МИНИСТЕРСТВО РАЗВИТИЯ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

Ташкентский университет информационных технологий

ВВЕДЕНИЕ В ПРОГРАММНЫЙ ИНЖИНИРИНГ

Учебник

Ташкент – 2017

Авторы: К.Ф.Керимов, Ш.Ш.Мухсинов. "Введение в программный инжиниринг". Учебник /ТУИТ. 418 с. Ташкент, 2017

Учебная дисциплина "Введение в программный инжиниринг" является одной из специальных дисциплин в учебной программе подготовки бакалавров по направлению образования "5330600 – Программный инжиниринг". Она создает теоретическую базу для изложения и понимания последующих специальных дисциплин учебного плана подготовки бакалавров данного направления.
Основная цель учебного пособия - представить программную инженерию в виде целостного изложения, концентрируясь на концепции процесса, различных методологиях разработки программного обеспечения, отдельных видах деятельности процесса - разработке архитектуры, конфигурационном управлении, работе с требованиями, тестировании.

Напечатано на основе утверждения учебно-методическим советом Ташкентского Университета Информационных Технологий

Рецензенты:

Рахманов А.Т. – ТУИТ, кафедра «Системное и прикладное программирование», к.т.н., доцент

Худайбердиев М.Х. – Центр разработки программных продуктов и аппаратно – программных комплексов при ТУИТ, к.т.н., директор

Ташкентский Университет Информационных Технологий, 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ ... 6
ГЛАВА 1. О ПРЕДМЕТЕ ИЗУЧЕНИЯ ... 7
1.1. Программная инженерия ... 7
1.2. Программное обеспечение ... 10
ГЛАВА 2. ПРОЦЕСС РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 13
2.1. Процесс создания программного обеспечения ... 13
2.2. Совершенствование процесса ... 15
2.3. Классические модели процесса ... 18
ГЛАВА 3. УПРАВЛЕНИЕ ПРОЕКТАМИ ... 25
3.1. Рабочий продукт ... 25
3.2. Дисциплина обязательств ... 28
3.3. Проект и управление проектами ... 30
ГЛАВА 4. УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ ... 33
4.1. Понятия о требованиях ... 33
4.2. Виды и свойства требований ... 36
4.3. Варианты формализации требований ... 38
4.4. Некоторые ошибки при документировании требований. ... 40
4.5. Цикл работы с требованиями ... 40
ГЛАВА 5. СИСТЕМНОЕ МОДЕЛИРОВАНИЕ ... 43
5.1. Понятие о модели ... 43
5.2. Модели системного окружения ... 45
5.3. Поведенческие модели ... 48
5.4. Модели потоков данных ... 49
5.5. Модели конечных автоматов ... 51
5.6. Модели данных ... 55
5.7. Объектные модели ... 58
5.8. Инструментальные CASE-средства ... 61
ГЛАВА 6. ПРОТОТИПИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ ... 66
6.1. Прототип программного обеспечения ... 66
6.2. Прототипирование в процессе разработки ПО ... 69
6.3. Эволюционное прототипирование ... 72
6.4. Экспериментальное прототипирование ... 76
6.5. Технологии быстрого прототипирования ... 79
6.6. Применение динамических языков высокого уровня ... 80
6.7. Программирование баз данных ... 82
6.8. Сборка приложений с повторным использованием компонентов ... 86
6.9. Прототипирование пользовательских интерфейсов ... 90
ГЛАВА 7. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ ... 95
7.1. Структурирование системы ... 99
7.2. Модель репозитория ... 101
7.3. Модель клиент/сервер ... 103
7.4. Модель абстрактной машины ... 105
7.5. Модели управления ... 107
7.6. Централизованное управление ... 108
ГЛАВА 8. АРХИТЕКТУРА РАСПРЕДЕЛЕННЫХ СИСТЕМ ... 112
8.1. Многопроцессорная архитектура ... 118
8.2. Архитектура клиент/сервер ... 119
8.3. Архитектура распределенных объектов ... 126

8.4. CORBA....131
ГЛАВА 9. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ 140
9.1. Объекты и классы объектов....142
9.2. Параллельные объекты....147
9.3. Процесс объектно-ориентированного проектирования....150
9.4. Проектирование архитектуры....156
9.5. Определение объектов....157
ГЛАВА 10. ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ 163
10.1. Проектирование систем....166
10.2. Моделирование систем реального времени....168
10.3. Программирование систем реального времени....170
10.4. Управляющие программы....172
10.5. Управление процессами....175
10.6. Системы наблюдения и управления....178
10.7.Системы сбора данных....185
ГЛАВА 11. ПРОЕКТИРОВАНИЕ С ПОВТОРНЫМ ИСПОЛЬЗОВАНИЕМ КОМПОНЕНТОВ....192
11.1. Покомпонентная разработка....198
11.2. Объектные структуры приложений....203
11.3. Повторное использование коммерческих программных продуктов....205
11.4. Разработка повторно используемых компонентов....209
11.5. Семейства приложений....211
11.6. Проектные паттерны....216
ГЛАВА 12. ПРОЕКТИРОВАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ. 223
12.1. Принципы проектирования интерфейсов пользователя....226
12.2. Взаимодействие с пользователем....230
12.3. Представление информации....233
12.4. Использование в интерфейсах цвета....239
12.5. Средства поддержки пользователя....241
12.6. Сообщения об ошибках....242
12.7. Проектирование справочной системы....244
12.8. Документация пользователя....247
12.9. Оценивание интерфейса....249
ГЛАВА 13. НАДЕЖНОСТЬ СИСТЕМ....254
13.1. Критические системы....257
13.2. Системы, критические по обеспечению безопасности....259
13.3. Работоспособность и безотказность....262
13.4. Безопасность....268
13.5. Защищенность....273
ГЛАВА 14. ВЕРИФИКАЦИЯ И АТТЕСТАЦИЯ ПО....278
14.1. Планирование верификации и аттестации....284
14.2. Инспектирование программных систем....287
14.3. Инспектирование программ....289
14.4. Автоматический статический анализ программ....296
14.5. Метод "чистая комната"....301
ГЛАВА 15. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ..307
15.1. Тестирование дефектов....309
15.2. Тестирование методом черного ящика....311
15.3. Области эквивалентности....312

15.4. Структурное тестирование....317
15.5. Тестирование сборки....323
15.6. Нисходящее и восходящее тестирование....325
15.7. Тестирование интерфейсов....328
15.8. Инструментальные средства тестирования....332
ГЛАВА 16. УПРАВЛЕНИЕ ПЕРСОНАЛОМ....337
16.1. Пределы мышления....337
Оценка стоимости программного продукта....338
16.2. Производительность....341
16.3. Методы оценивания....349
16.4. Алгоритмическое моделирование стоимости....354
16.5. Обеспечение качества и стандарты....362
16.6. Стандарты на техническую документацию....367
ГЛАВА 17. МОДЕРНИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 372
17.1. Динамика развития программ....374
17.2. Сопровождение программного обеспечения....377
17.3. Процесс сопровождения....383
17.4. Прогнозирование сопровождения....386
17.5. Реинжениринг программного обеспечения....389
ГЛАВА 18. УПРАВЛЕНИЕ КОНФИГУРАЦИЯМИ....397
18.1. Планирование управления конфигурацией....400
18.2. Определение конфигурационных объектов....401
18.3. База данных конфигураций....403
18.4. Управление изменениями....404
18.5. Управление версиями и выпусками....409
18.6. Идентификация версий....410
18.7. Нумерация версий....411
18.8. Сборка системы....412
ЛИТЕРАТУРА....416

ВВЕДЕНИЕ

Цель данного курса - представить программную инженерию в виде целостного изложения, концентрируясь на концепции процесса, различных методологиях разработки ПО, отдельных видах деятельности процесса - разработке архитектуры, конфигурационном управлении, работе с требованиями, тестировании. В стороне умышленно оставлены вопросы, собственно, программирования, поскольку в рамках общего курса их невозможно эффективно рассмотреть.

В пособии систематически изложены методы программирования, их теория и практика с учетом ядра знаний SWEBOK (SoftWare Engineering of Body Knowledge) и положений стандартов программной инженерии. Представлены методы прикладного и теоретического проектирования, методы доказательства, верификация и тестирование, а также методы интеграции и преобразования программ и данных. Определены основы инженерной дисциплины разработки – управление проектом, риском и качеством. Описана инженерия приложений и предметной области на основе повторного использования компонентов, определены подходы и методы их аннотации для накопления, выбора и оценки применимости в новых программных проектах.

Для студентов факультета программной инженериии, разработчиков и докторантов в области программирования, желающих ознакомиться с систематизированными знаниями по современным методам анализа, проектирования, интеграции и тестирования, а также по методам инженерии программирования – управление проектом, рисками и качеством проектируемых систем.

ГЛАВА 1. О ПРЕДМЕТЕ ИЗУЧЕНИЯ

1.1. Программная инженерия

Чем программирование отличается от программной инженерии? Тем, что первое является некоторой абстрактной деятельностью и может происходить во многих различных контекстах. Можно программировать для удовольствия, для того, чтобы научиться (например, на уроках, на семинарах в университете), можно программировать в рамках научных разработок. А можно заниматься промышленным программированием. Как правило, это происходит в команде, и совершенно точно – для заказчика, который платит за работу деньги. При этом необходимо точно понимать, что нужно заказчику, выполнить работу в определенные сроки и результат должен быть нужного качества – того, которое удовлетворит заказчика и за которое он заплатит. Чтобы удовлетворить этим дополнительным требованиям, программирование "обрастает" различными дополнительными видами деятельности: разработкой требований, планированием, тестированием, конфигурационным управлением, проектным менеджментом, созданием различной документации (проектной, пользовательской и пр.).

Разработка программного кода предваряется анализом и проектированием (первое означает создание функциональной модели будущей системы без учета реализации, для осознания программистами требований и ожиданий заказчика; второе означает предварительный макет, эскиз, план системы на бумаге). Трудозатраты на анализ и проектирование, а также форма представления их результатов сильно варьируются от видов проектов и предпочтений разработчиков и заказчиков.

Требуются также специальные усилия по организации процесса разработки. В общем виде это итеративно-инкрементальная модель, когда требуемая функциональность создается порциями, которые менеджеры и заказчик могут оценить, и тем самым есть возможность управления ходом

разработки. Однако эта общая модель имеет множество модификаций и вариантов.

Разработку системы также необходимо выполнять с учетом удобств ее дальнейшего сопровождения, повторного использования и интеграции с другими системами. Это значит, что система разбивается на компоненты, удобные в разработке, годные для повторного использования и интеграции. А также имеющие необходимые характеристики по быстродействию. Для этих компонент тщательно прорабатываются интерфейсы. Сама же система документируется на многих уровнях, создаются правила оформления программного кода – то есть оставляются многочисленные семантические следы, помогающие создать и сохранить, поддерживать единую, стройную архитектуру, единообразный стиль, порядок…

Все эти и другие дополнительные виды деятельности, выполняемые в процессе промышленного программирования и необходимые для успешного выполнения заказов и будем называть **программной инженерией** (software engineering). Получается, что так мы обозначаем, во-первых, некоторую практическую деятельность, а во-вторых, специальную **область знания**. Или другими словами, научную дисциплину. Ведь для облегчения выполнения каждого отдельного проекта, для возможности использовать разнообразный положительный опыт, достигнутый другими командами и разработчиками, этот самый опыт подвергается осмыслению, обобщению и надлежащему оформлению. Так появляются различные методы и практики (best practices) – тестирования, проектирования, работы над требованиями и пр., архитектурных шаблонов и пр. А также стандарты и методологии, касающиеся всего процесса в целом (например, MSF, RUP, CMMI, Scrum). Вот эти-то обобщения и входят в программную инженерию как в область знания.

Необходимость в программной инженерии как в специальной области знаний была осознана мировым сообществом в конце 60-х годов прошлого века, более чем на 20 лет позже рождения самого программирования, если

считать таковым знаменитый отчет фон Неймана "First Draft of a Report on the EDVAC", обнародованный им в 1945 году. Рождением программной инженерии является 1968 год – конференция NATO Software Engineering, г. Гармиш (ФРГ), которая целиком была посвящена рассмотрению этих вопросов. В сферу программной инженерии попадают все вопросы и темы, связанные с организацией и улучшением процесса разработки ПО, управлением коллективом разработчиков, разработкой и внедрением программных средств поддержки жизненного цикла разработки ПО. Программная инженерия использует достижения информатики, тесно связана с системотехникой, часто предваряется бизнес-реинжинирингом. Немного подробнее об этом контексте программной инженерии.

Информатика (computer science) – это свод теоретических наук, основанных на математике и посвященных формальным основам вычислимости. Сюда относят математическую логику, теорию грамматик, методы построения компиляторов, математические формальные методы, используемые в верификации и модельном тестировании и т.д. Трудно строго отделить программную инженерию от информатики, но в целом направленность этих дисциплин различна. Программная инженерия нацелена на решение проблем производства, информатика – на разработку формальных, математизированных подходов к программированию.

Системотехника (system engineering) объединяет различные инженерные дисциплины по разработке всевозможных искусственных систем – энергоустановок, телекоммуникационных систем, встроенных систем реального времени и т.д. Очень часто ПО оказывается частью таких систем, выполняя задачу управления соответствующего оборудования. Такие системы называются *программно-аппаратными*, и участвуя в их создании, программисты вынуждены глубоко разбираться в особенностях соответствующей аппаратуры.

Бизнес-реинжиниринг (business reengineering) – в широком смысле обозначает модернизацию бизнеса в определенной компании, внедрение

новых практик, поддерживаемых соответствующими новыми информационными системами. При этом акцент может быть как на внутреннем переустройстве компании так и на разработке нового клиентского сервиса (как правило, эти вопросы взаимосвязаны). Бизнес-реинжиниринг часто предваряет разработку и внедрение информационных систем на предприятии, так как требуется сначала навести определенный порядок в делопроизводстве, а лишь потом закрепить его информационной системой.

Связь программной инженерии (как области практической деятельности) с информатикой, системотехникой и бизнес-реинжинирингом показана на рис. 1.1.

Рис. 1.1. Связь программной инженерии

1.2. Программное обеспечение

Определение. Будем понимать под **программным обеспечением** (ПО) множество развивающихся во времени логических предписаний, с помощью которых некоторый коллектив людей управляет и использует многопроцессорную и распределенную систему вычислительных устройств.

Это определение, данное Харальдом Милсом, известным специалистом в области программной инженерии из компании IBM, заключает в себе следующее.

1. Логические предписания – это не только сами программы, но и различная документация (например, по эксплуатации программ) и шире – определенная система отношений между людьми, использующими эти программы в рамках некоторого процесса деятельности.

2. Современное ПО предназначено, как правило, для одновременной работы со многими пользователями, которые могут быть значительно удалены друг от друга в физическом пространстве. Таким образом, вычислительная среда (персональные компьютеры, сервера и т.д.), в которой ПО функционирует, оказывается распределенной.

3. Задачи решаемые современным ПО, часто требуют различных вычислительных ресурсов в силу различной специализации этих задач, из-за большого объема выполняемой работы, а также из соображений безопасности. Например, появляется сервер базы данных, сервер приложений и пр. Таким образом, вычислительная среда, в которой ПО функционирует, оказывается многопроцессорной.

4. ПО развивается во времени – исправляются ошибки, добавляются новые функции, выпускаются новые версии, меняется его аппаратная база.

Свойства. Таким образом, ПО является сложной динамической системой, включающей в себя технические, психологические и социальные аспекты. ПО заметно отличается от других видов систем, создаваемых (созданных) человеком – механических, социальных, научных и пр., и имеет следующие особенности, выделенные Фредериком Бруксом в его знаменитой статье "Серебряной пули нет".

1. Сложность программных объектов, которая существенно зависит от их размеров. Как правило, большее ПО (большее количество пользователей, больший объем обрабатываемых данных, более жесткие требования по быстродействию и пр.) с аналогичной функциональностью – это другое ПО. Классическая наука строила простые модели сложных явлений, и это удавалось, так как сложность не была характеристической чертой рассматриваемых явлений. (Сравнение программирования именно с

наукой, а не с театром, кинематографом, спортом и другими областями человеческой деятельности, оправдано, поскольку оно возникло, главным образом, из математики, а первые его плоды – программы – предназначались для использования при научных расчетах. Кроме того, большинство программистов имеют естественнонаучное, математическое или техническое образование. Таким образом, парадигмы научного мышления широко используются при программировании – явно или неявно.)

2. Согласованность – ПО основывается не на объективных посылках (подобно тому, как различные системы в классической науке основываются на постулатах и аксиомах), а должно быть согласовано с большим количеством интерфейсов, с которыми впоследствии оно должно взаимодействовать. Эти интерфейсы плохо поддаются стандартизации, поскольку основываются на многочисленных и плохо формализуемых человеческих соглашениях.

3. Изменяемость – ПО легко изменить и, как следствие, требования к нему постоянно меняются в процессе разработки. Это создает много дополнительных трудностей при его разработке и эволюции.

4. Нематериальность – ПО невозможно увидеть, оно виртуально. Поэтому, например, трудно воспользоваться технологиями, основанными на предварительном создании чертежей, успешно используемыми в других промышленных областях (например, в строительстве, машиностроении). Там на чертежах в схематичном виде воспроизводятся геометрические формы создаваемых объектов. Когда объект создан, эти формы можно увидеть. А на чем мы основываемся, когда изображаем ПО?

Контрольные вопросы

1. Что такое программная инженерия?
2. Что такое информатика?
3. Чем программирование отличается от программной инженерии?
4. Как связаны программная инженерия, информатика, системотехника и бизнес-инжиниринг?
5. Определение программного обеспечения(ПО).

6. Перечислите свойства программного обеспечения.

Ключевые слова: *реинжиниринг, программное обеспечение, логические предписания, согласованность, изменяемость, нематериальность.*

The keywords: *reengineering, software, logical prescriptions, consensus, alterability,* immateriality.

Kalit so'zlar: *reinjiniring, dasturiy ta'minot, mantiqiy ko'rsatmalar, muvofiqlik, o'zgaruvchanlik, moddiymaslik.*

Упражнения

1. Приведите пример анализа.
2. Приведите пример проектирования.
3. Почему нужна программный инженерия?
4. Перечислите ПО, которые вы знаете, по определению Харальда Милса.
5. Какие именно области схемотехники взаимодействуют с программной инженерией?

ГЛАВА 2. ПРОЦЕСС РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Процесс создания программного обеспечения

Как мы работаем, какова последовательность наших шагов, каковы нормы и правила в поведении и работе, каков регламент отношений между членами команды, как проект взаимодействует с внешним миром и т.д.? Все это вместе мы склонны называть процессом. Его осознание, выстраивание и улучшение - основа любой эффективной групповой деятельности. Поэтому не случайно, что процесс оказался одним из основных понятий программной инженерии.

Центральным объектом изучения программной инженерии является **процесс** создания ПО – множество различных видов деятельности, методов, методик и шагов, используемых для разработки и эволюции ПО и связанных с ним продуктов (проектных планов, документации, программного кода, тестов, пользовательской документации и пр.).

Однако на сегодняшний день не существует **универсального процесса** разработки ПО – набора методик, правил и предписаний, подходящих для ПО любого вида, для любых компаний, для команд любой национальности. Каждый **текущий процесс** разработки, осуществляемый некоторой командой в рамках определенного проекта, имеет большое количество особенностей и индивидуальностей. Однако целесообразно перед началом проекта спланировать процесс работы, определив роли и обязанности в команде, рабочие продукты (промежуточные и финальные), порядок участия в их разработке членов команды и т.д. Будем называть это предварительное описание **конкретным процессом**, отличая его от плана работ, проектных спецификаций и пр. Например, в системе Microsoft Visual Team System оказывается шаблон процесса, создаваемый или адаптируемый (в случае использования стандартного) перед началом разработки. В VSTS существуют заготовки для конкретных процессов на базе CMMI, Scrum и др.

В рамках компании возможна и полезна стандартизация всех текущих процессов, которую будем называть **стандартным процессом**. Последний, таким образом, оказывается некоторой базой данных, содержащей следующее:

- информацию, правила использования, документацию и инсталляционные пакеты средств разработки, используемых в проектах компании (систем версионного контроля, средств контроля ошибок, средств программирования – различных IDE, СУБД и т.д.);
- описание практик разработки – проектного менеджмента, правил работы с заказчиком и т.д.;
- шаблоны проектных документов – технических заданий, проектных спецификаций, планов тестирования и т.д. и пр.

Также возможна стандартизация процедуры разработки конкретного процесса как "вырезки" из стандартного. Основная идея стандартного процесса – курсирование внутри компании передового опыта, а также унификация средств разработки. Очень уж часто в компаниях различные

департаменты и проекты сильно отличаются по зрелости процесса разработки, а также затруднено повторное использование передового опыта. Кроме того, случается, что компания использует несколько средств параллельных инструментов разработки, например, СУБД средства версионного контроля. Иногда это бывает оправдано (например, таковы требования заказчика), часто это необходимо – например, Java, .NET (большая компетентность оффшорной компании позволяет ей брать более широкий спектр заказов). Но очень часто это произвольный выбор самих разработчиков. В любом случае, такая множественность существенно затрудняет миграцию специалистов из проекта в проект, использование результатов одного проекта в другом и т.д. Однако при организации стандартного процесса необходимо следить, чтобы стандартный процесс не оказался всего лишь формальным, бюрократическим аппаратом. Понятие стандартного процесса введено и подробно описано в подходе CMMI.

Необходимо отметить, что наличие стандартного процесса свидетельствует о наличии "единой воли" в организации, существующей именно на уровне процесса. На уровне продаж, бухгалтерии и др. привычных для всех компаний процессов и активов единство осуществить не трудно. А вот на уровне процессов разработки очень часто каждый проект оказывается сам по себе (особенно в оффшорных проектах) – "текучка" захватывает и изолирует проекты друг от друга очень прочно.

2.2. Совершенствование процесса

Определение. Совершенствование процесса (software process improvement) – это деятельность по изменению существующего процесса (как текущего, в рамках одного проекта, так и стандартного, для всей компании) с целью улучшения качества создаваемых продуктов и/или снижения цены и времени их разработки. Причины актуальности этой деятельности для компаний-производителей ПО заключается в следующем.

1. Происходит быстрая смена технологий разработки ПО, требуются изучение и внедрение новых средств разработки.

2. Наблюдается быстрый рост компаний и их выход на новые рынки, что требует новой организации работ.

3. Имеет место высокая конкуренция, которая требует поиска более эффективных, более экономичных способов разработки.

4. Что и каким образом можно улучшать.

5. Переход на новые средства разработки, языки программирования и т.д.

6. Улучшение отдельных управленческих и инженерных практик – тестирования, управления требованиями и пр.

7. Полная, комплексная перестройка всех процессов в проекте, департаменте, компании (в соответствии, например, с CMMI).

8. Сертификация компании (CMM/CMMI, ISO 9000 и пр.).

Мы отделили п. 3 от п. 4 потому, что на практике 4 далеко не всегда означает действительную созидательную работу по улучшению процессов разработки ПО, а часто сводится к поддержанию соответствующего документооборота, необходимого для получения сертификации. Сертификат потом используется как средство, козырь в борьбе за заказы.

Главная трудность реального совершенствования процессов в компании заключается в том, что она при этом должна работать и создавать ПО, ее нельзя "закрыть на учет".

Отсюда вытекает идея непрерывного улучшения процесса, так сказать, малыми порциями, чтобы не так болезненно. Это тем более разумно, что новые технологии разработки, появляющиеся на рынке, а также развитие уже существующих нужно постоянно отслеживать. Эта стратегия, в частности, отражена в стандарте совершенствования процессов разработки CMMI.

Pull/Push стратегии. В контексте внедрения инноваций в производственные процессы бизнес-компаний (не обязательно компаний по созданию ПО) существуют две следующие парадигмы.

1. Organization pull – инновации нацелены на решение конкретных проблем компании.

2. Technology push – широкомасштабное внедрение инноваций из стратегических соображений. Вместо конкретных проблем, которые будут решены после внедрения инновации, в этом случае рассматриваются показатели компании (эффективность, производительность, годовой оборот средств, увеличение стоимости акций публичной компании), которые будут увеличены, улучшены после внедрения инновации. При этом предполагается, что будут автоматически решены многочисленные частные проблемы организации, в том числе и те, о которых в данный момент ничего не известно.

Пример использования стратегии organization pull – внедрение новых средств тестирования в ситуации, когда высоки требования по качеству в проекте, либо когда качество программной системы не удовлетворяет заказчика.

Пример использования стратегии technology push – переход компании со средств структурной разработки на объектно-ориентрованные. Еще один пример использования той же стратегии – внедрение стандартов качества ISO 9000 или CMMI. В обоих этих случаях компания не решает какую-то одну проблему или ряд проблем – она хочет радикально изменить ситуацию, выйти на новые рубежи и т.д.

Проблемы применения стратегии technology push в том, что требуется глобальная перестройка процесса. Но компанию нельзя "закрыть на реконструкцию" – за это время положение на рынке может оказаться занято конкурентами, акции компании могут упасть и т.д. Таким образом, внедрение инноваций, как правило, происходит параллельно с обычной деятельностью компании, поэтапно, что в случае с technology push сопряжено с большими трудностями и рисками.

Использование стратегии organization pull менее рискованно, вносимые ею изменения в процесс менее глобальны, более локальны. Но и выгод такие

инновации приносят меньше, по сравнению с удачными внедрениями в соответствии со стратегией technology push.

Необходимо также отметить, что существуют проблемы, которые невозможно устранить точечными переделками процесса, то есть необходимо применять стратегию technology push. Приведем в качестве примера зашедший в тупик процесс сопровождения и развития семейства программных продуктов – компания терпит большие убытки, сопровождая уже поставленные продукты, инструментальные средства проекта безнадежно устарели и находятся в плачевном состоянии, менеджмент расстроен, все попытки руководства изменить процесс наталкиваются на непонимание коллектива, ссоры и конфликты. Возможно, что в таком случае без "революции" не обойтись.

Еще одно различие обеих стратегий: в случае с organization pull, как правило, возврат инвестиций от внедрения происходит быстрее, чем в случае с technology push.

2.3. Классические модели процесса

Определение модели процесса. Процесс создания программного обеспечения не является однородным. Тот или иной метод разработки ПО, как правило, определяет некоторую динамику развертывания тех или иных видов деятельности, то есть, определяет *модель процесса* (process model).

Модель является хорошей абстракцией различных методов разработки ПО, позволяя лаконично, сжато и информативно их представить. Однако, сама идея модели процесса является одной из самых ранних в программной инженерии, когда считалось, что удачная модель – самое главное, что способствует успеху разработки. Позднее пришло осознание, что существует множество других аспектов (принципы управления и разработки, структуру команды и т.д.), которые должны быть определены согласовано друг с другом. И стали развиваться интегральные методологии разработки. Тем не

менее существует несколько классических моделей процесса, которые полезны на практике и которые будут рассмотрены ниже.

Фазы и виды деятельности. Говоря о моделях процессов, необходимо различать фазы и виды деятельности.

Фаза (phase) – это определенный этап процесса, имеющий начало, конец и выходной результат. Например, фаза проверки осуществимости проекта, сдачи проекта и т.д. Фазы следуют друг за другом в линейном порядке, характеризуются предоставлением отчетности заказчику и, часто, выплатой денег за выполненную часть работы.

Редко какой заказчик согласится первый раз увидеть результаты только после завершения проекта. С другой стороны, подрядчики предпочитают получать деньги постепенно, по мере того, как выполняются отдельные части работы. Таким образом, появляются фазы, позволяющие создавать и предъявлять промежуточные результаты проекта. Фазы полезны также безотносительно взаимодействия с заказчиком – с их помощью можно синхронизировать деятельность разных рабочих групп, а также отслеживать продвижение проекта. Примерами фаз может служить согласование с заказчиком технического задания, реализация определенной функциональности ПО, этап разработки, оканчивающийся сдачей системы на тестирование или выпуском альфа-версии.

Вид деятельности (activity) – это определенный тип работы, выполняемый в процессе разработки ПО. Разные виды деятельности часто требуют разные профессиональные навыки и выполняются разными специалистами. Например, управление проектом выполняется менеджером проекта, кодирование – программистом, тестирование – тестировщиком. Есть виды деятельности, которые могут выполняться одними и теми же специалистами – например, кодирование и проектирование (особенно в небольшом проекте) часто выполняют одни и те же люди.

В рамках одной фазы может выполняться много различных видов деятельности. Кроме того, один вид деятельности может выполняться на

разных фазах – например, тестирование: на фазе анализа и проектирования можно писать тесты и налаживать тестовое окружение, при разработке и перед сдачей производить, собственно, само тестирование. На настоящий момент для сложного программного обеспечения используются многомерные модели процесса, в которых отделение фаз от видов деятельности существенно облегчает управление разработкой ПО.

Виды деятельности, фактически, присутствуют, под разными названиями, в каждом методе разработки ПО. В RUP они называются рабочими процессами (work flow), в CMM – ключевыми областями процесса (key process area). Мы будем сохранять традиционные названия, принятые в том или ином методе, чтобы не создавать путаницы.

Водопадная модель была предложена в 1970 году Винстоном Ройсом. Фактически, впервые в процессе разработки ПО были выделены различные шаги разработки и поколеблены примитивные представления о разработке ПО в виде анализа системы и ее кодирования.

Были определены следующие шаги: разработка системных требований, разработка требований к ПО, анализ, проектирование, кодирование, тестирование, использование – см. рис. 2.1.

Достоинством этой модели явилось ограничение возможности возвратов на произвольный шаг назад, например, от тестирования – к анализу, от разработки – к работе над требованиями и т.д. Отмечалось, что такие возвраты могут катастрофически увеличить стоимость проекта и сроки его выполнения. Например, если при тестировании обнаруживаются ошибки проектирования или анализа, то их исправление часто приводит к полной переделке системы. Этой моделью допускались возвраты только на предыдущий шаг, например, от тестирования к кодированию, от кодирования к проектированию и т.д.

Наконец, в рамках этой модели было введено прототипирование, то есть предлагалось разрабатывать систему дважды, чтобы уменьшить риски разработки. Первая версия – прототип – позволяет увидеть основные риски и

обосновано принять главные архитектурные решения. На создание прототипа отводилось до одной трети времени всей разработки.

В 70-80 годах прошлого века эта модель прочно укоренилась в разработке ПО в силу своей простоты и сходности с моделями разработки иных, не программных систем. В дальнейшем, в связи с развитием программной инженерии и осознанием итеративного характера процесса разработки ПО эта модель активно критиковалась, практически, каждым автором соответствующих статей и учебников.

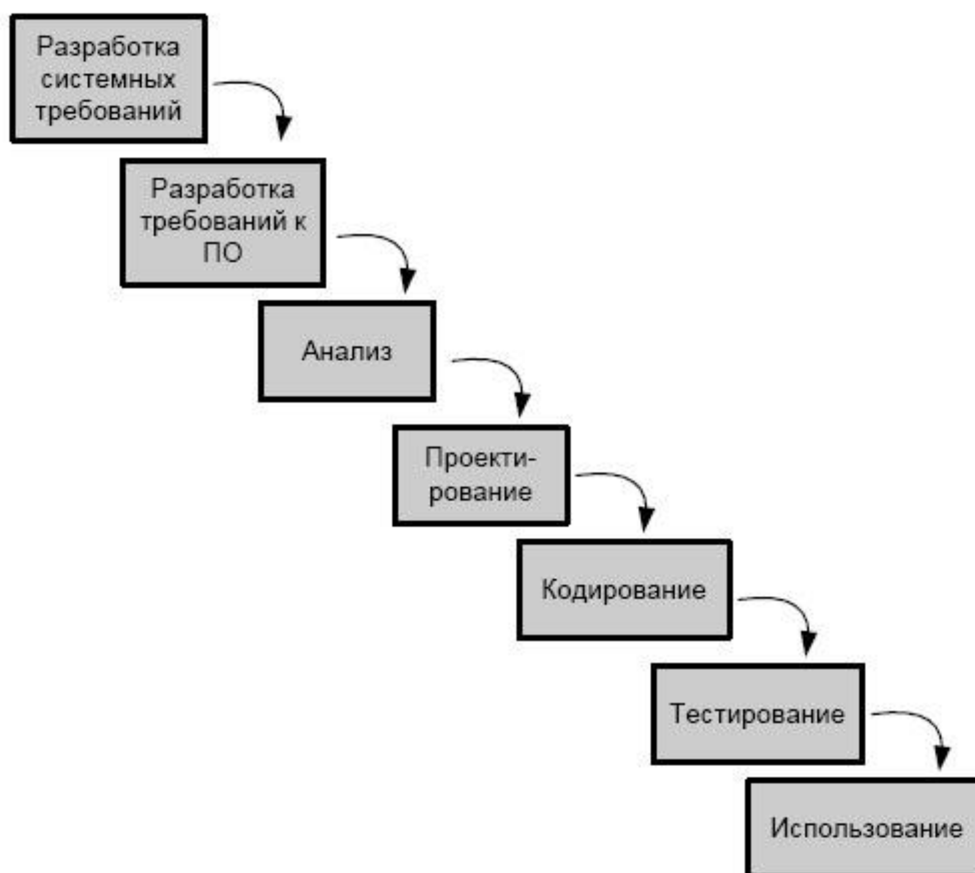

Рис. 2.1. Водопадная модель процесса

Стало общепринятым мнение, что она не отражает особенностей разработки ПО. Недостатками водопадной модели являются:

- отождествление фаз и видов деятельности, что влечет потерю гибкости разработки, в частности, трудности поддержки итеративного процесса разработки;
- требование полного окончания фазы-деятельности, закрепление результатов в виде подробного исходного документа (технического задания,

проектной спецификации); однако опыт разработки ПО показывает, что невозможно полностью завершить разработку требований, дизайн системы и т.д. – все это подвержено изменениям; и причины тут не только в том, что подвижно окружение проекта, но и в том, что заранее не удается точно определить и сформулировать многие решения, они проясняются и уточняются лишь впоследствии;

- интеграция всех результатов разработки происходит в конце, вследствие чего интеграционные проблемы дают о себе знать слишком поздно;
- пользователи и заказчик не могут ознакомиться с вариантами системы во время разработки, и видят результат только в самом конце; тем самым, они не могут повлиять на процесс создания системы, и поэтому увеличиваются риски непонимания между разработчиками и пользователями/заказчиком;
- модель неустойчива к сбоям в финансировании проекта или перераспределению денежных средств, начатая разработка, фактически, не имеет альтернатив "по ходу дела".

Однако данная модель продолжает использоваться на практике – для небольших проектов или при разработке типовых систем, где итеративность не так востребована. С ее помощью удобно отслеживать разработку и осуществлять поэтапный контроль за проектом. Эта модель также часто используется в оффшорных проектах[1] с почасовой оплатой труда. Водопадная модель вошла в качестве составной части в другие модели и методологии, например, в MSF.

Спиральная модель была предложена Бэри Боемом в 1988 году для преодоления недостатков водопадной модели, прежде всего, для лучшего управления рисками. Согласно этой модели разработка продукта осуществляется по спирали, каждый виток которой является определенной фазой разработки. В отличие от водопадной модели в спиральной нет предопределенного и обязательного набора витков, каждый виток может

стать последним при разработке системы, при его завершении составляются планы следующего витка. Наконец, виток является именно фазой, а не видом деятельности, как в водопадной модели, в его рамках может осуществляться много различных видов деятельности, то есть модель является двумерной.

Последовательность витков может быть такой: на первом витке принимается решение о целесообразности создания ПО, на следующем определяются системные требования, потом осуществляется проектирование системы и т.д. Витки могут иметь и иные значения.

Каждый виток имеет следующую структуру (секторы):

- определение целей, ограничений и альтернатив проекта;
- оценка альтернатив, оценка и разрешение рисков; возможно использование прототипирования (в том числе создание серии прототипов), симуляция системы, визуальное моделирование и анализ спецификаций; фокусировка на самых рисковых частях проекта;
- разработка и тестирование – здесь возможна водопадная модель или использование иных моделей и методов разработки ПО;
- планирование следующих итераций – анализируются результаты, планы и ресурсы на последующую разработку, принимается (или не принимается) решение о новом витке; анализируется, имеет ли смысл продолжать разрабатывать систему или нет; разработку можно и приостановить, например, из-за сбоев в финансировании; спиральная модель позволяет сделать это корректно.

Отдельная спираль может соответствовать разработке некоторой программной компоненты или внесению очередных изменений в продукт. Таким образом, у модели может появиться третье измерение.

Спиральную модель нецелесообразно применять в проектах с небольшой степенью риска, с ограниченным бюджетом, для небольших проектов. Кроме того, отсутствие хороших средств прототипирования может также сделать неудобным использование спиральной модели.

Спиральная модель не нашла широкого применения в индустрии и важна, скорее в историко-методологическом плане: она является первой итеративной моделью, имеет красивую метафору – спираль, – и, подобно водопадной модели, использовалась в дальнейшем при создании других моделей процесса и методологий разработки ПО.

Контрольные вопросы

1. Что такое универсальный процесс разработки ПО?
2. Что включает в себя стандартный процесс разработки ПО?
3. Дайте определение совершенствования процесса разработки ПО.
4. В чем заключаются Pull/Push стратегии?
5. В чем различие organization pull и technology push стратегий?
6. Чем характеризуются фазы деятельности?
7. Перечислить шаги разработки ПО.
8. В чем отличие водопадной и спиральной моделей разработки ПО?

Ключевые слова: *универсальный процесс, текущий процесс, конкретный процесс, стандартный процесс, совершенствование процесса, Pull/Push стратегии*, *Organization pull , Technology push, модель процесса, фазы деятельности, виды деятельности, водопадная модель, прототипирование, спиральная модель, виток спирали.*

Keywords: *universal process, the current process, the specific process, a standard process, process improvement, Pull / Push Strategy, Organization pull, - Technology push, the process model, the phases of activity, activities, waterfall model, prototyping, spiral model, turn of the helix.*

Kalit so'zlar: *universal jarayon, hozirgi jarayon, muayyan jarayon, standart jarayon, jarayonni takomillashtirish, Pull/Push strategiyalari, Organization pull , Technology push, jarayon modeli, faoliyat davri, faoliyat turlari, sharshara modeli, prototiplashtirish, spiral model, spiral o'rami.*

Упражнения

1. Перечислите по степени важности причины актуальности совершенствования процесса.

2. Выберите наиболее выгодную, по вашему мнению, Pull/Push стратегию. Обоснуйте.

3. Разделите процесс разработки ПО, на ваше усмотрение, на фазы.

4. Какая модель разработки выгодней по-вашему? Объясните.

5. Составьте спиральную модель разработки вашего ПО.

ГЛАВА 3. УПРАВЛЕНИЕ ПРОЕКТАМИ

В силу творческого характера программирования, существенной молодости участников разработки *ПО*, оказываются актуальными некоторые вопросы обычного промышленного производства, ставшие давно общим местом. Прежде всего, это дисциплина обязательств и *рабочий продукт*. Данные знания, будучи освоенными на практике, чрезвычайно полезны в командной работе. Кроме того, широко применяемые сейчас на практике *методологии разработки ПО*, поддержанные соответствующим программным *инструментарием*, *активно* используют эти понятия, уточняя и конкретизируя их.

3.1. Рабочий продукт

Одним из существенных условий для управляемости промышленного процесса является наличие отдельно оформленных результатов работы – как в окончательной поставке так и промежуточных. Эти отдельные результаты в составе общих результатов*работ* помогают *идентифицировать*, планировать и оценивать различные части результата. Промежуточные результаты помогают*менеджерам* разных уровней отслеживать процесс воплощения проекта, заказчик получает возможность ознакомиться с результатами задолго до окончания проекта. Более того, сами участники проекта в своей

ежедневной работе получают простой и эффективный способ обмена рабочей информацией – обмен результатами.

Таким результатом является **рабочий продукт** (*work product*) – любой *артефакт*, *произведенный* в процессе разработки *ПО*, например, *файл* или набор файлов, документы, составные части продукта, *сервисы*, процессы, спецификации, счета и т.д.

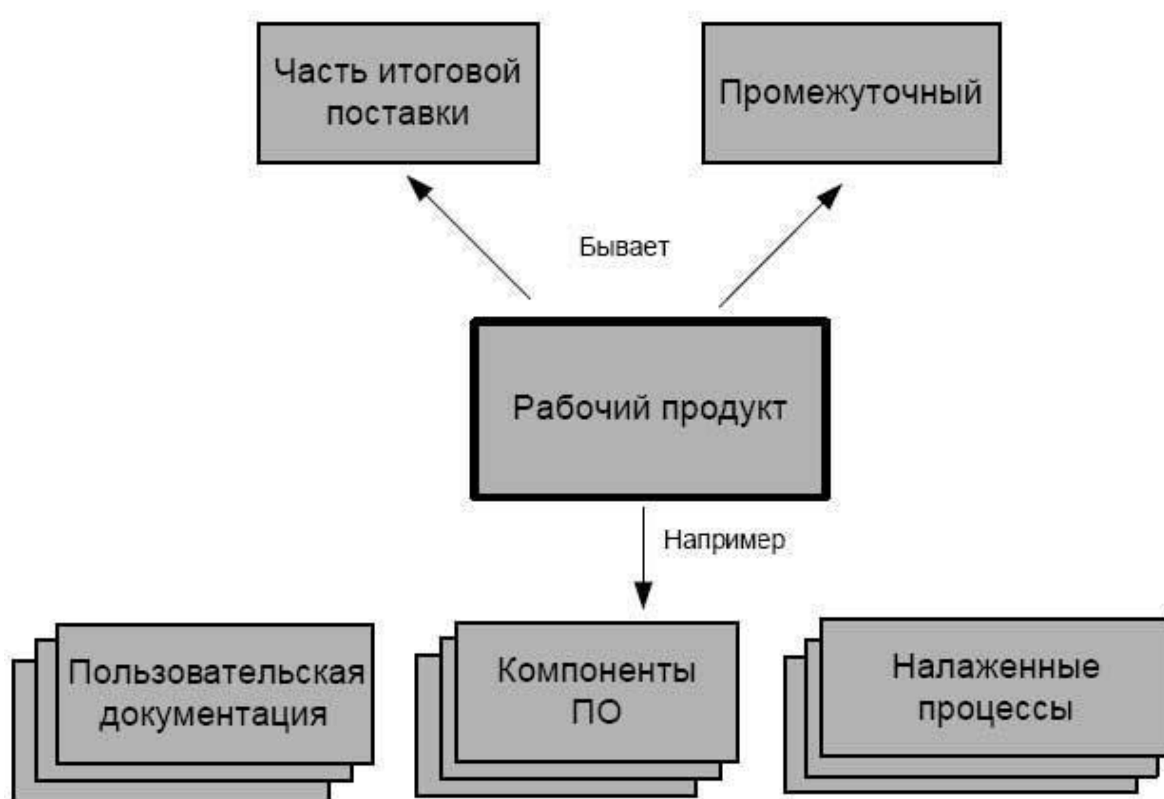

Рис. 3.1. Результаты рабочего продукта

Ключевая разница между рабочим продуктом и компонентой *ПО* заключается в том, что первый необязательно материален и осязаем (*not* to be *engineered*), хотя может быть таковым. Нематериальный *рабочий продукт* – это, как правило, некоторый налаженный процесс – промышленный процесс производства какой-либо продукции, учебный процесс в университете (на факультете, на кафедре) и т.д.

Важно отметить, что *рабочий продукт* совсем не обязательно является составной частью итоговой поставки. Например, налаженный процесс тестирования системы не поставляется заказчику вместе с самой системой. Умение управлять проектами (не только в области программирования) во

многом связано с искусством определять нужные рабочие продукты, настаивать на их создании и в их *терминах* вести приемку промежуточных этапов работы, организовывать синхронизацию различных *рабочих групп*и отдельных специалистов.

Многие методологии включают в себя описание специфичных рабочих продуктов, используемых в процессе – *CMMI*, *MSF*, *RUP* и др. Например, в *MSF* это программный код, *диаграммы* приложений и классов (*application diagrams* и *class diagrams*), план *итераций*(*iteration plan*), *модульный* тест (*unit test*) и др. Для каждого из них точно описано содержание, ответственные за разработку,*место* в процессе и др. аспекты.

Остановимся чуть детальнее на промежуточных рабочих продуктах. Компонента *ПО*, созданная в проекте одним разработчиком и предоставленная для использования другому разработчику, оказывается рабочим продуктом. Ее надо минимально протестировать, поправить имена интерфейсных классов и методов, быть может, убрать лишнее, не имеющее *отношение* к функциональности данной компоненты, разделить *public* и *private*, и т.д. То есть проделать некоторую дополнительную работу, которую, быть может, разработчик и не стал делать, если бы продолжал использовать компоненту только сам. Объем этих дополнительных *работ* существенно возрастает, если компонента должна быть представлена для использования в разработке, например, в другой центр разработки (например, иностранным партнерам, что является частой ситуацией в оффшорной разработке). Итак, изготовление хороших промежуточных рабочих продуктов очень важно для успешности проекта, но требует дополнительной работы от их *авторов*. Работать одному, не предоставляя рабочих продуктов – легче и для многих предпочтительнее. Но работа в команде требует накладных издержек, в том числе и в виде трат на создание промежуточных рабочих продуктов. Конечно, качество этих продуктов и трудозатраты на их изготовление сильно варьируются в зависимости от ситуации, но тут важно понимать сам принцип.

Итак, подытожим, что промежуточный *рабочий продукт* должен обязательно иметь ясную цель и конкретных пользователей, чтобы минимизировать накладные *расходы* на его создание.

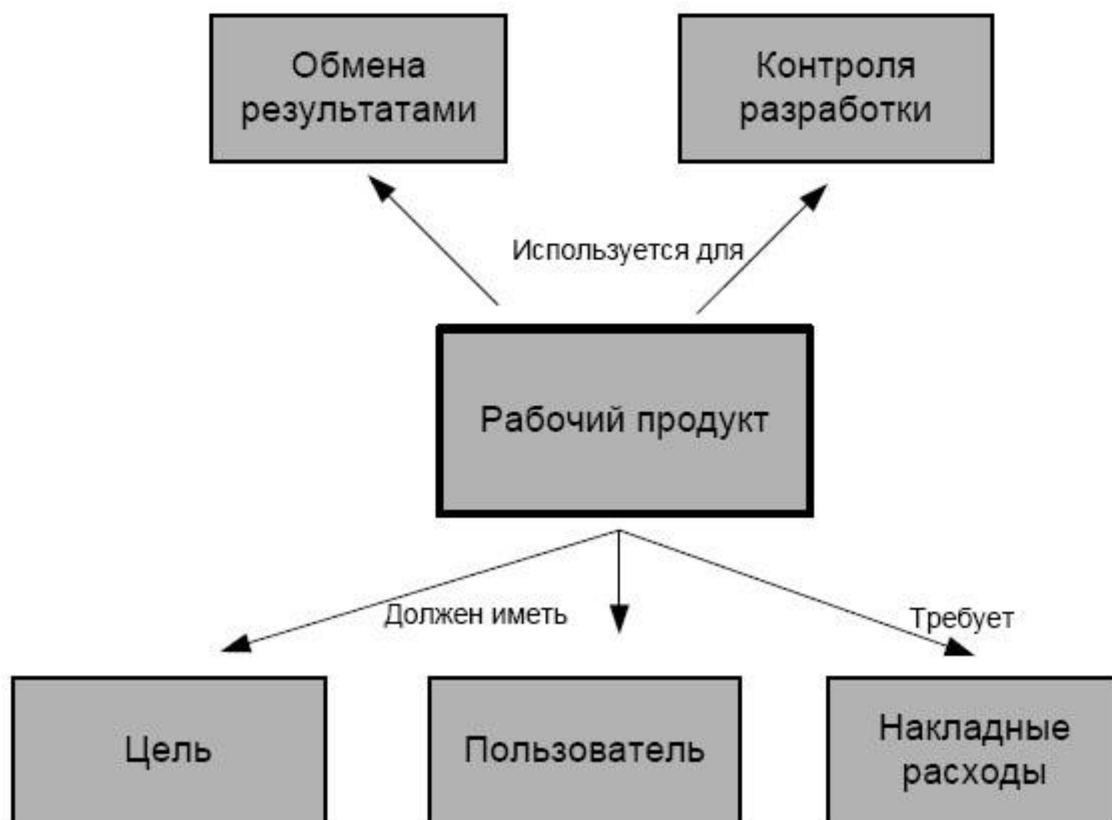

Рис. 3.2. Рабочий продукт

3.2. Дисциплина обязательств

В основе *разделения обязанностей* в бизнесе и промышленном производстве, корпоративных правил и *норм* лежит определенная деловая этика, форма отношений – **дисциплина обязательств**. Она широко используется на практике и является одной из возможных форм социального взаимоотношения между людьми. Привнесение в бизнес и промышленность иных моделей человеческих отношений – семейных, сексуальных, дружеских и т.д. часто наносит делам серьезный урон, порождает конфликтность, понижает эффективность.

Основой этой формы отношений являются обязательства, которые:

- даются добровольно;

- не даются легко – работа, ресурсы, расписание должны быть тщательно учтены;
- между сторонами включает в себя то, *что* будет сделано, *кем* и в какие *сроки* ;
- открыто и публично сформулированы (то есть это не "тайное знание").

Кроме того:

- ответственная сторона стремится выполнить обязательства, даже если нужна помощь;
- *до* наступления *deadline*, как только становится очевидно, что работа не может быть закончена в срок, обсуждаются новые обязательства.

Отметим, что дисциплина обязательств не является каким-то сводом правил, законов, она отличается также от корпоративной культуры. Это – определенный групповой психический феномен, существующий в обществе современных людей. Приведенные выше пункты не являются исчерпывающим описанием этого феномена, но лишь проявляют и обозначают его, так сказать, вызывают нужные воспоминания.

Дисциплина обязательств, несмотря на очевидность, порой, не просто реализуется на практике, например, в творческих областях человеческой *деятельности*, в области обучения и т.д. Существуют отдельные люди, которым эта дисциплина внутренне чужда вне зависимости от их рода *деятельности*.

С другой стороны, люди, освоившие эту дисциплину, часто стремятся применять ее в других областях жизни и человеческих отношений, что оказывается не всегда оправданным. Подчеркнем, что данная дисциплина является далеко не единственной моделью отношений между людьми. В качестве примера можно рассмотреть отношения в семье или дружбу, что, с очевидностью, не могут быть выражены дисциплиной обязательств. Так, вместо точности и пунктуальности в этих отношениях важно эмоционально-чувственное сопереживание, без которого они невозможны.

Дисциплине обязательств уделяется много внимания в рамках *MSF*, поскольку там в модели команды нет лидера, начальника. Эта дисциплина реализована также в Scrum: Scrum-*команда* имеет много свобод, и в силу этого – большую ответственность. Регламентируются также правила действий, когда обязательства не могут быть выполнены такой командой.

3.3. Проект и управление проектами

Классическое операционное разделение труда идет еще от Адама Смита и является сутью массового индустриального производства. То есть существует четко налаженный процесс работы и имеются области специализации – один цех точит, другой строгает, третий собирает, четвертый красит и т.д. *Пропускная способность* такого производства намного превосходит выполнение всей работы одним человеком или одной группой. Таким образом в XIX веке операционное разделение труда стало основой мануфактур, вытеснивших индивидуальное, ремесленное производство. В начале *XX* века эту структуру *работ* перенесли и на управление – то есть многочисленные менеджеры контролировали отдельные участки *работ*.

Однако высокий уровень сложности ряда задач в промышленности и бизнесе не позволяет (к счастью!) так работать везде. Существует много творческих, новых задач, где, быть может, в будущем и удастся создать конвейеры, но в данный момент для их решения требуется существенная концентрация сил и *энергии* людей, неожиданные решения, а также удача и легкая рука. Это и есть область проектов.

Проект – это уникальная (в отличии от традиционной пооперационного промышленного производства) *деятельность*, имеющая начало и конец во времени, направленная на достижение определённого результата/цель, создание определённого, уникального продукта или услуги,

при заданных ограничениях *по* ресурсам и срокам, а также требованиям к качеству и допустимому уровню риска.

В частности, *разработка программного обеспечения*, является, преимущественно, проектной областью.

Необходимо различать проекты промышленные и проекты творческие. У них разные принципы управления. Сложность промышленных проектов – в большом количестве разных организаций, компаний и относительной уникальности самих *работ*. Пример – строительство многоэтажного дома. Сюда же относятся различные международные проекты и не только промышленные – образовательные, культурные и пр. Задача в управлении такими проектами – это все охватить, все проконтролировать, ничего не забыть, все свести воедино, добиться движения, причем движения согласованного.

Творческие проекты характеризуются абсолютной новизной идеи – новый сервис, абсолютно новый *программный продукт*, какого еще не было на рынке, проекты в области искусства и науки. Любой начинающий бизнес, как правило, является таким вот творческим проектом. Причем новизна в подобных проектах не только абсолютная – такого еще не было. Такое, может, уже и было, но только не с нами, командой проекта. То есть присутствует огромный объем относительной новизны для самих людей, которые воплощают этот проект.

Проекты *по* разработке *программного обеспечения* находятся между двумя этими полюсами, занимая в этом пространстве различное положение. Часто они сложны потому, что объемны и находятся на стыке различных дисциплин – того целевого бизнеса, куда должен встроиться *программный продукт*, и сложного, нетривиального программирования. Часто сюда добавляется еще разработка уникального электронно-механического оборудования. С другой стороны, поскольку *программирование активно*продвигается в разные сферы человеческой *деятельности*, то

происходит это путем создания абсолютно новых, уникальных продуктов, и их разработка и продвижение обладают всеми чертами творческих проектов.

Управление проектами (*project management*) – область *деятельности*, в ходе которой, в рамках *определенных проектов*, определяются и достигаются четкие цели при нахождении компромисса между объемом *работ*, ресурсами (такими как время, деньги, труд, материалы, *энергия*, *пространство* и др.), временем, качеством и рисками.

Отметим несколько важных аспектов управления проектами.

- **Stakeholders** – это люди со стороны, которые не участвуют непосредственно в проекте, но влияют на него и/или заинтересованы в его результатах. Это могут быть будущие пользователи системы (например, в ситуации, когда они и заказчик – это не одно и то же), высшее руководство компании-разработчика и т.д. *Идентификация* всех *stakeholders* и грамотная работа с ними – важная составляющая успешного проектного менеджмента
- **Project scope** – это границы проекта. Это очень важное понятие для программных проектов в виду *изменчивости требований*. Часто бывает, что разработчики начинают создавать одну систему, а после, постепенно, она превращается в другую. Причем для менеджеров по продажам, а также заказчика, ничего радикально не произошло, а с точки зрения внутреннего устройства ПО, технологий, *алгоритмов* реализации, архитектуры – все радикально меняется. За подобными тенденциями должен следить и грамотно с ними разбираться проектный *менеджмент*.
- **Компромиссы** – важнейший аспект управления программными проектами в силу согласовываемости ПО. Важно не потерять все согласуемые параметры и стороны и найти приемлемый *компромисс*. Одна из техник управления *компромиссами* будет рассказана в контексте изучения методологии *MSF*.

Контрольные вопросы

1. Что может быть рабочим продуктом в процессе разработки ПО?

2. В чем различие рабочего продукта и компонентой ПО?
3. Основные характеристики рабочего продукта в процессе разработки ПО?
4. Дать определение проекта.
5. Перечислить аспекты управления проектами.

Ключевые слова: *рабочий продукт, нематериальный рабочий продукт, дисциплина обязательств, пропускная способность, проект, управление проектами, посредники, границы проекта, компромиссы.*

Keywords: *worker product, immaterial working product, discipline of the obligations, reception capacity, project, management project, stakeholders, borders of the project, compromises.*

Kalit so'zlar: *ishchi mahsulot, moddiymas ishchi mahsulot, majburiyatlar intizomi, o'tkazish qobiliyati, loyiha, loyihalarni boshqarish, vositachilar, loyiha chegaralari, o'zaro kelishuvlar.*

Упражнения

1. Возьмите любой программный продукт и перечислите цель и пользователей.
2. Возьмите ПО для примера. Составьте проект разработки.
3. Определите Stakeholders.
4. Определите Project scope.
5. Определите Компромиссы.

ГЛАВА 4. УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ

4.1. Понятия о требованиях

Например, строители строят дома, пусть разные: многоэтажные, отдельные коттеджи, офисные здания и пр. – однако, весь этот спектр вполне может охватить одна компания. Но все это дома. Строительной компании не приходится строить летающую тарелку, гиперболоид инженера Гарина, *луноход*, систему мгновенной телепортации и пр. А разработчики *ПО*, во многом, находятся именно в таком положении.

Велико разнообразие систем, которые создает одна компания, одна *команда*. Хотя сейчас и намечаются тенденции к специализации рынка

разработки *ПО*, однако, причуды мировой экономики и многие другие причины приводят к тому, что строго специализированных компаний не так много, как хотелось бы. Многие области испытывают большой дефицит отдельных программистов и целых коллективов и компаний, хорошо разбирающихся в их специфике. Примером такой области может служить телевидение, где о данной проблеме открыто говорят на заседаниях различных международных сообществ.

Кроме того, *ПО* продолжает проникать во все новые и новые области человеческой *деятельности*, и сформулировать *адекватные* требования в этом случае вообще оказывается сверхтрудной задачей.

Но даже если речь идет об одной, определенной области, то *процент* новых, уникальных черт систем, принадлежащих этой области, высок: *по сочетанию* пользовательских характеристик, *по* особенностям среды исполнения и требованиям к *интеграции*, *по распределенности информации* о требованиях среди работников компании-заказчика. Все это несет на себе очень большой отпечаток индивидуальности заказчика – персональной или его компании, – сильно связано со спецификой его бизнеса, используемого в этой области оборудования.

Кроме того, существуют трудности в понимании между заказчиком и программистами, а еще – в изменчивости *ПО* (требования имеют тенденцию меняться в ходе разработки).

В итоге, далеко не очевидно, что та система, которую хочет заказчик, вообще может быть сделана. Трудно найти черную кошку в темной комнате, особенно если ее там нет. Или то, как поняли и воплотили задачу разработчики, окажется удобным, востребованным на рынке.

Ошибки и разночтения, которые возникают при выявлении требований к системе, оказываются одними из самых дорогих. Требования – это то исходное понимание задачи разработчиками, которое является основой всей разработки.

Несколько слов о трудности взаимопонимания заказчика и разработчиков. Здесь сказывается большой разрыв между программистами и другими людьми. Во-первых, потому, что чтобы хорошо разобраться, какой должна быть система *автоматизации* больницы и система поддержки химических экспериментов – надо поработать в соответствующей области достаточное время. Или как-то иным способом научиться видеть проблемы данной *предметной области* изнутри. Во-вторых, сказывается специфичность программирования как сферы *деятельности*. Для большинства пользователей и заказчиков крайне не просто сформулировать точное *знание*, которое необходимо программистам. На вопрос, сколько типов анализов существует в вашей лаборатории, доктор, подумав, отвечает - 43. И уже потом, случайно, программист уточнил, а нет ли других типов? Конечно, есть, ответил доктор, только они случаются редко и могут быть в некотором смысле, какими угодно. В первый же раз он назвал лишь типовые. Но, конечно же, *информационная система* должна хранить информацию обо всех анализах, проведенных в лаборатории.

Теперь чуть подробнее об изменчивости *ПО* и ее причинах.

- Меняется ситуация на рынке, для которого предназначалась система или требования к системе ползут из-за быстро сменяющихся перспектив продажи еще неготовой системы.
- В ходе разработки возникают проблемы и трудности, в силу которых итоговая функциональность меняется (видоизменяется, урезается).
- Заказчик может менять свое собственное видение системы: то ли он лучше понимает, что же ему на самом деле надо, то ли выясняется, что он что-то упустил с самого начала, то ли выясняется, что разработчики его не так поняли. В общем, всякое бывает, важно лишь, что теперь заказчик определенно хочет иного.

Нечего и говорить, что *изменчивость требований по* ходу разработки очень болезненно сказывается на продукте. *Авторы* сталкивались, например, с такой ситуацией, что еще не созданную систему отдел продаж начинает

активно продавать, в силу чего поступает огромный *поток* дополнительных требований. Все их реализовать в полном объеме не удается, в итоге система оказывается набором демо-функциональности….

4.2. Виды и свойства требований

Разделим требования на две большие группы – функциональные и нефункциональные.

Функциональные требования являются детальным описанием поведения и *сервисов* системы, ее функционала. Они определяют то, что система должна уметь делать.

Нефункциональные требования не являются описанием функций системы. Этот вид требований описывает такие характеристики системы, как *надежность*, особенности поставки (наличие инсталлятора, документации), определенный уровень качества (например, для новой Java-машины это будет означать, что она удовлетворяет набору тестов, поддерживаемому компанией *Sun*). Сюда же могут относиться требования на средства и процесс разработки системы, требования к *переносимости*, соответствию стандартам и т.д. Требования этого вида часто относятся ко всей системе в целом. На практике, особенно начинающие специалисты, часто забывают про некоторые важные нефункциональные требования.

Сформулируем ряд важных свойств требований.

- *Ясность, недвусмысленность* — однозначность понимания *требований заказчиком* и разработчиками. Часто этого трудно достичь, поскольку конечная формализация требований, выполненная с точки зрения потребностей дальнейшей разработки, трудна для восприятия заказчиком или *специалистом предметной области*, которые должны проинспектировать правильность формализации.
- *Полнота и непротиворечивость*.

- *Необходимый уровень детализации*. Требования должны обладать ясно осознаваемым уровнем *детализации*, стилем описания, способом формализации: либо это описание свойств *предметной области*, для которой предназначается ПО, либо это*техническое задание*, которое прилагается к контракту, либо это проектная спецификация, которая должна быть уточнена в дальнейшем, при детальном проектировании. Либо это еще что-нибудь. Важно также ясно видеть и понимать тех, для кого данное описание требований предназначено, иначе не избежать недопонимания и последующих за этим трудностей. Ведь в разработке ПО задействовано много различных специалистов – инженеров, программистов, *тестировщиков*, представителей заказчика, возможно, будущих пользователей – и все они имеют разное образование, профессиональные навыки и специализацию, часто говорят на разных языках. Здесь также важно, чтобы требования были максимально абстрактны и независимы от реализации.
- *Прослеживаемость* — важно видеть то или иное требование в различных моделях, документах, наконец, в коде системы. А то часто возникают вопросы типа – "Кто знает, почему мы решили, что такой-то модуль должен работать следующим образом ….?". Прослеживаемость функциональных требований достигается путем их дробления на отдельные, элементарные требования, присвоение им *идентификаторов* и создание трассировочной модели, которая в идеале должна протягиваться до программного кода. Хочется например, знать, где нужно изменить код, если данное требование изменилось. На практике полная формальная прослеживаемость труднодостижима, поскольку логика и структура реализации системы могут сильно не совпадать с таковыми для модели требований. В итоге одно требование оказывается сильно "размазано" по коду, а тот или иной участок кода может влиять на много требований. Но стремиться к прослеживаемости необходимо, разумно совмещая формальные и неформальные подходы.

- *Тестируемость и проверяемость* — необходимо, чтобы существовали способы оттестировать и проверить данное требование. Причем, важны оба аспекта, поскольку часто проверить-то заказчик может, а вот тестировать данное требование очень трудно или невозможно в виду *ограниченности доступа* (например, по соображениям безопасности) к окружению системы для команды разработчика. Итак, необходимы процедуры проверки –выполнение тестов, проведение инспекций, проведение формальной *верификации* части требований и пр. Нужно также определять "планку" качества (чем выше качество, тем оно дороже стоит!), а также критерии полноты проверок, чтобы выполняющие их и *руководители проекта* четко осознавали, что именно проверено, а что еще нет.
- *Модифицируемость*. Определяет процедуры внесения изменений в требования.

4.3. Варианты формализации требований

Вообще говоря, требования как таковые – это некоторая *абстракция*. В реальной практике они всегда существуют в виде какого-то представления – документа, модели, *формальной спецификации*, списка и т.д. Требования важны как таковые, потому что оседают в виде понимания разработчиками нужд заказчика и будущих пользователей создаваемой системы. Но так как в программном проекте много различных аспектов, видов *деятельности* и фаз разработки, то это понимание может принимать очень разные представления. Каждое *представление требований* выполняет определенную задачу, например, служит "мостом", фиксацией соглашения между разными группами специалистов, или используется для оперативного *управления проектом*(отслеживается, в какой фазе реализации находится то или иное требование, кто за него отвечает и пр.), или используется для*верификации* и модельно-ориентированного тестирования. И в первом, и во втором, и в

третьем примере мы имеем дело с требованиями, но формализованы они будут *по*-разному.

Итак, формализация требований в проекте может быть очень разной – это зависит от его величины, принятого процесса разработки, используемых инструментальных средств, а также тех задач, которые решают формализованные требования. Более того, может существовать параллельно несколько формализаций, решающих различные задачи. Рассмотрим варианты.

1. *Неформальная постановка требований в переписке по электронной почте*. Хорошо работает в небольших проектах, при вовлеченности заказчика в разработку (например, команда выполняет субподряд). Хорошо также при таком стиле, когда есть взаимопонимание между заказчиком и командой, то есть лишние формальности не требуются. Однако, электронные письма в такой ситуации часто оказываются важными документами – важно уметь вести деловую переписку, подводить итоги, хранить важные письма и пользоваться ими при разногласиях. Важно также вовремя понять, когда такой способ перестает работать и необходимы более формальные подходы.

2. *Требования в виде документа* – описание *предметной области* и ее свойств, *техническое задание* как приложение к контракту, *функциональная спецификация* для разработчиков и т.д.

3. *Требования в виде графа с зависимостями* в одном из средств поддержки требований (*IBM* Rational RequisitePro, *DOORS*, Borland CaliberRM и нек. др.). Такое представление удобно при частом изменении требований, при отслеживании выполнения требований, при организации "привязки" к требованиям задач, людей, тестов, кода. Важно также, чтобы была возможность легко создавать такие графы из текстовых документов, и наоборот, создавать презентационные документы по таким графам.

4. *Формальная модель требований* для *верификации*, модельно-ориентированного тестирования и т.д.

Итак, каждый способ представления требований должен отвечать на следующие вопросы: кто потребитель, *пользователь* этого представления, как именно, с какой целью это *представление* используется.

4.4. Некоторые ошибки при документировании требований.

Перечислим ряд ошибок, встречающихся при составлении технических заданий и иных документов с требованиями.

- Описание возможных решений вместо требований.
- Нечеткие требования, которые не допускают однозначную проверку, оставляют недосказанности, имеют оттенок советов, обсуждений, рекомендаций: "Возможно, что имеет смысл реализовать также…..", "и т.д.".
- Игнорирование аудитории, для которой предназначено *представление требований*. Например, если спецификацию составляет инженер заказчика, то часто встречается переизбыток информации об оборудовании, с которым должна работать программная система, отсутствует *глоссарий* терминов и определений основных понятий, используются многочисленные синонимы и т.д. Или допущен слишком большой уклон в сторону программирования, что делает данную спецификацию непонятной всем непрограммистам.
- Пропуск важных аспектов, связанных с нефункциональными требованиями, в частности, информации об окружении системы, о сроках готовности других систем, с которыми должна взаимодействовать данная. Последнее случается, например, когда данная программная система является частью более крупного проекта. Типичны проблемы при создании программно-аппаратных систем, когда аппаратура не успевает вовремя и ПО невозможно тестировать, а в сроках и требованиях это не предусмотрено….

4.5. Цикл работы с требованиями

В своде знаний *по программной инженерии* SWEBOK определяются следующие виды *деятельности* при работе с требованиями.

- *Выделение требований (requirements elicitation)*, нацеленное на выявление всех возможных источников требований и ограничений на работу системы и *извлечение требований* из этих источников.
- *Анализ требований (requirements analysis)*, целью которого является обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация.
- *Описание требований (requirements specification)*. В результате этой *деятельности* требования должны быть оформлены в виде структурированного набора документов и моделей, который может *систематически* анализироваться, оцениваться с разных *позиций* и в итоге должен быть утвержден как официальная формулировка требований к системе.
- *Валидация требований (requirements validation)*, которая решает задачу оценки понятности сформулированных требований и их характеристик, необходимых, чтобы разрабатывать ПО на их основе, в первую очередь, непротиворечивости и полноты, а также соответствия корпоративным стандартам на техническую документацию.

Вопросы для самопроверки и контроля

1. Чем обусловлена изменчивость ПО?
2. Перечислить свойства требований.
3. Представление требований.
4. Перечислить ошибки при документировании требований.
5. Перечислить виды деятельности при работе с требованиями.

Ключевые слова: *изменчивость ПО, изменчивость требований, функциональные требования, нефункциональные требования, ясность, полнота и непротиворечивость, уровень детализации, техническое задание, прослеживаемость, тестируемость и проверяемость, модифицируемость, формальная спецификация, неформальная*

постановка требований, требование документ, требование-граф, формальная модель требований, документирование требований, выделение требований, анализ требований, описание требований, валидация требований

Keywords: *variability, variability of requirements, functional requirements, non-functional requirements, clarity, completeness and consistency, level of detail, the terms of reference, traceability, testability and testability, modifiability, formal specification, informal setting requirements, requirement document requirement graph, formal model requirements, documentation requirements, allocation of requirements, requirements analysis, description of requirements, validation requirements.*

Kalit so'zlar: *DT o'zgaruvchanligi, talablar o'zgaruvchanligi, funksional talablar, funksional bo'lmagan talablar, oydinlik, to'liqlik, zidmaslik, detalizatsiya darajasi, tehnik topshiriq, tekshirib turish, test o'tkazish, turlanish, formal spetsifikatsiya, talablarning noformal qo'yilishi, talablarni dokumentatsiasi, talablar belgilanishi, talablar tahlili, talablar ta'rifi, talablar to'g'riligi*

Упражнения

1. Определие функциональные требования для произвольного ПО.
2. Определие нефункциональные требования для произвольного ПО.
3. Какой вариант формализации требований подходит лучше всех? Почему?
4. Проанализируйте требования в задании 1.
5. Выполните валидацию требований в задании 1.

.

ГЛАВА 5. СИСТЕМНОЕ МОДЕЛИРОВАНИЕ

5.1. Понятие о модели

Пользовательские требования обычно пишутся на естественном языке, поскольку они должны быть понятны даже не специалистам в области разработки ПО. Однако более детализированные системные требования должны описываться более "техническим" способом. Одной из широко используемых методик документирования системных требований является построение ряда моделей системы. Эти модели используют графические представления, показывающие решение как исходной задачи, для которой создается система, так и разрабатываемой системы. Как правило, графические представления более понятны, чем детальное описание системных требований на естественном языке. Модели являются связующим звеном между процессом анализа исходной задачи и процессом проектирования системы.

Модели можно использовать в процессе анализа существующей системы, которую нужно или заменить, или модифицировать, а также для формирования системных требований. Модели могут представить систему в различных аспектах.

1. Внешнее представление, когда моделируется окружение или рабочая среда системы.
2. Описание поведения системы, когда моделируется ее поведение.
3. Описание структуры системы, когда моделируется системная архитектура или структуры данных, обрабатываемых системой.

Эти три типа представления систем раскрыты в данной главе; кроме того, здесь рассматривается объектное моделирование, которое до некоторой степени объединяет поведенческое и структурное моделирование.

Такие структурные методы, как структурный анализ систем [91, 12*, 24*] и объектно-ориентированный анализ [302, 54, 33*], обеспечивают основу для детального моделирования системы как части процесса

постановки и анализа требований. Большинство структурных методов работают с определенными типами системных моделей. Эти методы обычно определяют процесс, который используются для построения моделей, и набор правил, которые применяются к этим моделям. Для поддержки структурных методов существуют различные CASE-средства (обсуждаемые в разделе 5.5), включающие редакторы моделей, автоматизированную систему документирования и инструменты проверки моделей.

Однако методы структурного анализа имеют ряд недостатков.

1. Они не обеспечивают эффективной поддержки формирования нефункциональных системных требований.

2. Обычно руководства по этим методам не содержат советов, которые помогали бы пользователям решить, подходит ли данный метод для решения конкретной задачи.

3. В результате применения этих методов часто получается объемная документация, при этом суть системных требований скрывается за массой несущественных деталей.

4. Построенные модели очень детализированы и трудны для понимания. Обычные пользователи не могут реально проверить действенность этих моделей.

Практическая разработка требований не должна ограничиваться только моделями, построенными с помощью тех или иных методов. Например, объектно-ориентированные методы обычно не предполагают применения для разработки моделей потоков данных. Однако, исходя из моего опыта, такие модели полезны для объектно-ориентированного анализа, поскольку отражают понимание системы конечным пользователем и могут помочь идентифицировать объекты и действия с ними.

Наиболее важным аспектом системного моделирования является то, что оно опускает детали. Модель является абстракцией системы и легче поддается анализу, чем любое другое представление этой системы. В идеале *представление* системы должно сохранять всю информацию относительно

представляемого объекта. *Абстракция* является упрощением и определяется выбором наиболее важных характеристик системы.

Различные типы системных моделей основаны на разных подхода, к абстракции. Например, модель потоков данных концентрирует внимание на прохождении данных через систему и на функциональных преобразованиях этих данных. Модель оставляет без внимания структуру данных. И наоборот, модель "сущность-связь" предполагает документирование системных данных и их взаимосвязь, не касаясь системных функций.

Приведем типы системных моделей, которые могут создаваться в процессе анализа систем.

1. *Модель обработки данных.* Диаграммы потоков данных показывают последовательность обработки данных в системе.

2. *Композиционная модель.* Диаграммы "сущность-связь" показывают, как системные сущности составляются из других сущностей.

3. *Архитектурная модель.* Эти модели показывают основные подсистемы, из которых строится система.

4. *Классификационная модель.* Диаграммы наследования классов показывают, какие объекты имеют общие характеристики.

5. *Модель "стимул-ответ".* Диаграммы изменения состояний показывают, как система реагирует на внутренние и внешние события.

Эти типы моделей описаны далее в главе. Везде, где возможно, я использую обозначения унифицированного языка моделирования UML, который является стандартом языка моделирования, особенно для объектно-ориентированного моделирования [55, 303, 5*, 30*]. В тех случаях, когда UML не предусматривает подходящих нотаций, для описания моделей я использую простые интуитивные обозначения.

5.2. Модели системного окружения

На ранних этапах формирования требований необходимо определить границы системы. Этот этап предполагает работу с лицами, участвующими в

формировании требований (см. главу 4), для того чтобы разграничить систему и ее рабочее окружение. В некоторых случаях границы между системой и ее окружением относительно ясны. Например, когда новая система заменяет существующую, ее рабочее окружение обычно совпадает с окружением существующей системы.

В других случаях необходим дополнительный анализ. Например, рабочее окружение разрабатываемого набора CASE-средств может включать существующую базу данных, сервисы которой используются системой, но набор средств может также иметь внутреннюю базу данных. Если база данных уже существует, определение границ между ними может оказаться сложной технической и управленческой проблемой. Только после проведения дополнительного анализа можно будет принять решение о том, что является, а что не является частью разрабатываемой системы.

На определение системного окружения могут также влиять социальные и организационные ограничения, т.е. граница системы может определяться не только техническими факторами. Например, система может быть очерчена так, что при ее разработке не будет необходимости консультироваться с менеджерами; при другом определении границ может возрасти ее стоимость или возникнет необходимость расширить отдел разработки и т.п.

После определения границ между системой и ее окружением далее специфицируется само рабочее окружение и связи между ним и системой. Обычно на этом этапе строится простая структурная модель, подобная представленной на рис. 5.1 модели структуры окружения информационной системы, управляющей сетью банкоматов. Структурные модели высокого уровня обычно являются простыми блок-схемами, где каждая подсистема представлена именованным прямоугольником, а линии показывают, что существуют некоторые связи между подсистемами.

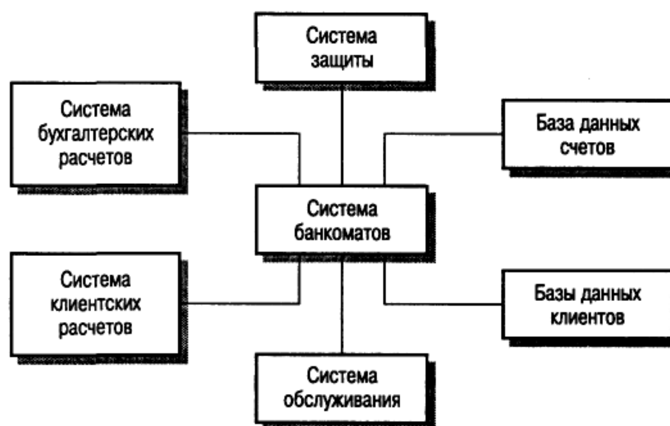

Рис. 5.1. Рабочее окружение системы управления банкоматами

На рис. 5.1 показано, что каждый банкомат присоединен к базе данных счетов, к локальной системе клиентских расчетов, к системе защиты и к системе обслуживания банкоматов. Система также соединена с базой данных клиентов, контролирующей сеть банкоматов, и с локальной бухгалтерской системой.

Структурные модели описывают непосредственное рабочее окружение системы. Но они не показывают связи между другими системами в окружающей среде, которые не соединены непосредственно с разрабатываемой системой, но могут на нее влиять. Например, внешние системы могут производить данные для системы или использовать данные, произведенные системой. При этом они могут быть соединены между собой и системой через сеть или не соединены вообще. Они могут физически соприкасаться или располагаться в разных зданиях. Все эти взаимоотношения могут влиять на требования к разрабатываемой системе и должны быть приняты во внимание.

Таким образом, простые структурные модели обычно дополняются моделями других типов, например моделями процессов, которые показывают взаимодействия в системе, или моделями потоков данных (описаны в следующем разделе), которые показывают последовательность обработки и перемещения данных внутри системы и между другими системами в окружающей среде.

На рис. 5.2 представлена модель процесса заказа оборудования организацией. Она включает определение необходимого оборудования, поиск и выбор поставщиков, заказ, поставку и проверку оборудования после поставки. Для определения системы компьютерной поддержки этого процесса необходимо решить, какие из этих действий будут выполняться системой, а какие окажутся внешними по отношению к ней. На рис. 5.2 пунктирной линией ограничены действия, выполняемые внутри такой системы.

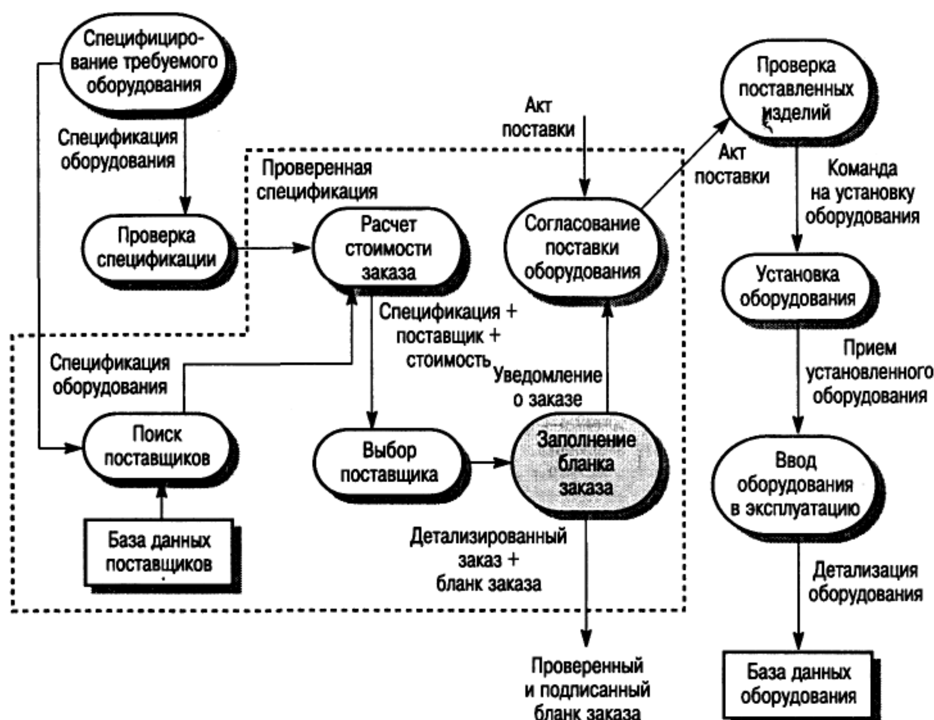

Рис. 5.2. Модель процесса приобретения оборудования

5.3. Поведенческие модели

Эти модели используются для описания общего поведения системы. Здесь рассматривается два типа поведенческих моделей – модель потоков данных, которая представляет обработку данных в системе, и модель конечного автомата, которая показывает реакцию системы на события. Эти модели можно использовать отдельно или совместно, в зависимости от типа разрабатываемой системы.

Большинство бизнес-систем прежде всего управляют данными. Они также управляют вводом данных в систему и сравнительно мало занимаются обработкой внешних событий. Для таких систем модель потоков данных может содержать все, что необходимо для описания поведения системы. В противоположность им системы реального времени управляют событиями с минимальной обработкой данных. Модель конечного автомата (обсуждаемая в разделе 5.2.2) является наиболее эффективным способом описания их поведение. Другие классы систем управляют как данными, так и событиями, поэтому для их представления необходимы оба типа моделей.

5.4. Модели потоков данных

Модели потока данных – это интуитивно понятный способ показа последовательности обработки данных внутри системы. Нотации, используемые в этих моделях, описывают обработку данных с помощью системных функций, а также хранение и перемещения данных между системными функциями. Модели потоков данных стали широко использоваться после публикации книги о структурном системном анализе [9]. На базе этого фундаментального исследования было разработано множество методов анализа систем.

Модели потоков данных используются для показа последовательности шагов обработки данных. Эти шаги обработки или преобразования данных выполняются программными функциями. В сущности, диаграммы потоков данных используются для документирования программных функций перед проектированием системы. Анализ модели обработки данных может быть выполнен специалистами вручную или с помощью компьютера.

Модель потоков данных, показанная на рис. 5.3, представляет действия, выполняемые при оформлении заказа на оборудование. Это описание части процесса размещения заказа на оборудования, представленного на рис. 5.2.

Данная модель показывает процесс перемещения бланка заказа при его обработке.

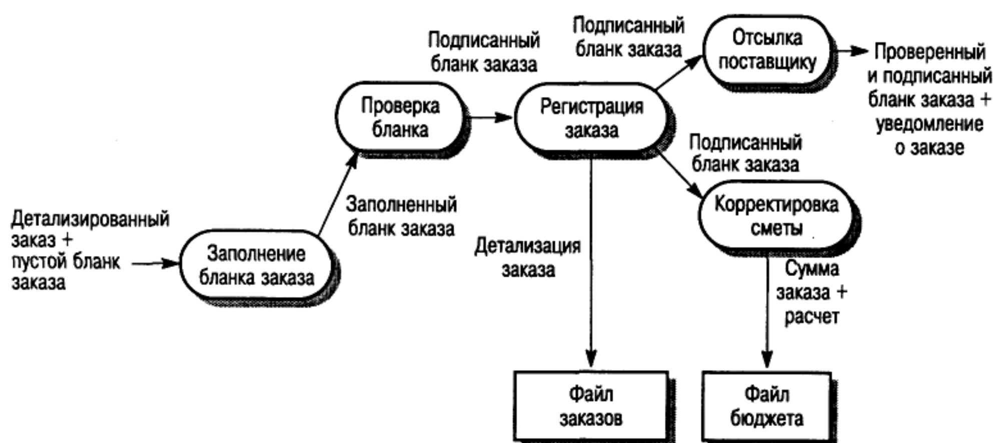

Рис. 5.3. Диаграмма потоков данных при обработке бланка заказа

В диаграммах потоков данных используются следующие обозначения (см. рис. 5.3): закругленные прямоугольники соответствуют этапам обработки данных; стрелки, снабженные примечаниями с названием данных, представляют потоки данных; прямоугольники соответствуют хранилищам или источникам данных.

Модели потоков данных ценны тем, что они прослеживают и документируют перемещение данных по системе, помогая тем самым аналитикам понять этот процесс. Преимущество диаграмм потоков данных в том, что они, в отличие от других моделей, просты и интуитивно понятны. Поэтому их можно объяснить потенциальным пользователям системы, которые затем могут участвовать в ее анализе.

Модели потоков данных показывают функциональную структуру системы, где каждое преобразование данных соответствует одной системной функции. Иногда модели потоков данных используют для описания потоков данных в рабочем окружении системы. Такая модель показывает, как различные системы и подсистемы обмениваются информацией. Подсистемы окружения не обязаны быть простыми функциями. Например, одна подсистема может быть сервером базы данных с довольно сложным

интерфейсом. На рис. 5.4 показана подобная диаграмма потоков данных. В этом примере закругленные прямоугольники представляют подсистемы.

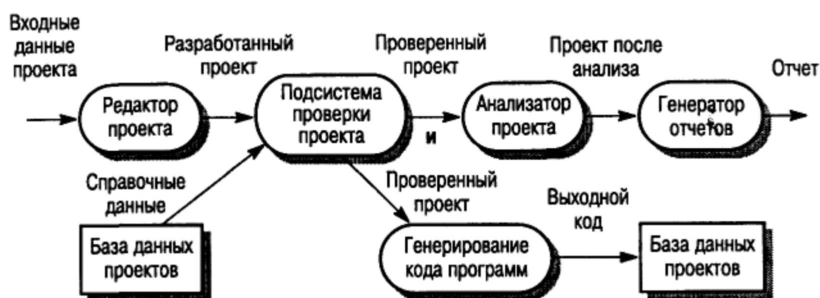

Рис. 5.4. Диаграмма потоков данных комплекса CASE-средства

5.5. Модели конечных автоматов

Модели конечных автоматов* используются для моделирования поведения системы, реагирующей на внутренние или внешние события. Такая модель показывает состояние системы и события, которые служат причиной перехода системы из одного состояния в другое. Модель не показывает поток данных внутри системы. Этот тип модели особенно полезен для моделирования систем реального времени, поскольку этими системами обычно управляют входные сигналы, приходящие из окружения системы. Например, система сигнализации, рассмотренная в главе 13, реагирует на сигналы датчиков перемещения и дверных датчиков и т.д.

Модели конечных автоматов являются неотъемлемой частью методов проектирования систем реального времени [155, 156, 338, 35*]. В работе [155] определены *диаграммы состояний,* которые стали основой системы нотаций в языке моделирования UML.

Модель конечного автомата системы предполагает, что в любое время система находится в одном из возможных состояний. При получении входного сигнала или стимула система может изменить свое состояние. Например, система, управляющая клапаном, при получении команды оператора (стимул) может перейти из состояния "Клапан открыт" к состоянию "Клапан закрыт".

На рис. 5.5 показана модель конечного автомата (диаграмма состояний) простой микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы.

Реальная микроволновая печь на самом деле намного сложнее описанной здесь системы. Вместе с тем эта модель показывает все основные средства системы. Для упрощения модели я предполагаю такую последовательность действий при использовании печи.

1. Выбор уровня мощности (половинная или полная).

2. Ввод времени работы печи.

3. Нажатие кнопки запуска, после чего печь работает заданное время.

Для безопасности печь не должна действовать при открытой двери, по окончании работы должен прозвучать звуковой сигнал. Печь имеет простой дисплей, на котором отображаются различные предупреждения и сообщения.

Нотация UML, которую я использую для описания модели конечного автомата и диаграммы состояний, разработана для моделирования поведения объектов. Но ее можно использовать для любого типа моделей конечного автомата. В этой нотации закругленные прямоугольники соответствуют состояниям системы. Они содержат краткое описание действий, выполняемых в этом состоянии. Помеченные стрелки представляют стимулы (или входные сигналы), которые приводят к переходу системы из одного состояния в другое.

На рис. 5.5 видно, что система первоначально реагирует на нажатие кнопок полной или половинной мощности. Пользователь может изменить свое решение после нажатия одной из этих кнопок и выбрать другую кнопку. Кнопка запуска сработает только после задания времени работы печи и закрытия двери. Нажатием кнопки запуска начинается работа печи в течение указанного времени.

Нотации UML позволяют определять действия, которые выполняются в том или ином состоянии. Но для создания системной спецификации необходима более детальная информация о стимулах и состояниях системы

(табл. 5.1). Эта информация может храниться в словаре данных, как будет показано далее в этом разделе.

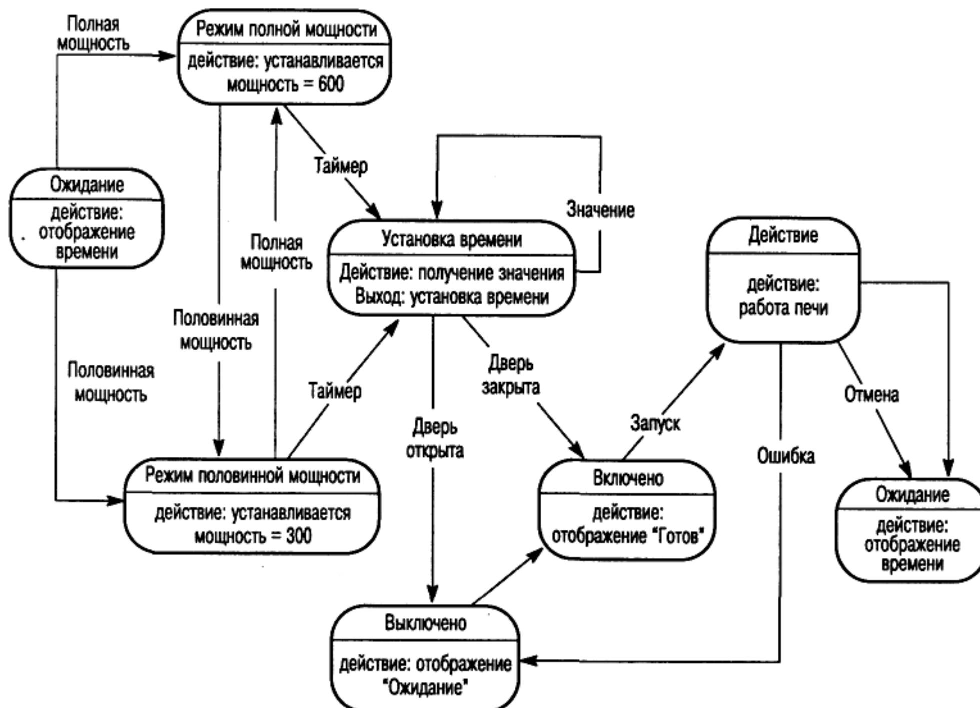

Рис. 5.5. Диаграмма состояний автомата микроволновой печи

Таблица 5.1. Описание состояний и стимулов микроволновой печи

Состояние	**Описание**
Ожидание	Печь находится в состоянии ожидания входных данных. На дисплее высвечивается текущее время
Режим половинной	Мощность печи устанавливается на 300 Вт. На дисплее отображается "Половинная мощность"
Режим полной мощности	Мощность печи устанавливается на 600 Вт. На дисплее отображается "Полная мощность"
Установка времени	Пользователем устанавливается время работы печи. Дисплей показывает заданное время
Выключено	В целях безопасности печь выключена. Внутренность печи освещена. Дисплей показывает "Не готов"
Включено	Питание печи включено. Внутри печи света нет. Дисплей
Работа	Печь работает. Внутри печи включается свет. Дисплей отображает обратный отсчет таймера. По окончании работы звучит звуковой сигнал в течение 5 секунд. Свет включен. Пока звучит сигнал, дисплей высвечивает "Приготовление закончено"
Стимулы	Описание
Половинная	Пользователь нажимает кнопку режима половинной
Полная мощность	Пользователь нажимает кнопку режима полной мощности
Таймер	Пользователь нажимает одну из кнопок таймера

Число	Пользователь вводит число
Дверь открыта	Переключатель двери печи в состоянии "Не закрыто"
Дверь закрыта	Переключатель двери печи в положении "Закрыто"
Запуск	Пользователь нажимает кнопку запуска
Отмена	Пользователь нажимает кнопку отмены

Основная проблема метода конечного автомата состоит в том, что число возможных состояний может быть очень велико. Поэтому для моделей больших систем необходима структуризация возможных состояний системы. Один из способов структуризации состоит в использовании суперсостояний, которые объединяют ряд отдельных состояний. Такое суперсостояние подобно одному состоянию модели высокого уровня, которое детализируется на отдельной диаграмме. Для иллюстрации этого понятия рассмотрим состояние Работа модели микроволновой печи (см. рис. 5.5). Это суперсостояние более детально показано на рис. 5.6.

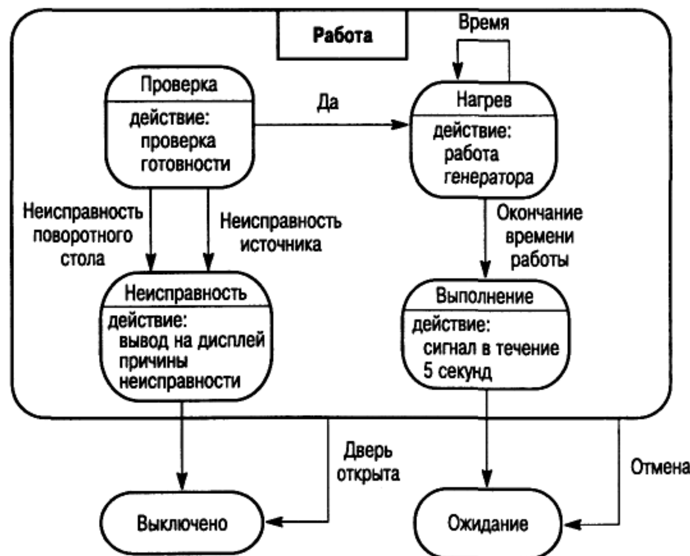

Рис. 5.6. Работа микроволновой печи

Состояние Работа включает ряд подсостояний. Диаграмма на рис. 5.6 показывает, что в состоянии Работа сначала проверяется готовность печи к работе и, если обнаружены какие-либо проблемы, включается аварийная сигнализация и печь выключается. В состоянии Нагрев работает микроволновой генератор в течение указанного времени, по завершении его

работы автоматически подается звуковой сигнал. Если во время работы печи будет открыта дверь, система переходит в состояние Выключено, как показано на рис. 5.6.

5.6. Модели данных

Многие большие программные системы используют информационные базы данных. В одних случаях эта база данных существует независимо от программной системы, в других – специально создается для разрабатываемой системы. Важной частью моделирования систем является определение логической формы данных, обрабатываемых системой.

Наиболее широко используемой методологией моделирования данных является моделирование типа "сущность-связь-атрибут", которое показывает структуру данных, их атрибуты и отношения между ними*. Этот метод моделирования был предложен в середине 1970-х годов Ченом (Chen, [71]); с тех пор разработано несколько вариантов этого метода [76,154,170,1*].

Язык моделирования UML не имеет определенных обозначений для этого типа моделей данных, что желательно для объектно-ориентированного процесса разработки ПО, где для описания систем используются объекты и их отношения. Если сущностям поставить в соответствие простейшие классы объектов (без ассоциированных методов), тогда в качестве моделей данных можно использовать модели классов UML совместно с именованными ассоциациями между классами. Хотя такие модели данных не могут служить примером "хорошего" языка моделирования, удобство использования стандартных обозначений UML перевешивает возможные несовершенства таких конструкций.

Для описания структуры обрабатываемой информации модели данных часто используются совместно с моделями потоков данных. На рис. 5.7 представлена модель данных для системы проектирования ПО. Такую

систему можно реализовать на основе комплекса инструментальных CASE-средств, показанного на рис. 5.4.

Проекты структуры ПО представляются ориентированными графами. Они состоят из набора узлов различных типов, соединенных дугами, отображающими связи между структурными узлами. В системе проектирования присутствуют средства вывода на дисплей этого графа (т.е. структурной диаграммы) и его преобразования к виду, удобному для хранения в базе данных проектов. Система редактирования выполняет преобразования структурной диаграммы из формата базы данных в формат, позволяющий отобразить ее на экране монитора в виде блок-схемы. Информация, предоставляемая редактором другим средствам анализа проекта, должна включить логическое представление графа проекта. Конечно, эти средства "не интересуются" деталями физического представления растрового изображения графа. Они работают с объектами, их логическими атрибутами и связями между ними.

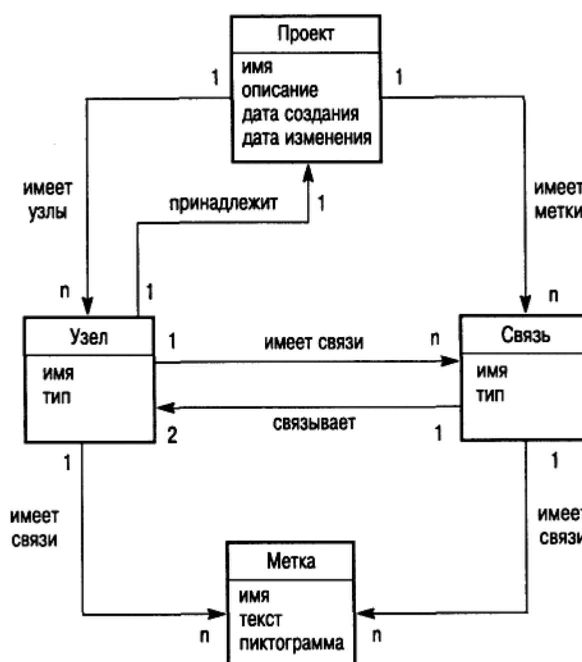

Рис. 5.7. Модель данных для системы проектирования ПО

На рис. 5.7 видно, что проект имеет атрибуты **имя, описание, дата создания и дата изменения**. Проект состоит из узлов и связей между ними

(т.е. дуг). Узлы и связи имеют атрибуты **имя и тип**. Они могут иметь набор меток, которые хранят другую описательную информацию. Каждая метка имеет атрибуты **имя, пиктограмма и текст**.

Подобно всем графическим моделям, модели "сущность-связь-атрибут" недостаточно детализированы, поэтому они обычно дополняются более подробным описанием объектов, связей и атрибутов, включенных в модель. Эти описания собираются в словари данных или репозитории. Словари данных необходимы при разработке моделей системы и могут использоваться для управления информацией, содержащейся во всех моделях системы.

Упрощенно словарь данных – это просто алфавитный список имен, которые включены в различные модели системы. Вместе с именем словарь должен содержать описание именованного объекта, а если имя соответствует сложному объекту, может быть представлено описание построения этого объекта. Другая информация, например дата создания или фамилия разработчика, может приводиться в зависимости от типа разрабатываемой модели.

Перечислим преимущества использования словаря данных.

1. Существует механизм управления именами. При разработке большой системной модели, вероятно, придется изобретать имена для сущностей и связей. Эти имена должны быть уникальными. Программа словаря данных может проверять уникальность имен и сообщать об их дублировании.

2. Словарь может служить хранилищем организационной информации, которая может связать анализ, проектирование, реализацию и модернизацию системы. Вся информация о системных объектах и сущностях находится в одном месте.

Все имена, используемые в системе (имена сущностей, типов, связей, атрибутов и системных сервисов), должны быть введены в словарь данных. Программные средства словаря должны обеспечить создание новых записей,

их хранение и запросы к словарю. Такое программное обеспечение может быть интегрировано с другими программными средствами. Большинство CASE-средств, которые применяются для моделирования систем, могут поддерживать словари данных.

В примере словаря данных, приведенном в табл. 5.2, представлены имена, взятые из модели данных системы проектирования (см. рис. 5.7). Это упрощенный пример, где опущены некоторые имена и сокращена соответствующая информация.

Таблица 5.2. Пример словаря данных

Имя	Описание	Тип	Дата
имеет метки	Отношение 1:N между сущностями типа Узел или Связь и сущностями типа Метка*	Связь	5.10.1998
Метка	Содержит информацию об узлах и связях. Метки представляются пиктограммами и соответствующим текстом	Сущность	8.12.1998
Связь	Отношение 1:1 между сущностями, представленными узлами. Связи имеют тип и имя	Связь	8.12.1998
имя (метка)	Каждая метка имеет имя, которое должно быть уникальным	Атрибут	8.12.1998
имя (узел)	Каждый узел имеет имя, которое должно быть уникальным. Имя может содержать до 64 символов	Атрибут	5.11.1998

5.7. Объектные модели

Объектно-ориентированный подход широко используется при разработке программного обеспечения, особенно для разработки интерактивных систем. В этом случае системные требования формируются на основе объектной модели, а программирование выполняется с помощью объектно-ориентированных языков, таких, как Java или C++.

Объектные модели, разработанные для формирования требований, могут использоваться как для представления данных, так и для процессов их обработки. В этом отношении они объединяют модели потоков данных и семантические модели данных. Они также полезны для классификации системных сущностей и могут представлять сущности, состоящие из других сущностей.

Для некоторых классов систем объектные модели – естественный способ отображения реально существующих объектов, которые находятся под управлением системы. Например, для систем, обрабатывающих информацию относительно конкретных объектов (таких, как автомобили, самолеты, книги и т.д.), которые имеют четко определенные атрибуты. Более абстрактные высокоуровневые сущности, например библиотеки, медицинские регистрирующие системы или текстовые редакторы, труднее моделировать в виде классов объектов, поскольку они имеют достаточно сложный интерфейс, состоящий из независимых атрибутов и методов.

Объектные модели, разработанные во время анализа требований, несомненно, упрощают переход к объектно-ориентированному проектированию и программированию. Однако конечные пользователи часто считают объектные модели неестественными и трудными для понимания. Часто они предпочитают функциональные представления процессов обработки данных. Поэтому полезно дополнить их моделями потоков данных, чтобы показать сквозную обработку данных в системе.

Класс объектов – это абстракция множества объектов, которые определяются общими атрибутами (как в семантической модели данных) и сервисами или операциями, которые обеспечиваются каждым объектом. Объекты - это исполняемые сущности с атрибутами и сервисами класса объектов. Объекты представляют собой реализацию класса, на основе одного класса можно создать много различных объектов. Обычно при разработке объектных моделей основное внимание сосредоточено на классах объектов и их отношениях.

Модели систем, разрабатываемые при формировании требований, должны отображать реальные сущности, принадлежащие классам объектов. Классы не должны содержать информацию об отдельных системных объектах. Можно разработать различные типы объектных моделей, показывающие, как классы связаны друг с другом, как объекты агрегируются из других объектов, как объекты взаимодействуют с другими объектами и т.д. Эти модели расширяют понимание разрабатываемой системы.

Идентификация объектов и классов объектов считается наиболее сложной задачей в процессе объектно-ориентированной разработки систем. Определение объектов – это основа для анализа и проектирования системы. Методы определения объектов описаны в главе 12. Здесь рассматриваются лишь некоторые объектные модели, полезные для анализа систем.

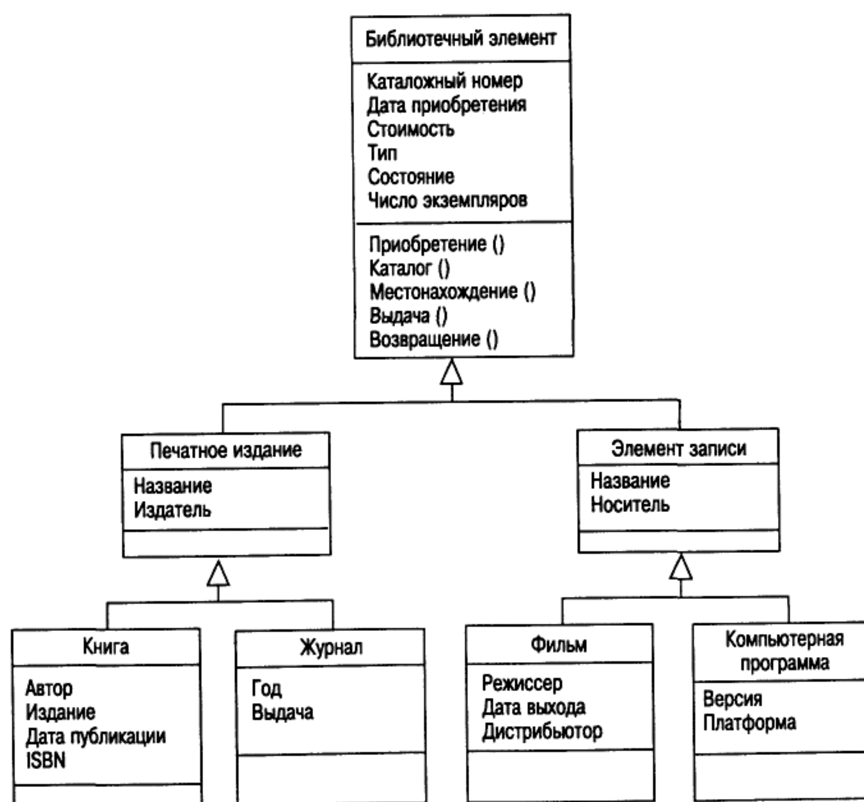

Рис. 5.8. Часть иерархии классов для библиотечной системы

Различные методы объектно-ориентированного анализа были предложены в 1990-х годах Бучем (Booch, [54]), Кодом и Джордоном (Goad and Yourdon, [74]), а также Рамбо (Rumbaugh, [302]). Эти методы имели

много общего, поэтому три главных разработчика – Буч, Рамбо и Якобсон (Jacobson) – решили интегрировать их для разработки унифицированного метода [304]. В результате разработанный ими унифицированный язык моделирования (Unified Modeling Language – UML) стал фактическим стандартом для моделирования объектов. UML предлагает нотации для различных типов системных моделей. Вы уже видели модели вариантов использования и диаграммы последовательностей в главе 5, а также диаграммы состояний ранее в этой главе.

В UML класс объектов представлен вертикально ориентированным прямоугольником с тремя секциями (рис. 5.8).

1. В верхней секции располагается имя класса.

2. Атрибуты класса находятся в средней секции.

3. Операции, связанные с классом, приводятся в нижней секции прямоугольника.

Поскольку полностью описать UML в данной книге нет возможности, здесь на объектных моделях будет показано, как классифицируются объекты, как наследуются атрибуты и операции от других объектов, а также будут приведены модели агрегирования и простые поведенческие модели.

5.8. Инструментальные CASE-средства

Это пакет программных средств, который поддерживает отдельные этапы процесса разработки программного обеспечения: проектирование, написание программного кода или тестирование. Преимущество группирования CASE-средств в инструментальный пакет заключается в том, что, работая вместе, они обеспечивают более всестороннюю поддержку процесса разработки ПО, чем могут предложить отдельные инструментальные средства. Общие сервисы могут вызываться всеми средствами. Инструментальные средства можно объединить в пакет с помощью общих файлов, репозитория или общей структуры данных.

Инструментальные средства анализа и проектирования ПО созданы для поддержки моделирования систем на этапах анализа и проектирования процесса разработки программного обеспечения. Они поддерживают создание, редактирование и анализ графических нотаций, используемых в структурных методах. Инструментальные средства анализа и проектирования часто поддерживают только определенные методы проектирования и анализа, например объектно-ориентированные. Другие инструментальные средства являются общими системами редактирования диаграмм многих типов, которые используются разными методами проектирования и анализа. Инструментальные средства, ориентированные на определенные методы, обычно автоматически поддерживают правила и базовые принципы этих методов, что позволяет выполнять автоматический контроль диаграмм.

На рис. 5.13 показана схема пакета инструментальных средств поддержки анализа и проектирования ПО. Инструментальные средства обычно объединяются через общий репозиторий, структура которого является собственностью разработчика пакета инструментальных средств. Пакеты инструментальных средств обычно закрыты, т.е. не рассчитаны на добавление пользователями собственных инструментов или на изменение средств пакета.

Рис. 5.13. Пакет инструментальных средств для анализа и проектирования ПО

Ниже перечислены средства, которые входят в пакет инструментальных средств, показанный на рис. 5.13.

1. *Редакторы диаграмм* предназначены для создания диаграмм потоков данных, иерархий объектов, диаграмм "сущность-связь" и т.д. Эти редакторы не только имеют средства рисования, но и поддерживают различные типы объектов, используемые в диаграммах.

2. *Средства проектирования, анализа и проверки* выполняют проектирование ПО и создают отчет об ошибках и дефектах в системной архитектуре. Они могут работать совместно с системой редактирования, поэтому обнаруженные ошибки можно устранить на ранней стадии процесса проектирования.

3. *Центральный репозиторий* позволяет проектировщику найти нужный проект и соответствующую проектную информацию.

4. *Словарь* данных хранит информацию об объектах, которые используются в структуре системы.

5. *Средства генерирования отчетов* на основе информации из центрального репозитория автоматически генерируют системную документацию.

6. *Средства создания форм* определяют форматы документов и экранных форм.

7. *Средства импортирования и экспортирования* позволяют обмениваться информацией из центрального репозитория различным инструментальным средствам.

8. *Генераторы программного кода* автоматически генерируют программы на основе проектов, хранящихся в центральном репозитории.

В некоторых случаях возможно генерировать программы или фрагменты программ на основе информации, представленной в системной модели. Генераторы кода, которые включены в пакеты инструментальных средств, могут генерировать код на таких языках, как Java, C++ или C. Поскольку в моделях не предусмотрена детализация низкого уровня,

генератор программного кода не в состоянии сгенерировать законченную систему. Обычно необходимы программисты для завершения автоматически сгенерированных программ.

Некоторые пакеты инструментальных средств анализа и проектирования предназначены для поддержки методов разработки программных приложений деловой сферы. Обычно для создания общего репозитория инструментов они используют системы баз данных типа Sybase или Oracle. Эти пакеты инструментальных средств содержат большое количество средств языков программирования четвертого поколения, предназначенных для генерирования программного кода на основе системной архитектуры, они также могут генерировать базы данных с использованием языков программирования четвертого поколения.

Контрольные вопросы

1. Аспекты представления моделей.
2. Что такое абстракция?
3. Перечислите типы системных моделей
4. Место системного окружения в системе разработки ПО, определение границ.
5. Что такое модели потоков данных?
6. Для чего используют модели конечных автоматов?

Ключевые слова: *модель системы, внешнее представление, поведение системы, структура системы, структурный анализ, абстракция, модель обработки данных, композиционная модель, архитектурная модель, классификационная модель, модель "стимул-ответ", границы системы, системное окружение, поведение системы, модели потоков данных.*

Keywords: *system model, external presentation, behaviour of the system, structure of the system, structured analysis, abstraction, model data processing, compositional model, architectral model, taxonomic model, model "stimulus-answer", borders of the system, system encirclement, behaviour of the system, models dataflow.*

Kalit so'zlar: *tizim modeli, tashqi tasavvur, tizim hulqi, tizim strukturasi, strukturaviy tahlil, abstraktlash, ma'lumotlar bilan ishlash modeli, kompozitsion model, arhitekturaviy*

model, klassifikatsion model, "stiml-javob" modeli, tizim chegaralari, tizim muhiti, ma'lumotlar oqimi modeli.

Упражнения

1. Разработайте модель рабочего окружения для информационной системы больницы. Модель должна предусматривать ввод данных о новых пациентах и систему хранения рентгеновских снимков.
2. Создайте модель обработки данных в системе электронной почты. Необходимо отдельно смоделировать отправку почты и ее получение.
3. Нарисуйте модель конечного автомата управляющей системы:
   - для автоматической стиральной машины, которая имеет различные программы для разных типов белья;
   - для программного обеспечения проигрывателя компакт-дисков;
   - для телефонного автоответчика, который регистрирует входные сообщения и показывает число принятых сообщений на дисплее. Система должна соединять владельца телефона с абонентом после ввода им последовательности чисел (телефонного номера абонента), а также, имея записанные сообщения, повторять их по телефону.
4. Разработайте модель классов объектов для системы электронной почты. Если вы выполнили упражнение 7.3, опишите различия и сходства между моделью обработки данных и объектной моделью.
5. Используя подход "сущность-связь", опишите возможную модель данных для системы библиотечного каталога, представленную в этой главе (см. рис. 7.8).
6. Разработайте объектную модель, включающую диаграммы иерархии классов и агрегирования, и показывающую основные элементы системы персонального компьютера и его программного обеспечения.
7. Разработайте диаграмму последовательностей, которая показывает действия студента, регистрирующегося на определенный курс в университете. Курс может иметь ограниченное число мест, поэтому процесс регистрации должен проверять количество доступных мест. Предположите, что студент обращается к электронному каталогу курсов, чтобы выяснить количество доступных мест.
8. Опишите три действия, выполняемых при моделировании систем, которые могут быть поддержаны пакетом инструментальных CASE-средств при выполнении некоторых методов анализа систем. Опишите три действия, которые невозможно легко автоматизировать.

ГЛАВА 6. ПРОТОТИПИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ

6.1. Прототип программного обеспечения

Заказчикам программного обеспечения и конечным пользователям обычно сложно четко сформулировать требования к разрабатываемой программной системе. Трудно предвидеть, как система будет влиять на трудовой процесс, как она будет взаимодействовать с другими системами и какие операции, выполняемые пользователями, необходимо автоматизировать. Тщательный анализ требований помогает уменьшить неопределенность относительно того, что система должна делать. Однако реально проверить требования, прежде чем их утвердить, практически невозможно. В этой ситуации может помочь прототип системы.

Прототип является начальной версией программной системы, которая используется для демонстрации концепций, заложенных в системе, проверки вариантов требований, а также поиска проблем, которые могут возникнуть как в ходе разработки, так и при эксплуатации системы, и возможных вариантов их решения. Очень важна быстрая разработка прототипа системы, чтобы пользователи могли начать экспериментировать с ним как можно раньше.

Прототип ПО помогает на двух этапах процесса разработки системных требований.

1. *Постановка требований.* Пользователи могут экспериментировать с системными прототипами, что позволяет им проверять, как будет работать система. Пользователи получают новые идеи для постановки требований, могут определить сильные и слабые стороны ПО. В результате могут сформироваться новые требования.

2. *Проверка требований.* Прототип позволяет обнаружить ошибки и упущения в ранее принятых требованиях. Например, системные функции, определенные в требованиях, могут быть полезными и нужными (с точки зрения пользователя). Однако в процессе применения этих функций

совместно с другими функциями пользователи могут изменить первоначальное мнение о них. В результате требования к системе изменятся, отражая измененное понимание пользователями системных функций.

Прототипирование можно использовать при анализе рисков и на начальном этапе разработки планов управления программным проектом (см. главу 3). Основной опасностью при разработке ПО являются ошибки и упущения в требованиях. Затраты на устранение ошибок в требованиях на более поздних стадиях процесса разработки могут быть очень высокими. Эксперименты показывают [48], что прототипирование уменьшает число проблем, связанных с разработкой требований. Кроме того, прототипирование уменьшает общую стоимость разработки системы. По этим причинам оно часто используется в процессе разработки требований.

Однако различие между прототипированием, как отдельным этапом процесса разработки ПО, и разработкой основной программной системы неочевидно. В настоящее время многие системы разрабатываются с использованием эволюционного подхода, когда быстро создается первоначальная версия системы, которая затем постепенно изменяется до ее окончательного варианта. При этом часто используются методы быстрой разработки приложений, которые также можно использовать при создании прототипов. Эти вопросы обсуждаются в разделе 6.2.

Наряду с тем что прототипы помогают формировать требования, они имеют и другие достоинства.

1. Различное толкование требований разработчиками ПО и пользователями можно выявить при демонстрации действующего прототипа системы.

2. В процессе создания прототипа разработчики могут выявить неполные или несогласованные требования.

3. Работая, хотя и ограниченно, в виде прототипа, система может продемонстрировать свои слабые и сильные стороны.

4. Прототип может служить основой для написания спецификации высококачественной системы. Разработка прототипа обычно ведет к улучшению спецификации системы.

Действующий прототип может также использоваться для других целей [17].

1. *Обучение пользователя.* Прототип системы можно использовать для обучения персонала перед поставкой окончательного варианта системы.

2. *Тестирование системы.* Прототипы позволяют "прокручивать" тесты. Один и тот же тест запускается на прототипе и на системе. Если получаются одинаковые результаты, это означает, что тест не обнаружил дефектов в системе. Если результаты отличаются, то необходимо исследовать причины различия, что позволяет выявить возможные ошибки в системе.

На основе изучения 39 различных программных проектов, использовавших прототипирование, в работе [13] сделан вывод, что эффективность применения прототипов при разработке ПО состоит в следующем.

1. Улучшаются эксплуатационные качества системы.

2. Система больше соответствует потребностям пользователей.

3. Системная архитектура становится более совершенной.

4. Сопровождение системы упрощается и становится более удобным.

5. Сокращаются расходы на разработку системы.

Эти исследования показывают, что улучшение эксплуатационных качеств системы и увеличение соответствия системы потребностям пользователя не требуют увеличения общей стоимости разработки системы. Прототипирование обычно повышает стоимость начальных этапов разработки ПО, но снижает затраты на более поздних этапах.

Модель процесса разработки прототипа показана на рис.6.1. На первом этапе данного процесса определяются назначение прототипа и цель прототипирования. Целью может быть разработка макета пользовательского

интерфейса, проверка функциональных системных требований или демонстрация реализуемости системы для руководства. Один и тот же прототип не может служить одновременно всем целям. Если цели определены неточно, функции прототипа могут быть восприняты неверно.

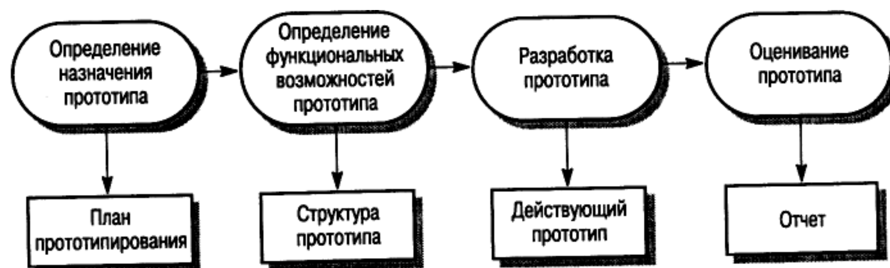

Рис. 6.1. Процесс разработки прототипа

На следующем этапе процесса разработки прототипа определяются его функциональные возможности, т.е. принимается решение о том, какие свойства системы должен отражать прототип, а какие (что, возможно, более важно) – нет. Для уменьшения затрат на создание прототипа можно исключить некоторые системные функции. Например, можно ослабить временные характеристики и требования к использованию памяти. Средства управления и обработки ошибок могут игнорироваться либо быть элементарными, если, конечно, целью прототипирования не является модель интерфейса пользователя. Также могут быть снижены требования к надежности и качеству программ.

Заключительный этап процесса прототипирования – оценивание созданного прототипа. В работе [178] утверждается, что это наиболее важный этап процесса прототипирования. Здесь проверяется, насколько созданный прототип соответствует своему назначению и целям, а также на его основе создается план мероприятий по совершенствованию разрабатываемой системы.

6.2. Прототипирование в процессе разработки ПО

Как уже отмечалось, конечным пользователям трудно представить, как они будут использовать новую систему ПО в повседневной работе. Если система большая и сложная, то это невозможно сделать, прежде чем система будет создана и введена в эксплуатацию.

Один из способов преодоления этой трудности состоит в использовании эволюционного метода разработки систем. Это означает, что пользователю предоставляется незавершенная система, которая затем изменяется и дополняется до тех пор, пока не станут ясны все требования пользователя. В качестве альтернативы можно построить "экспериментальный" прототип, который поможет проанализировать и проверить требования. После этого создается система. На рис. 6.2 показаны оба подхода к использованию прототипов.

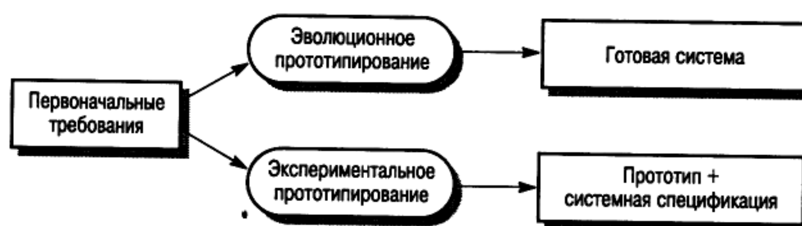

Рис.6.2. Эволюционное и экспериментальное прототипирование

Эволюционное прототипирование начинается с построения относительно простой системы, которая реализует наиболее важные требования пользователя. По мере выявления новых требований прототип изменяется и дополняется. В конечном счете он становится той системой, которая требуется. В этом процессе не используется детальная системная спецификация, во многих случаях нет даже формального документа с системными требованиями. В настоящее время эволюционное прототипирование является обычной технологией разработки программных систем, которая широко используется при разработке Web-узлов и приложений электронной коммерции.

В противоположность эволюционному подходу метод экспериментального прототипирования предназначен для разработки и

уточнения системной спецификации. Прототип создается, оценивается и модифицируется. Данные оценивания прототипа используются для дальнейшей детализации спецификации. Когда системные требования сформированы, прототип больше не нужен.

Существует различие между целями эволюционного и экспериментального прототипирования.

- Целью эволюционного прототипирования является поставка работающей системы конечному пользователю. Это означает, что необходимо начать создание системы, реализующей требования пользователя, которые наиболее понятны и которые имеют наивысший приоритет. Требования с более низким приоритетом и нечеткие требования реализуются по запросам пользователей.
- Целью экспериментального прототипирования является проверка и формирование системных требований. Здесь сначала создается прототип, реализующий те требования, которые сформулированы нечетко и с которыми необходимо "разобраться". Требования, которые сформулированы четко и понятно, не нуждаются в прототипировании.

Другое важное различие между этими подходами касается управления качеством разрабатываемой системы. Экспериментальные прототипы имеют очень короткий срок жизни. Они быстро меняются и для них высокая эксплуатационная надежность не требуется. Для экспериментального прототипа допускается пониженная эффективность и безотказность, поскольку прототип должен выполнить только свою основную функцию – помочь в понимании требований.

В противоположность этому прототипы, которые эволюционируют в законченную систему, должны быть разработаны с такими же стандартами качества, что и любое другое программное обеспечение. Они должны иметь устойчивую структуру и высокую эксплуатационную надежность. Они должны быть безотказны, эффективны и отвечать соответствующим стандартам.

6.3. Эволюционное прототипирование

В основе эволюционного прототипирования лежит идея разработки первоначальной версии системы, демонстрации ее пользователям и последующей модификации вплоть до получения системы, отвечающей всем требованиям (рис. 6.3). Такой подход сначала использовался для разработки систем, которые трудно или невозможно специфицировать (например, систем искусственного интеллекта). В настоящее время он становится основной методикой при разработке программных систем. Эволюционное прототипирование имеет много общего с методами быстрой разработки приложений и часто входит в эти методы как их составная часть [238, 346, 327, 6*].

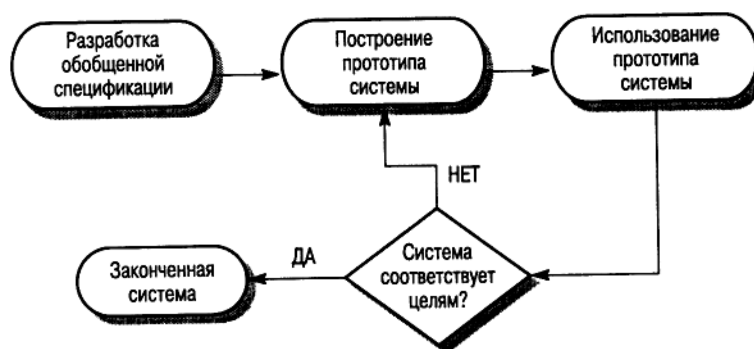

Рис. 6.3. Эволюционное прототипирование

Этот метод прототипирования имеет два основных преимущества.

1. *Ускорение разработки системы.* Как указывалось во введении, современные темпы изменений в деловой сфере требуют быстрых изменений программного обеспечения. В некоторых случаях быстрая поставка ПО, удобство и простота его использования более важны, чем полный спектр функциональных возможностей системы или долгосрочные возможности ее сопровождения.

2. *Взаимодействие пользователя с системой.* Участие пользователей в процессе разработки означает, что в системе более полно будут учтены пользовательские требования.

Между отдельными методами быстрой разработки ПО существуют различия, но все они имеют некоторые общие свойства.

1. Этапы разработки технических требований, проектирования и реализации перемежаются. Не существует детальной системной спецификации, проектная документация обычно зависит от инструментальных средств, используемых для реализации системы. Пользовательские требования определяют только наиболее важные характеристики системы.

2. Система разрабатывается пошагово. Конечные пользователи и другие лица, формирующие требования, участвуют на каждом шаге проектирования и оценивания новой версии системы. Они могут предлагать изменения и новые требования, которые будут реализованы в следующей версии системы.

3. Применение методов быстрой разработки систем (см. раздел 6.2). Они могут использовать инструментальные CASE-средства и языки четвертого поколения.

4. Пользовательский интерфейс системы обычно создается с использованием интерактивных систем разработки (см. раздел 6.3), которые позволяют быстро спроектировать и создать интерфейс.

Эволюционное прототипирование и методы, основанные на использовании детальной системной спецификации, отличаются подходами к верификации и аттестации систем. Верификация – процесс проверки системы на соответствие спецификации. Поскольку для прототипа не создается подробной спецификации, его верификация невозможна.

Аттестация системы должна показать, что программа соответствует тем целям, для которых она создавалась. Аттестацию также трудно провести без детальной спецификации, поскольку нет четких формулировок целей.

Конечные пользователи, участвующие в процессе разработки, могут быть удовлетворены системой, в то время как другие пользователи – неудовлетворены, поскольку система не полностью соответствует тем целям, которые они неявно перед ней поставили.

Верификацию и аттестацию системы, разработанной с использованием эволюционного прототипирования, можно осуществить, если она в достаточной степени соответствует поставленной цели и своему назначению. Это соответствие, конечно, нельзя измерить, можно сделать лишь субъективные оценки. Такой подход, как будет показано ниже, может породить проблемы, если программная система создается сторонними организациями-разработчиками.

Существует три основные проблемы эволюционного прототипирования, которые необходимо учитывать, особенно при разработке больших систем с длительным сроком жизненного цикла.

1. *Проблемы управления.* Структура управления разработкой программных систем строится в соответствии с утвержденной моделью процесса создания ПО, где для оценивания очередного этапа разработки используются специальные контрольные проектные элементы (см. главу 4). Прототипы эволюционируют настолько быстро, что создавать контрольные элементы становится нерентабельно. Кроме того, быстрая разработка прототипа может потребовать применения новых технологий. В этом случае может возникнуть необходимость привлечения специалистов с более высокой квалификацией.

2. *Проблемы сопровождения системы.* Из-за непрерывных изменений в прототипах изменяется также структура системы. Это означает, что система будет трудна для понимания всем, кроме первоначальных разработчиков. Кроме того, может устареть специальная технология быстрой разработки, которая использовалась при создании прототипов. Поэтому могут возникнуть трудности при поиске людей, которые имеют знания, необходимые для сопровождения системы.

3. *Проблемы заключения контрактов.* Обычно контракт на разработку систем между заказчиком и разработчиками ПО основывается на системной спецификации. При отсутствии таковой трудно составить контракт на разработку системы. Для заказчика может быть невыгоден контракт, по которому приходится просто платить разработчикам за время, потраченное на разработку проекта; также маловероятно, что разработчики согласятся на контракт с фиксированной ценой, поскольку они не могут предвидеть все прототипы, которые потребуется создать в процессе разработки системы.

Из этих проблем вытекает, что заказчики должны понимать, насколько эффективно эволюционное прототипирование в качестве метода разработки ПО. Этот метод позволяет быстро создавать системы малого и среднего размера, при этом стоимость разработки снижается, а качество повышается. Если к процессу разработки привлекаются конечные пользователи, то, вероятно, система будет соответствовать их реальным потребностям. Однако организации-разработчики, использующие этот метод, должны учитывать, что жизненный цикл таких систем будет относительно короток. При возрастании проблем с сопровождением систему необходимо заменить или полностью переписать. Для больших систем, когда к разработке привлекаются субподрядчики, на первый план выходят проблемы управления эволюционным прототипированием. В этом случае лучше применять экспериментальное прототипирование.

Пошаговая разработка (рис. 6.4) позволяет избежать некоторых проблем, характерных для эволюционного прототипирования. Общая архитектура системы, определенная на раннем этапе ее разработки, выступает в роли системного каркаса. Компоненты системы разрабатываются пошагово, затем включаются в этот каркас. Если компоненты аттестованы и включены в каркас, ни архитектура, ни компоненты уже не меняются, за исключением случая, когда обнаруживаются ошибки.

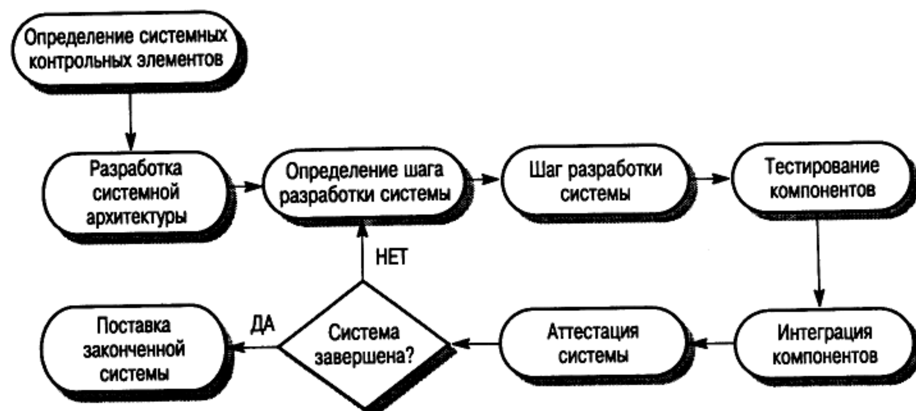

Рис.6.4. Пошаговый процесс разработки

Процесс пошаговой разработки более управляем, чем эволюционное прототипирование, поскольку следует обычным стандартам разработки ПО. Здесь планы и документация создаются для каждого шага разработки системы, что уменьшает количество ошибок. Как только системные компоненты интегрированы в каркас, их интерфейсы больше не изменяются.

6.4. Экспериментальное прототипирование

Модель процесса разработки ПО, основанная на экспериментальном прототипировании, показана на рис. 6.5. В этой модели расширен этап анализа требований в целях уменьшения общих затрат на разработку. Основное назначение прототипа – сделать понятными требования и предоставить дополнительную информацию для оценки рисков. После этого прототип больше не используется и не участвует в дальнейшем процессе разработки системы.

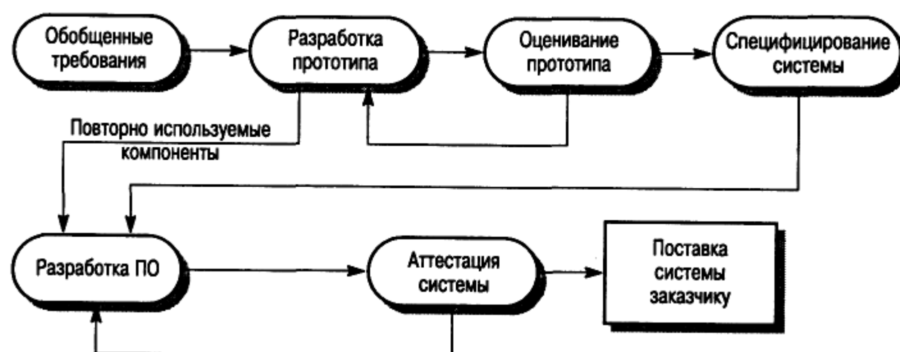

Рис.6.5. Разработка ПО с использованием экспериментальных прототипов

Этот метод прототипирования обычно используется для разработки аппаратных систем. Прежде чем будет начата дорогостоящая разработка системы, создается макет (прототип), который используется для проверки структуры системы. Электронный макет системы создается с использованием готовых компонентов, что позволяет разработать версию системы до того, как будут вложены денежные средства в разработку специализированных интегральных схем.

Экспериментальный прототип программных систем обычно не используется для проверки архитектуры системы, он помогает разработать системные требования. Прототип часто совершенно не похож на конечную систему. Система разрабатывается по возможности быстро, поэтому для ускорения формирования требований используется упрощенный прототип системы. В экспериментальный прототип закладываются только обязательные системные функции, стандарты качества для прототипа могут быть снижены, критерии эффективности игнорируются. Язык программирования прототипа часто отличается от языка программирования, на котором будет создаваться окончательный вариант системы.

В модели процесса разработки ПО, показанной на рис. 6.5, предполагается, что прототип разрабатывается исходя из обобщенных системных требований, далее над прототипом проводятся эксперименты и он изменяется до тех пор, пока его функциональные возможности не удовлетворят заказчика. После этого на основе прототипа детализируются системные требования, реализуется обычная для организации-разработчика технология разработки ПО и система доводится до окончательной версии. Некоторые компоненты прототипа могут использоваться в системе, поэтому стоимость разработки может быть снижена.

В описываемой модели разработки ПО основная проблема состоит в том, что экспериментальный прототип может не соответствовать конечной системе, поставляемой заказчику. Специалист, тестирующий прототип,

может иметь собственные интересы к системе, не типичные для ее пользователей. Время тестирования прототипа может быть недостаточным для его полного оценивания. Если прототип работает медленно, эксперты могут внести в него изменения, которые ускоряют работу, но не будут совпадать со средствами ускорения работы конечной системы.

Разработчики иногда подвергаются давлению менеджеров для ускорения работы над прототипом, особенно если намечается задержка в поставке окончательной версии системы. Обычно такое "ускорение" порождает ряд проблем.

1. Невозможно быстро настроить прототип для выполнения таких нефункциональных требований, как производительность, защищенность, устойчивость к сбоям и безотказность, которые игнорировались во время разработки прототипа.

2. Частые изменение во время разработки неизбежно приводят к тому, что прототип плохо документирован. Для разработки прототипа используется только спецификация системной архитектуры. Этого недостаточно для долговременного сопровождения системы.

3. Изменения, сделанные во время разработки прототипа могут нарушить архитектуру системы. Ее обслуживание будет сложным и дорогостоящим.

4. В процессе разработки прототипа ослабляются стандарты качества.

Чтобы быть полезными в процессе разработки требований, экспериментальные прототипы не обязательно должны выполнять роль реальных макетов систем. Бумажные формы, имитирующие пользовательские интерфейсы систем [292], показали свою эффективность при формировании требований пользователя, в уточнении проекта интерфейса и при создании сценариев работы конечного пользователя. Они очень дешевы в разработке и могут быть созданы за несколько дней. Расширением этой методики является макет пользовательского интерфейса

"Wizard of Oz" (Волшебник страны Оз). Пользователи взаимодействуют с этим интерфейсом, но их запросы направлены к специалисту, который интерпретирует их и имитирует соответствующую реакцию. Подобные подходы к прототипированию рассмотрены в работе [321].

6.5. Технологии быстрого прототипирования

Эти технологии рассчитаны главным образом на обеспечение быстрой разработки прототипов, а не на такие их системные характеристики, как производительность, удобство эксплуатации или безотказность. Существует три основных метода быстрой разработки прототипов.

1. Разработка с применением динамических языков высокого уровня.
2. Использование языков программирования баз данных.
3. Сборка приложений с повторным использованием компонентов.

Для удобства эти методы описаны в отдельных разделах. Но на практике они часто совместно используются при разработке прототипов систем. Например, язык программирования баз данных может применяться для извлечения данных с их последующей обработкой с помощью повторно используемых компонентов. Интерфейс пользователя системы можно разработать, используя визуальное программирование. В статье [224] описано смешанное применение этих методов при создании прототипа управляющей системы.

В настоящее время разработка прототипов обычно опирается на набор инструментов, поддерживающих по крайней мере два из этих методов. Например, система Smalltalk VisualWorks поддерживает язык очень высокого уровня и обеспечивает повторное использование компонентов. Пакет Lotus Notes включает поддержку программирования баз данных с помощью языка высокого уровня и повторное использование компонентов, которые могут обеспечить операции над базой данных.

Большинство систем прототипирования сегодня поддерживают визуальное программирование, при котором некоторые части или весь прототип разрабатываются в интерактивном режиме. Вместо последовательного написания программ разработчик прототипа предпочитает работать с графическими пиктограммами, представляющими функции, данные или компоненты интерфейса пользователя, и соответствующими сценариями управления этими пиктограммами. Программа, готовая к исполнению, генерируется автоматически из визуального представления системы. Это упрощает разработку программы и уменьшает затраты на прототипирование. Более подробно визуальное программирование рассматривается в разделе 6.2.3.

6.6. Применение динамических языков высокого уровня

Динамические языки высокого уровня – это языки программирования, которые имеют мощные средства контроля данных во время выполнения программы. Они упрощают разработку программ, так как уменьшают число проблем, связанных с распределением памяти и управлением ею. Такие языки имеют средства, которые обычно должны быть построены из более примитивных конструкций в языках, подобных Ada или C. Примеры языков очень высокого уровня – Lisp (основанный на структурах списков), Prolog (основанный на алгебре логики) и Smalltalk (основанный на объектах).

До недавнего времени динамические языки высокого уровня широко не использовались для разработки больших систем, поскольку они нуждаются в основательных средствах динамической поддержки. Эти средства увеличивали объем необходимой памяти и уменьшали скорость выполнения программ, написанных на этих языках. Однако возрастание мощности и снижение стоимости компьютерного оборудования сделало эти факторы не столь существенными!

Таким образом, для многих деловых приложений эти языки могут заменить такие традиционные языки программирования, как C, COBOL и Ada. Язык Java, несомненно, является основным языком разработки, имеющим корни в языке C++, но с включением многих средств языка Smalltalk наподобие платформенной независимости и автоматического управления памятью. Язык Java объединяет в себе многие преимущества языков высокого уровня, совмещая это с точностью и возможностью оптимизации выполнения, обычно предлагаемой языками третьего поколения. В языке Java много компонентов, доступных для повторного использования, все это делает его подходящим для эволюционного прототипирования.

В табл.6.1 представлены динамические языки, которые более всего используются при разработке прототипов. При выборе языка для написания прототипа необходимо ответить на ряд вопросов.

1. *Каков тип разрабатываемого приложения?* Как показано в табл. 6.1, для каждого типа приложения можно применить несколько различных языков. Если необходим прототип приложения, которое обрабатывает данных на естественном языке, то языки: Lisp или Prolog более подходят, чем Java или Smalltalk.

2. *Каков тип взаимодействия с пользователем?* Различные языки обеспечивают разные типы взаимодействия с пользователем. Некоторые языки, такие как Smalltalk и Java, хорошо интегрируются с Web-броузерами, в то время как язык Prolog лучше всего подходит для разработки текстовых интерфейсов.

3. *Какую рабочую среду обеспечивает язык ?* Развитая рабочая среда поддержки языка со своими инструментальными средствами и легким доступом к повторно используемым компонентам упрощает процесс разработки прототипа.

Таблица 6.1. Языки высокого уровня, используемые при прототипировании

Язык	Тип языка	Тип приложения
Smalltalk	Объектно-ориентированный	Интерактивные системы
Java	Объектно-ориентированный	Интерактивные системы
Prolog	Логический	Системы обработки символьной информации
Lisp	Основанный на списках	Системы обработки символьной информации

Динамические языки высокого уровня для создания прототипа можно использовать совместно, когда различные части прототипа программируются на разных языках. В работе [350] описывается разработка прототипа телефонной сетевой системы, где были использованы четыре различных языка: Prolog для макетирования баз данных, Awk [5] для составления счетов, CSP [163] для спецификации протоколов и PAISLey [351] для имитирования работы системы.

Не существует идеального языка для прототипирования больших систем, поскольку обычно различные части системы разнотипны. Преимущество многоязычного подхода в том, что для создания каждого компонента можно подобрать наиболее подходящий язык и таким образом ускорить разработку прототипа. Недостаток такого подхода в том, что трудно разработать коммуникационные связи для компонентов, написанных на разнородных языках.

6.7. Программирование баз данных

Эволюционная разработка в настоящее время является стандартной методикой для создания бизнес-приложений малого и среднего размера.

Большинство бизнес-приложений включают в себя систему управления *базой* данных и обработку данных, находящихся в ней.

Для поддержки разработки таких приложений все коммерческие системы управления базами данных имеют внутренние средства программирования. Программирование баз данных выполняется на основе специализированных языков, которые имеют встроенную базу знаний и средства, необходимые для работы с базами данных. Рабочая среда поддержки языка обеспечивает инструментальные средства для создания пользовательских интерфейсов, числовых вычислений и отчетов. Термин *язык четвертого поколения* применяется как к самому языку программирования баз данных, так и к его рабочей среде.

Языки четвертого поколения успешно применяются на практике, поскольку большинство современных приложений в той или иной мере занимаются обработкой информации, заключенной в базах данных. Основные операции, выполняемые такими приложениями, – это модификация базы данных и создание отчетов на основе информации, извлеченной из базы данных. Обычно для ввода и вывода данных используются стандартные формы. Языки четвертого поколения имеют средства для создания интерактивных приложений, позволяющие пользователям вносить изменения в базу данных. Пользовательский интерфейс обычно состоит из набора стандартных форм или электронной таблицы.

Обычно рабочая среда языков четвертого поколения включает следующие инструментальные средства (рис. 6.6).

1. В качестве языка программирования баз данных (точнее, языка запросов к базе данных) обычно используется SQL [87].

2. Генератор интерфейсов используется для создания форм ввода и отображения данных.

3. Электронная таблица применяется для анализа данных и выполнения различных действий над числовой информацией.

4. Генератор отчетов предназначен для создания отчетов на основе информации, содержащейся в базе данных.

Рис.6.6. Компоненты языка четвертого поколения

Большинство бизнес-приложений предполагают структурированные формы для ввода и вывода данных, поэтому языки четвертого поколения обеспечивают мощные средства для определения экранных форм и создания отчетов. Экранные формы часто определяются как ряд взаимосвязанных форм (в одном приложении, которое мы исследовали, было 137 различных форм), поэтому система, генерирующая экраны, должна обеспечивать следующее.

1. *Интерактивное определение форм,* когда разработчик определяет поля ввода и их организацию.

2. *Связывание форм,* когда разработчик задает определенные данные, ввод которых вызывает отображение дальнейших форм.

3. *Проверка входных данных,* когда разработчик при формировании полей форм определяет дозволенный диапазон входных величин.

В настоящее время большинством языков четвертого поколения поддерживается разработка интерфейсов баз данных, основанных на Web-броузерах. Они делают базу данных доступной с помощью Internet. Это снижает стоимость обучения и программного обеспечения и позволяет внешним пользователям иметь доступ к базе данных. Однако ограничения протоколов Internet и медленный просмотр Web-страниц делают этот метод не подходящим для систем, в которых требуется быстрое взаимодействие с пользователем.

Методы, основанные на языках четвертого поколения, могут использоваться для эволюционного прототипирования или для генерирования "одноразового" прототипа системы. Структура, которую CASE-средства накладывают на разрабатываемое приложение и сопутствующую документацию, определяет более удобное сопровождение прототипов, чем предлагают прототипы, разработанные вручную. CASE-средства могут генерировать код SQL или код на языке низшего уровня, например COBOL. В статье [116] в кратком обзоре языков четвертого поколения описан ряд инструментальных средств этого типа.

Хотя языки четвертого поколения подходят для разработки прототипов, все же они имеют ряд недостатков, проявляющихся при разработке систем. Программы, написанные на языках четвертого поколения, как правило, выполняются медленнее подобных программ, написанных на обычных языках программирования, и требуют намного больше памяти. Например, я участвовал в эксперименте, в котором перезапись на язык С++ программы, написанной на языке четвертого поколения, привела к 50%-му сокращению необходимой памяти. Программа на С также выполнялась в 10 раз быстрее, чем аналогичная программа, написанная с использованием языка четвертого поколения.

Несмотря на то что применение языков четвертого поколения снижает стоимость разработки систем, общая сумма затрат за полный жизненный цикл таких систем пока не ясна. Их программы обычно плохо структурированы и трудны в сопровождении. Специфические проблемы могут возникать при модификации подобных систем. В настоящее время языки четвертого поколения не стандартизированы и не унифицированы, поэтому при модификации систем, скорее всего, придется переписать программы, поскольку язык, на котором они написаны, устареет.

6.8. Сборка приложений с повторным использованием компонентов

Время, необходимое для разработки системы, можно уменьшить, если многие части такой системы будут использованы неоднократно. Для быстрого построения прототипа необходимо иметь набор компонентов, пригодных для повторного использования, и механизм сборки системы из этих компонентов. Этот подход показан на рис. 6.7.

Рис. 6.7. Сборка повторно используемых компонентов

Прототипирование с повторно используемыми компонентами применяется при разработке требований, конечно, если есть подходящие компоненты. Если подходящих компонентов нет, то для реализации некоторых требований будет необходим компромиссный подход. Функциональные возможности доступных компонентов могут не точно соответствовать пользовательским требованиям. Но, с другой стороны, эти требования обычно достаточно гибкие, поэтому во многих случаях возможно создание прототипа.

Разработку прототипа с повторным использованием компонентов можно реализовать на двух уровнях.

1. Уровень приложения, когда целые прикладные системы интегрируются с прототипом так, чтобы были объединены их функциональные возможности. Например, если прототипу требуются средства обработки текста, то это можно обеспечить путем интеграции в прототип стандартной системой текстового процессора. Отметим, что приложения Microsoft Office поддерживают интеграцию со сторонними системами.

2. Уровень компонентов, когда отдельные компоненты объединяются внутри структуры, реализующей систему. Такая структура может быть создана с помощью одного из языков описания сценариев, таких, как Visual Basic, TCL/TK [267], Python [225] или Perl [337]. В качестве альтернативы могут применяться такие системы, как CORBA, DCOM или JavaBeans [311, 264, 280, 27*].

Повторно используемые приложения дают доступ ко всем своим функциональным возможностям. Если, кроме того, приложение обеспечивает создание сценариев или средства автоматизации (например, макросы Excel), они также могут использоваться для расширения функциональных возможностей прототипа.

Для понимания этого метода разработки прототипа полезен составной документ, который представляет собой схему обработки данных прототипом и который можно рассматривать как контейнер для нескольких различных объектов. Эти объекты содержат разные типы данных (такие, как таблица, диаграмма, форма), которые могут обрабатываться различными приложениями.

На рис.6.8 представлен составной документ для прототипа системы, включающего текстовые элементы, элементы электронной таблицы и звуковые файлы. Текстовые элементы обрабатываются текстовым процессором, таблицы – электронной таблицей, а звуковые файлы – аудиопроигрывателем. Когда пользователь системы обращается к объекту определенного типа, вызывается связанное с ним приложение.

Рис.6.8. Связывание приложений посредством составного документа

Рассмотрим прототип системы, поддерживающей управление разработкой требований (см. главу 6). Для этой системы необходимы средства фиксации требований, их хранения, создания отчетов, поиска зависимостей между требованиями и управления этими зависимостями с помощью матриц оперативного контроля. В прототипе должна быть база данных (для хранения требований), текстовый процессор (для ввода требований и создания отчетов), электронная таблица (для управления матрицами контроля) и специально написанная программа для поиска зависимостей между требованиями.

Основное преимущество описываемого подхода к прототипированию состоит в том, что многие функциональные средства прототипа можно реализовать быстро и дешево. Если пользователи, тестирующие прототип, знакомы с приложениями, интегрированными в прототип, им нет необходимости учиться использовать новые средства прототипа. Проблемы при работе с прототипом могут возникнуть только при переключении с одного приложения на другое. Но это в значительной степени зависит от используемой операционной системы. Для организации переключения между приложениями наиболее широко используется механизм связывания и внедрения объектов OLE от Microsoft [31].

Не всегда возможно или удобно использовать целые приложения. Для создания прототипов можно использовать более "тонкие" компоненты. Это могут быть отдельные функции или объекты, которые выполняют специальные действия, например сортировку, поиск, отображение данных и т.д. Прототипирование начинается с определения общей структуры прототипа, затем компоненты интегрируются в соответствии с этой структурой. Если нет компонентов, выполняющих требующиеся функции, вместо них разрабатываются отдельные программы, которые в будущем также можно повторно использовать.

Визуальные системы разработки приложений, подобные Visual Basic, поддерживают повторное использование компонентов. Разработчики строят систему в интерактивном режиме, определяя интерфейс в виде набора экранных форм, полей, кнопок и меню. Эти элементы интерфейса именованы, и каждый из них связан со сценарием обработки событий и данных. Эти сценарии могут вызывать повторно используемые программные компоненты.

На рис.6.9 показан экран приложения, содержащий меню (в верхней части), поля ввода (белые области слева на экране), поля вывода (затенная область слева) и кнопки, представленные скругленными прямоугольниками справа на экране. После расположения этих графических элементов на экране разработчик определяет, какие готовые компоненты будут связаны с ними, либо пишет программы, выполняющие необходимые действия. На рис.6.9 показаны компоненты, связанные с некоторыми отображаемыми элементами экрана.

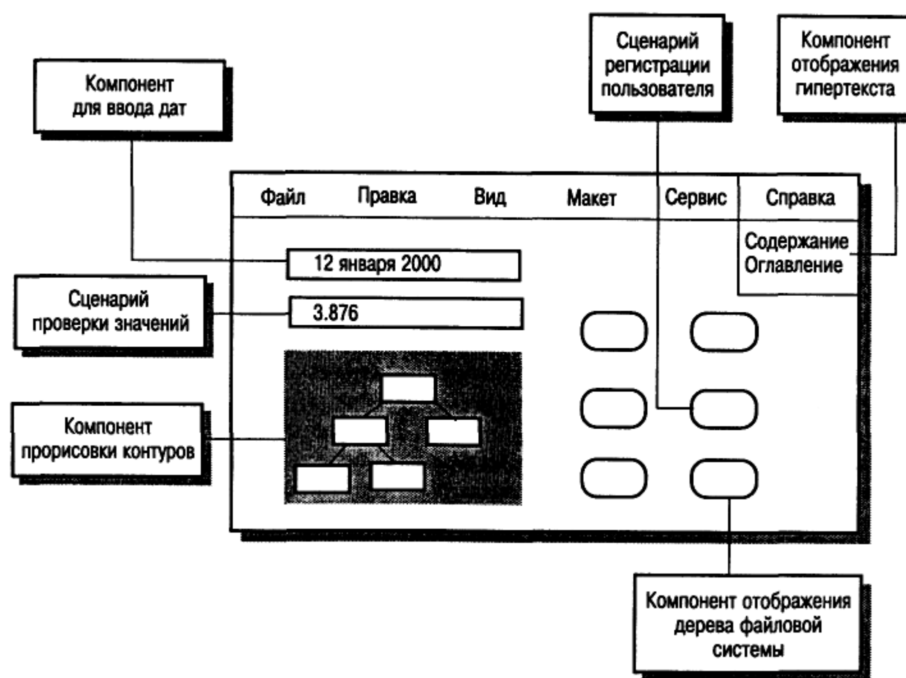

Рис.6.9. Визуальное программирования с повторным использованием компонентов

Visual Basic – пример семейства языков, названных языками сценариев [26]. Языки сценариев – нетипичные языки высокого уровня, разработанные

для интеграции компонентов в единую систему. Ранним примером языка сценариев может служить оболочка Unix [36], с тех пор были созданы другие более мощные языки создания сценариев [26, 22, 33, 4*]. Эти языки имеют управляющие структуры и графические инструментальные средства, радикально уменьшающие время разработки систем.

Описанный подход к разработке систем предоставляет возможность быстро разработать относительно малые и простые приложения, которые могут быть построены одним лицом или небольшим коллективом разработчиков. Для больших систем, которые должны разрабатываться большими коллективами, подобный подход организовать более сложно, поскольку не существует явной архитектуры системы и часто имеются сложные зависимости между различными компонентами системы. В этом причина трудностей при внесении изменений в систему. Кроме того, ограничен набор компонентов, поэтому бывает трудно реализовать нестандартные пользовательские интерфейсы. Общий покомпонентный метод разработки, обсуждаемый в главе 14, является более подходящим для больших программных систем.

6.9. Прототипирование пользовательских интерфейсов

Графические интерфейсы пользователя в настоящее время являются стандартом для интерактивных систем. Усилия, вкладываемые в определение, проектирование и реализацию такого интерфейса, составляют значительную часть стоимости разработки приложения. Как отмечается в главе 15, разработчики не должны навязывать пользователям свою точку зрения на проектируемый интерфейс. Пользователи должны принимать активное участие в процессе проектирования интерфейса. Такой взгляд на разработку интерфейса привел к подходу, названному проектированием, ориентированным на пользователя (user-centred design) [258], который

основан на прототипировании интерфейса и участии пользователя в процессе его проектирования.

В этом аспекте прототипирование – необходимая часть процесса проектирования пользовательского интерфейса. Из-за динамической природы пользовательских интерфейсов текстовых описаний и диаграмм недостаточно для формирования требований к интерфейсу. Поэтому эволюционное прототипирование с участием конечного пользователя – единственный приемлемый способ разработки графического интерфейса для программных систем.

Генераторы интерфейсов – это графические системы проектирования экранных форм, где интерфейсы компонуются из элементов типа меню, полей, пиктограмм и кнопок, которые, в свою очередь, можно просто выбрать из меню и поместить в экранную форму. Как уже упоминалось, системы этого типа – необходимая часть систем программирования баз данных. В книге [31] рассмотрен ряд таких систем. Генераторы интерфейсов создают хорошо структурированную программу, сгенерированную по спецификации интерфейса.

Миллионы людей сегодня имеют доступ к Web-броузерам. Они поддерживают язык разметки гипертекста HTML, который позволяет создавать пользовательские интерфейсы. Кнопки, поля, формы и таблицы могут быть включены в Web-страницы так же, как и средства мультимедиа. Сценарии обработки событий и данных, связанные с объектами интерфейса, могут выполняться или на машине Web-клиента, или на Web-сервере.

Из-за широких возможностей Web-броузеров и мощности языка HTML в настоящее время все больше пользовательских интерфейсов строятся как Web-ориентированные. Как показано в главе 26, посвященной наследуемым системам, такие интерфейсы – принадлежность не только новых систем; они заменяют интерфейсы, построенные на текстовых формах, в широком круге наследуемых систем.

Для Web-ориентированных интерфейсов прототипы можно создавать с помощью стандартных редакторов Web-страниц, которые, по существу, строят пользовательские интерфейсы. Объекты на Web-странице определяются, как и связанные с ними операции, с помощью встроенных средств языка HTML (например, связывание с другой страницей) или с помощью языка Java либо сценариев.

Контрольные вопросы

1. Что такое прототип?
2. Для чего применяют прототип?
3. На каких этапах процесса разработки системных требований помогает прототип ПО?
4. Перечислите преимущества прототипов?
5. Какие различия между целями эволюционного и экспериментального прототипирования?
6. В чем преимущества и недостатки эволюционного прототипирования?
7. В чем преимущества и недостатки экспериментального прототипирования?
8. Какие методы существуют для быстрой разработки прототипов?
9. В чем особенность применения языков четвертого поколения?

Ключевые слова: *прототипирование, постановка требований, проверка требований, проблемы управления, проблемы сопровождения системы, проблемы заключения контрактов, язык четвертого поколения, генераторы интерфейсов.*

Keywords: *prototyping, production requirements, inspection requirements, management problems, issues tracking system, contracting problems, fourth-generation languages, generators interfaces.*

Kalit so'zlar: *prototiplashtirish, talablar o'rnatilishi, talablarni tekshirish, boshqaruv muammolarim tizim kuzatuvining muammolari, shartnomalar tuzishdagi muammolar, to'rtinchi avlod tili, interfeys generatorlari.*

Упражнения

1. Исследуйте возможность прототипирования в процессе разработки программного обеспечения в вашей организации. Напишите отчет для вашего менеджера, показывая классы проектов, где должно использоваться прототипирование, и рассчитайте ожидаемые затраты и выгоды от использования прототипирования.

2. Объясните, почему для разработки больших систем рекомендуется экспериментальное прототипирование.

3. Какие особенности языков, подобных Smalltalk и Lisp, способствуют поддержке быстрого прототипирования?

4. В каких обстоятельствах вы рекомендовали бы прототипирование как средство обоснования системных требований?

5. Опишите трудности, которые могут возникнуть при прототипировании встроенных компьютерных систем реального времени.

6. Спроектируйте программную систему преобразования требований в формальную спецификацию. Прокомментируйте преимущества и недостатки следующих стратегий разработки такой системы.

   - Разработайте экспериментальный прототип с помощью языка, подобного Smalltalk. Оцените этот прототип, затем сделайте обзор требований. Разработайте конечную систему, используя язык С.
   - Разработайте систему согласно существующим требованиям, используя язык Java, и затем модифицируйте ее, чтобы адаптировать к изменениям требований пользователя.
   - Разработайте систему, используя эволюционное прототипирование, с помощью языка типа Smalltalk. Измените прототип в соответствии с новыми пользовательскими запросами.

7. Обсудите прототипирование на основе повторного использования компонентов и опишите проблемы, которые могут при этом возникнуть. Как наиболее эффективно определить пригодные для повторного использования компоненты?

8. Каковы преимущества и недостатки использования механизма OLE для быстрой разработки приложений?

9. Благотворительная организация попросила вас создать макет системы, которая следила бы за всеми получаемыми ими пожертвованиями. Эта система должна сохранять имена и адреса жертвующих, их интересы, пожертвованную сумму и дату пожертвования. Если пожертвование достигает определенной суммы, жертвующий может добавить условия к пожертвованию (например, пожертвование должно быть израсходовано на определенный проект), система должна следить за такими пожертвованиями и за тем, как они были израсходованы. Обсудите, как использовать прототип системы, имея в виду, что системой будут пользоваться как постоянные работники благотворительной организации, так и добровольцы. Многие из добровольцев – пенсионеры, которые имеют малый опыт работы с компьютером или вовсе не имеют такого опыта.

10. Вы разработали экспериментальный прототип системы для заказчика, который его полностью удовлетворил. Заказчик утверждает, что нет необходимости разрабатывать конечную систему, а вы можете поставить прототип, и предлагает за это хорошую цену. Вы знаете, что в будущем могут быть проблемы с сопровождением системы. Обсудите, что вы ответите этому заказчику.

ГЛАВА 7. АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ

Большие системы всегда можно разбить на подсистемы, предоставляющие связанные наборы сервисов. *Архитектурным проектированием* называют первый этап процесса проектирования, на котором определяются подсистемы, а также структура управления и взаимодействия подсистем. Целью архитектурного проектирования является описание *архитектуры программного обеспечения.*

В разделе 3.4 рассматривалась общая структура процесса проектирования. На рис. 3.9 была представлена модель процесса проектирования, первым этапом которого является архитектурное проектирование, служащее соединяющим звеном между процессом проектирования и процессом разработки требований к создаваемой системе. В идеале в спецификации требований не должно быть информации о структуре системы. В действительности же это справедливо только для небольших систем. Архитектурная декомпозиция системы необходима для структуризации и организации системной спецификации. Хорошим примером тому может служить изображенная на рис. 2.3 система управления воздушными полетами. Модель системной архитектуры часто является отправной точкой для создания спецификации различных частей системы. В процессе архитектурного проектирования разрабатывается базовая структура системы, т.е. определяются основные компоненты системы и взаимодействия между ними.

Существуют различные подходы к процессу архитектурного проектирования, которые зависят от профессионального опыта, а также мастерства и интуиции разработчиков. И все же можно выделить несколько этапов, общих для всех процессов архитектурного проектирования.

1. *Структурирование системы.* Программная система структурируется в виде совокупности относительно независимых подсистем.

Также определяются взаимодействия между подсистемами. Этот этап рассматривается в разделе 7.1.

2. *Моделирование управления.* Разрабатывается базовая модель управления взаимоотношениями между частями системы. Этот этап рассматривается в разделе 7.2.

3. *Модульная декомпозиция.* Каждая определенная на первом этапе подсистема разбивается на отдельные модули. Здесь определяются типы модулей и типы их взаимосвязей. Этот этап рассматривается в разделе 7.3.

Как правило, эти этапы перемежаются и накладываются друг на друга. Этапы повторяются для все более детальной проработки архитектуры до тех пор, пока архитектурный проект не будет удовлетворять системным требованиям.

Четких различий между подсистемами и модулями нет, но, думаю, будут полезными следующие определения.

1. *Подсистема* – это система (т.е. удовлетворяет "классическому" определению "система"), операции (методы) которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими подсистемами.

2. *Модуль* – это обычно компонент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Как правило, модуль никогда не рассматривается как независимая система. Модули обычно состоят из ряда других, более простых компонентов.

Результатом процесса архитектурного проектирования является документ, отображающий архитектуру системы. Он состоит из набора графических схем представлений моделей системы с соответствующим описанием. В описании должно быть указано, из каких подсистем состоит система и из каких модулей слагается каждая подсистема. Графические

схемы моделей системы позволяют взглянуть на архитектуру с разных сторон. Как правило, разрабатывается четыре архитектурные модели.

1. Статическая структурная модель, в которой представлены подсистемы или компоненты, разрабатываемые в дальнейшем независимо.

2. Динамическая модель процессов, в которой представлена организация процессов во время работы системы.

3. Интерфейсная модель, которая определяет сервисы, предоставляемые каждой подсистемой через общий интерфейс.

4. Модели отношений, в которых показаны взаимоотношения между частями системы, например поток данных между подсистемами.

Ряд исследователей при описании архитектуры систем предлагают использовать специальные языки описания архитектур. В книге [29] рассматриваются основные свойства этих языков. В них основными архитектурными элементами являются компоненты и коннекторы (объединяющие звенья); эти языки также предлагают принципы и правила построения архитектур. Однако, как и другие специализированные языки, они имеют один недостаток, а именно: все они понятны только освоившим их специалистам и почти не используются на практике. Фактически использование языков описания архитектур только усложняет анализ систем. Поэтому я считаю, что для описания архитектур лучше использовать неформальные модели и системы нотации, подобные предлагаемой, например унифицированный язык моделирования UML.

Архитектура системы может строиться в соответствии с определенной архитектурной моделью [126]. Очень важно знать эти модели, их недостатки, преимущества и возможности применения. В этой главе рассматриваются структурные модели, модели управления и декомпозиции.

Вместе с тем архитектуру больших систем невозможно описать с помощью какой-либо одной модели. При разработке отдельных частей больших систем можно использовать разные архитектурные модели. Но в этом случае архитектура системы может оказаться слишком сложной,

поскольку будет построена на комбинации различных архитектурных моделей. Разработчик должен подобрать наиболее подходящую модель, затем модифицировать ее соответственно требованиям разрабатываемого ПО. В разделе 7.4 рассматривается пример архитектуры компилятора, базирующейся на комбинации модели репозитория и модели потоков данных.

Архитектура системы влияет на производительность, надежность, удобство сопровождения и другие характеристики системы. Поэтому модели архитектуры, выбранные для данной системы, могут зависеть от нефункциональных системных требований.

1. *Производительность.* Если критическим требованием является производительность системы, следует разработать такую архитектуру, чтобы за все критические операции отвечало как можно меньше подсистем с максимально малым взаимодействием между ними. Чтобы уменьшить взаимодействие между компонентами, лучше использовать крупномодульные компоненты, а не мелкие структурные элементы.

2. *Защищенность.* В этом случае архитектура должна иметь многоуровневую структуру, в которой наиболее критические системные элементы защищены на внутренних уровнях, а проверка безопасности этих уровней осуществляется на более высоком уровне.

3. *Безопасность.* В этом случае архитектуру следует спроектировать так, чтобы за все операции, влияющие на безопасность системы, отвечало как можно меньше подсистем. Такой подход позволяет снизить стоимость разработки и решает проблему проверки надежности.

4. *Надежность.* В этом случае следует разработать архитектуру с включением избыточных компонентов, чтобы можно было заменять и обновлять их, не прерывая работу системы. Архитектуры отказоустойчивых систем с высокой работоспособностью рассматриваются в главе 18.

5. *Удобство сопровождения.* В этом случае архитектуру системы следует проектировать на уровне мелких структурных компонентов, которые

можно легко изменять. Программы, создающие данные, должны быть отделены от программ, использующих эти данные. Следует также избегать структуры совместного использования данных.

Очевидно, что некоторые из перечисленных архитектур противоречат друг другу. Например, для того чтобы повысить производительность, необходимо использовать крупномодульные компоненты, в то же время сопровождение системы намного упрощается, если она состоит из мелких структурных компонентов. Если необходимо учесть оба требования, следует искать компромиссное решение. Ранее уже было сказано, что один из способов решения подобных проблем состоит в применении различных архитектурных моделей для разных частей системы.

7.1. Структурирование системы

На первом этапе процесса проектирования архитектуры система разбивается на несколько взаимодействующих подсистем. На самом абстрактном уровне архитектуру системы можно изобразить графически с помощью блок-схемы, в которой отдельные подсистемы представлены отдельными блоками. Если подсистему также можно разбить на несколько частей, на диаграмме эти части изображаются прямоугольниками внутри больших блоков. Потоки данных и/или потоки управления между подсистемами обозначается стрелками. Такая блок-схема дает общее представление о структуре системы.

На рис. 7.1 представлена структурная модель архитектуры для системы управления автоматической упаковкой различных типов объектов. Она состоит из нескольких частей. Подсистема наблюдения изучает объекты на конвейере, определяет тип объекта и выбирает для него соответствующий тип упаковки. Затем объекты снимаются с конвейера, упаковываются и помещаются на другой конвейер. Примеры других архитектур приведены на рис. 2.2 и 2.3.

Бэсс (Bass, [29]) считает, что подобные блок-схемы являются бесполезными представлениями системной архитектуры, поскольку из них нельзя ничего узнать ни о природе взаимоотношений между компонентами системы, ни об их свойствах. С точки зрения разработчика программного обеспечения, это абсолютно верно. Однако такие модели оказываются эффективными на этапе предварительного проектирования системы. Эта модель не перегружена деталями, с ее помощью удобно представить структуру системы. В структурной модели определены все основные подсистемы, которые можно разрабатывать независимо от остальных подсистем, следовательно, руководитель проекта может распределить разработку этих подсистем между различными исполнителями. Конечно, для представления архитектуры используются не только блок-схемы, однако подобное представление системы не менее полезно, чем другие архитектурные модели.

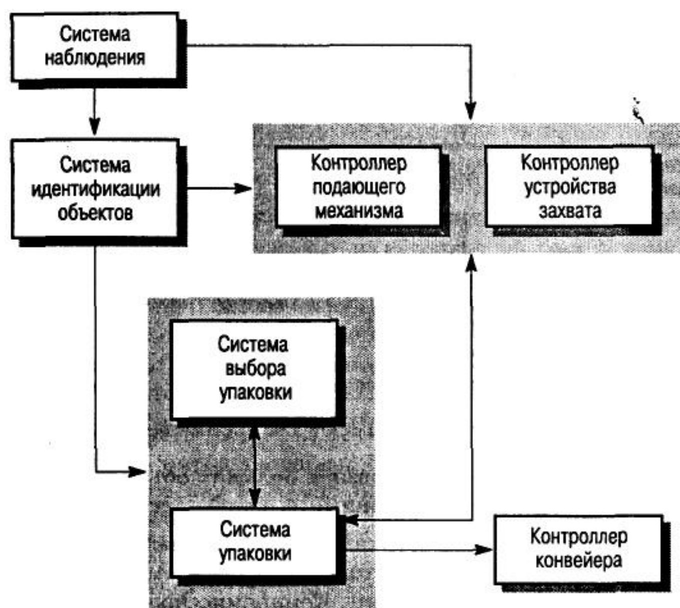

Рис. 7.1. Блок-схема системы управления автоматической упаковкой

Конечно, можно разрабатывать более детализированные модели структуры, в которых было бы показано, как именно подсистемы разделяют данные и как взаимодействуют друг с другом. В этом разделе рассматриваются три стандартные модели, а именно: модель репозитория, модель клиент/сервер и модель абстрактной машины.

7.2. Модель репозитория

Для того чтобы подсистемы, составляющие систему, работали эффективнее, между ними должен идти обмен информацией. Обмен можно организовать двумя способами.

1. Все совместно используемые данные хранятся в центральной базе данных, доступной всем подсистемам. Модель системы, основанная на совместном использовании базы данных, часто называют *моделью репозитория.*

2. Каждая подсистема имеет собственную базу данных. Взаимообмен данными между подсистемами происходит посредством передачи сообщений.

Большинство систем, обрабатывающих большие объемы данных, организованы вокруг совместно используемой базы данных, или репозитория. Поэтому *такая* модель подойдет к приложениям, в которых данные создаются в одной подсистеме, а используются в другой. Примерами могут служить системы управления информацией, системы автоматического проектирования и CASE-средства.

На рис. 7.2 представлен пример архитектуры интегрированного набора CASE-инструментов, основанный на совместно используемом репозитории. Считается, что для CASE-средств первый совместно используемый репозитории был разработан в начале 1970-х годов английской компанией ICL в процессе создания своей операционной системы [234]. Широкую известность эта модель получила после того, как была применена для поддержки разработки систем, написанных на языке Ada. С тех пор многие CASE-средства разрабатываются с использованием общего репозитория.

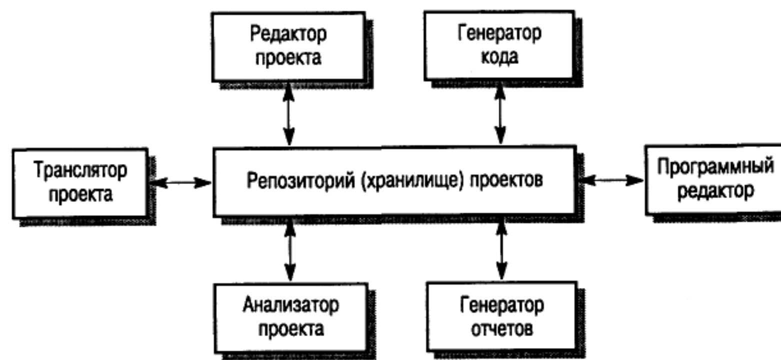

Рис. 7.2. Архитектура интегрированного набора CASE-средств

Совместно используемые репозитории имеют как преимущества, так и недостатки.

1. Очевидно, что совместное использование больших объемов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие.

2. С другой стороны, подсистемы должны быть согласованы с моделью репозитория данных. Это всегда приводит к необходимости компромисса между требованиями, предъявляемыми к каждой подсистеме. Компромиссное решение может понизить их производительность. Если форматы данных новых подсистем не подходят под согласованную модель представления данных, интегрировать такие подсистемы сложно или невозможно.

3. Подсистемам, в которых создаются данные, не нужно знать, как эти данные используются в других подсистемах.

4. Поскольку в соответствии с согласованной моделью данных генерируются большие объемы информации, модернизация таких систем проблематична. Перевод системы на новую модель данных будет дорогостоящим и сложным, а порой даже невозможным.

5. В системах с репозиторием такие средства, как резервное копирование, обеспечение безопасности, управление доступом и восстановление данных, централизованы, поскольку входят в систему управления репозиторием. Эти средства выполняют только свои основные операции и не занимаются другими вопросами.

6. С другой стороны, к разным подсистемам предъявляются разные требования, касающиеся безопасности, восстановления и резервирования данных. В модели репозитория ко всем подсистемам применяется одинаковая политика.

7. Модель совместного использования репозитория прозрачна: если новые подсистемы совместимы с согласованной моделью данных, их можно непосредственно интегрировать в систему.

8. Однако сложно разместить репозитории на нескольких машинах, поскольку могут возникнуть проблемы, связанные с избыточностью и нарушением целостности данных.

В рассматриваемой модели репозитории является пассивным элементом, а управление им возложено на подсистемы, использующие данные из репозитория. Для систем искусственного интеллекта разработан альтернативный подход. Он основан на модели "рабочей области", которая инициирует подсистемы тогда, когда конкретные данные становятся доступными. Такой подход применим к системам, в которых форма данных хорошо структурирована. Эта модель обсуждается в работе [255].

7.3. Модель клиент/сервер

Модель архитектуры клиент/сервер – это модель распределенной системы, в которой показано распределение данных и процессов между несколькими процессорами. Модель включает три основных компонента.

1. Набор автономных серверов, предоставляющих сервисы другим подсистемам. Например, сервер печати, который предоставляет услуги печати, файловые серверы, предоставляющие сервисы управления файлами, и сервер-компилятор, который предлагает сервисы по компилированию исходных кодов программ.

2. Набор клиентов, которые вызывают сервисы, предоставляемые серверами. В контексте системы клиенты являются обычными подсистемами.

Допускается параллельное выполнение нескольких экземпляров клиентской программы.

3. Сеть, посредством которой клиенты получают доступ к сервисам. В принципе нет никакого запрета на то, чтобы клиенты и серверы запускались на одной машине. На практике, однако, модель клиент/сервер в такой ситуации не используется.

Клиенты должны знать имена доступных серверов и сервисов, которые они предоставляют. В то же время серверам не нужно знать ни имена клиентов, ни их количество. Клиенты получают доступ к сервисам, предоставляемым сервером, посредством удаленного вызова процедур.

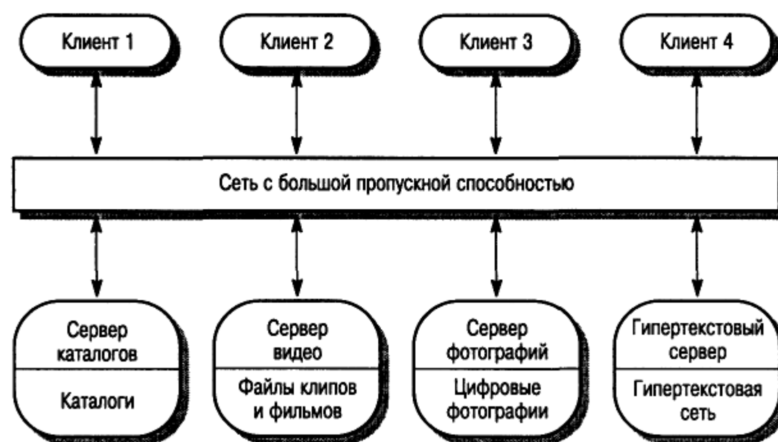

Рис. 7.3. Архитектура библиотечной системы фильмов и фотографий

Пример системы, организованной по типу модели клиент/сервер, показан на рис. 7.3. Это многопользовательская гипертекстовая система, предназначенная для поддержки библиотек фильмов и фотографий. В ней содержится несколько серверов, которые размещают различные типы медиафайлов и управляют ими. Видеофайлы требуется передавать быстро и синхронно, но с относительно малым разрешением. Они могут храниться в сжатом состоянии. Фотографии должны передаваться с высоким разрешением. Каталоги должны обеспечивать работу с множеством запросов и поддерживать связи с использованием гипертекстовой системы. Здесь клиентская программа является просто интегрированным интерфейсом пользователя.

Подход клиент/сервер можно использовать при реализации систем, основанных на репозитории, который поддерживается как сервер системы. Подсистемы, имеющие доступ к репозиторию, являются клиентами. Но обычно каждая подсистема управляет собственными данными. Во время работы серверы и клиенты обмениваются данными, однако при обмене большими объемами данных могут возникнуть проблемы, связанные с пропускной способностью сети. Правда, с развитием все более быстрых сетей эта проблема теряет свое значение.

Наиболее важное преимущество модели клиент/сервер состоит в том, что она является распределенной архитектурой. Ее эффективно использовать в сетевых системах с множеством распределенных процессоров. В систему легко добавить новый сервер и интегрировать его с остальной частью системы или же обновить серверы, не воздействуя на другие части системы. В главе 11 архитектуры распределенных систем рассматриваются более подробно.

7.4. Модель абстрактной машины

Модель архитектуры абстрактной машины (иногда называемая многоуровневой моделью) моделирует взаимодействие подсистем. Она организует систему в виде набора уровней, каждый из которых предоставляет свои сервисы. Каждый уровень определяет *абстрактную машину,* машинный язык которой (сервисы, предоставляемые уровнем) используется для реализации следующего уровня абстрактной машины. Например, наиболее распространенный способ реализации языка программирования состоит в определении идеальной "языковой машины" и компилировании программ, написанных на данном языке, в код этой машины. На следующем шаге трансляции код абстрактной машины конвертируется в реальный машинный код.

Хорошо известным примером такого похода может служить модель OSI* сетевых протоколов [352], обсуждаемая в разделе 7.4. Другим примером является трехуровневая модель среды программирования на языке Ada [66]. На рис. 7.4 изображена подобная модель и показано, как с помощью модели абстрактной машины можно представить систему администрирования версий.

Система администрирования версий основана на управлении версиями объектов и предоставляет средства для полного управления конфигурацией системы (см. главу 29). Для поддержки средств управления конфигурацией используется система администрирования объектов, поддерживающая систему базы данных и сервисы управления объектами. В свою очередь, в системе баз данных поддерживаются различные сервисы, например управления транзакциями, отката назад, восстановления и управления доступом. Для управления базами данных используются средства основной операционной системы и ее файловая система.

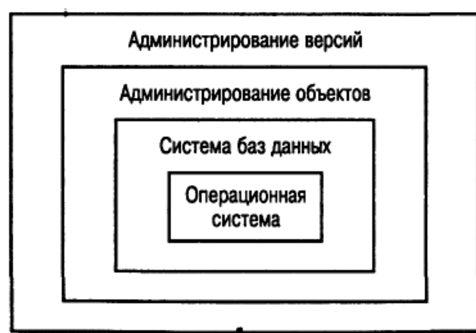

Рис. 7.4. Модель абстрактной машины для системы администрирования версий

Многоуровневый подход обеспечивает пошаговое развитие систем – при разработке какого-либо уровня предоставляемые им сервисы становятся доступны пользователям. Кроме того, такая архитектура легко изменяема и переносима на разные платформы. Изменение интерфейса любого уровня повлияет только на смежный уровень. Так как в многоуровневых системах зависимости от машинной платформы локализованы на внутренних уровнях,

такие системы можно реализовать на других платформах, поскольку потребуется изменить только самые внутренние уровни.

Недостатком многоуровневого подхода является довольно сложная структура системы. Основные средства, такие как управление файлами, необходимые всем абстрактным машинам, предоставляются внутренними уровнями. Поэтому сервисам, запрашиваемым пользователем, возможно, потребуется доступ к внутренним уровням абстрактной машины. Такая ситуация приводит к разрушению модели, так как внешний уровень зависит не только от предшествующего ему уровня, но и от более низких уровней.

7.5. Модели управления

В модели структуры системы показаны все подсистемы, из которых она состоит. Для того чтобы подсистемы функционировали как единое целое, необходимо управлять ими. В структурных моделях нет (и не должно быть) никакой информации по управлению. Однако разработчик архитектуры должен организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры. В моделях управления на уровне архитектуры проектируется поток управления между подсистемами.

Можно выделить два основных типа управления в программных системах.

1. *Централизованное управление.* Одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем. Управление от первой подсистемы может перейти к другой подсистеме, однако потом обязательно возвращается к первой.

2. *Управление, основанное на событиях.* Здесь вместо одной подсистемы, ответственной за управление, на внешние события может отвечать любая подсистема. События, на которые реагирует система, могут

происходить либо в других подсистемах, либо во внешнем окружении системы.

Модель управления дополняет структурные модели. Все описанные ранее структурные модели можно реализовать с помощью централизованного управления или управления, основанного на событиях.

7.6. Централизованное управление

В модели централизованного управления одна из систем назначается главной и управляет работой других подсистем. Такие модели можно разбить на два класса, в зависимости от того, последовательно или параллельно реализовано выполнение управляемых подсистем.

1. *Модель вызова-возврата.* Это известная модель организации вызова программных процедур "сверху вниз", в которой управление начинается на вершине иерархии процедур и через вызовы передается на более нижние уровни иерархии. Данная модель применима только в последовательных системах.

2. *Модель диспетчера.* Применяется в параллельных системах. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс (выполняемая подсистема или модуль) может протекать параллельно с другими процессами. Модель такого типа применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния. Обычно такое управление реализуется через оператор case.

Модель вызова-возврата представлена на рис. 7.5. Из главной программы можно вызвать подпрограммы 1, 2 и 3, из подпрограммы 1 – подпрограммы 1.1 и 1.2, из подпрограммы 3 – подпрограммы 3.1 и 3.2 и т.д. Такая модель выполнения подпрограмм *не является структурной* – подпрограмма 1.1 не обязательно является частью подпрограммы 1.

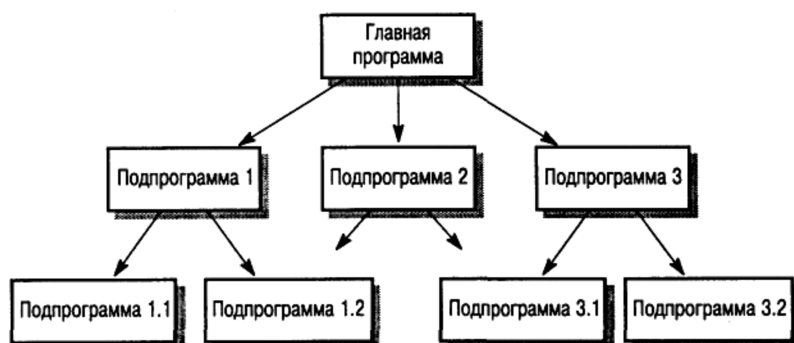

Рис. 7.5. Модель вызова-возврата

Подобная модель встроена в языки программирования Ada, Pascal и С. Управление переходит от программы, расположенной на самом верхнем уровне иерархии, к подпрограмме более нижнего уровня. Затем происходит возврат управления в точку вызова подпрограммы. За управление отвечает та подпрограмма, которая выполняется в текущий момент; она может либо вызывать другие подпрограммы, либо вернуть управление вызвавшей ее подпрограмме. Несовершенство данного стиля программирования при возврате к определенной точке в программе очевидно.

Модель вызова-возврата можно использовать на уровне модулей для управления функциями и объектами. Подпрограммы в языке программирования, которые вызываются из других подпрограмм, являются естественно функциональными. Однако во многих объектно-ориентированных системах операции в объектах (методы) реализованы в виде процедур или функций. Например, объект Java запрашивает сервис из другого объекта посредством вызова соответствующего метода.

Жесткая и ограниченная природа модели вызова-возврата является одновременно и преимуществом и недостатком. Преимущества модели проявляются в относительно простом анализе потоков управления, а также при выборе системы, отвечающей за конкретный ввод данных. Недостаток модели, как вы узнаете из главы 18, состоит в сложной обработке исключительных ситуаций.

На рис. 10.6 представлена модель централизованного управления для параллельной системы. Подобная модель часто используется в "мягких"

системах реального времени, в которых нет чересчур строгих временых. ограничений. Центральный контроллер управляет выполнением множества процессов, связанных с датчиками и исполнительными механизмами. Система, использующая такую модель управления, рассмотрена в главе 13.

Рис. 7.6. Модель централизованного управления для системы реального времени

Контроллер системы, в зависимости от переменных состояния системы, определяет моменты запуска или завершения процессов. Он проверяет, генерируется ли в остальных процессах информация, для того чтобы затем обработать ее или передать другим процессам на обработку. Обычно контроллер работает постоянно, проверяя датчики и другие процессы или отслеживая изменения состояния, поэтому данную модель иногда называют моделью с обратной связью.

Вопросы для самопроверки и контроля

1. Что такое архитектурное проектирование?
2. Какие существуют этапы архитектурного проектирования?
3. Какие разрабатываются архитектурные модели?
4. Каковы достоинства и недостатки совместно используемых репозиториев?
5. Дать определение модели клиент-сервер, какие компоненты эта модель включает?
6. Что такое модель архитектуры абстрактной машины?
7. Какие существуют типы управления в программных системах?
8. Какие существуют модели централизованного управления?

Ключевые слова: *архитектурное проектирование, архитектура программного обеспечения, структурирование системы, моделирование управления, модульная декомпозиция, подсистема, модуль, производительность, защищенность, безопасность, надежность, удобство сопровождения, модель репозитория, модель архитектуры клиент/сервер, модель архитектуры абстрактной машины, централизованное управление, управление, основанное на событиях, модель вызова-возврата, модель диспетчера.*

Keywords: *architectural design, software architecture, system structuring, modeling management, modular decomposition, subsystem, module, performance, security, safety, reliability, maintainability, the model repository, model client / server architecture model of the abstract machine, centralized management management, based on events, call-return model, the model of the controller.*

Kalit so'zlar: *arhitekturaviy loyihalashtirish, dastriy ta'minot arhitekturasi, tizim strukturasi, boshqaruvni modellashtirish, modul dekompozitsiya, tizim bo'lagi, modul, unumdorlik, himoyalanish, havfsizlik, ishonchlilik, kuzatuv qulayligi, repozitoriy modeli, klient/server arhitekturasi modeli, abstrakt mashina arhitekturasi modeli, markazlashtirilgan boshqaruv, hodisalarga asoslangan boshqaruv, chaqiruv-qaytaruv modeli, dispetcher modeli.*

Упражнения

1. Объясните, почему архитектуру системы необходимо разработать до окончания создания спецификации.

2. Создайте таблицу, описывающую преимущества и недостатки различных структурных моделей, обсуждавшихся в данной главе.

3. Предложите подходящую структурную модель для перечисленных ниже систем. Обоснуйте свой выбор.

   - Система автоматической продажи железнодорожных билетов.
   - Система видеоконференций, управляемая компьютером, с возможностью одновременного просмотра компьютерных, аудио- и видеоданных несколькими участниками.
   - Робот-уборщик, который убирает относительно свободные пространства, например коридоры. Робот должен фиксировать стены и другие преграды.

4. На основе выбранной модели разработайте архитектуру для систем из предыдущего упражнения. Сделайте предположения о системных требованиях.

5. Объясните, почему модель управления вызова-возврата обычно не подходит для систем реального времени, управляющих определенным процессом.

6. Предложите подходящую модель управления для перечисленных ниже систем. Обоснуйте свой выбор.

- Пакетная система обработки данных, которая на основании информации об отработанных часах и ставок заработной платы формирует заявки на зарплату персонала и передает информацию в банк.
- Набор инструментальных программных средств от разных производителей, которые должны работать совместно.
- Телевизионный контроллер, который отвечает на сигналы, поступающие от удаленного блока управления.

7. Обсудите преимущества и недостатки модели потоков данных и объектной модели в предположении, что необходимо разработать как локальную, так и распределенную версии программного приложения.

8. Существует два набора инструментальных CASE-средств. Необходимо сравнить их. Продумайте, как это сделать с помощью базовой модели CASE-средств [61].

9. Предположим, существует конкретная должность "архитектор программного обеспечения"; его роль состоит в проектировании системной архитектуры независимо от того, для какого заказчика выполняется данный проект. Такая должность может быть, например, в компании, занимающейся разработкой ПО. Какие трудности могут возникнуть при введении данной должности?

ГЛАВА 8. АРХИТЕКТУРА РАСПРЕДЕЛЕННЫХ СИСТЕМ

В настоящее время практически все большие программные системы являются распределенными. Распределенной называется такая система, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами. При проектировании распределенных систем, которое имеет много общего с проектированием любого другого ПО, все же следует учитывать ряд специфических особенностей. Некоторые из них уже упоминалось во введении к главе 10 при рассмотрении архитектуры клиент/сервер, здесь они обсуждаются более подробно.

Поскольку в наши дни распределенные системы получили широкое распространение, разработчики ПО должны быть знакомы с особенностями их проектирования. До недавнего времени все большие системы в основном являлись централизованными, которые запускались на одной главной вычислительной машине (мэйнфрейме) с подключенными к ней терминалами. Терминалы практически не занимались обработкой информации – все вычисления выполнялись на главной машине. Разработчикам таких систем не приходилось задумываться о проблемах распределенных вычислений.

Все современные программные системы можно разделить на три больших класса.

1. Прикладные программные системы, предназначенные для работы только на одном персональном компьютере или рабочей станции. К ним относятся текстовые процессоры, электронные таблицы, графические системы и т.п.

2. Встроенные системы, предназначенные для работы на одном процессоре либо на интегрированной группе процессоров. К ним относятся системы управления бытовыми устройствами, различными приборами и др.

3. Распределенные системы, в которых программное обеспечение выполняется на слабо интегрированной группе параллельно работающих процессоров, связанных через сеть. К ним относятся системы банкоматов, принадлежащих какому-либо банку, издательские системы, системы ПО коллективного пользования и др.

В настоящее время между перечисленными классами программных систем существуют четкие границы, которые в дальнейшем будут все более стираться. Со временем, когда высокоскоростные беспроводные сети станут широкодоступными, появится возможность динамически интегрировать устройства со встроенными программными системами, например электронные органайзеры с более общими системами.

В книге [81] выделено шесть основных характеристик распределенных систем.

1. *Совместное использование ресурсов.* Распределенные системы допускают совместное использование аппаратных и программных ресурсов, например жестких дисков, принтеров, файлов, компиляторов и т.п., связанных посредством сети. Очевидно, что разделение ресурсов возможно также в многопользовательских системах, однако в этом случае за предоставление ресурсов и их управление должен отвечать центральный компьютер.

2. *Открытость.* Это возможность расширять систему путем добавления новых ресурсов. Распределенные системы – это открытые системы, к которым подключают аппаратное и программное обеспечение от разных производителей.

3. *Параллельность.* В распределенных системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут (но не обязательно) взаимодействовать друг с другом во время их выполнения.

4. *Масштабируемость.* В принципе все распределенные системы являются масштабируемыми: чтобы система соответствовала новым требованиям, ее можно наращивать посредством добавления новых вычислительных ресурсов. Но на практике наращивание может ограничиваться сетью, объединяющей отдельные компьютеры системы. Если подключить много новых машин, пропускная способность сети может оказаться недостаточной.

5. *Отказоустойчивость.* Наличие нескольких компьютеров и возможность дублирования информации означает, что распределенные системы устойчивы к определенным аппаратным и программным ошибкам (см. главу 18). Большинство распределенных систем в случае ошибки, как правило, могут поддерживать хотя бы частичную функциональность. Полный сбой в работе системы происходит только в случае сетевых ошибок.

6. *Прозрачность.* Это свойство означает, что пользователям предоставлен полностью прозрачный доступ к ресурсам и в то же время от них скрыта информация о распределении ресурсов в системе. Однако во многих случаях конкретные знания об организации системы помогают пользователю лучше использовать ресурсы.

Разумеется, распределенным системам присущ ряд недостатков.

- *Сложность.* Распределенные системы сложнее централизованных. Намного труднее понять и оценить свойства распределенных систем в целом, а также тестировать эти системы. Например, здесь производительность системы зависит не от скорости работы одного процессора, а от полосы пропускания сети и скорости работы разных процессоров. Перемещая ресурсы из одной части системы в другую, можно радикально повлиять на производительность системы.
- *Безопасность.* Обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться или перехватываться. Поэтому, в распределенной системе намного сложнее поддерживать безопасность.
- *Управляемость.* Система может состоять из разнотипных компьютеров, на которых могут быть установлены разные версии операционных систем. Ошибки на одной машине могут распространиться на другие машины с непредсказуемыми последствиями. Поэтому требуется значительно больше усилий, чтобы управлять и поддерживать систему в рабочем состоянии.
- *Непредсказуемость.* Как известно всем пользователям Web-сети, реакция распределенных систем на определенные события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как все эти параметры могут постоянно меняться, время, затраченное на выполнение запроса пользователя, в тот или иной момент может существенно различаться.

При обсуждении преимуществ и недостатков распределенных систем в книге [81] определяется ряд критических проблем проектирования таких систем (табл. 8.1). В этой главе основное внимание уделяется архитектуре распределенного ПО, так как я полагаю, что при разработке программных продуктов наиболее значимым является именно этот момент. Если вас интересуют другие темы, обратитесь к специализированным книгам по распределенным системам.

Таблица 8.1. Проблемы проектирования распределенных систем

Проблема проектирования	Описание
Идентификация ресурсов	Ресурсы в распределенной системе располагаются на разных компьютерах, поэтому систему имен ресурсов следует продумать так, чтобы пользователи могли без труда открывать необходимые им ресурсы и ссылаться на них. Примером может служить система унифицированного указателя ресурсов URL, которая определяет адреса Web-страниц. Без легковоспринимаемой и универсальной системы идентификации большая часть ресурсов окажется недоступной пользователям системы
Коммуникации	Универсальная работоспособность Internet и эффективная реализация протоколов TCP/IP в Internet для большинства распределенных систем служат примером наиболее эффективного способа организации взаимодействия между компьютерами. Однако там, где на производительность, надежность и прочее накладываются специальные требования, можно воспользоваться альтернативными способами системных коммуникаций
Качество системного сервиса	Качество сервиса, предлагаемое системой, отражает ее производительность, работоспособность и надежность. На качество сервиса влияет целый ряд факторов: распределение системных процессов, распределение ресурсов, системные и сетевые аппаратные средства и возможности адаптации системы
Архитектура программного обеспечения	Архитектура программного обеспечения описывает распределение системных функций по компонентам системы, а также распределение этих компонентов по

процессорам. Если необходимо поддерживать высокое качество системного сервиса, выбор правильной архитектуры оказывается решающим фактором

Задача разработчиков распределенных систем – спроектировать программное или аппаратное обеспечение так, чтобы предоставить все необходимые характеристики распределенной системы. А для этого требуется знать преимущества и недостатки различных архитектур распределенных систем. Здесь выделяется два родственных типа архитектур распределенных систем.

1. *Архитектура клиент/сервер.* В этой модели систему можно представить как набор сервисов, предоставляемых серверами клиентам. В таких системах серверы и клиенты значительно отличаются друг от друга.

2. *Архитектура распределенных объектов.* В этом случае между серверами и клиентами нет различий и систему можно представить как набор взаимодействующих объектов, местоположение которых не имеет особого значения. Между поставщиком сервисов и их пользователями не существует различий.

В распределенной системе разные системные компоненты могут быть реализованы на разных языках программирования и выполняться на разных типах процессоров. Модели данных, представление информации и протоколы взаимодействия – все это не обязательно будет однотипным в распределенной системе. Следовательно, для распределенных систем необходимо такое программное обеспечение, которое могло бы управлять этими разнотипными частями и гарантировать взаимодействие и обмен данными между ними. *Промежуточное программное обеспечение* относится именно к такому классу ПО. Оно находится как бы посередине между разными частями распределенных компонентов системы.

В статье [37] описаны различные типы промежуточного ПО, которое может поддерживать распределенные вычисления. Как правило, такое ПО

составляется из готовых компонентов и не требует от разработчиков специальных доработок. В качестве примеров промежуточного ПО можно привести программы управления взаимодействием с базами данных, менеджеры транзакций, преобразователи данных, коммуникационные инспекторы и др. Далее в главе будет описана структура распределенных систем как класс промежуточного ПО.

Распределенные системы обычно разрабатываются на основе объектно-ориентированного подхода. Эти системы создаются из слабо интегрированных частей, каждая из которых может непосредственно взаимодействовать как с пользователем, так и с другими частями системы. Эти части по возможности должны реагировать на независимые события. Программные объекты, построенные на основе таких принципов, являются естественными компонентами распределенных систем. Если вы еще не знакомы с концепцией объектов, рекомендую сначала прочитать главу 12, а затем вновь вернуться к данной главе.

8.1. Многопроцессорная архитектура

Самой простой распределенной системой является многопроцессорная система. Она состоит из множества различных процессов, которые могут (но не обязательно) выполняться на разных процессорах. Данная модель часто используется в больших системах реального времени. Как вы узнаете из главы 13, эти системы собирают информацию, принимают на ее основе решения и отправляют сигналы исполнительному механизму, который изменяет системное окружение. В принципе все процессы, связанные со сбором информации, принятием решений и управлением исполнительным механизмом, могут выполняться на одном процессоре под управлением планировщика заданий. Использование нескольких процессоров повышает производительность системы и ее способность к восстановлению. Распределение процессов между процессорами может переопределяться

(присуще критическим системам) или же находиться под управлением диспетчера процессов.

На рис. 8.1 показан пример системы такого типа. Это упрощенная модель системы управления транспортным потоком. Группа распределенных датчиков собирает информацию о величине потока. Собранные данные перед отправкой в диспетчерскую обрабатываются на месте. На основании полученной информации операторы принимают решения и управляют светофорами. В этом примере для управления датчиками, диспетчерской и светофорами имеются отдельные логические процессы. Это могут быть как отдельные процессы, так и группа процессов. В нашем примере они выполняются на разных процессорах.

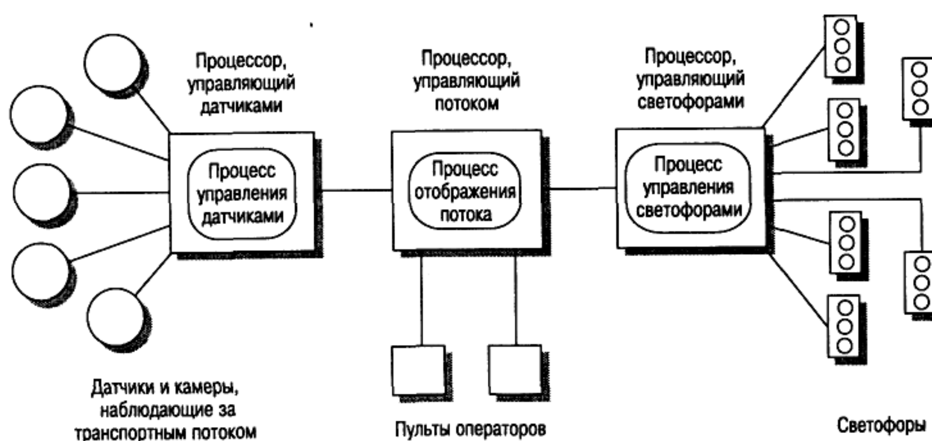

Рис. 8.1. Многопроцессорная система управления движением транспорта

Системы ПО, одновременно выполняющие множество процессов, не обязательно являются распределенными. Если в системе более одного процессора, реализовать распределение процессов не представляет труда. Однако при создании многопроцессорных программных систем не обязательно отталкиваться только от распределенных систем. При проектировании систем такого типа, по существу, используется тот же подход, что и при проектировании систем реального времени, которые рассматриваются в главе 13.

8.2. Архитектура клиент/сервер

В главе 10 уже рассматривалась концепция клиент/сервер. В архитектуре клиент/сервер программное приложение моделируется как набор сервисов, предоставляемых серверами, и множество клиентов,

использующих эти сервисы [264, 2*]. Клиенты должны знать о доступных (имеющихся) серверах, хотя могут и не иметь представления о существовании других клиентов. Как видно из рис. 8.2, на котором представлена схема распределенной архитектуры клиент/сервер, клиенты и серверы представляют разные процессы.

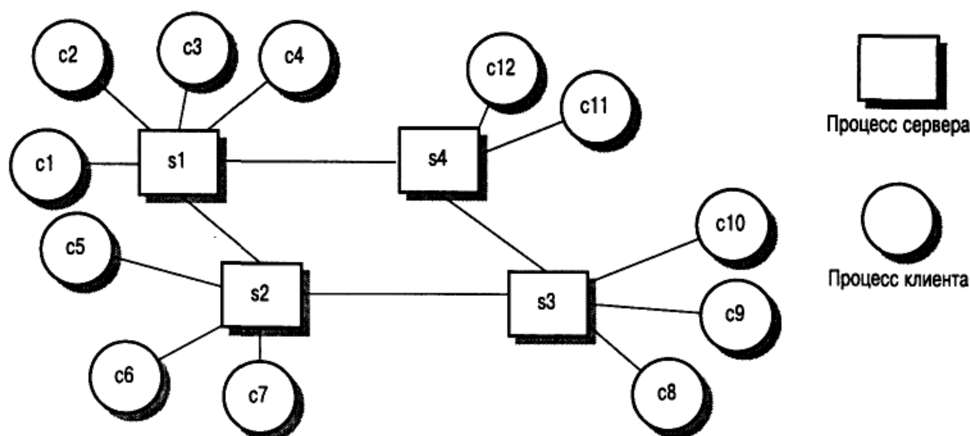

Рис. 8.2. Система клиент/сервер

В системе между процессами и процессорами не обязательно должно соблюдаться отношение "один к одному". На рис. 8.3 показана физическая архитектура системы, которая состоит из шести клиентских машин и двух серверов. На них запускаются клиентские и серверные процессы, изображенные на рис. 8.2. В общем случае, говоря о клиентах и серверах, я подразумеваю скорее логические процессы, чем физические машины, на которых выполняются эти процессы.

Архитектура системы клиент/сервер должна отражать логическую структуру разрабатываемого программного приложения. На рис. 8.4 предлагается еще один взгляд на программное приложение, структурированное в виде трех уровней. Уровень представления обеспечивает информацию для пользователей и взаимодействие с ними. Уровень выполнения приложения реализует логику работы приложения. На уровне управления данными выполняются все операции с базами данных. В централизованных системах между этими уровнями нет четкого разделения. Однако при проектировании распределенных систем необходимо разделять

эти уровни, чтобы затем расположить каждый уровень на разных компьютерах.

Самой простой архитектурой клиент/сервер является двухуровневая, в которой приложение состоит из сервера (или множества идентичных серверов) и группы клиентов. Существует два вида такой архитектуры (рис. 8.5).

1. *Модель тонкого клиента.* В этой модели вся работа приложения и управление данными выполняются на сервере. На клиентской машине запускается только ПО уровня представления.

2. *Модель толстого клиента.* В этой модели сервер только управляет данными. На клиентской машине реализована работа приложения и взаимодействие с пользователем системы.

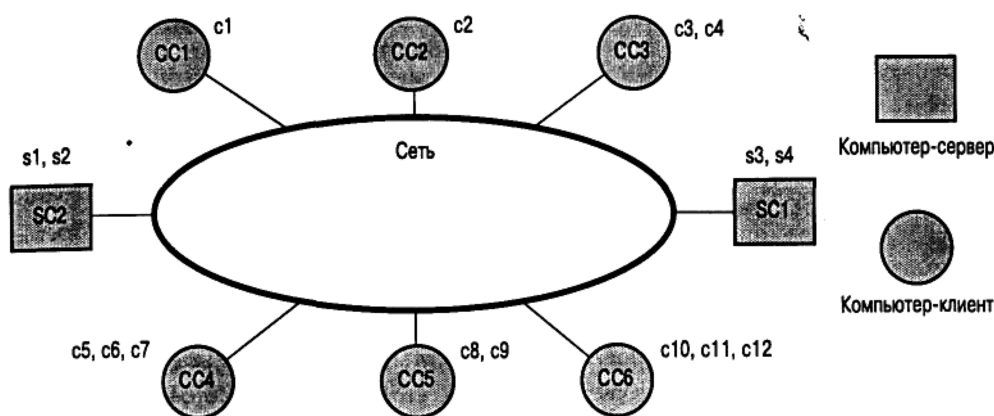

Рис. 8.3. Компьютеры в сети клиент/сервер

Рис. 8.4. Уровни программного приложения

Тонкий клиент двухуровневой архитектуры – самый простой способ перевода существующих централизованных систем (см. главу 26) в архитектуру клиент/сервер. Пользовательский интерфейс в этих системах

"переселяется" на персональный компьютер, а само программное приложение выполняет функции сервера, т.е. выполняет все процессы приложения и управляет данными. Модель тонкого клиента можно также реализовать там, где клиенты представляют собой обычные сетевые устройства, а не персональные компьютеры или рабочие станции. Сетевые устройства запускают Internet-броузер и пользовательский интерфейс, реализованный внутри системы.

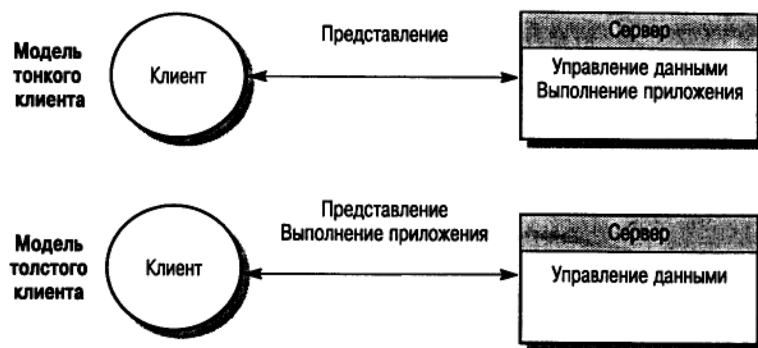

Рис. 8.5. Модели тонкого и толстого клиентов

Главный недостаток модели тонкого клиента – большая загруженность сервера и сети. Все вычисления выполняются на сервере, а это может привести к значительному сетевому трафику между клиентом и сервером. В современных компьютерах достаточно вычислительной мощности, но она практически не используется в модели тонкого клиента банка.

Напротив, модель толстого клиента использует вычислительную мощность локальных машин: и уровень выполнения приложения, и уровень представления помещаются на клиентский компьютер. Сервер здесь, по существу, является сервером транзакций, который управляет всеми транзакциями баз данных. Примером архитектуры такого типа могут служить системы банкоматов, в которых банкомат является клиентом, а сервер – центральным компьютером, обслуживающим базу данных по расчетам с клиентами.

На рис. 8.6 показана сетевая система банкоматов. Заметим, что банкоматы связаны с базой данных расчетов не напрямую, а через монитор телеобработки. Этот монитор является промежуточным звеном, которое

взаимодействует с удаленными клиентами и организует запросы клиентов в последовательность транзакций для работы с базой данных. Использование последовательных транзакций при возникновении сбоев позволяет системе восстановиться без потери данных.

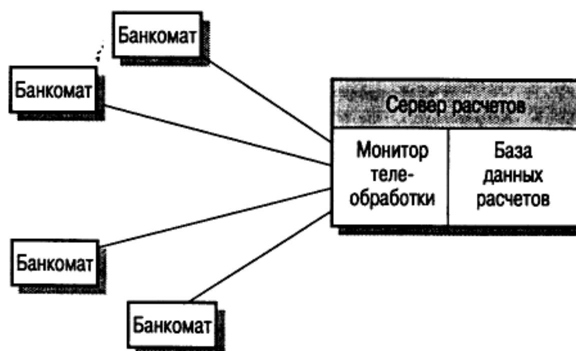

Рис. 8.6. Система клиент/сервер для сети банкоматов

Поскольку в модели толстого клиента выполнение программного приложения организовано более эффективно, чем в модели тонкого клиента, управлять такой системой сложнее. Здесь функции приложения распределены между множеством разных машин. Необходимость замены приложения приводит к его повторной инсталляции на всех клиентских компьютерах, что требует больших расходов, если в системе сотни клиентов.

Появление языка Java и загружаемых аплетов позволили разрабатывать модели клиент/сервер, которые находятся где-то посередине между моделями тонкого и толстого клиента. Часть программ, составляющих приложение, можно загружать на клиентской машине как аплеты Java и тем самым разгрузить сервер. Интерфейс пользователя строится посредством Web-броузера, который запускает аплеты Java. Однако Web-броузеры от различных производителей и даже различные версии Web-броузеров от одного производителя не всегда выполняются одинаково. Более ранние версии броузеров на старых машинах не всегда могут запустить аплеты Java. Следовательно, такой подход можно использовать только тогда, когда вы уверены, что у всех пользователей системы установлены броузеры, совместимые с Java.

В двухуровневой модели клиент/сервер существенной проблемой является размещение на двух компьютерных системах трех логических уровней – представления, выполнения приложения и управления данными. Поэтому в данной модели часто возникают либо проблемы с масштабируемостью и производительностью, если выбрана модель тонкого клиента, либо проблемы, связанные с управлением системой, если используется модель толстого клиента. Чтобы избежать этих проблем, необходимо применить альтернативный подход – трехуровневую модель архитектуры клиент/сервер (рис. 8.7). В этой архитектуре уровням представления, выполнения приложения и управления данными соответствуют отдельные процессы.

Рис. 8.7. Трехуровневая архитектура клиент/сервер

Архитектура ПО, построенная по трехуровневой модели клиент/сервер, не требует, чтобы в сеть были объединены три компьютерных системы. На одном компьютере-сервере можно запустить и выполнение приложения, и управление данными как отдельные логические серверы. В то же время, если требования к системе возрастут, можно будет относительно просто разделить выполнение приложения и управление данными и выполнять их на разных процессорах.

Банковскую систему, использующую Internet-сервисы, можно реализовать с помощью трехуровневой архитектуры клиент/сервер. База данных расчетов (обычно расположенная на главном компьютере) предоставляет сервисы управления данными, Web-сервер поддерживает сервисы приложения, например средства перевода денег, генерацию отчетов, оплату счетов и др. А компьютер пользователя с Internet-броузером является клиентом. Как показано на рис. 8.8, эта система масштабируема, так как в нее

относительно просто добавить новые Web-серверы при увеличении количества клиентов.

Использование трехуровневой архитектуры в этом примере позволило оптимизировать передачу данных между Web-сервером и сервером базы данных. Взаимодействие между этими системами не обязательно строить на стандартах Internet, можно использовать более быстрые коммуникационные протоколы низкого уровня. Обычно информацию от базы данных обрабатывает эффективное промежуточное ПО, которое поддерживает запросы к базе данных на языке структурированных запросов SQL.

В некоторых случаях трехуровневую модель клиент/сервер можно перевести в многоуровневую, добавив в систему дополнительные серверы. Многоуровневые системы можно использовать и там, где приложениям необходимо иметь доступ к информации, находящейся в разных базах данных. В этом случае объединяющий сервер располагается между сервером, на котором выполняется приложение, и серверами баз данных. Объединяющий сервер собирает распределенные данные и представляет их в приложении таким образом, будто они находятся в одной базе данных.

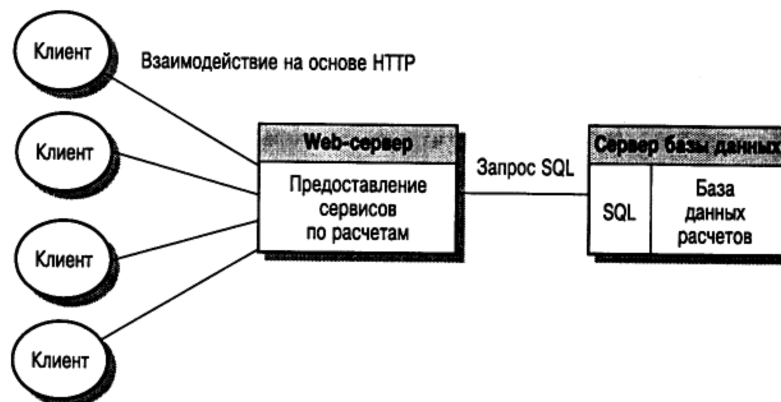

Рис. 8.8. Распределенная архитектура банковской системы с использованием Internet-сервисов

Разработчики архитектур клиент/сервер, выбирая наиболее подходящую, должны учитывать ряд факторов. В табл. 8.2 перечислены различные случаи применения архитектуры клиент/сервер.

Таблица 8.2. Применение разных типов архитектуры клиент/сервер

Архитектура	Приложения
Двухуровневая архитектура тонкого клиента	Наследуемые системы, в которых нецелесообразно разделять выполнение приложения и управления данными. Приложения с интенсивными вычислениями, например компиляторы, но с незначительным объемом управления данными. Приложения, в которых обрабатываются большие массивы данных (запросы), но с небольшим объемом вычислений в самом приложении
Двухуровневая архитектура толстого клиента	Приложения, где пользователю требуется интенсивная обработка данных (например, визуализация данных или большие объемы вычисления). Приложения с относительно постоянным набором функций на стороне пользователя, применяемых в среде с хорошо отлаженным системным управлением
Трехуровневая и многоуровневая архитектуры клиент/сервер	Большие приложения с сотнями и тысячами клиентов. Приложения, в которых часто меняются и данные, и методы обработки. Приложения, в которых выполняется интеграция данных из многих источников

8.3. Архитектура распределенных объектов

В модели клиент/сервер распределенной системы между клиентами и серверами существуют различия. Клиент запрашивает сервисы только у сервера, hq не у других клиентов; серверы могут функционировать как клиенты и запрашивать сервисы у других серверов, но не у клиентов; клиенты должны знать о сервисах, предоставляемых определенными серверами, и о том, как взаимодействуют эти серверы. Такая модель отлично

подходит ко многим типам приложений, но в то же время ограничивает разработчиков системы, которые вынуждены решать, где предоставлять сервисы. Они также должны обеспечить поддержку масштабируемости и разработать средства включения клиентов в систему на распределенных серверах.

Более общим подходом, применяемым в проектировании распределенных систем, является стирание различий между клиентом и сервером и проектирование архитектуры системы как архитектуры распределенных объектов. В этой архитектуре (рис. 8.9) основными компонентами системы являются объекты, предоставляющие набор сервисов через свои интерфейсы. Другие объекты вызывают эти сервисы, не делая различий между клиентом (пользователем сервиса) и сервером (поставщиком сервиса).

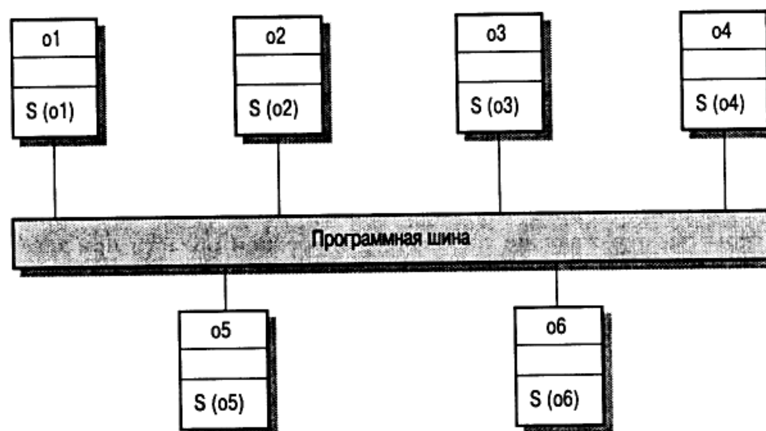

Рис. 8.9. Архитектура распределенных объектов

Объекты могут располагаться на разных компьютерах в сети и взаимодействовать посредством промежуточного ПО. По аналогии с системной шиной, которая позволяет подключать различные устройства и поддерживать взаимодействие между аппаратными средствами, промежуточное ПО можно рассматривать как шину программного обеспечения. Она предоставляет набор сервисов, позволяющий объектам взаимодействовать друг с другом, добавлять или удалять их из системы. Промежуточное ПО называют брокером запросов к объектам. Его задача –

обеспечивать интерфейс между объектами. Брокеры запросов к объектам рассматриваются в разделе 8.4.

Ниже перечислены основные преимущества модели архитектуры распределенных объектов.

- Разработчики системы могут не спешить с принятием решений относительно того, где и как будут предоставляться сервисы. Объекты, предоставляющие сервисы, могут выполняться в любом месте (узле) сети. Следовательно, различие между моделями толстого и тонкого клиентов становятся несущественными, так как нет необходимости заранее планировать размещение объектов для выполнения приложения.
- Системная архитектура достаточно открыта, что позволяет при необходимости добавлять в систему новые ресурсы. В следующем разделе отмечается, что стандарты программной шины постоянно совершенствуются, что позволяет объектам, написанным на разных языках программирования, взаимодействовать и предоставлять сервисы друг другу.
- Гибкость и масштабируемость системы. Для того чтобы справиться с системными нагрузками, можно создавать экземпляры системы с одинаковыми сервисами, которые будут предоставляться разными объектами или разными экземплярами (копиями) объектов. При увеличении нагрузки в систему можно добавить новые объекты, не прерывая при этом работу других ее объектов.
- Существует возможность динамически переконфигурировать систему посредством объектов, мигрирующих в сети по запросам. Объекты, предоставляющие сервисы, могут мигрировать на тот же процессор, что и объекты, запрашивающие сервисы, тем самым повышая производительность системы.

В процессе проектирования систем архитектуру распределенных объектов можно использовать двояко.

1. В виде логической модели, которая позволяет разработчикам структурировать и спланировать систему. В этом случае функциональность

приложения описывается только в терминах и комбинациях сервисов. Затем разрабатываются способы предоставления сервисов с помощью нескольких распределенных объектов. На этом уровне, как правило, проектируют крупномодульные объекты, которые предоставляют сервисы, отражающие специфику конкретной области приложения. Например, в программу учета розничной торговли можно включить объекты, которые бы вели учет состояния запасов, отслеживали взаимодействие с клиентами, классифицировали товары и др.

2. Как гибкий подход к реализации систем клиент/сервер. В этом случае логическая модель системы – это модель клиент/сервер, в которой клиенты и серверы реализованы как распределенные объекты, взаимодействующие посредством программной шины. При таком подходе легко заменить систему, например двухуровневую на многоуровневую. В этом случае ни сервер, ни клиент не могут быть реализованы в одном объекте, однако могут состоять из множества небольших объектов, каждый из которых предоставляет определенный сервис.

Примером системы, которой подходит архитектура распределенных объектов, может служить система обработки данных, хранящихся в разных базах данных (рис. 8.10). В этом примере любую базу данных можно представить как объект с интерфейсом, предоставляющим доступ к данным "только чтение". Каждый из объектов-интеграторов занимается определенными типами зависимостей между данными, собирая информацию из баз данных, чтобы попытаться проследить эти зависимости.

Объекты-визуализаторы взаимодействуют с объектами-интеграторами для представления данных в графическом виде либо для составления отчетов по анализируемым данным. Способы представление графической информации рассматриваются в главе 15.

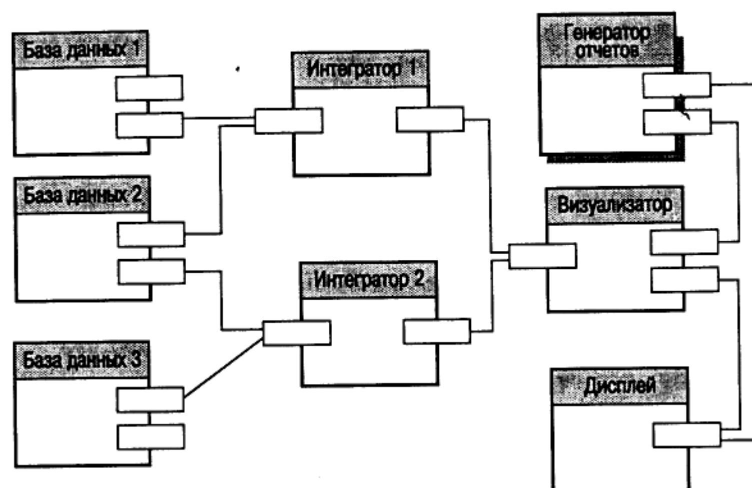

Рис. 8.10. Архитектура распределенной системы обработки данных

Для такого типа приложений архитектура распределенных объектов подходит больше, чем архитектура клиент/сервер, по трем причинам.

1. В этих системах (в отличие, например, от системы банкоматов) нет одного поставщика сервиса, на котором были бы сосредоточены все сервисы управления данными.

2. Можно увеличивать количество доступных баз данных, не прерывая работу системы, поскольку каждая база данных представляет собой просто объект. Эти объекты поддерживают упрощенный интерфейс, который управляет доступом к данным. Доступные базы данных можно разместить на разных машинах.

3. Посредством добавления новых объектов-интеграторов можно отслеживать новые типы зависимостей между данными.

Главным недостатком архитектур распределенных объектов является то, что их сложнее проектировать, чем системы клиент/сервер. Оказывается, что системы клиент/сервер предоставляют более естественный подход к созданию распределенных систем. В нем отражаются взаимоотношения между людьми, при которых одни люди пользуются услугами других людей, специализирующихся на предоставлении конкретных услуг. Намного труднее разработать систему в соответствии с архитектурой распределенных объектов, поскольку индустрия создания ПО пока еще не накопила

достаточного опыта в проектировании и разработке крупномодульных объектов.

8.4. CORBA

Как уже отмечалось в предыдущем разделе, при реализации архитектуры распределенных объектов необходимо промежуточное программное обеспечение (брокеры запросов к объектам), организующее взаимодействие между распределенными объектами. Здесь могут возникнуть определенные проблемы, поскольку объекты в системе могут быть реализованы на разных языках программирования, могут запускаться на разных платформах и их имена не должны быть известны всем другим объектам системы. Поэтому промежуточное ПО должно выполнять большую работу для того, чтобы поддерживалось постоянное взаимодействие объектов.

В настоящий момент для поддержки распределенных объектных вычислений существует два основных стандарта промежуточного ПО.

1. CORBA (Common Object Request Broker Architecture– архитектура брокеров запросов к общим объектам). Это набор стандартов для промежуточного ПО, разработанный группой OMG (Object Management Group – группа по управлению объектами). OMG является консорциумом фирм-производителей программного и аппаратного обеспечения, в числе которых такие компании, как Sun, Hewlett-Packard и IBM. Стандарты CORBA определяют общий машинонезависимый подход к распределенным объектным вычислениям. Разными производителями разработано множество реализаций этого стандарта. Стандарты CORBA поддерживаются операционной системой Unix и операционными системами от Microsoft.

2. DCOM (Distributed Component Object Model – объектная модель распределенных компонентов). DCOM представляет собой стандарт, разработанный и реализованный компанией Microsoft и интегрированный в

ее операционные системы. Данная модель распределенных вычислений менее универсальна, чем CORBA и предлагает более ограниченные возможности сетевых взаимодействий. В настоящий момент использование DCOM ограничивается операционными системами Microsoft.

Здесь я решил уделить внимание технологии CORBA, поскольку она более универсальна. Кроме того, я считаю, что, вероятно, CORBA, DCOM и другие технологии, например RMI (Remote Method Invocation – вызов удаленного метода, технология построения распределенных приложений на языке Java), будут постепенно сближаться друг с другом и это сближение будет базироваться на стандартах CORBA. Поэтому нет необходимости в еще одном стандарте. Различные стандарты будут только помехой в дальнейшем развитии.

Стандарты CORBA определены группой OMG, которая объединяет более 500 компаний, поддерживающих объектно-ориентированные разработки. Роль OMG – создание стандартов для объектно-ориентированных разработок, а не обеспечение конкретных реализаций этих стандартов. Эти стандарты находятся в свободном доступе на Web-узле OMG. Группа занимается не только стандартами CORBA, но также определяет широкий диапазон других стандартов, включая язык моделирования UML.

Представление распределенных приложений в рамках CORBA показано на рис. 8.11. Это упрощенная схема архитектуры управления объектами, взятая из статьи [317]. Предполагается, что распределенное приложение должно состоять из перечисленных ниже компонентов.

1. Объекты приложения, которые созданы и разработаны для данного программного продукта.

2. Стандартные объекты, которые определены группой OMG для специфических задач. Во время написания книги множество специалистов занимались разработкой стандартов объектов в области финансирования, страхования, электронной коммерции, здравоохранения и многих других.

3. Основные сервисы CORBA, поддерживающие базовые сервисы распределенных вычислений, например каталоги, управление защитой и др.

4. Горизонтальные средства CORBA, например пользовательские интерфейсы, средства управления системой и т.п. Под горизонтальными подразумеваются средства, общие для многих приложений.

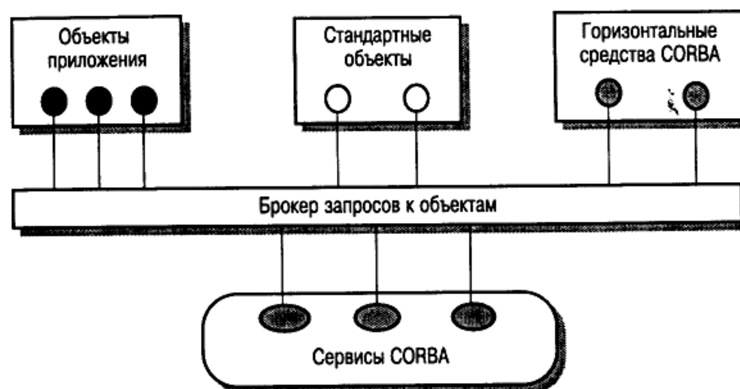

Рис. 8.11. Структура распределенного приложения, основанного на стандартах CORBA

Стандарты CORBA описывают четыре основных элемента.

1. Модель объектов, в которой объект CORBA инкапсулирует состояния посредством четкого описания на языке IDL (Interface Definition Language – язык описания интерфейсов).

2. Брокер запросов к объектам (Object Request Broker– ORB), который управляет запросами к сервисам объектов. ORB размещает объекты, предоставляющие сервисы, подготавливает их к получению запросов, передает запрос к сервису и возвращает результаты объекту, сделавшему запрос.

3. Совокупность сервисов объектов, которые являются основными сервисами, и необходимы во многих распределенных приложениях. Примерами могут быть службы каталогов, сервисы транзакций и сервисы поддержки временных объектов.

4. Совокупность общих компонентов, построенных на верхнем уровне основных сервисов. Они могут быть как вертикальными, отражающими специфику конкретной области, так и горизонтальными

универсальными компонентами, используемыми во многих программных приложениях. Эти компоненты рассматриваются в главе 14.

В модели CORBA объект инкапсулирует атрибуты и сервисы как обычный объект. Вместе с тем в объектах CORBA еще должно содержаться определение различных интерфейсов, описывающих глобальные атрибуты и операции объекта. Интерфейсы объектов CORBA определяются на стандартном универсальном языке описания интерфейсов IDL. Если один объект запрашивает сервисы, предоставляемые другими объектами, он получает доступ к этим сервисам через IDL-интерфейс. Объекты CORBA имеют уникальный идентификатор, называемый IOR (Interoperable Object Reference – ссылка на взаимодействующий объект). Когда один объект отправляет запросы к сервису, предоставляемому другим объектом, используется идентификатор IOR.

Брокеру запросов к объектам известны объекты, запрашивающие сервисы и их интерфейсы. Он организует взаимодействие между объектами. Взаимодействующим объектам не требуется что-либо знать о размещении других объектов, а также об их реализации. Так как интерфейс IDL отделяет объекты от брокера, реализацию объектов можно изменять, не затрагивая другие компоненты системы.

На рис. 8.12 показано, как объекты ol и o2 взаимодействуют посредством брокера запросов к объектам. Вызывающий объект (ol) связан с заглушкой (stub) IDL, которая определяет интерфейс объекта, предоставляющего сервис. Конструктор объекта ol при запросе к сервису внедряет вызовы в заглушку своей реализации объекта. Язык IDL является расширением C++, поэтому, если вы программируете на языках C++, С или Java, получить доступ к заглушке совсем просто. Перевод описания интерфейса объекта на IDL также возможен и для других языков, например Ada или COBOL. Но в этих случаях необходима соответствующая инструментальная поддержка.

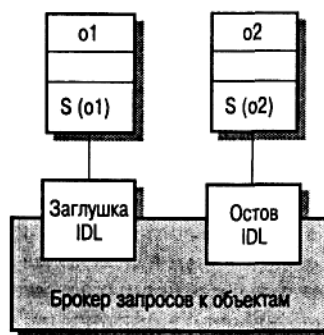

Рис. 8.12. Взаимодействие объектов посредством брокера запросов к объектам

Объект, предоставляющий сервис, связан с остовом (skeleton) IDL, который связывает интерфейс с реализацией сервисов. Иными словами, когда сервис вызывается через интерфейс, остов IDL транслирует вызов к сервису независимо от того, какой язык использовался в реализации. После завершения метода или процедуры остов транслирует результаты в язык IDL, так что они становятся доступными вызывающему объекту. Если объект одновременно предоставляет сервисы другим объектам или использует сервисы, которые предоставлены еще где-то, ему требуются и остов IDL, и заглушка IDL. Последняя необходима всем используемым объектам.

Брокер запросов к объектам обычно реализуется не в виде отдельных процессов, а как каркас (см. главу 14), который связан с реализацией объектов. Поэтому в распределенной системе каждый компьютер, на котором работают объекты, должен иметь собственный брокер запросов к объектам, который будет обрабатывать все локальные вызовы объектов. Но если запрос сделан к сервису, который предоставлен удаленным объектом, требуется взаимодействие между брокерами.

Такая ситуация проиллюстрирована на рис. 8.13. В данном примере, если объект ol или o2 отправляет запросы к сервисам, предоставляемым объектами o3 или o4, то необходимо взаимодействие связанных с этими объектами брокеров. Стандарты CORBA поддерживают взаимодействие "брокер-брокер", которое обеспечивает брокерам доступ к описаниям интерфейсов IDL, и предлагают разработанный группой OMG стандарт

обобщенного протокола взаимодействия брокеров GIOP (Generic Inter-ORB Protocol). Данный протокол определяет стандартные сообщения, которыми могут обмениваться брокеры при выполнении вызовов удаленного объекта и передаче информации. В сочетании с протоколом Internet низкого уровня TCP/IP этот протокол позволяет брокерам взаимодействовать через Internet.

Первые варианты CORBA были разработаны еще в 1980-х годах. Ранние версии CORBA просто были связаны с поддержкой распределенных объектов. Однако со временем стандарты развивались, становились более расширенными. Подобно механизмам взаимодействия распределенных объектов, стандарты CORBA сейчас определяют некоторые стандартные сервисы, которые можно использовать для поддержки объектно-ориентированных приложений.

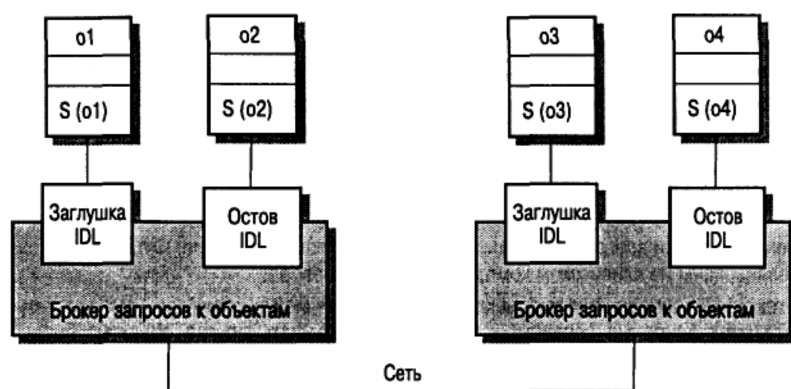

Рис. 8.13. Взаимодействие между брокерами запросов к объектам

Сервисы CORBA являются средствами, которые необходимы во многих распределенных системах. Эти стандарты определяют примерно 15 общих служб (сервисов). Вот некоторые из них.

1. Служба имен, которая позволяет объектам находить другие объекты в сети и ссылаться на них. Служба имен является сервисом каталогов, который присваивает имена объектам. При необходимости объекты через эту службу могут находить идентификаторы IOR других объектов.

2. Служба регистрации, которая позволяет объектам регистрировать другие объекты после совершения некоторых событий. С помощью этой службы объекты можно регистрировать по их участию в определенном

событии, а когда данное событие уже произошло, оно автоматически регистрируется сервисом.

3. Служба транзакций, которая поддерживает элементарные транзакции и откат назад в случае ошибок или сбоев. Эта служба является отказоустойчивым средством (см. главу 18), обеспечивающим восстановление в случае ошибок во время операции обновления. Если действия по обновлению объекта приведут к ошибкам или сбою системы, данный объект всегда можно вернуть назад к тому состоянию, которое было перед началом обновления.

Считается, что стандарты CORBA должны содержать определения интерфейсов для широкого диапазона компонентов, которые могут использоваться при построении распределенных приложений. Эти компоненты могут быть вертикальными или горизонтальными. Вертикальные компоненты разрабатываются специально для конкретных приложений. Как уже отмечалось, разработкой определений этих компонентов занято множество специалистов из различных сфер деятельности. Горизонтальные компоненты универсальны, например компоненты пользовательского интерфейса.

Во время написания этой книги спецификации компонентов были уже разработаны, но еще не согласованы. С моей точки зрения, вероятно, именно здесь наиболее слабое место стандартов CORBA, и, возможно, потребуется несколько лет, чтобы достичь того, что в наличии будут и спецификации, и реализации компонентов.

Контрольные вопросы

1. Какие характеристики распределенных систем существуют?
2. Архитектура клиент/сервер.
3. Архитектура распределенных объектов
4. CORBA (Common Object Request Broker Architecture).
5. DCOM (Distributed Component Object Model)

Ключевые слова: *совместное использование ресурсов, открытость, параллельность, масштабируемость, отказоустойчивость, прозрачность, сложность, безопасность, управляемость, непредсказуемость, идентификация ресурсов, коммуникации, качество системного сервиса, архитектура программного обеспечения, архитектура клиент/сервер, архитектура распределенных объектов, промежуточное программное обеспечение, модель тонкого клиента, модель толстого клиента, CORBA, DCOM.*

Keywords: *resource sharing, openness, concurrency, scalability, fault tolerance, transparency, complexity, security, manageability, unpredictability, identification of resources, communication, quality of service system, software architecture, client / server architecture, the architecture of distributed object middleware model thin client, thick client model, CORBA, DCOM.*

Kalit so'zlar: *manbalarni umumiy ishlatish, ochiqlik, paralellik, masshtablashtirish, rad etishga chidamiylik, shaffoflik, qiyinlik, havfsizlik, boshqarilish, oldindan ko'rib bilmaslik, manbalarni identifikatsiyalash, kommunikatsiyalar, tizim servisi sifati, dasturiy ta'minot arhitekturasi, klient/server arhitekturasi, ajratilgan obektlar arhitekturasi, o'rtadagi dasturiy ta'minot, nozik mijoz modeli, qalin mijoz modeli, CORBA, DCOM.*

Упражнения

1. Объясните, почему распределенные системы всегда более масштабируемы, чем централизованные. Какой вероятный предел масштабируемости программных систем?

2. В чем основное отличие между моделями толстого и тонкого клиента в разработке систем клиент/сервер? Объясните, почему использование Java как языка реализации сглаживает различия между этими моделями?

3. На основе модели приложения, изображенной на рис. 11.4, рассмотрите возможные проблемы, которые могут возникнуть при преобразовании системы 1980-х годов, реализованной на мейнфрейме и предназначенной для работы в сфере здравоохранения, в систему архитектуры клиент/сервер.

4. Распределенные системы, базирующиеся на модели клиент/сервер, разрабатывались с 1980-х годов, но только недавно такие системы, основанные на распределенных объектах, были реализованы. Приведите три причины, почему так получилось.

5. Объясните, почему использование распределенных объектов совместно с брокером запросов к объектам упрощает реализацию масштабируемых систем клиент/сервер. Проиллюстрируйте свой ответ примером.

6. Каким образом используется язык IDL для поддержки взаимодействия между объектами, реализованными на разных языках программирования? Объясните, почему такой подход может вызвать проблемы, связанные с производительностью, если между языками, которые используются при реализации объектов, имеются радикальные различия.

7. Какие базовые средства должен предоставлять брокер запросов к объектам?

8. Можно показать, что разработка стандартов CORBA для горизонтальных и вертикальных компонентов ограничивает конкуренцию. Если они уже созданы и адаптированы, это препятствует разработке лучших компонентов более мелкими компаниями. Обсудите роль стандартизации в поддержке или ограничении конкуренции на рынке программного обеспечения.

ГЛАВА 9. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

Объектно-ориентированное проектирование представляет собой стратегию, в рамках которой разработчики системы вместо операций и функций мыслят в понятиях *объекты.* Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций, определяемый состоянием объекта (рис. 9.1). Объекты скрывают информацию о представлении состояний и, следовательно, ограничивают к ним доступ. Под процессом объектно-ориентированного проектирования подразумевается проектирование классов объектов и взаимоотношений между этими классами. Когда проект реализован в виде исполняемой программы, все необходимые объекты создаются динамически с помощью определений

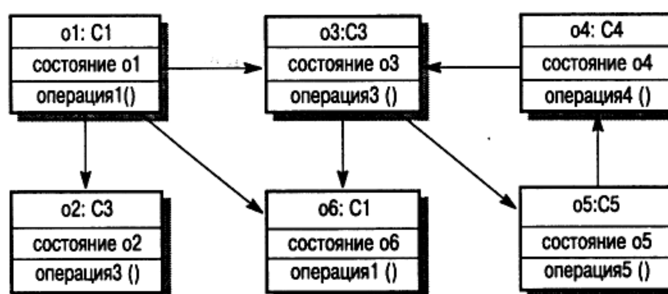

Рис. 9.1. Система взаимодействующих объектов

Объектно-ориентированное проектирование – только часть *объектно-ориентированного процесса разработки системы,* где на протяжении всего процесса создания ПО используется объектно-ориентированный подход. Этот подход подразумевает выполнение трех этапов.

- *Объектно-ориентированный анализ.* Создание объектно-ориентированной модели предметной области приложения ПО. Здесь объекты отражают реальные объекты-сущности, также определяются операции, выполняемые объектами.

- *Объектно-ориентированное проектирование.* Разработка объектно-ориентированной модели системы ПО (системной архитектуры) с учетом системных требований. В объектно-ориентированной модели определение всех объектов подчинено решению конкретной задачи.
- *Объектно-ориентированное программирование.* Реализация архитектуры (модели) системы с помощью объектно-ориентированного языка программирования. Такие языки, например Java, непосредственно выполняют реализацию определенных объектов и предоставляют средства для определения классов объектов.

Данные этапы могут "перетекать" друг в друга, т.е. могут не иметь четких рамок, причем на каждом этапе обычно применяется одна и та же система нотации. Переход на следующий этап приводит к усовершенствованию результатов предыдущего этапа путем более детального описания определенных ранее классов объектов и определения новых классов. Так как данные скрыты внутри объектов, детальные решения о представлении данных можно отложить до этапа реализации системы. В некоторых случаях можно также не спешить с принятием решений о расположении объектов и о том, будут ли эти объекты последовательными или параллельными. Все сказанное означает, что разработчики ПО не стеснены деталями реализации системы.

Объектно-ориентированные системы можно рассматривать как совокупность автономных и в определенной мере независимых объектов. Изменение реализации какого-нибудь объекта или добавление новых функций не влияет на другие объекты системы. Часто существует четкое соответствие между реальными объектами (например, аппаратными средствами) и управляющими ими объектами программной системы. Такой подход облегчает понимание и реализацию проекта.

Потенциально все объекты являются повторно используемыми компонентами, так как они независимо инкапсулируют данные о состоянии и операции. Архитектуру ПО можно разрабатывать на базе объектов, уже

созданных в предыдущих проектах. Такой подход снижает стоимость проектирования, программирования и тестирования ПО. Кроме того, появляется возможность использовать стандартные объекты, что уменьшает риск, связанный с разработкой программного обеспечения. Однако, как показано в главе 14, иногда повторное использование эффективнее всего реализовать с помощью коллекций объектов (компонентов или объектных структур), а не через отдельные объекты.

В книгах [74, 295, 186, 54, 137, 13*, 32*, 34*] предлагаются различные методы объектно-ориентированного проектирования. В этих методах на протяжении всего процесса проектирования используется единообразная нотация, принятая в UML [304]. В данной главе не предлагаются какие-либо особые методы проектирования, а рассматриваются лишь общие концепции объектно-ориентированного проектирования. В разделе 9.2 рассмотрены этапы процесса проектирования. По всей главе используется система обозначений, принятая в UML.

9.1. Объекты и классы объектов

В настоящее время широко используются понятия *объект* и *объектно-ориентированный.* Эти термины применяются к различным типам объектов, методам проектирования, системам и языкам программирования. Во всех случаях применяется общее правило, согласно которому объект инкапсулирует данные о своем внутреннем строении. Это правило отражено в моем определении объекта и класса объектов.

Объект–это нечто, способное пребывать в различных состояниях и имеющее определенное множество операций. Состояние определяется как набор атрибутов объекта. Операции, связанные с объектом, предоставляют сервисы (функциональные возможности) другим объектам (клиентам) для выполнения определенных вычислений. Объекты создаются в соответствии с определением класса объектов, которое служит шаблоном для создания

объектов. В него включены объявления всех атрибутов и операций, связанных с объектом данного класса.

Нотация, которая используется здесь для обозначения классов объектов, определена в UML. Класс объектов представляется как прямоугольник с названием класса, разделенный на две секции. В верхней секции перечислены атрибуты объектов. Операции, связанные с данным объектом, расположены в нижней секции. Пример такой нотации представлен на рис. 9.2, где показан класс объектов, моделирующий служащего некой организации. В UML термин *операция* является спецификацией некоторого действия, а термин *метод* обычно относится к реализации данной операции.

Класс **Работник** определяется рядом атрибутов, в которых содержатся данные о служащих, в том числе их имена и адрес, коды социального обеспечения, налоговые коды и т.д. Конечно, на самом деле атрибутов, ассоциированных с классом, больше, чем изображено на рисунке. Определены также операции, связанные с объектами: **принять** (выполняется при поступлении на работу), **уволить** (выполняется при увольнении служащего из организации), **пенсия** (выполняется, если служащий становится пенсионером организации) и **изменитьДанные** (выполняется в случаях, если требуется внести изменения в имеющиеся данные о работнике).

Работник
имя: строковое адрес: строковое датаРождения: дата табельныйНомер: целое номерСоцСтраха: строковое подразделение: Отдел менеджер: Работник оклад: целое статус: {постоянный, уволен, на пенсии} налогКод: целое ...
принять () уволить () пенсия () изменитьДанные ()

Рис. 9.2. Объект Работник

Взаимодействие между объектами осуществляется посредством запросов к сервисам (вызов методов) из других объектов и при необходимости путем обмена данными, требующимися для поддержки сервиса. Копии данных, необходимых для работы сервиса, и результаты работы сервиса передаются как параметры. Вот несколько примеров такого стиля взаимодействия.

```
// Вызов метода, ассоциированного с объектом Buffer (Буфер),
// который возвращает следующее значение в буфер
v = circularBuffer.Get () ;
// Вызов метода, связанного с объектом thermostat (термостат),
// который поддерживает нужную температуру
thermostat.setTemp (20);
```

В некоторых распределенных системах взаимодействие между объектами реализовано непосредственно в виде текстовых сообщений, которыми обмениваются объекты. Объект, получивший сообщение, выполняет его грамматический разбор, идентифицирует сервис и связанные с ним данные и запускает запрашиваемый сервис. Однако, если объекты сосуществуют в одной программе, вызовы методов реализованы аналогично

вызовам процедур или функций в языках программирования, например таких, как C или Ada.

Если запросы к сервису реализованы именно таким образом, взаимодействие между объектами синхронно. Это означает, что объект, отправивший запрос к сервису, ожидает окончания выполнения запроса. Однако, если объекты реализованы как параллельные процессы или потоки, взаимодействие объектов может быть асинхронным. Отправив запрос к сервису, объект может продолжить работу и не ждать, пока сервис выполнит его запрос. Ниже в этом разделе показано, каким образом можно реализовать объекты как параллельные процессы.

Как отмечалось в главе 7, в которой описан ряд объектных моделей, классы объектов можно упорядочить или в виде иерархии обобщения или в виде иерархии наследования, которые показывают отношения между основными и частными классами объектов. Эти частные классы объектов полностью совместимы с основными классами, но содержат больше информации. В системе обозначений UML направление обобщения указывается стрелками, направленными на родительский класс. В объектно-ориентированных языках программирования обобщение обычно реализуется через механизм наследования. Производный класс (класс-потомок) наследует атрибуты и операции от родительского класса.

Пример такой иерархии изображен на рис. 9.3, где показаны различные классы работников. Классы, расположенные внизу иерархии, имеют те же атрибуты и операции, что и родительские классы, но могут содержать новые атрибуты и операции или же изменять имеющиеся в родительских классах. Если в модели используется имя родительского класса, значит, объект в системе может быть определен либо самим классом, либо любым из его потомков.

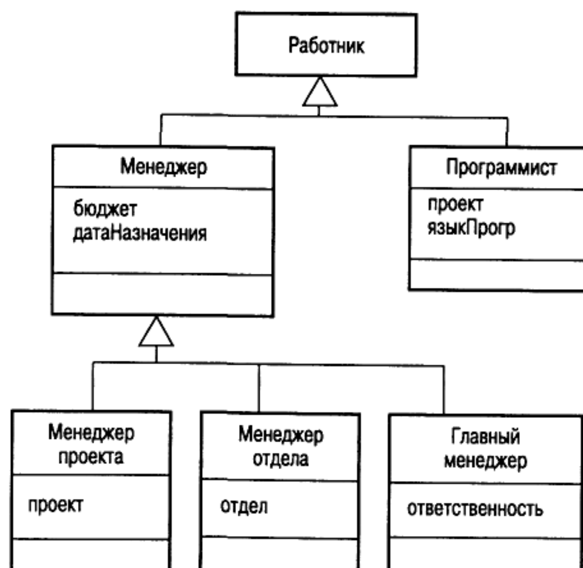

Рис. 9.3. Иерархия обобщения

На рис. 9.3 видно, что класс **Менеджер** обладает всеми атрибутами и операциями класса **Работник** и, кроме того, имеет два новых атрибута: ресурсы, которыми управляет менеджер (**бюджет**), и дата назначения его на должность менеджера (**датаНазначения**). Также добавлены новые атрибуты в класс **Программист**. Один из них определяет проект, над которым работает программист, другой характеризует уровень его профессионализма при использовании определенного языка программирования (**языкПрогр**). Таким образом, объекты класса **Менеджер** и **Программист** можно использовать вместо объектов класса **Работник**.

Объекты, являющиеся членами класса объектов, взаимодействуют с другими объектами. Эти взаимоотношения моделируются с помощью описания связей (ассоциаций) между классами объектов. В UML связь обозначается линией, которая соединяет классы объектов, причем линия может быть снабжена информацией о данной связи. На рис. 9.4 показаны связи между объектами классов **Работник** и **Отдел** и между объектами классов **Работник** и **Менеджер**.

Рис. 9.4. Модель связей

Связи представляют самые общие отношения и часто используются в UML там, где требуется указать, что какое-то свойство объекта является связанным с объектом или же реализация метода объекта полагается на связанный объект. Однако в принципе тип связи может быть каким угодно. Одним из наиболее распространенных типов связи, который служит для создания новых объектов из уже имеющихся, является агрегирование. Этот тип связи рассмотрен в главе 7.

9.2. Параллельные объекты

В общем случае объекты запрашивают сервис от любого объекта посредством передачи ему сообщения "запрос к сервису". Обычно нет необходимости в последовательном выполнении, при котором один объект ожидает завершения работы сервиса по сделанному запросу. Общая модель взаимодействия объектов позволяет их одновременное выполнение в виде параллельных процессов. Такие объекты могут выполняться на одном компьютере или на разных машинах как распределенные объекты.

На практике в большинстве объектно-ориентированных языков программирования по умолчанию реализована модель последовательного выполнения, в которой запросы к сервисам объектов и вызовы функций реализованы одним и тем же способом. Например, на языке Java, когда объект, вызвавший объект **theList** (Список), создается из обычного класса объектов, это запишется так:

theList.append(17)

Здесь вызывается метод **append** (добавить), связанный с объектом **theList**, который добавляет элемент 17 в список **theList**, а выполнение объекта, сделавшего вызов, приостанавливается до тех пор, пока не завершится операция добавления. Однако в Java существует очень простой механизм потоков (threads), который позволяет создавать параллельно выполняющиеся объекты. Поэтому объектно-ориентированную архитектуру программной системы можно преобразовать так, чтобы объекты стали параллельными процессами.

Существует два типа параллельных объектов.

1. *Серверы,* в которых объект реализован как параллельный процесс с методами, соответствующими определенным операциям объекта. Методы запускаются в ответ на внешнее сообщение и могут выполняться параллельно с методами, связанными с другими объектами. По окончании всех действий выполнение объекта приостанавливается и он ожидает дальнейших запросов к сервису.

2. *Активные объекты,* у которых состояние может изменяться посредством операций, выполняющихся внутри самого объекта. Процесс, представляющий объект, постоянно выполняет эти операции, а следовательно, никогда не останавливается.

Серверы наиболее полезны в распределенных средах, где вызывающий и вызываемый объекты выполняются на разных компьютерах. Время ответа, которое требуется сервису, заранее не известно, поэтому где только можно следует спроектировать систему так, чтобы объект, отправивший запрос к сервису, не ждал, пока сервис выполнит запрос. Также серверы могут использоваться на одной машине, где им требуется некоторое время для выполнения запроса (например, печать документа) и где есть вероятность отправки запросов к сервису от нескольких разных объектов.

Активные объекты используются там, где объектам необходимо обновлять свое состояние через определенные интервалы времени. Такие объекты характерны для систем реального времени, в которых объекты

связаны с аппаратными устройствами, собирающими информацию из окружения среды. Методы объектов позволяют другим объектам получить доступ к информации, определяющей состояние объекта.

В листинге 9.1 показано, как на языке Java можно определить и реализовать активный объект. Данный класс объектов представляет бортовой радиомаяк-ответчик (transponder) самолета. С помощью спутниковой навигационной системы радиомаяк-ответчик отслеживает положение самолета. Он может отвечать на сообщения, приходящие от компьютеров, управляющих воздушными полетами. В ответ на запрос метод **givePosition** сообщает текущее положение самолета.

Листинг 9.1. Реализация активного объекта, использующего потоки языка Java

```
class Transponder extends Thread {
Position currentPosition ;
Coords cl, c2 ;
Satellite satl, sat2 ;
Navigator theNavigator ;
public Position givePosition ()
{
return currentPosition;
}
public void run ()
{
while (true)
{
cl = satl.position ();
c2 = sat2.position ();
currentPosition = theNavigator.compute (cl, c2) ;
}
}
```

}//Transponder

Данный объект реализован как поток, где в непрерывном цикле метода **run** содержится код, вычисляющий положение самолета с помощью сигналов, полученных от спутников. В Java потоки создаются с помощью встроенного класса **Thread** (Поток), выступающего в объявлении классов в качестве базового.

9.3. Процесс объектно-ориентированного проектирования

В этом разделе процесс объектно-ориентированного проектирования показан на примере разработки структуры управляющей программной системы, встроенной в автоматизированную метеостанцию. Как отмечалось выше, есть несколько методов объектно-ориентированного проектирования, причем какого-либо предпочтительного метода или процесса проектирования не существует. Рассматриваемый здесь процесс является достаточно общим, т.е. состоит из операций, характерных для большинства процессов объектно-ориентированного проектирования. В этом отношении он сравним с процессом, предлагаемым языком UML [304], однако я значительно упростил его.

Общий процесс объектно-ориентированного проектирования состоит из нескольких этапов.

1. Определение рабочего окружения системы и разработка моделей ее использования.
2. Проектирование архитектуры системы.
3. Определение основных объектов системы.
4. Разработка моделей архитектуры системы.
5. Определение интерфейсов объекта.

Процесс проектирования нельзя представить в виде простой схемы, в которой предполагается четкая последовательность этапов. Фактически все перечисленные этапы в значительной мере можно выполнять параллельно, с

взаимным влиянием друг на друга. Как только разработана архитектура системы, определяются объекты и (частично или полностью) интерфейсы. После создания моделей объектов отдельные объекты можно переопределить, а это может привести к изменениям в архитектуре системы. Далее в этом разделе каждый этап процесса проектирования обсуждается отдельно.

Пример ПО, которым я воспользуюсь для иллюстрации объектно-ориентированного проектирования, представляет собой часть системы, создающей метеорологические карты на основе автоматически собранных метеорологических данных. Подробное перечисление требований для такой системы займет много страниц. Однако, даже ограничившись кратким описанием системы, можно разработать ее общую архитектуру.

Одним из требований системы построения карты погоды является регулярное обновление метеорологических карт на основе данных, полученных от удаленных метеостанций и других источников, например наблюдателей, метеозондов и спутников. В ответ на запрос регионального компьютера системы обслуживания метеостанций передают ему свои данные.

Региональная компьютерная система объединяет данные из различных источников. Собранные данные архивируются и с помощью данных из этого архива и базы данных цифровых карт создается набор локальных метеорологических карт. Карты можно распечатать, направив их на специальный принтер, или же отобразить в разных форматах.

Из данного описания видно, что одна часть общей системы занимается сбором данных, другая обобщает данные, полученные из различных источников, третья выполняет архивирование данных и наконец четвертая создает метеорологические карты. На рис. 9.5 изображена одна из возможных архитектур системы, которую можно построить на основе предложенного описания. Она представляет собой многоуровневую архитектуру (обсуждаемую в главе 10), в которой отражены все этапы

обработки данных в системе, т.е. сбор данных, обобщение данных, архивирование данных и создание карт. Такая многоуровневая архитектура вполне годится для нашей системы, так как каждый этап основывается только на обработке данных, выполненной на предыдущем этапе.

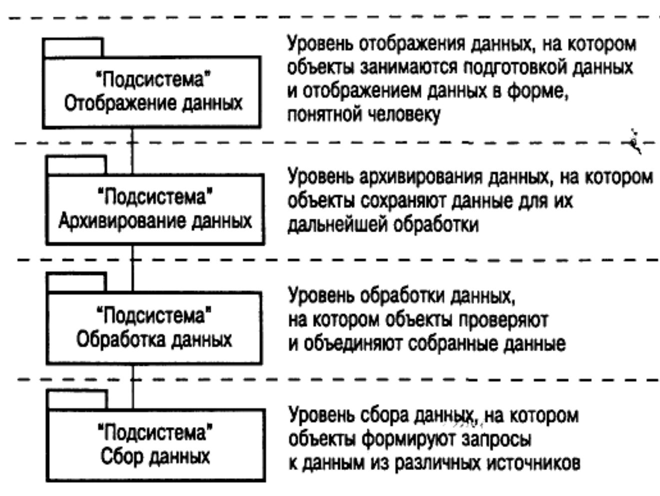

Рис. 9.5. Многоуровневая архитектура системы построения карт погоды

На рис. 9.5 показаны все уровни системы. Названия уровней заключены в прямоугольники, что в нотации UML обозначает подсистемы. Прямоугольники UML (т.е. подсистемы) – это набор объектов и других подсистем. Я использую здесь это обозначение, чтобы показать, что каждый уровень включает в себя множество других компонентов.

На рис. 9.6 изображена расширенная модель архитектуры, в которой показаны компоненты подсистем. Эти компоненты также очень абстрактны и построены на информации, содержащейся в описании системы. Продолжим рассматривать этот пример, уделяя особое внимание подсистеме **Метеостанция**, которая является частью уровня **Сбор данных**.

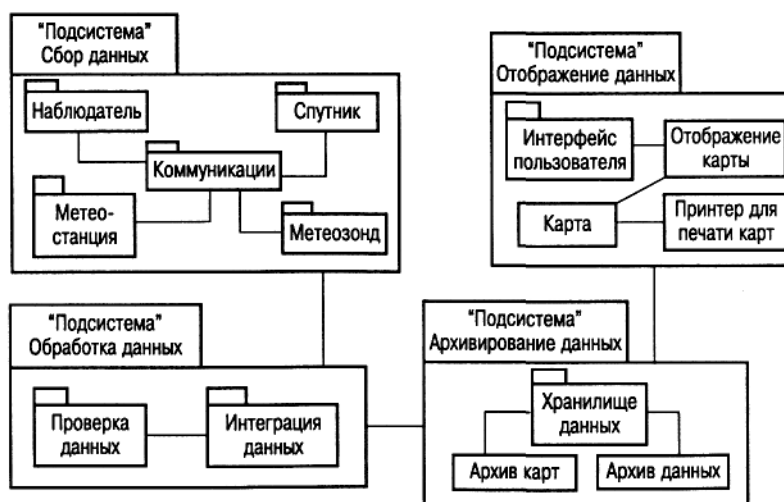

Рис. 9.6. Подсистемы в системе построения карт погоды

Окружение системы и модели ее использования

Первый этап в любом процессе проектирования состоит в выявлении взаимоотношений между проектируемым программным обеспечением и его окружением. Выявление этих взаимоотношений помогает решить, как обеспечить необходимую функциональность системы и как структурировать систему, чтобы она могла эффективно взаимодействовать со своим окружением.

Модель окружения системы и модель использования системы представляют собой две дополняющие друг друга модели взаимоотношений между данной системой и ее окружением.

1. Модель окружения системы – это статическая модель, которая описывает другие системы из окружения разрабатываемого ПО.

2. Модель использования системы – динамическая модель, которая показывает взаимодействие данной системы со своим окружением.

Модель окружения системы можно представить с помощью схемы связей (см. рис. 9.4), которая дает простую блок-схему общей архитектуры системы. С помощью *пакетов* языка UML ее можно представить в развернутом виде как совокупность подсистем (см. рис. 9.6). Такое представление показывает, что рабочее окружение системы Метеостанция находится внутри подсистемы, занимающейся сбором данных. Там же

показаны другие подсистемы, которые образуют систему построения карт погоды.

При моделировании взаимодействия системы с ее окружением применяется абстрактный подход, который не требует больших объемов данных для описания этих взаимодействий. Подход, предлагаемый UML, состоит в том, чтобы разработать модель вариантов использования, в которой каждый вариант представляет собой определенное взаимодействие с системой (см. главу 6). В модели вариантов использования каждое возможное взаимодействие изображается в виде эллипса, а внешняя сущность, включенная во взаимодействие, представлена стилизованной фигуркой человека. В нашем примере внешняя сущность, хотя и представлена фигуркой человека, является системой обработки метеорологических данных.

Модель вариантов использования для метеостанции показана на рис. 9.7. В этой модели метеостанция взаимодействует с внешними объектами во время запуска и завершения работы, при составлении отчетов на основе собранных данных, а также при тестировании и калибровке метеорологических приборов.

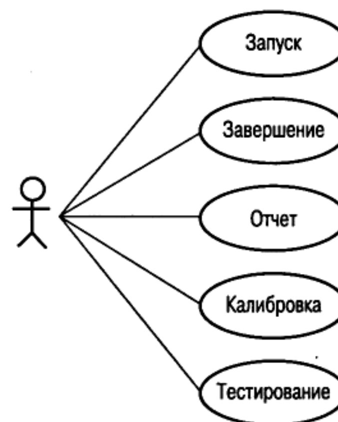

Рис. 9.7. Варианты использования метеостанции

Каждый из имеющихся вариантов использования можно описать с помощью простого естественного языка. Такое описание помогает разработчикам проекта идентифицировать объекты в системе и понять, что система должна делать. Я использую стилизованную форму описания,

которая четко определяет, как происходит обмен информацией, как инициируется взаимодействие и т.д. Эта форма описания показана в табл. 9.1, где представлен вариант использования **Отчет** (см. рис. 9.7).

Таблица 9.1. Описание варианта использования Отчет

Система	**Метеостанция**
Вариант использования	Отчет
Участники	Система сбора метеорологических данных, метеостанция
Данные	Метеостанция отправляет сводку с данными, снятыми с различных приборов в определенный временной период системой сбора метеорологических данных. В сообщении содержатся максимальные, минимальные и средние значения температуры почвы и воздуха, атмосферного давления, скорости ветра, общее количество выпавших осадков и направление ветра, взятые через пятиминутные интервалы времени
Входные сигналы	Система сбора метеорологических данных устанавливает модемную связь с метеостанцией и отправляет запрос на передачу данных
Ответ	Итоговые данные отправляются в систему сбора метеорологических данных
Комментарии	Обычно от метеостанций запрашивают отчет каждый час, но эта частота запросов может отличаться для разных станций, а также может измениться в будущее

Конечно, для описания вариантов использования можно прибегнуть к любой другой методике при условии, что предложенное описание краткое и понятное. Как правило, требуется разработать описания для всех вариантов использования, имеющихся в данной модели.

Описание вариантов использования помогает идентифицировать объекты и операции в системе. Из описания варианта использования Отчет видно, что в системе должны быть объекты, представляющие приборы для

сбора метеорологических данных, а также объекты, предоставляющие итоговые метеорологические данные. Должны также быть операции, формирующие запрос, и операции, пересылающие метеорологические данные.

9.4. Проектирование архитектуры

Когда взаимодействия между проектируемой системой ПО и ее окружением определены, эти данные можно использовать как основу для разработки архитектуры системы. Конечно, при этом необходимо применять знания об общих принципах проектирования системных архитектур и данные о конкретной предметной области.

Автоматизированная метеостанция является относительно простой системой, поэтому ее архитектуру можно вновь представить как многоуровневую модель. На рис. 9.8 внутри большого прямоугольника **Метеостанция** расположены три прямоугольника UML. Здесь я использовал систему нотации UML (текст в прямоугольниках с загнутыми углами) с тем, чтобы представить дополнительную информацию.

Рис. 9.8. Архитектура метеостанции

В программном обеспечении метеостанции можно выделить три уровня.

1. Уровень интерфейсов, который занимается всеми взаимодействиями с другими частями системы и предоставлением внешних интерфейсов системы.

2. Уровень сбора данных, управляющий сбором данных с приборов и обобщающий метеорологические данные перед отправкой их в систему построения карт погоды.

3. Уровень приборов, в котором представлены все приборы, используемые в процессе сбора исходных метеорологических данных.

В общем случае следует попытаться разложить систему на части так, чтобы архитектура была как можно проще. Согласно хорошему практическому правилу, модель архитектуры должна состоять не более чем из семи основных объектов. Каждый такой объект можно описать отдельно, однако для того, чтобы отобразить структуру этих объектов и их взаимосвязи, можно воспользоваться схемой, подобной показанной на рис. 9.6.

9.5. Определение объектов

Перед выполнением данного этапа проектирования уже должны быть сформированы представления относительно основных объектов проектируемой системы. В системе метеостанции очевидно, что приборы являются объектами и требуется по крайней мере один объект на каждом уровне архитектуры. Это проявление основного принципа, согласно которому объекты обычно появляются в процессе проектирования. Вместе с тем требуется определить и документировать все другие объекты системы.

Хотя этот раздел назван "Определение *объектов",* на самом деле на данном этапе проектирования определяются *классы* объектов. Структура системы описывается в терминах этих классов. Классы объектов, определенные ранее, неизбежно получают более детальное описание,

поэтому иногда приходится возвращаться на данный этап проектирования для переопределения классов.

Существует множество подходов к определению классов объектов.

1. Использование грамматического анализа естественного языкового описания системы. Объекты и атрибуты – это существительные, операции и сервисы – глаголы [1]. Такой подход реализован в иерархическом методе объектно-ориентированного проектирования [295], который широко используется в аэрокосмической промышленности Европы.

2. Использование в качестве объектов ПО событий, объектов и ситуаций реального мира из области приложения, например самолетов, ролевых ситуаций менеджера, взаимодействий, подобных интерактивному общению на научных конференциях и т.д. [313, 74, 343, 13*, 33*]. Для реализации таких объектов могут потребоваться специальные структуры хранения данных (абстрактные структуры данных).

3. Применение подхода, при котором разработчик сначала полностью определяет поведение системы. Затем определяются компоненты системы, отвечающие за различные поведенческие акты (режимы работы системы), при этом основное внимание уделяется тому, кто инициирует и кто осуществляет данные режимы. Компоненты системы, отвечающие за основные режимы работы, считаются объектами [301].

4. Применение подхода, основанного на сценариях, в котором по очереди определяются и анализируются различные сценарии использования системы. Поскольку анализируется каждый сценарий, группа, отвечающая за анализ, должна идентифицировать необходимые объекты, атрибуты и операции. Метод анализа, при котором аналитики и разработчики присваивают роли объектам, показывает эффективность подхода, основанного на сценариях [33].

Каждый из описанных подходов помогает начать процесс определения объектов. Но для описания объектов и классов объектов необходимо использовать информацию, полученную из разных источников. Объекты и

операции, первоначально определенные на основе неформального описания системы, вполне могут послужить отправной точкой при проектировании. Затем для усовершенствования и расширения описания первоначальных объектов можно использовать дополнительную информацию, полученную из области применения ПО или анализа сценариев. Дополнительную информацию также можно получить в ходе обсуждения с пользователями разрабатываемой системы или анализа имеющихся систем.

При определении объектов метеостанции я использую смешанный подход. Чтобы описать все объекты, потребуется много места, поэтому на рис. 9.9 я показал только пять классов объектов. **ПочвенныйТермометр**, **Анемометр** и **Барометр** являются объектами области приложения, а объекты

Метеостанция и **МетеоДанные** определены на основе описания системы и описания сценариев (вариантов использования).

Все объекты связаны с различными уровнями в архитектуре системы.

1. Класс объектов **Метеостанция** предоставляет основной интерфейс метеостанции при работе с внешним окружением. Поэтому операции класса соответствуют взаимодействиям, показанным на рис. 9.7. В данном случае, чтобы описать все эти взаимодействия, я использую один класс объектов, но в других проектах для представления интерфейса системы, возможно, потребуется использовать несколько классов.

2. Класс объектов **МетеоДанные** инкапсулирует итоговые данные от различных приборов метеостанции. Связанные с ним операции собирают и обобщают данные.

3. Классы объектов **ПочвенныйТермометр**, **Анемометр** и **Барометр** отображают реальные аппаратные средства метеостанции, соответствующие операции этих классов должны управлять данными приборами.

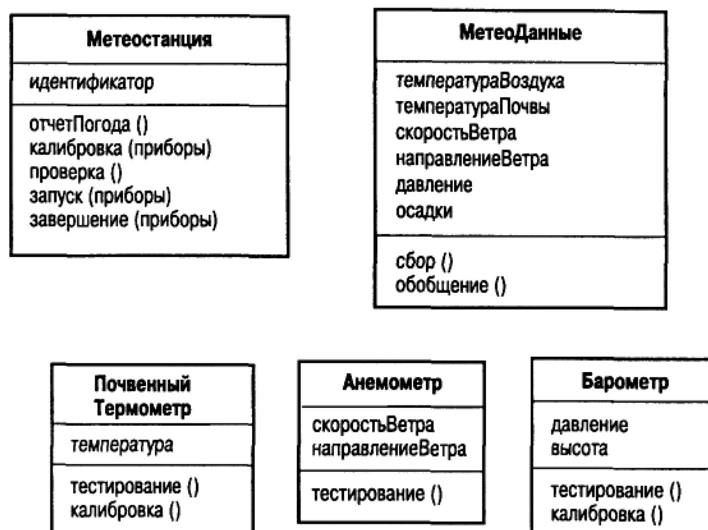

Рис. 9.9. Примеры классов объектов системы метеостанции

На этом этапе проектирования знания из области приложения ПО можно использовать для идентификации будущих объектов и сервисов. В нашем примере известно, что метеостанции обычно расположены в удаленных местах. Они оснащены различными приборами, которые иногда дают сбой в работе. Отчет о неполадках в приборах должен отправляться автоматически. А это значит, что необходимы атрибуты и операции, которые проверяли бы правильность функционирования приборов. Кроме того, необходимо идентифицировать данные, собранные со всех станций; таким образом, каждая метеостанция должна иметь собственный идентификатор.

Я решил не создавать объекты, ассоциированные с активными объектами каждого прибора. Чтобы снять данные в нужное время, объекты приборов вызывают операцию **сбор** объекта **МетеоДанные**. Активные объекты имеют собственное управление, в нашем примере предполагается, что каждый прибор сам определяет, когда нужно проводить замеры. Однако такой подход имеет недостаток: если принято решение изменить временной интервал сбора данных или если разные метеостанции собирают данные через разные промежутки времени, тогда необходимо вводить новые классы объектов. Создавая объекты приборов, снимающие показания по запросу, любые изменения в стратегии сбора данных можно легко реализовать без изменения объектов, связанных с приборами.

Вопросы для самопроверки и контроля

1. Какие этапы объектно-ориентированный подхода существуют?
2. Какие типы параллельных объектов существуют?
3. Что такое окружение системы и какие модели ее использования вы знаете?
4. Какие подходы к определению классов объектов используют?

Ключевые слова: *объекты, объектно-ориентированный анализ, объектно-ориентированное проектирование, объектно-ориентированное программирование, объект, серверы, активные объекты, модель окружения системы, модель использования системы.*

Keywords: *objects, object-oriented analysis, object-oriented designing, object-oriented programming, object, servers, active objects, model of the encirclement of the system, model of the use the system.*

Kalit so'zlar: *obyektlar, obyektga yo'naltirilgan tahlil, obyektga yo'naltirilgan loyihalashtirish, obyektga yo'naltirilgan dasturlash, serverlar, faol obyektlar, tizim muhiti modeli, tizimdan foydalanish modeli.*

Упражнения

1. Объясните, почему в проектировании систем применение подхода, который полагается на слабо связанные объекты, скрывающие информацию о своем представлении, приводит к созданию системной архитектуры, которую затем можно легко модифицировать.

2. Покажите на примерах разницу между объектом и классом объектов.

3. При каких условиях можно разрабатывать систему, в которой объекты выполняются параллельно?

4. С помощью графической системы нотации UML спроектируйте следующие классы объектов с определенными атрибутами и операциями:

• телефон;

• принтер персонального компьютера;

• персональная стереосистема;

• банковские расчеты;

• каталог библиотеки.

5. Разработайте более детальный проект метеостанции, добавив описания интерфейсов объектов, изображенных на рис. 12.9. Они могут быть записаны с помощью языков Java, C++ или UML.

6. Разработайте проект метеостанции, показывающий взаимодействие между подсистемой сбора данных и приборами, собирающими данные. Воспользуйтесь диаграммой последовательностей.

7. Определите возможные объекты в следующих системах, применяя при этом объектно-ориентированный подход.

   - Система "Дневник группы" поддерживает расписание собраний и встреч в группе сотрудников. Для организации встречи, в которой участвует группа людей, система находит общие для всех личных дневников свободные "окна" и назначает эту встречу на определенное время. Если система не находит общих "окон", то начинает взаимодействовать с пользователями, чтобы реорганизовать личные дневники и тем самым создать "окно" для встречи.

   - Установлена полностью автоматизированная бензоколонка. Водитель вставляет кредитную карточку в считывающее устройство, связанное с насосом; карточка по линиям коммуникаций проверяется кредитной компанией, устанавливается требуемое количество бензина. Затем автомобиль заправляется горючим. Когда подача прекращается, с кредитной карточки водителя снимается стоимость полученного бензина. Кредитная карточка возвращается после вычета водителю. Если карточка неверна, она возвращается водителю перед подачей топлива.

8. Запишите точные определения интерфейсов на языке Java или C++ для объектов, определенных в упражнении 12.7.

9. Нарисуйте диаграмму последовательностей, в которой отображены взаимодействия между объектами в системе "Дневник группы".

10. Нарисуйте диаграмму состояний, на которой отображены возможные изменения состояний для одного или более объектов, определенных в упражнении 12.7.

ГЛАВА 10. ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

В настоящее время компьютеры применяются для управления широким спектром разнообразных систем, начиная от простых домашних устройств и заканчивая крупными промышленными комплексами. Эти компьютеры непосредственно взаимодействуют с аппаратными устройствами. Программное обеспечение таких систем (управляющий компьютер плюс управляемые объекты) представляет собой встроенную систему реального времени, задача которой – реагировать на события, генерируемые оборудованием, т.е. в ответ на эти события вырабатывать управляющие сигналы. Такое ПО **встраивается** в большие аппаратные системы и должно обеспечивать реакцию на события, происходящие в окружении системы, в режиме **реального времени.**

Системы реального времени отличаются от других типов программных систем. Их корректное функционирование зависит от способности системы реагировать на события через заданный (как правило, короткий) интервал времени. Вот как я определяю систему реального времени.

Система реального времени – это программная система, правильное функционирование которой зависит от результатов ее работы и от периода времени, в течение которого получен результат. "Мягкая" система реального времени – это система, в которой операции **удаляются**, если в течение определенного интервала времени не выдан результат. "Жесткая" система реального времени – это система, операции которой становятся **некорректными**, т.е. вырабатывается сигнал об ошибке, если в течение определенного интервала времени результат не выдан.

Систему реального времени можно рассматривать как систему "стимул-отклик". При получении определенного входного стимула (входного сигнала) система генерирует связанный с ним отклик (ответное действие или

ответный сигнал). Следовательно, поведение системы реального времени можно определить с помощью списка входных сигналов, получаемых системой, связанных с ними ответных сигналов (откликов) и интервала времени, в течение которого система должна отреагировать на входной сигнал.

Входные сигналы делятся на два класса.

1. **Периодические сигналы** происходят через предопределенные интервалы времени. Например, система проверяет датчик каждые 50 миллисекунд, и предпринимает действия (реагирует) в зависимости от значений, полученных от датчика (стимула).

2. **Апериодические сигналы** происходят нерегулярно. Обычно они "сообщают о себе" посредством механизма прерываний. Примером апериодического сигнала может быть прерывание, которое вырабатывается по завершении передачи вход/выход и размещения данных в буфере обмена.

В системах реального времени периодические входные сигналы обычно генерируются сенсорами (датчиками), взаимодействующими с системой. Они предоставляют информацию о состоянии внешнего окружения системы. Системные отклики (ответные сигналы) направляются группе исполнительных механизмов, управляющих аппаратными устройствами, которые затем воздействуют на окружение системы. Апериодические входные сигналы могут генерироваться и сенсорами, и исполнительными механизмами. Как правило, апериодические сигналы означают исключительные ситуации, например ошибки в работе аппаратуры. На рис. 10.1 показана модель "сенсор-система-исполнительный механизм" для встроенной системы реального времени.

Рис. 10.1. Общая модель системы реального времени

Системы реального времени должны реагировать на входные сигналы, происходящие в разные моменты времени. Следовательно, архитектуру такой системы необходимо организовать так, чтобы управление переходило к соответствующему обработчику как можно быстрее после получения входного сигнала. В последовательных программах такой механизм передачи управления невозможен. Поэтому обычно системы реального времени проектируют как множество параллельных взаимодействующих процессов. Часть системы - управляющая программа, часто называемая диспетчером, - управляет всеми процессами.

Большинство моделей "стимул-отклик" систем реального времени сводятся к обобщенной архитектурной модели, состоящей из трех типов процессов (рис. 10.2). Для каждого типа сенсора имеется процесс управления сенсором; вычислительный процесс определяет необходимый ответный сигнал на полученный системой входной сигнал; процессы управления исполнительными механизмами управляют действиями этих механизмов. Такая модель позволяет быстро собрать данные со всех имеющихся сенсоров (до того, как произойдет следующий ввод данных), обработать их и получить ответный сигнал от соответствующего исполнительного механизма.

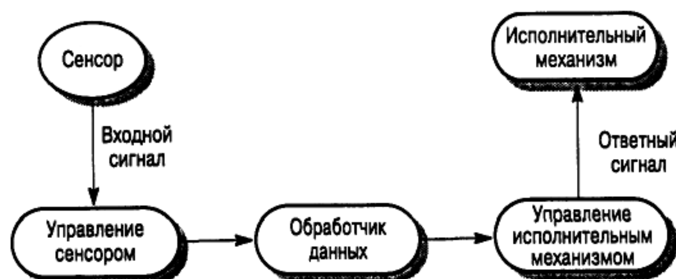

Рис. 10.2. Процессы управления сенсорами и исполнительными механизмами

10.1. Проектирование систем

Как указывалось в главе 2, в процессе проектирования системы принимаются решения о том, какие свойства будут реализованы программной частью системы, а какие - аппаратными средствами. Временные ограничения и другие требования предполагают, что некоторые функции системы, например обработку сигналов, необходимо реализовать на специально разработанном оборудовании. Таким образом, процесс проектирования систем реального времени включает в себя проектирование оборудования (аппаратуры) специального назначения и проектирование программного обеспечения.

Аппаратные компоненты обеспечивают более высокую производительность, чем эквивалентное им (по выполняемым функциям) программное обеспечение. Аппаратным средствам можно поручить "узкие" места системной обработки сигналов и, таким образом, избежать дорогостоящей оптимизации ПО. Если за производительность системы отвечают аппаратные компоненты, при проектировании ПО основное внимание можно уделить его переносимости, а вопросы, связанные с производительностью, отходят на второй план.

Решения о распределении функций по аппаратным и программным компонентам следует принимать как можно позже, поскольку архитектура системы должна состоять из автономных компонентов, которые можно реализовать как аппаратно, так и программно. Именно такая структура соответствует целям разработчика, проектирующего удобную в обслуживании систему. Следовательно, результатом процесса

проектирования высококачественной системы должна быть система, которую можно реализовать и аппаратными и программными средствами.

Отличие процесса проектирования систем реального времени от других систем состоит в том, что уже на первых этапах проектирования необходимо учитывать время реакции системы. В центре процесса проектирования системы реального времени – события (входные сигналы), а не объекты или функции. Процесс проектирования таких систем состоит из нескольких этапов.

1. Определение множества входных сигналов, которые будут обрабатываться системой, и соответствующих им системных реакций, т.е. ответных сигналов.

2. Для каждого входного сигнала и соответствующего ему ответного сигнала вычисляются временные ограничения. Они применяются к обработке как входных, так и ответных сигналов.

3. Объединение процессов обработки входных и ответных сигналов в виде совокупности параллельных процессов. В корректной модели системной архитектуры каждый процесс связан с определенным классом входных и ответных сигналов (как показано на рис. 10.2).

4. Разработка алгоритмов, выполняющих необходимые вычисления для всех входных и ответных сигналов. Чтобы получить представление об объемах вычислительных и временных затрат в процессе обработки сигналов, разработка алгоритмов обычно проводится на ранних этапах процесса проектирования.

5. Разработка временного графика работы системы.

6. Сборка системы, работающей под управлением диспетчера – управляющей программы.

Конечно, описанный процесс проектирования является итерационным. Как только определена структура вычислительных процессов и временной график работы, необходимо сделать всесторонний анализ и провести имитацию работы системы, чтобы удостовериться в том, что она

удовлетворяет временным ограничениям. В результате анализа может обнаружиться, что система не отвечает временным требованиям. В таком случае для повышения производительности системы необходимо изменить структуру вычислительных процессов, алгоритм управления, управляющую программу или все эти компоненты вместе.

В системах реального времени сложно анализировать временные зависимости. Из-за непредсказуемой природы апериодических входных сигналов разработчики вынуждены делать некоторые предварительные предположения относительно вероятности появлений апериодических сигналов. Сделанные предположения могут оказаться неверными, и после разработки системы ее показатели производительности не будут удовлетворять временным требованиям. В работах [86, 62] обсуждаются общие проблемы проверки временных параметров систем. В книге [132] всесторонне рассмотрены методы, используемые при анализе производительности систем реального времени.

Все процессы в системе реального времени должны быть скоординированы. Механизм координации процессов обеспечивает исключение конфликтов при использовании общих ресурсов. Когда один процесс использует общий ресурс (или объект), другие процессы не должны иметь доступ к этому ресурсу. К механизмам, обеспечивающим взаимное исключение процессов, относятся семафоры [97], мониторы [162] и метод критических областей [59]. Здесь я не буду рассматривать эти механизмы, так как все они хорошо документированы в описаниях операционных систем [332, 318].

10.2. Моделирование систем реального времени

Системы реального времени должны реагировать на события, происходящие через нерегулярные интервалы времени. Такие события (или входные сигналы) часто приводят к переходу системы из одного состояния в

другое. Поэтому одним из способов описания систем реального времени может быть модель конечного автомата и соответствующая диаграмма состояний, рассмотренные в главе 7.

В модели конечного автомата в каждый момент времени система находится в одном из своих состояний. Получив входной сигнал, она переходит в другое состояние. Например, система управления клапаном может перейти из состояния "Клапан открыт" в состояние "Клапан закрыт" после получения определенной команды оператора (входной сигнал).

Описанный выше подход к моделированию системы я проиллюстрирую на рассмотренном в главе 7 примере микроволновой печи. На рис. 10.3 показана модель конечного автомата для обычной микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы. Состояния системы обозначены скругленными прямоугольниками, входные сигналы, вызывающие переход системы из одного состояния в другое, показаны стрелками. На диаграмме показаны все состояния печи, также названы действия исполнительных механизмов системы или действия по выводу информации.

Просматривать последовательность работы системы нужно слева направо. В начальном состоянии **Ожидание**, пользователь может выбрать режим полной или половинной мощности. Следующее состояние наступает при нажатии на кнопку таймера и установке времени работы печи. Если дверь печи закрыта, система переходит в состояние **Действие**. В этом состоянии идет процесс приготовления пищи, после завершения которого печь возвращается в состояние **Ожидание**.

Модели конечного автомата – хороший способ представления структуры систем реального времени. Поэтому такие модели являются неотъемлемой частью методов проектирования систем реального времени [338]. Метод Харела (Harel) [115], базирующийся на **диаграммах состояний,** направлен на решение проблемы внутренней сложности моделей конечного автомата. Диаграмма состояний структурирует модели таким образом, что

группы состояния можно было бы рассматривать как единые сущности. Кроме того, с помощью диаграмм состояний параллельные системы можно представить в виде модели состояний. Модели состояний поддерживаются также UML [304, 30*]. В этой книге я также использую систему нотации, принятую в UML.

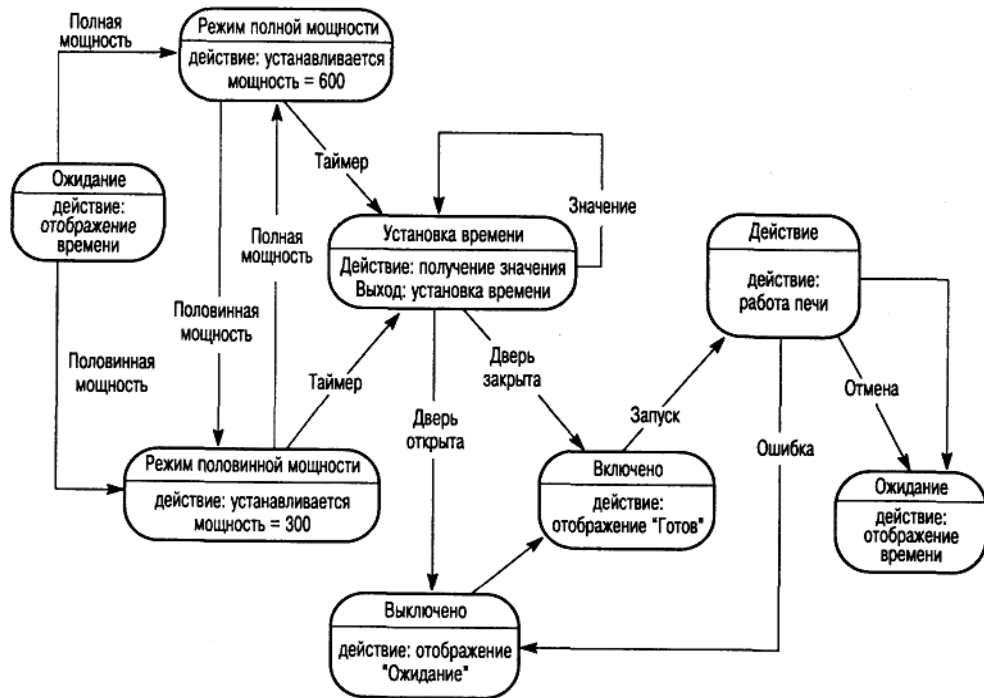

Рис. 10.3. Модель конечного автомата для микроволновой печи

10.3. Программирование систем реального времени

На архитектуру системы реального времени оказывает влияние язык программирования, который используется для реализации системы. До сих пор жесткие системы реального времени часто программируются на ассемблерных языках. Языки более высокого уровня также дают возможность сгенерировать эффективный программный код. Например, язык С позволяет писать весьма эффективные программы. Однако в нем нет конструкций, поддерживающих параллельность процессов и управление совместно используемыми ресурсами. Кроме того, программы на С часто сложны для понимания.

Язык Ada изначально разрабатывался для реализации встроенных систем, а потому располагает такими средствами, как управление процессами, исключения и правила представления. Его средство *рандеву* (rendezvous) - отличный механизм для синхронизации задач (процессов) [63, 24, 4*]. К сожалению, первая версия языка Ada (Ada 83) оказалась непригодной для реализации жестких систем реального времени. В ней отсутствовали средства, позволяющие установить предельные сроки завершения задач, не было встроенных исключений для случая превышения предельных сроков и предлагался строгий алгоритм обслуживания очереди "первым пришел - первым вышел". При пересмотре стандартов языка Ada [23] главное внимание уделялось именно этим моментам. В пересмотренной версии языка поддерживаются защищенные типы, что позволило более просто реализовывать защищенные разделяемые структуры данных и обеспечивать более полный контроль при выполнении и синхронизации задач. Однако при программировании систем реального времени улучшенная версия языка Ada все-таки не обеспечивает достаточного контроля над жесткими системами реального времени.

Первые версии языка Java разрабатывались для создания небольших встраиваемых систем, таких, например, как контроллеры устройств и приборов. Разработчики Java включили несколько средств для поддержки параллельных процессов в виде параллельных объектов (потоков) и синхронизированных методов. Но поскольку в подобных системах нет строгих временных ограничений, то и в языке Java не предусмотрены средства, позволяющие управлять планированием потоков или запускать потоки в конкретные моменты времени.

Поэтому Java не подходит для программирования жестких систем реального времени или систем, в которых имеется строгий временной график процессов. Перечислим основные проблемы Java как языка программирования систем реального времени.

1. Нельзя указать время, в течение которого должен выполняться поток.

2. Неконтролируем процесс очистки памяти – он может начаться в любое время. Поэтому невозможно предсказать поведение потоков во времени.

3. Нельзя определить размеры очереди, связанной с разделяемыми ресурсами.

4. Реализация виртуальной машины Java отличается для разных компьютеров.

5. В языке нет средств для детального анализа распределения времени работы процессоров.

В настоящее время ведется работа по решению некоторых из этих проблем и формируется новая версия языка Java для программирования систем реального времени [256]. Однако не совсем понятно, каким образом эту версию можно отделить от лежащей в ее основе виртуальной машины Java: свойство переносимости языка всегда конфликтовало с характеристиками режима реального времени.

10.4. Управляющие программы

Управляющая программа (диспетчер) системы реального времени является аналогом операционной системы компьютера. Она управляет процессами и распределением ресурсов в системах реального времени, запускает и останавливает соответствующие процессы для обработки входных сигналов и распределяет ресурсы памяти и процессора. Однако обычно в управляющих программах отсутствуют более сложные средства, присущие операционным системам, например средства управления файлами.

В работе [17] представлен полный обзор средств, необходимых управляющим программам систем реального времени. Данная тема обсуждается в монографии [80], где также кратко рассмотрены коммерческие

разработки управляющих программ для систем реального времени. Несмотря на то что на рынке программных продуктов существует несколько управляющих программ систем реального времени, их часто проектируют самостоятельно как части систем из-за специальных требований, предъявляемых к конкретным системам реального времени.

Компоненты управляющей программы (рис. 10.4) зависят от размеров и сложности проектируемой системы реального времени. Обычно управляющие программы, за исключением самых простых, состоят из следующих компонентов:

1. **Часы реального времени** периодически предоставляют информацию для планирования процессов.

2. **Обработчик прерываний управляет** апериодическими запросами к сервисам.

3. **Планировщик** просматривает список процессов, которые назначены на выполнение, и выбирает один из них.

4. **Администратор ресурсов**, получив процесс, запланированный на выполнение, выделяет необходимые ресурсы памяти и процессора.

5. **Диспетчер** запускает на выполнение какой-либо процесс.

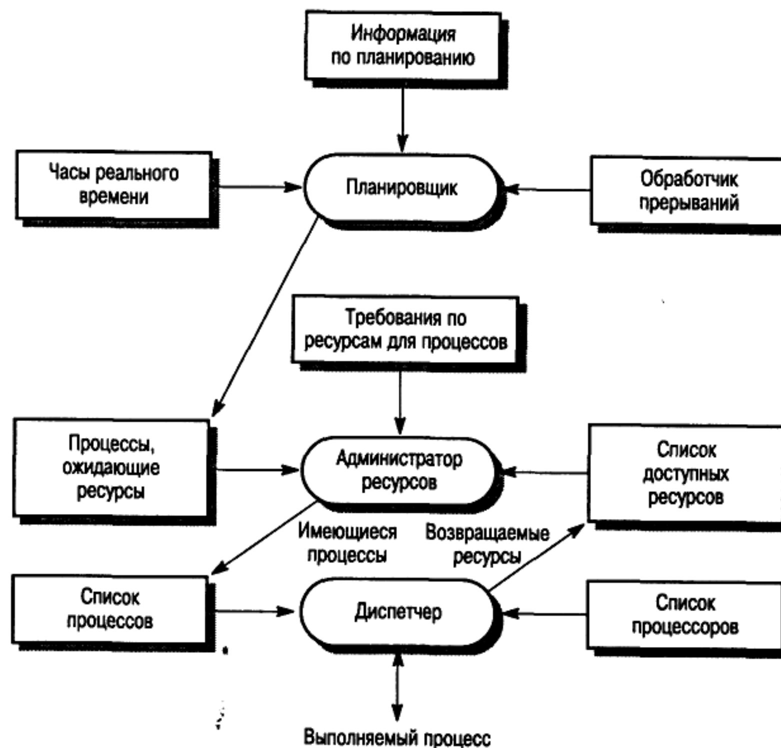

Рис. 10.4. Компоненты управляющей программы реального времени

Управляющие программы систем, предоставляющих сервисы на постоянной основе, например, телекоммуникационных или мониторинговых систем с высокими требованиями к надежности, могут иметь еще несколько компонентов:

- **Конфигуратор** отвечает за динамическое переконфигурирование аппаратных средств [205]. Не прекращая работу системы, из нее можно извлечь аппаратные модули и изменить систему посредством добавления новых аппаратных средств.
- **Менеджер неисправностей** отвечает за обнаружение аппаратных и программных неисправностей и предпринимает соответствующие действия по их исправлению. Вопросы отказоустойчивости и восстановления систем рассматриваются в главе 18.

Входные сигналы, обрабатываемые системой реального времени, обычно имеют несколько уровней приоритетов. Для одних сигналов, например связанных с исключительными ситуациями, важно, чтобы их обработка завершалась в течение определенного интервала времени. Если процесс с более высоким приоритетом запрашивает сервис, то выполнение других процессов должно быть приостановлено. Вследствие этого администратор системы должен уметь управлять по крайней мере двумя уровнями приоритетов системных процессов.

1. **Уровень прерываний** является наивысшим уровнем приоритетов. Он присваивается тем процессам, на которые необходимо быстро отреагировать. Примером такого процесса может быть процесс часов реального времени.
2. **Тактовый уровень** приоритетов присваивается периодическим процессам.

Еще один уровень приоритетов может быть у фоновых процессов, на выполнение которых не накладываются жесткие временные ограничения,

(например, процесс самотестирования). Эти процессы выполняются тогда, когда есть свободные ресурсы процессора.

Внутри каждого уровня приоритетов разным классам процессов можно назначить другие приоритеты. Например, может быть несколько уровней прерываний. Во избежание потери данных прерывание от более быстрого устройства должно вытеснять обработку прерываний от более медленного устройства.

10.5. Управление процессами

Управление процессами – это выбор процесса на выполнение, выделение для него ресурсов памяти и процессора и запуск процесса.

Периодическими называются процессы, которые должны выполняться через фиксированный предопределенный промежуток времени (например, при сборе данных или управлении исполнительными механизмами). Управляющая программа системы реального времени для определения момента запуска процесса использует свои часы реального времени. В большинстве систем реального времени есть несколько классов периодических процессов с разными периодами (интервалами времени между выполнением процессов) и длительностью выполнения. Управляющая программа должна быть способна в любой момент времени выбрать процесс, назначенный на выполнение.

Часы реального времени конфигурируются так, чтобы периодически подавать тактовый сигнал, период между сигналами составляет обычно несколько миллисекунд. Сигнал часов инициирует процесс на уровне прерываний, который запускает планировщик процессов для управления периодическими процессами. Процесс на уровне прерываний обычно сам не управляет периодическими процессами, поскольку обработка прерываний должна завершаться как можно быстрее.

Действия, выполняемые управляющей программой при управлении периодическими процессами, показаны на рис. 10.5. Планировщик просматривает список периодических процессов и выбирает из него на выполнение один процесс. Выбор зависит от приоритета процесса, периода процесса, предполагаемой длительности выполнения и конечных сроков завершения процесса. Иногда за один период между тактовыми сигналами часов необходимо выполнить два процесса с разными длительностями выполнения. В такой ситуации один процесс необходимо приостановить на время, соответствующее его длительности.

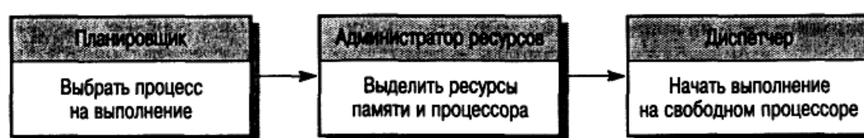

Рис. 10.5. Действия управляющей программы при запуске процесса

Если управляющей программой зарегистрировано прерывание, это означает, что к одному из сервисов сделан запрос. Механизм прерываний передает управление предопределенной ячейке памяти, в которой содержится команда переключения на программу обслуживания прерываний. Эта программа должна быть простой, короткой и быстро выполняться. Во время обслуживания прерываний все другие прерывания системой игнорируются. Чтобы уменьшить вероятность потери данных, время пребывания системы в таком состоянии должно быть минимальным.

Программа, выполняющая сервисную функцию, должна перекрыть доступ следующим прерываниям, чтобы не прервать саму себя. Она должна выявить причину прерывания и инициировать процесс с высоким приоритетом для обработки сигнала, вызвавшего прерывание. В некоторых системах высокоскоростного сбора данных обработчик прерываний сохраняет для последующей обработки данные, которые в момент получения прерывания находились в буфере. После обработки прерывания управление вновь переходит к управляющей программе.

В любой момент времени может быть несколько назначенных на выполнение процессов с разными уровнями приоритетов. Планировщик устанавливает порядок выполнения процессов. Эффективное планирование играет важную роль, если необходимо соответствовать требованиям, которые предъявляются к системе реального времени. Существует две основные стратегии планирования процессов.

1. **Невытесняющее планирования**. Один процесс планируется на выполнение, он запускается и выполняется до конца или блокируется по каким-либо причинам, например при ожидании ввода данных. При таком планировании могут возникнуть проблемы, связанные с тем, что в случае нескольких процессов с разными приоритетами процесс с высоким приоритетом должен ждать завершения процесса с низким приоритетом.

2. **Вытесняющее планирование**. Выполнение процесса может быть приостановлено, если к сервису поступили запросы от процессов с более высоким приоритетом. Процесс с более высоким приоритетом имеет преимущество перед процессом с более низким уровнем приоритета, и поэтому ему выделяется процессор.

В рамках этих стратегий разработано множество различных алгоритмов планирования. К ним относится циклическое планирование, при котором каждый процесс выполняется по очереди, и планирование по скорости, когда при первом выполнении получают более высокий приоритет процессы с коротким периодом выполнения [64]. Каждый из алгоритмов планирования имеет определенные преимущества и недостатки, однако здесь мы их рассматривать не будем.

Информация о назначенном на выполнение процессе передается администратору ресурсов. Он выделяет для выбранного процесса необходимую память, а в многопроцессорной системе – еще и процессор. Затем процесс помещается в "список назначений", т.е. в список процессов, назначенных на выполнение. Когда процессор завершает выполнение какого-либо процесса и становится свободным, вызывается диспетчер. Он

просматривает имеющийся список, выбирает процесс, который можно выполнять на свободном процессоре, и запускает его на выполнение.

10.6. Системы наблюдения и управления

В настоящее время можно выделить несколько классов стандартных систем реального времени: мониторинговые системы (системы наблюдения), системы сбора данных, системы управления и др. Каждому типу систем соответствует особая структура процессов, поэтому при проектировании системы, как правило, архитектуру создают по одному из существующих стандартных типов. Таким образом, вместо обсуждения общих проблем проектирования систем реального времени здесь лучше рассмотреть проектирование с помощью обобщенных моделей.

Системы наблюдения и управления – важный класс систем реального времени. Их основным назначением является проверка сенсоров (датчиков), предоставляющих информацию об окружении системы, и выполнение соответствующих действий в зависимости от поступившей от сенсоров информации. Системы наблюдения выполняют действия после регистрации особого значения сенсора. Системы управления непрерывно управляют аппаратными исполнительными механизмами на основании значений, получаемых от сенсоров.

Рассмотрим следующий пример.

Пусть в здании установлена система охранной сигнализации. В системе используется несколько типов сенсоров: датчики движения, установленные в отдельных комнатах; датчики на окнах первого этажа, которые подают сигнал, если разбивается окно; дверные датчики, фиксирующие открывание дверей. Всего в системе 50 датчиков на окнах, 30 на дверях и 200 датчиков движения.

Когда какой-либо датчик фиксирует присутствие постороннего, система автоматически вызывает местную полицию и, используя звуковой синтезатор, сообщает местоположение датчика (номер комнаты), от которого идет сигнал. В комнатах, расположенных возле активного датчика, включается световая сигнализация и звуковой аварийный сигнал. Система сигнализации обычно включается через сеть, но может работать и от батарей. Проблемы с электропитанием регистрируются специальной программой, контролирующей напряжение электросети. Если в сети регистрируется падение напряжения, программа переключает систему сигнализации на резервное питание от батарей.

В этом примере описана "мягкая" система реального времени, так как здесь нет жестких временных требований. В такой системе не *нужно* регистрировать события, происходящие с высокой скоростью, поэтому опрос датчиков может проводиться два раза в секунду.

Процесс проектирования начинается с описания апериодических входных сигналов, получаемых системой, и связанных с ними реакций системы. Здесь мы ограничимся упрощенным проектом, в котором не учитываются сигналы, порождаемые процедурами самопроверки, и внешние сигналы, генерируемые при тестировании системы или при ее выключении в случае ложной тревоги. В конечном счете система обрабатывает только два типа входных сигналов.

1. **Отключение электропитания** генерируется программой, контролирующей электрическую цепь. В ответ на этот сигнал система переключает сеть на резервное питание посредством подачи сигнала электронному прибору переключения питания.

2. **Сигнал о вторжении** является входным и генерируется одним из датчиков системы. В ответ на него система определяет номер комнаты, в которой находится активный датчик, вызывает полицию, инициируя звуковой синтезатор, и включает звуковой сигнал тревоги и световую сигнализацию здания в месте нарушения.

На следующем шаге процесса проектирования определяются временные ограничения для каждого входного и ответного сигналов системы. В табл. 10.1 перечислены эти временные ограничения. К разным типам датчиков, генерирующих входные сигналы, предъявлены разные временные требования.

Таблица 10.1. Временные ограничения на входные и ответные сигналы системы

Сигнал	**Временные ограничения**
Отключение электропитания	Переключение на питание от батарей должно произойти в течение 50 мс
Сигнализация на двери	Каждый сигнальный датчик на дверях проверяется дважды в секунду
Сигнализация на окнах	Каждый датчик на окне проверяется дважды в секунду
Датчик движения	Каждый датчик движения опрашивается дважды в секунду
Звуковой сигнал	Звуковой сигнал должен прозвучать через полсекунды после сигнала датчика
Включение световой сигнализации	Световая сигнализация должна включиться через полсекунды после сигнала датчика
Связь	Вызов в полицию должен начаться в течение 2 с после сигнала датчика
Синтезатор речи	Синтезированное сообщение должно быть готово через 4 с после сигнала датчика

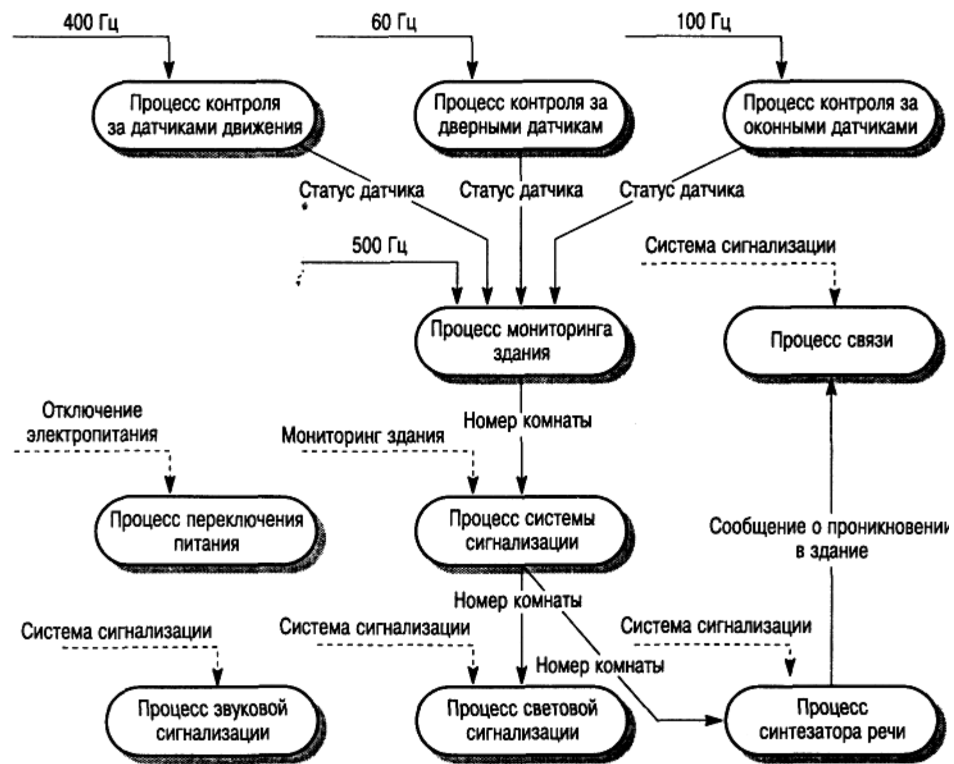

Рис. 10.6. Архитектура процессов системы охранной сигнализации

На следующем этапе проектирования распределяются системные функции по параллельным процессам. Периодически нужно опрашивать три типа датчиков, поэтому у каждого типа датчиков имеется связанный с ними процесс. Кроме этого, есть система управления прерываниями, контролирующая электропитание, система связи с полицией, синтезатор речи, система включения звуковой сигнализации и система, включающая световую сигнализацию возле датчика. Каждой системой управляет независимый процесс. На рис. 10.6 показана архитектура процессов системы.

На схеме, представленной на рис. 10.6, стрелки (с примечаниями), соединяющие отдельные процессы, обозначают потоки данных между процессами с указанием типа данных. Надписи над стрелками справа над каждым процессом указывают систему, управляющую данным процессом. На стрелках вверху указано минимальное значение частоты выполнения процесса.

Частота выполнения процессов определяется количеством датчиков и временными требованиями, предъявляемыми системе. Предположим, что в системе имеется 30 дверных датчиков, которые требуется проверять два раза

в секунду. Следовательно, связанный с дверным датчиком процесс должен выполняться 60 раз в секунду (частота 60 Гц). Также 400 раз в секунду выполняется процесс, контролирующий датчик движения.

Апериодические процессы обозначены стрелками с пунктирными линиями. На этих линиях указаны события, которые вызывают данный процесс. Все основные исполнительные процессы (звуковой и световой сигнализации и др.) начинаются командой из процесса **Система безопасности**; им не нужны данные из других процессов. Процессу, управляющему электропитанием, также не нужны данные из других частей системы.

Все представленные процессы можно реализовать на языке Java как потоки. Листинг 10.1 содержит код Java реализации процесса **BuildingMonitor** (мониторинг здания), опрашивающего датчики системы. В случае сигнала тревоги программа активизирует систему сигнализации. Пусть в нашем примере система соответствует временным требованиям. Как уже отмечалось, в языке Java 2.0 нет средств для задания частоты выполнения потоков.

Листинг 10.1. Реализация процесса мониторинга здания

```
//См. Web-страницу http://www.software-engin.com/,
//где представлен полный Java-код этого примера
class BuildingMonitor extends Thread {
BuildingSensor win, door, move;
Siren siren = new SirenO;
Lights lights = new Lights{);
Synthesizer synthesizer = new Synthesizer();
DoorSensors doors = new DoorSensors (30);
WindowSensors windows = new WindowSensors(50);
MovementSensors movements = new MovementSensors(200);
PowerMonitor pm = new PowerMonitor();
```

```
BuildingMonitor()
{
//инициализация датчиков и запуск процессов
siren.start();lights.start();
synthesizer.start();windows.start();
doors.start()/movements.start();pm.start();
}
public void run ()
{
int room = 0;
while (true)
{
//проверка датчиков движения два раза в секунду (400 Гц) move =
movements.getVal();
//проверка оконных датчиков два раза в секунду (100 Гц) win =
windows.getVal();
//проверка дверных датчиков два раза в секунду (60 Гц) door =
doors.getVal();
if(move.sensorVal==l|door.sensorVal==l|win.sensorVal==l)
{
//датчик зарегистрировал нарушение
if(move.sensorVal == 1) room = move.room;
if(door.sensorVal == 1) room = door.room;
if(win.sensorVal == 1) room = win.room;

lights.on(room);siren.on();synthesizer.on(room);
break;
}
}
```

```
lights.shutdown() ; siren.shutdown();synthesizer.shutdown();
windows.shutdown();doors.shutdown();movements.shutdown();
}//run
)//BuildingMonitor
```

Поскольку данная система не содержит строгих временных требований, ее можно реализовать на языке Java. Конечно, в Java 2.0 нет никакой гарантии соответствия временным спецификациям. В нашей системе все датчики опрашиваются одинаковое количество раз, чего не бывает в реальных системах.

После определения архитектуры процессов системы начинается разработка алгоритмов обработки входных сигналов и генерации ответных сигналов. Как уже отмечалось в разделе 3.1, этот этап проектирования необходимо выполнять как можно раньше, чтобы удостовериться, что система будет соответствовать временным требованиям. Если соответствующие алгоритмы оказываются сложными, может возникнуть необходимость в изменении временных ограничений. Обычно алгоритмы систем реального времени сравнительно просты. Они проверяют ячейки памяти, выполняют некоторые простые расчеты и управляют передачей сигнала. В примере системы сигнализации проектирование алгоритмов не рассматривается.

Последним этапом процесса проектирования является составление временного графика выполнения процессов. В нашем примере нет процессов со строгими сроками выполнения. Рассмотрим приоритеты процессов. Все процессы, опрашивающие датчики, имеют один и тот же приоритет. Процессу, управляющему электропитанием, необходимо назначить более высокий приоритет. Приоритеты процессов, управляющих системой сигнализации, и процессов, опрашивающих датчики, должны быть одинаковы.

Систему охранной сигнализации можно отнести скорее к системам наблюдения, чем к системам управления, так как в ней нет исполнительных

механизмов, напрямую зависящих от значений датчиков. Примером системы управления может служить система управления отоплением здания. Система наблюдает за температурными датчиками, установленными в разных комнатах здания и переключает нагревательные приборы в зависимости от реальной температуры и температуры, установленной в термореле. Термореле, в свою очередь, контролирует отопительный котел.

Архитектура процессов такой системы показана на рис. 10.7; в общем виде она выглядит подобно системе сигнализации. Дальнейшее рассмотрение этого примера предлагается читателю в качестве упражнения.

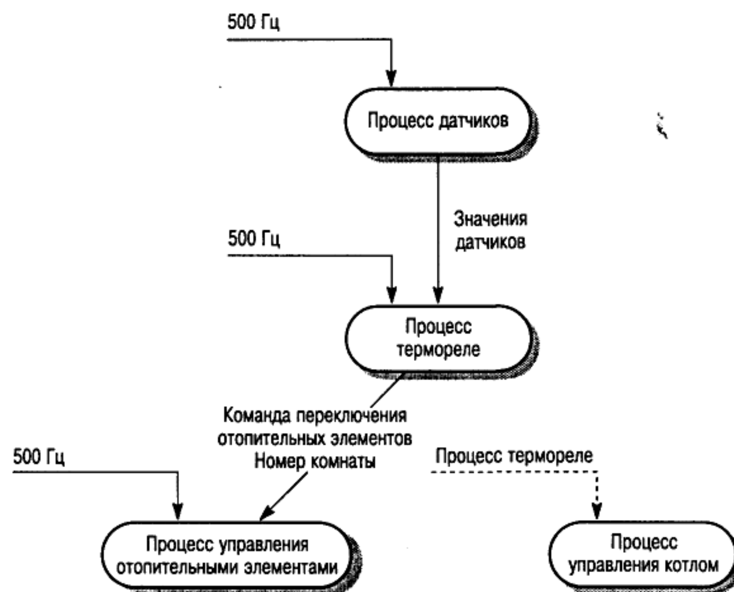

Рис. 10.7. Архитектура процессов системы управления отоплением

10.7.Системы сбора данных

Эти системы представляют другой класс систем реального времени, которые обычно базируются на обобщенной архитектурной модели. Такие системы собирают данные с сенсоров в целях их последующей обработки и анализа.

Для иллюстрации этого класса систем рассмотрим модель, представленную на рис. 10.8. Здесь изображена система, собирающая данные с датчиков, которые измеряют поток нейтронов в ядерном реакторе. Данные,

собранные с разных датчиков, помещаются в буфер, из которого затем извлекаются и обрабатываются. На мониторе оператора отображается среднее значение интенсивности потока нейтронов.

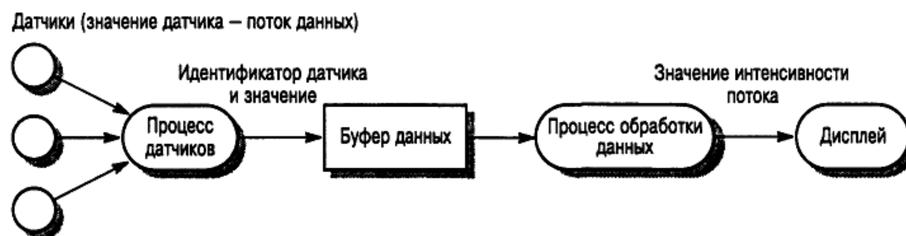

Рис. 10.8. Архитектура системы наблюдения за интенсивностью потока нейтронов

Каждый датчик связан с процессом, который преобразует аналоговый сигнал, показывающий интенсивность входного потока, в цифровой. Сигнал совместно с идентификатором датчика записывается в буфер, где хранятся данные. Процесс, отвечающий за обработку данных, берет их из буфера, обрабатывает и передает процессу отображения для вывода на операторную консоль.

В системах реального времени, ведущих сбор и обработку данных, скорости выполнения и периоды процесса сбора и процесса обработки могут не совпадать. Если обрабатываются большие объемы данных, сбор данных может выполнятся быстрее, чем их обработка. Если же выполняются только простые вычисления, быстрее происходит обработка данных, а не их сбор.

Чтобы сгладить разницу в скоростях сбора и обработки данных, в большинстве подобных систем для хранения входных данных используется кольцевой буфер. Процессы, создающие данные (процессы-производители), поставляют информацию в буфер. Процессы, обрабатывающие данные (процессы-потребители), берут данные из буфера (рис. 10.9).

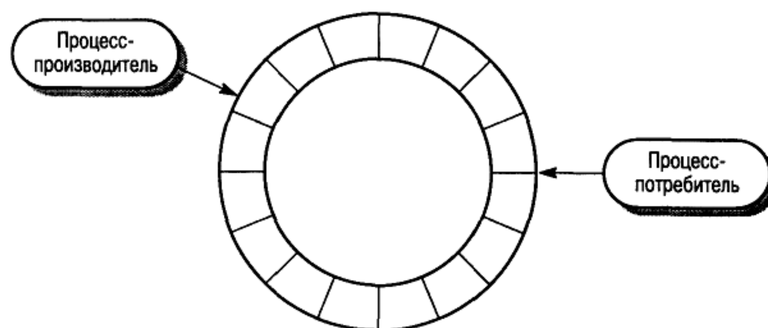

Рис. 10.9. Кольцевой буфер в системе сбора данных

Очевидно, необходимо предотвратить одновременный доступ процесса-производителя и процесса-потребителя к одним и тем же элементам буфера. Кроме того, система должна отслеживать, чтобы процесс-производитель не добавлял данные в полный буфер, а процесс-потребитель не забирал данные из пустого буфера.

В листинге 10.2 показана возможная реализация буфера данных как объекта Java. Значения в буфере имеют тип SensorRecord (запись данных датчика). Определены два метода– get и put: метод get берет элементы из буфера, метод put добавляет элемент в буфер. При объявлении типа CircularBuffer (кольцевой буфер) программный конструктор задает размер буфера.

Листинг 10.2. Реализация кольцевого буфера

```
class CircularBuffer
{
int bufsize;
SensorRecord [] std're;
int numberOfEntries = 0;
int front = 0, back = 0;

CircularBuffer (int n) {
bufsize = n;
store = new SensorRecord [bufsize] ;
```

```
}//CircularBuffer

synchronized void put (SensorRecord rec) throws
InterruptedException
{
if(numberOfEntries == bufsize)
waitt) ;
store [back] = new SensorRecord (rec.sensorld,rec.sensorVal);
back = back + 1;
if(back == bufsize)
back = 0;
numberOfEntries = numberOfEntries + 1;
notify();
}//put
synchronized SensorRecord get() throws InterruptedException
{
SensorRecord result = new SensorRecord(-1,-1);
if(numberOfEntries == 0)
wait () ;
result = store [front];
front = front + 1;
if(front == bufsize)
front = 0;
numberOfEntries = numberOfEntries - 1;
notify();
return result;
}// get
}//CircularBuffer
```

Модификатор **synchronized**, связанный с методами **get** и **put**, указывает на то, что данные методы не должны выполняться параллельно. При вызове

одного из этих методов система реального времени блокирует экземпляр объекта, чтобы в это же время не произошел вызов другого метода и соответственно не производились манипуляции на том же участке буфера. Вызовы методов **wait** и **notify** из методов **get** и **put** гарантируют, что входные данные нельзя положить в полный буфер или взять из пустого буфера. Метод **wait** вызывает поток и приостанавливается, пока другой поток с помощью метода **notify** не отправит ему сообщение о снятии ожидания. При вызове метода **wait** блокировка на защищенные данные объекта снимается. Метод **notify** возобновляет выполнение одного из ожидающих потоков.

Контрольные вопросы:

1) Что такое система реального времени?
2) Этапы процесса проектирования системы реального времени.
3) Основные проблемы Java как языка программирования систем реального времени.
4) Из каких компонентов состоят управляющие программы?
5) Укажите основные стратегии планирования процессов.

Ключевые слова: *система реального времени, периодические сигналы, апериодические сигналы, модели конечного автомата, часы реального времени, обработчик прерываний управляет, планировщик, администратор ресурсов, диспетчер, конфигуратор, менеджер неисправностей, уровень прерываний, тактовый уровень, управление процессами, невытесняющее планирования, вытесняющее планирование, система безопасности.*

Key words: *real-time, periodic signals, aperiodic signals, a finite state machine model, real-time clock interrupt handler manages scheduling, resource manager, manager, configurator, manager fault interrupt level, the clock level, process management, non-preemptive scheduling, preemptive planning, security system.*

Kalit so'zlar: *real vaqt tizimi, davriy signallar, nodavriy signallar, yakuniy avtomat modellari, real vaqt soatlari, boshqaruv uzulishlariga ishlov berish, rejalashtiruvchi, manbalar administratori, dispetcher, konfigurator, shikastlanganlik menedjeri, uzulishlar darajasi, takt darajasi, jarayonlarni boshqarish, siqib chiqarmaslik rejalashtiruvi, siqib chiqarish rejalashtiruvi, havfsizlik tizimi.*

Упражнения

1. Почему системы реального времени обычно реализованы как множество параллельных процессов? Проиллюстрируйте свой ответ примерами.

2. Объясните, почему объектно-ориентированные методы разработки ПО не всегда подходят к системам реального времени.

3. Нарисуйте диаграммы состояний управляющего ПО для следующих систем.

- Автоматическая стиральная машина с разными программами для разных типов белья.
- Программное обеспечение для проигрывателя компакт-дисков.
- Телефонный автоответчик, который записывает входящие сообщения и отображает количество полученных сообщений на жидкокристаллическом экране. Система должна определить телефон звонившего, вывести на экран последовательность чисел (идентифицированных как тоновый набор) и хранить записанные сообщения, которые затем можно прослушать.
- Автомат по выдаче напитков, который может налить кофе с молоком и сахаром или без них. Пользователь бросает монету и с помощью нажатия кнопок на автомате выбирает нужный режим. Автомат выдает чашку с растворимым кофе. Пользователь затем ставит чашку под кран, нажимает другую кнопку и автомат наливает в чашку горячую воду.

4. Используя методы проектирования систем реального времени спроектируйте заново систему сбора данных от метеостанций, рассмотренную в главе 12, в виде системы "стимул-ответ".

5. Спроектируйте архитектуру процессов для системы наблюдения, собирающей данные с группы датчиков, измеряющих состав воздуха и расположенных вокруг города. В системе 5000 датчиков, организованных в группы по 100 штук. Каждый датчик должен проверяться 4 раза в секунду. Если более 30% датчиков в группе зафиксируют, что качество воздуха ниже допустимого уровня, активизируется предупреждающий световой сигнал. Все датчики передают собранные данные центральному компьютеру, который каждые 15 мин генерирует отчет о составе воздуха в городе.

6. Обсудите сильные и слабые стороны Java как языка программирования для реализации систем реального времени.

7. Система безопасности поезда автоматически закрывает двери, если скорость поезда превышает предельную для данного участка трассы или если при выходе на участок пути горит красный свет (т.е. въезд на участок запрещен). Остальные подробности перечислены во врезке 13.1. Идентифицируйте входные сигналы, которые должна

обрабатывать бортовая система управления поездом, и связанные с ними ответные сигналы.

8. Предположите вероятную архитектуру процессов такой системы.

9. Если в бортовой системе безопасности поезда при сборе данных с путевых передатчиков используются периодические процессы, какую частоту сбора данных следует запланировать, чтобы система гарантированно получала информацию от передатчиков? Обоснуйте свой ответ.

ГЛАВА 11. ПРОЕКТИРОВАНИЕ С ПОВТОРНЫМ ИСПОЛЬЗОВАНИЕМ КОМПОНЕНТОВ

В большинстве инженерных разработок процесс проектирования основан на повторном использовании уже имеющихся компонентов. В таких сферах, как механика или электротехника, инженеры никогда не разрабатывают проект "с нуля". Их проекты базируются на компонентах, уже проверенных и протестированных в других системах. Как правило, это не только малые компоненты, например фланцы и клапаны, но также целые подсистемы, например двигатели, компрессоры или турбины.

В настоящее время не вызывает сомнений тот факт, что необходимо сравнивать различные подходы к разработке программного обеспечения. Если программное обеспечение рассматривать как актив, то повторное использование этих активов позволит существенно сократить расходы на его разработку. Только с помощью систематического повторного использования ПО можно уменьшить расходы на его создание и обслуживание, сократить сроки разработки систем и повысить качество программных продуктов.

Чтобы повторное использование ПО было эффективным, его необходимо учитывать на всех этапах процесса проектирования ПО или процесса разработки требований. Во время программирования возможно повторное использование на этапе подбора компонентов, соответствующих требованиям. Однако для **систематического** повторного использования необходим такой процесс проектирования, в ходе которого постоянно рассматривалась бы возможность повторного использования уже существующих архитектур, где система была бы явно организована из доступных имеющихся компонентов ПО.

Метод проектирования ПО, основанный на повторном использовании, предполагает максимальное использование уже имеющихся программных объектов. Такие объекты могут радикально различаться размерами.

1. **Повторно используемые приложения**. Можно повторно использовать целые приложения либо путем включения их в систему без изменения других подсистем (например, коммерческие готовые продукты, см. раздел 11.1.2), либо с помощью разработки семейств приложений, работающих на разных платформах и адаптированных к требованиям конкретных заказчиков (см. раздел 11.2).

2. **Повторно используемые компоненты**. Можно повторно использовать компоненты приложений - от подсистем до отдельных объектов. Например, система распознавания текста, разработанная как часть системы обработки текстов, может повторно использоваться в системах управления базами данных. Этот вид повторного использования рассматривается в разделе 11.3.

3. **Повторно используемые функции.** Можно повторно использовать программные компоненты, которые реализуют отдельные функции, например математические. Основанный на стандартных библиотеках метод повторного использования применяется в программировании последние 40 лет.

Повторное использование целых приложений практикуется довольно широко; при этом компании, занимающиеся разработкой ПО, адаптируют свои системы для разных платформ и для работы в различных условиях. Также хорошо известно повторное использование функций через стандартные библиотеки, например графические и математические. Интерес к повторному использованию компонентов возник еще в начале 1980-х годов, однако на практике такой подход к разработке систем ПО применяется лишь последние несколько лет.

Очевидным преимуществом повторного использования ПО является снижение общей стоимости проекта, так как в целом требуется специфицировать, спроектировать, реализовать и проверить меньшее количество системных компонентов. Но снижение стоимости проекта – это

только потенциальное преимущество повторного использования. Как видно из табл. 11.1, повторное использование ПО имеет ряд других преимуществ.

Таблица 11.1. Преимущества повторного использования ПО

Преимущество	Описание
Повышение надежности	Компоненты, повторно используемые в других системах, оказываются значительно надежнее новых компонентов. Они протестированы и проверены в разных условиях работы. Ошибки, допущенные при их проектировании и реализации, обнаружены и устранены еще при первом их применении. Поэтому повторное использование компонентов сокращает общее количество ошибок в системе
Уменьшение проектных рисков	Для уже существующих компонентов можно более точно прогнозировать расходы, связанные с их повторным использованием, чем расходы, необходимые на их разработку. Такой прогноз – важный фактор администрирования проекта, так как позволяет уменьшить неточности при предварительной оценке сметы проекта
Эффективное использование специалистов	Часть специалистов, выполняющих одинаковую работу в разных проектах, может заниматься разработкой компонентов для их дальнейшего повторного использования, эффективно применяя накопленные ранее знания
Соблюдение стандартов	Некоторые стандарты, такие как стандарты интерфейса пользователя, можно реализовать в виде набора стандартных компонентов. Например, можно разработать повторно используемые компоненты для реализации различных меню пользовательского интерфейса. Все приложения предоставляют меню пользователям в одном формате. Использование стандартного пользовательского интерфейса повышает надежность систем, так как, работая со знакомым интерфейсом, пользователи совершают меньше ошибок
Ускорение разработки	Часто для успешного продвижения системы на рынке необходимо как можно более раннее ее появление, причем независимо от полной стоимости ее создания. Повторное использование компонентов ускоряет создание систем, так как сокращается время на их разработку и тестирование

Для успешного проектирования и разработки ПО с повторным

использованием компонентов должны выполняться три основных условия.

1. Возможность поиска необходимых системных компонентов. В организациях должен быть каталог документированных компонентов, предназначенных для повторного использования, который обеспечивал бы быстрый поиск нужных компонентов.

2. При повторном использовании необходимо удостовериться, что поведение компонентов предсказуемо и надежно. В идеале все компоненты, представленные в каталоге, должны быть сертифицированы, чтобы подтвердить соответствие определенным стандартам качества.

3. На каждый компонент должна быть соответствующая документация, цель которой – помочь разработчику получить нужную информацию о компоненте и адаптировать его к новому приложению. В документации должна содержаться информация о том, где используется данный компонент, и другие вопросы, которые могут возникнуть при повторном использовании компонента.

Успешное использование компонентов в приложениях Visual Basic, Visual C++ и Java продемонстрировало важность повторного использования. Разработка ПО, основанная на повторном использовании компонентов, становится широко распространенным рентабельным подходом к разработке программных продуктов [331, 3*].

Вместе с тем подходу к разработке ПО с повторным использованием компонентов присущ ряд недостатков и проблем (табл. 11.2), которые препятствуют запланированному сокращению расходов на разработку проекта.

Таблица 11.2. Проблемы повторного использования

Проблема	Описание
Повышение стоимости сопровождения системы	Недоступность исходного кода компонента может привести к увеличению расходов на сопровождение системы, так как повторно используемые системные элементы могут со временем оказаться не совместимыми с изменениями, производимыми в системе

Недостаточная инструментальная поддержка	CASE-средства не поддерживают разработку ПО с повторным использованием компонентов. Интегрирование этих средств с системой библиотек компонентов затруднительно или даже невозможно. Если процесс разработки ПО осуществляется с помощью CASE-средств, повторное использование компонентов можно полностью исключить
Синдром "изобретения велосипеда"	Некоторые разработчики ПО предпочитают переписать компоненты, так как полагают, что смогут при этом их усовершенствовать. Кроме того, многие считают, что создание программ "с нуля" перспективнее и "благороднее" повторного использования написанных другими программ
Содержание библиотеки компонентов	Заполнение библиотеки компонентов и ее сопровождение может стоить дорого. В настоящее время еще недостаточно хорошо продуманы методы классификации, каталогизации и извлечения информации о программных компонентах
Поиск и адаптация компонентов	Компоненты ПО нужно найти в библиотеке, изучить и адаптировать к работе в новых условиях, что "не укладывается" в обычный процесс разработки ПО

Из перечисленного выше следует, что повторное использование компонентов должно быть систематическим, плановым и включенным во все организационные программы организации-разработчика. В Японии повторное использование известно много лет [231] и является неотъемлемой частью "японского" метода разработки ПО [85]. Многие компании, например Hewlett-Packard, успешно применяют повторное использование в своих разработках [139]. Опыт этой компании представлен в фундаментальной книге [187].

Альтернативой повторному использованию программных компонентов является применение программных генераторов. Согласно этому подходу информация, необходимая для повторного использования, записывается в систему генератора программ с учетом знаний о той предметной области, где будет эксплуатироваться разрабатываемая система. В данном случае в системной спецификации должно быть точно указано, какие именно

компоненты выбраны для повторного использования, а также описаны их интерфейсы и то, как они должны компоноваться. На основе такой информации генерируется система ПО (рис. 11.1).

Рис. 11.1. Генерирование программ

Повторное использование, основанное на генераторах программ, возможно только тогда, когда можно идентифицировать предметные абстракции и их отображение в исполняемый код. Поэтому для компоновки и управления предметными абстракциями используются, как правило, проблемно-зависимые языки (например, языки четвертого поколения). Вот предметные области, в которых применение такого подхода может быть успешным:

1. **Генераторы приложений для обработки экономических данных**. На входе генератора – описание приложения на языке четвертого поколения или диалоговая система, где пользователь определяет экранные формы и способы обработки данных. На выходе – программа на каком-либо языке программирования, например COBOL или SQL.

2. **Генераторы программ синтаксического анализатора**. На входе генератора – грамматическое описание языка, на выходе – программа грамматического разбора языковых конструкций.

3. **Генераторы кодов CASE-средств**. На входе генераторов – архитектура ПО, а на выходе - программная реализация проектируемой системы.

Разработка ПО с использованием программных генераторов экономически выгодна, однако существенно зависит от полноты и корректности определения абстракций предметной области. Данный подход можно широко использовать в перечисленных выше предметных областях и

в меньшей степени при разработке систем управления и контроля [261]. Главное преимущество этого подхода состоит в относительной легкости разработки программ с помощью генераторов. Однако необходимость глубокого понимания предметной области и ее моделей ограничивает применимость данного метода

11.1. Покомпонентная разработка

Метод покомпонентной разработки ПО с повторным использованием компонентов появился в конце 1990-х годов как альтернатива объектно-ориентированному подходу к разработке систем, который не привел к повсеместному повторному использованию программных компонентов, как предполагалось изначально. Отдельные классы объектов оказались слишком детализированными и специфическими: их требовалось связывать с приложением либо во время компиляции, либо при компоновке системы. Использование классов обычно предполагает наличие детальных данных о классах, что делает доступным исходный код, но для коммерческих продуктов исходный код открыт очень редко. Несмотря на ранние оптимистические прогнозы, значительное развитие рынка отдельных программных объектов и компонентов так и не состоялось.

Компоненты более абстрактны, чем классы объектов. Поэтому их можно считать независимыми поставщиками сервисов. Если система запрашивает какой-либо сервис, вызывается компонент, предоставляющий этот сервис независимо от того, где выполняется компонент и на каком языке написан. Примером простейшего компонента может быть отдельная математическая функция, вычисляющая, например, квадратный корень числа. Для вычисления квадратного корня программа вызывает компонент, который может выполнить данное вычисление. На другом конце масштабной линейки компонентов находятся системы, которые предоставляют полный вычислительный сервис.

Взгляд на компонент как на поставщика сервисов определяется двумя основными характеристиками компонентов, допускающими их повторное использование.

1. **Компонент** – это независимо выполняемый программный объект. Исходный код компонента может быть недоступен, поэтому такой компонент не компилируется совместно с другими компонентами системы.

2. Компоненты объявляют свой интерфейс и все взаимодействия с ними осуществляются с его помощью. Интерфейс компонента описывается в терминах параметризованных операций, а внутреннее состояние компонента всегда скрыто.

Компоненты определяются через свои интерфейсы. В большинстве случаев компоненты можно описать в виде двух взаимосвязанных интерфейсов, как показано на рис. 11.2.

- **Интерфейс поставщика сервисов**, который определяет сервисы, предоставляемые компонентом.
- **Интерфейс запросов**, который определяет, какие сервисы доступны компоненту из системы, использующей этот компонент.

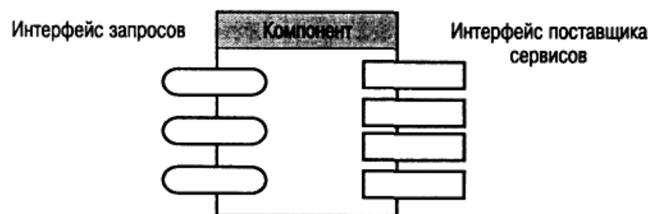

Рис. 11.2. Интерфейсы компонента

В качестве примера рассмотрим компонент (рис. 11.3), который предоставляет сервисы вывода документов на печать. В нашем примере поддерживаются следующие сервисы: печать документов, просмотр состояния очереди на конкретном принтере, регистрация и удаление принтеров из системы, передача документа с одного принтера на другой и удаление документа из очереди на печать. Очень важно, чтобы компьютерная платформа, на которой исполняется компонент, предоставляла сервис (назовем его **ФайлОпПринтер**), позволяющий извлечь файл

описания принтера, и сервис **КомПринтер**, передающий команды на конкретный принтер.

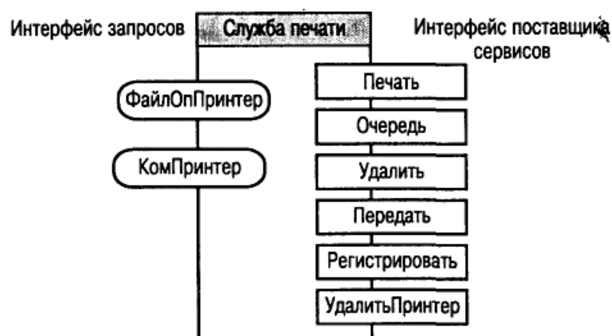

Рис. 11.3. Компонент службы печати

Компоненты могут существовать на разных уровнях абстракции – от простой библиотечной подпрограммы до целых приложений, таких как Microsoft Excel. В работе [236] определено пять уровней абстракции компонентов.

1. **Функциональная абстракция**. Компонент реализует отдельную функцию, например математическую. В сущности, интерфейсом поставщика сервисов здесь является сама функция.

2. **Бессистемная группировка**. В данном случае компонент – это набор слабо связанных между собой программных объектов и подпрограмм, например объявлений данных, функций и т.п. Интерфейс поставщика сервисов состоит из названий всех объектов в группировке.

3. **Абстракции данных**. Компонент является абстракцией данных или классом, описанным на объектно-ориентированном языке. Интерфейс поставщика сервисов состоит из методов (операций), обеспечивающих создание, изменение и получение доступа к абстракции данных.

4. **Абстракции кластеров**. Здесь компонент – это группа связанных классов, работающих совместно. Такие компоненты иногда называют структурой. Интерфейс поставщика сервисов является композицией всех интерфейсов объектов, составляющих структуру (см. раздел 11.1.1).

5. **Системные абстракции**. Компонент является полностью автономной системой. Повторное использование абстракций системного

уровня иногда называют повторным использованием коммерческих продуктов. Интерфейсом поставщика сервисов на этом уровне является так называемый программный интерфейс приложений (Application Programming Interface – API), который предоставляет доступ к системным командам и методам. Повторное использование коммерческих продуктов рассматривается в разделе 11.1.2.

Подход покомпонентной разработки систем можно интегрировать в общий процесс создания ПО путем добавления специальных этапов, на которых отбираются и адаптируются повторно используемые компоненты (рис. 11.4). Проектировщики системы разрабатывают системную архитектуру на высоком уровне абстракции, составляя спецификации системных компонентов. В дальнейшем эти спецификации используются для поиска повторно используемых компонентов. Они включаются в систему либо на уровне системной архитектуры, либо на более низких детализированных уровнях.

Рис. 11.4. Интеграция повторного использования компонентов в процесс разработки ПО

Хотя такой подход может привести к значительному увеличению количества повторно используемых компонентов, он, в сущности, противоположен подходу, применяемому в других инженерных дисциплинах, где процесс проектирования подчинен идее повторного использования. Перед началом этапа проектирования разработчики выполняют поиск компонентов, подходящих для повторного использования. Системная архитектура строится на основе уже имеющихся (готовых) компонентов (рис. 11.5).

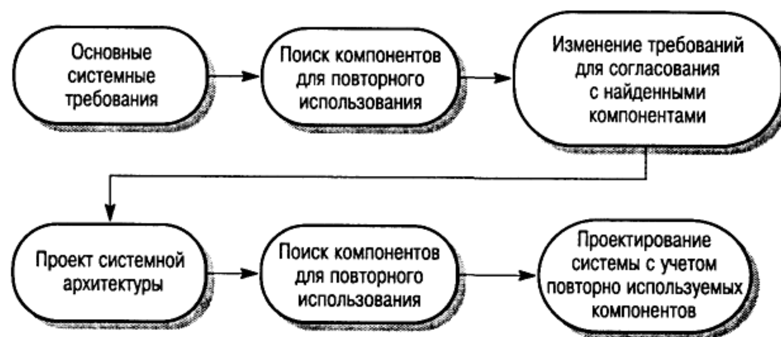

Рис. 11.5. Процесс проектирования с повторным использованием компонентов

В данном случае требования к системе изменяются с учетом имеющихся компонентов, выбранных для повторного использования. Системная архитектура также базируется на имеющихся компонентах. Конечно, такой подход предполагает определенные компромиссы в реализации требований. Хотя такая система может оказаться менее эффективной, чем система, разработанная без повторного использования компонентов, этот недостаток компенсируется более низкой стоимостью разработки, более высокими темпами создания системы и ее повышенной надежностью.

Обычно покомпонентный процесс реализации системы является либо процессом макетирования (прототипирования), либо процессом пошаговой разработки. Вместе с библиотеками компонентов можно использовать стандартные языки программирования, например Java. Альтернативой ему (и более распространенным языком) является язык сценариев, который специально разработан для интегрирования повторно используемых компонентов и обеспечивает быструю разработку программ.

Первым языком сценариев, разработанным для интеграции повторно используемых компонентов, был Unix shell [56]. После него разработано множество других языков сценариев, например Visual Basic и TCL/TK. В работе [268] обсуждаются преимущества языков сценариев, в том числе отсутствие определения типов данных, и тот факт, что они не компилируются, а интерпретируются.

Пожалуй, главной проблемой, связанной с покомпонентной разработкой систем, является их сопровождение и модернизация. При изменении требований к системе часто в компоненты необходимо внести изменения, соответствующие новым требованиям, однако в большинстве случаев это невозможно, поскольку исходный код компонентов недоступен. Также, как правило, не подходит альтернативный вариант: замена одного компонента другим. Таким образом, необходима дополнительная работа по адаптации повторно используемых компонентов, что приводит к повышению стоимости обслуживания системы. Однако, поскольку разработка с повторным использованием компонентов позволяет быстрее создавать ПО, организации согласны оплачивать дополнительные расходы на сопровождение и модернизацию систем.

11.2. Объектные структуры приложений

Первые сторонники объектно-ориентированной разработки ПО полагали, что наиболее подходящими абстракциями для повторного использования являются объекты. Однако, как следует из предыдущего раздела, объекты обычно слишком мелкие структурные единицы, чересчур "привязанные" в конкретному приложению. Очевидно, что в процессе объектно-ориентированного проектирования вместо повторного использования объектов намного эффективнее повторно использовать крупномодульные абстракции, так называемые объектные структуры приложения.

Объектные структуры приложения представляют собой структуры подсистем, состоящих из множества абстрактных и конкретных классов объектов и интерфейсов между ними [343]. Отдельные детали подсистем реализуются с помощью компонентов и обеспечивают конкретные реализации абстрактных классов. Объектные структуры, как правило, не

являются сами по себе приложениями. Обычно приложения строятся посредством интегрирования нескольких объектных структур.

Существует три основных класса объектных структур [113].

1. **Инфраструктуры систем**. Обеспечивают разработку инфраструктур для систем связи (коммуникационных систем), пользовательских интерфейсов и компиляторов [306].

2. **Интеграционные структуры**. Как правило, состоят из набора стандартов и связанных с ними классов объектов, обеспечивающих взаимодействие и обмен данными между компонентами. К этому типу структур относятся CORBA, COM и DCOM от Microsoft, а также Java Beans [264]. Данный тип объектных структур рассматривался в главе 11 при обсуждении архитектур распределенных объектов.

3. **Структуры инструментальных сред разработки приложений**. Связаны с отдельными прикладными областями, такими как телекоммуникации или финансы [30]. Они встраиваются в систему знаний области приложения и поддерживают разработку приложений конечного пользователя. Данные структуры связаны с семействами приложений, которые рассматриваются в разделе 11.2.

Объектная структура – это обобщенная структура, которую можно детализировать и расширить при создании конкретной подсистемы или приложения. Детализация объектной структуры обычно предполагает добавление конкретных классов, наследующих методы от абстрактных классов объектной структуры. Кроме того, определяются методы, которые вызываются в ответ на события, определенные объектной структурой.

На момент написания книги были достаточно хорошо разработаны инфраструктуры систем, особенно те из них, которые связаны с графическими интерфейсами пользователя. Их постепенно вытесняют структуры инструментальных сред разработки приложений [77]. Попытаемся разобрать наиболее эффективные представления и организацию данных объектных структур.

Одной из самых известных и распространенных объектных структур для графических интерфейсов пользователя (GUI) является объектная структура "модель-представление-контроллер" (рис. 11.6). Эта модель появилась в 1980-х годах как метод проектирования графических интерфейсов пользователя, который поддерживает различные представления объекта и различает взаимодействия с каждым из этих представлений.

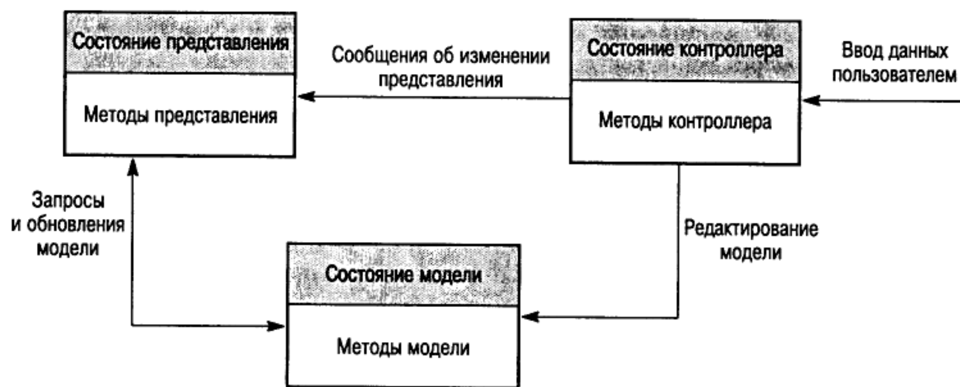

Рис. 11.6. Объектная структура "модель-представление-контроллер"

Объектные структуры часто реализуются в виде паттернов (см. раздел 11.2). Например, объектная структура "модель-представление-контроллер" включена в паттерн Обозреватель, описанный во врезке 11.1, а также в ряд других паттернов, подробно рассмотренных в работе [124].

Основные недостатки объектных структур – их сложность и время, необходимое для того, чтобы научиться работать с ними. Для полного изучения объектных структур может понадобиться несколько месяцев. Именно поэтому в больших организациях некоторые разработчики ПО специализируются по объектным структурам. Нет никаких сомнений в эффективности данного подхода к повторному использованию, однако высокие затраты на изучение объектных структур ограничивают его повсеместное распространение.

11.3. Повторное использование коммерческих программных продуктов

Термин "коммерческие программные продукты" можно применить к любому компоненту, созданному независимым производителем. Вместе с тем под этим термином часто подразумевается программное обеспечение системного уровня. Я также предпочитаю говорить о коммерческих системах. Функциональность, предлагаемая этими системами, намного шире функциональности более специализированных компонентов, и поэтому увеличивается потенциальный выигрыш, полученный от повторного использования.

Некоторые типы коммерческих систем используются повторно на протяжении многих лет. Лучшим примером тому, по-видимому, служат базы данных. Очень немногие разработчики создают собственные системы управления базами данных. Но до недавнего времени существовало лишь несколько больших коммерческих систем, таких как системы управления базами данных и мониторы телеобработки, используемые повторно.

Новые подходы к проектированию систем, предоставляющие программам доступ к системным функциям, показывают, что создание больших систем (например, систем электронной коммерции) посредством интегрирования ряда коммерческих систем сегодня рассматривается как один из приемлемых вариантов проектирования. Благодаря функциональности, предлагаемой этими системами, сокращение финансовых и временных затрат может достичь величины, сравнимой с разработкой нового ПО "с нуля". Более того, уменьшаются риски, так как коммерческие системы уже существуют и разработчики могут увидеть, удовлетворяют ли они предъявляемым к ним требованиям.

В принципе использование крупномодульных коммерческих систем не отличается от использования любого другого более специализированного компонента. Для этого необходимо изучить интерфейсы системы и использовать их для организации взаимодействия с другими системными компонентами, также необходимо разработать системную архитектуру, которая поддерживала бы коммерческие системы при совместной работе.

Однако тот факт, что коммерческие программные продукты представляют собой крупные системы и часто продаются как отдельные автономные системы, вносит дополнительные проблемы. При интеграции таких систем могут возникнуть, как минимум, четыре проблемы [42].

1. **Недостаточный контроль над функциональностью и производительностью коммерческих продуктов**. Хотя считается, что их интерфейсы известны, не исключена вероятность наличия скрытых операций, которые будут "пересекаться" с системными операциями. Решение этой проблемы может стать приоритетом для системных разработчиков, использующих коммерческие продукты, причем эта проблема, очевидно, не связана с производителем данного продукта.

2. **Проблемы, связанные с организацией, взаимодействия коммерческих систем**. Иногда сложно подобрать коммерческие продукты для совместной работы, поскольку каждый продукт разрабатывается на основе различных предположений по поводу его использования. В [127] приведены результаты эксперимента по интегрированию четырех различных коммерческих продуктов, при этом было установлено, что работа трех продуктов основывалась на одних и тех же событиях, однако все они использовали разные модели событий и каждый из них считал, что имеет первоочередной доступ к очереди событий. Как следствие, на разработку системы было потрачено в пять раз больше усилий, чем предполагалось, а на составление временного графика работы системы ушло два года вместо предполагавшихся шести месяцев.

3. **Отсутствие контроля за модификацией коммерческих продуктов.** Производители коммерческих продуктов принимают решения по изменению своих систем под давлением рынка. В частности, новые версии программных продуктов, разработанных для персональных компьютеров, создаются очень часто и могут оказаться не совместимыми с предыдущими версиями. Новые версии могут обладать дополнительной функциональностью, неподдерживаемой предыдущими версиями.

4. **Поддержка производителями коммерческих продуктов**. Уровень поддержки, оказываемой производителями коммерческих продуктов, варьируется в широких пределах, поскольку эти системы распространяются свободно. Поддержка производителей особенно важна в тех **случаях**, когда у разработчиков возникают проблемы, связанные с получением доступа к исходному коду и к подробной документации системы. Несмотря на то что производитель берет на себя обязательства по поддержке своих систем, изменение ситуации на рынке и экономических условий может привести к тому, что ему станет трудно продолжать выполнение взятых обязательств. Например, производитель коммерческой системы решил больше не поддерживать развитие какого-либо продукта из-за ограниченного спроса или, возможно, передал его другой компании, которая не хочет поддерживать все его продукты.

Конечно, маловероятно, что все эти проблемы возникнут в каждом случае использования коммерческих продуктов. По моим приблизительным подсчетам, во многих программных проектах, интегрирующих коммерческие системы, может появиться по крайней мере одна из перечисленных проблем. Соответственно преимущества в стоимости и времени выполнения работ по использованию коммерческих продуктов окажутся меньше, чем предполагалось в первоначальном оптимистическом варианте.

Все перечисленные проблемы являются проблемами жизненного цикла ПО и влияют только на начальную разработку системы. Но во многих случаях при использовании коммерческих продуктов расходы на сопровождение и модернизацию систем также могут возрасти [42], поскольку люди, участвующие в обслуживании системы, со временем все больше отдаляются от разработчиков исходной системы.

Несмотря на все эти проблемы, преимущества, получаемые при использовании коммерческих продуктов, весьма существенны, так как в этом случае можно сэкономить месяцы, а иногда и годы на разработке системы. Так как быстрое создание систем является одним из ключевых факторов для

большинства программных проектов, данный вид повторного использования компонентов, вероятно, получит со временем широкое практическое применение.

11.4. Разработка повторно используемых компонентов

Разработка идеального компонента для повторного использования должна быть процессом (основанным на опыте и знаниях о проблемах повторного использования) создания обобщенных компонентов, которые можно адаптировать для разных вариантов их использования.

Программный компонент, предназначенный для повторного использования, имеет ряд особенностей.

1. Должен отражать стабильные абстракции предметной области, т.е. фундаментальные понятия области приложения, которые меняются медленно. Например, в банковской системе абстракциями предметной области могут быть счета, форма вкладчика, бюллетени и т.п.

2. Должен скрывать способ представления своего состояния и предоставлять операции, которые позволяют обновлять состояния и получать к нему доступ. Например, в компоненте, который представляет счет в банке, должны быть операции, позволяющие выполнить запросы по остаткам на счетах, по изменениям в остатках счета, записать операции (транзакции) на счетах и т.п.

3. Должен быть максимально независимым. В идеале компонент должен быть настолько автономным, чтобы не нуждаться в других компонентах. В действительности такое выполнимо только для совсем простых компонентов, более сложные всегда зависят от других компонентов. Лучше всего имеющиеся зависимости свести к минимуму, особенно если они связаны с такими компонентами, как изменяемые функции операционной системы.

4. Все исключительные ситуации должны быть частью интерфейса компонента. Компоненты не должны сами обрабатывать исключения, так как в разных приложениях существуют разные требования для обработки исключительных ситуаций. Лучше определить те исключения, которые необходимо обрабатывать, и объявить их как часть интерфейса компонента. Например, простой компонент, реализующий структуру данных стека, должен определять и объявить исключениями переполнение и обнуление стека.

В большинстве существующих систем имеются большие сегменты кода, которые реализуют абстракции предметной области, однако их нельзя непосредственно использовать как компоненты. Причина в несоответствии программного кода модели, показанной на рис. 11.2, четко определенному интерфейсу запросов и поставщиков сервисов. Чтобы повторно использовать такие компоненты, как правило, необходимо построить упаковщик (программное средство для создания оболочки и стандартизации внешних обращений). Упаковщик скрывает исходный код и предоставляет интерфейс для внешних компонентов, открывающий доступ к предоставляемым сервисам.

При создании компонентов, предназначенных для повторного использования, предполагается предоставление очень общего интерфейса с операциями, которые обеспечивают разные способы использования компонентов. Чтобы сделать компоненты практичными в использовании, требуется минимальный интерфейс, простой для понимания. С другой стороны, предполагаемая возможность повторного использования усложняет компоненты и потому уменьшает их понятность. Поэтому разработчики компонентов, предназначенных для повторного использования, должны прийти к некоторому компромиссу между обобщенностью и понятностью компонентов.

11.5. Семейства приложений

Один из наиболее эффективных подходов к повторному использованию базируется на понятии семейства приложений. Семейство приложений, или серия программных продуктов, – это набор приложений, имеющих общую архитектуру, отражающую специфику конкретной предметной области (см. главу 10). Вместе с тем все приложения одной серии различны. Каждый раз при создании нового приложения повторно используется общее ядро семейства приложений. Далее в процессе разработки создается несколько дополнительных компонентов, а некоторые компоненты адаптируются согласно новым требованиям.

Существуют различные специализации семейств приложений, приведем некоторые из них.

1. **Платформенная специализация,** при которой для разных платформ разрабатываются свои версии приложения. Например, приложение может иметь версии для платформ Windows NT, Solaris или Linux. В данном случае функциональность приложения обычно не меняется; подвергаются изменениям только те компоненты, которые отвечают за взаимодействие с аппаратными средствами и операционной системой.

2. **Конфигурационная специализация**, при которой разные версии приложения создаются для управления различными периферийными устройствами. Например, разные версии системы безопасности могут зависеть от типа используемой радиосистемы. В этом случае изменяется функциональность приложения для того, чтобы соответствовать периферийным устройствам, и необходимо изменить те компоненты, которые связаны с периферийными устройствами.

3. **Функциональная специализация**, при которой создаются разные версии приложения для заказчиков с различными требованиями. Например, система автоматизации библиотек может иметь несколько модификаций в зависимости от того, где она применяется – в публичной,

справочной или университетской библиотеке. В этом случае изменяются компоненты, реализующие функциональность системы, и добавляются новые компоненты.

Чтобы наглядно представить эту технологию повторного использования, рассмотрим архитектуру системы управления ресурсами, изображенную на рис. 11.7. Подобные системы используются в организациях для отслеживания активов (ресурсов, запасов) и управления ими. Например, система управления ресурсами энергосистемы должна отслеживать все стационарные энергообъекты и соответствующее оборудование. В университетах система управления ресурсами может отслеживать оборудование, используемое в учебных лабораториях.

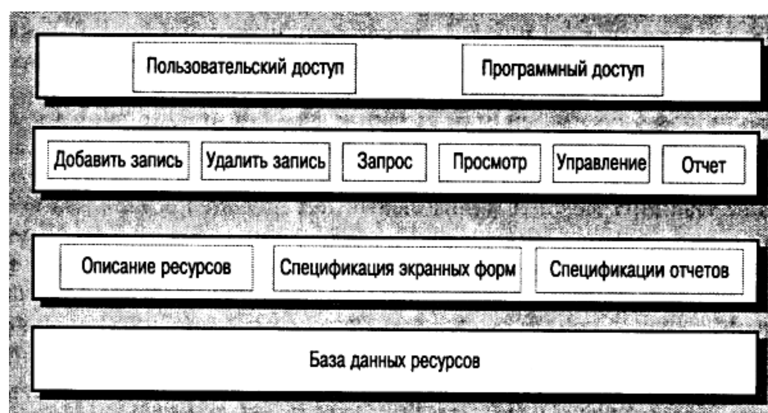

Рис. 11.7. Обобщенная система управления ресурсами

Очевидно, системы управления ресурсами будут отличаться друг от друга в зависимости от типа ресурсов и от информации, необходимой для управления ими. Например, в приложении для энергосистемы не нужны средства, позволяющие изменять размещение ресурсов, так как все объекты энергосистемы стационарны. Однако система учета ресурсов для университета должна иметь такую возможность, так как оборудование может переходить из одной лаборатории в другую.

Тем не менее все эти системы должны предоставлять основные средства для управления ресурсами: возможность добавления и удаления ресурсов, формирование запросов, просмотр базы данных ресурсов и формирование отчетов. Следовательно, архитектура систем управления

ресурсами будет одинакова для целого семейства приложений, в котором отдельные приложения поддерживают разные типы ресурсов.

Для того чтобы повторное использование систем было эффективным, на этапе создания архитектуры необходимо отделить основные средства системы от конкретной информации об управляемых ресурсах и от доступа пользователей к этой информации. На рис. 11.7 разделение достигнуто благодаря многоуровневой архитектуре, в которой на одном уровне встроены описания ресурсов, формирование экранных форм и отчетов. Верхние уровни системы используют эти описания в своих методах и не содержат конкретной информации о ресурсах. Посредством изменения уровня описаний можно создавать различные приложения управления ресурсами.

Конечно, такой тип систем можно выполнить в виде объектно-ориентированных, определив сначала объект абстрактного ресурса, а затем с помощью наследования – объект, зависящий от типа управляемого ресурса. В итоге такая архитектура будет немногим отличаться от архитектуры, изображенной на рис. 11.7. Однако для систем данного типа объектно-ориентированный подход не годится. Когда приложения рассчитаны на большие базы данных, содержащие миллионы записей, но относительно малое количество типов логических модулей (соответствующих объектам и сущностям), очевидно, что объектно-ориентированная система работает менее эффективно, чем системы с реляционными базами данных. На момент написания книги коммерческие объектно-ориентированные базы все еще остаются относительно медленными и не способными поддерживать сотни транзакций в секунду.

Подобно тому как посредством описаний новых ресурсов можно создавать новые члены семейства приложений, с помощью включения новых модулей на системном уровне в систему можно добавить новые функциональные возможности. Для создания библиотечной системы (рис. 11.8) я адаптировал систему управления ресурсами, показанную на рис. 11.7. В результате в систему добавлены новые возможности для выдачи и возврата

ресурсов и регистрации пользователей системой. На рис. 11.8 эти средства расположены справа. Так как программный доступ здесь не нужен, самый верхний уровень системы поддерживает доступ к ресурсам только на уровне пользователя.

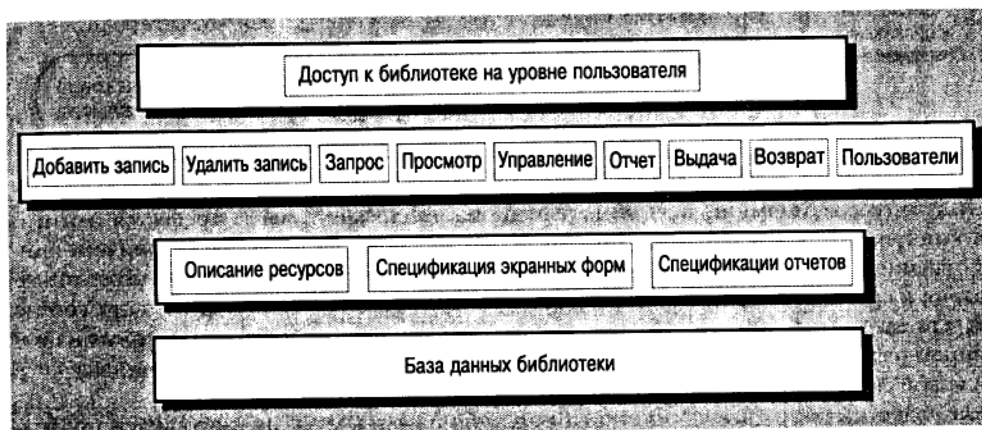

Рис. 11.8. Библиотечная система

В целом адаптация версии приложения в процессе его разработки приводит к тому, что значительная часть кода приложения используется повторно. Более того, накопленный опыт часто можно использовать для разработки других систем, поэтому объединение разработчиков ПО в отдельную группу сокращает процесс их обучения. Так как тесты для большинства компонентов приложения также можно использовать повторно, то полное время, необходимое на разработку приложения, значительно уменьшается.

Процесс адаптации семейства приложений для создания нового приложения состоит из нескольких этапов, представленных на рис. 11.9. Детали процесса могут значительно отличаться для разных прикладных областей и для различных организаций. Обобщенный процесс создания нового приложения состоит из следующих этапов.

1. **Определение требований для нового приложения**. Данный этап – обычный процесс разработки требований. Но так как система уже существует, естественно провести экспериментирование с ней и выявить те системные требования, которые необходимо изменить.

2. **Выбор наиболее подходящего члена семейства приложений**. Выполняется анализ требований, после чего выбирается наиболее подходящий член семейства, требующий внесения минимальных изменений.

3. **Пересмотр требований**. Как правило, появляется дополнительная информация, требующая внесения изменений в существующую систему, поэтому пересмотр требований на этом этапе позволяет уменьшить количество необходимых изменений.

4. **Адаптация выбранной системы**. *Для* системы разрабатываются новые модули, а существующие адаптируются к новым требованиям.

5. **Создание нового члена семейства приложений**. *Для* заказчика создается новый член семейства приложений. На этом этапе выполняется документирование ключевых особенностей системы, чтобы в дальнейшем ее можно было использовать как основу для разработки других систем.

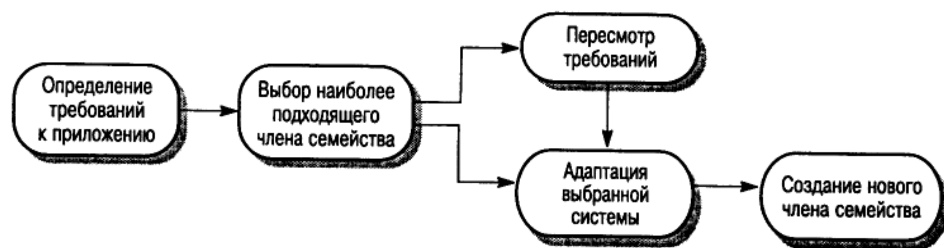

Рис. 11.9. Процесс разработки нового члена семейства приложений

В процессе создания нового члена семейства приложений часто требуется найти некий компромисс между наиболее полным использованием существующих приложений и выполнением конкретных требований для нового приложения. Чем более детальны требования к системе, тем меньше вероятность, что имеющиеся компоненты будут соответствовать этим требованиям. Но практически всегда можно достигнуть определенного компромисса и ограничить объем изменений, вносимых в систему, тогда процесс создания новой системы выполняется быстро и с небольшими затратами.

За редким исключением, семейства приложений создаются из имеющихся приложений, т.е. организация создает приложения и затем при необходимости использует их как основу для разработки нового приложения.

Но вместе с тем внесение изменений нарушает структуру приложения, поэтому рано или поздно принимается решение о создании семейства обобщенных приложений. В этом случае активно используются знания, собранные в процессе создания исходной группы приложений.

11.6. Проектные паттерны

Попытки повторно использовать действующие компоненты постоянно ограничиваются конкретными решениями, принятыми системными разработчиками. Решения могут относиться к отдельным алгоритмам, используемым при реализации компонентов, к объектам и к типам данных в интерфейсах компонентов. Если решения противоречат конкретным требованиям к компонентам, то повторное использование компонентов либо невозможно, либо делает систему неэффективной.

Один из способов решения данной проблемы – повторное использование более абстрактных структур, не содержащих деталей реализации. Такие структуры разрабатываются специально для того, чтобы соответствовать определенному приложению. Первые реализации этого подхода привели к документированию и опубликованию фундаментальных алгоритмов [201], а затем к документированию абстрактных типов данных, таких как стеки, деревья и списки [53]. Совсем недавно такой способ повторного использования обобщен в понятии паттерна.

Проектные паттерны (design patterns) [124] появились из идей, выдвинутых Кристофером Александером (Alexander, [8]), который предложил удобные и эффективные обобщенные паттерны разработки конкретных проектов. Паттерн – это описание проблемы и метода ее решения, позволяющее в дальнейшем использовать это решение в разных условиях. Паттерн не является детальной спецификацией. Скорее, он представляет собой описание, в котором аккумулированы знания и опыт. Паттерн – гарантированное решение общей проблемы.

При создании ПО проектные паттерны всегда связаны с объектно-ориентированным проектированием. Чтобы обеспечить всеобщность, паттерны часто используют такие объектно-ориентированные понятия, как наследование и полиморфизм. Однако общий принцип использования паттернов одинаково применим при любом подходе к проектированию ПО.

В [124] определены четыре основных элемента проектного паттерна.

1. Содержательное имя, которое является ссылкой на паттерн.

2. Описание проблемной области с перечислением всех ситуаций, в которых можно использовать паттерн.

3. Описание решений с отдельным описанием различных частей решения и их взаимоотношений. Это не описание конкретного проекта, а **шаблон** проектных решений, который можно использовать различными способами. В описании решений часто используются графические представления, которые показывают взаимоотношения между объектами и классами объектов в данном решении.

4. Описание "выходных" результатов – это описание результатов и компромиссов, необходимых для применения паттерна. Обычно используется для того, чтобы помочь разработчикам оценить конкретную ситуацию и выбрать для нее наиболее подходящий паттерн.

Часто в описание паттерна вводятся также разделы **мотивации** (обоснование полезности паттерна) и **применимости** (описание ситуаций, в которых можно использовать паттерн).

В качестве примера рассмотрим один из наиболее часто используемых паттернов, предложенных в работе [124], а именно Обозреватель (Observer) (см. врезку 11.1). Данный паттерн используется тогда, когда необходимы разные представления состояния объекта. Он выделяет нужный объект и представляет его в разных формах; на рис. 11.10 показаны два графических представления одного и того же набора данных.

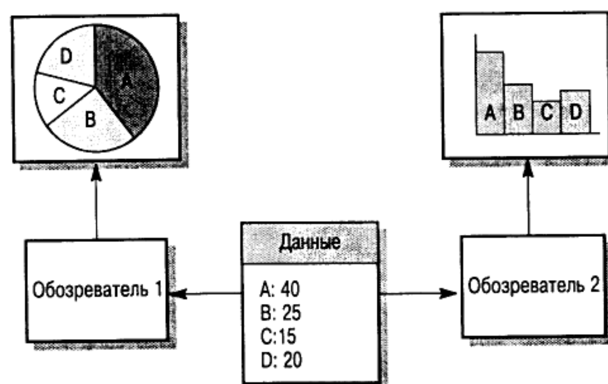

Рис. 11.10. Различные представления данных

Врезка 11.1. Описание паттерна Обозреватель

Имя паттера. Обозреватель

Описание. Отделяет отображение состояния объекта от самого и предлагает различные способы представления состояния. При изменении состояния объекта все представления автоматически обновляются, чтобы отобразить происшедшие изменения.

Описание проблемы. Во многих ситуациях требуется представить информацию о состоянии некоторого объекта несколькими разными способами, например используя графическое и табличное представления. Все представления взаимосвязаны и должны обновляться при изменении состояний.

Данный паттерн можно использовать во всех ситуациях, где требуется несколько разных представлений информации о состояний объекта и где нет необходимости знать форматы представления данных о состоянии объекта.

Описание решения. Структура паттерна показана на рис. 11.11. В нем определены два абстрактных объектам Subject (Данные) и Observer (Обозреватель), а также два конкретных объекта: ConcreteSubject (Конкретные данные) и ConcreteObserver (Конкретный обозреватель) которые наследуют свойства соответствующих абстрактных объектов. Отображаемое состояние поддерживается объектом ConcreteSubject, который также наследует методы от Subject, позволяющие ему добавлять и удалять объекты Observer (методы Attach и Detach) и выдавать оповещение при изменении состояния (метод Notify).

Объект ConcreteObserver обрабатывает копию состояния ConcreteSubject (копию subjectState, полученную с помощью метода GetState (Получить состояние)) и реализует метод Update (Обновить) интерфейса Observer, который позволяет сохранять копии состояния. ConcreteObserver автоматически отображает это состояние.

Результаты. Для оптимизации обозревателя необходима дополнительная информация об объектах. Изменения в формате отображаемых данных вызовут серию связанных изменений в созданных обозревателях.

Обычно в паттернах классы объектов и взаимоотношения между ними изображаются с помощью специальных графических нотаций. На рис. 11.11 представлен паттерн Обозреватель в нотации языка UML.

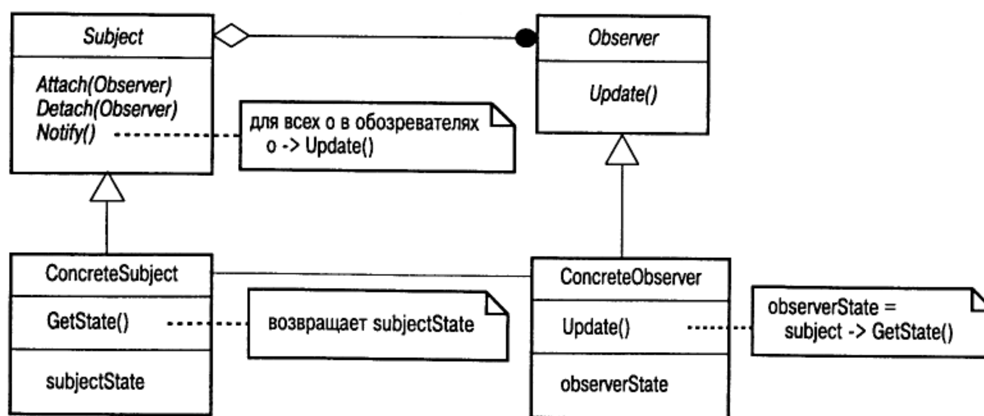

Рис. 11.11. Паттерн Обозреватель

Применение паттернов является весьма эффективным способом повторного использования; однако, по моему мнению, данный метод требует значительных затрат на освоение и может эффективно использоваться в проектировании систем только опытными программистами. Причина кроется в достаточно высокой сложности паттернов. Паттерн не похож на исполняемый компонент, для использования которого достаточно изучить только его интерфейс. Очевидно, что на изучение паттерна требуется определенное время.

Использовать паттерны могут только опытные программисты, поскольку лишь они способны распознать общие ситуации, в которых можно

применить тот или иной паттерн. Неопытные программисты, даже если они прочитали несколько книг, описывающих паттерны, на практике зачастую не могут определить, где следует использовать паттерн, а где необходимо нестандартное специальное решение.

Контрольные вопросы:

1) Укажите преимущества повторного использования ПО
2) Какие три основных условия должны выполняться для успешного проектирования и разработки ПО с повторным использованием компонентов
3) Укажите проблемы повторного использования
4) Перечислите пять уровней абстракции компонентов
5) Укажите три основных класса объектных структур
6) Что такое объектная структура?
7) Какие бывают специализации семейств приложений?
8) Из каких этапов состоит обобщенный процесс создания нового приложения?

Ключевые слова: *Повторно используемые приложения, повторно используемые компоненты, повторно используемые функции, компонент, функциональная абстракция, бессистемная группировка, абстракции данных, абстракции кластеров, системные абстракции, инфраструктуры систем, интеграционные структуры, объектная структура, платформенная специализация, конфигурационная специализация, функциональная специализация*

Keywords: *Re-used applications, reusable components, reusable functions, components, functional abstraction, haphazard grouping, data abstraction, abstraction cluster system abstraction infrastructure systems integration structures, object structure, specialization platform, the configuration specialization, functional specialization.*

Kalit so'zlar: *qayta ishlatiladigan ilovalar, qayta ishlatiladigan komponentlar, qayta ishlatiladigan funksiyalar, funksional abstraksiya, tizimsiz guruhlashtirish, ma'lumotlar abstraksiyasi, klasterlar abstraksiyasi, tizim abstraksiyalari, tizimlar infrostrukturasi, integratsion struktralar, obyekt strukturasi, platforma ixtisoslashtiruvi, konfiguratsion ixtisoslashtiruv, funksional ixtisoslashtiruv.*

Упражнения

1. Каковы основные технические и нетехнические факторы, затрудняющие повторное использование программного обеспечения?

2. Объясните, почему сокращение расходов при повторном использовании компонентов не прямо пропорционально размерам повторно используемых компонентов.

3. Приведите четыре аргумента против повторного использования компонентов.

4. Предположите возможные интерфейсы запросов и поставщиков сервисов для следующих компонентов.

- Компонент, реализующий счет в банке.
- Компонент, реализующий не зависящую от языка клавиатуру. Клавиатуры в разных странах имеют различную организацию клавиш и разные наборы символов.
- Компонент, реализующий средство управления версиями, рассмотренное в главе 29.

5. Чем отличается повторное использование объектной структуры приложения от повторного использования коммерческих продуктов? Почему иногда проще повторно использовать коммерческий продукт, чем объектную структуру приложения?

6. На примере метеорологической станции, описанной в главе 12, предложите архитектуру семейства приложений, которые связаны с удаленными наблюдением и сбором метеоданных.

7. На примере семейства приложений по управлению ресурсами (см. рис. 14.7) подумайте, какие методы необходимо добавить или изменить, чтобы можно было делать повторный заказ на отдельные виды ресурсов, если их количество становится меньше некоторой заданной величины.

8. Почему паттерны – эффективный способ повторного использования в проектировании? Каковы недостатки этого подхода?

9. Повторное использование увеличивает количество вопросов о собственности, охраняемой авторским и интеллектуальным правом. Если заказчик оплачивает разработчику ПО заказ на разработку какой-либо системы, кто имеет право повторно использовать созданный код? Имеет ли право разработчик использовать этот код в качестве основы для базового компонента? Какими должны быть механизмы оплаты труда разработчика повторно используемых компонентов?

Обсудите эти и другие этические вопросы, связанные с повторным использованием программного обеспечения.

ГЛАВА 12. ПРОЕКТИРОВАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Проектирование вычислительных систем охватывает широкий спектр проектных действий – от проектирования аппаратных средств до проектирования интерфейса пользователя. Организации-разработчики часто нанимают специалистов для проектирования аппаратных средств и очень редко для проектирования интерфейсов. Таким образом, специалистам по разработке ПО зачастую приходится проектировать и интерфейс пользователя. Если в больших компаниях в этот процесс вовлекаются специалисты по инженерной психологии, то в небольших компаниях услугами таких специалистов практически не пользуются.

Грамотно спроектированный интерфейс пользователя крайне важен для успешной работы системы. Сложный в применении интерфейс, как минимум, приводит к ошибкам пользователя. Иногда они просто отказываются работать с программной системой, несмотря на ее функциональные возможности. Если информация представляется сбивчиво или непоследовательно, пользователи могут понять ее неправильно, в результате чего их последующие действия могут привести к повреждению данных или даже к сбою в работе системы.

В 1982 году, во время выхода первой редакции этой книги, стандартным устройством взаимодействия между пользователем и программой был "беззвучный" буквенно-цифровой (текстовый) терминал, отображающий на черном поле символы зеленого или синего цвета. В то время интерфейсы пользователя были текстовыми или создавались в виде специальных форм. Сейчас почти все пользователи работают на персональных компьютерах. Все современные персональные компьютеры поддерживают графический интерфейс пользователя (graphical user interface – GUI), который подразумевает использование цветного графического экрана с высоким разрешением и позволяет работать с мышью и с клавиатурой.

Хотя текстовые интерфейсы еще достаточно широко применяются, особенно в наследуемых системах, в наше время пользователи предпочитают работать с графическим интерфейсом. В табл. 12.1 перечислены основные элементы GUI.

Таблица 12.1. Элементы графических интерфейсов пользователя

Элементы	Описание
Окна	Позволяют отображать на экране информацию разного рода
Пиктограммы	Представляют различные типы данных. В одних системах пиктограммы представляют файлы, в других – процессы
Меню	Ввод команд заменяется выбором команд из меню
Указатели	Мышь используется как устройство указания для выбора команд из меню и для выделения отдельных элементов в окне
Графические элементы	Могут использоваться совместно с текстовыми

Графические интерфейсы обладают рядом преимуществ.

1. Их относительно просто изучить и использовать. Пользователи, не имеющие опыта работы с компьютером, могут легко и быстро научиться работать с графическим интерфейсом.

2. Каждая программа выполняется в своем окне (экране). Можно переключаться из одной программы в другую, не теряя при этом данные, полученные в ходе выполнения программ.

3. Режим полноэкранного отображения окон дает возможность прямого доступа к любому месту экрана.

Цель данной главы - привлечь внимание разработчиков ПО к некоторым ключевым проблемам, лежащим в основе проектирования интерфейсов пользователя. Разработчики и программисты обычно компетентны в использовании таких технологий, как классы Swing в языке Java [103] или HTML [249], являющиеся основой реализации интерфейсов

пользователя. Однако эту технологию далеко не всегда применяют надлежащим образом, в результате чего интерфейсы пользователя получаются неэлегантными, неудобными и сложными в использовании.

В этой главе я приведу несколько рекомендаций по проектированию средств конечного пользователя, не рассматривая весь процесс проектирования этих средств. Из-за нехватки места рассматриваются только графические интерфейсы. Специальные интерфейсы, например для мобильных телефонов, телевизионных приемников, копировальной техники или факсимильных аппаратов, рассматриваться не будут. Здесь я сделаю только краткое введение в тему проектирования интерфейсов пользователя. Дополнительную информацию по данной теме можно найти в книгах [316, 99, 281].

На рис. 12.1 изображен итерационный процесс проектирования пользовательского интерфейса. Как отмечалось в главе 8, наиболее эффективным подходом к проектированию интерфейса пользователя является разработка с применением моделирования пользовательских функций. В начале процесса прототипирования создаются бумажные макеты интерфейса, затем разрабатываются экранные формы, моделирующие взаимодействие с пользователем. Желательно, чтобы конечные пользователи принимали активное участие в процессе проектирования интерфейса [258]. В одних случаях пользователи помогут оценить интерфейс; в других будут полноправными членами проектной группы [207, 138].

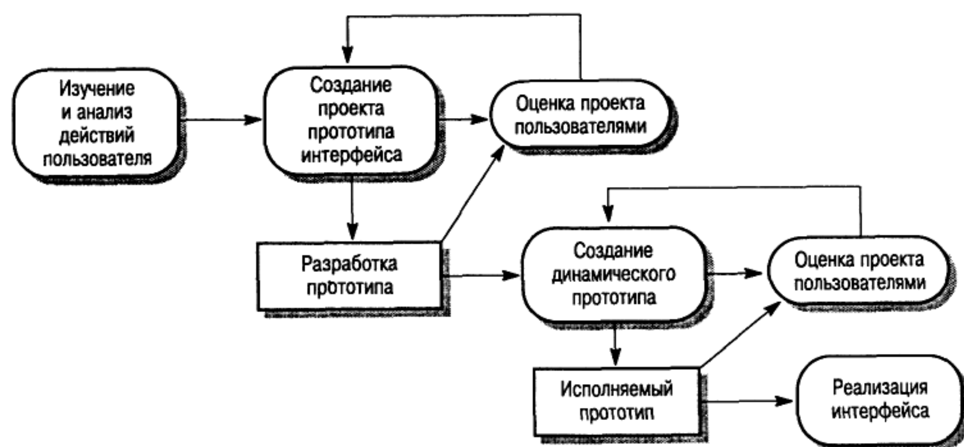

Рис. 12.1. Процесс проектирования интерфейса пользователя

Важным этапом процесса проектирования интерфейса пользователя является анализ деятельности пользователей, которую должна обеспечить вычислительная система. Не изучив того, что, с точки зрения пользователя, должна делать система, невозможно сформировать реалистический взгляд на проектирование эффективного интерфейса. Для анализа нужно (как правило, одновременно) применять различные методики, а именно: анализ задач [94], этнографический подход (см. главу 6) [328, 167], опросы пользователей и наблюдения за их работой.

12.1. Принципы проектирования интерфейсов пользователя

Разработчики интерфейсов всегда должны учитывать физические и умственные способности людей, которые будут работать с программным обеспечением. Люди на короткое время могут запомнить весьма ограниченный объем информации (см. главу 22) и совершают ошибки, если приходится вводить вручную большие объемы данных или работать в напряженных условиях. Физические возможности людей могут существенно различаться, поэтому при проектировании интерфейсов пользователя необходимо постоянно помнить об этом.

Основой принципов проектирования интерфейсов пользователя являются человеческие возможности. В табл. 12.2 представлены основные принципы, применимые при проектировании любых интерфейсов пользователя. Более детальный перечень "руководящих" правил проектирования интерфейсов можно найти в книге [316].

Таблица 12.2. Принципы проектирования интерфейсов пользователя

Принцип	Описание
Учет знаний пользователя	В интерфейсе необходимо использовать термины и понятия, взятые из опыта будущих пользователей системы

Согласованность	Интерфейс должен быть согласованным в том смысле, что однотипные (но различные) операции должны выполняться одним и тем же способом
Минимум неожиданностей	Поведение системы должно быть прогнозируемым
Способность к восстановлению	Интерфейс должен иметь средства, позволяющие пользователям восстановить данные после ошибочных действий
Руководство пользователя	Интерфейс должен предоставлять необходимую информацию в случае ошибок пользователя и поддерживать средства контекстно-зависимой справки
Учет разнородности пользователей	В интерфейсе должны быть средства для удобного взаимодействия с пользователями, имеющими разный уровень квалификации и различные возможности

Принцип учета знаний пользователя предполагает следующее: интерфейс должен быть настолько удобен при реализации, чтобы пользователям не понадобилось особых усилий, чтобы привыкнуть к нему. В интерфейсе должны использоваться термины, понятные пользователю, а объекты, управляемые системой, должны быть напрямую связаны с рабочей средой пользователя. Например, если разрабатывается система, предназначенная для авиадиспетчеров, то управляемыми объектами в ней должны быть самолеты, траектории полетов, сигнальные знаки и т.п. Основную реализацию интерфейса в терминах файловых структур и структур данных необходимо скрыть от конечного пользователя.

Принцип согласованности интерфейса пользователя предполагает, что команды и меню системы должны быть одного формата, параметры должны передаваться во все команды одинаково и пунктуация команд должна быть схожей. Такие интерфейсы сокращают время на обучение пользователей. Знания, полученные при изучении какой-либо команды или части приложения, можно затем применить при работе с другими частями системы.

В данном случае речь идет о согласованности низкого уровня. И создатели интерфейса всегда должны стремиться к нему. Однако желательна согласованность и более высокого уровня. Например, удобно, когда для всех типов объектов системы поддерживаются одинаковые методы (такие, как печать, копирование и т.п.). Однако полная согласованность невозможна и даже нежелательна [140]. Например, операцию удаления объектов рабочего стола целесообразно реализовать посредством их перетаскивания в корзину. Но в текстовом редакторе такой способ удаления фрагментов текста кажется неестественным.

Всегда нужно соблюдать следующий принцип: количество неожиданностей должно быть минимальным, так как пользователей раздражает, когда система вдруг начинает вести себя непредсказуемо. При работе с системой у пользователей формируется определенная модель ее функционирования. Если его действие в одной ситуации вызывает определенную реакцию системы, естественно ожидать, что такое же действие в другой ситуации приведет к аналогичной реакции. Если же происходит совсем не то, что ожидалось, пользователь либо удивляется, либо не знает, что делать. Поэтому разработчики интерфейсов должны гарантировать, что схожие действия произведут похожий эффект.

Очень важен принцип восстанавливаемости системы, так как пользователи всегда допускают ошибки. Правильно спроектированный интерфейс может уменьшить количество ошибок пользователя (например, использование меню позволяет избежать ошибок, которые возникают при вводе команд с клавиатуры), однако все ошибки устранить невозможно. В интерфейсах должны быть средства, по возможности предотвращающие ошибки пользователя, а также позволяющие корректно восстановить информацию после ошибок. Эти средства бывают двух видов.

1. **Подтверждение деструктивных действий**. Если пользователь выбрал потенциально деструктивную операцию, то он должен еще раз подтвердить свое намерение.

2. **Возможность отмены действий**. Отмена действия возвращает систему в то состояние, в котором она находилась до их выполнения. Не лишней будет поддержка многоуровневой отмены действий, так как пользователи не всегда сразу понимают, что совершили ошибку.

Следующий принцип – **поддержка пользователя**. Средства поддержки пользователей должны быть встроены в интерфейс и систему и обеспечивать разные уровни помощи и справочной информации. Должно быть несколько уровней справочной информации – от основ для начинающих до полного описания возможностей системы. Справочная система должна быть структурированной и не перегружать пользователя излишней информацией при простых запросах к ней (см. раздел 12.4).

Принцип учета разнородности пользователей предполагает, что с системой могут работать разные их типы. Часть пользователей работает с системой нерегулярно, время от времени. Но существует и другой тип– "опытные пользователи", которые работают с приложением каждый день по несколько часов. Случайные пользователи нуждаются в таком интерфейсе, который "руководил" бы их работой с системой, в то время как опытным пользователям требуется интерфейс, который позволил бы им максимально быстро взаимодействовать с системой. Кроме того, поскольку некоторые пользователи могут иметь разные физические недостатки, в интерфейсе должны быть средства, которые помогли бы им перенастроить интерфейс под себя. Это могут быть средства, позволяющие отображать увеличенный текст, замещать звук текстом, создавать кнопки больших размеров и т.п.

Принцип признания многообразия категорий пользователей может противоречить другим принципам проектирования интерфейсов, например согласованности интерфейса. Аналогично, необходимый уровень справочной информации для разных типов пользователей может радикально отличаться. Невозможно создать такую справочную систему, которая подошла бы всем пользователям. Разработчик интерфейса должен всегда быть готовым к

компромиссным решениям в зависимости от реальных пользователей системы.

12.2. Взаимодействие с пользователем

Разработчику интерфейса пользователя вычислительных систем необходимо решить две главные задачи: каким образом пользователь будет вводить данные в систему и как данные будут представлены пользователю. "Правильный" интерфейс должен обеспечивать и взаимодействие с пользователем, и представление информации.

В этом разделе обсуждаются вопросы взаимодействия системы с пользователем. Представление данных рассматривается в разделе 12.3. Интерфейс пользователя обеспечивает ввод команд и данных в вычислительную систему. На первых вычислительных машинах был только один способ ввода данных – через интерфейс командной строки, причем для взаимодействия с машиной использовался специальный командный язык. Такой способ годился только для опытных пользователей, поэтому позже были разработаны более упрощенные способы ввода данных. Все эти виды взаимодействия можно отнести к одному из пяти основных стилей взаимодействия [316].

1. **Непосредственное манипулирование**. Пользователь взаимодействует с объектами на экране. Например, для удаления файла пользователь просто перетаскивает его в корзину.

2. **Выбор из меню**. Пользователь выбирает команду из списка пунктов меню. Очень часто выбранная команда воздействует только на тот объект, который выделен (выбран) на экране. При таком подходе для удаления файла пользователь сначала выбирает файл, а затем команду на удаление.

3. **Заполнение форм**. Пользователь заполняет поля экранной формы. Некоторые поля могут иметь свое меню (выпадающее меню или списки). В форме могут быть командные кнопки, при щелчке мышью на которых инициируют некоторое действие. Чтобы удалить файл с помощью

интерфейса, основанного на форме, надо ввести в поле формы имя файла и затем щелкнуть на кнопке удаления, присутствующей в форме.

4. **Командный язык**. Пользователь вводит конкретную команду с параметрами, чтобы указать системе, что она должна дальше делать. Чтобы удалить файл, пользователь вводит команду удаления с именем файла в качестве параметра этой команды.

5. **Естественный язык**. Пользователь вводит команду на естественном языке. Чтобы удалить файл, пользователь может ввести команду "удалить файл с именем XXX".

Каждый из этих стилей взаимодействия имеет преимущества и недостатки и наилучшим образом подходит разным типам приложений и различным категориям пользователей [316]. В табл. 12.3 перечислены основные преимущества и недостатки перечисленных стилей взаимодействия и указаны типы приложений, в которых они обычно используются.

Конечно, стили взаимодействия редко используются в чистом виде, в одном приложении может использоваться одновременно несколько разных стилей. Например, в операционной системе Microsoft Window поддерживается несколько стилей: прямое манипулирование пиктограммами, представляющими файлы и папки, выбор команд из меню, ручной ввод некоторых команд, таких как команды конфигурирования системы, использование форм (диалоговых окон).

Таблица 12.3. Преимущества и недостатки стилей взаимодействия пользователя с системой

Стиль взаимодействия	Основные преимущества	Основные недостатки	Примеры приложений
Прямое манипулирование	Быстрое и интуитивно понятное взаимодействие. Легок в изучении	Сложная реализация. Подходит только там, где есть зрительный образ задач и объектов	Видеоигры; системы автоматического проектирования

Выбор из меню	Сокращение количества ошибок пользователя. Ввод с клавиатуры минимальный	Медленный вариант для опытных пользователей. Может быть сложным, если меню состоит из большого количества вложенных пунктов	Главным образом системы общего назначения
Заполнение форм	Простой ввод данных. Легок в изучении	Занимает пространство на экране	Системы управления запасами; обработка финансовой информации
Командный язык	Мощный и гибкий	Труден в изучении. Сложно предотвратить ошибки ввода	Операционные системы; библиотечные системы
Естественный язык	Подходит неопытным пользователям. Легко настраивается	Требует большого ручного набора	Системы расписания; системы хранения данных WWW

Пользовательские интерфейсы приложений World Wide Web базируются на средствах, предоставляемых языком HTML (язык разметки Web-страниц) вместе с другими языками, например Java, который связывает программы с компонентами Web-страниц. В основном интерфейсы Web-страниц проектируются для случайных пользователей и представляют собой интерфейсы в виде форм. В Web-приложениях можно создавать интерфейсы, в которых применялся бы стиль прямого манипулирования, однако к моменту написания книги проектирование таких интерфейсов представляло достаточно сложную в аспекте программирования задачу.

В принципе необходимо применять различные стили взаимодействия для управления разными системными объектами. Данный принцип

составляет основу модели Сихейма (Seeheim) пользовательских интерфейсов [278]. В этой модели разделяются представление информации, управление диалоговыми средствами и управление приложением. На самом деле такая модель является скорее идеальной, чем практической, однако почти всегда есть возможность разделить интерфейсы для разных классов пользователей (например, начинающих и опытных). На рис. 12.2 изображена подобная модель с разделенными интерфейсом командного языка и графическим интерфейсом, лежащая в основе некоторых операционных систем, в частности Linux.

Рис. 12.2. Множественный интерфейс

Разделение представления, взаимодействия и объектов, включенных в интерфейс пользователя, является основным принципом подхода "модель-представление-контроллер", который обсуждается в следующем разделе. Эта модель сравнима с моделью Сихейма, однако используется при реализации отдельных объектов интерфейса, а не всего приложения.

12.3. Представление информации

В любой интерактивной системе должны быть средства для представления данных пользователям. Данные в системе могут отображаться по-разному: например, вводимая информация может отображаться непосредственно на дисплее (как, скажем, текст в текстовом редакторе) или преобразовываться в графическую форму. Хорошим тоном при

проектировании систем считается отделение представления данных от самих данных. До некоторой степени разработка такого ПО противоречит объектно-ориентированному подходу, при котором методы, выполняемые над данными, должны быть определены самими данными. Однако в нащем случае предполагается, что разработчик объектов всегда знает наилучший способ представления данных; хотя это, конечно, не всегда так. Часто определить наилучший способ представления данных конкретного типа довольно трудно, в таком случае объектные структуры не должны быть "жесткими".

После того как представление данных в системе отделено от самих данных, изменения в представлении данных на экране пользователя происходят без изменения самой системы (рис. 12.3).

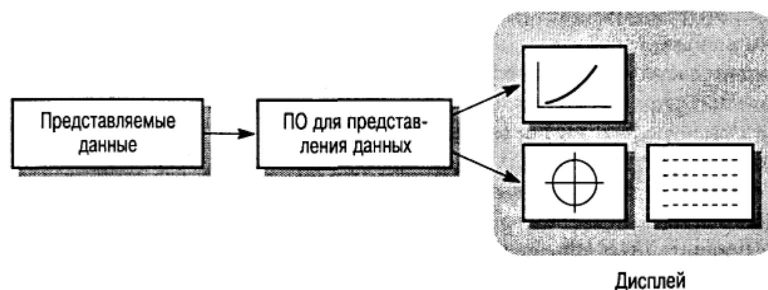

Рис. 12.3. Представление данных

Подход "модель-представление-контроллер" (МПК), представленный на рис. 12.4, получил первоначальное применение в языке Smalltalk как эффективный способ поддержки различных представлений данных [130]. Пользователь может взаимодействовать с каждым типом представления. Отображаемые данные инкапсулированы в объекты модели. Каждый объект модели может иметь несколько отдельных объектов представлений, где каждое представление – это разные отображения модели. Я уже иллюстрировал этот подход в предыдущей главе, где рассматривались объектно-ориентированные структуры приложений.

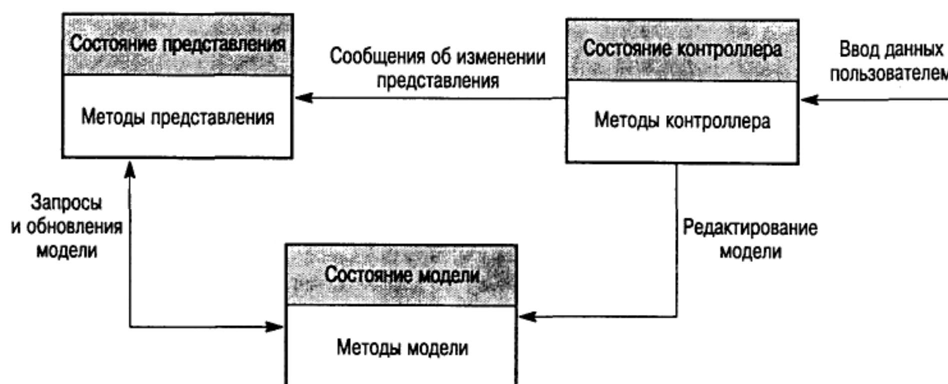

Рис. 12.4. Модель МПК взаимодействия с пользователем

Каждое представление имеет связанный с ним объект контроллера, который обрабатывает введенные пользователем данные и обеспечивает взаимодействие с устройствами. Такая модель может представить числовые данные, например, в виде диаграмм или таблиц. Модель можно редактировать, изменяя значения в таблице или параметры диаграммы. Такой подход рассмотрен в главе 14 при описании паттерна Обозреватель (раздел 14.3).

Чтобы найти наилучшее представление информации, необходимо знать, с какими данными работают пользователи и каким образом они применяются в системе. Принимая решение по представлению данных, разработчик должен учитывать ряд факторов.

1. Что нужно пользователю – точные значения данных или соотношения между значениями?

2. Насколько быстро будут происходить изменения значений данных? Нужно ли немедленно показывать пользователю изменение значений?

3. Должен ли пользователь предпринимать какие-либо действия в ответ на изменение данных?

4. Нужно ли пользователю взаимодействовать с отображаемой информацией посредством интерфейса с прямым манипулированием?

5. Информация должна отображаться в текстовом (описательно) или числовом формате? Важны ли относительные значения элементов данных?

Если данные не изменяются в течение сеанса работы с системой, их можно представить либо в графическом, либо в текстовом виде, в зависимости от типа приложения. Текстовое представление данных занимает на экране мало места, но в таком случае данные нельзя охватить одним взглядом. С помощью разных стилей представления неизменяемые данные следует отделить от динамически изменяющихся данных. Например, статические данные можно выделить особым шрифтом, подчеркнуть особым цветом либо обозначить пиктограммами.

Если требуется точная цифровая информация и данные изменяются относительно медленно, их можно отображать в текстовом виде. Там, где данные изменяются быстро, обычно используется графическое представление.

В качестве примера рассмотрим систему, которая помесячно записывает и подбивает итоги по данным продаж некой компании. На рис. 12.5 видно, что одни и те же данные можно представить в виде текста и в графическом виде.

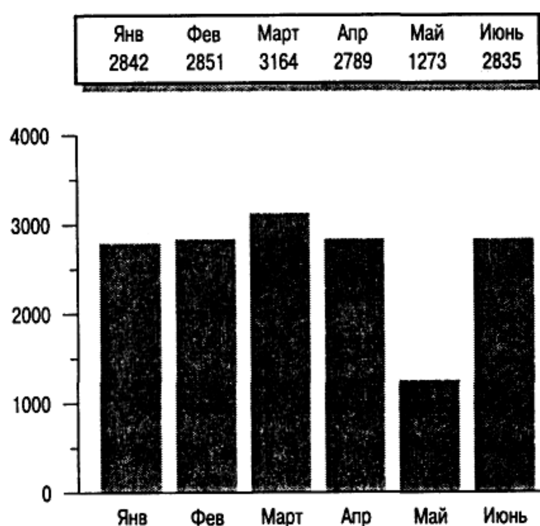

Рис. 12.5. Альтернативные представления данных

Менеджерам, изучающим данные о продажах, обычно больше нужны тенденции изменения или аномальные данные, чем их точные значения. Графическое представление этой информации в виде гистограммы позволяет выделить аномальные данные за март и май, значительно отличающиеся от

остальных данных. Как видно из рис. 12.5, данные в текстовом представлении занимают меньше места, чем в графическом.

Динамические изменения числовых данных лучше отображать графически, используя аналоговые представления. Постоянно изменяющиеся цифровые экраны сбивают пользователей с толку, поскольку точные значения данных быстро не воспринимаются. Графическое отображение данных при необходимости можно дополнить точными значениями. Различные способы представления изменяющихся числовых данных показаны на рис. 12.6.

Непрерывные аналоговые отображения помогают наблюдателю оценить относительные значения данных. На рис. 12.7 числовые значения температуры и давления приблизительно одинаковы. Но при графическом отображении видно, что значение температуры близко к максимальному, в то время как значение давления не достигло даже 25% от максимума. Обычно, кроме текущего значения, наблюдателю требуется знать максимальные (или минимальные) возможные значения. Он должен в уме вычислять относительное состояние считываемых данных. Дополнительное время, необходимое для расчетов, может привести к ошибкам оператора в стрессовых ситуациях, когда возникают проблемы и на дисплее отображаются аномальные данные.

Рис. 12.6. Способы представления динамически изменяющихся числовых данных

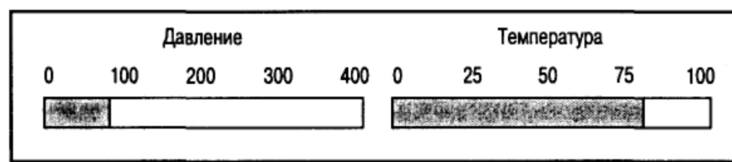

Рис. 12.7. Графическое представление данных, показывающее значения по отношению к максимальным

При представлении точных буквенно-цифровых данных для выделения особой информации можно использовать графические элементы. Вместо обычной строки данные лучше поместить в прямоугольник или отметить пиктограммой (рис. 12.8). Прямоугольник с сообщением помещается поверх текущего экрана, тем самым привлекая к нему внимание пользователя.

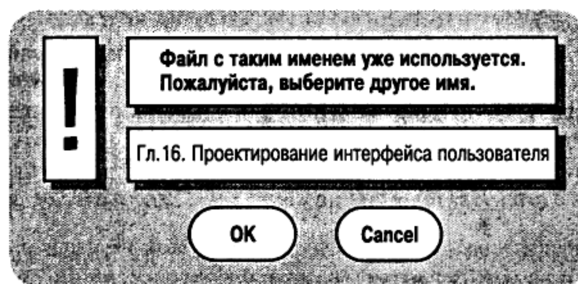

Рис. 12.8. Выделение буквенно-цифровых данных

Выделение информации с помощью графических элементов можно также использовать для привлечения внимания к изменениям, происходящим в разных частях экрана. Но, если изменения происходят очень быстро, не следует использовать графические элементы, поскольку быстрые изменения могут привести к наложению экранов, что сбивает с толку и раздражает пользователей.

При представлении больших объемов данных можно использовать разные приемы визуализации, которые указывают на родственные элементы данных. Разработчики интерфейсов должны помнить о возможностях визуализации, особенно если интерфейс системы должен отображать физические сущности (объекты). Вот несколько примеров визуализации данных.

1. Отображение метеорологических данных, собранных из разных источников, в виде метеорологических карт с изобарами, воздушными фронтами и т.п.

2. Графическое отображение состояния телефонной сети в виде связанного множества узлов.

3. Визуализация состояния химического процесса с показом давлений и температур в группе связанных между собой резервуаров и труб.

4. Модель молекулы и манипулирование ею в трехмерном пространстве посредством системы виртуальной реальности.

5. Отображение множества Web-страниц в виде дерева гипертекстовых ссылок [208].

В книге [316] содержится хорошее описание различных подходов к визуализации. Там же представлены различные классы визуализации, часто используемые на практике. В частности, к ним относится визуализация данных с использованием двух- и трехмерных деревьев и сетей. Большинство из них связаны с отображением больших объемов данных, управляемых компьютером. Наиболее широко визуализация применяется в интерфейсах для представления некоторых физических структур, например молекулярной структуры лекарства, телекоммуникационных сетей и т.п. Трехмерные представления, для создания которых используется специальное оборудование виртуальной реальности, являются особенно эффективным способом визуализации.

12.4. Использование в интерфейсах цвета

Во всех интерактивных системах, независимо от их назначения, поддерживаются цветные экраны, поэтому в пользовательских интерфейсах часто используются различные цвета. В некоторых системах цвета применяют в основном для выделения определенных элементов (например, в текстовых редакторах для выделения фрагментов текста); в других системах (таких, как системы автоматического проектирования) цветами обозначают разные уровни проектов.

Правильное использование цветов делает интерфейс пользователя более удобным для понимания и управления. Вместе с тем использование цветов может быть неправильным, в результате чего создаются интерфейсы,

которые визуально неприглядны и даже провоцируют ошибки. Основным принципом разработчиков интерфейсов должно быть осторожное использование цветов на экранах. В работе [316] дается 14 правил эффективного использования цвета в пользовательских интерфейсах. Вот наиболее важные из них:

1. **Используйте ограниченное количество цветов**. Для окон не следует использовать более четырех или пяти разных цветов, в интерфейсе системы не должно быть более семи цветов.

2. **Используйте разные цвета для показа изменений в состоянии системы**. Если на экране изменились цвета, значит, произошло какое-то событие. Выделение цветом особенно важно в сложных экранах, в которых отображаются сотни разных объектов.

3. **Для помощи пользователю используйте цветовое кодирование.** Если пользователям необходимо выделять аномальные элементы, выделите их цветом; если требуется найти подобные элементы, выделите их одинаковым цветом.

4. **Используйте цветовое кодирование продуманно и последовательно.** Если в какой-либо части системы сообщения об ошибке отображаются, например, красным цветом, то во всех других частях подобные сообщения должны отображаться таким же цветом. Тогда красный цвет не следует использовать где-либо еще. Если же красный цвет используется еще где-то в системе, пользователь может интерпретировать появление красного цвета как сообщение об ошибке. Следует помнить, что у определенных типов пользователей имеются свои представления о значении отдельных цветов.

5. **Осторожно используйте дополняющие цвета.** Физиологические особенности человеческого глаза не позволяют одновременно сфокусироваться на красном и синем цветах. Поэтому последовательность красных и синих изображений вызывает зрительное

напряжение. Некоторые комбинации цветов также могут визуально нарушать или затруднять чтение.

Чаще всего разработчики интерфейсов допускают две ошибки: привязка значения к определенному цвету и использование большого количества цветов на экране. Использовать цвета для представления значения не следует по двум причинам. Около 10% людей имеют нечеткое представление о цветах и поэтому могут неправильно интерпретировать значение. У разных групп людей различное восприятие цветов; кроме того, в разных профессиях существуют свои соглашения о значении отдельных цветов. Пользователи на основании полученных знаний могут неадекватно интерпретировать один и тот же цвет. Например, водителем красный цвет воспринимается как **опасность.** А у химика красный цвет означает **горячий**.

При использовании слишком ярких цветов или слишком большого их количества отображения становятся путанными. Многообразие цветов сбивает с толку пользователя (так, например, на некоторые абстрактные картины нельзя смотреть длительное время без напряжения) и вызывает у него зрительное утомление. Непоследовательное использование цветов также дезориентирует пользователя.

12.5. Средства поддержки пользователя

В первом разделе этой главы был предложен принцип проектирования, согласно которому интерфейс пользователя должен всегда обеспечивать некоторый тип оперативной справочной системы. **Справочные системы** – один из основных аспектов проектирования интерфейса пользователя. Справочную систему приложения составляют:

- сообщения, генерируемые системой в ответ на действия пользователя;
- диалоговая справочная система;
- документация, поставляемая с системой.

Поскольку проектирование полезной и содержательной информации для пользователя – дело весьма серьезное, оно должно оцениваться на том же уровне, что и архитектура системы или программный код. Проектирование сообщений требует значительного времени и немалых усилий. Уместно привлекать к этому процессу профессиональных писателей и художников-графиков. При проектировании сообщений об ошибках или текстовой справки необходимо учитывать факторы, перечисленные в табл. 12.4.

Таблица 12.4. Факторы проектирования текстовых сообщений

Фактор	**Описание**
Содержание	Справочная система должна знать, что делает пользователь, и реагировать на его действия сообщениями соответствующего содержания
Опыт пользователя	Если пользователи хорошо знакомы с системой, им не нужны длинные и подробные сообщения. В то же время начинающим пользователям такие сообщения покажутся сложными, малопонятными и слишком краткими. В справочной системе должны поддерживаться оба типа сообщений, а также должны быть средства, позволяющие пользователю управлять сложностью сообщений
Профессиональный уровень пользователя	Сообщения должны содержать сведения, соответствующие профессиональному уровню пользователей. В сообщениях для пользователей разного уровня необходимо применять разную терминологию
Стиль сообщений	Сообщения должны иметь положительный, а не отрицательный оттенок. Всегда следует использовать активный, а не пассивный тон обращения. В сообщениях не должно быть оскорблений или попыток пошутить
Культура	Разработчик сообщений должен быть знаком с культурой той страны, где продается система. Сообщение, вполне уместное в культуре одной страны, может оказаться неприемлемым в другой

12.6. Сообщения об ошибках

Первое впечатление, которое пользователь получает при работе с программной системой, основывается на сообщениях об ошибках. Неопытные пользователи, совершив ошибку, должны понять появившееся сообщение об ошибке.

Новички и опытные пользователи должны предвидеть ситуации, при которых могут возникнуть сообщения об ошибках. Например, пусть пользователем системы является медсестра госпиталя, работающая в отделении интенсивной терапии. Обследование пациентов выполняется на соответствующем оборудовании, связанном с вычислительной системой. Чтобы просмотреть текущее состояние пациента, пользователь системы выбирает пункт меню Показать и набирает имя пациента в поле ввода (рис. 12.9).

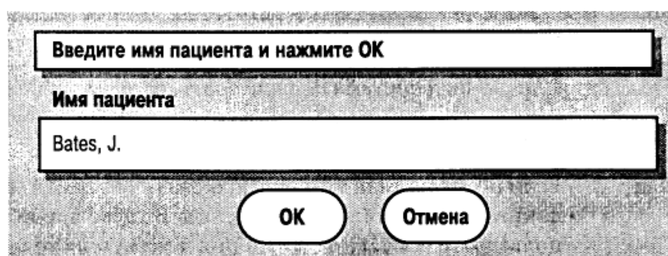

Рис. 12.9. Ввод имени пациента

Пусть медсестра ввела имя пациента Bates, вместо Pates. Система не находит пациента с таким именем и генерирует сообщение об ошибке. Сообщения об ошибке должны быть всегда вежливыми, краткими, последовательными и конструктивными, не содержать оскорблений. Не следует также использовать звуковые сигналы или другие звуки, которые могут сбить с толку пользователя. Неплохо включить в сообщения варианты исправления ошибки. Сообщение об ошибке должно быть связано с контекстно-зависимой справкой.

На рис. 12.10 показаны примеры двух сообщений об ошибке. Сообщение, расположенное слева, спроектировано плохо. Оно негативно (обвиняет пользователя в совершении ошибки), не адаптировано к уровню знаний и опытности пользователя, не учитывает содержания ошибки. В этом сообщении не предлагаются способы исправления сложившейся ситуации.

Кроме того, в сообщении использованы специфические термины (номер ошибки), не понятные пользователю. Сообщение справа гораздо лучше. Оно положительно, в нем используются медицинские термины и предлагается простой способ исправления ошибки посредством щелчка на одной из кнопок. В случае необходимости пользователь может вызвать справку.

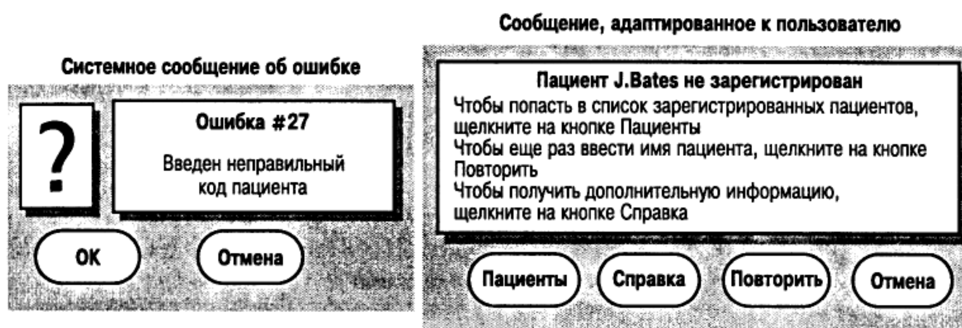

Рис. 12.10. Примеры сообщений об ошибках

12.7. Проектирование справочной системы

При получении сообщения об ошибке пользователь часто не знает, что делать, и обращается к справочной системе за информацией. Справочная система должна предоставлять разные типы информации: как ту, что помогает пользователю в затруднительных ситуациях, так и конкретную информацию, которую ищет пользователь. Для этого справочная система должна иметь разные средства и разные структуры сообщений.

Справочная система должна обеспечивать пользователю несколько различных точек входа (рис. 12.11). Пользователь может войти в нее на верхнем уровне ее иерархической структуры и здесь обозреть все разделы справочной информации. Другие точки входа в справочную систему – с помощью окон сообщений об ошибках или путем получения описания конкретной команды приложения.

Все справочные системы имеют сложную сетевую структуру, в которой каждый раздел справочной информации может ссылаться на несколько других информационных разделов. Структура такой сети, как

правило, иерархическая, с перекрестными ссылками, как показано на рис. 12.11. Наверху структурной иерархии содержится общая информация, внизу – более подробная.

При использовании справочной системы возникают проблемы, связанные с тем, что пользователи входят в сеть после совершения ошибки и затем перемещаются по сети справочной системы. Через некоторое время они запутываются и не могут определить, в каком месте справочной системы находятся. В такой ситуации пользователи должны завершить сеанс работы со справочной системой и вновь начать работу с некоторой известной точки справочной сети.

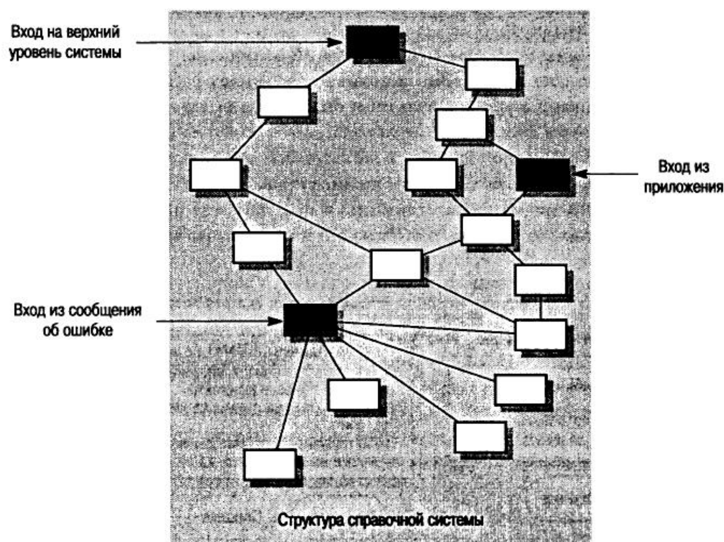

Рис. 12.11. Точки входа в справочную систему

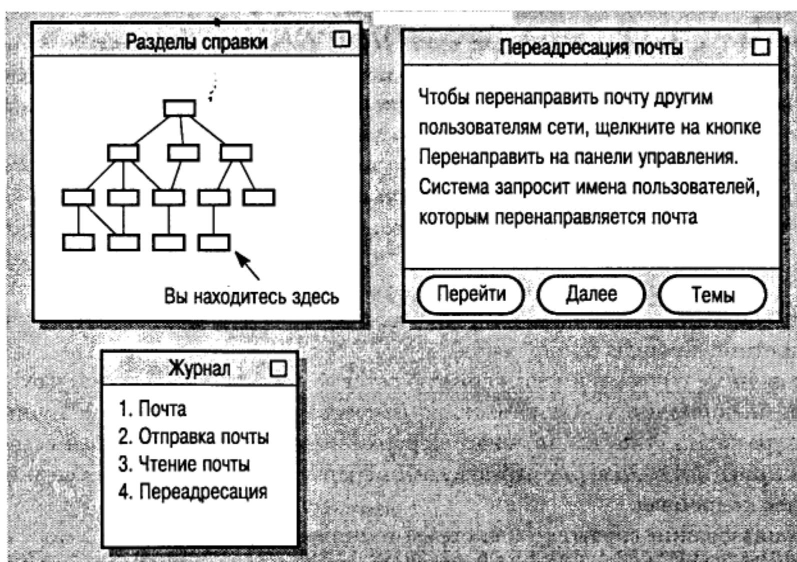

Рис. 12.12. Окна справочной системы

Отображение справочной информации в нескольких окнах упрощает подобную ситуацию. На рис. 12.12 показан экран, на котором расположены три окна справки. Однако пространство экрана всегда ограничено и разработчику следует помнить, что дополнительные окна могут скрыть другую нужную информацию.

Тексты справочной системы необходимо готовить совместно с разработчиками приложения. Справочные разделы не должны быть просто воспроизведением руководства пользователя, поскольку информация на бумаге и на экране воспринимается по-разному. Сам текст (а также его расположение и стиль) должен быть тщательно продуман, чтобы его можно было прочитать в окне относительно малого размера. Раздел справки **Переадресация** почты на рис. 12.12 сравнительно небольшой – в любом справочном разделе должна быть только самая необходимая информация. В окне, отображающем справочный раздел, расположены три кнопки: для показа дополнительной информации, для перемещения по тексту раздела и для вызова списка справочных тем.

В окне **Журнал*** показан список уже просмотренных разделов. Можно вернуться в один из них, выбрав соответствующий пункт из списка. Окно навигации по справочной системе – это графическая "карта" сети справочной системы. Текущая позиция на карте должна быть выделена цветом, тенями или, как в нашем случае, подписью.

У пользователей есть несколько возможностей перемещения между разделами справочной системы: можно перейти к разделу непосредственно из отображаемого раздела, можно выбрать нужный раздел из окна **Журнал**, чтобы просмотреть его еще раз, и, наконец, можно выбрать соответствующий узел на карте справочной сети и перейти к этому узлу.

Справочную систему можно реализовать в виде группы связанных Web-страниц или с помощью обобщенной гипертекстовой системы, интегрированной с приложением. Иерархическая структура легко реализуется в виде гипертекстовых ссылок. Web-системы имеют

преимущества: они просты в реализации и не требуют специального программного обеспечения. Однако при создании контекстно-зависимой справки могут возникнуть трудности при связывании ее с приложением.

12.8. Документация пользователя

Строго говоря, документация не является частью пользовательского интерфейса, однако проектирование оперативной справочной поддержки вместе с документацией является хорошим правилом. Системные руководства предоставляют более подробную информацию, чем диалоговые справочные системы, и строятся так, чтобы быть полезными пользователям разного уровня.

Для того чтобы документация, поставляемая совместно с программной системой, была полезна всем системным пользователям, она должна содержать пять описанных ниже документов (или, может быть, глав в одном документе) (рис. 12.13).

1. **Функциональное описание**, в котором кратко представлены функциональные возможности системы. Прочитав функциональное описание и вводное руководство, пользователь должен определить, та ли это система, которая ему нужна.

2. **Документ по инсталляции системы**, *в* котором содержится информация по установке системы. Здесь должны быть сведения о дисках, на которых поставляется система, описание файлов, находящихся на этих дисках, и минимальные требования к конфигурации. В документе должна быть инструкция по инсталляции и более подробная информация по установке файлов, зависящих от конфигурации системы.

3. **Вводное руководство**, представляющее неформальное введение в систему, описывающее ее "повседневное" использование. В этом документе должна содержаться информация о том, как начать работу с системой, как использовать общие возможности системы. Все описания должны быть снабжены примерами и содержать сведения о том, как восстановить систему после ошибки и как начать заново работу. В книге [68] предложен

эффективный способ составления вводного руководства, при котором основное внимание уделено восстановлению системы после ошибок, а вся остальная информация, необходимая пользователям, сводится к минимуму.

4. **Справочное руководство**, в котором описаны возможности системы и их использование, представлен список сообщений об ошибках и возможные причины их появления, рассмотрены способы восстановления системы после выявления ошибок.

5. **Руководство администратора**, необходимое для некоторых типов программных систем. В нем дано описание сообщений, генерируемых системой при взаимодействии с другими системами, и описаны способы реагирования на эти сообщения. Если в систему включена аппаратная часть, то в руководстве администратора должна быть информация о том, как выявить и устранить неисправности, связанные с аппаратурой, как подключить новые периферийные устройства и т.п.

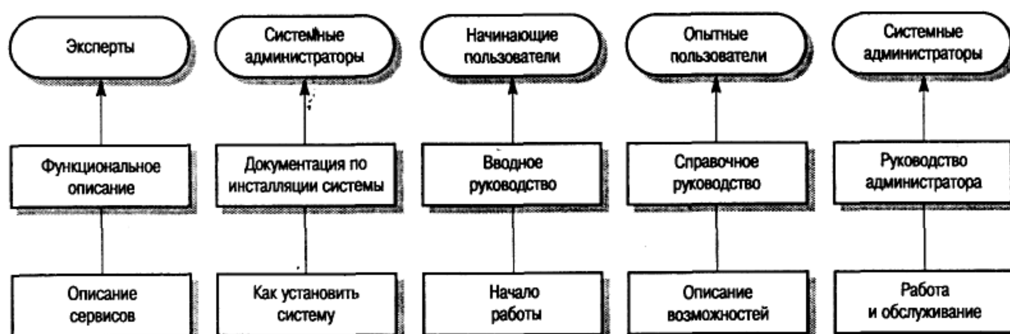

Рис. 12.13. Типы пользовательской документации

Вместе с перечисленными руководствами необходимо предоставлять другую удобную в работе документацию. Для опытных пользователей системы удобны разного вида предметные указатели, которые помогают быстро просмотреть список возможностей системы и способы их использования.

12.9. Оценивание интерфейса

Это процесс, в котором оценивается удобство использования интерфейса и степень его соответствия требованиям пользователя. Таким образом, оценивание интерфейса является частью общего процесса тестирования и аттестации систем ПОЛ

В идеале оценивание должно проводиться в соответствии с показателями удобства использования интерфейса, перечисленными в табл. 12.5. Каждый из этих показателей можно оценить численно. Например, изучаемость можно оценить следующим образом: опытный оператор после трехчасового обучения должен уметь использовать 80% функциональных возможностей системы. Однако чаще удобство использования интерфейса оценивается качественно, а не через числовые показатели.

Таблица 12.5. Показатели удобства использования интерфейса

Показатель	Описание
Изучаемость	Количество времени обучения, необходимое для начала продуктивной работы с системой
Скорость работы	Скорость реакции системы на действия пользователя
Устойчивость	Устойчивость системы к ошибкам пользователя
Восстанавливаемость	Способность системы восстанавливаться после ошибок пользователя
Адаптируемость	Способность системы "подстраиваться" к разным стилям работы пользователей

Полное оценивание пользовательского интерфейса может оказаться весьма дорогостоящим, в этот процесс будут вовлечены специалисты по когнитивной психологии и дизайнеры. В процесс оценивания могут входить разработка и выполнение ряда статистических экспериментов с пользователями в специально созданных лабораториях и с необходимым для наблюдения оборудованием. Такое оценивание интерфейса экономически нерентабельно для систем, разрабатываемых в небольших организациях с

ограниченными ресурсами.

Существуют более простые и менее дорогостоящие методики оценивания интерфейсов пользователя, позволяющие выявить отдельные дефекты в интерфейсах.

1. Анкеты, в которых пользователь дает оценку интерфейсу.
2. Наблюдения за работой пользователей с последующим обсуждением их способов использования системы при решении конкретных задач.
3. Видеонаблюдения типичного использования системы.
4. Добавление в систему программного кода, который собирал бы информацию о наиболее часто используемых системных сервисах и наиболее распространенных ошибках.

Анкетирование пользователей – относительно дешевый способ оценки интерфейса. Вопросы должны быть точными, а не общими. Не следует использовать вопросы типа "Пожалуйста, прокомментируйте практичность системы", так как ответы, вероятно, будут существенно различаться. Лучше задавать конкретные вопросы, например: "Оцените понятность сообщений об ошибках по шкале от 1 до 5. Оценка 1 означает полностью понятное сообщение, 5 – малопонятное". На такие вопросы легче ответить и более вероятно получить в результате полезную для улучшения интерфейса информацию.

Во время заполнения анкеты пользователи должны обязательно оценить собственный опыт и знания. Такого рода сведения позволят разработчикам зафиксировать, пользователи с каким уровнем знаний имеют проблемы с интерфейсом. Если проект интерфейса уже создан и прошел оценивание в бумажном виде, анкеты можно использовать даже до полной реализации системы.

При наблюдении пользователей за работой оценивается, как они взаимодействуют с системой, какие используют сервисы, какие совершают ошибки и т.п. Вместе с наблюдениями могут проводиться семинары, на

которых пользователи рассказывают о своих попытках решить те или иные проблемы и о том, как они понимают систему и как используют ее для достижения целей.

Видеооборудование относительно недорого, поэтому к непосредственному наблюдению можно добавить видеозапись пользовательских семинаров для последующего анализа. Полный анализ видеоматериалов дорогостоящий и требует специально оснащенного комплекта с несколькими камерами, направленными на пользователя и на экран. Однако видеозапись отдельных действий пользователя может оказаться полезной для обнаружения проблем. Чтобы определить, какие именно действия вызывают проблемы у пользователя, следует прибегнуть к другим методам оценивания.

Анализ видеозаписей позволяет разработчику установить, много ли движений руками вынужден совершать пользователь (в некоторых системах пользователю постоянно приходится переходить с клавиатуры на мышь), и обнаружить неестественные движения глаз. Если при работе с интерфейсом требуется часто смещать зрительный фокус, пользователь может совершить больше ошибок и пропустить какие-либо части изображения.

Вставка в программу кода, собирающего статистические данные при использовании системы, улучшает интерфейс несколькими способами. Обнаруживаются наиболее часто используемые операции. Интерфейс изменяется так, чтобы эти операции выбирались более быстро по сравнению с другими. Например, в вертикальном или выпадающем меню наиболее часто используемые команды должны находиться вверху списка. Такой код также позволит обнаружить и изменить команды, способствующие появлению ошибок.

Наконец, в каждой программе должны быть несложные средства, с помощью которых пользователь сможет передавать разработчикам сообщения с "жалобами". Такие средства убеждают пользователей в том, что с их мнением считаются. А разработчики интерфейса и другие специалисты

могут получить быструю обратную связь относительно отдельных проблем интерфейса.

Ни один из этих далеко не сложных методов оценки пользовательского интерфейса не является надежным и не гарантирует решения всех проблем интерфейса. Вместе с тем перед выпуском системы эти методы можно применить в группе добровольцев, не затрачивая значительных средств. При этом обнаруживается и исправляется большинство проблем в интерфейсе пользователя.

Контрольные вопросы:

1. Каким рядом преимуществ обладают графические интерфейсы?
2. Укажите принципы проектирования интерфейсов пользователя
3. Перечислите пять основных стилей взаимодействия
4. Перечислите наиболее важные правила эффективного использования цвета в пользовательских интерфейсах.
5. что такое справочные системы?

Ключевые слова: *поддержка пользователя, непосредственное манипулирование, выбор из меню, заполнение форм, командный язык, естественный язык, справочные системы, переадресация, функциональное описание, документ по инсталляции системы, вводное руководство, справочное руководство, руководство администратора.*

Keywords: *user support, direct manipulation, menu selection, filling out forms, command language, natural language, help systems, call forwarding, functional description, document the installation of the system, an introductory guide, reference guide, the Administrator's Guide.*

Kalit so'zlar: *foydalanuvchiga yordam berish, bevosita manipulyatsiya, menyudan tanlov, formalarni to'ldirish, buyruqlar tili, tabiiy til, ma'lumotlar tizimi, pereadresatsiya, funksional ta'rif, tizim instalyatsiyasi bo'yicha hujjat, kiritish qo'llanmasi, ma'lumotnoma, administrator qo'llanmasi.*

Упражнения

1. В разделе 15.1 отмечалось, что объекты, которыми манипулирует пользователь, должны отображать его понятия предметной области приложения ПО (а не компьютерной предметной области). Предложите подходящие объекты манипулирования для следующих типов пользователей и систем.

• Автоматизированный каталог товаров для ассистента на складе.

• Система наблюдения за безопасностью самолета для летчика гражданской авиации.

• Финансовая база данных для менеджера.

• Система управления патрульными машинами для полицейского.

2. Опишите ситуации, в которых неразумно или невозможно поддерживать интерфейс пользователя.

3. Какие факторы следует учитывать при проектировании интерфейсов, использующих меню, для таких систем, как банкоматы? Опишите основные черты интерфейса банкомата, которым вы пользуетесь.

4. Предложите способы адаптации пользовательского интерфейса в системах электронной коммерции (например, виртуального книжного магазина или магазина музыкальных дисков) для пользователей, имеющих физические недостатки, например плохое зрение или проблемы опорно-двигательной системы.

5. Обсудите преимущества графического способа отображения информации и приведите четыре примера приложений, в которых более уместно использовать графическое представление числовых данных, а не табличное.

6. Какими основными принципами следует руководствоваться при использовании цветов в интерфейсах пользователя? Предложите более эффективный способ использования цветов в интерфейсе любого известного вам приложения.

7. Рассмотрите сообщения об ошибках, генерируемые операционными системами MS Windows, Unix, MacOS или какой-либо другой. Как их можно улучшить?

8. Составьте анкету по сбору данных об интерфейсе какой-либо известной вам программы (например, текстового редактора). Если есть возможность, распространите эту анкету среди других пользователей и попытайтесь оценить результаты анкетирования. Что вы узнали об интерфейсе программы из анкет?

9. Обсудите, этично ли разрабатывать программные системы, не согласовав с конечными пользователями те элементы системы, которые они будут контролировать.

10. С какими этическими проблемами сталкиваются разработчики интерфейсов, когда пытаются согласовать запросы конечных пользователей системы с требованиями организации, которая оплачивает разработку данной системы?

ГЛАВА 13. НАДЕЖНОСТЬ СИСТЕМ

Известно, что компьютерные системы имеют недостатки, т.е. без явных причин иногда выходят из строя, и не всегда ясно, что требуется для восстановления их работоспособности. Программы, выполняемые на таких компьютерах, работают неверно, порой искажая данные. Необходимо учиться жить с этими недостатками, не теряя веры в то, что существуют персональные компьютеры, которые обычно работают нормально.

Функциональную надежность компьютерных систем можно определить степенью доверия к ним, т.е. уверенностью, что система будет работать так, как предполагается, и что сбоев не будет. Это свойство нельзя оценить количественно. Для этого используются такие относительные термины, как "ненадежные", "очень надежные" или "сверхнадежные", отражающие различную степень доверия к системе.

Надежность и полезность – это, конечно, разные вещи. Программа текстового редактора, которую я использовал при написании книги, является не очень надежной, но весьма полезной системой. Зная это, я часто сохранял работу, многократно ее копируя. Этими действиями я компенсировал недостатки системы, снижая риск потери информации в случае ее отказа.

Существует четыре основные составляющие функциональной надежности программных систем (рис. 13.1), неформальные определения которых приведены ниже.

1. **Работоспособность** – свойство системы выполнять свои функции в любое время эксплуатации.

2. **Безотказность** – свойство системы корректно (так, как ожидает пользователь) работать весь заданный период эксплуатации.

3. **Безопасность** – свойство системы, гарантирующее, что она безопасна для людей и окружающей среды.

4. **Защищенность** – свойство системы противостоять случайным или намеренным вторжениям в нее.

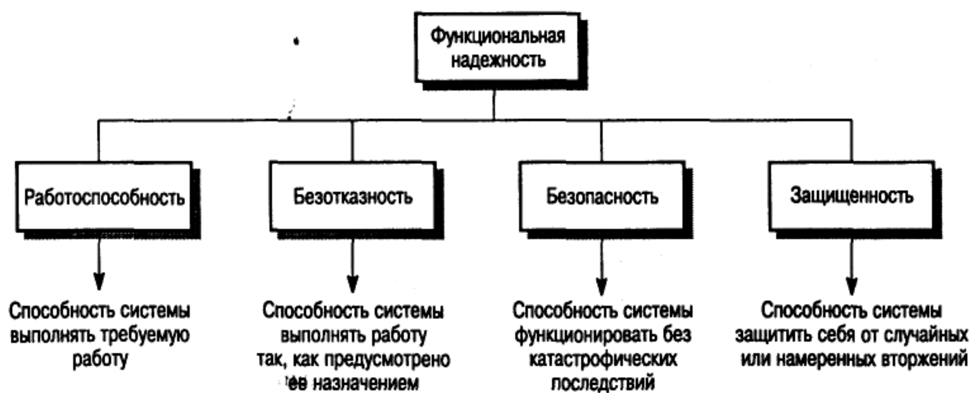

Рис. 13.1. Составляющие надежности системы

Работоспособность и безотказность систем обсуждаются в главе 17. Эти показатели носят вероятностный характер и могут быть выражены количественно. Безопасность и защищенность редко выражаются в виде числовых показателей, но их можно сравнивать по относительной шкале уровней. Например, безопасность уровня 1 меньше безопасности уровня 2, которая, в свою очередь, меньше безопасности уровня 3, и т.д.

Дополнительные меры, повышающие функциональную надежность системы, могут резко увеличивать стоимость ее разработки. На рис. 13.2 показана зависимость между стоимостью разработки и различными уровнями функциональной надежности. Здесь подразумевается, что функциональная надежность содержит все составляющие: работоспособность, безотказность, безопасность и защищенность. Экспоненциальный характер зависимости "стоимость-надежность" не позволяет говорить о возможности создания систем со стопроцентной надежностью, так как стоимость их создания была бы очень большой.

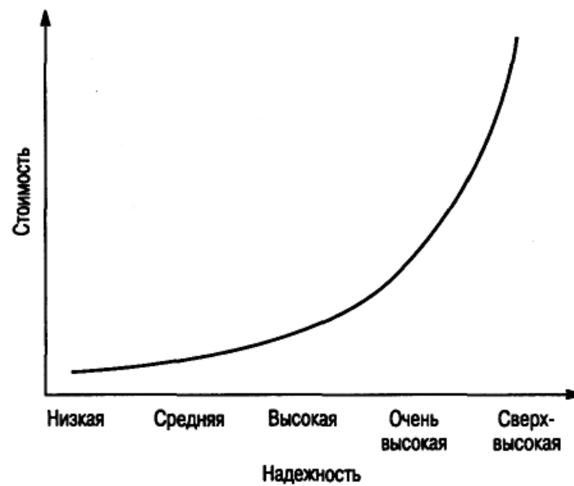

Рис. 13.2. Зависимость между стоимостью разработки системы и ее надежностью

Высокие уровни функциональной надежности могут быть достигнуты только за счет уменьшения эффективности работы системы. Например, надежное программное обеспечение предусматривает дополнительные, часто избыточные, коды для проверки нештатных состояний системы. Это усложняет систему и увеличивает объем памяти, необходимый для ее эффективной работы. Но в ряде случаев надежность более важна, чем эффективность системы.

1. **Ненадежные системы часто остаются невостребованными.** Если к системе нет доверия пользователя, она не будет востребована. Более того, пользователи могут отказаться от других программных продуктов той же компании-разработчика, поскольку будут также считать их ненадежными.

2. **Стоимость отказа системы может быть огромна**. Для некоторых приложений, таких, как системы управления реакторами или системы навигации, стоимость последствий отказа может превышать стоимость самой системы.

3. **Трудно модернизировать ненадежную систему для повышения ее надежности.** Обычно есть возможность улучшить неэффективную систему, так как в этом случае основные усилия будут затрачены на модернизацию отдельных программных модулей. Систему, к

которой нет доверия, трудно улучшить, поскольку ненадежность "распределена" по всей системе.

4. **Существуют возможности компенсировать недостаточную эффективность системы**. Если программная система работает неэффективно, то это постоянный фактор, к которому пользователь может приспособиться, построив свою работу с его учетом. Ненадежность системы, как правило, проявляется внезапно. Ненадежное программное обеспечение может нарушить работу всей системы, в которую оно интегрировано, и разрушить данные пользователя без предупреждения, что может иметь серьезные последствия.

5. **Ненадежные системы могут быть причиной потери информации.** Сбор и хранение данных – дорогостоящая процедура, часто данные стоят больше, чем компьютерная система, на которой они обрабатываются. Дублирование данных для предотвращения их потери вследствие ненадежности системы потребует значительных усилий и финансовых средств.

Надежность системы зависит от технологии разработки ПО. Многократные тестирования с целью исключения ошибок способствуют разработке надежных систем. Однако нет простой связи между качеством процесса создания и качеством готовой системы. Вопросы взаимосвязи между процессом создания ПО и качеством готовой продукции рассматриваются в главах 24 и 25, где представлены темы управления качеством и модернизации систем.

13.1. Критические системы

Обычно отказ систем, управляемых с помощью ПО, вызывает неудобства, но они не приводят к длительным последствиям. Однако имеются системы, отказы которых могут приводить к значительным экономическим потерям, физическим повреждениям или создавать угрозу

человеческой жизни. Такие системы обычно называют *критическими.* **Функциональная надежность** – необходимое требование к критическим системам, и все ее составляющие (работоспособность, безотказность, безопасность и защищенность) очень важны. Не менее важен для критических систем и высокий уровень надежности.

Существует три основных типа критических систем.

1. **Системы, критические по обеспечению безопасности**. Системы, отказ которых приводит к разрушениям, создает угрозу жизни человека или наносит вред окружающей среде. В качестве примера можно привести систему управления производством на химическом заводе.

2. **Системы, критические для целевого назначения**. Системы, отказ которых может привести к ошибкам в действиях, направленных на обеспечение определенной цели. Примером может служить навигационная система космического корабля.

3. **Системы, критические для бизнеса**. Отказ таких систем может нанести вред делу, в котором они используется. Примером является система, обслуживающая счета клиентов в банке.

Цена ошибки критической системы часто очень велика. Она включает прямые расходы, связанные с внесением изменений в систему или ее заменой, косвенные расходы, например судебные, и расходы, связанные с потерями в бизнесе. Из высокой возможной цены отказа системы следует, что качество методов разработки и сам процесс создания ПО обычно более важны, чем стоимость применения этих методов.

Поэтому при создании критических систем обычно используются испытанные методы разработки, а не новые, еще не имевшие большого практического применения. Только сравнительно недавно такие относительно новые методы, как, например, объектно-ориентированные, стали использоваться для разработки критических систем, вместе с тем до сих пор при разработке многих критических систем все еще применяются функционально-ориентированные методы.

С другой стороны, методы разработки ПО, которые обычно нерентабельны, могут использоваться для разработки критических систем, например метод формальных спецификаций и формализованной проверки программ на соответствие таким спецификациям. Одной из причин использования этих методов является уменьшение количества требующегося тестирования. Для критических систем стоимость проверки и аттестации обычно очень высока и может составлять более 50% общей стоимости системы.

Хотя эта книга посвящена разработке программных систем, а не общей теории систем, необходимо отметить, что функциональная надежность – общесистемное понятие. Рассматривая надежность критических систем, можно выделить три типа системных "компонентов", склонных к отказу.

1. Аппаратные средства системы, отказывающие либо из-за ошибок конструирования, либо из-за ошибок изготовления, либо из-за полного износа.

2. Программное обеспечение системы, которое может отказывать из-за ошибок либо в технических требованиях к системе, либо в архитектуре системы, либо в программном коде.

3. Человеческий фактор, который своими действиями нарушает правильную работу системы.

Таким образом, если цель состоит в том, чтобы повысить надежность системы, необходимо рассматривать все эти аспекты во взаимосвязи. Я поясняю это положение на нескольких примерах в других главах этой части книги.

13.2. Системы, критические по обеспечению безопасности

В этом разделе в качестве примера системы, критической по обеспечению безопасности, рассмотрим систему, управляющую дозировкой инъекций инсулина при заболевании диабетом. Предполагается, что большинство читателей имеют общее представление об этом заболевании и

его лечении. Эта система описана в главе 9, где была представлена ее формальная спецификация.

Диабет – заболевание, при котором человеческий организм не может выработать достаточное количество гормона, называемого инсулином и регулирующего содержание сахара в крови. Если уровень инсулина в организме будет избыточным, содержание сахара в крови может понизиться, что влечет за собой очень серьезные последствия – прекращение питания мозга, приводящее к потере сознания и даже летальному исходу. Если же организм вырабатывает инсулин в недостаточном количестве, то уровень сахара повышается, что приводит к нарушению зрения, заболеванию почек и всего организма.

Благодаря появлению миниатюрных датчиков стало возможным создание автоматизированной системы инъекций инсулина. Она контролирует уровень сахара в крови, и, если необходимо, в организм вводится соответствующая доза инсулина. Конечно, такие системы могут пока работать только в стационарных условиях. В дальнейшем, будучи подключенными к человеку, они станут доступны широкому кругу больных диабетом.

Для работы такой системы микродатчик вживляется в тело больного. Этот датчик контролирует определенный параметр крови, который характеризует уровень сахара. Результат посылается в контрольный блок, который вычисляет уровень сахара, определяет необходимую дозу инсулина и посылает сигнал для инъекций постоянно подсоединенной игле. Совершенно очевидно, что такой системой должна управлять надежная программа. На рис. 13.3 показаны компоненты и организация системы дозировки инъекций инсулина. На рис. 13.4 показана модель потока данных, где видно, как входное значение уровня сахара в крови преобразуется в последовательность команд управления дозировкой инъекций инсулина.

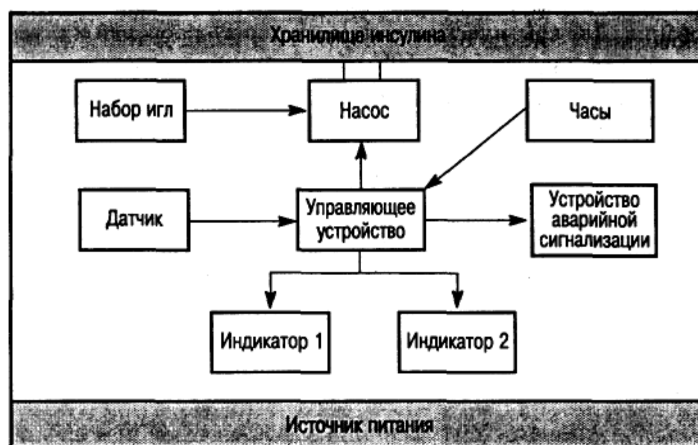

Рис. 13.3. Структура системы дозировки инъекций инсулина

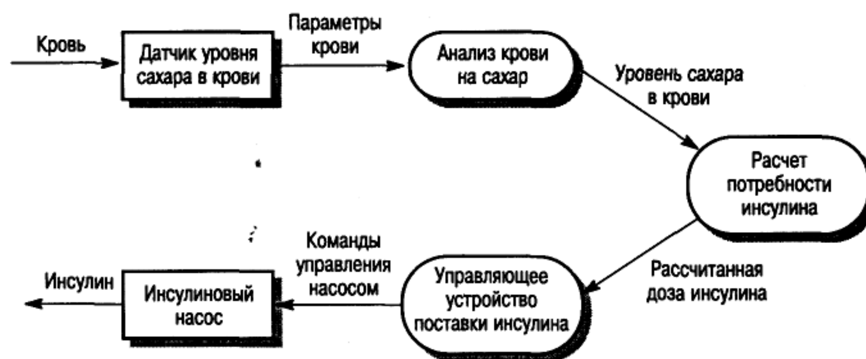

Рис. 13.4. Модель потока данных системы дозировки инъекций инсулина

Надежность системы дозировки инъекций инсулина характеризуется тремя составляющими.

1. **Работоспособность**. При возникновении критической ситуации система должна быть готовой ввести необходимую дозу инсулина.

2. **Безотказность.** В зависимости от уровня сахара в крови система должна правильно определить дозу инсулина и ввести ее пациенту.

3. **Безопасность.** Отказ такой системы может привести к чрезмерной дозе инсулина, что представляет опасность для жизни больного. Необходимо добиться, чтобы такого рода отказов в системе не было.

В последующих главах разъясняется, как эти составляющие можно точно определить и проверить.

13.3. Работоспособность и безотказность

В этом разделе обсуждаются две близко связанные составляющие функциональной надежности – **работоспособность** и **безотказность**. Под работоспособностью системы подразумевается способность предоставлять пользователю все необходимые системные сервисы по мере возникновения потребности в них. Безотказность – это способность системы предоставлять именно те сервисы, которые заложены в систему ее спецификацией. Из этого следует, что безотказность более общий показатель, включающий в себя свойство работоспособности, поскольку если система не работоспособна, то о безотказности вообще речи идти не может.

Однако необходимо и различать эти свойства, так как требования к работоспособности и безотказности в разных системах могут быть различны. Например, некоторые системы могут иметь сравнительно частые сбои, но они также могут быстро восстанавливаться после сбоев. У таких систем сравнительно низкие требования к безотказности. В то же время они могут иметь высокие требования к работоспособности в связи с необходимостью непрерывного обслуживания пользователей.

Наглядный пример такой системы – коммутатор телефонной станции. Снимая трубку телефона, пользователь слышит зуммер, означающий, что "линия свободна" и система готова выполнить требования пользователя. В случае сбоя такая система должна быть легко восстанавливаема. Коммутатор телефонной станции имеет средства для восстановления неправильной работы и разрешает повторную попытку соединения. Это выполняется очень быстро, и абонент телефона может даже не почувствовать, что был сбой. В такой ситуации основным требованием к функциональной надежности системы является безотказность.

Другое различие между этими показателями заключается в том, что работоспособность системы зависит также от времени, которое требуется для устранения неисправности. Так как, если система А отказывает один раз в год, а система В отказывает раз в месяц, то система А более надежна, чем В.

Однако, если системе A необходимо три дня для восстановления работоспособности, а системе B для этого достаточно всего 10 минут, то безотказность системы B выше, чем системы A. Исходя из этих соображений пользователь скорее выберет систему B, чем A

Безотказность и работоспособность системы могут быть определены более точно следующим образом.

1. **Безотказность** – это способность системы безотказно работать определенное время с указанной целью в определенном окружении.

2. **Работоспособность** – это способность системы правильно функционировать и предоставлять вовремя требуемые сервисы.

Одна из практических проблем, возникающая при разработке систем, заключается в том, что наши интуитивные понятия безотказности и работоспособности часто оказываются шире приведенных выше формулировок. При рассмотрении функциональной надежности системы должны быть приняты во внимание окружение, в котором будет действовать система, и цель, для которой она используется. Следовательно, оценка безотказности в одном окружении не обязательно переносится в другое окружение, где система используется по-другому.

Для примера рассмотрим безотказность системы программного обеспечения в двух средах (окружении) – в офисе и в университете. В офисе пользователи строго следуют инструкциям по работе с системой и не имеют времени и возможностей экспериментировать с ней. В университетской среде студенты пробуют все возможности системы и часто непредвиденными способами, что может привести к отказам системы, которые никогда бы не встретились в офисной среде.

Также важны способы использования системы и реакция человека на ее работу. Представим себе, что у автомобиля неисправна стеклоочистительная система – происходит сбой в работе стеклоочистителей. В дождливую погоду работа этой системы очень важна. Безотказность такой системы будет определяться местностью, где происходит действие, и реакцией водителя.

Для водителя из Сиэтла (влажный климат) этот отказ будет более чувствителен, чем для водителя из Лас-Вегаса (сухой климат). Водитель из Сиэтла будет считать, что система не является безотказной, в то время как водитель из Лас-Вегаса вообще никогда не столкнется с этой проблемой.

В определении работоспособности и безотказности системы не рассматривается тяжесть и последствия отказов системы. Людям особенно небезразличны отказы, которые имеют серьезные последствия; от этого зависит восприятие безотказности системы. Например, работа автомобильного двигателя, который дает сбои сразу после запуска, а затем после перезапуска работает исправно, раздражает. Но это не затрагивает нормального режима эксплуатации автомобиля.

Считается, что система ведет себя надежно, если ее работа соответствует заданному алгоритму. Однако возможны ситуации, когда работа системы не совсем соответствует ожиданиям пользователей. К сожалению, системные требования часто бывают или неполны или некорректны. Разработчики в таких случаях должны сами решать, как будет вести себя система, но, не всегда являясь специалистами в конкретной области применения системы, они не могут учесть все факторы и запрограммировать такое поведение системы, которое необходимо пользователю.

Как показывает опыт, наиболее важными составляющими функциональной надежности являются безотказность и работоспособность. Если же система ненадежна, то трудно гарантировать ее безопасность или защищенность. Ненадежность системы влечет за собой большие материальные потери, такие системы приобретают репутацию некачественных и в дальнейшем теряют доверие потребителей.

Безотказность системы определяется отсутствием сбоев. Отказы системы могут происходить из-за плохого или неправильного ее обслуживания, могут быть следствием ошибок в алгоритме, а могут быть вызваны неисправностями систем связи. Однако во многих случаях

причиной ошибочного поведения системы являются дефекты в самой системе. При рассмотрении безотказности полезно понимать различие в терминах *сбой, ошибка* и *отказ,* которые определены в табл. 13.1.

Таблица 13.1. Терминология безотказности

Термин	Описание
Отказ системы	Прекращение функционирования системы
Системные ошибки	Ошибочное поведение системы, не соответствующее ее спецификации
Сбой системы	Неправильное поведение системы, непредвиденное ее разработчиками
Ошибка оператора	Неверные действия пользователя, вызвавшие сбой в работе системы

Сбои не обязательно приводят к отказам системы, поскольку они могут быть кратковременными и система может прийти к нормальному функционированию раньше, чем произойдет отказ. Системные ошибки также не обязательно приводят к отказам системы, так как системы имеют защиту, гарантирующую, что ошибочный режим будет обнаружен и исправлен.

Терминология, приведенная в табл. 13.1, помогает понять три дополняющих друг друга подхода, используемых для повышения безотказности систем.

1. **Предотвращение сбоев**. Подход к разработке ПО, минимизирующий возможность появления ошибок и/или обнаруживающий ошибки прежде, чем они приведут к сбоям системы. Пример такого подхода – исключение подверженных ошибкам определенных конструкций языков программирования (например, указателей) и постоянный анализ программ для обнаружения различных аномалий программного кода (см. главу 19).

2. **Обнаружение ошибок и их устранение**. Использование разнообразных методов проверки системы в различных режимах позволяет обнаружить ошибки и устранить их до ввода системы в эксплуатацию. Регулярное тестирование системы и ее отладка – пример данного подхода.

3. **Устойчивость к сбоям.** Использование специальных методов, гарантирующих, что ошибки в системе не приведут к сбоям и что сбои не приведут к отказам системы. Пример такого подхода – применение средств самовосстановления системы с использованием дублирования модулей.

Разработка систем, устойчивых к сбоям, описана в главе 18, где также кратко обсуждаются некоторые методы предотвращения сбоев. Методы разработки ПО, предотвращающие сбои, приведены в главе 16. Вопросы обнаружения системных ошибок обсуждаются в главах 19 и 20.

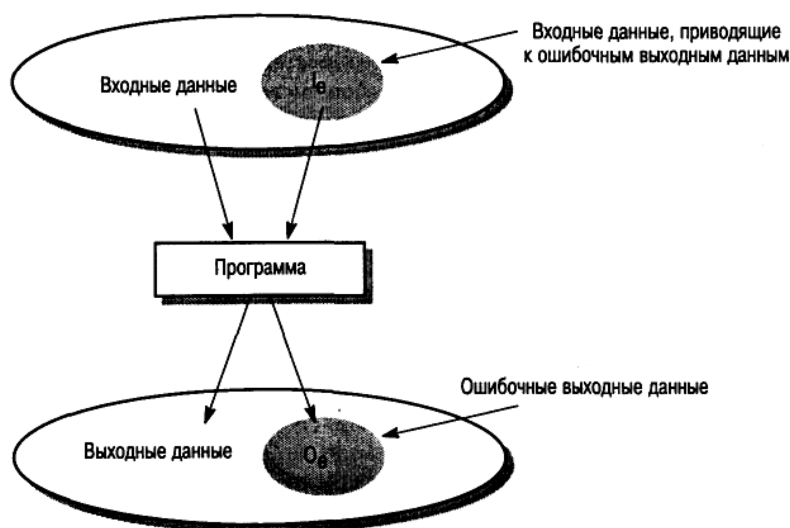

Рис. 13.5. Программная система как отображение входных данных на множество выходных данных

Программный сбой приводит к отказу системы, когда "сбойный" программный код выполняется над определенными входными данными, приводящими к сбою системы. При этом программа может корректно работать на других входных данных. На рис. 13.5, заимствованным из работы [221], программная система представлена как отображение множества входных данных (входов) на множество выходных данных (выходов). Программа обрабатывает входные данные, получая выходные данные.

Некоторые из входных данных (см. рис. 13.5) приводят к отказам системы, в результате программа генерирует ошибочные выходные данные. Безотказность программного обеспечения характеризуется вероятностью, с которой при выполнении программы среди множества входных данных встретятся такие, что приведут к ошибочным результатам вычислений.

Существует сложная взаимосвязь между наблюдаемой безотказностью системы и количеством скрытых программных дефектов. В работе [239] подчеркивается, что не все программные ошибки в равной степени вызывают отказ системы. Обычно в множестве входных данных $I_{ê}$, приводящих к ошибочным выходным данным, имеется ряд данных, вероятность выбора которых больше, чем у других данных. Если эти входные данные не требуют для своей обработки той части ПО, которая содержит ошибки, то системных сбоев не будет. Таким образом, безотказность системы зависит преимущественно от количества входных данных, приводящих к ошибочным результатам во время нормальной эксплуатации системы. Сбои системы, которые проявляются только в исключительных ситуациях, мало влияют на ее надежность.

Надежность системы связана с вероятностью ошибки, проявляющейся во время эксплуатации системы. Устранение программных ошибок в редко используемых системных модулях мало повлияет на повышение безотказности системы. В работе [239] показано, что устранение 60% программных ошибок только на 3% повысит безотказность системы. Это подтверждается и исследованиями ошибок в программных продуктах IBM. В работе [4] показано, что многие ошибки в программных продуктах реально вызывают сбои системы после сотен или тысяч месяцев эксплуатации.

Следовательно, программа может содержать ошибки и все же вызывать доверие пользователей. Сбои программы "никогда не будут возникать, если не выбирать входных данных, ведущих к сбоям. Кроме того, опытные пользователи часто работают, зная об ошибках программного обеспечения,

вызывающих сбои системы, и умело избегают их. Устранение ошибок в таких случаях не даст практически никакого повышения надежности.

Рис. 13.6. Использование множества входных данных разными пользователями

Каждый пользователь системы по-своему избегает "встречи" с системными ошибками. Ошибки, с которыми встречается один пользователь, могут никогда не встретиться другому. На рис. 13.6 показано множество входных данных, используемых разными пользователями. Входные данные, выбранные пользователем 2 пересекаются с входными данными, приводящими к сбою системы. Поэтому пользователь 2 будет сталкиваться со сбоями системы. Пользователи 1 и 3 никогда не используют входных данных, приводящих к сбою системы, – для них программное обеспечение всегда будет безотказно.

13.4. Безопасность

Безопасность системы – это свойство, отражающее способность системы функционировать, не угрожая людям или окружающей среде. Там, где безопасность является необходимым атрибутом системы, говорят о системе, критической по обеспечению безопасности. Примерами могут служить контролирующие и управляющие системы в авиации, системы управления процессами на химических и фармацевтических заводах и системы управления автомобилями.

Управление систем, критических по обеспечению безопасности, гораздо проще организовать с помощью аппаратных средств, чем ПО. Однако сейчас строятся системы такой сложности, что управление не может осуществляться только аппаратными средствами. Из-за необходимости управлять большим числом сенсоров и исполнительных механизмов со сложными законами управления требуется управляющее программное обеспечение. В качестве примера можно привести системы управление военными самолетами. Они требуют постоянного программного управления самолетом, гарантирующего безопасность полета.

Программное обеспечение рассматриваемых в этом разделе систем подразделяется на два класса.

1. **Первичное программное обеспечение, критическое по критерию безопасности**. Это ПО включается в систему в виде отдельного блок управления. Неправильная работа такого ПО может быть причиной отказа оборудования, вследствие которого может возникнуть угроза жизни человека или нанесения вреда окружающей среде. Я особо обращаю внимание на этот класс программного обеспечения.

2. **Вторичное программное обеспечение, критическое по критерию безопасности.** Это ПО косвенным образом может привести к непредвиденным последствиям. Примерами могут служить автоматизированные системы в технике, неправильная работа которых может привести к ошибкам в работе объекта и поставить под угрозу жизнь людей. Другой пример такого ПО – медицинская база данных, содержащая описание лекарств, предназначенных пациентам. Ошибки в этой системе могут привести к неправильной дозировке препаратов.

Безотказность и безопасность системы – взаимосвязанные, но, очевидно, различные составляющие функциональной надежности. Конечно, система, критическая по обеспечению безопасности, должна соответствовать своему назначению и функционировать без отказов. Для обеспечения непрерывного функционирования даже в случае ошибок она должна иметь

защиту от сбоев. Однако отказоустойчивость не гарантирует безопасности системы. Программное обеспечение может только один раз сработать неправильно, и это приведет к несчастным случаям.

Нельзя быть на сто процентов уверенным, что системное программное обеспечение безопасно и отказоустойчиво. Безотказность ПО не гарантирует его безопасность по ряду причин.

1. В системной спецификации может быть не определено поведение системы в некоторых критических ситуациях. Высокий процент сбоев систем – результат скорее неверных или неполных требований, чем ошибок программирования [49, 109, 226, 252, 8*, 20*, 29*]. После изучения ошибок в программных системах Лутц (Lutz, [226]) делает вывод:

"...трудности с формулировкой требований к разрабатываемой системе – основная причина ошибок программного обеспечения, связанных с безопасностью, которые нельзя выявить до сборки и тестирования системы".

2. Сбои в работе аппаратных средств могут привести к непредсказуемому поведению системы, в результате чего программное обеспечение сталкивается с непредвиденной ситуацией. Когда системные компоненты близки к состоянию отказа, они могут вести себя неустойчиво и генерировать сигналы, которые могут быть обработаны программным обеспечением непредвиденным образом.

3. Операторы, работающие с системой, могут внести ошибки, которые в особых ситуациях способны привести к сбою системы. Анекдотический пример – механик дал команду системе, управляющей полетом самолета, поднять шасси. Управляющая система выполнила команду механика несмотря на то, что самолет был на земле!

При разработке систем, критических по обеспечению безопасности, используется специальная терминология, приведенная в табл. 13.2. Эти термины взяты из работы [214] и уточнены мною.

Таблица 13.2. Терминология безопасности

Термин	**Описание**
Авария (или несчастный случай)	Незапланированное событие или последовательность событий, приводящие к человеческой смерти или ранению; нанесение ущерба собственности или окружающей среде. Пример несчастного случая – нанесение увечий оператору машиной, управляемой компьютерной системой
Опасность (опасные ситуации)	Ситуации, при которых возможны несчастные случаи и аварии. Пример опасности – отказ сенсора, который определяет наличие препятствия впереди машины
Повреждения	Оцениваются как ущерб от опасных случаев. Повреждения могут быть как незначительными, так и катастрофическими, приводящими к гибели людей
Серьезность опасности	Оценивается по самым большим повреждениям в результате самых опасных случаев. Серьезность опасности может ранжироваться от катастрофической, приводящей к гибели людей, до незначительной
Вероятность опасности	Вероятность появления событий, которые создают опасные ситуации. Значение вероятности определяется обычным образом. Опасные события ранжируются от вероятного (если, например, вероятность равна 1/100, т.е. на 100 "нормальных" событий произойдет одно опасное) и до невозможного (когда ни при каких условиях не возникнет опасной ситуации)
Риск	Измеряется как вероятность того, что система будет причиной несчастного случая. Для оценки риска определяются вероятность опасности, серьезность опасности и вероятность того, что опасная ситуация приведет к аварии

Считается, что система безопасна, если ее эксплуатация исключает аварии (несчастные случаи) или их последствия незначительны. Этого можно достичь тремя дополняющими друг друга способами.

1. **Предотвращение опасности.** Система разрабатывается таким образом, чтобы избежать опасных ситуаций. Например, чтобы во время эксплуатации машины избежать попадания рук оператора под лезвие, в системе раскроя предусматривается обязательное одновременное нажатие двух отдельных кнопок управления.

2. **Обнаружение и устранение опасности**. Система разрабатывается таким образом, чтобы возможные опасные ситуации были обнаружены и устранены до того, как они приведут к аварии. Например, система, управляющая химическим предприятием, для предотвращения взрыва от высокого давления должна вовремя обнаружить избыточное давление и открыть предохранительный клапан, чтобы уменьшить это давление.

3. **Ограничение последствий**. Система может включать способы защиты, минимизирующие повреждения, возникающие в результате происшедшей аварии. Например, в систему управления двигателями самолета обычно включается автоматическая система огнетушения. В случае возгорания такая система позволяет предотвратить пожар и не ставит под угрозу жизнь пассажиров и экипажа.

Аварии и несчастные случаи обычно являются результатом нескольких событий, которые происходят одновременно с непредвиденными последствиями. Анализируя серьезные аварии, в работе [275] показано, что почти все они произошли из-за комбинации системных сбоев, а не вследствие отдельных сбоев. Непредвиденная комбинация сбоев приводила к отказу системы. В той же работе утверждается, что невозможно предупредить все комбинации сбоев системы и эти аварии – неизбежное следствие использования сложных систем. Программное обеспечение имеет тенденцию разрастаться и усложняться, а сложность программно-управляемых систем увеличивает вероятность аварий и несчастных случаев.

Это, конечно, не означает, что программное управление обязательно увеличивает риск, связанный с системой. Программное управление и текущий контроль могут повысить безопасность систем. Кроме того, программно-управляемые системы могут контролировать более широкий диапазон условий по сравнению, например, с электромеханическими системами. Они также довольно легко настраиваются. Они предполагают использование компьютерных средств, которым свойственна высокая

надежность и которые относительно компактны. Программно-управляемые сложные системы могут блокировать опасность. Они могут поддерживать управление во вредных условиях, уменьшая количество необходимого обслуживающего персонала.

13.5. Защищенность

Это способность системы защищать себя от внешних случайных или преднамеренных воздействий. Примером внешних воздействий на систему могли бы быть компьютерные вирусы, несанкционированное использование системы, несанкционированное изменение системы или данных и т.д. Защищенность важна для всех критических систем. Без приемлемого уровня защищенности работоспособность, безотказность и безопасность системы теряют смысл, поскольку причиной повреждения системы могут быть внешние воздействия.

Это связано с тем, что все методы подтверждения работоспособности, безотказности и защищенности полагаются на неизменность системы при эксплуатации и на соответствие ее параметров первоначально установленным. Если установленная система была повреждена каким-то образом (например, если программное обеспечение было изменено в результате проникновения в систему вируса), то параметры надежности и безопасности, которые первоначально были заложены, не могут больше поддерживаться. В этом случае программное обеспечение может вести себя непредсказуемо.

Имеются определенные типы критических систем, для которых защищенность – наиболее важный показатель надежности системы. Военные системы, системы для электронной торговли и системы, включающие создание и обмен конфиденциальной информацией, должны разрабатываться с очень высоким уровнем защищенности. Например, если система резервирования билетов авиакомпании недоступна, это причиняет

неудобство в связи с некоторой задержкой продажи билетов, но если система не защищена и может принимать поддельные заказы, то авиакомпания может понести большие убытки.

Существует три типа повреждений системы, которые могут быть вызваны внешними воздействиями.

1. **Отказ в предоставлении системных сервисов**. Система может быть переведена в такое состояние, когда нормальный доступ к системным сервисам становится невозможным. Очевидно, это отражается на работоспособности системы.

2. **Разрушение программ и данных**. Компоненты программного обеспечения системы могут быть несанкционированно изменены. Это может повлиять на поведение системы, а следовательно, на надежность и безопасность. Если повреждение серьезно, система может стать не пригодной к эксплуатации.

3. **Раскрытие конфиденциальной информации.** Информация, находящаяся под управлением системы, может быть конфиденциальной, внешнее проникновение в систему может сделать ее публично доступной. В зависимости от типа данных, это может повлиять на безопасность системы и вызвать дальнейшие изменения в системе, которые скажутся на ее работоспособности и безотказности.

Как и в случае с другими составляющими надежности, есть специальная терминология, связанная с защищенностью систем. Некоторые термины, определенные в монографии [279], приведены в табл. 13.3.

Таблица 13.3. Терминология защищенности

Термин	Описание
Внешнее воздействие	Воздействие, в результате которого возможна потеря данных и/или повреждение системы
Уязвимость	Дефект системы, который может стать причиной потери данных или ее повреждения
Атака	Использование уязвимости системы

Угрозы	Обстоятельства, которые могут привести к потере данных или повреждению системы
Контроль	Защитные меры, уменьшающие уязвимость системы

В терминологии защищенности есть много общего с терминологией безопасности. Так, внешнее воздействие аналогично аварии (несчастному случаю), а уязвимость – опасности. Следовательно, существуют аналогичные подходы, увеличивающие защищенность системы.

1 **Предотвращение уязвимости**. Система разрабатывается таким образом, чтобы ее уязвимость была как можно ниже. Например, система не соединяется с внешней сетью, чтобы избежать воздействия из нее.

2. **Обнаружение и устранение атак**. Система разрабатывается таким образом, чтобы обнаружить предпринятую на нее атаку и устранить ее, пока она не привела к повреждениям и потерям. Пример обнаружения атак и их устранения - использование антивирусных программ, которые анализируют поступающую информацию на наличие вирусов и устраняют их в случае проникновения в систему.

3. **Ограничение последствий**. Система разрабатывается таким образом, чтобы свести к минимуму последствия внешнего воздействия. Например, регулярная проверка системы и возможность переустановить ее в случае повреждения.

Защищенность становится еще более актуальной при подключении системы к Internet. И хотя Internet-связь обеспечивает дополнительные функциональные возможности системы (например, клиент может получить удаленный доступ к своему банковскому счету), такая система может быть разрушена злоумышленниками. При подключенности к Internet уязвимые места системы становятся доступными для большого количества людей, которые могут воздействовать на систему.

Очень важным атрибутом систем, подключенных к Internet, является их жизнеспособность [107] т е. способность системы продолжать работать, в то время как она подвергается внешним воздействиям и часть ее повреждена. Жизнеспособность, конечно, связана и с защищенностью и с работоспособностью. Для обеспечения жизнеспособности следует определить основные ключевые компоненты системы, которые необходимы для ее функционирования [108]. Для повышения жизнеспособности используется три стратегии: противодействие внешним воздействиям, распознавание их и восстановление системы после атаки. Здесь нет возможности охватить эту тему, но на Web-странице данной книги имеются ссылки на источники с информацией относительно исследования жизнеспособности систем.

Контрольные вопросы:

1) Какие четыре основные составляющие функциональной надежности программных систем существуют?
2) Укажите три основных типа критических систем.
3) Что такое безотказность?
4) Что такое работоспособность?
5) Что такое безопасность системы?

Ключевые слова: работоспособность, безотказность, безопасность, защищенность, функциональная надежность, предотвращение сбоев, обнаружение ошибок и их устранение, устойчивость к сбоям, безопасность системы, предотвращение опасности, обнаружение и устранение опасности, ограничение последствий, отказ в предоставлении системных сервисов, разрушение программ и данных, раскрытие конфиденциальной информации, предотвращение уязвимости, обнаружение и устранение атак, ограничение последствий.

Keywords: *capacity to work, unrefusality, safety, protect, functional reliability, prevention of the malfunctions, finding error and their removal, resistance to malfunction, safety of the system, prevention to dangers, finding and removal to dangers, restriction consequence, refusal of granting system service, destruction of the programs and data, opening to proprietary information, prevention to criticality, finding and eliminating the attacks, restriction consequence.*

Kalit so’zlar: *ishga layoqat, to’htovsizlik, havfsizlik, himoyalanganlik, funksional ishonchlilik, to’htab qolishlarni oldini olish, hatolarni topish va to’g’irlash, tizim havfsizligi, havfni oldini*

olish, havfni izlash va oldini olish, oqibatlarni chegaralash, tizim servislarini taqdim qilishdagi hatoliklar, dastur va ma'lumotlar buzilishi, mahfiy ma'lumotlarni ochilishi, nozik taraflarni tuzatish, xujumlarni topish va bartaraf etish.

Упражнения

1. Перечислите наиболее важные составляющие надежности систем. Почему зависимость между стоимостью разработки системы и ее надежностью имеет экспоненциальный вид?

2. Почему функциональная надежность важна для критических систем? Назовите шесть причин.

3. Объясните на примерах трудности точного определения безотказности программных систем.

4. Оцените надежность какой-либо системы, которую вы регулярно используете, перечислив отказы системы и наблюдаемые сбои. Составьте руководство пользователя, в котором опишите, что нужно делать для эффективного использования системы при наличии этих сбоев.

5. Назовите шесть промышленных изделий, которые содержат или могут содержать в будущем программные системы, критические по обеспечению безопасности.

6. Объясните, почему обеспечение безотказности системы не гарантирует ее безопасности.

7. В медицинской системе, управляющей облучением опухолей, предусмотрите опасность, которая может возникнуть в процессе работы системы, и предложите программное средство, которое определяет эту опасность и предотвращает возможные несчастные случае, обусловленные ею.

8. Объясните зависимость между работоспособностью системы и ее защищенностью.

9. В терминах компьютерной безопасности объясните разницу между атакой на систему и угрозой ей.

10. Этично ли поставлять клиенту программную систему с известными ошибками? Разумно ли сообщать клиенту о существовании этих ошибок и следует ли после этого принимать претензии относительно надежности программного обеспечения?

11. Предположим, вы входили в состав группы разработчиков программного обеспечения для химического завода, которое сработало неправильно и стало причиной серьезного загрязнения окружающей среды. Ваш босс заявил в телевизионном интервью, что нет никаких ошибок в программном обеспечении и что проблемы возникли из-за неправильной эксплуатации системы. Вы согласитесь с таким объяснением? Обсудите, как вы должны отнестись к такому заявлению.

ГЛАВА 14. ВЕРИФИКАЦИЯ И АТТЕСТАЦИЯ ПО

Верификацией и аттестацией называют процессы проверки и анализа, в ходе которых проверяется соответствие программного обеспечения своей спецификации и требованиям заказчиков. Верификация и аттестация охватывают полный жизненный цикл ПО – они начинаются на этапе анализа требований и завершаются проверкой программного кода на этапе тестирования готовой программной системы.

Верификация и аттестация не одно и то же, хотя их легко перепутать. Кратко различие между ними можно определить следующим образом [45]:

• верификация отвечает на вопрос, правильно ли создана система;

• аттестация отвечает на вопрос, правильно ли работает система.

Согласно этим определениям, верификация проверяет соответствие ПО системной спецификации, в частности функциональным и нефункциональным требованиям. Аттестация– более общий процесс. Во время аттестации необходимо убедиться, что программный продукт соответствует ожиданиям заказчика. Аттестация проводится после верификации, для того чтобы определить, насколько система соответствует не только спецификации, но и ожиданиям заказчика.

Как уже отмечалось в главе 6, на ранних этапах разработки ПО очень важна аттестация системных требований. В требованиях часто встречаются ошибки и упущения; в таких случаях конечный продукт, вероятно, не будет соответствовать ожиданиям заказчика. Но, конечно, аттестация требований не может выявить все проблемы в спецификации требований. Иногда недоработки и ошибки в требованиях обнаруживаются только после завершения реализации системы.

В процессах верификации и аттестации используются две основные методики проверки и анализа систем.

1. **Инспектирование ПО.** Анализ и проверка различных

представлений системы, например документации спецификации требований, архитектурных схем или исходного кода программ. Инспектирование выполняется на всех этапах процесса разработки программной системы. Параллельно с инспектированием может выполняться автоматический анализ исходного кода программ и соответствующих документов. Инспектирование и автоматический анализ – это статические методы верификации и аттестации, поскольку им не требуется исполняемая система.

2. **Тестирование ПО**. Запуск исполняемого кода с тестовыми данными и исследование выходных данных и рабочих характеристик программного продукта для проверки правильности работы системы. Тестирование – это динамический метод верификации и аттестации, так как применяется к исполняемой системе.

На рис. 14.1 показано место инспектирования и тестирования в процессе разработки ПО. Стрелки указывают на те этапы процесса разработки, на которых можно применять данные методы. Согласно этой схеме, инспектирование можно выполнять на всех этапах процесса разработки системы, а тестирование – в тех случаях, когда создан прототип или исполняемая программа.

К методам инспектирования относятся: инспектирование программ, автоматический анализ исходного кода и формальная верификация. Но статические методы могут проверить только соответствие программ спецификации, с их помощью невозможно проверить правильность функционирования системы. Кроме того, статическими методами нельзя проверить такие нефункциональные характеристики, как производительность и надежность. Поэтому для оценивания нефункциональных характеристик проводится тестирование системы.

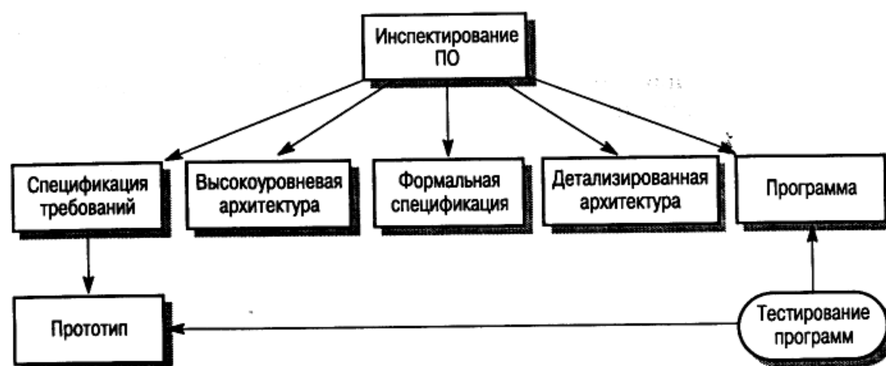

Рис. 14.1. Статическая и динамическая верификация и аттестация

В настоящее время, несмотря на широкое применение инспектирования ПО, преобладающим методом верификации и аттестации все еще остается тестирование. Тестирование – это проверка работы программ с данными, подобными реальным, которые будут обрабатываться в процессе эксплуатации системы. Наличие в программе дефектов и несоответствий требованиям обнаруживается путем исследования выходных данных и выявления среди них аномальных. Тестирование выполняется на этапе реализации системы (для проверки соответствия системы ожиданиям разработчиков) и после завершения ее реализации.

На разных этапах процесса разработки ПО применяют различные виды тестирования.

1. **Тестирование дефектов** проводится для обнаружения несоответствий между программой и ее спецификацией, которые обусловлены ошибками или дефектами в программах. Такие тесты разрабатываются для выявления ошибок в системе, а не для имитации ее работы. Данный вид тестирования рассматривается в главе 20.

2. **Статистическое тестирование** оценивает производительность и надежность программ, а также работу системы в различных режимах эксплуатации. Тесты разрабатываются так, чтобы имитировать реальную работу системы с реальными входными данными. Надежность функционирования системы оценивается по количеству сбоев, отмеченных в работе программ. Производительность оценивается по результатам измерения полного времени выполнения операций и времени отклика

системы при обработке тестовых данных. Статистическое тестирование и оценивание надежности рассматриваются в главе 21.

Конечно, между этими методами не существует жестких, четко установленных границ. Во время тестирования дефектов испытатель может получить интуитивное представление о надежности ПО, а во время статистического тестирования есть возможность выявления программных дефектов.

Главная цель верификации и аттестации – удостовериться в том, что система "соответствует своему назначению". Соответствие программной системы своему назначению отнюдь не предполагает, что в ней совершенно не должно быть ошибок. Скорее, система должна достаточно хорошо соответствовать тем целям, для которых планировалась. Уровень необходимой **достоверности соответствия** зависит от назначения системы, ожиданий пользователей и условий на рынке программных продуктов.

1. **Назначение ПО**. Уровень достоверности соответствия зависит от того, насколько критическим является разрабатываемое программное обеспечение по тем или иным критериям. Например, уровень достоверности для систем, критическим по обеспечению безопасности, должен быть значительно выше аналогичного уровня достоверности для опытных образцов программных систем, разрабатываемых для демонстрации некоторых новых идей.

2. **Ожидания пользователей**. Следует с грустью отметить, что в настоящее время у большинства пользователей невысокие требования к программному обеспечению. Пользователи настолько привыкли к отказам, происходящим во время работы программ, что не удивляются этому. Они согласны терпеть сбои в работе системы, если преимущества ее использования компенсируют недостатки. Вместе с тем с начала 1990-х годов терпимость пользователей к отказам в работе программных систем постепенно снижается. В последнее время создание ненадежных систем стало практически неприемлемым, поэтому компаниям, занимающимся

разработкой программных продуктов, необходимо все больше внимания уделять верификации и аттестации программного обеспечения.

3. **Условия рынка программных продуктов**. При оценке программной системы продавец должен знать конкурирующие системы, цену, которую покупатель согласен заплатить за систему, и назначенный срок выхода этой системы на рынок. Если у компании-разработчика несколько конкурентов, необходимо определить дату выхода системы на рынок до окончания полного тестирования и отладки, иначе первыми на рынке могут оказаться конкуренты. Если покупатели не желают приобретать ПО по высокой цене, возможно, они согласны терпеть большее количество отказов в работе системы. При определении расходов на процесс верификации и аттестации необходимо учитывать все эти факторы.

Как правило, в ходе верификации и аттестации в системе обнаруживаются ошибки. Для исправления ошибок в систему вносятся изменения. Этот **процесс отладки** обычно интегрирован с другими процессами верификации и аттестации. Вместе с тем тестирование (или более обобщенно – верификация и аттестация) и отладка являются разными процессами, которые имеют различные цели.

1. **Верификация и аттестация** – процесс обнаружения дефектов в программной системе.

2. **Отладка** – процесс локализации дефектов (ошибок) и их исправления (рис. 14.2).

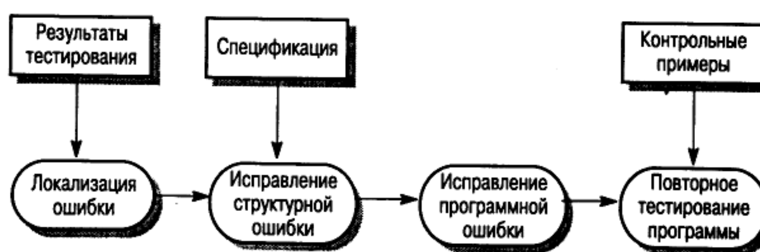

Рис. 14.2. Процесс отладки

Простых методов отладки программ не существует. Опытные отладчики обнаруживают ошибки путем сравнения шаблонов тестовых выходных данных с выходными данными тестируемых систем. Чтобы

определить местоположение ошибки, необходимы знания о типах ошибок, шаблонах выходных данных, языке программирования и процессе программирования. Очень важны знания о процессе разработке ПО. Отладчикам известны наиболее распространенные ошибки программистов (например, связанные с пошаговым увеличением значения счетчика). Также учитываются ошибки, типичные для определенных языков программирования, например связанные с использованием указателей в языке С.

Определение местонахождения ошибок в программном коде не всегда простой процесс, поскольку ошибка необязательно находится возле того места в коде программы, где произошел сбой. Чтобы локализовать ошибки, программист-отладчик разрабатывает дополнительные программные тесты, которые помогают выявить источник ошибки в программе. Может возникнуть необходимость в ручной трассировке выполнения программы.

Интерактивные средства отладки являются частью набора средств поддержки языка, интегрированных с системой компиляции программного кода. Они обеспечивают специальную среду выполнения программ, посредством которой можно получить доступ к таблице идентификаторов, а оттуда к значениям переменных. Пользователи часто контролируют выполнение программы пошаговым способом, последовательно переходя от оператора к оператору. После выполнения каждого оператора проверяются значения переменных и выявляются возможные ошибки.

Обнаруженная в программе ошибка исправляется, после чего необходимо снова проверить программу. Для этого можно еще раз выполнить инспектирование программы или повторить предыдущее тестирование. Повторное тестирование используется для того, чтобы убедиться, что сделанные в программе изменения не внесли в систему новых ошибок, поскольку на практике высокий процент "исправления ошибок" либо не завершается полностью, либо вносит новые ошибки в программу.

В принципе во время повторного тестирования после каждого исправления необходимо еще раз запускать все тесты, однако на практике такой подход оказывается слишком дорогостоящим. Поэтому при планировании процесса тестирования определяются зависимости между частями системы и назначаются тесты для каждой части. Тогда можно трассировать программные элементы с помощью специальных контрольных примеров (контрольных данных), подобранных для этих элементов. Если результаты трассировки задокументированы, то для проверки измененного программного элемента и зависимых от него компонентов можно использовать только некоторое подмножество всего множества тестовых данных.

14.1. Планирование верификации и аттестации

Верификация и аттестация – дорогостоящий процесс. Для больших систем, например систем реального времени со сложными нефункциональными ограничениями, половина бюджета, выделенного на разработку системы, тратится на процесс верификации и аттестации. Поэтому очевидна необходимость тщательного планирования данного процесса.

Планирование верификации и аттестации, как один из этапов разработки программных систем, должно начинаться как можно раньше. На рис. 14.3 показана модель разработки ПО, учитывающая процесс планирования испытаний. Здесь планирование начинается еще на этапах создания спецификации и проектирования системы. Данную модель иногда называют V-моделью (чтобы увидеть букву V, необходимо повернуть рис. 14.3 на 90°). На этой схеме также показано разделение процесса верификации и аттестации на несколько этапов, причем на каждом этапе выполняются соответствующие тесты.

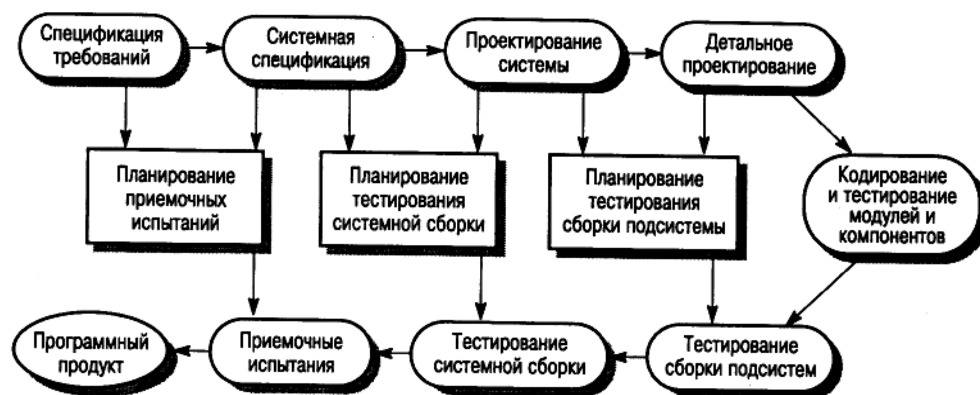

Рис. 14.3. Планирование испытаний в процессе разработки и тестирования

В процессе планирования верификации и аттестации необходимо определить соотношение между статическими и динамическими методами проверки системы, определить стандарты и процедуры инспектирования и тестирования ПО, утвердить технологическую карту проверок программ (см. раздел 14.2) и составить план тестирования программ. Чему уделить больше внимания – инспектированию или тестированию, зависит от типа разрабатываемой системы и опыта организации. Чем более критична система, тем больше внимания необходимо уделить статическим методам верификации.

Процесс тестирования

Описание основных этапов процесса тестирования.

Возможность отслеживания требований

Тестирование следует спланировать таким образом, чтобы протестировать все требования в отдельности.

Тестируемые элементы

Следует определить все "выходные" продукты процесса разработки ПО, которые необходимо тестировать.

График тестирования

Составляется временной график тестирования и распределение ресурсов согласно этому графику. Очевидно, что график тестирования привязан к более общему графику разработки проекта.

Процедуры записи тестов

Недостаточно только проводить тесты – результаты тестирования необходимо систематически записывать, чтобы потом можно было исследовать процесс тестирования и проверить правильность его выполнения.

Аппаратные и программные требования

В этом разделе определяются все необходимые для тестирования инструментальные программные и аппаратные средства,

Ограничения

В этом разделе следует попытаться предвидеть все неблагоприятные факторы, влияющие на процесс тестирования, например нехватку персонала.

В плане верификации и аттестации основное внимание уделяется стандартам процесса тестирования, а не описанию конкретных тестов. Этот план предназначен не только для руководства, он в основном предназначен специалистам, занимающимся разработкой и тестированием систем. План дает возможность техническому персоналу получить полную картину испытаний системы и в этом контексте спланировать свою работу. Кроме того, план предоставляет информацию менеджерам, отвечающим за то, чтобы у группы тестирования были все необходимые аппаратные и программные средства.

Основные компоненты плана испытаний ПО перечислены во врезке 14.1. Хорошее описание подобных планов и их взаимосвязи с более общими планами обеспечения качества представлено в работе [118].

Подобно другим планам, план испытаний не является неизменным документом. Его следует регулярно пересматривать, так как тестирование зависит от процесса реализации системы. Например, если реализация какой-либо части системы не завершена, то невозможно провести тестирование сборки системы. Поэтому план необходимо периодически пересматривать, чтобы сотрудников, занятых тестированием, можно было использовать на других работах.

14.2. Инспектирование программных систем

Системное тестирование программ требует разработки огромного количества тестов, их выполнения и проверки. Это значит, что данный процесс достаточно трудоемкий и дорогостоящий. Каждый тест позволяет обнаруживать одну, а в лучшем случае несколько ошибок в программе. Причина такого положения заключается в том, что сбои в работе, происходящие из-за ошибок в системе, часто приводят к разрушению данных. Поэтому трудно сказать, какое количество ошибок "ответственно" за сбой в системе.

Инспектирование программ не требует их исполнения, поэтому данный метод можно использовать до завершения полной реализации программ. Во время инспектирования проверяется исходное представление системы. Это может быть модель системы, спецификация или программа, написанная на языке высокого уровня. Для обнаружения ошибок используется знание разрабатываемой системы и семантика ее исходного представления. Каждую ошибку можно рассматривать отдельно, не обращая внимания на то, как она влияет на поведение системы.

Доказано, что инспектирование является эффективным методом обнаружения ошибок. Также немаловажно, что инспектирование значительно дешевле экстенсивного тестирования программ. В экспериментах, описанных в работе [27], сравнивалась эффективность инспектирования и тестирования. Инспектирование программного кода оказалось более эффективным и менее дорогостоящим, чем тестирование. Такие же выводы сделаны в работе [129].

В статье [112] утверждается, что более 60% ошибок в программах можно обнаружить с помощью неформального исследования (инспектирования) программ. При более формальном подходе, использующем математические методы, в программе можно обнаружить

более 90% всех ошибок [239]. Такая проверка используется в процессе разработки систем методом "чистая комната", который рассматривается в разделе 14.4. Процесс инспектирования также может оценить другие качественные характеристики систем: соответствие стандартам, переносимость и удобство сопровождения. Качественные характеристики систем рассматриваются в главе 16.

В системных компонентах и подсистемах выявление ошибок путем просмотра и инспектирования обычно более эффективно, чем с помощью тестирования, по двум причинам.

1. За один сеанс инспектирования можно выявить множество разнообразных программных дефектов. Недостатком тестирования является то, что обычно за один сеанс тестирования можно обнаружить только одну ошибку, поскольку ошибки могут привести к отказу системы или их эффекты могут накладываться друг на друга.

2. Инспектирование использует знания о предметной области и языке программирования. Специалист, проводящий инспектирование, должен знать типы ошибок, присущие конкретным языкам программирования и приложениям определенного типа. Поэтому в ходе анализа программ есть возможность сосредоточиться только на конкретных типах ошибок.

Конечно, инспектирование не может полностью заменить тестирование. Инспектирование лучше использовать как начальный процесс верификации для обнаружения большей части программных дефектов. Путем инспектирования проверяют соответствие ПО ее спецификации, однако таким способом нельзя проверить динамическое поведение системы. Более того, нерационально инспектировать законченные системы, собранные из нескольких подсистем. На этом уровне возможно только тестирование. Тестирование также необходимо для оценки надежности и производительности, проверки пользовательского интерфейса и соответствия системы требованиям заказчика.

Инспектирование и тестирование не являются конкурирующими методами верификации и аттестации. Каждому из них присущи свои преимущества и недостатки, поэтому в процессе верификации и аттестации их следует использовать совместно. Одним из наиболее эффективных методов инспектирования является применение контрольных примеров [129]. В этом случае можно обнаружить программные дефекты и разработать более эффективные методы тестирования системы.

Иногда при инспектировании в организации, разрабатывающей традиционное программное обеспечение, возникают трудности. Разработчики, имеющие опыт тестирования программ, неохотно соглашаются с тем, что инспектирование оказывается более эффективным методом выявления ошибок, чем тестирование. Менеджеры относятся к этим технологиям с недоверием, потому что внедрение инспектирования на этапах проектирования и разработки требует дополнительных расходов. Инспектирование всегда требует расходов, причем на начальном этапе разработки ПО, а конечная экономия средств вследствие применения инспектирования достигается только благодаря опыту проводящих его специалистов.

В этой главе рассматривается инспектирование программ, т.е. исходный код проверяется на наличие ошибок. Однако метод инспектирования можно также использовать для верификации любых текстовых документов, созданных в процессе разработки ПО. Метод инспектирования можно применять к спецификации требований, для детализированного определения системной архитектуры, при разработке структур данных, планировании тестирования и в процессе создания системной документации.

14.3. Инспектирование программ

Инспектирование программ – это просмотр и проверка программ с целью обнаружения в них ошибок. Идея формализованного процесса проверки (инспекции) программ впервые сформулирована IBM в 1970-х годах и описана в работах [111, 112]. В настоящее время данный метод верификации программ получил широкое применение. На базе исходного метода инспектирования разработано много других вариантов инспектирования программ [129]. Но все они основываются на базовой идее метода инспектирования, согласно которому группа специалистов выполняет тщательный построчный просмотр и анализ исходного кода программы.

Основное отличие инспектирования от других видов оценивания качества программ состоит в том, что главная его цель – обнаружение дефектов, а не исследование общих проблем проекта. Дефектами являются либо ошибки в программе, либо несоответствие программы организационным или проектным стандартам. В противоположность инспектированию другие методы анализа программ основное внимание удаляют организационным вопросам, временному графику работ, затратам, сравнению с промежуточными контрольными элементами или оценке соответствия ПО определенным целям организации-разработчика.

Процесс инспектирования – это формализованный процесс, выполняемый небольшой группой специалистов, состоящей не более чем из четырех человек. Члены группы системно анализируют программу и определяют возможные дефекты. Согласно исходной концепции метода инспектирования члены группы должны выполнять следующие роли: автора, рецензента, инспектора и координатора. Рецензент "озвучивает" программный код, инспектор проверяет код с помощью тестов, координатор отвечает за организацию процесса.

По мере накопления опыта инспектирования в организациях могут появляться другие предложения по распределению ролей в группе. В ходе обсуждения результатов использования инспектирования, внедренного в процесс разработки программ в компании Hewlett-Packard, в статье [136]

предлагается шесть ролей (табл. 14.1). Одно лицо может исполнять несколько ролей, поэтому количество членов в группе инспектирования может варьироваться.

Таблица 14.1. Роли в процессе инспектирования

Роль	Описание
Автор или владелец	Программист или разработчик, который отвечает за создание программы или документа, а также несет ответственность за исправление дефектов, обнаруженных в процессе инспектирования
Инспектор	Находит ошибки, упущения и противоречия в программах и документах; может также указать на более общие проблемы, находящиеся вне сферы действия инспекционной группы
Рецензент	Излагает код или документ на собрании инспекционной группы
Секретарь	Записывает результаты собрания инспекционной группы
Председатель или координатор	Управляет и организует процесс инспектирования. Докладывает о результатах инспектирования руководству компании
Руководитель группы	Занимается совершенствованием процесса инспектирования, обновлениями технологических карт, разработкой стандартов и т.п.

Как показано в статье [136], роль рецензента необязательна. В этом случае исходный процесс инспектирования, в котором рецензирование программы является важной составляющей, соответственно изменяется. Такой же вывод содержится в работе [129].

Для начала процесса инспектирования программы необходимы следующие условия:

1. Наличие точной спецификации кода, предназначенного для инспектирования. Без полной спецификации невозможно обнаружить дефекты в проверяемом программном компоненте.

2. Члены инспекционной группы должны хорошо знать стандарты

разработки.

3. В распоряжении группы должна была синтаксически корректная последняя версия программы. Нет никакого смысла рассматривать код, который "почти завершен".

На рис. 14.4 показан общий процесс инспектирования. Он адаптирован к требованиям организаций, использующих инспектирование программ.

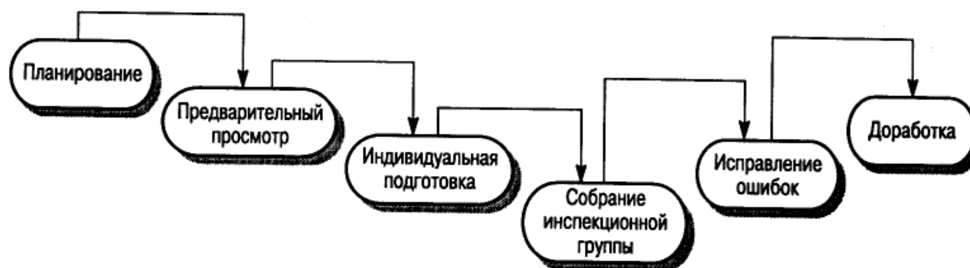

Рис. 14.4. Процесс инспектирования

Координатор составляет план инспектирований, подбирает инспекционную группу, организует собрание и убеждается, что программа и ее спецификация закончены. Программа, предназначенная для инспектирования, передается на рассмотрение инспекционной группе, где автор программы описывает ее назначение (этап предварительного просмотра). После этого следует этап индивидуальной подготовки, на котором каждый член инспекционной группы изучает программу и ее спецификацию и выявляет дефекты в программном коде.

Сам процесс инспектирования должен быть относительно коротким (не более двух часов) и сосредоточенным исключительно на выявлении дефектов, аномалий и несоответствий стандартам. Инспекционная группа не должна предлагать способы исправления обнаруженных дефектов или рекомендовать какие-либо изменения в других программных компонентах.

После инспектирования автор изменяет программу, исправляя обнаруженные ошибки. На этапе доработки координатор принимает решение о том, необходимо ли повторно проводить инспектирование. Если повторного инспектирования не требуется, все обнаруженные дефекты

документально фиксируются и документ с результатами инспектирования утверждается председателем.

Процесс инспектирования должен проводиться с учетом технологической карты, описывающей возможные ошибки программирования. Карта разрабатывается квалифицированными специалистами и регулярно обновляется по мере накопления опыта в процессе инспектирования. Для разных языков программирования составляются разные технологические карты.

Технологические карты, составленные для разных языков программирования, различаются между собой, поскольку учитывают возможности проверки, которую обеспечивают компиляторы языков. Например, компилятор языка Ada проверяет количество параметров функций, а компилятор языка С – нет. Ошибки, которые можно выявить в процессе инспектирования, перечислены в табл. 14.2. Подчеркнем, что каждая организация должна разрабатывать собственные технологические карты для инспектирования, которые бы основывались на стандартах и опыте данной организации и обновлялись по мере обнаружения новых типов программных дефектов [129].

Таблица 14.2. Ошибки, выявляемые при инспектировании

Класс ошибок	**Вопросы, помогающие выявлять ошибки**
Ошибки данных	Все ли переменные в программе инициализированы до начала использования их значений? Все ли константы именованы? Равна ли верхняя граница массива его размеру или на единицу меньше этого размера? Какой разделитель используется для разделения символьных строк? Возможно ли переполнение буфера?
Ошибки	Выполняются ли условия для каждого условного оператора?

управления	Все ли циклы завершаются?
	Правильно ли в составных операторах расставлены скобки?
	Все ли выборы выполняются в операторах выбора?
Ошибки ввода-вывода	Используются ли в программе входные переменные?
	Всем ли выходным переменным перед выводом присваиваются значения?
	Могут ли какие-нибудь входные данные привести к нарушению системных данных?
Ошибки интерфейса	Все ли вызовы процедур и функций содержат правильное количество параметров?
	Согласованы ли типы формальных и фактических параметров?
	В правильном ли порядке расположены параметры?
	Если компоненты обращаются к разделяемой памяти, имеют ли они такую же модель структуры разделяемой памяти?
Ошибки управления памятью	Если связанная структура данных изменяется, правильно ли переопределяются все связи?
	Если используется динамическая память, правильно ли она распределяется?
	Происходит ли перераспределение памяти после того, как она больше не используется?
Ошибки управления исключениями	Все ли возможные ошибки рассмотрены в условиях, определяющих исключительные ситуации?

В процессе инспектирования организация накапливает определенный опыт, поэтому результаты инспектирования можно использовать для улучшения всего процесса разработки ПО. В ходе инспектирования

выполняется анализ обнаруженных дефектов. Группа инспектирования и авторы инспектируемого кода определяют причины возникновения дефектов. Чтобы подобные дефекты не возникали в будущих системах, необходимо по возможности устранить причины возникновения дефектов, что означает внесение изменений в процесс разработки программных систем.

Количество кода, проверяемого за определенное время, зависит от опыта группы инспектирования, языка программирования и предметной области приложения. На основе опыта проведения инспектирования в компании IBM сделаны следующие оценки.

1. На этапе предварительного просмотра за один час можно просмотреть приблизительно 500 операторов исходного кода.

2. Во время индивидуальной подготовки за один час можно проверить примерно 125 операторов исходного кода.

3. На собрании за один час можно проверить от 90 до 125 операторов.

Эти цифры также подтверждаются данными, полученными во время проведения инспектирования компании AT&T [22].

Принято считать, что максимальная длительность инспектирования не должна превышать двух часов, поскольку потом эффективность обнаружения дефектов снижается. Поэтому в процессе разработки ПО инспектирование относительно малых системных компонентов должно выполняться достаточно часто.

Если команда инспектирования состоит из четырех человек, на проверку 100 строк кода требуется примерно один человеко-день. Считается, что сам процесс инспектирования занимает около часа, плюс каждый член команды тратит 1-2 часа на подготовку к инспектированию. Расходы на тестирование также существенно зависят от количества ошибок в программе. Но, с другой стороны, на инспектирование программы требуется вполовину меньше затрат, чем на эквивалентное тестирование программ.

Обеспечение инспектирования ПО требует квалифицированного управления и "правильного" отношения к результатам его проведения. Инспектирование – открытый процесс обнаружения ошибок, когда ошибки, допущенные отдельным программистом, неизбежно выявляются и становятся известны всей группе программистов. Менеджеры должны четко разграничивать инспектирование программного кода и оценку кадров. При оценке профессиональных качеств специалистов ни в коем случае нельзя учитывать ошибки, обнаруженные в процессе инспектирования. Руководителям инспекционных групп необходимо пройти тщательную подготовку, чтобы грамотно управлять процессом и совершенствовать культуру отношений, которая гарантировала бы поддержку в процессе обнаружения ошибок и отсутствие каких-либо обвинений в связи с этими ошибками.

14.4. Автоматический статический анализ программ

Статические анализаторы программ – это инструментальные программные средства, которые сканируют исходный текст программы и выявляют возможные ошибки и противоречия. Для анализаторов не требуется исполняемая программа. Они выполняют синтаксический разбор текста программы и опознают различные типы операторов. Таким образом, с помощью анализаторов можно проверить, правильно ли составлены операторы, сделать выводы относительно потока управления в программе и во многих случаях вычислить множество значений данных, используемых программой. Анализаторы дополняют средства обнаружения ошибок, предоставляемых компилятором языка.

Цель автоматического статического анализа – привлечь внимание проверяющего к аномалиям в программе, например к переменным, которые используются без инициализации или совсем не используются, или к данным, значения которых превышают заданное, и т.п. В табл. 14.3

перечислены типы ошибок, которые можно выявить с помощью статического анализа. Автоматический статический анализ лучше всего применять вместе с инспектированием ПО, так как он предоставляет дополнительную информацию инспекционной группе.

Таблица 14.3. Ошибки, обнаруживаемые статическим анализом

Тип ошибки	Описание ошибки
Ошибки данных	Переменные используются до их инициализации; переменные определены, но нигде не используются; переменным дважды присваиваются значения, однако между этими присвоениями они нигде не используются; выход за границы массива; переменные не определены
Ошибки управления	Неиспользуемый код; безусловные переходы в циклах
Ошибки ввода-вывода	Переменные выводятся дважды без промежуточного присвоения
Ошибки интерфейса	Неправильный тип параметра; неправильное количество параметров; результаты функции не используются; есть невызываемые процедуры и функции
Ошибки управления памятью	Ошибки в использовании указателей

Статический анализ состоит из нескольких этапов:

1. **Анализ потока управления**. На этом этапе идентифицируются и выделяются циклы, их точки входа и выхода, а также неиспользуемый код (это код, окруженный безусловными операторами перехода, или код одной из ветвей условного оператора, условие перехода к которой никогда не будет истинным).

2. **Анализ использования данных**. На этом этапе проверяется использование переменных в программе. Анализ позволяет обнаружить переменные, которые используются без предварительной инициализации, переменные, которые описаны дважды без промежуточного присвоения, а также объявленные, но нигде не используемые переменные. На этом этапе также можно выявить условные операторы с избыточными условиями. Это

такие условия, значения которых никогда не изменяются: они либо всегда истинны, либо всегда ложны.

3. **Анализ интерфейса**. На этом этапе проверяется согласованность различных частей программы, правильность объявления процедур и их использования. Данный этап оказывается лишним, если используется язык со строгим контролем типов, например Java, так как подобный анализ выполняет компилятор этого языка. Анализ интерфейса помогает выявить ошибки в программах, написанных на языках со слабым контролем типов, например FORTRAN или C. В процессе анализа интерфейса можно также выявить объявленные функции и процедуры, которые нигде не вызываются, и функции, результаты которых не используются.

4. **Анализ потоков данных**. На этом этапе анализа определяются зависимости между исходными (входными) и результирующими (выходными) переменными. Хотя такой анализ не выявляет конкретных ошибок, он дает полный список значений, используемых в программе, благодаря чему легче обнаружить ошибочный вывод данных. На этом этапе также можно явно определить условия, которые влияют на значения переменных.

5. **Анализ ветвей программы**. На этом этапе семантического анализа определяются все ветви программы и выделяются операторы, исполняемые в каждой ветви. Анализ ветвей программы существенно помогает разобраться в управлении программой и позволяет проанализировать каждую ветвь отдельно.

Анализ потока данных и анализ ветвей генерируют огромное количество информации. Эта информация не выявляет конкретных ошибок, а представляет программу в разных аспектах. Из-за огромного количества генерируемой информации эти этапы статического анализа иногда исключают из процесса анализа и используют только на ранних стадиях для обнаружения аномалий в разрабатываемой программе.

Статические анализаторы особенно полезны в тех случаях, когда используются языки программирования, подобные С. В языке С нет строгого контроля типов, и потому проверка, осуществляемая компилятором языка С, ограниченна. В этом случае средствами статического анализа можно автоматически выявить широкий спектр ошибок программирования. Данный анализ особенно важен при разработке критических систем. В этом случае статический анализ позволяет значительно сократить расходы на тестирование.

В системах Unix и Linux есть статический анализатор LINT для программ, написанных на С. Он обеспечивает статическую проверку, эквивалентную проверке компилятором в языках со строгим контролем типов, например Java. В листинге 14.1 представлен образец результата проверки программы с помощью анализатора LINT. Первая команда анализатора просматривает программу. В программе определена функция printarray с одним параметром, которая затем вызывает ее с тремя параметрами. Переменные i и с определены, однако значения им нигде не присваиваются. Возвращаемое функцией значение также нигде не используется.

Листинг 14.1. Статический анализ, выполненный анализатором LINT

```
138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main()
```

```
{
int Anarray[5]; int i; char c;
printarray(Anarray,i/c);
printarray(Anarray);
}
139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args.lint_ex.c(4)::lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4)::lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4)::lint_ex.c(11)
printf returns value which is always ignored
```

В строке 139 выполняется компиляция С-программы, в результате которой компилятор не обнаружил ни одной ошибки. В следующей за ней строке следует вызов анализатора LINT, который находит ошибки и предоставляет отчет.

Статический анализатор выявил, что используемые скалярные переменные **С** и **i** не инициализированы, функция **printarray** вызывается с другим количеством аргументов, чем указано при ее определении. Также обнаружено непоследовательное использование первого аргумента в функции **printarray**, кроме того, анализатор обнаружил, что значение функции нигде не используется.

Анализ с помощью инструментальных средств не может заменить инспектирования, так как существуют такие типы ошибок, которые невозможно выявить с помощью статического анализа. Например, анализаторы могут обнаружить необъявленные переменные, однако они не в состоянии определить неправильные присвоения. В языках со слабым контролем типов, например С, статические анализаторы могут определить

функции с неверным количеством и типом аргументов, однако не способны распознать ситуации, когда в функции пропущен неверный аргумент правильного типа.

Конечно, для таких языков, как C, статический анализ является эффективным методом обнаружения ошибок. Но в современных языках программирования, например Java, из языка удалены конструкции, способствующие появлению многих ошибок. Все переменные должны быть объявлены, отсутствуют операторы безусловного перехода, вследствие чего маловероятно случайное создание неиспользуемого кода, и осуществляется автоматическое управление памятью. Такой подход к устранению ошибок более эффективен для повышения надежности программ, чем любые методы обнаружения ошибок. Поэтому для Java-программ использовать автоматический статический анализ нерентабельно.

14.5. Метод "чистая комната"

При разработке ПО методом "чистая комната" (cleanroom) для устранения дефектов используется процесс строгого инспектирования [239, 75, 219, 284]. Цель данного метода– создание ПО без дефектов. Название "чистая комната" взято по аналогии с производством кристаллов полупроводников, где выращивание кристаллов без дефектов происходит в сверхчистой атмосфере (чистых комнатах). Я описываю этот метод в данной главе, поскольку согласно ему в процессе разработки ПО при проверке соответствия системных компонентов спецификациям тестирование заменяется инспектированием.

На рис. 14.5 представлена модель процесса разработки ПО методом "чистая комната", построенная на основе описания, приведенного в работе [219].

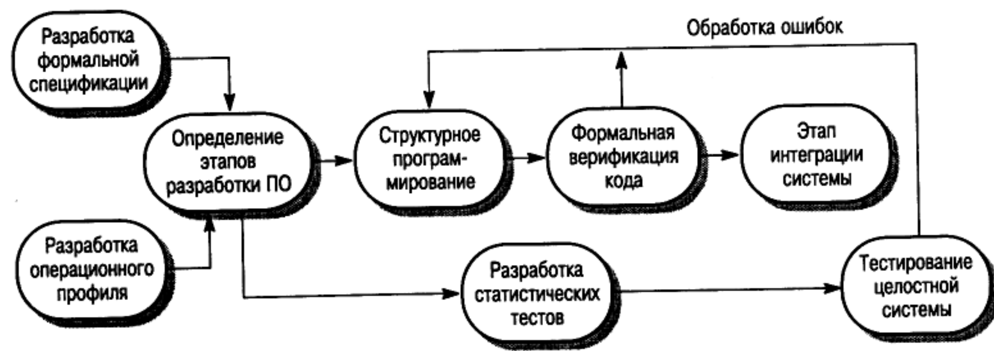

Рис. 14.5. Процесс разработки ПО методом "чистая комната"

В разработке ПО методом "чистая комната" можно выделить пять ключевых моментов.

1. **Формальная спецификация**. Для создаваемой системы разрабатывается формальная спецификация. Для записи спецификации используется модель состояний, в которой отображены отклики системы на стимулы.

2. **Пошаговая разработка**. Разработка ПО разбивается на несколько этапов, которые выполняются и пррверяются методом "чистая комната" независимо друг от друга. Этапы определяются совместно с заказчиком на ранних стадиях процесса создания программного продукта.

3. **Структурное программирование**. Используется только ограниченное количество управляющих конструкций и абстракций данных. Процесс разработки программы – это процедура поэтапной детализации спецификации.

4. **Статическая верификация**. Разрабатываемое ПО проверяется статическим методом строгого инспектирования ПО. Для модулей или отдельных элементов тестирование кода не проводится.

5. **Статистическое тестирование системы**. На каждом шаге разработки проводится тестирование статистическими методами, позволяющими оценить надежность программной системы. Статистические методы рассматриваются в главе 21. Как показано на рис. 14.5, статистические тесты базируются на операционном профиле, который разрабатывается параллельно созданию спецификации системы.

Пошаговая разработка ПО, схема которой показана на рис. 14.6, описана в главе 3. Выполнение каждого этапа определяется пользователями. На каждом отдельном этапе получается вполне работоспособная система, но с ограниченными возможностями. Пользователи возвращают отчеты о функционировании системы вместе с предложениями необходимых изменений. Пошаговая разработка ПО позволяет уменьшить количество ошибок, возникающих из-за изменений требований заказчика.

Рис. 14.6. Процесс пошаговой разработки

Если спецификация определена как единое целое, изменения в требованиях заказчика (которые неизбежны) влекут за собой изменения в спецификации и в процессе разработки. В этом случае спецификация и системная архитектура должны постоянно пересматриваться. В методе пошаговой разработки спецификация на каждом шаге фиксирована, хотя требования по изменению других частей системы принимаются. Каждый шаг разработки завершается готовым программным продуктом.

На первых этапах пошаговой разработки ПО методом "чистая комната" реализуются наиболее критические для заказчика системные функции. Менее важные системные функции добавляются на последующих этапах. Таким образом, у заказчика есть возможность проверить и испытать систему (ее основные функции) до окончательного завершения разработки. Если возникают проблемы с требованиями, заказчик сообщает об этом группе разработчиков и запрашивает новую версию продукта.

Таким образом, наиболее важные для заказчика системные функции проверяются наибольшее количество раз. По мере добавления в систему новые функции комбинируются с уже имеющимися, и интегрированная

система тестируется. Поэтому те части системы, которые созданы на первых этапах разработки, на каждом из последующих этапов проверяются еще раз с помощью других контрольных тестов.

Процесс разработки ПО методом "чистая комната" планируется таким образом, чтобы обеспечить строгое инспектирование программ. Спецификация системы представлена моделью состояний, которая через ряд последовательных моделей постепенно преобразуется в исполняемую программу. Этот подход к разработке ПО описан в главе 3. Инспектирование программ дополняются строгими математическими доказательствами согласованности и корректности преобразований.

Обычно разработкой больших систем методом "чистая комната" занимаются три группы разработчиков.

1. **Группа спецификации**. Отвечает за разработку и поддержку системной спецификации. Этой группой создаются спецификации пользовательских требований и формальные спецификации для верификации системы. В некоторых случаях, например после окончания разработки спецификации, эта группа может присоединиться к группе разработки.

2. **Группа разработки**. Занимается разработкой и проверкой ПО. При проверке используется структурированный формальный подход, основанный на инспектировании кода, подкрепленный доказательством правильности работы системы.

3. **Группа сертификации**. Занимается разработкой статистических тестов, применяемых после окончания разработки ПО. Все тесты основаны на использовании формальной спецификации. Контрольные тесты разрабатываются параллельно с созданием системы и используются для сертификации надежности ПО.

В результате использования метода "чистая комната" готовый программный продукт содержит крайне мало ошибок и его стоимость меньше, чем у разработанного традиционными методами. В работе [75] описано несколько успешных проектов, разработанных методом "чистая

комната", с неизменно низким процентом ошибок в разработанных системах. Расходы на эти проекты сравнимы с расходами на проекты, которые разрабатывались с использованием традиционных методов.

В процессе разработки методом "чистая комната" оказывается рентабельной статическая проверка. Огромное количество дефектов обнаруживается еще до исполнения программы и исправляется в процессе разработки ПО. В статье [219] утверждается, что во время тестирования проектов, которые разрабатывались с использованием метода "чистая комната", в среднем обнаруживается только 2,3 дефекта на тысячу строк исходного кода. В целом расходы на разработку не увеличиваются, так как сокращаются расходы на тестирование и исправление ошибок в разрабатываемой программной системе.

Контрольные вопросы:

1) Какие две основные методики проверки и анализа систем используются в процессах верификации и аттестации?
2) От чего зависит уровень необходимой достоверности соответствия?
3) Что такое инспектирование программ?
4) Перечислите условия необходимые для начала процесса инспектирования программы.
5) Что такое статические анализаторы программ?
6) Из каких этапов состоит статический анализ?
7) Каких пять ключевых моментов можно выделить в разработке ПО методом "чистая комната"?

Ключевые слова: *инспектирование по, тестирование по, тестирование дефектов, статистическое тестирование, назначение по, ожидания пользователей, условия рынка программных продуктов процесс отладки, верификация и аттестация, отладка, инспектирование программ, процесс инспектирования, статические анализаторы программ, анализ потока управления, анализ использования данных, анализ интерфейса, анализ потоков данных, анализ ветвей программы, формальная спецификация, пошаговая разработка, структурное программирование, статическая верификация, статистическое тестирование системы, группа спецификации, группа*

разработки, группа сертификации.

Keywords: *inspection by testing for testing defects, statistical tests, the appointment to the expectations of users, the conditions of the market software debugging, verification and validation, debugging, inspection programs, inspection process, static analyzers program control flow analysis, analysis of use data analysis interface, data flow analysis, analysis of program branches, formal specification, step by step development, structured programming, static verification, statistical testing of the system, a group of specifications, development team, a group certification.*

Kalit so'zlar: *dasturiy ta'minotni nazorat qilish, tekshirish, nuqsonlarni tekshirish, statistik tekshirish, DT maqsadi, foydalanuvchi kutgan natija, dasturiy mahsulotlar bozori sharoitlari, sozlash jarayoni, attestatsiya, dasturni nazorat qilish, nazorat qilish jarayoni, tekshiruv oqimi tahlili, ma'lumotlarni ishlatishi tahlili, interfeys tahlili, dastur sohalari tahlili, ishlab chiqarish guruhi, srtifikatlash guruhi.*

Упражнения

1. Обсудите различия между верификацией и аттестацией и объясните, почему аттестация является более сложным процессом.

2. Объясните, почему не нужно устранять все дефекты в программе перед ее поставкой заказчику. До каких пор следует тестировать программу, чтобы удостовериться, что она соответствует своему назначению?

3. Объясните, почему инспектирование программы является эффективным методом обнаружения в ней ошибок. Какие типы ошибок нельзя обнаружить методом инспектирования?

4. Разработайте технологическую карту наиболее распространенных ошибок (не синтаксических), которые не обнаруживаются компилятором, но которые можно выявить при инспектировании программы. При составлении таблицы используйте свои знания Java, C++, C или других языков программирования.

5. Составьте список вопросов, ответы на которые позволяют во время проведения статического анализа обнаружить ошибки в программах, написанных на языках Java, Ada и C++. Сравните свой список со списком, представленным в табл. 19.2.

6. Составьте отчет, в котором бы приводились преимущества метода "чистая комната", а также связанные с ним расходы и риски.

7. Менеджер решил для оценки специалистов в качестве исходных данных воспользоваться отчетами о результатах инспектирования программ. В отчетах содержится информация о том, кто совершил и кто обнаружил ошибки в программе.

Этичны ли действия менеджера? Этично ли заранее проинформировать персонал об этом? Как это решение может повлиять на процесс инспектирования?

8. Один из подходов, широко используемых при тестировании ПО, состоит в тестировании системы до тех пор, пока не будут израсходованы все средства, выделенные на тестирование. Затем система передается заказчикам. Обсудите этичность такого подхода.

ГЛАВА 15. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В главе 3 рассматривалась общая схема процесса тестирования. Он начинается с тестирования отдельных программных модулей, например процедур и объектов. Затем модули компонуются в подсистемы и потом в систему, при этом проводится тестирование взаимодействий между модулями. Наконец, после сборки системы, заказчик может провести серию приемочных тестов, во время которых проверяется соответствие системы ее спецификации.

На рис. 15.1 показана схема двухэтапного процесса тестирования. На этапе покомпонентного тестирования проверяются отдельные компоненты. Это могут быть функции, наборы методов, собранные в один модуль, или объекты. На этапе тестирования сборки эти компоненты интегрируются в подсистемы или законченную систему. На этом этапе основное внимание уделяется тестированию взаимодействий между компонентами, а также показателям функциональности и производительности системы как единого целого. Но, конечно, на этапе тестирования сборки также могут обнаруживаться ошибки в отдельных компонентах, не замеченные на этапе покомпонентного тестирования.

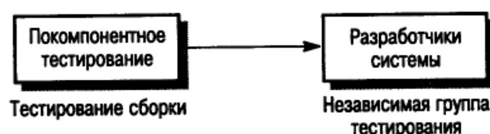

Рис. 15.1. Этапы тестирования ПО

При планировании процесса верификации и аттестации ПО менеджеры проекта должны определить, кто будет отвечать за разные этапы тестирования. Во многих случаях за тестирование своих программ (модулей или объектов) несут ответственность программисты. За следующий этап отвечает группа системной интеграции (сборки), которая интегрирует отдельные программные модули (возможно, полученные от разных разработчиков) в единую систему и тестирует эту систему в целом.

Для критических систем процесс тестирования должен быть более формальным. Такая формализация предполагает, что за все этапы тестирования отвечают независимые испытатели, все тесты разрабатываются отдельно и во время тестирования ведутся подробные записи. Чтобы протестировать критические системы, независимая группа разрабатывает тесты, исходя из спецификации каждого системного компонента.

При разработке некритических, "обычных" систем подробные спецификации для каждого системного компонента, как правило, не создаются. Определяются только интерфейсы компонентов, причем за проектирование, разработку и тестирование этих компонентов несут ответственность отдельные программисты или группы программистов. Таким образом, тестирование компонентов, как правило, основывается только на понимании разработчиками того, что должен делать компонент.

Тестирование сборки должно основываться на имеющейся спецификации системы. При составлении плана тестирования обычно используется спецификация системных требований или спецификация пользовательских требований (см. главу 3). Тестированием сборки всегда занимается независимая группа.

Во многих книгах, посвященных тестированию программного обеспечения, например [34, 197, 276, 22*], описывается процесс тестирования программных систем, реализующих функциональную модель ПО, но не рассматривается отдельно тестирование объектно-

ориентированных систем. В контексте тестирования между объектно-ориентированными и функционально-ориентированными системами имеется ряд отличий.

1. В функционально-ориентированных системах существует четко определенное различие между основными программными элементами (функциями) и совокупностью этих элементов (модулями). В объектно-ориентированных системах этого нет. Объекты могут быть простыми элементами, например списком, или сложными, например такими, как объект метеорологической станции из главы 12, состоящий из ряда других объектов.

2. В объектно-ориентированных системах, как правило, нет такой четкой иерархии объектов, как в функционально-ориентированных системах. Поэтому такие методы интеграции систем, как нисходящая или восходящая сборка (см. раздел 15.2), часто не подходят для объектно-ориентированных систем.

Таким образом, в объектно-ориентированных системах между тестированием компонентов и тестированием сборки нет четких границ. В таких системах процесс тестирования является продолжением процесса разработки, где основной системной структурой являются объекты. Несмотря на то что большая часть методов тестирования подходит для систем любых видов, для тестирования объектно-ориентированных систем необходимы специальные методы. Такие методы рассмотрены в разделе 15.3.

15.1. Тестирование дефектов

Целью тестирования дефектов является выявление в программной системе скрытых дефектов до того, как она будет сдана заказчику. Тестирование дефектов противоположно аттестации, в ходе которой проверяется соответствие системы своей спецификации. Во время аттестации система должна корректно работать со всеми заданными тестовыми данными. При тестировании дефектов запускается такой тест, который

вызывает **некорректную** работу программы и, следовательно, выявляет дефект. Обратите внимание на эту важную особенность: тестирование дефектов демонстрирует **наличие**, а не отсутствие дефектов в программе.

Общая модель процесса тестирования дефектов показана на рис. 15.2. **Тестовые сценарии** – это спецификации входных тестовых данных и ожидаемых выходных данных плюс описание процедуры тестирования. Тестовые данные иногда генерируются автоматически. Автоматическая генерация тестовых сценариев невозможна, поскольку результаты проведения теста не всегда можно предсказать заранее.

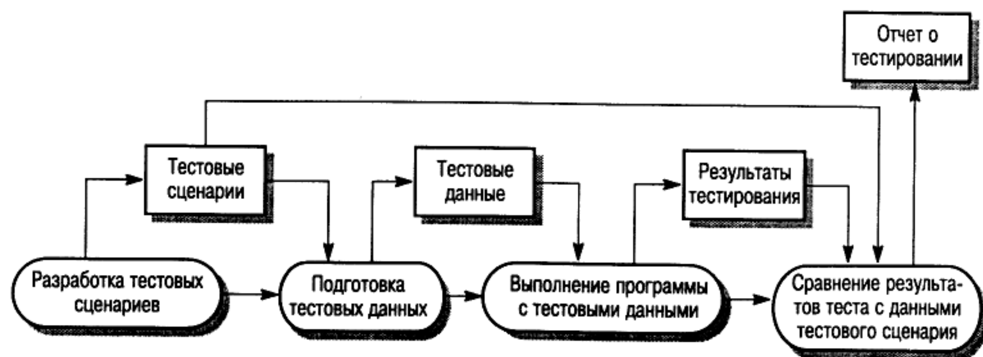

Рис. 15.2. Процесс тестирования дефектов

Полное тестирование, когда проверяются все возможные последовательности выполнения программы, нереально. Поэтому тестирование должно базироваться на некотором подмножестве всевозможных тестовых сценариев. Существуют различные методики выбора этого подмножества. Например, тестовые сценарии могут предусмотреть выполнение всех операторов в программе по меньшей мере один раз. Альтернативная методика отбора тестовых сценариев базируется на опыте использования подобных систем, в этом случае тестированию подвергаются только определенные средства и функции работающей системы, например следующие:

1. Все системные функции, доступные через меню.
2. Комбинации функций, доступные через меню (например, сложное форматирование текста).

3. Если в системе предполагается ввод пользователем каких-либо входных данных, тестируются функции с правильным и неправильным вводом данных.

Из опыта тестирования (и эксплуатации) больших программных продуктов, таких, как текстовые процессоры или электронные таблицы, вытекает, что необычные комбинации функций иногда могут вызывать ошибки, но наиболее часто используемые функции всегда работают правильно.

15.2. Тестирование методом черного ящика

Функциональное тестирование, или тестирование методом черного ящика базируется на том, что все тесты основываются на спецификации системы или ее компонентов. Система представляется как "черный ящик", поведение которого можно определить только посредством изучения ее входных и соответствующих выходных данных. Другое название этого метода – **функциональное тестирование** – связано с тем, что испытатель проверяет не реализацию ПО, а только его выполняемые функции.

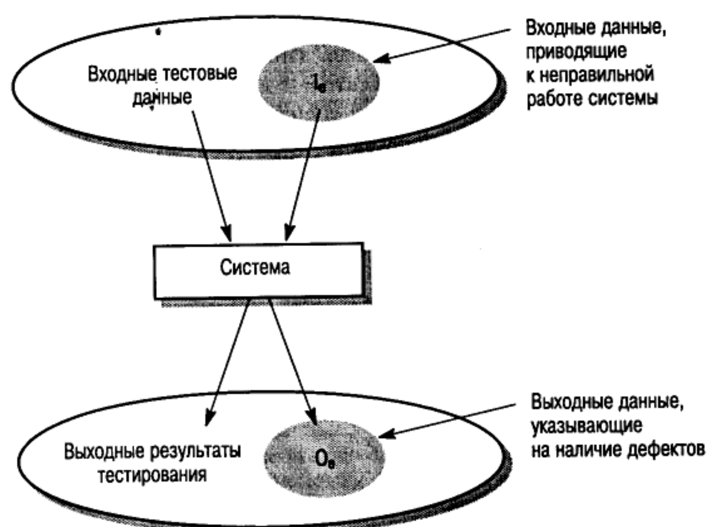

Рис. 15.3. Тестирование методом черного ящика

На рис. 15.3 показана модель системы, тестируемая методом черного ящика. Этот метод также применим к системам, организованным в виде

набора функций или объектов. Испытатель подставляет в компонент или систему входные данные и исследует соответствующие выходные данные. Если выходные данные не совпадают с предсказанными, значит, во время тестирования ПО **успешно** обнаружена ошибка (дефект).

Основная задача испытателя – подобрать такие входные данные, чтобы среди них с высокой вероятностью присутствовали элементы множества $I_ê$. Во многих случаях выбор тестовых данных основывается на предварительном опыте испытателя. Однако дополнительно к этим эвристическим знаниям можно также использовать систематический метод выбора входных данных, обсуждаемый в следующем разделе.

15.3. Области эквивалентности

Входные данные программ часто можно разбить на несколько классов. Входные данные, принадлежащие одному классу, имеют общие свойства, например это положительные числа, отрицательные числа, строки без пробелов и т.п. Обычно для всех данных из какого-либо класса поведение программы одинаково (эквивалентно). Из-за этого такие классы данных иногда называют областями эквивалентности [34]. Один из систематических методов обнаружения дефектов состоит в определении всех областей эквивалентности, обрабатываемых программой. Контрольные тесты разрабатываются так, чтобы входные и выходные данные лежали в пределах этих областей.

На рис. 15.4 каждая область эквивалентности изображена в виде эллипса. Области эквивалентности входных данных – это множества данных, все элементы которых обрабатываются одинаково. Области эквивалентности выходных данных – это данные на выходе программы, имеющие общие свойства, которые позволяют считать их отдельным классом. Корректные и некорректные входные данные также образуют две области эквивалентности.

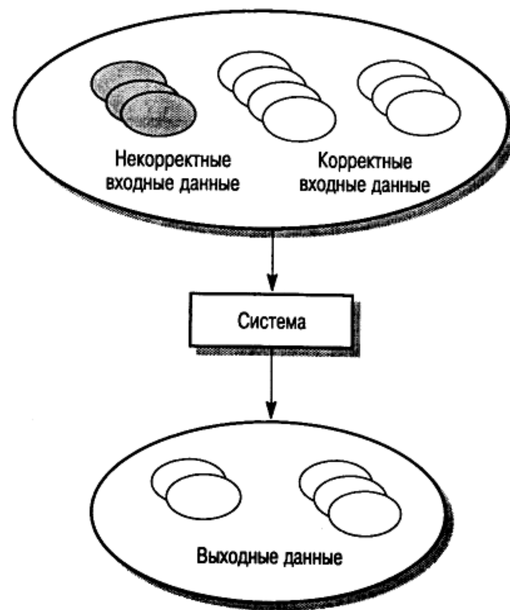

Рис. 15.4. Области эквивалентности

После определения областей эквивалентности для каждой из них подбираются тестовые данные. При выборе тестовых данных можно руководствоваться следующим полезным правилом: для тестов выбираются данные, расположенные на границе области эквивалентности, и отдельно данные, лежащие внутри этой области. Основная причина такого выбора данных заключается в следующем. В процессе разработки системы разработчики и программисты используют для тестов типичные значения входных данных, находящиеся внутри области эквивалентности. Граничные значения часто нетипичны (например, нулевое значение обрабатывается не так, как неотрицательные числа) и потому игнорируются программистами. Хотя чаще всего ошибки в программе возникают именно при обработке подобных нетипичных значений.

Области эквивалентности определяются на основании программной спецификации или документации пользователя и опыта испытателя, выбирающего классы значений входных данных, пригодные для обнаружения дефектов. Пусть, например, в спецификации программы указано, что в программу могут вводиться от 4 до 10 целых пятизначных

чисел. Области эквивалентности и возможные значения тестовых входных данных для этого примера показаны на рис. 15.5.

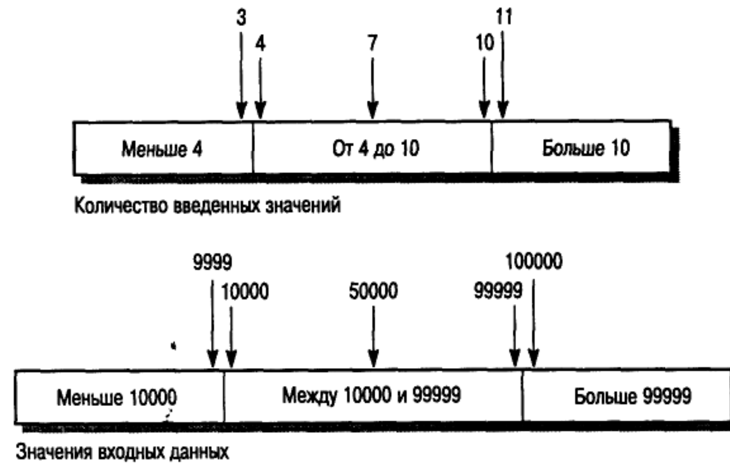

Рис. 15.5. Области эквивалентности

Покажем получение тестовых данных на примере спецификации упрощенной программы **Seach** (поиск), которая выполняет поиск заданного элемента **key** (ключ) в последовательности элементов. Программа возвращает номер позиции этого элемента в последовательности. Спецификация программы, представленная во врезке 15.1, содержит предусловие и постусловие. Предусловие указывает, что программа поиска не работает с пустыми последовательностями. Постусловие определяет, что если элемент, равный ключу, есть в последовательности, то переменная **Found** принимает значение **true** (истина). Индекс **L** обозначает позицию ключевого элемента в последовательности. Если элемент, равный ключу, в последовательности отсутствует, то этот индекс не определен.

Врезка 15.1. Спецификация программы поиска

Процедура: Seach (Key: ELEM; T: SEQ of ELEM;

Found: **in out** BOOLEAN; L: **in out** ELEM_INDEX);

Предусловие

-- в последовательности должен быть хотя бы один элемент

T’FIRST<= T’LAST

Постусловие

-- если элемент обнаружен под номером L

(Found **and** T(L) = Key)

или

-- если элемента нет в последовательности

(**not** Found **and**

not (**exists** i, T’FIRST >= i <= T’LAST, T(i) = Key))

Согласно данной спецификации, можно определить две очевидные области эквивалентности:

• последовательности входных данных, содержащие ключевой элемент (**Found = true**);

• последовательности входных данных, не содержащие ключевого элемента (**Found = false**).

При определении областей эквивалентности руководствуются различными правилами. Вот несколько правил выбора тестирующих последовательностей.

1. Тестирующая последовательность может состоять из одного элемента. Обычно считается, что последовательности состоят из нескольких элементов и программисты иногда закладывают такое представление в свои программы. Следовательно если ввести последовательность из одного элемента, программа может сработать неправильно.

2. Следует использовать в разных тестах различные последовательности, содержащие разное количество элементов. Это уменьшает вероятность того, что программа имеющая дефекты, случайно

выдаст правильные результаты в силу некоторых случайных свойств входных данных.

3. Следует использовать тестирующие последовательности, в которых ключевой элемент является первым, средним и последним элементом последовательности. Такой метод помогает выявить проблемы на границах областей эквивалентности.

Исходя из этих правил, можно определить еще две области эквивалентности входных данных для программы **Seach**.

• Входная последовательность состоит из одного элемента.

• Во входной последовательности больше одного элемента.

Эти области комбинируются с определенными ранее областями эквивалентности в результате будут получены области эквивалентности, представленные в табл. 15.1.

Таблица 15.1. Области эквивалентности для программы поиска

Последовательность	**Ключевой элемент**
Один элемент	Есть в последовательности
Один элемент	Нет в последовательности
Несколько элементов	Первый элемент последовательности
Несколько элементов	Последний элемент последовательности
Несколько элементов	Средний элемент последовательности
Несколько элементов	Нет в последовательности

Входная последовательность (T)	**Key**	**Выходные данные (Found, L)**
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

В табл. 15.1 также представлен набор возможных тестовых данных, взятых из этих областей. Если ключевого элемента нет в последовательности,

значение L не определено. При подборе тестовых данных применялось правило выбора последовательностей, согласно которому в разных тестах следует использовать последовательности разных размеров.

Множество вводимых значений, используемых для тестирования программы поиска, не является полным. Например, в работе программы может произойти сбой, если входная последовательность содержит элементы 1, 2, 3 или 4. Однако разумно предположить, что если не обнаружены дефекты при обработке одного элемента какого-либо класса эквивалентности, то тесты с любыми другими элементами этого класса также не выявят дефектов. Конечно, это не означает, что в программе отсутствуют дефекты. Возможно, не все области эквивалентности определены или определены неверно, или неправильно подобраны тестовые данные.

Здесь намеренно не рассматриваются тесты, которые проверяют порядок и тип используемых параметров. Возможные ошибки в использовании параметров лучше всего может выявить инспектирование программ или автоматический статический анализ. По этой же причине при тестировании не проверяется непредвиденное искажение данных на выходе программного компонента. Проблемы такого типа можно выявить во время инспектирования программ, которое рассматривается в главе 19.

15.4. Структурное тестирование

Метод структурного тестирования (рис. 15.6) предполагает создание тестов на основе структуры системы и ее реализации. Такой подход иногда называют тестированием методом "белого ящика", "стеклянного ящика" или "прозрачного ящика", чтобы отличать его от тестирования методом черного ящика.

Рис.15.6. Структурное тестирование

Как правило, структурное тестирование применяется к относительно небольшим программным элементам, например к подпрограммам или методам, ассоциированным с объектами. При таком подходе испытатель анализирует программный код и для получения тестовых данных использует знания о структуре компонента. Например, из анализа кода можно определить, сколько контрольных тестов нужно выполнить для того, чтобы в процессе тестирования все операторы выполнились по крайней мере один раз.

Знание алгоритма, используемого при реализации некоторой функции, можно применять для определения областей эквивалентности. В качестве примера возьмем спецификацию программы поиска (см. врезку 15.1), реализованную на языке Java в виде процедуры бинарного поиска (листинг 15.1). Здесь реализованы более строгие предусловия. Последовательность представлена в виде массива, массив должен быть упорядоченным, значение нижней границы массива должно быть меньше значения верхней границы.

Листинг 15.1. Процедура бинарного поиска

```
class BinSearch {
//Реализация функции бинарного поиска;
//на входе: упорядоченный массив объектов и ключевой элемент key
//Возвращает объект с двумя атрибутами:
//index - значение индекса массива
//found - логическая переменная,
//показывает, есть или нет ключевой элемент в массиве
//Если в массиве нет элемента, совпадающего с key, key = -1
```

```
public static void search (int key, int [] elemArray, Result r)
{
int bottom = 0;
int top = elemArray.length - 1;
int mid;
r.found = false; r.index = -1;
while ( bottom <= top )
{
mid = (top + bottom) / 2;
if (elemArray [mid] == key)
{
r.index = mid;
r.found = true;
return;
} //часть if
else
{
if (elemArray[mid] < key)
bottom = mid + 1;
else
top = mid - 1 ;
}
} //цикл while
} // поиск
} //BinSearch
```

Из текста программы видно, что во время ее выполнения область поиска разделяется на три части, каждая из которых является областью эквивалентности (рис. 15.7). При проверке программы в качестве тестовых данных необходимо взять последовательности с ключевыми элементами, расположенными на границах этих областей.

Рис. 15.7. Классы эквивалентности для бинарного поиска

Тестовые данные, представленные в табл. 15.7, необходимо изменить, поскольку элементы входного массива должны быть отсортированы в возрастающем порядке. Кроме того, следует добавить тестовые данные, где ключевой элемент расположен возле среднего элемента массива. Полученное множество тестовых данных для программы бинарного поиска представлено в табл. 15.2.

Таблица 15.2. Тестовые данные для программы бинарного поиска

Входной массив (T)	Ключ (Key)	Результат (Found, L)
17	17	true, 1
17	0;	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Тестирование ветвей

Это метод структурного тестирования, при котором проверяются все независимо выполняемые ветви компонента или программы. Если выполняются все независимые ветви, то и все операторы должны выполняться по крайней мере один раз. Более того, все условные операторы тестируются как с истинными, так и с ложными значениями условий. В объектно-ориентированных системах тестирование ветвей используется для тестирования методов, ассоциированных с объектами.

Количество ветвей в программе обычно пропорционально ее размеру. После интеграции программных модулей в систему, методы структурного

тестирования оказываются невыполнимыми. Поэтому методы тестирования ветвей, как правило, используются при тестировании отдельных программных элементов и модулей.

При тестировании ветвей не проверяются все возможные комбинации ветвей программы. Не считая самых тривиальных программных компонентов без циклов, подобная полная проверка компонента оказывается нереальной, так как в программах с циклами существует бесконечное число возможных комбинаций ветвей. В программе могут быть дефекты, которые проявляются только при определенных комбинациях ветвей, даже если все операторы программы протестированы (т.е. выполнились) хотя бы один раз.

Метод тестирования ветвей основывается на графе потоков управления программы. Этот граф представляет собой скелетную модель всех ветвей программы. Граф потоков управления состоит из узлов, соответствующих ветвлениям решений, и дуг, показывающих поток управления. Если в программе нет операторов безусловного перехода, то создание графа – достаточно простой процесс. При построении графа потоков все последовательные операторы (операторы присвоения, вызова процедур и ввода-вывода) можно проигнорировать. Каждое ветвление операторов условного перехода (if-then-else или case) представлено отдельной ветвью, а циклы обозначаются стрелками, концы которых замкнуты на узле с условием цикла. На рис. 15.8 показаны циклы и ветвления в графе потоков управления программы бинарного поиска.

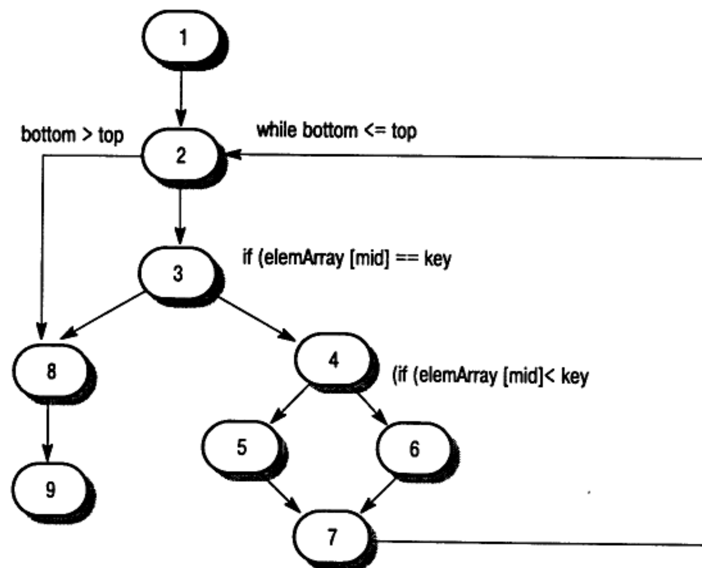

Рис. 15.8. Граф потоков управления программы бинарного поиска

Цель структурного тестирования – удостовериться, что каждая независимая ветвь программы выполняется хотя бы один раз. Независимая ветвь программы – это ветвь, которая проходит по крайней мере по одной новой дуге графа потоков. В терминах программы это означает ее выполнение при новых условиях. С помощью трассировки в графе потоков управления программы бинарного поиска можно выделить следующие независимых ветвей.

1, 2, 3, 8, 9

1, 2, 3, 4, 6, 7, 2

1, 2, 3, 4, 5, 7, 2

1, 2, 3, 4, 6, 7, 2, 8, 9

Если все эти ветви выполняются, можно быть уверенным в том, что, во-первых, каждый оператор выполняется по крайней мере один раз и, во-вторых, каждая ветвь выполняется при условиях, принимающих как истинные, так и ложные значения.

Количество независимых ветвей в программе можно определить, вычислив цикломатическое число графа потоков управления программы [232]. Цикломатическое число C любого связанного графа G вычисляется по формуле

C (G) = количество дуг – количество узлов + 2.

Для программ, не содержащих операторов безусловного перехода, значение цикломатического числа всегда больше количества проверяемых условий. В составных условиях, содержащих более одного логического оператора, следует учитывать каждый логический оператор. Например, если в программе шесть операторов **if** и один цикл **while**, то цикломатическое число равно 8. Если одно условное выражение является составным выражением с двумя логическими операторами (объединенными операторами and или or), то цикломатическое число будет равно 10. Цикломатическое число программы бинарного поиска равно 4.

После определения количества независимых ветвей в программе путем вычисления цикломатического числа разрабатываются контрольные тесты для проверки каждой ветви. Минимальное количество тестов, требующееся для проверки всех ветвей программы, равно цикломатическому числу.

Проектирование контрольных тестов для программы бинарного поиска не вызывает затруднений. Однако, если программы имеют сложную структуру ветвлений, трудно предсказать, как будет выполняться какой-либо отдельный контрольный тест. В таких случаях используется динамический анализатор программ для составления рабочего профиля программы.

Динамические анализаторы программ – это инструментальные средства, которые работают совместно с компиляторами. Во время компилирования в сгенерированный код добавляются дополнительные инструкции, подсчитывающие, сколько раз выполняется каждый оператор программы. Чтобы при выполнении отдельных контрольных тестов увидеть, какие ветви в программе выполнялись, а какие нет, распечатывается рабочий профиль программы, где видны непроверенные участки.

15.5. Тестирование сборки

После того как протестированы все отдельные программные компоненты, выполняется сборка системы, в результате чего создается

частичная или полная система. Процесс интеграции системы включает сборку и тестирования полученной системы, в ходе которого выявляются проблемы, возникающие при взаимодействии компонентов. Тесты, проверяющие сборку системы, должны разрабатываться на основе системной спецификации, причем тестирование сборки следует начинать сразу после создания работоспособных версий компонентов системы.

Во время тестирования сборки возникает проблема локализации выявленных ошибок. Между компонентами системы существуют сложные взаимоотношения, и при обнаружении аномальных выходных данных бывает трудно установить источник ошибки. Чтобы облегчить локализацию ошибок, следует использовать пошаговый метод сборки и тестирования системы. Сначала следует создать минимальную конфигурацию системы и ее протестировать. Затем в минимальную конфигурацию нужно добавить новые компоненты и снова протестировать, и так далее до полной сборки системы.

В примере на рис. 15.9 последовательность тестов Tl, T2 и T3 сначала выполняется в системе, состоящей из модулей А и В (минимальная конфигурация системы). Если во время тестирования обнаружены дефекты, они исправляются. Затем в систему добавляется модуль С. Тесты Tl, T2 и T3 повторяются, чтобы убедиться, что в новой системе нет никаких неожиданных взаимодействий между модулями А и В. Если в ходе тестирования появились какие-то проблемы, то, вероятно, они возникли во взаимодействиях с новым модулем С. Источник проблемы локализован, таким образом упрощается определение дефекта и его исправление. Затем система запускается с тестами Т4. На последнем шаге добавляется модуль D и система тестируется еще раз выполняемыми ранее тестами, а затем новыми тестами Т5.

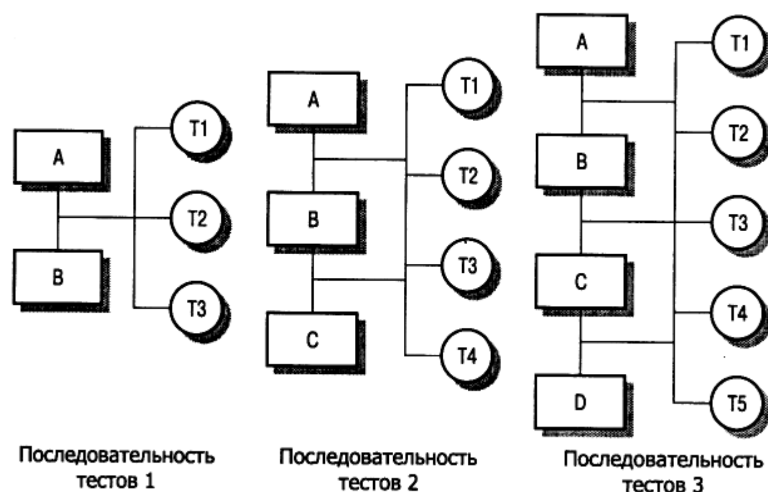

Рис. 15.9. Тестирование сборки

Конечно, на практике редко встречаются такие простые модели. Функции системы могут быть реализованы в нескольких компонентах. Тестирование новой функции, таким образом, требует интеграции сразу нескольких компонентов. В этом случае тестирование может выявить ошибки во взаимодействиях между этими компонентами и другими частями системы. Исправление ошибок может оказаться сложным, так как в данном случае ошибки влияют на целую группу компонентов, реализующих конкретную функцию. Более того, при интеграции нового компонента может измениться структура взаимосвязей между уже протестированными компонентами. Вследствие этого могут выявиться ошибки, которые не были выявлены при тестировании более простой конфигурации.

15.6. Нисходящее и восходящее тестирование

Методики нисходящего и восходящего тестирования (рис. 15.10) отражают разные подходы к системной интеграции. При нисходящей интеграции компоненты высокого уровня интегрируются и тестируются еще до окончания их проектирования и реализации. При восходящей интеграции перед разработкой компонентов более высокого уровня сначала интегрируются и тестируются компоненты нижнего уровня.

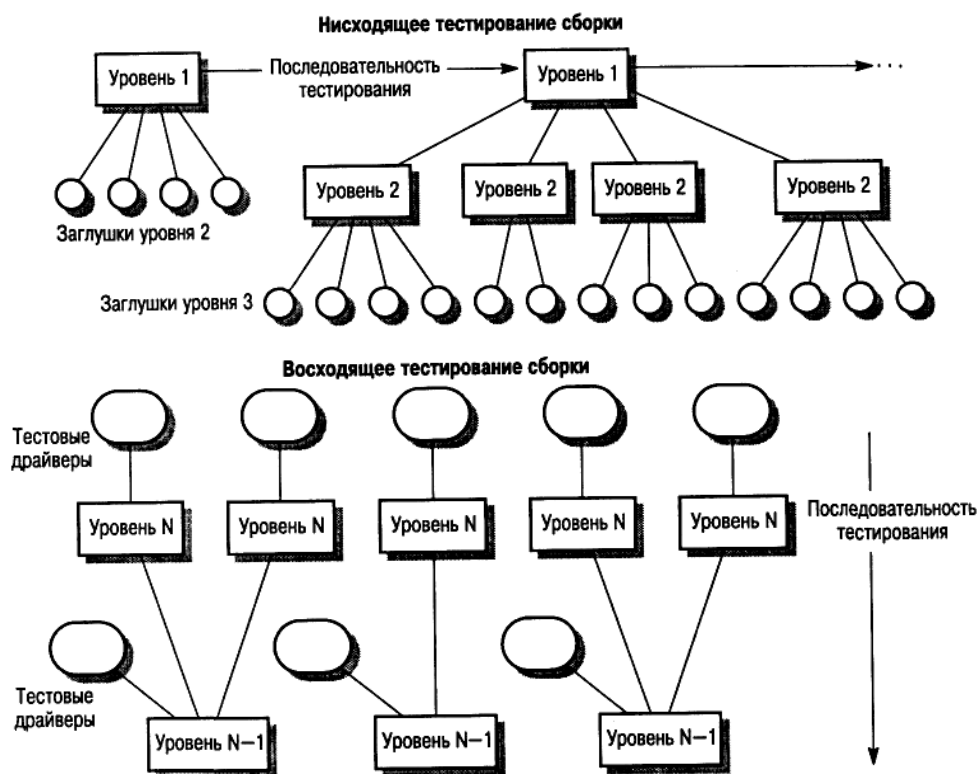

Рис. 15.10. Нисходящее и восходящее тестирование сборки

Нисходящее тестирование является неотъемлемой частью процесса нисходящей разработки систем, при котором сначала разрабатываются компоненты верхнего уровня, а затем компоненты, находящиеся на нижних уровнях иерархии. Программу можно представить в виде одного абстрактного компонента с субкомпонентами, являющимися заглушками. Заглушки имеют такой же интерфейс, что и компонент, но с ограниченной функциональностью. После того как компонент верхнего уровня запрограммирован и протестирован, таким же образом реализуются и тестируются его субкомпоненты. Процесс продолжается до тех пор, пока не будут реализованы компоненты самого нижнего уровня. Затем вся система тестируется целиком.

При восходящем тестировании, наоборот, сначала интегрируются и тестируются модули, расположенные на более низких уровнях иерархии. Затем выполняется сборка и тестирование модулей, расположенных на верхнем уровне иерархии, и так до тех пор, пока не будет протестирован последний модуль. При таком подходе не требуется наличие законченного архитектурного проекта системы, и поэтому он может начинаться на раннем

этапе процесса разработки. Обычно такой подход применяется тогда, когда в системе есть повторно используемые компоненты или модифицированные компоненты из других систем.

Нисходящее и восходящее тестирование можно сравнить по четырем направлениям:

1. **Верификация и аттестация системной архитектуры**. При нисходящем тестировании больше возможностей выявить ошибки в архитектуре системы на раннем этапе процесса разработки. Обычно это структурные ошибки, раннее выявление которых предполагает их исправление без дополнительных затрат. При восходящем тестировании структура высокого уровня не утверждается вплоть до последнего этапа разработки системы.

2. **Демонстрация системы**. При нисходящей разработке незаконченная система вполне пригодна для работы уже на ранних этапах разработки. Этот факт является важным психологическим стимулом использования нисходящей модели разработки систем, поскольку демонстрирует осуществимость управления системой. Аттестация проводится в начале процесса тестирования путем создания демонстрационной версии системы. Но если система создается из повторно используемых компонентов, то и при восходящей разработке также можно создать ее демонстрационную версию.

3. **Реализация тестов**. Нисходящее тестирование сложно реализовать, так как необходимо моделировать программы-заглушки нижних уровней. Программы-заглушки могут быть упрощенными версиями представляемых компонентов. При восходящем тестировании для того, чтобы использовать компоненты нижних уровней, необходимо разработать тестовые драйверы, которые эмулируют окружение компонента в процессе тестирования.

4. **Наблюдение за ходом испытаний**. При нисходящем и восходящем тестировании могут возникать проблемы, связанные с

наблюдениями за ходом тестирования. В большинстве систем, разрабатываемых сверху вниз, более верхние уровни системы, которые реализованы первыми, не генерируют выходные данные, однако для проверки этих уровней нужны какие-либо выходные результаты. Испытатель должен создать искусственную среду для генерации результатов теста. При восходящем тестировании также может возникнуть необходимость в создании искусственной среды (тестовых драйверов) для исследования компонентов нижних уровней.

На практике при разработке и тестировании систем чаще всего используется композиция восходящих и нисходящих методов. Разные сроки разработки для разных частей системы предполагают, что группа, проводящая тестирование и интеграцию, должна работать с каким-либо готовым компонентом. Поэтому во время процесса тестирования сборки в любом случае необходимо разрабатывать как заглушки, так и тестовые драйверы.

15.7. Тестирование интерфейсов

Как правило, тестирование интерфейса выполняется в тех случаях, когда модули или подсистемы интегрируются в большие системы. Каждый **модуль** или подсистема имеет заданный интерфейс, который вызывается другими компонентами системы. **Цель тестирования интерфейса** – выявить ошибки, возникающие в системе вследствие ошибок в интерфейсах или неправильных предположений об интерфейсах.

Схема тестирования интерфейса показана на рис. 15.11. Стрелки в верхней части схемы означают, что контрольные тесты применяются не к отдельным компонентам, а к подсистемам, полученным в результате комбинирования этих компонентов.

Данный тип тестирования особенно важен в объектно-ориентированном проектировании, в частности при повторном

использовании объектов и классов объектов. Объекты в значительной степени определяются с помощью интерфейсов и могут повторно использоваться в различных комбинациях с разными объектами и в разных системах. Во время тестирования отдельных объектов невозможно выявить ошибки интерфейса, так как они являются скорее результатом взаимодействия между объектами, чем изолированного поведения одного объекта.

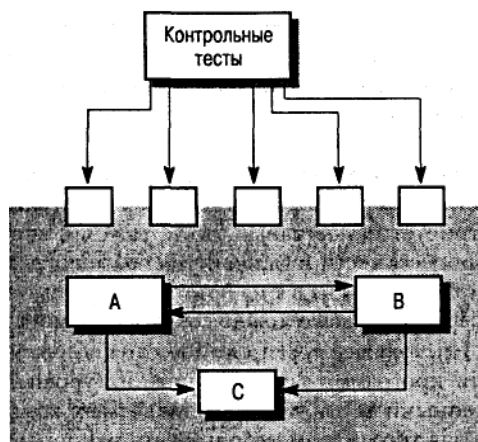

Рис. 15.11. Тестирование интерфейсов

Между компонентами программы могут быть разные типы интерфейсов и соответственно разные типы ошибок интерфейсов.

1. **Параметрические интерфейсы**. Интерфейсы, в которых ссылки на данные и иногда функции передаются в виде параметров от одного компонента к другому.

2. **Интерфейсы разделяемой памяти**. Интерфейсы, в которых какой-либо блок памяти совместно используется разными подсистемами. Одна подсистема помещает данные в память, а другие подсистемы используют эти данные.

3. **Процедурные интерфейсы.** Интерфейсы, в которых одна подсистема инкапсулирует набор процедур, вызываемых из других подсистем. Такой тип интерфейса имеют объекты и абстрактные типы данных.

4. **Интерфейсы передачи сообщений**. Интерфейсы, в которых одна подсистема запрашивает сервис у другой подсистемы посредством

передачи ей сообщения. Ответное сообщение содержит, результаты выполнения сервиса. Некоторые объектно-ориентированные системы имеют такой тип интерфейсов; например, так работают системы клиент/сервер.

Ошибки в интерфейсах являются наиболее распространенными типами ошибок в сложных системах [226] и делятся на три класса.

- **Неправильное использование интерфейсов**. Компонент вызывает другой компонент и совершает ошибку при использовании его интерфейса. Данный тип ошибки особенно распространен в параметрических интерфейсах; например, параметры могут иметь неправильный тип, следовать в неправильном порядке или же иметь неверное количество параметров.
- **Неправильное понимание интерфейсов.** Вызывающий компонент, в который заложена неправильная интерпретация спецификации интерфейса вызываемого компонента, предполагает определенное поведение этого компонента. Если поведение вызываемого компонента не совпадает с ожидаемым, поведение вызывающего компонента становится непредсказуемым. Например, если программа бинарного поиска вызывается для поиска заданного элемента в неупорядоченном массиве, то в работе программы произойдет сбой.
- **Ошибки синхронизации**. Такие ошибки встречаются в системах реального времени, где используются интерфейсы разделяемой памяти или передачи сообщений. Подсистема – производитель данных и подсистема – потребитель данных могут работать с разной скоростью. Если при проектировании интерфейса не учитывать этот фактор, потребитель может, например, получить доступ к устаревшим данным, потому что производитель к тому моменту еще не успел обновить совместно используемые данные.

Тестирование дефектов интерфейсов сложно, поскольку некоторые ошибки могут проявиться только в необычных условиях. Например, пусть некий объект реализует очередь в виде структуры списка фиксированного размера. Вызывающий его объект при вводе очередного элемента не

проверяет переполнение очереди, так как предполагает, что очередь реализована как структура неограниченного размера. Такую ситуацию можно обнаружить только во время выполнения специальных тестов: специально вызывается переполнение очереди, которое приводит к непредсказуемому поведению объекта.

Другая проблема может возникнуть из-за взаимодействий между ошибками в разных программных модулях или объектах. Ошибки в одном объекте можно выявить только тогда, когда поведение другого объекта становится непредсказуемым. Например, для получения сервиса один объект вызывает другой объект и полагает, что полученный ответ правильный. Если объект неправильно понимает вычисленные значения, возвращаемое значение может быть достоверным, но неправильным. Такие ошибки можно выявить только тогда, когда оказываются неправильными дальнейшие вычисления.

Вот несколько общих правил тестирования интерфейсов.

1. Просмотрите тестируемый код и составьте список всех вызовов, направленных к внешним компонентам. Разработайте такие наборы тестовых данных, при которых параметры, передаваемые внешним компонентам, принимают крайние значения из диапазонов их допустимых значений. Использование экстремальных значений параметров с высокой вероятностью обнаруживает несоответствия в интерфейсах.

2. Если между интерфейсами передаются указатели, всегда тестируйте интерфейс с нулевыми параметрами указателя.

3. При вызове компонента через процедурный интерфейс используйте тесты, вызывающие сбой в работе компонента. Одна из наиболее распространенных причин ошибок в интерфейсе – неправильное понимание спецификации компонентов.

4. В системах передачи сообщений используйте тесты с нагрузкой, которые рассматриваются в следующем разделе. Разрабатывайте тесты, генерирующие в несколько раз большее количество сообщений, чем будет в

обычной работе системы. Эти же тесты позволяют обнаружить проблемы синхронизации.

5. При взаимодействии нескольких компонентов через разделяемую память разрабатывайте тесты, которые изменяют порядок активизации компонентов. С помощью таких тестов можно выявить сделанные программистом неявные предположения о порядке использования компонентами разделяемых данных.

Обычно статические методы тестирования более рентабельны, чем специальное тестирование интерфейсов. В языках со строгим контролем типов, например Java, многие ошибки интерфейсов помогает обнаружить компилятор. В языках со слабым контролем типов (например, С) ошибки интерфейса может выявить статический анализатор, такой как LINT (см. главу 19). Кроме того, при инспектировании программ можно сосредоточиться именно на проверке интерфейсов компонентов.

15.8. Инструментальные средства тестирования

Тестирование – дорогой и трудоемкий этап разработки программных систем. Поэтому создан широкий спектр инструментальных средств для поддержки процесса тестирования, которые значительно сокращают расходы на него.

На рис. 15.14 показаны возможные инструментальные средства тестирования и отношения между ними. Перечислим их.

1. **Организатор тестов**. Управляет выполнением тестов. Он отслеживает тестовые данные, ожидаемые результаты и тестируемые функции программы.

2. **Генератор тестовых данных**. Генерирует тестовые данные для тестируемой программы. Он может выбирать тестовые данные из базы

данных или использовать специальные шаблоны для генерации случайных данных необходимого вида.

3. **Оракул.** Генерирует ожидаемые результаты тестов. В качестве оракулов могут выступать предыдущие версии программы или исследуемого объекта. При тестировании параллельно запускаются оракул и тестируемая программа и сравниваются результаты их выполнения.

4. **Компаратор файлов.** Сравнивает результаты тестирования с результатами предыдущего тестирования и составляет отчет об обнаруженных различиях. Компараторы особенно важны при сравнении различных версий программы. Различия в результатах указывают на возможные проблемы, существующие в новой версии системы.

5. **Генератор отчетов**. Формирует отчеты по результатам проведения тестов.

6. **Динамический анализатор**. Добавляет в программу код, который подсчитывает, сколько раз выполняется каждый оператор. После запуска теста создает исполняемый профиль, в котором показано, сколько раз в программе выполняется каждый оператор.

7. **Имитатор**. Существует несколько типов имитаторов. Целевые имитаторы моделируют машину, на которой будет выполняться программа. Имитатор пользовательского интерфейса – это программа, управляемая сценариями, которая моделирует взаимодействия с интерфейсом пользователя. Имитатор ввода-вывода генерирует последовательности повторяющихся транзакций.

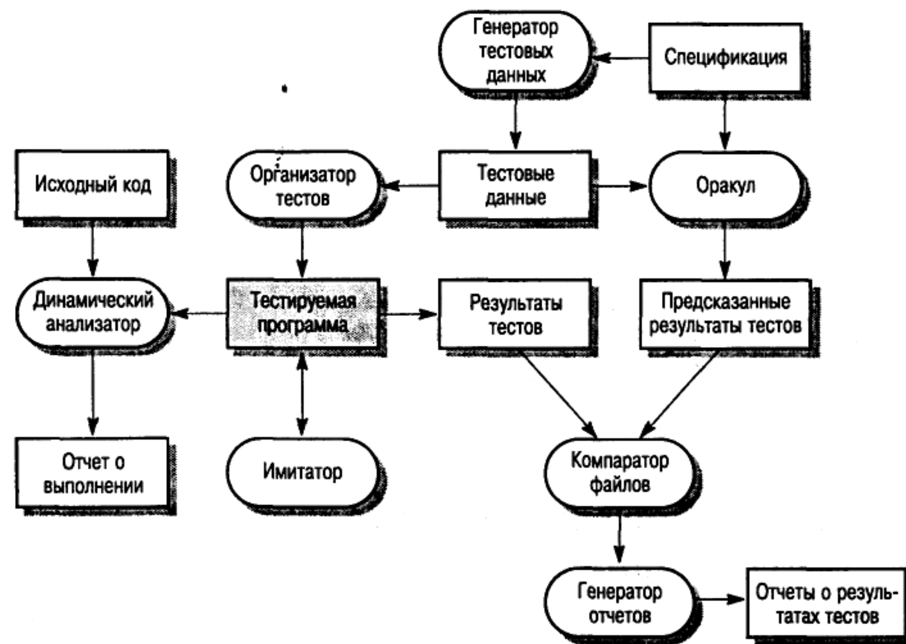

Рис. 15.14. Инструментальные средства тестирования

Требования, предъявляемые к процессу тестирования больших систем, зависят от типа разрабатываемого приложения. Поэтому инструментальные средства тестирования неизменно приходится адаптировать к процессу тестирования конкретной системы.

Для создания полного комплекса инструментального средства тестирования, как правило, требуется много сил и времени. Весь набор инструментальных средств, показанных на рис. 15.14, используется только при тестировании больших систем. Для таких систем полная стоимость тестирования может достигать 50% от всей стоимости разработки системы. Вот почему выгодно инвестировать разработку высококачественных и производительных CASE-средств тестирования.

Контрольные вопросы:

1) Что такое тестовые сценарии?
2) Перечислите правила выбора тестирующих последовательностей?
3) Что такое динамические анализаторы программ?
4) По каким четырем направлениям можно сравнить нисходящее и восходящее тестирование?
5) Перечислите общие правила тестирования интерфейсов.

Ключевые слова: *тестовые сценарии функциональное тестирование, основная задача испытателя, динамические анализаторы программ, верификация и аттестация системной архитектуры, демонстрация системы, реализация тестов, наблюдение за ходом испытаний, параметрические интерфейсы, интерфейсы разделяемой памяти, процедурные интерфейсы, интерфейсы передачи сообщений, неправильное использование интерфейсов, неправильное понимание интерфейсов, ошибки синхронизации, тестирование, организатор тестов, генератор тестовых данных, оракул, компаратор файлов, генератор отчетов, динамический анализатор, имитатор.*

Keywords: *functional testing test cases, the main objective test, dynamic analyzers programs, verification and certification of the system architecture, the demonstration system, the implementation of test, monitoring the progress of the test, parametric interfaces, shared memory, procedural interfaces, messaging misuse interfaces , misunderstanding interfaces, synchronization errors, testing, organizer test generator test data, oracle comparator files, report generator, dynamic analyzer, simulator.*

Kalit so'zlar: *test ssenariylari, funksional test, sinovchining asosiy masalasi, dinamik analizatorlar, tizim demonstratsiyasi, testlar realizatsiyasi, sinovlar kuzatish, parametrik interfeyslar, bo'limli xotira interfeyslari, protsedura interfeyslari, xabarlar yuborish interfeyslari, sinxronizatsiya hatoliklari, test ma'lumotlar generatori, orakul, dinamik analizator, simulyator.*

Упражнения

1. Обсудите различия между тестированием методом черного ящика и структурным тестированием. Подумайте, каким образом можно совместно использовать эти методы в процессе тестирования дефектов.

2. Какие проблемы тестирования могут возникнуть в программах, которые обрабатывают как очень большие, так и очень малые числа?

3. Разработайте набор тестовых данных для следующих компонентов:

- программа сортировки массивов целых чисел;
- программа, которая вычисляет количество символов (отличных от пробелов) в текстовых строках;
- программа, которая проверяет текстовые строки и заменяет последовательности пробелов одним пробелом;
- объект, реализующий символьные строки разной длины. Среди операций, ассоциированных с этим объектом, должны быть операция конкатенации, операция определения длины строки и операция выбора подстроки.

4. Напишите код для первых трех перечисленных выше программ, используя любой язык программирования. Для каждой программы рассчитайте цикломатическое число.

5. На примере небольшой программы покажите, почему практически невозможно полностью протестировать программу.

6. Для проверки созданных вами программ, разработайте контрольные тесты в дополнение к тем, которые уже были рассмотрены. Выявил ли анализ кода упущения в первоначальных наборах тестовых данных?

7. Реализуйте (на языке Java или C++) класс объектов SYMBOL_TABLE, который будет использоваться как часть системы компиляции. Этот класс должен иметь следующие методы: добавление имени и типа данных в таблицу идентификаторов, удаление имени, изменение информации, связанной с именем, и поиск по таблице. Организуйте инспектирование кода этого объекта (см. главу 19) и сделайте подсчет обнаруженных ошибок. Протестируйте объект методом черного ящика и сравните ошибки, выявленные при тестировании и при инспектировании.

8. Объясните, почему методы нисходящего и восходящего тестирования не подходят для объектно-ориентированных систем.

9. Создайте сценарии тестирования состояний микроволновой печи, модель состояний которой показана на рис. 7.5.

ГЛАВА 16. УПРАВЛЕНИЕ ПЕРСОНАЛОМ

Люди, работающие в компаниях по разработке ПО, являются их самым ценным "активом". Именно они представляют интеллектуальный капитал, и от менеджеров по разработке ПО зависит, получит ли компания наилучшие из возможных дивиденды от инвестиций в человеческие ресурсы. В успешно развивающихся компаниях и экономических структурах это достигается в том случае, если организация уважает своих сотрудников. Круг выполняемых ими обязанностей и уровень вознаграждения должны соответствовать их умению, которое, в свою очередь, зависит от квалификации.

Таким образом, **эффективный менеджмент** – это эффективное управление персоналом компании. **Менеджеры** – руководители проектов должны решать как технические, так и далекие от технических вопросы, используя при этом членов команды с оптимальной эффективностью. Они должны уметь мотивировать действия людей, организовывать их работу и гарантировать ее качественное исполнение. Слабый менеджмент персонала – одна из основных составляющих провала программных проектов.

В своих рассуждениях я опираюсь в основном не на принятые в наше время модные модели теории менеджмента, а на два фактора – познавательный и социальный. Процесс разработки ПО включает в себя познавательный и социальный процессы, поэтому именно эти факторы я считаю самыми важными в углублении понимания того, как создаются программы. Если у менеджеров есть понятие об этих основах, тогда они лучше управляют людьми, получая наибольшую отдачу.

16.1. Пределы мышления

Человеческие возможности достаточно разнообразны, и это, прежде всего, зависит от уровня интеллекта, образования и личного опыта, однако у

всех нас есть нечто общее: в своей умственной деятельности мы подчиняемся определенным ограничениям. Эти ограничения не что иное, как следствие того, каким образом наш мозг сохраняет и обрабатывает полученную информацию. Нам вовсе не обязательно подробно изучать процесс обработки информации об окружающем мире человеческим мозгом. Однако я считаю, что знание границ нашего мышления имеет исключительную важность. Именно эти знания помогут выяснить, почему некоторые технологии разработки программного обеспечения достигают высокой эффективности, а также проникнуть вглубь взаимоотношений между членами команды разработчиков ПО.

Оценка стоимости программного продукта

В главе 4 был описан процесс планирования программного проекта. Под этим подразумевается разбивка проекта на ряд этапов, выполняемых параллельно или последовательно. Ранее процесс планирования рассматривался в аспекте определения этих этапов, их взаимозависимости и распределения специалистов по работам, выполняемым на каждом этапе.

В этой главе я вернусь к общей проблеме оценки затрат и времени, необходимых для выполнения определенных этапов проекта. Менеджеры проекта должны научиться давать правильные ответы на следующие вопросы.

1. Какие затраты необходимы для выполнения этапа?
2. Сколько это займет времени?
3. Какова стоимость выполнения данного этапа?

Оценка стоимости проекта и планирование графика работ проводятся параллельно. Однако некоторые предварительные расчеты должны быть выполнены на ранней стадии, еще до начала разработки точного плана проекта. Такие расчеты необходимы для утверждения бюджета проекта или для выставления цены заказчику.

Как только проект начинает действовать, все расчеты должны регулярно обновляться. Это помогает планировать работу и содействует

эффективному использованию средств. Если фактические расходы значительно превышают планируемые, менеджеру необходимо предпринять какие-либо действия. Это может быть перечисление дополнительных средств на проект либо изменение будущих этапов работ в соответствии с фактическим бюджетом.

Обычно для оценки проекта по разработке программного обеспечения используются три параметра.

- Стоимость аппаратных средств и программного обеспечения, включая их обслуживание.
- Расходы на командировки и обучение.
- Расходы на персонал, в основном на привлечение со стороны специалистов по программному обеспечению.

В большинстве проектов доминируют расходы на персонал. Компьютеры, имеющие достаточно мощности для разработки программного продукта, в наше время относительно дешевые. Значительными могут быть затраты на командировки, если проект разрабатывается в разных местах, однако для большинства проектов они все же не очень существенны. Более того, расходы на командировки можно сократить, используя вместо них электронную почту, факс или телеконференции.

Расходы на персонал - это не только оплата труда работников. В них могут включаться накладные расходы, т.е. все расходы, которые касаются работы организации, деленные на количество работающего персонала. Таким образом, общая сумма расходов на персонал состоит из нескольких статей расходов.

1. Расходы на содержание, отопление и освещение офисов.
2. Расходы на содержание вспомогательного персонала – бухгалтеров, секретарей, уборщиц и технического персонала.
3. Расходы на содержание компьютерной сети и средств связи.
4. Расходы на централизованные услуги – библиотеки, места отдыха и развлечения и т.д.

5. Расходы на социальное обеспечение и выплаты служащим (например, пенсии и медицинская страховка).

Обычно накладные расходы приравниваются к удвоенной зарплате программиста, в зависимости от размера компании и расходов на ее содержание. Например, если специалист по программному обеспечению получает 90 000 долларов в год, расходы организации на этот год составляют сумму 180 000 долларов, или 15 000 долларов в месяц.

Оценка стоимости должна быть объективной, чтобы дать компании-разработчику достаточно точный прогноз себестоимости проекта. Если себестоимость рассчитывается для включения в коммерческое предложение заказчику, следует принять решение о том, какую цену назначить за проект. Традиционно в цену продукта включают издержки производства плюс предлагаемая прибыль. Однако определить соотношение себестоимости проекта и цены, выставляемой заказчику, не всегда просто.

На определение цены программного продукта могут повлиять организационные, экономические, политические и коммерческие соображения. Эти факторы показаны в табл. 16.1. Таким образом, взаимоотношения цены и себестоимости совсем не так просты, как может показаться. Кроме того, необходимо учитывать глобальные цели организации, поэтому назначением цены за программный продукт вместе с менеджерами проектов занимается и старший руководящий состав компании.

Таблица 16.1. Факторы, влияющие на стоимость программного продукта

Фактор	Описание
Возможности рынка ПО	Организация-разработчик может выставить низкие цены на программный продукт из-за намерения переместиться в другой сегмент рынка ПО. Даже если организация примет более низкую прибыль в первом проекте, это все равно может привести к более высоким доходам в будущем, поскольку полученный опыт позволит заниматься разработкой подобных программных продуктов и в дальнейшем

Невозможно учесть все факторы, влияющие на стоимость	Если организация примет фиксированную величину стоимости, издержки производства могут возрасти из-за непредвиденных расходов
Условия контракта	Заказчик может позволить разработчику сохранить за собой право владения программным кодом с последующим его использованием в других проектах. При этом назначенная цена может быть ниже, чем в том случае, если право на программный код передано заказчику
Изменение требований	При изменении требований к ПО организация может снизить цену с тем, чтобы выиграть контракт. Если контракт уже заключен, за изменение требований можно назначить дополнительную цену
Финансовая стабильность	Фирмы, испытывающие финансовые затруднения, для получения заказа могут снизить цены на свои разработки. Как правило, лучше сегодня получить более низкую прибыль или даже работать на уровне самоокупаемости, чем обанкротиться в будущем

16.2. Производительность

Производительность в промышленности обычно измеряется путем деления количества единиц выпущенной продукции на количество человеко-часов, необходимых для их производства. Однако в области разработки ПО любая задача имеет несколько вариантов решения, каждый со своими особенностями. Одно проектное решение отличается эффективным способом выполнения, другое имеет программный код, который легко читается либо удобен в эксплуатации. Поэтому не имеет смысла сравнивать уровни производительности при разработке решений с разными качественными характеристиками.

Несмотря на это, менеджеры могут оценить производительность самих специалистов по разработке программного обеспечения. Это понадобится при оценивании проекта и определении эффективности усовершенствования

процесса и технологии разработки ПО. Оценка производительности в этом случае будет основана на измерении количественных показателей программных продуктов и последующем делении их на количество усилий, затраченных на разработку этих продуктов. При этом можно использовать два типа показателей.

1. **Показатель размера**. Зависит от размера выходного результата очередного этапа работ. Наиболее часто применяемым критерием такого типа является количество строк разработанного программного кода. За аналогичный показатель также можно взять количество инструкций объектной программы или количество страниц системной документации.

2. **Функциональный показатель**. Зависит от функциональных возможностей программного продукта в целом. Производительность в этом случае выражается количеством полезных выполняемых функций, разработанных в определенный отрезок времени. К наиболее распространенным показателям этого типа относится количество функциональных и объектных точек.

Количество строк программного кода за человеко-месяц – наиболее популярный критерий оценки производительности. Он определяется путем деления общего количества строк кода на количество времени в человеко-месяцах, которое потребуется для завершения проекта. Это время, потраченное на анализ, проектирование, кодирование, тестирование и разработку документации программного продукта.

Данный подход впервые появился еще во время массового использования таких языков программирования, как FORTRAN, язык ассемблера и COBOL. Затем программы переводились на перфокарты, каждая из которых содержала по одному оператору. Таким образом, было легко подсчитывать количество строк кода. Оно соответствовало количеству перфокарт в колоде. Однако программы, написанные на языках типа Java или C++, состоят из описаний, выполняемых операторов и комментариев. Они также могут включать макрокоманды, которые состоят из нескольких строк

кода. С другой стороны, в одной строке может находиться не один, а несколько операторов. Таким образом, соотношения между операторами и строками в листинге могут быть достаточно сложными.

Одни методы подсчета строк основываются только на выполняемых операторах; другие подсчитывают выполняемые операторы и объявления данных; третьи ведут подсчет всех строк программы, независимо от их содержимого. Были предложены определенные стандарты подсчета строк в различных языках программирования [269], однако они не приобрели широкой популярности. Поэтому практически невозможно сравнивать производительность различных компаний-разработчиков, если только все они не используют один и тот же метод подсчета строк кода.

Также может ввести в заблуждение и сравнение производительности программистов, пишущих на разных языках программирования. Чем выразительнее язык, тем ниже производительность. Это "странное" утверждение объясняется тем, что при оценивании производительности создания ПО берется во внимание вся деятельность по разработке программного продукта, тогда как данная система измерения основывается лишь на оценивании процесса программирования.

Для примера возьмем систему, которая должна быть написана с помощью кода ассемблера (5000 строк) или с помощью языка высокого уровня (1500 строк). Время выполнения различных этапов создания ПО показано в табл. 16.2. Специалист, программирующий на языке ассемблера, будет иметь производительность 714 строк в месяц, а у программиста, работающего с языком высокого уровня, производительность будет в два раза ниже – 300 строк в месяц. И это несмотря на то, что программирование на языке высокого уровня стоит дешевле и занимает меньше времени.

Таблица 16.2. Время выполнения этапов разработки программной системы

	Анализ (недели)	**Проектирование**	**Кодирование**	**Тестирование**	**Документирование**

		(недели)	(недели)	(недели)	(недели)
Язык ассемблера	3	5	8	10	2
Язык высокого уровня	3	5	8	6	2

	Размер (строки кода)	Затраченное время (недели)	Производительность (строк в месяц)
Язык ассемблера	5000	28	714
Язык высокого уровня	1500	20	300

Альтернативой показателю размера при оценивании производительности может служить функциональный показатель. В этом случае можно избежать тех "аномалий" в оценке производительности, которые встречаются при использовании показателя размера, так как функциональность не зависит от языка программирования. В статье [228] дается краткое описание и сравнение способов оценки производительности, основанных на функциональных показателях.

Одним из наиболее распространенных методов этого типа является метод функциональных точек. Впервые он был предложен в работе [6], а затем его усовершенствованный вариант был представлен в статье [7]. Функциональные точки не зависят от применяемого языка программирования, благодаря чему появилась возможность сравнения производительности разработки программных систем, написанных на различных языках программирования. Критерием оценки производительности выступает количество функциональных точек, созданных за человеко-месяц. Функциональная точка – это не какая-то отдельная характеристика программного продукта, а целая комбинация его свойств. Подсчет общего количества функциональных точек в программе проводится путем измерения или оценивания следующих свойств программы:

• Интенсивность использования ввода и вывода внешних данных.

• Взаимодействие системы с пользователем.

• Внешние интерфейсы.

• Файлы, используемые системой.

Сложность каждого из указанных критериев оценивается отдельно, в результате каждому критерию присваивается определенная весовая величина, которая может колебаться от 3 (для простого ввода данных) до 15 баллов за использование сложных внутренних файлов. Можно использовать весовые величины, предложенные в статье [7], или величины, основанные наличном опыте.

Нескорректированный подсчет функциональных точек (unadjusted function-point count – UFC) выполняется путем вычисления суммы произведений оценки каждого фактора (количество элементов, составляющих данный фактор) на выбранную весовую величину этого фактора:

UFC = ∑ (количество элементов данного типа) х (весовая величина).

Первоначальный метод подсчета функциональных точек был в дальнейшем усовершенствован путем добавления тех факторов, значение которых зависит от общей сложности проекта. Здесь принимается во внимание степень распределенности обработки данных, многократность использования программных элементов, качество функционирования и т.п. Значение, полученное при нескоректированном подсчете функциональных точек, нужно умножить на факторы, определяющие сложность проекта, в результате будет получено итоговое значение.

Вместе с тем в [330] отмечено, что оценка сложности несет в себе также и субъективный фактор, так как подсчет функциональных точек зависит от лица, проводящего оценивание. Люди имеют разные понятия о сложности. Так как подсчет функциональных точек зависит от мнения оценивающего, существует множество вариаций подсчета функциональных точек. Это приводит к разным взглядам на значимость функциональных

точек [122]. Однако многие заявляют, что, несмотря на все недостатки, на практике этот метод себя оправдывает [196].

Альтернативой функциональным точкам являются объектные точки [19], особенно если при разработке ПО используется язык программирования четвертого поколения. Метод объектных точек применяется в модели оценивания COCOMO 2, которая рассматривается далее в главе. (Объектные точки – это отнюдь не классы объектов, которые производятся в результате применения объектно-ориентированного подхода в работе над программой, что можно было бы предположить, исходя из названия.) Количество объектных точек в программе можно получить путем предварительного подсчета ряда элементов.

1. Количество изображений на дисплее. Простые изображения принимаются за 1 объектную точку, изображения умеренной сложности принимаются за 2 точки, а очень сложные изображения принято считать за 3 точки.

2. Количество представленных отчетов. Для простых отчетов назначаются 2 объектные точки, умеренно сложным отчетам назначаются 5 точек. Написание сложных отчетов оценивается в 8 точек.

3. Количество модулей, которые написаны на языках третьего поколения и разработаны в дополнение к коду, написанному на языке программирования четвертого поколения. Каждый модуль на языке третьего поколения считается за 10 объектных точек.

Преимущество данного метода состоит в том, что объектные точки легко оценить исходя из высокоуровневой спецификации программного продукта, поскольку они связаны с конкретными объектами – изображениями, отчетами и модулями на языках программирования третьего поколения.

Количество функциональных и объектных точек можно оценить уже на ранней стадии выполнения проекта. Оценку этих параметров можно начинать сразу после разработки внешних взаимосвязей системы. Именно на

этой стадии очень сложно провести точную оценку размера программы, беря за основу только строки кода. Более того, язык программирования на этом этапе может быть еще не выбран. Вместе с тем оценка на ранней стадии особенно необходима, если используются модели алгоритмического оценивания себестоимости, которые рассматриваются далее.

Подсчет функциональных точек можно проводить параллельно с методом подсчета количества строк кода. Количество функциональных точек при этом используется для оценивания окончательной величины кода. На основе анализа выполнения предыдущих программных проектов для определенных языков программирования можно оценить среднее количество строк кода (average number of lines of code – AVC), необходимых для реализации одной функциональной точки. В этом случае получим оценку размер кода нового проекта, рассчитанную следующим образом:

размер кода = AVC x количество функциональных точек.

Значение AVC варьируется от 200 до 300 строк кода на одну функциональную точку в языке ассемблера и от 2 до 40 строк кода для языков программирования четвертого поколения.

Производительность отдельных программистов, работающих в организации-разработчике, зависит от множества факторов, влияющих на их работу. Некоторые из наиболее значимых факторов представлены в табл. 16.3. Однако различия в индивидуальных способностях программистов намного важнее всех этих факторов. В исследовании ранних этапов программирования [305] отмечено, что производительность некоторых программистов в 10 раз выше, чем у других. Это наблюдение подтверждается и моим личным опытом. Большие команды, члены которых имеют необходимый набор личностных качеств и способностей, будут иметь "среднюю" производительность. Вместе с тем в командах небольшого размера общая производительность во многом зависит от индивидуальных умений и навыков каждого члена команды.

Таблица 16.3. Факторы, влияющие на производительность программиста

Фактор	**Описание**
Опыт разработки ПО для данной предметной области	Для эффективной разработки программного продукта необходимо знание той предметной области, где будет эксплуатироваться разрабатываемое ПО. Инженеры, имеющие понятие об этой предметной области, выявят наивысшую производительность
Процесс управления качеством	Применяемый метод программирования может оказать существенное влияние на производительность написания кода. Этот фактор рассматривается в главе 25
Размер проекта	Чем больше проект, тем больше времени уходит на согласование различных вопросов внутри группы разработчиков. Тем самым уменьшается время, расходуемое непосредственно на разработку ПО, и снижается производительность
Поддержка технологии разработки ПО	Хорошая поддержка технологии разработки ПО, например CASE-средства или системы управления конфигурацией, может значительно повысить производительность труда программиста
Рабочая обстановка	Как уже упоминалось в главе 22, спокойное рабочее окружение с индивидуальными рабочими местами способствует повышению производительности

Должен отметить, что не существует какой-либо величины, определяющей "среднюю" производительность программиста, которую можно было бы применять в разных организациях и при создании разных программных продуктов. Например, при разработке больших систем производительность может быть очень низкой и доходить до 30 строк за человеко-месяц. На простых программных системах производительность может подняться до 900 строк в месяц. В [44] высказано предположение, что производительность программиста может колебаться от 4 до 50 объектных точек в месяц, в зависимости от наличия средств поддержки и от способностей программиста.

Недостатком оценок, которые основываются на подсчете объема выполненной работы или определении количества затраченного времени, является то, что они не принимают во внимание такие важные нефункциональные свойства разрабатываемой системы, как надежность, удобство эксплуатации и т.п. Обычно здесь работает простое правило: больше – значит, лучше. Бек (Beck, [32]) приводит удачное замечание, что если вы работаете над постоянным усовершенствованием и упрощением системы, то подсчет строк ничего не даст.

Описанные методы также не учитывают многократность использования программного продукта. В действительности необходимо оценить стоимость повторного использования определенной системы с данным набором функциональных и качественных характеристик, имеющей собственные показатели удобства сопровождения и т.д. Все эти параметры только косвенно соотносятся с такими количественными показателями, как, например, размер системы.

Трудности также могут возникнуть в случае, если менеджеры используют показатели производительности для оценивания способностей персонала. В этом случае качество выполненной программистом работы может отойти на второй план по отношению к производительности. Может случиться, что "менее продуктивный" программист создаст код, который будет надежнее, понятнее и дешевле в использовании. Поэтому нельзя пользоваться показателями производительности как единственным источником оценивания труда программиста.

16.3. Методы оценивания

Не существует простого метода определения будущих затрат, необходимых для разработки программного продукта. Начальное оценивание можно провести, основываясь на определенных пользовательских требованиях высокого уровня. Но заранее не известно, какие технологии

будут применяться при разработке ПО и какие компьютеры будет использоваться. Также невозможно предугадать, какие люди будут работать над проектом и какие у них будут навыки и опыт. Это показывает, что чрезвычайно трудно провести точную оценку стоимости проекта на самом раннем этапе. Более того, основная проблема в оценке себестоимости проектов заключается в низкой точности применяемых методов оценивания.

Часто в расчет себестоимости проекта закладывается его окупаемость. Таким образом, первоначальная оценка себестоимости определяет бюджет проекта, за рамки которого нельзя выходить при реализации проекта. Вместе с тем я не знаю ни одного реализованного проекта, где бы стоимость не корректировалась по ходу его выполнения. Но всегда в ходе реализации проекта фактические расходы сравниваются с предварительной оценкой затрат.

Несмотря ни на что, организации-разработчики обязательно должны оценивать затраты на разработку и себестоимость программного продукта. Для этого можно применять методы, описанные в табл. 16.4 [46].

Таблица 16.4. Методы оценки себестоимости

Метод	**Описание**
Алгоритмическое моделирование себестоимости	Метод основан на анализе статистических данных о ранее выполненных проектах, при этом определяется зависимость себестоимости проекта от какого-нибудь количественного показателя програ.ммного продукта (обычно это размер программного кода). Проводится оценка этого показателя для данного проекта, после чего с помощью модели прогнозируются будущие затраты
Оценка эксперта	Проводится опрос нескольких экспертов по технологии разработки ПО, знающих область применения создаваемого программного продукта. Каждый из них дает свою оценку себестоимости проекта. Потом все оценки сравниваются и обсуждаются. Этот процесс повторяется до тех пор, пока не будет достигнуто согласие по окончательному варианту предварительной сметы проекта
Оценка по аналогии	Этот метод используется в том случае, если в данной области применения создаваемого ПО уже реализованы аналогичные

	проекты. В таком случае при оценке затрат для сравнения берутся предыдущие проекты. В книге [251] дано достаточно ясное описание этого метода
Закон Паркинсона	Согласно этому закону усилия, затраченные на работу, распределяются равномерно по выделенному на проект времени. Здесь критерием для оценки затрат по проекту являются человеческие ресурсы, а не целевая оценка самого программного продукта. Если проект, над которым работает пять человек, должен быть закончен в течение 12 месяцев, то затраты на его выполнение исчисляются в 60 человеко-месяцев
Назначение цены с целью выиграть контракт	Затраты на проект определяются наличием тех средств, которые имеются у заказчика. Поэтому себестоимость проекта зависит от бюджета заказчика, а не от функциональных характеристик создаваемого продукта

В интересной статье [16] описан эксперимент, в котором менеджеров попросили провести предварительную оценку размера будущей программной системы и необходимых для ее разработки затрат. Менеджеры при этом использовали мнения экспертов и оценку по аналогии. Результаты эксперимента показали, что менеджеры провели достаточно точную оценку затрат, однако определение размера будущей системы при этом было менее точным. Это означает, что оценка затрат, основанная только на данных о размере программы, будет неточной.

Методы предварительной оценки себестоимости могут выполняться с применением нисходящего или восходящего подходов. При нисходящем подходе оценка себестоимости начинается на уровне системы: рассматриваются функциональные возможности программы в целом и то, как эти возможности реализуются посредством функций более низкого уровня. Здесь учитывается себестоимость таких этапов разработки, как сборка системы, управление конфигурацией и создание технической документации.

В отличие от нисходящего подхода, восходящий начинается на уровне системных компонентов. Система разбивается на компоненты и

определяются затраты на разработку каждого из них. Затем эти затраты суммируются для определения полной стоимости проекта.

Недостатки восходящего подхода являются достоинствами нисходящего и наоборот. Восходящий подход может недооценить затраты, необходимые для решения сложных проблем, возникающих при разработке таких специфических компонентов системы, как интерфейсы для нестандартных аппаратных средств. Детального обоснования для составления сметы затрат в этом случае не существует. Нисходящий подход, напротив, дает такое обоснование, а также возможность рассмотреть каждый компонент в отдельности. Вместе с тем данный подход больше акцентирует внимание на таких этапах реализации проекта, как, например, сборка системы. Кроме того, нисходящий подход является более дорогостоящим. Для его применения нужно иметь хотя бы предварительный результат проектирования системы с тем, чтобы оценить каждый ее компонент.

Каждый метод оценивания, безусловно, имеет слабые и сильные стороны. Для работы с большими проектами необходимо применить несколько методов оценивания себестоимости для их последующего сравнения. Если при этом получаются совершенно разные результаты, значит, информации для получения более точной оценки недостаточно. В этом случае необходимо воспользоваться дополнительной информацией, после чего повторить оценивание, и так до тех пор, пока результаты разных методов не станут достаточно близкими.

Описанные методы оценивания применимы, если документированы требования для будущей системы. В таком случае существует возможность определить функциональные характеристики разрабатываемой системы. Обычно все большие проекты разработки ПО имеют документ, в котором определены требования к системе.

Однако во многих проектах оценка затрат проводится только на основании проекта требований к системе. В этом случае лица, участвующие в оценке стоимости проекта, будут иметь минимум информации для работы.

Процедуры анализа требований и создания спецификации весьма дорогостоящи. Поэтому менеджерам компании следует составить смету на их выполнение еще до утверждения бюджета для всего проекта.

Во многих случаях популярной становится стратегия ценообразования с целью "выиграть контракт". Можно согласиться, что данная фраза звучит несколько некорректно и не по-деловому, однако эта стратегия на самом деле себя оправдывает. Стоимость работы согласовывается на основании предварительного проекта предложения. Далее проводятся переговоры между компанией-исполнителем и заказчиком с тем, чтобы обсудить детальное техническое задание, которое, однако, ограничивается согласованной суммой. Продавец и заказчик также должны обсудить приемлемые функциональные возможности системы. Здесь основополагающим фактором для многих проектов становятся возможности бюджета, а не требования к системе. Требования всегда можно изменить так, чтобы не выходить за рамки принятого бюджета.

При оценке себестоимости проекта менеджеры должны всегда помнить о том, что между прошлыми проектами и будущими разработками может быть существенная разница. Только за последние 10 лет на свет появился целый ряд новейших разработок и технологий. Многие менеджеры очень мало ознакомлены, а иногда просто не имеют понятия об этих технологиях и о том, какое влияние они могут оказать на проект. Вот несколько примеров технических и технологических новшеств, которые могут повлиять на оценку стоимости проекта, основанную на предыдущем опыте.

- Появление объектно-ориентированного программирования вместо процедурного.
- Применение систем типа клиент/сервер вместо систем, основанных на мэйнфреймах.
- Применение готовых коммерческих пакетов программного обеспечения вместо собственной разработки компонентов системы.

- Повторное использование компонентов системы вместо новых разработок.
- Использование CASE-средств и генераторов программ вместо разработки ПО без применения средств поддержки.

Все эти факторы отнюдь не облегчают задачу менеджера в оценке стоимости программной продукции. И в этом случае предыдущий опыт не всегда оказывается полезным для проведения такой оценки.

16.4. Алгоритмическое моделирование стоимости

Алгоритмическое моделирование считается наиболее системным подходом к определению стоимости, однако это не значит, что он всегда дает точные результаты. Алгоритмическую модель стоимости можно построить с помощью анализа затрат и параметров уже разработанных проектов. Для прогнозирования затрат применяется математическая формула, в которой учтены данные о размере проекта, количестве программистов, *а* также другие факторы и процессы. В работе [198] приведены 13 алгоритмических моделей прогнозирования затрат, построенных на анализе выполнения предыдущих проектов.

В большинстве алгоритмических моделей формулы вычисления затрат имеют экспоненциальный вид. Причина этого – отсутствие линейной зависимости себестоимости проекта от его размера. С увеличением проекта появляются дополнительные расходы, связанные с ростом затрат на коммуникации, усложнением управления конфигурацией, увеличением объема работ по сборке системы и т.д. Оценки затрат также могут умножаться на коэффициенты, учитывающие свойства разрабатываемого программного продукта, платформу разработки, технологию создания ПО и квалификацию привлеченных специалистов.

В общем случае формула для вычисления алгоритмической оценки стоимости записывается следующим образом:

затраты = A x размер^{B} x M,

где **A** – постоянный коэффициент, который зависит от организации выполнения проекта и типа разрабатываемого программного обеспечения; показатель **размер** может соотноситься либо с размером кода программы, либо с функциональной оценкой, выраженной в количестве объектных или функциональных точек; показатель степени **B** может варьироваться в пределах от 1 до 1.5, он отображает объем работ, требующийся для реализации больших проектов; множитель **M** отображает характеристики различных этапов разработки, а также характеристики создаваемого продукта.

Алгоритмическим моделям присущи общие проблемы.

1. На ранней стадии выполнения проекта бывает сложно определить показатель **размер** при наличии информации только о системных требованиях. Несмотря на то что оценка, основанная на функциональных, или объектных, точках, проще оценки размера кода, результаты тоже не всегда будут точными.

2. Оценка факторов, которые влияют на показатели **B** и **M**, носит субъективный характер. Значения этих показателей могут отличаться, если этим занимаются люди с разным опытом и квалификацией.

Основой для многих алгоритмических моделей оценки себестоимости является количество строк программного кода в созданной системе. Оценку размера кода можно получить по аналогии с другими проектами путем преобразования функциональных точек в размер кода, сравнения размеров компонентов системы и сравнения с эталонными компонентами, а также просто на основе инженерной интуиции.

На размер окончательной системы могут повлиять решения, которые были приняты в процессе реализации проекта уже после утверждения начальной сметы. Например, это может быть решение о том, использовать ли в создаваемом ПО, требующем сложной системы управления данными, базы данных сторонних производителей или разработать свои системы управления

данными. Очевидно, что при использовании сторонних баз данных программный код, который следует создать, будет меньшего размера. Кроме того, имеет значение и язык программирования. Для таких языков, как Java, потребуется больше строк кода, чем если бы применялся, скажем, язык С. В то же время "лишний" код на языке Java потребует провести больше проверок программы в процессе компиляции, вследствие чего расходы на аттестацию системы наверняка снизятся. Чему в данном случае отдать предпочтение? Кроме прочего, также необходимо оценить объем повторного использования кода.

При использовании алгоритмических моделей для ценообразования проекта в них должен учитываться тип проекта, при этом результаты оценивания необходимо тщательно анализировать. Менеджер должен составить не одну, а несколько оценок стоимости (среди них наименее выгодную, ожидаемую и наиболее выгодную). Следует также помнить о большой вероятности значительных ошибок при раннем прогнозировании себестоимости. Наиболее точные оценки можно получить в том случае, если создаваемый продукт хорошо структурирован, модель учитывает интересы организации-заказчика, заранее определены язык программирования и необходимые аппаратные средства.

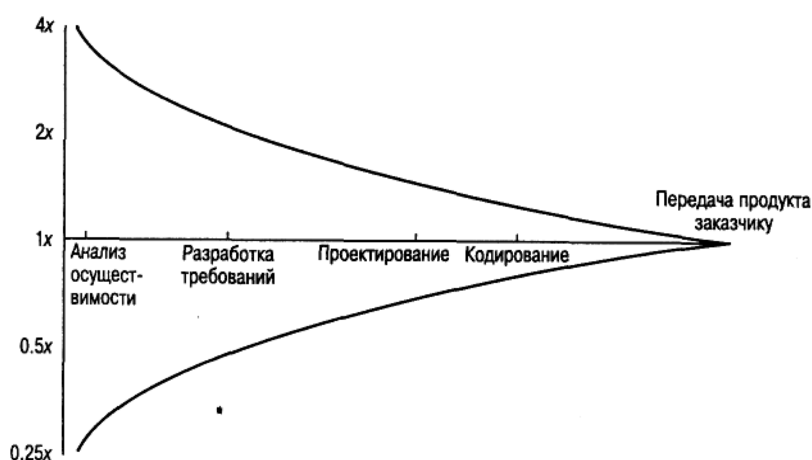

Рис. 16.1. Изменчивость оценивания затрат

Точность результатов прогнозирования себестоимости также зависит от количества информации о создаваемой системе. По ходу реализации

проекта увеличивается количество информации, вследствие чего оценка себестоимости становится все более точной. Если при начальном оценивании временных затрат для разработки системы требовалось x месяцев, то реальная длительность выполнения проекта может колебаться от 0.25x до 4x. Однако этот диапазон постепенно сужается в процессе выполнения проекта, как показано на рис. 16.1. Эта схема была заимствована из статьи [44] и основана на опыте реализации многих проектов разработки программных продуктов.

Управление качеством

Для многих организаций основным критерием деятельности является достижение высокого уровня качества производимой продукции либо предоставляемых услуг. В наше время невозможна поставка продуктов низкого качества, требующих устранения недоработок после доставки заказчику. К программным продуктам это относится не в меньшей мере, чем к таким промышленным товарам, как автомобили, телевизоры или вычислительная техника.

Однако качество программного продукта является достаточно сложным понятием, трудным для определения. Традиционно продукт считается качественным в том случае, если полностью соответствует техническим требованиям [82]. В идеале такое определение должно быть применимо ко всем продуктам, в том числе и к программным, однако здесь нас подстерегают некоторые проблемы.

1. Технические требования ориентированы на те свойства продукта, которые необходимы заказчику. Однако организация-разработчик может также иметь свои требования к разрабатываемому программному продукту (например, удобство сопровождения), которые обычно не включаются в технические требования заказчика.

2. Неизвестно, как точно определить и измерить определенные показатели качества (например, то же удобство сопровождения).

3. Как уже упоминалось в первой части книги, трудно создать полную спецификацию программного продукта. Поэтому, хотя созданный программный продукт будет полностью соответствовать спецификации, заказчик все равно может не получить высококачественного продукта.

Очевидно, необходимо прилагать усилия для совершенствования спецификации, однако на данном этапе следует смириться с тем, что она будет не лишена недостатков. Таким образом, следует признать существование проблемы несовершенства спецификаций и привести в действие ряд процедур для улучшения качества ПО в рамках ограничений, возникающих вследствие этой проблемы. Особенно это касается таких определяющих качественных характеристик программных продуктов, как удобство сопровождения, переносимость и эффективность, которые детально не определены в технических требованиях, однако оказываются критическими показателями для качества программных систем. В разделе 16.2, раскрывающем вопросы планирования качества, эти показатели рассматриваются более подробно.

Достижение необходимого уровня качества зависит от менеджеров по качеству компании-разработчика. Теоретически управление качеством основывается на принципе определения стандартов и процедурных норм, в соответствии с которыми должно разрабатываться программное обеспечение, а также на проверке выполнения этих норм всеми разработчиками. На практике, однако, понятие управления качеством имеет более емкое содержание.

Хорошие менеджеры по управлению качеством стремятся к созданию в компании атмосферы "культивирования качества", где каждый, кто занимается разработкой продукта, берет на себя обязательство достичь наивысшего уровня качества создаваемого продукта. Такие менеджеры стимулируют команду к качественному выполнению работы и к постоянному поиску идей повышения качества. При том что стандарты и процедурные нормы являются основой качества, опытные менеджеры по управлению

качеством осознают значение тех неосязаемых аспектов качества программных продуктов, которые не могут быть включены в стандарты (например, изящество, читабельность и т.п.). Они поддерживают служащих, заинтересованных именно в таких нематериальных аспектах, а также поощряют профессиональное отношение к работе всех членов команды.

Процесс управления качеством состоит из трех основных видов деятельности.

1. **Обеспечение качества**. Определение множества организационных процедур и стандартов в целях создания ПО высокого качества.

2. **Планирование качества**. Выбор из этого множества соответствующего подмножества процедур и стандартов и адаптация их к данному проекту разработки ПО.

3. **Контроль качества**. Определение и проведение мероприятий, гарантирующих выполнение нормативных процедур и стандартов качества всеми членами команды разработчиков ПО.

Управление качеством предполагает возможность независимого контроля за процессом разработки ПО. Контрольные проектные элементы, получаемые в процессе разработки ПО, являются основой контроля качества. Они тщательно проверяются на соответствие стандартам и целям проекта (рис. 16.1.). Так как работы, выполняемые по обеспечению и контролю качества, в определенной степени независимы, это предполагает возможность объективного взгляда на процесс разработки ПО, благодаря чему руководство компании может своевременно получить информацию о проблемах или трудностях, которые возникают в работе над проектом.

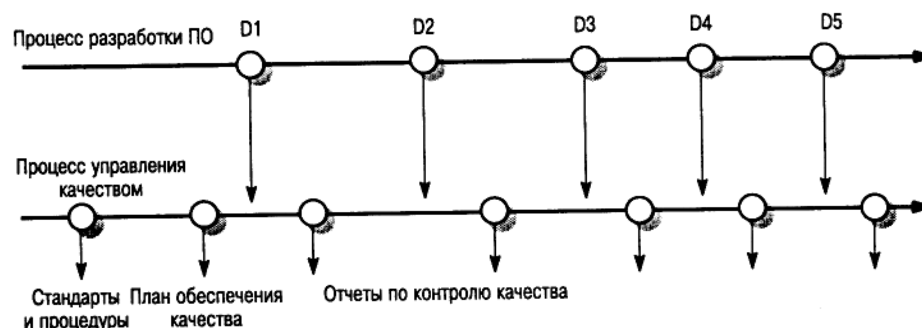

Рис. 16.1. Управление качеством и разработка ПО (буквой D обозначены контрольные проектные элементы)

Процесс управления качеством необходимо отделять от процесса управления проектом с тем, чтобы не ставить вопрос о компромиссе между качеством создаваемого ПО и бюджетом или графиком выполнения проекта. Над контролем качества должна работать независимая команда, которая отчитывается непосредственно руководству компании, минуя звено менеджера проекта. Команда контроля за качеством не должна быть также связана с группами разработки ПО, вместе с тем она берет на себя ответственность за качество на уровне всей организации.

Международно признанным стандартом, который любая компания в любых сферах производства может принять за основу развития системы управления качеством, можно назвать ISO 9000, разработанный Международной организации по стандартизации (ISO). ISO 9000 – это целый ряд всевозможных стандартов, применимых как в промышленности, так и в сфере услуг. ISO 9001 является наиболее обобщенным из этих стандартов и относится к организациям, занимающимся разработкой, производством и сопровождением различных товаров. Поддерживающая документация (ISO 9000-3) адаптирует ISO 9000 к разработке программных продуктов. Стандарт ISO 9000 описан во многих книгах, например [191, 265, 274, 9*, 20*].

Стандарт ISO 9001 является типовой моделью для процесса обеспечения качества. В этом нормативе описываются разнообразные аспекты данного процесса, а также определяются те стандарты и нормативы, которые должны быть приняты за основу произведственной деятельности

компании. Так как процесс обеспечения качества не относится к разряду производственных видов деятельности, нормативы здесь детально не описаны. Любая организация, специализирующаяся на определенном виде услуг, должна самостоятельно провести детализацию своих нормативов и представить ее в специальном руководстве по управлению качеством.

В табл. 16.1 показаны те виды деятельности, которые охвачены в модели ISO 9001. Здесь у меня нет возможности пространно рассуждать о стандарте ISO и вникать в его глубины. Более подробное описание этой модели читатель найдет в книгах [178, 265], где можно получить сведения о применении данного стандарта к процессу управления качеством.

Таблица 16.1. Виды деятельности, охватываемые моделью обеспечения качества ISO 9001

За что отвечает менеджмент	Элементы системы качества
Выявление изделий, не удовлетворяющих техническим требованиям	Контроль за разработкой изделий
Обработка, хранение, упаковка и доставка товара	Материально-техническое обслуживание
Товары, поставляемые заказчиком	Идентификация и отслеживание товара
Управление производственным процессом	Контроль и испытания готовой продукции
Оборудование для контроля и испытаний	Проведение обследования и тестирования
Проверка контракта	Корректирующая деятельность
Проверка документации	Отчеты об обеспечении качества
Внутренняя проверка качества	Обучение
Обслуживание	Статистические методы контроля за качеством

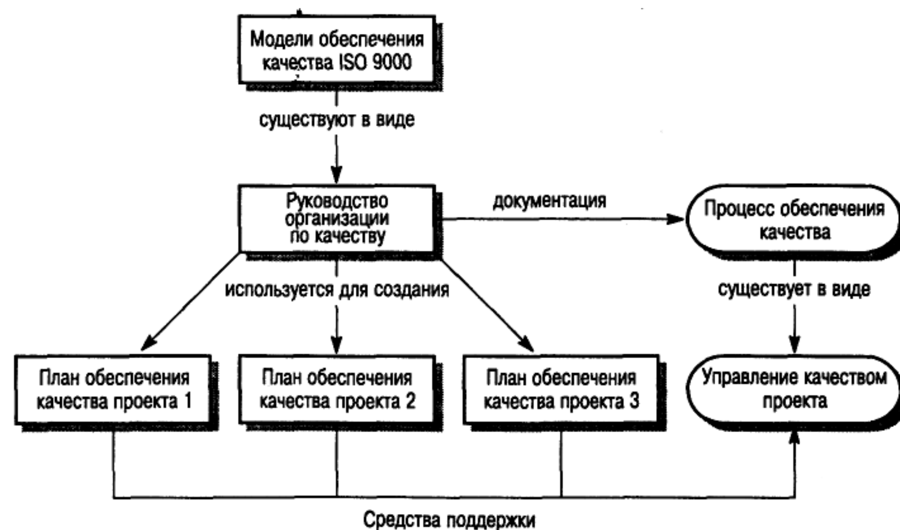

Рис. 16.2. Стандарт ISO 9000 и управление качеством

Нормативы по обеспечению качества занесены в специальное руководство, определяющее ход процесса по управлению качеством. В некоторых странах существуют специальные органы, подтверждающие соответствие процесса обеспечения качества, описанного в руководстве организации, стандарту ISO 9001. Более того, заказчики часто требуют от поставщиков сертификат по стандарту ISO 9001 как подтверждение того, насколько серьезно компания относится к изготовлению качественной продукции.

Взаимосвязь между ISO 9000, управлением качества и планами обеспечения качества отдельных проектов показана на рис. 16.2. Она заимствована из книги [178].

16.5. Обеспечение качества и стандарты

Деятельность по обеспечению качества направлена на достижение определенного уровня качества при разработке программного обеспечения. Она предполагает определение или выбор стандартов, применяемых либо к самому процессу разработки ПО, либо к готовому продукту. Эти стандарты могут быть частью процессов производства ПО. В ходе выполнения таких

процессов могут применяться средства поддержки, учитывающие выбранные (или разработанные) стандарты качества.

В процессе обеспечения качества могут применяться два вида стандартов.

1. **Стандарты на продукцию**. Применимы к уже готовым программным продуктам. Они включают стандарты на сопроводительную документацию, например структуру документа, описывающего системные требования, а также такие стандарты, как, например, стандарт заголовка комментариев в определении класса объекта, стандарты написания программного кода, определяющие способ использования языка программирования.

2. **Стандарты на процесс создания ПО**. Определяют ход самого процесса создания программного продукта, например разработку спецификации, процессы проектирования и аттестации. Кроме того, они могут описывать документацию, создаваемую в ходе выполнения этих процессов.

Между стандартами на продукцию и стандартами на процесс существует очень сильная взаимосвязь. Стандарты на продукцию применимы к результату процесса разработки ПО, а стандарты на процесс в большинстве случаев подразумевают выполнение определенных действий, направленных на получение товара, соответствующего стандартам на продукцию. Более подробно этот вопрос рассматривается в разделе 16.1.2.

Стандарты в разработке программной продукции важны по целому ряду причин, основные из которых перечислены ниже.

1. Стандарты аккумулируют все лучшее из практической деятельности по созданию ПО. Как правило, практические знания приобретаются путем долгого поиска и ошибок. Привнесение этого опыта в определенный стандарт помогает избежать повторения прошлых ошибок. Стандарты в данном случае собирают знания и опыт, имеющие значение для организации-разработчика.

2. Стандарты предоставляют необходимую основу для реализации процесса обеспечения качества. Имея в наличии стандарты, обобщающие лучшие знания и опыт, для обеспечения качества достаточно контролировать, чтобы они выполнялись в процессе создания ПО.

3. Стандарты незаменимы, когда работа переходит от одного сотрудника к другому. В этом случае деятельность всех специалистов в организации подчиняется единому нормативу. Следовательно, требуется меньше затрат на изучение сотрудником новой работы.

Создание стандартов по разработке ПО – процесс долгий и утомительный. Такие национальные и международные организации, как Министерство обороны США, Американский национальный институт стандартизации (ANSI), Британский институт стандартов (BSI), НАТО, Институт инженеров по электротехнике и электронике (IEEE), специализируются на создании общих стандартов, которые могут применяться к широкому диапазону возможных программных проектов. Такие органы, как НАТО, или другие оборонные организации могут требовать соблюдения их собственных стандартов в контрактах на создание программного обеспечения.

Национальные (американские) и международные стандарты были разработаны для таких сфер программной деятельности, как терминология инженерии ПО, языки программирования типа Ada и C++, система обозначений, например для символов в схемах и чертежах, процедуры для разработки системных требований, деятельность по обеспечению качества, а также для аттестации ПО [177]*.*

Группы обеспечения качества, которые занимаются составлением стандартов, обычно основывают нормативы организации на общих национальных и международных стандартах. Используя их в качестве отправного пункта, группа обеспечения качества разрабатывает свой "справочник" по стандартам. В нем содержатся стандарты, отражающие

специфику деятельности данной организации. В табл. 16.2 приведены примеры стандартов, которые могут входить в состав такого справочника.

Таблица 16.2. Стандарты на продукцию и процесс разработки ПО

Стандарты на продукцию	**Стандарты на процесс разработки ПО**
Форма пересмотра архитектуры ПО	Руководство по проведению пересмотра архитектуры ПО
Структура документа, содержащего системные требования	Представление документации по нормативам ЕЭС
Формат заголовков программ и процедур	Процесс выпуска версии ПО
Стиль программирования языка Java	Процесс утверждения плана реализации проекта
Формат плана реализации проекта	Процесс контроля изменений
Форма запроса на изменения	Процесс регистрации выполнения тестов

Иногда специалисты по разработке ПО относятся к стандартам как к бюрократическому наследию, неприменимому к разработке ПО. Особенно это проявляется при выполнении такой утомительной и скучной (однако необходимой для соблюдения стандартов) процедуры, как заполнение всевозможных форм и регистрация работ. Конечно, с общей идеей полезности стандартов все согласны, однако при этом разработчики находят любую удобную причину, по которой именно для их проекта эти стандарты не так уж необходимы.

Чтобы не возникало подобных проблем в работе, менеджеры по качеству, отвечающие за разработку стандартов, должны быть достаточно подготовленными и действовать следующим образом.

1. Вовлечь самих программистов в разработку стандартов. Они должны ясно понимать, с какой целью разрабатывается стандарт, и четко следовать установленным правилам и нормативам. Важно, чтобы документ с описанием стандарта включал не только изложение самого норматива качества, но и объяснение необходимости именно этого норматива.

2. Регулярно просматривать и обновлять стандарты, чтобы идти в ногу с быстро развивающимися технологиями. Как только стандарт разработан, его помещают в справочник организации по стандартам, где его холят и лелеют, меняя с большой неохотой. Справочник по стандартам – вещь для организации необходимая, однако он должен развиваться по мере развития новых технологий.

3. Подумать о том, чтобы обеспечить поддержку стандартов программными средствами везде, где только можно. Всяческие "канцелярские" стандарты вызывают огромное количество жалоб, связанных с нудной и утомительной работой по их выполнению. Если же имеются в наличии средства поддержки, выполнение стандартов не требует больших усилий.

Стандарты на процессы разработки ПО также могут вызвать ряд проблем, если ставят перед командой разработчиков практически неосуществимые задачи. Такие стандарты дают руководящие советы по выполнению работы, при этом менеджеры проектов могут интерпретировать их каждый по-своему. Нет смысла указывать определенное направление работы, если оно неприменимо к данному проекту или к самой команде разработчиков. Поэтому менеджер проекта должен иметь право изменять стандарты процесса создания ПО в соответствии со специфическими условиями именно данного проекта. Здесь следует оговориться, что это утверждение не относится к стандартам на качество готовой продукции и на процесс сопровождения программной системы, которые могут быть изменены только после глубокого изучения данного вопроса.

Менеджер проекта и менеджер по управлению качеством могут легко избежать подводных камней, связанных с "неподходящими" стандартами, путем тщательной разработки плана мероприятий по обеспечению качества. Именно они должны решить, какие стандарты из справочника можно использовать без изменений, какие из них подлежат изменению, а какие следует исключить. Иногда возникает необходимость в разработке нового

стандарта, что может быть вызвано условиями выполнения определенного проекта. Например, требуется установить стандарт для формальной спецификации, если прежде в проектах он не использовался. Такие стандарты должны разрабатываться в процессе выполнения проекта.

16.6. Стандарты на техническую документацию

Необходимость стандартов на документацию в программном проекте становится очевидной, если не существует никакого другого реального способа отображения процесса разработки ПО. Стандартные документы имеют четкую последовательную структуру, вид и качество, а значит, их легко читать и воспринимать. Существует три типа стандартов на документацию.

1. **Стандарты на процесс создания документации.** Определяют способ создания технической документации.
2. **Стандарты на документ**. Определяют структуру и внешний вид документов.
3. **Стандарты на обмен документами**. Гарантируют совместимость всех электронных версий документов.

Стандарт на процесс создания документации предоставляет описание способа изготовления документов. Он включает описания действий по созданию документов и предполагает программные средства для их создания. Кроме этого, нужно описать процедуры проверки и редактирования документов, благодаря которым обеспечивается необходимый уровень их качества.

Стандарты качества на процесс документирования должны быть достаточно гибкими, чтобы их можно было применять ко всем типам документации. Естественно, совсем необязательно проводить детальные проверки качества рабочей документации или служебных записок. Однако при работе с официальными документами, имеющими отношение к

дальнейшей разработке продукта либо предназначенными для заказчика, нужно иметь установленную процедуру проверки их качества. На рис. 16.3 показана одна из возможных моделей такого процесса.

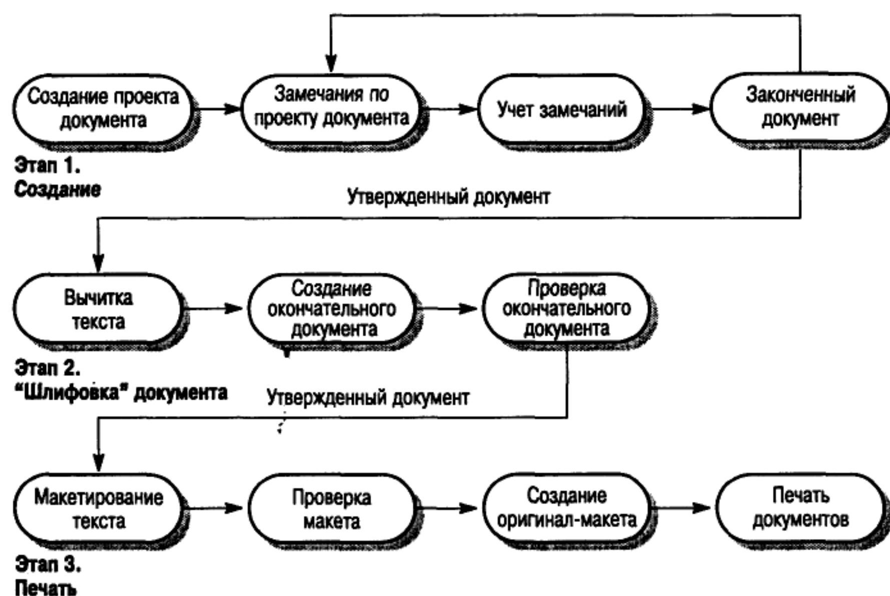

Рис. 16.3. Процесс создания документации, включающий проверку качества документов

Этапы сбора, учета и внесения замечаний в проект или очередную версию документа повторяются до тех пор, пока не будет создан документ соответствующего уровня качества. Уровень качества документа зависит от типа документации и от того, кому предназначен данный документ.

Стандарты на документацию следует применять ко всем документам, которые создаются в процессе работы над программным продуктом. Такие документы должны быть одинаковыми по стилю изложения и внешнему виду, а документы, которые относятся к одному типу, должны также иметь одинаковую структуру. Несмотря на то что стандарты на документацию могут быть адаптированы к определенного вида проектам, все же для организации неплохо было бы выработать собственный "фирменный" стиль, применяемый ко всем документам.

Приведем примеры различных стандартов на документацию.

1. **Стандарты идентификационных документов**. В процессе создания больших систем производятся тысячи документов, каждому из

которых должно быть присвоено уникальное имя, т.е. каждый документ должен быть идентифицирован по определенной системе. Для формальных документов может применяться формальный идентификатор, определенный менеджером по конфигурации. Для неформальных документов идентификатор может быть определен менеджером проекта.

2. **Стандарты на структуру документа.** Каждый класс документов, создаваемый в процессе разработки ПО, должен иметь стандартную структуру. Стандартами на структуру определяются разделы, которые входят в документ, а также элементы форматирования и макетирования документа, например нумерация страниц, содержание верхнего и нижнего колонтитулов, нумерация разделов и подразделов.

3. **Стандарты внешнего вида документации**. Эти стандарты определяют “фирменный” стиль документов: использование шрифтов и стилей, логотипов, названия фирмы, цветовых выделений для элементов структуры документа и т.д.

4. **Стандарты на обновление документации**. Так как документы должны отображать изменения, возникающие в процессе разработки системы, желательно применять последовательный способ обозначения изменений в документации. Чтобы выделить новую, измененную версию документа, для печатных документов можно использовать обложки разных цветов, а для выделения изменений в тексте можно делать цветовые указатели на полях.

Особая роль отводится стандарту на совместимость документов, так как часто приходится работать с разными электронными версиями документов. Использование стандартов на совместимость позволяет осуществлять передачу документов в электронном виде и при необходимости восстанавливать их в первоначальном виде.

Поскольку стандартами на процесс создания документации предусмотрено использование инструментальных средств, стандарты на совместимость очерчивают сферу и способ использования таких средств в

работе над документами. В качестве примера подобных стандартов на совместимость можно привести согласованный стандарт на комплект макросов при форматировании текста документа или на использование стандартной таблицы стилей (шаблонов) при работе с системами обработки текстов. Кроме того, введение в действие стандартов на совместимость может ограничить количество используемых шрифтов и стилей, что вызвано различными возможностями принтеров и компьютерных дисплеев.

Контрольные вопросы:

1) Что такое эффективный менеджмент?
2) На какие вопросы менеджеры проекта должны научиться давать правильные ответы?
3) Какие три параметра используются для оценки проекта по разработке программного обеспечения?
4) Какие общие проблемы присущи алгоритмическим моделям?
5) Какие виды стандартов могут применяться в процессе обеспечения качества?
6) Перечислите три типа стандартов на документацию.

Ключевые слова: *эффективный менеджмент, менеджеры, показатель размера, функциональный показатель, обеспечение качества, планирование качества, контроль качества, стандарты на продукцию, стандарты на процесс создания по, стандарты на процесс создания документации, стандарты на документ, стандарты на обмен документами.*

Keywords: *effective management, managers, index size, functional index, quality assurance, quality planning, quality control, product standards, standards in the process of building on standards for the process of creating documentation, standards, standards for document exchange.*

Kalit so'zlar: *unumli menedjment, menedjer, kattalik ko'rsatkichi, funksional ko'rsatkich, sifat ta'minoti, sifat rejasi, sifat nazorati, mahsulot uchun standart, DT yaratish jarayoni standarti, hujjat standarti, hujjatlar almashinuvi standarti.*

Упражнения

1. Создайте модели следующих процессов.

• Разжигание костра.

• Приготовление комплексного обеда (меню по вашему выбору).

• Написание небольшой программы (не более 50 строк).

2. В каких условиях качество работы команды разработчиков будет определять качество готового программного продукта? Приведите примеры программных систем, при разработке которых особое значение имеют талант и способности отдельного программиста.

3. Предположим, что целью совершенствования процесса разработки ПО является увеличение количества повторно используемых компонентов. Сформулируйте три вопроса в соответствии с парадигмой "цель-вопрос-показатель".

4. Опишите три типа числовых показателей, которые могут использоваться в качестве данных для совершенствования процесса разработки ПО. Приведите по одному примеру каждого типа показателя.

5. Назовите два главных достоинства и два основных недостатка концепции оценивания и совершенствования процесса разработки ПО, которая положена в основу модели оценки уровня развития SEI.

6. Приведите две области применения ПО, для которых модель SEI неприменима. Обоснуйте ваш выбор.

7. Рассмотрите технологический процесс создания ПО в вашей компании и определите его тип. Сколько ключевых составля'ющих процесса по модели SEI находятся в применении? Какого уровня развития достигла ваша компания по этой модели?

8. Приведите пример трех средств поддержки, которые могут использоваться для сопровождения процесса совершенствования производства ПО.

9. Объясните, почему методически обоснованный процесс не обязательно должен быть управляемым.

10. Считаете ли вы унижающими достоинство программы совершенствования производства, которые предусматривают оценивание работы персонала компании? Что, по-вашему, вызовет сопротивление введению такой программы в действие?

ГЛАВА 17. МОДЕРНИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Невозможно создать систему, которая не потребует изменений в будущем. Как только программное обеспечение вводится в эксплуатацию, возникают новые требования к системе, обусловленные непрерывным развитием бизнес-процессов и все возрастающими общими требованиями к программным системам. Иногда в системе следует изменить некоторые составляющие в целях повышения производительности или улучшения других характеристик, а также для исправления обнаруженных ошибок. Все это требует дальнейшего развития системы после ее ввода в эксплуатацию.

Полная зависимость организаций от программного обеспечения, которое к тому же обходится в достаточно круглую сумму, объясняет исключительную важность серьезного отношения к ПО. Это предусматривает дополнительные вложения в эволюцию уже эксплуатируемой системы с тем, чтобы обеспечить прежний уровень ее производительности.

Существует несколько стратегических подходов к процессу модернизации ПО [339].

1. **Сопровождение программного обеспечения**. Это наиболее часто используемый подход, который заключается в изменении отдельных частей ПО в ответ на растущие требования, но с сохранением основной системной структуры. Подробнее этот вопрос освещен в разделе 17.2.

2. **Эволюция системной архитектуры**. Этот подход более радикальный, чем сопровождение ПО, так как предполагает существенные изменения в программной системе. Эта стратегия модернизации ПО подробно раскрыта в разделе 17.3.

3. **Реинженириng программного обеспечения.** Кардинально отличается от других подходов, так как модернизация предусматривает не внесение каких-то новых компонентов, а наоборот, упрощение системы и удаление из нее всего лишнего. При этом возможны изменения в архитектуре, но без серьезных переделок. Этому вопросу посвящена глава 17.

Приведенные стратегии не исключают одна другую. Иногда для упрощения системы перед изменением архитектуры или для переделки некоторых ее компонентов применяется реинжениринг. Некоторые части системы заменяются серийными, а более стабильные системные компоненты продолжают функционирование. Как уже упоминалось в главе 26, выбор стратегии модернизации системы основывается не только на технических характеристиках, но и на том, насколько хорошо система поддерживает деловую активность компании.

Разные стратегии могут также применяться к отдельным частям системы или к отдельным программам наследуемой системы. Сопровождение приемлемо для программ со стабильной и четкой структурой, не требующей особого внимания. Для других программ, которые постоянно контактируют со многими пользователями, можно изменить архитектуру так, чтобы интерфейс пользователя запускался на машине клиента. Еще один компонент в этой же системе можно заменить аналогичной программой стороннего производителя. Однако при реинжениринге обычно необходимо изменять все компоненты системы.

Изменения в ПО служат причиной появления многочисленных версий системы и ее компонентов. Поэтому особенно важно внимательно следить за всеми этими изменениями, а также за тем, чтобы версия компонента соответствовала той версии системы, в которой он применяется. Управление изменениями системы называется управлением конфигурацией и обсуждается в главе 29.

17.1. Динамика развития программ

Под динамикой развития программ подразумевается исследование изменений в программной системе. Основной работой в этой области является [213]. Результатом этих исследований стало появление ряда "законов" Лемана (Lehman), относящихся к модернизации систем. Считается, что эти законы неизменны и применимы практически во всех случаях. Они сформулированы после исследования процесса создания и эволюции ряда больших программных систем. Эти законы (в сущности, не законы, а гипотезы) приведены в табл. 17.1.

Таблица 17.1. Законы Лемана

Закон	Описание
Непрерывность модернизации	Для программ, эксплуатируемых в реальных условиях, модернизация – это необходимость, иначе их полезность снижается
Возрастающая сложность	По мере развития программы становятся все более сложными. Для упрощения или сохранения их структуры необходимы дополнительные затраты
Эволюция больших систем	Процесс развития систем саморегулируемый. Такие характеристики системы, как размер, время между выпусками очередных версий и количество регистрируемых ошибок, для каждой версии программы остаются практически неизменными
Организационная стабильность	Жизненный цикл системы относительно стабилен, независимо от средств, выделяемых (или не выделяемых) на ее развитие
Стабильность количества изменений	За весь жизненный цикл системы количество изменений в каждой версии остается приблизительно одинаковым

Из первого закона вытекает необходимость постоянного сопровождения системы. При изменении окружения, в котором работает система, появляются новые требования, и система должна неизбежно

изменяться с тем, чтобы им соответствовать. Изменения системы носят циклический характер, когда новые требования порождают появление новой версии системы, что, в свою очередь, вызывает изменения системного окружения; это находит отражение в формировании новых требований к системе и т.д.

Второй закон констатирует нарушение структуры системы после каждой модификации. Это в полной мере демонстрируют наследуемые системы, которые рассматриваются в главе 26. Единственным способом избежать этого, по всей видимости, является только профилактическое обслуживание, которое, однако, требует средств и времени. При этом совершенствуется структура программы без изменения ее функциональности. Поэтому в бюджете, предусмотренном на содержание системы, следует также учесть и эти дополнительные затраты.

Самым спорным и, пожалуй, самым интересным законом Лемана является третий. Согласно этому закону, все большие системы имеют собственную динамику изменений, которая устанавливается на начальном этапе разработки системы. Этим определяются возможности сопровождения системы и ограничивается количество модификаций. Предполагается, что этот закон является результатом действия фундаментальных структурных и организационных факторов. Как только система превышает определенный размер, она начинает действовать подобно некой инерционной массе. Размер становится препятствием для новых изменений, поскольку эти изменения с большой вероятностью станут причиной ошибок в системе, которые снизят эффективность нововведений в новой версии системы.

Четвертый закон Лемана утверждает, что крупные проекты по разработке программного обеспечения действуют в режиме "насыщения". Это означает, что изменения ресурсов или персонала оказывает незначительное влияние на долгосрочное развитие системы. Это, правда, уже указано в третьем законе, который утверждает, что развитие программы не зависит от решений менеджмента. Этим законом также утверждается, что

крупные команды программистов неэффективны, так как время, потраченное на общение и внутрикомандные связи, превышает время непосредственной работы над системой.

Пятый закон затрагивает проблему увеличения количества изменений с каждой новой версией программы. Расширение функциональных возможностей системы каждый раз сопровождается новыми ошибками в системе. Таким образом, масштабное расширение функциональных возможностей в одной версии означает необходимость последующих доработок и исправления ошибок. Поэтому в следующей версии уже будут проведены незначительные модификации. Таким образом, менеджер, формируя бюджет для внесения крупных изменений в версию системы, не должен забывать о необходимости разработки следующей версии с исправленными ошибками предыдущей версии.

В основном законы Лемана выглядят весьма разумными и убедительными. При планировании сопровождения они обязательно должны учитываться. Случается, что по коммерческим соображениям законами следует пренебречь. Например, это может быть обусловлено маркетингом, если существует необходимость провести ряд модификаций системы в одной версии. В результате все равно получится так, что одна или несколько следующих версий будут связаны с исправлением ошибок.

Может показаться, что большие различия между последовательными версиями одной и той же программы опровергнут законы Лемана. Например, Microsoft Word превратилась из простой программы текстовой обработки, требующей 25 Кбайт памяти, в огромную систему с множеством функций. Теперь, для того чтобы работать с этой программой, нужно много памяти и быстродействующий процессор. Эволюция этой программы противоречит четвертому и пятому законам Лемана. Однако я подозреваю, что это все-таки не одна и та же программа, которая просто подверглась ряду изменений. Думаю, программа была существенно переработана; по сути, была

разработана новая программа, но в рекламных целях был сохранен единый логотип.

17.2. Сопровождение программного обеспечения

Сопровождение – это обычный процесс изменения системы после ее поставки заказчику. Эти изменения могут быть как элементарно простыми (исправление ошибок программирования), так и более серьезными, связанными с корректировкой отдельных недоработок либо приведением в соответствие с новыми требованиями. Как упоминалось в вводной части главы, сопровождение не связано со значительным изменением архитектуры системы. При сопровождении тактика простая: изменение существующих компонентов системы либо добавление новых.

Существует три вида сопровождения системы.

1. **Сопровождение с целью исправления ошибок**. Обычно ошибки в программировании достаточно легко устранимы, однако ошибки проектирования стоят дорого и требуют корректировки или перепрограммирования некоторых компонентов. Самые дорогие исправления связаны с ошибками в системных требованиях, так как здесь может понадобиться перепроектирование системы.

2. **Сопровождение с целью адаптации ПО к специфическим условиям эксплуатации**. Это может потребоваться при изменении определенных составляющих рабочего окружения системы, например аппаратных средств, операционной системы или программных средств поддержки. Чтобы адаптироваться к этим изменениям, система должна быть подвергнута определенным модификациям.

3. **Сопровождение с целью изменения функциональных возможностей системы**. В ответ на организационные или деловые изменения в организации могут измениться требования к программным средствам. В таких случаях применяется данный тип сопровождения.

Наиболее существенные изменения при этом претерпевает именно программное обеспечение.

На практике однозначно четкое разграничение между различными видами сопровождения провести достаточно сложно. Ошибки в системе могут быть выявлены в том случае, если, например, система использовалась непредсказуемым способом. Поэтому наилучший способ исправления ошибок – расширение функциональных возможностей программы с тем, чтобы сделать работу с ней как можно проще. При адаптации программного обеспечения к новому рабочему окружению расширение функциональных возможностей системы будет способствовать улучшению ее работы. Также добавление определенных функций в программу может оказаться полезным, если в случае ошибок был изменен шаблон использования системы и побочным действием при расширении функциональных возможностей будет удаление ошибок.

Перечисленные типы сопровождения широко используются, хотя им подчас дают разные названия. Сопровождение с целью исправления ошибок обычно называют корректирующим. Название "адаптивное сопровождение" может относиться как к адаптации к новому рабочему окружению, так и к новым требованиям. Усовершенствование программного обеспечения может означать улучшение путем соответствия новым требованиям, а также усовершенствование структуры и производительности с сохранением функциональных возможностей. Я намеренно не употребляю здесь все эти названия, чтобы избежать излишней путаницы.

Рис. 17.1. Распределение типов сопровождения

Найти современные данные относительно того, как часто используется тот или иной тип сопровождения, будет нелегко. Согласно исследованиям [218], которые уже несколько устарели, 65% сопровождения связано с выполнением новых требований, 18% отводится на изменения системы с целью адаптации к новому окружению и 17% связано с исправлением ошибок (рис. 17.1.). Десятилетие спустя в работе [259] определены похожие соотношения.

Из этого можно определить, что исправление ошибок не является самым распространенным видом сопровождения. Модернизация системы в соответствии с новым рабочим окружением либо в соответствии с новыми требованиями более эффективна. Поэтому сопровождение само по себе является естественным процессом продолжения разработки системы со своими процессами проектирования, реализации и тестирования. Таким образом, спиральная модель, показанная на рис. 17.2, лучше представляет процесс развития ПО, чем каскадная модель (см. рис. 3.1.), где сопровождение рассматривается как отдельный процесс.

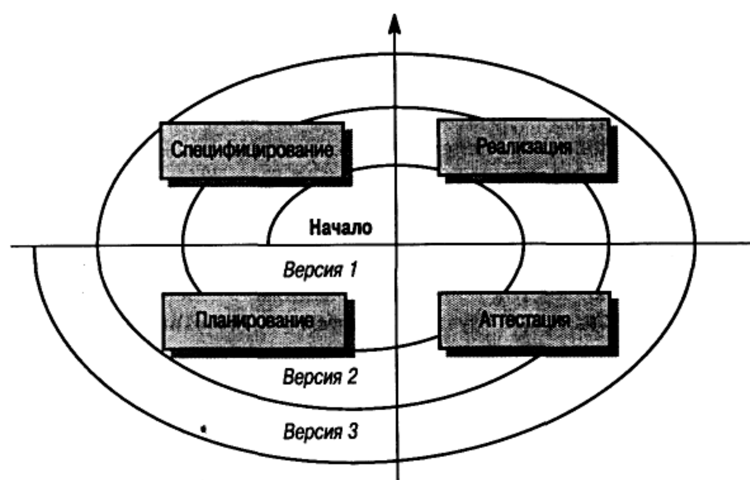

Рис. 17.2. Спиральная модель развития ПО

Значительная часть бюджета большинства организаций уходит на сопровождение ПО, а не на само использование программных систем. В 1980-х годах было обнаружено [218], что во многих организациях по меньшей мере 50% всех средств, потраченных на программирование, идет на развитие уже существующих систем. В работе [235] определено похожее соотношение затрат на различные виды сопровождения, при этом от 65 до 75% средств общего бюджета расходуется на сопровождение. Так как предприятия заменяют старые системы коммерческим ПО, например программами планирования ресурсов, эти цифры никак не будут уменьшаться. Поэтому можно утверждать, что изменение ПО все еще остается доминирующим в статье затрат организаций на программное обеспечение.

Соотношение между величинами средств на сопровождение и на разработку может быть разным в зависимости от предметной области, где эксплуатируется система. Для прикладных систем, работающих в деловой сфере, соотношение затрат на сопровождение в основном сравнимо со средствами, потраченными на разработку. Для встроенных систем реального времени затраты на сопровождение могут в четыре раза превышать стоимость самой разработки. Высокие требования в отношении

производительности и надежности таких систем предполагают их жесткую структуру, которая труднее поддается модификации.

Можно получить значительную общую экономию средств, если заранее потратить финансы и усилия на создание системы, не требующей дорогостоящего сопровождения. Весьма затратно проводить изменения в системе после ее поставки заказчику, поскольку для этого требуется хорошо знать систему и провести анализ реализации этих изменений. Поэтому усилия, потраченные во время разработки программы на снижение стоимости такого анализа, автоматически снизят и затраты на сопровождение. Такие технологии разработки ПО, как формирование четких требований, объектно-ориентированное программирование и управление конфигурацией, способствуют снижению стоимости сопровождения.

На рис. 17.3 показано, как снижается стоимость сопровождения хорошо разработанных систем. Здесь при разработке системы 1 выделено дополнительно $25 000 для облегчения процесса сопровождения. В результате за время эксплуатации системы это помогло сэкономить около $100 000. Из сказанного следует, что увеличение средств на разработку системы пропорционально снизит затраты на ее сопровождение.

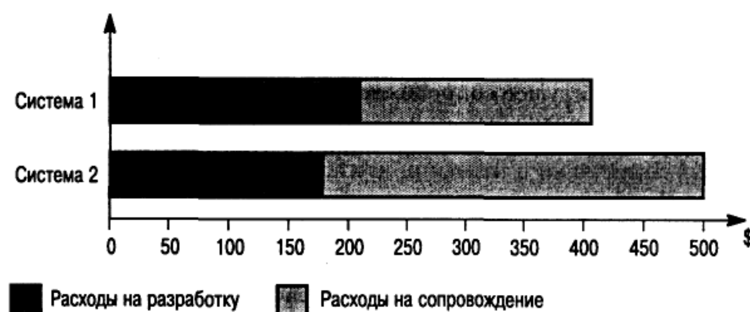

Рис. 17.3. Расходы на разработку и сопровождение систем

Причиной высоких затрат на сопровождение является сложность модернизации системы после ее внедрения, поскольку расширить функциональные возможности намного легче в процессе создания системы. Ниже приведены ключевые факторы, которые определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения.

1. **Стабильность команды разработчиков**. Вполне естественно, что после внедрения системы команда разработчиков распадается, специалисты будут работать над другими проектами. Новым членам команды или же отдельным специалистам, которые возьмут на себя дальнейшее сопровождение системы, будет трудно понять все ее особенности. Поэтому на понимание системы перед внесением в нее изменений уходит много времени и средств.

2. **Ответственность согласно контракту**. Контракт на сопровождение обычно заключается отдельно от договора на разработку программы. Более того, часто контракт на сопровождение может получить фирма, сама не занимающаяся разработкой. Вместе с фактором нестабильности команды это может стать причиной отсутствия в команде стимула создать легкоизменяемую, удобную в сопровождении систему. Если членам команды выгодно пойти кратчайшим путем с минимальными затратами усилий, то вряд ли они откажутся от этого даже с риском повышения последующих затрат на сопровождение.

3. **Квалификация специалистов**. Специалисты, занимающиеся сопровождением, часто не знакомы с предметной областью, где эксплуатируется система. Сопровождение не пользуется популярностью среди разработчиков. Это считается менее квалифицированной разработкой и часто поручается младшему персоналу. Более того, старые системы могут быть написаны на устаревших языках программирования, не знакомых молодым специалистам и требующих дополнительного изучения.

4. **Возраст и структура программы**. *С* возрастом структура программ нарушается вследствие частых изменений, поэтому их становится сложнее понимать и изменять. Кроме того, многие наследуемые системы были созданы без использования современных технологий. Они никогда не отличались хорошей качественной структурой; изменения, сделанные в них, были направлены скорее на повышение эффективности функционирования,

чем на повышение удобства сопровождения. Документация на старые системы часто бывает неполной либо вообще отсутствует.

Первые три проблемы объясняются тем, что многие организации все еще делают различие между разработкой системы и ее сопровождением. Сопровождение считается делом второстепенным, поэтому нет никакого желания инвестировать средства для снижения затрат на будущее сопровождение. Руководству организаций необходимо помнить, что у систем редко бывает четко определенный срок функционирования, наоборот, они могут находиться в эксплуатации в той либо иной форме неограниченное время.

Дилемма заключается в следующем: или создавать системы и поддерживать их до тех пор, пока это возможно, и затем заменять их новыми, или разрабатывать постоянно эволюционирующие системы, которые могут изменяться в соответствии с новыми требованиями. Их можно создавать на основе наследуемых систем, улучшая структуру последних с помощью реинженириНга (см. главу 28), либо путем изменения архитектуры этих систем, что подробно рассматривается в разделе 17.3.

Последняя проблема, а именно нарушение структуры, является самой простой из них. Технология реинжениринга поможет усовершенствовать структуру и повысить понимаемость системы. В подходящих случаях адаптировать систему к новым аппаратным средствам может и преобразование архитектуры (см. далее). Профилактические меры при сопровождении будут полезны, если возникнет необходимость усовершенствовать систему и сделать ее более удобной для изменений.

17.3. Процесс сопровождения

Процессы сопровождения могут быть самыми разными, что зависит от типа программного обеспечения, технологии его разработки, а также от специалистов, которые непосредственно занимались созданием системы. Во

многих организациях сопровождение носит неформальный характер. В большинстве случаев разработчики получают информацию о проблемах системы от самих пользователей в устной форме. Другие же компании имеют формальный процесс сопровождения со структурированной документацией на каждый его этап. Но на самом общем уровне любые процессы сопровождения имеют общие этапы, а именно: анализ изменений, планирование версий, реализация новой версии системы и поставка системы заказчику.

Процесс сопровождения начинается при наличии достаточного количества запросов на изменения от пользователей, менеджеров или покупателей. Далее оцениваются возможные изменения с тем, чтобы определить уровень модернизации системы, а также стоимость внедрения этих изменений. Если принимается решение о модернизации системы, начинается этап планирования новой версии системы. Во время планирования анализируется возможность реализации всех необходимых изменений, будь то исправление ошибок, адаптация или расширение функциональных возможностей системы. Только после этого принимается окончательное решение о том, какие именно изменения будут внесены в систему. Когда изменения реализованы, выходит очередная версия системы. На рис. 17.4, заимствованном из книги [13], представлен весь процесс модернизации.

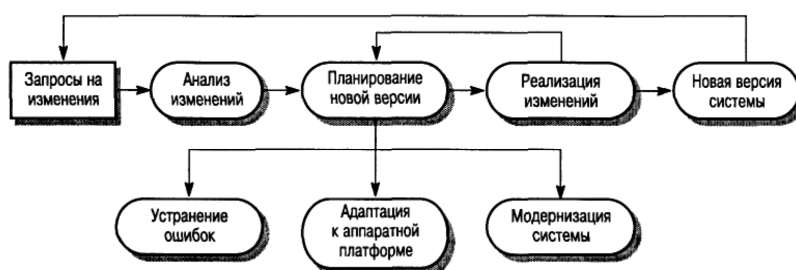

Рис. 17.4. Схема процесса модернизации

В идеале процесс модернизации должен привести к изменению системной спецификации, архитектуры и программной реализации (рис. 17.5). Новые требования должны отражать изменения, вносимые в систему.

Ввод в систему новых компонентов требует ее перепроектирования, после чего необходимо повторное тестирование системы. Для анализа вносимых изменений при необходимости можно создать прототип системы. На этой стадии проводится подробный анализ изменений, при котором могут выявиться те последствия модернизации, которые не были замечены при начальном анализе изменений.

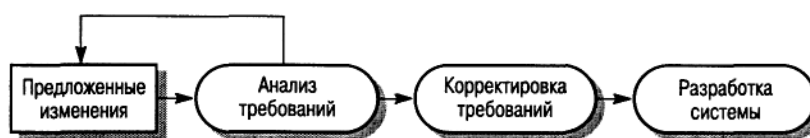

Рис. 17.5. Выполнение модернизации

Иногда в экстренных случаях требуется быстрое внесение изменений, например по следующим причинам.

- Сбой в системе, вследствие чего возникла чрезвычайная ситуация, требующая экстренного вмешательства для продолжения нормальной работы системы.
- Изменение рабочего окружения системы с непредусмотренным влиянием на систему.
- Неожиданные изменения в деловой сфере организации (из-за действий конкурентов или из-за введения нового законодательства).

В таких случаях быстрая реализация изменений имеет большую важность, чем четкое следование формальностям процесса модернизации системы. Вместо того чтобы изменять требования или структуру системы, лучше быстро внести коррективы в программный код (рис. 17.6). Однако этот подход опасен тем, что требования, системная архитектура и программный код постепенно теряют целостность. Этого трудно избежать, если принять во внимание необходимость быстрого выполнения задания, когда тщательная доработка системы откладывается на потом. Если разработчик, который изменил код, внезапно уходит из команды, то его коллеге будет сложно привести в соответствие сделанным изменениям спецификацию и структуру системы.

Рис. 17.6. Процесс экстренной модернизации системы

Еще одна проблема срочных изменений системы состоит в том, что из-за дефицита времени из двух возможных решений будет принято не самое лучшее (в аспекте сохранения структуры системы), а то, которое можно быстрее и эффективнее реализовать. В идеале после срочной коррекции кода системы запрос на изменения должен все еще оставаться в силе. Поэтому после тщательного анализа изменений можно отменить внесенные изменения и принять более оптимальное решение для модернизации системы. Однако на практике такая возможность используется крайне редко, чаще всего в силу субъективных причин.

17.4. Прогнозирование сопровождения

Менеджеры терпеть не могут сюрпризов, особенно если они выливаются в непредсказуемо высокие затраты. Поэтому лучше предусмотреть заранее, какие изменения возможны в системе, с какими компонентами системы будет больше всего проблем при сопровождении, а также рассчитать общие затраты на сопровождение в течение определенного периода времени. На рис. 17.7 представлены различные типы прогнозов, связанные с сопровождением, и показано, на какие вопросы они должны ответить.

Прогнозирование количества запросов на изменения системы зависит от понимания взаимосвязей между системой и ее окружением. Некоторые системы находятся в достаточно сложной взаимозависимости с внешним окружением и изменение окружения обязательно повлияет на систему. Для того чтобы правильно судить об этих взаимоотношениях, необходимо оценить следующие показатели:

1. **Количество и сложность системных интерфейсов**. Чем больше системных интерфейсов и чем более сложными они являются, тем выше вероятность изменений в будущем.

2. **Количество изменяемых системных требований.** Как упоминалось в главе 6, требования, отражающие деловую сферу или стандарты организации, чаще изменяются, чем требования, описывающие предметную область.

3. **Бизнес-процессы, в которых используется данная система**. По мере развития бизнес-процессы приводят к появлению новых требований к системе.

Чтобы корректно спрогнозировать процесс сопровождения, нужно знать количество и типы взаимосвязей между разными компонентами системы, а также учитывать сложность этих компонентов. Различные виды сложности систем изучались в работах [151, 232]. Другие исследования посвящены взаимосвязям между сложностью систем и процессом сопровождения [18, 195]. Все эти исследования показали, что, чем выше сложность системы и ее компонентов, тем более дорогостоящим окажется сопровождение, чего и следовало ожидать.

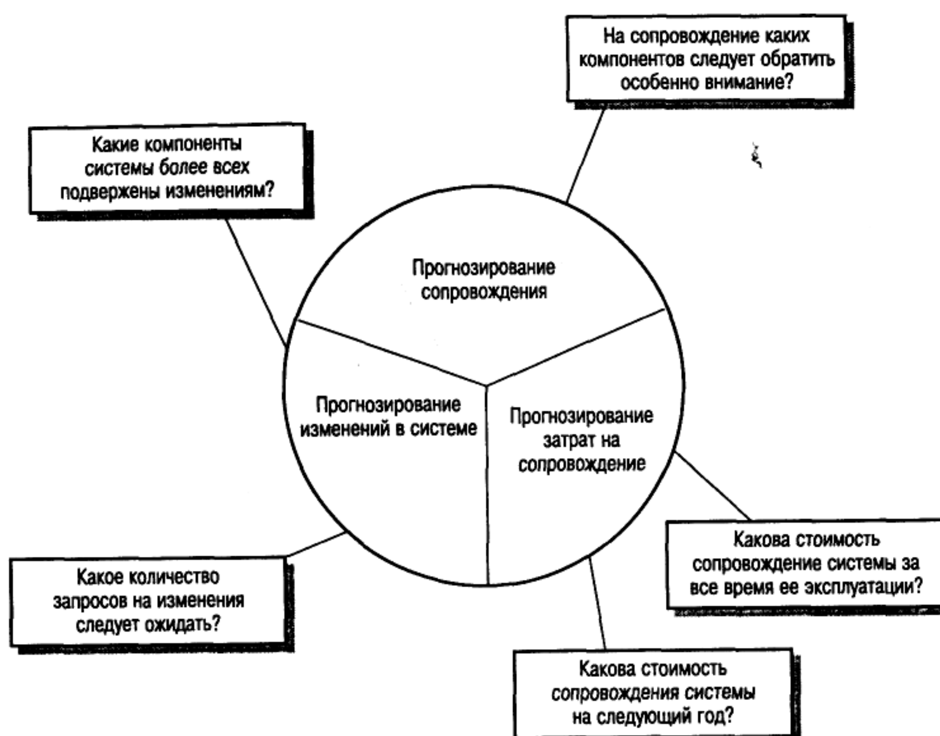

Рис. 17.7. Прогнозирование сопровождения

В работе [18] проведено исследование ряда коммерческих программ, написанных на языке COBOL, с использованием разных методик измерения сложности, включая размер процедур, размер модулей и количество ветвлений, что определяет сложность системы. Сравнивая сложность отдельных системных компонентов с отчетами по сопровождению, исследователи обнаружили, что снижение сложности программирования значительно сокращает расходы на сопровождение системы.

Измерение уровня сложности систем оказалось весьма полезным для выявления тех компонентов систем, которые будут особенно сложны для сопровождения. Результаты анализа ряда системных компонентов [195] показали, что сопровождение часто сосредоточено на обслуживании небольшого количества частей системы, которые отличаются особой сложностью. Поэтому экономически выгодно заменить сложные системные компоненты более простыми их версиями.

После введения системы в эксплуатацию появляются данные, позволяющие прогнозировать дальнейшее сопровождение системы. Перечисленные ниже показатели полезны для оценивания удобства сопровождения.

1. **Количество запросов на корректировку системы**. Возрастание количества отчетов о сбоях в системе означает увеличение количества ошибок, подлежащих исправлению при сопровождении. Это говорит об ухудшении удобства сопровождения.

2. **Среднее время, потраченное на анализ причин системных сбоев и отказов.** Этот показатель пропорционален количеству системных компонентов, в которые требуется внести изменения. Если этот показатель возрастает, система требует многочисленных изменений.

3. **Среднее время, необходимое на реализацию изменений**. Не следует путать этот показатель с предыдущим, хотя они тесно связаны. Здесь

учитывается не время анализа системы по выявлению причин сбоев, а время реализации изменений и их документирования, которое зависит от сложности программного кода. Увеличение этого показателя означает сложность сопровождения.

4. **Количество незавершенных запросов на изменения**. С возрастанием количества таких запросов затрудняется сопровождение системы.

Для определения стоимости сопровождения используется предварительная информация о запросах на изменения и прогнозирование относительно удобства сопровождения системы. В решении этого вопроса большинству менеджеров поможет также интуиция и опыт. В модели определения стоимости СОСОМО 2, описанной в главе 23, предполагается, что для оценки стоимости сопровождения понадобятся сведения об усилиях, потраченных на понимание существующего кода системы и на разработку нового.

17.5. Реинжениринг программного обеспечения

Реинжениринг – это повторная реализация наследуемой системы в целях повышения удобства ее эксплуатации и сопровождения. В это понятие входят разные процессы, среди которых назовем повторное документирование системы, ее реорганизацию и реструктуризацию, перевод системы на один из более современных языков программирования, модификацию и модернизацию структуры и системных данных. При этом функциональность системы и ее архитектура остаются неизменными.

С технической точки зрения реинжениринг – это решение "второго сорта" проблемы системной эволюции. Если учесть, что архитектура системы не изменяется, то сделать централизованную систему распределенной представляется делом довольно сложным. Обычно нельзя

изменить язык программирования старых систем на объектно-ориентированные языки (например, Java или C++). Эти ограничения вводятся для сохранения архитектуры системы.

Однако с коммерческой точки зрения реинжениринг часто принимается за единственный способ сохранения наследуемых систем в эксплуатации. Другие подходы к эволюции системы либо слишком дорогостоящие, либо рискованные. Чтобы понять причины такой позиции, следует рассмотреть проблемы, связанные с наследуемыми системами.

Код эксплуатируемых в настоящее время программных систем чрезвычайно огромен. В 1990 году Улрич (Ulrich, [336]) насчитал 120 млрд. строк исходного кода эксплуатируемых в то время программ. При этом большинство программ были написаны на языке COBOL, который лучше всего подходит для обработки данных в деловой сфере, и на языке FORTRAN. У этих языков достаточно ограниченные возможности в плане структуризации программ, а FORTRAN к тому же отличается ограниченной поддержкой структурирования данных.

Несмотря на постоянную замену подобных систем, многие из них все еще используются. С 1990 года отмечается резкое возрастание использования вычислительной техники в деловой сфере. При грубом подсчете можно говорить о 250 млрд. строк исходного кода, которые нуждаются в сопровождении. Большинство создано отнюдь не с помощью объектно-ориентированных языков программирования, многие из них функционируют все еще на мэйнфреймах.

Программных систем настолько много, что говорить о полной замене или радикальной реструктуризации их в большинстве организаций не приходится. Сопровождение старых систем действительно стоит дорого, однако реинжениринг может продлить время их существования. Как отмечалось в главе 26, реинжениринг систем выгоден в том случае, если система обладает определенной коммерческой ценностью, но дорога в сопровождении. С помощью реинжениринга совершенствуется системная

структура, создается новая документация и облегчается сопровождение системы.

По сравнению с более радикальными подходами к совершенствованию систем реинженеринг имеет два преимущества.

1. **Снижение рисков.** При повторной разработке ПО существуют большие риски – высока вероятность ошибок в системной спецификации и возникновения проблем во время разработки системы. Реинженеринг снижает эти риски.

2. **Снижение затрат**. Себестоимость реинженеринга значительно ниже, чем разработка нового программного обеспечения. В статье [336] приводится пример системы, эксплуатируемой в коммерческой структуре, повторная разработка которой оценивалась в 50 млн. долларов. Для этой системы был успешно выполнен реинженеринг стоимостью всего 12 млн. долларов. Приведенные цифры типичны: считается, что реинженеринг в четыре раза дешевле, чем повторная разработка системы.

Реинженеринг ПО тесно связан с реинженерингом деловых процессов [153, 9*]. Последний означает преобразование бизнес-процессов для снижения количества излишних видов деятельности и повышения эффективности делового процесса. Обычно реинженеринг бизнес-процессов предполагает внедрение новых программ для поддержки деловых процессов или модификацию существующих программ, при этом наследуемые системы существенно зависят от делового процесса. Такую зависимость следует выявлять как можно раньше и устранять, прежде чем начнется планирование каких-либо изменений в самом бизнес-процессе. Поэтому решение о реинженеринге ПО может возникнуть, если наследуемую систему не удается адаптировать к новым деловым процессам путем изменений в обычном сопровождении системы.

Основное различие между реинженерингом и новой разработкой системы связано со стартовой точкой начала работы над системой. При реинженеринге вместо написания системной спецификации "с нуля" старая

система служит основой для разработки спецификации новой системы. В статье [72] традиционная разработка ПО названа *разработкой вперед* (forward engineering), чтобы подчеркнуть различие между ней и реинженирингом. Это различие проиллюстрировано на рис. 17.1. Традиционная разработка начинается с этапа создания системной спецификации, за которой следует проектирование и реализация новой системы. Реинжениринг основывается на существующей системе, которая разработчиками изучается и преобразуется в новую.

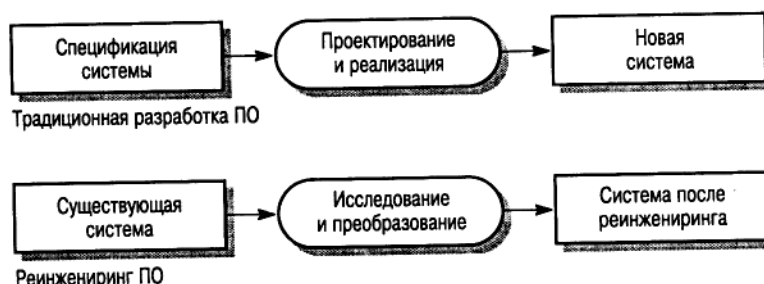

Рис. 17.1. Традиционная разработка и реинжениринг ПО

На рис. 17.2 показан возможный процесс реинжениринга. В начале этого процесса имеем наследуемую систему, а в результате – структурированную и заново скомпонованную версию той же системы. Перечислим основные этапы этого процесса.

1. **Перевод исходного кода**. Конвертирование программы со старого языка программирования на современную версию того же языка либо на другой язык.

2. **Анализ программ**. Документирование структуры и функциональных возможностей программ на основе их анализа.

3. **Модификация структуры программ**. Анализируется и модифицируется управляющая структура программ с целью сделать их более простыми и понятными.

4. **Разбиение на модули.** Взаимосвязанные части программ группируются в модули; там, где возможно, устраняется избыточность. В некоторых случаях изменяется структура системы.

5. **Изменение системных данных**. Данные, с которыми работает программа, изменяются с тем, чтобы соответствовать нововведениям.

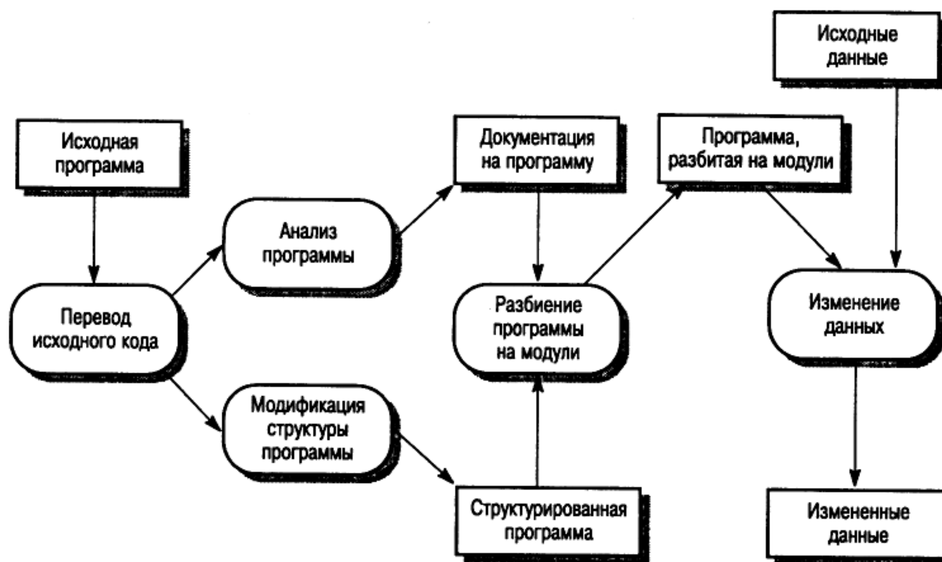

Рис. 17.2. Процесс реинжениринга

При реинжениринге программ необязательно проходить все стадии, показанные на рис. 17.2. Например, не всегда нужно переводить исходный код, если язык программирования, на котором написана программа, все еще поддерживается разработчиком компилятора. Если реинжениринг проводится с помощью автоматизированных средств, то не обязательно восстанавливать документацию на программу. Изменение системных данных необходимо, если в результате реинжениринга изменяется их структура. Однако реструктуризация данных в процессе реинжениринга требуется всегда.

Стоимость реинжениринга обычно определяется объемом выполненных работ. На рис. 17.3 показано несколько различных подходов к процессу реинжениринга и динамика изменения стоимости работ для этих подходов.

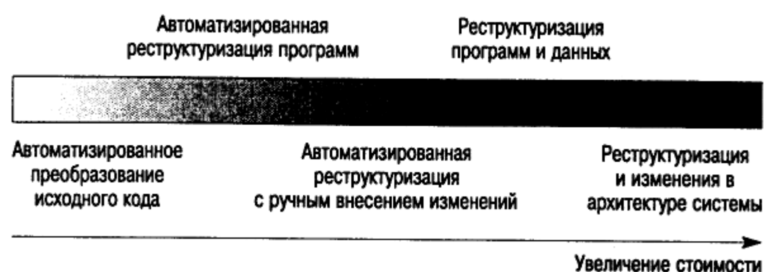

Рис. 17.3. Стоимость реинженеринга

Кроме объема выполняемых работ, есть и другие факторы, обусловливающие стоимость реинжениринга.

1. **Качество программного обеспечения, которое подвергается реинженирингу.** Чем ниже качество программ и их документации (если она есть в наличии), тем выше стоимость реинжениринга.

2. **Наличие средств поддержки процесса реинжениринга**. Обычно реинжениринг экономически выгоден, если применяются CASE-средства для автоматизированного внесения изменений в программы.

3. **Объем необходимого преобразования данных**. Стоимость процесса реинжениринга возрастет при увеличении объема преобразуемых данных.

4. **Наличие необходимых специалистов**. Если персонал, который занимается сопровождением системы, не может выполнить реинжениринг, это также может стать причиной повышения стоимости процесса. Вновь привлеченные специалисты потратят много времени на изучение системы.

Основным недостатком реинжениринга принято считать то, что с его помощью систему можно улучшить только до определенной степени. Например, с помощью реинжениринга невозможно функционально-ориентированную систему сделать объектно-ориентированной. Основные архитектурные изменения или полную реструктуризацию программ невозможно выполнить автоматически, что также увеличивает стоимость реинжениринга. И, несмотря на то что реинжениринг поможет улучшить сопровождение системы, все равно она будет намного хуже в сопровождении, чем новая, созданная с помощью современных методов инженерии ПО.

Контрольные вопросы:

1) Какие стратегические подходы к процессу модернизации ПО существует?
2) Что такое сопровождение?
3) Перечистите виды сопровождения системы.

4) Какие ключевые факторы определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения?
5) Что такое реинжениринг?

Ключевые слова: сопровождение программного обеспечения, эволюция системной архитектуры, реинжиниринг программного обеспечения, стабильность команды разработчиков, ответственность согласно контракту, квалификация специалистов, возраст и структура программы, реинжениринг, снижение рисков, снижение затрат, перевод исходного кода, анализ программ, модификация структуры программ, разбиение на модули, изменение системных данных.

Keywords: *software support, the evolution of the system architecture, software reengineering, the stability of the development team, responsible under the contract, qualified specialists, age and structure of the program, re-engineering, risk reduction, cost reduction, the translation of the source code, program analysis, modification of the structure of programs modularization, the change of system data.*

Kalit so'zlar: *dasturiy ta'minot kuzatuvi, tizim arhitekturasi evolyutsiyasi, dasturiy ta'minot reinjiniringi, ishlabchiqaruvchilar guruhining barqarorligi, shartnomaga muvofiq javobgarlik, mutahassislarning malakasi, dasturning yoshi va strukturasi, chiqim va havf kamayishi, dasturlar tahlili, modullarga bo'linish, tizim ma'lumotlarini o'zgartirish.*

Упражнения

1. В каких случаях необходимо заменить старое ПО новым вместо его реинжениринга?
2. Сравните управляющие структуры (циклы и условные операторы) в двух любых известных вам языках программирования. Кратко опишите процесс перевода управляющих структур с одного языка на другой.
3. Переведите процедуру, приведенную в листинге 28.4, в эквивалентную структурированную, выполнив все необходимые для этого действия.
4. Напишите ряд указаний по определению модулей в неструктурированной программе.
5. Предложите значащие имена для переменных, которые используются в листинге 28.4, и создайте словарь данных для этих имен.
6. Какие трудности возникнут при переносе данных из СУБД одного типа в СУБД другого типа (например, из иерархической базы данных в реляционную или из реляционной в объектно-ориентированную)?
7. Объясните, почему невозможно восстановить системную спецификацию путем автоматического анализа исходного кода системы.
8. Приведите примеры для описания проблем, связанных с нарушением данных при их чистке.

9. Проблема 2000 года (когда даты представлены с помощью двух цифр) стала одной из основных причин для многих организаций внести коррективы в сопровождение программ. Как это повлияло на процесс изменения данных?

ГЛАВА 18. УПРАВЛЕНИЕ КОНФИГУРАЦИЯМИ

Управление конфигурацией – это процесс разработки и применения стандартов и правил по управлению эволюцией программных продуктов. Эволюционирующие системы нуждаются в управлении по той простой причине, что в процессе их эволюции создается несколько версий одних и тех же программ. В эти версии обязательно вносятся некоторые изменения, исправляются ошибки предыдущих версий; кроме того, версии могут адаптироваться к новым аппаратным средствам и операционным системам. При этом в разработке и эксплуатации могут одновременно находиться сразу несколько версий. Поэтому нужно четко отслеживать все вносимые в систему изменения.

Процедуры управления конфигурацией регулируют процессы регистрации и внесения изменений в систему с указанием измененных компонентов, а также способы идентификации различных версий системы. Средства управления конфигурацией применяют для хранения всех версий системных компонентов, для компоновки из этих компонентов системы и для отслеживания поставки заказчикам разных версий системы.

Как указывалось в главе 24, управление конфигурацией нередко рассматривается как часть общего процесса управления качеством. Поэтому иногда одно и то же лицо может отвечать как за управление качеством, так и за управление конфигурацией. Но обычно разрабатываемая программная система сначала контролируется командой по управлению качеством, которая проверяет ПО на соответствие определенным стандартам качества. Далее ПО передается команде по управлению конфигурацией, которая контролирует изменения, вносимые в систему.

Существует много причин, объясняющих наличие разных конфигураций одной и той же системы. Различные версии создаются для разных компьютеров или операционных систем, включающих специальные

функции, нужные заказчикам, и т.д. (рис. 18.1). Менеджеры по управлению конфигурацией обязаны следить за различиями между разными версиями, чтобы обеспечить возможность выпуска следующих вариантов системы и своевременную поставку нужных версий соответствующим заказчикам.

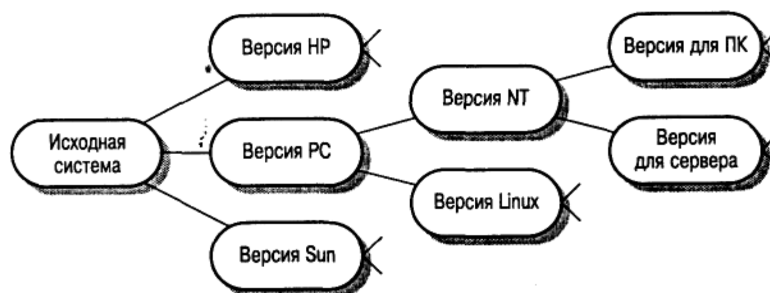

Рис. 18.1. Семейство версий системы

Процесс управления конфигурацией и связанная с ним документация должны подчиняться определенным стандартам. В качестве примера можно привести стандарт IEEE 828-1983, определяющий составление планов управления конфигурацией. Каждая организация должна иметь справочник, в котором указаны эти стандарты, либо они должны входить в общий справочник стандартов качества. Общенациональные или международные стандарты могут быть также использованы как основа для разработки детализированных специальных норм и стандартов для конкретных организаций. За основу можно взять любой тип стандарта, поскольку все они содержат описания однотипных процессов. Для сертификации качества своих программных продуктов организация должна придерживаться официальных стандартов управления конфигурацией, которые приведены в стандартах ISO 9000 [178] и в модели оценки уровня развития SEI [273].

При традиционной разработке ПО в соответствии с каскадной моделью (см. главу 3) разрабатываемая система попадает в группу по управлению конфигурацией уже после полного завершения разработки и тестирования ПО. Именно такой подход лежит в основе стандартов управления конфигурацией, которые, в свою очередь, обусловливают необходимость использования для разработки систем моделей, подобных каскадной [38]. Поэтому упомянутые стандарты не в полной мере подходят при

использовании таких методов разработки ПО, как эволюционное прототипирование и пошаговая разработка. В этой ситуации некоторые организации изменили подход к управлению конфигурацией, сделав возможным параллельную разработку и тестирование системы. Такой подход основан на регулярной (иногда ежедневной) сборке системы из ее компонентов.

1. Устанавливается время, к которому должна быть завершена поставка компонентов системы (например, к 14.00). Программисты, работающие над новыми версиями компонентов, должны предоставить их к указанному времени. Работу над компонентами не обязательно завершать, достаточно представить основные рабочие функции для проведения тестирования.

2. Создается новая версия системы с новыми компонентами, которые компилируются и связываются в единую систему.

3. После этого система попадает к группе тестирования. В то же время разработчики продолжают работу над компонентами, добавляя новые функции и исправляя ошибки, обнаруженные в ходе предыдущего тестирования.

4. Дефекты, замеченные при тестировании, регистрируются, соответствующий документ пересылается разработчикам. В следующей версии компонента эти дефекты будут учтены и исправлены.

Основным преимуществом ежедневной сборки системы является возможность выявления ошибок во взаимодействиях между компонентами, которые в противном случае могут накапливаться. Более того, ежедневная сборка системы поощряет тщательную проверку компонентов. Разработчики работают под давлением: нельзя прерывать сборку систем и поставлять неисправные версии компонентов. Поэтому программисты неохотно поставляют новые версии компонентов, если они не были предварительно тщательно проверены. Таким образом, на тестирование и исправление ошибок ПО уходит меньше времени.

Для ежедневных сборок системы требуется достаточно строгое управление процессом изменений, позволяющее отслеживать проблемы, которые выявляются и исправляются в ходе тестирования. Кроме того, в результате возникает множество версий компонентов системы, для управления которыми необходимы средства управления конфигурацией.

18.1. Планирование управления конфигурацией

В плане управления конфигурацией представлены стандарты, процедуры и мероприятия, необходимые для управления. Отправной точкой создания такого плана является набор общих стандартов по управлению конфигурацией, применяемых в организации-разработчике ПО, которые адаптируются к каждому отдельному проекту. Обычно план управления конфигурацией имеет несколько разделов.

1. Определение контролируемых объектов, подпадающих под управление конфигурацией, а также формальная схема определения этих объектов.
2. Перечень лиц, ответственных за управление конфигурацией и за поставку контролируемых объектов в команду по управлению конфигурацией.
3. Политика ведения управления конфигурацией, т.е. процедуры управления изменениями и версиями.
4. Описание форм записей о самом процессе управления конфигурацией.
5. Описание средств поддержки процесса управления конфигурацией и способов их использования.
6. Определение базы данных конфигураций, применяемой для хранения всей информации о конфигурациях системы.

Распределение обязанностей по конкретным исполнителям является важной частью плана. Необходимо четко определить ответственных за

поставку каждого документа или компонента ПО для команд по управлению качеством и конфигурацией. Лицо, отвечающее за поставку какого-либо документа или компонента, должно отвечать и за их разработку. Для упрощения процедур согласования удобно назначать менеджеров проекта или ведущих специалистов команды разработчиков ответственными за все документы, созданные под их руководством.

18.2. Определение конфигурационных объектов

В процессе разработки больших систем создаются тысячи различных документов. Большинство из них – это текущие рабочие документы, связанные с различными этапами разработки ПО. Есть также внутренние записки, протоколы заседания рабочих групп, проекты планов и предложений и т.п. Такие документы представляют разве что исторический интерес и не нужны для дальнейшего сопровождения системы.

Для планирования процесса управления конфигурацией необходимо точно определить, какие проектные элементы (или классы элементов) будут объектами управления. Такие элементы называются **конфигурационными элементами**. Как правило, они представляют собой официальные документы. Конфигурационными элементами обычно являются планы проектов, спецификации, схемы системной архитектуры, программы и наборы тестовых данных. Кроме того, управлению подлежат все документы, необходимые для будущего сопровождения системы.

В процессе управления конфигурацией каждому документу необходимо присвоить уникальное имя, причем отображающее связи с другими документами. Для этого используется иерархическая система имен, где они имеют, например, такой вид:

PLC-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ/КОД

PLC-TOOLS/ПРАВКА/СПРАВКА/ЗАПРОС/ОКНО_СПРАВКИ/FR-l

Начальная часть имени – это название проекта PLC-TOOLS. В проекте разрабатываются четыре отдельных средства (рис. 18.2). Имя средства используется в следующей части имени. Каждое средство создается из именованных модулей. Такое разбиение продолжается до тех пор, пока не появится ссылка на официальный документ базового уровня. Листья дерева иерархии документов являются официальными документами проекта. На рис. 18.2 показано, что для каждого объекта требуется три формальных документа. Это описание объектов (документ ОБЪЕКТЫ), код компонента (документ КОД) и набор тестов для этого кода (документ ТЕСТЫ).

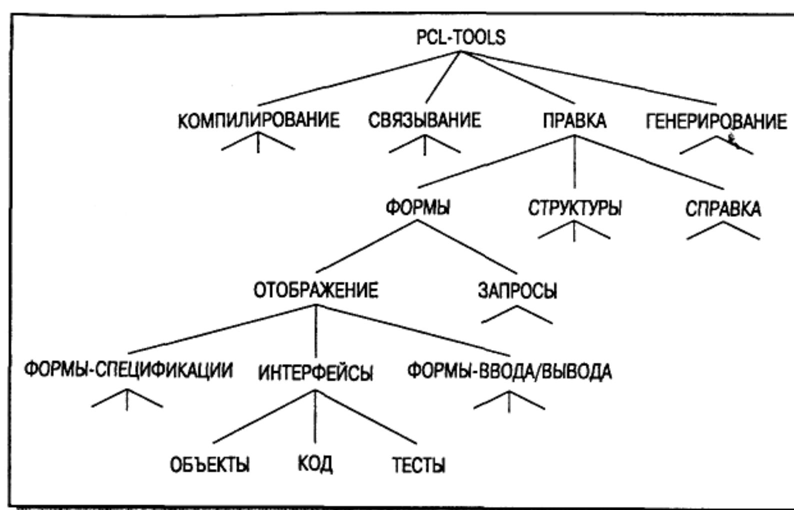

Рис. 18.2. Иерархия конфигурации

Подобные схемы имен основаны на структуре проекта, когда имена соотносятся с соответствующими проектными компонентами. Такой подход к именованию документов порождает определенные проблемы. Например, снижается возможность повторного использования компонентов. Обычно в таких случаях из схемы берутся копии компонентов, которые можно повторно использовать, и переименовываются в соответствии с новой областью применения. Другие проблемы могут появиться, если эта схема именования документов используется как основа структуры хранения компонентов. Тогда пользователь должен знать названия документов, чтобы найти нужные компоненты, при этом не все документы одного типа (например, по проектированию) хранятся в одном месте. Также могут

встретиться трудности при установлении соответствия между схемой имен и схемой идентификации, используемой в системе управления версиями.

18.3. База данных конфигураций

Такая база данных используется для хранения всей информации о системных конфигурациях. Основными функциями базы данных конфигураций являются поддержка оценивания влияния планируемых изменений в системе и предоставление информации о процессе управления конфигурацией. Задание структуры базы данных конфигураций, определение процедур записи и поиска информации в этой базе данных – все это является частью процесса планирования управления конфигурацией.

Информация, заключенная в базе данных конфигураций, должна помочь ответить на ряд вопросов, среди которых основными и часто запрашиваемыми будут следующие:

- Каким заказчикам поставлена определенная версия системы?
- Какие аппаратные средства и какая операционная система необходимы для работы данной версии системы?
- Сколько было выпущено версий данной системы и когда?
- На какие версии системы повлияют изменения, вносимые в определенный компонент?
- Сколько запросов на изменения было реализовано в данной версии?
- Какое количество ошибок было зарегистрировано в данной версии системы?

В идеале база данных конфигураций должна быть объединена с системой управления версиями, которая создается для хранения и управления формальными проектными документами. Такой подход, к тому же поддерживаемый некоторыми интегрированными CASE-средствами, предоставляет возможность связать изменения, вносимые в систему, и с

документами, и с теми компонентами, которые подверглись изменениям. В этом случае упрощается поиск измененных компонентов, *поскольку* установлены связи между документами (например, между документами по системной архитектуре и кодом программ) и этими связями можно управлять.

Однако многие организации вместо использования интегрированных CASE-средств для управления конфигурацией рассматривают базу данных конфигураций как отдельную систему. Конфигурационные элементы могут храниться в отдельных файлах или в системе управления версиями, например RCS (известная система управления версиями для Unix [335]). В этом случае в базе данных конфигураций хранится информация о конфигурационных элементах и ссылки на имена соответствующих файлов в системе управления версиями. Несмотря на относительную дешевизну и гибкость такого подхода, основным недостатком его является то, что конфигурационные элементы могут быть изменены без внесения необходимых записей в базу данных. Поэтому нельзя гарантировать, что в базе данных конфигураций содержится обновленная и корректная информация о состоянии системы.

18.4. Управление изменениями

Изменения в больших программных системах неизбежны. Как отмечалось в предыдущих главах, в течение жизненного цикла системы изменяются пользовательские и системные требования, а также приоритеты и запросы организаций. Процесс управления изменениями и соответствующие CASE-средства предназначены для того, чтобы зарегистрировать изменения и внести их в систему наиболее эффективным способом.

Процесс управления изменениями (листинг 18.1) начинается после того, как программное обеспечение или соответствующая документация передается команде по управлению конфигурацией. Он может начаться во время тестирования системы или даже после ее поставки заказчику.

Процедуры управления изменениями создаются для обеспечения корректного анализа необходимости изменений и их стоимости, а также для контроля за вносимыми изменения.

Листинг 18.1. Процесс управления изменениями

Запрос на изменение, заполнение формы запроса

Анализ запроса

if изменение допустимо **then**

Оценка способа внесения изменения

Оценка стоимости изменения

Запись запроса в базу данных

Передача запроса группе контроля за изменениями

if запрос принят **then**

repeat

внесение изменений в ПО

регистрация изменений

передача измененного ПО группе управления качеством

until качество ПО соответствует нормам

создание ноной версии системы

else

запрос на изменение отвергнут

else

запрос на изменение отвергнут

Первым этапом в процессе управления изменениями является заполнение формы запроса на изменения, в которой указываются те изменения, которые планируется внести в систему. В форме запроса также приводятся рекомендации относительно изменений, предварительная оценка затрат и даты запроса, его утверждения, внедрения и проверки. Также форма может включать раздел, в котором указывается способ выполнения изменения. Запросы на изменения регистрируются в базе данных

конфигураций. Таким образом, команда управления конфигурациями может следить за выполнением изменений, а также контролировать изменения определенных программных компонентов.

Во врезке 18.1 приведен пример заполненной формы запроса на изменения. Такая форма обычно определяется на этапе планирования управления конфигурацией. По условиям некоторых контрактов (например, в контрактах с правительственными органами), такая форма должна соответствовать стандартам заказчика.

Врезка 18.1. Форма запроса на изменения

Проект. Proteus/PCL-Tools **Номер.** 23/94

Лицо, заполняющее запрос. Соммервиль Дата. 1/12/98

Требуемые изменения. После выбора компонента структуры необходимо отображение имени файла, в котором он хранится.

Лицо, осуществляющее анализ запроса. Г. Дин

Дата анализа. 10/12/98

Изменяемые компоненты. Display-lcon.Select, Display-lcon.Display

Связанные компоненты. FileTable

Оценка изменения. Изменение достаточно легко выполнить из-за наличия таблицы имен файлов. Требуется вставить соответствующее поле. Изменений связанных компонентов не требуется.

Уровень приоритета. Низкий

Выполнение изменения.

Оценка затрат. 0,5 дня

Дата передачи в группу контроля за изменениями. 15/12/98

Дата принятия решения группой контроля за изменениями. 1/2/99

Решение группы контроля за изменениями. Принять изменение. Будет реализовано в версии 2.1.

Кто вносит изменения.

Дата внесения изменений.

Дата передачи запроса в группу управления качеством.

Решение группы управления качеством.

Дата передачи запроса в группу по управлению конфигурацией.

Примечание.

Сразу после представления заполненной формы запроса проводится проверка необходимости и допустимости изменения. Это объясняется тем, что некоторые изменения вызваны не ошибками в программе, а неправильным пониманием требований, другие могут дублировать исправление ранее обнаруженных ошибок. Если в процессе проверки выявляется, что изменение недопустимо, повторяется или уже было рассмотрено, то изменение отклоняется. Лицу, представившему запрос на изменение, объясняется причина отказа.

Для принятых изменений начинается вторая стадия – оценка изменений и предварительное определение стоимости. Сначала следует проверить влияние изменения на всю систему. Для этого делается технический анализ способа внесения изменения. Затем определяется стоимость внесения изменения в определенные компоненты, что регистрируется в форме запроса. В процессе оценивания полезна база данных конфигураций с информацией о взаимосвязях между компонентами, благодаря чему есть возможность оценить влияние изменений на другие компоненты системы.

Все изменения, кроме тех, которые относятся к исправлению мелких недоработок, должны быть переданы в группу контроля за изменениями, где принимается решение о принятии изменения либо отказе. Эта группа оценивает воздействие изменения не с технической, а скорее с организационной или стратегической точек зрения. Во внимание принимаются такие соображения, как экономическая выгодность изменения и организационные факторы, которые оправдывают необходимость изменения.

Группа контроля за изменениями состоит из лиц, на которых возлагается ответственность за решения о внесении изменений. Такие группы со структурой, включающей старшего менеджера компании-заказчика и сотрудников фирмы-разработчика, обязательны при выполнении военных проектов. Для небольших или среднего размера проектов в эту группу может входить только менеджер проекта и один-два инженера, которые не занимались разработкой данного ПО. В отдельных случаях допускается участие аналитика по изменениям, который дает рекомендации относительно того, оправданы эти изменения либо нет.

После принятия решения о внесении изменений программная система для внесения изменений передается разработчикам или команде по сопровождению системы. По окончании этой процедуры система обязательно должна пройти проверку на правильность внесения изменений. После этого именно команда по управлению конфигурацией, а не разработчики, займется выпуском новой версии.

Изменение каждого компонента системы должно регистрироваться. Таким образом создается история компонента. Самый лучший способ для этого – создавать стандартизированные комментарии в начале кода компонента (листинг 18.2), где содержатся ссылки на запросы изменений данного компонента. Для составления отчетов об изменениях компонента и обработки их историй используются специальные средства.

Листинг 18.2. Заголовок компонента

```
// Проект PROTEUS (ESPRIT 6087)
//
// PCL-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ
//
// Объект: PCL-Tool-jbesc
// Автор: Г.Дин
// Дата создания: 10 ноября 1998 г.
```

```
//
// © Lancaster University 1998
//
// История изменений
// Версия   Кто внес изменения Дата       Изменение      Причина
// 1.0.     Дж. Джонс          1/12/1998  Добавление     Предложена
//                                        заголовка      группой по
//                                                       управлению
//
конфигурацией
// 1.1.     Г. Дин             9/4/1999                  Новое    поле
Запрос R07/99
```

18.5. Управление версиями и выпусками

Управление версиями и выпусками ПО необходимо для идентификации и слежения за всеми версиями и выпусками системы. Менеджеры, отвечающие за управление версиями и выпусками ПО, разрабатывают процедуры поиска нужных версий системы и следят за тем, чтобы изменения не осуществлялись произвольно. Они также работают с заказчиками и планируют время выпуска следующих версий системы. Над новыми версиями системы должна работать команда по управлению конфигурацией, а не разработчики, даже если новые версии предназначены только для внутреннего использования. Только в том случае, если информация об изменениях в версиях вносится исключительно командой по управлению конфигурацией, можно гарантировать согласованность версий.

Версией системы называют экземпляр системы, имеющий определенные отличия от других экземпляров этой же системы. Новые версии могут отличаться функциональными возможностями, эффективностью или исправлениями ошибок. Некоторые версии имеют

одинаковую функциональность, однако разработаны под различные конфигурации аппаратного или программного обеспечения. Если отличия между версиями незначительны, они называются **вариантами** одной версии.

Выходная версия (release) системы – это та версия, которая поставляется заказчику. В каждой выходной версии либо обязательно присутствуют новые функциональные возможности, либо она разработана под новую платформу. Количество версий обычно намного превышает количество выходных версий, поскольку версии создаются в основном для внутреннего пользования и не поставляются заказчику.

В настоящее время для поддержки управления версиями разработано много разнообразных CASE-средств (см. раздел 18.5). С помощью этих средств осуществляется управление хранением каждой версии и контроль за допуском к компонентам системы. Компоненты могут извлекаться из системы для внесения в них изменений. После введения в систему измененных компонентов получается новая версия, для которой с помощью системы управления версиями создается новое имя.

18.6. Идентификация версий

Любая большая программная система состоит из сотен компонентов, каждый из которых может иметь несколько версий. Процедуры управления версиями должны четко идентифицировать каждую версию компонента. Существует три основных способа идентификации версий.

1. **Нумерация версий**. Каждый компонент имеет уникальный и явный номер версии. Эта схема идентификации используется наиболее широко.

2. **Идентификация, основанная на знамениях атрибутов**. Каждый компонент идентифицируется именем, которое, однако, не является уникальным для разных версий, и набором значений атрибутов, разных для

каждой версии компонента [110]. Здесь версия компонента идентифицируется комбинацией имени и набора значений атрибутов.

3. **Идентификация на основе изменений**. Каждая версия системы именуется так же, как в способе идентификации, основанном на значениях атрибутов, плюс ссылки на запросы на изменения, которые реализованы в данной версии системы [244]. Таким образом, версия системы идентифицируется именем и теми изменениями, которые реализованы в системных компонентах.

18.7. Нумерация версий

По самой простой схеме нумерации версий к имени компонента или системы добавляется номер версии. Например, Solaris 2.6 обозначает версию 2.6 системы Solaris. Первая версия обычно обозначается 1.0, последующими версиями будут 1.1, 1.2 и т.д. На каком-то этапе создается новая выходная версия – версия 2.0, нумерация этой версии начинается заново – 2.1, 2.2 и т.д. Эта линейная схема нумерации основана на предположении о последовательности создания версий. Подобный подход к идентификации версий поддерживается многими программными средствами управления версиями, например RCS (см. раздел 18.5).

На рис. 18.3 графически проиллюстрирован описанный способ нумерации версий. Стрелки на рисунке проведены от исходной версии к новой, которая создается на ее основе. Отметим, что последовательность версий не обязательно линейная – версии с последовательными номерами могут создаваться на основе разных базовых версий. Например, на рис. 18.3 видно, что версия 2.2 создана на основе версии 1.2, а не версии 2.1. В принципе каждая существующая версия может служить основой для создания новой версии системы.

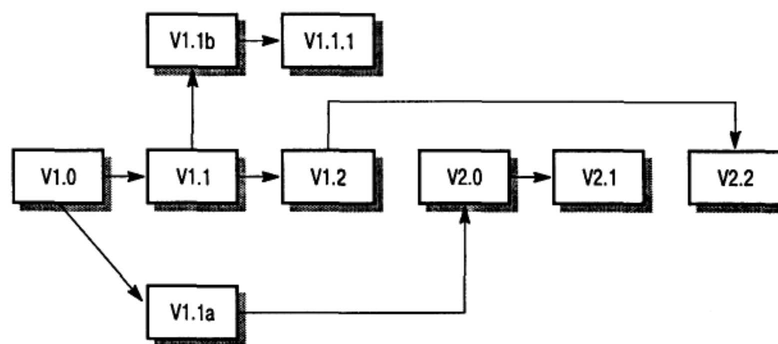

Рис. 18.3. Структура системных версий

Данная схема идентификации версий достаточно проста, однако она требует довольно большого количества информации для сопоставления версий, что позволяло бы отслеживать различия между версиями и связи между запросами на изменения и версиями. Поэтому поиск отдельной версии системы или компонента может быть достаточно трудным, особенно при отсутствии интеграции между базой данных конфигураций и системой хранения версий.

18.8. Сборка системы

Сборкой системы называют процесс компиляции и связывания программных компонентов в единую исполняемую программу. Перед сборкой системы полезно ответить на следующие вопросы.

1. Все ли компоненты, составляющие систему, включены в инструкцию по сборке?

2. Каковы версии компонента, перечисленные в инструкции по сборке?

3. Доступны ли все необходимые файлы данных?

4. Если на файлы данных используются ссылки внутри компонентов, то каковы имена этих файлов в выходной версии?

5. Доступны ли нужные версии компилятора и других необходимых средств? Действующие версии программных средств могут быть несовместимы с более старыми версиями, которые применялись при разработке системы.

В настоящее время существует много средств управления конфигурацией, автоматизирующих процесс сборки системы. Команда управления конфигурацией пишет сценарий, в котором определены зависимости между различными компонентами системы. В нем также указаны средства компилирования и связывания компонентов системы. Средства компоновки интерпретируют сценарий сборки системы и вызывают программы, необходимые для сборки исполняемой системы. Процесс сборки системы представлен на рис. 18.4.

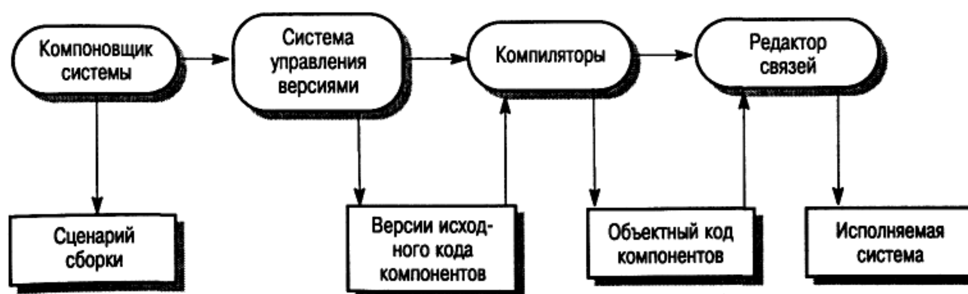

Рис. 18.4. Сборка системы

В сценарии сборки указаны зависимости между компонентами, поэтому компоновщик системы сам принимает решение, когда перекомпилировать компоненты, а когда можно многократно использовать существующий объектный код. Зависимости в сценарии сборки указаны в основном как зависимости между файлами, содержащими исходный код компонентов. Однако, если файлов с исходным кодом разных версий много, возникает проблема выбора нужных файлов. Проблема усугубляется, если файлы исходного и объектного кода имеют одинаковые имена (но, конечно, с разными расширениями).

Чтобы избежать трудностей, связанных с зависимостью физических файлов, было разработано несколько экспериментальных систем, основанных на языках описания модулей [320]. В них используется описание логической структуры ПО и схемы зависимостей между файлами, содержащими компоненты исходного кода. Такой подход снижает количество ошибок и приводит к более понятным описаниям процесса сборки системы.

Контрольные вопросы:

1) Дайте определение управлению конфигурацией.
2) Перечислите разделы плана управления конфигурацией.
3) Что такое начальная часть имени?
4) Что называют версией?
5) Какие три основных способа идентификации версий существует?

Ключевые слова: *управление конфигурацией конфигурационные элементы; начальная часть имени; версия; выходная версия; нумерация версий; идентификация, основанная на знамениях атрибутов; идентификация на основе изменений.*

Keywords: *configuration management configuration items; the initial part of the name; version; output version; numbering versions; identification based on the signs of attributes; identification on the basis of the changes.*

Kalit so'zlar: *konfiguratsiyani boshqarish, konfiguratsion elementlar, nomning bosh qismi, yangi naql, atributlar nomiga asoslangan identifikatsiya, o'zgarishlarga asoslangan identifikatsiya.*

Упражнения

1. Объясните, почему в системе управления конфигурацией для идентификации документов не используются названия документов. Предложите схему идентификации документов, которую можно было бы использовать во всех проектах вашей организации.

2. Используя модель "сущность-связь" или объектно-ориентированный подход (см. главу 7), разработайте модель базы данных конфигураций, которая должна содержать информацию о системных компонентах, их версиях, выходных версиях системы и об изменениях, реализованных в системе. База данных должна обладать следующими функциями:

- извлечение всех версий или отдельной указанной версии компонента;
- извлечение последней по времени изменения версии компонента;
- поиск запросов на изменения, которые были реализованы в указанной версии системы;
- определение версий компонентов, включенных в указанную версию системы;
- извлечение выходной версии системы, определяемой по дате выпуска или по имени заказчика, которому она поставлена.

3. С помощью диаграммы потока данных представьте модель управления изменениями, которую можно было бы применить в большой организации, занимающейся разработкой

ПО для внешних заказчиков. Запросы на изменения могут поступать как от внешних, так и от внутренних источников.

4. Опишите трудности, которые могут встретиться при сборке системы. В частности, рассмотрите проблемы сборки системы на хост-компьютере.

5. Со ссылкой на систему сборки объясните, почему иногда необходимо сохранять устаревшие компьютеры, на которых разрабатывались большие программные системы.

6. При сборке систем часто возникает сложная проблема, состоящая в том, что имена физических файлов встроены в системный код и используемая файловая структура отличается от файловой структуры на конечной машине, где будет установлена система. Составьте руководство для программистов, которое поможет избежать этой и подобных проблем при сборке систем.

7. Приведите пять факторов, которые необходимо учитывать при сборке выходных версий больших программных систем.

8. Опишите два способа оптимизации процесса сборки системы из ее компонентов с помощью соответствующих CASE-средств компоновки систем.

ЛИТЕРАТУРА

1*. Андон А.И., Яшунин А.Е., Резниченко А.А. Логические модели интеллектуальных информационных систем. – К.: Наук, думка, 1999.

2*. Андон Ф.И., Лаврищева Е.М. Методы инженерии распределенных компьютерных систем. – К.: Наук, думка, 1997.

3*. Бабенко Л.П., Лавріщева К.М. Основи програмної шженерії. Навч. Посіб. – К.: Т-во "Знания", 2001.

4*. Бен-Ари М. Языки программирования. Практический сравнительный анализ. – М.: Мир, 2000.

5*. Боггс Ч., Боггс М. UML Rational. - М.: ЛОРИ, 2000.

6*. Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998.

7*. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2000.

8*. Гласе Г. Руководство по надежному программированию. – М.: Финансы и статистика, 1982.

9*. Калянов Г.Н. CASE-технодогии. Консалтинг при автоматизации бизнес-процессов, 2-е изд. – М.: Горячая линия – Телеком, 2000.

10*. Калянов Г.Н. Методы и средства системного структурного анализа и проектирования. – М.: НИВЦ МГУ, 1995.

11*. Кантор М. Управление программными проектами. – М.: Издат. дом "Вильямс", 2002.

12*. Коллинз Г., Блей Д. Структурные методы разработки систем: от стратегического планирования до тестирования. – М.: Финансы и статистика, 1986.

13*. Коуд П., Норт Д., Мейфидд М. Объектные модели. Стратегии, шаблоны и приложения. – М.: ЛОРИ, 1999.

14*. Кратчен Ф. Введение в Rational Unified Process, 2-е изд. – М.: Издат. дом "Вильямс", 2002.

15*. Кулаков А.Ф. Оценка качества программ ЭВМ. – К.: Техніка, 1984.

16*. Лаврищева Е.М., Грищенко В.Н. Сборочное программирование. – К.: Наук, думка, 1991.

17*. Ларман К. Применение UML и шаблонов проектирования, 2-е изд. – М.: Издат. дом "Вильямс", 2001.

18*. Лефингвел Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. - М.: Издат. дом "Вильямс", 2002.

19*. Липаев В.В. Документирование и управление конфигурацией программных средств. Методы и стандарты. – М.: СИНТЕГ, 1998.

20*. Липаев В.В. Надежность программных средств. - М.: СИНТЕГ, 1998.

21*. Липаев В.В. Отладка сложных программ. – М.: Энергоатомиздат, 1993.

22*. Липаев В.В. Тестирование программ. – М.: Радио и связь, 1986.

23*. Липаев В.В. Управление разработкой программных комплексов. – М.: Финансы и статистика, 1993.

24*. Марка Д.А., Мак-Гоуэн К. Методология структурного анализа и проектирования. - М.: Мета Технология, 1993.

25*. Мацяшек Л. А. Анализ требований и разработка информационных систем с использованием UML. - М.: Издат. дом "Вильяме", 2002.

26*. Мороз Г.Б., Лаврищева Е.М. Модели роста надежности программного обеспечения. - К., 1992. - (Препринт / АНУ; Ин-т кибернетики ; 92-38).

27*. Слама Д., Гарбис Дж., Перри Р. Корпоративные системы на основе CORBA. - М.: Издат. дом "Вильямс", 1999.

28*. Спирли Э. Кооперативные хранилища данных. Планирование, разработка, реализация. Т. 1. – М.: Издат. дом "Вильямс", 2001.

29*. Тейер Т., Липов М., Нельсон Э. Надежность программного обеспечения. – М.: Мир, 1981.

30*. Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. – М.: Мир, 1999.

31*. Фокс Д. Программное обеспечение и его разработка. – М.: Мир, 1985.

32*. Шаллоуей А., Тротт Дж. Р. Шаблоны проектирования. Новый подход к объектно-ориентированной разработке. – М.: Издат. дом "Вильямс", 2002.

33*. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. – К.: Диалектика, 1993.

34*. Элиенс А. Принципы объектно-ориентированной разработки программ, 2-е изд. – М.: Издат. дом "Вильямс", 2002.

35*. Юдицкий С.А., Барон Ю.Л., Жукова Г.Н. Построение и анализ логического портрета сложных систем. – М.: ИПУ, 1997.