

ПРИНЦИПЫ РАБОТЫ С ТРЕБОВАНИЯМИ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Унифицированный подход

ДИН ЛЕФФИНГУЭЛЛ
ДОН УИДРИГ

Предисловие Эда Йордана

**“Самая полезная из всех
когда-либо написанных
книг о требованиях.”**

**Авторы определенно
знают, о чем пишут.
Ее обязательно должны
прочитать все
практики”.**

— Ал Дэвис, Omni-Vista, Inc.



Managing Software Requirements

A Unified Approach

Dean Leffingwell

Don Widrig



ADDISON-WESLEY

Boston ♦ San Francisco ♦ New York ♦ Toronto ♦ Montreal

London ♦ Munich ♦ Paris ♦ Madrid ♦ Capetown ♦ Sidney ♦ Tokyo

Singapore ♦ Mexico City

Принципы работы с требованиями к программному обеспечению

Унифицированный подход

**Дин Леффингуэлл
Дон Уидриг**



Издательский дом “Вильямс”
Москва ♦ Санкт-Петербург ♦ Киев
2002

ББК 32.973.26-018.2.75

Л53

УДК 681.3.07

Издательский дом "Вильямс"

Перевод с английского и редакция Н.А. Ореховой

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

Леффингуэлл, Дин, Уидриг, Дон.

Л53 Принципы работы с требованиями к программному обеспечению. Унифицированный подход. : Пер. с англ. – М. : Издательский дом "Вильямс", 2002. – 448 с. : ил. – Парал. тит. англ.

ISBN 5-8459-0275-4 (рус.)

Книга посвящена вопросам формирования требований и работе с ними при разработке сложных систем программного обеспечения. Недостаточное внимание к этому аспекту разработки может привести к превышению расходов, затягиванию сроков выполнения и даже полной неудаче проекта. Авторы предлагают хорошо зарекомендовавшие себя методы выявления, документирования, реализации и тестирования требований, используя для их описания как прецеденты, так и более традиционные методы. Особое внимание уделяется вопросам уяснения потребностей пользователей, определения масштаба проекта и эффективной обработки изменений. Все этапы иллюстрируются обсуждением полномасштабного рабочего примера.

Книга предназначена для всех участников проекта – как членов команды разработчиков, так и пользователей или заказчиков. Ее задача – помочь создать в рамках отведенного времени и бюджета высококачественную систему программного обеспечения, удовлетворяющую реальные потребности клиентов.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc., Copyright © 2000

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2002

ISBN 5-8459-0275-4 (рус.)

ISBN 0-2016-1593-2 (англ.)

© Издательский дом "Вильямс", 2002

© Addison-Wesley, 2000

Оглавление

Предисловие	19
Введение	29
1 Проблема требований	33
2 Введение в управление требованиями	41
3 Команда разработчиков	49
Часть 1. Анализ проблемы	57
4 Пять этапов анализа проблемы	59
5 Моделирование бизнес-процессов	73
6 Инженерия систем, интенсивно использующих программное обеспечение	81
Часть 2. Понимание потребностей пользователей	99
7 Задача выявления требований	101
8 Функции продукта или системы	105
9 Интервьюирование	111
10 Совещания, посвященные требованиям	121
11 Мозговой штурм и отбор идей	129
12 Раскадровка	141
13 Применение прецедентов	149
14 Обыгрывание ролей	155
15 Создание прототипов	159
Часть 3. Определение системы	165
16 Организация информации о требованиях	169
17 Документ-концепция	177
18 Лидер продукта	185
Часть 4. Управление масштабом	191
19 Проблема масштаба проекта	193

20	Задание масштаба проекта	197
21	Умение обращаться с заказчиком	209
22	Управление масштабом и модели процесса разработки программного обеспечения	213
Часть 5. Уточнение определения системы		223
23	Требования к программному обеспечению	225
24	Уточнение прецедентов	243
25	Спецификация требований к программному обеспечению (Modern Software Requirements Specification)	253
26	Неоднозначность и уровень конкретизации	261
27	Критерии качества требований к программному обеспечению	267
28	Теоретически обоснованные формальные методы спецификации требований	281
Часть 6. Построение правильной системы		293
29	Как правильно построить “правильную” систему: общие положения	295
30	От понимания требований к реализации системы	301
31	Использование трассировки для поддержки верификации	313
32	Проверка правильности системы	325
33	Применение метода анализа дивидендов для определения объема V&V-действий	335
34	Управление изменениями	341
35	С чего начать	359
Приложения		369
Приложение А. Артефакты системы HOLIS		371
Приложение Б. Образец документа-концепции		403
Приложение В. Образец пакета Modern SRS Package		411
Приложение Г. Принципы управления требованиями в стандартах SEI-CMM и ISO 9000		419
Приложение Д. Принципы управления требованиями в Rational Unified Process		427
Предметный указатель		441

Содержание

Предисловие	19
Введение	29
1 Проблема требований	33
Цель	33
Немного статистики	34
Основные причины успеха и провала проекта	35
Частота возникновения ошибок, связанных с требованиями	36
Высокая цена ошибок требований	37
Заключение	40
2 Введение в управление требованиями	41
Определения	41
Что такое требование	41
Что такое управление требованиями	42
Применение методов управления требованиями	43
Типы программных приложений	43
Применение методов управления требованиями к системам общего вида	44
Наш маршрут	44
Область проблемы	44
Потребности заинтересованных лиц	45
Переход к области решения	45
Функции системы	45
Требования к программному обеспечению	46
Понятие прецедентов	46
Заключение	46
3 Команда разработчиков	49
Разработка программного обеспечения как командная деятельность	50
Профессиональные навыки, которыми должна обладать команда для эффективного управления требованиями	51
Члены команды имеют различные профессиональные навыки	52
Организация команд, разрабатывающих программное обеспечение	52
Рабочий пример	53
Предварительная информация для рабочего примера	53
Команда разработчиков программного обеспечения HOLIS	53
Заключение	54
Часть 1. Анализ проблемы	57
4 Пять этапов анализа проблемы	59
Этап 1. Достижение соглашения об определении проблемы	61

Постановка проблемы	61
Этап 2. Выделение основных причин – проблем, стоящих за проблемой	62
Устранение корневых причин	63
Этап 3. Выявление заинтересованных лиц и пользователей	64
Этап 4. Определение границ системы-решения	66
Этап 5. Выявление ограничений, налагаемых на решение	68
Заключение	70
Далее...	70
5 Моделирование бизнес-процессов	73
Цели моделирования бизнес-процесса	74
Использование методов инженерии программного обеспечения для моделирования бизнес-процессов	74
Выбор подходящего метода	75
Унифицированный язык моделирования (UML)	75
Моделирование бизнес-процесса с использованием концепций UML	76
От моделей бизнес-процесса к модели системы	77
Когда использовать моделирование бизнес-процесса	78
Заключение	78
Далее...	79
6 Инженерия систем, интенсивно использующих программное обеспечение	81
Что такое системная инженерия	82
Основные принципы системной инженерии	82
Декомпозиция сложных систем	83
Размещение требований в системной инженерии	84
Производные требования	84
“Тихая” революция	85
Столкновение поколений: седобородые встречаются с молодыми и самонадеянными	86
Как избежать проблемы создания системы типа “печной трубы”	87
Когда подсистемы являются субконтрактами	88
Как сделать систему работоспособной	88
Рабочий пример	90
Предварительные потребности пользователя	90
Анализ проблемы	91
HOLIS: система, акторы и заинтересованные лица	92
Применение принципов системной инженерии к HOLIS	93
Подсистемы системы HOLIS	94
Часть 2. Понимание потребностей пользователей	99
7 Задача выявления требований	101
Преграды на пути выявления требований	101
Синдром “да, но...”	101
Синдром “шестокрытых руин”	103
Синдром “пользователя и разработчика”	103

Методы выявления требований	104
8 Функции продукта или системы	105
Потребности заинтересованных лиц и пользователей	105
Функции	106
Управление сложностью путем выбора уровня абстракции	108
Атрибуты функций продукта	108
9 Интервьюирование	111
Контекст интервью	112
Контекстно-свободные вопросы	112
Добавление контекста	112
Момент истины: интервью	115
Сбор данных о потребностях	116
Заключение аналитика: $10+10+10 \neq 30$	116
Рабочий пример	116
Замечание по поводу анкетирования	118
10 Совещания, посвященные требованиям	121
Ускорение процесса принятия решений	121
Подготовка к совещанию	122
Распространение концепции	122
Гарантия участия основных заинтересованных лиц	122
Логистика	122
Подготовительные материалы	123
Роль ведущего	125
Составление повестки дня	125
Проведение совещания	126
Проблемы и приемы	126
Мозговой штурм и отбор идей	128
Результат и продолжение	128
11 Мозговой штурм и отбор идей	129
“Живой” мозговой штурм	130
Отбор идей	131
Отсечение	131
Группировка идей	132
Определение функций	132
Расстановка приоритетов	133
Мозговой штурм с использованием Web	134
Рабочий пример: совещание по вопросу требований к системе HOLIS 2000	135
Присутствующие	135
Совещание	136
Заседание	136
Анализ результатов	138

12	Раскадровка	14
	Типы раскадровок	14
	Что делают раскадровки	14
	Средства и методы раскадровки	14
	Советы по раскадровке	14
	Заключение	14
13	Применение прецедентов	14
	Построение модели прецедентов	15
	Применение прецедентов к выявлению требований	15
	Рабочий пример. Прецеденты системы HOLIS	15
	Заключение	15
14	Обыгрывание ролей	15
	Как играть роль	15
	Методы, аналогичные обыгрыванию ролей	15
	Сценарный просмотр	15
	CRC-карточки (Class-Responsibility-Collaboration, класс-обязанность-взаимодействие)	15
	Заключение	15
15	Создание прототипов	15
	Виды прототипов	15
	Прототипы требований	16
	Что прототипировать	16
	Построение прототипа	16
	Оценка результатов	16
	Заключение	16
Часть 3. Определение системы		16
16	Организация информации о требованиях	16
	Организация требований к сложным аппаратным и программным системам	17
	Организация требований к семействам продуктов	17
	“Будущие” требования	17
	Оличие бизнес-требований и требований маркетинга от требований к продукту	17
	Рабочий пример	17
	Заключение	17
17	Документ-концепция	17
	Компоненты документа-концепции	17
	Документ Delta Vision	18
	Документ-концепция версии 1.0	18
	Документ-концепция версии 2.0	18
	Документ Delta Vision для уже существующей системы	18

18 Лидер продукта	185
Роль лидера продукта	185
Лидер продукта в среде программных продуктов	186
Лидер продукта в отделе информационных технологий и систем (IS / IT)	188
Часть 4. Управление масштабом	191
19 Проблема масштаба проекта	193
Составляющие масштаба проекта	193
Трудный вопрос	196
20 Задание масштаба проекта	197
Базовый уровень требований	197
Установка приоритетов	198
Оценка трудозатрат	199
Добавление элемента риска	200
Сокращение масштаба	201
Обоснованная первая оценка	202
Рабочий пример	203
21 Умение обращаться с заказчиком	209
Привлечение заказчиков к управлению масштабом их проекта	209
Сообщение о результате	210
Переговоры с заказчиком	210
Управление базовым уровнем	211
Официальное изменение	212
Неофициальное изменение	212
22 Управление масштабом и модели процесса разработки программного обеспечения	213
“Модель водопада”	214
Сpirальная модель	216
Итеративный подход	217
Фазы жизненного цикла	218
Итерации	218
Рабочие процессы	219
Что делать, что делать...	220
Часть 5. Уточнение определения системы	223
23 Требования к программному обеспечению	225
Определение требований к программному обеспечению	226
Взаимосвязь между функциями и требованиями к программному обеспечению	228
Дilemma требований: что или как	229
Исключение информации, связанный с управлением проектом	229
Исключение информации, относящейся к проектированию	230
Больше внимания требованиям, а не проектированию	231
Итерационный цикл разработки требований и проектирования	232

Дальнейшая характеристика требований	233
Функциональные требования к программному обеспечению	233
Нефункциональные программные требования	234
Ограничения проектирования	238
Являются ли ограничения проектирования истинными требованиями?	240
Использование “дочерних” требований для повышения уровня конкретизации	241
Организация дочерних требований	242
Далее...	242
24 Уточнение прецедентов	243
Вопросы, на которые нужно ответить	244
Когда следует использовать методологию прецедентов	244
Когда прецеденты не являются наилучшим вариантом	244
Проблема избыточности	245
Совершенствование спецификаций прецедентов	245
Эволюция прецедентов	247
Какие действия включить в прецедент	247
Рабочий пример. Строение простого прецедента	248
Определение акторов	248
Дать название прецеденту	248
Составление краткого описания	249
Определение потока событий	249
Выявление пред- и постусловий	251
Далее...	252
25 Спецификация требований к программному обеспечению (Modern Software Requirements Specification)	253
Пакет спецификаций требований к программному обеспечению (Modern SRS Package)	254
Кто отвечает за Modern SRS Package	256
Организация пакета Modern SRS Package	256
Документирование функциональных требований	258
Далее...	260
26 Неоднозначность и уровень конкретизации	261
Нахождение “золотой середины”	261
Пример неоднозначности. У Мери был маленький барашек	264
Как избежать неоднозначности	265
Что делать?	266
27 Критерии качества требований к программному обеспечению	267
Девять показателей качества	267
Корректные требования	268
Недвусмысленные требования	269
Полнота набора требований	269
Непротиворечивость набора требований	271
Упорядочение требований по их важности и стабильности	272

Проверяемые требования	273
Модифицируемый набор требований	274
Трассируемые требования	274
Понимаемые требования	275
Показатели качества для модели прецедентов	276
Спецификации прецедентов	276
Акторы прецедента	278
Критерии качества пакета Modern SRS Package	278
Хорошо составленное оглавление	278
Хороший индекс	279
История исправлений	280
Глоссарий	280
28 Теоретически обоснованные формальные методы спецификации требований	281
Псевдокод	282
Конечные автоматы	283
Таблицы решений	284
Графические деревья решений	285
Диаграммы деятельности	285
Модели сущность-связь	286
Объектно-ориентированные модели	287
Схемы потоков данных	288
Ведение спецификаций	289
Рабочий пример	290
Часть 6. Построение правильной системы	293
29 Как правильно построить “правильную” систему: общие положения	295
Проверка того, что разработка находится на правильном пути	296
Принципы верификации программного обеспечения	296
Затраты на верификацию	297
Верификация на всех уровнях	297
Доводы в пользу верификации	298
Проверка корректности результатов разработки	298
Обработка изменений, возникающих в процессе разработки	299
Далее...	299
30 От понимания требований к реализации системы	301
Отображение требований в технический проект и программный код	301
Проблема ортогональности	302
Объектно-ориентированный подход	303
Прецедент в роли требования	304
Осуществление перехода	304
Моделирование систем программного обеспечения	304
Роль модели прецедентов в архитектуре	307
Реализация прецедентов в модели проектирования	308

Структурная и поведенческая части коопераций	308
Использование коопераций для реализации наборов отдельных требований	310
От проектирования к реализации	310
Заключение	311
Далее...	311
31 Использование трассировки для поддержки верификации	313
Роль трассировки при верификации требований	313
Неявная и явная трассировка	314
Дополнительные возможности, предоставляемые трассировкой	316
Использование автоматических средств трассировки	316
Поддержка отношений трассировки	318
Работа без автоматических средств трассировки	320
Пропущенные отношения	321
“Лишние” отношения	322
Размышления о верификации и трассировке	323
Далее...	323
32 Проверка правильности системы	325
Проверка правильности	326
Приемо-сдаточные испытания	326
Тестирование с целью проверки правильности	326
Трассировка при проверке правильности	327
Основанное на требованиях тестирование	327
Рабочий пример. Тестирование прецедентов	328
Описание тестового примера 1	328
Трассировка тестовых примеров	330
Тестирование дискретных требований	330
Пропущенные отношения проверки правильности	331
“Лишние” отношения проверки правильности	332
Тестирование ограничений проектирования	333
Далее...	334
33 Применение метода анализа дивидендов для определения объема V&V-действий	335
Глубина и покрытие	336
Глубина V&V	336
V&V-покрытие	337
Что подвергать верификации и проверке правильности	337
Вариант 1. Верифицировать и проверять правильность всех элементов	337
Вариант 2. Анализ рисков для определения необходимости V&V	338
Анализ рисков и анализ дивидендов (ROI)	339
Далее...	340
34 Управление изменениями	341
Почему изменяются требования	341

Внешние факторы	341
Внутренние факторы	342
Наш враг – мы сами	343
Процесс управления изменениями	344
Шаг 1. Осознать, что изменения неизбежны, и разработать план управления изменениями	344
Шаг 2. Формирование базового уровня требований	345
Шаг 3. Задание единого канала контроля изменений	346
Шаг 4. Использование системы контроля изменений для их фиксации	346
Шаг 5. Иерархическое управление изменениями	348
Управление конфигурацией требований	350
Управление изменениями при поддержке программных средств	352
Элементы, на которые воздействует изменение	352
Контрольный журнал изменений	354
Управление конфигурацией и управление изменениями	354
Заключение	355
35 С чего начать	359
Посвящение	359
Чему мы научились	359
Введение	359
Набор приемов 1. Анализ проблемы	360
Набор приемов 2. Понимание потребностей пользователей	361
Набор приемов 3. Определение системы	361
Набор приемов 4. Управление масштабом	362
Набор приемов 5. Уточнение определения системы	363
Набор приемов 6. Построение правильной системы	363
Рецепт работы с требованиями	364
Упрощающие предположения	364
Рецепт	364
Теперь – к следующей версии!	367
Приложения	369
Приложение А. Артефакты системы HOLIS	371
Предварительная информация для рабочего примера	371
Компания Lumenations, Ltd.	371
Команда разработчиков программного обеспечения HOLIS	371
Набор приемов 1. Анализ проблемы	373
Постановка существующей в компании Lumenations проблемы	373
Блок-схема системы с указанием акторов	374
Описание акторов	375
Описание других заинтересованных лиц	376
Налагаемые на решение ограничения	376
Набор приемов 2. Понимание потребностей пользователя	377

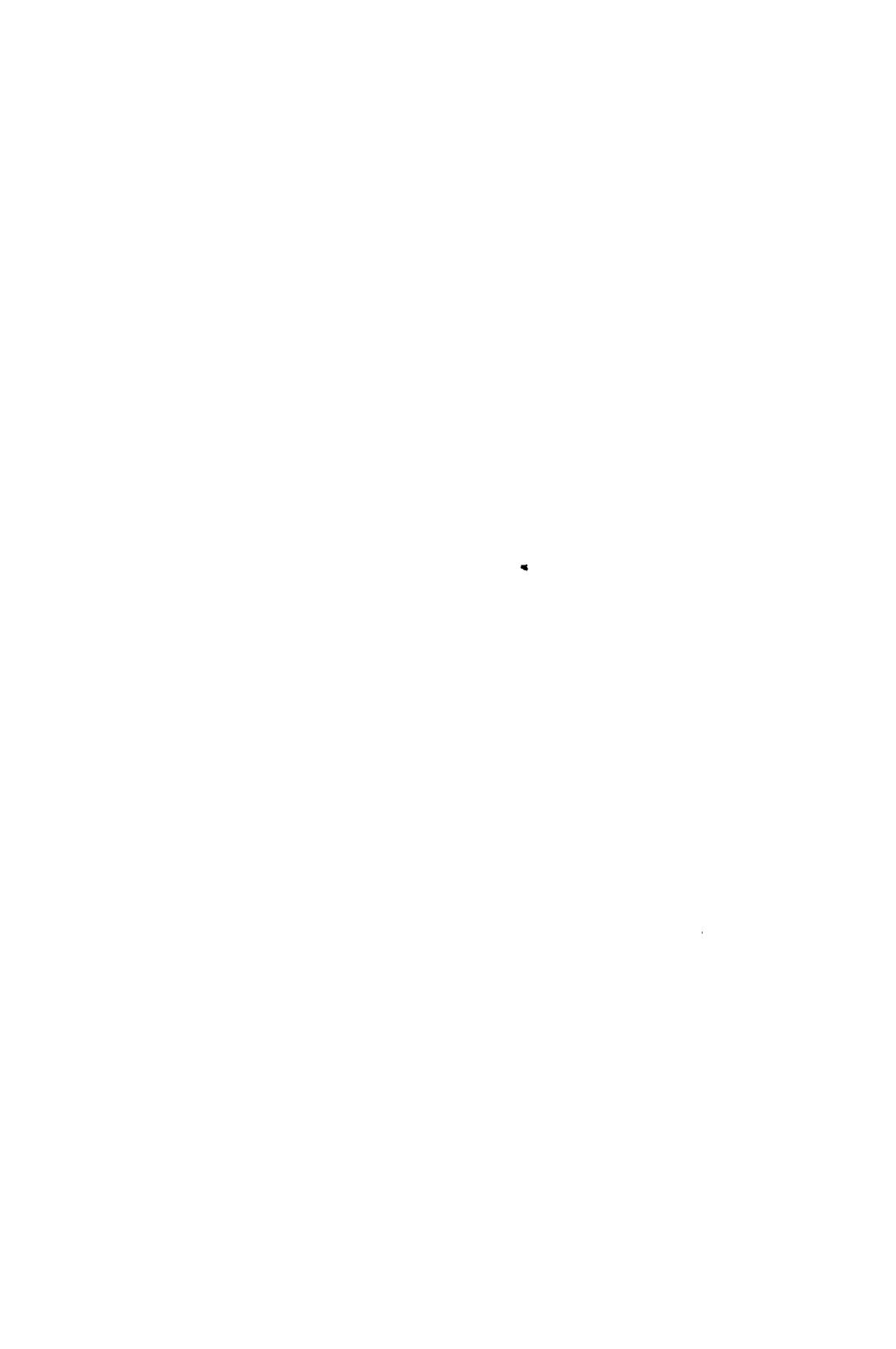
Краткий обзор потребностей пользователей, выявленных с помощью интервью	377
Совещание по вопросу требований к системе HOLIS 2000	378
Краткое описание модели прецедентов системного уровня системы HOLIS	382
Набор приемов 3. Определение системы	382
Организация требований к системе HOLIS	382
Документ-концепция (Vision Document) системы HOLIS	383
Набор приемов 4. Управление масштабом	388
Набор приемов 5. Уточнение определения системы	393
Образец описания прецедента системы HOLIS:	
Управление освещением	393
Спецификация требований к программному обеспечению подсистемы “Центральный блок управления” системы HOLIS	395
Набор приемов 6. Построение правильной системы	400
HOLIS 2000. Образец тестового примера 01: тест прецедента “Управление освещением”	400
HOLIS 2000. Образец тестового примера 02: тест протокола обмена сообщениями	401
Приложение Б. Образец документа-концепции	403
Приложение В. Образец пакета Modern SRS Package	411
Приложение Г. Принципы управления требованиями в стандартах SEI-CMM и ISO 9000	419
Принципы управления требованиями в стандарте SEI-CMM	419
Принципы управления требованиями в стандарте ISO 9000	424
Приложение Д. Принципы управления требованиями в Rational Unified Process	427
Структура RUP	428
Принципы управления требованиями в RUP	429
Анализ проблемы	431
Понимание потребностей заинтересованных лиц	432
Определение системы	433
Управление масштабом проекта	433
Уточнение определения системы	435
Обработка изменений требований	435
Интеграция процессов	437
Список литературы	439
Предметный указатель	441

Эту книгу я посвящаю Бекки, моей жене и лучшему другу.

- Дин Леббинз

Эту книгу я посвящаю моему соавтору, Дину, благодаря которому эта работа обрела смысл, и моей подруге Барбаре, благодаря которой моя жизнь обрела смысл.

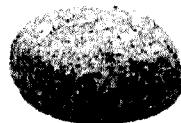
- Дон Уидриг



Предисловие

Проблема камня

Одна из моих студенток назвала совокупность рассматриваемых в данной книге вопросов проблемой "камня". Она занимается разработкой программного обеспечения в исследовательской лаборатории, и ее заказчики часто предлагают ей задания, которые можно условно представить как "Принесите мне камень". Когда камень доставлен, заказчик некоторое время изучает его и говорит: "Да, но на самом деле то, что я хотел, — это небольшой голубой камень". После доставки небольшого голубого камня выясняется, что это должен быть *круглый* небольшой голубой камень.



Наконец, может оказаться, что заказчик все время думал о небольшом голубом кусочке мрамора или он и сам в точности не уверен, чего же он хочет, но небольшого кусочка голубого мрамора — или даже кусочка голубого мрамора, имитирующего кошачий глаз, — вполне достаточно. Кроме того, его мнение о требованиях могло меняться в промежутке между доставкой первого (большого) и третьего (небольшого голубого) камня.



При каждой последующей встрече с заказчиком разработчик, как правило, вопрошают: "Чего же Вы хотите?". Разработчик недоволен, поскольку он представлял себе нечто совершенно иное, долго и трудно работая над созданием камня с ранее описанными характеристиками; заказчик также расстроен, поскольку, даже если ему и сложно объяснить, чего же он хочет, он считает, что выразил это достаточно ясно. Эти разработчики просто ничего не понимают!

Ситуация усложняется тем, что в большинстве реальных проектов принимает участие гораздо больше людей. Помимо заказчика и разработчика, которые, конечно же, могут иметь различные имена и названия, вероятно, должны быть маркетологи, специалисты по контролю качества, а также менеджеры нижнего и верхнего уровней и множество заинтересованных лиц, на чьи повседневные действия повлияет разработка новой системы.

Все они могут пострадать от проблем определения приемлемых "камней", в особенности потому, что зачастую в современном быстроменяющемся деловом мире с высоким уровнем конкуренции нет времени для того, чтобы отбросить дорогостоящий двухгодичный "проект по созданию камня" и начать все сначала. Задача состоит в том, чтобы сделать все правильно с первого раза с помощью некоего итеративного процесса, в рамках которого заказчик постепенно определяет, какой собственно камень он хочет получить.

Это достаточно сложно сделать, когда мы имеем дело с осозаемыми физическими предметами типа настоящего камня. Большинство коммерческих и государственных организаций сегодня интенсивно используют информационные технологии, поэтому, даже если они формально занимаются созданием и продажей неких "камней", с высокой степенью вероятности эти "камни" будут оснащены встроенным компьютерными системами. Даже если это не так, вполне возможно, что предприятие нуждается в тщательно разработанных системах для отслеживания продаж "камней" с помощью средств электрон-

ной коммерции, учета заказчиков, конкурентов и поставщиков, а также для хранения другой информации, необходимой для поддержания конкурентоспособности на рынке.

Системы программного обеспечения по своей природе являются неосязаемыми, абстрактными, сложными и, по крайней мере в теории, бесконечно изменяемыми. Поэтому, когда заказчик высказывает требования к "каменной системе" нечетко, он делает это, предполагая, что впоследствии сможет их уточнить, изменить и дополнить. Все было бы прекрасно, если бы разработчик и остальные участники процесса создания, тестирования, развертывания и обслуживания данной системы могли осуществлять это с нулевыми затратами времени и денег, но так не получается.

К сожалению, часто вообще ничего не получается: более половины разрабатываемых в настоящее время проектов систем программного обеспечения значительно превысили отведенное на них время и бюджет, а выполнение 25–33% проектов было прекращено, несмотря на то что уже были истрачены баснословные средства.

Цель данной книги – предотвратить подобные сбои и предложить рациональный подход, позволяющий создать систему, которую в действительности хочет видеть заказчик. Следовательно, необходимо понимать, что это книга не по программированию и она ориентирована не только на разработчиков программного обеспечения. Это книга об управлении требованиями в сложных программных приложениях. Таким образом, данная книга написана для всех членов команды разработчиков – аналитиков, разработчиков, персонала, проводящего тестирование и обеспечивающего качество, людей, осуществляющих управление проектом, разработчиков документации и т.д., а также для всех из внешней команды "клиентов" – пользователей и других заинтересованных лиц, представителей маркетинга и менеджмента. Иными словами, эта книга предназначена для всех, кто должен принимать участие в решении проблемы требований.

Оказывается, очень важно, чтобы члены обеих команд, в том числе и не имеющие отношения к технике члены внешней команды, овладели навыками, необходимыми для успешного осуществления процесса определения требований к новой системе и управления ими, хотя бы потому, что именно они изначально создают требования, а в итоге определяют успех (или неуспех) системы. Работающий в одиночку герой-программист – анахронизм, оставшийся в прошлом.

Надеюсь, никто не будет возражать, что при строительстве дома необходимо не раз посоветоваться с его владельцем, иначе можно построить дом с двумя спальнями, в то

время как заказчик хотел, чтобы их было три. Но не менее важно обсудить и согласовать "требования" с властями относительно того, что касается строительных норм и регулирования застройки в данном районе; также может понадобиться проконсультироваться с ближайшими соседями, прежде чем вырубить некоторые деревья на участке, где будет расположен дом.

Испектор по строительству и ближайшие соседи являются теми заинтересованными лицами, которые вместе с владельцем этого дома будут определять, удовлетворяет ли построенный дом всем требованиям. Понятно, что многие важные заинтересованные лица, такие как соседи и представители власти, в данной ситуации не являются пользователями (владельцами дома), также очевидно, что их представления о том, что такое качественный дом, могут существенно отличаться.



В данной книге обсуждаются программные приложения, а не дома или камни. Предъявляемые к дому требования можно, по крайней мере частично, описать посредством ряда чертежей и инженерных схем. Аналогично систему программного обеспечения можно описать с помощью моделей и диаграмм. Чертежи дома служат средством общения и переговоров между инженерами и непрофессионалами (юристами, инспекторами и соседями), точно так же связанные с программной системой технические диаграммы можно создавать так, чтобы "обычные" люди могли их понимать.

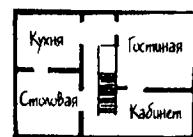
Для многих чрезвычайно важных требований диаграммы не нужны вовсе. Например, будущий владелец дома может просто написать (например, по-английски) требование, которое гласит: "В моем доме должны быть три спальни и достаточно большой гараж для двух машин и шести велосипедов". Как будет показано в этой книге, большинство критически важных требований к программной системе также можно написать с помощью естественного языка.

Многие *профессиональные приемы*, которыми необходимо овладеть для решения данной проблемы, также можно описать посредством практических советов на уровне здравого смысла. Строителю-новичку можно посоветовать: "Не забудьте побеседовать с инспектором по строительству до того, как начнете рыть котлован под фундамент, а не после того, как зальете его раствором и начнете возведение стен". В программном проекте уместен аналогичный совет: "Не забудьте задать правильные вопросы, определить приоритеты требований и *не позволяйте* заказчику говорить вам, что все 100% требований критически важны, поскольку вряд ли вы сумеете удовлетворить их все к моменту сдачи".

О данной книге

В книге Леффингвэлла (Leffingwell) и Уидрига (Widrig) выбран прагматический подход к описанию решения "проблемы камня". Книга состоит из введения и шести частей, соответствующих шести группам профессиональных приемов в сфере разработки требований. Во введении (главы 1–3) предлагаются определения и основные положения, необходимые для понимания последующего материала. В главе 1 содержится обзор возникающих при разработке систем проблем. Данные свидетельствуют, что некоторые проекты разработки программного обеспечения действительно потерпели неудачу из-за небрежного программирования, но последние исследования достаточно убедительно показали, что плохое управление требованиями само по себе может быть основной причиной провала проекта. В данном предисловии основные положения управления требованиями описаны свободно и неформально, в главе 2 авторы определяют их более тщательно, создавая основу для последующих глав. В главе 3 содержится краткое описание структуры команд, занимающихся разработкой программного обеспечения, чтобы можно было осуществить привязку разрабатываемых профессиональных приемов к среде, в которой они будут применяться.

В структуре книги выделяется шесть частей, описывающих шесть необходимых команд для эффективного управления требованиями профессиональных навыков.



Каждая из перечисленных ниже шести основных частей ориентирована на то, чтобы помочь вам и вашей группе овладеть одним из *шести необходимых для успешного управления требованиями профессиональных навыков*.

- Для начала, конечно, необходимо правильное понимание проблемы, решить которую призвана новая система программного обеспечения. Этому посвящена часть 1, "Анализ проблемы".
- Часть 2, "Понимание потребностей пользователей", также очень важна. Профессиональные приемы в этой области образуют основу данной части.
- Часть 3, "Определение системы", описывает процесс создания исходного определения системы, призванной удовлетворить эти потребности.
- В части 4, "Управление масштабом", рассматривается чрезвычайно важный и часто игнорируемый процесс управления масштабом проекта.
- Часть 5, "Уточнение определения системы", иллюстрирует основные методы, используемые для доработки системы до уровня детализации, достаточного для проведения проектирования и реализации, чтобы вся расширенная команда в точности знала, какая система создается.
- В части 6, "Построение правильной системы", обсуждается процесс построения действительно удовлетворяющей требованиям системы. Здесь также рассматриваются методы, позволяющие проверить, что система удовлетворяет поставленным требованиям, не наносит никакого вреда своим пользователям и не имеет никаких неприятных свойств, не оговоренных требованиями. Так как требования к любой нетривиальной системе не могут быть "заморожены" во времени, авторы описывают способы, позволяющие команде успешно справляться с изменениями, не разрушая созданную систему.

Наконец, после краткого подведения итогов авторы предлагают рецепт, которым вы можете воспользоваться для управления требованиями в вашем следующем проекте. Заканчивается книга главой 35, "С чего начать".

Мы надеемся, что, вооружившись новыми профессиональными приемами, вы сможете создать продукт, достойный восхищения. Но это будет и легко. Даже если в вашем распоряжении имеются наилучшие методы и процессы, а также поддерживающие их автоматические средства, вы все равно увидите, что это сложная работа. Кроме того, она связана с риском; даже при использовании содержащихся в книге полезных советов, некоторые проекты будут заканчиваться неудачей, так как мы пытаемся создать как можно более сложные системы в наиболее короткое время. Но описанные в данной книге профессиональные приемы помогут значительно снизить риск и таким образом достигнуть успеха в вашем предприятии.

Эд Йордан

Предисловие автора

Дин Леффингуэлл

Источники предлагаемых методов

Предлагаемый в данной книге материал представляет собой опыт, накопленный множеством людей, занимавшихся определением, разработкой и распространением систем программного обеспечения мирового класса. Эта книга не является академическим исследованием, посвященным управлению требованиями. На протяжении 80-х годов мы с Доном Уидригом (Don Widrig) входили в руководство небольшой компании, занимавшейся созданием программного обеспечения для заказчиков. Разрабатывая многие представленные в книге приемы управления требованиями, мы старались сосредоточить внимание на тех из них, которые важны как для создания систем программного обеспечения, так и с точки зрения результатов, которые необходимо представить заинтересованным лицам. Поскольку функционирование программного обеспечения исключительно важно для успешной работы предприятия в целом, мы старались не отвлекаться на мелочи, избегать предвзятых мнений и экспериментов с непроверенными методами.

За последние десять лет методы претерпели эволюцию. Они были усовершенствованы в результате приобретенного в различных компаниях при разнообразных обстоятельствах опыта. Однако все представленные методы проверены на практике и прошли испытание временем. Возможно, даже более важным является то, что на них практически не повлияли произошедшие за этот период в отрасли технологические изменения. Действительно, большая часть изложенных в книге принципов не зависит от изменений в технологиях программного обеспечения. Поэтому мы надеемся, что полученные знания не утратят со временем своего значения.

Уроки работы с требованиями, извлеченные из создания систем программного обеспечения по заказу других пользователей

Поначалу я возненавидел компьютеры. ("Что? Я проторчал здесь всю ночь и должен спускать этот накет, так как оставил лишний пробел? Вы сошли с ума?") Моим первым "настоящим компьютером" был мини-компьютер, который, хотя и имел чрезвычайно ограниченные возможности по современным меркам, был уписан в том смысле, что я мог его трогать, программировать и заставлять его делать все, что я хотел. Это был мой компьютер.

Мои первые исследования касались применения компьютеров к анализу физиологических сигналов человеческого тела, главным образом, электрокардиограмм, и мой компьютер был прекрасным средством для этих целей. Помимо этого я начал применять

свои программистские навыки и опыт в системах реального времени, работающих в промышленности.

Наконец, я вошел в корпорацию RELA и начал долгую и временами невероятно сложную карьеру исполнительного директора по разработке контрактного программного обеспечения.

RELA Мой соавтор, Дин Уидриг (Din Widrig), вскоре присоединился ко мне в RELA в качестве вице-президента отдела исследований и разработок. Он внес неоценимый вклад в успех многих разработанных нами систем.

С годами компания быстро расширялась. В настоящее время в ней работают сотни людей, и она перешла от разработки простых систем программного обеспечения к созданию полнофункциональных сложных медицинских приборов и систем, которые наряду с программным обеспечением включают в себя механические, электронные и оптические системы. Однако в сердце любой машины, в том числе и самой современной клинической лаборатории для ДНК-диагностики, находится один или несколько компьютеров, которые постоянно надежно выполняют свои функции в заданном ритме в многозадачной системе реального времени.

Первоначально мы программировали все и для всех, от программного обеспечения для позиционирования антенн до игр с лазерными мишенями, автоматически управляемых движущихся аттракционов, обучающих систем, сварочных роботов и средств автоматизированного управления механизмами. Мы даже разработали крупную распределенную компьютерную систему, которая автоматически выявляла и подсчитывала количество рекламных роликов на телевидении. (Наш девиз выглядел так: "Мы делаем компьютеры, которые будут смотреть рекламу вместо вас!") Возможно, единственной объединяющей особенностью разрабатываемого тогда программного обеспечения было то, что оно разрабатывалось для других, т. е. мы не были экспертами в данной области и не могли покрывать расходы по своему усмотрению. Мы полностью зависели от того, насколько заказчик будет удовлетворен конечным результатом работы. Такие условия во многом благоприятствовали формированию эффективного управления требованиями. И вот почему.

- Мы мало знали о предметной области, поэтому зависели от клиента при определении требований. У нас не было искушения выработать их самостоятельно; нам приходилось спрашивать, и мы должны были учиться задавать нужные вопросы в пушное время.
- Наши клиенты зачастую плохо разбирались в компьютерах, поэтому они зависели от нас в том, что касалось преобразования их потребностей и пожеланий в техническое задание.
- То, что работа оплачивалась, придавало строгость отношениям между разработчиком и клиентом.
- Качество было легко измерить: нам или платили, или нет.

Основной вопрос 1. "Что же на самом деле должна делать программа?"

Именно в этой ситуации мы открыли для себя два основных вопроса, с которыми сталкиваются разработчики программного обеспечения во всех проектах. Первый из них руководил нашим поведением многие годы и по сей день остается, пожалуй, сложнейшим вопросом, на который необходимо ответить в любом проекте разработки программного обеспечения.

Что же на самом деле должна делать программа?

Именно для ответа на этот вопрос мы на протяжении более 10 лет разрабатывали принципы и методы, представленные в частях 1, "Анализ проблемы", 2, "Понимание потребностей пользователя", и 3, "Определение системы". Каждый из этих методов доказал свою полезность и продемонстрировал эффективность во многих реальных проектах. Именно в этот период я также впервые познакомился с работами Дональда Гауса (Donald Gause) и Джерри Вайнберга (Jerry Weinberg), в частности с их книгой *Exploring Requirements: Quality Before Design* (1989). Поскольку она значительно повлияла на нашу работу, мы использовали некоторые ее основные концепции в нашей книге; во-первых, потому, что они работают, а во-вторых, будет хорошо, если вы также воспользуетесь опытом Гауса и Вайнберга.

Уроки, извлеченные из создания высоконадежных систем

Спустя некоторое время корпорация RELA стала специализироваться на разработке различных медицинских приборов и систем, использующих компьютеры: систем искусственной вентиляции легких, инфузионных насосов, водителей ритма, клинических диагностических систем, систем искусственного кровообращения, оборудования, контролирующего состояние пациента, и других диагностических и терапевтических приборов.

Именно в начале работы над проектом системы искусственной вентиляции мы осознали, какая ответственность возложена на нас: *Если мы перекроем кран, кто-то умрет!* В первую очередь мы думали о пациенте; его жизнь зависела от прибора, для которого мы создавали одну из первых в мире жестко ограниченных по срокам и средствам систем программного обеспечения. (Представьте себе ответственность первопроходца. Вы – первый!)

Понятно, что подобное задание, когда ставки столь высоки, заставило нас очень серьезно подойти к разработке программного обеспечения на самых ранних этапах развития индустрии разработки встроенных систем. Очень быстро выяснилось, что для стабильного успеха необходимо сочетание следующих компонентов.

- Прагматический процесс определения требований к программному обеспечению и управления ими.
- Жесткая методология проектирования и разработки программного обеспечения.
- Применение разнообразных проверенных современных методов верификации и проверки того, что программное обеспечение является правильным и безопасным.
- Высочайшая квалификация и ответственность как команды разработчиков, так и группы, гарантирующей качество программной системы.

Я твердо верил в то время и еще более убежден сегодня, что *все эти элементы необходимы для создания любой более-менее надежной программной системы значительного объема*. В корпорации RELA это был единственный способ обеспечить безопасность пациентов, выживание нашей компании и экономическое будущее ее служащих.

Основной вопрос 2. "Как точно узнать, что программа делает именно то, что нужно, и ничего другого?"

Добившись успеха в разработке и применении различных методов для ответа на основной вопрос 1, перейдем ко второму фундаментальному вопросу, с которым постоянно сталкивается команда разработчиков программного обеспечения.

Как можно точно узнать, что программа делает именно то, что нужно, и ничего другого?

Методы, используемые для ответа на данный вопрос, составляют основу части 5, "Уточнение определения системы", и части 6, "Построение правильной системы".

Итак, вы можете быть уверены, что представленные в данной книге приемы проверены временем и хорошо зарекомендовали себя. Даже если вы не связаны с разработкой высокобезопасных систем, увидите, что это полезные практические и эффективные (в плане затрат) советы, которые можно использовать для разработки систем программного обеспечения наивысшего качества.

Хотя методы, которые мы заимствовали, модифицировали, разрабатывали и применяли в корпорации RELA для решения двух основных вопросов, были достаточно эффективны, существовала проблема, которая постоянно держала меня в напряжении во время наиболее сложных моментов этих проектов.

При том, что значительная часть действий в процессе определения требований в силу его природы выполняется вручную, сколько времени может пройти, прежде чем мы совершим единственную, но потенциально опасную ошибку?

Существовал также вопрос затрат, так как проводимые вручную испытания были дорогостоящими и не исключали возможности ошибок. За этот период механическое конструирование прошло путь от выполняемых вручную чертежей до трехмерных компьютерных систем автоматического проектирования. За то же время наше продвижение в области программирования (в том, что касается практических целей) ограничилось повышением уровня абстракции наших языков программирования, что, конечно же, хорошо, но такие показатели, как частота ошибок, производительность написания строк кода, а также уровень качества, остались относительно неизменными. Проводимые нами в этот период эксперименты с CASE-средствами принесли неоднозначные результаты. Говоря откровенно, как разработчик программного обеспечения и руководитель, я считал состояние дел в отрасли инженерии программного обеспечения непростым.

Безусловно, автоматизация никогда не устранит потребность в необходимых для разработки программ мыслительных способностях. Тем не менее я согласен с тем, что автоматизация некоторых проводимых вручную операций записи и внесения изменений может высвободить дефицитные ресурсы, чтобы направить их на более продуктивную деятельность. Ну и, конечно, мы ожидали, что затраты на разработку снизятся, а надежность возрастет.

Уроки, извлеченные из деятельности в сфере управления требованиями

В 1993 году была образована корпорация REQUISITE, и некоторых из нас пригласили принять участие в разработке и внедрении нового инструментального средства разработки требований, RequisitePro. Поскольку в это время мы постоянно помогали пользователям в том, что касалось проблем управления требованиями, появилось много дополнительного материала для данной книги. Мы обязаны значительной частью этой работы клиентам корпорации, а также клиентам компании RELA, у которых многому научились.



В это время на мою карьеру большое влияние оказал д-р Аллан Дэвис (Alan Davis), главный редактор журнала IEEE Software и заведующий лабораторней программирования им. Эла Помара (El Pomar) университета Колорадо в Колорадо Спрингс. В самом начале он вошел в компанию как директор и советник и в этом качестве оказывал заметное воздействие на нашу технологию и направления развития компании. Д-р Дэвис хорошо известен как специалист в области разработки требований. Он часто выступает в роли консультанта и разработал множество методов, помогающих компаниям усовершенствовать процесс управления требованиями. Эти методы были объединены с некоторыми методами, разработанными в RELA, и стали основой курса профессиональной подготовки "Requirements College", а также некоторых частей данной книги.

Кроме того, следуя недостаточно популярной бизнес-теории, согласно которой *профессиональной помощи никогда не бывает слишком много*, мы пригласили в компанию известного эксперта и автора книг о программном обеспечении Эда Йордана (Ed Yourdan). Он также оказал большое влияние на выработку курса в том, что касалось технологии и бизнес-направлений. Эд и Аллан стояли у истоков этой работы, и многие приводимые в книге высказывания принадлежат им. На самом деле мы хотели выпустить эту книгу совместно с ними несколько лет назад. Но времена меняются, и они любезно предоставили нам возможность распоряжаться их результатами. Поэтому вы часто будете ощущать их присутствие в данной книге.

Опыт, приобретенный в корпорации Rational Software

В 1997 году корпорация Rational Software купила компанию Requisite. В Rational мы приобрели значительный опыт в управлении требованиями применительно к разработке и реализации полного спектра средств разработки приложений, по-прежнему продолжая помогать пользователям в решении их проблем при работе с требованиями. Доп продолжает сотрудничать с нами в вопросах совершенствования методов. Кроме того, в компании Rational я имел прекрасную возможность работать с такими экспертами в области программного обеспечения и авторами популярных книг, как Грейди Буч (Grady Booch), Ивар Джейкобсон (Ivar Jacobson), Джеймс Рамбо (James Rumbaugh), Уолкер Ройс (Walker Royce) и Филипп Крачтен (Philippe Kruchten). Все они оказали влияние на мое понимание проблемы управления требованиями, а Уолкер и Филипп были первыми читателями этой книги.

Rational
uniting software teams

Мы стали обращаться к методу прецедентов для фиксации требований, а также использовать прецеденты в модели проектирования для того, чтобы обеспечить общую логическую связь при создании архитектуры, проведении реализации и тестирования.

Я также являюсь сторонником принятого в компании Rational *информационного подхода* к разработке программного обеспечения, и мне хочется думать, что как в RELA мы в свое время были первопроходцами, так это происходит и теперь с *Rational Unified Process* — процессом, описывающим полный жизненный цикл разработки программного обеспечения.

Компания Rational помогла мне завершить данную работу, и я благодарен ей за это. Она также любезно разрешила использовать некоторые идеи, тексты и диаграммы.

Наконец, мы хотим поблагодарить всех, кто принимал участие в работе над книгой, в том числе Алана Дэвиса (Alan Davis), Эда Йордана (Ed Yourdan), Грейди Буча (Grady

Booch), Филиппа Крачтена (Philippe Kruchten), Лесли Пробаско (Leslee Probasco), Яна Спенса (Ian Spence), Джин Белл (Jean Bell), Уолкера Ройса (Walker Royce), Джо Мараско (Joe Marasco), Элмера Мэгэзинера (Elmer Magaziner), а также рецензентов издательства Addison-Wesley: Эга Марсония (Ag Marsonia), Грэнвилла Миллера (Granville Miller), Фрэнка Армура (Frank Armour), д-ра Ральфа Р. Янга (Ralf R. Young) (директора по разработке программного обеспечения компании Litton PRC Systems and Process Engineering), профессора Дэвида Райна (David Rine) (Университет Джорджа Мэйсона) и Дэна Роусторна (Dan Rawsthorn) (ACCESS).

Заключение

По сути, большинство идей этой книги не являются оригинальными. Они представляют собой результат обобщения опыта двух десятилетий совместной работы в сфере разработки программного обеспечения, в которой пристальное внимание уделялось целостному подходу к проблеме требований. Мы полагаем, что нам удалось объединить лучшие идеи и практические приемы в этой чрезвычайно важной и сложной области. Мы верим, что результат — *шесть необходимых команд для эффективного управления требованиями наборов профессиональных приемов* — поможет вам создать качественную систему программного обеспечения, уложившись в отведенные сроки и бюджет.

Введение

- Глава 1. Проблема требований
- Глава 2. Введение в управление требованиями
- Глава 3. Команда разработчиков



Поначалу кажется, что это совсем несложно

Садимся вместе с заказчиком. Записываем, что, по его мнению, должна делать система. Используем "классные" новые языки программирования и средства разработки программного обеспечения, которых пару лет назад еще не было. Создаем приложение, применяя эти самые современные языки и средства разработки. Быстро и эффективно проводим имитационные эксперименты и отладку. Производим загрузку нового клиентского приложения с удаленного компьютера. Сидим и ждем вознаграждения. Все идет просто замечательно. Итог – отличные премиальные!

Реальное положение вещей выглядит совсем иначе

Однако ситуация, как правило, оказывается совсем иной. В нашей жизни преобладают работа по ночам, меняющиеся требования, непостоянство заказчиков, серьезные проблемы с качеством программного обеспечения, технология, которая устаревает прежде, чем ее успевают применить, значительные задержки в реализации проекта и невыполненные обязательства. В лучшем случае наши заказчики довольны, и мы получаем хорошее вознаграждение. Но даже тогда это нам дорого обходится, и мы уверены, что могли сделать лучше. В худшем случае мы получаем остановленные проекты и сплошное разочарование. Давайте следующий проект! Слава Богу, мы любим этот бизнес!

Темы, рассматриваемые во введении

В главе 1 вводится понятие управления требованиями, а также описываются некоторые основные проблемы разработки программного обеспечения, обуславливающие успешное или неуспешное завершение проекта. В этой главе также содержатся рекомендации, на что стоит тратить время и ресурсы при разработке требований к приложению. Если вы – опытный разработчик программного обеспечения или имеете опыт руководства проектами либо работали в любой другой сфере программирования и принимали участие во многих сложных проектах, вы можете пропустить эту главу и сразу перейти к главе 2.

Но если вы новичок или по роду своей деятельности далеки от сферы разработки программного обеспечения вашей компании (например вы – сотрудник отдела маркетинга и вам поручен выбор нового программного обеспечения или вы работаете в отделе обеспечения качества и вам нужно пройти через процесс аккредитации компании по стандарту ISO 9000 или же вы работаете в "отделе-пользователе", которому для обеспечения его деятельности необходимы информационные системы), вам следует прочитать главу 1, равно как и всю остальную книгу!

Вы, скорее всего, знаете, что проекты разработки систем могут быть сложными, дорогостоящими, а также могут сопровождаться высоким риском и могут быть подвержены сбоям, но вы можете не знать, почему это типично для большинства организаций. Если вы думаете, что только ваша организация испытывает подобные проблемы, вас успокоит вытекающий из приведенной в главе 1 статистики факт, что практически все организации страдают от них. Знание того, почему возникают эти проблемы и где они наиболее острые и дорогостоящие, – первый важнейший шаг на пути совершенствования.

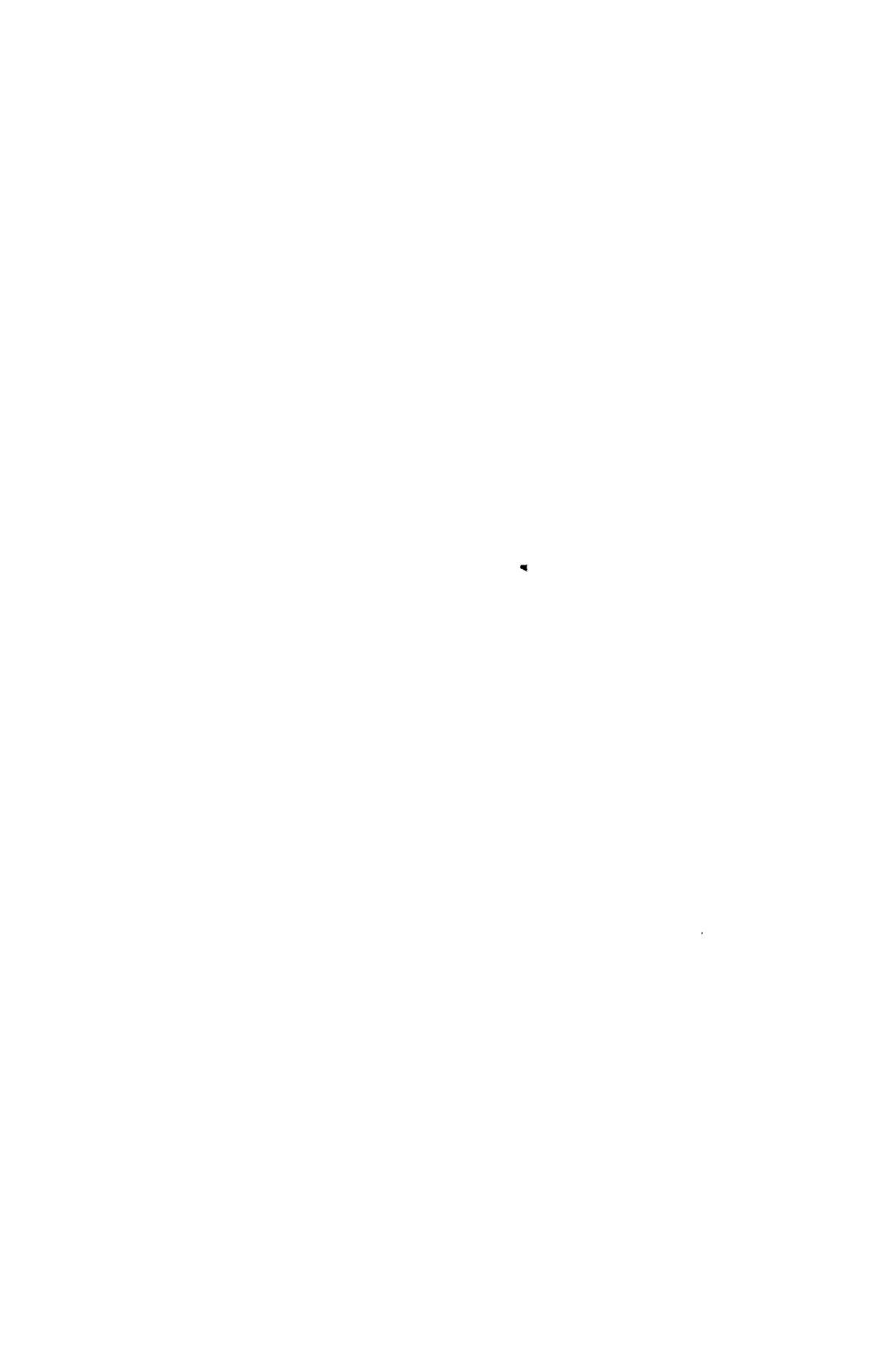
В главе 2 вводятся используемые в управлении требованиями понятия, которые будут обсуждаться на протяжении всей книги. Даже если вам кажется, что вы знаете, что такое

“требования” — и действительно, кто *не знает* этого? — мы рекомендуем прочитать этот материал, так как в нем содержатся некоторые определения и общие предположения, которые используются в дальнейшем.

Наконец, глава 3 посвящена организации команды создателей программного обеспечения.

Успешное управление требованиями может осуществляться только хорошо организованной командой создателей программного обеспечения.

Однако данная книга не о командах, и рассмотрение таких важных тем, как формирование высокопроизводительных команд, их мотивация и управление ими в процессе разработки программного обеспечения, не входит в нашу задачу. Книга посвящена управлению требованиями к программному обеспечению, а для успешного осуществления этой задачи необходима поддержка всей команды создателей программного обеспечения. Дело в том, что, возможно, управление требованиями в большей степени, чем любая другая деятельность в рамках разработки программного обеспечения, затрагивает каждого члена как основной, так и расширенной команды, в которую входят пользователи и другие заинтересованные лица. Мы исходим из того, что во всех проектах, за исключением самых тривиальных, *успешное управление требованиями может осуществляться только хорошо организованной командой создателей программного обеспечения*. Более того, чтобы добиться успеха, каждый член команды должен тем или иным образом участвовать в процессе управления требованиями и знать о своей роли в нем.



Глава 1

Проблема требований

Основные положения

- Цель разработки программного обеспечения состоит в том, чтобы, уложившись в отведенное время и бюджет, разработать качественное программное обеспечение, удовлетворяющее реальные потребности пользователей.
- Успех проекта зависит от хорошо организованного управления требованиями.
- Ошибки в требованиях – наиболее часто встречающийся тип ошибок при разработке систем, а их устранение является наиболее дорогостоящим.
- Использование нескольких ключевых приемов может значительно снизить вероятность ошибок в требованиях и, следовательно, повысить качество программного обеспечения.

Цель

В настоящее время тысячи команд разработчиков программного обеспечения заняты разработкой самых разнообразных программных приложений в различных отраслях. Но, работая в разных отраслях и говоря на разных языках, все они работают с одними и теми же известными технологиями, читают одни и те же журналы, учились в одних и тех же школах и, к счастью, имеют одну ясную цель: *разработать удовлетворяющее реальные потребности клиента качественное программное обеспечение, уложившись в отведенное время и бюджет*.

Клиенты могут быть достаточно разными...

- Для некоторых из нас клиент является некой внешней сущностью, источником заказов, и его нужно убедить отклонить предложения наших конкурентов и приобрести наш готовый программный продукт, поскольку он проще в использовании, имеет большие функциональных возможностей и, в конечном счете, лучше по всем параметрам.
- Для других клиентом является нацившая их для разработки своего программного обеспечения компания, которая ожидает, что программное обеспечение будет наивысшего качества (возможного при современном состоянии дел) и сделает компанию более конкурентоспособной и более прибыльной.
- Для большинства же клиент работает вместе с нами, сидит дальше по коридору, этажом ниже или где-то в другом месте и с нетерпением ожидает нового приложения, позволяющего более эффективно обрабатывать заказы или использовать средства электронной коммерции для продажи товаров и услуг, чтобы компания, и

которой мы вместе работаем, стала более прибыльной, а наша работа лучше оплачивалась и приносила большее удовлетворение.

Таким образом, хотя клиенты и различны, цель у всех одна.

Немного статистики

Необходимо признать, что при разработке нетривиальных программных систем картина получается достаточно пестрой. Безусловно, некоторые системы работают достаточно хорошо, и непрофессионалы, равно как и ветераны, поражаются тому, что удалось сделать: Internet, интерфейсы пользователя, карманные вычислительные устройства, интеллектуальные приборы, управление процессами в реальном времени, интерактивное ведение счетов клиентов брокерских фирм и т.д. Также верно и то, что многие работающие системы весьма несовершены. Например, текстовый процессор и операционная система персонального компьютера, которые авторы использовали при написании данной книги, вместе вызывали примерно два системных сбоя в день при создании этой главы, а также демонстрировали множество других неприятных сюрпризов и досадных мелких неточностей. Конечно же, их нельзя считать примерами совершенного программного обеспечения, однако они были "достаточно хороши", чтобы с их помощью написать данную главу.

Но зачастую ситуация более серьезна. Исследования группы Стендиша (Standish Group) (1994) свидетельствуют о следующем.

В США ежегодно тратится более 250 миллиардов долларов на разработку приложений информационных технологий в рамках примерно 175000 проектов. Средняя стоимость проекта для крупной компании составляет \$2322000, для средней — \$1331000, а для мелкой — \$434000...

Исследования группы Стендиша показали, что 31% проектов будет прекращен до завершения. Затраты на 52,7% проектов составят 189% от первоначальной оценки...

*Исходя из этого, группа Стендиша оценивает, что... американские компании и правительственные учреждения потратят 81 миллиард долларов на программные проекты, которые так и не будут завершены. Эти же организации заплатят дополнительно 59 миллиардов долларов за программные проекты, которые хотя и завершатся, но значительно превысят первоначально отведенное на них время.*¹

В целом к таким данным принято относиться с некоторой долей скептицизма, но каждый из работающих в данной отрасли может легко привести примеры подобного рода. У каждого из нас есть любимый проект, который так и не вышел на рынок, или информационная система, своего рода "смолянная яма", которую мы создали и до сих пор страдаем от этого. Таким образом, если есть очевидные эмпирические доказательства того, что проблема существует, и эти доказательства подтверждаются нашим собственным опытом, лучшее, что можно сделать, — признать ее существование и попытаться ее решить. Не так ли?

¹ Используется с разрешения.

Основные причины успеха и провала проекта

Первым шагом на пути решения любой проблемы является *осознание основных причин ее возникновения*. К счастью, в своем обзоре группа Стендиша не ограничилась цифрами и предложила своим респондентам указать наиболее значительные факторы, повлиявшие на "успешные", "проблемные" (с которыми возникла задержка или не оправдавшие ожиданий) и "потерпевшие неудачу" (прекрашенные) проекты.

И тогда оказалось, что не случайно в данной книге такое пристальное внимание уделяется вопросу требований; многие наиболее часто встречающиеся серьезные проблемы при разработке программного обеспечения связаны именно с требованиями. В отчете группы Стендиша (1994) указано три наиболее часто встречающихся ключевых фактора, создающих "проблемы" в проектах.

- Недостаток исходной информации от клиента: 13% всех проектов
- Неполные требования и спецификации: 12% проектов
- Изменение требований и спецификаций: 12% всех проектов

В остальном данные сильно расходятся. Конечно, проект может потерпеть неудачу из-за нереалистичного подхода к составлению графика или выделению времени (эта причина указана для 4% проектов), неправильного подбора персонала и выделения ресурсов (6%), несоответствия технологических навыков (7%) и по другим причинам. Но если считать, что приведенные группой Стендиша цифры представляют реальное положение дел в отрасли, то, по крайней мере, третья часть проектов испытывает проблемы из-за причин, непосредственно связанных со сбором и документированием требований, а также с управлением ими.

Хотя большинство проектов действительно превышают отведенное время и бюджет, если их вообще удается закончить, группа Стендиша обнаружила, что около 9% проектов крупных компаний были завершены вовремя и в пределах бюджета; аналогичного успеха удалось достигнуть в 16% проектов мелких компаний. Возникает очевидный вопрос: *Каковы главные "факторы успеха" в этих проектах?* Согласно проведенному исследованию тремя наиболее важными факторами были следующие.

- Подключение к разработке пользователя: 16% всех успешных проектов
- Поддержка со стороны исполнительного руководства: 14% всех успешных проектов
- Ясная постановка требований: 12% всех успешных проектов

Другие источники приводят даже более впечатляющие результаты. Например, организация ESPITI (European Software Process Improvement Training Initiative – Европейская инициатива по обучению совершенствованию процесса программирования) провела в 1995 году опрос с целью определить относительную важность различных типов существующих в отрасли проблем. Результаты этого широкого опроса, в котором приняли участие 3800 респондентов, отражены на рис. 1.1.

Двумя самыми главными проблемами, упоминавшимися почти в половине ответов, оказались следующие.

1. Спецификации требований
2. Управление требованиями клиента

В подтверждение результатов опроса группы Стендиша, вопросы собственно написания кода вновь оказались относительно "непроблемными".

Из сказанного следует, что требования являются одной из основных причин возникновения проблем при разработке программного обеспечения. Рассмотрим экономические факторы, связанные с этой конкретной причиной.

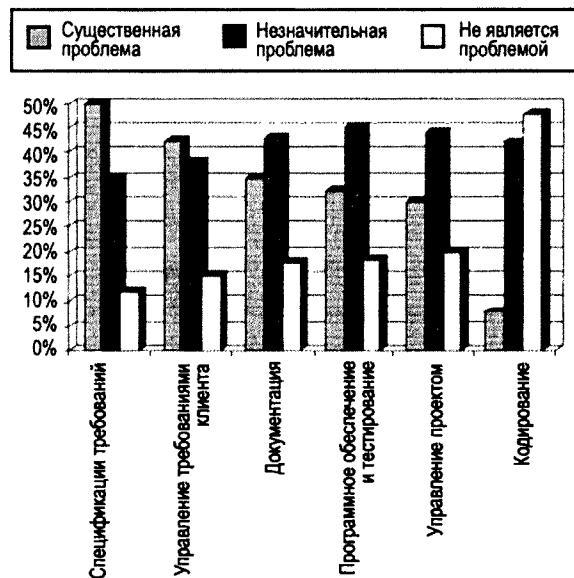


Рис. 1.1. Основные типы проблем, возникающих при разработке программного обеспечения

Частота возникновения ошибок, связанных с требованиями

В исследованиях как группы Стендиша, так и ESPITI содержатся качественные данные: респонденты считают, что проблемы требований, судя по всему, превосходят остальные в плане риска неполадок, которые они вызывают при разработке приложений. Но действительно ли проблемы требований влияют на итоговый код?

В табл. 1.1 представлены результаты исследования, проведенного в 1994 году Каперсом Джонсом (Capers Jones), и приводятся данные о вероятном количестве возможных дефектов и средней эффективности их устранения организацией-разработчиком посредством различных комбинаций тестирования, контроля и других методов.

Таблица 1.1. Краткая характеристика недоработок

Источники дефектов	Возможность возникновения	Эффективность устранения	Оставшиеся дефекты
Требования	1.00	77%	0.23
Проектирование	1.25	85%	0.19
Кодирование	1.75	95%	0.09

Окончание табл. 1.1

Источники дефектов	Возможность возникновения	Эффективность устранения	Оставшиеся дефекты
Документация	0.60	80%	0.12
Неправильная установка	0.40	70%	0.12
В целом	5.00	85%	0.75

В колонке "Возможность возникновения" произведена нормализация в соответствии с тем, какова доля каждой категории, если общее количество дефектов принять за 5.00. Это число выбрано произвольно и ничего не говорит об абсолютном количестве недоработок. Колонка "Оставшиеся дефекты", относящаяся к тому, что видит в конечном итоге пользователь, нормализована аналогичным образом.

Ошибки требований занимают первое место среди оставшихся недоработок и составляют примерно одну треть всех неустранимых дефектов. Таким образом, это исследование служит доказательством того, что ошибки требований – наиболее распространенная категория ошибок при разработке систем.

Высокая цена ошибок требований

Если бы ошибки требований можно было быстро, просто и экономно устранять, проблема не была бы столь серьезна. Но приводимая ниже статистика не оставляет никаких надежд. Все обстоит иначе. В проведенных в компаниях GTE, TRW, IBM и IIP исследованиях была оценена стоимость ошибок, возникающих на различных стадиях жизненного цикла проекта. Дэвис (Davis) в своей работе (1993) подвел итоги нескольких подобных исследований. Результаты представлены на рис. 1.2. Хотя эти исследования проводились независимо, они имели приблизительно одинаковые результаты: если стоимость усилий, необходимых для обнаружения и устранения ошибки на стадии написания кода, принять за единицу, то стоимость выявления и устранения ошибки на стадии выработки требований будет в пять–десять раз меньше. А стоимость обнаружения и устранения ошибки на стадии сопровождения – в 20 раз больше.

Из рисунка видно, что при обнаружении ошибок на стадии формирования требований получаем экономию средств в соотношении 200:1 по сравнению с их обнаружением на стадии сопровождения жизненного цикла программы.

Хотя это может быть преувеличением, легко заметить, что в данном случае проявляется эффект умножения. Причина состоит в том, что многие из этих ошибок достаточно долго остаются не обнаруженными.

При внимательном чтении этого раздела можно заметить, что на рис. 1.2 не вполне корректно объединены два аспекта: относительная стоимость различных категорий ошибок и стоимость их устранения на различных стадиях жизненного цикла программного обеспечения. Например, элемент "период формирования требований" фактически подразумевает все ошибки, обнаруженные и устранившие во время, официально обозначенное как "период формирования требований". Но поскольку маловероятно, что техническое проектирование или программирование будут производиться на столь ранней стадии (исключая возможные действия по созданию прототипа), ошибки, которые обнаруживаются и устраняются на этой стадии, действительно являются ошибками требований.



Рис. 1.2. Относительная стоимость устранения ошибки в различных фазах жизненного цикла

Однако ошибки, обнаруженные на стадии проектирования, могут относиться к одной из двух категорий: (1) ошибки, возникшие при создании технического проекта из правильного множества требований или (2) ошибки, которые следовало обнаружить ранее как ошибки требований, каким-то образом “просочившиеся” на фазу проектирования. Именно вторая категория ошибок особенно дорого обходится по двум причинам.

1. Ко времени обнаружения ошибки в требованиях группа разработчиков уже могла потратить время и усилия на создание проекта по этим ошибочным требованиям. В результате проект, вероятно, придется отбросить или пересмотреть.
2. Истинная природа ошибки может быть замаскирована; при проведении тестирования и проверок на данной стадии все думают, что имеют дело с ошибками проектирования, и значительное время и усилия могут быть потрачены впустую, пока кто-нибудь не скажет: “Минуточку! Это вовсе не ошибка проектирования; мы получили неверные требования”.

Для уточнения, в чем же состоит ошибка требований, необходимо общение с клиентом, который участвовал в его разработке на самом первом этапе. Но может оказаться, что в данный момент связаться с этим человеком невозможно или он забыл, в чем состояла суть инструкции, которую он давал команде разработчиков, либо не помнит, чем руководствовался при этом, или же просто переменил свою точку зрения. Аналогично может оказаться, что принимавший участие в этом этапе член команды разработчиков (часто такой человек имеет статус “бизнес-аналитика” или “системного аналитика”) перешел в подразделение, занятое другим проектом, или также страдает кратковременной потерей памяти. В результате определенная часть работы проделывается вхолостую, и это приводит к потере времени.

Проблемы, связанные с “просачиванием” недостатков из одного этапа в другой, являются вполне очевидными, если о них задуматься, однако подавляющее большинство орга-

низаций ими всерьез не занималось. Одной из организаций, действительно работавшей над данным вопросом, является компания Hughes Aircraft; исследование Снайдера (Snyder) (1992) обнаружило явление "просачивания" дефектов во многих проектах, выполненных компанией на протяжении 15 лет. В данном исследовании говорится о том, что 74% дефектов, связанных с формированием требований, были обнаружены на этапе анализа требований, когда клиенты совместно с системными аналитиками обсуждают, подвергают мозговому штурму, согласовывают и документируют требования к проекту. Это – идеальное время (и обстановка) для обнаружения таких ошибок с наименьшими затратами. Однако исследование также свидетельствует, что 4 процента дефектов "просачиваются" на этап предварительного (высокоуровневого) проектирования, а 7 процентов – еще дальше, на этап детализированного проектирования. Такое "просачивание" движется все дальше и дальше по жизненному циклу системы, и 4 процента ошибок требований оказываются не найденными вплоть до этапа сопровождения, когда система уже передана клиентам и считается полностью работоспособной.

Таким образом, в зависимости от того, где и когда при работе над проектом разработки программного приложения был обнаружен дефект, цена его может разниться в 50–100 раз. Причина состоит в том, что для его исправления придется затратить средства на некоторые (или все) ниже перечисленные действия.

- Повторная спецификация
- Повторное проектирование
- Повторное кодирование
- Повторное тестирование
- Замена заказа – сообщить клиентам и операторам о необходимости заменить дефектную версию исправленной
- Внесение исправлений – выявить и устраниить все неточности, вызванные неправильным функционированием ошибочно специфицированной системы, что может потребовать выплаты определенных сумм возмущенным клиентам, повторного выполнения определенных вычислительных задач на ЭВМ и т.п.
- Списание той части работы (кода, части проектов и т.п.), которая выполнялась с наилучшими побуждениями, но оказалась ненужной, когда обнаружилось, что все это создавалось на основе неверных требований
- Отозвание дефектных версий встроенного программного обеспечения и соответствующих руководств. Если принять во внимание, что программное обеспечение сегодня встраивается в различные изделия – от наручных часов и микроволновых печей до автомобилей – такая замена может коснуться как этих изделий, так и встроенного в них программного обеспечения
- Выплаты по гарантийным обязательствам
- Ответственность за изделие – если клиент через суд требует возмещения ущерба, причиненного некачественным программным продуктом
- Затраты на обслуживание – представитель компании должен посетить заказчика, чтобы установить новую версию программного обеспечения
- Создание документации

Заключение

Итак, из приведенных данных можно сделать два вывода.

1. Ошибки в определении требований являются наиболее распространенными.
2. Стоимость устранения таких ошибок, как правило, одна из самых высоких.

Ошибки в определении требований приводят к затратам, составляющим 25–40% бюджета проекта разработки в целом.

Учитывая частоту возникновения ошибок в определении требований и эффект умножения стоимостей устранения, легко предсказать, что эти ошибки обусловят большую часть – часто 70% или более – затрат на повторную работу. А поскольку на переделки, как правило, расходуется 30–50% бюджета проекта (Бом (Boehm) и Папаччо (Papaccio), 1988, б), на устранение ошибок в определении требований может быть истрачено 25–40% всего бюджета проекта!

Наш опыт подтверждает эти данные, и это послужило основным побудительным мотивом для написания данной книги. Если, затратив относительно небольшие средства на несколько ключевых приемов, удастся продвинуться в данной области, это позволит сберечь значительные средства, повысить производительность, сэкономить время, не выбиться из графика и, в конце концов, предложить заказчику результаты более высокого качества, не говоря уже о сохранении сил и первов команды разработчиков.

Глава 2

Введение в управление требованиями

Основные положения

- Требование — это возможность, которую должна обеспечивать система.
- Управление требованиями — это процесс систематического выявления, организации и документирования требований к сложной системе.
- *Наша* задача состоит в том, чтобы понимать проблемы заказчиков в их предметной области и на их языке и создавать системы, удовлетворяющие их потребности.
- Функция — это предоставляемое системой обслуживание для удовлетворения одной или нескольких потребностей клиентов.
- Прецедент (вариант использования, use case) описывает последовательность выполняемых системой действий, дающих клиенту некий полезный результат.

Обратившись к приведенным в главе 1 данным, нетрудно понять, почему столь большое внимание уделяется управлению требованиями. Но прежде чем излагать различные методы и стратегии, необходимо привести несколько определений и примеров. Начнем с определения того, что понимается под требованием.

Определения

Что такое требование

Хотя за долгие годы уже появилось множество определений требований к программному обеспечению, вполне приемлемым является следующее определение, предложенное специалистами в области разработки требований Дорфманом (Dorfman) и Тэйлером (Thayer) (1990).

- Некое свойство программного обеспечения, необходимое пользователю для решения проблемы при достижении поставленной цели.
- Некое свойство программного обеспечения, которым должна обладать система или ее компонент, чтобы удовлетворить требования контракта, стандарта, спецификации либо иной формальной документации.

Данное определение кажется несколько расплывчатым, но в дальнейшем мы более подробно рассмотрим входящие в него понятия, поэтому можно считать, что на данный момент оно вполне удовлетворительно.

Что такое управление требованиями

Требования задают возможности, которые должна предоставлять система, так что соответствие или несоответствие некоторому множеству требований часто определяет успех или неудачу проекта. Поэтому имеет смысл узнать, что собой представляют требования, записать их, упорядочить и отслеживать их изменения. Иными словами, определение управления требованиями выглядит следующим образом.

Управление требованиями – это систематический подход к выявлению, организации и документированию требований к системе, а также процесс, в ходе которого вырабатываются и обеспечивается соглашение между заказчиком и выполняющей проект группой по поводу меняющихся требований к системе.

- Любой, кому довелось хотя бы однажды иметь дело со сложными программными системами – в качестве пользователя или разработчика – знает, насколько важным является умение выявить суть требований из общения с пользователями и прочими участниками процесса.
- Учитывая, что системе будут предъявлены сотни, если не тысячи, требований, очень важно организовать их.
- Поскольку невозможно удерживать в памяти более нескольких десятков фактов, для успешного взаимодействия различных участников процесса необходимо обеспечить документирование требований. Требования следует записать так, чтобы они были доступны для ознакомления; это может быть документ, модель, база данных или листок на доске объявлений.

Но как быть с управлением требованиями? Основными факторами в данном случае являются размер проекта и его сложность: никто не будет вспоминать об “управлении” требованиями, если в проекте участвуют два человека и необходимо обеспечить выполнение десяти требований. Однако при проверке 1000 требований (как в небольшом коммерческом программном продукте) или 300 000 требований (как в системе управления самолетом Boeing 777) нам, очевидно, придется столкнуться с задачами организации, определения приоритетов, управления доступом, а также обеспечения ресурсов для выполнения всех этих различных требований.

- Кто из участников команды проектировщиков отвечает за требование, касающееся скорости ветра (№278), и кому из них разрешено вносить в него изменения или вообще удалить его?
- Если в требование №278 вносятся изменения, на какие другие требования это повлияет?
- Как удостовериться в том, что в системе программного обеспечения уже написан код для выполнения требования №278, и какие тестовые примеры из общего набора тестов предназначены для проверки того, что данное требование действительно выполнено?

Именно это, а также многое другое подразумевается под управлением требованиями.

Это не есть что-то совершенно новое, что мы сами придумали. Это – один из основанных на здравом смысле видов деятельности, которые в той или иной форме реализуются большинством организаций-разработчиков. Однако, как правило, эта деятельность не является формальной, и ею занимаются от случая к случаю. В результате некоторые важнейшие виды деятельности могут пропускаться или производиться на скорую руку под давлением обстоятельств или связанных с разработкой политических моментов. Таким образом, под управлением требованиями следует понимать набор организованных, стандартизованных и систематических процессов и методов работы с требованиями в значительном и сложном проекте.

Конечно, мы – не первые, кто предложил идею организованных, формализованных процессов; хорошо известны модель повышения уровня зрелости процессов разработки программного обеспечения Института программирования (SEI-CMM – Software Engineering Institute's Capability Maturity Model), а также набор стандартов управления качеством ISO 9000. Мы кратко изложим подходы, принятые в SEI-CMM и ISO 9000, в Приложении Г.

Применение методов управления требованиями

Типы программных приложений

Ранее мы предложили разбить программные приложения на следующие категории.

- Информационные системы и другие приложения, разработанные для использования внутри компании (такие, как система обработки платежных ведомостей, используемая для расчета выдаваемых на руки сумм). Эта категория является основой индустрии информационных систем/информационных технологий, или IS/IT.
- Программное обеспечение (ПО), разрабатываемое и продаваемое как коммерческий продукт (например, текстовый процессор, которым мы пользовались при написании данной книги). Разрабатывающие такое ПО компании обычно называют независимыми поставщиками программного обеспечения (independent software vendors), или ISV.
- Программное обеспечение компьютеров, встраиваемых в другие приборы, машины или сложные системы (такие, как находящиеся в самолете, о котором мы уже писали, мобильном телефоне, которым только что пользовались, автомобиле, в котором поедем к месту следующей встречи). Программное обеспечение этого типа назовем приложениями для встроенных систем или встроенными приложениями.

Эти три типа приложений существенно отличаются по своей природе. Приложения для больших универсальных ЭВМ могут состоять из 5 000 000 строк на языке СОБОЛ и создаваться на протяжении десяти и более лет командой из 50–100 сотрудников. Приложения Web-клиента могут состоять из 10 000 строк программного кода на языке Java, и их может создать за год команда из одного-двух человек. Наконец, программное приложение для системы учета телефонных разговоров в реальном времени может состоять из 1 000 000 строк, написанных на языке С с целью максимально сократить время выполнения.

Мы позаботимся о том, чтобы методы управления требованиями, представленные в данной книге, можно было применять к любому из этих типов. Многие методы не зависят от типа приложения; для других может потребоваться дополнительная (учитывающая тип приложе-

ния) настройка перед их применением. Для более глубокого понимания проблемы будут приводиться разнообразные примеры, иллюстрирующие применение различных методов.

Применение методов управления требованиями к системам общего вида

Принципы управления требованиями можно также применять при разработке систем общего вида. Большинство предлагаемых в данной книге методов окажется пригодным для управления требованиями при разработке произвольных сложных систем, состоящих из механических, вычислительных и химических подсистем, а также их взаимосвязанных элементов и частей. Понятно, что эта дисциплина носит достаточно общий характер, и необходимо продемонстрировать определенную избирательность, чтобы приведенные сведения оказались полезными для членов команды, разрабатывающей программное обеспечение. Поэтому основное внимание будет уделено процессу управления требованиями и соответствующим методам, которые можно непосредственно применять при разработке значительных *программных приложений, относящихся к типам IS/IT, ISV или к встроенным системам*.

Наш маршрут

Так как нам предстоит *в рамках отведенного срока и бюджета разработать качественное программное обеспечение, удовлетворяющее реальные потребности клиентов*, было бы полезно иметь “карту территории”. Задача достаточно сложная, если учесть, что на этом пути нам попадется множество людей, говорящих на самых разных языках. Может возникнуть множество вопросов.

- Это потребность или требование?
- Это то, что “хорошо бы иметь”, или то, что “должно быть”?
- Это постановка задачи или же речь идет о формулировке решения?
- Это – цель системы или одно из условий контракта?
- Действительно ли нужно программировать на языке Java? И кто будет это делать?
- Кому это не нравится наша новая система и где был этот человек, когда мы приходили сюда раньше?

Для успешного продвижения нам в любой момент времени необходимо четко понимать, где мы находимся, с какими людьми встречаемся, на каком языке они говорят и какая информация нам нужна от них для успешного завершения нашего путешествия. Давайте начнем движение с территории под названием “проблема”.

Область проблемы

Наиболее успешные путешествия в “стране требований” начинаются с *области проблемы*. Это – вотчина реальных пользователей и других заинтересованных лиц, чьи потребности мы должны удовлетворить, чтобы построить совершенную систему. Это вотчина людей, которым нужен “камень” – новая система ввода заказов или система управления конфигурацией, которая позволит им обойти конкурентов. В любом случае эти люди не похожи на нас. Их техни-

ческие и экономические познания отличаются от наших, они говорят смешанными архангелами, ходят на другие вечеринки и пьют другое пиво, они не надевают тенишки, идя на работу, и их мотивация кажется нам страшной и несвойственной.

Иногда это могут быть программисты, которым понадобилось новое инструментальное средство, или разработчики системы, попросившие нас разработать некую ее часть. В этих случаях данная часть нашего путешествия может оказаться проще (но может оказаться и сложнее!).

Но, как правило, мы находимся во владениях пользователя-чужестранца. У пользователей есть *технические* или *бизнес-задачи*, для решения которых им нужна наша помощь. Таким образом, *наша задача* состоит в том, чтобы понять *их* проблемы в *их* предметной области и на *их* языке и построить системы, удовлетворяющие *их* потребности. Поскольку эта территория кажется нам несколько туманной, мы будем изображать ее в виде облака. Это будет служить нам постоянным напоминанием, чтобы мы не забыли удостовериться, что правильно поняли все аспекты предметной области.

В этой области мы должны использовать ряд *профессиональных приемов* (*team skills*), чтобы понять *проблему*, которая должна быть решена.



Область проблемы

Потребности заинтересованных лиц

Мы также должны понять потребности пользователей и других заинтересованных лиц, на чью жизнь повлияет наше решение. По мере выявления мы образуем из этих потребностей заинтересованных лиц кластер, который будем представлять в виде небольшой пирамиды.



Переход к области решения

Путешествие по проблемной области не всегда будет сложным, а собранных там артефактов будет не так уж много. Однако даже этого небольшого количества данных нам хватит для следующей части нашего путешествия, к которой мы, пожалуй, подготовлены лучше, — решения поставленной задачи. В области решения мы уделяем основное внимание определению решения проблемы клиента; это сфера компьютеров, программирования, операционных систем, сетей и функциональных узлов. Здесь мы можем непосредственно применять свои профессиональные навыки.

Функции системы

Первым делом полезно сформулировать, что мы узнали в проблемной области и как собираемся этого достичь посредством решения. Это будет небольшой список, состоящий из элементов следующего вида.

- “У автомобиля будут автоматические стеклоочистители”
- “Диаграммы динамики обнаружения неисправностей будут снабжены визуальными средствами оценки прогресса”
- “Необходимо предусмотреть возможность ввода заказов через Internet”
- “Автоматический цикл двойной сварки”

Это просто описания на языке клиента, которые мы будем использовать в качестве ярлыков, чтобы пояснить заказчику, как наша система решает его задачу. Они станут частью нашего повседневного языка, и много сил будет потрачено на их определение, обсуждение и расстановку приоритетов. Эти описания мы назовем “функциями” (*features*) создаваемой системы и определим их следующим образом.



Функция – это предоставляемое системой обслуживание для удовлетворения одной или нескольких потребностей заинтересованных лиц

Графически функции будут представляться как основание описанной ранее пирамиды потребностей.

Требования к программному обеспечению

После того как определен набор функций и достигнуто соглашение с клиентом, можно переходить к определению более конкретизированных требований, которым должно удовлетворять решение. Построив систему, удовлетворяющую этим требованиям, можно будет с уверенностью утверждать, что она предоставит обещанные функции.

В свою очередь, поскольку функции предназначены для удовлетворения одной или нескольких потребностей клиента, эти потребности будут непосредственно удовлетворяться решением.

Этими более конкретизированными требованиями являются *требования к программному обеспечению (software requirements)*. Мы будем представлять их в виде части пирамиды аналогично представлению функций. Отметим, что они находятся в пирамиде гораздо ниже, и это означает, что нам придется проделать немалую работу, прежде чем мы достигнем этого уровня детализации.

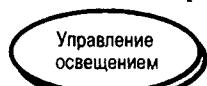
Понятие прецедентов

Наконец, еще одной важной конструкцией, которая поможет нам в нашем путешествии, является *прецедент (use case)*. В простейшем случае прецедент описывает последовательность действий, выполняемых системой, чтобы предоставить значимый результат пользователю. Другими словами, прецедент описывает серии взаимодействий пользователь/система, в результате которых пользователь решает некие свои задачи. Мы будем изображать прецедент в виде простой овальной пиктограммы, содержащей его название. Например,

если мы хотим описать прецедент, когда клиент использует компьютер просто для включения или выключения света, его можно назвать “Управление освещением” и поместить это название в овал.

Заключение

Посмотрим теперь на составленную нами карту. Из рис. 2.1 видно, что в процессе обсуждения мы незаметно совершили очень важное продвижение в нашем понимании задачи. Мы перешли от представленной облаком области проблемы и выявленных нами



потребностей клиентов к образующему область решения определению системы, представленному функциями системы и требованиями к программному обеспечению, согласно которым будет выполняться проектирование и реализация системы. Более того, мы осуществили этот переход логично и постепенно, предварительно удостоверившись в том, что осознали проблему и потребности пользователя, и лишь затем приступив к рассмотрению и определению решения. Эту "карту территории" мы будем еще раз использовать в данной книге.

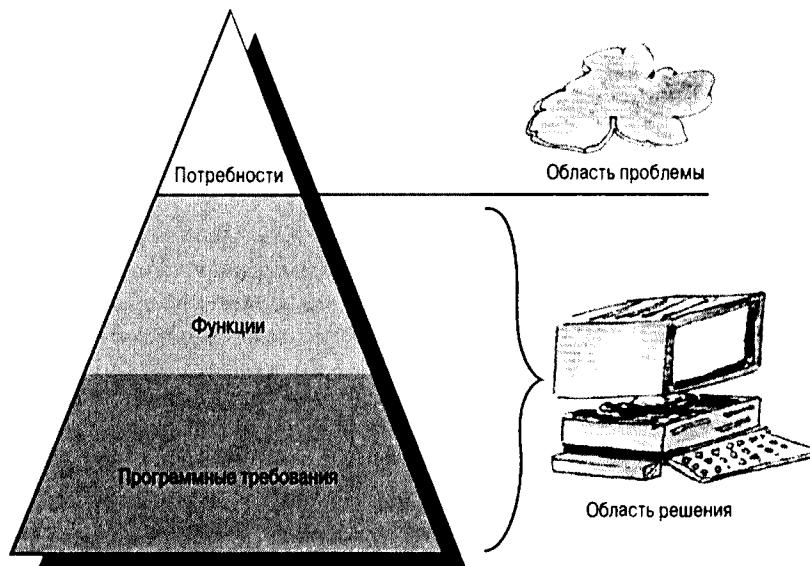
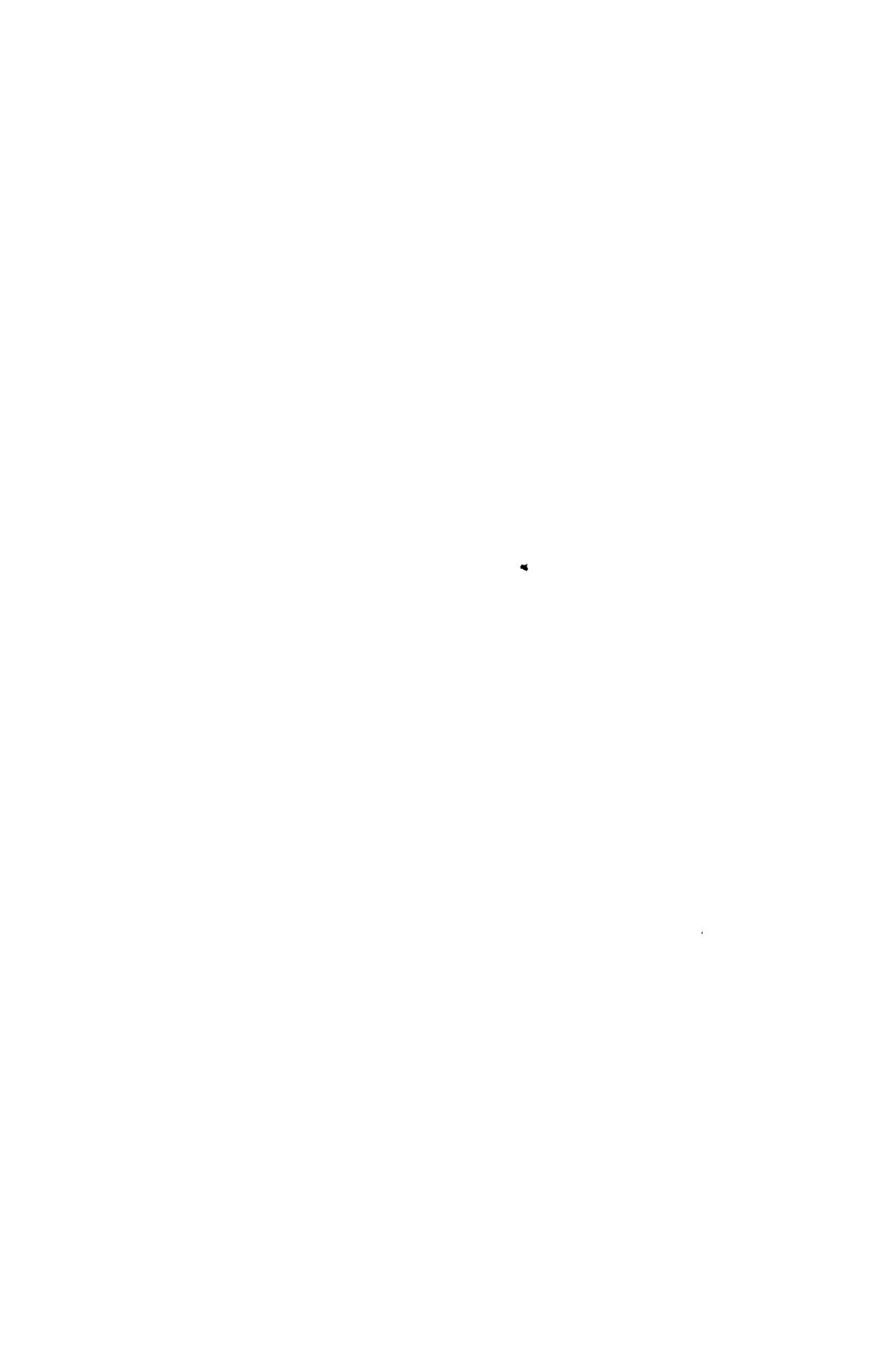


Рис. 2.1. Характеристика области проблемы и области решения



Глава 3

Команда разработчиков

*“Программирование компьютера – сфера деятельности человека.”
(Вайнберг, 1971)*

Основные положения

- Управление требованиями, хотя и по-разному, затрагивает каждого члена команды.
- Успешное управление требованиями может осуществлять только хорошо организованная команда разработчиков программного обеспечения.
- Для управления требованиями команде необходимо овладеть профессиональными приемами в шести основных областях.

Причины, по которым человек решает посвятить свою карьеру разработке программного обеспечения, могут быть различными. Некоторые начитались журналов “Popular Science” и “Popular Mechanics”, увлеклись курсом программирования еще в старших классах школы, специализировались в инженерных дисциплинах или информатике в колледже и таким образом направили свою жизнь по этому особому техническому пути. Для других это произошло само собой; наступил такой момент, когда возникла очевидная потребность в программном обеспечении, они приняли участие в разрешении этой потребности и постепенно оказались полностью вовлечеными в работу в данной отрасли.

В любом случае имению “очарование” технологии поддерживает горение пламени: *нам нравятся биты и байты, операционные системы, базы данных, средства разработки, краткие подстановки и языки программирования*. Кто, кроме разработчиков программного обеспечения, мог разработать операционную систему UNIX? Мы увлечены технологией, и это наша основная мотивация. Возможно, из-за врожденной генетической направленности или из-за того, что в колледже мы пропустили “ненужные” курсы по психологии, драматическому искусству или, что еще хуже, английскому, мы в нашей работе, как правило, уделяем меньше внимания человеческому фактору, чем битам и байтам. Мы не очень приятны в общении¹, а многие вообще испытывают сложности в отношениях с людьми вне работы, когда нет общих технических тем для обсуждения.

¹Например, в день открытых дверей в компании RELA (1979 год) один программист на протяжении всей вечеринки оставался за своим рабочим столом, успевшо программируя даже тогда, когда его стол оказался в центре комнаты, где проходила вечеринка. Закончив свою работу, он просто встал, убрал свои бумаги и ушел, никому не сказав ни слова. Что необычного в его поведении? В нашей отрасли? Ничего!

Одним из результатов этого, которому также способствовала простая однопользовательская природа используемых средств и достаточно скромные размеры разрабатываемых приложений, явилась тенденция рассматривать разработку программного обеспечения как индивидуальную деятельность. Программист самостоятельно определял, проектировал, писал и, как правило, тестировал свою работу. Возможно, приглашались тестологи для того, чтобы помочь на этапе окончания, но в основном упор делался на индивидуальную деятельность. Программист-герой был основной парадигмой.

Разработка программного обеспечения как командная деятельность

"Разработка программного обеспечения стала командным видом спорта." (Буч, 1998)

Но в некоторый момент правила игры изменились. Почему? Уоттс Хамфри (Watts Humphrey, 1989) отметил следующее.

"История развития разработки программного обеспечения — это история роста ее масштабов."

История развития разработки программного обеспечения — это история роста ее масштабов. Первоначально несколько индивидуально работающих программистов могли предложить "хитрые" маленькие программы; вскоре этого стало недостаточно. Начали создаваться команды, состоящие из дюжины, а то и двух дюжин человек, но успех был переменным. Пока многие организации занимались решением проблем небольших систем, масштабы наших работ продолжали расти. На сегодняшний день крупные проекты, как правило, требуют координированной работы многих команд.

Хамфри также заметил, что возрастание сложности снижает способность человека решать задачи интуитивно по мере их возникновения. Например, мы занимаемся проектом разработки требований, который одновременно оказывает влияние примерно на 30 продуктов, входящих в некую достаточно большую группу. Генерируемые требования воздействуют в реальном времени на программное обеспечение, которое пишут более 400 находящихся в разных местах программистов. Чтобы добиться успеха в этом проекте, необходима четкая координация действий "команды команд", которая должна работать по общей методологии, чтобы решить проблему требований.

Процесс управления требованиями, хотя и по-разному, затрагивает каждого члена команды.

Успешное управление требованиями может осуществляться только эффективно организованной командой разработчиков.

Что необходимо сделать? Очевидно, необходимо заставить работать "командный фактор". Как указывал Бом (Boehm, 1981) в модели оценки стоимости СОСМО, возможности команды оказывают наибольшее влияние на производительность в сфере раз-

работки программного обеспечения. Дэвис (Davis, 1995, b) согласился с ним и подчеркнул, что *оптимизация производительности каждого отдельного разработчика не обязательно приведет к оптимизации производительности команды* (с.170). Поэтому представляется логичным потратить некоторые усилия на то, чтобы повысить производительность команд разработчиков программного обеспечения.

Профессиональные навыки, которыми должна обладать команда для эффективного управления требованиями

В данной книге выделяются шесть частей, соответствующих шести основным областям, профессиональными приемами в которых необходимо овладеть современной группе разработчиков программного обеспечения, чтобы справиться с задачами управления требованиями.

- В части 1, “Анализ проблемы”, предлагается ряд методов, применяемых для достижения правильного понимания *проблемы*, решить которую призвана новая система программного обеспечения.
- В части 2, “Понимание потребностей пользователей”, представлены разнообразные приемы *выявления требований* посредством общения с пользователями системы и другими заинтересованными лицами. Невозможно указать набор методов, подходящих для всех ситуаций, также нет необходимости применять все методы. Пройдет немного времени, и команда сама сможет выбрать те из них, которые позволят ей гораздо лучше понимать реальные потребности, удовлетворить которые призвана разрабатываемая система.
- В части 3, “Определение системы”, описывается процесс преобразования понимания проблемы и потребностей клиентов в исходное определение системы, которая будет удовлетворять эти потребности.
- В части 4, “Управление масштабом”, мы научим команду более успешно управлять масштабом проекта. В конце концов, насколько бы хорошо потребности не были осознаны, команда не может сделать невозможное, и нам часто придется вести переговоры о том, что считать приемлемым, прежде чем будет достигнут успех.
- В части 5, “Уточнение определения системы”, описывается, как *организовать* информацию о требованиях. Затем предлагается ряд методов *уточнения* определения системы до уровня детализации, пригодного для проведения проектирования и реализации, чтобы вся расширенная команда точно знала, какую систему она разрабатывает.
- Наконец, в части 6, “Построение правильной системы”, рассматриваются некоторые более формальные аспекты *достоверности технического проекта, верификации, проверки правильности системы и управления изменениями*. Мы покажем, как можно с помощью *трассировки* удостовериться в качестве результата.

Члены команды имеют различные профессиональные навыки

Одним из наиболее важных фактов является то, что члены команды имеют различные профессиональные навыки. В конце концов, именно это делает команду командой. Уолкер Ройс (Walker Royce, 1998) отмечал следующее.

Баланс и разнообразие навыков – вот две наилучшие составляющие отличной команды... Как в футбольной команде существует потребность в разнообразных навыках, точно так же происходит и в команде, разрабатывающей программное обеспечение... Вряд ли можно представить себе хорошую футбольную команду², в которой нет профессионалов различного профиля: нападающих, защитников, тренеров, игроков первого состава, запасных, разводящих и бегущих. Хорошей команде необходимо, чтобы все ключевые позиции занимали сильные игроки. Но команду, перегруженную "звездами", где каждый игрок стремится установить индивидуальный рекорд и борется за лидерство, может победить сбалансированная команда, в которой немного лидеров и они нацелены на общий успех – победу команды в игре.

В команде, разрабатывающей программное обеспечение, одни "игроки" способны эффективно работать с заказчиками, другие – имеют навыки программирования, а третьи – умеют тестировать программы. Еще команде нужны специалисты по проектированию и архитектуре. Понадобятся и многие другие навыки. Мы рассчитываем, что предлагаемые нам необходимые для управления требованиями профессиональные приемы пригодятся различным членам команды. Мы надеемся развить способность каждого члена команды содействовать эффективному управлению требованиями. И мы, по возможности, будем указывать, каким именно членам команды наиболее необходим тот или иной навык.

Организация команд, разрабатывающих программное обеспечение

Процесс разработки программ чрезвычайно трудоемок, а области применения наших навыков весьма различны. Поэтому трудно ожидать, что некий способ организации команды будет во всех случаях предпочтительнее, чем альтернативные варианты. Тем не менее определенные общие элементы присутствуют во многих успешных командах. Поэтому нам кажется важным предложить некую гипотетическую структуру команды. Но вместо того, чтобы рассматривать идеальную команду, слишком простую и академичную, мы решили взять за образец реальную команду разработчиков.

Прототипом команды, которую мы будем моделировать, послужила реальная команда разработчиков программного обеспечения, добившаяся успеха в двух важнейших областях: (1) управлении требованиями и (2) предоставлении программного продукта в рамках отведенного времени и бюджета. (Мы надеемся, что это очевидная причинно-следственная взаимосвязь!) Безусловно, команда должна также обладать многими другими навыками, чтобы действительно успешно работать из года в год. В нашем рабочем примере команда работает для компании Lumenations, Ltd., разрабатывающей "автоматизированную систему освещения жилища" нового поколения для оснащения жилья самого высокого класса.

²Речь идет об американском футболе. – Прим. перев.

Рабочий пример

Мы сможем попутно достигнуть еще одной цели в данной книге, если разрабатываем пе-кий рабочий пример, который сможем проследить от начала разработки требований до конца. Таким образом мы сможем не только применить рассматриваемые методы к нашему примеру, но и предложить образцы рабочих продуктов (артефакты) для более полной иллюстрации основных положений. Эти артефакты вы можете использовать в качестве образцов в ваших проектах (они содержатся в приложении А).

Предварительная информация для рабочего примера

Компания Lumenations, Ltd., более 40 лет являлась всемирно известным поставщиком коммерческих осветительных систем для театра. В 1999 году ее годовой доход составил приблизительно 120 миллионов долларов, а объем продаж стабилизировался. Lumenations – открытая компания, и недостаточный рост продаж и, что еще хуже, отсутствие реальных предложений по его повышению оказали крайне негативное влияние на компанию и ее акционеров. Последнее ежегодное собрание акционеров было весьма неожиданным, так как не было сказано ничего нового о перспективах роста компании. Цена акции недолго поднялась до 25\$, прошлой весной на волне новых заказов, но затем вновь опустилась до 15\$.

В отрасли, которая занимается созданием оборудования для театров, в целом мало новых разработок. Данная отрасль является устоявшейся и весьма консолидированной. Поскольку оборотные средства компании Lumenations ограничены, а капитализация весьма умеренна, приобретение другой компании в данном случае не является выходом.

Что действительно нужно – это новая позиция на рынке сбыта, не очень отличающаяся от того, в чем специализируется компания, но где есть простор для роста дохода и прибыли. Проведя тщательное исследование рынка и истратив немало средств на консультации по маркетингу, компания приняла решение выйти на новый рынок *систем автоматического освещения для жилых помещений самого высокого класса*. Этот рынок растет приблизительно на 25–35% ежегодно. Еще отраднее то, что этот рынок был образован недавно, и на нем еще нет постоянных игроков, занимающих доминирующие позиции. Мощные разветвленные по всему миру каналы дистрибуторов будут настоящим активом на рынке, а дистрибуторы всегда охочи до новых продуктов. Это прекрасная возможность!

Команда разработчиков программного обеспечения HOLIS

Проектом, который мы выбрали в качестве рабочего примера, будет разработка системы HOLIS (это кодовое наименование новой домашней системы автоматического управления освещением – Home Lighting automation System), распространяемой компанией Lumenations. Команда HOLIS имеет типичные размеры. В нашем примере она достаточно небольшая (всего 15 человек), но этого вполне достаточно для того, чтобы все необходимые профессиональные навыки были представлены отдельными членами команды с определенной степенью специализации их ролей. Наиболее важна именно структура команды; если добавить дополнительных разработчиков и тестологов, команда будет включать 30–50 человек и сможет разрабатывать пропорционально более объемные программные приложения.

Для работы на новом рынке сбыта компания Lumenations учредила новое подразделение, отдел автоматического управления домашним освещением. Поскольку отдел и технология являются достаточно новыми для компании Lumenations, команда HOLIS, в основном, образована из новых сотрудников, хотя некоторые члены команды перешли в нее из отдела коммерческого освещения.

На рис. 3.1 представлена организационная схема команды разработчиков и взаимосвязи между ее членами. Мы периодически будем возвращаться к этой схеме далее по ходу изложения, чтобы посмотреть, как команда применяет новые профессиональные навыки для решения проблем требований к HOLIS.

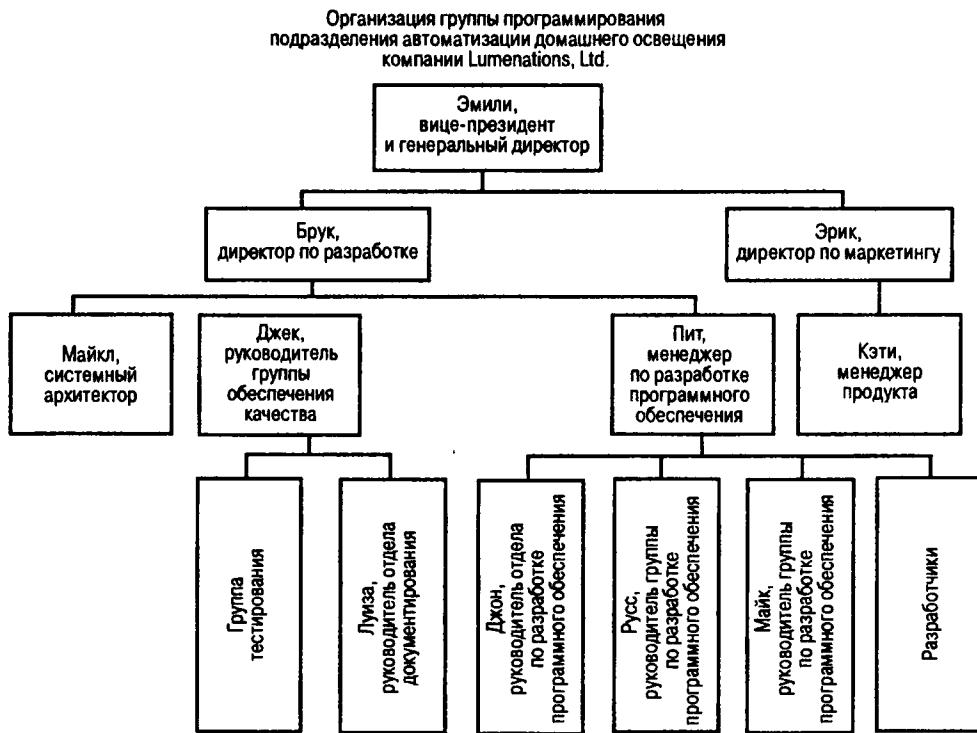


Рис. 3.1. Организационная схема команды разработчиков программного обеспечения системы HOLIS

Заключение

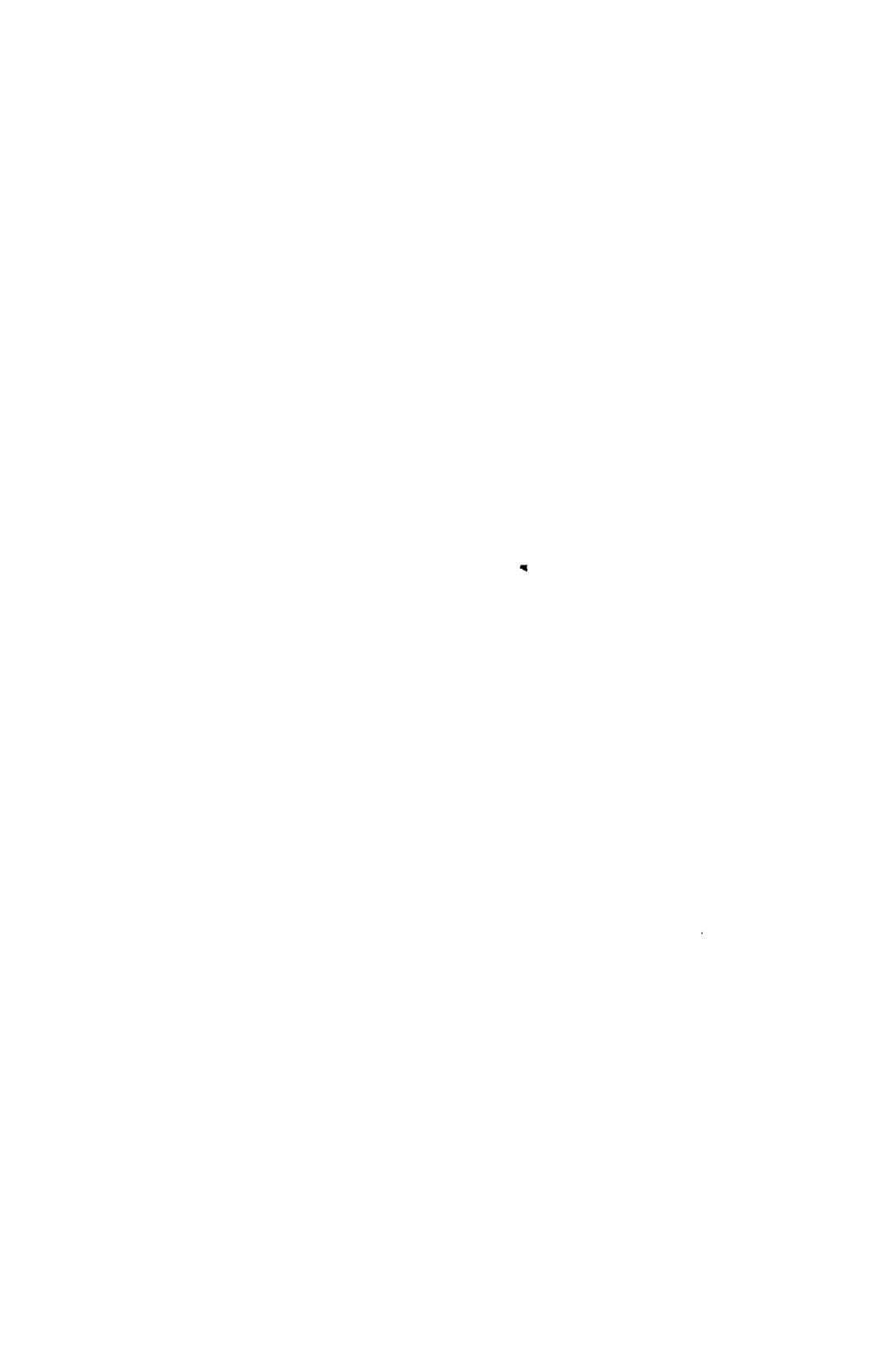
Вряд ли кто-нибудь будет выступать против управления требованиями к системе и их документирования с целью удостовериться, что мы предлагаем клиенту именно то, что он хотел. Однако данные свидетельствуют о том, что в отрасли этому уделяется недостаточное внимание. Недостаток информации от клиента, неподные требования и спецификации, а также изменения требований и спецификаций наиболее часто указываются в качестве основных причин возникновения проблем в проектах, не достигших поставленных целей. А число программных проектов, потерпевших неудачу в достижении намеченных целей, действительно велико.

Часто разработчики и клиенты придерживаются позиции, что “даже если мы в действительности точно не знаем, чего хотим, лучше приступить к реализации сейчас, поскольку у нас мало времени. Мы сможем уточнить требования позже”. Но слишком часто этот продуктивный благими намерениями подход превращается в хаотические действия разработчиков, когда никто в точности не знает, чего в действительности хочет клиент и что на самом деле делает созданная на данный момент система. При наличии современных мощных и простых в использовании средств построения прототипов создается впечатление, что если разработчики сумеют создать хотя бы грубый прототип, то клиент сможет указать, какие функции необходимо добавить, удалить или изменить. Это *может* сработать, и это один из важных аспектов итеративной разработки. Но из-за слишком высокой стоимости устранения ошибок требований данный процесс должен происходить в *рамках общей стратегии управления требованиями*, иначе результаты будут хаотичными.

Как узнать, что должна делать система? Как отслеживать текущее состояние требований? Как определить, какое воздействие окажет определенное изменение? Именно из-за подобных вопросов управление требованиями стало необходимой практической дисциплиной в инженерии программного обеспечения. Мы предложили вашему вниманию основополагающую философию управления требованиями и ряд определений, поддерживающих данную деятельность.

Поскольку история развития программного обеспечения и его будущее, по крайней мере обозримое, связаны с возрастанием сложности, нужно отдавать себе отчет, что задачи разработки программного обеспечения должны решаться хорошо организованными и хорошо подготовленными *командами* разработчиков. В частности, в сфере управления требованиями каждый член команды будет время от времени привлекаться к разработке требований к системе. Команды должны обладать необходимыми профессиональными навыками, чтобы понимать потребности клиента, управлять масштабом приложения и создавать системы, удовлетворяющие указанные потребности. Команда должна работать как команда, чтобы решить задачу управления требованиями.

Первый шаг в процессе управления требованиями должен состоять в том, чтобы удостовериться, что разработчики действительно поняли “проблему”, которую пытается решить пользователь. Эта тема рассматривается в последующих трех главах, объединенных в часть 1, “Анализ проблемы”.



Часть 1

Анализ проблемы

- Глава 4. Пять этапов анализа проблемы
- Глава 5. Моделирование бизнес-процессов
- Глава 6. Инженерия систем, интенсивно использующих программное обеспечение



В последние годы наблюдался беспрецедентный рост возможностей средств и технологий, используемых разработчиками программного обеспечения для создания приложений в различных современных предметных областях. Новые языки программирования повысили уровень абстракции и производительность решения проблем пользователей. Применение объектно-ориентированных методов позволило создавать более устойчивые и расширяемые проектные решения. Автоматические средства управления конфигурацией, управления требованиями, проектирования и анализа, обнаружения неисправностей и автоматического тестирования помогают разработчикам программного обеспечения справиться с тысячами требований и сотнями тысяч строк программного кода.

Казалось бы, увеличение производительности средств разработки программного обеспечения должно существенно упростить разработку систем, удовлетворяющих реальные бизнес-потребности. Однако данные свидетельствуют, что проблемы, связанные с правильным пониманием и удовлетворением этих потребностей, по-прежнему существуют. Возможно, этому есть простое объяснение. *Команды разработчиков тратят слишком мало времени на то, чтобы понять проблемы реального предприятия, потребности пользователей и других заинтересованных лиц, а также природу той среды, в которой должны функционировать их приложения.* Они торопятся и предлагают технические решения, основанные на неадекватном понимании решаемой проблемы.

Команды разработчиков забегают вперед, предлагая технические решения, основанные на неадекватном понимании решаемой проблемы.

Полученные в результате системы не могут удовлетворить потребности пользователей и заинтересованных лиц в той мере, как ожидалось. Последствия – недостаточный экономический эффект для заказчиков и разработчиков системы и неудовлетворенность пользователей. Кроме того, это может отрицательно отразиться на репутации разработчиков. Поэтому мы считаем, что увеличение инвестиций в анализ проблемы приведет к ощущимой финансовой отдаче. В данной части книги излагаются основные принципы анализа проблем, а также указываются конкретные цели его применения при разработке приложений.

В последующих главах описывается, как точно определить, в чем состоит проблема. Ведь без этого невозможно предложить подходящее решение.

Глава 4

Пять этапов анализа проблемы

Основные положения

- Анализ проблем – это процесс осознания реальных проблем и потребностей пользователей и предложения решений, позволяющих удовлетворить эти потребности.
- Цель анализа проблемы состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки.
- Чтобы выявить причины (или проблемы, стоящие за проблемой), необходимо опросить людей, которых непосредственно затрагивает данная проблема.
- Выявление акторов системы является ключевым шагом в анализе проблемы.

Эта глава посвящена тому, как команда разработчиков может осознать реальные потребности заинтересованных лиц и пользователей новой системы (или приложения). Большинство систем создается для решения определенной проблемы. Чтобы правильно понять, в чем состоит проблема, мы будем использовать методы *анализа проблем*.

Однако не каждое приложение разрабатывается для решения определенной проблемы, некоторые из них создаются для того, чтобы воспользоваться предоставляемыми рынком возможностями, даже если существование проблемы еще не очевидно. Например, замечательные программные приложения, такие как SimCity и Myst, оказались нужны тем, кто любит компьютерные игры и умственные упражнения, по-настоящему увлекается моделированием и имитацией. Поэтому, хотя и сложно сказать, какую проблему решали эти приложения – возможно, проблему “недостатка классных вещей, которые можно проделывать с компьютером” или “наличия слишком большого количества свободного времени у некоторых” – очевидно, что эти продукты представляют реальную ценность для значительного числа пользователей.

В некотором смысле проблемы и возможности – это две стороны одной медали; ваша проблема является моей возможностью. Все зависит от точки зрения. Но поскольку большинство систем действительно предназначено для решения существующих проблем, можно упростить обсуждение и, не вдаваясь в путаницу проблема/возможность, заняться только проблемной стороной. В конце концов, нам нравится представлять себя в роли “решателей проблем”.

Анализ проблемы – это процесс осознания реальных проблем и потребностей пользователя и предложения решений для удовлетворения этих потребностей.

При этом необходимо проанализировать и понять область проблемы и исследовать разнообразные области решений. Как правило, возможных решений множество, и нам необходимо найти то, которое наиболее соответствует решаемой проблеме.

Чтобы иметь возможность провести анализ проблемы, полезно определить, что же собой представляет проблема. По определению Гауса и Вайнберга (Gause, Weinberg, 1989)

проблема – это разница между желаемым и воспринимаемым.

Это определение достаточно разумно, по крайней мере, оно устранит часто встречающееся среди разработчиков заблуждение, что подлинная проблема заключается в том, что пользователь не понимает, в чем состоит проблема! Согласно данному определению, если пользователь ощущает нечто как проблему, это и есть настоящая проблема, и она достойна внимания.

Иногда самым простым решением является изменение бизнес-процесса, а не создание новой системы.

Опираясь на приведенное выше определение, наш коллега Элмер Мэгэзинер (Elmer Magaziner) заметил, что существует множество путей решения проблемы. Например, изменение желаний или восприятия пользователей может быть наиболее эффективным (в плане затрат) подходом. Это может осуществляться путем формирования ожиданий и управления ими, внесения предложений по организации работы или частичному улучшению существующих систем, предложения альтернативных решений, не требующих разработки новой системы, или проведения дополнительного обучения. Практика знает много примеров, когда именно изменение восприятия приводило к наилучшим, наискорейшим и самым дешевым из возможных решений! Если мы взялись решать проблемы, то на нас возложена обязанность исследовать эти альтернативные решения прежде, чем приступить к решению с помощью новой системы.

Однако когда указанные альтернативные действия оказываются не в состоянии заметно уменьшить расхождение между воспринимаемым и желаемым, перед нами встает наиболее сложная и дорогостоящая задача: уменьшить разницу между *желаемым* и *воспринимаемым* путем определения и реализации *новых систем*.

Цель анализа проблемы состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки.

Как и всегда, начинать следует с определения цели. Цель анализа проблемы состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки. Для этого необходимо осуществить следующие пять этапов.

1. Достигнуть соглашения об определении проблемы.
2. Выделить основные причины – проблемы, стоящие за проблемой.
3. Выявить заинтересованных лиц и пользователей.
4. Определить границу системы решения.
5. Выявить ограничения, которые необходимо наложить на решение.

Давайте подробно рассмотрим каждый из этих этапов.

Этап 1. Достижение соглашения об определении проблемы

Первый шаг состоит в достижении соглашения об определении проблемы, которую необходимо решить. Один из простейших способов заключается в том, чтобы *просто записать проблему и выяснить, все ли согласны с такой постановкой*.

В рамках этого процесса зачастую полезно рассмотреть преимущества предлагаемого решения, причем их следует описывать на языке клиентов/пользователей. Это обеспечивает дополнительную содержательную основу для понимания реальной проблемы. Рассматривая с точки зрения клиента эти преимущества, мы также достигаем лучшего понимания их взгляда на проблему в целом.

Постановка проблемы

Часто бывает полезно записать проблему в стандартной форме (табл. 4.1). Создание подобной таблицы является простым, но действенным средством, чтобы удостовериться в том, что все участники проекта работают вместе над осуществлением общей цели.

Таблица 4.1. Стандартная форма постановки проблемы

Элемент	Описание
Проблема	[Описание проблемы]
воздействует на	[Указание лиц, на которых оказывает влияние данная проблема]
результатом чего является	[Описание воздействия данной проблемы на заинтересованных лиц и бизнес-деятельность]
Выигрыш от	[Указание предлагаемого решения]
может состоять	[Список основных предоставляемых решением преимуществ]
в следующем	

Может показаться, что достижение согласия относительно определения решаемой проблемы — шаг небольшой и малозначащий, и чаще всего так оно и есть. Но это всегда. Например, один из наших клиентов, производитель оборудования, занялся усовершенствованием своей IS/IT-системы, обеспечивающей пересылку счетов и финансовых отчетов между компанией и ее дилерами. Задачей новой программы было “улучшить средства коммуникации дилеров”. В свете этого команда подготовилась разрабатывать новую масштабную систему. Данный пример достижения соглашения о решаемой проблеме весьма показателен. Предложенное командой определение решения предполагало создание мощной новой системы, предусматривающей улучшенную финансовую отчетность, усовершенствованные формы счетов и отчетов, возможность заказа запчастей в интерактивном режиме и электронную почту. Помимо этого, команда *надеялась* (если получится) обеспечить возможность электронного перевода средств между компанией и дилером.

Во время согласования постановки проблемы руководство компании имело возможность вносить свои предложения. Точка зрения руководства оказалась совершенно иной. По его мнению, главная цель новой системы должна была состоять в том, чтобы *обеспечить электронный перевод средств, который улучшит движение наличных средств компаний*. После бурной дискуссии стало очевидно, что первостепенной проблемой, которую при-

звана решить новая система, является электронный перевод средств; а электронная почта и другие средства коммуникации дилеров рассматривались только как "желательные". Нечего и говорить, налицо была значительная переориентация целей новой системы. В новой постановке в качестве решаемой проблемы было указано обеспечение электронного перевода средств. Эта переориентация также привела к разработке отличной от предложенной ранее системной архитектуры, дополненной средствами обеспечения безопасности, которые соответствуют присущим электронной банковской деятельности рискам.

Этап 2. Выделение основных причин — проблем, стоящих за проблемой

Для понимания реальной проблемы и ее причин можно использовать множество методов. Одним из них является *метод анализа корневых причин* (root cause analysis), представляющий собой семантический способ нахождения причин, лежащих в основе рассматриваемой проблемы или ее проявления.

Рассмотрим реальный пример. Компания GoodsAreUs, занимающаяся торговлей по каталогу, производит и рассыпает на дом множество недорогих товаров различных наименований. Решив заняться проблемой недостаточной прибыльности, компания использует рекомендуемую ее программой обеспечения качества методику "качество — во всем" (total quality management, TQM). Применив данный подход, компания практически сразу обратила внимание на *ущерб от несоответствия* (*cost of nonconformance*), который представляет собой стоимость всего, что идет не так, как надо, и приводит к бесполезным затратам. Этот ущерб включает в себя переделки, остатки, неудовлетворенность клиента, текучесть кадров и другие негативные факторы. Проанализировав ущерб от несоответствия, компания заподозрила, что наибольший вклад в него вносят "остатки".

Следующим шагом должно стать определение того, какие факторы оказывают влияние на величину остатков. TQM советует для обнаружения проблем, стоящих за проблемой, использовать диаграмму в форме рыбного скелета (рис. 4.1). В нашем случае компания выявила много источников, вносящих свой вклад в остатки. Каждый источник указан как одна из "косточек" на диаграмме.

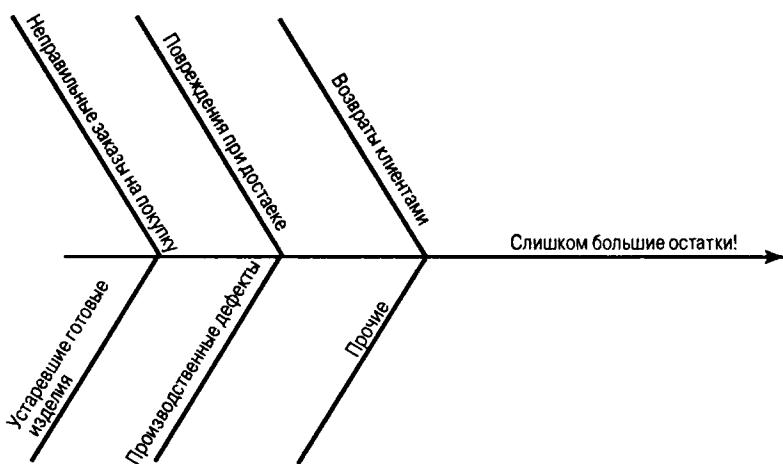


Рис. 4.1. Диаграмма в форме рыбного скелета для отображения корневых причин

Способ выявления корневых причин зависит от конкретного случая. Иногда просто нужно спросить людей, непосредственно занимающихся этим делом, что они считают корневыми причинами. Даже удивительно, сколько людей *на самом деле* знают о проблемах, стоящих за проблемой, но никто — мы имеем в виду руководство — не находил прежде времени спросить их об этом. Итак, спросите их, а затем спросите их еще раз.

Но если проблема более серьезна, простого опроса сотрудников недостаточно. Может понадобиться произвести детальное исследование каждой из перечисленных проблем и количественно определить вклад каждой в отдельности. Этого можно добиться как посредством “мозгового штурма” с участием тех, кто знаком с данной областью, так и с помощью небольшого проекта по сбору данных или, возможно, более строгого научного исследования. В любом случае цель состоит в том, чтобы количественно оценить вероятный вклад каждой корневой причины.

Устранение корневых причин

Качественные данные свидетельствуют, что многие корневые причины не стоят того, чтобы их устранять.

Конечно, инженер в каждом из нас захочет устранить *все* корневые причины на “косточках” диаграммы. Кажется, что это правильно. Но так ли это? Зачастую — нет. Качественные данные свидетельствуют, что многие корневые причины просто не стоят того, чтобы их устранять, поскольку затраты на их устранение превысят причиняемый проблемой ущерб. Как же узнать, какие из них устраниТЬ? Ответ: необходимо определить влияние каждой корневой причины. Результат этого исследования можно отобразить в виде *Парето-диаграммы*, визуально отражающей реальных “виновников”.

Вернемся к нашему примеру. Предположим, что в результате сбора данных получилась картина, изображенная на рис. 4.2. Как видно, команда обнаружила, что *одна* корневая причина — “неправильные заказы на покупку” — является источником половины всех остатков. Если, в свою очередь, окажется, что существующая система заказов на покупку является образцом ошибочного кода, недружественного интерфейса пользователя и не предоставляет возможность устранения ошибок в интерактивном режиме, тогда действительно можно сократить остатки с помощью разработки нового программного обеспечения.

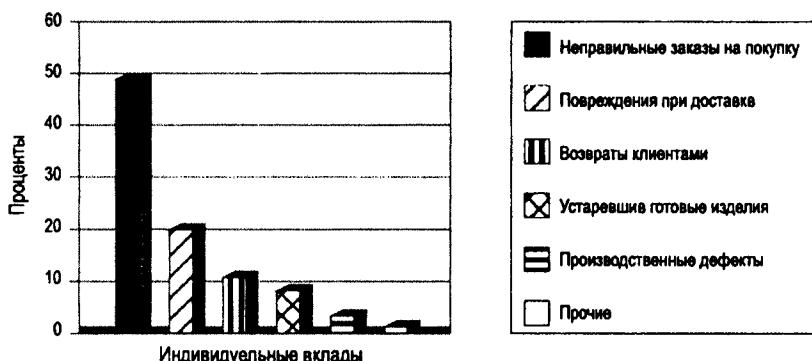


Рис. 4.2. Парето-диаграмма корневых причин

В этот момент, и только в этот момент, можно считать аргументированным предложение команды заменить существующую систему ввода заказов на покупку. Затем можно предоставить стоимостное обоснование такой системы, оценив затраты на разработку и дивиденды от уменьшения остатков.

Продолжая анализ, можно применить новую диаграмму в виде "рыбного скелета" для определения того, какие типы ошибок вносят наибольший вклад в проблему неправильности заказов. Полученные данные можно затем использовать для определения функций новой системы программного обеспечения, которая призвана устранить эти ошибки. В нашем случае, однако, можно завершить анализ выводом, что замена системы обработки заказов на покупку поможет, по крайней мере частично, решить проблему слишком больших остатков.

Раз мы приняли решение, что проблема неправильных заказов на покупку достойна решения, можно создать для нее формальную постановку, как показано в табл. 4.2.

Таблица 4.2. Постановка проблемы ввода заказов на покупку

Элементы	Описание
Проблема	неправильных заказов на покупку
воздействует на	выполняющий заказы персонал, клиентов, производство, продажи и обслуживание клиентов,
результатом чего является	увеличение остатков, повышение производственных затрат, недовольственность клиентов, уменьшение прибыли.
Выигрыш от	<p>новой системы, направленной на решение данной проблемы, может состоять в следующем.</p> <ul style="list-style-type: none"> ■ Повышение точности заказов на покупку в точке ввода ■ Совершенствование учета данных о покупках <p>В конечном счете – увеличение прибыли.</p>

После написания постановки проблемы, ее следует передать для ознакомления заказчикам и заинтересованным лицам, чтобы они внесли свои комментарии. Затем постановка проблемы доводится до сведения всех членов команды разработчиков с тем, чтобы все работали в направлении достижения общей цели.

Этап 3. Выявление заинтересованных лиц и пользователей

При решении любой сложной проблемы, как правило, приходится удовлетворять потребности различных групп заинтересованных лиц. Эти группы обычно имеют различные точки зрения на проблему и различные потребности, которые должны быть учтены в решении.

Заинтересованные лица – это все, на кого реализация новой системы или приложения может оказать материальное воздействие.

Понимание потребностей пользователей и других заинтересованных лиц является ключевым фактором в выработке успешного решения.

Первая категория заинтересованных лиц – это пользователи системы. Их потребности легко учесть, поскольку они будут непосредственно привлекаться к определению и использованию системы. Вторую категорию составляют непрямые пользователи, а также те, на кого воздействуют только бизнес-последствия разработки. Этих заинтересованных лиц можно найти в соответствующей бизнес-области или в "окрестностях" среди конкретного приложения. Третья категория заинтересованных лиц может находиться еще дальше от среды приложения. Среди них могут быть люди и организации, вовлеченные в разработку системы, субподрядчики, клиенты клиентов, внешние регулирующие инстанции, например Федеральное управление гражданской авиации США (U.S. Federal Aviation Administration, FAA), Управление по санитарному надзору за пищевыми продуктами и медикаментами (Food and Drug Administration, FDA), или другие агентства, взаимодействующие с системой или участвующие в процессе разработки. Каждая из перечисленных категорий заинтересованных лиц может оказывать влияние на требования к системе или будет каким-либо образом связана с результатом работы системы.

Потребности заинтересованных лиц, не являющихся пользователями, также необходимо выявить и учесть.

Понимание того, кто же такие эти заинтересованные лица, и выявление их потребностей являются важными факторами разработки успешного решения. В зависимости от того, в какой предметной области работает команда, выявление заинтересованных лиц может оказаться как тривиальным, так и нетривиальным этапом анализа проблемы. Часто достаточно провести простой опрос среди тех, кто принимает решения, а также опросить потенциальных пользователей и другие заинтересованные стороны. В этом процессе могут оказаться полезными следующие вопросы.

- Кто является пользователями системы?
- Кто является заказчиком (экономическим покупателем) системы?
- На кого еще окажут влияние результаты работы системы?
- Кто будет оценивать и принимать систему, когда она будет представлена и развернута?
- Существуют ли другие внутренние или внешние пользователи системы, чьи потребности необходимо учесть?
- Кто будет заниматься сопровождением новой системы?
- Не забыли ли мы кого-нибудь?

В нашем примере замены системы заказов на покупку основными и наиболее очевидными пользователями являются служащие, занимающиеся вводом заказов на покупку. Они определенно являются заинтересованными лицами, так как их производительность, удобство, комфорт, выполнение работы и ее результаты зависят от системы. Кого еще из заинтересованных лиц можно выделить?

На руководителя отдела приема заказов система также оказывает непосредственное воздействие, но он взаимодействует с системой не напрямую, а посредством различных интерфейсов пользователя и форм отчетов. Главный финансист компании также, очевидно, принадлежит к заинтересованным лицам, так как ожидается, что система повлияет на производительность, качество предоставляемых услуг и прибыльность компании. Наконец, администратор информационной системы и члены команды, разрабатываю-

щей приложение, также являются заинтересованными лицами, так как они будут отвечать за разработку и сопровождение системы. Они также, как и пользователи, будут зависеть от поведения системы. Результаты выявления пользователей и заинтересованных лиц новой системы ввода заказов на покупку представлены в табл. 4.3.

Таблица 4.3. Пользователи и лица, заинтересованные в новой системе

Пользователи	Другие заинтересованные лица
Служащие, занимающиеся вводом заказов	Администратор информацией системы и команда разработчиков
Руководитель отдела приема заказов	Главный финансист
Контроль производства	Управляющий производством
Служащий, выписывающий счета	

Этап 4. Определение границ системы-решения

После того как согласована постановка проблемы и выявлены пользователи и заинтересованные лица, можно перейти к *определению системы*, разрабатываемой для решения данной проблемы. Это важный момент, когда необходимо постоянно помнить как о понимании проблемы, так и о свойствах потенциального решения.

Следующий важный этап состоит в том, чтобы определить границы системы-решения. Границы системы – это “водораздел” между решением и окружающим его реальным миром (рис. 4.3.) Иными словами, граница системы описывает оболочку, в которой заключена система. Информация в виде ввода и вывода передается от находящихся вне системы пользователей системе и обратно. Все взаимодействия с системой осуществляются посредством интерфейсов между системой и внешним миром.



Рис. 4.3. Отношение ввод/система/вывод

Мы делим мир на две части.

1. Наша система
2. То, что взаимодействует с нашей системой

Другими словами, если мы собираемся нечто создать или модифицировать – это часть нашего решения, которая находится внутри границы; если нет – это нечто внешнее по отношению к системе. Таким образом, мы делим мир на два интересующих нас класса.

- Наша система
- То, что взаимодействует с нашей системой

Определим “то, что взаимодействует с нашей системой”, общим понятием “акторы” (actors). Они выполняют некоторые действия, заставляя систему делать ее работу. Актор

изображается простой пиктограммой в виде человечка. Его определение выглядит следующим образом.

Актор – это находящееся вне системы нечто (или некто), взаимодействующее с системой.

С помощью данного понятия мы можем проиллюстрировать границы системы (рис. 4.4).

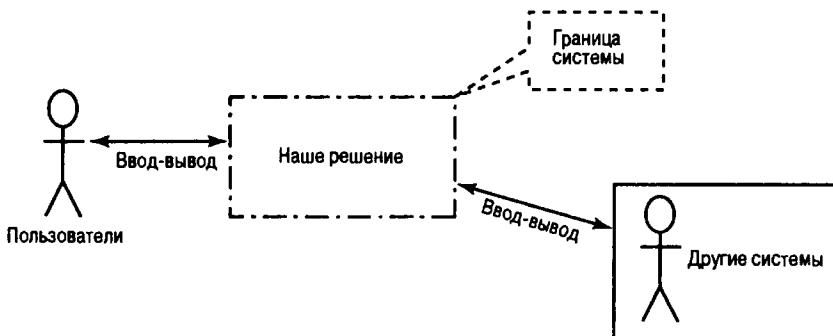


Рис. 4.4. Границы системы

Во многих случаях границы системы очевидны. Например, однопользовательский персональный планировщик контактов, работающий на автономной платформе Windows 2000, имеет достаточно хорошо определенные границы. Имеется всего один пользователь и одна платформа. Интерфейсы между пользователем и приложением состоят из диалогов, посредством которых пользователь получает доступ к информации системы, и неких выходных сообщений и коммуникационных путей, которые система использует для документирования или передачи этой информации.

Для системы ввода заказов из нашего примера, которая должна быть объединена с уже существующей информационной системой компании, границы не столь очевидны. Аналитик должен определить, будут ли данные использоваться совместно с другими приложениями, должно ли новое приложение распределяться по разным хостам и клиентам, а также кто будет пользователем. Например, должен ли персонал, занятый в производстве, иметь интерактивный доступ к заказам на покупку? Обеспечивается ли контроль качества или функции аудита? Будет ли система выполняться на компьютере-мейнфрейме или на новом компьютере-клиенте? Должны ли предоставляться специальные отчеты?

Выявление акторов является ключевым аналитическим этапом в анализе проблемы. Ответы на следующие вопросы помогут их обнаружить.

- Кто будет поставлять, использовать или удалять информацию из системы?
- Кто будет управлять системой?
- Кто будет осуществлять сопровождение системы?
- Где будет использоваться система?
- Откуда система получает информацию?
- Какие внешние системы будут взаимодействовать с системой?

Имея ответы на эти вопросы, аналитик может создать блок-схему, описывающую границы системы, пользователей и другие интерфейсы. На рис. 4.5 представлена новая система ввода заказов на покупку и ее окружение.

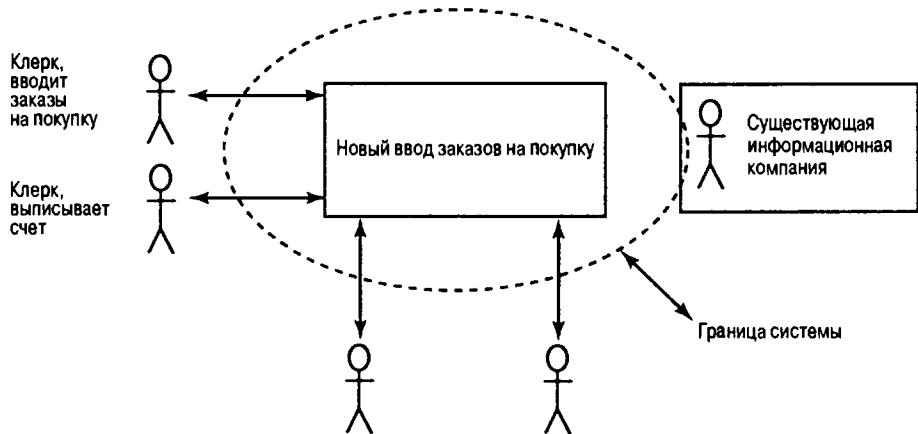


Рис. 4.5. Система и ее окружение

Точечная линия иллюстрирует границу системы для предлагаемого решения. Из рисунка видно, что основная часть нового приложения будет развернута в новой системе ввода заказов на покупку, но часть кода решения должна разрабатываться и разворачиваться в уже существующей унаследованной системе.

Этап 5. Выявление ограничений, налагаемых на решение

Ограничения уменьшают степень свободы, которой мы располагаем при предложении решения.

Перед тем как предпринимать продиктованные благими намерениями и стоящие больших денег усилия по "революционизированию" состояния дел в области ввода заказов на покупку, необходимо остановиться и рассмотреть ограничения, которые будут наложены на решение. Мы будем определять ограничение следующим образом.

Ограничение уменьшает степень свободы, которой мы располагаем при предложении решения.

Каждое ограничение может значительно сузить нашу возможность создать предполагаемое решение. Следовательно, в процессе планирования необходимо тщательно изучить все ограничения. Многие из них могут даже заставить нас пересмотреть изначально предполагавшийся технологический подход.

Необходимо учитывать, что существуют различные источники ограничений (экономические, технические, политические и т.д.). Ограничения могут быть заданы

еще до начала работы (“Никакой новой аппаратуры!”), но может получиться, что нам действительно придется их выявлять.

Чтобы их выявить, полезно знать, на что следует обратить внимание. В табл. 4.4 указаны возможные источники системных ограничений. Перечисленные в таблице вопросы помогут выявить большую часть ограничений. Часто полезно получить объяснение ограничения, как для того, чтобы убедиться, что вы поняли его назначение, так и для того, чтобы можно было обнаружить (если такое произойдет), что данное ограничение больше не применимо к вашему решению.

Таблица 4.4. Возможные источники ограничений системы

Источник	Образцы вопросов
Экономический	<ul style="list-style-type: none"> ■ Какие финансовые или бюджетные ограничения следует учесть? ■ Существуют ли соображения, касающиеся себестоимости и ценообразования? ■ Существуют ли вопросы лицензирования?
Политический	<ul style="list-style-type: none"> ■ Существуют ли внешние или внутренние политические вопросы, влияющие на потенциальное решение? ■ Существуют ли проблемы в отношениях между подразделениями?
Технический	<ul style="list-style-type: none"> ■ Существуют ли ограничения в выборе технологий? ■ Должны ли мы работать в рамках существующих платформ или технологий? ■ Запрещено ли использование любых новых технологий? ■ Должны ли мы использовать какие-либо закупаемые пакеты программного обеспечения?
Системный	<ul style="list-style-type: none"> ■ Будет ли решение создаваться для наших существующих систем? ■ Должны ли мы обеспечивать совместимость с существующими решениями? ■ Какие операционные системы и среды должны поддерживаться?
Эксплуатационный	<ul style="list-style-type: none"> ■ Существуют ли ограничения информационной среды или правовые ограничения? ■ Юридические ограничения? ■ Требования безопасности?
График и ресурсы	<ul style="list-style-type: none"> ■ Какими другими стандартами мы ограничены? ■ Определен ли график? ■ Ограничены ли мы существующими ресурсами? ■ Можем ли мы привлекать работников со стороны? ■ Можем ли мы увеличить штат? Временно? Постоянно?

После того как ограничения выявлены, некоторые из них станут требованиями к новой системе (“использовать MRP-систему, предлагаемую поставщиком нашей нынешней системы учета”). Другие ограничения будут оказывать влияние на ресурсы и планы реализации. Именно при анализе проблемы необходимо выявить потенциальные источники ограничений и понять, какое влияние каждое ограничение окажет на область возможных решений.

Возвратимся к нашему примеру. Ограничения, которые могут налагаться на новую систему ввода заказов на покупку, представлены в табл. 4.5.

Таблица 4.5. Ограничения, налагаемые на систему ввода заказов на покупку

Источник	Ограничение	Объяснение
Эксплуатационный	Копия данных заказа на покупку должна оставаться в унаследованной базе данных в течение одного года	Риск потери данных слишком высок; нам необходимо работать параллельно в течение года
Системы и операционные системы	Приложение должно занимать на сервере не более 20 мегабайт	Количество доступной памяти сервера ограничено
Средства, выделенные на оборудование	Система должна быть разработана на существующем сервере или хосте; можно предложить новое клиентское аппаратное обеспечение для пользователей	Сокращение издержек и поддержка существующих систем
Средства, выделенные на оплату труда персонала	Фиксированный штат; не привлекать работников со стороны	Фиксированные расходы на зарплату по отношению к текущему бюджету
Технологические требования	Должна использоваться новая объектно-ориентированная методология	Мы надеемся, что эта технология повысит производительность и надежность программного обеспечения

Заключение

После завершения этого этапа можно со всей ответственностью заявить, что мы достигли следующего.

- Хорошо поняли решаемую проблему и лежащие в ее основе причины.
- Выявили заинтересованных лиц, чье коллективное суждение в конце концов будет определять успех или неудачу нашей системы.
- Выяснили, где, по всей видимости, должны находиться границы решения.
- Поняли существующие ограничения и определили степень свободы, которой мы обладаем при решении проблемы.

Далее...

Основываясь на полученных знаниях, мы можем перейти к рассмотрению двух специальных методов анализа проблем, которые можно применять в определенных предметных областях. В главе 5 мы рассмотрим *бизнес-моделирование*, метод, который можно применить для IS/IT-приложений. В главе 6 будет описано *системное проектирование* систем, интенсивно использующих программное обеспечение — наиболее подходящий метод анализа проблем при создании приложений в области встроенных систем.



Что касается ISV-приложений (разрабатываемых независимыми производителями программного обеспечения), методы анализа проблемы для них, как правило, заключаются в следующем.

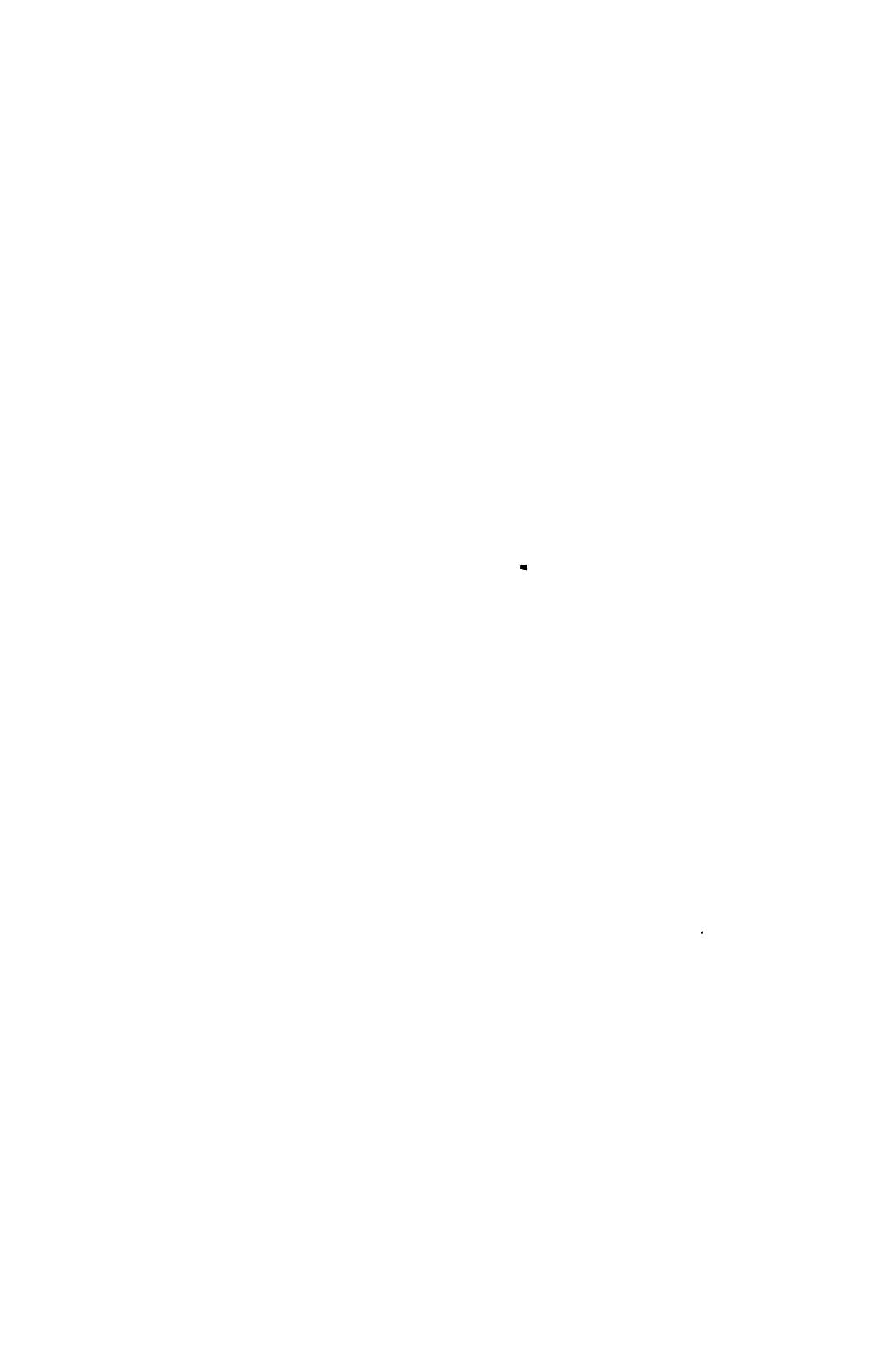
- Выявление имеющихся на рынке возможностей и существующих ниш
- Определение классов потенциальных пользователей и их потребностей
- Изучение демографии потенциальной пользовательской базы
- Оценка потенциального спроса, цен и эластичности спроса
- Понимание стратегии продаж и организации каналов сбытов

Это, безусловно, интересные вопросы, но, чтобы справиться с намеченными в данной книге задачами, мы не будем подробно обсуждать их. Однако, как вы сможете убедиться, предлагаемые в последующих главах профессиональные приемы могут с равным успехом применяться и к этому классу приложений.

Примечание. При написании этой книги сложнее всего было представить разнообразие методов для создания необходимых профессиональных навыков. Не существует метода, который можно применять во всех ситуациях, а двух одинаковых ситуаций не бывает.

В предыдущих главах основное внимание уделялось общефилософскому подходу к анализу проблемы, который, по всей видимости, можно применять практически для любых систем. Однако проблема выбора применяемого метода далее становится более острой. В последующих двух главах мы опишем моделирование бизнес-процессов и системное проектирование, а затем продолжим определять разнообразные методы в части 2, "Понимание потребностей пользователей". В этой части будет представлен широкий спектр методов, позволяющих понять потребности заинтересованных лиц и пользователей по отношению к *создаваемой системе*.

Однако мы считаем важным подчеркнуть, что описанные в данной книге методы – от анализа проблем до "мозгового штурма" – можно использовать на различных этапах процесса разработки, а не только на том этапе, применительно к которому мы решили их описать. Например, команда может использовать анализ проблемы как на этапе постановки проблемы системы ввода заказов на покупку, так и для решения технической проблемы на этапе реализации данной системы. Аналогично команда может использовать "мозговой штурм" как для выявления возможных причин возникновения проблемы на этапе анализа проблемы, так и для определения возможных новых функций системы, как это делается в главе 11. Мы не будем пытаться описать все ситуации, в которых будет применяться определенный метод. Вместо этого мы уделим основное внимание тому, чтобы команда овладела этими навыками, могла добавить в свой арсенал эти методы и использовать их в нужный момент.



Глава 5

Моделирование бизнес-процессов

Основные положения

- Для анализа проблем в среде IS/IT наиболее подходящим является метод моделирования бизнес-процессов.
- Модель бизнес-процесса помогает нам при определении систем и их приложений.
- Модель прецедентов бизнес-процесса, состоящая из акторов и прецедентов, является моделью предполагаемых функций предприятия.
- Модель объектов бизнес-процесса описывает сущности, которые обеспечивают функциональные возможности для реализации прецедентов бизнес-процесса, а также их взаимодействие.

В среде информационных технологий и систем (IS/IT), изобилующей сложностями бизнес-процесса, прежде чем пытаться определить конкретные проблемы, достойные решения, необходимо понять некоторые моменты самого бизнес-процесса. Эта среда состоит не просто из одного–двух пользователей и их интерфейса с компьютером, а из организаций, предприятий, департаментов, глобальных и корпоративных сетей, заказчиков, пользователей, готовой продукции, систем управления и т.д.

Кроме того, даже при создании отдельного приложения необходимо помнить о многосторонности среды, в которой оно будет функционировать. Возможно, нам удастся достичь понимания, задавая правильные вопросы, но, как правило, для частного случая должен существовать некий метод, позволяющий добиться большего, чем в более общем случае.

В среде IS/IT полезно иметь метод, с помощью которого можно ответить на следующие вопросы.

- Зачем вообще создается система?
- Где она должна размещаться?
- Как определить, какие функциональные возможности лучше всего разместить в отдельно взятой системе?
- Когда следует применять этапы ручной обработки?

- Когда для решения проблемы следует рассматривать возможность реструктуризации самой организации?

К счастью, существует метод, который идеально подходит для решения этой конкретной проблемы. Это – *моделирование бизнес-процесса*.

Цели моделирования бизнес-процесса

В данной книге мы можем воспринимать термины “бизнес” и “бизнес-моделирование” в самом широком смысле. Например, бизнес может заключаться в разработке программного обеспечения или производстве сварочных роботов. Можно моделировать некоммерческое предприятие, сервисную организацию, процесс внутри подразделения или внутренний рабочий процесс.

В любом случае цель бизнес-моделирования двояка.

- Разобраться в структуре и динамике организации
- Удостовериться в том, что заказчики, конечные пользователи и разработчики имеют одинаковое понимание организации

Бизнес-моделирование дает команде возможность понять, где приложения программного обеспечения могут улучшить производительность данного предприятия, и помогает определить требования к этим приложениям.

Использование методов инженерии программного обеспечения для моделирования бизнес-процессов

Для моделирования бизнес-процессов, безусловно, можно применить множество методов. Но мы, как разработчики программного обеспечения, имеем в своем распоряжении множество разнообразных средств и методов, которые уже использовались для моделирования программного обеспечения. Мы знаем, как моделировать сущности (объекты и классы), отношения (зависимости, ассоциации и т.д.), сложные процессы (последовательности действий, переходы состояний, события, условную зависимость и т.д.) и другие конструкции, естественным образом возникающие при разработке программного приложения.

При правильном выборе метода моделирования бизнес-процесса некоторые рабочие продукты, а именно прецеденты и модели объектов, в дальнейшем будут полезны при создании решения.

Если мы сможем применить эти же методы для моделирования бизнес-процесса, нам удастся использовать в обоих случаях одни и те же понятия. Например, сущность бизнес-области “расчетный лист” можно связать с программной сущностью “строка платежной ведомости”. Если нам удастся использовать одинаковые или очень близкие методы как при анализе проблемы, так и при конструировании решения, это позволит применять общие рабочие продукты.

Выбор подходящего метода

Исторически сложилось так, что разработанные в области программного обеспечения методы моделирования привели к возникновению новых способов визуализации организации. Поскольку методы объектно-ориентированного визуального моделирования применяются практически во всех новых проектах разработки программного обеспечения, естественно попытаться использовать аналогичные методы в области моделирования бизнес-процессов. Эта методология хорошо описана в работе Айвара Джейкобсона (Jacobson), Марии Эрикссон (Ericsson) и Агнеты Джейкобсон (Jacobson) (1994) и др.

В 1980–90-х годах наблюдалось быстрое развитие как методов моделирования бизнес-процессов, так и методологий разработки программного обеспечения. Но увы, они очень отличались! За основу брались различные объектно-ориентированные (ОО) методы и нотации, разработанные различными экспертами в области программного обеспечения и специалистами по методологии.¹ К счастью, методологические “войны” завершились, и в отрасли воцарился стандарт моделирования систем, интенсивно использующих программное обеспечение, — *Унифицированный язык моделирования* (UML, Unified Modeling Language).

Унифицированный язык моделирования (UML)

В конце 1997 года в качестве отраслевого стандарта был принят графический язык “визуализации, спецификации, конструирования и документирования артефактов систем, интенсивно использующих программное обеспечение” (Booch, Jacobson and Rumbaugh). Язык UML² предлагает набор элементов моделирования, обозначений, отображений и правил использования, которые можно применять при разработке программного обеспечения. По UML можно также использовать для моделирования систем и бизнес-процессов. Данная книга не содержит руководства по UML. (Для этого можно обратиться к следующим трем книгам по UML: Booch, Rumbaugh and Jacobson. *The Unified Modeling Language User Guide* (1999); Jacobson, Booch and Rumbaugh. *The Unified Software Development Process* (1999); Rumbaugh, Booch and Jacobson. *The Unified Modeling Language Reference Manual* (1998).) Но в данном разделе мы будем использовать некоторые основные концепции UML и на их основе будем строить дальнейшее изложение.

¹ Среди ОО-методов можно назвать метод Буча, разработанный Грейди Бучем (Grady Booch) из корпорации Rational Software; Object Modeling Technique (OMT) Джеймса Рамбо (James Rumbaugh), работавшего тогда в компании General Electric; Responsibility-Driven Design Ребекки Уирфс-Брок (Rebecca Wirfs-Brock) из компании Текtronикс; Object Oriented Software Engineering Айвара Джейкобсона (Ivar Jacobson), работавшего тогда в шведской компании Objectory; метод Коуда-Йордана Питера Коуда (Peter Coad) и Эда Йордана (Ed Yourdon); а также полдюжины других.

² Язык UML, версия 1.1, был принят в 1997 году международной организацией Object Management Group (OMG) после того, как создатели его исходной версии из корпорации Rational Software (Буч (Booch), Джейкобсон (Jacobson), Рамбо (Rumbaugh)) основали широкий отраслевой консорциум и включили в язык концепции других методов, обеспечив также обратную связь и процесс пересмотра.

Моделирование бизнес-процесса с использованием концепций UML

Одной из целей моделирования бизнес-процесса является создание такой его модели, которой можно руководствоваться при разработке приложения. Для этого можно использовать две основные модельные конструкции: *модель прецедентов бизнес-процесса* (*business use-case model*) и *модель объектов бизнес-процесса* (*business object model*).

Модель прецедентов бизнес-процесса представляет собой модель предполагаемых функций бизнес-единицы и используется в качестве исходной информации для выявления ролей и взаимосвязей в организации. Она состоит из акторов (пользователей и систем, которые взаимодействуют с данной бизнес-единицей) и прецедентов (последовательностей событий, посредством которых акторы взаимодействуют с ее элементами, чтобы выполнить нужную им работу). Акторы и прецеденты совместно описывают, кто участвует в деятельности бизнес-единицы и как эта деятельность протекает. На рис. 5.1 изображена модель прецедентов бизнес-процесса. Обратите внимание, что используемые для представления прецедентов овальные пиктограммы содержат слэш, означающий, что это прецедент бизнес-уровня, а не системного уровня.³

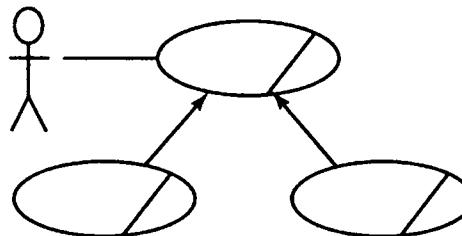


Рис. 5.1. Модель прецедентов бизнес-процесса

Акторы в модели прецедентов бизнес-процесса представляют внешние по отношению к бизнес-единице роли (например, служащие и клиенты), а прецеденты представляют процессы. Приведем несколько примеров прецедентов бизнес-процесса.

- Предоставление служащему электронной версии расчетного листа.
- Встреча с заказчиком для согласования сроков контракта.

Ниже перечислены возможные акторы бизнес-процесса.

1. Клиент
2. Служащий
3. Разработчик программного обеспечения

Модель объектов бизнес-процесса описывает сущности (подразделения, платежные чеки, системы) и то, как они взаимодействуют в процессе создания функциональных возможностей, необходимых для осуществления прецедентов бизнес-процесса. На рис. 5.2 представлена модель объектов бизнес-процесса. Пиктограмма в виде кружочка с актором внутри представляет

³ Данная пиктограмма является одним из многих стандартных стереотипов UML. Более подробно о пиктограммах моделирования см. Rational Software Corporation (1999).

некоего сотрудника, который находится внутри бизнес-процесса, например служащего, обрабатывающего платежные ведомости, или системного администратора. Кружок со слэшем представляет некую сущность бизнес-процесса или нечто, что сотрудники производят (например, платежную ведомость, шарикоподшипник или файл исходного кода).⁴

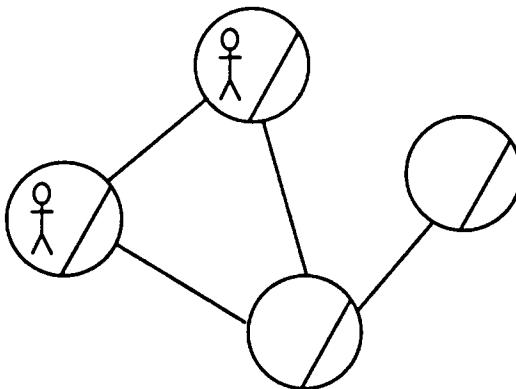


Рис. 5.2. Модель объектов бизнес-процесса

Модель объектов бизнес-процесса также включает в себя реализации прецедентов бизнес-процесса, которые показывают, как эти прецеденты “осуществляются” при взаимодействии сотрудников и сущностей бизнес-процесса. Чтобы отразить существование в организации групп или подразделений, можно сгруппировать сотрудников и бизнес-сущности в организационные единицы.

Модель прецедентов и модель объектов бизнес-процесса в совокупности обеспечивают исчерпывающее представление о том, как функционирует бизнес-единица, и позволяют команде разработчиков сосредоточить внимание на тех областях, для которых можно предложить системы, способные повысить общую эффективность предприятия. Они также помогают команде понять, какие изменения нужно будет внести в сами бизнес-процессы, чтобы успешно реализовать новую систему.

От моделей бизнес-процесса к модели системы

Одно из преимуществ данного подхода к моделированию бизнес-процесса состоит в простоте описания зависимостей между моделями бизнес-процесса и системы. Это повышает производительность процесса разработки программного обеспечения, а также помогает удостовериться в том, что разрабатываемая система решает реальные бизнес-потребности (рис. 5.3).

Преобразование одной модели в другую можно кратко описать следующим образом.

- Сотрудники предприятия становятся акторами разрабатываемой системы.
- Описанные для сотрудников предприятия варианты поведения можно автоматизировать; это помогает нам выявить прецеденты системы и определить необходимые функциональные возможности.

⁴ См. Rational Software Corporation (1999).

- Сущности бизнес-процесса — это то, в поддержке чего нам должна помочь система, поэтому они помогают нам выявить классы сущностей при анализе модели системы.

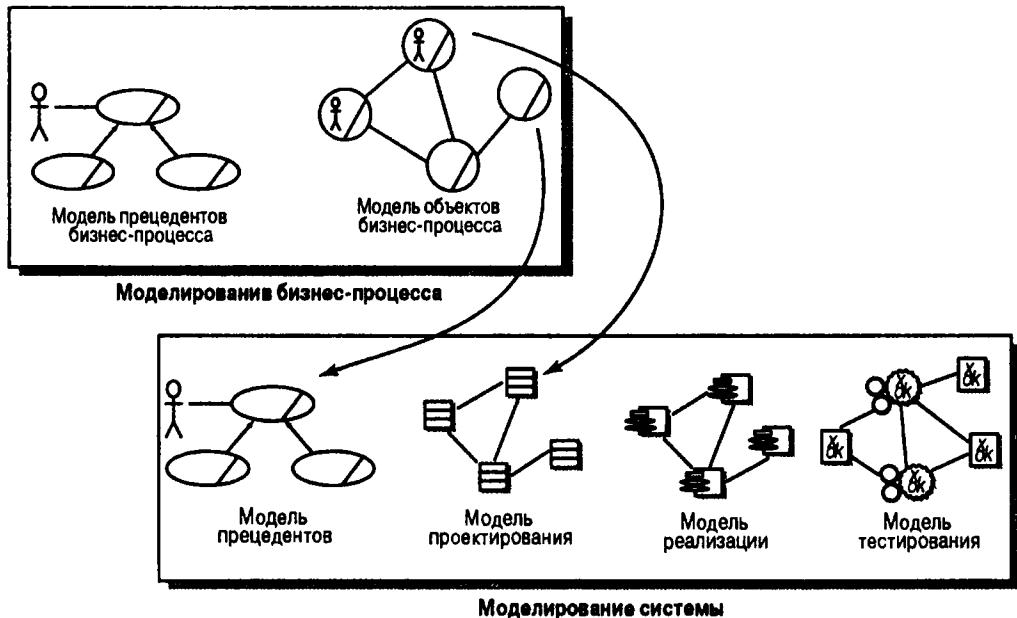


Рис. 5.3. Модели бизнес-процесса и системы

Модель бизнес-процесса и последующее ее преобразование упрощают процесс перехода от понимания бизнес-процесса и существующих в нем проблем к созданию приложений, способствующих их решению.

Когда использовать моделирование бизнес-процесса

Моделирование бизнес-процесса не является универсальным методом, который мы рекомендуем применять в процессе каждой разработки программного обеспечения. Бизнес-модели, в основном, нужны в сложной многомерной прикладной среде, когда множество людей непосредственно вовлекаются в использование системы. Например, при создании дополнительной функции для существующего телекоммуникационного коммутатора моделировать бизнес-процесс не понадобится. Но при создании системы ввода заказов для компании GoodsAreUs моделирование бизнес-процесса позволит получить определенные преимущества при проведении анализа проблемы.

Заключение

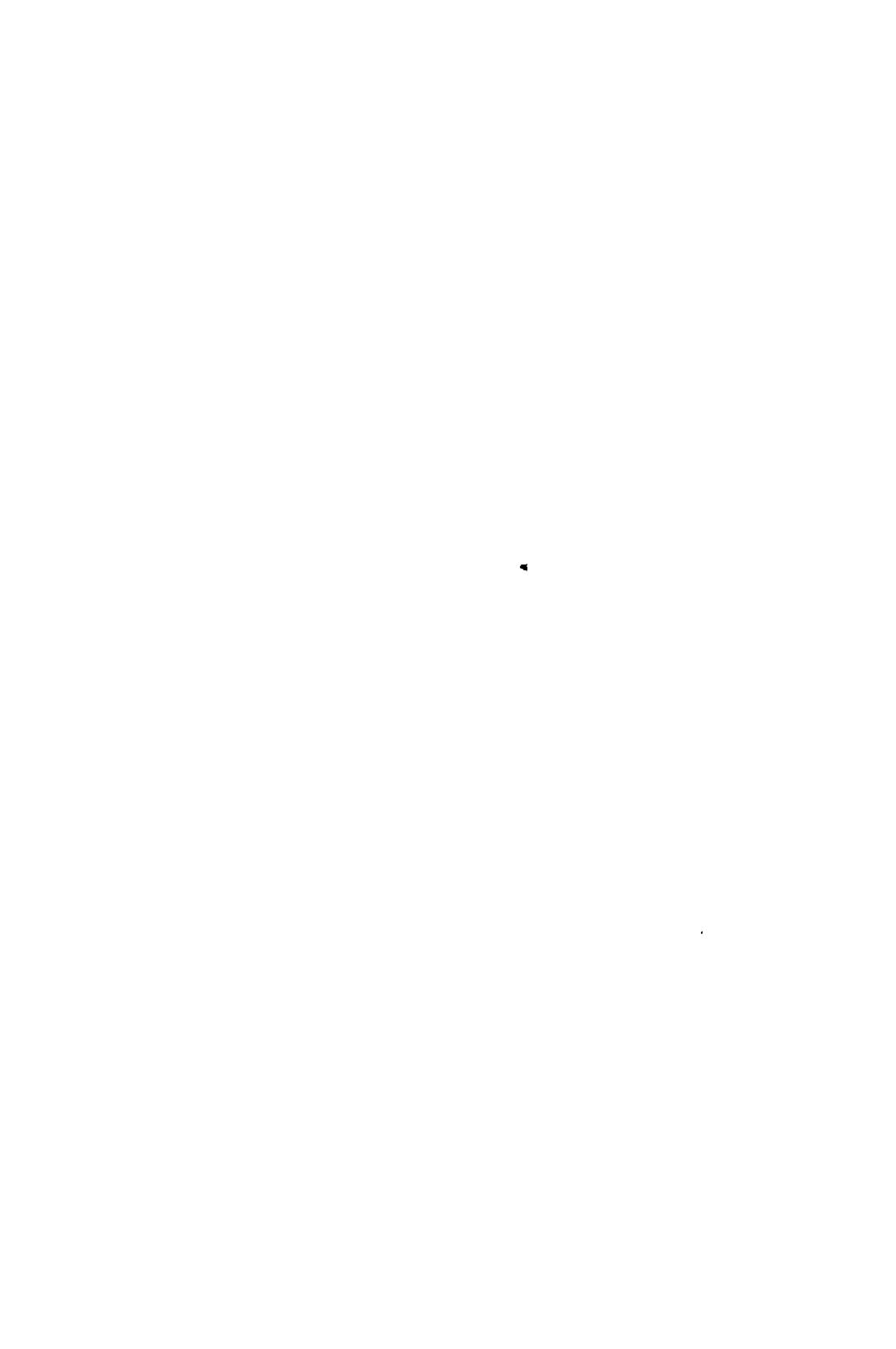
В этой главе описан специальный метод анализа проблем — моделирование бизнес-процесса. При этом обсуждались следующие вопросы.

- Зачем нужно моделировать бизнес-процесс
- Как, используя язык UML, преобразовать разработанные в инженерии программного обеспечения методы и применить их в моделировании бизнес-процесса
- Основные артефакты моделирования бизнес-процесса — модель прецедентов бизнес-процесса и модель объектов бизнес-процесса
- Как, используя модель бизнес-процесса, перейти к определению программных приложений и выявить требования к программному обеспечению

Далее...

В следующей главе мы рассмотрим инженерию систем, интенсивно использующих программное обеспечение. Это еще один метод анализа проблем, который поможет придать форму приложениям, разрабатываемым для систем встроенного типа.





Глава 6

Инженерия систем, интенсивно использующих программное обеспечение

Основные положения

- Системная инженерия – это наиболее подходящий для разработки встроенных систем метод анализа проблем.
- Системная инженерия помогает понять требования, предъявляемые к программным приложениям, выполняемым внутри создаваемой системы.
- Нисходящий процесс разработки требований гарантирует, что каждое системное требование выполняется определенной подсистемой или совокупностью взаимодействующих подсистем.
- На современном этапе зачастую необходимо оптимизировать затраты на разработку программного обеспечения, а не затраты на технические компоненты системы.

В главе 5 мы рассмотрели моделирование бизнес-процессов. Этот метод применяется для анализа проблем в области IS/IT-приложений. Он помогает определить, какие приложения следует создать и где они должны выполняться, принимая во внимание вычислительную среду компании, расположение ее подразделений, зданий, политических и физических конструкций. Другими словами, он помогает определить, *почему* и *где* должно появиться некое приложение. В процессе анализа происходит постепенный переход из *области проблемы* к первому приближению *области решения*, где в одном или нескольких приложениях будут существовать функциональные возможности, решающие данную проблему и удовлетворяющие конечную потребность пользователя.

Однако в случае встроенных систем область проблемы и область решения выглядят совершенно иначе. Вместо подразделений и процессов эти области состоят из разъемов, блоков питания, стоек с оборудованием, электронных и электрических компонентов, гидравлических и жидкостных приборов, а также других программных, механических и оптических подсистем и т.п. В данном случае моделирование бизнес-процесса не может принести заметной пользы. Следует найти другой метод, который поможет ответить на вопросы *почему* и *где*. Этим и занимается *системная инженерия* (*системное проектирование* – *system engineering*).

Что такое системная инженерия

Согласно определению Международной ассоциации системного проектирования (International Council on Engineering Systems, INCOSE 1999)

Системная инженерия – это междисциплинарный подход и соответствующие ему средства, которые позволяют создавать успешно работающие системы. Это дисциплина, которая занимается выявлением на ранних этапах цикла разработки потребностей клиента и необходимых функциональных возможностей, а также документированием требований. За этим должны следовать разработка технического проекта как единого целого и проверка правильности создаваемой системы. В рамках системной инженерии рассматриваются

- функционирование;
- рабочие характеристики;
- тестирование;
- производство;
- затраты и сроки;
- обучение и поддержка;
- передача.

Системная инженерия объединяет действия различных групп специалистов в командные действия, образующие структурированный процесс разработки, от концепции к производству и функционированию. Системная инженерия учитывает как потребности бизнес-процесса, так и технические потребности всех клиентов с целью предоставить качественный продукт, удовлетворяющий потребности пользователя.

Как можно убедиться, определение слишком сложное. Но мы не будем полностью обсуждать данный подход в книге по требованиям к программному обеспечению! (Более подробно о системной инженерии см. Rechtin (1997).)

В рамках данной книги мы используем системную инженерию в качестве метода анализа проблемы, чтобы понять потребности проблемной области и требования, которые должны быть предъявлены к решению. Иными словами, в нашем случае системная инженерия поможет понять требования, которые будут предъявлены ко всем программным приложениям, выполняемым внутри системы-решения (независимо от того, выполняются ли они встроенным микропроцессором или системой UNIX в рамках всемирной телекоммуникационной системы).

Основные принципы системной инженерии

Системная инженерия предлагает восемь основных принципов.

Если мы собираемся использовать системную инженерию в качестве метода анализа проблемы, основные принципы дисциплины должны указать нам, как следует действовать, анализируя проблему с целью выработки требований. Рабочая группа по применению методов системной инженерии INCOSE (INCOSE Systems Engineering Practices working group, 1993) сформулировала восемь основных принципов.

- Знание проблемы, клиента и потребителя.
- Использование основанных на потребностях критериев эффективности для принятия системных решений.
- Задание требований и управление ими.
- Выявление и оценка альтернатив при выработке решения.
- Верификация и проверка правильности требований, а также функционирования решения.
- Обеспечение целостности системы.
- Использование согласованного и документированного процесса.
- Организация действий согласно плану.

В этом списке перечислены некоторые основные принципы системной инженерии. Но в данной дисциплине существует раздел, в основе которого лежит иной процесс – процесс последовательной декомпозиции сложных систем на более простые.

Декомпозиция сложных систем

С помощью процесса декомпозиции сложная проблема (система) (рис. 6.1) разбивается на две меньшие проблемы – подсистемы (рис. 6.2). Эти подсистемы можно обсудить, спроектировать и изготовить, а затем объединить и получить систему, являющуюся решением. Разделы инженерии, посвященные вопросам поддержки декомпозиции систем, также указаны в приведенном выше определении, где речь идет о понимании операционных характеристик, возможности изготовления и тестирования и т.д.

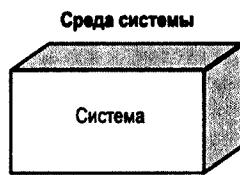


Рис. 6.1. Система и ее окружение

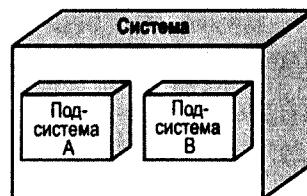


Рис. 6.2. Система, состоящая из двух подсистем

Процесс декомпозиции (или последовательного разбиения) производится до тех пор, пока специалист по системам не достигнет нужных результатов, что подтверждается конкретными количественными оценками, свойственными каждой области системной инженерии. Как правило, полученные после первого разбиения подсистемы подвергаются дальнейшей декомпозиции (рис. 6.3). Для наиболее сложных систем процесс продолжается достаточно долго, и в результате получается большое количество подсистем. (Говорят, что истребитель F22, например, состоит из 152 подсистем.)

Показателями того, что дело сделано и сделано “правильно”, являются следующие критерии.

- Распределение и разбиение функций оптимизировано так, чтобы добиться достижения общей функциональности системы с минимальными затратами и максимальной гибкостью.

- Каждая подсистема может быть определена, спроектирована и построена небольшой или, в крайнем случае, средней командой.

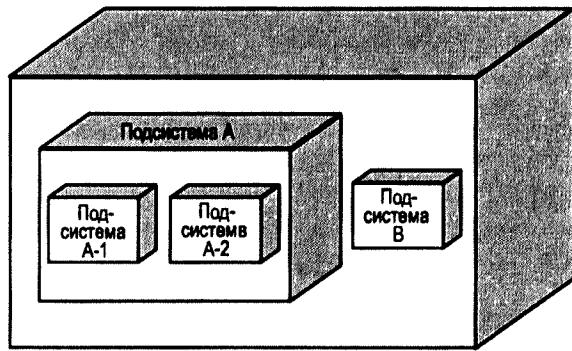


Рис. 6.3. Подсистема, состоящая из двух подсистем

- Каждая подсистема может быть изготовлена в рамках физических ограничений и технологий существующих производственных процессов.
- Каждую подсистему можно надежно протестировать, причем существует возможность подходящих установок и сопряжений, имитирующих интерфейсы с другими подсистемами.
- Соответствующее внимание уделено физической стороне (размеру, весу, размещению и распределению подсистем), которая также оптимизирована исходя из общесистемных соображений.

Размещение требований в системной инженерии

Нынешний процесс разработки требований распределяет функциональные возможности системы по подсистемам.

Даже если предположить, что с помощью системной инженерии можно определить требования к системе, проблема управления требований все еще остается нерешенной. Что делать с полученными подсистемами? Какие требования должны быть предъявлены к ним? В некоторых случаях процесс заключается в распределении требований системного уровня по подсистемам ("подсистема В будет выполнять алгоритм расчета скорости ветра и осуществлять непосредственное управление дисплеем лобового стекла"). *Нынешний процесс разработки требований* предназначен для того, чтобы гарантировать, что все системные требования выполняются некой определенной подсистемой или совокупностью взаимодействующих подсистем.

Производные требования

Иногда создается совершенно новый тип требований, предъявляемых к подсистемам — *производные требования*. Как правило, существует два класса производных требований.

- Требования к подсистемам** – это требования, которые необходимо предъявить к самим подсистемам, но они не обязательно доставляют осязаемый результат конечному пользователю (“подсистема А должна выполнять алгоритм, вычисляющий скорость ветра относительно самолета”).
- Требования к интерфейсам**, которые могут возникнуть при взаимодействии подсистем для достижения общего результата. Эти подсистемы должны совместно использовать данные, мощность или некий алгоритм вычисления. В этих случаях создание подсистем приводит также к созданию *интерфейсов* между ними (рис. 6.4).



Рис. 6.4. Интерфейс между двумя подсистемами

Однако являются ли эти производные требования “настоящими”? Можно ли их трактовать как остальные требования? По всей видимости, они не соответствуют определениям, данным в главе 2 (хотя они могут вполне соответствовать определениям ограничений проектирования, которые будут даны далее).

Важно понимать, что эти требования, хотя они и важны для успеха проекта, являются производными от процесса декомпозиции. Альтернативная декомпозиция может привести к созданию других производных требований, так что эти требования не являются полноправными участниками процесса в том смысле, что не отражают требования, поступившие от заказчика. Но, с точки зрения поставщика подсистемы, они являются полноправными, так как отражают требования, предъявляемые заказчиком (в роли которого в данном случае выступает разработчик системы).

Магического ответа на все случаи жизни не существует. Как трактовать эти требования – зависит от роли команды разработчиков в проекте, декомпозиции системы и других технологических факторов. Таким образом, важно знать, “как мы сюда попали”, и трактовать требования соответственно обстоятельствам. Важно понимать, что спецификация производных требований будет в конечном счете влиять на способность системы выполнять ее работу, а также на удобство эксплуатации и робастность системы.

“Тихая” революция

В постиндустриальном обществе “ум” прибора переместился из аппаратных компонентов в компоненты программного обеспечения.

Системная инженерия (системное проектирование) традиционно применялась преимущественно к физическим системам, таким как самолеты, энергогенераторы, потребляющие энергию приборы и т.д. Однако за последние 20 лет (или около того) произошла “тихая революция” в инженерии сложных систем. Постепенно системы и приборы в различных отраслях становились все более “умными”. Все большая часть необходимых функциональных возможностей стала размещаться в подсистемах программного обеспечения, а не в аппаратных компонентах. Причина в том, что программное обеспечение более гибкое и многие алгоритмы измерения, учета, количественного анализа и обнару-

жения гораздо проще (или, по крайней мере, гораздо дешевле) реализовать в программном обеспечении, чем в аппаратных компонентах. Еще важнее то, что в этом случае их гораздо проще изменять.

Таким образом, в постиндустриальном обществе внутренняя "разумность" прибора переместилась из компонентов аппаратуры, где она реализовывалась прежде наряду с электрическими, электронными, механическими системами и даже системами физической химии, в компоненты программного обеспечения, где она реализуется в программном обеспечении или программно-аппаратных средствах.

Столкновение поколений: седобородые встречаются с молодыми и самонадеянными

На протяжении нескольких десятилетий специалисты по системному проектированию были одними из самых уважаемых в отрасли. Покрытые шрамами битв и проверенные временем многие из них являлись специалистами в отдельных дисциплинах, таких как механика или электроника, и, кроме того, были одними из лучших универсалов команды. Они были свидетелями величайших неудач и испытывали множество триумфов. Теперь, став старше и мудрее, они в совершенстве знают некую прикладную область (радиотехнику, самолетостроение и т.п.), а также хорошо представляют себе различные технические, экономические и политические стороны реализации.



По внезапно какие-то люди вторглись в их владения. Эти пришельцы — программисты или, в лучшем случае, специалисты по инженерии программного обеспечения, — были относительно неопытны в том, что касалось сложных систем, но они могли с помощью языка ассемблера заставить микропроцессор звенеть. Кроме того, казалось, что они сделаны из другого генетического материала или, по крайней мере, относятся к другому поколению, что создало дополнительные сложности в отрасли.

На некоторое время системным проектировщикам удалось закрепить за собой право на разбиение системы и размещение функций. Но во многих отраслях технологии программного обеспечения все же взяли верх, и над системным проектированием возобладала, по крайней мере частично, потребность в создании гибкого функционально полноценного программного обеспечения системы в целом. Для этого перехода существовал ряд веских технических причин. Со временем стало очевидно следующее.

- Программное обеспечение (а не аппаратные компоненты) будет определять итоговые функциональные возможности системы, а также ее успех у конечного потребителя и на рынке.
- На программное обеспечение (а не на аппаратные компоненты) будет затрачена основная часть средств, выделенных на исследование и разработку системы.
- Именно программное обеспечение (а не аппаратные компоненты) будет в конечном счете определять, когда система попадет на рынок.
- Программное обеспечение (а не аппаратные компоненты) будет вбирать в себя большую часть изменений, которые будут возникать во время разработки системы, и будет эволюционировать несколько последующих лет, чтобы удовлетворить меняющиеся потребности системы.

И, что самое удивительное,

- вклад затрат на разработку и сопровождение программного обеспечения (с учетом амортизации за весь период жизни продукта) в общие производственные затраты становится сравнимым с вкладом затрат на аппаратные компоненты, а иногда даже превышает его.

В настоящее время во многих системах необходимо оптимизировать возможности их программного обеспечения, а не аппаратных компонентов.

Последнее означает, что теперь систему следует оптимизировать, исходя из затрат на разработку, сопровождение, эволюцию и усовершенствование содержащегося в системе программного обеспечения, а не из затрат на аппаратные компоненты или производство. Это существенно меняет правила игры. Теперь системное проектирование систем, использующих встроенное программное обеспечение, должно осуществляться с учетом используемых компьютеров. Это означает, что необходимо сделать следующее.

- Максимизировать способность системы выполнять программу путем предоставления более чем достаточных вычислительных ресурсов даже за счет увеличения стоимости продаваемых изделий, добавляя дополнительные микропроцессоры, оперативную память, ПЗУ, массовую память, полосу частот или любые другие ресурсы, которые требуются системе для выполнения ее программы.
- Обеспечить адекватные коммуникационные интерфейсы между подсистемами и гарантировать, что выбранный коммуникационный механизм (Ethernet, Fireware, последовательный порт или одиночная линия связи) можно расширить с помощью дополнительного программного обеспечения, а не аппаратных компонентов.

В свою очередь это изменение повлияло на задачи управления требованиями в двух направлениях.

- Каждый из этих параметров будет создавать новые требования, которые должна выполнять система аппаратного обеспечения для построения успешного решения.
- Основная масса проблем требований переходит к программному приложению.

К счастью, по крайней мере, последнее является темой данной книги, и мы надеемся хорошо подготовить вас к решению этой конкретной проблемы.

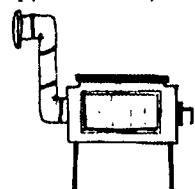
Как избежать проблемы создания системы типа “печной трубы”

Итак, для того чтобы справиться со сложностью, мы решили использовать подход, в котором система составляется из подсистем. В противном случае придется иметь дело с невероятно сложными системами с непредсказуемым поведением, которые никто не может понять, с проектом, основанным на совместно используемых функциях, плохом разбиении и последовательном коде, который невозможно распутать. Разбиение системы и использование методов системной инженерии выглядит вполне оправданно.

Как это отражается на управлении требованиями? Когда будет подведен окончательный итог, обнаружится, что требований подсистем гораздо больше, чем тех (внешних) требований, которые влияют на поведение системы в среде пользователя. Мы затратим гораздо больше сил на выявление требований к подсистемам, расстановку приоритетов и

управление ими, чем на те требования, которые оказывают влияние на конечного потребителя. Это уже нельзя считать позитивным.

Что случится, если разбиение системы проведено не совсем удачно? Система станет хрупкой и будет сопротивляться изменениям, так как производные требования будут



привязывать нас к конкретной реализации. Требования к подсистемам будут оказывать воздействие на гибкость проекта, и изменение в одной подсистеме вызовет цепную реакцию в других подсистемах. Такие системы по своим свойствам аналогичны печной трубе, которая мешает перестроить дом. Они *противодействуют* изменениям. Проблемы интерфейсов могут быть еще серьезнее. Если интерфейсы не специфицированы надлежащим образом, система будет уязвимой.

Она не будет способна к эволюции в условиях меняющихся потребностей без полной замены интерфейсов и целых подсистем, основанных на них.

Когда подсистемы являются субконтрактами

Часто возникают дальнейшие осложнения. Поскольку подсистемы обычно разрабатываются различными командами (ведь, в частности, для этого мы осуществляли разбиение системы на подсистемы), требования и интерфейсы подсистем становятся *контрактами* между командами. ("Моя подсистема предоставляет результаты расчета скорости ветра в частности в указанном формате..."). Иногда подсистема может разрабатываться субподрядчиком из другой компании. В этом случае задача требований перестает быть чисто системной и технической и превращается в политический "футбол". ("Требования нельзя изменить, если не пересмотреть контракт.") В результате весь проект может застопориться. Следует заметить, что *многие попытки создания крупномасштабных систем были "похоронены"* из-за столкновения с данной проблемой.

Как сделать систему работоспособной

Что же делать? Первоочередной задачей является тщательное проведение проектирования системы в целом с использованием методов системной инженерии. Для систем с интенсивным использованием программного обеспечения, мы можем дать следующие рекомендации.

- Следует разработать, понять и сопровождать требования высокого уровня и прецеденты, которые *соединяют* подсистемы и описывают функции системы в целом. Эти прецеденты покажут, как предположительно будет работать система, и позволят сосредоточить внимание на главном. Они также помогут убедиться, что разработанная системная архитектура поддерживает наиболее вероятные сценарии использования.
- Необходимо сделать все возможное для наилучшего разбиения и изоляции функциональных возможностей внутри подсистем. При этом следует использовать принципы объектной технологии: инкапсуляцию и сокрытие информации, интерфейс по контракту, обмен сообщениями вместо совместного использования данных.
- Если это возможно, следует разрабатывать программное обеспечение как целое, а не в виде нескольких отдельных частей, каждая из которых связана с определенной физической подсистемой. Одной из характерных особенностей систем типа печной трубы является то, что на обеих сторонах интерфейса (неважно, хорошо

или плохо определенного) программа должна реконструировать состояние ключевых элементов (объектов), которые нужны для принятия решений; в отличие от аппаратного обеспечения, размещение требований на обеих сторонах не является признаком хорошего разбиения.

- При кодировании интерфейсов необходимо использовать одинаковый код на обеих его сторонах. В противном случае могут быть незначительные вариации, часто называемые "оптимизациями", которые сделают синхронизацию состояний очень затруднительной. И если граница между двумя физическими подсистемами позднее исчезнет (например, выяснится, что процессоры достаточно хороши, чтобы поддерживать обе подсистемы, и A, и B), то разработчикам программного обеспечения будет достаточно сложно "соединить" две части программного обеспечения.
- Следует задать такие спецификации интерфейсов, возможности которых больше, чем это необходимо для выполнения уже известных условий. Пусть будет немного большие полоса частот, дополнительный порт ввода-вывода или некие другие интеллектуальные средства, чтобы обеспечить пространство для расширения.

Наконец, нужно постараться найти кого-нибудь из "стариков", чтобы они помогли вам применить методы системной инженерии. Они уже делали это раньше, и их опыт будет полезен.

Поучительная история. Разбиение крупных систем программного обеспечения на подсистемы в случае, когда команда разработчиков состоит из нескольких отдельных групп

Однажды на занятиях Расти, опытный менеджер программного обеспечения, подошел к нам и поставил следующую задачу. Между нами произошел приведенный ниже диалог.

Расти: Мы создаем крупное приложение, которое выполняется одиночной системой-хостом. У нас есть две отдельные команды разработчиков по 30 человек каждая; одна команда находится на восточном берегу реки в Нью-Йорке, а вторая — на западном. Команды имеют различных менеджеров и разные уровни компетентности. Как можно распределить работу, чтобы создать систему, которая будет работоспособной сразу после окончания разработки?

Мы: Данную задачу можно рассматривать как задачу системной инженерии. Иными словами, представь, как можно разумным образом разбить систему на две подсистемы. Назови их *East* и *West* и размести требования к подсистемам так, как если бы они были отдельными физическими системами. Определи некий интерфейс, дополненный определением общих используемых классов и сервисов, что позволит двум подсистемам (приложениям) кооперироваться для осуществления общесистемных функций.

Расти: А не получится ли в результате произвольная система, не подчиняющаяся никаким архитектурным правилам?

Мы: Вполне возможно, в техническом смысле. Но соображения по разбиению, учитывающие логистические линии и специфику навыков команды проекта, могут оказаться не менее важными.

Расти: Не получится ли так, что будут созданы искусственные интерфейсы и система, которая может превратиться в “печную трубу”?

Мы: В некотором смысле, это возможно, но мы бы посоветовали, чтобы код интерфейса для обеих сторон разрабатывался только одной командой. В противном случае обе команды проделают много избыточной работы. При этом вы действительно создадите новые требования для системы, в том числе интерфейсы, которые не были бы не нужны или, по крайней мере, были бы не настолько формализованы. И действительно, важно помнить о проблеме печной трубы и сделать все возможное для того, чтобы минимизировать связь между системами и политические аспекты, которые могут при этом возникнуть.

Рабочий пример

Получилось слишком много для краткого введения в системную инженерию. Теперь попробуем применить то, что мы изучили, к HOLIS, нашей домашней системе автоматического освещения. Сейчас мы не будем тратить много времени на понимание требований к HOLIS. Мы сделаем это в последующих главах. В этом смысле системное проектирование будет несколько преждевременным. С другой стороны, мы, возможно, знаем достаточно, чтобы принять некоторые первоочередные решения по проекту системы, основываясь на нашем опыте и понимании вероятных требований. В любом случае на данном этапе мы не принимаем никаких окончательных решений относительно программного или аппаратного обеспечения и можем пересмотреть их позднее. В итеративном процессе, описанном далее, мы будем постоянно возвращаться к системной архитектуре и требованиям, поэтому начать можно и сейчас.

Предварительные потребности пользователя

Предположим, что для HOLIS уже были определены некоторые хорошо понимаемые потребности пользователя.

- HOLIS должна поддерживать “программируемые” клавиши-переключатели – индивидуально программируемые переключатели, используемые для активизации функций освещения в различных комнатах.
- Домовладельцы потребовали обеспечить возможность программировать HOLIS из удаленного центра, чтобы они имели возможность просто сообщить туда о своих потребностях и не заниматься вопросами “программирования” HOLIS вообще.
- Другие потенциальные покупатели потребовали, чтобы HOLIS можно было программировать с их домашнего персонального компьютера и чтобы им была предоставлена возможность самостоятельно выполнять все действия, связанные с установкой, программированием и обслуживанием системы.
- Еще одна категория пользователей пожелала, чтобы система предоставляла простой интерфейс в виде панели с кнопками, который можно было бы использовать для изменения программы HOLIS, установки режима “жильцы в отпуске” и т.д., не прибегая к применению персонального компьютера.
- Необходимо, чтобы HOLIS предлагала некую систему контактов в случае опасности.

Анализ проблемы

На первом этапе анализа проблемы (см. главу 4) команда разработала три постановки проблемы, первая из которых отражает точку зрения компании Lumenations.

Постановка проблемы с точки зрения компании Lumenations

Проблема	замедления роста в основной сфере деятельности компании на рынке профессионального театра
воздействует на	компанию, ее служащих и акционеров,
результатом чего	является неприемлемая эффективность и недостаток возможностей для роста доходов и прибыльности.
Выигрыш от	новых продуктов и потенциально новых рынков для продуктов и услуг компании может состоять в следующем. <ul style="list-style-type: none"> ■ Оживление компании и ее служащих. ■ Возрастание лояльности и сохранение дистрибуторов компании. ■ Более быстрый рост доходов и прибыльности. ■ Подъем цены акций компании.

Команда также решила посмотреть, способна ли она понять проблему с точки зрения будущих клиентов (конечных потребителей) и потенциальных дистрибуторов/строителей (клиентов компании Lumenations).

Постановка проблемы с позиции домовладельца

Проблема	недостаточного выбора продуктов, ограниченности функциональных возможностей и высокой цены существующих автоматических систем домашнего освещения
воздействует на	владельцев систем для жилья высшего класса,
результатом чего	является неприемлемое функционирование приобретенных систем или, что случается чаще всего, решение "вообще не автоматизировать".

Выигрыш от	"правильного" решения по автоматизации может состоять в следующем. <ul style="list-style-type: none"> ■ Удовлетворение домовладельца и гордость от обладания системой. ■ Возрастание гибкости и удобства использования жилья. ■ Совершенствование безопасности, комфорта и удобства.
-------------------	---

Постановка проблемы с позиции дистрибутора

Проблема	недостаточного выбора, ограниченности функциональных возможностей и высокой цены существующих автоматических систем домашнего освещения
воздействует на	дистрибуторов систем и строителей жилья высшего класса,
результатом чего	является недостаток возможностей для рыночной дифференциации и отсутствие новых высокоприбыльных продуктов.

Выигрыш от

“правильного” решения по автоматизации может состоять в следующем.

- Дифференциация.
- Более высокие доходы и прибыль.
- Возросшая доля рынка.

HOLIS: система, акторы и заинтересованные лица

Вернемся к анализу системы. Первое впечатление о HOLIS состоит в том, что это некая система, находящаяся внутри дома владельца (рис. 6.5).

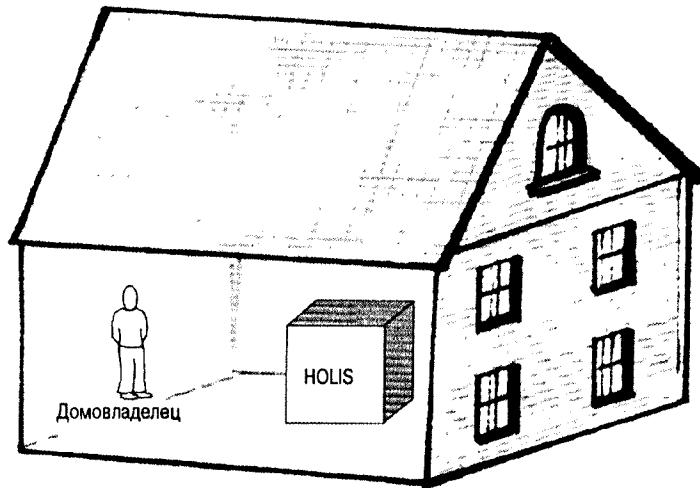


Рис. 6.5. HOLIS и ее окружение

Этап 3 анализа проблемы заключается в *выявлении заинтересованных лиц и пользователей системы*. Этап 4 состоит в *определении границ системы, являющейся решением*. С помощью предварительных данных о потребностях пользователей, которые мы уже получили, можно усовершенствовать понимание среды системы HOLIS, выявив акторы, которые будут с ней взаимодействовать. На рис. 6.6 представлены четыре актора.

1. Домовладелец, использующий систему HOLIS для управления освещением.
2. Лампы и т.п., различные осветительные элементы, которыми управляет HOLIS.
3. Службы компании Lumenations, представитель производителя, который имеет возможность настраивать HOLIS и осуществлять ее программирование с помощью удаленного доступа.
4. Получатель сообщений об опасных ситуациях, неопределенный актор, который будет получать сообщения об опасных ситуациях.

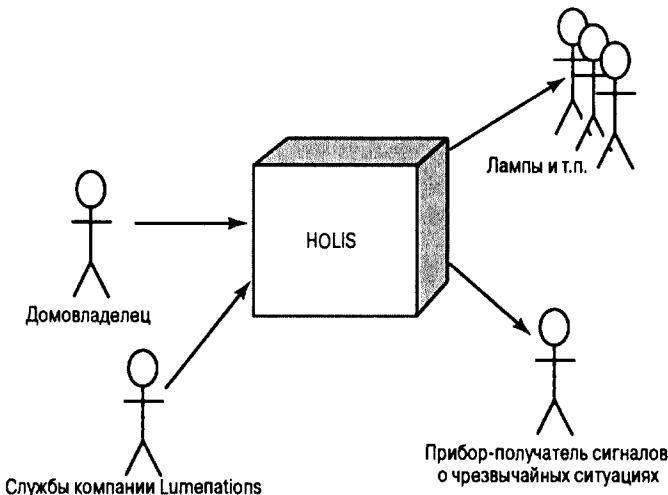


Рис. 6.6. Система HOLIS и акторы

Конечно же, команда выявит, что множество заинтересованных лиц, не являющихся акторами, как внутри, так и вне компании, беспокоятся о требованиях к HOLIS, что отражено в табл. 6.1.

Таблица 6.1. Заинтересованные лица, не являющиеся акторами системы HOLIS

Название	Комментарии
<i>Внешние</i>	
Дистрибуторы	Прямые клиенты компании Lumenations
Строители	Клиенты клиентов компании Lumenations: основной подрядчик, отвечающий перед домовладельцем за конечный результат
Подрядчики электрики	Отвечают за установку и сопровождение
<i>Внутренне</i>	
Команда разработчиков	Команда компании Lumenations
Управление производством/сбытом	Будет представлено Кэти, менеджером продукта
Руководство компании Lumenations	Занимается ведением отчетов о финансировании и выпуске продукции

Применение принципов системной инженерии к HOLIS

После того как мы выявили внешние акторы системы HOLIS, можно приступить к размышлению системного уровня, чтобы решить, как можно осуществить разбиение HOLIS на подсистемы. Этот процесс хорошо производить с помощью следующих рассуждений из области системной инженерии.

- Было бы хорошо, чтобы управляющее устройство и персональный компьютер домовладельца имели общее программное обеспечение; следовательно, мы будем подбирать основавшую на ПК реализацию для обоих элементов системы.

- Мы еще не знаем точно, какая гибкость будет нужна для дистанционных программируемых переключателей, но уже понятно, что их должно быть много; некоторые из них будут находиться достаточно далеко от центрального блока управления, и, вероятно, понадобится некое интеллектуальное общение между ними и блоком управления.

В результате систему HOLIS можно разбить на три подсистемы: *Управление включением*, представляющая собой дистанционный программируемый пульт управления; *Центральный блок управления*, т. е. центральная компьютерная система управления; и *ПК-программатор*, дополнительно поставляемая система на базе ПК, которую заказали некоторые домовладельцы. Теперь структурная схема выглядит так, как на рис. 6.7.

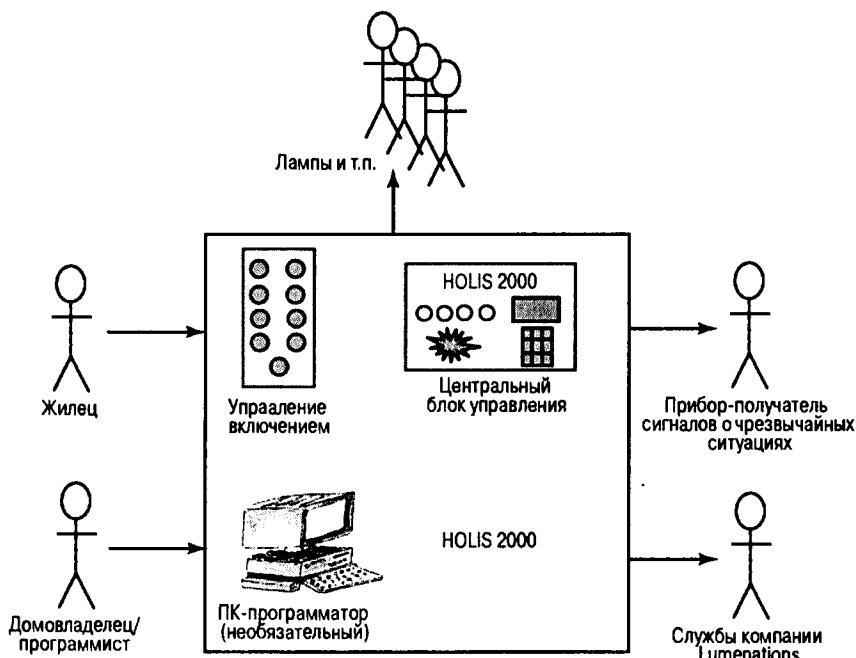


Рис. 6.7. Система HOLIS, ее подсистемы и акторы

Заметим, что мы, похоже, выявили пятый актор, который также является домовладельцем, но использует ПК для программирования HOLIS, а не занимается включением/выключением света. Домовладелец/программист в этой роли имеет другие потребности в системе, и поэтому является для нее отдельным актором. Позднее мы будем рассматривать различные варианты поведения, ожидаемые этим актором от системы HOLIS.

Подсистемы системы HOLIS

С точки зрения требований задача стала немного сложнее. Кроме необходимости понять требования к системе HOLIS в целом, теперь нам также нужно будет понять отдельные требования к каждой из ее трех подсистем. Можно снова воспользоваться парадиг-

мой акторов на следующем уровне декомпозиции системы. При этом мы получим три новые структурные диаграммы (рис. 6.8–6.10).

На рис. 6.8, рассматривая систему с точки зрения “Управления включением”, мы находим еще один актор, “Центральный блок управления” (ЦБУ), другую подсистему. Другими словами, на подсистемном уровне, ЦБУ является актором для “Управления включением”, и нам нужно будет понять, какие требования и прецеденты ЦБУ будет создавать для “Управления включением”. Это множество требований является производным от нашей декомпозиции системы.

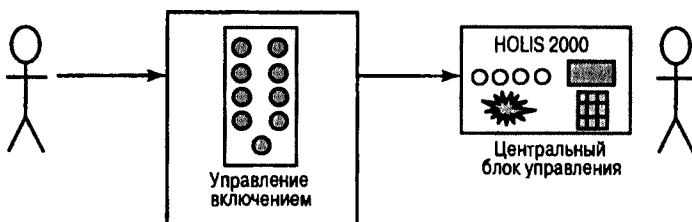


Рис. 6.8. Подсистема “Управление включением” и ее акторы

На рис. 6.9 представлен вид системы с точки зрения ПК домовладельца. Здесь нет ничего нового, по крайней мере в том, что касается подсистем и акторов, так как они уже были определены ранее. Рис. 6.10, однако, представляет более интересную картину, и мы видим, что ЦБУ имеет акторов больше, чем какая-либо другая подсистема. Это интуитивно ясно, ведь ЦБУ – это “мозг” системы HOLIS, поэтому данная подсистема имеет больше обязанностей и наибольшее количество взаимодействующих с ней акторов.



Рис. 6.9. Подсистема “ПК-программатор” и ее акторы

Для завершения анализа проблемы рассмотрим табл. 6.2, где перечислены выявленные командой ограничения, которые обсуждены и согласованы разработчиками системы HOLIS и руководством компании Lumenations.

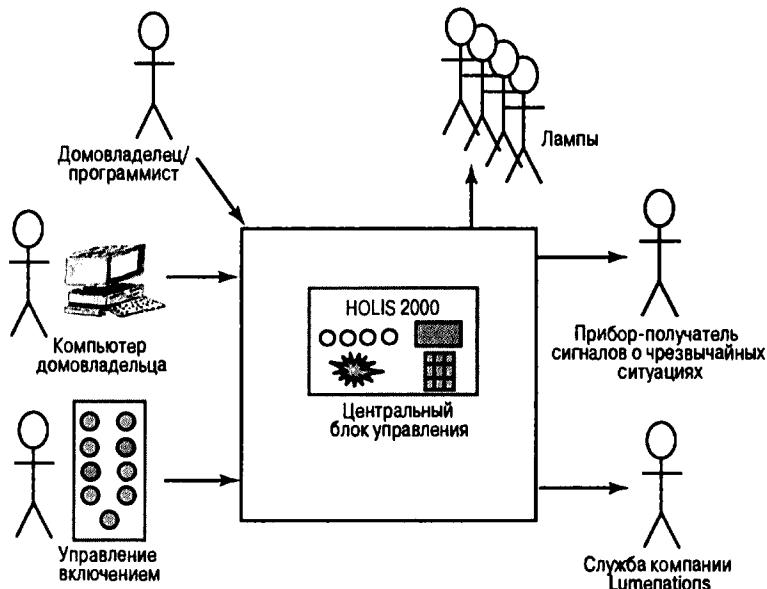


Рис. 6.10. Подсистема “Центральный блок управления” и ее акторы

Таблица 6.2. Ограничения проектирования системы HOLIS

№#	Описание	Пояснения
1	Версия 1.0 должна быть запущена в производство до 5 января 2000 года	Единственная возможность запустить продукт в этом году
2	Команда должна использовать UML-моделирование, ОО-методологии и унифицированный процесс разработки ПО (Unified Software Development Process)	Мы надеемся, что эти технологии обеспечат более высокую производительность и робастность
3	Программное обеспечение для ЦБУ и “ПК-программатора” будет написано на языке C++. Для “Управления включением” будет использован ассемблер	Для целостности и удобства сопровождения; помимо прочего, команда знает эти языки
4	Система-прототип должна быть показана на декабрьской торговой выставке средств домашней автоматизации	Чтобы получить заказы дистрибуторов на 1-й квартал 2000-го финансового года
5	Микропроцессорная подсистема для ЦБУ будет скопирована с усовершенствованного проекта системы освещения (УПСО), разработанного подразделением профессионального освещения	Готовый проект и материальная часть
6	Единственная поддерживаемая конфигурация ПК-программатора домовладельца должна быть совместима с Windows 98	Сокращение масштаба версии 1.0
7	Команде разрешается взять двух новых сотрудников на условиях полной занятости после успешного испытательного срока; при условии, что они обладают необходимыми навыками	Максимально разрешенное увеличение бюджета

Окончание табл. 6.2

№#	Описание	Пояснения
8	В "Управлении включением" будет использоваться монокристаллический микропроцессор KCH5444	Уже используется в компании
9	Разрешено использовать закупаемые компоненты программного обеспечения, если это не накладывает на компанию долгосрочных обязательств по выплате роялти	Не должно быть долгосрочного влияния на программное обеспечение затрат на приобретенные товары

На данном этапе этого достаточно для анализа проблемы и системного проектирования HOLIS. Мы вернемся к нашему рабочему примеру в следующих главах.

Заключение части 1

В части 1, “Анализ проблемы”, описывается ряд профессиональных приемов, применяемых командой, чтобы понять проблему, которую предстоит решить, до того, как приступить к разработке приложения. Предлагается метод анализа проблемы, состоящий из пяти этапов.

1. Достигнуть соглашения по определению проблемы.
2. Выделить основные причины – проблемы, стоящие за проблемами.
3. Выявить заинтересованных лиц и пользователей, чье мнение будет в конечном счете определять успех или неудачу системы.
4. Определить, где следует искать границы решения.
5. Понять ограничения, которые будут наложены на команду и решение.

Систематически проведенный анализ проблемы повысит способность команды решить следующую задачу – *предложить решение существующей проблемы*.

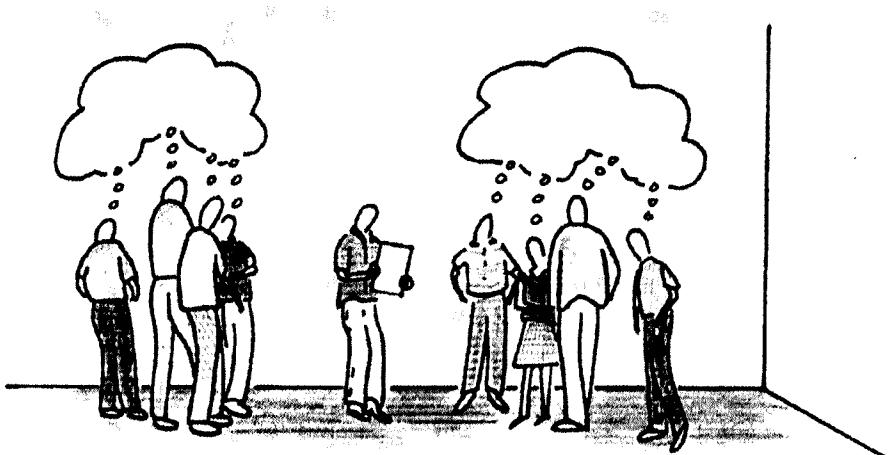
Мы также описали различные методы, которые можно использовать при анализе проблемы. В частности, мы рассмотрели моделирование бизнес-процесса, которое очень хорошо зарекомендовало себя в сложных информационных системах, поддерживающих основные бизнес-инфраструктуры. Команда может использовать моделирование бизнес-процесса как для понимания пути развития бизнеса, так и для определения, где внутри системы можно наиболее эффективно развернуть приложения. Мы также выяснили, что определенная нами бизнес-модель будет иметь параллельные конструкции в программном приложении, и будем использовать эту общность при проектировании программного обеспечения. Выявленные на данном этапе бизнес-прецеденты будут позднее применяться для определения требований к самому приложению.

Для класса программных приложений, которые мы отнесли к встроенным системам, при анализе проблемы мы использовали методы системной инженерии, позволяющие осуществить разбиение сложной системы на подсистемы. Этот процесс помогает понять, где должны находиться программные приложения и каким общим целям они служат. При этом оказалось, что мы в чем-то усложняем проблему требований, создавая новые подсистемы, для которых в свою очередь нужно заниматься пониманием требований.

Часть 2

Понимание потребностей пользователей

- Глава 7. Задача выявления требований
- Глава 8. Функции продукта или системы
- Глава 9. Интервьюирование
- Глава 10. Совещания, посвященные требованиям
- Глава 11. Мозговой штурм и отбор идей
- Глава 12. Раскадровка
- Глава 13. Применение прецедентов
- Глава 14. Обыгрывание ролей
- Глава 15. Создание прототипов



Среди причин возникновения проблем в проектах чаще всего упоминается “недостаток информации от пользователя”. (Исследование группы Стендиша, 1994)

В отчете группы Стендиша одной из наиболее часто упоминаемых причин возникновения проблем в проектах назван “недостаток информации от пользователя”. 13% респондентов указали эту причину в качестве основной, другие 12% назвали “неполные требования и спецификации”. Из этих данных следует, что примерно для четверти всех испытывающих затруднения проектов недостаточное понимание реальных требований пользователей (а точнее, всех заинтересованных лиц) является серьезной проблемой, влияющей на успех проекта.

Очевидно, что инициативу должны взять на себя команды разработчиков, так как вряд ли все пользователи, проснувшись однажды утром, вдруг начнут делать все возможное для прояснения своих требований. Иными словами, нашим командам необходимо выработать профессиональные приемы выявления требований.



В части 1 мы разработали приемы, которые помогут понять решаемую проблему. В части 2 описываются методы, которые команда разработчиков может использовать для достижения понимания реальных потребностей будущих пользователей и других заинтересованных лиц. При этом мы также начнем понимать возможные требования к системе, которая будет разрабатываться для удовлетворения этих потребностей. На данном этапе основное внимание уделяется потребностям заинтересованных лиц, которые находятся в верхней части пирамиды требований.

В этой части будут рассматриваться различные методы — от простых и недорогих, таких как интервьюирование, до весьма дорогостоящих и достаточно формальных, таких как создание прототипов. Ни один из этих методов не является универсальным, однако, ознакомившись с ними, команда сможет выбирать. Для каждого конкретного проекта команда сможет определить подходящие методы и применить опыт и знания, полученные при работе над выявлением требований в предыдущих проектах. Таким образом, команда создаст уникальный набор подходящих для данной среды приемов, которые могут внести заметный вклад в совершенствование конечного результата.

Глава 7

Задача выявления требований

Основные положения

- Выявление требований осложняется тремя эндемическими синдромами.
- Синдром “да, но...” является следствием человеческой природы и неспособности пользователя воспринимать программное обеспечение как физический прибор.
- Поиск требований сродни археологическим раскопкам (поиску “ненайденных руин”); чем больше вы находитите, тем больше вы знаете о том, что еще осталось ненайденным.
- Синдром “пользователь и разработчик” отражает глубокое различие между ними, затрудняющее общение.

В последующих нескольких главах мы будем рассматривать множество методов выявления требований пользователей системы и других заинтересованных лиц¹. Этот процесс кажется достаточно очевидным: нужно собраться вместе с будущими пользователями системы и другими заинтересованными лицами и спросить их, что должна, по их мнению, делать система.

Что в этом сложного? Почему нужно так много методов? И зачем нужны эти профессиональные приемы вообще? Чтобы лучше понять эту конкретную проблему, сначала рассмотрим три синдрома, которые могут значительно усложнить предмет.

Преграды на пути выявления требований

Синдром “да, но • •”

Одной из самых неприятных проблем при разработке приложений является то, что мы назвали синдромом “да, но...”. Это наблюдаемая нами реакция пользователя на каждый когда-либо разработанный фрагмент программного обеспечения. Именно благодаря ему мы наблюдаем две немедленные противоположные реакции, когда пользователь впервые видит реализацию системы.

¹ Иногда для их обозначения мы будем использовать понятие *пользователь* в обобщенном смысле. Методы применяются для выявления требований всех заинтересованных лиц.

- “О, это действительно здорово, мы можем реально использовать это, классная работа, молодцы, мальчики,” и т. д.
- “Да, но, как насчет...? Нельзя ли было бы...? А что будет, если...?”

Причина синдрома “да, но...” кроется глубоко в природе программного обеспечения как интеллектуального неосозаемого процесса. Проблема усугубляется тем, что команды разработчиков крайне редко предоставляют что-либо пользователям для обсуждения и взаимодействия до окончания разработки программного кода.

Реакция пользователей является следствием человеческой природы. Подобную реакцию можно часто наблюдать и при других повседневных обстоятельствах. Пользователи



никогда ранее не видели новую систему или что-либо подобное; они не понимают, что вы подразумеваете, когда описываете ее. И вот теперь она перед ними – впервые после стольких месяцев (или лет) ожидания они имеют возможность взаимодействовать с системой. И оказывается, что это не совсем то, чего они ожидали!

Давайте сравним создание программного обеспечения с созданием механических приборов, технология и процесс разработки которых старше программирования на несколько сотен лет. Для механических систем существует хорошо разработанная методология принципиальных моделей, макетов, пошаговых прототипов, пробных промышленных изделий и т.д. Все они являются осозаемыми, и большинство из них выглядит и действует аналогично разрабатываемому прибору.

Пользователи могут увидеть макет прибора, потрогать его, всесторонне обсудить и даже попытаться поработать с ним *гораздо раньше*, чем будет завершена детальная реализация. Исключительно для получения ранней непосредственной реакции на концептуальное определение продукта были разработаны специальные технологии, такие как стереолитография, в которых быстрый прототип создается сегодня на сегодня. И только в программном обеспечении, несмотря на всю его сложность, мы почему-то надеемся получить удовлетворительный результат с первой попытки.

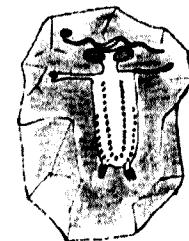
Как это ни грустно, но нужно принять факт существования синдрома “да, но...” в качестве объективной реальности и сделать некоторые выводы, которые помогут членам команды смягчить влияние этого синдрома в будущих проектах.

- Синдром “да, но...” является следствием человеческой природы и неотъемлемой частью разработки любого приложения.
- Мы можем существенно уменьшить воздействие этого синдрома путем применения методов, которые выявят эти “но” как можно раньше. Выявив их на более ранних этапах, мы можем направить большую часть наших усилий на разработку программ, которые уже прошли тест на “да, но...”.

Синдром “неоткрытых руин”

Один из наших друзей однажды был экскурсоводом на автобусном маршруте в районе Four Corners, области, определенной общими границами штатов Колорадо, Нью-Мексико, Юта и Аризона. Экскурсия проходила по живописным вершинам горной цепи Ла-Плата, античным руинам Анасази в Меса-Верде и окрестностям. Вопросы туристов являются постоянным источником развлечения для экскурсоводов и создают определенный фольклор туристического бизнеса. У нашего друга самым любимым "дуряцким" вопросом был "А сколько здесь неоткрытых руин?".

Во многом поиск требований напоминает поиск неоткрытых руин: чем больше их найдено, тем лучше известно, что это еще не все. Вы никогда не сможете почувствовать, что нашли все, и, пожалуй, так оно и есть на самом деле. Повсеместно команды разработчиков программного обеспечения пытаются определить, когда же можно закончить с выявлением требований, т.е. когда можно считать, что они выявили все существенные требования или хотя бы достаточное их количество.

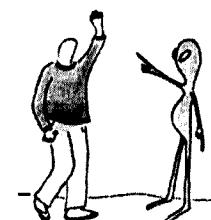


Чтобы помочь команде справиться с этой проблемой, мы предлагаем множество методов, как в рамках части 2, так и в последующих главах. Конечно, как отмечалось в части 1, огромное значение имеет выявление всех заинтересованных лиц на этапе анализа проблемы. Многие из этих заинтересованных лиц (не являющихся пользователями) зачастую являются источниками требований, которые в противном случае так и не будут выявлены. Как и в случае поиска неоткрытых руин, мы должны осознавать, что эту миссию никогда нельзя считать завершенной. Но мы также понимаем, что на определенном этапе можно будет с уверенностью сказать: "Мы узнали достаточно".

Синдром “пользователя и разработчика”

Методы выявления требований не новы. Разработчики приложений более 40 лет работали над их совершенствованием. Почему же тогда понимание потребностей пользователя остается одной из основных проблем? Поскольку лишь немногие разработчики приложений имеют опыт применения методов выявления требований, это не так уж удивительно.

Синдром "пользователь и разработчик" является следствием расхождения взглядов пользователей и разработчиков. Пользователи и разработчики, как правило, принадлежат к различным мирам, говорят на разных языках и имеют различный опыт, мотивацию и цели.



Иногда мы должны учиться более эффективно общаться с этими "пользователями из другого мира". В своей статье Лора Шерер (*Laura Scharer, 1981*) описывает эту проблему и предлагает некие рекомендации по ее смягчению. В табл. 7.1 представлены некоторые аспекты данной проблемы и предлагаемые решения.

Таблица 7.1. Синдром “пользователь и разработчик”

Проблема	Решение
Пользователи не знают, чего хотят, а если и знают, то не могут это выразить	Признавать пользователя экспертом в предметной области и ценить его в этом качестве; пытаться использовать альтернативные методы общения и выявления требований

Окончание табл. 7.1

Проблема	Решение
Пользователи думают, что они знают, чего хотят, до тех пор, пока разработчики не предстают им то, что они якобы хотели	Как можно раньше предлагать альтернативные методы выявления: раскадровку, ролевые игры, прототипы и т.п.
Аналитики думают, что они понимают проблемы пользователя лучше его самого	Поставить аналитика на место пользователя. Провести ролевую игру в течение часа или всего дня
Все считают, что другие руководствуются политическими мотивами	Такова человеческая натура, поэтому пусть все остается как есть

Мы надеемся, что лучше понимая природу этих проблем и некоторые подходы по их смягчению, разработчики будут лучше подготовлены к дальнейшей работе.

Методы выявления требований

Для достижения лучшего понимания потребностей пользователя нам нужно переместиться из области битов и байтов, где многие разработчики чувствуют себя более комфортно, в область, где нужно общаться с реальными людьми и понимать проблемы реального мира. Существует множество методов, которые можно использовать для анализа и проектирования программных решений. Аналогично существует множество методов для понимания требований пользователей и заинтересованных лиц.

В главах 4–6 мы начали свой путь с анализа проблемы, рассмотрели ряд вопросов, которые можно задать относительно налагаемых на систему ограничений, описали моделирование бизнес-процесса, которое можно использовать для многих приложений, а также ознакомились с методами системной инженерии, которые можно применить к сложным системам со встроенным программным обеспечением. В следующих главах будут описаны методы, доказавшие свою эффективность в преодолении трех перечисленных выше синдромов.

- Интервьюирование и анкетирование
- Совещания, посвященные требованиям
- Мозговой штурм и отбор идей
- Раскадровки
- Прецеденты
- Обыгрывание ролей
- Создание прототипов

Выбор конкретного метода будет зависеть от типа приложения, опыта и уровня подготовки команды разработчиков, заказчика, масштаба проблемы, критичности приложения, используемой технологии и уникальности приложения.

Глава 8

Функции продукта или системы

Основные положения

- Команда разработчиков должна играть более активную роль в выявлении требований к системе.
- Функции системы или продукта являются высокогородневым выражением желаемого поведения системы.
- Количество функций системы должно находиться в пределах 25–99; предпочтительно, чтобы оно не превышало 50.
- Атрибуты обеспечивают дополнительную информацию о функции.

В предыдущих главах мы описали ряд проблем, приводящих к тому, что команда разработчиков практически никогда не получает совершенной или, по крайней мере, достаточно хорошей спецификации, которую можно использовать в качестве основы для разработки системы. Один из выводов, который из этого следует, состоит в том, что если нам не дают хороших определений, мы должны сами пойти и добить их. Другими словами, чтобы добиться успеха, команда разработчиков должна играть гораздо более активную роль в процессе *выявления требований*. Мы увидим, что хотя и можно возложить основную часть этой ответственности на руководителя, аналитика или менеджера продукта, в конечном итоге вся команда будет вовлечена в тот или иной этап данного процесса.

Потребности заинтересованных лиц и пользователей

Очевидно, что команда разработчиков может создать хорошую систему только в том случае, если она понимает реальные потребности заинтересованных лиц. Эта информация необходима команде для принятия правильных решений при определении и реализации системы. Совокупность исходных данных, которые мы называем *потребностями заинтересованных лиц* или *пользователей*, представляет собой критически важный фрагмент собираемой картины.

Зачастую эти потребности пользователя будут неоднозначными и размытыми. Заказчик может сказать: "Мне нужны простые способы, позволяющие узнать состояние моих складских запасов" или "Мне бы хотелось, чтобы существенно возросла производительность ввода заказов на покупку". Но, несмотря на нечеткость формулировки, именно эти высказывания создают основу для всех последующих действий. Раз они



так важны, мы направим свои усилия на то, чтобы как следует понять их. Мы определим потребность заинтересованного лица следующим образом.

Отражение некой личной, рабочей или бизнес-проблемы (или возможности), решение которой отравливает замысел, покупку или использование новой системы.

Функции

Интересно, что, говоря о своих потребностях или требованиях к новой системе, заинтересованные лица, как правило, описывают их не так, как в приведенных выше высказываниях. Они часто называют вам не свою реальную потребность ("Если я не повышу производительность этого отдела, то не получу премию за этот год" или "Я хочу иметь возможность затормозить эту машину как можно быстрее без пробуксовки") и не реальное требование к системе ("Я должен снизить время обработки ввода заказов на покупку на 50 процентов" или "Автомобиль должен иметь систему компьютерного контроля для каждого колеса"). Вместо этого они описывают некую абстракцию, что-то вроде "Мне нужен новый экран на основе GUI для ввода заказов на покупку" или "Я хочу, чтобы машина была оснащена антиблокировочной тормозной системой".

Мы называем эти высокоуровневые выражения желаемого поведения системы *функциями* (*features*) продукта или системы. Эти функции часто не очень хорошо определены и могут даже противоречить друг другу. "Я хочу увеличить скорость обработки заказов" и "Я хочу обеспечить более дружественный пользователю интерфейс, чтобы помочь нашим новым служащим изучить систему", ибо так или иначе они являются отражением реальных потребностей.

Что происходит при обсуждении? Пользователь уже преобразовал реальную потребность (производительность или безопасность) в поведение системы, которое, по его мнению, будет служить реальной потребности (рис. 8.1). При этом *что* ("Мне нужно") незаметно заменилось на *как* ("что, по моему мнению, должна делать система, чтобы удовлетворить данную потребность"). Это неплохо, так как он имеет реальный опыт в данной предметной области и реальное понимание значения функции. Кроме того, такие функции легко обсуждать и документировать на обычном языке, а также объяснять их другим, что существенно обогащает схему требований.

Использование функций – удобный способ описания возможностей без лишних подробностей.

Но такой подход имеет недостаток. Если команда при обсуждении не поймет, какая потребность стоит за функцией, это может привести к неприятным последствиям. Если по какой-либо причине функция не служит реальной потребности, то система может потерпеть неудачу в удовлетворении целей пользователя, несмотря на то что в ней будет реализована запрашиваемая функция.

Тем не менее мы считаем этот высокий уровень абстракции, эти *функции*, весьма полезным и удобным способом описания функциональных возможностей новой системы без излишних подробностей. Данные функции будут служить основой нашей деятельности по разработке требований.

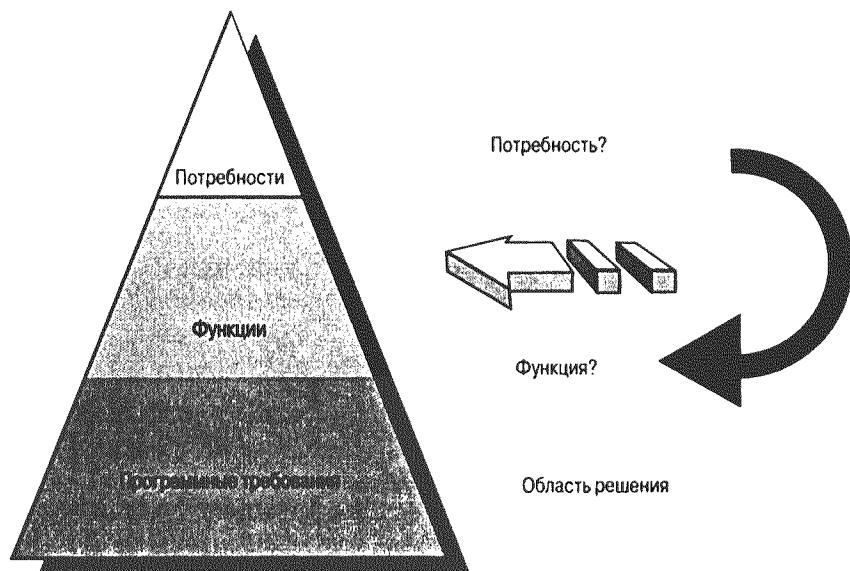


Рис. 8.1. Требования и функции тесно взаимосвязаны

Ранее мы определили функцию следующим образом.

Обслуживание, предоставляемое системой для выполнения одной или нескольких потребностей заказчика.

В таком определении описываемые пользователями функции не могут быть столь уж далеки от их потребностей, так что у нас есть удобный способ, чтобы начать определять систему.

Основным при понимании нужд пользователя является выявление и организация потребностей и функций предлагаемой системы. Иногда мы будем получать все потребности и ни одной функции, а в других случаях нам будут указаны все функции и ни одной потребности. Порой мы не сможем отделить их друг от друга. Но, помня о различии между ними, мы в любом случае должны выделять информацию о том, что система должна делать.

Функции легко описать на обычном языке. Их описания состоят из коротких фраз (табл. 8.1), лишь изредка функции разрабатываются более подробно. Они представляют собой очень полезные конструкции для управления масштабом продукта на ранних этапах взаимных согласований и поиска компромиссов. Формулировка функций не требует значительных инвестиций; их легко описать и перечислить.

Таблица 8.1. Примеры функций

Прикладная область	Пример функции
Система управления элеватором	Осуществляемое вручную управление дверью при угрозе пожара
Система управления запасами	Предоставлять свежую информацию о состоянии всех инвентарных единиц
Система обнаружения неисправностей	Обеспечивать текущие данные для оценки качества продукции

Окончание табл. 8.1

Прикладная область	Пример функции
Система обработки платежных ведомостей	Сообщать текущие начисления по категориям
Автоматическая система домашнего освещения (HOLIS)	Установка специального режима на период длительного отсутствия
Система контроля вооружений	Требуется, как минимум, две независимые конфигурации авторизации для запуска
Готовое приложение	Совместимость с Windows 2000

Управление сложностью путем выбора уровня абстракции

Систему произвольной сложности можно определить с помощью списка из 25–99 функций.

Количество функций, которое мы позволяем себе рассматривать, будет существенно влиять на уровень абстракции определения. Для того чтобы справиться со сложностью разрабатываемой системы, мы рекомендуем описывать возможности каждой новой системы или дополнения к уже существующей системе как можно более абстрактно, чтобы в результате получить не более 25–99 функций, причем желательно, чтобы их число не превышало 50.

При этом относительно небольшой объем информации обеспечивает всестороннюю и полную основу для определения продукта, общения с заказчиками, управления масштабом и проектом. С помощью 25–99 функций, удобным образом разбитых на категории и упорядоченных, мы сможем описывать и обсуждать самые разнообразные системы, будь то космический корабль многоразового использования или программное средство (например, автоматическое обнаружение неисправностей). В части 5 эти функции будут преобразованы в детальные требования, достаточно конкретные, чтобы их можно было реализовать. Мы будем называть их *требованиями к программному обеспечению*, или *программными требованиями (software requirements)*, в отличие от высокоуровневых функций. Необходимость в дополнительной конкретизации возникнет позднее. На данном этапе нам достаточно рассуждать на уровне функций.

После того как возможные функции перечислены, можно приступить к принятию решений вида “отложить до следующей версии”, “реализовать немедленно”, “полностью отвергнуть” или “исследовать дополнительно”. Этот процесс *корректировки масштаба* лучше проводить на уровне функций, а не на уровне требований, иначе можно просто увязнуть в деталях. Мы рассмотрим проблему масштаба более подробно в части 4, “Управление масштабом”.

Атрибуты функций продукта

Чтобы было легче оперировать этой информацией, мы будем рассматривать *атрибуты функций* – элементы данных, которые обеспечивают дополнительную информацию о каждой функции. Атрибуты используются для того, чтобы связать функции или требования с другой информацией проекта. С помощью атрибутов можно отслеживать функции (имя или уникальный идентификатор, состояние, исторические данные, распределен из, трассируется к и т.д.), задавать их приоритет (поле приоритета), а также управ-

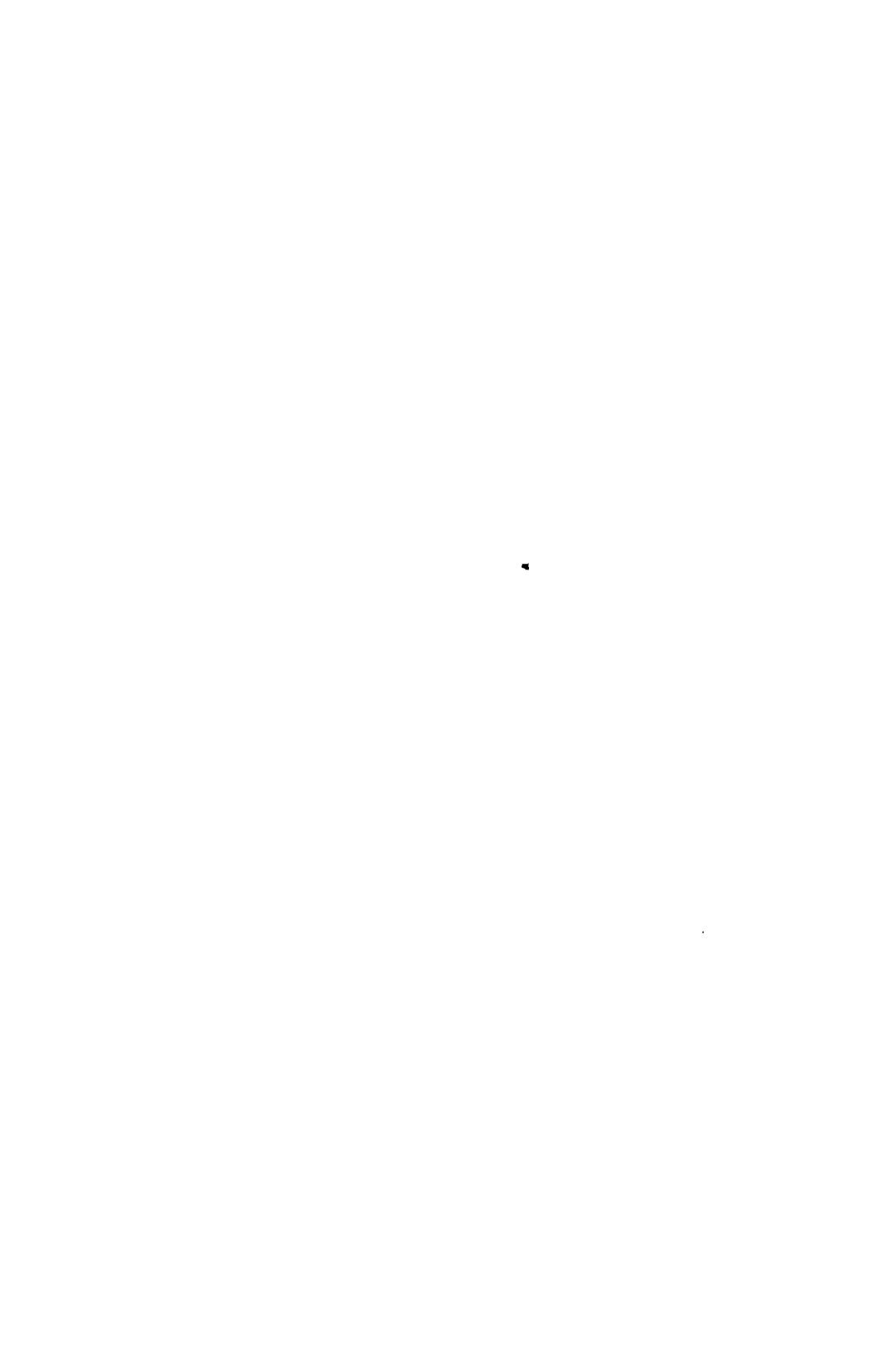
лять (статус) функциями, предложенными для реализации. Например, атрибут *приоритет* можно использовать для хранения результатов накопительного голосования во время “мозгового штурма”; атрибут *номер версии* – для фиксации конкретной версии программного обеспечения, в которой предполагается реализовать данную функцию.

Присваивая функциям различные атрибуты, можно более успешно управлять сложностью информации. Хотя атрибуты могут быть самыми разнообразными, опыт свидетельствует, что существуют некие общеупотребительные атрибуты, которые применяются в большинстве проектов (табл. 8.2). Далее в данной книге мы будем использовать эти атрибуты при работе с функциями и требованиями, а также отношениями между различными типами системных требований.

Таблица 8.2. Атрибуты функций

Атрибут	Описание
Статус	Отслеживает ход процесса определения базового уровня проекта и последующей разработки. <i>Пример:</i> функция может иметь статус предлагаемая, утвержденная, включенная
Приоритет/Полезность	Функции не одинаковы по своей важности. Определение относительных приоритетов или полезности для конечного пользователя открывает путь к диалогу между заинтересованными лицами и членами команды разработчиков. Этот атрибут используется при управлении масштабом и определении очередности. <i>Пример:</i> определение функции как критической, важной, полезной
Трудоемкость	Оценка количества командо- или человеко-недель, строк кода или общего уровня трудоемкости помогает определить, что можно, а что нельзя осуществить за определенный период времени. <i>Пример:</i> низкий, средний или высокий уровень трудоемкости
Риск	Вероятность того, что данная функция вызовет нежелательные последствия, такие как увеличение расходов, отставание от графика или даже закрытие проекта. <i>Пример:</i> высокий, средний и низкий уровень риска
Стабильность	Вероятность того, что будет меняться данная функция или ее описание командой. Используется для того, чтобы помочь при определении приоритетов разработки и выявлении тех элементов, для которых следующим действием должно стать дополнительное исследование
Целевая версия	Указание версии продукта, в которой впервые появится реализация данной функции. Комбинирование этого атрибута с полем статуса дает команде возможность предлагать, записывать и обсуждать различные функции, не принимая их к разработке
Назначение	Во многих проектах функции будут предназначаться “функциональным командам”, ответственным за дальнейшую их доработку, написание программных требований, а также, возможно, их реализацию
Обоснование	Используется для отслеживания источника запрашиваемой функции. Например, ссылка может указывать на страницу или номер строки спецификации продукта или временной маркер на видеозаписи важного интервью с клиентом.

Итак, давайте перейдем к некоторым профессиональным приемам, которые помогут нам получить необходимую информацию. Начнем с интервьюирования (глава 9).



Глава 9

Интервьюирование

Основные положения

- Интервьюирование — простой и понятный метод.
- Контекстно-свободные вопросы помогают получить свободные от предубеждений интервью.
- Еще не обнаруженные требования можно искать путем объяснения решений.
- Соглашение о некоторых общих потребностях положит начало "архиву требований", который будет использоваться при дальнейшей разработке проекта.
- Анкетирование не может заменить интервьюирование.

Одним из наиболее важных и понятных методов получения требований является *интервью с клиентом*; это метод, который можно использовать практически в любой ситуации. В данной главе описывается процесс интервьюирования и предлагается общая схема проведения интервью с пользователем или заказчиком. Однако данный процесс не так прост, как может показаться на первый взгляд, в нем мы оказываемся лицом к лицу с синдромом “пользователя и разработчика”.

Кроме того, одна из основных задач интервьюирования — сделать все возможное, чтобы предубеждения и предпочтения интервьюируемых не повлияли на свободный обмен информацией. Это сложная проблема. Социология (еще один предмет, который мы в свое время пропустили!) учит нас, что *невозможно воспринимать окружающий мир, не фильтруя его в соответствии со своим происхождением и накопленным опытом*.

Поскольку решать проблемы — наша профессия, мы редко оказываемся в ситуации, когда у нас нет идей по поводу того, какой тип решений будет соответствовать конкретной проблеме. Действительно, в большинстве случаев мы работаем в некой определенной предметной области или среде, где определенные элементы решения очевидны или, по крайней мере, кажутся таковыми. (“Мы решили подобную проблему ранее и полностью уверены, что наш опыт будет применим и в этом случае.”) Конечно, это неплохо, так как наши представления являются частью того, за что нам платят. Но мы не должны позволить, чтобы они оказывали влияние на понимание реальной проблемы, которую следует решить.

Контекст интервью

Контекстно-свободные вопросы

Контекстно-свободные вопросы помогают достичь понимания реальной проблемы, не оказывая влияния на ответы пользователя.

Как же избежать предубеждения пользователя при ответах на наши вопросы? Для этого необходимо задавать вопросы о природе проблемы пользователя, никак не связывая их с возможным решением. Для решения данной проблемы Гаус (Cause) и Вайнберг (Weinberg) (1989) ввели понятие *контекстно-свободный вопрос*. Рассмотрим примеры таких вопросов.

- Кто является пользователем?
- Кто является клиентом?
- Отличаются ли их потребности?
- Где еще можно найти решение данной проблемы?

Эти вопросы заставляют нас выслушать заказчика, прежде чем пытаться предложить или описать потенциальное решение. Это позволяет нам лучше понять проблему заказчика и все проблемы, стоящие за ней. Такие проблемы влияют на мотивацию или поведение заказчика, и их необходимо учесть, прежде чем мы сможем предложить успешное решение.

Контекстно-свободные вопросы аналогичны вопросам, которые учатся задавать продавцы, когда осваивают метод, получивший называние “продажа решений”. Используя этот метод, продавец задает ряд вопросов, чтобы получить реальное представление о проблеме клиента, а также о том, какие решения, если такие существуют, предлагает сам клиент. Это позволит ему предложить успешные решения и сравнить их достоинства. Данный процесс иллюстрирует вклад продавца в предложение исчерпывающего решения проблемы, стоящей перед клиентом.

Добавление контекста

После того как заданы контекстно-свободные вопросы, можно исследовать предложенные решения.

После ответов на контекстно-свободные вопросы, чтобы найти еще не обнаруженные требования, полезно “сместить” вопросы в область исследования решений. Как правило, от нас требуется не только понять проблему, но и предложить подходящие решения. Обсуждение предлагаемых решений поможет пользователю углубить или даже изменить взгляд на проблему. И конечно, наши пользователи зависят от того, владеем ли мы предметом; в противном случае они должны научить нас всему, что они знают о нем.

Чтобы помочь команде разработчиков овладеть данным приемом, мы объединили вопросы в “обобщенное практически контекстно-свободное интервью”. Это структурированное интервью можно использовать при выявлении требований пользователей или заинтересованных лиц практически для любых программных приложений. Образец интервью представлен на рис. 9.1. В нем содержатся как контекстно-свободные, так и не-

контекстно-свободные вопросы. Кроме того, в интервью предлагаются вопросы, предназначенные для исследования таких аспектов требований, как надежность, возможность сопровождения и т.д.

Часть I. Определение профиля заказчика или пользователя

Имя:

Компания:

Отрасль:

Должность:

(Выше приведенная информация, как правило, может быть внесена заранее.)

Каковы ваши основные обязанности?

Что вы в основном производите?

Для кого?

Как измеряется успех вашей деятельности?

Какие проблемы влияют на успешность вашей деятельности?

Какие тенденции, если такие существуют, делают вашу работу проще или сложнее?

Часть II. Оценка проблемы

Для каких проблем (прикладного типа) вы ощущаете нехватку хороших решений?

Назовите их. (Замечание. Не забывайте спрашивать: "А еще?..")

Для каждой проблемы выясните следующее.

- Почему существует эта проблема?
- Как она решается в настоящее время?
- Как заказчик (пользователь) хотел бы ее решать?

Часть III. Понимание пользовательской среды

Кто такие пользователи?

Какое у них образование?

Каковы их навыки в компьютерной области?

Имеют ли пользователи опыт работы с данным типом приложений?

Какая платформа используется?

Каковы ваши планы относительно будущих платформ?

Используются ли дополнительные приложения, которые имеют отношение к данному приложению? Если да, то пусть о них немного расскажут.

Каковы ожидания заказчика относительно практичности продукта?

Сколько времени необходимо для обучения?

В каком виде должна быть представлена справочная информация для пользователя (в интерактивном или в виде печатной копии)?

Часть IV. Резюме (перечисляются основные пункты, чтобы проверить, все ли правильно вы поняли)

Итак, вы сказали мне

(перечислите описанные заказчиком проблемы своими словами)

-
-

-

Адекватно ли этот список представляет проблемы, которые имеются при существующем решении?

Какие еще проблемы (если такие существуют) вы испытываете?

Часть V. Предположения аналитика относительно проблемы заказчика

(проверенные или непроверенные предположения)

(те проблемы, которые не были упомянуты) Какие проблемы, если они есть, связаны с (перечислите все потребности или дополнительные проблемы, которые, по-вашему, может испытывать заказчик или пользователь)

-
-
-

Для каждой из указанных проблем выясните следующее.

- Является ли она реальной?
- Каковы ее причины?
- Как она решается в настоящее время?
- Как бы заказчик (пользователь) хотел ее решать?
- Насколько важно для заказчика (пользователя) решение этой проблемы в сравнении с другими, упомянутыми им?

Часть VI. Оценка предлагаемого вами решения (если это уместно)

(Охарактеризуйте основные возможности предлагаемого вами решения. А потом задайте пользователю следующие вопросы.)

Что, если вы сможете

-
-

Как вы расцениваете важность этого?

Часть VII. Оценка возможностн

Кто в организации нуждается в данном приложении?

Сколько пользователей указанных типов будет использовать его?

Насколько значимо для вас успешное решение?

Часть VIII. Оценка необходимого уровня надежности и производительности, а также потребности в сопровождении

Каковы ваши ожидания относительно надежности?

Какой, по-вашему, должна быть производительность?

Будете ли вы заниматься поддержкой продукта или этим будут заниматься другие?

Испытываете ли вы потребности в поддержке?

Что вы думаете о доступе для сопровождения и обслуживания?

Каковы требования относительно безопасности?

Какие требования относительно установки и конфигурации?

Существуют ли специальные требования по лицензированию?
Как будет распределено программное обеспечение?
Есть ли требования на маркировку и упаковку?

Часть IX. Другие требования

Существуют ли законодательные требования, требования информационной среды, инструкции или другие стандарты, которых необходимо придерживаться?
Нет ли других требований, о которых нам следовало бы знать?

Часть X. Окончание

Существуют ли другие вопросы, которые мне следовало бы вам задать?
Если мне еще понадобится задать вам несколько вопросов, могу ли я вам позвонить?
Будете ли вы принимать участие в обсуждении требований?

Часть XI. Заключение аналитика

После интервью, пока его данные еще свежи в вашей памяти, зафиксируйте три потребности или проблемы с наивысшими приоритетами, выявленные вами в беседе с данным заказчиком (пользователем).

- 3.
- 4.
- 5.

Рис. 9.1. Обобщенное практическое контекстно-свободное интервью

Момент истины: интервью

После небольшой подготовки, со структурированным интервью в кармане, любой член команды может вполне удовлетворительно выполнить задание по интервьюированию пользователя или заказчика. (Но лучше выбрать тех членов команды, которые наиболее коммуникабельны.) Ниже приводятся некоторые советы для успешного проведения интервью.

- Подготовьте соответствующее контекстно-свободное интервью и занесите его в записную книжку, чтобы сверяться во время беседы. Просмотрите вопросы непосредственно перед беседой.
- Перед интервью ознакомьтесь с информацией о клиенте, с которым проводится беседа, и его компании. Не задавайте ему вопросы, ответы на которые можно узнать заранее. С другой стороны, стоит кратко сверить эти ответы с опрашиваемым.
- Кратко записывайте ответы в вашу записную книжку во время беседы. (Не пытайтесь в это время сохранить эти данные в электронном виде!)
- Сверяйтесь с образцом во время интервью, чтобы убедиться, что задаете правильные вопросы.

Не страшно, если вы немного отклонитесь от темы. Главное — не забудьте о цели вашей беседы.

Интервьюер должен стараться, чтобы сценарий не был чересчур жестким. После того как установится дружеская атмосфера, беседа может развиваться по своим собственным законам. Увлеченный потоком сознания клиент будет описывать с красочными подробностями весь ужас существующего положения. Это именно то, чего вы добивались! Если это произойдет, не спугните его в самом начале, задав следующий запланированный вопрос; записывайте все как можно быстрее, пока он не исчерпает свой поток мыслей! Уточните полученную информацию с помощью дополнительных вопросов. И лишь затем, когда эта тема подойдет к своему логическому концу, вернитесь к вопросам из списка.

Уже после нескольких таких интервью разработчик/аналитик получит некоторые знания о проблемной области и будет иметь углубленное понимание как решаемой проблемы, так и представлений пользователя об успешном решении. Кроме того, разработчик может подытожить основные потребности пользователя или функции продукта, которые были определены в интервью. Эти “потребности пользователя” находятся в верхней части нашей пирамиды требований и являются движущей силой всей последующей работы.

Сбор данных о потребностях

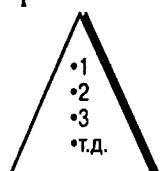
Аналитик должен выявить основных пользователей и заинтересованных лиц, которых необходимо опросить для достижения понимания потребностей клиентов. Как правило, не нужно проводить много интервью, чтобы получить о них достаточно хорошее представление.

Заключение аналитика: $10+10+10 \neq 30$

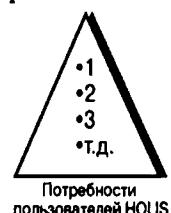
Последний раздел интервью, заключение аналитика, используется для записи “трех наиболее важных потребностей или проблем”, выявленных в ходе данной беседы. Как правило, уже после нескольких бесед эти высокоприоритетные потребности начинают повторяться. Это означает, что вы вышли на некие общие потребности. Такой результат является ожидаемым, особенно для тех пользователей и заинтересованных лиц, которые смотрят на проблему с одной точки зрения. Таким образом, десять интервью, как правило, дают только 10–15 различных потребностей. Они служат началом “архива требований”, множества создаваемых наработок, которые затем будут использоваться в процессе работы над проектом. Эти данные, получить которые совсем несложно, помогут вам и вашей команде заложить более основательный фундамент для начала вашего проекта.

Рабочий пример

Команда проекта HOLIS решила поручить группе маркетинга (Эрику и Кэти) разработать вопросы для интервью и выразила желание, чтобы все члены команды ознакомились с процессом и имели возможность встретиться с клиентом лицом к лицу, чтобы таким образом увидеть проблему и возможное ее решение с точки зрения клиента. Команда



поделила список клиентов и дистрибуторов и выделила каждому члену команды двух человек для проведения интервью. Для составления итогового списка потребностей команда использовала представленные “заключения аналитика”, исключив дублирующие друг друга записи. После проведения пятнадцати интервью было выявлено примерно 20 потребностей, которые следовало поместить на вершину пирамиды требований.



С точки зрения домовладельца

- Гибкое и модифицируемое управление освещением всего здания
- “Защищенность от будущего” (“Поскольку технология меняется, мне бы хотелось обеспечить совместимость с технологиями, которые могут появиться.”)
- Привлекательность, ненавязчивость, эргономичность
- Полностью независимое и программируемое (или реконфигурируемое) включение освещения в каждой комнате
- Дополнительная безопасность и отсутствие головной боли у владельца
- Интуитивно понятная организация работы (“Я бы хотел иметь возможность объяснить, как система работает, моей маме, страдающей технофобией.”)
- Разумная стоимость системы и низкая стоимость переналадки
- Простой и недорогой ремонт
- Гибкие конфигурации схем включения (от одной до семи “кнопок” в схеме)
- С глаз долой – из сердца вон (не требует постоянного наблюдения)
- 100%-ная надежность
- Возможность установки безопасного режима на время длительного отсутствия
- Возможность создания декораций, например специальный режим освещения всего дома во время вечеринки
- Система не должна повышать уровень риска пожара или поражения электрическим током в доме
- Возможность после аварийного отключения электроэнергии восстановить освещение на прежнем уровне
- Возможность программировать систему самостоятельно, с помощью своего ПК
- Возможность установки переключателей, допускающих изменение яркости, там, где я пожелаю
- Возможность программировать систему самостоятельно, без помощи ПК
- Пусть кто-либо другой программирует ее для меня
- Если система выйдет из строя, я хочу иметь возможность включать кое-где свет
- Интерфейсы для моей домашней системы безопасности
- Интерфейсы с другими домашними автоматами (системой вентиляции, аудио/видео и т.д.)

С точки зрения дистрибуторов

- Предложение конкурентоспособного продукта
- Наличие некой выделяющей его характеристики
- Простота обучения продавцов
- Возможность демонстрировать продукт в моем магазине
- Высокая разность между себестоимостью и ценой продажи

Замечание по поводу анкетирования

Ничто не может заменить интервьюирование.

- Проводите его первым делом!
- Проводите его для каждого нового класса проблем!
- Проводите его для каждого нового проекта!

Нас часто спрашивают, может ли команда заменить процесс интервьюирования анкетированием. Иногда это продиктовано желанием добиться большей эффективности ("Я мог бы сделать 100 анкет за время, потраченное на проведение одного интервью"). В других случаях сама необходимость беседы ставится под сомнение ("Неужели мне нужно разговаривать с этими людьми? Почему бы не послать им письмо?").

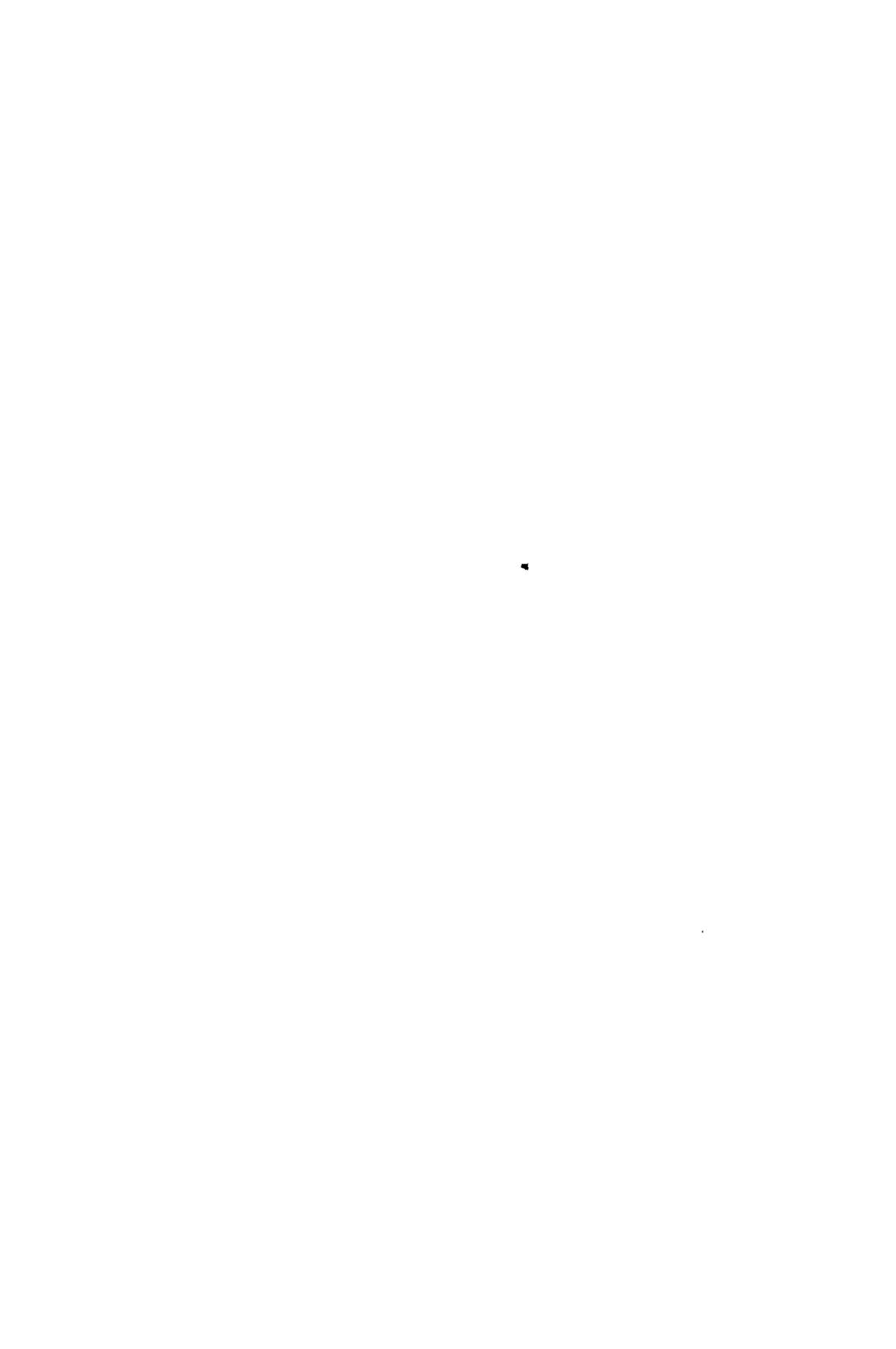
Неважно, какова мотивация, ответом является *нет*. Не существует замены личному общению, достижению взаимопонимания и взаимодействию в форме интервью. Мы уверены в том, что после одного или двух интервью вы измените свое мнение. Что еще важнее, вместе с этим изменится и ваше представление о решении. Проводите интервью первым делом! Проводите его для каждого нового класса проблем, для каждого нового проекта!

При правильном использовании метод анкетирования также может сыграть положительную роль в выяснении потребностей пользователя. Хотя данный метод применяется часто и выглядит вполне научно благодаря возможности статистического анализа результатов опроса, все же он не может заменить интервьюирование. При выявлении требований метод анкетирования имеет ряд фундаментальных недостатков.

- Относящиеся к делу вопросы невозможно разработать заранее.
- Предположения, стоящие за вопросами, оказывают влияние на ответы.
- Пример. Соответствует ли данный класс вашим ожиданиям? Предположение: у вас есть некие ожидания, т.е. это значащий вопрос.
- Трудно исследовать новые области и нет обратной связи, чтобы дополнительно исследовать те области, которые в этом нуждаются.
- Неясные ответы пользователя трудно интерпретировать.

Анкетирование можно использовать для проверки предположений и сбора статистических данных о предпочтениях.

Из этого можно заключить, что метод анкетирования не приемлем для сбора требований и мы вообще не рекомендуем его использовать. Однако это не совсем так. Метод анкетирования можно с пользой применять в качестве вспомогательного средства после проведения интервьюирования и анализа его результатов. Например, если у приложения много существующих или потенциальных пользователей и цель состоит в том, чтобы получить статистические данные о предпочтениях пользователей (или клиентов) среди ограниченного множества вариантов, можно с успехом использовать анкетирование для получения значительного объема необходимых данных за короткий период времени. Говоря коротко, метод анкетирования, как и все другие методы выявления требований, подходит для некоего подмножества задач, с которыми может столкнуться организация.



Глава 10

Совещания, посвященные требованиям

Основные положения

- Посвященное требованиям совещание является, пожалуй, самым мощным методом выявления требований.
- Оно собирает вместе всех основных заинтересованных лиц на короткий, но интенсивно используемый период.
- Приглашение стороннего ведущего, имеющего опыт в управлении требованиями, может помочь успешному проведению совещания.
- Наиболее важной частью совещания является мозговой штурм.

Ускорение процесса принятия решений

Одна из используемых нами схем определения приоритетов чрезвычайно проста. Клиента спрашивают, какой бы пункт он выделил, если бы в следующей версии реализовалась только *одна* функция. Как и мысль о том, что завтра ты будешь казнен, этот вопрос фокусирует внимание на том, что действительно важно.

Если бы нам разрешили выбрать всего один метод для выявления требований, который нужно было бы применять в любых обстоятельствах, мы бы выбрали посвященное требованиям совещание. Такое совещание по праву может считаться наиболее мощным методом из рассматриваемых в данной книге; кроме того, это один из немногих методов, который при правильном применении может реально изменить результаты проекта.

Совещание созывается для достижения консенсуса в вопросах определения требований к приложению и быстрого принятия решения о том, в каком направлении действовать; причем все это осуществляется за очень короткий промежуток времени. Основные заинтересованные лица собираются на небольшой, но интенсивно используемый период; как правило, не более чем на 1–2 дня. Совещание проводится одним из членов команды или, что даже лучше, опытным ведущим со стороны. Оно посвящено созданию или пересмотру высокочувствительных функций, которые должно предоставлять новое приложение.

Хорошо проведенное совещание по вопросам требований имеет множество преимуществ.

- Оно помогает создать команду, подчиненную одной общей цели — успеху *данного* проекта.
- Все заинтересованные лица получают возможность высказать свое мнение, никто не остается в стороне.

- Оно формирует соглашение между заинтересованными лицами и командой разработчиков по поводу того, что должно делать приложение.
- Оно может высветить и разрешить политические вопросы, которые влияют на успех проекта.
- Результат, предварительное определение системы на уровне функций, немедленно становится известным.

Многие организации достигли большого успеха в использовании этого метода. Мы участвовали во многих подобных совещаниях, и крайне редко не удавалось достигнуть поставленных целей. Совещание предоставляет заинтересованным лицам из различных организаций уникальную возможность работать вместе над достижением общей цели.

В данной главе описывается, как планировать и проводить успешное совещание, посвященное требованиям. В конце главы этот метод применяется к нашему рабочему примеру — разработке системы HOLIS.

Подготовка к совещанию

Соответствующая *подготовка к совещанию* является основой успеха.

Распространение концепции

Надлежащая подготовка является залогом успеха совещания.

Первым делом необходимо распространить идею внутри организации, разъясняя преимущества проведения совещаний будущим членам команды. Как правило, это несложно, но нередко можно встретить возражения типа: “Больше никаких собраний!”, “Мы не сможем собрать всех нужных людей вместе на один день”, “Вам никогда не удастся заставить принять в нем участие [имя вашего любимого клиента].” Не расстраивайтесь, если вы организуете совещание, все (включая скептиков) придут.

Гарантия участия основных заинтересованных лиц

Подготовка включает в себя также выявление заинтересованных лиц, которые могут повлиять на процесс и чьи потребности необходимо учесть, чтобы гарантировать успешный результат. Заинтересованные лица уже выявлены, если команда проводила этапы анализа проблемы, но сейчас необходимо еще раз вернуться к этому вопросу, чтобы удостовериться, что выявлены *все* важные заинтересованные лица.

Логистика

Необходим добросовестный подход к логистике, и он принесет свои плоды, так как плохо организованное совещание вряд ли достигнет желаемого результата. Логистика включает в себя все: от создания соответствующего приглашения до организации приезда участников и освещения помещения, в котором проходит совещание. При этом нужно исходить из справедливости закона Мэрфи “Все, что может сломаться — сломается”. Если вы подойдете к логистике с высокой степенью профессионализма, для участников бу-

дет очевидно, что это действительно важное событие, и они будут вести себя соответственно. В результате совещание будет более успешным.

Подготовительные материалы

Необходимо заранее разослать подготовительные материалы, чтобы подготовить участников, а также повысить производительность проводимого совещания. Эти материалы направляют образ мыслей участников. Мы называем это “правильной настройкой мышления”. Информация, которую необходимо довести до сведения участников в первую очередь, заключается в том, что *это не еще одно очередное собрание*. Это, может быть, *наш единственный шанс сделать все как следует*.

Подготовительные материалы должны стимулировать как конкретное, так и свободное мышление.

Мы рекомендуем вам предложить отдельно два вида подготовительных материалов.

1. *Информация, относящаяся к данному конкретному проекту.* Она может содержать планы описывающих требования документов, списки предлагаемых функций, копии интервью с будущими пользователями, доклады аналитиков о наблюдаемых в отрасли тенденциях, письма от клиентов, доклады об ошибках существующей системы, новые директивы по менеджменту, новые данные маркетинга и т.д. Хотя важно не завалить будущих участников данными, также важно удостовериться, что они располагают необходимой информацией.
2. *Информация для подготовки свободного мышления.* Правильная настройка мышления побуждает участников думать свободно (выйти из рамок, “out of box”). “Забудьте на минуту то, что вы знаете и что невозможно сделать из-за политических моментов.” “Забудьте, что в прошлый раз нам не удалось нанять хороших менеджеров.” “Забудьте, что мы еще не согласовали наш процесс разработки.” “Просто сберите воедино все соображения о функциях этого нового проекта и приготовьтесь думать непредвзято.” Как руководитель совещания, вы можете помочь этому процессу, предлагая стимулирующие мышление и процесс творчества статьи, а также правила проведения мозгового штурма, разработки требований, управления масштабом и т.д. В такой атмосфере более вероятно появление творческих решений.

Предупреждение. Не посыпайте данные слишком рано. Вы же не хотите, чтобы участники прочитали их и забыли, и не хотите, чтобы длительная подготовка снизила их осознание срочности происходящего. Посыпайте данные за 2–7 дней. По всей вероятности, участники все равно прочитают план в последнюю минуту. Все нормально! Это поможет им правильно настроить свои мысли для совещания.

Мы предлагаем образец уведомления о совещании (рис. 10.1); в скобках описаны некоторые проблемы, которые могли уже к этому моменту возникнуть в проекте, и показано, как совещание может помочь в их решении.

Уведомление

Кому: участникам проекта _____

Предмет: предстоящее совещание, посвященное требованиям

От:

Я являюсь администратором проекта _____. Проект инициирован (или будет инициирован) ____ и будет завершен _____.
(Мы настроены закончить его вовремя.)

Как и в большинстве проектов, оказалось сложно достигнуть консенсуса по новым функциям данного приложения и определить исходный базовый уровень версии, который удовлетворяет потребности разнородной группы участников.

(Не хотелось бы и далее по каждому поводу забрасывать участников группы вопросами, поэтому мы хотим попробовать нечто иное, и вот в чем оно заключается...)

Чтобы упростить данный процесс, мы проведем совещание, посвященное требованиям, которое состоится _____.

Цель совещания состоит в окончательном определении новых функций базового уровня следующей версии продукта. Для того чтобы это сделать, важно выслушать мнения всех участников. Совещание будет проводить _____, который обладает значительным опытом в сфере разработки требований.

(Поскольку мы, как и участники, также можем иметь предубеждения, нам будет помогать человек, не являющийся членом команды, чтобы гарантировать, что совещание будет проходить непредвзято.)

Результаты совещания будут известны немедленно и будут доведены до групп разработчиков и маркетинга на следующий день. Мы приглашаем вас принять участие в совещании и внести свои предложения, которые будут представлять потребности вашего [клиента, отдела, команды]. Если вы не можете приехать, мы настоятельно рекомендуем вам прислать члена вашей команды, который уполномочен принимать решения, представляющие ваши потребности.

(Мы собираемся начать разработку буквально на днях. Если вы хотите, чтобы ваши пожелания были учтены в этом проекте, ваше присутствие обязательно или же пришлите кого-нибудь, уполномоченного говорить от вашего имени. Другими словами, говорите сейчас или никогда.)

Вместе с этим уведомлением вам высыпается краткое описание предполагаемых функций продукта, а также некоторые справочные материалы о данном совещании и процессе мозгового штурма. Совещание начнется в 8:30 А.М. и будет продолжаться до 5:30 р.п.

(Данное совещание (как и проект в целом) будет проводиться профессионально; чтобы продемонстрировать это, мы предлагаем справочные материалы, которые помогут вам лучше подготовиться. Мы нуждаемся в вашем присутствии, вашем участии и вашей помощи, чтобы надлежащим образом приступить к данному проекту.)

Мы надеемся, что вы примете участие в совещании.

Искренне ваш

[Руководитель проекта]

Рис. 10.1. Образец уведомления о проведении совещания, посвященного требованиям

Роль ведущего

Для гарантии успеха мы рекомендуем, чтобы совещание проводилось сторонним человеком, имеющим опыт в решении уникальных задач управления требованиями. Если это не практикуется в вашей среде, то совещание может проводиться членом команды, но *в том и только в том* случае, если этот человек обладает следующими качествами.

- Был обучен данному процессу
- Продемонстрировал нерядовые способности в достижении консенсуса или создании команды
- Его авторитет признается как внутренними, так и внешними членами команды
- Достаточно энергичен, чтобы руководить совещанием

Если возможно, постарайтесь найти ведущего, не являющегося членом команды.

Если совещание проводится членом команды, этот человек *не должен* вносить свои идеи и участвовать в обсуждении. Иначе существует опасность, что совещание утратит необходимую для получения реальных фактов объективность и не будет способствовать созданию атмосферы, в которой можно достигнуть консенсуса.

В любом случае ведущий играет ключевую роль в успехе совещания. Возможно, все основные участники собираются вместе в первый и последний раз за время работы над проектом, и вы не можете допустить, чтобы совещание прошло впустую. Некоторые из обязанностей ведущего перечислены ниже.

- Задать профессиональный и объективный тон встречи.
- Начать и окончить совещание вовремя.
- Установить правила проведения встречи и добиваться их выполнения.
- Предложить цели встречи и повестку дня.
- Управлять течением дискуссии и удерживать команду “на правильном пути”.
- Способствовать процессу принятия решения и достижения консенсуса, но избегать участия в содержательной части дискуссии.
- Заниматься всеми организационными вопросами и вопросами логистики, чтобы основное внимание было сконцентрировано на повестке дня.
- Удостовериться, что все заинтересованные лица участвуют и их пожелания учтены.
- Контролировать поведение, которое ведет к расколу или мешает продуктивной работе.

Составление повестки дня

Повестка дня совещания будет зависеть от потребностей конкретного проекта и содержания обсуждаемых на совещании вопросов. Ни одна повестка дня не в состоянии учесть абсолютно все. Однако большинство организованных совещаний, посвященных требованиям, имеют достаточно стандартную форму. В табл. 10.1 представлена типичная повестка дня.

Таблица 10.1. Образец повестки дня совещания по вопросу требований

Время	Пункт повестки дня	Описание
8:00–8:30	Открытие	Повестка дня, организация и правила
8:30–10:00	Обсуждение контекста	Состояние проекта, потребности рынка, результаты интервью и т.д.
10:00–12:00	Мозговой штурм	Интенсивное обсуждение функций приложения
12:00–1:00	Обеденный перерыв	Продолжая работать во время обеда, можно не потерять ход мыслей
1:00–2:00	Мозговой штурм	Продолжение
2:00–3:00	Определение функций	Запись определений функций посредством 2–3 предложений
3:00–4:00	Отбор идей и расстановка приоритетов	Расстановка приоритетов функций
4:00–5:00	Завершение	Подведение итогов и другие завершающие действия

Проведение совещания

Проблемы и приемы

Вы можете убедиться, что ведущий играет ключевую роль в совещании. Следует отметить, что подобные совещания часто характеризуются весьма напряженной атмосферой. Другими словами, существуют причины, из-за которых сложно достигнуть согласия в проектах; и практически *все* эти причины будут представлены на совещании.

Действительно, аудитория может даже быть политически предубежденной и враждебной. Это еще один аргумент в пользу стороннего ведущего; пусть ведущий успокоит разгоряченных участников и ведет встречу так, чтобы не обострить никакие проблемы — прошлые, настоящие или будущие — среди них.

Многие ведущие приносят с собой “чемодан безделушек”, помогающих им управлять столь взрывоопасной атмосферой. В корпорации RELA мы применяли набор очень полезных “совещательных билетов”. Хотя они кажутся весьма странными и несерьезными на первый взгляд, уверяем вас, что они доказали свою полезность в самых разнообразных обстоятельствах. Чем сложнее совещание, тем более полезными они становятся! Они стимулируют непредвзятое мышление. Более того, они забавны и способствуют заданию положительного тона встречи. На рис. 10.2 представлено несколько образцов “совещательных билетов”. Вы можете свободно использовать их вместе с “инструкциями”.

В табл. 10.2 описаны некоторые проблемы, которые могут возникнуть в ходе совещания, и способы их решения с помощью совещательных билетов. Ведущий должен в начале встречи ознакомить участников с правилами и, в идеале, получить согласие использовать сегодня эти “дурманские” билетики.



Правило. Каждый участник изначально получает один билет на опоздание.
Истратив его, участник вносит \$1 в штрафную коробку.
Цель. Экономия времени.



Правило. Каждый участник вначале получает один билет, позволяющий "пнуть" некого человека или отдел.
После этого он должен будет вносить \$1 в коробку штрафов.
Цель. Это немного смешно и предостерегает людей от политических выступлений в проекте.



Правило. Участник получает два билета. Он передает его любому участнику, который предлагает хорошую идею.
Задача состоит в том, чтобы раздать эти билеты.
Цель. Побудить и вознаградить творческое мышление.



Правило. Участник расходует билет в любое время.
Ведущий предоставляет ему слово и устанавливает таймер.
Все слушают. Никто не перебивает!
Цель. Создание структурированного процесса представления слова. Предоставить каждому право высказать свое мнение.

Рис. 10.2. Совещательные билеты

Таблица 10.2. Проблемы в ходе совещания и их решения

Проблема	Решение
Управление временем	У ведущего может стоять кухонный таймер, который отмечает время перерывов. Опоздавший должен отдать свой билетик “опоздание с перерыва”, если он у него есть, или внести \$1 в “штрафную” коробку
■ Трудно возобновлять работу после перерывов или обеда ■ Основные участники опаздывают	
Доминирующие позиции отдельных участников	Ведущий предлагает использовать билеты “5-минутное выступление” для регулирования права на высказывание. Также создается “парковочный” список идей, которые заслуживают обсуждения, но не входят в повестку дня
Недостаток предложений от участников	Ведущий предлагает участникам использовать свои билеты “5-минутное выступление” и “замечательная идея”. Он объясняет, что ни один участник не должен покинуть совещание, не использовав свои билеты или не получив билет “замечательная идея” от других. (<i>Предложение</i> . Придумайте простую награду за использование 5-минутного билета или получение билета “замечательная идея”)
Негативные комментарии, мелочное поведение и скрытые враждебные действия	Используйте билеты “бесплатный выстрел”, пока они не закончатся у участников, а затем пусть они делают благотворительные взносы в штрафную коробку (группа сама решает, каковы размеры этих взносов)
Ослабление энергии после обеда	Легкий ланч, легкие закуски во время перерывов в середине дня, изменение оформления комнаты, мест участников, освещения или температуры, возможно, помогут поддержать рабочую атмосферу

Мозговой штурм и отбор идей

Наиболее важная часть совещания — процесс мозгового штурма. Этот метод идеально приспособлен для совещания. Он способствует созданию творческой и позитивной атмосферы, а также дает всем участникам возможность высказать свое мнение. Мозговой штурм обсуждается в следующей главе.

Результат и продолжение

После завершения совещания ведущий раздаст его протоколы и записывает любые другие итоги. На этом работа ведущего заканчивается, и ответственность за успех вновь переходит к команде разработчиков.

Руководитель проекта несет ответственность за то, чтобы завершить все начатое на встрече и организовать информацию для распространения среди участников. Часто результаты встречи представлены простым списком идей или рекомендуемых функций продукта, который может быть немедленно передан команде разработчиков для дальнейших действий. В некоторых случаях может понадобиться провести дополнительные совещания с другими участниками или потребуются дополнительные усилия для достижения понимания идей, принятых на совещании. Выработка этих идей является сущностью всего совещательного процесса. В следующей главе мы более подробно рассмотрим эту часть.

Глава 11

Мозговой штурм и отбор идей

Основные положения

- Мозговой штурм включает в себя как генерацию, так и отбор идей.
- Наиболее творческие, инновационные идеи обычно являются результатом объединения многих, казалось бы, несвязанных идей.
- Для определения приоритетов созданных идей можно использовать различные методы голосования.
- “Живой” мозговой штурм наиболее предпочтителен, но в некоторых ситуациях может быть приемлемым и мозговой штурм посредством Web.

При проведении совещания, описанного в главе 10, или в тех случаях, когда вам нужны новые идеи или творческие решения проблем, мозговой штурм является очень полезным методом.

Возможно, к моменту проведения совещания вы уже имеете представление о функциях нового продукта. В конечном счете, проекты редко начинаются с чистого листа. Но помимо рассмотрения предложенных функций продукта, совещание предоставляет возможность получить новые предложения, а затем соединить и скомбинировать эти новые функции с ранее рассматривавшимися. Этот процесс также поможет в “нахождении ненайденных руин” и тем самым придаст нам уверенность в полноте полученных предложений и в том, что все потребности участников учтены. Как правило, часть совещания посвящена мозговому штурму новых идей и функций приложения. Мозговой штурм представляет собой набор прислов, полезных в тех случаях, когда участники собираются вместе.

Данный процесс имеет ряд очевидных преимуществ.

- Поддерживает участие всех присутствующих.
- Позволяет участникам развивать идеи друг друга.
- Ведущий или секретарь ведет запись всего хода обсуждения.
- Его можно применять при различных обстоятельствах.
- Как правило, в результате получаем множество возможных решений для любой поставленной проблемы.
- Метод способствует свободному мышлению, не ограниченному обычными рамками.

Мозговой штурм состоит из двух фаз: генерация идей и их отбор. Основная цель на этапе генерации состоит в том, чтобы описать как можно больше идей, не обязательно

глубоких. На этапе отбора главной задачей является анализ всех возникших идей. Отбор идей включает в себя отсечение, организацию, упорядочение, развитие, группировку, уточнение и т.п.

“Живой” мозговой штурм

Мозговой штурм можно производить различными способами. Ниже мы описываем простой процесс, который подтвердил свою эффективность в различных обстоятельствах. Все основные участники собираются в одной комнатае, и им раздаются материалы для заметок. Это может быть просто стопка бумаги и черный толстый маркер. Листы бумаги должны быть не менее 7×12 см и не более 12×17 см. Каждому участнику нужно выдать не менее 25 листов на каждый сеанс мозгового штурма. (Самоклеющиеся листочки или индексные карточки также подойдут.) Если используются индексные карточки, пригодятся кнопки и мягкая поверхность типа большой пробковой доски.

Затем объясняются правила проведения мозгового штурма (рис. 11.1) и определяется цель заседания.

Правила проведения мозгового штурма

1. Не допускается критика или дебаты.
2. Дайте свободу фантазии.
3. Генерируйте как можно больше идей.
4. Переделывайте и комбинируйте идеи.

Рис. 11.1. Правила проведения мозгового штурма

Ведущий также объясняет цель процесса. Может показаться, что цель, с которой начинается процесс, достаточно очевидна, но это не так. Способ постановки цели оказывает влияние на результаты заседания. Например, следующие вопросы представляют собой несколько вариантов постановки цели.

- Какими свойствами, по-вашему, должен обладать продукт?
- Какие услуги, по-вашему, должен предоставлять продукт?
- Какие параметры, по-вашему, должен отслеживать продукт?

(Цель также поможет решить, когда закончить процесс. Когда цели достигнуты и никто не может ничего добавить, *заканчивайте!*)

После того как сформулированы цели процесса, ведущий предлагает участникам высказывать свои идеи и записывать их по одной на листке. Идеи оглашаются громко, чтобы все, находящиеся в комнате, могли развивать их, т. е. предлагать связанные с ними идеи и, следя правилу 4, переделывать и комбинировать их. В этом процессе, однако, правило 1 – не критиковать и не дебатировать – должно иметь первостепенное значение. Если оно не соблюдается, процесс будет уничтожен, и многие яркие личности, чувствительные к критике, не будут чувствовать себя комфортно при выдвижении новых идей.

Замечание. В нашей практике наиболее творческие и инновационные идеи, которые действительно революционизировали концепцию продукта, были не идеями одного человека, а возникли в результате комбинирования многих, на первый взгляд, не связанных друг с другом идей различных участников. Любой процесс, способствующий этому результату, является по-настоящему мощным.

Каждый участник записывает свои идеи на листочки. Это важно по следующим причинам.

- Чтобы они были сформулированы именно словами автора.
- Чтобы гарантировать, что они не будут утрачены.
- Чтобы их можно было развить в дальнейшем.
- Чтобы предотвратить задержку в творческом процессе, которая может возникнуть, если один секретарь пытается записать все идеи на доске для всеобщего обозрения.

По мере создания идей ведущий просто собирает их и прикрепляет к стене комнаты заседаний. Опять же, никакая критика не допускается. Недопустимо сказать: “это дурацкая идея” или даже “эта идея уже есть на стене”. Единственная задача состоит в генерировании идей. Даже случайно брошенная негативная реплика может оказать уничтожающее воздействие и подавить дальнейшее участие “жертвы”. Напротив, замечания типа “Отличная идея!” очень кстати, и награда “билет за отличную идею” может стимулировать дальнейшее участие всех заинтересованных лиц. Генерация идей продолжается до тех пор, пока все стороны не почувствуют, что она дошла до своего логического конца.

Иногда во время генерации идей возникают периоды затишья. Это не значит, что пора останавливать процесс. Такое затишье заканчивается, как только появляется новая идея. Более длительные периоды затишья могут быть поводом для того, чтобы ведущий вновь повторил цель или задал аналогичный вопрос. Большинство заседаний по генерации идей длится около часа (иногда 2–3 часа). Ни при каких условиях ведущий не должен заканчивать активно идущее заседание репликой вроде: “Я знаю, что мы все делаем прекрасно, но нам необходимо заканчивать”. Для участников это означает: “Ваши идеи не так важны, как мой график”. Число возникших идей зависит от того, насколько плодотворен обсуждаемый предмет (обычно их 100–200).

Процесс подходит к концу; в какой-то момент участники просто исчерпают свои идеи. Это заметно по все более длительным перерывам между возникновением идей. Ведущий объявляет конец заседания; это подходящее время для перерыва.

Отбор идей

После завершения фазы генерации идей наступает время их отбора. Этот процесс состоит из нескольких этапов.

Отсечение

Первый этап состоит в отсечении тех идей, которые не достойны внимания. Ведущий начинает с краткого представления каждой идеи и одновременно спрашивает у собравшихся, является ли эта идея допустимой. Никому из участников не нужно защищаться или провозглашать свое авторство; каждый может поддержать или отвергнуть любую идею.

Замечание. Наличие идей, которые можно легко отсечь, является индикатором качества процесса. Отсутствие некоего количества "диких" и "сумасбродных" идей свидетельствует о том, что участники не были достаточно свободны в своих мыслях.

Ведущий спрашивает участников, заслуживает ли идея дальнейшего рассмотрения. Если это неправильная идея, то он просто удаляет ее, но если есть хотя бы малейшее несогласие среди участников, она остается в списке. Если участники обнаруживают два листка с одинаковыми идеями, эти идеи объединяются. (Это обычно предпочтительнее, чем убрать одну идею; ее автор будет узвлен.)

Группировка идей

Полезно по ходу процесса группировать аналогичные идеи. Наиболее удобно, когда участники заседания могут по желанию подходить к стене и осуществлять группировку. Взаимосвязанные идеи группируются рядом на стене. Группы получают названия в зависимости от того, по какому принципу осуществляется группировка. Например, группы могут иметь следующие названия.

- Новые функции
- Вопросы производительности
- Предложения по усовершенствованию существующих функций
- Интерфейс пользователя и вопросы простоты обращения

Группы могут быть специально ориентированы на возможности системы и способы поддержки различных типов пользователей. Например, при создании новой службы перевозки и доставки функции могут быть сгруппированы следующим образом.

- Упаковка и адресация
- Обслуживание клиента
- Маркетинг и продажи
- Услуги, основанные на Web
- Выставление счета
- Управление транспортировкой

Для любой из этих групп можно возобновить генерацию идей, если окажется, что процесс группировки стимулировал возникновение новых идей или некоторая область важных функциональных возможностей осталась неохваченной.

Определение функций

В этот момент человеку, предложившему идею, нужно предоставить возможность дать ее краткое описание. Это позволит ему более подробно описать функцию и поможет удостовериться, что участники понимают ее одинаково. Кроме того, это позволит избежать грубых ошибок в процессе определения приоритетов. Итак, ведущий называет все идеи, оставшиеся в списке, и просит авторов дать их описание, состоящее из одного предложения.

Область применения приложения	Штурмуюемая функция	Описание функции
Автоматизация домашнего освещения	"Автоматическое задание освещения"	Домовладелец может предварительно задавать основанные на времени последовательности возникновения определенных осветительных событий в зависимости от времени дня
Система ввода заказов "Быстрота" на покупку		Достаточно быстрое время ответа, чтобы не мешать проведению обычных операций
Система обнаружения неполадок	"Автоматическое уведомление"	Все зарегистрированные стороны будут уведомлены посредством электронной почты, когда что-нибудь изменится

Функция сварочного робота, такая как "автоматическая повторная сварка," может считаться достаточно описанной и не требует дальнейших объяснений. Важно не увязнуть в этом процессе; на каждую идею должно уйти не более нескольких минут. Необходимо ухватить только суть идеи.

Расстановка приоритетов

Иногда генерация идей является единственной целью, и процесс на этом заканчивается. Однако в большинстве случаев, в том числе на посвященных требованиям совещаниях, необходимо определить приоритеты идей, оставшихся после отсечения. В конечном счете, ни одна команда разработчиков не может сделать "все, что только пожелается". После того как проведена и согласована группировка, пора приступить к следующему этапу. И снова можно использовать множество методов; мы опишем два из них, которые обычно применяем сами.

Накопительное голосование: стодолларовый тест. Это простой тест; забавный, понятный и легко выполнимый. Каждому участнику выдается \$100 "идейных денег", которые можно потратить на "покупку идей". (Вы можете даже добавить в чемодан инвентаря для совещания набор "идейных долларов"). Каждому участнику предлагают записать на листе бумаги, сколько денег он выделяет на каждую идею. Затем, после того как участники проголосуют, ведущий подсчитывает результаты и предлагает порядок классификации. Можно набросать гистограмму, чтобы наглядно представить результат участникам.

Результаты накопительного голосования

Идея 1 \$380

Идея 2 \$200

Идея 3 \$180

Идея 4 \$140

Идея 5...

...

...

Идея 27...

Обычно этот процесс работает просто замечательно. Однако вам следует помнить о следующем. Во-первых, он срабатывает только однажды. Вы не можете повторно использовать его в одном и том же проекте, так как если результаты первой попытки известны,

это повлияет на мнения участников при следующем голосовании. Например, если ваша любимая функция является первой в списке, а функция, которую вы ставите на второе место, даже не получила достойного упоминания, вы можете поставить все ваши деньги на вторую. Вы уверены, что остальные участники позаботятся о том, чтобы ваша любимая функция по-прежнему осталась в списке. Во-вторых, может оказаться необходимым ограничить сумму, которую каждый может потратить на одну функцию. В противном случае хитрый участник, прекрасно знающий, что такие функции, как "быстродействие" и "простота использования," и так попадут в верхнюю часть списка, может поставить все свои деньги на "работает на платформе Mac" и поднять приоритет этой функции. С другой стороны, можно разрешить более высокий лимит, чтобы иметь возможность понять, куда поступили по-настоящему крупные взносы. Они могут представлять высокоприоритетные потребности ограниченных сообществ заинтересованных лиц.

Разбиение на категории "критическая, важная, полезная". В свое время коллега научил нас методу, который также оказался очень эффективным, особенно для небольших групп заинтересованных лиц или даже одного такого лица (например, когда нужно узнать мнение начальника о ваших приоритетах). Каждому участнику дается число голосов, равное количеству идей, каждый голос должен относиться к одной из трех категорий "критический", "важный" или "полезный". Суть метода в том, что голоса, принадлежащие каждому участнику, распределены по категориям равномерно (одна треть "критических", одна треть "важных" и одна треть "полезных"); следовательно, только одну треть идей участник может отнести к критическим.

- **Критическая.** означает обязательная. При отсутствии данной функции участник не сможет использовать систему. Без нее система не будет выполнять свою основную миссию. Поэтому нет смысла делать систему без этой функции.
- **Важная.** При отсутствии данной функции произойдет значительная потеря потребительской ценности, доли на рынке или прибыли либо уменьшится приток новых обслуживаемых клиентов. Если важные пункты не будут реализованы, некоторым пользователям продукт не понравится и они не будут приобретать его.
- **Полезная.** Это означает, что было бы хорошо ее иметь. Такая функция делает жизнь проще, систему привлекательнее и приятнее или приносит большую выгоду.

Замечание. При использовании данного метода все идеи, которые "пережили" этап отсечения, получают, как минимум, статус "полезных", что позволяет избежать обид со стороны их авторов.

Когда участников много, одна и та же функция может быть отнесена разными участниками к различным категориям, но это не страшно. Ведущий выполняет следующее действие: умножает "критические" голоса на 9, "важные" на 3, а "полезные" на единицу и подсчитывает сумму! Это распределит результаты в пользу "критических" голосов, и каждая "критическая" потребность клиента всплынет на вершину списка.

Мозговой штурм с использованием Web

До сих пор мы обсуждали процесс "живого" мозгового штурма, который очень эффективен, когда всех заинтересованных лиц можно собрать вместе и они являются относительно активными и не очень застенчивыми, ведущий – опытным, а политика заинте-

ресурсованных лиц — управляемой. Время, проведенное разработчиками и внешними участниками проекта вместе, очень плодотворно. Процесс обмена мнениями способствует взаимопониманию. Поэтому мы всегда предпочитаем проводить посвященное требованиям совещание и “живой” мозговой штурм.

Но иногда “живой” мозговой штурм невозможен. В этих ситуациях альтернативой является использование Internet или локальной сети для организации мозгового штурма посредством создания дискуссионной группы. Этот метод особенно подходит для разработки перспективных приложений, когда необходимы исследования или долгосрочные прогнозы, концепция изначально расплывчата и требуется широкий диапазон мнений большого количества пользователей и заинтересованных лиц.

При использовании этого метода руководитель проекта спонсирует почтовый сервер или Web-страницу для фиксации свойств продукта и комментариев к ним. Запись идей и комментариев может производиться как анонимно, так и с указанием авторства, в зависимости от созданной администратором схемы. Преимуществом этого метода является его перманентность; идеи и комментарии могут циркулировать в сети на протяжении значительного периода времени. При этом отражается весь процесс эволюции каждой идеи. Самой важной особенностью является совершенствование идей с течением времени.

Рабочий пример: совещание по вопросу требований к системе HOLIS 2000

Вернемся к нашему рабочему примеру. В то время, пока проводилось интервьюирование, на встрече команды разработчиков с представителями маркетинга было принято решение созвать совещание, посвященное требованиям к проекту HOLIS 2000.

Присутствующие

Команда решила не приглашать стороннего ведущего, а поручить провести совещание Эрику, директору по маркетингу. Команда также приняла решение, что ее интересы на совещании будут представлять два человека: Кэти, менеджер продукта, и Пит, менеджер разработки. Команда посчитала, что они смогут говорить от ее имени, а также смогут вносить предложения по существу, так как оба недавно стали домовладельцами. Другие члены команды не будут принимать участие в совещании, но могут посещать заседания в качестве наблюдателей, чтобы услышать мнения клиентов и непосредственно увидеть результаты.

Команда приняла решение пригласить следующих представителей четырех “классов” заказчиков.

1. Дистрибуторы: Джон, главный управляющий крупнейшего дистрибутора компании, и Ракель, генеральный менеджер эксклюзивного дистрибутора компании в Европе.
2. Дэвид, строитель типовых домов в данном районе, имеющий опыт в закупке на рынке и установке конкурирующих систем.
3. Бетти, местный поставщик электрических систем.
4. Будущие домовладельцы, найденные с помощью Бетти, которые в настоящее время строят или собираются строить жилье высокого класса.

Следующий список более подробно представляет участников совещания.



Имя	Роль	Должность	Комментарии
Эрик	Ведущий	Директор по маркетингу	
Кэти	Участник	Менеджер продукта HOLIS 2000	Лидер проекта
Пит	Участник	Менеджер разработки программного обеспечения	Ответственный за разработку HOLIS 2000
Дженифер	Участник		Будущий домовладелец
Элмер	Участник		Будущий домовладелец
Жене	Участник		Будущий домовладелец
Джои	Участник	Исполнительный директор компании Automation Equip	Крупнейший дистрибутор компании Lumenations
Ракель	Участник	Генеральный менеджер, EuroControls	Европейский дистрибутор компании Lumenations
Бетти	Участник	Президент, Krystel Electric	Местный поставщик электрических систем
Дэвид	Участник	Президент, Rosewind Construction	Домостроитель
Различные члены команды	Наблюдатели	Команда разработчиков	Все члены команды, кто может прибыть

Совещание

Перед совещанием команда подготовила следующий пакет предварительных материалов.

- Несколько свежих журнальных статей, высвечивающих тенденции в домашней автоматизации.
- Копии нескольких проведенных интервью.
- Итоговый список потребностей, которые выявлены к настоящему времени.

Эрик "освежил" свои навыки ведущего, а Кэти занялась вопросами логистики.

Заседание

Заседание проводилось в отеле возле аэропорта и началось в 8:00. Эрик огласил повестку дня и правила проведения совещания, в том числе правила использования "совещательных билетов". На рис. 11.2 представлена схема совещания.

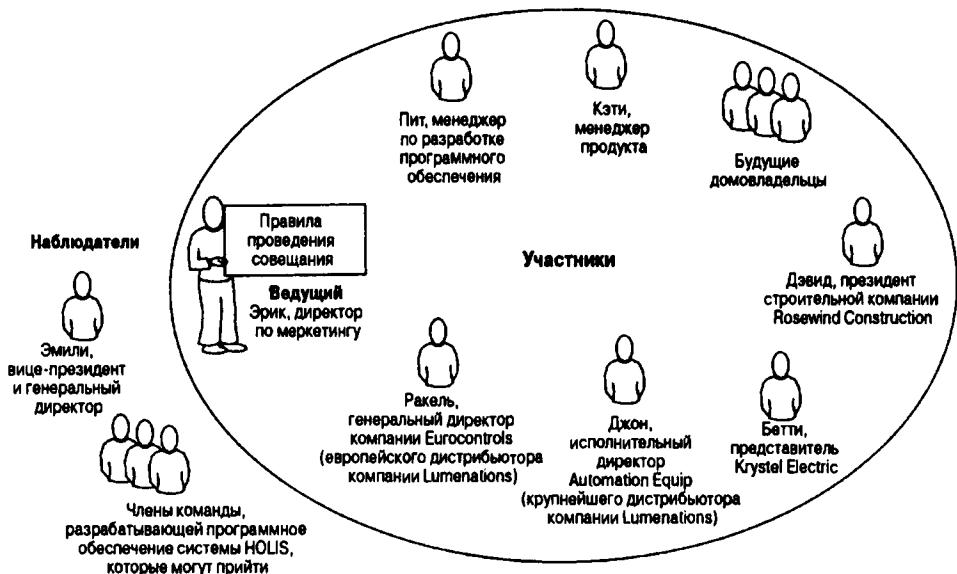


Рис. 11.2. Структура совещания по вопросам требований к проекту HOLIS 2000

В целом, совещание проходило очень хорошо, все участники имели возможность внести свои предложения так, чтобы они были услышаны. Эрик прекрасно справился с ролью ведущего, лишь однажды возник неловкий момент, когда он вступил в дискуссию с Кэти относительно приоритетов некой группы функций. (Команда пришла к решению на все следующие совещания приглашать стороннего ведущего.) Эрик провел сеанс мозгового штурма потенциальных функций HOLIS, и команда использовала накопительное голосование для выяснения относительных приоритетов. Результаты показаны в табл. 11.1.

Таблица 11.1. Определенные совещанием функции HOLIS, упорядоченные по приоритету

ID	Функции	Количество голосов
23	Возможность произвольного выбора зон освещения	121
16	Автоматическая установка длительности работы для различных источников света и т.п.	107
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105
6	100%-надежность	90
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88
1	Легко программируемые станции управления	77
5	Возможность программирования режима "жильцы в отпуске"	77
13	Любой источник света может плавно изменять яркость	74

Окончание табл. 11.1

ID	Функции	Количество голосов
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73
14	Возможность программирования работы в режиме обслуживания здравницких мероприятий	66
20	Функция закрытия гаражных ворот	66
19	Автоматическое включение света в туалете при открытии двери	55
3	Интерфейс с системой охраны дома	52
2	Простота монтажа	50
18	Автоматическое включение света, когда кто-то подходит к двери	50
7	Мгновенное включение/выключение света	44
11	Возможность управлять шторами, жалюзи, насосами и движками	44
15	Управление освещением и т.п. по телефону	44
10	Наличие интерфейса с системой управления автоматикой в доме	43
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34
26	Наличие главного блока управления	31
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25
25	Интернационализированный пользовательский интерфейс	24
21	Интерфейс с видео- и аудиосистемой	23
24	Восстановление функций после сбоя в энергоснабжении	23
17	Управление работой кондиционера	22
28	Активация голосом	7
27	Поддержка презентационного веб-сайта	4

Анализ результатов

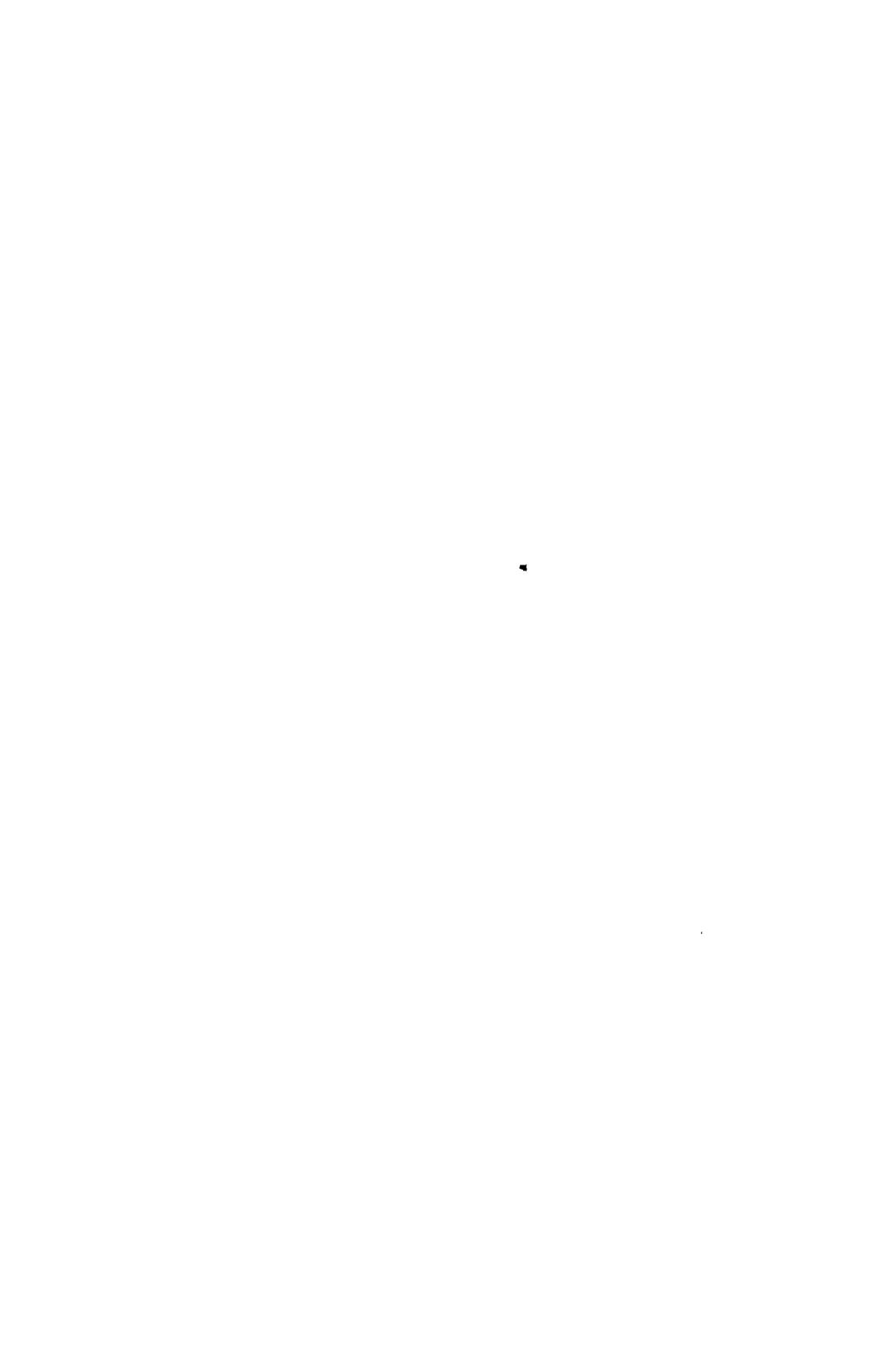
Результаты процесса можно охарактеризовать как ожидаемые, за исключением двух важных моментов.

1. Функция 4, “Встроенные средства безопасности” оказалась одной из первых в списке приоритетов. Эта функция упоминалась в предварительных интервью, но в индивидуальных списках приоритетов находилась не слишком высоко. Кэти выяснила, что встроенная безопасность (возможность гасить свет, дополнительная сирена, вызов внешних служб при возникновении опасности) практически не предлагается конкурирующими системами. Дистрибуторы прокомментировали это так, что хотя они и были удивлены таким предложением, но они считают, что это будет выгодным отличием, и согласны с тем, что данная функция должна быть высокоприоритетной. Дэвид также согласился. Основываясь на этом, отдел маркетинга

тинга решил включить данную функциональную возможность в продукт и выделить ее как уникальное дифференцирующее отличие на рынке. Она стала одной из определяющих функций системы HOLIS.

- 2. Функция 25, "Интернационализированный пользовательский интерфейс", не набрала большого количества голосов. (Этого следовало ожидать, исходя из состава участников, так как домовладельцев в США мало заботит то, насколько хорошо продукт будет продаваться в Европе.) Дистрибутор, одиако, настаивал, что если продукт не будет интернационализирован уже в версии 1.0, он не будет предлагаться в Европе. Команда зафиксировала это и согласилась приложить все необходимые усилия, чтобы добиться интернационализации в версии 1.0.¹**

¹ Этот пример демонстрирует одну из проблем, связанных с накопительным голосованием. Не все участники равны. Неудача в достижении интернационализации, которая не рассматривалась командой перед совещанием, была бы существенной стратегической ошибкой требований.



Глава 12

Раскадровка

Основные положения

- Цель раскадровки состоит в раннем выявлении реакций типа “да, но...”.
- Раскадровки могут быть пассивными, активными и интерактивными.
- Они идентифицируют игроков, объясняют, что с ними происходит, и описывают, как это происходит.
- Раскадровку следует делать краткой, легко модифицируемой и недолговечной.
- Применяйте раскадровку как можно раньше и чаще в каждом проекте с новым или инновационным содержанием.

Возможно, ни один из методов выявления требований не имеет столько интерпретаций, как “раскадровка”. Тем не менее большинство этих интерпретаций сходится в том, что целью раскадровки является получение ранней реакции пользователей на предложенные концепции приложения. При этом раскадровка предлагает наиболее эффективные методы преодоления синдрома “да, но...”. С ее помощью можно на самых ранних этапах жизненного цикла наблюдать реакцию пользователей, до того как концепции будут превращены в код, а во многих случаях даже до разработки подробной спецификации. Психологи годами твердят нам, что не следует недооценивать возможности раскадровок. В частности, киноиндустрия успешно использовала этот метод со времен появления первых картин на экране.

При создании раскадровки применяются недорогие и простые в обращении средства. Раскадровка имеет следующие преимущества.

- Предельно недорога
- Дружественна пользователю, неформальна и интерактивна
- Обеспечивает ранний анализ пользовательских интерфейсов системы
- Легко создаваема и модифицируема

Раскадровки также являются мощным средством в борьбе с синдромом “чистого листа”. Когда пользователи не знают, чего они хотят, даже примитивная раскадровка, скорее всего, сможет вызвать ответ: “Нет, это не то, мы скорее хотели бы следующее,” — и процесс пойдет.

Раскадровки можно использовать для ускорения концептуальной разработки различных граэй приложения. Их можно применять для понимания визуализации данных, определения и понимания бизнес-правил, которые будут реализованы в новом бизнес-

приложении, для определения алгоритмов и других математических конструкций, которые будут выполняться внутри встроенных систем, или для демонстрации отчетов и других результатов на ранних этапах. Раскадровки можно (и нужно!) использовать практически для всех приложений, в которых раннее получение реакции пользователей является ключевым фактором успеха.

Типы раскадровок

В основном, раскадровку можно делать такой, как пожелает команда; команда свободна в выборе способов раскадровки для конкретного приложения. Раскадровки делятся на три типа в зависимости от режима взаимодействия с пользователем: пассивные, активные и интерактивные.

- **Пассивные** представляют собой историю, рассказываемую пользователю. Они могут состоять из схем, картилок, моментальных копий экрана, презентаций PowerPoint или образцов выходной информации системы. В пассивной раскадровке аналитик играет роль системы и просто проводит пользователя по раскадровке, объясняя следующее: "Когда вы делаете это, происходит вот это".
- **Активные** пытаются показать пользователю "еще не созданный фильм". Они создаются с помощью анимации или автоматизации, возможно, посредством автоматического последовательного показа слайдов, анимационных средств или даже фильма. Активные раскадровки обеспечивают автоматизированное описание поведения системы при типовом использовании или в операционном сценарии.
- **Интерактивные** дают пользователю опыт обращения с системой почти такой же реальный, как на практике. Для своего выполнения они требуют участия пользователя. Интерактивные раскадровки могут быть имитационными, в виде макетов или могут даже представлять собой отбрасываемый впоследствии код. Сложная интерактивная раскадровка, основанная на отбрасываемом коде, может быть весьма похожа на отбрасываемый прототип (который обсуждается в следующей главе).

Как видно из рис. 12.1, эти три типа раскадровки предлагают широкий спектр возможностей — от образцов выходной информации до "живых" демонстрационных версий. Различие между сложными раскадровками и ранними прототипами продукта весьма условно.

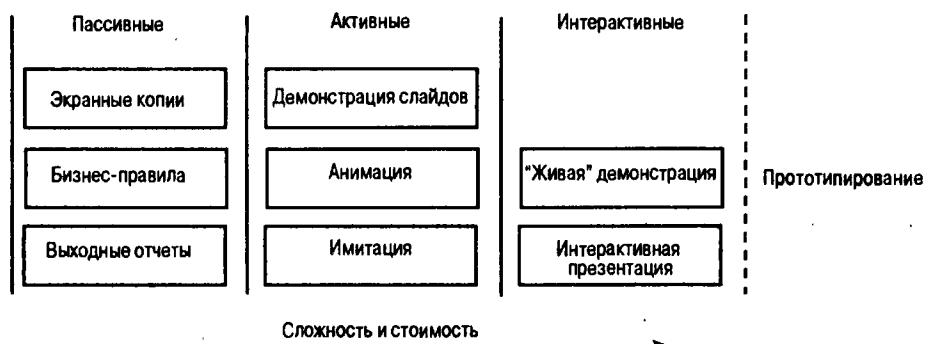


Рис. 12.1. Различные виды раскадровок

Выбор типа раскадровки зависит от сложности системы и того, насколько велик риск, что команда неправильно понимает ее назначение. Для беспрецедентной новой системы, имеющей расплывчатое определение, может потребоваться несколько раскадровок, от пассивной до интерактивной, по мере того как команда совершенствует свое понимание системы.

Что делают раскадровки

Раскадровки использовались уже в первом мультфильме Диснея *Белоснежка и семь гномов* и по сей день остаются неотъемлемой частью творческого процесса при создании кинокартин и мультфильмов. Практически все фильмы, анимационные функции и мультфильмы начинаются с раскадровок, которые представляют собой творческий материал для развития характеров и сюжетных линий.

В программировании раскадровки чаще всего используются при работе с деталями человека-машинного интерфейса, где каждый пользователь может иметь свое мнение о том, как должен работать интерфейс. Раскадровки для ориентированных на пользователя систем описывают три основных элемента любой деятельности.

1. Кто такие игроки?
2. Что с ними происходит?
3. Как это происходит?

Элемент *кто* определяет игроков или пользователей системы. В системах программного обеспечения, обсуждавшихся ранее, “кто” – это такие игроки, как пользователи, другие системы или приборы, т.е. акторы, которые взаимодействуют с создаваемой системой-решением. Для пользователей взаимодействие, как правило, описывается с помощью специально организованного экранного ввода или форм ввода данных, выходной информации в виде данных или отчетов, а также устройств ввода и вывода других типов, таких как киопки, переключатели, дисплеи и мониторы. Для приборов и систем взаимодействие будет осуществляться посредством программных или аппаратных интерфейсов, таких как коммуникационный протокол или сигнал привода контроллера мотора.

Элемент *что* представляет поведение пользователей при взаимодействии с системой или, наоборот, поведение системы при взаимодействии с пользователем. Элемент *как* описывает состояния, в которых пребывает игрок или система при взаимодействии.

Например, в свое время мы создали раскадровку для автоматически управляемых развлекательных аттракционов в парке.

- Элемент *кто* представляет посетителей аттракциона.
- *Что* представляет поведение аттракциона, предлагавшего посетителям различные события.
- *Как* демонстрирует более подробное описание того, как это взаимодействие происходит – события, переход из одного состояния в другое, – и описывает как состояния посетителей (удивление, испуг), так и состояния аттракциона (ускорение, торможение, разгрузка).

Средства и методы раскадровки

Раскадровки могут быть настолько разнообразны, насколько позволяет воображение членов команды.

Средства и методы раскадровки могут быть настолько разнообразны, насколько позволяет воображение членов команды и пользователей системы. Самые простые пассивные раскадровки можно создавать с помощью бумаги и карандаша или заметок на самоклеящихся листочках. Для построения более сложных пассивных раскадровок можно использовать администраторы представлений, такие как PowerPoint или Harvard Business Graphics. Интерактивные раскадровки пассивного и активного типа создаются с помощью HyperCard, SuperCard и различных пакетов, которые позволяют быстро разрабатывать пользовательские экраны и выходные отчеты. Интерактивные раскадровки можно создавать с помощью разнообразных специальных пакетов программного обеспечения для интерактивного прототипирования, таких как Dan Brickin's Demo It. Для создания более сложных анимаций и имитаций можно использовать такие средства, как Macromedia's Director и Cinemation компании Vividus.

Так, например, в компании RELA один из членов команды умел неплохо делать зарисовки. На концептуальной стадии разработки проекта он просто набрасывал полдюжины (или около того) простых схематических картинок, демонстрирующих варианты типичного использования продукта или различные аспекты его интерфейса. Это был быстрый и недорогой способ получить реакцию потенциальных пользователей. Кроме того, как мы увидим позднее, то, что раскадровка при этом напоминала комикс, позволяло избежать некоторых проблем, которые могут возникать при создании раскадровок. К сожалению, после того как этот человек оставил компанию, нам пришлось искать другие методы раскадровки.

Теперь, когда мы в основном занимаемся IVS-приложениями, для решения наших задач оказалось возможным ограничиться PowerPoint или другими обычными настольными администраторами представлений, используемыми совместно с образцами экранных копий, созданных с помощью тех же средств, которые применяются при построении графических интерфейсов пользователя в приложении. Следует заметить, что простое добавление анимационных возможностей к PowerPoint стало, пожалуй, самым грандиозным прорывом в технике раскадровки. С их появлением в программе способность выражать динамику и интерактивность возросла на порядок.

Советы по раскадровке

Раскадровка является мощным методом быстрого получения обратной реакции пользователя с помощью недорогих средств. Поэтому раскадровки особенно эффективны в преодолении синдрома “да, и...”. Они также помогают в разрешении синдрома “ненаходимых руин” путем обеспечения быстрой реакции пользователя на то, что система, по-видимому, “не должна делать”. Но, как и для любого другого метода, возможны определенные сложности. Ниже приводятся некоторые рекомендации, о которых необходимо помнить при использовании на практике метода раскадровки.

- *Не вкладывайте многое средства в раскадровку.* Если она будет выглядеть как реальный рабочий продукт, клиенты будут опасаться вносить в нее изменения. Поэтому хо-

рошо, если удается сделать раскадровку схематичной, даже грубой. (См. историю о раскадровке в конце данной главы.)

- *Если вы ничего не меняете, вы ничему не научитесь.* Делайте раскадровку такой, чтобы ее было легко изменять. Следует иметь возможность модифицировать раскадровку за считанные часы.
- *Не делайте раскадровку слишком хорошей.* Если вы сделаете это, клиенты захотят "получить ее" (т.е. использовать как некий готовый продукт). (В одном проекте нам пришлось много лет поддерживать Excel/VB-продукт, который изначально был не более чем раскадровкой.) Делайте раскадровку схематичной; используйте средства и методы, которые не позволяют превратить ее в реально используемый продукт. Это особенно важно для раскадровок, выполненных в форме кода. (*Совет.* Если приложение должно быть реализовано на языке Java, пишите раскадровку на Visual Basic.)
- *Если возможно, делайте раскадровку интерактивной.* Если клиент получит опыт использования продукта, это вызовет больше реакций и приведет к выявлению большего числа новых требований, чем может дать пассивная раскадровка.

Заключение

В данной главе мы рассматривали простой и недорогой метод выявления требований. Раскадровка предназначена для быстрого выявления реакций пользователя типа "да, но..." с помощью недорогих средств.

Мы можем с уверенностью сказать, что раскадровки очень полезны и всегда вносят нечто новое в понимание системы. Поэтому мы советуем команде разработчиков следующее.

- Создавать раскадровку как можно раньше.
- Делать раскадровки чаще.
- Применять раскадровки во всех проектах, имеющих новое или новаторское содержание.

При этом удастся на ранних этапах получить реакцию "да, но...", что, в свою очередь, поможет создать системы, которые лучше справляются со своей задачей по удовлетворению реальных потребностей пользователей, и сделать это быстрее и экономнее!

История о раскадровке

(Чтобы никого не обидеть, мы изменили некоторые факты о проекте в этой достаточно правдивой истории.) Эта история произошла во время разработки сложного электромеханического прибора для больничной аптеки. Заказчиком был производитель Fortune 1,000; наша компания занималась разработкой этой новой сложной электромеханической оптической инфузионной системы. Проект зашел в тупик.

Однажды главному руководителю проекта (будем называть его "автором") позвонил один из руководителей заказчика (Старший ВР, "М-р Биг"), влиятельный человек, с которым мы никогда прежде не имели удовольствия встречаться.

М-р Биг: Автор, как продвигается наш любимый проект?

Автор: Не очень хорошо.

М-р Биг: Так я и знал. Послушайте, не бывает нерешаемых проблем. Приезжайте со всей вашей командой на встречу. Как насчет среды?

Автор: (поспешно отменяя все указания для членов команды на среду) Среда подходит.

М-р Биг: Отлично. Приезжайте. Не беспокойтесь о расходах на поездку. Мы их возместим. Только покупайте билеты в один конец.

Автор: (сдержанно) Благодарю. Увидимся в среду.

В указанный день мы вошли в большой конференц-зал, где в дальнем конце уже сидели все представители команды заказчика. Видно было, что они находятся здесь уже некоторое время. (Возник вопрос: Почему команда почувствовала необходимость собраться до начала встречи?) Автор, который не в первый раз сталкивался с подобной ситуацией, прошел в другой конец комнаты и сел рядом с м-ром Бигом (рассуждая, что м-ру Бигу будет сложнее кричать на него, если он будет сидеть рядом; если же тот побьет Автора, то будет возможность выиграть судебный процесс и возместить потерянную прибыль от проекта!).

После короткой дискуссии Автор заметил, что среди многих значительных проблем, с которыми столкнулся проект, проблема "недостатка единства в требованиях" вызывает задержку проекта и превышение расходов. М-р Биг сказал: "Приведите мне пример". Автор привел отличный пример. Члены команды заказчика немедленно начали спорить между собой, демонстрируя, что это действительно являлось проблемой. Субподрядчик вздохнул с некоторым облегчением. М-р Биг с минуту посмотрел на команду и сказал: "Очень смешно. Приведите мне другой пример." Команда автора предъявила пять цветных изображений предлагаемой передней панели, каждое из которых было достаточно профессионально сделано, и заявила: "Мы представили все эти варианты дизайна несколько недель назад и до сих пор не можем получить единого мнения относительно дизайна, мы уже пришли к выводу о необходимости взимания платы за простой". М-р Биг заявил: "Этого не может быть. Команда, выберите какой-нибудь вариант." Члены команды заказчика снова вступили в перепалку друг с другом. Весь день прошел подобным образом. Согласия не было. Надежды практически не осталось.

Утром на следующий день Автора попросил о встрече некий член команды проекта ("Член команды"). Член команды, который, помимо всего прочего, умел шить, достал кусок войлока, ножницы, цветные маркеры и сказал: "Я хочу использовать эти средства, чтобы помочь в организации той части встречи, которая касается интерфейса пользователя".

Автор: Не будьте наивным; это не сработает. Это будет выглядеть глупо и непрофессионально.

Член команды: Я понимаю, но какова была эффективность вчера?

Автор, будучи человеком воспитанным, не стал произносить первое слово, которое пришло ему в голову. Вторым словом было: "Ладно".

На следующий день атмосфера в зале была совершенно иной. Команда заказчиков опять явилась заранее, но в этот раз она была менее возбужденной. (Выводы: теперь они знали, что они так же беспомощны, как и мы. Они планировали уничтожить нас, но оказалось, что мы все обречены.)

В начале встречи Член команды прикрепил на стену кусок войлока размером 1×2м, вызвав оживление (но не безучастность) со стороны заказчика.

Член команды поместил большие войлочные выключатели и кнопки различных терапевтических режимов на переднюю панель и сказал: "Как будет работать этот вариант?".

Заказчик посмотрел на стену и сказал: "А почему бы вам не переместить аварийный останов назад?".

Член команды сказал: "Почему бы вам не сделать это," и дал ножницы заказчику.

Заказчик взял ножницы, а член команды отошел в дальний конец зала. Заказчик продолжил сеанс интерактивного дизайна с помощью войлока и ножниц. Через час он посмотрел на стену и сказал: "Достаточно неплохо; пусть будет так".

Какова мораль этой истории? Опишем ее с помощью вопроса и ответа.

Вопрос: Почему обычный войлок сработал, в то время как профессиональные визуализации – нет?

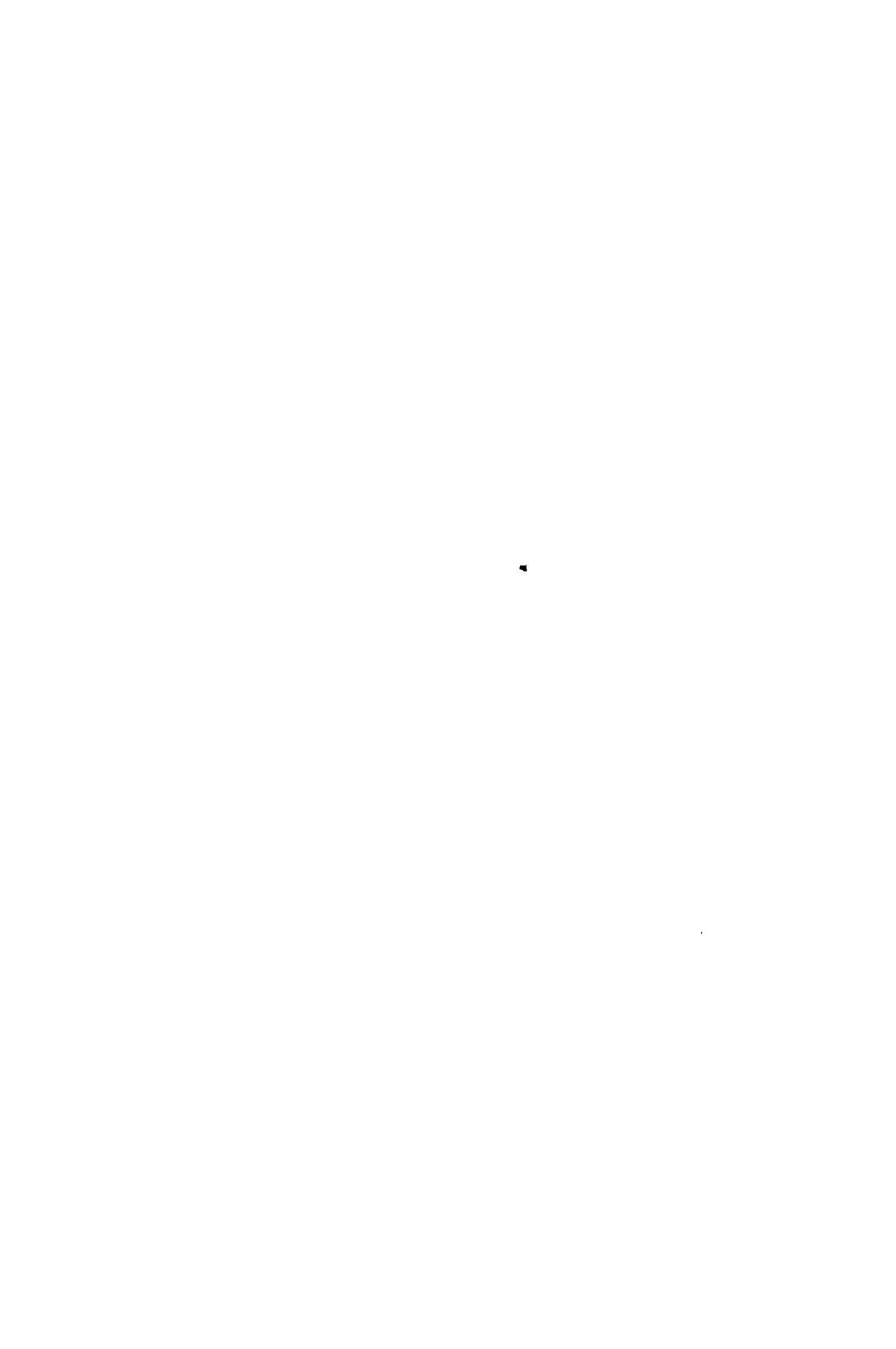
Ответ: Существует две причины.

- **Интерактивность:** что может заказчик сделать с пятью рисунками, в каждом из которых его устраивает только некоторая часть?
- **Практичность:** совсем несложно раскроить большой кусок войлока.

Заказчик, эксперт в предметной области, но не обязательно в дизайне, самостоятельно создал приемлемое решение своей проблемы.

Мы забрали с собой этот войлок и повесили его на стену в качестве постоянного напоминания о том, чему мы научились. Этот интерфейс пользователя, хотя, возможно, и не оптимальный, никогда уже больше не менялся и оказался вполне приемлемым для тех целей, для которых был предназначен. Но, увы, данный проект не стал грандиозным успехом его создателей, хотя со временем поступил на рынок и добился признания. Как мы отмечали ранее, рассматриваемая проблема была только одной из проблем данного проекта.

- **Урок 1.** Понимание потребностей пользователя – сложная и тонкая проблема. Используйте гибкие и тонкие средства – раскадровки и войлок, если это нужно, – для ее решения.
- **Урок 2.** Технология является сложной. Дважды подумайте, прежде чем приступить к разработке медицинских приборов.



Глава 13

Применение прецедентов

Основные положения

- Прецеденты, как и раскадровки, описывают элементы *кто, что и как* поведения системы.
- Прецеденты описывают взаимодействия между системой и пользователем, уделяя основное внимание тому, что система “делает” для пользователя.
- Модель прецедентов описывает функциональное поведение системы в целом.

В главе 12 описывался процесс создания раскадровок и обсуждалось, как их можно использовать, чтобы показать элементы *кто, что и как* поведения системы и пользователя. Еще один метод описания поведения состоит в использовании прецедентов. Данный метод кратко рассматривался в главах 2 и 5, где он применялся при моделировании бизнес-процесса.

В данной главе мы продолжим изучение метода прецедентов и опишем, как его можно использовать для понимания поведения системы, которую мы разрабатываем, в отличие от понимания бизнес-процесса, внутри которого будет оперировать система. Другими словами, мы будем применять прецеденты для выявления требований, чтобы понять, как должно вести себя приложение, которое мы собираемся разработать для решения проблемы пользователей. Прецеденты настолько важны для фиксации и спецификации системных требований, что мы продолжим их дальнейшую разработку в части 5, “Уточнение определения системы”, и части 6, “Построение правильной системы”.

Метод прецедентов является частью методологии объектно-ориентированного программирования (Object-Oriented Software Engineering), как описано в книге *Object-Oriented Software Engineering, A Use Case Driven Approach* (Джейкобсон (Jacobson) и др., 1992). Это метод анализа и проектирования сложных систем, представляющий собой “основанный на прецедентах” способ описания поведения системы с точки зрения того, как различные пользователи взаимодействуют с ней для достижения своих целей. Такой ориентированный на пользователя подход предоставляет возможность исследовать различные варианты поведения системы при раннем привлечении пользователя.

Как уже отмечалось ранее, прецеденты служат также UML-представлением требований к системе. Прецеденты можно успешно использовать на протяжении всего жизненного цикла программного обеспечения. Помимо фиксации требований к системе, разработанные в процессе выявления требований прецеденты можно применять при анализе и проектировании; они также могут играть значительную роль в процессе тестирования. В последующих главах метод прецедентов будет рассматриваться более подробно, а в настоящий момент необходимо только понять, как можно использовать прецеденты для фиксации исходных требований к системе.

Начнем с более формального определения, чем предлагалось ранее.

Прецедент описывает последовательность действий, выполняемых системой с целью предоставить полезный результат конкретному актору.

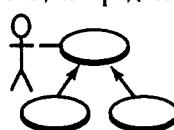
Другими словами, прецеденты описывают взаимодействие между пользователем и системой, уделяя основное внимание тому, что система “делает” для пользователя. Кроме того, поскольку действия описываются последовательно, легко “проследить действие” и понять, что делает система. В UML прецедент изображается овальной пиктограммой, содержащей имя данного прецедента.



В процессе определения требований прецеденты служат для выявления и фиксации системных требований. Каждый прецедент описывает серию событий, в которых некий актор, скажем “Дженни-Модель”, взаимодействует, например, с системой планирования работы с клиентами некоего модельного агентства, чтобы получить нужный ему результат, направление на следующий показ моделей.

Построение модели прецедентов

Модель прецедентов системы состоит из всех акторов системы и различных прецедентов, посредством которых акторы взаимодействуют с системой, тем самым описывая многообразие ее функционального поведения. Она также показывает связи между прецедентами, что углубляет наше понимание системы.



Первый шаг моделирования прецедентов состоит в создании системной диаграммы, описывающей границы системы и определяющей ее акторы. Это позволяет параллельно осуществить этапы 3 и 4 (из описанных в главе 4 пяти этапов анализа проблемы), в которых требуется выявить заинтересованных лиц системы и определить ее границы.

Например, для системы управления складом (Джейкобсон (Jacobson) и др., 1992) граница системы может выглядеть так, как на рис. 13.1.

Как видно из рисунка, система используется многими пользователями, каждый из которых взаимодействует с системой для достижения определенной операционной цели.

Дальнейший анализ системы показывает, что для поддержки потребностей пользователей необходимы определенные нити системного поведения. Эти нити и есть прецеденты, или специфические последовательности, с помощью которых пользователи взаимодействуют с системой для достижения определенных целей. Рассмотрим примеры прецедентов данной системы.

- Производимое вручную распределение изделий на складе.
- Включение новой единицы хранения.
- Проверка изделий, находящихся на складе.

Применение прецедентов к выявлению требований

Идею прецедентов можно доходчиво описать будущим пользователям системы. Прецеденты пишутся на естественном языке пользователя. Их легко описывать и докумен-

тировать. Это обеспечивает простой структурированный формат для совместной работы команды-разработчика и пользователя над описанием поведения существующей системы или определением поведения новой системы. Конечно же, каждый пользователь будет обращать внимание на те возможности системы, которые необходимы для выполнения именно его работы. Если поведение полностью исследуется всеми потенциальными пользователями, команде удастся значительно приблизиться к цели полного понимания желаемого поведения системы. После окончания данного процесса останется не так уж много «неоткрытых руин» функциональных возможностей.

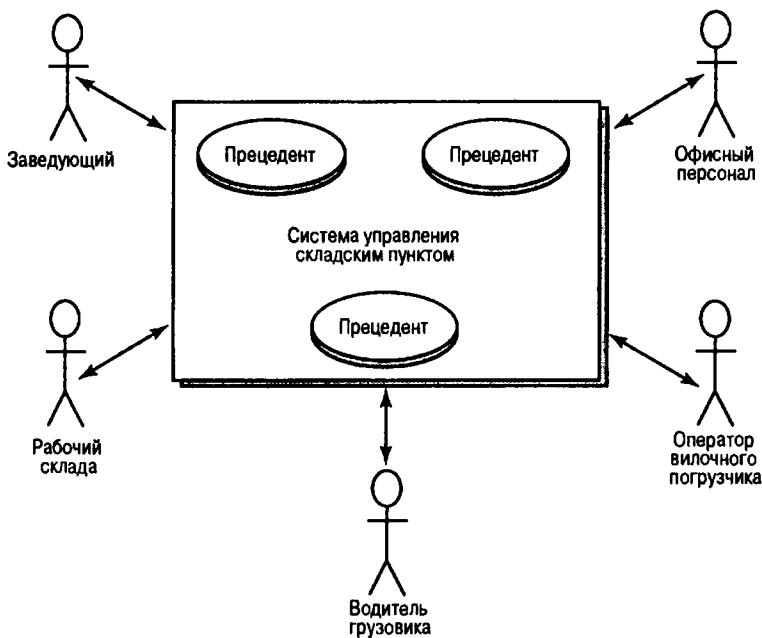


Рис. 13.1. Диаграмма складской системы с указанием акторов

Кроме того, применяя метод прецедентов для непосредственного исследования интерфейсов пользователя, можно получить ранние отклики на этот важный и изменчивый аспект системной спецификации и проекта.

Однако необходимо помнить, что пользователи системы представляют только один, хотя и важный, класс заинтересованных лиц, и нам могут понадобиться другие методы выявления для получения требований иных участников, таких как заказчики, не являющиеся пользователями, руководители, субподрядчики и т.д. Кроме того, прецеденты не очень подходят для определения нефункциональных аспектов системных требований, таких как требования практичности, надежности, производительности и т.п. При решении этих задач мы полагаемся на другие методы.

После того как выявлены все прецеденты, акторы и объекты системы, следующий шаг состоит в уточнении деталей функционального поведения каждого прецедента. Спецификации прецедентов состоят из текстовых и графических описаний каждого прецедента, написанных с позиции пользователя.

Спецификацию прецедента можно рассматривать как контейнер, описывающий последовательности связанных событий, которые, в свою очередь, могут подразумевать появление в будущем других требований. Так, спецификация прецедента может содержать такой пункт “Специалист по обслуживанию вводит свою фамилию (максимум 16 символов), имя и т.д.”.

Поскольку прецеденты описывают взаимодействие пользователь-система, параллельно с их разработкой стоит определить, хотя бы концептуально, экраны, дисплеи и передние панели, с которыми пользователь будет взаимодействовать. Если для представления информации применяется система окон, можно задать высокоуровневое графическое описание данных, которые необходимо отобразить на экране. Детали проектирования формального графического интерфейса пользователя (GUI), такие как определение данных, цвета и шрифтов, можно оставить на потом. На рис. 13.2 представлен пример фрагмента спецификации прецедента.

Прецедент: перераспределение товаров на складе
1. Заведующий дает команду перераспределить предметы на складе.
2. Заведующему предлагается окно на рис. xxx.
3. Товары можно упорядочить различными способами, которые представлены в меню.
■ Упорядочить по алфавиту
■ Упорядочить по значению индекса
■ Упорядочить по срокам хранения
4. Таблица “Откуда” позволяет выбрать режим рассмотрения (можно рассматривать все помещения склада или только те помещения, где есть указанный товар)

Рис. 13.2. Спецификация прецедента производимого вручную перераспределения товаров на складе

Рабочий пример. Прецеденты системы HOLIS

Команда разработчиков системы HOLIS решила воспользоваться методом прецедентов для описания высокоуровневых системных функций HOLIS. Команда провела сеанс мозгового штурма для определения важных прецедентов, которые заслуживают дальнейшей разработки на более поздних этапах. Это “исследование моделей прецедентов” выявило 20 прецедентов, которые следует уточнить в ходе дальнейших действий; некоторые из них приводятся ниже.

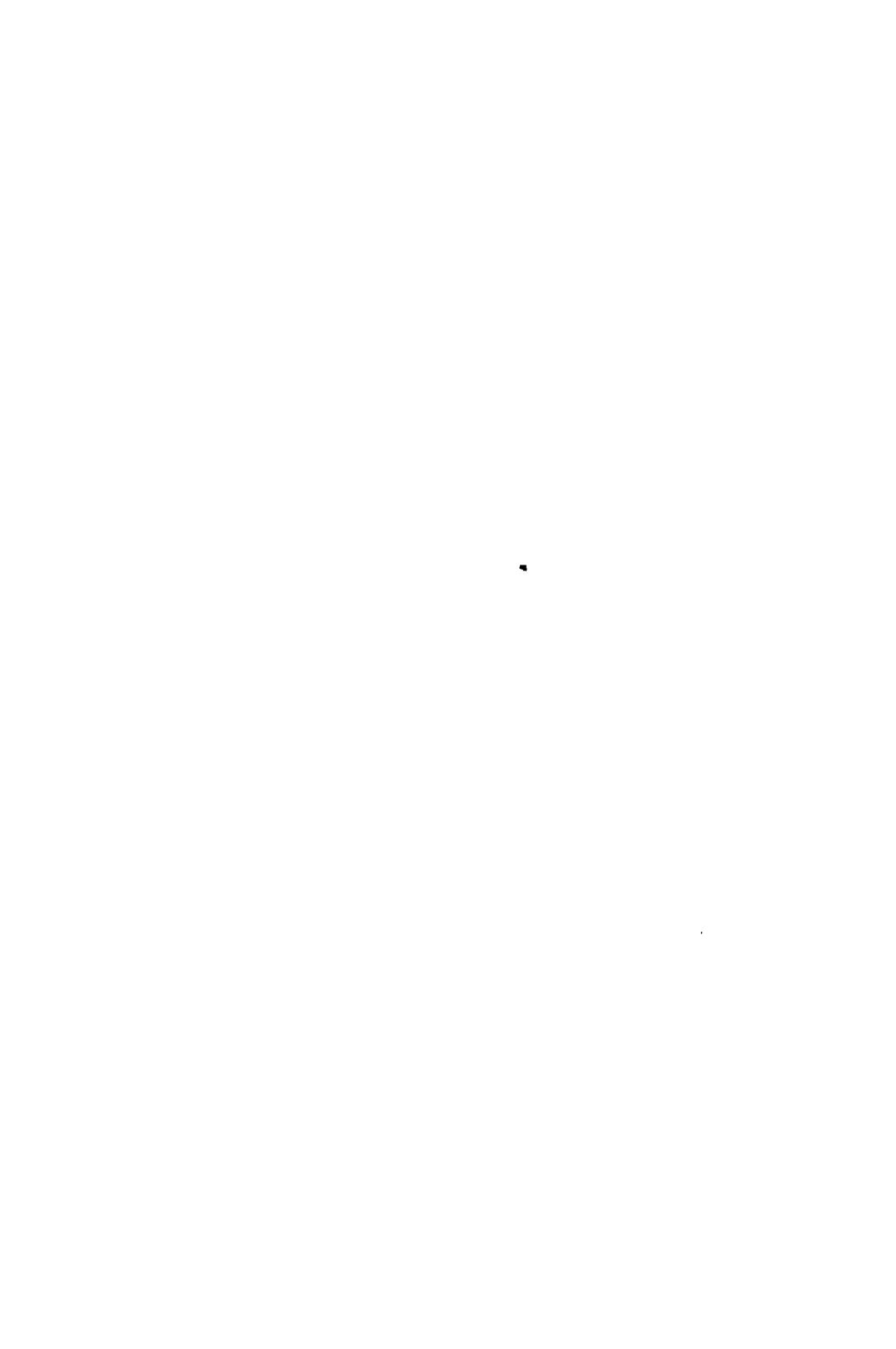
Название	Описание	Актор(ы)
Создание обычной осветительной сцены	Жилец создает обычную осветительную сцену	Жилец, лампы
Инициирование чрезвычайных действий	Жилец инициирует чрезвычайные действия	Жилец

Название	Описание	Актор(ы)
Управление освещением	Жилец включает/выключает лампу(ы) или задает желаемый уровень яркости света	Жилец, лампы
Переключение программы	Изменение или задание действий для определенной клавиши/переключателя	Домовладелец/программист
Удаленное программирование	Провайдер услуг компании Lumenations осуществляет удаленное программирование, основываясь на запросах жильца	Службы компании Lumenations
Продолжительное отсутствие	Домовладелец устанавливает специальный режим на время длительного отсутствия	Домовладелец/программист
Задание временной последовательности	Домовладелец программирует основанную на времени последовательность автоматического возникновения осветительных событий	Домовладелец/программист

Заключение

Прецеденты являются структурированной и разумно формальной нотацией для фиксации очень важного подмножества информации о требованиях: как система взаимодействует с пользователем при предоставлении своих функциональных возможностей. Во многих приложениях это подмножество составляет львиную долю рабочей нагрузки, поэтому прецеденты можно применять для выражения большей части требований к системе. Каждый выявленный прецедент определяет требуемое поведение системы с точки зрения определенного класса пользователей. Поэтому данный метод очень полезен в выявлении потребностей пользователей и помогает команде разработчиков представить эти потребности в понятной пользователем форме.

Поскольку прецеденты можно применять позднее в процессе проектирования и тестирования, они обеспечивают целостное представление и нить согласованных действий по выявлению требований, анализу, проектированию и тестированию. Таким образом, данный метод на раннем этапе создает активы проекта, которые можно повторно использовать. Кроме того, благодаря согласованности представления и поддержке, обеспечиваемой UML и различными вспомогательными средствами разработки приложений, прецеденты можно использовать для автоматизации различных действий с требованиями. Прецеденты настолько важны, что мы будем применять их, начиная с этого момента и далее, в качестве неотъемлемой составной части деятельности команды по управлению требованиями.



Глава 14

Обыгрывание ролей

Основные положения

- С помощью обыгрывания ролей команда разработчиков может почувствовать мир пользователя, поставив себя на его место.
- В некоторых случаях обыгрывание ролей можно заменить просмотром сценария, и тогда сценарий становится "живой" раскадровкой.
- Часто используемые в объектно-ориентированном моделировании CRC-карточки (Class-Responsibility-Collaboration) являются одним из средств обыгрывания ролей.

Итак, в части 2 мы уже обсудили множество методов выявления потребностей заинтересованных лиц по отношению к создаваемой системе. Мы *поговорили о ней* с глазу на глаз (интервьюирование); *обсудили ее* в групповом режиме (совещания); *представили свои идеи* (мозговой штурм) и подумали с том, как акторы *взаимодействуют с ней* (прецеденты). Все это хорошо, но мы еще не опробовали ее.

В данной главе обсуждается метод обыгрывания ролей, который позволяет команде разработчиков прочувствовать мир пользователя, побывав в его роли. Концепция, лежащая в основе данного метода, достаточно проста: хотя, наблюдая и задавая вопросы, мы повышаем уровень своего понимания, *наивно полагать, что посредством одного наблюдения разработчик/аналитик может получить истинно глубокое понимание решаемой проблемы или требований к системе, которая призвана данную проблему решить*.

Это одна из основных причин возникновения синдрома "да, но...". Как учит социология, каждый из нас видит мир через свой собственный концептуальный фильтр. Наш жизненный опыт и предубеждения невозможно отделить от производимых нами наблюдений. Например, мы можем наблюдать другую культуру и принимать участие в некой ритуальной церемонии так часто, как нам вздумается, но вряд ли нам удастся *понять*, что это значит для представителей этой культуры! Применительно к нашей задаче выявления требований это означает следующее.

- Мы должны понимать, что многие пользователи не могут описать процедуры, которые они выполняют, или потребности, которые необходимо учесть. Зачастую это не входит в их задачу, и их никогда прежде не спрашивали об этом. Кроме того, это гораздо сложнее, чем кажется! Например, попытайтесь описать процедуру, с помощью которой вы зашнуровываете туфли.

- Многие пользователи опасаются признаться, что они не следуют предписанной процедуре; следовательно, то, что они вам говорят, может являться (а может и не являться) тем, что они делают в действительности.
- Некоторые пользователи применяют нестандартные или уникальные способы выполнения рабочих действий, которые могут маскировать реальные проблемы от наблюдателя.
- Ни один разработчик не в состоянии предвидеть все вопросы, которые необходимо задать, и ни один пользователь не знает, какие вопросы должен задать разработчик.

Обыгрывание ролей может помочь при решении этих проблем. Оно не требует значительных затрат времени и средств. Обычно достаточно часа или половины дня, чтобы все стало ясно.

Как играть роль

Как правило, разработчик, аналитик или любой член команды занимает место пользователя и выполняет его обычные действия. Рассмотрим в качестве примера проблему ввода заказов на покупку. В главе 4 мы выяснили, что неправильные заказы на покупку вносят основной вклад в стоимость остатков и, тем самым, снижают рентабельность. Рассматривая существующий процесс оформления заказов на покупку, мы ожидаем найти различные источники ошибок. Существует по крайней мере два пути выявить основные причины.

1. Использовать описанный ранее метод построения диаграммы в виде "рыбного скелета" совместно с интервьюированием пользователей и проанализировать некорректные заказы. Распределить ошибки по типам и учесть наиболее типичные при проектировании новой системы. Это обеспечит количественное понимание проблемы и, возможно, будет весьма эффективно.

Однако это не позволит вам "ощутить" проблему, что могло бы изменить как ваше восприятие, так и стратегию решения. Для этого существует более простой способ.

2. Разработчик/аналитик может "прочувствовать" проблемы и неточности, присущие существующей системе ввода заказов на покупку, *просто заняв место оператора и попытавшись ввести несколько заказов*. Полученный в течение часа опыт навсегда изменит понимание командой сути проблемы.



Как мы убедились, впечатления, полученные при исполнении роли, остаются с нами навсегда. Наш взгляд на мир изменился под влиянием множества сыгранных ролей, среди которых "сварка сложных швов тем способом, как это должен делать робот", "смешивание фармацевтических составляющих в ламинарном потоке", "выявление телерекламы по четырем сжатым экранным изображениям и сжатому аудиотреку", "использование несовершенного инструментального средства управления требованиями" и др. Мы каждый раз учились чему-нибудь и проявляли гораздо больше сочувствия к пользователю, чем до этого!

Методы, аналогичные обыгрыванию ролей

Конечно, обыгрывание ролей применимо не во всех ситуациях. Во многих случаях роль пользователя минимальна, а решаемая проблема является скорее алгоритмической, чем функциональной. В других случаях это просто не практикуется. Мы бы не хотели выступать в роли "пациента" электрохирургии, "оператора ядерного реактора в ночную смену" или "пилота Боинга 747". В данной ситуации можно применить другие методы, которые позволят нам получить представление об ощущениях пользователя, без потери "крови".

Сценарный просмотр

Сценарный просмотр – это исполнение роли на бумаге.

При *сценарном просмотре* каждый участник следует сценарию, который задает конкретную роль в "пьесе". Просмотр будет демонстрировать любые неточности в понимании ролей, недостаток информации, доступной актеру или подсистеме, или недостаток конкретного описания поведения, необходимого актерам для успешного выполнения их роли.



Например, мы однажды создавали сценарный просмотр, который показывал, как студенты и преподаватели будут взаимодействовать с прибором, автоматически оценивающим тесты. Мы использовали прототип прибора и попросили членов команды и представителей заказчика разыграть роли студентов и преподавателей. Просмотр состоял из множества сцен, таких как "студенты оценивают свои собственные тесты" и "преподаватель оценивает большой пакет тестов во время занятий". В данной ситуации сценарный просмотр оказался весьма полезным; команда ощутила атмосферу занятий и получила некие новые знания. Кроме того, это было забавно!

Одним из преимуществ сценарного просмотра является то, что сценарий можно модифицировать и проигрывать снова столько раз, сколько необходимо, пока актеры не сочтут его правильным. Сценарий можно также повторно использовать для обучения новых членов команды. Его можно модифицировать и проигрывать вновь, когда необходимо изменить поведение системы. В определенном смысле данный сценарий становится "живой" раскадровкой для проекта.

CRC-карточки (Class-Responsibility-Collaboration, класс-обязанность-взаимодействие)

Один из методов обыгрывания ролей часто применяется в объектно-ориентированном анализе. В данном случае каждому участнику выдается набор индексных карточек, описывающих класс (объект); обязанности (поведение); а также взаимодействия (с какими из моделируемых сущностей взаимодействует объект). Эти взаимодействия могут просто представлять сущности проблемной области (такие, как пользователи, нажатые кнопки, лампы и подъемники) или объекты, существующие в области решения (такие, как подсветка выключателя в холле, окно многодокументного интерфейса или кабина лифта).

Когда актер-инициатор инициирует определенное поведение, все участники следуют поведению, заданному на их карточках. Если процесс прерывается из-за недостатка ин-

формации или если одной сущности необходимо переговорить с другой, а взаимодействие не определено, то карточки модифицируются, и роли разыгрываются снова.

Например, в нашем рабочем примере при разработке системы HOLIS команде нужно понять взаимодействия между тремя подсистемами, чтобы определить, как в системе осуществляется кооперация для достижения общей цели и какие производные требования при этом создаются. Одним из способов сделать это может быть ролевая игра с использованием CRC-карточек. Некий член команды будет играть роль подсистемы или актора, и команда будет просматривать прецедент или сценарий. Ниже предлагается образец проигрывания одного из возможных прецедентов.

Джон ("Управление включением", пульт)

Мой домовладелец только что нажал кнопку, управляющую набором ламп. Он все еще удерживает кнопку в нажатом состоянии. Я послал Бобу сообщение, как только кнопка была нажата, и собираюсь посыпать ему сообщения каждую секунду, пока кнопка нажата.

Боб ("Центральный блок управления")

Когда я получил первое сообщение, то изменил состояние выхода с *выкл* на *вкл*. Когда я получил второе сообщение, стало очевидно, что домовладелец хочет изменить яркость освещения, поэтому при получении каждого сообщения я собираюсь изменять яркость на 10%. Между прочим, Джон, не забывай говорить мне, какая кнопка нажата.

Майк (лампа)

Я аппаратно запрограммирован на изменяемый накал. Я изменяю яркость света при нашем разговоре.

Замечание. При выявлении потребностей пользователей посредством CRC-карточек основное внимание уделяется внешнему поведению. Однако данный метод можно также использовать для проектирования объектно-ориентированных систем программного обеспечения. При этом внимание концентрируется на понимании внутреннего функционирования программного обеспечения, а не на взаимодействии с внешней средой. Но даже в этом случае данный метод часто приводит к обнаружению неправильных или невыявленных требований к системе.

Интересно отметить, что игроки неизбежно обнаруживают узкие места и недостатки в сценарии, и в результате их исправления, как правило, совершенствуется понимание системы.

Заключение

Обыгрывание ролей является замечательным методом, хотя он используется не очень часто. Почему? Причин много. Во-первых, существует фактор дискомфорта. Не очень приятно портить простой заказ на покупку, когда за вами наблюдает заказчик или клерк, вводящий заказы. Во-вторых, имеется фактор смущения: столкнувшись с необходимостью общаться с реальными людьми вместо клавиатуры, мы оказываемся не в своей тарелке; в конце концов, мы посещали занятия по теории компиляторов, когда наши ровесники участвовали в драмкружках!

Однако нет сомнений, стоит сделать небольшое усилие, и обыгрывание ролей станет для вас одним из наиболее полезных и недорогих методов, призванных помочь в выявлении требований.

Глава 15

Создание прототипов

Основные положения

- Прототипирование особенно эффективно для преодоления синдромов “да, но...” и “неоткрытые руины”.
- Прототип программных требований является частичной реализацией программной системы и помогает разработчикам, пользователям и заказчикам лучше понять требования к системе.
- Следует создавать прототип “неясных” требований, т.е. тех, которые хотя и известны, но плохо определены и малопонятны.

Программные прототипы, как ранние воплощения программной системы, демонстрируют часть функциональных возможностей новой системы. Принимая во внимание то, что мы уже обсудили, можно предположить, что прототипы можно с успехом применять для выявления потребностей пользователей. Пользователи могут трогать, ощущать прототип системы и взаимодействовать с ним, что не может обеспечить ни один другой метод. Прототипирование исключительно эффективно в преодолении как синдрома “да, но...” (“Это не совсем то, что я думал”), так и синдрома “неоткрытых руин” (“Теперь, после того как я ее увидел, мне бы хотелось добавить еще одно требование”).

Виды прототипов

Существует много способов разбиения прототипов на категории. Например, Дэвис (Davis, 1995,а) в своей работе выделял следующие прототипы: отбрасываемые, эволюционирующие и операционные; вертикальные и горизонтальные; пользовательские интерфейсы и алгоритмические; и т.д. Каким должен быть прототип в каждом конкретном случае, зависит от того, какую проблему вы пытаетесь решить путем построения прототипа.

Например, если риск проекта связан преимущественно с применением технологического подхода — просто это никогда прежде не делалось таким способом и вы не уверены, сможет ли применяемая технология обеспечить нужную производительность или пропускную способность — можно создать архитектурный прототип, который главным образом демонстрирует осуществимость используемой технологии. Архитектурный прототип может быть *отбрасываемым* или *эволюционирующим*. “*Отбрасываемый*” означает, что его задача состоит только в определении осуществимости; можно использовать всевозможные сокращения, альтернативные технологии, имитации для ее достижения. По окончании прототип просто отбрасывается, сохраняется только полученное в результате зна-

ние. “Эволюционирующий” означает, что прототип реализуется в той же архитектуре, которую вы намерены использовать в конечной версии системы, и можно строить рабочую версию системы, развивая данный прототип.

Если же основной риск в проекте представляет интерфейс пользователя, можно разработать прототип требований с помощью любых технологий, позволяющих создать прототип пользовательского интерфейса как можно быстрее. На рис. 15.1 представлено дерево решений, которое можно использовать при выборе вида прототипа, наиболее подходящего для вашего проекта.

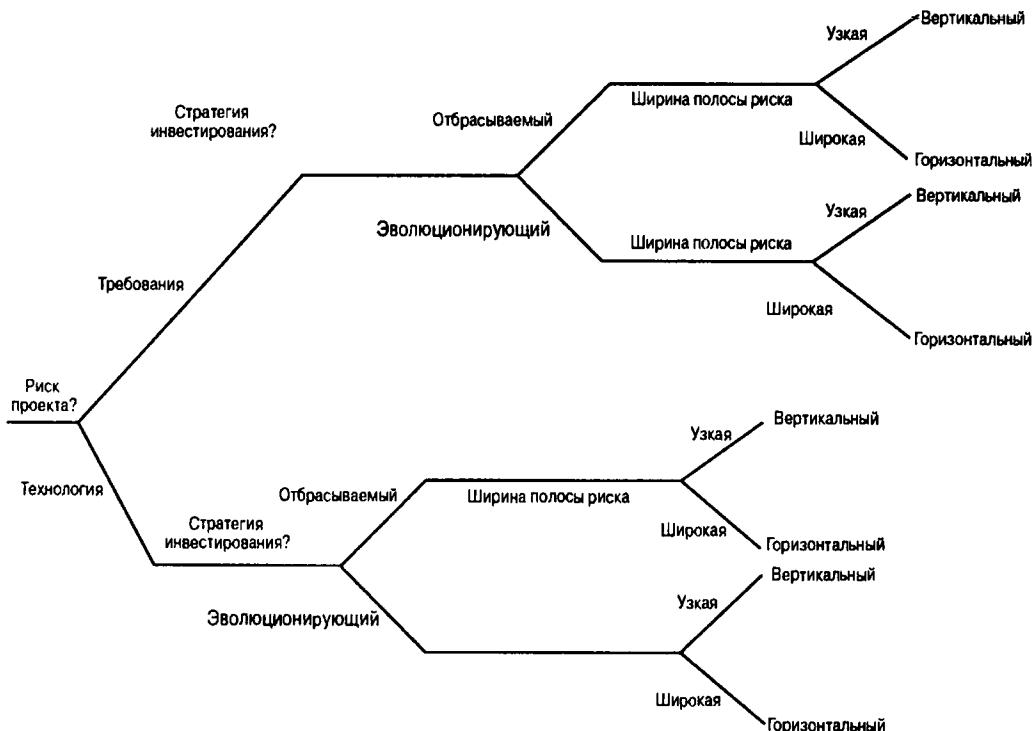


Рис. 15.1. Дерево решений для выбора вида прототипа: верхняя ветвь – прототипы требований; нижняя – архитектурные прототипы

Прототипы требований

Для выявления требований мы сосредоточим наше внимание на видах прототипов, расположенных на верхней ветви данного дерева. Для начала рассмотрим определение.

Прототип требований к программному обеспечению – это частичная реализация системы программного обеспечения, созданная с целью помочь разработчикам, пользователям и клиентам лучше понять требования к системе.

Для выявления требований зачастую можно использовать прототип в виде “отбрасываемого горизонтального интерфейса пользователя”. “Горизонтальный” озна-

чает, что мы будем стараться создать широкий спектр функциональных возможностей системы; в противоположность этому “вертикальный” прототип создает довольно мало требований, но делает это качественно. “Интерфейс пользователя” означает, что мы будем создавать в основном интерфейс системы с ее пользователями, а не заниматься реализацией логики и алгоритмов, лежащих внутри программы, или созданием интерфейсов с другими устройствами или системами. В качестве средства выявления требований, прототип выполняет свою роль несколькими способами.

- Созданный разработчиком прототип может использоваться для получения подтверждения заказчика, что разработчик правильно понимает требования.
- Созданный разработчиком прототип может быть своего рода катализатором, заставляющим заказчика подумать о дополнительных требованиях.
- Созданный заказчиком прототип может помочь донести требования до разработчика.

Во всех трех случаях цель состоит в создании прототипа с наименьшими затратами. Если его создание обойдется слишком дорого, может оказаться более эффективным сразу создавать реальную систему!

Многие прототипы программного обеспечения являются прототипами требований и используются в основном для фиксации элементов интерфейса пользователя создаваемой системы. Тому есть две причины.

1. Появление множества недорогих и широко доступных средств быстрого построения интерфейсов пользователя.
2. Для систем, активно взаимодействующих с пользователем, создание прототипа интерфейса пользователя выявляет также много других требований, а именно, какие функции представляются пользователю, когда каждая функция доступна ему и какие функции для него пропущены.

Однако необходимо следить за тем, чтобы доступность этих средств не заставила нас прототипировать те части системы, которые на самом деле не имеют самого высокого риска и не нуждаются в первоочередном прототипировании.

Что прототипировать

Как же узнать, какую часть системы нужно прототипировать? В типичной ситуации наше понимание потребностей пользователей находится в диапазоне от хорошо понятых и легко объяснимых до совершенно неизвестных (рис. 15.2).

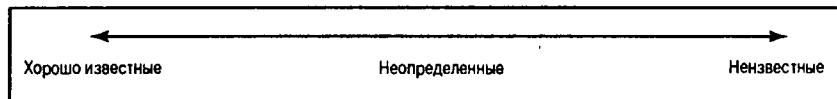


Рис. 15.2. Наше понимание потребностей пользователей

Хорошо понимаемые требования могут очевидным образом вытекать из содержания приложения и опыта пользователей и команды, полученного при работе с системами подобного типа. Например, если мы просто совершенствуем существующую систему, понятно, каким должно быть большинство новых функциональных возможностей. Хорошо

известные и понимаемые требования не нуждаются в прототипировании, если они не являются необходимыми для того, чтобы помочь пользователям визуализировать содержание других их потребностей. Создание прототипов таких требований будет излишней тратой ресурсов, а поскольку они и так уже хорошо понятны, из их рассмотрения мало что можно будет почерпнуть.

Неизвестные требования являются теми “неоткрытыми руинами”, которые мы хотели бы найти. К сожалению, мы не можем по-настоящему прототипировать их, так как в противном случае они уже не были бы неизвестными! Таким образом в качестве объекта прототипирования остаются расположенные в середине “неясные” требования. Эти требования могут быть известны, но плохо определены и плохо понимаемы.

Построение прототипа

Выбор технологии, используемой для построения прототипа, зависит от принятия дальнейших решений в правой части дерева решений на рис. 15.1. Например, для отображаемого GUI-прототипа можно выбрать любой метод (при условии, что он является самым быстрым, самым недорогим способом реализации образцов GUI).

Если выбран эволюционирующий прототип, нужно применить тот же язык реализации и ту же среду разработки, которые будут использоваться в промышленном изделии. Придется затратить значительные усилия на проектирование архитектуры системы, а также применить все стандарты кодирования и программные процессы, которые вы намереваетесь использовать при создании системы. В противном случае вы будете пытаться развивать систему, которая имеет существенные недостатки в одном или нескольких из этих аспектов. В конце концов, может оказаться, что прототип придется отбросить! Или, что еще хуже, созданный из лучших побуждений прототип требований отрицательно влияет на качество развертываемой системы.

Оценка результатов

После создания прототипа пользователи должны поработать с ним в среде, которая как можно более точно имитирует производственную среду использования результирующей системы. Тогда проявятся факторы среды и другие внешние факторы, оказывающие влияние на требования к изделию. Кроме того, важно, чтобы с продуктом поработали пользователи, представляющие различные их классы, в противном случае результаты могут оказаться однобокими.

Результат процесса прототипирования будет следующим.

1. Неясные требования становятся более понятными.
2. Практическая работа с прототипом неизбежно вскроет реакцию пользователя типа “да, но...”; поэтому станут очевидны неизвестные ранее потребности. Даже то, что пользователь просто увидит множество вариантов поведения, поможет ему понять другие требования, которые должны быть наложены на систему.

В любом случае прототипирование практически *всегда* приносит свои плоды. Поэтому, как правило, следует прототипировать любое новое или инновационное приложение. Главное, чтобы полученные в результате знания о требованиях стоили произведенных затрат. Поэтому мы всегда стараемся использовать при создании прототипов (или, по

крайней мере, при реализации наиболее ранних прототипов) самые быстрые и недорогие методы. Ограничивая расходы, мы максимизируем отдачу от инвестиций в получение знаний о требованиях.

Заключение

Поскольку прототипы программного обеспечения демонстрируют часть желаемых функциональных возможностей новой системы, их можно с успехом применять для уточнения реальных требований к системе. Секрет их успеха в том, что пользователи могут взаимодействовать с прототипом в своей среде, которая близка к реальной настолько, насколько этого можно добиться, не разрабатывая промышленную версию программы.

Выбирать метод создания прототипа следует в зависимости от типа риска, присутствующего в вашей системе. Недорогие и простые в разработке прототипы требований помогут вам исключить большую часть связанных с требованиями рисков в проекте.

Среди всех имеющихся возможностей следует выбрать ту, которая позволит затратить на прототип как можно меньше средств. Использование любого из существующих методов прототипирования, а еще лучше – комбинации нескольких методов, доказало свою исключительную эффективность в совершенствовании понимания командой проекта реальных требований к программной системе.

Заключение части 2

Задача понимания реальных потребностей пользователей и других заинтересованных лиц осложняется тремя “синдромами”. Описание синдромов “да, но...”, “неоткрытые руины” и “пользователь и разработчик” помогает лучше понять предстоящие проблемы и дает представление о том, в какой среде будут использоваться методы выявления, которые мы разработали для понимания потребностей пользователей.

Так как команды редко получают совершенные спецификации требований к создаваемой системе, они должны сами добывать информацию, необходимую им для успешной работы. Термин “выявление требований” очень точно отражает данный процесс, в котором команда должна играть более активную роль.

Чтобы помочь команде решить эти проблемы и лучше понять потребности пользователей и других заинтересованных лиц, можно использовать различные методы.

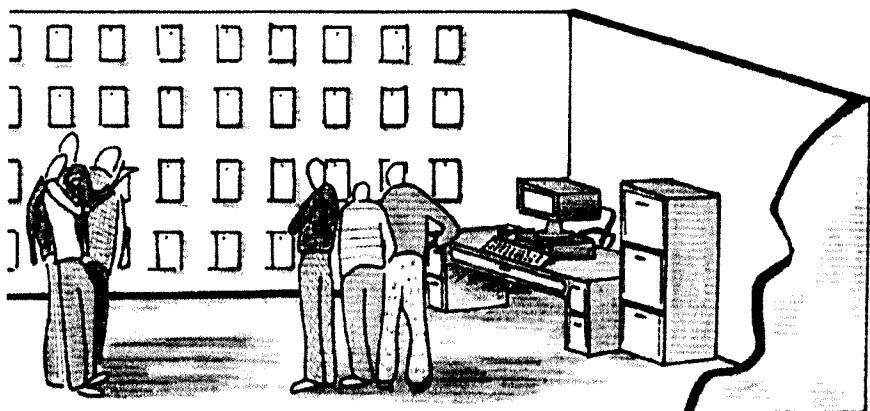
- Интервьюирование и анкетирование
- Совещания, посвященные требованиям
- Мозговой штурм и отбор идей
- Создание раскадровок
- Прецеденты
- Обыгрывание ролей
- Создание прототипов

Хотя ни один метод не является универсальным, каждый из них позволяет лучше понять потребности пользователей и тем самым превратить “неясные” требования в требования, которые “лучше известны”. Каждый из этих методов “срабатывает” в определенных ситуациях, однако мы отдаляем предпочтение совещанию, посвященному требованиям, и мозговому штурму.

Часть 3

Определение системы

- Глава 16. Организация информации о требованиях
- Глава 17. Документ-концепция
- Глава 18. Лидер продукта



В части 1 рассматривались методы анализа проблем. Они позволяют достичь понимания проблемы до того, как будут затрачены значительные усилия на ее решение. В этой части мы занимались исключительно областью проблемы.

В части 2 был описан ряд методов, которые команда может использовать для понимания потребностей пользователей. Эти потребности находятся на вершине пирамиды требований и таким образом представляют наиболее важную информацию, которая является основой всех последующих действий.

В части 3 мы переходим из области проблемы к области решения и концентрируем свои усилия на *определении системы*, которую можно создать для удовлетворения потребностей пользователей и заинтересованных лиц. Чем ниже мы спускаемся по пирамиде (рис. 1), тем больше объем информации. Например, для выполнения одной потребности пользователя может понадобиться значительное число системных функций. Для дальнейшего определения поведения системы необходима дополнительная конкретизация; таким образом, объем необходимой информации увеличивается.

Объем информации, которой мы должны управлять, увеличивается по мере спуска к основанию пирамиды.

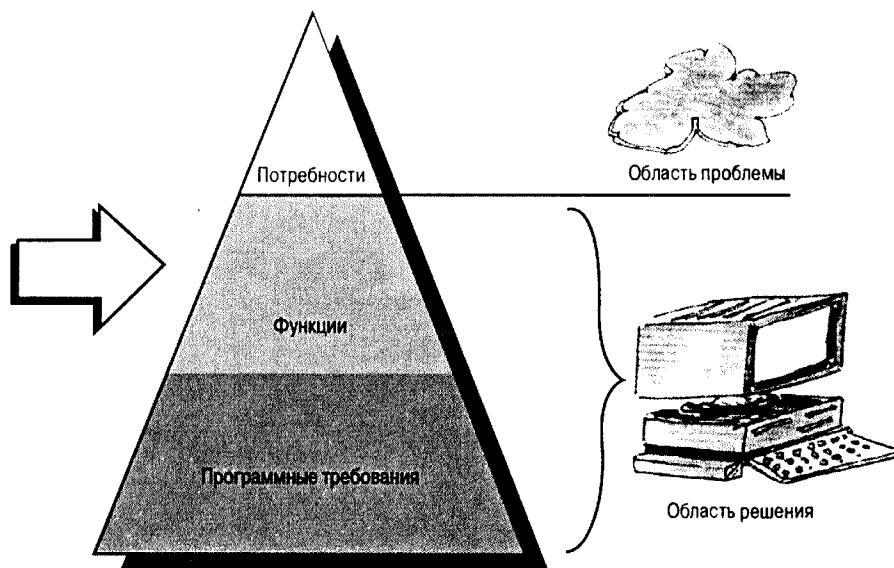
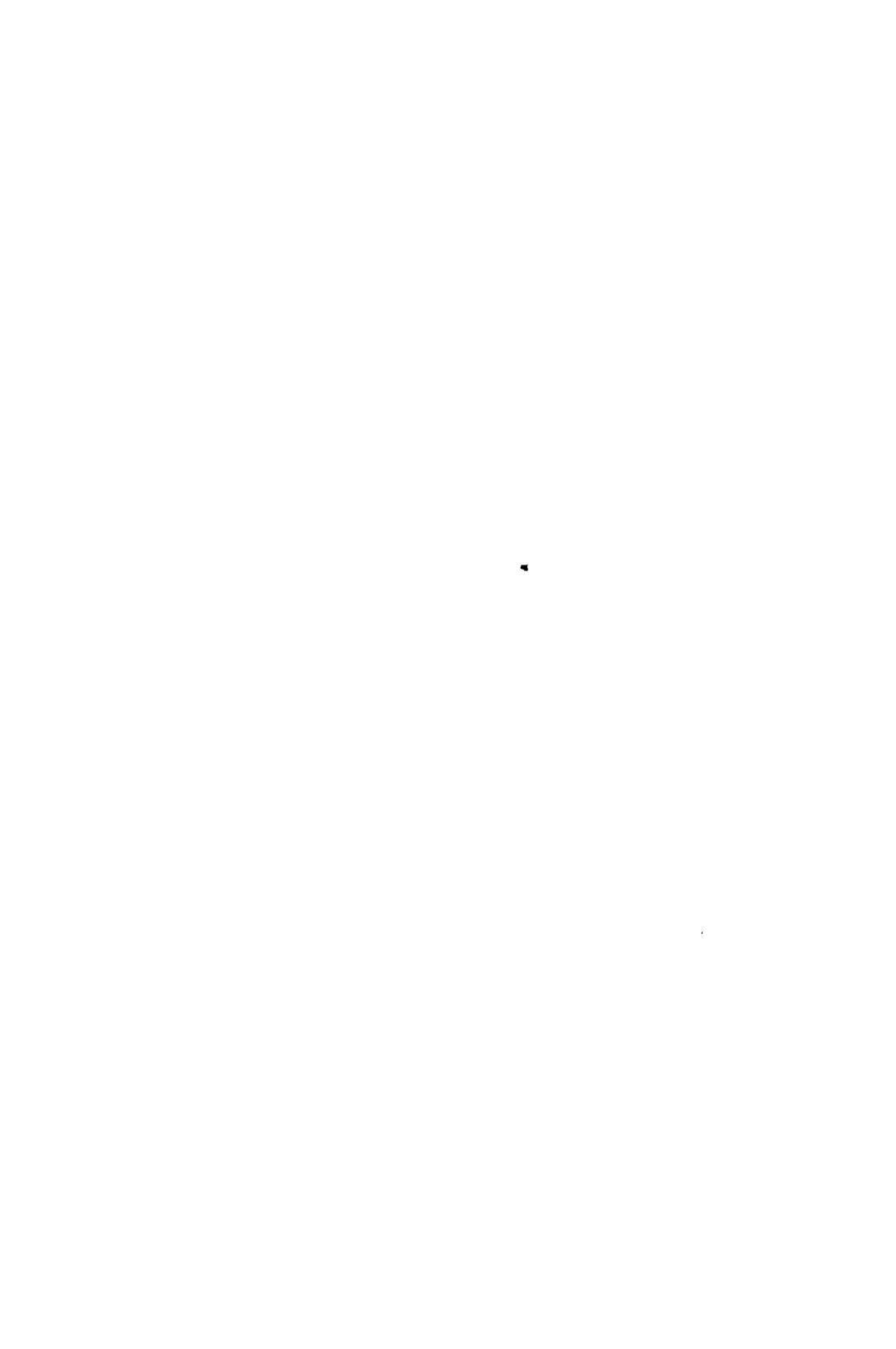


Рис. 1. Функции в пирамиде требований

Кроме того, на данном этапе команде необходимо обратить внимание на множество других вопросов, присущих области решения, но мало связанных с областью проблемы. Например, при разработке программного продукта для продажи приходится заниматься вопросами упаковки, инсталляции и лицензирования, каждый из которых может быть уникальным для предлагаемого нами решения. Если же разрабатываемая система предназначена для удовлетворения некой внутренней потребности в IS/IT, может понадобиться рассмотреть требования к развертыванию и поддержке, которые практически не имеют отношения к пользователю, не использующему такую систему в настоящее время.

Однако нам по-прежнему следует оставаться на достаточно высоком уровне абстракции, потому что если мы слишком рано перейдем к подробному рассмотрению, то будем не в состоянии “за деревьями увидеть лес”. Кроме того, нужно на секунду остановиться и воспользоваться этим временем, чтобы организовать информацию о требованиях, прежде чем перейти к части пирамиды, содержащей программные требования (она будет рассматриваться в части 5, “Уточнение определения системы”). На данном этапе мы займемся организацией информации о требованиях (глава 16), определением концепции (глава 17) и организацией нашей команды для решения задачи управления требованиями к системе (глава 18).



Глава 16

Организация информации о требованиях

Основные положения

- Требования к нетривиальным приложениям следует фиксировать и сохранять в виде базы данных, модели или в форме, которая обеспечивается специальной вспомогательной программой.
- В зависимости от типа приложения необходимо применять различные методы организации требований.
- В сложных системах создаются спецификации требований для каждой подсистемы.

Требования необходимо записывать и документировать. Если вы в одиночку разрабатываете систему, в которой будете единственным пользователем и ответственным за сопровождение, можно попытаться приступить к ее проектированию и кодированию непосредственно после выявления своих потребностей. Однако разработка систем редко бывает столь простой. Как правило, разработчики не являются пользователями, а в процесс разработки вовлечены заинтересованные лица, пользователи, разработчики, аналитики, специалисты по архитектуре и другие члены команды. Все участники должны прийти к соглашению о том, какая система должна быть создана.

Реалии бюджета и графика таковы, что в конкретной версии вряд ли возможно удовлетворить все потребности пользователей. В любой деятельности, в которой участвует много людей, неизбежно возникают проблемы взаимодействия. Поэтому необходимо создать документ, с которым могут сверяться и на который могут ссылаться все участники.

Такой документ называется спецификацией требований. *Спецификация требований* к системе (или приложению) описывает ее внешнее поведение.

Замечание. Следуя исторической традиции, мы будем использовать в данном разделе термин *документ*, но требования могут храниться в виде документа, базы данных, модели прецедентов, архива требований или комбинации этих элементов. Как будет показано далее, для хранения данной информации можно использовать “пакет требований к программному обеспечению”.

Существует ряд причин, из-за которых требования редко удается определить с помощью единого однородного документа.

- Система может быть очень сложной.
- Потребности заказчиков могут быть документированы до того, как производится документирование подробных требований.
- Система может быть членом семейства родственных продуктов.
- Создаваемая система может удовлетворять только некоторое подмножество всех выявленных требований.
- Цели маркетинга и бизнеса следует отделить от подробных требований к продукту.

В любой из этих ситуаций приходится вести множество документов. Рассмотрим несколько специальных случаев.

- Некий “родительский” документ определяет требования ко всей системе в целом: к аппаратному и программному обеспечению, персоналу и процедурам; а другой документ – только к программному обеспечению. Первый документ называется *спецификацией требований к системе* (*system requirement specification*), а второй – *спецификацией требований к программному обеспечению* (*software requirement specification, SRS*).
- Один документ содержит общие определения функций системы, а другой – более конкретные. Первый документ часто называется *документом-концепцией* (*Vision document*), а второй – *спецификацией требований к программному обеспечению* (*software requirement specification*).
- Один документ содержит полный набор требований к семейству продуктов, а в другом представлены требования к конкретному приложению и конкретной версии. Первый из них именуется документом *требований к семейству продуктов* (*product family requirements*), или *концепцией семейства продуктов* (*product family Vision document*), а второй – *спецификацией требований к программному обеспечению* (*software requirements specification*) определенной реализации некоего конкретного приложения из семейства продуктов.
- Один документ описывает общие бизнес-требования и бизнес-окружение, в котором будет существовать система, а другой определяет ее внешнее поведение. Первый называется *документом бизнес-требований* (*business requirements document*), или *документом требований маркетинга* (*marketing requirements document*), а второй – *спецификацией требований к программному обеспечению* (*software requirement specification*).

В следующих разделах описывается, что делать в каждом случае. Эти случаи можно комбинировать; например, некий документ может наряду с бизнес-требованиями содержать полный набор требований, из которого отбираются подмножества и используются для конкретных версий.

Организация требований к сложным аппаратным и программным системам

Хотя данная книга посвящена, в основном, требованиям к программному обеспечению, важно понимать, что они являются только подмножеством процесса управления требованиями при разработке большинства систем. Как уже описывалось в главе 6, некоторые системы настолько сложны, что единственным разумным способом их визуализации и создания является представление в виде систем, состоящих из подсистем, которые

в свою очередь также являются системами подсистем, и т.д., как показано на рис. 16.1. В экстремальном случае, когда система представляет собой, например, авиапоезд, она может состоять из сотен подсистем, каждая из которых в свою очередь имеет компоненты аппаратного и программного обеспечения.



Рис. 16.1. Система, состоящая из подсистем

В таких случаях создается спецификация требований системного уровня, которая описывает внешнее поведение системы в целом (без учета ее разбиения на подсистемы), такое как запас топлива или скорость набора высоты. Как уже отмечалось в главе 6, после согласования требований к системе в рамках системной инженерии производится разбиение системы на подсистемы, подробно описываются интерфейсы между ними, и каждое требование системного уровня размещается в одной или нескольких подсистемах. Полученная в результате системная архитектура описывает это разбиение и интерфейсы между полученными подсистемами.

Процесс проектирования системы сам по себе создает новые классы требований.

Затем разрабатываются спецификации требований для каждой из подсистем. Эти спецификации должны полностью описывать внешнее поведение подсистем (без учета их разбиения на подсистемы следующего уровня). Данный процесс приводит к возникновению нового класса требований – производных требований. Требования этого нового типа описывают уже внешнее поведение не *системы* (разве что в агрегированном виде), а *новой подсистемы*. Таким образом, процесс проектирования системы создает новые требования для подсистем, из которых состоит система. В частности, интерфейсы между этими подсистемами становятся ключевыми требованиями: они представляют собой, по сути, контракты между подсистемами или *обещания функционировать так, как установлено*.

После того как эти требования согласованы, можно при необходимости продолжить проектирование системы путем разбиения каждой из подсистем на подсистемы следующего уровня и разработки спецификаций требований для каждой из них. В результате получается иерархия спецификаций, представленная на рис. 16.2.

На каждом уровне требования предыдущего уровня размещаются в соответствующих спецификациях более низкого уровня. (Например, требование, касающееся запаса топлива, размещается в подсистемах управления топливом и хранения топлива.) Также выявляются и соответствующим образом описываются новые требования.

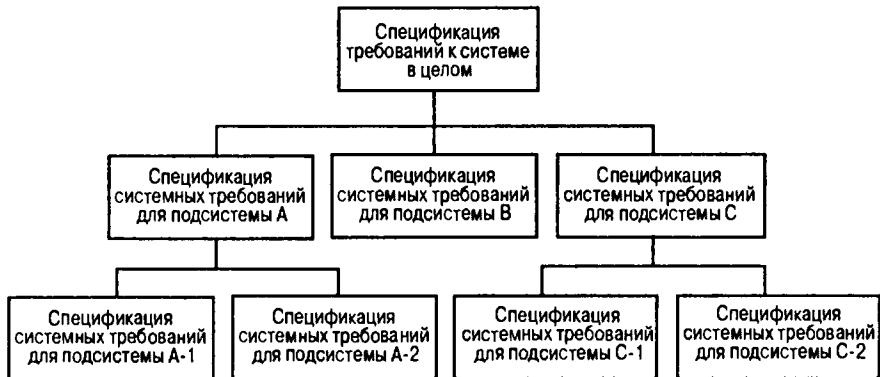


Рис. 16.2. Иерархия спецификаций, полученная в результате проектирования системы

Как показано на рис. 16.3, спецификации, которые в свою очередь уточняются дополнительными спецификациями подсистем, называются *спецификациями системных требований* (или спецификациями требований системного уровня). Спецификации самого нижнего уровня – те, которые не подвергаются дальнейшей декомпозиции, – как правило, соответствуют только аппаратным или программным подсистемам и называются *спецификациями требований к аппаратному или программному обеспечению* соответственно. По мере того как становятся более понятны детали, каждая из приведенных на рис. 16.3 спецификаций может подвергаться эволюционным процессам.

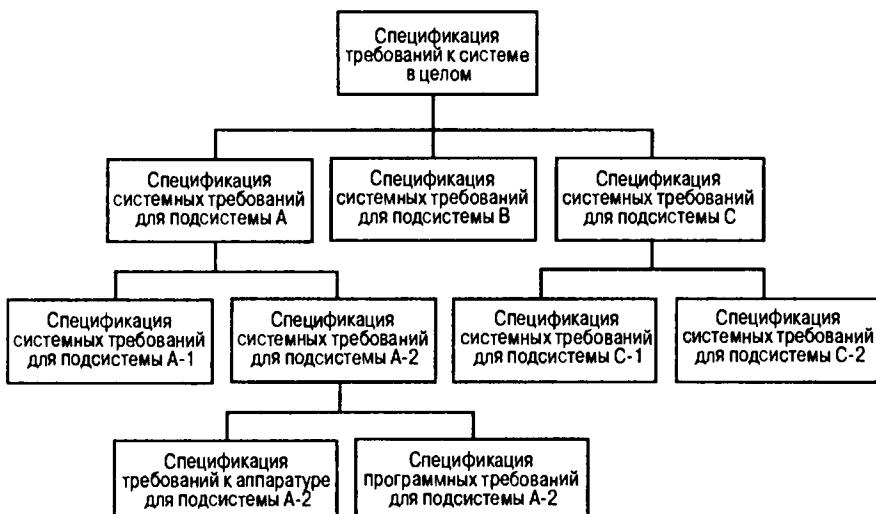


Рис. 16.3. Иерархия результатирующих спецификаций, в том числе аппаратного и программного уровней

Организация требований к семействам продуктов

Во многих отраслях разрабатываются семейства продуктов, имеющих много общих функциональных возможностей. Но каждый из них обладает некоторыми уникальными свойствами. Примерами таких семейств могут служить системы управления складами, автоответчики, противоугонные системы и т.д.

При создании набора программных продуктов, которые имеют некие общие функциональные возможности и потребность в совместном использовании данных или в обмене информацией друг с другом в процессе работы, можно попытаться использовать следующий подход.

- Разработать *документ-концепцию* семейства продуктов, который описывает способы совместной работы продуктов и их общие совместно используемые функции.
- Для более полного понимания модели совместного использования можно также разработать набор *прецедентов*, показывающих, как пользователи будут взаимодействовать с различными совместно выполняемыми приложениями.
- Разработать *спецификацию требований* к общему программному обеспечению, определяющую конкретные требования к общим функциональным возможностям, таким как структуры меню и коммуникационные протоколы.
- Для каждого продукта семейства разработать *документ-концепцию*, *спецификацию требований* к программному обеспечению и модель *прецедентов*, определяющие его конкретные функциональные возможности.

Полученная в результате организация требований представлена на рис. 16.4.

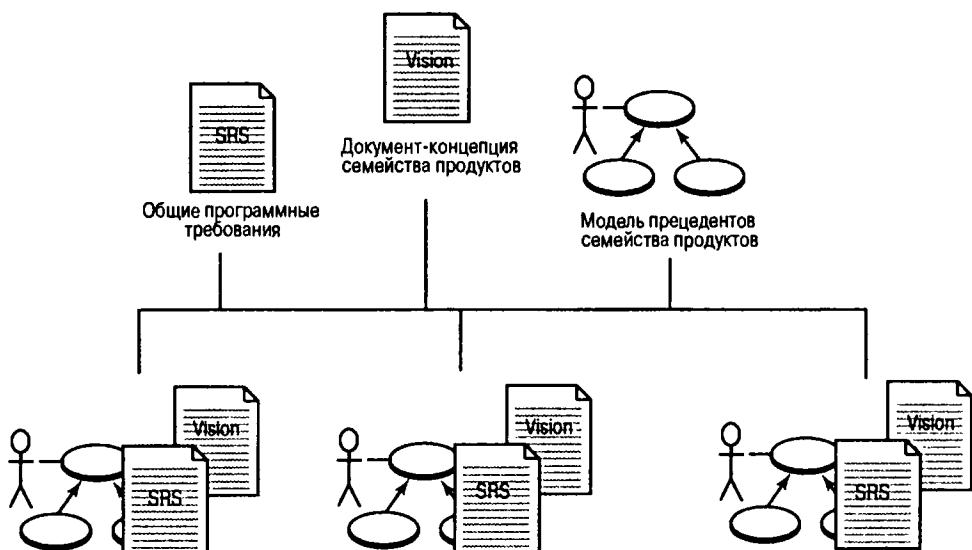


Рис. 16.4. Организация требований для семейства программных продуктов

Спецификации требований для отдельных членов могут содержать ссылки (*связи или трансифровки*) на документы семейства или могут воспроизводить все требования данного документа. Преимущество использования ссылок состоит в том, что изменения в требованиях, относящиеся ко всем членам семейства, можно вносить в одном месте. Но для управления этими зависимостями понадобится некое автоматическое средство управления требованиями; в противном случае каждый раз при изменении родительского документа вам придется самостоятельно анализировать все документы требований отдельных членов семейства.

“Будущие” требования

При разработке мы крайне редко имеем дело с постоянным набором требований или можем построить систему, удовлетворяющую всем известным требованиям. В любом процессе выявления будут возникать требования, которые окажутся неприемлемыми для разрабатываемой в данный момент версии создаваемого продукта.

Было бы неправильно включать такие требования в спецификации требований (не должно быть никаких сомнений относительно того, какие требования должны быть реализованы, а какие – нет). С другой стороны, неправильно было бы отбрасывать их, так как они представляют собой полезные рабочие продукты; мы сможем использовать их при создании последующих версий. Еще более важно то, что проектировщики могут иначе разработать проект системы, зная, что в будущем возможны требования определенного типа. Лучше всего записывать оба типа требований в документ, но *необходимо четко выделять те из них, которые планируется реализовать в текущей версии*.

Оличие бизнес-требований и требований маркетинга от требований к продукту

Планирование нового продукта не производится в чисто технической среде без учета бизнес-соображений. Необходимо подумать о целевых рынках, оформлении продукта, каналах распределения, функциональных возможностях, затратах на маркетинг, доступности ресурсов, прибыли, возможности возмещения затрат путем продажи большого количества копий и т.д.

Эти соображения следует задокументировать; но они не являются частью спецификаций требований. Некоторые организации используют *документ требований маркетинга* (*marketing requirement document, MRD*), чтобы упростить общение между руководством, службой маркетинга и разработчиками и помочь в принятии разумных бизнес-решений, в том числе самого важного решения “делать, не делать”. MRD также предоставляет заказчикам и разработчикам возможность очень ранней верификации взаимодействия, что способствует пониманию продукта в его наиболее общем виде и определению общего масштаба продукта. MRD отвечает на следующие вопросы.

- Кто является заказчиком?
- Кто является пользователем?
- На каких рынках предполагается продавать продукт?
- Как поделены эти рынки?
- Различаются ли требования пользователей в различных сегментах рынка?

- Какие классы пользователей существуют?
- Какие потребности удовлетворяет продукт?
- Что это за продукт?
- В чем заключаются основные преимущества продукта; почему его будут покупать?
- Что собой представляют конкуренты?
- Что отличает продукт от конкурирующих продуктов?
- В какой среде будет использоваться система?
- Какими будут затраты на разработку?
- По какой цене предполагается продавать продукт?
- Как будет осуществляться инсталляция, дистрибуция и сопровождение продукта?

Рабочий пример

В главе 6 мы провели некие действия по разбиению системы HOLIS (системы автоматического домашнего освещения) на подсистемы. К настоящему моменту мы не так уж много знаем о HOLIS, но все же этого, вероятно, достаточно, чтобы предпринять первую попытку организовать имеющуюся информацию о требованияниях. На рис. 16.5 показано, что команда при описании требований к системе HOLIS использует следующие элементы.

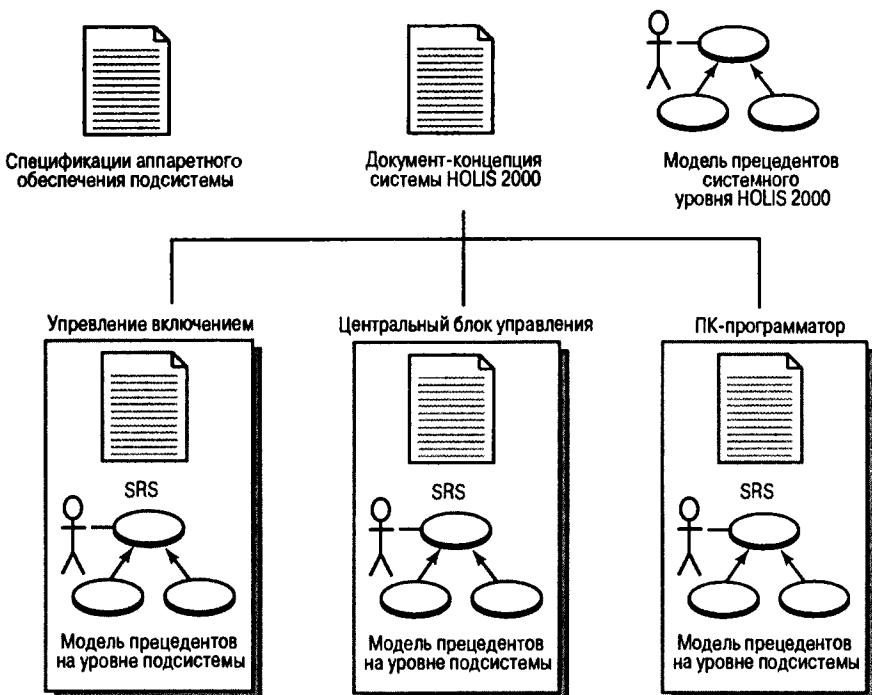


Рис. 16.5. Организация информации о требованияниях к системе HOLIS

- *Документ-концепция*, который содержит краткосрочные и долгосрочные концепции HOLIS, в том числе основные требования системного уровня и предлагаемые функции.
- *Модель прецедентов системного уровня*, которая содержит прецеденты, с помощью которых различные акторы системы взаимодействуют с HOLIS.
- После некоторых дебатов команда приняла решение документировать требования к аппаратным компонентам (размер, вес, мощность, упаковка) всех трех подсистем системы HOLIS в единой спецификации требований к аппаратному обеспечению.
- Так как в каждой из подсистем HOLIS достаточно много программного обеспечения, команда решила разработать для каждой из трех подсистем отдельную спецификацию требований к программному обеспечению, а также модель прецедентов, отражающую, как каждая подсистема взаимодействует с различными акторами.

Дальнейшую разработку данных артефактов требований можно будет наблюдать при продолжении изучения рабочего примера в последующих главах. Их образцы приводятся в приложении А.

Заключение

В этой главе мы рассмотрели разнообразные документы требований для систем разной сложности. Но в большинстве случаев разработка требований производится для отдельной подсистемы программного обеспечения, упакованного программного продукта или отдельного приложения. Это может быть программный продукт, разработанный независимым производителем программного обеспечения, такой как Microsoft Excel, Rational ClearCase, или система управления освещением HOLIS.

В следующих нескольких главах мы подробно рассмотрим процесс определения требований для отдельного программного приложения и таким образом продемонстрируем, как не практике осуществляется процесс управления требованиями.

Глава 17

Документ-концепция

Основные положения

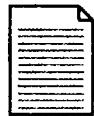
- Документ-концепция (*Vision document*) описывает приложение в целом, включая описания целевых рынков, пользователей системы и функций приложения.
- Документ-концепция определяет на наивысшем уровне абстракции как проблему, так и решение.
- Практически все программные проекты выигрывают от наличия документа-концепции.
- Документ *Delta Vision* акцентирует внимание на том, что изменилось.

Данная глава посвящена рассмотрению документа-концепции. Наш коллега Филипп Крачтен (Philippe Kruchten) как-то сказал: “Если бы мне разрешили разработать только один документ, модель или другой артефакт для поддержки программного проекта, я бы выбрал краткий, хорошо сформулированный документ-концепцию”.

Документ-концепция сочетает в себе некоторые основные элементы документа маркетинговых требований и документа требований к продукту. Нам необходимо разработать этот конкретный документ по двум причинам.

1. Каждому проекту нужен документ-концепция.
2. Это поможет нам в демонстрации процесса работы с требованиями, поскольку некоторые ключевые элементы данного процесса будут записаны в этом документе.

Документ-концепция описывает приложение в общих чертах, а также содержит описания целевых рынков, пользователей системы и функций приложения. Мы неоднократно имели возможность убедиться в его полезности, и у нас стало уже хорошей традицией разрабатывать его при определении любого программного приложения.



Документ-концепция моего проекта

Компоненты документа-концепции

Документ-концепция – это важнейший документ программного проекта, который фиксирует потребности пользователя, функции системы и другие общие требования к проекту. Сфера действия документа-концепции распространяется на два верхних уровня пирамиды требований. Таким образом, он описывает на высоком уровне абстракции как проблему, так и решение.



Документ-концепция программного продукта также служит основой для достижения согласия между тремя основными внутренними сообществами заинтересованных лиц проекта.

1. Отделом маркетинга, который выступает в качестве доверенного лица заказчика и пользователя и отвечает за успех продукта после реализации.
2. Командой проекта, разрабатывающей приложение.
3. Руководством, которое несет ответственность за бизнес-результат попытки.

Документ-концепция является мощным средством, так как представляет все существенные аспекты продукта с различных точек зрения в краткой, абстрактной, доступной и управляемой форме. Документ-концепция крайне важен на ранних фазах проекта, и все затраты, вложенные в процесс получения информации, принесут щедрые плоды на более поздних стадиях.

Поскольку все программные проекты выигрывают от наличия документа-концепции, мы собираемся описать его более подробно. Хотя наш пример ориентирован на программный продукт, его достаточно легко модифицировать, чтобы он отражал содержание любого конкретного продукта.

На рис. 17.1 представлена краткая схема документа-концепции, которая (с некоторыми настройками) использовалась для сотен программных продуктов и приложений. Полная версия данного документа приводится в приложении Б.

1. Введение

В данном разделе предлагается общий обзор документа-концепции.

1.1. Назначение документа-концепции

В данном документе фиксируются, анализируются и задаются высоковзвешенные потребности пользователей и функции продукта.

1.2. Краткое описание продукта

Формулируется цель приложения, версии и новые предоставляемые функции.

1.3. Ссылки

Приводится полный список всех документов, упоминаемых в документе-концепции.

2. Описание пользователя

Кратко представлены общие сведения о пользователях системы.

2.1. Данные о пользователе/рынке

Кратко представлены основные данные о рынке, которые мотивируют решения относительно продукта.

2.2. Типы пользователей

Кратко описываются будущие пользователи системы.

2.3. Среда пользователя

2.4. Основные потребности пользователей

Перечисляются основные проблемы или потребности пользователей.

2.5. Альтернативы и конкуренты

Выявляются все приемлемые (с точки зрения пользователя) альтернативы.

3. Краткое описание продукта

3.1. Общий вид продукта

Предлагается блок-схема продукта или системы и ее интерфейсов с внешней средой.

3.2. Определение позиции продукта на рынке

Предлагается обобщенная краткая характеристика (на самом высоком уровне абстракции) уникальной позиции, которую должен занять продукт на рынке. Мур (Moore) (1991) рекомендует использовать следующую форму.

Для	[целевой клиент]
Который	[формулировка потребности или возможности]
[Название продукта]	является [категория продукта]
Который	[формулировка основных преимуществ, т.е. указание причин, по которым продукт будет покупаться]
В отличие от	[основные конкурирующие альтернативы]
Наш продукт	[формулировка основных отличий]

3.3. Характеристика возможностей

Перечисляются основные возможности и функции, которые будут предоставлены продуктом.

Возможности клиентов	Поддерживающие функции
Возможность 1	Функция
Возможность 2	Функция
Возможность 3	Функция

3.4. Предположения и зависимости

3.5. Затраты и цены

4. Атрибуты функций

Описывается атрибуты функций, которые будут использоваться для оценки, отслеживания, задания приоритетов и управления функциями. Некоторые из них перечислены ниже.

Статус	Предлагаемый, принятый, включенный
Приоритет	Число голосов по результатам накопительного

	голосования или критический, важный, полезный
Трудоемкость	Низкая, средняя, высокая; командо-недели; человеко-месяцы
Риск	Низкий, средний, высокий
Стабильность	Низкая, средняя, высокая
Целевая версия	Номер версии
Предназначен для	Фамилия
Причина	Текстовое поле

5. Функции продукта

В данном разделе перечисляются функции продукта.

5.1. Функция №1

5.2. Функция №2

6. Ключевые прецеденты

Описываются основные прецеденты, которые важны с точки зрения архитектуры или наиболее полезны для того, чтобы помочь читателю понять, как будет использоваться система.

7. Другие требования к продукту

7.1. Применимые стандарты

Перечисляются все стандарты, которым должен соответствовать продукт.

7.2. Системные требования

Задаются все системные требования, которым должно соответствовать приложение.

7.3. Лицензирование и установка

Описываются все инсталляционные требования, которые оказывают влияние на создание программного кода или вызывают потребность в создании отдельного инсталляционного программного обеспечения.

7.4. Требования к производительности

Этот раздел используется для подробного описания требований к производительности.

8. Требования к документации

Описывается, какую документацию необходимо разработать для успешного развертывания приложения.

8.1. Руководство пользователя

Описывается цель и содержание руководства пользователя.

8.2. Интерактивные подсказки

Требования к интерактивным подсказкам, средствам предупреждения и т.п.

8.3. Руководства по инсталляции, конфигурация и файлы "Read Me"

8.4. Маркировка и упаковка

9. Глоссарий

Рис. 17.1. Схема документа-концепции некоего программного продукта

Итак, документ-концепция кратко описывает все, что вы считаете наиболее важным для продукта или приложения. Он представляет собой достаточно подробное описание на естественном языке, поэтому основным участникам проекта легко с ним работать.

Документ Delta Vision

Разработка документа-концепции и работа с ним, являясь центром приложения действий многих участников (заказчиков, пользователей, представителей руководства проекта и маркетинга), могут играть заметную роль в успехе (или неудаче) программного проекта. Зачастую к разработке и пересмотру этого документа привлекается даже дирекция компании. Ведение документа-концепции является важным профессиональным приемом, который в состоянии значительно повысить общую производительность работы над проектом.

Чтобы это было легче осуществить, нужно сделать документ-концепцию как можно более кратким, сжатым и "по существу". При создании первой версии документа это не так уж сложно, так как практически все пункты в перечне будут новыми для данного проекта или, по крайней мере, должны быть переформулированы с учетом содержания данного приложения.

Однако неэффективно в последующие версии повторно записывать вошедшие в предыдущие версии функции, а также прочую информацию, которая не претерпела изменений (описание пользователей и обслуживаемых рынков). Для решения данной проблемы мы предлагаем использовать *документ изменений концепции (Delta Vision document)*. Однако перед тем, как заняться его разработкой, рассмотрим развитие документа-концепции на протяжении жизненного цикла нового проекта.

Документ-концепция версии 1.0

Для нового продукта или приложения необходимо разработать и исследовать практически все элементы документа-концепции. Если некий элемент не рассматривается, вы просто удаляете его из оглавления документа и ничего не пишете о нем. Обязательными элементами документа-концепции являются следующие (рис. 17.2).

- Общая информация и введение
- Сведения о пользователях системы и описание обслуживаемых рынков, функций, которые предполагается реализовать в версии 1.0
- Прочие требования (регуляторные и требования среды)
- Будущие функции, которые были выявлены, но не вошли в версию 1.0

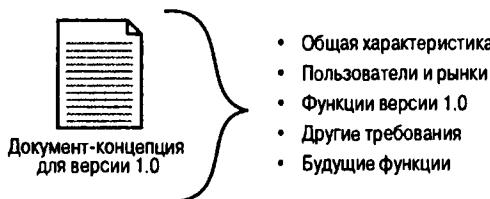


Рис. 17.2. Документ-концепция версии 1.0

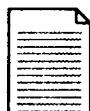
Данный документ служит основой версии 1.0 и разработки более подробных программных требований и прецедентов, которые будут более полно описывать систему.

Документ-концепция версии 2.0

По мере развития проекта более четко определяются функции; это означает, что они будут более полно описаны в документе-концепции. Кроме того, выявляются новые функции и добавляются в него. Таким образом, документ разрастается, и одновременно возрастает его значение для команды. Приступая к разработке версии 2.0, мы, безусловно, хотели бы сохранить документ, который так хорошо нам служил. В данной ситуации представляется логичным "откопать" будущие функции, которые включены в документ для версии 1.0, но не реализованы, и запланировать их для реализации в версии 2.0. Другими словами, мы хотим найти и "раскрутить" некоторые будущие функции, представляющие интерес для версии 2.0. Можно также запланировать проведение дополнительного совещания, посвященного требованиям, или осуществление любого другого процесса выявления требований для обнаружения новых функций, запланированных для реализации в версии 2.0, а также тех, которые нужно будет внести в документ в качестве новых будущих функций. Некоторые из этих функций, основанные на обратной реакции заказчика, уже будут очевидны, другие возникнут как следствие полученного командой опыта. В любом случае эти вновь обнаруженные функции следует записать в документ-концепцию версии 2.0 как запланированные для реализации в версии 2.0 или будущие функции.

Может оказаться, что некоторые реализованные в версии 1.0 функции не достигают поставленной цели (возможно, из-за того, что внешняя среда изменилась за время разработки и функция больше не нужна или должна быть заменена новой, либо из-за того, что она просто не нужна клиентам, хотя они предполагали обратное). В любом случае скорее всего обнаружится, что в следующей версии некоторые функции необходимо удалить. Как отразить эти "антитребования"? В данной ситуации нужно просто использовать документ-концепцию для указания того, что определенная функция должна быть удалена из следующей версии.

В процессе работы документ постоянно растет. Это естественно, так как он определяет растущую систему. К сожалению, может случиться, что со временем документ будет все труднее читать и понимать. Почему? Потому, что он теперь гораздо длиннее и содержит много информации, которая не претерпела изменений со времени предыдущей реализации. Например, определение позиции продукта и целевые пользователи, скорее всего, остались неизменными, как и 25–50 функций, реализованных в версии 1.0, которые сохранились в документе-концепции версии 2.0.

 Документ
Delta Vision v2.0

Поэтому мы предлагаем вести *документ изменений концепции* (*Delta Vision document*). В нем отражается только то, что изменилось, а также любая другая информация, которую следует включить для ясности. Эта информация включается для того, чтобы напомнить команде концепцию проекта или помочь войти в курс дела новым членам команды.

В результате получается документ изменений, в котором *основное внимание уделяется тому, что нового включено в данную версию и что отличает ее от предыдущих версий*. При работе со сложными системами информации полезно применять данный метод, который позволяет *сконцентрировать внимание на том, что изменилось*. Воспользовавшись им, мы получаем модель, представленную на рис. 17.3.

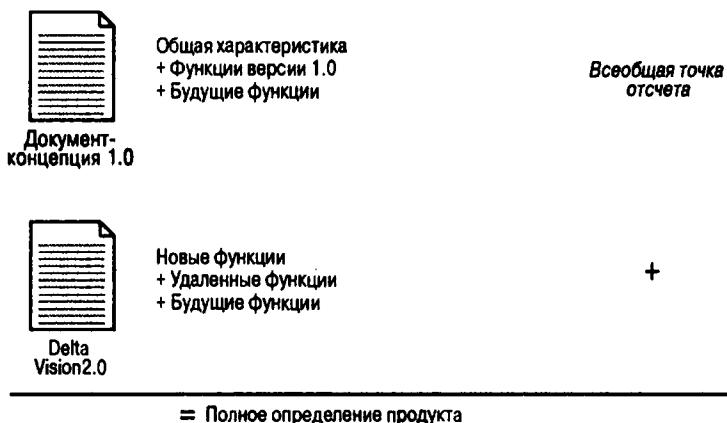


Рис. 17.3. Документ Delta Vision

- Документ-концепция 1.0 является *всебъемлющей точкой отсчета*; здесь представлено все, что необходимо знать о нашем проекте.
- Delta Vision 2.0 определяет то, что отличает данную версию.
- Объединение этих двух документов задает *полное определение продукта*.

Следует использовать обе версии вместе, если согласно требованиям заказчика или регулирующих инструкций необходимо предоставить полное определение продукта. Их совместное использование, несомненно, полезно для новых членов команды. Однако в этом случае приходится читать о функциях версии 1.0, которых уже нет в версии 2.0, так как они были позднее удалены, и нужно всегда отслеживать эти изменения при воссоздании полного определения.

Если это необходимо, можно достаточно просто соединить содержимое документа-концепции 1.0 и Delta 2.0 в новый документ-концепцию 2.0, который представляет *всебъемлющую и полную картину проекта*.

Не существует строгих правил относительно определений этих документов или того, что каждый из них содержит. В других обстоятельствах может оказаться удобным использовать Delta Vision только при относительно небольших модификациях (таких, как версии 1.1 и 1.2) и начинать все сначала и пересматривать определение продукта в целом для каждой крупной реализации (версии 2.0 или 3.0). В любом случае применение документа Delta Vision поможет лучше справиться с процессом управления требованиями, так как позволит команде сконцентрироваться на "том, что действительно важно" на каждом конкретном этапе.

Документ Delta Vision для уже существующей системы

Крайне редко практикуется документирование полных требований крупномасштабной существующей системы.

Одна из сложнейших проблем при управлении требованиями состоит в применении методов управления требованиями к эволюции существующих IS/IT-систем. Крайне редко существуют полные и адекватные спецификации требований для миллионов строк

кода и сотен человеско-лет трудозатрат, отражением которых являются эти системы. Так же непрактично останавливаться и повторно документировать прошлое. При этом можно упустить время и не выполнить свою задачу, записывая исторические требования тогда, когда следовало писать код!

Таким образом, если приходится начинать с нуля или с минимальной документации, следует использовать все имеющиеся в вашем распоряжении ресурсы (программный код, спецификации, знания членов команды о предыстории), чтобы прийти к пониманию того, что система делает в настоящий момент. Затем мы рекомендуем применить процесс создания Delta Vision и задать функции и прецеденты, описывающие *изменения*, которые вы собираетесь вносить в существующую систему. Следуя этому процессу, можно сконцентрироваться на том, что нового в данной реализации и что отличает ее от предыдущих реализаций; в результате ваши заказчики и команда получат несомненную пользу от хорошо организованного процесса управления требованиями. Кроме того, созданные вами записи требований будут служить документацией для ваших последователей.

Глава 18

Лидер продукта

Основные положения

- Лидер продукта отвечает за концепцию проекта.
- Каждому проекту нужен лидер или небольшая лидирующая группа для защиты интересов продукта.
- При разработке программных продуктов лидером часто является представитель маркетинга.

В главе 1 мы проанализировали проекты, в которых возникли затруднения, и выявили множество разнообразных причин этого, причем управление требованиями оказалось в верхней части списка. В главе 17 мы определили документ-концепцию как ключевой документ сложного жизненного цикла программы. Он непосредственно ориентирован на решение проблемы требований и является единственным документом, к которому можно обратиться в любой момент, чтобы увидеть, что продукт, приложение или система должны делать, а что не должны. В целом документ-концепция представляет суть продукта, и его необходимо защищать так, как если бы весь успех проекта зависел от него (потому что *так оно и есть*).

В некоторый момент возникает закономерный вопрос: “Кто же разрабатывает и поддерживает этот исключительно важный документ? Кто управляет ожиданиями заказчика? Кто ведет переговоры с командой разработчиков, заказчиком, отделом маркетинга, менеджером проекта и руководством компании, которое проявило такой интерес к проекту именно теперь, когда подходит срок сдачи?”.

Практически в каждом успешном проекте (от машинок для аттракционов до систем искусственного дыхания, которые спасли десятки тысяч жизней, не допустив ни единого сбоя программного обеспечения), которым мы занимались, был лидер. Мы можем оглянуться на эти проекты и указать некоего человека или (в случае более крупного проекта) небольшую группу из нескольких человек, которые играли роль, “большую, чем жизнь”. Лидеры ставят концепцию продукта (системы или приложения) во главу угла, как будто это самое важное в их жизни. Они думают о ней постоянно, за едой и во время сна.

Роль лидера продукта

Лидером продукта может быть кто угодно: менеджер продукта, менеджер проекта, менеджер по маркетингу, менеджер проектирования, менеджер информационных технологий, руководитель проекта. Однако название должности в данной ситуации не имеет никакого значения; обязанности одинаковы. И они достаточно велики. Лидер должен заниматься следующим.

- Руководить процессом выявления требований и успокаиваться лишь тогда, когда обнаружено достаточное их количество.
- Рассматривать конфликтующие пожелания, поступающие от различных участников.
- Находить компромиссы, необходимые для определения набора функций, представляющих наибольшую ценность для максимального числа участников.
- Владеть концепцией продукта.
- Защищать интересы продукта.
- Вести переговоры с руководством, пользователями и разработчиками.
- Противодействовать "просачиванию" функций.
- Поддерживать "здравое равновесие" между тем, чего хочет заказчик, и тем, что может предоставить команда разработчиков за время, отведенное для реализации.
- Выступать в роли представителя официального канала общения между заказчиком и командой разработчиков.
- Управлять ожиданиями клиентов, исполнительной дирекции и внутренних отделов маркетинга и проектирования.
- Сообщать о реализуемых функциях всем заинтересованным лицам.
- Осуществлять проверку спецификаций программного обеспечения, чтобы удостовериться, что они соответствуют истинной концепции, представленной функциями.
- Осуществлять управление изменением приоритетов, а также добавлением и исключением функций.



Это единственный человек, которому действительно можно доверить судьбу документа-концепции, и выбор "правильного" лидера может оказаться ключом к успеху проекта.

Лидер продукта в среде программных продуктов

Однажды мы проводили семинар для провайдера интерактивных услуг, который столкнулся с проблемами при управлении требованиями. Когда мы затронули вопрос о лидере продукта, в комнате воцарилась тишина. Мы спросили 25 присутствующих, среди которых были разработчики и руководители отделов проектирования и маркетинга, как в их организации принимались эти сложные решения. Выяснилось, что никто не принимал этих решений. То, что описали присутствующие, являлось так называемым "групповым нашупыванием", когда предложения прибывают и убывают как волны прилива. *Никто* не брал на себя ответственность за жесткие решения. *Никто* не решал, когда уже стоит остановиться.

В конце концов, команда оглядывалась на исполнительного директора по маркетингу, ожидая от него ответов, возможно, потому, что этот человек был наиболее активен. Он сказал: *"Вы знаете, что больше всего беспокоит меня в этой команде? Я в любое время могу попросить, чтобы была добавлена некая новая функция, и никто никогда не скажет мне "нет". Как же мы можем надеяться хоть когда-нибудь отгрузить продукт?"*

Из этого заявления можно сделать вывод, что время потрачено, а заказчику все не удается получить готовый продукт. Понятно также, что осознание этого факта для него весьма болезненно. Компания развивалась от традиционных информационных технологий и со временем подошла к предоставлению услуг в интерактивном режиме. Понятие

приложения как *программного продукта* являлось новым. Члены команды не имели опыта, которым могли бы руководствоваться в процессе работы.

Хотя не существует единого рецепта выдвижения лидера продукта, возможно, нам удастся кое-что предложить в результате рассмотрения нашего рабочего примера. В конце концов, мы создавали этот пример, моделируя реальную успешно работающую команду проекта.

На рис. 18.1 роль лидера выполняет Кэти, *менеджер продукта*. Заметим, что в данном случае менеджер продукта отвечает за маркетинг, а не за техническую сторону. Это достаточно типично для компаний, создающих программные продукты, так как, по крайней мере в теории, менеджер продукта ближе всех находится к заказчику, определяющему конечный успех или неудачу проекта. Более важная причина заключается в том, что именно маркетинг в конечном счете определяет "Итог", т.е. доход, связанный с выпуском продукта. Так и должно быть, ведь маркетинг отвечает за продажи и, соответственно, должен отвечать за принятие трудных решений.



Рис. 18.1. Рабочий пример: структура команды, разрабатывающей программное обеспечение

Как же распределить роли и ответственность в среде с множеством требований, технологий, заказчиков и разработчиков? Представим себе следующий диалог, возникший на ранней стадии работы некой новой организации, которая занималась распределением ролей.

Менеджер продукта (лидер):

Продукт должен иметь функции А, В и С, и вы должны создавать его, используя технологию Х.

Менеджер разработки:

Я думаю, он должен иметь функцию D, а не С, и основываться на технологии Y.

Менеджер продукта:

А я говорю, что он должен иметь функции А, В и С. В конце концов, я отвечаю за достижение квоты сбыта и поэтому должен удовлетворить потребности моих клиентов. Если вы хотите отвечать за бизнес-

Менеджер разработки:

результат, вы можете добавить функцию D, если при этом еще сможете вписаться в график.

Менеджер продукта:

Х-м-м (*раздумывая о том, что означает для него помогать команде в достижении квоты*). Я не уверен, что это хорошая мысль. Мы действительно сделаем А, В и С. Но готовы ли вы отвечать за то, будет ли программа работать на самом деле?

Менеджер разработки:

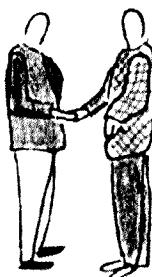
(*представляя себе, что в последующие 48 часов вместо подготовки к выходу на рынок с новым товаром он будет учиться тому, как пишется код*) Ну что же, вы можете создавать программу, используя любую наиболее подходящую технологию.

Менеджер продукта:

Решено. Вы принимаете решения о функциях, а мы выбираем технологию; пусть каждый отвечает за то, в чем лучше разбирается.

Менеджер продукта:

Договорились.



Этот сценарий подчинен обычному здравому смыслу, однако зачастую (как и в случае упоминаемого провайдера интерактивных услуг) подобное распределение ответственности произвести не удастся.

В некотором смысле для независимых производителей программного обеспечения (ISV) ситуация несколько проще. Клиент является внешним по отношению к организации-разработчику. Как правило, существуют достаточно серьезные маркетинговые организации, которые можно привлечь к выявлению требований и определению того, кто будет отвечать за сбалансированность всех конфликтующих потребностей.

Клиент, потребности которого не удовлетворяются, просто *не будет приобретать продукт*. Хотя это, может, и не очень хорошо, но по крайней мере клиенты не стоят над душой со своими вопросами.

Лидер продукта в отделе информационных технологий и систем (IS/IT)

В подразделении предприятия, занимающемся IS/IT, ситуация совершенно иная. Здесь нет отдела маркетинга, все ваши клиенты работают вместе с вами, и они неизбежно будут толпиться вокруг после реализации, чтобы сообщить о своих ощущениях.

Как искать лидера в такой среде? Давайте снова рассмотрим пример. В одном подразделении разрабатывалась новая система, чтобы обеспечить глобальный круглосуточный доступ к записям о клиентах для поддержки продаж и управления лицензированием. В процессе анализа проблемы были выявлены следующие заинтересованные стороны: отдел маркетинга корпорации, отдел удаленных продаж, отдел лицензирования и поддержки, отдел маркетинга предприятия, финансовое руководство предприятия, отдел выполнения заказов. Каждый из этих отделов достаточно активно отстаивал свои потребности, хотя было ясно, что не все потребности удастся удовлетворить. Вопрос: "Кто отвечает за документ-концепцию?" имел подтекст: "Кто захочет сделать весьма опасный для карьеры шаг, ввязавшись в управление данным проектом?".

После анализа ситуации стало ясно, что никто из руководителей команды разработчиков не имеет необходимого авторитета для принятия столь сложных решений, но, тем не менее, лидера нужно было назначить. Команда выбрала Трейси, занимавшую должность руководителя проекта, и уполномочила ее выявлять и организовывать требования. Она владела документом-концепцией. Кроме того, она интервьюировала пользователей, устанавливала их относительные приоритеты и фиксировала данные в функционально-ориентированном формате. Но одновременно был также создан специальный управляющий комитет, или совет по контролю за изменениями проекта (project change control board, CCB). В его состав вошли три представителя руководства, каждый из которых имел свои функциональные обязанности.

Вопрос:

Лидер
+ Совет по контролю за изменениями

Что получится?

Ответ:

Шанс добиться успеха

Сначала Трейси упростила процесс принятия решений, предложив ССВ задать относительные приоритеты в исходной версии. С этого времени ССВ (и только ССВ) имел право добавлять или удалять функции согласно рекомендациям, поступающим от лидера продукта. Таким образом, был только один лидер, Трейси. Результаты выявления функций и концепция проекта находились в ее голове и в документе-концепции, но ответственность за трудные решения была передана ССВ. Именно ССВ отвечал за принятие жестких решений. Лидер “только” следил, чтобы принятые функции были надлежащим образом проработаны и доведены до команды разработчиков.

После того как Трейси была уполномочена вести процесс и был создан принимавший наиболее сложные решения ССВ, в который вошли члены вышестоящего руководства, проект развивался успешно и использовался позднее в качестве организационной модели для новых проектов. В каждом новом проекте был новый лидер. Это создавало возможности для профессионального роста сотрудников. Данная роль стала одной из главных в компании. И конечно, не следует забывать о ССВ. В каждом новом проекте создавался ССВ, в который входили представители различных отделов, в зависимости от содержания каждой новой версии и того, на какие организации она будет оказывать непосредственное влияние.

Хотя не существует общих предписаний, которым можно следовать при выборе лидера проекта, для каждой команды чрезвычайно важно найти его, поддержать или наделить полномочиями того, кто уже фактически стал лидером. После этого задача команды состоит во всесторонней помощи лидеру в управлении требованиями к приложению. Это поможет добиться успеха. Кстати, если вы не помогаете этому человеку, он может предложить вам быть лидером в следующем проекте.

Заключение части 3

В части 3 мы начали переход от понимания потребностей пользователя к определению решения. При этом мы сделали свои первые шаги из области проблемы, вотчины пользователя, в область решения, где нашей задачей является определить систему для решения имеющейся проблемы.

Мы поняли, что для сложных систем требуются всеобъемлющие стратегии управления информацией о требованиях, и рассмотрели несколько способов организации данной информации. Оказалось, что на самом деле существует информационная иерархия, которая начинается с потребностей пользователей, переданных с помощью функций, а затем превращается в более подробные требования к программному обеспечению, выраженные посредством прецедентов или традиционных форм описания. Мы также отметили, что эта иерархия отражает уровень абстракции при рассмотрении области проблемы и области решения.

После этого мы рассмотрели процесс определения отдельного программного приложения и затратили некоторое время на определение документа-концепции для приложения такого типа. Мы считаем, что документ-концепция, модифицированный в соответствии с конкретным содержанием программных приложений компании, является чрезвычайно важным, и его необходимо иметь в *каждом* проекте.

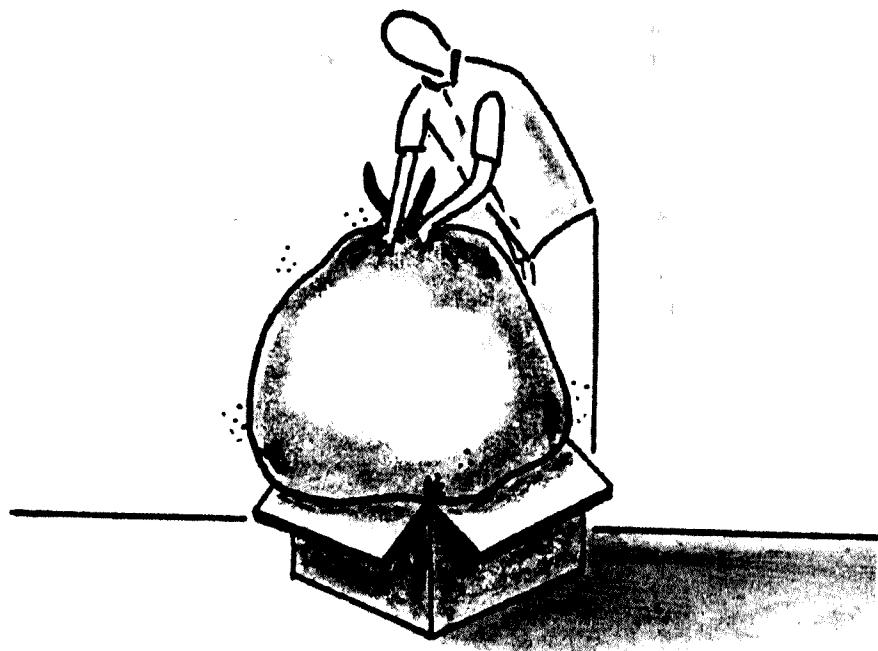
Мы также осознали, что без лидера — человека, который будет защищать требования к приложению и поддерживать потребности клиента и команды разработчиков — нельзя быть уверенными в том, что будут приняты необходимые жесткие решения. Вероятно возникновение дрейфа требований, задержек, а также принятие неоптимальных решений, вызванное приближением срока окончания проекта. Поэтому мы решили назначить лидера, который будет владеть документом-концепцией и содержащимися в нем функциями. В свою очередь, лидер и команда должны создать совет по контролю за изменениями, который призван помогать лидеру в принятии действительно сложных решений и гарантировать, что изменения требований будут приниматься только после их обсуждения.

Вооружившись методом организации требований (с лидером во главе), мы теперь лучше подготовлены к предстоящей работе. Но сначала нам нужно рассмотреть проблему масштаба проекта, которой посвящена часть 4.

Часть 4

Управление масштабом

- Глава 19. Проблема масштаба проекта
- Глава 20. Задание масштаба проекта
- Глава 21. Умение обращаться с заказчиком
- Глава 22. Управление масштабом и модели процесса разработки программного обеспечения





К этому моменту мы уже ознакомились с приемами, используемыми для анализа проблемы, понимания потребностей пользователя и определения системы. Все они направлены на устранение основной причины возникновения проблем при разработке программного обеспечения: при переходе к области решения команда не имеет адекватного представления о решаемой проблеме. Конечно же, членам команды необходимо попрактиковаться в использовании этих приемов, чтобы полностью овладеть ими, однако это не потребует чрезмерных усилий. Мы настоятельно рекомендуем потратить чуть больше времени на это в начале жизненного цикла проекта; затраты на все действия, описанные до сих пор, составят только малую часть выделенных на проект средств (5% или около того). Несмотря на сложность рассматриваемых вопросов, до настоящего момента их решением серьезно занимались только несколько членов команды (аналитики, менеджер проекта, технический руководитель, менеджер продукта/лидер проекта).

Далее, однако, правила игры кардинально меняются, так как команда значительно увеличивается. Все новые ее члены должны участвовать в общих координированных действиях и общаться друг с другом. Кроме того, затраты также существенно возрастают. Мы создаем документы для планов тестирования, строим модели архитектуры, уточняем требования, дорабатываем прецеденты и разрабатываем программный код, тем самым создавая объем работы, которую придется менять, если определение не будет правильно понято или изменятся внешние требования.

Форма пирамиды требований, которая становится гораздо шире у основания, наглядно отражает тот факт, что нам предстоит гораздо больше работы. Часть 4 посвящена разработке стратегии управления масштабом. О важности этой деятельности говорит следующее. Согласно результатам исследований группы Стендиша (1994), "затраты на 54% проектов будут составлять 189% от запланированных". Наш опыт свидетельствует, что практически все программные проекты будут завершены с опозданием; коэффициент запаздывания составит 50–100%. Из этого можно сделать вывод или о недостаточном уровне компетентности в нашей отрасли, или о том, что мы пытаемся сделать слишком много с помощью недостаточных ресурсов, навыков и средств. Мы пытаемся впихнуть десятифунтовые функции в пятифунтовый портфель. Хотя физика процесса разработки программного обеспечения еще не ясна, очевидно, что этот элемент нашей стратегии ведет к неприятностям, и от этого зависит как качество наших продуктов, так и наша репутация.

Поэтому, *прежде чем расширять команду, разрабатывать более подробные спецификации, окончательно формулировать технологические идеи проектирования и создавать сценарии тестирования*, следует остановиться и научиться *управлять масштабом проекта*. Немного психологии, немного технологий и немного просто хорошего управления проектом – и вы овладеете этим приемом, что значительно повысит вероятность успешного завершения проекта.

Глава 19

Проблема масштаба проекта

Основные положения

- Масштаб проекта определяется набором функций продукта, ресурсами проекта и выделенным на него временем.
- Закон Брукса (Brooks) гласит, что привлечение дополнительной рабочей силы к запаздывающему проекту приводит к еще большему запаздыванию.
- Если объем работ, необходимый для реализации функций системы, равен имеющимся ресурсам, умноженным на выделенное время, проект имеет достижимый масштаб.
- Слишком масштабные проекты типичны для отрасли.
- Во многих случаях, чтобы иметь возможность успешно завершить проект, необходимо сократить его масштаб, как минимум, вдвое.

Составляющие масштаба проекта

Как и в любой профессиональной деятельности, приступая к разработке приложения, необходимо сначала произвести реалистическую оценку ресурсов проекта, выделенного времени и поставленных целей. При разработке программного обеспечения эти факторы задают "масштаб" проекта, который определяется следующими переменными.

- Набором функций, которые необходимо предоставить для удовлетворения потребностей пользователей.
- Ресурсами, которыми располагает проект.
- Временем, выделенным на реализацию.

На рис. 19.1 изображен "прямоугольник", который мы можем использовать для представления масштаба проекта.

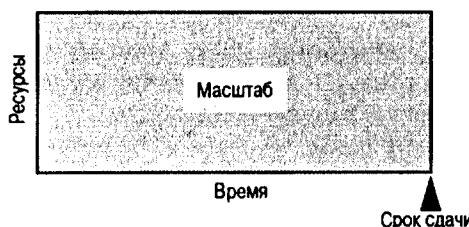


Рис. 19.1. Масштаб проекта

Площадь прямоугольника на рисунке представляет достижимый масштаб проекта. Его определяют следующие элементы.

- *Ресурсы*, состоящие в основном из труда разработчиков, тестологов, составителей руководств, персонала отдела обеспечения качества и др.

Еще в 1970-х годах Фред Брукс (Fred Brooks) (1975) показал, что предложение добавить ресурсы в программный проект с целью увеличить отдачу является, в лучшем случае, рискованным. Закон Брукса гласит, что дополнительное привлечение рабочей силы к запаздывающему программному проекту приведет к еще большему его запаздыванию.

Если временная шкала достаточно протяженная, то результат может действительно возрасти. Но он не возрастет пропорционально добавленным ресурсам; таким образом, общая эффективность проекта снизится. Добавление ресурсов может даже замедлить проект, так как необходимость обучать и поддерживать новых сотрудников снижает производительность тех, кто уже был занят в проекте. По мере того как условия конкуренции на рынке побуждают нас сокращать сроки разработки, добавление ресурсов в ходе работы над проектом практикуется все реже и реже.

Для целей анализа масштаба предположим, что ресурсы, откладываемые по оси у (рис. 19.1), являются постоянными во время выполнения проекта.

- *Время*. Возможно, здесь мы имеем "мягкое" ограничение, которое подвержено изменениям, если наличные ресурсы недостаточны для достижения желаемых функциональных возможностей. Но при анализе масштаба мы будем считать время, откладываемое по оси x, фиксированным фактором.

К сожалению, предоставление программ с опозданием является "нормальным явлением". Но с другой стороны, многие приложения имеют жесткие фиксированные сроки сдачи. Примерами могут быть новая программа начисления налогов, которая должна быть представлена ко времени начала периода налогообложения, демонстрация нового продукта, приуроченная к торговой выставке, или оговоренный контрактом с заказчиком срок сдачи. Кроме того, если мы хотим доказать свою профессиональную состоятельность и завоевать доверие наших клиентов, очень важно не ошибиться в графике.

Набор функциональных возможностей, который мы в состоянии создать, естественным образом ограничен временем (фиксированным), выделенным на реализацию, и доступными ресурсами (также фиксированными). Поэтому достижимый масштаб представляет собой площадь прямоугольника.

В данной книге мы использовали понятие "функции" (features) для обозначения дополнительных функциональных возможностей, которые мы должны предоставить пользователю. Если необходимый для реализации требуемых функций объем работ равен имеющимся ресурсам, умноженным на выделенное время, масштаб проекта является достижимым и проблем не возникает. В этом случае, за исключением непредвиденных обстоятельств, команда создаст программу вовремя и без потери качества.

Однако опыт свидетельствует, что зачастую части в уравнении масштаба отнюдь не равны. На семинарах, посвященных вопросам требований, мы всегда спрашиваем: "Какой масштаб в начале проекта вам обычно задает руководство, заказчики или заинтересованные лица?". В ответ только изредка можно услышать "до 100 процентов". Как

правило, числа в ответах варьируются от 125 до 500 процентов. Среднее значение каждый раз получается приблизительно одинаковым и составляет около 200 процентов. Эти данные в значительной мере совпадают с результатами исследований группы Стендиша, свидетельствующими, что более половины проектов будут стоить примерно вдвое больше, чем предполагалось. Теперь мы, возможно, понимаем, почему.

Небольшая история о масштабе проекта

Однажды одну из наших студенток назначили на новую для нее должность менеджера разработки программного продукта. В прошлом она занималась многими аспектами разработки новых продуктов и маркетинга, но не имела непосредственного опыта в разработке программного обеспечения. Она выслушала ответы на наш вопрос о масштабе с недоверием. Оглядев присутствующих, она сказала: “Неужели вы действительно хотите сказать, что постоянно беретесь сделать вдвое больше работы, чем можно успеть за отведенное время?¹ Что же это за профессия? Вы что, сумасшедшие?”. Разработчики смущенно посмотрели друг на друга и согласно ответили: “Да”.

Что происходит, когда проект развивается с изначально заданным масштабом 200 процентов?

- Если функции приложения полностью независимы (что маловероятно), только половина из них будет работать к моменту сдачи. Продукт будет “хромать”, обеспечивая только половину предполагаемых возможностей. И эта половина не является целостной! Функции не работают совместно, не производят никакой полезной суммарной работы. Приложение с решительно “урезанным” масштабом быстро собирается в единое целое и отправляется. Следствием является серьезное недовольство заказчиков, чьи ожидания не оправдались, невыполнение маркетинговых обязательств, некачественные руководства пользователя и рекламные материалы, которые необходимо срочно переделать. Команда страдает и теряет мотивацию.
- К моменту сдачи каждая функция готова только на 50 процентов. Поскольку, как правило, существуют взаимозависимости между отдельными частями функции, *ничто вообще не работает к окончанию срока*. Срок сдачи проходит. Все обязательства нарушаются; определяется новый срок сдачи и зачастую начинается новый “марш смерти”. В худшем случае вся команда увольняется после нескольких месяцев работы сверх установленного срока; объявляется финальная “фаза” первой попытки разработки проекта и назначается новый менеджер.

Каково качество программного обеспечения в каждом из этих случаев? Код, который создавался ближе к концу, плохо сконструирован и содержит дефекты; тестирование сведено к минимуму или вообще не производится; документация и системы подсказок отсутствуют. Заказчикам приходится выполнять как тестирование, так и действия по обеспечению качества. В итоге заказчики реагируют на наши сверхусилия следующим образом: *Мы сразу были разочарованы тем, насколько вы опоздали (или как мало сделали по сравнению с нашими ожиданиями), но теперь мы по-настоящему недовольны, так как обнаружили, что то, что вы нам предоставили, – это программный мусор.*

¹ Многие студенты отметили, что часто руководство подписывает контракт, не спрашивая их мнения.

Трудный вопрос

Для того чтобы команда проекта имела хоть какую-то надежду на успех, ей необходимо оценивать масштаб как до начала, так и во время проведения разработки. Однако в типичном случае задача состоит в сокращении масштаба: *Если мы действительно начинаем разработку с 200-процентным масштабом, необходимо сократить масштаб проекта, как минимум, в два раза, чтобы иметь шанс на успех.*

Перед командой встает дилемма: *как сократить масштаб и удовлетворить заказчика?* Это один из самых трудных вопросов. Но еще не все потеряно! Мы рассмотрим способы решения данной проблемы в следующих двух главах.

Глава 20

Задание масштаба проекта

Основные положения

- При определении масштаба проекта первым шагом является задание базового уровня требований, т.е. упорядоченного списка высокогорневых функций, которые будут реализовываться в указанной версии продукта.
- Второй шаг состоит в приблизительном определении объема трудозатрат, необходимого для каждой из перечисленных базовых функций.
- Третьим шагом является оценка риска для каждой функции или вероятности того, что ее реализация окажет неблагоприятное воздействие на график и бюджет.
- Используя эти данные, команда задает базовый уровень таким образом, чтобы гарантировать предоставление критически важных для успеха проекта функций.

Базовый уровень требований

Задача управления масштабом состоит в задании высокогорневой базы требований для данного проекта.

Базовый уровень – это разбитое на элементы множество функций или требований, которые намечено реализовать в конкретной версии приложения.

Представленный на рис. 20.1 базовый уровень версии должен быть согласован как с заказчиком, так и с командой разработчиков. Другими словами, базовый уровень должен обладать следующими свойствами.

- Он должен быть, как минимум, “приемлемым” для заказчика.
- Должен иметь разумную вероятность успеха с точки зрения команды разработчиков.

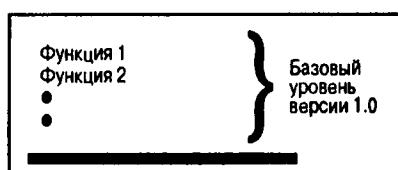


Рис. 20.1. Базовый уровень требований

Первый шаг при задании базового уровня состоит в простом перечислении функций, которые были определены для приложения. Очень важно при этом следить за уровнем детали-

зации. В части 3 мы утверждали, что любую новую систему, независимо от того, насколько сложной она является, можно описать с помощью списка из 25–99 функций. Если их будет больше, то описание проекта будет слишком детализировано, что будет помехой при обсуждении его с заказчиками и командой разработчиков. Если же меньше – уровень детализации будет недостаточным для обеспечения удовлетворительного понимания приложения и оценки уровня необходимых для его реализации трудозатрат.

Если мы следовали рекомендуемому для совещания по вопросу требований процессу (глава 10) или пришли к аналогичному результату другим путем, в нашем распоряжении будет список из 25–99 функций. Этот список представляет собой разбитое на пункты высокоуровневое описание возможностей новой или модифицируемой системы. Он является исключительно важным артефактом проекта, который мы будем использовать для оценки масштаба проекта *перед тем*, как будут затрачены значительные средства на уточнение требований, проектирование, разработку программного кода, тестирование и другие действия.

В качестве примера рассмотрим некий программный продукт, для которого был составлен список из следующих восьми функций.

- Функция 1. Поддержка внешней реляционной базы данных
- Функция 2. Многопользовательская безопасность
- Функция 3. Возможность клонирования проекта
- Функция 4. Порт для новой версии операционной системы (ОС)
- Функция 5. Новый “мастер” проекта
- Функция 6. Импортирование внешних данных по стилям
- Функция 7. Реализация средств предупреждения
- Функция 8. Интеграция с подсистемой управления версиями

Установка приоритетов

Как уже отмечалось в части 2, “Понимание потребностей пользователей”, задание относительных приоритетов функций набора является составной частью управления масштабом. Важно, чтобы заказчики и пользователи, менеджеры нижнего уровня или их представители – не команда разработчиков – определяли приоритеты, находясь в вашем отделе маркетинга. Первичная расстановка приоритетов должна производиться без заметного влияния со стороны технического сообщества; в противном случае уровень сложности реализации функций будет влиять на приоритеты клиентов, результаты процесса могут исказиться и окажется, что приложение не будет удовлетворять реальные потребности клиентов. Технические моменты можно будет учесть на более поздних фазах процесса расстановки приоритетов. Для нашего примера предположим, что мы провели голосование относительно приоритета каждой функции, используя шкалу “критически-важный-полезный”; результаты этого голосования представлены в табл. 20.1.

Таблица 20.1. Упорядоченные по приоритетам функции

Функция	Приоритет
Функция 1. Поддержка внешней реляционной базы данных	Критический
Функция 4. Порт для новой версии ОС	Критический
Функция 6. Импортирование внешних данных по стилям	Критический

Окончание табл. 20.1

Функция	Приоритет
Функция 3. Возможность клонирования проекта	Важный
Функция 2. Многопользовательская безопасность	Важный
Функция 5. Новый “мастер” проекта	Важный
Функция 7. Реализация средств предупреждения	Полезный
Функция 8. Интеграция с подсистемой управления версиями	Полезный

Оценка трудозатрат

Расстановка приоритетов – только часть представляющей масштаб картины. Если бы мы были в состоянии выполнить всю работу, не нужно было бы задавать приоритеты. Если же мы не можем выполнить всю работу, необходимо определить, какую часть мы можем сделать и где, следовательно, нам провести базовый уровень для данного проекта.

Второй шаг заключается в определении приблизительного уровня трудозатрат для каждой из предлагаемых функций. Это достаточно сложно сделать, так как еще мало информации для оценки предстоящей работы; у нас нет подробных требований или результатов проектирования, на которых можно основывать эти оценки. Лучшее, что можно сделать – определить “примерный порядок величины” уровня трудозатрат для каждой функции.

Оценка трудозатрат на столь раннем этапе является сложной задачей. Команда программистов, естественно, неохотно будет проводить оценки до достижения полного понимания осуществимости проекта и требований к нему. Тем не менее, первое сокращение масштаба проекта должно состояться до того, как станет известен этот следующий уровень детализации.

Предположим, что менеджер продукта является лидером нашего проекта и у него происходит следующий диалог с разработчиками проекта¹.

Менеджер продукта:	Насколько трудно сделать эту функцию?
Команда разработчиков:	Мы не знаем. У нас до сих пор еще нет никаких требований или результатов проектирования.
Менеджер продукта:	Я это понимаю, но является ли эта функция самой простой, которую вам доводилось когда-либо создавать?
Команда разработчиков:	Нет.
Менеджер продукта:	Хорошо, является ли она наиболее сложной функцией в списке?
Команда разработчиков:	Нет.
Менеджер продукта:	По шкале “низкая-средняя-высокая” мы присвоим ей

¹ Команда может по своему желанию использовать “командо-месяцы” или “командо-недели” для оценки общего вклада функции в проект. Эта грубая эвристическая оценка служит заменой более детальной оценки, и неизвестно, что лучше, этот метод или результат данного диалога. Затем, учитывая эти оценки и общее отведенное на проект время, команда может определить, где изначально проводить базовый уровень. Если он проходит ниже критических функций, все хорошо; если же нет, проект слишком масштабный и необходимо определять новый (меньший) проект.

Команда разработчиков: среднюю степень сложности. Вы согласны?
Менеджер продукта: Да. Она является средней.
Хорошо. Переходим к следующей функции.

Почему мы не рекомендуем использовать процесс определения подробных требований и информации проектирования для каждой функции, чтобы получить более обоснованные оценки? Разве это не является профессиональным подходом к данной проблеме? Если график позволяет на данном этапе выделить время для более детальной оценки, без всяких сомнений стоит это сделать!

Однако мы считаем, что очень важно уметь делать некоторые быстрые предварительные заключения о масштабе предстоящих действий без проведения более детальных оценок. Почему? Потому что в противном случае ресурсы тратятся на то, что впоследствии будет определено как “ненужные продукты”, среди которых окажутся спецификации требований для нереализуемых функций, информация по проектированию этих функций, сценарии проверки требований, которые будут позднее удалены в процессе сокращения масштаба проекта, неправильное определение критического пути проекта и т.д. Мы не можем позволить себе тратить ресурсы на эти напрасные действия, в противном случае мы не сможем оптимизировать затраты. Другими словами, в процессе управления масштабом будет уменьшено количество разрабатываемых для исходной версии функций, а так как ресурсы чрезвычайно ограничены, мы не можем тратить дополнительные средства на функции, которые не будут реализованы в текущем базовом уровне. В табл. 20.1 представлен список наших функций с добавлением информации о трудозатратах.

Таблица 20.2. Список функций с добавленными оценками трудоемкости

Функция	Приоритет	Трудоемкость
Функция 1. Поддержка внешней реляционной базы данных	Критический	Средняя
Функция 4. Порт для новой версии ОС	Критический	Высокая
Функция 6. Импортирование внешних данных по стилям	Критический	Низкая
Функция 3. Возможность клонирования проекта	Важный	Высокая
Функция 2. Многопользовательская безопасность	Важный	Низкая
Функция 5. Новый “мастер” проекта	Важный	Низкая
Функция 7. Реализация средств предупреждения	Полезный	Низкая
Функция 8. Интеграция с подсистемой управления версиями	Полезный	Высокая

Добавление элемента риска

Третий шаг управления масштабом – это оценка риска, связанного с каждой функцией. В данном случае мы понимаем под риском вероятность того, что разработка функции окажет негативное воздействие на график и бюджет. Риск дает возможность оценить, какими могут быть последствия включения конкретной функции в базовый уровень проекта. Функция с высоким риском может негативно повлиять на проект, даже если все остальные функции будут завершены в установленное время.

Команда разработчиков задает риск с помощью любой эвристики, используя ту же градацию “низкий-средний-высокий”, что и при определении трудоемкости. В табл. 20.3 представлен список функций для нашего примера с указанием рисков.

Таблица 20.3. Список функций с добавленными оценками риска

Функция	Приоритет	Трудоемкость	Риск
Функция 1. Поддержка внешней реляционной базы данных	Критический	Средняя	Низкий
Функция 4. Порт для новой версии ОС	Критический	Высокая	Средний
Функция 6. Импортирование внешних данных по стилям	Критический	Низкая	Высокий
Функция 3. Возможность клонирования проекта	Важный	Высокая	Средний
Функция 2. Многопользовательская безопасность	Важный	Низкая	Высокий
Функция 5. Новый “мастер” проекта	Важный	Низкая	Низкий
Функция 7. Реализация средств предупреждения	Полезный	Низкая	Высокий
Функция 8. Интеграция с подсистемой управления версиями	Полезный	Высокая	Низкий

В различных проектах стратегии уменьшения риска отличаются, и мы не будем здесь обсуждать эту тему. Для управления масштабом нам достаточно просто знать о риске, связанном с каждой функцией, чтобы можно было принимать осмысленные решения на ранней стадии осуществления проекта. Например, если функция является *критической* и имеет *высокий* риск, то нужна эффективная стратегия уменьшения риска. Если функция с *высоким* риском характеризуется как *важная* и находится в списке близко к черте базового уровня, ее можно удалить или разрабатывать в том случае, “если останется время”. Если окончательное решение о включении этого элемента в данную реализацию еще не принято, никакого вреда от подобных действий не будет. Если же функция с *высоким* риском является всего лишь *полезной*, следует рассмотреть вариант ее полного удаления.

Сокращение масштаба

Мы добились значительного прогресса. Теперь у нас есть набор упорядоченных по приоритету функций, для каждой из которых определены ее относительная трудоемкость и риск. Необходимо отметить, что зачастую приоритет мало связан с трудоемкостью или риском. Действительно, многие критические функции требуют невысоких трудозатрат, а некоторые полезные являются очень сложными. Это может помочь команде при определении очередности реализации функций. Например, критическая функция со средней трудоемкостью и низким риском может считаться кандидатом на немедленное финансирование. Наша задача в том, чтобы применить имеющиеся ограниченные ресурсы с наибольшей выгодой для клиента. В табл. 20.4 предлагается несколько основанных на значениях указанных атрибутов рекомендаций относительно очередности разработки критических функций.

Таблица 20.4. Методы определения очередности разработки функций

Атрибуты и их значения	Что делать
<i>Приоритет:</i> критический <i>Трудоемкость:</i> высокая <i>Rиск:</i> высокий	Внимание! Нужно немедленно определить стратегию снижения риска; немедленно финансировать; основное внимание необходимо уделить достижимости посредством архитектуры
<i>Приоритет:</i> критический <i>Трудоемкость:</i> высокая <i>Rиск:</i> низкий	По-видимому, трудоемкий элемент; немедленно финансировать
<i>Приоритет:</i> критический <i>Трудоемкость:</i> низкая <i>Rиск:</i> низкий	Финансировать как безрисковый фактор или отложить на потом

Обоснованная первая оценка

Команда, которая имеет даже грубую (основанную на трудозатратах) оценку, может определить базовый уровень, суммируя оценки трудозатрат, пока сумма не станет равна произведению наличных ресурсов и выделенного времени. Однако часто команда не имеет в своем распоряжении даже этих данных, но вынуждена произвести первое сокращение объема проекта. В этом случае неизвестно, где провести базовый уровень, но если команда чувствует, что объем проекта превышает 100%, список функций, вероятно, придется сократить.

Следующий шаг наиболее сложный. Если предположить, что объем работ, необходимый для разработки существующего набора функций, составляет более 200% от возможного, то базовый уровень должен отсечь половину или больше. Как же осуществить этот процесс?

Первым делом надо выяснить, возможно ли за отведенное время завершить разработку хотя бы только критических элементов списка. Спросите менеджера проекта: “Если все остальное не делать, можем ли мы быть уверены в получении хотя бы *критических* элементов к моменту сдачи?”. В конечном итоге, если мы правильно составили схему определения приоритетов, то только одна треть (или около того) элементов списка будет признана *критической* для данной версии. За исключением случая, когда одна из критических функций имеет непропорционально большую трудоемкость, ответ должен быть “да”, даже если наши функции представляют объем работ, равный 200%. Если ответом является “да” (*а наш опыт показывает, что практически всегда так и есть*), то даже после сокращения у нас останется основа для будущего плана. Если же ответ “нет”, то объем проекта еще больше (300–400% или более) и нужно определить проект меньшего масштаба и повторить процесс расстановки приоритетов.

Так как в данном процессе получаются только грубые оценки, невозможно с уверенностью сказать, сколько элементов, помимо *критических*, удастся осуществить. Для дальнейшего уточнения базового уровня можно использовать оценки, основанные на более детальных требованиях и оценке технической достижимости. (Параллельно можно создать подробный план проекта, чтобы проверить достоверность сделанных нами предположений.)

Как следует из нашего опыта, для многих реальных проектов достаточно провести базовый уровень по *критическим* требованиям, возможно, включив один или два *важных* элемента, и оставить на усмотрение команды разработчиков принятие дальнейших решений о включении *важных* элементов, в зависимости от того, как продвигается проект. Нет, это ненаучно. Но зато это действительно срабатывает!

Если правильно формировать ожидания и обращаться с ними, то все, что удастся осуществить помимо базового уровня, будет бонусом. В табл. 20.5 эта простая эвристика применяется к нашему образцу проекта.

Таблица 20.5. Упорядоченный список функций

Функция	Приоритет	Трудоемкость
Функция 1. Поддержка внешней реляционной базы данных	Критический	Средняя
Функция 4. Порт для новой версии ОС	Критический	Высокая
Функция 6. Импортирование внешних данных по стилям	Критический	Низкая
Функция 3. Возможность клонирования проекта	Важный	Высокая
Базовый уровень (функции выше этой линии являются обязательными)		
Функция 2. Многопользовательская безопасность	Важный	Низкая
Функция 5. Новый “мастер” проекта	Важный	Низкая
Функция 7. Реализация средств предупреждения	Полезный	Низкая
Функция 8. Интеграция с подсистемой управления версиями	Полезный	Высокая

Функции, находящиеся ниже базового уровня, теперь являются *будущими* и будут рассматриваться при реализации следующих версий. Приоритет этих функций в дальнейшем может быть повышен (на основании того, что сделано, и исходя из будущих пожеланий заказчика).

Конечно, функции не всегда независимы. Во многих случаях одна из функций, находящаяся ниже базового уровня, связана с некой функцией, расположенной выше базового уровня, или же ее легче реализовать с помощью другой функции. Может оказаться, что команда повезло и она опережает график или находит библиотеку классов, которая существенно упрощает реализацию функции, находящейся ниже базовой линии. В таких случаях команда должна иметь право включить эту функцию в число базовых и в разрабатываемую версию, переопределить очередность и переустановить базовый уровень, естественно, уведомив всех, кого нужно.

Таким образом, команда сможет создать план проекта, по крайней мере, в первом приближении. Однако, по всей вероятности, многие из желаемых функций не войдут в первый срез, и нужно будет что-то делать с ожиданиями как внутри, так и вне компании. Эта тема рассматривается в следующей главе. Но сначала мы рассмотрим наш рабочий пример и увидим, с чем команда подошла к реализации версии 1.0 HOLIS.

Рабочий пример

После совещания перед командой HOLIS возникла задача оценить уровень трудозатрат для каждой функции и попытаться определить базовый уровень для версии 1.0. Нужно было принять жесткие меры по ограничению масштаба в связи с имеющимися временными ограничениями, среди которых – необходимость представить прототип для показа на торговой выставке в декабре, а также запуск системы в производство в ян-

варе², что еще сильнее лимитировало возможности. С помощью эвристического метода оценки по шкале "высокая-средняя-низкая" команда оценила предполагаемый уровень трудозатрат, а также добавила оценки риска для каждой функции. В табл. 20.6 представлен полный перечень функций с указанными атрибутами.

Таблица 20.6. Упорядоченные по числу набранных голосов функции системы HOLIS 2000 с атрибутами риска и трудоемкости

ID	Функция	Число голосов	Трудоемкость	Риск
23	Возможность произвольного выбора зон освещения	121	Средняя	Низкий
16	Автоматическая установка длительности работы для различных источников света и т.п.	107	Низкая	Низкий
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105	Низкая	Высокий
6	100%-надежность	90	Высокая	Высокий
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88	Высокая	Средний
1	Легко программируемые станции управления	77	Средняя	Средний
5	Возможность программирования режима "жильцы в отпуске"	77	Низкая	Средний
13	Любой источник света может плавно изменять мощность	74	Низкая	Низкий
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73	Высокая	Средний
14	Возможность программирования работы в режиме обслуживания зрелищных мероприятий	66	Низкая	Низкий
20	Функция закрытия гаражных ворот	66	Низкая	Низкий
19	Автоматическое включение света в туалете при открытии двери	55	Низкая	Высокий
3	Интерфейс с системой охраны дома	52	Высокая	Высокий
2	Простота монтажа	50	Средняя	Средний

²Хотя руководство производством указало этот срок, команда приняла решение, что на самом деле она имеет время до конца февраля, чтобы представить окончательную версию программного обеспечения 1.0. Это критически важные 6 недель, которые, по мнению команды, необходимы, чтобы, основываясь на полученных после демонстрации на выставке отзывах, внести окончательные изменения.

Окончание табл. 20.6

ID	Функция	Число голосов	Трудоемкость	Риск
18	Автоматическое включение света, когда кто-то подходит к двери	50	Средняя	Средний
7	Мгновенное включение/выключение света	44	Высокая	Высокий
11	Возможность управлять шторами, жалюзи, насосами и движками	44	Низкая	Низкий
15	Управление освещением и т.п. по телефону	44	Высокая	Высокий
10	Наличие интерфейса с системой управления автоматикой в доме	43	Высокая	Высокий
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34	Средняя	Низкий
26	Наличие центральных станций управления	31	Высокая	Высокий
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25	Средняя	Средний
25	Интернационализированный пользовательский интерфейс	24	Средняя	Высокий
21	Интерфейс с видео- и аудиосистемой	23	Высокая	Высокий
24	Восстановление функций после сбоя в энергоснабжении	23	Нет данных	Нет данных
17	Управление системой кондиционирования воздуха	22	Высокая	Высокий
28	Активация голосом	7	Высокая	Высокий
27	Поддержка презентационного веб-сайта	4	Средняя	Низкий

На следующем этапе команда предложила предварительные оценки каждой функции и разработала подробный план проекта, отражающий определенные зависимости и важнейшие вехи. После переговоров с отделом маркетинга, который в свою очередь проконсультировался с Ракель (международным дистрибутором компании), команда определила, что в версии 1.0 достаточно интернационализировать только ЦБУ-интерфейс пользователя, что заметно уменьшит объем трудозатрат на данную функцию. Программное обеспечение интернационализации интерфейса дополнительного ПК-программатора может подождать до версии 2.0. Это привело к тому, что команда изменила название функции 25 с "интернационализированный пользовательский интерфейс" на "интернационализированный ЦБУ-интерфейс" и добавила в список новую функцию "интернационализированный интерфейс ПК-программатора".

Затем, основываясь на пересмотренных оценках трудоемкости, команда предложила провести базовый уровень так, как показано в табл. 20.7. Предложенный базовый уро-

вень был передан в исполнительную дирекцию, и вице-президент Эмили приняла окончательное решение. Однако перед этим она пожелала ознакомиться с планом проекта, чтобы “увидеть зависимости”. (Команда понимала, что на самом деле она хотела увидеть, “выполнила ли команда свое домашнее задание” или это просто “пустышка”, предназначенная для того, чтобы добиться некоего послабления графика.) В конце концов, ее решение было положительным, но Эмили заметила: “Мы принимаем эти предложения для версии 1.0 системы HOLIS, но вам следует знать, что мой начальник сказал мне, что я не должна допустить, чтобы вы сорвали выпуск продукта в январе, как вы, похоже, решили”. Далее она продолжила: “Я не уверена, но думаю, что этим он хотел сказать, что если мы потерпим неудачу, то отвечать буду я, но я даже не хочу думать об этом. Понятно?”.

С большим вниманием выслушав Эмили, члены команды взяли на себя ответственность за дату выпуска продукта и перешли к следующей фазе. Следующая веха, согласно плану проекта, состояла в тщательно разработанной итерации, включающей в себя быстрое создание прототипа системы HOLIS для демонстрации 1 августа.

Таблица 20.7. Базовый уровень для HOLIS версия 1.0

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
23	Возможность произвольного выбора зон освещения	121	Средняя	Низкий	Максимально возможная гибкость
16	Автоматическая установка длительности работы для различных источников света и т.п.	107	Низкая	Низкий	Максимально возможная гибкость
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105	Низкая	Высокий	Необходимы дальнейшие маркетинговые исследования
6	100%-надежность	90	Высокая	Высокий	Подойти к 100% максимально близко
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88	Высокая	Средний	Обеспечить соответствующий контроллер
1	Легко программируемые станции управления	77	Средняя	Средний	Сделать простыми, насколько этого можно добиться при соответствующей трудоемкости

Продолжение табл. 20.7

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
5	Возможность программирования режима "жильцы в отпуске"	77	Низкая	Средний	
13	Любой источник света может плавно понижать мощность	74	Низкая	Низкий	
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73	Высокая	Средний	В версии 1.0 поддерживается только одна конфигурация
25	Интернационализированный ЦБУ-интерфейс	24	Средняя	Высокий	По настоянию Европейского дистрибутора
14	Возможность программирования работы в режиме обслуживания зрелищных мероприятий	66	Низкая	Низкий	(Не применяется, включена в 23)
7	Мгновенное включение/выключение света	44	Высокая	Высокий	Вложить разумные средства

V 1.0. Обязательный базовый уровень: все, что находится выше данной линии, должно быть реализовано, в противном случае выпуск версии будет задержан.

20	Функция закрытия гаражных ворот	66	Низкая	Низкий	Может незначительно повлиять на программное обеспечение
2	Простота монтажа	50	Средняя	Средний	Базис для трудоемкости
11	Возможность управлять занавесками, жалюзи, насосами и движками	44	Низкая	Низкий	Может незначительно повлиять на программное обеспечение
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34	Средняя	Низкий	Хорошо, если получится сделать это

Окончание табл. 20.7

V 1.0. Дополнительные функции: постарайтесь разработать те из них, которые сможете (Кэти).

Будущие функции: находящиеся ниже этой линии функции для данной версии не разрабатываются.

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
29	Интернационализированный интерфейс ПК-программатора	24	Средняя	Высокий	Обязателен для версии 2.0
3	Интерфейс с системой охраны дома	52	Высокая	Высокий	Можем ли мы предложить хотя бы аппаратный интерфейс? (Эрик)
19	Автоматическое включение света в туалете при открытии двери	55	Низкая	Высокий	
18	Автоматическое включение света, когда кто-то подходит к двери	50	Средняя	Средний	
15	Управление освещением и т.п. по телефону	44	Высокая	Высокий	
10	Наличие интерфейса с системой управления автоматикой в доме	43	Высокая	Высокий	
26	Центральные станции управления	31	Высокая	Высокий	
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25	Средняя	Средний	
25	Пульты дистанционного управления	24	Средняя	Высокий	
21	Интерфейс с видео- и аудиосистемой	23	Высокая	Высокий	
24	Восстановление функций после сбоя в энергоснабжении	23	Нет данных	Нет данных	
17	Управление системой кондиционирования воздуха	22	Высокая	Высокий	
28	Активация голосом	7	Высокая	Высокий	
27	Поддержка презентационного веб-сайта	4	Средняя	Низкий	

Глава 21

Умение обращаться с заказчиком

Основные положения

- Необходимо привлекать заказчиков к управлению *собственными* требованиями и масштабом *их* проекта.
- Заказчики, вовлеченные в процесс, будут владеть результатом.
- Выполнить задачу как следует — означает предоставить к указанному времени достаточные функциональные возможности для удовлетворения реальной потребности клиента.
- Приемы ведения переговоров являются неоценимым подспорьем при решении проблем управления масштабом.

Привлечение заказчиков к управлению масштабом *их* проекта

Сокращение масштаба проекта до размеров, сопоставимых с имеющимся временем и ресурсами, может привести к враждебным отношениям между командой проекта и ее заказчиками, потребности которых необходимо удовлетворить. Если быть честными, со всеми такое случалось. К счастью, это не обязательно должно быть так. *Мы можем активно привлекать наших заказчиков к управлению их требованиями и масштабом их проекта, чтобы обеспечить как качество, так и своевременность разработки программного обеспечения.*

Это заключение основывается на нескольких важных моментах.

- Именно заказчики несут финансовую ответственность за выполнение внешних обязательств перед клиентами. Поэтому лучшее, что может предложить команда — это приложение высокого качества, выполненное в срок и в пределах бюджета (пусть и в несколько сокращением, в случае необходимости, масштабе).
- Приложение, его важнейшие функции, удовлетворяемые им бизнес-потребности — все это принадлежит заказчикам, а *не* команде разработчиков. Мы нуждаемся в указаниях заказчиков при принятии основных решений, и только они могут реально определить, как, сократив масштаб, получить полезное приложение. Мы занимаемся только технологическими аспектами. Это *их* проект.

Сообщение о результате

Если масштаб проекта необходимо сократить, убедитесь, что заказчик является непосредственным участником процесса. Заказчик, участвующий в процессе, будет владеть результатом. Исключенный из процесса заказчик будет недоволен и, естественно, будет стремиться обвинить разработчиков в недостаточной старательности.

Привлечение заказчика помогает наименее болезненно решить проблемы управления масштабом. Следуя философии, описанной в предыдущей главе, разумные заказчики пообещают своим клиентам только критические функции, включенные в базовый уровень. Таким образом удастся избежать неприятностей с отставанием от графика и пропущенными функциями. Любые дополнительные функции (которые удастся разработать помимо базовых) будут восприняты позитивно как превзошедшие ожидания.

Иногда необходимость сокращения масштаба обнаруживается в отсутствие заказчика; тогда, очевидно, необходимо сообщить ему об этом. Доставка подобного сообщения нашим заказчикам и/или руководству — деликатный процесс, требующий как умения вести переговоры, так и принятия определенных обязательств по полученным в результате сокращения графику и масштабу. После такого сообщения мы не можем позволить себе потерпеть неудачу в предоставлении нового обещанного продукта, иначе все доверие будет потеряно.

Переговоры с заказчиком

Почти во всех бизнес-процессах требуется вести переговоры. Например, переговоры относительно даты поставки шариковых подшипников, о цене на крупный заказ, переговоры о вашем ежегодном повышении зарплаты с вашим менеджером, переговоры о достижимой квоте с вашим отделом продаж или переговоры о дополнительных ресурсах для вашего проекта.

Для защиты как вашего проекта, так и бизнес-целей вашего заказчика вам может понадобиться вести переговоры об объеме работ для вашей команды. Команде следует знать, что зачастую заказчик владеет приемами ведения переговоров и, естественно, будет использовать их в дискуссии с командой. Следовательно, руководителю команды, менеджеру или лидеру проекта также необходимо *владеТЬ соответствующими приемами*. Ведение переговоров представляет собой вид профессиональной деятельности в деловой сфере. Это не особенно сложный процесс, и его можно осуществлять честно, красиво и стильно. Воспользуйтесь возможностью попрактиковаться в этом; возможно, вам сможет помочь ваш отдел, занимающийся человеческим фактором, или вы можете принять участие во вчерашнем семинаре. Если вам это не удастся, следует хотя бы ознакомиться с некоторыми правилами игры. Например, хороший обзор процесса ведения переговоров содержится в книге Фишера (Fisher), Ури (Ury) и Паттона (Patton) *Getting to Yes* (1983), которую можно прочитать буквально за несколько часов. Они дают несколько полезных советов относительно проведения любых переговоров.

- Начинайте высоко, но не слишком.
- Отделяйте человека от проблемы.
- Концентрируйте внимание на интересах, а не позициях.
- Поймите, с чем вы уйдете.

- Придумайте взаимовыгодные варианты.
- Примените объективный критерий.

Руководящий принцип при управлении масштабом: “меньше обещать и больше делать”.

Во время переговоров с заказчиком о задании базового уровня следует руководствоваться принципом *меньше обещать и больше делать*. Тогда неизбежные издержки разработки программного обеспечения (непредвиденные технологические риски, изменения требований, задержки при приобретении закупаемых компонентов, непредвиденный уход основных членов команды и т.п.) не приведут к нарушению графика вашего проекта. Если вам посчастливится работать с тем одним проектом из тысячи, в котором эти неприятные обстоятельства не возникнут, — прекрасно! В худшем случае вы придете в замешательство, закончив проект раньше! Это даже развеселит вашу компанию!

Управление базовым уровнем

Преуспевающие менеджеры задают маржи для ошибок при оценке трудоемкости и разрешают иногда включать узаконенные изменения в требования во время разработки. Эти менеджеры также препятствуют “просачиванию” функций, которое, по оценкам Джерри Вайнберга (Jerry Weinberg, 1995), может увеличить объем проекта на 50–100% после начала работы. Концентрируя усилия на разработке критических для заказчиков функций, можно даже смягчить политически враждебное окружение. Если масштаб согласован на достижимом уровне и разработка концентрируется практически полностью на том, что, по мнению клиента, “должно быть”, команда завоевывает доверие, соблюдая график и критерии качества, и, быть может, предоставляя возможности, которые не были обещаны заранее.

Однако ваши заказчики, внешние или внутренние, естественно желают получить как можно больше функциональных возможностей в каждой версии программной системы. В конце концов, именно эти функциональные возможности создают добавленную стоимость, которая нужна им для достижения их бизнес-целей. Мы должны с пониманием относиться к требовательным клиентам, поскольку именно они в конечном счете добиваются успеха на рынке. Только таких требовательных, компетентных клиентов и стоит иметь!

Однако если пойти на поводу требований все большей и большей функциональности, это может негативно сказаться на качестве и общей жизнеспособности проекта. *Большее становится врагом достаточного*. Лучшее — враг хорошего!

Если бы мы работали в той сфере деятельности, где физика процесса лучше определена, а отрасль имеет несколько столетий опыта в поставке надежных товаров, ситуация была бы иной. Но мы имеем дело с миром программирования; физика не определена, процесс еще не устоялся, а технологии меняются с каждым новым приложением. Первым делом остановимся на том, как добиться удовлетворительного выполнения задания: *в установленное время предоставить достаточно функциональных возможностей для удовлетворения реальной потребности клиента*. Позднее мы можем соответствующим образом настроить наш процесс, чтобы увидеть, можем ли мы превысить ожидания клиента, но на данном этапе давайте сконцентрируем внимание только на их *достижении*! Для этого необходимо научиться управлять базовым уровнем.

После того как базовый уровень задан, он представляет собой центр, вокруг которого концентрируется множество разнообразных видов деятельности. Функции базового уровня могут использоваться для того, чтобы реалистически оценить прогресс в развитии проекта. Исходя из достигнутого по отношению к базовому уровню, можно манипулировать ресурсами. Базовые функции можно разрабатывать более детально, тем самым подготавливая их к разработке кода. Можно применять трассировку требований от потребностей пользователя к функциям базового уровня. Трассировку можно затем продолжить от функций к дополнительным спецификациям и реализации.

Возможно, наиболее важным является то, что высокоуровневый базовый уровень можно использовать для успешного управления изменениями. Изменения являются неотъемлемой частью любой разработки. Управление изменениями настолько важно, что мы посвятили этой теме отдельную главу 34. На данном этапе мы рассмотрим только то, как для этих целей можно использовать базовый уровень функций.

Официальное изменение

Базовый уровень функций обеспечивает удобный механизм управления высокоуровневыми изменениями. Рассмотрим официальное изменение, когда заказчик запрашивает новую возможность системы, которая не является частью базового уровня. Прежде чем включить эту новую функцию в базовый уровень, следует оценить воздействие этого изменения. Если команда проекта изначально тщательно определила базовый уровень, то следует исходить из предположения, что любое изменение в базовом уровне повлияет на *ресурсы, график или набор функций*, которые должны быть представлены в данной версии.

Если ресурсы фиксированы и график изменить нельзя, команда проекта должна привлечь заказчика к процессу принятия решения о приоритете новой функции по отношению к другим функциям, определенным для реализации в данной версии. Если новая функция является критической, она, по определению, должна быть включена в реализацию. Заказчик совместно с командой проекта должен определить, какие функции следует исключить из реализации или, по крайней мере, как понизить их приоритеты с соответствующим уменьшением ожиданий. Но если функция является важной, а не критической, команда может исходить из того, что в процессе разработки будет ясно, можно ли данную функцию реализовать в этой версии.

Неофициальное изменение

Парадоксально, но изменение по инициативе заказчика может оказаться одной из самых простых проблем управления масштабом, с которыми приходится сталкиваться. Она ставится извне; мы можем предпринять некие меры предосторожности; воздействие изменения можно оценить и объяснить заказчику.

Однако опыт свидетельствует, что существует еще один тип изменений, который является более опасным для процесса разработки. В главе 34 мы будем обсуждать опасность неявных изменений и опишем дополнительные методы решения проблемы управления масштабом.

Глава 22

Управление масштабом и модели процесса разработки программного обеспечения

Основные положения

- Процесс разработки определяет *кто, что, когда и как* делает.
- В модели водопада деятельность по разработке программного обеспечения осуществляется с помощью последовательности шагов, причем каждый шаг основывается на результатах деятельности на предыдущем шаге.
- Спиральная модель начинается с создания набора основанных на рисках прототипов, после чего выполняется структурированный процесс, аналогичный модели водопада.
- Итеративный подход сочетает в себе свойства модели водопада и спиральной модели, а фазы жизненного цикла в нем отделяются от производимых в пределах каждой фазы действий по разработке.
- Независимо от используемой модели, необходимо разработать, как минимум, один ранний прототип, чтобы выяснить реакцию клиента.

До сих пор мы не обсуждали подробно процесс программной разработки в целом; в частности, не рассматривали, как сам процесс разработки влияет на способность команды достичь желаемых результатов. Однако понятно, что эффективное управление требованиями не может существовать без хорошо организованного процесса разработки, который полностью определяет множество действий, выполняемых командой при работе над программным продуктом. Некоторые процессы разработки программного обеспечения достаточно формальны, другие – неформальны, но процесс существует всегда, даже если он строго не определен и не документирован.

Процесс разработки программного обеспечения *определяет, кто (какой член команды), что (jakie действия), когда (данные действия по отношению к другим действиям) и как (детали и этапы этих действий) делает для достижения цели*. Процессы разработки программного обеспечения оказывают заметное воздействие на способность команды разработать программу в срок и в пределах бюджета. В данной главе мы рассмотрим некоторые высокогородневые аспекты различных процессов разработки программного обеспечения, в том числе временные фазы и основные типы деятельности во время этих фаз, а затем проанализируем, как сказывается на задаче управления масштабом проекта то, какой модели процесса следует команда.

“Модель водопада”

Бом (Boehm) (1988, а) отмечал, что еще в начале 1950-х, когда в программной отрасли осознали стоимость обнаружения программных дефектов на поздних этапах жизненного цикла, была принята логическая пошаговая модель процесса — от фазы разработки требований к фазе проектирования, кодирования и т.д. Это было значительным усовершенствованием по сравнению с существовавшей ранее двухфазной моделью “кодирования и исправления”, согласно которой программисты сначала писали код, а затем поправляли его до тех пор, пока уже нечего было поправлять.

В 1970-х Уинстон Ройс (Winston Royce), работавший в компании TRW, предложил модель разработки программного обеспечения, известную как “модель водопада”. В ней содержались следующие усовершенствования строго пошаговой модели.

- Появились петли обратной связи между стадиями; это отражает тот факт, что проектирование воздействует на разработку требований, а написание кода системы будет вызывать повторные обращения к проектированию и т. д.
- Параллельно с анализом требований и проектированием предлагалось разрабатывать систему-прототип.

Как показано на рис. 22.1, в модели водопада разработка программного обеспечения осуществляется посредством последовательности шагов. Каждый шаг основывается на действиях предыдущего шага. Проектирование логически следует за разработкой требований, кодирование — за проектированием и т.д. Модель водопада широко использовалась в последующие два десятилетия и успешно служила в качестве модели процесса для различных средних и больших проектов.

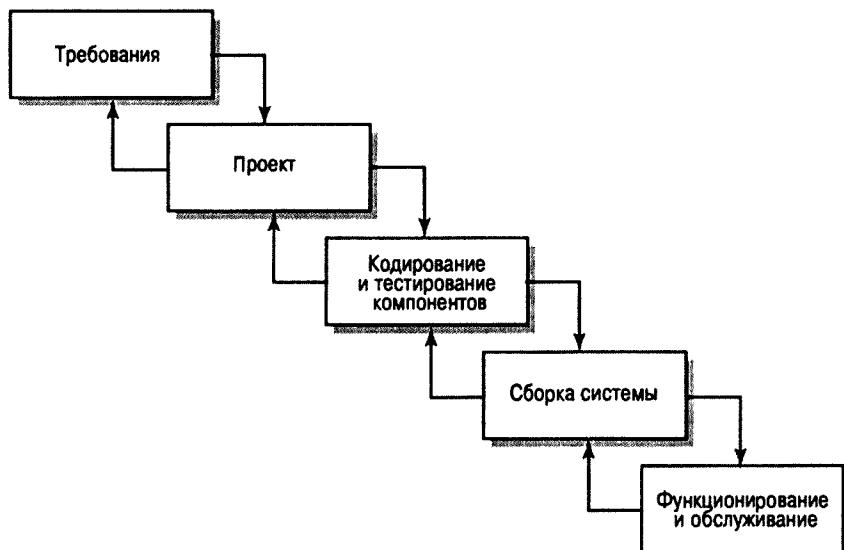


Рис. 22.1. “Модель водопада” процесса разработки программного обеспечения

Заметим, что в рис. 22.1 не указывается на необходимость создания прототипа (как, к сожалению, и было принято при применении данной модели). Это прискорбная ошибка, к которой мы еще вернемся.

В модели водопада возросла роль требований. Их разработка стала необходимым первым шагом при создании программного обеспечения, а также являлась основой проектирования и написания программного кода. Однако это же стало источником трудностей, так как в данной модели полная тщательная разработка требований и документов проектирования была обязательным условием окончания каждой из этих фаз. Кроме того, возможно, из-за неправильного применения слишком рьяными командами разработчиков, эта модель стала олицетворять застывший косный подход к разработке, когда требования "заморожены" на время жизни проекта, изменения запрещены, а процесс разработки "живет" своей собственной жизнью. В таком случае со временем команда может оказаться совершенно оторванной от реальности, на которой изначально основывался проект.

Дополнительные проблемы возникают, когда приходится решать задачу управления масштабом (рис. 22.2). Если же пытаться применять модель водопада к проекту, который изначально имеет масштаб 200%, результаты могут быть катастрофическими. К сроку сдачи ничего не работает, тестирование компонентов и сборка системы искусственно ускорены или не выполнены вообще, значительные средства затрачены на спецификацию, проектирование и написание кода тех функций системы, которые никогда не будут предоставлены. В результате нет ничего, что можно представить клиенту.

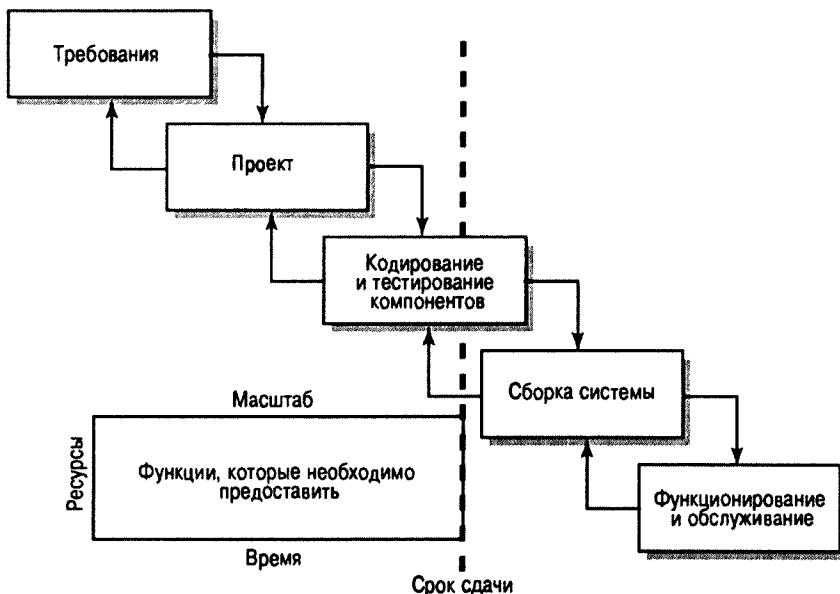


Рис. 22.2. Применение модели водопада к проекту с 200%-ным масштабом

В основном, именно эти причины привели к тому, что со временем модель водопада стала менее популярной. В результате вновь возникла тенденция переходить непосредственно к кодированию, не имея адекватного представления о требованиях к системе, что и было одной из основных проблем, которую пыталась решить модель водопада!

Сpirальная модель

В своей работе Барри Бом (Barry Boehm) (1988, а) рекомендует другую схему процесса разработки программного обеспечения. Его "спиральной моделью" программной разработки руководствуются те, кто верит, что путь к успеху состоит из инкрементных разработок на основе рисков (рис. 22.3).

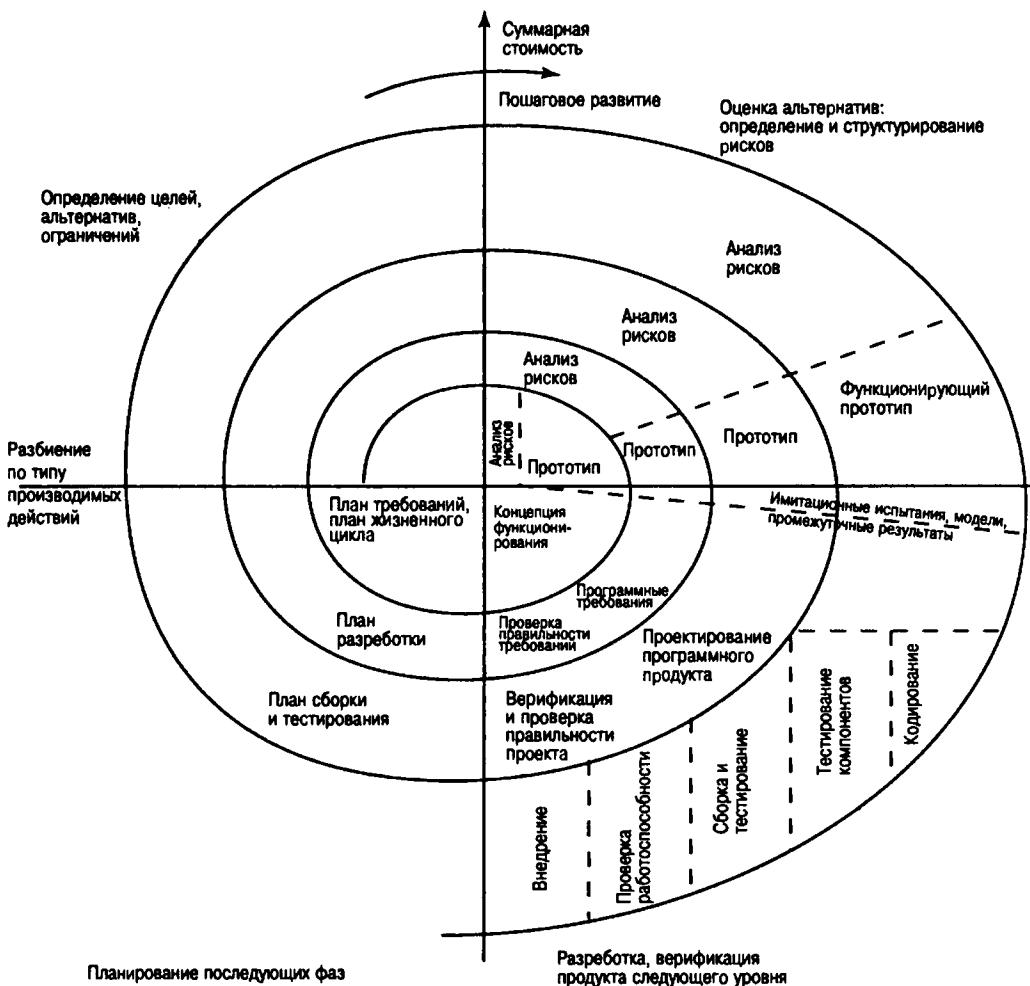


Рис. 22.3. Спиральная модель разработки

При использовании спиральной модели сначала создаются серии основанных на рисках прототипов, а затем проводится структурированный процесс построения конечной системы (аналогичный модели водопада). Конечно, при неправильном использовании спиральной модели могут возникать те же проблемы, что и в модели водопада. Проекты иногда сводятся к методу проб и ошибок; производятся отдельные составные части, которые необходимо наращивать и скреплять с помощью жевательной резинки или прово-

локи, что иногда еще называют процессом создания постоянно переделываемого кода. Преимущество заключается только в способности создавать необслуживаемый и непонимаемый код в два-три раза быстрее, чем при использовании более ранней технологии.

Однако если посмотреть на спиральную модель более внимательно, она может помочь в решении некоторых задач управления требованиями, рассматриваемых в данной книге. В частности, спиральная модель начинается с планирования требований и проверки правильности концепции, затем следует создание одного или нескольких прототипов, призванных на ранних этапах подтвердить наше понимание требований к системе. Положительной особенностью этого процесса является множество возможностей получения реакции пользователей и клиентов, что направлено на более раннее выявление синдрома “да, но...”. Оппоненты этого строгого подхода отмечают, что в современных условиях непозволительной роскошью является тратить время на полную проверку концепции, два-три прототипа, а также на строгое следование модели водопада.

Рассмотрим, что произойдет, если спиральную модель применить к проекту с 200%-ным масштабом. Результат представлен на рис. 22.4. Приходится согласиться, что он не пампного лучше, чем при использовании модели водопада; с другой стороны, можно отметить, что к моменту сдачи, по крайней мере, один или два прототипа работоспособны и получены отзывы заказчика. (Конечно, значительная часть этих отзывов будет касаться отсутствия готового к тиражированию программного обеспечения!)



Рис. 22.4. Применение спиральной модели к проекту с 200%-ным масштабом

Итеративный подход

В 1990-е годы многие команды перешли к использованию нового подхода, который сочетает лучшие качества модели водопада и спиральной модели. Кроме того, он также содержит некоторые дополнительные конструкции из новой дисциплины инженерии программных процессов. Впервые предложенный Крачтеном (Kruchten, 1995), “итеративный подход” к настоящему времени хорошо описан в ряде книг, в том числе в

работах Крачтена (Kruchten, 1999) и Ройса (Royce, 1998). Этот подход доказал свою эффективность для широкого спектра проектов и имеет ряд преимуществ по сравнению с водопадной и спиральной моделями разработки.

В традиционных моделях процесса разработки развитие проекта осуществляется путем последовательного выполнения определенных действий, когда разработка требований предшествует проектированию, проектирование – реализации и т.д. Это достаточно разумно. В итеративном процессе фазы жизненного цикла отделяются от логической деятельности по разработке программного обеспечения, проводимой в каждой фазе, что позволяет повторно возвращаться к деятельности по разработке требований, проектированию и реализации на различных итерациях проекта. Кроме того, как и в спиральной модели, каждая итерация проектируется так, чтобы уменьшить всевозможные риски на данной стадии разработки.

Фазы жизненного цикла

Итеративный подход состоит из четырех фаз жизненного цикла: *начало*, *исследование*, *построение* и *внедрение*, что соответствует естественным состояниям проекта в указанные периоды времени (рис. 22.5).

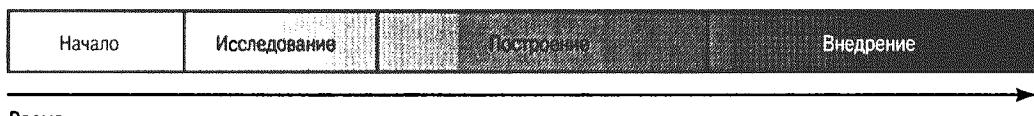


Рис. 22.5. Фазы жизненного цикла в итеративном подходе

- Начало.** На данной стадии команда уделяет основное внимание пониманию бизнесварианта проекта, определению его масштаба, достижимости реализации. Производится анализ проблемы, создается документ-концепция (Vision document), предварительно оцениваются график, бюджет, а также факторы риска проекта.
- Исследование.** Уточняются требования к системе, задается исходная выполняемая архитектура и, как правило, разрабатывается и демонстрируется ранний прототип достижимости.
- Построение.** Главное внимание уделяется реализации. На этой стадии пишется большая часть программного кода, завершается проектирование и доработка архитектуры.
- Внедрение.** Производится бета-тестирование; пользователи и команда сопровождения системы получают опыт работы с приложением. Протестированная базовая версия приложения передается сообществу пользователей и разворачивается для использования.

Итерации

В каждой фазе проект, как правило, проходит множество итераций (рис. 22.6). *Итерация* – это приводящая к появлению некой выполняемой программы последовательность действий, для которой заданы план и критерий оценки. Каждая итерация основы-

вается на функциональных возможностях предыдущей итерации; таким образом, проект разрабатывается "итеративным инкрементным" методом.

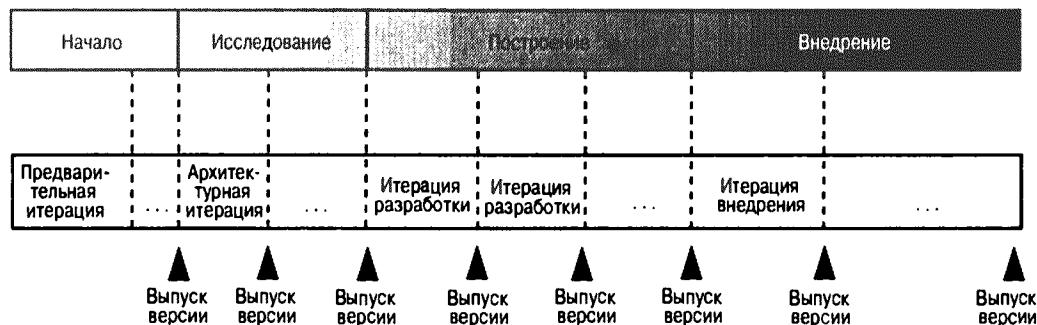


Рис. 22.6. Фазы и итерации, приводящие к появлению жизнеспособных версий

Существует ряд критериев отбора итераций. Ранние итерации следует разрабатывать для оценки жизнеспособности выбранной архитектуры, для некоторых самых важных прецедентов и прецедентов с высоким риском.

Рабочие процессы

В итеративном подходе деятельности, связанные с разработкой программного обеспечения, организованы в *рабочие процессы*. Каждый рабочий процесс состоит из логически связанного множества деятельности и определяет, в каком порядке их следует проводить, чтобы создать жизнеспособный рабочий продукт или "артефакт". Хотя количество (и качество) рабочих процессов может варьироваться в зависимости от конкретной компании и проекта, обычно имеется, как минимум, шесть рабочих процессов (рис. 22.7).

На каждой итерации команда уделяет каждому рабочему процессу столько времени, сколько считает нужным. Таким образом, итерация может рассматриваться как мини-водопад, где представлены действия по разработке требований, анализу и проектированию и т. д., но каждый мини-водопад "настраивается" на специфические потребности данной итерации. Размеры "холмов" на рис. 22.7 отражают относительные трудозатраты на соответствующие рабочие процессы. Например, в фазе исследования значительное время уделяется уточнению требований и определению архитектуры, которая будет поддерживать функциональные возможности. Деятельности могут осуществляться последовательно (истинный мини-водопад) или параллельно, в зависимости от нужд конкретного проекта.

С точки зрения управления требованиями, итеративный подход имеет два существенных преимущества.

1. *Раннее получение "да, но..."*. В результате каждой итерации получается выполняемая версия; так что уже на самых ранних этапах проекта клиенты имеют возможность видеть рабочий продукт. Конечно же, их реакцией будет "да, но...", но на этой ранней стадии были затрачены только минимальные средства. При каждой последующей итерации размеры этих "но..." будут уменьшаться, и вы и ваш клиент постепенно подойдете к удовлетворительной системе.

	Начало	Исследование	Построение	Внедрение
Рабочие процессы				
Управление требованиями				
Анализ и проектирование				
Реализация				
Тестирование				
Развертывание				
Вспомогательные процессы				
Управление конфигурацией и изменениями				
Управление проектом и процессом				

Рис. 22.7. Рабочие процессы итеративного подхода

2. Легче управлять масштабом. Если в первой итерации процент невыполнения составил 30% и более, это свидетельствует о том, что масштаб проекта, возможно, задан неправильно и его необходимо сократить. Однако даже с неверно заданным масштабом к сроку сдачи будет разработано несколько выполняемых итераций, а последняя может даже быть развернута. Даже если некоторых функциональных возможностей не хватает, версия будет представлять определенную ценность для пользователя. Если функции были правильно выбраны и упорядочены, это позволит заказчику достичь своих целей, по крайней мере частично, в то время как команда продолжит работу над итерациями разработки. Если архитектура устойчива и отвечает основным техническим требованиям, у команды будет надежная платформа для предоставления дополнительных функциональных возможностей.

Что делать, что делать...

Независимо от того, какая модель используется, команда должна предложить по крайней мере один устойчивый оценочный прототип для раннего получения реакции клиента.

Одна из предпосылок данной книги состоит в том, что раннее получение реакции “да, но...” является одной из самых главных задач в процессе разработки программного обеспечения.

- Сколько раз придется делать прототип?
- Изменяются ли требования заказчика для каждого нового прототипа?
- Обречены ли мы на неудачу, независимо от того, какому процессу следуем?

Ответы таковы. Да, клиенты будут требовать изменений при каждом представлении. Нет, мы не обречены. Причина в том, что проблемы, связанные с внесением изменений, которые возникают после того, как клиент имеет возможность увидеть предлагаемую реализацию и поработать с ней, невелики по сравнению с важностью отзыва клиента на первый осозаемый артефакт процесса.

Таким образом, хотя мы предпочитаем использовать в наших разработках итеративную модель, независимо от того, какую модель процесса разработки используете вы, необходимо *обеспечить создание по крайней мере одного устойчивого оценочного прототипа для получения реакции клиента до того, как будет выполнен основной объем действий по проектированию и написанию программного кода*. (Помните о действиях по созданию прототипа, которые Ройс (Royce, 1970) изначально предлагал для модели водопада?!). Благодаря уменьшению числа изменений до разумного уровня разработчику удается путем внесения инкрементных изменений (как правило, в интерфейсы пользователя, отчеты и другие выходные результаты) создать качественный устойчивый технический проект и реализовать его. Затем тщательно организованный процесс завершения проектирования, кодирования, тестирования компонентов и сборки системы обеспечит для продукта прочный фундамент и в значительной мере будет способствовать действиям по обеспечению качества и тестированию.

Заключение части 4

В части 4, “Управление масштабом”, мы выяснили, что проблема масштаба проекта является весьма типичной. Проекты, как правило, инициируются с объемом функциональных возможностей, вдвое превышающим тот, который команда может реализовать, обеспечив приемлемое качество. Это не должно нас удивлять: заказчики хотят большего, маркетинг хочет большего и мы также желаем большего. Тем не менее, нам необходимо ограничить себя, чтобы иметь возможность предоставить в срок хоть *что-нибудь*.

Мы рассмотрели различные методы задания очередности выполнения (приоритетов) и ввели понятие базового уровня (совместно согласованного представления о том, в чем будут состоять ключевые функции системы как рабочего продукта нашего проекта) — понятие, задающее точку отсчета и ориентир для принятия решений и их оценки. Если масштаб и сопутствующие ожидания превышают реальные, в любом случае придется сообщать некие неприятные новости. Мы остановились на философии привлечения заказчика к процессу принятия трудных решений. В конце концов, мы являемся только исполнителями, а принимать решения должны наши заказчики, ведь это их проект. Поэтому вопрос стоит так: что *обязательно* должно быть сделано в следующей версии при имеющихся ресурсах проекта?

Даже в этом случае нам придется вести переговоры; в некотором смысле вся жизнь и, конечно, весь бизнес состоят из переговоров, и мы не должны этому удивляться. Мы кратко перечислили некоторые приемы ведения переговоров и намекнули, что они могут понадобиться команде.

Нельзя ожидать, что данный процесс полностью решит проблему масштаба, точно так же как никакой другой процесс в отдельности не решит проблемы разработки приложений. Но указанные шаги окажут заметное воздействие на размеры проблемы, позволят разработчикам приложения сконцентрировать свои усилия на критически важных подмножествах функций и в несколько приемов предоставить высококачественные системы, удовлетворяющие или превосходящие ожидания пользователей. Привлечение заказчика к решению проблемы управления масштабом повышает обязательства сторон, способствует взаимопониманию и доверию между заказчиком и командой разработчиков приложения. Имея всеобъемлющее определение продукта (документ-концепцию) и сократив масштаб проекта до разумного уровня, мы можем *надеяться* на успех в следующих фазах проекта.

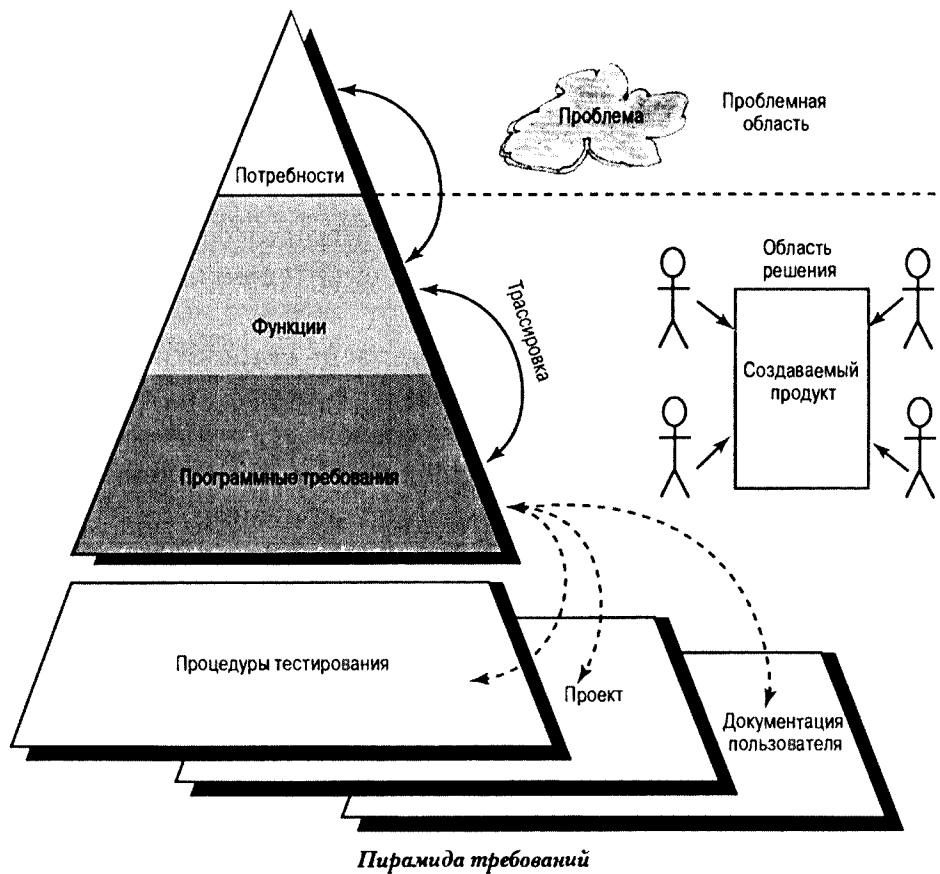
Часть 5

Уточнение определения системы

- Глава 23. Требования к программному обеспечению
- Глава 24. Уточнение прецедентов
- Глава 25. Спецификация требований к программному обеспечению (Modern Software Requirements Specification)
- Глава 26. Неоднозначность и уровень конкретизации
- Глава 27. Критерии качества требований к программному обеспечению
- Глава 28. Теоретически обоснованные формальные методы спецификации требований



Предыдущие части были посвящены анализу проблемы, выявлению потребностей пользователей, сбору и документированию желаемых функций продукта и управлению ими. После того как функции продукта определены, следующая задача состоит в уточнении спецификации до уровня детализации, пригодного для проведения процессов проектирования, написания программного кода и тестирования. Мы оказываемся в самой середине пирамиды требований, на уровне спецификации, как показано на рисунке.



Пирамида требований

В части 5 будут обсуждаться методы исследования и организации требований к программному обеспечению, а также способы доведения их до всех участников. Часть завершается рассмотрением одной из важнейших тем как сформулировать требования коротко и ясно.

Независимо от того, каким методом вы пользуетесь для сбора требований, необходимо помнить, что *собранные требования* (*и только они!*) служат движущей силой проекта. Если окажется, что они недостаточны или даже ошибочны, их нужно быстро и официально изменить; так что правило остается в силе. При таком подходе вся команда имеет перед собой ясную цель и может сосредоточить свои усилия на выявлении и реализации требований, сократив время, потраченное напрасно. Начнем с изучения самой природы требований.

Глава 23

Требования к программному обеспечению

Основные положения

- Полный набор требований можно получить, определив вводы, выводы, функции и атрибуты системы, а также атрибуты ее окружения.
- В требования не следует включать общую информацию (графики, планы разработки, выделенные средства, тесты), а также информацию, касающуюся проектирования.
- Процесс разработки требований/проектирования является итеративным; выявленные требования ведут к выбору конкретных вариантов проектирования, которые в свою очередь приводят к возникновению новых требований.
- Ограничения проектирования касаются вариантов проектирования системы или процессов, с помощью которых система разрабатывается.

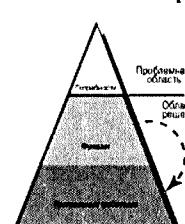
В предыдущих частях мы намеренно оставили определенные для системы функции на высоком уровне абстракции с тем, чтобы удобнее было выполнить следующие действия.

- Понять форму системы, сосредоточив внимание на ее функциях и на том, как они выполняют потребности пользователя.
- Оценить полноту и целостность системы, а также ее соответствие среде.
- Использовать данную информацию для определения возможности построения системы и управления ее масштабом перед тем, как будут производиться значительные инвестиции.

Кроме того, оставаясь на высоком уровне абстракции, мы воздерживаемся от преждевременного принятия чрезмерно ограничительных решений по требованиям, т. е. до того, как люди, непосредственно занимающиеся реализацией системы, получат возможность внести свой вклад в определение системы. В части 5, "Уточнение определения системы," мы переходим к разработке функций системы до уровня детализации, достаточного, чтобы гарантировать, что в ходе проектирования и кодирования будет создана система, полностью соответствующая потребностям пользователя. При этом мы переходим на следующий уровень конкретизации и создаем более полную, глубокую модель требований к разрабатываемой системе. Конечно, количество информации, которой надо управлять, также увеличивается, и нам необходимо подготовиться к работе с ней.

Уровень конкретизации, необходимый на этом шаге, зависит от множества факторов, среди которых содержание приложения и профессиональные навыки команды разработчиков. Для высокобезопасных систем программного обеспечения в медицине, авиации или интерактивной торговле уровень конкретизации особенно высок. Процесс уточнения может включать в себя использование формальных средств обеспечения качества, просмотры, ревизии, моделирование и т.п. В системах, где последствия сбоев не столь катастрофичны, уровень трудоемкости более умеренный. В этих случаях просто необходимо сформулировать определение системы достаточно четко, чтобы его могли понять все участники процесса, а также для того, чтобы обеспечить эффективные условия разработки и "достаточно высокую" вероятность успеха. Руководствуясь pragматическими и экономическими соображениями, следует создать достаточное количество спецификаций требований, чтобы разрабатываемая программа была именно тем, чего хочет пользователь.

Точно так же, как не существует языка программирования, подходящего для всех без исключения приложений, нет и универсального способа разработки более детальных спецификаций.



В различных средах требуются различные методы, и тем, кто пишет требования и управляет ими, необходимо овладеть разнообразными профессиональными приемами. Нам приходилось в своей практике применять множество методов – от применения достаточно строгих документов требований, традиционных баз данных или архивов требований до использования моделей прецедентов и более формальных методов. Но всегда главное внимание уделялось спецификации на естественном языке, написанной достаточно ясно, чтобы ее могли понять все заинтересованные лица, заказчики, пользователи, разработчики и тестологи, но также достаточно конкретно ("Максимальная горизонтальная скорость оси 4 должна составлять 1м/с"), чтобы можно было выполнить верификацию и продемонстрировать соответствие. Перед тем как организовывать требования к системе, рассмотрим саму природу этих требований.

Определение требований к программному обеспечению

В главе 2 мы дали следующее определение требования.

- Некое свойство программного обеспечения, необходимое пользователю для решения проблемы при достижении поставленной цели.
- Некое свойство программного обеспечения, которым должна обладать система или ее компонент, чтобы удовлетворить требования контракта, стандарта, спецификации либо иной формальной документации.

Требования к программному обеспечению – это то, что данная программа делает для пользователя, прибора или другой системы. Начинать их поиск следует среди того, что "входит" в систему и "выходит" из нее, т.е. необходимо рассмотреть взаимодействия системы с ее пользователями.

Для этого проще всего сначала представить себе систему как некий черный ящик и подумать о том, что следует определить, чтобы полностью описать, что делает этот черный ящик.

Кроме входящей и выходящей информации, также необходимо обратить внимание на некоторые другие характеристики системы, в том числе на ее производительность и другие типы сложного поведения, а также на иные способы взаимодействия системы с ее средой (рис. 23.1).

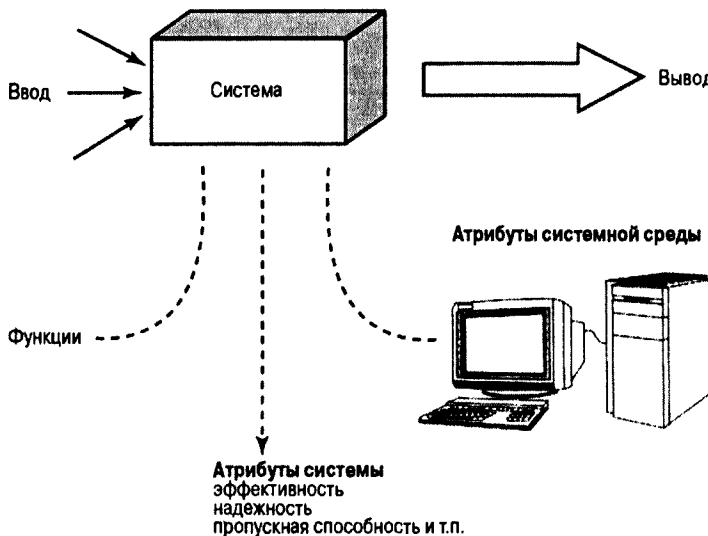


Рис. 23.1. Элементы системы

Используя аналогичный подход, Дэвис (Davis, 1999) отметил, что для полного определения системы необходимо описать следующие пять основных категорий элементов.

- 1. Вводы системы.** Необходимо не только указать содержимое ввода, но и, если нужно, подробно описать устройства, а также протокол (форму, внешний вид и содержание) ввода. Как известно большинству разработчиков, этот класс может содержать значительный объем сведений и подвергаться частым изменениям, особенно в средах GUI, мультимедиа и Internet.
- 2. Выводы системы.** Нужно описать поддерживаемые устройства вывода, такие как речевой вывод или видеотерминал, а также протокол и форматы генерируемой системой информации.
- 3. Функции системы.** Отображение вводов в выводы и их различные комбинации.
- 4. Атрибуты системы.** Типичные неповеденческие требования, такие как надежность, удобство сопровождения, доступность и пропускная способность, которые должны учитывать разработчики.
- 5. Атрибуты системной среды.** Это такие дополнительные неповеденческие требования, как способность системы функционировать в условиях определенных операционных ограничений и нагрузок, а также совместимость с операционной системой.

На протяжении ряда лет мы использовали это разбиение на категории и убедились в его работоспособности. Оно способствует целостному и полному восприятию проблем требований. Таким образом, можно предложить следующее определение.

Полный набор требований к программному обеспечению можно задать, определив следующее:

- вводы системы;
- выводы системы;
- функции системы;
- атрибуты системы;
- атрибуты системной среды.

В результате мы сможем оценить, является ли некая "вещь" требованием к программному обеспечению, проверив, соответствует ли она данному подробному определению.

Взаимосвязь между функциями и требованиями к программному обеспечению

Ранее мы потратили некоторое время на изучение "функций" системы. Функции представляют собой описания желательного и полезного поведения. Теперь мы увидим, что существует соответствие между функциями и требованиями к программному обеспечению.

В документе-концепции описаны функции на языке пользователя. Требования к программному обеспечению, в свою очередь, описывают эти функции более подробно. Чтобы предоставить пользователю некую функцию, разработчики должны выполнить одно или несколько конкретизированных программных требований. Другими словами, функции помогают понимать и обсуждать систему на высоком уровне абстракции, но с их помощью невозможно описать систему и создать на основании этого описания код. Для этой цели функции слишком абстрактны.



Требования к программному обеспечению более конкретизированы. Мы собираемся на их основе создавать код; следовательно, они должны быть "тестируемыми", т.е. достаточно конкретными, чтобы можно было проверить, действительно ли система реализует некое заданное требование. Предположим, мы разрабатываем систему обнаружения неполадок для конвейерного производства или организации, разрабатывающей программное обеспечение. В табл. 23.1 представлена некая функция документа-концепции и связанный с ней набор требований. Эта связь и возможность осуществлять трассировку функций к требованиям (и наоборот) лежат в основе очень важного понятия в области управления требованиями, известного как "трассируемость" (traceability). (Эту тему мы будем обсуждать позднее.)

Таблица 23.1. Требования, связанные с некоторой функцией документа-концепции

Документ-концепция	Программные требования
Функция 63. Система обнаружения неполадок будет предоставлять информацию об обнаруженных дефектах, чтобы помочь пользователю оценить состояние проекта	SR63.1. Информация будет предоставляться в виде отчета-гистограммы, где по оси x откладывается время, а по оси у – количество обнаруженных дефектов

Окончание табл. 23.1

Документ-концепция	Программные требования
	SR63.2. Пользователь может задавать временной период в днях, неделях или месяцах
	SR63.3. Пример отчета об обнаруженных дефектах представлен на прилагаемом рисунке

Дilemma требований: что или как

Требования сообщают разработчикам, что должна делать система, и должны описывать системные вводы, выводы, функции, атрибуты, а также атрибуты системной среды. Но существует много другой информации, которую требования *не должны* содержать. В частности, не следует указывать не являющиеся необходимыми подробности проектирования или реализации, а также информацию, связанную с управлением проектом. Кроме того, не следует описывать, как система будет тестироваться. Тогда требования будут сосредоточены на поведении системы и будут изменяться только тогда, когда меняется поведение.

Исключение информации, связанной с управлением проектом

Иногда из соображений удобства менеджер проекта может включить в набор требований информацию, связанную с управлением проектом, а именно графики, планы



График



Проектные планы



Бюджет



Испытания

управления конфигурацией, планы испытаний, бюджеты и штатные расписания. Как правило, этого следует избегать, так как изменения в этой информации (например, изменения графика) будут приводить к необходимости замены устаревших "требований". (Когда требования датируются, им меньше доверяют и чаще игнорируют.) Кроме того, неизбежные дебаты относительно вопросов управления проектом следует отделить от обсуждения того, что должна делать система. Существуют различные заинтересованные лица, и у каждого из них свои цели.

Бюджет тоже может выглядеть как требование; тем не менее это информация другого рода, не удовлетворяющая нашему определению и, следовательно, не относящаяся к системным или программным требованиям. Бюджет — важная информация, особенно когда разработчики решают, какую стратегию реализации выбрать, поскольку некоторые стратегии могут быть слишком дорогостоящими, а время их выполнения слишком длительным. И все же это не требование. Аналогично описание процедур тестирования или приемки (с помощью которых мы будем определять, что требования действительно выполнены) также не удовлетворяет определению и, следовательно, не является требованием.

Как следует из нашей практики, все же иногда полезно включить некоторую дополнительную информацию. Например, часто составитель требования может "намекнуть", какой тест подойдет для разработанного им требования. В конечном счете, он имеет представление о том, какое именно поведение соответствует данному требованию, и разумно воспользоваться его помощью.

Исключение информации, относящейся к проектированию

Требования также не должны содержать информацию о системной архитектуре и техническом проекте. В противном случае можно неизвестно ограничить команду в выборе наиболее подходящих для данного приложения вариантов проектирования.



(“Эй, мы должны проектировать его таким способом; так сказано в требованиях.”)

Исключить из требований детали, относящиеся к управлению проектом и тестированию, достаточно просто. Но исключение деталей проектирования и реализации обычно гораздо более сложная и тонкая работа. Предположим, что первое требование в табл. 23.1 сформулировано следующим образом: “SR63.1. Информация об обнаруженных дефектах будет предоставляться в виде отчета-гистограммы, написанного на Visual Basic, причем основные причины будут откладываться по оси x, а количество дефектов – по оси у” (рис. 23.2).

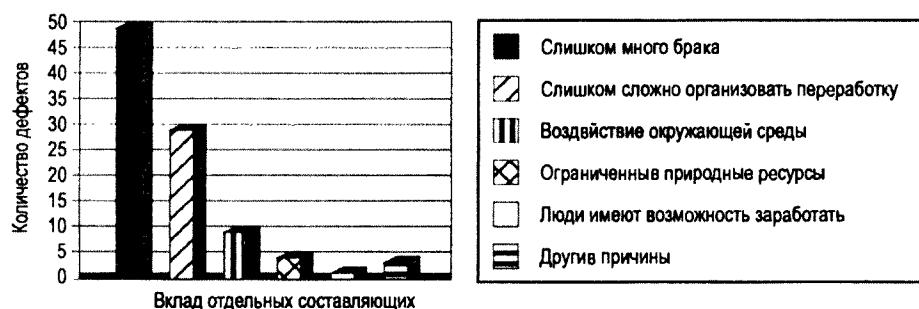


Рис. 23.2. Отчет-гистограмма

Хотя указание Visual Basic в качестве языка написания программы является достаточно явным нарушением наших рекомендаций (поскольку оно не представляет ни ввод, ни вывод, ни функцию, ни атрибут поведения), полезно задать вопрос: “Кто принял решение, что гистограмма должна быть реализована на Visual Basic, и почему оно было принято?” Возможны следующие ответы.

- Один из технически ориентированных членов группы, определяя документ-концепцию, решил, что следует указать Visual Basic, так как это “наилучшее” решение проблемы.
- Это может быть указание пользователя. Опасаясь, что разработчики могут применить некую более дорогостоящую и менее доступную технологию, пользователь решает, что технология VB – доступная и относительно дешевая, и хочет, чтобы использовали именно ее.
- Политическое решение, принятное организацией-разработчиком, может диктовать, чтобы все приложения разрабатывались на Visual Basic. Чтобы пресечь все попытки проигнорировать данную политику, руководство настаивает, чтобы упоминание Visual Basic было повсюду, где это возможно, включено в документы требований.

Если технический разработчик решил включить упоминание о Visual Basic, поскольку просто предпочитает данный язык, оно, бесспорно, не является легитимным элементом списка требований. Если это требование предложено пользователем, ситуация несколько

иная. Если клиент отказывается платить за систему, написанную не на Visual Basic, лучше всего трактовать подобное пожелание как требование, хотя мы поместим его в специальный класс, называемый ограничениями проектирования, так что оно будет отделено от обычных требований, описывающих только внешнее поведение. Тем не менее это именно ограничение реализации, налагаемое на команду разработчиков. (Кстати, если вы думаете, что этот пример нереалистичен, напомним, что до конца 1990-х обычным требованием Министерства обороны США к разработчикам программного обеспечения было “создавать системы, используя язык Ada”.)

Однако рассуждая в этом примере о Visual Basic, мы можем пропустить более тонкий и, возможно, более важный момент анализа требования: *почему информация об обнаруженных дефектах должна предоставляться в виде отчета-диаграммы?* Почему не круговая диаграмма или другое представление информации? Подразумевает ли слово “отчет” некий бумажный документ или информация может отображаться на экране компьютера? Нужно ли хранить эту информацию с тем, чтобы ее можно было передавать другим программам или пересылать в корпоративную сеть? Функцию, описанную в документе-концепции, практически всегда можно осуществить множеством способов, причем некоторые из них имеют совершенно определенные реализационные последствия.

Во многих случаях на описание проблемы, которое служит основой формулирования требования, влияют представления пользователя о возможных способах ее решения. То же верно и по отношению к разработчикам, которые совместно с пользователем участвуют в процессе формулирования функций документа-концепции и требований. Как гласит старая пословица, если ваш единственный инструмент — молоток, все ваши проблемы похожи на гвозди. Но нам нужно проявлять бдительность, чтобы необязательные ограничения реализации не попадали в требования, и мы должны удалять подобные ограничения везде, где это возможно.

Больше внимания требованиям, а не проектированию

До сих пор мы трактовали требования к программному обеспечению, решения и ограничения проектирования так, как если бы они были различными сущностями, которые можно четко разграничить. Иными словами, мы предполагали следующее.

- Разработка требований предшествует проектированию.
- Пользователи и заказчики принимают решения относительно требований.
- Разработчики принимают решения относительно проектирования, так как они лучше подготовлены к тому, чтобы выбрать среди многих вариантов проектирования наиболее подходящий для удовлетворения конкретной потребности.

Это удачная модель, от которой можно отталкиваться при решении задач управления требованиями. Дэвис (Davis, 1993) назвал ее парадигмой “что вместо как”, где “что” — это требования (“что” система должна делать), а “как” представляет собой проектное решение, которое следует реализовать для достижения этой цели.

Для такого представления есть причины. Действительно, лучше понять требования перед тем, как приступить к проектированию, а боль-

Требования:
что система
должна
делать

Проектирование:
как система
должна
это делать

шипство ограничений проектирования ("использовать XYZ-библиотеку классов для доступа к базе данных") являются важными проектными решениями, записанными в активы требований с тем, чтобы мы знали, что они получены по условиям контракта или по достаточно веским техническим причинам.

Если мы не сможем произвести такую классификацию вовсе, картина будет очень запутанной, и мы не сумеем отделить разработку требований от проектирования. Более того, мы не будем знать, кто за что отвечает в процессе разработки. Или того хуже, наши заказчики будут диктовать нам проектные решения, а разработчики — требования.

Однако существует пока невидимая, но достаточно серьезная проблема, которая противоречит представленной нами простой парадигме. Вернемся к нашему рабочему примеру. Если команда принимает проектное решение использовать ПК-технологию для подсистемы "Центральный блок управления" (ЦБУ) системы HOLIS, это, вероятно, окажет некое внешнее воздействие на пользователя (он будет видеть подсказку или экран-приглашение). Если мы захотим воспользоваться некоторыми возможностями ОС, то соответствующие библиотеки классов будут, конечно же, демонстрировать внешнее поведение пользователю. (Заметим, что его можно скрыть, но это не нужно.)

Если воспользоваться предложенным в данной главе определением, возникает вопрос: *если проектное решение воздействует на внешнее поведение, заметное пользователю, становится ли такое решение (явно воздействующее на ввод или вывод системы) требованием?* Ответ ("да", "нет" или "не имеет значения") будет зависеть от вашей интерпретации определения и проведенного нами анализа. Но это проливает свет на очень важный аспект, так как его понимание жизненно необходимо для уяснения природы итеративного процесса в целом. Давайте рассмотрим его более подробно.

Итерационный цикл разработки требований и проектирования

В реальности деятельности по разработке требований и проектированию должны осуществляться итеративно. *Выявление требований, их определение и принятие проектных решений циклически чередуются*. Процесс заключается в непрерывном крутовороте.

Имеющиеся требования приводят к выбору определенных вариантов проектирования,
а

те в свою очередь могут инициировать новые требования.

Иногда появление новой технологии может привести к тому, что мы отбросим множество предположений о том, какими должны быть требования; мы можем найти совершенно иной подход, перечеркивающий старую стратегию. ("Давайте целиком уберем модуль *клиент/доступ к данным/GUI* и заменим его навигационным интерфейсом.") Это важный и правомерный источник изменения требований.

Такой процесс закономерен, попытка поступать по-другому будет безрассудством. С другой стороны, во всем этом есть серьезная опасность: если мы не достигли истинного понимания потребностей заказчика и не привлекали его к процессу разработки требований (а иногда даже и к процессу понимания наших действий по проектированию), может быть принято *неверное* решение. При правильном осуществлении процесс "непрерывного пересмотра требований и проектирования" является просто фантастическим, так как позволяет постоянно совершенст-

вовать нашу способность удовлетворить реальные потребности клиентов. *Именно в этом и состоит суть эффективного интерактивного управления требованиями.* Но если данный процесс осуществляется неправильно, мы постоянно “гоняемся за хвостом нашей технологии”, и результаты весьма плачевны. Мы никогда не говорили, что это будет легко.

Дальнейшая характеристика требований

Итак, существуют различные “разновидности” требований. В частности, мы считаем полезным выделить следующие три типа (рис. 23.3).

- Функциональные требования к программному обеспечению
- Нефункциональные требования к программному обеспечению
- Ограничения проектирования

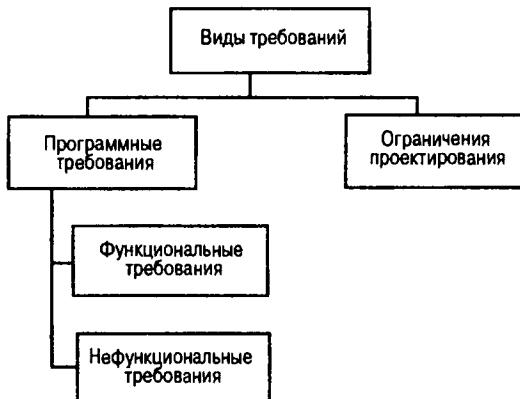


Рис. 23.3. Типы требований

Функциональные требования к программному обеспечению

Функциональные требования описывают, как ведет себя система. Эти требования обычно ориентированы на действия (“Когда пользователь делает x, система будет делать y”). Большинство продуктов и приложений, предназначенных для выполнения полезной работы, содержит множество функциональных требований к программному обеспечению. Программное обеспечение используется для реализации большей части функциональных возможностей.

При определении функциональных требований следует искать золотую середину между слишком конкретизированной формулировкой требования и слишком общей и неоднозначной. Например, как правило, ни к чему иметь обобщенное функциональное требование следующего вида: “Когда вы нажимаете эту кнопку, система включается и работает”. С другой стороны, формулировка требования, которая состоит из нескольких страниц текста, по-видимому, слишком конкретизирована, но может быть правомерной в некоторых частных случаях. Мы вернемся к рассмотрению этого вопроса в главе 26.

Оказывается, что большинство функциональных требований можно сформулировать в виде простых декларативных определений или в форме прецедентов, которым посвя-

щена следующая глава. Опыт свидетельствует, что определение требований, состоящее из одного-двух предложений, обычно является наилучшим, когда нужно соотнести потребность пользователя с приемлемым для работы разработчика уровнем конкретизации.

Нефункциональные программные требования

До сих пор в данной главе приводились, в основном, примеры *поведенческих (функциональных)* требований к системе, основное внимание в которых уделялось вопросам ввода, вывода и обработки. Функциональные требования описывают, как система должна вести себя, когда ей предоставляются определенные входные данные или условия.

Но этого недостаточно для полного описания требований к системе. Необходимо также учитывать следующие характеристики, которые Грейди (Grady) (1992) назвал *“нефункциональными требованиями”*.

- Практичность (Usability)
- Надежность (Reliability)
- Производительность (Performance)
- Возможность обслуживания (Supportability)

Эти требования, как правило, используются для описания некоторых “атрибутов системы” или “атрибутов системного окружения” из нашего сложного определения. Благодаря их удобной классификации мы можем больше узнать о системе, которую необходимо создать. Рассмотрим каждый из перечисленных пунктов более подробно.

Практичность

Важно описать, насколько легко будущие пользователи смогут освоить данную систему. Может понадобиться определить различные категории пользователей: начинающие, “средние”, опытные, а также неграмотные пользователи – те, которые не владеют свободно родным языком “средних” пользователей, и т.д. Если предполагается, что заказчик будет просматривать требования и участвовать в их обсуждении (а так и должно быть!), следует все требования в этой сфере писать на естественном языке (описание практичности не может быть в форме конечного автомата).

Поскольку практичность зависит от точки зрения наблюдателя, как описать такой неясный набор требований? Ниже приводятся некоторые рекомендации.

- Указать необходимое время подготовки пользователя для достижения минимальной производительности (способности выполнять простые задачи) и операционной производительности (способности выполнять обычные текущие задачи). Как отмечалось, могут понадобиться отдельные описания для начинающих пользователей (которые, быть может, никогда прежде не видели компьютер), средних и опытных.
- Указать время выполнения типичных задач или транзакций, осуществляемых конечным пользователем. При создании системы ввода заказов на покупку, вероятно, наиболее типичными задачами, выполняемыми конечными пользователями, будут ввод, удаление или модификация заказа, а также проверка состояния заказа. Если пользователь знает, как выполнять эти задачи, сколько времени он потратит на ввод типичного заказа? Минуту? Пять минут? Час? Конечно, это будет зависеть от технической реализации (скорости модема, пропускной способности сети, ОП, мощности ЦП, которые совместно определяют время ответа, обеспечивающее сис-

темой), но время выполнения задач также напрямую зависит от практичности системы, и нужно иметь возможность определить это отдельно.

- Сравнить практичность новой системы с уже существующими современными системами, которые известны и пользуются успехом у пользователей. Иными словами, требование может выглядеть так: "Новая система должна быть признана подавляющим большинством (90%) пользователей такой же практичной, как и существующая XYZ-система". Напоминаем, что такие требования, равно как и все остальные, должны допускать возможность проверки соответствия; мы должны описывать требование так, чтобы пользователи могли проверить соответствие практичности установленному нами критерию.
- Оговорить существование систем интерактивных подсказок, программ-помощников, средств предупреждения, руководств пользователя и других форм документации и помощи, а также определить их необходимые функции.
- Следовать соглашениям и стандартам, разработанным для человека-машинного интерфейса. Чтобы новая система работала "почти так же, как та, которую я уже использовал", необходимо следовать согласованным стандартам от приложения к приложению. Например, вы можете указать требование соответствия общим стандартам практичности, таким как стандарты CUA (Common user access) компании IBM или стандарты Windows-приложений, опубликованные компанией Microsoft.

Примером подлинного прорыва в сфере практичности в компьютерном мире может служить разница между командными строками операционных систем DOS и UNIX и GUI-интерфейсами операционных систем Windows и Macintosh. GUI-интерфейсы значительно упростили использование компьютера для пользователей, не имеющих технического образования. Еще одним примером является Internet-браузер, который стал окном в мир World Wide Web и радикально ускорил освоение Internet средним пользователем.

Было предпринято несколько интересных попыток сделать более строгим весьма расплывчатое понятие практичности. Наибольший интерес представляет так называемый "Билль о правах пользователей" (Карат (Karat), 1998). Последняя его версия состоит из десяти основных пунктов.

1. Пользователь всегда прав. Если возникает проблема с использованием системы, то дело в системе, а не в пользователе.
2. Пользователь имеет право на программное и аппаратное обеспечение, которое легко монтируется и демонтируется без негативных последствий.
3. Пользователь имеет право на то, чтобы система делала в точности то, что обещано.
4. Пользователь имеет право на простые в использовании инструкции (руководства пользователя, интерактивные или контекстно-зависимые подсказки, сообщения об ошибках), которые позволяют ему понимать систему и использовать ее для достижения желаемых целей, а также эффективно и легко выходить из сложных ситуаций.
5. Пользователь имеет право на внимание со стороны системы, а также на то, чтобы иметь возможность получить ответ системы на запрос о внимании.
6. Пользователь имеет право на то, чтобы система предоставляла четкую, понятную и точную информацию о выполняемой задаче и ее выполнении.



7. Пользователь имеет право на то, чтобы его информировали о всех системных требованиях для успешного использования программного обеспечения или аппаратуры.
8. Пользователь имеет право знать о пределах возможностей системы.
9. Пользователь имеет право общаться с провайдером технологии и получать полные исчерпывающие ответы, когда в этом возникает необходимость.
10. Пользователь должен быть хозяином программных и аппаратных технологий, а не наоборот. Продукты должны использоваться естественно и интуитивно.

Отметим, что некоторые из перечисленных пунктов по своей сути являются неизменными и не могут быть кандидатами в требования. С другой стороны, ясно, что этот документ может служить отправной точкой при разработке вопросов и определении требований, касающихся практичности предлагаемого продукта.

Надежность

Конечно, никому не правятся ошибки, системные сбои или потери данных, и если подобные явления никогда в требованиях не упоминаются, пользователь, естественно, может предположить, что их не будет вовсе. Но в современном мире даже самый оптимистически настроенный пользователь знает, что ошибки и сбои неизбежны. Таким образом, в требованиях следует



указать, в какой степени система *обязана* вести себя приемлемым для пользователя образом. Как правило, здесь описываются следующие аспекты.

- *Доступность (availability)*. Система должна быть доступна для операционного использования в течение указанного времени (в процентном выражении). Иногда требование может указывать непрерывную "nonstop" доступность, т.е. 24 часа в сутки, 365 дней в году. Но чаще можно встретить указание 99-процентной или 99.9-процентной доступности между 8 часами утра и полуночью. Отметим, что требования должны указывать, что понимается под "доступностью". Означает ли 100-процентная доступность, что все пользователи должны иметь возможность использовать все системные услуги в любое время?
- *Среднее время между отказами (Mean time between failures, MTBF)*. Оно обычно указывается в часах, но может также указываться в днях, месяцах или годах. Здесь тоже нужна точность: требования должны четко определять, что понимается под "сбоем".
- *Среднее время восстановления (mean time to repair, MTTR)*. Как долго системе разрешается не работать после сбоя? MTTR может иметь несколько значений; например, пользователь может указать, что 90% всех системных сбоев должны ликвидироваться за 5 минут, а 99.9% — в течение часа. Вновь важна точность: требования должны четко указывать, означает ли восстановление, что все пользователи снова будут иметь возможность получать доступ ко всем услугам, или допускается частичное восстановление.
- *Точность (accuracy)*. Какая точность требуется системам с числовым выводом? Например, должны ли результаты в финансовых системах быть с точностью до пяти или доллара?
- *Максимальный коэффициент ошибок*. Как правило, выражается как число ошибок, приходящееся на тысячу строк кода (bugs/KLOC) или на одну функцию.

- **Количество различных ошибок.** Обычно ошибки делятся на незначительные, серьезные и критические. Здесь также важны четкие определения: требования должны определять, что понимается под "критической" ошибкой — полная потеря данных или невозможность использовать определенную часть функциональных возможностей системы.

В некоторых случаях требования могут указывать некоторые "ориентировочные" метрики надежности. Типичным примером этого является использование некой "метрики сложности" для оценки сложности и, тем самым, потенциальной "ошибочности" программы.

Производительность

К требованиям производительности обычно относится следующее.

- Время ответа для транзакции: среднее, максимальное
- Пропускная способность: число транзакций в секунду
- Емкость: сколько пользователей или транзакций может обслужить система
- Режимы снижения производительности: допустимые режимы работы при ухудшении параметров системы

Если новая система должна совместно с другими системами или приложениями использовать аппаратные ресурсы (ЦП, память, каналы, дисковую память, сетевой диапазон частот), может также потребоваться указать, насколько "цивилизованно" она себя при этом ведет.

Возможность обслуживания

Возможность обслуживания заключается в способности легко модифицировать программное обеспечение с целью внесения изменений и исправлений. В некоторых предметных областях можно заранее предвидеть вероятную природу будущих изменений, и требования могут содержать указание "времени ответа" группы поддержки для простых, средних и сложных изменений.

Предположим, мы создаем новую систему расчета заработной платы. Одно из многих требований к такой системе состоит в том, что она должна вычислять удерживаемые правительственные налоги для каждого работника. Пользователь, конечно, знает, что правительство каждый год меняет алгоритм вычисления налогов. Это изменение затрагивает две величины: вместо удержания X процентов от общей заработной платы работника, но не более \$P, новый закон требует удержания Y процентов, но не более \$Q. Таким образом, требование можно сформулировать так: "Модификации системы с целью задания новых коэффициентов налогообложения должны осуществляться командой в течение одного дня после получения уведомления от официальных властей".

Но предположим, что налоговая инспекция периодически вносит в данный алгоритм поправки, аналогичные следующей: "Для левшей с голубыми глазами ставка налогообложения должна составлять Z процентов, но не более \$R". Подобные модификации сложнее предусмотреть в программе; хотя можно попытаться сделать ее максимально гибкой. Команда, вероятно, согласится, что данная модификация попадает в категорию изменений "среднего уровня" сложности, для которых требование может задавать время реагирования — одна неделя.

Представим себе, что перед началом проекта менеджер отдела заработной платы сказал: "Возможно, мы будем расширять сферу нашей деятельности. В этом случае нужно

иметь возможность сделать так, чтобы алгоритм вычисления удерживаемого налога отражал действующее законодательство Франции, Германии или Гонконга". Если предложить, что такое "требование" вообще имеет смысл, его можно сформулировать только в виде намерений и целей; и будет сложно измерить и проверить его выполнение. Чтобы действительно повысить вероятность возможности обслуживания системы в данной ситуации, нужно потребовать использования определенных языков программирования, систем управления базами данных (СУБД), программных средств, стандартных процедур поддержки, стилей и стандартов программирования и т.д. (В этом случае, как мы увидим далее, требования, в действительности, становятся ограничениями проектирования.) Нельзя утверждать, что в результате система станет легко обслуживаемой, но, по крайней мере, мы можем приблизиться к цели.

Ограничения проектирования

Ограничения проектирования, как правило, касаются вариантов проектирования системы или процессов, используемых при ее построении. Ниже кратко описаны различные формы ограничений проектирования.

- Несколько требования, которое допускает несколько вариантов проектирования, проект является осознанным выбором среди этих вариантов. Если это возможно, хотелось бы не указывать конкретный вариант в требованиях, а оставить выбор за разработчиками, так как они смогут лучше оценить технические и экономические характеристики каждого варианта. Если мы не оставляем возможности выбора ("Использовать СУБД Oracle"), возможности проектирования сужаются, утрачивается гибкость и свобода разработки.
- Требование, налагаемое на процесс создания программы ("Программировать на VB" или "Использовать XYZ-библиотеку классов").

Как было показано в примере с Visual Basic, источники и причины таких ограничений могут быть различны, и разработчики иногда вынуждены принимать их, независимо от того, нравятся они им или нет. Но важно отделять их от обычных требований, так как подобные ограничения могут быть достаточно произвольными, они могут быть обусловлены политическими соображениями, а также могут подвергаться изменениям по мере развития технологий.

Рассмотрим определение.

Ограничения проектирования налагаются на проект системы или процессы, с помощью которых система создается. Они не влияют на внешнее поведение системы, но должны выполняться для удовлетворения технических, деловых или контрактных обязательств.

Можно указать следующие источники ограничений проектирования.

- Операционные среды: *Программы пишутся на Visual Basic.*
- Совместимость с существующими системами: *Приложение должно выполняться как на новой, так и на прежней нашей платформе.*
- Прикладные стандарты: *Использовать библиотеку классов из Developer's Library 99-724 на корпоративном сервере ИТ.*

- Корпоративные практические наработки и стандарты: *Должна обеспечиваться совместимость с существующей базой данных, Использовать стандарты программирования C++.*

Еще одним важным источником ограничений проектирования являются разнообразные инструкции и стандарты, которым подчиняется разработка проекта. Например, разработка медицинских продуктов в США регулируется множеством инструкций и стандартов FDA (Управление по санитарию и надзору за продуктами и медикаментами), которые касаются не только продукта, но и процесса его разработки и документирования. Среди организаций, инструкциям и стандартам которых должно отвечать проектирование, можно указать следующие.

- Управление по санитарию и надзору за продуктами и медикаментами (FDA)
- Федеральная правительственные комиссия США по средствам связи (FCC)
- Министерство обороны (DOD)
- Международная организация по стандартизации (ISO)
- Лаборатории по технике безопасности (UL)

Как правило, сформулированные в виде разнообразных инструкций проектные ограничения этого типа слишком длинные, чтобы их можно было непосредственно включить в требования. В большинстве случаев достаточно внести их в список ограничений проектирования в форме ссылок. Таким образом, требование может иметь вид: *Программа будет полностью соответствовать стандарту TÜV Software Standard, разделы 3.1-3.4.*

Однако при включении подобных ссылок тоже имеется определенный риск. Нужно внимательно следить за тем, чтобы включать конкретные, имеющие отношение к делу, а не общие ссылки. Например, одна ссылка вида *Продукт должен соответствовать стандарту ISO 601* фактически "связет" ваш продукт со всеми стандартами документа. Обычно следует стремиться найти "золотую середину" между чрезмерной и недостаточной конкретизацией.

Практически все проекты будут иметь те или иные ограничения проектирования. При работе с ними мы предлагаем руководствоваться следующими рекомендациями.

- Следует отличать их от других требований. Например, если программные требования обозначены ярлыком "SR" (software requirement), для ограничений проектирования можно использовать ярлык "DC" (design constraint). Можно также попытаться различать истинные ограничения проектирования и регулирующие ограничения, но мы пришли к выводу, что это редко бывает полезно и может привести к непомерным затратам на поддержку.
- Лучше включить все ограничения проектирования в специальный раздел пакета требований или использовать специальный атрибут, чтобы их можно было легко собрать вместе. Это позволит при необходимости легко находить их и пересматривать.
- Необходимо указывать источник каждого ограничения проектирования. Тогда вы сможете позже вернуться к обсуждению этого требования. "Так, это ограничение поступило от Билла из отдела маркетинга. Давайте поговорим с ним об этом!" Если имеются ссылки на некие стандарты, следует составить специальную библиографическую справку. Тогда в будущем будет проще найти данный стандарт.



- Следует документировать объяснения для каждого ограничения проектирования. Записывайте одно-два предложения, объясняющие, почему то или иное ограничение проектирования налагается на проект. Это поможет вам позднее вспомнить, что послужило мотивом для наложения конкретного ограничения. Как следует из нашего опыта, практически всегда возникает вопрос: “Почему мы наложили здесь это ограничение?”. Документирование пояснений позволяет более эффективно работать с ограничениями проектирования на более поздних фазах проекта, когда о них (неизбежно) зайдет речь.

Являются ли ограничения проектирования истинными требованиями?

Можно считать, что ограничения проектирования не являются требованиями к программному обеспечению, так как они не представляют ни один из пяти пунктов нашего сложного определения. Но когда ограничение проектирования поднимается до уровня полноправного политического, технического или бизнес-условия, оно будет удовлетворять нашему определению, как нечто, необходимое для “соответствия контракту, стандарту, спецификации или другой формальной документации”.

В таких случаях проще всего трактовать ограничение проектирования так, как любое другое требование, и удостовериться, что система проектируется и разрабатывается в соответствии с этим ограничением. Однако всегда следует стремиться к тому, чтобы таких ограничений было как можно меньше, так как их наличие может зачастую ограничить наш выбор при реализации других требований, непосредственно выполняющих потребности пользователя.

Поучительная история

Мы работали с компанией Fortune 500, хорошо известной в отрасли благодаря ее приверженности процессу и процедуре. Представьте себе наше удивление, когда мы обнаружили, что работа компании по сбору требований полностью парализована из-за того, что команда не может прийти к согласию о том, являются ли определенные требования функциональными, нефункциональными или ограничениями проектирования. В результате способность команды двигаться вперед в осуществлении проекта оказалась под вопросом из-за игры слов! Мы сказали команде, что неважно, как это называется, давайте продвигаться хоть в чем-нибудь.

Мораль такова. Назначение классификации в том, чтобы стимулировать мышление, помогать при поиске “неоткрытых руин”, и в том, чтобы помочь по-разному воспринимать эти вещи. Но в действительности классификация не имеет значения, если вы понимаете, что требования – это нечто, с чем вас или систему будут сравнивать. Предпочтительнее двигаться вперед (пусть и с несовершенной организацией), чем топтаться на месте, разрабатывая план совершенного разбиения требований по категориям.

Использование “дочерних” требований для повышения уровня конкретизации

Мы обнаружили, что многие проекты выиграют от использования концепции *дочерних требований* в качестве средства дополнения неких базовых требований. *Дочернее требование служит для повышения уровня конкретизации, выраженного в родительском требовании.*

Рассмотрим пример. В этот раз мы будем использовать в качестве иллюстрации пример из области аппаратного обеспечения. Предположим, вы разрабатываете электронный прибор, работающий от стандартной электросети. Иными словами, пользователь собирается поместить прибор в отверстие в стене. Возникает вопрос: “Как сформулировать требования, касающиеся питания данного прибора?”.

Совершенно естественным является следующее требование: “Прибор должен работать от стандартной электросети Северной Америки”. Но что это значит? Ваши инженеры забросают вас вопросами о напряжении, токе, частоте и т.д. Конечно, вы можете переписать требование так, чтобы оно содержало все необходимые детали, но вы, вероятно, обнаружите, что включение всех технических характеристик скрыло первоначальную цель требования. В конце концов, вы просто хотели, чтобы прибор работал, если его поместить в отверстие в стене!

В таком случае вы можете создать несколько требований для задания напряжения, тока, частоты и т.д. Эти требования следует рассматривать как “дочерние” определенного родительского требования. В дальнейшем мы будем часто использовать отношения “родитель-ребенок” в иерархической структуре требований. Таким образом, спецификация энергетических потребностей данного прибора будет выглядеть следующим образом.

Родительское требование. Прибор должен работать от стандартной электросети Северной Америки.



Дочернее 1. Прибор должен работать при напряжении в диапазоне xxx–yyy вольт AC.

Дочернее 2. Для нормального функционирования прибор должен потреблять не более xxx AC ампер.

Дочернее 3. Прибор должен работать так, как указано в спецификации, если входная частота варьируется в пределах xx–yy герц.

Использование дочерних требований является способом гибкого расширения и дополнения спецификации и одновременно обеспечивает контроль глубины представляемых деталей. В нашем примере естественно представить спецификацию верхнего уровня в таком виде, чтобы пользователи могли легко понять ее. В то же время разработчики могут просмотреть подробные дочерние спецификации, чтобы убедиться, что они понимают все детали реализации.

Это понятие можно использовать и в том случае, если необходима дальнейшая конкретизация. Например, легко представить себе ситуацию, когда “дочернее” требование в свою очередь становится “родительским” для следующего уровня детализации. Другими словами, можно расширять иерархию далее, до такого уровня детализации, в котором нуждается продукт.

Родительское:

Дочернее 1:

Внучатое 1:

Внучатое 2:

Но и здесь необходимо сделать некое предостережение. Хотя понятие дочерних требований чрезвычайно полезно, необходимо избегать слишком большого числа иерархических уровней детализации, просто потому, что вы запутаетесь в массе микроскопических деталей и потеряете перспективу основной цели пользователя. Как следует из нашего опыта, для большинства проектов достаточно одного подуровня детализации. В крайнем случае может оказаться полезным перейти к двум подуровням — “дочернему” и “внучатому” — но вряд ли понадобится спускаться ниже этого уровня детализации.

Организация дочерних требований

Мы обнаружили, что лучше всего не отделять дочерние требования от родительских, а включать их в главный пакет требований.

Чтобы при чтении проще было связать дочерние требования с родительским, обозначение дочерних требований должно основываться на обозначении родительских. Предположим, что требование к программному обеспечению SR63.1 (см. табл. 23.1) имеет одно или несколько дочерних требований. Естественно будет обозначить дочерние требования SR63.1.1, SR63.1.2, SR63.1.3 и т.д. Иерархический вид табл. 23.1 будет тогда выглядеть следующим образом.

Функция 63

SR63.1

SR63.1.1

SR63.1.2

SR63.1.3

SR63.2

При работе в среде программных/дочерних требований полезно иметь возможность расширять/сужать полный набор требований так, чтобы можно было рассматривать только родительские требования (отдельно) или родительские вместе с дочерними.

Далее...



После того как мы изучили природу требований, можно переходить к методам их *фиксации* и *организации*. Следующая глава будет посвящена мощному методу фиксации требований. Последующие главы посвящаются вопросам организации коллекции требований.

Глава 24

Уточнение прецедентов

Основные положения

- Для поддержки действий по проектированию и написанию кода необходимо детализировать разработанные при выявлении требований прецеденты.
- Прецеденты лучше всего использовать в тех случаях, когда система имеет много функций и должна поддерживать различные типы пользователей.
- Если у системы пользователей мало или они вовсе отсутствуют, а интерфейсы минимальны, т.е. основную массу требований составляют нефункциональные требования и ограничения проектирования, то применять прецеденты не эффективно.

В частях 1 и 2 мы начали рассматривать прецеденты как средство выражения требований к системе. В последнее время данный метод приобрел популярность.

Использование прецедентов имеет ряд преимуществ по сравнению с традиционным подходом, когда определяются отдельные декларативные программные требования.

- Прецеденты относительно легко писать.
- Прецеденты пишутся на языке пользователя.
- Прецеденты предлагают связные нити поведения (или сценарии), которые понятны как пользователю, так и разработчику.
- Благодаря описанию "нитей поведения" и содержащимся в языке UML специализированным элементам и нотациям моделирования, прецеденты обеспечивают дополнительные возможности связывать деятельность по разработке требований с проектированием и реализацией. (Мы будем более подробно обсуждать эту тему в главе 30.)
- Графическое представление прецедентов в UML и их поддержка различными инструментальными средствами моделирования обеспечивают визуализацию связей между прецедентами, что может содействовать пониманию сложной программной системы.
- Сценарий, описанный с помощью прецедента, может практически без изменений использоваться в качестве сценария тестирования во время проверки правильности.

Вопросы, на которые нужно ответить

Когда следует использовать методологию прецедентов

Для фиксации основной массы требований к системе следует использовать прецеденты в том случае, если для приложения выполнено одно или оба из приведенных ниже условий.

- Система является функционально ориентированной; существует несколько различных типов пользователей и функционального поведения. Так как прецеденты отдельно описывают поведение системы для *каждого типа пользователя*, их применение наиболее эффективно, когда существует много типов пользователей системы и система должна предоставлять различные наборы функций каждому из них.
- Команда реализует систему, используя UML и объектно-ориентированные (ОО) методы. Определенные ОО-понятия (такие, как унаследованное поведение акторов и прецедентов, абстрактные акторы) хорошо приспособлены к методу прецедентов и создают дополнительные удобства для аналитика или разработчика моделей. UML-нотация прецедентов также поддерживает визуальное моделирование системы и обеспечивает парадигму моделирования, которая поддерживает представление требуемого поведения системы (прецедент), а также того, как это поведение реализуется в программе (посредством реализаций прецедентов).

Когда прецеденты не являются наилучшим вариантом

Однако прецеденты подходят не для всех типов систем и требований. Особенно это касается рассматриваемых ниже разновидностей систем. Работая над такой системой, необходимо дополнить модель прецедентов или даже вовсе от нее отказаться.

Системы, где пользователей мало или нет вообще, а интерфейсы минимальны. Существует много классов функционально наполненных систем, которые имеют мало внешних интерфейсов и пользователей. Примерами могут служить системы, проектируемые для проведения научных вычислений или имитаций, встроенные системы, системы управления процессами, система проверки наличия вирусов, работающая без вмешательства оператора, а также различные служебные программы, такие как компиляторы и программы управления памятью. Хотя и здесь можно применять прецеденты и, возможно, они будут полезны в качестве дополнения к традиционному подходу, в данном случае существуют более простые способы выразить большую часть требований.

Системы, в которых доминируют нефункциональные требования и ограничения проектирования. Как уже отмечалось ранее, прецеденты не предназначены для представления нефункциональных требований – атрибутов системы и ее среды, специальных требований и ограничений проектирования. Для включения требований данного типа в каждом прецеденте есть раздел “специальные требования”. Это работает удовлетворительно, если подобные требования применяются к одному или нескольким прецедентам. Но в общем случае не все такие требования удается связать с конкретным прецедентом.

Глобальные нефункциональные требования вообще не удается удовлетворительно представить с помощью прецедентов. Это относится к требованиям соответствия зако-

нодательству и инструкциям, операционным средам и стандартам разработки программ. (Например, в Rational одна спецификация используется исключительно для задания требований глобализации программных продуктов. Эти требования практически полностью состоят из ограничений, которыми следует руководствоваться при проектировании программного обеспечения, чтобы сделать возможным и относительно дешевым его перевод на другие языки. Прецеденты нужны только для описания отдельных вариантов предполагаемого использования, таких как "Человек, говорящий по-французски, использует немецкий вариант ОС".)

Проблема избыточности

Многие прецеденты весьма похожи, но в то же время и достаточно различны, чтобы требовать отдельного описания. Это приводит к значительной избыточности описания, что увеличивает объем документации требований. Кроме того, возникают проблемы обслуживания, если необходимо изменить общее поведение, выраженное во многих прецедентах. В этом случае для уменьшения избыточности можно использовать такие отношения прецедентов, как генерализация, отношения включения и наследования (Буч (Booch), 1990).

Однако использование этих отношений само по себе создает дополнительные сложности и может оказаться неэффективным, если поведение можно описать другими способами. Действительно, некоторые примеры достаточно сложного поведения проще описать на естественном языке (например, "если система находится в состоянии готовности и каждый из двух офицеров нажимает на кнопку запуска и удерживает ее в этом положении более 1 секунды, произойдет запуск ракеты"). Конечно, можно попытаться и в этих случаях применить прецеденты, но задача состоит в том, чтобы выбрать наиболее подходящий метод для существующих обстоятельств, который обеспечит простое представление и возможность понимания, а не в том, чтобы использовать некий метод потому, что вы думаете, что так надо. В большинстве проектов для создания оптимального подхода можно использовать прецеденты совместно с традиционными методами.

Совершенствование спецификаций прецедентов

В данной главе мы продолжим изучение прецедентов, которые начали рассматривать в главах 2 и 18, и используем их для уточнения спецификации системы. Это удобно, так как можно вернуться к прецедентам, разработанным на более ранних этапах, и описать их более детально. В зависимости от достигнутого в процессе их выявления уровня конкретизации, разработанные ранее прецеденты могут быть уже достаточно подробными для проведения проектирования и реализации. Однако более вероятно (и мы рекомендуем поступать именно так), что на стадии выявления требований был сохранен достаточно высокий уровень абстракции, чтобы не запутаться в деталях. Может оказаться, что определены еще не все необходимые прецеденты, исключительные условия, условия состояния и другие специальные условия, которые менее интересны для пользователя, но могут заметно повлиять на проектирование системы. Теперь пришло время обеспечить этот дополнительный уровень конкретизации.

Замечание. В задачу данной книги не входит изложение полного курса по использованию прецедентов. Если вы заинтересованы в более полном овладении данной методологией и сопутствующими технологиями, рекомендуем обратиться к двум хорошим книгам по данной теме: Шнейдер и Уинтерс (Schneider, Winters, 1998), а также Джейкобсон (Jacobson, 1999). Тем не менее, мы рассмотрим некоторые основополагающие принципы методологии прецедентов.

Для того чтобы конкретизировать прецеденты, нам нужно использовать более строгий подход, чтобы лучше понять некоторые нюансы. Рассмотрим еще раз определение прецедентов, обращая основное внимание на то, что говорится о них в UML.

Прецедент – это описание множества действий (включая варианты), которые система выполняет для того, чтобы доставить полезный осозаемый результат определенному актору (Буч (Booch), 1999).

Вот это да! Выглядит так, как будто это определение составлялось на собрании адвокатов!¹ Как мы уже отмечали ранее, методология прецедентов выделяет два элемента, которые должны присутствовать во всех реализациях прецедентов.

1. **Прецедент.** В UML прецедент изображается в виде овала. Несмотря на то что прецедент является текстовым описанием, данная шиктограмма служит вспомогательным средством, помогающим визуально моделировать систему и отображать взаимодействия между прецедентами и другими элементами модели.



Прецедент

2. **Акторы.** Актор – это некто или нечто, взаимодействующее с нашей системой. Существует три типа акторов: пользователи ("техник Билл"), приборы ("контроллер двигателя руки робота") и другие системы ("контроллер ЦБУ системы НО-LIS"). Акторы не являются частью описываемой системы, а находятся за ее пределами.



Актор

Рассмотрим некоторые другие ключевые фразы UML-определения: "Прецедент – это описание множества действий (включая варианты), которые выполняет система, чтобы доставить полезный осозаемый результат определенному актору".

- **Варианты.** Прецедент описывает основной поток событий (или нить), а также варианты (или альтернативные потоки).
- **Множество действий.** Описывает выполняемую функцию или алгоритмическую процедуру, которая приводит к результату. Когда актор инициирует прецедент, предлагая системе некий ввод, происходит вызов этого множества. Отдельное действие атомарно, т.е. оно либо выполняется целиком, либо не выполняется вовсе. Требование атомарности строго обязательно при выборе уровня детализации прецедента. Следует изучить предлагаемый Прецедент и, если некое действие не является атомарным, обеспечить дальнейшую детализацию.

¹ В действительности это было собрание специалистов по методологии. Айвар Джейкобсон (Ivar Jacobson) как-то рассказал анекдот: В чем разница между специалистом по методологии и террористом? С террористом можно договориться.

- **Система выполняет.** Это означает, что система обеспечивает описанные в прецеденте функциональные возможности. Это то, что делает система в зависимости от полученного ввода.
- **Полезный осозаемый результат.** Важно отметить, что результат прецедента должен быть полезен пользователю, т.е. "жилец нажимает кнопку выключателя" не является правильным прецедентом; (система ничего не делает для пользователя). Но "жилец нажимает кнопку выключателя, и система включает свет" является осмысленным прецедентом.
- **Определенный актор.** Это человек или прибор (жилец по имени Линда или сигнал от кнопки нештатной ситуации), инициирующий данное действие (переключение света или активацию системы безопасности).

Эволюция прецедентов

На ранних итерациях в части 3, "Определение системы", было выявлено большинство наиболее важных прецедентов. Однако описывались только немногие из них — те, которые считались архитектурно значимыми или особенно хорошо описывали поведение системы. Как правило, описания этих прецедентов выполнены в виде дополнения к документу-концепции, в котором описывается, как предполагается использовать содержащиеся в документе функции.

В процессе уточнения завершается разработка всех прецедентов, необходимых для определения системы. Показателем того, что прецедентов "достаточно", является то, что полный набор прецедентов описывает все возможные способы использования системы на уровне конкретизации, пригодном для проектирования, реализации и тестирования.

Следует отметить, что детализация прецедента *не является* декомпозицией системы. Другими словами, мы не начинаем с прецедента высокого уровня, чтобы затем разбивать его на все большее число прецедентов. Вместо этого мы стараемся более точно описать взаимодействия акторов с системой. Таким образом, разработка прецедентов более похожа на уточнение последовательностей действий, а не на иерархическое разбиение их на более мелкие действия. В модели часто есть прецеденты, которые настолько просты, что не нуждаются в детальном описании потока событий; достаточно краткого описания. Критерием для принятия такого решения является то, что пользователи понимают, что означает прецедент, а для разработчиков и тестологов данный уровень детализации удобен.

Какие действия включить в прецедент

Часто трудно решить, является ли множество взаимодействий пользователя с системой (или диалог) одним прецедентом или несколькими. Рассмотрим использование машины, принимающей банки и бутылки. Клиент загружает в нее консервные банки и бутылки, нажимает кнопку и получает квитанцию, по которой может затем получить деньги.

Является ли загрузка сдаваемых предметов одним прецедентом, а получение квитанции — другим? Или это все один прецедент? Имеют место два действия, но в отдельности они не приносят пользы клиенту. Именно полный диалог, со всеми загрузками и получением квитанции, имеет смысл для клиента. Следовательно, именно полный диалог — от загрузки первого сдаваемого предмета до нажатия кнопки и получения квитанции — является полным вариантом использования, т.е. прецедентом.

Кроме того, хотелось бы хранить эти два действия вместе, чтобы иметь возможность одновременно их просматривать, модифицировать, вместе проверять, изменять, когда это необходимо, писать характеризующую их пользовательскую документацию и обращаться с ними, как с единым целым. Это становится особенно важным в более крупных системах.

Рабочий пример. Строение простого прецедента

Рассмотрим шаг за шагом процедуру определения прецедента. Будем использовать простой пример из системы HOLIS: жилец включает освещение в комнате, используя автоматическую систему управления освещением HOLIS.

Определение акторов

Первым делом необходимо точно установить, кто взаимодействует с прецедентом. Во многих системах, спроектированных для пользователей, следует сначала выявить людей, которые будут использовать систему. В нашем случае жилец взаимодействует с системой, чтобы управлять освещением в комнате. Таким образом, выявлен единственный актор, пользователь (Жилец), нажимающий кнопку пульта.



Предостережение. При определении акторов полезно вести "именной список", чтобы видеть, какие из них уже определены, и случайно не создать некий актор повторно под другим именем.

Дать название прецеденту

Каждый прецедент должен иметь имя, показывающее, что достигается при его взаимодействии с актором(ами). Имя может состоять из нескольких слов, проясняющих его смысл. Различные прецеденты не должны иметь одно и то же имя.

К заданию имени следует подходить ответственно. Оно должно быть уникальным и в то же время таким, чтобы его можно было легко отличить от имен других прецедентов данного проекта. В английском языке имена прецедентов часто начинаются с глагола, обозначающего действие, которое отражает предназначение данного прецедента. Назовем наш прецедент "Управление освещением".



Управление освещением

Можно создавать структуру имен формальным методом, чтобы сгруппировать аналогичные прецеденты в аналогично именованные группы. Можно также ввести порядковый номер или другой уникальный идентификатор в имя прецедента, чтобы было удобнее работать со списком прецедентов. Например, разработчик может указать имя данного прецедента как "031 Управление освещением". Однако опыт свидетельствует, что простого именования прецедентов и, возможно, применения некоторых вспомогательных программ, позволяющих нам их находить, сортировать и анализировать, обычно вполне достаточно.

Составление краткого описания

Краткое описание прецедента должно отражать его роль и цель. При его написании следует рассмотреть, какие акторы участвуют в нем, а также обратиться к глоссарию. Если необходимо, можно определить новые понятия.

Это описание должно служить своего рода неформальным обзором функциональных возможностей. Их полное описание содержится в следующем разделе прецедента "Поток событий". Описание прецедента предназначено для получения "общего впечатления" и ничего более. В нашем случае можно описать прецедент следующим образом.

Описание прецедента "Управление освещением"

Данный прецедент определяет способ включения и выключения света, а также изменения его яркости, в зависимости от того, как долго пользователь удерживает кнопку пульта в нажатом состоянии.

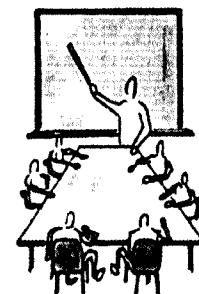
Определение потока событий

Сердцевиной прецедента является поток событий. Как правило, это текстовое описание операций актора и различных ответов системы. Поток событий описывает, что, как предполагается, будет делать система в зависимости от поведения актора. Между прочим, не обязательно описывать поток в текстовой форме. Для этого можно использовать диаграммы взаимодействия UML, а также другие формальные методы, обсуждаемые в главе 28. Все они вполне пригодны для документации прецедентов. Главное — обеспечить *понимание*, и не существует единственного подхода на все случаи жизни. Однако, как правило, вполне подходит описание на естественном языке.

Поток событий описывает достижение цели прецедента и предназначен для рассмотрения следующими заинтересованными лицами.

- Клиентами, которые одобряют результат и "благословляют" функции
- Пользователями, которые будут работать с системой
- Разработчиками прецедентов, которые заинтересованы в точном описании желаемого поведения системы
- Рецензентами, которые составляют непредвзятое мнение о системе
- Проектировщиками, которые анализируют прецеденты в поисках классов, объектов и т.п.
- Тестологами, которым нужно создать тестовые примеры
- Менеджером проекта, которому необходимо понимать весь проект в целом
- Составителем технической документации, которому нужно документировать функции системы так, чтобы было понятно пользователю
- Людьми, из отдела маркетинга и продаж, которым необходимо понимать функции продукта и объяснять его достоинства остальным

Вы, возможно, думаете, что *практически не бывает ситуаций*, когда можно описать *простой поток событий*, который работает во всех случаях; всегда нужен способ для описания некоторых *альтернативных потоков*. Ничего страшного. Определение потока событий



прецедента допускает альтернативные потоки. Но сначала создадим основной поток для нашего примера.

Основной поток для precedента “Управление освещением”. Отметим, что данный поток событий нигде не указывает, как система выполняет то или иное действие. Он указывает только, что происходит.

Основной поток начинается, когда Жилец нажимает любую кнопку пульта. Если Жилец отпускает кнопку в течение периода времени, отсчитываемого таймером, система “переключает” состояние освещения.

- Если освещение было включено, оно полностью выключается.
- Если свет был выключен, освещение включается с тем же уровнем яркости, который был перед последним выключением.

Конец основного потока.

Альтернативный поток событий. Во многих случаях precedент может иметь различные потоки, в зависимости от возникающих условий. Иногда эти потоки связаны с выявленными в процессе обработки ошибочными условиями, в других случаях они могут описывать дополнительные способы обработки конкретных условий. Например, precedент, печатающий квитанцию для операции с кредитной карточкой, может обнаружить, что у принтера закончилась бумага. Этот специальный случай будет описан в precedенте как альтернативный поток событий. При записи альтернативных потоков не забывайте документировать условия, приводящие к ним! На количество альтернативных потоков не налагаются ограничения, поэтому будьте внимательны и документируйте все альтернативные потоки, в том числе все возможные условия возникновения ошибок.

В нашем примере альтернативный поток событий возникает, когда Жилец удерживает кнопку пульта в нажатом состоянии дольше одной секунды. Итак, нам нужно добавить в precedент альтернативный поток.

Альтернативный поток событий: постепенное изменение яркости

Если Жилец удерживает кнопку пульта в нажатом состоянии дольше одной секунды, система инициирует действия по изменению яркости для данной кнопки пульта.

Пока Жилец продолжает удерживать кнопку, происходит следующее.

1. Яркость контролируемого источника постепенно повышается до максимального значения со скоростью 10 процентов в секунду.
2. Когда достигнуто максимальное значение, яркость контролируемого источника постепенно понижается до минимального уровня со скоростью 10 процентов в секунду.
3. Когда достигнуто минимальное значение, процесс продолжается с шага 1.

Когда Жилец отпускает кнопку, происходит следующее.

4. Система прекращает изменять яркость освещения.

Выявление пред- и постусловий

Иногда необходимо выявить предусловия, которые влияют на поведение системы, описанное прецедентом, а также описать постусловия, такие как состояние системы и перманентные данные, полученные после завершения прецедента. Но использовать пред- и постусловия нужно только тогда, когда необходимо прояснить поведение, выраженное прецедентом.

Важно проводить различие между событиями, которые запускают потоки прецедента, и предусловиями, которые должны быть выполнены до того, как можно будет инициировать прецедент. Например, предусловием для прецедента "Управление освещением" является то, что домовладелец (Жилец) должен обеспечить наличие определенного набора осветительных приборов, способных изменять яркость. Еще одним предусловием является то, что выбранная кнопка пульта должна быть запрограммирована для управления этим набором. (Предполагается, что другие прецеденты описывают, как осуществляются эти предусловия.) Итак, нам нужно сформулировать предусловия.

Предусловия для прецедента "Управление освещением"

- Для выбранной кнопки пульта должен быть предусмотрен режим "Изменение яркости".
- Выбранная кнопка пульта должна быть предварительно запрограммирована для управления неким набором осветительных приборов.

Часто необходимо выявлять и включать в документацию постусловия. Они позволяют точно указывать состояние, которое должно быть истинным по окончании прецедента, даже если использовались альтернативные пути.

Чтобы яркость была такой, как нужно, когда Жилец включает свет в следующий раз, система должна "помнить" уровень яркости, который был установлен для выбранной кнопки пульта после действий по изменению яркости. Следовательно, это постуслжение, которое мы должны записать для данного прецедента.

Постусловия для прецедента "Управление освещением"

- После окончания данного прецедента запоминается текущий уровень яркости для выбранной кнопки пульта.

Теперь соберем все это вместе. В табл. 24.1 содержится все, что получится после того, как мы объединим все важные "кусочки" нашего прецедента. (Хотя для прецедента можно определить еще много других элементов, они не так важны для нас на данном этапе.) Этот прецедент документирован в повествовательном стиле, его можно найти среди артефактов системы HOLIS в приложении А.

Таблица 24.1. Определение прецедента

Элемент	Значение
<i>Название прецедента</i>	<i>Управление освещением</i>
<i>Краткое описание</i>	Данный прецедент определяет способ включения и выключения света, а также изменения его яркости в зависимости от того, как долго пользователь удерживает кнопку пульта в нажатом состоянии

Окончание табл. 24.1

Элемент	Значение
Поток событий	<p>Основной поток событий начинается, когда Жилемц нажимает кнопку пульта ("Управление включением").</p> <p>Если Жилемц отпускает кнопку в течение периода времени, отсчитываемого таймером, система "переключает" состояние освещения. Это означает следующее.</p> <ul style="list-style-type: none"> ■ Если свет был включен, он полностью выключается ■ Если свет был выключен, он включается с тем же уровнем яркости, который был перед последним выключением
Альтернативный поток событий	<p>Если Жилемц удерживает кнопку пульта в нажатом состоянии более одной секунды, система инициирует действия по изменению яркости для указанной кнопки.</p> <p>Пока Жилемц продолжает удерживать кнопку в нажатом состоянии, производятся следующие действия.</p> <ol style="list-style-type: none"> 1. Яркость контролируемого источника постепенно повышается до максимального значения со скоростью 10 процентов в секунду 2. Когда достигнуто максимальное значение, яркость контролируемого источника постепенно снижается до минимального уровня со скоростью 10 процентов в секунду 3. Когда достигнуто минимальное значение, процесс продолжается с шага 1 <p>Когда Жилемц отпускает кнопку пульта происходит следующее.</p> <ol style="list-style-type: none"> 4. Система прекращает изменять яркость источника
Предусловия	<ul style="list-style-type: none"> ■ Выбранная кнопка пульта должна иметь режим, допускающий изменение яркости ■ Выбранная кнопка должна быть предварительно запрограммирована для управления неким набором осветительных приборов
Постусловия	По окончании прецедента установленная яркость запоминается системой
Специальные требования	Минимальный уровень освещения не должен быть равен 0. Он должен быть не очень ярким, соответствующим уровню ночного освещения

Далее...

После того как все прецеденты выявлены и описаны приблизительно на таком уровне детализации, процесс уточнения той части системы, которую мы решили описывать с помощью прецедентов, заканчивается. В следующей главе мы рассмотрим создание спецификаций.

Глава 25

Спецификация требований к программному обеспечению (Modern Software Requirements Specification)

Основные положения

- Пакет спецификаций требований к программному обеспечению (Modern SRS Package) представляет собой набор артефактов, полностью описывающих внешнее поведение системы. Он создает концептуальную модель создаваемой системы.
- Документ-концепция (Vision document) служит исходной информацией для создания Modern SRS Package. Он определяет потребности пользователей, цели, задачи, целевые рынки и функции системы, в то время как в пакете Modern SRS Package основное внимание уделяется деталям реализации этих функций.
- Сбалансированный подход, как правило, заключается в совместном использовании модели прецедентов и традиционной спецификации требований.

После уточнения понимания системы настало время разработать стратегию организации и документирования требований. Хотя большая часть усилий в этом процессе действительно направлена на организацию выявленных и детализированных требований к программному обеспечению, документов, прецедентов и моделей, самым важным является то, что *совокупность данных артефактов представляет полную концептуальную модель создаваемой системы*.

Говоря коротко, для начала у нас есть части относительно полной концептуальной структуры, с помощью которой мы можем рассуждать о будущей системе. Скорее всего, ее еще нельзя потрогать (хотя, возможно, мы уже видели некоторые раскадровки и прототипы), но уже можно начать ее анализировать и проверять наше понимание системы, несмотря на то что она до сих пор существует только на бумаге и в виде моделей. Можно сказать, что мы подошли к важному рубежу: созданию концептуальной модели или *посредника (proxy) системы*, которую надо построить.

Пакет спецификаций требований к программному обеспечению (Modern SRS Package)

Исторически сложилось, что наиболее популярный метод документирования требований состоял в упорядоченной их записи на естественном языке. Этот метод пересматривался и совершенствовался в ходе выполнения множества проектов, и постепенно был разработан ряд стандартов для подобных документов, в том числе стандарт IEEE (Institute of Electrical and Electronics Engineers) 830: Standard for Software Requirements Specification (1994).

Но сегодня при наличии современных инструментальных средств и методов мы предпочитаем представлять спецификацию программных требований (SRS) в виде логической структуры, а не физического документа. Различные детально описанные требования к системе заключаются в пакет, который мы называем современным пакетом спецификаций требований к программному обеспечению (Modern Software Requirements Specification Package), в отличие от более ранних форм, которые именуются просто SRS. Пакет Modern SRS Package связан с документом-концепцией, который служит исходной информацией для его создания.

Однако эти два артефакта служат разным целям и, как правило, пишутся разными авторами. На данной стадии проекта акцент смещается от общей формулировки потребностей пользователя, целей, задач, целевых рынков и функций системы к детальному описанию того, как эти функции предполагается реализовывать в решении. На рис. 25.1 схематически представлены элементы пакета.

На данном этапе нам нужен пакет информации, полностью описывающей *внешнее* поведение системы, т.е. набор артефактов следующего содержания: “Вот то, что система должна делать, чтобы предоставить эти функции”. Это и есть Modern SRS Package.

Нет веских причин уделять внимание различиям между используемыми инструментальными средствами. В конце концов, мы собираем требования и должны обращать внимание на эффективность их сбора и организации безотносительно к тому, какие средства используются. Итак, мы будем предполагать, что набор требований образует *пакет* информации. Поэтому на рис. 25.1 изображены не только элементы этого пакета, но и их связи.

Пакет Modern SRS Package – это не том замороженной информации, созданный в соответствии с требованиями ISO 9000, который затем положат на полку при продолжении проекта. Это активный “живой” пакет, во многих случаях играющий чрезвычайно важную роль при переходе к реализации.

- Он служит основой для общения между всеми сторонами-участниками: между самими разработчиками, между разработчиками и внешними группами, пользователями и другими заинтересованными лицами.
- Формально или неформально, он представляет соглашение между этими различными сторонами: если чего-то нет в пакете, разработчики не должны над этим работать. А если это в пакете есть, они отвечают за то, чтобы предоставить данную функциональную возможность.

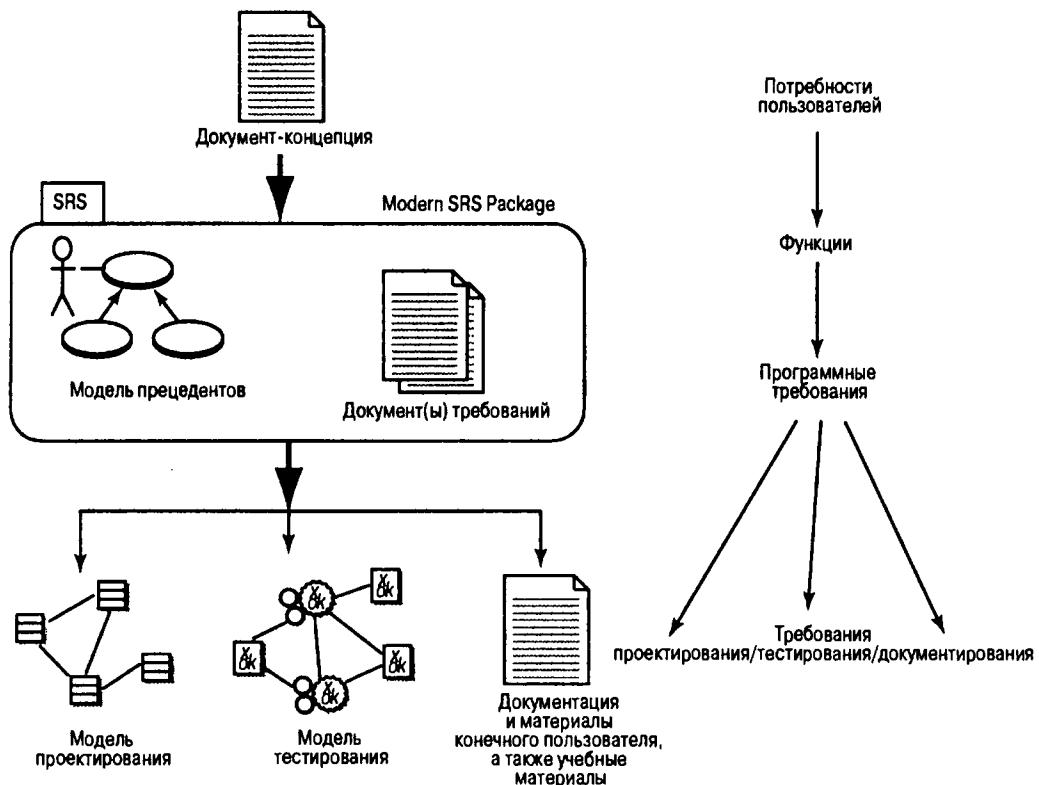


Рис. 25.1. Элементы Modern SRS Package

- Он служит в качестве справочника стандартов для администратора программного изделия. Скорее всего, у него не хватит времени, чтобы читать созданный разработчиками код и сравнивать его непосредственно с документом-концепцией; он должен использовать данный пакет в качестве стандарта при обсуждении вопросов с командой разработчиков.
- Как уже отмечалось ранее, пакет служит исходной информацией для групп, выполняющих проектирование и реализацию. В зависимости от того, как организован проект, люди, занимающиеся реализацией, могли ранее привлекаться к действиям по анализу проблемы и определению функций, что привело к созданию документа-концепции. Но им нужен именно Modern SRS Package, чтобы принять решение о том, что должен делать их код.
- Modern SRS Package служит исходной информацией также для групп, выполняющих тестирование и гарантирующих качество (QA). Эти группы также следует привлекать к определенным действиям на более ранних этапах, и им, безусловно, нужно понимать концепцию, лежащую в основе Vision-документа. Но их основная задача – создать тестовые примеры и QA-процедуры, чтобы гарантировать, что разработанная система действительно выполняет отраженные в Modern SRS Package требования. Пакет служит в качестве стандарта при планировании и выполнении тестов.

- Modern SRS Package управляет развитием системы на фазе построения; когда функции документа-концепции модифицируются или когда в него добавляются новые функции, они детализируются в данном пакете.

Кто отвечает за Modern SRS Package

На самом деле не так уж важно, кто пишет пакет. Гораздо важнее, чтобы Modern SRS Package существовал и являлся основой для предстоящих действий по построению и тестированию.

Возникает естественный вопрос: *кто отвечает за создание и поддержку компонентов Modern SRS Package?* Обычно члены команды разработчиков самостоятельно берут на себя эту задачу. Команда разработчиков кровно заинтересована в полном понимании пакета и всех его требований и, владея им, может соответствующим образом влиять на многие системные решения. В конечном итоге, кто может лучше написать требования к программному обеспечению, чем люди, которые затем будут отвечать за соответствие им? Часто за эту задачу (в качестве детальной доработки документа-концепции) берется системный аналитик, в других случаях тестологи работают рука об руку с командой проекта и берут на себя ответственность за требования.

Каждый подход имеет свои плюсы и минусы, и каждая команда сама будет решать, какая стратегия наилучшая. Из опыта следует, что если к пакету относиться серьезно, не так важно, кто его написал, хотя и есть некоторая разница в том, отвечает ли за него руководство или команда. Действительно важно то, что Modern SRS Package существует и является основой для дальнейших действий по разработке и тестированию.

Конечно, авторы SRS не пишут требования в вакууме. Мы обнаружили, что пересмотр данного пакета – наиболее эффективный шаг, позволяющий убедиться, что разработчики, маркетологи, пользователи и другие участники “поют под одну и ту же музыку”. Modern SRS Package – “живой” артефакт, который нуждается в обновлениях по мере того, как развивается проект и становится более понятными различные функции. *Никогда не следует допускать*, чтобы написанный пакет впоследствии игнорировался.

Организация пакета Modern SRS Package

Для большой системы пакет Modern SRS Package может быть весьма объемным. Он может состоять из сотен страниц текста и диаграмм precedентов, а также содержать множество детальной информации, которой разработчики должны уделить пристальное внимание. Но если пакет правильно написан и хорошо организован, его размеры не будут помехой при использовании; они только будут свидетельствовать об относительной сложности создаваемой системы.



Итак, возникает вопрос: *как следует организовать пакет?* Например, если в команду приходит новый разработчик и получает задание работать над функцией XYZ, где ему искать детальное описание этой функции? Или предположим, что администратор проекта уходит в самый разгар работы; как новый администратор может быстро войти в курс дела и вникнуть в детали управления текущей деятельностью команды проекта?

Понятно, что структура пакета должна отвечать природе приложения и организации. Пакет для разработки текстового процессора в компании по производству программного обеспечения в Силиконовой долине, вероятно, будет несколько отличаться от аналогичного пакета для системы управления авиаполетами. Нас волнует не то, сможет ли эксперт по управлению полетами посетить программистскую фирму в Силиконовой долине и объяснить, что в действительности означают его SRS. Нас заботит, чтобы разработчики и пользователи внутри конкретной организации могли объяснить смысл созданных SRS, чтобы их пакет оставался актуальным на протяжении всей разработки, а также при сопровождении системы после того, как она поступит в производство.

Если организация желает при аттестации достичь более высокого уровня по шкале SEI-CMM или получить сертификат ISO 9000, она более заинтересована в стандартизации организации и формата своего пакета. Даже без оглядки на CMM и ISO, ранее приведенные примеры иллюстрируют выгоды стандартизации: возможность быстрее вникнуть, когда имеется текучесть кадров или в команду приходит новый человек. Она также позволяет гарантировать, что важнейшая информация не потеряется и все будут знать, где ее искать.

Следует помнить, что пакет Modern SRS Package не предназначен для того, чтобы его читали как роман от начала до конца. Это скорее справочник, и каждый разработчик будет, как правило, просматривать только необходимые ему разделы. Таким образом, нужно так организовать SRS, чтобы можно было легко и просто находить нужную информацию.

Пакет должен подчиняться некой организационной концепции. Мы пришли к выводу, что следующая организационная схема неплохо подходит практически для всех типов проектов. Эта схема вместе с комментариями, поясняющими некоторые детали структуры, приводится в приложении В.

Общая информация

История пересмотров

Содержание

1. Введение
 - 1.1. Цель
 - 1.2. Масштаб
 - 1.3. Ссылки
 - 1.4. Предположения и зависимости
2. Краткая характеристика модели precedентов
3. Краткая характеристика акторов
4. Требования
 - 4.1. Функциональные требования
 - 4.2. Нефункциональные требования
 - 4.2.1. Практичность
 - 4.2.2. Надежность
 - 4.2.3. Производительность
 - 4.2.4. Возможность сопровождения
5. Требования к интерактивной пользовательской документации и системе подсказок
6. Ограничения проектирования
7. Закупаемые компоненты
8. Интерфейсы
 - 8.1. Пользовательские интерфейсы

- 8.2. Аппаратные интерфейсы
- 8.3. Программные интерфейсы
- 8.4. Коммуникационные интерфейсы
- 9. Требования лицензирования
- 10. Примечания об авторских правах и т.п.
- 11. Применяемые стандарты

Индекс

Глоссарий

Приложение. Спецификации прецедентов

История пересмотров прецедента

Дата	Версия	Описание	Автор

Название прецедента

Краткое описание

Потоки событий

Основной поток

Альтернативные потоки

Первый альтернативный поток

Второй альтернативный поток

Специальные требования

Первое специальное требование

Второе специальное требование

Предусловия

Предусловие 1

Постусловия

Постусловие 1

Точки расширения

Название точки расширения

Другие материалы прецедента

Документирование функциональных требований

Итак, рассмотрим, чего мы достигли.

- Мы пришли к решению, что документацию системных требований следует организовать в виде *пакета* артефактов.
- Эти артефакты могут состоять из текстовых документов, таблиц, схем, прецедентов и других элементов, помогающих разработчикам понять, что хочет клиент.
- Мы создали шаблон структуры пакета *Modern SRS Package*. Данный шаблон позволяет организовать и распределить по категориям различные элементы, но основное внимание в нем уделяется *текстовым* элементам (таким, как спецификации на

естественном языке отдельных требований) и элементам в виде *прецедентов* (представление которых включает в себя графическую модель прецедентов и их текстовые описания).

Для сбора требований можно применять как традиционные методы спецификации требований, так и метод прецедентов. Каждый из этих методов сам по себе использовался в тысячах успешных проектов. Но как оптимизировать нашу работу? Какой метод следует применить в каждом конкретном случае?

Что касается спецификации функциональных требований к системе, мы пришли к выводу, что наилучшим подходом является совместное использование моделирования прецедентов и традиционных спецификаций требований. Рассмотрим “баланс требований”, представленный на рис. 25.2.

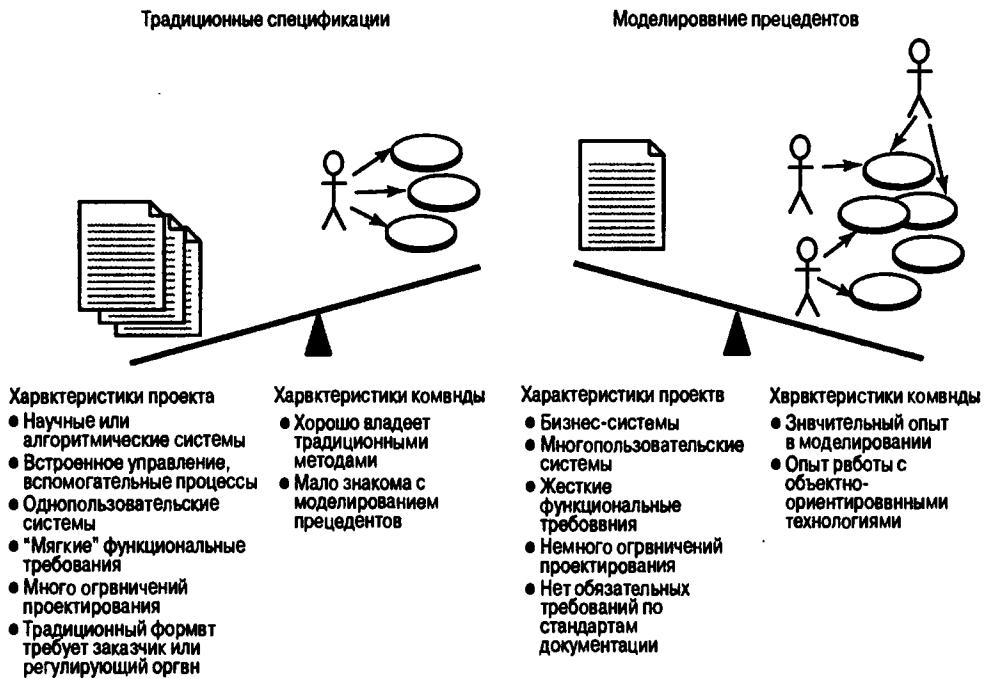


Рис. 25.2. Баланс требований

Отметим, что существует *множество* возможных вариантов. Некоторые проекты характеризуются незначительным числом (или полным отсутствием) спецификаций, которые можно сформулировать с помощью прецедентов. Эти проекты отличаются высоким удельным весом вычислений и алгоритмов; примерами могут служить системы расчета прогноза погоды или научные системы. Еще одной крайностью являются проекты разработки систем, предназначенных для удовлетворения функциональных потребностей пользователей. Такие системы обычно делают уйму *вещей* для пользователя. Кроме того, система может обслуживать *различные типы пользователей*, для описания чего идеально подходит метод прецедентов.

Необходимо также учитывать, какими профессиональными навыками обладает команда проекта. Некоторые команды не имеют опыта (или имеют незначительный опыт) в использовании прецедентов и объектно-ориентированных методов. Другие команды владеют этими методами в совершенстве.

Чтобы использовать представленный на рис. 25.2 "вид с точки зрения баланса", нужно просто заполнить окна флагжков, соответствующие характеристикам вашего проекта и навыкам вашей команды. В зависимости от того, в какую сторону склонится чаша весов, можно планировать, какой метод документирования требований выбрать.

В любом случае Modern SRS Package позволяет сочетать лучшие качества моделирования прецедентов и традиционных методов спецификации требований. Задача состоит в том, чтобы найти правильное их сочетание для вашего проекта и команды. Как следует из нашего опыта, необходимо внимательно рассмотреть следующие моменты.

- Возможности организации
- Общие тенденции программных процессов и методов, принятых в организации
- Внешние факторы, такие как требования инструкций и другие ограничения
- Конкретное содержание проекта

Только после этого можно выбрать определенный метод или комбинацию методов. Разобравшись в этих факторах, команда сможет выбрать верный работоспособный баланс, а затем двигаться к наиболее эффективной комбинации, если это возможно.

Далее...



Пакет Modern SRS Package является мощным средством отражения потребностей проекта. Однако написать такой пакет непросто. Как и всему остальному, написанию хороших спецификаций требований к программному обеспечению нужно учиться. В следующей главе рассматриваются некоторые проблемы, с которыми приходится сталкиваться, когда мы стремимся написать ясный, не допускающий неоднозначных толкований набор спецификаций.

Глава 26

Неоднозначность и уровень конкретизации

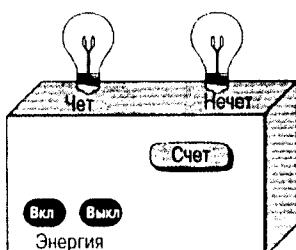
Основные положения

- “Золотая середина” в требованиях достигается, когда удается добиться наибольшей понятности при наименьшей неоднозначности.
- Способность найти золотую середину зависит от навыков членов команды, содержания приложения и необходимого уровня гарантий, что система работает так, как требуется.
- Если необходимо исключить возможность неправильного понимания требований, может понадобиться применить более формальные методы их задания.

Нахождение “золотой середины”

В процессе определения требований достаточно сложно сделать требования настолько детализированными, чтобы их можно было хорошо понять, и при этом избежать чрезмерных ограничений на систему и предопределения всего того, что лучше оставить для других потоков процесса. (“Действительно ли нам нужно задавать фоновый цвет Pantone 287 в нашей спецификации GIL? Нет? Как вам прагается цвет, который был в прошлый раз?”)

Каждый раз слушатели задают нам один и тот же вопрос, который вызывает у них наибольшее затруднение: *каким должен быть уровень конкретизации формулировки требований, чтобы исключить возможность неправильного понимания?* Они надеются услышать простой ответ, но, к сожалению, его не существует. Единственное, что можно сказать: “когда как”. Рассмотрим в качестве примера упражнение, в котором предлагается написать требования для “лампового ящика” (рис. 26.1).



Функции

- Управляется микропроцессором
- Отслеживает, четное или нечетное число раз была нажата кнопка счета
- Детектор перегорания лампы заставляет мигать оставшуюся лампу

Рис. 26.1. Ламповый ящик

Задача упражнения состоит в том, чтобы с помощью естественного языка или метода прецедентов написать требования, позволяющие описать поведение данного прибора. В упражнении можно проводить интервью с пользователем, так что тот, кто пишет требования, может уточнять спецификацию на основании прямых указаний пользователя. В качестве примера достаточно удачной попытки на естественном языке рассмотрим следующую спецификацию требований (Дэвис (Davis), 1993).

После нажатия кнопки *Вкл* до нажатия кнопки *Выкл* система является "включенной".

После нажатия кнопки *Выкл* до нажатия кнопки *Вкл* система является "выключенной", и ни одна лампа не должна гореть.

Если после последнего нажатия кнопки *Вкл* кнопка *Счетчик* была нажата нечетное число раз, должна гореть лампа *Нечет*.

Если после последнего нажатия кнопки *Вкл* кнопка *Счетчик* была нажата четное число раз, должна гореть лампа *Чет*.

Если любая из ламп перегорает, другая лампа должна мигать каждую секунду.

Эта спецификация достаточно сжатая и для многих целей вполне удовлетворительная. Она отражает то, как, по мнению пользователя, должен работать прибор. Но программист, который должен написать программу для имитации такого поведения, немедленно обнаружит в ней по меньшей мере одну неоднозначность: что означает, что лампочка должна мигать каждую секунду? Вам это все еще кажется очевидным? Посмотрим на циклы работы, изображенные на рис. 26.2.

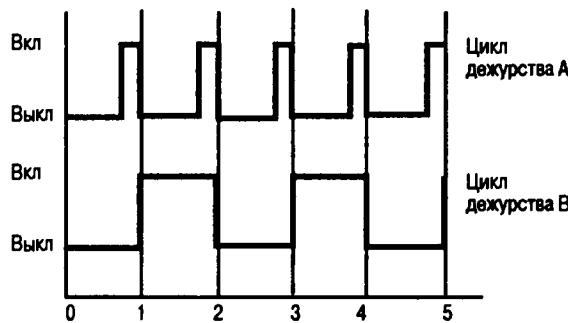


Рис. 26.2. Возможные циклы работы ламп

На месте программиста, какой цикл вы бы выбрали, А или В? Хотя большинство выбирает цикл В, становится ясно, что данное требование неоднозначно. Итак, внимательный программист обнаружит эту неоднозначность и попытается разрешить ее, спросив клиента, какой рабочий цикл он должен использовать. Но если программист не столь сообразителен или не заметил эту неоднозначность либо принимает решение на свое усмотрение ("Я знаю, что имеется в виду, поскольку знаю, как этот прибор должен работать"), поведение изготовленного прибора может значительно отличаться от того, что подразумевали высказанные пользователем требования. Проект может оказаться под вопросом.

В более сложных приложениях, возможно, не имеет значения, загорается лампочка на 1 с или на 0.25 с. Но если это приложение – электрохирургический инструмент, то

значение данного факта огромно. Мощность, подведенная к электроду, может быть на 100 процентов выше в рабочем цикле В, чем в А, и результаты могут быть печальными.

Итак, на вопрос: *какой уровень конкретизации необходимо обеспечить?* можно ответить следующим образом: *это зависит от содержания приложения и того, насколько те, кто выполняет реализацию, способны принять правильные решения или хотя бы задать вопросы там, где есть неоднозначность.*

Для прибора, определяющего четное и нечетное число нажатий кнопки, приведенная спецификация может считаться удовлетворительной. Для электрохирургического прибора потребуется затратить дополнительные средства на описание требований. Понадобится временная диаграмма; кроме того, в спецификации, возможно, нужно будет определить такие аспекты, как время нарастания тока при включении, точность, с которой должно контролироваться время включения тока ($\pm x$ мс), и другие факторы. В противном случае прибор будет работать некорректно.

Рассмотрим рис. 26.3. Задача состоит в том, чтобы найти “золотую середину”, когда требования обеспечивают необходимый уровень конкретизации и в то же время оставляют именно столько неоднозначности, сколько требуется, чтобы разработчики в дальнейшем могли выбрать пути продвижения.



Рис. 26.3. Зависимость понятности от уровня конкретизации

При движении вправо от золотой середины по изображенной на рис. 26.3 кривой, уменьшается как понятность, так и неоднозначность. Например, если мы предложим неискусенному пользователю временные диаграммы, на которых представлены временные допуски, и будем повсюду придерживаться этого уровня конкретизации, пользователь может и вовсе не понять спецификацию или даже просто не пожелает тратить время на ее чтение. Еще хуже то, что из-за этой чрезмерной тщательности пользователь может слишком довериться нам и не станет тратить время на проверку. Есть также риск, что пользователь не сумеет “за деревьями увидеть лес” (“Я не хотел, чтобы здесь был источник света. Я хотел, чтобы вы включили аварийное освещение в конце конвейера”).

При движении влево от оптимальной области неоднозначность возрастает, а понятность снижается. Например, в предельном случае можно просто сказать: “Постройте прибор, подсчитывающий четное/нечетное количество”, и никто не сможет понять, что имеется в виду.

Умение находить золотую середину – достаточно сложное дело. Оно будет зависеть от способностей членов команды, содержания приложения и требуемого уровня гарантий, что система “работает так, как нужно”.

Пример неоднозначности. У Мери был маленький барашек

Давайте рассмотрим забавный пример неоднозначности, а также попробуем найти дополнительные рецепты, которые помогут нам справиться с ней, когда это необходимо. (Сторонники формального подхода могут непосредственно перейти к главе 28, "Теоретически обоснованные формальные методы".)

Остальным предлагаем шутливое упражнение, которое мы нашли в книге Гауса и Вайнберга (Cause, Weinberg, 1989). Оно иллюстрирует проблему неоднозначности, а также некоторые глубокие выводы относительно возможных решений.

Рассмотрим известную строчку из стиха, который няни читают детям: "Мери имела маленького барашка (Mary had a little lamb)". Хотя вряд ли кто-то будет создавать информационную систему на основе этого предложения, тем не менее, интересно рассмотреть, что же оно означает? Для этого можно выделить *ключевые слова* данного предложения и рассмотреть варианты, соответствующие различным значениям каждого из них. Сосредоточим внимание на словах "имела" (had) и "барашек" (lamb). "Имела" – прошедшее время глагола "иметь"; следовательно, можно использовать значения глагола "иметь" (have), а также значения существительного "барашек". Итак, что же такое "иметь".

иметь 1а: иметь во владении в качестве собственности... **4а:** приобретать или получать во владение... **4в: ПРИНИМАТЬ;** состоять в браке... **5а:** отметка или отличительная характеристика (иметь рыжие волосы)... **10а:** удерживать в невыгодном положении или как-то ущемлять... **10б: ОБМАНУТЬ, ОДУРАЧИТЬ ...12: ПРОИЗВЕСТИ НА СВЕТ, РОДИТЬ** (иметь ребенка)... **13: счастье... 14: ДАВАТЬ ВЗЯТКУ, ПОДКУПАТЬ**

Теперь посмотрим, какие значения есть у слова "барашек".

барашек 1а: молодая овца в возрасте до одного года или без постоянных зубов... **1б:** молодняк других животных (в частности, небольших антилоп)... **2а:** человек, слабый и симпатичный как маленький барашек... **2б: ДОРОГОЙ, ЛЮБИМЫЙ...** **2в:** человек, легко идущий на обман (нечистый на руку), особенно в сфере торговли... **3а:** "седло барашка" как кулинарное блюдо²

Таким образом, фраза "Мери имела маленького барашка" может иметь любое из следующих значений.

"Овечьи интерпретации"

"Иметь"	"Барашек"	Интерпретация
1а	1а	Мери владела маленькой овечкой в возрасте до года либо без постоянных зубов
4а	1а	Мери приобрела маленькую овечку в возрасте до года либо без постоянных зубов

¹ Соответствующие значения глагола *have* в английском языке были взяты авторами из Webster's Seventh New Collegiate Dictionary (Springfield, MA: Merriam Co., 1967).

² Там же.

“Иметь”	“Барашек”	Интерпретация
5а	1а	Мери – владелец маленькой овечки в возрасте до года либо без постоянных зубов
10а	1а	Мери держала в руках (не пускала порезвиться) маленького ягненка в возрасте до года либо без постоянных зубов
10б	1а	Мери обманула маленькую овечку в возрасте до года либо без постоянных зубов
12	1б	Мери родила маленькую антилопу
12	2а	Мери является (или была) матерью некоего маленького, симпатичного существа
13	3а	Мери съела небольшую порцию седла ягненка
14	2в	Мери подкупила мелкого торговца цennыми бумагами, который нечист на руку

Для тех, кто помнит этот стих с детства и сам читает его на ночь своим детям, все это может казаться совершенно нелепым: “Разве здравомыслящий человек будет интерпретировать такую знакомую фразу столь странными способами?”. Но это перестанет быть столь уж невероятным, если мы представим, что человек из другой среды, принадлежащий к другой культуре, попытается интерпретировать данную фразу, основываясь исключительно на словарном определении двух ключевых слов. Раз это возможно при интерпретации строчки из стиха, это тем более может происходить в сложных программных системах, не имеющих аналогов в прошлом.

Как избежать неоднозначности

Один из способов избежать неоднозначности состоит в том, чтобы вместо естественного языка использовать методы “формальной” спецификации требований, которые приводятся в главе 28. Пользователи и заинтересованные лица, не входящие в команду разработчиков, предпочитают естественный язык, и даже компьютерщики в повседневном общении отдают ему предпочтение. Но хотя обе группы считают естественный язык удобным, все же их интерпретации и предположения могут существенно отличаться.

Может оказаться, что полностью устраниТЬ неоднозначность невозможно, но можно предпринять определенные усилия в нескольких направлениях. Гаус (Gause) и Вайнберг (Weinberg) в своей книге (1989) предложили несколько методов.

- **Эвристика запоминания.** Необходимо попросить несколько человек, как из группы разработчиков, так и из группы пользователей/заинтересованных лиц, постараться по памяти восстановить исходное требование клиента. Непонятная часть, которую трудно вспомнить, вероятно, является наиболее неоднозначной. Стоит обратить на нее внимание и постараться переопределить ее более понятно, чтобы она поддавалась запоминанию.
- **Метод ключевых слов.** Как было показано на примере анализа строчки из стиха, нужно выявить ключевые слова и выписать их определения, пользуясь авторитетным источником, признанным различными участниками проекта. Затем нужно соединить различные интерпретации, как это проделывалось с Мери и ее барашком. В качестве быстрой проверки действенности данного метода, можно отметить, что интерпретация (1а и 1а) “Мери владела маленькой овечкой в возрасте до года либо без постоянных зубов”, вероятно, более всего соответствует смыслу сказочной истории.

- *Метод ударения.* Необходимо прочесть требование, выделяя с помощью интонации отдельные слова, пока не будут найдены все возможные интерпретации. Если корректна только одна интерпретация, следует переформулировать требование надлежащим образом; если существует несколько корректных интерпретаций, возможно, необходимо создать соответствующие дополнительные требования. Мы проиллюстрируем применение этого метода в процессе дальнейшего исследования примера с Мери и ее барашком.
- *Другие методы.* Если нужно, используйте рисунки, графики или формальные методы для выявления неоднозначности и ее устранения.

Предположим, что мы увидели фразу “Мери имела маленького барашка” в нашем наборе требований и пытаемся удостовериться, что мы правильно поняли, что на самом деле имел в виду пользователь. Произнося предложение и выделяя интонацией различные слова, можно выявить следующие варианты.

- *Мери имела маленького барашка;* в данном случае пользователь, вероятно, хотел сказать, что это был барашек Мери, а не Ричарда или кого-нибудь другого.
- *Мери имела маленького барашка;* возможно, его у нее больше нет. Возможно, все дело именно во времени данного глагола.
- *Мери имела маленького барашка (Mary had a little lamb);* ударение на единственном числе этого барашка, т.е. у Мери он был один, а не целое стадо.
- *Мери имела маленького барашка;* т.е. это был один из самых маленьких барашков, каких вам доводилось видеть.
- *Мери имела маленького барашка;* суть в том, что у Мери был именно барашек, а не свинья, корова или взрослый баран. Но мы и в этом случае можем ошибаться, думая, что у нее был детеныш антилопы.

Что делать

Ни один из методов не является универсальным. Чтобы найти оптимальное сочетание неоднозначности и конкретизации, нужно разработать практические навыки применительно к условиям вашей организации и содержанию конкретного проекта. Уровень конкретизации, который необходимо обеспечить, может даже меняться со временем в зависимости от изменения профессионального уровня участников процесса и их понимания области, в которой вы работаете.

Ниже приводятся наши рекомендации относительно нахождения “золотой середины” для вашего проекта.

- Используйте везде, где это возможно, естественный язык.
- Используйте рисунки и диаграммы, чтобы лучше проиллюстрировать сущность.
- Если сомневаетесь – спрашивайте! Даже если сомнений нет, все равно старайтесь спрашивать.
- В тех случаях, когда неверное понимание недопустимо, дополните ваши спецификации более формальными методами.

Учите ваших людей выявлению проблемы неоднозначности и возможным способам ее решения.

Глава 27

Критерии качества требований к программному обеспечению

Основные положения

- Основным критерием качества является сам факт наличия набора требований; в свою очередь, эти требования также должны удовлетворять критериям качества.
- Для того чтобы удостовериться в качестве требований, модели прецедентов, спецификаций прецедентов и акторов, можно использовать контрольные перечни.
- Высококачественный пакет Modern SRS Package должен иметь хорошее оглавление, индекс, глоссарий, а также должен содержать историю исправлений.

Качество, как и мастерство в искусстве, трудно измерить. (“Я узнаю настоящее искусство, когда увижу его.”) Тем не менее нам нужно найти способ измерять и повышать качество наших спецификаций. Очевидно, что одной из мер качества является получение в результате завершения проекта хорошего продукта. Но от такой меры проку немного, так как помимо требований существует много других факторов, влияющих на результат.

Мы предлагаем оценивать качество спецификаций в целом, рассматривая следующие основные элементы.

- Качество каждой отдельной спецификации
- Качество применяемых методов (например, качество спецификаций прецедентов, акторов и т.п.)
- Качество пакета, объединяющего все отдельные спецификации

Начнем с рассмотрения качества отдельных спецификаций. Чтобы нечто измерить, естественно, необходимо иметь способ проведения измерений.

Девять показателей качества

Мы уже обсуждали, что собой представляют “хорошие” требования: они удовлетворяют предложенному определению (23) и не содержат описания деталей проектирования и реализации, а также вопросов процесса программирования или управления проек-

том. Но как отличить качественный набор требований от некачественного? Руководствуясь стандартом IEEE 830, который выделяет восемь “критериев качества” для оценки SRS, мы добавим еще один и объясним, как он поможет нам разработать качественный набор требований. Итак, высококачественный пакет Modern SRS Package должен быть

- корректным;
- недвусмыслиенным;
- полным;
- непротиворечивым;
- упорядоченным по важности и стабильности;
- поддающимся проверке;
- модифицируемым;
- трассируемым;
- понимаемым.

Мы добавили девятый критерий, *возможность понимания*, так как твердо верим, что важнейшим залогом успеха проекта является *общение его участников*.

Обеспечение качества набора требований второстепенно по сравнению с задачей создания этого набора. Тем не менее, обсудим более подробно каждый из этих девяти элементов, так как они придают дополнительную глубину процессу разработки требований, а также позволяют нам лучше понять природу “хороших” требований.

Корректные требования

Дэвис (Davis, 1993) предложил следующую формулировку и ее иллюстрацию (рис. 27.1): “SRS (набор требований к программному обеспечению) является корректным тогда и только тогда, когда каждое требование, сформулированное в нем, представляет нечто, требуемое от создаваемой системы”.

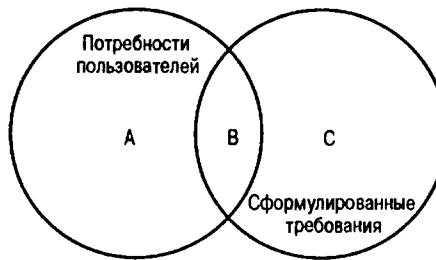


Рис. 27.1. Множества потребностей и требований

Если левый круг (область А) представляет множество потребностей пользователя, а правый (область С) – требования, корректные требования будут находиться в области пересечения кругов, области В.

Просто записывая некую информацию в документ, нельзя гарантировать, что она корректна; такую гарантию не сможет обеспечить и применение любого средства автоматизированного проектирования. Если истинное требование пользователя для системы расчета заработной платы состоит в том, чтобы ставка всех служащих была автоматиче-

ски повышена на 5 процентов, а команда разработчиков по невнимательности создает требование, предусматривающее 10-процентное повышение ставки, это, безусловно, некорректно. Но проверить это можно только с помощью просмотра пользователем.

Это явление нельзя назвать новым и неизвестным; с ним команды проектировщиков сталкивались с начала работы над самыми первыми проектами программных систем, равно как и при работе над другими проектами. Но в программных проектах часто встречается ситуация, когда опускается информация из области А и включается нежелательная информация из области С. Информация из области С может представлять собой детали проектирования/реализации, о чем мы предупреждали ранее, но это могут быть также требования, которые пользователь никогда не высказывал. Иногда эту информацию включают энтузиасты из числа персонала, занимающегося маркетингом или техническими аспектами, которые думают: «Мы уверены, что пользователь придет в восторг от этой функции, как только увидит ее». (Мы вновь вернемся к теме неправомерных требований в части 6.)

Недвусмысленные требования

Требование является недвусмысленным *тогда и только тогда, когда его можно однозначно интерпретировать* (IEEE 830-1998, § 4.3.2, 1994). Хотя главным свойством любого требования по праву считается корректность, неоднозначность зачастую представляет собой более сложную проблему. Если формулировка требований может по-разному интерпретироваться разработчиками, пользователями и другими участниками проекта, вполне может оказаться, что построенная система будет полностью отличаться от того, что представлял себе пользователь. Эта проблема всегда может возникнуть, если требования пишутся на естественном языке, так как члены организации, относящиеся к разным группам, настолько привыкли к своей интерпретации слова или фразы, что им никогда и в голову не придет, что другие могут интерпретировать это слово иначе (этую проблему мы обсуждали в главе 26).

Полнота набора требований

Набор требований является полным *тогда и только тогда, когда он описывает все важные требования, интересующие пользователя, в том числе требования, связанные с функциональными возможностями, производительностью, ограничениями проектирования, атрибутами или внешними интерфейсами* (IEEE 830-1998, § 4.3.3, 1994). Полный набор требований должен также задавать требуемый ответ программы на всевозможные классы ввода – как правильные, так и неправильные – во всевозможных ситуациях. Помимо этого, он должен содержать полные ссылки и подписи всех рисунков, таблиц и диаграмм набора требований, а также определения всех терминов и единиц измерения.

Гарантия полноты

О некоторых аспектах полноты может судить любой компетентный рецензент, который критически оценивает пакет программных требований, чтобы удостовериться, что все рисунки, таблицы и диаграммы снабжены соответствующими ссылками и подписями. Кое-что может оценить даже сам разработчик, не имея специальных знаний о приложении. Например, если в требованиях указано: *система должна принимать вводимое пользователем отдельное число и возвращать квадратный корень из него с точностью до трех знаков после запятой*, возникает очевидный вопрос: *что будет, если пользователь попытается ввести отрицательное число?*

Вообще-то, ничего страшного в попытке вычислить квадратный корень из отрицательного числа нет, если в приложении имеет смысл возвращение в качестве ответа системы мнимого числа. Но в таком случае уже требуется понимание проблемной области, чтобы отличить правильный и неправильный ввод. Эта проблема особенно часто встречается при задании верхнего и нижнего пределов вводимых числовых параметров, длины символьных строк и т.п. Поскольку эти детали зачастую игнорируются в требованиях, разработчику приходится осознанно или неосознанно принимать решение, и в результате получается система, которая отказывается принимать имя клиента, состоящее из более чем 25 символов, или выдает причудливые результаты при ошибочном вводе пользователя.

Просмотр возможных классов вводов с целью удостовериться в том, что полный набор требований надлежащим образом описывает поведение системы при правильных и неправильных вводах, — это то, о чем знает каждый разработчик, хотя мы до сих пор делаем ошибки в написании таких требований. Именно для пользователей типично игнорировать эту область при обсуждении: “Почему нормальный человек будет пытаться ввести отрицательное число, когда система спрашивает о возрасте?”. Опытный разработчик знает, что это может случиться из-за простой опечатки или потому, что пользователь умышленно пытается “повредить” систему либо по другим неясным причинам.

Полнота нефункциональных требований

Чаще других пропускают аспекты производительности и ограничения проектирования, а также предположения о внешних интерфейсах с другими системами. Мы советуем (следуя нашим указаниям, которые мы предлагали при обсуждении практичности, надежности, производительности и возможности сопровождения) создать простой контрольный перечень (checklist), содержащий вопросы, которые необходимо задать при выявлении ограничений проектирования. При этом разработчики и пользователи, по крайней мере, могут быть уверены в том, что они задали соответствующие вопросы при создании требований. Пользователь-новичок вновь может выразить недовольство: “Конечно же, я хочу, чтобы система имела хорошие характеристики производительности, это настолько очевидно, что я не понимаю, зачем специально указывать это”. Опытный разработчик знает, насколько важно указать требования производительности в виде максимального и среднего времени ответа или в виде утверждения: “Время ответа для 90 процентов всех транзакций будет составлять менее 3 секунд”.

Полнота функциональных требований

Вопросы полноты функциональных требований более сложные. Не будучи экспертом в области приложения, техническому разработчику очень трудно узнать, не пропущена ли важная часть функциональных возможностей. В конце концов, поскольку все функциональные возможности новые, откуда вы можете знать, сколько их еще должно быть?

Иногда эти возможности настолько привычны и “очевидны”, что пользователь даже не осознает, что они есть. (“Конечно, мы запускаем систему расчета заработной платы на день раньше, если день выплаты приходится на праздник. Мы всегда так делаем! Какие другие варианты вы можете себе представить?”)

В этом случае может помочь метод прецедентов.

Достижение полноты с помощью прототипирования

Раскадровка, проверка требований и создание прототипов системы при использовании итеративного подхода к разработке помогут справиться с большей частью этих проблем. Чем ближе мы подходим к реальному использованию и чем больший опыт работы с внедряемой системой приобретают наши пользователи, тем выше вероятность заметить проблемы в нашем определении.

Но даже тогда команда разработчиков должна идти на шаг вперед в своем анализе и задавать всевозможные вопросы "что, если..." с целью гарантировать полноту требований. При этом следует обратить внимание на граничные условия, исключения и неординарные события.

Например, иногда в описании функциональных возможностей представлены ситуации столь редкие, что они никогда не возникали в процессе деловой жизни пользователя; никто и не думает составлять требования для подобной ситуации. Пользователь системы



начисления зарплаты может считать, что требуемое поведение системы "очевидно" в случае, если день выплаты приходится на выходной. Но что если национальный праздник, праздник штата и городской праздник придется на три последовательных рабочих дня? "Такого не бывает", — может возразить пользователь, но разработчик в состоянии продемонстрировать, что такое может произойти в течение последующих 5 лет.

Эти вопросы не так уж нереальны, как может показаться. Суматоха вокруг "проблемы 2000" наглядно иллюстрирует последствия близоруких решений, основанных на "обоснованных" ожиданиях, касающихся будущих событий.¹

Непротиворечивость набора требований

Множество требований является внутренне непротиворечивым *тогда и только тогда, когда ни одно его подмножество, состоящее из отдельных требований, не противоречит другим подмножествам* (IEEE 830-1993, § 4.3.4.1, 1994). Конфликты могут иметь различную форму и проявляться на различных уровнях детализации; если набор требований был написан достаточно формально и поддерживается соответствующими автоматическими средствами, конфликт иногда удается обнаружить посредством механического анализа. Но, скорее всего, разработчикам вместе с другими участниками проекта придется провести проверку множества требований вручную, чтобы удалить все потенциальные конфликты.

Иногда конфликты бывают явными и очевидными. Например, одна часть требований гласит: *когда происходит X, следует выполнять действие P*, в то время как другая часть утверждает: *при возникновении X выполнять Q*. Порой неясно, является ли проблема конфликтом или следствием неоднозначности. Например, часть требований в системе расчета заработной платы может выглядеть так: *все служащие, достигшие 65 лет и более, в конце календарного года должны получить поощрение \$1000*, а другая часть гласит: *все сотрудни-*

¹ Известно, что в свое время были приняты неблагоразумные решения, приведшие к возникновению "проблемы 2000". Существует и множество других примеров. Например, при подготовке полета космического корабля Джеминай (Gemini) была допущена ошибка в программе бортового компьютера, вызванная сознательным пренебрежением определенными законами физики с целью повышения эффективности программы. Приятое тогда решение привело к тому, что место приземления на несколько сотен миль отличалось от расчетного.

ки, имеющие стаж работы 10 лет и более, в конце календарного года должны получить поощрение в сумме \$500. Как в этом случае поступить с работниками, для которых выполнены оба условия?

В данном случае прототипы мало чем могут нам помочь. Хотя они весьма успешно позволяют выявить пропущенные функции и очень эффективны при проверке требований пользователя, касающихся деталей ввода-вывода, редко бывает, чтобы пользователь или специалисты по тестированию или обеспечению качества работали с прототипом настолько тщательно, чтобы выявить наиболее скрытые ошибки-конфликты. Их необходимо выявлять с помощью тщательной выполняемой вручную ревизии и анализа полночь набора требований, опираясь на профессиональные навыки команды разработчиков и имеющиеся в наличии автоматизированные средства.

Упорядочение требований по их важности и стабильности

В высококачественном наборе требований разработчики, клиенты и другие заинтересованные лица упорядочивают отдельные требования по их важности для клиента и стабильности (IEEE 830-1993, § 4.3.5, 1994). Этот процесс упорядочения особенно важен для управления масштабом. Если ресурсы недостаточны, чтобы в пределах выделенного времени и бюджета реализовать все требования, очень полезно знать, какие требования являются не столь уж обязательными, а какие пользователь считает критическими.

Можно присвоить дополнительные атрибуты каждому требованию, как мы делали при описании требований в документе-концепции; особенно полезны стоимость, риск, сложность и т.п. Но многие из них, по всей видимости, будут зависеть от оценки стратегий реализации. Например, посмотрев на требование, разработчик может сказать: "Хм, я думаю, что реализовать это требование будет действительно очень сложно, я даже не уверен, что мы знаем, как вообще это сделать". Хотя эта информация и важна для управления масштабом и принятия решений об очередности реализации, она, как правило, описывается на более высоком уровне абстракции, представленном документом-концепцией. По причинам, которые мы уже обсуждали, эти элементы обычно не следует включать в набор требований.

На данном этапе лучше использовать атрибуты "важности" и "стабильности", которые более тесно связаны с мировосприятием пользователя. Пользователь может сказать: "Это требование не очень стабильно, поскольку мы ожидаем, что в будущем месяце изменятся государственные инструкции, влияющие на него. Но с другой стороны, оно очень важно для нас, так как влияет на нашу конкурентоспособность". До того, как команда разработчиков начнет разрабатывать стратегии, основанные на технологических возможностях, полезно упорядочить требования, например, с помощью состоящей из двух столбцов таблицы.

Требования, упорядоченные по их важности	Требования, упорядоченные по их стабильности
SR103	SR172
SR172	SR103
SR192	SR063
SR071	SR071
SR063	SR192

Обладая этой информацией (в случае равенства других факторов), благоразумный менеджер разработки выделит пропорционально большую часть ресурсов на SR103 и SR172 и, скорее всего, вычеркнет SR071, так как оно не столь важное и относительно не-постоянное по своей природе.

Проверяемые требования

Требования должны быть верифицируемыми (или “тестируемыми”).

Требование в целом является верифицируемым *тогда и только тогда, когда каждое из составляющих его элементарных требований является верифицируемым*. Элементарное требование считается верифицируемым *тогда и только тогда, когда существует конечный финансово эффективный процесс, с помощью которого человек или машина могут определить, что разработанная программная система действительно удовлетворяет данному требованию* (IEEE 830-1993, § 4.3.6, 1994). Если сказать проще, практическая задача состоит в таком определении требований, чтобы можно было впоследствии протестировать их и выяснить, действительно ли они выполняются.

Вряд ли можно предложить строго научное доказательство того, что каждое требование является верифицируемым. В большинстве случаев это и не нужно. Создание соответствующих тестовых примеров и процедур для проведения верификации разработанной программы является задачей персонала, занимающегося тестированием и обеспечением качества. Конечно, чтобы иметь возможность сделать это, они нуждаются в хорошо определенных и недвусмысленных требованиях. На совещании, посвященном проверке, типичной является картина, когда каждый из участников обращается к специалистам по тестированию и спрашивает: “Вы уверены, что можете создать тестовый сценарий для проверки того, что данное требование выполнено?”.

Ниже приводятся примеры требований и типичные высказывания разработчиков и/или специалистов по тестированию о возможности их верификации.

- *Система должна поддерживать до 1000 пользователей одновременно.* “Это зависит от того, что разрешено делать этим пользователям после регистрации. Если пользователи имеют неограниченные возможности и теоретически могут ввести транзакцию, которая приведет к тому, что прикладная программа будет последовательно просматривать каждую запись базы данных, очень сложно проверить, сможет ли система работать с 1000 пользователями; существует ничтожная (но не нулевая) вероятность, что все эти пользователи одновременно захотят запустить такие транзакции. Но если пользователи ограничены в выборе запускаемых транзакций и мы сможем определить, какая из этих транзакций является наиболее трудоемкой, можно будет проверить, как удовлетворяется это требование (с разумной степенью достоверности), хотя нам придется использовать наше средство контроля загрузки для имитации 1000 активных терминалов.”
- *Система должна отвечать на произвольный запрос в течение 500 миллисекунд.* “Все зависит от того, что подразумевается под словом *произвольный*. Если число возможных запросов конечно и нам удастся выявить наиболее сложные из них, мы сможем проверить поведение системы.”

- Цифры на экране, показывающем время, должны хорошо выглядеть.
“Даже не думайте об этом. Красота – вопрос вкуса.”
- Система должна быть дружественной пользователю. “Это еще хуже, чем приятная форма! Если не будут тщательно определены условия и детали, дружественность пользователю является просто приглашением для введения аргументов.”
- Система должна экспортить данные для просмотра (*view data*) в формате, в котором в качестве разделителя используется запятая. “Хотелось бы уточнить некоторые детали; например, что будет, если эти данные представляют собой пустое множество? Но в принципе, мы можем проверить, будет ли система вести себя нужным образом в данном вопросе.”

88:88

Верификация и проверка правильности являются важными вопросами при разработке высококачественной программы. Мы вернемся к этой теме в части 6.

Модифицируемый набор требований

Множество требований является модифицируемым *тогда и только тогда*, когда его структура и стиль таковы, что любое изменение требований можно произвести просто, полно и согласованно, не нарушая существующей структуры и стиля всего множества (IEEE 830-1993, § 4.3.7, 1994). Для этого требуется, чтобы пакет имел минимальную избыточность и был хорошо организован, с соответствующим содержанием, индексом и возможностью перекрестных ссылок. Это не всегда означает, что пакет ведется и поддерживается с помощью некоего автоматического средства, но в больших системах, которые могут иметь тысячи требований, использование подобных средств становится практически необходимым.

Требования будут модифицироваться, нравится это кому-то или нет; альтернатива – это “замороженный” пакет требований, что равносильно его отсутствию и, соответственно, “провалу” проекта. Если требование (или содержащий его пакет) немодифицируемо, оно фактически становится недействительным через несколько недель или месяцев, так как команда постепенно откажется от своих усилий по его изменению и поддержанию его актуальности.

Каждому администратору программного продукта хочется думать, что его множество требований является модифицируемым, и каждый производитель вспомогательных программ хвастается, что одним из главных достоинств его средства является то, что оно действительно обеспечивает модифицируемость. Все это звучит красиво, но нужно его опробовать, чтобы увидеть, работает ли оно. И это нужно делать для того же масштаба и уровня сложности, что и в будущем проекте.

Трассируемые требования

Требование в целом является трассируемым *тогда и только тогда*, когда ясно происхождение каждого из составляющих его элементарных требований и существует механизм, который делает возможным обращение к этому требованию при дальнейших действиях по разработке (IEEE 830-1993, § 4.3.8, 1994). На практике это обычно означает, что каждое требование имеет уникальный номер или идентификатор. Иногда можно использовать ключевое слово, например “должна” (*shall*), чтобы выделить требование и отличить его от других не столь важных утверждений, комментариев и т.п., которые также могут содержаться в набо-

ре требований. При использовании автоматического средства работы с требованиями, идентификация требований может осуществляться системой автоматически.

В пределах одного проекта или, возможно, одного пакета будет необходимо трассировать одни компоненты к другим. Например, некоторые компоненты будут зависеть от других; если одни меняются, это повлияет и на другие. Некоторые утверждения требований могут уточняться "дочерними" требованиями (субтребованиями), для которых возможность трассировки является очевидным свойством. Нам необходима возможность обратной трассировки (backward traceability) – от текущих стадий к предыдущим стадиям определения или разработки, в частности, к документу-концепции, который мы обсуждали в части 3. Например, из табл. 28.1 видно, что все требования SR63.1, SR63.2, SR63.3 можно трассировать к Функции 63 документа-концепции (Vision document). Это очень существенно, если нам понадобится добавить или исключить некоторые функции; это также существенно, если возникают трудности с определенными требованиями и необходимо вновь согласовывать с пользователем сроки или бюджет для затронутой этим процессом функции. Также необходима возможность прямой трассировки (forward traceability) – от текущего требования ко всем подчиненным ему требованиям, независимо от того, какие контейнеры (документы проектирования, блок-схемы, программный код, тестовые примеры и т.п.) порождаются данным контейнером.

Возможность трассировки имеет огромное значение. Разработчики могут использовать ее как для достижения лучшего понимания проекта, так и для обеспечения более высокой степени уверенности, что все требования выполняются данной реализацией. Например, мы использовали полную трассировку, чтобы соединить все элементы одного нашего медицинского проекта (среди которых были элементы документа-концепции, элементы SRS, тестирования, кодирования и ревизии проекта), которые возникали на протяжении жизни проекта. Когда все эти элементы взаимосвязаны, разработчики гораздо лучше подготовлены к управлению взаимодействиями между ними.

Трассировка также позволяет команде решать вопросы "что, если...". ("Что, если мы прямо сейчас изменим это требование? Повлияет ли это на разработку программы, и, если да, то на какие элементы? Придется ли нам пересматривать планы тестирования, и, если да, то какие?")

В том же проекте мы создавали матрицы трассировки, требуемые FDA для того, чтобы удостовериться, что продукт соответствует ее собственным требованиям. Матрицы трассировки – бесценный способ "проверить" действия по разработке и убедиться, что вы делаете все необходимое (и не делаете того, что делать не следует). Мы проведем масштабное исследование возможностей трассировки в части 6.

Понимаемые требования

Множество требований является *понимаемым*, если пользователи и разработчики способны прийти к полному согласию относительно отдельных требований и общих функциональных возможностей, подразумеваемых данным множеством. В документах, описанных в предыдущих главах данной книги, основное внимание уделяется общим описаниям и функциям системы, и их понять, как правило, не сложно. Но по мере уточнения определения системы, т.е. при разработке детальных требований, элементы становятся все более конкретизированными и детализированными и возникает искушение использовать более специальные термины. Таким образом, человек, который пишет требования, должен знать терминологию обеих сторон-участниц. Важно также, чтобы те,

кто пользуется набором требований, могли понять поведение системы в целом. Этого можно добиться с помощью раскадровок, сценариев или иллюстративных прецедентов, которые показывают, как предполагается использовать систему в ее операционной среде.

Показатели качества для модели прецедентов



Замечание. В этом разделе обсуждается широкий спектр вопросов, связанных с прецедентами. Иногда элемент контрольного перечня будет содержать ссылки на специфические элементы прецедента, которые не обсуждались в данной книге, так как они не очень важны для управления требованиями. Чтобы изучить эти вопросы, можно обратиться к соответствующей литературе, посвященной прецедентам. Мы рекомендуем две книги: Буч (Booch, 1999), а также Джейкобсон, Буч и Рамбо (Jacobson, Booch, Rumbaugh, 1999).

- Все ли прецеденты найдены? Выявленные прецеденты должны иметь возможность осуществить все варианты поведения системы; если это не так, значит, некоторые прецеденты пропущены.
- Все ли функциональные требования описываются прецедентами? Если вы намеренно отложили некоторые требования, чтобы заняться ими при объектном моделировании (например, нефункциональные требования), это необходимо где-то отразить. Если подобное требование затрагивает конкретный прецедент, это следует отметить в специальном разделе данного прецедента (Специальные требования, Special Requirements).
- Не содержит ли модель прецедентов ненужное поведение, т.е. не представляет ли больше функций, чем указано в требованиях?
- Действительно ли в модели необходимы все выявленные связи включения, наследования и генерализации? Если это не так, они могут быть избыточными, и их следует удалить.
- Зависят ли связи модели друг от друга? Важно, чтобы этого не происходило, поэтому нужно проверить это.
- Правильно ли произведено деление модели на пакеты прецедентов? Стала ли модель в результате проще и удобнее для восприятия и сопровождения?
- Можете ли вы, изучив модель прецедентов, составить четкое представление о функциях системы и о том, как они связаны?
- Содержит ли введение (Introduction) к модели всю необходимую информацию?
- Содержит ли общее описание (Survey Description) модели прецедентов всю необходимую информацию; например, представлены ли там наиболее типичные последовательности прецедентов?

Спецификации прецедентов

- Каждый ли прецедент имеет хотя бы один актор? В противном случае что-то неправильно; прецедент, который не взаимодействует ни с одним актором, следует удалить.

- Все ли прецеденты независимы друг от друга? Если два прецедента всегда активизируются в одной и той же последовательности, возможно, их удастся объединить в один прецедент.
- Есть ли прецеденты с очень похожим поведением или похожими потоками событий? Если да и вы хотите, чтобы их поведение оставалось таким же и в будущем, нужно объединить их в единый прецедент. Тогда в будущем будет проще производить необходимые изменения. *Замечание.* Если вы принимаете решение о слиянии прецедентов, необходимо проинформировать об этом пользователей, так как это может оказывать влияние на тех из них, кто взаимодействует с объединяемыми прецедентами.
- Не получилось ли так, что часть потока событий уже моделировалась в качестве другого прецедента? Если да, то можно в новом прецеденте использовать старый.
- Не является ли часть потока событий частью другого прецедента? Если да, то следует выделить этот субпоток и предоставить возможность рассматриваемым прецедентам использовать его. *Замечание.* Необходимо проинформировать пользователей при принятии решения “повторно использовать” субпоток, так как это может повлиять на тех из них, кто использует существующий прецедент.
- Будет ли поток событий одного прецедента включаться в поток событий другого? Если да, это следует моделировать с помощью отношений наследования прецедентов. (Мы не обсуждали этот вопрос, так как он не был столь важен для общей концепции прецедентов.)
- Имеют ли прецеденты уникальные, понятные и содержательные имена, чтобы не перепутать их на последующих этапах? Если нет, имена следует изменить.
- Понимают ли пользователи и клиенты имена и описания прецедентов? Каждое имя должно описывать поведение, поддерживаемое соответствующим прецедентом.
- Удовлетворяет ли прецедент все требования, которые очевидным образом регулируют его выполнение? Можно включить любые нефункциональные требования, описанные в других частях пакета Modern SRS Package.
- Соответствует ли последовательность взаимодействий актора и прецедента ожиданиям пользователя?
- Понятно ли, как и когда начинается и заканчивается поток событий прецедента?
- Может существовать поведение, которое активизируется только в случае невыполнения некоторого условия. Есть ли описание того, что произойдет, если данное условие не выполняется?
- Нет ли слишком сложных прецедентов? Если вы хотите, чтобы модель прецедентов можно было легко понять, нужно “расщепить” сложные прецеденты.
- Содержит ли прецедент раздельные потоки событий? Если это так, лучше разделить его на два или более отдельных прецедентов. Прецедент, содержащий независимые потоки событий, очень сложно понимать и обслуживать.
- Тщательно ли смоделирован субпоток событий в прецеденте?
- Ясно ли, кто хочет выполнять прецедент? Понятно ли назначение прецедента?
- Понятны ли взаимодействия с акторами и обмен информацией?
- Передает ли краткое описание истинную природу прецедента?

Акторы прецедента

- Все ли акторы иайдены? Иными словами, все ли роли в окружении системы учтены и смоделированы? Несмотря на проводимые проверки, нельзя быть уверенным в этом до тех пор, пока не будут иайдены и описаны все прецеденты.
- Каждый актор должен входить, по крайней мере, в один прецедент. Акторы, которые не упоминаются в описаниях прецедентов или не имеют коммуникационных связей с прецедентами, нужно удалить. Но актор, упоминаемый в описании прецедента, скорее всего, имеет с ним иекую связь.
- Можете ли вы назвать хотя бы двух человек, которые смогут выполнять действия, как конкретный актор? Если нет, проверьте, не является ли роль, моделируемая актором, частью другой роли. Если это так, следует произвести операцию слияния акторов.
- Есть ли акторы, играющие аналогичные роли по отношению к системе? Если это так, их следует объединить. Коммуникационные ассоциации и описания прецедентов показывают, как взаимосвязаны акторы и система.
- Если два актора играют по отношению к прецеденту одну и ту же роль, необходимо использовать генерализацию для моделирования их общего поведения.
- Если некий актор использует систему с помощью нескольких полнотью отличных способов или имеет несколько совершение различных целей при использовании прецедента, то, вероятно, вы имеете дело с несколькими акторами.
- Акторы должны иметь простые осмысленные имена, которые смогут понять пользователи и клиенты. Важно, чтобы имена акторов соответствовали их ролям. В противном случае их необходимо переименовать.

Критерии качества пакета Modern SRS Package

Помимо индивидуальных критериев качества существуют критерии, применимые к самому пакету. Эти критерии позволяют удостовериться, что сам пакет является высококачественным и понимаемым.

При работе с документацией сложнее всего иайти необходимую информацию. Сколько раз вы говорили: "Я знаю, что требование, касающееся безотказности, находится где-то здесь," но не могли иайти его. Слишком часто мы думаем, что достаточно просто зафиксировать собранные требования. Но это не так.

Мы обнаружили, что Modern SRS Package не просто организовывает и хранит требования; он также упрощает их использование. Хороший пакет спецификаций обладает следующими характеристиками.

Хорошо составленное оглавление

Обязательным элементом хорошего SRS является хорошо составленное оглавление (Table of Contents, ТОС). Мы убедились, что ТОС предоставляет еще больше преимуществ, чем может показаться на первый взгляд. Например, стремление к хорошему ТОС подталкивает автора к использованию полезных заголовков, которые затем возникают в оглавлении SRS. Посмотрите на оглавление данной книги, и вы заметите, что заголовки

сами по себе могут служить руководством для читателя. Таким образом, оглавление напоминает сжатую версию пакета.

При наличии современных инструментальных средств непростительно не иметь оглавления. Оглавление создается автоматически; автор может выбрать необходимый уровень детализации и форматирование. Мы считаем, что трех или четырех уровней заголовков вполне достаточно для обеспечения нужного уровня детализации пакета Modern SRS Package. Для того чтобы отделить друг от друга основные элементы оглавления, полезно использовать дополнительные промежутки.

Одной из часто встречающихся проблем является то, что ТОС не обновляется так, как это нужно для отражения текущего вида пакета. Следует удостовериться, что в процессе публикации SRS всегда обновляется ТОС, чтобы гарантировать правильность разбиения на страницы. Оглавление становится ненужным, если оно не обновляется. Из-за этого читатель начинает беспокоиться, что в SRS что-то не так.

К сожалению, оглавление сложно создавать, если необходимо включить в него несколько разнородных документов. Предположим, готовится SRS-пакет, содержащий несколько документов Word, несколько разработанных с помощью другого средства спецификаций прецедентов и несколько выполненных в Excel схем. Невозможно найти средство, которое сможет одновременно работать с подобными входными данными и подготовить хорошее оглавление. В таких случаях можно попытаться создать многотомное оглавление, в котором верхний уровень используется для указания основных групп, таких как документы Word, файлы модели прецедентов, электронные таблицы Excel. Затем можно применять отдельные ТОС-средства для создания оглавлений каждой из групп.

Хороший индекс

Индексы являются важной составной частью любого SRS. В отличие от оглавлений, создание индексов — более сложное дело, так как авторы должны определить, по каким элементам будет проводиться индексация. После этого создание индекса не представляет труда.

Часть проблем индексации является следствием различия представлений, поддерживаемых командой проекта. Например, в медицинском приборе требования восстановления после ошибки предохранительного устройства могут рассматриваться как “предохранительный” элемент одними членами команды и как элемент “исправления ошибки” другими членами. Обе точки зрения правильны; данные требования следует проиндексировать двумя способами.

Откровенно говоря, над индексацией приходится поработать, но, как правило, это нужно делать только один раз для каждого добавляемого в SRS нового требования. После этого элементы индекса существуют вместе с пакетом и становятся важной частью понимания проекта.

Индексование следует использовать для того, чтобы иметь доступ к пакету, помимо ТОС. Иными словами, не имеет смысла создавать индекс, элементы которого уже есть в оглавлении. Необходимо такое индексирование, чтобы читатель мог обратиться к понятиям, а не к заголовкам. Как и оглавления, индексы всегда следует обновлять в процессе публикации, чтобы обеспечить целостность пакета.

История исправлений

Крайне неприятно обнаружить, что вы просматривали устаревшую версию SRS. Каждая спецификация требований должна содержать страницу, посвященную истории исправлений, где хранится информация о соответствующих изменениях каждой версии элементов пакета. Как минимум, эта страница должна содержать следующую информацию.

- Номер исправления или код каждого изменения опубликованной информации
- Дату каждого исправления опубликованной информации
- Краткое описание произведенных исправлений
- Имя человека, ответственного за исправление

Кроме того, может оказаться полезным снабдить каждый изменяемый элемент SRS маркером исправлений. Отметки о внесении исправлений на полях помогают читателю найти изменения.

Большинство современных автоматических средств работы с документацией и требованиями обеспечивают мощные возможности контроля исправлений и автоматического ведения истории версий. Пользуйтесь ими!

Предостережение. Не устанавливайте контроль исправлений слишком рано (см. главу 34, “Управление изменениями”). Во время внесения исправлений одно и то же требование можно переделывать несколько раз, прежде чем его удастся удовлетворительно сформулировать. Не имеет смысла записывать все эти попытки; поэтому не включайте для пакета режим контроля исправлений, пока не достигнете относительной стабильности в процессе разработки программы. С другой стороны, не стоит слишком откладывать, в противном случае вас захлестнет неуправляемый процесс.

Глоссарий

Так как природа каждой прикладной области уникальна, в проектах со временем стремятся разработать специальный язык или систему сокращений. Чаще всего сокращения бывают мнемоническими, например “SRS”. Их следует по возможности избегать. Также нужно воздерживаться от использования терминов, имеющих смысл только в контексте определенной ситуации. Хороший SRS содержит словарь таких терминов, чтобы помочь пользователям понять язык данной прикладной области. Следует позаботиться о том, чтобы включить в глоссарий объяснения всех специфических терминов проекта, аббревиатур и специальных фраз.

Глава 28

Теоретически обоснованные формальные методы спецификации требований

Основные положения

- Формальные методы спецификации требований применяются тогда, когда описание требований слишком сложно для естественного языка или вам не удается создать недвусмысленную спецификацию.
- К формальным методам относится использование псевдокода, конечных автоматов, деревьев решений, диаграмм деятельности, моделей сущность-связь, объектно-ориентированных моделей и схем потоков данных.

До сих пор мы исходили из предположения, что большинство требований будет написано на естественном языке команды разработчиков (в форме традиционных утверждений или precedентов). Кроме того, предлагалось сопроводить требования диаграммами, таблицами, схемами и т.п., чтобы прояснить значение требований пользователя. Но иногда присущая естественному языку неоднозначность просто неприемлема, особенно когда требования касаются жизненно важных вопросов или когда неправильное поведение системы может привести к чрезвычайным экономическим или юридическим последствиям. Если определение требования сложно сформулировать на естественном языке и невозможно предотвратить неправильное понимание спецификации, следует попытаться написать эту часть требований с помощью теоретически обоснованных формальных методов.

Можно выбрать одно из перечисленных ниже формальных средств спецификации.

- Псевдокод
- Конечные автоматы
- Деревья решений
- Диаграммы деятельности (блок-схемы)
- Модели сущность-связь
- Объектно-ориентированные модели
- Схемы потоков данных

Мы не будем подробно описывать применение какого-либо из этих средств, так как каждое из них достойно отдельной книги. Мы просто предлагаем краткий их обзор.

В Modern SRS Package формальные методы следует использовать экономно и последовательно. При выборе конкретного формального метода необходимо руководствоваться здравым смыслом. При создании системы управления ядерным реактором, возможно, каждый аспект системы является критическим; однако в большинстве систем не более 10 процентов требований требуют такой степени формальности.

Если это возможно, следует использовать только один из этих формальных методов для всех требований определенной системы. Это упростит не очень подготовленному читателю задачу прочтения и понимания элементов пакета. Если все разрабатываемые организацией системы относятся к одной прикладной области, такой, например, как телефонные системы коммутации, то можно использовать один и тот же формальный метод для всех систем. Но в большинстве организаций нереально придерживаться единственного метода для всех требований во всех системах; тот, кто пишет требования, должен выбрать подход, наиболее соответствующий конкретной ситуации.

Псевдокод

Как и подразумевает его название, псевдокод – это квазязык программирования; попытка соединить неформальный естественный язык со строгими синтаксическими и управляющими структурами языка программирования. В чистом виде псевдокод состоит из комбинации следующих элементов.

- Императивных предложений с одним глаголом и одним объектом.
- Ограниченнего множества (как правило, не более 40–50) “ориентированных на действия” глаголов, из которых должны конструироваться предложения.
- Решений, представленных формальной структурой IF-ELSE-ENDIF.
- Итеративных действий, представленных структурами DO-WHILE и FOR-NEXT.

На рис. 28.1 представлен пример спецификации с помощью псевдокода алгоритма вычисления отложенного дохода от услуг в течение данного месяца в бизнес-приложении. Фрагменты текста на псевдокоде расположены уступами; такой формат используется для того, чтобы выделить логические “блоки”. Сочетание синтаксических ограничений, формата и разбивки существенно уменьшает неоднозначность требования, которое в противном случае было бы очень сложным и расплывчатым. (Так оно и было до написания псевдокода!) В то же время такое представление требования вполне понятно для человека, не являющегося программистом. Не нужно быть выдающимся ученым, чтобы понимать псевдокод, и нет необходимости знать C++ или Java.

Алгоритм вычисления отложенного дохода от услуг в течение месяца:

```
Set Sum(X)=0
FOR каждого клиента X
    IF клиент оплатил услуги вперед
        AND ((Текущий месяц)>=(2 мес. после даты приобретения))
        AND ((Текущий месяц)<=(14 мес. после даты приобретения))
    THEN Sum(X) = Sum(X) + (сумма, заплаченная клиентом)/12
```

Рис. 28.1. Пример спецификации с использованием псевдокода

Конечные автоматы

В некоторых ситуациях систему или ее дискретное подмножество удобно рассматривать как "гипотетическую машину, которая в конкретный момент времени может находиться только в одном из указанных состояний" (Дэвис (Davis, 1993)). В ответ на ввод, будь то данные, вводимые пользователем, или информация, поступившая от внешнего устройства, машина изменяет свое состояние и генерирует выводимую информацию или выполняет некое действие. Как вывод, так и следующее состояние можно определить, основываясь исключительно на знании текущего состояния и события, вызывающего транзакцию. Таким образом, поведение системы можно назвать детерминированным; можно математическим путем определить все возможные состояния и, как следствие, выводы системы, основываясь на множестве предлагаемых вводов.

Разработчики аппаратного обеспечения десятилетиями использовали конечные автоматы (Finite state machines, FSMs). Существует огромное количество литературы, описывающей создание и анализ таких автоматов. Математическая природа FSMs располагает к проведению строго формального анализа, поэтому описанные ранее в данной части проблемы непротиворечивости, полноты и неоднозначности значительно уменьшаются при их использовании.

Конечные автоматы зачастую представляются в виде диаграмм перехода состояний, как показано на рис. 28.2. В этой системе обозначений прямоугольники представляют состояние, в котором находится устройство, а стрелки — действия, переводящие устройство в другие состояния. Рисунок иллюстрирует переходы состояний "лампового ящика", рассматривавшегося в главе 26. В том примере выражение на естественном языке "лампа будет мигать каждую секунду" было несколько неоднозначным. Представленная на рис. 28.2 диаграмма перехода состояний не является неоднозначной и иллюстрирует, что цикл В действительно был выбран правильно. Если лампа перегорает, прибор чередует попытки включить четную и нечетную лампы; каждую в течение 1 секунды.

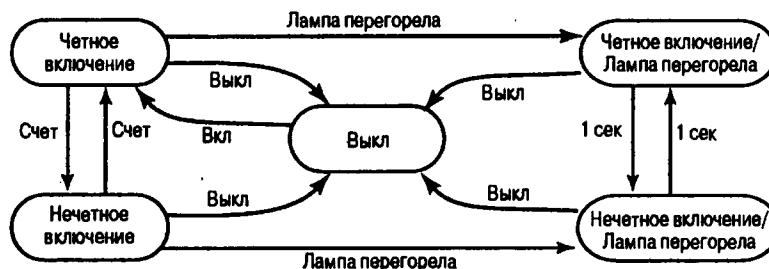


Рис. 28.2. Диаграмма перехода состояний

Предлагаем вам применить FSM для повторного определения прецедента "Управление освещением" системы HOLIS. Вы сразу заметите, что альтернативный поток Dim (изменение яркости) прекрасно подходит для представления с помощью FSM.

В более строгой форме конечный автомат представляется в виде таблицы или матрицы, где показаны все состояния, в которых может находиться устройство, вывод системы для каждого состояния и воздействия всех возможных событий на каждое возможное состояние. Это приводит к более высокому уровню детализации, так как каждое состояние и воздействие каждого события должны быть представлены в таблице. Например, табл. 28.1 определяет поведение светильниковой коробки в виде матрицы перехода состояний.

Таблица 28.1. Матрица перехода состояний для прибора подсчитывающего четное/нечетное число нажатий

Состояние	Событие						Вывод
	Нажатие Вкл	Нажатие Выкл	Нажатие Счет	Лампа перегорает	Каждая секунда		
Выключен	Четная горит	—	—	—	—	—	Обе лампы выключены
Четная горит	—	Выключен	Нечетное включение	Лампа перегорела/Четная горит	—	—	Четная горит
Нечетная горит	—	Выключен	Четное включение	Лампа перегорела/Нечетная горит	—	—	Нечетная горит
Лампа перегорела/Четная горит	—	Выключен	—	Выключен	Лампа перегорела/Нечетная горит	—	Четная горит
Лампа перегорела/Нечетная горит	—	Выключен	—	Выключен	Лампа перегорела/Четная горит	—	Нечетная горит

В таком представлении можно разрешить дополнительные неоднозначности, которые могут возникнуть при попытке понять поведение прибора.

- Что происходит, если пользователь нажимает кнопку Вкл, когда прибор уже включен? *Ответ:* ничего.
- Что происходит, если обе лампы перегорают? *Ответ:* прибор выключается.

Конечные автоматы широко применяются для некоторых категорий системных программных приложений, таких как системы обмена сообщениями, операционные системы и системы управления процессами. Они также являются удобным средством описания взаимодействия между внешним пользователем-человеком и системой, например взаимодействия клиента банка и банкомата, когда клиент хочет снять деньги со счета. Но конечные автоматы становятся слишком громоздкими, если нужно представить поведение системы как функцию нескольких вводов. В таких случаях требуемое поведение системы, как правило, является функцией всех текущих условий и событий, а не одного текущего события или некой цепочки прежних событий.

Таблицы решений

Достаточно часто встречаются требования, связанные с комбинацией вводов; различные комбинации вводов приводят к различным вариантам поведения или вывода. Предположим, имеется система с пятью возможными вводами (A, B, C, D, E) и требование, заключающееся в утверждении, напоминающем оператор псевдокода: “Если A истинно, то, если B и C также истинны, генерировать вывод X при условии, что E не является истинным, иначе требуемый

вывод – Y". Комбинация предложений IF-THEN-ELSE (если-то-иначе) быстро становится запутанной, особенно если, как и в этом примере, имеются вложенные предложения IF. Как правило, обычные пользователи не могут их понять, и невозможно гарантировать, что все возможные комбинации A, B, C, D и E рассмотрены.

Решением в данном случае является перечисление всех комбинаций вводов и описание каждой из них в явном виде в таблице. В нашем примере, когда допустимых значений вводов только два ("истинно" и "ложно"), получается 2^5 или 32 комбинации. Их можно представить с помощью таблицы, содержащей 5 строк (по одной для каждой вводимой переменной) и 32 столбца.

Графические деревья решений

Дерево решений применяется для графического отображения информации. Мы использовали это представление в главе 15, когда нужно было принять решение, какой прототип создавать. На рис. 28.3 показано дерево решений, используемое для описания последовательности действий системы HOLIS в чрезвычайной ситуации.

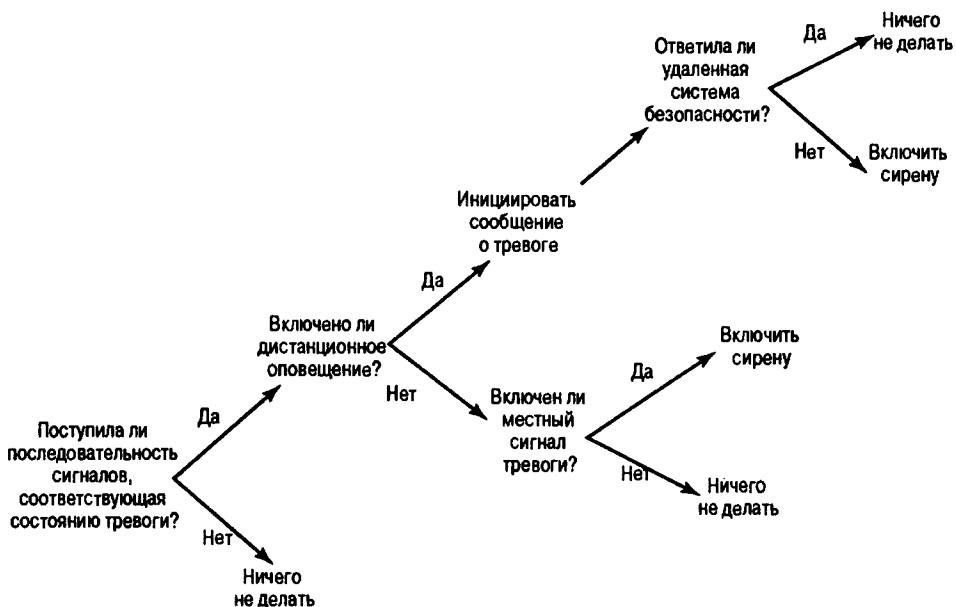


Рис. 28.3. Графическое дерево решений

Диаграммы деятельности

Блок-схемы (и их разновидность, UML-диаграммы деятельности) имеют несомненное преимущество — они достаточно известны. Даже люди, далекие от всего, что связано с компьютерами, знают, что такое блок-схема. Например, местная газета недавно поместила блок-схему, описывающую алгоритм, с помощью которого человеческий мозг принимает решение о покупке кабриолета SAAB. По понятным причинам все пути в этой блок-схеме заканчивались одним и тем же решением: "Купить SAAB". Где-то должна была быть логическая ошибка, но мы не смогли ее найти. Нам действительно нравится эта машина!

На рис. 28.4 показана типичная диаграмма деятельности в принятых в UML обозначениях. Хотя ту же информацию можно представить в форме псевдокода, UML обеспечивает визуальное представление, которое проще понять.

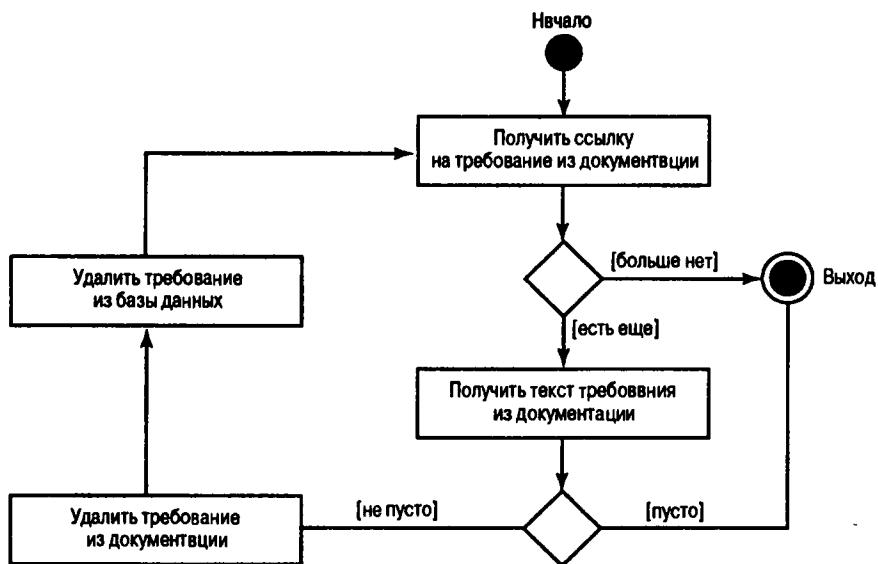


Рис. 28.4. Диаграмма деятельности

Единственная проблема при использовании диаграмм деятельности, как выяснили программисты за последние 30 лет, состоит в том, что достаточно скучно поддерживать их соответствие современному состоянию разработки. Такая проблема существует для всех графических представлений при отсутствии автоматических средств; никому не хочется перерисовывать диаграмму перехода состояний или дерево решений.

Модели сущность-связь

Если набор требований описывает структуру *данных* системы и связи между ними, удобно представить эту информацию в виде диаграммы сущность-связь (Entity-relationship diagram, ERD). На рис. 28.5 представлена типичная ERD.

Диаграмма сущность-связь обеспечивает высокоуровневое "архитектурное" представление данных (на рисунке – это заказчики, счета-фактуры, заказы и т.д.). Ее можно затем дополнить соответствующими подробностями, содержащими необходимую информацию (например, описание заказчика). В ERD основное внимание уделяется внешнему поведению системы, что позволяет ответить на такие вопросы, как "Можно ли в счет-фактуре указать более одного адреса выставления счета?" Ответ: нет.

Хотя модели сущность-связь являются мощным инструментом моделирования, они имеют существенный недостаток, который заключается в том, что далекому от техники читателю трудно их понять. Как видно из рис. 28.5, линии, соединяющие заказчика и заказ, а также заказ и счет-фактуру, отмечены кружками и "птичьими лапками". Возникает вопрос: что это все означает? Попытка ответить на него в рам-

ках данной книги была бы значительным отклонением от темы, чего мы решили избежать. Но если уклониться от ответа на этот вопрос при описании множества требований, это может привести к тому, что некоторые пользователи просто не поймут, что происходит. Можно организовать для пользователей двухдневные курсы по принятой в ERD-моделях системе обозначений, но неизвестно, воспримут ли они ее. Можно использовать ERD в качестве "формальной" формы документации в среде разработчиков.

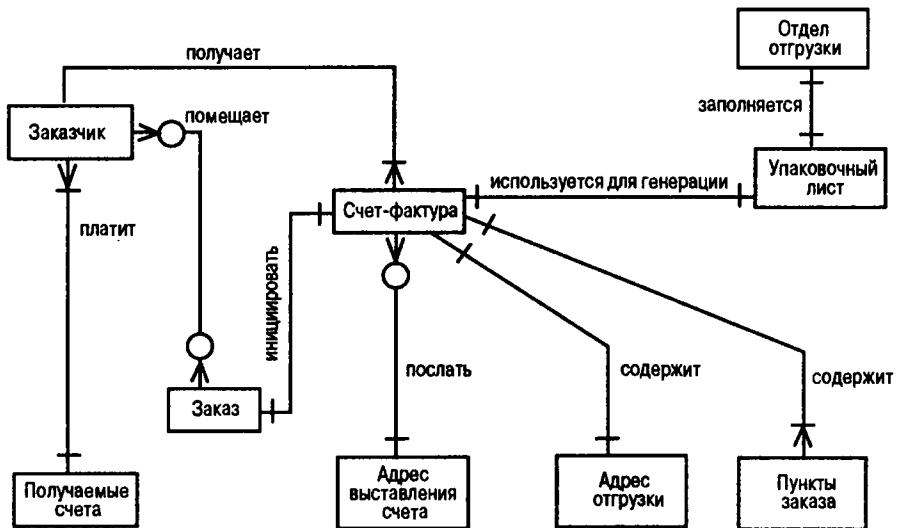


Рис. 28.5. Диаграмма сущность-связь

Объектно-ориентированные модели

Если детально разрабатываемые требования должны содержать описание структуры и связей *сущностей* системы (например, расчетных листов, служащих, сотрудников отдела заработка платы и т.д.), для более полного описания поведения системы можно использовать объектно-ориентированные модели. При современном уровне популярности объектно-ориентированных методов и быстром внедрении языка UML эти модели постепенно превращаются в спецификации и, более того, в реализационные модели, используемые при реализации функциональных возможностей системы.

Удобство заключается в том, что применение стандартов UML обеспечивает всем общее понимание того, что означает данное представление, и тем самым уменьшает неоднозначность, заставляя всех "говорить на одном языке", пусть и техническом.

Например, объект "Сотрудник" на рис. 28.6 будет описываться с помощью содержащихся в нем ориентированных на данные *атрибутов*, таких как название подразделения и должность, а также предоставляемых им *услуг*, таких как добавление нового сотрудника, удаление сотрудника и поиск конкретного экземпляра сотрудника.



Рис. 28.6. Объектно-ориентированная модель

Схемы потоков данных

При обсуждении требований в данной главе предполагалось, что мы имеем дело с требованиями "атомарного уровня". Как правило, так и бывает в типичном документе, но часто полезно иметь визуальное представление, иллюстрирующее структуру и организацию этих атомарных требований, а также связей ввода-вывода между ними. Для этого широко используется представление подобной информации в виде схемы потоков данных (рис. 28.7).

Модели, использующие схемы потоков данных (DFD), испытывают те же трудности, что и модели сущность-связь (ERD), хотя далекому от техники читателю обычно несколько проще понять значение DFD без специальной подготовки. Некоторые организации добились значительного успеха, используя DFD в качестве основы общения нетехнически ориентированных пользователей и разработчиков; в то время как другие обнаружили, что их пользователи блокируют любые попытки использовать столь "формальные" обозначения. Если схему потоков данных *в принципе* удаётся использовать, то возможно провести декомпозицию каждого из "кружков" (рис. 28.7) на DFD более низкого уровня. Так требования для кружка 5 ("закупаемые ресурсы") можно описать подробнее с помощью схемы более низкого уровня, которая иллюстрирует соответствующие детали. Процесс декомпозиции продолжается до тех пор, пока кружки действительно станут "атомарными"; на этом уровне связанные с кружком требования уже можно описать с помощью псевдокода, конечного автомата, дерева решений или блок-схем. (На самом деле сегодня существует более серьезная опасность при использовании DFD; приверженцы объектно-ориентированного программирования могут посчитать, что вы занимаетесь функциональной декомпозицией данных и, следовательно, являетесь ретроградом, и в дальнейшем будут игнорировать все, что вы скажете по любому поводу.)

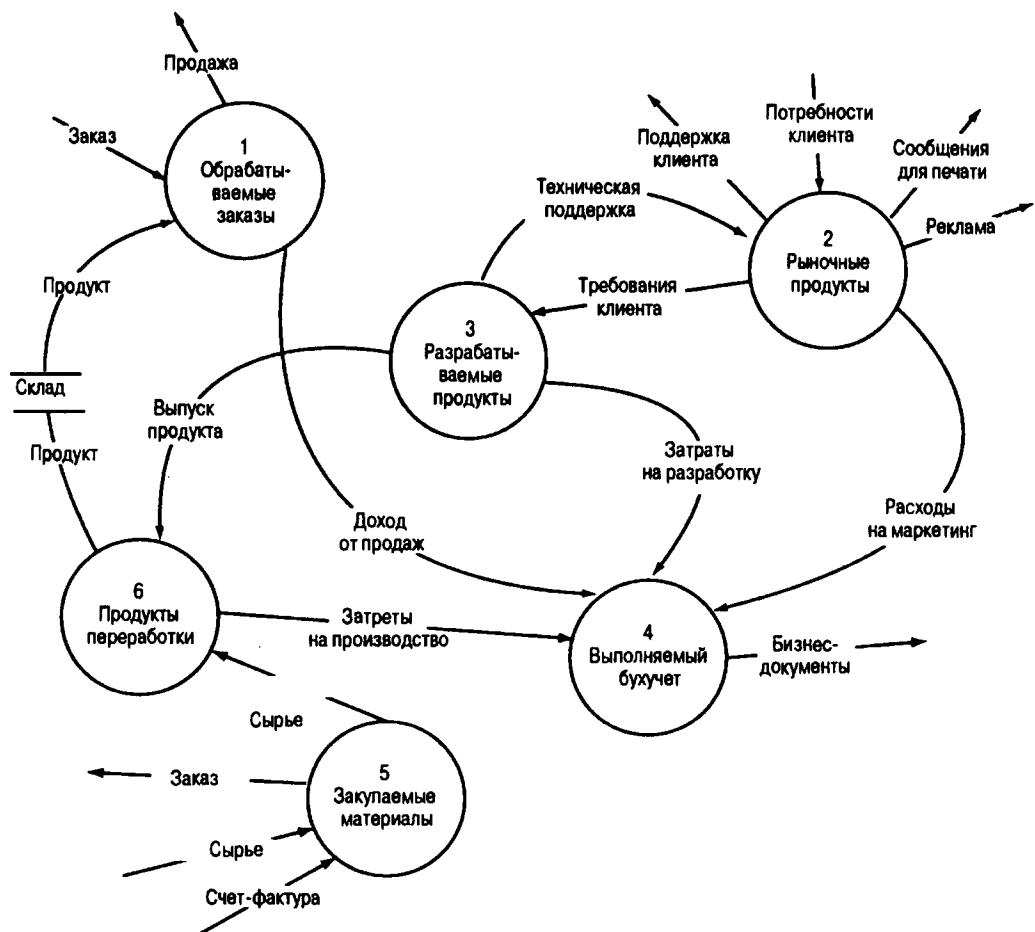


Рис. 28.7. Схема потоков данных

Ведение спецификаций

Работая с требованиями, мы не злоупотребляем этими формальными методами и используем их только для того, чтобы проиллюстрировать поведение системы. Это значительно уменьшает головную боль от необходимости их сопровождения. Кроме того, новое поколение автоматических средств разработки программ обеспечивает существенную поддержку прямого и обратного проектирования, т.е. они предоставляют возможность автоматически поддерживать синхронизацию кода и представления в модели. По мере развития этих средств появится возможность рассматривать изменения требований как принятые в процессе кодирования решения, которые влияют на внешнее поведение системы.

Когда дело касается сопровождения, на первом плане должен быть здравый смысл.

Типичная проблема сопровождения возникает, когда код или спецификации пересматриваются, а соответствующие формальные спецификации одновременно не исправляются. Теория гласит: "код – это спецификация". Мы не утверждаем, что абсолютным правилом является *обязательное* обновление формальных спецификаций по мере развития проекта. Но тех, кто случайно обратится к устаревшей документации, подстерегает множество ловушек. С нами это также случалось в процессе написания данной книги, когда мы говорили, что "не нужно обновлять этот рисунок", а затем по ошибке использовали устаревшую версию.

В том, что касается обновления, следует руководствоваться здравым смыслом. Обновляйте то, что необходимо, основываясь на важности данной информации. Если это не является жизненно важным, может быть, можно отказаться от обновления. Один из методов, который мы в свое время использовали, состоит в том, чтобы при принятии решения не обновлять некий документ, а делать на нем пометку "Устаревший".

Мы считаем устаревшую модель меньшим из двух зол. Лучше иметь устаревшую модель, чем не иметь модели спецификаций вовсе!

Рабочий пример

Все эти методы рассматривались командой проекта HOLIS при подготовке пакета HOLIS SRS Package. Первая версия данного пакета представлена среди артефактов системы HOLIS в приложении А.

Заключение части 5

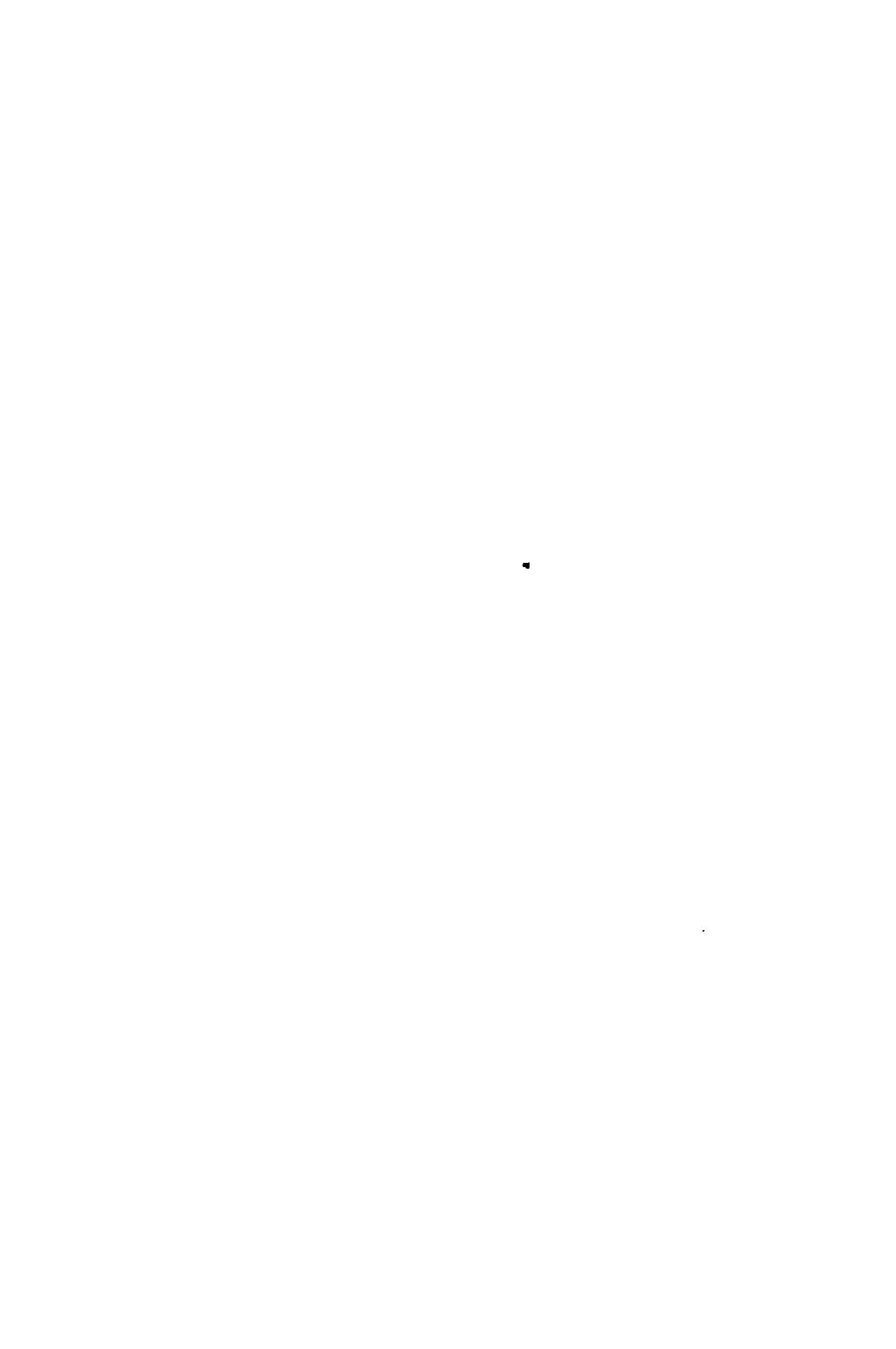
В части 5 мы выяснили, что требования должны полно и сжато фиксировать потребности пользователей в таком виде, чтобы разработчик мог построить удовлетворяющее их приложение. Кроме того, требования должны быть достаточно конкретными, чтобы можно было определить, когда они удовлетворены. Зачастую именно команда обеспечивает эту конкретизацию; это еще одна возможность убедиться, что определяется правильная система.

Существуют различные возможности организации и документирования этих требований. Мы остановились на пакете *Modern SRS Package*, логической конструкции, которая позволяет документировать требования в виде прецедентов, документов, форм баз данных и т.д. Хотя мы внесли несколько предложений относительно того, как организовать такой пакет, мы считаем, что форма не так важна, лишь бы пакет содержал все, что необходимо.

Вся разработка должна вытекать из требований, зафиксированных в пакете *Modern SRS Package*, а все спецификации пакета должны найти отражение в действиях разработки. Таким образом, все виды деятельности являются отражением пакета и наоборот. Пакет *Modern SRS Package* является "живой" сущностью; его следует пересматривать и обновлять на протяжении жизненного цикла проекта. В пакете указывается, *какие* функции должны осуществляться, а не то, *как* они осуществляются. Он используется для задания функциональных требований, нефункциональных требований и ограничений проектирования.

Мы также предложили набор показателей, которые можно использовать для оценки качества пакета и содержащихся в нем элементов. Если необходимо, документация требований может дополняться одним или несколькими более формальными либо более структурированными методами спецификации.

Пакет *Modern SRS Package* содержит детали, которые необходимы для *построения (реализации)* правильной системы. Эту часть работы над проектом мы обсудим в части 6, "Построение правильной системы".



Часть 6

Построение правильной системы

- Глава 29. Как правильно построить правильную систему: общие положения
- Глава 30. От понимания требований к реализации системы
- Глава 31. Использование трассировки для поддержки верификации
- Глава 32. Проверка правильности системы
- Глава 33. Применение метода анализа дивидендов для определения объема V&V-действий
- Глава 34. Управление изменениями



В части 5 мы ознакомились с некоторыми методами сбора, организации и документирования требований к разработке. Мы рассмотрели различные методы спецификации требований и отметили, что самое главное — *собрать все требования к проекту, а также способствовать тому, чтобы заинтересованные лица их поняли и одобрили.*

В данной части мы переходим от определения системы-решения к ее построению. Этот шаг наиболее сложный, и для его осуществления потребуется гораздо больше ресурсов. В последующих иескольких главах мы рассмотрим, как на основании спецификаций (посредством прецедентов) выполнить проектирование и реализацию системы.

Зачастую разработчики достаточно быстро переходят к построению системы и по ряду направлений достигают заметного прогресса. Но в результате оказывается, что, несмотря на затраченные усилия, клиент так и не получил ту систему, которую он хотел. В этой части книги мы хотим выйти за рамки действий по разработке и узнать: “Как определить, что мы создаем “правильную” систему?”.

Ответить на данный вопрос нам помогут верификация и проверка правильности. Мы подробно рассмотрим, как включить эти методы в деятельность по реализации, чтобы заострить внимание на том, что действительно важно для проекта.

Наконец, в процессе разработки никогда не удается избежать изменений. Поэтому в данной части исследуется природа изменений и обсуждаются такие способы их внесения и управления ими, которые позволяют не выпустить проект из-под контроля.

Как правильно построить “правильную” систему: общие положения

Основные положения

- Для правильного построения “правильной” системы необходимо постоянно убеждаться, что разработка проводится надлежащим образом, ее результаты корректны, и обрабатывать возникающие в процессе разработки изменения.
- Верификация (verification) позволяет удостовериться, что деятельности разработки неизменно соответствуют потребностям клиента.
- Проверка правильности (validation) демонстрирует, что продукт соответствует предъявляемым к нему требованиям и конечный результат получает одобрение заказчика.
- Поскольку изменения неизбежны, следует учитывать это при планировании и знать, как ими управлять.

Как мы уже неоднократно подчеркивали, крайне важно, чтобы каждый член команды понимал требования к проекту. Но на практике, к сожалению, невозможно ждать, пока абсолютно все станет ясно. Поэтому мы описали методы итеративной разработки, которые позволяют постепенно уточнять понимание системы по мере продвижения вперед. Но даже тогда проект, в котором до этого момента все шло хорошо, может разрушиться и превратиться в дезорганизованные действия. Причина в том, что команда уже понимает требования, но еще не в состоянии построить систему, которая удовлетворяет им. В данной главе мы рассмотрим несколько важных дополнительных концепций, которые позволят убедиться, что вы надлежащим образом строите “правильную” систему. Это подразумевает следующее.

- Необходимо постоянно убеждаться, что разработка находится на правильном пути.
- Необходимо убеждаться в корректности результатов разработки.
- Необходимо научиться обрабатывать изменения, возникающие в процессе разработки.

Рассмотрим каждый из этих пунктов более подробно.

Проверка того, что разработка находится на правильном пути

При проектировании и реализации команда должна иметь возможность постоянно проверять, не сбился ли проект “с пути”, т.е. соответствуют ли результаты разработки потребностям клиента.

Постоянно выполняемый процесс проверки того, что каждый шаг разработки является корректным, удовлетворяет потребности последующей деятельности и не является излишним, мы будем называть *верификацией (verification)*. К сожалению, в отрасли имеется множество различных толкований того, что же собой представляет верификация. Поэтому мы обсудим это важное понятие более подробно.

Принципы верификации программного обеспечения

Стандарт IEEE (1994) определяет верификацию следующим образом.

Процесс оценивания системы или компонента с целью определить, удовлетворяют ли результаты некой фазы условиям, наложенным в начале данной фазы (IEEE 1012-1986, §2, 1994).

Таким образом, верификация в значительной мере является аналитической деятельностью, в ходе которой необходимо убедиться, что каждая стадия разработки (например, реализация в программном обеспечении одного или нескольких требований) соответствует заданным на предыдущей стадии требованиям. Как минимум, хотелось бы верифицировать следующее.

- Описанные нами функции действительно соответствуют потребностям.
- Производные от этих функций прецеденты и требования действительно поддерживают данные функции.
- Прецеденты реализуются при проектировании.
- Проектирование поддерживает функциональные и нефункциональные аспекты поведения системы.
- Код действительно соответствует результатам и целям проектирования.
- Тесты обеспечивают полное покрытие разработанных требований и прецедентов.

Итак, как узнать, что собой представляет наша разработка в целом? Нужен метод, который позволит гарантировать, что мы провели верификацию всего необходимого (и ничего лишнего!).

Для этого нужен план верификации и (желательно) ценные автоматические средства, которые помогут выполнить его. Но самое главное, команде необходимо понять, что такая верификация, и взять на себя ответственность за ее проведение. Верификация – это *не только* деятельность, осуществляемая группой обеспечения качества проекта (QA team), которая, безусловно, играет огромную роль в этом процессе.

Одним из методов осуществления постоянного контроля верификационных действий является *трассировка (traceability)*. Мы уже упоминали о трассировке в части 5, а в главе 31 обсудим, как ее можно использовать для организации верификационных действий.

Независимо от того, как организована верификация, необходимо помнить, что суть ее в том, чтобы убедиться, что шаг, над которым мы работаем, имеет соответствующую предысторию и выполняется непротиворечивым и надежным способом. Более того, необходимо удостовериться, что каждое предпринимаемое нами действие необходимо, а не нужные шаги отсутствуют.

В значительной мере этому способствует применение эффективных процессов разработки программного обеспечения, а также осуществление анализа. Например, можно исследовать требования и убедиться, что они корректно, полно и сжато выражают более высокоровневые потребности пользователей. Затем можно убедиться, что проектирование проводилось на основании требований и прецедентов и полученный технический проект является полным и не имеет лишних элементов. В некоторых ситуациях анализ сокращается до *просмотров и ревизий*. В других случаях можно использовать модели и автоматизированные средства, чтобы проверить полноту, семантику и т.д. Напоминаем, что на данном этапе мы не пытаемся проверить работоспособность, мы займемся этим позднее. Сейчас же необходимо убедиться, что мы сделали то, что должны были сделать, и это следует логике разработки.

Итак, достаточно отступлений. Вернемся к обсуждению некоторых аспектов верификации.

Затраты на верификацию

Неприятным моментом верификации является то, что можно потратить время на верификационную деятельность, которая не принесет отдачи. Поэтому нужен некий способ вычисления “экономических дивидендов” (*return on investment, ROI*) верификационной деятельности. Необходим подход, который поможет *правильно* вложить средства в проверку наших действий. Не хочется выполнять лишние проверки, но нельзя пропускать проверку жизненно важных аспектов. В главе 33 содержится краткое описание методов оценки и анализа рисков и предлагаются рекомендации по использованию этих методов для экономии средств при проведении проверок.

Команда должна сконструировать и реализовать систему, соответствующую требованиям. Другими словами, команда должна иметь возможность убедиться, что планы реализации соответствуют потребностям.

Но как на основании требований осуществить проектирование и реализацию? Нельзя же передать требования компьютеру и ждать, что проектирование и реализация произойдут сами собой! В главе 30 рассматриваются способы перехода от разработки требований к проектированию и описывается, как можно использовать полученные в процессе создания требований артефакты при проектировании и реализации системы.

Верификация на всех уровнях

Верификацию можно применять на всех стадиях жизненного цикла разработки; методы и ориентированы на какую-либо одну фазу. Можно (и нужно!) верифицировать элементы требований, проектирования, реализации, тестирования и другие важные элементы разработки, основываясь на результатах ROI-анализа. В той или иной степени верификацией придется заниматься на *каждой* стадии разработки. Она необходима, чтобы удостовериться в правильности следующих переходов.

- От потребностей пользователя к функциям продукта
- От функций продукта к требованиям
- От требований к архитектуре
- От архитектуры к модели проектирования
- От модели проектирования к реализации
- От реализации к планированию тестов

(Замечание. Мы будем рассматривать тестирование как часть проверки правильности (validation).)¹

Хотелось бы вновь подчеркнуть, что верификация чрезвычайно важна на стадии проектирования, так как возникшие на этой стадии ошибки очень сложно устранять на стадии реализации.

Доводы в пользу верификации

Мы рекомендуем во всех разработках проводить процесс верификации, который должен начинаться в начале разработки проекта и продолжаться на протяжении всего ее жизненного цикла. При работе над проектом, в котором процесс разработки регулируется, верификацию, вероятно, придется проводить независимо от вашего желания. Верификация обязательна при разработке систем высокой надежности (систем жизнеобеспечения или таких систем, стоимость сбоя в которых недопустимо высока), в противном случае вы подвергаете недопустимому риску себя или других. Но каждый нетривиальный проект лишь выигрывает от хорошо спланированной верификации.

Проверка корректности результатов разработки

По мере достижения важных вех (этапов) разработки, таких как исполняемые итерации, важно проверить правильность созданных функциональных возможностей, т.е. убедиться, что они работают корректно и соответствуют требованиям. Мало пользы в прохождении определенного этапа, если созданные части не работают так, как предполагалось, хотя даже в этом случае можно кое-чему научиться.

Особое внимание следует уделять этапу завершения проекта. В успешном процессе это просто еще один шаг, подтверждающий, что система сконструирована и реализована в соответствии с потребностями клиента. Кроме того, на этом этапе необходимо показать, что система действительно работает так, как предполагалось.

Последней контрольной точкой является демонстрация работы системы в среде клиента.

В процессе разработки существует последняя и очень важная контрольная точка. Нужно продемонстрировать, как продукт работает в среде клиента, и продукт должен понравиться заказчику настолько, чтобы тот принял его. Суть в том, чтобы по окончании проекта клиент был доволен или, в крайнем случае, не слишком недоволен.

Работа по проверке правильности разработки (*validating*) имеет два важных аспекта: (1) показать, что продукт соответствует предъявляемым к нему требованиям и (2) сосредоточить внимание на приемке клиентом конечного результата. Хотя теоретически это

одно и то же, на практике заключительные приемочные испытания часто демонстрируют, что мы следовали требованиям, но потребности пользователя “сместились” по сравнению с более ранними стадиями проекта. В главе 32 обсуждаются оба аспекта проверки правильности, а также высказываются различные мнения о том, что она подразумевает.

Как и при верификации, здесь тоже нужен способ, позволяющий убедиться, что затраты времени и средств на проверку правильности или тестирование являются эффективными. И снова в этом может помочь анализ дивидендов (ROI).

Обработка изменений, возникающих в процессе разработки

Наконец, необходимо рассмотреть влияние изменений требований. Приходилось ли вам когда-нибудь работать над системой, в которой требования *ни разу* не менялись с первого дня?

Необходимо строить свои планы с учетом возможности изменений и знать, как этими изменениями управлять. В главе 34 обсуждается возможное разрушительное влияние неконтролируемых изменений и предлагается организованный способ их выявления, оценки воздействия, а также их внесения упорядоченным образом.

Далее...



Начнем с того, как на основании требований осуществить проектирование и реализацию решения имеющейся проблемы. Этому посвящена следующая глава.

A

Глава 30

От понимания требований к реализации системы

Основные положения

- Многие требования достаточно легко отобразить в технический проект и реализаций код.
- Существуют требования, слабо связанные с проектированием и реализацией; форма требования существенно иная; возникает проблема ортогональности.
- Смягчить проблему ортогональности можно посредством применения объектно-ориентированных методов и прецедентов.
- При проектировании с использованием прецедентов все заинтересованные лица получают возможность исследовать предложенную реализацию системы на соответствие вариантам использования и требованиям.
- Хороший проект системы не обязательно оптимален в том, что касается возможности видеть, где реализовано каждое конкретное требование.

Мы создавали сложные программные системы на протяжении более 40 лет. В истории отрасли были трудности и неудачи, но были и несомненные успехи: интерактивная торговля, обработка текстов, медицинские системы жизнеобеспечения, безопасные энергетические установки и многое другое.

Очевидно, что мы должны были как-то научиться переходить из мира требований к проектированию и реализации. Мы имеем опыт реализации многих сложных систем, соответствующих предъявляемым к ним требованиям. Однако когда дело касается построения сложных систем, требующих высокой степени надежности проекта, это не всегда приятное или, по крайней мере, строго научное занятие. Причина в том, что не так-то просто увидеть проверяемые требования внутри реализации. Доказательство того, что требования осуществляются в программном коде, является нетривиальной задачей.

Отображение требований в технический проект и программный код

К счастью, для значительного количества требований относительно несложно разработать программу, которая непосредственно преобразует их в технический проект, а за-

тем и программный код. Это также означает, что можно протестировать значительную часть кода, используя тесты на соответствие определенного модуля определенному требованию, так как будет обеспечена высокая корреляция между формулировкой требования и реализующим его кодом. Например, достаточно несложно обнаружить и проверить правильность кода, выполняющего требование "Поддерживать входные параметры с плавающей точкой до восьми знаков включительно" или "Показывать ход компиляции пользователю" (рис. 30.1). В зависимости от типа создаваемой системы данный подход можно применять для значительной части программного кода, и в этом случае процесс преобразования требований в технический проект и реализацию не столь уж сложен.

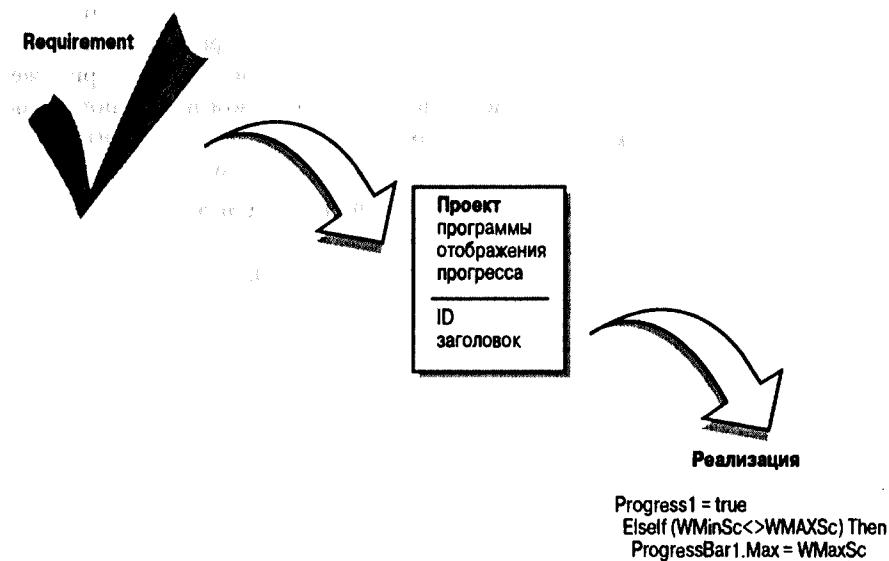


Рис. 30.1. Преобразование требований в проект и реализацию

Проблема ортогональности

Но когда дело касается требований вида "Система должна обслуживать до 100 тысяч торговых операций в час" или "Система должна разрешать редактирование, основываясь на

 Требование: 100000 торговых операций в час

 Код

уровне безопасности, установленном системным администратором", все становится несколько сложнее. Эти требования сложно связать с проектом системы и реализацией кодом; они ортогональны или практически ортогональны. Не существует соответствия один-к-одному, которое значительно упрощает реализацию и проверку правильности. И тому есть несколько причин.

- Требования говорят об элементах реального мира, таких как двигатели и чеки, в то время как код оперирует стеками, очередями и алгоритмами вычислений. Это два совершенно различных языка.
- Определенные требования, такие как требования производительности, имеют мало общего с логической структурой кода, но тесно связаны со структурой обработки (как взаимодействуют различные части кода, насколько быстро выполняется определенный

фрагмент, сколько прерываний поступает при нахождении в модуле А и т.д.). Когда физически невозможно отобразить требование в логическую структуру, то невозможно и "указать", где в реализации отражено данное требование.

- Для осуществления функций, предусмотренных некоторыми функциональными требованиями, необходимо взаимодействие нескольких элементов системы. Рассмотреть часть – это ие то же самое, что рассматривать целое; реализация требования оказывается распределенной по нескольким фрагментам кода.
- Возможно, самым важным является то, что при проектировании системы зачастую руководствуются не соображениями легкости доказательства того факта, что некое требование удовлетворяется, а другими, гораздо более важными мотивами. Например, стараются оптимизировать дефицитные ресурсы с помощью архитектурных образцов (которые хорошо зарекомендовали себя в других приложениях, но не полностью соответствуют настоящему), посредством повторного использования программного кода, а также применения закупаемых компонентов, которые имеют собственное управление и функциональное поведение и т.д.

Те из нас, кто создавал системы высокой надежности и/или должен был по условиям контракта продемонстрировать на бумаге прямую корреляцию между требованиями и программным кодом, конечно, старались сделать это. Но, надо признаться, такая демонстрация состояла частично из механизмов трассировки реальных (важных и серьезных) требований, а частично – чего-то эфемерного.



Объектно-ориентированный подход

Проблему ортогональности (которая заключается в отсутствии прямой связи между отражающими проблемную область требованиями и реализацией кодом) удалось смягчить благодаря внедрению объектно-ориентированных (ОО) методов. При использовании ОО-методов создаваемые кодовые сущности теснее связаны с проблемной областью, что, как следствие, повышает степеньробастности. Это происходит не только благодаря объектно-ориентированным принципам абстрактности, сокрытия информации, наследования и т.п., но и благодаря тому, что сущности реального мира просто изменяются не так часто, как транзакции и данные, которые мы раньше использовали для моделирования нашей системы. Поэтому код также изменяется реже. (Например, люди до сих пор получают расчетные листы, как и 40 лет назад, хотя их форма – электронные вместо бумажных – изменилась коренным образом.)

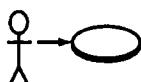
Применяя объектно-ориентированный подход, можно отыскать в программном коде объекты, ответственные за реализацию механизма расчета, и объекты платежных ведомостей, а затем использовать их при верификации требований. Можно посмотреть на требования для "расчетного листа" и увидеть, поддерживаются ли предполагаемые операции и атрибуты в модели проектирования.

Но нужно быть внимательным, так как благое намерение обеспечить взаимно однозначное соответствие между требованиями и кодом может привести к функционально-организованной архитектуре, совершенно не соответствующей объектно-ориентированному подходу. Согласно основным принципам ОО-методологии, разработчик должен описать небольшое число механизмов, выполняющих ключевые требования системы. В результате получается множество классов, взаимодействие которых приводит

к более содержательному поведению, чем просто сумма поведений отдельных классов. Это "более содержательное поведение" должно обеспечить более устойчивый и способный к расширению проект, чем могут дать текущие (а в идеале, и будущие) требования *в сумме*, но оно не является взаимно однозначным отображением требований. Таким образом, даже при следовании объектно-ориентированным принципам ортогональность требований все же сохраняется.

Прецедент в роли требования

Обособленный характер отдельных требований может еще более усложнять проблему ортогональности. Каждое требование само по себе может и не представлять серьезной проблемы, но сложно становится рассматривать поведение системы в целом и определять, действительно ли она делает все, что нужно. Как нам исследовать систему, чтобы


определить, действительно ли за требованием 3 (отобразить полоску, отражающую выполнение) непосредственно следует требование 7 (во время компиляции алгоритм состоит...)?

Прецедент, который описывает последовательность действий системы и пользователя, а не отдельные требования, значительно упрощает данную проблему. В этом случае сами требования, представленные в виде прецедентов, гораздо лучше отражают последовательность действий системы вместе со всеми альтернативами и исключительными ситуациями. Прецеденты просто "лучше рассказывают" о том, что должна делать система. Они также содержат важнейшую информацию для начала проектирования.

Осуществление перехода

Итак, хотя мы и не решили проблему ортогональности, у нас все же есть определенные активы и несколько новых методов, которые могут помочь смягчить эту проблему. Если мы сможем с их помощью увеличить параллельность между требованиями и кодом, это позволит нам использовать понимание требований при проектировании системы. При этом также будет проще осуществить переход между этими различными мирами, улучшить технический проект и общее качество полученной в результате системы. Однако прежде необходимо совершить небольшой экскурс в мир моделирования и программной архитектуры.

Моделирование систем программного обеспечения

Современные системы программного обеспечения чрезвычайно сложны. Не являются исключением системы или приложения, состоящие из миллионов строк программного кода. Они могут в свою очередь встраиваться в другие системы, которые чрезвычайно сложны сами по себе, не говоря уже о сложности возникающих между ними взаимодействий. Вероятно, ни один человек или даже группа людей не в состоянии понять все детали каждой из этих систем и их планируемые взаимодействия.

При таком уровне сложности полезно представить систему в виде абстрактной упрощенной модели, отбросив незначительные детали, чтобы получить более понятную версию. Цель моделирования состоит в том, чтобы упростить систему до понимаемой

- Клиент вводит в банкомат карточку
- Система запрашивает PIN-код
- Клиент вводит PIN-код

“сущи”. Но важно не переусердствовать, чтобы модель не перестала адекватно представлять реальную систему. Итак, модель позволяет думать о системе, не путаясь в деталях.

Выбор модели – очень важный вопрос. Нужно, чтобы модель помогла нам надлежащим образом понять систему, и мы не хотим, чтобы она ввела нас в заблуждение из-за допущенных в ней ошибок или принятых абстракций. Всем известны рисунки и приспособления, которые помогали древним философам, астрономам и математикам понять движение тел Солнечной системы. Многие из них основывались на *геоцентрическом* представлении о Солнечной системе, согласно которому Земля является неподвижным центром Вселенной, что в результате приводило к неправильным теориям. Только после того, как были предложены *гелиоцентрические* модели, удалось лучше понять природу нашей Солнечной системы.

Благодаря гелиоцентрическим моделям Вселенной возникли новые возможности и идеи, касающиеся исследования Вселенной в целом. Ученые могли, исходя из них, предлагать уточненные математические теории относительного движения, гравитации и т.д. Но модель не является реальностью. В некоторых случаях основанные на модели механические представления Вселенной не совсем совпадают с наблюдаемыми явлениями. Например, одним из первых подтверждений теории относительности Эйнштейна были наблюдавшиеся и прежде не находившие объяснения аномалии орбиты планеты Меркурий.

Помните, модель не является реальностью!

Модели обеспечивают возможность обсуждать сложную проблему и получать полезные гипотезы. Но необходимо помнить, что модель – это не реальность, и все время следить, чтобы она не ввела нас в заблуждение.

Можно моделировать различные аспекты системы. В зависимости от того, что нас интересует в данный момент, можно создать модель параллельной обработки или модель логической структуры системы. Эти модели должны как-то взаимодействовать, и этот аспект также можно моделировать. Каждая из этих моделей вносит свой вклад в наше понимание системы, а вместе они позволяют нам рассматривать *архитектуру системы* в целом.

Архитектура систем программного обеспечения

Рассмотрим определение Шоу и Галана (Shaw, Garlan, 1996).

Программная архитектура представляет собой описание элементов, из которых конструируется система, взаимодействий между ними, а также образцов, согласно которым они составляются, и ограничений, налагаемых на эти образцы.

Согласно Крачтену (Kruhten, 1998), с помощью архитектуры решаются следующие задачи.

- Понять, что делает система
- Понять, как она работает
- Иметь возможность обдумывать и разрабатывать части системы
- Расширять систему
- Повторно использовать часть (части) системы при создании новых систем

Архитектура является средством, с помощью которого принимаются решения о том, как будет строиться система. Зачастую мы с самого начала разработки знаем, как будем

осуществлять сборку частей системы, так как мы (или другие) уже разрабатывали аналогичные системы раньше. Простые стартовые решения обозначаются понятием “доминирующая архитектура”, которое есть не что иное, как красивое выражение мысли “все знают, как создавать систему обработки платежных ведомостей”.

Доминирующая архитектура помогает придать начальный импульс процессу принятия решений и минимизирует риск посредством повторного использования частей успешного решения. Создавая систему обработки платежных ведомостей, было бы глупо начинать с нуля и изобретать концепции социального страхования, выписки чеков и медицинских отчислений. Следует начать с изучения моделей существующих систем и использовать их в своих рассуждениях.

Различным участникам нужны различные представления системы.

Архитектурные модели нужны различным группам заинтересованных лиц, которые будут рассматривать предложенную архитектуру с разных точек зрения. Так, например, при строительстве дома нужно иметь его представления, предназначенные для монтажников, кровельщиков, электриков, водопроводчиков и т. д. Это все один и тот же дом, но мы можем рассматривать его по-разному, в зависимости от потребностей.

4+1 представление архитектуры

Как правило, при рассмотрении системной архитектуры имеется небольшой набор общих потребностей. Наши коллеги Филипп Краутен (Philippe Kruchten, 1995) указал в своей книге 4+1 представление (или вид, views), которые лучше всего иллюстрируют эти обобщенные потребности. Эти виды изображены на рис. 30.2; различные заинтересованные лица (программисты, руководство, пользователи) расположены возле тех представлений, которые они обычно используют.

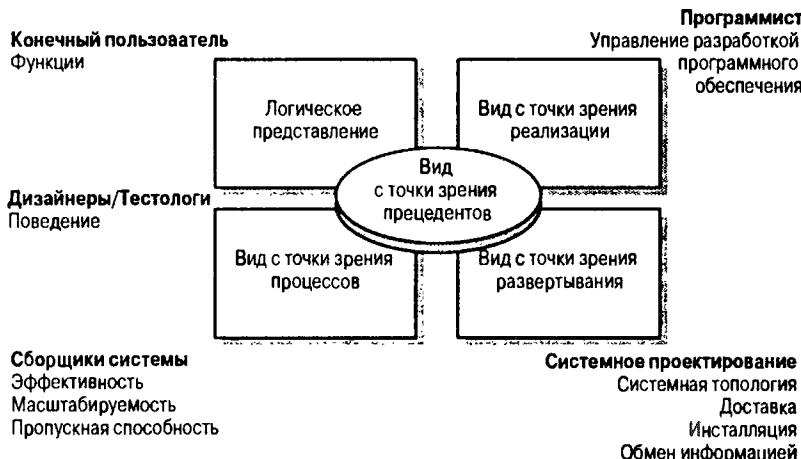


Рис. 30.2. 4+1 представление системной архитектуры

1. *Логическое представление (Logical view)* характеризует функции системы. Эта абстракция модели проектирования представляет логическую структуру системы с по-

мощью подсистем и классов, которые в свою очередь являются сущностями, предоставляющими функциональные возможности пользователю.

2. *Вид с точки зрения реализации (implementation view)* описывает компоненты, относящиеся к реализации системы: исходный код, библиотеки, классы объектов и т.д. Это статическое представление данных компонентов, не описывающее их взаимодействие.
3. *Вид с точки зрения процессов (Process view)* больше подходит для описания операций системы. Такие представления чрезвычайно важны для систем, в которых имеются параллельные задачи, интерфейсы с другими системами, а также другие взаимодействия, возникающие в ходе выполнения. Так как многие современные системы отличаются высокой степенью параллелизма и многопоточной обработкой, данный вид позволяет рецензенту определить потенциальные проблемы, такие как условия состязания или блокировки. Можно также использовать вид с точки зрения процессов для исследования пропускной способности и других вопросов производительности, которые пользователь указал в иефункциональных требованиях.
4. *Вид с точки зрения развертывания (Deployment view)* демонстрирует размещение элементов реализации по поддерживающей их инфраструктуре, состоящей из операционных систем и вычислительных платформ. В этом представлении внимание уделяется не столько тому, какими являются взаимодействия, сколько тому, что взаимодействия и ограничения существуют там, где встречаются две системы.

Роль модели прецедентов в архитектуре

Вернемся к проблеме ортогональности. *Вид с точки зрения прецедентов* играет особую роль в архитектурной модели, выступая в качестве хранилища требований. В нем представлены основные прецеденты модели прецедентов, поэтому он используется в процессе проектирования, а также объединяет все архитектурные представления. Мы выделяем этот вид, поскольку он позволяет всем заинтересованным лицам исследовать планы реализации на соответствие реальному использованию и требованиям к системе. Итак, вид с точки зрения прецедентов, представляющий функциональные возможности системы, является "связующим звеном", объединяющим остальные виды.

Например, прецедент "Инициация последовательности действий в чрезвычайной ситуации" системы HOLIS повлияет на проектирование системы в каждом из четырех представлений следующим образом.

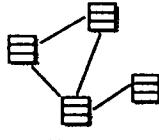
1. *Логическое представление* будет описывать различные классы и подсистемы, реализующие варианты поведения, предусмотренные чрезвычайной последовательностью действий.
2. *Вид с точки зрения процессов* будет отражать, каким образом способность HOLIS к многозадачности позволит ей начать чрезвычайную последовательность в любой момент, даже тогда, когда производится программирование системы или она занята выполнением других задач.
3. *Вид с точки зрения развертывания* будет показывать, как функциональные возможности HOLIS распределяются по трем ее узлам или подсистемам ("Управление включением", "Центральный блок управления" и "ПК домовладельца").



4. Вид с точки зрения реализации будет содержать описание различных кодовых артефактов HOLIS, в том числе исходных и выполняемых файлов.

Реализация прецедентов в модели проектирования

“Основанное на прецедентах проектирование” является основной темой Унифицированного языка моделирования (Unified Modeling Language) и посвященной ему книги



Модель проектирования

Unified Software Development Process (Джейкобсон (Jacobson), Буч (Booch), Рамбо (Rumbaugh), 1999). Предлагаемый в рамках UML и унифицированного процесса метод позволяет группе разработчиков перейти от понимания требований к проектированию и реализации решения.

Язык UML содержит специальные модельные конструкции, которые поддерживают *реализацию (realizing)* прецедентов. Прецеденты реализуются посредством *коопераций (collaborations)*, которые представляют собой сообщества классов, интерфейсов, подсистем или других элементов, кооперируемых для получения определенного поведения. Для этого используется UML-стереотип, *реализация прецедента (use-case realization)*. Он является просто специальной формой кооперации, которая показывает, как функциональные возможности конкретного прецедента осуществляются в модели проектирования.

Таким образом, кооперации являются основными модельными конструкциями в рассматриваемой области, так как именно внутри них заключены общие поведенческие аспекты системы (т.е. механизмы достижения системой своих глобальных целей). Посредством деятельности классов-участников и других логических элементов эти конструкции дают более сложное поведение, чем просто “сумма вариантов поведения их составляющих”. Графическим символом кооперации является эллипс, нарисованный пунктирной линией, внутри которого указано имя (рис. 30.3). (Авторы UML отметили, что сходство с обозначением прецедента не случайно.)

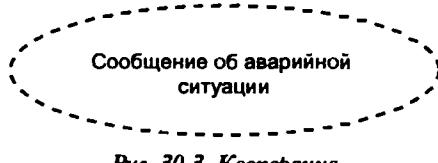


Рис. 30.3. Кооперация

Кооперации имеют еще одно полезное свойство. С их помощью можно осуществлять трассировку модели прецедентов к модели проектирования, как показано на рис. 30.4.

Структурная и поведенческая части коопераций

Кооперации имеют *структурную часть*, задающую статическую структуру системы (классы, элементы, интерфейсы и подсистемы), и *поведенческую*, задающую динамику взаимодействия элементов для получения результата. Однако кооперация не является физическим предметом, это всего лишь описание того, как совместно работают коопери-

рованные элементы системы. Чтобы больше узнать о том, как действует кооперация, необходимо рассмотреть ее внутреннее строение.

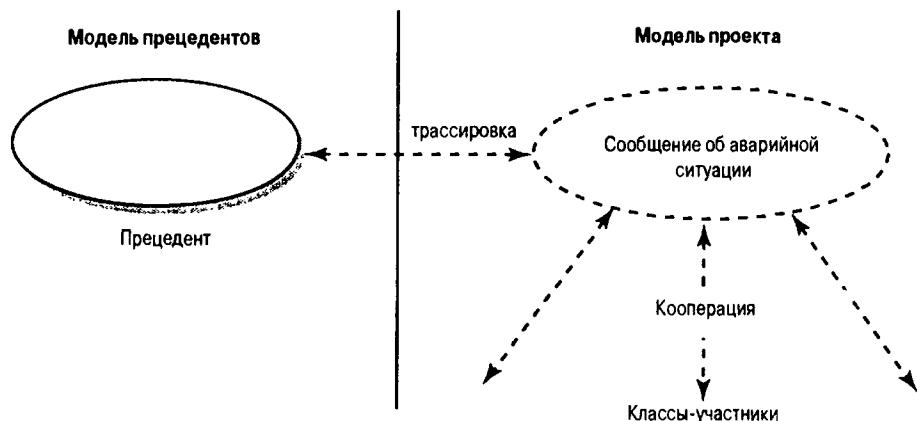


Рис. 30.4. Реализация прецедента в модели проектирования

Внутри кооперации структурные элементы можно представить с помощью диаграммы классов. На рис. 30.5 показана диаграмма классов для кооперации “Последовательность действий системы HOLIS при создании чрезвычайных сообщений”. Для моделирования поведенческих аспектов кооперации используются диаграммы взаимодействий, подобные представленной на рис. 30.6.

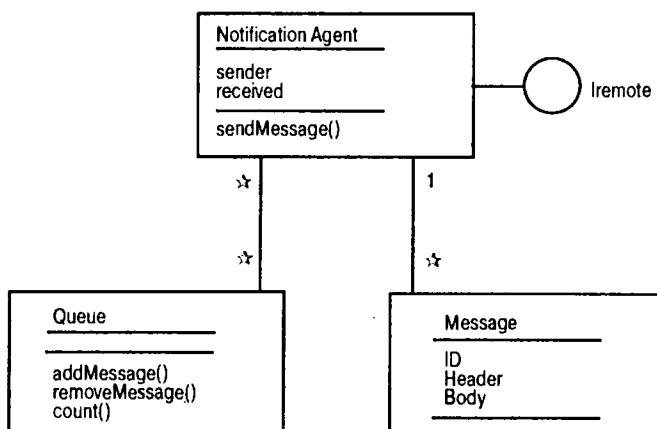


Рис. 30.5. Диаграмма классов кооперации “Последовательность действий системы HOLIS при создании чрезвычайных сообщений”

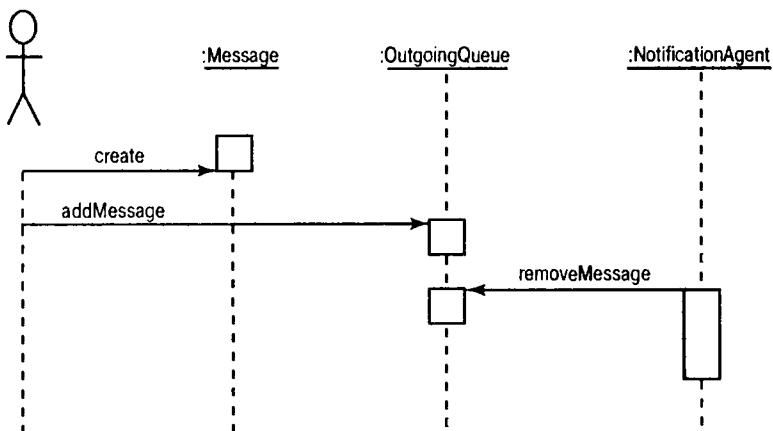


Рис. 30.6. Поведенческие аспекты кооперации “Последовательность действий системы HOLIS при создании чрезвычайных сообщений”

Использование коопераций для реализации наборов отдельных требований

С помощью кооперации можно моделировать реализацию не только прецедента, но и любого отдельного требования или набора требований (рис. 30.7). Несмотря на то что прецеденты имеют некие особые свойства (а именно последовательность событий), всегда можно так составить перечень требований, чтобы добиться той же цели. Таким образом, у нас есть способ, позволяющий использовать требования всех типов для проектирования и реализации.

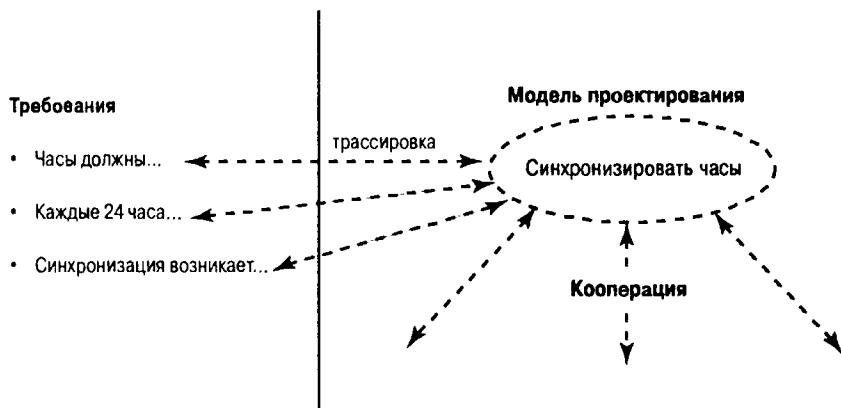


Рис. 30.7. Модель реализации требований в виде реализации прецедента

От проектирования к реализации

Моделируя систему подобным образом, можно гарантировать, что важные прецеденты и требования надлежащим образом отражены в модели проектирования. В свою оче-

редь, это помогает удостовериться, что проект программы соответствует требованиям, а это является значительным шагом в процессе *верификации проектирования*.

Следующий шаг совершенно логичный, хотя и непростой. Классы и объекты, определенные в модели проектирования, реализуются посредством физических программных компонентов (исходных, двоичных, выполняемых файлов и т.п.), которые будут использоваться для создания выполняемой программы. Но даже в этом отображении есть свои сложности. Например, решения, которые приводят к определенному разбиению логической модели на компоненты, часто зависят от таких требований, как гибкость, производительность, ограничения на топологию развертывания системы и т.д. Главным является то, что на каждое представление системной архитектуры влияют или, по крайней мере, должны влиять системные требования. Если помнить об этом, то мы сумеем завершить процесс перехода от понимания требований к проектированию, а затем — к реализации системы.

Заключение

Было бы замечательно, если бы можно было непосредственно перейти от требований к программному коду. К сожалению, так не бывает. Лучшее, что может предложить современная практика, — это набор конструкций, которые помогают приблизиться к цели непосредственного преобразования.

Один из этих методов, *реализация прецедентов*, использует преимущества уникальных характеристик прецедента и модельных конструкций языка UML для того, чтобы упростить проектирование и значительно сократить путь от понимания требований до реализации.

Еще один современный подход предлагает нам рассматривать разработку с помощью “4+1” представлений архитектуры, что позволяет обеспечить разделение интересов. При этом различным участникам процесса реализации становится проще работать над техническим проектом и получать доступ к нему по мере его развития.

Далее...



Мы признаем, что лишь едва коснулись темы реализации, которая сама по себе является весьма обширной. К счастью, данная книга не о реализации программных систем, поэтому нам нет необходимости погружаться в эту проблему. Мы оставляем тему реализации и переходим к тому, что объединяет все эти элементы разработки в монолит, — к *верификации и проверке правильности* (*verification and validation, V&V*). В последующих нескольких главах мы рассмотрим некоторые подходы, разработанные для минимизации внесения ошибок на разных стадиях разработки, а также опишем методы *тестирования и проверки правильности*, которые помогут удостовериться, что система осуществляет свои цели.

Верификация и проверка правильности играют важную роль в достижении цели разработки качественного программного обеспечения. В частности, действия по верификации удерживают нас от того, чтобы заниматься “тестированием качества в продукте”. Зачастую результаты разработки не оправдывают ожиданий из-за первого представления о том, что “все дефекты можно выявить в процессе тестирования”. Это вряд ли возможно. Выявлять и исправлять ошибки гораздо дороже, чем просто не вносить их с самого начала.

Мощным средством, помогающим проводить верификацию и минимизировать выявление случайных ошибок, является трассировка. В свою очередь, проверка правильности (validation) помогает направить тестирование на выявление все же “просочившихся” в систему дефектов.

Глава 31

Использование трассировки для поддержки верификации

Основные положения

- Трассировка является эффективным методом поддержки верификации.
- Программные требования трассируются от одной или нескольких функций продукта, заданных в документе-концепции.
- Автоматические средства трассировки позволяют проверять верификационные отношения, чтобы удостовериться, что никакие необходимые отношения не пропущены и нет лишних верификационных отношений.
- Сами по себе автоматические средства не могут выполнить всю работу; верификация требует размышлений.

Мы рекомендуем проводить верификацию постоянно, чтобы убедиться, что каждый шаг разработки корректен, удовлетворяет потребности следующего шага и не содержит лишних действий. Данная глава посвящена методам трассировки, которые можно использовать для поддержки всех стадий верификации проекта, начиная с самых первых шагов.

Роль трассировки при верификации требований

Важным показателем качества реализации программы является возможность ее трассировки на стадиях спецификации, архитектуры, проектирования, реализации и тестирования. Способность отслеживать отношения и находить их связи при возникновении изменений является основной нитью многих современных высоконадежных программных процессов, особенно в медицине (устройства жизнеобеспечения) и других критически важных областях. Причина в том, что, как свидетельствуют данные, касающиеся безопасности, воздействие изменения часто в полной мере не отслеживается и незначительные изменения системы могут вызвать значительные проблемы с безопасностью и надежностью.

Инструкции Управления по санитарному надзору за продуктами и медикаментами США (FDA) (FDA Office of Device Evaluations (ODE) Guidance Document (1996, б) и FDA Current Good Manufacturing Practices (CGMP) (FDA 1996, а) указывают на необходимость трассировки при проведении действий по разработке программного обеспечения для медицинских целей.

Стандарт IEEE (1994) предлагает два рабочих определения трассировки.

- “Степень, до которой можно установить связь между двумя или большим числом продуктов процесса разработки, особенно продуктами, которые являются по отношению друг к другу предшествующим-последующим или главным-подчиненным; например, степень соответствия между требованиями и проектом конкретного программного компонента” (IEEE 610.12-1990 §3).
- “Степень, до которой каждый элемент продукта программной разработки определяет смысл своего существования; например, насколько каждый элемент схемы, изображаемой кружками и стрелками, соответствует удовлетворяемому им требованию” (IEEE 610.12-1990 §3).

Ключевым элементом трассировки является “отношение трассировки” (traceability relationship). Удобно определить это отношение с помощью простой модели, использующей понятия “трассируется к” (traced-to) “трассируется от” (traced-from). Например, одно или несколько требований к программному обеспечению создаются с целью поддержки некой функции, заданной в документе-концепции. Можно сказать, что программное требование трассируется от некоторой функции (рис. 31.1).

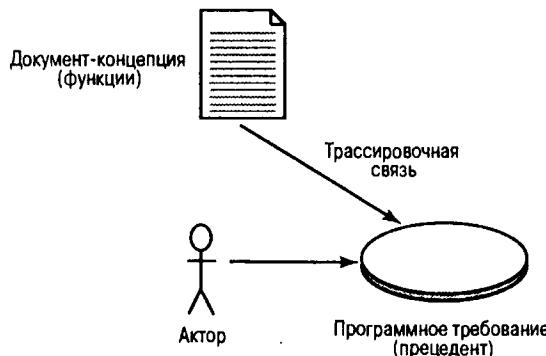


Рис. 31.1. Трассировочная связь от документа-концепции к программному требованию

В зависимости от типов создаваемых требований данное отношение приобретает дополнительный смысл. Например, то, что некое требование к программному обеспечению “трассируется к” определенному тестовому примеру, означает, что данное требование “тестируется” этим тестовым примером. То, что описание объекта “трассируется от” конкретного программного требования, подразумевает, что это требование “реализуется” указанным объектом. Между этими элементами проекта имеются отношения вида один-ко-многим, многие-к-одному и многие-ко-многим.

Основываясь на том, что говорилось о различных средствах выражения требований и их организаций (глава 25), можно считать, что структура проекта с полным набором отношений будет выглядеть так, как показано на рис. 31.2.

Неявная и явная трассировка

На рис. 31.2 видно, что команда разработчиков явным образом задает отношения между элементами. Это *явная трассировка* — разработка отношений, основанная на соображениях команды. Например, связь, или отношение, между функцией продук-

та и прецедентом, осуществляющим поддержку этой функции, определяется исключительно решением команды о том, что такое отношение имеет смысл. Не существует внутренне присущей элементам связи между ними; только внешние решения могут привести к заданию этой связи.

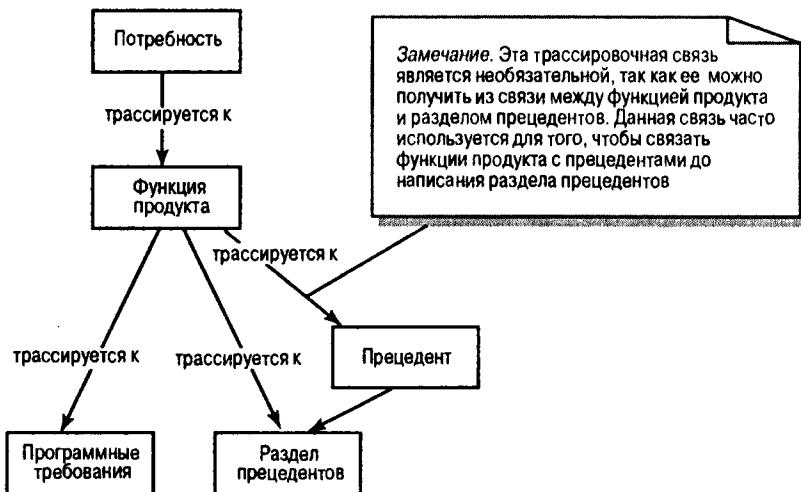


Рис. 31.2. Отношения трассировки между элементами проекта

Структура модели может обеспечить неявные отношения трассировки.

С другой стороны, методология и структура модели могут задавать *неявные отношения трассировки*. Например, в части 5 обсуждалось понятие "дочерних" требований; между ними и "родительскими" требованиями существуют формальные иерархические отношения. В этом случае существуют неявные отношения трассировки между родительскими и соответствующими дочерними требованиями. Такое неявное отношение нет необходимости задавать в виде явного; более того, его *не следует* задавать явно, иначе могут возникнуть недоразумения.

Неявная трассировка может также возникнуть вследствие использования определенной парадигмы моделирования. Например, применяемые в процессе разработки инструментальные средства моделирования могут автоматически обеспечивать отношения трассировки между моделируемыми элементами. Если средство моделирования обеспечивает неявные связи между элементами модели прецедента и взаимодействующими с прецедентом акторами, существует реальная возможность использовать эти неявные отношения трассировки. Можно продолжить эту трассировку далее к реализации, трассируя кооперации прецедентов к объектам реализации.

В конечном счете, различие между двумя классами трассировки невелико. Единственное, что мы хотели бы посоветовать, это убедиться, что вы знаете о всех формах трассировки, предлагаемых вашими автоматическими средствами моделирования. Если они действительно предлагают определенные формы неявной трассировки, используйте их. Если в интересующих вас областях средства не обеспечивают неявную трассировку, вам понадобится создать явные трассировочные связи, необходимые для поддержки ваших действий.

Дополнительные возможности, предоставляемые трассировкой

Трассировка зачастую помогает понять другие части проекта. Мы в своей практике часто вносили в проект менее традиционные элементы и включали их в процессы трассировки, поскольку они были полезны для понимания проекта.

Например, иногда полезно определить новый элемент, назвав его “спорный вопрос”, и хранить все текущие нерешенные вопросы как подобные элементы проекта. Используя трассировку, можно связать спорные вопросы с пунктами, где они упоминаются. Например, если есть нерешенный вопрос о функциональных возможностях продукта, можно связать его с соответствующими функциями продукта и требованиями к программному обеспечению. Поддерживая связи “спорных вопросов”, можно потом легко вернуться назад и найти все элементы проекта, связанные с вопросом, который только что решен. В процесс трассировки проекта можно также включить следующие элементы.

- Предположения и объяснения
- Элементы действий
- Запросы на новые/пересмотренные функции
- Термины глоссария и акронимы
- Библиографические ссылки

Другими словами, трассировку можно использовать для того, чтобы понять отюпнения между элементами проекта. Пусть воображение поможет вам добавить такие элементы, которые позволяют вашей команде лучше понять проект. Трассируйте все полезные отношения. Например, существует нерешенный вопрос относительно определения некоторого элемента глоссария. В таком случае можно связать вопрос с элементом глоссария (и, возможно, другими функциями). Это будет служить полезным напоминанием команде, что в проекте все еще существуют нерешенные вопросы. Дополнительные элементы можно добавить к типичной структуре трассировки, как показано на рис. 31.3.

Всегда следите за балансом важности элементов трассировки и затрат на их поддержку.

Предупреждение. Не следует злоупотреблять этими элементами. Мы обнаружили, что добавление слишком большого числа таких элементов в процесс трассировки делает его поддержку обременительной. Как и всегда, нужно стремиться к разумному балансу между важностью дополнительных элементов, которые вы хотите трассировать, и затратами на их поддержку.

Использование автоматических средств трассировки

Мощные инструментальные средства разработки программного обеспечения предлагают простые осуществляемые пользователем процедуры “указать и щелкнуть” для задания отношений между двумя элементами жизненного цикла. Мы интенсивно использовали в своей практике предлагаемое компанией Rational Software средство RequisitePro и

выбрали его для иллюстрации возможностей инструментальных средств в данной книге. Вы можете выбрать другое средство, но конечный результат будет аналогичен. (Более подробное описание артефактов проекта HOLIS, используемых в следующих примерах, содержится в приложении А.)

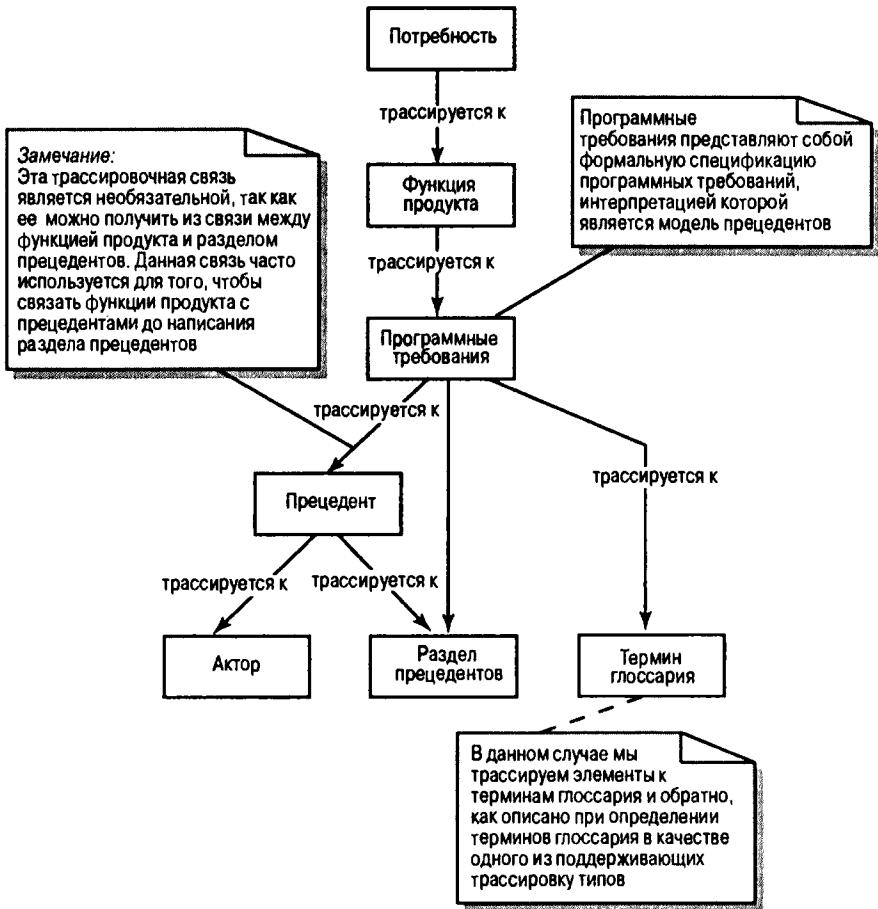


Рис. 31.3. Дополнительные отношения трассировки

Используя автоматическое средство, можно получить множество дополнительных сведений о проекте. Например, после того как определены отношения между функциями и требованиями к программному обеспечению системы HOLIS, можно отобразить матричную версию этих отношений, как показано на рис. 31.4.

Интерпретация представленной на рис. 31.4 матрицы трассировки достаточно очевидна. Рассмотрим пересечение FEA1 ("Легко программируемые станции управления") и SR3 ("HOLIS должна поддерживать до..."). Стрелочка в соответствующей ячейке указывает, что FEA1 трассируется к SR3; это означает, что SR3 некоторым образом осуществляет функцию FEA1.

Relationships: - direct only		SR1: System Clock The system...	SR2: On-line illumination	SR3: HQ1S shall support up to	SR4: Message protocol from	SR5: The CCU must have no	SR6: Include standard connector	SR7: In steady state mode when
FEA1: Easy to program control								
FEA2: Easy to install								
FEA3: Interface to home security...								
FEA4: Built-in security features ...								
FEA5: Vacation settings								
FEA6: 100% reliability								
FEA7: Instant lighting on/off								
FEA8: Easy to program, non-PC								
FEA9: Uses my own PC for...								

Рис. 31.4. Матрица отношений трассировки

После того как с помощью автоматического средства заданы все известные отношения, обязательным действием (выполнения которого требуют не только инструкции, но и наш собственный опыт) является проверка матрицы трассировки на наличие двух возможных индикаторов ошибок.

- Если при просмотре некой *строки* не удается обнаружить никаких отношений трассировки ("стрелок" нет), вероятно, еще не определено требование к программному обеспечению, отвечающее функции документа-концепции. Такая ситуация допустима, если функция не имеет отношения к программе ("Коробка должна быть из небьющегося пластика"). Тем не менее пустые строки являются индикаторами возможных ошибок и нуждаются в тщательной проверке. Современные инструментальные средства управления требованиями должны предоставлять возможность автоматического проведения такой проверки.
- Если в некотором *столбце* не оказывается отношений трассировки, вероятно, было создано требование к программному обеспечению, для которого нет требующей его функции продукта. Это может указывать на неправильное понимание роли программного требования, недостаток исходного документа-концепции, а также на то, что код неправильный, не соответствует системному требованию или является "сюрпризом" программиста, и в таком случае его следует немедленно удалить. (Мы будем более подробно обсуждать сюрпризы в главе 34.) В любом случае необходима тщательная проверка.

Поддержка отношений трассировки

Автоматическое средство разработки должно обеспечивать возможность производить опрос заданных отношений трассировки, а также запоминать эти запросы и вызывать их

впоследствии. Это позволит повторно обращаться к отношениям в будущем (например, после внесения изменений) и быстро проводить их повторную проверку, чтобы обнаружить точки, в которых могли возникнуть несоответствия. Как будет показано в главе 34, в процессе внесения изменений появление таких точек практически неизбежно.

Простое и очевидное применение этих методов позволяет связать многие элементы проекта. Следует рассматривать связывание следующих элементов.

- Программных требований и прецедентов с планами тестов/спецификациями тестов/результатами тестов
- Потребностей пользователей с функциями продукта
- Функций продукта с программными требованиями и прецедентами
- Программных требований и прецедентов с компонентами реализации, такими как функции, модули, объекты и кооперации
- Компонентов реализации с планами/спецификациями/результатами тестов

После связывания различных элементов различных документов, получится схема отношений, аналогичная изображенной на рис. 31.5.

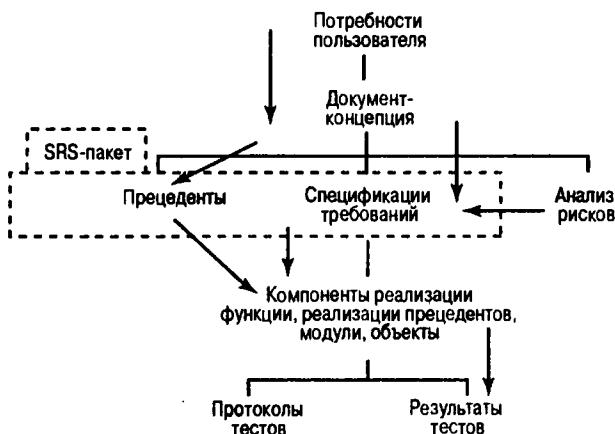


Рис. 31.5. Отношения различных документов и элементов

Инструментальное средство управления требованиями должно также предоставлять возможность отобразить полное множество отношений трассировки внутри проекта. На рис. 31.6 изображено такое “древовидное” представление. Заметим, что оно позволяет рассматривать одновременно все заданные к настоящему времени связи. Например, из рис. 31.6 видно, что FEA5 (“Установка режима длительного отсутствия”) связана с SR1, SR3 и UC1.

После того как установлены связи между элементами, инструментальное средство должно осуществлять поддержку этих связей. Это позволит использовать возможности данного средства для выборочной проверки отношений между элементами проекта. Возможно, вы уже догадались, что эти отношения трассировки можно применять для поддержки верификации.

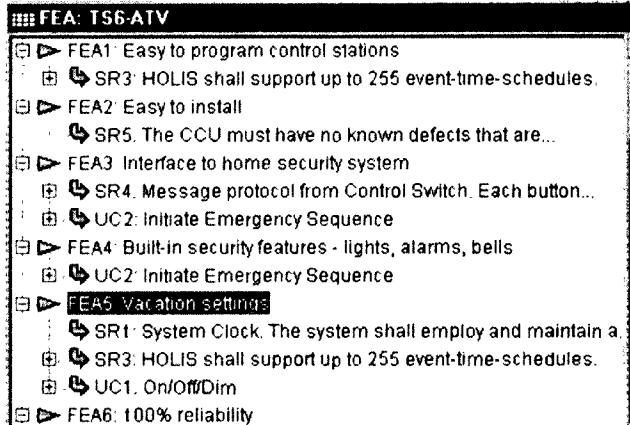


Рис. 31.6. Фрагмент “древовидного” представления

Работа без автоматических средств трассировки

Как действовать, если в вашем распоряжении нет автоматического средства, специально разработанного для поддержки определенных в предыдущем разделе операций?

В те времена, когда подобных средств еще не существовало, мы использовали для поддержки отношений трассировки электронные таблицы и базы данных. Многие матричные отношения можно легко обрабатывать с помощью простой электронной таблицы. В 1980-90-е годы мы активно пользовались электронными таблицами и считаем их весьма полезными.

Для реализации трассировки можно использовать электронные таблицы и базы данных.

Однако поддержка электронных таблиц представляет определенную проблему, особенно при общирных иерархиях отношений. Например, мы обнаружили, что изменение одной связи может оказать воздействие как на непосредственно затрагиваемые отношения, так и на отношения из других частей иерархии. Честно говоря, если нужно было выполнить масштабные изменения связей, это было настоящим кошмаром.

Из-за того, что было настолько сложно вручную отслеживать изменения и их “возмущающее воздействие”, мы пришли к выводу, что надо выбрать один из двух вариантов.

- Сопротивляясь всем предложенным изменить отношения
- Отказаться от матриц, так как работа становится просто невыносимой

Какой бы вариант мы ни выбрали,ineизбежно возникали проблемы в проекте. Представьте себе, с каким восторгом мы восприняли появление современных инструментальных средств, которые стали помогать нам в проведении этих действий!

Еще одной альтернативой было использование баз данных. Мы широко применяли реляционные базы данных и пришли к выводу, что создавать их и вводить в них данные нечто проще. До появления современных средств мы использовали их при работе над проектами разработки медицинских приборов, от которых зависела человеческая жизнь. Реляционные базы данных зарекомендовали себя достаточно хорошо. Несмотря на то что было сложно расширить приложение базы данных, чтобы отслеживать возмущающее

воздействие изменений, все же сделать это было возможно. Но проблема состояла в том, что мы слишком много времени уделяли совершенствованию возможностей нашего вспомогательного средства. Это отрицательно сказывалось на других участниках проекта, которым приходилось делать что-то другое.

Итак, для поддержки отношений трассировки можно использовать электронные таблицы или базы данных, хотя это совсем не просто. Если ваш проект небольшой, неудобства будут минимальны, и, может быть, имеет смысл рассматривать эти, более простые, средства. С другой стороны, мы не рекомендуем браться за более крупный проект, если в вашем распоряжении нет специализированных автоматических средств управления требованиями.

Пропущенные отношения

Чтобы обнаружить пропущенные отношения, надо искать *строки* матрицы трассировки, которые показывают, что некая функция не связана ни с одним программным требованием (прецедентом). Например, исследование матрицы, изображенной на рис. 31.7, показывает, что функция FEA4 не связана ни с одним программным требованием. (Хотя это и не отражено на рисунке, но она также не связана ни с каким прецедентом (UC).)

Relationships: - direct only		SR1: System Clock... SR2: On/Off illumination... SR3: HOIS shall support up to 100 nodes... SR4: Message protocol from... SR5: The CCU must have no more than 100 nodes... SR6: Include standard... SR7: In steady state mode...
FEA1: Easy to program control...		
FEA2: Easy to install...		
FEA3: Interface to home security...		
FEA4: Built-in security features -		
FEA5: Vacation settings		
FEA6: 100% reliability		
FEA7: Instant lighting on/off		

Рис. 31.7. Использование трассировки для обнаружения пропущенных отношений

При обнаружении "дыры" в отношениях нужно вернуться к исходному набору требований к продукту и связанным с ними программным требованиям / прецедентам.

- Может оказаться, что связь была случайно пропущена при задании трассировки. В таком случае простое добавление новой связи и пересчет матрицы трассировки решат проблему. Ошибки такого типа часто встречаются на ранних стадиях проекта.
- Но может выясниться, что при разработке программных требований просто не удалось учесть потребности одной из необходимых функций продукта. В таком случае следует или исправить проект и добавить подходящие требования, отве-

чающие данной функции, или поместить пропущенную функцию в категорию “будущих” либо удалить вовсе.

В любом случае при верификации важно убедиться, что ни одна связь не пропущена и все элементы более низкого уровня, такие как программные требования/прецеденты, надлежащим образом связаны с более высокоуровневыми требованиями к продукту.

“Лишние” отношения

Кроме того, верификация может выявить следующее. При проверке *столбцов* матрицы трассировки могут обнаружиться столбцы, не связанные ни с одной строкой. На рис. 31.8 прецедент 3 (UC3) не связан ни с одной функцией продукта (FEA).

Relationships: - direct only	UC1: On/Off	UC2: Infrared Emergency	UC3: Occupant Detection
FEA1: Easy to program control stations			
FEA2: Easy to install			
FEA3: Interface to home security system		⌚	
FEA4: Built-in security features - lights, alarms, bells		⌚	
FEA5: Vacation settings	⌚		
FEA6: 100% reliability	⌚		
FEA7: Instant lighting on/off	⌚		

Рис. 31.8. Прецедент, для которого пропущено отношение

Подобные ситуации свидетельствуют, что для созданного прецедента (или требования) не существует связанной с ним функции продукта. Иными словами, по-видимому, требование является лишним. Как и ранее, следует проверить отношения трассировки.

- Может оказаться, что связь пропущена случайно при задании трассировки. В таких случаях нужно просто добавить новую связь, пересчитать матрицу трассировки, и проблема будет решена.
- Но может быть и так, что при задании функций продукта просто не были учтены потребности одного из необходимых требований к программному обеспечению. Такая ситуация может возникнуть, если есть определенные ограничения проектирования, которые необходимы при реализации, но меняют функции продукта. В таком случае необходимо исправить проект, чтобы учесть осуществимость и необходимость требований. Можно удалить соответствующее требование(я) или поместить его в список “будущих”. С другой стороны, при пересмотре проекта может оказаться, что пропущенную функцию следует поместить в категорию “будущих” или добавить уже сейчас. В нашем примере мы в ходе проверки проекта пришли к выводу, что UC3 является лишней спецификацией, и ее не должно быть в проекте вовсе.

При верификационном просмотре такого типа основное внимание уделяется тому, чтобы в проект не закрались беспричинные элементы. Опыт свидетельствует, что они значительно увеличивают объем проекта и редко способствуют повышению качества результата.

Размышления о верификации и трассировке

Проблемой при поиске пропущенных/лишних отношений является то, что эта работа очень механическая. Другими словами, возникает соблазн сказать: "Мы посмотрели на все строки и столбцы, и все выглядит нормально. Так что верификация закончена. Давайте двигаться дальше!".

Недостатком подобной аргументации является то, что мы не рассмотрели, *полно ли корректно ли* отражены все связи, которые следует (или не следует) установить. Мы обнаружили, что более глубокое изучение исходных трассировочных связей всегда приводит к неким их исправлениям. Могут добавляться новые связи, а старые – изменяться или удаляться.

Мы призываем вас рассматривать исходные связи как некую точку отсчета при проведении верификации. После задания исходных связей и завершения проверки исходных строк/столбцов нужно провести формальные (или неформальные) проверки. Только после проведения по крайней мере одной полномасштабной проверки и внесения изменений можно считать, что верификация на данной фазе проведена.

Связи, как и сами требования, будут эволюционировать со временем.

Изменения, конечно же, неизбежны. Поэтому необходимо трактовать все связи как "живые", подверженные пересмотрам по мере продвижения проекта. Следует проводить проверки с целью верификации всякий раз, когда становится очевидно, что будущая, текущая или прошлая фаза внесла существенные изменения в трассировочные связи. Часто нужны ретроспективные пересмотры, когда новое понимание проекта приводит к пересмотру старых связей и отношений. Главным является то, что верификация требует *вдумчивого отношения* и нельзя полагаться на чисто механические действия.

Далее...

Верификация является мощным методом, который помогает команде проектировать и реализовать правильную систему. К сожалению, до сих пор во многих проектах методы

верификации недостаточно используются для подтверждения того, что проект движется в правильном направлении. Такие проекты неизбежно "сбиваются с пути", а члены команды даже не подозревают об этом. Конечно, на каком-то этапе несоответствие случайно обнаруживается, но это происходит слишком поздно и разочарование неизбежно. Проекты, которые подвергаются частой верификации, гораздо реже преподносят подобные сюрпризы.

Однако никакая верификация не может гарантировать, что конечный результат будет таким, как предполагалось. Необходим другой принцип, который поможет убедиться, что созданная система является правильной. Этот принцип – *проверка правильности (validation)* – обсуждается в следующей главе.



Глава 32

Проверка правильности системы

Основные положения

- Проверка правильности (*validation*) представляет собой процесс подтверждения того, что разработанная система соответствует предъявляемым к ней требованиям.
- Приемочное тестирование проверяет правильность работы системы в среде клиента и в сценариях использования.
- Чтобы добиться высокого качества, нужно проводить тестирование не только выполнения, но и соответствия требованиям и реальному использованию клиентом.

Стандарт IEEE (1994) определяет проверку правильности (*validation*) следующим образом.

Процесс оценивания системы или компонента во время или по окончании процесса разработки с целью определить, удовлетворяет ли она указанным требованиям (IEEE 1012-1986, § 2, 1994).

Другими словами, проверка правильности призвана подтвердить, что реализованная система соответствует заданным требованиям.

Но данное определение недостаточно. Хотя тестирование на соответствие требованиям является, безусловно, важным шагом, все же существует вероятность, что предоставленная система окажется не такой, как хотел клиент. Иногда много времени и сил уходит на то, чтобы собрать и понять потребности клиентов, затем следует реализация системы, причем показывается (с помощью тестов по проверке правильности), что она корректно выполняет все собранные требования, после чего конечный продукт доставляется клиенту, а тот упирается и заявляет, что продукт не такой, как он хотел.

Что же не так? Все очень просто. Не удалось превратить туманное "облако" проблемы пользователя в организованную структуру, представленную требованиями. Но это слабое утешение для команды проекта, затратившей огромные усилия на создание программного продукта. Проводя приемочные тесты на каждой итерации, можно минимизировать этот синдром.

Проверка правильности

Приемо-сдаточные испытания

Приемо-сдаточные испытания привлекают заказчика к окончательной проверке системы.

Приемо-сдаточные испытания привлекают заказчика к процессу окончательной проверки правильности системы, чтобы убедиться, что “продукт работает именно так, как нужно клиенту”. Для внешних заказчиков приемо-сдаточные испытания могут разрабатываться и выполняться в соответствии с условиями контракта. В среде IS/IT или ISV те же задачи, как правило, выполняются клиентом в процессе альфа- или бета-тестирования.

Приемо-сдаточные испытания обычно основываются на определенном количестве “сценариев”, которые пользователь задает и выполняет в среде использования. Клиент имеет право конструировать оригинальные способы проверки системы, чтобы убедиться, что система работает так, как ему необходимо. Если мы все сделали правильно, приемо-сдаточные испытания будут основываться на ключевых precedентах, которые нами уже были определены и реализованы. Но при приемке эти precedенты будут применяться в различных комбинациях при разных типах загрузки системы и при наличии других факторов среды – параллельная работа с другими приложениями, зависимость от ОС и т.д. – которые, скорее всего, присутствуют в среде пользователя.

В итеративном процессе разработки следует проводить приемочное тестирование при достижении различных важных вех в построении системы, чтобы конечные приемо-сдаточные испытания не преподнесли сюрприз команде разработчиков. В процессе, следующем модели водопада, это не применяется, и значительные неожиданности являются обычным делом. В любой модели никогда не поздно обнаружить по крайней мере несколько “неоткрытых руин”, которым необходимо уделить внимание. В главе 34 обсуждается, как управлять теми изменениями, которые могут появиться в результате.

Тестирование с целью проверки правильности

Основной деятельностью при проверке правильности является тестирование. Но как должен выглядеть хороший план тестирования? Один из вариантов предлагается стандартом IEEE 829-1983, *IEEE Standard for Software Test Documentation* (IEEE, 1994). Данный стандарт содержит восемь образцов документов, которыми следует руководствоваться при выборе методологии тестирования, проведении тестов, сообщении о результатах и устранении аномалий. Другие методологии (Rational, 1999) используют несколько иные подходы, но все они сходятся в главном.

- Процесс разработки должен включать в себя *планирование действий по тестированию*. (В итеративной модели планирование тестов осуществляется преимущественно в фазе исследования.)
- Процесс разработки должен предусматривать выделение времени и ресурсов для *проектирования тестов*. Полезно иметь общий шаблон, разработанный таким образом, чтобы проект каждого отдельного теста больше внимания уделял его индивидуальным особенностям.

- Необходимо предусмотреть выделение времени и ресурсов на выполнение тестов, как на уровне отдельных тестов (при необходимости), так и на общесистемном уровне. Тестовые документы образуют часть документации реализации. Дерево документации реализации вместе с документацией тестирования представлено на рис. 32.1.

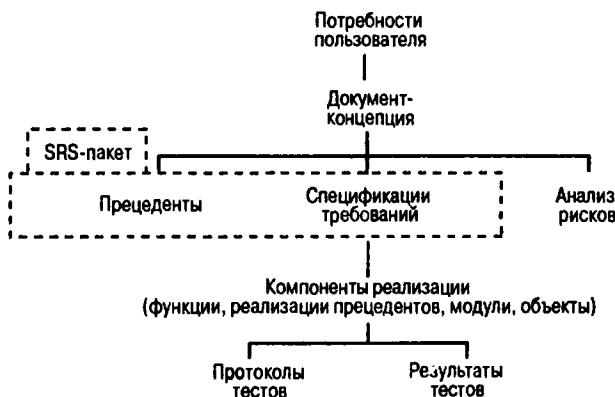


Рис. 32.1. Документация реализации

Мы рекомендуем вести журнал, в котором отражается соответствие действий по проверке правильности/тестированию спецификациям реализации. Эта информация обеспечивается трассировкой.

Трассировка при проверке правильности

При проверке правильности трассировка должна ответить на два важных вопроса.

1. Достаточно ли тестов, чтобы проверить все, что нуждается в тестировании?
2. Нет ли лишних или ненужных тестов?

В процессе проверки правильности выясняется, действительно ли продукт работает так, как предполагалось. На этом этапе больше не проверяются отношения различных спецификаций и элементов проектирования; вместо этого рассматриваются отношения между тестами (и их результатами) и тестируемой системой. Как и при верификации, цель состоит в том, чтобы удостовериться, что все элементы, которые в этом нуждаются, тестируются на соответствие требованиям.

Основанное на требованиях тестирование

Как определить, что нуждается в тестировании? Один достаточно распространенный подход заключается в тестировании продукта посредством проверки правильности реализации содержащихся в нем функций. Зачастую к тестированию подходят следующим образом: “Вот некая выполняемая функция, скажем менеджер базы данных, протестируем ее с помощью интерфейсов менеджера базы данных”. Хотя такой подход и правомерен, все же он выполняет только половину работы.

Высокого качества можно добиться, только протестировав систему на соответствие ее требованиям.

Чтобы добиться высокого качества, необходимо протестировать систему на соответствие *требованиям*. Конечно, может оказаться полезным провести *тестирование модулей* для различных элементов проекта, таких как менеджер базы данных. Но тестирование отдельных модулей не может гарантировать, что *система в целом* работает так, как нужно. Многие сложные разработки успешно проходили все тесты модулей, но не выдерживали испытаний в качестве системы. Почему? Потому, что модули взаимодействуют в более сложных вариантах поведения, а результирующая система не была адекватно протестирована на соответствие системным требованиям.

Посмотрим, как можно использовать методы, разработанные нами применительно к верификации, для проверки правильности системы. Вернемся к рассмотрению нашего рабочего примера.

Рабочий пример. Тестирование прецедентов

Написание тестовых примеров, как и сбор требований, является одновременно и наукой, и искусством. Нам не хотелось бы исследовать данный предмет чересчур глубоко, однако будет полезно хотя бы в общих чертах рассмотреть, как составляются тестовые примеры на основании функциональных возможностей, представленных прецедентами и требованиями, с помощью которых мы описали систему. Для этого вернемся к нашему рабочему примеру и рассмотрим разработанный в части 5 прецедент “Управление освещением”.

Описание тестового примера 1

В табл. 32.1 представлен образец тестового примера для прецедента “Управление освещением”. Тестовый пример 1 предназначен для тестирования экземпляров прецедента “Управление освещением”. Он используется только для тестирования тех кнопок пульта (Управление включением), которые отвечают за набор осветительных приборов, допускающих изменение яркости.

Таблица 32.1. Тестовый пример 1 (упрощенное представление)

ID ситуации	Описание события	Ввод 1	Ввод 2	Ожидаемый результат
<i>Основной поток</i>				
2001	Жилец нажимает кнопку пульта	Любая допустимая кнопка	Перед нажатием кнопки свет был включен (тестолог должен запомнить уровень).	Свет выключается
2002			Перед нажатием кнопки свет был выключен	Свет включается с уровнем яркости OnLevel

Окончание табл. 32.1

ID ситуации	Описание события	Ввод 1	Ввод 2	Ожидаемый результат
2003	Жилец отпускает кнопку в течение 1 секунды	Свет включен		Остается выключенным
2005	Жилец отпускает кнопку в течение 1 секунды. (Этим заканчивается первый путь председента.)	Свет выключен		Остается включенным с прежним уровнем яркости (OnLevel)
2006	Жилец нажимает кнопку снова и отпускает в течение 1 секунды.	Та же кнопка, что и в 2003	Перед этим свет выключен	Свет включается с тем же уровнем яркости, что и в 2002
	Жилец нажимает кнопку вновь и отпускает в течение 1 секунды		Перед этим свет выключен	Свет выключается
<i>Альтернативный поток</i>				
2007	Кнопка удерживается в нажатом положении дольше 1 секунды	Допустимая кнопка	Перед этим свет выключен	Свет включается. Яркость возрастает до максимального уровня (по 10% в секунду), а затем убывает (по 10% в секунду) до минимального уровня, после чего снова возрастает. Циклы продолжаются, пока нажата кнопка
2008	Жилец отпускает кнопку			Яркость остается на последнем достигнутом уровне

Замечание. Тест выполняется много раз при разном времени нажатия кнопки, чтобы проверить, что система правильно запоминает уровень яркости OnLevel.

Тестовый пример 1 предназначен для тестирования взаимодействий с системой, которые достаточно точно имитируют реальный поток событий, описанный в прецеденте "Управление освещением". Таким образом, прецедент служит шаблоном того, как тестировать систему. Это одно из основных преимуществ метода прецедентов.

Полная версия этого тестового примера приводится в приложении А вместе с другими артефактами системы HOLIS. Тестовый пример 2, описанный в приложении А, предназначен для тестирования целого набора дискретных требований, а не отдельного прецедента. Мы рассмотрим оба тестовых примера в следующем разделе, посвященном трассировке.

Трассировка тестовых примеров

Трассировка помогает убедиться, что тестовые примеры покрывают все требуемые функциональные возможности.

Методы трассировки позволяют убедиться, что тестовые примеры полностью покрывают все необходимые функциональные возможности системы. Нужно просто создать последовательности планов тестирования, которые можно связать с исходными системными требованиями и прецедентами.

Предположим, у нас имеется матрица трассировки, сопоставляющая тесты и прецеденты (рис. 32.2). Так же, как и при проведении верификации, можно исследовать данную матрицу, чтобы убедиться, что тестовые примеры надлежащим образом покрывают все системные спецификации. Аналогично можно связать тестовые примеры с прецедентами, как это показано на рис. 32.3.

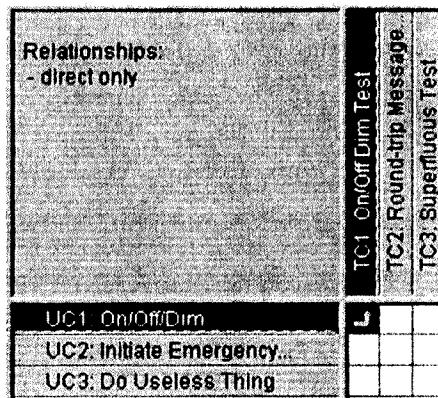


Рис. 32.2. Тесты и прецеденты

Тестирование дискретных требований

Мы использовали отношения трассировки, чтобы связать прецеденты с тестовыми примерами. Трассировку можно также применять для задания отношений между дискретными (обособленными) требованиями и последующего их связывания с тестовыми примерами. На рис. 32.4 показан фрагмент матрицы трассировки тестовых примеров. В

нем присутствует тестовый пример 2 ("TC2. Двустороннее сообщение..."), который связан с требованиями к программному обеспечению SRS-пакета HOLIS. Заметим, что трактовка тестовых примеров не отличается от трактовки других элементов, трассируемых при верификации и проверке правильности.

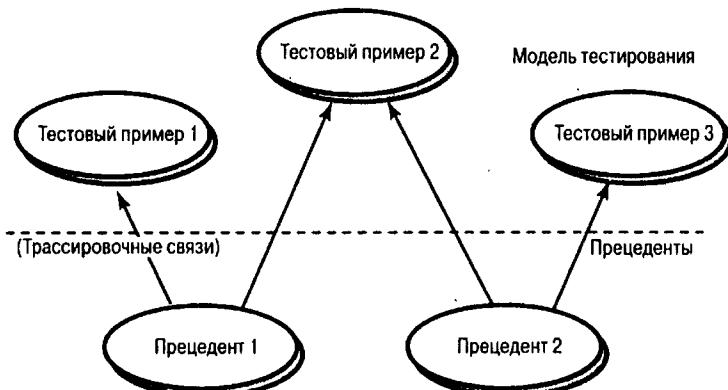


Рис. 32.3. Прецеденты и тестовые примеры

Relationships: - direct only		TC1: On/Off Dim Test	TC2: Round-trip Message...	TC3: Superfluous Test	Relationships: - direct only	
UC1: On/Off Dim	UC2: Initiate Emergency...				SR1: System Clock. The system shall...	SR2: OnLevel illumination parameter...
UC1: On/Off Dim	UC2: Initiate Emergency...	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/> SR3: HOLIS shall support up to 255...	<input checked="" type="checkbox"/> SR4: Message protocol from Control...
UC3: Do Useless Thing					<input checked="" type="checkbox"/> SR5: The CCU must have no known...	<input checked="" type="checkbox"/>

Рис. 32.4. Фрагмент матрицы трассировки тестовых примеров

Итак, мы внесли тестовые примеры в матрицы трассировки. Теперь нужно исследовать заданные связи, как это делалось при верификационных просмотрах.

Пропущенные отношения проверки правильности

Для выявления пропущенных отношений нужно искать *строки* матрицы трассировки, из которых видно, что определенная функция (или требование) не связана ни с каким тестом. На рис. 32.5, например, прецедент 2 (UC2) не связан ни с одним тестовым примером (TC). (Связи для прецедента UC3 тоже отсутствуют, но в процессе верификации мы уже пришли к решению, что данный прецедент вообще не нужен.)

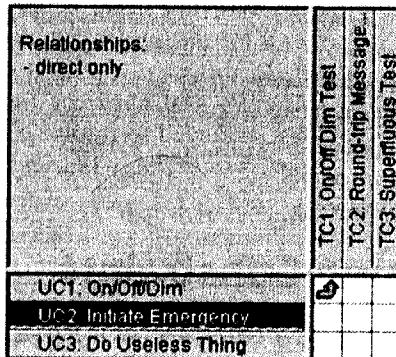


Рис. 32.5. Пропущенный тестовый пример

Выявив такую “дыру” в отношениях, необходимо проверить исходный набор требований к продукту и соответствующие прецеденты.

- Если окажется, что связь случайно пропущена при задании трассировки, нужно просто добавить новую связь и пересчитать матрицу трассировки. Подобные пропуски часто возникают на ранних стадиях задания трассировки проверки правильности.
- Если обнаружится, что при разработке тестовых примеров просто не удалось протестировать одну из необходимых функций продукта, может понадобиться пересмотреть проект, чтобы добавить подходящие тесты, соответствующие данной функции. В отличие от аналогичного случая при верификации, мы *не рекомендуем* отмечать пропущенные тестовые примеры как “будущие” действия. Если есть непротестированная функция, можно не сомневаться, что заказчик будет ее тестировать (часто с неприятными для вас последствиями). Кроме того, регулируемые разработки, такие как контролируемая FDA разработка программного обеспечения для медицинских целей, не допускают откладывания проведения необходимых тестов.

При проверке правильности трассировка помогает удостовериться, что никакие связи не пропущены и все тесты продукта надлежащим образом связаны с более высоконеобходимыми требованиями к продукту.

“Лишние” отношения проверки правильности

Как и при верификации, в ходе проверки правильности можно при просмотре *столбцов* трассировочной матрицы обнаружить столбцы, не связанные ни с одним элементом строки. Например, на рис. 32.5 тестовый пример 3 (TC3) не связан ни с одним прецедентом. Из рис. 32.4 также видно, что он не связан ни с каким программным требованием. Это означает, что был создан тестовый пример, для которого не существует связанной с ним функции продукта. Иначе говоря, тест выглядит лишним. Как и ранее, следует рассмотреть отношения трассировки.

- Возможно, связь была случайно пропущена при задании трассировки. Если так, нужно просто добавить новую связь и пересчитать матрицу трассировки. Такие пропуски часто возникают на ранних стадиях задания трассировки проверки правильности.

- Может оказаться, что при разработке функций продукта просто не были учтены потребности одного из необходимых программных тестов. Это может произойти в том случае, если определенные нефункциональные требования к реализации фактически меняют функции продукта. Тогда может понадобиться пересмотреть проект, чтобы учесть достижимость и необходимость требований. Как и при верификации, команде нужно будет принять решение, нужен ли данный тест вообще и, если нужен, какие трассировочные связи необходимы.

Тестирование ограничений проектирования

Итак, теперь понятно, как обращаться с тестами для прецедентов и требований. Но возникает вопрос: «Как тестировать ограничения проектирования?».

В части 5 мы отмечали, что хотя ограничения проектирования имеют свои особенности, наиболее простым способом их трактовки является просто рассматривать их как требования. Другими словами, их связи трассируются так же и их верификация тоже производится аналогичным образом. Было бы логично учитывать ограничения проектирования и при проверке правильности и тестировать их наравне со всем остальным. Многие ограничения проектирования тестируются с помощью элементарного просмотра. Например, ограничение, требующее, чтобы программа была написана на Visual Basic (VB), можно протестировать, просто посмотрев на исходный код.

Так как проверка многих ограничений проектирования будет элементарной, следует иметь для них сокращенную процедуру тестирования. Нет необходимости использовать сложные формы, перечисляющие калиброванное оборудование, процедуры по установке, настройки среды и т.д. Вместо этого можно использовать упрощенную форму, свидетельствующую, что была проведена проверка кода или другого артефакта и обнаружено, что он удовлетворяет ограничению проектирования. Некоторые способы тестирования ограничений проектирования перечислены в табл. 32.2.

Таблица 32.2. Способы проверки ограничений проектирования

Ограничение проектирования	Способ проверки
«Писать программу на VB 5.0»	Проверить исходный код
«Приложение должно основываться на архитектурных шаблонах Fuji Project»	Выявить шаблоны моделей проектирования Fuji; сравнить с архитектурой текущего проекта
«Использовать библиотеку классов Developer's Library 99-724 корпорации XYZ Corporation»	Проверить записи о заказах и получении; просмотреть полученную документацию продукта и номера редакций; проверить, что библиотеки правильно загружены и правильно используются

Далее...



Верификация представляет собой аналитический процесс, выполняемый на всех стадиях разработки проекта с целью убедиться, что вы все делаете правильно. *Проверка правильности (validation)* позволяет убедиться, что система работает так, как предполагалось, т.е. удовлетворяет как документированным требованиям клиента, так и реальным сценариям использования. Вместе верификация и проверка правильности помогают команде убедиться, что она действительно “правильно создает “правильную” систему”.

Но кое-что осталось невыясненным. Возникает вопрос: “Как определить, каким должен быть объем работ по верификации и проверке правильности (V&V)?”. Ответ на этот вопрос вы найдете в следующей главе.

Глава 33

Применение метода анализа дивидендов для определения объема V&V-действий

Основные положения

- Глубина верификации элемента системы может варьироваться от промсмотра (который обеспечивает минимальную глубину) до сквозного контроля, независимой ревизии, тестирования черного ящика (имитации) и прозрачного тестирования (тестирования каждой строки кода).
- Зона действия (покрытие) отражает, для какой части элементов системы проводится верификация и проверка правильности.
- Имеет смысл проводить выборочные испытания, если известно, зачем вы их проводите и в чем состоят риски.
- При анализе рисков необходимо принять решение о том, какие ошибки нужно предотвратить и какой объем действий по верификации и проверке правильности необходим, чтобы гарантировать, что они не возникнут.

Как и все остальное, иницирование и проведение V&V-действий требуют определенных затрат. Естественно возникает вопрос: "Как провести анализ затрат/прибыли, чтобы увидеть, действительно ли полученные результаты того стоят?". Данная глава поможет составить план действий по проведению испытаний с помощью неких квазикономических понятий для ответа на вопрос о затратах/прибыли. Мы будем планировать наши V&V-действия, исходя из ответов на следующие два вопроса.

1. Какими будут социальные или экономические последствия сбоя системы?
2. Какой необходим объем V&V-действий, чтобы гарантировать, что эти последствия не возникнут?

Безусловно, не очень приятно по окончании проекта обнаружить, что было проведено слишком много всяких проверок, результаты которых имели для него минимальное значение. Еще более неприятно, если окажется, что проведенных проверок недостаточно. Это может привести даже к возврату продукта. (Представьте себе, что значит отозвать первые 10000 экземпляров HOLIS, которые уже установлены в домах.) Следовательно, нужно постараться найти аналитический подход, который поможет выбрать со-

ответствующий объем V&V-действий *до того*, как мы к ним приступим. Мы начнем с рассмотрения вопросов “глубины” и “покрытия”.

Глубина и покрытие

Глубина V&V

Глубина определяет уровень детализации при проведении проверки элемента системы.

Глубина задает уровень детализации, выбранный для верификации или оценки элемента системы. В среднем, чем больше глубина, тем больше понадобится времени и средств для выполнения этих действий. Поэтому полезно убедиться, что для каждого элемента выбрана глубина проверки, которая соответствует важности элемента.

Не все элементы нуждаются в одинаковой глубине проверки. Например, для неосновных элементов может быть достаточно просмотра или простого системного теста, а для критических может понадобиться масштабное прозрачное тестирование.

В главе 26, “Неоднозначность и уровень конкретизации,” мы говорили, что выбранный уровень конкретизации будет определять как глубину, так и количество задаваемых требований. Выбор правильного уровня конкретизации (глубины) будет также определять и глубину проведения V&V. Рассмотрим некоторые методы проверки и предлагаемую ими глубину.

- *Просмотр (Examination).* Просматривается код или осуществляются некие измерения. Суть просмотра в том, что тестируемые элементы подвергаются заранее определенному минимально глубокому рассмотрению. Это считается минимальной глубиной проверки элемента.
- *Сквозной контроль (Walkthrough).* Осуществляющая проверку группа рассматривает все этапы работы элемента. В некотором смысле этот процесс является структурированным просмотром, выполняемым более широкой группой, которая ищет недостатки, упущения и т.п. в коде. Такой тип проверки обеспечивает большую глубину, чем простой просмотр.
- *Независимые рецензии (Independent reviews).* Этот метод аналогичен двум предыдущим. Группа независимых специалистов просматривает элемент и выявляет недоработки. В результате такая проверка может выявить дополнительные вопросы, которые не были замечены группой проекта.
- *Тестирование черного ящика (black-box test).* В данном случае элемент рассматривается как модуль, внутренняя структура которого неизвестна. Таким образом, можно поставлять ему вводы и наблюдать его выводы, чтобы убедиться, что элемент работает в соответствии с требуемыми стандартами. Такие тесты обычно выполняются с помощью специального контрольного кода или системных эмуляторов, а также других средств имитации и записи функционирования системы.
- *Прозрачное тестирование (white-box test).* При таком тестировании позволено “открыть ящик” и исследовать внутреннее устройство элемента. Большинство модулей кода имеет слишком много комбинаций возможных вариантов развития

процесса вычислений, чтобы их можно было протестировать за разумное время. Поэтому, чтобы не тратить на прозрачное тестирование слишком много времени, необходимо применить к нему некую разумную концепцию тестирования. Общепринятым компромиссом является исследование каждой строки кода, но не всех возможных комбинаций магистральных путей. Такой тип проверки обычно требует значительного объема ресурсов и привлечения дополнительного персонала.

V&V-покрытие

Покрытие определяет, какая часть элементов системы подвергается верификации и проверке правильности.

Покрытие (зона действия, *coverage*) означает степень покрытия элементов системы V&V-действиями. Как обсуждалось в предыдущих двух главах, в рамках V&V-деятельности можно применить к различным элементам методы трассировки. Например, можно трассировать требования к тестовым примерам, функции к прецедентам, прецеденты к реализации и т.д. *Качество* задаваемых трассировок и уровень конкретизации требований являются основными факторами при определении V&V-покрытия.

Иными словами, нужно решить, какие требования (как текстовые, так и в виде прецедентов), элементы реализации, тестовые примеры и т.д. необходимо рассмотреть. Кроме того, может понадобиться исследовать, верифицировать и проверить правильность программного обеспечения, поставляемого другими, такого как закупаемые компоненты и операционная система, выполняющая приложение.

Что подвергать верификации и проверке правильности

Итак, при составлении плана V&V необходимо решить, какие элементы следует подвергать верификации и проверке правильности, чтобы создать высококачественный продукт и при этом минимизировать общие затраты на его разработку. Есть несколько способов решения этого вопроса.

Вариант 1. Верифицировать и проверять правильность всех элементов

В небольших проектах, где требуется достаточно высокий уровень качества предоставляемого продукта, можно выполнять V&V-процессы практически для всех элементов приложения. Преимущество данного подхода в его понятности и в одинаковой трактовке элементов разработки. Кроме того, не нужно проводить анализ перед началом разработки и строить предположения относительно стоимостных элементов V&V.

При выборе такой политики команда вскоре приходит к соглашению, что нет необходимости проводить испытания некоего элемента, поскольку он слишком тривиален, чтобы о нем беспокоиться. В принципе, такой подход допустим, но, как правило, такие решения чаще принимаются, когда исчерпывается бюджет или подходит к концу отведенный срок.

Допустимо выборочное проведение V&V, если известно, в чем состоят риски.

Выборочное проведение V&V допустимо в тех случаях, когда известно, почему они проводятся и в чем состоят риски. Этот подход заслуживает внимания при условии, что элементы пропускаются по весомым причинам, не связанным с недостатком времени или денег. Если элементы не подвергаются V&V, важно документировать причины этого.

Существует опасность, что могут быть не выполнены V&V-действия по отношению к элементу проекта, который вдруг окажется более важным, чем предполагалось изначально. Это может иметь перечисленные ниже последствия.

- Элемент не будет соответствовать спецификации клиента (как следствие – необходимость неоплачиваемой переработки этой части системы).
- Элементы будут работать не так, как полагается согласно спецификации (следствие – дорогостоящий отзыв изделий).
- Наихудший случай: в результате получится небезопасный продукт, который может принести вред пользователям.

Таким образом, даже в случае неформального выборочного сокращения V&V-действий существует вероятность непредусмотренного развития событий.

Но если в крупных проектах не практикуется тотальное проведение V&V, а необходимо гарантировать высокое качество, то как решить, какие элементы важны для V&V-процесса, а какие – нет? Для ответа нам нужно обратиться к варианту 2.

Вариант 2. Анализ рисков для определения необходимости V&V

Одним из систематических подходов к выявлению важных элементов проекта является анализ рисков (hazard analysis) и связанные с ним действия по оценке рисков. Мы не будем углубляться в эти дисциплины в данной книге, но представим краткий обзор основных понятий.

Регулирующие органы, такие как FDA (Управление по санитарному надзору за продуктами и медикаментами США), рассматривают анализ рисков как ключевой фактор улучшения качества продукта. Современные инструкции содержат целые разделы, посвященные оценке и анализу рисков. В частности, Вуд и Эрмес (Wood, Ermes, 1993, a) предложили полезное определение анализа рисков для медицинского продукта.

Анализ рисков представляет собой подробное исследование прибора с точки зрения пользователя и пациента. Его цель состоит в выявлении потенциальных недоработок проектирования (возможностей отказа, которые могут принести вред) и предоставлении производителю возможности исправить их до того, как прибор будет запущен в производство.

Разработчик должен рассмотреть различные типы ошибок, которые могут возникать в создаваемом продукте. Все потенциальные риски исследуются и фиксируются в специальном документе, что позволяет разработчику предложить стратегии проектирования, которые помогут их избежать.

На последующих стадиях жизненного цикла разработки продукта документ анализа рисков будет служить для фиксации как потенциальных рисков, так и методов, применяемых для их уменьшения. Позднее при проверке правильности системы данный документ используется для того, чтобы убедиться, что все предполагаемые риски полностью учтены и устранены, а тестирование будет преимущественно направлено на эти области, для достижения более высокой степени гарантий.

Конечно же, анализ рисков можно применять не только к системам, в которых опасности подвергается жизнь людей (медицинским, транспортным или промышленным). Следует применять его повсюду, где, по вашему мнению, наиболее велик риск сбоев в системе. Например, в системе интерактивной торговли анализ рисков может быть направлен на то, чтобы "гарантировать, что предоставленная заявка является правильной", или "проверить число введенных пользователем акций". Телекоммуникационная компания может уделять основное внимание риску "катастрофического сбоя программы управления коммутацией".

В любом случае анализ рисков используется для того, чтобы решить, какие риски необходимо предотвратить в системе и каков объем необходимых для этого V&V-действий. Для элементов проектирования, имеющих, согласно документу анализа рисков, большое значение для общей безопасности и успеха разработки, следует обеспечить проведение полномасштабных V&V-действий. Для элементов, имеющих меньший или незначительный риск, можно уменьшить объем V&V-действий или вовсе пропустить их, хотя общее тестирование системы все равно необходимо.

Анализ рисков служит руководством при выборе элементов проекта для проведения V&V.

Анализ рисков можно использовать при выборе элементов проекта, нуждающихся в верификации. Независимо от верификации, анализ рисков может также служить руководством при выборе тестов и тестового покрытия при проверке правильности. Нет правила, требующего тесной связи верификации и проверки правильности, поэтому команда может использовать результаты анализа рисков для независимой разработки планов верификации и проверки правильности.

Анализ рисков и анализ дивидендов (ROI)

Можно рассматривать анализ и оценку рисков, интерпретируя их как стандартный анализ затрат/прибыли, и использовать результаты анализа рисков в качестве исходной информации для стандартного анализа дивидендов (return on investment, ROI).

Сначала производятся оценки затрат (времени, ресурсов и денег) для действий по проверке некоего элемента или сегмента проекта. Эти затраты вводятся в стандартные экономические модели ROI, чтобы получить представление о затратах этапа. Затем выполняется оценка возможного воздействия предварительно выявленных в результате анализа рисков негативных последствий, которые могут возникнуть при неправильной работе элемента, не подвергавшегося V&V. После сравнения двух полученных результатов можно принять обоснованное решение о том, стоит ли проводить V&V-действия и какой должна быть их глубина.

Во многих случаях анализ показывает, что решение "да/нет" слишком упрощенное. Гораздо более типичная ситуация, когда одна часть V&V-действий для сегмента эффек-

тивна, а другая – нет. В таких случаях, возможно, стоит рассмотреть модифицированную стратегию V&V, чтобы оптимизировать дивиденды от произведенных затрат.

Всегда необходимо выполнять анализ рисков для аспектов, имеющих отношение к безопасности людей.

Следует понимать, что вычисление дивидендов отличается от анализа безопасности и эффективности. Например, анализ рисков может показать, что с определенной частью разработки и реализации программы связана проблема, касающаяся безопасности людей. В таких случаях просто недопустимо игнорировать факторы безопасности из-за размеров дивидендов. *Следует всегда проводить полный цикл V&Vдействий по отношению к сегментам проекта, затрагивающим проблемы безопасности людей.* Другими словами, ROI-методы – это только вопрос финансовых затрат проекта. Если на карту поставлены вопросы безопасности и эффективности, необходимо использовать анализ и оценку рисков. Если это уместно, можно применять комбинацию различных методов.

Далее...



Итак, посмотрим, чего мы достигли.

- Мы задали способ, который позволяет рассматривать и использовать артефакты требований в качестве основы при проектировании и реализации системы.
- Мы применили методы верификации, чтобы убедиться, что каждый шаг проекта трассируется к более ранним шагам.
- Мы описали два процесса проверки правильности (тестирование и приемо-сдаточные испытания), совместное применение которых помогает убедиться, что в результате реализации получилась работоспособная система.
- Мы исследовали методы оценки и анализа рисков, которые помогают принять решение, как наиболее эффективно использовать ресурсы проекта.

Единственной нерассмотренной темой осталась проблема изменений. В следующей главе мы научимся, как справляться с изменениями и правильно их обрабатывать в ходе разработки проекта.

Глава 34

Управление изменениями

Основные положения

- Процесс управления требованиями может быть полезным только в том случае, если он позволяет распознавать и решать проблему изменений.
- Внутренними факторами, приводящими к возникновению изменений, являются неспособность задать нужные вопросы нужным людям в нужное время и отсутствие практического процесса, призванного справиться с изменениями требований.
- Чтобы повысить шансы на успех, необходимо предотвратить "просачивание" требований или, по крайней мере, существенно уменьшить его.

Почему меняются требования

Если бы можно было раз и навсегда определить множество требований к системе, жизнь была бы намного проще и данная глава была бы не нужна. Мы бы просто создали совершенный документ-концепцию, совершенную спецификацию программных требований и модель precedентов, заморозили бы их на время разработки, а затем объявили, что за все, что произошло после этого, отвечает команда сопровождения. Увы, так не бывает; так никогда не было и не будет даже при самом систематическом подходе к управлению требованиями.

Есть несколько причин неизбежности изменений требований. Среди них внутренние факторы, которые мы можем контролировать, и внешние, которые не подвластны контролю со стороны разработчиков и пользователей.

Внешние факторы

Внешними факторами являются те источники изменений, которые команда проекта не может контролировать. вне зависимости от того, как мы справимся с ними, необходимо подготовиться технически, эмоционально и методологически, чтобы воспринять эти изменения как часть "нормального развития событий при разработке программного обеспечения". Изменения возникают по следующим причинам.

- Произошли изменения проблемы, которую мы пытались решить с помощью новой системы. Возможно, возникли изменения в экономике, в правительственные инструкциях, на рынке или же изменились предпочтения потребителей. Из-за быстрых темпов развития технологий сейчас становится все более вероятным, что такие изменения произойдут до того, как мы закончим решение исходной проблемы, описанной пользователем.

- Пользователи *изменили свое мнение* о том, чего они хотят от системы, или свои предпочтения. Это, в свою очередь, может произойти вследствие ряда причин: не только из-за непостоянства пользователей (которое особенно ярко проявляется при спецификации деталей интерфейса человек/система), но и из-за того, что их предпочтения зависят от ситуации на рынке, экономики, различных инструкций и т.д. Более того, часто меняется сама личность пользователя; например, тот, кто описывал требования к системе, покидает команду заказчика, а вместо него может оказаться некто, кто имеет существенно иное мнение (и предпочтения).
- Изменилась *внешняя среда*, что привело к появлению новых ограничений и/или новых возможностей. Одним из наиболее очевидных примеров изменения среды является постоянно происходящее совершенствование систем аппаратного и программного обеспечения: если компьютеры будут работать в два раза быстрее, станут на 50% дешевле и компактнее и будут способны выполнять более сложные приложения, они, скорее всего, вызовут изменения в требованиях к системе. В начале 90-х вряд ли кто-нибудь предвидел развитие Internet и World Wide Web. Современные требования к широкому спектру информационных систем – от текстовых процессоров до информационных и банковских систем – естественно, сильно отличаются от требований доинтернетовой поры.
- Восла в строй *новая система*. Одним из самых неожиданных внешних факторов возникновения изменений (и главной причиной возникновения синдрома “да, но...”) является то, что *само появление новой системы приводит к тому, что меняются требования к ней*. Благодаря новой системе меняется поведение организации, старые способы выполнения действий больше не применяются; возникает потребность в новых типах информации и неизбежно разрабатываются новые требования к системе. Таким образом, сам факт ввода в строй новой системы выявляет новые требования к ней.

Процесс управления требованиями может быть полезным только в том случае, если он выявляет и решает проблему изменений.

Процесс управления требованиями может быть полезен только в том случае, если он позволяет выявлять и решать проблему изменений. Невозможно предотвратить изменения, но можно научиться ими управлять.

Внутренние факторы

Помимо внешних факторов, существует ряд внутренних, которые также приводят к возникновению изменений.

- При первоначальном выявлении требований нам не удалось задать правильные вопросы нужным людям в нужное время. Если процесс коснулся не *всех* заинтересованных лиц или им не были заданы *правильные вопросы*, это усугубляет проблему изменений, так как нет понимания истинных требований к системе. Иначе говоря, существует гораздо больше, чем нужно “неоткрытых руин”, и в ходе разработки приходится производить значительные изменения, которых можно было бы избежать, если бы на более раннем этапе удалось добиться более полного понимания.

- Нам не удалось создать практический процесс, позволяющий *справиться с изменениями* требований, которые являются нормой при пошаговой разработке. Возможно, мы пытались “заморозить” требования, и “латентные” необходимые изменения накапливались до тех пор, пока не “взорвались” перед лицом разработчиков и пользователей, вызвав переделки и стресс. Или *процесс внесения изменений* отсутствовал вовсе; все могли изменять все, что угодно и когда угодно. В таком случае на некотором этапе практически все оказывается измененным и невозможно понять, что к чему.

Перед тем как заняться этими проблемами, посмотрим, какие еще факторы можно обнаружить.

Наш враг — мы сами

Вайнберг (Weinberg, 1995) заметил, что изменения могут быть скрытыми. После неудачного завершения одного проекта он сравнил известные требования к системе на момент окончания с требованиями, сформулированными вначале. При этом он обнаружил множество источников изменений требований. Некоторые были “официальными”. Они представляли собой запросы пользователей, сделанные по соответствующим каналам общения. Но многие оказались на удивление “неофициальными”. Данное явление Вайнберг назвал “просачиванием” требований. Среди них можно указать следующие.

- Упомянутые дистрибуторами улучшения, о которых программисты случайно услышали на переговорах.
- Прямые запросы клиентов, обращенные к программистам.
- Ошибки, оставшиеся в отгруженной версии продукта, которые теперь нуждаются в сопровождении.
- Аппаратные функции, которые в итоге не вошли в продукт или не работают.
- Изменение масштаба в ответ на действия конкурентов.
- Функции, включенные программистом из “лучших побуждений” (в расчете, что это понравится клиенту).
- “Сюрпризы” программистов.

В одном проекте половина всех функций рабочего продукта системы была направлена на удовлетворение “просочившихся” требований!

Подобных изменений может быть сравнительно немного, но в целом *требования, полученные из неофициальных источников, могут составлять до половины масштаба всего проекта!* Другими словами, половина всех функций рабочего продукта системы будет создаваться для выполнения просочившихся требований, т.е. тех, которые вошли в систему незаметно для членов команды, отвечающих за график, бюджет и критерии качества.

Как менеджер проекта может учесть эти изменения и при этом не выйти из графика и удовлетворить критерии качества? Это сделать невозможно! Чтобы иметь шансы на успех, “просачивание” требований должно быть остановлено или, по крайней мере, сведено к минимуму.

Сюрпризы программиста

Сюрпризы программиста (Programmers' Easter Eggs) являются наиболее опасной формой “просачивания” требований. Сюрприз – это скрытое поведение, встроенное в систему для целей отладки, просто шутки ради или злонамеренно. Как свидетельствует опыт, сюрпризы очень опасны, и программисты должны знать, что их включение совершенно недопустимо и нарушитель этого правила будет подвергнут суворому наказанию. Ниже приводятся два подлинных случая.

1. Для выполнения некой крупной воинской имитационной системы требовалось достаточно много времени, и программисты встроили фоновую игру “Линкор”, чтобы занять себя во время имитации. К несчастью, впоследствии они не убрали ее и не указали на ее существование ни в процессе верификации и проверки правильности, ни в отчете. Когда все обнаружилось, заказчик, потерявший доверие к подрядчику, отказался от всей программы. В результате подрядчик потерял миллионы долларов и серьезно подорвал свои позиции в деловом мире.
2. Молодой программист, участвующий в разработке некоего программного средства, развлекал себя тем, что встраивал уничтожительные сообщения об ошибках в ранние версии заглушек кода обнаружения ошибок. Одно из таких сообщений случайно осталось и было обнаружено заказчиком во время проведения официального сеанса обучения. В результате пришлось переделывать и повторно сдавать программу, что привело к потере драгоценного времени.

Процесс управления изменениями

Поскольку изменения являются неотъемлемой частью процесса, а источники их поступления могут быть как внешними, так и внутренними, необходим *процесс управления изменениями требований*. Такой процесс *дисциплинирует команду*, позволяет ей выявлять изменения, производить анализ их воздействия и систематическим способом включать те из них, которые наиболее необходимы и приемлемы для системы. Согласно рекомендациям Вайнберга (Weinberg), процесс обработки изменений должен включать в себя следующие шаги.

1. Осознать, что изменения неизбежны, и разработать план управления изменениями.
2. Сформировать базовый уровень требований.
3. Установить единый канал контроля изменений.
4. Использовать систему контроля изменений для их фиксации.
5. Обрабатывать изменения по иерархическому принципу.

Рассмотрим более подробно каждый из этих шагов.

Шаг 1. Осознать, что изменения неизбежны, и разработать план управления изменениями

Команда должна признать, что изменения требований к системе неизбежны и даже необходимы. Изменения будут возникать, и команда, зная об этом, должна разработать соответствующий *план управления изменениями*, который разрешит внесение определенных изменений в исходный базовый уровень.

Что касается легитимности изменений, все они (за исключением сюрпризов) могут считаться таковыми, так как исходят от заинтересованных лиц, которые имеют как реальную потребность, так и возможность внести реальный вклад в приложение.

Например, запросы изменений со стороны команды разработчиков являются правомерными, так как она знает о системе больше, чем кто-либо другой. Естественно, разработчики будут вносить множество предложений о том, что должна делать система. Некоторые требования поступают от тех, кто непосредственно занимается реализацией системы; только они в полной мере осознают, что в действительности она может делать. Следует прислушиваться к их мнению; в результате получится более удачная система для наших пользователей.

Шаг 2. Формирование базового уровня требований

Ближе к концу фазы исследования жизненного цикла разработки команда должна скомпоновать все известные требования к системе. Процесс формирования базового уровня может заключаться просто в наложении контроля исправлений на соответствующие артефакты (документ-концепцию, программные требования, модели прецедентов) и публикации базового уровня для команды разработчиков. Собранные в этих документах отдельные требования создают базовый уровень информации о требованиях и предполагаемых прецедентах системы.

Этот простой шаг позволяет команде различать известные ("старые") требования и новые (те, которые были добавлены, удалены или модифицированы). После задания базового уровня гораздо легче выявлять и обрабатывать новые требования. Запрос на новое требование можно сравнить с существующей базой и определить, где оно будет размещаться и не будет ли оно конфликтовать с другими требованиями. К сожалению, этот шаг пользователи зачастую пропускают, стараясь побыстрее отреагировать на изменения в своей среде. Если изменение принимается, можно проследить его эволюцию от концепции к программным требованиям, от программных требований к соответствующим документам и моделям технического проектирования, а затем к коду и тестовым процедурам.

Упорядоченное и ответственное внесение изменений позволяет наладить сотрудничество с сообществом пользователей. В прошлом пользователи зачастую чувствовали сопротивление со стороны разработчиков, когда просили об изменениях. Причина заключалась в том, что команда имела в своем распоряжении лишь хаотический бессистемный процесс внесения изменений или не могла описать природу этого процесса пользователям.

Упорядочение процесса внесения изменений не означает, что можно вносить огромное количество всевозможных изменений.

Однако упорядочение процесса внесения изменений не означает, что можно вносить их сколько угодно. С точки зрения как пользователей, так и разработчиков жизнь будет гораздо проще, если удастся создать набор стабильных корректных требований. Даже при достаточно хорошо организованном процессе управления изменениями существуют ограничения на количество изменений, которые разработчик сможет учесть, особенно на стадиях проектирования и реализации. Как правило, коэффициент изменения требований во время разработки составляет 1–4% в месяц. Но если его величина превышает 2% в месяц, проект подвергается высокому риску "перемешивания" требований.

Шаг 3. Задание единого канала контроля изменений

Изменения могут быть коварными. Хотя очевидно, что появление новой функции может оказать существенное влияние на требования к программному обеспечению, системную архитектуру, планы тестов и т.д., все мы оказывались в ситуации, когда “простое изменение” кода вызывало непредвиденные последствия, иногда даже катастрофические. Кроме того, предлагаемая новая функция может устранять важную будущую функцию системы или затруднять ее реализацию (т.е. воздействие распространяется не только на текущую версию). Есть также трудности, связанные с графиком и бюджетом проекта, за которые отвечает руководство. Пожелания заказчиков о внесении изменений не предполагают официального изменения графика и бюджета, и следует инициировать процесс переговоров до того, как изменение будет принято.

Пожелания заказчиков о внесении изменений не предполагают официального изменения графика и бюджета.

Таким образом, очень важно, чтобы все изменения поступали по *одному каналу*, чтобы определить их воздействие на систему и принять официальное решение, стоит ли вносить это изменение в систему вообще. В небольшом проекте этим официальным каналом может быть лидер проекта, менеджер или кто-нибудь другой, кто “владеет” документом-концепцией и другими артефактами требований, а также имеет полное представление о требованиях к системе и ее проекте.

В более крупных системах или в системах, где затрагиваются интересы множества заинтересованных лиц, такой официальный канал может состоять из нескольких человек (образующих Совет по контролю за изменениями – Change Control Board, CCB), которые совместно несут ответственность за принимаемые решения и обладают знаниями и полномочиями, необходимыми для официального принятия запроса об изменении. (Мы кратко описали CCB в главе 18.)

В любом случае изменение системы нельзя инициировать до тех пор, пока механизм контроля за изменениями не признает его “официальным”.

Шаг 4. Использование системы контроля изменений для их фиксации

Проще всего дело обстоит с внешними изменениями, которые производятся по запросу клиента. Их легко выявлять, и они будут естественным образом включены в проект руководством или органом, осуществляющим контроль за изменениями. Но во время разработки возникает огромное множество иных изменений системы.

Многие предлагаемые изменения, возникающие во время проектирования, кодирования и тестирования системы, могут казаться не связанными с требованиями (например, исправление ошибок кода или проектирования). Тем не менее необходимо оценить их воздействие. А если подходит срок сдачи, следует даже принять сознательное решение о том, какие ошибки оставить в системе (из-за того, что их исправление может дестабилизировать систему в целом и тем самым поставить под угрозу дату сдачи), а какие – устраниТЬ. Помимо этого, многие ошибки могут влиять на требования, вызывать необходимость их согласования или устранения неоднозначности отдельного известного требования.

Нам придется принять сознательное решение о том, какие недостатки оставить в системе.

Иногда даже иепонятно, какой тип изменений запрашивается. Это особенно часто происходит, когда конечные пользователи жалуются на проблемы после окончания разработки или когда члены службы оперативной поддержки передают результаты анализа жалоб пользователя техническим разработчикам. Например, конечный пользователь обращается к службе поддержки с жалобой: "Я пытаюсь ввести данные о новом служащем в мою систему начисления зарплаты, но так как число букв в его фамилии больше 16, программа аварийно завершается". То, что программа прекращает работать, предположительно, является недостатком на уровне кода или проекта. (Возможно, неправильно производится вызов операционной системы или СУБД.) Но даже если программа выдает для таких фамилий нормальное сообщение об ошибке, может существовать ошибка в требованиях; возможно, их необходимо изменить, чтобы фамилии могли содержать до 256 символов. Иногда это может даже затронуть "функцию", так как отдел маркетинга может посчитать большим плюсом то, что их система начисления зарплаты, единственная из предлагаемых, способна работать с фамилиями, содержащими до 256 символов.

Команде следует разработать некую систему для фиксации всех запросов на изменения.

В любом случае необходимо проанализировать ситуацию, а также принять решение о том, где изменение будет реализовано в иерархии документов. Следовательно, команде необходимо разработать формальный метод фиксации *всех* запрашиваемых изменений системы (рис. 34.1). Это может осуществляться с помощью системы отслеживания изменений и неполадок, которая обеспечивает создание централизованного архива запросов, web-ввод элементов из любого физического местоположения, автоматическое отслеживание состояния, автоматическую отметку затрагиваемых частей, а также механизм передачи запросов изменений в систему управления требованиями, если это необходимо. (Мы использовали в рис. 34.1 метафору "огнеупорная стена", чтобы подчеркнуть, что процесс контролируется с целью предотвратить неконтролируемое распространение "пожара" изменений по системе.)

Эту систему следует использовать, чтобы фиксировать *все* предложения и передавать их руководству Совета по контролю за изменениями (CCB) для принятия решения. Совет играет ключевую роль в успехе проекта. Он должен состоять из трех–пяти человек, представляющих интересы основных участников проекта: заказчиков, представителей маркетинга и руководства проекта.

Когда решается вопрос, принять ли запрос об изменении, CCB должен учитывать следующие факторы.

- Влияние изменения на стоимость и функциональные возможности системы.
- Воздействие изменения на заказчиков и внешних участников, не представленных в CCB (других подрядчиков проекта, поставщиков компонентов и т.д.).
- Возможность того, что изменение дестабилизирует систему.

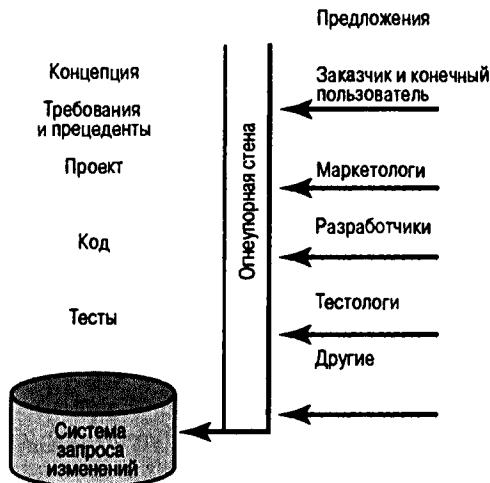


Рис. 34.1. Фиксация изменений

При принятии решения ССВ также несет ответственность за то, чтобы отметить все, на что повлияет изменение, даже если оно не принимается.

После принятия изменения необходимо решить, куда его вставить. (Например, нужно определить, предлагается ли изменить требование или тест.) Последующие изменения будут распространяться по иерархии так, как показано на рис. 34.2.

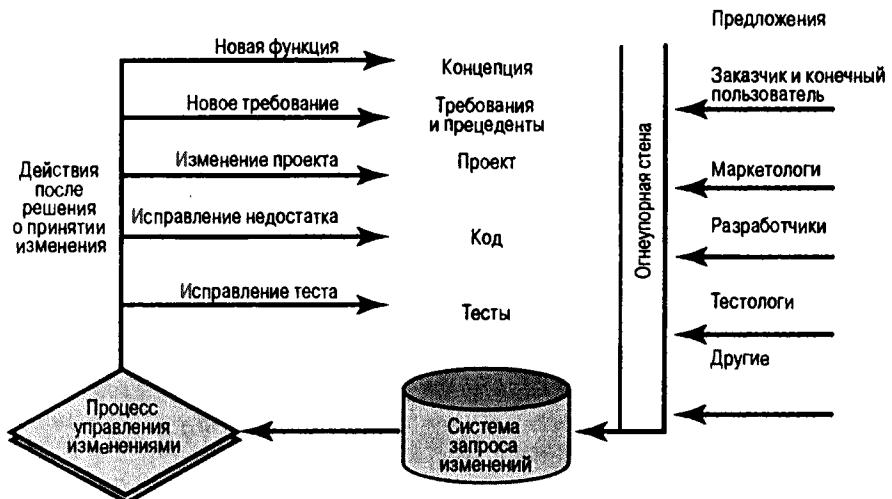


Рис. 34.2. Движение запроса об изменениях

Шаг 5. Иерархическое управление изменениями

То, что все вокруг заинтересованы во внесении изменений, не так уж плохо. Можно даже предположить, что все эти изменения, за исключением сюрпризов, полезны. Проблемой является то, что изменения могут не документироваться и не анализироваться.

Если их тщательным образом не обрабатывать, это может привести к неприятным последствиям. Изменение одного требования может оказать возмущающее воздействие на связанные с ним требования, проектирование или другие подсистемы. К сожалению, это не учитывают представители маркетинга, которые часто просят программиста “быстро и просто” изменить систему.

Проблема осложняется тем, что при отсутствии явно заданного процесса изменения обычно производятся “восходящим” образом. Это означает, что если изменение появляется во время написания кода новой системы, оно, как правило, вносится непосредственно в программный код. Если разработчики предельно дисциплинированы, они могут затем задуматься: “Не вызовут ли изменения, которые мы вносим в код, изменений в проекте? А повлияют ли изменения на уровне проектирования на требования? Окажут ли изменения требований к программе воздействие на документ-концепцию?” (Тем не менее никто не вспомнит о том, что надо хоть что-нибудь сказать об этом изменении группе тестирования, члены которой полагают, что им нужно создавать планы тестирования, ориентируясь на исходные программные требования!)

Программист не имеет права от имени пользователя вводить новые функции и требования непосредственно в код.

Теоретически можно справиться с этим явлением “обратного” возмущения, если все соответствующие документы контролируются сложными автоматическими средствами. Но даже если все документы синхронизируются, обсуждаемые восходящие изменения требований нежелательны. *Программист не имеет права от имени пользователя вносить новые функции и требования непосредственно в код, независимо от его благих намерений.* Аналогично представитель маркетинга, который обращается к программисту с просьбой осуществить такое изменение, когда они потягивают вдвоем пиво в ближайшем пабе, не имеет *официальных* полномочий. Каждая новая функция/требование оказывает влияние на стоимость, график, надежность и риск, связанные с проектом.

Чтобы ослабить возмущающий эффект изменений требований, необходимо выполнять их в иерархии нисходящим образом (рис. 34.3). Как мы уже отмечали, изменения базового уровня документа-концепции можно отражать в отдельном документе Delta Vision, который, как правило, является совсем небольшим подмножеством исходного документа. Но так как изменение документа-концепции может заключаться в удалении функций, нам может понадобиться создать совершенно новый исходный набор требований к программному обеспечению, а это может привести к соответствующим изменениям проектирования, кода и планов тестирования.

Если следовать предложенному в данной книге процессу и воспользоваться поддержкой автоматических средств, нисходящее распространение возмущения будет отражено механизмом трассировки, который используется при построении пирамиды требований. Это позволит нам работать с пирамидой сверху вниз, внося дальнейшие изменения там, где необходимо. Каждое последующее изменение обнаруживает дополнительные “подозрительные связи” или точки более нижнего уровня пирамиды, где необходимо провести дополнительный анализ.

Таким образом, изменение распространяется по иерархии логичным и контролируемым образом. Кроме того, если мы правильно провели инкапсуляцию систем и подсистем и использовали хорошо структурированную стратегию задания требований, изменения будут ограничены областями, непосредственно связанными с измененными требованиями.

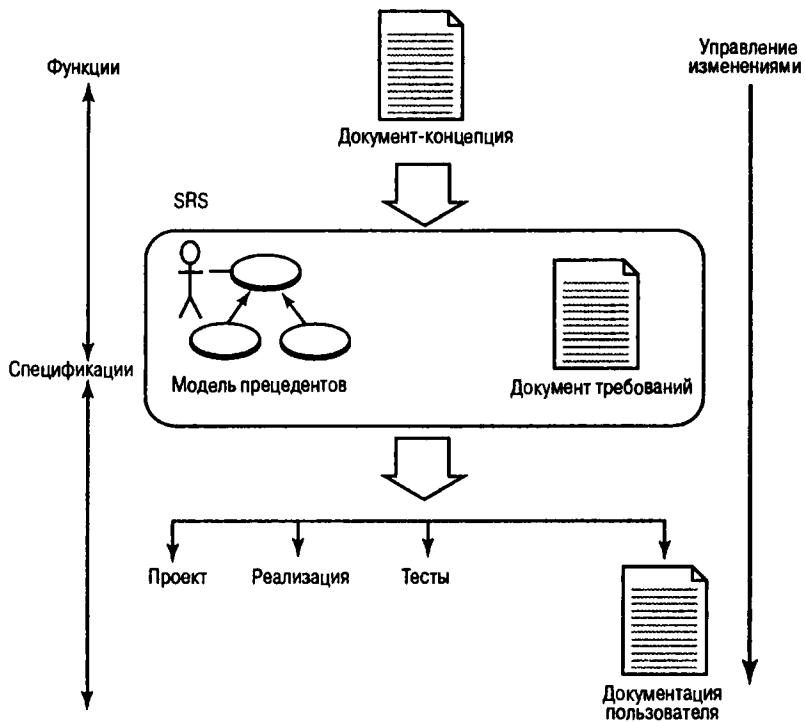


Рис. 34.3. Иерархический характер распространения возмущения

Например, на рис. 34.4 представлен отчет о трассировке, полученный после внесения некоего изменения в HOLIS. Автоматизированное средство отметило трассировочные связи двух функций, FEA3 и FEA5, как подозрительные. Это явилось результатом предложенных изменений указанных двух функций. Необходимо проверить возможное воздействие результатов изменений FEA3 и FEA5 на SR3 и SR4. В свою очередь, возможные исправления SR3 и SR4 могут распространяться вниз к реализации и т.д. Мы вернемся к рассмотрению этого явления далее в этой главе.

Управление конфигурацией требований

Процесс рассмотрения и принятия изменений в некоторых организациях носит название "контроль изменений", "контроль версий", или *управление конфигурацией* (*configuration management*, CM). Интересно, что большинство организаций имеет достаточно строгий процесс управления конфигурацией исходного кода, производимого на стадии реализации жизненного цикла проекта, но не имеет соответствующего процесса для требований к проекту. Даже если в организации имеется формальный процесс создания документа-концепции и требований к программному обеспечению, он зачастую игнорирует многие затрагивающие требования изменения, которые закрадываются в проект на стадии кодирования.

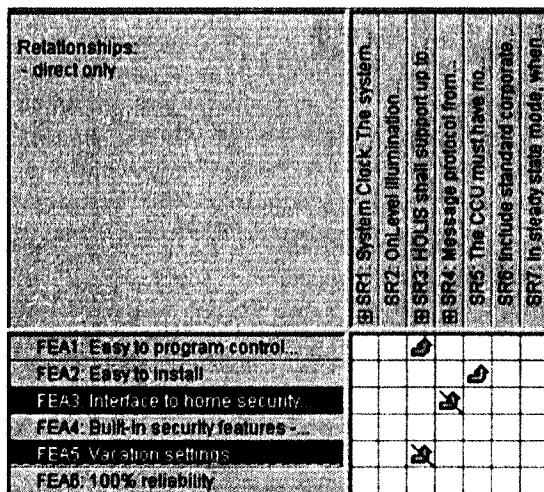


Рис. 34.4. Анализ воздействия изменений с помощью связей траассировки

При наличии современных инструментальных сред достаточно несложно контролировать все элементы иерархии требований посредством управления конфигурацией (рис. 34.5).

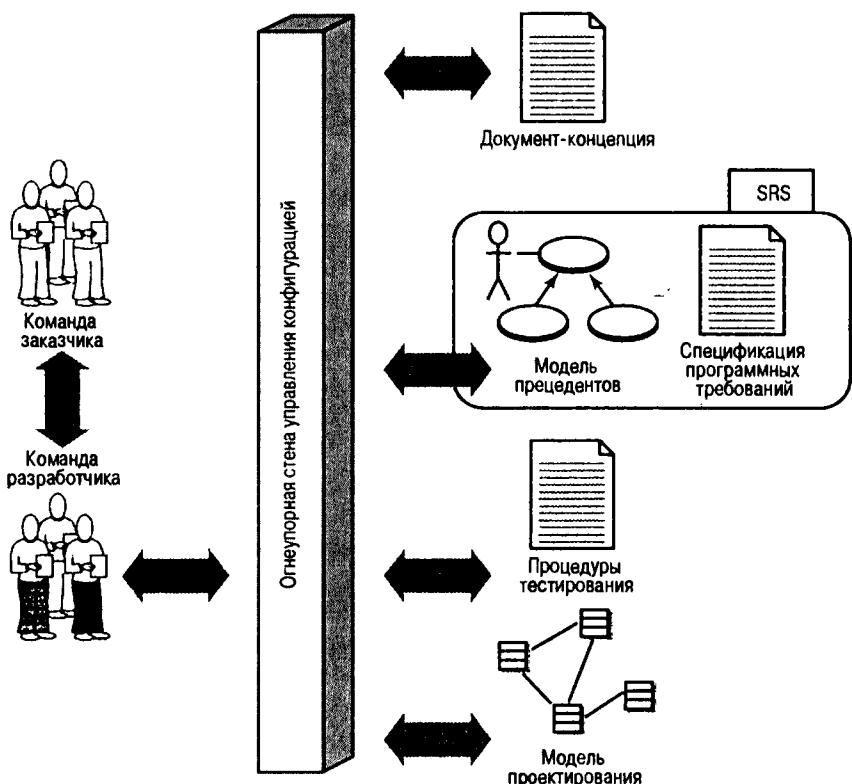


Рис. 34.5. Схема управления конфигурацией требований

Преимущества основанного на СМ процесса управления требованиями очевидны, но мы все же кратко перечислим их.

- Предотвращаются недозволенные и потенциально деструктивные изменения требований.
- Предотвращаются исправления документов требований.
- Упрощается получение и/или восстановление предыдущих версий документов.
- Осуществляется поддержка реализационной стратегии, основанной на задании базового уровня и инкрементных улучшениях и обновлениях системы.
- Предотвращается одновременное обновление документов (т.е. некоординированное обновление различных документов в одно и то же время).

Управление изменениями при поддержке программных средств

Итак, мы предлагаем практический подход к управлению изменениями, предполагая, что у вас имеется набор автоматических средств поддержки данных действий. Если вы захотите использовать собственные методы, реализуемые вручную, часть данного раздела вам не пригодится, но общие идеи все равно заслуживают внимания.

Здесь будут рассмотрены следующие важные вопросы.

- Если предлагается изменить отдельную функцию продукта, какими будут последствия данного изменения? Другими словами, управление изменениями поможет определить, какое количество переделок потребуется. Это может оказать существенное влияние на планирование ресурсов проекта и распределение нагрузки.
- Если предлагается изменить некий элемент, какие другие элементы системы может затронуть данное изменение? Это имеет первостепенную важность как для планирования разработки, так и для клиента.
- В ходе разработки неизбежны неизбежны "повороты". В такой ситуации нужно иметь возможность совершить "откат" требования и исследовать предыдущую его версию. Кроме того, полезно помнить, как и почему было изменено требование. Другими словами, полезно иметь контрольный журнал для каждого требования (это может даже предписываться регулирующими органами как часть процесса проектирования).

Элементы, на которые воздействует изменение

После задания отношений трассировки трассировочные связи можно использовать для управления изменениями. Предположим, что в системе HOLIS необходимо изменить формулировку FEA5 ("Установка режима на время отпуска"), чтобы отразить пересмотренное определение функции продукта (рис. 34.6). Обратите внимание на диагональные линии на трассировочных стрелках в строке для FEA5. Эти линии, "подозрительные связи", предназначены для того, чтобы предупредить о том, что изменение данной функции может повлиять на SR1 и SR3, и поэтому следует произвести их ревизию.

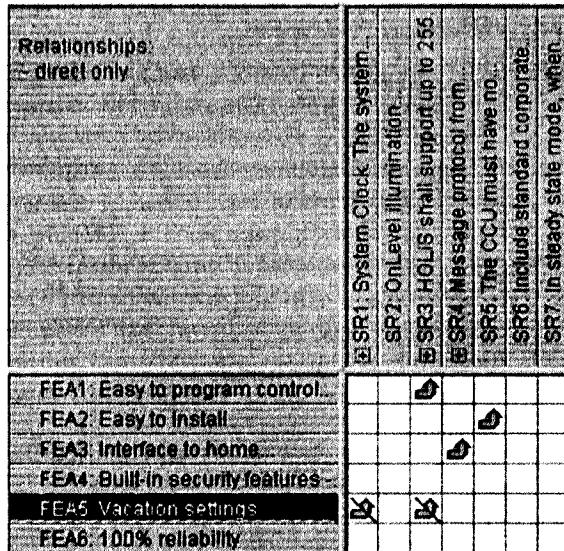


Рис. 34.6. Фрагмент матрицы трассировки после изменения FEA5

По мере развития проекта неизбежно будут предлагаться изменения различных его элементов: от высокоуровневого документа-концепции до спецификации, реализации и тестов. Повсюду при возникновении изменения следует использовать “подозрительные связи”, чтобы отметить отношения, на которые могло повлиять данное изменение. Действия по управлению изменениями обычно заключаются в одном из следующих двух шагов.

1. Если изменение функции не влияет на требование, необходимо только очистить “подозрительную связь”. Заметим, что если позднее данная функция вновь будет меняться, связь снова будет отмечена как подозрительная.
2. Если функция действительно влияет на требование, может понадобиться переделать подвергшийся воздействию элемент. Например, предлагаемое изменение функции может потребовать изменения спецификации требования. После его редактирования обнаружится, что дополнительные “подозрительные связи” теперь предупреждают о возможном воздействии уже этого изменения. Значит, необходимо будет исследовать на предмет изменений эти связи и т.д.

Возможность управления изменениями должна существовать на различных уровнях отношений трассировки. Так, изменение функции документа-концепции может затронуть несколько требований к программному обеспечению в SRS и/или некоторые прецеденты, которые могут, в свою очередь, воздействовать на несколько компонентов реализации, а те – на один или несколько планов тестирования. Необходимо отслеживать трассировочные связи в двух направлениях. Например, изменение спецификации плана тестирования может заставить рассматривать возможное влияние на компоненты реализации. В свою очередь, изменение компонента реализации может потребовать повторной проверки затронутых программных требований и даже функций верхнего уровня, которые связаны с этим компонентом посредством трассировочных отношений.

Контрольный журнал изменений

Полезно вести контрольный журнал изменений, где, в частности, фиксируются изменения отдельных требований. При поддержке вспомогательной программы это позволит работать с каждым требованием отдельно, независимо от того, частью какого документа или модели оно является. Все изменения требования автоматически фиксируются (образуя историю изменений данного требования), и их можно впоследствии проверять и пересматривать.

В истории изменений указывается текущая формулировка требования, в том числе текущие значения всех его атрибутов (т.е. краткое описание данного требования).

Кроме того, в истории изменений в хронологическом порядке представлена последовательность всех предшествующих изменений данного требования и его атрибутов. Вспомогательная программа должна автоматически фиксировать все изменения формулировки требования, а также значений его атрибутов.

Как только программа обнаруживает изменения, она автоматически фиксирует его. Кроме того, автоматически записывается автор, дата и время изменения.

Программа также должна предоставлять возможность вводить описание изменения. Как правило, вводится одно-два предложения, чтобы объяснить, почему было сделано изменение, сославшись на другие документы, имеющие отношение к этому изменению, и т.д. Документирование изменения позволит впоследствии вспомнить мотивы его внесения. Это очень важно при любой проверке тех изменений, которые влияют на возможные претензии, эффективность и безопасность изделия и его программного обеспечения.

На рис. 34.7 представлена распечатка фрагмента истории изменений требования SRS (SR4.4). Отметим, что история изменений организована в обратном хронологическом порядке и содержит записи изменений как текста (1.0001 в сравнении с 1.0000), так и значений отдельных атрибутов.

Изменения текста могут быть как незначительными (например, изменение пунктуации), так и более существенными (как в случае SR4.4). В любом случае всякое изменение должно надлежащим образом фиксироваться автоматическим средством управления изменениями.

Управление конфигурацией и управление изменениями

История изменений должна существовать на трех уровнях детализации проекта.

1. При самой подробной детализации в историю изменений записываются все изменения каждого отдельного требования. Этот уровень детализации отражен на рис. 34.7.
2. На среднем уровне следует автоматически вести аналогичную историю изменений для каждого документа проекта. История на уровне документов обычно ведется системой контроля исходного кода или системой контроля документов.
3. На общем уровне необходимо автоматически вести историю изменений проекта в целом. Проект и архивы могут полностью включаться в систему управления конфигурацией.

Другими словами, необходим набор вспомогательных средств, обеспечивающих полностью автоматическую всеобъемлющую и естественную интеграцию со стандартными приложениями, которые будут помогать в решении задач управления конфигурацией, возникающих при выполнении крупных проектов разработки программного обеспечения.

Название проекта
Версия: 1.0003. Код версии:

SR4.4
Опубликовал: Дон

SR4.

Текст: Подтверждение приема сообщения. В ответ на получение сообщения от Управления включением (пульта), ЦБУ (CCU) должен послать следующее сообщение:
Rqmt
Местоположение: HOLIS CCU SRS
Трассируется от:
Трассируется к:
Атрибуты

Приоритет	Статус	Стоим.	Сложность	Стабильность	Предназн.
Средний	Подтвержд.		Средняя	Средняя	

Исправления:

1.0001	04/18/99 9:06 AM выполнил: Дон
Описание изменения:	Значительное изменение требования для удобства тестирования. Изменен текст требования.
Текст:	Подтверждение приема сообщения. В ответ на получение сообщения от Управления включением (пульта), ЦБУ (CCU) должен послать следующее сообщение:
1.0000	04/15/99 5:32 PM выполнил: Дон
Описание изменения:	Требование создано. СЛОЖНОСТЬ: <нет ввода>-Сред. ПРИОРИТЕТ: <нет ввода>-Сред. СТАБИЛЬНОСТЬ: <нет ввода>-Сред. СТАТУС: <нет ввода>-Подтвержд.
Текст	Контрольная сумма. Контрольная сумма для сообщения вычисляется и проверяется по стандарту CRC2.3.

Рис. 34.7. История изменений SR4.4

Заключение

Безусловно, по мере развития проекта требования будут меняться, но эти изменения не обязательно дестабилизируют процесс разработки. Если создан всеобъемлющий процесс контроля изменений, а артефакты требований подконтрольны используемой командой разработчиков системе управления конфигурацией, команда будет хорошо подготовлена к решению основных задач управления изменениями.

Важно понимать, что управление изменениями в крупном проекте обычно является слишком объемной задачей, чтобы ее можно было решать вручную. Необходим *процесс*, чтобы контролировать, каким образом изменения попадают в проект. Также необходимы автоматические средства для того, чтобы *понять разветвления* изменения, т.е. найти все затронутые им элементы проекта.

Заключение части 6

Часть 6, “Построение правильной системы”, завершает переход от понимания проблемы к реализации системы-решения.

Спроектировать и реализовать правильную систему достаточно сложно. Один из методов состоит в том, чтобы использовать требования и прецеденты для создания архитектуры и проекта реализации. Мы также обсудили *верификацию*, аналитический подход, который позволяет постоянно отслеживать эволюцию функций, требований, проекта и реализации системы. Поддержка верификации осуществляется с помощью методов *трассировки*, которые позволяют связать друг с другом части проекта.

Методы трассировки позволяют убедиться, что все необходимое в проекте присутствует и учитывается. С их помощью можно также исключить ненужные элементы. Хотя верификация является аналитическим методом, необходимо помнить о важности *размышлений*, нельзя ограничиться чисто механическим применением методов верификации.

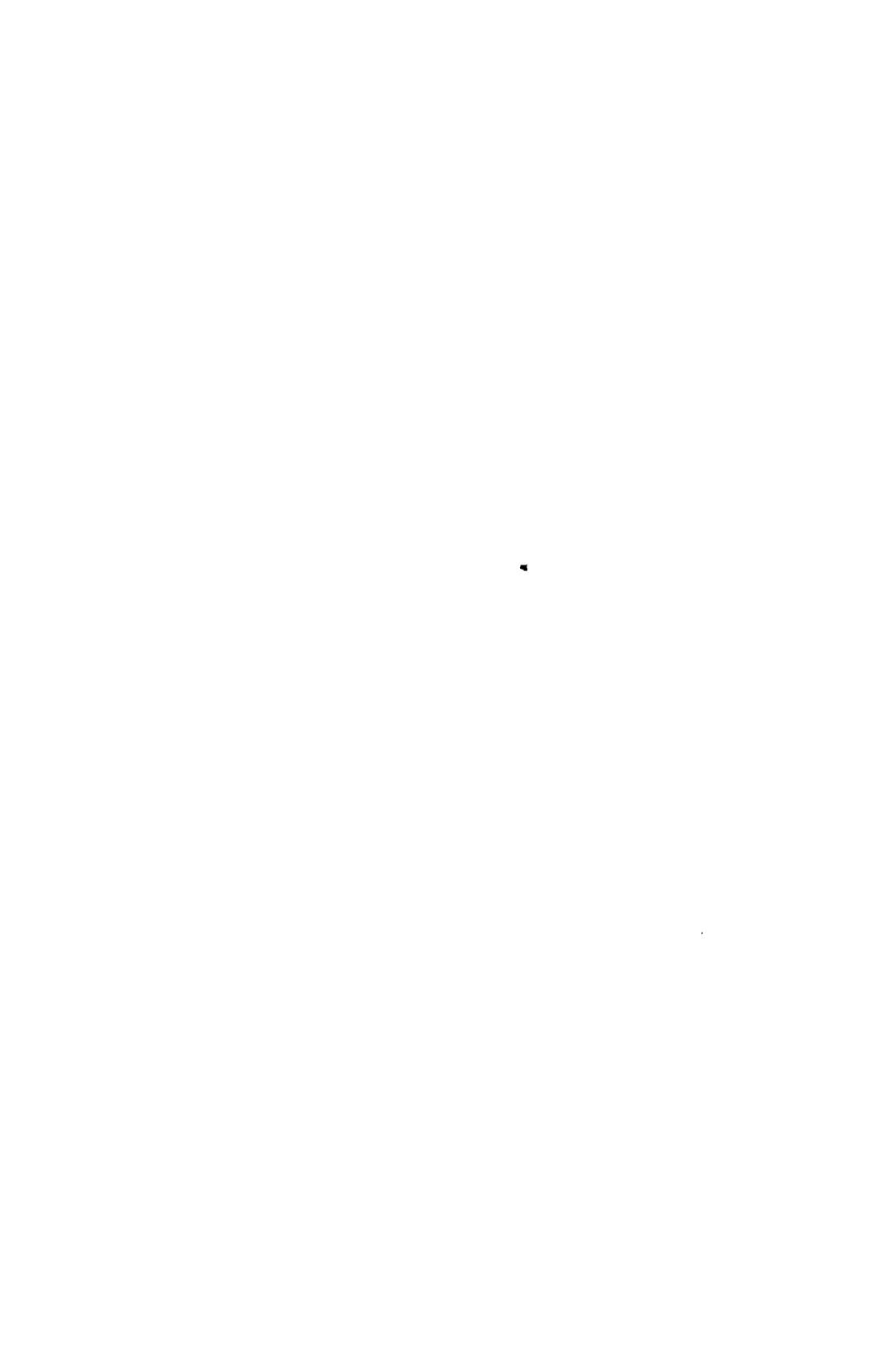
Еще одной составной частью подхода, призванного подтвердить корректность создаваемой системы, является *проверка правильности* (*validation*). В данном случае выполняются действия по тестированию и применяются методы трассировки, чтобы убедиться, что система соответствует предъявляемым к ней требованиям. Затем проводятся приемо-сдаточные испытания, призванные продемонстрировать, что в среде заказчика система работает так, как нужно, и действительно решает поставленную проблему.

Чтобы решить, какая часть системы нуждается в верификации и проверке правильности и в каком объеме, производится оценка и анализ рисков. Затраты на эти действия следует контролировать с помощью анализа дивидендов (ROI).

Наконец, критически важным аспектом построения правильной системы является процесс управления изменениями. Изменения — это жизнь; их можно и нужно *планировать* и *обрабатывать*. Управление изменениями дает уверенность в том, что созданная система является правильной и, более того, будет правильной и в дальнейшем.

По окончании части 6 можно переходить к последней главе. Глава 35 поможет вам применить приобретенные навыки и взять хороший старт в вашем следующем проекте.





Глава 35

С чего начать

Посвящение

На протяжении многих лет мы (все, кто внес свой вклад в появление этой книги) обучили тысячи студентов, интересующихся вопросами управления требованиями, а также сами многому научились у них. Как известно, не существует единственно верного способа осуществления управления требованиями. Ни один метод выявления требований не является универсальным; ни один процесс не подойдет всем без исключения командам. Проекты очень отличаются по уровню сложности и масштабу. Приложения имеют совершенно различное содержание и относятся к разным отраслям.

Управление требованиями – очень обширная и глубокая тема. В аудитории постоянно возникает чувство, что студенты ощущают потребность в более четко определенном процессе; если хотите, неком рецепте применения полученных на занятиях знаний. “Вы рассказали нам слишком много, – говорят они. – Укажите нам один основной процесс, от которого можно оттолкнуться. Мы знаем, что это непросто, но мы будем счастливы при необходимости модифицировать его применительно к нашему проекту. Опишите четкий пошаговый процесс, чтобы мы могли более успешно применить то, чему научились. Скажите же наконец, с чего начать!”

Данная глава посвящается этим студентам и тем, кто разделяет их точку зрения.

Чему мы научились

Итак, подведем итог того, что мы изучили в данной книге.

Введение

В главах введения обсуждалась проблема, которая состоит в том, что в нашей отрасли зачастую не удается создать качественные приложения в срок и в рамках бюджета. Некоторые причины этого известны. Чаще всего среди проблем, с которыми столкнулись не достигшие своих целей проекты, упоминаются недостаток информации от пользователя, неполные требования и спецификации, а также изменения требований и спецификаций.

Возможно, это происходит потому, что зачастую разработчики и заказчики считают, что “даже если мы не очень точно знаем, чего хотим, лучше побыстрее приступить к реализации, так как мы и так выбились из графика и нам никогда. Мы можем уточнить требования позднее”. Но слишком часто подобный подход приводит к хаотическим действиям при разработке, когда никто не знает, чего на самом деле хочет пользователь и что в действительности делает созданная на данный момент система.

Как узнать, что должна делать система? Как отслеживать текущее состояние требований? Как определить воздействие изменений? Чтобы решить эти проблемы, мы рекомендовали руководствоваться философией управления требованиями, которое определили следующим образом.

Управление требованиями – это систематический подход к выявлению, организации и документированию требований к системе, а также процесс, в ходе которого вырабатывается и обеспечивается соглашение между заказчиком и выполняющей проект группой по поводу меняющихся требований к системе.

Поскольку история развития программирования и его обозримое будущее связаны с возрастанием сложности, мы отаем себе отчет, что разработкой программного обеспечения должны заниматься хорошо организованные и хорошо обученные команды разработчиков. Каждый член команды в той или иной степени будет привлекаться к управлению требованиями к проекту. Команде необходимо выработать профессиональные приемы для понимания потребностей пользователей, управления масштабом приложения и построения системы, удовлетворяющей эти потребности. Чтобы успешно справиться с задачей управления требованиями, группа разработчиков должна работать как настоящая команда.

Набор приемов 1. Анализ проблемы

В части 1 рассматривались приемы, которые команда может применить для того, чтобы *понять решаемую проблему до начала разработки приложения*. Для этого мы предложили использовать состоящий из пяти этапов метод анализа проблемы.

1. Достигнуть соглашения по определению проблемы.
2. Выделить основные причины – проблемы, стоящие за проблемой.
3. Выявить заинтересованных лиц и пользователей, чье колективное мнение в конечном итоге определяет успех или неудачу системы.
4. Определить, где приблизительно находятся границы решения.
5. Понять ограничения, которые будут наложены на команду и решение.

Следуя этому процессу, команда будет легче справиться с предстоящей задачей – *предложить решение рассматриваемой проблемы*.

Мы также отмечали, что для анализа проблемы можно использовать самые разнообразные методы. В частности, мы рассмотрели моделирование бизнес-процессов, специальный метод анализа проблемы, который достаточно хорошо зарекомендовал себя для сложных информационных систем, осуществляющих поддержку ключевых бизнес-инфраструктур. Команда может применять этот метод как для понимания путей развития данного бизнеса, так и для выявления, где внутри системы можно наиболее эффективно развернуть приложения. Мы также выяснили, что определенная нами бизнес-модель будет иметь параллельные конструкции в программном приложении, и мы будем использовать эту общность при проектировании программного обеспечения. Выявленные на данном этапе бизнес-прецеденты будут позднее применяться для определения требований к самому приложению.

В программных приложениях, которые мы отнесли к классу встроенных систем, для анализа проблемы применяются методы системной инженерии, позволяющие осуществить разбиение сложной системы на подсистемы. Этот процесс помогает понять, где должны находиться программные приложения и каким общим целям они служат. При

этом оказалось, что мы в чем-то усложняем проблему требований, создавая новые подсистемы, для которых в свою очередь нужно заниматься пониманием требований.

Набор приемов 2. Понимание потребностей пользователей

Мы начали часть 2 с рассмотрения трех “синдромов”, которые значительно усложняют задачу понимания реальных потребностей пользователей и других заинтересованных лиц. Описание синдромов “да, но...”, “неоткрытые руины” и “пользователь и разработчик” помогло нам лучше понять предстоящие проблемы и получить представление о том, в какой среде будут применяться методы выявления, разработанные для понимания потребностей пользователей.

Так как команды редко получают совершенные спецификации требований к создаваемой системе, они должны сами добывать информацию, необходимую им для успешной работы. Термин *выявление требований* очень точно отражает данный процесс, в котором команда должна играть более активную роль.

Чтобы помочь команде решить эти проблемы и лучше понять потребности пользователей и других заинтересованных лиц, можно использовать различные методы.

- Интервьюирование и анкетирование
- Совещание, посвященное требованиям
- Мозговой штурм и отбор идей
- Создание раскадровок
- Прецеденты
- Обыгрывание ролей
- Создание прототипов

Хотя ни один метод не является универсальным, каждый из них позволяет лучше понять потребности пользователей и тем самым превратить “неясные” требования в требования, которые “лучше известны”. Каждый из этих методов “срабатывает” в определенных ситуациях, однако мы отдаляем предпочтение совещанию, посвященному требованиям, и мозговому штурму.

Набор приемов 3. Определение системы

В части 3 мы перешли от понимания потребностей пользователя к определению решения. При этом мы сделали свои первые шаги из области проблемы, вотчины пользователя, в область решения, где наша задача состоит в том, чтобы определить систему для решения имеющейся проблемы.

Мы поняли, что для сложных систем требуются всеобъемлющие стратегии управления информацией о требованиях, и рассмотрели несколько способов организации данной информации. Оказалось, что на самом деле существует информационная иерархия; она начинается с потребностей пользователей, переданных с помощью функций, которые затем превращаются в более подробные требования к программному обеспечению, выраженные посредством прецедентов или традиционных форм описания. Эта иерархия отражает уровень абстракции при рассмотрении области проблемы и области решения.

После этого мы рассмотрели процесс определения отдельного программного приложения и затратили некоторое время на определение документа-концепции для приложе-

ния такого типа. Мы считаем, что документ-концепция, модифицированный в соответствии с конкретным содержанием программных приложений компании, является чрезвычайно важным, и его необходимо иметь в каждом проекте.

Мы также осознали, что без лидера – человека, который будет защищать требования к приложению и поддерживать потребности клиента и команды разработчиков – нельзя быть уверенным в том, что будут приняты необходимые жесткие решения. Возможно возникновение дрейфа требований, задержек, а также принятие неоптимальных решений, вызванное приближением срока окончания проекта. Поэтому мы решили назначить лидера, который будет владеть документом-концепцией и содержащимися в нем функциями. В свою очередь, лидер и команда должны создать совет по контролю за изменениями, который призван помогать лидеру в принятии действительно сложных решений и гарантировать, что изменения требований будут приниматься только после их обсуждения.

Набор приемов 4. Управление масштабом

В части 4, “Управление масштабом”, мы выяснили, что проблема масштаба проекта является весьма типичной. Проекты, как правило, инициируются с объемом функциональных возможностей, вдвое превышающим тот, который команда может реализовать, обеспечив приемлемое качество. Это не должно нас удивлять: заказчики хотят большего, маркетинг хочет большего и мы также желаем большего. Тем не менее нам необходимо ограничить себя, чтобы иметь возможность представить в срок хоть что-нибудь.

Мы рассмотрели различные методы задания очередности выполнения (приоритетов) и ввели понятие базового уровня (совместно согласованного представления о том, в чем будут состоять ключевые функции системы как рабочего продукта нашего проекта) – понятие, задающее точку отсчета и ориентир для принятия решений и их оценки. Если масштаб и сопутствующие ожидания заказчиков превышают реальные, в любом случае придется сообщать некие неприятные новости. Мы остановились на философии привлечения заказчика к процессу принятия трудных решений. В конце концов, мы являемся только исполнителями, а принимать решения должны наши заказчики, ведь это их проект. Поэтому вопрос стоит так: что обязательно должно быть сделано в следующей версии при имеющихся ресурсах проекта?

Даже в этом случае нам придется вести переговоры; в некотором смысле вся жизнь и, конечно, весь бизнес состоят из переговоров, и мы не должны этому удивляться. Мы кратко перечислили некоторые приемы ведения переговоров и намекнули, что они могут понадобиться команде.

Нельзя ожидать, что данный процесс полностью решит проблему масштаба, точно так же как никакой другой процесс в отдельности не решит проблемы разработки приложений. Но указанные шаги окажут заметное воздействие на размеры проблемы, позволят разработчикам приложения сконцентрировать свои усилия на критически важных подмножествах функций и в несколько приемов предоставить высококачественные системы, удовлетворяющие или превосходящие ожидания пользователей. Привлечение заказчика к решению проблемы управления масштабом повышает обязательства сторон, способствует росту взаимопонимания и доверия между заказчиком и командой разработчиков приложения. Имея всеобъемлющее определение продукта (документ-концепцию) и сократив масштаб проекта до разумного уровня, мы можем надеяться на успех в следующих фазах проекта.

Набор приемов 5. Уточнение определения системы

В части 5 мы выяснили, что в требованиях должны быть полно и сжато зафиксированы потребности пользователей в таком виде, чтобы разработчик мог построить удовлетворяющее их приложение. Кроме того, требования должны быть достаточно конкретными, чтобы можно было определить, когда они удовлетворены. Зачастую именно команда обеспечивает эту конкретизацию; это еще одна возможность убедиться, что определяется правильная система.

Существуют различные возможности организации и документирования этих требований. Мы остановились на пакете Modern SRS Package, логической конструкции, которая позволяет документировать требования в виде прецедентов, документов, форм баз данных и т.д. Хотя мы внесли несколько предложений относительно того, как организовать такой пакет, мы считаем, что форма не так важна, лишь бы пакет содержал все, что необходимо.

Вся разработка должна вытекать из требований, зафиксированных в пакете Modern SRS Package, а все спецификации пакета должны найти отражение в действиях разработки. Таким образом, все виды деятельности являются отражением пакета и наоборот. Пакет Modern SRS Package является "живой" сущностью; его следует пересматривать и обновлять на протяжении жизненного цикла проекта. В пакете указывается, какие функции должны осуществляться, а не то, как они осуществляются. Он используется для задания функциональных и нефункциональных требований, а также ограничений проектирования.

Мы также предложили набор показателей, которые можно использовать для оценки качества пакета и содержащихся в нем элементов. Если необходимо, документация требований может дополняться одним или несколькими более формальными либо более структурированными методами спецификации. Таким образом, пакет Modern SRS Package содержит все детали, которые необходимы для построения (*реализации*) правильной системы.

Набор приемов 6. Построение правильной системы

Проектирование и реализация правильной системы – самая трудоемкая задача. Один из полезных методов состоит в использовании требований и прецедентов в качестве основы архитектуры и проекта реализации.

Постоянно отслеживать эволюцию функций и требований, а также технического проекта и реализации позволяет *верификация* (*verification*). Ее поддержка осуществляется путем использования методов *трассировки*, что позволяет связать друг с другом части проекта. С помощью трассировки можно удостовериться в том, что

- все элементы проекта учтены;
- все элементы служат некой цели.

Хотя верификация является аналитическим методом, необходимо помнить о важности *размышлений*. Нельзя ограничиться механическим применением методов верификации.

Проверка правильности (*validation*) является второй составной частью приемо-сдаточных испытаний (V&V), призванных подтвердить корректность построенной системы. Основное внимание при ее проведении уделяется тестированию и использованию методов трассировки для отбора компонентов системы, нуждающихся в тестировании. Методы проверки правильности призваны гарантировать, что

- все элементы надлежащим образом тестируются;
- все тесты служат некой полезной цели.

Чтобы принять решение о том, какая часть системы нуждается в верификации и проверке правильности и в каком объеме, можно применить анализ и оценку рисков.

Мы также отметили, что периодическое проведение приемочных испытаний поможет нашим проектам “не свернуть” с правильного пути.

Наконец, построение правильной системы во многом зависит от надлежащей обработки изменений. Изменения – неотъемлемая часть жизни, это нужно учитывать при создании *планов*, а также необходимо разработать процесс, с помощью которого можно управлять изменениями. Управление изменениями дает уверенность в том, что созданная система является *правильной* и, более того, будет *правильной* и в дальнейшем.

Рецепт работы с требованиями

После того как мы “освежили” в памяти изученный материал, можно приступить к изложению “рецепта”. Чтобы рецепт не был слишком сложным и объемным, необходимо сперва принять некие упрощающие предположения. Это поможет понять, для каких типов систем его можно применять, а также *чего можно ожидать от подобного рецепта*.

Упрощающие предположения

- Те, кто применяют наш рецепт, прочитали и поняли книгу и/или прошли обучение, соответствующее ее методологии.
- Рассматриваемое приложение является отдельным приложением, а не состоящей из подсистем системой или еще более крупным проектом. Также предполагается, что не существует оговоренных контрактом требований о специальном формате документов.
- Команда разработчиков небольшая или средняя (примерно 10–30 человек).
- Программное обеспечение разрабатывается для использования другими: неким внешним заказчиком, с которым команда может достаточно легко связываться.
- Это новое приложение, т.е. команда может при создании проекта “начинать с нуля”.
- Команда будет использовать современные методы разработки программного обеспечения и знакома с основными концепциями прецедентов и итеративной разработки.
- В распоряжении команды имеется некий набор вспомогательных автоматических средств, в том числе средства управления требованиями, средства моделирования, а также система запросов изменений и средства управления изменениями.

Рецепт

Шаг 1. Понимание решаемой проблемы

- A. Выполните состоящий из пяти этапов процесс анализа проблемы.
 1. Достигнуть соглашения по определению проблемы.
 2. Выделить основные причины – проблемы, стоящие за проблемами.
 3. Выявить заинтересованных лиц и пользователей.
 4. Определить границу системы решения.
 5. Выявить ограничения, которые необходимо наложить на решение.
- (Используйте в качестве руководства часть 1.)

- Б. Доведите до сведения внешних участников постановку проблемы и убедитесь, что вам удалось достигнуть согласия по определению проблемы, прежде чем двигаться дальше.

Шаг 2. Понимание потребностей пользователей

- А. Создайте структурированное интервью, используя в качестве образца шаблон из части 2, модифицированный применительно к конкретному приложению.
- Б. Проведите интервью с 5–15 пользователями/заинтересованными лицами, которые были выявлены на шаге 1.
- В. Подведите итоги интервью, сформулировав 10–15 наиболее часто упоминавшихся потребностей пользователей или используя “метод выразительных цитат”; т.е. запишите 10–15 особенно запомнившихся высказываний участников, в которых их потребности описаны их собственными словами.
- Г. Используйте цитаты или сформулированные вами описания потребностей для начала создания пирамиды требований. Начинайте задание трассировки требований.
- Д. Проведите совещание по вопросу требований к проекту. Используйте как общие, так и связанные с конкретным проектом подготовительные документы (связанные с конкретным проектом данные берутся из п. В).
1. Проведите сеанс мозгового штурма, чтобы выявить/уточнить функции.
 2. Выполните отсечение идей и определите приоритеты функций.
 3. Используйте классификацию функций, чтобы определить их как критические, важные и полезные.
- Е. Проводите совещание один или два раза в год, чтобы обеспечить постоянный приток структурированных мнений заказчика.
- Ж. Создайте раскадровки для всех инновационных концепций. Представьте их и покажите соответствующее множество прецедентов пользователям, чтобы удостовериться, что вы правильно их поняли.
- З. Страйтесь сделать так, чтобы ваш процесс предоставлял пользователю хотя бы один оценочный прототип, который пользователи могут тестировать в своей среде.

Шаг 3. Определение системы

- А. Воспользуйтесь понятием документа-концепции и создайте его шаблон применительно к нуждам вашего проекта.
- Б. Сформулируйте определение позиции продукта. Ознакомьтесь с ним всех участников и убедитесь, что они с ним согласны. Если это не так, остановитесь и добейтесь согласия. Обязательно убедитесь в согласии ваших клиентов.
- В. Введите все выявленные на шаге 2 и полученные от разработчиков и представителей маркетинга функции в документ-концепцию. Произведите их обратную трассировку к потребностям пользователей. Используйте в документе-концепции атрибуты приоритета (критический, важный, полезный), риска (высокий, низкий, средний), трудоемкости (команда-месяцы), стабильности (высокая, низкая, средняя) и реализации (v1.0 и т.д.).

- Г. Разработайте иллюстративные прецеденты в приложении документа-концепции, чтобы его функции были всем понятны.
- Д. Сделайте документ-концепцию “живым” документом проекта. Опубликуйте его, чтобы было легче его использовать и пересматривать. Автоматически сделайте ответственного за документ-концепцию *официальным каналом* изменения функций. Используйте документ Delta Vision (подробнее о нем далее). Следует сделать так, чтобы ваша компания всегда имела соответствующий современному состоянию проекта документ-концепцию.

Шаг 4. Постоянное управление масштабом и контроль изменений

- А. На основе выполненных командой оценок трудозатрат определите *базовый уровень* для каждой версии документа-концепции, используя атрибут “номер версии”.
- Б. Следует достигнуть *соглашения с заказчиком относительно масштаба*. Помогите команде принять жесткие решения по масштабу и двигайтесь дальше.
- В. Придерживайтесь принципов *итеративной разработки* и обучайте этим принципам команду. Повсюду обсуждайте и корректируйте ожидания.
- Г. Управляйте изменениями, используя базовый уровень. Для фиксации новых функций, возникающих в результате нормального течения событий, используйте документ Delta Vision. Убедитесь, что все предложенные функции записаны и ничего не потеряно. Уполномочьте совет по контролю за изменениями принимать жесткие решения.
- Д. Установите *систему управления запросами изменений*, чтобы фиксировать все запросы на изменения и гарантировать, что они поступают через эту систему в совет по контролю за изменениями.

Шаг 5. Уточнение определения системы

- А. Обеспечьте наличие на всех этапах спецификации требований к программному обеспечению (использующей организационную форму пакета Modern SRS Package), которая задает полный набор функционального и нефункционального поведения продукта. Разработайте подробные прецеденты для основных функций системы.
- Б. Пусть этой задачей займется группа разработчиков или группа тестирования. Помогите им получить необходимые навыки и помогайте по мере необходимости. Используйте формальные методы анализа только в тех случаях, когда неправильное понимание недопустимо.
- В. Осуществите трассировку требований к прецедентам и функциям и обратно.
- Г. Убедитесь, что заданы все нефункциональные требования к системе. Используемый образец поможет удостовериться, что вы задали необходимые вопросы.

Шаг 6. Построение правильной системы

- А. Проведите анализ и оценку рисков, чтобы определить, для каких элементов неправильная реализация недопустима. Разработайте план действий по верификации и проверке правильности, основываясь на результатах этих оценок.

- Б. На этом этапе привлекайте к решению задач управления требованиями отдел тестирования. Пусть сотрудники отдела подключаются к планированию тестов с самого начала. Группа тестирования должна разработать тестовые процедуры и примеры, которые трассируются к прецедентам, а также функциональным и ненесущим требованиям.
- В. Если в вашей компании есть независимый отдел гарантии качества, ему должна отводиться роль в отслеживании и оценке процесса управления требованиями и плана V&V.
- Г. При создании модели проектирования следует полагаться на прецеденты и их реализации, чтобы связать элементы проекта с требованиями.
- Д. Обеспечьте периодическое проведение приемочных испытаний по окончании значительных этапов (milestones), чтобы проверить правильность результатов работы и обеспечить постоянное участие заказчиков.

Шаг 7. Управление процессом работы с требованиями

- А. Лидер должен нести персональную ответственность за документ-концепцию, еженедельно (вместе с командой) оценивать состояние дел и регулярно готовить отчеты и запросы, способствующие этим действиям.
- Б. Отслеживайте процесс спецификации требований к программному обеспечению, чтобы убедиться, что концепция надлежащим образом осуществляется в детализированных требованиях.
- В. Осуществляйте руководство процессом контроля за изменениями, который проводит ССВ, чтобы гарантировать, что перед тем, как разрешить внесение существенных изменений в систему, производится оценка поступивших предложений.

Шаг 8. Примите наши поздравления! Вы выпустили продукт!

Теперь — к следующей версии!

Итак, поздравляем Вас! Вам и вашей команде удалось выпустить качественную (хотя и в несколько сокращенном масштабе) первую версию новой системы. Вы сделали это качественно и даже стильно и заслужили провести в конце года отпуск в кругу семьи. Ваши заказчики довольны. Нельзя сказать, что они в восторге; многие из них рассчитывали на большее. Но они все еще остаются вашими заказчиками и с нетерпением ожидают следующей версии.

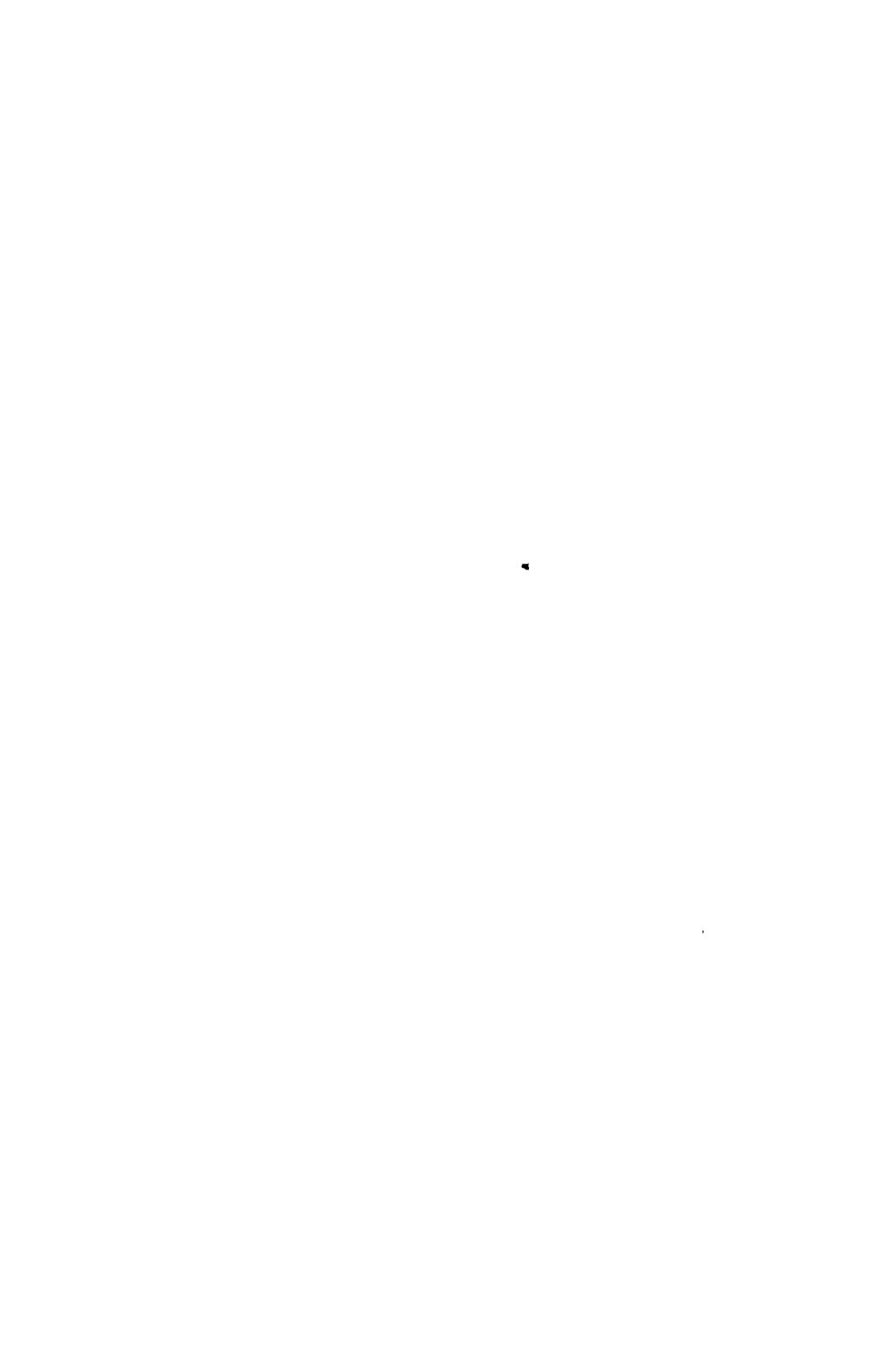
- I. Возвращайтесь (примерно) к шагу 2(Д) и стройте следующую итерацию вашей системы.

Между прочим, не забудьте отметить это событие! Создание классных продуктов и систем – очень захватывающее занятие! И нам оно нравится!



Приложения

- **Приложение А.** Артефакты системы HOLIS
- **Приложение Б.** Образец документа-концепции
- **Приложение В.** Образец пакета Modern SRS Package
- **Приложение Г.** Принципы управления требованиями в стандартах SEI-CMM и ISO 9000
- **Приложение Д.** Принципы управления требованиями в Rational Unified Process



Приложение А

Артефакты системы HOLIS

Замечание. В данном рабочем примере все названия, а также имена участников являются вымышленными.

Предварительная информация для рабочего примера

Компания Lumenations, Ltd.

Компания Lumenations, Ltd., более 40 лет являлась всемирно известным поставщиком коммерческих осветительных систем для театра. В 1999 году ее годовой доход составил приблизительно 120 миллионов долларов, а объем продаж стабилизировался. Lumenations – открытая компания, и недостаточный рост продаж не, что еще хуже, отсутствие реальных предложений по его повышению оказали крайне негативное влияние на компанию и ее акционеров. Последнее ежегодное собрание акционеров было весьма неприятным, так как не было сказано ничего нового о перспективах роста компании. Цена акции ненадолго поднялась до 25\$ прошлой весной на волне новых заказов, но затем вновь опустилась до 15\$.

В индустрии производства оборудования для театров в целом мало новых разработок. Данная отрасль является устоявшейся и весьма консолидированной. Поскольку оборотные средства компании Lumenations ограничены, а капитализация весьма умерена, приобретение другой компании в данном случае не является выходом.

Что действительно нужно – это новая позиция на рынке сбыта, не очень отличающаяся от того, в чем специализируется компания, но где есть простор для роста дохода и прибыли. Проведя тщательное исследование рынка и истратив немало средств на консультации по маркетингу, компания приняла решение выйти на новый рынок *систем автоматического освещения для жилых помещений самого высокого класса*. Этот рынок растет приблизительно на 25–35% ежегодно. Еще отраднее то, что он был образован недавно, и на нем еще нет постоянных игроков, занимающих доминирующие позиции. Мощные разветвленные по всему миру каналы дистрибуторов будут настоящим активом на рынке, а дистрибуторы всегда охочи до новых продуктов. Это прекрасная возможность!

Команда разработчиков программного обеспечения HOLIS

В качестве рабочего примера мы выбрали проект разработки системы HOLIS, новой домашней системы автоматического управления освещением (Home Lighting automation System), которую будет распространять компания Lumenations. Команда разработ-

чиков HOLIS имеет типичные размеры. В нашем примере она достаточно небольшая (всего 15 человек), но этого вполне достаточно для того, чтобы все необходимые профессиональные навыки были представлены отдельными членами команды с определенной степенью специализации их ролей. Наиболее важна именно структура команды, и, если добавить дополнительных разработчиков и тестологов, структура команды HOLIS окажется вполне пригодной для разработки пропорционально более объемных программных приложений.

Для работы на новом рынке сбыта компания Lumenations учредила новое подразделение, отдел автоматического управления домашним освещением. Поскольку отдел и технология являются достаточно новыми для компании Lumenations, команда HOLIS, в основном, образована из новых сотрудников, хотя некоторые члены команды перешли в нее из отдела коммерческого освещения. На рис. А.1 представлена организационная схема команды разработчиков и взаимосвязи между ее членами.

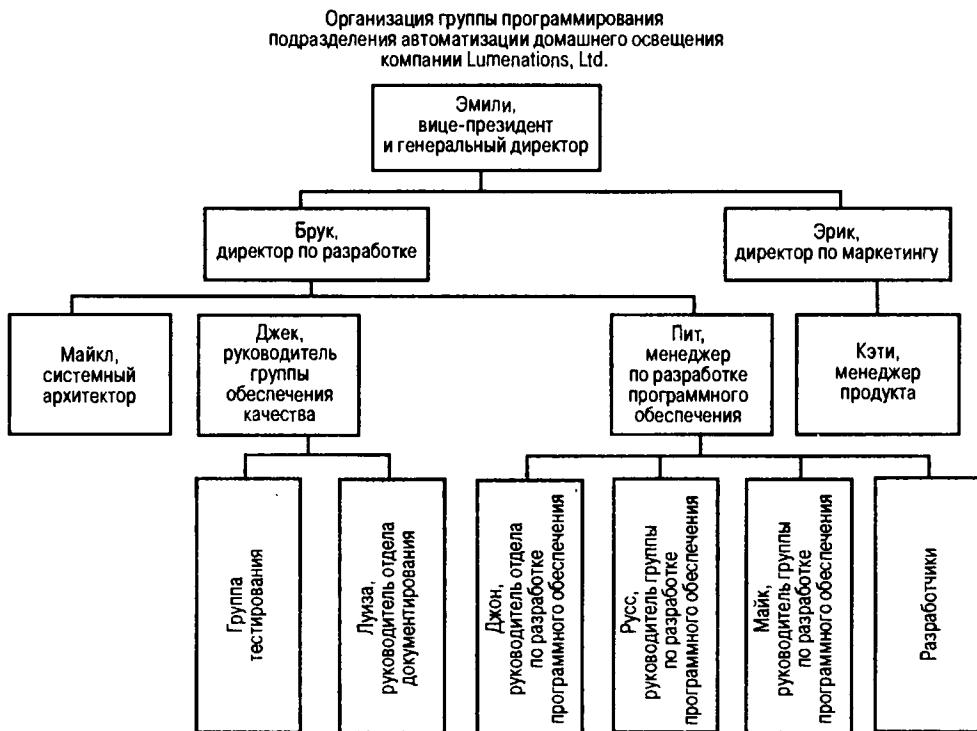


Рис. А.1. Организационная схема команды разработчиков программного обеспечения HOLIS

Набор приемов 1. Анализ проблемы

Постановка существующей в компании Lumenations проблемы

Команда приняла решение разработать три определения проблемы; первое представляет собой формулировку существующей проблемы с точки зрения компании.

Постановка проблемы с точки зрения компании Lumenations

Проблема	замедления роста в основной сфере деятельности компании на рынке профессионального театра
воздействует на	компанию, ее служащих и акционеров,
результатом чего	является неприемлемая эффективность и недостаток возможностей для роста доходов и прибыльности.
Выигрыши от	новых продуктов и потенциально новых рынков для продуктов и услуг компании может состоять в следующем. <ul style="list-style-type: none"> ■ Оживление компании и ее служащих. ■ Возрастание лояльности и сохранение дистрибуторов компании. ■ Более быстрый рост доходов и прибыльности. ■ Подъем цены акций компании.

Команда также решила посмотреть, действительно ли она может понять проблему с точки зрения будущих клиентов (конечных потребителей) и потенциальных дистрибуторов/строителей (клиентов компании Lumenations). Ниже приводится, к чему пришла команда.

Постановка проблемы с позиции домовладельца

Проблема	недостаточного выбора продуктов, ограниченности функциональных возможностей и высокой цены существующих автоматических систем домашнего освещения
воздействует на	владельцев систем для жилья высшего класса,
результатом чего	является неприемлемое функционирование приобретенных систем или, что случается чаще всего, решение "вообще не автоматизировать".
Выигрыши от	"правильного" решения по автоматизации может состоять в следующем. <ul style="list-style-type: none"> ■ Удовлетворение домовладельца от обладания системой. ■ Возрастание удобства использования жилья. ■ Совершенствование безопасности, комфорта и удобства.

Постановка проблемы с позиции дистрибутора

Проблема	недостаточного выбора продуктов, ограниченности функциональных возможностей и высокой цены существующих автоматических систем домашнего освещения
воздействует на	дистрибуторов систем и строителей жилья высшего класса,
результатом чего	является недостаточность возможностей для рыночной дифференциации и отсутствие новых высокоприбыльных продуктов.
Выигрыш от	"правильного" решения по автоматизации может состоять в <ul style="list-style-type: none"> ■ дифференциации; ■ более высоких доходах и прибыли; ■ возросшей доле рынка.

Блок-схема системы с указанием акторов

На рис. А.2 представлена блок-схема системы и ее акторы. На рис. А.3–А.5 представлены блок-схемы подсистем.

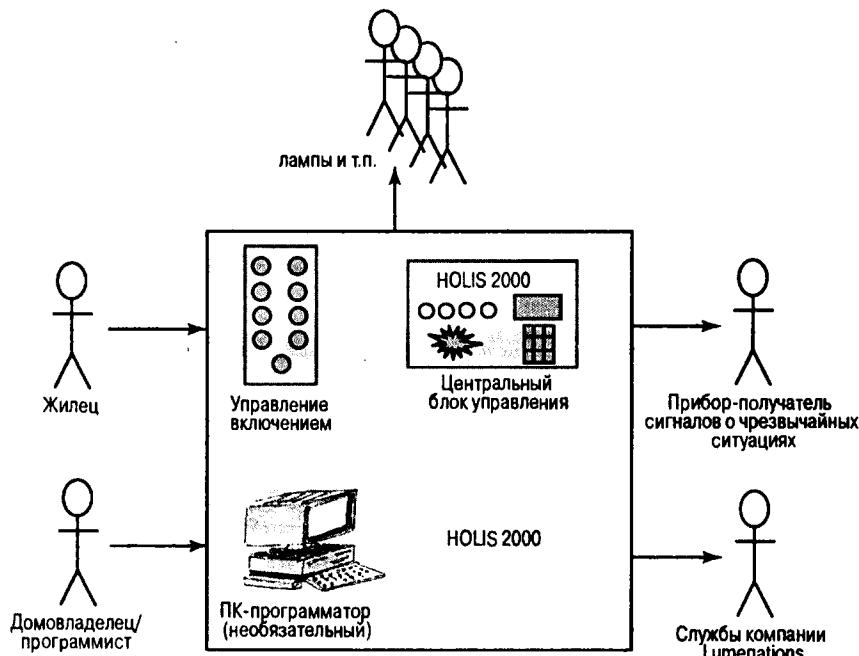


Рис. А.2. Система HOLIS и ее акторы

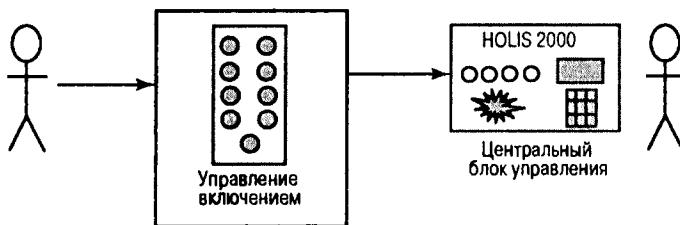


Рис. А.3. Блок-схема и акторы подсистемы “Управление включением”

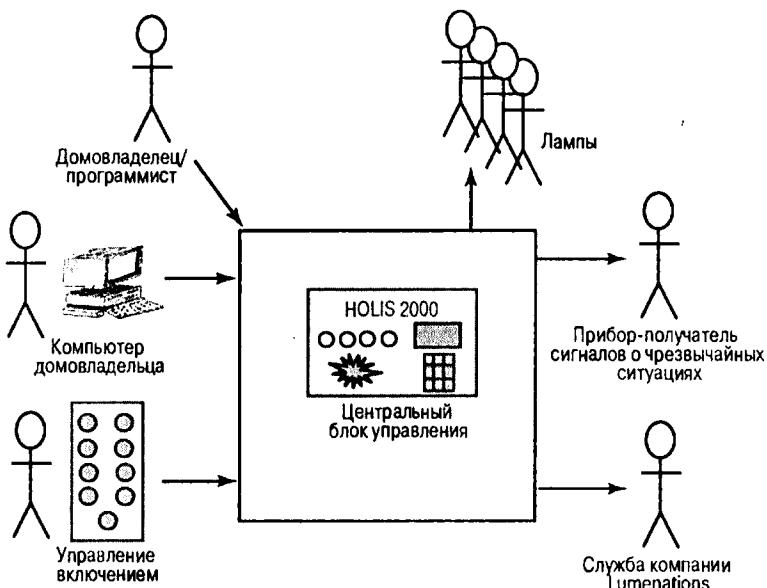


Рис. А.4. Акторы подсистемы “Центральный блок управления”

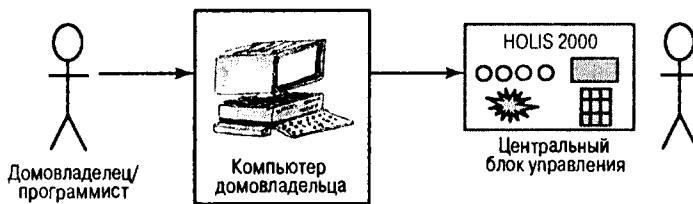


Рис. А.5. Подсистема “ПК-программатор” и ее акторы

Описание акторов

Название	Комментарии
Лампы и т.п.	Устройства вывода, лампы, контроллеры яркости и т.д.
Домовладелец/программист	Программы, написанные домовладельцем непосредственно для ЦБУ или посредством ПК-программатора

Название	Комментарии
Получатель сигнала о нестандартных ситуациях	Находится в стадии исследования
Жилец	Домовладелец, использующий "Управление включением" (пульт) для изменения освещения
Службы компании Lumenations	Служащие компании Lumenations, осуществляющие поддержку дистанционного программирования и действия по сопровождению

Описание других заинтересованных лиц

У системы HOLIS есть много других (не являющихся акторами) заинтересованных лиц – как внешних, так и внутренних.

Название	Комментарии
<i>Внешние</i>	
Дистрибуторы	Непосредственные клиенты компании Lumenations
Строители	Клиенты клиентов компании Lumenations; основной подрядчик, отвечающий перед домовладельцем за конечный результат
Подрядчики электрики	Отвечают за установку и сопровождение
<i>Внутренние</i>	
Команда разработчиков	Команда компании Lumenations
Управление продукцией/сбытом	Будет представлено Кэти, менеджером продукта
Руководство компании Lumenations	Занимается ведением отчетов о финансировании и выпуске продукции

Налагаемые на решение ограничения

По прошествии 15 дней с начала работы над продуктом команда разработчиков системы HOLIS и руководство компании Lumenations выявили, обсудили и согласовали следующие ограничения.

№#	Описание	Пояснения
1	Версия 1.0 должна быть запущена в производство до 5 января 2000 года	Единственная возможность запустить продукт в этом году
2	Команда должна использовать UML-моделирование, ОО-методологии и унифицированный процесс разработки ПО (Unified Software Development Process)	Мы надеемся, что эти технологии обеспечат более высокую производительность и робастность

№#	Описание	Пояснения
3	Программное обеспечение для ЦБУ и "ПК-программатора" будет написано на языке С++. Для "Управления включением" будет использован ассемблер	Для целостности и удобства сопровождения; помимо прочего, команда знает эти языки
4	Система-прототип должна быть показана на декабрьской торговой выставке средств домашней автоматизации	Чтобы получить заказы дистрибуторов на 1-й квартал 2000-го финансового года
5	Микропроцессорная подсистема для ЦБУ будет скопирована с усовершенствованного проекта системы освещения (УПСО), разработанного подразделением профессионального освещения	Готовый проект и материальная часть
6	Единственная поддерживаемая конфигурация ПК-программатора домовладельца должна быть совместима с Windows 98	Сокращение масштаба версии 1.0
7	Команде разрешается взять двух новых сотрудников на условиях полной занятости после успешного испытательного срока; при условии, что они обладают необходимыми навыками	Максимально разрешенное увеличение бюджета
8	В "Управлении включением" будет использоваться моноцикристаллический микропроцессор KCH5444	Уже используется в компании
9	Разрешено использовать закупаемые компоненты программного обеспечения, если это не накладывает на компанию долгосрочных обязательств по выплате роялти	Не должно быть долгосрочного влияния на программное обеспечение затрат на приобретенные товары

Набор приемов 2. Понимание потребностей пользователя

Краткий обзор потребностей пользователей, выявленных с помощью интервью

Команда провела интервью с тремя домовладельцами, двумя дистрибуторами и подрядчиком-электриком.

С точки зрения домовладельца необходимо следующее.

- Гибкое и модифицируемое управление освещением всего здания
- "Защищенность от будущего" ("Поскольку технология меняется, мне бы хотелось обеспечить совместимость с технологиями, которые могут появиться.")
- Привлекательность, ненавязчивость, эргономичность

- Полностью независимое и программируемое (реконфигурируемое) включение освещения в каждой комнате здания
- Дополнительная безопасность и отсутствие головной боли у владельца
- Интуитивно понятная организация работы ("Я бы хотел иметь возможность объяснить, как она работает, моей маме, страдающей техофобией.")
- Разумная стоимость системы и низкая стоимость переналадки
- Простой и недорогой ремонт
- Гибкие конфигурации схем включения (от одной до семи "кнопок" в схеме)
- С глаз долой — из сердца вон (не требует постоянного наблюдения)
- 100%-надежность
- Возможность установки безопасного режима на время длительного отсутствия
- Возможность создания декораций, например специальный режим освещения всего дома во время вечеринки
- Система не должна повышать уровень риска пожара или поражения электрическим током в доме
- Возможность после аварийного отключения электроэнергии восстановить освещение так, как было до того
- Возможность программировать систему самостоятельно, используя свой ПК
- Возможность установки переключателей, допускающих изменение яркости, там, где я пожелаю
- Возможность программировать систему самому, не используя ПК
- Пусть кто-либо другой программирует ее для меня
- Если система выйдет из строя, я хочу иметь возможность включать кое-где свет
- Интерфейсы для моей домашней системы безопасности
- Интерфейсы с другими домашними автоматами (системой вентиляции, аудио/видео и т.д.)

С точки зрения дистрибуторов необходимо следующее.

- Предложение конкурентоспособного продукта
- Наличие некой выделяющей его характеристики
- Простота обучения продавцов
- Возможность демонстрировать в моем магазине
- Высокая разность между себестоимостью и ценой продажи

Совещание по вопросу требований к системе HOLIS 2000

В то время как процесс интервьюирования подходил к концу, команда разработчиков совместно с представителями маркетинга приняла решение созвать совещание по вопросу требований к проекту HOLIS 2000. В нем приняли участие следующие представители команды и заинтересованные лица.

Имя	Роль	Должность	Комментарии
Эрик	Ведущий	Директор по маркетингу	
Кэти	Участник	Менеджер продукта HOLIS 2000	Сторонник проекта
Пит	Участник	Менеджер разработки программного обеспечения	Ответственный за разработку HOLIS 2000
Дженифер	Участник		Будущий домовладелец
Элмер	Участник		Будущий домовладелец
Жене	Участник		Будущий домовладелец
Джон	Участник	Исполнительный директор компаний Automation Equip	Крупнейший дистрибутор компаний Lumenations
Ракель	Участник	Генеральный менеджер, EuroControls	Европейский дистрибутор компаний Lumenations
Бетти	Участник	Президент, Krystel Electric	Местный поставщик электрических систем
Дэвид	Участник	Президент, Rosewind Construction	Домостроитель
Члены команды разработчиков	Наблюдатели	Команда разработчиков	Все члены команды, кто может прибыть

Совещание

Перед совещанием команда подготовила следующий пакет предварительных материалов.

- Несколько свежих журнальных статей, высвечивающих тенденции в домашней автоматизации
 - Копии некоторых проведенных интервью
 - Итоговый список потребностей, которые были выявлены к этому времени
- Эрик “освежил” в памяти приемы ведущего, а Кэти занялась вопросами логистики.

Заседание

Заседание проводилось в отеле недалеко от аэропорта и началось в 8.00. На рис. А.6 показано, как располагались участники и наблюдатели.

Эрик огласил распорядок и правила проведения совещания, а также правила использования специальных “совещательных билетов”. В целом, совещание проходило весьма успешно, и все участники получили возможность высказать свое мнение так, чтобы оно было услышано. Эрик хорошо справился с ролью ведущего, за исключением одного момента, когда он вступил в спор с Кэти о приоритетах групп функций. (Команда приняла решение, что если в будущем вновь придется проводить совещание, нужно будет пригласить независимого ведущего.) Эрик провел сеанс мозгового штурма потенциальных функций системы HOLIS, и команда воспользовалась накопительным голосованием,

чтобы определить относительные приоритеты. Ниже представлены упорядоченные по приоритету функции.

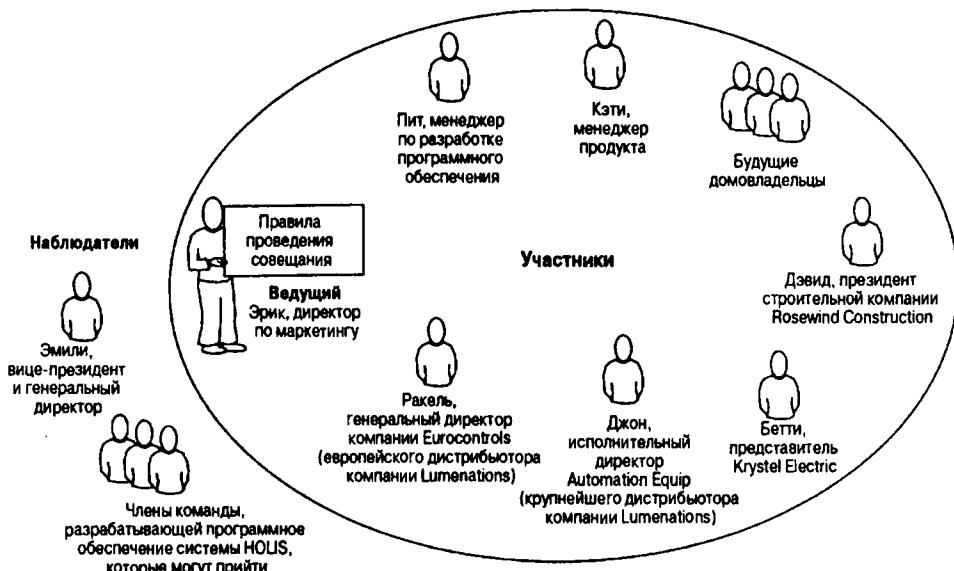


Рис. А.6. Структура совещания по вопросу требований к системе HOLIS 2000

ID	Функции	Количество голосов
23	Возможность произвольного выбора зон освещения	121
16	Автоматическая установка длительности работы для различных источников света и т.п.	107
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105
6	100%-надежность	90
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88
1	Легко программируемые станции управления	77
5	Возможность программирования режима "жильцы в отпуске"	77
13	Любой источник света может плавно изменять яркость	74
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73
14	Возможность программирования работы в режиме обслуживания зданий мероприятий	66
20	Функция закрытия гаражных ворот	66
19	Автоматическое включение света в туалете при открытии двери	55

ID	Функции	Количество голосов
3	Интерфейс с системой охраны дома	52
2	Простота монтажа	50
18	Автоматическое включение света, когда кто-то подходит к двери	50
7	Мгновенное включение/выключение света	44
11	Возможность управлять шторами, жалюзи, насосами и движками	44
15	Управление освещением и т.п. по телефону	44
10	Наличие интерфейса с системой управления автоматикой в доме	43
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34
26	Наличие главного блока управления	31
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25
25	Интернационализированный пользовательский интерфейс	24
21	Интерфейс с видео- и аудиосистемой	23
24	Восстановление функций после сбоя в энергоснабжении	23
17	Управление работой кондиционера	22
28	Активация голосом	7
27	Поддержка презентационного веб-сайта	4

Анализ результатов

Результаты процесса совпали с ожидаемыми, за исключением двух важных моментов.

1. Функция 4, "Встроенные средства системы безопасности", оказалась очень высоко в списке приоритетов. Эта функция упоминалась в предварительных интервью, но не находилась в верхней части чьего-либо списка приоритетов. Кэти выяснила, что встроенная безопасность (возможность гасить свет, дополнительная сирена, вызов внешних служб при возникновении опасности) практически не предлагается конкурирующими системами. Дистрибуторы отметили, что они были удивлены таким предложением, но считают, что это будет выгодным отличием, и согласны с тем, что данная функция должна быть высокоприоритетной. Дэвид также согласился. Основываясь на этом, отдел маркетинга решил включить данную функциональную возможность в продукт и выделить ее как уникальное дифференцирующее отличие на рынке. Она стала одной из определяющих функций системы HOLIS.

2. Функция 25, "Интернационализированный пользовательский интерфейс", не набрала большого количества голосов. (Этого следовало ожидать, исходя из состава участников, так как домовладельцев в США мало заботит то, насколько хорошо продукт будет продаваться в Европе.) Дистрибутор, однако, настаивал, что если продукт не будет интернационализирован уже в версии 1.0, он не будет предлагать-

ся в Европе. Команда зафиксировала это и согласилась приложить все необходимые усилия, чтобы добиться интернационализации в версии 1.0.¹

Краткое описание модели прецедентов системного уровня системы HOLIS

Название	Описание	Акторы
Создание обычной осветительной сцены	Жилец создает обычную осветительную сцену	Жилец, лампы
Инициирование чрезвычайных действий	Жилец инициирует чрезвычайные действия	Жилец
Управление освещением	Жилец включает/выключает лампу(ы) или задает желаемый уровень яркости света	Жилец, лампы
Переключение программы	Изменение или задание действий для определенной клавиши/переключателя	Домовладелец/программист
Удаленное программирование	Провайдер услуг компании Lumenations осуществляет удаленное программирование, основываясь на запросах жильца	Службы компании Lumenations
Продолжительное отсутствие	Домовладелец устанавливает специальный режим на время длительного отсутствия	Домовладелец/программист
Задание временной последовательности	Домовладелец программирует основанную на времени последовательность автоматического возникновения осветительных событий	Домовладелец/программист

Замечание. Остальные прецеденты не приведены из соображений краткости; всего для версии 1.0 было определено 20 прецедентов системного уровня.

Набор приемов 3. Определение системы

Организация требований к системе HOLIS

На рис. А.7 представлена схема организации требований к системе HOLIS.

¹ Этот пример демонстрирует одну из проблем, связанных с накопительным голосованием. Не все участники равны. Неудача в достижении интернационализации, которая не рассматривалась командой перед совещанием, была бы существенной стратегической ошибкой требований.

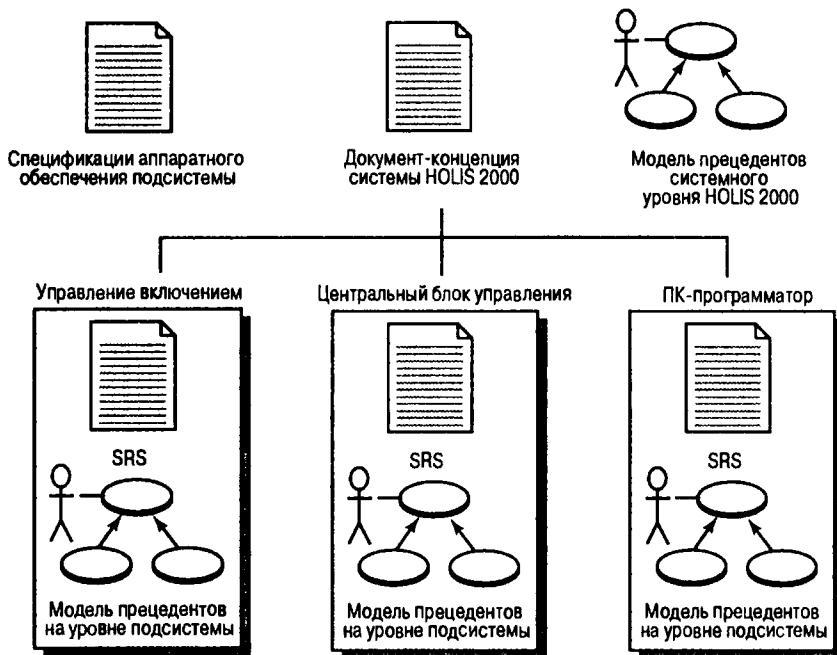


Рис. A.7. Организация требований к системе HOLIS

Документ-концепция (Vision Document) системы HOLIS

Для краткости здесь представлена сокращенная версия документа-концепции системы HOLIS; некоторые разделы опущены. Полный образец документа представлен в приложении Б

Lumenations, Ltd.

Документ- концепция HOLIS 2000

© 1999 Lumenations, Ltd. 102872
Cambridge Ave.
Marcy, NJ 12345

История изменений

Дата	Версия	Описание	Автор
1/21/99	1.0	Исходная версия	Кэти Моул
2/11/99	1.1	Изменения, внесенные после посвященного требованием совещания	Кэти Моул

Содержание

1. Введение

1.1. Цель документа-концепции

Этот документ описывает текущее состояние концепции автоматической системы домашнего освещения HOLIS 2000.

1.2. Характеристика продукта

1.3. Ссылки

- Спецификация требований к программному обеспечению блока управления системы HOLIS 2000
- Спецификация требований к программному обеспечению пульта системы HOLIS 2000
- Спецификация требований к программному обеспечению ПК-программатора системы HOLIS 2000
- Стандарты безопасности и надежности для систем безопасности жилища, Overwriters Laboratory 345.22, 1999

2. Описание пользователя

2.1. Демографические данные о пользователе/рынке

2.2. Род занятий пользователей

2.3. Среда пользователя

2.4. Основные потребности пользователей

Следующие потребности пользователей были выявлены отделом маркетинга в ходе проведения ряда интервью с будущими домовладельцами и дистрибуторами в 1998 году. Эти интервью находятся в файле локальной корпоративной сети по адресу: www.HOLISHomepage.com/marketing/HOLIS/interviews.

2.4.1. С точки зрения домовладельца

- Гибкое и модифицируемое управление освещением всего здания
- "Защищенность от будущего" ("Поскольку технология меняется, мне бы хотелось совместимости с технологиями, которые могут появиться.")
- Привлекательность, ненавязчивость, эргономичность
- Дополнительная безопасность и отсутствие головной боли у владельца

- Интуитивно понятная организация работы (“Я бы хотел иметь возможность объяснить, как она работает, моей маме, страдающей технофобией.”)
- Разумная стоимость системы и низкая стоимость переналадки
- Простой и недорогой ремонт
- Гибкие конфигурации схем включения (от одной до семи “кнопок” в схеме)
- С глаз долой – из сердца вон (не требует постоянного наблюдения)
- 100%-надежность
- Не повышает уровень риска пожара или поражения электрическим током в доме
- Возможность после аварийного отключения электроэнергии восстановить освещение так, как было до того
- Возможность легко модифицировать функции пульта
- Возможность программировать систему самостоятельно, используя свой ПК
- Возможность самостоятельно программировать систему, не используя ПК
- Пусть кто-либо другой программирует ее для меня
- Если система выйдет из строя, я хочу иметь возможность включать кое-где свет
- Интерфейсы с моей домашней системой безопасности
- Интерфейсы с другими домашними автоматами (системой вентиляции, аудио- и видеосистемой и т.д.)

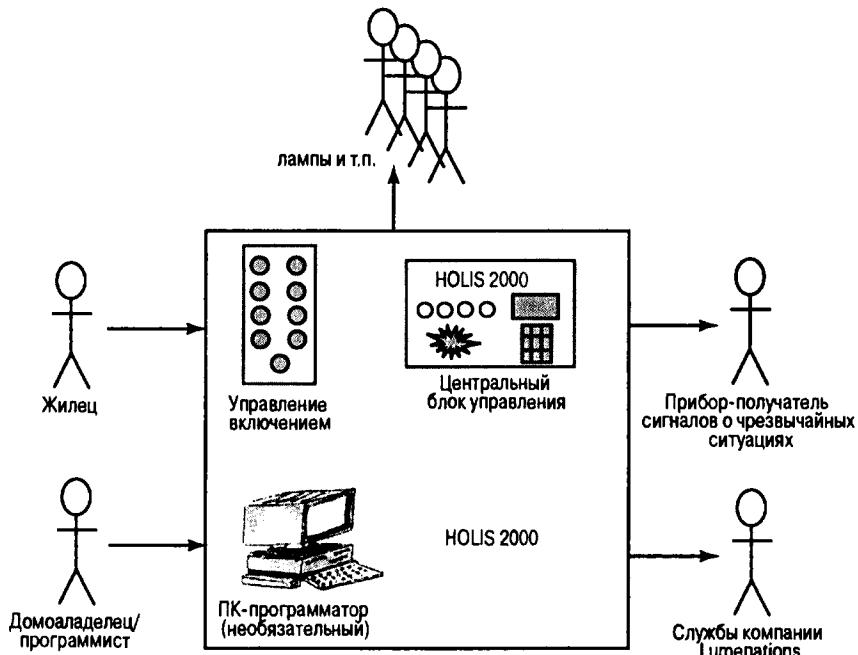
2.4.2. С точки зрения дистрибуторов

- Предложение конкурентоспособного продукта
- Наличие некой выделяющей его характеристики
- Простота обучения продавцов
- Возможность демонстрировать в моем магазине
- Высокая разность между себестоимостью и продажной ценой

2.5. Альтернативы и конкуренты

3. Характеристика продукта

3.1. Общий вид продукта (система и ее окружение)



(Текст не приводится из соображений краткости.)

3.2. Определение назначения продукта HOLIS 2000

- | | |
|--------------|---|
| Для | домовладельцев, строящих новые дома высокого класса, |
| которые | хотят повысить уровень комфорта, удобства и безопасности своего жилья, |
| HOLIS 2000 | представляет собой автоматизированную систему управления освещением жилища, |
| которая | предлагает беспрецедентные, простые в использовании современные возможности по автоматизации освещения по разумной цене. |
| В отличие от | серии систем Lightomation Systems компании Herb's Industrial Controls, |
| наш продукт | сочетает в себе последние достижения в сфере возможностей автоматизации домашнего освещения со встроенными функциями безопасности, а его установка и сопровождение обходятся дешевле. |

- 3.3. Краткий обзор возможностей
- 3.4. Предположения и зависимости
- 3.5. Себестоимость и цены
- 3.6. Лицензирование и инсталляция

4. Атрибуты функций

- 4.1. Приоритет
Носит обязательный (а не рекомендательный) характер.
- 4.2. Статус
- 4.3. Количество голосов

Количество голосов отражает приоритет, определенный на совещании, посвященном требованиям к системе HOLIS 2000.

4.4. Трудоемкость

Низкая, средняя и высокая; задается командой разработчиков.

4.5. Риск

Задается командой разработчиков.

4.6. Стабильность

4.7. Целевая версия

4.8. Предназначен для...

4.9. Причина

5. Функции продукта

5.1. Обязательные функции для версии 1.0

- **Fea23. Возможность произвольного задания сценариев освещения.** Система позволяет домовладельцу самостоятельно задавать сценарии освещения. Каждый сценарий обеспечивает предварительно заданный уровень яркости для каждого набора ламп в доме. Сценарии могут инициироваться с помощью пульта ("Управление включением") или "Центрального блока управления".
- **Fea16. Автоматическая установка длительности работы для различных источников света.** Домовладелец может создавать заранее определенные, основанные на времени, графики проведения различных манипуляций с освещением.
- **Fea4. Встроенная безопасность.** Система имеет встроенную функцию безопасности, которая состоит из одной кнопки, позволяющей инициировать последовательность подачи сигнала тревоги от каждого пульта в доме. Эта последовательность обеспечивает включение света по заранее определенному сценарию, а также (необязательно в каждом случае) выключение света, включение сигнала тревоги и звонок по заранее определенному номеру с предварительно запрограммированным голосовым сообщением. Система также замыкает контакты, которые домовладелец может использовать для подключения различных устройств по своему усмотрению.
- **Fea6. Надежность.** Домовладельцы неоднократно подчеркивали, что система должна иметь практически 100%-надежность. Это особенно важно для последовательности действий по обеспечению безопасности.
(Остальные функции не включены из соображений краткости.)

5.2. Дополнительные функции

- **Fea20. Функция управления дверью гаража.** Система воспринимает "дверь гаража" как одно из контролируемых устройств вывода. Программа должна осуществлять соответствующий контроль вывода, а также должна обеспечивать изображение/пиктограмму двери гаража и поддерживать программирование данной функции.
- **Fea2. Простота монтажа.** Простота монтажа особенно важна для дистрибуторов/заказчиков компании и будет главной отличительной особенностью продукта. Программное обеспечение должно способствовать осуществлению этой задачи всеми возможными средствами. Это могут быть ин-

терактивные подсказки к руководству по монтажу, руководство по исправлению ошибок, индикация оценки состояния процесса, автоматическое обнаружение ошибок и т.д.

(*Замечание для Брука.* Группа инженеров должна исследовать эту потребность и представить отделу маркетинга список идей и приблизительную оценку стоимостных параметров, чтобы определить, как далеко можно продвинуться в этом направлении в версии 1.0.)

(Остальные дополнительные функции не приводятся из соображений краткости.)

5.3. Будущие функции

В приложении А данного документа-концепции перечисляются функции, которые были выявлены для возможных будущих версий системы. Несмотря на то что при работе над версией 1.0 не стоит тратить на это значительные средства, мы все-таки попросили группу инженеров и отдел маркетинга ознакомиться с этим списком и по возможности учитывать эти потребности в ходе работы над проектом и разработкой версии 1.0.

6. Важные precedents

(Текст для краткости не приводится.)

7. Прочие требования к продукту

- 7.1. Применимые стандарты
- 7.2. Системные требования
- 7.3. Лицензирование и инсталляция
- 7.4. Требования эффективности

8. Требования к документации

- 8.1. Руководство пользователя
- 8.2. Интерактивная подсказка
- 8.3. Руководства по монтажу, схема, файл Read Me
- 8.4. Маркировка и упаковка

9. Глоссарий

Приложение А. Будущие функции, выявленные совещанием по вопросу требований

Приложение В. Представленные участникам совещания раскладовки

Приложение С. Основные precedents

Набор приемов 4. Управление масштабом

После проведения совещания перед командой возникла задача оценить уровень трудозатрат для каждой функции и попытаться определить базовый уровень версии 1.0. Нужно было принять жесткие меры по сокращению масштаба в связи с имеющимися ограничениями, среди которых – необходимость представить прототип для показа на тор-

говой выставке в декабре, а также запуск системы в производство в январе², что еще сильнее лимитировало возможности. Для задания предполагаемого уровня трудозатрат команда воспользовалась эвристическим методом оценки по принципу высокий-средний-низкий, а затем добавила оценки риска для каждой функции. Результаты этой работы, а также дальнейших действий по ограничению масштаба отражены в табл. А.1 и А.2.

Таблица А.1. Упорядоченные по числу набранных голосов функции системы HOLIS 2000 с атрибутами риска и трудоемкости

ID	Функция	Число голосов	Трудоемкость	Риск
23	Возможность произвольного задания сценариев освещения	121	Средняя	Низкий
16	Автоматическая установка длительности работы для различных источников света и т.п.	107	Низкая	Низкий
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105	Низкая	Высокий
6	100%-надежность	90	Высокая	Высокий
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88	Высокая	Средний
1	Легко программируемые станции управления	77	Средняя	Средний
5	Возможность программирования режима "жильцы в отпуске"	77	Низкая	Средний
13	Любой источник света может плавно понижать мощность	74	Низкая	Низкий
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73	Высокая	Средний
14	Возможность программирования работы в режиме обслуживания зрелищных мероприятий	66	Низкая	Низкий
20	Функция закрытия гаражных ворот	66	Низкая	Низкий
19	Автоматическое включение света в тумане при открытии двери	55	Низкая	Высокий

² Относительно этого срока команда приняла решение, что в действительности окончательный вариант версии 1.0 программного обеспечения будет представлен до конца февраля. Это критически важные 6 недель, которые, по мнению команды, необходимы, чтобы, основываясь на полученных после показа на выставке отзывах, внести окончательные изменения.

Продолжение табл. А.1

ID	Функция	Число голосов	Трудоемкость	Риск
3	Интерфейс с системой охраны дома	52	Высокая	Высокий
2	Простота монтажа	50	Средняя	Средний
18	Автоматическое включение света, когда кто-то подходит к двери	50	Средняя	Средний
7	Мгновенное включение/выключение света	44	Высокая	Высокий
11	Возможность управлять занавесками, насосами и движками	44	Низкая	Низкий
15	Управление освещением и т.п. по телефону	44	Высокая	Высокий
10	Наличие интерфейса с системой управления автоматикой в доме	43	Высокая	Высокий
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34	Средняя	Низкий
26	Наличие центральных станций управления	31	Высокая	Высокий
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25	Средняя	Средний
25	Интернационализированный пользовательский интерфейс	24	Средняя	Высокий
21	Интерфейс с видео- и аудиосистемой	23	Высокая	Высокий
24	Восстановление функций после сбоя в энергоснабжении	23	Нет данных	Нет данных
17	Управление системой кондиционирования воздуха	22	Высокая	Высокий
28	Активация голосом	7	Высокая	Высокий
27	Поддержка презентационного веб-сайта	4	Средняя	Низкий

Таблица А.2. Базовый уровень версии 1.0 системы HOLIS 2000

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
23	Возможность произвольного задания сценариев освещения	121	Средняя	Низкий	Максимально возможная гибкость
16	Автоматическая установка длительности работы для различных источников света и т.п.	107	Низкая	Низкий	Максимально возможная гибкость

Продолжение табл. А.2

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
4	Встроенные средства системы безопасности, например аварийные лампы, звуковые сирены, звонки	105	Низкая	Высокий	Необходимы дальнейшие маркетинговые исследования
6	100%-надежность	90	Высокая	Высокий	Подойти к 100% максимально близко
8	Легко программируемый блок управления, не требующий использования персонального компьютера	88	Высокая	Средний	Обеспечить соответствующий контроллер
1	Легко программируемые станции управления	77	Средняя	Средний	Сделать простым, насколько этого можно добиться при соответствующей трудоемкости
5	Возможность программирования режима "жильцы в отпуске"	77	Низкая	Средний	
13	Любой источник света может плавно понижать мощность	74	Низкая	Низкий	
9	Можно использовать собственный персональный компьютер для программирования режимов работы	73	Высокая	Средний	В версии 1.0 поддерживается только одна конфигурация
25	Интернационализированный пользовательский интерфейс ЦБУ	24	Средняя	Высокий	По настоящию Европейского дистрибутора
14	Возможность программирования работы в режиме обслуживания зрелищных мероприятий	66	Низкая	Низкий	(Не применяется, включена в 23)
7	Мгновенное включение/выключение света	44	Высокая	Высокий	Вложить разумные средства
<i>V 1.0. Обязательный базовый уровень. Все, что находится выше данной линии, должно быть реализовано, иначе выпуск версии будет задержан.</i>					
20	Функция закрытия гаражных ворот	66	Низкая	Низкий	Может незначительно повлиять на программное обеспечение

Продолжение табл. А.2

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
2	Простота монтажа	50	Средняя	Средний	Базис для тру-доемкости
11	Возможность управлять шторами, жалюзи, насосами и движками	44	Низкая	Низкий	Может незначительно по-влиять на про-граммное обес-пече-ние
22	Наличие режима плавного перехода: постепенное увеличение/уменьшение яркости света	34	Средняя	Низкий	Хорошо, если получится сде-лать это

V 1.0. Дополнительные функции. Постарайтесь разработать те из них, которые сможете (Кэти).

Будущие функции. Находящиеся ниже этой линии функции для данной версии не разрабатываются.

29	Интернационализиро-ванный интерфейс ПК-программатора	24	Средняя	Высокий	Будет обяза-телен для версии 2.0
3	Интерфейс с системой охраны дома	52	Высокая	Высокий	Можем ли мы предложить хотя бы аппар-атный интерфейс? (Эрик)
19	Автоматическое включение света в туалете при открытии двери	55	Низкая	Высокий	
18	Автоматическое включение света, когда кто-то подходит к двери	50	Средняя	Средний	
15	Управление освещением и т.п. по телефону	44	Высокая	Высокий	
10	Наличие интерфейса с системой управления автоматаикой в доме	43	Высокая	Высокий	
26	Центральные станции управления	31	Высокая	Высокий	
12	Легко дополняется новыми элементами при изменении схемы эксплуатации	25	Средняя	Средний	
25	Пульты дистанционного управления	24	Средняя	Высокий	

Окончание табл. А.2

ID	Функция	Число голосов	Трудоемкость	Риск	Маркетинговые комментарии
21	Интерфейс с видео- и аудиосистемой	23	Высокая	Высокий	
24	Восстановление функций после сбоя в энергоснабжении	23	Нет данных	Нет данных	
17	Управление системой кондиционирования воздуха	22	Высокая	Высокий	
28	Активация голосом	7	Высокая	Высокий	
27	Поддержка презентационного веб-сайта	4	Средняя	Низкий	

Набор приемов 5. Уточнение определения системы

Образец описания прецедента системы HOLIS: Управление освещением

История изменений

Дата	Версия	Описание	Автор
14.04.99	1.0	Создание исходной версии прецедента Управление освещением	Дон Уидриг
15.04.99	1.1	Добавлено второе предусловие для конкретизации действий	Джек Бигриг, руководитель группы гарантии качества

Краткое описание. Данный прецедент задает способ включения и выключения света, а также изменения его яркости в зависимости от того, как долго пользователь удерживает в нажатом состоянии кнопку пульта.

Основной поток событий. Основной поток начинается, когда Жилец нажимает любую кнопку пульта. Если Жилец отпускает кнопку в период времени, отсчитываемый таймером, система "переключает" состояние освещения.

- Если освещение было включено, оно полностью выключается.
- Если свет был выключен, освещение включается с тем же уровнем яркости, который был задан перед последним выключением.

Конец основного потока.

Альтернативный поток событий. Если Жилема удерживает кнопку пульта в нажатом состоянии дольше 1 секунды, система инициирует действия по изменению яркости для указанной кнопки пульта. Пока Жилема продолжает удерживать кнопку, происходит следующее.

1. Яркость контролируемого источника постепенно повышается до максимального значения со скоростью 10 процентов в секунду.
 2. Когда достигнуто максимальное значение, яркость контролируемого источника постепенно понижается до минимального уровня со скоростью 10 процентов в секунду.
 3. Когда достигнуто минимальное значение, процесс продолжается с шага 1.
- Когда Жилема отпускает кнопку, происходит следующее.
4. Система прекращает изменять яркость освещения.

Предусловия для прецедента Управление освещением. Для выбранной кнопки пульта должен быть предусмотрен режим "Изменение яркости". Выбранная кнопка пульта должна быть предварительно запрограммирована для управления неким набором осветительных приборов.

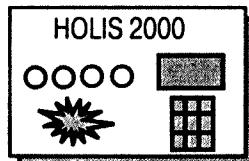
Постусловия для прецедента Управление освещением После окончания данного прецедента запоминается текущий уровень яркости для выбранной кнопки пульта.

Точки расширения. Нет.

HOLIS 2000

Центральный блок управления

Спецификация требований
к программному обеспечению



© 1999 Lumenetions, Ltd

Рис. A.8. Титульная страница пакета Modern SRS Package системы HOLIS

Спецификация требований к программному обеспечению подсистемы “Центральный блок управления” системы HOLIS

История изменений

Дата	Версия	Описание	Автор
11.04.99	1.0	Исходная версия	Джон Алтиибай
15.04.99	1.1	Преобразована в ReqPro	Дон Уидриг
18.04.99	1.2	Исправлена с учетом поддержки плаинов тестирования	Дон Уидриг и Дин Леффингтун- зелл

Содержание

1. Введение

1.1. Цель

Это спецификация требований к программному обеспечению для подсистемы “Центральный блок управления” версии 1.0 системы HOLIS 2000.

1.2. Масштаб

1.3. Ссылки на другие используемые документы

- Документ-концепция HOLIS 2000
- Модель прецедентов системного уровня системы HOLIS 2000: <http://www.Lumenations.com/Engineering/HOLIS/Rose.mdl>
- Спецификация требований к программному обеспечению “Управления включением” (пульта) системы HOLIS 2000
- Спецификация требований к программному обеспечению “ПК-программатора” системы HOLIS 2000

1.4. Предположения и зависимости

2. Описание модели прецедентов

Центральный блок управления системы домашнего освещения участвует во всех прецедентах системного уровня. Кроме того, в качестве ведущей подсистемы, ЦБУ участвует в ряде других прецедентов, ориентированных главным образом на инсталляцию, запуск, мониторинг и взаимодействие с пультом и “ПК-программатором”. Ниже перечислены эти прецеденты подсистемы.

Название	Описание	Акторы
Диагностика системы	Этот прецедент выполняется по команде или инициируется вручную	Управление включением (пульт), домовладелец/программист, службы компании Lumenations, получатель сигналов о чрезвычайных ситуациях, лампы

Название	Описание	Акторы
Калибровка	Калибруются изменяемые уровни яркости	Домовладелец/программист, лампы
Назначение байков освещения	Устанавливается соответствие физических и логических байков осветительных приборов	Домовладелец/программист
Переключение программы	Изменяется последовательность действий для конкретной кнопки/переключателя	Домовладелец/программист

Замечание. Остальные прецеденты не включены из соображений краткости. Всего же для версии 1.0 было определено 11 прецедентов.

3. Система и ее окружение

На рис. А.9 представлена блок-схема ЦБУ в общесистемном контексте.

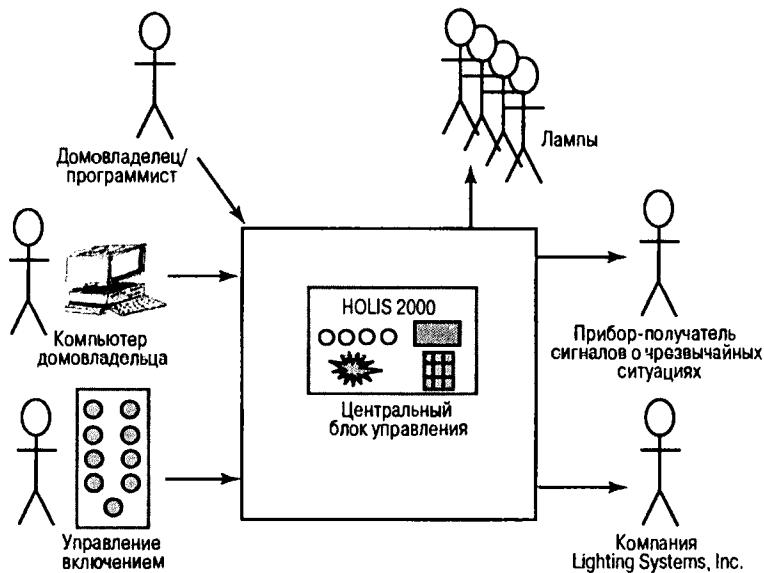


Рис. А.9. Подсистема ЦБУ и ее место в системе

4. Описание акторов

Ниже описаны акторы, взаимодействующие с ЦБУ.

Актор	Описание
Домовладелец/программист	Домовладелец/программист (а также специалист по сопровождению), который взаимодействует с панелью управления

Актор	Описание
Службы компании Lumenations	Актор, соединенный с ЦБУ посредством телефонной линии; осуществляет удаленное программирование и диагностику
Получатель сигналов о чрезвычайных ситуациях	Актор, получающий сообщения о чрезвычайных ситуациях
Лампы	Лампы и другие управляемые выводы, двери гаража и т.п.
"Управление включением" (пульт)	Прибор управления переключением света
"ПК-программатор"	Дополнительно подключенный персональный компьютер домовладельца

Замечание. Актор Жилец не взаимодействует с ЦБУ. Жилец может взаимодействовать с "Управлением включением" (пультом), который в свою очередь взаимодействует с ЦБУ.

5. Требования

5.1. Функциональные требования

- **SR1. Системные часы.** Система должна использовать и поддерживать системные часы. Точность этих часов должна быть такой, как указано в требованиях ALSP.
 - SR1.1. Синхронизация времени часов. Домовладелец должен иметь возможность устанавливать часы, используя числовые клавиши и специальные функциональные кнопки панели оператора ЦБУ. Графический интерфейс пользователя для этих целей должен выглядеть как снимок экрана.
 - SR1.2. Синхронизация месяца. Домовладелец должен также иметь возможность задавать месяц с помощью числовых клавиш и специальных функциональных кнопок панели оператора ЦБУ.
- **SR2. Уровень освещения OnLevel.** Каждый управляемый банк освещения, для которого возможно изменение яркости, должен иметь поле данных. Управление уровнем освещения осуществляется с помощью параметра OnLevel, который задает процент яркости света. Параметр OnLevel может иметь одно из девяти возможных значений: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.
- **SR3. Поддержка до 255 зависящих от времени графиков событий (event-time-schedules).**
 - SR3.1. Допустимая программируемая точность временных графиков событий должна составлять 1 минуту.
 - SR3.2. HOLIS должна выполнять временной график событий с точностью до 1 минуты ± 5 секунд по системным часам.
 - SR3.3. Графики могут программироваться как для 12, так и для 24 часовых форматов. Пользователь должен вводить данные в следующем формате.

SR3.3.1. Номер события (1-256), Время дня (в 24-х часовом формате НН:ММ).

SR3.3.2. Затем для каждого осветительного банка, на который влияет событие, пользователь должен указать следующие данные, чтобы завершить создание графика.

ID осветительного банка	Действие (вкл, выкл, OnLevel) (вводится как % от максимального с 10%-приращениями)
73	Вкл
34	Выкл
73	60%

SR3.3.3. После окончания введения данных пользователь должен нажать клавишу <End>, чтобы сигнализировать, что задание графика окончено.

- **SR4.** Протокол сообщений от “Управления включением” (пульта). Каждое нажатие кнопки пульта инициирует 4-байтовое сообщение ЦБУ. Ниже представлено, как выглядит протокол сообщения.

Адрес устройства Номер сообщения Данные Контрольная сумма отправителя

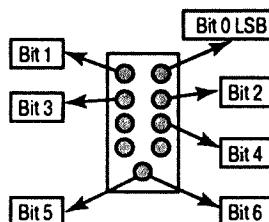
Поля сообщения интерпретируются следующим образом.

SR4.1. Адрес 0-254, логический адрес конкретной кнопки пульта, от которой исходит сообщение.

SR4.2. Номер сообщения 0-255. Осуществляется поддержка следующих номеров сообщений.

1. Нажатие обычной клавиши
2. Чрезвычайная ситуация
3. Была в нажатом состоянии последние 0.5 секунды

SR4.3. Поле данных, где каждый бит соответствует определенной кнопке пульта.



SR4.4. Подтверждение приема сообщения. В ответ на сообщение от пульта, ЦБУ должен послать следующее сообщение.

[55] [FF] Полученные данные Контрольная сумма

Здесь 55(шестнадцатеричное) – это адрес ЦБУ, FF (шестнадцатеричное) – код подтверждающего сообщения, Полученные данные возвращают полученный ЦБУ байт данных, а Контрольная сумма – это вычисленная контрольная сумма возвращаемого сообщения.

(Все остальные требования не приводятся из соображений краткости.)

5.2. Нефункциональные требования

5.2.1. Практичность

5.2.2. Надежность

- Заказчики системы HOLIS требовали, чтобы ее надежность была максимально возможно приближена к 100%.
- SR5. ЦБУ не должен иметь дефектов, которые могут повлиять на нормальное использование жилья домовладельцем.

5.2.3. Эффективность

5.2.4. Возможность сопровождения

6. Требования к интерактивной документации пользователя и системе подсказок

7. Ограничения проектирования

DC1. Проект управляющей подсистемы основывается на модуле контроллера из семейства продуктов ALSP. BIOS не должен модифицироваться без крайней необходимости.

DC2. Прецедент и поддерживающая его инфраструктура для последовательности действий в чрезвычайной ситуации должны быть подвергнуты испытаниям на уровне наивысших коммерческих стандартов.

8. Закупаемые компоненты

9. Интерфейсы

- 9.1. Интерфейсы пользователя
- 9.2. Интерфейсы аппаратуры
- 9.3. Интерфейсы программного обеспечения
- 9.4. Интерфейсы связи

10. Требования лицензирования

Для ЦБУ нет требований лицензирования.

11. Замечания, касающиеся авторских прав, юридических аспектов и т.п.

- SR6. Предусмотреть отображение на экране извещения о корпоративном авторском праве, символики компании и продукта HOLIS 2000 в течение не менее 5 секунд в режиме начала работы.
- SR7. В режиме постоянной работы, когда не производятся действия по программированию, дисплей должен все время отображать логотип HOLIS 2000.

12. Применимые стандарты

SRS-индекс

SRS-глоссарий

SRS-приложение. Спецификации прецедентов подсистемы ЦБУ

Набор приемов 6. Построение правильной системы

HOLIS 2000. Образец тестового примера 01: тест прецедента “Управление освещением”

История изменений

Дата	Версия	Описание	Автор
14.04.99	1.0	Первая версия	Джек Бигриг
15.04.99	1.1	Коррекция для увеличения/уменьшения яркости	Джин Мак-Билл

Описание. Данный тестовый пример, предназначенный для тестирования экземпляров прецедента Управление освещением, используется для тестирования только предварительно приписанных к осветительному банку клавиш пульта, для которых возможен режим изменения яркости.

ID события	Описание события	Ввод 1	Ввод 2	Ожидаемый результат
		<i>Основной поток</i>		
2001	Жильтя нажимает кнопку пульта	Любая допустимая кнопка	Перед нажатием кнопки свет был включен (тестолог должен запомнить уровень)	Свет выключается
2002			Перед нажатием кнопки свет был выключен	Свет включается с прежним уровнем яркости OnLevel
2003	Жильтя отпускает кнопку в течение 1с		Свет выключен	Остается выключенным
2005	Жильтя отпускает кнопку в течение 1с (Этим заканчивается первый путь прецедента.)		Свет включен	Остается включенным с прежним уровнем яркости (OnLevel)
2006	Жильтя нажимает кнопку снова и отпускает в течение 1с	Та же кнопка, что и в 2003	Перед этим свет выключен	Свет включается с тем же уровнем яркости, что и в 2002

ID события	Описание события	Ввод 1	Ввод 2	Ожидаемый результат
	Жилец нажимает кнопку вновь и отпускает в течение 1с		Перед этим свет включен	Свет выключается
2007	Кнопка удерживается в нажатом состоянии дольше 1с	Допустимая кнопка <i>Альтернативный поток</i>	Перед этим свет выключен	Свет включается. Яркость возрастает до максимального уровня со скоростью 10% в секунду, а затем уменьшается со скоростью 10% в секунду до минимального уровня, после чего снова возрастает. Циклы продолжаются, пока нажата кнопка
2008	Жилец отпускает кнопку			Яркость остается на последнем достигнутом уровне

Замечание. Тест выполняется много раз при различных вариантах нажатия кнопки для проверки того, что система правильно запоминает уровень яркости OnLevel.

HOLIS 2000. Образец тестового примера 02: тест протокола обмена сообщениями

История изменений

Дата	Версия	Описание	Автор
14.04.99	1.0	Исходный план	Джин Мак-Билл

Описание. Данный тестовый пример служит для проверки протокола обмена сообщениями между ЦБУ и "Управлением включением" (УВ, пультом). При этом тестируются следующие требования из SRS к ЦБУ и УВ.

SRS ЦБУ	SRS УВ
SR4, SR4.1, SR4.2, SR4.3, SR4.4	CSS88, CSSR91-97, CSSR100-107

(Замечание. Данную таблицу можно удалить после задания матрицы трассировки. Для минимизации усилий по сопровождению подобные связи поддерживаются только в матрице трассировки.)

События

ID события	Описание события	Ввод 1	Ввод 2	Ожидаемый результат
5300	Нажать на пульте 1 кнопку 0 и инициировать сообщение от УВ ЦБУ	Одна кнопка		Горят ЦБУ-индикатор приема сообщения и УВ-индикатор приема сообщения
5301	Проверить полученное сообщение на строке диагностики дисплея ЦБУ			[01][01][01][5A]
5302	Проверить посланное сообщение на дисплее ЦБУ			[55][FF][01][F7]
5303	Нажать кнопки пульта 0-5 одновременно и держать в таком положении 3с Проверить сообщение 1 (Остальные события не приводятся из сообщений краткости)	Все кнопки нажаты 3 и более секунд		ЦБУ-индикатор приема сообщения горит. В буфере сообщений дисплея должно находиться 3 сообщения [01][01][3F][3C]

Приложение Б

Образец документа-концепции

Исключительную важность для успеха проекта имеет документ-концепция, в котором представлены высокоуровневые потребности пользователя и функции приложения. Этот документ по мере необходимости обновляется и доводится до сведения членов команды разработчиков и других участников проекта. Предлагаемый ниже образец документа можно использовать в качестве отправной точки и модифицировать в соответствии с потребностями конкретной организации.

Название компании

Документ-концепция [Название проекта]

© 1999 Название
компании

История исправлений

Дата	Версия	Описание	Автор
23/06/99	1.0	Исходная версия	Имя автора
число/месяц/год			

Содержание

1. Введение

В данном разделе необходимо представить общую характеристику документа-концепции в целом; наряду с этим он должен содержать следующие подразделы.

1.1. Цель документа-концепции

Цель данного документа состоит в сборе, анализе и определении высокоуровневых потребностей пользователей и функций продукта. Основное внимание уделяется возможностям, в которых нуждаются будущие пользователи, и причинам существования этих потребностей. Конкретные требования, касающиеся того, как приложение выполняет эти потребности, должны быть представлены в *спецификациях требований к программному обеспечению* или *спецификациях прецедентов*.

1.2. Общая характеристика продукта

В данном разделе определяется цель приложения, его версия и новые предоставляемые функции. Здесь следует

- указать продукт или приложение, которое создается или изменяется;
- дать общее описание того, что продукт будет делать и, если необходимо, чего не будет делать;
- описать применение продукта, в том числе достижимые с его помощью выгоды, цели и задачи.

1.3. Ссылки

Этот подраздел содержит следующее.

- Список всех документов, упоминаемых где-либо в документе-концепции
Для каждого из этих документов указывается полное его название, номер (если нужно), дата публикации, а также название опубликовавшей его организации.
- Список источников, к которым можно обратиться за справками
Эта информация может быть представлена ссылкой на приложение или другой документ.

2. Описание пользователя

Чтобы преуспеть в предоставлении продуктов и услуг, удовлетворяющих потребности заказчиков, необходимо знать, с какими проблемами сталкиваются пользователи при выполнении своей работы. Данный раздел должен содержать описание профиля потенциальных пользователей приложения и основных проблем, ограничивающих их производительность. Этот раздел не следует использовать для формулировки конкретных требований. В нем должны содержаться обоснования того, почему необходимы перечисленные в разделе 5 требования.

2.1. Характеристика рынка/пользователя

Здесь необходимо кратко перечислить основные характеристики рынка, которые послужили мотивацией решений, касающихся продукта: описать и указать целевые сегменты, а также оценить объем и перспективы роста рынка, ориентируясь на число потенциальных пользователей или количество денег, которые в настоящее время тратят ваши заказчики, пытаясь решить те задачи, что будут решаться с помощью вашего приложения (или усовершенствования). Нужно также рассмотреть основные существующие в отрасли тенденции и технологии. При этом следует ответить на следующие стратегические вопросы: *какова репутация вашей организации на этих рынках и как данный продукт или услуги помогают достижению вашей цели.*

2.2. Описания пользователей

Здесь следует описать все типы пользователей. Пользователи могут сильно отличаться по своему уровню: от новичков до искушенных профессионалов. Опытному пользователю может потребоваться сложное гибкое средство поддержки межплатформенного взаимодействия, в то время как новичку нужно простое в обращении дружественное пользователю средство. Описание профиля должно для каждого типа пользователей освещать следующие вопросы.

- Технический уровень и опыт
- Основные обязанности

- Что делает пользователь и для кого
- Тенденции, упрощающие или усложняющие работу пользователя
- Проблемы, от которых зависит успех
- В чем пользователь видит успех и как пользователь вознаграждается

2.3. Среда пользователя

Подробное описание рабочей среды целевого пользователя. Ниже представлены некоторые вопросы, которые желательно осветить.

- Сколько человек участвует в выполнении данной задачи? Изменится ли их число?
- Сколько времени длится цикл выполнения задачи? Сколько времени отводится на выполнение каждого действия? Изменится ли это?
- Существуют ли некие уникальные ограничения среды: на мобильную связь, по работе вне помещения, в полете и т.д.
- Какие системные платформы используются в настоящее время? Какие платформы предполагается использовать в будущем?
- Какие еще приложения используются? Должно ли ваше приложение объединяться с ними?

2.4. Основные потребности пользователя

Следует перечислить основные проблемы или потребности так, как они осознаются пользователем. Для каждой проблемы нужно прояснить следующие моменты.

- В чем причины данной проблемы?
- Как она решается в настоящее время?
- Какие решения представляет себе пользователь?

Нужно понимать относительную важность для пользователя решения каждой из проблем. Методы упорядочения и накопительного голосования позволяют выделить проблемы, которые должны быть решены, и вопросы, которые желательно учесть.

2.5. Альтернативы и конкуренты

Нужно указать возможные альтернативы поведения пользователя. Среди них может быть покупка продукта конкурентов, создание собственного решения или просто сохранение существующей ситуации. Перечислите все известные конкурирующие варианты, которые существуют или могут возникнуть. Опишите основные преимущества и недостатки каждого варианта с точки зрения конечного пользователя.

2.5.1. Конкурент 1

3. Характеристика продукта

В данном разделе предлагается общее описание возможностей продукта, интерфейсов с другими приложениями и конфигураций систем. Как правило, он состоит из следующих трех подразделов.

3.1. Общее описание продукта

В данном подразделе следует описать, как продукт взаимодействует с другими связанными с ним продуктами и средой пользователя. Если продукт является независимым и самодостаточным, это необходимо указать. Если продукт является компонентом более крупной системы, в данном подразделе необходимо

описать, как эти системы взаимодействуют, а также указать соответствующие интерфейсы между системами. Простым способом отображения основных компонентов более крупной системы, взаимосвязей и внешних интерфейсов является блок-схема.

3.2. Определение позиции продукта

Предлагается общее определение, характеризующее на самом высоком уровне абстракции особое положение, которое продукт должен занять на рынке. Мур (Moore, 1991) назвал это определением позиции продукта и рекомендовал использовать для него следующую форму.

Для	(целевые потребители),
которые	(определение потребности или возможности),
(название продукта)	является (категория продукта),
который	(описание основных преимуществ, т.е. почему его обязательно нужно купить).
В отличие от	(перечисление основных альтернативных вариантов),
наш продукт	(описание основных его особенностей).

Это определение должно довести до сведения всех заинтересованных лиц назначение продукта и важность проекта.

3.3. Краткий обзор возможностей

Краткая характеристика основных возможностей и функций продукта. Например, в документе-концепции системы поддержки клиента данный подраздел может описывать решение проблем документирования, маршрутизации и отслеживания статуса, не вдаваясь в подробности осуществления этих функций.

Функции должны быть организованы так, чтобы список был понятен заказчику или тому, кто впервые читает данный документ. Ниже приводится образец, в котором в форме простой таблицы перечислены основные возможности и осуществляющие их поддержку функции.

Система поддержки заказчика

Предоставляемая пользователю возможность	Поддерживающая функция
Преимущество 1	Функция 1

3.4. Предположения и зависимости

Описываются предположения, изменение которых приведет к изменению концепции продукта. Например, предположение может состоять в том, что для аппаратного обеспечения программного продукта можно будет использовать определенную операционную систему. Если такой операционной системы не окажется, необходимо будет менять концепцию.

3.5. Вопросы затрат и цены

Для продаваемых внешним потребителям продуктов и многих приложений “для внутреннего использования” вопросы цены и затрат оказывают непосредственное влияние на определение и реализацию приложения. В данном разделе записываются все имеющиеся ограничения на затраты и цены. Например, затраты, связанные с дистрибуцией (количество дискет и компакт-дисков, создание мастер-компакт-диска), или другие затраты, входящие в стоимость проданных товаров (на руководство, упаковку), могут оказывать влияние на успех проекта или не иметь особого значения, в зависимости от природы приложения.

4. Атрибуты функций

Как и требования, функции имеют атрибуты, предоставляющие дополнительную информацию, которую можно использовать для оценки, отслеживания и определения очередности предлагаемых для реализации элементов разработки, а также управления ими. Ниже мы описали атрибуты, которые можно использовать в документе-концепции. Вам нужно описывать в данном разделе только те атрибуты (и их значения), которые вы выберете, чтобы все участники могли лучше понять содержание каждой функции.

4.1. Статус

Задается в результате переговоров и рассмотрения руководством проекта. Информация о статусе отражает ход процесса определения базового уровня проекта. Атрибут статуса функции может иметь следующие значения.

- **Предложена.** Используется для описания обсуждаемых функций, которые еще не рассмотрены и не приняты “официальным органом” – рабочей группой, состоящей из представителей команды проекта, руководства и пользователей или заказчиков.
- **Принята.** Возможности, которые “официальный орган” признал полезными и достижимыми и принял к реализации.
- **Включена.** Функции, включенные в базовый уровень на данный момент времени.

4.2. Приоритет

Приоритеты функций задаются представителями маркетинга, менеджером продукта или аналитиком базового уровня. Упорядочение функций по их относительной важности для конечного потребителя открывает диалог между заказчиками, аналитиками и членами команды разработчиков. Приоритеты используются для управления масштабом и определения очередности разработки. Ниже предложена одна из возможных схем задания приоритетов.

- **Критический.** Основные функции. Если их не удастся реализовать, система не будет удовлетворять потребности заказчика. В версии должны быть реализованы все критические функции, в противном случае график является нереальным.
- **Важный.** Функции, важные для успешной и эффективной работы системы в большинстве приложений. Данные функциональные возможности нельзя легко обеспечить иным способом. Если важные функции не войдут в реализацию, это может повлиять на удовлетворение пользователя или заказчика результатом работы или даже на доходы от продаж, но выпуск версии не должен задерживаться из-за нехватки некой важной функции.

■ **Полезный.** Функции, которые нужны в менее распространенных приложениях, будут использоваться не так часто или их можно достаточно эффективно заменить другими действиями. Если они не войдут в реализацию, это не окажет заметного воздействия на отношение заказчика или доходы.

4.3. Уровень трудозатрат

Определяется командой разработчиков и используется для управления масштабом и определения очередности разработки. Поскольку некоторые функции требуют больше времени и ресурсов, чем другие, оценка количества командо- или человеко-недель, строк кода или функциональных единиц помогает соразмерить сложность и оценить, что можно, а что нельзя осуществить за определенный период времени.

4.4. Риск

Задается командой разработчиков на основе вероятности того, что данная функция вызовет нежелательные последствия для проекта, такие как превышение средств, отставание от графика или даже закрытие проекта. Большинство менеджеров продукта считают достаточным деление рисков на категории *низкий, средний, высокий*, хотя возможна и более тонкая градация. Иногда риск можно оценить, измеряя меру неопределенности (диапазон) оценок времени работы команды.

4.5. Стабильность

Определяется аналитиком и командой разработчиков, исходя из вероятности того, что может измениться данная функция или понимание командой этой функции. Эта информация используется для того, чтобы помочь при определении приоритетов разработки и выявить те элементы, для которых следующим действием должно стать дополнительное исследование.

4.6. Целевая версия

Записывается, в какой версии продукта предполагается впервые реализовать данную функцию. Это поле можно использовать, чтобы поместить функции в базовый уровень конкретной версии. Комбинируя этот атрибут с полем статуса, команда может предлагать, записывать и обсуждать для версии различные функции, не приступая к их разработке. Будут реализовываться только функции, имеющие статус "Включенная", для которых определена целевая версия. При необходимости сокращения масштаба номер целевой версии может быть увеличен, так что элемент остается в документе-концепции, но его реализация будет отложена на более поздний срок.

4.7. Кому предназначена

Во многих проектах функции будут предназначаться "функциональным группам", ответственным за их дальнейшее исследование, написание программных требований, а также, возможно, реализацию. Это помогает членам команды разработчиков лучше понять свои обязанности.

4.8. Обоснование

Данное текстовое поле используется для отслеживания источника запрашиваемой функции. В этом поле записывается объяснение причины существования данной функции или ссылка на него. Например, ссылка может указывать на страницу, номер строки спецификации требований к продукту или временной маркер на видеозаписи важного интервью с клиентом.

5. Функции продукта

В данном разделе документируются функции продукта, которые обеспечивают необходимые возможности для удовлетворения потребностей пользователей. Каждая функция выполняет некую потребность пользователя. Например, функцией системы отслеживания состояния задачи может быть способность “предоставлять отчеты о выполнении”. Отчеты о выполнении, в свою очередь, помогают пользователю “лучше понять состояние задачи”.

Поскольку документ-концепция изучается широким кругом причастных к проекту лиц и служит основой для достижения соглашения, функции должны описываться на естественном языке пользователя. Описание функций должно быть кратким и ясным, как правило, одно-два предложения.

Для эффективного управления сложностью приложения мы рекомендуем, чтобы описание возможностей любой новой системы (или усовершенствования существующей) производилось на достаточно высоком уровне абстракции и состояло из 25–99 функций. Эти функции составляют основу для определения продукта, а также управления масштабом и проектом в целом. Каждая из них будет описана более подробно в последующих спецификациях.

Каждая функция данного раздела должна описывать внешнее поведение системы, которое ощущается пользователями, операторами или другими внешними системами.

5.1. Функция 1

5.2. Функция 2

6. Основные прецеденты

Следует описать несколько основных прецедентов, которые важны для архитектуры или лучше всего помогут читателю понять, как предполагается использовать систему.

7. Другие требования к продукту

7.1. Применимые стандарты

Перечисляются все стандарты, которым должен соответствовать продукт, такие как законы и инструкции (FDA, FCC), коммуникационные стандарты (TCP/IP, ISDN), стандарты совместимости платформ (Windows, UNIX), а также стандарты качества и безопасности (UL, ISO, CMM).

7.2. Системные требования

Определяются все системные требования, необходимые для поддержки приложения. Среди них могут быть поддерживаемые хостом операционные системы и сетевые платформы, конфигурации, память, периферические устройства и сопутствующее программное обеспечение.

7.3. Лицензирование и инсталляция

Вопросы лицензирования и инсталляции также могут оказывать непосредственное воздействие на трудоемкость разработки. Например, необходимость обеспечения серийного выпуска продукта, поддержки системы безопасности на основе паролей или сетевого лицензирования будет создавать дополнительные системные требования, которые следует учитывать при разработке. Инсталляционные требования могут также влиять на кодирование или вызывать потребность в отдельном инсталляционном программном обеспечении.

7.4. Требования производительности

К вопросам производительности относятся фактор нагрузки, создаваемой пользователем, ширина коммуникационного канала, пропускная способность, точность, надежность или время ответа при различных условиях загрузки.

8. Требования к документации

В данном разделе описывается, какую документацию необходимо разработать для поддержки успешного внедрения приложения.

8.1. Руководство пользователя

Нужно описать цель и содержание руководства пользователя, рассмотреть его желаемый объем, уровень детализации, потребность в индексе и глоссарии, а также, должно ли оно служить учебным пособием или скорее справочником и т.д. Следует также указать ограничения, связанные с форматированием и печатью.

8.2. Интерактивная подсказка

Многие приложения предлагают для помощи пользователю систему интерактивных подсказок. Подобные системы имеют уникальную природу: они сочетают в себе моменты программирования (такие, как создание гиперссылок) с моментами написания технических текстов (такими, как организация и презентация). Многие считают, что разработка системы интерактивных подсказок является проектом внутри проекта, который весьма выигрывает от предварительно проведенного ограничения масштаба и планирования действий.

8.3. Руководства по инсталляции, конфигурация и файл Read Me

Данный документ, содержащий инструкции по инсталляции и руководства по конфигурированию, важен для предложения всеобъемлющего решения. В качестве стандартного компонента обычно включается файл Read Me. Он может содержать раздел "Что нового в данной версии" и обсуждение совместимости с более ранними версиями. Большинство пользователей также приветствуют наличие в данном файле документации, где указаны все известные недоработки.

8.4. Маркировка и упаковка

Современные приложения должны иметь соответствующее внешнее оформление, которое начинается с упаковки продукта и его самообъявления в инсталляционных меню, открывающихся экранах, системах подсказок, GUI-диалогах и т.п. Примерами являются отметки об авторском праве и патентовании, а также логотипы компаний, стандартизованные пиктограммы и другие графические элементы и т.д.

9. Глоссарий

Глоссарий описывает все присущие данному проекту термины, в том числе все аббревиатуры, которые могут быть непонятны пользователю или другим читателям данного документа.

Приложение В

Образец пакета Modern SRS Package

Ниже приводится схема пакета Modern Software Requirement Specification (SRS), в котором используются как традиционные методы документирования, так и методы моделирования прецедентов. Для крупномасштабных систем рекомендуется дополнительная упаковка на уровне функций (или на другом подходящем уровне группировки подсистем). Например, если используется упаковка на уровне функций, спецификация будет содержать *все* относящиеся к реализации данной функции требования к программному обеспечению (или ссылки на них). В некоторых случаях пакет Modern SRS может представлять собой один или несколько документов, для которых данная схема служит отправной точкой. В других случаях он является логической конструкцией, которая может состоять из единственного физического документа со ссылками на другие, основанные на моделях и инструментальных средствах (UML-модели, прецеденты, архивы специальных средств работы с требованиями и др.), физические представления данных.

Название компании

**Название подразделения
(если нужно)**

Название проекта

**Номер документа спецификации
требований к программному
обеспечению (если нужно)**

© 1999 Название компании

История изменений

Дата	Версия	Описание	Автор
число/месяц/год	1.0	Исходная версия	Имя автора

Содержание

1. Введение

1.1. Цель

В данном разделе нужно указать цель данной SRS, которая должна полностью описывать внешнее поведение конкретного приложения или подсистемы, а также нефункциональные требования, ограничения проектирования и другие элементы, необходимые для обеспечения всестороннего описания требований к программному обеспечению.

1.2. Масштаб

Данный раздел содержит краткое описание программного приложения (функций или подсистем, на которые разбита система), для которого создается спецификация; кроме того, описывается, с какой моделью (моделями) прецедентов оно связано, а также все остальное, на что оказывает влияние данный документ.

1.3. Ссылки

Список ссылок или прилагаемых документов, связанных с данным проектом.

1.4. Предположения и зависимости

В данном разделе описывается техническая достижимость, доступность подсистем или компонентов и другие предположения, от которых может зависеть жизнеспособность описываемого данной SRS программного обеспечения.

2. Краткая характеристика модели прецедентов

Данный раздел содержит краткую характеристику модели прецедентов. Она предназначена для тех, кто интересуется поведением системы, — заказчиков, пользователей, архитекторов, авторов прецедентов, разработчиков, разработчиков прецедентов, тестологов, менеджеров, ревизоров и авторов документации. Для каждого прецедента необходимо указать следующее.

- Название прецедента.
- Краткое описание, объясняющее функцию прецедента и его роль в системе.
- Перечень акторов данного прецедента. (Более подробное определение этих акторов содержится в прилагаемом описании акторов.)
- Диаграмма модели прецедентов. (Здесь следует поместить диаграмму модели прецедентов в целом.)

3. Характеристика акторов

Здесь описываются все упомянутые в характеристике модели прецедентов акторы. Для каждого актора следует указать следующее.

- Имя
- Краткое описание

4. Требования

4.1. Функциональные требования

В данном разделе описываются функциональные требования к системе, выраженные на естественном языке. Для многих приложений это достаточно объемная информация, и следует продумать, как организовать данный раздел. Как правило, его организуют по функциям, но можно применять и другие методы, например по пользователям или подсистемам.

При использовании для сбора функций вспомогательных средств разработки приложений (инструментальных средств разработки требований, средств моделирования и т.д.) данный раздел документа будет содержать ссылки на эти данные и указывать местоположение и название применяемого для сбора данных инструментального средства.

4.2. Нефункциональные требования

Большая часть нефункциональных требований обычно записывается на естественном языке в данном разделе спецификации. Но нефункциональные требования могут также входить в спецификации прецедентов.

4.2.1. Практичность

В данный раздел следует включить все требования, влияющие на практичность программного обеспечения. Как правило, указывается следующее.

- Время, необходимое для обучения рядовых пользователей и пользователей с большими полномочиями, чтобы они научились эффективно выполнять определенные действия.
- Время выполнения типичных задач; или же практичность новой системы сравнивается с практичностью известных систем, которые пользователь знает и любит.
- Требования соответствия общепринятым стандартам практичности, таким как CUA IBM или опубликованные компанией Microsoft стандарты GUI для системы Windows 98.

Более подробная информация содержится в "Билле о правах пользователя" в главе 28.

4.2.2. Надежность

В данном разделе указываются требования к надежности системы.

- *Доступность.* Указывается, какой процент времени система доступна (xx.xx%), определяются часы использования и доступа для обслуживания, операции при ухудшении параметров системы и т.д.
- *Среднее время между отказами* (*mean time between failures, MTBF*). Обычно выражается в часах, но может указываться в днях, месяцах и годах.
- *Среднее время восстановления* (*mean time to repair, MTTR*). Сколько времени система может находиться в нерабочем состоянии после сбоя.
- *Точность.* С помощью некоего известного стандарта указывается требуемая точность (разрешающая способность) выводимой системой информации.
- *Максимально допустимый коэффициент ошибок и дефектов.* Как правило, выражается как число ошибок, приходящееся на KLOS (тысячу строк кода), или число ошибок, приходящихся на отдельную функцию.
- *Доля ошибок или дефектов различных типов.* Обычно ошибки разбиваются на следующие категории: незначительные, серьезные и критические. Требования должны определять, что понимается под "критической" ошибкой (такой, как полная потеря данных или и-

возможность использовать определенную часть функциональных возможностей системы).

4.2.3. Производительность

Здесь описываются характеристики производительности системы.

Следует указать время ответа для различных ситуаций. Если требуется, указываются названия соответствующих прецедентов.

- Время ответа для транзакции (среднее, максимальное)
- Пропускная способность (транзакций в секунду)
- Емкость (число пользователей или транзакций, которые может обслужить система)
- Режимы снижения производительности (допустимые режимы работы при ухудшении параметров системы)
- Использование ресурсов (память, диск, каналы связи)

4.2.4. Возможность сопровождения

Данный раздел содержит требования, способствующие улучшению возможности сопровождения и обслуживания создаваемой системы, в том числе стандарты кодирования, определенные соглашения, библиотеки классов, доступ для обслуживания и вспомогательные обслуживающие программы.

5. Требования к интерактивной документации пользователя и системе подсказок

Здесь описываются требования (если таковые имеются) к интерактивной документации пользователя, системе подсказок и т.д.

6. Ограничения проектирования

В данном разделе следует описать все ограничения проектирования создаваемой системы. Ограничения проектирования представляют решения по проектированию, которые являются обязательными и должны быть выполнены. Например, может задаваться язык программирования, требования к программным процессам, а также может предписываться использование определенных средств разработки, архитектурных и проектных ограничений, закупаемых компонентов и библиотек классов.

7. Закупаемые компоненты

В этом разделе описываются все используемые в системе закупаемые компоненты и соответствующие ограничения лицензирования или использования, а также все связанные с ними стандарты совместимости/взаимодействия или интерфейсов.

8. Интерфейсы

В данном разделе определяются интерфейсы, которые должны поддерживаться приложением. Раздел должен содержать достаточно подробное описание протоколов, портов, логических адресов и т.п., чтобы можно было разработать программное обеспечение и проверить его соответствие налагаемым на интерфейсы требованиям.

8.1. Интерфейсы пользователя

Описываются интерфейсы пользователя, которые должны быть реализованы программным обеспечением.

8.2. Аппаратные интерфейсы

Определяются все аппаратные интерфейсы, поддержку которых должно осуществлять программное обеспечение, в том числе логическая структура, физические адреса и ожидаемое поведение.

8.3. Интерфейсы программного обеспечения

Описываются программные интерфейсы с другими компонентами системы программного обеспечения. Это могут быть закупаемые компоненты, повторно используемые компоненты другого приложения или компоненты, разработанные для подсистем, не описываемых данной SRS, но с которыми данное программное приложение должно взаимодействовать.

8.4. Коммуникационные интерфейсы

Описываются все коммуникационные интерфейсы с другими системами или устройствами, такими как локальные сети или удаленные последовательные порты.

9. Требования лицензирования

Определяются все требования лицензирования или другие ограничивающие использование требования, которые оказывают влияние на программное обеспечение.

10. Замечания, касающиеся законности, авторских прав и т.д.

Описываются все необходимые гарантии, все отказы от ответственности, отметки об авторском праве, торговой марке или вопросы соответствия логотипу для программного обеспечения.

11. Применимые стандарты

Посредством ссылок указываются все стандарты (а также конкретные их разделы), которые применяются к описываемой системе. Например, это могут быть стандарты качества, некоторые законы или инструкции, а также отраслевые стандарты практичности, взаимодействия, интернационализации, соответствия операционной системы и т.д.

Индекс

Наличие индекса помогает читателю определять местонахождение в документе ключевых понятий и тем.

Глоссарий

Здесь описываются все термины данного приложения, а также все определения и принятые в проекте или компании сокращения, которые необходимы для понимания данного документа и приложения.

Приложения

В данный раздел следует включить все необходимые приложения. Ниже представлен образец приложения, который демонстрирует, как записывать прецеденты. Вы можете включать столько приложений, сколько сочтете нужным.

Приложение. Спецификации прецедентов

Данное приложение содержит подробное описание прецедентов системы. Предлагаемый ниже образец может служить отправной точкой.

История изменений

Дата	Версия	Описание	Автор
число/месяц/год	1.0	Исходная версия	Имя автора

Отметим, что история изменений приводится для каждого прецедента, включенного в приложение. Таблица истории изменений должна быть на первой странице каждого прецедента.

Содержание

Как правило, спецификация прецедента не слишком объемна, чтобы составлять для нее содержание. Но этот элемент может понадобиться, если возникают сложности при поиске отдельных фрагментов спецификации данного прецедента.

Название прецедента

Краткое описание

Роль и цель прецедента. (Для описания достаточно одного абзаца.)

Поток событий

Основной поток

Прецедент начинается, когда актор производит некое действие. Прецедент всегда инициируется неким актором. Прецедент должен описывать, что делает актор и что система делает в ответ; он должен выглядеть как диалог между актором и системой.

Прецедент должен описывать, что происходит внутри системы, а не как или почему это происходит. Если происходит обмен информацией, то нужно указать, какая информация поступает, а какая — отправляется. Кроме того, не очень понятно, что имеется в виду, если сказать, что актор вводит информацию о клиенте; лучше сказать, что актор вводит имя и адрес клиента. Для того чтобы не делать прецедент очень сложным и не запутаться в деталях, полезно использовать глоссарий, где можно определить, что понимается под информацией о клиенте.

Простые альтернативы можно описать в тексте прецедента. Если для описания того, что происходит в альтернативном случае, достаточно нескольких предложений, следует сделать это непосредственно в разделе, посвященном основному потоку событий. Если альтернативные потоки более сложные, нужно использовать отдельный раздел (Альтернативные потоки).

Иногда рисунок информативнее тысячи слов, хотя ничто не заменит чистой и ясной прозы. Если это поможет добиться большей ясности, можно включать в прецедент графические описания интерфейсов пользователя, потоков процессов или другие рисунки. Если для представления сложного процесса принятия решения необходимо использовать формальные средства спецификации, такие как диаграммы деятельности, несомненно, это следует делать! Аналогично, если поведение системы зависит от состояния, то диаграмма перехода состояний лучше прояснит его, чем это сделают страницы текста. Используйте представление, которое лучше всего подходит для вашей проблемы, но

будьте осторожны с терминологией, обозначениями или рисунками, которые могут быть непонятны вашей аудитории. Помните, что ваша задача – прояснить, а не запутать.

Альтернативные потоки

1. Первый альтернативный поток. Более сложные альтернативы следует описывать в отдельном разделе. Следует воспринимать альтернативные потоки как *варианты альтернативного поведения*; каждый альтернативный поток представляет некое альтернативное поведение (вариантов много из-за исключительных ситуаций, возникающих в основном потоке). Они могут быть произвольной длины, которая требуется для описания связанных с альтернативным поведением событий. Когда альтернативный поток заканчивается, события основного потока продолжаются, если не оговорено противное.

Альтернативные потоки могут, в свою очередь, также состоять из подразделов.

2. Второй альтернативный поток. Достаточно часто в прецеденте встречается несколько альтернативных потоков. Для большей ясности следует описывать каждую альтернативу отдельно. Использование альтернативных потоков упрощает понимание прецедента, а также предотвращает *декомпозицию* прецедентов на иерархии прецедентов. Следует помнить, что прецеденты – это только текстовые описания, и их главная задача в том, чтобы документировать поведение системы ясно, сжато и понятно.

Специальные требования

Здесь обычно приводятся имеющие отношение к данному прецеденту нефункциональные требования, которые непросто описать в тексте потока событий прецедента. Примерами таких требований могут служить требования законоодательства и инструкций, применяемые стандарты, атрибуты качества создаваемой системы, в том числе требования практичности, надежности, производительности или возможности сопровождения. В данном разделе следует также фиксировать другие требования, такие как описание операционных систем и сред, требования совместимости и ограничения проектирования.

1. Первое специальное требование

Предусловия

Предусловие прецедента – это состояние системы, в котором она должна находиться перед началом выполнения прецедента.

1. Предусловие 1

Постусловия

Постусловия прецедента – это перечень возможных состояний системы непосредственно после завершения прецедента.

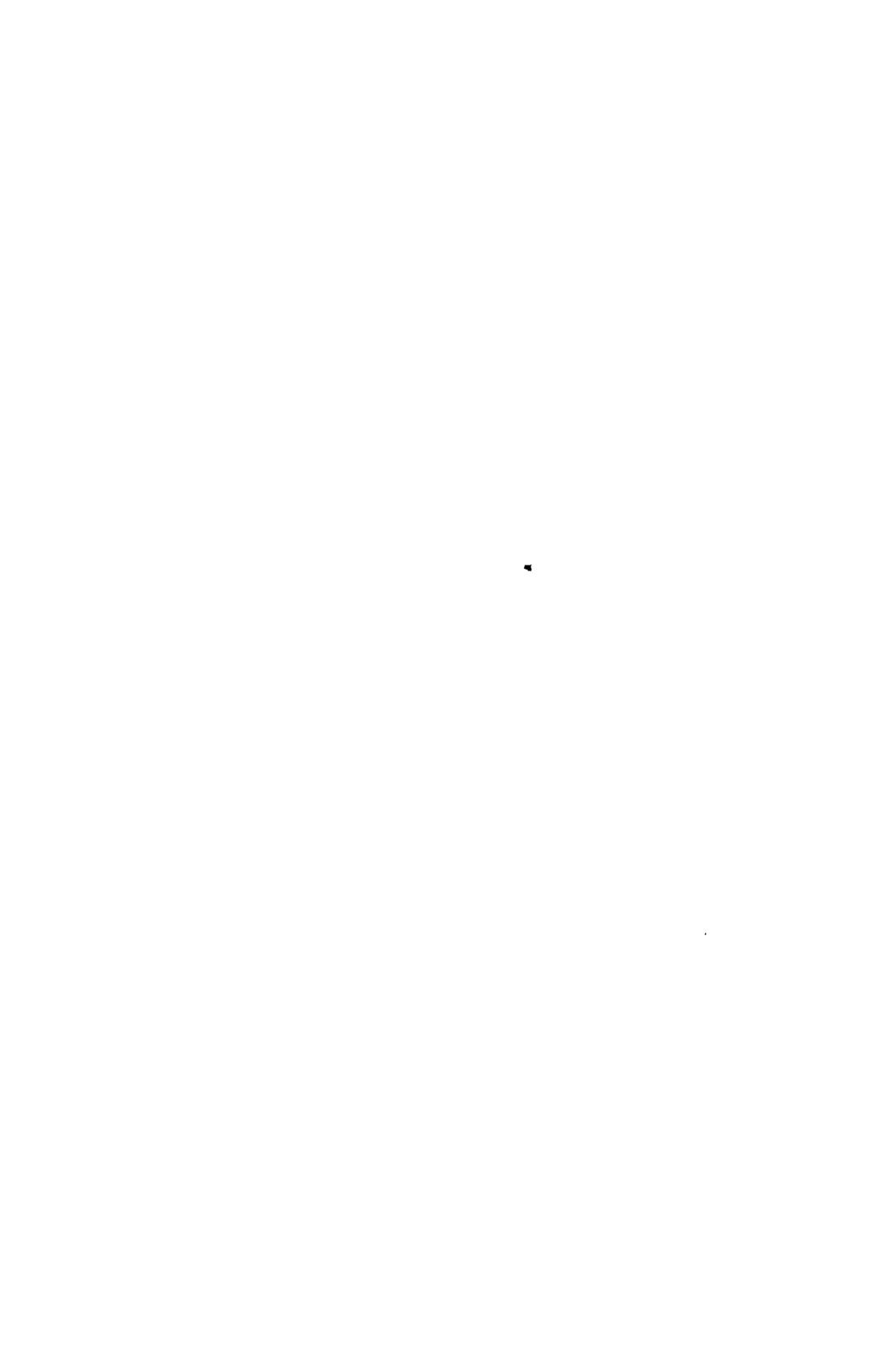
1. Постусловие 1

Точки расширения

Точки расширения прецедента.

1. Название точки расширения

Определение местоположения точки расширения в потоке событий.



Приложение Г

Принципы управления требованиями в стандартах SEI-CMM и ISO 9000

Принципы управления требованиями в стандарте SEI-CMM

В ноябре 1986 года в Институте программирования (Software Engineering Institute, SEI) при Университете Карнеги-Меллон (Carnegie-Mellon University) была предложена концепция совершенствования процесса разработки, которая была призвана повысить качество процесса программирования. В сентябре 1987 года SEI выпустил краткое описание структуры процесса, получившей дальнейшее развитие в работе Уотса Хамфри (Watts Humphrey's) *Managing the Software Process* (1989). К 1991 году развитие данной концепции привело к появлению "модели повышения уровня зрелости процессов разработки программного обеспечения" (Capability Maturity Model, CMM), версия 1.0. В 1993 году была выпущена версия 1.1 CMM (SEI 1993). В этой версии было определено пять "уровней зрелости" процесса разработки программного обеспечения в организации и предложены пути перехода от более низкого уровня к более высокому, как показано на рис. Г.1. СММ определяет для разработчиков набор действий, выполнение которых поможет организации усовершенствовать процесс программирования и добиться повторяемости, управляемости и измеримости.

Несмотря на продолжающиеся дебаты относительно преимуществ и недостатков СММ, накопленные данные свидетельствуют, что, следуя СММ, многим компаниям удалось повысить качество программирования и значительно снизить затраты на разработку приложений. Стандарт СММ достаточно давно используется многими организациями, так что существует статистика положительных результатов его применения. В идеале эти преимущества должны приводить к повышению производительности и значительному сокращению времени до выхода продукта на рынок. В условиях возрастания конкуренции нельзя игнорировать никакие усовершенствования, которые могут привести к повышению производительности процесса разработки программного обеспечения.

Стандарт СММ предлагает для совершенствования процесса структуру, состоящую из "ключевых областей процесса" или видов деятельности, которые, как следует из опыта, оказывают влияние на различные аспекты процесса разработки и качество программного обеспечения. В табл. Г.1 представлены ключевые области процесса для каждого из пяти уровней СММ. (Мы рассматриваем эту таблицу в данной книге потому, что в ней в качестве первой ключевой области процесса, которой необходимо уделить внимание для перехода с уровня 1 на уровень 2, названо управление требованиями.)

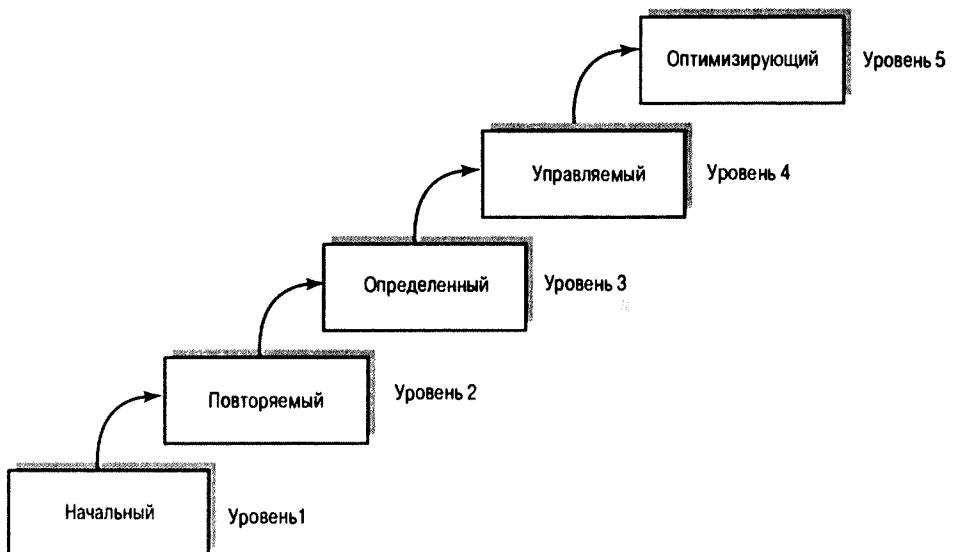


Рис. Г.1. "Уровни зрелости", согласно СММ

Таблица Г.1. Уровни СММ и ключевые области процесса

Уровень	Ключевые области процесса
1. Начальный. Разработка осуществляется произвольно (даже хаотично); успех зависит исключительно от индивидуальных героических усилий	Не выделяются
2. Повторяемый. Осуществляется базовое управление проектом с целью отслеживания функциональных возможностей приложения, затрат и графика	Управление требованиями Планирование разработки программного обеспечения Отслеживание и контроль разработки программы Управление программными субконтрактами Обеспечение качества программного обеспечения Управление конфигурацией программного обеспечения
3. Определенный. Процесс управления и проектирования является целостным, стандартизованным и документированным. Все проекты осуществляются на основе проверенной, приспособленной к конкретной задаче версии процесса	Фокусировка организационного процесса Определение организационного процесса Программа обучения Всеобъемлющее управление программированием Инженерия программного обеспечения Координация деятельности различных групп Тщательно организованный промежуточный контроль

Окончание табл. Г.1

Уровень	Ключевые области процесса
4. Управляемый. Производятся оценки характеристик процесса программирования и показателей качества программного обеспечения. Как процесс, так и программный продукт понятны и управляемы	Управления процессом на основе количественного анализа Управление качеством программного обеспечения
5. Оптимизирующий. Постоянное совершенствование процесса обеспечивается путем использования метрик и внедрения инновационных идей и технологий	Предупреждение дефектов Управление технологическими изменениями Управление изменениями процесса

Стандарт СММ характеризует деятельность по управлению требованиями следующим образом. Цель управления требованиями состоит в достижении общего понимания между заказчиком и командой программистов в том, что касается требований заказчика.

Это общее понимание служит основой соглашения между заказчиком и командой разработчиков, которое является основным документом, определяющим все последующие действия. Формируется базовый уровень требований, на основании которого осуществляется управление разработкой программного обеспечения. Стандартом СММ предусматривается, чтобы все планы, графики и рабочие программные продукты разрабатывались и, если нужно, модифицировались в соответствии с требованиями, налагаемыми на программное обеспечение. Таким образом, СММ способствует продвижению организации к целостному представлению, в котором технические требования должны находиться в соответствии с планами и действиями по разработке. Для осуществления поддержки данного процесса менеджеры программного обеспечения и заинтересованные лица (включая представителей заказчика и пользователей) должны документировать и пересматривать требования к программному обеспечению.

Спецификация программных требований является основным документом, который играет определяющую роль по отношению к другим элементам плана разработки. Среди требований есть как технические (поведение приложения), так и нетехнические (прочие требования к разработке, включая график, бюджет и т.п.). Помимо этого должны быть определены и документированы критерии приемки, т.е. тесты и измерения, которые будут использоваться для проверки того, что программное обеспечение удовлетворяет предъявляемым к нему требованиям.

Чтобы добиться осуществления этих целей и соответствия стандарту СММ в области управления требованиями, необходимо выделить определенные ресурсы и средства на управление требованиями. Нужно обучить членов группы программирования и других участников разработки деятельности по управлению требованиями. Обучение должно предусматривать изложение методов и стандартов, а также способствовать формированию понимания командой разработчиков особенностей прикладной области и существующих в ней проблем.

Требованиями нужно управлять, и они должны служить основой для планирования деятельности по разработке программного обеспечения и создания рабочих продуктов. Изменения требований должны изучаться и вноситься в планы разработки, а их воздействие необходимо оценивать и обсуждать с заинтересованными группами. Чтобы получить отклик на результаты разработки, а также проверить их соответствие требованиям, стандарт СММ предлагает методические указания по оценке и анализу, а также по верификации реализации. Предлагается отслеживать следующее.

- Статус каждого существующего требования
- Деятельность по внесению изменений в требования, накопительный эффект изменений
- Общее количество открытых, предложенных, принятых и включенных в базовый уровень изменений

Одним из наиболее прогрессивных аспектов модели СММ является понимание того, что управление требованиями является не просто процессом создания документа, после чего можно двигаться дальше, как часто предписывалось "водопадными" методологиями 70-х годов. В СММ требования являются живыми сущностями, находящимися в самом центре процесса разработки приложения. Неудивительно, что управление требованиями присутствует фактически на всех уровнях модели процесса разработки и во многих его ключевых областях. Когда организация переходит на третий уровень шкалы СММ, основное внимание уделяется управлению деятельностью по разработке программного обеспечения в соответствии с определенными и документированными стандартными принципами. Ключевыми областями процесса для уровня 3 являются фокусировка организационного процесса, его определение, программа обучения, всеобъемлющее управление разработкой программного обеспечения, инженерия программного обеспечения, координация действий различных групп и тщательно организованный промежуточный контроль. Задача инженерии программного обеспечения в том, чтобы организация проводила все действия по программированию согласованно и в результате могла успешно создавать высококачественные программные продукты. Согласно основополагающим принципам инженерии программного обеспечения, требования к программному обеспечению разрабатываются, поддерживаются, документируются и верифицируются посредством систематического анализа требований в соответствии с заранее определенным процессом разработки программного обеспечения (SEI 1993).

Процесс анализа необходим, чтобы убедиться в том, что требования имеют смысл, четко сформулированы, являются полными и недвусмыслимыми, согласованы друг с другом и тестируемы. Предлагаются различные методы анализа, в том числе имитация, моделирование, создание сценариев, а также функциональная и объектно-ориентированная декомпозиция. Результатом данного процесса будет более глубокое понимание требований к приложению, что должно быть отражено в пересмотренной документации требований. Кроме того, требования также анализируются группой, отвечающей за тестирование и приемку системы, чтобы удостовериться в возможности их тестирования.

Полученный в результате документ требований к программному обеспечению изучается и признается заинтересованными сторонами, что призвано обеспечить отражение в требованиях позиций всех этих сторон. Среди них — заказчики, конечные пользователи, руководство проекта, разработчики и тестологи. Чтобы можно было управлять изме-

нениямн требований, СММ требует помещать документ требований к программному обеспечению под контроль управления конфигурацией.

Еще одним важным аспектом СММ является *трассировка*. Следуя стандарту СММ, все важные рабочие программные продукты должны документироваться, должно быть обеспечено ведение документации и легкий доступ к ней. Требования к программному обеспечению, технический проект, программный код и тестовые примеры трассируются к источнику, из которого они были получены, и к продуктам последующих действий по разработке. С помощью трассировки требований можно анализировать воздействие изменения до того, как оно произведено, а также определять, на какие компоненты повлияет внесение изменения. Кроме того, трассировка предлагает механизм определения адекватности тестового покрытия.

Все принятые изменения полностью отслеживаются. Ведется также документация трассировки всех имеющихся требований. Чтобы определить функциональные возможности и качество программных продуктов, а также статус определенной деятельности по разработке программного обеспечения, предлагается отслеживать следующее.

- Статус каждого существующего требования на протяжении жизненного цикла
- Деятельность по изменению существующих требований
- Распределение существующих требований по категориям

В СММ изменения считаются неотъемлемой составной частью действий по разработке программного обеспечения. Вместо замороженных спецификаций создается стабильный базовый уровень требований, которые тщательно изучены, документированы и контролируются соответствующими системами, обеспечивающими управление изменениями. В частности, ниже приводятся некоторые требования СММ.

- По мере совершенствования понимания программы предлагаются, анализируются и вносятся соответствующие изменения в программные рабочие продукты и действия по их разработке. При необходимости внесения изменений в требования, они принимаются и вносятся до того, как производятся изменения любых рабочих продуктов или действий.
- Воздействие изменения должно определяться до того, как оно производится.
- Изменения обсуждаются и доводятся до сведения групп, на работу которых влияют данные изменения.
- Все изменения полностью отслеживаются.

В результате, стандарт СММ предлагает всеобъемлющее представление действий, необходимых для повышения качества программного обеспечения и увеличения производительности. Управление требованиями является неотъемлемой составной частью этого процесса, в котором требования выступают живыми сущностями и находятся в центре действий по разработке. Выявленные требования документируются и обрабатываются с той же тщательностью, что и состоящие из программного кода рабочие продукты. Этот процесс позволяет команде управлять проектом и его масштабом. Наконец, контролируемое внесение изменений в требования позволяет удержать проект под контролем и гарантировать надежное повторяемое производство высококачественных программных продуктов.

Хотя все изложенное выше обеспечивает важную возможность “проверить правильность” концепции управления требованиями и содержит некие высокоуровневые советы

по включению ориентированных на требования процессов в жизненный цикл разработки, там не говорится, как осуществлять управление требованиями. Подробно деятельность по выявлению, организации, документированию и обработке требований описывается в данной книге, и на нее, несомненно, повлияли принципы СММ.

Принципы управления требованиями в стандарте ISO 9000

На протяжении последнего десятилетия многие организации во всем мире использовали серию стандартов определения качества, известные как ISO 9000, для повышения эффективности и производительности работы, а также снижения затрат. ISO 9000 был принят Европейским сообществом (в Европе стандарт получил название EN29000) и стал важным фактом в международной торговле; организации, желающие вести бизнес в Европе, как правило, должны иметь сертификат ISO 9000. Для сертификации требуется оценка "на месте" уполномоченным ISO чиновником; прошедшие оценку компании извещаются о том, что они будут периодически подвергаться повторной оценке, чтобы подтвердить свой сертификат.

Стандарт ISO 9000 состоит из пяти частей.

1. ISO 9000. Руководящие указания по выбору и применению
2. ISO 9001. Руководящие указания по обеспечению качества при проектировании, разработке, производстве, монтаже и обслуживании
3. ISO 9002. Руководящие указания по обеспечению качества для компаний, в основном занимающихся производством
4. ISO 9003. Руководящие указания по обеспечению качества для компаний, в основном занимающихся дистрибуцией
5. ISO 9004. Общее руководство качеством и элементы системы качества

Один из документов стандарта, ISO 9000-3, содержит руководящие указания по применению стандартов ISO 9001 при разработке, поставке и обслуживании программного обеспечения. Часть 5.3 этого документа предписывает следующее: *чтобы заниматься разработкой, поставщик должен иметь полный набор недвусмысленных функциональных требований*. Этот же документ указывает, что предоставляемая поставщику (которого мы повсюду называли "разработчиком") информация должна содержать все требования по производительности, безопасности, надежности, защите и соблюдению секретности, которые в совокупности определяют, является ли разработанная система приемлемой.

Как и СММ, стандарты ISO 9000 долго были предметом споров, особенно в некоторых организациях США, которые беспокоились о том, что стандарты могут превратиться в бюрократические требования чрезмерного документирования. Мы не ставим перед собой задачу защищать или критиковать ISO 9000; как и все подобные концепции на уровне "здравого смысла", данный стандарт можно использовать правильно и неправильно. Но поскольку многие организации по разным причинам приняли ISO 9000 (то ли потому, что считали его удачным, то ли потому, что это было необходимым условием ведения бизнеса в Европе и других частях мира), интересно рассмотреть, какое внимание уделяется в нем принципам управления требованиями. Так, ISO 9000 подчеркивает

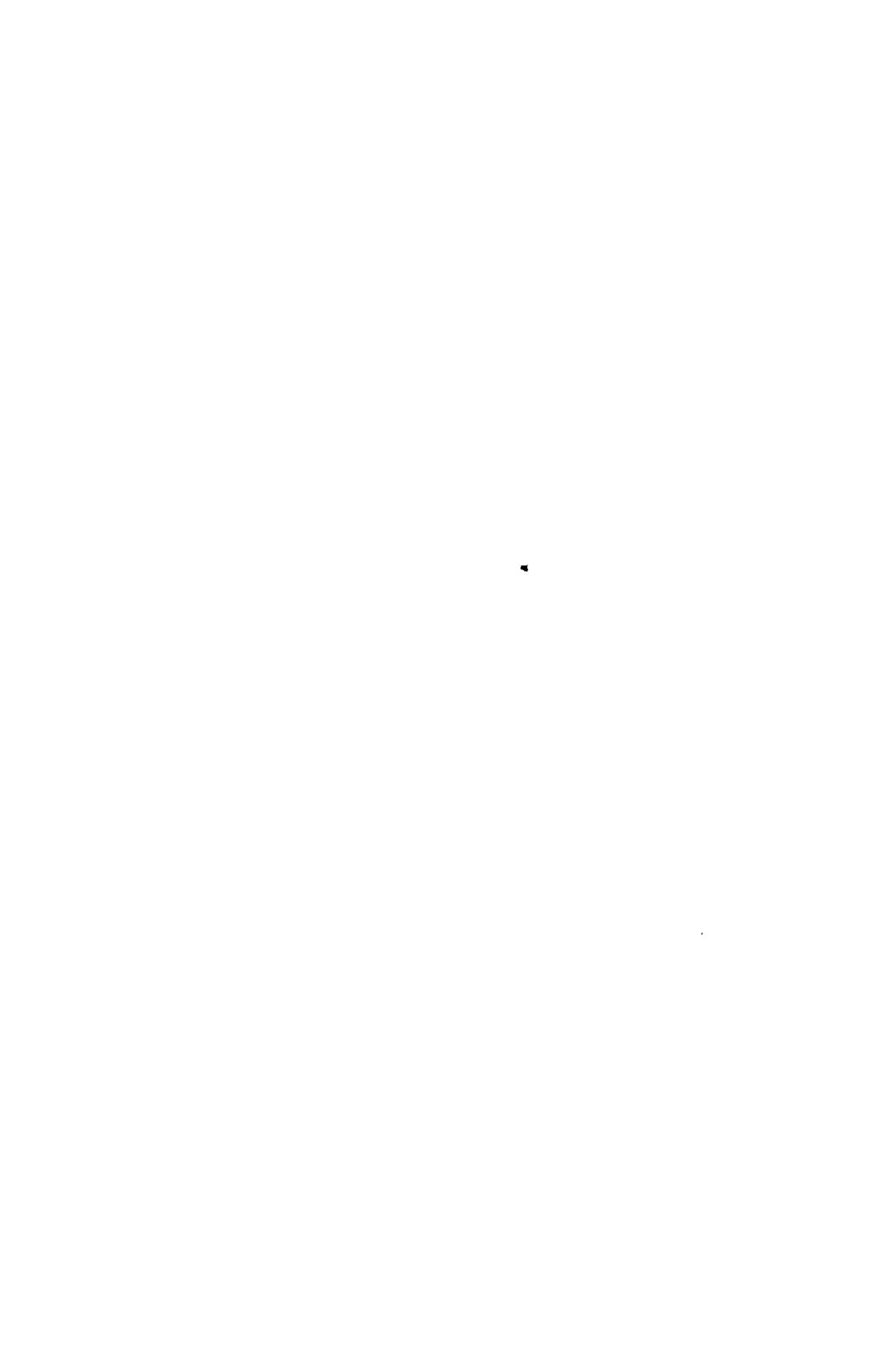
необходимость *взаимной кооперации* заказчика и разработчика систем программного обеспечения; в частности, он требует выполнения следующего.

- Назначить представителей обеих групп, ответственных за задание требований
- Определить методы и процедуры согласования и принятия изменений требований
- Предпринять усилия по предотвращению неправильного понимания требований
- Определить процедуры записи и пересмотра результатов обсуждений требований

Хотя кажется, что это очевидные и всем известные действия, которыми можно пренебречь, стоит вспомнить, что происходит при оценке, необходимой для прохождения сертификации. В организацию приходит инспектор и спрашивает: “Где ваши методы и процедуры принятия изменений требований? Предъявите мне их в письменном виде. Я хотел бы провести внеплановые проверки отдельных команд разработчиков и убедиться, что данные процедуры действительно выполняются”.

ISO 9000 также требует, чтобы исходная информация для фазы разработки проекта – фазы жизненного цикла, когда обычно выполняется техническое проектирование и программирование – была определена и документирована. Этой исходной информацией, очевидно, являются требования, и ISO 9000 требует, чтобы они были сформулированы таким образом, чтобы их выполнение можно было проверить. Стандарт ISO 9000 также предписывает использовать процессы систематического разрешения проблем неполноты, неоднозначности и конфликта требования. Одним из важных следствий такого внимательного отношения к требованиям с самого начала разработки является то, что при организованном проведении технического проектирования и разработки можно рассчитывать, что в результате работы появится система, которая действительно соответствует спецификации (т.е. набору требований), а не полагаться на то, что качество работы системы будет обеспечено в результате проведения чрезмерного количества испытаний в конце жизненного цикла разработки.

Как и SEI-CMM, ISO 9000 ничего не говорит о том, как осуществлять управление требованиями. Факт существования официального процесса, который предписывает выбрать “официальных” представителей из числа пользователей и разработчиков для обсуждения требований к системе, очевидно, не гарантирует, что эти два индивидуума будут в состоянии выявить и документировать правильные требования. Но вооружившись описанными в данной книге процедурами и методами, мы сможем создать всеобъемлющий подход к управлению требованиями, который удовлетворит самых придирчивых инспекторов ISO 9000 и CMM.



Приложение Д

Принципы управления требованиями в Rational Unified Process

Совместно с Филиппом Крачтеном (Philippe Kruchten) и Лесли Пробаско (Leslee Probasco)

В данной книге предлагается обзор лучших практических наработок в управлении требованиями. Описанные в книге профессиональные приемы и предложенный в главе 35 рецепт управления требованиями помогут команде пойти по правильному пути при разработке следующего проекта. Но для того чтобы повысить вероятность успеха, необходимо стимулировать и поддерживать применение этих полезных практических навыков в процессе разработки. Это должно осуществляться способом, который органично соединяет управление требованиями с другими видами деятельности по разработке программного обеспечения, включая проектирование, реализацию, тестирование и развертывание. В идеале информация должна предоставляться в интерактивном режиме в рабочей среде команды. Нужно описать, какие виды деятельности осуществляют определенные члены команды и когда им нужно предоставить результаты их деятельности для использования другими членами команды. Именно в этом состоит роль *процесса разработки программного обеспечения*. В данном приложении мы рассмотрим пример промышленного процесса разработки программного обеспечения, *Rational Unified Process* (RUP), и увидим, как в нем отражаются представленные в книге профессиональные приемы.

Rational Unified Process (процесс разработки программного обеспечения, который был разработан и превращен в коммерческий продукт корпорацией Rational Software Corporation (1999)), вобрал в себя лучшие практические приемы в отрасли разработки программного обеспечения. Процесс основан на precedентах и использует итеративный подход к разработке программного обеспечения. Он включает в себя объектно-ориентированные методы, и многие виды деятельности в нем направлены на разработку *моделей*, описанных посредством UML. RUP является развитием методологий Objectory (Джейкобсон (Jacobson), Кристерсон (Christerson), Джонсон (Jonsson), 1992) и Rational Approach. В него внесли свой вклад многие эксперты отрасли, среди которых авторы данной книги, команды разработчиков из компаний Requisite, Inc.; SQA, Inc.; и многие другие.

Как продукт, RUP представляет собой доступное посредством Web руководство, которое попадает непосредственно на рабочий стол разработчика программного обеспечения. Он состоит примерно из 2800 файлов, представляющих основанные на HTML ин-

терактивные настольные инструкции, составленные так, что они могут удовлетворить потребности широкого круга организаций, разрабатывающих программное обеспечение.

Хотя в нем используется несколько иная терминология, чем в данной книге, RUP обеспечивает реализацию предложенных в книге основных принципов управления требованиями в такой форме, что они могут непосредственно применяться командой разработчиков программного обеспечения.

Структура RUP¹

Процесс описывает, *кто*, *как* и *когда* делает. Rational Unified Process описывается с помощью четырех ключевых элементов моделирования.

- Сотрудники (workers), "кто"
- Деятельности (activities), "как"
- Артефакты (artifacts), "что"
- Рабочие процессы (workflows), "когда"

Рассмотрим рис. Д.1. Сотрудник задает поведение и обязанности отдельного индивидуума или группы совместно работающих индивидуумов (команды). Поведение описывается с помощью *деятельностей*, осуществляемых сотрудником, и каждый сотрудник ассоциируется с набором связанных *деятельностей*. Обязанности каждого сотрудника выражаются по отношению к определенным *артефактам* или рабочим продуктам, которые сотрудник создает, модифицирует или контролирует.

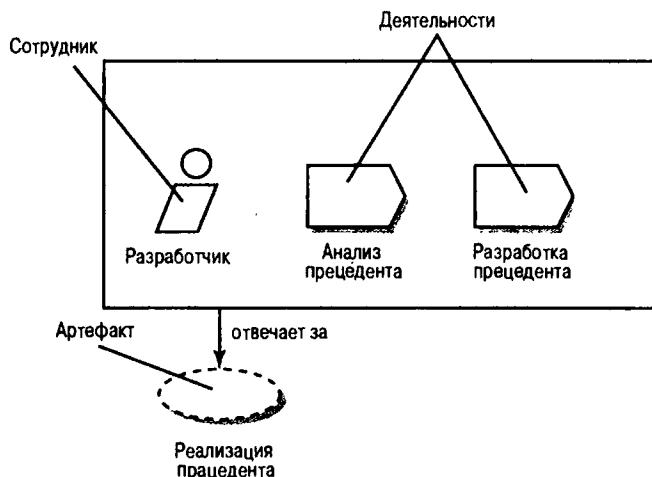


Рис. Д.1. Сотрудники, деятельности и артефакты

¹ Этот раздел взят из работы Филиппа Крачтена (Philippe Kruchten), The Rational Unified Process – An Introduction (Reading, MA: Addison-Wesley Longman, 1999) стр. 35–48, и воспроизведется с согласия издателя.

Рабочие процессы позволяют группировать деятельности в осмыслиенные последовательности, которые обеспечивают некий результат для организации-разработчика, и показывают, как взаимодействуют различные сотрудники. В дополнение к этим четырем основным понятиям, RUP предлагает *методические указания* по осуществлению соответствующих деятельности, *шаблоны* основных артефактов, а также *руководства* по применению автоматических средств разработки программного обеспечения.

Принципы управления требованиями в RUP

Приемы управления требованиями организованы в RUP в *рабочий процесс разработки требований*, один из девяти основных рабочих процессов. В ходе данного процесса производятся и обновляются следующие артефакты (рис. Д.2).

- *Запросы заинтересованных лиц*; т.е. коллекция различных запросов, в том числе формальные запросы изменений, потребности или другие пожелания заинтересованных лиц на протяжении жизненного цикла проекта, которые могут повлиять на требования к продукту.
- *Документ-концепция (Vision document)*, который кратко характеризует концепцию рассматриваемой системы: основные характеристики, функции, потребности заинтересованных лиц, а также основные предоставляемые услуги.
- *Модель прецедентов*, которая представляет собой организованный набор прецедентов, составляющих основную массу требований.
- *Дополнительные спецификации*, фиксирующие все требования, которые невозможно непосредственно связать с конкретными прецедентами; в частности, многие нефункциональные требования и ограничения проектирования.

Два последних артефакта вместе образуют единую форму, которую мы в данной книге назвали пакетом *Modern SRS Package*.

В результате данного рабочего процесса разрабатываются также другие артефакты, которые перечислены ниже.

- *Атрибуты требований*; информационный архив, содержащий связанную с требованиями информацию, которая используется для отслеживания статуса требований и обеспечения их трассировки к другим элементам проекта.
- *Раскладовки прецедентов*, систематически производимые для основных прецедентов с участием акторов-людей, чтобы моделировать интерфейс пользователя и исследовать некоторые требования практичности.
- *Прототипы интерфейса пользователя*, разрабатываемые для получения реакции различных заинтересованных лиц.
- *Глоссарий*, который содержит определения используемых в данном проекте терминов. В данном рабочем процессе участвуют следующие сотрудники.
- *Заинтересованные лица, заказчик, конечный пользователь*; тот, кто играет для организации-разработчика роль поставщика исходной информации для процесса разработки требований (например, это может быть менеджер по маркетингу).

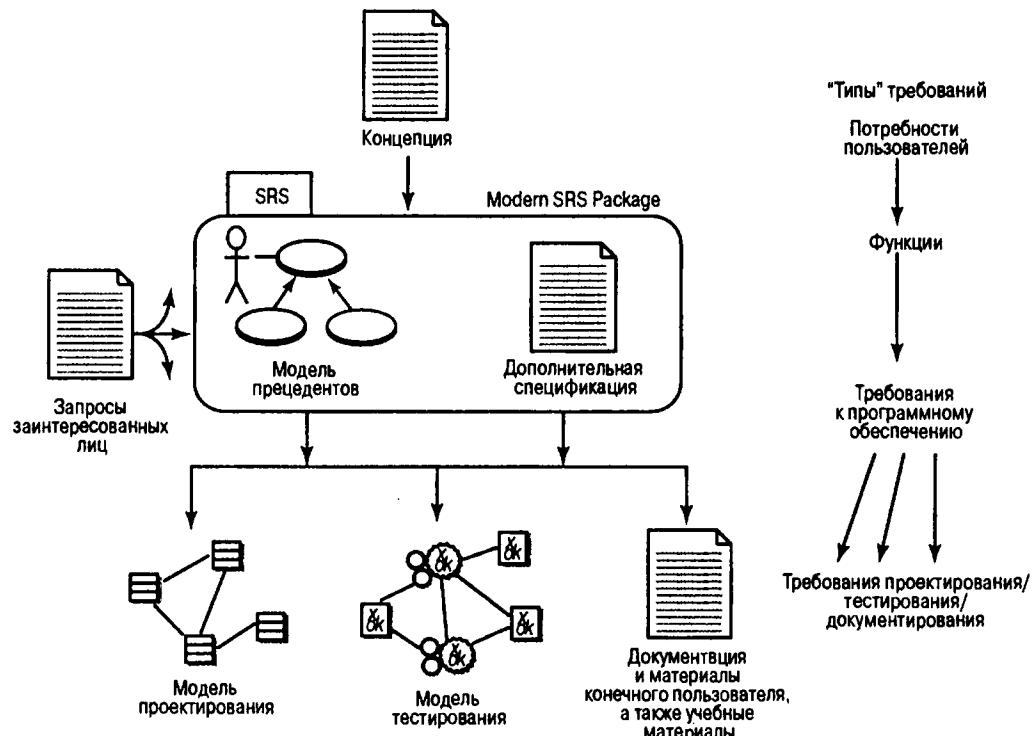


Рис. Д.2. Рабочий процесс разработки требований

- **Системный аналитик**, который осуществляет руководство и координацию действий по выявлению требований и моделированию прецедентов, определяя функции и очерчивая границы системы; например, он устанавливает, какие существуют акторы и прецеденты, как они взаимодействуют, а также задает нефункциональные требования и ограничения проектирования.
- **Составитель спецификаций прецедентов**, который составляет подробную спецификацию некой части функциональных возможностей системы, описывая аспект требований одного или нескольких прецедентов.
- **Проектировщик интерфейса пользователя (UI)**, который разрабатывает раскладовки прецедентов и прототипы UI и привлекает других участников к их оценке.
- **Ревизор требований** (этую роль обычно выполняют отдельные члены команды), который планирует и проводит формальную ревизию модели прецедентов и других требований, указанных в дополнительных спецификациях.

Этапы рабочего процесса разработки требований организованы в RUP в шесть рабочих процессов следующего уровня (подпроцессов), которые полностью соответствуют шести описанным в данной книге наборам приемов работы с требованиями.

Анализ проблемы

Как видно из рис. Д.3, задача данного подпроцесса состоит в следующем.

- Создать документ-концепцию проекта
- Согласовать функции и цели

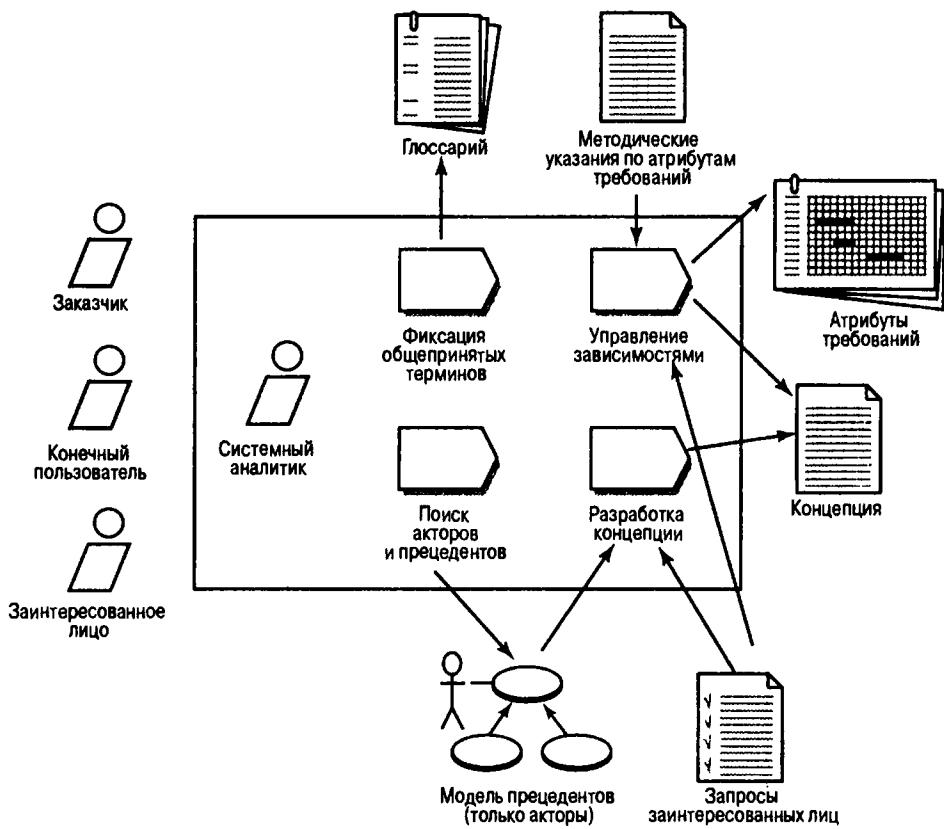


Рис. Д.3. Анализ проблемы

К данному подпроцессу можно несколько раз возвращаться в фазах *начала* и *раннего исследования*. По мере более четкого понимания *запросов заинтересованных лиц* будут эволюционировать как решения, касающиеся бизнес-процессов, так и технические решения.

Основная деятельность данного подпроцесса состоит в разработке *документа-концепции*, который является высокогоревым представлением взгляда пользователя или заказчика на создаваемую систему. В документе-концепции исходные требования выражаются в виде ключевых функций, которые должна предоставить система, чтобы решить наиболее критические проблемы. Функциям присваиваются следующие *атрибуты*: объяснение, относительная важность (или приоритет), источник запроса и т.д., что создает возможность управлять ими. По мере развития *концепции*, *системный аналитик* выявляет пользователей и интерфейсы системы – *акторы* системы.

Понимание потребностей заинтересованных лиц

Задача данного подпроцесса состоит в сборе информации от заинтересованных лиц проекта (рис. Д.4). Собранные запросы заинтересованных лиц могут рассматриваться как “список пожеланий”, который можно использовать в качестве исходной информации для определения модели прецедентов, прецедентов и дополнительных спецификаций. Как правило, этот подпроцесс выполняется только в итерациях фаз начала и исследования.

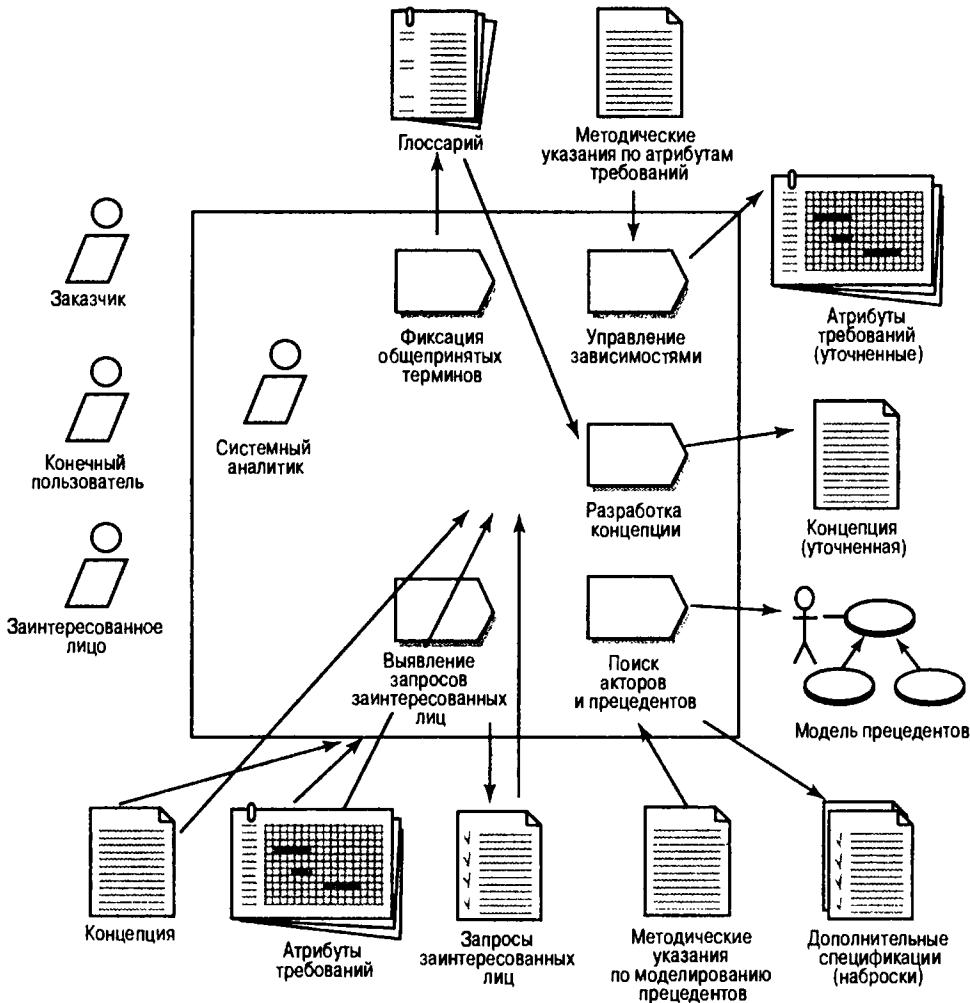


Рис. Д.4. Понимание потребностей заинтересованных лиц

Главная деятельность заключается в выявлении запросов заинтересованных лиц. Основной результат — коллекция упорядоченных по приоритету запросов заинтересованных лиц, что дает возможность уточнить документ-концепцию, а также лучше понять атрибуты требований. В рамках данного процесса можно также начать обсуждение системы, ис-

пользуя *прецеденты* и *акторы*. Еще одним важным результатом является обновленный *глоссарий*, который служит общим терминологическим словарем членов команды.

Определение системы

Задача этого подпроцесса (рис. Д.5) заключается в следующем.

- Достичь единого понимания системы командой проекта
- Выполнить высокоуровневый анализ результатов сбора *запросов заинтересованных лиц*
- Более формально документировать результаты в виде моделей и документов

Как правило, это выполняется только в итерациях фаз начала и исследования.

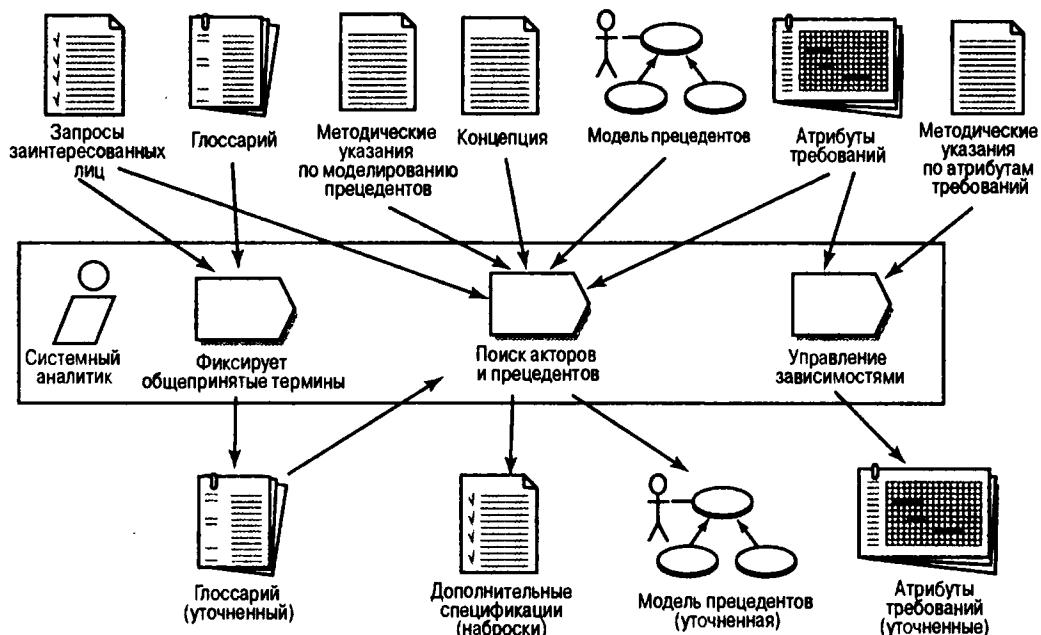


Рис. Д.5. Определение системы

Процессы анализа проблемы и понимания потребностей заинтересованных лиц приводят к созданию ранних итераций ключевых определений системы, в том числе *документ-концепции*, первого наброска *модели precedentов*, а также *атрибутов требований*. При определении системы основное внимание уделяется более полному определению *акторов* и *precedентов* и разработке *дополнительных спецификаций*.

Управление масштабом проекта

Задача данного подпроцесса (рис. Д.6) состоит в следующем.

- Определить исходные данные для отбора *требований*, которые включаются в текущую итерацию

- Определить набор функций и прецедентов (или сценариев), представляющих наиболее важные функциональные возможности
- Определить, какие атрибуты требований и трансформации поддерживать

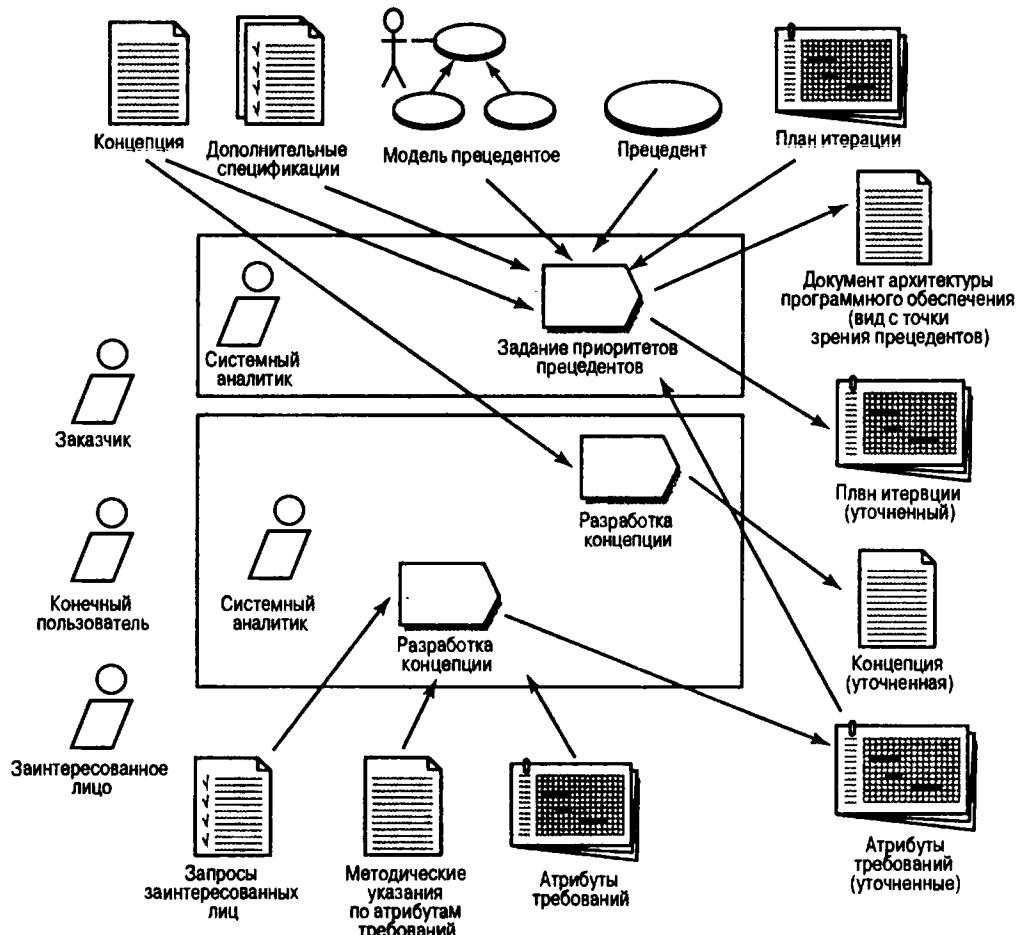


Рис. Д.6. Управление масштабом системы

Хотя управлением масштабом проекта нужно заниматься постоянно, более глубокое понимание функциональных возможностей системы, полученное при определении основных акторов, прецедентов и дополнительных спецификаций, позволит системному аналитику более точно определить атрибуты требований (приоритет, трудоемкость, стоимость, значение риска и т.д.) и предоставит архитектору возможность выявить важные для архитектуры прецеденты. Среди элементов, разрабатываемых в процессе управления масштабом, есть такой, который не встречался в других подпроцессах процесса разработки требований, – это план итераций, параллельно разрабатываемый руководством проекта. План итераций определяет количество и частоту планируемых для данной версии итераций. Масштаб проекта, определенный в процессе управления масштабом, будет

оказывать существенное влияние на *план итераций*, так как элементы с самым высоким риском будут запланированы для ранних итераций. Другими важными результатами процесса управления масштабом являются исходная итерация *документа архитектуры программного обеспечения* и пересмотренный *документ-концепция*, который отражает более совершенное понимание *системным аналитиком и основными заинтересованными лицами функциональных возможностей системы и имеющихся ресурсов*.

Уточнение определения системы

Задача данного подпроцесса (рис. Д.7) состоит в дальнейшем уточнении *требований*. Для этого необходимо следующее.

- Подробно описать потоки событий *прецедентов*
- Детализировать *дополнительные спецификации*
- Создать модели и прототипы интерфейсов пользователя

Уточнение системы начинается со схематически определенных *прецедентов*, хотя бы кратко описанных *акторов* и нового понимания масштаба проекта, отраженного в переопределенных приоритетах *функций концепции*, которые считаются достижимыми практически неизменных сроках и бюджете. Результатом данного рабочего процесса является углубление понимание функциональных возможностей системы, выраженное в детализированных *прецедентах*, исправленных и *детализированных дополнительных спецификациях*, а также элементах интерфейса пользователя.

Обработка изменений требований

Задача данного подпроцесса (рис. Д.8) состоит в следующем.

- Структурировать модель *прецедентов*
- Задать соответствующие значения *атрибутов требований* и *трассировочные связи*
- Провести формальную проверку того, что результаты процесса разработки требований соответствуют представлениям заказчика о системе

Изменения *требований*, естественно, оказывают воздействие на *модели*, созданные в *процессе анализа и проектирования*, а также на *модели тестирования*, созданные в рамках *рабочего процесса тестирования*. Ключевыми для понимания этих действий являются отношения *трассировки требований*, выявленные при осуществлении деятельности по *управлению зависимостями* в рамках данного и других рабочих процессов.

Еще одним важным моментом является отслеживание истории изменения требований. Фиксируя, в чем состоит данное изменение требования и почему оно производится, *ревизоры* (в данном случае эту роль выполняет кто-нибудь из команды разработчиков программного обеспечения, на чью работу влияет это изменение) получают информацию, необходимую для того, чтобы правильно отреагировать на данное изменение.

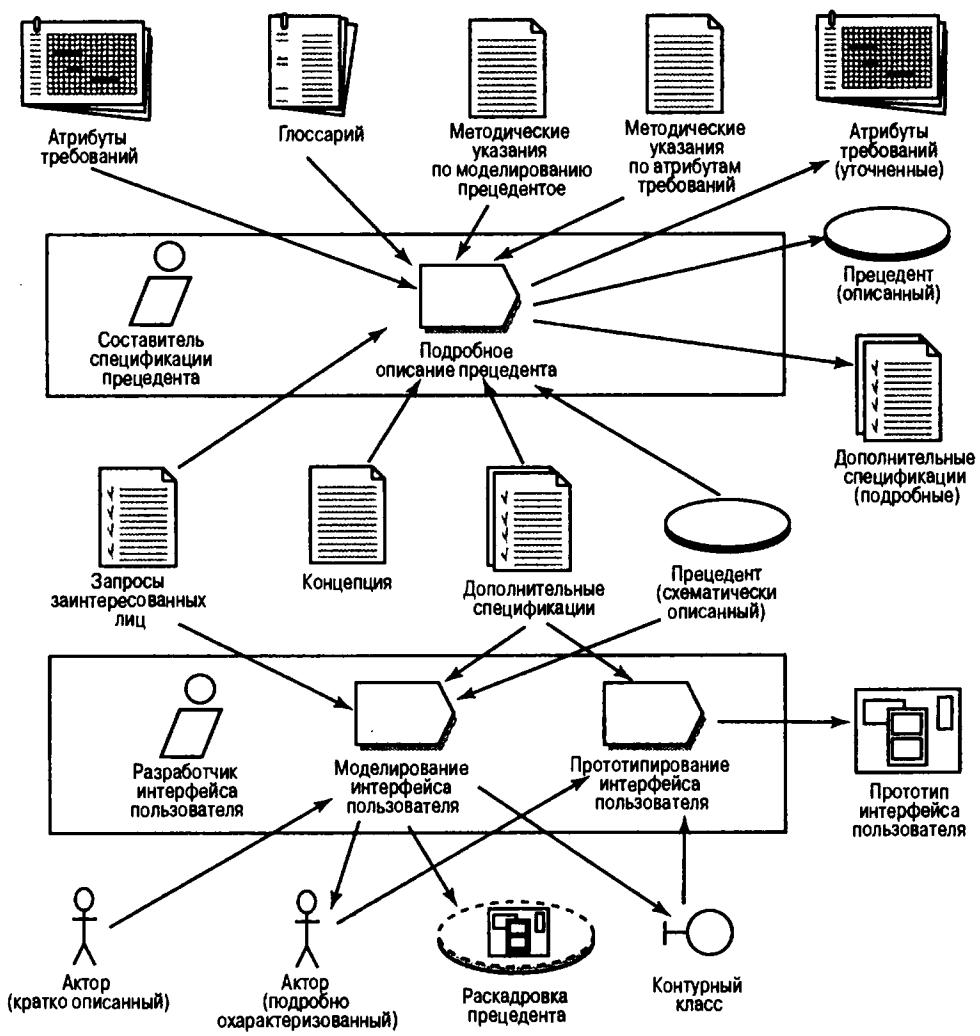


Рис. Д.7. Уточнение определения системы

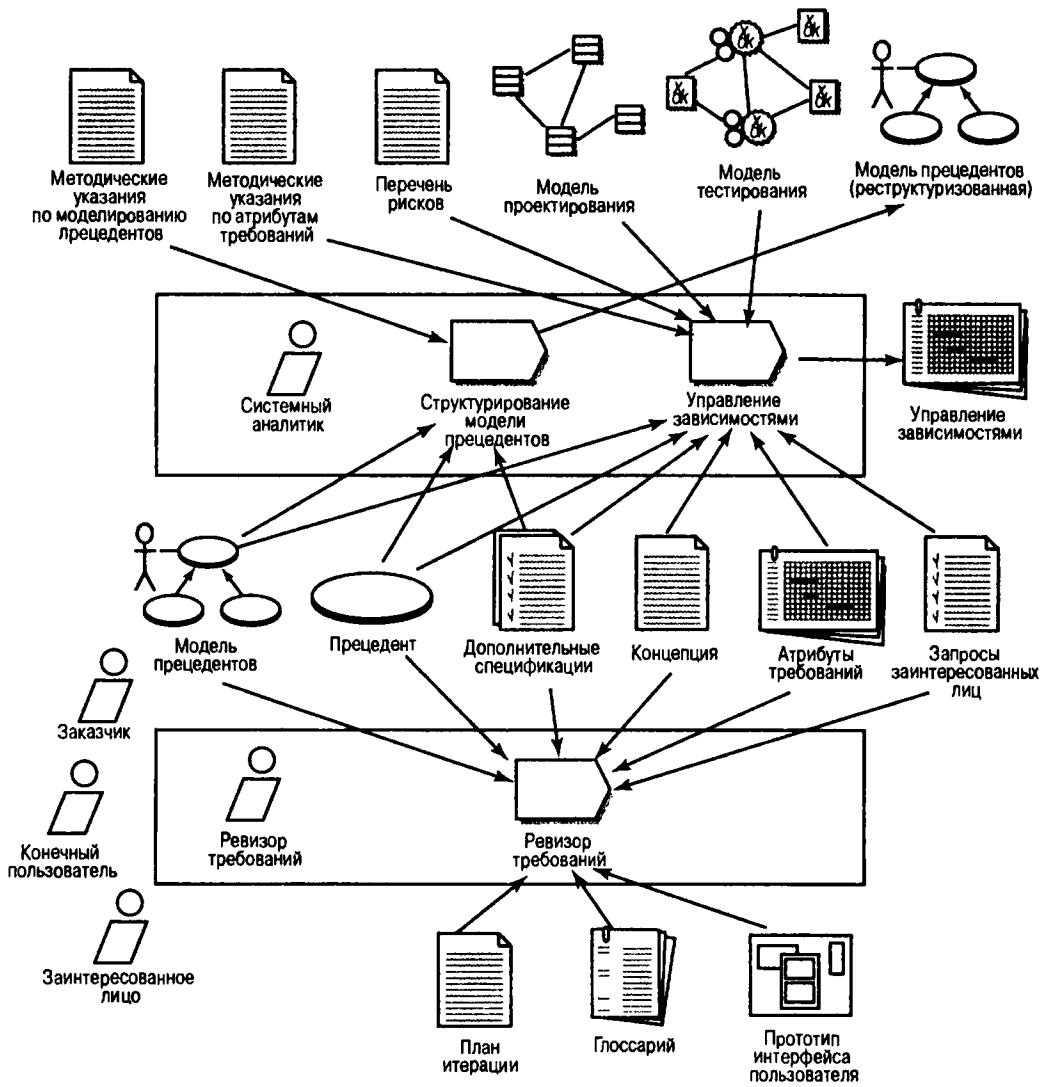


Рис. Д.8. Обработка изменений требований

Интеграция процессов

Rational Unified Process определяет потоки информации, преобразования, методические указания, эвристики и формальные трассировочные связи, которые связывают эти артефакты с другими видами деятельности по разработке программного обеспечения и другими артефактами. Например, артефакт требований в процессе может быть связан восходящим образом с бизнес-моделью (также созданной с использованием объектно-ориентированной технологии и бизнес-precedентов) и нисходящим образом с такими

артефактами, как модель анализа или модель проектирования, а также с тестовыми примерами и другой документацией (см. рис. Д.2).

Автоматические средства разработки программных систем поддерживают многие представленные в RUP принципы – от разработки требований и визуального моделирования до генерации отчетов, управления конфигурацией и автоматического тестирования. В поставляемый пакет включены также руководства по применению инструментальных средств, содержащие подробное описание правил использования вспомогательных программных средств Rational Unified Process для поддержки его отдельных этапов и видов деятельности.

Список литературы

- Boehm B. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- Boehm B. *A Spiral Model of Software Development and Enhancement*. IEEE Computer 21, 15; May, 1988, pp.61-72.
- Boehm B. and Papaccio P. *Understanding and Controlling Software Costs*. IEEE Transactions on Software Engineering 14, 10; October, 1988, pp. 1462–1473).
- Booch G., Rumbaugh J. and Jacobson I. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley Longman, 1999.
- Brooks F. *The Mythical Man Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.
- Davis A. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
- Davis A. *Software Prototyping*. In Advances in Computers, Vol. 40, pp. 39–62. Chestnut Hill, MA: Academic Press, 1995.
- Davis A. *201 Principles of Software Development*. New York: McGraw-Hill, 1995.
- Davis A. *Achieving Quality in Software Requirements*. Software Quality Professional 1, 3; June, 1999, pp. 37–44.
- Dorfman M. and Thayer R. *Standards, Guidelines, and Examples of System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1990.
- European Software Process Improvement Training Initiative. *User Survey Report*. 1995.
- FDA. *Medical Devices; Current Good Manufacturing Practice (CGMP) Final Rule; Quality System Regulation*. Federal Register 61, 195; 7 October, 1996, Subpart C, pp. 52657–52658.
- FDA/ODE. *ODE Guidance for the Content of Premarket Submission for Medical Devices Containing Software*. (Draft 1.3, 12 August, 1996).
- Fisher R., Ury W. and Patton B. *Getting to Yes: Negotiating Agreement without Giving In.*, 2nd ed. New York: Penguin Books, 1983.
- Gause D. and Weinberg G. *Exploring Requirements: Quality Before Design*. New York: Dorset House Publishing, 1989.
- Grady R. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
- Humphrey W. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989.
- IEEE. *IEEE Standards Collection, Software Engineering*. IEEE Standards Collection, Software Engineering. New York, NY: IEEE, 1994.
- International Council on Systems Engineering (INCOSE). *An Identification of Pragmatic Principles – Final Report*. INCOSE WMA Chapter, 1993. Содержится в www.incose.org/workgrps/practice.html.
- International Council on Systems Engineering (INCOSE). 1999. Содержится в www.incose.org.

- Jacobson I., Booch G. and Rumbaugh J. *The Unified Software Development Process*. Reading, MA: Addison-Wesley Longman, 1999.
- Jacobson I., Christerson M., Jonsson P. and Övergaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Harlow, Essex, England: Addison-Wesley Longman, 1992.
- Jacobson I., Ericsson M. and Jacobson A. *The Object Advantage: Business Process Reengineering with Object Technology*. Wokingham, England: Addison-Wesley, 1995.
- Jones C. *Revitalizing Software Project Management*. American Programmer 6,7; June, 1994, pp. 3–12.
- Karat C.-M. *Guaranteeing Rights for the User*. Communications of the ACM 41,12; December, 1998, p. 29.
- Kruchten P. *The 4+1 View of Architecture*. IEEE Software 12,6; November, 1995, pp. 45–50.
- Kruchten P. *The Rational Unified Process: An Introduction*. Reading, MA: Addison-Wesley Longman, 1999.
- Moore G. *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*. New York, NY: HarperCollins, 1991.
- Rational Software Corporation. *Rational Unified Process V5.1*. Cupertino, CA: Rational Software Corporation, 1999.
- Rechtin E. and Maier M. *The Art of System Architecting*. Boca Raton, FL: CRC Press, 1997.
- Royce W. *Managing the Development of Large Software Systems: Concepts and Techniques*. Proceedings of WESCON, August 1970. Также содержится в: ICSE9 Proceedings, IEEE-CS, 1987.
- Rumbaugh J., Jacobson I. and Booch G. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley Longman, 1998.
- Scharer L. *Pinpointing Requirements*. (Опубликовано в *Software Requirements Engineering*. Los Alamos, CA: IEEE Computer Society Press, 1990; Статья перепечатана из журнала Datamation, 1981.)
- Schneider G. and Winters J. *Applying Use Cases: A Practical Guide*. Reading, MA: Addison-Wesley Longman, 1998.
- SEI. *Capability Maturity Model for Software*. Version 1.1. Document No. CMU/SEI-93-TR-25, ESC-TR-93-178. Pittsburgh, PA: Carnegie-Mellon University Software Engineering Institute, 1993.
- Shaw M. and Garlan D. *Software Architecture: Perspective on an Emerging Discipline*. Upper Saddle River, NJ: Prentice-Hall, 1996.
- Snyder T. and Shumate K. *Kaizen Project Management*. American Programmer 5, 10; December, 1992, pp. 12–22.
- The Standish Group. *Charting the Seas of Information Technology – Chaos*. The Standish Group International, 1994.
- Weinberg G. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- Weinberg G. *Just Say No! Improving the Requirements Process*. American Programmer 8, 10; October, 1995, pp. 19–23.
- Wood B. and Ernes J. *Applying Hazard Analysis to Medical Devices (Part I)*. Medical Device & Diagnostic Industry Magazine 15, 1; January, 1993, pp. 79–83.
- Wood B. and Ernes J. *Applying Hazard Analysis to Medical Devices (Part II)*. Medical Device & Diagnostic Industry Magazine 15, 3; March, 1993, pp. 58–64.

Предметный указатель

C

CCB, 189
CMM, 419
Collaborations, 308

D

Delta Vision document, 181; 182; 183; 184;
349, 366

I

ISO 9000, 43, 424

M

Modern SRS Package, 254; 282

R

Rational Unified Process, 427
ROI, 297; 299, 339
RUP, 427

S

SEI, 419
SEI-CMM, 43

T

Traceability, 228; 296

U

UML, 75; 244; 287; 427
Use-case realization, 308

V

V&V, 311; 363
Validating, 298
Verification, 296
Vision document, 170, 218

A

Акторы, 66; 246; 247; 248; 276; 278
выявление, 67
Анализ
дивидендов, 297; 299, 339, 340, 356
проблемы, 360
рисков, 338; 339, 340, 356
Анкетирование, 118
Артефакты, 53; 258; 345; 428; 429
требований, 253
Архитектура
вид с точки зрения
прецедентов, 307
процессов, 307
развертывания, 307
реализации, 307
доминирующая, 306
логическое представление, 306
систем ПО, 305
4+1 представление, 306

Б

Базовый уровень требований, 197; 345;
362; 366
задание, 197; 202
управление, 212

В

Верификация, 296; 297; 311; 334; 356; 363

глубина, 336
определение, 296
покрытие, 337
Возможность обслуживания, 234; 237

Г

Границы системы-решения, 66
Группы профессиональных приемов, 21;
51

Д

Деревья решений, 281; 285
Деятельности, 428
Диаграмма
 деятельности, 281; 286
 сущность-связь, 286
Документ
 бизнес-требований, 170
 изменений концепции, 181; 182
 требований к семейству продуктов, 170
 требований маркетинга, 170; 174
Документ-концепция, 170; 177; 185; 218;
228; 247; 254; 272; 345; 349; 361; 365;
429
Дополнительные спецификации, 429

З

Заинтересованные лица, 64
 выявление, 65
 категории, 65

И

Иерархия требований, 351; 361
Изменения, 364; 423
 неофициальные, 212
 официальные, 212
 требований, 341; 435
 возмущающий эффект, 349
 скрытые, 343
Инженерия программного обеспечения,
422
Интервьюирование, 111
История изменений, 354
 требований, 435

уровни детализации, 354
Итеративный подход, 217; 271; 326; 366;
427
Итерация, 218

К

Конечный автомат, 281; 283
диаграмма перехода состояний, 283
матрица перехода состояний, 283
Контекстно-свободный вопрос, 112
Контрольный журнал изменений, 354
Кооперация
 поведенческая часть, 308
Кооперация, 308; 310
 структурная часть, 308

Л

Лидер проекта, 346

М

Масштаб, 362; 366
 достижимый, 194
 сокращение, 202
Матрица трассировки, 275; 317; 330; 331
Метод
 анализа корневых причин, 62
 прецедентов, 149; 151; 243; 244
Моделирование, 305
 бизнес-процесса, 74; 360
 методы, 74
 цель, 74
Модель
 бизнес-процесса, 77
 объектно-ориентированная, 281; 287
 объектов бизнес-процесса, 76
 прецедентов, 259; 276; 345; 429
 бизнес-процесса, 76
 системы, 150
 проектирования, 310
 процесса разработки ПО
 водопада, 214; 215
 спиральная, 216
 системы, 77
 сущность-связь, 281; 286
Мозговой штурм, 129

генерация идей, 129
отбор идей, 129
правила проведения, 130

H

- Набор требований
 - к программному обеспечению, 227
 - модифицируемый, 274
 - непротиворечивый, 271
 - полный, 269
- Надежность, 234; 236
- Накопительное голосование, 133
- Независимые ревизии, 336
- Неоднозначность, 263
 - методы устранения, 265

Q

- Область
 - проблемы, 44; 361
 - решения, 45; 361
- Объективно-ориентированный подход
 - реализация, 303
- Объем V&V-действий, 336; 340
- Ограничения
 - выявление, 68
 - источники, 68
 - проектирования, 231; 238; 239, 270; 333
 - источники, 238
- Организационная схема Modern SRS Package, 257
- Отношения трассировки, 314; 352
- Оценка трудоемкости, 199

III

- Пакет Modern SRS Package, 258; 291; 363; 366
- глоссарий, 280
- индекс, 279
- история исправлений, 280
- критерии качества, 268; 278
- оглавление, 279

Пирамида требований, 166; 192; 224; 349

План управления изменениями, 344

Потребности пользователей, 45; 105

Практичность, 234

Прецедент, 46; 149, 150, 153, 276, 277;
304, 330, 356, 366, 367, 416
UML-нотация, 246
имя, 248, 277
определение, 246
постусловия, 251
поток событий, 249, 277
альтернативный, 250, 417
основной, 250, 416
предусловия, 251
специальные требования, 417
спецификация, 151

Приемо-сдаочные испытания 326

- Приложение
- IS/IT, 43; 70; 73
- ISV, 43; 70
- встроенные, 43; 70; 360
- Приоритет функций
- задание, 198
- Проблема, 60
- анализ, 59
- пять этапов анализа, 60
- стандартная форма постановки, 61
- Проверка правильности, 298; 311; 325;
334; 356; 363

- Проектирование, 232
- Прозрачное тестирование, 336
- Производительность, 234; 237
- Просмотр, 336
- Прототип, 214; 216; 218; 221; 271
 - архитектурный, 159
 - виды, 159
 - отбрасываемый, 159
 - требований, 160
 - эволюционирующий, 160

Процесс декомпозиции сложных систем, 83
разработки программного обеспечения, 213; 427
управления изменениями, 344
Псевдокод, 281; 282

P

Рабочие процессы, 219; 428; 429
Разработка требований -
подпроцессы, 430

рабочий процесс, 429
 Раскадровка, 141; 271; 276
 активная, 142
 интерактивная, 142
 пассивная, 142

Реализация прецедента, 308

Риск
 оценка, 200

C

Синдром, 155; 217

Система контроля изменений, 347

Системная инженерия (системное проектирование), 81; 360

Сквозной контроль, 336

Совет по контролю за изменениями, 189; 346; 347; 362; 366

Совещание, посвященное требованиям, 121

 ведущий, 125

 подготовка, 122

Совещание, посвященные требованиям
 ведущий, 126

Сотрудники, 428

Спецификация требований
 к системе, 170

Спецификация требований к
 программному обеспечению, 170
 системе, 169

Схемы потоков данных, 281; 288

T

Тестирование
 методологии, 326

Тестирование
 черного ящика, 336

Трассировка, 275; 296; 308; 312; 327; 330;
 337; 349; 356; 363; 365; 366; 423; 435

неявная, 315

обратная, 275

определение, 313

прямая, 275

явная, 314

Требования

 атрибуты, 272

 важность, 272

стабильность, 272
 будущие, 174
 верифицируемые, 273
 выявление, 42; 232; 361
 документирование, 42
 дочерние, 241; 275
 к интерфейсам, 85; 171
 к подсистемам, 85; 171
 к программному обеспечению, 41; 46;
 108; 226; 228
 категории, 227

атрибуты системной среды, 227

атрибуты системы, 227

вводы системы, 227

выводы системы, 227

функции системы, 227

качество спецификации, 267

корректные, 268

недвусмысленные, 269

нефункциональные, 234; 244; 270

ограничения проектирования, 233

организация, 42

понимаемые, 275

проблема ортогональности, 302; 304;
 307

производные, 84

родительские, 241

трассируемые, 274

управление, 42

формальные средства спецификации,
 281

функциональные, 233; 234; 270

У

Управление

изменениями, 212; 352; 366

конфигурацией, 350

требованиями, 360; 421; 422

 интегративный подход, 233

Уровень

детализации, 198

зрелости СММ, 419

конкретизации, 226; 233; 241; 247; 261;
 263; 336; 337

Ф

Фазы жизненного цикла, 218

внедрение, 218

исследование, 218; 345

начало, 218

построение, 218

Фазы жизненного цикла ПО, 431

Функции, 46; 106; 194; 228; 365

атрибут, 108

назначение, 109

обоснование, 109

приоритет, 109

риск, 109

стабильность, 109

статус, 109

трудоемкость, 109

целевая версия, 109

будущие, 182; 203

важная, 134; 201

количество, 108

критическая, 134; 201; 202

описание, 107

полезная, 134; 201

Э

Элементы

моделирования

Rational Unified

Process, 428

поведения, 143

Научно-популярное издание

Дин Леффингуэлл, Дон Уидриг

**Принципы работы с требованиями
к программному обеспечению.
Унифицированный подход**

Литературный редактор *Е.Д. Даудян*

Верстка *О.В. Линник*

Художественный редактор *В.Г. Паавютин*

Корректор *Л.В. Коровкина*

Издательский дом “Вильямс”.

101509, Москва, ул. Лесная, д. 43, стр. 1.

Изд. лиц. ЛР № 090230 от 23.06.99

Госкомитета РФ по печати.

Подписано в печать 17.04.2002. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 36,0. Уч.-изд. л. 25,0.

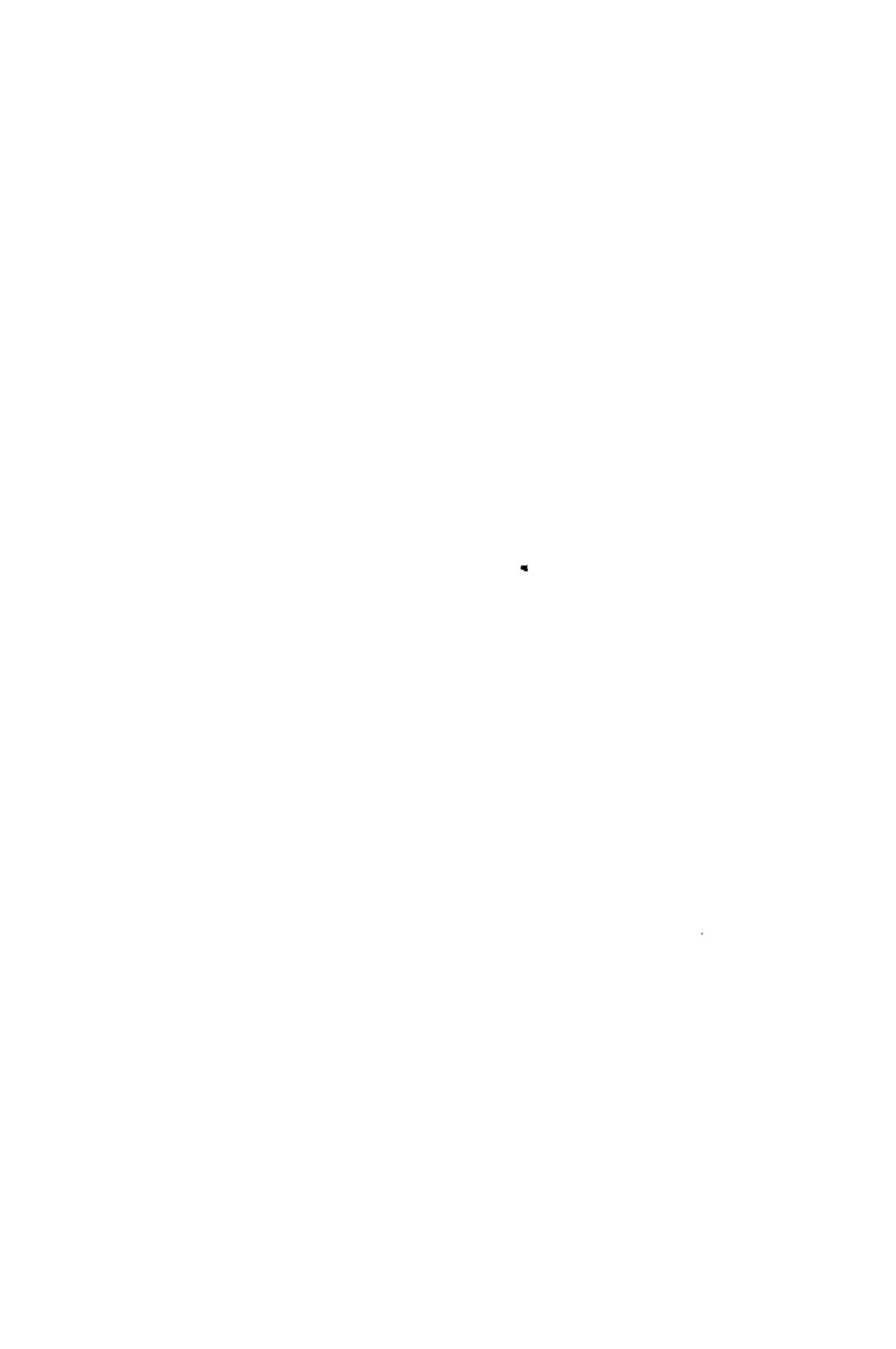
Тираж 3500 экз. Заказ № 376.

Отпечатано с диапозитивов в ФГУП “Печатный двор”

Министерства РФ по делам печати,

телерадиовещания и средств массовых коммуникаций.

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



“Универсальное решение разнообразных проблем требований, с которыми сталкивается любая команда разработчиков. Содержит поучительные для всех разработчиков идеи и откровения.”

— Айвар Джейкобсон

“Многие проекты потерпели неудачу по той простой причине, что разработчики не сумели создать то, что нужно. Опираясь на богатый опыт, Дин и Дон демонстрируют, как организовать мощный индустриальный процесс работы с требованиями, позволяющий удостовериться, что вы создаете то, что требуется. Этую книгу следует прочитать всем без исключения разработчикам приложений.”

— Грейди Буч

Несмотря на богатый опыт разработки и наличие современных инструментальных средств, значительное число проектов создания ПО все еще оканчивается неудачей. Чаще всего из-за того, что в начале проекта требования к ПО определяются и формулируются неправильно, или же неверно обрабатываются во время реализации проекта. В книге исследуются причины неудач и предлагается практический и надежный подход к получению требований, удовлетворяющих заказчика, с соблюдением рамок отведенного на проект времени и бюджета.

Книга включает полномасштабный рабочий пример и имеет неформальный, доступный стиль изложения. Используя свой опыт, авторы показывают, как можно эффективно формулировать требования с помощью методов прецедентов и более традиционных форм выражения требований. В книге описываются методы определения, реализации, верификации и проверки достоверности требований, а также шесть важнейших процессов разработки, которыми профессионально должна овладеть команда для успешного управления требованиями на протяжении жизненного цикла проекта. Особое внимание здесь уделяется проблеме обработки возникающих изменений, а также описанию процесса, гарантирующего, что масштаб проекта задан правильно и согласован со всеми заинтересованными лицами.

Среди основных рассматриваемых тем выделим следующие.

- Пять этапов анализа проблемы
- Моделирование бизнес-процессов и системное проектирование
- Методы выявления требований клиентов, пользователей, разработчиков и других участников
- Применение и уточнение прецедентов
- Создание прототипов
- Организация информации о требованиях и ее обработка
- Задание масштаба проекта и работа с заказчиками
- Переход от требований к реализации
- Верификация и проверка правильности работы системы
- Обработка изменений

Дин Леффингуэлл является вице-президентом компании Rational Software и генеральным директором Rational University, где ведет исследования в области методологии Rational Unified Process™ и проводит обучение заказчиков. Ранее, работая в компании Requisite, Inc., он создал очень удачное программное средство управления требованиями RequisitePro™, а позднее разработал основы популярной серии курсов компании Rational Software по управлению требованиями под названием Requirements College™.

Дон Уидриг — независимый технический писатель и консультант. Он разработал и преподает курс обучения работе с инструментальным средством RequisitePro компании Rational Software.

Более подробную информацию о серии Object Technology можно найти по адресу: <http://www.aw.com/cseng/otseries>

ISBN 5-8459- 0275-4



Издательский дом “Вильямс”
www.williamspublishing.com



Addison-Wesley
www.aw.com

0 2045

9 785845 902757