. Сервисные пакеты используются на нижнем уровне иерархии пакетов анализа (содержимое), чтобы структурировать систему по услугам, которые она обеспечивает. О сервисных пакетах можно сказать следующее.

- Сервисные пакеты содержат набор функционально связанных классов.
- Сервисные пакеты неделимы. Каждый заказчик получает все классы пакета или ни одного.
- При реализации варианта использования в реализации могут принимать участие один или несколько сервисных пакетов. И наоборот, для некоторого сервисного пакета характерно участие в нескольких различных реализациях вариантов использования.

- Сервисные пакеты обычно очень слабо зависят от других сервисных пакетов.
- Сервисные пакеты обычно работают с одним актантом или немногочисленными актантами.
- Функциональность, определяемая сервисным пакетом, может, при соответствующей разработке и проектировании, управляться как отдельная программная утилита. Сервисные пакеты могут, таким образом, рассматриваться как некая «подключаемая» функциональность системы. При исключении сервисного пакета то же самое происходит со всеми реализациями, требующими этого сервисного пакета.
- Сервисные пакеты могут быть взаимно исключающими либо представлять различные аспекты или варианты одного и того же сервиса. Например, «Проверка орфографии Британского Английского» и «Проверка орфографии Американского Английского» могут быть двумя разными сервисными пакетами, предоставляемыми системой.
- Сервисные пакеты поставляют важные данные для последующих действий по проектированию и реализации, в которых они помогают структурировать модели проектирования и реализации в терминах сервисных подсистем. Сервисные же подсистемы дают главный эффект при разложении системы на двоичные и выполняемые компоненты.

Структурируя систему согласно сервисам, которые она предоставляет, мы заранее готовимся к внесению изменений в отдельные сервисы, так как такие изменения обычно ограничиваются соответствующим сервисным пакетом. Это позволяет нам создать устойчивую систему, не боящуюся изменений.

Метод выделения сервисных пакетов вместе с примерами сервисных пакетов описывается в подразделе «Идентификация сервисных пакетов» данной главы.

Заметим, что стандартный способ организации артефактов аналитической модели — использование обычных пакетов анализа, обсуждавшихся в предыдущем пункте. Однако здесь мы вводим стереотип «сервисного пакета», чтобы быть в состоянии явно регистрировать пакеты, предоставляющие сервисы. Это особенно важно в больших системах (содержащих много пакетов), чтобы иметь возможность просто распознавать различные типы пакетов. Этот стереотип также обрисовывает разработку и использование стереотипов по правилам UML.

Сервисные пакеты можно использовать многократно

Как утверждалось в предыдущем подразделе, сервисные пакеты имеют много полезных характеристик, будучи связанными (приложение В), неделимыми, слабо связанными, независимо устанавливаемыми и т. д. Это делает большинство сервисных пакетов основными кандидатами на многократное использование как внутри системы, так и в других аналогичных системах. Если говорить более конкретно, сервисные пакеты, сервисы которых сосредоточены на одном или более классах сущности (см. подраздел «Классы сущностей» данной главы), могут быть многократного использованы в различных системах, создаваемых для того же бизнеса или области. Это возможно потому, что классы сущностей получены из бизнес-классов сущностей или доменных классов, что делает классы сущностей и связанные с ними сервисы кандидатами на многократное использование внутри бизнеса или области в целом и большинства систем, которые с ними работают, а не только

данной конкретной системы. Сервисные пакеты и сервисы ортогональны вариантам использования, в том смысле, что сервисный пакет может быть использован в нескольких различных реализациях вариантов использования. Это особенно верно, когда сервисный пакет находится на общем уровне, наравне с общими и совместно используемыми функциями (см. подраздел «Определение зависимостей между пакетами анализа»). Такой сервисный пакет прекрасно подходит для многократного использования в различных приложениях (конфигурациях) системы, поддерживающих варианты использования, реализация которых нуждается в сервисном пакете. Сервисные пакеты переходят в сервисные подсистемы стадии проектирования (см. подраздел «Сервисные подсистемы» главы 9), а затем в компоненты реализации (см. подраздел «Артефакт: Компонент» главы 10). Эти компоненты могут многократно использоваться по тем же причинам, что и сервисные пакеты.

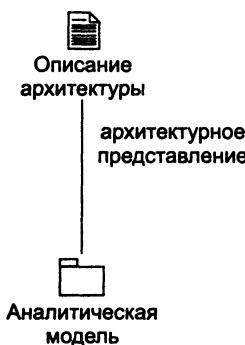


Рис. 8.14. Описание архитектуры содержит архитектурное представление аналитической модели

Итак, сервисные пакеты являются нашим первым инструментом для построения многократно используемых фрагментов в ходе анализа. Это вклад в проектирование и реализацию системы, дающий поразительный выход повторно используемых компонентов.

Артефакт: Описание архитектуры (представление модели анализа)

Описание архитектуры содержит архитектурное представление аналитической модели (приложение B), описывая ее архитектурно значимые артефакты (рис. 8.14).

Архитектурно значимыми обычно считаются следующие артефакты аналитической модели:

- Разложение аналитической модели на пакеты анализа и их зависимости. Это разложение часто выделяет подсистемы верхних уровней проектирования и реализации и потому значительно для всей архитектуры системы.
- Ключевые классы анализа, типа классов сущностей, которые инкапсулируют важные для прикладной области явления; граничные классы, которые инкапсулируют важные механизмы коммуникационных и пользовательских интер-

фейсов; классы управления, которые инкапсулируют важные последовательности с большим распространением (то есть координирующие важные реализации вариантов использования); и классы анализа, которые являются общими, центральными или имеют много связей с другими классами анализа. Обычно как архитектурно значимый рассматривается абстрактный класс, но не его подклассы.

- Реализации вариантов использования, которые реализуют некую важную или критическую функциональность, включают в себя множество классов анализа, и вследствие этого имеют большое поле действия, возможно, пересекаясь с несколькими пакетами анализа; или сосредоточены на некотором важном варианте использования, который должен быть рано разработан в жизненном цикле программы и вследствие этого должен находиться в описании архитектуры аналитической модели (приложение В).

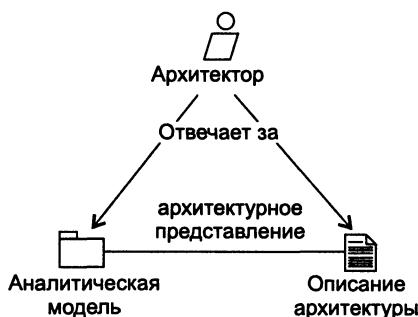


Рис. 8.15. Обязанности архитектора в процессе анализа

Сотрудники

Сотрудник: Архитектор

В течение рабочего процесса анализа архитектор несет ответственность за целостность аналитической модели, гарантируя, что аналитическая модель в целом правильна, согласована и читаема (рис. 8.15). Для больших и сложных систем эти обязательства после нескольких итераций могут потребовать обслуживания, а это работа довольно рутинная. В таких случаях архитектор может поручить эту работу другому сотруднику, возможно, инженеру по компонентам «высокого уровня» (см. подраздел «Сотрудник: Инженер по компонентам»). Однако архитектор останется ответственным за то, что важно для архитектуры, — за описание архитектуры. Другой сотрудник может отвечать за высокоуровневые пакеты аналитической модели, которые должны быть согласованы с описанием архитектуры.

Модель анализа верна, когда она реализует функциональные возможности, описанные в модели вариантов использования, и только их.

Архитектор также отвечает за архитектуру аналитической модели, то есть за существование ее архитектурно значимых частей, описанных в архитектурном

представлении модели. Напомним, что это представление — часть описания архитектуры системы.

Заметим, что архитектор не отвечает за непрекращающееся развитие и обслуживание различных артефактов аналитической модели. За это отвечает соответствующие инженеры по вариантам использования и компонентам (см. следующие подразделы).

Сотрудник: Инженер по вариантам использования

Инженер по вариантам использования отвечает за целостность одного или нескольких реализаций вариантов использования, гарантируя, что они выполняют требования, предъявляемые к ним (рис. 8.16). Реализация варианта использования должна правильно реализовать поведение соответствующего ей варианта использования из модели вариантов использования, и только его. Это включает обеспечение читаемости и соответствия своим задачам всей графической и текстовой документации, описывающей реализацию варианта использования.



Рис. 8.16. Обязанности инженера по вариантам использования в процессе анализа

Заметим, что инженер по вариантам использования не отвечает за классы анализа и отношения, задействованные в реализации варианта использования. За это отвечает инженер по компонентам (см. следующий подраздел).

Как мы увидим в следующей главе, инженер по вариантам использования также отвечает и за проект реализации варианта использования. Таким образом, инженер по вариантам использования ответствен и за анализ, и за проектирование варианта использования. Это сделано для того, чтобы переход от анализа к проектированию не вызывал дополнительных проблем.

Сотрудник: Инженер по компонентам

Инженер по компонентам определяет и поддерживает обязанности, атрибуты, отношения и специальные требования одного или нескольких классов анализа, проверяя, выполняет ли каждый класс анализа требования, сделанные на основе соответствующих требований реализаций вариантов использования, в которых он участвует (рис. 8.17).

Инженер по компонентам также сохраняет целостность одного или нескольких пакетов анализа. Это подразумевает контроль за тем, чтобы их содержание (например, классы и их отношения) было верно и чтобы их зависимость от других пакетов анализа также была правильна и минимальна.

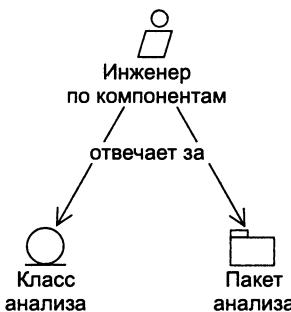


Рис. 8.17. Обязанности инженера по компонентам в процессе анализа

Часто бывает разумно назначить инженера по компонентам, отвечающего за пакет анализа, ответственным также и за входящие в него классы анализа. Кроме того, если имеется прямое соответствие между пакетом анализа и соответствующими подсистемами проекта (см. подраздел «Артефакт: Пакет анализа»), инженер по компонентам должен также отвечать и за эти подсистемы и использовать знания, приобретенные в ходе анализа, при проектировании и исполнении пакета анализа. Если такого прямого отображения нет, в проектирование и исполнение пакета анализа могут быть вовлечены дополнительные инженеры по компонентам.

Рабочий процесс

Ранее в этой главе мы описывали работу по анализу в статических понятиях. Теперь мы используем диаграмму деятельности для разговора о ее динамике (рис. 8.18).

Создание модели анализа (как определено ранее в этой главе) начинается архитекторами, которые начинают работу с выделения основных пакетов анализа, очевидных классов сущностей и общих требований. Затем в дело вступают инженеры вариантов использования, реализующие каждый вариант использования в понятиях участвующих в нем классов анализа, формулируя требования к поведению каждого класса. Эти требования затем доопределяются инженерами по компонентам и интегрируются в соответствующие классы, создавая согласованные обязанности, атрибуты и отношения для каждого класса. В течение всего процесса анализа архитектор находит новые пакеты анализа, классы и общие требования, и, поскольку аналитическая модель развивается, инженеры по компонентам, ответственные за индивидуальные пакеты анализа, непрерывно уточняют и дорабатывают эти пакеты.

Деятельность: Анализ архитектуры

Цель анализа архитектуры — в построении наброска аналитической модели и архитектуры, отыскании пакетов анализа, очевидных классов анализа и общих специальных требований (рис. 8.19).

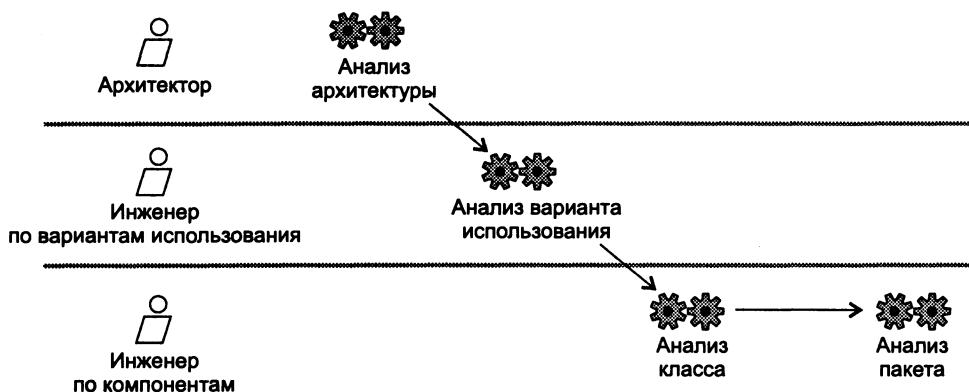


Рис. 8.18. Рабочие процессы анализа, включая участников и их действия

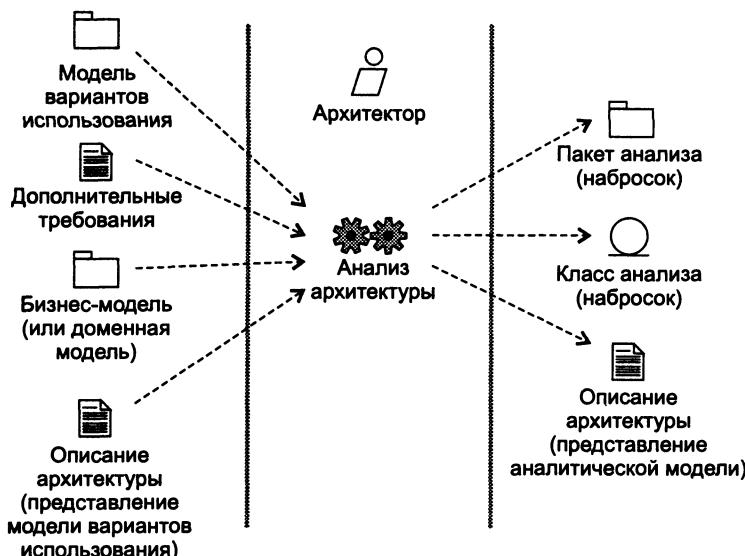


Рис. 8.19. Исходные данные и результат анализа архитектуры

Идентификация пакетов анализа

Пакеты анализа применяются для того, чтобы разделить модель анализа на меньшие, легче управляемые части. Они могут быть или идентифицированы в начале

работы в виде способа разделения работы по анализу, или найдены в ходе развития и вырастания аналитической модели в большую структуру, которая должна быть разделена.

Первичная идентификация пакетов анализа, естественно, совершается на основе функциональных требований и прикладной области, то есть рассматриваемого бизнеса или приложения. Так как мы фиксируем функциональные требования как варианты использования, непосредственный способ опознать пакеты анализа состоит в том, чтобы распределить основную часть всех вариантов использования к одному пакету, а затем реализовывать соответствующую функциональность внутри пакетов. Соответствующее «распределение» вариантов использования по определенным пакетам включает следующее.

- Варианты использования, необходимые для поддержки определенного бизнес-процесса.
- Варианты использования, необходимые для поддержки определенного актанта системы.
- Варианты использования, которые связаны через отношения обобщения и расширения. Такие наборы вариантов использования когерентны в том смысле, что варианты использования или специализируют, или обобщают друг друга.

Такие пакеты локализуют изменения в бизнес-процессах, поведении актантов и наборах тесно связанных вариантов использования соответственно. Это приближение лишь помогает с самого начала распределять варианты использования по пакетам. Варианты использования обычно не привязаны к одному пакету, а используются в нескольких. Следовательно, поскольку работа по анализу продолжается, когда варианты использования будут реализованы как кооперация между классами, возможно, в различных пакетах, возникнет уточненная структура пакетов.

Пример. Идентификация пакетов анализа. Этот пример показывает, как в Interbank Software могли бы извлекать некоторые пакеты анализа из модели вариантов использования. Варианты использования *Оплатить Счет*, *Послать Напоминание* и *Послать счет покупателю* вовлечены в один и тот же бизнес-процесс *Продажа: От Заказа до Поставки*. Следовательно, их можно включить в один пакет анализа.

Однако Interbank Software собирается предлагать свою систему различным клиентам, у которых будут различные потребности. Некоторые клиенты используют систему только как покупатели, другие только как продавцы, а некоторые клиенты — и так и эдак. Разработчики из Interbank Software решают отделить реализацию вариантов использования для продавца от реализации вариантов использования для покупателя. Мы можем увидеть, как исходное предположение об одном пакете анализа для бизнес-процесса *Продажа: От Заказа до Поставки* было исправлено для выполнения требований клиентов. Результатом стали два пакета анализа, которые могут быть поставлены клиентам отдельно друг от друга в зависимости от их потребностей: *Управление Счетами для Покупателя* и *Управление Счетами для Продавца* (рис. 8.20).

Заметим, что в бизнес-процессе имеются также и другие варианты использования, но мы их не учитывали, чтобы пример получился более простым.

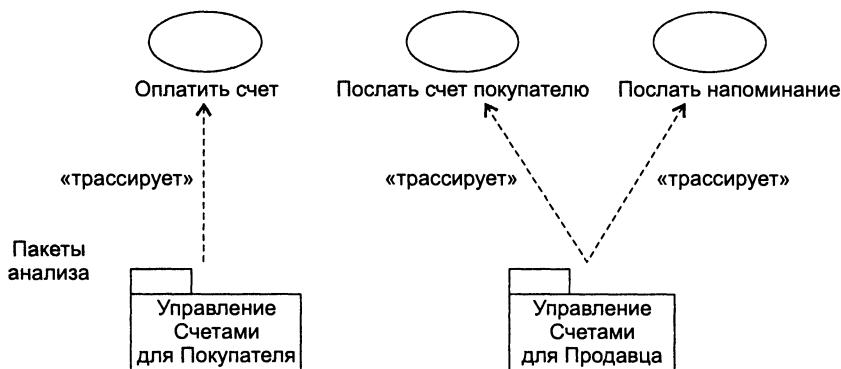


Рис. 8.20. Поиск пакетов анализа в вариантах использования

Работа с совместно используемыми частями пакетов анализа

Нередки случаи, когда среди пакетов, идентифицированных в предыдущем пункте, обнаруживаются общие фрагменты. Например, два или более пакета анализа должны совместно использовать один и тот же класс анализа. Чтобы разобраться в этой ситуации, надо извлечь общий класс, поместить его в его собственный пакет или просто снаружи всех пакетов и указать другим пакетам зависимость от этого более общего пакета или класса.

Весьма вероятно, что такие общие классы, представляющие собой совместно используемые части пакетов, будут классами сущностей, которые происходят из классов сущности домена или бизнес-классов сущностей. Таким образом, следует исследовать классы сущностей домена или бизнес-классы сущностей на предмет возможности совместного использования и найти общие пакеты на фазе анализа.

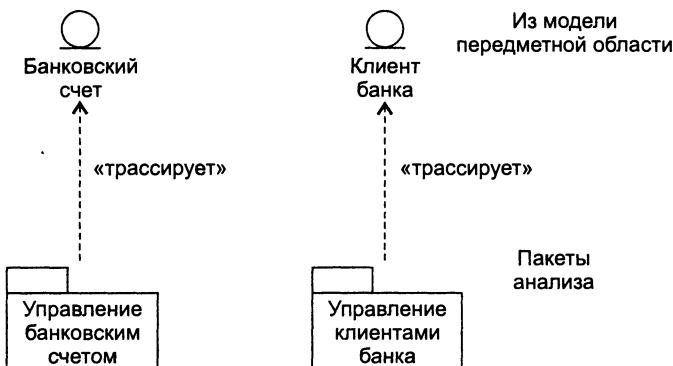


Рис. 8.21. Поиск основных пакетов анализа в классах предметной области

Пример. *Определение общих пакетов анализа.* Этот пример поясняет, как в Interbank Software могли бы опознавать общие пакеты анализа по модели предметной области. Каждый из классов предметной области *Клиент банка* и *Банковский счет* представляет собой в реальном мире важные и сложные объекты. Interbank пони-

мает, что эти классы требуют сложной поддержки информационной системы и что их совместно используют другие, в том числе более специфические пакеты анализа. Поэтому Interbank Software создаст отдельные пакеты, *Управление банковским счетом* и *Управление клиентами банка*, для каждого класса (рис. 8.21).

Отметим, что пакеты *Управление банковским счетом* и *Управление клиентами банка* будут, возможно, содержать множество классов анализа, например классы управления и граничные классы, относящиеся, соответственно, к управлению счетами и клиентами банка. В общем случае маловероятно, чтобы эти пакеты содержали только один или несколько классов сущностей, относящихся к соответствующим классам предметной области.

Идентификация сервисных пакетов

Далее в ходе работы по анализу часто производится идентификация соответствующих сервисных пакетов, в ходе которой тщательно разбираются функциональные требования и создается большинство классов анализа. Все классы анализа в некотором сервисном пакете работают на один сервис.

Для идентификации сервисного пакета следует сделать следующее: идентифицировать по одному сервисному пакету на каждый необязательный сервис. Сервисный пакет будет отдельным модулем.

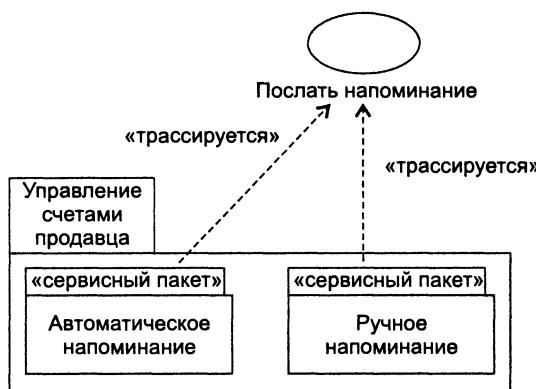


Рис. 8.22. Сервисные пакеты Автоматическое напоминание и Ручное напоминание находятся в пакете Управление счетами продавца

Пример. Пакеты необязательных сервисов. Большинство продавцов, использующих систему Interbank, желают иметь услугу по рассылке напоминаний. Этот сервис описан в (необязательном) варианте использования *Послать Напоминание*. Некоторые продавцы хотят, чтобы напоминания рассылались автоматически в том случае, если оплата по счету будет просрочена, а другие предпочитают, чтобы система просто предупредила их об этом факте, а они уже сами решат, стоит ли посыпать напоминание. Эти варианты представлены двумя необязательными и взаимоисключающими сервисными пакетами: *Автоматическое напоминание*, используемое для автоматической рассылки напоминаний, и *Ручное напоминание*, уведомляющее продавца, который решает, в какой форме преподнести это известие покупателю (рис. 8.22). Если продавец вообще не желает какой-либо поддерж-

ки напоминаний, в состав системы не включается ни один из этих пакетов. Сервисные пакеты входят в состав пакета *Управление счетами продавца*.

Определяйте по одному пакету на каждый сервис, который признан необязательным, даже в том случае, если воспользоваться им пожелают все клиенты. Сервисные пакеты содержат функционально связанные классы, и в результате изменения в структуре пакета будут сводиться к изменениям в отдельных сервисных пакетах. Этот критерий можно описать следующим образом: определяйте один сервисный пакет на каждый сервис, предоставляемый функционально связанными классами. Например, если класс А и класс В функционально связаны, происходит следующее: изменения в А обычно требуют изменений в В. Удаление А делает В ненужным. Объекты А постоянно взаимодействуют с объектами В, возможно, посредством различных сообщений.

Пример. Определение сервисных пакетов, которые содержат функционально связанные классы. Пакет *Управление банковскими счетами* включает в себя основной сервисный пакет *Банковские счета*, используемый для организации доступа к банковским счетам тех видов деятельности, которым это необходимо, например перевода денег или получения истории транзакций. Пакет также включает в себя сервисный пакет под названием *Риски* для оценки рисков, связанных с данным банковским счетом. Эти различные сервисные пакеты используются совместно в некоторых реализациях вариантов использования (рис. 8.23).

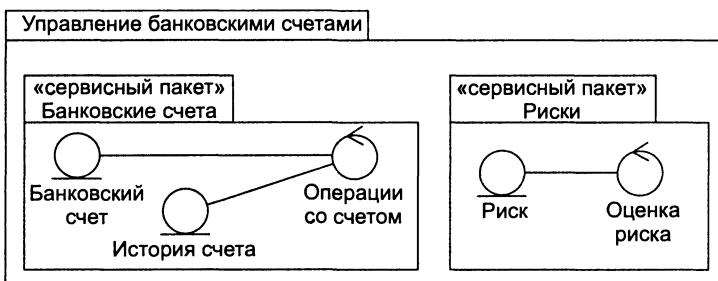


Рис. 8.23. Сервисные пакеты, Банковские счета и Риски, каждый из которых содержит функционально связанные классы

Определение зависимостей между пакетами анализа

Зависимости между пакетами анализа возникают в том случае, если их содержимое связано между собой. Направление этой зависимости будет совпадать с направлением (навигации между ними) зависимости между ними.

Наша цель — найти пакеты, которые относительно независимы и слабо связаны, но имеют высокую внутреннюю связность. Высокая внутренняя связность и низкая внешняя делают такие пакеты легкими в обслуживании, поскольку изменение некоторых классов внутри пакета скажется в основном на классах, входящих в этот пакет. Весьма разумно постараться уменьшить число связей между классами различных пакетов, поскольку это уменьшит зависимость между пакетами.

Для выделения зависимостей можно использовать уровни модели анализа, чтобы разложить специфические для приложения пакеты на верхнем уровне, а более

общие пакеты — на нижнем. Это позволяет внести ясность в разницу между специфическим и общим функциональными уровнями.

Пример. Уровни и зависимости между пакетами анализа. Пакет *Управление банковскими счетами* содержит некоторые классы, например *Банковский счет*, используемые другими пакетами. Так, класс *Банковский счет* используется классами из пакетов *Управление счетами покупателя* и *Управление счетами продавца*. Эти пакеты, таким образом, зависят от пакета *Управление банковскими счетами* (рис. 8.24).

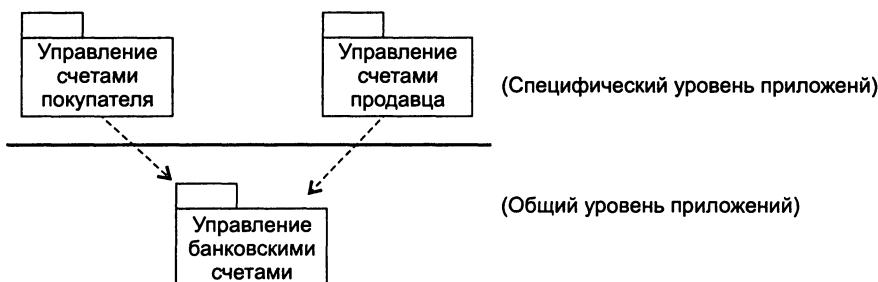


Рис. 8.24. Уровни и зависимости между пакетами анализа

В ходе проектирования и разработки мы уточним эти уровни и добавим еще множество нижних уровней, руководствуясь в основном нуждами среды исполнения и нефункциональными требованиями.

Определение очевидных классов сущностей

Часто бывает предпочтительно разработать предварительное предложение наиболее важных (от 10 до 20) классов сущностей, основываясь на классах предметной области или бизнес-сущностях, выделенных в ходе определения требований. Однако большинство классов идентифицируется в ходе реализации вариантов использования (о деятельности по анализу вариантов использования см. подраздел «Деятельность: Анализ варианта использования»). Поэтому следует быть осторожнее с определением большого количества классов на этой стадии — есть опасность запутаться во множестве деталей. Достаточно будет исходного наброска важных для архитектуры классов (см. подраздел «Артефакт: Описание архитектуры (представление модели анализа)»). С другой стороны, может потребоваться переделать очень много работы после того, как впоследствии для определения действительно необходимых классов сущностей будут применены варианты использования. Необходимы классы сущностей, участвующие в реализациях вариантов использования. Классы сущностей, не принимающие участия в реализациях вариантов использования, необходимыми не являются.

Для определения предварительного набора ассоциаций между соответствующими классами сущностей могут использоваться агрегация и ассоциация между классами предметной области в модели предметной области (или между бизнес-сущностями в бизнес-модели).

Пример. Класс *сущности, определенный из класса предметной области*. Счет — это класс предметной области, упоминавшийся в главе 6. Мы будем использовать

этот класс предметной области для получения одного из первичных классов сущности. Для начала мы предложим для класса *Счет* некоторые атрибуты: сумма, день выписки и крайний срок оплаты. Мы также можем определить ассоциацию между классами сущности доменной модели, например такую, как ассоциация платежа между *Заказом* и *Счетом*.

Определение общих специальных требований

Специальные требования — это требования, которые выявляются на этапе анализа. Их необходимо выявить, поскольку они оказывают непосредственное влияние на последующие процессы проектирования и реализации. Примерами могут быть ужесточение ограничений на:

- персистентность данных;
- распределение и параллельность;
- особенности безопасности;
- устойчивость к сбоям;
- управление транзакциями.

Архитектор несет ответственность за определение общих специальных требований, которые будут использованы разработчиками для определения специальных требований для конкретных реализаций вариантов использования и классов анализа. В некоторых случаях специальные требования не выходят на передний план, обнаруживаясь в ходе разработки реализаций вариантов использования и классов анализа. Заметим также, что определение нескольких различных специальных требований для класса или варианта использования — это необычный случай.

Для того чтобы поддержать последующее проектирование и реализацию, следует определить основные характеристики каждого из общих специальных требований.

Пример. *Определение основных характеристик специальных требований.* Требования, относящиеся к хранению, имеют следующие характеристики.

- Диапазон размеров: диапазон размеров объектов, которые должны храниться длительное время.
- Объем: число объектов, которые должны храниться длительное время.
- Период хранения: срок, в течение которого в среднем должны сохраняться персистентные объекты.
- Частота обновлений: частота обновлений объектов.
- Надежность: надежность показывает, каким образом объекты будут защищены от сбоев программного или аппаратного обеспечения.

Характеристики каждого из специальных требований должны быть приписаны к каждому классу или реализации варианта использования, которые ссылаются на эти специальные требования.

Деятельность: Анализ варианта использования

Мы анализируем варианты использования для того, чтобы (рис. 8.25):

- Определить классы анализа, объекты которых нуждаются в осуществлении потока событий варианта использования.

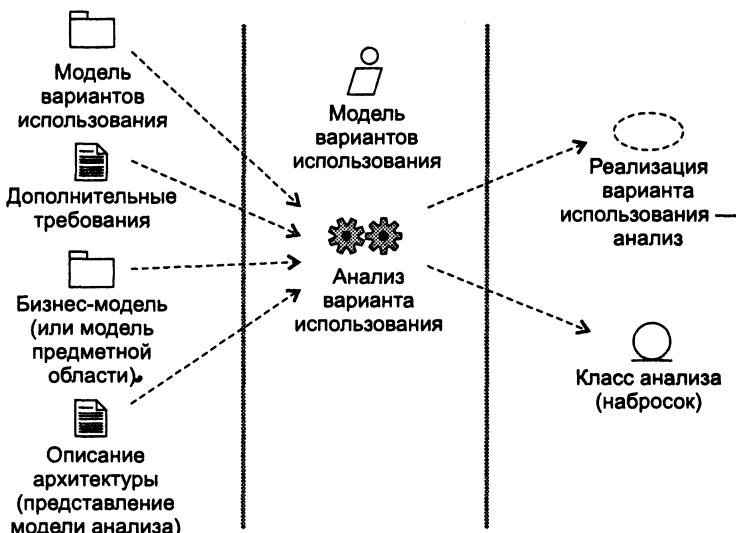


Рис. 8.25. Исходные данные и результат анализа варианта использования

- Разнести поведение варианта использования по взаимодействующим объектам анализа.
- Определить специальные требования к реализации варианта использования.

Другое название анализа варианта использования — это *уточнение* варианта использования. Мы уточняем каждый вариант использования в форме кооперации классов анализа.

Определение классов анализа

На этом шаге мы определяем управляющие, граничные классы и классы сущностей, необходимые для того, чтобы реализовать варианты использования, и набираем их названия, ответственности, атрибуты и отношения.

Варианты использования, определенные в процессе определения требований, не всегда детализированы настолько, чтобы можно было сразу заняться определением классов анализа. Информация о внутреннем строении системы неинтересна нам во время процесса определения требований и может попросту отсутствовать. Поэтому для определения классов анализа мы должны уточнить описание варианта использования в части, относящейся к внутреннему строению системы. Это уточнение может иметь вид текстового описания анализа потока событий реализации варианта использования.

При определении классов анализа можно придерживаться следующего общего порядка.

- Определите классы сущностей путем детального изучения описаний вариантов использования и любых существующих моделей предметной области и решите, какая информация должна быть включена в реализации вариантов использования. Однако будьте осторожны с информацией, которую лучше опре-

делять в виде атрибутов (см. подраздел «Определение атрибутов») — ее лучше включать в соответствующие граничные или управляющие классы — или информацией, которая в данной реализации варианта использования просто не нужна. Такую «информацию» нет нужды моделировать в классе сущности.

- Определите по одному основному граничному классу на каждого актанта-человека. Пусть этот класс отвечает за представление информации в главном окне интерфейса пользователя, с которым работает данный актант. Когда актант уже связан с граничным классом, вы можете решить переработать этот класс для того, чтобы добиться удобства работы с **пользовательским интерфейсом** (приложение В) и минимизировать число главных окон, необходимых каждому из актантов для работы с ним. Кроме того, эти основные граничные классы часто рассматриваются как наборы более простых граничных классов.
- Определите один простой граничный класс на каждый найденный ранее класс сущности. Эти классы будут соответствовать логическим объектам, с которыми через интерфейс пользователя взаимодействуют актанты-люди в ходе варианта использования. Эти простые граничные классы затем могут уточняться в соответствии с различными критериями удобства и способствовать построению «правильного» интерфейса пользователя.
- Определите по одному основному граничному классу на каждого внешнего системного актанта, и пусть этот класс соответствует коммуникационному интерфейсу. Напомним, что системным актантом могут быть любые программные или аппаратные модули, взаимодействующие с данной системой, например принтеры, терминалы, сигналы тревоги, датчики и т. д. Если коммуникации разделены на несколько уровней протоколов, может быть необходимо описать некоторые из этих уровней в модели анализа. Если это так, следует создать отдельные граничные классы для каждого из интересующих нас уровней протоколов.
- Определите один управляющий класс для осуществления управления и координации реализацией варианта использования и уточните этот управляющий класс в соответствии с требованиями к варианту использования. Например, в некоторых случаях управление лучше спрятать в граничный класс, особенно в том случае, когда большая часть управления осуществляется актантом. В таких случаях в управляющем классе нет необходимости. В других случаях управление может оказаться настолько сложным, что его требуется разложить на два или более класса управления. В этом случае управляющий класс будет нуждаться в разделении.

При осуществлении этих шагов классы анализа, которые уже присутствуют в модели анализа, разумеется, также участвуют в рассмотрении. Некоторые из них следует повторно использовать в текущей реализации варианта использования, некоторые осуществляются почти одновременно, что облегчит поиск классов анализа в отдельных реализациях вариантов использования.

Подбор классов анализа для реализации варианта использования отражается в диаграмме классов. Используйте эту диаграмму классов для показа отношений, имеющих место в реализации варианта использования.

Описание взаимодействий объектов анализа

После того, как мы получили описание классов анализа, необходимых для реализации варианта использования, нам следует описать процессы взаимодействия соответствующих объектов этих классов анализа. Это делается при помощи диаграмм кооперации, которые содержат участвующие во взаимодействии экземпляры актантов, объектов анализа и их связи. Если вариант использования содержит различные устойчивые потоки или подпотоки, часто бывает полезно создать по одной диаграмме кооперации на каждый поток. Это позволит сделать реализацию варианта использования более ясной и выделить диаграммы кооперации, отражающие общие и многократно используемые взаимодействия.

Диаграммы кооперации создаются начиная с начала потока событий варианта использования и продолжаются далее по одному шагу за раз, показывая, взаимодействия экземпляров каких объектов анализа и актантов необходимы для его реализации. Обычно объекты естественно занимают свое место в последовательности взаимодействий в реализации варианта использования. Относительно диаграмм кооперации следует заметить следующее.

- Вариант использования порождается сообщением, посылаемым актантом граничному объекту.
- Каждый класс анализа, обнаруженный на предыдущем шаге, будет иметь как минимум один объект анализа, изображенный в диаграмме кооперации. Если это не так, данный класс анализа лишний, поскольку он не принимает участия ни в одной реализации варианта использования.
- Сообщения не назначаются операциям, поскольку мы не определяем операций для классов анализа. Вместо этого сообщения будут обозначать склонность вызываемого объекта к взаимодействию с вызывающим объектом. Эта «склонность» — семя, которое разрастется в ответственность объекта, принимающего сообщения. Она должна иметь название соответствующей ответственности.
- Связи в диаграмме часто должны быть экземплярами ассоциаций между классами анализа. Или эти ассоциации должны быть уже созданы, или связи определяют требования к ассоциациям. На этом шаге все очевидные ассоциации должны быть описаны и изображены в диаграмме классов, ассоциированной с реализацией варианта использования.
- Номера и другие пометки в диаграмме, указывающие последовательность операций, не должны находиться в центре внимания и могут быть приведены отдельно в том случае, если их трудно понять или если они запутывают диаграмму. Основное внимание следует уделять отношениям (связям) между объектами и требованиям (определяемым сообщениями) каждого отдельного объекта.
- Диаграмма кооперации должна содержать все отношения рассматриваемого варианта использования. Например, если вариант использования А специализирует вариант использования В посредством отношения обобщения, диаграмма кооперации, описывающая вариант использования А, должна ссылаться на реализацию (то есть на диаграмму кооперации) варианта использования В.

В некоторых случаях необходимо дополнить диаграмму кооперации текстовыми пояснениями. Особенно это бывает нужно в том случае, когда один вариант использования описывается несколькими диаграммами или если диаграмма опи-

сывает сложные потоки. Это текстовое описание может быть сделано в ходе анализа потока событий реализации варианта использования.

Определение специальных требований

На этом шаге мы определяем все требования реализации варианта использования, в том числе нефункциональные требования. Определенные в ходе анализа, они будут использоваться в проектировании и реализации.

Пример. *Специальные требования реализации варианта использования.* Специальные требования, предъявляемые к реализации варианта использования *Оплатить счет*, включают в себя:

- Класс *Счет* должен быть классом длительного хранения.
- Класс *Обработчик заказов* должен быть в состоянии производить до 10 000 транзакций в час.

При определении этих требований можно сослаться на общие специальные требования, которые были при необходимости определены архитектором.

Деятельность: Анализ класса

Цели анализа класса (рис. 8.26).

- Определить и поддерживать ответственности класса анализа, основанные на его роли в реализации варианта использования.
- Определить и поддерживать атрибуты и отношения класса анализа.
- Определить специальные требования класса анализа.

Определение ответственности

Ответственности класса могут быть выбраны из комбинации всех ролей, которые он исполняет в различных реализациях вариантов использования. Мы можем найти все реализации вариантов использования, в которых участвует класс, изучая этот класс и диаграммы взаимодействий. Также напомним, что требования к каждой реализации варианта использования, в которой участвует этот класс, нередко определяются в текстовом виде в ходе анализа потока событий реализации варианта использования.

Пример. *Роли класса.* Объекты класса *Счет* создаются в варианте использования *Выписать счет покупателю*. Продавец осуществляет этот вариант использования для того, чтобы запросить покупателя оплатить свой заказ (заказ, который был создан в ходе варианта использования *Заказать товары или услуги*). В ходе этого варианта использования счет переправляется покупателю, который позже принимает решение об оплате.

Оплата является результатом варианта использования *Оплатить счет*, при этом объект *Планировщик Платежей* ведет учет объектов *Счет* и планирует проведение по ним платежей. После того как счет будет оплачен, объект *Счет* закрывается.

Заметим, однако, что один и тот же экземпляр *Счета* принимает участие в вариантах использования *Выписать счет покупателю* и *Оплатить счет*.

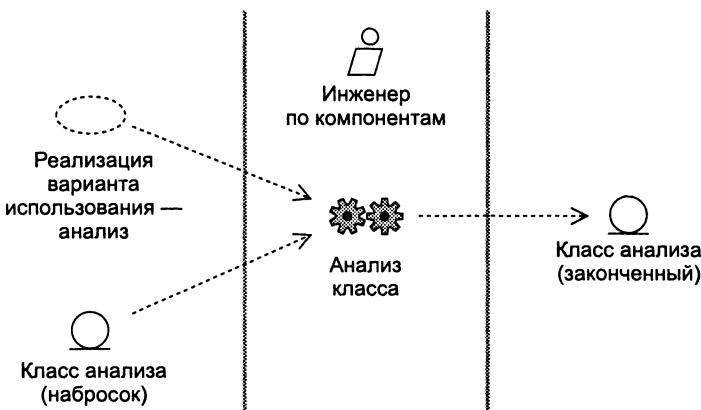


Рис. 8.26. Исходные данные и результат анализа класса

Существует несколько способов составить список обязанностей класса. Согласно упрощенному подходу, следует извлечь ответственности из каждой роли и добавить дополнительные или изменившиеся готовые ответственности из каждой реализации варианта использования.

Пример. *Ответственности класса. Планировщик оплат* имеет следующие ответственности:

- создание запроса на платеж;
- отслеживание запланированных платежей и посылка сообщений в случае, если платеж был произведен или прерван;
- инициирование перечисления денег в день оплаты;
- уведомление *Счета* о том, что он был помечен к оплате, и о том, что он был оплачен (то есть закрыт).

Определение атрибутов

Атрибут определяет свойства класса анализа и часто включается в требующие этого ответственности класса (как обсуждалось на предшествующем шаге). При определении атрибутов следует держать в голове следующую общую последовательность действий.

- Имя атрибута должно быть существительным [34, 60].
- Помните, что типы атрибутов в анализе концептуальны и, если это возможно, они не должны ограничиваться средой разработки. Так, например, «сумма» является вполне подходящим для анализа типом, а при проектировании соответствующим ей типом станет «целое».
- При выборе типа атрибута старайтесь использовать уже определенные типы.
- Одиночный экземпляр атрибута не может использоваться несколькими объектами анализа. Если это необходимо, определяйте для атрибута его собственный класс.
- Если класс анализа из-за своих атрибутов становится труден для понимания, некоторые из его атрибутов следует выделить в отдельные классы.

- Атрибуты классов-сущностей часто достаточно просты. Если класс сущности трассируется от класса предметной области или бизнес-сущности, атрибуты такого класса обычно представляют собой полезные исходные данные.
- Атрибуты граничных классов, которые взаимодействуют с актантами-людьми, часто представляют собой информацию, предназначенную для работы с актантами, например именованные текстовые поля.
- Атрибуты граничных классов, которые взаимодействуют с актантами-системами, часто представляют собой свойства коммуникационных интерфейсов.
- Атрибуты у управляющих классов встречаются редко, потому что эти классы имеют краткий срок жизни. Однако управляющий класс может иметь атрибуты, сохраняющие в ходе реализации варианта использования аккумулированные или порожденные значения.
- Иногда формальные атрибуты не нужны. Вместо них бывает достаточно использовать простое объяснение свойства, используемого классом анализа. Оно может быть помещено в описании ответственостей этого класса.
- Если у класса имеется много атрибутов или они сложны, можно проиллюстрировать эту ситуацию отдельной диаграммой класса, отражающей исключительно атрибуты.

Определение ассоциаций и агрегаций

Объекты анализа взаимодействуют друг с другом посредством связей, отображаемых в диаграмме кооперации. Эти связи часто представляют собой экземпляры ассоциаций между соответствующими классами. Инженер по компонентам должен исследовать связи, используемые в диаграмме кооперации, для того, чтобы определить необходимые ассоциации. Связи могут означать необходимость ссылок и агрегации между объектами.

Число связей между классами должно быть минимальным. Посредством агрегаций и ассоциаций моделируются не связи, существующие в реальности, а связи, создаваемые в ответ на запрос различных реализаций вариантов использования. При анализе не следует сосредоточиваться на моделировании оптимального поиска пути через агрегации и ассоциации. Это лучше делать на этапах проектирования и анализа.

Инженер по компонентам также определяет множество ассоциаций, имен ролей, ассоциаций «с самим собой», классов ассоциаций, ролей в ассоциациях, классов ассоциаций, упорядоченных ролей, условных ролей и **n-арных ассоциаций** (приложение А). Подробнее это рассматривается в работах [60, 57].

Пример. Ассоциация между классами анализа. *Счет* – это запрос на оплату одного или более заказов (рис. 8.27). Этот факт представлен в виде ассоциации, множественность которой равна «1..*» (поскольку с каждым счетом ассоциируется как минимум один заказ), и именем роли *Заказ к оплате*.

Агрегацию можно использовать в том случае, если объекты представляют собой:

- концепции, физически вложенные одна в другую, как, например, автомобиль содержит водителя и пассажиров;
- концепции, содержащие одна другую, как, например, автомобиль содержит двигатель и колеса;

- концепции, имеющие вид концептуальной коллекции объектов, как, например, семья содержит отца, мать и детей.



Рис. 8.27. Счет является запросом на оплату одного или более заказов

Определение обобщений

Обобщения могут быть использованы для получения в ходе анализа разделяемого или общего для нескольких классов анализа поведения. Обобщения могут быть сохранены на более высоком и концептуальном уровне. Их основная задача — сделать модель анализа понятнее.

Пример. *Определение обобщений.* Счета и Заказы имеют схожие ответственности. Оба они являются специализациями более общего Торгового объекта (рис. 8.28).

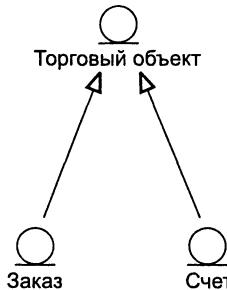


Рис. 8.28. Торговый объект обобщает Счет и Заказ

В ходе проектирования обобщения будут подправлены для того, чтобы лучше подходить под выбранную среду разработки, то есть язык программирования. Обобщение может исчезнуть, а вместо него появится другое отношение, например ассоциация.

Определение специальных требований

На этом шаге мы определяем все требования к классу анализа, которые определяются на стадии анализа, а реализованы будут на стадиях проектирования и реализации (например, нефункциональные требования.). При выполнении этой работы будьте готовы изучать специальные требования реализации варианта использования, которые могут содержать дополнительные (нефункциональные) требования к классу анализа.

Пример. Определение специальных требований в классе анализа. Характеристики требований по хранению объектов класса *Счет* могут быть описаны следующим образом:

- пределы размера: от 2 до 24 Кбайт на объект;
- объем: до 100 000;
- частота обновлений:
 - создание/удаление: 1000 в день;
 - обновление: 30 обновлений в час;
 - чтение: 1 в час.

При определении этих требований, по возможности, следует использовать ссылку на общие специальные требования, определенные архитектором.

Деятельность: Анализ пакетов

Целью анализа пакетов, как показано на рис. 8.29, является.

- Обеспечить, насколько это возможно, независимость данного пакета анализа от остальных.
 - Обеспечить выполнение пакетом анализа возложенных на него задач или реализацию определенных классов предметной области или вариантов использования.
 - Определить зависимости, по которым в будущем можно будет рассчитать эффект внесения изменений.
- Для этой деятельности общая последовательность действий будет следующей:
- Определить и поддерживать зависимости данного пакета от других пакетов, содержащих ассоциированные с ним классы.

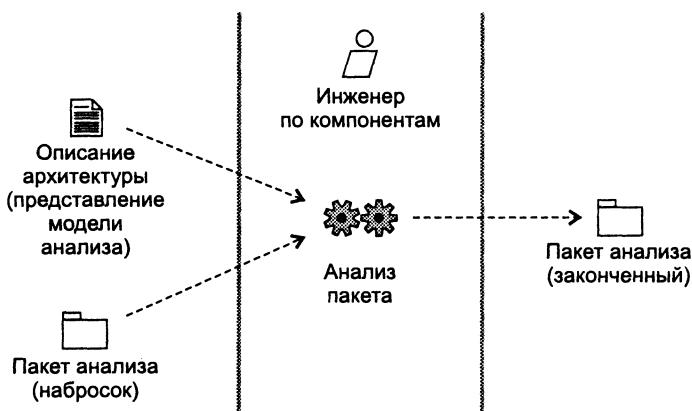


Рис. 8.29. Исходные данные и результат анализа пакетов

- Убедиться в том, что пакет содержит правильные классы. Постарайтесь сделать пакет связанным, включая в него только функционально-ориентированные классы.

- Ограничить зависимость данного пакета от других пакетов. Рассмотрите возможность перемещения в другие пакеты классов, которые сильно от них зависят.

Пример. Зависимости пакетов. Пакет *Управление счетами покупателей* содержит класс *Обработка счетов*, ассоциированный с классом *Банковский счет* пакета *Управление банковскими счетами*. Это требует соответствующей зависимости между пакетами (рис. 8.30).

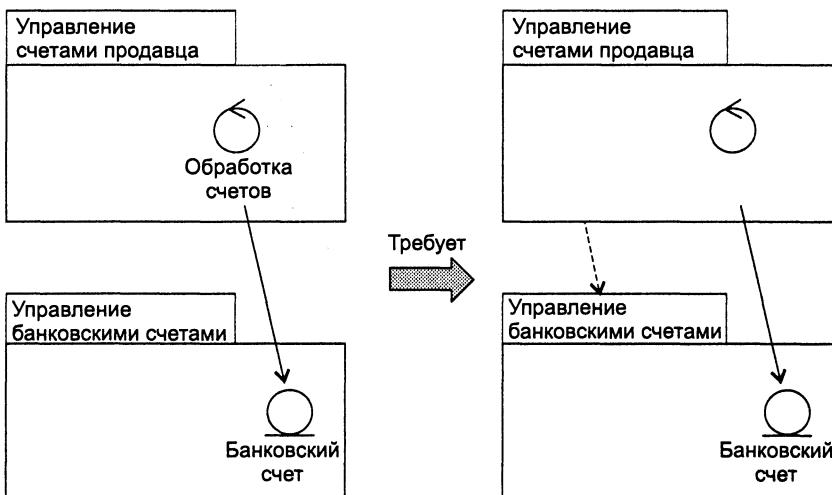


Рис. 8.30. Требуемые зависимости пакетов

Рабочий процесс анализа — резюме

Результатом рабочего процесса анализа (приложение В) является модель анализа. Это концептуальная модель объекта, в которой производится анализ требований. В ходе анализа требования уточняются и структурируются. Модель анализа включает в себя следующие элементы.

- Пакеты анализа и сервисные пакеты, их зависимости и содержание. Пакеты анализа могут локализовать вносимые в бизнес-процессы поведение актантов или набор тесно связанных вариантов использования. Сервисные пакеты локализуют изменения в отдельных сервисах, предоставляемых системой, и представляют собой основной инструмент построения в ходе анализа.
- Классы анализа, их ответственности, атрибуты, связи и специальные требования. Каждый из управляющих, граничных классов или классов сущности локализует изменения (стереотипного) поведения и информацию, которую он предоставляет. Изменения пользовательских или коммуникационных интерфейсов обычно локализуются в одном или более граничных классах, а изменения долгоживущей, обычно хранимой информации, использующейся в системе,

обычно содержатся в классах сущностей. Изменения же в управлении, координации, последовательности, транзакциях, а иногда и сложной бизнес-логике, затрагивающие несколько объектов (границых и/или сущности), обычно содержатся в одном или более классах управления.

- Реализации анализа вариантов использования, описывающие, как уточнены варианты использования в понятиях кооперации в аналитической модели и ее специальных требований. Изменения реализации вариантов использования содержатся в вариантах использования, поэтому если вариант использования изменяется, то его реализация изменяется вместе с ним.
- Архитектурное представление модели анализа, включающее ее архитектурно значащие элементы. **Представление архитектуры** (приложение В) локализует изменения в архитектуре системы.

Как мы писали в этой главе, модель анализа представляет из себя основные исходные данные для последующей деятельности по проектированию. При использовании модели анализа в этих целях мы как можно более полно сохраняем структуру, определенную нами при проектировании разрабатываемой системы. Мы добиваемся этого путем приведения основной части нефункциональных требований и других ограничений в соответствие со средой реализации. Говоря более точно, модель анализа вносит в модель проектирования следующее.

- Пакеты анализа и сервисные пакеты внесут основной вклад в подсистемы проектирования и сервисные подсистемы, на, соответственно, специфическом и общем уровнях приложения. Во многих случаях существует однозначная (изоморфная) трассировка между пакетами и соответствующими им подсистемами.
- Классы анализа служат спецификацией при проектировании классов. При проектировании классов анализа по различным стереотипам требуются различные навыки и технологии. Так, например, проектирование классов сущностей обычно требует использования технологий баз данных, в то время как проектирование граничных классов — использования технологий пользовательского интерфейса. Однако классы анализа с их ответственостями, атрибутами и связями используются в качестве (логических) исходных данных для создания соответствующих операций, атрибутов и отношений классов проекта. Кроме того, большая часть специальных требований, определенных в классах анализа, используется в соответствующих классах проектирования при обсуждении выбора технологии, будь то база данных или пользовательский интерфейс. Анализ реализации варианта использования преследует две задачи. Одна состоит в том, что он помогает создать более точное описание вариантов использования. Вместо того чтобы детализировать каждый вариант использования в модели вариантов использования при помощи диаграммы состояний или деятельности, мы описываем варианты использования в виде кооперации классов анализа. Это дает нам исчерпывающее формальное описание требований к системе.
- Анализы реализации варианта использования также применяются в качестве исходных данных при проектировании вариантов использования. Они помогают определить классы проектирования, необходимые для участия в соответствую-

ющих проектах реализации вариантов использования. Они также помогают сделать набросок исходной последовательности взаимодействий объектов проектирования. Кроме того, большинство специальных требований, определенных в анализе реализации вариантов использования, будут использованы в виде соответствующих проектов реализации вариантов использования при обсуждении вопроса о выборе технологии, будь то базы данных или интерфейсы пользователя.

Архитектурное представление модели анализа используется при создании архитектурного представления модели проектирования в качестве исходных данных. Вероятно, что элементы различных представлений (из различных моделей) трассируются однозначно. Это происходит потому, что понятие о важности для архитектуры имеет тенденцию плавно перетекать из модели в модель в соответствии с трассировкой.

9 Проектирование

Введение

При проектировании мы оформляем систему, придавая ей такую форму (и такую архитектуру), которая позволит нам внести в нее и поддерживать в рабочем состоянии все наши требования, в том числе нефункциональные требования и другие ограничения. Основными исходными данными для проектирования будут результаты анализа, в частности, модель анализа (табл. 9.1). Модель анализа дает нам детальное понимание требований. Наиболее важно то, что она содержит структуру системы, которую мы постараемся максимально сохранить при дальнейшем построении системы (см. раздел «Кратко об анализе» главы 8). Задачи проектирования, в частности, таковы.

- Получить глубокое понимание моментов, относящихся к нефункциональным требованиям и ограничениям, связанным с языками программирования, многократным использованием компонентов, операционными системами, технологиями распределенной и параллельной обработки, технологиями баз данных, управления транзакциями и т. д.
- Создать соответствующие исходные данные и базовые точки для последующей реализации определенных нами требований в подсистемах, интерфейсах и классах.
- Получить возможность разбить работу по реализации на множество управляемых частей для того, чтобы снабдить заданиями различные команды разработчиков, возможно, действующие параллельно. Это необходимо в тех случаях, когда декомпозиция не может быть сделана только на основании данных, полученных в результате определения требований (включая модель вариантов использования) или анализа (включая модель анализа). Примером могут быть случаи, когда реализация этих результатов нетривиальна.
- Определить основные интерфейсы между подсистемами на ранней стадии жизненного цикла системы. Это поможет нам объяснить архитектуру и даст воз-

можность использовать эти интерфейсы в качестве средства синхронизации разных команд разработчиков.



Рис. 9.1. Сотрудники и артефакты, вовлеченные в проектирование

- Получить возможность визуализировать и объяснить проект, используя типовую нотацию.
- Создать хорошую абстракцию реализации системы, понимая, что реализация есть простое уточнение проекта, связанное с наращиванием «мяса», но не с изменением структуры. Это допускает использование технологий автоматической генерации кода и циклической разработки в промежутке между проектированием и реализацией.

В этой и последующей главах мы расскажем, как этого добиться. Мы рассмотрим рабочий процесс проектирования точно так же, как делали это для анализа (рис. 9.1).

Роль проектирования в жизненном цикле разработки программного обеспечения

Проектирование находится в центре внимания разработчиков во время итераций окончания фазы проектирования и начала фазы построения (рис. 9.2). В ходе этого процесса мы получаем законченную, стабильную архитектуру и наброски модели реализации. Позже, в ходе фазы построения, когда архитектура окончательно стабилизируется и требования будут определены, внимание переключается на реализацию.

Поскольку модель проектирования весьма близка к реализации, будет естественно сохранить ее на все время жизненного цикла системы. Это особенно верно при циклической разработке, когда модель проектирования используется, чтобы визуализировать реализацию, и для поддержки технологий графического программирования.

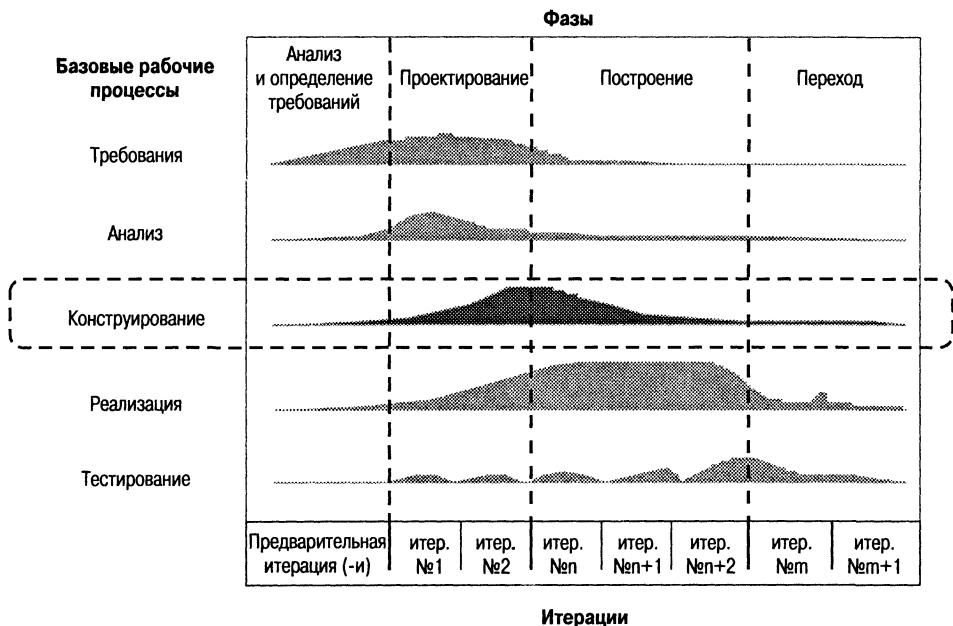


Рис. 9.2. Место проектирования в жизненном цикле системы

Артефакты

Артефакт: Модель проектирования

Модель проектирования — это объектная модель, которая описывает физическую реализацию вариантов использования. В этой модели основное внимание уделяется тому, каким образом функциональные и нефункциональные требования вместе с другими ограничениями, относящимися к среде реализации, составляют обсуждаемую систему. Кроме того, модель проектирования представляет собой абстракцию реализации системы и используется в качестве исходных данных для реализации.

Модель проектирования, как показано на рис. 9.3, определяет иерархию элементов.

Модель проектирования представляется в виде системы проектирования, обозначающей подсистему верхнего уровня в модели. Использование других подсистем — это способ разделить модель проектирования на легче управляемые части.

Подсистемы проектирования и классы проектирования представляют собой абстракции (приложение В) подсистем и компонентов реализации системы. Эти абстракции тривиальны и представляют собой простое отображение проекта на реализацию.

В модели проектирования варианты использования реализуются в виде классов проектирования и их объектов. Эта реализация представляется в виде кооперации и называется *проектом реализации варианта использования*. Отметим, что

проект реализации варианта использования отличается от анализа реализации варианта использования. Первый описывает реализацию варианта использования в понятиях взаимодействующих объектов проектирования, в то время как второй делает то же самое в понятиях взаимодействующих объектов анализа.

Артефакты модели проектирования представлены в деталях в следующем подразделе.

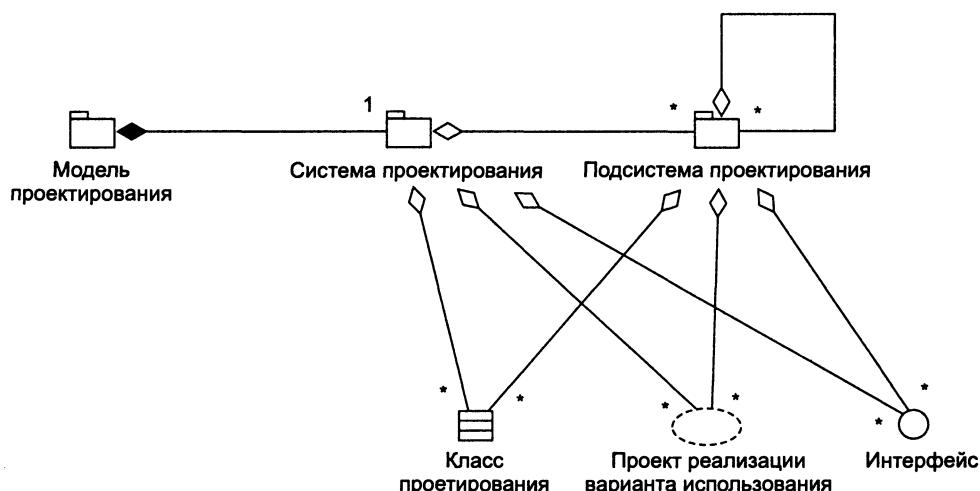


Рис. 9.3. Модель проектирования представляет собой иерархию подсистем проектирования, содержащих классы проектирования, проекты реализаций вариантов использования и интерфейсы

Артефакт: Класс проектирования

Класс проектирования — это наиболее приближенная к реальности абстракция класса или похожей конструкции реализации системы (рис. 9.4). Мы называем эту абстракцию наиболее близкой к реальности по следующим причинам.

- Язык, используемый для описания класса проектирования, тот же, что и язык программирования для реализации. Соответственно, операции, параметры, атрибуты, типы и другие подробности определяются с использованием синтаксиса выбранного языка программирования.
- Часто задается видимость атрибутов и операций класса проектирования. Например, в языке C++ для этого обычно используются ключевые слова *public*, *protected* и *private*.
- Отношения с другими классами, в которых участвует данный класс проектирования, часто получают явное выражение при реализации этого класса. Например, обобщение во всех стереотипах обобщения имеет семантику, которая соответствует обобщению (или наследованию) в языке программирования. Таким образом, обобщения и агрегации часто отображаются на соответствующие переменные (атрибуты) реализации, соответствующие ссылкам на объекты.

- Методы (то есть реализации операций) класса проектирования прямо отображаются на соответствующие методы классов реализации (то есть код). Методы, определяемые в ходе проектирования, часто определяются на естественном языке или на псевдокоде и могут быть в таком виде использованы в аннотации к реализации этих методов. Это одна из немногих серьезных абстракций, переходящих из проектирования в реализацию, и непосредственная причина нашей рекомендации задействовать для выполнения проектирования и реализации класса одного и того же разработчика.

Таблица 9.1. Краткое сравнение моделей анализа и проектирования

Модель анализа	Модель проектирования
Концептуальная модель, так как она является абстракцией системы и не затрагивает вопросов реализации	Физическая модель, так как она является наброском реализации
Не зависит от проекта (подходит к разным проектам)	Не является независимой, специфична для данной реализации
Три (концептуальных) стереотипа — «Сущность», «Управление» и «Граница»	Любое число (физических) стереотипов в классах, зависит от языка реализации
Слабо формализована	Сильно формализована
Невелика (соотношение с моделью проектирования 1:5)	Значительна по размеру (соотношение с моделью анализа 5:1)
Мало уровней	Много уровней
Динамическая (но последовательности уделяется мало внимания)	Динамическая (больше внимания к последовательности)
Набросок проекта системы, включая архитектуру	Описание проекта системы, включая архитектуру (одно из представлений)
В основном создается посредством «работы ногами», на семинаре или в похожих условиях	В основном создается путем «визуального программирования» в среде циклической разработки, модель проектирования находится в «циклической разработке» с моделью реализации (описывается в главе 10)
Может не поддерживаться в течение всего жизненного цикла системы	Должна поддерживаться в течение всего жизненного цикла системы
Определяет структуру, которая будет использована при оформлении системы, включая создание классов проектирования	Оформляет систему, сохраняя, насколько это возможно, структуру, определенную моделью анализа

- Класс проектирования может переложить обработку некоторых требований на последующую реализацию, передав ей *требования к реализации* класса. В результате становится возможным отложить принятие решений, которые невозможно сделать на основе модели проектирования, например, касающихся вопросов кодирования класса.
- Класс проектирования часто задается стереотипом, который напрямую отображается в конструкцию соответствующего языка программирования. Так, класс проектирования для приложения на Visual Basic будет содержать стереотипы «модуль класса», «форма», «элемент управления» и т. д.

- Класс проектирования может быть реализован — и реализуется — в виде интерфейса, если это понятие существует в выбранном языке программирования. Например, класс проектирования, представляющий класс языка Java, может быть реализован в виде интерфейса.

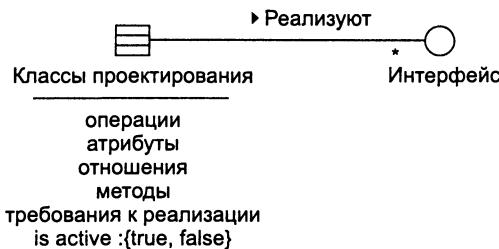


Рис. 9.4. Ключевые атрибуты и ассоциации класса проектирования

- Класс проектирования может быть активным. Это означает, что объекты класса будут использовать свою собственную нить управления, работая параллельно с другими активными объектами. Однако обычно классы проектирования неактивны. Это означает, что все их объекты «запускаются» в едином адресном пространстве под управлением другого активного объекта. Детали семантики этого процесса зависят от языка программирования и используемых в нем технологий параллельной или распределенной обработки. Отметим важность различия между активными классами и неактивными. Активные классы могут, следовательно, в качестве альтернативы, содержаться не только в модели проектирования, но и в их собственной модели процесса. Такое характерно для случая множества активных классов, объекты которых тесно интегрированы, — например, в системах реального времени.

Пример. Класс проектирования Счет. На рис. 9.5 изображен класс проектирования *Счет*, в таком виде, как он был разработан при проектировании. Атрибут *Банковский счет*, внесенный при анализе, превратился в ассоциацию с классом *Банковский счет*.

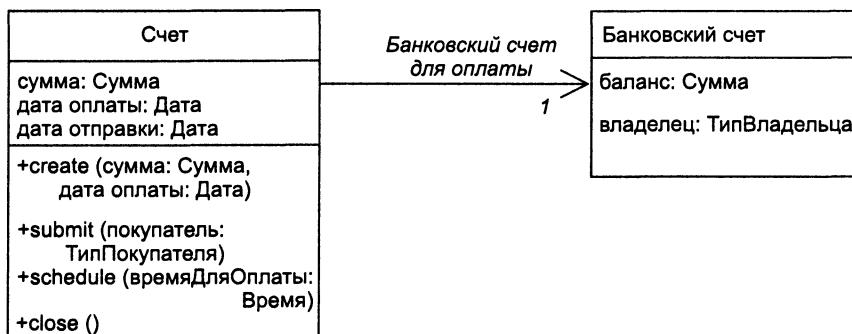


Рис. 9.5. Класс проектирования Счет с его атрибутами, операциями и ассоциацией, которая связывает объект Счет с объектом Банковский счет

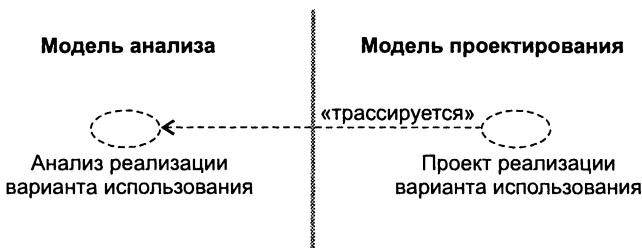


Рис. 9.6. Трассировки между реализациами варианта использования в различных моделях

Артефакт: Проект реализации варианта использования

Проект реализации вариантов использования — это кооперация, входящая в модель проектирования, которая определяет реализацию конкретного варианта использования в терминах классов проектирования и их объектов. *Проект реализации вариантов использования* непосредственно трассируется в *анализ реализации варианта использования* из модели анализа (рис. 9.6). Отметим, что проект реализации вариантов использования также может быть через анализ реализации варианта использования трассирован в вариант использования из модели вариантов использования.

Если модель анализа используется только для построения хорошей модели проектирования и не сохраняется в течение всего жизненного цикла системы, анализы реализации вариантов использования также не будут сохранены. Зависимость трассировки будет тогда напрямую вести от проекта реализации варианта использования к варианту использования из соответствующей модели.

Проект реализации вариантов использования содержит текстовое описание потока событий, диаграммы классов, отображающие участвующие в нем классы проектирования, и диаграмму взаимодействий, отображающую конкретный поток событий сценария варианта использования в терминах взаимодействий между объектами проектирования (рис. 9.7). Если в этом существует необходимость, диаграммы также могут отображать подсистемы и интерфейсы, входящие в реализацию варианта использования (то есть подсистемы, содержащие входящие в него классы проектирования).

Проект реализации вариантов использования осуществляет физическую реализацию анализа реализации вариантов использования, с которым он связан за счет трассировки, и поддерживает большинство нефункциональных требований (то есть специальных требований), определенных в анализе реализации варианта использования. Однако проект реализации варианта использования может, поскольку такая возможность есть у классов проектирования, возлагать ответственность за осуществление некоторых требований на следующий за ним процесс реализации, извещая его об этом посредством *требований к реализации*.

Диаграммы классов

Класс проектирования и его объекты, равно как и подсистема, в которую входит класс проектирования, часто участвуют в реализациях вариантов использования.

Кроме того, в определенных случаях некоторые операции, атрибуты и ассоциации определенного класса относятся только к одной реализации варианта использования. Это может быть важно для координации всех требований к классу и его объектам, а также содержащей этот класс подсистеме, которая участвует в различных вариантах использования. Для того чтобы понять ситуацию, мы используем диаграммы классов, добавляемые к реализации варианта использования, на которых показаны участвующие в ней классы, подсистемы и другие зависимости. Так мы можем отследить участвующие в реализации варианта использования элементы.



Рис. 9.7. Ключевые атрибуты и ассоциации проекта реализации варианта использования

Примеры диаграммы класса приведены в подразделах «Определение участвующих классов проектирования» и «Определение участвующих подсистем и интерфейсов» данной главы.

Диаграммы взаимодействий

Последовательность действий варианта использования начинается с того, что актант инициирует вариант использования, посыпая системе некоторое сообщение.

Если мы рассмотрим «внутренности» системы, мы обнаружим, что некий объект проектирования принимает от актанта это сообщение. Затем этот объект проектирования вызывает другие объекты, и все вовлеченные объекты взаимодействуют между собой с целью выполнить вариант использования. В ходе проектирования мы предпочитаем изображать этот процесс при помощи диаграмм последовательностей, которые фокусируют наше внимание на получении детальной хронологической последовательности взаимодействий.

В некоторых случаях мы включаем в диаграммы последовательностей и подсистемы. Это делается для того, чтобы определить, какие подсистемы участвуют в реализации варианта использования и, возможно, какие интерфейсы, предоставляемые этими подсистемами (то есть реализованные в них) участвуют в этом. Для этого следует сначала спроектировать на верхнем уровне варианты использования, и лишь затем заниматься проектированием внутреннего строения участвующих в них подсистем. Это может быть использовано, например, для определения интерфейсов подсистем на ранних стадиях цикла разработки программного обеспечения, до начала разработки их внутреннего строения.

Используя диаграммы последовательностей, мы иллюстрируем взаимодействие объектов при помощи обмена сообщениями между линиями жизни объектов и подсистем. Когда мы говорим, что то или иное сообщение было «принято» линией жизни подсистемы, мы имеем в виду, что в подсистеме существует объект класса, который и принял это сообщение. Когда сообщение «посыпается» линией жизни системы, это означает, что некий объект класса, имеющийся в подсистеме, послал сообщение. Имя сообщения должно указывать на операцию внутреннего объекта или интерфейса, предоставляемого объектом.

Примеры диаграмм взаимодействия можно найти в подразделе «Описание взаимодействия объектов проектирования» данной главы.

Поток событий — проект

Диаграммы реализации вариантов использования, особенно диаграммы взаимодействий, часто трудно читать без вспомогательных средств. В качестве такого средства может выступать проект потока событий, представляющий из себя текстовое описание, поясняющее и дополняющее диаграммы и метки на них. Текст должен быть составлен в терминах объектов, взаимодействующих для осуществления вариантов использования, или в терминах участвующих в этом взаимодействии подсистем. Описание не должно, однако, пояснять все атрибуты, операции или ассоциации объекта. Подобное описание будет трудно поддерживать, поскольку атрибуты, операции и ассоциации часто изменяются. Кроме того описание не должно включать в себя какие-либо операции с интерфейсами, если они упоминаются в диаграммах. Пользуясь таким подходом, мы минимизируем необходимость обновления описания проекта потока событий в случае изменения описываемых диаграмм. Особенно это характерно для операций, указанных в сообщениях диаграмм взаимодействия.

Проект потока событий особенно часто используется в случае нескольких диаграмм последовательностей, описывающих одну и ту же реализацию или диаграмм, описывающих сложные потоки. Отметим следующее:

- Проект потока событий реализации варианта использования не ограничен отдельной диаграммой. Он может быть использован и для описания связи нескольких диаграмм.
- Метки (отметки времени или описания действий при активации) диаграммы последовательности ограничены отдельной диаграммой. Однако, если меток много, они могут затруднить чтение диаграммы. Если это так, текст некоторых меток может быть включен в проект потока событий.

При одновременном использовании как проекта потока событий, так и меток они дополняют друг друга.

Примеры проекта потока событий приводятся в подразделе «Описание взаимодействия объектов проектирования».

Требования к реализации

Требования к реализации — это текстовое описание, в котором собраны различные требования, например нефункциональные, к реализации варианта использования. Входящие в это описание требования были обнаружены только на стадии проектирования, и с ними было решено разобраться в процессе реализации. Некоторые из этих требований могли быть определены в ходе предыдущих рабочих процессов, но реализации вариантов использования изменили их. Другие, новые или порожденные, были впервые обнаружены в ходе проектирования.

Примеры требований к реализации в проекте реализации варианта использования приведены в подразделе «Определение требований к реализации».

Артефакт: Подсистема проектирования

Подсистемы проектирования предоставляют способ организации артефактов модели проектирования в виде частей, которыми легче управлять (рис. 9.8). Подсистема может содержать классы проектирования, реализации вариантов использования, интерфейсы и другие подсистемы (рекурсивно). Кроме того, подсистема может содержать интерфейсы, предоставляющие то, что в терминах операций называется экспортируемой функциональностью.

Подсистемы должны быть плотными, то есть их содержимое должно быть сильно связано. С другой стороны, между собой подсистемы должны быть связаны слабо, то есть их зависимость друг от друга или от любых других интерфейсов должна быть минимальна.

Кроме того, подсистемы проектирования должны иметь следующие характеристики.

- Подсистемы могут предоставлять разделение проектируемых сущностей. Например, в больших системах некоторые подсистемы могут проектироваться по отдельности, возможно, параллельно, разными командами разработчиков с различными навыками в проектировании.
- Два верхних уровня приложения и их подсистемы в модели проектирования часто имеют прямую трассировку к пакетам анализа и/или классам анализа.
- Подсистемы представляют собой компоненты «крупного помола» в реализации системы; так, компоненты, предоставляющие несколько интерфейсов, набираются из более мелких компонентов, которые представляют собой отдельные классы реализации. Эти компоненты реализуются в виде исполняемых файлов, двоичного кода или других сущностей подобного рода, загружаемых на различные узлы.
- Подсистемы могут использовать многократно применяемые программные продукты, являясь, фактически, оберткой для них. Подсистемы могут быть использованы для этого в целях интеграции многократно используемых программных

продуктов в модель проектирования. Эти подсистемы могут находиться на промежуточном уровне и уровне системного программного обеспечения.

- Подсистемы могут использовать унаследованные системы, являясь оберткой для них (или их частей). Подсистемы могут быть использованы для этого в целях интеграции унаследованных систем в модель проектирования.

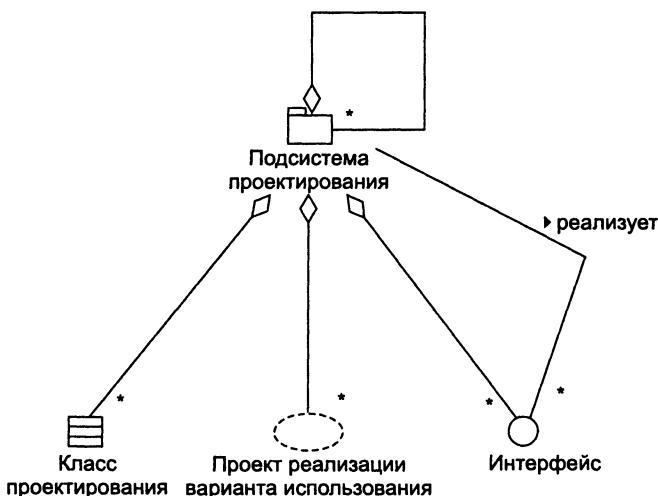


Рис. 9.8. Содержимое подсистемы и ключевые ассоциации

В подразделе «Определение подсистем и их интерфейсов» данной главы мы приведем дополнительные детали, касающиеся подсистем и их прагматики, включая способы идентификации их и их интерфейсов.

Сервисные подсистемы

Сервисные подсистемы используются в качестве нижнего уровня в иерархии подсистем проектирования по тем же причинам, по которым сервисные пакеты используются в модели анализа, а именно для того, чтобы подготовиться к изменениям в сервисах путем локализации изменений в соответствующих сервисных подсистемах. Что такое сервис, мы обсуждали в подразделе «Сервисные пакеты» главы 8.

Идентификация сервисных подсистем основывается на сервисных пакетах модели анализа. Обычно между ними существует однозначная (изоморфная) трасировка. Соответственно, сервисные подсистемы наиболее часто встречаются на двух верхних уровнях системы, а именно на общем уровне приложений и специфическом уровне приложений. Однако сервисные подсистемы требуют для функционирования большего количества элементов, чем соответствующие сервисные пакеты. Это происходит по следующим причинам.

- Сервисные подсистемы могут быть вынуждены предоставлять свои сервисы в терминах интерфейсов и их операций.
- Сервисные подсистемы содержат классы проектирования, а не анализа. Поэтому сервисные подсистемы вынуждены обеспечивать выполнение множества

нефункциональных требований и других ограничений, связанных со средой реализации. В результате сервисные подсистемы вынуждены содержать большее количество классов, чем соответствующие им сервисные пакеты. Им может потребоваться дальнейшая декомпозиция на (меньшие) подсистемы с целью сохранить размеры частей управляемыми.

- Сервисные подсистемы при реализации часто превращаются в двоичные файлы или исполняемые компоненты. Но в некоторых случаях сервисные подсистемы должны быть разделены на части для загрузки каждой из частей на отдельный узел. Для каждого узла в этом случае потребуется свой двоичный файл или исполняемый компонент. При этом может потребоваться декомпозиция сервисной системы на (меньшие) подсистемы, каждая из которых будет содержать функциональность, загружаемую на один из узлов.

Примеры сервисных подсистем приведены в подразделе «Определение прикладных подсистем».

Отметим, что основной способ организации артефактов в модели проектирования заключается в использовании стандартных подсистем проектирования, обсуждавшихся в предыдущем пункте. Однако мы вводим стереотип «сервисная подсистема», чтобы иметь возможность особо отметить подсистемы, предоставляющие сервисы. Возможность легко разделять различные типы подсистем особенно важна в случае больших подсистем (со множеством внутренних подсистем). Сравните это с соответствующими причинами введения стереотипа «сервисный пакет» в главе 8.

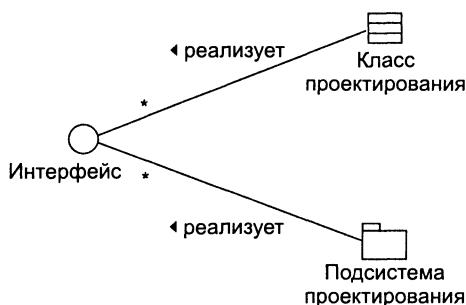


Рис. 9.9. Ключевые ассоциации интерфейса

Артефакт: Интерфейс

Интерфейсы используются для задания операций, выполняемых классом проектирования или подсистемой (рис. 9.9).

Класс проектирования, предоставляющий интерфейс, может также предоставлять методы реализации операций, заявленных в интерфейсе. Подсистема, предоставляющая интерфейс, также может содержать классы проектирования или другие подсистемы, непосредственно предоставляющие интерфейс.

Интерфейсы представляют способ отделения спецификации функциональности в терминах операций от ее реализации в терминах методов. Это отделение де-

ляет клиентов, которые зависят от интерфейса или используют его, независимыми от реализации интерфейса. Отдельная реализация интерфейса, такая как класс проектирования или подсистема, может быть заменена другой реализацией, при этом никаких изменений в клиенте делать не потребуется.

Большинство интерфейсов между подсистемами считается архитектурно значимыми, поскольку они определяют способы взаимодействия подсистем. В некоторых случаях они также используются для создания описаний стабильных интерфейсов в начале жизненного цикла системы до того, как в подсистеме будет реализована функциональность, которая в них объявлена. Эти интерфейсы могут рассматриваться как требования на разработку подсистем к команде разработчиков, а также могут быть использованы как инструмент синхронизации различных команд, одновременно разрабатывающих различные подсистемы.

В подразделе «Определение интерфейсов подсистем» данной главы мы приводим множество информации об интерфейсах и их прагматике, включая метод их идентификации.

Артефакт: Описание архитектуры (представление модели проектирования)

Описание архитектуры, содержащееся в **архитектурном представлении модели проектирования** (приложение B), содержит описание архитектурно значимых артефактов (рис. 9.10).

Архитектурно значимыми обычно считаются следующие артефакты модели проектирования.

- Декомпозиция модели проектирования на подсистемы, интерфейсы и зависимости между ними. Эта декомпозиция очень важна для архитектуры вообще, поскольку подсистемы и их интерфейсы образуют общую структуру системы.
- Ключевые классы проектирования, те, которые трассируются от архитектурно значимых классов анализа, активные классы¹ и классы проектирования, которые являются общими и основными, отражают общую концепцию проектирования и имеют множество связей с другими классами проектирования. Обычно достаточно считать архитектурно значимыми абстрактные классы, но не их дочерние классы. Исключением могут быть случаи, когда в подклассах описывается некоторое важное и архитектурно значимое поведение, отличное от поведения абстрактного класса.
- Проекты реализации варианта использования, описывающие некую важную и критичную функциональность, которые требуется разработать в начале жизненного цикла программы, содержат в себе множество классов разработки и поэтому имеют отношение к множеству элементов, включая, возможно несколько подсистем или ключевых классов проектирования, таких как описанные в предыдущем пункте. Обычным делом считается обнаружение соответствующих вариантов использования в архитектурном представлении модели вариантов использования и соответствующих анализов реализаций вариантов использования в архитектурном представлении модели анализа.

¹ Если активные классы входят в их собственную модель процесса (будет рассмотрена позже), они вместо этого могут быть описаны в архитектурном представлении модели этого процесса.

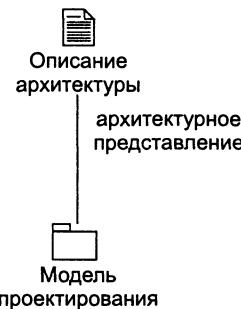


Рис. 9.10. Описание архитектуры, содержащее архитектурное представление модели проектирования

В подразделах «Определение подсистем и их интерфейсов», «Определение архитектурно значимых классов проектирования» и «Определение обобщенных механизмов проектирования» мы приведем примеры того, что может содержаться в архитектурном представлении модели проектирования.

Артефакт: Модель развертывания

Модель развертывания — это объектная модель, которая определяет физическое размещение системы, то есть то, каким образом функциональность системы распределена по вычислительным узлам (рис. 9.11). Модель развертывания является исходными данными для работы по проектированию и реализации. Распределение системы оказывает серьезнейшее влияние на ее проект.



Рис. 9.11. Модель развертывания включает в себя узлы

О модели развертывания следует сказать следующее.

- Каждый узел представляет собой вычислительный ресурс, обычно процессор или другое устройство.
- Узлы имеют связи, представляющие собой каналы обмена информацией, например *Интернет*, *инTRANET*, *внутренняя шина* и т. п.
- Модель развертывания описывает несколько различных конфигураций сети, включая конфигурации для тестирования и моделирования.

- Функциональность (или процесс), выполняющийся на узле, определяется компонентами, загруженными на узел.
- Именно модель развертывания описывает отображение архитектуры программы на архитектуру системы (оборудование).

В подразделе «Определение узлов и сетевых конфигураций» мы приведем пример работы с моделью развертывания.

Артефакт: Описание архитектуры (представление модели развертывания)

Описание архитектуры содержит **архитектурное представление модели развертывания** (приложение В) и описывает этот архитектурно значимый артефакт (рис. 9.12).

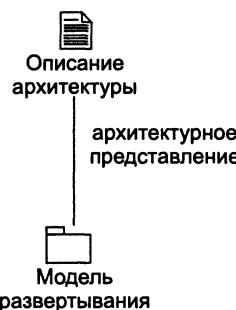


Рис. 9.12. Описание архитектуры содержит архитектурное представление модели развертывания

Все аспекты модели развертывания, включая отображение компонентов на узлы в соответствии с реализацией, по причине их важности должны быть описаны в архитектурном представлении. В подразделе «Определение узлов и сетевых конфигураций» мы приведем примеры того, что может включаться в архитектурное представление модели развертывания.

Сотрудники

Сотрудник: Архитектор

В проектировании архитектор отвечает за целостность моделей проектирования и развертывания и гарантирует корректность, согласованность и читаемость моделей в целом (рис. 9.13). Как и в модели анализа, для больших и сложных систем эти обязанности для подсистемы верхнего уровня (то есть для системы проектирования) могут возлагаться на нескольких сотрудников.

Модели корректны, если они реализуют функциональность, причем только ту функциональность, которая описана в модели вариантов использования, дополнительных требованиях и модели анализа.

Архитектор также отвечает за архитектуру моделей проектирования и развертывания, то есть за создание этих архитектурно значимых частей проекта в соответствии с данными, которые содержатся в архитектурных представлениях моделей. Напомним, что эти представления являются частями архитектурного описания системы.

Отметим, что архитектор не отвечает за последовательную разработку и обслуживание различных артефактов модели проектирования. Это зоны ответственности соответствующих инженеров по вариантам использования и инженеров по компонентам (см. подразделы «Сотрудник: Инженер по вариантам использования» и «Сотрудник: Инженер по компонентам»).

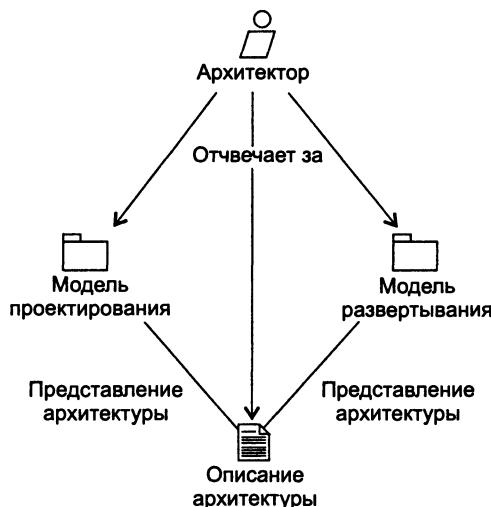


Рис. 9.13. Сфера ответственности архитектора в проектировании



Рис. 9.14. Сфера ответственности инженера по вариантам использования в проектировании

Сотрудник: Инженер по вариантам использования

Инженер по вариантам использования отвечает за целостность одной или нескольких реализаций варианта использования и гарантирует выполнение предъявляемых к ним требований (рис. 9.14). Проект реализации варианта использования должен корректно описывать поведение соответствующего анализа реализации варианта использования из модели анализа, и только его, так же как и поведение соответствующих вариантов использования из модели вариантов использования.

В эту работу входит создание всей текстовой информации и диаграмм, дающих читаемое и пригодное для работы описание реализации варианта использования.

Отметим, что инженер по вариантам использования не отвечает за создание классов проектирования, подсистем, интерфейсов и зависимостей, используемых в реализациях вариантов использования. Это работа соответствующего инженера по компонентам.

Сотрудник: Инженер по компонентам

Инженер по компонентам определяет и поддерживает операции, методы, атрибуты, отношения и требования к реализации одного или нескольких классов проектирования и гарантирует выполнение предъявляемых к каждому из классов проектирования требований в ходе осуществления реализации варианта использования, в которой участвует этот класс (рис. 9.15).

Инженер по компонентам также может поддерживать целостность одной или более подсистем. Под этим понимается обеспечение правильности содержимого (то есть классов и связей между ними), а также того, что зависимость подсистем от других подсистем и/или интерфейсов верна и минимальна, и подсистемы правильно реализуют предоставляемые интерфейсы.

Часто считается правильным, чтобы инженер по компонентам, отвечающий за подсистему, отвечал также и за ее внутренние элементы. Более того, для осуществления гладкой, неразрывной разработки будет естественно, если артефакты модели проектирования (классы проектирования и подсистемы) будут переноситься в рабочий процесс реализации и реализовываться тем же самым инженером по компонентам.

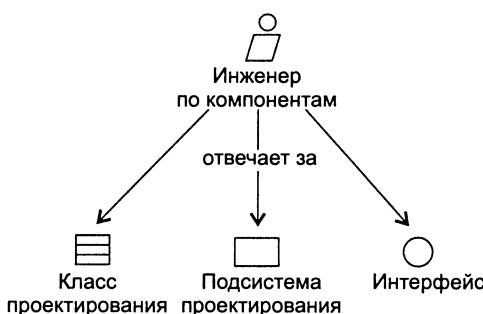


Рис. 9.15. Сфера ответственности инженера по компонентам в проектировании

Рабочий процесс

Ранее в этой главе мы рассматривали работу по проектированию в статике. Теперь используем диаграмму деятельности для изучения ее динамического поведения (рис. 9.16).

Создание моделей проектирования и развертывания (описанное ранее в этой главе) инициируется архитектором, делающим краткое описание узлов модели развертывания, основных подсистем и их интерфейсов, важнейших классов проектирования, включая активные, и обобщенных механизмов проектирования модели проектирования. Затем инженеры по вариантам использования описывают все варианты использования в терминах взаимодействующих классов проектирования и/или подсистем и их интерфейсов. Получившиеся реализации вариантов использования содержат требования к поведению каждого из классов или подсистем, участвующих в реализации варианта использования. Эти требования затем определяются инженерами по компонентам и встраиваются в классы — путем создания соответствующих операций, атрибутов и зависимостей в классах или соответствующих операций в интерфейсах, которые предоставляет подсистема. В ходе рабочего процесса проектирования по мере того, как развивается модель проектирования, разработчики находят кандидатов на использование в качестве новых подсистем, интерфейсов, классов и обобщенных механизмов проектирования. Инженеры по компонентам, отвечающие за отдельные подсистемы, уточняют и поддерживают эти подсистемы.

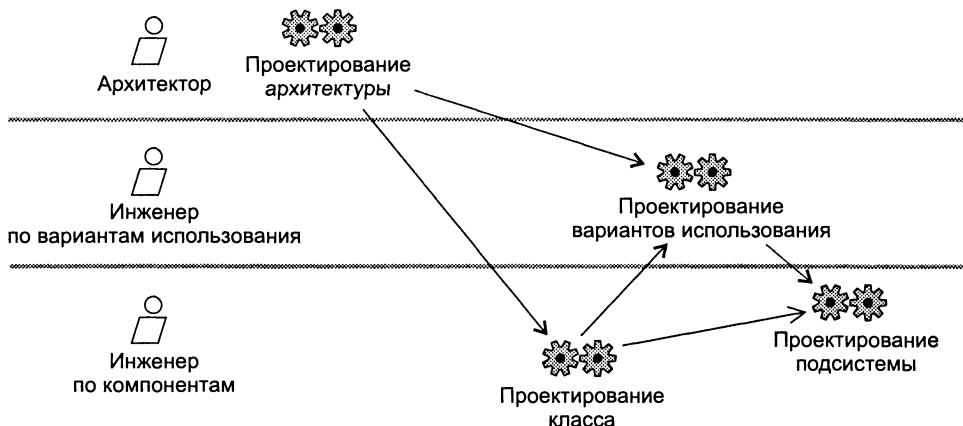


Рис. 9.16. Рабочий процесс проектирования, включая участвующих сотрудников и их деятельность

Деятельность: Проектирование архитектуры

Цель проектирования архитектуры — описание моделей проектирования и развертывания и их архитектуры для решения следующих проблем (рис. 9.17):

- узлы и их сетевые конфигурации;

- подсистемы и их интерфейсы;
- архитектурно значимые классы, включая активные классы;
- обобщенные механизмы проектирования, отрабатывающие общие требования, такие как специальные требования по длительному хранению, распространению, производительности и т. д., обнаруженные в ходе анализа в классах анализа и анализах реализаций вариантов использования.

В ходе этой деятельности архитектор определяет возможность повторного использования различных модулей, таких как многократно используемые части похожих систем или стандартные программные продукты. Обнаруженные в результате подсистемы, интерфейсы или другие элементы проекта затем включаются в модель проектирования. Также архитектор поддерживает, уточняет и обновляет описание архитектуры и входящие в него архитектурные представления моделей проектирования и развертывания.

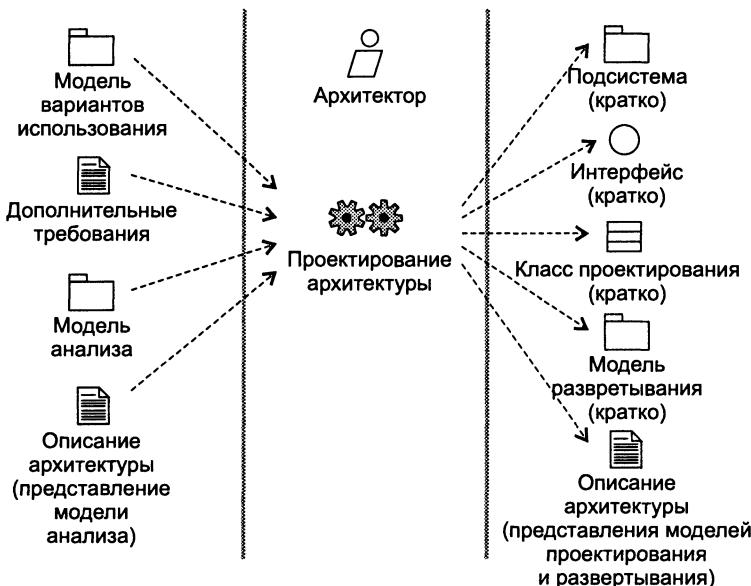


Рис. 9.17. Исходные данные и результат проектирования архитектуры

Определение узлов и сетевых конфигураций

Физическая конфигурация сети часто вносит важный вклад в архитектуру программы, включая то, какие активные классы потребуются и как будет распределена функциональность по узлам сети. Традиционная конфигурация сети предполагает использование образца трехуровневой системы, когда один уровень занимает клиент (взаимодействие с пользователем), один уровень — функциональность, связанная с базой данных, и один уровень — прикладная функциональность. Простой образец клиент-серверной архитектуры представляет собой специальный вариант образца трехуровневой системы, в котором прикладная логика перенесена в один из двух оставшихся уровней (то есть уровень базы данных или клиента).

Вопросы конфигурации сети.

- Какие существуют узлы и каковы их возможности в понятиях вычислительной мощности и размера памяти?
- Как узлы соединены между собой и какой коммуникационный протокол используется для их связи?
- Каковы характеристики связей и коммуникационных протоколов, например скорость, доступность и качество?
- Существует ли необходимость в наличии процедур защиты от ошибок, перемещения процессов, создания резервных копий данных и т. п.?

Определив пределы возможностей узлов и их связей, архитектор может включить в проект такие технологии, как брокера объектных запросов или системы репликации данных, которые сделают реализацию распределения системы проще.

Пример. *Сетевая конфигурация системы Interbank.* Система Interbank будет работать на трех серверных узлах и множестве узлов-клиентов. Прежде всего, один сервер будет предназначен для продавцов и один – для покупателей. Причина этого в том, что организация-продавец и организация-покупатель нуждаются в центральных серверах для работы со своими бизнес-объектами. Конечные пользователи получают доступ к этой системе через клиентские узлы, такие как *Клиент покупатель*. Узлы связаны через Интернет и Интранет при помощи протокола TCP/IP (рис. 9.18).

Кроме того, существует и третий сервер непосредственно в банке, то есть там, где, собственно, и происходит реальная оплата счетов (деньги перечисляются со счета на счет).



Рис. 9.18. Диаграмма развертывания системы Interbank

Все конфигурации сети, включая специальные конфигурации, предназначенные для тестирования и моделирования, должны быть описаны отдельными диаграммами развертывания. Получив эти конфигурации, можно начинать обсуждение того, как распределить между ними функциональность системы (см. подраздел «Определение активных классов»).

Определение подсистем и их интерфейсов

Подсистемы представляют собой метод организации модели проектирования в виде набора обозримых частей. Они могут быть выделены в начале проектирования при разделении работы по проектированию на части или найдены после того, как модель проектирования разрастется в большую, требующую декомпозиции структуру.

Заметим также, что не все подсистемы разрабатываются с нуля в рамках текущего проекта. Некоторые из них представляют собой повторно используемые продукты или другие готовые единицы. Наличие таких подсистем в модели проектирования делает возможным обсуждение и оценку возможности повторного использования компонентов.

Определение прикладных подсистем

На этом шаге мы определяем подсистемы общего и специфического уровней приложений (то есть двух верхних уровней), рис. 9.19.

Если во время анализа была проведена декомпозиция на пакеты анализа, мы можем, используя эти пакеты, определять соответствующие подсистемы в модели проектирования. Это особенно важно для сервисных пакетов, на основе которых мы определим соответствующие им сервисные подсистемы, не нарушающие структурирования системы в соответствии с предоставляемыми ею сервисами. Однако это первичное определение подсистем в ходе проектирования будет слегка уточнено в вопросах, касающихся проектирования, реализации и развертывания системы. Рассмотрим это в общих чертах.

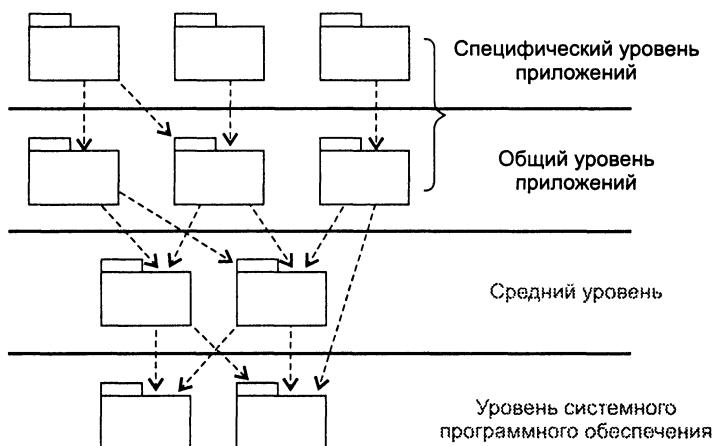


Рис. 9.19. Подсистемы общего и специфического уровней приложения

Пример. Определение подсистем проектирования на базе пакетов анализа. Пакеты Управление счетами покупателя и Управление банковскими счетами из модели анализа используются для определения соответствующих подсистем модели проектирования (рис. 9.20).

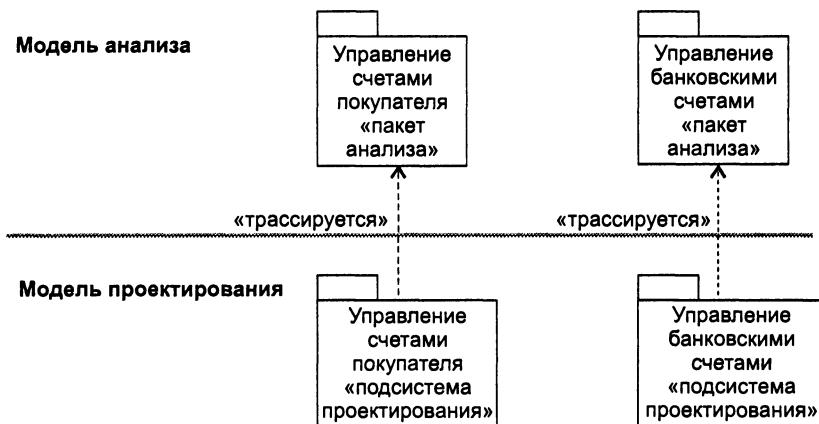


Рис. 9.20. Определение подсистем на основе существующих пакетов анализа

Более того, сервисные пакеты *Банковские счета* и *Риски*, содержащиеся в пакете *Управление банковскими счетами* модели анализа, могут использоваться для определения соответствующих сервисных подсистем модели проектирования (рис. 9.21).

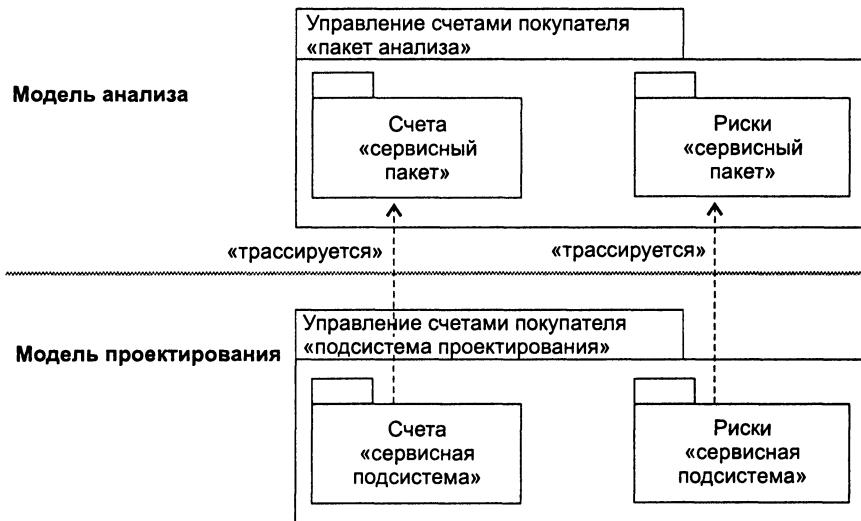


Рис. 9.21. Определение сервисных подсистем на основе существующих сервисных пакетов

Уточнение исходной декомпозиции подсистем по сравнению с (предыдущими) пакетами анализа может потребоваться, когда часть (предыдущего) пакета анализа проецируется на подсистему целиком (то есть если обнаружено, что эта часть может совместно использоваться несколькими подсистемами).

Пример. Уточнение подсистем для работы с совместно используемой функциональностью. В Interbank Software, проанализировав варианты использования, связанные со счетами, решили ввести все сервисные пакеты для оплаты счетов в подсистему *Оплата счетов покупателя*. Затем, когда разработчики обнаружили, что некоторые дополнительные варианты использования улучшатся, если использовать общую подсистему платежного сервиса, они решили свести всю функциональность платежей в одну сервисную подсистему под названием *Управление планированием платежей*. Это означает, что подсистема *Управление счетами покупателя* будет использовать функциональность планирования платежей, взятую из сервисной подсистемы *Управление планированием платежей*. Впоследствии, когда наступит время запланированного платежа, подсистема *Управление планированием платежей* использует подсистему *Управление банковскими счетами* для перечисления денег с одного банковского счета на другой (рис. 9.22).



Рис. 9.22. Сервисная подсистема Управление планированием платежей предоставляет общий сервис, который может использоваться несколькими реализациями вариантов использования

Некоторые части (предыдущих) пакетов анализа реализованы при помощи многократно используемых программных продуктов. Некоторая часть функциональности может быть затем вынесена в подсистемы среднего уровня или уровня системного программного обеспечения (см. подраздел «Определение подсистем среднего уровня и уровня системного программного обеспечения»).

(Предыдущие) пакеты анализа не отражают разделения работ.

(Предыдущие) пакеты анализа не отражают использования унаследованных систем. Унаследованная система или ее часть могут быть вынесены в отдельную подсистему.

(Предыдущие) пакеты анализа не подготовлены к непосредственному развертыванию на узлах. Для того чтобы осуществить развертывание, потребуется декомпозиция подсистем, при этом часть подсистем придется разбить на меньшие подсистемы, каждая из которых будет загружаться на отдельный узел. Затем эти (небольшие) подсистемы следует уточнить с целью минимизации сетевого трафика и т. п.

Пример. Распределение системы по узлам. Для развертывания подсистемы Управление счетами покупателя она должна быть разделена между несколькими различными узлами. Мы разделим подсистему на три части — подсистемы: *Интерфейс*

покупателя, Управление запросами на оплату и Управление счетами (рис. 9.23). Компоненты, созданные для реализации каждой из этих трех меньших подсистем, будут загружены на узлы *Клиент покупатель*, *Сервер покупателя* и *Сервер продавца* соответственно.

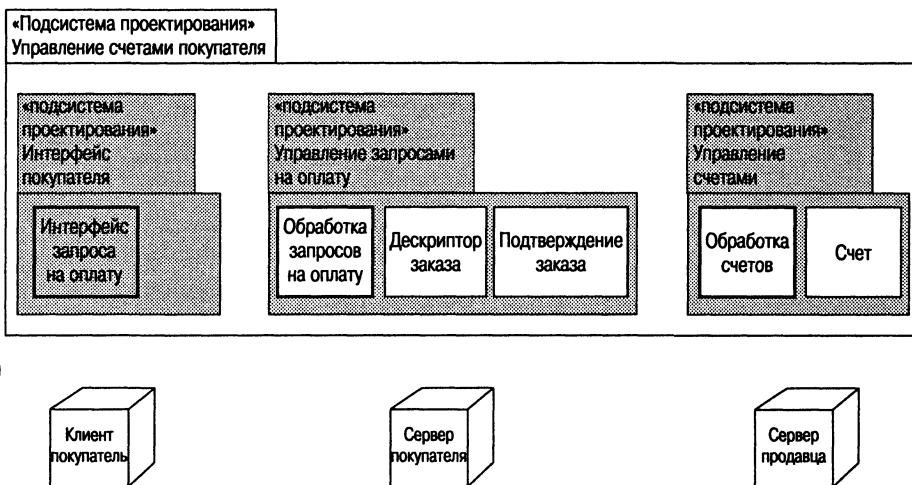


Рис. 9.23. Подсистема для осуществления загрузки разбивается на три подсистемы

Определение подсистем среднего уровня и уровня системного программного обеспечения

Системное программное обеспечение и программное обеспечение среднего уровня представляет собой основание системы, поскольку вся функциональность системы опирается на такие программы, как операционные системы, СУБД, коммуникационные программы, технологии распределения продуктов, комплекты для разработки интерфейсов и технологии управления транзакциями [50] (рис. 9.24). Подбор и интеграция покупаемых или создаваемых программных продуктов — это две главных задачи фаз анализа и определения требований и проектирования. Архитектор проверяет пригодность выбираемых программных продуктов для принятой архитектуры и выгодность их применения в разрабатываемой системе.

Внимание! Когда приобретается системное программное обеспечение или программное обеспечение среднего уровня, то контроль за его развитием очень слаб или вообще не ведется. Важно поддерживать определенную свободу действий и не стать загнанным в угол полной зависимостью от каких-либо продуктов или поставщиков, слабо заинтересованных в проекте. Страйтесь ограничивать зависимость от приобретенных продуктов, а, следовательно, и риск, связанный с их применением, который заключается в возможных изменениях этого продукта в будущем или вероятностью возникновения необходимости сменить поставщика.

Одним из способов контролировать эту зависимость может быть обращение с каждым из закупаемых продуктов как с отдельной подсистемой, с явно заданными интерфейсами между ним и остальной системой. Например, если вам нужно

встроить в систему механизм скрытого распределения объектов, это можно сделать путем определения четкого интерфейса подсистемы, которая реализует этот механизм. Это оставляет за вами свободу выбора из всей гаммы продуктов, при этом стоимость изменения системы ограничена.

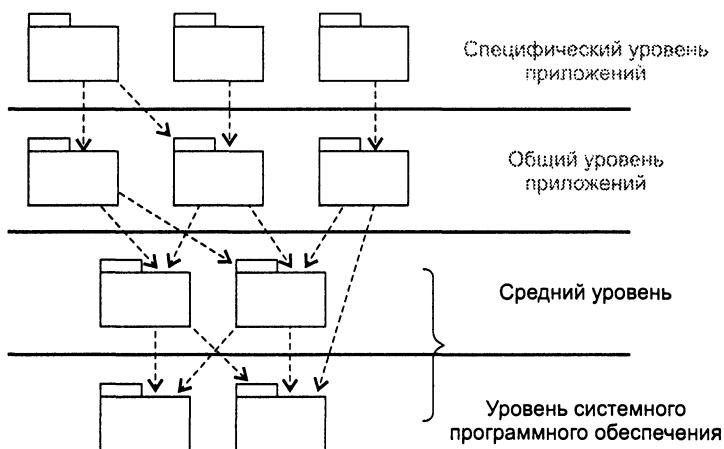


Рис. 9.24. Подсистемы уровня системного программного обеспечения и программного обеспечения среднего уровня содержат приобретаемые программные продукты

Пример. Использование Java при построении программного обеспечения среднего уровня. Некоторые из реализаций прикладных подсистем, разработанных в Interbank Software, должны иметь возможность работать на машинах разных типов, например PC-совместимых компьютерах и рабочих станциях на базе UNIX, а следовательно, должны быть переносимыми с одной платформы на другую. В Interbank Software решили реализовать эту переносимость путем использования программного обеспечения среднего уровня, в данном случае — Java-пакетов Abstract Windowing Toolkit (AWT), Remote Message Invocation (RMI) и апплетов. Эти Java-пакеты изображены на рис. 9.25 в виде подсистем. Работа этих подсистем основана на виртуальной Java-машине, которая представляет собой интерпретатор языка Java. Для того чтобы компьютер мог исполнять код, написанный на Java, на нем должна быть установлена эта виртуальная машина. В наших примерах для загрузки web-страниц и запуска апплетов используется web-браузер.

На самом низком уровне Interbank Software опирается на такое системное программное обеспечение, как средства реализации протокола TCP/IP для Интернета (рис. 9.25).

Подсистема Java-апплетов — это программное обеспечение среднего уровня, которое позволяет Interbank создавать Java-апплеты.

Каждое окно интерфейса пользователя разрабатывается с использованием класса *Java Applet* и других классов пользовательского интерфейса, входящих в подсистему AWT, таких как *List*. Пакет Java Abstract Windowing Toolkit (*java.awt*) создавался для поддержки платформонезависимого пользовательского интерфейса. В него включены, например, такие классы, как *Field*, *Scrollbar* и *CheckBox*.

Java RMI – это механизм распространения объектов, встроенный в стандартную библиотеку классов Java. Мы выбираем RMI в качестве средства для распространения объектов, чтобы сделать наш пример проще и избежать смешения различных техник и языков программирования. Также могут быть использованы решения на базе CORBA или ActiveX/DCOM.

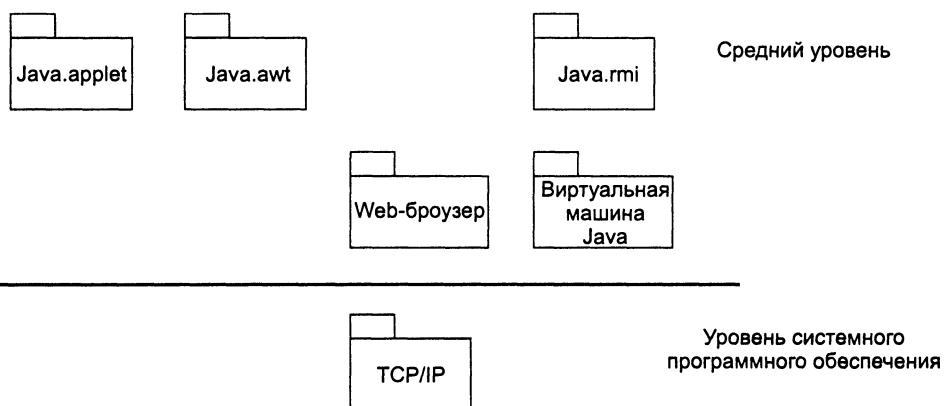


Рис. 9.25. Средний уровень, построенный на базе Java, предоставляет платформонезависимый графический интерфейс пользователя (java.awt) и обработку распределенных объектов (java.rmi)

Рисунок 9.25 показывает, каким образом сервисы Java могут быть организованы в подсистемы среднего уровня. Каждая подсистема содержит классы, разработанные для совместной работы с другими с целью предоставления соответствующего сервиса (на рисунке не отражены зависимости между системами, они представлены на рис. 9.26). Также в виде подсистем будут представлены и компоненты Active X, такие, как электронные таблицы, мультимедиа, обработчики изображений, пакеты защиты, долговременного хранения, распределения, логического вывода, поддержки процессов и параллельной работы. Часто бывает полезно создать для стандартных типов данных или базовых классов отдельные подсистемы, которые будут импортироваться и использоваться всеми прочими подсистемами (например, java.lang, которая содержит множество базовых классов, таких как Boolean, Integer, Float).

Определение зависимостей между подсистемами

В том случае, когда внутреннее содержание подсистем связано, между ними определяются зависимости. Направление зависимостей будет совпадать с направлением отношений (навигации по отношениям). Если зависимости необходимо описать до того, как станет известно содержание подсистем, мы пользуемся зависимостями между теми пакетами анализа, которые соответствуют рассматриваемым подсистемам проектирования. Зависимости между ними соответствуют зависимостям модели проектирования. Если используются интерфейсы между подсистемами, будут существовать зависимости между интерфейсами, а не между собственно подсистемами (см. следующий подраздел).

Пример. Зависимости и уровни. На рисунке 9.26 изображены подсистемы системы Interbank и некоторые из (первичных) зависимостей между ними.

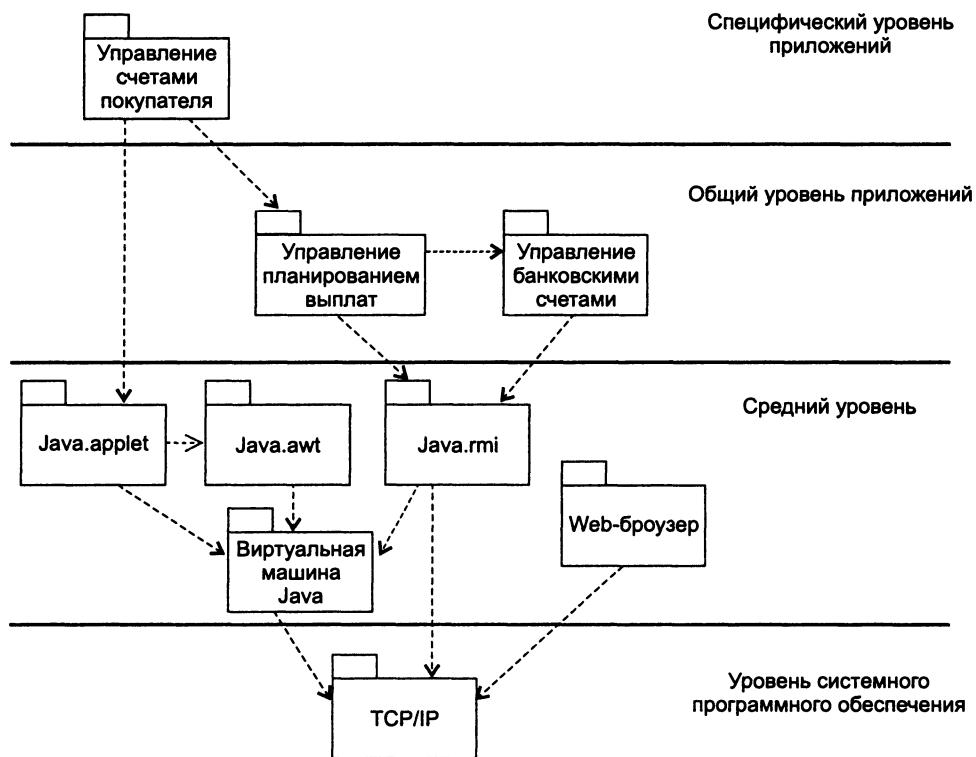


Рис. 9.26. Зависимости и уровни некоторых подсистем системы Interbank

Отметим, что некоторые из этих зависимостей, особенно на среднем уровне и уровне системного программного обеспечения, могут в той или иной степени подразумеваться средой реализации (в данном случае Java). Однако, если зависимости вносят вклад в архитектуру системы и особенно если они соединяют разные уровни, их следует включить в модель проектирования и отобразить на диаграмме классов подобно тому, как это изображено на рис. 9.26. Эта диаграмма затем должна быть передана инженерам по вариантам использования и инженерам по компонентам, которые будут строить варианты использования, классы и подсистемы.

Определение интерфейсов подсистем

Интерфейсы, предоставляемые подсистемами, определяют операции, доступные «извне» подсистемы. Эти интерфейсы представляются классами или другими подсистемами (рекурсивно), содержащимися в данной подсистеме.

Для первичного описания интерфейсов до того, как станет известно внутреннее содержание подсистемы, мы начинаем с определения зависимостей между подсистемами, как было описано в предыдущем подразделе. Если подсистема имеет направленную на нее зависимость, это скорее всего означает, что она должна пред-

ставлять интерфейс. Кроме того, если существует пакет анализа, трассирующийся к подсистеме, то любой из классов анализа, к которому обращаются извне этого пакета, как показано в следующем примере, может рассматриваться в качестве кандидата на поставщика интерфейса подсистемы.

Пример. Поиск кандидатов для интерфейсов на основе модели анализа. Сервисный пакет *Банковские счета* содержит класс анализа под названием *Перечисление денег*, обращающийся к сущностям вне пакета. Поэтому мы можем определить базовый интерфейс, который назовем *Перечисления*, предоставляемый соответствующей сервисной подсистемой *Банковские счета* модели проектирования (рис. 9.27).

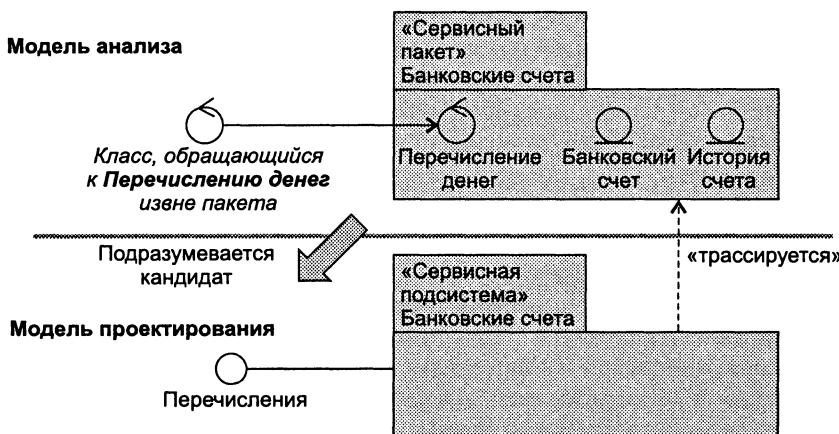


Рис. 9.27. Первичное определение интерфейса на основе модели анализа

Используя этот подход, мы сперва определяем интерфейсы двух верхних уровней модели проектирования, как показано на рис. 9.28.

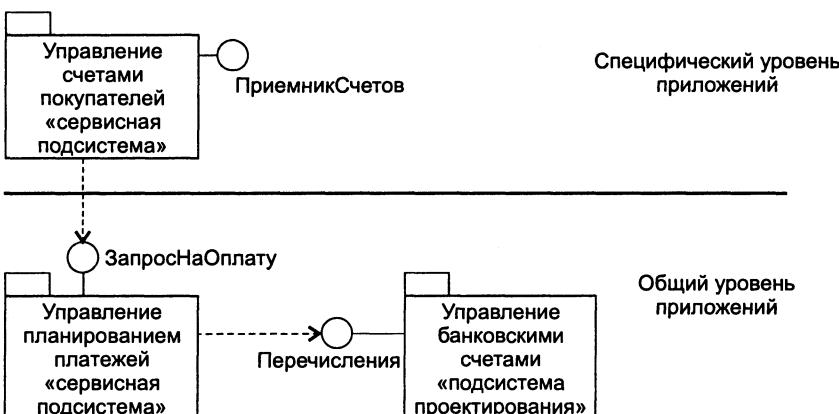


Рис. 9.28. Интерфейсы двух верхних уровней модели проектирования

Подсистема *Управление банковскими счетами* предоставляет интерфейс *Перечисления* для перечисления денег со счета на счет. Это тот же интерфейс, который предоставляется сервисной подсистемой *Банковские счета* из *Управления банковскими счетами* (см. предыдущий пример). Подсистема *Управление планированием платежей* предоставляет интерфейс *Запрос На Оплату*, используемый для планирования платежей. Подсистема *Управление счетами покупателей* предоставляет интерфейс *Приемник Счетов* для получения от продавца новых счетов. Этот интерфейс используется в варианте использования *Выставить счет покупателю*, когда покупателю посыпается новый счет.

Отметим, что описанные здесь интерфейсы заставляют нас уточнить зависимости между подсистемами, как это рассматривалось в подразделе «Определение зависимостей между подсистемами», чтобы решить, представлять эти зависимости интерфейсами или подсистемами.

Что касается интерфейсов двух нижних уровней (то есть среднего уровня и уровня системного программного обеспечения), задача идентификации интерфейсов для них проще, поскольку подсистемы этих уровней скрывают в себе программные продукты, которые часто уже имеют какие-то предопределенные интерфейсы.

Однако для определения интерфейсов этого недостаточно, и мы сами должны определить операции, в которых нуждается каждый из определяемых интерфейсов. Это все, что нужно для проектирования вариантов использования в понятиях подсистем и их интерфейсов в ходе деятельности по проектированию вариантов использования, описанной в подразделах «Определение участвующих подсистем и интерфейсов» и «Описание взаимодействия подсистем». В ходе этой работы устанавливаются требования к операциям, которые должны быть определены в интерфейсе. Затем требования различных вариантов использования обрабатываются и собираются вместе для каждого интерфейса, как описано в подразделе «Сохранение предоставляемых подсистемой интерфейсов».

Определение архитектурно значимых классов проектирования

Часто бывает полезно определить архитектурно значимые классы проектирования в начале жизненного цикла системы, чтобы инициировать работу по проектированию. Однако многие классы проектирования могут быть определены только в ходе деятельности по определению классов проектирования и уточнены на основе результатов деятельности по проектированию вариантов использования (см. подразделы «Деятельность: Проектирование вариантов использования» и «Деятельность: Проектирование класса»). По этой причине разработчики должны быть осторожны. На этой стадии им не следует пытаться определить много классов или проработать слишком много деталей. Исходного наброска архитектурно значимых классов будет вполне достаточно (см. подраздел «Артефакт: Описание архитектуры (представление модели проектирования)»). В противном случае, не уточнив состав классов проектирования при помощи вариантов использования (то есть не уточнив, участвует ли класс в реализации вариантов использования), мы рискуем выполнить очень много лишней работы. Классы проектирования, не участвующие в реализации варианта использования, не являются необходимыми.

Определение классов проектирования на основе классов анализа

Некоторые классы проектирования могут быть изначально описаны по архитектурно значимым классам анализа, найденным в ходе анализа. Таким же образом отношения между этими классами анализа могут быть использованы для получения пробного набора отношений между соответствующими классами проектирования.

Пример. Описание классов проектирования по классам анализа. Класс проектирования *Счет* первоначально описывается по классу сущности *Счет* модели анализа (рис. 9.29).

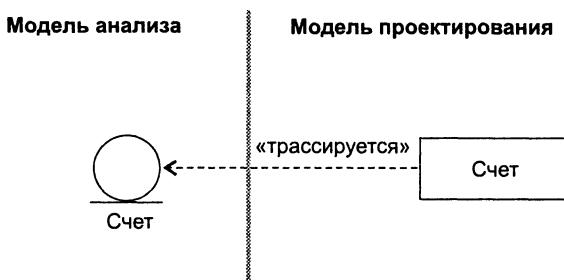


Рис. 9.29. Класс проектирования Счет первоначально описывается по классу сущности Счет

Определение активных классов

Архитектор также определяет активные классы, необходимые системе для выполнения требований, касающихся параллельной работы, например:

- Требования к производительности, пропускной способности и доступности для различных актантов, взаимодействующих с системой. Если, например, один из актантов имеет повышенные требования к времени отклика, то он должен обслуживаться специально предназначенным для этого объектом, который будет принимать от актанта исходные данные и пересыпать ему результаты работы. Работа объекта не должна прерываться только потому, что был загружен другой активный объект (которому для работы требуется процессорная мощность или память).
- Распределение системы по узлам. Активные объекты должны поддерживать распределение на несколько различных узлов, что, например, требует как минимум одного активного объекта на узел и отдельного активного объекта — для обмена между узлами.
- Другие требования, например, требования к запуску и остановке системы, восстановлению, снятию взаимных блокировок, снятию зависаний, реконфигурации узлов и пропускной способности соединений.

Каждый активный класс описывается с учетом жизненного цикла его активного объекта и того, как этот активный объект обменивается данными, синхронизируется и разделяет используемую информацию. Затем проводится привязка активного объекта к узлам модели развертывания. В ходе привязки активного объекта к узлам необходимо учитывать возможности узлов, например тип процессора и раз-

мер памяти, и характеристики соединения между ними, такие как скорость связи и доступность. Основное правило, которым следует при этом пользоваться, состоит в том, что сетевой трафик обычно сильно влияет на размеры и тип вычислительных ресурсов (включая как программы, так и аппаратуру), которые необходимы системе, и потому должен постоянно находится под контролем. Это может внести основной вклад в модель проектирования.

Для первичного описания активных классов можно использовать в качестве исходных данных результаты анализа и модель развертывания, а затем спроектировать соответствующие проекты классов анализа (или их части) на узлы посредством активных классов.

Пример. Использование классов анализа для описания активных классов. Как мы решили, система Interbank должна быть распределена по таким узлам, как *Клиент покупатель*, *Сервер покупателя*, *Сервер банка* и т. д. Кроме того, мы уже определили классы анализа — *Интерфейс запроса на оплату*, *Дескриптор заказа*, *Подтверждение заказа*, *Счет* и т. д. (рис. 9.30). Итак, мы обнаружили, что покупатель заинтересован в функциональности, предоставляемой классами анализа *Подтверждение заказа* и *Дескриптор заказа*, но эта функциональность требует от узла *Покупатель клиент* значительной вычислительной мощности. Вместо этого основные части этих классов анализа следует разместить на *Сервере покупателя*, и с этого узла управлять отдельными активными классами (*Управление запросами на оплату*).

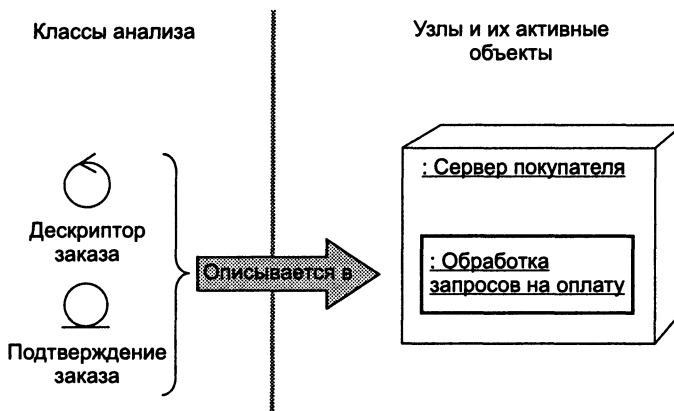


Рис. 9.30. Активный класс Обработка запросов на оплату, который будет содержать нить управления классов Дескриптор заказа и Подтверждение заказа на узле Сервер покупателя

Есть и другой способ описать активные классы. Для этого следует использовать ранее определенные подсистемы и привязать к отдельным узлам подсистемы целиком. Тогда активные классы будут определены на базе целых подсистем (см. подраздел «Определение прикладных подсистем»). Напомним, что для того, чтобы это было возможно, может понадобиться декомпозиция подсистем.

Каждый из активных классов, представляющих полноценные процессы, является кандидатом на превращение в исполняемый файл. Это определяется окончательно в ходе реализации. Так, распределение активных классов по узлам, сделан-

ное на этом этапе, является важными исходными данными для распределения по узлам (исполняемых) компонентов, которое проводится в ходе реализации. Кроме того, когда к узлу привязывается подсистема целиком, все составные части подсистемы обычно составляют единый (исполнимый) компонент, также привязанный к соответствующему узлу.

Определение обобщенных механизмов проектирования

На этом этапе мы изучим общие требования к анализам реализации вариантов использования и классам анализа, подобные специальным требованиям, определенным во время анализа. Мы обсудим способы их разрешения при помощи имеющихся в нашем распоряжении технологий проектирования и реализации. Результатом этой работы будет набор обобщенных механизмов проектирования, при помощи которых мы представим эти требования в виде классов проектирования, коопераций или даже подсистем, подобных описанным в [23].

Требования, которые нам предстоит обработать, часто связаны со следующими проблемами:

- длительное хранение;
- скрытое распределение объектов;
- средства безопасности;
- контроль и исправление ошибок;
- управление транзакциями.

В некоторых случаях механизмы не лежат на поверхности, и их приходится искать, исследуя реализации вариантов использования и классы проектирования.

Пример. *Механизм проектирования скрытого распределения объектов.* Некоторые объекты, такие, как *Счет*, должны быть доступны на нескольких узлах. Поэтому их следует проектировать для работы в распределенной среде. В Interbank Software решили реализовать распределение объектов, сделав каждый распределенный класс наследником абстрактного класса Java, `java.rmi.UnicastRemoteObject`, который поддерживает Remote Message Invocation (RMI), рис. 9.31. RMI – это техника, используемая в Java для организации скрытого распределения объектов (когда объекты распределяются таким образом, что клиентский объект ничего не знает об их истинном местоположении).



Рис. 9.31. Любой класс, который требует распределенной обработки, должен быть дочерним классом `UnicastRemoteObject`

Пример. Механизм проектирования для длительного хранения. Некоторые объекты, например Счет или Заказ, должны сохраняться длительное время. Чтобы достичь этого, можно использовать систему управления объектной базой данных, систему управления реляционной базой данных или просто двоичный файл. Что лучше, определяется требованиями к доступу и изменению объектов, а также тем, что будет легче реализовать и развивать в дальнейшем. Реляционная база данных обычно дает наилучшую производительность на табличных данных, в то время как объектная база данных больше подходит для объектов сложной структуры. Системы управления реляционными базами данных также более проработаны, чем системы управления объектными базами. Процесс разработки системы на основе системы управления реляционными базами данных с последующей заменой ее на объектную базу данных (когда они будут вполне проработаны) обойдется недешево. Какое бы решение не было выбрано, архитектор документирует его как обобщенный механизм проектирования для решения вопросов длительного хранения данных.

Последний пример показывает, что любой механизм обычно имеет несколько путей реализации, и у каждого из этих путей есть свои «за» и «против». Если невозможно предложить один подходящий во всех случаях механизм, следует найти несколько механизмов и в каждой ситуации использовать наилучший.

Архитектор также определяет обобщенные кооперации, работающие как образцы, которые можно неоднократно использовать в реализациях вариантов использования модели проектирования.

Пример. Обобщенная кооперация между отдельными реализациями вариантов использования. В ходе работы с вариантами использования и их реализациями архитекторы определяют образцы, согласно которым Торговый объект, такой, как Счет или Заказ, создается одним актантом и пересыпается другому.

- Когда Покупатель решает заказать какие-то товары или услуги у продавца, покупатель активирует вариант использования Заказать товары или услуги. Вариант использования позволяет покупателю определить заказ и послать его продавцу по сетям связи.
- Когда Продавец решает послать счет покупателю, продавец активирует вариант использования Послать счет покупателю, который посыпает счет покупателю по сетям связи.

Это общий тип поведения, который может быть описан обобщенной кооперацией, как показано на диаграмме коопераций на рис. 9.32.

Сначала Интерфейс Создания Торгового объекта получает информацию от актанта Посьлающий. Эта информация используется в качестве исходных данных для создания Торгового объекта (1) (числа в скобках относятся к рис. 9.32). Затем Интерфейс создания торгового объекта просит Обработку отправителя создать Торговый объект (2). Обработка отправителя запрашивает у соответствующего класса Торговый объект создание его экземпляра (3). Обработка отправителя передает ссылку на Торговый объект Обработке получателя (4). Обработка получателя может затем запросить у Торгового объекта дополнительную информацию о нем (5). После этого Обработка получателя передает Торговый объект для представления Интерфейсу представления торгового объекта (6), который представляет Торговый объект актанту Получатель (7).

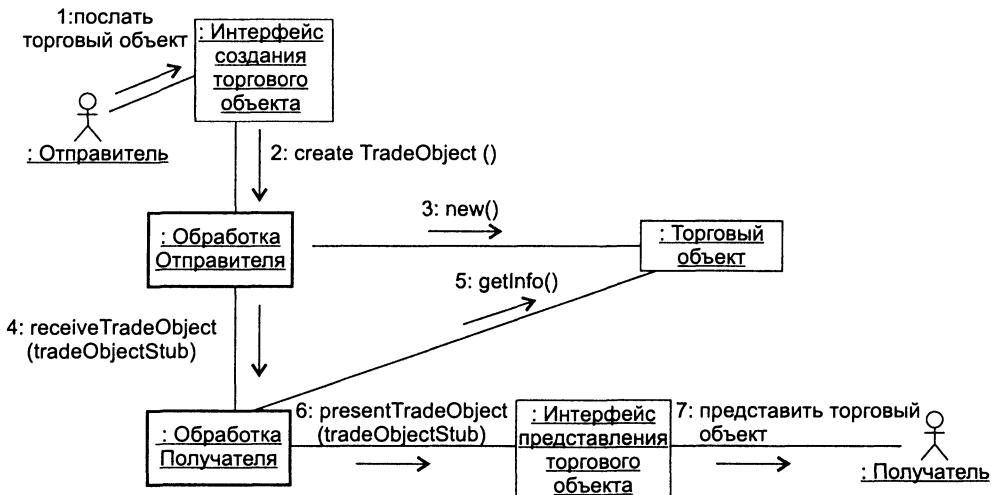


Рис. 9.32. Диаграмма кооперации дает пример обобщенной кооперации создания, посылки, приема и представления торгового объекта

При реализации, например, варианта использования *Послать счет покупателю*, каждый из абстрактных классификаторов, участвующих в обобщенной кооперации, подменяется конкретным классификатором, как показано на рис. 9.33.

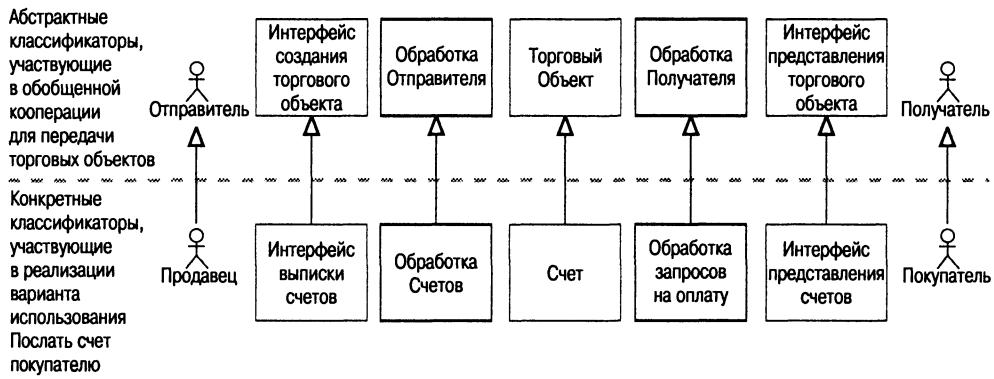


Рис. 9.33. Абстрактные классификаторы, участвующие в обобщенной кооперации, подменяются конкретными классификаторами, участвующими в конкретной реализации варианта использования

В соответствии с этим реализация варианта использования *Послать счет покупателю* должна просто ссылаться на обобщенную кооперацию, но не повторять или уточнять ее, реализуя предоставляемые (виртуальные) операции абстрактных классификаторов при помощи методов и конкретных классификаторов.

Такой же подход применяется и при реализации варианта использования *Заказать товары или услуги*.

Отметим, что использование обобщений — не единственный путь применения обобщенных коопераций. Так, например, образцы, которые представляют собой параметризованную кооперацию (то есть параметризованные классы), также являются обобщенными и могут быть использованы путем ассоциирования с параметрами конкретных классов.

Большая часть обобщенных механизмов выявляется в ходе фазы проектирования. Осторожно подходя к этому вопросу, архитектор может разработать набор механизмов, которые разрешают наиболее сложные проблемы проектирования и делают реализацию вариантов использования значительно проще и лучше подготовленной для реализации на фазе построения. Механизмы, относящиеся к поставляемым программным продуктам, будут естественными кандидатами на средний уровень программного обеспечения. Другие механизмы обычно занимают свое место на общем уровне приложений.

Деятельность: Проектирование вариантов использования

Цели проектирования вариантов использования.

- Определение классов проектирования и/или подсистем, экземпляры которых необходимы для осуществления потока событий варианта использования.
 - Распространение поведения варианта использования на взаимодействующие объекты проектирования и/или участвующие подсистемы.
 - Определение требований к операциям классов проектирования и/или подсистем и их интерфейсам.
- Определение требований к реализации вариантов использования.

Определение участвующих классов проектирования

На этом этапе мы определяем классы проектирования, необходимые для реализации варианта использования (рис. 9.34). Сделаем это так.

- Изучим классы анализа, участвующие в соответствующем анализе реализации варианта использования. Определим классы проектирования, трассируемые к этим классам анализа, которые были созданы инженером по компонентам в ходе проектирования классов или архитектором во время проектирования архитектуры.
 - Изучим специальные требования соответствующих анализов реализации вариантов использования. Определим классы проектирования, которые реализуют эти специальные требования. Их находит либо архитектор в ходе проектирования архитектуры (в виде обобщенных механизмов), либо инженер по компонентам в ходе проектирования классов.
 - В результате происходит определение требуемого класса, и инженер по компонентам приписывает ему сферы ответственности.
- Если соответствующий класс проектирования, необходимый для проектирования варианта использования, отсутствует, инженер по вариантам использования должен связаться с архитектором или инженером по компонентам. Следует определить необходимый класс, этим должен заняться инженер по компонентам.

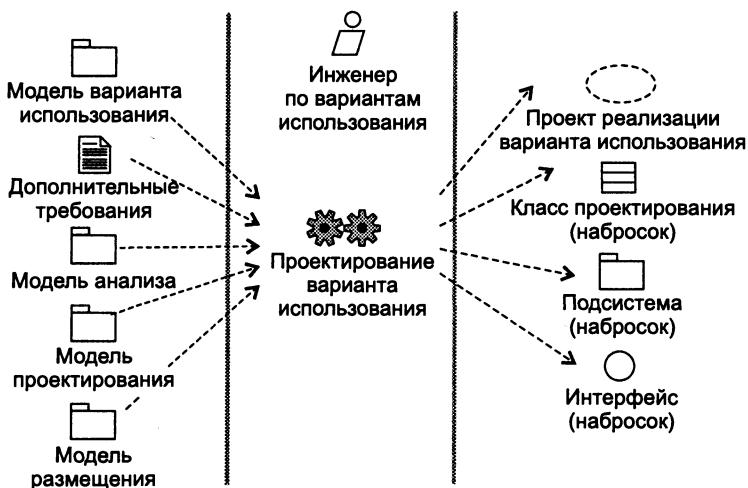


Рис. 9.34. Исходные данные и результаты проектирования вариантов использования. Исходными данными для этой деятельности является соответствующий анализ реализации варианта использования

Соберите классы проектирования, участвующие в реализации варианта использования, в диаграмму классов, ассоциирующуюся с реализацией. Используйте эту диаграмму классов для демонстрации отношений, существующих в реализации варианта использования.

Пример. Классы, участвующие в реализации варианта использования *Оплатить счет*. На рисунке 9.35 изображена диаграмма классов. На ней показаны классы, участвующие в реализации варианта использования *Оплатить счет*, и их взаимные ассоциации.

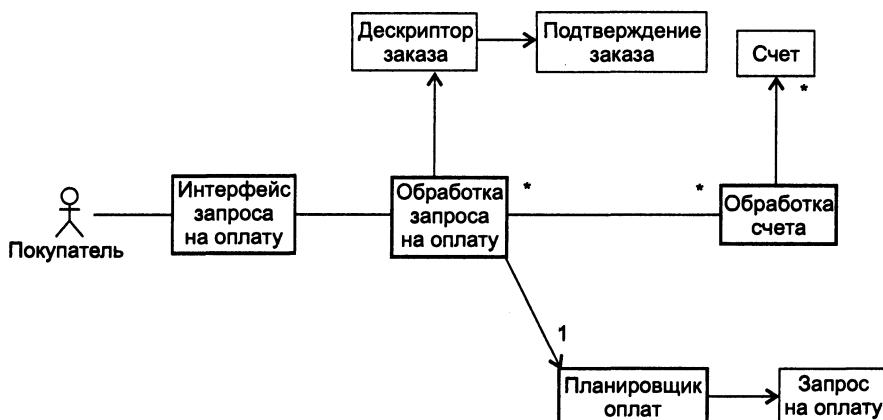


Рис. 9.35. Классы, участвующие в реализации варианта использования *Оплатить счет*, и их ассоциации. Активные классы на диаграмме выделены толстой рамкой

Некоторые активные классы, например *Обработка запроса на оплату* и *Обработка Счетов*, сохраняют систему Interbank в рабочем состоянии. Они делают это, пересылая торговые объекты через различные узлы от создавшего их класса к принимающему, как, например, счета от продавца к покупателю.

Описание взаимодействия объектов проектирования

Создавая набросок описания классов проектирования, необходимых для реализации варианта использования, мы нуждаемся в описании взаимодействий между соответствующими объектами проектирования. Мы можем создать это описание, используя диаграммы последовательностей, содержащие экземпляры участвующих во взаимодействии актантов, объекты проектирования и сообщения, которыми они обмениваются. Если варианты использования имеют различные определенные потоки или подпотоки, обычно создается по одной диаграмме последовательности на каждый поток. Это позволяет прояснить реализацию варианта использования, а также выделить диаграммы последовательностей, на которых представлены общие или пригодные для повторного использования взаимодействия.

Чтобы начать этот этап, изучим соответствующий анализ варианта использования. Его можно использовать для получения наброска последовательности сообщений, которыми должны обмениваться объекты проектирования. Нам не помешает в этом даже то, что при проектировании в реализацию варианта использования может быть добавлено множество новых объектов проектирования. В некоторых случаях возможно даже преобразование диаграммы коопераций соответствующего анализа реализации варианта использования в базовое описание соответствующей диаграммы последовательностей.

Диаграмма последовательности создается по шагам варианта использования. Этот процесс начинается с начала потока событий варианта использования, а затем проходит по шагам весь этот поток событий. При этом мы решаем, какие взаимодействия объектов проектирования и экземпляров актантов необходимы для его осуществления. В большинстве случаев объекты естественно занимают свои места в последовательности взаимодействий реализации варианта использования. О диаграмме последовательности можно сказать следующее.

- Вариант использования порождается сообщением, посыпаемым от экземпляра актанта объекту проектирования.
- Каждый класс проектирования, определенный на предыдущем шаге, должен содержать как минимум один объект проектирования, входящий в диаграмму последовательности.
- Для осуществления варианта использования между линиями жизни (приложение А) объектов пересыпаются сообщения. Сообщения могут иметь временные названия. После того, как инженер по компонентам, который отвечает за класс объекта, определит операции класса, эти названия станут названиями операций.
- Последовательность действий на диаграмме должна быть в центре внимания, поскольку проект реализации варианта использования представляет собой основные исходные данные при реализации вариантов использования. Она важна для определения хронологического порядка обмена сообщениями.
- Используйте метки и формат потока событий для дополнения диаграммы последовательности.

- Диаграмма последовательности должна охватывать все существующие отношения варианта использования. Например, если вариант использования А специализирует вариант использования В посредством отношения обобщения, диаграмма последовательности, описывающая вариант использования А, должна ссылаться на описание (то есть диаграмму последовательности) варианта использования В. Отметим, что эта ссылка также присутствует и в соответствующем анализе реализации варианта использования.

Пример. Диаграмма последовательности для первой части варианта использования *Оплатить счет*. На рис. 9.36 приведена диаграмма последовательности для первой части варианта использования *Оплатить счет*.

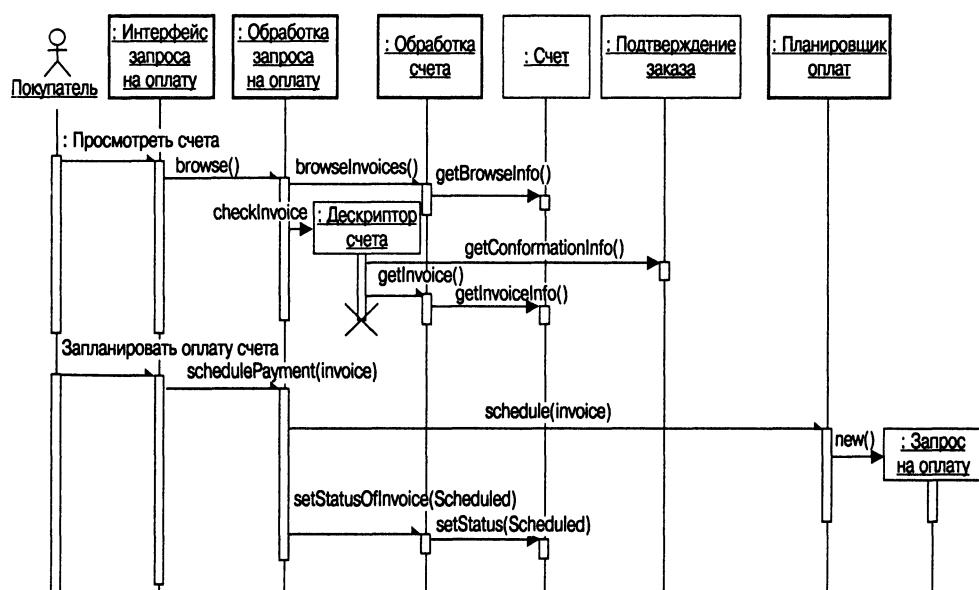


Рис. 9.36. Диаграмма последовательности для объектов, осуществляющих первую часть реализации варианта использования *Оплатить счет*

Проект потока событий, дополняющий диаграмму последовательности, выглядит следующим образом:

Покупатель работает с системой, просматривая присланные счета, через апплет *Интерфейс запроса на оплату* и приложение *Обработка запросов на оплату*. *Обработка запросов на оплату* использует *Дескриптор счета* для сравнения присланных счетов с заказами до того, как *Интерфейс запроса на оплату* покажет список счетов покупателю.

Покупатель через *Интерфейс запроса на оплату* выбирает счет и помечает его к оплате, а *Интерфейс запроса на оплату* передает этот запрос в *Обработку запросов на оплату*. *Обработка запросов на оплату* просит *Планировщик оплат* запланировать оплату счета. Затем *Планировщик оплат* создает запрос на оплату. Тогда *Обработка запросов на оплату* запрашивает приложение *Обработка счетов* изменить состояние счета на «Помечен к оплате».

После того как мы детализировали диаграмму взаимодействий, мы легко можем найти альтернативные пути осуществления вариантов использования. Эти пути могут быть обозначены метками на диаграмме или передаваться самой диаграммой взаимодействий. Добавляя новую информацию, инженер по вариантам использования часто находит новые исключения, которые не были замечены в ходе определения или анализа требований. Это исключения типа:

- Обработка тайм-аутов в случае, если узел или соединение прекращает функционировать.
- Неверные данные, введенные актантами — людьми или вспомогательными системами.
- Сообщения об ошибках, генерируемые программным обеспечением среднего уровня, системным программным обеспечением или аппаратными средствами.

Определение участвующих подсистем и интерфейсов

Итак, мы спроектировали вариант использования в виде кооперации классов и их объектов. Иногда, однако, бывает удобнее создавать проект варианта использования в понятиях взаимодействующих подсистем и/или их интерфейсов. Так, например, при разработке сверху вниз может понадобиться определить требования к подсистемам и их интерфейсам до начала проектирования их внутреннего содержания. Или, в некоторых случаях, подсистема с ее специфическим внутренним содержанием должна легко заменяться на другую подсистему с другим внутренним содержанием. В этом случае проекты реализации варианта использования могут быть определены на различных уровнях системной иерархии. На рисунке 9.37 линии жизни на средней диаграмме (а) представляют собой подсистемы и показывают сообщения, посылаемые и передаваемые подсистемам. Другие диаграммы (б, с) представляют внутреннее строение подсистем и показывают, как сообщения (из а) посылаются и принимаются внутренними элементами подсистем. Диаграмма (а) может быть создана раньше, чем (б) и (с).

Для начала следует определить подсистемы, необходимые для реализации варианта использования.

- Изучим классы анализа, задействованные в соответствующем анализе реализации классов использования. Определим пакеты анализа, в которых содержатся эти классы, если таковые существуют. Затем определим подсистемы проектирования, которые трассируются к этим пакетам анализа.
- Изучим специальные требования соответствующих анализов реализации вариантов использования. Определим классы проектирования, реализующие эти специальные требования, если они существуют. Затем определим подсистемы проектирования, содержащие эти классы.

Сберем подсистемы, участвующие в реализации варианта использования, в диаграмму классов, ассоциированную с этой реализацией. Используем эту диаграмму классов для демонстрации зависимостей между подсистемами и интерфейсами, применяемыми в реализации варианта использования.

Пример. Подсистемы и интерфейсы, участвующие в реализации варианта использования *Оплатить счет*. На рисунке 9.38 изображена диаграмма классов, содержащая подсистемы, интерфейсы и их зависимости, входящие в первую часть варианта использования *Оплатить счет*.

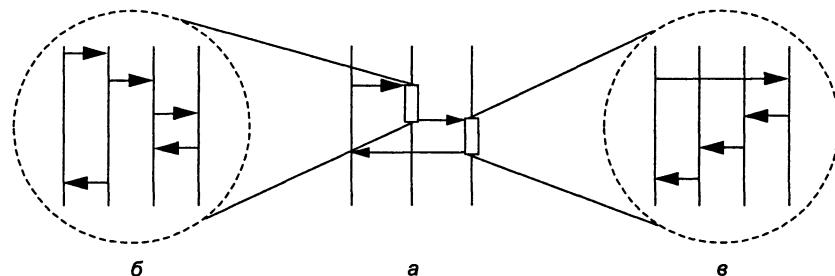


Рис. 9.37. Определение проектов реализации варианта использования на различных уровнях системной иерархии

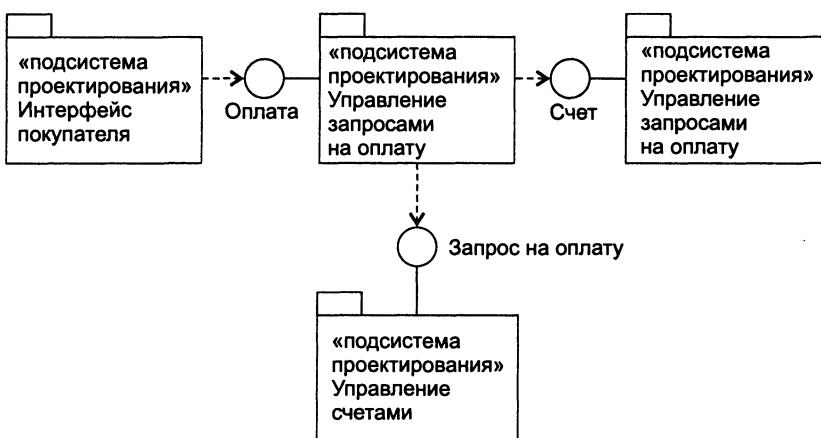


Рис. 9.38. Диаграмма классов с подсистемами, интерфейсами и их зависимостями

Описание взаимодействия подсистем

После получения описания подсистем, необходимых для реализации варианта использования, нам нужно показать, как взаимодействуют на уровне подсистем содержащиеся в этих подсистемах классы. Это делается путем использования диаграммы последовательности, которая содержит участвующие в варианте использования экземпляры актантов, подсистем и передаваемые между ними сообщения. Для выполнения этой работы мы воспользуемся способом, описанным в подразделе «Описание взаимодействия объектов проектирования», но внесем в него некоторые изменения.

- **Линии жизни** (приложение А) диаграммы последовательности определяют подсистемы, а не объекты проектирования.
- Каждая подсистема, определенная в подразделе «Определение участвующих подсистем и интерфейсов», должна описываться хотя бы одной линией жизни на диаграмме последовательности.
- Если за операцией или интерфейсом закреплено сообщение, может потребоваться описывать это сообщение вместе с интерфейсом, представляющим операцию. Это, например, будет актуально в том случае, если подсистема предоставляет несколько интерфейсов, и нужно выбрать тот интерфейс, который используется при посылке сообщения.

Определение требований к реализации

На этом этапе мы определяем все требования, необходимые для реализации варианта использования. В их число входят, например, нефункциональные требования, определенные при проектировании, которые следует решить на стадии реализации.

Пример. Требования к реализации варианта использования *Оплатить счет*. Рассмотрим требования к реализации варианта использования *Оплатить счет*: объект (активного) класса *Обработка запросов на оплату* должен быть в состоянии работать одновременно с 10 различными *Покупателями клиентами*, не создавая ощутимых задержек для каждого отдельного покупателя.

Деятельность: Проектирование класса

Цель проектирования класса – создать класс проектирования, исполняющий свои роли в реализациях вариантов использования и отвечающий предъявляемым к нему нефункциональным требованиям (рис. 9.39). В это понятие входит поддержка самого класса, а также следующих его характеристик:

- операций;
- атрибутов;
- отношений, в которые он вступает;
- методов (которые реализуют его операции);
- состояний, в которых он находится;
- зависимостей от обобщенных механизмов проектирования;
- требований, относящихся к его реализации;
- правильной реализации всех интерфейсов, которые он должен предоставить.

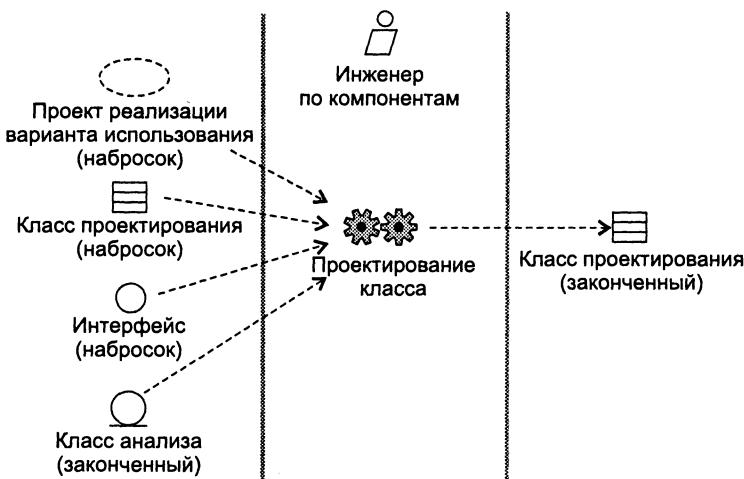


Рис. 9.39. Исходные данные и результат проектирования класса

Описание класса проектирования

Для начала нам нужно дать краткое описание одного или нескольких классов проектирования, которые мы получили в качестве исходных данных, в терминах классов и/или интерфейсов анализа. Если в качестве исходных данных мы получили интерфейс, можно просто назначить один класс проектирования предоставлять этот интерфейс.

Получив в качестве исходных данных один или несколько классов анализа, мы выбираем метод их обработки в зависимости от стереотипа класса анализа.

- Проектирование граничных классов определяется спецификой используемой технологии интерфейса. Так, например, граничные классы, проектируемые под Visual Basic, должны включать такие стереотипы классов проектирования, как «Форма», а также классы проектирования, которые представляют собой управляющие элементы пользовательского интерфейса, возможно, элементы Active X. Отметим также, что в некоторых современных утилитах для построения пользовательского интерфейса его можно создавать визуально, прямо на экране. Одновременно будут создаваться соответствующие классы проектирования. В качестве исходных данных для этой работы будут использованы заранее созданные прототипы пользовательского интерфейса.
- Проектирование классов сущности, содержащих информацию длительного хранения (или классов с другими требованиями к длительному хранению информации), часто определяется спецификой используемой технологии базы данных. Так, например, могут появиться классы проектирования, соответствующие таблицам реляционной модели данных. Этот шаг можно частично автоматизировать при помощи существующих в настоящее время средств моделирования. Делать это следует весьма аккуратно, не забывая о подстройке этих средств под принятую стратегию длительного хранения. Выбранная стратегия, особенно отображение объектно-ориентированной модели проектирования на модель реляционной базы данных, при проектировании порождает множество проблем. Эти проблемы могут, в свою очередь, потребовать включения в процесс разработки дополнительных сотрудников (проектировщиков баз данных), деятельности (проектирования баз данных) и моделей (моделей данных). Эти случаи выходят за пределы тем, рассмотренных в этой книге.
- Проектирование управляющих классов — тонкое дело. Поскольку они содержат последовательности, координацию с другими объектами, а иногда — и чистую бизнес-логику, следует рассмотреть следующие вопросы:
 - Проблемы размещения. Если последовательность должна быть размещена и управляться на нескольких различных узлах сети, при реализации управляющего класса должны быть созданы различные классы для разных узлов.
 - Проблемы производительности. Не должно быть оправдания для создания различных классов проектирования, реализующих управляющий класс. Вместо этого управляющий класс может быть реализован в том же классе проектирования, который реализует связанный с ним граничный класс и/или класс сущности.

- Проблемы транзакций. Управляющие классы часто содержат транзакции. Соответствующие классы проектирования должны включать в себя использование какой-либо из существующих технологий управления транзакциями.

Классы проектирования, определяемые на этом этапе, должны быть связаны трассировкой с соответствующими классами анализа, от которых они порождаются. Важно помнить о «природе» классов проектирования, поскольку мы будем уточнять их на последующих шагах.

Определение операций

На этом шаге мы определяем операции, которые должен выполнять класс проектирования, и описываем эти операции с использованием синтаксиса языка программирования. Сюда входит и определение уровня видимости каждой из операций (например, *public*, *protected* и *private* в C++). Исходными данными для этого послужат:

- Зависимости каждого из классов анализа, который трассируется в класс проектирования. Зависимости часто подразумевают наличие одной или нескольких операций. Более того, если в ответственностих описываются исходные данные и результаты, их можно рассматривать в качестве первоначального наброска формальных параметров и возвращаемых значений операций.
- Специальные требования каждого из классов анализа, который трассируется в класс проектирования. Напомним, что эти требования часто должны быть обработаны в модели проектирования, возможно, путем включения некоторых обобщенных механизмов или технологий проектирования, таких как технология баз данных.
- Интерфейсы, которые должен предоставлять класс проектирования. Операции с интерфейсами также должны быть представлены в классе проектирования.
- Проекты реализации вариантов использования, в которых участвует класс (см. подраздел «Деятельность: Проектирование вариантов использования»).

Операции класса проектирования должны поддерживать все роли, которые исполняет этот класс в различных реализациях вариантов использования. Роли выявляются путем просмотра реализаций вариантов использования на предмет того, входят ли класс или его объекты в диаграммы или описания проекта потока событий этих реализаций.

Пример. *Операции класса Счет.* Класс *Счет* участвует в нескольких реализациях вариантов использования, в частности, *Оплатить счет*, *Послать напоминание* и *Послать счет покупателю*. Каждая из этих реализаций читает или изменяет состояние объекта *Счет*. Вариант использования *Послать счет покупателю* создает и рассыпает *Счета*. Вариант использования *Оплатить счет* планирует оплату счетов и т. д. Каждая из этих реализаций вариантов использования использует объекты *Счета* по разному, другими словами, класс *Счет* и его объекты используются в этих реализациях вариантов использования различные роли.

Сначала инженеры по компонентам должны продумать, как поместить все эти изменения состояния в одну операцию под названием *setStatus*, которая будет

иметь параметр, указывающий на предполагаемое действие и состояние, в которое должен перейти объект (например, `setStatus(Scheduled)`). Затем они приходят к выводу, что предпочтительнее будет определить отдельные операции для каждого перехода из состояния в состояние. Более того, этот подход будет использоваться ими не только для класса *Счет*, но и для класса *Торговый объект*, подтипом которого является *Счет* (рис. 9.40). Класс *Торговый объект* будет поддерживать следующие операции (каждая из которых изменяет состояние *Торгового объекта*): `Create`, `Submit`, `Schedule` и `Close`. Однако это лишь виртуальные операции, которые определяют этикетки. Каждый класс — потомок класса *Торговый объект* — будет определять конкретные методы, реализующие эти операции.

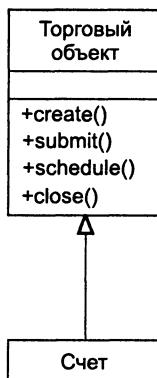


Рис. 9.40. Класс Счет, его родительский класс Торговый объект и операции, которые он поддерживает

Для некоторых классов поведение их объектов сильно зависит от состояния объекта. Такие классы удобнее всего описывать при помощи диаграмм состояний, см. подраздел «Описание состояний».

Определение атрибутов

На этом этапе мы определяем атрибуты, необходимые классу проектирования и описываем их, используя синтаксис языка программирования. Атрибут определяет свойство класса проектирования и часто участвует в операциях класса (мы указывали на это в предыдущем пункте). При определении атрибутов следует учитывать следующие общие соображения.

- Рассмотрите атрибуты классов анализа, трассируемых в классы проектирования. Иногда эти атрибуты соответствуют одному или нескольким атрибутам класса проектирования.
- Доступные типы атрибутов выбираются из языка программирования.
- Выбирая тип атрибута, постарайтесь использовать уже употреблявшиеся типы.
- Одиночный экземпляр атрибута не может совместно использоваться несколькими объектами проектирования. Если в этом существует необходимость, атрибут следует переопределить в виде отдельного класса.

- Если класс проектирования из-за своих атрибутов становится чересчур сложен для понимания, некоторые из этих атрибутов можно выделить из класса проектирования, переопределив их в виде отдельных классов.
- Если у класса имеется множество атрибутов или они чересчур сложны, его можно проиллюстрировать отдельной диаграммой класса, на которой изображается только та информация, которая относится к атрибутам.

Определение ассоциаций и агрегаций

Взаимодействие объектов проектирования показано на диаграмме последовательности. Это взаимодействие нередко требует ассоциаций между соответствующими классами. Инженер по компонентам должен изучить обмен сообщениями по диаграмме последовательности, чтобы определить, какие ассоциации необходимы в данном случае. Экземпляры ассоциаций могут быть использованы для хранения ссылок на другие объекты и группирования объектов в агрегации для посылки им сообщений.

Число отношений между классами должно быть сведено к минимуму. Ассоциациями и агрегациями моделируются не первоначальные отношения, взятые из реального мира, а отношения, созданные в ответ на требования различных реализаций вариантов использования. Кроме того, отметим, что поскольку в ходе проектирования должны быть решены проблемы производительности, следует провести моделирование оптимальных маршрутов поиска среди ассоциаций и агрегаций.

При определении и уточнении ассоциаций и агрегаций следует учитывать следующие общие соображения.

- Рассмотрите ассоциации или агрегации, включающие в себя соответствующие классы анализа. Иногда эти отношения (модели анализа) соответствуют одному или нескольким отношениям (модели проектирования), в которые входит класс проектирования.
- Уточните сложность ассоциаций, названия ролей, классы ассоциаций, упорядоченные роли, именованные роли и п-арные ассоциации в плане поддержки этих структур используемым языком программирования. Так, например, имена ролей могут при генерации кода стать атрибутами класса проектирования, зафиксировав таким образом форму имен ролей. Или класс ассоциации может стать новым классом, соединяющим два (оригинальных) класса, что потребует новых ассоциаций соответствующей сложности между классом «ассоциации» и двумя другими классами.
- Уточните направление просмотра ассоциаций. Воспользуйтесь диаграммой взаимодействий, в которых участвуют эти ассоциации. Направление передачи сообщений между объектами проектирования будет соответствовать направлению просмотра ассоциаций между этими классами.

Определение обобщений

Обобщения используются с той же семантикой, которая определена в языках программирования. Если язык программирования не поддерживает обобщений (или

наследования), то для делегирования (то есть посылки сообщений – требования начать работу, сигнала о выполнении работы и посылки подтверждающего сообщения) от объектов более специализированного класса объектам более общих классов следует использовать ассоциации и/или агрегации.

Пример. *Обобщения в модели проектирования.* Как *Счета*, так и *Заказы* изменяют свое состояние одинаковым образом и поддерживают схожие операции. Оба этих класса являются специализациями более общего *Торгового объекта* (рис. 9.41).



Рис. 9.41. Торговый объект обобщает Счет и Заказ. Отметим, что это обобщение существует также и в модели анализа между соответствующими классами анализа

Описание методов

Методы используются в проектировании для определения того, как будут реализованы операции. Так, например, метод может определить алгоритм, используемый для реализации операции. Метод можно определять, используя естественный язык или псевдокод, если он больше подходит для этой цели.

Однако в большинстве случаев методы в ходе проектирования не определяются. Вместо этого они создаются в ходе реализации сразу на выбранном языке программирования. Это происходит потому, что проектирование и реализацию класса выполняет один и тот же инженер по компонентам. Такое положение исключает необходимость передачи таких спецификаций, как спецификации методов.

Отметим, что утилиты проектирования поддерживают непосредственную генерацию кода для методов классов проектирования, причем методы могут быть определены прямо внутри утилиты проектирования на языке программирования. Этот процесс будет обсуждаться вместе с другими видами деятельности, относящимися к реализации, он не имеет отношения к собственно проектированию. За деталями мы отошлем читателей к обсуждению деятельности по реализации классов в главе 10 (подраздел «Деятельность: Реализация класса»).

Описание состояний

Некоторые объекты проектирования управляются состояниями. Это означает, что их состояние определяет их поведение при получении сообщения. В этих слу-

чаях для описания различных переходов объекта проектирования из состояния в состояние разумно будет использовать диаграмму состояний. Эта диаграмма состояний будет использована для реализации соответствующего класса проектирования.

Пример. Диаграмма состояний для класса Счет. Объекты *Счета* изменяют свое состояние при создании, отсылке, постановке в план оплат и закрытии. Как и для прочих торговых объектов, эти состояния сменяют одно другое в строгой последовательности. Так, счет не может быть поставлен в план на оплату до того, как он будет отослан. Эта последовательность изменения состояний может быть определена при помощи диаграммы состояний (рис. 9.42).

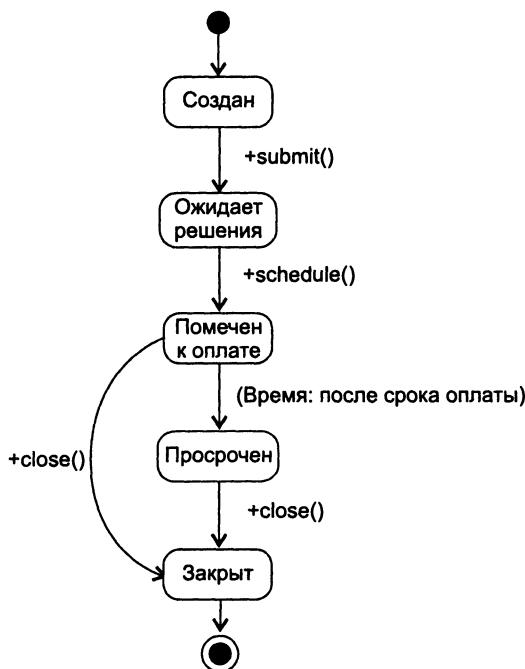


Рис. 9.42. Торговый объект обобщает Счет и Заказ. Отметим, что это обобщение существует также и в модели анализа между соответствующими классами анализа

Счет создается, когда продавец желает, чтобы покупатель оплатил свой заказ. Затем счет пересыпается покупателю и его состояние меняется на *Ожидает решения*. Когда покупатель принимает решение оплатить счет, состояние счета меняется на *Помечен к оплате*. Затем, если покупатель не оплачивает счет вовремя, состояние счета меняется на *Просрочен*. Когда наконец счет оплачивается, его итоговое состояние — *Закрыт*.

Учет специальных требований

На этой стадии обрабатываются все те требования, которые не были учтены на предыдущих шагах. Для этого следует рассмотреть все требования, предъявляемые

шиеся к реализациям вариантов использования, в которых участвует этот класс. Это даст нам (нефункциональные) требования к классу проектирования. Кроме того, необходимо изучить все нефункциональные требования к классу анализа, от которого трассируется этот класс проектирования. Эти специальные требования нередко бывает необходимо обработать в классе проектирования.

Пример. Использование механизма проектирования для обработки специальных требований. Объекты *Счета* должны быть доступны с нескольких узлов, в частности, как с *Сервером покупателя*, так и с *Сервером продавца*. *Счет* не является активным классом, но он должен быть спроектирован для использования в распределенных системах. В нашем примере мы реализуем такую распределенность объекта, сделав класс *Счет* дочерним классом абстрактного класса Java, java.rmi.UnicastRemoteObject, который поддерживает Remote Message Invocation (RMI), рис. 9.43. Отметим, что этот механизм проектирования определяется и описывается архитектором в ходе деятельности по проектированию архитектуры.

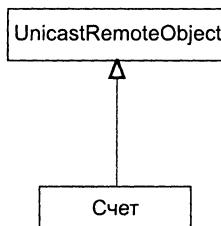


Рис. 9.43. Объекты класса Счет должны быть распределенными.
Это достигается при помощи наследования класса Счет
от класса UnicastRemoteObject

При обработке этих требований следует по возможности использовать все обобщенные механизмы проектирования, обнаруженные архитектором.

Тем не менее может быть необходимо отложить обработку некоторых требований до реализации. Такие требования будут обозначены как *требования к реализации* класса проектирования.

Пример. Требования к реализации активного класса. Вот пример требований к реализации класса *Обработка запросов на оплату*.

Объект класса *Обработка запросов на оплату* должен быть в состоянии одновременно обслуживать 10 различных *Покупателей клиентов* без заметной для покупателей задержки.

Деятельность: Проектирование подсистемы

Целью проектирования подсистемы является (рис. 9.44):

- Обеспечить максимально возможную независимость подсистемы от других подсистем и/или их интерфейсов.
- Обеспечить предоставление системой правильных интерфейсов.
- Обеспечить выполнение подсистемой ее задач, что включает в себя корректную реализацию операций в соответствии с определениями предоставляемых ею интерфейсов.

Сохранение зависимостей между подсистемами

Должны быть выявлены и сохранены зависимости подсистемы от других подсистем, содержащих элементы, ассоциированные с элементами данной подсистемы. Однако, если эти другие подсистемы предоставляют интерфейсы, зависимости должны быть определены между интерфейсами. Лучше зависимость от интерфейса, чем от подсистемы, поскольку подсистема может быть заменена другой подсистемой с другим внутренним строением, в то время как в замене интерфейса в таком случае необходимости не будет.

Старайтесь минимизировать зависимость от других систем и/или их интерфейсов. Рассмотрите возможность перемещения классов, которые слишком сильно зависят от других подсистем, в эти подсистемы.

Сохранение предоставляемых подсистемой интерфейсов

Операции, определенные в интерфейсах, предоставляемых подсистемой, должны поддерживать все роли, исполняемые подсистемой в различных реализациях вариантов использования. Даже если интерфейсы были описаны архитектором, они нуждаются в уточнении инженером по компонентам по мере создания модели проектирования и проектирования вариантов использования.

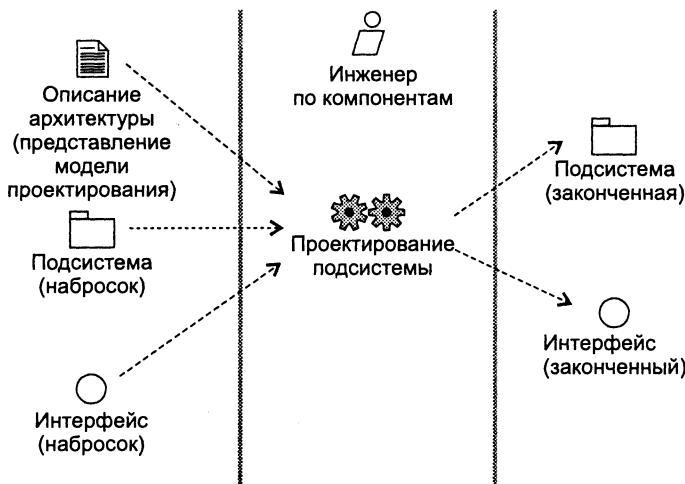


Рис. 9.44. Исходные данные и результат проектирования подсистемы

Напомним, что подсистема и ее интерфейсы могут быть задействованы инженером по вариантам использования в различных реализациях вариантов использования, а значит они должны, кроме всего прочего, предоставлять требования к интерфейсам (см. подраздел «Описание взаимодействия подсистем»).

Подход к сохранению интерфейсов и их операций подобен подходу к сохранению классов проектирования и их операций, описанному в подразделе «Определение операций».

Сохранение содержимого подсистемы

Подсистема выполняет свои задачи, предоставляя правильную реализацию операций, определенных в ее интерфейсе. Даже если содержимое подсистемы было описано архитектором, оно может потребовать уточнения инженером по компонентам по мере создания модели проектирования. К этому содержимому подходят следующие общие правила.

- Для каждого интерфейса, предоставляемого подсистемой, внутри этой подсистемы существует класс проектирования или другая подсистема, непосредственно предоставляющие этот интерфейс.
- Для выяснения, каким именно образом внутреннее строение системы реализует любые из этих интерфейсов или вариантов использования, должна быть создана кооперация в терминах содержащихся в подсистеме элементов. Это делается для уточнения содержащихся в подсистеме элементов.

Пример. Класс проектирования, предоставляющий интерфейс внутри подсистемы. Подсистема Управление счетами продавца предоставляет интерфейс Счет. Инженер по компонентам, ответственный за подсистему, решает, что класс Счет должен реализовывать собственный интерфейс (рис. 9.45). Альтернативной реализацией могла бы быть реализация интерфейса другим классом проектирования с последующим его использованием классом Счет.



Рис. 9.45. Интерфейс Счет, предоставляемый подсистемой Управление счетами покупателя, предоставляет класс Счет

Рабочий процесс проектирования — резюме

Главным результатом проектирования является модель проектирования, которая, сохраняя структуру системы, определенную в модели анализа, служит чертежом для последующей реализации. Модель проектирования включает в себя следующие элементы:

- Подсистемы проектирования, сервисные подсистемы, а также их зависимости, интерфейсы и содержание. Подсистемы проектирования двух верхних уровней (а именно, специфического уровня приложений и общего уровня приложений) описываются на основании пакетов анализа (подраздел «Определение прикладных подсистем»). Некоторые из зависимостей подсистем проектирования опи-

сываются на основании соответствующих зависимостей пакетов анализа (подраздел «Определение зависимостей между подсистемами»). Некоторые из интерфейсов описываются на основании классов анализа (подраздел «Определение интерфейсов подсистем»).

- Классы проектирования, включая активные классы и их операции, атрибуты, отношения и требования к реализации. Некоторые архитектурно значимые классы проектирования описываются на основе архитектурно значимых классов анализа (подраздел «Определение классов проектирования на основе классов анализа»). Некоторые активные классы описываются на основании классов анализа (подраздел «Определение активных классов»). В основном классы анализа при описании классов проектирования используются в качестве спецификации (подраздел «Описание класса проектирования»).
- Проекты реализации вариантов использования, описывающие проектирование вариантов использования в терминах коопераций внутри модели проектирования. В основном при описании проектов реализации вариантов использования в качестве спецификации используются анализы реализации вариантов использования (подраздел «Деятельность: Проектирование вариантов использования»).
- Архитектурное представление модели проектирования, включая архитектурно значимые элементы. Как указывалось ранее, при описании архитектурно значимых элементов модели проектирования в качестве спецификации используются архитектурно значимые элементы модели анализа.

Результатом проектирования также является модель развертывания, определяющая все конфигурации сетей, на которых может выполняться система. Модель развертывания включает в себя.

- Узлы, их характеристики и типы соединений.
- Исходное распределение активных классов по узлам.
- Архитектурное представление модели развертывания, включая архитектурно значимые элементы.

Модели проектирования и развертывания являются основными исходными данными для последующей деятельности по реализации и тестированию. Мы поговорим об этом в следующих главах. Более конкретно можно указать, что:

- Подсистемы проектирования и сервисные подсистемы будут реализованы в виде подсистем реализации, содержащих реальные компоненты, такие как файлы исходных текстов программ, сценариев, двоичные, исполняемые файлы и т. д. Эти подсистемы реализации будут трассироваться от подсистем проектирования один в один (изоморфно).
- Классы проектирования будут реализованы в виде файлов компонентов, содержащих исходные тексты программ. Часто несколько классов проектирования реализуются в виде одного файла; это зависит от используемого языка программирования. Кроме того, в качестве исходных данных для поиска исполняемых компонентов будут использоваться активные классы проектирования, соответствующие тяжеловесным процессам.

- Проекты реализаций вариантов использования будут использованы при планировании реализации и разбиении ее на небольшие управляемые части, вовлекаемые в билдах. Каждый билд в первую очередь реализует набор реализаций вариантов использования или их частей.
- Модель развертывания и конфигурации сетей используются для распределения системы и развертывания исполняемых компонентов на узлах.

10 Реализация

Введение

В ходе реализации мы, отталкиваясь от результатов проектирования, реализуем систему в виде компонентов — исходных текстов программ, сценариев, двоичных файлов, исполняемых модулей и т. д.

К счастью, большая часть архитектуры системы была определена в ходе проектирования. Основная цель реализации — «нарастить мясо» как на архитектуру, так и на систему в целом. Более конкретно, целью реализации является:

- Планирование необходимой на каждой итерации сборки системы. Мы используем инкрементный подход к разработке, результатом чего является реализация системы посредством последовательности малых управляемых шагов.
- Распределение системы путем отображения исполняемых компонентов на узлы модели размещения. Эта деятельность базируется на активных классах, обнаруженных в ходе анализа.
- Реализация классов и подсистем проектирования, обнаруженных в ходе проектирования. Так, классы проектирования реализуются в виде файлов компонентов, содержащих исходные тексты программ.

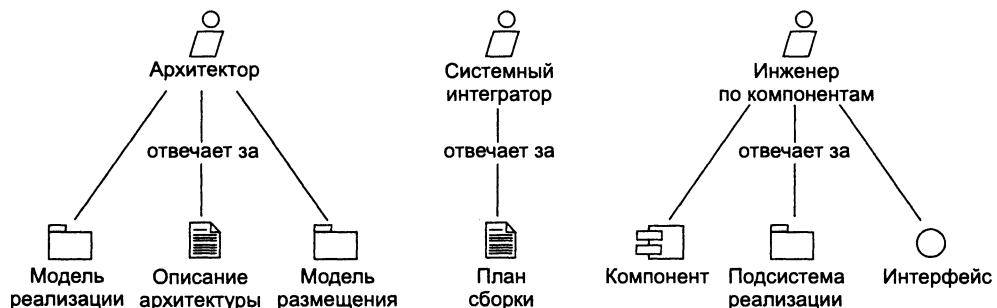


Рис. 10.1. Сотрудники и артефакты, вовлеченные в реализацию

- Покомпонентное тестирование с последующей сборкой компонентов путем компиляции и связывания их в один или более исполняемых модулей, осуществляющееся перед передачей их на тестирование целостности, и системные тесты.

В этой и следующей главах мы опишем, как происходит реализация и какие сотрудники и артефакты в нее вовлечены (рис. 10.1). Мы рассмотрим рабочий процесс реализации точно так же, как делали это для проектирования.

Роль реализации в жизненном цикле разработки программного обеспечения

Проектирование находится в центре внимания разработчиков в течение итераций фазы построения. Реализация осуществляется также в ходе фаз проектирования — для создания исполняемого базового уровня архитектуры и в ходе внедрения — для исправления последних дефектов, например, обнаруженных в ходе тестирования бета-версии системы (рис. 10.2, пик на фазе внедрения).

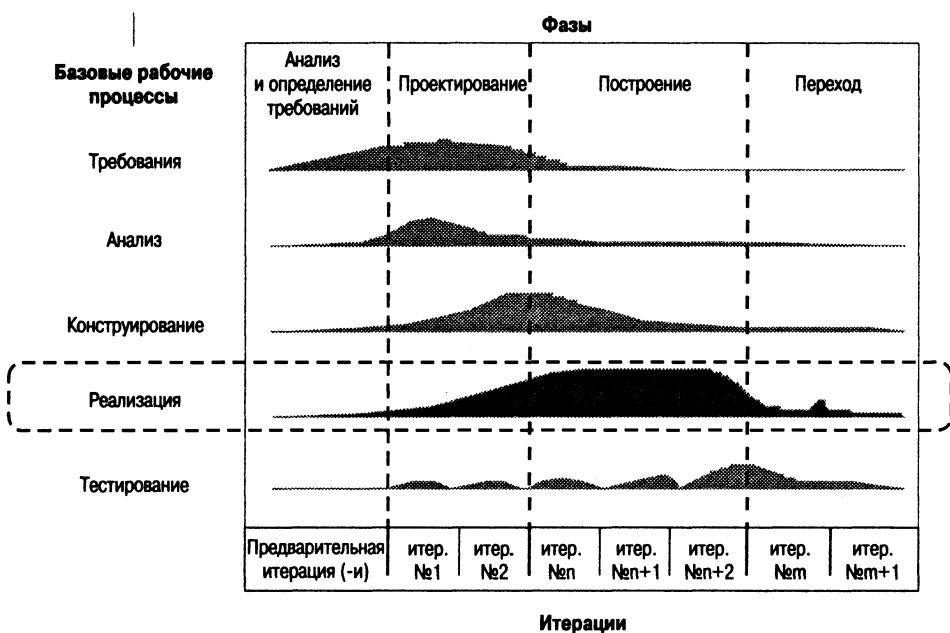


Рис. 10.2. Место реализации в жизненном цикле системы

Поскольку модель реализации описывает текущую реализацию системы в терминах компонентов и подсистем реализации, будет естественно сохранить ее на все время жизненного цикла системы.

Артефакты

Артефакт: Модель реализации

Модель реализации описывает, как реализуются в виде компонентов элементы модели проектирования, такие как классы проектирования. Под компонентами мы понимаем файлы с исходным текстом программ, исполняемые модули и т. д. Модель реализации также описывает способ организации этих компонентов в соответствии с механизмами структурирования и разбиения на модули, принятыми в выбранной среде разработки и используемых языках программирования, и тем, как компоненты реализации зависят друг от друга.

Модель реализации, как показано на рис. 10.3, определяет иерархию элементов.

Модель реализации представляется в виде системы реализации, обозначающей подсистему верхнего уровня в модели. Использование других подсистем — это способ разделить модель реализации на легче управляемые части.

Артефакты модели реализации представлены в деталях в следующих пунктах.

Артефакт: Компонент

Компонент — это физический контейнер для элементов модели, таких, как классы проектирования из модели проектирования [63]. Вот часть стандартных стереотипов компонентов.

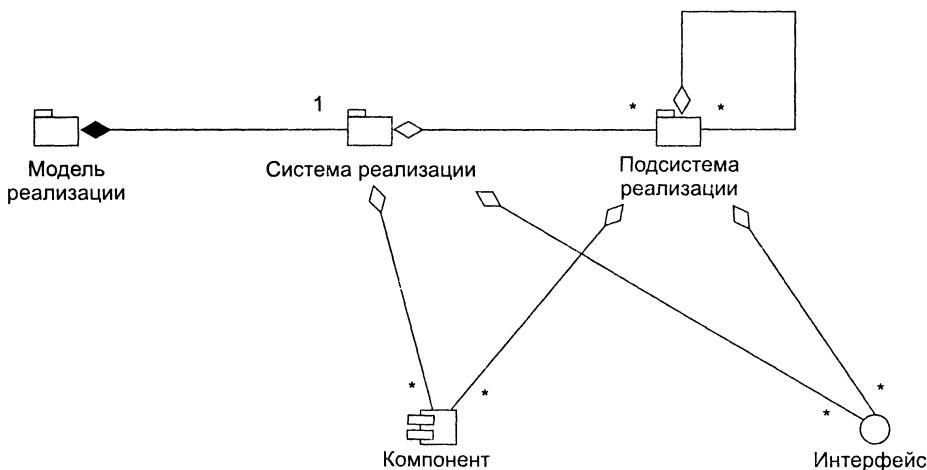


Рис. 10.3. Модель реализации представляет собой иерархию подсистем реализации, содержащих компоненты и интерфейсы

- «Исполняемый модуль» — программа, которую можно запустить на узле.
- «Файл» — файл, содержащий исходные тексты программ или данные.
- «Библиотека» — статически или динамически компонуемая библиотека.
- «Таблица» — таблица реляционной базы данных.
- «Документ» — документ.

При создании компонентов в конкретной среде реализации эти стереотипы могут изменяться, отражая истинное положение дел. Компоненты имеют следующие особенности.

- Компоненты отслеживают отношения тех элементов модели, которые они реализуют.
- Обычно компонент реализует несколько элементов, например классов проектирования, однако реальный способ реализации зависит от того, как, с учетом используемого языка программирования, будут структурированы и разбиты на модули файлы с исходным текстом программ.

Пример. Трассировка компонентов от класса проектирования. В системе Interbank класс проектирования *Перечисление денег со счета на счет* реализован в виде компонента исходного текста программы AccountTransfers.java. Это сделано именно так, потому что создание одного файла компонента с расширением «.java» с исходным текстом на каждый класс – это предписанный и традиционный способ использования языка Java, несмотря на то, что эта специфика не подкреплена никакими силовыми методами. Тем не менее, эта ситуация моделируется при помощи зависимости трассировки между моделями проектирования и реализации (рис. 10.4).

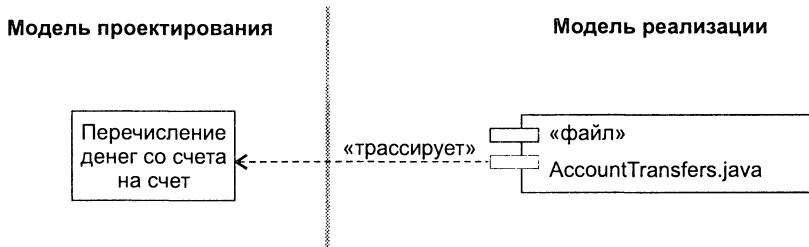


Рис. 10.4. Зависимость трассировки между компонентом и классом проектирования

Компоненты предоставляют те же интерфейсы, что и элементы моделей, реализацией которых они являются.

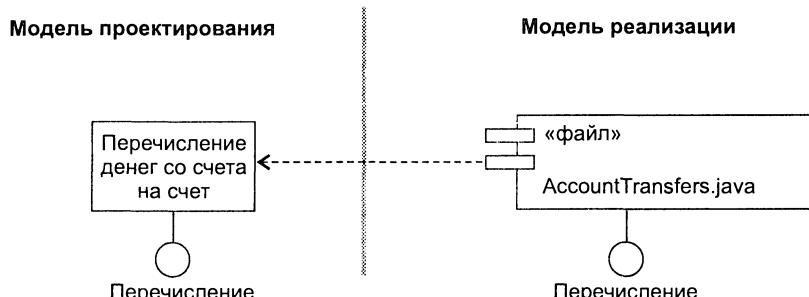


Рис. 10.5. Компоненты предоставляют те же интерфейсы, что и классы, которые они реализуют

Пример. Интерфейсы в проектировании и реализации. В системе Interbank класс проектирования *Перечисление денег со счета на счет* предоставляет интерфейс *Перечисление*.

речисление. Этот интерфейс также предоставляет и компонент AccountTransfers.java, который реализует класс *Перечисление денег со счета на счет* (рис. 10.5).

Между компонентами может существовать зависимость компиляции, показывающая, какие компоненты следует скомпилировать для получения определенного компонента.

Пример. Зависимость компиляции между компонентами. В системе Interbank компонент AccountTransfers.java (файл) компилируется в компонент AccountTransfers.class (исполняемый компонент)¹ (рис. 10.6).

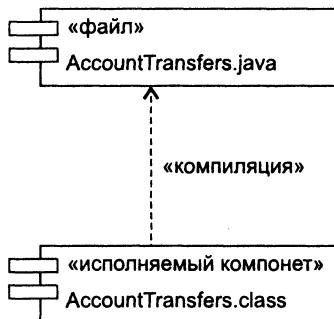


Рис. 10.6. Зависимость компиляции между двумя компонентами

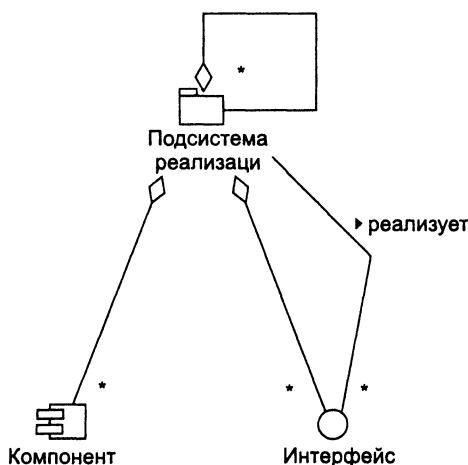


Рис. 10.7. Содержимое подсистемы и ключевые ассоциации

Заглушки

Заглушка — это компонент, реализованный особым образом или просто не полностью, который поэтому может быть использован для разработки или тестирования другого компонента, нуждающегося в такой заглушки. Более подробное опре-

¹ Точнее, это интерпретируемый компонент, который интерпретируется виртуальной машиной Java.

деление заглушки дано в [28]. Заглушки могут использоваться для минимизации числа новых компонентов, требуемых для выпуска каждой новой (промежуточной) версии системы, а также уменьшения проблем со сборкой и тестирования сборки (см. подраздел «Деятельность: Сборка системы»).

Артефакт: Подсистема реализации

Подсистемы реализации предоставляют способ организовать артефакты модели реализации в удобные в управлении группы (рис. 10.7). Подсистема может содержать компоненты, интерфейсы и другие подсистемы (рекурсивно). Более того, подсистема может реализовывать — и, соответственно, предоставлять — интерфейсы для передачи в терминах операций экспортруемой функциональности.

Важно понять, что реализуемая подсистема в реальности существует в виде, определяемом действующими в данной среде разработки «механизмами пакетирования», как-то:

- Пакет в Java [4].
- Проект в Visual Basic [54].
- Каталог с файлами в проекте на C++.
- Подсистема в интегрированной среде разработки, например Rational Apex.
- Пакет представлений компонентов в средствах визуального моделирования, например, в Rational Rose.

Итак, основная идея *подсистемы реализации* в том, чтобы дать семантику, уточненную в соответствии с выбранной средой реализации. Однако в этой главе мы обсудим общие концепции реализации, которые можно применить к различным средам реализации.

Подсистемы реализации крепко связаны с подсистемами проектирования из модели проектирования (см. подраздел «Артефакт: Подсистема проектирования» главы 9). Подсистема реализации фактически трассируется от соответствующей подсистемы проектирования один в один (изоморфно). Напомним, что подсистема проектирования полностью определяет:

- Зависимости от других подсистем и/или интерфейсов других систем.
- Интерфейсы, которые предоставляет подсистема.
- Классы проектирования или, рекурсивно, другие подсистемы проектирования внутри данной, которые на самом деле предоставляют интерфейсы данной подсистемы.

Эти аспекты важны для соответствующей подсистемы реализации по следующим причинам:

- Подсистема реализации должна определить аналогичные зависимости от других (соответствующих) подсистем реализации и/или интерфейсов.
- Подсистема реализации должна предоставить соответствующие интерфейсы.
- Подсистема реализации должна определить, какие именно компоненты или, рекурсивно, другие подсистемы реализации внутри данной на самом деле предоставляют интерфейсы данной подсистемы. Более того, эти внутренние компоненты (или подсистемы реализации) должны трассироваться к соответствую-

ющим классам (или подсистемам) проектирования, которые реализованы в подсистеме проектирования. Рисунок 10.8 поясняет отношения между моделями проектирования и реализации. Подсистемы различных моделей представляют одинаковые интерфейсы (α) и нуждаются в одинаковых интерфейсах (β). Компоненты подсистемы реализации реализуют классы подсистемы проектирования.

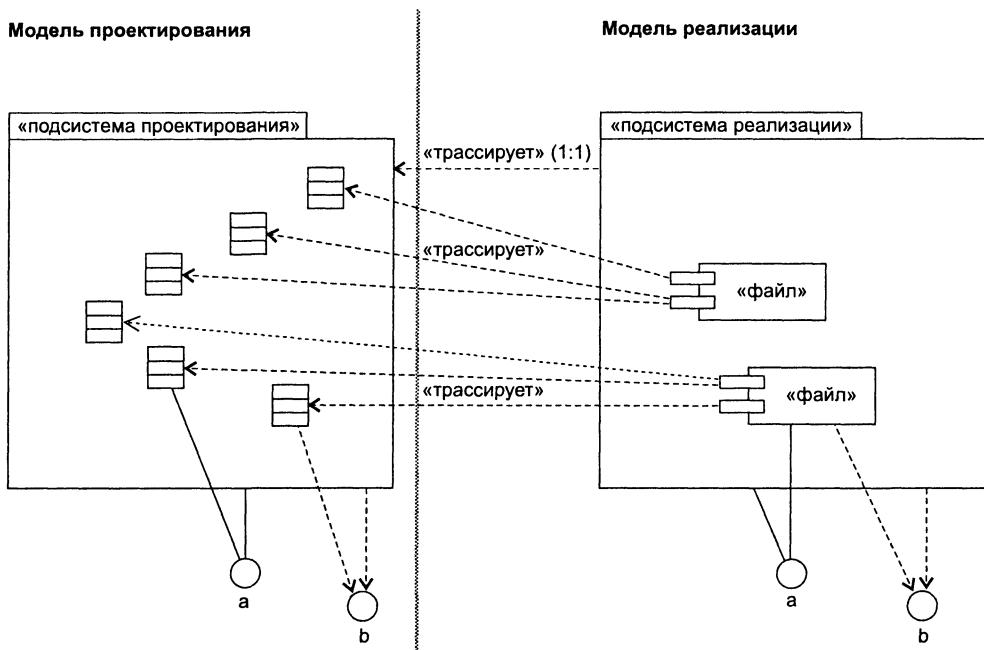


Рис. 10.8. Подсистема реализации из модели реализации трассируется один в один в подсистему проектирования в модели проектирования

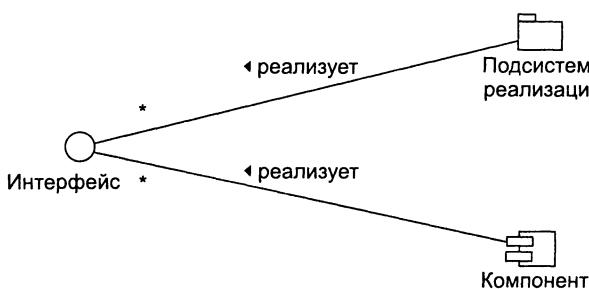


Рис. 10.9. Ключевые ассоциации интерфейса

Любые изменения способа, которым подсистема предоставляет и использует интерфейсы, или изменения, вносимые в сами интерфейсы, были рассмотрены в ходе рабочего процесса проектирования (см. подраздел «Деятельность: Проек-

тирование подсистемы» главы 9). Соответственно, в модели реализации эти изменения сделаны быть не могут, хотя они и затрагивают в том числе и модель реализации.

Отметим, что отображение, которое было здесь определено, таким же образом работает и с сервисными подсистемами модели проектирования. В результате подсистемы реализации, которые трассируются к сервисным подсистемам, находятся на самом нижнем уровне иерархии модели реализации и преследуют те же цели, что и сервисные подсистемы, а именно, структурирования системы в соответствии с предоставляемым ей сервисом. Поэтому подсистемы реализации, которые трассируются к сервисным подсистемам модели проектирования, обычно содержат исполняемые компоненты, предоставляющие различные системные сервисы.

Артефакт: Интерфейс

Интерфейсы детально описывались в главе 9. Однако некоторые интерфейсы могут использоваться в модели реализации для определения операций, реализованных в компонентах и подсистемах реализации. Более того, как говорилось ранее, компоненты и подсистемы реализации могут находиться с интерфейсами в «зависимости использования» (рис. 10.9).

Компоненты, которые реализуют (а значит, и предоставляют) интерфейсы, должны верно реализовывать все операции, которые предоставляют интерфейсы. Подсистемы реализации, предоставляющие интерфейс, должны также содержать компоненты или подсистемы реализации, которые непосредственно предоставляют этот интерфейс.

Пример. Система реализации, предоставляющая интерфейс. Система Interbank включает в себя подсистему реализации под названием AccountManagement (пакет Java), предоставляющий интерфейс Перечисление (рис. 10.10).

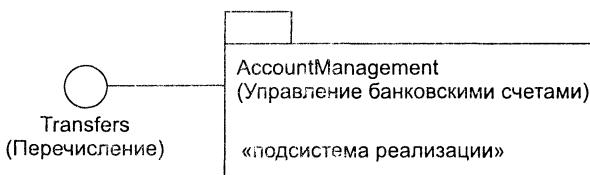


Рис. 10.10. Подсистема AccountManagement предоставляет интерфейс Перечисление

Код интерфейса Transfers на языке Java представлен ниже, в листинге 10.1.

Листинг 10.1. Код Java для интерфейса Transfers, предоставляемого подсистемой AccountManagement

```

package AccountManagement;
// provided interfaces:
public interface Transfers {
    public Account create(Customer owner, Money balance, AccountNumber account_d);
    public void Deposit (Money amount, String reason);
    public void Withdraw (Money amount, String reason);
}

```

Артефакт: Описание архитектуры (Представление модели реализации)

Описание архитектуры содержит архитектурное представление модели реализации (приложение В), описывающее архитектурно значимые артефакты (рис. 10.11). Архитектурно значимыми обычно считаются следующие артефакты модели реализации:

- Декомпозиция модели реализации на подсистемы, их интерфейсы и зависимости между ними. Вообще говоря, эта декомпозиция очень важна для архитектуры. Однако, поскольку подсистемы реализации один в один трассируются в подсистемы проектирования, а подсистемы проектирования уже описаны в архитектурном представлении модели проектирования, обычно в описании подсистем реализации в архитектурном представлении модели реализации нет необходимости.
- Ключевые компоненты (например, компоненты, трассируемые в архитектурно значимые классы проектирования или исполняемые компоненты), общие компоненты, центральные компоненты или компоненты, которые реализуют обобщенные механизмы проектирования, используемые многими другими компонентами.



Рис. 10.11. Архитектурное представление модели реализации

В подразделе «Определение архитектурно значимых компонентов» мы приведем примеры элементов, включаемых в архитектурное представление модели реализации.

Артефакт: План сборки

Важной особенностью создания программного обеспечения является его пошаговое построение в ходе небольших управляемых шагов. Каждый из шагов вызывает некоторые проблемы, связанные со сборкой или тестированием. Результат каждого шага называется «билд». Это исполняемая версия системы, обычно некоторая ее часть. Каждый билд становится затем объектом тестирования сборки (как описано в главе 11), которое должно быть проведено до создания нового билда. Для предупреждения появления плохих (то есть не прошедших тестирования) сборок

билдов каждый билд помещается под надзор системы контроля версий. После этого появляется возможность отката к предыдущему билду. Преимущества инкрементного подхода состоят в том, что:

- Исполняемая версия системы может быть создана на ранней стадии работ. Нет необходимости дожидаться более полной версии. Можно сразу начинать тестирование сборки, чтобы продемонстрировать особенности системы как членам команды разработчиков, так и внешним заинтересованным лицам.
- В ходе тестирования сборки появляется возможность раннего обнаружения дефектов, поскольку к созданному билду добавилась или была уточнена только малая часть, поддающаяся управлению. Более того, дефект часто (хотя и не всегда) связан с этой новой или уточненной частью.
- Тестирование сборки обычно более тщательное, чем полные тесты системы, поскольку внимание при тестировании сборки можно сосредоточить на небольших частях системы, лучше поддающихся управлению.

Инкрементная интеграция (приложение В) относится к системной интеграции так же, как управляемая итеративная разработка — к разработке программного обеспечения в целом. В ходе каждого из этих процессов внимание сконцентрировано на достаточно малых и управляемых приращениях функциональности.

Перейдем в контекст итеративной разработки. Результатом каждой итерации будет как минимум один билд. Однако функциональность, добавляемая на каждой итерации, часто чересчур сложна для того, чтобы интегрировать ее всю в один билд. Вместо этого мы создаем в пределах одной итерации последовательность билдов, каждый из которых представляет собой небольшой шаг вперед. В каждом из билдов к системе добавляется малая часть новой функциональности. В результате каждая итерация приносит большое приращение системной функциональности, возможно распределенное на несколько билдов (см. главу 5).

План сборки описывает последовательность итераций. Если говорить более точно, этот план описывает для каждого билда:

- Функциональность, которая должна быть реализована в этом билде. Это список вариантов использования и/или сценариев или их частей, обсуждавшихся в предыдущих главах. Этот список также может ссылаться на дополнительные требования.
- Части модели реализации, которые должен затрагивать билд. Это список подсистем и компонентов, требуемых для реализации функциональности, заявленной для билда.

В подразделе «Деятельность: Сборка системы» мы представим систематический подход к составлению плана сборки.

Сотрудники

Сотрудник: Архитектор

В ходе реализации архитектор отвечает за целостность модели реализации и обеспечивает корректность, целостность и читаемость модели в целом. Как и в ходе

анализа и проектирования, для больших и сложных систем за подсистему верхнего уровня (то есть за систему реализации) модели реализации может быть назначен ответственным отдельный сотрудник.

Модель верна, если она реализует функциональность, описанную в модели проектирования и в некоторых дополнительных требованиях, — и только эту функциональность.

Архитектор также отвечает за архитектуру модели реализации, то есть за существование ее архитектурно значимых частей, перечисленных в архитектурном представлении модели. Напомним, что это представление есть часть описания архитектуры системы, рис. 10.12.

Наконец, важным результатом реализации является распределение исполняемых компонентов по узлам. Архитектор отвечает за это распределение, расписанное в архитектурном представлении модели размещения. Детали можно посмотреть в главе 9, подраздел «Артефакт: Модель развертывания».

Отметим, что архитектор не отвечает за текущую разработку и поддержку различных артефактов модели реализации. Эта деятельность находится в компетенции инженера по компонентам.

Сотрудник: Инженер по компонентам

Инженер по компонентам определяет и поддерживает исходный текст компонента в одном или нескольких файлах, гарантируя, что этот компонент реализует правильную функциональность (в соответствии с определением в классе проектирования).

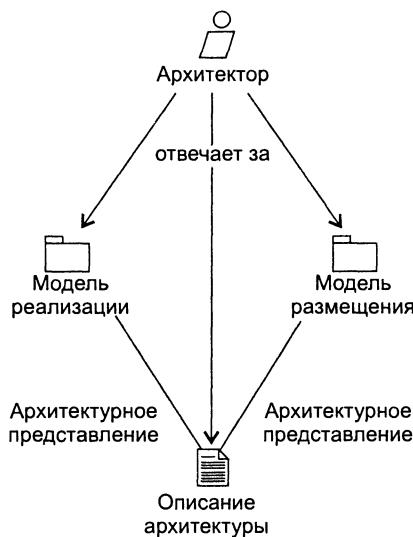


Рис. 10.12. Обязанности архитектора в процессе реализации

Инженер по компонентам нередко также отвечает за поддержание целостности одной или нескольких подсистем реализации. Поскольку подсистемы реализации

трассируются один к одному в подсистемы проектирования, большинство изменений в эти подсистемы вносится при проектировании. Однако инженер по компонентам должен постоянно проверять правильность содержимого (то есть компонентов) подсистем реализации, правильность их зависимостей от других подсистем и/или интерфейсов и т.о., насколько правильно подсистемы реализации реализуют предоставляемые интерфейсы, рис. 10.13.

Часто тот же инженер по компонентам, который отвечает за подсистему, отвечает также и за содержащиеся в ней компоненты. Более того, для того, чтобы разработка была плавной, естественно было бы, чтобы один и тот же инженер по компонентам отвечал за подсистему и ее содержимое в ходе рабочих процессов проектирования и реализации. Инженер по компонентам должен отвечать и за проектирование классов, и за их реализацию.

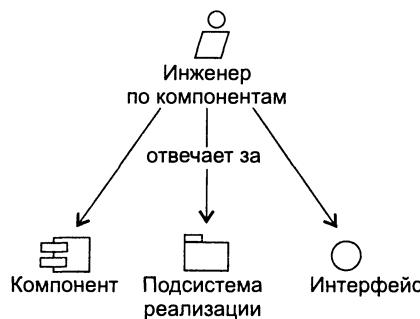


Рис. 10.13. Обязанности инженера по компонентам в процессе реализации



Рис. 10.14. Обязанности системного интегратора в процессе реализации

Сотрудник: Системный интегратор

Интеграция системы выходит из круга деятельности каждого конкретного инженера по компонентам. Это обязанность системного интегратора. В нее входит пла-

нирование последовательности билдов, необходимых на каждой итерации, и интеграция каждого билда по мере реализации его частей. Результатом планирования становится план сборки (рис. 10.14).

Рабочий процесс

В предыдущих пунктах мы рассматривали процесс реализации в статике. Теперь используем диаграмму деятельности для изучения его динамического поведения (рис. 10.15).

Основной целью процесса реализации является реализация системы. Она инициируется архитектором¹ путем предварительного описания ключевых компонентов модели реализации. После этого системный интегратор раскладывает текущую итерацию на последовательность билдов. Для каждого билда системный интегратор определяет ту функциональность, которая должна быть в нем реализована, и те части модели реализации (подсистемы и компоненты), которые должны быть в него включены. Требования, предъявляемые к компонентам и подсистемам билда, должны быть затем реализованы инженером по компонентам. Получившиеся компоненты тестируются по отдельности и передаются системному интегратору для включения в систему. Затем системный интегратор включает их в билд и передает тестерам интеграции для тестирования сборки (см. главу 11). После этого разработчики инициируют реализацию следующего билда, принимая во внимание дефекты предыдущего.

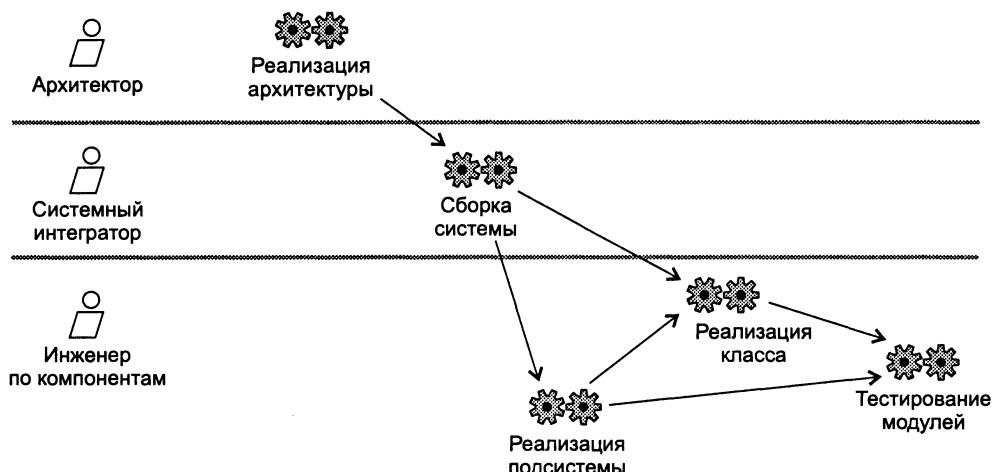


Рис. 10.15. Рабочий процесс реализации, включая участвующих в нем сотрудников и их деятельность

¹ То есть командой, совместно действующей в качестве сотрудника-архитектора, возможно, архитектурной группой.

Деятельность: Реализация архитектуры

Цель реализации архитектуры в том, чтобы описать модель реализации и ее архитектуру путем:

- определения архитектурно значимых компонентов, таких как исполняемые файлы;
- отображения компонентов на узлы существующих сетевых конфигураций.

Вспомним, что в ходе проектирования архитектуры (см. главу 9, подраздел «Деятельность: Проектирование архитектуры») были описаны подсистемы проектирования, их содержимое и интерфейсы. В ходе реализации мы используем подсистемы реализации, трассируемые один в один от соответствующих подсистем проектирования и предоставляющие такие же интерфейсы. Определение подсистем реализации и их интерфейсов – более или менее тривиальная задача, и мы не станем обсуждать ее подробно. Главной проблемой реализации является создание внутри подсистем реализации компонентов, реализующих соответствующие подсистемы проектирования.

В ходе этой деятельности архитектор сохраняет, уточняет и улучшает определение архитектуры и архитектурные представления моделей реализации и размещения.

Определение архитектурно значимых компонентов

Обычно для инициирования работ по реализации бывает полезно определять архитектурно значимые компоненты в начале жизненного цикла программы (рис. 10.16).

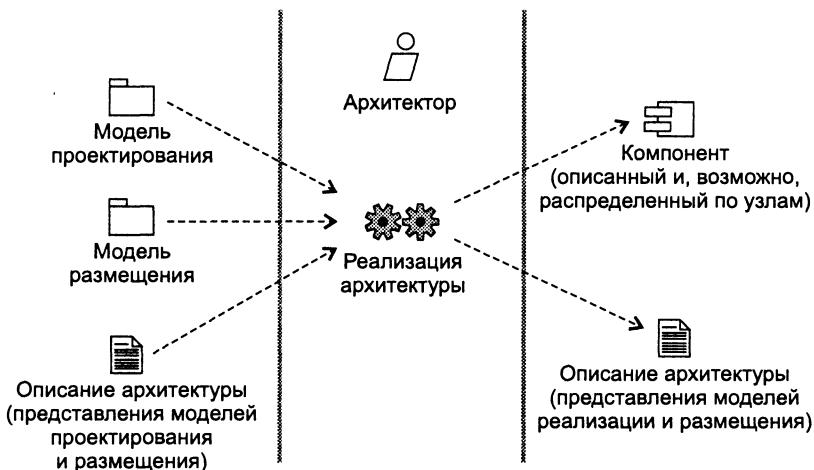


Рис. 10.16. Исходные данные и результаты реализации архитектуры

Однако множество компонентов, особенно файловых компонентов, разумнее первоначально создавать после реализации классов, поскольку в своей основе эти компоненты просто предоставляют средство для упаковки реализованных классов в виде файлов с исходными текстами программ. По этой причине разработчики должны быть осторожны с определением на этой стадии множества компонен-

тов или переходом к многочисленным деталям. Тем не менее большая часть работы должна быть проделана во время реализации классов. Первичного описания архитектурно значимых компонентов будет вполне достаточно (см. подраздел «Артефакт: Описание архитектуры (Представление модели реализации)»).

Определение исполняемых компонентов и распределение их по узлам

Для определения исполняемых компонентов, которые можно будет распределять по узлам, мы должны рассмотреть активные классы, обнаруженные в ходе проектирования, и сопоставить каждому из них один исполняемый компонент, указывающий на полноценный процесс. Сюда же входит и определение других файлов и/или бинарных компонентов, которые необходимы для создания исполняемых компонентов, но уже во вторую очередь.

Пример. *Определение исполняемых компонентов.* В модели проектирования имеется активный класс *Обработка запросов на оплату*. В ходе реализации мы определяем соответствующий компонент под названием *PaymentRequestProcessing* (рис. 10.17).

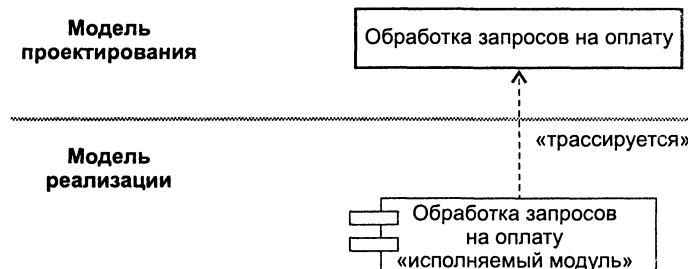


Рис. 10.17. Для определения исполняемых компонентов используются активные классы

Продолжая рассмотрение моделей проектирования и распределения, мы можем проверить, не следует ли распределить активные объекты по узлам. Если это так, компоненты, трассируемые к соответствующим активным классам, должны быть распределены по тем же самым узлам.

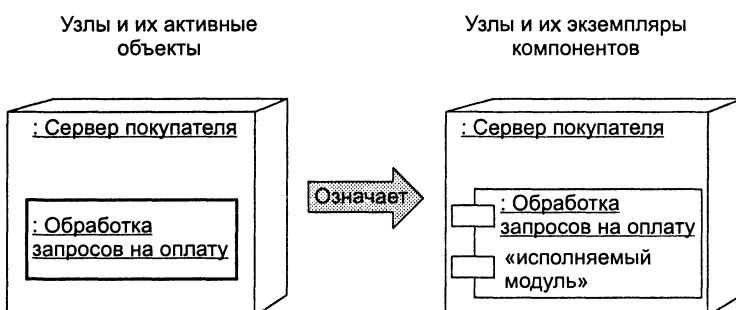


Рис. 10.18. Распределение активных компонентов по узлам означает соответствующее размещение компонентов

Пример. Распределение объектов по узлам. Активный объект класса *Обработка запросов на оплату* находится на узле Сервер покупателя.

Поскольку компонент PaymentRequestProcessing реализует класс *Обработка запросов на оплату* (см. предыдущий пример), он также должен быть размещен на узле *Сервер покупателя* (рис. 10.18).

Распределение компонентов по узлам очень важно для архитектуры системы и должно быть изложено в архитектурном представлении модели распределения.

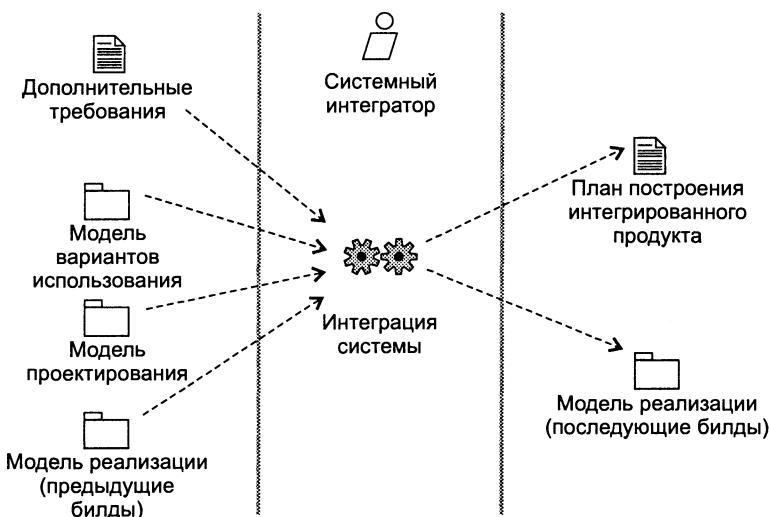


Рис. 10.19. Исходные данные и результаты системной интеграции.

Проекты реализации вариантов использования — основные исходные данные для этой деятельности

Деятельность: Сборка системы

Целью сборки системы является:

- Создание плана сборки, описывающего входящие в итерацию билды и требования к каждому из них (рис. 10.19).
- Сборка каждого из билдов до передачи его на тестирование сборки.

Планирование последующего билда

В этом пункте мы обсудим планирование содержания последующего билда. При этом мы исходим из имеющегося у нас предыдущего билда или некоторых общих соображений. Мы также допускаем, что у нас имеется набор вариантов использования и/или сценариев (то есть путей реализации вариантов использования) и других требований, которые следует реализовать на текущей итерации.

К последующему билду предъявляются следующие требования:

- Билд должен добавлять функциональность к предыдущим билдам, реализуя полные варианты использования и/или их сценарии. (Упрощая изложение для

сокращения списка, мы будем называть *варианты использования и/или сценарии* просто *вариантами использования*). На этих вариантах использования будут базироваться сборочные тесты билда, причем полные варианты использования тестируются легче, чем их фрагменты (см. главу 11).

- Билд не должен включать в себя слишком много новых или уточненных компонентов. В противном случае собрать билд и провести сборочные тесты будет очень сложно. Если это все же необходимо, часть компонентов может быть реализована в виде заглушек для минимизации числа добавляемых в билд новых компонентов (см. подраздел «Заглушки»).
- Билд может базироваться на предыдущем билде и развиваться вверх и в стороны по уровням иерархии подсистем. Это означает, что исходный билд начинается с подсистем нижних уровней (то есть среднего уровня и уровня системного программного обеспечения). Последующие билды развиваются дальше, захватывая общий и специфический уровни приложения. Это делается потому, что реализовать компоненты верхних уровней до помещения на место правильно функционирующих необходимых компонентов нижних уровней значительно сложнее.

Учитывая эти критерии, мы можем начать с оценки реализуемых требований, таких, как варианты использования (и/или их сценарии). Отметим, что выполняя условия таким образом, необходимо идти на компромисс. Так, реализация полного варианта использования (первый момент) может потребовать множества новых компонентов (второй момент), но если вариант использования важен для реализации текущего билда, его все равно следует реализовать. В любом случае важно правильно определить требования — те, которые следует реализовать в текущем билде и те, которые следует отложить на следующие билды. Для каждого потенциально реализуемого варианта использования следует сделать следующее:

1. Рассмотреть проект варианта использования для определения соответствующих ему проектов реализации вариантов использования из модели проектирования. Напомним, что проект реализации варианта использования может быть трансформирован к варианту использования (и соответственно, к каждому из его сценариев) через зависимости трансформации.
2. Определить классы и подсистемы проектирования, участвующие в проекте реализации варианта использования.
3. Определить подсистемы реализации и компоненты модели реализации, трансформируемые к подсистемам проектирования и классам, обнаруженным на шаге 2. Эти подсистемы реализации и компоненты понадобятся нам для реализации варианта использования.
4. Рассмотреть действия по реализации требований, предпринимаемые в этих подсистемах реализации и в компонентах текущего билда. Отметим, что эти требования определены в понятиях подсистем проектирования и классов, упомянутых на шаге 3. Решить, допустимы ли такие действия, согласуются ли они с описанными ранее критериями. Если это так, запланировать реализацию варианта использования в следующем билде. В противном случае отложить ее на будущее.

Результаты определяются в плане сборки и передаются инженерам по компонентам, которые отвечают за указанные в нем подсистемы реализации и компоненты. После этого инженеры по компонентам могут начать реализацию требований к подсистемам реализации и компонентам текущего билда (этот процесс описан в подразделах «Деятельность: Реализация подсистемы» и «Деятельность: Реализация класса») и тестирование частей (как описано в подразделе «Деятельность: Выполнение тестирования модулей»).

После этого подсистемы реализации и компоненты передаются системному интегратору для интеграции (см. следующий подраздел).

Сборка билда

Если билд был аккуратно спланирован в соответствии с описанием, данным в предыдущем пункте, можно сразу перейти к его сборке. Это делается путем подбора правильных версий подсистем реализации и компонентов, последующей их компиляции и сборки в билд. Отметим, что может потребоваться последовательная компиляция сперва модулей нижнего уровня иерархии, а затем верхнего. Это может быть вызвано наличием зависимостей компиляции модулей верхних уровней от нижних.

Получившийся билд становится затем объектом тестов интеграции, а затем, если билд пройдет их и если это последний билд в итерации, — системного тестирования (см. главу 11).

Деятельность: Реализация подсистемы

Цель реализации подсистемы в том, чтобы обеспечить исполнение подсистемой ее ролей в каждом из билдов, как ей и положено по плану сборки. Это означает, что следует добиться, чтобы в билде были правильно реализованы требования (то есть сценарии и/или варианты использования). Для этого подсистемы должны быть правильно реализованы компонентами или (рекурсивно) другими подсистемами, входящими в них (рис. 10.20).

Работа с содержимым подсистем

Подсистема выполняет свою задачу в том случае, если требования к ней реализуются в текущем билде. Это зависит от того, насколько корректно подсистема реализуется своими компонентами.

Несмотря на то, что архитектор описывал содержимое подсистемы (то есть входящие в нее компоненты), описание нуждается в уточнениях, которые при реализации системы делает инженер по компонентам. Уточняемыми являются следующие моменты:

- Каждый класс в соответствующей подсистеме реализации, необходимый для текущего билда, должен быть реализован в виде компонентов подсистемы реализации (рис. 10.21).

Отметим, что если подсистемы реализации содержат (рекурсивно) другие подсистемы, которые необходимы в текущем билде, мы действуем аналогично: каждая из этих внутренних подсистем проектирования должна быть реализована соответствующей подсистемой реализации внутри главной подсистемы реализации.

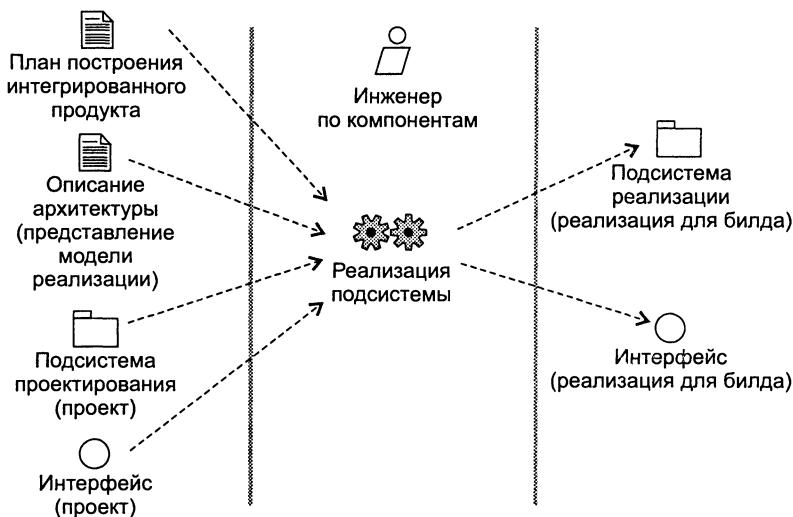


Рис. 10.20. Исходные данные и результаты реализации подсистемы

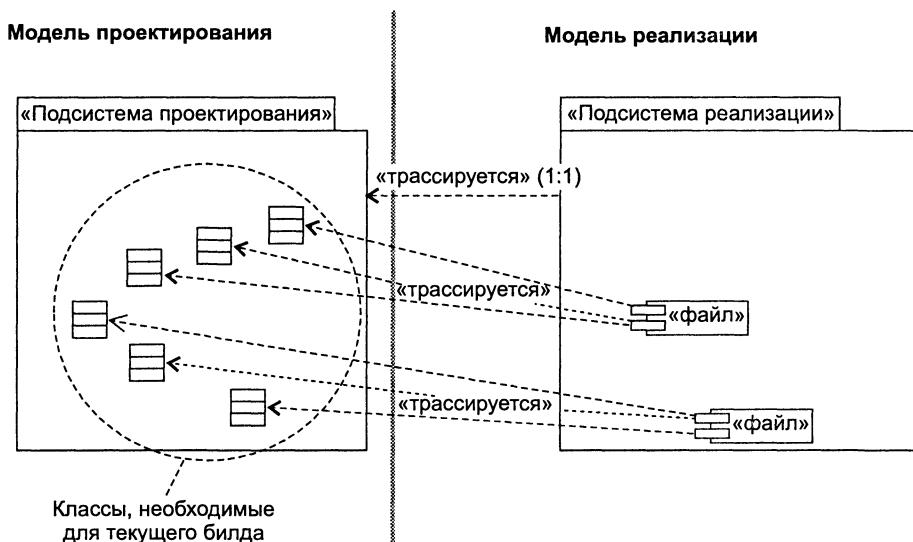


Рис. 10.21. Классы проектирования, необходимые для билда, реализуются в виде компонентов

- Каждый интерфейс, предоставляемый соответствующей подсистемой проектирования, который необходим в текущем билде, должен также предоставляться подсистемой реализации. Поэтому подсистема реализации должна содержать компоненты или подсистемы реализации (рекурсивно), представляющие этот интерфейс (рис. 10.22).

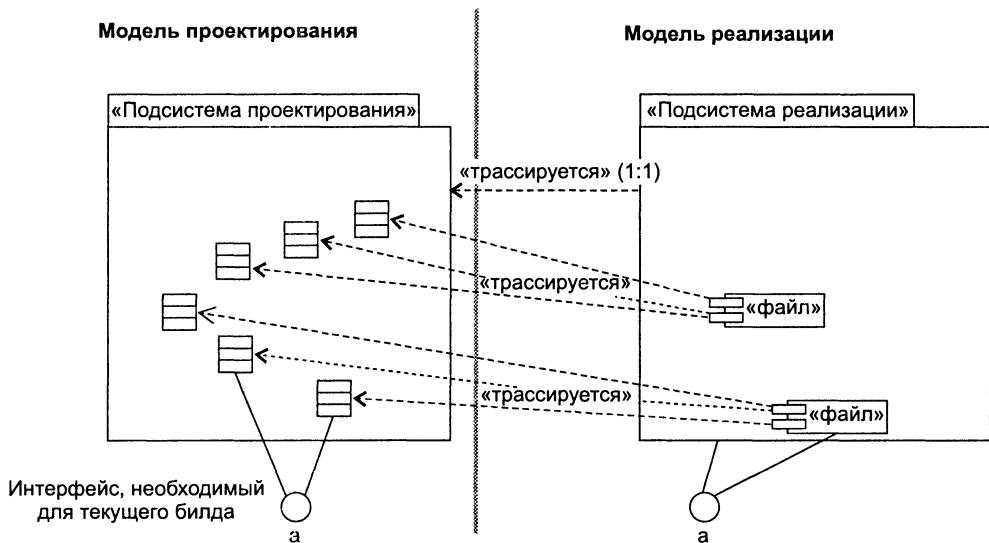


Рис. 10.22. Интерфейс, необходимый для билда (α), реализуется в виде подсистемы реализации

Пример. Подсистема, предоставляющая интерфейс. Подсистема Управление счетами покупателя должна в текущем билде предоставлять интерфейс Счет. Инженер по компонентам, который отвечает за подсистему, приходит к решению ввести компонент Обработка Счетов, в котором реализован этот интерфейс (рис. 10.23).

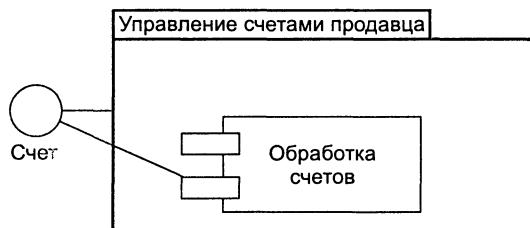


Рис. 10.23. Компонент Обработка счетов предоставляет интерфейс Счет

После этого инженер по компонентам может начинать реализовывать то, что необходимо для компонентов подсистемы (как описано в подразделе «Деятельность: Реализация класса») и тестирования модуля (как описано в подразделе «Деятельность: Выполнение тестирования модулей»).

Получившаяся подсистема затем передается для сборки системному интегратору (как описано в подразделе «Деятельность: Сборка системы»).

Деятельность: Реализация класса

Цель реализации класса состоит в том, чтобы реализовать класс в виде файлового компонента. В это понятие входит (рис. 10.24):

- Описание файлового компонента, содержащего исходный текст программы.
- Генерация исходного текста программы для класса проектирования и отношений, в которых он участвует.
- Реализация операций класса проектирования в понятиях методов.
- Проверка того, что компонент предоставляет такой же интерфейс, как класс проектирования.

Хотя мы и не описываем этого, подобная деятельность также включает в себя различные аспекты поддержки реализуемого класса, такие как исправление дефектов, выявляемых при тестировании класса.

Описание файловых компонентов

Исходный текст программы, реализующей класс проектирования, содержится в файловом компоненте. Поэтому мы должны описать файловый компонент и определить область его видимости. Это обычное дело при помещении нескольких классов проектирования в один файловый компонент. Напомним, что вид описания файловых компонентов будет зависеть от подхода к модульности файлов и правилами используемого языка программирования (см. подраздел «Артефакт: Компонент»). Так, например, при использовании Java мы создаем по одному файловому компоненту с расширением «.java» на каждый реализуемый класс. Вообще же выбираемые файловые компоненты должны обеспечивать компиляцию, установку и обслуживание системы.

Генерация кода для класса проектирования

В ходе проектирования множество деталей, относящихся к классу проектирования и его отношениям, описывается с использованием языка программирования. Это позволяет при реализации класса прямо генерировать куски исходного текста программ. В частности, так можно создавать код для операций и атрибутов класса, а также отношений, в которых этот класс участвует. Однако генерируются только сигнатуры операций, сами операции следует реализовывать вручную (см. следующий подраздел).

Отметим также, что генерация кода для ассоциаций и агрегаций — дело достаточно тонкое, и то, как она будет осуществлена, в значительной степени зависит от используемого языка программирования. Так, например, односторонние ассоциации обычно реализуются «ссылками» на объекты. Эти ссылки принимают вид атрибутов объектов, которым подобная ссылка необходима, причем имя атрибута соответствует имени роли противоположного полюса ассоциации. Множественность противоположного полюса ассоциации будет отражена в том, что типом атрибута (или ссылки) будет простой указатель (если множественность — это «один или менее») или коллекция указателей (если множественность — это «больше одного») [60].

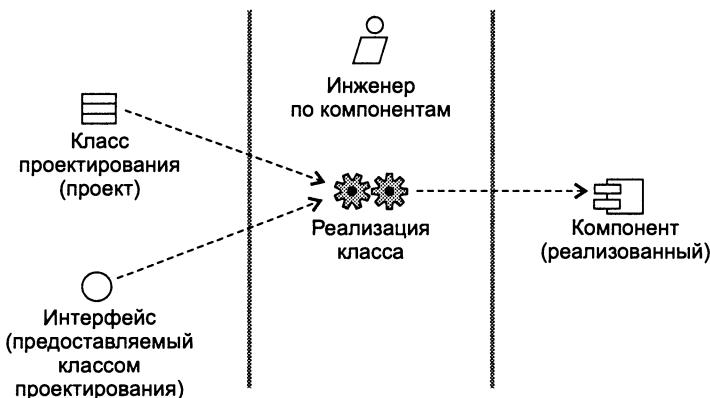


Рис. 10.24. Исходные данные и реализация класса

Реализация операций

Каждая операция, определенная в классе проектирования, может быть реализована, если, конечно, она не является «виртуальной» (или «абстрактной») и не реализуется в порожденных (то есть дочерних) классах. Для описания реализации операций мы используем термин *методы* [63]. Примерами методов в реальных файловых компонентах являются *методы Java* [4], *методы Visual Basic* [54] и *функции-члены* в C++ [62].

Реализация операций включает в себя выбор подходящего алгоритма и необходимых структур данных (например, локальных переменных метода) и кодирование действий, предусматриваемых алгоритмом. Напомним, что метод в ходе проектирования соответствующего класса проектирования может быть описан на естественном языке или с использованием псевдокода (хотя это и редко делается, так как ведет к потерям времени; см. подраздел «Описание методов»). Однако любые «методы проектирования», разумеется, послужат исходными данными для следующего шага работ. Отметим также, что вклад в способ реализации операций могут внести любые состояния, описанные в классе проектирования, поскольку они определяют поведение операций при приеме ими сообщений.

Создание компонентов, предоставляющих правильные интерфейсы

Полученные компоненты должны предоставлять те же интерфейсы, что и классы проектирования, которые они реализуют. Пример приведен в подразделе «Артефакт: Компонент».

Деятельность: Выполнение тестирования модулей

Цель проведения тестирования модулей состоит в том, чтобы протестировать реализованные компоненты как отдельные модули (рис. 10.25). Проводятся следующие виды тестирования:

- *Тестирование спецификации*, или тестирование «черного ящика», проверяет свойства модуля, видимые извне.
- *Тестирование структуры*, или тестирование «белого ящика», проверяет реализацию модуля.

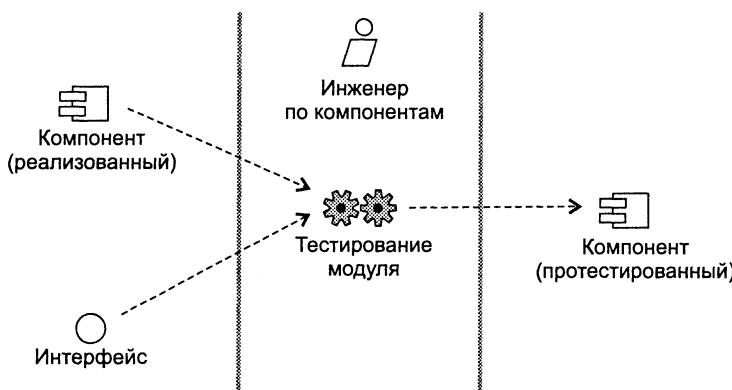


Рис. 10.25. Исходные данные и результат тестирования модуля

Отметим, что для тестирования модуля следует проделать и другие тесты, такие, как тесты производительности, использования памяти, загрузки и мощности. Кроме того, для подтверждения того, что компоненты продолжают корректно работать и после сборки, следует провести сборочные и системные тесты (см. главу 11).

Проведение тестов спецификации

Тестирование спецификации выполняется с целью проверки поведения компонентов без обсуждения того, как это поведение реализовано внутри компонентов. Поэтому тесты спецификации рассматривают результаты, которые дают компоненты при задании им некоторых исходных данных и некоторого исходного состояния. Порядок числа комбинаций всех возможных исходных данных, исходных состояний и результатов часто весьма значителен. Это делает непрактичным тестирование всех индивидуальных комбинаций. Вместо этого все исходные данные, результаты и состояния делятся на *эквивалентные классы*. Эквивалентным классом называется набор исходных данных, результатов или состояний, для которых объект демонстрирует одинаковое поведение. При тестировании компонента всеми комбинациями эквивалентных классов исходных данных, результатов и исходных состояний можно добиться почти столь же эффективного покрытия тестами, как и при тестировании всеми возможными комбинациями данных, а усилий это требует значительно меньших.

Пример. Эквивалентные классы. Состояние банковского счета имеет три эквивалентных класса: *пуст*, *отрицательный баланс* (допущен перерасход) и *положительный баланс*. Также исходные данные могут быть разделены на два эквивалентных класса: *ноль* и *положительные числа*. И наконец, результат операции попада-

ет в один из двух эквивалентных классов: *положительное списание со счета и отсутствие списания*.

Инженер по компонентам выбирает значения на основании следующих эвристик:

- Нормальные значения в допустимом диапазоне каждого из эквивалентных классов, например, списание со счета сумм \$4, \$3.14 или \$5,923.
- Значения, которые находятся на границах эквивалентных классов, такие как списание со счета \$0, минимально возможного положительного числа (например, \$0.00000001) и максимально возможного положительного числа.
- Значения, которые выходят из границ допустимых для эквивалентного класса значений, например, списание со счета суммы, большей максимально возможной или меньшей минимально возможной.
- Неверные значения, например, списание со счета отрицательной суммы (-14) или буквы (A).

При выборе тестов инженер по компонентам должен стараться покрыть все комбинации эквивалентных классов исходных данных, состояний и результатов. Так, например, попытка снять со счета \$14 при условии, что:

- На счете \$-234.13 – приведет к отсутствию списания.
- На счете \$0 – приведет к отсутствию списания.
- На счете \$13.125 – приведет к отсутствию списания.
- На счете \$15 – приведет к списанию \$14.

Чистый результат этих четырех тестов в том, что проверялись все комбинации эквивалентных классов исходного состояния (*положительный, нулевой и отрицательный баланс*) и результата (*положительное списание со счета и отсутствие списания*) с одним значением на один эквивалентный класс. Инженер по компонентам должен теперь выбрать тестовые комбинации с похожими исходными состояниями (например, \$-234.13, 0, 3 и 15) и результатами (\$0 и 14), но с другими значениями исходных данных в том же эквивалентном классе (например, \$3.14).

После этого инженер по компонентам создает несколько вариантов тестовых данных с исходными данными из других эквивалентных классов, например, попытки снятия со счета сумм \$0, 4, 3.14, 5 923, 0.00000001, 37 000 000 000 000 000 000 000 (если это максимально возможное в системе число), 37 000 000 000 000 000 000 001, -14 и A.

Проведение тестов структуры

Тестирование структуры проводится для того, чтобы подтвердить, что внутреннее поведение компонентов соответствует ожиданиям. В ходе тестирования структуры инженер по компонентам должен быть уверен, что протестирован весь код. Это означает, что каждое выражение должно быть выполнено как минимум один раз. Инженер по компонентам должен также быть уверен в том, что наиболее интересные пути исполнения кода протестированы. В эти пути входят наиболее часто используемые пути, наиболее критичные пути, наименее исследованные пути алгоритмов и другие пути исполнения, ассоциируемые с повышенным риском.

Пример. Исходный текст метода на языке Java. В листинге 10.2 показана (упрощенная) реализация метода *Снять со счета*, определенного в классе Банковский счет.

Листинг 10.2. Исходный текст упрощенной реализации метода Снять со счета, определенного в классе Банковский счет

```
1 public class Account {  
2     // In this example the Account has a balance only  
3     private Money balance = new Money (0);  
4     public Money withdraw(Money amount) {  
5         // First we must ensure that the balance is at least  
6         // as big as the amount to withdraw  
7         if (balance >= amount)  
8             // Then we check that we will not withdraw negative amount  
9             if (amount >= 0)  
10                try {  
11                    balance = balance - amount;  
12                    return amount;  
13                }  
14                catch (Exception exc)  
15                    // Deal with failures reducing the balance  
16                    //... to be defined ...  
17                }  
18            }  
19            else {return 0}  
20        }  
21    else {return 0}  
22}
```

Тестируя код, мы хотим удостоверится, что все условные операторы правильно отрабатывают как ветви true, так и false, и что весь код выполним. Так, например, мы можем протестировать следующие варианты:

- Перечисление \$50 со счета с балансом в \$100, при котором система выполняет строки 10–13.
- Перечисление \$–50 со счета с балансом в \$10, при котором система выполняет строку 21.
- Перечисление \$50 со счета с балансом в \$10, при котором система выполняет строку 19.
- Переключение на исключительную ситуацию при вычислении системой выражения $balance = balance - amount$, при котором система выполняет строки 14–17.

Рабочий процесс реализации: резюме

Основным результатом реализации является модель реализации, которая включает в себя следующие элементы.

- Подсистемы реализации, а также их зависимости, интерфейсы и содержимое.
- Компоненты, включая файлы и исполняемые компоненты, и их взаимные зависимости. Компоненты подвергаются тестированию частей.
- Архитектурное представление модели реализации, включая архитектурно значимые элементы этой модели.

Реализация также дает уточнение архитектурного представления модели распределения, в котором исполняемые компоненты распределяются по узлам.

Как будет показано в следующей главе, модель реализации рассматривается как исходные данные для последующей деятельности по тестированию. Если говорить более конкретно, то каждый отдельный билд, полученный после реализации, в ходе тестирования подвергается тестированию сборки и, возможно, системному тестированию.

11 Тестирование

Введение

В рабочем процессе тестирования мы проверяем результаты реализации путем тестирования каждого билда, включая как внутренние и промежуточные, так и финальные версии системы, передаваемые внешним агентам. Достаточно полный общий обзор процесса тестирования можно найти в [26]. Если говорить конкретнее, задачей тестирования является:

- Планирование тестов, необходимых на каждой итерации, включая тесты на целостность и системные тесты. Тесты на целостность необходимо проводить после каждого билда, в то время как системные тесты требуются только в конце итерации.
- Проектирование и реализация тестов для создания тестовых примеров, определяющих предмет тестирования, процедур тестирования, определяющих метод проведения тестирования, и, по возможности, — исполняемых тестовых компонентов для автоматизации тестирования.

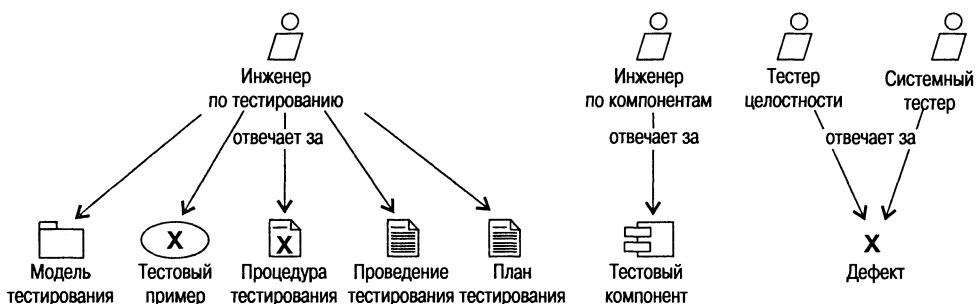


Рис. 11.1. Сотрудники и артефакты, вовлеченные в процесс тестирования

- Проведение разнообразных тестов и систематическая обработка результатов каждого теста. Билды, в которых обнаруживаются дефекты, подвергаются повторному тестированию. После этого может произойти возврат к предшествующим рабочим процессам с целью исправления серьезных ошибок.

В этой главе мы рассмотрим выполнение процесса тестирования и то, какие сотрудники и артефакты в нем участвуют (рис. 11.1). Наш подход будет аналогичен подходу, применявшемуся при рассмотрении рабочего процесса реализации.

Роль тестирования в жизненном цикле программы

Некоторое первоначальное планирование тестирования может проводиться в ходе фазы анализа и определения требований, когда на систему уже можно посмотреть. Однако в основном мы понимаем под тестированием процессы тестирования целостности и системные тесты каждого из билдов (результат реализации). Это означает, что тестирование находится в центре внимания в ходе фаз проектирования, когда тестируется исполняемый базовый уровень архитектуры, и построения, когда создается основная часть системы. В ходе фазы внедрения фокус смещается на исправление дефектов, обнаруженных в начале использования и в ходе регрессионного тестирования (рис. 11.2).

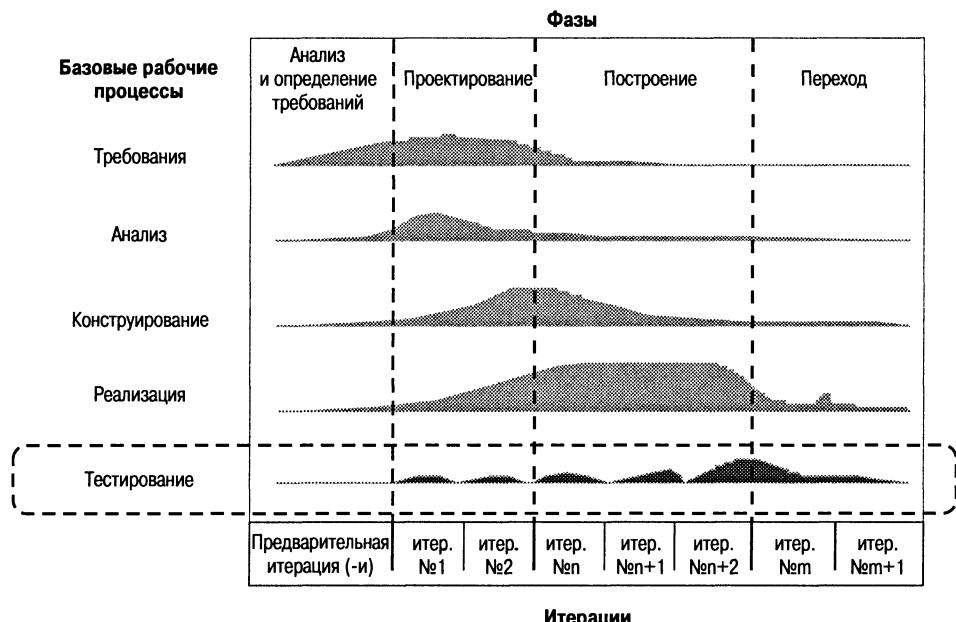


Рис. 11.2. Место тестирования в жизненном цикле системы

Поскольку процесс разработки носит итеративный характер, некоторые тестовые примеры, предназначенные для ранних билдов, могут быть применены и для регрессионного тестирования последующих билдов. Число регрессионных тестов растет от итерации к итерации, и поздние итерации требуют большого объема регрессионного тестирования. Из этого следует, что модель тестирования разумно поддерживать в течение всего жизненного цикла программного обеспечения, хотя при этом она обычно подвергается следующим изменениям:

- Удаление устаревших тестовых примеров (и соответствующих им процедур тестирования и тестовых компонентов).
- Уточнение некоторых тестовых примеров и преобразование их в примеры для регрессионного тестирования.
- Создание новых тестовых примеров для каждого последующего билда.

Артефакты

Артефакт: Модель тестирования

Модель тестирования в основном описывает то, каким образом при помощи тестов на целостность и системных тестов тестируются исполняемые компоненты (то есть билды) модели реализации. Модель тестирования может также описывать, как тестируются специфические аспекты системы (например, насколько удобен и полон интерфейс пользователя, или выполняет ли свои задачи руководство пользователя). Модель тестирования — это набор тестовых примеров, процедур тестирования и тестовых компонентов, как показано на рис. 11.3.

Артефакты модели тестирования подробно описаны в последующих пунктах. Отметим, что если модель тестирования велика, то есть если мы имеем большое число тестовых примеров, процедур тестирования и/или тестовых компонентов, то для управления размерами модели можно разбить ее на пакеты. Это более или менее тривиальное расширение модели тестирования, мы не станем обсуждать его в этой главе.

Артефакт: Тестовый пример

Тестовый пример определяет один путь тестирования системы, включающий в себя предмет тестирования вместе с исходными данными и результатом и условия тестирования (рис. 11.4). На практике предметом тестирования могут быть любые системные требования или набор требований, реализация которых возможна в ходе тестирования и не вызывает чересчур больших затрат.

Вот образцы тестовых примеров:

- Тестовый пример, определяющий порядок тестирования варианта использования или некоторого сценария варианта использования. Поскольку тестовый пример включает в себя подтверждение результатов взаимодействия между актантами и системой, то определяются пред- и постусловия, которым вариант использования должен удовлетворять, и последовательность действий,

которую вариант использования должен соблюсти. Отметим, что тестовый пример, базирующийся на варианте использования, обычно определяет тест системы как черного ящика (то есть тест поведения системы, наблюдаемого извне).

- Тестовый пример, определяющий порядок тестирования проекта реализации варианта использования или некоторого сценария этой реализации. Этот тестовый пример может включать в себя подтверждение взаимодействия между компонентами, реализующими вариант использования. Отметим, что тестовый пример, базирующийся на реализации варианта использования, обычно определяет тест системы как белого ящика (то есть тест внутреннего взаимодействия компонентов системы).

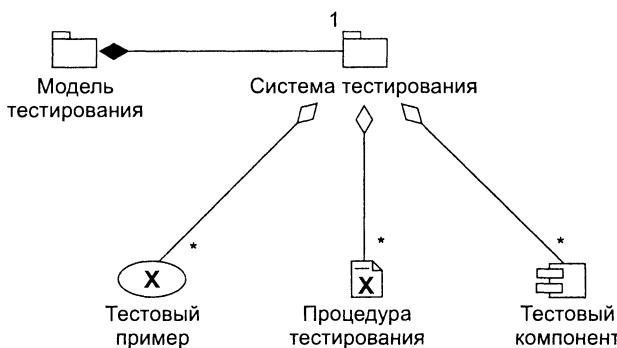


Рис. 11.3. Модель тестирования

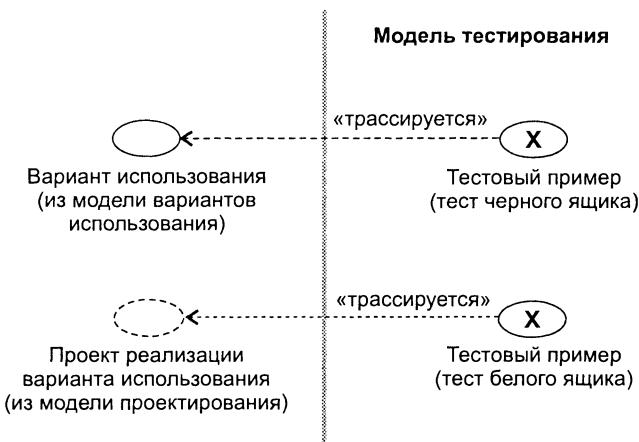


Рис. 11.4. Тестовый пример может быть порожден от варианта использования из модели вариантов использования или проекта реализации варианта использования из модели проектирования и трассирован к ним

Пример. Тестовый пример. Инженеры по тестированию разработали набор тестовых примеров для проверки варианта использования *Оплатить счет*, в котором каждый тестовый пример предназначен для проверки одного сценария варианта использования. Один из предлагаемых тестовых примеров — это оплата счета на сумму \$300, заказа на горный велосипед. Инженеры по тестированию назвали этот тестовый пример *Заплатить 300–Горный велосипед*.

Для того чтобы считаться законченным, тестовый пример должен определять исходные данные, ожидаемый результат и другие условия, относящиеся к проверке сценария варианта использования.

Исходные данные:

- Корректный заказ на горный велосипед, который был создан и послан продавцу, фирме Безумные Горцы, Inc. Цена велосипеда по прейскуранту \$300, включая доставку.
- Подтверждение заказа (ID 98765) на горный велосипед, которое было послано покупателю. Продавец подтвердил цену \$300, включая доставку.
- Счет (ID 12345), который был получен покупателем. Счет соответствует подтверждению заказа на горный велосипед. Это единственный счет, имеющийся в системе. Сумма к оплате по счету \$300, и счет находится в состоянии *Ожидавший решения*. В счете указан банковский счет 22–222–2222, на который следует перечислить деньги. Текущий баланс этого банковского счета \$963.456.00. Счет принадлежит продавцу.
- Банковский счет покупателя 11–111–1111, его баланс \$350.

Результаты:

- Счет переходит в состояние *Закрыт* (показывая, что он оплачен).
- У банковского счета покупателя 11–111–1111 баланс изменяется и составляет \$50.
- У банковского счета продавца 22–222–2222 баланс вырастает до \$963.756.00.

Условия:

- В ходе тестового примера никакие другие варианты использования (экземпляры) не имеют доступа к банковским счетам.

Отметим, что некоторые тестовые примеры могут быть очень похожи, отличаясь только одним значением в исходных данных или результатах. Это характерно для тестовых примеров, которые предназначены для тестирования различных сценариев одного варианта использования.

В этих случаях можно записать тесты в виде матрицы, в которой каждый тестовый пример будет занимать одну строку, а каждый перечень значений исходных данных и результатов — столбец. Такая матрица удобна для просмотра схожих тестовых примеров и предоставляет данные для последующего создания процедур тестирования и тестовых компонентов (см. подразделы «Артефакт: Процедура тестирования» и «Артефакт: Тестовый компонент»).

Для тестирования всей системы могут быть определены и другие варианты использования. Вот их примеры:

- *Тесты инсталляции.* Проверяют возможность установки системы на платформе пользователя и работу системы при установке.
- *Тесты конфигурации.* Проверяют корректность работы системы в различных конфигурациях, например в различных конфигурациях сети.
 - *Отрицательные тесты* используются для того, чтобы заставить систему сбить, для проверки ее устойчивости. Инженеры по тестированию определяют тестовые примеры, требующие от системы неверной работы, например в сети с неподходящей конфигурацией, недостаточной емкостью, недостаточной производительностью оборудования или в случае неверных действий пользователя («проверка на дурака»).¹
 - *Стресс-тесты* определяют проблемы, возникающие при работе системы в условиях недостатка ресурсов или в случае, когда ресурсы исчерпаны.

Большинство этих тестов могут быть разработаны после обсуждения вариантов использования системы. Мы поговорим об этом в подразделе «Деятельность: Разработка теста» данной главы.

Артефакт: Процедура тестирования

Процедура тестирования определяет, как запускать один или несколько тестовых примеров или их частей. Так, например, процедура тестирования может быть инструкцией по ручному проведению теста или спецификацией по работе со средствами автоматизации тестирования, описывающей создание исполняемых тестовых компонентов (см. следующий подраздел).

В одной процедуре тестирования может быть описан порядок запуска одного тестового примера, но чаще одна процедура тестирования применяется для нескольких тестовых примеров или нескольких процедур тестирования — для одного тестового примера (рис. 11.5).

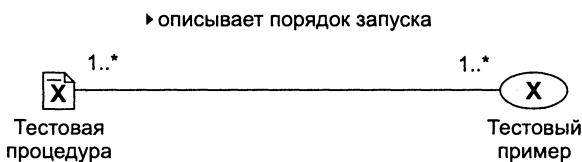


Рис. 11.5. Между тестовыми примерами и процедурами тестирования существует ассоциация «многие ко многим»

Пример. Процедура тестирования. Процедура тестирования требуется исполнителю для запуска тестового примера *Заплатить 300–Горный велосипед*, обсуждавшегося в последнем примере (в подразделе «Артефакт: Тестовый пример»). Первая часть процедуры тестирования определяется следующим образом (текст, заключенный в квадратные скобки, не обязательно включать в описание, поскольку он уже имеется в описании тестового примера):

¹ Эти тесты часто называются «проверкой на дурака».

Поддерживаемый вариант использования: *Заплатить 300-Горный велосипед.*

1. В главном окне выберите в меню **Просмотреть счета**. Откроется диалоговое окно *Запрос просмотра счетов*.
2. В поле *Состояние счета* выберите *Ожидаящие решения* и щелкните мышью на кнопке **Запрос**. Будет показано окно *Результат запроса*. Проверьте, что счет, определенный в тестовом примере (ID 12345) присутствует в окне *Результат запроса*.
3. Пометьте этот счет к оплате, дважды щелкнув на нем мышью. Будет показано окно *Счет подробно* для выбранного счета. Проверьте значения следующих полей:
 - Состояние: Ожидаящий решения.
 - Дата оплаты не заполнена.
 - ID подтверждения заказа соответствует ID, указанному в тестовом примере (ID 98765).
 - Сумма счета соответствует сумме, указанной в тестовом примере (\$300).
 - Банковский счет соответствует банковскому счету, указанному в тестовом примере (22-222-2222).
 - Поставьте флажок *Авторизовать оплату*, чтобы начать процесс оплаты по этому счету. Будет показано окно *Оплата счета*.
 - И так далее. (Далее описывается полная последовательность действий по запуску варианта использования *Оплатить счет* посредством пользовательского интерфейса путем задания системе исходных данных, а также для проверки, предполагаемый результат.)

Отметим, что эта процедура тестирования может быть также использована и для других похожих тестовых примеров с иными исходными данными и результатами (то есть значениями в скобках). Отметим также, что процедура тестирования схожа с описанием потока событий варианта использования *Оплатить счет* (см. подраздел «Структурирование описания варианта использования» главы 7), но включает в себя дополнительную информацию, в частности, исходные данные для использования в тестовом примере, способ ввода этих исходных данных посредством интерфейса пользователя и цель проверки.

Пример. Обобщенные процедуры тестирования. Когда разработчики тестов разрабатывают процедуры тестирования, они отмечают, что некоторые варианты использования (и тестовые примеры для их проверки) начинаются с проверки объектов при запуске варианта использования, например, сравнения счета с подтверждением заказа.

Разработчики тестов решили создать для проверки подобных последовательностей обобщенную процедуру тестирования под названием *Проверить бизнес-объект*. Процедура тестирования определяет, что проверяемый объект должен быть первым делом создан (путем чтения значений, которые он должен содержать, из файла и создания его в базе данных). Затем процедура тестирования предполагает

вызов активного объекта, который используется для проверки бизнес-объекта (как *Обработчик Заказов* в варианте использования *Оплатить счет*). И наконец, процедура тестирования определяет, что результат проверки должен быть сравниен с ожидаемым результатом (например, определенным в упоминавшемся файле).

Разработчики тестов также создают процедуру тестирования под названием *Проверить планирование*, которая определяет методы проверки планирования оплаты счетов, а позже — то, как *Планирование оплат* производит перемещение денег со счета на счет.

Разработчики тестов, зная, что перемещение денег со счета на счет входит в сценарии нескольких вариантов использования, разрабатывают отдельную процедуру тестирования под названием *Проверка перемещений денег между счетами*, которая определяет порядок проверки перемещений денег со счета на счет.

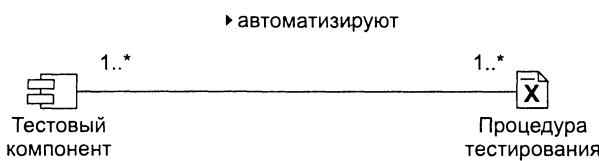


Рис. 11.6. Между тестовыми компонентами и процедурами тестирования существует ассоциация «многие ко многим»

Артефакт: Тестовый компонент

Тестовый компонент автоматизирует одну или несколько процедур тестирования или их частей (см. подраздел «Артефакт: Компонент» главы 10). Рисунок 11.6 поясняет это утверждение.

Тестовые компоненты могут разрабатываться с использованием языков сценариев, языков программирования или записываться при помощи утилит автоматизации тестирования.

Тестовые компоненты используются для тестирования компонентов модели реализации (см. главу 10) путем осуществления тестового ввода данных, управления и отслеживания процесса выполнения тестируемых компонентов и, возможно, сообщения о результатах тестирования. Тестовые компоненты также иногда называют «тестовыми драйверами», «тестовой обвязкой» [28] или «сценариями тестирования».

Отметим, что тестовые компоненты могут быть реализованы с использованием объектных технологий. Если некоторые тестовые компоненты содержат сложные внутренние взаимодействия или сложным образом взаимодействуют с обычными компонентами модели реализации, для моделирования тестовых компонентов может быть создана отдельная «модель проектирования тестов» (по аналогии с моделью проектирования), описывающая верхний уровень представления компонентов тестирования. Несмотря на то, что эта модель применяется на практике, в этой книге она рассматриваться не будет.

Артефакт: План тестирования

План тестирования описывает стратегию тестирования, выделяемые на него ресурсы и график работ. Стратегия тестирования включает в себя определение тестов, проводимых на каждой из итераций, их цели, требуемый уровень покрытия тестами кода и процент тестов, которые должны выполниться с соответствующим результатом (положительным или отрицательным).

Артефакт: Дефект

Дефект — это неправильность системы, например, симптом существования ошибки в системе или проблемы, обнаруженной на обзорном совещании. Мы будем использовать слово «дефект» для определения чего-то такого, что разработчики должны зарегистрировать в качестве симптома наличия в системе проблемы, которую следует найти и решить.

Артефакт: Оценка теста

Оценка теста — это оценка результатов тестирования, таких, как покрытие тестовыми примерами, покрытие тестами кода и статус дефектов.

Сотрудники

Сотрудник: Разработчик тестов

Разработчик тестов отвечает за целостность модели тестирования, гарантируя, что модель выполняет те задачи, для которых она создавалась. Разработчики тестов также планируют тесты, то есть определяют цели соответствующих тестов и график тестирования. Кроме того, разработчики тестов выбирают и определяют тестовые примеры и соответствующие им процедуры тестирования, а также отвечают за оценку результатов тестов интеграции и системных тестов, рис. 11.7.

Заметим, что разработчики тестов сами не проводят тестирования, концентрируясь на создании тестов и оценке их результатов. Непосредственно тестированием занимаются два других типа сотрудников, тестеры целостности и системные тестеры.

Сотрудник: Инженер по компонентам

Инженеры по компонентам отвечают за тестовые компоненты, которые автоматизируют некоторые процедуры тестирования (не все процедуры тестирования можно автоматизировать). Причина этого в том, что создание некоторых компонентов требует наличия определенных навыков программирования (то есть тех же навыков, которые требуются инженеру по компонентам в ходе рабочего процесса реализации; см. подраздел «Сотрудник: Инженер по компонентам» главы 10).

Сотрудник: Тестер целостности

Тестеры целостности отвечают за проведение тестов на целостность, необходимых для каждого билда, созданного в ходе рабочего процесса реализации (см. подраздел «Деятельность: Сборка системы» главы 10). Тестирование целостности проводится для того, чтобы проверить правильность интеграции компонентов в билд и их совместной работы. Поэтому тесты на целостность обычно порождаются из тестовых примеров, определяющих методы тестирования проектов реализации вариантов использования (см. подраздел «Артефакт: Тестовый пример» данной главы).

Результатом тестов на целостность являются дефекты, обнаруживаемые тестерами целостности.

Заметим, что тестер целостности тестирует результат интеграции (то есть билд), созданный системным интегратором в ходе рабочего процесса реализации. Следовательно, в некоторых проектах для осуществления работ этих сотрудников можно использовать одних и тех же людей. Это поможет минимизировать необходимую передачу информации.

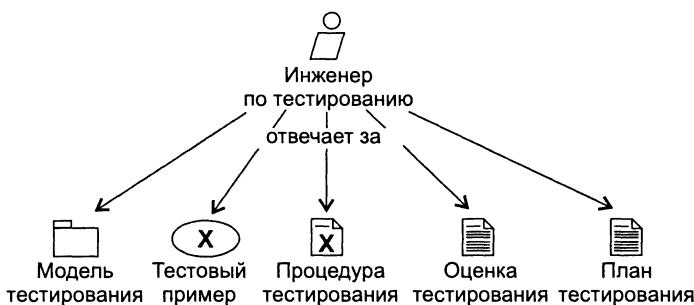


Рис. 11.7. Обязанности разработчика тестов в процессе тестирования

Сотрудник: Системный тестер

Системный тестер отвечает за осуществление системных тестов, необходимых для объявления (исполняемого) результата всей итерации (см. подраздел «Деятельность: Сборка системы» главы 10). Тестирование системы проводится в первую очередь для проверки взаимодействия между актантами и системой. Поэтому системные тесты обычно порождаются из тестовых примеров, определяющих тестирование вариантов использования. Однако для тестирования системы применяются и другие тесты (см. подраздел «Артефакт: Тестовый пример» данной главы).

Результатом системных тестов являются дефекты, обнаруживаемые системными тестерами.

Немного о сущности системных тестов. Люди, проводящие тестирование системы, не должны досконально знать внутренние подробности строения системы.

Однако они обязаны прекрасно разбираться во внешних сторонах поведения этой системы. Следовательно, некоторые системные тесты могут осуществляться другими членами команды разработчиков, например спецификаторами вариантов использования, или даже сторонними группами, например пользователями бета-версий.

Рабочий процесс

В предыдущих пунктах мы рассматривали процесс тестирования в статике. Теперь используем диаграмму деятельности для изучения его динамического поведения (рис. 11.8).

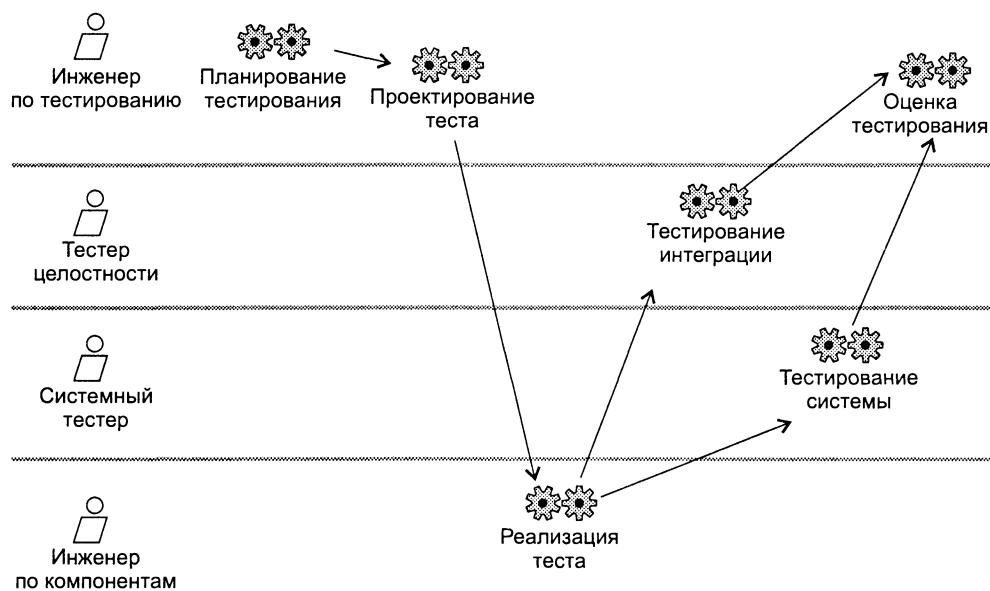


Рис. 11.8. Рабочий процесс тестирования, включая участвующих в нем сотрудников и их деятельность

Основная задача тестирования — это проведение и оценка тестов в соответствии с описанием в модели тестирования. Эта деятельность инициируется инженером по тестированию, который планирует работы по тестированию для каждой итерации. После этого инженеры по тестированию описывают необходимые тестовые примеры и соответствующие им процедуры тестирования, необходимые для проведения тестов. Затем инженеры по компонентам создают тестовые компоненты, автоматизирующие, если это возможно, некоторые из процедур тестирования. Эта работа делается для каждого билда, результата осуществления процесса реализации.

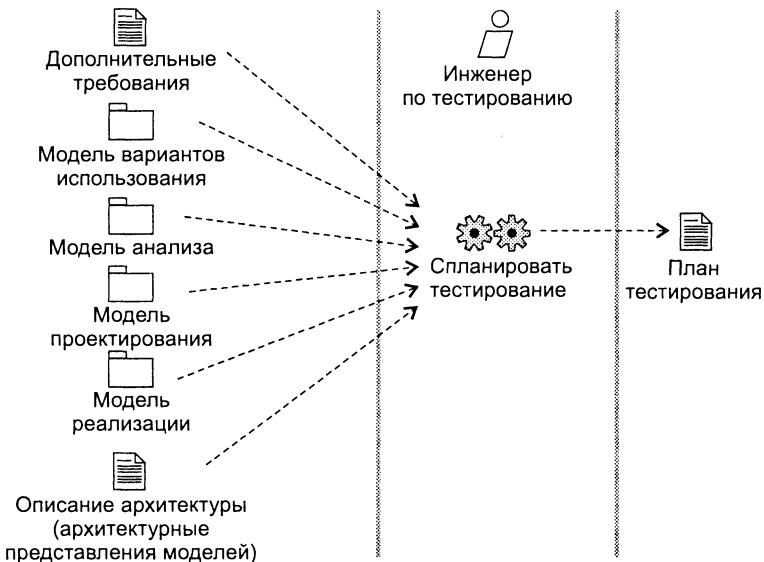


Рис. 11.9. Исходные данные и результат планирования тестирования

Используя все эти тестовые примеры, процедуры и компоненты в качестве исходных данных, тестеры целостности и системные тестеры тестируют каждый билд и отмечают все обнаруженные при этом дефекты. Затем дефекты передаются назад, на уровень рабочих процессов проектирования и реализации, и инженеру по тестированию для последовательной оценки результатов тестирования.

Деятельность: Планирование тестирования

Целью планирования тестирования (рис. 11.9) является определение плана работ по тестированию в течение итерации, включающее в себя:

- Определение стратегии тестирования.
- Оценку требований, необходимых для осуществления тестирования, например персонала и системных ресурсов.
- Составление графика работ.

Инженеры по тестированию, создавая план тестирования, используют соответствующие исходные данные. Модель вариантов использования и дополнительные требования помогают им определить подходящий порядок тестов и оценить усилия, необходимые для проведения тестирования. Разработчики тестов используют в своей работе и другие артефакты, например модель проектирования.

Разработчики тестов разрабатывают общую стратегию тестирования для потока итераций, например, тесты каких типов следует проводить, как их проводить, когда проводить и как определять успешность тестирования.

Пример. Стратегия тестирования системы для последней итерации фазы проектирования. Как минимум 75% тестов должны проводиться в автоматическом

режиме, остальные — вручную. Каждый подлежащий тестированию вариант использования должен быть протестирован в нормальном потоке событий и трех альтернативных потоках.

Критерием успешности является успешное выполнение 90% тестовых примеров и отсутствие неразрешенных дефектов высокого или среднего уровня приоритетности.

Для разработки, осуществления и оценки результатов каждого тестового примера, процедуры тестирования или тестового компонента требуются время и деньги. Никакая система не может быть протестирована полностью. Поэтому мы должны определить тестовые примеры, процедуры и компоненты, которые, в понятиях добавленной стоимости, дадут наиболее высокую прибыль на вложенный капитал [5]. Общий принцип разработки тестов состоит в том, чтобы создавать тестовые примеры и процедуры с минимальным перекрытием для тестирования наиболее важных вариантов использования и тестовые требования, ассоциирующиеся с максимальными рисками.

Деятельность: Разработка теста

Цели разработки теста (рис. 11.10):

- Определить и описать тестовые примеры для каждого билда.
- Определить и структурировать процедуры тестирования, описывающие условия осуществления тестовых примеров.

Разработка тестовых примеров для тестов на целостность

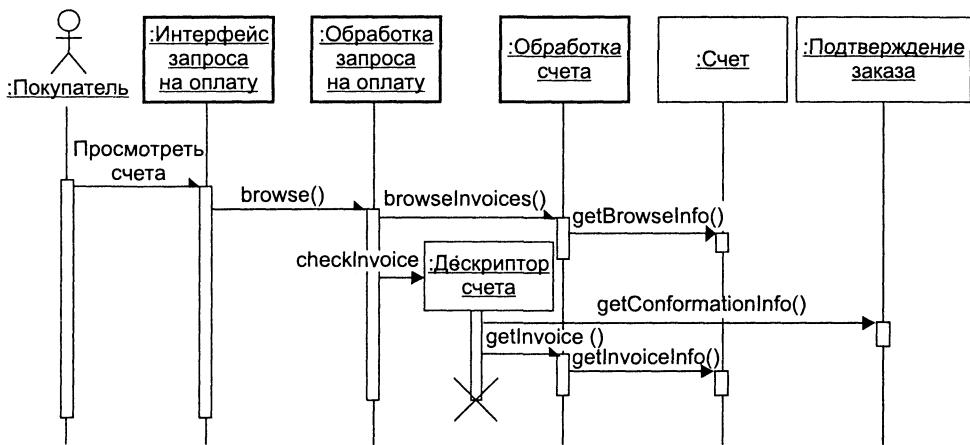
Тестовые примеры для тестирования целостности используются для подтверждения того, что компоненты правильно взаимодействуют друг с другом после сборки в билд (см. подраздел «Деятельность: Сборка системы» главы 10). Большинство тестовых примеров, предназначенных для проверки целостности, порождаются из проектов реализации вариантов использования, поскольку именно проекты реализации вариантов использования определяют правила взаимодействия классов и объектов (а значит, и компонентов) между собой.

Инженеры по тестированию должны создать набор тестовых примеров, который позволяет с минимальными затратами добиться выполнения задач, определенных в плане тестирования. Чтобы сделать это, разработчики тестов должны создать набор тестов с минимальным взаимным перекрытием, так, чтобы каждый из тестов тестировал свою часть сценария реализации нужного нам варианта использования.

При создании тестовых примеров для проверки целостности разработчики тестов используют в качестве исходных данных диаграмму деятельности реализации варианта использования. Разработчики тестов исследуют комбинации данных, вводимых актантом, исходных состояний системы и результатов, управляющих интересующими их сценариями, в которых задействованы присутствующие на диаграмме классы (и, соответственно, компоненты).



Рис. 11.10. Исходные данные и результат разработки теста

Рис. 11.11. Первая часть диаграммы последовательности проекта реализации варианта использования
Оплатить счет

Пример. Тестовый пример для проверки целостности. Разработчики тестов начинают с рассмотрения диаграммы последовательностей, которая является частью

проекта реализации варианта использования *Оплатить счет*. На рисунке 11.11 изображена первая часть этой диаграммы (см. подраздел «Описание взаимодействия объектов проектирования» главы 9).

Отметим, что на диаграмме последовательности могут присутствовать несколько различных «последовательностей». Это будет зависеть, например, от исходного состояния системы и данных, вводимых актантами. Так, рассматривая диаграмму последовательности на рис. 11.11, мы можем заметить, что в случае отсутствия счетов в системе многие сообщения не посылаются.

Тестовый пример, порождаемый из диаграммы последовательности, например такой, как на рис. 11.11, должен описывать способ тестирования конкретной интересующей нас последовательности, изображенной на диаграмме. Это делается путем определения требуемого исходного состояния системы, данных, вводимых актантами и других факторов, выводящих процессы, происходящие в системе, на нужную нам последовательность.

Впоследствии, после выполнения соответствующего теста на целостность, мы определяем реально происходящие в системе взаимодействия между объектами (например, с помощью отладочной печати или пошагового выполнения программы). Затем мы сравниваем эти истинные взаимодействия с диаграммой взаимодействий. Это позволяет нам сделать то же самое — обнаружить дефекты.

Разработка системных тестовых примеров

Системные тесты используются для проверки правильности работы системы в целом. Каждый системный тест в первую очередь проверяет, как комбинации вариантов использования будут работать в различных условиях. В перечень этих условий входят различные конфигурации аппаратных средств (процессоров, оперативной памяти, жестких дисков и т. д.), различная степень загруженности системы, различное число пользователей и различные размеры баз данных. При разработке системных тестовых примеров разработчики тестов должны установить приоритеты в комбинациях вариантов использования, которые:

- требуют параллельной работы;
- могут работать параллельно;
- зависят друг от друга в случае параллельной работы;
- включают в себя множественные процессы;
- часто используют системные ресурсы — процессы, процессор, базы данных, программы передачи данных, причем нередко совместно или непредсказуемым образом.

Многие системные тестовые примеры могут быть найдены при рассмотрении вариантов использования, в особенности их потоков событий и дополнительных требований (например, требований к производительности).

Разработка тестовых примеров для регрессионного тестирования

Некоторые тестовые примеры из ранних билдов могут быть использованы в качестве тестовых примеров для регрессионного тестирования в последующих. Одна-

ко для регрессионного тестирования подходят не все тестовые примеры. Чтобы удовлетворять условиям регрессионного тестирования и постоянно способствовать повышению качества системы, тестовый пример должен быть достаточно гибким, приспосабливаться ко всем изменениям тестируемой системы. Гибкость, требуемая от тестовых примеров для регрессионного тестирования, означает дополнительные затраты на разработку, а значит, нам следует быть осторожными и делать тестовые примеры пригодными для регрессионного тестирования только в том случае, если они того заслуживают.

Определение и структурирование процедур тестирования

Разработчики тестов могут отрабатывать один тестовый пример за другим, сопоставляя каждому из них процедуру тестирования. Мы по возможности должны стараться повторно использовать существующие процедуры тестирования, возможно, модифицируя их так, чтобы они могли описывать применение новых или измененных тестовых примеров. Разработчики тестов, в свою очередь, должны создавать процедуры тестирования так, чтобы их можно было повторно использовать с другими тестовыми примерами. Это позволяет разработчикам тестов использовать для работы с множеством тестовых примеров небольшой набор процедур тестирования.

Большинство тестовых примеров проверяют несколько классов в нескольких сервисных подсистемах (поскольку тестовые примеры базируются на вариантах использования или реализациях вариантов использования), но каждая процедура тестирования должна по возможности определять, как тестировать классы одной сервисной подсистемы. Однако некоторые процедуры тестирования могут целиком определять тестирование одной из сервисных подсистем. Каждый тестовый пример должен поэтому поддерживаться несколькими процедурами тестирования, по одной на каждую сервисную подсистему, к которой применяется тестовый пример. Распределение тестовых процедур по сервисным подсистемам повышает удобство работы с ними. При изменении в сервисной подсистеме эффект от этого изменения, касающийся процедур тестирования, будет ограничен той процедурой тестирования, которая используется для проверки этой сервисной подсистемы, все же прочие процедуры тестирования останутся неизменными.

Пример. Тестовая процедура. Сервисная подсистема *Банковские счета* представляет функциональность для перечисления денег со счета на счет. Эта функциональность включена в несколько реализаций вариантов использования, например *Оплатить счет* и *Перечислить деньги со счета на счет*. Для тестирования перечисления денег со счета на счет будет определена процедура тестирования под названием *Проверить Перечисления*. Процедура, определенная под этим именем, будет использовать в качестве входных параметров номера счетов и сумму перечисления и проверять результат перевода путем запроса баланса счетов до перевода и после него.

Разработчики тестов создали 8 тестовых примеров для варианта использования *Оплатить счет* и 14 тестовых примеров для варианта использования *Перечислить деньги со счета на счет*. Тестовая процедура *Проверить Перечисления* определяет, как следует выполнять все эти тестовые примеры (или их часть).

Деятельность: Реализация теста

Цель реализации тестов состоит в том, чтобы автоматизировать процедуры тестирования путем создания, если это возможно, тестовых компонентов (не все тестовые процедуры можно автоматизировать) (рис. 11.12).

Тестовые компоненты создаются из процедур тестирования:

- При использовании средств автоматизированного тестирования мы производим или определяем действия в соответствии с описанием в тестовой процедуре. Эти действия записываются и преобразуются в тестовые компоненты, например, тестовые программы на Visual Basic.
- При непосредственном программировании компонентов тестирования мы используем процедуры тестирования в качестве первичной спецификации для программирования. Отметим, что программирование может потребовать наличия людей, имеющих в этом деле хороший навык.

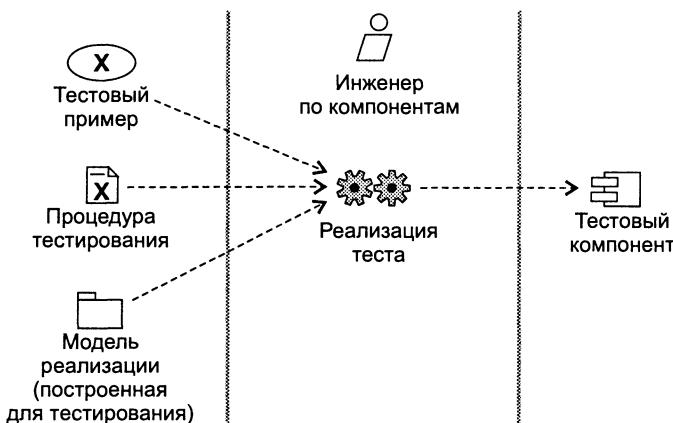


Рис. 11.12. Исходные данные и результаты реализации теста

Тестовые компоненты часто используют для своей работы большие объемы исходных данных и генерируют значительные объемы результатов тестирования. При их использовании следует позаботиться о простой и понятной визуализации этих данных, позволяющей убедиться в том, что исходные данные заданы верно, и легко интерпретировать результаты. Для этой цели используются электронные таблицы и приложения, работающие с базами данных.

Деятельность: Проведение тестирования целостности

В этой деятельности каждый созданный в ходе итерации билд (см. подраздел «Деятельность: Сборка системы» главы 10) подвергается тестам на целостность (см. подраздел «Разработка тестовых примеров для тестов на целостность» данной главы). Нас интересуют результаты тестов (рис. 11.13).

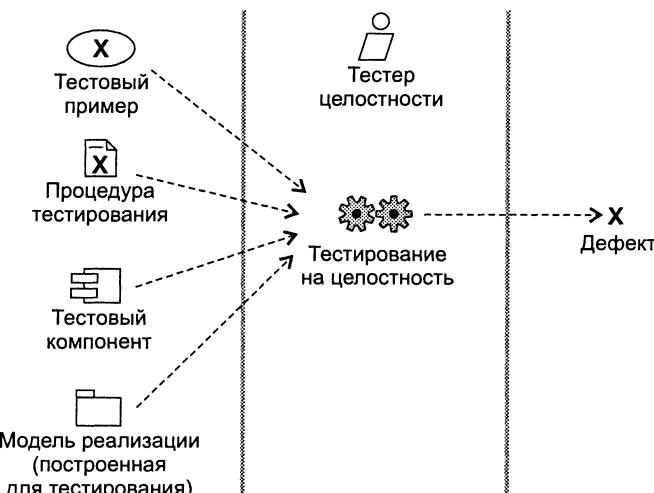


Рис. 11.13. Исходные данные и результат тестирования целостности

Тестирование целостности проходит следующим образом:

1. Проводится тестирование целостности билда путем ручного осуществления процедур тестирования для каждого из тестовых примеров или запуска тестовых компонентов, автоматически выполняющих процедуры тестирования.
2. Результаты тестирования сравниваются с ожидаемыми результатами и проводится исследование тех результатов, которые отличаются от ожидаемых.
3. Сведения о дефектах передаются тому инженеру по компонентам, который отвечает за компоненты, содержащие ошибку.
4. Сведения о дефектах передаются разработчикам тестов для их использования в составлении общего отчета о тестировании (как описано в подразделе «Деятельность: Оценка результатов тестирования» данной главы).

Деятельность: Проведение тестирования системы

Цель тестирования системы состоит в проведении на каждой итерации системных тестов. Нас интересуют результаты тестирования (рис. 11.14).

Тестирование системы можно начинать после того, как система будет удовлетворять параметрам по тестированию целостности, заданным в плане тестирования на текущую итерацию (например, корректное выполнение 95% тестов на целостность).

Тестирование системы проводится аналогично тестированию интеграции (см. подраздел «Деятельность: Проведение тестирования целостности»).

Деятельность: Оценка результатов тестирования

Цель оценки тестирования в том, чтобы оценить результаты тестирования для данной итерации (рис. 11.15).



Рис. 11.14. Исходные данные и результат тестирования системы

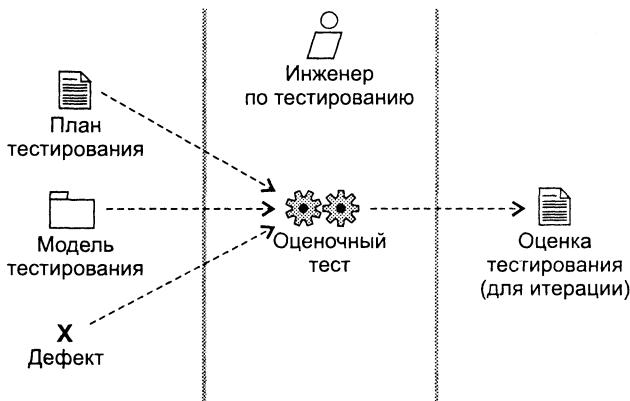


Рис. 11.15. Исходные данные и результат оценки тестирования

Разработчики тестов оценивают результаты тестирования путем сравнения результатов тестирования с целями, указанными в плане тестирования. Они должны найти способ измерения и сравнения уровней качества программного обеспечения и определить необходимые затраты на тестирование. Инженеры по тестированию обычно пользуются для оценки двумя шкалами:

- *Полнота тестирования*, определяемая уровнем покрытия тестовыми примерами и уровнем покрытия тестируемых компонентов. Она показывает, какой процент тестов выполнен и какой процент кода протестирован.
- *Надежность*, основанная на анализе тенденций обнаружения дефектов и тенденций выполнения тестов с ожидаемыми результатами.

Для определения надежности системы разработчик тестов создает диаграммы тенденций обнаружения дефектов, иллюстрирующие распределение отдельных видов дефектов (например, новых или критических) от времени. По тестам также могут быть созданы диаграммы тенденций, отражающие, например, зависимость доли успешно выполнившихся тестов (то есть тестов, результаты которых соответствуют ожидаемым) от времени.

Тенденции дефектов часто следуют образцам, повторяющимся из проекта в проект. Так, например, число новых дефектов, обнаруживаемых при тестировании системы, обычно быстро растет в начале тестирования, через некоторое время выходит на постоянный уровень и на конец начинает медленно спадать. Поэтому, сравнивая текущие тенденции с тенденциями из предыдущих проектов, можно предсказать, какие усилия необходимы для достижения приемлемого уровня качества.

Основываясь на результатах анализа тенденций дефектов, разработчики тестов могут предпринимать определенные действия, а именно:

- Проведение дополнительных тестов с целью локализации дополнительных дефектов, если оценка надежности свидетельствует о том, что система еще не готова.
- Смягчение критериев оценки результатов тестирования, если уровень качественности для текущей итерации был установлен слишком высоким.
- Выделение частей системы, которые, похоже, имеют надлежащее качество, и предоставление их в качестве результатов текущей итерации. Те части системы, которые не соответствуют критериям качества, должны быть пересмотрены и протестираны снова.

Разработчики тестов документируют полноту тестирования, надежность и производимые действия в описании оценки тестирования.

Тестирование: резюме

Основным результатом тестирования является модель тестирования, которая включает в себя следующие элементы.

- Тестовые примеры, определяющие предмет тестирования.
- Процедуры тестирования, определяющие условия осуществления тестовых примеров.
- Тестовые компоненты, автоматизирующие процедуры тестирования.

Также результатами тестирования являются план тестирования, оценка проведенных тестов и дефекты, которые передаются на предыдущие рабочие процессы, такие, как проектирование и реализация.

ЧАСТЬ 3

Итеративная и инкрементная разработка

Программные системы в течение своего срока жизни проходят через несколько циклов разработки. Результатом каждого из циклов является новый выпуск продукта, передаваемый пользователям и заказчикам, причем первый из циклов вполне может оказаться самым сложным. Он закладывает основание, архитектуру системы, при его создании мы вторгаемся в новую область, которая может содержать существенные риски. Цикл разработки протекает по-разному. Его наполнение зависит от того, на каком этапе жизненного цикла находится система. Если во время подготовки поздних выпусков в архитектуру системы вносятся серьезные изменения, это вызывает значительную нагрузку на ранние фазы разработки. Однако если базовая архитектура сделана расширяемой, для большинства поздних выпусков новый проект просто строится на базе уже созданного, то есть последующие выпуски продукта создаются на базе предыдущих.

Множество людей поддержали идею того, что решать проблемы в ходе каждого цикла разработки лучше раньше, чем позже. Они применили понятие *итерации* к последовательности решения проблем на фазах анализа и планирования требований и проектирования, а также к каждой серии билдов фазы построения.

Никто не пришлет нам риски в красивом пакете с идентификационной карточкой, аккуратно вложенной под розовую ленточку. Они должны быть определены, разграничены, отслежены и снижены — и лучше всего будет разобраться сначала с наиболее существенными рисками. Соответственно, порядок добавления итераций должен быть продуман так, чтобы наиболее серьезные проблемы решались в первую очередь. Короче говоря, *делайте сперва то, что труднее*.

В части 2 мы описывали каждый рабочий процесс по отдельности. Так, например, рис. 6.2 показывает, сколько на каждой из фаз уделяется внимание процессу

определения требований. Соответственно, рис. 8.2 показывает то же самое для анализа, рис. 9.2 — для проектирования, рис. 10.2 — для реализации, а рис. 11.2 — для тестирования.

В этой части мы покажем, каким образом, в зависимости от положения итерации в жизненном цикле системы, комбинируются рабочие процессы. Сначала, в главе 12, мы рассмотрим общие стороны всех фаз, то есть работы, которые производятся на каждой фазе, как-то: планирование итераций, определение критериев выполнения для каждой из них, составление списка рисков, расстановка приоритетов вариантов использования и оценка итераций. В последующих главах мы сосредоточимся на каждой из фаз по отдельности.

В фазе анализа и определения требований (глава 13) наша деятельность сосредоточится на первом рабочем процессе, определении требований, с небольшими добавками второго и третьего рабочих процессов (анализа и проектирования). В этой фазе два последних рабочих процесса, реализация и тестирование, редко вступают в игру.

В фазе проектирования (глава 14), когда деятельность, связанная с определением требований, все еще далека от завершения, проявляется большая активность во втором и третьем рабочих процессах, анализе и проектировании. Эта активность ложится в основание создающейся архитектуры. Для создания исполняемого основания архитектуры необходима также некоторая деятельность и во время двух последних рабочих процессов, реализации и тестирования.

В фазе построения (глава 15) рабочий процесс определения требований заканчивается, активность анализа снижается, а основной объем работ приходится на три последних рабочих процесса.

В фазе внедрения (глава 16) соотношение рабочих процессов зависит от результатов использования бета-версии. Так, если в бета-версии обнаруживаются дефекты реализации, то возникает активность во вновь запускаемых рабочих процессах реализации и тестирования.

В последней главе, главе 17, мы возвращаемся к основной теме книги. В одной главе мы рассмотрим, как множество отдельных прядей — рабочие процессы, фазы, итерации, — сплетаются в хорошо спроектированный процесс разработки определяемого задачей программного обеспечения. В эту главу также включено несколько параграфов о том, как управлять этими отношениями и как организация может перейти от того, что она делает сейчас, к Унифицированному процессу.

12 Обобщенный рабочий процесс итерации

В этой главе мы вернемся к идеи обобщенной итерации, обсуждавшейся в главе 5. Задача этой главы — выделить основные образцы, которые характеризуют все итерации в том их разнообразии, которое проходит перед нашими глазами на протяжении четырех фаз.

Мы используем эти обобщенные образцы в качестве основы для построения на каждой из фаз конкретных итераций, наполнение которых будет меняться, чтобы поддерживалось соответствие специальным задачам фаз разработки (рис. 12.1 показывает, как структурирована часть 3: обобщенная итерация рассматривается в главе 12, ее специализации для различных фаз — в главах 13–16).



Рис. 12.1. Рабочий процесс обобщенной итерации используется для описания конкретных итераций каждой фазы

Рабочий процесс обобщенной итерации включает в себя пять базовых рабочих процессов: определение требований, анализ, проектирование, реализацию и тестирование. Также в него входит планирование, которое предшествует рабочим процессам, и оценка, которая следует за ними (в главах 6–11 рассмотрен по от-

дельности каждый из базовых рабочих процессов). В этой главе мы сосредоточимся на планировании, оценке и других видах деятельности, общих для всех рабочих процессов.

Планирование необходимо осуществлять в течение всего цикла разработки. Однако до того, как мы сможем планировать, мы должны знать, что делать. Пять базовых рабочих процессов будут для нас отправной точкой. Другой ключевой аспект планирования — это управление рисками, то есть их определение и последующее снижение посредством реализации соответствующего набора вариантов использования. Разумеется, никакой план не может считаться полным без оценки требуемых ресурсов, наконец, следует оценить выполнение каждой итерации и фазы после ее завершения.

Необходимость баланса

В каждый момент жизненного цикла проекта по созданию программного обеспечения в нем протекают различные последовательности действий. Мы работаем с новыми функциями, строим архитектуру, получаем отзывы пользователей, снижаем риски, планируем будущее развитие и т. д. В любой момент мы должны балансировать и синхронизировать эти различные последовательности действий во всей их полноте.

Разработчики делят работу, чрезмерно сложную для цельного восприятия, на меньшие, более удобные для понимания части. В ходе жизненного цикла разработки они разбивают работу на фазы, а внутри фаз — на итерации. Мы говорили об этом в части 1.

В ходе каждой итерации проект устремлен к достижению баланса между работами, производимыми в ходе этой итерации. Это означает, что в ходе каждой итерации мы должны делать правильные вещи. Какие правильные вещи мы должны делать и как они изменяются в зависимости от положения итерации в жизненном цикле? Задача проекта — выбирать для выполнения в ходе каждой последовательности действий правильные вещи. Для определения баланса последовательностей действий часто бывает важно убедиться, что они имеют сравнимую важность, а значит, могут быть эффективно расставлены по приоритетам и синхронизированы. Невозможность достижения такого рода баланса в сочетании с энергичным выполнением приводит к необходимости отката назад на много итеративных инкрементных жизненных циклов.

На ранних итерациях мы занимаемся критическими рисками, ключевыми вариантами использования, проблемами архитектуры, выбором среды разработки. Вся эта деятельность ориентирована на исследование. В то же время на поздних итерациях мы производим деятельность, ориентированную на разработку, — реализацию и тестирование, оценку производительности и загрузку системы на узлы. Соотнесение всех этих видов деятельности между собой — дело, требующее большой аккуратности. Необходимость такой аккуратности делает разработку программного обеспечения особенно трудной.

Осознание этих последовательностей действий и их балансировка и есть то, что мы делаем на каждой итерации. В Унифицированном процессе некоторые из этих

последовательностей действий определяются и описываются в базовых рабочих процессах. Существуют также и другие последовательности, которые невозможно определить формально, однако в большинстве случаев они могут быть рассмотрены так же, как рабочие процессы. В их число входят:

- Взаимодействие с заказчиками для получения новых требований.
- Подготовка предложения для клиента.
- Изучение контекста системы для создания бизнес-модели.
- Планирование и управление проектом.
- Выбор и управление средой разработки — процессом и средствами.
- Управление рисками.
- Загрузка продукта на рабочие места заказчика.
- Реагирование на отклики пользователей.

Фазы, на которые предварительно разбивается работа

Первый этап разделения процесса разработки программного обеспечения на части — это выделение во временных рамках его осуществления четырех фаз: анализа и планирования требований, проектирования, построения и внедрения. Каждая фаза затем делится на одну или более итераций. В этой главе описываются общие черты этих фаз и итераций. Следующие четыре главы посвящены детальному описанию каждой из фаз.

Фаза анализа и планирования требований определяет выполнимость

Главная цель этой фазы — составить бизнес-план, который ляжет в основу проекта. Этот план будет дополняться в фазе проектирования, когда нам будет доступна дополнительная информация. В фазе анализа и планирования требований полное изучение предполагаемой системы остается делом будущего. Мы проследили лишь небольшой процент вариантов использования — столько, сколько необходимо для поддержки исходного бизнес-плана. Для того, чтобы создать этот бизнес-план, мы должны предпринять следующие четыре шага:

1. Ограничить предполагаемую систему, то есть определить границы предполагаемой системы и начать определение интерфейсов с системами, находящимися вне этих границ.
2. Определить или кратко описать предполагаемую архитектуру системы, особенно те ее части, которые являются новыми, рискованными или трудными. Создаваемое на этом этапе описание архитектуры редко сопровождается созданием действующего прототипа. Описание архитектуры включает в себя первые наброски представлений моделей. Наша цель — определиться, в состоянии ли мы создать на следующей фазе стабильную архитектуру системы. На этой фазе мы не строим архитектуру, мы просто убеждаемся в том, что мы в состоянии ее

- построить. Собственно построение архитектуры — это основная задача фазы проектирования.
3. Определить наиболее опасные риски, способные повлиять на нашу способность построить систему и определить, в состоянии ли мы найти способы их снижения, возможно, на последующих фазах. На этой фазе мы обсуждаем исключительно риски, влияющие на осуществимость проекта, иными словами, риски, угрожающие успешной разработке системы. Риски, не являющиеся критическими, которые мы также можем обнаружить при определении рисков, помещаются в список рисков для последующего детального рассмотрения в ходе следующей фазы.
 4. Продемонстрировать потенциальным пользователям или заказчикам, что предлагаемая система в состоянии решить их проблемы или помочь в их бизнесе путем построения концептуального прототипа. На фазе анализа и планирования требований мы можем построить прототип для демонстрации решения проблемы потенциальным заказчикам и пользователям. Прототип позволяет продемонстрировать основные идеи новой системы. Основное внимание уделяется интерфейсам пользователя и/или новым интересным алгоритмам. Этот прототип в основном предназначен для привлечения внимания, например для демонстрации возможного решения, и не предназначен для преобразования в конечный продукт. Обычно в ходе дальнейшей разработки этот прототип идет в мусорную корзину. В противоположность ему, архитектурный прототип, разработанный в ходе фазы анализа и определения требований, пригоден для доработки на следующей фазе и получает дальнейшее развитие.

Мы продолжаем эту деятельность до тех пор, пока не поймем, будет ли экономически выгодна разработка этого продукта. Мы должны увидеть, что система может обеспечить, хотя бы очень приблизительно, доход или другую соизмеримую ценность при вложении в нее средств, необходимых для ее создания. Другими словами, мы должны сделать первоначальный приблизительный вариант бизнес-плана. Мы уточним его в ходе следующей фазы, проектирования.

Существует тенденция минимизации времени, усилий и затрат до тех пор, пока мы не выясним, что система на самом деле осуществима. В случае новых больших систем в малоисследованной области определение факта осуществимости может потребовать существенного времени и усилий. До наступления этого события может пройти несколько итераций. Для хорошо известных систем в изученной области или создания новой версии существующей системы риск и неизвестность минимальны, и первую фазу можно начать и закончить за несколько дней.

Фаза проектирования обеспечивает возможность выполнения

Главный результат фазы проектирования — это устойчивая архитектура, в соответствии с которой система будет строиться в ходе всей ее последующей жизни. В этой фазе также производится изучение предполагаемой системы для высокоточного планирования фазы построения. Для выполнения этих двух задач — построения архитектуры и точной оценки затрат — команда разработчиков должна сделать следующее:

1. Создать базовый уровень архитектуры, который включает в себя архитектурно значимые функциональные возможности системы и особенности, важные для заинтересованных лиц. Мы говорили об этом в главе 4. Этот базовый уровень состоит из представлений моделей, описания архитектуры и исполняемой реализации. Он не только демонстрирует нашу способность построить устойчивую архитектуру, но и собственно очерчивает архитектуру.
2. Определить существенные риски, то есть риски, которые могут расстроить наши планы, изменить затраты и график последующих фаз и свести их к деятельности, затраты и время на которую мы в состоянии оценить.
3. Определить уровень качества, которого мы хотим достичь, например надежность (нормы по дефектам) и время отклика.
4. Охватить вариантами использования до 80% функциональных требований. Этого будет достаточно для перехода к фазе построения. (Мы уточним, почему мы считаем достаточным именно 80%, ниже в этой главе.)
5. Создать финансовый план, определить необходимый персонал и затраты в пределах, определяемых деловой практикой.

Основную долю работ на фазах анализа и планирования требований и проектирования составляют определение требований и архитектура (рабочие процессы анализа, проектирования и реализации).

Фаза построения создает систему

Основная цель фазы построения указана на ее главной вехе — начальная работоспособность. Это указание на продукт, готовый к бета-тестированию. Данная фаза требует усилий большего количества работников в течение более долгого срока, чем любая другая. Поэтому так важно до начала этой фазы сделать все, что может понизить ее объемность. Обычно фаза построения включает в себя больше итераций, чем предшествующие ей фазы.

Во многих случаях оказывается, что построение потребовало дополнительных затрат времени из-за слабой проработки требований, анализа и проектирования. В результате разработчики были вынуждены прорубаться сквозь систему, а это значительно дольше, чем просто выполнять требования и устранять дефекты (ошибки). Одно из важных преимуществ такого подхода к проектированию программного обеспечения, который предполагает набор фаз и итеративную, инкрементную разработку, в том, что он позволяет нам сбалансировать затраты времени и ресурсов в течение жизненного цикла системы (см. подраздел «Необходимость баланса» данной главы).

Деятельность на фазе построения включает в себя:

1. Расширенное определение и описание вариантов использования и их полную реализацию.
2. Завершение анализа (к началу фазы построения может быть проанализировано больше половины вариантов использования), проектирования, реализации и тестирования (к началу фазы построения до 90% вариантов использования не проработаны).
3. Поддержание целостности архитектуры и ее модификацию при необходимости.

4. Отслеживание критических и существенных рисков, обнаруженных в ходе первых двух фаз и, в случае их материализации, смягчение вызванных ими последствий.

Фаза внедрения переносит систему в среду пользователей

Фаза внедрения обычно начинается с бета-версии. Это означает, что организация-разработчик распространяет программный продукт, имеющий первоначальную функциональность, среди представительной выборки из сообщества действительных пользователей. Использование в суровых условиях организации-пользователя — обычно более жесткое испытание для продукта, чем тестирование в кругу разработчиков.

Деятельность по переносу включает в себя:

1. Деятельность по подготовке, например подготовка места развертывания продукта.
2. Рекомендации заказчикам по модернизации среды (оборудования, операционных систем, коммуникационных протоколов и т. д.), в которой должна работать программа.
3. Создание руководств и другой документации на выпуск продукта. В фазе построения мы создавали предварительную документацию для бета-тестеров.
4. Коррекция программного обеспечения для обеспечения его работоспособности в текущих условиях пользовательской среды.
5. Коррекция дефектов, о которых сообщили бета-тестеры.
6. Модификация программного обеспечения в свете непредвиденных проблем.

Фаза внедрения заканчивается формальным выпуском продукта. Однако перед тем, как команда разработчиков покинет проект, их руководитель проводит «разбор полетов», посвященный двум задачам:

- Определению, обсуждению, оценке и сохранению на будущее «полученных уроков».
- Записи вопросов, которые следует рассмотреть в следующем выпуске или следующем поколении.

Еще раз об обобщенной итерации

Мы указали на отличия между базовыми рабочими процессами и рабочими процессами итераций. Базовые рабочие процессы — определение требований, анализ, проектирование, реализация и тестирование — были описаны в главах 6–11. В Унифицированном процессе эти базовые рабочие процессы происходят не один раз, как теоретически должно было бы быть в случае водопадной разработки. Они повторяются раз за разом в ходе рабочих процессов итераций на каждой итерации. В этих повторениях, однако, детали их различны — они определяются задачей, решаемой на данной итерации.

Основные рабочие процессы повторяются на каждой итерации

Обобщенная итерация состоит из пяти рабочих процессов — определения требований, анализа, проектирования, реализации и тестирования, и включает в себя дополнительно планирование и оценку результатов. Это можно увидеть на рис. 12.2.

В разделах «Планирование предваряет деятельность», «Риски влияют на планирование проекта», «Расстановка приоритетов вариантов использования» и «Требуемые ресурсы» мы рассмотрим планирование, которое предшествует итерации, а в разделе «Оценка итераций и фаз» — оценку результатов итерации. Затем в главах 13–16 мы детально обсудим, как протекают в каждой из фаз базовые рабочие процессы.

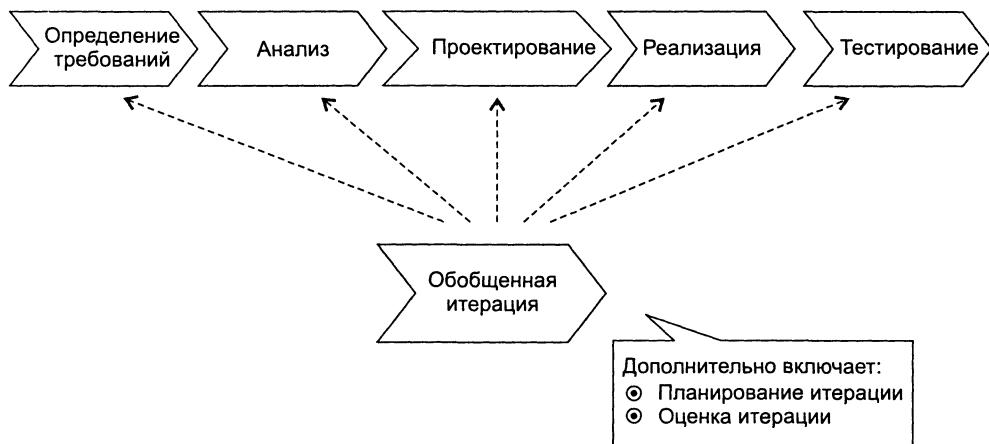


Рис. 12.2. Пять базовых рабочих процессов повторяются на каждой итерации, предваряясь планированием и завершаясь оценкой

Сотрудники участвуют в рабочих процессах

Мы время от времени говорим о сложности разработки программного обеспечения. На рис. 12.3 приведен схематичный обзор этого процесса. Даже теперь этот рисунок выглядит далеко не простым, а реальность, как вы знаете, еще более запутанна. В этой главе мы не будем детально описывать ни процесс создания каждым из сотрудников артефактов, за которые он отвечает, ни сами эти артефакты. Небольшие шестеренки на рисунке обозначают виды деятельности, а стрелки между ними отражают временные зависимости. Детальное описание этих видов деятельности можно найти в главах 6–11.

Рисунок 12.3 дает представление о том, что же происходит в ходе итерации. Наименования сотрудников перечислены по вертикали слева и справа, определяя каждую из «плавательных дорожек». Время нарастает слева направо. Базовые рабочие процессы, охватывающие сотрудников и деятельность, которую они выполняют, очерчены

ны выполнеными от руки контурами. Например, начиная с верхнего левого угла, системный аналитик определяет варианты использования и актантов и объединяет их в модель вариантов использования. Затем спецификатор вариантов использования детализирует каждый вариант использования, а разработчик пользовательского интерфейса создает прототип интерфейса пользователя. Архитектор присваивает вариантам использования приоритеты, в соответствии с которыми они будут разрабатываться в ходе итерации, принимая при этом во внимание связанные с ними риски. Инженер по компонентам анализирует классы и пакеты в рабочем процессе анализа, проектирует классы и подсистемы в рабочем потоке проектирования, реализует классы и подсистемы и проводит тестирование модулей в рабочем потоке реализации.

Вы можете заметить, что некоторые виды деятельности, осуществляемые в «цикле» определения требований, могут осуществляться в различных итерациях общего процесса разработки. В фазе проектирования, например, сотрудников сдерживает детализация и расстановка приоритетов небольшого числа вариантов использования, необходимых на этой фазе. Первые четыре рабочих процесса следуют друг за другом в строгой последовательности, возможно, с небольшим перекрытием. Три сотрудника, участвующих в рабочем процессе анализа, продолжают свою работу в процессе проектирования.

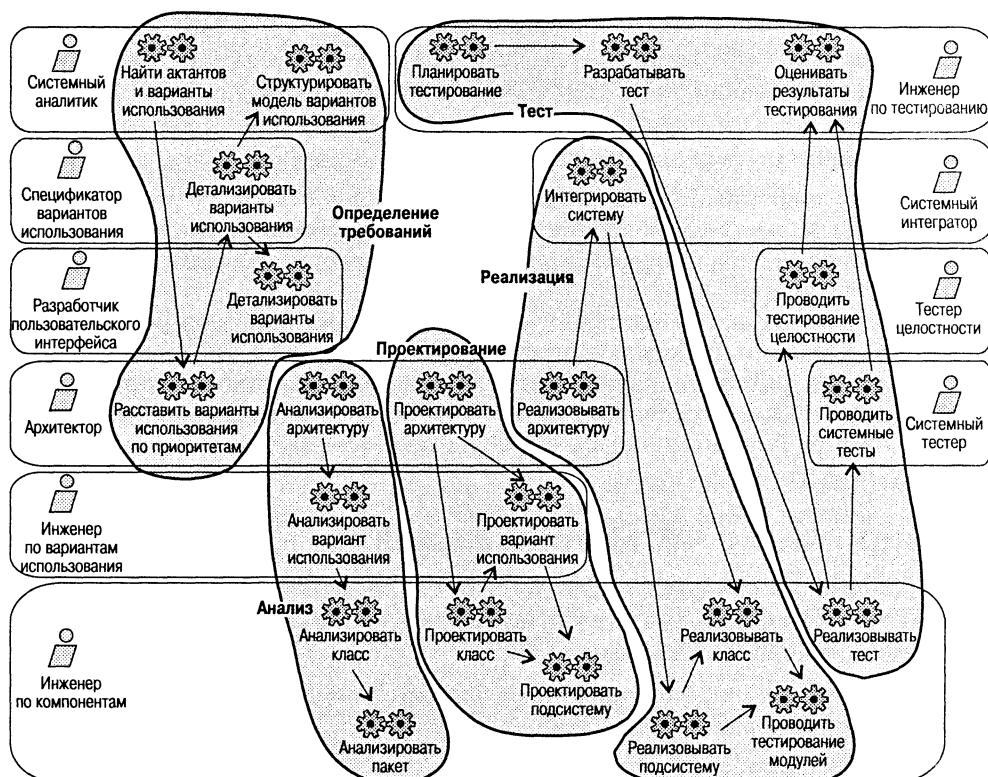


Рис. 12.3. Схема процесса разработки программного обеспечения

В отличие от них, рабочий процесс тестирования начинается очень рано, сразу же, когда инженер по тестированию начинает планировать работу. Как только в ходе процесса реализации накапливается достаточно информации, инженер по тестированию разрабатывает тесты. По мере того как в систему встраиваются компоненты, прошедшие модульное тестирование, системный тестер и тестер целостности тестируют результаты очередных уровней интеграции. Инженер по тестированию оценивает, адекватны ли результаты предписанного тестирования.

В последующих главах мы добавим к рис. 12.3 то, что необходимо нам для представления рабочих процессов итерации на каждой из фаз. Все рабочие потоки построены по образцу этой общей фигуры. Однако, поскольку центр внимания в зависимости от фазы разработки перемещается с одного базового процесса на другой, соответствующие рабочие процессы итераций в определенной степени отличаются друг от друга.

Планирование предваряет деятельность

Находясь в начале фазы анализа и планирования требований, мы знаем следующее:

- Мы собираемся выполнять проект в ходе серии итераций в четырех фазах.
- У нас имеется информация по предлагаемой системе, собранная нашими предшественниками (которая подвигла нас начать этот проект).
- У нас есть наша собственная информация о предметной области и о тех системах, которые мы делали в прошлом.

Имея эту информацию, мы должны спланировать как проект в целом (план проекта), так и каждую итерацию (план итерации). В начале, учитывая ограниченность информации, с которой мы вынуждены работать, оба плана будут очень приблизительными.

Для начала мы опишем, как планировать фазы и итерации и каким образом осуществлять итерации. В разделе «Риски влияют на планирование проекта» мы обсудим влияние рисков на планирование, а в разделе «Расстановка приоритетов вариантов использования» — как понизить риски, выбирая правильные варианты использования. В конце, в разделе «Требуемые ресурсы», мы обсудим проблему выделения ресурсов.

Планирование четырех фаз цикла

Из рекомендаций Унифицированного процесса нам известно, что должна включать в себя каждая фаза. При планировании проекта нашей задачей будет перевод этих рекомендаций в конкретные термины:

- Выделение времени. Мы определяем, сколько времени потребуется на каждую из фаз, и называем даты окончания каждой из фаз. Эти временные отметки, определенные пока приблизительно, в начале фазы анализа и планирования требований могут быть довольно расплывчаты, но должны уточняться по мере того, как мы получаем дополнительную информацию. Окончательно планы

определяются к концу фазы проектирования, когда мы готовим окончательное предложение.

- Главные вехи. Фаза заканчивается тогда, когда выполняются заранее установленные условия. Эти условия должны быть чем-то большим, нежели вопросы из учебника, они должны быть основаны на нашем прошлом опыте в данной предметной области, на информации о степени отличия новой системы от ее предшественников, о характеристиках производительности, которых мы собираемся достичь, и на способности нашей организации создать это программное обеспечение.
- Итерации в ходе фазы. В ходе каждой фазы работа производится за одну или несколько итераций. Обобщенный образ каждой итерации содержится в укрупненном плане проекта.
- План проекта. План проекта представляет собой «дорожную карту», содержащую график работ, даты и критерии прохождения главных вех и разбивку фаз на итерации.

Следует ожидать, что первая итерация фазы анализа и определения требований будет нелегкой. В добавление к самой итерации вы должны еще:

- Провести подгонку Унифицированного процесса под текущий проект и выбрать средства автоматизации процесса.
- Начать подбирать людей с талантами, нужными для проекта.
- Построить отношения, позволяющие создать успешную команду.
- Изучить предметную область, нередко новую для команды.
- Ощутить природу проекта. Для целинного проекта это будет сделать сложнее, чем при создании нового поколения существующего продукта.
- Добиться автоматизма в работе членов команды с утилитами, используемыми для интенсификации процесса и проекта.

Планирование итераций

Каждая фаза содержит одну или более итераций. Планирование итераций включает в себя набор действий, сравнимый с набором действий при планировании фазы:

- График итерации. Мы определяем, какого времени требует выполнение каждой итерации и ее дату окончания, сначала приблизительно, а затем точнее и точнее по мере того, как у нас прибавляется знаний.
- Содержимое итерации. После того как в план проекта внесены спланированные в общих чертах (таких, как даты начала) итерации, мы должны запланировать задачи на итерацию более детально. Содержимое итерации основано на том:
 - какие варианты использования должны быть описаны в ходе итерации, хотя бы частично;
 - какие технические риски следует определить, перевести в варианты использования и снизить;

- какие изменения требований следует учесть и какие дефекты следует устранить;
- какие подсистемы должны быть полностью или частично реализованы. (Содержание этого пункта меняется в зависимости от рассматриваемой фазы. В ходе фазы проектирования, например, мы определяем большую часть подсистем и все архитектурно значимые классы. В ходе фазы построения мы дополняем подсистемы всем новым и новым поведением, получая все более законченные компоненты.)

План текущей итерации полностью детализирован, а план следующей обрашает все новыми деталями по мере того, как мы узнаем что-то новое. Детализация последующих итераций ограничена объемом знаний, полученных нами до настоящего момента.

- **Промежуточные вехи** (приложение В). Выполнение определенных заранее условий (предпочтительно к установленной дате) сигнализирует нам об окончании каждой из итераций.
- **План итерации** (приложение В). Деятельность, происходящая в ходе каждой итерации, записывается в подробном плане итерации. В начале каждой итерации мы должны затребовать персонал, который сможет выполнять деятельность, ожидаемую от участников в итерации сотрудников.

Число итераций, которые планируется пройти на каждой фазе, различно и существенно зависит от сложности разрабатываемой системы. Очень простой проект можно построить, используя по одной итерации на каждую фазу. Более сложные системы потребуют большего числа итераций. Например:

- **Фаза анализа и планирования требований** (приложение В). Одна итерация, предназначенная в основном для получения представления о системе.
- **Фаза проектирования** (приложение В). Две итерации, первая — для первоначального прохода по архитектуре, вторая — для создания базового уровня архитектуры.
- **Фаза построения** (приложение В). Две итерации для обеспечения удовлетворительной работы итоговых приращений.
- **Фаза внедрения** (приложение В). Одна итерация.

Если разрабатываемый проект становится все больше и сложнее и требует все больших затрат труда, следует обдумать, а не пора ли организации-разработчику разрастаться. Впереди может быть еще немало итераций, и их продолжительность, в зависимости от размера системы, может варьироваться от недели до, по слухам, трех месяцев.

Взгляд в будущее

В течение длительного периода существования системы вокруг нее могут происходить исключительно экстенсивные изменения — появление новых технологий, новых интерфейсов для обмена со средой и продвинутых платформ. Кроме того, люди, осуществляющие планирование, проверяют, нет ли необходимости адаптировать бизнес для других организаций, таких как дочерние фирмы или присоединенные

ненные компании. Однако забота о будущем не должна доходить до полного абсурда. Не следует пытаться создать твердокаменную системную архитектуру, если вы уже видите, что она должна быть изменена. Но с другой стороны, не стоит пытаться внести в систему излишнюю гибкость — такую гибкость, которая вам никогда не понадобится.

Планирование критериев оценки

Итерации похожи на традиционные проекты. Чтобы не допустить их полета в неизвестность, руководители проекта вдобавок к созданию графиков должны описать основные задачи каждой операции. Для выполнения этих задач они определяют критерии, по которым в состоянии установить окончание итерации. Эти критерии, например минимальный набор функций в установленный момент времени, определяют, на что в ходе итерации будет направлено внимание, и помогают своевременно перенести его на необходимые аспекты итерации. Примеры таких критериев включают в себя:

1. Функциональные требования, записанные в форме вариантов использования.
2. Нефункциональные требования, привязанные к вариантам использования, в которых они применяются.
3. Нефункциональные требования, не относящиеся к определенным вариантам использования, описанные в виде дополнительных требований, как описано в разделе «Дополнительные требования» главы 6.

Одной из целей на первых итерациях, например, может быть разрешение двусмысленностей в текущем описании требований, полученном от клиента. Критерий определяет, как создатели плана намерены определять окончание итерации по измеряемым сущностям, таким как производительность, или наблюдаемым, например ожидаемым функциям.

Менеджер проекта заблаговременно устанавливает критерии завершения для каждой итерации и фазы целиком. Каждая из них должна иметь ясную точку окончания, позволяющую разработчикам самим видеть, когда они ее завершают. Кроме того, эти точки окончания представляют собой вехи, которые могут использоваться руководством для отслеживания прогресса в работе.

Вообще говоря, все критерии оценки делятся на две категории — проверяемые требования и более общие критерии.

Инженеры по тестированию должны быть подключены к работе над проектом начиная с фазы анализа и планирования требований, как указывалось в главе 11. Они определяют, какие характеристики вариантов использования годятся для тестирования. Они планируют тестовые примеры для тестирования целостности и регрессионного тестирования, а также системные тесты. В итеративной разработке цикл тестирования также итеративен. Каждый билд, создаваемый в ходе итерации, представляет собой объект для тестирования. Инженеры по тестированию добавляют новые и уточняют старые тесты для проверки каждого билда и накапливают объем тестов для регрессионного тестирования на последующих стадиях. На первых итерациях добавляется значительно больше новых функций и новых тестов, чем на последующих. По мере того, как происходит интеграция билдов, число новых тестов уменьшается, но нарастает число регрессионных тестов, вы-

полняемых для проверки реализации получающейся системы. Поэтому первые билды и итерации требуют больших усилий по планированию и разработке тестов, а в последующих основная нагрузка приходится на проведение тестов и оценку их результатов.

Общие критерии оценки не сводятся к порядку выполнения кода, который может быть проверен тестерами. Они могут, однако, быть восприняты сначала через прототипы, а затем через серии рабочих билдов и итераций. Пользователи, заинтересованные лица и разработчики с большей пользой воспринимают дисплеи и пользовательские интерфейсы, чем статическую информацию, содержащуюся в артефактах моделей.

Критерии оценки сообщают нам, как проверить, что требования к итерации были правильно выполнены. При помощи таких условий, выполнение которых может быть проверено или просто отмечено, они позволяют менеджеру проекта определить, достигнуты ли результаты итерации. Их особое значение в том, что они дают концептуальное представление о достижении результатов. Критерии становятся более специфическими по мере того, как варианты использования, сценарии вариантов использования, требования к производительности и тестовые примеры конкретнее выражают понятие успешного приращения.

Риски влияют на планирование проекта

Способ планирования разработки новой системы предполагает определенную степень зависимости от предполагаемых рисков. Поэтому один из первых шагов в начале фазы анализа и планирования требований — это создание списка рисков. Сначала нам будет мешать недостаток информации, но мы, вероятно, получим некоторое представление о том, какие наиболее опасные риски — риски, которые определят возможность построения нами системы. По мере того, как мы выполним первую работу, у нас появится возможность определить существенные риски — риски, которые должны быть снижены, если мы хотим уложиться в планируемый график и затраты и получить качественный продукт.

Управление списком рисков

Следует заметить одну вещь: разработка программного обеспечения как бы подразумевает риски. Считается, что следует извлекать риски на поверхность, чтобы каждый мог рассмотреть их, руководствоваться ими и предпринимать в связи с ними какие-то действия. В этом и состоит задача списка рисков. Имеется в виду нечто большее, чем складывание их в ящик или папку персонального компьютера. В списке содержится все, что необходимо вам для работы с рисками, в том числе и их уникальные идентификаторы. В список входят:

1. Описание. Сначала это краткое описание. Оно разрастается по мере того, как мы узнаем дополнительные подробности.
2. Приоритеты. Установите приоритетность рисков, для начала хватит трех уровней: критические, существенные и обычные. По мере разработки списка вы можете пожелать добавить еще несколько категорий.

3. Влияние. На какие части проекта повлияет данный риск?
4. Наблюдение. Кто отвечает за отслеживание продолжительного риска?
5. Ответственность. Какое лицо или организация отвечает за устранение риска?
6. Непредвиденные случаи. Что следует делать, если риск реализуется?

В проектах разумного размера могут быть найдены сотни рисков. В больших проектах риски, для облегчения их сортировки и поиска, могут храниться в базе данных. Команда разработчиков не должна пытаться за один раз справиться с ними всеми. Это еще один плюс итеративной разработки. Риски сортируются по степени значимости или по тому эффекту, который они могут оказать на разработку, и обрабатываются в порядке очередности. Как мы неоднократно подчеркивали, первыми должны рассматриваться риски, способные привести к срыву проекта. Не все риски не удастся легко разрешить, и часть рисков на некоторое время «зависнет» в списке рисков. Некоторые организации считут полезным для концентрации усилий свести этот список к «верхней десятке» рисков.

Список рисков — не статический инструмент. По мере обнаружения новых рисков список разрастается. Если мы устранием риска или проходим тот этап разработки, когда риск мог материализоваться, мы удаляем его из списка. Менеджер проекта должен время от времени, обычно при оценке результатов итерации, собирать совещания для обзора состояния наиболее опасных рисков. Аналогичные совещания по менее критичным рискам проводят другие руководители.

Влияние рисков на план итераций

В ходе фазы анализа и планирования требований определяются наиболее опасные риски, и команда разработчиков старается их понизить. Они исследуют природу этих рисков для того, чтобы разработать план итераций. Чтобы понять, как именно следует создавать план, они, например, могут разработать небольшой набор вариантов использования, связанных с рисками, и включить его в концептуальный прототип. Тогда, вводя некоторые данные, они могут обнаружить, что прототип дает неприемлемые результаты и это вызвано, например, критическим риском. (Эти исходные данные должны находиться в заданном диапазоне, возможно, на его границах, и давать неприемлемые результаты.)

В добавок к тому эффекту, который оказывают на успех разработки наиболее серьезные риски, все риски нарушают график разработки, ее стоимость или качество. Некоторые из этих рисков могут серьезнее других увеличить сроки разработки или потребовать больших усилий, чем планировалось, поэтому они должны быть снижены до того, как эти неприятности произойдут. Почти всегда нарушение графика влечет за собой рост усилий и затрат. Иногда риск может незначительно нарушать график или затраты, нанося основной ущерб другим характеристикам продукта, например качеству или производительности.

Выделение рискованных действий

Основное правило — проводить акции по борьбе с рисками на основе плана. Фазы, а внутри них итерации, предоставляют нам механизмы для выделения рискованных действий. Это, например, план работы с рисками, угрожающими возможнос-

ти создания системы в ходе итераций фазы анализа и планирования требований. То есть мы по возможности уходим от них, но, по крайней мере, имеем план действий на случай их материализации.

Как показывает наш опыт, альтернатива — не заниматься выделением рисков — не слишком хороша. Не прилагая сознательных усилий к выявлению рисков на ранних стадиях процесса разработки, мы частенько сталкиваемся с ними позже, во время осуществления тестов на целостность или системных тестов. К этому моменту решение каких-либо серьезных проблем, которые могут потребовать многочисленных модификаций, приведет к задержке на несколько недель или более. При итеративном подходе раннее, начиная с первой фазы, создание прототипов, бильдов и артефактов позволит нам обнаружить риски тогда, когда у нас еще есть время для их понижения.

Мы понимаем, что некоторые виды рисков достаточно трудно обнаружить. Отдельные риски могут быть «неясными» по многим причинам, чаще всего потому, что люди не подозревают об их истинной серьезности. Другая причина, по которой риски могут остаться незамеченными, состоит в том, что многие люди, участвующие в разработке, считают шуткой разговоры о действительной возможности создания продукта за предложенную цену и в расчетное время. Если какие-то риски проскользнут через сито идентификации, они не будут учтены при планировании итераций, не будут также запланированы меры по их снижению.

Независимо от причины, некоторые риски в некоторых проектах могут оставаться незамеченными до поздних стадий разработки, особенно если команда разработчиков неопытна в управлении рисками. Набирая практику и опыт, команда повышает свои способности расставлять риски в такой последовательности, которая позволяет вести работу над проектом в соответствии с его логикой. В фазе построения, например, риски, способные вывести из графика вторую итерацию, должны быть снижены не позднее, чем на первой итерации этой фазы. Наша задача состоит в том, чтобы каждая итерация протекала без проблем и в соответствии с планом. Будет не слишком хорошо, если проект вдруг столкнется с незамеченным вовремя риском, с которым уже невозможно быстро разобраться.

Расстановка приоритетов вариантов использования

В этом пункте мы обсудим выбор ведущих вариантов использования для одиночной итерации. Напомним, что каждая итерация в Унифицированном процессе управляется набором вариантов использования. На самом деле правильнее будет сказать, что итерация управляется набором сценариев через варианты использования. Это будет точнее, поскольку на ранних итерациях мы можем не иметь полных вариантов использования. У нас есть только сценарии или пути, связывающие варианты использования с нашей задачей. Иногда, говоря о выборе варианта использования, мы подразумеваем выбор сценариев, относящихся к данной итерации.

Работа, которой мы при этом занимаемся, называется *расстановкой приоритетов вариантов использования* (см. подраздел «Деятельность: Определение приори-

тетности вариантов использования» главы 7). Приоритет вариантов использования определяется тем порядком, в котором они — или входящие в них сценарии — будут использованы в итерациях. Расстановка приоритетов производится в течение нескольких итераций. Некоторые варианты использования (или их сценарии) ранжируются на ранних итерациях, но многие из них в этот момент еще не определены и не могут быть расставлены в соответствии с приоритетом. Все определенные варианты использования должны получить приоритет. Результатом расстановки приоритетов будет ранжированный список вариантов использования.

Это ранжирование управляет рисками. Мы расставляем приоритеты вариантов использования в соответствии с рисками, которые в них входят. Здесь мы используем термин *rиск* в широком смысле. Так, например, необходимость внесения изменений в архитектуру на поздних фазах — это риск, которого мы хотим избежать. Невозможность создания правильной системы — это риск, который мы хотим уменьшить в самом начале разработки путем правильного определения требований. Процесс выбора также управляет рисками. Мы вносим обнаруженные риски в список рисков (см. подраздел «Управление списком рисков» данной главы) и перемещаем каждый из них в тот вариант использования, реализация которого будет снижать этот риск. Этот вариант использования будет затем помещен в ранжированный список вариантов использования, причем его позиция в этом списке будет соответствовать уровню его рискованности.

На ранних итерациях мы занимаемся расстановкой приоритетов вариантов использования в соответствии с рисками, которые относятся к целям системы и архитектуре. На более поздних итерациях мы выбираем новые варианты использования для наполнения уже определенной архитектуры дополнительными функциями. Мы наращиваем мышцы на скелете. Поздние варианты использования добавляются в некотором логическом порядке. Этот логический порядок соответствует положениям вариантов использования в ранжированном списке вариантов использования. Так, например, варианты использования, нуждающиеся для работы в других вариантах использования, имеют более низкий приоритет и разрабатываются позже. Обсуждение итеративного подхода как попытки управлять процессом при помощи рисков приведено в разделе «Итеративный подход — управляемый рисками» главы 5. В следующих трех пунктах мы рассмотрим три категории рисков: риски, специфические для отдельных продуктов, риски, связанные с архитектурой, и риски неправильного определения требований.

Риски, характерные для отдельных продуктов

Этот тип рисков — технические риски — мы обсуждали в подразделе «Итерации снижают технические риски» главы 5. Мы переносим их в варианты использования, которые, будучи правильно реализованы, уменьшают риски этого типа. Каждый риск отображается на вариант использования, который при реализации снижает его. Мы можем определять эти риски один за другим, поскольку их рассмотрение невозможно формально включить в процесс разработки. Под «формальным включением в процесс разработки» мы понимаем наличие в процессе специального момента, когда мы занимались бы рассмотрением рисков этого типа. Так, например, как обсуждается в следующем подразделе, риски, связанные с архитектурой, разбираются частью в фазе анализа и планирования требований, а частью

в фазе проектирования. Риски, которые не могут быть формально включены в процесс, следует рассматривать один за другим и снижать до того, как их существование сможет помешать разработке.

Риск не создать правильную архитектуру

Один из наиболее серьезных рисков — это риск не построить систему, которая сможет плавно развиваться в течение будущих фаз или времени жизни, то есть риск не создать гибкую архитектуру. Этот риск должен быть со всей определенностью рассмотрен в ходе фаз анализа и определения требований и проектирования, и мы должны быть уверены в том, что создали правильную архитектуру и в состоянии сохранять ее (исключая небольшие изменения на фазе построения). Мы говорили об этом в предыдущем пункте, когда касались включения в Унифицированный процесс обнаружения и рассмотрения некоторых видов рисков. В данном случае, например, полное рассмотрение архитектуры должно произойти на фазах анализа и определения требований и проектирования.

Как определить, какой вариант использования более важен для построения правильной архитектуры? Как вообще мы можем понизить риск не создать правильную архитектуру? Нам нужны архитектурно значимые варианты использования. Именно они содержат основные задачи или функции, которые выполняет система. Задайте себе вопрос: зачем мы создаем эту систему?

Ответ содержится в основных вариантах использования — тех, которые наиболее важны для пользователей системы. Кроме того, в эту категорию попадут варианты использования, содержащие важнейшие нефункциональные требования — по производительности, времени отклика и т. д. Эти варианты использования обычно помогают обозначить каркас системы, на который мы будем навешивать остальные необходимые функции (см. подраздел «Артефакт: Описание архитектуры» главы 7). Другие существующие категории вариантов использования:

- *Вторичные.* Эти варианты использования поддерживают выполнение основных вариантов использования. В них входят второстепенные функции, такие, как мониторинг или сбор оперативной статистики. В основном варианты использования этой категории вносят исключительно скромный вклад в архитектуру, несмотря на то, что может потребоваться достаточно ранняя их разработка, если, например, заинтересованные лица остро нуждаются в получении определенных данных, таких, как плата за транзакцию, описанная в примере в подразделе «Риск неправильного определения требований» данной главы. В этом случае их приоритет должен быть повышен, поскольку мы хотим снизить риск неполучения корректных требований.
- *Вспомогательные (желательные).* Эти варианты использования не являются ключевыми для архитектуры или для наиболее серьезных рисков. Мы редко начинаем заниматься этим уровнем вариантов использования в ходе итераций фаз анализа и планирования требований и проектирования. Такое может случиться только при необходимости разгрузить критические или важные варианты использования.
- *Необязательные.* Некоторые варианты использования могут быть критически или важными, даже если они не всегда присутствуют в модели. Нам может

понадобиться их обработать, поскольку если они присутствуют, то оказывают влияние на архитектуру.

Кроме того, мы хотим быть уверены, что прошлись по всем вариантам использования, которые могут оказывать влияние на архитектуру. Мы не хотим упустить какую-либо функциональность, разработать ее слишком поздно и получить в результате неустойчивую архитектуру. Нам нужно хорошее покрытие вариантов использования, которые могут повлиять на архитектуру. Важно именно хорошее покрытие, не только для того, чтобы определить архитектуру, но и для того, чтобы быть уверенными в том, что мы точно предсказали стоимость разработки продукта на первом цикле. Мы должны избежать риска слишком позднего обнаружения, что мы не можем добавить к системе только что найденную функциональность.

По этой причине мы и должны добиться покрытия около 80% вариантов использования в фазе проектирования. Под «покрытием» мы понимаем разбор вариантов использования и влияния, которое они могут оказать на систему. Практически, обычно мы определяем около 80% вариантов использования и включаем их в модель вариантов использования, но обычно не считаем нужным описывать их детально. В обычном проекте мы можем лишь частично определить варианты использования. Некоторые из этих описаний могут быть короткими, всего в несколько строк, если этого хватит для объяснения того, что нам следует знать на этой фазе. В относительно несложных проектах при работе с требованиями мы можем детализировать незначительную часть вариантов использования. В более крупных проектах с высокими рисками мы можем считать желательным описание 80% вариантов использования или даже больше.

Риск неправильного определения требований

Еще один серьезный риск — риск не получить систему, которая делает то, чего хотят от нее пользователи. Это означает, что рассмотрение этого риска тоже входит в процесс разработки. В конце фазы проектирования мы хотим быть уверены в том, что создаем правильную систему. Это решение нельзя откладывать, поскольку на фазе построения деньги начинают утекать рекой. Какие варианты использования нужны нам для подтверждения того факта, что разрабатываемая нами система действительно требуется пользователям? Какие варианты использования убедят нас в том, что система на следующих фазах сможет развиваться, поскольку мы будем в состоянии добавить в нее всю необходимую для первого выпуска функциональность? Мы не хотим забыть какие-то функции. Мы хотим знать, что строим систему, способную к развитию.

Первая часть вопроса, разумеется, относится к рабочему процессу определения требований. Мы можем создать бизнес-модель (или в некоторых случаях более ограниченную модель предметной области). Решение второй части состоит в том, чтобы в ходе ранних итераций путем создания прототипа построить систему, которую ждут пользователи, и как можно раньше получить их отзывы. Только в ходе реального использования мы можем удостовериться в том, что создали правильную систему.

Пример. Биллинговая и Платежная система. В Биллинговой и Платежной системе мы можем предположить, что банк считает очень важным делом получе-

ние платы за свои услуги. Допустим, он хочет получать небольшую плату за каждую транзакцию. Включение этой платы будет добавлением новых функций к базовым вариантам использования *Заказать товары или услуги*, *Подтвердить заказ*, *Выставить счет покупателю* и *Оплатить счет*. С точки зрения архитектора, однако, функции начисления оплаты могут быть не слишком существенны для создания правильной архитектуры. *Начисление оплаты* можно рассматривать как расширение других вариантов использования, его можно комбинировать с другими вариантами использования, которые отвечают за оплату. Насколько разработчики понимают, функции начисления оплаты ничем не заслужили первоочередного исполнения. Однако с точки зрения заказчика крайне важно, чтобы варианты использования, отвечающие за оплату, были правильно реализованы до начала поставки. По этой причине они объявляются имеющими высокий риск и становятся важными.

В результате, когда менеджер проекта обдумывает порядок итераций, он учитывает важность функций начисления оплаты. С одной стороны, если он посчитает, что начисление оплаты в том варианте использования, который он изучает, — это простая функция, которая не представляет проблемы для разработчиков, он может решить, что разработчикам нет необходимости создавать ее на фазе проектирования, и отложит ее на фазу построения. С другой стороны, если он обнаружит, что с начислением оплаты (отдельно от других вариантов использования) связано несколько сложных внутренних проблем, он запланирует создание функций оплаты на итерацию фазы проектирования. Одной из этих «сложных проблем» может быть желание заказчика как можно быстрее увидеть вопрос начислений решенным.

Требуемые ресурсы

Вы чувствуете, что план итераций, основанный на фазах разработки программного обеспечения, обладает серьезными достоинствами, но вас беспокоят несколько вопросов:

- Каких затрат потребуют фазы анализа и планирования требований с точки зрения усилий и с точки зрения необходимости квалифицированного персонала?
- Сколько на эти фазы необходимо потратить денег?
- Сколько времени уйдет на эти две фазы?
- На сколько месяцев из-за первых фаз задержится то, что многие люди, собственно, и считают настоящим делом, или разработкой программного обеспечения, то есть построение?

Проекты сильно различаются

Разумеется, не секрет, что предлагаемые системы сильно отличаются по своей готовности к началу разработки. Перечислим четыре примера:

1. Абсолютно новый или беспрецедентный проект в неисследованной предметной области — целина. Никто не знает толком, что надо делать и вообще можно

ли это сделать. Опыта мало. Мы можем связаться со знающими людьми, чтобы получить несколько полезных советов. В таких условиях кто бы ни хотел заказать эту систему, он должен обеспечить финансирование фаз анализа и определения требований и анализа. Эти фазы финансируются едва ли не на условиях исследования, то есть с оплатой затрат. Для проведения фаз анализа и определения требований и анализа с фиксированным бюджетом или графиком не хватает данных. В такой ситуации определение контекста, поиск возможной архитектуры, определение наиболее опасных рисков и создание бизнес-предложения — это основные пожиратели времени в фазе анализа и планирования требований. Точно так же и выполнение задач фазы проектирования, то есть приведение проекта в состояние, когда возможно планирование фазы построения, требует длительного времени.

2. Продукт такого типа, который создавался раньше, в области, в которой имеются примеры в виде ранее созданных программ. Компоненты, пригодные для повторного использования, отсутствуют. Ранее созданные продукты предоставляют нам руководство по выбору возможной архитектуры, но чтобы удостовериться в том, что ранее использовавшаяся архитектура подходит и нам, уйдет несколько дней. В таких условиях фаза анализа и планирования требований, вероятно, будет краткой (насколько недель). Она может потребовать найма всего лишь одного или двух людей с опытом на полный рабочий день, но желательно привлечь знания и других опытных людей, с которыми нужно будет консультироваться этой маленькой проектной группе. Поскольку продукты такого типа уже создавались, основные риски маловероятны, но для проверки этого факта потребуется еще несколько дней.
3. Существует унаследованный продукт, который должен быть сделан более современным, например, путем перевода с мэйнфрейма на архитектуру клиент-сервер. Части кода унаследованной системы до некоторой степени следует выделить и использовать в новой системе. В фазе анализа и планирования требований подыскивается возможная архитектура. Если следует обеспечить возможность повторного использования, команда знает, что это можно сделать. Однако, если организация, заказавшая разработку, никогда не делала этого раньше, информация по графику и затратам отсутствует. Следует создать базовый уровень архитектуры. Следует определить интерфейс между новой и старой системами начиная с вариантов использования и найти подсистемы — одна из подсистем будет содержать скрытые неизменные части унаследованной системы.
4. Существуют готовые компоненты на рынке или внутри организации. Организация-разработчик определяет, что значительную часть новой системы, обычно от 50 до 90%, можно набрать из этих компонентов, однако имеются пустоты, которые потребуют создания нового кода. Проект требует определения и описания интерфейсов между повторно используемыми и новыми компонентами, а также интерфейсов с внешними системами и пользователями. Разработка из компонентов также требует времени и усилий и может содержать риски. Однако в целом разработка из компонентов быстрее и менее дорога, чем разработка с нуля.

Эти примеры не предназначены для выделения индивидуальных категорий. Скорее они описывают перекрывающиеся состояния. Таким образом, мы можем мыслить в понятиях исходных состояний. Какой опыт имеет наша организация в предметной области? Насколько велика наша база повторно используемых компонентов? Предлагается ли что-то большее, нежели новый выпуск существующего продукта? Требуется ли высокий уровень новизны? Распределенная ли это система (будет ли достаточно версии под одну платформу)?

Как выглядит типичный проект

Несмотря на неопределенность различных исходных состояний, первые циклы разработки средних по размеру проектов требуют усилий и укладываются в примерный график, показанный на рис. 12.4. Обычно для создания архитектуры и снижения рисков при основанной на фазах разработке работы, по сравнению с водопадным подходом, переносится на начало цикла.

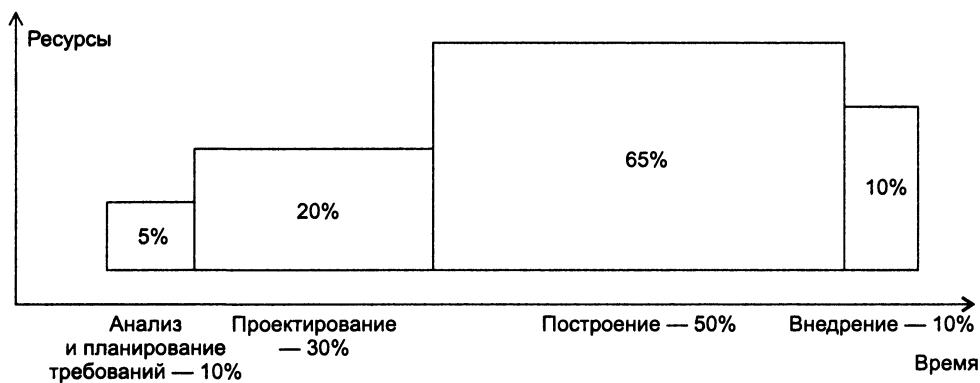


Рис. 12.4. Максимума времени и усилий требует фаза построения, однако и на три другие фазы также приходится значительная их доля

Сложный проект требует большего

Что получится, если мы представим себе большой, сложный проект — например построенный на основе новых технологий, с распределенной архитектурой или работающий в реальном времени? Нам, вероятно, понадобится большое количество итераций. На фазы анализа и планирования и проектирования нам понадобится больше времени и усилий. В результате, как показано на рис. 12.5, эти две фазы вырастут.

Поспешим подчеркнуть, что проценты, указанные на этих рисунках, приблизительные. Их не следует слепо применять к настоящим проектам. Они приведены исключительно с целью показать, что чем больше неизвестности присутствует в проекте, тем больше времени и усилий требуется для осуществления фаз анализа и определения требований и проектирования. Так, в примере мы отводим на эти

фазы больше времени, чем на две остальные фазы. Однако усилия возрастают не так сильно, поскольку у нас нет необходимости увеличивать ресурсы, необходимые для проведения этих фаз в такой же пропорции. В то же время растут усилия и время, затрачиваемые на проект в целом.



Рис. 12.5. В условиях сложного проекта на ранних фазах по сравнению с поздними осуществляется более значительная доля работы. В первую очередь это относится к календарному времени

Использование каркаса сильно сокращает фазу построения, оказывая на ранние фазы менее значительное действие. Фазы анализа и планирования требований и проектирования в случае повторного использования архитектуры удлиняются, однако функции, составляющие каркас, не потребуют анализа и проектирования, поэтому проект в целом потребует меньшего времени и усилий.

Новая линия продуктов требует опыта

В многих случаях, особенно для новых или сложных систем, команда должна получить информацию из обладающих ею источников. Стандартный источник информации — это люди, знающие предметную область будущей системы. Даже если в наличии имеются детальные требования, беседа с такими людьми поможет команде разработчиков найти правильные архитектурные решения и определить наиболее важные риски. Найти знающих людей — это половина победы. Обычные пользователи могут иметь представление исключительно о своей части общего процесса, ничего не зная о том, что способна делать компьютерная система в целом.

Стандартная ошибка для периода начала работ над новой линейкой продуктов — неиспользование *предыдущего опыта*. Поскольку большая часть опыта в тех направлениях, над которыми компания ранее работала, уже осела в головах сотрудников гораздо прочнее, чем в документации (которую все равно никто не читает), использование предыдущего опыта в основном означает использование опытных сотрудников. Эта ошибка проявляется в назначении в команду разработки новых сотрудников вместо направления в нее людей, знающих если и не новую линию продуктов, то хотя бы как вообще принято работать в компании.

Когда компания планирует разработку новой линии продуктов, то частью своего коллективного разума она понимает, что эта работа может потребовать наиболее компетентных и опытных сотрудников. В то же время, однако, другая часть коллективного разума считает, что люди такого калибра необходимы для поддержания текущей работы. Они должны обслуживать существующих заказчиков. Компания должна сохранить текущие финансовые поступления.

У этой дилеммы нет простого решения. Руководители, у которых в настоящее время работают эти опытные сотрудники, разрываются между этими двумя задачами — очевидно, что важны и существующая, и новая линии продуктов. На практике многие старшие разработчики бывают прикреплены к существующей линии продуктов в течение многих лет. Они психологически привязаны к ней. Обычно многие из них не желают рисковать существующим положением, отдавая своих лучших разработчиков в новый проект. В результате в новой команде оказывается всего несколько опытных сотрудников, которые могут и не быть жесткими лидерами — техническими или административными. Это могут быть и люди, на уход которых руководители с охотой согласились. Эти новые лидеры начинают заполнять кадровый недостаток большим количеством новых сотрудников, часто прямо из колледжа.

Вдобавок к неопытности, новые сотрудники приносят с собой особую культуру. Они имеют привычку считать все старое — то, что было сделано компанией, — вышедшим из моды. Только новое хорошо. Новички недооценивают важность тех вещей, которым их не учили в школе, — таких, как методы разработки или управление жизненным циклом.

Мы заметили, что компании часто срывали разработку новой линии продуктов из-за персонала, имея необходимых для разработки талантливых сотрудников, работавших полный день для заполнения рыночной ниши. Недостатки обычно не исправлялись до выхода второго поколения продукта. В сущности, первое поколение продукта было только широкомасштабной пробой, хотя это и не было в действительности первоначальным намерением компании.

Цена за использование ресурсов

Поддержка команды, занимающейся первыми двумя фазами, как бы мала она ни была, требует средств. Фактически трудность нахождения средств на поддержку этих двух фаз является одной из причин, побуждающих организации, занимающиеся разработкой программного обеспечения, преждевременно переходить к - фазе построения, что иногда приводит к широко обсуждаемым ошибкам. Откуда же берутся эти средства?

- В случае организаций, занимающихся разработкой программного обеспечения на продажу, средства берутся из накладных расходов и, следовательно, находятся под контролем руководства. В дальнейшей перспективе, однако, они будут получены с клиентов или заказчиков. Другими словами, организация, разрабатывающая программное обеспечение, повышает текущие цены на программное обеспечение, чтобы покрыть стоимость разработки будущих продуктов.
- В случае организаций, занимающихся разработкой программного обеспечения для внутренних клиентов, затраты на первые две фазы поступают как из ее накладных расходов, так и из средств, ассигнованных клиентом специально для

этой цели, или из средств, выделенных на это высшим руководством. Будет ли применены два последних способа финансирования, зависит от того, насколько руководство других подразделений разберется в суммах, необходимых для проведения ранних фаз, и выделит ли оно на это деньги.

- В случае организаций, занимающихся разработкой программного обеспечения для отдельных корпоративных клиентов, затраты на первые две фазы могут быть взяты из их собственных накладных расходов. В том случае, если предложение, на подготовку которого пошли эти расходы, будет отклонено, это может повлечь за собой истощение накладных средств. Однако, если предлагаемый проект будет принят, сделан и сдан в рамках нормальных бизнес-процедур фирмы-разработчика, этого источника финансирования будет вполне достаточно. Если предлагаемый проект выглядит более рискованным, чем обычно, клиент должен возместить затраты на первые две фазы.

Действительность такова, что в первых двух фазах разработки выполняется важная работа, требующая времени и денег. Кроме того, вдобавок к деньгам, она требует участия в работе клиента. Такое сотрудничество — предоставление людей, от которых организация-разработчик получает необходимую информацию, — также стоит денег (даже если эти расходы и не вписаны в счет).

Оценка итераций и фаз

Если преимущества итеративной разработки реализуются полностью, после окончания каждой итерации или фазы их результаты должны быть подвергнуты оценке. За это отвечает менеджер проекта. Такая работа проводится не только для того, чтобы оценить выполнение итерации. Она преследует также две дополнительные цели:

- Пересмотреть план следующей итерации в свете информации, полученной командой в ходе работы над предыдущей итерацией, и внести в него необходимые изменения.
- Модифицировать процесс, подработать утилиты, интенсифицировать обучение, и провести другие действия, диктуемые опытом оцениваемой итерации.

Первая цель оценки — оценить проделанную работу в понятиях заранее заданных критериев выполнения. Вторая — рассмотреть прогресс, достигнутый в ходе итерации или плана проекта:

- Протекает ли работа в рамках бюджета и в соответствии с графиком?
- Соответствуют ли требованиям к качеству, высказанным заинтересованными лицами, результаты, полученные в ходе тестирования или рассмотрения прототипа, компонента, билда или приращения?

В идеале проект должен удовлетворять этим критериям. Менеджер проекта распространяет результаты оценки среди занимающихся этим вопросом заинтересованных лиц и регистрирует этот документ. Он неизменен, отчет о следующей оценке — это уже другой документ.

Критерии не достигнуты

Именно такой результат получается очень редко. Чаще итерация не полностью удовлетворяет критериям завершения. Невыполненную работу можно перенести на следующую итерацию проекта (или более поздние итерации). Эта работа может включать в себя:

- Модификацию или расширение модели вариантов использования.
- Модификацию или расширение архитектуры.
- Модификацию или расширение подсистем, если разработка уже дошла до этой стадии.
- Поиск дополнительных рисков.
- Приобретение командой некоторых навыков или умений.

Вероятно также, что для выполнения существующего плана может потребоваться дополнительное время. В этом случае меняется график первой итерации. Если это произошло, внутрифирменная дата окончания работ переопределяется.

Критерии сами по себе

На самом деле вопрос о критериях тоже важен. Что такое эти критерии окончания? Команда разработчиков может достигнуть всех критериев, не имея при этом всей необходимой информации. В ходе итерации может потребоваться решить дополнительные задачи, или обнаружится, что описанные задачи были неверно сформулированы. В результате лица, оценивающие результаты, могут изменить критерии, а не просто проверить их выполнение.

Следующая итерация

Главная веха (приложение В) отмечает окончание фазы, точку, в которой не только работающая над проектом команда, но и прочие заинтересованные лица, в частности, лица, финансирующие проект, и представители пользователей, оценивают, соответствует ли проект критериям вехи и следует ли утвердить его переход к следующей фазе.

В качестве основы оценки менеджер проекта (к которому помогают некоторые из тех, кто выполнял итерацию или фазу, подгоняемые теми, кто планировал будущую итерацию) делает следующее:

- Определяет, что работа может перейти к следующей итерации.
- Если необходимы доработки, распределяет их по последующим итерациям.
- Детально планирует следующую итерацию.
- Дополняет, с меньшей степенью детализации, итерации, идущие за следующей.
- Дополняет список рисков и план проекта.
- Сравнивает реальные затраты и график итерации с запланированными.

Мы можем заметить, что в разработке, основанной на применении компонентов, стандартная шкала (число написанных строк кода) не является надежным индикатором прогресса. На протяжении этого процесса разработчики могут повторно использовать ранее созданные строительные блоки (подсистемы, классы

и компоненты), можно достичь значительного прогресса путем написания очень небольшого количества нового кода.

Развитие набора моделей

Основной характеристикой пофазовой итеративной разработки является эволюция набора моделей. Эта эволюция отличается от развития по модели водопада, в которой, как мы представляем, сначала завершается процесс определения требований, затем анализ и т. д. При итеративной разработке модели в ходе фаз развиваются совместно. На ранних итерациях одни модели опережают другие. Например, модель вариантов используется перед моделью реализации. Несмотря на то, что каждая модель развивается практически независимо от других, мы можем думать об общем состоянии всей системы, которая переходит в более развитое состояние, как мы показывали на рис. 5.7. Каждая итерация — возможно, каждый билд итерации — демонстрирует прогресс в состоянии всей системы. Развитие состоит в постепенном движении к завершению набора моделей. Степень продвижения в этом направлении есть важнейший индикатор, исследуемый группой оценки.

В следующей главе мы вернемся к началу проекта и обсудим фазу анализа и планирования требований со всеми ее проблемами.

13 Анализ и планирование требований инициирует проект

Общая цель фазы анализа и планирования требований — инициировать проект. До этой фазы у нас может быть только витающая в воздухе хорошая идея. Она может возбуждать, если это абсолютно новая идея в новой области. Она может успокаивать, если это идея нового выпуска существующего продукта. Анализ и планирование требований могут быть просты настолько, что один человек создаст концепцию, набросает архитектуру, включая различные диаграммы, и подготовит разумный бизнес-план. Они могут быть сложны, как сложен полномасштабный исследовательский проект. Проблема в том, что мы не можем свести анализ и планирование требований к одной методике. На этой стадии мы очерчиваем проблемы, которые хотим решить, придавая себе этим ту долю уверенности, которая необходима и достаточна для успешной разработки системы. Это особенно важно при создании новой системы.

Разумеется, уверенность должна быть не только у вас. Работа, выполняемая в фазе анализа и планирования требований, демонстрируется клиенту (или его представителям), организации-разработчику и другим заинтересованным лицам, чтобы уверить их, что архитектор и разработчики в состоянии снизить наиболее опасные риски, сформулировать предложения по архитектуре и создать первичный бизнес-план.

Введение

Цель фазы анализа и определения требований состоит в создании бизнес-плана, достаточно точного для принятия решения об утверждении и запуске проекта. Чтобы создать этот бизнес-план, нам для начала нужно обозначить область действия предполагаемой системы. Нам следует знать область действия для того, чтобы понять, что мы должны включить в разрабатываемый проект. Нам следует знать область действия системы для того, чтобы понять, что следует включать в архи-

текстуру. Нам следует знать ее для того, чтобы определить область поиска наиболее опасных рисков. Нам следует знать ее для того, чтобы определить пределы затрат, рабочего графика и прибыли на вложенный капитал — параметров, входящих в бизнес-план.

Мы должны постараться узнать хоть что-нибудь об архитектуре, чтобы удостовериться, что эта архитектура в состоянии поддерживать область действия системы. Это то, что мы понимаем под «возможной архитектурой».

Мы также должны предупредить возможный крах. Множество сложных проектов провалились потому, что наиболее опасные риски упускались из виду. Нередко до этапа сборки и тестирования о них никто и не вспоминал. В результате, когда эти риски проявляли себя, их не удавалось смягчить без выхода за пределы бюджета и/или отведенного на проект времени. Неизвестность, или риск, всегда присутствует в работе, нравится нам это или нет. Для того, чтобы выйти из этой ситуации, надо или как можно раньше снижать риски, или уменьшать область действия системы, или увеличивать время и ресурсы (а значит, затраты), отводимые на предупреждение неснижаемых рисков.

Чтобы иметь возможность взглянуть на систему сквозь призму экономических понятий, таких как предварительная оценка инвестиций, график работ или возврат инвестиций, мы создаем первоначальный бизнес-план. Мы задаемся следующими вопросами:

- Будут ли прибыли, приносимые использованием или продажей программного обеспечения, превышать затраты на его создание?
- Дойдет ли оно до рынка (или внутренних нужд) достаточно быстро, чтобы привести эту прибыль?

Мы стараемся обеспечить команду, работающую над проектом, средой разработки — как процессом, так и утилитами. Разумеется, среда разработки организации-разработчика программного обеспечения — это продукт многолетних усилий, большая часть которых была сделана до начала текущего проекта. Однако мы приспосабливаем Унифицированный процесс к тому типу систем, которые собираемся разрабатывать, и к уровню компетентности организации, которая будет вести разработку. Фаза анализа и планирования требований предназначена для решения такого рода проблем.

В начале фазы анализа и планирования

Начнем с *планирования*, разовьем *концепцию* системы и приступим к подбору *критериев оценки*.

Перед началом фазы анализа и планирования требований

Перед тем, как приступить к фазе анализа и планирования требований, мы имеем некоторое представление о том, что нам предстоит сделать. Кто-нибудь — организация-клиент, наш отдел маркетинга или некто в нашей организации-разработчике — предлагает идею и обосновывает ее в степени, достаточной для проявления

некоторого интереса. Размер работы, выполняемой до начала фазы анализа и планирования требований, бывает весьма разным. Несмотря на то, что варианты этой работы крайне разнообразны, мы можем выделить из этого разнообразия три варианта:

1. Организация-разработчик программного обеспечения, которая создает продукт для свободной продажи. Объем информации, который необходим нам для того, чтобы начать работу, может быть весьма значительным. Сотрудники отделов менеджмента и продаж обычно вполне серьезно изучают предполагаемый продукт и привлекают к исследованиям нескольких разработчиков. Другими словами, они уже успевают выполнить часть работ фазы анализа и планирования требований.
2. Организация-разработчик программного обеспечения, которая создает системы для других подразделений той же компании, то есть типичное натуральное хозяйство. Мы, вообще говоря, можем выделить две ситуации. В первом случае одно из подразделений ощущает необходимость в программном комплексе и просит разработчиков программного обеспечения создать его. Запросившее программу подразделение передает разработчикам описание того, что им хочется получить, однако составитель этого описания может слабо представлять себе программные структуры. Другими словами, организация-разработчик имеет недостаточно информации для того, чтобы начать фазу анализа и планирования требований. Во втором случае высшее руководство ощущает необходимость в системе масштаба компании и привлекает к работе группу бизнес-проектирования, которая определяет задачи этой системы. В этом случае фаза анализа и планирования требований может начаться в условиях хорошего понимания требований.
3. Организация-разработчик программного обеспечения, которая создает системы для клиентов. Исходный запрос на разработку предложений обычно содержит множество деталей и часто много страниц требований. С другой стороны, если отношения с клиентом далеки от формальных, он может вообще иметь слабое представление о том, что ему нужно.

Если продукт создается на основе предыдущего выпуска, это означает, что большая доля работы по планированию первой итерации нового проекта была проведена еще на последней итерации предыдущего цикла. В случае создания продукта со значительной степенью новизны, однако, инициаторы должны проделать большую работу. Они должны быть в состоянии грубо оценить необходимые усилия, достать средства на покрытие затрат фазы анализа и планирования требований и разработать график работ.

Планирование фазы анализа и планирования требований

В начале проекта перед нами встает дилемма. Здравомыслящие люди рекомендуют нам создать план, а у нас не хватает информации, на основе которой этот план можно было бы построить. Мы пытаемся понять, что должны запланировать, чтобы получить сведения о том, что должны сделать. Об этом мы поговорим позже, в разделах «Типичный поток работ итерации» и «Выполнение основных рабочих процессов». Тем временем мы начинаем работу со следующих шагов:

- Собираем вместе ту информацию, которую сотрудники нашли до начала проекта.
- Организовываем ее так, чтобы ее можно было использовать.
- Собираем вместе тех немногих сотрудников, которые знают, как ее использовать.
- Отмечаем, чего не хватает, но не для всех четырех фаз, а для ограниченных задач фазы анализа и определения требований.

Другими словами, ограничим свою деятельность тем, что необходимо для выполнения ключевых задач этой фазы, которые были суммированы во введении к данной главе. Теперь создадим план уточнения требований, согласно которому будем собирать эти исходные цели и переадресовывать их к соответствующим вариантам использования. Запланируем создание возможной архитектуры. Мы собираемся работать с этой архитектурой до тех пор, пока не убедимся, что проект выполним, обычно лишь до создания набросков представлений архитектуры. По возможности постараитесь разобраться с этим за одну, максимум две итерации.

Не забывайте, что исходный план является *пробным*. По мере сбора дополнительной информации мы модифицируем план в соответствии с новыми знаниями. Мы, разумеется, стараемся окончить фазу анализа и планирования требований в сроки, определенные графиком не истратив на нее времени больше, чем было изначально выделено руководством (или клиентом). Поскольку руководство, естественно, установило эти значения на основании ограниченной информации, постоянно уведомляйте руководителей (и других заинтересованных лиц) о продвижении работ. Это может помочь вам доказать необходимость изменения первоначально определенных значений.

Расширение концепции системы

К началу жизненного цикла системы команда может прийти, имея только описание концепции системы размером в одну страницу. Оно содержит список предложений (см. раздел «Обзор процесса определения требований» главы 6.), краткие данные о производительности, ограниченную информацию о рисках, с которыми могут столкнуться разработчики, неопределенную ссылку на возможную архитектуру (нередко в виде фразы «клиент/сервер» или вроде того) и приблизительные, округленные до порядков, экономические показатели (например, 10 миллионов долларов и два года).

В базовую команду могут входить менеджер проекта, архитектор, один-два опытных разработчика, инженер по тестированию (особенно если проблемы тестирования очень неясны) и, вероятно, представители клиента или пользователей. Первый этап — развитие концепции до такого состояния, чтобы ею можно было руководствоваться в ходе фазы анализа и планирования.

Это легко сказать, но нелегко сделать. Причина в том, что в этой работе участвуют представители интересов заказчика. Они — люди с опытом. Они хотят объединить различные представления и не соглашаются на компромисс или половинчатое решение. Кроме всего прочего, у них недостаточно времени для раздумий над этой (очень сложной) проблемой. Но дело в том, что мы не ищем консенсуса, мы ищем нужные ответы. Нужные нам ответы исходят от лидера, от человека, от-

ветственного за управление финансами, которые будут вложены в разработку, особенно если его информировали знающие специалисты.

В том случае, если продукт похож на то, что уже делалось раньше, базовой команде достаточно будет лишь установить, что такое сходство существует, причем это займет у них всего несколько дней. Некоторые фазы анализа и определения требований занимают всего один день (например, второй цикл для существующего несложного продукта). Для целинного проекта может понадобиться несколько месяцев.

В случае очень специфических систем фаза анализа и планирования требований может оказаться дорогой и длительной.

Задание критериев оценки

Когда менеджер проекта имеет всю информацию, включенную в подробный план первой итерации, он устанавливает критерии оценки, которые должны показать, что итерация выполнила свою задачу. Подробный план первой итерации может быть грубоват, поскольку информации еще немного. Если это так, критерии оценки могут быть довольно общими. В ходе итерации, по мере получения дополнительных знаний, менеджер получает возможность уточнить критерии. Однако для оценки достижения четырех целей фазы анализа и планирования требований нам будет достаточно весьма общих критериев.

Определение назначения и области применения системы. Исходная концепция, несомненно, должна содержать какие-либо идеи по поводу области применения, но не должна точно ее определять. В фазе анализа и определения требований в проекте проводится линия, отделяющая то, что входит в предлагаемую систему, от того, что в нее не входит. После этого определяются внешние актанты — как другие системы, так и сотрудники, с которыми система взаимодействует. Это определяет на верхнем уровне природу взаимодействий. Сюда в результате входят следующие пункты:

- Понятно ли, что происходит внутри системы?
- Всех ли актантов удалось определить?
- Установлена ли общая природа интерфейсов (пользовательских интерфейсов и коммуникационных протоколов) для работы с актантами?
- Можно ли выделить некоторый фрагмент области действия системы в отдельную функционирующую систему?

Разрешение двусмысленностей в требованиях, необходимых для этой фазы. Требования, представленные в начале фазы анализа и планирования требований, могут варьироваться от общей концепции до многостраничных описаний. Однако эти исходные требования, вероятно, содержат двусмысленности. Они оцениваются по следующим показателям:

- Было ли выделено и детализировано ограниченное число требований (функциональных или нефункциональных) к вариантам использования, необходимых для выполнения целей этой фазы?
- Были ли выделены и детализированы требования, относящиеся к Дополнительным требованиям (см. раздел «Дополнительные требования» главы 6)?

Создание возможной архитектуры. Руководствуясь опытом, сотрудники в ходе фазы анализа и планирования могут довольно быстро сосредоточиться на новых или требующих беспрецедентной производительности функциях. Между тем они должны сделать выбор, который может подвергнуть опасности разработку всей системы. Для этих новых функций они должны разработать как минимум одну рабочую архитектуру. Критерии ее создания:

- Соответствует ли она требованиям пользователей?
- Удобно ли с ней работать? (Рассматривайте этот критерий в свете дальнейшего развития возможной архитектуры. Поскольку прототип создаваться не будет, описание вероятной архитектуры оценивается по тому, что она «обещает» принести в дальнейшем.)

Чтобы ответить на эти вопросы, мы должны рассмотреть несколько спорных вопросов. Может ли архитектура соответствующим образом использовать технологии (базы данных, сети и т. п.), которые мы будем в нее встраивать? Может ли она быть эффективной? Может ли она использовать ресурсы? Будет ли она надежной и устойчивой к сбоям? Будет ли она прочной и гибкой? Можно ли будет легко дополнить ее при добавлении новых требований?

Снижение наиболее опасных рисков. Наиболее опасными рисками считаются те, которые, не будучи сниженными, способны подвергнуть опасности успешность проекта. Пункты для оценки результата этой деятельности включают в себя:

- Определены ли все наиболее опасные риски?
- Снижены ли найденные риски или существует ли план по их снижению?

«Снижен» вовсе не обязательно означает, что наиболее опасный риск на этой фазе полностью устранен. Это может означать, что клиент согласен скорее модифицировать соответствующие требования, чем столкнуться с вероятностью срыва проекта. Это может означать, что команда разработчиков видит способ избежать риска, однако он не будет детально прорабатываться до следующей фазы. Другими словами, это может означать, что команда разработчиков знает способ уменьшить вероятность реализации риска или облегчить его последствия, если он реализуется. Это может означать создание плана на случай непредвиденных обстоятельств, описывающего действия, которые будут предприняты в такой ситуации. Ни один из этих пунктов не сводится к простому механическому принятию решения, особенно в предельных ситуациях. Основная задача фазы анализа и планирования требований — задать в процессе разработки программного обеспечения точку, в которой менеджер проекта, старшие руководители и отвечающие за финансирование заинтересованные лица (включая заказчиков) могли бы принять деловое решение.

Обсуждение ценности исходного бизнес-плана. Вопрос для оценки звучит так: Достаточно ли хорош исходный бизнес-план, чтобы его стоило в дальнейшем уточнять в ходе проекта?

Исходный бизнес-план, охватывающий различные области проекта, используется для точной оценки проекта ответственными руководителями. В знакомой области вы можете приблизительно назвать правильные цифры для бизнес-плана к концу фазы анализа и планирования требований. Однако в новых, трудных областях оценка будет базироваться на том широком диапазоне значений, который вам только удастся получить.

Типичный поток работ итерации на фазе анализа и планирования требований

В ходе анализа и планирования требований мы осуществляем три вида деятельности. Первый — планирование итераций — описан в разделах «Планирование предваряет деятельность», «Риски влияют на планирование проекта», «Расстановка приоритетов вариантов использования» и «Требуемые ресурсы» главы 12, а также в разделе «Расширение концепции системы» данной главы; второй — это пять базовых рабочих процессов, которые кратко обсуждаются в следующем подразделе и подробно — в разделе «Выполнение основных рабочих процессов»; а третий — выбор среды разработки, подходящей для проекта, — рассматривается в подразделе «Погружение проекта в среду разработки». Кроме того, в разделе «Определение исходных деловых перспектив» в этой главе описывается исходный бизнес-план, а в разделе «Определение итераций в фазе анализа и планирования требований» — оценка результатов этой фазы.

Введение в пять основных потоков работ

Основная цель фазы анализа и планирования требований — создание бизнес-плана — документа, из которого следует, есть ли смысл продолжать проект с деловой точки зрения. Для достижения этой цели мы должны рассмотреть область действия рассматриваемой системы, создать набросок архитектуры, определить риски, опасные для успешного выполнения проекта, и создать предварительный план их снижения. Если мы рассматриваем систему нового типа, мы можем захотеть продемонстрировать заинтересованным лицам ее работоспособность путем построения концептуального прототипа, который впоследствии обычно выбрасывается. Прототипы могут также выборочно использоваться для управления рисками: определения областей с высокой степенью риска и прототипирования ключевых частей систем с хитроумной функциональностью или известными проблемами производительности и пропускной способности. Так, для защищенной от сбоев финансово-торговой системы мы можем пожелать как можно раньше создать прототип механизма защиты от сбоев.

Большая часть работы в фазе анализа и определения требований осуществляется в рамках первого рабочего процесса, определения требований, как показано на рис. 13.1, также происходит некоторая деятельность в рабочих процессах анализа и проектирования. Относительные размеры прямоугольников на рисунке демонстрируют, какие рабочие процессы привлекают к себе минимальное и максимальное внимание. На заключительные рабочие процессы, реализацию и тестирование, приходится очень мало работы. В этой фазе наше дело — представить новые концепции, а не убеждаться, что исследуемый прототип работает абсолютно правильно.

Системный аналитик находит варианты использования и актантов, которые определяют область действия системы. Архитектор присваивает этим вариантам использование приоритеты и выбирает те из них, которые нужны для возможной архитектуры. Он создает базовое описание возможной архитектуры. Спецификатор вариантов использования детализирует некоторые сценарии ранее определенных вариантов использования. Детальное описание вариантов использования необходимо для понимания области действия системы и ее возможной архитектуры, для понимания наиболее важных рисков, то есть для создания исходного бизнес-

плана. Одним из результатов этой деятельности станет первая модель вариантов использования, если проект целинный, или новая версия модели вариантов использования, если дорабатывается существующая система. Мы можем также создать список приоритетов вариантов использования, обсуждавшийся в главе 12.

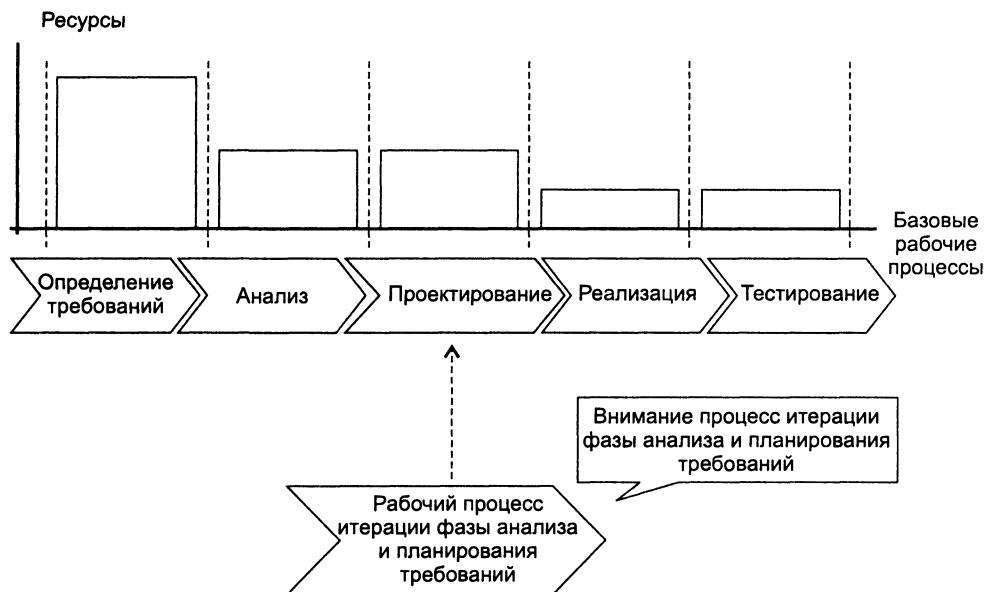


Рис. 13.1. На итерациях фазы анализа и определения требований осуществляются пять базовых рабочих процессов

В ходе рабочего процесса анализа мы создаем первую модель анализа для большинства вариантов использования или сценариев вариантов использования, которые мы выявили в ходе анализа и определения требований. Модель анализа необходима для четкого понимания вариантов использования. Она важна также и для понимания того, что лежит в основании нашего первого наброска архитектуры.

Если система новая или беспрецедентная, команда разработчиков в ходе анализа и планирования требований может создать исследовательский прототип, чтобы быстро получить средство для демонстрации ключевых идей разработки. Такой прототип, направленный на демонстрацию вошедших в разработку концепций, не включается в последующую реализацию. Другими словами, он, вероятнее всего, будет выброшен. В некоторых случаях вывода о существования алгоритма для проведения новых вычислений будет достаточно, чтобы продемонстрировать команде разработчиков отсутствие у инженеров по компонентам проблем с реализацией алгоритмов в компонентах, имеющих необходимую производительность.

Итерация может быть закончена в форме «описания возможной архитектуры», включая наброски представлений моделей, сразу же, как только менеджер проекта определит возможную архитектуру. При этом он должен ориентироваться на то, чтобы функции были реализуемы, а риски — меньше критических или требующих понижения. В этом случае с точки зрения предыдущего опыта и заинтересованных лиц архитектура оценивается как удачная.

Погружение проекта в среду разработки

Среда разработки включает в себя процесс, утилиты для поддержки этого процесса и сервисы, обслуживающие проект. Она включает в себя конфигурирование и развитие процесса, выбор, получение и изготовление утилит, технические сервисы, обучение и руководство.

В понятие «утилиты» мы включаем утилиты для поддержки основных рабочих процессов: определения требований, анализа, проектирования, реализации и тестирования, а также утилиты административного назначения — для управления проектом (планирования, оценки, отслеживания), конфигурирования, управления изменениями, генерации документов и онлайновой обработки. Вдобавок к этим утилитам, которые представляют собой средства общего назначения и получаются от поставщиков, для проекта могут быть разработаны специальные утилиты, как в рамках проекта, так и в рамках компании, для поддержки нестандартных требований. Под сервисами обычно понимаются системное администрирование, резервное копирование данных и связь.

В фазе анализа и планирования требований среда разработки организации подрабатывается так, чтобы она могла поддерживать начинающийся проект. Эта работа продолжается и в фазе проектирования, по мере того, как у проекта появляются дополнительные требования к средствам поддержки и другим сервисам. Организация проекта обеспечивает доводку среды разработки параллельно с прочей работой, осуществляющейся в этой фазе.

Определение критических рисков

В главе 12 мы посвятили разделы «Риски влияют на планирование проекта» и «Расстановка приоритетов вариантов использования» определению и снижению рисков, способных угрожать разработке. В фазе анализа и планирования требований мы занимаемся определением и снижением или планированием снижения особо опасных рисков. Особо опасными мы считаем риски, способные сделать проект неосуществимым. В подразделе «Задание критериев оценки» мы рассматривали, что следует предпринимать для снижения особо опасных рисков. Крайне важно обнаружить риски такого уровня в ходе фазы анализа и планирования требований. Если мы обнаруживаем подобный риск и не в состоянии найти способ его понижения или придумать план на случай его реализации, мы должны обсудить вопрос закрытия проекта.

Выполнение основных рабочих процессов от определения требований до тестирования

В этом разделе мы детально опишем, что нам следует сделать в ходе фазы анализа и планирования требований. Описание работы в этой фазе для целинного, то есть нового продукта, начинается с чистого листа. Разработка существующего продукта обычно выглядит значительно проще.

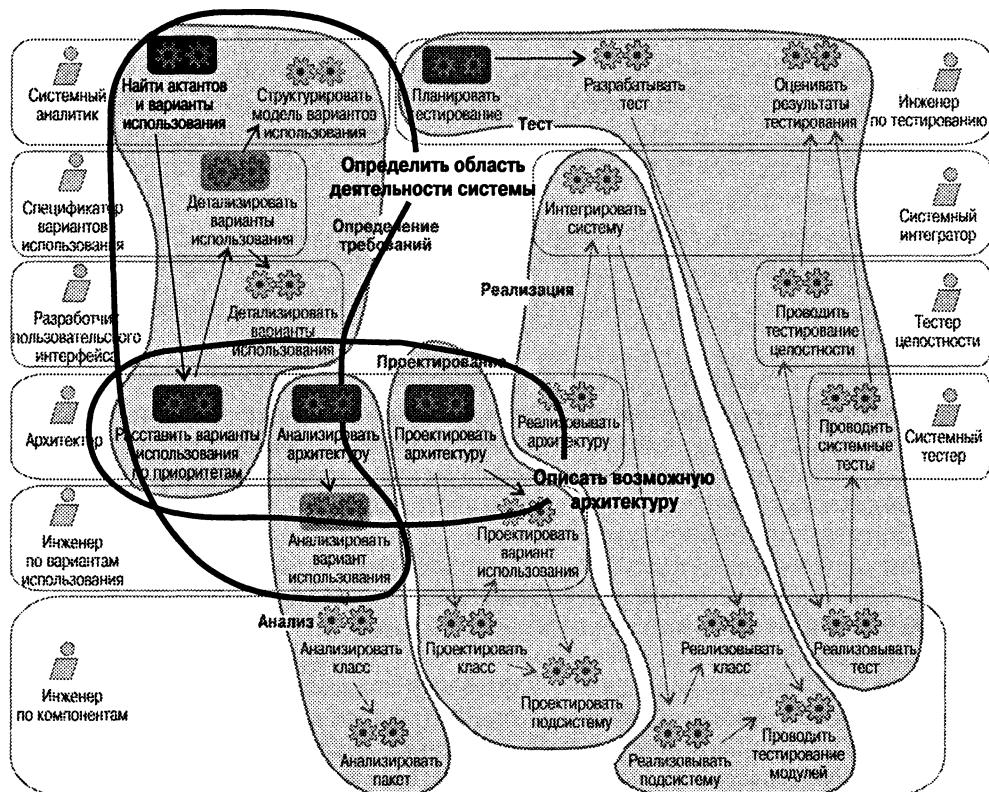


Рис. 13.2. Контуры показывают на основную деятельность в ходе фазы анализа и планирования требований:
определение области деятельности системы
и описание возможной архитектуры. Несмотря на детальность этого рисунка, это только схематическое изображение сотрудников и артефактов, описанных в главах 6–11

Каждая итерация — это маленький водопад, в ходе которого производятся все пять базовых рабочих процессов, от определения требований до тестирования. В части 2 мы посвятили каждому из базовых рабочих процессов по главе, а первому из них, определению требований, даже две. В этом разделе мы посвятим каждому из пяти рабочих процессов по одному подразделу, поскольку собираемся рассмотреть их в контексте фазы анализа и планирования требований. Чтобы указать центр внимания, мы изобразили на рис. 13.2 два контура. Один из них заключает в себе работы, связанные с определением области деятельности; а другой — с выбором возможной архитектуры. Несмотря на детальность рис. 13.2, это только схематическое изображение сотрудников и артефактов, описанных в главах 6–11.

Названия сотрудников, показанные на этом рисунке, соответствуют ролям, которые исполняют члены группы в ходе разработки. В этой фазе, однако, поскольку команда еще мала, на одного человека может приходиться несколько ролей.

Определение требований

В фазе анализа и планирования требований акцент делается в основном на первом из базовых рабочих процессов, процессе определения требований. Этот рабочий процесс включает в себя определение и детализацию вариантов использования, которые требуются на данной фазе. В него входят следующие операции, перечисленные в главе 6:

1. Перечисление требований — кандидатов в список предложений.
2. Осознание контекста системы.
3. Определение разумных функциональных требований в виде вариантов использования.
4. Определение связанных с ними нефункциональных требований.

Перечисление требований—кандидатов. Предложения по функциональности системы часто порождаются опытом, который пользователи или заказчики получили при работе с предшествующим или похожими системами. В случае массового продукта новые предложения порождаются требованиями рынка. Кроме того, сотрудники организации-разработчика сами вносят множество предложений. Некоторым предложениям дает начало взаимодействие организации-разработчика с пользователями, некоторым — изучение рынка. Предложения, полученные из всех этих источников, становятся кандидатами в требования и вносятся в список предложений, описанный в главе 6.

Осознание контекста системы. В фазе анализа и планирования требований может быть использована бизнес-модель (или модель предметной области), если она есть у клиента. Если модели у клиента в настоящий момент нет, мы должны подвигнуть его к созданию такой модели, невзирая на то, что обычно это требует времени и ресурсов, значительно превосходящих выделенное на единичный проект. Будьте агрессивны.

Вы должны определить варианты использования технической или бизнес-системы. Это скажет вам, какой процесс вы поддерживаете. Вы должны определить сотрудников и основные результаты работы (рабочие модули и бизнес-сущности). Всего до начала выяснения деталей моделирования соответствующих вариантов использования вы должны проделать от 50% до 70% работы по созданию бизнес-модели. Вы не можете разрабатывать программное обеспечение, не зная процесса, который оно должно поддерживать.

Определение функциональных требований. Функциональные требования представляются в виде вариантов использования. Это рассматривается в главе 7 и следующем подразделе.

Определение нефункциональных требований Нефункциональные требования, которые относятся к конкретным вариантам использования, присоединяются к своим вариантам использования (раздел «Обзор процесса определения требований» главы 6 и подраздел «Специальные требования» главы 7). Те из них, которые специфичны для объектов бизнес-модели или модели предметной области, вносятся в гlosсарий и добавляются к модели вариантов использования («Обзор процесса определения требований», глава 6), более общие, которых гораздо меньше, превращаются в дополнительные требования (раздел «Дополнительные требования», глава 6). Некоторые из общих нефункциональных требований крайне важны для выбора платформы, системного программного обеспечения, программ-

ного обеспечения среднего уровня. Они могут быть очень важны для этой фазы (см. раздел «Варианты использования и архитектура» главы 4).

Определение требований в виде вариантов использования

А теперь мы вернемся к рис. 13.2 и обсудим указанные там виды деятельности. В этом пункте мы рассмотрим процесс определения требований в виде вариантов использования в терминах видов деятельности, изображенных на рис. 13.2.

Определение актантов и вариантов использования (см. подраздел «Деятельность: Нахождение актантов и вариантов использования» главы 7). Получение «полных» требований, или завершение модели вариантов использования, превышает возможности фазы анализа и планирования требований. В ходе этой фазы сотрудники должны, во-первых, отобрать те варианты использования, которые необходимы для выполнения работ этой фазы, и во-вторых, детализировать их в соответствии с описанием в главе 7. Как определить эти варианты использования, было описано в разделе «Расстановка приоритетов вариантов использования» главы 12.

В ходе фазы анализа и планирования следует позаботиться об уменьшении объема работы, детализируя только те варианты использования, которые необходимы для выполнения задач этой фазы. Игнорируются те ветви вариантов использования и варианты использования целиком, которые недостаточно важны для области действия системы и возможной архитектуры, не содержат особо опасных рисков и слабо влияют на первоначальный бизнес-план.

Отбором вариантов использования, которые будут рассмотрены на фазе анализа и планирования требований, занимается менеджер проекта вместе с системным аналитиком и архитектором. Системный аналитик делает вывод о принадлежности варианта использования к области применения системы. Архитектор излагает свои размышления обо всех вариантах использования для исключения тех из них, которые в дальнейшем будут не важны. Архитектор должен выделить особо опасные и серьезные риски и определить те варианты использования, которые необходимы для планирования работ по архитектуре. Эта работа должна быть закончена в следующей фазе в первую очередь. Величина требуемых усилий в тех частях системы, где архитектор рассматривает варианты использования предшествующей системы, зная, что их важность для этой фазы минимальна, должна быть очень невелика. Архитектор должен просмотреть все варианты использования как минимум для того, чтобы исключить их из текущего рассмотрения.

Расстановка приоритетов вариантов использования (см. подраздел «Деятельность: Определение приоритетности вариантов использования» главы 7). Далее архитектор, рассматривая модель вариантов использования и получая данные от менеджера проекта, создает план проекта/план итераций. Эта работа происходит параллельно базовым рабочим процессам каждой фазы; то есть планировать будущие итерации можно одновременно с детализацией обнаруженных ранее вариантов использования. План итераций — это итоговый результат, демонстрирующий приоритетность вариантов использования в соответствии с целями данной фазы.

Детализация вариантов использования (см. подраздел «Деятельность: Детализация вариантов использования» главы 7). Принимая необходимость сокращения объема работ, мы, тем не менее, повторим свои слова о том, как важно полностью детализировать все относящиеся к делу ветви вариантов использования, которые мы исследуем на фазе анализа и определения требований, — вариантов ис-

пользования, необходимых нам для определения области применения системы, планирования снижения особо опасных рисков и создания базового уровня архитектуры. Очень часто сотрудники уверены, что они определили все необходимые требования, а на самом деле некоторые ключевые варианты использования оказываются незамеченными. Они имеют привычку полагать, что преимущества детализации необходимых вариантов использования не окупаются. Это неверно. Они окупаются всегда. Наличие деталей действительно необходимо.

Наша цель — понять, куда мы идем. Неважно, однако, продирались ли мы для этого через громадное количество вариантов использования. Обычно, поскольку на этой фазе не обязательно строить прототип архитектуры, нам вообще не нужно заниматься реализацией и тестированием. Однако, если мы хотим продемонстрировать базовые идеи при помощи одноразового прототипа, мы должны обсудить необходимую для создания прототипа долю **объема вариантов использования** (приложение В), который рассмотрим позднее. Нам будет достаточно детализировать порядка 10% общего объема вариантов использования модели вариантов использования.

Это утверждение означает, что мы рассматриваем множество вариантов использования, но детализируем только часть. Мы занимаемся деталями только тех вариантов использования, которые необходимы нам в данный момент. Например, если мы выбрали 50% всех возможных вариантов использования как потенциально нужных на этой фазе, а затем в итоге выбрали для дальнейшей детализации в среднем только 20% сценариев каждого из них, то мы рассмотрим в деталях 10% всей массы вариантов использования. Смысл этого отбора в понижении затрат и сжатии графика выполнения фазы анализа и планирования. Разумеется, проценты указаны для индивидуальных проектов и зависят от того, насколько проект сложен или необычен.

Как указывалось в главе 7, все необходимые на этой фазе функциональные требования представляются в виде вариантов использования.

Создание прототипов интерфейсов пользователя (см. подраздел «Деятельность: Создание прототипа интерфейса пользователя» главы 7). На этой фазе интереса не представляет.

Структурирование модели вариантов использования (см. подраздел «Деятельность: Структурирование модели вариантов использования» главы 7). Эта деятельность здесь не рассматривается.

Анализ

Цели рабочего процесса анализа — проанализировать требования, уточнить их и структурировать в объектную модель, которая послужит первой зарубкой для создания модели проектирования. Результатом рабочего процесса анализа на этой фазе будет базовая модель анализа. Мы используем эту модель анализа для более точного определения вариантов использования и для помощи в задании направления возможной архитектуры — более полно проработанный базовый уровень архитектуры будет задачей фазы проектирования. Это означает, разумеется, что в фазе анализа и определения требований будет создана крайне малая часть модели анализа (мы говорим о 5%). На самом деле мы можем охарактеризовать модель анализа в фазе анализа и проектирования требований как первую грубую зарубку. Это первый шаг создания архитектурного представления модели анализа.

Анализ архитектуры (см. подраздел «Деятельность: Анализ архитектуры» главы 8). Задача на фазе анализа и проектирования требований — отсортировать варианты использования или сценарии, которые нам нужно аккуратно рассмотреть для выполнения задач данной фазы, прежде всего для того, чтобы понять и уточнить их. Получив исходный набор вариантов использования и сценариев, архитектор строит для этих частей системы первую упрощенную версию модели анализа. Она не должна быть дорогой и не обязана быть завершенной. Проект не предполагает легкого ее построения в следующей фазе. Фактически она может быть полностью переделана с сохранением только общего направления.

Анализ варианта использования (см. подраздел «Деятельность: Анализ варианта использования» главы 8). В некоторых ситуациях бывает недостаточно простого рассмотрения одного за другим всех вариантов использования. В модели вариантов использования варианты использования рассматриваются только по одному. Однако в реальности варианты использования совместно используют ресурсы системы — базы данных, вычислительные мощности и т. п. Модель анализа позволяет найти подобные совместно используемые ресурсы. Поэтому для разрешения подобных конфликтов мы часто вынуждены проводить довольно глубокий анализ.

В рабочем процессе этой итерации мы можем анализировать, а затем уточнять некоторые из вариантов использования (10% от объема вариантов использования), о чем подробнее рассказывали в подразделе «Определение требований в виде вариантов использования». Мы можем детализировать половину из них, что составит примерно 5% от общего объема вариантов использования.

Анализ класса и Анализ пакета. Если эти работы в текущей фазе и проводятся, то в минимальном объеме.

Проектирование

Первичная цель рабочего процесса проектирования в этой фазе — создать набросок модели проектирования возможной архитектуры для включения его в предварительное описание архитектуры.

Если вы нуждаетесь в разработке демонстрационного прототипа, то создаете его, используя готовые модули, языки четвертого поколения или другую технику быстрой разработки, которая позволяет продемонстрировать основную идею. Демонстрация непривычных пользовательских интерфейсов и алгоритмов делает их понятными для всех заинтересованных лиц, мнение которых учитывается при решении вопроса о продолжении разработки. Мы показываем этот прототип соответствующим пользователям, чтобы убедиться, что он соответствует их ожиданиям и, при необходимости, для выяснения, какие изменения необходимо внести для лучшего соответствия их ожиданиям.

Проектирование архитектуры (см подраздел «Сотрудник: Архитектор» главы 9). Цель рабочего процесса проектирования состоит в том, чтобы разработать предварительный или первый набросок модели проектирования, сделать первый шаг в построении архитектурного представления модели проектирования, которая осуществляет варианты использования (мы определили их раньше, в рабочем процессе определения требований) посредством коопераций между подсистемами/классами. Мы стараемся искать интерфейсы (максимум определяя некоторые из их операций) между подсистемами/классами, даже если это всего лишь набросок в процессе проектирования. Эти интерфейсы важны тем, что они — основа ар-

хитектуры. Кроме того, мы нуждаемся в определении обобщенных механизмов проектирования, которые станут базовыми уровнями подсистем, реализующих обнаруженные нами варианты использования. Мы выбираем системное программное обеспечение и каркасы, которые будут использоваться в программном обеспечении среднего уровня (см. раздел «Деятельность: Проектирование архитектуры» главы 9). Мы создадим эту базовую модель проектирования, даже понимая, что она будет крайне приблизительной, и что в этой фазе мы дойдем только до описания возможной архитектуры.

Модель проектирования реализует не только функциональные требования, представленные зафиксированными вариантами использования, но и нефункциональные требования, такие как производительность, которые могут сопровождаться рисками.

Если представляемая система будет загружаться на узлы сети, архитектор должен спроектировать уменьшенную версию модели развертывания, ограниченную, например, по производительности действующих узлов или соединений между узлами. В этом месте менеджер проекта должен потребовать у инженера по вариантам использования смоделировать части двух узлов и взаимодействий между ними в случае прихода вызова.

Проектирование вариантов использования (см. раздел «Сотрудник: Инженер по вариантам использования» главы 9). В фазе анализа и планирования требований работы по проектированию вариантов использования сведены к минимуму.

Проектирование классов и проектирование подсистем. Если что-то из этого выполняется на фазе анализа и планирования требований, то в минимальной степени.

Реализация

Уровень активности в этом рабочем процессе зависит от того, какое решение принял ранее менеджер проекта. Посчитал ли он нужным прервать фазу анализа и определения требований после описания возможной архитектуры?

С одной стороны, существует мнение, что нельзя быть уверенным в правильном функционировании возможной архитектуры, пока вы (и заинтересованные лица) не видели работающего прототипа. Мы не можем быть уверены в том, что мы устранили риск до того, как часть прототипа, к которой он относится, не начнет работать. С другой стороны, для сохранения времени и числа сотрудников, занятых на фазе анализа и планирования, минимальными, мы можем прервать рабочий процесс фазы сразу после того, как получим описание архитектуры, которое покажется нам работоспособным. Определить, как глубоко следует разрабатывать возможную архитектуру — дело менеджера проекта.

В нормальной ситуации он завершает фазу после получения описания возможной архитектуры. В этом случае в рабочем процессе реализации нет необходимости.

Однако может потребоваться демонстрационный прототип (одноразовый прототип). В этом случае проект перейдет к рабочему процессу реализации, который, однако, будет коротким.

Тестирование

Параллельно с деятельностью по анализу, проектированию и реализации, как показано на рис. 13.2, с общей природой предлагаемой системы знакомятся инжене-

ры по тестированию. Они обсуждают необходимое тестирование и разрабатывают некоторые пробные планы тестирования. Однако в ходе анализа и планирования требований сколько-нибудь существенной работы по тестированию не ведется. Это иллюстрируется рис. 11.2 и 13.1. Демонстрационный прототип предназначен в основном для показов, а не для работы. Однако менеджер проекта может счесть полезным провести некоторое количество тестов.

Определение исходных деловых перспектив

Когда в конце фазы анализа и определения требований мы приходим к тому, что проект содержит возможную архитектуру и может справиться с наиболее опасными рисками, наступает время перевести концепцию в экономические категории, чтобы определить потребности проекта в ресурсах, капиталовложениях, прогнозируемого дохода и потребности рынка (или внутренней потребности). Одна сторона бизнес-плана — это бизнес-предложение, а другая — экономическая выгода от возможного использования продукта.

Формулировка бизнес-предложения

Оценочные формулы, лежащие в основе бизнес-предложения, обычно находятся в зависимости от «размера» конечного продукта. В конце фазы анализа и планирования требований, однако, этот размер (напомним, что обработана лишь малая часть всего объема вариантов использования) может существенно, например, наполовину, отличаться от итогового размера продукта. Соответственно и оценка в этот момент может отличаться от реальной стоимости в конце проекта на те же 50%.

Применим, например, варианты использования в качестве шкалы для оценки размера. Усилия, необходимые для проектирования, реализации и тестирования вариантов использования, находятся в диапазоне от сотен до нескольких тысяч человеко-часов. Конкретный вариант использования занимает свое место в этом диапазоне под действием нескольких факторов:

- *Стиль.* Если разработчики реализуют в варианте использования множество предложений, он будет более мощным, чем обычный вариант использования, но потребует большего количества человеко-часов.
- *Сложность разрабатываемой системы.* Большая сложность системы при одинаковом размере будет причиной повышенных затрат. Подумайте, не стоит ли вам упростить систему путем снижения ее функциональности.
- *Размер.* Варианты использования небольшой системы обычно реализуются проще, чем варианты использования крупного проекта. Фактор размера системы может изменить затраты человека-часов на один вариант использования более чем на порядок.
- *Тип приложения.* Мощные системы реального времени, предназначенные для работы в распределенной среде, например высокозащищенные/постоянно доступные системы, вносят важный вклад в затраты человека-часов на один вариант использования, увеличивая их в 3–5 раз по сравнению со стандартными приложениями архитектуры клиент/сервер.

Этот список не полон. Организации-разработчики и заинтересованные лица могут исследовать переменные, используемые для оценки, для построения собственной оценочной базы. В отсутствие оценочной базы мы советуем проводить оценки традиционным способом. Затем прибавим к ним оценки затрат и времени на изучение новых подходов, новых используемых утилит и адаптацию других новых свойств Унифицированного процесса. После одного или двух случаев работы с Унифицированным процессом организации обнаруживают, что их затраты существенно снизились, время создания продукта от идеи до выхода на рынок просто потрясающе сократилось, а качество и надежность систем возросли.

Команды, работающие над проектом, в конце фазы анализа и планирования требований обычно не в состоянии создать итоговый бизнес-план. Собрано еще недостаточно фактов. Кстати, Унифицированный процесс не используется для «предложения» до конца фазы проектирования. Это происходит потому, что мы считаем бизнес-план, полученный в фазе анализа и планирования требований, предварительным бизнес-планом. Предварительный бизнес-план хорошо подходит только для уточнения — на самом деле, очень грубого — действий в фазе проектирования. Например, при текущем положении на рынке персональных компьютеров вам недостаточно будет провести фазу анализа и планирования, чтобы обнаружить, что бизнес-планы по поставке на рынок нового типового текстового редактора обречены на провал. В случае проектов, которые действительно рассматриваются компаниями, однако, бизнес-план всегда является результатом многочисленных исследований.

Требования к персоналу и график первых итераций сами по себе тоже должны иметь финансовую поддержку. Требования, которые менеджеры проектов предъявляют к первым итерациям, — получить данные, на которых будут основаны дальнейшие усилия и графики последующей работы, — часто не совсем понятны. Поэтому организация должна найти подходящий способ измерения результатов фазы анализа и планирования требований. Эти значения послужат им базой для оценки необходимого числа итераций первой фазы следующего проекта. Позднее эти значения могут быть изменены в соответствии с информированным мнением о том, что следующий проект, например, сложнее или проще базового.

Оценка доходности инвестиций

Вышеприведенная оценка представляет собой одну из сторон бизнес-плана. Другая его сторона, вычисление прибыли, которую принесет программное обеспечение, обходится без красивых формул. Для программ, которые поставляются на рынок, число продаваемых пакетов, цена, по которой следует продавать продукт и период, в течение которого будут происходить продажи, рассчитываются маркетологами и оцениваются руководством. В случае программ для внутреннего употребления следует предложить отделу, заинтересованному в разработке, оценить ожидаемую им экономию. Ошибка определения обычно немаленькая, но при оценке доходности инвестиций эти данные можно, по крайней мере, взять за основу.

Для коммерческих продуктов бизнес-план должен включать в себя набор предположений о проекте и порядок величины доходности инвестиций, ожидаемой в том случае, если эти предположения сбудутся. Для подстраховки, чтобы быть уверенным, что бизнес-план вообще будет выгодным, руководство часто вписывает в него большие ограничения прибыльности.

В этот момент мы в состоянии создать общий бизнес-план. Это значит, что наш бизнес-план будет выглядеть вполне выгодным или достойным того, чтобы начать проект. Похоже, что проект окупится. Для того, чтобы закончить фазу анализа и планирования требований, нам следует пытаться получить не точные цифры, а информацию о том, что система будет рентабельна для организации-разработчика и клиентов. Сейчас мы еще не имеем детальной информации, необходимой для полного завершения финансовой части бизнес-плана. Чтобы закончить его, мы должны перейти к стадии наличия обоснованных цен и точных предложений и графиков. Мы проверяем сделанные допущения вплоть до окончания фазы планирования, пока не определим проект более точно.

Определение итераций в фазе анализа и планирования требований

В начале фазы анализа и определения требований, как только нам станет доступна адекватная информация, менеджер проекта задает критерии оценки окончания первой итерации и всей фазы. Мы говорили об этом в подразделе «Задание критериев оценки» данной главы. По мере того как фаза идет к концу, менеджер проекта также собирает группу (которая может состоять из пары человек) для оценки критериев. В группу оценки обычно включается представитель заказчика или пользователей. Для проектов определенного размера может оказаться необходимо включить в группу представителей всех заинтересованных лиц. Некоторые из критериев первоначального плана могут оказаться не выполненными. Вот примеры таких невыполненных критериев:

- Расширение модели вариантов использования до состояния, недостаточного для этой фазы.
- Неполная разработка концептуального прототипа, исключающая возможность его демонстрации.
- Подозрение, что найдены не все особо опасные риски.
- Недостаточное снижение или неучтенность в плане обнаруженных особо опасных рисков.

Менеджер проекта переносит невыполненные критерии на следующую итерацию и модифицирует план и график в соответствии с потребовавшейся дополнительной итерацией. Так, для выполнения последней итерации в команду может потребоваться включить дополнительных сотрудников с определенным опытом или базовыми знаниями.

Главный результат выполнения фазы анализа и планирования требований — глобальное решение продолжать работу или прекратить ее. Мы проверяем цели этой фазы — область действия, особо опасные риски, возможную архитектуру — и решаем, стоит ли продолжать работу. Мы можем выждать с решением «продолжать работу» до первой главной вехи. Прекратить ее мы должны сразу же, как только получим факты, оправдывающие такое решение, — не тратя дополнительных усилий. Однако это решение не может быть произвольным. Продолжение или прекращение работ требует обсуждения с заинтересованными лицами, особенно

с инвесторами и представителями пользователей. После этого в случае возможно-го прекращения работ заказчик может найти способ обойти указанные препятствия к разработке.

Планирование фазы проектирования

К концу фазы анализа и планирования требований мы начинаем интересоваться затратами и планом на фазу проектирования и планировать их. Мы хотим спроектировать около 80% всех требований, мы не хотим пропустить ни одного важного момента, связанного с архитектурой. Нам следует сделать это для того, чтобы быть в состоянии создать уточненное предложение, поскольку в нашем распоряжении теперь не только ограниченные данные предыдущей фазы, а также, чтобы иметь возможность использовать их для выбора архитектуры. Восемьдесят процентов общего объема — это приблизительная доля вариантов использования, необходимых нам для подготовки бизнес-предложения. Из этих 80% мы должны проанализировать 50%, чтобы добиться хорошего понимания требований.

Для разработки базового уровня архитектуры нам следует описать 80%, чтобы быть уверенными, что мы не пропустили ничего важного. Из этих 80% мы выбираем значимую часть вариантов использования, на основе которых создаем проект базового уровня архитектуры. Эти значимые варианты использования — значительно меньшая доля общего объема вариантов использования, и 80%, о которых мы говорим, превращаются в 40% вариантов использования и приблизительно 20% каждого из них. Результат этих двух дроблений по отношению к общему объему вариантов использования составит всего 8%. Другими словами, для того, чтобы показать нам все, что мы в этот момент хотим знать о значимых вариантах использования, достаточно менее 10% общего объема вариантов использования. Мы используем эту долю для управления работой по созданию базового уровня архитектуры, в который входят описание архитектуры и версии всех моделей.

При этом мы выполняем итерации, необходимые для осуществления фазы проектирования, если их требуется больше одной. Мы можем предполагать, что уложимся в одну итерацию, но в сложных случаях их понадобится больше. Мы решаем, что следует сделать в каждой итерации, каковы требования к реализации и тестированию и, таким образом, какие риски мы сможем уменьшить.

Опыт показывает, что в фазе проектирования осуществляется значительная доля проектирования и реализации, например, при разработке концептуальных прототипов, которые на следующей фазе использоваться не будут.

У нас накопились некоторые трудности с отслеживанием всех этих процентов. Поэтому для упрощения работы мы собрали их все в табл. 13.1.

Отметим, что числа в таблице — только приблизительные индикаторы. Мы различаем определение варианта использования в нескольких словах и более полное его описание, которое понимаем под видом деятельности «Детализация вариантов использования» (глава 7). Анализ вариантов использования выполняется в ходе «Анализа вариантов использования» (глава 8). Правый столбец таблицы показывает, какая доля вариантов использования вошла в базовый уровень к концу фазы.

Таблица 13.1. Работа с вариантами использования

Завершенность бизнес-модели	Определение вариантов использования	Описано, от общего объема вариантов использования	Проанализировано, от общего объема вариантов использования	Спроектировано, реализовано и оттестировано, от общего объема вариантов использования
Фаза анализа и планирования требований	50–70%	50%	10%	5% Небольшая доля, для концептуального прототипа
Фаза проектирования	Почти 100%	80% или более	40–80%	20–40% Менее 10%
Фаза построения	100%	100%	100% если поддерживается	100%
Фаза внедрения				

Результаты фазы анализа и планирования требований

Результатом фазы анализа и планирования требований является:

- Список предложений.
- Первая версия бизнес-модели (или модели предметной области), которая описывает контекст системы.
- Первые наметки моделей — модели вариантов использования, модели анализа и модели проектирования. Наброски моделей реализации и тестирования могут впоследствии не сохраняться. Также это первая версия дополнительных требований.
- Первая версия описания возможной архитектуры с набросками представлений моделей вариантов использования, анализа, проектирования и реализации.
- Возможно, концептуальный прототип, демонстрирующий работу новой системы.
- Базовый список рисков и список приоритетов вариантов использования.
- Зачатки плана всего проекта, включая общий план фаз.
- Первая версия бизнес-плана, включающая в себя описание среды ведения бизнеса и критерии успешности (предполагаемый доход, исследование рынка, оценку проекта).

Заинтересованные лица теперь хорошо понимают концепцию и выполнимость проекта. Определен порядок приоритетности вариантов использования. Полученная информация позволяет менеджеру проекта детально спланировать следующую фазу. Результаты, полученные в этой фазе, будут уточнены в фазе проектирования, которую мы рассмотрим в главе 14.

14 Фаза проектирования создает базовый уровень архитектуры

Раз мы перешли к фазе проектирования, значит, нами была пройдена первая из главных вех, а это говорит о выполнении трех действий:

- Мы сформулировали первоначальный вариант архитектуры — возможную архитектуру, а это говорит о том, что мы знаем, как построить для заявленной системы архитектуру, которая будет поддерживать все ее новые или сложные конструкции.
- Мы определили имеющие важное значение риски — наиболее опасные риски и, рассмотрев степень их опасности, пришли к выводу о возможности построения системы.
- Мы создали первоначальный бизнес-план, достаточно детальный, чтобы оправдать переход ко второй фазе, и получивший одобрение заинтересованных лиц, в том числе и тех, кто финансирует разработку.

Введение

Наши основные цели:

- Определить большинство оставшихся требований, сформулировав функциональные требования в виде вариантов использования.
- Заложить прочный фундамент архитектуры — базовый уровень архитектуры — для задания направления работ на фазах построения и внедрения и распространения этого направления на будущие поколения программы.
- Продолжить отслеживание оставшихся особо опасных рисков и определение существенных рисков до того момента, когда мы сможем оценить их влияние на бизнес-план и, в частности, на бизнес-предложение.
- Внести новые детали в план проекта.

Для выполнения этих целей мы изучаем систему «на милю в ширину и на дюйм в глубину». В тех случаях, когда решающими или наиболее опасными являются

технические риски, для построения прочной архитектуры нам придется копать вглубь. В большом проекте мы также можем осуществлять одновременное исследование горячих точек «на дюйм в ширину и на милю в глубину». Ранним определением наиболее сложных частей системы и инициацией прототипов-«разведчиков», определяющих риски и управляющих ими, занимается системный архитектор.

Мы создаем описание архитектуры, основываясь на своем понимании системы в целом, ее области деятельности, функциональных и нефункциональных требований, например производительности. Кроме того, разрабатывая это описание, мы балансируем требования, содержащиеся в вариантах использования в модели вариантов использования, в соответствии с архитектурой. Эти действия связаны друг с другом и взаимозависимы (см. раздел «Варианты использования и архитектура» главы 4).

Основное внимание в фазе проектирования уделяется разработке базового уровня архитектуры. Это означает такие действия, как детализацию примерно 80% вариантов использования и обработку рисков, которые могут повлиять на выполнение этой задачи. В ходе этой фазы мы дорабатываем среду разработки как придавая ей способность к поддержке фазы проектирования, так и готовя к поддержке фазу построения. До конца фазы мы будем собирать информацию, необходимую для планирования фазы построения. Кроме того, к этому моменту у нас будет достаточно информации, чтобы закончить ту работу, которую мы начинали в фазе анализа и планирования требований, — построить надежный бизнес-план.

В начале фазы проектирования

В начале фазы проектирования мы получаем созданные в фазе анализа и планирования требований план фазы проектирования, частично построенную модель вариантов использования и описание возможной архитектуры. Кроме того, мы можем получить первые варианты моделей анализа и проектирования. Однако мы не должны рассчитывать на повторное использование этих моделей, несмотря на то, что направление будущей работы они нам укажут. Наша задача в фазе проектирования фактически состоит в том, чтобы наполнить эти модели содержанием, опять же не полностью, а так, чтобы хватило на создание базового уровня архитектуры.

Мы также можем получить концептуальный прототип, созданный для демонстрации применения системы. Однако не следует ожидать развития этого прототипа. Обычно он создается как можно быстрее для доказательства реальности построения системы и не является базой для ее дальнейшего построения.

Планирование фазы проектирования

Планирование этой фазы, проводимое в конце фазы анализа и планирования требований, может быть не окончено. Нередко ресурсы, которые потребуются на фазе проектирования, в момент начала этой фазы известны не до конца. В некоторых случаях между окончанием фазы анализа и планирования требований и началом фазы проектирования проходит определенное время. Получив дополнительные

сведения о необходимых ресурсах, персонале и графике работ, менеджер проекта модифицирует созданные ранее план итераций и план фазы.

Построение команды

Не все, что поняла команда, занимавшаяся фазой анализа и планирования требований, было записано. Поэтому менеджер проекта переводит из команды, работавшей над предыдущей фазой, в команду, занимающуюся фазой проектирования, столько сотрудников, сколько сможет. Они будут, кроме всего прочего, «памятью команды». Помимо этого, для проектирования потребуются дополнительные специалисты. Мы имеем в виду, например, сотрудников, имеющих опыт работы с многократно используемыми модулями, которые можно будет применить в нашем проекте. Вообще команда для выполнения фазы проектирования немного больше по размеру, чем команда, работавшая над проектом на фазе анализа и планирования требований. В проект приходят новые люди. Менеджер проекта должен выбрать из этих людей будущих руководителей команд проектирования для фазы построения.

Модификация среды разработки

В свете того, что было сделано в фазе анализа и планирования требований и должно быть сделано в фазе проектирования, менеджер проекта продолжает вносить изменения в среду разработки, первоначально обсуждавшиеся в подразделе «Погружение проекта в среду разработки» главы 13.

Задание критериев оценки

Критерии оценки, соответствия которым мы добиваемся на каждой итерации фазы проектирования и в целом в фазе, уникальны для каждого проекта. Однако мы можем обсудить результаты проверки в понятиях задач фазы проектирования.

Проработка требований

Критерии выполнения:

- Получены ли требования, актанты и варианты использования, необходимые для проектирования базового уровня архитектуры, определены ли существенные риски и поддерживаются ли созданные бизнес-план и бизнес-предложение?
- Соответствует ли задачам фазы проектирования степень их детализации?

Создание базового уровня архитектуры

Критерии выполнения:

- Соответствует ли исполняемый базовый уровень архитектуры не только формально определенным требованиям, но и пожеланиям, высказанным заинтересованными лицами после предъявления им первых вариантов работающего базового уровня?
- Выглядит ли базовый уровень архитектуры достаточно надежным для того, чтобы выдержать фазу построения и то добавление функциональности, которое потребуется в следующих версиях?

Снижение существенных рисков

Критерии выполнения:

- Были ли в достаточной мере снижены особо опасные риски — исключены или смягчены путем составления плана на случай неожиданностей?
- Все ли существенные риски были определены? (О существенных рисках см. в подразделе «Фаза проектирования обеспечивает возможность выполнения» и разделе «Риски влияют на планирование проекта» главы 12.)
- Были ли существенные риски изучены настолько, чтобы их можно было учесть в бизнес-предложении?
- Можно ли справиться с рисками, вошедшими в список рисков, традиционными средствами фазы построения?

Оценка осмысленности бизнес-плана

Критерии выполнения:

- Достаточно ли хорошо определены цена предложения, график работ и качество проекта?
- Предполагает ли бизнес-план доходность инвестиций в проект, и достижим ли стандартный уровень прибыльности, принятый в деловой практике?
- Короче говоря, готовы ли мы согласиться на конкретную цену разработки и зафиксировать ее в контракте (или его эквиваленте для внутренней разработки)?

Типичный поток работ итерации на фазе проектирования

Как показано на рис. 14.1, типичная итерация включает в себя пять рабочих процессов. Основная работа совершается в рабочих процессах определения требований, анализа и проектирования. По сравнению с ними реализация и тестирование потребляют значительно меньше ресурсов. Рабочие процессы рассматривались в части 2, и в этой главе мы коснемся их только в той мере, в какой это необходимо для фазы проектирования. Итак, мы осуществляем, отчасти параллельно, четыре вида деятельности. Первый — это базовые рабочие процессы; второй — планирование итераций, описанное в разделах «Планирование предваряет деятельность», «Риски влияют на планирование проекта», «Расстановка приоритетов вариантов использования» и «Требуемые ресурсы» главы 12 в подразделе «Планирование фазы проектирования» данной главы; третий — оценка результатов, о которой мы говорили в разделах «Оценка итераций и фаз» главы 12 и «Оценка результатов итераций и фазы проектирования» данной главы; четвертый — дальнейшее развитие среды разработки, впервые описанное в подразделе «Определение исходных деловых перспектив» главы 13. В этом разделе мы приводим обзор базовых рабочих процессов и их ролей на фазе проектирования. В разделе «Выполнение основных рабочих процессов — от определения требований до тестирования» мы добавим к этому описанию дополнительные детали.

В ходе «проектирования архитектуры» мы определяем, анализируем, проектируем, реализуем и тестируем требования, которые имеют отношение к архитекту-

ре. Деталям, не влияющим на архитектуру, уделяется минимум внимания. Мы откладываем эти детали до фазы построения. Базовый уровень архитектуры, являющийся результатом этих усилий, представляет собой скелет системы. Сам по себе он может немногое, за исключением тех частей, которые мы для проверки работоспособности архитектуры в целом реализуем с достаточной степенью детализации. Мы разрабатываем базовый уровень архитектуры за одну-две итерации. Большее число итераций — это исключительный случай. Число итераций зависит от области действия системы, рисков, степени новизны, сложности технических решений и опытности разработчиков.

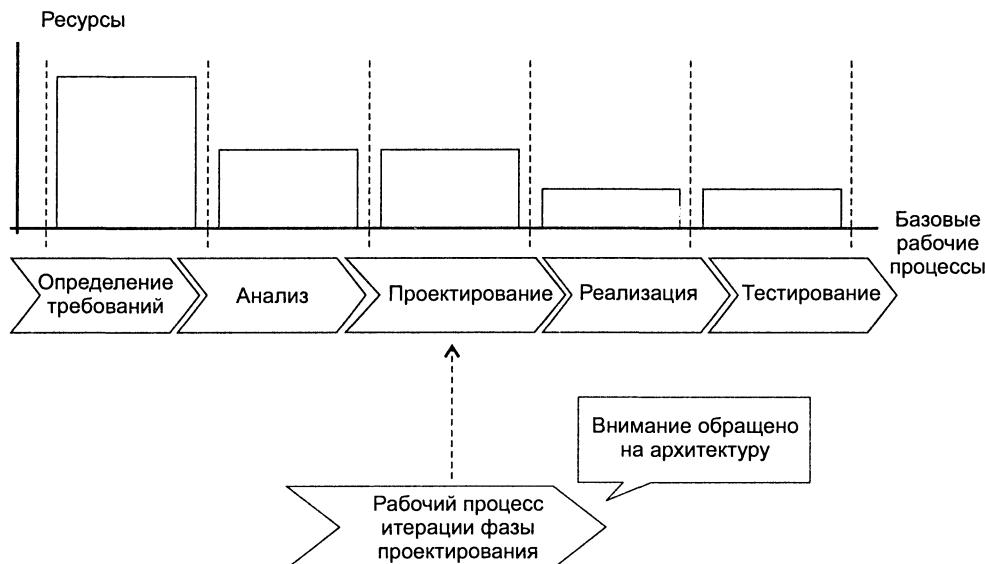


Рис. 14.1. На итерациях фазы анализа и определения требований осуществляются пять базовых рабочих процессов

Задача обработки рисков в этой фазе — не устраниТЬ их полностью, а понизить до уровня, приемлемого для фазы построения. Другими словами, в фазе проектирования технические риски, касающиеся архитектуры, откладываются до момента реализации этой архитектуры! Что такое «уровень риска, приемлемый для фазы построения»? Эта фраза означает, что риск был исследован до такой степени, что мы ясно видим способ его снижения и в состоянии оценить силы и время, необходимые для этого. Фактически, риск не должен устраняться до реализации варианта использования, к которому он относится, а это случается иногда в фазе проектирования, обычно в фазе построения и в редких случаях — в фазе внедрения.

Определение и уточнение большей части требований

Что мы понимаем под определением «большинства требований»? Мы начинали обсуждение этого вопроса в разделе «Планирование фазы проектирования» главы 13, когда приступали к планированию фазы анализа и планирования требова-

ний. Тогда мы говорили, что стремимся определить около 80% вариантов использования. Мы можем детально описать 40–80% объема вариантов использования. Нам не нужно определять все варианты использования и не нужно детально описывать все, что мы определили, поскольку мы знаем по опыту, что некоторые подсистемы и так готовы для проектирования (архитектуры), не содержат неожиданных рисков и могут быть точно просчитаны (см. также главы 6 и 7).

Из всего объема вариантов использования, который мы детально описываем, мы выбираем примерно половину для очень тщательного анализа. Из этой половины мы при необходимости можем выбрать часть сценариев для проектирования, реализации и тестирования с целью создания архитектуры и снижения рисков (см. табл. 13.1). Наша цель — определение такой доли требований, которой будет достаточно для выполнения целей текущей фазы.

Разработка базового уровня архитектуры

Архитектор расставляет приоритеты вариантов использования и производит анализ, проектирование и реализацию уровня архитектуры, как было описано в главах 8, 9 и 10. Другие сотрудники осуществляют анализ и проектирование, описанные в главах 8 и 9, то есть анализируют классы и пакеты (см. главу 8) и проектируют классы и подсистемы (см. главу 9).

Инженеры по тестированию сосредоточиваются на построении среды тестирования и тестировании компонентов и базового уровня, который реализует архитектурно значимые варианты использования.

Пока команда малочисленна — ищите путь

Пока команда малочисленна, а на фазе проектирования это именно так, самое время искать и проверять различные решения (технологии, каркасы, структуры и т. п.). Если проект сложен, для получения стабильной архитектуры может потребоваться три или четыре итерации. Позже, в фазе построения, когда в команде будет множество людей, а в проекте — сотни или тысячи строк кода, у вас должна быть стабильная архитектура, которая позволит системе расти.

Вам может хватить и одной итерации, если система мала и проста, но если система велика и сложна, это будет только первым шагом. Необходимость дополнительных итераций зависит от таких факторов, как значительность рисков, сложность системы и базовый уровень архитектуры, необходимый для управления системой.

Итерации продолжаются до тех пор, пока архитектура не станет стабильной, то есть доведенной до такого состояния, что ей могут понадобиться лишь небольшие изменения, и не будет с приемлемым качеством отображать систему.

Выполнение основных рабочих процессов — от определения требований до тестирования

В фазе проектирования мы полагаемся на результаты предыдущей фазы. Однако в фазе анализа и планирования требований мы могли только верить, что в следующей фазе сможем построить архитектуру. Теперь, в фазе проектирования, мы это

сделаем. Мы поглядим на то, что делали раньше, и, возможно, найдем кое-что, что можем использовать. Теперь мы смотрим не только на те варианты использования, которые содержат особо опасные риски, но и на те, которые имеют значение для архитектуры. Кроме того, для создания уточненного предложения мы нуждаемся в значительно большем количестве вариантов использования. Вдобавок для окончания фазы нам нужно решить проблему базового уровня архитектуры, прочного базового уровня, к которому на фазе построения мы сможем добавлять функциональность. Поэтому при разработке мы должны уделять качеству и расширяемости больше внимания, чем в фазе анализа и планирования требований.

В начале этой итерации следует проработать риски и определить варианты использования. Мы уже обсуждали это в фазе анализа и планирования требований. Нам следует определить порядка 80% требований, чтобы выделить из них те, которые оказывают влияние на архитектуру, и получить дополнительную информацию для создания предложения. Что касается обычных пропорций, отношение к разработке базового уровня архитектуры имеют 10% обнаруженных вариантов использования.

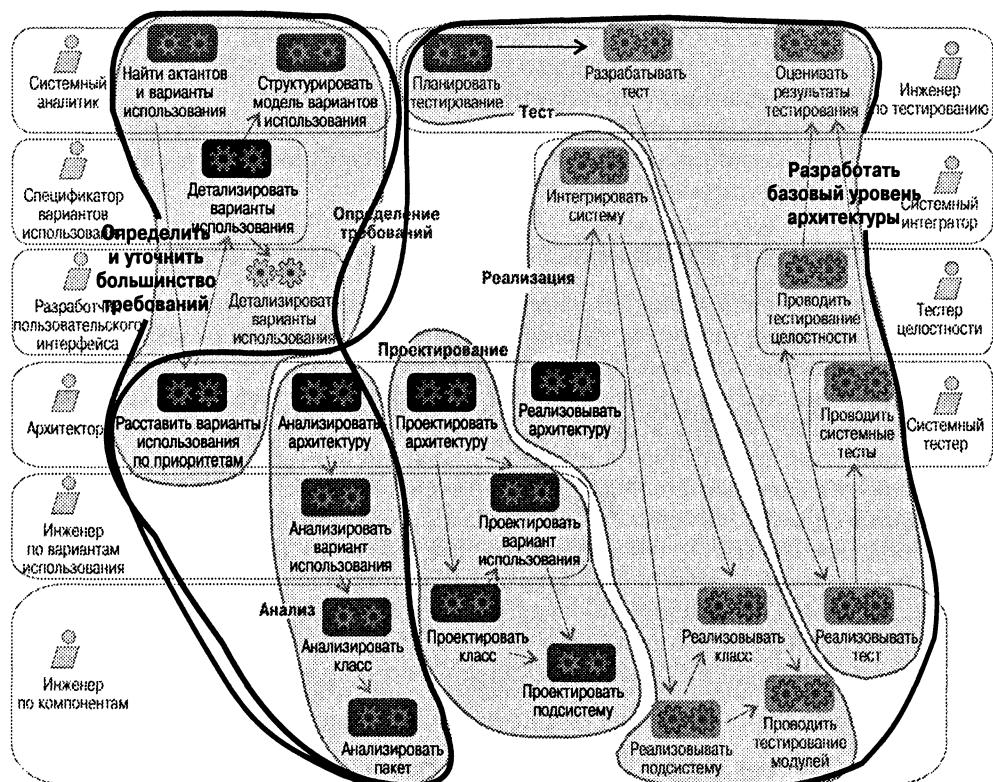


Рис. 14.2. Контуры показывают основную деятельность в ходе фазы проектирования

В приведенном гипотетическом проекте мы рассматриваем систему средней сложности, разработка базового архитектурного уровня которой занимает одну

итерацию. Мы предполагаем целинную разработку. Как мы уже отмечали, менеджер проекта имеет наметки плана проекта и вполне детальный план первой итерации, сделанный в конце предыдущей фазы. Первым делом мы добавим к плану итерации детали, подсказанные нам архитектором и опытными разработчиками.

Следующее описание сделано в терминах пяти рабочих процессов. Это перечисление рабочих процессов может склонить нас к мысли, что мы выполняем их последовательно, но работа внутри каждого из рабочих процессов может происходить параллельно, как показано на рис. 14.2.

Определение требований

В этом подразделе мы найдем, расставим в соответствии с приоритетами, детализируем и структурируем варианты использования (детальное обсуждение процесса определения требований см. в главах 6 и 7).

Нхождение вариантов использования и актантов

Системный аналитик определяет варианты использования и актантов дополнительно к тем, которые мы определили в фазе анализа и планирования требований (см. подраздел «Деятельность: Нхождение актантов и вариантов использования» главы 7). Пока нам нужно только *разобраться* с 80% вариантами использования, в *детализации* всех этих вариантов использования мы не нуждаемся. Мы должны идентифицировать почти все 80%, мы описываем только часть и анализируем только часть того, что описали. Под «пониманием» мы подразумеваем «определить важность для архитектуры» и убедиться в том, что мы не пропустили ничего, что может оказывать влияние на архитектуру или бизнес-предложение.

Сколько именно вариантов использования мы определим, зависит также от того, какая точность нам необходима. Если мы нацеливаемся на бизнес-предложение с фиксированной ценой, нам следует детализировать большее количество вариантов использования, возможно, до 80%. Для некоторых сложных систем мы можем определить почти все варианты использования и детализировать до 80%. Если мы финансируем свой проект сами, мы можем остановиться на более низкой доле определенных вариантов использования. Это, разумеется, повысит риск. Решение о возможном увеличении риска в обмен на сокращение усилий и времени на фазу проектирования принимается руководством. Подобная мелочность может привести к пропуску рисков, которые способны попортить нам немало крови в будущем, ради пустяковой, но немедленной выгоды.

Создание прототипа интерфейса пользователя

Еще один вид деятельности, осуществляемый в ходе определения требований, — это определение интерфейсов пользователей (см. подраздел «Деятельность: Создание прототипа интерфейса пользователя» главы 7).

Мы рассматриваем пользовательские интерфейсы в ходе фазы проектирования только в том случае, если они представляют интерес с точки зрения архитектуры. Однако это редкий случай — нечасто пользовательские интерфейсы хоть в чем-то уникальны. Если же в наших интерфейсах есть что-то новое, мы можем создать наш собственный каркас пользовательских интерфейсов. Так, например, когда система, которую мы разрабатываем, будет готова, она сама будет представлять собой каркас для пользовательских интерфейсов. Другим примером может послу-

жить систему с уникальным коммуникационным протоколом, оказывающим влияние на архитектуру, а точнее, на производительность и время отклика.

Еще одна причина разрабатывать пользовательский интерфейс, даже если он не оказывает влияния на архитектуру – это изучение процесса работы реальных пользователей. Однако на эту крайность следует идти только в том случае, если мы не смогли показать значимость системы во время фазы анализа и планирования требований. Вообще же создавать прототипы интерфейса пользователя на фазе проектирования нет необходимости.

Определение приоритетности вариантов использования

Продолжая разрабатывать частичную модель вариантов использования, созданную на фазе анализа и определения требований, мы решаем две задачи: дополнить модель новыми вариантами использования и разработать базовый уровень архитектуры (см. подраздел «Деятельность: Определение приоритетности вариантов использования» главы 7). Сначала наше время уходит на определение дополнительных вариантов использования, затем мы переходим к архитектуре. Однако мы можем координировать эти две задачи. Наши решения вызваны приоритетами рисков, которые мы зафиксировали, и порядком, в соответствии с которым мы собираемся продолжать разработку (см. подраздел «Деятельность: Определение приоритетности вариантов использования» главы 7 и раздел «Расстановка приоритетов вариантов использования» главы 12). Из модели вариантов использования архитектор создает представление, которое включается в описание архитектуры.

Детализация вариантов использования

Спецификаторы вариантов использования наполняют подробностями варианты использования, необходимые для полного понимания требований и создания базового уровня архитектуры. О работе спецификаторов вариантов использования см. подраздел «Деятельность: Детализация вариантов использования» главы 7. В этой фазе мы ограничим наши усилия первичным описанием архитектурно значимых и сложных вариантов использования. В вариантах использования, выбираемых нами для детализации, мы обычно не детализируем все их части, ограничиваясь теми, которые понадобятся нам еще в этой фазе.

Мы избегаем описывать то, что нам не нужно. Однако, как мы говорили ранее, в некоторых запутанных случаях нам может понадобиться детализировать все сценарии всех вариантов использования, то есть 100% объема вариантов использования.

Структурирование вариантов использования

Системный аналитик просматривает сделанное и ищет подобия, упрощения и другие возможности для совершенствования структуры модели вариантов использования. Для создания хорошо структурированной, легкой для понимания модели аналитик разрабатывает такие механизмы, как расширения и обобщения (см. подраздел «Деятельность: Структурирование модели вариантов использования» главы 7). Модель будет легко модифицировать, расширять и обслуживать, если мы, например, уменьшим ее избыточность. Однако иногда аналитик не в состоянии в этот момент подыскать удачную структуру. В этом случае следует подождать несколько итераций, иногда до тех пор, пока варианты использования не будут пропущены через рабочие процессы анализа и проектирования.

Пример. Структурирование модели варианта использования. Работая над моделью вариантов использования, разработчики обнаруживают, что некоторые варианты использования имеют сходные реализации. Так, например, варианты использования *Заказать товары или услуги*, *Подтвердить заказ*, *Выслать счет покупателю* и *Послать напоминание* включают в себя пересылку *Торгового объекта* между покупателем и продавцом. Для реструктурирования модели вариантов использования будет создан единый вариант использования для этого общего поведения — *Послать Торговый Объект*. Разработчики, реализуя варианты использования, будут многократно использовать вариант использования *Послать Торговый Объект*. *Торговый объект* войдет в ряд таких объектов, как *Счет*, *Заказ* и *Подтверждение заказа*. Он так же изменяет свое состояние, поддерживает такие же операции, и так же курсирует между *Продавцом* и *Покупателем*. Факт создания такого упрощения означает, что все эти классы могут быть порождены от одного абстрактного класса *Торговый объект*.

Анализ

Мы создали черновую версию модели анализа в фазе анализа и планирования требований (детальное описание модели анализа см. в главе 8). Сейчас мы отталкиваемся от нее, но может оказаться, что некоторые важные фрагменты мы пропустили. В фазе проектирования мы должны проработать архитектурно значимые или просто сложные варианты использования, которые следует уточнить для лучшего понимания того, на чем основано бизнес-предложение.

В этом подразделе мы рассмотрим такие виды деятельности, как анализ архитектуры, анализ вариантов использования, анализ классов и пакетов. Мы хотим пронаблюдать за теми вариантами использования, которые важны для архитектуры. Их доля обычно менее 10% общего объема вариантов использования. Мы также анализируем варианты использования для того, чтобы более верно понять их и рассмотреть их взаимное пересечение. Всего нам может понадобиться рассмотреть около 50% общего объема детально описанных вариантов использования.

ПРИСПОСОБЛЕНИЕ ТРЕБОВАНИЙ К АРХИТЕКТУРЕ

Определяя требования — и заполняя соответствующие варианты использования — мы используем появляющееся у нас при этом осознание разрабатываемой архитектуры для того, чтобы делать это грамотнее (см. раздел «Варианты использования и архитектура» главы 4). Оценивая значимость и стоимость каждого из новых требований или вариантов использования, мы делаем это в свете базового уровня архитектуры, которую уже создали. Архитектура подсказывает нам, что одни требования будет легко реализовать, а другие — сложно.

После изучения ситуации создаются новые требования. Мы можем обнаружить, например, что изменение требований, — такое, которое предполагает минимум или вовсе не предполагает смысловых изменений, — может сделать реализацию проще. Эта работа станет проще потому, что изменения требований приведут к проекту, более совместимому с существующей архитектурой. Мы должны обсудить эти изменения требований с заказчиком.

По мере того как мы производим анализ, проектирование, реализацию и тестирование системы, нам необходимо объединять изменения в проекте и действующую архитектуру в действующую модель проектирования. Под осуществлением такого объединения мы понимаем изменение ранее созданных подсистем, компо-

нентов, интерфейсов, реализаций вариантов использования, активных классов и т. д. Таким образом мы можем создать новый проект, более выгодный, чем существующий.

Пример. *Повторное обсуждение требований.* Представим, что мы работаем в компании, которая разработала пакет программ под названием ПортфолиоПлюс, анализирующий портфели акций частных заказчиков. Три года назад заказчики вводили изменения биржевых цен на акции вручную и были счастливы. Когда началось взрывное распространение World Wide Web, их запросы возросли. Теперь они могут получать биржевые сводки легко, бесплатно и почти мгновенно. Чтобы остаться на коне, мы должны сделать так, чтобы ПортфолиоПлюс умел получать биржевые сводки не хуже, чем его пользователи.

Из-за особенностей выбранной архитектуры ПортфолиоПлюс измененить ее так, чтобы прямо «настроиться на нужную волну и слушать биржевые сводки» будет нелегким делом. Менее дорогостоящей альтернативой будет использование разработанных ранее библиотек для организации приема и ввода данных в таблицы Excel. Простым и экономически эффективным решением будет включение в ПортфолиоПлюс макрокоманды, которая будет не только загружать таблицу Excel, но и запрашивать с web-сайта новую ее версию с теми сводками, которые интересуют пользователя. Наша задача сводится к трем пунктам:

- Создавать на нашем web-сайте электронную таблицу, когда пользователь ее запросит.
- Поддерживать web-сайт на достаточном техническом уровне, чтобы его ресурсов хватило для обслуживания пользователей ПортфолиоПлюс.
- Написать макрос для получения таблицы с сайта.

Пример. *Реальный случай.* Этот пример демонстрирует реальный случай повторного использования и показывает, какой вклад в проект может внести обсуждение.

Обсуждая требования с заказчиком в разрезе доступной архитектуры, компании получают возможность создавать системы лучшего качества за более низкую цену. Как это происходит, показывает типичный пример, произошедший в одной телекоммуникационной компании.

Заказчик подготовил основательный список требований в форме, схожей с вариантами использования. При оценке затрат на разработку системы было обнаружено, что изготовление ее с нуля потребует около 25 человеко-лет. Поставщик программного обеспечения показал телекоммуникационной компании, что, модифицировав эти требования в сторону использования существующей архитектуры, они получат очень похожую, если не точно такую же систему. Путем выделения образцов из этой существующей архитектуры стало возможным снизить затраты на изготовление системы на 90%!

Телекоммуникационная компания решила использовать предложенный метод. Она получила систему на основе стандартного продукта, слегка доработанного в соответствии со специальными требованиями. Удалось сэкономить более 20 человеко-лет. Кроме того, в результате этой работы заказчику не придется сталкиваться со все возрастающей стоимостью поддержки заказных программ и оборудования, он может обойтись значительно более дешевой поддержкой, которой обеспечены стандартные продукты.

Разница между тем, чего заказчик хотел сначала, и тем, на что он согласился в результате, вызвана тем, как поставщик расставил варианты использования по

архитектуре. Минимальные изменения в интерфейсе пользователя, другие способы наблюдения за основным процессом, другие пути управления и отображения потока трафика и прочие подобные изменения сложились в 90% снижение затрат. Кроме того, заказчик получил большую функциональность, чем ожидал. Получение основной программы за ту низкую цену, которую предложил ему поставщик, на самом деле было уже просто замечательно. Но поставщик программ смог установить низкую цену и на дополнительные функции, поскольку они уже были реализованы и протестированы раньше.

Анализ архитектуры

В фазе анализа и планирования требований мы окончили анализ архитектуры, как только доказали, что созданная архитектура работоспособна (см. подраздел «Деятельность: Анализ архитектуры» главы 8). Обычно это не слишком большой объем работы. Теперь, на фазе проектирования, мы продолжим анализ архитектуры вплоть до того момента, когда сможем поддерживать полномасштабную архитектуру, то есть исполняемый базовый уровень архитектуры.

Для этой цели архитектор, работая с архитектурным представлением модели вариантов использования, соответствующими требованиями, гlosсарием и сведениями о предметной области, доступными в виде бизнес-модели (или упрощенной модели предметной области), делает исходное разбиение системы на пакеты анализа (высокого уровня). Он может работать с многоуровневой архитектурой. Он определяет пакеты специфического и общего уровней приложения, которые наиболее важны для понимания проблемы. Рассматривая ключевые варианты использования в архитектурном представлении вариантов использования, архитектор может определить очевидные и архитектурно значимые сервисные пакеты и классы анализа.

Кроме того, в ходе работы над групповыми требованиями вариантов использования архитектор всматривается в скрытые механизмы, необходимые для поддержки реализации вариантов использования. Он определяет требуемые обобщенные механизмы анализа (см. подраздел «Определение общих специальных требований» главы 8). Эти механизмы включают в себя обобщенные кооперации (см. главу 3) и обобщенные пакеты. Обобщенная кооперация включает в себя такие функции, как восстановление после сбоев и обработку транзакций. Обобщенные пакеты имеют отношение к длительному хранению данных (перsistентность), графическому интерфейсу пользователя и распределенной обработке объектов.

После этого архитектор занимается улучшением представления модели анализа.

Анализ варианта использования

Множество вариантов использования, описанных только в модели вариантов использования, так и не стали понятными до конца (см. подраздел «Деятельность: Анализ варианта использования» главы 8). Варианты использования должны быть уточнены путем построения классов анализа, создаваемых на основе требований, но не обязательно реализующих варианты использования напрямую. Такая необходимость уточнения, в частности, важна для тех вариантов использования, которые сложны или влияют друг на друга. Например, для варианта использования, который должен иметь доступ к информации, некий другой вариант использования предоставляет эту информацию.

Итак, интересующие нас с точки зрения архитектуры варианты использования и варианты использования, важные для понимания требований, уточняются по-

средством классов анализа. Прочие варианты использования, не интересные с точки зрения архитектуры или понимания требований, не уточняются и не анализируются. Для них инженерам по вариантам использования достаточно объяснить, что эти варианты использования из себя представляют, и указать тот факт, что они взаимно независимы. Инженеры знают, что с ними делать, когда настанет время реализации — в фазе построения.

Важные или сложные варианты использования не требуют детального описания, им хватит такого уровня, который необходим аналитикам для понимания, где используется данный вариант использования, — в базовом уровне архитектуры или в бизнес-плане. Если мы рассмотрим 80% вариантов использования с целью понять их роли в системе и опишем менее 40% общего объема вариантов использования, этого обычно хватит для того, чтобы меньше заботиться об анализе, поскольку некоторые из этих вариантов использования не окажут никакого влияния на бизнес-план (подробнее об этом соотношении см. табл. 13.1).

Затем инженеры по вариантам использования начинают поиск классов анализа, реализующих варианты использования. Определенные архитектором архитектурно значимые классы используются ими в качестве исходных данных. Инженеры привязывают требования к этим классам. Большая часть работы по анализу вариантов использования — это разбор каждого из вариантов использования модели вариантов использования и его детальное описание в терминах классов и их ответственостей. Показываются также отношения между классами и их (классов) атрибутами.

Пример. Ответственности класса. В ходе работы с вариантом использования *Оплатить счет* на первой итерации разработчики создали класс для планирования и проведения платежей — *Планировщик оплат* со следующими ответственостями:

- Создание запроса на оплату.
- В день платежа инициирование перевода денег.

На следующей итерации разработчики могут решить, что необходимы дополнительные ответственности. Их добавление не потребует реструктуризации классов. В хорошей модели анализа мы в состоянии добавлять новые ответственности, не разрушая все, что было создано. В крайнем случае, на следующих итерациях потребуется реструктурировать созданные классы. Позднее, когда разработчики расширят зону деятельности, например, в ходе второй итерации фазы проектирования, они могут обнаружить, что класс требует больше ответственостей, чем он может выдержать без реструктуризования. Эти ответственности могут включать в себя:

- Отслеживание запланированных платежей.
- Уведомление *Счета* о том, что он был *Принят к оплате* и оплачен (*Закрыт*).

На основе работы по анализу вариантов использования архитектор выбирает архитектурно значимые классы. Эти классы станут основанием архитектурного представления модели анализа.

Анализ класса

Инженеры по компонентам уточняют классы, определенные на предыдущих шагах. Они объединяют назначенные им ответственности с ответственностями других вариантов использования. Они определяют имеющиеся механизмы анализа

и находят, как они используются в каждом из классов (см. подраздел «Деятельность: Анализ класса» главы 8).

Анализ пакета

Как мы отмечали ранее, проводя анализ архитектуры, архитектор рассматривает сервисы системы и группирует классы в сервисные пакеты. Это делается в ходе анализа архитектуры. Чтобы сделать из этих группировок сервисные пакеты, за них берутся инженеры по компонентам, которые их уточняют и поддерживают (см. подраздел «Деятельность: Анализ пакетов» главы 8).

Проектирование

На этой фазе мы обычно проектируем и реализуем менее 10% объема вариантов использования. Этот небольшой процент отсчитывается только от тех вариантов использования, которые к данному моменту определены. На фазе проектирования мы проектируем уровень архитектуры. Это означает, что мы создаем проекты архитектурно значимых вариантов использования, классов и подсистем. Пакеты в анализе и подсистемы в проектировании особо важны для определения представлений архитектуры. Классификаторы могут быть, а могут и не быть архитектурно значимыми, но пакеты и подсистемы важны для архитектуры почти всегда (см. главу 9).

Проектирование архитектуры

Архитектор отвечает за проектирование архитектурно значимых аспектов системы, описанных в представлении архитектуры модели проектирования (см. подраздел «Артефакт: Описание архитектуры (представление модели проектирования)»). Архитектурное представление модели проектирования включает в себя подсистемы, классы, интерфейсы и реализации архитектурно значимых вариантов использования, входящих в представление модели вариантов использования. Другие аспекты проектирования ложатся на инженера по вариантам использования и инженера по компонентам.

Архитектор определяет многоуровневую архитектуру (включая обобщенные механизмы проектирования), подсистемы и их интерфейсы, архитектурно значимые классы проектирования и конфигурации узлов.

Определение многоуровневой архитектуры. Архитектор продолжает работу, начатую в фазе анализа и планирования требований, и проектирует многоуровневую архитектуру. Он пересматривает уровень системного программного обеспечения и средний уровень программного обеспечения (см. подраздел «Определение подсистем среднего уровня и уровня системного программного обеспечения» главы 9) и выбирает в итоге те продукты, которые будут использоваться на этих уровнях. Архитектор может включить в их число существующие унаследованные разработки его собственной организации, в этих случаях он определяет части, которые можно использовать повторно, и интерфейсы с этими частями. Архитектор подбирает в качестве продуктов для нижних уровней реализации механизмы проектирования, соответствующих механизмам анализа, обнаруженным на предыдущих шагах (см. подраздел «Анализ архитектуры» данной главы). Напомним, что под «механизмом проектирования» мы понимаем механизмы операционных систем, под которыми работает наша система, языки программирования, системы управления базами данных, брокеры объектных запросов и т. п. Среда реализации ограничивает механизмы проектирования, которые могут быть использованы в продукте. Они могут быть получены как путем разработки, так и покупкой реа-

лизующих их продуктов. Они часто представляют собой подсистемы среднего уровня программного обеспечения и уровня системного программного обеспечения. Их можно строить или подбирать параллельно с рабочим процессом анализа. Инженер по компонентам ведет проектирование, пользуясь их терминологией.

Пример. *Java RMI для распределенной обработки объектов.* Java RMI используется для распределенной обработки объектов. Это значит, что пакет `java.rmi` используется для реализации варианта использования *Передать торговый объект* на фазе проектирования.

Определение подсистем и их интерфейсов. Затем к делу снова приступает архитектор, который занимается верхними уровнями архитектуры, уровнями приложения. Взяв за основу пакеты модели анализа, он определяет соответствующие им подсистемы, которые включаются в модель проектирования. Обычно он старается превратить каждый сервисный пакет модели анализа в сервисную подсистему модели проектирования, высокуюровневые пакеты анализа становятся в модели проектирования подсистемами проектирования.

Этот подход в некоторых случаях прекрасно срабатывает, но нередко в дело вмешивается сопротивление перехода между анализом и проектированием. В некоторых ситуациях ни один пакет анализа не может быть точно отображен на подсистему проектирования, если не считать унаследованных систем (или их частей). Точнее говоря, унаследованная система может реализовывать несколько пакетов анализа или их частей или пакет анализа может быть отображен на несколько различных унаследованных систем, в общем, между ними существует отношение «многие-ко-многим».

В других ситуациях архитектор может выбирать многократно используемые строительные блоки, например каркасы, как собственной разработки, так и поставляемые внешними разработчиками. Эти блоки могут не полностью соответствовать структуре пакетов, предоставляемых моделью анализа, поэтому архитектор может выбрать структуру подсистем для проектирования архитектуры, немного отличающуюся от структуры, определенной при анализе архитектуры.

Определение архитектурно значимых классов проектирования. Архитектор «переводит» архитектурно значимые классы анализа в классы проектирования. По мере того как определяются классы проектирования, он отбирает те из них, которые имеют значение для архитектуры, например активные классы, и вносит их в описание архитектуры.

При создании распределенной системы — определение узлов и их сетевых конфигураций. Архитектор разбирается с параллельностью и распределенной обработкой, в которых нуждается система, путем изучения необходимых задач и процессов, а также физической сети процессоров и других устройств. Уже спроектированные варианты использования, частично визуализированные в виде диаграмм взаимодействий, будут исходными данными для этой задачи. Архитектор привязывает объекты, показанные на диаграммах взаимодействия, к активным классам, а те, в свою очередь, привязываются к процессорам или другим устройствам. Эти действия распределяют функциональность на логическом и физическом уровне.

Архитектор создает новую версию архитектурного представления модели проектирования и новую версию представления модели развертывания, которые включаются в описание архитектуры.

Проектирование варианта использования

Теперь архитектурно значимые варианты использования спроектированы в виде подсистем проектирования, сервисных подсистем или классов проектирования

(см. подраздел «Деятельность: Проектирование вариантов использования» главы 9). Прочие варианты использования, которые мы определили, детализировали и проанализировали, на этой фазе не проектируются. Эта работа похожа на ту, что мы делали при анализе («Деятельность: Анализ варианта использования»), но имеет несколько важных особенностей. При анализе мы хотим проанализировать и уточнить варианты использования и получить описания, которые были бы надежными, гибкими и годились бы для повторного использования. Это описание было первой операцией проектирования. Мы специально занимались поисками ответственности определенных классов анализа.

При проектировании мы углубляемся в детали. Переходя от анализа к проектированию, инженер по компонентам приспосабливает модель анализа для построения работающей модели проектирования. Это необходимо потому, что модель проектирования ограничена механизмами проектирования. Однако пакеты и классы анализа предоставляют нам простой способ получения подсистем и классов проектирования. Как только они будут обнаружены, мы описываем не только ответственности, которые эти элементы проектирования должны в себя включать, но и их детальные взаимодействия.

При анализе мы описываем, как при осуществлении варианта использования внимание переходит от одного элемента к другому. Мы используем для демонстрации этого процесса различные виды диаграмм взаимодействия. При проектировании мы вдобавок описываем операции, используемые для обмена. Кроме того, мы также должны принимать во внимание то, какие у нас есть многоократно используемые системы, каркасы или унаследованные системы и какие операции они нам предоставляют. Если в этом трудно разобраться, значит, трудно будет разобраться и в проектировании.

Результатом этой деятельности станет набор проектов реализаций вариантов использования, по одному на каждый архитектурно значимый вариант использования.

Проектирование класса

Мы проектируем классы, входящие в реализации вариантов использования. Отметим, что классы обычно не завершаются, они будут участвовать во многих реализациях вариантов использования, которые будут разработаны на следующих итерациях. Инженер по компонентам собирает различные роли каждого из классов в согласованный класс как описано в подразделе «Артефакты» главы 10.

Проектирование подсистемы

Инженер по компонентам, отталкиваясь от проектирования архитектуры, проектирует подсистемы. В это время архитектор при необходимости поправляет архитектурное представление модели проектирования.

Реализация

Этот рабочий процесс реализует и тестирует архитектурно значимые компоненты, работающие с архитектурно значимыми компонентами. Результатом будет базовый уровень архитектуры, реализующий обычно менее 10% от объема вариантов использования. В этом подразделе мы рассмотрим реализацию архитектуры, класса или подсистемы и интеграцию системы (см. главу 10).

Реализация архитектуры

Компоненты, необходимые для реализации сервисных подсистем, определяются на основании архитектурных представлений моделей проектирования и развертывания. Исполняемые компоненты проецируются на узлы компьютерной сети, на которых они смогут выполняться. Затем архитектор описывает происходящее в архитектурном представлении модели реализации (см. подраздел «Деятельность: Реализация архитектуры» главы 10).

Реализация класса и реализация подсистемы

Во время рабочего процесса проектирования инженер по компонентам проектирует несколько классов, относящихся к созданию базового уровня архитектуры. Этот базовый уровень представляет собой облегченную исполняемую версию системы, которую мы строим. В ходе этого процесса инженер по компонентам реализует классы в виде файловых компонентов (это обычно один или более компонентов, реализованных в виде сервисной подсистемы модели проектирования). Осуществляемая затем деятельность *Провести тестирование модуля* подтверждает правильную работу отдельных компонентов (см. подразделы «Деятельность: Реализация подсистемы» и «Деятельность: Реализация класса» главы 10).

Сборка системы

На основе той малой доли вариантов использования, которые были реализованы в ходе текущей итерации, системный интегратор разрабатывает план сборки. Затем он последовательно включает подсистемы и соответствующие им компоненты в исполняемый базовый уровень архитектуры (см. подраздел «Деятельность: Сборка системы» главы 10).

Пример. *Три билда.* Системный интегратор предполагает создать три исходных билда (см. рис. 14.3).

Пример, приведенный на рис. 14.3, включает в себя три билда, каждый из которых пройдет тестирование модулей:

1. Классов подсистемы *Управление банковскими счетами*, в которые обернута унаследованная банковская система.
2. Классов пакета *Счета покупателя*, которые участвуют в реализации варианта использования *Оплатить счет* вместе с первым билдом.
3. Классов пакетов *Счета продавца* и *Счета покупателя*, которые участвуют в реализации варианта использования *Переслать торговый объект*. Эти подсистемы изначально содержат обобщенные классы абстрактного варианта использования *Переслать торговый объект*. Позднее эти обобщенные классы будут выделены в подсистему, которая может многократно использоваться также и некоторыми другими подсистемами. К этому билду присоединяется также пакет Java RMI.

Пока мы занимались фазой анализа и планирования требований, для управления работами мы могли обойтись и без специальных утилит (не скажем, правда, что это было легко). Но отказываться от использования утилит на фазе проектирования совершенно непрактично. Например, сейчас у нас появляется необходимость в управлении версиями, и мы обзаводимся средствами управления конфигурацией. Мы должны быть в состоянии окунуть взглядом то, что мы делаем. В начале фазы проектирования это можно сделать при помощи бумаги и карандаша, но к концу фазы без утилит контроля базового уровня архитектуры это невозможно.

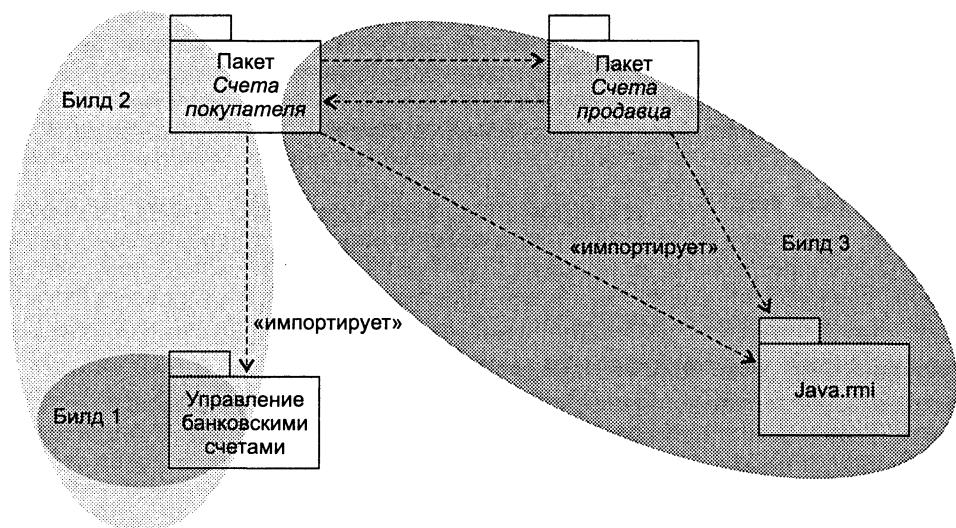


Рис. 14.3. Первая итерация предполагает три билда. Заметим, что второй билд включает в себя результаты первого

Еще один пример. Сотрудники должны быть в состоянии использовать «производственный» язык, например Java, и набор утилит, а их использование приведет к преимущественному применению сред разработки и получению сотрудниками опыта в использовании новых утилит и методов. В этом случае первое приращение более правильно использует инфраструктуру системы — системное программное обеспечение и программное обеспечение среднего уровня. Прототип становится проще превратить в реальную систему.

Тестирование

В этом процессе основное внимание уделяется проверке того факта, что подсистемы и их интерфейсы всех типов (как подсистемы проектирования, так и сервисные подсистемы) и всех уровней (от системного программного обеспечения до специфического уровня приложений) работоспособны (см. главу 11). Разумеется, мы в состоянии протестировать только исполняемые компоненты. Если они работают, мы можем быть до некоторой степени уверены в том, что и остальные компоненты (других моделей) также правильны.

«Начиная с нижних уровней архитектуры» означает, что мы тестируем распределенное управление объектами, сохранение и восстановление объектов (персистентность) и параллельные объекты и другие механизмы нижнего уровня. В список тестов входит проверка не только функциональности, но и достаточной производительности. Часто нет необходимости отдельно тестировать нижние уровни, все, что нам нужно — это проверить, как верхние уровни используют функциональность нижних.

Для специфических и общих уровней приложений тестирование определяет, насколько хорошо масштабируется система при добавлении к ней новых подсистем, использующих старые интерфейсы.

Планирование тестирования

Инженер по тестированию выбирает задачи, которые должны решаться базовым уровнем архитектуры. Такой задачей может быть, например, выполнение сценария варианта использования с заданным временем отклика при определенном уровне загруженности системы (см. подраздел «Деятельность: Планирование тестирования» главы 11).

Разработка теста

На основе этих целей инженер по тестированию определяет необходимые тестовые примеры и разрабатывает процедуры тестирования для последующего тестирования целостности подсистем, а затем и системного тестирования всего базового уровня (см. подраздел «Деятельность: Планирование тестирования» главы 11).

Проведение тестирования целостности

Компоненты, оттестированные по отдельности, передаются на тестирование целостности. Тестер целостности тестирует каждый билд (см. подраздел «Деятельность: Проведение тестирования целостности» главы 11).

Проведение тестирования системы

Когда система, определенная архитектурно значимыми вариантами использования, собрана, ее проверяет системный тестер. Для фазы проектирования эта система (базовая версия полной системы) — базовый уровень архитектуры. Тестер сообщает о найденных ошибках инженеру по компонентам или архитектору, а те должны их исправлять (см. подраздел «Деятельность: Проведение тестирования системы» главы 11).

Инженеры по тестированию пересматривают результаты системного тестирования, чтобы убедиться, что они выполнили исходные задачи, или определить, как для выполнения исходных задач следует модифицировать тестовые примеры.

Пример. Непредсказуемость перевода представляет собой серьезный риск. Системное тестирование показало, что большая часть функциональности может быть признана удовлетворяющей параметрам качества, за одним исключением. Когда тестер выполняет вариант использования *Оплатить счет*, часть результатов неверна. Оболочка унаследованной системы, подсистема *Управление счетами*, иногда дает непредсказуемые результаты при попытке перечисления суммы «с копейками», то есть не целого числа долларов, например \$134.65. В других случаях переводы проходят нормально, например для суммы \$124.54. Тестер указывает на эту проблему, и архитектор помечает ее как серьезный риск. Менеджер проекта назначает команду быстрого реагирования для немедленного его устранения.

Определение деловых перспектив

Необходимость снижения рисков и создания базового уровня архитектуры вызваны тем, что проект подошел к такому моменту, когда команда может начать фазу построения. Этот переход требует полной уверенности в том, что, создавая продукт, мы уложимся в ограничения бизнеса. В особенности нас беспокоят два ограничения. Одно из них — это смета — график работ, усилия и затраты при заданном качестве. Другое — это прибыль на вложенный капитал или другой подобный параметр, показывающий, будет ли создаваемая система успешной с коммерческой точки зрения.

В конце фазы анализа и планирования требований производитель может оценивать достоинства бизнес-плана очень приблизительно — по крайней мере, в случае новой, большой или сложной системы. В ходе фазы проектирования понимание проекта росло, и в конце этой фазы наступил момент значительно уточнить первоначальные оценки. Это позволит подготовить уточненное бизнес-предложение и создать уточненный бизнес-план, более соответствующие деловой практике.

Подготовка бизнес-предложения

Команда разработчиков решает вопрос о переходе к фазе построения на основании деловых документов. Деловым соглашением в данном случае будет контракт с внешним клиентом, заявка другого отдела той же компании или приказ о разработке продукта для массовой продажи.

Мы замечали, что оценка программного обеспечения часто основывается на размере проекта и производительности организации-разработчика. Производительность разработчиков определяется опытом разработок, а размер прикидывается на основе информации, полученной в фазе проектирования. Чтобы создать реалистичную оценку, следует продолжать работы в фазе проектирования до тех, пока нам не будет обеспечено хорошее понимание предстоящей нам в фазе построения работы. Мы имеем в виду создание базового уровня архитектуры. Кроме того, если в проекте присутствуют риски — любой величины, — следует исследовать их достаточно глубоко, чтобы получить ясное представление, сколько времени и усилий потребует их преодоление.

Уточнение доходности инвестиций

Квинтэссенция бизнес-плана содержится в следующем вопросе: будут ли прибыли, полученные от использования или продаж продукта, превышать затраты на его разработку? Будут ли продукты удовлетворять потребности рынка (или внутреннего применения) в течение времени, достаточного для получения этих прибылей?

Теперь организация-разработчик в состоянии оценить затраты на фазы построения и внедрения и изложить их в виде бизнес-предложения. Это предложение — одна сторона бизнес-плана. Для другой его стороны, оценки прибыли, которую принесет продукт, не существует точных формул.

В случае разработки программного обеспечения для массовой продажи возможное число проданных копий, цена, при которой продукт будет пользоваться спросом, и период, в течение которого будут происходить продажи, должны определяться отделом маркетинга и утверждаться руководством. В случае программ для внутреннего использования оценить ожидаемый эффект должен отдел, по заказу которого они создаются. Запас на ошибку при оценке потенциальной прибыли обычно не мал, но эта процедура по крайней мере позволяет получить базу для сравнения доходов с затратами.

Оценка результатов итераций и фазы проектирования

При завершении каждой итерации она оценивается по набору критериев из плана итерации, который был разработан до ее начала. Группа оценки просматривает результаты каждой итерации, проверяя, в самом ли деле архитектура базового уровня соответствует первоначальным задачам и снижает риски.

Если было проделано несколько итераций, результат первой из них может быть только первым наброском архитектуры. Этот набросок может быть набором достаточно хорошо описанных архитектурных представлений различных моделей: вариантов использования, анализа, проектирования, развертывания и реализации. Результатом второй итерации будет второй набросок архитектуры. Результатом последней итерации — базовый уровень архитектуры. В заключение каждой итерации менеджер проекта, обычно вместе с группой оценки, оценивает, как в действительности соотносятся достигнутые результаты с предварительно заданными критериями. Если проект не соответствует этим критериям, менеджер проекта переориентирует его, как описано в главе 12. В ходе фазы проектирования это может означать перенос незаконченного дела на следующую итерацию.

Поскольку менеджер проекта встречался с клиентами и другими заинтересованными лицами и совместно с ними прикладывал усилия для достижения каждой промежуточной вехи, главная веха (конец фазы) обычно кажется им менее болезненной. Команда налаживает с клиентом более тесную связь, чем в случае работы по модели водопада. Клиент получает возможность вносить улучшения в разрабатываемые модели в ходе всего процесса разработки.

Итак, в конце фазы проектирования оценка убеждает заинтересованных лиц, что на завершенной фазе было проведено снижение значительных рисков и построен стабильный базовый уровень архитектуры. Это, в свою очередь, убеждает их в том, что система может быть построена в соответствии с планом проекта и заявленной стоимостью фазы построения.

Планирование фазы построения

В конце фазы проектирования менеджер проекта приступает к детальному планированию первой итерации построения и изложению невыполненного в более общих понятиях. Число требуемых итераций зависит от размеров и сложности проекта. Менеджеры проекта обычно планируют две или три, иногда, в случае большого и сложного проекта, — четыре или более. В каждой итерации они намечают несколько билдов, каждый из которых будет добавлять небольшой кусочек, создавая запланированный результат.

Руководство проекта все еще видит в списке рисков немало оставшихся рисков. Менеджер проекта планирует порядок исследования сохранившихся рисков с целью снижения каждого из них до того, как они всплынут в билде или последовательности итераций. Принцип действия остается прежним: понижать риски до того, как они нарушают последовательность билдов.

Фаза проектирования лишь частично заполнила каждую из моделей. Менеджер проекта определяет порядок работы над оставшимися вариантами использования и сценариями и завершением моделей. Его решения определят порядок билдов и итераций. В больших проектах для снижения общего времени работы путем привлечения дополнительных работников менеджер проекта выбирает работы, которые могут проводиться параллельно. Разработка больших промышленных систем требует от команды выделения в проекте параллельных ветвей и параллельной работы, поскольку проект имеет жесткие ограничения по времени. Команда ищет пути занять в разработке большее количество людей, часто на порядки.

Этот путь основан на подсистемах, составляющих базовый уровень архитектуры. В рабочем процессе проектирования (движимые пакетами анализа) мы обнаруживаем подсистемы различного уровня. Подсистемы имеют интерфейсы, и одна из задач верхнего уровня – определение этих интерфейсов. Интерфейсы являются ядром архитектуры. Мы подготовились к параллельной работе, определив и описав подсистемы и интерфейсы.

Один разработчик отвечает за сервисную подсистему внутри подсистемы проектирования. Группа отвечает за подсистему проектирования.

Если одиночки или малые группы работают с высокой степенью независимости, интерфейсы, соединяющие области, в которых они работают, должны быть надежны. Чтобы подчеркнуть важность описания этих интерфейсов, их иногда называют контрактами. Контракт заключается между текущей группой разработчиков и группой, которая будет работать с этим интерфейсом в дальнейшем. Этот интерфейс, будучи правильно создан, делает возможным применение архитектуры подключаемых модулей. Впоследствии, если контракт не разорван, разработчики могут заменить подсистему другой. Построение систем, взаимодействующих через контрактный интерфейс (или его эквивалент), – основной принцип построения системы систем, которые мы обсуждали в подразделе «Вариант использования» главы 7 «Моделирование больших систем».

Основные результаты

Результаты фазы проектирования:

- Вероятно, полная бизнес-модель (или модель предметной области), определяющая контекст системы.
- Новые версии всех моделей – вариантов использования, анализа, проектирования, развертывания и реализации. В конце фазы проектирования эти модели завершены менее чем на 10%. Особняком стоят модели вариантов использования и анализа, в которые, для того чтобы удостовериться, что требования поняты правильно, может быть включено значительно больше (в некоторых случаях до 80%) вариантов использования. Большинство вариантов использования разобрано, чтобы удостовериться в том, что ни одного архитектурно значимого варианта использования мы не пропустили, и иметь возможность оценить затраты на их проработку.
- Исполняемый базовый уровень архитектуры.
- Описание архитектуры, включающее в себя представления моделей вариантов использования, анализа, проектирования, развертывания и реализации.
- Переработанный список рисков.
- План проекта на фазы построения и внедрения.
- Предварительное руководство пользователя (возможно).
- Законченный бизнес-план, включая бизнес-предложение.

15 Фаза построения приводит к появлению базовых функциональных возможностей

Основная цель этой фазы — создать программный продукт, готовый к предварительному распространению, который иногда называют «бета-версией». Этот продукт должен соответствовать целям, для которых он будет применяться, и поддерживать заявленные требования. Построение должно происходить в соответствии с бизнес-планом.

В конце фазы проектирования мы довели разрабатываемую систему до состояния исполняемого базового уровня архитектуры. На предыдущих фазах особо опасные и просто важные риски были снижены до обычного уровня. В таком виде мы можем управлять ими при помощи плана построения. В ходе фазы проектирования команда разработчиков заложила основание для архитектурно значимых элементов моделей проектирования и развертывания. Это основание включает в себя подсистемы, классы (в том числе и активные), а также компоненты и интерфейсы между ними. Также в основание входят реализации значимых вариантов использования. Это разбиение достигается путем детализации всего лишь 10% объема вариантов использования. Напомним, что мы уже определили почти все варианты использования (обычно около 80%), но не пытались полностью их детализировать, поскольку это не входило в задачи фазы проектирования. Мы сделаем это на фазе построения.

Введение

Команда, занимающаяся фазой построения, начинает свою работу, имея в качестве исходных данных исполняемый базовый уровень архитектуры, и в ходе серии итераций и приращений разрабатывает продукт, предназначенный для испытания его в среде пользователей в ходе их обычной работы (этот процесс также называется бета-

тестированием). Команда разработчиков детализирует оставшиеся варианты использования и сценарии, при необходимости модифицирует описание архитектуры и осуществляет в ходе дополнительных итераций рабочие процессы, заполняя оставшиеся пустые места в моделях анализа, проектирования и реализации. Она собирает подсистемы и тестирует их, собирает систему и тестирует ее тоже.

При переходе проекта от фазы проектирования к фазе построения происходит смещение акцентов. Если фазы анализа и планирования требований и проектирования можно сравнить с исследованием, то фаза построения аналогична производству. Акцент смещается от наполнения базы знаний, необходимой для построения проекта, к реальному построению системы или продукта с заданным графиком работ, затратами и усилиями.

В ходе фазы построения менеджер проекта, архитектор и старшие разработчики обеспечивают расстановку приоритетов вариантов использования, группировку их по билдам и итерациям и реализацию в таком порядке, который позволяет исключить переделку уже сделанного.

Они поддерживают в актуальном состоянии список рисков, постоянно уточняя его, чтобы он отражал действующие текущие риски проекта. Задача разработчиков — закончить эту фазу, компенсировав все риски (исключая те, которые возникают в процессе работы и будут компенсированы в фазе внедрения). Архитектор наблюдает за тем, чтобы разработчики придерживались архитектуры и при необходимости модифицирует ее, чтобы учесть возникающие при построении изменения.

В начале фазы построения

Менеджер проекта создает план фазы построения в конце фазы проектирования. Когда он запрашивает разрешение начинать построение у тех, кто отвечает за финансирование, он может модифицировать план фазы построения в том случае, если изменятся обстоятельства.

Одной из причин такого изменения может быть возможный разрыв между фазами проектирования и построения. Лица, финансирующие разработку, могут разрешить выполнение фазы построения немедленно, что позволит менеджеру проекта и его команде продолжать работу без пауз, сохранив детальное знание проекта. К сожалению, между подачей предложения и заявки и выигрышем тендера или другой формой согласия на проведение работ могут пройти месяцы. Тем временем менеджер проекта и его команда расходятся по другим проектам и не всегда могут быть собраны вместе в прежнем составе после получения приказа «На старт!». В наихудших обстоятельствах на фазе построения у проекта будет новый менеджер и по большей части новые сотрудники.

Второе обстоятельство, которое может потребовать от менеджера изменения плана фазы, состоит в том, что финансирование может оказаться меньше, а график работ жестче, чем планировалось. Область действия системы при этом может быть соответственно уменьшена, но может и остаться прежней. В тот момент, когда мы начнем готовиться к старту работ фазы построения, ситуация может измениться — сильнее или слабее — по сравнению с тем положением дел, в соответствии с которым мы в конце фазы проектирования планировали будущее построение.

Нравится нам это или нет, — а обычно нам это не нравится, — менеджер проекта частично изменяет план. Едва ли не в каждом случае мы должны адаптировать план, составленный в конце фазы проектирования, под наличные людские ресурсы и график, предписанный нам заинтересованными лицами.

Персонал для осуществления фазы

В фазе построения работа начинается с архитектуры (то есть все элементы, включенные в модели, архитектурно значимы). Сервисные подсистемы, которые обнаружил архитектор на фазе проектирования, не закончены — в них реализованы только базовые варианты использования и их базовые сценарии. На эту работу менеджер проекта должен дополнительно выделить людей.

Базовый уровень архитектуры с его представлениями и интерфейсами и будет тем источником, из которого менеджер проекта будет раздавать отдельные части работы. Каждая подсистема получит своего разработчика — инженера по компонентам, который за нее отвечает. Как мы говорили в главе 9, обычно разработчик, отвечающий за подсистему, отвечает также и за классы этой подсистемы. Разработчик, отвечающий только за один класс, — это крайне непрактично. Это слишком мелкая разбивка работы.

На фазе построения для выполнения основных работ необходимо набрать людей для заполнения следующих позиций: инженер по вариантам использования, инженер по компонентам, инженер по тестированию, системный интегратор, тестер интеграции и системный тестер. По сравнению с фазой проектирования, число людей, занятых в работе, возрастает примерно вдвое (см. раздел «Требуемые ресурсы» главы 12). Таблица 13.1 показывает приблизительную долю оставшейся работы для разных рабочих процессов фазы построения — 60% в анализе и 90% в проектировании, реализации и тестировании.

Кроме вышеперечисленных сотрудников, в работе продолжает участвовать архитектор, однако время, которое он затрачивает на эту фазу, обычно мало. Кроме того, поскольку у нас осталось неопределенным приблизительно 20% требований, системному аналитику и спецификатору вариантов использования также придется, чем заняться.

Задание критериев оценки

На итерации фазы построения мы добиваемся выполнения специфических критериев, для каждого проекта своих. В действительности они определяются во время разработки вариантов использования. Как мы отмечали в предыдущей главе, варианты использования связаны с функциональными требованиями. Они также содержат относящиеся к этим вариантам использования нефункциональные требования, например производительность. Такие требования используются для понижения рисков. Каждый билд или итерация реализуют набор вариантов использования. Критерии оценки набора вариантов использования при этом основаны на функциональных и нефункциональных требованиях к этому набору.

Эти критерии оценки, относящиеся к вариантам использования, позволяют разработчикам лучше почувствовать момент окончания очередного билда или итерации.

Кроме того, в ходе фазы построения разрабатываются дополнительные материалы, для которых также нужны критерии оценки. Например:

Материалы для пользователя. Первая попытка создать печатные материалы для конечных пользователей — руководство пользователя, текст подсказки, примечания к выпуску, инструкции оператору — предпринимается на фазе построения. Критерий завершения: достаточно ли этих материалов для поддержки пользователя в фазе внедрения?

Материалы для обучения. В ходе этой фазы создается первая версия материалов для обучения, таких, как слайды, конспекты, учебники и экзаменационные задания. Критерий завершения: достаточно ли их для поддержки пользователей в ходе фазы переноса?

Для фазы построения в целом критерии оценки состоят в том, достаточно ли устойчив и работоспособен созданный продукт для того, чтобы перенос бета-версий в пользовательскую среду не сопровождался опасностью проявления неучтенных рисков для организации-разработчика или сообщества пользователей.

Типичный рабочий процесс итерации в фазе построения

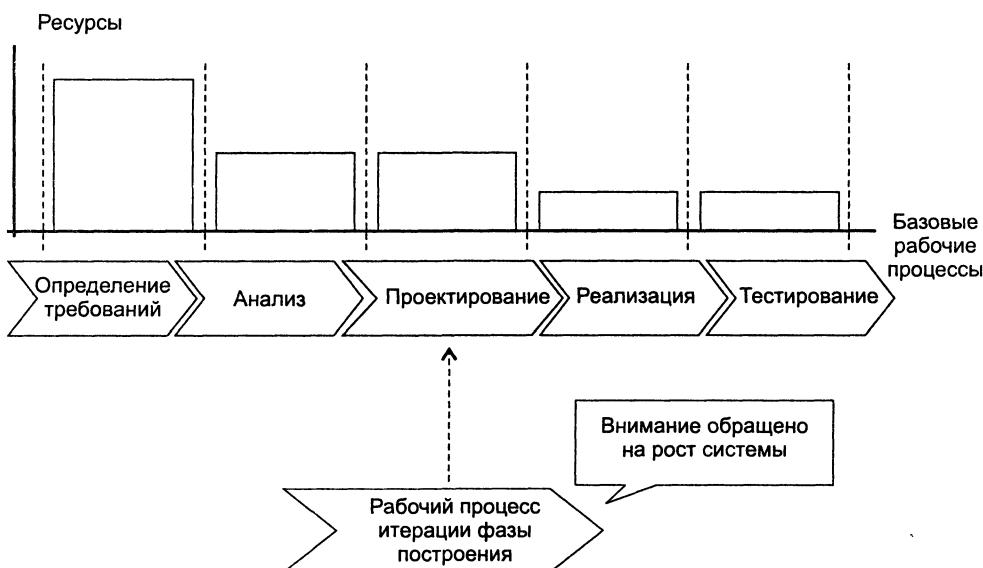


Рис. 15.1. На итерациях фазы построения осуществляются пять базовых рабочих процессов — определение требований, анализ, проектирование, реализация и тестирование

Типичная итерация включает в себя пять рабочих процессов, показанных на рис. 15.1, — определение требований, анализ, проектирование, реализация и тестирование. Происходит определение и анализ оставшихся требований. Нагрузка на эти два рабочих процесса относительно невелика, поскольку большая часть этой работы была выполнена на двух предыдущих фазах. Проектирование играет более важную роль. Кроме того, на этой фазе осуществляется большая часть работы по реализации и тес-

тированию (размеры прямоугольников приведены только для иллюстрации и могут изменяться в зависимости от проекта). Рабочие процессы детально описаны в части 2, а в этой главе мы рассмотрим только ту их часть, которая выполняется в ходе построения. И снова мы можем выполнять четыре набора видов деятельности параллельно. Первый из них — это пять базовых рабочих потоков, второй — это планирование итераций, описанное в разделах «Планирование предваряет деятельность», «Риски влияют на планирование проекта», «Расстановка приоритетов вариантов использования» и «Требуемые ресурсы» главы 12 и в разделе «В начале фазы построения» данной главы, третий — это бизнес-план, описанный в разделах «Определение исходных деловых перспектив» главы 13 и «Контроль бизнес-плана» данной главы, а четвертый — это оценка, описанная в разделах «Оценка итераций и фаз» главы 12 и «Оценка результатов итераций и фазы построения» данной главы. В этом разделе мы сделаем обзор пяти рабочих процессов. В следующем разделе мы опишем их детально.

На ранних итерациях фазы построения первые рабочие процессы имеют большее значение, на поздних — меньшее. В ходе итераций фазы построения центр внимания смещается от первых рабочих процессов к последним. Например, если мы построим колоколообразную кривую, иллюстрирующую уровень нашего внимания, отложив по оси X рабочие процессы, то пик кривой в ходе работы будет смещаться слева направо. Каждая удачная итерация все больше будет смещать центр внимания к процессу реализации.

Построение системы. В настоящий момент требования и архитектура стабильны. Необходимо завершить создание реализаций вариантов использования для всех вариантов использования, спроектировать необходимые подсистемы и классы, реализовать их в виде компонентов и протестировать как индивидуально, так и в составе билдов. В каждом билде разработчики берут указанный менеджером проекта и системным интегратором набор вариантов использования и реализуют его.

Управляемая вариантами использования, архитектуро-центрированная, итеративная разработка позволяет создавать программы путем разработки малых приращений и добавки каждого нового приращения к сумме предыдущих приращений, таким же образом, как и при создании исполняемого билда. Порядок последовательности приращений задается так, чтобы разработчикам никогда не приходилось откатываться на несколько итераций назад, а затем делать их снова.

Построение программ относительно малыми приращениями делает проект легче управляемым. Это уменьшает область действия рабочих потоков анализа, проектирования, реализации и тестирования до нескольких вопросов, приходящихся на одно приращение. Таким образом в значительной степени локализуются риски, дефекты и исправления в узком кругу единичного билда, что делает проще их поиск и работу с ними.

Исследование особо опасных и серьезных рисков было проведено на предыдущих фазах, но у руководителей проекта в списке рисков осталось еще немало рисков. Кроме того, пока разработчики продолжали строить билды и итерации, а пользователи — проверять приращения, могли появиться и новые. Так, многие языки программирования долгое время были в работе, и разработчики могли вполне обоснованно решить, что компилятор, который они планировали использовать, отвечает их требованиям. Однако языки развиваются, выпускаются новые версии компиляторов, а новые версии могут содержать дефекты. В одном проекте размером в 80 000 строк исходных текстов менеджер проекта в конце концов обнаружил, что повторяющиеся дефекты в последовательности тестов были вызваны ошибкой ком-

пилятора. Пришлось приостановить работу до тех пор, пока производитель компьютера не нашел в нем ошибку и не исправил ее.

Выполнение основных потоков работ — от определения требований до тестирования

В предыдущем пункте мы описали основную цель фазы построения. В данном разделе мы опишем эти виды деятельности более детально, используя в качестве руководства рис. 15.2. Как и в предыдущей главе, этот раздел организован в соответствии с порядком пяти рабочих процессов. Как и ранее, несмотря на последовательную организацию изложения, рабочие процессы могут выполняться параллельно, что и показано на рисунке.

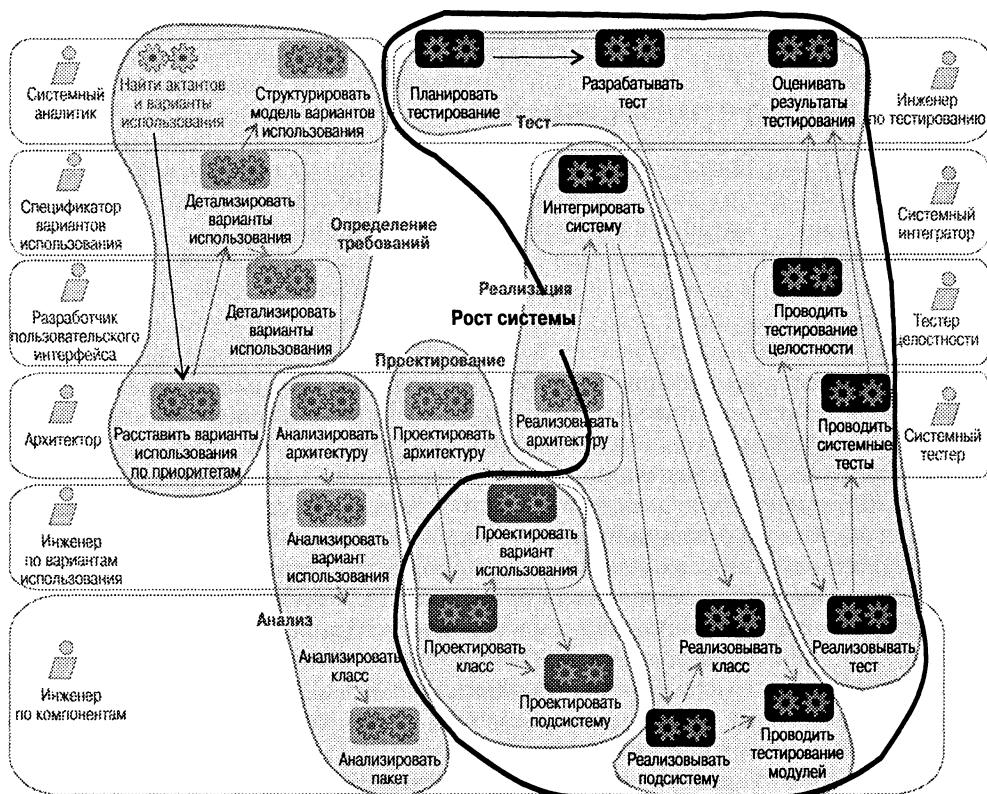


Рис. 15.2. Контуры показывают основную деятельность в ходе фазы построения

В ходе этой работы сотрудники участвуют в рабочих процессах в порядке, описанном в части 2. Они строят все артефакты — базового уровня архитектуры на ранних итерациях, прочие — на поздних.

В фазе проектирования проект может быть спроектирован и реализован менее чем на 10% общего объема вариантов использования — ровно настолько, чтобы хватило на базовый уровень архитектуры. Теперь, в фазе построения, выполняется задача наращивания мышц на этот архитектурный скелет. В результате это приводит к наполнению моделей. Мы описывали этот процесс в главах 4 и 5. В частности, на рис. 5.7 показано заполнение моделей в ходе четырех фаз. Каждый билд и каждая итерация добавляют к растущей структуре моделей дополнительные реализации вариантов использования, классы, подсистемы и компоненты.

В следующих подразделах мы опишем отдельные рабочие процессы, показанные на рис. 15.2. Однако сотрудники не обязательно выполняют работу в приведенной последовательности. Например, в начале каждой итерации мы выполняем планирование тестов для подготовки к тестированию, которое требуется провести на данной итерации. Планирование тестов должно быть выполнено до детализации вариантов использования, анализа, проектирования и реализации. Эта работа, проводимая параллельно, в данном тексте не отражена. У нас появляется повод повторно отметить, что все пять рабочих процессов повторяются на каждой итерации.

Определение требований

Начиная фазу построения, мы снова переходим к поиску вариантов использования и актантов, прототипированию пользовательских интерфейсов, детализации вариантов использования и структурированию вариантов использования. В фазе проектирования мы определили все варианты использования и актантов, мы *проработали* около 80% их объема, но детально описали порядка 20%. Из этого числа вариантов использования в дело пошла только часть (менее 10% того, что мы должны реализовать сейчас), поскольку полная детализация требовалась только для создания базового уровня архитектуры. В фазе построения, разумеется, мы пройдемся по системе и определим требования на всех уровнях, начиная с операционной системы (100% определения и детализацию).

Нахождение актантов и вариантов использования. Обычно в фазе построения остается определить лишь небольшую часть вариантов использования и актантов (менее 20%). Системный аналитик при необходимости обновляет варианты использования и актантов, входящих в модель вариантов использования.

Создание прототипа интерфейса пользователя. В фазах анализа и определения требований и проектирования мы в основном создавали прототипы пользовательских интерфейсов в том случае, если это были интерфейсы нового типа или если они были нужны нам в демонстрационных целях. Однако теперь настало время их спроектировать. Насколько серьезно мы будем этим заниматься, зависит от типа разрабатываемой системы.

Для некоторых систем — особенно для тех, в которых отдельные варианты использования требуют очень сложного интерфейса пользователя, — трудно построить интерфейс, не создав прототипа. Итак, мы строим прототип (а если нужно, так и несколько) и предлагаем пользователям рассмотреть его. На основе получаемой от них информации мы изменяем прототип до тех пор, пока он не будет удовлетворять нуждам пользователей. Проектирование интерфейса пользователя является частью рабочего процесса определения требований, а не рабочего процесса проек-

тирования (даже если название утверждает иначе) и должно быть закончено до перехода к следующим рабочим процессам. Прототип при этом перейдет в спецификацию пользовательского интерфейса (детали см. в главе 7).

Для систем, которые предполагается продавать массовыми тиражами, мы строим прототипы пользовательских интерфейсов, даже если они и не очень сложны. Цена замены неудовлетворительного интерфейса на проданном тираже будет чрезвычайно велика.

Определение приоритетности вариантов использования. В фазе проектирования мы определяем приоритетность вариантов использования, необходимых для разработки базового уровня архитектуры. В этой фазе по мере определения вариантов использования мы добавляем их к списку вариантов использования в соответствии с приоритетом (см. раздел «Расстановка приоритетов вариантов использования» главы 12 и подраздел «Деятельность: Определение приоритетности вариантов использования» главы 7).

Детализация вариантов использования. Спецификаторы вариантов использования завершают детализацию, проводя ее в порядке важности каждого из пропущенных вариантов использования и сценария варианта использования.

Структурирование модели вариантов использования. Системный аналитик может пожелать улучшить структуру модели вариантов использования. Однако, поскольку к этому моменту система уже имеет стабильную архитектуру, изменения могут касаться в основном тех вариантов использования, которые не были задействованы в работе. Каждый измененный вариант использования требует соответствующих реализаций варианта использования в моделях анализа и проектирования.

Анализ

В этом разделе мы вновь рассмотрим анализ архитектуры, вариантов использования, классов и пакетов, начатый на фазе проектирования. Тогда мы нуждались в рассмотрении только тех вариантов использования, которые важны для построения архитектуры или необходимы для поддержки бизнес-предложения. Чтобы дать читателю представление, где мы находимся в настоящий момент, скажем, что на фазе проектирования мы проанализировали 40% объема вариантов использования. Это почти половина объема вариантов использования и порядка 80% того, что нам обычно необходимо для поддержки предложения. Не следует воспринимать эти цифры буквально. Мы снова повторим, что они могут быть разными в зависимости от особенностей данного проекта. Теперь, на фазе построения, мы должны проработать все варианты использования, но не обязаны при этом расширять с их помощью модель анализа.

Как рассматривалось в главе 8 (см. раздел «Роль анализа в жизненном цикле программы»), бывают случаи, когда модель анализа нового продукта после фазы проектирования не поддерживается. Однако нередко менеджер проекта может продолжать использовать модель анализа на фазе построения и далее, до самого конца проекта, и даже после завершения его жизненного цикла. Поскольку последний вариант очень сложен, мы в будущем рассмотрим его подробно. Основная разница между фазами проектирования и построения в том, что при построении мы заканчиваем модель анализа. Модель анализа, которую мы имеем в конце проектирования, — это

архитектурное представление, она сильно смещена в сторону архитектуры. Теперь архитектурное представление модели анализа будет одной из частей полной модели анализа. В конце фазы построения мы получим полную модель анализа. Архитектурное представление будет лишь малым ее подмножеством.

Анализ архитектуры. К концу фазы проектирования архитектор заканчивает создание архитектурного представления модели анализа. В ходе фазы построения он занимается этой деятельностью только в том случае, если новые версии моделей требуют изменений в архитектуре.

Анализ варианта использования. В фазе проектирования архитектор использует только те варианты использования, которые важны для архитектуры, и применяет их для построения архитектурного представления модели анализа. В ходе каждой итерации фазы построения мы расширяем модель анализа новыми вариантами использования, добавляемыми в ходе этой итерации.

Анализ класса. Инженер по компонентам продолжает работу, начатую в фазе проектирования.

Анализ пакета. Архитектор определяет пакеты в фазе проектирования, а в фазе построения уточняет их по мере появления новых вариантов использования. Инженер по компонентам поддерживает пакеты в ходе фазы.

Проектирование

В этой фазе мы обычно проектируем и реализуем оставшиеся 90% вариантов использования, не потребовавшихся при разработке базового уровня системы. Мы вновь подчеркнем, как уже делали это при описании рабочего процесса проектирования, что этот и другие рабочие процессы повторяются в ходе каждой итерации.

Проектирование архитектуры. В фазе построения архитектор обычно не добавляет новых подсистем проектирования или сервисных подсистем. Эти элементы уже существуют в первом приближении в базовом уровне архитектуры.

Архитектор может добавлять подсистемы, если они похожи на уже существующие подсистемы или, возможно, являются их альтернативой. Так, если имеется подсистема, реализующая коммуникационный протокол, а мы добавляем другой коммуникационный протокол, которому не нужен новый интерфейс, то мы можем добавить для этого протокола новую подсистему.

Поведение сервисных подсистем обычно бывает порождено частями одного варианта использования или набора вариантов использования. Иначе говоря, сервисная подсистема исполняет одну основную роль, помогая реализации одиночного варианта использования. От 40 до 60% обязанностей сервисной подсистемы унаследовано от этого варианта использования. Когда эта доля особенно велика, например 80%, менеджер проекта оценивает, можно ли закончить оставшиеся 20% сервисной системы в том же самом билде или их придется отложить на потом. Сразу же закончить создание сервисной подсистемы — это хорошая идея даже в том случае, если другие части подсистемы берутся из вариантов использования с приоритетом более низким, чем у основного. Даже если эти пропущенные варианты использования имели более низкий приоритет и могли подождать, более продуктивно будет закончить модуль, раз уж мы за него взялись.

С одной стороны, работа с низкоприоритетным вариантом использования в этот момент заставит инженера по компонентам разработать данную подсистему за один прием. Ему гораздо проще сделать ее структурно правильной, рассматри-

вая всю подсистему за один раз. Если ему или кому-либо еще придется возвращаться к подсистеме в ходе создания последующих билдов, чтобы доработать пропущенные ранее низкоприоритетные варианты использования, он может обнаружить несоответствия, которые сделают необходимыми изменения проекта. Он может переписать плохо работающие части подсистемы, добавляя к ней код, порожденный низкоприоритетными вариантами использования. Таким образом, включение в подсистему частей, основанных на вариантах использования, которые будут спроектированы позже, хорошо тогда — и только тогда, — когда влияние этих «поздних частей» вполне понятно. Причина этого в том, что лучше сделать кусок работы как единое целое, чем рассредоточить его части на несколько итераций.

С другой стороны, проработка низкоприоритетных вариантов использования в ходе создания ранних билдов противоречит нашему общему правилу исследования вариантов использования. Общее правило состоит в том, что если подхватить в придачу к высокоприоритетному несколько низкоприоритетных вариантов использования выгодно и не вызывает чрезмерно больших затрат времени, это следует сделать. Поэтому, однако, будет правильно заниматься только проектированием низкоприоритетных вариантов использования, но не их реализацией или тестированием. Менеджер проекта должен отложить эту деятельность (проектирование, реализацию и тестирование низкоприоритетных вариантов использования) на тот бидд, при создании которого будут обрабатываться объекты с аналогичным приоритетом.

Архитектор улучшает представления архитектуры моделей проектирования и развертывания, отражая опыт, появившийся у нас в ходе фазы построения. Однако в общем мы завершили архитектуру к концу фазы проектирования, и все, что делаем сейчас, — это вносим изменения. Оценку прочей деятельности по этому рабочему процессу (варианты использования, классы и подсистемы) можно найти в главе 9. Здесь достаточно просто сказать, что проектированию на фазе построения уделяется огромное внимание (а реализации еще большее), это показано на рис. 15.1. Результатом проектирования будут модели проектирования и развертывания. Обе они поддерживаются в ходе всего цикла разработки и переносятся на следующие циклы. Модель проектирования является чертежом модели реализации и собственно реализации.

Реализация

В ходе этого рабочего процесса реализуются и проходят тесты модулей все компоненты, первоначально разработанные в модели проектирования. Результат этого процесса после некоторого количества итераций плюс сборка и тестирование системы — готовая система с базовыми функциональными возможностями, реализующая 100% объема вариантов использования. В этом пункте мы рассмотрим такие виды деятельности, как реализация архитектуры, класса и подсистемы, проведение тестирования модулей и сборка системы.

В ходе этого рабочего процесса выполняется большая часть работы над проектом, построение компонентов. Эта деятельность описана в главе 10. Каждый компонент заполняется кодом все больше и больше, бидд за биддом, итерация за итерацией, до самого конца фазы построения и «наполнения» всех компонентов.

Реализация архитектуры. К этому моменту архитектура полностью завершена. Роль архитектора, за исключением постоянного надзора, сводится к внесению необходимых изменений.

Реализация класса и реализация подсистемы. Инженер по компонентам реализует классы и подсистемы модели реализации. Он также реализует классы-заглушки, необходимые для построения билдов.

Выполнение тестов модулей. Инженер по компонентам отвечает за тестирование тех компонентов, которые он реализовал. При необходимости он корректирует проект и реализацию этих компонентов.

Сборка системы. Системный интегратор создает план сборки системы, в котором изложена последовательность билдов. Этот план устанавливает, какие варианты использования или их сценарии должен реализовывать каждый из билдов. Варианты использования и сценарии привязываются к подсистемам и компонентам.

Системные интеграторы обычно считают, что желательно начинать построение с нижних уровней иерархии многоуровневой архитектуры, то есть с уровня системного программного обеспечения или программного обеспечения среднего уровня (см. рис. 4.5). Последующие билды распространяются вверх по уровням, на общий или специфический уровень приложений. Сложно складывать билды, не имея функций поддержки, предоставляемых нижними уровнями.

Так, например, отдел программного обеспечения одной химической компании запланировал добавлять по приращению в среднем каждые две недели. На проектах в Microsoft, по слухам, билд делается каждый день — пока не будет создан весь код проекта. Каждый билд подвергается тестированию — тестированию новых добавлений и регрессионному тестированию добавленного кода вместе со старым, чтобы убедиться, что разработанный код работоспособен. Этот процесс ежедневного создания билдов предполагает ежедневную проверку совместимости свежих модулей проверяемого кода со вчерашними. Такая практика давит на разработчиков «непрерывным билдом». В то же время, однако, она снижает долгосрочное давление на организацию, поскольку проблемы со сборкой выявляются при тестировании обычно каждую ночь и исправляются на следующий же день. Билд каждый день может означать постоянный цейтнот, а может и не означать. Разработчики проверяют свой код, пока они еще помнят его. Мало смысла в проверке кода, который уже позабыт, — так вполне можно сорвать билд. Однако отдельные разработчики, проверяя свой код в сроки, определенные планом сборки, находятся в цейтноте.

Продолжительность каждого периода построения билда — это вопрос, который каждая организация решает для себя сама. Руководящий принцип: строить билды настолько часто, чтобы иметь возможность пользоваться преимуществами частых билдов.

Чтобы каждый билд оставался небольшим, системный интегратор часто требует установки заглушки или драйвера для имитации компонента, который еще не создан. Заглушка — это простейший элемент, дающий ответ на запрос или все запросы, которые данный компонент получает от других, возможно, незаконченных, компонентов билда. В свою очередь, драйвер посылает запрос другим компонентам, входящим в тестируемый билд. Заглушки и драйвера просты и поэтому обычно не создают дополнительных проблем.

Итак, системный интегратор принимает во внимание порядок сборки компонентов в функционирующую тестируемую конфигурацию. Он документирует свои решения в виде плана сборки, показывая, когда приходит время каждого из билдов принять участие в сборке и тестировании. Системный интегратор на основании плана построения соединяет готовые классы и классы-заглушки вместе в ис-

полняется билд. Он создает очередные дополненные билды, а тестеры целостности проводят их тестирование, как тестирование целостности, так и регрессионное. На последнем шаге каждой итерации и фазы целиком системный интегратор завершает связывание системы, и уже системные тестеры проводят ее системное и регрессионное тестирование.

Это планирование задает порядок билдов на каждую итерацию и последовательность итераций фазы построения. Поскольку нередки зависимости компиляции верхних уровней многослойной архитектуры от нижних, системные интеграторы должны планировать компиляцию сперва модулей нижних уровней архитектуры, а лишь затем верхних.

Тестирование

Усилия инженеров по тестированию по определению объектов тестирования, придумыванию для них тестовых примеров и процедур тестирования, описанные в главе 11, ведут к плодотворности фазы построения. Это основная деятельность фазы построения, что показано на рис. 15.1. Хотя за проведение тестов модулей отвечают инженеры по компонентам, инженеры по тестированию оказывают им техническую поддержку. Однако инженеры по компонентам и их коллеги, тестеры целостности и системные тестеры отвечают и за тестирование билдов, то есть приращений в конце итераций, и — в особенности — за финальный билд, полностью реализованную систему.

Планирование тестирования. Инженеры по тестированию выбирают задачи для тестирования последовательных билдов и системы целиком.

Разработка теста. Инженеры по тестированию определяют, как протестировать требования к билду в порядке проверки тестируемых требований. Для этой цели они создают тестовые примеры и процедуры тестирования. Из тестовых примеров и процедур, разработанных для предыдущих билдов, они выбирают те, которые по-прежнему относятся к делу, и модифицируют их для использования в регрессионном тестировании последующих билдов. Инженеры по тестированию проверяют компоненты, которые, как изначально было определено при планировании тестов, должны тестироваться вместе. Цель тестирования целостности состоит в проверке интерфейсов между компонентами и тестировании совместной работы компонентов.

Проведение тестирования целостности. Тестеры целостности выполняют тестовые примеры в соответствии с процедурами тестирования. Когда один билд прошел тестирование, системные интеграторы добавляют к системе что-то новое, создавая еще один билд, и как только он появляется, тестеры целостности продолжают свою работу. Если тест целостности обнаружил ошибки, тестеры фиксируют их и уведомляют менеджера проекта. Менеджер проекта (или уполномоченный им сотрудник) определяет дальнейшие действия. Этим уполномоченным сотрудником может быть системный интегратор, обладающий достаточными техническими знаниями. Следующим шагом может быть, например, дальнейшая работа с тем же самым билдом, или, в случае особо серьезной ошибки, выделение для ее исследования группы особо квалифицированных сотрудников.

Проведение тестирования системы. Когда последовательность билдов дойдет до конца итерации, она достигнет состояния частично построенной системы и попадет под юрисдикцию системного тестера. Системный тестер выполняет тесто-

вые примеры для всей системы, руководствуясь системными процедурами тестирования. Если этот тест обнаружит ошибки, системный тестер зафиксирует их и уведомит менеджера проекта или уполномоченного им сотрудника о необходимости внесения исправлений.

В конце последней итерации фазы построения системный тестер тестирует систему, обладающую базовыми функциональными возможностями. И вновь он уведомляет о найденных ошибках менеджера проекта для того, чтобы тот поручил ответственному за эти ошибки инженеру по компонентам их исправление.

Оценка результатов теста. По мере тестирования целостности и системного тестирования инженеры по тестированию в конце каждого билда просматривают результаты тестирования в свете задач, изначально определенных в плане тестирования (возможно, модифицированном на предыдущих итерациях). Цель оценки результатов тестирования в том, чтобы убедиться в выполнении задач тестирования. Если тестирование не достигло своей цели, следует модифицировать тестовые примеры и процедуры, чтобы задачи тестирования могли быть выполнены (см. подраздел «Деятельность: Оценка результатов тестирования» главы 11).

Контроль бизнес-плана

Единственная задача бизнес-предложения, разработанного в конце фазы проектирования, — быть руководством по выполнению фазы построения для менеджеров проекта и заинтересованных лиц. Завершая фазу, менеджер проекта сравнивает реальный прогресс в конце каждой итерации с запланированным графиком, усилиями и затратами. Он просматривает данные по производительности, доле написанного кода, размеру баз данных и применяет другие способы оценки.

Число написанных строк кода редко бывает хорошим показателем прогресса в разработке, основанной на компонентах. Поскольку одна из наших задач — повторное использование компонентов, инженер по компонентам и другие сотрудники могут добиться хорошего прогресса в билдах и итерациях, даже написав мало нового кода. Более подходящим критерием завершенности работы в этом случае может быть завершенность билдов и итераций в соответствии с планом.

Расхождение более чем на несколько процентов, особенно в отрицательную сторону, требует от менеджера проекта проведения корректирующих акций. В частности, особенно значительные расхождения требуют проведения совещания с заинтересованными лицами.

По мере того, как менеджер проекта в ходе фазы построения лучше понимает затраты и возможности продукта, он может счесть необходимым изменение бизнес-плана и передачу нового варианта бизнес-плана заинтересованным лицам.

Оценка результатов итераций и фазы построения

На основании оценки результатов тестирования и других критериев выполнения, включая перечисленные в подразделе «Задание критерии оценки» данной главы, менеджер проекта и группа оценки:

- Сравнивают сделанное в ходе итерации с тем, что было запланировано.
- Планируют, на какой из следующих итераций должна быть выполнена не сделанная на данной итерации работа.
- Определяют готовность билда к началу новой итерации.
- Вносят изменения в список рисков.
- Детализируют план следующей итерации.
- Вносят изменения в планы итераций, которые идут за следующей.
- В конце последней итерации этой фазы определяют, прошел ли продукт системный тест и приобрел ли он базовые функциональные возможности.
- Санкционируют переход к фазе внедрения.
- Вносят изменения в план проекта.

Планирование фазы внедрения

Команда разработчиков не рассчитывает планировать фазу внедрения с такой же точностью, как предыдущие фазы. Члены команды знают, разумеется, что они должны раздать бета-версии (или их эквиваленты) для оценки выбранным пользователям. Эта часть фазы внедрения — выбор бета-тестеров, создание копий работающего кода, подготовка инструкций по тестированию и т. п. — планируется детально.

Сообщения, которые они получают в ответ, — риски, проблемы, дефекты, соображения — невозможно предусмотреть заранее. Если команда имеет некоторый опыт в бета-тестировании, она представляет, чего можно ожидать. В этом случае руководство сможет примерно оценить необходимое количество опытных сотрудников, необходимых для того, чтобы справиться с проблемами, обнаруженными при бета-тестировании.

Основные результаты

Результаты фазы построения:

- План проекта на фазу внедрения.
- Собственно программное обеспечение — выпуск, обладающий базовыми функциональными возможностями. Это заключительный билд фазы построения.
- Все артефакты, включая модели системы.
- Поддерживаемое и минимально изменяемое описание архитектуры.
- Предварительное руководство пользователя с полной детализацией для бета-тестеров.
- Бизнес-план, отражающий ситуацию на конец фазы.

Наша мечта о том, чтобы в графе «Результаты» стояли отметки «Выполнено», таким образом, исполнилась. На фазе внедрения при работе пользователей с программой может обнаружиться, что некоторые из этих отметок не соответствуют действительности. Такую ситуацию следует исправить.

16 Внедрение завершается выпуском продукта

Когда разрабатываемая система обладает начальной функциональностью, проект переходит на фазу внедрения.

Менеджер проекта полагает, что система вполне достойна работать в окружении пользователей, хотя еще и не отшлифована окончательно. В окружении пользователей могут проявиться некоторые проблемы, риски и дефекты, не обнаруженные в конце фазы построения. Могут обнаружиться требования, которые пользователи вспомнят в последний момент. Если они очень важны и составляют с проектом одно целое, менеджер проекта может согласиться добавить их. Однако изменения должны быть невелики, поскольку их придется вносить, избегая значительного вмешательства в план проекта. Если предлагаемые поправки требуют изменения графика работ, необходимость в них должна быть действительно острой. Мы полагаем, что в большинстве случаев лучше добавить их в список предложений и добавить к системе на следующем цикле разработки, то есть при разработке следующей версии системы.

Основные задачи этой фазы:

- Сравнить функциональность системы, разработанной на предыдущих фазах, с требованиями и выяснить степень удовлетворенности заинтересованных лиц.
- Рассмотреть все вопросы, необходимые для работы пользователей с системой, включая недостатки, сообщения о которых приходят от бета-тестеров и группы приемо-сдаточного тестирования.

Приемо-сдаточное тестирование — это задача заказчика, однако некоторые заказчики заключают контракт на тестирование со специализированными фирмами, занимающимися приемо-сдаточным тестированием.

Введение

В этой фазе внимание сосредоточено на том, чтобы способствовать утверждению продукта в сообществе пользователей. Способ, которым это делается, зависит от сущности отношений программы и ее рынка. Так, если программа выводится на

массовый рынок, команда разработчиков распространяет бета-версию среди типичных пользователей, найденных на специальных площадках, где «водятся» бета-тестеры. Если продукт предназначен для одиночного клиента или нескольких площадок в крупной организации, команда устанавливает продукт на одной из этих площадок.¹.

В ходе проекта осуществляется обратная связь с площадкой, на которой размещена программа, с целью:

- Определить, действительно ли система соответствует задачам бизнеса и пользователей.
- Изучить непредвиденные риски.
- Отметить нерешенные проблемы.
- Найти дефекты.
- Исправить двусмысленности и пробелы в пользовательской документации.
- Сосредоточить внимание на областях, в которых пользователи неопытны, и необходимой им информации или обучении.

На основе такого sorta сообщений пользователей команда модифицирует систему или отдельные ее артефакты. Команда готовится к тиражированию продукта, разбивке его по пакетам, развертыванию и резервному копированию, и восстановлению системы.

На этой фазе мы не вносим в продукт коренных изменений. Команда разработчиков и клиент должны вносить серьезные изменения в требования на более ранних фазах. Однако команда отыскивает небольшие недостатки, которые не были замечены на фазе построения и могут быть исправлены в рамках существующего базового уровня.

В обязанности команды по отношению к клиенту может также входить представление поддержки при создании подходящей для работы с проектом среды и обучение организации клиента правильному использованию продукта. Команда может оказывать пользователям помощь при параллельной работе с новой системой и с той унаследованной системой, которую она заменяет. Также она может способствовать конверсии баз данных в новую конфигурацию.

В случае, если продукт предназначен для широкого распространения (рынок коробочных продуктов), эти сервисы обычно встраиваются в программу установки, которая является частью продукта и поддерживается службой поддержки.

Фаза внедрения завершается выпуском продукта.

В начале фазы внедрения

Производство программного обеспечения сопровождается множеством деловых договоренностей. Если не вдаваться в детали, мы можем уложить эти договоренности в два шаблона:

¹ Мы можем рассмотреть две другие возможности: альфа-тестирование и проверка сторонней фирмой. Альфа-тестирование похоже на бета-тестирование, за исключением того, что оно проводится внутри компании, разрабатывающей программу, но вне группы разработки. Люди, проводящие альфа-тестирование, — это настоящие пользователи. В случае проверки сторонней фирмой тестированием занимается компания, специализирующаяся на проведении тестов по контракту с заказчиком.

Производство продавцом программ для массовой продажи потребителям на открытом рынке (коробочный продукт). Иногда этот рынок чрезвычайно широк, например рынок пользователей персональных компьютеров. Иногда узок, например при производстве компонентов многократного использования для программистов или программ-полуфабрикатов, которые индивидуально адаптируются под каждую установку. На этих рынках, несмотря на разницу в размерах, действует соотношение «один-ко-многим» (один продавец ко многим клиентам), которое и определяет действия на фазе внедрения.

Производство фирмой-разработчиком по контракту с одиночным заказчиком (заказной продукт).

Этот шаблон имеет несколько вариантов, а именно:

- Фирма-разработчик создает продукт для установки на одной площадке одиночного клиента.
- Фирма-разработчик создает продукт, который будет установлен в нескольких подразделениях клиента или на нескольких площадках.
- Аутсорсинговая фирма-разработчик, такая как EDS, Computer Science или Andersen Consulting. Иногда программное обеспечение разрабатывается для одной площадки или одного клиента, а затем может быть адаптировано и для других площадок или клиентов.
- Внутренняя группа разработки программного обеспечения, которая создает программное обеспечение для нужд других отделов той же компании.

Отношения между организацией и пользователем или заказчиком в фазе внедрения различны и зависят от шаблона. В соответствии с этими шаблонами фаза внедрения начинается после того, как выпуск, обладающий базовой функциональностью, пройдет системные тесты. Это будет означать выполнение условий, определенных для главной вехи в конце фазы построения. Однако команда разработчиков в фазе внедрения создает дополнительные артефакты, например сценарии установки, программы конвертирования или переноса данных, или модифицирует результаты фазы построения, подготавливая исполняемую программу к выходу в свет.

Планирование фазы внедрения

Едва ли можно рассчитывать на то, чтобы заранее спланировать фазу внедрения с такой же детализацией, как и фазу построения. С одной стороны, менеджер проекта знает, что на этой фазе происходит передача бета-версии (или ее аналога), разработанного на фазе построения, отдельным пользователям для тестирования. Это знание дает нам базу для предварительного планирования фазы внедрения. Объем работы по созданию бета-выпуска, подготовке документации к бета-тесту, выбору бета-тестеров и т. д. хорошо известен. С другой стороны, нам неизвестен заранее объем работы, которого потребуют сообщения, пришедшие от бета-тестеров. Менеджер проекта хочет посадить на работу с бета-тестерами несколько человек. Он может параллельно дать им работу в других проектах, но они должны быть постоянно готовы решать проблемы фазы внедрения.

Планируя фазу внедрения, менеджер проекта решает, что выпуск с начальной функциональностью после фазы построения потребует по результатам бета-тест-

стирования небольшой переделки. На самом деле, если проект строился на основе итераций, так оно и будет. Этот процесс разработки побуждает разработчиков к экспериментам на ранних итерациях, поиску своих концептуальных ошибок при помощи тестирования на ранних итерациях и наблюдению за этими операциями. Таким образом, простые ошибки обнаруживаются и исправляются билд за билдом по мере ведения работ. Короче говоря, ранние переделки — это *хорошо*. На фазе внедрения, на излете итеративной разработки, их останется менее 5%. Однако менеджер проекта должен считать, что их больше нуля. Как минимум несколько упущений наверняка проскочили все проверки. Тенденция недооценивать возможность внесения исправлений в проект приводит к эффектам:

- слишком плотного графика, приводящего, в свою очередь, к эффекту «поспешил — людей насмешил»;
- недостаточного системного тестирования и оценки в конце фазы построения;
- невнимательности и, в результате, пропуску в фазе внедрения важной работы;
- ощущения, что размыщление над необходимостью переделок приведет к неминуемой необходимости их внесения;
- предубеждения, что переделки — это плохо, что это признание в некомпетентности.

При планировании фазы внедрения может возникнуть разговор о том, что в программе «все в порядке». На самом деле ни один программный продукт не идеален. Так, например, продукты поставляются с некоторой долей пропущенных дефектов, выполнение некоторых требований откладывается до следующего выпуска, а на реализацию некоторых необходимых пользователям вещей, о существовании которых сообщили бета-тестеры, на фазе внедрения не хватает ресурсов. Вот три «ответа» на разговоры о «все в порядке».

Во-первых, фазы и итерации Унифицированного процесса направлены на определение рисков, правильных требований и соответствующее планирование проекта. Согласно Унифицированному процессу, команда разработчиков осуществляет эту деятельность в тесном контакте с пользователями и заказчиками. Следовательно, выпуск с базовой функциональностью, бета-выпуск, должен как можно ближе подойти к тому, что ожидают получить как разработчики, так и заинтересованные лица.

Во-вторых, поскольку ни разработчики, ни заинтересованные лица не ожидают, что выпуск с базовой функциональностью не будет содержать ошибок, они специально выделили время и ресурсы на фазу внедрения.

В-третьих, поскольку разработчики получили первые два «ответа» в кооперации с заинтересованными лицами, результатом может быть удлинение фазы или откладывание неожиданно свалившейся работы на следующий цикл разработки.

Персонал для осуществления фазы

Поскольку основная задача этой фазы — передать пользователям выпуск, сперва на бета-тестирование и приемку, а затем и на использование, требования к персоналу выше, чем во время фазы построения, хотя задачи могут не слишком отличаться.

Анализ сообщений от бета-тестеров и сотрудников, осуществляющих приемку, и реакция на них могут потребовать сотрудников, ориентированных скорее на поддержку, чем на разработку. Обсуждение даже небольших улучшений может потребовать специалистов не только в крупных частях системы, но и в природе приложения, в котором их предполагается делать. Кроме того, когда тестирование обнаруживает дефект, отслеживание причин его возникновения часто требует хорошего знания системы или как минимум тех ее частей, в которых находится причина сбоя. Вдобавок материалы для пользователей и учебные материалы, которые мы начали создавать на фазе построения, до поставки продукта рядовым пользователям, обычно требуют обработки техническим писателем. В ходе этой фазы архитектор не участвует в работе постоянно, а вызывается при необходимости, обычно для поддержания и обеспечения архитектурной целостности, но иногда и для обсуждения небольших изменений в архитектуре.

Задание критериев оценки

В конце фазы построения системные тесты показали наличие у продукта базовой функциональности, или, говоря другими словами, соответствие его заданным требованиям. В фазе внедрения необходимо решать лишь те проблемы, которые возникают на этой фазе. Существует пять основных типов подобных проблем:

1. Все ли ключевые функции просматриваются бета-тестерами, например, все ли варианты использования задействуются при работе с пакетом таким образом, как его используют бета-тестеры?
2. Руководит ли заказчик приемо-сдаточным тестированием продукта? Критерии тестирования должны быть записаны в контракте, который клиент подписывает с организацией-разработчиком. В добавок приемо-сдаточное тестирование продукта обычно предполагает запуск программного обеспечения в рабочем режиме в течение предусмотренного контрактом времени.
3. Достаточно ли хороши материалы для пользователей?
4. Есть ли необходимость в готовых учебных материалах (включая методические руководства для преподавателей)?
5. И наконец, считают ли пользователи и заказчики, что они удовлетворены этим продуктом? Мы прокомментируем эту проблему в подразделе «Когда заканчивается проект» данной главы.

Основные потоки работ на этой фазе не играют почти никакой роли

Как показано на рис. 16.1, на этой фазе активность по всем пяти основным рабочим процессам невелика. Поскольку основная работа была сделана на фазе построения, уровень активности на этой фазе невысокий, ровно такой, чтобы хватило на решение проблем, обнаруженных при тестировании в среде пользователей. Однако мы не хотим сказать, будто на этой фазе нам нечего делать (см. раздел «Что мы делаем на фазе внедрения» данной главы). Практически отсутствует работа по определению требований, анализу и проектированию. Проектирование на фазе

внедрения обычно уходит в тень, однако в некоторых случаях в него попадает кое-что большее, чем небольшие изменения в проекте или последние (обычно минимальные) улучшения. Внимание переключается на исправление дефектов, на устранение ошибок, возникающих при повседневном использовании, на проверку корректности самих исправлений и на регрессионное тестирование, которое мы проводим, чтобы убедиться, что исправления не внесли помех в работу системы.

Как мы говорили в предыдущих главах, стандартная итерация включает в себя пять основных рабочих процессов. В этой главе мы рассматриваем только ту их часть, которая имеет значение во время внедрения. Говоря коротко, мы занимаемся четырьмя видами деятельности, отчасти параллельно. Первый из них — это пять основных рабочих процессов, второй — планирование итераций в соответствии с описанием в разделах «Планирование предваряет деятельность», «Риски влияют на планирование проекта», «Расстановка приоритетов вариантов использования» и «Требуемые ресурсы» главы 12; третий — текущие согласования бизнес-плана, описанные в пункте 16.5, а четвертый — оценка, описание которой содержится в пунктах разделов «Завершение бизнес-плана», «Оценка итераций и фаз» главы 12 и «Оценка фазы внедрения» данной главы. В этом разделе мы приводим обзор фазы внедрения. На рисунке 16.1 показано, что основное внимание в этой фазе уделяется процессам реализации и тестирования. Во время них находятсяся, определяются, исправляются и подвергаются повторному тестированию ошибки, обнаруженные при бета-тестировании и приемо-сдаточном тестировании. В некоторых случаях для исправления дефектов может потребоваться частичное перепроектирование, что вызовет некоторую активность на процессе проектирования. Определение требований задействовано слабо. В следующем разделе мы подробнее опишем рабочие процессы этой фазы.

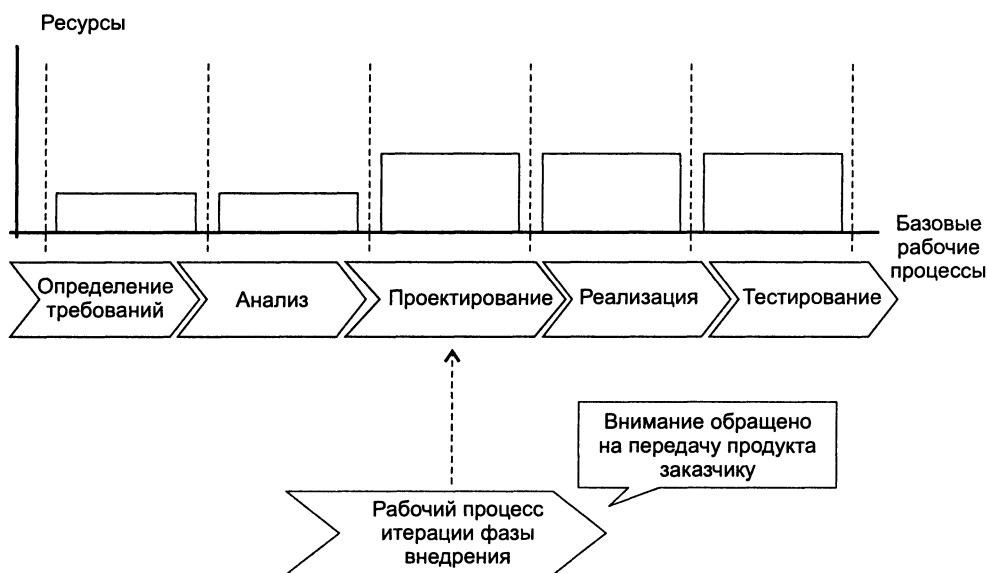


Рис. 16.1. На итерациях фазы внедрения осуществляются те же самые пять рабочих процессов, что и на предыдущей фазе (размеры прямоугольников приведены только для иллюстрации и могут изменяться в зависимости от проекта)

Что мы делаем на фазе внедрения

В ходе внедрения осуществляются следующие виды деятельности:

- Подготовка бета-версии (или версии для приемки), созданной на фазе построения.
- Установка (или подготовка к установке) этой версии на площадке тестирования плюс дополнительная работа на этой площадке, например перенос данных со старых систем.
- Работа с сообщениями бета-тестеров.
- Адаптация исправленного продукта под условия пользователей.
- Завершение артефактов проекта.
- Определение факта окончания проекта.

Эта последовательность действий может различаться в деталях в зависимости от того, создается ли данное программное обеспечение для рынка или для конкретного клиента. В первом случае у программы множество потенциальных пользователей, и выбор и руководство бета-тестерами становится серьезной задачей. Кроме того, как настоящие пользователи, они не следуют предписанным графикам тестирования. Они просто используют продукт так, как захотят, и пишут сообщения, если что-то найдут.

Во втором случае клиент, скорее всего, выбирает место для первоначальной установки, и приемо-сдаточное тестирование будет следовать формальной, заранее определенной систематической процедуре. Сообщаемые результаты в основном будут касаться отклонений от формальной спецификации. Если обнаружатся проблемы в области действия контрактных соглашений, стороны примут дополнительное соглашение.

Схема деятельности также сильно различна и зависит от того, является ли программный продукт «целинным» или развитием существующей или похожей системы. И снова у производителей программ есть возможность специализировать процесс в соответствии с ситуацией. Мы говорили об этом в главе 2.

Выпуск бета-версии

Большинство базовых групп пользователей для бета-тестирования (или приемо-сдаточного тестирования) будут обладать определенным опытом. Проводящая внедрение организация предполагает, что они будут использовать еще не полностью готовую документацию, а также снабжает их специальными инструкциями по написанию и отсылке сообщений о своих находках и замечаниях разработчикам. В начале фазы внедрения команда разработчиков передает бета-тестерам или тестерам приемо-сдаточного тестирования заранее подготовленную документацию. Они дополняют эту информацию инструкциями для бета-тестеров. Они выбирают бета-тестеров и рассылают им бета-версии и сопутствующие материалы к ним.

Установка бета-версии

Деятельность, происходящая на тестовых площадках, различается для бета-тестирования и приемо-сдаточного тестирования. В обычной ситуации имеется множество площадок бета-тестирования, ни на одной из которых не присутствует персо-

нал разработчика (внедренцы). На них могут быть специальные инструкции о том, как установить тестируемую систему, как с ней работать, на каких проблемах следовало бы сконцентрироваться бета-тестерам и как отправить отчет о найденных ошибках. Если новый выпуск улучшает или заменяет прежний продукт, внедренцы должны получить инструкции по переносу данных или конвертировании баз данных в формат нового выпуска. Инструкции могут предусматривать параллельную работу старой и новой программ в течение некоторого времени.

В противоположность этому при приемо-сдаточном тестировании, вероятно, будет присутствовать персонал разработчика. Существует формальный документ о тестировании при сдаче, но он дополняется неформальным общением. Дефекты и проблемы по возможности устраняются на месте или, при необходимости, передаются разработчикам соответствующей квалификации.

Реакция на результаты тестирования

Команда собирает и анализирует результаты тестирования, определяя действия, которые следует предпринять. Результаты можно разделить на два класса — относительно небольшие ошибки кодирования, которые просто нужно обнаружить и устраниить (хотя в некоторых случаях это и непросто), и более серьезные проблемы, которые могут иметь множественные проявления и исправление которых может потребовать значительных усилий, вплоть до необходимости изменить архитектуру.

Ошибки. Ошибки появляются в первую очередь в результате проявления дефектов компонентов, однако эти дефекты могут быть прослежены до места их возникновения в результате неточностей моделей проекта, анализа или более ранних. Если подобное отслеживание затруднено, группа, занимающаяся внедрением, может попытаться привлечь инженера по компонентам или других сотрудников, работавших с той частью программы, в которой обнаружена ошибка. Во всяком случае, компетентный сотрудник устраниет ошибку, а тестеры проводят регрессионное тестирование. Кроме того, команда рассматривает вопросы:

- Не может ли этот дефект быть связан с другими, еще не найденными?
- Может ли он быть исправлен без изменения архитектуры или проекта?
- Может ли он быть исправлен без внесения новых дефектов?

Серьезные проблемы. Некоторые из трудностей, обнаруженных при бета-тестировании, могут потребовать больших усилий, чем «исправление ошибок». Так, они могут потребовать проведения полной дополнительной итерации вместе с тестированием. Они могут требовать изменений, улучшений или добавки новой функциональности. Этими дополнительными требованиями следует формально управлять, например, через механизм Группы Управления Изменениями. В этой связи мы подчеркнем важность поддержки и управления конфигурациями. Как мы уже говорили, существенные изменения, которые потребуют затраты ресурсов, смещения сроков или изменений в архитектуре, лучше по возможности откладывать на следующий цикл разработки.

При решении проблем и устранении дефектов важно поддерживать архитектурную целостность системы. Для этого архитектор наблюдает за развитием дел с внедрением. Он работает «по вызову» для того, чтобы убедиться, что проблемы

и дефекты разрешены в соответствии с общим архитектурным решением. Он (или группа архитекторов) убеждаются, что проблему не решали для скорости таким способом, что вся архитектура развалилась. Достигение этой цели требует тонкой настройки архитектуры. Если деятельность на этой фазе может повлиять на архитектуру, архитектор изменяет ее описание.

Разумеется, немногие фирмы выпускают идеальное программное обеспечение. Учитывая это, менеджер проекта должен взвесить, как затраты и задержка выпуска в случае некоторых переделок согласуются с бизнес-планом.

Адаптация продукта к различным операционным средам

Как мы говорили ранее, производители программного обеспечения создают продукты двух разных по отношению к рынку типов: рыночные продукты (отношение с заказчиками — «один ко многим») и продукты для одного клиента (отношение с заказчиками — «один к одному»). При любом из этих отношений могут возникать задачи переноса данных или конвертирования базы данных старой системы в новый формат.

Отношения рыночного производства. Рынок может выдвигать очень разнообразный набор условий, для которых команда разработчиков должна создать различные версии исполняемых файлов. Так, например, меняться могут страна, язык, валюта и другие единицы измерения. Если продукт предназначен для работы на различных узлах сети, может потребоваться создать отдельную версию для каждого типа узлов.

Пользователи первого общего выпуска могут иметь малый опыт бета-тестирования, в этом случае команда должна улучшить документацию, выдаваемую бета-тестерам, чтобы она удовлетворяла обычных пользователей.

Продукт, предназначенный для распространения на рынке, обычно устанавливается самими пользователями или системными администраторами компаний. В противном случае установку проводит местный поставщик программы. Команда создает процедуры для установщика и сценарии для службы поддержки. Эти процедуры могут быть достаточно сложны, например, в том случае, когда продукт устанавливается на группу персональных компьютеров или рабочих станций, соединенных во внутреннюю сеть. Эти процедуры еще более сложны в том случае, когда части продукта устанавливаются на различные узлы, а узлы определенным образом настраиваются. Различные типы узлов требуют различных процедур установки.

Отношения одного клиента. Внедрение системы у одного клиента согласно контракту происходит по нескольким шаблонам, которые мы перечисляли. Однако имеются некоторые различия:

- Представители клиента, вероятно, участвуют в ранних фазах, реагируя на прращения.
- Они наблюдают за финальными тестами на площадке разработчика.
- Они могут принимать участие в некоторых совещаниях по оценке работ, которые отмечают вехи построения.
- Производитель, вероятно, помогает установить систему на базовой площадке клиента. В случае больших сложных систем это может означать полную установку.

- Представители производителя наблюдают за приемо-сдаточным тестированием и могут вносить немедленные поправки, если это необходимо. В случае более сложных проблем они передают данные об этом в свою фирму, сотрудники которой помогают им во внесении изменений или поправок.
- Приемо-сдаточное тестирование заканчивается, когда клиент и производитель приходят к мнению, что система соответствует предварительно определенным требованиям. Разумеется, они могут определить дополнительные задачи или внести в задачи изменения, что повлечет за собой подписание дополнительного контракта.

Перенос или конверсия данных. Обычно при замене существующей системы возникает необходимость в процедурах для переноса данных или конверсии баз данных. Старые данные могут принадлежать продукту, который создавал тот же самый разработчик, и тогда их будет очень легко перенести в новый продукт. Однако если продукт был разработан другим поставщиком, возможно, конкурентом, перенести данные окажется гораздо сложнее. В число задач могут входить:

- Замена старой системы новой, как с полным перекладыванием существующих задач на новую систему, так и с параллельной работой обоих систем до тех пор, пока пользователь не убедится в правильной работе новой системы.
- Перенос данных между старой системой и новой, иногда с изменением формата данных.
- Вдобавок к предоставлению инструкций по этому переносу документация может содержать тесты для пользователей, позволяющие им про kontrolировать правильность установки.
- Команда может добавить дополнительную информацию для службы поддержки, особенно информацию, необходимую для поддержки пользователей при сбоях на этапе установки.

Завершение артефактов

Фаза внедрения не закрыта, пока не завершены все артефакты, включая модели и описание архитектуры. Мы подчеркиваем, в частности, что набор моделей должен быть полностью закончен на фазе построения, и внезапной необходимости дополнять его на фазе внедрения возникнуть не может. В конце концов, модели прошли все итерации и фазы в соответствии с описанием из подраздела «Развитие набора моделей» главы 12. В фазе внедрения они лишь корректируются при необходимости. До конца фазы внедрения мы подтверждаем полноту артефактов путем реального их использования.

Когда заканчивается проект

Фаза внедрения заканчивается не тогда, когда выполнена вся работа и созданы все артефакты, а тогда, когда пользователь удовлетворен. Но когда пользователя можно считать удовлетворенным? Это зависит от отношения с рынком.

В случае создания продукции для массового рынка менеджер проекта решает, что основная масса клиентов будет удовлетворена после того, как команда проработает отзывы бета-тестеров. В этот момент и выпускается основной выпуск.

В большинстве случаев, разумеется, среда и продукт продолжают развиваться, но производитель откликнется на это развитие новым выпуском или, в случае серьезных изменений, новым циклом разработки. Поскольку успешные продукты и заказные системы чаще вносятся небольшие изменения, чем большие, возникает чувство, что проект никогда не кончится. Фаза внедрения обрывается, когда обязанности по дальнейшей поддержке проекта передаются отделу технической поддержки или организации, предоставляющей подобные услуги.

В случае, если продукт создан по контракту с клиентом, удовлетворенность приходит к клиенту после того, как система пройдет приемо-сдаточное тестирование. Этот момент, разумеется, зависит от интерпретации требований, перечисленных в первоначальном контракте (подписанном при окончании фазы проектирования) и измененных добавлениями в ходе последующих фаз. Клиент может заключить с поставщиком контракт на техническую поддержку. С другой стороны, клиент может поддерживать программу своими силами или заключить подобный контракт с третьей стороной. Детали могут различаться, но смысл в том, что фаза внедрения окончена.

Завершение бизнес-плана

Затраты и график на фазу внедрения указывались в бизнес-предложении, составленном в конце фазы проектирования. В конце фазы внедрения становится доступной информация, по которой можно оценить успешность бизнес-плана.

Контроль прогресса

Менеджер проекта сравнивает реальный прогресс в фазе с графиком, усилиями и затратами, запланированными на эту фазу.

В конце фазы внедрения, который также по финансовым понятиям является и концом проекта, менеджер проекта собирает группу для определения истинного графика, затраченных человеко-часов, уровня ошибок и других величин, определяющих работу компании по сравнению с запланированными на этот проект величинами, чтобы:

- Посмотреть, выполнила ли команда поставленные перед ней задачи.
- Установить причины невыполнения (если это произошло).
- Добавить параметры проекта в базу данных компании для использования в будущем.

Пересмотр бизнес-плана

Этот план предсказывал, будет ли проект экономически успешным. И вот опять два шаблона: производство по контракту и производство на рынок. В первом случае успешность определить легко: превосходит ли цена по контракту затраты на проект? Разумеется, это относительно простое рассуждение может быть усложнено дополнительными соображениями, например, успешностью открытия компанией нового делового направления или тем, что часть подсистем или компо-

нентов, произведенных по контракту, помогут снизить затраты по будущим контрактам.

Во втором случае бизнес-план более сложен. Успешность определяется тем, были ли достигнуты такие параметры, как минимальная для компании прибыль на вложенный в разработку капитал. В конце фазы внедрения и проекта в целом известны не все данные, например, неизвестен уровень будущих продаж. Однако фирма к этому моменту уже знает, сколько она израсходовала и имеет более четкое понимание будущей перспективы, чем в начале проекта. Ответственные за это руководители обеспечивают сбор имеющихся данных и собирают группу для их оценки.

Оценка фазы внедрения

Менеджер проекта собирает небольшую группу для оценки фазы внедрения (см. также раздел «Оценка итераций и фаз» главы 12) и проведения разбора всего цикла разработки. Эта оценка отличается от процедуры оценки при окончании предыдущих фаз по двум причинам. Во-первых, это последняя фаза. Следующие фазы, на которые можно было бы свалить незаконченную работу, отсутствуют. Однако в системе определенного размера существует группа по развитию продукта, которой передаются все полезные идеи. Во вторых, хотя это и последняя фаза текущего цикла разработки, за ним будут и еще циклы. Группа по оценке должна записать все находки, которые можно будет использовать.

Эти находки можно разделить на две категории, рассматриваемые в следующих подразделах.

Оценка итерации и фазы внедрения

С одной стороны, если проектная организация успешно провела первые три фазы, то фаза внедрения должна проходить гладко и завершиться в соответствии с графиком и без выхода за рамки бюджета. Бета-тестирование обнаружит лишь простые ошибки, которые команда в состоянии исправить. Группа оценки найдет небольшое количество замечаний, ценных для разработчиков или для работы над проектом на следующем жизненном цикле.

С другой стороны, если разработчики не смогли определить все важные риски, не смогли разработать архитектуру, которая соответствовала бы требованиям, или правильно реализовать проект системы, то недостатки подобного типа болезненно скажутся на фазе внедрения. По причине этих недостатков менеджер проекта может принять решение увеличить фазу внедрения, чтобы получить хотя бы минимально годную в дело систему. Он может передать сообществу пользователей «вполне приличную» систему. Он может отложить некоторые функции, первоначально определенные в требованиях, на следующий выпуск.

Эти недостатки лягут в основу деятельности группы оценки. Группа оценки фактически оценивает весь проект целиком — все четыре фазы. Они записывают неудовлетворительный опыт разработки. С точки зрения процесса, цель оценки — сделать так, чтобы работа над следующим выпуском проекта была успешнее. В частности, оценка не должна быть основанием для репрессий среди сотрудников.

Подобные акции должны основываться на других доказательствах. Если группа оценки видит, что ее находки могут быть применены не по назначению, они должны стукнуть кулаком по столу. Полная фактическая картина важна для будущих успехов организации.

Результаты экспертизы законченного проекта

В отличие от оценки, экспертиза больше ориентирована на анализ того, что в ходе проекта было сделано правильно, а что — неправильно. Цель этой операции — получить сведения, которые помогут лучше организовать дальнейшие проекты и сделать процесс разработки более успешным, например.

- Должны быть записаны вопросы, относящиеся к разработанной системе, для отдела поддержки и команды разработки следующего выпуска. Например, часто записываются причины выбора используемой архитектуры; причины отказа от других возможных видов архитектуры также могут пригодиться следующей команде разработчиков.
- Должны быть обдуманы вопросы, относящиеся непосредственно к процессу разработки. Например:
 - Требуется ли сотрудникам общее обучение?
 - В каких областях необходимо специальное обучение?
 - Следует ли продолжать наставничество?
 - Принесет ли опыт в специализации Унифицированного процесса (см. главу 2) то понимание, которое поможет нам в будущих проектах?

Планирование следующего выпуска или версии

Опыт десятилетий производства программ показывает, что немногие программные продукты долго остаются неизменными. Аппаратное обеспечение и операционные системы, под которыми они работают, продолжают развиваться. Деловая или правительственная среда меняются. Все больше и больше деловых, промышленных, государственных и частных потребителей понимают силу программного обеспечения. Почти все программы немедленно по завершении одного цикла разработки входят в следующий. В новом цикле повторяются фазы анализа и определения требований, проектирования, построения и внедрения.

Основные результаты

Результаты этой фазы очень похожи на результаты фазы построения (см. главу 15), но подкорректированные и более полные:

- Собственно программное обеспечение, включая средства установки.
- Юридическая документация — контракты, лицензионные соглашения, отказы от претензий, гарантии.

- Полный корректный базовый уровень выпуска продукта, включая все модели системы.
- Полное и исправленное архитектурное описание продукта.
- Руководства для пользователей, операторов, системных администраторов и учебные материалы.
- Ссылки на службу поддержки пользователей и web-страницы, на которых можно найти дополнительную информацию, сообщить об ошибках и получить «заплатки» и обновления.

17 Заставим Универсальный процесс работать

Как объяснялось в предыдущих главах, разработка программного обеспечения для серьезных задач по-прежнему остается непростым делом. В этой последней главе мы еще раз вкратце рассмотрим все множество взаимосвязанных тем этой книги. Наша цель — обзорно рассмотреть материал предыдущих глав, выяснить, как связаны эти темы и как они работают вместе.

Кроме того, мы рассмотрим в этой главе две новых темы. Процесс разработки программного обеспечения сложен, и управление этим процессом тоже непростая работа. Это будет первой из новых тем. Второй темой будет переход из состояния «без процессов» или от некоего предшествующего процесса к Унифицированному процессу. Это тоже не самое легкое дело.

Универсальный процесс помогает справиться со сложностью

Начнем с того, что существует четыре фазы: анализ и планирование требований, проектирование, построение и внедрение. Поскольку серьезные программные системы продолжают развиваться, за пределами фаз главного цикла остаются будущие выпуски и, при серьезных изменениях, следующие поколения. Внутри этих фаз находятся связанные рабочие процессы, архитектура, управление рисками, итерации, приращения, например:

- Разработка программного обеспечения, управляемая вариантами использования посредством рабочих процессов определения требований, анализа, проектирования, реализации и тестирования.
- Разработка, ориентированная на архитектуру, — скелет элементов структуры и поведения, допускающий плавное развитие системы.
- Разработка с использованием строительных блоков и компонентов, с поддержкой активного повторного их использования.

- Разработка, при которой работа производится путем итераций и билдов внутри итераций, что приводит к последовательному росту системы.
- Разработка, управляемая рисками.
- Разработка, визуализируемая и записываемая при помощи Унифицированного языка моделирования (UML).
- Разработка, проверяемая на вехах.

Четыре вехи закрепляют процесс — цели жизненного цикла, архитектура жизненного цикла, базовая функциональность и выпуск продукта [6].

Цели жизненного цикла

На первой вехе проясняются цели жизненного цикла продукта. Это происходит путем ответов на вопросы типа:

- Можем ли мы четко определить область действия системы? Можем ли сказать, что находится в системе, а что вне ее?
- Подписали ли вы соглашение с заинтересованными лицами по ключевым требованиям к системе?
- Лежит ли на виду та архитектура, которая будет поддерживать такую функциональность?
- Определили ли вы риски, угрожающие успешному выполнению проекта? Известен ли вам способ их снижения?
- В состоянии ли будет созданный продукт приносить доход, оправдывающий его разработку?
- Способна ли ваша организация выполнить этот проект?
- Разделяют ли с вами эти цели заинтересованные лица?

Получение ответов на эти вопросы — это дело *фазы анализа и планирования требований*.

Архитектура жизненного цикла

На второй вехе проясняется архитектура жизненного цикла путем ответов на вопросы:

- Удалось ли вам создать исполняемый базовый уровень архитектуры? Гибок ли он? Надежен ли он? Может ли он развиваться в дальнейшем в течение всего срока жизни системы?
- Определили ли вы серьезные риски? Удалось ли вам снизить их настолько, чтобы получить уверенность, что они не смогут нарушить план проекта?
- Довели ли вы план разработки до состояния, необходимого для создания реалистического бизнес-предложения, включающего в себя график работ, затраты и качество?
- Сможет ли проект, созданный в соответствии с предложением, обеспечить соответствующую доходность инвестиций?
- Добились ли вы согласия заинтересованных лиц?

Получение ответов на эти вопросы — это дело *фазы проектирования*.

Базовые функциональные возможности

Третья веха определяет, имеет ли продукт базовые функциональные возможности. Достигли ли мы уровня функциональности продукта, достаточного для начала работы в среде пользователей, в частности, бета-тестирования?

Это ключевой вопрос третьей вехи. Когда мы начинаем приближаться к этой вехе, у нас имеется базовый уровень архитектуры, мы исследовали риски, создали план проекта и получили ресурсы. Теперь мы строим продукт. Если мы разобрались с целями жизненного цикла и его архитектурой, построение проходит гладко. Важная задача этой фазы — построение последовательности билдов и итераций. Правильная последовательность означает, что предпосылки для каждой новой итерации уже созданы. Правильная последовательность исключает необходимость возвратов и повторного осуществления предыдущих итераций для дальнейшего продвижения вперед.

Итак, несмотря на наши усилия по решению проблем, некоторые из них еще остались. Оставим эту работу — решим насущные проблемы. Дело фазы построения — создание системы.

Выпуск продукта

Четвертая веха определяет готовность продукта к неограниченному распространению среди сообщества пользователей. Может ли продукт успешно использоваться в типичном окружении пользователей?

После того как мы добьемся базовой функциональности, должна быть проделана следующая работа: бета-тестирование, приемо-сдаточное тестирование, исправление дефектов и проблем, обнаруженных при функционировании программы в рабочих условиях. Когда мы удовлетворим пользователей, мы выпустим продукт.

Эти задачи — дело фазы внедрения.

Основные темы

Темы, например требования, архитектура, основанная на компонентах разработки, Унифицированный язык моделирования, итерации и управление рисками, проходят четыре главных вехи и связывают их воедино. (Порядок, в котором мы перечисляли темы, не отражает уровня их важности. Все они важны. Все они работают вместе.)

Правильно определить требования. Правильное определение требований осуществляется за счет моделирования вариантов использования, анализа, наличия обратной связи и т. д. Наилучшее начало Унифицированного процесса — варианты использования.

Правильно определить архитектуру. Проект сколько-нибудь значительного размера должен ориентироваться на архитектуру. Архитектура позволяет разбивать систему на части и кооперировать их друг с другом. Архитектура упрочняет интерфейсы между частями, позволяя командам разработчиков работать независимо от того, что находится с другой стороны интерфейса, и поддерживает части системы в правильном состоянии. Ориентация на архитектуру не позволяет проекту выйти из колеи.

Использовать компоненты. Надежные интерфейсы в архитектуре (так же как стандартные интерфейсы в стандартных группах) — это один из тех элементов, которые делают возможной разработку, основанную на использовании компонентов. Многократно используемые строительные блоки снижают затраты на разработку, позволяют быстрее представить продукт на рынок и способствуют повышению качества.

Думать и общаться на UML. Разработка программного обеспечения — это нечто большее, чем написание кода. UML (вместе с другими возможностями Унифицированного процесса) превращает разработку программ в инженерную дисциплину. Это графический язык, на котором люди, имеющие отношение к программному обеспечению, могут думать, визуализировать данные, заниматься анализом, вести записи и общаться.

Без стандартного средства общения, такого как UML, люди будут испытывать большие трудности, пытаясь понять других людей или команды. Им будет нелегко передать информацию с фазы на фазу или от текущего выпуска к будущим выпускам и итерациям.

Делать итерации. Итерации и билды предоставляют разработчикам многочисленные преимущества: работа разбита на короткие отрезки, небольшие рабочие группы, привязка к управлению рисками, частые точки контроля и частая обратная связь.

Управлять рисками. Обнаружив риск — особо опасный, существенный или обычный, — помещаем его в список рисков и снижаем до того, как он сможет нарушить процесс разработки.

Руководство управляет переходом на Универсальный процесс

Перевод компаний или общественной организации со старых рельсов на новые лежит вне задач управления и контроля за текущими процессами. В настоящее время в деловом мире и правительствах существует множество организаций, слабо подкованных по части процесса производства программного обеспечения. Разумеется, они создают программы, поскольку иначе мы бы о них и не упоминали, но они делают это неорганизованно, не регулярным образом. Существует также немало других организаций, имеющих те или другие варианты процессов разработки программ, но возникает ощущение, что эти процессы их не удовлетворяют. Они также разрабатывают программное обеспечение, иногда вполне успешно, но чувствуют, что существует лучший путь. Да, он существует. Как этим компаниям начать переход на такой лучший путь?

Профессионал-одиночка на невысокой должности — не тот человек, который мог бы выполнить этот переход. Он может стать одним из поборников нового пути, когда организация решит вступить на него. Однако отвечать за это должно высшее руководство фирмы-разработчика. Это должно быть их инициативой, поскольку будет затронута вся организация. Они должны воспринять идею о том, что существует такая вещь, как лучший путь, и что эту вещь хорошо бы использовать. Они должны почувствовать, что конкуренция вызывает крайнюю необходимость что-

то срочно предпринять. Может быть, они воспримут зародыш этой идеи от знакомых руководителей другой фирмы, которая переходит к производству программного обеспечения.

Момент истины

Поскольку новый способ приходит в отдел разработки программного обеспечения и во всю компанию различными путями, руководитель фирмы должен искать поддержку у других руководителей. Он может сделать это посредством заявления о «моменте истины». В основе этого заявления лежат утверждения, что настал такой момент, когда путь, которым мы шли раньше, перестал нас устраивать. Он слишком дорог. Качество низкое, и, что, вероятно, важнее всего, время выпуска продукта на рынок чересчур велико. Даже если мы используем наше программное обеспечение внутри фирмы, мы не получаем того значительного выигрыша, который могло бы дать хорошее программное обеспечение, созданное в сжатые сроки. Прирост в пять процентов, который мы имели год от года, перестал нас устраивать. В этих трех областях — затратах, сроках и качестве — теперь возможен больший прирост, ведь у других компаний он уже есть.

Заявление о моменте истины должно содержать описание предполагаемых действий. Для фирм, разрабатывающих программное обеспечение, методы разработки, утилиты, строительные блоки и компоненты не являются основным профилем деятельности. Всем этим занимаются другие организации. Так, существуют организации, которые разрабатывают и продают компоненты. Существуют организации, подобные SAP, предлагающие полуфабрикаты систем, которые адаптируются под ваш бизнес. (Полуфабрикат системы — это большой каркас, в который заказчик с помощью поставщика вносит небольшие изменения для учета специфики его бизнеса.) Существуют организации, например Rational Software Corporation, которые поставляют Унифицированный процесс. Задача руководителя фирмы — выбрать ту из них, которая поможет ему в решении задач его фирмы.

Так получилось, что выбор невелик. Для большинства производителей программного обеспечения есть только один путь вперед. Это компоненто-ориентированная разработка. Как бы мы ни старались сделать все сами, например в банковской сфере, страховании, военно-промышленном комплексе, связи или в других отраслях, как бы мы ни искали полуфабрикаты, построение продуктов из строительных блоков, — это вариант, который руководитель, понимающий ситуацию, предпочтет. В большинстве случаев это будет более или менее внутренняя разработка. Кто-то купит строительные блоки, кто-то приспособит их под нужды компании. Это и будет процесс разработки. Во многих случаях руководители разработчиков обнаружат, что Унифицированный процесс будет решением этой части их проблем.

Приказ о реинжиниринге убеждает в необходимости перехода

Начиная эту работу руководитель заботливо, подробно и без недомолвок объясняет в приказе о реинжиниринге, почему компания переходит на улучшенный процесс разработки. В приказе должны быть учтены:

- Текущая деловая ситуация и тот факт, что она меняется.
- Чего сейчас ожидают от нас заказчики.
- Конкуренция нашей компании усиливается.
- Проблемы, с которым столкнулась компания.
- Диагноз. К чему могут привести эти проблемы.
- Риск того, что не изменившись, фирма рухнет.
- Что компания может сделать, особенно в отношении процесса разработки [50].

В [50] описывается содержание приказа о реинжиниринге, относящегося к многократному использованию. Психологические моменты, лежащие в основе этого приказа, также применимы и к процессу разработки.

Уверенность в поддержке. Менеджеры проектов должны чувствовать уверенность в длительной финансовой поддержке. Среди прочего, эта поддержка должна покрывать первоначальное обучение, помочь инструктора менеджерам проекта при создании первого приложения по новому процессу и продолжение обучения, если необходимо дальнейшее совершенствование. Не пытайтесь переходить на новый процесс, не имея под рукой знающих его людей. Инструкторы должны знать, как это делается. Они могут быть как внешними, так и внутренними. Успех первого проекта, который создается по новому процессу, зависит от объединения четырех видов поддержки: процессом, утилитами, обучением и инструкторами.

Продолжение существующих проектов. В компании солидного размера со множеством проектов текущие и многие из возобновляемых в ближайшем будущем проектов будут продолжать создаваться с использованием существующего процесса разработки. Не все будут обучены единовременно. Не всякий проект может допустить тот хаос, который сопровождает значительные изменения процесса.

В то же время приказ о реинжиниринге должен разъяснять, что менеджеры проектов и персонал, продолжающий или завершающий проекты, не останутся в стороне от прогресса. Они пройдут обучение в следующий раз и будут назначены на проект, создаваемый по Унифицированному процессу. Переход требует времени. Необходимость для компании продолжать текущий бизнес — это часть платы за то, что она сможет делать это и в будущем. Если объяснить это людям, они смогут наравне со всеми участвовать в обсуждениях о том, что будет дальше. Они не должны чувствовать себя аутсайдерами. Если такое произойдет, это сильно повредит переходу на новый процесс.

Уверенность в собственном будущем. Люди, вовлеченные в этот переход, — профессиональные разработчики. За время их трудовой деятельности работа сильно изменилась. По историческим меркам это очень короткий срок. Кто-то испытывает трудности с поддержкой. Руководители среднего звена, озабоченные тем, как они справляются со своими административными обязанностями, особенно хорошо ощущают эту ношу. Если они увидят разумные причины для уверенности в своем будущем, это будет способствовать переходу. Компании, которые записывают свою историю, будут иметь в этом преимущество, но все должны быть уверены в важности этого дела.

Осуществление внедрения

Руководитель фирмы-разработчика сталкивается с проблемами реализации.

Поборник. Первый пункт — это организация, в которой намечаются изменения. Поскольку руководитель организации обычно постоянно занят, для каждодневной борьбы за изменения, то есть для руководства переходом, ему нужен технически квалифицированный специалист. Этот поборник должен интересоваться Унифицированным процессом. Чтобы изучить его, он проходит обучение и советуется со знающими людьми.

Поборник — это обычно технически подкованный менеджер проекта. Он также может быть архитектором с данными менеджера. Он изучил Унифицированный процесс, он убежден, что этот процесс может помочь компании, и готов пойти за него на баррикады. В то же время он уверен как в руководителе, который его поддерживает, так и в сотрудниках, задействованных в проекте. Он умеет работать и с руководством, и с техническим персоналом. Он умеет убеждать людей. Он обычно интересуется методами и процессом, но в то же время достаточно зрелый человек, чтобы не пытаться начать все снова, по его уникальной методике. Он наилучшим образом подходит для того, чтобы адаптировать Унифицированный процесс под особенности первого проекта.

Поборник может не иметь опыта в применении Унифицированного процесса. Фактически ему и негде было взять этот опыт, поскольку он некоторое время работал в данной организации. Он просто понял, что нужно делать. И он охотно готов сидеть и делать это — при условии помощи от менеджера проекта и инструктора, о которых мы скажем ниже.

Менеджер первого проекта. Кроме поборника (если вам не удастся найти такого исключительного человека, как мы описали, выберите лучшего из тех, что у вас есть, и предоставьте ему полную поддержку), ответственный руководитель также нуждается в исключительно способном менеджере проекта. Он нуждается в таком человеке, который спинным мозгом чувствовал бы важность как адаптации нового процесса, так и работы по нему. Проект должен выстоять перед лицом трудностей, которые могут встретиться при адаптации нового процесса, и успешно завершиться.

Инструктор. Обучение — это еще не все. В частности, это не опыт работы по новому. Поэтому поборник и менеджер проекта нуждаются в поддержке консультанта (внутреннего или внешнего), имеющего опыт проектов, сделанных по Унифицированному процессу. Инструктор не нуждается в опыте руководящей работы, хотя иметь такой опыт ему не повредит. У него должны быть два специальных таланта. Во-первых, он должен предчувствовать проблемы проекта. Эта способность, разумеется, основана на его предыдущем опыте адаптации Унифицированного процесса для реальных проектов. Во-вторых, он должен быть в состоянии уживаться с различными сотрудниками, с которыми ему придется общаться, как с поборником, так и с менеджером проекта, сотрудниками, занятыми в проекте, и поддерживающим переход руководителем.

Когда начать. Первый проект — реален, и успех в нем идет в счет. Наш опыт в пробных учебных проектах не впечатляет. Это все были упражнения на мелком

месте. Их результаты, если таковые были, тоже не производят сильного впечатления. Этот проект будет решать важную задачу, но график не должен быть чересчур плотным. Ну да, скажете вы, все важные проекты всегда должны были быть сделаны вчера.

Мы знаем. На самом деле работа с использованием Унифицированного процесса происходит быстрее, чем старым способом. Мы рано замечаем риски, мы рано создаем рабочую архитектуру, и построение действительно идет быстрее. Кроме того, первый проект создается под руководством инструкторов. Они являются инструкторами, поскольку прошли через это раньше. Они делают процесс работоспособным.

Рассчитывать реальный проект на дураков безопаснее, хотя невозможно понять, как сделать одновременно такое количество новых вещей. Новый процесс. Новые утилиты, которые помогают использовать процесс? Да. Они помогут использовать процесс, если хорошо интегрированы в него. Однако постарайтесь обойтись без новой операционной системы, новой технологии баз данных, нового языка программирования или новой платформы распределенных систем. Не складывайте в кучу так много новых понятий одновременно, особенно если технологии не полностью интегрированы. В зависимости от обстоятельств, вы можете включить в первый проект еще одну новую технологию.

Обсуждение. Наш опыт подсказывает:

- Описанный здесь подход — это не полное внедрение Унифицированного процесса, это систематический путь его внедрения.
- Более последовательный, пошаговый процесс попадает в обратную ловушку. Поскольку прогресс почти незаметен, поддержка исчезает и внедрение срывается.
- Иногда персонал в конце концов добивается успеха в последовательном, пошаговом реинжиниринге процесса разработки, но это приводит к множеству ошибок и занимает много времени. Мы не можем считать это успехом.
- В любом случае правильно будет проводить внедрение под активным управлением и контролем руководства. Ошибки, возникающие из-за утраты контроля, трудно исправлять.

Специализация Унифицированного процесса

Унифицированный процесс, как показано в этой книге, это не только слова в заголовке. И в самом деле, есть еще две важные вещи, которые можно сказать про него. Во-первых, это каркас. Он может быть привязан к различным переменным — размеру разрабатываемой системы, предметной области системы, ее сложности, опыта, способностям и уровню организации, выполняющей проекты, и ее сотрудников. Во-вторых, эта книга, как может показаться, достаточно детальная, является лишь обзором реального процесса. Чтобы применить его в деле, вам необходимо

мо значительно больше информации. Мы обсудим эти две особенности Унифицированного процесса в следующих подразделах.

Привязка процесса

Программные системы, продукты и организации, которые их создают, по-прежнему разнообразны. А значит, в Унифицированном процессе должны быть как постоянная часть, например, те же четыре фазы и пять основных рабочих процессов, так и множество переменных факторов. Скажем, какими должны быть относительные продолжительности фаз? Сколько итераций следует отвести на каждую из фаз в различных условиях? В какой момент вариант архитектуры (или снижение особо опасных рисков, базовый уровень архитектуры, бизнес-план и т. п.) можно считать успешно законченным?

Ответы на вопросы такого типа зависят от размера системы, природы приложения, степени опытности организации в предметной области, сложности системы, опыта команды разработчиков, качества управления и даже от способности заинтересованных лиц к эффективной работе в группе.

Приведем пример. Если система относительно мала, а команда разработчиков имеет опыт работы в данной предметной области (полученный ранее при работе над сходными продуктами) — фаза анализа и планирования требований может быть весьма коротка. Команда (и, возможно, заинтересованные лица) знают, что на пути успешной разработки не стоит никаких особо опасных рисков. Им известно, что ранее использовавшуюся архитектуру можно использовать снова. В результате на подтверждение области действия системы и проверку, не появилось ли в этой области действия новых особо опасных рисков, будет затрачено несколько дней. Этого времени для выполнения фазы анализа и планирования требований будет достаточно.

Приведем другой пример. Если система велика, сложна и в новинку для команды разработчиков, мы можем ожидать, что фаза анализа и планирования требований, а вместе с ней и фаза проектирования, будут значительно длиннее и потребуют большего числа итераций.

Число вариантов значений этих параметров так же велико, как и число создаваемых систем. Чутье на них приходит с опытом. Поэтому менеджеру проекта следует посоветовать перенимать опыт других членов группы, равно как и знания заинтересованных лиц. Вам следует подогнать процесс под свои условия [59].

Заполнение каркаса процесса

Унифицированный процесс, описанный в этой книге, — это лишь обзор процесса, который необходим вам для управления командой согласованно работающих людей. Мы старались в этой книге дать техническим работникам и менеджерам понимание процесса, но книга не дает детального руководства по выполнению ежедневной работы. В ней перечислены некоторые создаваемые/используемые артефакты, но существует еще множество других. В ней не приводятся шаблоны документов. В ней не определяются все возможные типы сотрудников; их больше, чем перечислено. Время от времени в книге попадаются ссылки на то, что

утилиты могут сильно помочь при работе по процессу. Действительно, множество рутинной работы, входящей в реализацию процесса, может быть сделано при помощи утилит, причем более точно и быстро, чем голыми руками. Книга, однако, не приводит детальной информации по утилитам — какие утилиты для каких целей служат и как их использовать. По существу, книга содержит базовые идеи. Она описывает некоторые рабочие потоки, некоторые артефакты, некоторых сотрудников и некоторые виды их деятельности и применение Унифицированного языка моделирования.

Этого вполне достаточно. Более проработанной версией этого процесса является Rational Unified Process. Этот процесс содержит онлайновую базу знаний с возможностью поиска объемом в 1800 страниц. Он повышает продуктивность команды, предоставляя каждому члену команды разработчиков руководства по повышению продуктивности, шаблоны и помочь утилитами в особо важной деятельности. Этот процесс встроен в утилиты Rational. Фактически, утилиты и процесс разрабатывались совместно. Его наполнение постоянно изменяется. Он поддерживается обширным набором курсов. Это та база, при наличии которой будет эффективным персональное руководство инструктора.

Rational Unified Process — это дальнейшее развитие старого Rational Objectory Process. Если вы работали с ним раньше, переход к новому Унифицированному процессу для вас будет несложен.

Универсальный процесс для широкого круга лиц

Кроме предоставления сотрудникам более эффективной рабочей среды и более эффективных взаимоотношений с пользователями и заинтересованными лицами, Унифицированный процесс предоставляет широкому кругу лиц весьма полезный интерфейс:

- *Обучение.* Преподаватели и инструкторы в своих курсах могут сконцентрировать внимание на том, что нужно студентам для работы по Унифицированному процессу и для понимания и визуализации информации при помощи Унифицированного языка моделирования.
- *Разработчики программного обеспечения.* Архитекторы, разработчики и другие категории сотрудников могут переходить от проекта к проекту и из компании в компанию, и им не потребуется долгое время приспосабливаться к уникальным методикам новой компании.
- *Многократное использование.* В проектах можно повторно использовать подсистемы и компоненты, поскольку имеется их представление на Унифицированном языке моделирования.
- *Утилиты.* Проекты будут поддерживаться более эффективными утилитами, поскольку широкий рынок утилит для Унифицированного процесса предоставляет хорошую финансовую поддержку для разработки таких утилит.

Определение пользы от использования Универсального процесса

Раннее определение требований путем разработки вариантов использования в ходе совместных усилий действительных пользователей и людей, проектирующих процессы, переход от требований к последующим рабочим процессам (анализу, проектированию и т. д.) очень эффективны, потому что Унифицированный процесс управляемся вариантами использования (приложение В).

Определение задач проекта, руководство его выполнением, предоставление рекомендаций для будущих поколений продуктов, — все это в течение всего жизненного цикла архитектуры, понятной, гибкой, надежной и способной к длительному развитию, потому что Унифицированный процесс **ориентирован на архитектуру**.

Обучение разработчиков и заинтересованных лиц на опыте каждого успешного билда и итерации, потому что Унифицированный процесс является **итеративным** (приложение В) и **инкрементным**.

Минимизация возможности того, что риски различной степени тяжести подвергнут опасности как саму успешность проекта, так и его бюджет или график работ, потому что Унифицированный процесс **направляется рисками** (приложение В).

Увеличение скорости процесса разработки, уменьшение его стоимости и повышение качества продукта в результате применения строительных блоков многократного использования, потому что Унифицированный процесс **основан на компонентах**.

Возможность подключить каждого члена команды разработчиков или заинтересованное лицо к совместной работе, потому что Унифицированный процесс — это больше, чем руководство отдельного разработчика, это **спроектированный процесс**.

Наличие каркаса этого процесса из фаз, итераций, вех и проверок дает множество точек, в которых заинтересованные лица могут посмотреть на разработку и понять, как она проходит. Зная, что происходит, и имея данные из своего знания приложения, заинтересованные лица могут высказать свои соображения по улучшению системы. Особенно важно, что, поскольку на ранних фазах рассматриваются такие важные аспекты, как архитектура и риски, они могут внести свои идеи тогда, когда у разработчиков еще есть время для их успешной реализации.

Основанный на 30 годах практической работы, Унифицированный процесс вобрал в себя опыт нескольких лидеров и имеющих опыт организаций. Он унифицирован на множестве приложений и техник, а именно:

- **Визуализируемость.** Визуальные модели и артефакты, существующие в Унифицированном процессе, взяты из Унифицированного языка моделирования. Это очень выгодно для процесса, например, в связи с возможностью активно применять компоненты многократного использования и чертежи программ.
- **Утилитность.** Факт существования унифицированного процесса и его стандартного языка вызывает финансовую поддержку активного создания утилит, что, в свою очередь, делает процесс эффективнее.

- *Привязываемость.* Это каркас процесса, а не жесткий процесс. Его можно специализировать под различные прикладные области и организационные требования.
- *Расширяемость.* Унифицированный процесс не ограничивает тех, кто его использует, единственным вариантом осуществления некоторой деятельности. Пользователи могут работать и по другим методикам. Унифицированный процесс же указывает на то место в процессе, в котором должна осуществляться эта деятельность.

Унифицированный процесс дает организациям возможность работать по-разному. Самое главное, что он дает способ организации совместной работы команды разработчиков. Более того, он дает способ организации совместной работы команды разработчиков, пользователей и заинтересованных лиц.

A Обзор языка UML

Введение

Старые отрасли технологии давно уже считают полезным изображать конструкции при помощи рисунков. С самого начала развития программирования эти идеи были восприняты программистами и реализованы в различных типах рисунков, или, в более общем смысле, моделей. Сообщество разработчиков программ нуждается в средствах передачи этих моделей не только между членами команды разработчиков, но и вовне, заинтересованным лицам, и во времени, разработчикам следующих поколений программы. Для этого необходим язык, который не только связывал бы разработчиков между собой, но и предоставлял бы каждому индивидуальному разработчику среду для размышлений и анализа. Кроме того, модели не могут сохраняться в человеческой памяти месяцы и годы. Сотрудники хотят иметь возможность записывать их — на бумаге или в электронном виде. Унифицированный язык моделирования (UML) — это стандартный язык моделирования программного обеспечения, язык для визуализации, определения, конструирования и документирования артефактов программных систем. Первоначально UML позволял разработчикам визуализировать продукты их работы с помощью стандартизованных чертежей и диаграмм. Так, например, стандартный символы или пиктограммы, используемые при определении требований, — это эллипс для представления варианта использования и фигурка человека для представления пользователя, применяющего вариант использования. Главная пиктограмма проектирования — прямоугольник, изображающий класс. Эти пиктограммы — не просто графическая нотация, это синтаксис. В разделе «Графическая нотация» мы приводим обзор графической нотации UML. Расширенное ее описание можно найти в [James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1998], а руководство пользователя в [Grady Booch, Jim Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998]. Однако, кроме определения графической но-

тации, UML также задает и терминологию, то есть семантику. Мы приводим краткий обзор семантики с разъяснением основных понятий UML в разделе пункта «Глоссарий терминов». Более расширенное описание семантики можно найти опять-таки в [James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1998] и [Grady Booch, Jim Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998]. Другой источник информации по нотации и семантике UML — это комплект общедоступной документации [OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org], хотя он и более формален по природе. Дополнительные общие ссылки по UML — Martin Fowler, *UML Distilled*, Reading, MA: Addison-Wesley, 1997[5] Hans-Erik Eriksson, Magnus Penker, *UML Toolkit*, New York: John Wiley & Sons, 1998 [4] и [5].

Словарь

UML предоставляет разработчикам словарь, содержащий три категории: сущности, отношения и диаграммы. Мы просто упомянем их здесь, чтобы дать вам понятие об основных структурах языка.

Существует четыре типа сущностей: структурные, поведенческие, группировки и примечаний. Семь первичных вариантов структурных сущностей: варианты использования, классы, активные классы, интерфейсы, компоненты, кооперации и узлы. Два варианта поведенческих сущностей — взаимодействия и конечные автоматы. Четыре варианта группировок — пакеты, модели, подсистемы и каркасы. И всего один вариант сущности примечаний: примечание.

Внутри второй категории отношений мы обнаруживаем следующие типы: зависимости, ассоциации и обобщения.

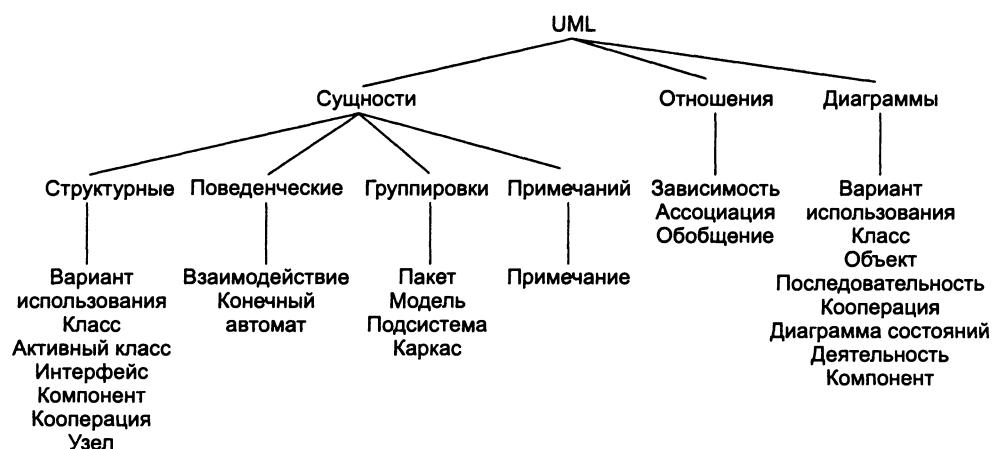


Рис. A.1. Словарь UML

И в третьей категории, диаграммах, UML предоставляет девять типов: вариантов использования, классов, объектов, последовательности, кооперации, состояний, деятельности, компонентов и развертывания.

Механизмы расширения

UML предоставляет *механизм расширения*, который применяется пользователем для уточнения необходимых ему синтаксиса и семантики. Так, UML можно при необходимости привязать к специфической системе, проекту или процессу разработки. Примеры привязки можно найти в приложении В.

Механизм расширения включает в себя стереотипы, именованные значения и ограничения. Стереотипы дают возможность определения новых элементов путем расширения и уточнения семантики уже существующих, таких как сущности и отношения (см. предыдущий подраздел). Именованные значения используются для придания новых свойств существующим элементам. И наконец, ограничения позволяют задать правила (например, правила целостности или бизнес-правила) для элементов и их свойств.

Графическая нотация

Следующие рисунки взяты из книги «*The Unified Modeling Language User Guide*» Гради Буча, Джеймса Рембо и Ивара Джекобсона [Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998].

Понятия, относящиеся к структуре

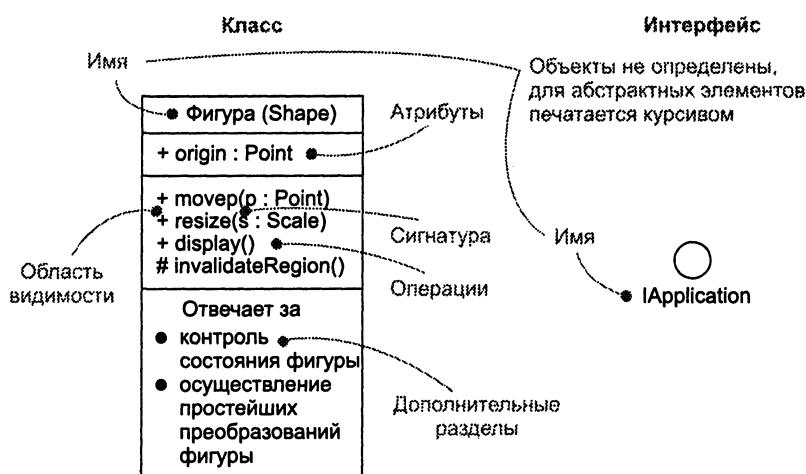


Рис. А.2. Классы и интерфейсы

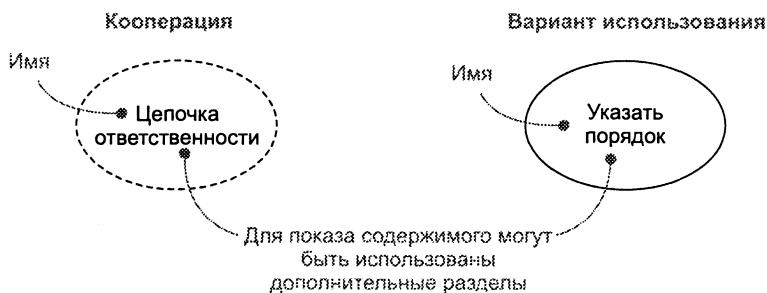


Рис. А.3. Варианты использования (отметим, что имена, например, вариантов использования можно помещать внутри значка, если имеется такая возможность)

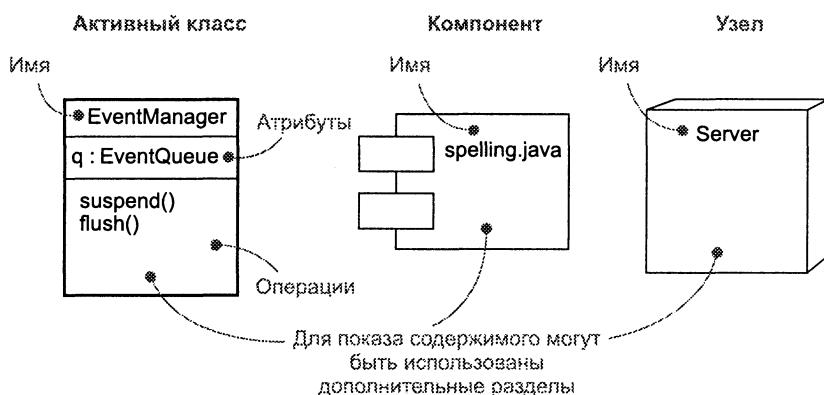


Рис. А.4. Активные классы, компоненты и узлы

Понятия, относящиеся к поведению

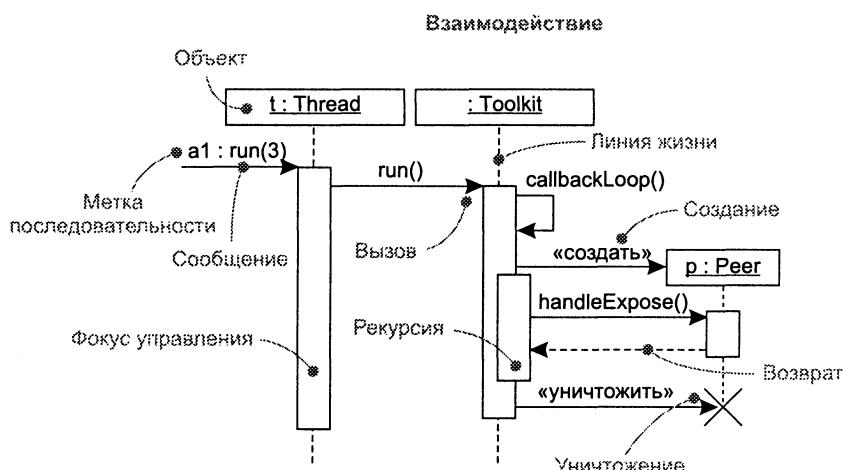


Рис. А.5. Взаимодействия

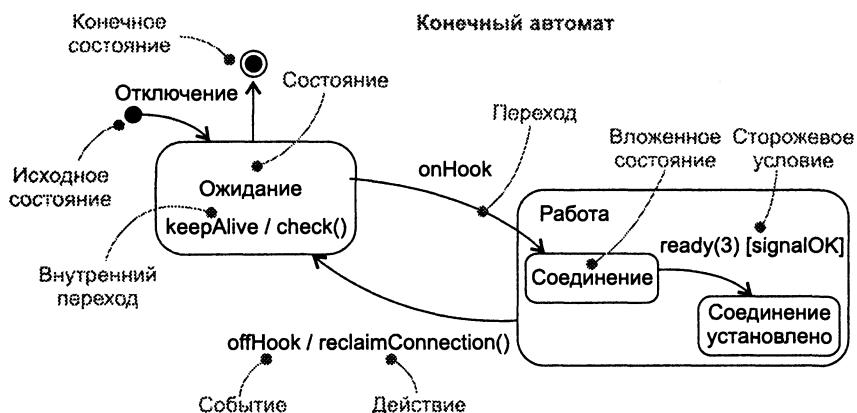


Рис. А.6. Конечные автоматы

Понятия, относящиеся к группировке

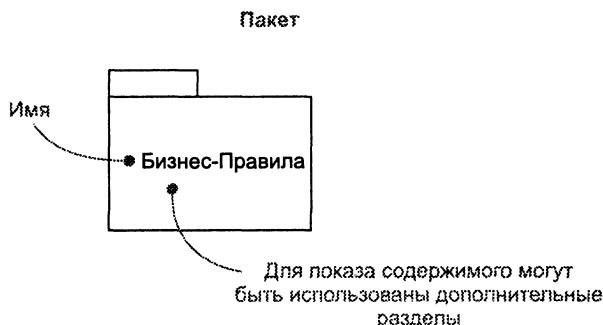


Рис. А.7. Пакеты

Понятия, относящиеся к примечаниям

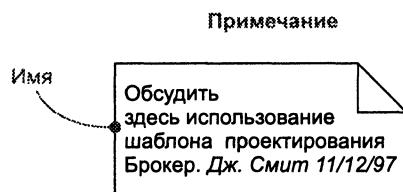


Рис. А.8. Примечания

Отношения зависимости

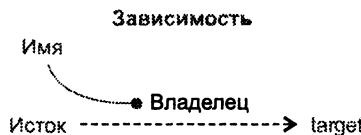


Рис. А.9. Отношения зависимости

Отношения ассоциации



Рис. А.10. Отношения ассоциации

Отношения обобщения

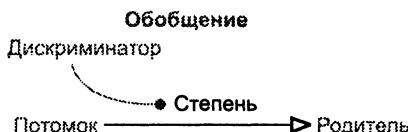


Рис. А.11. Отношения обобщения

Механизмы расширения



Рис. А.12. Механизмы расширения

Глоссарий терминов

Абстрактный класс (abstract class)	Класс, объект которого не может быть создан непосредственно
Действие (action)	Определение исполняемого выражения, которое создает абстракцию вычисляемой процедуры. Действие порождается при получении объектом сообщения или изменении значения его атрибута. В результате действия изменяется состояние объекта
Состояние действия (action state)	Состояние, которое представляет собой исполнение единичного действия, обычно вызов операции
Активация (activation)	Выполнение соответствующего действия
Активный класс (active class)	Класс, экземпляры которого являются активными объектами. См. <i>процесс, задача, поток</i>
Активный объект (active object)	Объект, являющийся владельцем процесса или потока, которые инициируют управляющую деятельность
Деятельность (activity)	Состояние, в котором проявляется некоторое поведение
Диаграмма деятельности (activity diagram)	Диаграмма, показывающая переходы от одного вида деятельности к другому. Диаграмма деятельности — это динамическое представление системы. Особый случай — диаграмма состояний, на которой все или большинство состояний привязаны к действиям и все или большинство переходов из состояния в состояние привязаны к выполнению действий в исходных состояниях
Актант (actor)	Связанный набор ролей, исполняемый пользователями варианта использования при взаимодействии с вариантами использования
Украшение (adornment)	Детализация спецификации элемента, добавляемая к его основной графической нотации
Агрегат (aggregate)	Класс, описывающий «целое» в отношении агрегации
Агрегация (aggregation)	Специальная форма ассоциации, определяющая отношение «часть-целое» между агрегатом (целое) и частями
Ассоциация (association)	Структурное отношение, описывающее набор связей, где под связью имеется в виду связь между объектами. Семантическое отношение между двумя или более классификаторами, включенными в связь между своими экземплярами
Класс ассоциации (association class)	Элемент моделирования, имеющий одновременно свойства класса и ассоциации. Класс ассоциации может рассматриваться как ассоциация, имеющая также свойства класса, или как класс, обладающий свойствами ассоциации
Полюс ассоциации (association end)	Конечная точка ассоциации, которая связывает ассоциацию с классификатором
Асинхронное действие (asynchronous action)	Запрос, отправляемый объекту без паузы для ожидания результата
Атрибут (attribute)	Именованное свойство классификатора, описывающее набор значений, которые могут иметь экземпляры свойства
Бинарная ассоциация (binary association)	Ассоциация между двумя классами
Связывание (binding)	Создание элемента из шаблона путем предоставления аргументов для параметров шаблона
Кардинальное число (cardinality)	Число элементов в наборе
Класс (class)	Описание набора объектов, имеющих одинаковые атрибуты, операции, отношения и семантику

Диаграмма классов (class diagram)	Диаграмма, показывающая набор классов, интерфейсов и коопераций и их отношения. Диаграмма классов — это статическое представление системы. Эта диаграмма показывает набор декларативных (статических) элементов
Классификатор (classifier)	Механизм описания структурных и поведенческих особенностей. Классификаторами являются интерфейсы, классы, типы данных, компоненты и узлы
Клиент (client)	Классификатор, запрашивающий сервисы у другого классификатора
Кооперация (collaboration)	Сообщество классов, интерфейсов и других элементов, работающих вместе с целью реализации некоторого кооперативного поведения. Кооперация больше, чем простая сумма элементов. Описание того, как элементы, такие как варианты использования или операции, реализуются в наборе классификаторов и ассоциаций, исполняющих определенным образом определенные роли
Диаграмма кооперации (collaboration diagram)	Диаграмма взаимодействий, которая отражает структурную организацию объекта, отправляющего и принимающего сообщения, диаграмма, которая демонстрирует организацию взаимодействия между экземплярами и их взаимосвязь
Примечание (comment)	Примечание, добавляемое к элементу или группе элементов
Компонент (component)	Физическая заменяемая часть системы, которая согласована с набором интерфейсов и предоставляет его реализацию
Диаграмма компонентов (component diagram)	Диаграмма, показывающая набор компонентов и их отношений. Диаграмма компонентов — это статическое представление компонентов системы
Композитная агрегация (composite)	Класс, связанный с одним или более классами отношением композиции
Композиция (composition)	Форма агрегации со строгим владением и совпадающим временем жизни частей целого. Части в неограниченном количестве могут создаваться после создания самой композиции, но после создания не существуют отдельно и перед уничтожением композиции должны быть удалены
Конкретный класс (concrete class)	Класс, для которого возможно создание экземпляров
Параллелизм (concurrency)	Осуществление двух или более видов деятельности в один и тот же временной интервал. Параллелизм может быть осуществлен путем квантования процессорного времени или одновременного выполнения двух или более потоков
Ограничение (constraint)	Расширение семантики элемента UML, позволяющее добавлять к нему новые правила или изменять существующие
Контейнер (container)	Объект, создаваемый для хранения других объектов и представляющий операции для доступа или перебора своего содержимого
Иерархия вложенности (containment hierarchy)	Иерархия пространств имен, содержащих элементы и отношения хранения между ними
Контекст (context)	Набор родственных элементов, предназначенных для определенной цели, например описания операции
Тип данных (datatype)	Тип, значения которого не тождественны. Типы данных включают в себя как простые встроенные типы (такие, как числа и строки), так и перечислимые типы (например, логический тип)
Делегирование (delegation)	Способность объекта посылать сообщение другому объекту в ответ на чужое сообщение
Зависимость (dependency)	Семантическое отношение между двумя сущностями, при котором изменение одной сущности (независимой сущности) влияет на семантику другой сущности (зависимой сущности)

Диаграмма развертывания (deployment diagram)	Диаграмма, показывающая набор узлов и их отношения. Диаграмма развертывания — это статическое представление развертывания системы
Диаграмма (diagram)	Графическое представление набора элементов, обычно в виде связного графа, в вершинах которого находятся сущности, а дуги представляют собой их отношения
Единица дистрибуции (distribution unit)	Набор объектов или компонентов, которые предназначены для выполнения одной задачи или работы на одном процессоре
Элемент (element)	Единичная составная часть модели
Событие (event)	Описание значительного происшествия, ограниченного во времени и пространстве, в контексте конечных автоматов. Событие — это побуждение, которое может запустить переход из состояния в состояние
Исполняемый модуль (executable)	Программа, которая может выполняться на узле
Экспорт (export)	В контексте пакетов — действие, делающее элемент видимым вне его собственного пространства имен
Механизмы расширения (extensibility mechanism)	Один из трех механизмов (стереотипы, именованные значения и ограничения), используемых для предписанного метода расширения UML
Фасад (facade)	Фасад — это стереотипный пакет, не содержащий ничего, кроме ссылок на элементы модели, находящиеся в другом пакете. Он используется для предоставления «публичного» представления части содержимого пакета
Запустить (fire)	Выполнить переход из состояния в состояние
Фокус управления (focus of control)	Символ на диаграмме последовательности, указывающий на период времени, в течение которого объект осуществляет деятельность как напрямую, так и через подчиненные операции
Каркас (framework)	Образец архитектуры, предоставляющий расширяемый шаблон приложения в какой-либо предметной области
Обобщение (generalization)	Отношение обобщения/специализации, когда объекты специализированных элементов (подтип) замещаются объектами обобщенных элементов (супертип)
Сторожевое условие (guard condition)	Условие, которое для запуска ассоциированного с ним перехода должно быть удовлетворено
Импорт (import)	В контексте пакетов — зависимость, демонстрирующая, на какие пакеты ссылаются классы данного пакета (включая пакеты, рекурсивно вложенные в данный)
Наследование (inheritance)	Механизм, при помощи которого более специфические элементы включают в себя структуру и поведение более общих элементов
Экземпляр (instance)	Конкретная реализация абстракции, сущность, к которой может быть применен набор операций, имеющая состояние для хранения эффекта операций. Синоним объекта
Взаимодействие (interaction)	Поведение, заключающееся в обмене группой сообщений между набором объектов в определенных случаях с целью выполнения некоторой задачи
Диаграмма взаимодействия (interaction diagram)	Диаграмма, показывающая взаимодействие, включающее в себя набор объектов и их отношений, в том числе и обмен сообщениями между ними. Диаграммы взаимодействия предоставляют динамическое представление системы. Это общий термин, применяемый к различным видам диаграмм, на которых изображено взаимодействие объектов, включая диаграммы кооперации, диаграммы последовательности и диаграммы деятельности

Интерфейс (interface)	Набор операций, используемых для описания сервиса класса или компонента
Наследование интерфейса (interface inheritance)	Наследование интерфейса более специализированным элементом, не включает наследование реализации
«Линия жизни» (lifeline)	См. <i>линия жизни объекта</i>
Связь (link)	Семантическая связь между объектами, экземпляр ассоциации
Полюс связи (link end)	Экземпляр полюса ассоциации
Местоположение (location)	Место размещения компонента на узле
Сообщение (message)	Специализация связи между объектами, которая передает информацию, ожидая в результате осуществления деятельности. Приход экземпляра сообщения обычно сопровождается возникновением экземпляра события
Метакласс (metaclass)	Класс, экземпляры которого являются классами
Метод (method)	Реализация операции
Модель (Model)	Семантически ограниченное абстрактное представление системы
Множественная классификация (multiple classification)	Семантическая вариация обобщения, в которой объект может принадлежать более чем одному классу
Множественное наследование (multiple inheritance)	Семантическая вариация обобщения, в которой тип может иметь более одного супертипа
Множественность (multiplicity)	Спецификация диапазона возможных кардинальных чисел набора
п-арная ассоциация (n-ary association)	Ассоциация между <i>n</i> классами. Если <i>n</i> равно двум, ассоциация бинарная. См. <i>бинарная ассоциация</i>
Имя (name)	То, как вы называете сущность, отношение или диаграмму; строка, используемая для идентификации элемента
Пространство имен (namespace)	Часть модели, в которой могут быть определены и использоваться имена. Внутри пространства имен каждое имя определяет единственную сущность
Узел (node)	Физический элемент, существующий во время работы системы и предоставляющий вычислительные ресурсы, обычно имеющий некоторый объем памяти, а часто — и возможность осуществления операций
Примечание (note)	Комментарий, добавляемый к элементу или набору элементов
Объект (object)	См. <i>экземпляр</i>
Объектный язык ограничений (OCL) (object constraint language (OCL))	Формальный язык, используемый для создания ограничений, не имеющих побочных эффектов
Диаграмма объектов (object diagram)	Диаграмма, показывающая набор объектов и их отношений в некоторый момент времени. Диаграмма объектов — это статическое представление проектирования или статическое представление процесса системы
Линия жизни объекта (object lifeline)	Линия на диаграмме последовательности, которая отражает существование объекта в течение некоторого периода времени
Операция (operation)	Реализация сервиса, который может запрашиваться любым объектом данного класса для реализации своего поведения
Пакет (package)	Механизм общего назначения для организации элементов в группы

Параметр (parameter)	Определение переменной, которая может быть изменена, передана или возвращена
Объект длительного хранения (persistent object)	Объект, сохраняющийся после завершения процесса или задачи, в ходе которой он был создан
Постусловие (postcondition)	Условие, которое должно выполняться после завершения операции
Предусловие (precondition)	Условие, которое должно выполняться перед запуском операции
Примитивный тип (primitive type)	Предопределенный базовый тип, например целое число или строка
Процесс (process)	Полновесный поток управления, который может выполняться параллельно с другими процессами
Свойство (property)	Именованное значение, содержащее характеристику элемента
Реализация (realization)	Семантическое отношение между классификаторами, когда один классификатор определяет контракт, который другие классификаторы должны гарантированно выполнять
Прием (receive)	Обработка экземпляров сообщений, поступивших от объекта-отправителя
Получатель (receiver)	Объект, обрабатывающий экземпляры сообщений, поступающие от объекта-отправителя
Отношение (relationship)	Семантическая связь между элементами
Ответственность (responsibility)	Контракт или обязательство типа или класса
Роль (role)	Специфическое поведение сущности в определенном контексте
Сценарий (scenario)	Специфическая последовательность действий, иллюстрирующая поведение
Область действия (scope)	Контекст, которому по некоторым причинам было дано имя
Отправление (send)	Посылка экземпляра сообщения от отправителя получателю
Отправитель (сообщения) (sender)	Объект, посылающий экземпляр сообщения объекту-получателю
Диаграмма последовательности (sequence diagram)	Диаграмма взаимодействия, показывающая временную последовательность обмена сообщениями
Сигнал (signal)	Спецификация асинхронного стимула, передаваемого от экземпляра к экземпляру
Сигнатура (signature)	Имя и параметры особенности поведения
Одиночное наследование (single inheritance)	Семантический вариант обобщения, при котором каждый тип может иметь только один супертип
Спецификация (specification)	Текстовая запись синтаксиса и семантики определенного строительного блока, описание того, что он из себя представляет
Состояние (state)	Условия или ситуация в ходе жизни объекта, когда он удовлетворяет некоторому условию, выполняет некоторую деятельность или ждет некоторого события
Диаграмма состояний (statechart diagram)	Диаграмма, показывающая конечный автомат Диаграмма состояний, — это динамическое представление системы
Конечный автомат (state machine)	Поведение, которое определяется последовательностью состояний, через которые проходит объект в течение времени жизни в ответ на пришедшие сообщения вместе с его реакцией на эти сообщения
Стереотип (stereotype)	Расширение словаря UML, позволяющее нам создавать новые типы строительных блоков, порождая их от существующих. Новые блоки специализированы для решения определенных проблем

Побуждение (stimulus)	Операция или сигнал
Подсистема (subsystem)	Группировка элементов, в которой каждый элемент содержит описание поведения, предоставляемого другим элементам подсистемы
Подтип (subtype)	В отношении обобщения — специализация другого типа, супертипа
Супертип (supertype)	В отношении обобщения — обобщение другого типа, подтипа
Элемент-поставщик (supplier)	Тип, класс или компонент, предоставляющие сервис, используемый другими
Плавательная дорожка (swim lane)	Область на диаграмме деятельности для назначения ответственного за действие
Синхронное действие (synchronous action)	Запрос, при котором отправивший его объект прерывает работу, ожидая результата
Система (system)	Набор подсистем, организованный в соответствии со специфической задачей и описанный набором моделей, возможно, с разных точек зрения
Именованное значение (tagged value)	Расширение свойств элементов UML, позволяющее помещать в описание элемента новую информацию
Задача (task)	Единичный путь выполнения программы, динамической модели или другого представления потока управления, нить или процесс
Шаблон (template)	Параметризованный элемент
Нить (thread)	Упрощенный поток управления, который может выполняться параллельно с другими нитями того же процесса
Отношение трассировки (trace)	Зависимость, указывающая на историческую связь или связь обработки между двумя элементами, представляющими одну и ту же идею, без определенных правил по порождению одного элемента из другого
Временный объект (transient object)	Объект, существующий только во время выполнения задачи или процесса, которые его создали
Переход (transition)	Отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, в случае некоторого события и выполнения определенных условий совершил некоторые действия и перейдет во второе состояние
Тип (type)	Стереотип класса, используемый для определения предметной области объекта и операций (но не методов), применимых к этому объекту
Использование (usage)	Зависимость, при которой один элемент (клиент) для корректного функционирования своей реализации нуждается в присутствии другого элемента (поставщика)
Вариант использования (use case)	Определение набора последовательностей действий, включая варианты, при которых система приноситциальному актанту полезный и понятный результат
Диаграмма вариантов использования (use-case diagram)	Диаграмма, показывающая набор вариантов использования и актантов и их отношений. Диаграмма вариантов использования является статическим представлением вариантов использования системы
Представление(view)	Проекция модели при рассмотрении в определенной перспективе или под определенным углом зрения, с исключением сущностей, не вошедших в эту перспективу
Видимость (visibility)	То, как имя может быть увидено и использовано другими

Б Расширения UML, специфичные для Универсального процесса

Введение

В этом приложении описаны расширения языка UML, необходимые для Унифицированного процесса. Эти расширения описаны в виде стереотипов и именованных значений, то есть в понятиях механизма расширения, предоставляемого UML, совместно с графической нотацией, используемой для отображения некоторых стереотипов. Стереотипы, которые не входят в стандартные расширения UML или отличаются от них [OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org [2]; James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1998 [1] и [2], помечены звездочкой (*).

Обзор UML можно найти в приложении А.

Стереотипы

Стереотип	Область применения	Краткое описание
Модель вариантов использования (use-case model)	Модель	Модель, содержащая актантов и варианты использования и их отношения. Модель описывает, что система делает для ее пользователей и какие ограничения на нее при этом налагаются
Система вариантов использования (use-case system)	Пакет верхнего уровня	Пакет верхнего уровня модели вариантов использования («система вариантов использования») — это подтип «пакета верхнего уровня»)

Стереотип	Область применения	Краткое описание
Модель анализа (analysis model)	Модель	Объектная модель, предназначенная для: (1) более точного определения требований; (2) структурирования их для улучшения понимания, подготовки, изменения, в общем, работы с ними; и (3) для использования ее в качестве исходных данных при проектировании и реализации системы — включая ее архитектуру
Система анализа (analysis system)	Пакет верхнего уровня	Пакет верхнего уровня аналитической модели («система анализа» — это подтип «пакета верхнего уровня»)
Управляющий класс (Control class)	Класс	Класс модели анализа, представляющий координацию, последовательность и управление другими объектами, часто используется для инкапсуляции управления для варианта использования
Класс сущности (entity class)	Класс	Класс модели анализа, используемый для моделирования долгоживущей, часто персистентной информации
Границный класс (boundary class)	Класс	Класс модели анализа, используемый для моделирования взаимодействия между системой и ее актантами, то есть пользователями и внешними системами
Анализ реализации варианта использования (use-case realization-analysis)*	Кооперация	Кооперация внутри модели анализа, описывающая, как реализован и выполняется некоторый вариант использования при помощи классов анализа (то есть управляющий класс, границный класс или класс сущности) и их взаимодействия с объектами анализа
Пакет анализа (analysis package)*	Пакет	Пакет, предоставляющий возможность организации артефактов модели анализа небольшими частями. Пакет анализа может содержать классы анализа (то есть управляющий класс, границный класс или класс сущности), анализ реализаций вариантов использования и другие пакеты анализа (рекурсивно)
Сервисный пакет (service package)*	Пакет	Вариант пакета анализа, используемый на нижнем уровне иерархии пакетов анализа (в модели анализа) для структурирования системы в соответствии с предоставляемыми ими сервисами
Модель проектирования (design model)	Модель	Объектная модель, описывающая физическую реализацию вариантов использования и сконцентрированная на том, как функциональные и нефункциональные требования вкупе с другими ограничениями, относящимися к среде разработки, реализуют рассматриваемую систему
Система проектирования (design system)	Подсистема верхнего уровня	Подсистема верхнего уровня модели проектирования («система проектирования» — это подтип «пакета верхнего уровня»)
Класс проектирования (design class)*	Класс	Класс проектирования представляет собой «непосредственную абстракцию» класса или подобной конструкции из реализации системы

Стереотип	Область применения	Краткое описание
Проект реализации варианта использования (use-case realization-design)*	Кооперация	Кооперация внутри модели проектирования, описывающая, как реализован и выполняется некоторый вариант использования при помощи классов и подсистем проектирования и их объектов
Подсистема проектирования (design subsystem)	Подсистема	Подсистема, предоставляющая возможность организации артефактов модели проектирования небольшими частями. Подсистема проектирования может содержать классы проектирования, проекты реализации вариантов использования, интерфейсы и другие подсистемы проектирования (рекурсивно)
Сервисная подсистема (service subsystem)*	Подсистема	Вариант подсистемы проектирования, используемый на нижнем уровне иерархии подсистем проектирования (в модели проектирования) для структурирования системы в соответствии с предоставляемыми именами сервисами
Модель развертывания (deployment model)*	Модель	Модель объекта, описывающая физическое распределение системы: как функциональность распределена по вычислительным узлам
Модель реализации (implementation model)	Модель	Модель, описывающая, как элементы модели проектирования, например классы проектирования, реализуются в виде компонентов, таких как файлы с исходным текстом программ и исполняемых файлов
Система реализации (implementation system)	Подсистема верхнего уровня	Подсистема верхнего уровня модели реализации («система реализации» — это подтип «пакета верхнего уровня»)
Подсистема реализации (implementation subsystem)	Подсистема	Подсистема, предоставляющая возможность организации артефактов модели реализации небольшими частями. Подсистема реализации может содержать компоненты, интерфейсы или другие подсистемы реализации (рекурсивно)
Модель тестирования (test model)*	Модель	Модель, описывающая, как исполняемые компоненты (например, билды) модели реализации тестируются на целостность и проходят системные тесты
Система тестирования (test system)*	Пакет верхнего уровня	Подсистема верхнего уровня модели тестирования («система тестирования» — это подтип «пакета верхнего уровня»)
Компонент тестирования (test component)*	Компонент	Компонент, автоматизирующий один или несколько процедур тестирования или их частей

Именованные значения

Именованное значение	Область применения	Краткое описание
Описание обследованной системы	Модель вариантов использования	Текстовое описание, предназначенное для объяснения модели вариантов использования целиком
Поток событий	Вариант использования	Текстовое описание последовательности действий варианта использования

Именованное значение	Область применения	Краткое описание
Специальные требования	Вариант использования	Текстовое описание, в которое входят все требования (нефункциональные требования, разумеется) варианта использования, которые не были привязаны к его потоку событий
Специальные требования	Класс анализа (то есть классы control, entity и boundary)	Текстовое описание, в которое входят нефункциональные требования к классу анализа. Это требования, определенные в ходе анализа, которые следует учитывать скорее при проектировании и реализации
Поток событий — анализ	Анализ реализации варианта использования	Текстовое описание, объясняющее и сопровождающее диаграммы (и их комментарии), определяющее реализацию варианта использования
Специальные требования	Анализ реализации варианта использования	Текстовое описание, в которое входят требования, а именно нефункциональные требования, к реализации варианта использования. Это требования, определенные в ходе анализа, которые следует учитывать скорее при проектировании и реализации
Требования к реализации	Класс проектирования	Текстовое описание, в которое входят требования, а именно нефункциональные требования, к классу проектирования. Это требования, определенные в ходе проектирования, которые следует учитывать скорее при реализации
Поток событий — проектирование	Проект реализации варианта использования	Текстовое описание, объясняющее и сопровождающее диаграммы (и их комментарии), определяющее реализацию варианта использования
Требования к реализации	Проект реализации варианта использования	Текстовое описание, в которое входят требования, а именно нефункциональные требования, к реализации варианта использования. Это требования, определенные в ходе проектирования, которые следует учитывать скорее при реализации

Графическая нотация

Большая часть стереотипов, перечисленных в разделе «Стереотипы» выше, не требует для отображения новых графических символов и может быть изображена путем прописывания понятия и того стереотипа, от которого оно было образовано, с помещением используемого стереотипа в кавычки (« и »).

Однако управляемому классу, граничному классу и классу сущности соответствуют собственные графические символы, приведенные на рисунке Б.1.

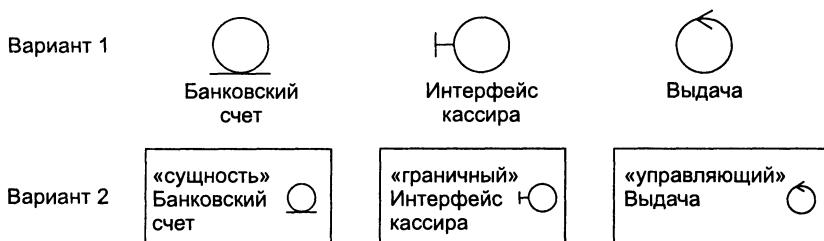


Рис. Б.1. Три стандартных стереотипа классов, используемые в анализе

B Основной глоссарий

Введение

В этом приложении собраны и определены основные понятия, используемые для описания Унифицированного универсального процесса, за исключением относящихся к языку UML или расширениям UML для Унифицированного процесса. Краткое определение этих понятий можно найти в приложении А, «Обзор языка UML» и приложении В, «Расширения UML для Унифицированного процесса».

Понятия

abstraction — абстракция

Важная характеристика сущности, выделяющая ее из всех прочих типов сущностей. Абстракция определяет границы поля зрения наблюдателя.

activity — деятельность

Существенная единица работы, осуществляемая сотрудником в ходе потока работ, которая (1) подразумевает четко определенную ответственность сотрудника, (2) дает четко определенный результат (набор артефактов), базирующийся на хорошо определенных исходных данных (другой набор артефактов), и (3) представляет собой единицу работы с жестко определенными границами, которые, вероятно, базируются на назначении задач исполнителям при планировании проекта. Также под деятельностью может подразумеваться выполнение сотрудником операции. См. *артефакт, сотрудник*.

analysis (workflow) — анализ (поток работ)

Один из основных потоков работ, первичная цель которого — анализ требований, полученных при определении требований для уточнения и структурирования. Цель этого действия состоит в том, чтобы (1) добиться более точного понимания требований и (2) получить такое описание требований, с которым было бы удобно работать и которое помогло бы структурировать систему целиком, включая ее архитектуру.

application system — прикладная система

Система, предоставляющая согласованный набор вариантов использования конечному пользователю.

application system suite — набор (комплект) прикладных программ

Набор различных прикладных систем, предназначенных для совместной работы, производящий результат, значимый для некоторых актантов. См. *Прикладная система*.

application-general layer — общий уровень приложений

Часть (пакетов или подсистем) системы, совместно используемая в пределах бизнеса или предметной области. Этот уровень используется специфическим уровнем приложений. См. *специфический уровень приложений*.

application-specific layer — специфический уровень приложений

Часть (пакетов или подсистем) системы, созданная специально для данного приложения и не используемая совместно с другими частями (подсистемами) системы. Этот уровень использует общий уровень приложений. См. *общий уровень приложений*.

architectural baseline — базовый уровень архитектуры

Базовый уровень, образующийся к концу фазы анализа и планирования требований, ориентированный на архитектуру системы. См. *уточнение, архитектура, базовый уровень*.

architectural pattern — образец архитектуры

Образец, определяющий некую структуру или поведение часто в архитектурном представлении соответствующей модели. Например, образцы *Уровень, Клиент-сервер, Трехуровневая система и Одноранговая система*, каждый из которых определяет соответствующую структуру развертывания и предлагает, как распределять компоненты (функциональность) по узлам. См. *образец, архитектурное представление*.

architectural prototype — прототип архитектуры

В первую очередь — исполняемый прототип, ориентированный на архитектурное представление модели реализации и компоненты, представляющие прототип. Если архитектурный прототип развивается, желательно сделать его базовое описание и объявить его более полно, чем на уровне прототипа (или наброска) описания архитектуры (включая все его архитектурные представления). См. *развивающийся прототип, базовый уровень, описание архитектуры, архитектурное представление*.

architectural style — стиль архитектуры

Системы, имеющие похожие структуры и ключевые механизмы высокого уровня, называются имеющими общий стиль архитектуры.

architectural view — архитектурное представление

Проекция структуры и поведения некоторой модели системы, ориентированная на существенные для архитектуры аспекты модели.

architectural view of the analysis model — архитектурное представление модели анализа

Представление архитектуры системы, выраженное в классах анализа, пакетах анализа и аналитических реализациях вариантов использования. Это представление предназначено в первую очередь для уточнения и структурирования требований к системе. Структура в этом представлении по возможности сохраняется до тех пор, пока не начнется проектирование и реализация архитектуры системы.

architectural view of the deployment model — архитектурное представление модели развертывания

Представление архитектуры системы, охватывающее узлы, из которых складывается физическая топология системы; представление, которое предназначено для распространения, поставки и установки составляющих систему частей.

architectural view of the design model — архитектурное представление модели проектирования

Представление архитектуры системы, охватывающее классы проектирования, подсистемы проектирования, интерфейсы проектирования и проектные реализации вариантов использования, которое формирует словарь системных решений. Также архитектурное представление модели проектирования охватывает задачи и процессы, формирующие механизмы параллелизма и синхронизации системы. Это представление ориентировано на представление нефункциональных требований, включая требования к производительности, масштабируемости и пропускной способности системы.

architectural view of the implementation model — архитектурное представление модели реализации

Представление архитектуры системы, охватывающее компоненты, используемые при сборке и выпуске физической системы. Это представление предназначено для управления конфигурацией версий системы, состоящей из частично независимых компонентов, которые могут быть по-разному скомпонованы для создания работающей системы.

architectural view of the use-case model — архитектурное представление модели вариантов использования

Представление архитектуры системы, охватывающее архитектурно значимые варианты использования.

architecture — архитектура

Набор решений по организации программной системы: выбору элементов структуры этой системы, их интерфейсов и поведения, задаваемого кооперациями между элементами, объединению элементов в прогрессивно растущие подсистемы, а также выбору архитектурного стиля, задающего организацию программной системы, — элементы, их интерфейсы, кооперацию и объединение. Архитектура программ имеет отношение не только к структуре и поведению, но также и к способам использования, функциональности, производительности, гибкости, возможностям повторного использования, постижимости, экономическим и технологическим ограничениям и компромиссам и вопросам эстетики.

architecture description — описание архитектуры

Описание архитектуры системы, включающее в себя архитектурные представления моделей. См. *архитектурное представление, архитектурное представление модели вариантов использования, архитектурное представление модели анализа, архитектурное представление модели проектирования, архитектурное представление модели развертывания, архитектурное представление модели реализации*.

architecture-centric — архитектуро-ориентированный

В контексте жизненного цикла программного обеспечения означает, что архитектура системы используется в качестве первичного артефакта для построения концепции, создания, управления и развития системы в ходе ее разработки.

artifact – артефакт

Существенная часть информации, которая (1) создается, изменяется и используется сотрудниками при осуществлении деятельности, (2) представляет область ответственности и (3) дает возможность управления своими версиями. Артефактом может быть модель, элемент модели или документ. См. *сотрудник, деятельность*.

baseline – базовый уровень

Набор проверенных и утвержденных артефактов, которые (1) представляют собой согласованный базис для дальнейшего развития и разработки и (2) могут быть изменены только путем формальной процедуры, такой, как управление конфигурацией и изменениями. См. *базовый уровень архитектуры, управление конфигурацией*.

build – билд

Исполняемая версия системы, обычно для некоторой части системы. Процесс разработки включает в себя последовательность билдов.

business process – бизнес-процесс

Общий набор видов деятельности, необходимых для получения ощутимого и измеримого результата, значимого для индивидуального клиента или бизнеса в целом.

cohesive – связующий

Способность сущности (системы, подсистемы или пакета) связывать свои части.

component-based development (CBD) – компонентно-ориентированная разработка

Создание и развертывание программных систем, собираемых из компонентов, а также разработка и поиск этих компонентов.

concurrency – параллельность

Означает, что несколько более или менее независимых работ (задач, процессов) одновременно используют одно аппаратное устройство (процессор).

configuration management – управление конфигурацией

Задача определения и поддержания конфигураций и версий артефактов. Включает в себя создание начальной версии, управление версиями, управление выпусками, управление состоянием и управление хранением артефактов. См. *артефакт, базовый уровень*.

construction phase – фаза построения

Третья фаза жизненного цикла программного обеспечения, когда программное обеспечение переходит из состояния исполняемого варианта базового уровня архитектуры в такое состояние, когда оно готово к переносу в сообщество пользователей.

contingency plan – план на случай неожиданностей

План, описывающий, что следует делать, если некий риск материализовался. См. *риск*.

core workflow – основные рабочие процессы

Одно из: требования, анализ, проектирование, реализация или тестирование. См. *рабочий процесс, требования, анализ, проектирование, реализация, тестирование*.

customer — заказчик

Человек, организация или группа людей, которые заказывают разработку системы, как с самого начала, так и последовательно уточняют задачу от версии к версии.

defect — дефект

Неправильность системы, например, ошибка в программе, обнаруженная при тестировании, или проблема, выявленная на совещании по проверке.

design (workflow) — проектирование (рабочий процесс)

Основной рабочий процесс, целью которого в первую очередь является формулирование модели, ориентированной на нефункциональные требования и предметную область. Он предназначен для подготовки к реализации и тестированию системы.

design mechanism — механизм проектирования

Множество классов проектирования, коопераций или даже подсистем модели проектирования, реализующих общие требования, такие как требования к согласованности, распространению и производительности.

developer — разработчик

Сотрудник, участвующий в основных рабочих процессах, например инженер по вариантам использования, инженер по компонентам и т. п. См. *основной рабочий процесс*.

distribution — распределенное выполнение

Происходит, когда несколько более или менее независимых работ (задач, процессов) распределены по различным аппаратным устройствам (процессорам).

domain area — предметная область

Область знаний или деятельности, характеризующаяся набором концепций и терминологией, которые понимаются людьми, работающими в этой области.

elaboration phase — фаза проектирования

Вторая фаза жизненного цикла программного обеспечения. В ходе этой фазы определяется архитектура программной системы.

engineering artifact — технический артефакт

Артефакт, созданный в ходе основных рабочих процессов. См. *основной рабочий процесс*.

evolutionary prototype — развивающийся прототип

Прототип, который развивается и уточняется, пока не станет частью разрабатываемой системы. Прототип, вероятно, будет субъектом управления конфигурацией. См. *управление конфигурацией*.

exploratory prototype — исследовательский прототип

Прототип, используемый только в исследовательских целях и отбрасываемый, когда эти цели достигнуты. Прототип, скорее всего, не будет субъектом управления конфигурацией. См. *управление конфигурацией*.

external release — внешний выпуск

Выпуск, предназначенный для клиентов и пользователей, являющихся внешними для проекта и его участников.

forward engineering — прямая разработка

В контексте разработки программного обеспечения — преобразование модели в код путем ее отображения на некоторый язык реализации. См. *обратная разработка*.

framework – каркас

Микроархитектура, предоставляющая незавершенные шаблоны для систем в некоторой предметной области. Может, например, быть подсистемой, созданной для последующего расширения и/или повторного использования.

functional requirement – функциональные требования

Требования, определяющие действия, которые система должна быть в состоянии совершать, без учета физических ограничений; требования, определяющие поведение системы как черного ящика.

green-field project – «целинный» проект

Беспрецедентный проект. См. *проект*.

implementation (workflow) – реализация (рабочий процесс)

Основной рабочий процесс, целью которого в первую очередь является реализация системы в понятиях компонентов, таких как исходные тексты программ, сценарии, бинарные файлы, исполняемые модули и т. п.

inception phase – фаза анализа и планирования требований

Первая фаза жизненного цикла программного обеспечения, в ходе которой ключевая идея разработки достигает той точки, когда правомочно начинать фазу детальной разработки.

increment – приращение

Малая и управляемая часть системы, обычно разность между двумя последовательными билдами. Каждая итерация дает в результате как минимум один (новый) билд, а значит, добавляет приращение к системе. Однако внутри итерации может быть создана последовательность билдов, каждый из которых добавляет к системе малое приращение. Таким образом, итерация добавит к системе большое приращение, возможно, накопленное за несколько билдов. См. *билд, итерация*.

incremental integration – инкрементная интеграция

В контексте жизненного цикла программного обеспечения – процесс, который включает в себя непрерывную интеграцию системной архитектуры для генерации выпусков, причем каждый новый выпуск включает в себя улучшение по сравнению с предыдущими.

integration – интеграция

См. *системная интеграция*.

internal release – внутренний выпуск

Выпуск, не демонстрируемый клиентам и пользователям, используемый только внутри проекта для нужд его участников.

iteration – итерация

Определенный набор действий, проводимый в соответствии с планом (итерации) и критериями оценки и приводящий к появлению выпуска, внешнего или внутреннего.

iteration plan – план итерации

Структурированный план итерации. План, устанавливающий ожидаемую стоимость итерации в понятиях времени и ресурсов и ожидаемый результат итерации в понятиях артефактов. План, определяющий, кто, что и в каком порядке должен делать в ходе итерации. Применяется для назначения людям обязанностей сотрудников и описания детализированного рабочего процесса итерации в ходе итерации. См. *итерация, артефакт, сотрудник, рабочий процесс итерации*.

iteration workflow — рабочий процесс итерации

Рабочий процесс, представляющий собой интеграцию основных рабочих процессов: определения требований, анализа, проектирования, разработки, реализации и тестирования. Описание итерации, включающее занятых в ней сотрудников, деятельность, которую они выполняют, и артефакты, которые они производят.

iterative — итеративный

В контексте жизненного цикла программного обеспечения — процесс, включающий в себя управление потоком исполняемых выпусков.

layer — уровень

Часть системы, определяемая пакетами или подсистемами. См. *общий уровень приложений, специфический уровень приложений, средний уровень, уровень системного программного обеспечения*.

legacy system — унаследованная система

Существующая система, унаследованная проектом. Обычно старая система, созданная с использованием более или менее устаревших технологий разработки, которую, несмотря на это, необходимо включить или повторно использовать — целиком или частично — в новой системе, создаваемой согласно проекту. См. *проектирование*.

life cycle — жизненный цикл

См. *жизненный цикл программного обеспечения*.

major milestone — главная веха

Контрольная точка, в которой руководство принимает важное деловое решение. Каждая фаза заканчивается главной контрольной точкой, в которой руководство принимает главное решение: идти вперед/оставаться на месте, а также решения по плану, бюджету и требованиям проекта. Мы можем рассматривать главные контрольные точки как точки синхронизации, в которых определенный набор целей достигнут, артефакты созданы, решение, переходить или нет к следующей фазе, принято, точки, в которых решения руководства пересекаются с действиями исполнителей. См. *фаза, проект, артефакт*.

management artifact — артефакт управления

Артефакт, не являющийся техническим артефактом, например план проекта, созданный менеджером проекта. См. *технический артефакт*.

mechanism — механизм

Общее решение общей проблемы или требования. Примеры — механизм проектирования, обеспечивающий возможности персистентности или распределенного выполнения в модели проектирования.

middleware layer — средний уровень

Уровень, предлагающий строительные блоки многократного использования (пакеты или подсистемы) для каркасов утилит и платформонезависимые сервисы для таких вещей, как распределенные объектные вычисления и интероперабельность в гетерогенных средах. Примерами могут быть брокеры объектных запросов, платформонезависимые каркасы для создания графического интерфейса пользователя, или продукты, реализующие обобщенный механизм проектирования. См. *уровень системного программного обеспечения, брокер объектных запросов, интерфейс пользователя, механизм проектирования*.

milestone — веха

См. *главная контрольная точка, вспомогательная контрольная точка*.

minor milestone — вспомогательная контрольная точка

Промежуточная контрольная точка между двумя главными контрольными точками. Может, например, ставиться в конце итерации или по завершении билда внутри итерации. См. *главная контрольная точка, итерация, билд*.

model-driven — управляемый моделью

В контексте жизненного цикла программного обеспечения означает, что разрабатываемая система организована в понятиях различных моделей с определенными целями, элементы которых связаны друг с другом.

nonfunctional requirement — нефункциональные требования

Требования, определяющие свойства системы, такие как ограничения реализации и окружения, производительность, зависимость от конкретной платформы, поддерживаемость, расширяемость и надежность. Требования, определяющие физические ограничения функциональных требований. См. *требования, требования к производительности, надежность, функциональные требования*.

nontechnical risk — нетехнический риск

Риск, относящийся к артефактам менеджмента и к таким аспектам, как доступные ресурсы (люди), их компетентность или даты поставок. См. *артефакт менеджмента, риск, технический риск*.

object request broker — брокер объектных запросов

Механизм прозрачного маршалинга и пересылки сообщений объектам, распределенным в гетерогенных средах. См. *распределенное выполнение*.

pattern — образец

Общее решение общей проблемы в данном контексте.

performance requirement — требования к производительности

Требования, которые налагают условия поведения на функциональные требования, например скорость, пропускная способность, время отклика и потребление памяти. См. *функциональные требования, требования*.

phase — фаза

Промежуток времени между двумя главными контрольными точками процесса разработки. См. *главная точка, анализ и планирование требований; проектирование, построение, внедрение*.

portability — переносимость

Степень легкости, с которой система, работающая в определенной среде исполнения, может быть изменена для получения системы, работающей в другой среде исполнения.

problem domain — проблемная область

Предметная область задачи, в которой определена проблема, — как правило, проблема, которую должна «решить» система. Проблемную область обычно понимает заказчик системы. См. *проблемная область, клиент*.

process — процесс

См. *бизнес-процесс, процесс разработки программного обеспечения, Универсальный процесс*.

project — проект

Усилие разработчика по созданию системы в течение жизненного цикла программного обеспечения. См. *жизненный цикл программного обеспечения*.

project plan — план проекта

План, обрисовывающий полную схему проекта, включая план-график, даты и критерии главных вех и разбиение фаз на итерации.

prototype — прототип

См. *прототип интерфейса пользователя, прототип архитектуры, развивающийся прототип, исследовательский прототип*.

regression test — регрессионное тестирование

Повторное тестирование (части) билда, которая уже тестировалась в предыдущем билде. Регрессионное тестирование используется в первую очередь для того, чтобы убедиться, что «старая функциональность» из «старого билда» продолжает работать и после добавления «новой функциональности» в «новом билде». См. *тестирование, билд*.

release — релиз

Относительно законченный и непротиворечивый набор артефактов — возможно, включающий билд, — передаваемый внутренним или внешним пользователям; поставка такого набора. См. *артефакт, билд*.

reliability — надежность

Способность системы демонстрировать правильное поведение в текущей среде выполнения. Может измеряться, например, в понятиях доступности системы, точности, среднего времени между отказами, ошибок на 1000 строк кода (KLOC) и ошибок на класс.

requirement — требование

Условие или возможность, которые система должна выполнять или поддерживать.

requirements (workflow) — требования (рабочий процесс)

Основной рабочий процесс, первичная цель которого — нацелить разработку на создание правильной системы. Это достигается грамотным описанием требований к системе, создаваемым в виде соглашения, которое может быть заключено между клиентом (включая пользователей) и разработчиками системы, о том, что система должна делать, а чего не должна. См. *требования, клиент, разработчик*.

reverse engineering — обратная разработка

В контексте разработки программного обеспечения, преобразование кода в модель путем отображения из некоторого языка реализации. См. *прямая разработка*.

risk — риск

Проектный фактор, который подвергает проект опасности. «Могут случаться риски» означает, что проект может столкнуться с нежелательной ситуацией, такой, как отставание от плана, перерасход средств или прекращение работ.

risk-driven — управляемый рисками

В контексте жизненного цикла программного обеспечения означает, что каждый новый релиз ориентирован на уменьшении наиболее существенных рисков для того, чтобы добиться успеха проекта.

robustness — устойчивость (робастность)

Стойкость некоторой сущности, обычно системы, к вносимым изменениям.

software development process — процесс разработки программного обеспечения

Бизнес-процесс (или бизнес-вариант использования) в бизнесе разработки программного обеспечения. Полный набор действий, необходимых для перера-

ботки требований клиента в согласованный набор артефактов, представляющих собой программное обеспечение, а позднее — для переработки изменений в этих требованиях в новые версии программного обеспечения. См. *бизнес-процесс*, *Универсальный процесс*.

software life cycle — жизненный цикл программного обеспечения

Цикл, содержащий 4 фазы в следующем порядке: анализ и планирование требований; проектирование, построение, внедрение. См. *анализ и планирование требований*; *проектирование, построение, внедрение*.

solution domain — предметная область решения

Предметная область, в которой определяются решения (проблем). Разработчики системы обычно понимают предметную область решений. См. *предметная область, разработчик*.

supplementary requirement — дополнительные требования

Обобщенные требования, которые не могут быть связаны с отдельным вариантом использования или отдельным классом реального мира, таким, как класс сущности предметной области или класс бизнес-сущности. См. *требования*.

system integration — системная интеграция

Компиляция и компоновка частей системных компонентов в один или более исполняемых файлов (они же компоненты).

system-software layer — уровень системного программного обеспечения

Уровень, содержащий программное обеспечение для инфраструктуры сетей и вычислений, такое, как операционные системы, системы управления базами данных, интерфейсы с конкретными аппаратными средствами и т. д. Это нижний уровень в иерархии уровней. См. *средний уровень*.

systemware — системное программное обеспечение

См. *уровень системного программного обеспечения*.

technical risk — технический риск

Риск, связанный с техническими артефактами и такими аспектами, как технологии реализации, архитектура или производительность. См. *технический артефакт, архитектура, требования к производительности, нетехнический риск*.

test (workflow) — тестирование (рабочий процесс)

Основной рабочий процесс, первичная цель которого — проверка результатов реализации путем тестирования каждого билда, включая внутренние и промежуточные, а также финальных версий системы, передаваемых внешним командам. См. *реализация, билд, внутренний релиз, внешний релиз*.

test case — тестовый пример

Спецификация одного варианта тестирования системы, включающая исходные данные для тестирования, результат и при каких условиях должно вестись тестирование.

test evaluation — оценка тестирования

Оценка результатов тестирования, например, покрытие вариантами тестирования, покрытие кода и статус дефектов. См. *тест, тестовый случай, дефект*.

test plan — план тестирования

План, описывающий стратегию, ресурсы и график тестирования.

test procedure — процедура тестирования

Спецификация, описывающая, как производить один или несколько вариантов тестирования или их части. См. *вариант тестирования*.

transition phase — фаза внедрения

Четвертая фаза жизненного цикла программного обеспечения, в ходе которой программное обеспечение выпускается из рук разработчиков в сообщество пользователей.

Unified Modeling Language — Унифицированный язык моделирования

Стандартный язык моделирования для программного обеспечения — язык для визуализации, описания, проектирования и документирования артефактов программных систем. Язык, используемый в унифицированном процессе. Язык, позволяющий разработчикам визуализировать их рабочие продукты (артефакты) в стандартизованных диаграммах. См. *артефакт, Универсальный процесс, разработчик*.

Unified Process — Унифицированный процесс

Процесс разработки программного обеспечения, основанный на Унифицированном языке моделирования, итеративный, архитектуро-ориентированный, управляемый вариантами использования и рисками. Процесс, организованный в четыре фазы, — анализа и планирования требований, проектирования, построения и внедрения, и в пять основных рабочих процессов — определение требований, анализ, проектирование, разработка и тестирование. Процесс, описанный в понятиях бизнес-модели, которая, в свою очередь, структурирована в понятиях трех базовых строительных блоков — сотрудников, деятельности и артефактов. См. *процесс разработки программного обеспечения, Унифицированный язык моделирования, итеративный, архитектуро-ориентированный, управляемый вариантами использования, управляемый рисками, фаза, анализ и планирование требований, проектирование, построение, внедрение, основные рабочие процессы, требования, анализ, проектирование, разработка, тестирование, сотрудник, деятельность и артефакт*.

use-case driven — управляемый вариантами использования

В контексте жизненного цикла программного обеспечения означает, что варианты использования применяются в качестве первичных артефактов для определения желаемого поведения системы и для передачи этого поведения владельцам системы. Также означает, что варианты использования являются первичными исходными данными для анализа, проектирования, реализации и тестирования системы, включая создание, проверку и обоснование архитектуры системы. См. *анализ, проектирование, реализация, тестирование, архитектура*.

use-case mass — масса вариантов использования

Полный набор видов деятельности всех вариантов использования в модели вариантов использования.

user — пользователь

Человек, взаимодействующий с системой.

user interface — интерфейс пользователя

Интерфейс, посредством которого пользователь взаимодействует с системой.

user-interface prototype — прототип интерфейса пользователя

В первую очередь исполняемый прототип интерфейса пользователя, однако на ранних стадиях разработки может содержать только бумажные наброски, электронные изображения и т. п.

view — представление

Проекция модели, рассматриваемой в соответствующей перспективе или с определенной точки зрения, опускающая сущности, не относящиеся к этой перспективе.

visual modeling – визуальное моделирование

Визуализация продуктов работы (артефактов) в стандартных диаграммах. См. *артефакт*.

waterfall approach – водопадная разработка

Подход к разработке программного обеспечения, при котором разработка выполняется как линейная последовательность операций, например в следующем порядке: определение требований, анализ, проектирование, разработка и тестирование. См. *требования, анализ, проектирование, разработка, тестирование*.

worker – сотрудник

Позиция, которая может быть поручена члену команды, требующая ответственности и способностей, например выполнение некоторой деятельности или разработка некоторого артефакта. См. *деятельность, артефакт*.

workflow – рабочий процесс

Реализация (части) бизнес-варианта использования. Может быть реализована в понятиях диаграммы деятельности, которые включают в себя участвующих сотрудников, деятельности, которые они выполняют и производимые ими артефакты. См. *основной рабочий процесс, рабочий процесс итерации*.

Литература

1. Ahlqvist Stefan and Jonsson Patrik, Techniques for systematic design of graphical user interfaces based on use cases, Proceedings OOPSLA'96.
2. Alexander Christopher, Sara Ishikawa, Silverstein Murray, with Jacobsen Max, Fiksdahl-King Ingrid, Angel Shlomo, A Pattern Language: Towns, Buildings, Construction, New York: Oxford University Press, 1977.
3. Archer James E. Jr., Devlin Michael T., Rational's Experience Using Ada for Very Large Systems. Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June, 1986.
4. Arnold Ken and Gosling James, The JavaTX1 Programming Language, Reading, MA Addison-Wesley, 1996.
5. Binder Robert V., Developing a test budget, Object Magazine, 7(4), June 1997.
6. Boehm Barry W., A spiral model of software development and enhancement, Computer, May 1988, pp. 61–72. (Reprinted in Boehm Barry W., Tutorial: Software Risk Management, IEEE Computer Society Press, Los Alamitos, CA, 1989.)
7. Boehm Barry, Anchoring the software process, IEEE Software, July 1996, pp. 73–82.
8. Booch Grady, Object-Oriented Analysis and Design with Applications, Redwood City, CA: Benjamin/Cummings, 1994.
9. Booch Grady, Object Solutions: Managing the Object-Oriented Project, Reading, MA: Addison-Wesley, 1996.
10. Booch Grady, Rumbaugh James, and Jacobson Ivar, The Unified Modeling Language User Guide. Reading, MA: Addison-Wesley, 1998.
11. Booch Grady, Rumbaugh James, and Jacobson Ivar, The Unified Modeling Language Reference Manual, Reading, MA: Addison-Wesley, 1998.
(Рамбо Дж., Якобсон А., Буч Г. UML: специальный справочник – СПб, Питер, 2001)
12. Buschmann F., Meurier R., Rohnert H., Sommerlad P., Stal M., A System of Patterns, New York: John Wiley and Sons, 1996.
13. Carrol John, Scenario-Based Design, New York: John Wiley & Sons, 1995.
14. CCITT, Specification and Description Language (SDL), Recommendation Z.100. Geneva, 1988.

15. Cockburn Alistair, Structuring use cases with goals, Report on Analysis & Design (ROAD), 1997.
16. Constantine L. L. and Lockwood L. A. D., Shareware for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. Reading, MA: Addison-Wesley, 1999.
17. Davis Alan M., Software Requirements: Objects, Functions, and States, Englewood Cliffs, NJ: Prentice Hall, 1993.
18. Drucker Peter R., Management: Tasks, Responsibilities, Practices, New York: Harper & Row, 1973.
19. Drucker Peter R., The Discipline of Innovation, Harvard Business Review, May–June, 1985; reprinted Nov. - Dec. 1998, pp. 149–157.
20. Ecklund E., Delcambre L., and Freiling M., Change cases: Use cases that identify future requirements, Proceedings, Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'96), ACM, 1996, pp. 342-358.
21. Eriksson Hans-Erik, Magnus Penker, UML Toolkit, New York: John Wiley & Sons, 1998
22. Fowler Martin, UML Distilled, Reading, MA: Addison-Wesley, 1997.
23. Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1994.
24. Garlan David and Shaw Mary, Software Architecture: Perspectives on an Emerging Discipline, Upper Saddle River, NJ: Prentice-Hall, 1996.
25. Harel David, Politi Michal, Modeling Reactive Systems With Statecharts: The STATEMATE Approach, New York: McGraw-Hill, 1998.
26. Hetzel Bill, The Complete Guide to Software Testing, Second Edition, Wellesley, MA: QED Information Sciences, Inc., 1988.
27. Humphrey Watts S., Managing the Software Process, Reading, MA: Addison-Wesley, 1989.
28. IEEE Std 610.12.1990.
29. ISO/IEC International Standard 10165-4 = ITU-T Recommendation X.722.
30. ITU-T Recommendation M.3010, Principles for a Telecommunication Management Network.
31. Jacobson Ivar, Concepts for Modeling Large Real Time Systems, Chapter 2, Dissertation, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.
32. Jacobson Ivar, Object-oriented development in an industrial environment, Proceedings of OOPSLA'87, Special issue of SIGPLAN Notices 22(12):183-191, December 1987.
33. Jacobson Ivar, Object-Orientation as a Competitive Advantage, American Programmer, Oct. 1992.

34. Jacobson Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, Object-Oriented Software Engineering: A Use-Case Driven Approach, Reading, MA: Addison-Wesley, 1992.
35. Jacobson Ivar, Basic use-case modelingl Report on Analysis and Design (ROAD), July-August, vol. I, no. 2, 1994.
36. Jacobson Ivar, Basic use-case modeling (continued), Report on Analysis and Design (ROAD), vol. 1, no. 3, September-October 1994.
37. Jacobson Ivar, Use cases and objects, ROAD 1 (4), November-December 1994.
38. Jacobson Ivar, Business process reengineering with object technology, Object Magazine, May 1994.
39. Jacobson Ivar, Ericsson Mafia, and Jacobson Agneta, The Object Advantage: Business Process Reengineering with Object Technology, Reading, MA: Addison-Wesley, 1994.
40. Jacobson Ivar, Modeling with use cases — Formalizing use-case modeling, Journal of Object-Oriented Programming, June 1995.
41. Jacobson Ivar and Magnus Christerson, Modeling with use cases — A growing consensus on use cases, Journal of Object-Oriented Programming, March-April 1995.
42. Jacobson Ivar and Sten Jacobson, Beyond methods and CASE: The software engineering process with its integral support environment, Object Magazine, January 1995.
43. Jacobson Ivar and Sten Jacobson, Designing a Software Engineering Process; Object Magazine, June 1995.
44. Jacobson Ivar and Sten Jacobson, Designing an integrated SEPSE, Object Magazine, September 1995.
45. Jacobson Ivar and Sten Jacobson, Building your own methodology by specializing a methodology framework, Object Magazine, November-December 1995.
46. Jacobson I., Palmqvist K., and Dyrhage S., Systems of interconnected systems, Report on Analysis and Design (ROAD), May-June 1995.
47. Jacobson Ivar, Stefan Bylund, Patrik Jonsson, Staffan Ehnebom, Using contracts and use cases to build pluggable architectures, Journal of Object-Oriented Programming, June, 1995.
48. Jacobson Ivar, A Large Commercial Success Story with Objects, Succeeding with Objects, Object Magazine, May 1996.
49. Jacobson Ivar and Sten Jacobson, Use-case engineering: Unlocking the power, Object Magazine, October 1996.
50. Jacobson Ivar, Griss Martin, and Jonsson Patrik, Software Reuse: Architecture, Process and Organization for Business Success, Reading, MA: Addison-Wesley, 1997.
51. Jones Capers, Assessment and Control of Software Risks, Upper Saddle River, NJ: Prentice-Hall, 1993.
52. Kruchten P.B. The 4+1 view model of architecture, IEEE Software, November 1995.

53. Kruchten Philippe, *The Rational Unified Process: An Introduction*, Reading, MA: Addison-Wesley, 1998.
54. Mann Anthony T., *Visual Basic5 Developer's Guide*, Indianapolis, IN: SAMS Publishing, 1997.
55. Mowbray Thomas J. and Malveau Raphael C., *CORBA Design Patterns*, New York: John Wiley and Sons, 1997.
56. OMG, Inc. *The Common Object Request Broker: Architecture and Specification (CORBA)*, Framingham, MA. 1996.
57. OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.
58. Royce Walker, TRW's Ada process model for incremental development of large software systems, Proceedings, 12th International Conference on Software Engineering, 1990, pp. 2–11.
59. Royce Walker, *Software Project Management: A Unified Framework*, Reading, MA: Addison-Wesley, 1998.
60. Rumbaugh James, Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice Hall, 1991.
61. Schneider Geri and Jason Winters, *Applying Use Cases: A Practical Approach*, Reading, MA: Addison-Wesley, 1998.
62. Stroustrup Bjarne, *The C++ Programming Language*, Third Edition, Reading, MA: Addison-Wesley, 1997.
63. The Unified Modeling Language for Object-Oriented Development, Documentation set, ver. 1.1, Rational Software Corp., September 1997.
64. Wieger Karl, *Use cases: Listening to the customer's voice*, Software Development, March 1997.

Алфавитный указатель

J

Java, 266, 318, 402

R

Rational Objectory Process, 447
Rational Unified Process, 447

A

абстрактный вариант
использования, 195
автоматизированный
процесс, 58
агрегирование, 236, 286
актант, 52, 64
бизнес-, 151, 155
конкретный, 175
определение, 379, 394
отдельный, 176
поиск, 174
пользовательский
интерфейс, 189
роли, 71, 161
системы, 155
активный класс, 111
и классы проектирования, 247
исполняемые компоненты, 308
определение, 271
требования к реализации, 289
Александер Кристофер
(Alexander Christopher), 101
альфа-тестирование, 425
артефакт, 50
бизнес-модель, 159
глоссарий, 167
и сотрудники, 54

артефакт (*продолжение*)
модель, 133
анализа, 209, 218, 222
проектирования, 244, 255
реализации, 296, 302
тестирования, 322, 326
набор требований, 146
рабочие процессы, 129
требования, 160, 167
управления, 50
фаза внедрения, 433
архитектор, 106
модель
анализа, 221
проектирования, 256
реализации, 303
требования, 170, 173
фаза
анализа и планирования
требований, 374, 379
внедрения, 431
построения, 411
архитектура, 88, 95, 106
билды, 96
выполняемые функции, 91
жизненный цикл, 439
и варианты использования, 36, 69,
95, 125, 177
итерации, 120
многоуровневая, 103, 401
модель
анализа, 220, 230
вариантов использования, 108
проектирования, 108, 270
реализации, 302, 309, 404

архитектура (*продолжение*)
 модель проектирования, 254
 образцы, 102
 подсистемы, 92, 94
 программ, 90
 простота понимания, 46
 риски, 124, 358
 требования, 397
 фаза

 анализа и планирования
 требований, 344, 371, 381
 построения, 418
 проектирования, 345, 402

ассоциация, 236

генерация кода, 314

класс проектирования, 286

атрибут

 имя, 314

 класса

 анализа, 209, 235
 проектирования, 285

Б

базовый

 рабочий процесс, 135, 347
 фаза, 377, 381, 393, 399, 406, 414,
 421, 428

уровень

 архитектуры, 98, 346, 385, 392
 объем вариантов
 использования, 386
 фаза проектирования, 346,
 392, 399

бета-выпуск, 43

 фаза

 внедрения, 347, 426, 430
 построения, 346, 410

бизнес-класс, 211

бизнес-модель, 40, 155

 артефакты, 159
 и модель предметной
 области, 144, 149
 поиск актантов, 174

бизнес-план, 114

 фаза

 анализа и планирования
 требований, 344, 368, 383, 384
 внедрения, 434
 построения, 422
 проектирования, 391, 407

бизнес-предложение, 383, 407

билд, 96, 121
 интеграция, 302, 311
 фаза

 построения, 420
 проектирования, 404

Боэм, Барри (Boehm, Barry), 115, 119

брокер, образец, 102

В

вариант использования, 34, 79

 абстрактные/конкретные, 195

актанты, 71, 174

бизнес, 151, 156

билды, 309, 310

выделение времени, 383

детализация, 182, 379, 396, 417

и архитектура, 36, 69, 95, 126, 177

и модель

 анализа, 64, 73, 230, 400
 проектирования, 65, 78, 276,
 282, 402

реализации, 85

как классификатор, 163

категории, 358

обобщение, 194

описание потока событий, 166,
 181, 188

определение, 71, 379, 395

отношение расширения, 196

пакет, 178

 анализа, 225

подсистемы, 164, 419

поиск, 176

 путь, 72, 163, 183, 280

 рабочие процессы, 67, 69

расстановка приоритетов, 181, 357,
 379, 396, 417

сервисные пакеты, 218

специальные

 требования, 166, 186

тестирование, 85, 323

трассируемость, 69

требования, 62, 64, 66, 70, 145, 201

веха, 40, 114, 132

 главная, 134, 351, 439
 промежуточная, 134, 352

Вигер, Карл (Wieger, Karl), 67

внешний выпуск, 117

внутренние выпуски, 100, 117

водопадный подход, 119, 367

возможная архитектура, 371

возможные варианты
использования, 71
возможный тестовый пример, 66
выделение времени, 350
варианты использования, 383
выполнимость проекта, 45, 344, 376
выпуск, 38, 49
внешний, 117
внутренний, 100, 117
итерация, 116
выражение — пример действия, 442
высшее руководство, 441, 444

Г

глоссарий (артефакт), 167
граничный класс, 74, 210
определение, 232
проектирование, 283

Д

действие, 34
дефект, 328, 338, 431
деятельность, 160, 174
Джонс, Каперс
(Jones,Capers), 120
диаграмма
взаимодействий
модель, 214, 249
описание вариантов
использования, 187
деятельности, 55, 163
описание варианта
использования, 186
классов, 75, 80
класс проектирования, 248
модель анализа, 213
кооперации, 76, 163
и диаграмма
последовательности, 81
модель анализа, 214, 233
последовательности, 81, 163
метки, 250
модель проектирования, 250,
278
тестовый пример на
интеграцию, 334
состояний, 64, 163
класс проектирования, 288
описание вариантов
использования, 187

дополнительные
требования, 146, 156
драйвер
в тестировании, 327, 420
Друкер, Питер Ф.
(Drucker, Peter F.), 118

Ж

жизненный цикл, 38, 49
архитектура, 439
итерации, 134
модель
анализа, 206
проектирования, 243
реализации, 295
тестирование, 321
условия, 439
утилиты, 61
жизнесспособность, 114

З

зависимость
компиляции, 298
компоненты, 298
пакеты анализа, 228
подсистемы, 267, 289
трассировка, 40, 53, 201
варианты
использования, 69
модели, 201, 297
заглушка, 298, 420
заинтересованное лицо, 50
заказной продукт, 432
затраты, 364
бизнес-предложение, 383, 407
модель анализа, 207
тестирование, 332
значение итоговое, 177

И

инженер
по вариантам использования,
модель, 222
анализа, 222
проектирования, 258
по компонентам
модель, 222, 258, 304, 418
тестирование, 317, 328, 422
по тестированию, 328, 353
инструктор, 444

интерфейс
 контракты, 409
 модель
 проектирования, 247, 290, 402
 реализации, 297, 312
 подсистемы, 82, 92, 269, 280
 пользователи, 145, 167, 188, 416
 требования, 156
 исполняемый компонент, 308
 использования, отношение, 194
 исходный текст программ, 50, 314
 итерация, 36, 114, 122
 архитектура, 97, 120
 базовый уровень, 132
 билды, 121, 303
 выпуски, 117
 жизненный цикл, 134
 и водопадный подход, 119, 123, 129
 интеграция, 122
 как мини-проект, 49, 116
 коллективная работа, 124
 модели, 135
 оценка, 365
 планирование и
 осуществление, 130, 351
 приоритеты вариантов
 использования, 357
 приращения, 132
 управление рисками, 119, 125, 128
 фазы, 129, 135
 итоговый доход, 176

K

кавычки, 74
 класс
 анализа, 211
 артефакты, 209, 219
 ассоциации и агрегации, 236
 атрибуты, 209, 235, 236
 границный класс, 74, 210, 215
 диаграмма классов, 213
 жизненный цикл, 206
 затраты, 207
 и варианты использования, 64,
 74, 78, 230, 399
 и классы проектирования, 270,
 271, 276
 и модель проектирования, 202,
 269
 и стереотипы, 75

класс (*продолжение*)
 интерфейсы, 269
 класс сущности, 74, 211, 216,
 220, 229
 модель анализа, 39, 40,
 200, 208, 233
 обобщения, 237
 описание, 216, 219
 определение, 231
 реализация варианта
 использования, 79,
 213, 242, 248
 роли и ответственности, 209,
 234, 235
 сервисные пакеты, 227
 совместно используемый, 226
 сотрудники, 221
 специальные
 требования, 217, 230
 стереотип, 283
 требования, 200, 206
 управляющий класс, 74, 212
 фазы, 397, 417, 418
 цели, 204
 варианты использования, 68
 граничный, 210, 283
 ответственности, 65, 78, 400
 подсистемы, 82
 предметной области, 148, 154, 226
 классы сущности, 229
 пакеты анализа, 226
 проектирования, 80, 84,
 245, 282, 288
 ассоциации и агрегации, 286
 атрибуты, 285
 диаграмма, 248, 288
 интерфейсы, 253, 290
 методы, 287
 обзор, 283
 обобщение, 286
 операции, 285
 определение, 270, 276
 реализация, 311
 специальные требования, 289
 трассировка компонентов, 297
 фаза проектирования, 402
 реализация, 311, 404
 роли, 73, 78, 149, 234
 совместно используемый, 227
 стереотипы, 75

- класс (*продолжение*)
 сущности, 74, 211, 220, 231
 определение, 230
 проектирование, 283
 сервисные пакеты, 220
 управляющий, 212, 232, 283
 эквивалентность, 316
- класс анализа
 модель анализа, 42
- классификатор, 64
 абстрактный/конкретный, 275
- вариант использования как, 163
 роль, 73
- клиент, 425, 432
 клиент/сервер, образец, 102
- код, 49
 билда, 420
 класса проектирования, 314
 тестирование, 317, 319
- компонент, 83, 296
 зависимость, 298
 интерфейс, 301
 исполняемый, 308
 тестирование, 327, 335
 файл, 314
- конкретный вариант использования, 195
- Константин, Ларри (Constantine, Larry), 192
- контракт интерфейса, 409
- контроль качества, 61
- конфигурация сети, 260, 402
- концепция, 371
- кооперация, 53, 65
 модель проектирования, 274
 нотация, 73
 шаблон, 101
- Л**
 линия жизни, 281
- М**
 матрица тестирования, 324
 машинный код, 49
 менеджер проекта, 353, 366
 внедрение Унифицированного
 процесса, 443
 фаза
 внедрения, 426, 434
 построения, 411, 422
 проектирования, 408
- механизм, 105
 минипроект, 36, 116
 модель, 50, 53
 базовый уровень, 132
 вариантов использования, 34, 39, 53, 160
 диаграмма вариантов
 использования, 70
 и модель анализа, 203
 описание архитектуры, 108, 166, 181
 основы, 192
 структурирование, 194, 396, 418
- и итерации, 135
 набор, 367, 433
 предметной области, 40, 148, 155
 проектирования, 40, 53, 242, 282
 артефакты, 245, 253
 архитектура, 108, 111, 253, 269, 274
 варианты использования, 78, 244, 276, 282, 402
 жизненный цикл, 243
 задачи, 242
 и модель, 65, 201, 246, 255, 269, 299, 312
 интерфейсы, 402
 конфигурация узлов и сети, 260, 283, 402
 кооперации, 274
 описание потока событий, 250
 персистентность, 274
 подсистемы, 251, 265, 280, 289
 рабочий процесс, 259
 распределение объектов, 273
 сотрудники, 256
 фаза, 382, 401, 418
 развертывания, 40
 и модель проектирования, 65, 255
 компоненты, 83
 фаза анализа и планирования
 требований, 382
 реализации, 40, 63, 294, 297, 309, 314, 317
 архитектура, 302, 309, 404
 жизненный цикл, 295
 и варианты использования, 83
 и модель проектирования, 65, 246, 297, 299, 312

модель (*продолжение*)

интеграция, 303, 309
 интерфейсы, 297, 313
 компоненты, 296, 301
 операции, 315
 подсистемы, 299, 311, 404
 рабочий процесс, 305
 сотрудники, 304
 требования, 248, 282, 289
 фаза, 382, 403, 419
 цели, 295

модуль рабочий, 152
 модульность функций, 94

Н**нагрузка**

тестирование, 327
 надежность, 145
 независимость, 52
 нефункциональные требования, 72, 145
 варианты использования, 166, 186

О

область деятельности системы, 372
 обобщение
 варианты использования, 194
 классы
 анализа, 237
 проектирования, 286
 кооперации, 276
 обобщенная итерация, 128, 342, 348
 обобщенный
 механизм проектирования, 273
 процесс, 56
 образцы
 архитектуры, 102
 проектирования, 102
 обратная связь, 46
 объект, 57
 активный, 111, 271, 308
 анализа, 213, 233
 бизнес-, 154
 предметной области, 144, 148
 проектирования, 250, 278, 280
 распределение, 102, 272
 реализация варианта использования, 76
 сущность, 211
 обычные процессы, 57

ограничения

проектирования, 157
 реализации, 157
 описание
 архитектуры, 90, 104, 107
 базовый уровень
 архитектуры, 100
 модель, 108, 166, 179, 219, 254, 255, 256, 302

варианта использования, 177

диаграмма
 деятельности, 186
 обзор, 178
 содержание, 184, 188

потока событий, 77
 модель, 216, 250

требования, 166, 182, 188

организация итерации, 136

образец, 49
 типы, 369

основной рабочий процесс, 128
 фаза анализа и планирования требований, 376

ответственность, класс, 65, 78
 анализа, 209, 234, 235, 400

отдельный актант, 175

отношение, 53, 65

включения, 198
 использования, 194

класс
 анализа, 236
 проектирования, 245

класс анализа, 209

обобщения, 194

расширения, 196

отрицательный тест, 325

оценка, 351, 366

итерации, 365

фазы

анализа и планирования требований, 385
 внедрения, 428, 435
 построения, 412, 421
 проектирования, 390, 407

П**пакет**

анализа, 217, 239
 зависимости, 228
 определение, 225

- пакет (*продолжение*)
 подсистемы
 проектирования, 262
 фазы проектирования, 401
 вариантов использования, 178
 сервисный, 218, 228, 252
 перенос данных, 433
 персистентность, 274, 283
 персонал, 44
 плавательная дорожка, 55
 план сборки, 302
 реализация, 309
 фаза
 построения, 420
 проектирования, 404
 планирование, 350
 итерации, 130, 352
 рисков, 354, 359
 фазы
 внедрения, 426
 построения, 408
 проектирования, 386
 поборник, 444
 повторное использование, 93
 подсистемы проектирования, 251
 сервисные пакеты, 220
 подсистема, 82
 архитектура, 92
 варианты использования, 156
 верхнего уровня, 52
 зависимости, 267, 290
 интерфейсы, 268
 модель
 проектирования, 251, 262, 280,
 290, 402
 реализации, 299, 311, 404
 сервис, 418
 совместно используемая
 функциональность, 264
 уровни, 103
 фаза построения, 418
 пользователь, 34
 представление, 52, 90
 прибыль на вложенный
 капитал, 384, 407
 приемо-сдаточное тестирование, 428
 приказ о реинжиниринге, 442
 приращение, 198
 варианты использования, 69
 интеграция, 302
 проверка третьей стороной, 425
 программное обеспечение, 50
 жизненный цикл, 61
 система, 265
 соглашение на разработку, 426
 продукт, 39, 44
 и клиенты, 432
 направление, 413
 новый, 363, 364
 размер, 383
 широкого распространения, 432
 проект, 44
 возможность создания, 45, 344, 376
 график, 46
 мини-проекты, 36
 типы, 360
 циклы, 49
 проектировщик пользовательских
 интерфейсов, 169, 173
 прокси, образец проектирования, 102
 прототип
 интерфейса пользователя, 167, 188,
 189, 193
 спецификация, 145
 фаза, 395, 416
 физический, 192
 концептуальный, 345
 фазы анализа и планирования
 требований, 374
 процедура тестирования, 85, 326, 335
 компоненты, 327, 336
 процесс разработки программного
 обеспечения, 33, 44, 58
 автоматизированный, 58
 общий, 56
 рабочие процессы, 54
 специализация, 56
 управляемый вариантами
 использования, 35
 утилиты, 58
 экземпляр, 54
 путь
 базовый, 185
 вариант использования, 72, 163, 280
 основной, 183
- P**
- рабочий процесс, 41, 56, 159
 артефакты в, 129
 вариант использования, 69

- рабочий процесс (*продолжение*)
итерации, 128
анализа и планирования
требований, 350, 374
обобщенный, 342, 348, 350
построения, 413, 414, 422
проектирования, 391, 407
фаза внедрения, 429, 436
- модель
анализа, 223
проектирования, 259
реализации, 306
обобщенная итерация, 342,
348, 350
тестирования, 35, 330
требования, 147, 171, 199
- ранжированный список вариантов
использования, 356, 357, 417
- расширение
отношение, 196
- реализация варианта использования
бизнес-, 152
диаграмма
классов, 75
кооперации, 76
интерфейсы, 270
- модель
анализа, 79, 213, 244
проектирования, 79, 245, 274,
278, 280
нотация, 73, 75
описание потока событий, 77
специальные требования, 234
тестовый пример, 324
- реальный вариант
использования, 195
- регрессионное
тестирование, 130,
322, 334
- результатирующая ценность, 176
- ресурс, 48
- риск производительности, 126
- роль актанта, 175
класс, 73, 78, 149, 234
сотрудник, 47, 155, 161
- руководство пользователя, 69, 413
- С**
- свойство, 143, 378
- связь, 53
- сервисная подсистема, 82, 84, 94
модель
проектирования, 252
реализации, 301
процедура
тестирования, 335
фаза построения, 418
- сервисный пакет, 219
модель
анализа, 227
проектирования, 252
- система
Ericsson AXE, 94
систем, 164
система-полуфабрикат, 442
системное программное
обеспечение, 265, 267
- системный
аналитик, 168, 174
требования, 175
фаза, 374, 379, 412
интегратор, 305, 420
тест, 334, 337, 406
тестер, 329
- сообщение, 71, 249
- интерфейс, 281
- объекты
анализа, 233
проектирования, 278
- сотрудник, 47, 50
актант как, 161
бизнес-модель, 154, 160
и артефакты, 54
итерации, 348
- модель
анализа, 221
проектирования, 256
роли, 161
тестирование, 329
требования, 167
фазы
внедрения, 427
построения, 412
- специализированный
процесс, 56
- специальные требования
варианты
использования, 166, 186
класс проектирования, 289
модель анализа, 217, 230, 234

спецификатор вариантов
использования, 169, 173, 182
фаза
построения, 412
среда разработки, 376
средний уровень, 95, 265
стандартизация, 93
стереотип, 56
бизнес-модель, 160
класс проектирования, 247
компоненты, 296
модель анализа, 74, 209, 283
сервисные
пакеты, 219
подсистемы, 253
тестовые примеры, 85
стиль архитектуры, 90
страницная нотация, 55
структура
группы, 46, 124, 390
команды, 45
сущность
бизнес-, 152, 153
сценарий тестирования, 327

Т

тест целостности, 303
осуществление, 336, 337, 406, 421
проектирование, 332
тестер целостности, 329
тестирование, 320, 330
альфа, 425
белого ящика, 66, 323
бета, 346, 411, 426, 430
варианты использования, 85, 86,
323
драйвера и заглушки, 298, 327, 420
жизненный цикл, 321
интеграции, 303, 333, 336
конфигурации, 325
матрица, 325
модуля, 315
планирование и
осуществление, 328, 338
под нагрузкой, 324
приемо-сдаточное, 428
рабочий процесс, 35, 330
регрессионное, 130, 321, 334
системы, 334, 337
сотрудники, 329

тестирование (*продолжение*)
спецификации, 316
структурь, 317
установки, 325
фазы
анализа и планирования
требований, 382
построения, 421
проектирования, 405
цели, 320
целостности, 332, 406, 421
черного ящика, 66, 323
тестовый пример, 85, 322
модель тестирования, 40, 66
регрессионный, 334
системный, 334, 337
целостность, 332
технический артефакт, 50
требования, 54, 140, 151
артефакты, 160, 166
архитектура, 397
бизнес-модель, 144, 151, 153
возможные, 143
двусмысленности, 372
дополнительные, 146, 156
исходная точка, 142
к производительности, 186
к реализации, 248, 251, 282, 289
модель
анализа, 201
предметной области, 144, 150
набор требований, 146
нефункциональные, 72, 145, 146,
166, 186
производительности, 145
рабочий процесс, 171, 199
риски, 126, 359
сотрудники, 167, 172
фаза
построения, 416
проектирования, 148, 390
функциональные, 66, 70, 72, 145

У

узел, 111
компоненты, 309
модель
проектирования, 260, 283
развертывания, 255
подсистемы, 264

узел (*продолжение*)

фаза проектирования, 402

унаследованная система, 95

модель

анализа, 206

проектирования, 252

Унифицированный процесс, 32

архитектура, 35

варианты использования, 34

жизненный путь, 38

итерации и приращения, 36, 41

обзор, 438

определение, 33

переход на, 441, 445

преимущества, 448

специализация, 445, 447

темы, 440

Унифицированный язык

моделирования (UML), 33, 441

визуальное моделирование, 60

нотация, 149

стандартизация, 93

управление, 441

рисками, 45

архитектура, 125, 358

итерация, 119, 124, 128

особо опасные риски, 373

планирование, 354, 359

расстановка приоритетов

вариантов

использования, 357

список рисков, 354, 355

требования, 126, 359

фаза, 345, 391, 414

требованиями, 61

управляемая итерация, 36

управляющий класс, 74, 213

определение, 232

проектирование, 283

уровень, 103, 401

пакеты анализа, 228, 229

подсистемы проектирования, 268

уровни, образец, 102

установка продукта, 432

утилиты, 44, 58, 61

программиста, 61

фазы

анализа и планирования

требований, 376

проектирования, 404

Ф

фаза, 39, 41

анализа и планирования

требований, 41, 114, 135, 368

бизнес-план, 344, 369,

373, 383, 384

жизненный цикл, 439

задачи, 133

и фаза проектирования, 386

итерация, 350

модель, 380, 382

область действия системы, 372

организации-разработчики, 370

оценка, 384

планирование и

осуществление, 370

результаты, 387

риски, 345

тестирование, 382

требования, 147, 378

внедрения, 43, 115, 136, 424, 427

бета-выпуск, 426, 430

бизнес-план, 435

задачи, 133, 424

и фаза построения, 423

итерация, 429

клиенты и продукты, 432

оценка, 428, 435

планирование, 426

программное обеспечение, 425

результаты, 436

сотрудники, 427

итерация, 129, 135

оценка, 365

планирование, 350

построения, 42, 115, 135, 410

бизнес-план, 422

жизненный цикл, 440

и фаза, 408, 411, 423

модель, 417, 418, 419

оценка, 422

планирование и

осуществление, 412, 422

рабочий процесс

итерации, 413, 422

результаты, 423

риски, 414

тестирование, 421

требования, 416, 417

цели, 133, 346

фаза (*продолжение*)
проектирования, 42, 114, 136, 387,
388, 399

архитектура, 345
бизнес-план, 391, 407
жизненный цикл, 439
и фаза построения, 408, 411
итерация, 391
модель, 397, 401, 403
оценка, 407
планирование и
осуществление, 386
результаты, 409
риски, 391
тестирование, 405
требования, 148, 390
цели, 133

файл компонента, 314
физические
требования, 157
функциональное
описание, 34

функциональные требования, 66
как варианты
использования, 70, 145

Ц
целииний проект, 49

Ч
черный ящик, тестированис, 66, 323

Ш
шаблон, 101

Э
эквивалентность классов, 316
экземпляр
вариант использования, 162
класса, 68
процесса, 54

Я
язык программирования, 245, 314