

А. М. Вендров

**ПРОЕКТИРОВАНИЕ
ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ
ЭКОНОМИЧЕСКИХ
ИНФОРМАЦИОННЫХ
СИСТЕМ**

Второе издание,
переработанное и дополненное

РЕЦЕНЗЕНТЫ:

**Кафедра проектирования экономических
информационных систем**

Московского государственного университета
экономики, статистики и информатики (МЭСИ);

Г.Н.Калянов,

доктор технических наук, профессор,
ведущий сотрудник ИПУ РАН

Вендров А.М. В29

Проектирование программного обеспечения экономических информационных систем: Учебник. — 2-е изд., перераб. и доп. - М.: Финансы и статистика, 2005. - 544 с: ил.

ISBN 5-279-02937-8

Описаны процессы, модели и стадии жизненного цикла программного обеспечения (ПО.) экономических информационных систем. Приведены структурный и объектно-ориентированный подходы к проектированию ПО. Отражено применение стандартного языка объектно-ориентированного моделирования UML. Рассмотрены функции и компоненты CASE-средств и их практическое воплощение в наиболее развитых программных продуктах. В новом издании (1-е изд. - 2000 г.) улучшена структура учебника, добавлены новые разделы и примеры.

Для студентов, обучающихся по специальностям «Прикладная информатика (по областям)» и «Прикладная математика». Может быть использован студентами и преподавателями других специальностей, а также разработчиками и пользователями систем ПО.

ОГЛАВЛЕНИЕ

<u>Предисловие</u>	9
<u>Введение</u>	12
<u>Глава 1. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ</u>	43
<u>1.1. Нормативно-методическое обеспечение создания ПО</u>	43
<u>1.2. Стандарт жизненного цикла ПО</u>	45
<u>1.2.1. Основные процессы ЖЦ ПО</u>	48
<u>1.2.2. Вспомогательные процессы ЖЦ ПО</u>	56
<u>1.2.3. Организационные процессы ЖЦ ПО</u>	62
<u>1.2.4. Взаимосвязь между процессами ЖЦ ПО</u>	64
<u>1.3. Модели жизненного цикла ПО</u>	67
<u>1.3.1. Каскадная модель ЖЦ</u>	70
<u>1.3.2. Итерационная модель жизненного цикла</u>	76
<u>1.4. Сертификация и оценка процессов создания ПО</u>	84
<u>1.4.1. Понятие зрелости процессов создания ПО.</u> <u> Модель оценки зрелости СММ</u>	84
<u>1.4.2. Методика SPMN</u>	99
<u>1.5. Пример процесса «Управление требованиями»</u>	107
<u>1.6. Пример процесса «Управление конфигурацией ПО»</u>	115
<u>Глава 2. МЕТОДИЧЕСКИЕ АСПЕКТЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ</u>	120
<u>2.1. Общие принципы проектирования систем</u>	120
<u>2.2. Визуальное моделирование</u>	124
<u>2.3. Структурные методы анализа и проектирования ПО</u>	130

2.3.1. Метод функционального моделирования SADT(IDEFO)133
2.3.2. Метод моделирования процессов IDEF3152
2.3.3. Моделирование потоков данных160
2.3.4. Количественный анализ диаграмм IDEF0 и DFD170
2.3.5. Сравнительный анализ SADT-моделей и диаграмм потоков данных171
2.3.6. Моделирование данных175
2.4. Объектно-ориентированные методы анализа и проектирования ПО187
2.4.1. Основные принципы построения объектной модели188
2.4.2. Основные элементы объектной модели191
2.5. Унифицированный язык моделирования UML204
2.5.1. Диаграммы вариантов использования206
2.5.2. Диаграммы взаимодействия216
2.5.3. Диаграммы классов220
2.5.4. Диаграммы состояний223
2.5.5. Диаграммы деятельности227
2.5.6. Диаграммы компонентов230
2.5.7. Диаграммы размещения232
2.5.8. Механизмы расширения UML234
2.5.9. Количественный анализ диаграмм UML238
2.6. Образцы240
2.7. Сопоставление и взаимосвязь структурного и объектно-ориентированного подходов247
Глава 3. МОДЕЛИРОВАНИЕ БИЗНЕС-ПРОЦЕССОВ И СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ252
3.1. Основные понятия моделирования бизнес-процессов253

<u>3.2. Структурный (процессный) подход к моделированию</u>	
<u>бизнес-процессов</u>	257
<u>3.2.1. Принципы процессного подхода</u>	257
<u>3.2.2. Применение диаграмм потоков данных</u>	258
<u>3.2.3. Система моделирования ARIS</u>	261
<u>3.2.4. Метод Ericsson-Penker</u>	266
<u>3.2.5. Пример использования процессного подхода</u>	268
<u>3.3. Объектно-ориентированный подход к моделированию</u>	
<u>бизнес-процессов</u>	280
<u>3.3.1. Методика моделирования Rational Unified Process</u>	280
<u>3.3.2. Пример использования объектно-ориентированного</u>	
<u>подхода</u>	290
<u>3.4. Спецификация требований к программному</u>	
<u>обеспечению</u>	295
<u>3.4.1. Основы спецификации требований к программному</u>	
<u>обеспечению</u>	295
<u>3.4.2. Пример спецификации требований к программному</u>	
<u>обеспечению</u>	310
<u>Глава 4. АНАЛИЗ И ПРОЕКТИРОВАНИЕ</u>	
<u>ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ</u>	322
<u>4.1. Структурное проектирование ПО</u>	323
<u>4.2. Пример структурного проектирования программного</u>	
<u>обеспечения</u>	328
<u>4.3. Объектно-ориентированный анализ</u>	330
<u>4.3.1. Архитектурный анализ</u>	331
<u>4.3.2. Анализ вариантов использования</u>	334
<u>4.4. Объектно-ориентированное проектирование</u>	356
<u>4.4.1. Проектирование архитектуры системы</u>	356

<u>4.4.2. Проектирование элементов системы</u>	371
--	-----

Глава 5. ТЕХНОЛОГИИ СОЗДАНИЯ ПРОГРАММНОГО

<u>ОБЕСПЕЧЕНИЯ</u>	384
---------------------------------	-----

<u>5.1. Определение технологии</u>	384
--	-----

<u>5.2. Общие требования, предъявляемые к ТС ПО</u>	387
---	-----

<u>5.3. Внедрение ТС ПО в организации</u>	388
---	-----

<u>5.3.1. Общие сведения</u>	388
------------------------------------	-----

<u>5.3.2. Определение потребностей в ТС ПО</u>	
--	--

.....392

<u>5.3.3. Оценка и выбор ТС ПО</u>	401
--	-----

<u>5.3.4. Критерии оценки и выбора ТС ПО</u>	407
--	-----

<u>5.3.5. Выполнение пилотного проекта</u>	419
--	-----

<u>5.3.6. Практическое внедрение ТС ПО</u>	431
--	-----

<u>5.4. Примеры ТС ПО</u>	442
---------------------------------	-----

<u>5.4.1. Технология RUP (Rational Unified. Process)</u>	443
--	-----

<u>5.4.2. Технология Oracle</u>	456
---------------------------------------	-----

<u>5.4.3. Технология Borland</u>	463
--	-----

<u>5.4.4. Технология Computer Associates</u>	466
--	-----

Глава 6. ОЦЕНКА ТРУДОЕМКОСТИ СОЗДАНИЯ

<u>ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ</u>	470
--	-----

<u>6.1. Методы оценки и их классификация</u>	471
--	-----

<u>6.2. Методика оценки трудоемкости разработки ПО на основе функциональных точек</u>	479
---	-----

<u>6.2.1. Определение функциональных типов</u>	481
--	-----

<u>6.2.2. Определение количества и сложности функциональных типов по данным</u>	484
---	-----

<u>6.2.3. Определение количества и сложности транзакционных функциональных типов</u>	485
<u>6.2.4. Подсчет количества функциональных точек</u>	487
<u>6.2.5. Оценка трудоемкости разработки</u>	495
<u>6.3. Алгоритмическое моделирование трудоемкости разработки программного обеспечения</u>	499
<u>6.3.1. Теоретические (математические) модели</u>	499
<u>6.3.2. Статистические (регрессионные) модели</u>	501
<u>6.4. Методика оценки трудоемкости разработки ПО на основе вариантов использования (по материалам компании Rational Software)</u>	511
<u>6.4.1. Определение весовых показателей действующих лиц</u>	511
<u>6.4.2. Определение весовых показателей вариантов использования</u>	513
<u>6.4.3. Определение технической сложности проекта</u>	514
<u>6.4.4. Определение уровня квалификации разработчиков</u>	516
<u>6.4.5. Оценка трудоемкости проекта</u>	517
<u>6.5. Методы, основанные на экспертных оценках</u>	518
<u>6.5.1. Метод Дельфи</u>	519
<u>6.5.2. Метод декомпозиции работ</u>	520
<u>6.6. Средства оценки трудоемкости</u>	521
<u>6.7. Планирование итерационного процесса создания ПО</u>	523
<u>Глава 7. ОСОБЕННОСТИ СОВРЕМЕННЫХ ПРОЕКТОВ</u>	527
<u>7.1. Категории «безнадежных» проектов</u>	528
<u>7.2. Причины, порождающие «безнадежные» проекты</u>	529
<u>7.3. Причины разногласий между участниками проекта</u>	532
<u>7.4. Переговоры в «безнадежном» проекте</u>	534

<u>7.5. Человеческий фактор в «безнадежных» проектах</u>	540
<u>7.6. Процессы в «безнадежных» проектах</u>	553
<u>7.7. Динамика процессов</u>	556
<u>7.8. Контроль над продвижением проекта</u>	563
<u>7.9. Технология и инструментальные средства «безнадежных» проектов</u>	569
<u>Дополнительная литература</u>	579
<u>Краткий словарь терминов</u>	582
<u>Список основных сокращений</u>	590
<u>Software Desingn</u>	594

ПРЕДИСЛОВИЕ

Цель учебника — введение в современные методы и средства проектирования программного обеспечения информационных систем (ПО ИС), основанные на международных стандартах и использовании CASE-технологии, а также формирование навыков их самостоятельного практического применения. При отборе материала автор стремился к следующему:

- осветить с системных позиций основные направления, существующие в области инженерного проектирования ПО или программной инженерии, не углубляясь в их детали, с тем чтобы сформировать у читателя целостное представление о данной области (в противном случае учебник мог бы превратиться в многотомную энциклопедию);
- заполнить пробел, имеющийся в отечественной учебной литературе по программной инженерии;
- учесть официально утвержденные и признанные де-факто международные и отечественные стандарты в области программной инженерии и прежде всего стандарт ISO 12207 «Процессы жизненного цикла ПО», на котором базируются почти все современные промышленные технологии создания ПО;
- рассмотреть современное состояние развития CASE-средств и промышленных технологий проектирования ПО.

В новом издании на основе опыта, накопленного автором в учебном процессе, а также новых материалов, появившихся с момента выпуска первого издания учебников 2002 и 2003 г., существенно пересмотрено и дополнено описание методов объектно-ориентированного анализа и проектирования ПО, добавлены новые разделы и примеры, изменена структура.

Учебник подготовлен в соответствии с Государственным образовательным стандартом по специальности 351400 «Прикладная информатика», но может быть использован также студентами и преподавателями других специальностей, связанными с проектированием информационных систем и программного

обеспечения, в частности 351500 «Математическое обеспечение и администрирование информационных систем» и 010200 «Прикладная математика и информатика». Он состоит из введения и семи глав.

Во введении рассматриваются основные проблемы современных проектов, причины их возникновения и способы разрешения.

В главе 1 описываются процессы и модели жизненного цикла (ЖЦ) ПО, модель оценки зрелости процессов создания ПО СММ и уровни зрелости процессов создания ПО. В качестве примеров процессов рассмотрены управление требованиями и управление конфигурацией ПО.

Глава 2 посвящена методическим аспектам проектирования ПО. Рассматриваются общие принципы проектирования систем, структурный и объектно-ориентированный подходы к анализу и проектированию ПО, унифицированный язык моделирования UML.

В главе 3 даны описание моделирования бизнес-процессов и спецификации требований к ПО. Представлены различные подходы к моделированию бизнес-процессов — структурные методы (диаграммы потоков данных, метод ARIS) и объектно-ориентированный подход к моделированию бизнес-процессов с использованием языка UML, а в главе 4 — методы анализа и проектирования ПО на основе структурного и объектно-ориентированного подхода.

Основная часть материала третьей и четвертой глав построена на методической базе одной из наиболее развитых современных технологий Rational Unified Process, ее применение иллюстрируется на примере учебного проекта.

Глава 5 посвящена технологиям создания ПО. Приводится система понятий, описывающих технологию создания ПО, состав компонентов технологии, требования, предъявляемые к технологии, факторы выбора технологии и пример технологии Rational Unified Process.

В главе 6 рассматриваются различные методы и стандартные метрики, применяемые для оценки трудоемкости создания ПО.

В главе 7 обсуждаются особенности управления современными проектами создания ПО в условиях жестких ресурсных ограничений.

В конце книги даются дополнительная литература, краткий словарь терминов и список основных сокращений.

Подготовка второго издания учебника во многом стала возможной благодаря той положительной реакции, которую я получал от своих многочисленных слушателей — специалистов различных организаций России и ближнего зарубежья, а также студентов факультета вычислительной математики и кибернетики МГУ.

Автор выражает также глубокую благодарность рецензентам — профессору Георгию Николаевичу Калянову и доценту Алексею Алексеевичу Сорокину, взявшим на себя труд прочитать рукопись и сделавшим ряд конструктивных замечаний. Я благодарен своей семье — жене Марине и дочери Александре за поддержку и терпение, которое они проявили в период написания книги.

Прочитав введение, вы узнаете:

- *Что представляет собой системный подход к проектированию программного обеспечения.*
- *В чем заключаются основные особенности и проблемы проектов современных систем программного обеспечения.*
- *Каковы современные тенденции в программной инженерии.*

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ. СИСТЕМНЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ПО

Методологическую основу проектирования ПО составляет системный подход. Под словом «система» понимается совокупность взаимодействующих компонентов и взаимосвязей между ними. Весь мир можно рассматривать как сложную взаимосвязанную совокупность естественных и искусственных систем. Это могут быть достаточно сложные системы (например, планеты в составе Солнечной системы), системы средней сложности (космический корабль) или сверхсложные системы (системы молекулярных взаимодействий в живых организмах). Искусственные системы, к которым относится ПО, по своей сложности, как правило, занимают среднее положение. Например, всемирная телефонная сеть содержит десятки или даже сотни тысяч переключателей, однако количество взаимодействий этих переключателей не идет ни в какое сравнение с количеством взаимодействий молекул даже в небольшом стакане воды. С точки зрения общей теории систем такие системы обычно рассматриваются как системы средней сложности.

Системный подход — это методология исследования объектов любой природы как систем, которая ориентирована на:

- раскрытие целостности объекта и обеспечивающих его механизмов;

- выявление многообразных типов связей объекта;
- сведение этих связей в единую картину.

Системный подход реализует представление сложного объекта в виде иерархической системы взаимосвязанных *моделей*, позволяющих фиксировать целостные свойства объекта, его структуру и динамику.

ПО как система, в свою очередь, является подсистемой некоторой информационной системы (ИС). По определению стандарта специальности 351400 «Прикладная информатика», **информационная система** — это совокупность:

- функциональных и информационных процессов конкретной предметной области;
- средств и методов сбора, хранения, анализа, обработки и передачи информации, зависящих от специфики области применения;
- методов управления процессами решения функциональных задач, а также информационными, материальными и денежными потоками в предметной области.

С другой стороны, ориентируясь на различные международные стандарты, ИС можно определить как совокупность следующих составных частей:

- система баз данных (база данных (БД) вместе с системой управления базами данных (СУБД));
- прикладное программное обеспечение;
- персонал;
- организационно-методическое (нормативное) обеспечение;
- технические средства.

Такая ИС функционирует: на конкретном уровне мирового хозяйства, в муниципальных, государственных, негосударственных и международных организациях различного назначения; в органах управления, министерствах, ведомствах и подчиненных им организациях; в экономических, банковских, налоговых учреждениях; на предприятиях различной организационно-правовой формы; в различных отраслях хозяйства страны или региона.

Программное обеспечение, в свою очередь, определяется как набор компьютерных программ, процедур и, возможно, связанной с ними документации и данных.

По определению Института управления проектами (Project Management Institute, **PMI**), **проект** — это временное предприятие, осуществляемое с целью создания уникального продукта или услуги. В любой инженерной дисциплине под **проектированием** обычно понимается некий унифицированный подход, с помощью которого мы ищем пути решения определенной проблемы, обеспечивая выполнение поставленной задачи. В контексте инженерного проектирования можно определить **цель проектирования** как создание системы, которая:

- удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;
- согласована с ограничениями, накладываемыми оборудованием;
- удовлетворяет явным и неявным требованиям по эксплуатационным качествам и потреблению ресурсов;
- удовлетворяет явным и неявным критериям дизайна продукта;
- удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств.

В другой формулировке **цель проектирования** — выявление ясной и относительно простой внутренней структуры, называемой **архитектурой системы**. Проект есть окончательный продукт процесса проектирования. Проектирование подразумевает учет противоречивых требований. Его продуктами являются модели, позволяющие понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

Таким образом, под **проектом ПО** будем понимать совокупность спецификаций ПО (включающих модели и проектную документацию), обеспечивающих создание ПО в конкретной программно-технической среде.

Проектирование ПО представляет собой процесс создания спецификаций ПО на основе исходных требований к нему. Проектирование ПО сводится к по-

следовательному уточнению его спецификаций на различных стадиях процесса создания ПО.

ОСНОВНЫЕ ОСОБЕННОСТИ ПРОЕКТОВ СОВРЕМЕННЫХ СИСТЕМ ПО

Индустрия ПО начала зарождаться в середине 50-х годов XIX в., однако почти до конца 60-х ей не уделялось серьезного внимания, поскольку ее доля в компьютерном бизнесе была слишком мала. В 1970 г. годового оборот всех фирм-разработчиков ПО в США не превышал 1/2 млрд долл. - около 3,7% всего оборота компьютерного бизнеса. Серьезный рост начался в 70-х годах XX в., начиная с принятого фирмой IBM в 1969 г. решения о развязывании цен (раздельном назначении цен на аппаратуру, ПО и услуги), и продолжился до конца декады и появления персонального компьютера. К 1979 г. годового объем продаж фирм-разработчиков ПО в США составлял около \$2 млрд. В 80-х годах рост составлял 20% в год и более, таким образом, годовые доходы фирм выросли до \$10 млрд. к 1982 г. и \$25 млрд. к 1985 г. Сегодня общий объем продаж ПО превышает \$100 млрд. Производство ПО сегодня — крупнейшая отрасль мировой экономики, в которой занято около трех миллионов специалистов, называющих себя программистами, разработчиками ПО и т.п. Еще несколько миллионов человек занимают рабочие места, напрямую зависящие от благополучия корпоративных информационных подразделений либо от производителей ПО, таких, как корпорации Microsoft или IBM.

Накопленный к настоящему времени опыт создания систем ПО показывает, что это логически сложная, трудоемкая и длительная работа, требующая высокой квалификации участвующих в ней специалистов. Однако до настоящего времени создание таких систем нередко выполняется на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования ПО. Кроме того, в процессе создания и

функционирования ПО потребности пользователей постоянно изменяются или уточняются, что еще более усложняет разработку и сопровождение таких систем.

Чтобы понять принципиальные отличия сложного ПО от обычной программы, рассмотрим рис. В.1¹.

В левом верхнем углу рисунка находится *программа*. Она является завершенным продуктом, пригодным для запуска только своим автором и только в той системе, где она была разработана.

При перемещении вниз через горизонтальную границу программа превращается в *программный продукт*. Это программа, которую любой человек может использовать и, возможно, при наличии программистской квалификации, сопровождать, т.е. вносить в нее различные изменения. Она может использоваться в различных операционных системах и с различными данными.

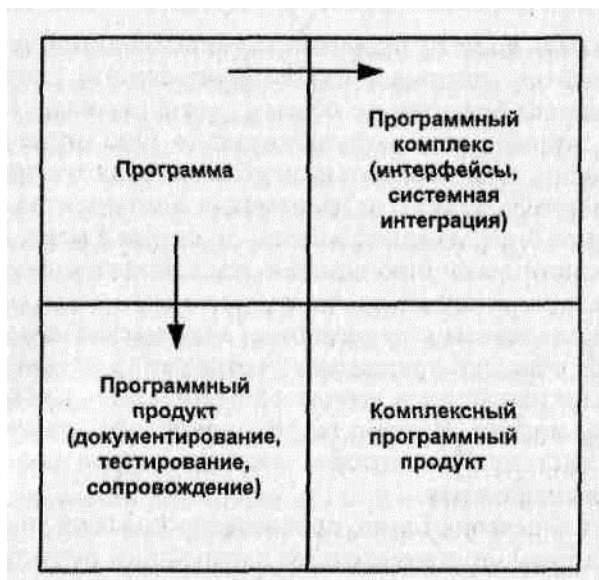


Рис. В.1. Отличие комплексного программного продукта от программы

Такую программу нужно тщательно протестировать, чтобы быть уверенным в ее надежности. Для этого нужно подготовить достаточное количество конт-

¹ Брукс Ф. Мифический человеко-месяц, или как создаются программные системы: Пер. с англ. — СПб.: Символ-Плюс, 1999.

рольных примеров для проверки диапазона допустимых значений входных данных, обработать эти примеры и зафиксировать результаты. Наконец, развитие программы в программный Продукт требует создания подробной документации, с помощью которой каждый мог бы использовать ее. Опыт говорит, что программный продукт стоит, по крайней мере, втрое дороже, чем просто отлаженная программа с такой же функциональностью.

При пересечении вертикальной границы программа становится компонентом *программного комплекса*. Он представляет собой набор взаимодействующих программ, согласованных по функциям и форматам данных, предназначенный для решения крупномасштабных задач. Чтобы стать частью программного комплекса, входные и выходные данные и сообщения программы должны удовлетворять точно определенным интерфейсам. Программа должна быть спроектирована таким образом, чтобы использовать точно определенные ресурсы - объем памяти, внешние устройства, процессорное время. Наконец, программу нужно протестировать вместе с прочими системными компонентами во всех сочетаниях, которые могут встретиться. Это тестирование может оказаться большим по объему, поскольку количество тестируемых случаев растет экспоненциально. Оно также занимает много времени, так как скрытые ошибки выявляются при неожиданных взаимодействиях отлаживаемых компонентов. Компонент программного комплекса стоит, по крайней мере, втрое дороже, чем автономная программа с теми же функциями. Стоимость может увеличиться, если в системе много компонентов.

В правом нижнем углу находится *комплексный программный продукт*. От обычной программы он отличается во всех перечисленных выше отношениях и стоит соответственно, как минимум, в девять раз дороже. Именно такой продукт представляет собой сложную систему ПО, которая является целью большинства программных проектов.

В 1975 г. Фредерик Брукс, проанализировав свой уникальный по тем временам опыт руководства крупнейшим проектом разработки операционной си-

системы OS/360, определил перечень неотъемлемых свойств ПО: сложность, согласованность, изменяемость и незримость.

Сложность. Благодаря уникальности и несхожести своих составных частей программные системы принципиально отличаются от Технических систем (например, компьютеров), в которых преобладают повторяющиеся элементы.

Сами компьютеры сложнее, чем большинство продуктов человеческой деятельности. Количество их возможных состояний очень велико, поэтому их так трудно понимать, описывать и тестировать. У программных систем количество возможных состояний на порядки величин превышает количество состояний компьютеров.

Аналогично, масштабирование ПО — это не просто увеличение в размере тех же самых элементов, это обязательно увеличение числа различных элементов. В большинстве случаев эти элементы взаимодействуют между собой нелинейным образом, и сложность целого также возрастает нелинейно.

Сложность ПО является существенным, а не второстепенным свойством. Поэтому попытки описания программных объектов, абстрагируясь от их сложности, приводят к абстрагированию и от их сущности. Математика и физика за три столетия достигли больших успехов, создавая упрощенные модели сложных физических явлений, получая из этих моделей свойства и проверяя их опытным путем. Это удавалось благодаря тому, что сложность, игнорировавшаяся в моделях, не была существенным свойством явлений. Такой подход не работает, когда сложность является сущностью.

Многие трудности разработки ПО следуют из этой сложности и ее нелинейного роста при увеличении размера. Сложность является причиной трудностей, возникающих в процессе общения между разработчиками, что ведет к ошибкам в продукте, превышению стоимости разработки, затягиванию выполнения графиков работ. Сложность служит причиной трудности понимания всех возможных состояний программ, что ведет к снижению их надежности. Сложность структуры затрудняет развитие ПО и добавление новых функций.

Согласованность. Во многих случаях новое ПО должно согласовываться с уже существующим, таким образом, значительная часть сложности происходит от необходимости согласования с различными интерфейсами, и эту проблему невозможно упростить только с помощью переделки ПО.

Изменяемость. ПО постоянно подвержено изменениям. Конечно, это относится и к зданиям, автомобилям, компьютерам. Но произведенные вещи редко подвергаются изменениям после изготовления. Их заменяют новые модели, или существенные изменения включают в более поздние серийные экземпляры того же базового проекта. Изменения здесь случаются значительно реже, чем модификация работающего ПО.

Отчасти это происходит потому, что ПО гораздо легче изменить. Здания тоже перестраиваются, но признаваемая всеми высокая стоимость изменений умеряет запросы энтузиастов.

Все удачные программные продукты подвергаются изменениям. При этом действуют два процесса. Во-первых, как только обнаруживается польза программного продукта, начинаются попытки применения его на грани или за пределами первоначальной области. Требование расширения функций исходит, в основном, от пользователей, которые удовлетворены основным назначением и изобретают для него новые применения.

Во-вторых, удачный программный продукт живет дольше обычного срока существования компьютера, для которого он первоначально был создан. Приходят новые компьютеры, и программа должна быть согласована с их возможностями.

Короче говоря, программный продукт является частью среды приложений, пользователей, законов и компьютеров. Все они непрерывно меняются, и их изменения неизбежно требуют изменения программного продукта.

Незримость. Программный продукт невидим. Для материальных объектов геометрические абстракции являются мощным инструментом. План здания помогает архитектору и заказчику оценить пространство, возможности перемещения, виды. Становятся очевидными противоречия, можно заметить упущения

ния. Масштабные чертежи механических деталей и объемные модели молекул, будучи абстракциями, служат той же цели. Геометрическая реальность отражается в геометрической абстракции.

Реальность ПО не встраивается естественным образом в пространство. Поэтому у него нет готового геометрического представления подобно тому, как местность представляется картой, кремниевые микросхемы — диаграммами, компьютеры — схемами соединений. При попытке графически представить структуру программы обнаруживается, что требуется не один, а несколько неориентированных графов, наложенных один на другой.

Незримость ПО не только затрудняет индивидуальный процесс проектирования, но и серьезно затрудняет общение между разработчиками.

Современные крупномасштабные проекты программных систем характеризуются, как правило, особенностями, представленными на рис. В.2.

Характеристики объекта внедрения:

- структурная сложность (многоуровневая иерархическая структура организации) и территориальная распределенность;
- функциональная сложность (многоуровневая иерархия и большое количество функций, выполняемых организацией; сложные взаимосвязи между ними);
- информационная сложность (большое количество источников и потребителей информации (министерства и ведомства, местные органы власти, организации-партнеры), разнообразные формы и форматы представления информации, сложная информационная модель объекта — большое количество информационных сущностей и сложные взаимосвязи между ними), сложная технология прохождения документов;
- сложная динамика поведения, обусловленная высокой изменчивостью внешней среды (изменения в законодательных и нормативных актах, нестабильность экономики и политики) и внутренней среды (структурные реорганизации, текучесть кадров).



Рис. В.2. Особенности современных крупномасштабных проектов программных систем

Технические характеристики проектов создания ПО:

- различная степень унифицированности проектных решений в рамках одного проекта;
- высокая техническая сложность, определяемая наличием совокупности тесно взаимодействующих компонентов (подсистем), имеющих свои локальные задачи и цели функционирования (транзакционных приложений, предъявляющих повышенные требования к надежности, безопасности и производительности, и приложений аналитической обработки (систем поддержки принятия решений), использующих нерегламентированные запросы к данным большого объема);

- отсутствие полных аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем, высокая доля вновь разрабатываемого ПО;
- большое количество и высокая стоимость унаследованных приложений (существующего прикладного ПО), функционирующих в различной среде (персональные компьютеры, миникомпьютеры, мэйнфреймы), необходимость интеграции унаследованных и вновь разрабатываемых приложений;
- большое количество локальных объектов внедрения, территориально распределенная и неоднородная среда функционирования (СУБД, операционные системы, аппаратные платформы);
- большое количество внешних взаимодействующих систем организаций с различными форматами обмена информацией (налоговая служба, налоговая полиция, Госстандарт, Госкомстат, Министерство финансов, МВД, местная администрация).

Организационные характеристики проектов создания ПО:

- различные формы организации и управления проектом: централизованно управляемая разработка тиражируемого ПО, экспериментальные пилотные проекты, инициативные разработки, проекты с участием как собственных разработчиков, так и сторонних компаний на контрактной основе;
- большое количество участников проекта как со стороны заказчиков (с разнородными требованиями), так и со стороны разработчиков (более 100 человек), разобщенность и разнородность отдельных групп разработчиков по уровню квалификации, сложившимся традициям и опыту использования тех или иных инструментальных средств;
- значительная длительность жизненного цикла системы, в том числе значительная временная протяженность проекта, обусловленная масштабами организации-заказчика, различной степенью готовности отдельных ее

подразделений к внедрению ПО и нестабильностью финансирования проекта;

- высокие требования со стороны заказчика к уровню технологической зрелости организаций-разработчиков (наличие сертификации в соответствии с международными и отечественными стандартами).

ОСНОВНЫЕ ПРОБЛЕМЫ СОВРЕМЕННЫХ ПРОЕКТОВ ПО

В конце 60-х годов прошлого века в США было отмечено явление под названием «software crisis» (кризис ПО). Это выражалось в том, что большие проекты стали выполняться с отставанием от графика или с превышением сметы расходов, разработанный продукт не обладал требуемыми функциональными возможностями, производительность его была низка, качество получаемого программного обеспечения не устраивало потребителей.

Аналитические исследования и обзоры, выполняемые в последние годы ведущими зарубежными аналитиками, показывали не слишком обнадеживающие результаты. Например, результаты исследований, выполненных в 1995 г. компанией Standish Group, которая проанализировала работу 364 американских корпораций и итоги выполнения более 23 тыс. проектов, связанных с разработкой ПО, выглядели следующим образом:

- только 16,2% завершились в срок, не превысили запланированный бюджет и реализовали все требуемые функции и возможности;
- 52,7% проектов завершились с опозданием, расходы превысили запланированный бюджет, требуемые функции не были реализованы в полном объеме;
- 31,1% проектов были аннулированы до завершения;
- для двух последних категорий проектов бюджет среднего проекта оказался превышенным на 89%, а срок выполнения - на 122%.

В 1998 г. процентное соотношение трех перечисленных категорий проектов лишь немного изменилось в лучшую сторону (26%, 46% и 28% соответственно).

В последние годы процентное соотношение трех перечисленных категорий проектов также незначительно меняется в лучшую сторону, однако, по оценкам ведущих аналитиков, это происходит в основном за счет снижения масштаба выполняемых проектов, а не за счет повышения управляемости и качества проектирования.

В числе причин возможных неудач, по мнению разработчиков, фигурируют:

- нечеткая и неполная формулировка требований к ПО;
- недостаточное вовлечение пользователей в работу над проектом;
- отсутствие необходимых ресурсов;
- неудовлетворительное планирование и отсутствие грамотного управления проектом;
- частое изменение требований и спецификаций;
- новизна и несовершенство используемой технологии;
- недостаточная поддержка со стороны высшего руководства;
- недостаточно высокая квалификация разработчиков, отсутствие необходимого опыта.

В последнее время ведущие зарубежные аналитики отмечают как одну из причин многих неудач тот факт, что множество проектов выполняется в экстремальных условиях. В англоязычной литературе с легкой руки Эдварда Йордона¹, одного из ведущих мировых специалистов в области ПО, утвердилось название «death march», буквально - «смертельный марш». Под ним понимается такой проект, параметры которого отклоняются от нормальных значений, по крайней мере, на 50%. По отношению к проектам создания ПО это означает наличие одного или более из следующих ограничений:

¹ Йордон Эд. Путь камикадзе.: Пер; с англ. - М.: ЛОРИ, 2001

- план проекта сжат более чем наполовину по сравнению с нормальным расчетным планом; таким образом, проект, требующий в нормальных условиях 12 календарных месяцев, приходится выполнять за 6 или менее месяцев. Жесткая конкуренция на мировом рынке делает такую ситуацию наиболее распространенной;
- количество разработчиков уменьшено более чем наполовину по сравнению с действительно необходимым для проекта данного размера и масштаба. На сегодняшний день наиболее общей причиной уменьшения количества разработчиков является сокращение штатов компании в результате кризиса, реорганизации и т.д.;
- бюджет и связанные с ним ресурсы урезаны наполовину. Зачастую это результат сокращения компании и других противозатратных мер, хотя это может быть и результатом конкурентной борьбы за выгодный контракт. Такое ограничение часто непосредственно влияет на количество нанимаемых разработчиков, однако последствия могут быть и менее явными — например, может быть принято решение привлечь относительно недорогих и неопытных молодых разработчиков вместо опытных дорогостоящих специалистов;
- требования к функциям, возможностям, производительности и другим техническим характеристикам вдвое превышают значения, которые они могли бы иметь в нормальных условиях.

Однако основная причина всех проблем заключается в ответе на вопрос: является ли проектирование ПО ремеслом, инженерной дисциплиной или чем-то средним? Многие разработчики ПО, вероятно, заявят, что программирование хотя и не сложилось в установившуюся инженерную дисциплину, но уже близко подошло к этому.

Однако можно утверждать, что разработка ПО на сегодняшний день является почти чистым ремеслом. Все знание о способах разработки ПО основано исключительно на пробах и ошибках. Конечно, со времени зарождения программирования сообщество программистов добилось некоторых успехов. При-

знанные корифеи программирования изобрели в помощь разработчикам ПО успешные методы и правила. Но, подобно средневековым архитекторам, разработчики ПО изучили и испытали эти методы путем проб и ошибок. Современным разработчикам не хватает системы основных принципов, на основе которых можно было бы строить свои правила и методы. Замечено, что ключевым фактором успеха проекта является хорошая архитектура.

Именно неспособность регулярно создавать хорошую архитектуру не дает права разработке ПО называться сложившейся инженерной дисциплиной. Для доказательства адекватности проекта до завершения любых строительных работ инженер, в отличие от программиста, использует систему основных принципов. Разработчик ПО, с другой стороны, при оценке качества архитектуры должен полагаться на тестирование. Он должен искать хорошую архитектуру путем проб и ошибок.

Это объясняет, почему двумя наиболее явными проблемами неудачных программных проектов являются переделка программ и обнаружение негодности проекта на его поздних стадиях. Разработчик проектирует архитектуру на ранних стадиях разработки ПО, но не имеет возможности сразу же оценить ее качество. У него отсутствуют под рукой основные принципы для доказательства адекватности проекта. Тестирование программного обеспечения постепенно выявляет все дефекты архитектуры, но только на поздних стадиях разработки, когда исправление ошибок становится дорогим и разрушительным для проекта.

Почему же идеология проб и ошибок так глубоко проникла в разработку ПО? Для ответа на этот вопрос необходимо понять, что разработка ПО изначально является *проектированием* и не имеет признаков *строительства* или *производства*. Это утверждение трудно принять, но оно может быть легко обосновано. Все хорошо понимают, что такое проектирование, где оно заканчивается и где начинается строительство или производство. Рассмотрим два следующих аргумента:

1. Граница между проектированием и строительством всегда четко обозначена чертежом. Проектирование включает в себя все операции, необходимые для создания чертежа, а строительство охватывает все операции, необходимые для создания продуктов по этому чертежу. В идеальном случае чертеж должен определять создаваемый продукт во всех подробностях, что, конечно же, бывает очень редко. Тем не менее, целью чертежа является настолько подробное описание конструируемого продукта, насколько это возможно. Описывает ли проект архитектуры программной системы создаваемый продукт «во всех подробностях»? — Нет. Проект архитектуры предназначен для описания существенных, но, безусловно, не всех подробностей программной системы. Поэтому очевидно, что проект архитектуры не является чертежом. Все подробности программной системы описываются только кодом на языке высокого уровня, который, таким образом, является чертежом программы. А поскольку все операции, ведущие к созданию чертежа, являются проектированием, то и вся разработка ПО должна считаться проектированием.

2. Объем работ (время, деньги, ресурсы), необходимый для создания продукта, всегда может быть разделен на проектировочную и производственную составляющие. В чем разница? — Объем работ проектирования является общим для всех копий продукта и должен быть затрачен только один раз. Объем работ для производства должен затрачиваться при создании каждой копии продукта. Программный продукт обычно представляет собой двоичный исполняемый файл программы, поставляемой на компакт-диске. Ясно, что усилия по созданию исходного кода программы, включая проект архитектуры, подробный проект и код на языке высокого уровня, должны быть затрачены лишь однажды, независимо от количества выпущенных копий программного обеспечения. Следовательно, усилия по созданию исходного кода программы являются целиком проектировочными, а вся разработка ПО является проектированием.

Разработчики не строят ПО - они его проектируют. Конечный результат проектирования — код на языке высокого уровня — является чертежом ПО. Компилятор и компоновщик механически строят программный продукт — дво-

ичный исполняемый файл— по этому спроектированному коду. Проект архитектуры программной системы наиболее близко соответствует картонным моделям или эскизам проекта, используемым в некоторых инженерных дисциплинах. Чтобы понять, почему разработчики ПО до сих пор не увидели и не нашли основных принципов, представим себе мир, в котором создание небоскреба не требует ничего, кроме подробного чертежа. Имея чертеж, архитектор мог бы одним нажатием кнопки построить небоскреб — мгновенно и практически без затрат. Затем архитектор мог бы проверить небоскреб и сравнить его с техническими требованиями. Если бы он разрушился или не смог пройти проверку, архитектор мог бы его снести и убрать обломки— мгновенно и опять же без затрат. Стал бы этот архитектор тратить много времени на формальную проверку согласованности проекта с физическими законами? Или хотя бы пытаться исследовать и понять эти законы? — Вряд ли. Он, вероятно, смог бы получить результаты быстрее путем многократного строительства, проверки и сноса небоскреба, каждый раз внося исправления в чертеж. В мире, где строительство и разрушение бесплатны, выбирается метод проб и ошибок, а фундаментальные исследования остаются на будущее.

ПО разрабатывается именно в таком мире. Программист создает чертеж в виде программы на языке высокого уровня. Затем он позволяет компилятору и компоновщику в мгновение ока и почти без затрат построить программный продукт. Создание чертежа требует значительных усилий, но строительство с помощью компилятора и компоновщика практически бесплатно. Программисту вообще не надо беспокоиться о сносе и уборке обломков, по крайней мере до тех пор, пока ему достаточно дискового пространства. Неудивительно, что идеология проб и ошибок так глубоко укоренилась в процессе разработки ПО, а сообщество программистов не удосужилось исследовать основные принципы разработки ПО.

Метод проб и ошибок завел достаточно далеко. Но рост сложности современных программных систем подводит нас к жесткому пределу. За пределами

определенного уровня сложности создание качественных архитектур методом проб и ошибок становится невозможным.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Объективная потребность контролировать процесс разработки сложных систем ПО, прогнозировать и гарантировать стоимость разработки, сроки и качество результатов привела в конце 60-х годов прошлого века к необходимости перехода от кустарных к индустриальным способам создания ПО и появлению совокупности инженерных методов и средств создания ПО, объединенных общим названием *«программная инженерия» (software engineering)*. Впервые термин software engineering был использован как тема исторической конференции Научного комитета НАТО в 1968 г. Спустя семь лет, в 1975 г., в Вашингтоне была проведена первая международная конференция, посвященная программной инженерии, тогда же появилось первое издание, посвященное программной инженерии, — IEEE Transactions on Software Engineering. **Программная инженерия** определяется, с одной стороны, как совокупность инженерных методов и средств создания ПО а, с другой стороны, как дисциплина, изучающая применение строгого систематического количественного (т.е инженерного) подхода к разработке, эксплуатации и сопровождению ПО.

За прошедшее с тех пор время так и не найдено общепризнанного русского эквивалента этому английскому термину: буквальный перевод «программная инженерия» звучит не совсем по-русски; специализированные англо-русские словари предлагают такие варианты, как «разработка ПО», «программотехника», «технология программирования» и даже просто «программирование». При этом предмет в лучшем случае сужается, а в худшем просто искажается. В ре-

зультате слишком многие трактуют понятие «software engineering» только как приложение технологий создания ПО.

Терминологические трудности отражают проблемы существенные: общеизвестны сложные отношения отечественных программистов с технологическими схемами, заключающими процесс создания ПО в жесткие рамки. Поэтому при наличии устойчивого мифа «наши программисты — лучшие в мире» в России так мало примеров коммерчески успешных на мировом рынке программных продуктов — не просто интересно задуманных и воплощенных в виде прототипа, но и прошедших весь жизненный цикл и попавших в руки широкому потребителю.

Официальной причиной рождения программной инженерии была реакция на те проблемы программной индустрии, которые на той же конференции НАТО получили название «кризиса ПО». Об этом кризисе говорят и сегодня; однако ныне программная инженерия — это обширная и хорошо разработанная область компьютерной науки и технологии, включающая в себя многообразные математические, инженерные, экономические и управленческие аспекты.

В основе программной инженерии лежит одна фундаментальная идея: *проектирование ПО является формальным процессом, который можно изучать и совершенствовать*. Освоение и правильное применение методов и средств создания ПО позволяет повысить его качество, обеспечить управляемость процесса проектирования ПО и увеличить срок его жизни.

В то же время попытки чрезмерной формализации процесса, а также прямого заимствования идей и методов из других областей инженерной деятельности (строительства, производства) привели к ряду серьезных проблем. После двух десятилетий напрасных ожиданий повышения продуктивности процессов создания ПО, возлагаемых на новые методы и технологии, специалисты в индустрии ПО пришли к пониманию, что фундаментальная проблема в этой области — неспособность эффективного управления проектами создания ПО. Невозможно достичь удовлетворительных результатов от применения даже самых совершенных технологий и инструментальных средств, если они применяются

бессистемно, разработчики не обладают необходимой квалификацией для работы с ними, и сам проект выполняется и управляется хаотически, в режиме «тушения пожара». Бессистемное применение технологий создания ПО, в свою очередь, порождает разочарование в используемых методах и средствах (анализ мнений разработчиков показывает, что среди факторов, влияющих на эффективность создания ПО, используемым методам и средствам придается гораздо меньшее значение, чем квалификации и опыту разработчиков). Если для управления проектами и внедрения технологий не создается необходимой инфраструктуры и не обеспечивается техническая поддержка, то на выполнение проектов затрачивается существенно больше времени и ресурсов по отношению к планируемым (да и само планирование ресурсов выполняется «потолочным» методом»). Если в таких условиях отдельные проекты завершаются успешно, то этот успех достигается за счет героических усилий фанатично настроенного коллектива разработчиков. Успех, который держится исключительно на способностях отдельных организаторов «вытаскивать» проекты из прорывов, не дает гарантии устойчивой производительности и качества при создании ПО. Постоянное повышение качества создаваемого ПО и снижение его стоимости может быть обеспечено только при условии достижения организацией необходимой технологической зрелости, создании эффективной инфраструктуры как в сфере разработки ПО, так и в управлении проектами. В соответствии с моделью CMM (Capability Maturity Model) Института программной инженерии (Software Engineering Institute, SEI), в хорошо подготовленной (зрелой) организации персонал обладает технологией и инструментарием оценки качества процессов создания ПО на протяжении всего жизненного цикла ПО и на уровне всей организации. Процессы выстраиваются таким образом, чтобы обеспечить реальные сроки создания ПО.

При решении проблемы повышения эффективности создания ПО основное внимание уделяется, как правило, процессу разработки ПО, что вызвано естественным желанием разработчиков и заказчиков экономить средства с самого начала проекта (особенно в условиях ограниченных финансовых возможностей

и высокой конкуренции). В то же время большинство крупномасштабных проектов создания ПО характеризуется длительным жизненным циклом (10 — 15 лет), в котором на стадию создания (разработки) приходится только первые 3—4 года, а остальное время эксплуатации созданной системы (стадия сопровождения и развития) распределяется, по оценкам Института программной инженерии (Software Engineering Institute, SEI), примерно поровну на следующие стадии:

- начальную стадию сопровождения, связанную с устранением ошибок и доработками, возникшими из-за неучтенных или вновь возникших требований пользователей;
- стадию «зрелого» сопровождения, характеризующуюся относительно полным удовлетворением основных потребностей пользователей, но в то же время связанную с накоплением ряда проблем, а именно:

в процессе развития и изменения ПО его первичная структура искажается и усложняется, что приводит к возрастанию сложности модификации ПО. В результате возрастает энтропия (меры неопределенности) ПО, если не предпринимаются специальные усилия для ее уменьшения. Рост энтропии и сложности ПО ведет к повышению сложности тестирования ПО, ухудшению его сопровождаемости. По мере развития ПО становится все более хаотичным по структуре и процессу функционирования и все более отличается от упорядоченного, первоначально задуманного проекта, характеризующегося наименьшей энтропией. При низком качестве ПО трудно отслеживать его версии и модификации, а накопление «критической массы» недокументированных изменений может привести к лавинообразному росту затрат на его сопровождение. Это происходит до момента возникновения нерентабельности дальнейшего сопровождения ПО и целесообразности его замены на новое ПО;

новые требования к системе выходят за рамки ограничений, заложенных при ее создании. Например, увеличение числа поддерживаемых в организации платформ или прекращение поддержки аппаратной платформы со стороны поставщика может привести к необходимости переноса ПО в новую среду.

текучесть кадров приводит к снижению количества специалистов, способных сопровождать ПО (разобраться в чужом недокументированном коде практически невозможно). Зависимость ПО от ведущих разработчиков может даже привести к невозможности дальнейшего сопровождения и развития ПО в случае прекращения их работы над продуктом.

- стадию эволюции/замены, характеризующуюся достижением системой порога своей сопровождаемости в существующем виде и невозможностью удовлетворить новые требования без внесения принципиальных изменений. Постоянное изменение и увеличение функций ПО приводит к тому, что становится экономически выгодным прекратить сопровождение и разработать полностью новое ПО или подвергнуть систему реинжинирингу, заключающемуся в ее существенной переработке и/или переносе в новую среду.

Под *сопровождением* в общем случае понимается внесение изменений в эксплуатируемый программный продукт в целях исправления обнаруженных ошибок (*корректирующее сопровождение*), повышения производительности и улучшения эксплуатационных характеристик системы (*совершенствующее сопровождение*), а также адаптации к изменившейся или изменяющейся среде (*адаптирующее сопровождение*). При этом более 50% общего объема работ по сопровождению приходится на совершенствующее сопровождение. В общих затратах различных организаций и предприятий на ПО доля затрат на сопровождение составляет от 60% до 80% (эта доля является максимальной для крупных государственных учреждений и частных компаний), и величина этих затрат продолжает расти. Негативными последствиями такого роста являются: 1) неудовлетворенность пользователей из-за большого времени, затрачиваемого на внесение изменений в ПО; 2) снижение качества ПО и 3) сокращение объема новых разработок в будущем, поскольку все большее количество разработчиков вынуждено переключаться на сопровождение. Так, по оценкам компании Hewlett-Packard, в 1994 г. от 60% до 80% ее разработчиков были заняты сопровождением ПО, а в отчетах федеральных служб США ранее отмечалось, что на

сопровождение уходит от 50% до 80% рабочего времени программистов. Перечисленные тенденции отражены на рис. В.3.

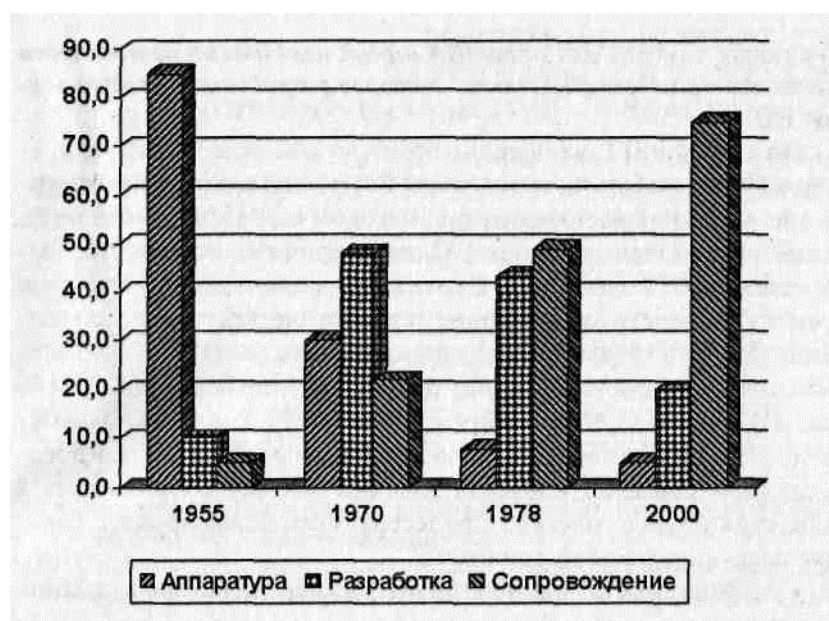


Рис. В.3. Тенденции изменения соотношения стоимости аппаратуры и ПО

Первоначально, в 1960 — 1970 гг., в понятие сопровождения ПО входило только исправление ошибок и устранение мелких замечаний пользователей. Технология сопровождения представлялась достаточно простой и сравнительно слабо влияла на методы и средства разработки ПО. При сопровождении подвергались изменениям одна — две версии ПО, и задача состояла в обеспечении и улучшении качества одной основной версии. В настоящее время сопровождение превратилось в весьма трудоемкий процесс модификации и развития множества версий крупномасштабного ПО и его компонентов, значительно различающихся функциями и качеством.

По мере развития новых технологий стало ясно, что суммарные затраты на сопровождение и создание новых версий могут значительно превосходить затраты на разработку первой версии ПО. Действительно, если разработка первой версии сложного ПО продолжается 3 года, а последующая его эксплуатация и создание версий происходит в течение 10 лет, то можно ожидать, что суммарные затраты на этих интервалах времени соизмеримы. Даже при предположе-

нии, что сопровождением и разработкой новых версий будет занята половина специалистов, осуществивших создание первых версий, суммарные затраты на эти работы превысят первичные. Опыт последних лет показал, что во многих случаях *для развития версий необходимо практически такое же число специалистов, которое разработало первую версию ПО.*

По данным SEI, в последние годы до 80% всего эксплуатируемого ПО разрабатывалось вообще без использования какой-либо дисциплины проектирования, методом «code and fix» (кодирования и исправления ошибок). Одна из причин — упомянутое выше стремление сэкономить на стадии разработки, не затрачивая времени и средств на внедрение технологического процесса создания ПО. Эти затраты до недавнего времени были довольно значительными и составляли, по различным оценкам, более \$100 тыс. и около трех лет на внедрение развитой технологии, охватывающей большинство процессов жизненного цикла ПО, в многочисленной команде разработчиков (до 100 чел.). Причина — в «тяжести» технологических процессов. «Тяжелый» процесс обладает следующими особенностями:

- необходимость документировать каждое действие разработчиков;
- множество рабочих продуктов (в первую очередь — документов), создаваемых в бюрократической атмосфере;
- отсутствие гибкости;
- детерминированность (долгосрочное детальное планирование и предсказуемость всех видов деятельности, а также распределение человеческих ресурсов на длительный срок, охватывающий большую часть проекта).

Для того чтобы проиллюстрировать насколько «тяжелыми» могут быть формальные процессы, эксперт в области использования метрик Каперс Джонс подсчитал, что процесс разработки ПО по стандарту DOD-2167A Министерства обороны США требует 400 слов в документации на английском языке для каждой строки исходного кода. Так, если создается среднее приложение размером 50 000 строк исходного кода, потребуется наличие армии технических специа-

листов для создания 20 миллионов слов документации с описанием того, что делает код, как он функционирует и почему это происходит именно так.

Альтернативой «тяжелому» процессу является адаптивный (гибкий) процесс, основанный на принципах «быстрой разработки ПО», интенсивно развиваемых в последнее десятилетие.

СОВРЕМЕННЫЕ ТЕНДЕНЦИИ В ПРОГРАММНОЙ ИНЖЕНЕРИИ (ПРИНЦИПЫ «БЫСТРОЙ РАЗРАБОТКИ ПО»)

В начале 2001 г. ряд ведущих специалистов в области программной инженерии (Алистер Коберн, Мартин Фаулер, Джим Хайсмит, Кент Бек и др.) сформировали группу под названием Agile Alliance. Слово agile (быстрый, ловкий, стремительный) отражало в целом их подход к разработке ПО, основанный на богатом опыте участия в разнообразных проектах в течение многих лет. Этот подход под названием «Быстрая разработка ПО» (Agile software development) базируется на четырех идеях, сформулированных ими в документе «Манифест быстрой разработки ПО» (Agile Alliance's Manifesto) и заключающихся в следующем¹:

- индивидуумы и взаимодействия между ними ценятся выше процессов и инструментов;
- работающее ПО ценится выше всеобъемлющей документации;
- сотрудничество с заказчиками ценится выше формальных договоров;
- реагирование на изменения ценится выше строгого следования плану.

Центральными для быстрой разработки ПО являются простые, но достаточные правила выполнения проекта, а также ориентация на людей и коммуникацию.

¹ Коберн А. Быстрая разработка программного обеспечения: Пер. с англ. - М.: ЛОРИ, 2002.

Быстрота предполагает маневренность, характеристику, которая в настоящее время становится более важной, чем когда-либо. Распространение ПО в Интернете еще больше усилило конкуренцию между программными продуктами. Нужно не только выпускать программы на рынок и сокращать число дефектов в них, но и постоянно следить за изменяющимися требованиями пользователей и рынка.

При таком подходе технология занимает в процессе создания ПО вполне определенное место. Она повышает эффективность деятельности разработчиков при наличии любых из следующих четырех условий:

- когда она позволяет людям легче выразить свои мысли. Языки высокого уровня дают возможность более сжато выражать идеи. Некоторые языки высокого уровня позволяют человеку мыслить в технологическом пространстве, приближенном к проблемному пространству, позволяя почти не отвлекаться на мысли об ограничениях реализации;
- когда она выполняет задачи, невыполнимые вручную. Инструменты измерения и анализа собирают данные, которые иначе собрать было бы невозможно. Программисты говорят о них, как об основных инструментах;
- когда она автоматизирует утомительные и подверженные ошибкам действия. Компиляторы, электронные таблицы и средства управления конфигурацией ПО настолько важны, что некоторые программисты даже не упоминают о них как об инструментах, а просто предполагают их присутствие;
- когда она облегчает общение между людьми. В среде распределенной разработки ПО все виды инструментов обмена информацией помогают команде работать.

Технология не должна действовать против характера культурных ценностей и познавательной способности человека.

При этом следует четко понимать: при всех достоинствах быстрой разработки ПО этот подход не является универсальным и применим только в проектах определенного класса. Для характеристики таких проектов Али-

стер Коберн ввел два параметра — критичность и масштаб. **Критичность** определяется последствиями, вызываемыми дефектами в ПО, и может иметь один из четырех уровней:

- G — дефекты вызывают потерю удобства;
- D - дефекты вызывают потерю возместимых средств (материальных или финансовых);
- E — дефекты вызывают потерю невозместимых средств;
- L — дефекты создают угрозу человеческой жизни. **Масштаб** определяется количеством разработчиков, участвующих в проекте:
 - от 1 до 6 человек — малый масштаб;
 - от 6 до 20 человек - средний масштаб;
 - свыше 20 человек — большой масштаб.

По оценке Коберна, быстрая разработка ПО применима **только в проектах малого и среднего масштаба с низкой критичностью (C или D)**, Общие принципы оценки технологий в таких проектах заключаются в следующем:

- интерактивное общение лицом к лицу — это самый дешевый и быстрый способ обмена информацией;
- избыточная «тяжесть» технологии стоит дорого;
- более многочисленные команды требуют более «тяжелых» и формальных технологий;
- большая формальность подходит для проектов с большей критичностью;
- возрастание обратной связи и коммуникации сокращает потребность в промежуточных и конечных продуктах;
- дисциплина, умение и понимание противостоят процессу, формальности и документированию;
- потеря эффективности в некритических видах деятельности вполне допустима.

Одним из наиболее известных примеров практической реализации подхода быстрой разработки ПО является «Экстремальное программирование» (Extreme Programming — XP¹). Далее приведено краткое изложение этого метода.

1. В команде работает от трех до десяти программистов. Один или несколько заказчиков должны быть на месте и обеспечивать текущую экспертизу. Все работают в одной комнате или в смежных комнатах, желательно вокруг установленных вместе компьютеров, которых вдвое меньше числа программистов, с мониторами, повернутыми наружу из круга.

2. Программы разрабатываются трехнедельными периодами, или итерациями. На каждой итерации производится работающий, протестированный код, который может сразу использоваться заказчиками. Собранная система переправляется к конечным пользователям в конце каждого периода выпуска версий, который может занимать от двух до пяти итераций.

3. Единицей собираемых требований к ПО является «пользовательская история» (user story), описывающая с точки зрения пользователя функциональность, которая может быть разработана за одну итерацию. Заказчики записывают истории на простых индексных карточках. Заказчики и программисты договариваются о планах на следующую итерацию таким образом:

- программисты оценивают время для завершения работы с каждой карточкой;
- заказчики расставляют приоритеты, изменяют и пересматривают их при необходимости, чтобы самые ценные истории с наибольшей вероятностью были выполнены в выделенный период времени.

Разработка истории начинается с ее обсуждения программистами и экспертом-заказчиком. Поскольку это обсуждение проходит в обязательном порядке, текст, записанный на карточке этой истории, должен быть очень кратким, достаточным лишь для напоминания, о чем пойдет разговор. Понимание требований растет благодаря этим разговорам и любым картинкам или документам, которые участники сочтут необходимыми.

¹ Бек К. Экстремальное программирование: Пер. с англ. — СПб.: Питер, 2002.

4. Программисты работают парами и следуют строгим стандартам кодирования, установленным ими в начале проекта. Они создают модульные тесты для всего, что они пишут, и добиваются, чтобы эти тесты выполнялись каждый раз при сдаче кода на обязательный контроль версий и в систему управления конфигурацией.

5. В любое время любые два программиста, сидящие рядом, могут изменить любую строку кода системы. Каждый раз, когда два программиста обнаруживают секцию кода, который кажется трудным для понимания или чрезмерно сложным, они должны его переработать, чтобы упростить и улучшить. Они должны поддерживать общий проект в как можно более простом состоянии, а код как можно более ясным. Эта постоянная реорганизация становится возможной благодаря всестороннему модульному тестированию. Она также возможна, поскольку назначения пар программистов чередуются примерно раз в день, и поэтому знание об изменениях в структуре кода передается через группу, благодаря изменению партнерства.

6. В то время как программисты работают, заказчики занимаются тремя вещами: они посещают программистов, чтобы прояснять идеи, пишут приемочные тесты системы для прогона во время итерации и в ее конце выбирают истории для реализации в следующей итерации. По своему решению они могут участвовать в проекте полное рабочее время или только часть времени.

7. Каждый день команда проводит оперативные совещания, на которых программисты рассказывают, над чем они работают, что продвигается хорошо и в чем требуется помощь. На этих совещаниях все стоят, чтобы они быстро заканчивались. В конце каждой итерации проводится другое совещание, на котором они оценивают, что было сделано хорошо, и над чем нужно работать в следующий раз. Этот перечень вывешивается, чтобы все могли его видеть, работая во время следующей итерации.

8. Один человек в команде назначается «наставником». Вместе с участниками команды он оценивает использование ими основных приемов: парного

программирования и тестирования, ротации пар, поддержания простоты проектных решений, коммуникации и т.д.

Одно из несомненных достоинств XP заключается в обратной связи, которая возникает достаточно быстро. Заказчики получают ее в отношении последствий реализации их требований, представленных в ходе сеанса планирования. Через несколько дней они видят работающий код и могут соответственно скорректировать свои представления о том, что в действительности следует программировать. Изменяя проект, программисты получают быструю обратную связь благодаря модульному и приемочному тестированию. Они получают реакцию на процесс разработки каждые несколько недель, благодаря итерационным циклам.

XP использует общение — сильную сторону людей. Парная работа и быстрая ответная реакция компенсирует склонность людей к совершению ошибок.

XP — метод с высокой дисциплиной. Он требует приверженности строгому кодированию и стандартам проектирования, многочисленным комплектам модульных тестов, которые должны регулярно выполняться, качественному приемочному тестированию, постоянной работе в парах, бдительности при поддержании простоты проектных решений и постоянной реорганизации.

В своем определении XP включает набор приемов и методик, которые команде нужно освоить: игру планирования, ежедневное оперативное совещание, реорганизацию и опережающую разработку тестов.

XP предназначен для небольших компактных команд, нацеленных на получение как можно более высокого качества и продуктивности. Метод достигает этого посредством насыщенной, неформальной коммуникации, приданием на персональном уровне особого значения умению и навыкам, дисциплине и пониманию, сводя к минимуму все промежуточные рабочие продукты.

! Следует запомнить

1. Методологическую основу проектирования ПО составляет системный подход. Он реализует представление сложного объекта в виде иерархической системы взаимосвязанных моделей, позволяющих фиксировать целостные свойства объекта, его структуру и динамику.

2. Проектирование ПО представляет собой процесс создания спецификаций ПО на основе исходных требований к нему. Проектирование ПО сводится к последовательному уточнению его спецификаций на различных стадиях процесса создания ПО.

3. Накопленный к настоящему времени опыт создания систем ПО показывает, что это логически сложная, трудоемкая и длительная работа, требующая высокой квалификации участвующих в ней специалистов.

4. Неотъемлемыми свойствами ПО являются сложность, согласованность, изменяемость и незримость.

5. Объективная потребность контролировать процесс разработки сложных систем ПО, прогнозировать и гарантировать стоимость разработки, сроки и качество результатов привела к необходимости перехода от кустарных к индустриальным способам создания ПО и появлению совокупности инженерных методов и средств создания ПО, объединенных общим названием «программная инженерия».

Основные понятия

Информационная система, программное обеспечение, проект, проектирование, программная инженерия, «быстрая разработка ПО».

? Вопросы для самоконтроля

1. В чем состоит цель проектирования ПО?
2. Каковы основные особенности и проблемы современных крупномасштабных проектов программных систем?
3. В чем заключаются основные причины неудач проектов и каким представляется выход из положения?

4. Что означает «быстрая разработка ПО»?

ГЛАВА 1

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой жизненный цикл программного обеспечения (ЖЦ ПО) и какие процессы входят в его состав.*
- *Что такое модель ЖЦПО.*
- *Какие стадии включает в себя жизненный цикл любого ПО.*
- *В чем заключаются каскадная и итерационная модели ЖЦ ПО.*
- *Какие требования предъявляются к уровню зрелости организаций — разработчиков ПО.*

1.1.

НОРМАТИВНО-МЕТОДИЧЕСКОЕ ОБЕСПЕЧЕНИЕ СОЗДАНИЯ ПО

Разработка больших проектов, связанная с работой коллективов размером в несколько десятков и даже сотен человек из нескольких организаций, немиссима без совокупности нормативно-методических документов, регламентиру-

ющих различные аспекты процессов деятельности разработчиков¹. Комплекс таких документов называют **нормативно-методическим обеспечением (НМО)**. Эти документы регламентируют:

- порядок разработки, внедрения и сопровождения ПО;
- общие требования к составу ПО и связям между его компонентами, а также к его качеству;
- виды, состав и содержание проектной и программной документации.

Следование требованиям НМО позволяет создавать ПО высокого качества, соответствующее требованиям международных стандартов в области информационных технологий.

В состав НМО входят стандарты и руководящие документы, методики выполнения сложных операций, шаблоны проектных и программных документов. Все входящие в состав НМО документы классифицируются по следующим признакам:

- виду регламентации (стандарт, руководящий документ, положение, инструкция и т.п.);
- статусу регламентирующего документа (международный, отраслевой, предприятия);
- области действия документа (заказчик, подрядчик, проект);
- объекту регламентации или методического обеспечения.

Нормативной базой НМО являются международные и отечественные стандарты в области информационных технологий и прежде всего:

- международные стандарты ISO/IEC (ISO — International Organization of Standardization — Международная организация по стандартизации, IEC — International Electrotechnical Commission — Международная комиссия по электротехнике);

¹Более подробно нормативно-методические документы (стандарты) рассматриваются в учебном пособии: *Благодатских В.А., Волнин В.А., Поска-калов К.Ф.* Стандартизация разработки программных средств. — М.: Финансы и статистика, 2003.

- стандарты Российской Федерации ГОСТ Р;
- стандарты организации-заказчика.

В СССР в 70-е годы прошлого века процесс создания ПО регламентировался стандартами ГОСТ ЕСПД (Единой Системы Программной Документации — серия ГОСТ 19.XXX), которые были ориентированы на класс относительно простых программ небольшого объема, создаваемых отдельными программистами. В настоящее время эти стандарты устарели концептуально и по форме, их сроки действия закончились и использование нецелесообразно. Процессы создания автоматизированных систем (АС), частью которых является ПО АС, регламентированы стандартами ГОСТ 34.601—90 «Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания»; ГОСТ 34.602—89 «Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы» и ГОСТ 34.603—92 «Информационная технология. Виды испытаний автоматизированных систем». Однако процессы создания ПО для современных распределенных систем, функционирующих в неоднородной среде, в этих стандартах отражены недостаточно, а отдельные их положения явно устарели. В результате для каждого серьезного проекта приходится создавать комплекты нормативных и методических документов, регламентирующих процессы, этапы, работы и документы конкретных программных продуктов, поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

1.2.

СТАНДАРТ ЖИЗНЕННОГО ЦИКЛА ПО

Понятие жизненного цикла ПО (ЖЦ ПО) является одним из базовых понятий программной инженерии. *ЖЦПО определяется как период времени, ко-*

торый начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации¹.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО, является международный стандарт ISO/IEC 12207:1995 «Information Technology - Software Life Cycle Processes». Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО (его российский аналог ГОСТ Р ИСО/МЭК 12207—99 введен в действие в июле 2000 г.). В данном стандарте *процесс* определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными от других процессов, и результатами.

Каждый процесс разделен на набор действий, каждое действие — на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

В соответствии со стандартом ГОСТ Р ИСО/МЭК 12207-99 все процессы ЖЦ ПО разделены на три группы (рис. 1.1):

¹ IEEE Std 610.12 - 1990. IEEE Standard Glossary of Software Engineering Terminology.

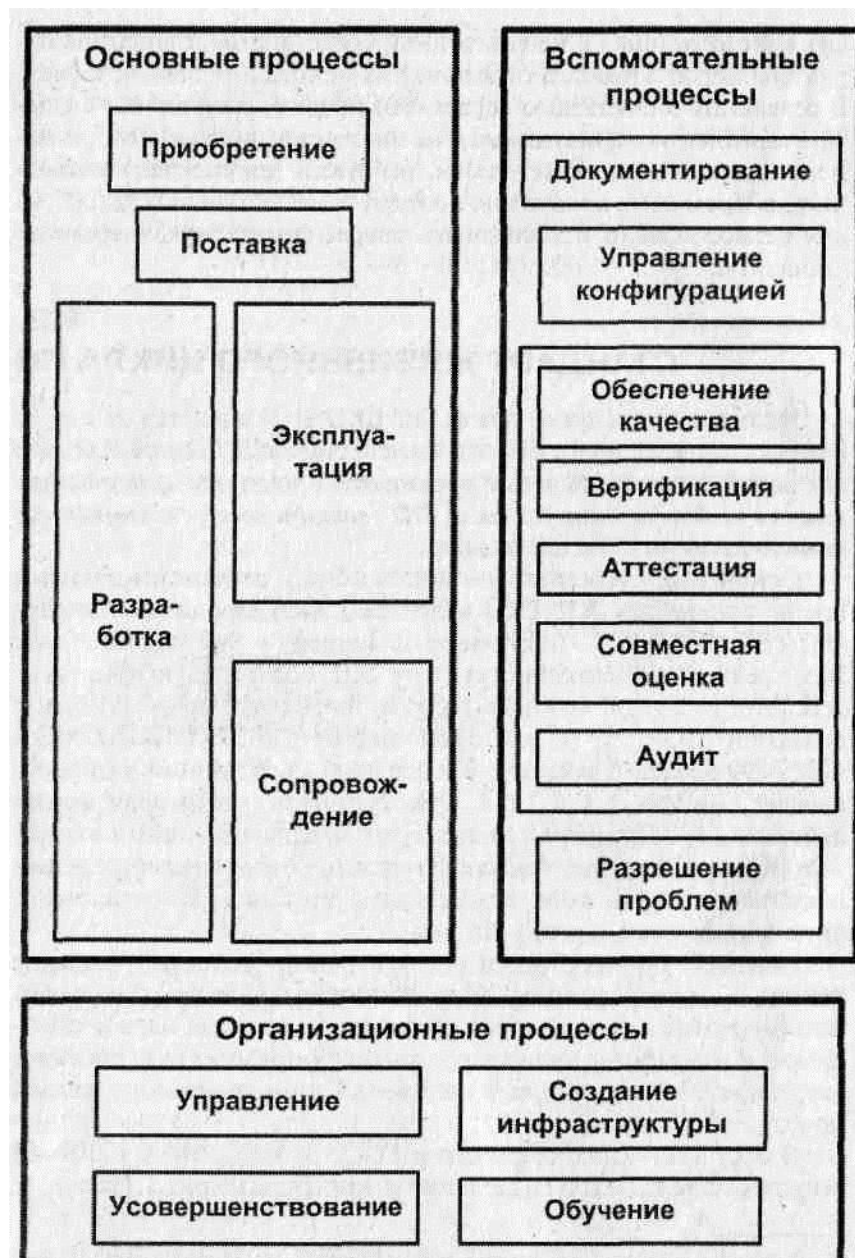


Рис. 1.1. Процессы ГОСТ Р ИСО/МЭК 12207-99

- пять основных процессов (приобретение, поставка, разработка, эксплуатация, сопровождение);
- восемь вспомогательных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
- четыре организационных процесса (управление, инфраструктура, усовершенствование, обучение).

ОСНОВНЫЕ ПРОЦЕССЫ ЖЦ ПО

Процесс приобретения (acquisition process) состоит из действий и задач заказчика, приобретающего ПО. Данный процесс охватывает следующие действия:

- 1) инициирование приобретения;
- 2) подготовку заявочных предложений;
- 3) подготовку и корректировку договора;
- 4) надзор за деятельностью поставщика;
- 5) приемку и завершение работ.

Инициирование приобретения включает следующие задачи:

- определение заказчиком своих потребностей в приобретении, разработке или усовершенствовании системы, программных продуктов или услуг;
- анализ требований к системе;
- принятие решения относительно приобретения, разработки или усовершенствования существующего ПО;
- проверку наличия необходимой документации, гарантий, сертификатов, лицензий и поддержки в случае приобретения программного продукта;
- подготовку и утверждение плана приобретения, включающего требования к системе, тип договора, ответственность сторон и т.д.

Заявочные предложения должны содержать:

- требования к системе;
- перечень программных продуктов;
- условия и соглашения;
- технические ограничения (например, среда функционирования системы).

Заявочные предложения направляются выбранному поставщику (или нескольким поставщикам в случае проведения тендера). *Поставщик* — это организация, которая заключает договор с заказчиком на поставку системы, ПО или программной услуги на условиях, оговоренных в договоре.

Подготовка и корректировка договора включают следующие задачи:

- определение заказчиком процедуры выбора поставщика, включающей критерии оценки предложений возможных поставщиков;
- выбор конкретного поставщика на основе анализа предложений;
- подготовку и заключение договора с поставщиком;
- внесение изменений (при необходимости) в договор в процессе его выполнения.

Надзор за деятельностью поставщика осуществляется в соответствии с действиями, предусмотренными в процессах *совместной оценки* и *аудита*,

В процессе *приемки* подготавливаются и выполняются необходимые тесты. *Завершение работ* по договору осуществляется в случае удовлетворения всех условий приемки.

Процесс поставки (supply process) охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Данный процесс включает следующие действия:

- 1) инициирование поставки;
- 2) подготовку ответа на заявочные предложения;
- 3) подготовку договора;
- 4) планирование;
- 5) выполнение и контроль;
- 6) проверку и оценку;
- 7) поставку и завершение работ.

Инициирование поставки заключается в рассмотрении поставщиком заявочных предложений и принятии решения согласиться с выставленными требованиями и условиями или предложить свои.

Планирование включает следующие задачи:

- принятие решения поставщиком относительно выполнения работ своими силами или с привлечением субподрядчика;
- разработку поставщиком плана управления проектом, содержащего организационную структуру проекта, разграничение ответственности, техни-

ческие требования к среде разработки и ресурсам, управление субподрядчиками и др.

Процесс разработки (development process) предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т.д.

Процесс разработки включает следующие действия:

- 1) подготовительную работу;
- 2) анализ требований к системе;
- 3) проектирование архитектуры системы;
- 4) анализ требований к ПО;
- 5) проектирование архитектуры ПО;
- 6) детальное проектирование ПО;
- 7) кодирование и тестирование ПО;
- 8) интеграцию ПО;
- 9) квалификационное тестирование ПО;
- 10) интеграцию системы;
- 11) квалификационное тестирование системы;
- 12) установку ПО;
- 13) приемку ПО.

Подготовительная работа начинается с выбора модели ЖЦ ПО, соответствующей масштабу, значимости и сложности проекта (см. подразд. 1.3). Действия и задачи процесса разработки должны соответствовать выбранной модели. Разработчик должен выбрать, адаптировать к условиям проекта и использовать согласованные с заказчиком стандарты, методы и средства разработки, а также составить план выполнения работ.

Анализ требований к системе подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, к внешним интерфейсам и т.д. Требования к системе оцениваются исходя из критериев реализуемости и возможности проверки при тестировании.

Проектирование архитектуры системы на высоком уровне заключается в определении компонентов ее оборудования, ПО и операций, выполняемых эксплуатирующим систему персоналом. Архитектура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым проектным стандартам и методам.

Анализ требований к ПО предполагает определение следующих характеристик для каждого компонента ПО:

- функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
- внешних интерфейсов;
- спецификаций надежности и безопасности;
- эргономических требований;
- требований к используемым данным;
- требований к установке и приемке;
- требований к пользовательской документации;
- требований к эксплуатации и сопровождению.

Требования к ПО оцениваются исходя из критериев соответствия требованиям к системе, реализуемости и возможности проверки при тестировании.

Проектирование архитектуры ПО включает следующие задачи (для каждого компонента ПО):

- трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
- разработку и документирование программных интерфейсов ПО и баз данных;
- разработку предварительной версии пользовательской документации;

- разработку и документирование предварительных требований к тестам и плана интеграции ПО.

Архитектура компонентов ПО должна соответствовать требованиям, предъявляемым к ним, а также принятым проектным стандартам и методам.

Детальное проектирование ПО включает следующие задачи:

- описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для их последующего самостоятельного кодирования и тестирования;
- разработку и документирование детального проекта базы данных;
- обновление (при необходимости) пользовательской документации;
- разработку и документирование требований к тестам и плана тестирования компонентов ПО;
- обновление плана интеграции ПО.

Кодирование и тестирование ПО охватывает следующие задачи:

- разработку (кодирование) и документирование каждого компонента ПО и базы данных, а также совокупности тестовых процедур и данных для их тестирования;
- тестирование каждого компонента ПО и базы данных на соответствие предъявляемым к ним требованиям. Результаты тестирования компонентов должны быть документированы;
- обновление (при необходимости) пользовательской документации;
- обновление плана интеграции ПО.

Интеграция ПО предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирование агрегированных компонентов. Для каждого из агрегированных компонентов разрабатываются наборы тестов и тестовые процедуры, предназначенные для проверки каждого из квалификационных требований при последующем квалификационном тестировании. *Квалификационное требование* — это набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как со-

ответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

Квалификационное тестирование ПО проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации. Квалификационное тестирование выполняется для каждого компонента ПО по всем разделам требований при широком варьировании тестов. При этом также проверяются полнота технической и пользовательской документации и ее адекватность самим компонентам ПО.

Интеграция системы заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, подвергается *квалификационному тестированию* на соответствие совокупности требований к ней. При этом также производится оформление и проверка полного комплекта документации на систему.

Установка ПО осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и баз данных. Если устанавливаемое ПО заменяет существующую систему, разработчик должен обеспечить их параллельное функционирование в соответствии с договором.

Приемка ПО предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика. Разработчик выполняет окончательную передачу ПО заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

Процесс эксплуатации (operation process) охватывает действия и задачи оператора — организации, эксплуатирующей систему. Данный процесс включает следующие действия:

- 1) подготовительную работу;
- 2) эксплуатационное тестирование;
- 3) эксплуатацию системы;

4) поддержку пользователей.

Подготовительная работа включает проведение оператором следующих задач:

- планирование действий и работ, выполняемых в процессе эксплуатации, и установка эксплуатационных стандартов;
- определение процедур локализации и разрешения проблем, возникающих в процессе эксплуатации.

Эксплуатационное тестирование осуществляется для каждой очередной редакции программного продукта, после чего она передается в эксплуатацию.

Эксплуатация системы выполняется в предназначенной для этого среде в соответствии с пользовательской документацией.

Поддержка пользователей заключается в оказании помощи и консультациях при обнаружении ошибок в процессе эксплуатации ПО.

Процесс сопровождения (maintenance process) предусматривает действия и задачи, выполняемые сопровождающей организацией (службой сопровождения). Данный процесс активизируется при изменениях (модификациях) программного продукта и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации либо адаптации ПО. В соответствии со стандартом IEEE—90 под *сопровождением* понимается внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Изменения, вносимые в существующее ПО, не должны нарушать его целостности. Процесс сопровождения включает перенос ПО в другую среду (миграцию) и заканчивается снятием ПО с эксплуатации.

Процесс сопровождения охватывает следующие действия:

- 1) подготовительную работу;
- 2) анализ проблем и запросов на модификацию ПО;
- 3) модификацию ПО;
- 4) проверку и приемку;
- 5) перенос ПО в другую среду;

б) снятие ПО с эксплуатации.

Подготовительная работа службы сопровождения включает следующие задачи:

- планирование действий и работ, выполняемых в процессе сопровождения;
- определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения.

Анализ проблем и запросов на модификацию ПО, выполняемый службой сопровождения, включает следующие задачи:

- анализ сообщения о возникшей проблеме или запроса на модификацию ПО относительно его влияния на организацию, существующую систему и интерфейсы с другими системами. При этом определяются следующие характеристики возможной модификации: тип (корректирующая, улучшающая, профилактическая или адаптирующая к новой среде); масштаб (размеры модификации, стоимость и время ее реализации); критичность (воздействие на производительность, надежность или безопасность);
- оценку целесообразности проведения модификации и возможных вариантов ее проведения;
- утверждение выбранного варианта модификации.

Модификация ПО предусматривает определение компонентов ПО, их версий и документации, подлежащих модификации, и внесение необходимых изменений в соответствии с правилами *процесса разработки*. Подготовленные изменения тестируются и проверяются по критериям, определенным в документации. При подтверждении корректности изменений в программах производится корректировка документации.

Проверка и приемка заключаются в проверке целостности модифицированной системы и утверждении внесенных изменений.

При *переносе ПО в другую среду* используются имеющиеся или разрабатываются новые средства переноса, затем выполняется конвертирование программ и данных в новую среду. С целью облегчить переход предусматривается

параллельная эксплуатация ПО в старой и новой среде в течение некоторого периода, когда проводится необходимое обучение пользователей работе в новой среде.

Снятие ПО с эксплуатации осуществляется по решению заказчика при участии эксплуатирующей организации, службы сопровождения и пользователей. При этом программные продукты и соответствующая документация подлежат архивированию в соответствии с договором. Аналогично переносу ПО в другую среду с целью облегчить переход к новой системе предусматривается параллельная эксплуатация старого и нового ПО в течение некоторого периода, когда выполняется необходимое обучение пользователей работе с новой системой.

1.2.2.

ВСПОМОГАТЕЛЬНЫЕ ПРОЦЕССЫ ЖИЗНЕННОГО ЦИКЛА ПО

Процесс документирования (documentation process) предусматривает формализованное описание информации, созданной в течение ЖЦ ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких, как руководство, технические специалисты и пользователи системы.

Процесс документирования включает следующие действия:

- 1) подготовительную работу;
- 2) проектирование и разработку;
- 3) выпуск документации;
- 4) сопровождение.

Процесс управления конфигурацией (configuration management process) предполагает применение административных и технических процедур на всем

протяжении ЖЦ ПО для определения состояния компонентов ПО в системе, управления модификациями ПО, описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПО, управления хранением и поставкой ПО. Согласно стандарту IEEE-90 под *конфигурацией ПО* понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО.

Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ.

Процесс управления конфигурацией включает следующие действия;

- 1) подготовительную работу;
- 2) идентификацию конфигурации;
- 3) контроль за конфигурацией;
- 4) учет состояния конфигурации;
- 5) оценку конфигурации;
- 6) управление выпуском и поставку.

Подготовительная работа заключается в планировании управления конфигурацией.

Идентификация конфигурации устанавливает правила, с помощью которых можно однозначно идентифицировать и различать компоненты ПО и их версии. Кроме того, каждому компоненту и его версиям соответствует однозначно обозначаемый комплект документации. В результате создается база для однозначного выбора и манипулирования версиями компонентов ПО, использующая ограниченную и упорядоченную систему символов, идентифицирующих различные версии ПО.

Контроль за конфигурацией предназначен для систематической оценки предполагаемых модификаций ПО и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение. Он обеспечивает контроль за состоянием и развитием компонентов ПО и их версий, а также

адекватность реально изменяющихся компонентов и их комплектной документации.

Учет состояния конфигурации представляет собой регистрацию состояния компонентов ПО, подготовку отчетов обо всех реализованных и отвергнутых модификациях версий компонентов ПО. Совокупность отчетов обеспечивает однозначное отражение текущего состояния системы и ее компонентов, а также ведение истории модификаций.

Оценка конфигурации заключается в оценке функциональной полноты компонентов ПО, а также соответствия их физического состояния текущему техническому описанию.

Управление выпуском и поставка охватывают изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком, принятым в организации.

Процесс обеспечения качества (quality assurance process) обеспечивает соответствующие гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям и утвержденным планам. Под *качеством ПО* понимается совокупность свойств, которые характеризуют способность ПО удовлетворять заданным требованиям.

Для получения достоверных оценок создаваемого ПО процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой ПО. При этом могут использоваться результаты других вспомогательных процессов, таких, как верификация, аттестация, совместная оценка, аудит и разрешение проблем.

Процесс обеспечения качества включает следующие действия:

- 1) подготовительную работу;
- 2) обеспечение качества продукта;
- 3) обеспечение качества процесса;
- 4) обеспечение прочих показателей качества системы.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса обеспечения качества с учетом используемых стандартов, методов, процедур и средств.

Обеспечение качества продукта подразумевает гарантирование полного соответствия программных продуктов и их документации требованиям заказчика, предусмотренным в договоре.

Обеспечение качества процесса предполагает гарантирование соответствия процессов ЖЦ ПО, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам.

Обеспечение прочих показателей качества системы осуществляется в соответствии с условиями договора и стандартом качества ISO 9001.

Процесс верификации (verification process) состоит в определении того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, обусловленным предшествующими действиями (*верификация* в узком смысле означает формальное доказательство правильности ПО). Для повышения эффективности верификация должна как можно раньше интегрироваться с использующими ее процессами (такими, как поставка, разработка, эксплуатация или сопровождение). Данный процесс может включать анализ, оценку и тестирование.

Верификация может проводиться с различными степенями независимости. Степень независимости может варьироваться от выполнения верификации самим исполнителем или другим специалистом данной организации до ее выполнения специалистом другой организации с различными вариациями. Если процесс верификации осуществляется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой верификации*.

Процесс верификации включает следующие действия:

- 1) подготовительную работу;
- 2) верификацию.

В процессе верификации проверяются следующие условия:

- непротиворечивость требований к системе и степень учета потребностей пользователей;
- возможности поставщика выполнить заданные требования;
- соответствие выбранных процессов ЖЦ ПО условиям договора;
- адекватность стандартов, процедур и среды разработки процессам ЖЦ ПО;
- соответствие проектных спецификаций ПО заданным требованиям;
- корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;
- соответствие кода проектным спецификациям и требованиям;
- тестируемость и корректность кода, его соответствие принятым стандартам кодирования;
- корректность интеграции компонентов ПО в систему;
- адекватность, полнота и непротиворечивость документации.

Процесс аттестации (validation process) предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению. Под *аттестацией* обычно понимается подтверждение и оценка достоверности проведенного тестирования ПО. Аттестация должна гарантировать полное соответствие ПО спецификациям, требованиям и документации, а также возможность его безопасного и надежного применения пользователем. Аттестацию рекомендуется выполнять путем тестирования во всех возможных ситуациях и использовать при этом независимых специалистов. Аттестация может проводиться на начальных стадиях ЖЦ ПО или как часть работы по приемке ПО.

Аттестация, так же как и верификация, может осуществляться с различными степенями независимости. Если процесс аттестации выполняется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой аттестации*.

Процесс аттестации включает следующие действия:

- 1) подготовительную работу;

2) аттестацию.

Процесс совместной оценки (*joint review process*) предназначен для оценки состояния работ по проекту и ПО, создаваемому при выполнении данных работ (действий). Он сосредоточен в основном на контроле планирования и управления ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.

Оценка применяется как на уровне управления проектом, так и на уровне технической реализации проекта и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя любыми сторонами, участвующими в договоре, при этом одна сторона проверяет другую.

Процесс совместной оценки включает следующие действия:

- 1) подготовительную работу;
- 2) оценку управления проектом;
- 3) техническую оценку.

Процесс аудита (*audit process*) представляет собой определение соответствия требованиям, планам и условиям договора. Аудит может выполняться двумя любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит — это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПО или процессов установленным требованиям. Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы (ревизоры) не должны иметь прямой зависимости от разработчиков ПО. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования.

Процесс аудита включает следующие действия:

- 1) подготовительную работу;
- 2) аудит.

Процесс разрешения проблем (*problem resolution process*) предусматривает анализ и решение проблем (включая обнаруженные несоответствия), незави-

симо от их происхождения или источника, которые обнаружены в ходе разработки, эксплуатации, сопровождения или других процессов. Каждая обнаруженная проблема должна быть идентифицирована, описана, проанализирована и разрешена.

Процесс разрешения проблем включает следующие действия:

- 1) подготовительную работу;
- 2) разрешение проблем.

1.2.3.

ОРГАНИЗАЦИОННЫЕ ПРОЦЕССЫ ЖИЗНЕННОГО ЦИКЛА ПО

Процесс управления (management process) состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и задачами соответствующих процессов, таких, как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

Процесс управления включает следующие действия:

- 1) инициирование и определение области управления;
- 2) планирование;
- 3) выполнение и контроль;
- 4) проверку и оценку;
- 5) завершение.

При *инициировании* менеджер должен убедиться, что необходимые для управления ресурсы (персонал, оборудование и технология) имеются в его распоряжении в достаточном количестве.

Планирование подразумевает выполнение, как минимум, следующих задач:

- составление графиков выполнения работ;
- оценку затрат;

- выделение требуемых ресурсов;
- распределение ответственности;
- оценку рисков, связанных с конкретными задачами;
- создание инфраструктуры управления.

Процесс создания инфраструктуры (*infrastructure process*) охватывает выбор и поддержку (сопровождение) технологии, стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПО. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс создания инфраструктуры включает следующие действия:

- 1) подготовительную работу;
- 2) создание инфраструктуры;
- 3) сопровождение инфраструктуры.

Процесс усовершенствования (*improvement process*) предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПО. Данный процесс включает следующие действия:

- 1) создание процесса;
- 2) оценку процесса;
- 3) усовершенствование процесса.

Усовершенствование процессов ЖЦ ПО направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу в большой степени способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.

Процесс обучения (*training process*) охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала. Приобрете-

ние, поставка, разработка, эксплуатация и сопровождение ПО в значительной степени зависят от уровня знаний и квалификации персонала. Например, разработчики ПО должны пройти необходимое обучение методам и средствам программной инженерии. Содержание процесса обучения определяется требованиями к проекту. Оно должно учитывать необходимые ресурсы и технические средства обучения. Должны быть разработаны и представлены методические материалы, необходимые для обучения пользователей в соответствии с учебным планом.

Процесс обучения включает следующие действия:

- 1) подготовительную работу;
- 2) разработку учебных материалов;
- 3) реализацию плана обучения.

1.2.4.

ВЗАИМОСВЯЗЬ МЕЖДУ ПРОЦЕССАМИ ЖЦ ПО

Процессы ЖЦ ПО, регламентируемые стандартом ISO/IEC 12207, могут использоваться различными организациями в конкретных проектах самым различным образом. Тем не менее, стандарт предлагает некоторый базовый набор взаимосвязей между процессами с различных точек зрения (или в различных аспектах), который показан на рис. 1.2. Такими аспектами являются:

- 1) договорной аспект;
- 2) аспект управления;
- 3) аспект эксплуатации;
- 4) инженерный аспект;
- 5) аспект поддержки.

В *договорном аспекте* заказчик и поставщик вступают в договорные отношения и реализуют соответственно процессы приобретения и поставки.

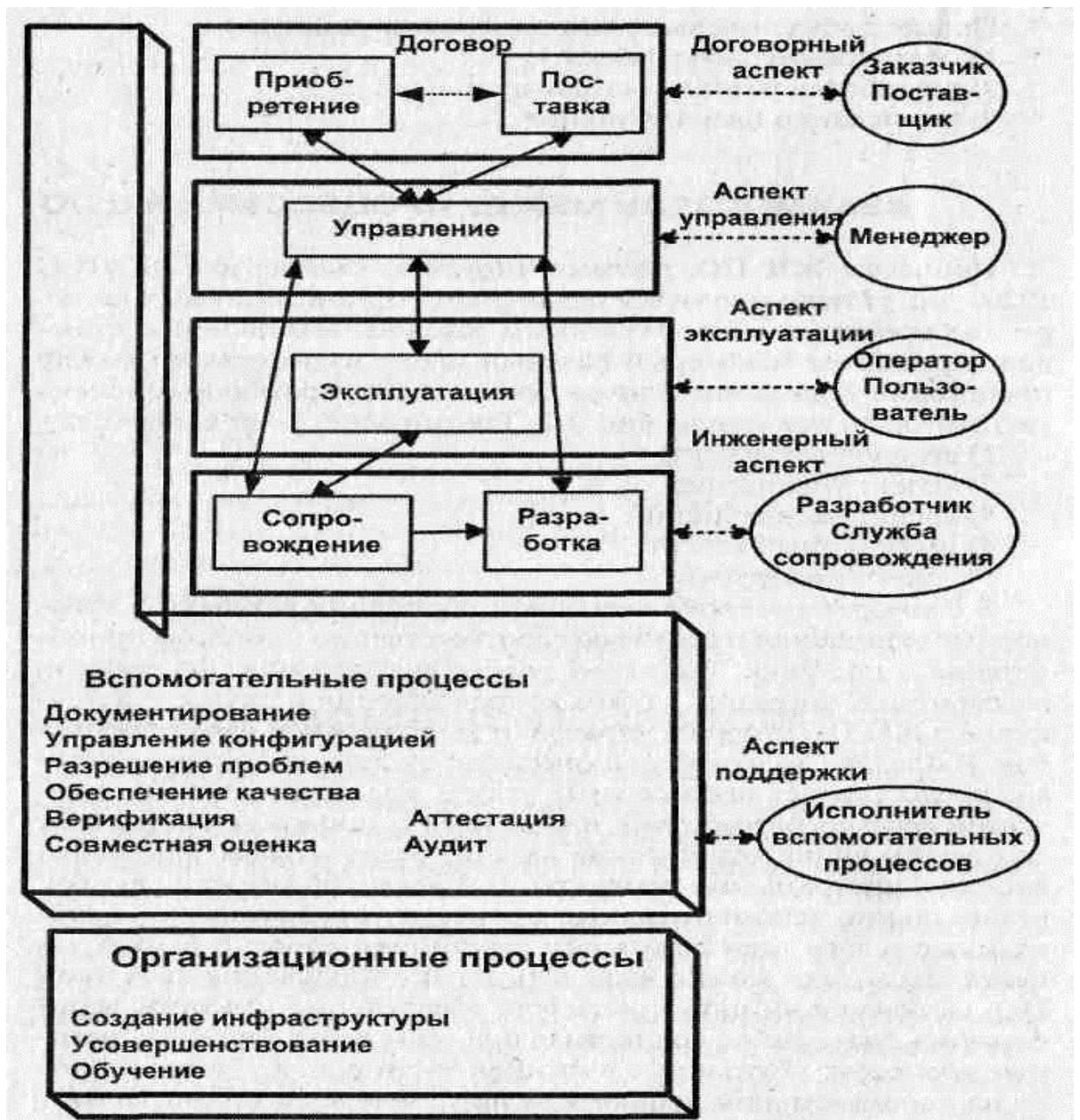


Рис. 1.2. Связи между процессами жизненного цикла ПО

В аспекте управления заказчик, поставщик, разработчик, оператор, служба сопровождения и другие участвующие в ЖЦ ПО стороны управляют выполнением своих процессов. В аспекте эксплуатации оператор, эксплуатирующий систему, предоставляет необходимые услуги пользователям. В инженерном аспекте разработчик или служба сопровождения решают соответствующие технические задачи, разрабатывая или модифицируя программные продукты. В аспекте поддержки службы, реализующие вспомогательные процессы, предоставляют необходимые услуги всем остальным участникам работ. В рамках ас-

пекта поддержки можно выделить аспект управления качеством ПО, включающий пять процессов: обеспечение качества, верификация, аттестация, совместная оценка и аудит. Организационные процессы, показанные в нижней части рис. 1.2, выполняются на корпоративном уровне или на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования остальных процессов ЖЦ ПО.

Процессы и реализующие их организации (или стороны) связаны между собой чисто функционально, при этом внутренняя структура и статус организаций никак не регламентируются. Одна и та же организация может выполнять различные роли: поставщика, разработчика и другие, и наоборот, одна и та же роль может выполняться несколькими организациями.

Взаимосвязи между процессами, описанные в стандарте, носят статический характер. Более важные динамические связи между процессами и реализующими их сторонами устанавливаются в реальных проектах. Соотношение процессов ЖЦ ПО и стадий ЖЦ, характеризующих временной аспект ЖЦ системы, рассматривается в рамках модели ЖЦ ПО.

Значение данного стандарта трудно переоценить, поскольку он формирует подход к выбору и оценке всех современных технологий и процессов создания и сопровождения ПО. Безусловно, на выбор конкретной технологии в проекте влияет целый ряд факторов, но принципы реализации и состав процессов ЖЦ ПО остаются стабильными. Большинство технологий, поставляемых ведущими производителями (IBM, Oracle, Microsoft и др.), соответствуют требованиям этого стандарта. Анализ различных технологий показывает, что общие принципы описания процессов ЖЦ ПО в стандарте ISO 12207 прошли практическую апробацию и стали общепризнанными.

МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПО

Под *моделью жизненного цикла ПО* понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Модель ЖЦ зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Стандарт ГОСТ Р ИСО/МЭК 12207-99 не предлагает конкретную модель ЖЦ ПО. Его положения являются общими для любых моделей ЖЦ, методов и технологий создания ПО. Он описывает структуру процессов ЖЦ ПО, не конкретизируя в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Модель ЖЦ ПО включает в себя:

- 1) стадии;
- 2) результаты выполнения работ на каждой стадии;
- 3) ключевые события — точки завершения работ и принятия решений.

Модель ЖЦ любого конкретного ПО определяет характер *процесса его создания*, который представляет собой совокупность упорядоченных во времени, взаимосвязанных и объединенных в *стадии* работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям. Под *стадией* понимается часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей, программных компонентов, документации), определяемого заданными для данной стадии требованиями. Стадии процесса создания ПО выделяются по соображениям рационального планирования и организации работ, заканчивающихся заданными результатами. Конкретный состав стадий ЖЦ ПО определяется используемой технологией создания ПО и соответствующими технологическими стандартами. Табл. 1.1 характеризует подходы к со-

ставу и наименованию стадий, используемые в некоторых технологиях и стандартах.

Таблица 1.1

Различные подходы к составу и наименованию стадий

ГОСТ 34	Барри У. Боэм ¹	Oracle CDM	Rational Unified Process
Формирование требований к АС. Разработка концепции АС. Техническое задание	Анализ осуществимости системы. Планирование и анализ требований к ПО	Стратегия. Анализ	Начальная стадия (Inception)
Эскизный проект. Технический проект	Проектирование изделия. Детальное проектирование	Проектирование	Разработка (Elaboration)
Рабочая документация	Кодирование	Реализация	Конструирование (Construction)
Ввод в действие. Сопровождение АС	Внедрение. Функционирование (эксплуатация) и сопровождение	Внедрение. Эксплуатация и сопровождение	Ввод в действие (Transition)

Как видно из табл. 1.1, наименования стадий в различных подходах во многом схожи и не отражают их внутреннее содержание, которое полностью определяется используемой моделью ЖЦ ПО.

На каждой стадии могут выполняться несколько процессов, определенных в стандарте ГОСТ Р ИСО/МЭК12207-99, и наоборот, один и тот же процесс

¹ Боэм Б. У. Инженерное проектирование программного обеспечения: Пер. с англ. — М.: Радио и связь, 1985

может выполняться на различных стадиях. Соотношение между процессами и стадиями также определяется используемой моделью ЖЦ ПО.

В мировой практике не существует международного стандарта, регламентирующего различные модели ЖЦ, однако существует ряд различных подходов к их классификации. Наибольшее распространение в этих классификациях получили две модели ЖЦ: каскадная и итерационная.

Крайним случаем модели ЖЦ можно считать так называемую модель «черного ящика» (black box), или «code and fix» (кодирование и исправление), что фактически означает отсутствие какой-либо модели (рис. 1.3). В этом случае выделить какие-либо рациональные стадии в процессе разработки ПО не представляется возможным, поскольку отсутствует планирование и организации работ.

Программистский фольклор, тем не менее, выделяет в такой модели следующие стадии:

- начало проекта;
- безудержный энтузиазм;
- разочарование;
- хаос;
- поиски виновных;
- наказание невиновных;
- награждение непричастных;
- определение требований к системе.

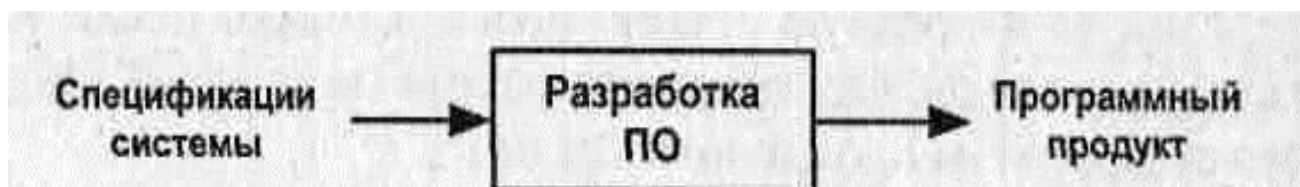


Рис. 1.3. Модель «черного ящика»

КАСКАДНАЯ МОДЕЛЬ ЖЦ

В 1970 г. эксперт в области ПО Уинстон Ройс опубликовал получившую широкую известность статью, в которой он излагал свое мнение о методике, которая позднее получила название «модель водопада» (waterfall model), или «каскадная модель» (рис. 1.4). Эта модель была разработана с учетом ограничений «тяжелых» процессов, налагавшихся на разработчиков государственными контрактами того времени. Многие ошибочно полагают, что в своей статье Ройс выступил в защиту однопроходной последовательной схемы. На самом же деле он рекомендовал подход, несколько отличный от того, который в конечном итоге вылился в концепцию «водопада» с ее строгой последовательностью стадий анализа требований, проектирования и разработки. Впоследствии эта модель была регламентирована множеством нормативных документов, в частности, широко известным стандартом Министерства обороны США Dod-STD-2167A и российскими стандартами серии ГОСТ 34. Принципиальными свойствами так называемой «чистой» каскадной модели являются следующие:

- фиксация требований к системе до ее сдачи заказчику;
- переход на очередную стадию проекта только после того, как будет полностью завершена работа на текущей стадии, без возвратов на пройденные стадии.

Каждая стадия заканчивается получением некоторых результатов, которые служат в качестве исходных данных для следующей стадии. Требования к разрабатываемому ПО, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Преимущества применения каскадной модели заключаются в следующем:

- на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадная модель может использоваться при создании ПО, для которого в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем, чтобы предоставить разработчикам свободу реализовать их технически как можно лучше. В эту категорию попадают, как правило, системы с высокой критичностью: сложные системы с большим количеством задач вычислительного характера, системы управления производственными процессами повышенной опасности и др.

В то же время этот подход обладает рядом недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. Процесс создания ПО носит, как правило, *итерационный* характер: результаты очередной стадии часто вызывают изменения в проектных решениях, выработанных на более ранних стадиях.

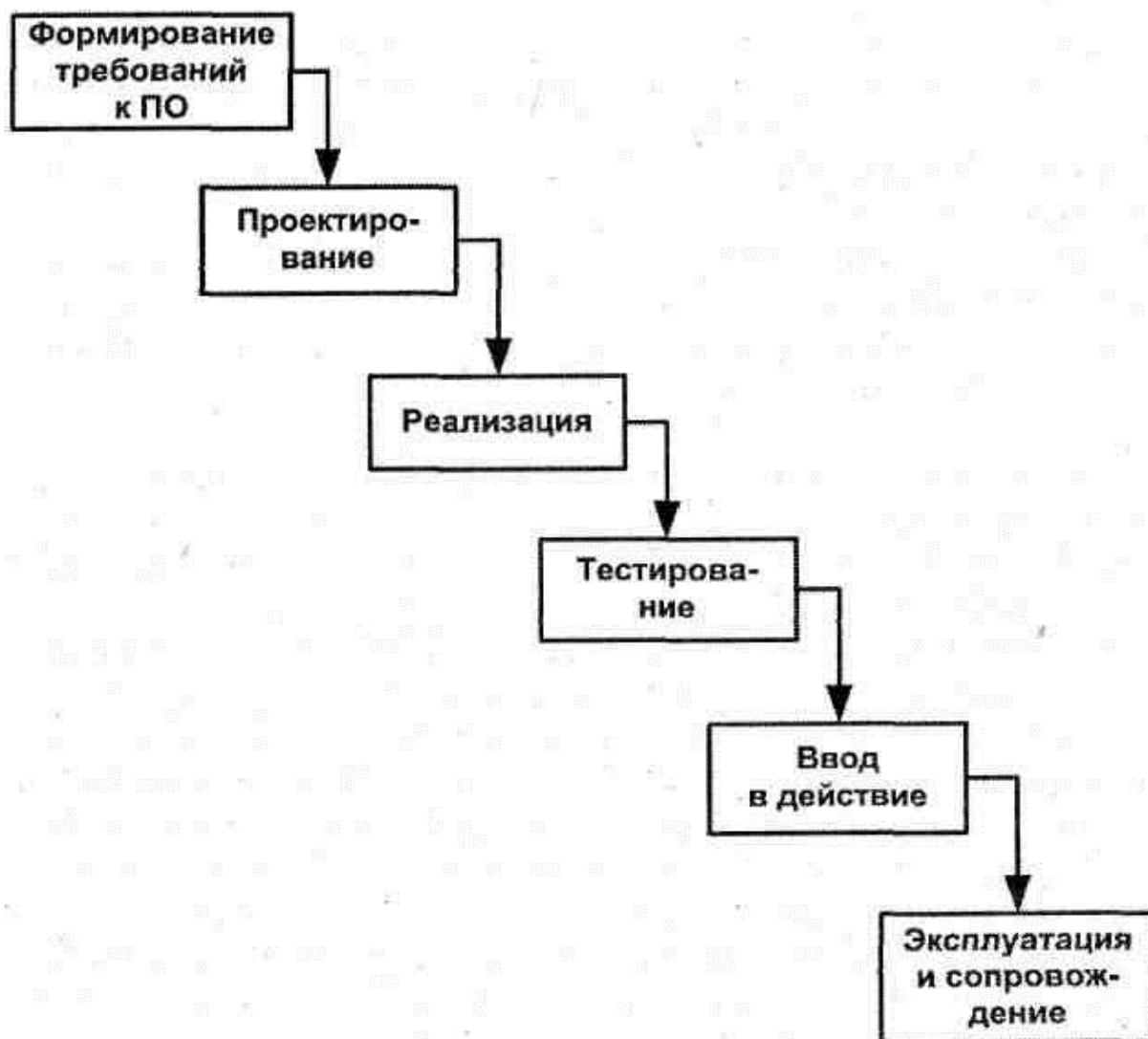


Рис. 1.4. Каскадная модель жизненного цикла ПО

Таким образом, постоянно возникает потребность в возврате к предыдущим стадиям и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимает вид, изображенный на рис. 1.5.

Основными недостатками каскадного подхода являются:

- позднее обнаружение проблем;
- выход из календарного графика, запаздывание с получением результатов;
- избыточное количество документации;
- невозможность разбить систему на части (весь продукт разрабатывается за один раз);

- высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей.

Практика показывает, что на начальной стадии проекта полностью и точно сформулировать все требования к будущей системе не удастся. Это объясняется двумя причинами: 1) пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки; 2) за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе.

В рамках каскадного подхода требования к ПО фиксируются в виде технического задания на все время его создания, а согласование получаемых результатов с пользователями производится только в точках, планируемых после завершения каждой стадии (при этом возможна корректировка результатов по замечаниям пользователей, если они не затрагивают требования, изложенные в техническом задании).

Таким образом, пользователи могут внести существенные замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО пользователи получают систему, не удовлетворяющую их потребностям. В результате приходится начинать новый проект, который может постигнуть та же участь.

В чем же заключается первоисточник проблем, связанных с каскадным подходом? Он, как уже отмечалось во введении, в заблуждении, что разработка ПО имеет много общего со строительными или инженерными проектами, например со строительством мостов. В строительных проектах задачи выполняются строго последовательно.

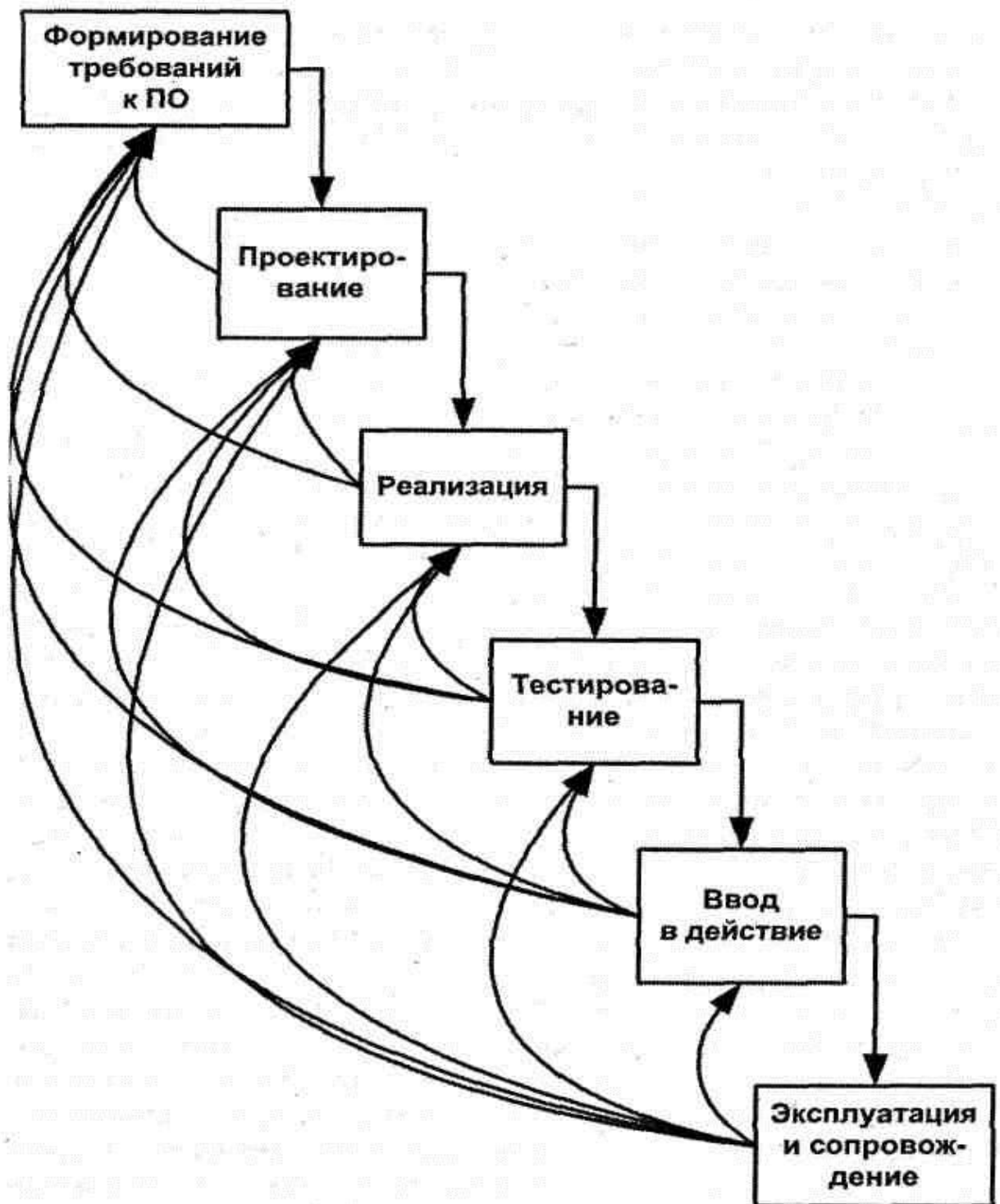


Рис. 1.5. Реальный процесс разработки ПО

Например, нельзя заложить опоры моста до тех пор, пока не будут вырыты ямы для них. Настил моста не может начаться до тех пор, пока не будет завершено строительство опор. В ранних подходах к процессу разработки ПО использовались те же принципы:

1. Группа аналитиков собирала и документировала требования.

2. Когда требования были утверждены, начиналось проектирование.
3. После утверждения проекта начиналось написание кода.
4. Каждая строка кода подлежала проверке. Если ее утверждали, то ее решалось интегрировать в продукт.

В этом заключается «чистый» каскадный подход. В свое время его расхваливали как процесс, позволяющий сделать разработку ПО более рентабельной. Считалось, что если написание кода начинать до утверждения проекта, то некоторые работы по созданию кода будут выполнены напрасно. На практике оказалось, что «строительный» подход привел к наиболее ярким примерам неудач при разработке ПО.

Первая проблема такого подхода заключается в том, что задачи по разработке ПО не так легко спланировать и определить, как задачи строительного проекта. При строительстве моста очень легко определить, когда, к примеру, этап настила выполнен наполовину. При написании кода гораздо труднее узнать, когда же он готов хотя бы наполовину. При строительстве моста очень легко оценить, какое время займет настил одного пролета, а при написании кода никто не знает, насколько большим будет конечный код и сколько времени займет написание и отладка определенной части кода.

Вторая проблема вызвана тем, что при каскадном подходе все действия по разработке системы являются последовательными: завершение одной задачи происходит до начала новой. Завершение одной задачи означает, что полученный результат — безукоризненный и что персонал, выполнявший эту задачу, может перейти к следующему проекту (как строителей после завершения одного объекта «перебрасывают» на другой). Несмотря на то что при строительстве какого-либо объекта (например, моста) можно сказать, что определенный этап работ был завершен, нельзя быть уверенным в том, что при этом были соблюдены все требования. Опыт показывает — практически всегда можно быть уверенным в том, что какие-то требования не соблюдались. Аналогичным образом при выполнении программного проекта всегда будут обнаруживаться какие-то недостатки в документации.

Попытки преобразовать действия по разработке ПО в последовательную форму всегда приводят к одному из двух возможных результатов: ранняя неудача или поздняя неудача. Успешные результаты бывают очень редко. Раннюю неудачу терпят наиболее «строго выполняемые» проекты. В таких случаях один из наиболее ранних промежуточных этапов, а именно составление спецификации требований или спецификаций проекта, никогда не будет выполнен. При каждом пересмотре документов возникают новые проблемы и сомнения, обнаруживаются недостатки и задаются вопросы, на которые нельзя дать ответы.

Наиболее распространенным результатом каскадного подхода к разработке ПО является поздняя неудача. Кажется, что проекты выполняются нормально, но только до тех пор, пока работы не вступят в завершающий этап, и тогда выясняется, что потребители недовольны созданным продуктом.

1.3.2.

ИТЕРАЦИОННАЯ МОДЕЛЬ ЖИЗНЕННОГО ЦИКЛА

Итак, опыт показал, что программные проекты в корне отличаются от строительных проектов и, следовательно, управлять ими тоже нужно по-другому. Наглядным подтверждением этого является тот факт, что к концу 1980-х гг. Министерство обороны США начало испытывать серьезные трудности с разработкой ПО в соответствии с жесткой, основанной на директивных документах и предусматривающей один проход модели, как это требовалось стандартом DoD-Std-2167A. Проведенная позже в 1999 г. проверка по выборке ранее утвержденных в Министерстве обороны проектов дала удручающие результаты. Из общего числа входящих в выборку проектов, на реализацию которых было выделено 37 млрд. долл., 75% проектов закончились неудачей или выделенные на них средства не были использованы, и только 2% выделенных средств были использованы без значительной модификации проектов. В ре-

зультате в конце 1987 г. министерство отступило от стандартов на базе каскадной модели и допустило применение итерационного подхода.

Истоки концепции итерационной разработки прослеживаются в относящихся к 1930-м годам работах эксперта по проблемам качества продукции Уолтера Шеварта из компании Bell Labs, который предложил ориентированную на повышение качества методику, состоящую из серии коротких циклов шагов по планированию, реализации, изучению и действию (plan-do-study-act, PDSA). С 1940-х годов энергичным поборником PDSA стал известный авторитет в области качества Эдвардс Деминг. В более поздних работах PDSA была исследована применительно к разработке ПО.

В середине 1980-х годов Барри Боэм предложил свой вариант итерационной модели под названием *«спиральная модель»* (spiral model) (рис. 1.6).

Принципиальные особенности спиральной модели:

- отказ от фиксации требований и назначение приоритетов пользовательским требованиям;
- разработка последовательности прототипов, начиная с требований наивысшего приоритета;
- идентификация и анализ риска на каждой итерации;
- использование каскадной модели для реализации окончательного прототипа;
- оценка результатов по завершении каждой итерации и планирование следующей итерации.

При использовании спиральной модели прикладное ПО создается в несколько итераций (витков спирали) методом прототипирования. Под *прототипом* понимается действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО. Создание прототипов осуществляется в несколько итераций, или витков спирали. Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации. На каждой итерации производит-

ся тщательная оценка риска превышения сроков и стоимости проекта, чтобы определить необходимость выполнения еще одной итерации, степень полноты и точности понимания требований к системе, а также целесообразность прекращения проекта.

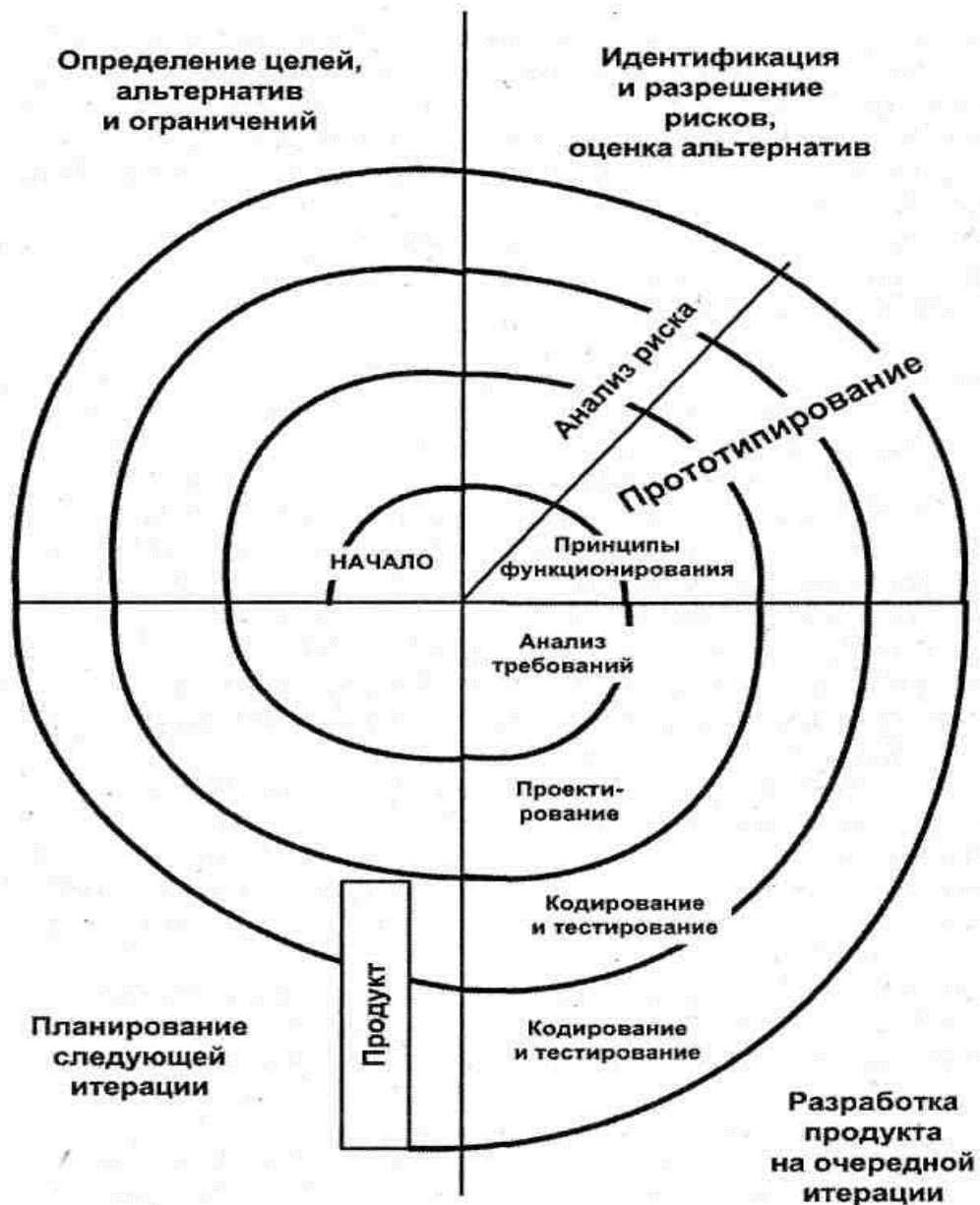


Рис. 1.6. Спиральная модель жизненного цикла ПО

Спиральная модель избавляет пользователей и разработчиков ПО от необходимости полного и точного формулирования требований к системе на начальной стадии, поскольку они уточняются на каждой итерации. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждой стадии позволяет переходить на следующую стадию, не дожидаясь полного завершения работы на текущей. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача — как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Достоинствами спиральной модели являются:

- ускорение разработки (раннее получение результата за счет прототипирования);
- постоянное участие заказчика в процессе разработки;
- разбиение большого объема работы на небольшие части;
- снижение риска (повышение вероятности предсказуемого поведения системы).

Спиральная модель не исключает использования каскадного подхода на завершающих стадиях проекта в тех случаях, когда требования к системе оказываются полностью определенными.

К недостаткам спиральной модели можно отнести:

- сложность планирования (определения количества и длительности итераций, оценки затрат и рисков);
- сложность применения модели с точки зрения менеджеров и заказчиков (из-за привычки к строгому и детальному планированию);
- напряженный режим работы для разработчиков (при краткосрочных итерациях).

Основная проблема спирального цикла — определение момента перехода на следующую стадию. Для ее решения необходимо ввести временные ограничения на каждую из стадий жизненного цикла.

Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

При использовании итерационной модели существует риск впасть в другую (по отношению к каскадной модели) крайность. Рассмотрим ее на примере следующей схемы, получившей название *«быстрого макетирования»*. Разработчики обсуждают требования к проекту с заказчиком. Затем в течение короткого промежутка времени, от четырех до шести недель, на основе понимания этих требований создается прототип системы. Разработчики вместе с заказчиком анализируют его работу. Заказчик может обнаружить, что для удовлетворения реальным потребностям прототип необходимо модифицировать. Выполнив оценку прототипа, разработчики получают возможность уточнить предъявляемые к системе требования путем детализации входных параметров. Например, заказчик может сказать, что необходимо изменить интерфейс или что отчеты, создаваемые программой, имеют неверный формат. На основе входных параметров в течение нескольких недель проводится корректировка прототипа, устраняются ошибки и добавляются определенные функции. Полученное ПО снова проверяется вместе с заказчиком. Процесс продолжается до тех пор, пока заказчик не согласится с тем, что продукт — удовлетворительный.

Легко объяснить, почему такой подход кажется привлекательным. Все усилия, направленные на удовлетворение нужд заказчика, являются позитивными. Кроме того, такая схема имеет и другие преимущества:

- *Производительность работы коллектива очень высока.* Разработчики тратят время не на создание большого количества спецификаций, которые никто и никогда не будет читать и которые станут бесполезными на следующий же день после своего выхода, а на разработку ПО. Кроме обеспечения обратной связи с заказчиком, разработчики не будут тратить время на работу, основанную на неправильном наборе допущений. Такое преимущество становится очевидным даже на ранней стадии развития проекта. В этом случае создание рабочего кода будет вопросом всего лишь нескольких недель.
- *Взаимосвязи с заказчиком являются конструктивными.* Всегда очень трудно завершить дискуссию о том, как в точности должна работать про-

грамма. Заказчик может не иметь четкого представления о том, что требуется, до тех пор, пока не начнется процесс разработки. Следовательно, сторонники такого процесса могут сказать: «Зачем составлять спецификации? Требования станут понятными в процессе работы».

Хотя процесс быстрого макетирования имеет определенные преимущества, его применимость ограничивается следующими недостатками.

- *При быстром макетировании очень тяжело привести проект к завершающей фазе.* Из-за наличия непрерывной обратной связи с заказчиком условие завершения проекта может ни когда не быть достигнуто. У разработчиков и заказчика могут возникать идеи по улучшению каждой из итераций. Постоянное усовершенствование приводит к постоянному добавлению или модификации существующих требований, и процесс может выйти из-под контроля. Заказчик никогда не удовлетворен полностью, и проект не завершится. При этом руководителю проекта требуется приложить огромные усилия для его завершения.
- *Проект, выполняемый с помощью метода быстрого макетирования, сложно планировать и финансировать.* Это положение прочно связано с предыдущим. Если проект тяжело завершить, то также тяжело составить график его выполнения и смету расходов. Не существует способа предсказать, какое время потребуется для завершения проекта.
- *Метод быстрого макетирования неприменим для разработки ПО большим коллективом разработчиков.* Этот метод хорош для небольшой группы разработчиков, работающих только с одним заказчиком. Гораздо тяжелее применить его к разработке больших систем, к которой привлекаются сотни разработчиков.
- *В результате быстрого макетирования можно не получить ничего, кроме прототипа системы.* Метод быстрого макетирования направлен в основном на достижение заданной функциональности. В конце концов, код может обладать правильными характеристиками и интерфейсами, но он никогда не станет пригодным для широкого применения.

Быстрое макетирование имеет много общего с подходом быстрой разработки ПО, рассмотренным во введении, и точно так же имеет ограниченное применение.

Еще одним примером реализации итерационной модели ЖЦ является получивший широкое распространение в 90-е годы XX века способ так называемой «быстрой разработки приложений», или **RAD** (*Rapid Application Development*). Подход RAD предусматривает наличие трех составляющих:

- небольших групп разработчиков (от 3 до 7 человек), выполняющих работы по проектированию отдельных подсистем ПО. Это обусловлено требованием максимальной управляемости коллектива;
- короткого, но тщательно проработанного производственного графика (до трех месяцев);
- повторяющегося цикла, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные в результате взаимодействия с заказчиком.

Основные принципы подхода RAD:

- разработка приложений итерациями;
- необязательность полного завершения работ на каждой из стадий жизненного цикла ПО;
- обязательность вовлечения пользователей в процесс разработки;
- применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности пользователей;
- тестирование и развитие проекта, осуществляемые одновременно с разработкой;
- ведение разработки немногочисленной хорошо управляемой командой профессионалов;
- грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

Команда разработчиков должна представлять собой группу профессионалов, имеющих опыт в проектировании, программировании и тестировании ПО, способных хорошо взаимодействовать с конечными пользователями и трансформировать их предложения в рабочие прототипы.

Жизненный цикл ПО в соответствии с подходом RAD состоит из четырех стадий:

- анализ и планирование требований;
- проектирование;
- реализация;
- внедрение.

RAD наиболее хорошо подходит для относительно небольших проектов, разрабатываемых для конкретного заказчика, и не применим для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т.е. программ, содержащих большой объем (сотни тысяч строк) уникального кода, а также систем, от которых зависит безопасность людей (например, управление самолетом или атомной электростанцией).

Естественное развитие каскадной и спиральной моделей привело к их сближению и появлению современного итерационного подхода, который по существу представляет собой рациональное сочетание этих моделей. Различные варианты итерационного подхода реализованы в большинстве современных технологий и методов: Rational Unified Process (RUP), Microsoft Solutions Framework (MSF), XP

СЕРТИФИКАЦИЯ И ОЦЕНКА ПРОЦЕССОВ СОЗДАНИЯ ПО

1.4.1.

ПОНЯТИЕ ЗРЕЛОСТИ ПРОЦЕССОВ СОЗДАНИЯ ПО. МОДЕЛЬ ОЦЕНКИ ЗРЕЛОСТИ СММ

Организации, работающие в области разработки, поставки, внедрения и сопровождения ПО и системной интеграции, все больше ощущают, что в основе их конкурентоспособности лежат качество и низкий уровень себестоимости, технологичность производства.

Руководители таких организаций не всегда могут сформировать стратегию совершенствования и развития технологии деятельности своей компании; на рынке труда специалистов необходимой квалификации явно недостаточно. Вместе с тем в области совершенствования технологических процессов разработки и эксплуатации ПО международный опыт долгие годы был недостаточно обобщен и формализован. Только в начале 1990-х годов американский Институт программной инженерии (SEI) сформировал модель технологической зрелости организаций СММ¹ (Capability Maturity Model), определив уровни технологической зрелости и их отличительные черты. В течение десятилетия СММ прошла апробацию в целом ряде организаций, ее эффективность и достоверность проверили заказывающие организации, поставщики ПО, компании, осуществляющие разработку заказного ПО, занимающиеся оффшорным программированием.

¹ Оценка и аттестация зрелости процессов создания и сопровождения программных средств и информационных систем (ISO/IEC TR 15504-СММ) / Пер. с англ. А.С. Агапова и др. — М.: Книга и бизнес, 2001.

Сегодня на западе компания-разработчик практически испытывает большие трудности с получением заказов, если она не аттестована по СММ. Заказчики требуют гарантий технологичности компании-исполнителя, гарантий того, что исполнитель не может оказать некачественную услугу.

Оценка технологической зрелости компаний может использоваться:

- заказчиком при отборе лучших исполнителей (например, в тендере);
- компаниями-производителями ПО для систематической оценки состояния своих технологических процессов и выбора направлений их совершенствования;
- компаниями, решившими пройти аттестацию, для оценки «размеров бедствия», т.е. своего текущего состояния;
- аудиторами для определения стандартной процедуры аттестации и проведения необходимых оценок;
- консалтинговыми фирмами, занимающимися реструктуризацией компаний и служб поставщиков информационных технологий и связанных с ними услуг.

По мере повышения технологической зрелости организации процессы создания и сопровождения ПО становятся более стандартизованными и согласованными. При этом формализация процессов позволяет стандартизовать ожидаемые результаты их выполнения и обеспечить предсказуемость результатов выполнения проектов.

Зрелость процессов (software process maturity) — это степень их управляемости, контролируемости и эффективности. Повышение технологической зрелости означает потенциальную возможность возрастания устойчивости процессов и указывает на степень эффективности и согласованности использования процессов создания и сопровождения ПО в рамках всей организации. Реальное использование процессов невозможно без их документирования и доведения до сведения персонала организации, без постоянного контроля и совершенствования их выполнения. Возможности хорошо продуманных процессов полностью определены. Повышение технологической зрелости процессов означает, что

эффективность и качество результатов их выполнения могут постоянно возрастать.

В организациях, достигших технологической зрелости, процессы создания и сопровождения ПО принимают статус стандарта, фиксируются в организационных структурах и определяют производственную тактику и стратегию. Введение их в статус закона влечет за собой необходимость построения необходимой инфраструктуры и создания требуемой корпоративной культуры производства, которые обеспечивают поддержку соответствующих методов, операций и процедур ведения дел даже после того, как из организации уйдут те, кто все это создал.

Модель СММ развивает положения о системе качества организации, формируя критерии ее совершенства — пять уровней технологической зрелости, которые в принципе могут быть достигнуты организацией-разработчиком. Наивысшие — четвертый и пятый уровни — это фактически характеристика организаций, овладевших методами коллективной разработки, в которых процессы создания и сопровождения ПО комплексно автоматизированы и поддерживаются технологически.

Начиная с 1990 г. SEI при поддержке правительственных структур США и организаций-разработчиков ПО постоянно развивает и совершенствует эту модель, учитывая все новейшие достижения в области создания и сопровождения ПО.

СММ представляет собой методический материал, определяющий правила формирования системы управления созданием и сопровождением ПО и методы постепенного и непрерывного повышения культуры производства. Назначение СММ — предоставление организациям-разработчикам необходимых инструкций по выбору стратегии повышения качества процессов путем анализа степени их технологической зрелости и факторов, в наибольшей степени влияющих на качество выпускаемой продукции. Фокусируя внимание на небольшом количестве наиболее критичных операций и планомерно повышая эффективность и

качество их выполнения, организация таким образом может добиться неуклонного постоянного повышения культуры создания и сопровождения ПО.

СММ - это описательная модель в том смысле, что она описывает существенные (или ключевые) атрибуты, которые определяют, на каком уровне технологической зрелости находится организация. Это нормативная модель в том смысле, что детальное описание методик устанавливает уровень организации, необходимый для выполнения проектов различной сложности и продолжительности по контрактам с правительственными структурами США. СММ не является предписанием, она не предписывает организации, каким образом развиваться. СММ описывает характеристики организации для каждого из уровней технологической зрелости, не давая каких-либо инструкций как переходить с уровня на уровень. Организации может потребоваться несколько лет для перехода с первого на второй уровень и совсем мало времени для перехода с уровня на уровень далее. Процесс совершенствования технологии создания ПО отражается в стратегических планах организации, ее структуре, используемых технологиях, общей социальной культуре и системе управления.

На каждом уровне устанавливаются требования, при выполнении которых достигается стабилизация наиболее существенных показателей процессов. Выход на каждый уровень технологической зрелости является результатом появления определенного количества компонентов в процессах создания ПО, что в свою очередь приводит к повышению их производительности и качества. На рис. 1.7 показаны пять уровней технологической зрелости СММ.

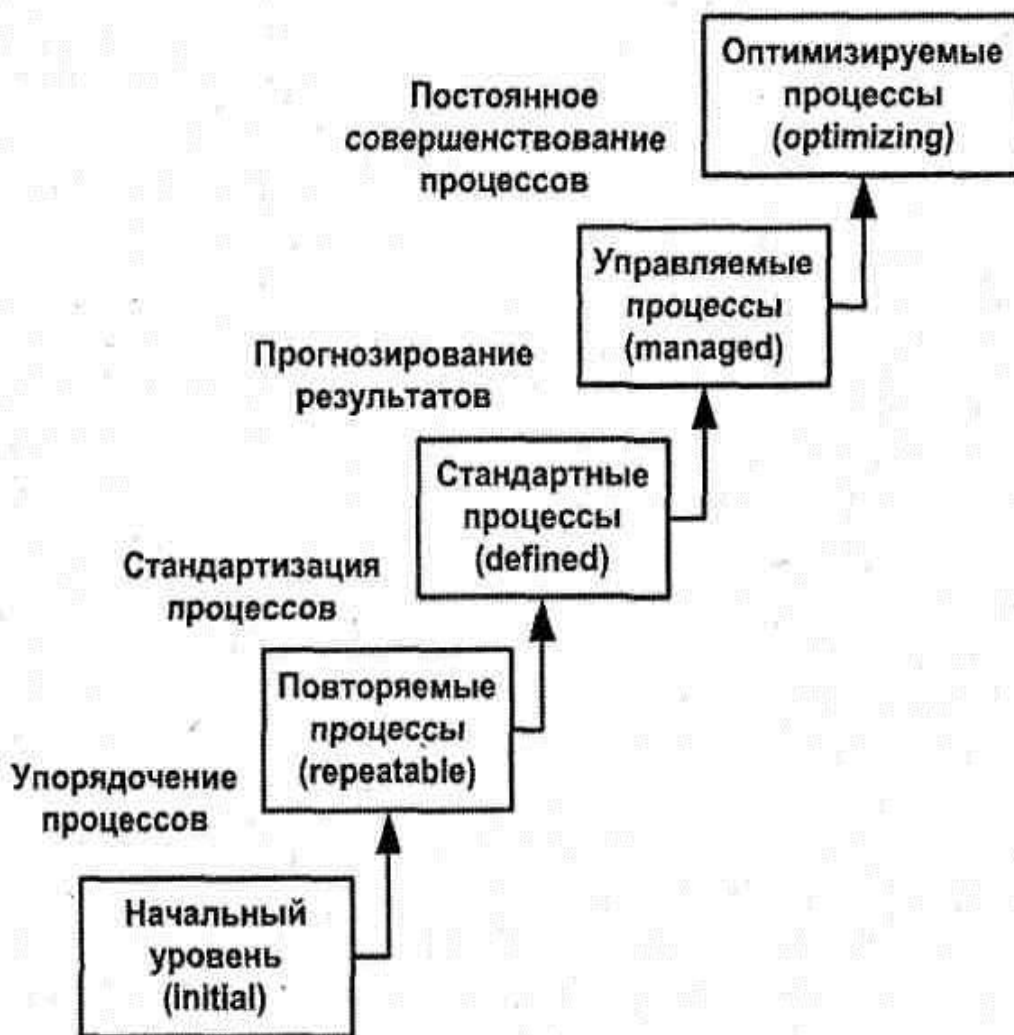


Рис. 1.7. Пять уровней технологической зрелости СММ

Надписи на стрелках определяют особенности совершенствования процессов при переходе с уровня на уровень.

Уровни со второго по пятый могут характеризоваться через операции, направленные на стандартизацию и (или) модернизацию процессов создания ПО, и через операции, составляющие сами процессы его создания. При этом первый уровень является как бы базой, фундаментом для сравнительного анализа верхних уровней.

На уровне 1 (начальном) основные процессы создания и сопровождения ПО носят случайный характер и выполняются хаотично. Успех выполнения проекта всецело зависит от индивидуальных усилий персонала. На этом уровне,

как правило, в организации не существует стабильной среды, необходимой для создания и сопровождения ПО.

Успех проекта при этом, как правило, полностью зависит от степени энергичности и опыта руководства и профессионального уровня исполнителей. Может случиться, что энергичный руководитель преодолеет все препятствия в процессе выполнения проекта и добьется выпуска действительно жизнеспособного программного продукта. Но после того, как такой руководитель оставит свой пост, исчезнет и гарантия того, что следующий подобный проект будет успешным. При отсутствии необходимого уровня управления проектом положение не сумеет спасти даже передовая технология.

Процессы на первом уровне характеризуются своей непредсказуемостью в связи с тем, что их состав, назначение и последовательность в процессе выполнения проекта меняются случайным образом в зависимости от текущей ситуации. Вследствие этого перерасходуются выделенные средства и срываются графики работ. Подготовка способных и знающих специалистов в организациях является основной из предпосылок успеха на всех уровнях зрелости, но на первом уровне это единственная возможность добиться хоть каких-либо положительных результатов.

На уровне 2 (уровне повторяемых процессов) процессы управления проектом позволяют обеспечивать текущий контроль стоимости проекта, графика его выполнения и выполняемых функций. Дисциплина выполнения проекта такова, что существует возможность прогнозирования успешности выполнения проектов по созданию аналогичных программных продуктов.

На этом уровне планирование проектных работ и управление новыми проектами базируется на опыте успешно выполненных аналогичных проектов. Основной особенностью второго уровня является наличие формализованных и документированных эффективных процессов управления проектами, что позволяет использовать положительный опыт успешно завершенных проектов. Эффективными могут называться процессы, которые документированы, реально ис-

пользуются, поддаются количественной оценке и пригодны для модернизации. Необходимо, чтобы персонал был обучен их применению.

Каждый уровень СММ, начиная со второго, характеризуется наличием ряда так называемых основных групп процессов (key process areas — КРА). Модель СММ содержит 18 таких групп, последняя версия модели СММІ — 20 групп. Уровень 2 СММ характеризуется наличием в организации следующих процессов (их наименования соответствуют СММІ):

- управление требованиями;
- управление конфигурацией;
- планирование проекта;
- мониторинг и контроль проекта;
- управление контрактами;
- измерения и анализ;
- обеспечение качества процесса и продукта.

Процессы на втором уровне можно охарактеризовать как упорядоченные в связи с тем, что они заранее планируются, и их выполнение строго контролируется, за счет чего достигается предсказуемость результатов выполнения проекта. С увеличением и усложнением проектов внимание смещается с технических аспектов их выполнения на организационные и управленческие аспекты. Выполнение процессов требует от персонала более эффективной и слаженной работы, что, в свою очередь, требует изучения документированного передового опыта в целях повышения профессионального мастерства.

На уровне 3 (уровне стандартизованных процессов) процессы создания ПО документированы, стандартизованы и представляют собой единую технологическую систему, обязательную для всех подразделений организации. Во всех проектах используется опробованная, утвержденная и возведенная в статус стандарта единая технология создания и сопровождения ПО.

На данном уровне к процессам уровня 2 добавляются следующие процессы:

- спецификация требований;

- интеграция продукта;
- верификация;
- аттестация;
- стандартизация процессов организации;
- обучение;
- интегрированное управление проектом;
- управление рисками;
- анализ и принятие решений.

Основным критерием использования и, при необходимости, корректировки процессов на этом уровне является помощь звену управления и техническим специалистам в повышении эффективности выполнения проектов. На этом уровне в организации создается специальная группа, ответственная за состав операций, из которых состоят процессы, — группа по разработке процессов создания ПО (software engineering process group — SEPG).

На основе единой технологии для каждого проекта могут разрабатываться свои процессы с учетом его особенностей. Такие процессы в СММ называются «проектно-ориентированные процессы создания ПО» (project's defined software process).

Описание каждого процесса включает в себя условия его выполнения, входные данные, стандарты и процедуры выполнения, механизмы проверки (например, экспертные оценки), выходные данные и условия завершения. В описание процесса также включаются сведения об инструментальных средствах, необходимых для его выполнения, и указание на роль, ответственную за его, выполнение.

Главное назначение уровня 4 (уровня управляемых процессов) — текущий контроль над процессами. Управление должно обеспечивать выполнение процессов в рамках заданного качества. Могут быть неизбежные потери и временные пики в измеряемых результатах, которые требуют вмешательства, но в целом система должна быть стабильна.

На уровне 4 добавляются следующие процессы:

- управление производительностью и продуктивностью;
- количественное управление проектом.

На этом уровне на практике применяется детальная оценка качества как процессов создания, так и самого создаваемого программного продукта. При этом применяются количественные критерии оценки.

В рамках всей организации разрабатывается единая программа количественного контроля производительности создания ПО и его качества. Для облегчения анализа процессов создается единая база данных процессов создания и сопровождения ПО для всех проектов, выполняемых в организации. Разрабатываются универсальные методики количественной оценки производительности процессов и качества их выполнения. Это позволяет проводить количественный анализ и оценку процессов создания и сопровождения ПО.

Результаты выполнения процессов на четвертом уровне становятся предсказуемы, поскольку они измеряемы и варьируются в рамках заданных количественных ограничений. На этом уровне появляется возможность заранее планировать заданное качество выполнения процессов и конечной продукции.

Основное назначение уровня 5 (уровня оптимизируемых процессов) — последовательное усовершенствование и модернизация процессов создания и сопровождения ПО. В целях плановой модернизации технологии создания ПО в организации создается специальное подразделение, основными обязанностями которого являются сбор данных по выполнению процессов, их анализ, модернизация имеющихся и создание новых процессов, их проверка на пилотных проектах и придание им статуса стандартов предприятия.

На уровне 5 добавляются следующие процессы:

- внедрение технологических и организационных инноваций;
- причинно-следственный анализ и разрешение проблем. Процессы создания и сопровождения ПО характеризуются

как последовательно совершенствуемые, поскольку организация прилагает постоянные усилия по их модернизации. Это совершенствование распространяется как на выявление дополнительных возможностей используемых процес-

сов, так и на создание новых оптимальных процессов и использование новых технологий.

Нововведения, которые могут принести наибольшую выгоду, стандартизуются и адаптируются в технологическую систему в рамках всей организации. Персонал, принимающий участие в выполнении проекта, анализирует дефекты и выявляет причины их возникновения. Процессы создания ПО оцениваются в целях предотвращения повторения ситуаций, приводящих к дефектам, а результаты оценки учитываются в последующих проектах.

Каждый последующий уровень дополнительно обеспечивает более полную обзорность самих процессов создания и сопровождения ПО.

На первом уровне процессы являются аморфными («черными ящиками»), и их обзорность весьма ограничена. С самого начала состав и назначение операций практически не определены, что порождает значительные трудности в определении состояния проекта и его продвижения. Требования по выполнению процессов задаются бесконтрольно. Разработка ПО в глазах менеджеров (особенно тех, кто сам не является программистом) иногда выглядит как черная магия.

На втором уровне выполнение требований пользователя и создание ПО контролируются, поскольку определена база для процессов управления проектом. Процесс создания ПО может рассматриваться как последовательность «черных ящиков», которые можно контролировать в точках перехода из одного «ящика» в другой - зафиксированных этапах. Даже если руководитель не знает, что делается «внутри ящика», точно установлено, что должно получиться в результате выполнения процесса, и определены контрольные точки его начала и завершения. Поэтому управление может распознавать проблемы в точках взаимодействия «черных ящиков» и своевременно на них реагировать.

На третьем уровне определена внутренняя структура «черных ящиков», т.е. задачи, из которых состоят процессы. Внутренняя структура представляет собой путь, по которому стандартные процессы в организации применяются в конкретных проектах. Звено управления и исполнители в необходимой степени

детализации знают свои роли и ответственность в рамках проекта. Руководство заранее подготовлено к рискам, которые могут возникнуть в процессе выполнения проекта. Так как стандартизированные и документированные процессы становятся «прозрачными» для обозрения, сотрудники, непосредственно не занятые в проекте, могут своевременно получать точные сведения о его текущем состоянии.

На четвертом уровне выполнение процессов жестко привязывается к инструментальным средствам, что дает возможность определения количественных характеристик их трудоемкости и качества выполнения. Руководители, имея объективную базу количественных измерений, получают возможность точного планирования стадий и этапов выполнения проекта, прогнозирования продвижения проекта, и могут своевременно и адекватно реагировать на возникающие проблемы. С уменьшением возможных отклонений от заданных сроков, стоимости и качества в процессе выполнения проекта их возможность предвидения результатов постоянно возрастает.

На пятом уровне в целях повышения качества продукции и повышения эффективности ее создания постоянно и планомерно проводится работа по созданию новых усовершенствованных методов и технологий создания ПО. Внимание обращается при этом не только на уже используемые, но и на новые более эффективные процессы и технологии. Руководители могут количественно оценивать влияние и эффективность изменений в технологии создания и сопровождения ПО.

Четвертый и пятый уровни редко встречаются в индустрии ПО. Так, если третьего уровня достигло в мире несколько сотен компаний, то фирм пятого уровня (по информации SEI на 2002 г.) насчитывалось 62, а четвертого — 72. При этом отметим, что объявляют о своем уровне зрелости далеко не все компании. Одни не заинтересованы в афишировании своих организационных технологий, другие выполняют сертификацию просто под давлением заказчика.

Для достижения высших уровней СММ требуется десять и более лет. Но даже уровень 3 позволяет смело выходить на международную арену. Для ис-

пользования СММ компании не надо искать сотрудников с какими-то уникальными способностями, ей достаточно понять общую идею. В описании модели СММ детально указано, что надо делать, чтобы развиваться в соответствии с этой моделью. Следовать регламентированным действиям СММ способен любой менеджер среднего класса.

Последняя версия СММ 1.1 в основном ориентирована на крупные компании, занимающиеся реализацией больших проектов, но она вполне может использоваться и группами из двух-трех человек или отдельными программистами для выполнения небольших проектов (продолжительностью до трех месяцев). В таких случаях модель СММ может сыграть жизненно важную роль, поскольку поступление новых заказов во многом определяется качеством реализации предыдущих проектов. Маленькие группы вполне удовлетворятся уровнем 2, так как для небольшого проекта отклонение от срока на пару недель неприципиально.

С 2002 г. официально распространяется специальная интеграционная версия СММІ. Это новая разработка SEI, охватывающая все аспекты деятельности компании: от разработки и выбора подрядчика до обучения, внедрения и сопровождения. Кроме того, модель СММІ расширена подходами из системной инженерии. В эту модель вошли наработки, сделанные в ходе проектирования версии СММ 2.0 (она не была закончена), основные изменения в которой были направлены на уточнение процессов для компаний четвертого и пятого уровней, что наиболее актуально для крупномасштабных американских проектов.

Модель СММ достаточно весома и важна, однако не стоит применять ее как единственную основу, определяющую весь процесс создания ПО. Она была предназначена в основном для компаний, которые занимаются разработкой ПО для Министерства обороны США. Эти системы отличаются большими размерами и длительным сроком эксплуатации, а также сложностью интерфейса с аппаратным обеспечением и другими системами ПО. Над созданием таких систем работают достаточно большие коллективы программистов, которые долж-

ны подчиняться требованиям и стандартам, разработанным Министерством обороны.

К недостаткам СММ относятся следующие:

1. Модель сосредоточена исключительно на управлении проектом, а не на процессе создания программного продукта. В модели не учтены такие важные факторы, как использование определенных методов, например прототипирования, формальных и структурных методов, средств статического анализа и т.п.

2. В модели отсутствует анализ рисков и решений, что не позволяет обнаруживать проблемы прежде, чем они окажут воздействие на процесс разработки.

3. Не определена область применения модели, хотя авторы признают, что она является универсальной и подходящей всем организациям. Однако авторы не дают четкого разграничения организаций, которые могут или не могут внедрять СММ в свою деятельность. Небольшие компании находят эту модель слишком бюрократичной. В ответ на эту критику были разработаны стратегии совершенствования технологического процесса для малых организаций.

Главной целью создания модели СММ было намерение Министерства обороны США оценить возможности поставщиков ПО. На данный момент пока не существует четких требований к достижению определенного уровня развития организаций-разработчиков. Однако принято считать, что у организации, достигшей высокого уровня, больше шансов выиграть тендер на поставку ПО. В качестве альтернативы СММ предлагается обобщенная классификация процессов совершенствования технологической зрелости, которая подходит для большинства организаций и программных проектов¹. Можно выделить несколько общих типов процессов совершенствования.

1. *Неформальный процесс*. Не имеет четко выраженной модели совершенствования. Его с успехом может использовать отдельная команда разработчи-

¹ Соммервилл И. Инженерия программного обеспечения. — 6-е изд.: Пер. с англ. — М.: Вильямс, 2002

ков. Неформальность процесса не исключает таких формальных действий, как управление конфигурацией, однако при этом сами действия и их взаимосвязи не predetermined заранее.

2. *Управляемый процесс.* Имеет подготовленную модель, которая управляет процессом совершенствования. Модель определяет действия, их график и взаимосвязи между ними.

3. *Методически обоснованный процесс.* Подразумевается, что введены в действие определенные методы (например, систематически применяются методы объектно-ориентированного проектирования). Для процессов этого типа будут полезными инструментальные средства поддержки проектирования и анализа процессов (CASE-средства).

4. *Процесс непосредственного совершенствования.* Имеет четко поставленную цель совершенствования технологического процесса, для чего существует отдельная строка в бюджете организации и определены нормы и процедуры внедрения нововведений. Частью такого процесса является количественный анализ процесса совершенствования.

Эту классификацию не назовешь четкой и исчерпывающей — некоторые процессы могут одновременно относиться к нескольким типам. Например, неформальность процесса является выбором команды разработчиков. Эта же команда может выбрать определенную методику разработки, имея при этом все возможности непосредственного совершенствования процесса. Такой процесс подпадает под классификацию *неформальный, методически обоснованный, непосредственного совершенствования.*

Необходимость приведенной классификации обусловлена тем, что она предоставляет основу для комплексного совершенствования технологии создания ПО и дает возможность организации выбирать разные типы процессов совершенствования. На рис. 1.8 показаны соотношения между разными типами программных систем и процессами совершенствования их разработки.



Рис. 1.8. Применимость процессов совершенствования

Знание типа разрабатываемого продукта сделает соответствие между программными системами и процессами совершенствования, показанное на рис. 1.8, полезным при выборе процесса совершенствования. Например, требуется создать программу поддержки перехода ПО с одной компьютерной платформы на другую. Такая программа имеет достаточно короткий срок эксплуатации, поэтому в ее разработке не требуются стандарты и специальное управление процессом совершенствования, как при создании долгоживущих систем.

Многие технологические процессы в настоящее время имеют CASE-средства поддержки, поэтому их можно назвать *поддерживаемыми процессами*. Методически обоснованные процессы поддерживаются инструментальными средствами анализа и проектирования. Эффективность средства поддержки зависит от применяемого процесса совершенствования. Например, в неформальном процессе могут использоваться типовые средства поддержки (средства прототипирования, компиляторы, средства отладки, текстовые процессоры и т.п.). Вряд ли в неформальных процессах будут использоваться на постоянной основе более специализированные средства поддержки.

Модель СММ не уникальна. Почти каждая крупная компания развивает собственные технологии создания ПО, иногда эти технологии становятся общедоступными и очень популярными. Широкая известность модели СММ связана с ее государственной поддержкой, но реальная эффективность СММ критикуется многими ведущими экспертами. Один из основных недостатков СММ связан с излишне жестким требованием не отклоняться от принципов данной модели, даже если здравый смысл подсказывает обратное. Подобные претензии на владение абсолютной истиной не могут не вызвать настороженности, поэтому в среде малых и средних компаний более популярны подходы, оставляющие гораздо больше свободы для индивидуального и коллективного творчества. Рассматриваемая далее методика SPMN занимает промежуточное место между жесткими, «тяжелыми», эффективными для крупных организаций решениями типа СММ и максимально гибкими технологиями. Она представляется оптимальным вариантом для фирм, которые, с одной стороны, хотят как можно быстрее упорядочить свою управленческую деятельность, а с другой стороны, планируют в перспективе выйти на международный уровень, где сертификация по СММ крайне желательна.

1.4.2.

МЕТОДИКА SPMN

Большинство современных технологий разработки ПО основаны на принципе улучшения процесса разработки, когда наибольшую свободу действий в выборе способа реализации имеет руководитель проекта. В то же время признанные мировые лидеры, создающие качественное ПО (их единицы во всем мире), активно используют принцип *лучшего практического навыка*. Этот принцип ориентирован на улучшение деталей работы и быстрое достижение конечного результата. В нем нет абстрактного «улучшения процесса», а есть конкретные рекомендации, использующие числовые характеристики проекта.

Другое преимущество принципа лучших практических навыков - возможность их немедленного применения в противовес «тяжелой» модели СММ, для сертификации по которой нужны годы труда. Несмотря на определенное число очень успешных результатов внедрения СММ, эта методика не получила массового признания среди небольших фирм в силу сложности и слишком больших усилий, требуемых для ее внедрения. Продолжающиеся неудачи в крупных программных проектах заставили Министерство обороны США сформировать подразделение SPMN (Software Program Managers Network), которое было призвано помочь военным быстро наладить эффективные процессы управления проектами в организациях-разработчиках ПО.

Эксперты Министерства обороны, выполнив масштабную работу по анализу хода различных проектов, представили руководству министерства заключение, в котором, в частности, говорилось: «Существующие методы разработки ПО в Министерстве обороны США не создают необходимых условий для управления крупномасштабными проектами по созданию ПО, которое является неотъемлемой частью комплексных боевых систем».

На основе этого заключения для SPMN были определены четыре главные цели ее работы.

1. Внедрить в Министерстве обороны лучшие практические навыки создания ПО.
2. Позволить руководителям проектов сфокусировать свои усилия на разработке качественного ПО, а не на следовании должностным инструкциям и формальным методикам, которые только ухудшали состояние проекта.
3. Позволить руководителям проектов использовать лучшие мировые практические навыки с учетом локальной корпоративной культуры.
4. Дать возможность быстро изучить и внедрить эти навыки в свою работу с помощью соответствующих методик обучения и программных систем.

В SPMN было создано три подразделения.

1. Группа оперативных советов определила важнейшие практические навыки (всего их набралось девять). В выявлении лучших навыков участвовали та-

кие общепризнанные эксперты, как Гради Буч, Эдвард Йордон и др. В дополнение к лучшим навыкам они разработали технологию Панели управления ходом программного проекта (Software Project Control Panel), описывающую ключевые индикаторы состояния проекта. Были также определены важнейшие цели типичного проекта, способы их количественной оценки и границы допустимых состояний. Составлен справочник ответов на вопросы, часто задаваемые руководителям проектов, и выделен набор самых плохих практик.

2. Группа периодических обновлений, состоящая из 180 специалистов по программной инженерии, которые обработали 163 методики 56 компаний и выделили 43 лучших практических навыка, расширивших и дополнивших 9 ключевых навыков.

3. Группа управления, контролирующая работу двух предыдущих групп и определяющая *способы* ее улучшения.

Подход, предложенный отделом SPMN, называется СВР (Critical Best Practices, критически важные практические навыки). Он позволяет тактическими изменениями в работе организации очень быстро (за полтора—два года) примерно на 80% достичь третьего уровня СММ (на что обычно требуется около десяти лет). При этом подход СВР проверен на сотнях реальных крупных программных проектов.

Рекомендации по применению практических навыков достаточно очевидны. В самом общем виде подход СВР предлагает:

- сфокусироваться на количественных параметрах завершения проекта (дате, бюджете, объеме);
- придумать быстро реализуемую стратегию выполнения проекта;
- измерять продвижение к цели;
- измерять активность разработки.

Результаты работы SPMN показали, что ход выполнения крупных проектов обычно находится на грани хаоса, и существует ряд факторов, от которых зависит, перейдет ли система в неуправляемое состояние. Чтобы правильно управлять проектом, надо придерживаться следующих принципов.

1. Ошибки и логические неувязки надо выявлять как можно раньше и устранять сразу после обнаружения. Между внесением ошибки разработчиком и ее выявлением должно пройти минимальное время (в проектах Министерства обороны США среднее время между внесением ошибки и ее устранением составляло 9 месяцев). Практика почасовой оплаты программистов (имевшая место при выполнении госзаказов в США) совершенно недопустима. Надо также совершенствовать механизмы выявления типичных причин ошибок и способы их устранения.

2. Необходимо планировать работу на основе правильно выбранных показателей. Невозможно реализовать крупный проект, если не подготовить в его рамках максимально подробный план всех видов деятельности с учетом производительности сотрудников, объема проекта, бюджета и других ресурсов.

3. Надо минимизировать неконтролируемые изменения проекта с учетом того, что они вносятся разработчиками на всех этапах, начиная с требований к системе и заканчивая ее пользовательскими интерфейсами.

4. Необходимо эффективно использовать сотрудников. Знания, опыт и мотивация сотрудников — важнейшие факторы успеха. Акцент в управлении проектами должен быть смещен на производительность труда, качество работы, выполнение планов и удовлетворение пользователя. Для этого требуются большие усилия по подготовке профессиональных руководителей проектов и изменения текущих способов их подготовки.

Девять лучших навыков, рекомендованных SPMN

Каждый из описываемых далее навыков полезен сам по себе, но их совместное использование значительно повышает общую эффективность. Немаловажно, что эти навыки могут быть внедрены без дополнительных расходов на оборудование, технологии и персонал.

Навык 1 — формальное управление рисками. Любой проект по разработке ПО — рискованный. Но отсутствие процедуры управления рисками в компании — это, пожалуй, самый показательный признак грядущей неудачи проекта. Поэтому необходимо уметь определять риск превышения бюджета и

времени выполнения, неверного выбора и возможного отказа оборудования, ошибок программирования и плохого сопровождения. Риск оценивается по вероятности возникновения и его последствиям.

Надо смягчать последствия рисков путем их раннего выявления и максимально ранней ликвидации, профилактической работы и изменения курса проекта «в обход» потенциальных неудач. Неустраненные риски надо отслеживать по параметрам «стоимость последствий риска» и «стоимость устранения риска». Желательно создавать резервные запасы ресурсов для устранения непредвиденных проблем.

В рамках проекта рекомендуется постоянно вести и анализировать списки 10 важнейших рисков; списки неустраненных рисков в наиболее критических точках проекта; отчеты по устраненным, неустраненным и новым рискам; учитывать возможную стоимость последствий рисков в зависимости от имеющегося резерва. SPMN советует также использовать анонимные каналы для получения информации от персонала о неизвестных «подводных камнях» в проекте, чтобы в коллективе не создавалась атмосфера замалчивания ошибочных действий (типичная рекомендация для американской корпоративной культуры). Одновременно надо «декриминализовать» сам риск. У сотрудников не должно быть никаких иллюзий о допустимости рисков, при этом необходимо понять, что каждый выявленный риск — это предупрежденный риск.

Навык 2 — соглашения об интерфейсах: пользовательских, внутренних (межмодульных) и внешних (для стыковки с другими компонентами и приложениями).

Интерфейсы программы - это необходимая часть системных требований и ее архитектуры, но руководители проектов часто забывают контролировать соответствие продукта этим соглашениям. Чем позже будут определены соглашения об интерфейсах, тем больше вероятность того, что систему придется заново проектировать, программировать и тестировать.

Для построения пользовательского интерфейса неплохо использовать подход RAD. При этом пользовательский интерфейс (как и все остальные) надо

полностью определить, согласовать с заказчиком и утвердить до начала этапов проектирования и разработки. Его описание должно быть включено в системную спецификацию на уровне определения каждого экранного поля, элемента ввода/вывода и средств навигации между формами/окнами/экранами.

Правильность интерфейса проверяет и утверждает только реальный пользователь каждого рабочего места из организации-заказчика. Для встраиваемой системы готовятся отдельные требования к ее внешнему интерфейсу.

Навык 3 — формальные проверки проекта. Нередко устранение ошибок начинается, только когда проект переходит к этапу тестирования. Такие этапы были придуманы 30 лет назад для создания небольших по сегодняшним меркам систем, и хотя в тестировании нет ничего плохого, выделять его в отдельный этап методически неверно. Стоимость этапа тестирования может достигать 40-60% стоимости всего проекта. Эти ненужные усилия можно сократить на порядок, однако немногие руководители знают, как это сделать.

Существует немало стандартных подходов раннего выявления ошибок, позволяющих обнаруживать 80% ошибок в момент их внесения, или многократные просмотры кода (выявляется 60% ошибок). Чтобы оперативно обнаружить 90—100% ошибок, надо использовать несколько подходов. Ведущие компании одновременно применяют 10 и более методик формальных проверок (анализ структуры проекта, проверки кода, редактирование документации, множественное тестирование и т. п.).

Не менее важны усилия по проверке корректности проекта на этапах формулирования требований, создания архитектуры системы и проектирования. Для выполнения формальных проверок надо использовать небольшие группы сотрудников с четко определенными ролями, привлекая при этом пользователей организации-заказчика. Персонал необходимо постоянно тренировать в умении анализировать код на наличие ошибок. Желательно отслеживать продолжительность усилий по проверкам проекта, число найденных ошибок по отношению к затраченным на их поиск усилиям и среднее время от внесения ошибки до ее устранения.

Навык 4 — управление проектом на основе метрик. Этот навык нужен для раннего обнаружения потенциальных проблем. Как уже говорилось, стоимость устранения дефекта в проекте увеличивается в геометрической прогрессии по мере роста проекта.

С помощью метрик (числовых характеристик) планируются все задачи проекта. Ход их выполнения надо регулярно отслеживать, как минимум, — по ключевым показателям (стоимость работы и производительность труда). Надо дополнительно контролировать время, затрачиваемое на устранение дефектов, и следить за важнейшими показателями, чтобы по их отклонениям (в любую сторону — например, слишком резвый старт) выявлять потенциальные «подводные камни». Метрики основываются на эмпирических данных, например, на основе результатов анализа схожих по размерам проектов.

Навык 5 — качество продукта должно контролироваться на глубоком уровне. Проблема реализации мелких деталей программного проекта очень важна при разработке ПО. Иногда мелкое на первый взгляд требование заказчика выливается в глобальную переделку проекта. Если же проект спланирован недостаточно подробно, обсуждать реальное положение дел в ходе его выполнения бессмысленно.

В проекте надо выделить задачи объемом не более 5% по продолжительности и усилиям, которые могут быть выполнены отдельной группой сотрудников как минимум на 95%. Каждая подобная задача должна быть ориентирована на выполнение однотипной работы. Результат выполнения задачи оценивается группой приемки, при этом работа не может быть принята с оговорками: она должна быть выполнена полностью и без ошибок (двоичная система оценки качества «готово/не готово»).

Навык 6 — информация о ходе проекта должна быть общедоступной. Чем больше сотрудников вовлечено в процесс контроля над ходом проекта, тем проще идентифицировать потенциальные проблемы и риски. Надо сделать показатели хода проекта доступными всем сотрудникам и заказчику и организовать канал приема анонимных сообщений о возникающих проблемах. Чаше

всего такой канал используется для сведения личных счетов, но лучше получить ложный сигнал, чем не узнать о реальной проблеме. К тому же открытость проекта — это залог снижения числа ложных сообщений.

Навык 7 — чтобы добиться высокого качества, надо отслеживать причины возникновения ошибок. Эффективность работы компании непосредственно зависит от наличия ошибок в проекте. Большинство компаний не контролируют их реальные источники: ошибки программистов, отклонения в графиках выполнения работ, превышение планируемой стоимости, неверно сформулированные требования, неправильно подготовленную документацию и плохо обученный персонал. В каждой фазе проекта ошибки должны отслеживаться формально. Для этого желательно использовать средства конфигурационного управления. Каждый случай обнаружения и устранения ошибки обязательно надо документировать.

Устранять ошибки необходимо по мере их возникновения. При этом учет ошибок удобнее всего вести в нормализованном виде (в расчете на единицу объема, например, на тысячу строк кода). Согласно принципу «снежного кома», с ростом объема проекта норма ошибок в нем увеличивается. Также надо контролировать среднее и максимальное время устранения ошибки и время от внесения до устранения ошибки в течение каждого этапа проекта и на протяжении первого года эксплуатации системы.

Навык 8 — управление конфигурацией. Неконтролируемые изменения в проекте могут быстро ввергнуть его в хаос. Поэтому на практике надо руководствоваться двумя простыми правилами:

- любую информацию, которую использует более чем один сотрудник, надо контролировать с помощью системы управления конфигурацией;
- любую информацию, учитываемую системой качества, надо контролировать с помощью системы управления конфигурацией.

Надо отслеживать все изменения в состоянии создаваемой системы, бюджете и сроках, интерфейсах, контрольных отчетах и т. п. Без систем управления конфигурацией при этом не обойтись, потому что в крупном проекте большие

объемы информации меняются очень быстро. Каждый учитываемый объект должен определяться его версией, при этом надо вести архив всех версий всех таких объектов.

Навык 9 — управление персоналом. Главный фактор успеха проекта - качество, опыт и мотивация сотрудников. Не надо забывать, что с помощью различных методик производительность труда программистов можно значительно повысить. К тому же, как бы подробно ни документировался проект, некоторые детали его архитектуры всегда хранятся только в головах разработчиков, и руководитель проекта должен помогать сотрудникам проявлять индивидуальные творческие способности.

Выявлена высокая степень корреляции между суммами, вкладываемыми в обучение персонала, и общим успехом проекта, поэтому надо постоянно проводить обучение и переподготовку сотрудников. Любые авралы необходимо минимизировать. К авралам (как это на первый взгляд ни парадоксально) обычно приводит работа более 40 часов в неделю, что говорит о неверной организации труда и скрытых ошибках в организационной структуре.

1.5.

ПРИМЕР ПРОЦЕССА

«УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ»

Одна из первых работ при проектировании ПО — сбор и упорядочение требований к нему. **Требование¹** - это условие, которому должна удовлетворять система, или свойство, которым она должна обладать, чтобы удовлетворить потребность пользователя в решении некоторой задачи и удовлетворить требования контракта, стандарта или спецификации.

¹Леффингуэлл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход: Пер. с англ. — М.: Вильямс, 2002.

Спецификация требований к ПО является основным документом, который играет определяющую роль по отношению к другим элементам плана разработки ПО. Все требования, определенные в спецификации, делятся на функциональные и нефункциональные. **Функциональные требования** определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией. Тем самым функциональные требования определяют поведение системы в процессе обработки информации. **Нефункциональные требования** не определяют поведение системы, но описывают ее атрибуты или атрибуты системного окружения. Можно выделить следующие типы нефункциональных требований:

- **требования к применению** определяют качество пользовательского интерфейса, документации и учебных курсов;
- **требования к производительности** накладывают ограничения на функциональные требования, задавая необходимую эффективность использования ресурсов, пропускную способность и время реакции;
- **требования к реализации** предписывают использовать определенные стандарты, языки программирования, операционную среду и др.;
- **требования к надежности** определяют допустимую частоту и воздействие сбоев, а также возможности восстановления;
- **требования к интерфейсу** определяют внешние сущности, с которыми может взаимодействовать система, и регламент этого взаимодействия.

Требование не должно быть проектным решением. Способ реализации требования будет определен на стадии проектирования системы. Если требование является проектным решением, то такое требование должно быть отмечено как ограничение.

К требованиям не относятся способы управления проектом, планы, методы верификации, управления конфигурацией, тестовые процедуры и т.д.

Работа с требованиями порождает целый ряд проблем:

- требования не очевидны, приходят из разных источников, их трудно сформулировать словами из-за внутренней неоднозначности языка;

- требования разнообразны по типам и детальности, могут достигать огромного количества, которое трудно контролировать;
- требования разнообразны по значимости (обязательность, риск, важность, стабильность);
- требования связаны между собой и с другими проектными данными, изменяются в жизненном цикле ПО.

Изначально требования собираются в виде протоколов совещаний и интервью с заказчиками и пользователями, копий и оригиналов различных документов, отчетов существующей системы и массы других материалов. Потом их начинают упорядочивать и «очищать» от противоречий. Затем на их основе вырабатываются требования к компонентам системы - базам данных, программным и техническим средствам. При этом аналитику, проводящему обследование, приходится иметь дело с большим количеством неструктурированных, часто противоречивых требований и пожеланий, разбросанных по всевозможным соглашениям о намерениях, приложениям к договорам, протоколам рабочих совещаний, черновым материалам обследований. Без организованных усилий по регистрации и контролю за выполнением этих требований велик риск их «потерять». Кроме того, известно, что ошибки в требованиях — самые дорогостоящие и самые распространенные. Переделка ПО обычно занимает 40—50% бюджета проекта, при этом ошибки в требованиях отнимают наибольшую часть стоимости переделки ПО (>70%) и 30-40% общей стоимости бюджета проекта.

Наиболее распространенные методы проектирования ПО сосредоточены на *моделировании* требований с помощью различного рода диаграмм. Но в данном случае мы имеем в виду *управление* требованиями. Эти два понятия — моделирование и управление - не являются противоречивыми или несовместимыми.

В реальных проектах пользовательские требования зачастую должным образом не документируются; в свое оправдание разработчики говорят, что это требует слишком много времени, требования слишком часто меняются и, кроме того, пользователи сами не знают, что им нужно. Таким образом, обычно пола-

гаются на методы и средства прототипирования, с помощью которых можно наглядно продемонстрировать всю важную проектную работу, а также выявить реальные требования к системе. Это порождает одну главную проблему: невозможность сколько-нибудь организованным способом *управлять* требованиями. Как можно в любой момент времени сказать, какие требования необходимо выполнить, а какие можно отложить на более позднее время? Структурные и объектно-ориентированные методы не дают ответа на этот вопрос, поскольку они предназначены в первую очередь для *понимания* и *объяснения* требований, а не для управления ими в динамике.

Именно *динамическая* составляющая управления требованиями обычно вызывает наибольшие трудности, поскольку сами требования и их приоритеты изменяются в течение проекта. Большинство крупных проектов включает сотни требований, а многие — даже тысячи (например, проект самолета Боинг-777, который называли мешком программ с крыльями, включал, по некоторым данным, около 300 000 требований). Кроме того, некоторые требования зависят от других требований, а некоторые в свою очередь порождают другие требования.

Все это подразумевает необходимость в методах и средствах для описания зависимостей между требованиями и управления большим количеством таких зависимостей. В решении данной проблемы могут частично помочь структурный анализ и объектно-ориентированный анализ, но эти методы традиционно игнорируют *атрибуты* требований, такие, как приоритет, стоимость, риск, план и разработчик, который занимается его реализацией. В результате проектным командам, испытывающим потребность в управлении требованиями, приходилось использовать доморощенные средства, базирующиеся на электронных таблицах, текстовых процессорах или наспех созданных приложениях, чтобы обеспечить хотя бы некоторую степень автоматизированной поддержки.

Управление требованиями (requirements management) представляет собой:

- систематический подход к выявлению, организации и документированию требований к системе;

- процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к системе, и обеспечивающий его выполнение.

Модель СММ характеризует деятельность по управлению требованиями следующим образом. Управление требованиями осуществляется для того, чтобы:

- достичь соглашения с заказчиком и пользователями о том, что система должна делать;
- улучшить понимание требований к системе со стороны разработчиков;
- определить границы системы;
- определить базис для планирования.

Понимание требований служит основой соглашения между заказчиком и командой разработчиков, которое является основным документом, определяющим все последующие действия. Формируется *базовый уровень требований*, на основе которого осуществляется управление разработкой ПО. Модель СММ предусматривает, чтобы все планы, графики и рабочие программные продукты разрабатывались и, если нужно, модифицировались в соответствии с требованиями, налагаемыми на ПО. Для осуществления данного процесса менеджеры проекта и заинтересованные лица (включая представителей заказчика и пользователей) должны документировать и пересматривать требования к ПО.

Например, в технологии Rational Unified Process определяются пять уровней зрелости процесса управления требованиями:

- 1) требования записаны в согласованном формате;
- 2) требования организованы, используются стандартные форматы и метаданные о требованиях, поддерживается контроль версий;
- 3) требования структурированы в соответствии со своими типами (функциональными и нефункциональными);
- 4) требования отслеживаются в соответствии с их типом;
- 5) требования интегрируются с процессами управления изменениями, моделирования, кодирования и т.д.

Чтобы соответствовать первому уровню, достаточно взять стандартный текстовый редактор или электронную таблицу для хранения требований; при этом принципы их использования разными группами команды разработки не стандартизованы. На втором уровне документы с описанием требований должны иметь согласованный формат, следовать определенной схеме нумерации и контроля версий. Уровень структурированных требований означает переход к созданию стандартных спецификаций с целым рядом атрибутов (приоритет требования, риск, стоимость и др.). Следующие уровни ставят задачу отслеживания зависимостей между различными требованиями, а также их влияния на другие модели в жизненном цикле ПО.

Чтобы добиться осуществления целей управления требованиями и соответствия модели СММ в области управления требованиями, необходимо выделить определенные ресурсы. Нужно обучить участников разработки деятельности по управлению требованиями. Обучение должно предусматривать изложение методов и стандартов, а также способствовать формированию понимания командой разработчиков особенностей предметной области и существующих в ней проблем. Помимо обучения, организация процесса управления требованиями предусматривает:

- организацию соответствующей инфраструктуры (ответственные исполнители, инструментальные средства, средства взаимодействия (интранет/интернет, электронная почта));
- определение источников возникновения и механизмов выявления требований;
- определение процедуры обсуждения требований и правил принятия решений по ним;
- определение механизмов регистрации и обработки изменений требований и принятия решений по ним.

В процессе работы с требованиями выделяются следующие этапы:

- определение типов требований и групп участников проекта, работающих с ними;

- первичный сбор требований, их классификация и занесение в базу данных требований;
- использование базы данных требований для управления проектом.

Требования, вносимые в базу данных, обладают следующим стандартным набором атрибутов, который может быть расширен при необходимости:

- приоритет (высокий, средний, низкий);
- статус (предложено, одобрено, утверждено, реализовано, верифицировано);
- стоимость (высокая, средняя, низкая или числовое значение);
- сложность реализации (высокая, средняя, низкая);
- стабильность (высокая, средняя, низкая);
- исполнитель.

Еще одним важным аспектом управления требованиями является трассировка. **Трассировка требований** — это установка связей требований с другими требованиями или проектными решениями. Цель трассировки требований:

- убедиться, что все требования к системе выполнены в процессе реализации;
- убедиться, что ПО делает только то, что предполагалось;
- облегчить внесение изменений в ПО (управление изменениями).

С помощью трассировки требований можно анализировать воздействие изменения до того, как оно произведено, а также определять, на какие компоненты повлияет внесение изменения. Все принятые изменения полностью отслеживаются. Изменения считаются неотъемлемой составной частью действий по разработке ПО. Вместо «замороженных» спецификаций формируется стабильный базовый уровень требований, которые тщательно изучены, документированы и контролируются соответствующими системами, обеспечивающими управление изменениями.

В рамках процесса управления требованиями необходимо иметь возможность расстановки приоритетов требований и их изменения. Один из наиболее полезных способов расстановки приоритетов предлагает учитывать два измере-

ния: важность и срочность, которые оцениваются по двоичной шкале. В результате получаются четыре комбинации для определения приоритетов:

- требования с высоким приоритетом - важные (пользователям нужны данные функции) и срочные (они необходимы в данной версии). Некоторые требования приходится включать в эту категорию согласно контрактным или юридическим обязательствам;
- требования со средним приоритетом — важные (пользователям нужны данные функции), но не срочные (они могут подождать до следующей версии);
- требования с низким приоритетом - не важные (пользователи при необходимости могут обойтись без этих функций), и не срочные (они могут ждать сколько угодно);
- требования, кажущиеся срочными, но в действительности не являющиеся важными, вообще не заслуживают внимания и не приносят никакой ценности.

При такой расстановке приоритетов в проекте стратегия его выполнения будет заключаться в следующем: в первую очередь сосредоточиться на требованиях с высоким приоритетом, затем сосредоточиться на требованиях со средним приоритетом, и в последнюю очередь, если останется время, заняться требованиями с низким приоритетом. Если не следовать такой стратегии с самого начала проекта, то к концу он окажется в кризисной ситуации. Дополнительными факторами ранжирования требований по приоритетам являются:

- существенное влияние на архитектуру системы;
- рискованные, сложные для реализации или срочные функции;
- применение новой, непробированной технологии;
- значимость в экономических процессах.

Управление требованиями относится к процессам, техническая поддержка которых не требует больших финансовых затрат, но они способны ощутимо повысить качество создаваемой системы.

ПРИМЕР ПРОЦЕССА

«УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ ПО»

Управление конфигурацией ПО¹ (см. подразд. 1.2.2) является одним из наиболее важных вспомогательных процессов жизненного цикла ПО. Цель управления конфигурацией — обеспечить управляемость и контролируемость процессов разработки и сопровождения ПО. Для этого необходима точная и достоверная информация о состоянии ПО и его компонентов в каждый момент времени, а также обо всех предполагаемых и выполненных в процессе сопровождения изменениях.

Изменения, вносимые в ПО, — это некоторые события, которые имеют автора, назначение, содержание и время исполнения. Они могут быть независимыми, согласованными (взаимосвязанными) или альтернативными (взаимоисключающими). Изменения разрабатываются и реализуются в разное время, вследствие чего корректность группы изменений может зависеть от синхронности их внесения в различные компоненты определенной версии ПО.

Изменения разделяются на следующие группы:

- срочные изменения, которые должны не только быть внесены в очередную версию ПО, но и сообщены пользователям для оперативной модификации ПО до внедрения официальной версии;
- изменения, которые целесообразно внести в очередную версию с учетом затрат на их реализацию ПО;
- изменения, которые требуют дополнительного анализа целесообразности и эффективности их реализации, в последующих версиях и могут не внедряться в очередную версию ПО;

¹ *Лунаев В. В.* Документирование и управление конфигурацией программных средств. Методы и стандарты. — М.: Синтег, 1998.

- изменения, которые не оправдывают затрат на их внесение или практически не влияют на качество и эффективность ПО, и поэтому не подлежат реализации.

Изменение конфигурации ПО должно планироваться и предусматривать в плане действия с четкими разделами:

- почему и с какой целью производится модификация ПО;
- кто выполняет и санкционирует проведение изменений;
- какие действия и процедуры должны быть выполнены для реализации изменений;
- когда по срокам и в координации с какими другими процедурами следует реализовать определенную модификацию ПО;
- как и с использованием каких средств и ресурсов должны быть выполнены запланированные изменения ПО.

Все проанализированные изменения регистрируются. Для принятых к внедрению изменений разрабатывается план доработок ПО и определяется ответственный за каждую модификацию ПО.

В процессе управления конфигурацией необходимо построить и использовать компактные и наглядные схемы однозначной иерархической идентификации:

- объектов - модулей и компонентов ПО, подвергающихся модификации;
- проводимых изменений (с целью отслеживания истории модификации компонентов любого уровня);
- специалистов, участвующих в управлении конфигурацией, и их прав на доступ к определенным компонентам ПО на конкретных стадиях разработки, реализации и утверждения изменений.

Для решения задач управления конфигурацией и изменениями (в современных технологиях эти процессы объединяются в один) применяются методы и средства, обеспечивающие идентификацию состояния компонентов, учет номенклатуры всех компонентов и модификаций системы в целом, контроль за вносимыми изменениями в компоненты, структуру системы и ее функции, а

также координированное управление развитием функций и улучшением характеристик системы.

Наиболее развитые современные средства управления конфигурацией и изменениями ПО обладают следующими возможностями:

- хранение в БД управления конфигурацией полных хронологий версий каждого объекта, созданного или измененного в процессе разработки системы (к ним относятся исходный программный код, библиотеки, исполняемые программы, модели, документация, тесты, web-страницы и каталоги);
- поддержка географически удаленных групп разработчиков;
- контроль изменений, вносимых во все объекты системы;
- сборка версий ПО из компонентов проекта.

Средства контроля версий могут использоваться в рабочих группах. Система блокировок, реализованная в них, позволяет предотвратить одновременное внесение изменений в один и тот же объект, давая разработчикам в то же время возможность работать с собственными версиями общего объекта с разрешением конфликтов между ними.

Средства контроля изменений обычно используются в комплексе со средствами управления требованиями. Поступающее требование или замечание проходит четыре этапа обработки:

- регистрация — внесение замечания в базу данных;
- распределение — назначение ответственного исполнителя и сроков исполнения;
- исполнение — устранение замечания, которое, в свою очередь может вызвать дополнительные замечания или требования на дополнительные работы;
- приемка — приемка работ и снятие их с контроля или направление на доработку.

Отчетные возможности включают множество разновидностей графиков и диаграмм, отражающих состояние проекта, срезы по различным компонентам

проекта, разработчикам и тестировщикам. С их помощью можно наглядно показать состояние работы над проектом и динамику ее развития.

Наличие средств управления конфигурацией и изменениями означает обеспечение целостности проекта и контроля за его состоянием, что является жизненно важным в условиях разобщенности разработчиков и временной протяженности крупного проекта. Это предполагает наличие единой технологической среды создания и сопровождения, которая должна обеспечиваться программно-технологическими интерфейсами между отдельными инструментальными средствами.

! Следует запомнить

1. Разработка больших проектов невозможна без совокупности нормативно-методических документов, регламентирующих различные аспекты процессов деятельности разработчиков.

2. Одним из базовых понятий программной инженерии является понятие жизненного цикла программного обеспечения (ЖЦ ПО). *Жизненный цикл программного обеспечения* определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

3. Под *моделью ЖЦ ПО* понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Наиболее распространенными моделями являются *каскадная* и *итерационная*.

4. *Зрелость процессов ЖЦПО* — это степень их управляемости, контролируемости и эффективности. Повышение технологической зрелости означает потенциальную возможность возрастания устойчивости процессов и указывает на степень эффективности и согласованности использования процессов создания и сопровождения ПО в рамках всей организации.

Основные понятия

Нормативно-методическое обеспечение, жизненный цикл программного обеспечения, процессы жизненного цикла, модель ЖЦ ПО, стадия процесса создания ПО, каскадная модель, итерационная модель, зрелость процессов.

? Вопросы для самоконтроля

1. Что такое жизненный цикл программного обеспечения?
2. Чем регламентируется ЖЦ ПО?
3. Какие группы процессов входят в состав ЖЦ ПО и какие процессы входят в состав каждой группы?
4. Какие из процессов, по вашему мнению, наиболее часто используются в реальных проектах, какие в меньшей степени и почему?
5. Какие стадии входят в процесс создания ПО?
6. Каково соотношение между стадиями и процессами ЖЦ ПО?
7. Каковы принципиальные особенности каскадной модели?
8. В чем заключаются преимущества и недостатки каскадной модели?
9. Каковы принципиальные особенности итерационной модели?
10. В чем состоят преимущества и недостатки итерационной модели?
11. Каким образом можно добиться повышения уровня зрелости процессов создания ПО?
12. Какую роль в повышении уровня зрелости играют процессы управления требованиями и управления конфигурацией ПО?

МЕТОДИЧЕСКИЕ АСПЕКТЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *В чем заключаются общие принципы проектирования систем и что такое визуальное моделирование.*
- *Что представляет собой структурный подход к анализу и проектированию ПО.*
- *В чем заключается метод функционального моделирования SADT.*
- *Как строятся диаграммы потоков данных и диаграммы «сущность — связь».*
- *Что представляет собой объектно-ориентированный подход к анализу и проектированию ПО.*
- *В чем заключаются основные особенности языка моделирования UML.*
- *Как строятся модели и диаграммы, входящие в состав UML.*
- *Что представляют собой образцы.*

2.1.

ОБЩИЕ ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ СИСТЕМ

Как было отмечено ранее, одной из основных проблем, которые приходится решать при создании больших и сложных систем любой природы, в том числе и ПО, является проблема *сложности*. Ни один разработчик не в состоянии выйти за пределы человеческих возможностей и понять всю систему в целом. Единственный эффективный подход к решению этой проблемы, который выра-

ботало человечество за всю свою историю, заключается в построении сложной системы из небольшого количества крупных частей, каждая из которых, в свою очередь, строится из частей меньшего размера, и т.д., до тех пор, пока самые небольшие части можно будет строить из имеющегося материала. Этот подход известен под самыми разными названиями, среди них такие, как «разделяй и властвуй» (*divide et impera*), иерархическая декомпозиция и др. По отношению к проектированию сложной программной системы это означает, что ее необходимо разделить (декомпонировать) на небольшие подсистемы, каждую из которых можно разрабатывать независимо от других. Это позволяет при разработке подсистемы любого уровня иметь дело только с ней, а не со всеми остальными частями системы. Правильная декомпозиция является главным способом преодоления сложности разработки больших систем ПО. Понятие «правильная» по отношению к декомпозиции означает следующее:

- количество связей между отдельными подсистемами должно быть минимальным (принцип «слабой связанности» — Low Coupling);
- связность отдельных частей внутри каждой подсистемы должна быть максимальной (принцип «сильного сцепления» — High Cohesion).

Более подробно эти принципы будут рассмотрены в рамках объектно-ориентированного анализа (подразд. 4.3.2).

Структура системы должна быть такой, чтобы все взаимодействия между ее подсистемами укладывались в ограниченные, стандартные рамки, т.е.:

- каждая подсистема должна **инкапсулировать** свое содержимое (скрывать его от других подсистем);
- каждая подсистема должна иметь четко определенный **интерфейс** с другими подсистемами.

Инкапсуляция (принцип «черного ящика») позволяет рассматривать структуру каждой подсистемы независимо от других подсистем. **Интерфейсы** позволяют строить систему более высокого уровня, рассматривая каждую подсистему как единое целое и игнорируя ее внутреннее устройство.

Существуют два основных подхода к декомпозиции систем. Первый подход называют *функционально-модульным*, он является частью более общего *структурного* подхода. В его основу положен принцип функциональной декомпозиции, при которой структура системы описывается в терминах иерархии ее *функций* и передачи информации между отдельными функциональными элементами. Вторым, *объектно-ориентированный* подход, использует объектную декомпозицию. При этом структура системы описывается в терминах *объектов* и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

В 1970—1980 годах при разработке ПО достаточно широко применялись структурные методы, предоставляющие в распоряжение разработчиков строгие формализованные методы описания проектных решений — спецификаций ПО (в настоящее время такое же распространение получают объектно-ориентированные методы). Эти методы основаны на использовании наглядных графических моделей: для описания архитектуры ПО с различных точек зрения (как статической структуры, так и динамики поведения системы) используются схемы и диаграммы. Наглядность и строгость средств структурного и объектно-ориентированного анализа позволяет разработчикам и будущим пользователям системы с самого начала неформально участвовать в ее создании, обсуждать и закреплять понимание основных технических решений. Однако широкое применение этих методов и следование их рекомендациям при разработке конкретных систем ПО сдерживалось отсутствием адекватных инструментальных средств, поскольку при неавтоматизированной (ручной) разработке все их преимущества практически сведены к нулю. Действительно, вручную очень трудно разработать и графически представить строгие формальные спецификации системы, проверить их на полноту и непротиворечивость и тем более изменить. Если все же удастся создать строгую систему проектных документов, то ее переработка при появлении серьезных изменений практически неосуществима. Ручная разработка обычно порождала следующие проблемы:

- неадекватная спецификация требований;

- неспособность обнаруживать ошибки в проектных решениях;
- низкое качество документации, снижающее эксплуатационные характеристики;
- затяжной цикл и неудовлетворительные результаты тестирования.

С другой стороны, разработчики ПО исторически всегда стояли последними в ряду тех, кто использовал компьютерные технологии для повышения качества, надежности и производительности в своей собственной работе (феномен «сапожника без сапог»).

Перечисленные факторы способствовали появлению программно-технологических средств специального класса - CASE-средств, реализующих CASE-технологии создания ПО. Понятие CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение этого понятия, ограниченное только задачами автоматизации разработки ПО, в настоящее время приобрело новый смысл, охватывающий большинство процессов жизненного цикла ПО.

Появлению CASE-технологии и CASE-средств предшествовали исследования в области программирования: разработка и внедрение языков высокого уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т.д. Кроме того, этому способствовали следующие факторы:

- подготовка аналитиков и программистов, восприимчивых к концепциям модульного и структурного программирования;
- широкое внедрение и постоянный рост производительности компьютеров, позволившие использовать эффективные графические средства и автоматизировать большинство этапов проектирования;
- внедрение сетевой технологии, предоставившей возможность объединения усилий отдельных исполнителей в единый процесс проектирования путем использования разделяемой базы данных, содержащей необходимую информацию о проекте.

CASE-технология представляет собой совокупность методов проектирования ПО, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех стадиях разработки и сопровождения ПО и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методах структурного или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

2.2.

ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ

Под *моделью ПО* в общем случае понимается формализованное описание системы ПО на определенном уровне абстракции. Каждая модель определяет конкретный аспект системы, использует набор диаграмм и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами.

Под термином «*моделирование*» понимается процесс создания формализованного описания системы в виде совокупности моделей. Особенно трудным оказывается описание систем средней сложности, таких, как система коммутаций в телефонных сетях, управление авиаперевозками или движением подводной лодки, сборка автомобилей, челночные космические рейсы, функционирование перерабатывающих предприятий (к таким системам относятся и ПО). С точки зрения человека, эти системы описать достаточно трудно, потому что они настолько велики, что практически невозможно перечислить все их компоненты со своими взаимосвязями, и в то же время недостаточно велики для применения общих упрощающих предположений (как это принято в физике). Неспособно

способность дать простое описание, а, следовательно, и обеспечить понимание таких систем делает их проектирование и создание трудоемким и дорогостоящим процессом и повышает степень их ненадежности. С ростом технического прогресса адекватное описание систем становится все более актуальной проблемой.

Модель должна давать полное, точное и адекватное описание системы, имеющее конкретное назначение. Это назначение, называемое *целью модели*, вытекает из формального определения модели:

М есть модель системы S, если M может быть использована для получения ответов на вопросы относительно S с точностью A.

Таким образом, целью модели является получение ответов на некоторую совокупность вопросов. Эти вопросы неявно присутствуют (подразумеваются) в процессе анализа и, следовательно, они руководят созданием модели и направляют его. Это означает, что сама модель должна будет дать ответы на эти вопросы с заданной степенью точности. Если модель отвечает не на все вопросы или ее ответы недостаточно точны, то говорят, что модель не достигла своей цели.

По мнению авторитетных специалистов в области проектирования ПО, моделирование является центральным звеном всей деятельности по созданию систем ПО. Модели строятся для того, чтобы понять и осмыслить структуру и поведение будущей системы, облегчить управление процессом ее создания и уменьшить возможный риск, а также документировать принимаемые проектные решения.

Визуальное моделирование — это способ восприятия проблем с помощью зримых абстракций, воспроизводящих понятия и объекты реального мира. Модели служат полезным инструментом анализа проблем, обмена информацией между всеми заинтересованными сторонами (пользователями, специалистами в предметной области, аналитиками, проектировщиками и т.д.), проектирования ПО, а также подготовки документации. Моделирование способствует более

полному усвоению требований, улучшению качества системы и повышению степени ее управляемости.

С помощью модели можно упростить проблему, отбросив несущественные детали и сосредоточив внимание на главном. Способность к абстрагированию — это фундаментальное свойство человеческого интеллекта, помогающее справиться с феноменом сложности. На протяжении тысяч лет художники, ремесленники и строители предпринимали попытки конструирования тех или иных моделей, предваряющие реальное воплощение творческих замыслов. Не составляет исключения и индустрия разработки ПО. Чтобы создать сложную программную систему, разработчики должны абстрагировать ее свойства с разных точек зрения, с помощью точных нотаций (систем обозначений) сконструировать адекватные модели, удостовериться, удовлетворяют ли они исходным требованиям, а затем реализовать модели на практике, постепенно пополняя систему новыми функциями.

К моделированию сложных систем приходится прибегать ввиду того, что мы не в состоянии постичь их одновременно во всей полноте. Способность человека к восприятию сложных вещей имеет свои естественные границы. В этом легко убедиться, обратившись к сфере строительства и архитектуры. Если нужно построить курятник, за работу можно приниматься тотчас; если речь идет о новом доме, тогда, вероятно, потребуется хотя бы эскиз; наконец, при возведении небоскреба без точных расчетов и чертежей обойтись уже явно не удастся. Те же законы действуют и в мире программирования. Делая главную ставку на написание строк исходного кода или даже на использование форм Visual Basic, разработчик вряд ли сможет сколько-нибудь полно охватить структуру объемного проекта в целом. Предварительное моделирование, в свою очередь, позволяет проектировщику увидеть общую картину взаимодействий компонентов проекта без необходимости анализа особых свойств каждого компонента.

Графические (визуальные) модели представляют собой средства для визуализации, описания, проектирования и документирования архитектуры систе-

мы. Под *архитектурой* понимается набор основных правил, определяющих организацию системы:

- совокупность структурных элементов системы и связей между ними;
- поведение элементов системы в процессе их взаимодействия;
- иерархию подсистем, объединяющих структурные элементы;
- архитектурный стиль (используемые методы и средства описания архитектуры, а также архитектурные образцы).

Архитектура является многомерной, поскольку различные специалисты работают с различными ее аспектами. Например, при строительстве здания используются разные типы чертежей, представляющие различные аспекты архитектуры:

- планы этажей;
- вертикальные проекции;
- планы монтажа кабельной проводки;
- чертежи водопроводов, системы центрального отопления и вентиляции;
- вид здания на фоне местности (эскизы).

Архитектура ПО также предусматривает различные представления, служащие разным целям:

- представлению функциональных возможностей системы;
- отображению логической организации системы;
- описанию физической структуры программных компонентов в среде реализации;
- отображению структуры потоков управления и аспектов параллельной работы;
- описанию физического размещения программных компонентов на базовой платформе.

Архитектурное представление — это упрощенное описание (абстракция) системы с конкретной точки зрения, охватывающее определенный круг интересов и опускающее объекты, несущественные с данной точки зрения. Архитектурные представления концентрируют внимание только на элементах, значи-

мых с точки зрения архитектуры. *Архитектурно значимый элемент* - это элемент, имеющий значительное влияние на структуру системы и ее производительность, надежность и возможность развития. Это элемент, важный для понимания системы. Например, в состав архитектурно значимых элементов объектно-ориентированной архитектуры входят основные классы предметной области, подсистемы и их интерфейсы, основные процессы или потоки управления.

Архитектурные представления подобны сечениям различных моделей, выделяющим только важные, значимые элементы моделей.

Разработка модели архитектуры системы ПО промышленного характера на стадии, предшествующей ее реализации или обновлению, в такой же мере необходима, как и наличие проекта для строительства большого здания. Это утверждение справедливо как в случае разработки новой системы, так и при адаптации типовых продуктов класса R/3 или BAAN, в составе которых также имеются собственные средства моделирования. Хорошие модели являются основой взаимодействия участников проекта и гарантируют корректность архитектуры. Поскольку сложность систем повышается, важно располагать хорошими методами моделирования. Хотя имеется много других факторов, от которых зависит успех проекта, но наличие строгого стандарта языка моделирования является весьма существенным.

Язык моделирования включает:

- элементы модели — фундаментальные концепции моделирования и их семантику;
- нотацию (систему обозначений) — визуальное представление элементов моделирования;
- руководство по использованию — правила применения элементов в рамках построения тех или иных типов моделей ПО.

Очевидно, что конечная цель разработки ПО — это не моделирование, а получение работающих приложений (кода). Диаграммы в конечном счете — это всего лишь наглядные изображения. Создание слишком большого количе-

ства диаграмм до начала программирования займет слишком много времени и не обеспечит построения готовой системы. Поэтому при использовании графических языков моделирования очень важно понимать, чем это поможет, когда дело дойдет до написания кода. Можно привести следующие причины, побуждающие прибегать к их использованию:

- **получение общего представления о системе.** Графические модели помогают быстро получить общее представление о системе, сказать о том, какого рода абстракции существуют в системе и какие ее части нуждаются в дальнейшем уточнении;
- **общение с экспертами организации.** Графические модели образуют внешнее представление системы и объясняют, что эта система будет делать;
- **изучение методов проектирования.** Множество людей отмечает наличие серьезных трудностей, связанных, например, с освоением объектно-ориентированных методов, и, в первую очередь, смену парадигмы. В некоторых случаях переход к объектно-ориентированным методам происходит относительно безболезненно. В других случаях при работе с объектами приходится сталкиваться с рядом препятствий, особенно в части максимального использования их потенциальных возможностей. Графические средства позволяют облегчить решение этой проблемы.

В процессе создания ПО, автоматизирующего деятельность некоторой организации, используются следующие виды моделей:

- модели деятельности организации (или модели бизнес-процессов):

модели «AS-IS» («как есть»), отражающие существующее на момент обследования положение дел в организации и позволяющие понять, каким образом функционирует данная организация, а также выявить узкие места и сформулировать предложения по улучшению ситуации;

модели «AS-TO-BE» («как должно быть»), отражающие представление о новых процессах и технологиях работы организации. Переход от модели «AS-IS» к модели «AS-TO-BE» может выполняться двумя способами: 1) совершен-

ствованием существующих технологий на основе оценки их эффективности; 2) радикальным изменением технологий и перепроектированием (реинжинирингом) бизнес-процессов.

- модели проектируемого ПО, которые строятся на основе модели «AS-TO-VE», уточняются и детализируются до необходимого уровня.

Состав моделей, используемых в каждом конкретном проекте, и степень их детальности в общем случае (как для структурного, так и для объектно-ориентированного подхода) зависят от следующих факторов:

- сложности проектируемой системы;
- необходимой полноты ее описания;
- знаний и навыков участников проекта;
- времени, отведенного на проектирование.

2.3.

СТРУКТУРНЫЕ МЕТОДЫ АНАЛИЗА И ПРОЕКТИРОВАНИЯ ПО

Структурные методы являются строгой дисциплиной системного анализа и проектирования. Методы структурного анализа и проектирования¹ стремятся преодолеть сложность больших систем путем расчленения их на части («черные ящики») и иерархической организации этих «черных ящиков». Выгода в использовании «черных ящиков» заключается в том, что их пользователю не требуется знать, как они работают, необходимо знать лишь их входы и выходы, а также назначение (т.е. функции, которые они выполняет).

¹Каляное Т.Н. Консалтинг при автоматизации предприятий Синтег. — М., 1997. (Серия «Информатизация России на пороге XXI века»).

Таким образом, первым шагом упрощения сложной системы является ее разбиение на «черные ящики», при этом такое разбиение должно удовлетворять следующим критериям:

- каждый «черный ящик» должен реализовывать единственную функцию системы;
- функция каждого «черного ящика» должна быть легко понимаема независимо от сложности ее реализации;
- связь между «черными ящиками» должна вводиться только при наличии связи между соответствующими функциями системы (например, в бухгалтерии один «черный ящик» не обходим для расчета общей заработной платы служащего, а другой для расчета налогов — необходима связь между этими «черными ящиками»: размер заработной платы требуется для расчета налогов);
- связи между «черными ящиками» должны быть простыми, насколько это возможно, для обеспечения независимости между ними.

Второй важной идеей, лежащей в основе структурных методов, является идея иерархии. Для понимания сложной системы недостаточно разбиения ее на части, необходимо эти части организовать определенным образом, а именно в виде иерархических структур.

Все сложные системы Вселенной организованы в иерархии: от галактик до элементарных частиц. Человек при создании сложных систем также подражает природе. Любая организация имеет директора, заместителей по направлениям, иерархию руководителей подразделений, рядовых служащих.

Кроме того, структурные методы широко используют визуальное моделирование, служащее для облегчения понимания сложных систем.

Структурным анализом принято называть метод исследования системы, начинающий с ее общего обзора, который затем детализируется, приобретая иерархическую структуру со все большим числом уровней. Для таких методов характерно:

- разбиение системы на уровни абстракции с ограничением числа элементов на каждом из уровней (обычно от 3 до 6—7);
- ограниченный контекст, включающий лишь существенные на каждом уровне детали;
- использование строгих формальных правил записи;
- последовательное приближение к конечному результату.

В структурном анализе основным методом разбиения на уровни абстракции является **функциональная декомпозиция**, заключающаяся в декомпозиции (разбиении) системы на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, те — на задачи и так далее до конкретных процедур. При этом система сохраняет целостное представление, в котором все составляющие компоненты взаимосвязаны. При разработке системы «снизу вверх» от отдельных задач ко всей системе целостность теряется, возникают проблемы при описании информационного взаимодействия отдельных компонентов.

Все наиболее распространенные методы структурного подхода базируются на ряде общих принципов. Базовыми принципами являются:

- **принцип «разделяй и властвуй»** — принцип решения трудных проблем путем разбиения их на множество меньших независимых задач, легких для понимания и решения;
- **принцип иерархического упорядочения** — принцип организации составных частей системы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к нежелательным последствиям (вплоть до неудачного завершения проекта). Основными из этих принципов являются:

- **принцип абстрагирования** — выделение существенных аспектов системы и отвлечение от несущественных;

- **принцип непротиворечивости** — обоснованность и согласованность элементов системы;
- **принцип структурирования данных** — данные должны быть структурированы и иерархически организованы.

В структурном анализе и проектировании используются различные модели, описывающие:

- 1) функциональную структуру системы;
- 2) последовательность выполняемых действий;
- 3) передачу информации между функциональными процессами;
- 4) отношения между данными.

Наиболее распространенными моделями первых трех групп являются:

- функциональная модель SADT (Structured Analysis and Design Technique);
- модель IDEF3;
- DFD (Data Flow Diagrams) — диаграммы потоков данных. Модель «сущность — связь» (ERM — Entity-Relationship Model), описывающая отношения между данными, традиционно используется в структурном анализе и проектировании, однако, по существу, представляет собой подмножество объектной модели предметной области.

2.3.1

МЕТОД ФУНКЦИОНАЛЬНОГО МОДЕЛИРОВАНИЯ

SADT (IDEFO)

Общие сведения

Метод SADT¹ разработан Дугласом Россом (SoftTech, Inc.) в 1969 г. для моделирования искусственных систем средней сложности. Данный метод успешно использовался в военных, промышленных и коммерческих организациях

¹ Марка Д.А., МакГоуэн К. Методология структурного анализа и проектирования. — М.: Ме-

США для решения широкого круга задач, таких, как долгосрочное и стратегическое планирование, автоматизированное производство и проектирование, разработка ПО для оборонных систем, управление финансами и материально-техническим снабжением и др. Метод SADT поддерживается Министерством обороны США, которое было инициатором разработки семейства стандартов IDEF (Icam DEFinition), являющегося основной частью программы ICAM (интегрированная компьютеризация производства), проводимой по инициативе ВВС США.

Метод SADT реализован в одном из стандартов этого семейства — IDEF0, который был утвержден в качестве федерального стандарта США в 1993 г., его подробные спецификации можно найти на сайте <http://www.idef.com>.

Метод SADT представляет собой совокупность правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. **Функциональная модель** SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этого метода основываются на следующих концепциях:

- графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа/выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описывается посредством интерфейсных дуг, выражающих «ограничения», которые, в свою очередь, определяют, когда и каким образом функции выполняются и управляются;
- строгость и точность. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают: ограничение количества блоков на каждом уровне декомпозиции (правило 3—6 блоков — ограничение мощности краткосрочной памяти человека), связность

диаграмм (номера блоков), уникальность меток и наименований (отсутствие повторяющихся имен), синтаксические правила для графики (блоков и дуг), разделение входов и управлений (правило определения роли данных);

- отделение организации от функции, т.е. исключение влияния административной структуры организации на функциональную модель.

Метод SADT может использоваться для моделирования самых разнообразных процессов и систем. В существующих системах метод SADT может быть использован для анализа функций, выполняемых системой, и указания механизмов, посредством которых они осуществляются.

Состав функциональной модели

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы — главные компоненты модели, все функции организации и интерфейсы на них представлены как блоки и дуги соответственно. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как входная информация, которая подвергается обработке, показана с левой стороны блока, а результаты (выход) показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рис. 2.1).

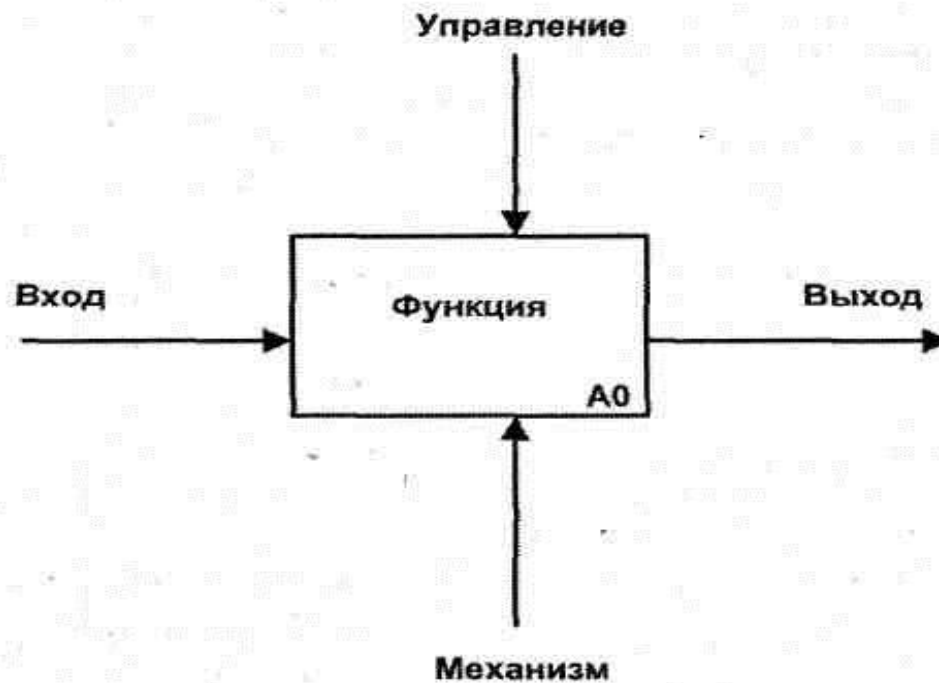


Рис. 2.1. Функциональный блок и интерфейсные дуги

Одной из наиболее важных особенностей метода SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

На рис. 2.2, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует «внутреннее строение» блока на родительской диаграмме.

Построение SADT-модели

Построение SADT-модели заключается в выполнении следующих действий:

- сбор информации об объекте, определение его границ;
- определение цели и точки зрения модели;
- построение, обобщение и декомпозиция диаграмм;
- критическая оценка, рецензирование и комментирование.

Построение диаграмм начинается с представления всей системы в виде простейшего компонента - одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок отражает систему как

единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг — они также соответствуют полному набору внешних интерфейсов системы в целом.

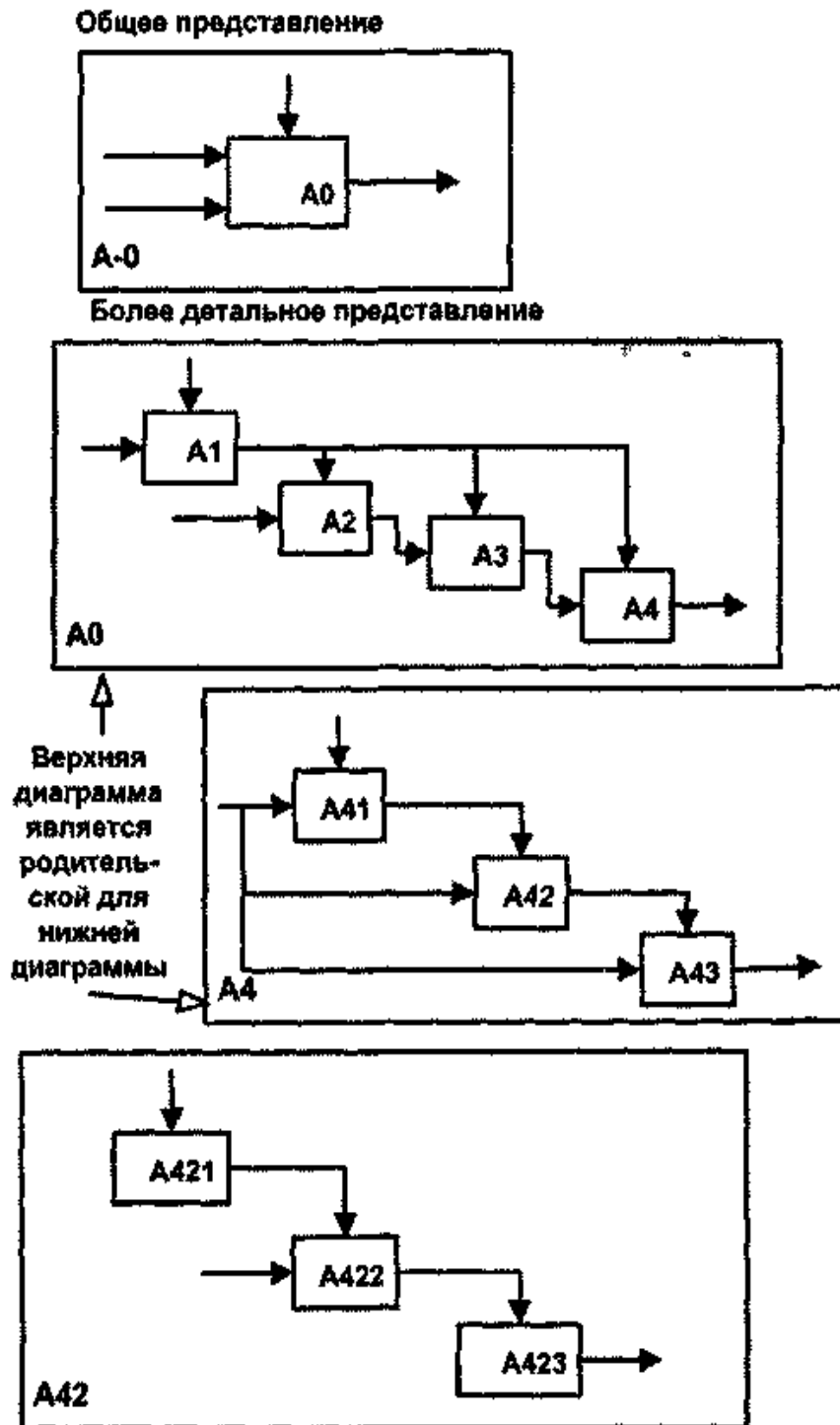


Рис. 2.2. Структура SADT-модели. Декомпозиция диаграмм

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединен-

ных интерфейсными дугами. Эти блоки определяют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых показана как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом в целях большей детализации.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые изображены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из диаграммы предыдущего уровня. На каждом шаге декомпозиции диаграмма предыдущего уровня называется родительской для более детальной диаграммы.

Синтаксис диаграмм определяется следующими правилами:

- диаграммы содержат блоки и дуги;
- блоки представляют функции;
- блоки имеют доминирование (выражающееся в их ступенчатом расположении, причем доминирующий блок располагается в верхнем левом углу диаграммы);
- дуги изображают наборы объектов, передаваемых между блоками;
- дуги изображают взаимосвязи между блоками:

выход-управление;

выход-вход;

обратная связь по управлению;

обратная связь по входу;

выход-механизм.

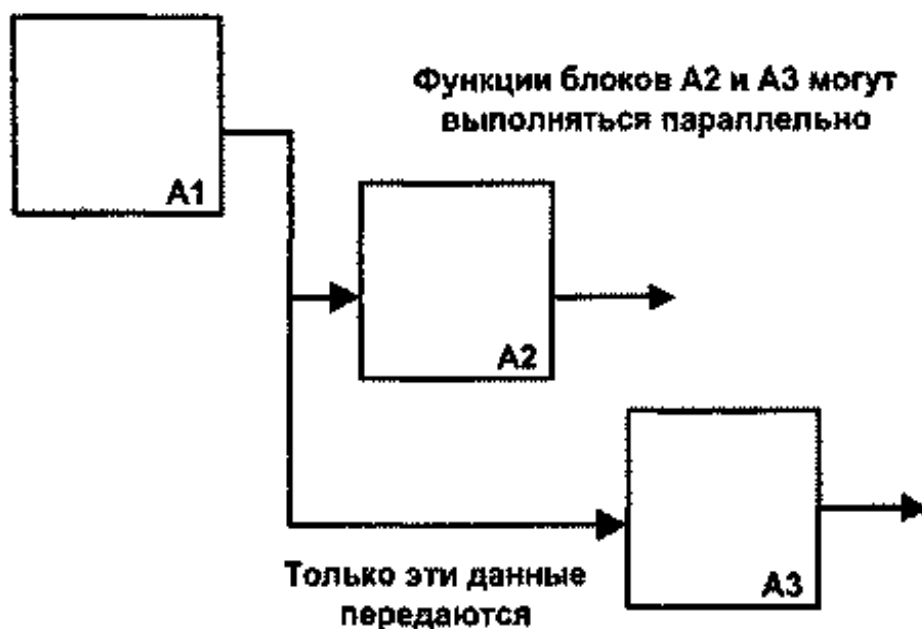


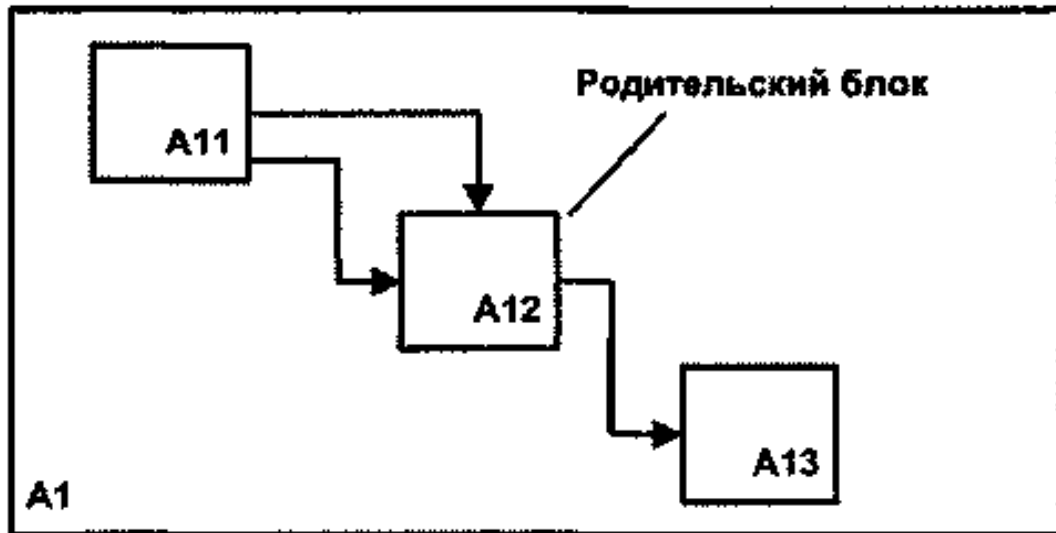
Рис. 2.3. Одновременное выполнение функций

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма изображают одну и ту же часть системы.

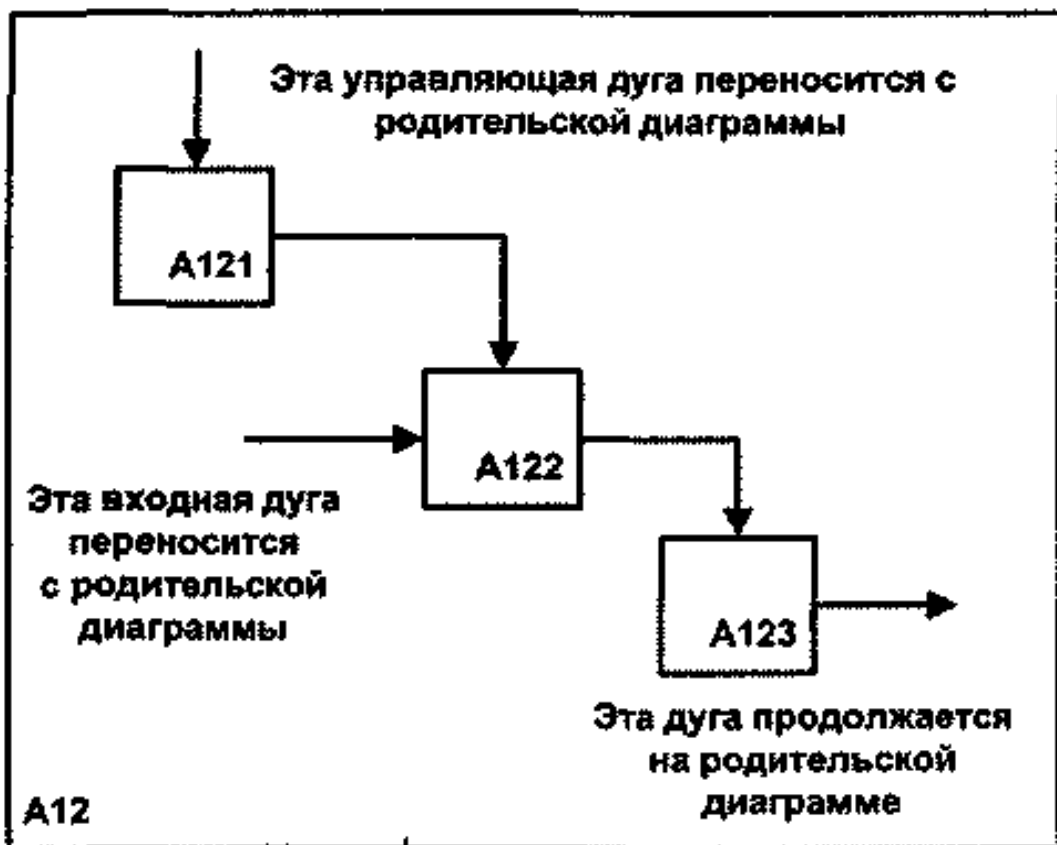
На последующих рис. 2.3—2.5 приведены различные варианты выполнения функций и соединения дуг с блоками.

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным. Неприсоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рис. 2.5).



а



б

Рис. 2.4. Соответствие интерфейсных дуг родительской (а) и детальной (б) диаграмм

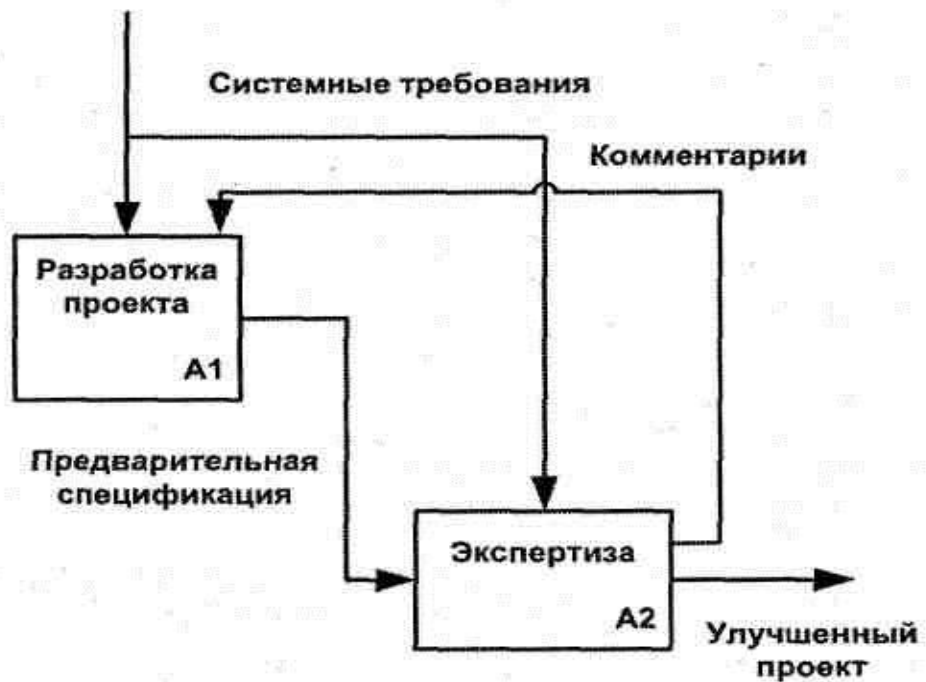


Рис. 2.5. Пример обратной связи

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рис. 2.6).

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть далее описан диаграммой нижнего уровня, которая, в свою очередь, может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом, формируется иерархия диаграмм.

Для того чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, A21 является диаграммой, которая детализирует блок A21 на диаграмме A2. Аналогично, диаграмма A2 детализирует блок A2 на диаграмме A0, которая является самой верхней диаграммой модели. На рис. 2.7 показан пример дерева диаграмм.

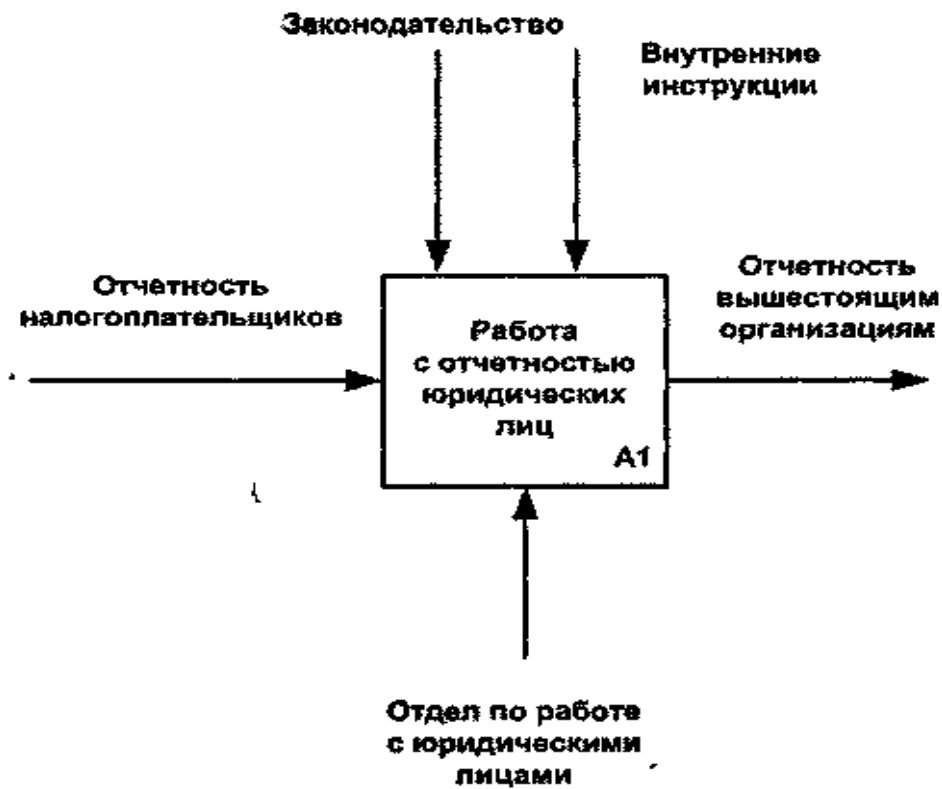


Рис. 2.6. Пример механизма

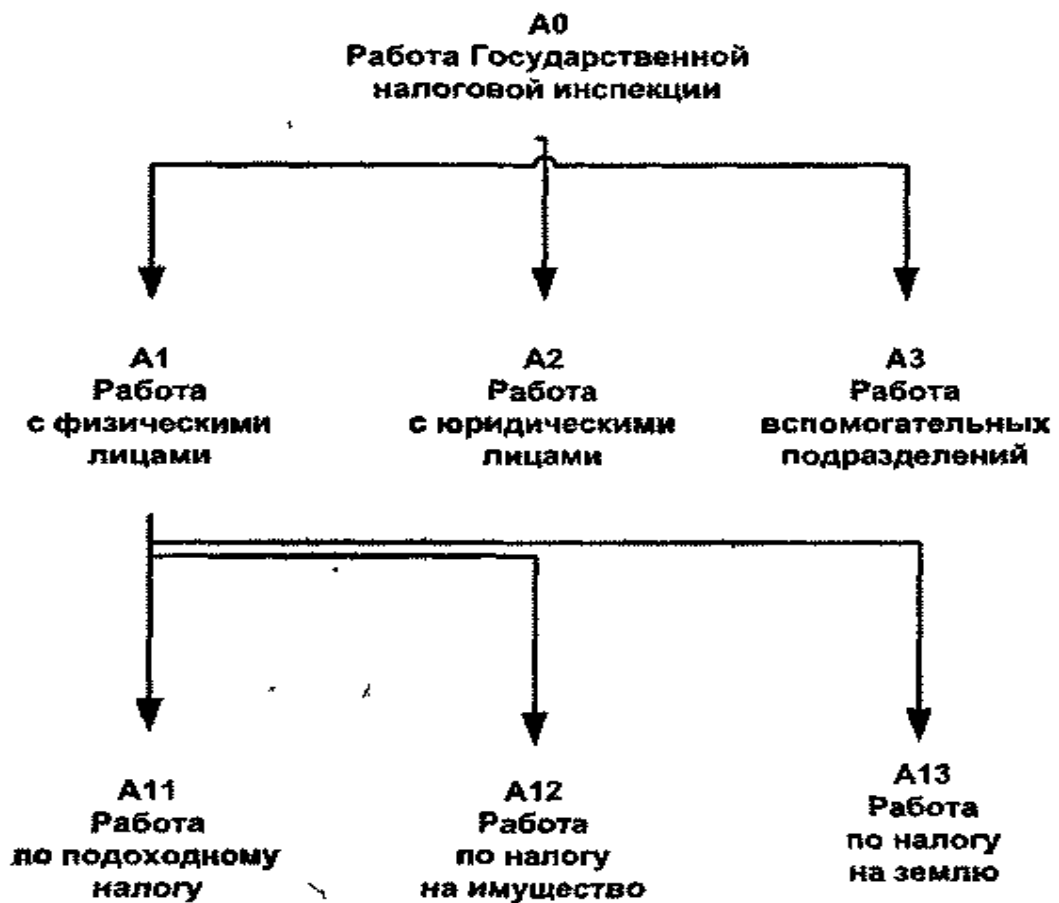


Рис. 2.7. Иерархия диаграмм

Стратегии декомпозиции

При построении иерархии диаграмм используются следующие стратегии декомпозиции:

- **Функциональная декомпозиция** — декомпозиция в соответствии с функциями, которые выполняют люди или организация. Может оказаться полезной стратегией для создания системы описаний, фиксирующей взаимодействие между людьми в процессе их работы. Очень часто, однако, взаимосвязи между функциями весьма многочисленны и сложны, поэтому рекомендуется использовать эту стратегию только в начале работы над моделью системы.
- **Декомпозиция в соответствии с известными стабильными подсистемами** — приводит к созданию набора моделей, по одной модели на каждую подсистему или важный компонент. Затем для описания всей системы должна быть построена составная модель, объединяющая все отдельные модели. Рекомендуется использовать разложение на подсистемы, только когда разделение на основные части системы не меняется. Нестабильность границ подсистем быстро обесценит как отдельные модели, так и их объединение.
- **Декомпозиция по физическому процессу** — выделение функциональных стадий, этапов завершения или шагов выполнения. Хотя эта стратегия полезна при описании существующих процессов (таких, например, как работа промышленного предприятия), результатом ее часто может стать слишком последовательное описание системы, которое не будет в полной мере учитывать ограничения, диктуемые функциями друг другу. При этом может оказаться скрытой последовательность управления. Эта стратегия рекомендуется, только если целью модели является описание физического процесса как такового или только в крайнем случае, когда неясно, как действовать.

Завершение моделирования

(определение момента прекращения декомпозиции)

Одна из наиболее частых проблем, возникающих в процессе построения SADT-моделей, — когда же следует завершить построение конкретной модели? На этот вопрос не всегда легко ответить, хотя существуют некоторые эвристики для определения разумной степени полноты. Здесь представлены правила, которыми пользуются опытные аналитики для определения момента завершения моделирования. Они носят характер рекомендаций. Только длительная практика позволит приобрести знания, необходимые для принятия правильного решения об окончании моделирования.

Прежде чем обсуждать критерии для определения завершения процесса моделирования, рассмотрим, как увеличивается размер модели. С точки зрения математики размер иерархических моделей увеличивается со скоростью геометрической прогрессии. В табл. 2.1 показаны размеры полной четырехуровневой SADT-модели, каждая диаграмма которой состоит из четырех блоков, причем каждый из этих блоков декомпозируется аналогичной диаграммой.

Таблица 2.1

Размеры четырехуровневой SADT-модели

Уровень модели	Общее число блоков в модели	
	4 блока/1 диаграмма	6 блоков/1 диаграмма
Тор	1	1
0	5	7
1	21	43
2	85	259
3	341	1555
4	1365	9331

В такой модели общее число блоков составляет 1365, а в четырехуровневой модели, содержащей по шесть блоков на диаграмме, общее число их 9331.

Хотя с математической точки зрения все верно, SADT-модели такого размера никогда не создаются по целому ряду причин. Ни одна SADT-модель не будет иметь одинаковую глубину. Обычно модель строится слоями, большинство из которых не являются глубокими. Чаще всего ограничиваются тремя

уровнями. Опыт показывает, что, как правило, создаются несколько диаграмм второго и третьего уровней только для того, чтобы убедиться, что для достижения цели уже первый уровень содержит достаточно информации.

Если SADT-модель декомпозируется на глубину 5-6 уровней, то в этом случае на такую глубину декомпозируется обычно один из блоков диаграммы АО. Функции, которые требуют такого уровня детализации, часто очень важны, и их детальное описание дает ключ к секретам работы всей системы. Но хотя важные функции могут нуждаться в глубокой детализации, таких функций при создании одной модели насчитывается, как правило, немного. Модели, обладающие такими функциями, имеют обычно форму зонтика с широким тонким куполом и длинной ручкой, на которой происходит детализация. Поэтому вторая причина, по которой размер SADT-моделей не растет в геометрической прогрессии, заключается в том, что, хотя нередко модель имеет глубину 5—6 уровней, она почти никогда не декомпозируется вся до такой степени детализации.

Большие аналитические проекты обычно разбиваются на несколько отдельных более мелких проектов, каждый из которых создает модель одного конкретного аспекта всей проблемы. Поэтому вместо одной гигантской модели создается сеть из нескольких небольших моделей. Исключительно большие проекты могут привести к созданию высококачественной модели, состоящей из тысяч блоков, но это случается редко. Большинство систем не требует для адекватного описания моделей такой величины.

Рекомендуется прекращать моделирование, когда уровень детализации модели удовлетворяет ее цели. Опыт показал, что для отдельной модели, которая создается независимо от какой-либо другой модели, декомпозиция одного из ее блоков должна прекращаться, если:

- **блок содержит достаточно деталей.** Одна из типичных ситуаций, встречающихся в конце моделирования, — это блок, который описывает систему с нужным уровнем подробности. Проверить достаточность деталей обычно совсем легко, достаточно просто спросить себя, отвечает ли

блок на все или на часть вопросов, составляющих цель модели. Если блок помогает ответить на один или более вопросов, то дальнейшая декомпозиция может не понадобиться;

- ***необходимо изменить уровень абстракции, чтобы достичь большей детализации блока.*** Блоки подвергаются декомпозиции, если они недостаточно детализированы для удовлетворения цели модели. Но иногда при декомпозиции блока выясняется, что диаграмма начинает описывать, как функционирует блок, вместо описания того, что блок делает. В этом случае происходит изменение уровня абстракции — изменение сути того, что должна представлять модель (т.е. изменение способа описания системы). В SADT изменение уровня абстракции часто означает выход за пределы цели модели и, следовательно, это указывает на прекращение декомпозиции;
- ***необходимо изменить точку зрения, чтобы детализировать блок.*** Изменение точки зрения происходит примерно так же, как изменение уровня абстракции. Это чаще всего характерно для ситуаций, когда точку зрения модели нельзя использовать для декомпозиции конкретного блока, т. е. этот блок можно декомпонировать, только если посмотреть на него с другой позиции. Об этом может свидетельствовать заметное изменение терминологии;
- ***блок очень похож на другой блок той же модели или на блок другой модели.*** Иногда встречается блок, чрезвычайно похожий на другой блок модели. Два блока похожи, если они выполняют примерно одну и ту же функцию и имеют почти одинаковые по типу и количеству входы, управления и выходы. Если второй блок уже декомпонирован, то разумно отложить декомпозицию и тщательно сравнить два блока. Если нужны ничтожные изменения для совпадения первого блока со вторым, то внесение этих изменений сократит усилия на декомпозицию и улучшит модульность модели (т.е. сходные функции уточняются согласованным образом);

- **блок представляет тривиальную функцию.** Тривиальная функция — это такая функция, понимание которой не требует никаких объяснений. В этом случае очевидна целесообразность отказа от декомпозиции, потому что роль SADT заключается в превращении сложного вопроса в понятный, а не в педантичной разработке очевидных деталей. В таких случаях декомпозиция определенных блоков может принести больше вреда, чем пользы. Тривиальные функции лучше всего описываются небольшим объемом текста. Следует заметить, что «тривиальный» не означает «бесполезный». Тривиальные функции выполняют очень важную роль, поясняя работу более сложных функций, а иногда и соединяя вместе основные подсистемы. Поэтому при анализе не следует пропускать тривиальные функции. Наоборот, их существование должно быть зафиксировано и они должны быть детализованы, как и любые другие функции. Однако следует предостеречь от больших затрат времени на анализ тривиальных функций системы. Усиленное внимание к мелочам может привести к созданию модели, которой будет не хватать абстракции, что сделает ее трудной для понимания и использования.

Типы связей между функциями

Одним из важных моментов при моделировании с помощью метода SADT является точная согласованность типов связей между функциями. Различают по крайней мере связи семи типов (в порядке возрастания их относительной значимости):

- случайная;
- логическая;
- временная;
- процедурная;
- коммуникационная;
- последовательная;
- функциональная.

Случайная связь показывает, что конкретная связь между функциями незначительна или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют слабую связь друг с другом. Крайний вариант этого случая показан на рис. 2.8.

Логическая связь — данные и функции собираются вместе благодаря тому, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

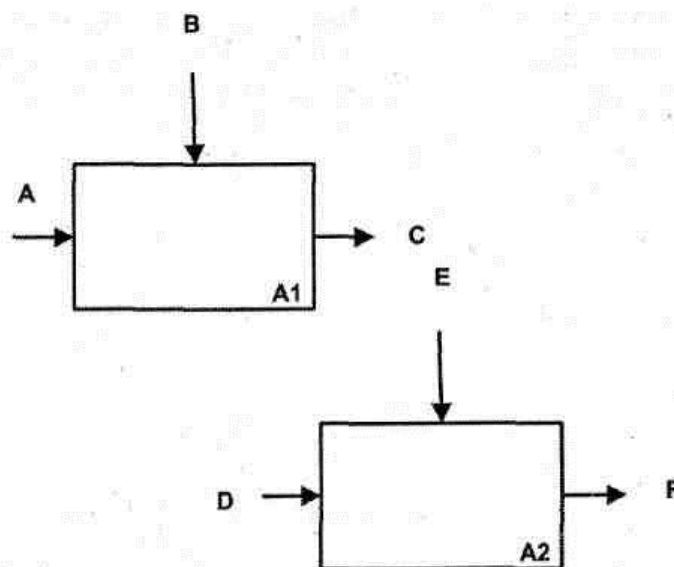


Рис. 2.8. Случайная связь

Временная связь — представляет функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

Процедурная связь (рис. 2.9) — функции сгруппированы вместе благодаря тому, что они выполняются в течение одной и той же части цикла или процесса.

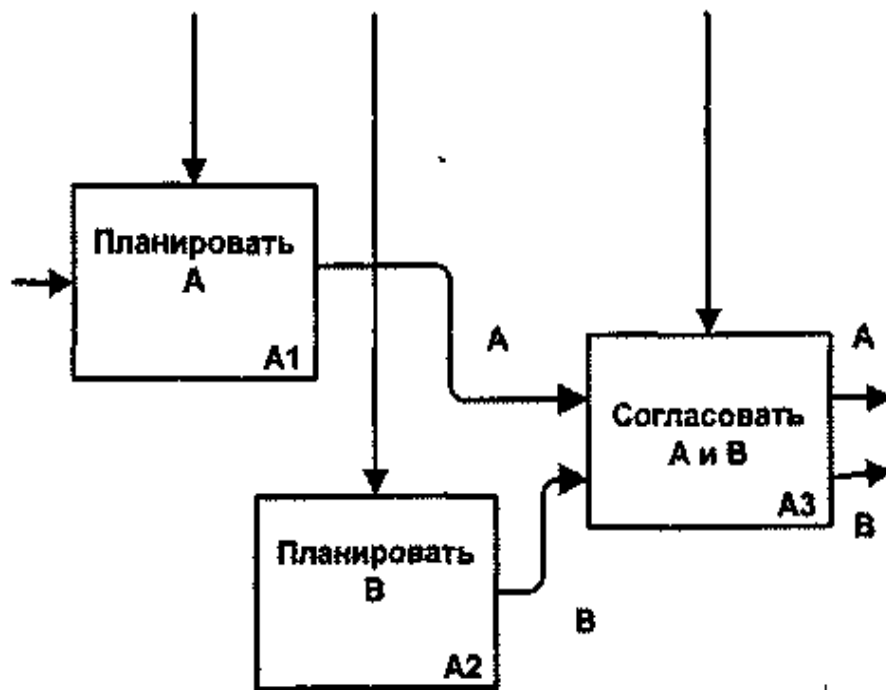


Рис. 2.9. Процедурная связь

Коммуникационная связь — функции группируются благодаря тому, что они используют одни и те же входные данные и/или производят одни и те же выходные данные (рис. 2.10).

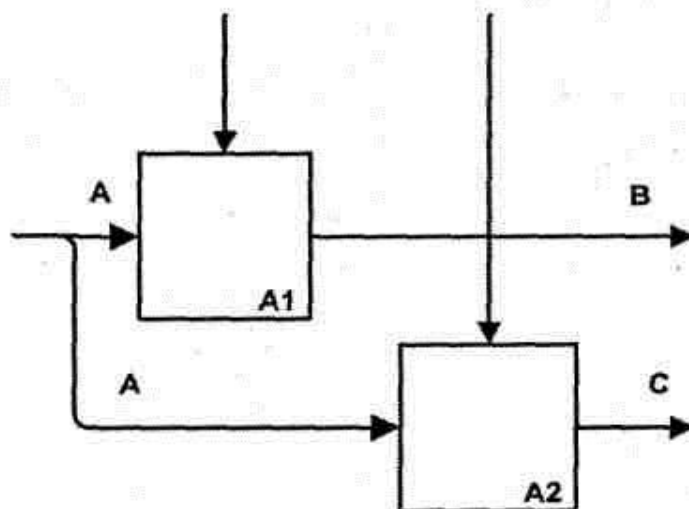


Рис. 2.10. Коммуникационная связь

Последовательная связь — выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является

ся более тесной, чем в рассмотренных выше случаях, поскольку моделируются причинно-следственные зависимости (рис. 2.11).

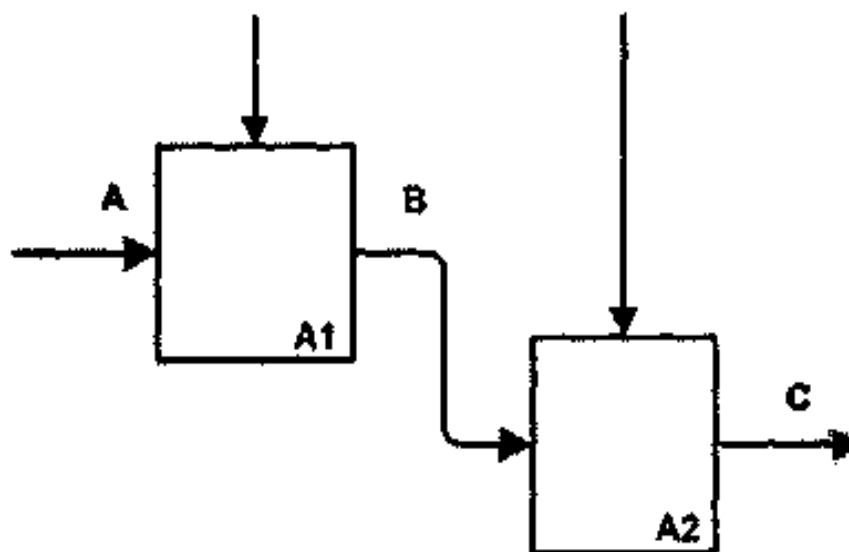


Рис. 2.11. Последовательная связь

Функциональная связь — все элементы функции влияют на выполнение одной и только одной функции. Диаграмма, являющаяся чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связи.

Одним из способов определения функционально-связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги, как показано на рис. 2.12.

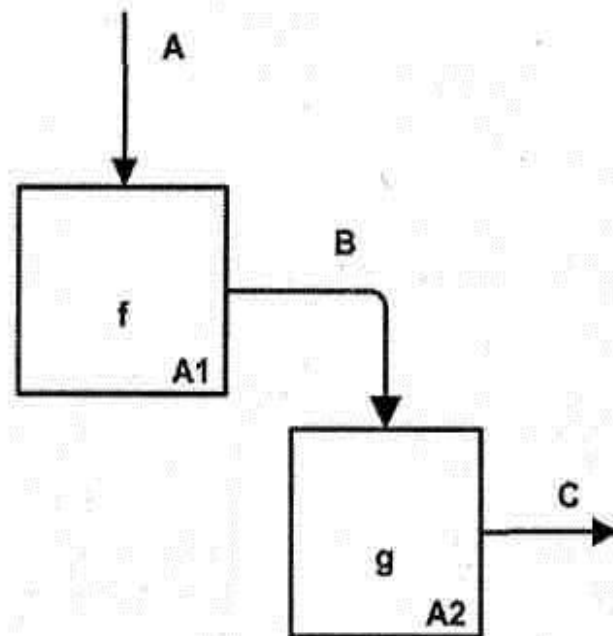


Рис. 2.12. Функциональная связь

В математических терминах необходимое условие для простейшего типа функциональной связи имеет следующий вид:

$$C = g(B)=g(f(A)).$$

В табл. 2.2 представлены все типы связей, рассмотренные выше.

Важно отметить, что уровни 4—6 устанавливают связи, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

Описание типов связей

Уровень значимости	Тип связи	Характеристика типа связи	
		для функций	для данных
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, «редактировать все входы»)	Данные одного и того же множества или типа
2	Временная	Функции одного и того же периода времени (например, «операции инициализации»)	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итерации (например, «первый проход компилятора»)	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

2.3.2.

МОДЕЛИРОВАНИЯ

ПРОЦЕССОВ IDEF3

Метод моделирования IDEF3¹, являющийся частью семейства стандартов IDEF, был разработан в конце 1980-х годов для закрытого проекта ВВС США.

¹ Черемных С.В., Семенов И.О., Ручкин В.С. Структурный анализ систем: IDEF-технологии. — М.: Финансы и статистика, 2001.

Этот метод предназначен для таких моделей процессов, в которых важно понять последовательность выполнения действий и взаимозависимости между ними. Хотя IDEF3 и не достиг статуса федерального стандарта США, он приобрел широкое распространение среди системных аналитиков как дополнение к методу функционального моделирования IDEF0 (модели IDEF3 могут использоваться для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции). Основой модели IDEF3 служит *сценарий* процесса, который выделяет последовательность действий и подпроцессов анализируемой системы.

Как и в методе IDEF0, основной единицей модели IDEF3 является диаграмма. Другой важный компонент модели — *действие*, или в терминах IDEF3 «единица работы» (*Unit of Work — UOW*). Диаграммы IDEF3 отображают действие в виде прямоугольника. Действия именуются с использованием глаголов или отглагольных существительных, каждому из действий присваивается уникальный идентификационный номер. Этот номер не используется вновь даже в том случае, если в процессе построения модели действие удаляется. В диаграммах IDEF3 номер действия обычно предваряется номером его родителя (рис. 2.13).

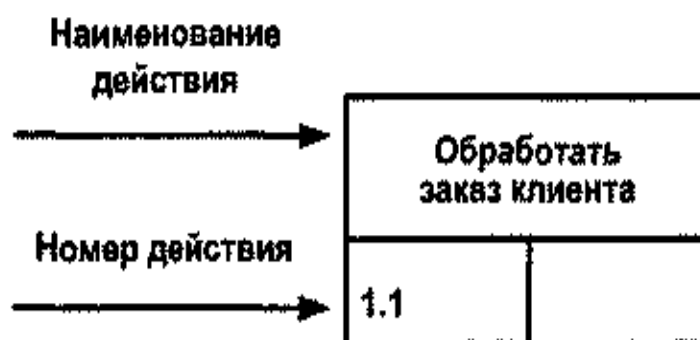




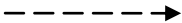
Рис. 2.13. Изображение и нумерация действия в диаграмме IDEF3

Существенные взаимоотношения между действиями изображаются с помощью связей. Все связи в IDEF3 являются однонаправленными, и хотя стрелка может начинаться или заканчиваться на любой стороне блока, обозначающего

действие, диаграммы IDEF3 обычно организуются слева направо таким образом, что стрелки начинаются на правой и заканчиваются на левой стороне блоков. В табл. 2.3 приведены три возможных типа связей.

Таблица 2.3

Типы связей IDEF3

Изображение	Название	Назначение
	Временное предшествование (Temporal precedence)	Исходное действие должно завершиться, прежде чем конечное действие сможет начаться
	Объектный поток (Object flow)	Выход исходного действия является входом конечного действия (исходное действие должно завершиться, прежде чем конечное действие сможет начаться)
	Нечеткое отношение (Relationship)	Вид взаимодействия между исходным и конечными действиями задается аналитиком отдельно для каждого случая использования такого отношения

Связь типа «временное предшествование» показывает, что исходное действие должно полностью завершиться, прежде чем начнется выполнение конечного действия.

Связь типа «объектный поток» используется в том случае, когда некоторый объект, являющийся результатом выполнения исходного действия, необходим для выполнения конечного действия. Обозначение такой связи отличается от связи временного предшествования двойной стрелкой. Наименования потоковых связей должны четко идентифицировать объект, который передается с их помощью. Временная семантика объектных связей аналогична связям предшествования, это означает, что порождающее объектную связь исходное действие

должно завершиться, прежде чем конечное действие может начать выполняться.

Связь типа «нечеткое отношение» используется для выделения отношений между действиями, которые невозможно описать с использованием связей предшествования или объектных связей. Значение каждой такой связи должно быть определено, поскольку связи типа «нечеткое отношение» сами по себе не предполагают никаких ограничений. Одно из применений нечетких отношений — отображение взаимоотношений между параллельно выполняющимися действиями.

Завершение одного действия может инициировать начало выполнения сразу нескольких других действий или, наоборот, определенное действие может требовать завершения нескольких других действий до начала своего выполнения. *Соединения* разбивают или соединяют внутренние потоки и используются для изображения ветвления процесса:

- *разворачивающие соединения* используются для разбиения потока. Завершение одного действия вызывает начало выполнения нескольких других;
- *сворачивающие соединения* объединяют потоки. Завершение одного или нескольких действий вызывает начало выполнения другого действия.

В табл. 2.4 описаны три типа соединений.

Таблица 2.4

Типы соединений

Графическое обозначение	Название	Вид	Правила инициации
&	Соединение «и»	Разворачивающее	Каждое конечное действие обязательно инициируется
		Сворачивающее	Каждое исходное действие обязательно должно завершиться
X	Соединение «исключающее «или»»	Разворачивающее	Одно и только одно конечное действие инициируется
		Сворачивающее	Одно и только одно исходное действие должно завершиться

О	Соединение «или»	Разворачивающее	Одно или несколько конечных действий инициируются
		Сворачивающее	Одно или несколько исходных действий должны завершиться

Соединения «и» инициируют выполнение конечных действий. Все действия, присоединенные к сворачивающему соединению «и», должны завершиться, прежде чем начнется выполнение следующего действия. На рис. 2.14 после обнаружения пожара инициируются включение пожарной сигнализации, вызов пожарной охраны, и начинается тушение пожара. Запись в журнал производится только тогда, когда все три перечисленных действия завершены.

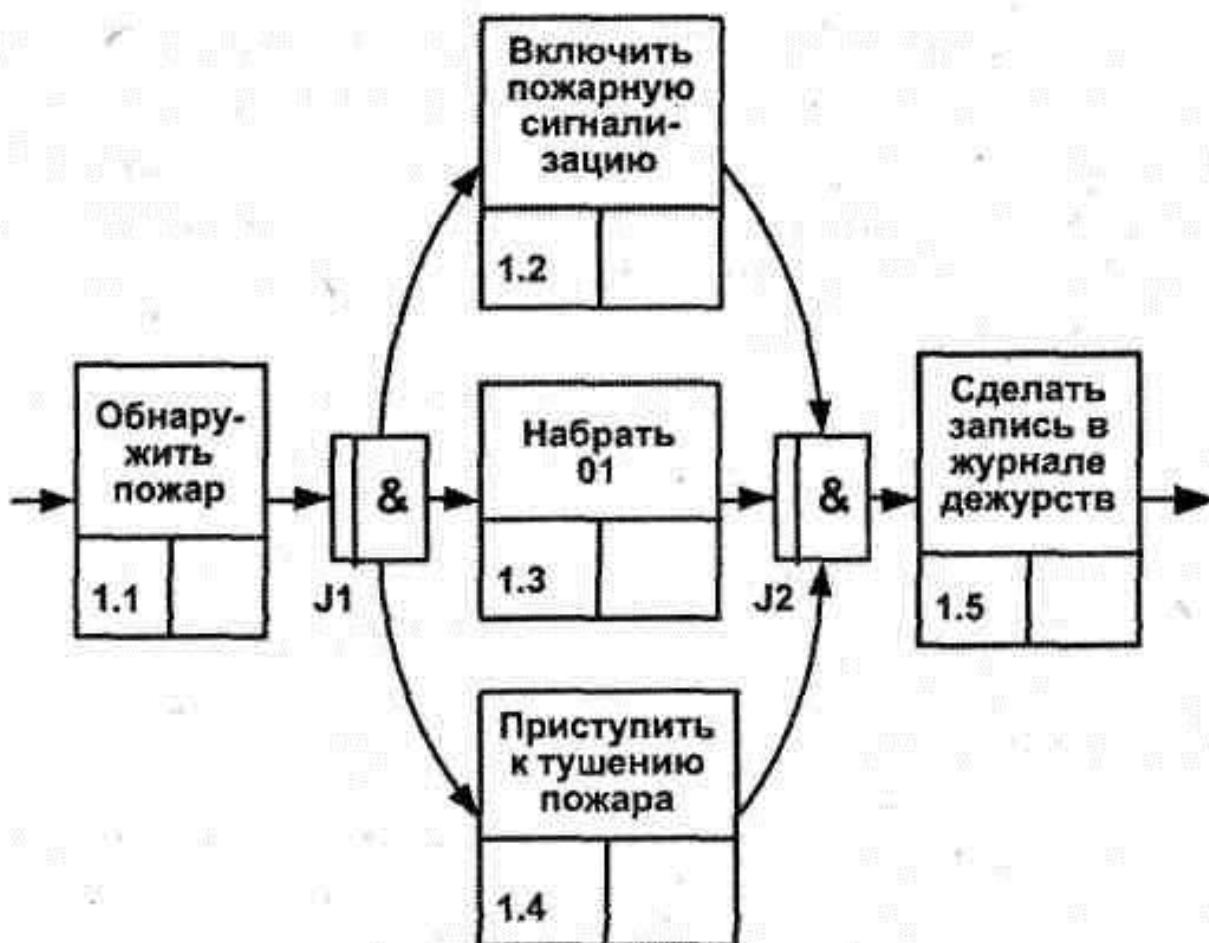


Рис. 2.14. Соединения «и»

Соединение «исключающее «или»» означает, что вне зависимости от количества действий, связанных со сворачивающим или разворачивающим соеди-

нением, инициировано будет только одно из них, и поэтому только оно будет завершено перед тем, как любое действие, следующее за сворачивающим соединением, сможет начаться. Если правила активации соединения известны, они обязательно должны быть документированы либо в его описании, либо пометкой стрелок, исходящих из разворачивающего соединения. На рис. 2.15 соединение «исключающее «или»» используется для отображения того факта, что студент не может одновременно быть направлен на лекции по двум разным курсам.

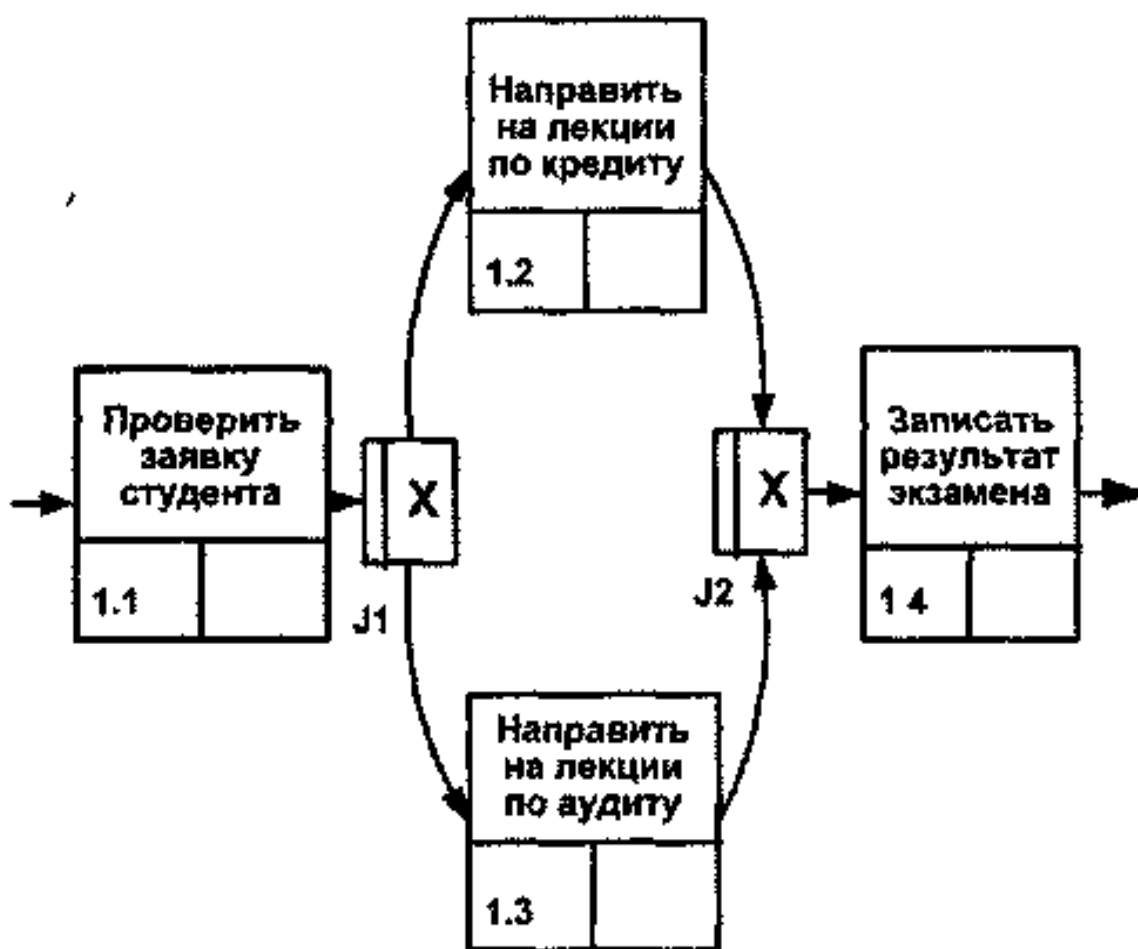


Рис. 2.15. Соединение «исключающее «или»»

Соединение «или» предназначено для описания ситуаций, которые не могут быть описаны двумя предыдущими типами соединений. Аналогично связи нечеткого отношения соединение «или» в основном определяется и описывает

ся непосредственно системным аналитиком. На рис. 2.16 соединение J2 может активизировать проверку данных чека и/или проверку суммы наличных. Проверка чека инициируется, если покупатель желает расплатиться чеком, проверка суммы наличных — при оплате наличными. То и другое действие инициируется при частичной оплате как чеком, так и наличными.

В рассмотренных примерах все действия выполнялись асинхронно, т.е. они не инициировались одновременно. Однако существуют случаи, когда время начала или окончания параллельно выполняемых действий должно быть одинаковым, т.е. действия должны выполняться синхронно. Для моделирования такого поведения системы используются различные виды синхронных соединений, которые обозначаются двумя двойными вертикальными линиями внутри прямоугольника.

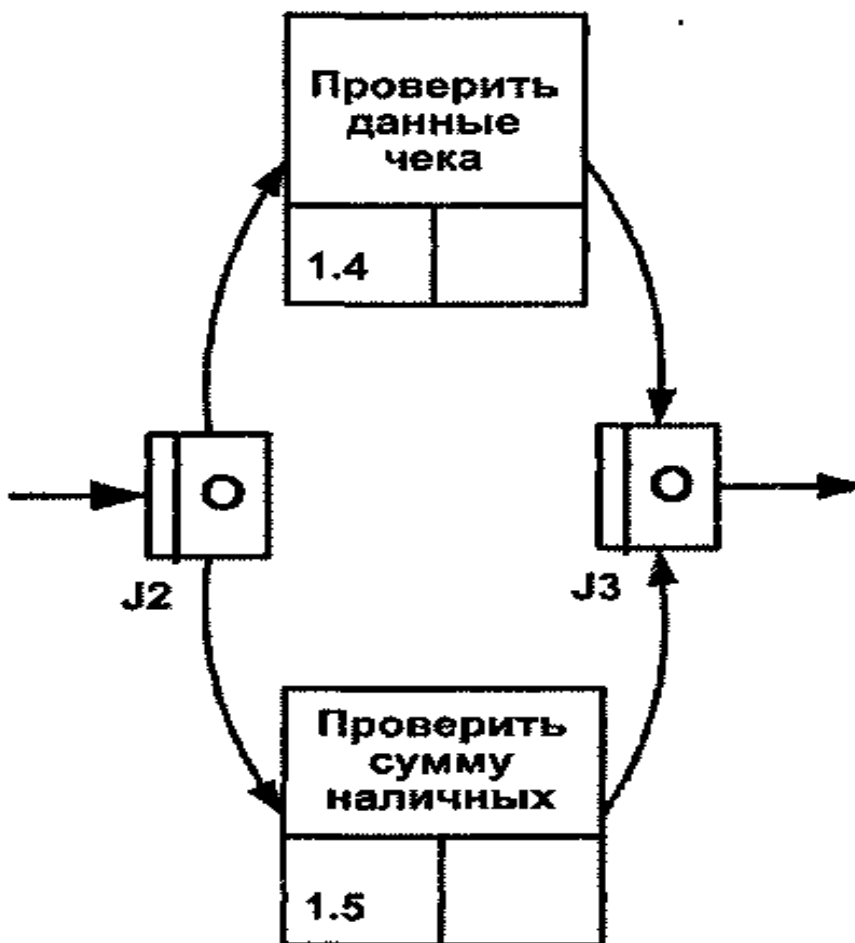


Рис. 2.16. Соединения «или»

Например, в спортивных состязаниях выстрел стартового пистолета, запуск секундомера и начало состязания должны произойти одновременно. На рис. 2.17 представлена модель этого примера, построенная с использованием синхронного соединения.

Синхронное разворачивающее соединение не обязательно должно иметь парное сворачивающее соединение, т.е. начинающиеся одновременно действия не обязаны оканчиваться одновременно. Возможны также ситуации синхронного окончания асинхронно начавшихся действий.

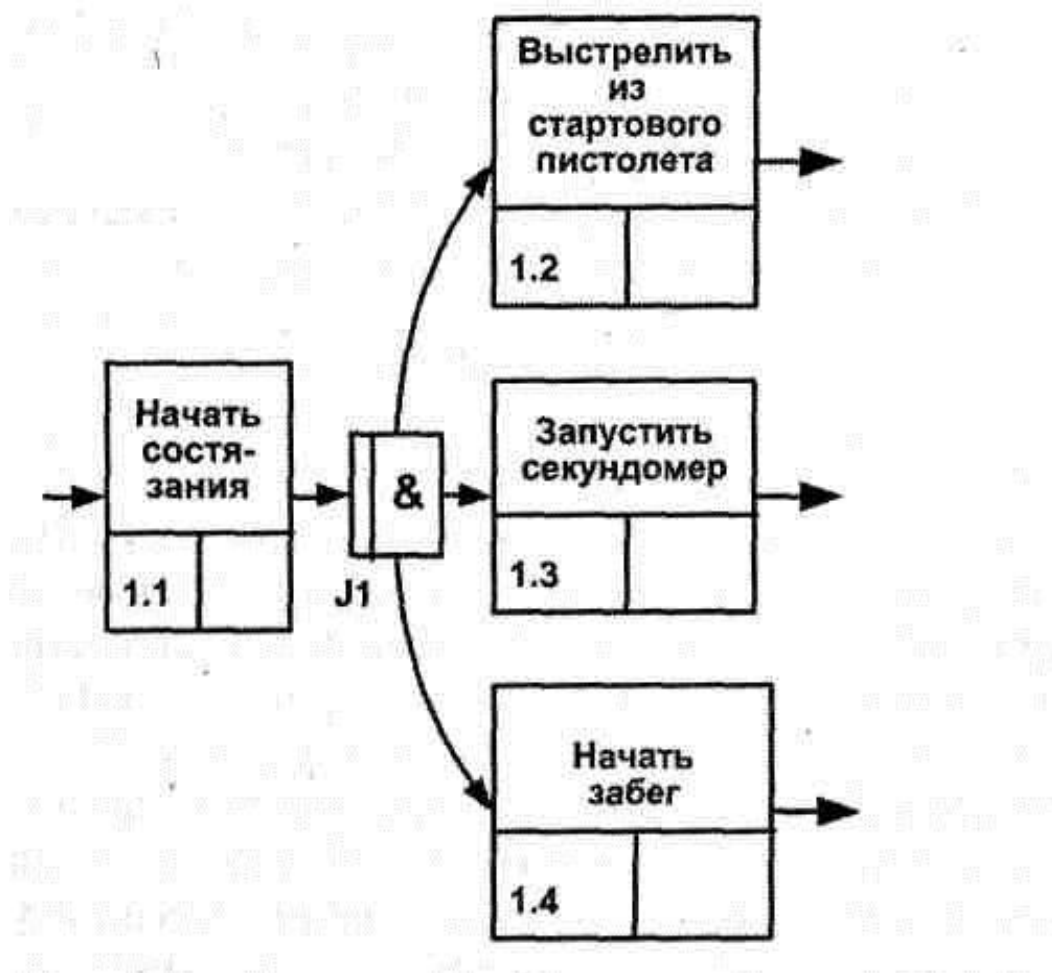


Рис. 2.17. Синхронное соединение

Все соединения на диаграммах должны быть парными, из чего следует, что любое разворачивающее соединение имеет парное себе сворачивающее. Однако типы соединений не обязательно должны совпадать.

Соединения могут комбинироваться для создания более сложных ветвлений. Комбинации соединений следует использовать с осторожностью, поскольку

ку перегруженные ветвлением диаграммы могут оказаться сложными для восприятия.

Действия в IDEF3 могут быть декомпозированы или разложены на составляющие для более детального анализа. Метод IDEF3 позволяет декомпонировать действие несколько раз, что обеспечивает документирование альтернативных потоков процесса в одной модели.

2.3.3.

МОДЕЛИРОВАНИЕ ПОТОКОВ ДАННЫХ

Общие сведения

*Диаграммы потоков данных*¹ (*Data Flow Diagrams — DFD*) представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордона—ДеМарко и Гейна—Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов. Далее в примерах будет использоваться нотация Гейна—Сэрсона.

В соответствии с данным методом модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи потребителю. Источники информации (внешние сущности) порождают информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам.

¹ *Калашиян А.Н., Каляное Г.Н.* Структурные модели бизнеса: DFD-технологии. — М.: Финансы и статистика, 2003

Те, в свою очередь, преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или внешним сущностям — потребителям информации.

Диаграммы верхних уровней иерархии (контекстные диаграммы) определяют основные процессы или подсистемы с внешними входами и выходами. Они детализируются при помощи диаграмм нижнего уровня. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут уровень декомпозиции, на котором детализировать процессы далее не имеет смысла.

Состав диаграмм потоков в данных

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы и подсистемы;
- процессы;
- накопители данных;
- потоки данных.

Внешняя сущность представляет собой материальный объект или физическое лицо, источник или приемник информации (например, заказчики, персонал, поставщики, клиенты, склад). Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой системы. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой системы, если это необходимо, или, наоборот, часть процессов может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (рис. 2.18), расположенным как бы над диаграммой и бросающим на нее тень для того, чтобы можно было выделить этот символ среди других обозначений.

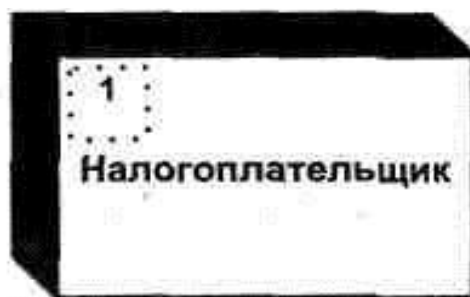


Рис. 2.18. Графическое изображение внешней сущности

При построении модели сложной системы она может быть представлена в самом общем виде на так называемой *контекстной диаграмме* в виде одной *системы* как единого целого, либо может быть декомпозирована на ряд *подсистем*.

Подсистема (или система) на контекстной диаграмме изображается так, как она представлена на рис. 2.19.

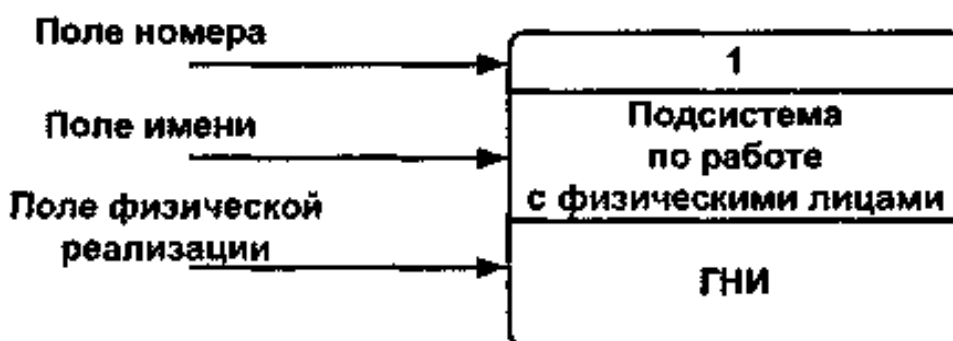


Рис. 2.19. Подсистема по работе с физическими лицами (ГНИ — Государственная налоговая инспекция)

Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями.

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющее обработку входных документов и выпуск отчетов; программа; аппаратно реализованное логическое устройство и т.д.

Процесс на диаграмме потоков данных изображается, как показано на рис. 2.20.

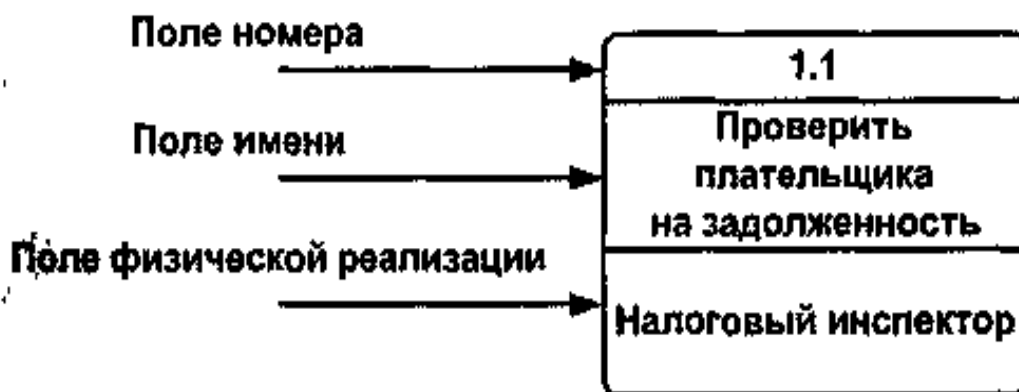


Рис. 2.20. Графическое изображение процесса

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например: «Ввести сведения о налогоплательщиках», «Выдать информацию о текущих расходах», «Проверить поступление денег».

Использование таких глаголов, как «обработать», «модернизировать» или «отредактировать» означает, как правило, недостаточно глубокое понимание данного процесса и требует дальнейшего анализа.

Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

Накопитель данных — это абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми.

Накопитель данных может быть реализован физически в виде микрофиши, ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носи-

теле и т.д. Накопитель данных на диаграмме потоков данных изображается, как показано на рис. 2.21.



Рис. 2.21. Графическое изображение накопителя данных

Накопитель данных идентифицируется буквой «D» и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика.

Накопитель данных в общем случае является прообразом будущей базы данных, и описание хранящихся в нем данных должно соответствовать информационной модели (ERM).

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой, и т.д.

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (рис. 2.22). Каждый поток данных имеет имя, отражающее его содержание.



Рис. 2.22. Поток данных

Построение иерархии диаграмм потоков данных

Главная цель построения иерархии DFD заключается в том, чтобы сделать описание системы ясным и понятным на каждом уровне детализации, а также разбить его на части с точно определенными отношениями между ними. Для достижения этого целесообразно пользоваться следующими рекомендациями:

- размещать на каждой диаграмме от 3 до 6—7 процессов (аналогично SADT). Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один или два процесса;
- не загромождать диаграммы несущественными на данном уровне деталями;
- декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой;
- выбирать ясные, отражающие суть дела, имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Первым шагом при построении иерархии DFD является построение контекстных диаграмм. Обычно при проектировании относительно простых систем строится единственная контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы. Перед построением контекстной DFD необходимо проанализировать внешние события (внешние сущности), оказывающие влияние на функционирование системы. Количество потоков на контекстной диаграмме должно быть по возможности небольшим, поскольку каждый из них может быть в дальнейшем разбит на несколько потоков на следующих уровнях диаграммы.

Для проверки контекстной диаграммы можно составить список событий, который должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на них. Каждое событие должно соответствовать одному или более потокам данных: входные потоки интерпретируются как воздействия, а выходные потоки — как реакции системы на входные потоки.

Если для сложной системы ограничиться единственной контекстной диаграммой, то она будет содержать слишком большое количество источников и приемников информации, которые трудно расположить на листе бумаги нормального формата, и, кроме того, единственный главный процесс не раскрывает структуры такой системы. Признаками сложности (в смысле контекста) могут быть: наличие большого количества внешних сущностей (десять и более), распределенная природа системы, многофункциональность системы с уже сложившейся или выявленной группировкой функций в отдельные подсистемы.

Для сложных систем строится иерархия контекстных диаграмм. При этом контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют контекст и структуру подсистем.

Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем между собой, с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует система.

Разработка контекстных диаграмм решает проблему строгого определения функциональной структуры системы на самой ранней стадии ее проектирования, что особенно важно для сложных многофункциональных систем, в создании которых участвуют разные организации и коллективы разработчиков.

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Это можно сделать путем построения диаграммы для каждого события. Каждое событие представляется в виде процесса с соответствующими входными и выходными потоками, накопителями данных, внешними сущностями и ссылками на другие процессы для описания связей между этим процессом и его окружением. Затем все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Каждый процесс на DFD, в свою очередь, может быть детализирован при помощи DFD или (если процесс элементарный) спецификации. При детализации должны выполняться следующие правила:

- *правило балансировки* — при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников или приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми имеют информационную связь детализируемые подсистема или процесс на родительской диаграмме;
- *правило нумерации* — при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3 и т.д.

Спецификация процесса должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Спецификация является конечной вершиной иерархии DFD. Решение о завершении детализации процесса и использовании спецификации принимается аналитиком исходя из следующих критериев:

- наличия у процесса относительно небольшого количества входных и выходных потоков данных (2—3 потока);
- возможности описания преобразования данных процессом в виде последовательного алгоритма;

- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности описания логики процесса при помощи спецификации небольшого объема (не более 20—30 строк).

Спецификации должны удовлетворять следующим требованиям:

- для каждого процесса нижнего уровня должна существовать одна и только одна спецификация;
- спецификация должна определять способ преобразования входных потоков в выходные;
- нет необходимости (по крайней мере, на стадии формирования требований) определять метод реализации этого преобразования;
- спецификация должна стремиться к ограничению избыточности — не следует переопределять то, что уже было определено на диаграмме;
- набор конструкций для построения спецификации должен быть простым и понятным.

Фактически спецификации представляют собой описания алгоритмов задач, выполняемых процессами. Спецификации содержат номер и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные. Известно большое количество разнообразных методов, позволяющих описать тело процесса. Соответствующие этим методам языки могут варьироваться от структурированного естественного языка или псевдокода до визуальных языков моделирования.

Структурированный естественный язык применяется для читабельного, достаточно строгого описания спецификаций процессов. Он представляет собой разумное сочетание строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в определенные логические структуры, арифметических выражений и диаграмм.

В состав языка входят следующие основные символы:

- глаголы, ориентированные на действие и применяемые к объектам;

- термины, определенные на любой стадии проекта ПО (например, задачи, процедуры, символы данных и т.п.);
- предлоги и союзы, используемые в логических отношениях;
- общеупотребительные математические, физические и технические термины;
- арифметические уравнения;
- таблицы, диаграммы, графы и т.п.;
- комментарии.

К управляющим структурам языка относятся последовательная конструкция, конструкция выбора и итерация (цикл).

При использовании структурированного естественного языка приняты следующие соглашения:

- логика процесса выражается в виде комбинации последовательных конструкций, конструкций выбора и итераций;
- глаголы должны быть активными, недвусмысленными и ориентированными на целевое действие (заполнить, вычислить, извлечь, а не модернизировать, обработать);
- логика процесса должна быть выражена четко и недвусмысленно.

При построении иерархии DFD переходить к детализации процессов следует только после определения содержания всех потоков и накопителей данных, которое описывается при помощи структур данных. Для каждого потока данных формируется список всех его элементов данных, затем элементы данных объединяются в структуры данных, соответствующие более крупным объектам данных (например, строкам документов или объектам предметной области). Каждый объект должен состоять из элементов, являющихся его атрибутами. Структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что данный компонент может отсутствовать в структуре (например, структура «данные о страховании» для объекта «служащий»). Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает вхождение любого числа элемен-

тов в указанном диапазоне (например, элемент «имя ребенка» для объекта «служащий»). Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для непрерывных данных могут указываться единица измерения (кг, см и т.п.), диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

2.3.4

КОЛИЧЕСТВЕННЫЙ АНАЛИЗ ДИАГРАММ

IDEFO И DFD

Для проведения количественного анализа диаграмм IDEF0 и DFD используются следующие показатели:

- количество блоков на диаграмме - N ;
- уровень декомпозиции диаграммы — L ;
- число стрелок, соединяющихся с i -м блоком диаграммы — A_i . Данный набор показателей относится к каждой диаграмме модели. Ниже перечислены рекомендации по их желательным значениям.

Необходимо стремиться к тому, чтобы количество блоков на диаграммах нижних уровней было бы ниже количества блоков на родительских диаграммах,

т. е. с увеличением уровня декомпозиции убывал бы коэффициент N/L . По мере декомпозиции модели функции должны упрощаться, следовательно, количество блоков должно убывать.

Диаграммы должны быть сбалансированы. Это означает, например, что у любого блока количество входящих стрелок и стрелок управления не должно быть значительно больше, чем количество выходящих. Следует отметить, что данная рекомендация может не выполняться в моделях, описывающих производственные процессы. Например, при описании процедуры сборки в блок может входить множество стрелок, описывающих компоненты изделия, а выходить одна стрелка — готовое изделие.

Количественная оценка сбалансированности диаграммы может быть выполнена с помощью коэффициента сбалансированности:

$$K_b = \left| \sum A_i / N - \max(A_i) \right|.$$

Необходимо стремиться к тому, чтобы значение K_b для диаграммы было минимальным.

2.3.5.

СРАВНИТЕЛЬНЫЙ АНАЛИЗ SADT-МОДЕЛЕЙ И ДИАГРАММ ПОТОКОВ ДАННЫХ

Сравнительный анализ данных методов структурного анализа проводится по следующим параметрам¹:

- адекватность средств решаемым задачам;
- согласованность с другими средствами структурного анализа;
- интеграция с другими процессами ЖЦ ПО (прежде всего с процессом проектирования).

¹ *Калашян А.Н., Каляное Т.Н.* Структурные модели бизнеса: DFD-технологии. — М.: Финансы и статистика, 2003

Адекватность средств решаемым задачам. Модели SADT (IDEF0) традиционно используются для моделирования организационных систем (бизнес-процессов). С другой стороны, не существует никаких принципиальных ограничений на использовании DFD в качестве средства моделирования бизнес-процессов. Следует отметить, что метод SADT успешно работает только при описании хорошо специфицированных и стандартизованных бизнес-процессов в зарубежных корпорациях, поэтому он и принят в США в качестве типового. Например, в Министерстве обороны США десятки лет существуют четкие должностные инструкции и методики, которые жестко регламентируют деятельность, делают ее высокотехнологичной и ориентированной на бизнес-процесс. Достоинствами применения моделей SADT для описания бизнес-процессов являются:

- полнота описания бизнес-процесса (управление, информационные и материальные потоки, обратные связи);
- комплексная декомпозиция; возможность агрегирования и детализации потоков данных и управления (разделение и слияние стрелок);
- жесткие требования метода, обеспечивающие получение моделей стандартного вида;
- соответствие подхода к описанию процессов стандартам ISO 9000.

В большинстве российских организаций бизнес-процессы начали формироваться и развиваться сравнительно недавно, они слабо типизированы, поэтому разумнее ориентироваться на модели, основанные на потоковых диаграммах. Кроме того, на практике у большинства моделей SADT отмечается ряд недостатков, в частности:

- сложность восприятия (большое количество стрелок);
- большое количество уровней декомпозиции;
- трудность увязки нескольких процессов, представленных в различных моделях одной и той же организации.

Если же речь идет не о системах вообще, а о ПО ИС, то здесь DFD вне конкуренции. Практически любой класс систем успешно моделируется при по-

мощи DFD-ориентированных методов. SADT-диаграммы оказываются значительно менее выразительными и удобными при моделировании ПО. Так, дуги в SADT жестко типизированы (вход, выход, управление, механизм). В то же время применительно к ПО стирается смысловое различие между входами и выходами, с одной стороны, и управлениями и механизмами, с другой: входы, выходы и управления являются потоками данных и правилами их преобразования. Анализ системы при помощи потоков данных и процессов, их преобразующих, является более прозрачным и недвусмысленным.

В SADT вообще отсутствуют выразительные средства для моделирования особенностей ИС. DFD же с самого начала создавались как средство проектирования ИС (тогда как SADT — как средство моделирования систем вообще) и имеют более богатый набор элементов, адекватно отражающих специфику таких систем (например, хранилища данных являются прообразами файлов или баз данных, внешние сущности отражают взаимодействие моделируемой системы с внешним миром).

Наличие в DFD спецификаций процессов нижнего уровня позволяет преодолеть логическую незавершенность SADT (а именно, обрыв модели на некотором достаточно низком уровне, когда дальнейшая ее детализация становится бессмысленной) и построить полную функциональную спецификацию разрабатываемой системы.

Жесткие ограничения SADT, запрещающие использовать более 6-7 блоков на диаграмме, в ряде случаев вынуждают искусственно детализировать процесс, что затрудняет понимание модели заказчиком, резко увеличивает ее объем и, как следствие, ведет к неадекватности модели реальной предметной области. В качестве примера достаточно рассмотреть модель операции по снятию денег с вклада физического лица в банке. В настоящий момент существуют более тридцати типов таких вкладов. Для моделирования соответствующих операций целесообразно использовать единственную DFD, поскольку все без исключения операции имеют одни и те же входы (сберегательная книжка и расходный ордер) и выходы (сберегательная книжка и наличные деньги) и разли-

чаются лишь механизмами начисления процентов. Если же попытаться структурировать эти операции путем группирования по какому-либо признаку (срочные, пенсионные, размеры процентов и т.п.) в соответствии с ограничениями SADT, то получится как минимум 6 диаграмм (верхний уровень и округленная в большую сторону дробь $30/7$), сложность каждой из которых не меньше сложности единственной диаграммы, моделирующей все операции.

Согласованность с другими средствами структурного анализа. Главным достоинством любых моделей является возможность их интеграции с моделями других типов. В данном случае речь идет о согласованности функциональных моделей со средствами моделирования данных. Согласование SADT-модели с ERM практически невозможно или носит искусственный характер. В свою очередь, DFD и ERM взаимно дополняют друг друга и являются согласованными, поскольку в DFD присутствует описание структур данных, непосредственно используемое для построения ERM.

Интеграция с другими процессами ЖЦ ПО. Важная характеристика модели — ее совместимость с моделями, используемыми в последующих процессах (прежде всего в процессе проектирования).

DFD могут быть легко преобразованы в модели проектируемой системы. Известен ряд алгоритмов автоматического преобразования иерархии DFD в структурные карты различных видов, что обеспечивает логичный и безболезненный переход от формирования требований к проектированию системы. С другой стороны, формальные методы преобразования SADT-диаграмм в проектные решения отсутствуют.

Необходимо отметить, что рассмотренные разновидности средств структурного анализа примерно одинаковы с учетом возможностей изобразительных средств моделирования. При этом одним из основных критериев выбора того или иного метода является степень владения им со стороны консультанта или аналитика, грамотность выражения своих мыслей на языке моделирования. В противном случае в моделях, построенных с использованием *любого* метода, будет невозможно разобраться.

МОДЕЛИРОВАНИЕ ДАННЫХ

Основные понятия модели «сущность — связь»

Цель моделирования данных состоит в обеспечении разработчика системы концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных.

Наиболее распространенным средством моделирования данных (предметной области) является модель «сущность-связь» (ERM). Она была впервые введена Питером Ченом в 1976 г. Базовыми понятиями ERM являются сущность, связь и атрибут.

Сущность (Entity) — реальный либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. Примерами сущностей могут быть такие классы объектов, как «Поставщик», «Сотрудник», «Заказ». Каждая сущность в модели изображается в виде прямоугольника с наименованием (рис. 2.23).



Рис. 2.23. Графическое представление сущности

Основной (неформальный) способ идентификации сущностей — это поиск абстракций, описывающих физические или материальные объекты, процессы и события, роли людей, организации и другие понятия. Единственным формальным способом идентификации сущностей является анализ текстовых описаний предметной области, выделение из описаний имен существительных и выбор их в качестве «кандидатов» на роль абстракций.

Экземпляр сущности — это конкретный представитель данной сущности. Например, экземпляром сущности «Сотрудник» может быть «Сотрудник Иванов».

Экземпляры сущностей должны быть *различимы*, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности. Каждый экземпляр сущности должен однозначно идентифицироваться и отличаться от всех других экземпляров данного типа сущности. Каждая сущность должна обладать некоторыми свойствами:

- иметь уникальное имя; к одному и тому же имени должна всегда применяться одна и та же интерпретация; одна и та же интерпретация не может применяться к различным именам, если только они не являются псевдонимами;
- обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;
- обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Атрибут (Attribute) — *любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.*

Атрибут представляет тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов (людей, мест, событий, состояний, идей, предметов и т.д.). Экземпляр атрибута — это определенная характеристика отдельного элемента множества. Экземпляр атрибута определяется типом характеристики и ее значением, называемым значением атрибута. В ERM атрибуты ассоциируются с конкретными сущностями. Таким образом, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута.

Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности «Сотрудник» могут быть такие атрибуты, как «Табельный номер», «Фамилия», «Имя», «Отчество», «Должность», «Зарплата» и т.п.

Атрибуты изображаются в пределах прямоугольника, определяющего сущность (рис. 2.24).

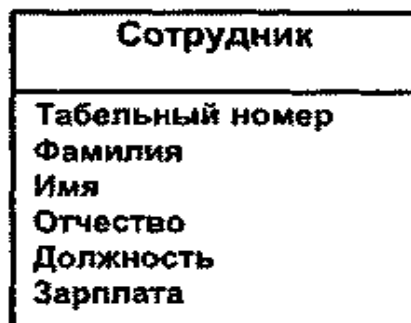


Рис. 2.24. Сущность с атрибутами

Виды атрибутов:

- простой — состоит из одного элемента данных;
- составной — состоит из нескольких элементов данных;
- однозначный — содержит одно значение для одной сущности;
- многозначный — содержит несколько значений для одной сущности;
- необязательный - может иметь пустое (неопределенное) значение;
- производный — представляет значение, производное от значения связанного с ним атрибута.

Уникальным идентификатором называется *неизбыточный* набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности. *Неизбыточность* заключается в том, что удаление любого атрибута из уникального идентификатора нарушает его уникальность.

Сущность может иметь несколько различных уникальных идентификаторов, они изображаются на диаграмме подчеркиванием (рис. 2.25).

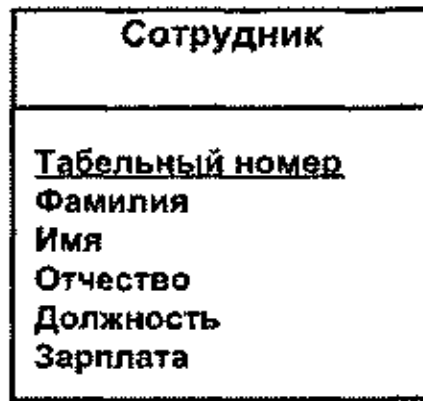


Рис. 2.25. Сущность с уникальным идентификатором

Каждая сущность может обладать любым количеством связей с другими сущностями модели. *Связь (Relationship)* — *поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области*. Связь - это ассоциация между сущностями, при которой каждый экземпляр одной сущности ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, и наоборот.

Степенью связи называется количество сущностей, участвующих в связи. Связь степени 2 называется *бинарной*, степени N — *N -арной*. Связь, в которой одна и та же сущность участвует в разных ролях, называется *рекурсивной*, или *унарной*. Один из возможных вариантов графического изображения связи показан на рис. 2.26.

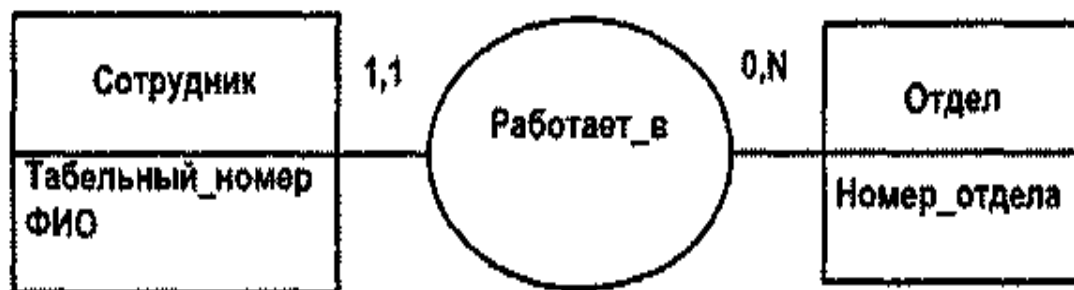


Рис. 2.26. Обозначение сущностей и связи

Пары чисел на диаграмме отражают две важные характеристики связи — мощность связи (второе число) и класс принадлежности (первое число).

Мощностью связи называется максимальное число экземпляров сущности, которое может быть связано с одним экземпляром данной сущности. Мощность связи может быть равна 1, N (любое число) и может быть конкретным числом. Мощности связи на рис. 2.26 означают: каждый сотрудник может работать не более чем в одном отделе, а в каждом отделе может работать любое число сотрудников.

Класс принадлежности характеризует обязательность участия экземпляра сущности в связи. Класс принадлежности может принимать значение 0 (*необязательное участие* — экземпляр одной сущности *может быть связан* с одним или несколькими экземплярами другой сущности, *а может быть и не связан* ни с одним экземпляром) или 1 (*обязательное участие* — экземпляр одной сущности *должен быть связан не менее чем с одним* экземпляром другой сущности). Классы принадлежности на рис. 2.26 означают: каждый сотрудник обязательно работает в каком-либо отделе, а в некоторых отделах может и не быть сотрудников.

Связь может иметь один из следующих трех типов (в зависимости от значения мощности):

1. *Один-к-одному* (обозначается 1:1), показана на рис. 2.27.

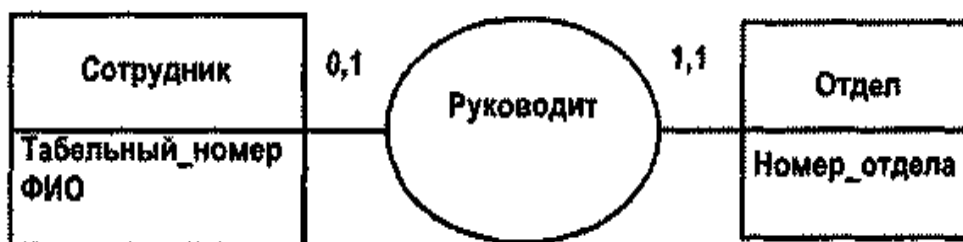


Рис. 2.27. Связь типа 1:1

2. *Один-ко-многим* (обозначается 1:n), показана на рис. 2.26.
3. *Многие-ко-многим* (обозначается m:n), показана на рис. 2.28.



Рис. 2.28. Связь типа min

Виды идентификаторов

Существуют следующие виды идентификаторов:

- *первичный/альтернативный*: сущность может иметь несколько идентификаторов (рис. 2.29). Один должен являться основным (первичным), а другие — альтернативными. Первичный идентификатор на диаграмме подчеркивается. Альтернативные идентификаторы предваряются символами <1> для первого альтернативного идентификатора, <2> для второго и т.д. В концептуальном моделировании данных различие первичных и альтернативных идентификаторов обычно не используется. В реляционной модели, полученной из концептуальной модели данных, первичные ключи используются в качестве внешних ключей. Альтернативные идентификаторы не копируются в качестве внешних ключей в другие таблицы;
- *простой/составной*: идентификатор, состоящий из одного атрибута, является простым, из нескольких атрибутов — составным (см. рис. 2.29);
- *абсолютный/относительный*: если все атрибуты, составляющие идентификатор, принадлежат сущности, то идентификатор является абсолютным. Если один или более атрибутов идентификатора принадлежат другой сущности, то идентификатор является относительным. Когда первичный идентификатор является относительным, сущность определяется как *зависимая сущность*, поскольку ее идентификатор зависит от другой сущности. В примере на рис. 2.30 идентификатор сущности «Строка-

заказа» является относительным. Он включает идентификатор сущности «Заказ», что показано на рисунке подчеркиванием 1.1.



Рис. 2.29. Составной альтернативный идентификатор

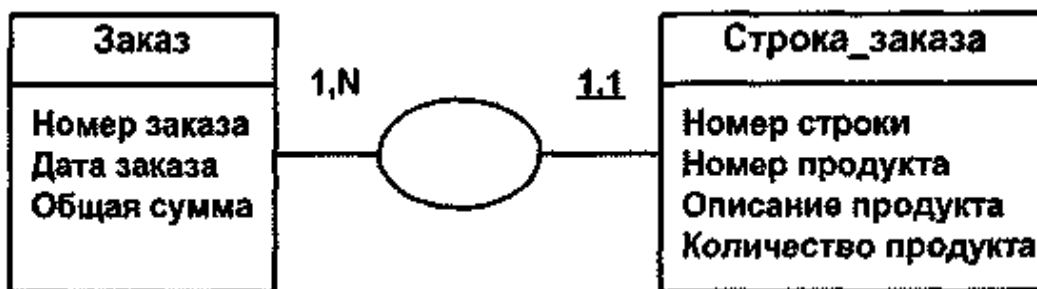


Рис. 2.30. Относительный идентификатор

Связи с атрибутами

Как и сущности, связи могут иметь атрибуты. В примере на рис. 2.31 для того, чтобы найти оценку студента, нужно знать не только идентификатор студента, но и номер курса. Оценка не является атрибутом студента или атрибутом курса; она является атрибутом обеих этих сущностей. Это атрибут связи между студентом и курсом, которая в примере называется «Регистрация».

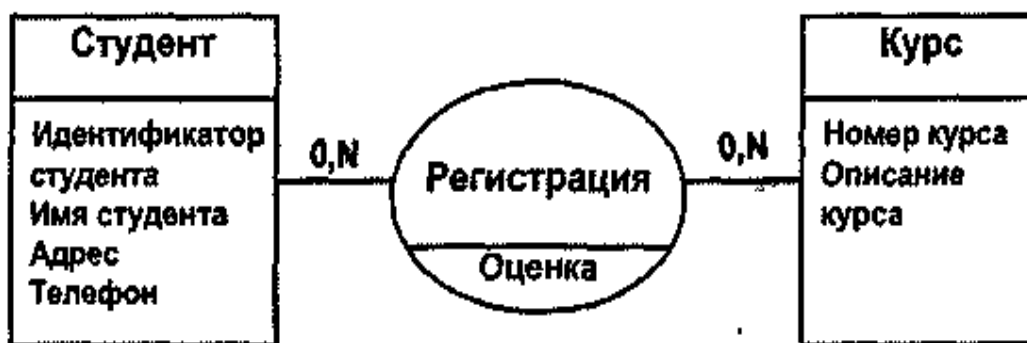


Рис. 2.31. Связь с атрибутами

Связь между сущностями в концептуальной модели данных является типом, который представляет множество экземпляров связи между экземплярами сущностей. Для того чтобы идентифицировать определенный экземпляр сущности, используется идентификатор сущности. Точно так же для определения экземпляров связи между сущностями требуется идентификатор связи. Так, в примере на рис. 2.31 идентификатором связи «Регистрация» является идентификатор студента и номер курса, поскольку вместе они определяют конкретный экземпляр связи студентов и курсов.

Связи «супертип-подтип»

В связи «супертип-подтип» (рис. 2.32) общие атрибуты типа определяются в сущности-супертипе, сущность-подтип наследует все атрибуты супертипа. Экземпляр подтипа существует только при условии существования определенного экземпляра супертипа. Подтип не может иметь идентификатора (он импортирует его из супертипа).



Рис. 2.32. Связь «супертип-подтип»

Пример нотации модели «сущность-связь» — метод IDEF1X

Метод IDEF1X, входящий в семейство стандартов IDEF, использует разновидность модели «сущность-связь» и реализован в ряде распространенных CASE-средств (в частности, ERwin).

Сущность в методе IDEF1X является *независимой от идентификаторов*, или просто *независимой*, если каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями. Сущность называется *зависимой от идентификаторов*, или просто *зависимой*, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности (рис. 2.33).

Каждой сущности присваивается уникальное имя и номер, разделяемые косой чертой «/» и помещаемые над блоком.

Связь может дополнительно определяться с помощью указания мощности (количества экземпляров сущности-потомка, которое может существовать для

каждого экземпляра сущности-родителя). В IDEF1X могут быть выражены следующие мощности связей:



Рис. 2.33. Независимые (а) и зависимые (б) от идентификатора сущности

- каждый экземпляр сущности-родителя может иметь нуль, один или более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.

Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь называется идентифицирующей, в противном случае — неидентифицирующей.

Связь изображается линией, проводимой между сущностью-родителем и сущностью-потомком с точкой на конце линии у сущности-потомка (рис. 2.34).



Рис. 2.34. Графическое изображение мощности связи

Мощность связи может принимать следующие значения: N — нуль, один или более, Z — нуль или один, P — один или более. По умолчанию мощность связи принимается равной N .

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией (рис. 2.35). Сущность-потомок в идентифицирующей связи является зависимой от идентификатора сущностью. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от идентификатора сущностью (это определяется ее связями с другими сущностями).

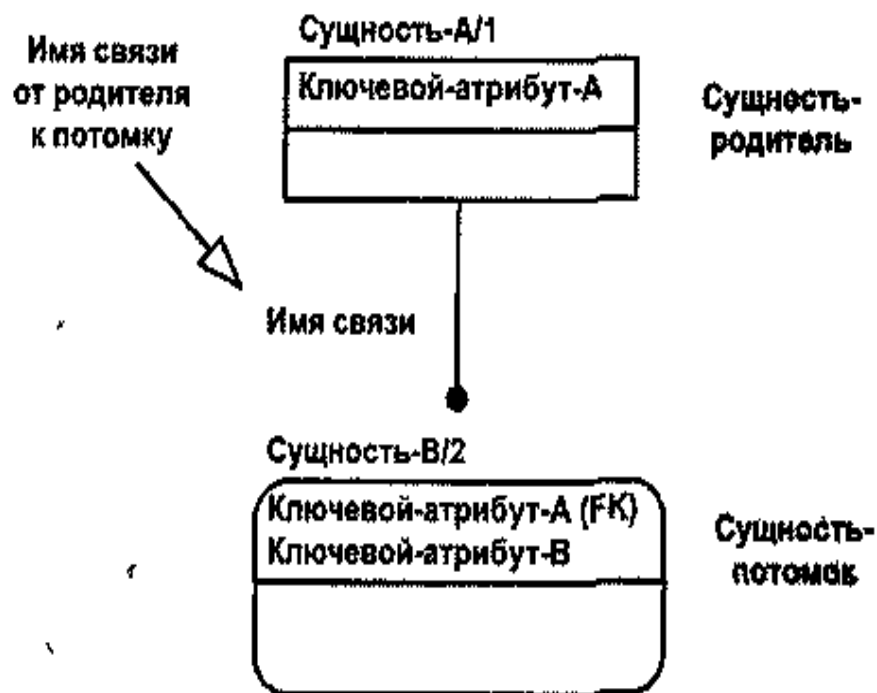


Рис. 2.35. Идентифицирующая связь

Пунктирная линия изображает неидентифицирующую связь (рис. 2.36).

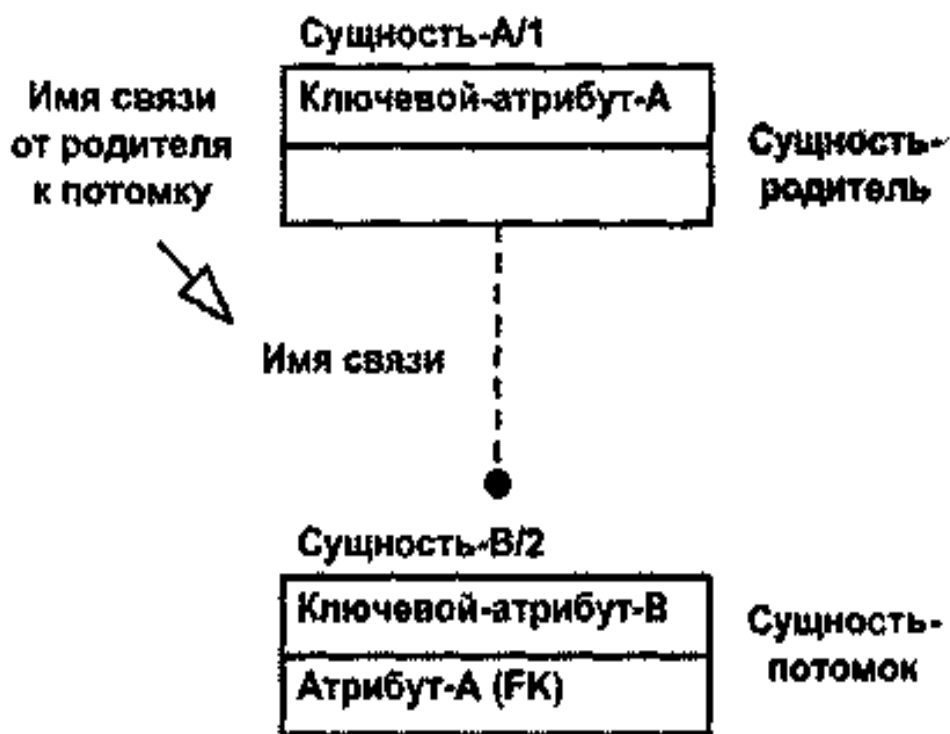


Рис. 2.36. Неидентифицирующая связь

Сущность-потомок в неидентифицирующей связи будет независимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.

Атрибуты изображаются в виде списка имен внутри блока сущности. Атрибуты, определяющие первичный ключ, размещаются наверху списка и отделяются от других атрибутов горизонтальной чертой.

Сущности могут иметь также внешние ключи (Foreign Key), которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута. Внешний ключ изображается с помощью помещения внутрь блока сущности имен атрибутов, после которых следуют буквы FK в скобках.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ АНАЛИЗА И ПРОЕКТИРОВАНИЯ ПО

В основе объектно-ориентированного подхода (ООП) лежит *объектная* декомпозиция, при этом статическая структура системы описывается в терминах *объектов* и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Проблемы, стимулировавшие развитие ООП:

- необходимость повышения производительности разработки за счет многократного (повторного) использования ПО;
- необходимость упрощения сопровождения и модификации разработанных систем (локализация вносимых изменений);
- облегчение проектирования систем (за счет сокращения семантического разрыва между структурой решаемых задач и структурой ПО).

Объектная модель является наиболее естественным способом представления реального мира. В разделе «Теория классификации» Британской энциклопедии сказано следующее:

«В постижении реального мира люди пользуются тремя методами, организующими их мышление:

(1) разделение окружающей действительности на конкретные объекты и их атрибуты (например, когда явно различаются понятия дерева и его высоты или пространственного расположения по отношению к другим объектам);

(2) различие между целыми объектами и их составными частями (например, ветви являются составными частями дерева);

(3) формирование и выделение различий между различными классами объектов (например, между классом всех деревьев и классом всех камней.)»

Понятие «объект» впервые было использовано около 30 лет назад в технических средствах при попытках отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования на уровне компьютеров. С объектно-ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Однако наиболее значительный вклад в объектный подход был внесен объектными и объектно-ориентированными языками программирования: Simula (1967), Smalltalk (1970-е гг.), C++ (1980-е гг.) и языком моделирования UML (1990-е гг.). На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования данных, в особенности модель «сущность-связь».

2.4.1.

ОСНОВНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ ОБЪЕКТНОЙ МОДЕЛИ

Концептуальной основой объектно-ориентированного подхода является *объектная модель*. Основными принципами ее построения являются¹:

- абстрагирование (abstraction);
- инкапсуляция (encapsulation);
- модульность (modularity);
- иерархия (hierarchy).

Абстрагирование — это выделение наиболее важных, существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей. Абстрагирование позво-

¹ Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. - 2-е изд.: Пер. с англ. - М.: Бином. - СПб.: Невский диалект, 1999.

ляет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Абстракция зависит от предметной области и точки зрения — то, что важно в одном контексте, может быть не важно в другом. **Объекты и классы** — основные абстракции предметной области.

Инкапсуляция — физическая локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающая их реализацию за общедоступным интерфейсом. Инкапсуляция — это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только операции самого объекта, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция (или иначе ограничение доступа) не позволяет объектам-пользователям различать внутреннее устройство объекта.

По-другому инкапсуляцию можно описать, сказав, что приложение разделяется на небольшие фрагменты связанной функциональности. Допустим, в банковской системе имеется информация, касающаяся банковского счета, такая как номер счета, баланс, имя и адрес его владельца, тип счета, начисляемые на него проценты и дата открытия. Со счетом также связаны определенные действия: открыть, закрыть его, положить или снять некоторую сумму денег, а также изменить тип, владельца или адрес. Вся эта информация и действия (поведение) совместно инкапсулируются в объект «счет». В результате все изменения банковской системы, связанные со счетами, могут быть реализованы в одном только объекте «счет».

Еще одним преимуществом инкапсуляции является ограничение последствий изменений, вносимых в систему. Применим принцип инкапсуляции к банковской системе. Допустим, управление банка постановило, что если клиент имеет кредитный счет, то этот кредит может быть использован как овердрафт на его счете «до востребования». В неинкапсулированной системе модификация начинается с узконаправленного анализа изменений, которые необходимо будет внести в систему. Как правило, неизвестно, где в системе находятся все обращения к функции снятия со счета, поэтому приходится искать их везде. После того, как они найдены, нужно осуществить в них некоторые изменения, чтобы реализовать новые требования. Если работать тщательно, то, вероятно, можно будет обнаружить около 80% случаев использования данной функции. В инкапсулированной системе не требуется осуществлять такой детальный анализ. Достаточно посмотреть на модель системы и определить, где инкапсулировано соответствующее поведение (действие снятия со счета). После его локализации остается внести требуемые поправки один раз только в этом объекте, и задача выполнена.

Инкапсуляция подобна понятию сокрытия информации (information hiding). Это возможность скрывать многочисленные детали объекта от внешнего мира. Внешний мир объекта — это все, что находится вне его, включая остальную часть системы. Сокрытие информации предоставляет то же преимущество, что и инкапсуляция, — гибкость.

Модульность — это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (модулей) (см. подразд. 2.1). Модульность снижает сложность системы, позволяя выполнять независимую разработку отдельных модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Иерархия — это ранжированная или упорядоченная система абстракций, расположение их по уровням. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу). Примерами

иерархии классов являются простое и множественное наследование (один класс использует структурную или функциональную часть соответственно одного или нескольких других классов), а иерархии объектов — агрегация.

2.4.2.

ОСНОВНЫЕ ЭЛЕМЕНТЫ ОБЪЕКТНОЙ МОДЕЛИ

К основным понятиям объектно-ориентированного подхода (элементам объектной модели) относятся:

- объект;
- класс;
- атрибут;
- операция;
- полиморфизм (интерфейс);
- компонент;
- связи.

Объект определяется как осязаемая сущность (tangible entity) — предмет или явление (процесс), имеющие четко определяемое поведение. Объект может представлять собой абстракцию некоторой сущности предметной области (объект реального мира) или программной системы (архитектурный объект). Любой объект обладает *состоянием (state), поведением (behavior) и индивидуальностью (identity)*.

Состояние объекта — одно из возможных условий, в которых он может существовать, оно изменяется со временем. Состояние объекта характеризуется перечнем всех возможных (статических) свойств данного объекта и текущими значениями (динамическими) каждого из этих свойств. Состояние объекта определяется значениями его свойств (*атрибутов*) и связями с другими объектами.

Поведение определяет действия объекта и его реакцию на запросы от других объектов. Поведение характеризует воздействие объекта на другие объекты, изменяющее их состояние. Иначе говоря, поведение объекта полностью определяется его действиями. Поведение представляется с помощью набора **сообщений**, воспринимаемых объектом (**операций**, которые может выполнять объект).

Каждый объект обладает уникальной **индивидуальностью**. Индивидуальность — это свойства объекта, отличающие его от всех других объектов.

Структура и поведение схожих объектов определяют общий для них класс. Термины «экземпляр класса» и «объект» являются эквивалентными.

Графическое представление объектов в языке моделирования UML (который будет рассматриваться в подразд. 2.5) показано на рис. 2.37.



Рис. 2.37. Графическое представление объектов

Класс — это множество объектов, связанных общностью свойств, поведения, связей и семантики. Класс инкапсулирует (объединяет) в себе данные (атрибуты) и поведение (операции). Класс является абстрактным определением объекта и служит в качестве шаблона для создания объектов. Графическое представление класса в языке UML показано на рис. 2.38. Класс изображается в виде прямоугольника, разделенного на три части. В первой содержится имя

класса, во второй — его атрибуты. В последней части содержатся операции класса, отражающие его поведение (действия, выполняемые классом).

Любой объект является экземпляром (instance) класса. Определение классов и объектов — одна из самых сложных задач объектно-ориентированного проектирования.

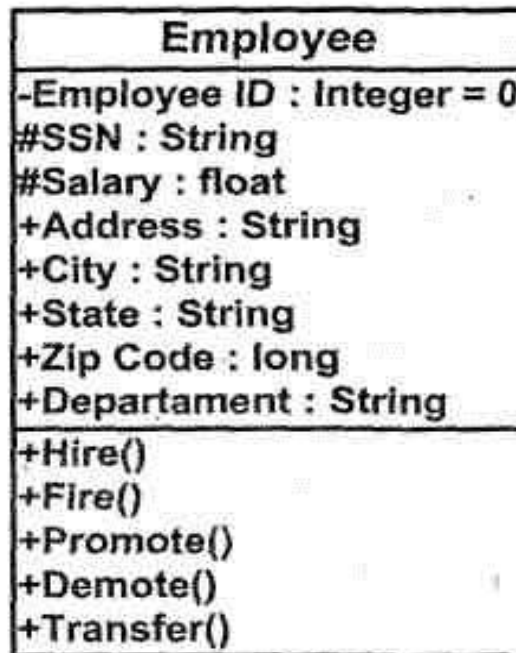


Рис. 2.38. Графическое представление класса

Атрибут — поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства.

Атрибут — это элемент информации, связанный с классом. Например, у класса Company (Компания) могут быть атрибуты Name (Название), Address (Адрес) и NumberOfEmployees (Число служащих).

Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим может понадобиться указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута (attribute visibility).

У атрибута можно определить три возможных значения этого параметра. Рассмотрим каждый из них в контексте примера (см. рис. 2.38). Пусть имеется класс Employee с атрибутом Address и класс Company:

Public (общий, открытый). Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В таком случае класс Company может изменить значение атрибута Address класса Employee. В соответствии с нотацией UML общему атрибуту предшествует знак «+».

Private (закрытый, секретный). Соответствующий атрибут не виден никаким другим классом. Класс Employee будет знать значение атрибута Address и сможет изменять его, но класс Company не сможет его ни увидеть, ни редактировать. Если это понадобится, он должен попросить класс Employee просмотреть или изменить значение этого атрибута, что обычно делается с помощью общих операций. Закрытый атрибут обозначается знаком «-» в соответствии с нотацией UML.

Protected (защищенный). Такой атрибут доступен только самому классу и его потомкам в иерархии наследования. Допустим, имеется два различных типа сотрудников — с почасовой оплатой и на окладе. Таким образом, потомками класса Employee будут два класса — HourlyEmp и SalariedEmp. Защищенный атрибут Address можно просмотреть или изменить из классов Employee, HourlyEmp и SalariedEmp, но не из класса Company. Нотация UML для защищенного атрибута — это знак «#».

В общем случае атрибуты рекомендуется делать закрытыми или защищенными. При этом удастся избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам этот атрибут.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется **операцией**. **Операция** — это реализация услуги, которую можно запросить у любого объекта данного класса.

Операции отражают поведение объекта. Операция-запрос не изменяет состояния объекта. Операция-команда может изменить состояние объекта. Результат операции зависит от текущего состояния объекта.

Как правило, в объектных и объектно-ориентированных языках программирования операции, выполняемые над данным объектом, называются **методами** и являются составной частью определения класса.

Операции реализуют связанное с классом поведение (иначе говоря, реализуют **обязанности класса** — *responsibilities*). В широком смысле обязанности класса делятся на две категории.

Знание (определяется атрибутами класса):

- наличие информации о данных или вычисляемых величинах;
- наличие информации о связанных объектах. Действие (определяется операциями класса):
- выполнение некоторых действий самим объектом;
- инициация действий других объектов;
- координация действий других объектов.

Операция включает три части — имя, параметры и тип возвращаемого значения. Параметры — это аргументы, получаемые операцией «на входе». Тип возвращаемого значения относится к результату действия операции.

В языке UML операции имеют следующую нотацию:

Имя Операции (аргумент1: тип данных аргумента1, аргумент2: тип данных аргумента2,...): тип возвращаемого значения.

Существуют четыре различных типа операций.

- **Операции реализации (implementor operations)** реализуют некоторые функции (процедуры). Такие операции выявляются путем анализа диаграмм взаимодействия UML.
- **Операции управления (manager operations)** управляют созданием и уничтожением объектов. В эту категорию попадают конструкторы и деструкторы классов.

- Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны просматривать или изменять их значения. Для этого существуют *операции доступа (access operations)*.
- *Вспомогательными (helper operations)* называются такие операции класса, которые необходимы ему для выполнения его обязанностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса.

Понятие *полиморфизма* может быть интерпретировано, как способность класса принадлежать более чем одному типу.

Полиморфизм — это способность скрывать множество различных реализаций под единственным общим интерфейсом.

Интерфейс — совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции имеют открытую видимость

Полиморфизм тесно связан с наследованием. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций).

Объектно-ориентированная система изначально строится с учетом ее эволюции. Наследование и полиморфизм обеспечивают возможность определения новой функциональности классов с помощью создания производных классов — потомков базовых классов. Потомки наследуют характеристики родительских классов без изменения их первоначального описания и добавляют при необходимости собственные структуры данных и методы. Определение производных классов, при котором задаются только различия или уточнения, в огромной степени экономит время и усилия при производстве и использовании спецификаций и программного кода.

Компонент — относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры. Компонент представляет собой физическую реализацию проектной абстракции и может быть:

- компонентом исходного кода;
- компонентом времени выполнения (run time);
- исполняемым компонентом.

Компонент обеспечивает физическую реализацию набора интерфейсов.

Между элементами объектной модели существуют различные виды связей. К основным типам связей относятся связи ассоциации, зависимости и обобщения.

Ассоциация (association) — это семантическая связь между классами. Ее изображают на диаграмме классов в виде обыкновенной линии (рис. 2.39). Ассоциация отражает структурные связи между объектами различных классов.

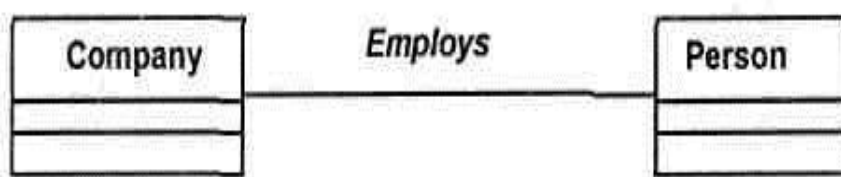


Рис. 2.39. Ассоциация

Агрегация (aggregation) представляет собой форму ассоциации — более сильный тип связи между целым (составным) объектом и его частями (компонентными объектами).

Существуют четыре возможных семантики для агрегации:

- 1) агрегация типа «Безраздельно обладает»;
- 2) агрегация типа «Обладает»;
- 3) агрегация типа «Включает»;
- 4) агрегация типа «Участник».

Агрегация типа «Безраздельно обладает» устанавливает следующее:

- между компонентными объектами и их составными объектами установлено отношение *зависимости по существованию* (следовательно, удаление составного объекта распространяется вниз по иерархии отношения, так что связанные компонентные объекты также удаляются);

- агрегация *транзитивна* (если объект C_1 является частью объекта B_1 , а B_1 — частью A_1 , тогда C_1 является частью A_1);
- агрегация *асимметрична (нерефлексивна)* (если объект B_1 является частью объекта A_1 , то объект A_1 не является частью B_1);
- агрегация *стационарна* (если объект B_1 является частью объекта A_1 , то он не может быть частью объекта A_i ($i \neq 1$)).

Агрегация типа «Обладает» поддерживает первые три свойства агрегации «Безраздельно обладает», к которым относятся:

- зависимость по существованию;
- транзитивность;
- асимметричность.

Агрегация типа «Включает» слабее, чем агрегация типа «Обладает», она поддерживает транзитивность и асимметричность.

Агрегация типа «Участник» обладает свойством целенаправленного группирования независимых объектов — группирования, при котором не делается предположений относительно свойства зависимости по существованию, транзитивности, асимметричности или стационарности. Компонентный объект в агрегации типа «Участник» может одновременно принадлежать более чем одному составному объекту.

Язык UML обеспечивает ограниченную поддержку агрегации. Сильная форма агрегации является в UML *композицией*. В композиции составной объект может физически содержать компонентные объекты. Компонентный объект может принадлежать только одному составному объекту. Композиция языка UML в большей или меньшей степени соответствует агрегациям типа «Безраздельно обладает» и «Обладает».

Слабая форма агрегации в UML называется просто *агрегацией*. При этом составной объект физически не содержит компонентный объект. Один компонентный объект может обладать несколькими связями ассоциации или агрегации. Иначе говоря, агрегация в языке UML соответствует агрегациям типа «Включает» и «Участник».

Агрегация изображается линией между классами с ромбом на стороне целого объекта (рис. 2.40). Сплошной ромб представляет композицию (рис. 2.41).

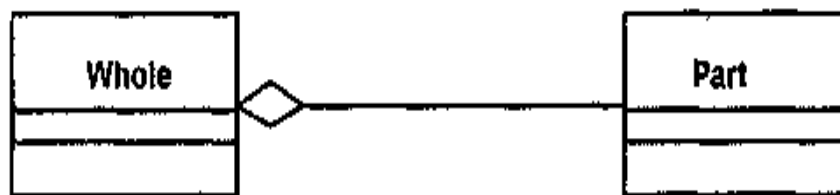


Рис. 2.40 Агрегация



Рис 2.41. Композиция

Хотя связи ассоциации и агрегации двунаправленные по умолчанию, часто накладываются ограничения на направление навигации (только в одном направлении). Если введено ограничение по направлению, то добавляется стрелка на конце связи. Направление ассоциации можно определить, изучая сообщения между классами. Если все сообщения на них отправляются только одним классом и принимаются только другим классом, но не наоборот, между этими классами имеет место однонаправленная связь. Если хотя бы одно сообщение отправляется в обратную сторону, ассоциация должна быть двунаправленной.

Ассоциации могут быть рефлексивными. Рефлексивная ассоциация предполагает, что один экземпляр класса взаимодействует с другими экземплярами этого же класса.

Связи можно уточнить с помощью имен связей или ролевых имен. Имя связи — это обычно глагол или глагольная фраза, описывающая, зачем она нужна. Например, между классом *Person* (человек) и классом *Company* (компания) может существовать ассоциация. Если человек является сотрудником компании, ассоциацию можно назвать «employs» (нанимает) (см. рис. 2.39).

Имена у связей определять не обязательно. Обычно это делают, если причина создания связи не очевидна. Имя показывают около линии соответствующей связи.

Для уточнения роли, которую играет каждый класс в связях ассоциации или агрегации, применяют ролевые имена (рис. 2.42). Возвращаясь к примеру с классами Person и Company, можно сказать, что класс Person играет роль сотрудника класса Company. Ролевые имена — это обычно имена существительные или основанные на них фразы, их показывают на диаграмме рядом с классом, играющим соответствующую роль. Как правило, пользуются или ролевым именем, или именем связи, но не обоими сразу. Как и имена связей, ролевые имена не обязательны, их дают, только если смысл связи не очевиден.

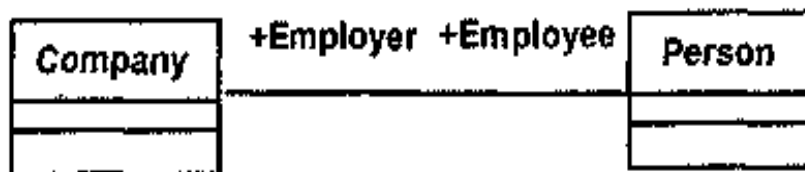


Рис 2.42. Ролевые имена

Мощность (multiplicity) показывает, как много объектов участвует в связи. Мощность — это число объектов одного класса, связанных с одним объектом другого класса. Понятие мощности связи в объектной модели аналогично понятиям мощности и класса принадлежности связи в модели «сущность-связь» (с точностью до расположения показателя мощности на диаграмме). Для каждой связи можно обозначить два показателя мощности — по одному на каждом конце связи.

В языке UML приняты следующие нотации для обозначения мощности.

Значения мощности

Мощность	Значение
*	Много
0	Ноль
1	Один
0..*	Ноль или больше
1..*	Один или больше

0..1 Нуль или один
1..1 Ровно один

Например, при разработке системы регистрации курсов в университете можно определить классы *Course* (учебный курс) и *Student* (студент). Между ними установлена связь, означающая посещение курсов студентами. Если один студент может посещать от нуля до четырех курсов, а на одном курсе могут заниматься от 10 до 20 студентов, то на диаграмме классов это можно изобразить следующим образом (рис. 2.43).

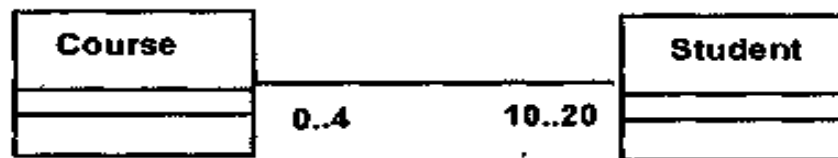


Рис 2.43. Мощность связи

Частным случаем ассоциации является *ассоциация-класс* (Association class), которая обладает как свойствами класса, так и свойствами ассоциации. Ассоциация-класс — это место, где хранятся относящиеся к ассоциации атрибуты и операции. Экземплярами ассоциации-класса являются связи, у которых есть не только ссылки на объекты, но и значения атрибутов. Ассоциация-класс подобна связи с атрибутами в модели «сущность-связь».

Допустим, что имеются два класса, *Student* и *Course*, и возникла необходимость добавить атрибут *Grade* (оценка). В таком случае возникает вопрос, в какой класс надо его добавить. Если поместить его внутри класса *Student*, то придется вводить его для каждого посещаемого студентом курса, что слишком сильно увеличит размер этого класса. Если же поместить его внутри класса *Course*, то придется задавать его для каждого посещающего этот курс студента.

Чтобы решить эту проблему, можно создать ассоциацию-класс. В этот класс следует поместить атрибут *Grade*, относящийся к связи между курсом и студентом. Нотация UML для ассоциации-класса представлена на рис. 2.44.

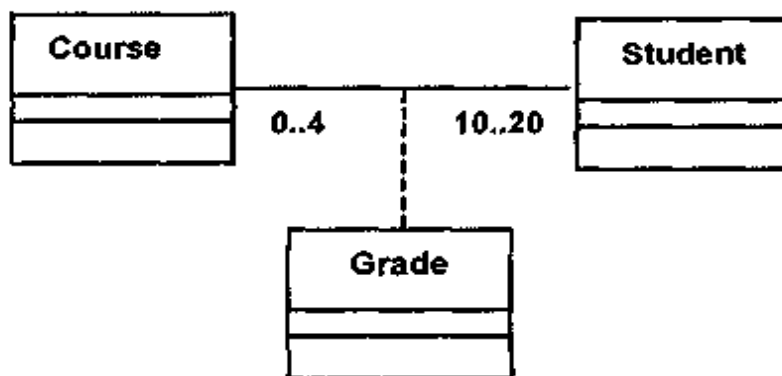


Рис 2.44. Ассоциация-класс

Ассоциация-класс определяет дополнительное ограничение, согласно которому двум участвующим в ассоциации объектам может соответствовать только один экземпляр ассоциации-класса. Диаграмма на рис. 2.44 не допускает, чтобы студент мог получать по курсу более чем одну оценку. Если необходимо, чтобы такое допускалось, то ассоциацию-класс Grade следует преобразовать в обычный класс, связанный ассоциациями с классами Student и Course.

Зависимость (dependency) — связь между двумя элементами модели, при которой изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе. Зависимость — слабая форма связи между клиентом и сервером (клиент зависит от сервера и не имеет знаний о сервере). Зависимость изображается пунктирной линией, направленной от клиента к серверу (рис. 2.45).

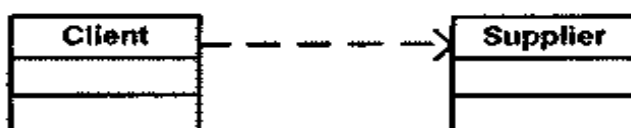


Рис 2.45. Зависимость

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Причины для зависимостей могут быть самыми разными: один класс посылает сообщение другому; один класс включает часть данных другого клас-

са; один класс использует другой в качестве параметра операции. Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может утратить свою силу.

Обобщение (generalization) — связь «тип-подтип» реализует механизм наследования (inheritance). Большинство объектно-ориентированных языков непосредственно поддерживают концепцию наследования. Она позволяет одному классу наследовать все атрибуты, операции и связи другого. В языке UML связи наследования называют обобщениями и изображают в виде стрелок от класса-потомка к классу-предку (рис. 2.46).

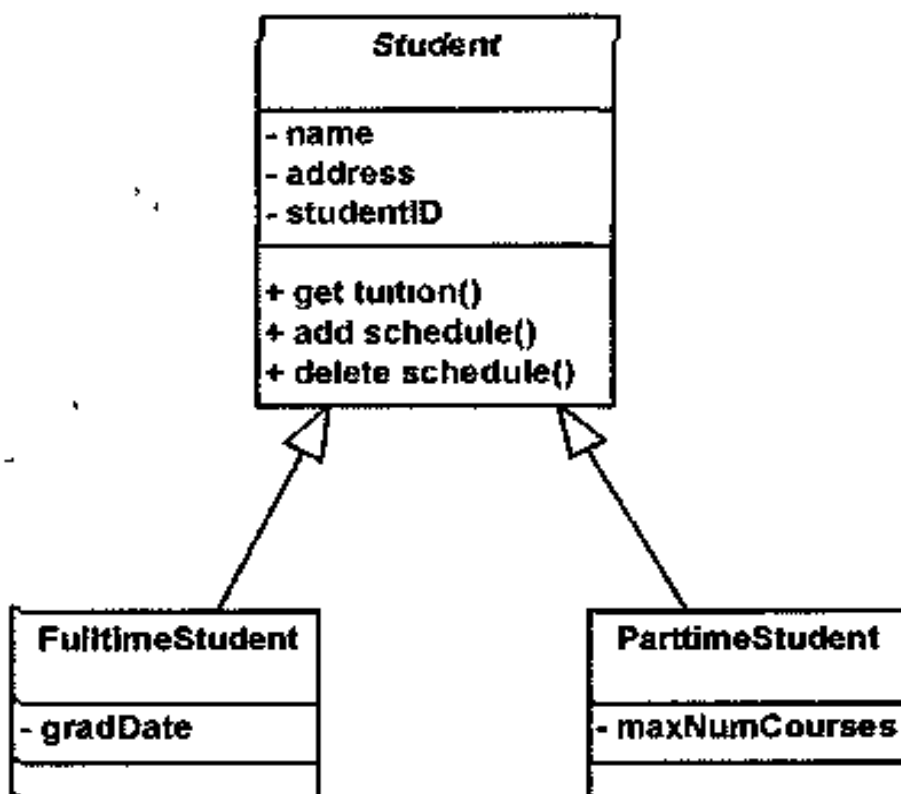


Рис 2.46. Обобщение

Общие атрибуты, операции и/или связи отображаются на верхнем уровне иерархии.

Помимо наследуемых, каждый подкласс имеет свои собственные уникальные атрибуты, операции и связи.

УНИФИЦИРОВАННЫЙ ЯЗЫК

МОДЕЛИРОВАНИЯ UML

Унифицированный язык моделирования UML¹ (Unified Modeling Language) представляет собой язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов.

UML — это преемник того поколения методов объектно-ориентированного анализа и проектирования, которые появились в конце 1980-х и начале 1990-х годов. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо начали работу по объединению их методов Booch и OMT (Object Modeling Technique) под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же в 1995 г. к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Буча, Рамбо и Якобсона, однако дополняет их новыми возможностями. Главными в разработке UML были следующие цели:

- предоставить пользователям готовый к использованию выразительный язык визуального моделирования, позволяющий им разрабатывать осмысленные модели и обмениваться ими;
- предусмотреть механизмы расширяемости и специализации для расширения базовых концепций;
- обеспечить независимость от конкретных языков программирования и процессов разработки;

¹ Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. — М.: Мир, 1999.

- обеспечить формальную основу для понимания этого языка моделирования (язык должен быть одновременно точным и доступным для понимания, без лишнего формализма);
- стимулировать рост рынка объектно-ориентированных инструментальных средств;
- интегрировать лучший практический опыт.

UML находится в процессе стандартизации, проводимом OMG (Object Management Group) — организацией по стандартизации в области объектно-ориентированных методов и технологий, в настоящее время принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО. UML принят на вооружение почти всеми крупнейшими компаниями — производителями ПО (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, почти все мировые производители CASE-средств, помимо IBM Rational

Software, поддерживают UML в своих продуктах (Together (Borland), Paradigm Plus (Computer Associates), System Architect (Popkin Software), Microsoft Visual Modeler и др.). Полное описание UML можно найти на сайтах <http://www.omg.org> и <http://www.rational.com>.

Стандарт UML версии 1.1, принятый OMG в 1997 г., предлагает следующий набор диаграмм:

- Структурные (structural) модели:
 диаграммы классов (class diagrams) - для моделирования статической структуры классов системы и связей между ними; диаграммы компонентов (component diagrams) — для моделирования иерархии компонентов (подсистем) системы; диаграммы размещения (deployment diagrams) — для моделирования физической архитектуры системы.
- Модели поведения (behavioral):
 диаграммы вариантов использования (use case diagrams) — для моделирования бизнес-процессов и функциональных требований к создаваемой системе; диаграммы взаимодействия (interaction diagrams):

диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams) — для моделирования процесса обмена сообщениями между объектами;

диаграммы состояний (statechart diagrams) — для моделирования поведения объектов системы при переходе из одного состояния в другое;

диаграммы деятельности (activity diagrams) — для моделирования поведения системы в рамках различных вариантов использования, или потоков управления.

2.5.1.

ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Идея описания функциональных требований в виде вариантов использования (use case) была сформулирована в 1986 г. Иваром Якобсоном. Эта идея была признана конструктивной и получила широкое одобрение. Впоследствии наиболее значительный вклад в решение проблемы определения сущности вариантов использования и способов их описания внес Алистер Коберн¹.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой и отражает представление о поведении системы с точки зрения пользователя. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать, или целей, которые он преследует по отношению к разрабатываемой системе.

Действующее лицо (actor) — это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ.

¹ Коберн А. Современные методы описания функциональных требований к системам.: Пер. с англ. - М.: ЛОРИ, 2002

Несмотря на то что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы.

Действующие лица делятся на три основных типа — пользователи системы, другие системы, взаимодействующие с данной, и время. Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Для наглядного представления вариантов использования применяются диаграммы вариантов использования. На рис. 2.47 показан пример такой диаграммы для банковской системы.

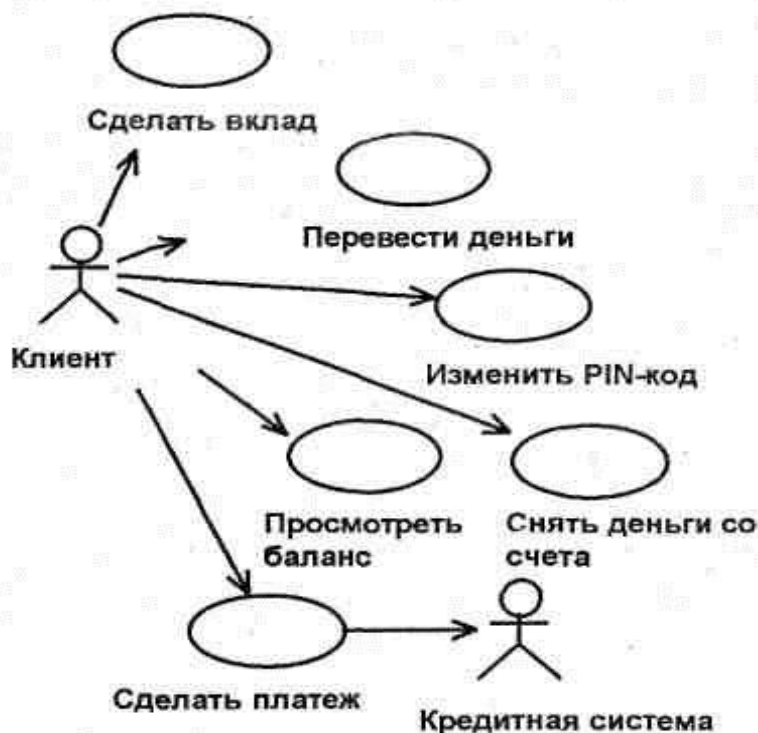


Рис. 2.47. Пример диаграммы вариантов использования

На данной диаграмме человеческие фигурки обозначают действующих лиц, овалы — варианты использования, а линии и стрелки — различные связи между действующими лицами и вариантами использования. На этой диаграмме показаны два действующих лица: клиент и кредитная система. Существует также шесть основных действий, выполняемых моделируемой системой: пере-

вести деньги, сделать вклад, снять деньги со счета, просмотреть баланс, изменить PIN-код и сделать платеж.

На диаграмме вариантов использования показано взаимодействие между вариантами использования и действующими лицами. Она отражает функциональные требования к системе с точки зрения пользователя. Таким образом, варианты использования — это функции, выполняемые системой, а действующие лица — это заинтересованные лица (stakeholders) по отношению к создаваемой системе. Такие диаграммы показывают, какие действующие лица инициируют варианты использования. Из них также видно, когда действующее лицо получает информацию от варианта использования. Направленная от варианта использования к действующему лицу стрелка показывает, что вариант использования предоставляет некоторую информацию, используемую действующим лицом. В данном случае вариант использования «Сделать платеж» предоставляет Кредитной системе информацию об оплате по кредитной карточке.

Действующие лица могут играть различные роли по отношению к варианту использования. Они могут пользоваться его результатами сами непосредственно в нем участвовать. Значимость различных ролей действующего лица зависит от того, каким образом используются его связи.

Цель построения диаграмм вариантов использования — документирование функциональных требований к системе в самом общем виде, поэтому они должны быть предельно простыми. При построении диаграмм вариантов использования нужно придерживаться следующих правил:

- Не моделируйте связи между действующими лицами. По определению действующие лица находятся вне сферы действия системы. Это означает, что связи между ними также не относятся к ее компетенции.
- Не соединяйте стрелкой два варианта использования непосредственно. Диаграммы данного типа описывают только сами варианты использования, а не порядок их выполнения. Для отображения порядка выполнения вариантов использования применяют диаграммы деятельности.

- Каждый вариант использования должен быть инициирован действующим лицом. Это означает, что всегда должна быть стрелка, начинающаяся на действующем лице и заканчивающаяся на варианте использования.

Хорошим источником для идентификации вариантов использования служат внешние события. Следует начать с перечисления всех событий, происходящих во внешнем мире, на которые система должна каким-то образом реагировать. Какое-либо конкретное событие может повлечь за собой реакцию системы, не требующую вмешательства пользователей, или, наоборот, вызвать чисто пользовательскую реакцию. Идентификация событий, на которые необходимо реагировать, помогает идентифицировать варианты использования.

Диаграмма вариантов использования является самым общим представлением функциональных требований к системе. Однако моделирование вариантов использования не сводится только к рисованию диаграмм. Для последующего проектирования системы требуются более конкретные детали. Эти детали описываются в документе, называемом **«сценарий варианта использования»** или **«поток событий» (flow of events)**. Целью потока событий является подробное документирование процесса взаимодействия действующего лица с системой, реализуемого в рамках варианта использования. В потоке событий должно быть описано все, что служит удовлетворению запросов действующих лиц.

Хотя поток событий и описывается подробно, он также не должен зависеть от реализации. Цель - описать, что будет делать система, а не как она будет делать это. Обычно описание потока событий включает следующие разделы:

- краткое описание;
- предусловия (pre-conditions);
- основной поток событий;
- альтернативные потоки событий;
- постусловия (post-conditions);
- расширения (extensions).

Последовательно рассмотрим эти составные части.

Краткое описание. Каждый вариант использования должен иметь краткое описание того, что в нем происходит. Например, вариант использования «Перевести деньги» может содержать следующее описание.

Вариант использования «Перевести деньги» позволяет клиенту или служащему банка переводить деньги с одного счета до востребования или сберегательного счета на другой.

Предусловия. Предусловия варианта использования - это такие условия, которые должны быть выполнены, прежде чем вариант использования начнет выполняться сам. Например, таким условием может быть выполнение другого варианта использования или наличие у пользователя прав доступа, требуемых для начала работы. Не у всех вариантов использования бывают предусловия. Ранее упоминалось, что диаграммы вариантов использования не должны отражать порядок их выполнения. Такую информацию можно описать с помощью предусловий. Например, предусловием одного варианта использования может быть то, что в это время должен выполняться другой.

Основной и альтернативный потоки событий. Конкретные детали вариантов использования описываются в основном в альтернативных потоках событий. Поток событий поэтапно описывает, что должно происходить во время выполнения заложенной в варианты использования функциональности. Поток событий уделяет внимание тому, *что* будет делать система, а не *как* она будет делать это, причем описывает все это с точки зрения пользователя. Основной поток событий описывает нормальный ход событий (при отсутствии ошибок), и при наличии нескольких возможных вариантов хода событий может разветвляться на подчиненные потоки (subflow). Альтернативные потоки описывают отклонения от нормального хода событий (ошибочные ситуации) и их обработку. Например, потоки событий варианта использования «Снять деньги со счета» могут выглядеть следующим образом:

Основной поток событий

1. Вариант использования начинается, когда клиент вставляет свою карточку в банкомат.
2. Банкомат выводит приветствие и предлагает клиенту ввести свой персональный PIN-код.
3. Клиент вводит PIN-код.
4. Банкомат подтверждает введенный код.
5. Банкомат выводит список доступных действий: сделать вклад, снять деньги со счета, перевести деньги
6. Клиент выбирает пункт «Снять деньги со счета».
7. Банкомат запрашивает, сколько денег надо снять.
8. Клиент вводит требуемую сумму.
9. Банкомат определяет, имеется ли на счете достаточно денег.
10. Банкомат вычитает требуемую сумму из счета клиента.
11. Банкомат выдает клиенту требуемую сумму наличными.
12. Банкомат возвращает клиенту его карточку.
13. Банкомат печатает чек для клиента.
14. Вариант использования завершается.

Альтернативный поток событий 1. Ввод неправильного PIN-кода.

- 4a1. Банкомат информирует клиента, что код введен неправильно.
- 4a2. Банкомат возвращает клиенту его карточку.
- 4a3. Вариант использования завершается.

Альтернативный поток событий 2. Недостаточно денег на счете.

- 9a1. Банкомат информирует клиента, что денег на его счете недостаточно.
- 9a2. Банкомат возвращает клиенту его карточку.
- 9a3. Вариант использования завершается.

Альтернативный поток событий 3. Ошибка в подтверждении запрашиваемой суммы.

9б1. Банкомат сообщает пользователю, что при подтверждении запрашиваемой суммы произошла ошибка, и дает ему номер телефона службы поддержки клиентов банка.

9б2. Банкомат заносит сведения об ошибке в журнал ошибок. Каждая запись содержит дату и время ошибки, имя клиента, номер его счета и код ошибки.

9б3. Банкомат возвращает клиенту его карточку.

9б4. Вариант использования завершается.

Как видно из приведенного примера, хорошо написанный поток событий должен легко читаться и состоять из предложений, написанных в единой грамматической форме. На обучение его чтению не должно уходить больше нескольких минут. При написании основного потока событий нужно придерживаться следующих правил:

- использовать простые предложения;
- явно указывать в каждом пункте, кто выполняет действие — действующее лицо или система;
- не показывать слишком незначительные действия;
- не показывать детальные действия пользователя в процессе работы с пользовательским интерфейсом;
- не рассматривать возможные ошибочные ситуации (использовать действия «подтвердить», а не «проверить»).

При выявлении альтернативных потоков событий нужно в первую очередь обратить внимание на ситуации, связанные с:

- некорректными действиями пользователя (например, ввод неверного пароля);
- бездействием действующего лица (например, истечением времени ожидания пароля);

- внутренними ошибками в разрабатываемой системе, которые должны быть обнаружены и обработаны в обычном порядке (например, заблокирован автомат для выдачи наличных);
- критически важными недостатками в производительности системы (например, время реакции не укладывается в 5 секунд).

Постусловия. Постусловиями называются такие условия, которые всегда должны быть выполнены после завершения варианта использования. Например, в конце варианта использования можно пометить флажком какой-нибудь переключатель. Информация такого типа входит в состав постусловий. Как и для предусловий, с помощью постусловий можно вводить информацию о порядке выполнения вариантов использования системы. Если, например, после одного из вариантов использования должен всегда выполняться другой, это можно описать как постусловие. Такие условия имеются не у каждого варианта использования.

Расширения. Этот пункт присутствует, если в основном потоке событий имеют место относительно редко встречающиеся ситуации (частные случаи). Описание таких ситуаций выносится в данный пункт.

В диаграммах вариантов использования может присутствовать несколько типов связей. Это связи коммуникации (communication), включения (include), расширения (extend) и обобщения (generalization).

Связь коммуникации — это связь между вариантом использования и действующим лицом, она изображается с помощью однонаправленной ассоциации (линии со стрелкой). Направление стрелки позволяет понять, кто инициирует коммуникацию.

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы (часть потока событий), который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность. В примере с банковской системой варианты использования «Снять деньги со счета» и «Сделать вклад» должны аутентифицировать клиента и его PIN-код перед тем, как допу-

стить осуществление самой транзакции. Вместо того чтобы подробно описывать процесс аутентификации для каждого из вариантов использования, можно поместить эту функциональность в свой собственный вариант использования под названием «Аутентифицировать клиента».

Связь расширения применяется при наличии изменений в нормальном поведении системы (описанных в пункте «Расширения»), которые также выносятся в отдельный вариант использования.

Связи включения и расширения изображаются в виде зависимостей, как показано на рис. 2.48.



Рис. 2.48. Связи включения и расширения

С помощью связи обобщения показывают, что у нескольких действующих лиц имеются общие черты и различия. Например, у служащих организации имеются как общие свойства, так и разные способы оплаты труда (рис. 2.49).

Нет необходимости всегда создавать связи этого типа. В общем случае они нужны, если отличия в поведении действующего лица одного типа от поведения другого затрагивают функции системы. Если оба подтипа используют одни и те же варианты использования, показывать обобщение действующего лица не требуется.

Варианты использования являются необходимым средством на стадии формирования требований к ПО. Каждый вариант использования — это потен-

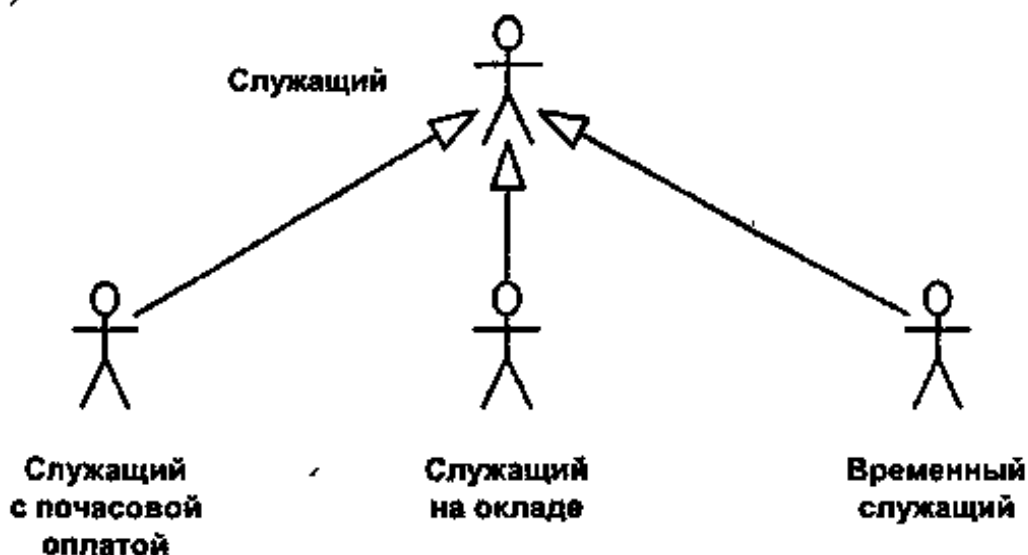


Рис. 2.49. Обобщение действующего лица

циальное требование к системе, и пока оно не выявлено, невозможно запланировать его реализацию.

Различные разработчики подходят к описанию вариантов использования с разной степенью детализации. Например, Ивар Якобсон утверждал, что для проекта с трудоемкостью 10 человеко-лет количество вариантов использования может составлять около 20 (не считая связей «включения» и «расширения»). Следует предпочитать небольшие и детализированные варианты использования, поскольку они облегчают составление и реализацию согласованного плана проекта.

Достоинства модели вариантов использования заключаются в том, что она:

- определяет пользователей и границы системы;
- определяет системный интерфейс;
- удобна для общения пользователей с разработчиками;
- используется для написания тестов;
- является основой для написания пользовательской документации;
- хорошо вписывается в любые методы проектирования (как объектно-ориентированные, так и структурные).

ДИАГРАММЫ ВЗАИМОДЕЙСТВИЯ

Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов (в рамках варианта использования или некоторой операции класса).

Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного потока событий варианта использования. На такой диаграмме отображается ряд объектов и те сообщения, которыми они обмениваются между собой.

Сообщение (*message*) — средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

Информационное (*informative*) сообщение — сообщение, снабжающее объект-получатель некоторой информацией для обновления его состояния.

Сообщение-запрос (*interrogative*) — сообщение, запрашивающее выдачу некоторой информации об объекте-получателе.

Императивное (*imperative*) сообщение — сообщение, запрашивающее у объекта-получателя выполнение некоторых действий.

Существуют два вида диаграмм взаимодействия: диаграммы последовательности и кооперативные диаграммы.

Диаграммы последовательности отражают временную последовательность событий, происходящих в рамках варианта использования. Например, вариант использования «Снять деньги со счета» предусматривает несколько возможных потоков событий, таких как снятие денег, попытка снять деньги, не имея их достаточного количества на счете, попытка снять деньги по неправильному PIN-коду и некоторых других. Нормальный сценарий (основной поток событий) снятия некоторой суммы денег со счета показан на рис. 2.50.

Все действующие лица показаны в верхней части диаграммы; и приведенном примере изображено действующее лицо Клиент (Customer).

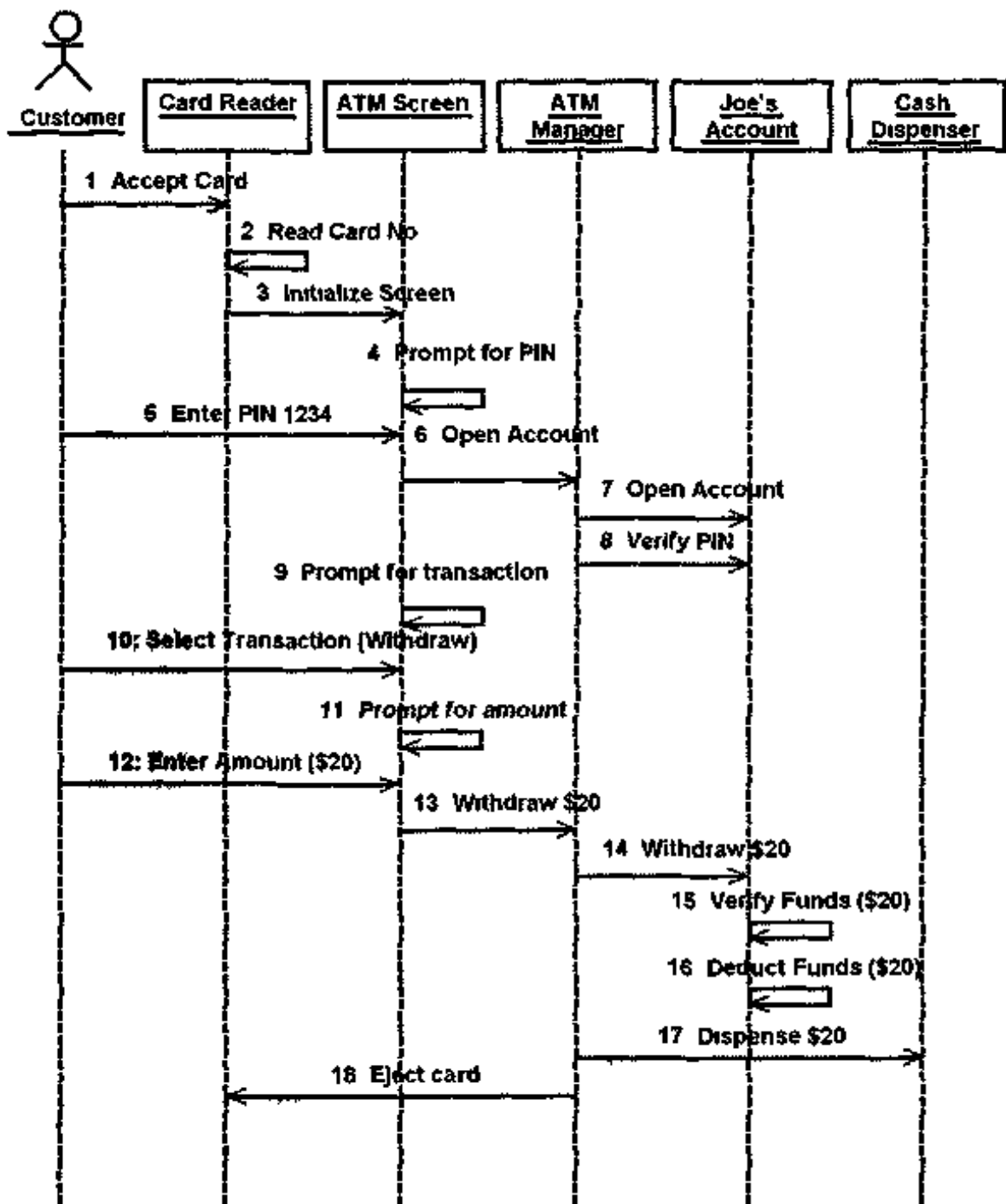


Рис. 2.50. Диаграмма последовательности

Объекты, требуемые системе для выполнения варианта использования «Снять деньги со счета», также представлены в верхней части диаграммы. Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций.

На диаграмме последовательности объект изображается в виде прямоугольника на вершине пунктирной вертикальной линии. Эта вертикальная линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия.

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое сообщение помечается, как минимум, именем сообщения; при желании можно добавить также аргументы и некоторую управляющую информацию, и, кроме того, можно показать самоделегирование (self-delegation) — сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни.

Один из способов первоначального обнаружения некоторых объектов - это изучение имен существительных в потоке событий. Поток событий для варианта использования «Снять деньги со счета» говорит о человеке, снимающем некоторую сумму денег со счета с помощью банкомата.

Не все объекты, показанные на диаграмме, явно присутствуют в потоке событий. Там, например, может не быть форм для заполнения, но их необходимо показать на диаграмме, чтобы позволить действующему лицу ввести новую информацию в систему или просмотреть ее. В потоке событий, скорее всего, также не будет и управляющих объектов (control objects). Эти объекты управляют последовательностью событий в варианте использования.

Вторым видом диаграммы взаимодействия является *кооперативная диаграмма* (рис. 2.51).

Подобно диаграммам последовательности кооперативные диаграммы отображают поток событий варианта использования. Диаграммы последовательности упорядочены по времени, а кооперативные диаграммы концентрируют внимание на связях между объектами. На рис. 2.51 приведена кооперативная диаграмма, описывающая, как клиент снимает деньги со счета. На ней представлена та же информация, которая была и на диаграмме последовательности, но кооперативная диаграмма по-другому описывает поток событий. Из

нее легче понять связи между объектами, однако труднее уяснить последовательность событий.

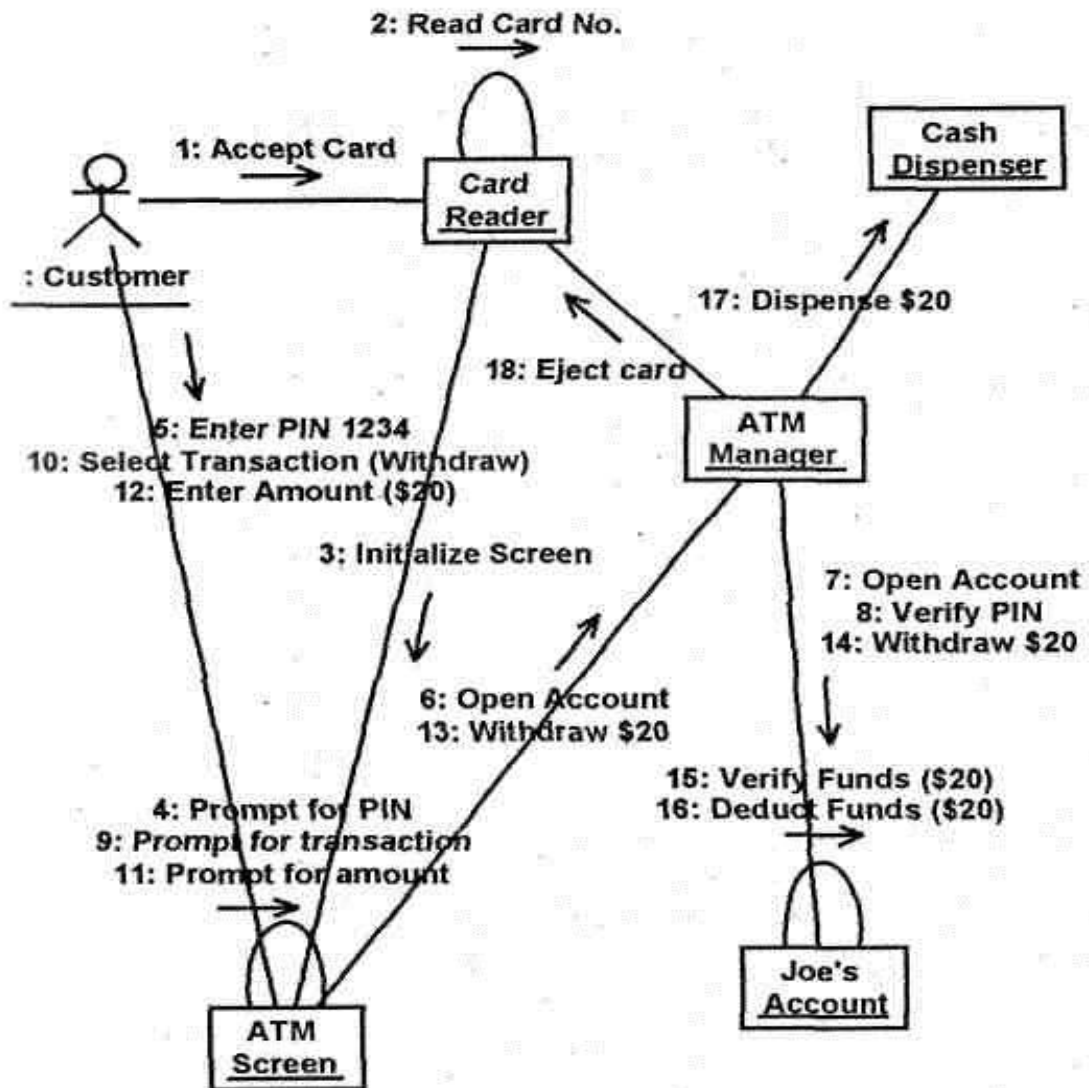


Рис. 2.51. Кооперативная диаграмма

По этой причине часто для какого-либо сценария создают диаграммы обоих типов. Хотя они служат одной и той же цели и содержат одну и ту же информацию, но представляют ее с различных точек зрения.

На кооперативной диаграмме так же, как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временная последовательность, однако, указывается путем нумерации сообщений.

ДИАГРАММЫ КЛАССОВ

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Диаграмма классов для варианта использования «Снять деньги со счета» показана на рис. 2.52.

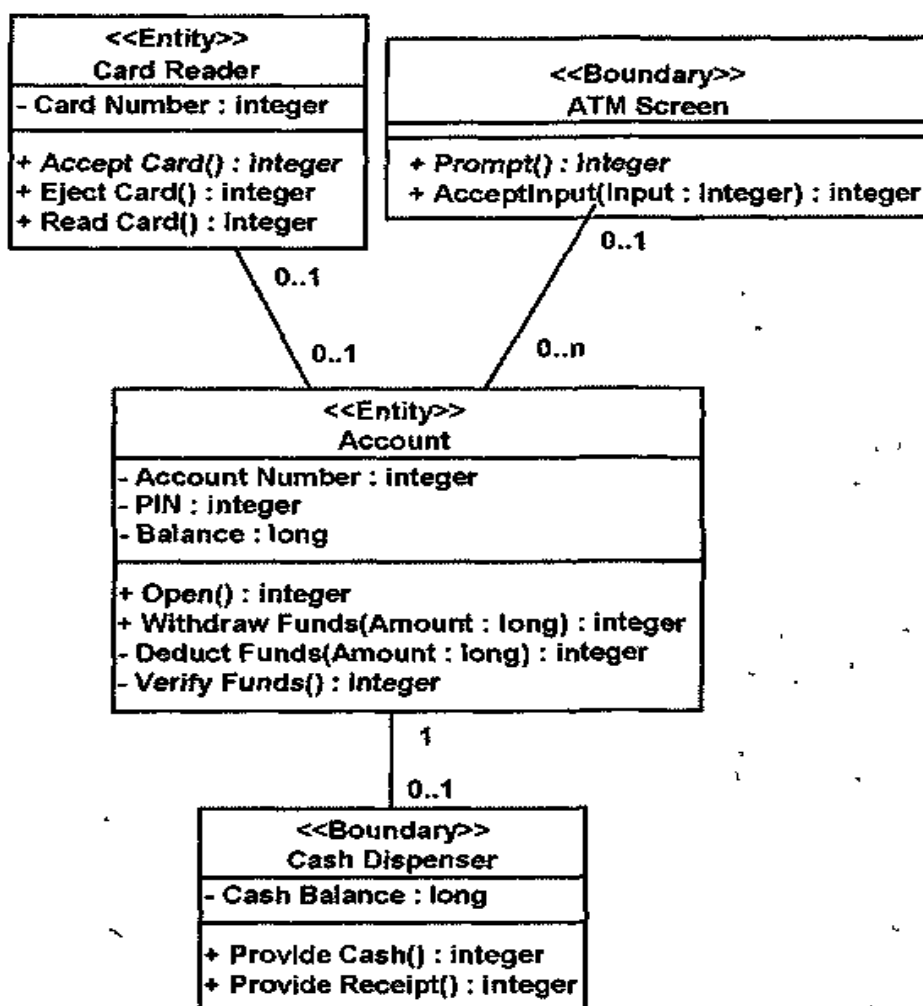


Рис. 2.52. Диаграмма классов для варианта использования «Снять деньги со счета»

На этой диаграмме присутствуют четыре класса: Card Reader (устройство для чтения карточек), Account (счет), ATM Screen (экран АТМ) и Cash Dispenser

(кассовый аппарат). Связывающие классы линии отражают взаимодействие между классами. Так, класс Account связан с классом ATM Screen, потому что они непосредственно сообщаются и взаимодействуют друг с другом. Класс Card Reader не связан с классом Cash Dispenser, поскольку они не сообщаются друг с другом непосредственно. В изображении классов присутствуют также стереотипы, которые будут рассматриваться в подразд. 2.5.8.

Для группировки классов, обладающих некоторой общностью, применяются пакеты. **Пакет** — общий механизм для организации элементов модели в группы. Пакет может включать другие элементы. Каждый элемент модели может входить только в один пакет. Пакет является средством организации модели в процессе разработки, повышения ее управляемости и читаемости, а также единицей управления конфигурацией.

Существуют несколько подходов к группировке классов. Во-первых, можно группировать их по стереотипу (типу класса). Например, один пакет содержит классы-сущности предметной области, другой — классы пользовательского интерфейса и т.д. Этот подход может быть полезен с точки зрения размещения системы в среде реализации.

Другой подход заключается в объединении классов по их функциональности. Например, в пакете Security (безопасность) содержатся все классы, отвечающие за безопасность приложения. В таком случае другие пакеты могут называться Employee Maintenance (Работа с сотрудниками), Reporting (Подготовка отчетов) и Error Handling (Обработка ошибок). Преимущество этого подхода заключается в возможности повторного использования.

Если между любыми двумя классами, находящимися в разных пакетах, существует некоторая зависимость, то имеет место зависимость и между этими двумя пакетами. Таким образом, **диаграмма пакетов** (рис. 2.53) представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости являются элементами диаграммы классов, т.е. диаграмма пакетов — это форма диаграммы классов. Диаграммы пакетов можно считать основным средством управления общей структурой системы.

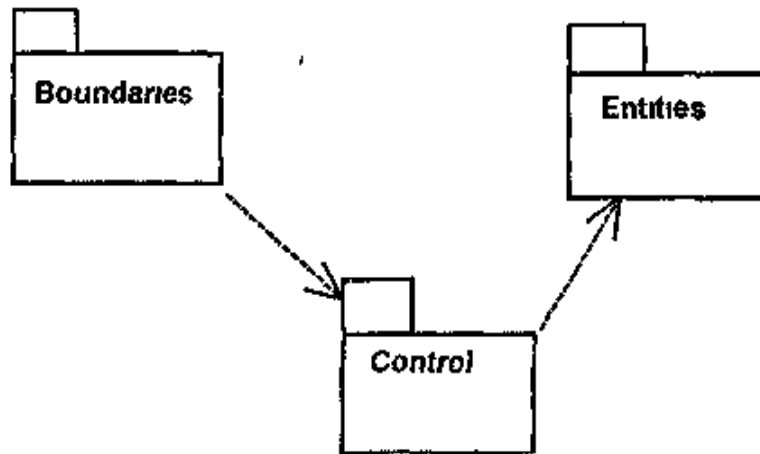


Рис. 2.53. Диаграмма пакетов

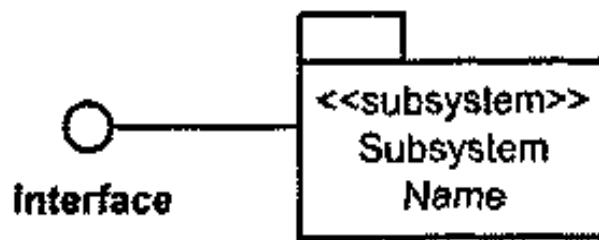


Рис. 2.54. Подсистема

Пакеты также используются для представления подсистем (модулей) системы (рис. 2.54). *Подсистема* — это комбинация пакета (поскольку она включает некоторое множество классов) и класса (поскольку она обладает поведением, т.е. реализует набор операций, которые определены в ее интерфейсах). Связь между подсистемой и интерфейсом называется *связью реализации*. Подсистема используется для представления компонента в процессе проектирования.

ДИАГРАММЫ СОСТОЯНИЙ

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий. Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой.

На рис. 2.55 приведен пример диаграммы состояний для банковского счета. Из данной диаграммы видно, в каких состояниях может существовать счет. Можно также видеть процесс перехода счета из одного состояния в другое. Например, если клиент требует закрыть открытый счет, он переходит в состояние «закрыт». Требование клиента называется событием (event), именно такие события и вызывают переход из одного состояния в другое.

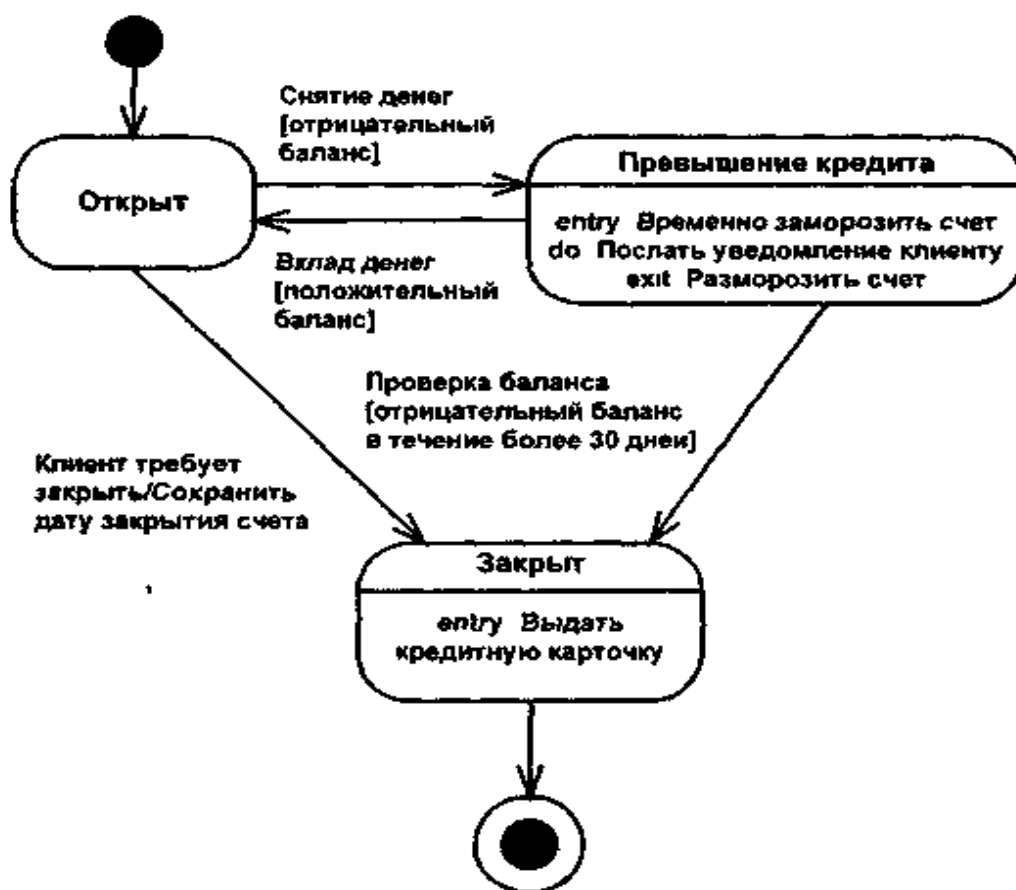


Рис. 2.55. Диаграмма состояний для класса Account

Если клиент снимает деньги с открытого счета, он может перейти в состояние «Превышение кредита». Это происходит, только если баланс по этому счету меньше нуля, что отражено условием [отрицательный баланс] на диаграмме. Заключенное в квадратных скобках ограничивающее условие (*guard condition*) определяет, когда может или не может произойти переход из одного состояния в другое.

На диаграмме имеются два специальных состояния — начальное (*start*) и конечное (*stop*). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть только одно начальное состояние, а конечных состояний может быть столько, сколько нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. На рис. 2.55 при превышении кредита клиенту посылается соответствующее сообщение. Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (*actions*).

С состоянием можно связывать данные пяти типов: деятельность, входное действие, выходное действие, событие и история состояния. Рассмотрим каждый из них в контексте диаграммы состояний для класса *Account* банковской системы.

Деятельность (*activity*) — это поведение, реализуемое объектом, пока он находится в данном состоянии. Например, когда счет находится в состоянии «Закрит», происходит возврат кредитной карточки пользователю. Деятельность — это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово *do* (выполнять) и двоеточие.

Входное действие (*entry action*) — это поведение, которое выполняется, когда объект переходит в данное состояние. Когда счет в банке переходит в со-

стояние «Превышение кредита», выполняется действие «Временно заморозить счет» независимо оттого, откуда объект перешел в это состояние. Таким образом, данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности входное действие рассматривается как непрерываемое.

Входное действие также показывают внутри состояния, ему предшествует слово *entry* (вход) и двоеточие.

Выходное действие (*exit action*) подобно входному, однако оно осуществляется как составная часть процесса выхода из данного состояния. Так, при выходе объекта Account из состояния «Превышение кредита» независимо оттого, куда он переходит, выполняется действие «Разморозить счет». Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым.

Выходное действие изображают внутри состояния, ему предшествует слово *exit* (выход) и двоеточие.

Переходом (*transition*) называется перемещение объекта из одного состояния в другое. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся на последующем.

Переходы могут быть рефлексивными. Объект может перейти и то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций, основными из которых являются события, ограждающие условия и действия.

Событие (*event*) вызывает переход из одного состояния в другое. Событие «Клиент требует закрыть» вызывает переход счета из открытого в закрытое состояние. События размещают на диаграмме вдоль линии перехода.

На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу, как в примере. Если нужно использовать операции, то событие «Клиент требует закрыть» можно было бы назвать Request-Closure().

Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществиться входным действиям, деятельности и выходным действиям.

Ограничивающие условия (*guard conditions*) определяют, когда переход может, а когда не может осуществиться. В примере событие «Вклад денег» переводит счет из состояния «Превышение кредита» в состояние «Открыт», но только если баланс будет больше нуля. В противном случае переход не осуществится.

Ограничивающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограничивающие условия задавать необязательно. Однако, если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия, чтобы понять, какой путь перехода будет автоматически выбран.

Действие может быть не только входным или выходным, но и частью перехода. Например, при переходе счета из открытого в закрытое состояние выполняется действие «Сохранить дату закрытия счета».

Действие изображают вдоль линии перехода после имени события, ему предшествует косая черта.

Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

В отличие от большинства других средств UML диаграммы деятельности заимствуют идеи из нескольких различных методов, в частности из метода моделирования состояний SDL и сетей Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов. Диаграммы деятельности также полезны при параллельном программировании, поскольку можно графически изобразить все ветви и определить, когда их необходимо синхронизировать.

Диаграммы деятельности можно применять для описания потоков событий в вариантах использования. С помощью текстового описания можно достаточно подробно рассказать о потоке событий, но в сложных и запутанных потоках с множеством альтернативных ветвей будет трудно понять логику событий. Диаграммы деятельности предоставляют ту же информацию, что и текстовое описание потока событий, но в наглядной графической форме.

На рис. 2.56 приведена диаграмма деятельности для потока событий, связанного с системой бронирования авиабилетов. Рассмотрим ее нотацию.

Основным элементом диаграммы является *деятельность (activity)*. Интерпретация этого термина зависит от той точки зрения, с которой строится диаграмма (это может быть некоторая задача, которую необходимо выполнить вручную или автоматизированным способом, или операция класса). Деятельность изображается в виде закругленного прямоугольника с текстовым описанием.

Любая диаграмма деятельности должна иметь начальную точку, определяющую начало потока событий. Конечная точка необязательна. На диаграмме может быть несколько конечных точек, но только одна начальная.

На диаграмме могут присутствовать объекты и *потоки объектов (object flow)*. Объект может использоваться или изменяться в одной из деятельностей. Показ объектов и их состояний (в дополнение к диаграммам состояний) помогает понять, когда и как происходит смена состояний объекта.

Объекты связаны с деятельностью через потоки объектов. Поток объектов отмечается пунктирной стрелкой от деятельности к изменяемому объекту или от объекта к деятельности, использующей объект.

На рис. 2.56 после ввода пользователем информации о кредитной карточке билет переходит в состояние «не подтвержден». Когда завершится процесс обработки кредитной карточки и будет подтвержден перевод денег, возникает деятельность «зарезервировать место», переводящая билет в состояние «приобретен»,

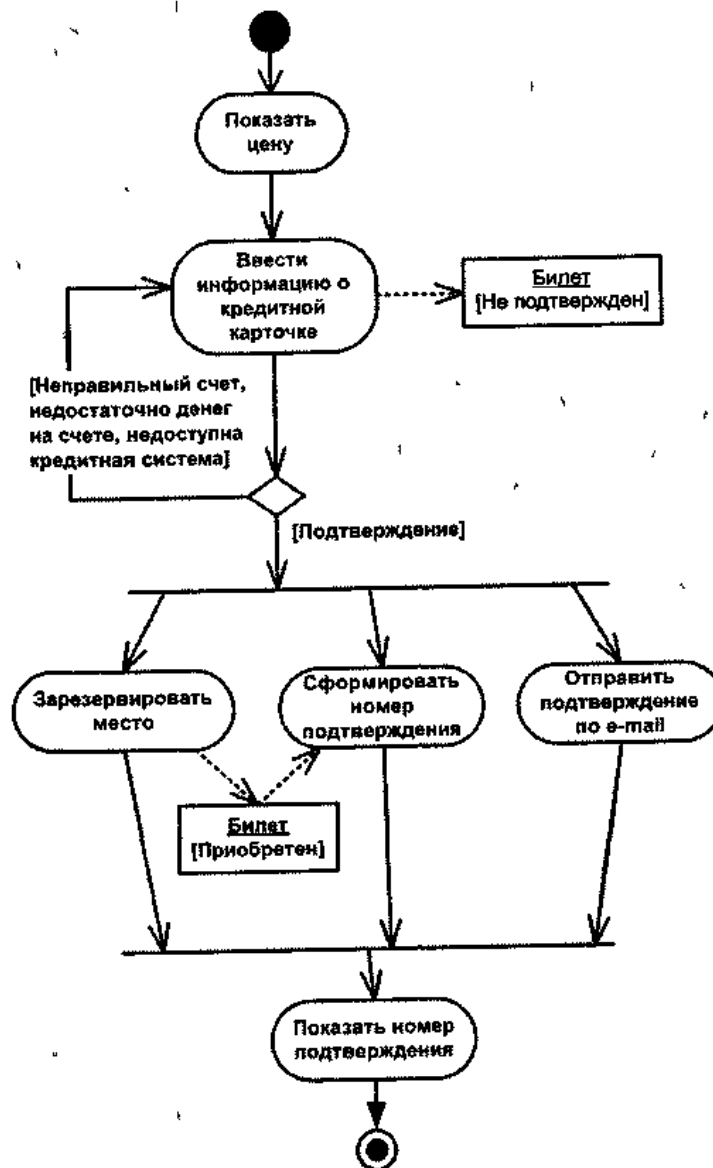


Рис. 2. 56. Диаграмма деятельности

и затем он используется в деятельности «формирование номера подтверждения».

Переход (стрелка) показывает, как поток управления переходит от одной дея-

тельности к другой. Если для перехода определено событие, то переход выполняется только после наступления такого события. Ограничивающие условия определяют, когда переход может, а когда не может осуществиться.

Если необходимо показать, что две или более ветвей потока выполняются параллельно, используются *линейки синхронизации*. В данном примере параллельно выполняются резервирование места, формирование номера подтверждения и отправка почтового сообщения, а после завершения всех трех процессов пользователю выводится номер подтверждения.

Любая деятельность может быть подвергнута дальнейшей декомпозиции. Описание декомпозированной деятельности может быть представлено в виде другой диаграммы деятельности.

Подобно большинству других средств, моделирующих поведение, диаграммы деятельности отражают только вполне определенные его аспекты, поэтому их лучше всего использовать в сочетании с другими средствами.

Диаграммы деятельности предпочтительнее использовать в следующих ситуациях:

- анализ потоков событий в конкретном варианте использования. Здесь нас не интересует связь между действиями и объектами, а нужно только понять, какие действия должны иметь место и каковы зависимости в поведении системы. Связывание действий и объектов выполняется позднее с помощью диаграмм взаимодействия;
- анализ потоков событий в различных вариантах использования. Когда варианты использования взаимодействуют друг с другом, на диаграмме деятельности удобно представить и проанализировать все их потоки событий (в этом случае диаграмма с помощью вертикальных пунктирных линий разделяется на зоны — так называемые «*плавательные дорожки*» (*swimlanes*). В каждой зоне изображаются потоки событий одного из вариантов использования, а связи между разными потоками — в виде переходов или потоков объектов).

ДИАГРАММЫ КОМПОНЕНТОВ

Диаграммы компонентов моделируют физический уровень системы. На них изображаются компоненты ПО и связи между ними. На такой диаграмме обычно выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

На рис. 2.57 изображена одна из диаграмм компонентов для банковской системы.

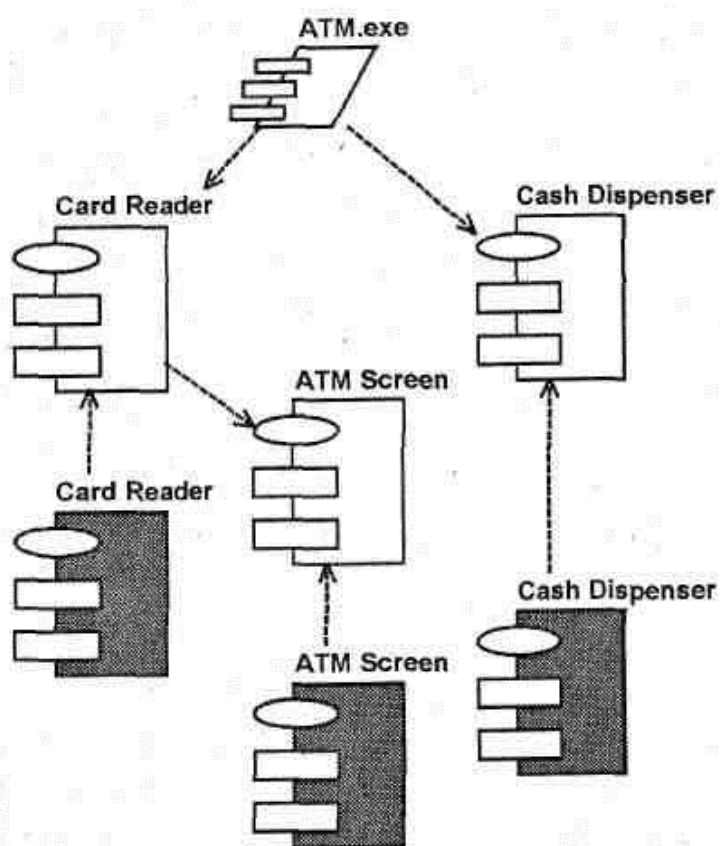


Рис. 2.57. Диаграмма компонентов для клиентской части системы

На этой диаграмме показаны компоненты клиентской части системы. В данном случае система разрабатывается на языке C++. У каждого класса имеется свой собственный заголовочный файл (файл с расширением .h) и файл тела класса (файл с

расширением .cpp). Например, класс ATM Screen преобразуется в компоненты ATM Screen: тело и заголовок класса. Выделенный темным компонент называется спецификацией пакета (package specification) и соответствует файлу тела класса ATM Screen. Невыделенный компонент также называется спецификацией пакета, но соответствует заголовочному файлу класса. Компонент ATM.exe называется спецификацией задачи и моделирует поток управления (thread of processing) — исполняемую программу.

Компоненты соединены зависимостями. Например, класс Card Reader зависит от класса ATM Screen. Это означает, что, для того чтобы класс Card Reader мог быть скомпилирован, класс ATM Screen должен уже существовать. После компиляции всех классов может быть создан исполняемый файл ATMClient.exe.

Банковская система содержит два потока управления и, таким образом, получаются два исполняемых файла. Один из них — это клиентская часть системы, она содержит компоненты Cash Dispenser, Card Reader и ATM Screen. Второй файл — это сервер, включающий в себя компонент Account. Диаграмма компонентов для сервера показана на рис. 2.58.

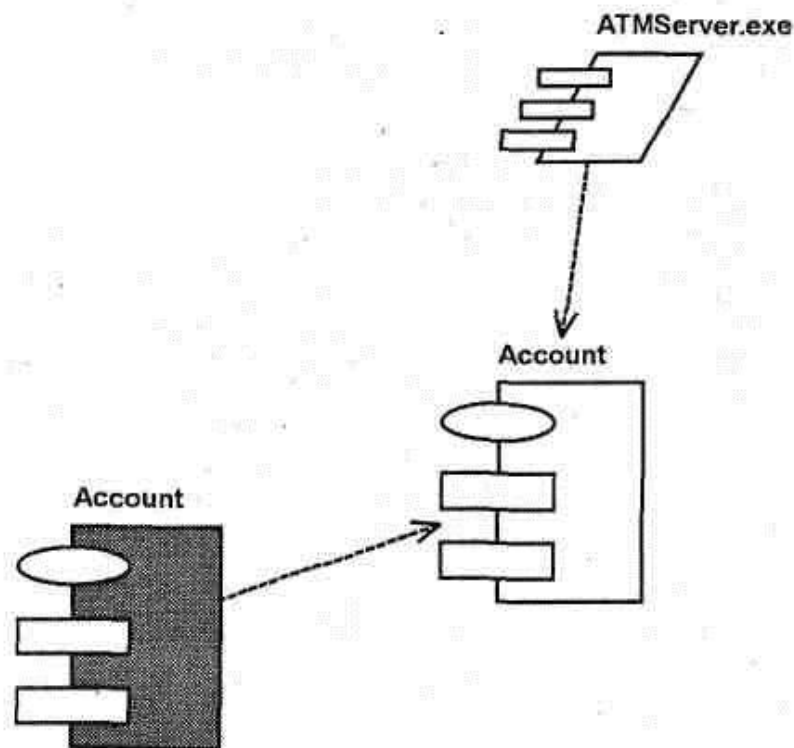


Рис. 2.58. Диаграмма компонентов для сервера

Как видно из примера, в модели системы может быть несколько диаграмм компонентов, в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема является пакетом компонентов.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию и сборку системы. Они нужны там, где начинается генерация кода.

2.5.7.

ДИАГРАММЫ РАЗМЕЩЕНИЯ

Диаграмма размещения отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать размещение объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства — в большинстве случаев, часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мэйнфреймом.

Диаграмма размещения показывает физическое расположение сети и местонахождение в ней различных компонентов. Ее основные элементы:

- узел (node) — вычислительный ресурс — процессор или другое устройство (дискковая память, контроллеры различных устройств и т.д.). Для узла можно задать выполняющиеся на нем процессы;
- соединение (connection) — канал взаимодействия узлов (сеть).

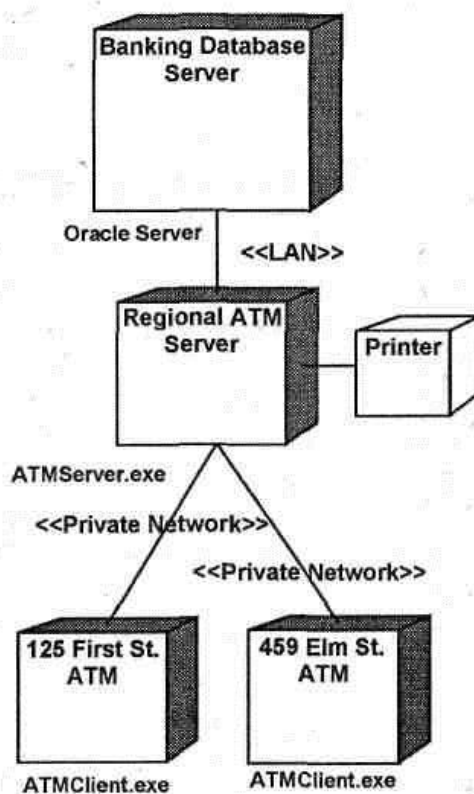


Рис. 2.59. Диаграмма размещения для банковской системы

Например, банковская система состоит из большого количества подсистем, выполняемых на отдельных физических устройствах, или узлах. Диаграмма размещения для такой системы показана на рис. 2.59.

Из данной диаграммы можно узнать о физическом размещении системы. Клиентские программы будут работать в нескольких местах на различных сайтах. Через закрытые сети будет осуществляться их сообщение с региональным сервером системы. На нем будет работать ПО сервера. В свою очередь, посредством локальной сети региональный сервер будет сообщаться с сервером банковской базы данных, работающим под управлением Oracle. Наконец, с региональным сервером соединен принтер.

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом, чтобы понять физическое размещение системы и расположение ее отдельных подсистем.

МЕХАНИЗМЫ РАСШИРЕНИЯ UML

Механизмы расширения UML предназначены для того, чтобы разработчики могли адаптировать язык моделирования к своим конкретным нуждам, не меняя при этом его метамодель. Наличие механизмов расширения принципиально отличает UML от таких средств моделирования, как IDEF0, IDEF1X, IDEF3, DFD и ERM. Перечисленные языки моделирования можно определить как сильно типизированные (по аналогии с языками программирования), поскольку они не допускают произвольной интерпретации семантики элементов моделей. UML, допуская такую интерпретацию (в основном за счет стереотипов), является слабо типизированным языком. К его механизмам расширения относятся:

- стереотипы;
- тегированные (именованные) значения;
- ограничения.

Стереотип — это новый тип элемента модели, который определяется на основе уже существующего элемента. Стереотипы расширяют нотацию модели, могут применяться к любым элементам модели и представляются в виде текстовой метки (см. рис. 2.52) или пиктограммы (иконки).

Стереотипы классов - это механизм, позволяющий разделять классы на категории. Например, основными стереотипами, используемыми в процессе анализа системы (см. подразд. 4.3.2), являются: Boundary (граница), Entity (сущность) и Control (управление).

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Они включают все формы, отчеты, интерфейсы с аппаратурой (такой, как принтеры или сканеры) и интерфейсы с другими системами.

Классы-сущности (entity classes) отражают основные понятия (абстракции) предметной области и, как правило, содержат хранимую информацию. Обычно для

каждого класса-сущности создают таблицу в базе данных.

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий потоки событий этого варианта использования. Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности — остальные классы не посылают ему большого количества сообщений. Вместо этого он сам посылает множество сообщений. Управляющий класс просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.

В системе могут быть и другие управляющие классы, общие для нескольких вариантов использования. Например, класс SecurityManager (менеджер безопасности) может отвечать за контроль событий, связанных с безопасностью. Класс TransactionManager (менеджер транзакций) занимается координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с другими элементами функционирования системы, такими, как разделение ресурсов, распределенная обработка данных или обработка ошибок.

Помимо упомянутых стереотипов, разработчики ПО могут создавать собственные наборы стереотипов, формируя тем самым специализированные подмножества UML (например, для описания бизнес-процессов, Web-приложений, баз данных и т.д.). Такие подмножества (наборы стереотипов) в стандарте языка UML носят название *профилей* языка.

Именованное значение — это пара строк «тег = значение», или «имя = содержимое», в которых хранится дополнительная информация о каком-либо элементе системы, например, время создания, статус разработки или тестирования, время окончания работы над ним и т.п.

Ограничение — это семантическое ограничение, имеющее вид текстового выражения на естественном или формальном языке (OCL — Object Constraint Language), которое невозможно выразить с помощью нотации UML.

Одних графических средств, таких как диаграмма классов или диаграмма состояний, недостаточно для разработки точных и непротиворечивых моделей сложных

систем. Существует необходимость задания дополнительных ограничений, которым должны удовлетворять различные компоненты или объекты модели. Традиционно подобные ограничения описывались с помощью естественного языка. Однако, как показала практика системного моделирования, такие сформулированные на естественном языке ограничения страдают неоднозначностью и нечеткостью.

Для преодоления этих недостатков и придания рассуждениям строгого характера были разработаны различные *формальные языки*. Однако традиционные формальные языки требуют для своего конструктивного использования знания основ математической логики и теории формального вывода. С одной стороны, это делает формальные языки естественным средством для построения математических моделей, а, с другой стороны, существенно ограничивает круг потенциальных пользователей, поскольку разработка формальных моделей требует специальной квалификации.

Другая особенность традиционных формальных языков заключается в присущей им бедной семантике. Базовые элементы формальных языков имеют слишком абстрактное содержание, что затрудняет их непосредственную интерпретацию в понятиях моделей конкретных технических систем и бизнес-процессов. Поскольку процесс объектно-ориентированного анализа и проектирования направлен на построение конструктивных моделей сложных систем, которые должны быть реализованы в форме программных систем и аппаратных комплексов, это является серьезным недостатком. Необходимо применение такого формального языка, базовые элементы которого адекватно отражают семантику основных конструкций объектной модели.

Именно для этих целей и был разработан язык OCL. По своей сути он является формальным языком, более простым для изучения, чем традиционные формальные языки. В то же время язык OCL специально ориентирован на описание бизнес-процессов и бизнес-логики, поскольку был разработан в одном из отделений корпорации IBM.

OCL представляет собой формальный язык для описания ограничений, которые могут быть использованы при определении различных компонентов языка UML.

Хотя язык OCL является частью языка UML, в общем случае он может иметь гораздо более широкую область приложений, чем конструкции языка UML. Средства языка OCL позволяют специфицировать не только объектные ограничения, но и другие выражения логико-лингвистического характера, такие, как предусловия, постусловия и ограничивающие условия. Этот язык ничего не изменяет в графических моделях, а только дополняет их. Это означает, что выражения языка OCL никогда не могут изменить состояние системы, хотя эти выражения и могут быть использованы для спецификации самого процесса изменения этого состояния. Выражения языка OCL также не могут изменять отдельные значения атрибутов и операций для объектов и их связей. Всякий раз, когда оценивается одно из таких выражений, на выходе получается лишь некоторое значение.

Язык OCL не является языком программирования в обычном смысле, поскольку не позволяет записать логику выполнения программы или последовательность управляющих действий. Средства этого языка не предназначены для описания процессов вычисления выражений, а только лишь фиксируют необходимость выполнения тех или иных условий применительно к отдельным компонентам моделей. Главное назначение языка OCL — гарантировать справедливость ограничений для отдельных объектов модели. В этом смысле OCL можно считать языком моделирования, который вовсе не обязан допускать строгую реализацию в инструментальных средствах.

Язык OCL может быть использован для решения следующих задач:

- описание инвариантов классов и типов в модели классов;
- описание пред- и постусловий в операциях и методах;
- описание ограничивающих условий элементов модели;
- навигация по структуре модели;
- спецификация ограничений на операции.

Описание языка OCL отличается от описаний традиционных формальных языков менее формальным характером. Базовым элементом языка OCL является выражение, которое строится по определенным правилам. При этом допускается расширение конструкций языка за счет включения в его состав дополнительных типов.

В модели выражение используется для записи некоторых условий, которым должны удовлетворять все экземпляры соответствующего классификатора. В этом случае говорят, что выражение служит для представления некоторых инвариантных свойств соответствующих элементов модели.

2.5.9.

КОЛИЧЕСТВЕННЫЙ АНАЛИЗ ДИАГРАММ UML

Методика количественной оценки и сравнения диаграмм UML основана на присвоении элементам диаграмм оценок, зависящих от их информационной ценности, а также от вносимой ими в диаграмму дополнительной сложности. Ценность отдельных элементов меняется в зависимости от типа диаграммы, на которой они находятся.

Количественную оценку диаграммы можно провести по следующей формуле:

$$S = \frac{(\sum S_{Obj} + \sum S_{Lnk})}{(1 + Obj + \sqrt{(T_{Obj} + T_{Lnk})})}$$

где S — оценка диаграммы, S_{Obj} — оценки для элементов диаграммы, S_{Lnk} — оценки для связей на диаграмме, Obj — число объектов на диаграмме, T_{Obj} — число типов объектов на диаграмме, T_{Lnk} — число типов связей на диаграмме.

Если диаграмма содержит большое число связей одного типа (например, модель предметной области), то число и тип связей можно не учитывать, и формула расчета приводится к виду:

$$S = \frac{(\sum S_{Obj})}{(1 + Obj + \sqrt{(T_{Obj})})}$$

Если на диаграмме классов показаны атрибуты и операции классов, можно учесть их при расчете, при этом оценка прибавляется к оценке соответствующего класса:

$$S_{cls} = \frac{(\sqrt{Op} + \sqrt{Atr}) * (Op + Atr)}{0,3}$$

где S_{cls} — оценка операций и атрибутов для класса, Op — число операций у класса, Atr - число атрибутов у класса. При этом учитываются только атрибуты и операции, отображенные на диаграмме.

Далее приводятся оценки для различных типов элементов и связей.

Основные элементы языка UML

Тип элемента	Оценка для элемента
Класс	5
Интерфейс	4
Вариант использования	2
Компонент	4
Узел (node)	3
Процессор	2
Взаимодействие	6
Пакет	4
Состояние	4
Примечание	2

Основные типы связей языка UML

Тип связи	Оценка для связи
Зависимость	2
Ассоциация	1
Агрегация	2
Композиция	3
Обобщение	3
Реализация	2

Остальные типы связей должны рассматриваться как ассоциации.

Недостатком диаграммы является как слишком низкая оценка (при этом

диаграмма недостаточно информативна), так и слишком высокая оценка (при этом диаграмма обычно слишком сложна для понимания). Далее приведены диапазоны оптимальных оценок для основных типов диаграмм.

Диапазоны оценок для диаграмм UML

Тип диаграммы	Диапазон оценок
Классов — с атрибутами и операциями	5-5,5
Классов — без атрибутов и операций	3-3,5
Компонентов	3,5-4
Вариантов использования	2,5-3
Размещения	2-2,5
Последовательности	3-3,5
Кооперативная	3,5-4
Пакетов	3,5-4
Состояний	2,5-3

2.6.

ОБРАЗЦЫ

Образцы (patterns) — это одни из важнейших составных частей объектно-ориентированной технологии разработки ПО. Они широко применяются в инструментах анализа, подробно описываются в книгах¹ и часто обсуждаются на конференциях семинарах по объектно-ориентированному проектированию. Существует множество групп и курсов по их изучению. Изучение образцов помогает лучше понять объектно-ориентированный анализ и проектирование.

¹ Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес: Пер. с англ. — СПб.: Питер, 2001.

Объектно-ориентированное проектирование ПО — достаточно сложный процесс, который еще больше усложняется в случае необходимости повторного использования проектных решений. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить существенные связи между классами. Проектное решение должно, с одной стороны, соответствовать решаемой задаче, а с другой — быть достаточно общим, чтобы удалось учесть все требования, которые могут возникнуть в будущем. Желательно также избежать или, по крайней мере, свести к минимуму необходимость перепроектирования. Опытные разработчики знают, что обеспечить «правильное», т.е. в достаточной мере гибкое и пригодное для повторного использования решение, с первого раза очень трудно, если вообще возможно. Прежде чем считать цель достигнутой, они обычно пытаются опробовать найденное решение на нескольких задачах и каждый раз модифицируют его.

И все же опытному разработчику удается создать хороший проект системы. Прежде всего ему понятно, что не нужно решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение один раз, он будет прибегать к нему снова и снова. Именно благодаря накопленному опыту проектировщик и становится экспертом в своей области. Во многих объектно-ориентированных системах можно встретить повторяющиеся проектные решения (образцы), состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего решение становится более гибким, и им можно воспользоваться повторно. Проектировщик, знакомый с образцами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз «изобретать велосипед».

В 1970-х годах Кристофер Александер, профессор архитектуры Калифорнийского университета написал несколько книг, документируя образцы в гражданском строительстве и архитектуре. Разработчики ПО впоследствии заимствовали идею образцов, основываясь на его трудах, тем более, что к тому времени среди программистов уже

присутствовал растущий интерес к этим идеям.

По словам Кристофера Александера, «любой образец описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново». Хотя Александер имел в виду образцы, возникающие при проектировании зданий и городов, но его слова верны и в отношении образцов объектно-ориентированного проектирования. Решения относительно ПО выражаются в терминах объектов и интерфейсов, а не стен и дверей, но в обоих случаях *образец можно определить как общее решение некоторой проблемной ситуации в заданном контексте*. Образец состоит из четырех основных элементов:

- имя;
- проблема;
- решение;
- следствия.

Сославшись на *имя образца*, можно сразу описать проблему проектирования, ее решения и их последствия. Присваивание образцам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря образцов можно вести обсуждение с коллегами, упоминать образцы в документации, в тонкостях представлять проект системы.

Проблема — это описание решаемой задачи. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный образец.

Решение — это описание элементов проектного решения, связей между ними и функций каждого элемента. Конкретное решение или реализация не имеются в виду, поскольку образец — это шаблон, применимый в самых разных ситуациях. Обычно дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (классов и объектов).

Следствия — это описание области применения, достоинств и недостатков образца. Хотя при описании проектных решений о следствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки применения данного образца. Поскольку в объектно-ориентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы.

Образцы могут рассматриваться на различных уровнях абстракции и в различных предметных областях. Наиболее общими категориями образцов ПО являются;

- образцы бизнес-моделирования;
- образцы анализа;
- образцы поведения;
- образцы проектирования;
- архитектурные образцы;
- образцы программирования.

Даже в этом кратком списке категорий присутствует большое количество уровней абстракции и ортогональных систем классификации. Например, под образцом проектирования понимается *описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.*

В языке UML образец представляется с помощью кооперации со стереотипом «pattern». **Кооперация** (collaboration) определяется как описание совокупности взаимодействующих объектов, реализующих некоторое поведение (например, в рамках варианта использования или операции класса). Кооперация имеет статическую и динамическую части. В статической части (на диаграмме классов) описываются роли, которые могут играть объекты и связи в экземпляре данной кооперации. Динамическая часть состоит из одной или более диаграмм взаимодействия, показывающих потоки сообщений, которыми обмениваются участники кооперации. Кроме того, любой образец содержит стандартную диаграмму классов под названием «Participants» («Участники»),

на которой изображается сам образец в виде кооперации с его именем и набор классов, участвующих в реализации образца.

В качестве примера приведем образец бизнес-моделирования под названием Employment (Занятость).

Проблема заключается в моделировании различных форм занятости в пределах организации. Данная задача решается в контексте системы планирования ресурсов предприятия.

Решение: занятость моделируется как контракт между личностью и организацией, указывающий выполняемые обязанности, контрактные условия, даты начала и конца работы. Личность характеризуется набором атрибутов (имя, адрес и дата рождения), может занимать более чем одну должность в одной и той же организации.

На рис. 2.60 приведена диаграмма «Участники» для данного образца. Она содержит кооперацию Employment и набор из пяти классов.

- Employee Profile (Служащий) — описание служащего с набором атрибутов.
- Organization Profile (Организация) — описание самой организации.
- Employment (Занятость) — описание связи между служащим и организацией.
- Position (Должность) — описание должности со своими атрибутами (такими, как должностной оклад и должностные инструкции).
- Position Assignment (Назначение на должность) — описание связи между служащим и занимаемыми должностями.

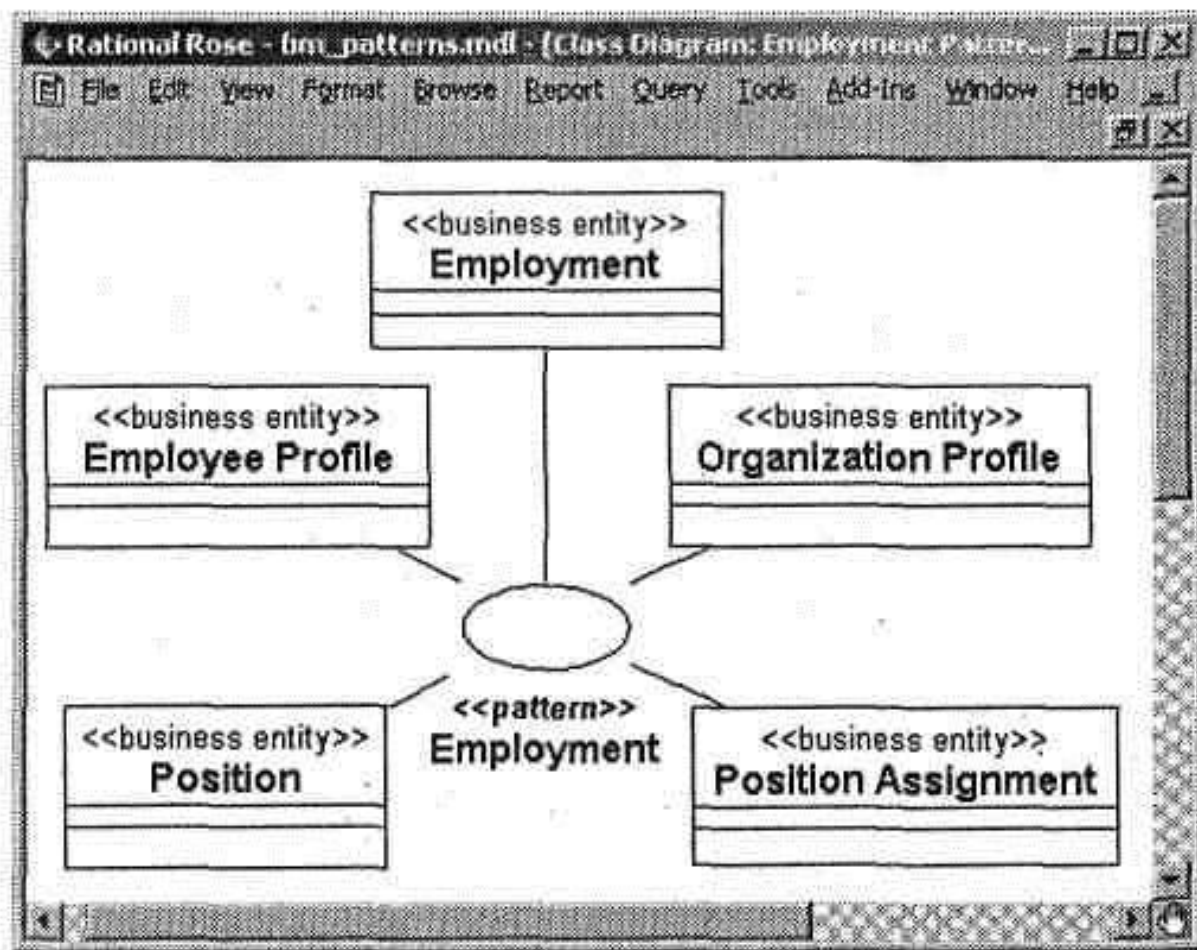


Рис. 2.60. Диаграмма «Участники» для образца Employment

Статическая часть образца (диаграмма классов) показана на рис. 2.61.

Достоинства применения образцов при проектировании ПО заключаются в следующем:

- *Возможность многократного использования.* Повторное использование решений из уже завершенных успешных проектов позволяет быстро приступить к решению новых проблем и избежать типичных ошибок. Разработчик получает прямую выгоду от использования опыта других разработчиков, избежав необходимости вновь и вновь «изобретать велосипед».
- *Применение единой терминологии.* Профессиональное общение и работа в группе (команде разработчиков) требует наличия единого базового словаря и единой точки зрения на проблему.

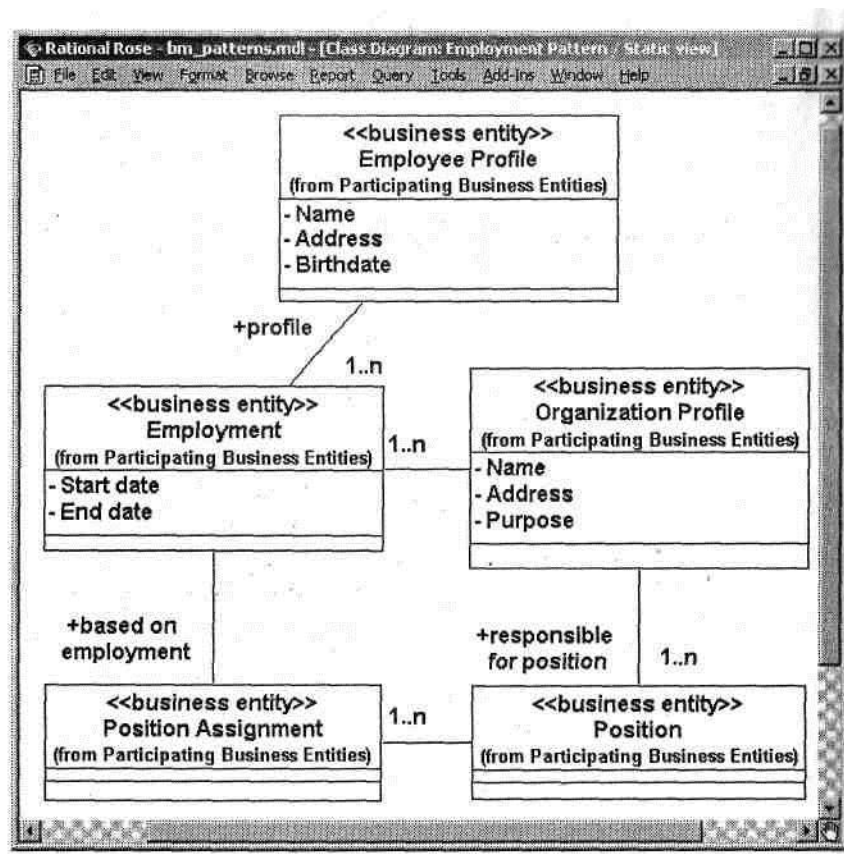


Рис. 2.61. Статическая часть образца Employment

Образцы предоставляют подобную общую точку зрения как на этапе анализа, так и при реализации проекта.

- *Повышение эффективности труда отдельных исполнителей и всей группы в целом.* Это происходит из-за того, что начинающие члены группы видят на примере более опытных разработчиков, как образцы проектирования могут применяться и какую пользу они приносят. Совместная работа дает новичкам стимул и реальную возможность быстрее изучить и Освоить эти новые концепции.
- *Создание модифицируемого и гибкого ПО.* Причина состоит в том, что образцы уже испытаны временем, поэтому их использование позволяет создавать структуры, допускающие модификацию в большей степени, чем это возможно в случае решения, первым пришедшего на ум.
- *Ограничение пространства решений.* Использование образцов создает границы в рамках пространства решений проектирования и реализации.

Таким образом, образец настоятельно рекомендует проектировщику границы, которых должна придерживаться реализация. Выход за эти границы нарушает строгое соблюдение образца и проектных решений и может привести к нежелательным последствиям. Тем не менее, образцы не подавляют творчество. Напротив, они описывают структуру или форму на некотором уровне абстракции. Проектировщики и разработчики все еще располагают многими возможностями для реализации образцов в этих границах.

2.7.

СОПОСТАВЛЕНИЕ И ВЗАИМОСВЯЗЬ СТРУКТУРНОГО И ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДОВ

Традиционно, до недавнего времени, у большинства разработчиков понятие «проектирование» ассоциировалось со структурным проектированием по методу «сверху вниз» на основе функциональной декомпозиции, согласно которой вся система в целом представляется как одна большая функция и разбивается на подфункции, которые в свою очередь разбиваются на подфункции и т.д. Эти функции подобны вариантам использования и объектно-ориентированной системе, которые соответствуют действиям, выполняемым системой в целом.

Главный недостаток структурного проектирования заключается в следующем: процессы и данные существуют отдельно друг ОТ друга, причем проектирование ведется от процессов к данным. Таким образом, помимо функциональной декомпозиции, существует также структура данных, находящаяся на втором плане.

В ООП основная категория объектной модели — класс — объединяет в себе на элементарном уровне как данные, так и операции, которые над ними вы-

полняются. Именно с этой точки зрения изменения, связанные с переходом от структурного подхода к ООП, являются наиболее заметными. Разделение процессов и данных преодолено, однако остается проблема преодоления сложности системы, которая решается путем использования механизма пакетов.

Поскольку данные по сравнению с процессами являются более стабильной и относительно редко изменяющейся частью системы, отсюда следует главное достоинство ООП. Гради Буч сформулировал его следующим образом: *объектно-ориентированные системы более открыты и легче поддаются внесению изменений, поскольку их конструкция базируется на устойчивых формах. Это дает возможность системе развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.*

Буч отметил также ряд следующих преимуществ ООП:

- объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование ООП существенно повышает уровень унификации разработки и пригодность для повторного использования не только ПО, но и проектов, что в конце концов ведет к сборочному созданию ПО. Системы зачастую получаются более компактными, чем их не объектно-ориентированные эквиваленты, что означает не только уменьшение объема программного кода, но и удешевление проекта за счет использования предыдущих разработок;
- объектная декомпозиция уменьшает риск создания сложных систем ПО, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем. Процесс интеграции системы растягивается на все время разработки, а не превращается в единовременное событие;
- объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную ре-

- ализацию;
- объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования.

К недостаткам ООП относятся некоторое снижение производительности функционирования ПО (которое, однако, по мере роста производительности компьютеров становится все менее заметным) и высокие начальные затраты. Объектная декомпозиция существенно отличается от функциональной, поэтому переход на новую технологию связан как с преодолением психологических трудностей, так и дополнительными финансовыми затратами. При переходе от структурного подхода к объектному, как и при всякой смене технологии, необходимо вкладывать деньги в приобретение новых инструментальных средств. Здесь следует учесть расходы на обучение методу, инструментальным средствам и языку программирования. Для некоторых организаций эти обстоятельства могут стать серьезными препятствиями.

Объектно-ориентированный подход не дает немедленной отдачи. Эффект от его применения начинает сказываться после разработки двух-трех проектов и накопления повторно используемых компонентов, отражающих типовые проектные решения в данной области. Переход организации на объектно-ориентированную технологию — это смена мировоззрения, а не просто изучение новых CASE-средств и языков программирования.

Таким образом, структурный подход по-прежнему сохраняет свою значимость и достаточно широко используется на практике. На примере языка UML хорошо видно, что его авторы заимствовали то рациональное, что можно было взять из структурного подхода: элементы функциональной декомпозиции в диаграммах вариантов использования, диаграммы состояний, диаграммы деятельности и др. Очевидно, что в конкретном проекте сложной системы невозможно обойтись только одним способом декомпозиции. Можно начать декомпозицию каким-либо одним способом, а затем, используя полученные результаты, попытаться рассмотреть систему с другой точки зрения.

Теперь рассмотрим взаимосвязь между структурным и объектно-ориентированным подходами. Основой взаимосвязи является общность ряда категорий и понятий обоих подходов (процесс и вариант использования, сущность и класс и др.). Эта взаимосвязь может проявляться в различных формах. Так, одним из возможных вариантов является использование структурного анализа как основы для объектно-ориентированного проектирования. При этом структурный анализ следует прекращать, как только структурные модели начнут отражать не только деятельность организации (бизнес-процессы), а и систему ПО. После выполнения структурного анализа можно различными способами приступить к определению классов и объектов. Так, если взять какую-либо отдельную диаграмму потоков данных, то кандидатами в классы могут быть элементы структур данных.

Другой формой проявления взаимосвязи можно считать интеграцию объектной и реляционной технологий. Реляционные СУБД являются на сегодняшний день основным средством реализации крупномасштабных баз данных и хранилищ данных. Причины этого достаточно очевидны: реляционная технология используется достаточно долго, освоена огромным количеством пользователей и разработчиков, стала промышленным стандартом, в нее вложены значительные средства и создано множество корпоративных БД в самых различных отраслях, реляционная модель проста и имеет строгое математическое основание; существует большое разнообразие промышленных средств проектирования, реализации и эксплуатации реляционных БД. Вследствие этого реляционные БД в основном используются для хранения и поиска объектов в так называемых объектно-реляционных системах.

Объектно-ориентированное проектирование имеет точки соприкосновения с реляционным проектированием. Например, как было отмечено выше, классы в объектной модели могут некоторым образом соответствовать сущностям (в качестве упражнения можно предложить детально проанализировать все сходства и различия диаграмм «сущность-связь» и диаграмм классов). Как правило, такое соответствие имеет место только на ранней стадии разработки системы. В

дальнейшем, разумеется, цели объектно-ориентированного проектирования (адекватное моделирование предметной области в терминах взаимодействия объектов) и разработки реляционной БД (нормализация данных) расходятся. Таким образом, единственно возможным средством преодоления данного разрыва является построение отображения между объектно-ориентированной и реляционными технологиями, которое в основном сводится к отображению между диаграммами классов и реляционной моделью.

Взаимосвязь между структурным и объектно-ориентированным подходами достаточно четко просматривается в методиках анализа и проектирования, которые будут рассмотрены в последующих главах.

! Следует запомнить

1. Главным способом преодоления сложности разработки больших систем ПО является правильная декомпозиция.

2. Сущность структурного подхода к разработке ПО заключается в его декомпозиции (разбиении) в соответствии с выполняемыми функциями.

3. Сущность объектно-ориентированного подхода к разработке ПО заключается в объектной декомпозиции. При этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

Основные понятия

Модель, архитектура.

Внешняя сущность, процесс, накопитель данных, поток дан-11 i.ix, сущность, связь, атрибут.

Класс, объект, абстрагирование, инкапсуляция, модульность, иерархия, наследование, вариант использования, действующее лицо, компонент, интерфейс.

Стереотип, образец, кооперация.

? Вопросы для самоконтроля

1. В чем заключаются основные принципы структурного подхода?
2. Что общего и в чем различия между методом SADT и моделированием потоков данных?
3. В чем заключаются достоинства и недостатки структурного подхода?
4. В чем заключаются основные принципы объектно-ориентированного подхода?
5. В чем состоят достоинства и недостатки объектно-ориентированного подхода?
6. Чем язык UML принципиально отличается от моделей SADT, DFD и ERM?
7. Каковы принципиальные различия и что общего между структурным и объектно-ориентированным подходами?

ГЛАВА 3

МОДЕЛИРОВАНИЕ БИЗНЕС-ПРОЦЕССОВ И СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ

Прочитав эту главу, вы узнаете:

- *Что представляет собой моделирование бизнес-процессов и спецификация требований к ПО.*
- *В чем заключается структурный (процессный) подход к моделированию бизнес-процессов.*
- *В чем заключается объектно-ориентированный подход к моделированию бизнес-процессов и спецификации требований.*

ОСНОВНЫЕ ПОНЯТИЯ МОДЕЛИРОВАНИЯ

БИЗНЕС-ПРОЦЕССОВ

Моделирование бизнес-процессов является важной составной частью проектов по созданию крупномасштабных систем ПО. Отсутствие таких моделей является одной из главных причин неудач многих проектов.

Бизнес-процесс определяется как логически заверченный набор взаимосвязанных и взаимодействующих видов деятельности, поддерживающий деятельность организации и реализующий ее политику, направленную на достижение поставленных целей. Международный стандарт ISO 9000 определяет организацию как группу работников и необходимых средств с распределением ответственности, полномочий и взаимоотношений. По-другому организацию можно определить как систематизированное, сознательное объединение действий людей, преследующих достижение конкретных целей. Организация может быть корпоративной, государственной или частной.

Бизнес-процесс в узком смысле можно определить как набор связанных процедур, направленных на достижение определенного результата, представляющего ценность для потребителя. Бизнес-процесс использует определенные ресурсы (*финансовые, материальные, человеческие, информационные*) для преобразования входных элементов в выходные.

Важным шагом структуризации деятельности любой организации являются выделение и классификация бизнес-процессов. Можно выделить следующие классы процессов: основные процессы; обеспечивающие процессы и процессы управления.

Основными бизнес-процессами являются процессы, непосредственно связанные с созданием стоимости, ориентированные на производство товаров или оказание услуг, составляющих основную деятельность организации и обеспечивающих получение дохода.

Обеспечивающие бизнес-процессы не увеличивают ценность продукта или услуги для потребителя, но необходимы для деятельности предприятия. Они предназначены для поддержки выполнения основных бизнес-процессов. Таковыми процессами являются финансовое обеспечение деятельности, обеспечение кадрами, юридическое обеспечение, администрирование, обеспечение безопасности, поставка комплектующих материалов, ремонт и техническое обслуживание и т.д.

Бизнес-процессы управления — это процессы, охватывающие весь комплекс функций управления на уровне каждого бизнес-процесса и системы в целом. Примерами таких процессов могут быть процессы стратегического, оперативного и текущего планирования, процессы формирования и выполнения управляющих воздействий. Процессы управления оказывают воздействие на все остальные процессы организации.

Бизнес-модель — это формализованное (в данном случае — графическое) описание процессов, связанных с ресурсами, и отражающих существующую или предполагаемую деятельность предприятия.

Построение бизнес-моделей заключается в применении различных методов и средств для визуального моделирования бизнес-процессов. Цели моделирования:

- обеспечить понимание структуры организации и динамики происходящих в ней процессов;
- обеспечить понимание текущих проблем организации и возможностей их решения;
- убедиться, что заказчики, пользователи и разработчики одинаково понимают цели и задачи организации;
- создать базу для формирования требований к будущей ИС организации.

Основная область применения бизнес-моделей — это реинжиниринг бизнес-процессов. При этом предполагается построение моделей текущей и перспективной деятельности, а также плана и программы перехода из первого состояния во второе. Любое современное предприятие является сложной системой, его деятельность

включает в себя исполнение десятков тысяч взаимовлияющих функций и операций. Человек не в состоянии понимать, как такая система функционирует в деталях — это выходит за границы его возможностей. Поэтому главная идея создания моделей «AS-IS» и «AS-TO-BE» (см. подразд. 2.2) — понять, что делает (будет делать) рассматриваемое предприятие и как оно функционирует (будет функционировать) для достижения своих целей.

Назначением будущих ИС является в первую очередь решение проблем бизнеса посредством современных информационных технологий. Требования к ИС формируются на основе бизнес-модели, а критерии проектирования систем прежде всего основываются на наиболее полном их удовлетворении.

Следует отметить, что модели бизнес-процессов являются не просто промежуточным результатом, используемым консультантом для выработки каких-либо рекомендаций и заключений. Они представляют собой самостоятельный результат, имеющий большое практическое значение, которое следует из целей их построения.

Модель бизнес-процесса должна давать ответы на вопросы:

1. Какие процедуры (функции, работы) необходимо выполнить для получения заданного конечного результата?
2. В какой последовательности выполняются эти процедуры?
3. Какие механизмы контроля и управления существуют в рамках рассматриваемого бизнес-процесса?
4. Кто выполняет процедуры процесса?
5. Какие входящие документы/информацию использует каждая процедура процесса?
6. Какие исходящие документы/информацию генерирует процедура процесса?
7. Какие ресурсы необходимы для выполнения каждой процедуры процесса?
8. Какая документация/условия регламентирует выполнение процедуры?
9. Какие параметры характеризуют выполнение процедур и процесса в целом?

Важным элементом модели бизнес-процессов являются *бизнес-правила*, или правила предметной области. Типичными бизнес-правилами являются корпоративная политика и государственные законы. Бизнес-правила обычно формулируются в

специальном документе и могут отражаться в моделях. Для организации бизнес-правил предлагается множество различных схем классификации. Наиболее полной можно считать следующую классификацию бизнес-правил (в скобках приведены примеры правил для гипотетической системы обработки заказов в торговой компании):

- Факты - достоверные утверждения о бизнес-процессах, называемых также инвариантами (оплачивается доставка каждого заказа; со стоимости доставки налог с продаж не берется).
- Правила-ограничения — определяют различные ограничения на выполняемые операции:
- Управляющие воздействия и реакции на воздействия (когда заказ отменен и еще не доставлен, то его обработка завершается).
- Операционные ограничения — предусловия и постусловия (доставить заказ клиенту только при наличии адреса доставки).
- Структурные ограничения (заказ включает по крайней мере один продукт).
- Активаторы операций — правила, при определенных условиях приводящие к выполнению каких-либо действий (если срок хранения товара на складе истек, об этом надо уведомить ответственное лицо).
- Правила вывода:
- Правила-следствия — правила, устанавливающие новые факты на основе достоверности определенных условий (клиент получает положительный статус только при условии оплаты счетов в течение 30 дней).
- Вычислительные правила — различные вычисления, выполняемые с использованием математических формул и алгоритмов ($\text{цена нетто} = \text{цена продукта} * (1 + \text{процент налога} / 100)$).
- Для моделирования бизнес-процессов необходимо использовать определенную методику, которая включает:
- описание методов моделирования — способов представления реальных объектов предприятия при помощи объектов модели;
- последовательность шагов по сбору информации, ее обработке и представле-

- нию в виде моделей;
- типовые формы документов.

3.2.

СТРУКТУРНЫЙ (ПРОЦЕССНЫЙ) ПОДХОД К МОДЕЛИРОВАНИЮ БИЗНЕС-ПРОЦЕССОВ

3.2.1.

ПРИНЦИПЫ ПРОЦЕССНОГО ПОДХОДА

Основной принцип процессного подхода заключается в структурировании деятельности организации в соответствии с ее бизнес-процессами, а не организационно-штатной структурой. Именно бизнес-процессы, формирующие значимый для потребителя результат, представляют ценность, и именно их улучшением предстоит в дальнейшем заниматься. Модель, основанная на организационно-штатной структуре, может продемонстрировать лишь хаос, царящий в организации (о котором в принципе руководству и так известно, иначе оно бы не инициировало соответствующие работы), на ее основе можно только внести предложения об изменении этой структуры. С другой стороны, модель, основанная на бизнес-процессах, содержит в себе и организационно-штатную структуру предприятия.

В соответствии с этим принципом бизнес-модель должна выглядеть следующим образом:

1. Верхний уровень модели должен отражать только контекст системы — взаимодействие моделируемого единственным контекстным процессом предприятия с внешним миром.
2. На втором уровне модели должны быть отражены основные виды деятельности (тематически сгруппированные бизнес-процессы) предприятия и их взаимосвязи. В случае большого их количества некоторые из них можно вынести на третий уровень модели. Но в любом случае под виды деятельности необходимо

отводить не более двух уровней модели.

3. Дальнейшая детализация бизнес-процессов осуществляется посредством бизнес-функций — совокупностей операций, сгруппированных по определенным признакам. Бизнес-функции детализируются с помощью элементарных бизнес-операций.
4. Описание элементарной бизнес-операции осуществляется посредством задания алгоритма ее выполнения.

Принципы формирования бизнес-модели на верхних уровнях декомпозиций:

- следует избегать чрезмерной детализации (модель бизнес-процесса верхнего уровня должна содержать не более 6-8 блоков функций);
- следует использовать реально существующие названия функций или работ;
- не следует пытаться детально отразить всю существующую логику процесса (это будет сделано при формировании детальных моделей);
- важно отразить общую последовательность работ, подразделения участвующие в их исполнении, основные ресурсы;
- важно отразить основную логику процесса.

Общее число уровней в модели (включая контекстный) не должно превышать 5—6. Практика показывает, что этого вполне достаточно для построения полной функциональной модели современного предприятия любой отрасли.

3.2.2.

ПРИМЕНЕНИЕ ДИАГРАММ

ПОТОКОВ ДАННЫХ

При моделировании бизнес-процессов диаграммы потоков данных (DFD) используются для построения моделей «AS-IS» и «AS-TO-BE», отражая, таким образом, существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними. При этом описание используемых в организации и данных на концептуальном уровне, независимо от Средств реализации базы данных

(СУБД), выполняется с помощью ERM.

Ниже перечислены основные виды и последовательность работ при построении бизнес-моделей с использованием методики Йордона (пример ее применения приведен в подразд. 3.2.5).

1. *Описание контекста процессов и построение начальной контекстной диаграммы.*

Начальная контекстная диаграмма потоков данных должна содержать нулевой процесс с именем, отражающим деятельность организации, внешние сущности, соединенные с нулевым процессом посредством потоков данных. Потоки данных соответствуют документам, запросам или сообщениям, которыми внешние сущности обмениваются с организацией.

2. *Спецификация структур данных.*

Определяется состав потоков данных и готовится исходная информация для построения концептуальной модели данных в виде структур данных. Выделяются все структуры и элементы данных типа «итерация», «условное вхождение» и «альтернатива». Простые структуры и элементы данных объединяются в более крупные структуры. В результате для каждого потока данных должна быть сформирована иерархическая (древовидная) структура, конечные элементы (листья) которой являются элементами данных, узлы дерева являются структурами данных, а верхний узел дерева соответствует потоку данных в целом. Результат можно представить в виде текстового описания, подобного описанию структур данных в языках программирования.

3. *Построение начального варианта концептуальной модели данных.*

Для каждого класса объектов предметной области выделяется сущность. Устанавливаются связи между сущностями и определяются их характеристики (мощность связи и класс принадлежности). Строится диаграмма «сущность-связь» (без атрибутов сущностей).

4. *Построение диаграмм потоков данных нулевого и последующих уровней.*

Для завершения анализа функционального аспекта деятельности организации детализируется (декомпозируется) начальная контекстная диаграмма. При этом

можно построить диаграмму для каждого события, поставив ему в соответствие процесс и описав входные и выходные потоки, накопители данных, внешние сущности и ссылки на другие процессы для описания связей между этим процессом и его окружением. После этого все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Проверяется соответствие между контекстной диаграммой и диаграммой нулевого уровня (каждый поток данных между системой и внешней сущностью на диаграмме нулевого уровня должен быть представлен и на контекстной диаграмме).

Процессы разделяются на группы, которые имеют много общего (работают с одинаковыми данными и/или имеют сходные функции). Они изображаются вместе на диаграмме более низкого (первого) уровня, а на диаграмме нулевого уровня объединяются в один процесс. Выделяются накопители данных, используемые процессами из одной группы.

Декомпозируются сложные процессы и проверяется соответствие различных уровней модели процессов.

Накопители данных описываются посредством структур данных, а процессы нижнего уровня — посредством спецификаций.

5. Уточнение концептуальной модели данных.

Определяются атрибуты сущностей. Выделяются атрибуты-идентификаторы. Проверяются связи, выделяются (при необходимости) зависимые от идентификатора сущности и связи «супертип-подтип».

Проверяется соответствие между описанием структур данных и концептуальной моделью (все элементы данных должны присутствовать на диаграмме в качестве атрибутов).

СИСТЕМА МОДЕЛИРОВАНИЯ ARIS

В настоящее время наблюдается тенденция интеграции разнообразных методов моделирования и анализа систем, проявляющаяся в форме создания интегрированных средств моделирования. Одним из таких средств является продукт, носящий название ARIS - Architecture of Integrated Information System, разработанный германской фирмой IDS Scheer.

Система ARIS представляет собой комплекс средств анализа и моделирования деятельности предприятия, а также разработки ИС. Ее методическую основу составляет совокупность различных методов моделирования, отражающих разные взгляды на исследуемую систему. Одна и та же модель может разрабатываться с использованием нескольких методов, что позволяет использовать ARIS специалистам с различными теоретическими знаниями и настраивать его на работу с системами, имеющими свою специфику.

Методика моделирования ARIS основывается на разработанной профессором Августом Шером теории построения интегрированных ИС, определяющей принципы визуального отображения всех аспектов функционирования анализируемых компаний. ARIS поддерживает четыре типа моделей, отражающих различные аспекты исследуемой системы:

- **организационные модели**, представляющие структуру системы — иерархию организационных подразделений, должностей и конкретных лиц, связи между ними, а также территориальную привязку структурных подразделений;
- **функциональные модели**, содержащие иерархию целей, стоящих перед аппаратом управления, с совокупностью деревьев функций, необходимых для достижения поставленных целей;
- **информационные модели**, отражающие структуру информации, необходимой

для реализации всей совокупности функций системы;

- *модели управления*, представляющие комплексный взгляд на реализацию бизнес-процессов в рамках системы.

Для построения перечисленных типов моделей используются собственные методы моделирования ARIS, а также известные методы и языки моделирования — ERM, UML, OMT и др.

В процессе моделирования каждый аспект деятельности предприятия сначала рассматривается отдельно, а после детальной проработки всех аспектов строится интегрированная модель, отражающая все связи между различными аспектами.

ARIS не накладывает ограничений на последовательность построения указанных выше типов моделей. Процесс моделирования можно начинать с любого из них в зависимости от конкретных условий и целей, преследуемых разработчиками.

Модели в ARIS представляют собой диаграммы, элементами которых являются разнообразные объекты — «функция», «событие», «структурное подразделение», «документ» и т.п. Между объектами устанавливаются разнообразные связи. Так, между объектами «функция» и «структурное подразделение» могут быть установлены связи следующих видов:

- выполняет;
- принимает решение;
- участвует в выполнении;
- должен быть проинформирован о результатах;
- консультирует исполнителей;
- принимает результаты.

Каждому объекту соответствует определенный набор атрибутов, которые позволяют ввести дополнительную информацию о конкретном объекте. Значения атрибутов могут использоваться при имитационном моделировании или для проведения стоимостного анализа.

Таким образом, по результатам выполнения этого этапа возникает набор взаимосвязанных моделей, представляющих собой исходный материал для дальнейшего анализа.

Основная бизнес-модель ARIS — eEPC (extended Event Driven Process Chain — расширенная модель цепочки процессов, управляемых событиями). Ниже приводятся основные объекты, используемые в данной нотации.

Объекты модели eEPC

Наименование объекта	Описание
Функция	Служит для описания функций (процедур, работ), выполняемых подразделениями/сотрудниками предприятия.
Событие	Служит для описания реальных событий, воздействующих на выполнение функций.
Организационная единица	Представляет различные организационные звенья предприятия (например, управление или отдел).
Документ	Отражает реальные носители информации, например бумажный документ.
Прикладная система	Отражает реальную прикладную систему, поддерживающую выполнение функции.
Кластер информации	Характеризует данные (набор сущностей и связей между ними). Используется для создания моделей данных.
Связь между объектами	Описывает тип отношений между некоторыми объектами, например, активацию выполнения функции некоторым событием.
Логический оператор	Оператор одного из трех типов («И», «ИЛИ», исключаящее «ИЛИ»), определяющий связи между событиями и функциями в рамках процесса. Позволяет описать ветвление процесса.

Помимо указанных в таблице основных объектов при построении диаграммы eEPC могут быть использованы многие другие объекты. По существу, модель eEPC расширяет возможности IDEFO, IDEF3 и DFD, обладая всеми их достоинствами и недостатками. Применение большого числа различных объектов, связанных различными типами связей, значительно увеличивает размер модели и делает ее плохо

читаемой. Для понимания смысла нотации eEPC достаточно рассмотреть основные типы объектов и связей. На рис. 3.1 представлена простейшая модель eEPC, описывающая фрагмент бизнес-процесса предприятия.

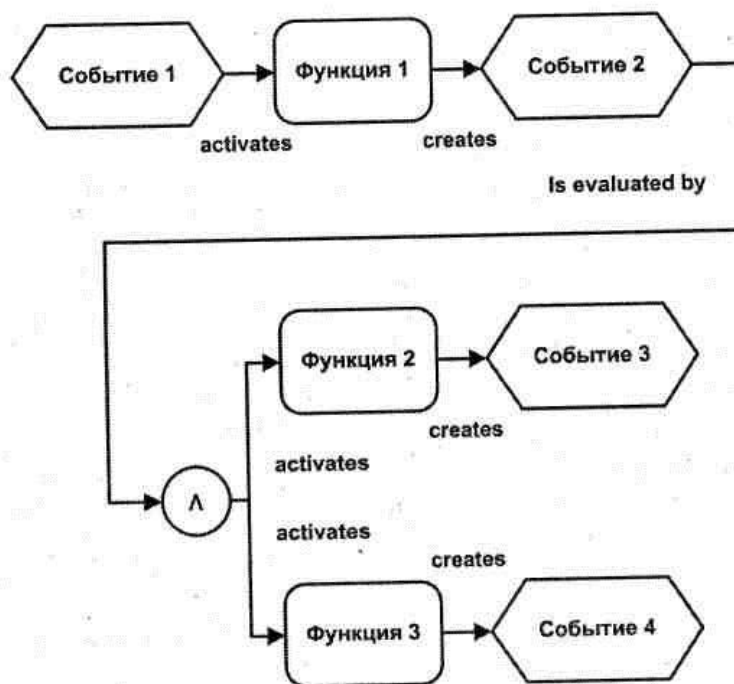


Рис. 3.1. Модель eEPC

Из рис. 3.1 видно, что связи между объектами имеют определенный смысл и отражают последовательность выполнения функций в рамках процесса. Стрелка, соединяющая Событие 1 и Функцию 1, «активирует» или инициирует выполнение Функции 1. Функция 1 «создает» Событие 2, за которым следует символ логического «И», «запускающий» выполнение Функций 2 и 3. Нотация eEPC построена на определенных правилах:

- каждая функция должна быть инициирована событием и должна завершаться событием;
- в каждую функцию не может входить более одной стрелки, «запускающей» выполнение функции, и выходить не более одной стрелки, описывающей завершение выполнения функции.

На рис. 3.2 показано применение различных объектов ARIS при создании модели бизнес-процесса.

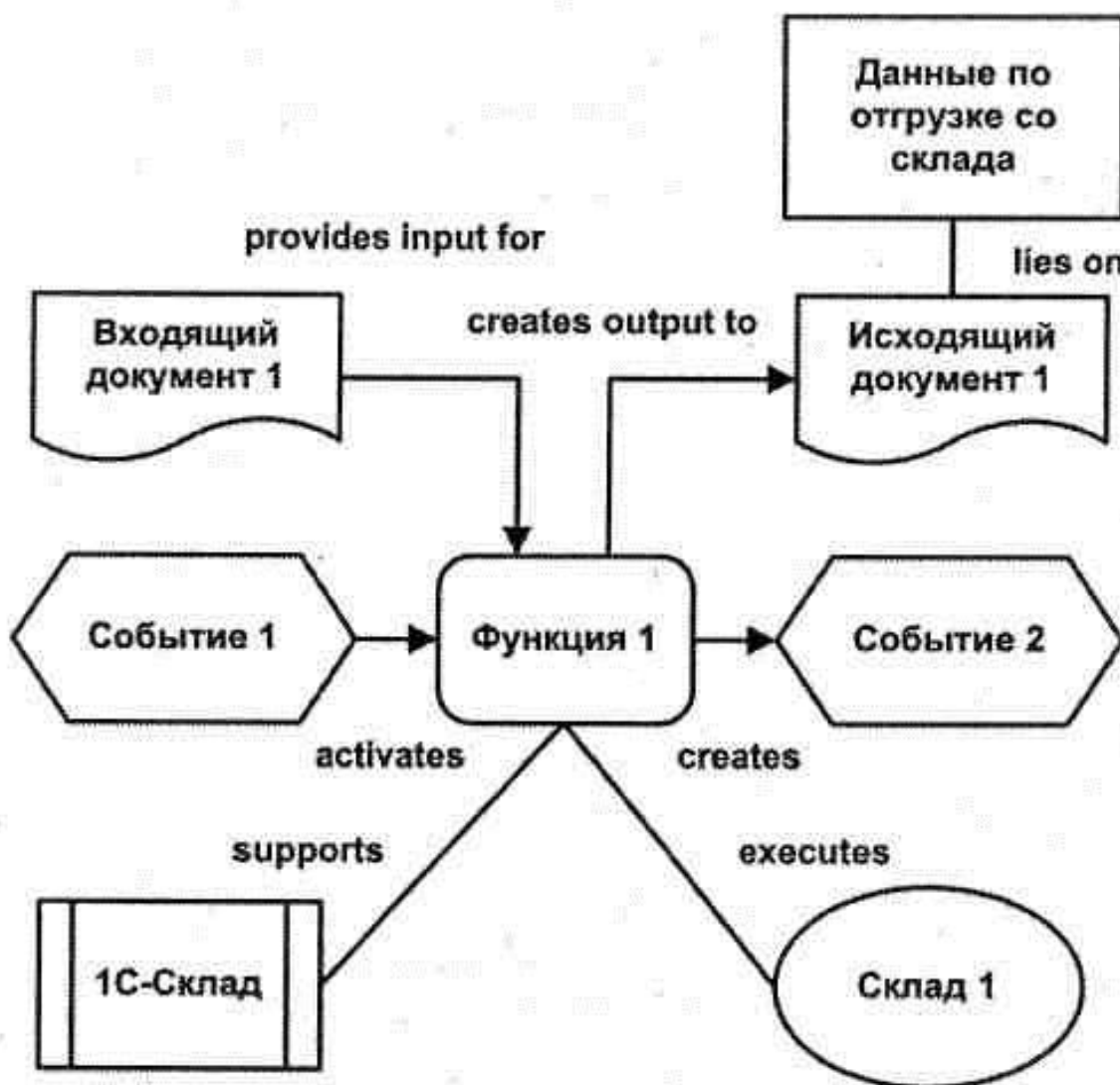


Рис. 3.2. Фрагмент модели бизнес-процесса

Из рис. 3.1 и 3.2 видно, что бизнес-процесс в нотации eEPC представляет собой поток последовательно выполняемых работ (процедур, функций), расположенных в порядке их выполнения. Реальная длительность выполнения процедур в eEPC визуально не отражается. Это приводит к тому, что при создании моделей возможны ситуации, когда на одного исполнителя будет возложено выполнение двух задач одновременно. Используемые при построении модели символы логики позволяют отразить ветвление и слияние бизнес-процесса. Для получения информации о реальной длительности процессов необходимо использовать другие инструменты описания, например графики Ганта в системе MS Project.

МЕТОД ERICSSON-PENKER¹

Метод Ericsson- Penker представляет интерес прежде всего в связи с попыткой применения языка объектного моделирования UML (изначально предназначенного для моделирования архитектуры систем ПО) в рамках процессного подхода к моделированию бизнес-процессов. Это стало возможным благодаря наличию в UML механизмов расширения (см. подразд. 2.5.8). Авторы метода создали свой профиль UML для моделирования бизнес-процессов, введя набор стереотипов, описывающих процессы, ресурсы, правила и цели деятельности организации.

Метод использует четыре основные категории бизнес-модели.

- **Ресурсы** — различные объекты, используемые или участвующие в бизнес-процессах (люди, материалы, информация или продукты). Ресурсы структурированы, взаимосвязаны и подразделяются на физические, абстрактные, информационные и человеческие.
- **Процессы** — виды деятельности, изменяющие состояние ресурсов в соответствии с бизнес-правилами.
- **Цели** — назначение бизнес-процессов. Цели могут быть разбиты на подцели и соотнесены с отдельными процессами. Цели достигаются в процессах и выражают требуемое состояние ресурсов. Цели могут быть выражены в виде одного или более правил.
- **Бизнес-правила** — условия или ограничения выполнения процессов (функциональные, поведенческие или структурные). Правила могут диктоваться внешней средой (инструкциями или законами), или могут быть определены в пределах бизнес-процессов. Правила могут быть определены с использованием языка OSM, который является частью стандарта UML.

Основной диаграммой UML, используемой в данном методе, является диаграм-

¹ Eriksson, Hans-Erik and Penker, Magnus «Business Modeling with UML: Business Patterns at work». Wiley Computer Publishing, 2000.

ма деятельности (см. подразд. 2.5.5). Процесс в самом простом виде может быть описан как множество деятельностей. Метод Eriksson-Penker представляет образец процесса на диаграмме деятельности (рис. 3.3) в виде деятельности со стереотипом «process» (в качестве основы данного образца использовано представление процесса в методе IDEF0, расширенное за счет введения цели процесса). Процесс использует входные ресурсы и формирует выходные ресурсы, показанные в виде объектов со стереотипом «resource», соединенных с процессом связями зависимости. Ресурсы, играющие в методе IDEF0 роли «управления» и «механизма», также соединены с процессом связями зависимости со стереотипами «supply» и «control». Цель процесса показана как объект со стереотипом «goal».

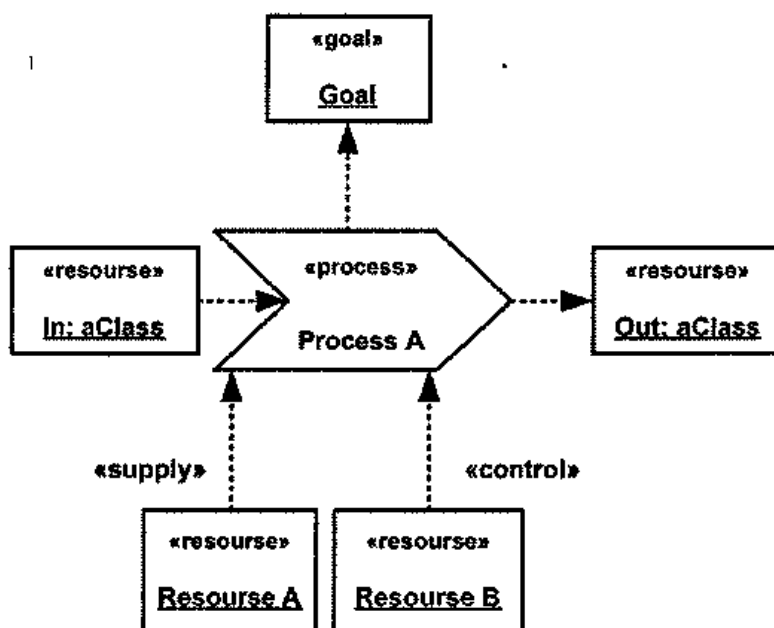


Рис. 3.3. Диаграмма деятельности для процесса

Полная бизнес-модель включает множество представлений, подобных представлениям архитектуры ПО. Каждое представление выражено в одной или более диаграммах. Диаграммы могут иметь различные типы и изображать процессы, правила, цели и ресурсы во взаимодействиях друг с другом. Метод Eriksson-Penker использует четыре различных представления бизнес-модели:

- концептуальное представление — структура целей и проблем (дерево целей, представленное в виде диаграммы объектов);

- представление процессов — взаимодействие между процессами и ресурсами (в виде набора диаграмм деятельности);
- структурное представление — структура организации и ресурсов (в виде диаграмм классов). В качестве примера одной из моделей этого представления можно привести образец Employment из подразд. 2.6;
- представление поведения — поведение отдельных ресурсов и детализация процессов (в виде диаграмм деятельности, состояний и взаимодействия).

3.2.5.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ПРОЦЕССНОГО

ПОДХОДА

В данном примере используется методика Йордона, описанная в подразд. 3.2.2.

Постановка задачи

Администрация больницы заказала разработку ИС для отдела приема пациентов и медицинского секретариата. Новая система предназначена для обработки данных о врачах, пациентах, приеме пациентов и лечении. Система должна выдавать отчеты по запросу врачей или администрации.

Во время обследования составлено следующее описание деятельности рассматриваемых подразделений.

Перед приемом в больницу проводится встреча пациента и врача. Врач информирует отдел приема пациентов об ожидаемом приеме больного и передает о нем данные. Пациент может быть принят в больницу более чем один раз, но если пациент ранее не лечился в больнице, то ему присваивается регистрационный номер и записываются его данные (фамилия, имя и отчество пациента, адрес и дата рождения). Пациент должен быть зарегистрирован в системе до приема в больницу.

Через некоторое время врач оформляет в отделе приема пациентов прием больного. При этом определяется порядковый номер приема и запоминаются данные

приема пациента. После этого отдел приема посылает сообщение врачу для подтверждения приема больного. В это сообщение включаются регистрационный номер пациента и его фамилия, порядковый номер приема, дата начала лечения и номер палаты.

В день приема пациент сообщает в отдел приема о своем прибытии и передает данные о себе (или изменения в данных). Отдел приема проверяет и при необходимости корректирует данные о пациенте. Если пациент не помнит свой регистрационный номер, то выполняется соответствующий запрос. После регистрации пациент получает регистрационную карту, содержащую фамилию, имя и отчество пациента, адрес, дату рождения, номер телефона, группу крови, название страховой компании и номер страховки.

Во время пребывания в больнице пациент может лечиться у нескольких врачей; каждый врач назначает один или более курсов лечения, но каждый курс лечения назначается только одним врачом. Данные о курсах лечения передаются в медицинский секретариат, который занимается координацией лечения пациентов, регистрируются и хранятся там. Данные включают номер врача, номер пациента, порядковый номер приема, название курса лечения, дату назначения, время и примечания.

При необходимости врач запрашивает в медицинском секретариате историю болезни пациента, содержащую данные о курсах лечения, полученных пациентом. История болезни представляется в виде табл. 3.1.

После окончания курсов лечения лечащий врач принимает решение о выписке пациента. Когда пациент выписывается, он сообщает об этом в отдел приема. Отдел приема регистрирует данные о выписке, включая дату выписки, и дает пациенту справку об окончании пребывания в больнице.

История болезни пациента

История болезни		Дата: 01-02-1998	
Данные с 01-01-1992 по 01-02-1998			
Номер пациента: 732	ФИО: Иванов С.П.	Дата рождения: 15-07-55	
Номер приема: 649	Дата приема: 06-07-1992	Палата: 4	
Заболевание: перелом ноги		Врач: Петров П.С, хирург	
	Дата	Время	Примечание
	06-07-1992	12.00	Наложено гипс
	20-07-1992	14.00	Состояние хорошее
	22-07-1992	10.00	Лечение закончено
Заболевание: ОРЗ		Врач: Иванов А.С., терапевт	
	Дата	Время	Примечание
	10-07-1992	10.00	Назначена терапия
	15-07-1992	10.00	Состояние удовлетворительное
	20-07-1992	10.00	Состояние хорошее
	25-07-1992	10.00	Выписан
Номер приема: 711	Дата приема: 12-04-1993	Палата: 8	
Заболевание: сердечный приступ		Врач: Сидоров И.И., кардиолог	
	Дата	Время	Примечание

Врач передает данные о себе и изменения в данных в отдел приема. Данные о врачах включают номер врача, фамилию, имя и отчество, адрес, дату рождения, домашний телефон, специализацию, номер кабинета, рабочий телефон.

Администрация больницы может запросить отчет о пациентах. В запросе ука-

зывается интервал времени (начальная и конечная даты). Отчет должен представляться в следующей форме (табл. 3.2).

Администрация больницы запрашивает также обзор заболеваний за последнюю неделю каждый понедельник в 9.00.

Описание контекста системы приема пациентов в больнице и построение начальной контекстной диаграммы

Построим начальную контекстную диаграмму потоков данных в нотации Гейна — Сэрсона (рис. 3.4). Нарисуем нулевой процесс и присвоим ему имя системы (Система приема пациентов). Поскольку моделируется деятельность отдела приема пациентов и медицинского секретариата, внешними сущностями являются Врач, Пациент и Администрация больницы. Нарисуем внешние сущности и соединим их с нулевым процессом посредством-С1 ном потоков данных.

Таблица 3.2

Отчет о пациентах

Отчет о пациентах						
Данные с 01-12-1997 по 01-02-1998						
Номер	ФИО	Адрес	Телефон	Дата рождения	Страховая компания	Номер страховки

Спецификация структур данных

Определим состав потоков данных и подготовим исходную информацию для конструирования концептуальной модели данных.

Пометим символами «*», «°» и «|» все структуры и элементы данных типа «итерация», «условное вхождение» и «альтернатива» соответственно.

После объединения структур и элементов данных в более крупные структуры для каждого потока данных должна быть сформирована структура данных.

Примеры спецификации структур данных.

ОТЧЕТ ПО ИСТОРИИ БОЛЕЗНИ

Текущая_дата

Начальная_дата

Конечная_дата

СТРОКА ПАЦИЕНТА

Номер_пациента

Фио

Дата_рождения

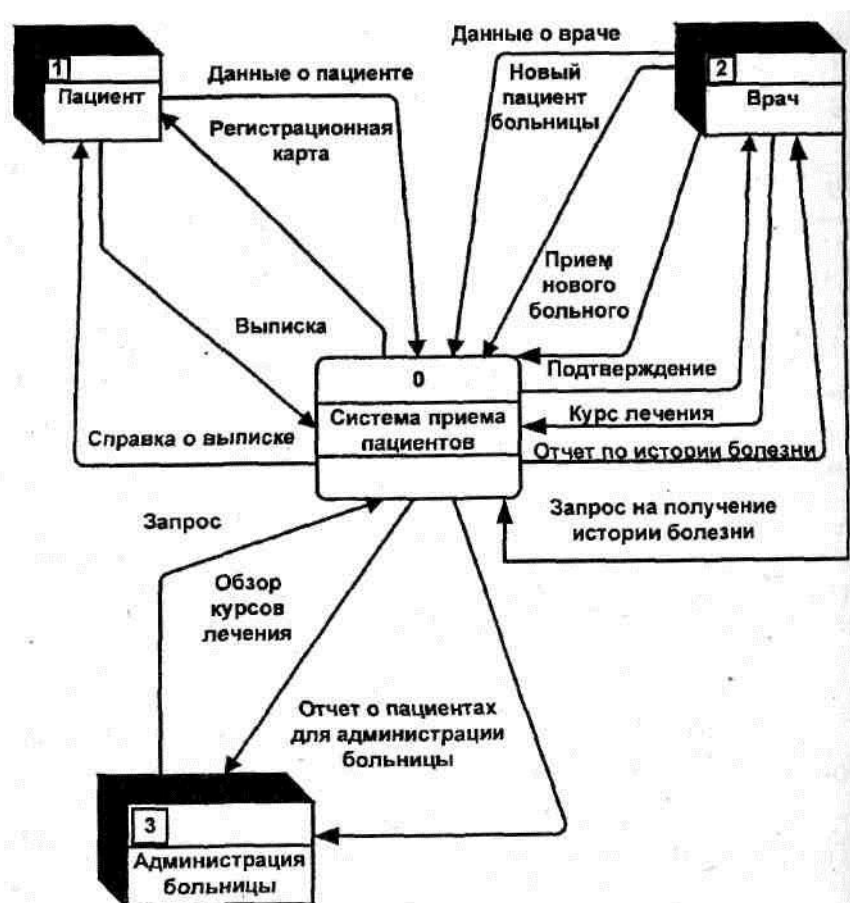


Рис. 3.4. Начальная контекстная диаграмма

СТРОКА ПРИЕМА*

Номер_приема /* порядковый номер приема нового больного*/

Дата_приема

Номер_палаты

СТРОКА КУРСА ЛЕЧЕНИЯ*

Наименование_заболевания

Имя_врача
Специализация
СТРОКА_РЕЗУЛЬТАТОВ_ЛЕЧЕНИЯ*
Дата
Время
Примечание

ИНФОРМАЦИЯ ОТ ВРАЧА

Присм_нового_больного
Данные_о_враче
Новый_пациент_больницы
Запрос_на_получение_истории_болезни
Курс_лечения

ОТЧЕТ О ПАЦИЕНТАХ

Текущая_дата
Начальная_дата
Конечная_дата
СТРОКА_ПАЦИЕНТА*
Номер_пациента
Фино
Адрес
Телефон
Дата_рождения
СТРАХОВАНИЕ⁰
Страховая_компания
Номер_страховки

Построение начального варианта концептуальной модели данных

Выделим и нарисуем сущности для каждого класса объектов данных в системе приема пациентов. Рассмотрим каждую возможную пару сущностей и установим существование связи между ними. Нарисуем диаграмму «сущность-связь». Присвоим наименование каждой связи и зададим ее характеристики

(рис. 3.5).

Предлагается ответить самостоятельно на следующие вопросы:

1. Почему мощность связи между Пациентом и Приемом, Приемом и Курсом лечения, Врачом и Курсом лечения равна 0, а не 1?
2. Почему отсутствует связь между Врачом и Пациентом?

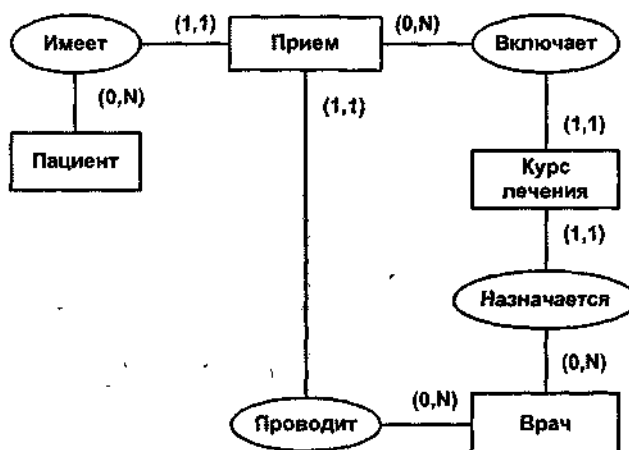


Рис. 3.5. Начальный вариант концептуальной модели данных

Построение диаграмм потоков данных нулевого и последующих уровней

Декомпозируем начальную контекстную диаграмму. Декомпозируем сложные процессы и проверим соответствие различных уровней модели процессов. Опишем накопители данных посредством структур данных. Опишем процессы нижнего уровня посредством спецификаций.

Результаты представлены на рис. 3.6—3.9.

Описание накопителей данных и пример спецификации процесса приведены ниже.

Накопитель данных: пациенты

Номер_пациента

Фино

Адрес

Телефон

Дата_рождения

Группа_крови

Страхование⁰

Накопитель данных: приемы

Номер_приема

Номер_пациента

Дата_приема

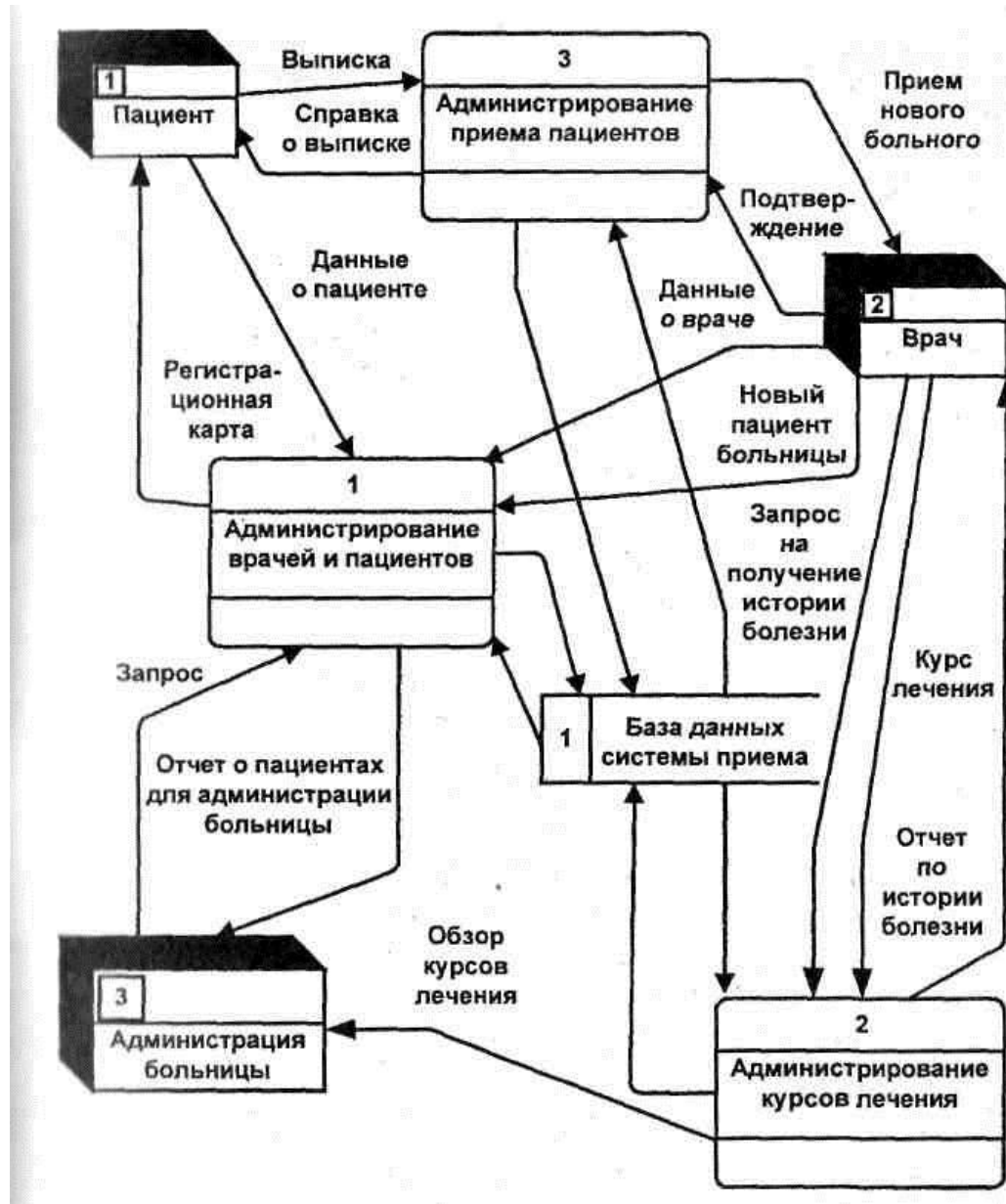


Рис. 3.6. Диаграмма потоков данных нулевого уровня

Дата_выписки

Номср_палаты

Накопитель данных: курсы лечения

Номер_приема

Номср_врача

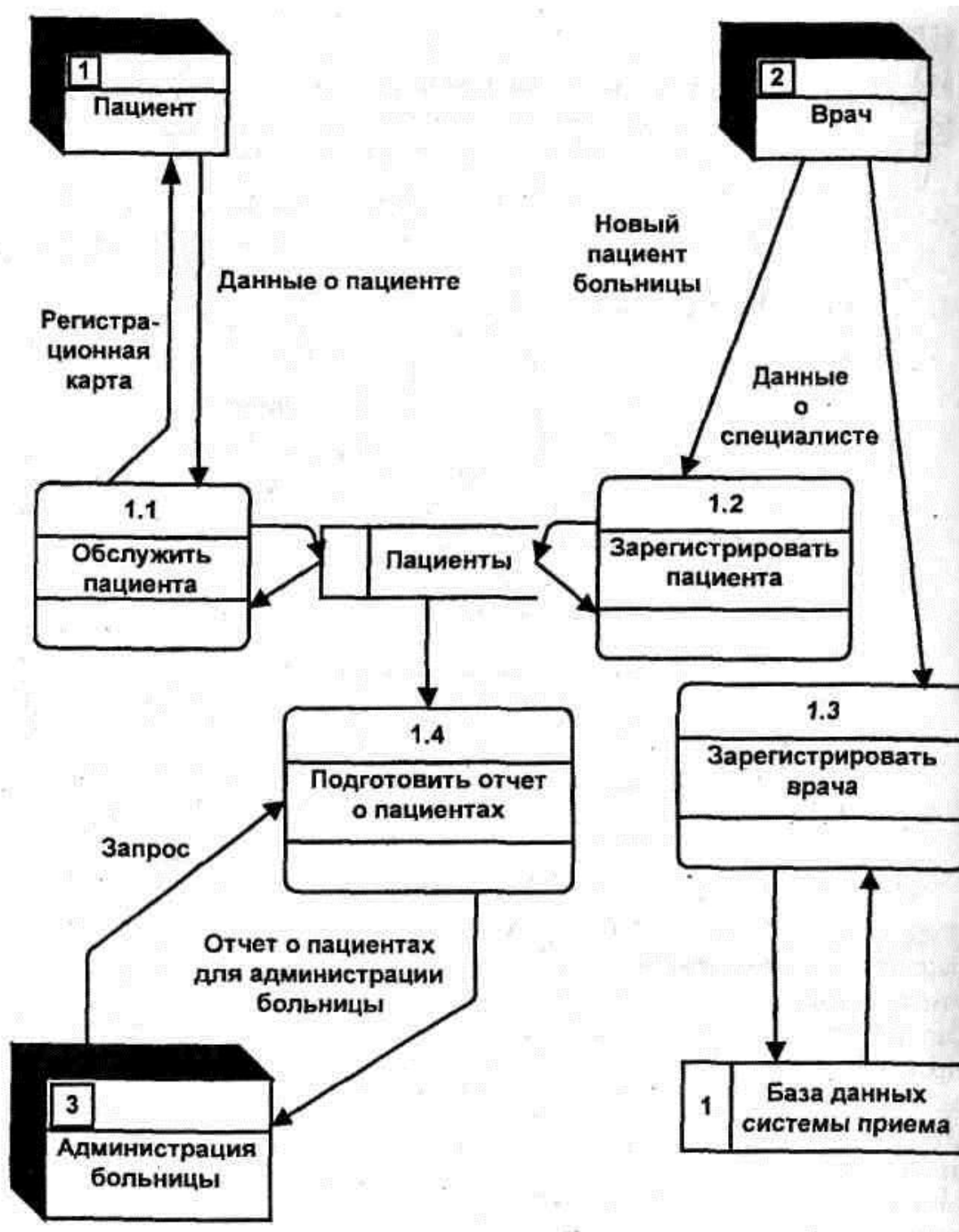


Рис. 3.7. Диаграмма потоков данных первого уровня для процесса 1

Наименование_заболевания

ПРИМЕЧАНИЕ

Дата

Время

Примечание

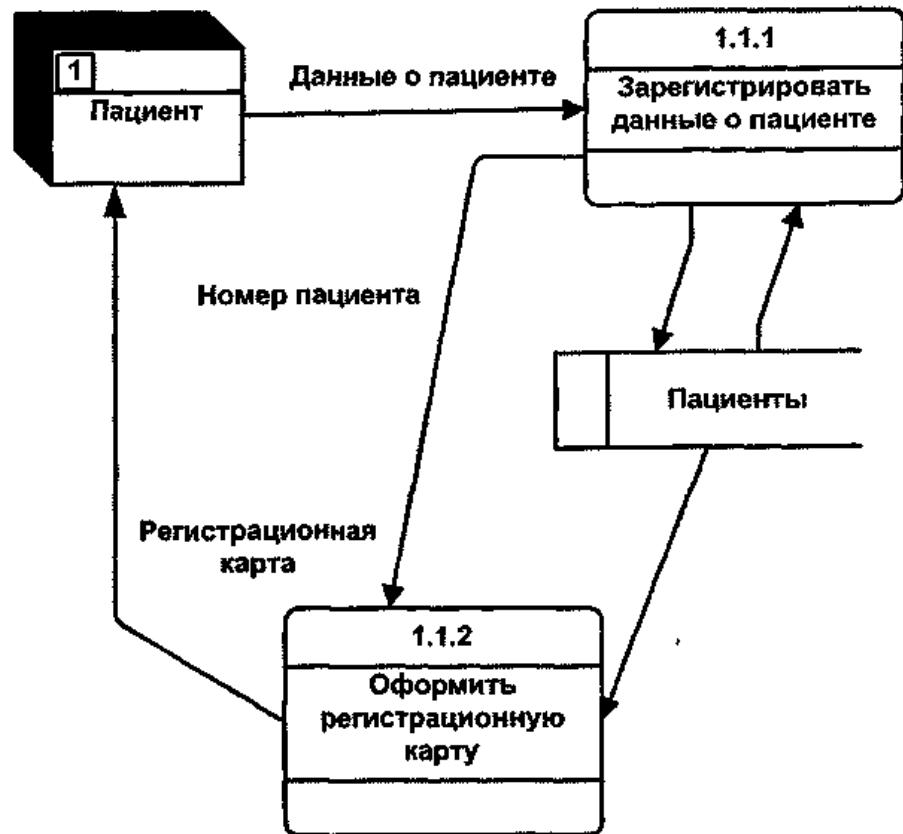


Рис. 3.8. Диаграмма потоков данных второго уровня для процесса 1.1

Накопитель данных: врачи

Номер_врача

Имя_врача

Адрес

Телефон

Дата_рождения

Специализация

Номер_кабинета

Рабочий_телефон

Спецификация процесса: Зарегистрировать пациента

BEGIN

GET Фио и Дата_рождения

FIND пациент

IF пациент найден THEN

DISPLAY данные пациента

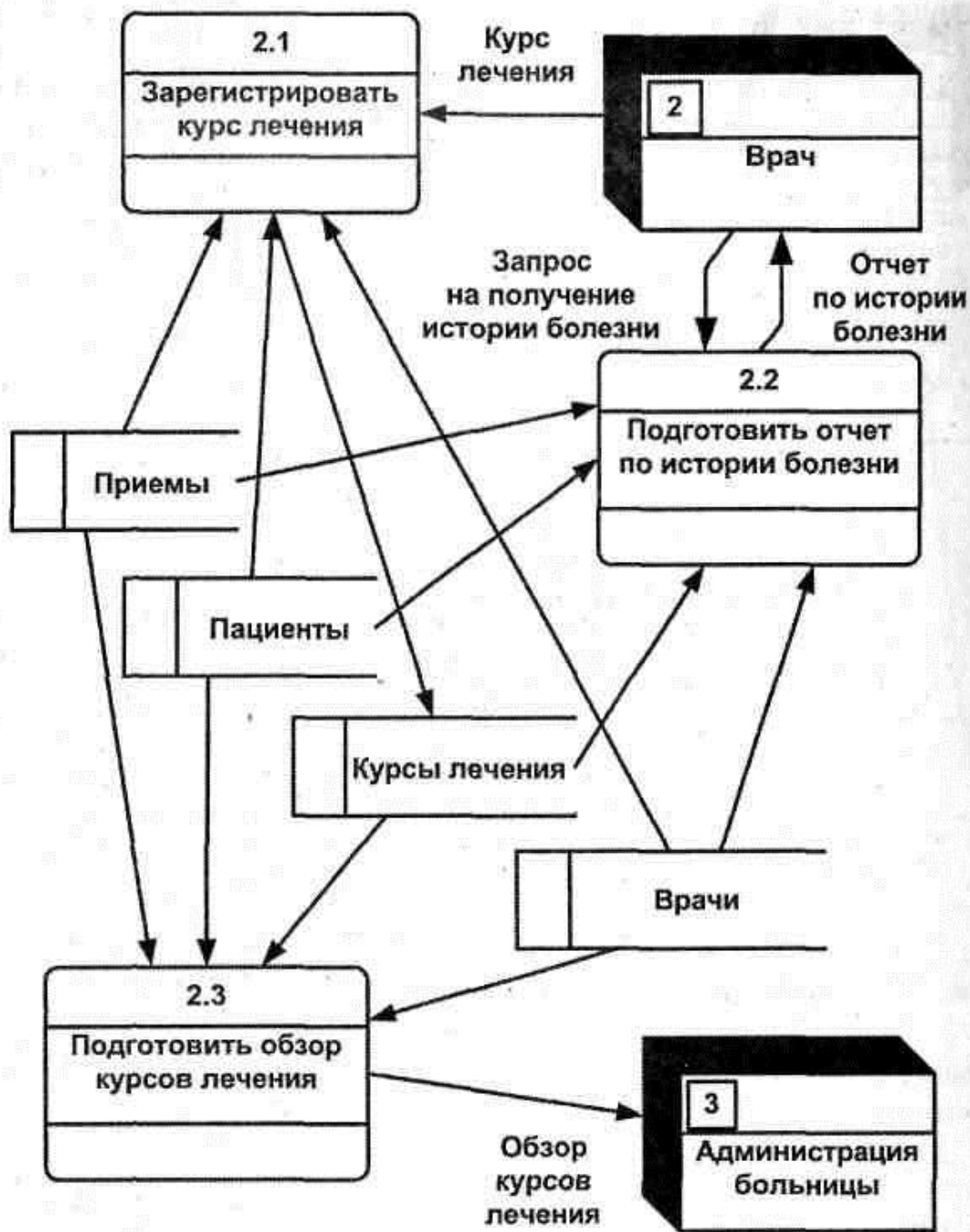


Рис. 3.9. Диаграмма потоков данных первого уровня для процесса 2

ELSE

GET Адрес

DETERMINE Номер_пациента

INSERT пациент

ENDIF

PRINT подтверждение

END

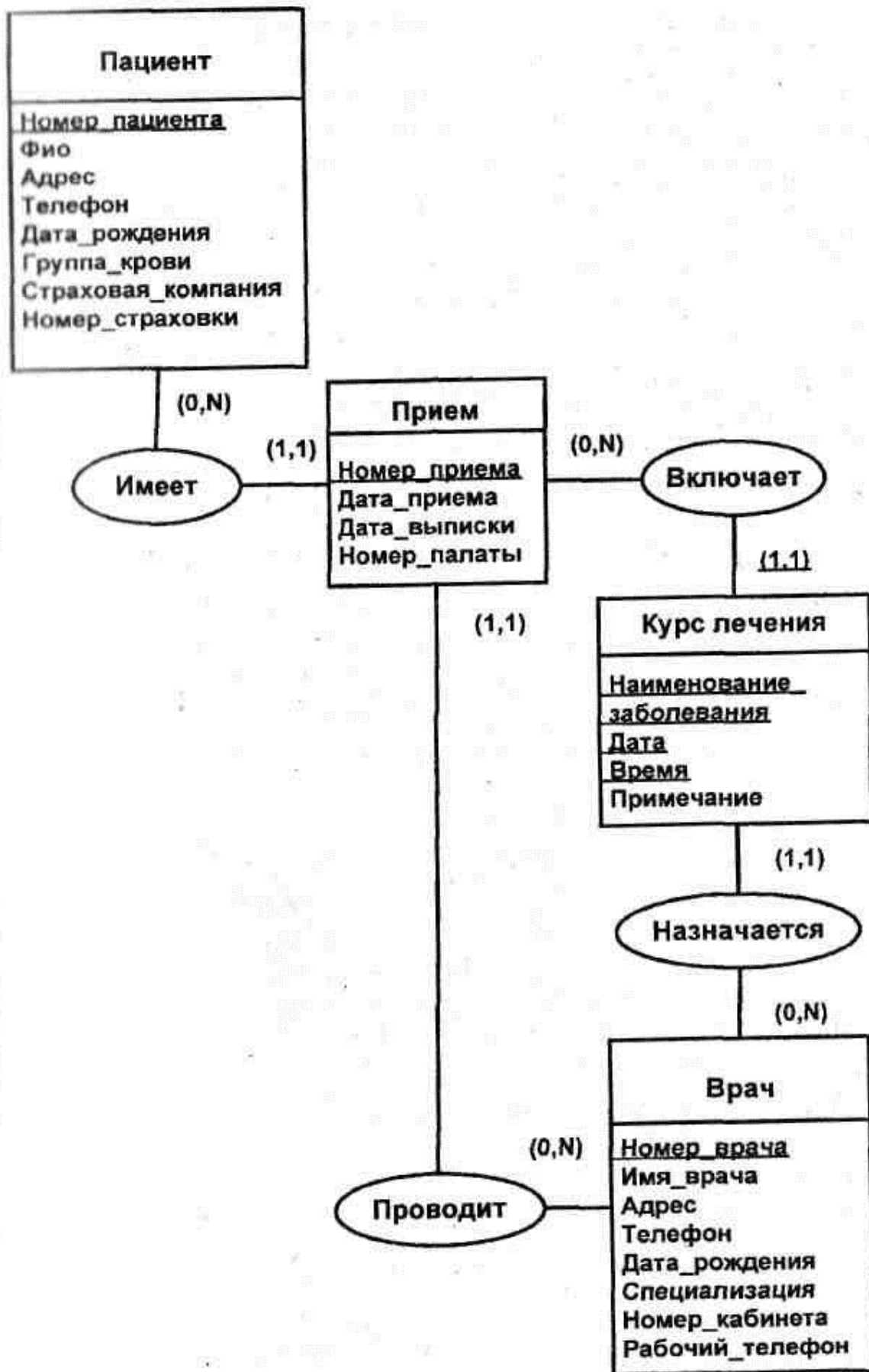


Рис. 3.10. Уточненный вариант концептуальной модели данных

Уточнение концептуальной модели данных

Используя построенные структуры данных, определим атрибуты сущностей и уточним построенную модель данных. Внешние ключи можно не показывать, поскольку они определяются связями между сущностями. Выделим атрибуты-идентификаторы и подчеркнем их.

Результат представлен на рис. 3.10.

3.3.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К МОДЕЛИРОВАНИЮ БИЗНЕС-ПРОЦЕССОВ

3.3.1.

МЕТОДИКА МОДЕЛИРОВАНИЯ

RATIONAL UNIFIED PROCESS

Объектно-ориентированный подход к моделированию бизнес-процессов с использованием языка UML реализован в технологии Rational Unified Process. Методика моделирования, являющаяся составной частью данной технологии, предусматривает построение двух моделей:

- модели бизнес-процессов (Business Use Case Model);
- модели бизнес-анализа (Business Analysis Model).

Модель бизнес-процессов — модель, описывающая бизнес-процессы организации в терминах ролей и их потребностей. Она представляет собой расширение модели вариантов использования UML за счет введения набора стереотипов Business Actor (стереотип действующего лица) и Business Use Case (стереотип варианта использования).

Business Actor (действующее лицо бизнес-процессов) — это некоторая роль, внешняя по отношению к бизнес-процессам организации. Потенциальны-

ми кандидатами в действующие лица, бизнес-процессов являются:

- акционеры;
- заказчики;
- поставщики;
- партнеры;
- потенциальные клиенты;
- местные органы власти;
- сотрудники подразделений организации, деятельность которых не охвачена моделью;
- внешние системы.

Список действующих лиц составляется путем ответа на следующие вопросы:

Кто извлекает пользу из существования организации?

Кто помогает организации осуществлять свою деятельность?

Кому организация передает информацию и от кого получает?

Business Use Case (вариант использования с точки зрения бизнес-процессов) определяется как описание последовательности действий (потока событий) в рамках некоторого бизнес-процесса, приносящий ощутимый результат конкретному действующему лицу.

Это определение подобно общему определению бизнес-процесса, но имеет более точный смысл. В терминах объектной модели Business Use Case представляет собой класс, объектами которого являются конкретные потоки событий в рамках описываемого бизнес-процесса.

Донная методика концентрирует внимание в первую очередь на элементарных бизнес-процессах. Такой процесс можно определить как задачу, выполняемую одним человеком в одном месте в одно время в ответ на некоторое событие, приносящую конкретный результат и переводящую данные в некоторое устойчивое состояние (например, подтверждение платежа по кредитной карте). Выполнение такой задачи обычно включает от пяти до десяти шагов и может занимать от нескольких минут до нескольких дней, но рассматривается

как один сеанс взаимодействия действующего лица с исполнителями.

Каждый Business Use Case отражает цель или потребность некоторого действующего лица. Например, если рассмотреть процесс регистрации пассажиров в аэропорту (рис. 3.11¹), то его основным действующим лицом будет сам Пассажир, главная цель которого в данном процессе — пройти регистрацию. Эта цель моделируется в виде Business Use Case с наименованием «Пройти регистрацию». Другим действующим лицом является Руководитель туристической группы, регистрирующий группу пассажиров. Стереотипы связей явно показывают роль действующих лиц по отношению к вариантам использования.

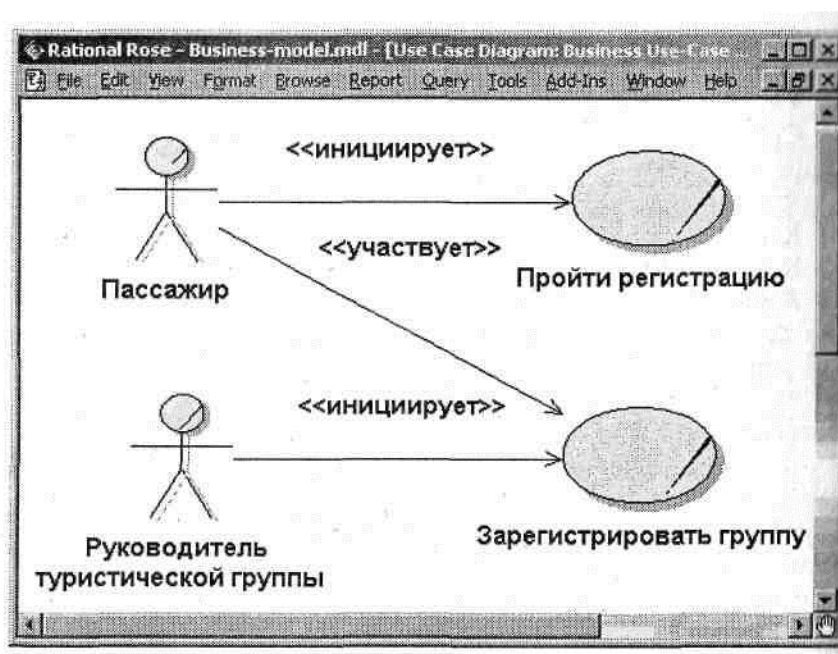


Рис. 3.11. Диаграмма вариантов использования для процесса регистрации пассажиров в аэропорту

Описание Business Use Case представляет собой спецификацию, которая, подобно обычному варианту использования, состоит из следующих пунктов:

- наименование;

¹ Примеры моделей, связанных с применением технологии Rational Unified Process, здесь и далее приводятся в среде CASE-средства Rational

- краткое описание;
- цели и результаты (с точки зрения действующего лица);
- описание сценариев (основного и альтернативных);
- специальные требования (ограничения по времени выполнения или другим ресурсам);
- расширения (исключительные ситуации);
- связи с другими Business Use Case;
- диаграммы деятельности (для наглядного описания сценариев при необходимости).

Пример спецификации Business Use Case:

Наименование — пройти регистрацию.

Краткое описание — данный Business Use Case реализует процесс регистрации пассажира на рейс.

Цели — получить посадочный талон и сдать багаж.

Основной сценарий.

1. Пассажир встает в очередь к стойке регистратора.
2. Пассажир предъявляет билет регистратору.
3. Регистратор подтверждает правильность билета.
4. Регистратор оформляет багаж.
5. Регистратор резервирует место для пассажира.
6. Регистратор печатает посадочный талон.
7. Регистратор выдает пассажиру посадочный талон и квитанцию на багаж.
8. Пассажир уходит от стойки регистратора.

Альтернативные сценарии.

3а. Билет неправильно оформлен — регистратор отправляет пассажира к агенту по перевозкам.

4а. Багаж превышает установленный вес — регистратор оформляет доплату.

Специальные требования - время регистрации не должно превышать одной минуты.

Описание Business Use Case может сопровождаться целью процесса, кото-

рая, так же, как и в методе Eriksson-Penker, моделируется с помощью класса со стереотипом «goal», а дерево целей изображается в виде диаграммы классов.

Для каждого Business Use Case строится модель бизнес-анализа - объектная модель, описывающая реализацию бизнес-процесса и терминах взаимодействующих объектов (бизнес-объектов — Business Object), принадлежащих к двум классам, — Business Worker и Business Entity.

Business Worker (исполнитель) — активный класс, представляющий собой абстракцию исполнителя, выполняющего некоторые действия в рамках бизнес-процесса. Исполнители взаимодействуют между собой и манипулируют различными сущностями, участвуя в реализациях сценариев Business Use Case. На диаграмме классов UML исполнитель представляется в виде класса со стереотипом «business worker». Например, если рассмотреть Business Use Case «Пройти регистрацию», можно определить в нём двух исполнителей — Регистратора и Координатора багажа.

Business Entity (сущность) — пассивный класс, не инициализирующий никаких взаимодействий. Объект такого класса может участвовать в реализациях различных Business Use Case. Сущность является объектом различных действий со стороны исполнителей. Понятие Business Entity аналогично понятию сущности в модели ERM, за исключением того, что в ERM не определяется поведение сущности, а в объектной модели сущность может иметь набор обязанностей. На диаграмме классов UML сущность представляется в виде класса со стереотипом «business entity». Например, в Business Use Case «Пройти регистрацию» можно определить следующие сущности: Билет, Рейс, Авиакомпания, Багаж, Багажная бирка.

Модель бизнес-анализа может состоять из диаграмм разных типов. В состав модели обязательно должна входить диаграмма классов, содержащая исполнителей и сущности. Пример такой диаграммы для Business Use Case «Пройти регистрацию» приведен на рис. 3.12.

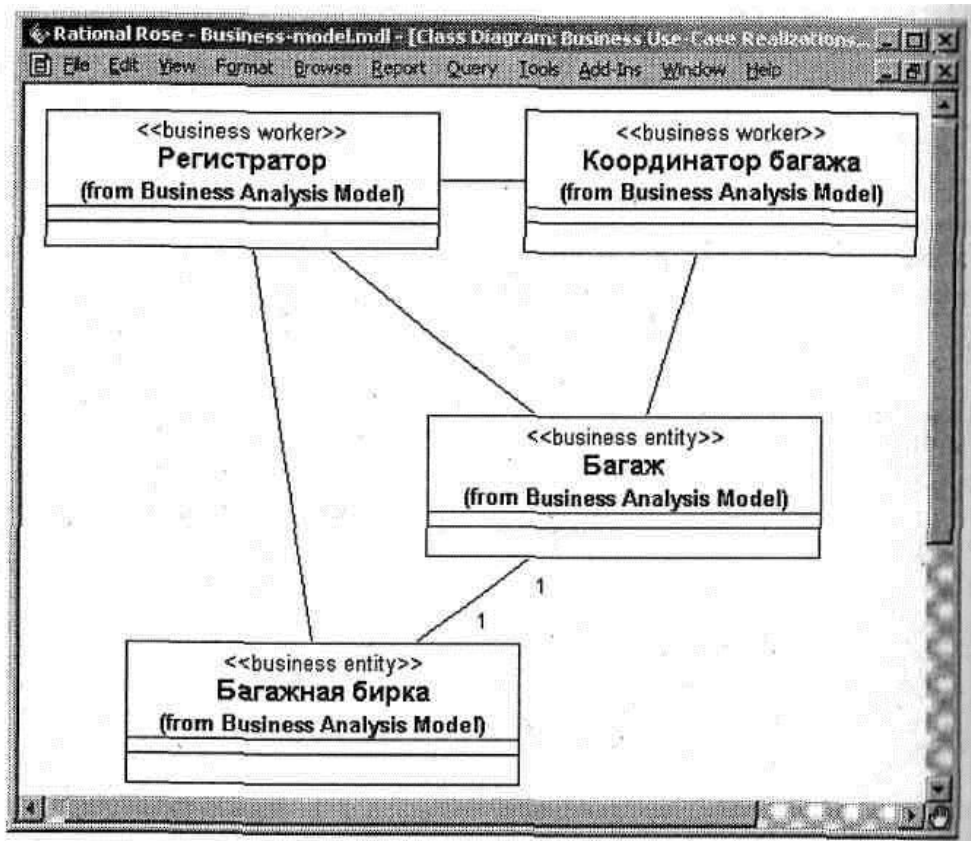


Рис. 3.12. Диаграмма классов модели бизнес-анализа

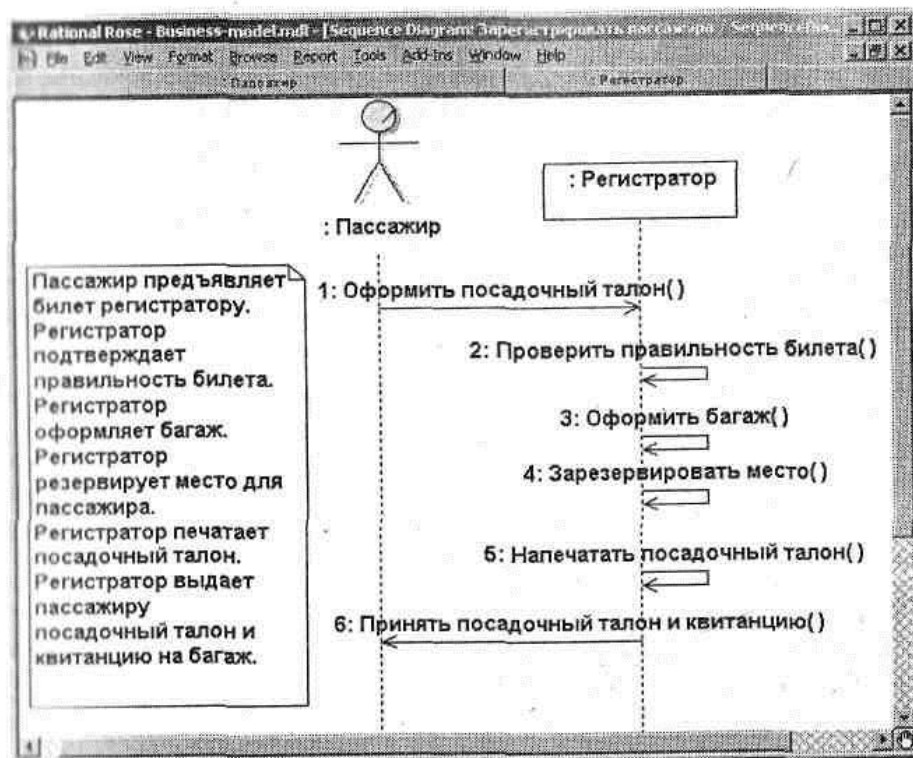


Рис. 3.13. Диаграмма последовательности для основного сценария Business Use Case «Пройти регистрацию»

На данной диаграмме ассоциации между классами-исполнителями отражают наличие взаимодействия между реальными исполнителями (Регистратором и Координатором багажа). Ассоциации между классами-исполнителями и классами-сущностями показывают, какими именно объектами манипулируют конкретные исполнители (Регистратор имеет дело с Багажом и Багажной биркой, а Координатор багажа - только с Багажом). Ассоциации между классами-сущностями отражают различные структурные связи (к одному месту багажа прикрепляется одна багажная бирка).

Кроме диаграммы классов, модель бизнес-анализа может включать:

- Диаграммы последовательности (и кооперативные диаграммы), описывающие сценарии Business Use Case в виде последовательности обмена сообщениями между объектами действующими лицами и объектами-исполнителями. Такие диаграммы помогают явно определить в модели обязанности каждого исполнителя в виде набора его операций. Пример диаграммы последовательности, описывающей основной сценарий Business Use Case «Пройти регистрацию», приведен на рис. 3.13. Модифицированная диаграмма классов модели бизнес-анализа с операциями приведена на рис. 3.14.
- Диаграммы деятельности с потоками объектов и «плавательными дорожками», описывающие взаимосвязи между сценариями одного или различных Business Use Case. Пример такой диаграммы для процесса заказа товаров в торговой компании приведен на рис. 3.15.
- Диаграммы состояний, описывающие поведение отдельных бизнес-объектов (например, для сущности «Багаж» можно описать переходы между состояниями «Определен вес», «Зарегистрирован», «Находится на транспортере» и т.д.).

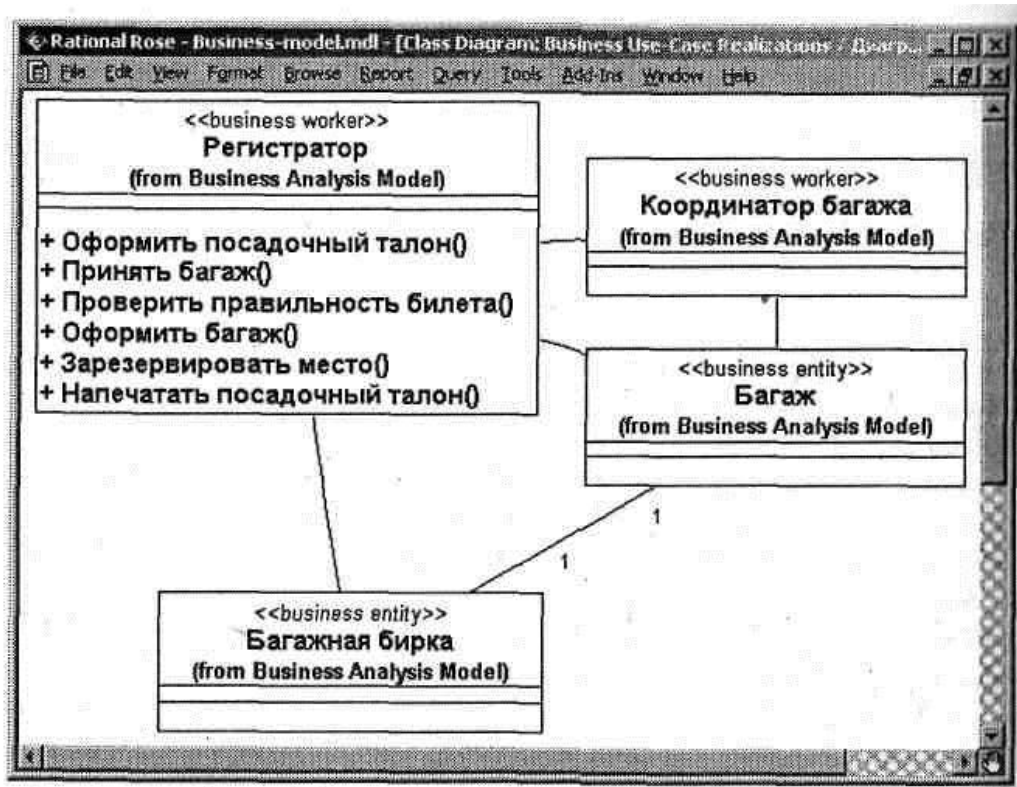


Рис. 3.14. Модифицированная диаграмма классов модели бизнес-анализа с операциями

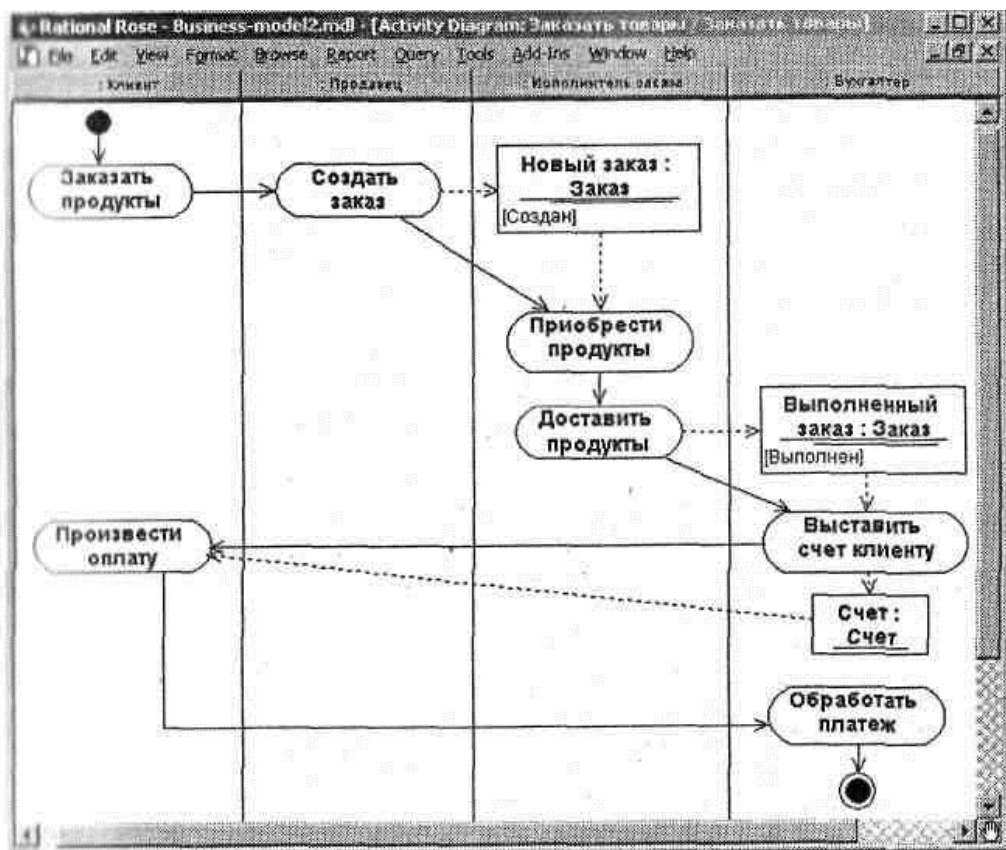


Рис. 3.15. Диаграмма деятельности для процесса заказа товаров

Методика моделирования Rational Unified Process предусматривает специальное соглашение, связанное с группировкой структурных элементов и диаграмм бизнес-модели. Это соглашение включает следующие правила:

- Все действующие лица, варианты использования и диаграммы вариантов использования для бизнес-процессов помещаются в пакет с именем Business Use Case Model.
- Все классы и диаграммы моделей бизнес-анализа помещаются в пакет с именем Business Analysis Model.
- Если моделируется деятельность более чем одного подразделения организации, то совокупность всех классов-исполнителей и классов-сущностей из моделей бизнес-анализа для различных Business Use Case разделяется на пакеты, соответствующие этим подразделениям. Этим пакетам присваиваются наименования подразделений (например, Бухгалтерия, Отдел доставки и т.п.) и стереотип «**organization unit**» (организационная единица).
- Диаграммы модели бизнес-анализа, относящиеся к конкретному Business Use Case (диаграммы классов, последовательности, деятельности и состояний) помещаются в кооперацию (см. подразд. 2.6) с именем данного Business Use Case и стереотипом «**business use-case realization**» (реализация бизнес-процесса). Все кооперации помещаются в пакет с именем Business Use Case Realizations.

Пример структуры бизнес-модели для процесса регистрации пассажиров в аэропорту приведен на рис. 3.16.

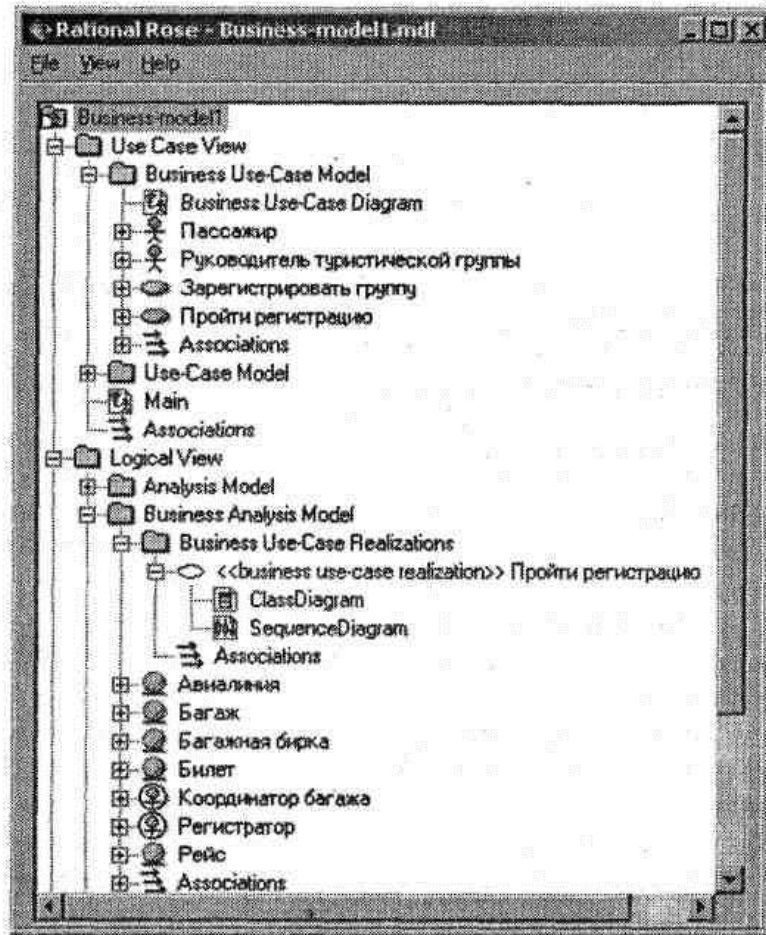


Рис. 3.16. Пример структуры бизнес-модели

Методика моделирования Rational Unified Process обладает следующими достоинствами:

- модель бизнес-процессов строится вокруг участников процессов (заинтересованных лиц) и их целей, помогая выявить все потребности клиентов организации. Нетрудно заметить, что такой подход в наибольшей степени применим для организаций, работающих в сфере оказания услуг (торговые организации, банки, страховые компании и т.д.);
- моделирование на основе вариантов использования способствует хорошему пониманию бизнес-модели со стороны заказчиков.

Однако следует отметить, что при моделировании деятельности крупной организации, занимающейся как производством продукции, так и оказанием услуг, необходимо применять различные методики моделирования, поскольку

для моделирования производственных процессов более предпочтительным является процессный подход (например, метод Eriksson-Penker).

3.3.2.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

В данном примере используется методика Rational Unified Process, описанная в предыдущем подразделе.

Постановка задачи

Перед руководителем информационной службы университета ставится задача разработки автоматизированной системы регистрации студентов на дополнительные платные курсы. Система должна позволять студентам регистрироваться на курсы и просматривать свои таблицы успеваемости с персональных компьютеров, подключенных к локальной сети университета. Профессора должны иметь доступ к системе, чтобы указать курсы, которые они будут читать, и проставить оценки за курсы.

В настоящее время в университете функционирует база данных, содержащая всю информацию о курсах (каталог курсов). Регистрация на курсы происходит следующим образом: в начале каждого семестра студенты могут запросить у регистратора каталог курсов, содержащий список курсов, предлагаемых в данном семестре. Информация о каждом курсе должна включать имя профессора, наименование кафедры и требования к предварительному уровню подготовки (прослушанным курсам).

Студент может выбрать 4 курса в предстоящем семестре. В дополнение к этому каждый студент может указать 2 альтернативных курса на тот случай, если какой-либо из выбранных им курсов окажется уже заполненным или отмененным. На каждый курс может записаться не более 10 и не менее 3 студентов (если менее 3, то курс будет отменен). В каждом семестре существует пери-

од времени, когда студенты могут изменить свои планы (добавить или отказаться от выбранных курсов). После того, как процесс регистрации некоторого студента завершен, регистратор направляет информацию в расчетную систему, чтобы студент мог внести плату за семестр. Если курс окажется заполненным в процессе регистрации, студент должен быть извещен об этом до окончательного формирования его личного учебного плана. В конце семестра студенты могут просмотреть свои таблицы успеваемости.

Создание модели бизнес-процессов

Действующие лица:

- Студент — записывается на курсы и просматривает свой таблицу успеваемости.
- Профессор — выбирает курсы для преподавания и ставит оценки за курсы.
- Расчетная система - получает информацию по оплате за курсы.
- Каталог курсов — база данных, содержащая информацию о курсах.

Варианты использования:

Исходя из потребностей действующих лиц, выделяются следующие варианты использования (Business Use Case):

- Зарегистрироваться на курсы;
- Просмотреть таблицу успеваемости;
- Выбрать курсы для преподавания;
- Проставить оценки.

Диаграмма вариантов использования для модели бизнес-процессов показана на рис. 3.17.

Пример спецификации Business Use Case: «Зарегистрироваться на курсы»:

Наименование:

Зарегистрироваться на курсы

Краткое описание

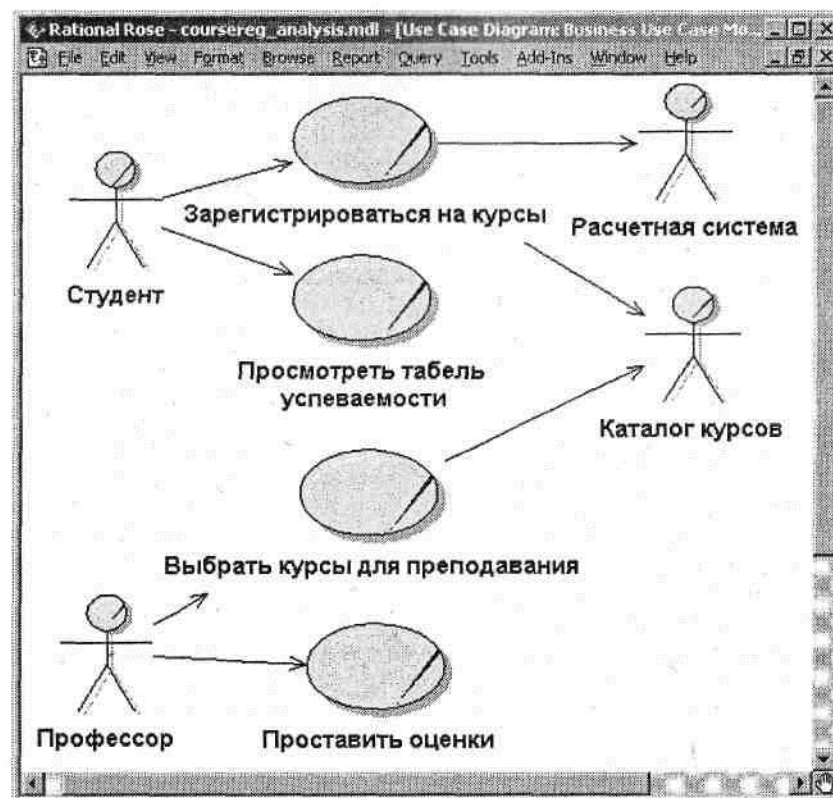


Рис. 3.17. Диаграмма вариантов использования для модели бизнес-процессов

Данный Business Use Case позволяет студенту зарегистрироваться на предлагаемые курсы в текущем семестре. Студент может изменить свой выбор, если изменение выполняется в установленное время в начале семестра.

Основной сценарий:

1. Студент приходит к регистратору и просит зарегистрировать его на предлагаемые курсы или изменить свой график курсов.
2. В зависимости от запроса студента, выполняется один из подчиненных сценариев (создать график или изменить график).

Подчиненный сценарий «Создать график»:

1. Регистратор выполняет поиск в каталоге доступных в настоящий момент курсов и выдает студенту их список.
2. Студент выбирает из списка 4 основных и 2 альтернативных курса.
3. Регистратор формирует график студента.
4. Выполняется подчиненный сценарий «Принять график».

Подчиненный сценарий «Изменить график»:

1. Регистратор находит текущий график студента.
2. Регистратор выполняет поиск в каталоге доступных в настоящий момент курсов и выдает студенту их список.
3. Студент может изменить свой выбор курсов, удаляя или добавляя конкретные курсы.
4. После выбора регистратор обновляет график.
5. Выполняется подчиненный сценарий «Принять график».

Подчиненный сценарий «Принять график»:

1. Для каждого выбранного студентом курса регистратор подтверждает выполнение студентом предварительных требований (прохождение определенных курсов), факт открытия предлагаемого курса и отсутствие конфликтов графика.
2. Регистратор вносит студента в список каждого выбранного предлагаемого курса. Курс фиксируется в графике.

Альтернативные сценарии.

Не выполнены предварительные требования, курс заполнен или имеют место конфликты графика.

Если во время выполнения подчиненного сценария «Принять график» регистратор обнаружит, что студент не выполнил необходимые предварительные требования, или выбранный им предлагаемый курс заполнен (уже записалось 10 студентов), или имеют место конфликты графика (два или более курсов с совпадающим расписанием), то он предлагает студенту изменить свой выбор курсов либо отменить формирование графика и вернуться к нему позже.

Система каталога курсов недоступна:

Если во время поиска в каталоге курсов окажется, что невозможно установить связь с системой каталога курсов, то регистрацию придется прервать и дождаться восстановления связи.

Регистрация на курсы закончена:

Если в самом начале выполнения регистрации окажется, что регистрация на текущий семестр уже закончена, то процесс завершится.

Создание модели бизнес-анализа

Исполнители.

- Регистратор — формирует учебный план и каталог курсов, записывает студентов на курсы, ведет все данные о курсах, профессорах, успеваемости и студентах.

Сущности.

- Студент.
- Профессор.
- График студента (список курсов).
- Курс (в программе обучения).
- Конкретный курс (курс в расписании).

Диаграмма классов для модели бизнес-анализа, описывающей Business Use Case «Зарегистрироваться на курсы», приведена на рис. 3.18.

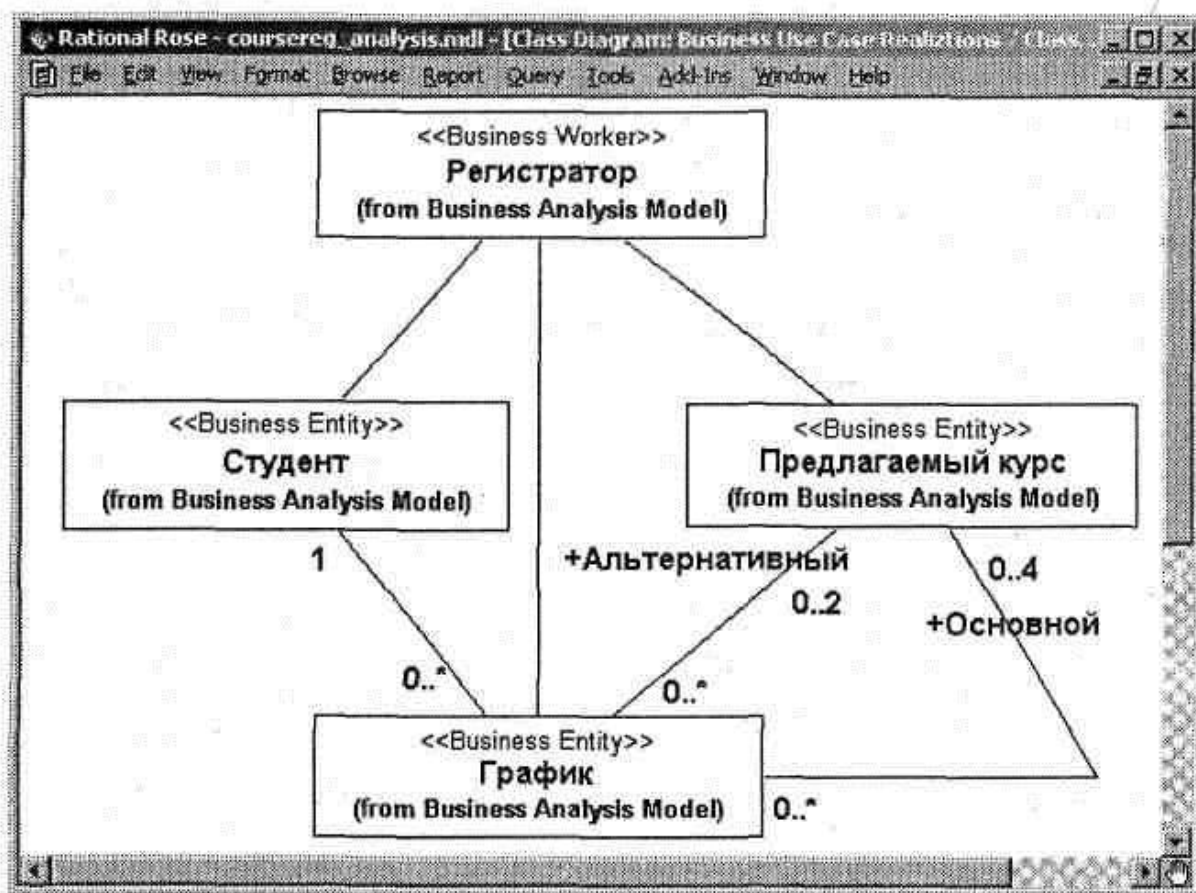


Рис. 3.18. Диаграмма классов модели бизнес-анализа

СПЕЦИФИКАЦИЯ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ¹

3.4.1.

ОСНОВЫ СПЕЦИФИКАЦИИ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Спецификация требований к ПО является составной частью процесса управления требованиями (см. подразд. 1.5). В качестве повторения отметим, что все требования к ПО делятся на функциональные и нефункциональные. **Функциональные требования** определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией. **Нефункциональные требования** описывают атрибуты системы (технические характеристики) и ее окружения.

Для выявления требований используются (в различных сочетаниях) следующие методы:

- собеседование (интервьюирование);
- анкетирование;
- моделирование и анализ бизнес-процессов (далее будет рассмотрена методика перехода от объектной бизнес-модели к системным требованиям);
- сессии по выявлению требований (мозговой штурм);
- создание и демонстрация пользователям работающих прототипов приложений (для выявления замечаний и дополнительных требований).

Собеседование — наиболее часто используемый и обычно наиболее полезный метод выявления требований. При использовании интервью может быть

¹ Методической основой данного подраздела является технология Rational Unified Process.

поставлено несколько целей, таких, как выяснение, проверка и толкование фактов, формирование заинтересованности, привлечение пользователей к работе, сбор предложений и мнений. Но метод собеседования требует для эффективного контакта хороших навыков общения с людьми, имеющими различные ценности, приоритеты, мнения, побуждения и индивидуальные особенности. Преимущества метода собеседования заключаются в следующем:

- опрашиваемое лицо может свободно и открыто отвечать на вопросы и почувствовать себя участником проекта;
- лицо, проводящее собеседование, может наблюдать за поведением опрашиваемого лица, изменять ход опроса, переформулировать или иначе строить вопросы во время собеседования.

Недостатки метода собеседования:

- метод трудоемкий и дорогой, поэтому может оказаться непрактичным;
- успех собеседования зависит от навыков общения лица, проводящего собеседование, и от желания опрашиваемых лиц участвовать в интервью.

Существуют два типа интервью: неструктурированное и структурированное. *Неструктурированные интервью* проводят с одной общей целью, которая явно не высказывается, и с помощью нескольких общих вопросов. Лицо, проводящее собеседование, рассчитывает на то, что опрашиваемое лицо должно само определять рамки и направление интервью. В таких интервью часто теряется нить рассуждений, и по этой причине они редко используются при проектировании систем.

В *структурированных интервью* лицо, проводящее собеседование, заранее подготавливает конкретный ряд вопросов к опрашиваемому лицу. В зависимости от ответов в процессе собеседования могут быть заданы дополнительные вопросы для уточнения или разъяснения некоторых тем. *Вопросы без подразумеваемых ответов* позволяют опрашиваемому лицу выбрать ответ, наиболее подходящий с его точки зрения. Вопросы, *допускающие единственный ответ*, ограничивают выбор возможного ответа или требуют коротких, прямых ответов.

Анкетирование позволяет с помощью анкет получать сведения от большого количества людей, контролируя правильность их ответов. При работе с большой аудиторией этот метод наиболее эффективен. Преимущества метода анкетирования заключаются в следующем:

- люди могут заполнять и возвращать анкеты в удобное для них время;
- люди склонны сообщать в ответах действительные факты, если анкетирование анонимное;
- это относительно недорогой способ сбора данных с участием большого количества людей.

Недостатки метода анкетирования:

- не все могут согласиться ответить на вопросы, анкеты могут возвращаться незаполненными;
- нет возможности пояснить или переформулировать неправильно понятые вопросы, наблюдать и анализировать реакцию респондента на отдельные вопросы;
- подготовка анкет может потребовать много времени.

Мозговой штурм особенно полезен в ситуациях, когда обсуждаются новые, нестандартные решения незнакомой проблемы. Большинство новаторских идей очень часто бывают результатом комбинации нескольких, на первый взгляд, не связанных друг с другом идей. Мозговой штурм включает в себя генерацию идей и расстановку приоритетов. Процесс генерации идей обычно подчиняется правилам, соблюдение которых преследует одну цель — выработку как можно большего числа идей. Критика и споры в этот период не поощряются; существующие пределы и ограничения снимаются, можно изменять и комбинировать идеи.

Основная идея **создания прототипа** состоит в быстрой постройке модели системы, которая, как предполагается, нужна пользователю. Обычно в прототипах многие детали опускаются, в том числе процедуры контроля входных данных, обработки ошибок, резервного копирования и восстановления данных. Не учитываются также производительность и масштабируемость. Создание

прототипа никак не противоречит и не препятствует применению других методов получения требований от пользователя. Однако иногда только с помощью прототипа удастся заставить клиента начать обсуждение требований, которые при другом подходе кажутся слишком абстрактными. Кроме того, работа с прототипом часто помогает получить требования, которые в противном случае так и остались бы неизвестными. Прототип хорош еще и тем, что он является неким осязаемым доказательством (или иллюзией) прогресса в разработке системы. А в некоторых случаях он может даже поддерживать какие-то ограниченные рабочие возможности системы. К сожалению, как отмечалось в главе 1, реальный опыт работы с прототипами свидетельствует, что их создание обычно препятствует любому официальному документированию требований, следовательно, требования формулируются только в виде кода. Детали работы системы, игнорируемые прототипом (обработка ошибок, резервное копирование и др.), могут так и не появиться в виде требований.

Выявленные в результате применения перечисленных методов требования к ПО оформляются в виде ряда документов и моделей. К основным документам, регламентируемым технологией Rational Unified Process, относятся:

- Концепция — определяет глобальные цели проекта и основные особенности разрабатываемой системы. Существенной частью концепции является постановка задачи разработки, определяющая требования к выполняемым системой функциям.
- Словарь предметной области (глоссарий) — определяет общую терминологию для всех моделей и описаний требований к системе. Глоссарий предназначен для описания терминологии предметной области и может быть использован как словарь данных системы.
- Дополнительные спецификации (технические требования) - содержит описание нефункциональных требований к системе, таких, как надежность, удобство использования, производительность, сопровождаемость и др.

Примеры документов приведены в подразд. 3.4.2.

Функциональные требования к системе моделируются и документируются с помощью вариантов использования (use case), которые в контексте процесса управления требованиями трактуются следующим образом:

- вариант использования фиксирует соглашение между участниками проекта относительно поведения системы;
- вариант использования описывает поведение системы при различных условиях, когда система отвечает на запрос одного из участников, называемого основным действующим лицом;
- основное действующее лицо инициирует взаимодействие с системой, чтобы добиться некоторой цели. Система отвечает, соблюдая интересы всех участников.

Варианты использования — это вид документации, применяемый, когда требуется сконцентрировать усилия на обсуждении принципиальных требований к разрабатываемой системе, а не на подробном их описании. Стиль их написания зависит от масштаба, количества участников и критичности проекта. Существуют четыре уровня точности¹ при описании вариантов использования (расположенные по степени повышения точности):

- Действующие лица и цели (перечисляются действующие лица и все их цели, которые будет обеспечивать система).
- Краткое изложение варианта использования (в один абзац) или основной поток событий (без анализа возможных ошибок).
- Условия отказа (анализ мест возникновения возможных ошибок в основном потоке событий).
- Обработка отказа (написание альтернативных потоков событий).

Введение перечисленных уровней преследует своей целью грамотное планирование и экономию времени разработки. В итерационном цикле создания системы не следует пытаться за один прием подробно описать все требования, их нужно постепенно уточнять, повышая уровень точности. Выбор первооче-

¹ Коберн А. Современные методы описания функциональных требований к системам: Пер. с англ. — М.: ЛОРИ, 2002.

редных вариантов использования для уточнения определяется их приоритетами (см. подразд. 1.5).

Методика моделирования вариантов использования в технологии Rational Unified Process предусматривает специальное соглашение, связанное с группировкой структурных элементов и диаграмм модели. Это соглашение включает следующие правила:

- Все действующие лица, варианты использования и диаграммы вариантов использования помещаются в пакет с именем Use Case Model.
- Если моделируется сложная многофункциональная система, то совокупность всех действующих лиц и вариантов использования может разделяться на пакеты. В качестве принципов разделения могут использоваться:

структуризации модели в соответствии с типами пользователей (действующих лиц);

функциональная декомпозиция;

разделение модели на пакеты между группами разработчиков (в качестве объектов управления конфигурацией).

Все рекомендации по написанию качественных вариантов использования, включая перечисленные в подразд. 2.5.1, можно изложить в виде набора образцов, которые перечислены в табл. 3.3.

Таблица 3.3

Образцы написания вариантов использования

Наименование образца	Проблема	Предлагаемое решение
Формирование команды разработчиков		
Небольшая команда разработчиков	Участие слишком многих людей в написании варианта использования неэффективно; компромисс	Ограничьте число разработчиков, отрабатывающих детали варианта использования, двумя или тремя

	между многими различными точками зрения может привести к неудовлетворительным результатам при создании системы	людьми. Для подключения дополнительных участников используйте образец Двухуровневое рецензирование
Состав сторонних участников	Невозможно удовлетворить потребности всех заинтересованных лиц, не обмениваясь информацией с ними	Активно привлекайте заказчиков и заинтересованных лиц в вашей организации к разработке вариантов использования
Сбалансированная команда	Команда из близких по роду деятельности, одинаково мыслящих людей может создать небольшой набор ограниченных вариантов использования, не удовлетворяющий общим потребностям	Формируйте команду из людей с различными специализациями, представляющими интересы различных заинтересованных лиц. Команда должна включать как разработчиков, так и пользователей
Организация процесса разработки		
Глубина после общего представления	Невозможно продвигаться вперед и создавать набор согласованных вариантов использования, если тратить впустую время на последовательное написание подробных вариантов использования	Разрабатывайте сначала общее представление вариантов использования, затем постепенно добавляйте детали, работая с группой взаимосвязанных вариантов использования
Итерационная разработка	Разработка вариантов использования за один проход	Разрабатывайте варианты использования итерационно, по-

	затруднительна, усложняет внесение дополнительной информации и затрудняет выявление факторов риска	вышая на каждой итерации их точность и корректность
Множество форм	Различные проекты требуют различную степень формальности и различные шаблоны. Использовать один и тот же шаблон вариантов использования непродуктивно	Выбирайте формат вариантов использования, основываясь на проектных рисках и предпочтениях участников
Двухуровневое рецензирование	Многие заинтересованные лица могут пожелать принять участие в рецензировании вариантов использования. Это слишком долгое и дорогостоящее занятие	Используйте два вида рецензирования: первое, проводимое небольшой группой внутренних участников, может выполняться многократно; второе, проводимое полной группой, выполняется, как правило, один раз
Своевременное завершение	Разработка модели вариантов использования сверх потребностей заинтересованных лиц и разработчиков приводит к напрасным тратам ресурсов и затягивает проект	Прекращайте разработку вариантов использования, как только они достигают необходимой полноты и удовлетворяют потребности участников проекта
Свобода творчества	Чрезмерное внимание к стилю написания вариантов использования тормозит работу	Небольшие различия в стиле написания вариантов использования несущественны. Если вариант использования достиг необходимой полноты, его ав-

		тор имеет право на стилистические отступления
Организация набора вариантов использования		
Общепринятая четкая концепция	Недостаточно четкое представление о системе может привести к неуверенности и противоположным мнениям среди заинтересованных лиц и быстро парализовать проект	Необходимо подготовить и утвердить концепцию системы, в которой четко определены ее цели и роль в деятельности организации. Распространите концепцию среди всех участников проекта
Видимые границы	Если вы не определили границы системы, ее масштаб будет расти неконтролируемым образом	Установите четкую и видимую границу между системой и внешней средой, перечислив людей и оборудование, взаимодействующих с данной системой
Ясный состав действующих лиц	Если выявлять только пользователей системы, игнорируя роли, которые они играют по отношению к системе, можно упустить существенную часть поведения системы или ввести избыточное поведение	Идентифицируйте действующих лиц, с которыми должна взаимодействовать система, и роли, которые они играют по отношению к системе. Четко опишите каждую роль
Транзакции, значимые для пользователей	Система несовершенна, если она не может предоставить пользователям необходимые услуги и не выполняет цели и задачи, определяемые ее	Идентифицируйте важные, значимые услуги, которые система предоставляет действующим лицам для удовлетворения их потребностей

	концепцией	
Разворачивающееся представление	Количество шагов в описании поведения системы превышает возможности охвата и интересы различных типов читателей вариантов использования	Создайте иерархическую организацию набора вариантов использования, которую можно развернуть для большей детализации, или свернуть, чтобы скрыть детали и показать контекст
Отдельный вариант использования		
Законченная единственная цель	Неправильно заданные цели не дают разработчикам четко определить, где заканчивается один вариант использования и начинается другой	Каждый вариант использования должен иметь законченную и четко определенную цель, которая может находиться на любом уровне образца Разворачивающееся представление
Имя в виде глагольной фразы	Бессмысленные, слишком общие имена вариантов использования не дают читателям правильного представления, на них сложно ссылаться	Именуйте каждый вариант использования активной глагольной фразой, отражающей цель основного действующего лица
Сценарий и фрагменты	Читатели должны иметь возможность легкого следования по интересующему их сценарию, в противном случае они могут утратить интерес или потерять важ-	Основной сценарий должен быть предельно простым, без анализа возможных ошибочных ситуаций. Фрагменты сценария, отражающие альтернативы, должны рас-

	ную информацию	полагаться под ним
Исчерпывающие альтернативы	Вариант использования может иметь много альтернатив. Отсутствие некоторых из них означает, что разработчики неправильно понимают поведение системы, и система может получиться несовершенной	Описывайте все альтернативы и ошибочные ситуации, которые могут иметь место в варианте использования
Перегрузка информацией	Включение нефункциональных требований в вариант использования может быстро привести его в неопределенное и беспорядочное состояние	Включите дополнительные позиции в шаблон варианта использования за пределами текста сценария, чтобы отразить полезную дополнительную информацию
Точность и читаемость	Варианты использования, слишком сложные для нетехнических специалистов или слишком неточные для разработчиков, несовершенны и могут привести к созданию неадекватной системы	Сделайте вариант использования достаточно читаемым, чтобы заинтересованные лица могли в нем разобраться и оценить его, и достаточно точным, чтобы разработчики понимали, что им делать
Сценарии и шаги		
Обнаруживаемые условия	Разработчики вариантов использования всегда решают проблему, сколько и какие условия включить в их опи-	Включайте только реально обнаруживаемые условия. Объединяйте условия, которые оказывают на систему одина-

	сание	ковое воздействие
Уровни шагов	Чрезмерно крупные или мелкие шаги сценария варианта использования делают вариант использования трудным для восприятия и понимания	Оставляйте в сценарии от трех до девяти шагов. В идеальном случае все шаги должны быть на близких уровнях и на уровне абстракции, следующем за целью варианта использования
Выполнение цели действующего лица	У читателей и разработчиков возникают проблемы в понимании поведения системы, если неясно, какое действующее лицо отвечает за выполнение шага сценария, и что оно делает для его завершения	При описании каждого четко указывайте, какое действующее лицо выполняет действие, и что оно получает по его завершении
Продвижение вперед	Разработчики должны решить, как много поведения включить в каждый шаг. Слишком много деталей делает вариант использования длинным и трудным для чтения	Исключайте или объединяйте шаги, которые не означают никакого движения вперед для действующего лица. Упрощайте фрагменты, которые отвлекают внимание читателя от Движения вперед
Нейтральность к технологии	Включение технологических ограничений и деталей реализации в описание варианта использования увеличивает сложность и за-	Описание варианта использования должно быть нейтральным по отношению к технологии

	трудняет понимание его цели	
Связи между вариантами использования		
Общее поведение	Описание одинаковых шагов в различных вариантах использования занимает лишнее время и затрудняет понимание общих процессов в модели вариантов использования	Выражайте общие действия в виде «включаемых» вариантов использования
Перемещение альтернатив	Длинные или сложные описания альтернатив могут занять доминирующее положение и показаться более важными, чем они заслуживают	Рассмотрите возможность выделения сложных альтернатив в отдельный вариант использования
Абстракция	Попытка описать в одном варианте использования две или более различных альтернатив, ни одна из которых не является доминирующей, приводит к проблемам	Создайте обобщенный абстрактный вариант использования. Поместите каждый отдельный вариант сценария, уточняющий абстракцию, в специализированный вариант использования
Модификация существующих вариантов использования		
Удаление излишеств	Чрезмерно длинный вариант использования громоздок и труден для работы, уводит в сторону внимание пользова-	Переместите длинный, громоздкий фрагмент или слишком сложное расширение в отдельный вариант использова-

	телей	ния
Объединение фрагментов	Варианты использования, описывающие очень маленькие или изолированные фрагменты поведения, не выражают достаточной для понимания информации относительно услуг, предоставляемых системой (Транзакции, значимые для пользователей)	Объединяйте небольшие взаимосвязанные варианты использования или фрагменты вариантов использования в варианты использования, связанные общей целью
Удаление лишних вариантов использования	Варианты использования, результаты которых незначительны, отвлекают внимание и затрудняют понимание системы	Удалите варианты использования, которые не дают ничего существенного для системы или выпали из текущего активного списка вариантов использования

Варианты использования, безусловно, не единственный способ выражения функциональных требований. Это объясняется следующими причинами:

- их описания слишком подробны. Заинтересованные лица зачастую желают видеть лишь краткий перечень основных функций;
- некоторые важные функции системы легче выразить в краткой форме без привязки к именам вариантов использования.

Следовательно, помимо вариантов использования, функции системы можно выразить через ее свойства (system features), представляющие собой высокоуровневые, краткие утверждения. Более строго в контексте Rational Unified Process системное свойство определяется как «наблюдаемая извне и обеспечи-

ваемая системой функция, которая непосредственно удовлетворяет потребности заинтересованного лица». Свойство можно облечь в следующую лингвистическую форму: *система будет выполнять «свойство X»*.

Спецификация требований в технологии Rational Unified Process не требует обязательного моделирования бизнес-процессов организации, для которых создается ПО, однако наличие бизнес-моделей существенно упрощает построение системной модели вариантов использования. При переходе от бизнес-модели к начальной версии модели вариантов использования применяются следующие правила.

- Для каждого исполнителя в модели бизнес-анализа, который в перспективе станет пользователем новой системы, в модели вариантов использования создается действующее лицо с таким же наименованием. В состав действующих лиц включаются также внешние системы, играющие в бизнес-процессах пассивную роль источников информации.
- Варианты использования для данного действующего лица создаются на основе анализа обязанностей соответствующего исполнителя (в простейшем случае для каждой операции исполнителя создается вариант использования, реализующий данную операцию в системе).

Такая начальная версия модели описывает минимальный вариант системы, пользователями которой являются только исполнители бизнес-процессов. Если в дальнейшем в процессе развития системы ее непосредственными пользователями будут становиться действующие лица бизнес-процессов, то модель вариантов использования будет соответствующим образом модифицироваться.

Применение данных правил будет проиллюстрировано в следующем примере.

ПРИМЕР СПЕЦИФИКАЦИИ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

В данном подразделе рассматривается спецификация требований к системе регистрации для учебного заведения, бизнес-модель которого описана в подразд. 3.3.2.

Уточненная постановка задачи для системы

Перед руководителем информационной службы университета ставится задача разработки автоматизированной системы регистрации студентов на дополнительные платные курсы. Система должна позволять студентам регистрироваться на курсы и просматривать свои таблицы успеваемости с персональных компьютеров, подключенных к локальной сети университета. Профессора должны иметь доступ к системе, чтобы указать курсы, которые они будут читать, и проставить оценки за курсы.

Из-за недостатка средств университет не в состоянии заменить всю существующую систему. Остается функционировать в прежнем виде база данных, содержащая всю информацию о курсах (каталог курсов). Эта база данных поддерживается реляционной СУБД. Новая система будет работать с существующей БД в режиме доступа, без обновления.

В начале каждого семестра студенты могут запросить каталог курсов, содержащий список курсов, предлагаемых в данном семестре. Информация о каждом курсе должна включать имя профессора, наименование кафедры и требования к предварительному уровню подготовки (прослушанным курсам).

Новая система должна позволять студентам выбирать 4 курса и предстоящем семестре. В дополнение к этому каждый студент может указать 2 альтернативных курса на тот случай, если какой-либо из выбранных им курсов окажется уже заполненным или отмененным. На каждый курс может записаться не более 10 и не менее 3 студентов (если менее 3, то курс будет отменен). В каж-

дом семестре существует период времени, когда студенты могут изменить свои планы (добавить или отказаться от выбранных курсов). После того, как процесс регистрации некоторого студента завершен, система регистрации направляет информацию в расчетную систему, чтобы студент мог внести плату за семестр. Если курс окажется заполненным в процессе регистрации, студент должен быть извещен об этом до окончательного формирования его личного учебного плана.

В конце семестра студенты должны иметь доступ к системе для просмотра своих электронных таблиц успеваемости. Поскольку эта информация конфиденциальная, система должна обеспечивать ее защиту от несанкционированного доступа.

Глоссарий проекта

Описание дополнительных спецификаций

Функциональные возможности:

Термин	Значение
Курс	Учебный курс, предлагаемый университетом
Конкретный курс	Конкретное чтение данного курса в конкретном семестре (один и тот же курс может вестись в нескольких параллельных сессиях). Включает точные дни недели и время.
Каталог курсов	Полный каталог всех курсов, предлагаемых университетом
Расчетная система	Система обработки информации об оплате за курсы.
Оценка	Оценка, полученная студентом за конкретный курс
Профессор	Преподаватель университета.
Табель успеваемости	Все оценки за все курсы, полученные студентом в данном семестре.
Список курса	Список всех студентов, записавшихся на конкретный курс.
Студент	Личность, проходящая обучение в университете.
Учебный график	Курсы, выбранные студентом в текущем семестре.

- Система должна обеспечивать многопользовательский режим работы.
- Если конкретный курс оказывается заполненным в то время, когда студент формирует свой учебный график, включающий данный курс, то система должна известить его об этом.

Удобство использования:

- Пользовательский интерфейс должен быть Windows-совместимым.

Надежность.

- Система должна быть в работоспособном состоянии 24 часа в день 7 дней в неделю, время простоя — не более 10%.

Производительность.

- Система должна поддерживать до 2000 одновременно работающих с центральной базой данных пользователей и до 500 пользователей, одновременно работающих с локальными серверами.

Безопасность.

- Система не должна позволять студентам изменять любые учебные графики, кроме своих собственных, а также не должна позволять профессорам модифицировать конкретные курсы, выбранные другими профессорами.
- Только профессора имеют право ставить студентам оценки.
- Только регистратор может изменять любую информацию о студентах.

Проектные ограничения.

- Система должна быть интегрирована с существующей системой каталога курсов, функционирующей на основе реляционной СУБД.

Создание начальной версии модели вариантов использования

Действующие лица:

- Регистратор — формирует учебный план и каталог курсов, записывает студентов на курсы, ведет все данные о курсах, профессорах, успеваемости и студентах.
- Расчетная система — получает от данной системы информацию по опла-

те за курсы.

- Каталог курсов — база данных, содержащая информацию о курсах.

Варианты использования.

Исходя из потребностей действующих лиц, выделяются следующие варианты использования:

- Войти в систему;
- Зарегистрировать студента на курсы;
- Вывести таблицу успеваемости;
- Назначить курсы для преподавания;
- Проставить оценки;
- Вести информацию о профессорах;
- Вести информацию о студентах;
- Закрыть регистрацию.

Начальная версия диаграммы вариантов использования показана на рис. 3.19.



Рис. 3.19. Начальная версия диаграммы вариантов использования

Модификация модели вариантов использования

Согласно постановке задачи в состав пользователей системы следует ввести студентов и профессоров. При этом в описание действующих лиц и вариантов использования вносятся изменения. Модифицированная версия диаграммы вариантов использования показана на рис. 3.20.

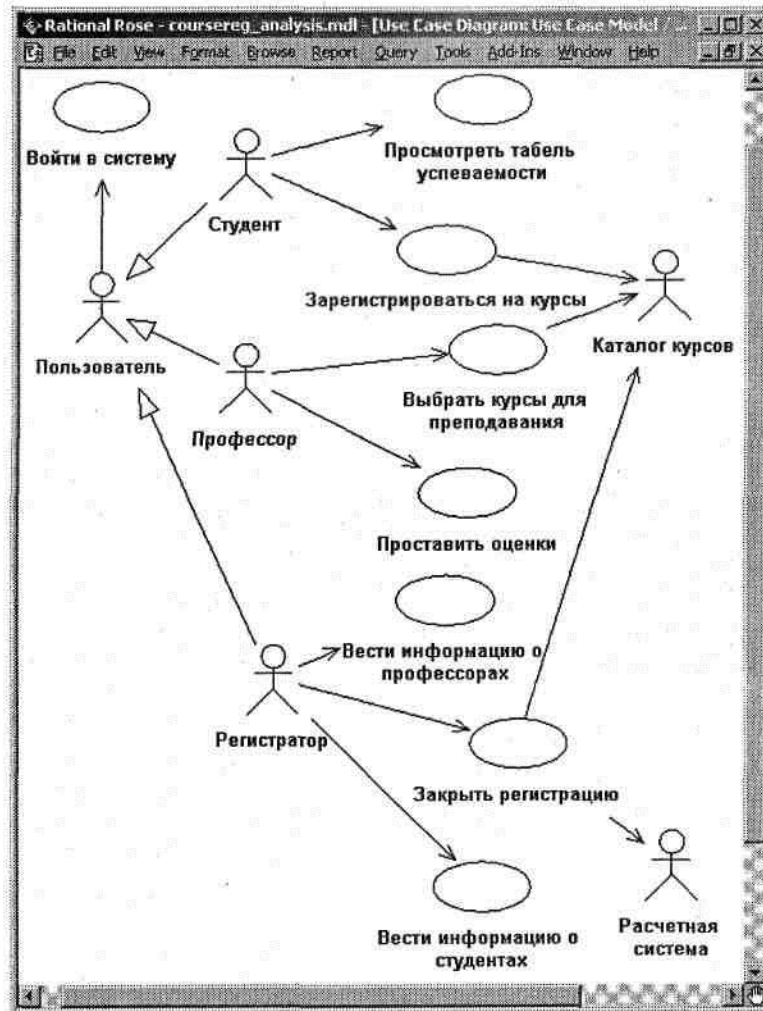


Рис. 3.20. Модифицированная диаграмма вариантов использования для системы регистрации

Поскольку вход в систему полностью одинаков для регистратора, студента и профессора, их поведение можно обобщить и ввести новое действующее «Пользователь» (супертип) с общим вариантом использования «Войти в систему», подтипами которого являются Регистратор, Студент и Профессор.

Действующие лица:

- Студент — записывается на курсы и просматривает таблицу успеваемости.
- Профессор - выбирает курсы для преподавания и ставит оценки.
- Регистратор — формирует учебный план и каталог курсов, ведет все данные о курсах, профессорах и студентах.
- Расчетная система — получает отданной системы информацию об оплате

за курсы.

- Каталог курсов — база данных, содержащая информацию о курсах.

Варианты использования:

- Войти в систему;
- Зарегистрироваться на курсы;
- Просмотреть таблицу успеваемости; ,
- Выбрать курсы для преподавания;
- Проставить оценки;
- Вести информацию о профессорах;
- Вести информацию о студентах;
- Закрыть регистрацию.

Примеры спецификаций вариантов использования

Вариант использования «Войти в систему».

Краткое описание.

Данный вариант использования описывает вход пользователя в систему регистрации курсов.

Основной поток событий.

Данный вариант использования начинается выполняться, когда пользователь хочет войти в систему регистрации курсов.

1. Система запрашивает имя пользователя и пароль.
2. Пользователь вводит имя и пароль.
3. Система подтверждает имя и пароль, после чего открывается доступ в систему.

Альтернативные потоки.

Неправильное имя/пароль.

Если во время выполнения основного потока обнаружится, что пользователь ввел неправильное имя и/или пароль, система выводит сообщение об ошибке. Пользователь может вернуться к началу основного потока или отказаться от входа в систему, при этом выполнение варианта использования завершается.

Предусловия.

Отсутствуют.

Постусловия.

Если вариант использования выполнен успешно, пользователь входит в систему. В противном случае состояние системы не изменяется.

Вариант использования «Зарегистрироваться на курсы»:

Краткое описание.

Данный вариант использования позволяет студенту зарегистрироваться на предлагаемые курсы в текущем семестре. Студент может изменить свой выбор (обновить или удалить курсы), если изменение выполняется в установленное время в начале семестра. Система каталога курсов предоставляет список всех конкретных курсов текущего семестра.

Основной поток событий.

Данный вариант использования начинается выполняться, когда студент хочет зарегистрироваться на предлагаемые курсы или изменить свой график курсов.

1. Система запрашивает требуемое действие (создать график, обновить график, удалить график).
2. Когда студент указывает действие, выполняется один из подчиненных потоков (создать, обновить, удалить или принять график).

Создать график.

1. Система выполняет поиск в каталоге доступных предлагаемых курсов и выводит их список.
2. Студент выбирает из списка 4 основных курса и 2 альтернативных курса.
3. После выбора система создает график студента.
4. Выполняется подчиненный поток «Принять график».

Обновить график.

1. Система выводит текущий график студента.
2. Система выполняет поиск в каталоге доступных предлагаемых курсов и выводит их список.
3. Студент может обновить свой выбор курсов, удаляя или добавляя предлага-

гаемые курсы.

4. После выбора система обновляет график.
5. Выполняется подчиненный поток «Принять график».

Удалить график.

1. Система выводит текущий график студента.
2. Система запрашивает у студента подтверждения удаления графика.
3. Студент подтверждает удаление.

4. Система удаляет график. Если график включает предлагаемые курсы, на которые записался студент, он должен быть удален из списков этих курсов.

Принять график.

Для каждого выбранного, но еще не «зафиксированного» предлагаемого курса в графике система проверяет выполнение студентом предварительных требований (прохождение определенных курсов), факт открытия предлагаемого курса и отсутствие конфликтов графика. Затем система добавляет студента в список выбранного предлагаемого курса. Курс фиксируется в графике и график сохраняется в системе.

Альтернативные потоки.

Сохранить график.

В любой момент студент может вместо принятия графика сохранить его. В этом случае шаг «Принять график» заменяется на следующий:

1. «Незафиксированные» предлагаемые курсы помечаются в графике как «выбранные».
2. График сохраняется в системе.

Не выполнены предварительные требования, курс заполнен или имеют место конфликты графика.

Если во время выполнения подчиненного потока «Принять график» система обнаружит, что студент не выполнил необходимые предварительные требования или выбранный им предлагаемый курс заполнен, или имеют место конфликты графика, то выдается сообщение об ошибке. Студент может либо выбрать другой предлагаемый курс и продолжить выполнение варианта ис-

пользования, либо сохранить график, либо отменить операцию, после чего основной поток начнется с начала.

График не найден.

Если во время выполнения подчиненных потоков «Обновить график» или «Удалить график» система не может найти график студента, то выдается сообщение об ошибке. После того, как студент подтвердит это сообщение, основной поток начнется с начала.

Система каталога курсов недоступна.

Если окажется, что невозможно установить связь с системой каталога курсов, то будет выдано сообщение об ошибке. После того, как студент подтвердит это сообщение, вариант использования завершится.

Регистрация на курсы закончена.

Если в самом начале выполнения варианта использования окажется, что регистрация на текущий семестр закончена, будет выдано сообщение, и вариант использования завершится.

Удаление отменено.

Если во время выполнения подчиненного потока «Удалить график» студент решит не удалять его, удаление отменяется, и основной поток начнется с начала.

Предусловия.

Перед началом выполнения данного варианта использования студент должен войти в систему.

Постусловия.

Если вариант использования завершится успешно, график студента будет создан, обновлен или удален. В противном случае состояние системы не изменится.

Вариант использования «Закрывать регистрацию».

Краткое описание.

Данный вариант использования позволяет регистратору закрывать процесс регистрации. Предлагаемые курсы, на которые не записалось достаточного ко-

личества студентов (менее трех), отменяются. В расчетную систему передается информация о каждом студенте по каждому предлагаемому курсу, чтобы студенты могли внести оплату за курсы.

Основной поток событий.

Данный вариант использования начинает выполняться, когда регистратор запрашивает прекращение регистрации.

1. Система проверяет состояние процесса регистрации. Если регистрация еще выполняется, выдается сообщение и вариант использования завершается.

2. Для каждого предлагаемого курса система проверяет, ведет ли его какой-либо профессор и записалось ли на него не менее трех студентов. Если эти условия выполняются, система фиксирует предлагаемый курс в каждом графике, который включает данный курс.

3. Для каждого студенческого графика проверяется наличие в нем максимального количества основных курсов; если их недостаточно, система пытается дополнить альтернативными курсами из списка данного графика. Выбирается первый доступный альтернативный курс. Если таких курсов нет, то никакое дополнение не происходит.

4. Система закрывает все предлагаемые курсы. Если в каком-либо предлагаемом курсе оказывается менее трех студентов (с учетом добавлений, сделанных в п.3), система отменяет его и исключает из каждого содержащего его графика.

5. Система рассчитывает плату за обучение для каждого студента в текущем семестре и направляет информацию в расчетную систему. Расчетная система посылает студентам счета для оплаты с копией их окончательных графиков.

Альтернативные потоки.

Предлагаемый курс никто не ведет.

Если во время выполнения основного потока обнаруживается, что некоторый курс не ведется никаким профессором, то этот курс отменяется. Система исключает данный курс из каждого содержащего его графика.

Расчетная система недоступна.

Если невозможно установить связь с расчетной системой, через некоторое

установленное время система вновь попытается связаться с ней. Попытки будут повторяться до тех пор, пока связь не установится.

Предусловия. ,

Перед началом выполнения данного варианта использования регистратор должен войти в систему.

Постусловия.

Если вариант использования завершится успешно, регистрация закрывается. В противном случае состояние системы не изменится.

! Следует запомнить.

1. Моделирование бизнес-процессов является важной составной частью проектов по созданию крупномасштабных систем ПО. Отсутствие таких моделей является одной из главных причин неудач многих проектов.

2. Бизнес-процесс определяется как логически заверченный набор взаимосвязанных и взаимодействующих видов деятельности, поддерживающий деятельность организации и реализующий ее политику, направленную на достижение поставленных целей.

3. Бизнес-модель — это формализованное описание процессов, связанных с ресурсами и отражающих существующую или предполагаемую деятельность предприятия. Построение бизнес-моделей заключается в применении различных методов и средств для визуального моделирования бизнес-процессов.

4. Для выявления требований используются (в различных сочетаниях) следующие методы:

- собеседование (интервьюирование);
- анкетирование;
- моделирование и анализ бизнес-процессов;
- сессии по выявлению требований (мозговой штурм);
- создание и демонстрация пользователям работающих прототипов приложений (для выявления замечаний и дополнительных требований).

Основные понятия

Бизнес-процесс, бизнес-модель, бизнес-правила, действующее лицо бизнес-процессов, вариант использования с точки зрения бизнес-процессов, бизнес-объект.

? Вопросы для самоконтроля

1. Что такое бизнес-процесс и бизнес-модель?
2. Что дает построение бизнес-моделей и какие проблемы с ним связаны?
3. Охарактеризуйте достоинства и область применения методики моделирования Rational Unified Process.
4. Какие проблемы могут возникнуть при спецификации требований в случае отсутствия бизнес-моделей?

ГЛАВА 4.

АНАЛИЗ И ПРОЕКТИРОВАНИЕ

ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой анализ и проектирование ПО.*
- *В чем заключается структурный подход к анализу и проектированию ПО.*
- *В чем заключается объектно-ориентированный подход к анализу и проектированию ПО.*

СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ ПО

В данном подразделе рассматривается вторая часть методики Йордона (см. подразд. 3.2.2), связанная с переходом от бизнес-модели к модели системы. Пример применения данной методики приведен в подразд. 4.2.

Согласно данной методике в процессе проектирования выполняется детальное описание функционирования системы, дальнейший анализ используемых данных и построение реляционной модели для последующего проектирования базы данных. Определяется также структура пользовательского интерфейса. Результатами проектирования являются:

- модель системных процессов;
- архитектура системы;
- модели данных приложений;
- модель пользовательского интерфейса.

Функциональные модели, используемые в процессе проектирования, предназначены для описания функциональной структуры проектируемой системы. Диаграммы потоков данных (DFD) используются для описания структуры проектируемой системы, при этом они могут уточняться, расширяться и дополняться новыми конструкциями. Аналогично, ERM преобразуется в схему базы данных. Данные модели могут дополняться диаграммами, отражающими системную архитектуру ПО, структурные схемы программ (структурные карты), иерархию экранных форм и меню и др. Состав диаграмм и степень их детализации определяются необходимой полнотой описания системы для непосредственного перехода к ее последующей реализации (программированию).

Например, для DFD переход от модели бизнес-процессов к *модели системных процессов* может происходить следующим образом:

- внешние сущности на контекстной диаграмме заменяются или дополняются

- ются техническими устройствами (например, рабочими станциями, принтерами и т.д.);
- для каждого потока данных определяется, посредством каких технических устройств информация передается или производится;
 - процессы на диаграмме нулевого уровня заменяются соответствующими процессорами — обрабатывающими устройствами (процессорами могут быть как технические устройства — настольные компьютеры конечных пользователей, рабочие станции, серверы баз данных, так и программные средства);
 - определяется и изображается на диаграмме тип связи между процессорами (например, локальная сеть);
 - определяются задачи для каждого процессора (приложения, необходимые для работы системы), для них строятся соответствующие диаграммы. Определяется тип связи между задачами;
 - устанавливаются ссылки между задачами и процессами диаграмм потоков данных следующих уровней.

Иерархия экранных форм Моделируется с помощью **диаграмм последовательностей экранных форм**. Совокупность таких диаграмм представляет собой абстрактную модель пользовательского интерфейса системы, отражающую последовательность появления экранных форм в приложении. Построение диаграмм последовательностей экранных форм выполняется следующим образом:

- на DFD выбираются интерактивные процессы нижнего уровня. Интерактивные процессы нуждаются в пользовательском интерфейсе, поэтому можно определить экранную форму для каждого такого процесса;
- построение диаграммы последовательностей форм начинается с изображения формы в виде прямоугольника для каждого интерактивного процесса на нижнем уровне диаграммы;
- определяется структура меню. Для этого интерактивные процессы группируются в меню (либо так же, как в модели процессов, либо другим способом — по функциональным признакам или в зависимости от при-

- надлежности к определенным объектам);
- формы с меню изображаются над формами, соответствующими интерактивным процессам, и соединяются с ними переходами в виде стрелок, направленных от меню к формам.
 - определяется верхняя форма (главная форма приложения), связывающая все формы с меню.

Техника *структурных карт* (схем) используется на стадии проектирования для описания структурных схем программ. При этом наиболее часто применяются две техники: структурные карты Константайна, предназначенные для описания отношений между модулями, и структурные карты Джексона, предназначенные для описания внутренней структуры модулей, являющихся базовыми строительными блоками программной системы. В настоящее время структурные карты применяются сравнительно редко.

В процессе проектирования базы данных концептуальная модель данных преобразуется в схему реляционной базы данных, основными элементами которой являются таблицы и столбцы. Для описания схемы базы данных может использоваться отдельная графическая нотация.

Для формирования схемы базы данных из ERM применяется ограниченный набор правил преобразования сущностей и связей между ними.

Правила преобразования сущностей:

Правило 1. Каждая простая сущность, не являющаяся подтипом и не имеющая подтипов, превращается в таблицу. Имя сущности становится именем таблицы.

Правило 2. Каждый атрибут становится возможным столбцом с тем же именем; может выбираться более точный формат. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, — не могут.

Правило 3. Уникальный идентификатор сущности превращается в первичный ключ таблицы. Если имеется несколько альтернативных уникальных идентификаторов, выбирается наиболее используемый. Если уникальный идентифи-

катор является относительным, то в качестве первичного ключа используется копия уникального идентификатора сущности, находящейся на дальнем конце связи, в которой данная сущность играет роль зависимой.

Правила преобразования связей:

Правило 1. Если тип бинарной связи — *один-к-одному* и класс принадлежности обеих сущностей является обязательным, то из двух связанных сущностей формируется одна таблица. Первичным ключом этой таблицы может быть идентификатор любой из двух сущностей.

Правило 2. Если тип бинарной связи — *один-к-одному* и класс принадлежности одной сущности является обязательным, а другой — необязательным, то формируются две таблицы (под каждую сущность), при этом идентификатор каждой сущности должен служить первичным ключом соответствующей таблицы. Кроме того, идентификатор сущности, для которой класс принадлежности является необязательным, добавляется в качестве атрибута в таблицу, выделенную для сущности с обязательным классом принадлежности.

Правило 3. Если тип бинарной связи - *один-к-одному* и класс принадлежности ни одной сущности не является обязательным, то формируются три таблицы: по одной для каждой сущности (при этом идентификатор каждой сущности должен служить первичным ключом соответствующей таблицы) и одна для связи. Таблица связи включает в качестве атрибутов по одному идентификатору от каждой сущности.

Правило 4. Если тип бинарной связи — *один-ко-многим* и класс принадлежности сущности с мощностью «n» является обязательным, то формируются две таблицы (под каждую сущность), при этом идентификатор каждой сущности должен служить первичным ключом соответствующей таблицы. Кроме того, идентификатор сущности с мощностью «1» добавляется в качестве атрибута в таблицу, выделенную для сущности с мощностью «n».

Правило 5. Если тип бинарной связи — *один-ко-многим* и класс принадлежности сущности с мощностью «n» является необязательным, см. правило 3.

Правило 6. Если тип бинарной связи — *многие-ко-многим*, см. правило 3.

Правило 7. Для представления N-арной связи требуется N+1 таблица. Например, в случае тернарной связи формируются четыре таблицы, по одной для каждой сущности и одна для связи. Таблица связи будет иметь среди своих атрибутов ключи от каждой сущности.

Правило 8. Для связи «супертип-подтип» возможны два способа преобразования:

- а) все подтипы в одной таблице;
- б) для каждого подтипа — отдельная таблица.

При применении способа (а) для супертипа создается таблица, а для подтипов могут создаваться представления (view). В таблицу добавляется по крайней мере один столбец, содержащий код типа; он становится частью первичного ключа.

При использовании способа (б) для каждого подтипа супертип воссоздается с помощью операции объединения (UNION) — из всех таблиц подтипов выбираются общие столбцы - столбцы супертипа.

Преимущества способа (а): все данные хранятся вместе, обеспечивается легкий доступ к супертипу и подтипам, требуется меньше таблиц.

Недостатки способа (а): усложняется логика работы с разными наборами столбцов и ограничениями, возможно снижение производительности (в связи с блокировками общей таблицы), столбцы подтипов должны допускать неопределенные значения.

Преимущества способа (б): наглядное соответствие схемы БД и ERM, приложения работают только с нужными таблицами.

Недостатки способа (б): формируется слишком много таблиц, возможно снижение производительности при выполнении операции объединения, невозможны модификации супертипа.

ПРИМЕР СТРУКТУРНОГО ПРОЕКТИРОВАНИЯ ПО

Здесь продолжено рассмотрение примера из подразд. 3.2.5.

Построение диаграмм системных процессов и диаграмм последовательностей экранных форм

Результаты перехода от модели бизнес-процессов к модели системных процессов представлены на рис. 4.1—4.2. На диаграмме системных процессов первого уровня вместо отдельных процессов введены процессоры — компьютеры, на которых выполняются соответствующие процессы.

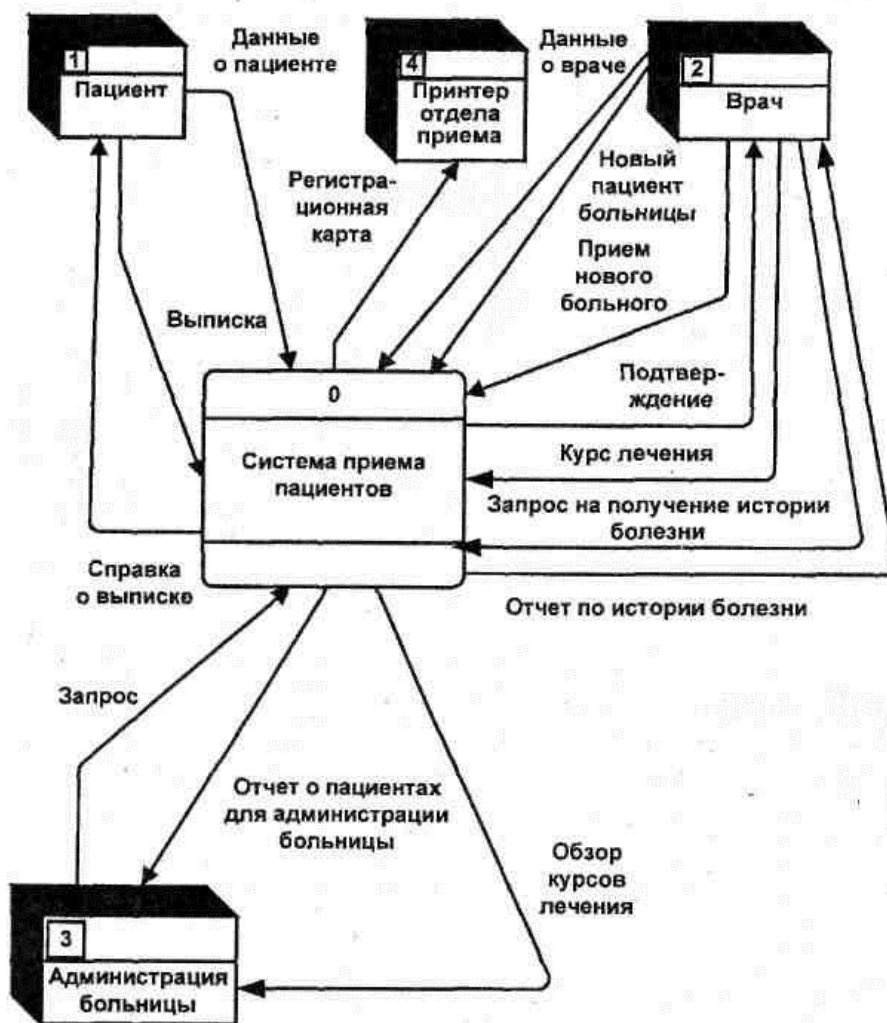


Рис. 4.1. Диаграмма системных процессов нулевого уровня

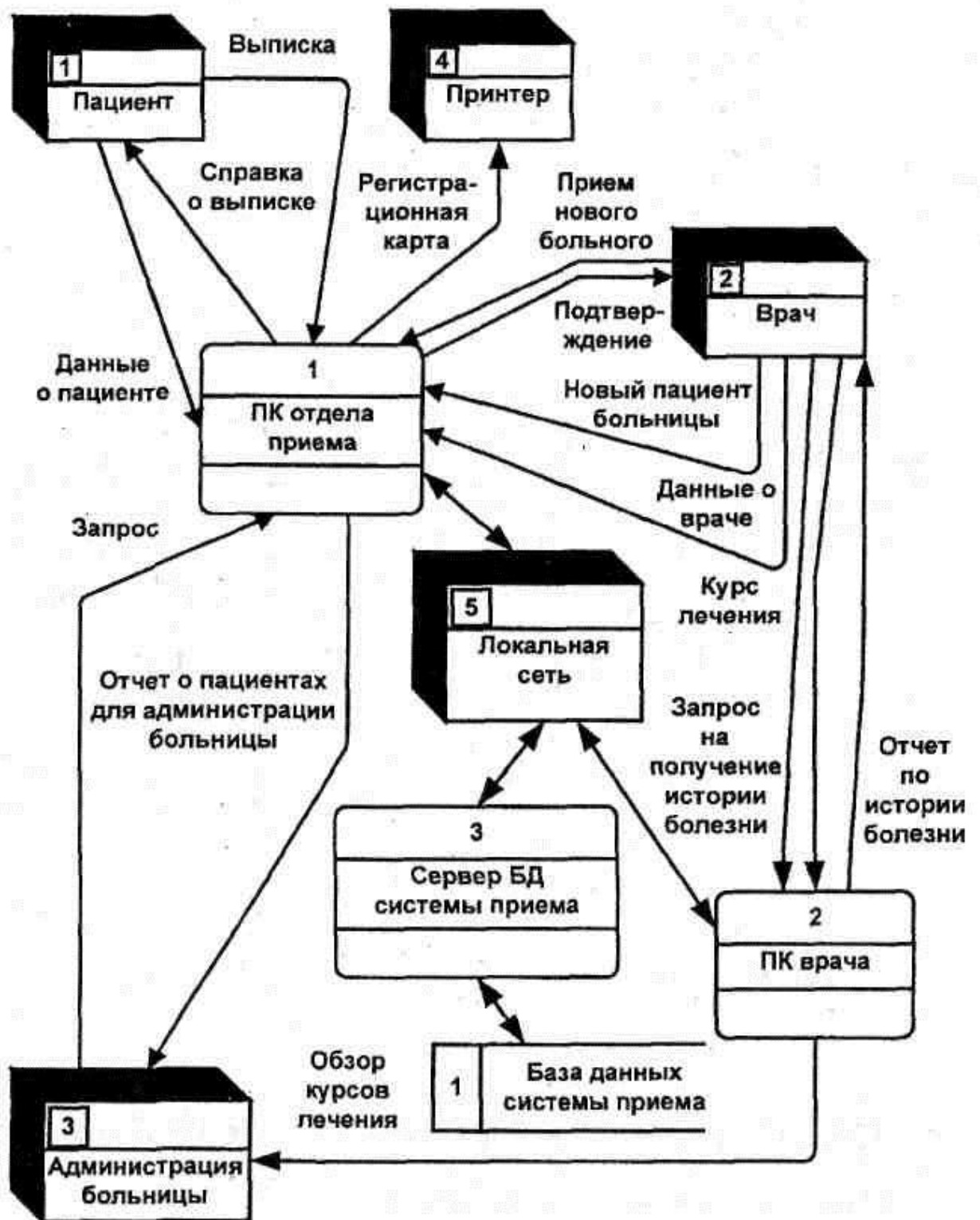


Рис. 4.2. Диаграмма системных процессов первого уровня

Результаты построения абстрактной модели пользовательского интерфейса системы, отражающей последовательность появления экранных форм в приложении, представлены на рис. 4.3.

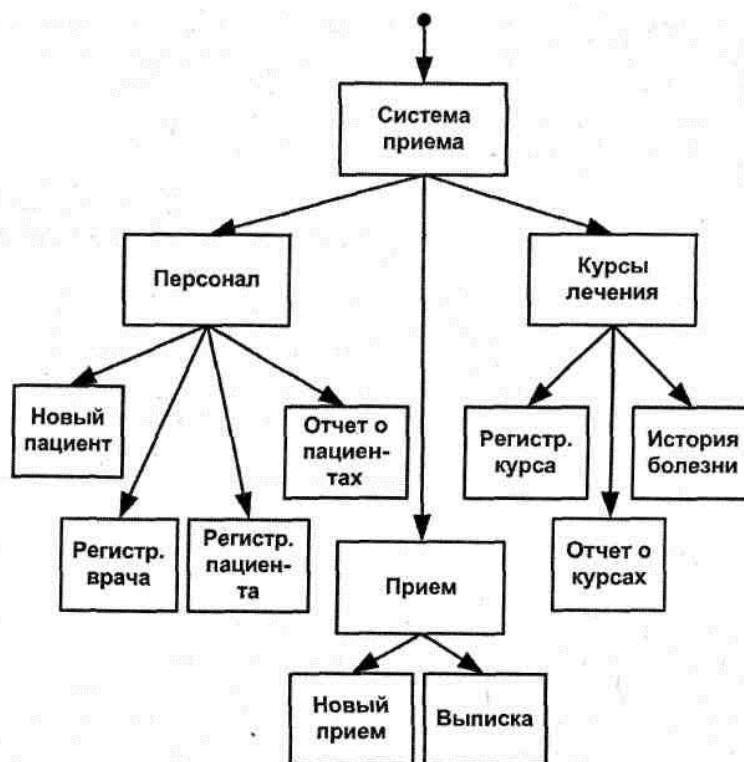


Рис. 4.3. Диаграмма последовательности экранных форм

4.3.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ

Целью объектно-ориентированного анализа¹ является трансформация функциональных требований к ПО в предварительный системный проект и создание стабильной основы архитектуры системы. В процессе проектирования системный проект «погружается» в среду реализации с учетом всех нефункциональных требований.

Объектно-ориентированный анализ включает два вида деятельности: архитектурный анализ и анализ вариантов использования. Изложение их методики будет сопровождаться примерами из системы регистрации учебного заведения, которая рассматривалась в главе 3.

¹ Методической основой данного и следующего подраздела является технология Rational Unified Process.

АРХИТЕКТУРНЫЙ АНАЛИЗ

Архитектурный анализ выполняется архитектором системы и включает в себя:

- утверждение общих стандартов (соглашений) моделирования и документирования системы;
- предварительное выявление архитектурных механизмов (механизмов анализа);
- формирование набора основных абстракций предметной области (классов анализа);
- формирование начального представления архитектурных уровней.

Соглашения моделирования определяют:

- используемые диаграммы и элементы модели;
- правила их применения;
- соглашения по именованию элементов модели;
- организацию модели (пакеты).

Пример набора соглашений моделирования:

- Имена вариантов использования должны быть короткими глагольными фразами.
- Имена классов должны быть существительными, соответствующими, по возможности, понятиям предметной области.
- Имена классов должны начинаться с заглавной буквы.
- Имена атрибутов и операций должны начинаться со строчной буквы.
- Составные имена должны быть сплошными, без подчеркиваний, каждое отдельное слово должно начинаться с заглавной буквы.
- Все классы и диаграммы, описывающие предварительный системный проект, помещаются в пакет с именем Analysis Model.
- Диаграммы классов, реализующих вариант использования, и диаграммы

взаимодействия, отражающие взаимодействие объектов в процессе реализации сценариев варианта использования, помещаются в кооперацию (см. подразд. 2.6) с именем данного варианта использования и стереотипом «**use-case realization**». Все кооперации помещаются в пакет с именем Use Case Realizations. Связь между вариантом использования и его реализацией изображается на специальной диаграмме трассировки (рис. 4.4).

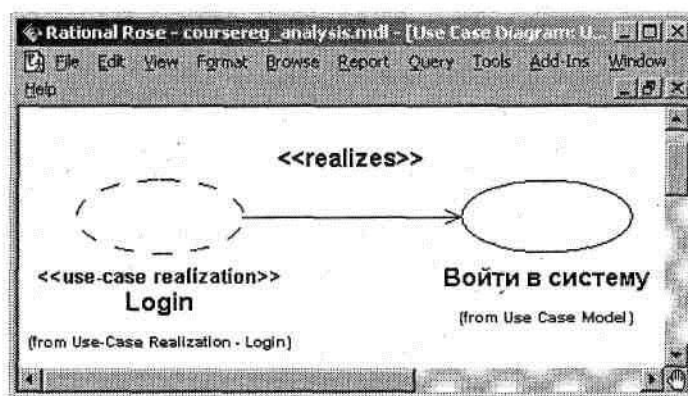


Рис. 4.4. Фрагмент диаграммы трассировки

Архитектурные механизмы отражают нефункциональные требования к системе (надежность, безопасность, хранение данных в конкретной среде, интерфейсы с внешними системами и т.д.) и их реализацию в архитектуре системы. В процессе анализа они только обозначаются, отвлекаясь от особенностей среды реализации, а все детали их реализации определяются в процессе проектирования.

Архитектурные механизмы, по существу, представляют собой набор типовых решений, или образцов, принятых в качестве стандарта данного проекта.

Идентификация основных абстракций заключается в предварительном определении набора классов системы (классов анализа) на основе описания предметной области и спецификации требований к системе (в частности, глоссария). Способы идентификации основных абстракций аналогичны способам идентификации сущностей в модели ERM.

Так, для системы регистрации идентифицировано пять классов анализа:

- Student (Студент);

- Professor (Профессор);
- Schedule (Учебный график);
- Course (Курс);
- CourseOffering (Предлагаемый курс).

Классы анализа показаны на рис. 4.5.

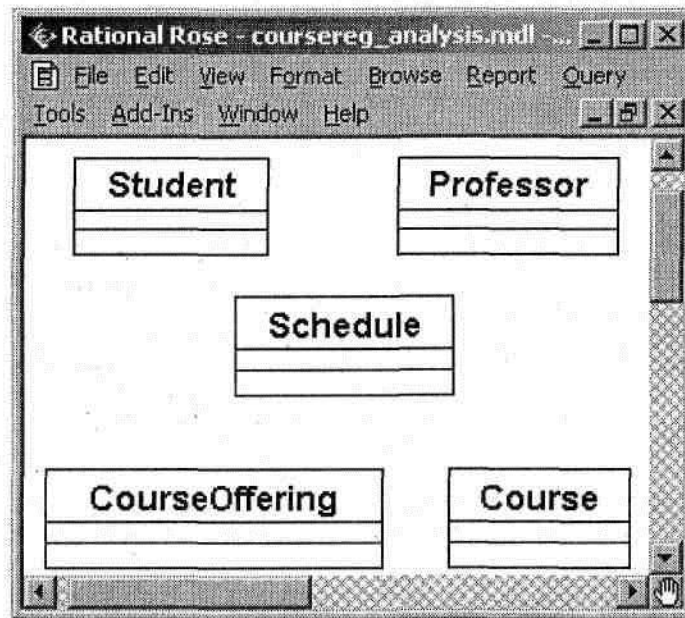


Рис. 4.5. Классы анализа системы регистрации

Архитектурные уровни образуют иерархию уровней представления любой крупномасштабной системы. В практике разработки таких систем существует ряд типовых решений — архитектурных образцов, среди которых наиболее распространенным является образец «Уровни» (Layers). Этот образец можно описать следующим образом:

Наименование образца:

«Уровни».

Контекст:

Крупномасштабные системы, нуждающиеся в декомпозиции.

Проблема:

Архитектура крупномасштабной системы должна удовлетворять следующим требованиям:

- компоненты системы должны иметь возможность замены;

- изменения в одних компонентах не должны сильно затрагивать другие;
- однородные функции должны группироваться вместе;
- размер компонентов не должен быть слишком большим.

Решение:

Предлагается базовый вариант, включающий следующие уровни (сверху вниз):

- прикладной (Application Subsystems) — набор компонентов, реализующих основную функциональность системы, отраженную в вариантах использования;
- бизнес-уровень (Business-specific) — набор компонентов, специфичных для конкретной предметной области;
- промежуточный (Middleware) — различные платформи-независимые сервисы (библиотеки пользовательского интерфейса, брокеры запросов и др.);
- системный (System software) — ПО для вычислительной и сетевой инфраструктуры (ОС, сетевые протоколы и др.).

Архитектурные уровни представляются в модели в виде пакетов со стереотипом «layer». Количество и структура уровней зависят от сложности предметной области и среды реализации. В рамках архитектурного анализа определяется начальная структура модели (набор пакетов и их зависимостей) и рассматриваются только верхние уровни (прикладной и бизнес-уровень).

4.3.2.

АНАЛИЗ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Анализ вариантов использования выполняется проектировщиками и включает в себя:

- идентификацию классов, участвующих в реализации потоков событий варианта использования;

- распределение поведения, реализуемого вариантом использования, между классами (определение обязанностей классов);
- определение атрибутов и ассоциаций классов;
- унификацию классов анализа.

Идентификация классов, участвующих в реализации потоков событий варианта использования

В потоках событий варианта использования выявляются классы трех типов.

Граничные классы (*Boundary*) — служат посредниками при взаимодействии внешних объектов с системой. Как правило, для каждой пары «действующее лицо — вариант использования» определяется один граничный класс. Типы граничных классов: пользовательский интерфейс (обмен информацией с пользователем без деталей интерфейса — кнопок, списков, окон), системный интерфейс и аппаратный интерфейс (используемые протоколы без деталей их реализации).

Классы-сущности (*Entity*) — представляют собой основные абстракции (понятия) разрабатываемой системы, рассматриваемые в рамках конкретного варианта использования. Источники выявления классов-сущностей: основные абстракции, созданные в процессе архитектурного анализа, глоссарий, описание потоков событий вариантов использования, сущности, описанные в модели бизнес-анализа (при наличии бизнес-модели).

Управляющие классы (*Control*) — обеспечивают координацию поведения объектов в системе. Они могут отсутствовать в некоторых вариантах использования, ограничивающихся простыми манипуляциями с хранимыми данными. Как правило, для каждого варианта использования определяется один управляющий класс. Примеры управляющих классов: менеджер транзакций, координатор ресурсов, обработчик ошибок.

Классы анализа отражают функциональные требования к системе и моделируют объекты предметной области. Совокупность классов анализа представляет собой начальную концептуальную модель системы.

Пример набора классов, участвующих в реализации варианта использования «Зарегистрироваться на курсы», приведен на рис. 4.6.

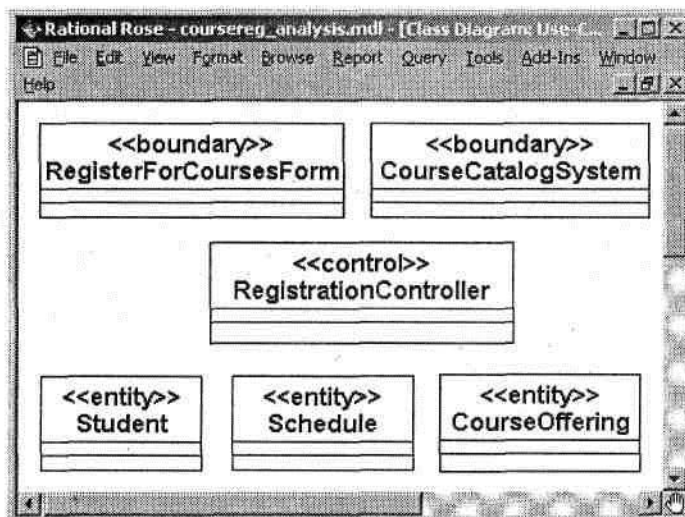


Рис. 4.6. Классы, участвующие в реализации варианта использования «Зарегистрироваться на курсы»

Распределение поведения, реализуемого вариантом использования, между классами

Квалифицированное распределение обязанностей между классами является наиболее важной частью объектно-ориентированного анализа. Исходя из назначения трех выделенных типов классов, можно кратко охарактеризовать распределение обязанностей между ними:

- граничные классы отвечают за взаимодействие с внешней средой системы (действующими лицами);
- классы-сущности отвечают за хранение и манипулирование данными;
- управляющие классы координируют потоки событий варианта использования.

Более детальное распределение обязанностей (в виде операций классов) выполняется с помощью диаграмм взаимодействия (диаграмм последовательности и кооперативных диаграмм).

В процессе анализа конкретного варианта использования в первую очередь строится диаграмма последовательности (одна или более), описывающая основной поток событий и его подчиненные потоки. Для каждого альтернативного потока событий строится отдельная диаграмма (из соображений простоты

наглядного восприятия). Нецелесообразно описывать тривиальные потоки событий (например, в потоке участвует только один объект).

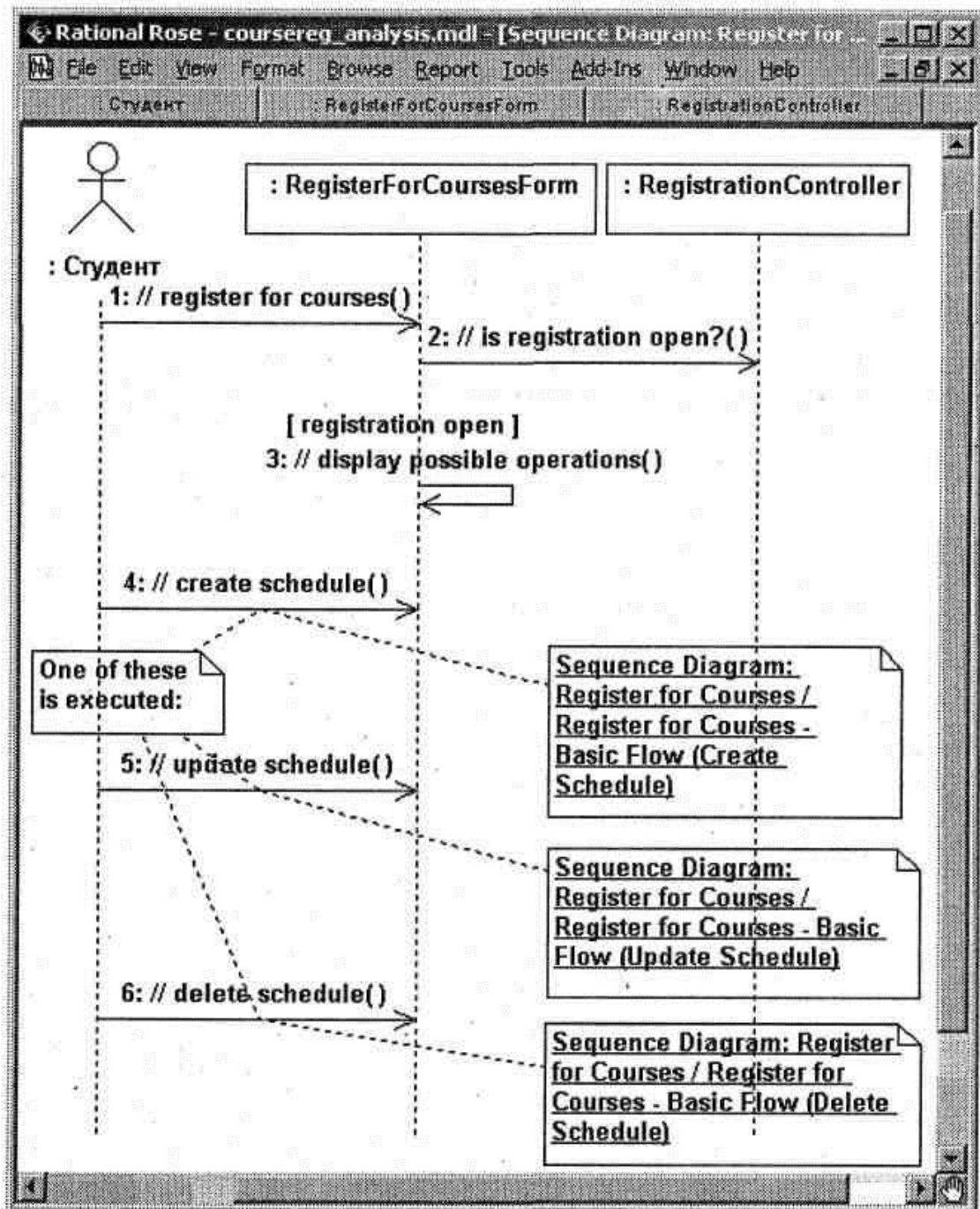


Рис. 4.7. Диаграмма последовательности «Зарегистрироваться на курсы» — Основной поток событий

На рис. 4.7—4.11 приведены диаграммы последовательности и кооперативные диаграммы для основного потока событий варианта использования «Зарегистрироваться на курсы».

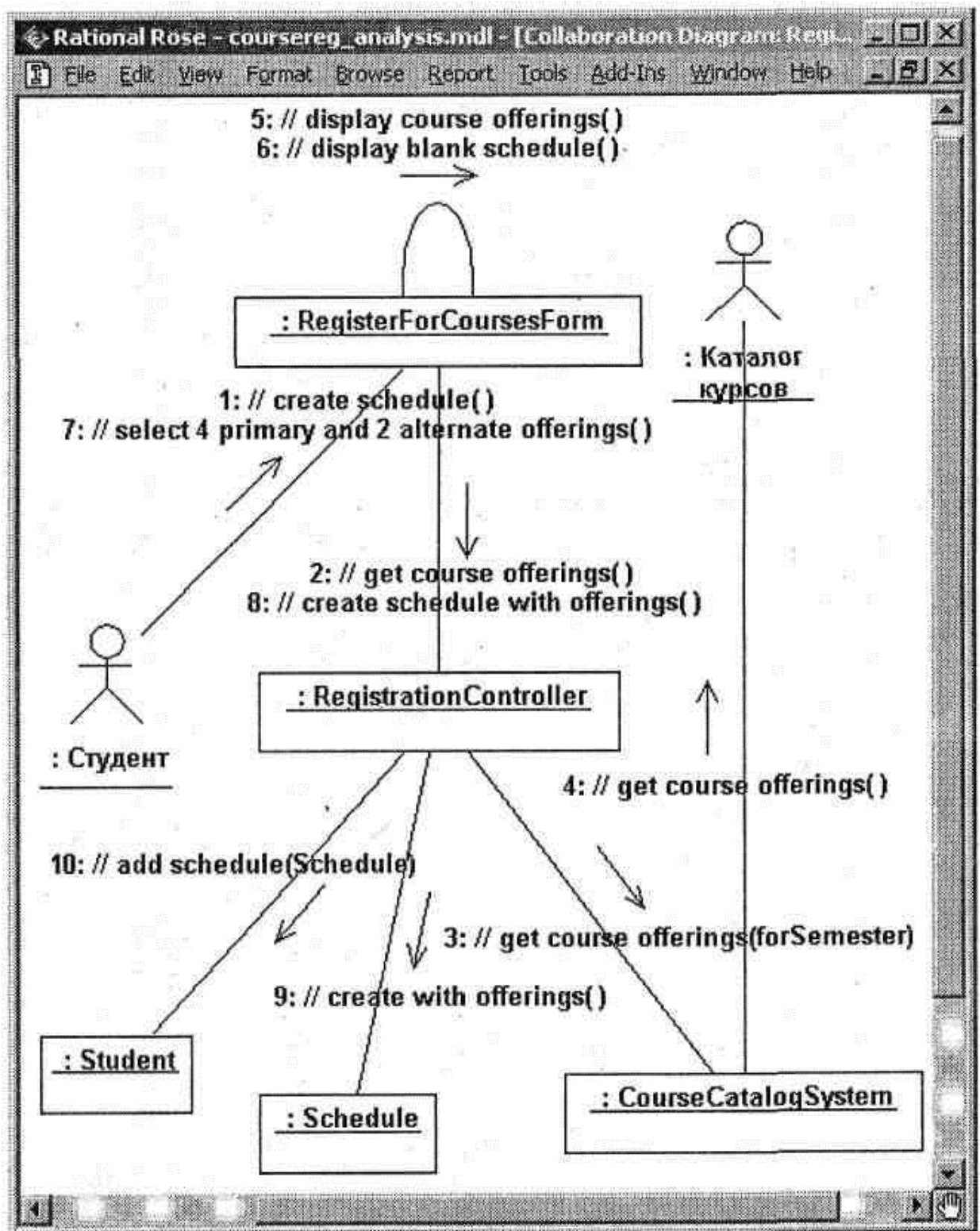


Рис. 4.8. Кооперативная диаграмма «Зарегистрироваться на курсы» - подчиненный поток «Создать график»

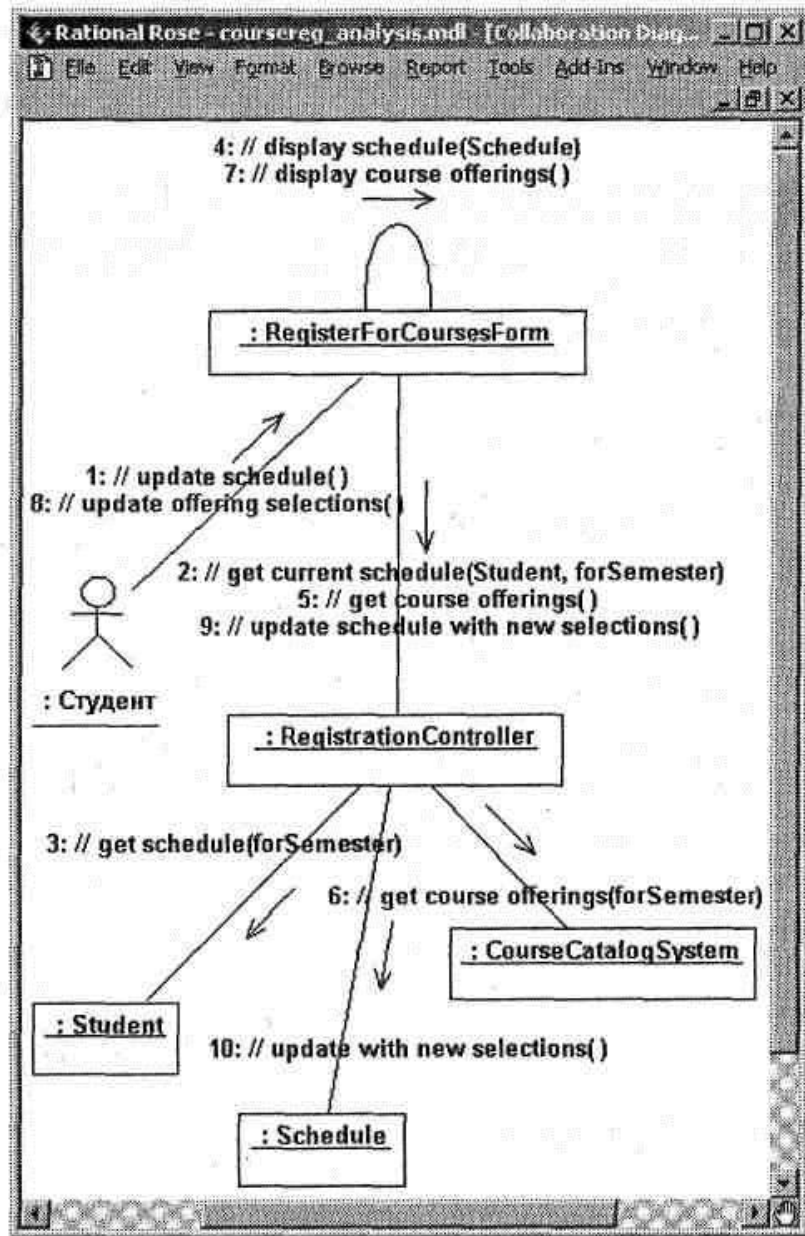


Рис. 4.9. Кооперативная диаграмма «Зарегистрироваться на курсы» — подчиненный поток «Обновить график»

На последней диаграмме (см. рис. 4.11) присутствует объект дополнительного класса-сущности PrimaryScheduleOfferingInfo. Его назначение будет описано ниже, при рассмотрении связей между классами.

Обязанности каждого класса определяются исходя из сообщений на диаграммах взаимодействия и документируются в классах в виде операций «анализа», которые появляются там в процессе построения диаграмм взаимодействия (соотнесения сообщений с операциями).

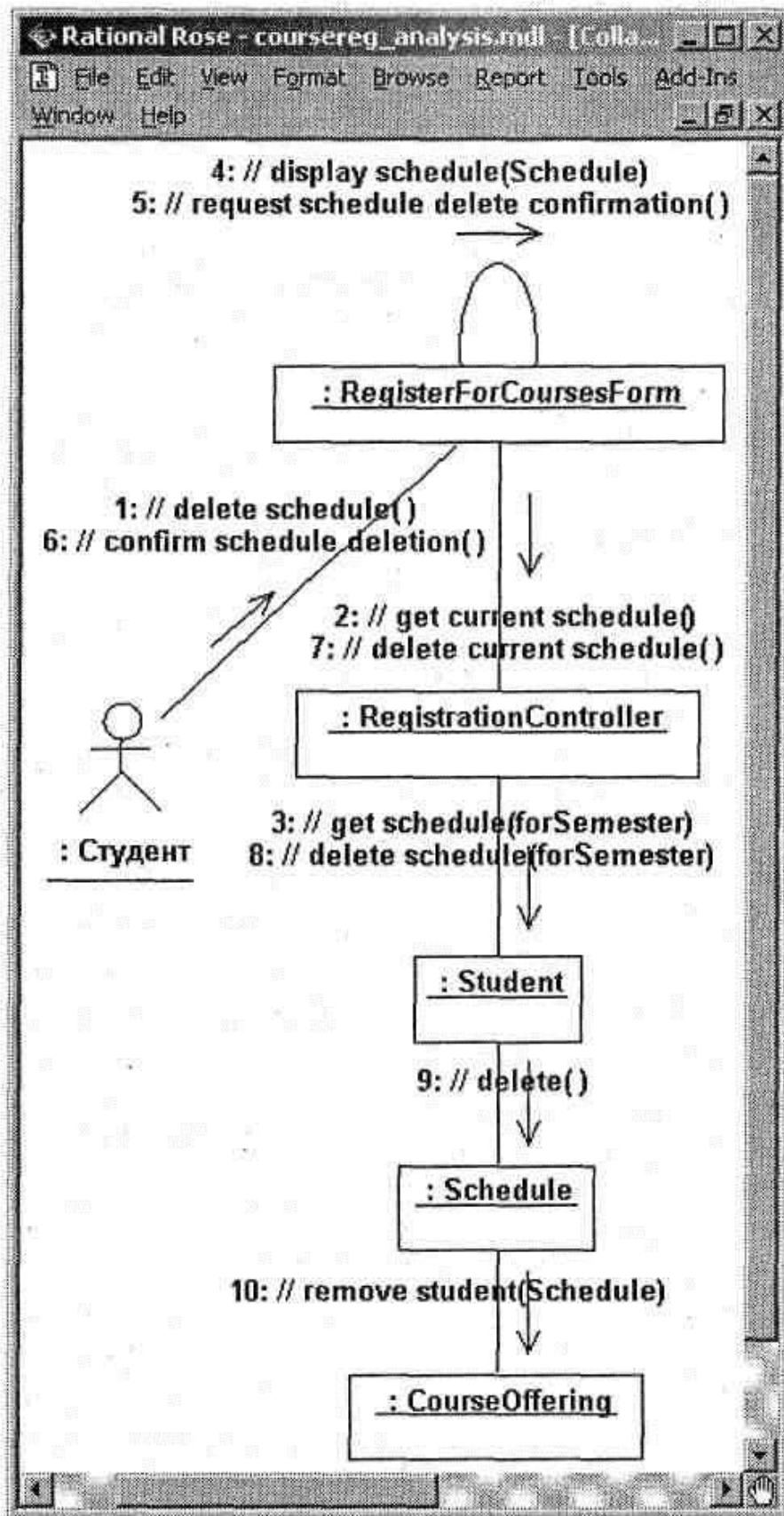


Рис. 4.10. Кооперативная диаграмма «Зарегистрироваться на курсы» — подчиненный поток «Удалить график»

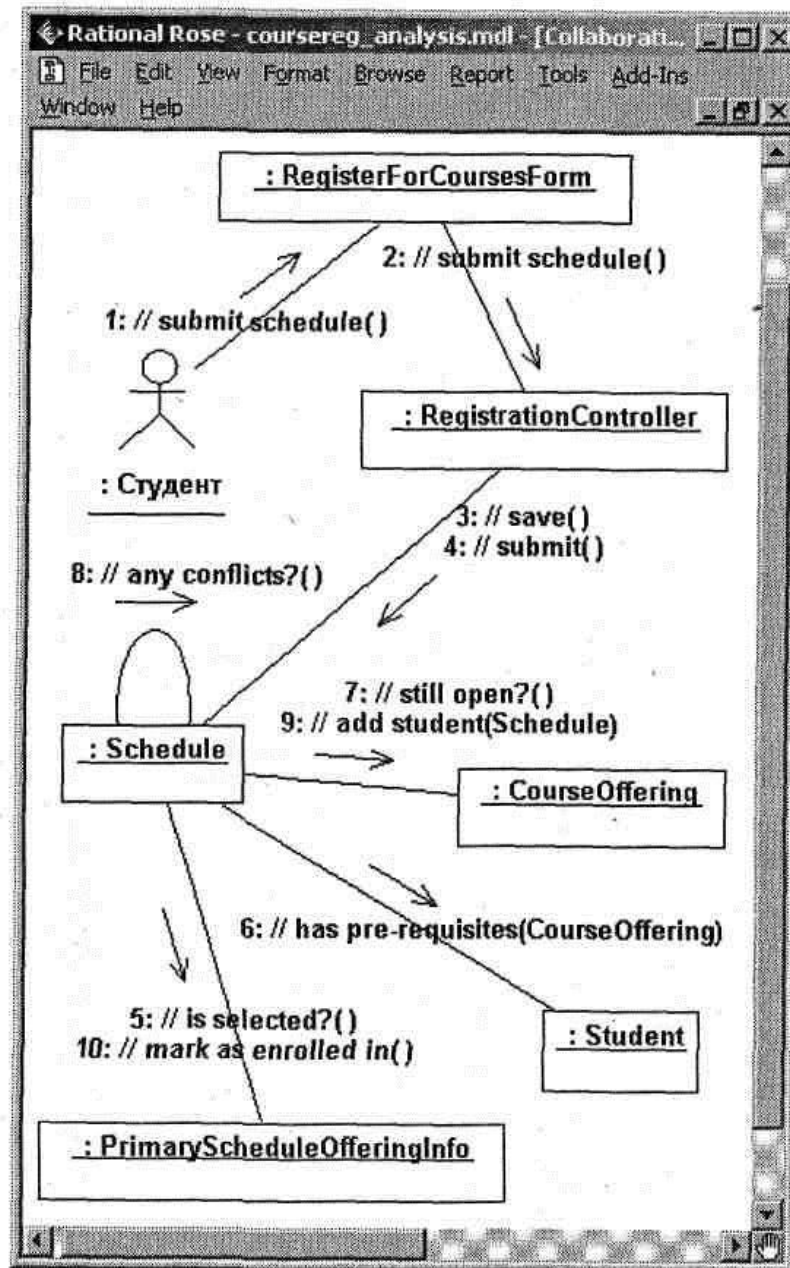


Рис. 4.11. Кооперативная диаграмма «Зарегистрироваться на курсы» — подчиненный поток «Принять график»

Каждая операция «анализа» класса соответствует некоторому сообщению, принимаемому объектами данного класса. В процессе проектирования каждая операция «анализа» преобразуется в одну или более операций класса, которые в дальнейшем будут реализованы в коде системы.

Так, диаграмма классов варианта использования «Зарегистрироваться на курсы» (см. рис. 4.6) после построения диаграмм взаимодействия должна принять следующий вид (рис. 4.12).

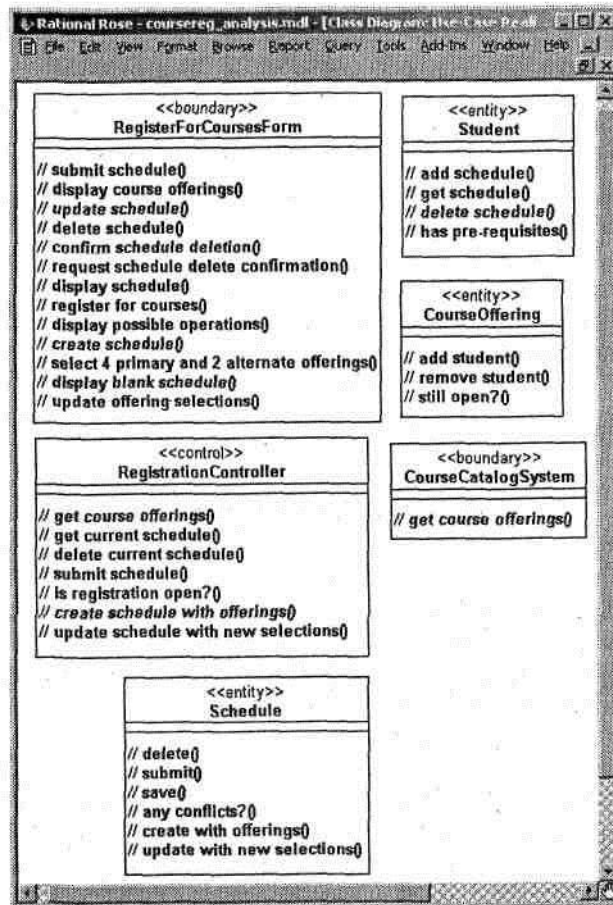


Рис. 4.12. Диаграмма классов с операциями «анализа»

При построении диаграмм взаимодействия возникают проблемы правильного распределения обязанностей между классами. Для их решения существует ряд образцов¹, некоторые из них приведены ниже.

Образец «Information Expert»

Проблема:

Нужно определить наиболее общий принцип распределения обязанностей между классами. В системе могут быть определены сотни классов, выполняющие тысячи обязанностей. При правильном их распределении система становится гораздо проще для понимания, сопровождения и развития. Кроме того, появляется возможность повторного использования уже разработанных компонентов в последующих приложениях.

¹ Ларман К. Применение UML и шаблонов проектирования. - 2-е изд.: Пер. с англ. — М.: Вильямс, 2002.

Решение:

Следует назначить обязанность информационному эксперту — классу, у которого имеется информация, требуемая для выполнения обязанности.

Пример:

При выполнении подчиненного потока событий «Обновить график» варианта использования «Зарегистрироваться на курсы» (рис. 4.9) студент-пользователь должен получить доступ к своему графику прежде, чем изменить его. Согласно образцу «Information Expert», нужно определить, объект какого класса содержит информацию, необходимую для доступа к графику. На эту роль информационного эксперта, очевидно, претендует объект класса-сущности Student, поскольку график принадлежит именно ему. Поэтому сообщение 3 «get schedule(forSemester)» должно быть направлено от контроллера объекту класса Student. После того, как студент получит график и внесет в него необходимые изменения, они должны быть зафиксированы в объекте Schedule. В данном случае уже сам объект Schedule будет играть роль информационного эксперта, поскольку он непосредственно доступен контроллеру, и сообщение 10 «update with new selections» будет направлено именно ему.

Следствия:

При распределении обязанностей образец Information Expert используется гораздо чаще любого другого образца. Большинство сообщений на приведенных выше диаграммах взаимодействия соответствуют данному образцу. В нем определены основные принципы, которые уже давно используются в объектно-ориентированном анализе и проектировании. Образец Information Expert не содержит неясных или запутанных идей и отражает обычный интуитивно понятный подход. Он заключается в том, что объекты осуществляют действия, связанные с имеющейся у них информацией. Если информация распределена между различными объектами, то при выполнении общей задачи они должны взаимодействовать с помощью сообщений.

Образец Information Expert, как и многие другие понятия объектной технологии, имеет соответствующую аналогию в реальном мире. Например, в орга-

низации обязанности обычно распределяются между теми служащими, у которых имеется необходимая для выполнения поставленной задачи информация. Ответственность за создание отчета о прибыли и убытках должен нести тот из служащих, кто имеет доступ ко всей информации, необходимой для создания такого отчета. Возможно, лучше всего для выполнения этой обязанности подойдет руководитель финансового отдела предприятия. Программные объекты взаимодействуют между собой и обмениваются информацией так же, как люди. Начальник финансового отдела компании может запросить требуемые данные у бухгалтеров, работающих со счетами по дебиторской и кредиторской задолженности, чтобы составить отдельные отчеты по кредиту и дебету.

В некоторых ситуациях применение образца Information Expert нежелательно. Например, в системе регистрации нужно определить, какой объект должен отвечать за сохранение информации об учебных курсах в базе данных. Информация, подлежащая сохранению, «известна» объекту Course, а значит, согласно образцу Information Expert, на класс Course следует возложить обязанность по сохранению. Логическим следствием такого рассуждения является вывод о том, что каждый объект должен отвечать за сохранение себя в базе данных. Однако при этом возникает проблема перегруженности класса лишними обязанностями, поскольку класс Course должен содержать методы обращения к базе данных, т.е. должен быть связан с вызовом операторов языка SQL или сервисов JDBC (Java Database Connectivity). Тогда этот класс не будет относиться к предметной области и моделировать учебные курсы. Кроме того, подобные действия будут дублироваться во многих других классах, информация которых подлежит постоянному хранению.

Все эти проблемы приводят к нарушению основного архитектурного принципа проектирования с разделением основных функций системы на уровни, отраженного в образце «Уровни». Разнородные функции не должны реализовываться в одном и том же компоненте. С этой точки зрения класс Course не должен отвечать за сохранение информации в базе данных.

Основное достоинство образца Information Expert - поддержка инкапсуля-

ции. Для выполнения требуемых задач объекты используют собственные данные. Необходимое поведение системы обеспечивается несколькими классами, содержащими требуемую информацию. Это приводит к определениям классов, которые гораздо проще понимать и поддерживать.

Образец «Creator»

Проблема:

Нужно определить, кто должен отвечать за создание нового экземпляра некоторого класса. Создание новых объектов в объектно-ориентированной системе является одним из стандартных видов деятельности. Следовательно, при назначении обязанностей, связанных с созданием объектов, полезно руководствоваться некоторым основным принципом.

Решение:

Следует назначить классу В обязанность создавать экземпляры класса А, если выполняется одно из следующих условий:

- класс В агрегирует, содержит или активно использует объекты класса А;
- класс В обладает данными инициализации, которые будут передаваться объектам класса А при их создании (т.е. класс В является информационным экспертом).

Класс В при этом определяется как создатель (creator) объектов класса А.

Если несколько классов удовлетворяют этим условиям, то предпочтительнее использовать в качестве создателя класс, агрегирующий или содержащий класс А.

Пример:

При выполнении подчиненного потока событий «Создать график» варианта использования «Зарегистрироваться на курсы» (см. рис. 4.8) необходимо решить, кто должен отвечать за создание нового графика в системе. На рисунке 4.13 показано два возможных варианта решения этой задачи.

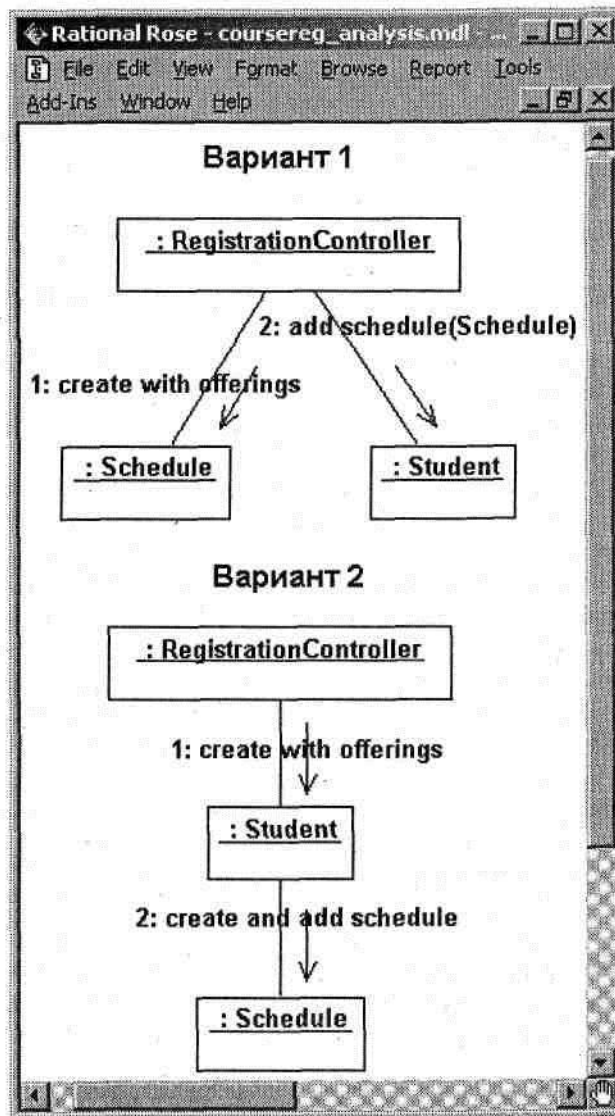


Рис. 4.13. Два варианта создания графика

В диаграмме на рис. 4.8 выбран вариант 1. Однако согласно образцу «Creator» наилучшим решением является вариант 2 (новый объект класса Schedule создается классом Student, а не RegistrationController, поскольку именно Student удовлетворяет первому из перечисленных выше условий).

Следствия:

Образец «Creator» определяет способ распределения обязанностей, связанный с процессом создания объектов. В объектно-ориентированных системах эта задача является одной из наиболее распространенных. Основным назначением образца Creator является выявление объекта-создателя, который при возникновении любого события должен быть связан со всеми созданными им объектами.

При таком подходе обеспечивается низкая степень связанности объектов.

В некоторых случаях в качестве создателя выбирается класс, который содержит данные инициализации, передаваемые объекту во время его создания. На самом деле это пример использования образца Information Expert. В процессе создания инициализирующие данные передаются с помощью метода инициализации некоторого вида, такого, как конструктор языка программирования.

В некоторых сложных случаях вместо данного образца предпочтительнее использовать известный образец Factory (Фабрика)¹ и делегировать обязанность создания объектов вспомогательному классу.

Образец «Low Coupling»

Проблема:

Нужно распределить обязанности между классами таким образом, чтобы снизить взаимное влияние изменений в них и повысить возможность повторного использования.

Решение:

Следует распределить обязанности таким образом, чтобы обеспечить слабую связанность. *Связанность* (coupling) — это мера, определяющая насколько жестко один элемент связан с другими элементами, или каким количеством данных о других элементах он обладает. Элемент со слабой связанностью зависит от небольшого числа других элементов. Класс с сильной связанностью зависит от множества других классов. Наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем.

- Изменения в связанных классах приводят к локальным изменениям в данном классе.
- Затрудняется понимание каждого класса в отдельности.
- Усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

Пример:

¹Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес: Пер. с англ. -СПб.: Питер, 2001

Рассмотрим подчиненный поток событий «Создать график» варианта использования «Зарегистрироваться на курсы» (см. рис. 4.8). На данной диаграмме при создании нового графика в системе выбран вариант 1 (рис. 4.13). Однако согласно образцу «Low Coupling» наилучшим решением является вариант 2, поскольку при этом у класса `RegistrationController` будет на одну связь меньше (т.е. будет обеспечена более слабая связанность).

Следствия:

Образец `Low Coupling` поддерживает независимость классов, что, в свою очередь, повышает возможности повторного использования и обеспечивает более высокую эффективность приложения. Его нельзя рассматривать изолированно от других образцов, таких как `Information Expert` и `High Cohesion`. Он также обеспечивает выполнение одного из основных принципов проектирования, применяемых при распределении обязанностей.

Не существует абсолютной меры для определения слишком сильной связанности. Важно лишь понимать связанность объектов и не упустить тот момент, когда дальнейшее повышение связанности может привести к возникновению проблем. В целом следует руководствоваться таким принципом: классы, которые являются достаточно общими по своей природе и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную связанность с другими классами.

Крайним случаем при реализации образца `Low Coupling` является полное отсутствие связывания между классами. Такая ситуация тоже нежелательна, поскольку основная идея объектного подхода выражается в системе связанных объектов, которые «общаются» между собой посредством передачи сообщений. При слишком частом использовании принципа слабого связывания система будет состоять из нескольких изолированных сложных активных объектов, самостоятельно выполняющих все операции, и множества пассивных объектов, основная функция которых сводится к хранению данных. Поэтому при создании объектно-ориентированной системы должна присутствовать некоторая оптимальная степень связывания между объектами, позволяющая выполнять основ-

ные функции посредством взаимодействия этих объектов.

Не следует применять данный образец, когда создаются связи с устойчивыми элементами. Сильная связанность с такими элементами не представляет проблемы. Например, приложение Java 2 Enterprise Edition можно жестко связать с библиотеками Java, поскольку они достаточно стабильны.

Сильная связанность сама по себе не является проблемой. Проблемой является жесткое связывание с неустойчивыми элементами. Важно понимать, что без убедительной мотивации не следует во что бы то ни стало бороться за уменьшение связанности объектов.

Образец «High Cohesion»

Проблема:

Нужно распределить обязанности между классами таким образом, чтобы каждый класс не выполнял много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению таких же проблем, как у классов с сильной связанностью.

Решение:

Следует распределить обязанности таким образом, чтобы обеспечить сильное сцепление. В терминах объектно-ориентированного проектирования сцепление (cohesion) — это мера связанности и непротиворечивости обязанностей класса. Считается, что элемент обладает сильным сцеплением, если его обязанности тесно связаны между собой, и он не выполняет излишнего объема работы. В роли таких элементов могут выступать классы, подсистемы, модули и т.д.

Классы со слабым сцеплением, как правило, выполняют обязанности, которые можно легко распределить между другими классами.

Пример:

Используем тот же пример, что и для предыдущего образца (см. рис. 4.13). Согласно образцу «High Cohesion», наилучшим решением также является вариант 2, поскольку при этом класс `RegistrationController` делегирует обязанность создания нового объекта класса `Shedule` классу `Student`, и у самого класса `Regis-`

trationController будет на одну обязанность меньше (т.е. его сцепление будет сильнее).

Следствия:

Как правило, класс с сильным сцеплением содержит сравнительно небольшое число методов, которые функционально тесно связаны между собой, и не выполняет слишком много функций. Он взаимодействует с другими классами для выполнения более сложных задач. Высокая степень однотипной функциональности в сочетании с небольшим числом операций упрощают поддержку и модификацию класса, а также возможность его повторного использования.

Образец High Cohesion, как и другие понятия объектно-ориентированной технологии проектирования, имеет аналогию в реальном мире. Известно, что человек, выполняющий большое число разнородных обязанностей (особенно тех, которые можно легко распределить между другими людьми), работает не очень эффективно. Особенно это касается менеджеров, которые не умеют распределять обязанности между своими подчиненными.

Слабая связанность и сильное сцепление — основные принципы проектирования ПО. Объектное проектирование полностью с ними согласуется, поскольку одним из его базовых принципов является модульность.

Однако существуют ситуации, когда слабое сцепление оказывается оправданным. Одна из таких ситуаций возникает в том случае, когда обязанности или код группируются в одном классе или компоненте для упрощения его поддержки одним человеком. Другой пример слабого сцепления имеет отношение к распределенным серверным объектам. Поскольку быстродействие системы определяется производительностью удаленных объектов и их взаимодействием, иногда желательно создать несколько крупных серверных объектов со слабым сцеплением, предоставляющих интерфейс многим операциям. Это приведет к уменьшению числа удаленных вызовов и, как следствие, к повышению производительности.

Набор обязанностей классов, полученный в результате их распределения, должен быть проанализирован на предмет выявления и устранения следующих

проблем:

- дублирования одинаковых обязанностей в различных классах;
- противоречивых обязанностей в рамках класса;
- классов с одной обязанностью или вообще без обязанностей;
- классов, взаимодействующих с большим количеством других классов.

Определение атрибутов и ассоциаций классов

Атрибуты классов анализа определяются, исходя из знаний о предметной области, требований к системе, глоссария и бизнес-модели. В процессе анализа атрибуты определяются только для классов-сущностей, и они имеют тот же смысл, что и в модели ERM. Атрибуты должны быть простыми, т.е. выразимыми с помощью простых типов данных (одной из распространенных ошибок является моделирование сложного понятия предметной области в виде атрибута). Так, после определения атрибутов для классов-сущностей, показанных на рис. 4.12, они примут следующий вид (рис. 4.14).

Связи между классами (ассоциации) определяются в два этапа:

1. Начальный набор связей определяется на основе анализа кооперативных диаграмм. Если два объекта взаимодействуют (обмениваются сообщениями), между ними на кооперативной диаграмме должна существовать связь (путь взаимодействия), которая преобразуется в двунаправленную ассоциацию между соответствующими классами. Если сообщения между некоторой парой объектов передаются только в одном направлении, то для соответствующей ассоциации вводится направление навигации.

2. Анализируются и уточняются ассоциации между классами-сущностями. Задаются мощности ассоциаций, могут использоваться множественные ассоциации, агрегации, обобщения и ассоциации-классы.

Следует избегать использования избыточных ассоциаций и сосредоточиться на тех ассоциациях, для которых данные о связи должны сохраняться в течение некоторого времени. Это достаточно важный момент. Если модель содержит N различных классов анализа, то между ними можно установить $N*(N-1)$ ассоциацию, большинство из которых будет просто создавать «визуальный

шум» и ухудшать наглядность диаграмм. Поэтому при добавлении ассоциаций нужно придерживаться принципа минимализма.

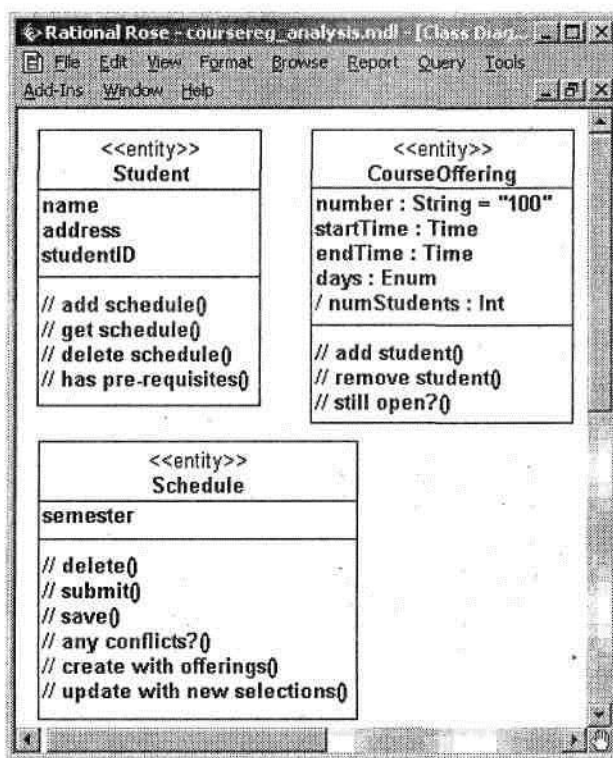


Рис. 4.14. Классы с операциями «анализа» и атрибутами

Результаты определения связей между классами, принимающими участие в реализации варианта использования «Зарегистрироваться на курсы», показаны на рис. 4.15 — 4.17.

На рис. 4.15 показаны только классы-сущности. Агрегация между классами `Student` и `Schedule` отражает тот факт, что каждый график является собственностью конкретного студента, принадлежит только ему. Предполагается также, что в системе будет храниться не только график текущего семестра, а все графики студента за разные семестры. Между классами `Schedule` и `CourseOffering` введено две ассоциации, поскольку конкретный курс может входить в график студента в качестве основного (не более четырех курсов) и альтернативного (не более двух курсов). К классу `Student`; добавлены два новых подкласса — `FulltimeStudent`: (студент очного отделения) и `ParttimeStudent` (студент вечернего отделения).

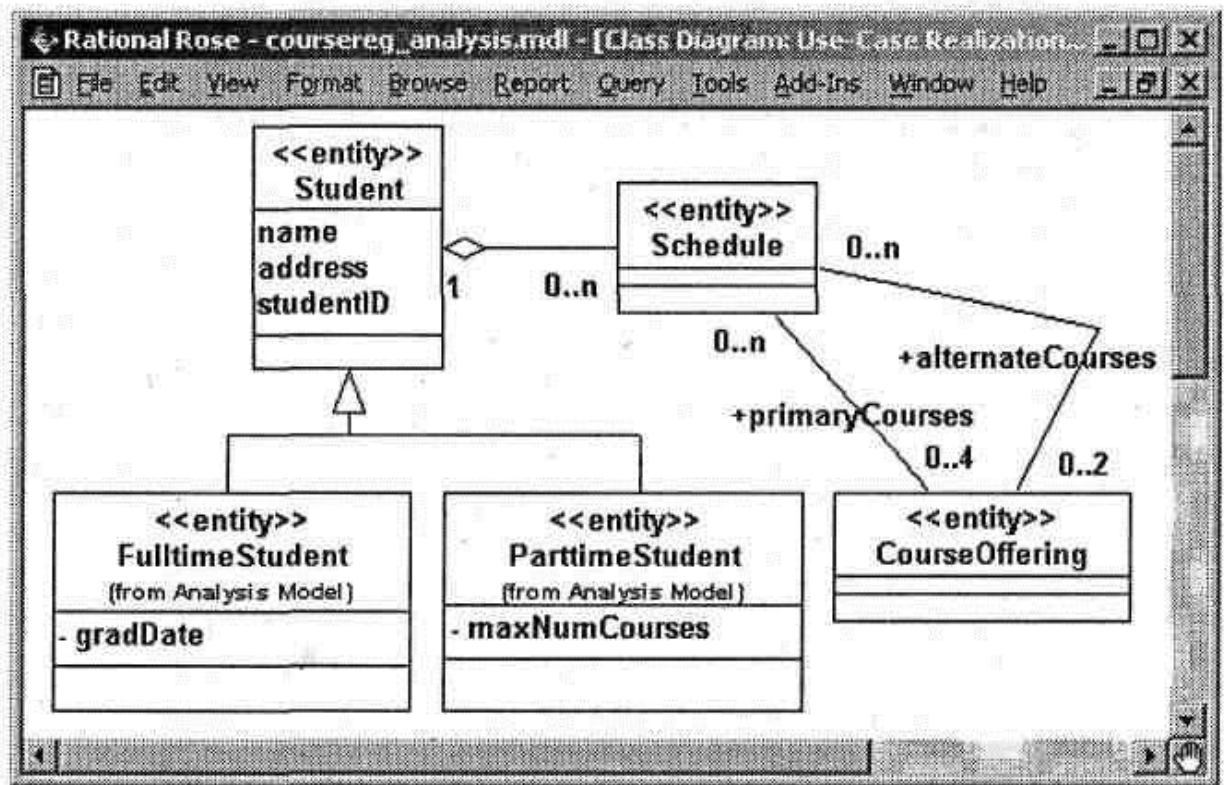


Рис. 4.15. Диаграмма классов-сущностей

На рис. 4.16 показаны ассоциации-классы, представляющие свойства связей между классами Schedule и CourseOffering. Ассоциация, связывающая график и альтернативный курс, имеет только один атрибут — статус курса в конкретном графике (status), который может принимать значения «включен в график», «отменен», «внесен в список курса» и «зафиксирован в графике». Если курс в процессе закрытия регистрации переходит из альтернативного в основные, то к соответствующей ассоциации добавляется атрибут «оценка» (grade). Таким образом, ассоциация-класс PrimaryScheduleOfferingInfo наследует свойства ассоциации-класса ScheduleOfferingInfo (атрибуты и операции, содержащиеся в этом классе, относятся как к основным, так и к альтернативным курсам) и добавляет свои собственные (оценка и окончательное включение курса в график могут иметь место только для основных курсов), что и показано с помощью связи обобщения.

ScheduleOfferingInfo

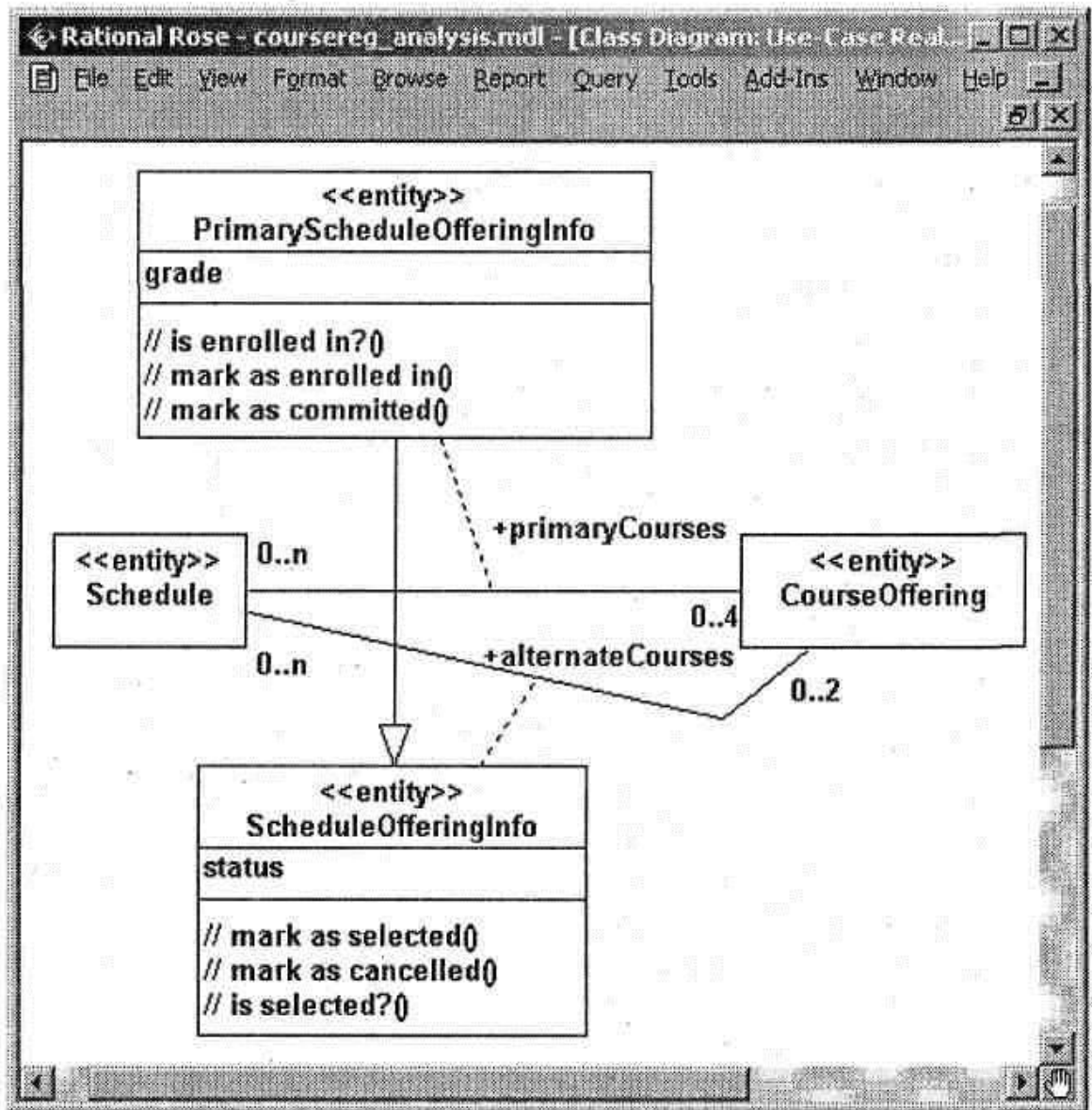


Рис. 4.16. Пример ассоциаций-классов

На рис. 4.17 показана полная диаграмма классов варианта использования «Зарегистрироваться на курсы» (без атрибутов и операций). Ассоциации между граничными и управляющими классами, а также между управляющими классами и классами-сущностями введены на основе анализа кооперативных диаграмм и в отличие от устойчивых структурных (семантических) связей между сущностями отражают связи, динамически возникающие между соответствующими объектами в потоке управления (в процессе работы приложения). Поскольку для ассоциаций это не свойственно, в дальнейшем (в процессе проектирования) они могут быть преобразованы в зависимости.

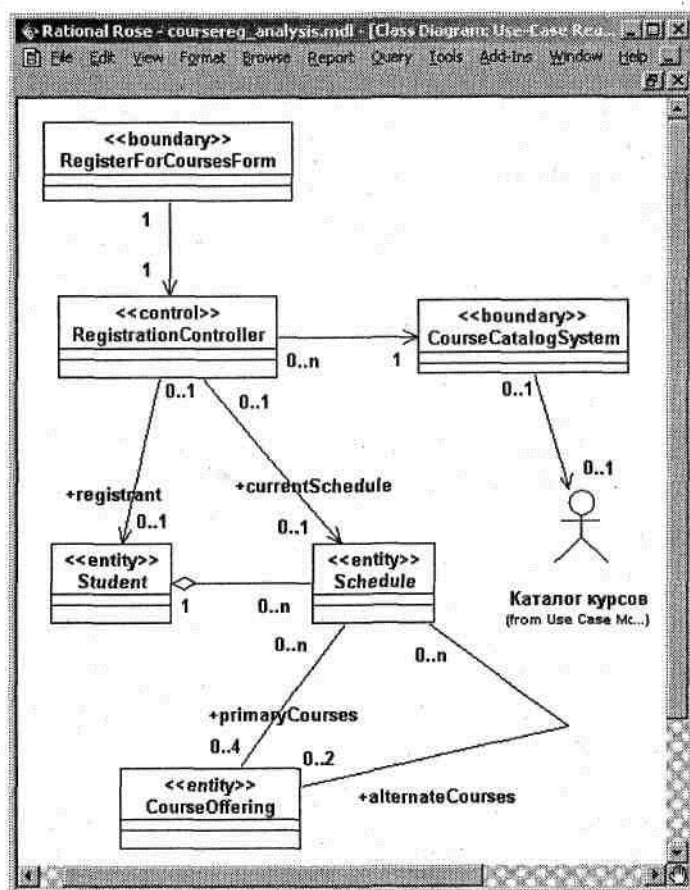


Рис. 4.17. Полная диаграмма классов (без атрибутов и операций)

Унификация классов анализа

Унификация классов анализа заключается в проверке созданных классов на предмет выполнения следующих условий:

- имя и описание каждого класса «анализа» должно отражать сущность его роли в системе. Имена должны быть уникальными;
- классы с одинаковым поведением, или представляющие одно и то же явление, должны объединяться;
- классы-сущности с одинаковыми атрибутами должны объединяться (даже если их поведение отличается). Объединенный класс будет обладать общим поведением.

При обновлении классов должны, при необходимости, обновляться описания вариантов использования.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ

ПРОЕКТИРОВАНИЕ

Целью объектно-ориентированного проектирования является адаптация предварительного системного проекта (набора классов «анализа»), составляющего стабильную основу архитектуры системы, к среде реализации с учетом всех нефункциональных требований.

Объектно-ориентированное проектирование включает два вида деятельности:

- проектирование архитектуры системы;
- проектирование элементов системы.

Их изложение будет сопровождаться примерами из системы регистрации учебного заведения.

4.4.1.

ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СИСТЕМЫ

Проектирование архитектуры системы выполняется архитектором системы и включает в себя:

- идентификацию архитектурных решений и механизмов, необходимых для проектирования системы;
- анализ взаимодействий между классами анализа, выявление подсистем и интерфейсов;
- формирование архитектурных уровней;
- проектирование структуры потоков управления;
- проектирование конфигурации системы.

Идентификация архитектурных решений и механизмов

В процессе проектирования определяются детали реализации архитектурных механизмов, обозначенных в процессе анализа. Поскольку практически все механизмы — это некоторые типовые решения (образцы), они документируются в проекте (модели) с помощью кооперации со стереотипом «mechanism», при этом структурная часть механизма описывается с помощью диаграмм классов, а поведение — с помощью диаграмм взаимодействия.

В качестве примера рассмотрим механизм хранения данных в БД. Предположим, что в проекте системы регистрации в качестве языка программирования используется Java. Поскольку существующая система каталога курсов функционирует на основе реляционной СУБД, таким механизмом, обеспечивающим доступ к этой внешней базе данных, должен быть JDBC (Java Database Connectivity). На рис. 4.18 показана диаграмма классов образца, описывающего JDBC.

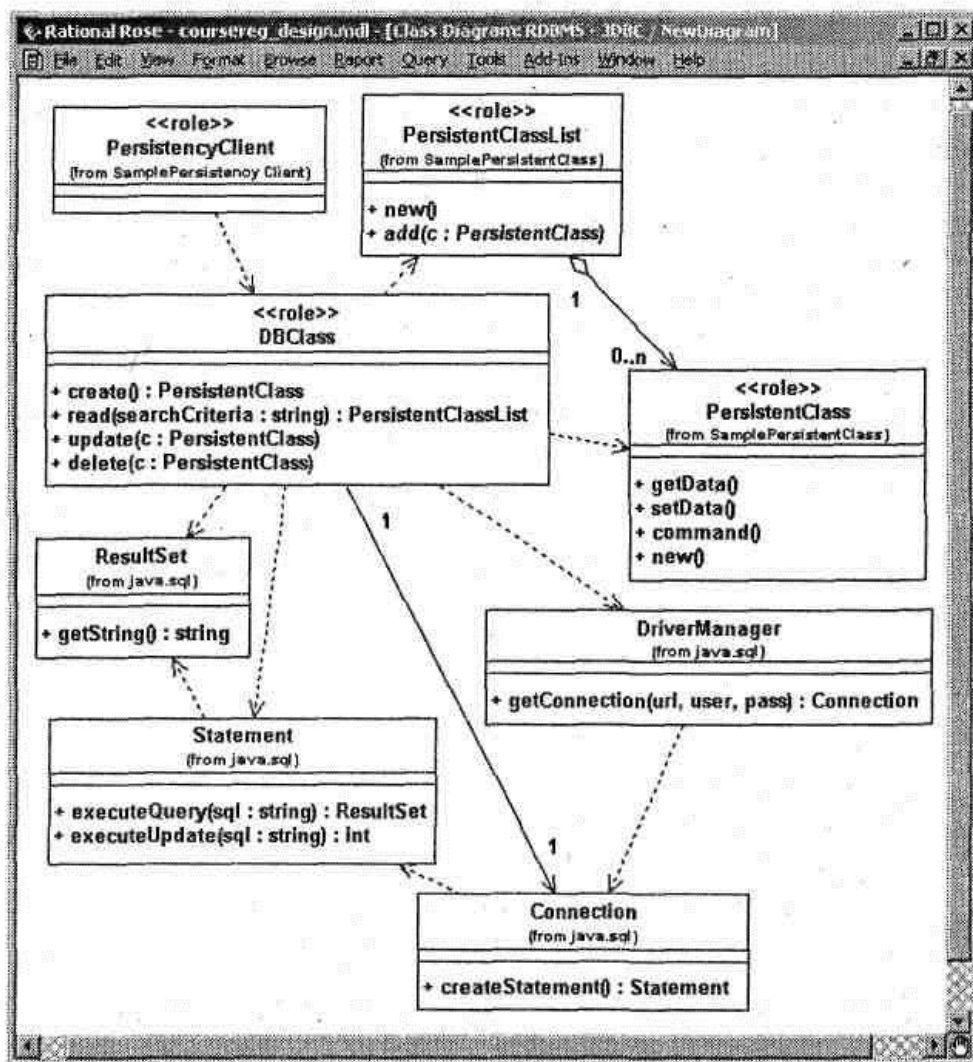


Рис. 4.18. Диаграмма классов кооперации, описывающей JDBC

Все классы, изображенные на данной диаграмме, можно разделить на две группы:

- Собственно классы JDBC (`DriverManager`, `Connection`, `Statement`, `ResultSet`), которые отвечают за реализацию запроса к БД (выполнение оператора SQL). Эти классы принадлежат к архитектурному уровню `Middleware` и входят в соответствующий пакет.
- Классы со стереотипом `<<role>>`, являющиеся «заполнителями» (placeholders) реальных классов, создаваемых разработчиком системы. Они выполняют следующее назначение:

`DBClass` — отвечает за чтение и запись данных. Класс такого типа создается для каждого устойчивого (persistent) класса, или, иначе говоря, класса, данные которого будут храниться в некоторой БД (в данном случае — в таблицах реляционной БД). Например, в системе регистрации на курсы в процессе анализа в соответствии с образцом `Information Expert` определено, что класс-сущность `Course` должен отвечать за сохранение информации об учебных курсах в базе данных. Однако при этом, как было сказано в данном образце, возникает проблема перегруженности класса лишними обязанностями, поскольку класс `Course` должен непосредственно содержать вызовы сервисов JDBC. Разделение основных функций системы на уровни (применение образца «Уровни») в данном случае означает, что обязанности взаимодействия с БД выносятся из класса `Course` в класс `DBCourse`.

`PersistencyClient` — класс, запрашивающий создание, чтение, обновление или удаление данных.

`PersistentClass` — класс-сущность, объекты которого содержат необходимые данные.

`PersistentClassList` — список объектов, являющихся результатом запроса к БД — выполнения операции `DBClass.read()`.

Выполнение операций, реализуемых механизмом JDBC (операций класса `DBClass`), документируется с помощью диаграмм взаимодействия. Одна из таких диаграмм — кооперативная диаграмма, показывающая выполнение операции создания новых данных (`create`), приведена на рис. 4.19.

Из диаграммы на рис. 4.19 видно, что для создания новых данных (нового класса) объект PersistencyClient запрашивает DBClass. DBClass создает новый объект PersistentClass и загружает в него данные. Затем DBClass создает новый оператор SQL, используя операцию createStatement() класса Connection. В результате выполнения этого оператора данные помещаются в БД.

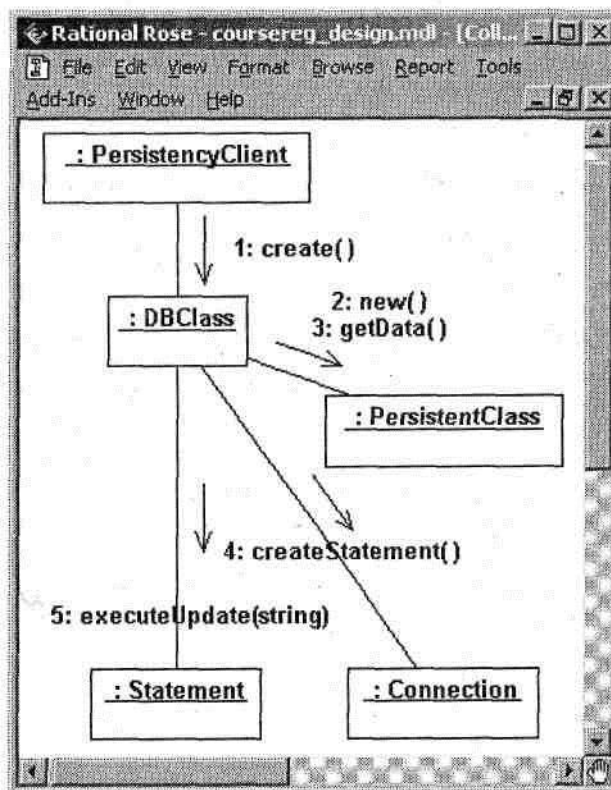


Рис. 4.19. Диаграмма выполнения операции create

Выявление подсистем и интерфейсов

Декомпозиция системы на подсистемы реализует принцип модульности (см. подразд. 2.4.1). Целями такой декомпозиции являются:

- повышение уровня абстракции системы;
- декомпозиция системы на части, которые могут независимо: конфигурироваться или поставляться;
разрабатываться (при условии стабильности интерфейсов);
размещаться в распределенной среде;
изменяться без воздействия на остальные части системы;
- разделение системы на части из соображений ограничения доступа к ос-

новным ресурсам;

- представление существующих продуктов или внешних систем (компонентов), которые не подлежат изменениям-

Первым действием архитектора при выявлении подсистем является преобразование классов анализа в проектные классы (design classes). По каждому классу анализа принимается одно из двух решений:

- класс анализа отображается в проектный класс, если он простой или представляет единственную логическую абстракцию;
- сложный класс анализа может быть разбит на несколько классов, преобразован в пакет или в подсистему.

Объединение классов в подсистемы осуществляется, исходя из следующих соображений:

- функциональная связь: объединяются классы, участвующие в реализации варианта использования и взаимодействующие только друг с другом;
- обязательность: совокупность классов, реализующая функциональность, которая может быть удалена из системы или заменена на альтернативную;
- связанность: объединение в подсистемы сильно связанных классов;
- распределение: объединение классов, размещенных на конкретном узле сети.

Примеры возможных подсистем:

- совокупность классов, обеспечивающих сложный комплекс функций (например, обеспечение безопасности и защиты данных);
- граничные классы, реализующие сложный пользовательский интерфейс или интерфейс с внешними системами;
- различные продукты: коммуникационное ПО, доступ к базам данных, общие утилиты (библиотеки), различные прикладные пакеты.

При создании подсистем в модели выполняются следующие преобразования:

- объединяемые классы помещаются в специальный пакет с

- именем подсистемы и стереотипом <<subsystem>>;
- спецификации операций классов, образующих подсистему, выносятся в интерфейс подсистемы — класс со стереотипом <<Interface>>;
- описание интерфейса должно включать:
 - имя интерфейса, отражающее его роль в системе;
 - текстовое описание интерфейса размером в небольшой абзац, отражающее его обязанности;
 - описание операций интерфейса (имя, отражающее результат операции, алгоритм выполнения операции, возвращаемое значение, параметры с их типами);
 - характер использования операций интерфейса и порядок их выполнения документируется с помощью диаграмм взаимодействия, описывающих взаимодействие классов подсистемы при реализации операций интерфейса, которые вместе с диаграммой классов подсистемы объединяются в кооперацию с именем интерфейса и стереотипом << Interface realization>>;
- в подсистеме создается класс-посредник со стереотипом <<subsystem proxy>>, управляющий реализацией операций интерфейса.

Все интерфейсы подсистем должны быть полностью определены в процессе проектирования архитектуры, поскольку они будут служить в качестве точек синхронизации при параллельной разработке системы.

В качестве примера (для системы регистрации) приведем подсистему CourseCatalogSystem, которая создана вместо граничного класса CourseCatalogSystem. Диаграммы классов и взаимодействия, описывающие данную подсистему и ее интерфейс, приведены на рис. 4.20-4.22. Классы DBCourseOffering и CourseOfferingList отвечают за взаимодействие с БД каталога курсов в рамках JDBC. Объект CourseCatalogSystem Client на рис. 4.22 может принадлежать либо классу CloseRegistrationController, либо классу RegistrationController, в зависимости от того, в каком из вариантов использования запрашивается каталог курсов.

Формирование архитектурных уровней

В процессе анализа было принято предварительное решение о выделении архитектурных уровней (в случае системы регистрации — четырех уровней в соответствии с образцом «Уровни»). В процессе проектирования все элементы системы должны быть распределены по данным уровням.

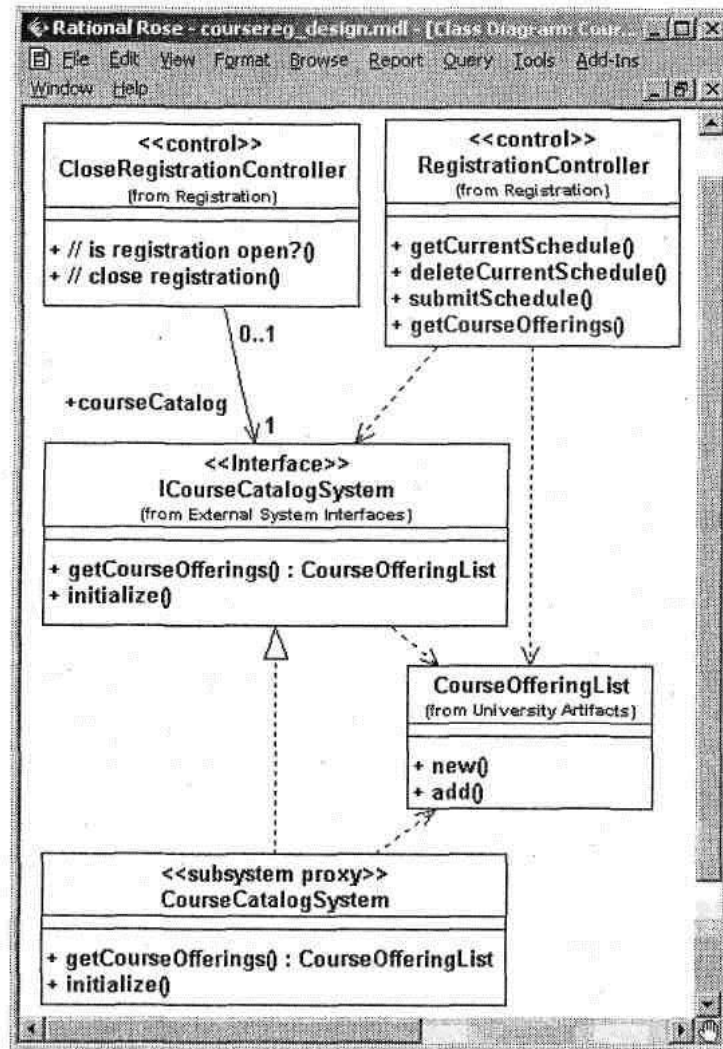


Рис. 4.20. Контекст подсистемы CourseCatalogSystem с точки зрения архитектора

С точки зрения модели это означает распределение проектных классов по пакетам, соответствующим архитектурным уровням (пакетам со стереотипом `<<layer>>`). В сложной системе архитектурные уровни могут разбиваться на подуровни, количество и структура которых, как было сказано выше, зависят от сложности предметной области и среды реализации.

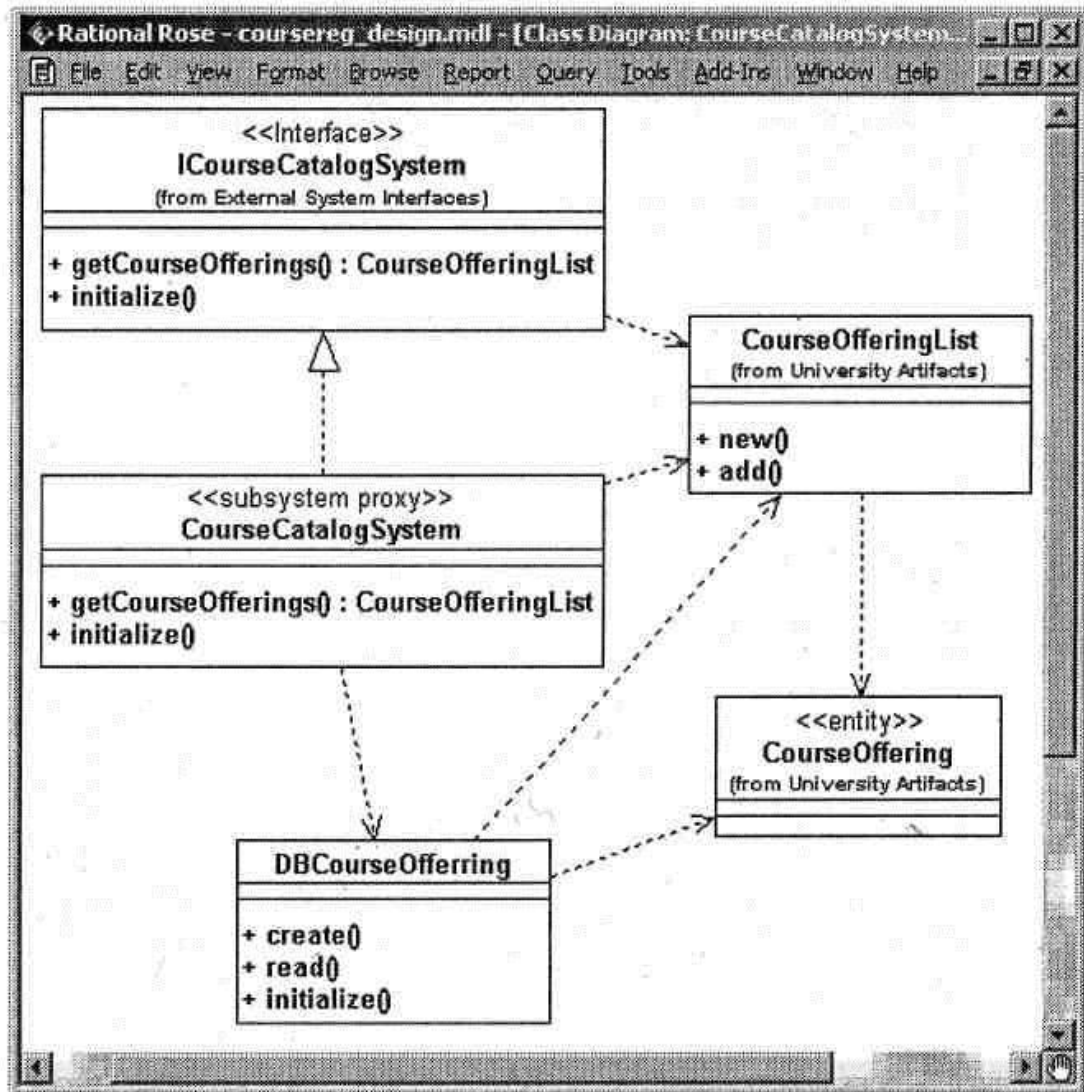


Рис. 4.21. Диаграмма классов подсистемы CourseCatalogSystem с точки зрения проектировщика

Подуровни могут формироваться, исходя из следующих критериев:

- группировка элементов с максимальной связанностью;
- распределение в соответствии со структурой организации (обычно это касается верхних уровней архитектуры);
- распределение в соответствии с различными областями компетенции разработчиков (предметная область, сети, коммуникации, базы данных, безопасность и др.);
- группировка отдельных компонентов распределенной системы;
- распределение в соответствии с различной степенью безопасности и секретности.

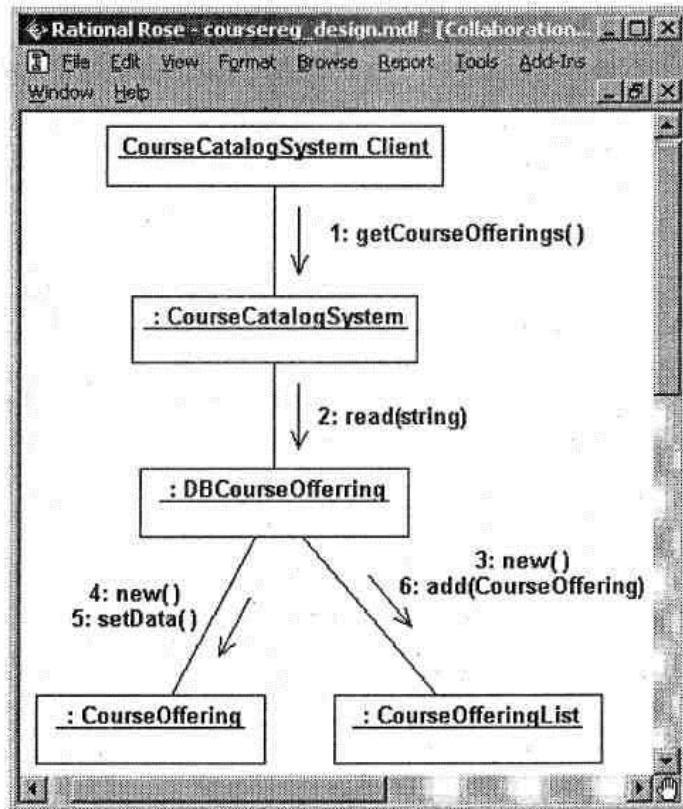


Рис. 4.22. Кооперативная диаграмма, описывающая реализацию операции интерфейса `getCourseOfferings`

Пример выделения архитектурных уровней и объединения элементов модели в пакеты для системы регистрации приведен на рис. 4.23.

Данное представление отражает следующие решения, принятые архитектором:

- выделены три архитектурных уровня (созданы три пакета со стереотипом «layer»);
- в пакете `Application` создан пакет `Registration`, куда включены граничные и управляющие классы;
- граничные классы `BillingSystem` и `CourseCatalogSystem` преобразованы в подсистемы;
- в пакет `Business Services`, помимо подсистем, включены еще два пакета: пакет `External System Interfaces` включает интерфейсы подсистем (классы со стереотипом «Interface»), а пакет `University Artifacts` — все классы-сущности.

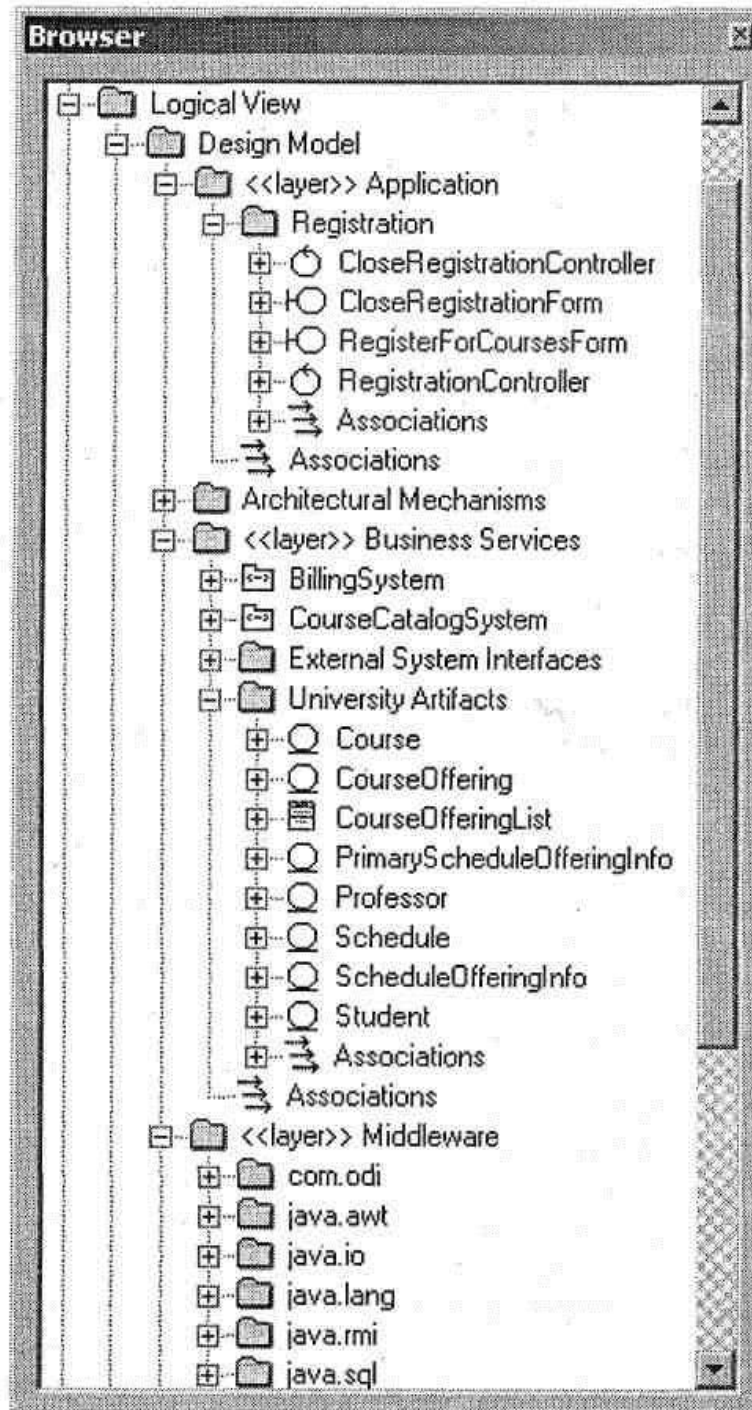


Рис. 4.23. Представление структуры модели в процессе проектирования

Проектирование структуры потоков управления

Проектирование структуры потоков управления выполняется при наличии в системе параллельных процессов (параллелизма). Цель проектирования — выявление существующих в системе процессов, характера их взаимодействия,

создания, уничтожения и отображения в среду реализации. Требование параллелизма возникает в следующих случаях:

- необходимо распределение обработки между различными процессорами или узлами;
- система управляется потоком событий (*event-driven system*);
- вычисления в системе обладают высокой интенсивностью;
- в системе одновременно работает много пользователей.

Например, система регистрации курсов обладает свойством параллелизма, поскольку она должна допускать одновременную работу многих пользователей (студентов и профессоров), каждый из которых порождает в системе отдельный процесс.

Понятие *процесс (process)* трактуется следующим образом:

- это ресурсоемкий поток управления, который может выполняться параллельно с другими процессами;
- он выполняется в независимом адресном пространстве и в случае высокой сложности может разделяться на два потока или более;
- объект любого класса должен существовать внутри хотя бы одного процесса.

Поток (thread) — это облегченный поток управления, который может выполняться параллельно с другими потоками в рамках одного и того же процесса в общем адресном пространстве. Необходимость создания потоков в системе регистрации курсов диктуется следующими требованиями:

- если курс окажется заполненным в то время, когда студент формирует свой учебный график, включающий данный курс, то он должен быть извещен об этом (необходим независимый процесс, управляющий доступом к информации конкретных курсов);
- существующая база данных каталога курсов не обеспечивает требуемую производительность (необходим процесс промежуточной обработки — подкачки данных).

Реализация процессов и потоков обеспечивается средствами операционной

системы.

Для моделирования структуры потоков управления используются так называемые активные классы¹ — классы со стереотипами <<process>> и <<thread>>. Активный класс владеет собственным процессом или потоком и может инициировать управляющие воздействия. Связи между процессами моделируются как зависимости. Потоки могут существовать только внутри процессов, поэтому связи между процессами и потоками моделируются как композиции. Модель потоков управления помещается в пакет Process View В качестве примера на рис. 4.24 — 4.26 приведены фрагменты диаграммы классов, описывающей структуру процесса регистрации студента на курсы. Активные классы, показанные на этих диаграммах, выполняют следующее назначение:

- StudentApplication — процесс, управляющий всеми функциями студента-пользователя в системе. Для каждого студента, начинающего регистрироваться на курсы, создается один объект данного класса.
- CourseRegistrationProcess — процесс, управляющий непосредственно регистрацией студента. Для каждого студента, начинающего регистрироваться на курсы, также создается один объект данного класса.
- CourseCatalogSystemAccess — управляет доступом к системе каталога курсов. Один и тот же объект данного класса используется всеми пользователями при доступе к каталогу курсов.
- CourseCache и OfferingCache используются для асинхронного доступа к данным в БД с целью повышения производительности системы. Они представляют собой кэш для промежуточного хранения данных о курсах, извлеченных из БД.

¹ Буч Г. и др. Язык UML. Руководство пользователя.: Пер. с англ. / Г. Буч, Дж. Рамбо, А. Джекобсон. - М.: ДМК, 2000.

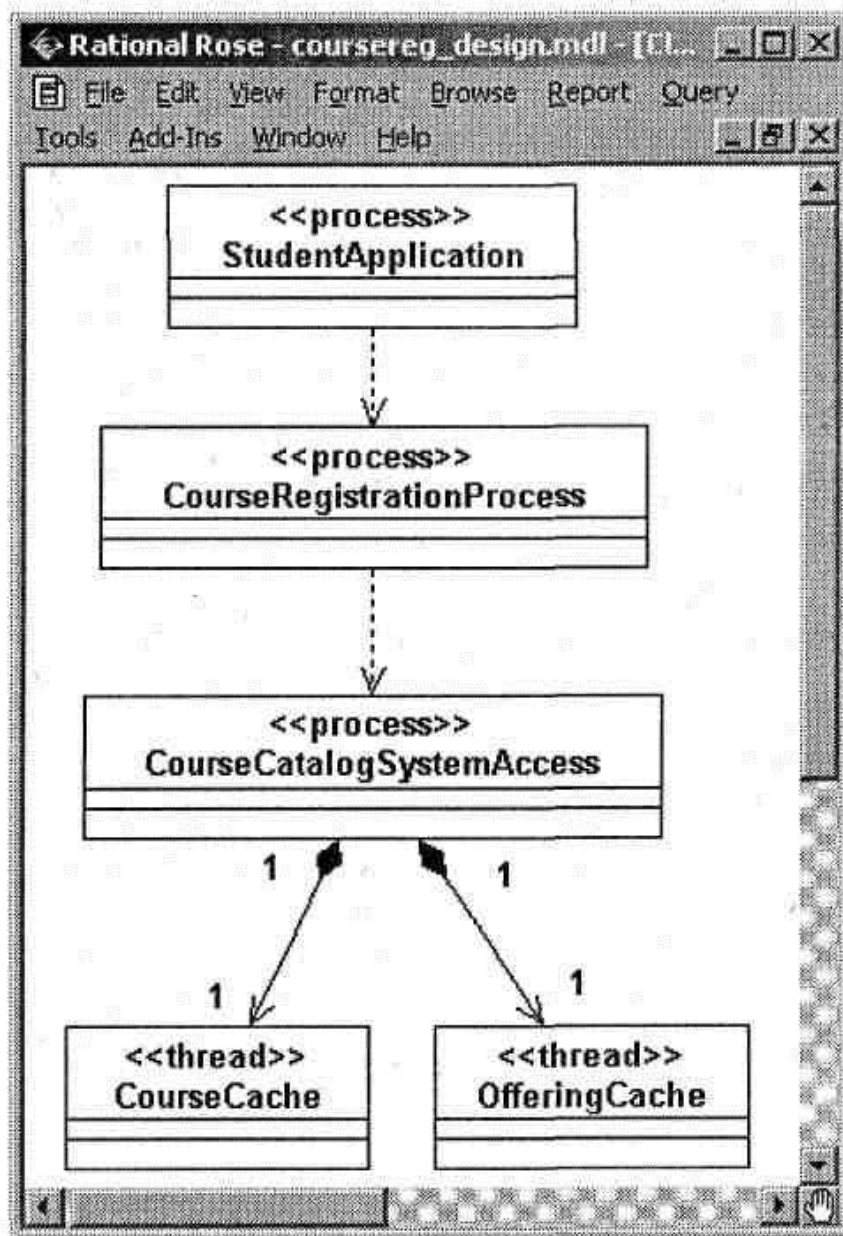


Рис. 4.24. Процессы и потоки

Проектирование конфигурации системы

Если создаваемая система является распределенной, то необходимо спроектировать ее конфигурацию в вычислительной среде, т.е. описать вычислительные ресурсы, коммуникации между ними и использование ресурсов различными системными процессами.

Распределенная сетевая конфигурация системы моделируется с помощью диаграммы размещения. Пример такой диаграммы для системы регистрации (без процессов) приведен на рис. 4.27.

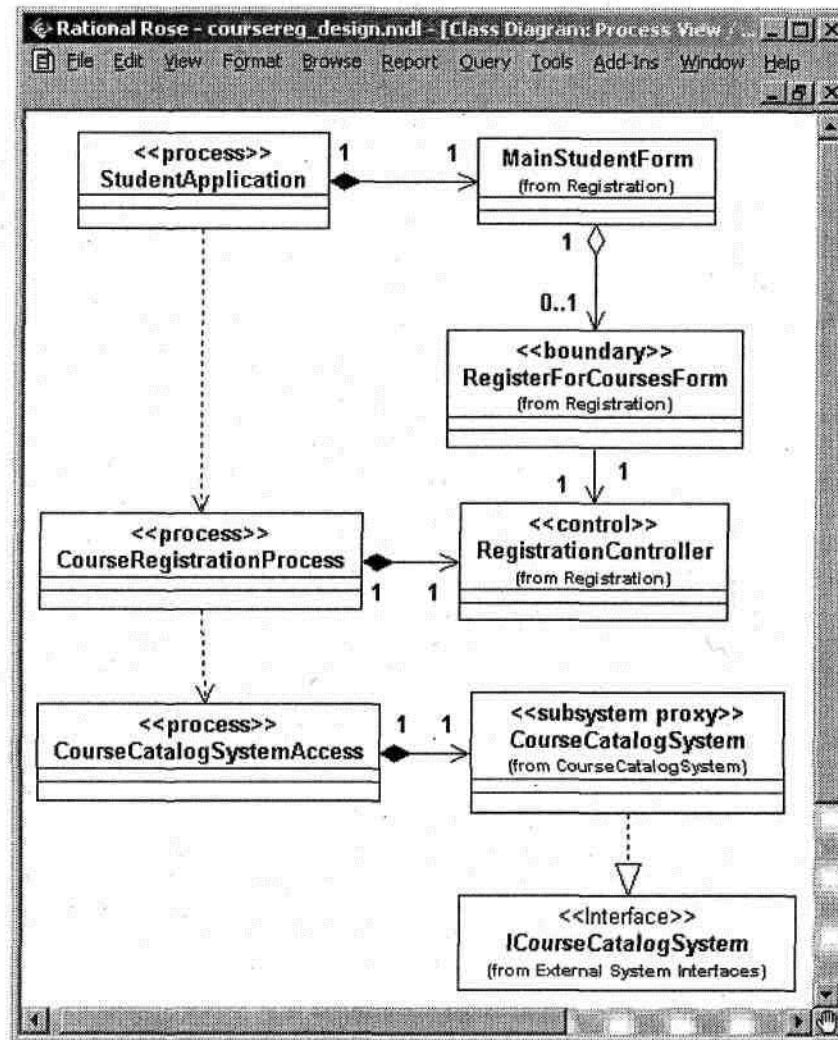


Рис. 4.25. Классы, связанные с процессами

Распределение процессов, составляющих структуру потоков управления, по узлам сети производится с учетом следующих факторов:

- используемые образцы распределения (трехзвенная клиент-серверная конфигурация, «толстый» клиент, «тонкий» клиент, равноправные узлы (peer-to-peer) и т.д.);
- время отклика;
- минимизация сетевого трафика;
- мощность узла;
- надежность оборудования и коммуникаций.

Пример распределения процессов по узлам системы регистрации приведен на рис. 4.28

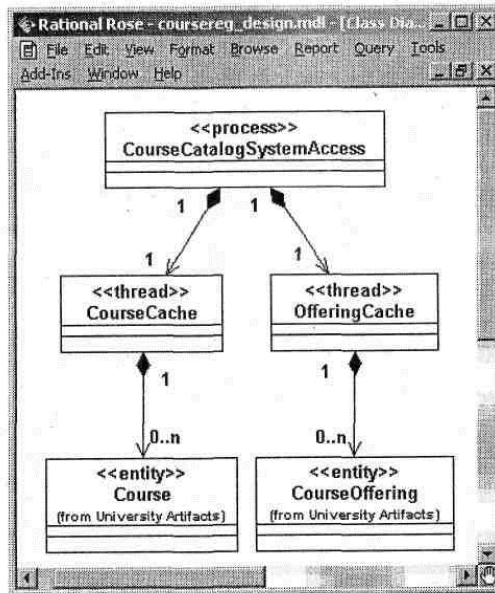


Рис. 4.26. Классы, связанные с потоками

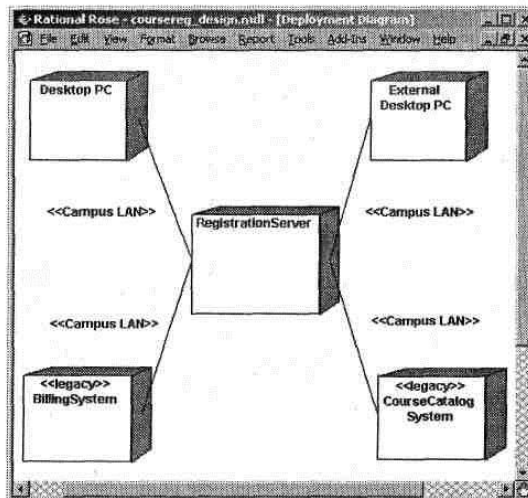


Рис. 4.27. Сетевая конфигурация системы регистрации

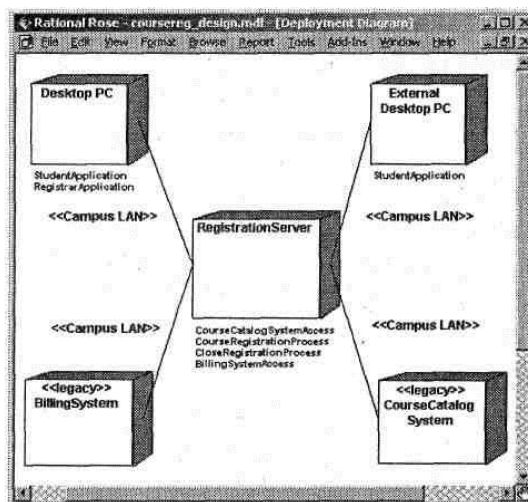


Рис. 4.28. Конфигурация системы регистрации с распределением процессов по узлам

ПРОЕКТИРОВАНИЕ ЭЛЕМЕНТОВ СИСТЕМЫ

Проектирование элементов системы выполняется проектировщиками и включает в себя:

- уточнение описания вариантов использования;
- проектирование классов;
- проектирование баз данных.

Уточнение описания вариантов использования

Уточнение описания вариантов использования заключается в модификации их диаграмм взаимодействия и диаграмм классов с учетом вновь появившихся на шаге проектирования классов и подсистем, а также проектных механизмов. Так, например, диаграммы взаимодействия варианта использования «Зарегистрироваться на курсы», изображенные на рис. 4.7 и 4.8, примут вид, показанный на рис. 4.29 и 4.30.

Объект класса CourseRegBDBManager на рис. 4.29 играет роль интерфейса с объектной базой данных системы регистрации (предполагается, что данные о студентах, графиках и профессорах будут храниться в объектной БД).

Проектирование классов

Проектирование классов включает следующие действия:

- детализация проектных классов;
- уточнение операций и атрибутов;
- моделирование состояний для классов;
- уточнение связей между классами.

Детализация проектных классов. Каждый граничный класс преобразуется в некоторый набор классов, в зависимости от своего назначения. Это может быть набор элементов пользовательского интерфейса, зависящий от возможностей среды разработки, или набор классов, реализующий системный или аппаратный интерфейс.

Классы-сущности с учетом соображений производительности и защиты данных могут разбиваться на ряд классов. Основанием для разбиения является наличие в классе атрибутов с различной частотой использования или видимостью. Такие атрибуты, как правило, выделяются в отдельные классы.

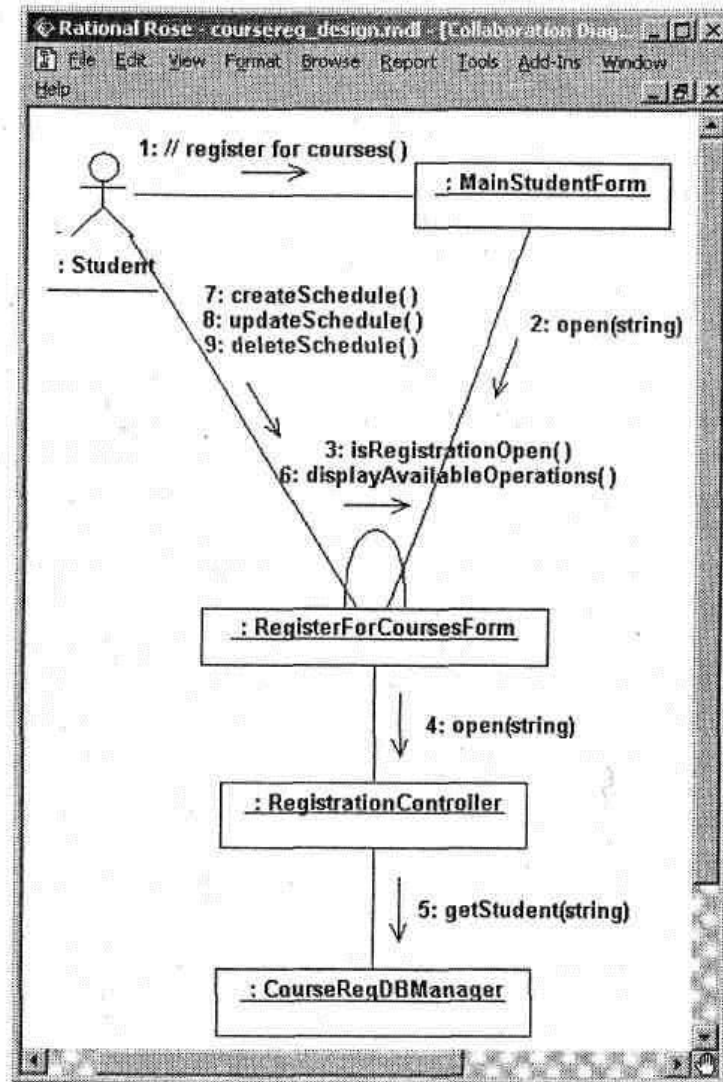


Рис. 4.29. Кооперативная диаграмма «Зарегистрироваться на курсы» — основной поток событий

Классы, реализующие простую передачу информации от граничных классов к сущностям, могут быть удалены. Сохраняются классы, выполняющие существенную работу по управлению потоками событий (управление транзакциями, распределенная обработка и т.д.).

Полученные в результате уточнения классы подлежат непосредственной реализации в коде системы.

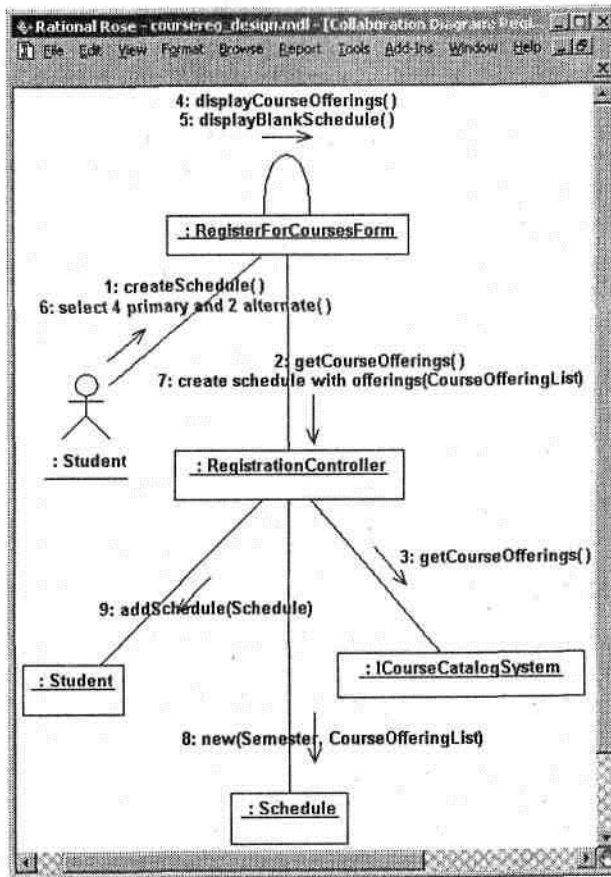


Рис. 4.30. Кооперативная диаграмма «Зарегистрироваться на курсы» - подчиненный поток «Создать график»

Уточнение операций и атрибутов. Обязанности классов, определенные в процессе анализа и документированные в виде операций «анализа», преобразуются в операции, которые будут реализованы в коде. При этом:

- каждой операции присваивается краткое имя, характеризующее ее результат;
- определяется полная сигнатура операции (в соответствии с нотацией, принятой в языке UML);
- создается краткое описание операции, включая смысл всех ее параметров;
- определяется видимость операции: public, private или protected;
- определяется область действия операции: экземпляр (операция объекта) или классификатор (операция класса);
- может быть составлено описание алгоритма выполнения операции (с использованием диаграмм деятельности в виде блок-схем, а также диаграмм

взаимодействия различных объектов при выполнении операции).

- Уточнение атрибутов классов заключается в следующем:
- кроме имени атрибута, задается его тип и значение по умолчанию (необязательное);
- учитываются соглашения по именованию атрибутов, принятые в проекте и языке реализации;
- задается видимость атрибутов: `public`, `private` или `protected`;
- при необходимости определяются производные (вычисляемые) атрибуты.

Пример уточнения операций и атрибутов приведен на рис. 4.31.

Моделирование состояний для классов. Если некоторый объект всегда одинаково реагирует на событие, то он считается *независимым от состояния* по отношению к этому событию. В отличие от него *зависимые от состояния* объекты по-разному реагируют на одно и то же событие в зависимости от своего состояния. Обычно в экономических ИС содержится очень мало объектов, зависимых от состояния, а системы управления технологическими процессами (системы реального времени) зачастую содержат множество таких объектов.

Если в системе присутствуют зависимые от состояния объекты со сложной динамикой поведения, то для них можно построить модель, описывающую состояния объектов и переходы между ними. Эта модель представляется в виде диаграмм состояний.

В качестве примера, связанного с системой регистрации, рассмотрим поведение объекта класса `CourseOffering`. Диаграмма состояний строится в несколько этапов:

1. *Идентификация состояний.* Признаками для выявления состояний являются изменение значений атрибутов объекта или создание и уничтожение связей с другими объектами. Так, объект `CourseOffering` может находиться в состоянии `Open` (прием на курс открыт) до тех пор, пока количество зарегистрировавшихся на него студентов не превышает 10, а как только оно станет равным 10, объект переходит в состояние `Closed` (прием на курс закрыт). Кроме того, объект `CourseOffering` может находиться в состоянии `Unassigned` (его никто не ведет, т.е. отсутствует связь с

каким-либо объектом Professor) или Assigned (такая связь существует).

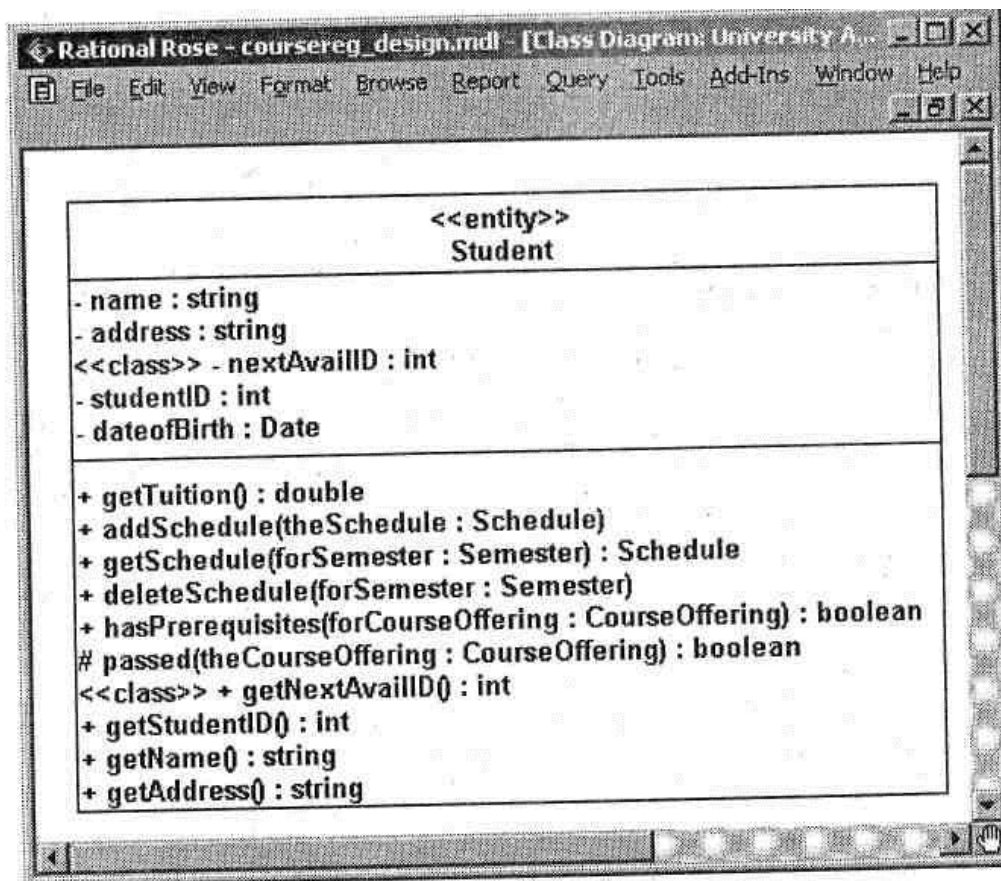


Рис. 4.31. Класс Student с полностью определенными операциями и атрибутами

2. *Идентификация событий.* События связаны, как правило, с выполнением некоторых операций. Так, в классе CourseOffering в результате распределения обязанностей при анализе варианта использования «Выбрать курсы для преподавания» определены две операции — addProfessor и removeProfessor, связанные с выбором курса некоторым профессором (созданием новой связи) и отказом от выбранного курса (разрывом связи). Этим операциям ставятся в соответствие два события - addProfessor и removeProfessor.

3. *Идентификация переходов между состояниями.* Переходы вызываются событиями. Таким образом, состояния Unassigned и Assigned соединяются двумя переходами (рис. 4.32).

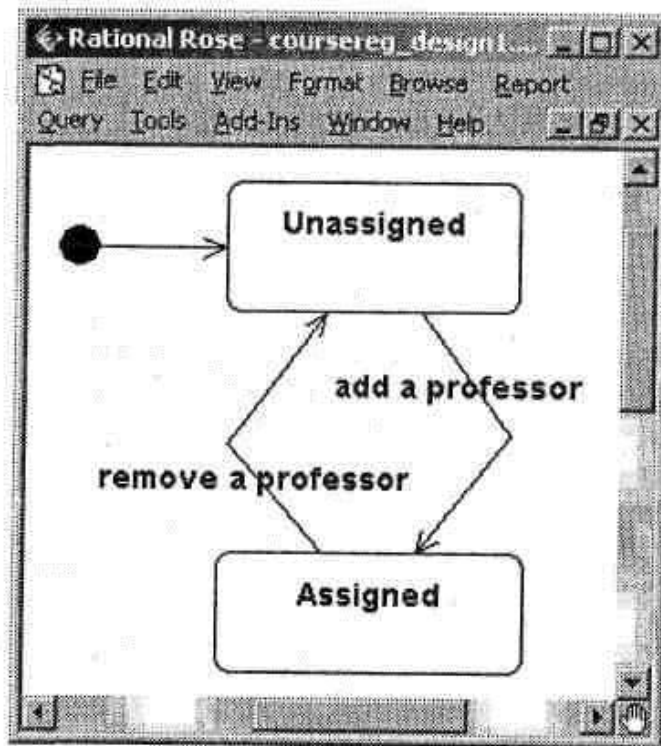


Рис. 4.32. Переходы между состояниями

Дальнейшая детализация поведения объекта CourseOffering приведет к построению диаграммы состояний, показанной на рис. 4.33. На данной диаграмме использованы такие возможности моделирования состояний, как композитные состояния (composite state) и историческое состояние (history state). В данном случае композитными состояниями являются Open и Closed, а вложенными состояниями — Unassigned, Assigned, Cancelled (курс отменен), Full (курс заполнен) и Committed (курс включен в расписание). Композитные состояния позволяют упростить диаграмму уменьшая количество переходов, поскольку вложенные состояния наследуют все свойства и переходы композитного состояния.

Историческое состояние (обозначенное на диаграмме окружностью с буквой «Н») — это псевдосостояние, которое восстанавливает предыдущее активное состояние в композитном состоянии. Оно позволяет композитному состоянию Open запоминать, какое из вложенных состояний (Unassigned или Assigned) было текущим в момент выхода из Open, для того, чтобы любой из переходов в Open (add student или remove student) возвращался именно в это вложенное состояние, а не в начальное состояние.

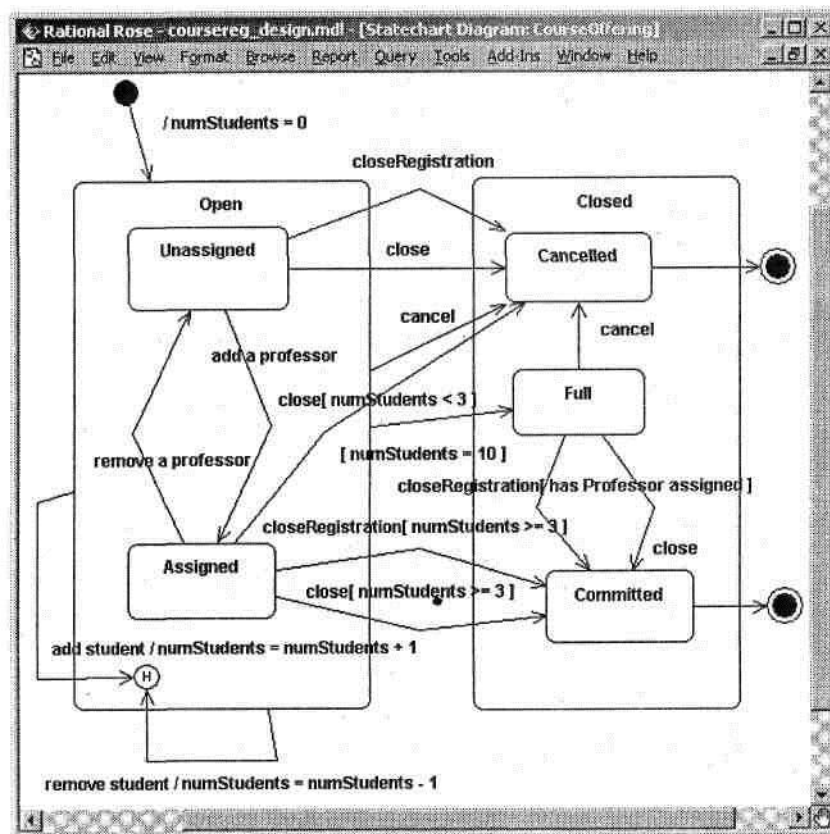


Рис. 4.33. Диаграммы состояний с композитными состояниями

Построение диаграмм состояний может оказать следующее воздействие на описание классов:

- события могут отображаться в операции класса (например, события, связанные с изменением количества студентов, записавшихся на курс, могут быть отображены в операции `addstudent` и `removestudent` класса `CourseOffering`);
- особенности конкретных состояний могут повлиять на детали выполнения операций;
- описание состояний и переходов может помочь при определении атрибутов класса (например, значение количества студентов, записавшихся на курс (`numStudents`), может быть добавлено в класс `CourseOffering` в качестве производного атрибута).

Уточнение связей между классами. В процессе проектирования связи между классами (ассоциации, агрегации и обобщения) подлежат уточнению.

- Ассоциации между граничными и управляющими классами отражают

связи, динамически возникающие между соответствующими объектами в потоке управления. Для таких связей достаточно обеспечить видимость классов, поэтому они преобразуются в зависимости.

- Если для некоторых ассоциаций нет необходимости в двунаправленной связи, то вводятся направления навигации.
- Агрегации, обладающие свойствами композиции (см. подразд. 2.4.2), преобразуются в связи композиции.

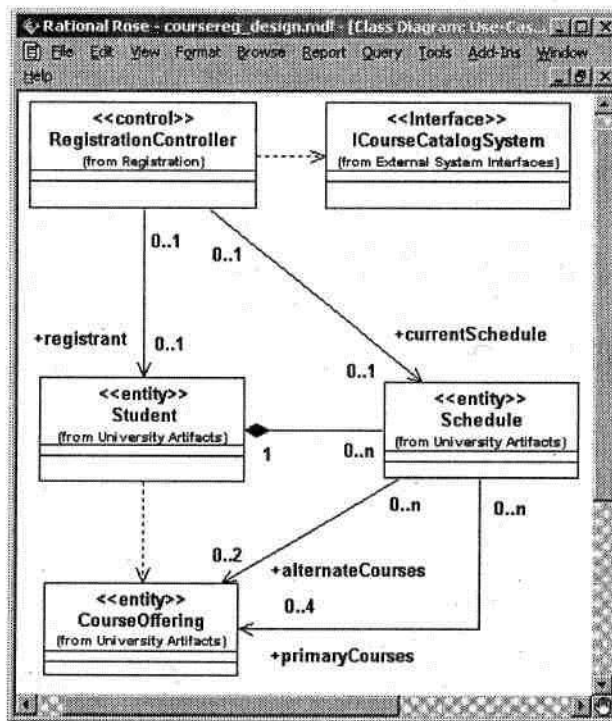


Рис. 4.34. Пример преобразования ассоциаций и агрегаций

Пример преобразования связей в соответствии с данными рекомендациями для классов варианта использования «Зарегистрироваться на курсы», приведен на рис. 4.34. Ассоциация между управляющим и граничным классами преобразована в зависимость. Агрегация между классами Student и Schedule обладает свойствами композиции. Направления навигации на ассоциациях между классами Schedule и CourseOffering введены по следующим соображениям: нет необходимости в получении списка графиков, в которых присутствует какой-либо курс, и количество графиков относительно мало по сравнению с количеством конкретных курсов.

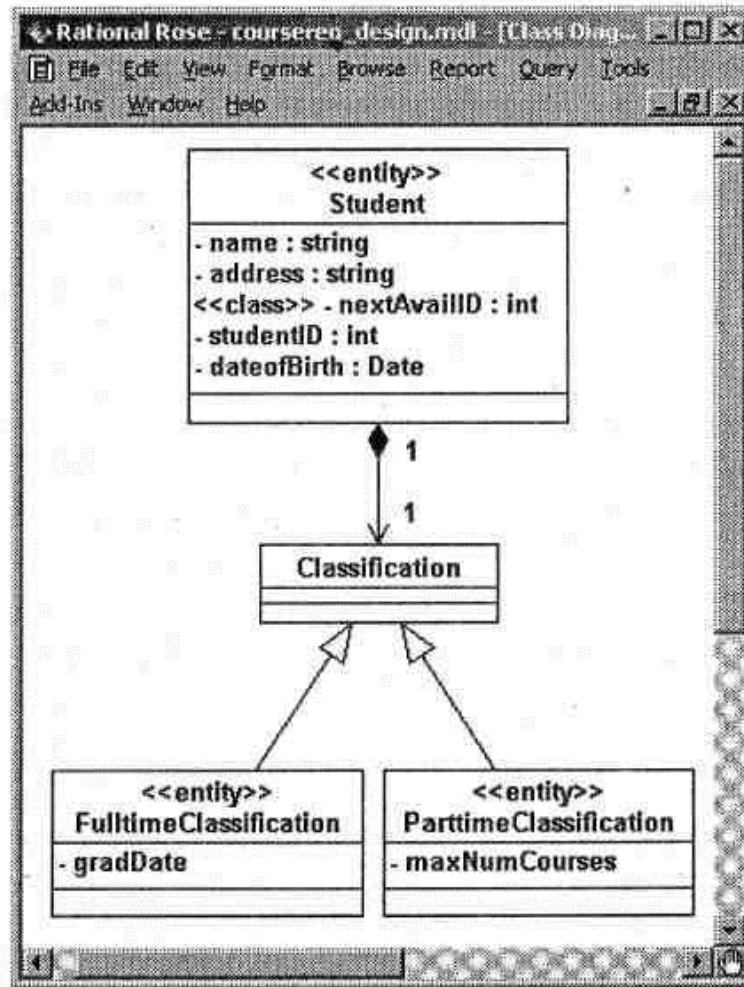


Рис. 4.35. Преобразование обобщения

Связи обобщения могут преобразовываться в ситуациях с так называемой метаморфозой подтипов. Например, в случае с системой регистрации студент может переходить с очной формы обучения на вечернюю, т.е. объект Student может менять свой подтип. При таком изменении придется модифицировать описание объекта в системе. Чтобы избежать этой модификации и тем самым повысить устойчивость системы, иерархия наследования реализуется с помощью классификации, как показано на рис. 4.35.

Проектирование баз данных

Проектирование БД зависит от типа используемой для хранения данных СУБД - объектной или реляционной. Для объектных БД никакого проектирования не требуется, поскольку классы-сущности непосредственно отображаются

в БД. Для реляционных БД классы-сущности объектной модели должны быть отображены в таблицы реляционной БД. Совокупность таблиц и связей между ними может быть представлена, в виде диаграммы классов, которая по существу является ER-диаграммой. Набор правил, применяемых при отображении классов в таблицы БД, фактически совпадает с правилами преобразования сущностей и связей, описанными в подразд. 4.1. В технологии RUP, в частности, для такого отображения используется специальный инструмент — Data Modeler. Он выполняет преобразование классов-сущностей в классы-таблицы с последующей генерацией описания БД на SQL.

Для описания схемы БД применяется следующий набор элементов языка UML со своими стереотипами (профиль UML):

- таблица представляется в виде класса со стереотипом «Table»;
- представление изображается в виде класса со стереотипом «View»;
- столбец таблицы представляется в виде атрибута класса с соответствующим типом данных;
- обычная ассоциация и агрегация представляются в виде ассоциации со стереотипом «Non-Identifying» (в терминологии IDEF1X — неидентифицирующей связи);
- композиция представляется в виде ассоциации со стереотипом «Identifying» (в терминологии IDEF1X — идентифицирующей связи);
- схема БД представляется в виде пакета со стереотипом «Schema», содержащего классы-таблицы;
- контейнер хранимых процедур представляется в виде класса со стереотипом «SP Container»;
- ограничения целостности, индексы и триггеры представляются в виде операций классов-таблиц со стереотипами «PK» (Primary key), «FK» (Foreign key), «Unique», «Check», «Index» и «Trigger»;
- физическая база данных представляется в виде компонента со стереотипом «Database».

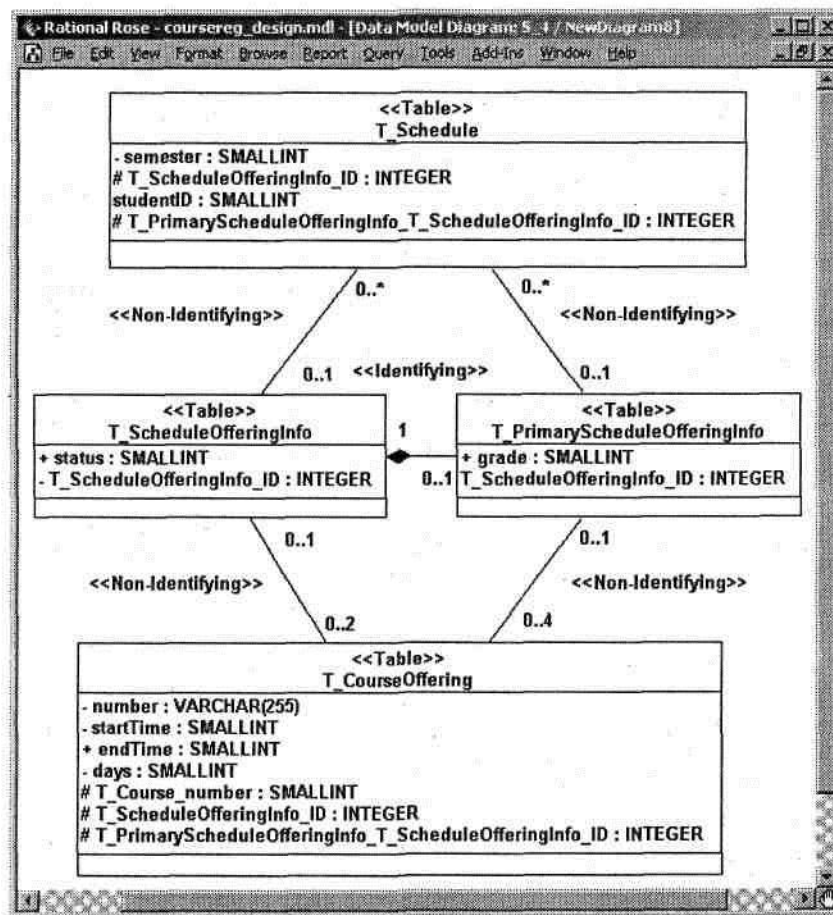


Рис. 4.36. Пример неидентифицирующих связей

Примеры преобразования, выполненного для классов варианта использования «Зарегистрироваться на курсы» средствами Data Modeler, приведены на рис. 4.36 - 4.38.

На рис. 4.36 ассоциации-классы, представляющие свойства связей между классами Schedule и CourseOffering (см. рис. 4.16), преобразованы в таблицы в соответствии с правилом преобразования бинарных связей типа «многие-ко-многим».

Идентифицирующая связь между таблицами T_Student и T_Schedule на рис. 4.37 соответствует связи композиции между классами Student и Schedule на рис. 4.34.

На рис. 4.38 показано преобразование связи обобщения, изображенной на рис. 4.35. Здесь используется отдельная таблица для каждого подтипа (в соответствии с правилом из подразд. 4.1). Достоинства и недостатки различных

способов преобразования обсуждались в подразд. 4.1). Преимуществом способа, применяемого в данном случае, является простота алгоритма автоматического преобразования, выполняемого средствами Data Modeler.

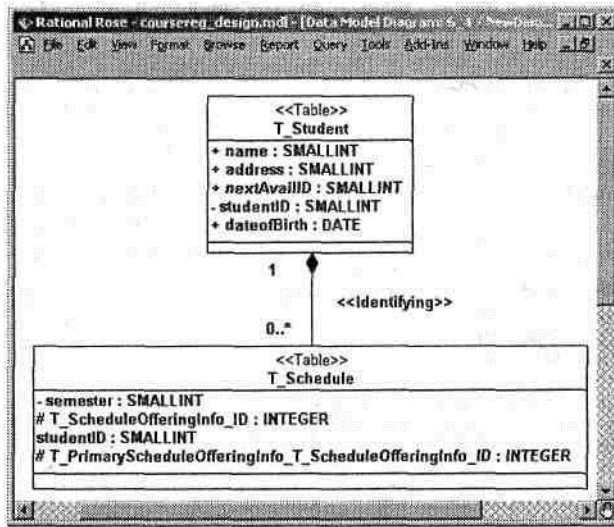


Рис. 4.37. Пример идентифицирующей связи

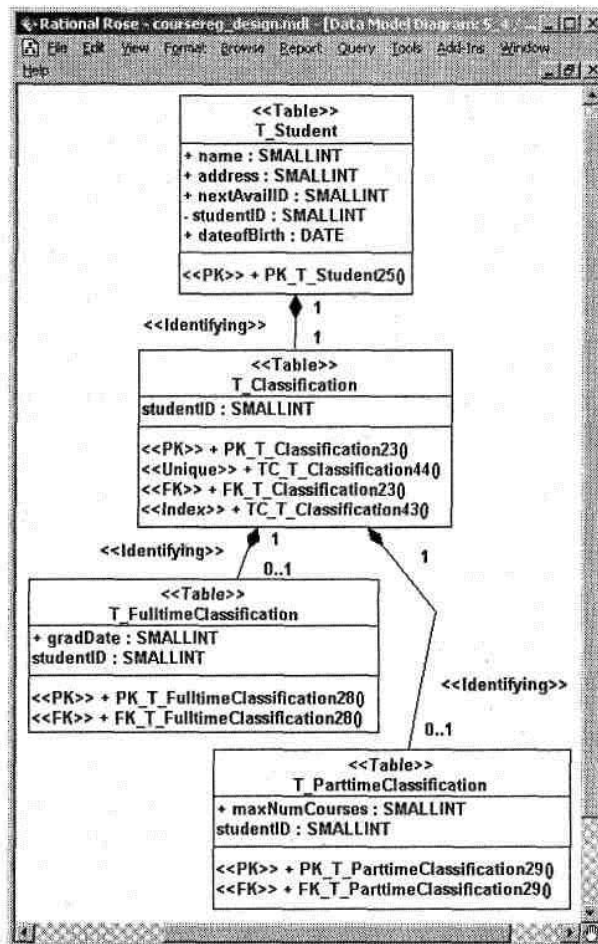


Рис. 4.38. Пример преобразования обобщения

! Следует запомнить

Целью объектно-ориентированного анализа является трансформация функциональных требований к ПО в предварительный системный проект и создание стабильной основы архитектуры системы. В процессе проектирования системный проект «погружается» в среду реализации с учетом всех нефункциональных требований.

Основные понятия

Архитектурные механизмы, архитектурные уровни, классы анализа, проектные классы, процесс, поток.

? Вопросы для самоконтроля

1. Перечислите правила формирования схемы базы данных из ERM.
2. К каким последствиям для системы может привести отсутствие архитектурного анализа?
3. Что дает использование образцов распределения обязанностей?
4. Какие классы нуждаются в моделировании состояний?
5. Охарактеризуйте достоинства и область применения методики анализа и проектирования Rational Unified Process.

ТЕХНОЛОГИИ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой технология создания ПО.*
- *Какие требования предъявляются к технологии создания ПО.*
- *В чем заключается процесс внедрения технологии в организации.*

5.1.

ОПРЕДЕЛЕНИЕ ТЕХНОЛОГИИ

Начиная с введения, в разных контекстах упоминались различные элементы технологии создания ПО (ТС ПО) — процессы, методы, языки и др. В предыдущем издании учебника ТС ПО неформально определялась как совокупность технологических операций проектирования в их последовательности и взаимосвязи, приводящая к разработке проекта ПО. Здесь мы дадим более развернутое и строгое определение ТС ПО.

Основные определения (система понятий, описывающих ТС ПО)

Технология создания ПО — упорядоченная совокупность взаимосвязанных технологических процессов в рамках ЖЦ ПО.

Технологический процесс — совокупность взаимосвязанных технологических операций.

Технологическая операция — основная единица работы, выполняемая определенной ролью, которая:

- подразумевает четко определенную ответственность роли;
- дает четко определенный результат (набор рабочих продуктов), базирующийся на определенных исходных данных (другом наборе рабочих

- продуктов);
- представляет собой единицу работы с жестко определенными границами, которые устанавливаются при планировании проекта.

Рабочий продукт — информационная или материальная сущность, которая создается, модифицируется или используется в некоторой технологической операции (модель, документ, код, тест и т.п.). Рабочий продукт определяет область ответственности роли и является объектом управления конфигурацией.

Роль — определение поведения и обязанностей отдельного лица или группы лиц в среде организации-разработчика ПО, осуществляющих деятельность в рамках некоторого технологического процесса и ответственных за определенные рабочие продукты.

Руководство — практическое руководство по выполнению одной или совокупности технологических операций. Руководства включают методические материалы, инструкции, нормативы, стандарты и критерии оценки качества рабочих продуктов.

Инструментальное средство (CASE-средство) — программное средство, обеспечивающее автоматизированную поддержку деятельности, выполняемой в рамках технологических операций.

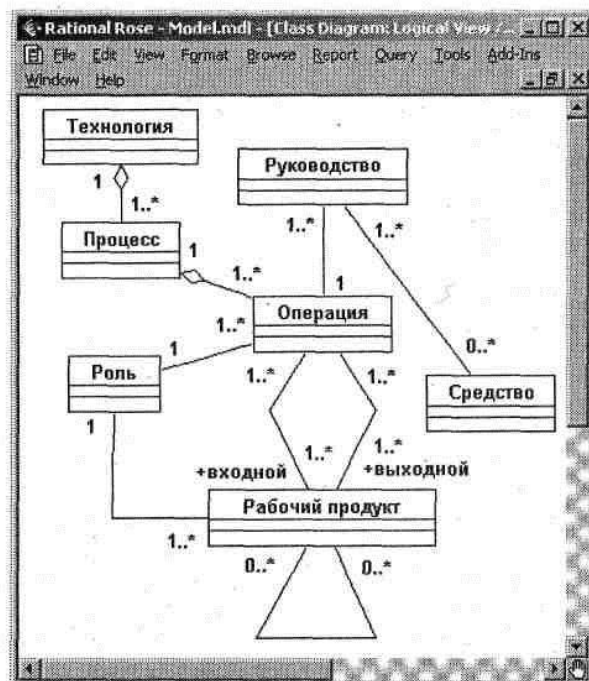


Рис. 5.1. Объектная модель ТС ПО

Данную систему понятий можно представить в виде объектной модели на языке UML в виде совокупности абстракций (классов), соответствующих приведенным выше понятиям (рис. 5.1). Каждому классу модели соответствует множество объектов (экземпляров), определяющих конкретные элементы ТС ПО: технологические процессы, технологические операции, рабочие продукты, роли, CASE-средства и руководства.

Рабочий вариант (экземпляр) конкретной ТС ПО представляет собой ТС ПО, адаптированную к условиям объекта внедрения и проектам создания ПО. Динамическая модель, описывающая поведение ТС ПО в жизненном цикле ПО, представляется в виде последовательности переходов между состояниями ТС ПО (рис. 5.2). Событием, инициирующим переход между различными состояниями ТС ПО, является изменение требований к ТС ПО, а ограничивающим условием перехода является соответствие комплексу критериев оценки и выбора ТС ПО, который будет рассмотрен далее. Каждое состояние ТС ПО определяется набором ее элементов и является композитным состоянием по отношению к состояниям отдельных элементов. Поведение каждого отдельного элемента ТС ПО (технологического процесса, технологической операции, рабочего продукта и др.) в ЖЦ ПО также представляется в виде последовательности переходов между его состояниями.

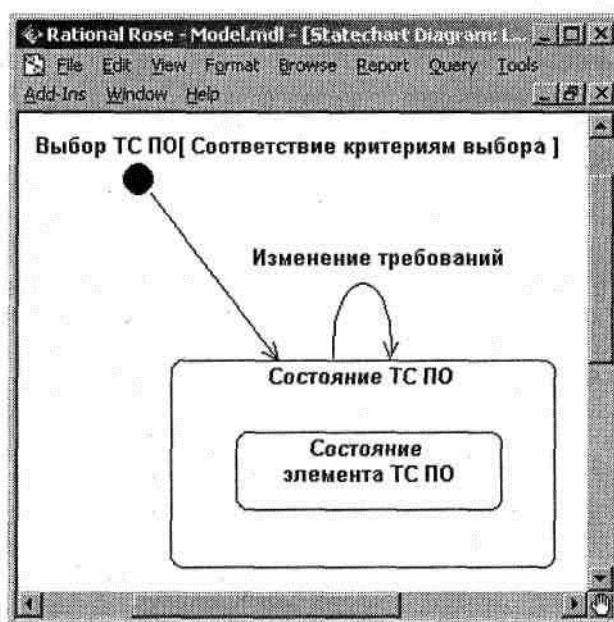


Рис. 5.2. Диаграмма состояний ТС ПО

ОБЩИЕ ТРЕБОВАНИЯ, ПРЕДЪЯВЛЯЕМЫЕ**К ТС ПО**

Основным требованием, предъявляемым к современным ТС ПО, является их соответствие стандартам и нормативным документам, связанным с процессами ЖЦ ПО и оценкой технологической зрелости организаций-разработчиков (ISO 12207, ISO 9000, CMM и др.). Согласно этим нормативам ТС ПО должна поддерживать следующие процессы:

- управление требованиями;
- анализ и проектирование ПО;
- разработка ПО;
- эксплуатация;
- сопровождение;
- документирование;
- управление конфигурацией и изменениями;
- тестирование;
- управление проектом.

Полнота поддержки процессов ЖЦ ПО должна поддерживаться комплексом инструментальных средств (CASE-средств).

Соответствие стандартам означает также, в частности, использование общепринятых, стандартных нотаций и соглашений. Для того чтобы проект мог выполняться разными коллективами разработчиков, необходимо использование стандартных методов моделирования и стандартных нотаций, которые должны быть оформлены в виде нормативов до начала процесса проектирования. Несоблюдение проектных стандартов ставит разработчиков в зависимость от фирмы — производителя данного средства, делает затруднительным формальный контроль корректности проектных решений и снижает возможности привлечения дополнительных коллективов разработчиков, смены исполнителей и отчужде-

ния проекта, поскольку число специалистов, знакомых с данным методом (нотацией), может быть ограниченным.

Другим важным требованием является *адаптируемость к условиям применения*, которая достигается за счет поставки технологии в электронном виде вместе с CASE-средствами и библиотеками процессов, шаблонов, методов, моделей и других компонентов, предназначенных для построения ПО того класса систем, на который ориентирована технология. Электронные технологии должны включать средства, обеспечивающие их адаптацию и развитие по результатам выполнения конкретных проектов. Процесс адаптации заключается в удалении ненужных процессов и действий ЖЦ ПО, в изменении неподходящих или в добавлении собственных процессов и действий, а также методик, стандартов и руководств.

5.3.

ВНЕДРЕНИЕ ТС ПО В ОРГАНИЗАЦИИ

5.3.1.

ОБЩИЕ СВЕДЕНИЯ

Термин «внедрение» используется в широком смысле и включает все действия — от оценки первоначальных потребностей до полномасштабного использования ТС ПО в различных подразделениях организации. Процесс внедрения ТС ПО состоит из следующих этапов.

1. Определение потребностей в ТС ПО, характеристики объекта внедрения и проектов создания ПО.
2. Определение требований, предъявляемых к ТС ПО (анализ характеристик объекта внедрения и проектов, обоснование требований к ТС ПО, определение приоритетов требований).
3. Оценка вариантов ТС ПО. Предварительная экспертная оценка заключа-

ется в анализе доступных ТС ПО на предмет соответствия требованиям, неудовлетворительные варианты (с точки зрения реализации наиболее приоритетных требований) отвергаются, формируется список претендентов. При детализированной оценке для каждой ТС ПО-претендента формируется ее детальное описание, основанное на общей модели (см. рис. 5.1). Источники информации для описания — техническая документация поставщика, доступные данные о реальных внедрениях, результаты выполнения пилотных проектов.

4. Выбор ТС ПО. Производится сравнительный анализ технологий и окончательный выбор ТС ПО с помощью экспертной оценки.

5. Адаптация ТС ПО к условиям применения. Производится формирование конкретной рабочей конфигурации ТС ПО, адаптированной к условиям объекта внедрения.

В процессе внедрения ТС ПО собирается статистика и оценивается эффективность ее внедрения с точки зрения ряда критериев (минимум трудоемкости сопровождения ПО, минимум затрат на сопровождение ПО и др.). При изменении условий объекта внедрения и по результатам анализа эффективности внедрения ТС ПО принимается решение: а) о внесении изменений в рабочую конфигурацию ТС ПО; б) о переходе на новую ТС ПО. В случае перехода повторяются пп. 3, 4, 5.

Процесс успешного внедрения ТС ПО не ограничивается только ее использованием. На самом деле он охватывает планирование и реализацию множества технических, организационных, структурных процессов, изменений в общей культуре организации и основан на четком понимании возможностей ТС ПО.

На способ внедрения ТС ПО может повлиять специфика конкретной ситуации. Например, если заказчик предпочитает конкретную технологию или она оговаривается требованиями контракта, этапы внедрения должны соответствовать такому predetermined выбору. В иных ситуациях относительная простота или сложность ТС ПО, степень согласованности или конфликтности с существующими в организации процессами, требуемая степень интеграции с другими технологиями, опыт и квалификация пользователей могут привести к

внесению соответствующих корректив в процесс внедрения.

Несмотря на все потенциальные возможности ТС ПО, существует множество примеров их неудачного внедрения. В связи с этим необходимо отметить следующее:

- ТС ПО не обязательно дают немедленный эффект; он может быть получен только спустя какое-то время;
- реальные затраты на внедрение ТС ПО обычно намного превышают затраты на ее приобретение;
- технология обеспечивает возможности для получения существенной выгоды только после успешного завершения процесса ее внедрения.

Ввиду разнообразной природы технологий было бы ошибочно делать безоговорочные утверждения относительно реального удовлетворения тех или иных ожиданий от их внедрения. Отметим факторы, усложняющие определение возможного эффекта от использования ТС ПО:

- широкое разнообразие качества и возможностей ТС ПО;
- относительно небольшое время использования ТС ПО в различных организациях и недостаток опыта их применения;
- разнообразие практики внедрения ТС ПО в различных организациях;
- отсутствие детальных метрик и данных для уже выполненных и текущих проектов;
- широкий диапазон предметных областей проектов;
- различная степень интеграции ТС ПО в различных проектах.

Вследствие этих сложностей доступная информация о реальных внедрениях крайне ограничена и противоречива. Она зависит от типа средств, характеристик проектов, уровня сопровождения и опыта пользователей. Некоторые аналитики полагают, что реальная выгода от использования ТС ПО может быть получена только после одно- или двухлетнего опыта. Другие считают, что воздействие может реально проявиться в процессе эксплуатации системы, когда технологические улучшения могут привести к снижению эксплуатационных затрат.

Чтобы принять взвешенное решение относительно инвестиций в ТС ПО, пользователи вынуждены производить оценку отдельных средств, опираясь на неполные и противоречивые данные. Эта проблема зачастую усугубляется недостаточным знанием всех возможных «подводных камней» использования ТС ПО. Среди наиболее важных проблем выделяются следующие:

- достоверная оценка отдачи от инвестиций в ТС ПО затруднительна ввиду отсутствия приемлемых метрик и данных по проектам и процессам разработки ПО;
- внедрение ТС ПО может представлять собой достаточно длительный процесс и может не принести немедленной отдачи. Возможно даже краткосрочное снижение продуктивности в результате усилий, затрачиваемых на внедрение. Вследствие этого руководство организации может утратить интерес к ТС ПО и прекратить поддержку ее внедрения;
- отсутствие полного соответствия между теми процессами и методами, которые поддерживаются ТС ПО, и теми, которые используются в данной организации, может привести к дополнительным трудностям;
- некоторые ТС ПО требуют слишком много усилий для того, чтобы оправдать их использование в небольшом проекте, при этом, тем не менее, можно извлечь выгоду из той дисциплины, к которой обязывает их применение;
- негативное отношение персонала к внедрению новой ТС ПО может быть главной причиной провала проекта.

Несмотря на все высказанные предостережения и некоторый пессимизм, грамотный и разумный подход к использованию ТС ПО позволяет преодолеть все перечисленные трудности. Успешное внедрение ТС ПО должно обеспечить:

- высокий уровень технологической поддержки процессов разработки и сопровождения ПО;
- положительное воздействие на производительность, качество продукции, соблюдение стандартов, документирование;
- приемлемый уровень отдачи от инвестиций в ТС ПО.

ОПРЕДЕЛЕНИЕ ПОТРЕБНОСТЕЙ В ТС ПО

Цель данного этапа (рис. 5.3) — достижение понимания потребностей организации в ТС ПО. Он должен привести к выделению тех областей деятельности организации, в которых применение ТС ПО может принести реальную пользу. Результатом данного этапа является документ, определяющий стратегию внедрения ТС ПО.

Анализ возможностей организации

Первым действием данного этапа является анализ возможностей организации в отношении ее технологической базы, персонала и используемого ПО. Такой анализ может быть формальным или неформальным.

Формальные подходы определяются моделью CMM, а также стандартами ISO 9001: 1994, ISO 9003-3: 1991 и ISO 9004-2:1991. Главное в этих подходах — анализ различных аспектов происходящих в организации процессов.

Для получения информации относительно положения и потребностей организации могут использоваться неформальные оценки и анкетирование. Список вопросов, которые могут помочь в неформальной оценке текущей практики использования ПО, ТС ПО и персонала, приведен ниже.

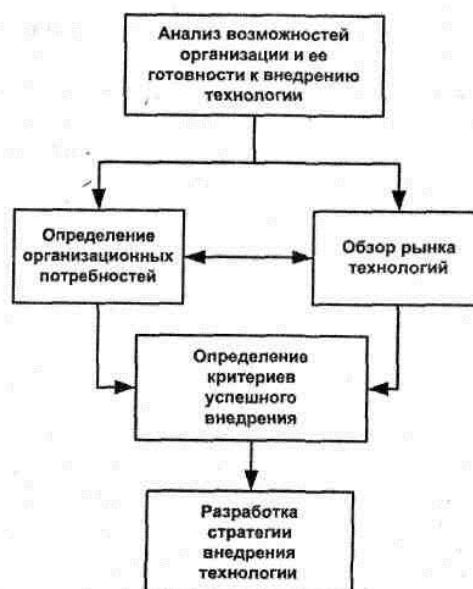


Рис. 5.3. Определение потребностей в ТС ПО

Общие вопросы (ответы на данные вопросы в целом характеризуют подход организации к разработке ПО):

- используемая модель ЖЦ ПО (каскадная или итерационная);
- используемые методы (структурные, объектно-ориентированные). Опыт, накопленный при использовании того или иного метода, полученное обучение. Степень адаптации метода к потребностям организации;
- наличие документированных стандартов (формальных или неформальных) по анализу требований, спецификациям и проектированию, кодированию и тестированию;
- количественные метрики, используемые в процессе разработки ПО, их использование;
- виды документации, выпускаемой в процессе создания ПО;
- наличие группы поддержки ТС ПО.

Вопросы, касающиеся проектов, ведущихся в организации:

- средняя продолжительность проекта в человеко-месяцах;
- среднее количество специалистов, участвующих в проектах различных категорий (небольших, средних и крупных);
- средний размер проектов различных категорий в терминах кодовых метрик (например, в функциональных точках или строках исходного кода), способ измерения.

Вопросы, касающиеся технологической базы:

- доступные вычислительные ресурсы, платформа разработки;
- уровень доступности ресурсов, «узкие места», среднее время ожидания ресурсов;
- ПО, используемое в организации, и его характер (готовые программные продукты, собственные разработки);
- степень интеграции применяемых программных продуктов, механизмы интеграции (существующие и планируемые);
- тип и уровень сетевых возможностей, доступных группе разработчиков;

- используемые языки программирования;
- средний процент вновь разрабатываемых, повторно используемых и реально эксплуатируемых приложений.

Целью оценки персонала является определение его отношения к возможным изменениям (позитивного, нейтрального или негативного).

Вопросы, касающиеся оценки персонала:

- реакция сотрудников организации (как отдельных людей, так и коллективов) на внедрение новой ТС ПО, наличие опыта успешных или безуспешных внедрений;
- наличие лидеров, способных серьезно повлиять на отношение к новым средствам;
- наличие стремления «снизу» к совершенствованию средств и ТС ПО;
- объем обучения, необходимого для ориентации пользователей в новой ТС ПО;
- стабильность и уровень текучести кадров.

Целью оценки готовности организации является определение того, насколько она способна воспринять как немедленные, так и долгосрочные последствия внедрения ТС ПО.

Вопросы, касающиеся оценки готовности:

- поддержка проекта со стороны высшего руководства;
- готовность организации к долгосрочному финансированию проекта;
- готовность организации к выделению необходимых специалистов для участия в процессе внедрения и их обучению;
- готовность персонала к изменению технологии своей работы и трудовых навыков в такой степени, в какой это потребуют новые средства;
- степень понимания персоналом масштаба изменений;
- готовность технических специалистов и менеджеров пойти на возможное кратковременное снижение продуктивности своей работы;
- готовность руководства к долговременному ожиданию отдачи от вложен-

ных средств.

Оценка готовности организации к внедрению ТС ПО должна быть объективной и тщательно выверенной, поскольку в случае отсутствия такой готовности все усилия по внедрению потерпят крах.

Определение организационных потребностей

Организационные потребности следуют непосредственно из проблем организации и целей, которые она стремится достичь. Проблемы и цели могут быть связаны с управлением, процессами, производством продукции, экономикой, персоналом или технологией. Вопросы, касающиеся определения целей, потребностей и ожидаемых результатов, приведены ниже. Определение потребностей должно выполняться в сочетании с обзором рынка технологий, поскольку информация о технологиях, доступных на рынке в данный момент, может оказать влияние на потребности.

Цели организации играют главную роль в определении ее конкретных потребностей и ожидаемых результатов. Для их понимания необходимо ответить на следующие вопросы:

- намерение организации использовать технологию для помощи в достижении определенных целей или ожиданий (например, определенного уровня СММ или сертификации в соответствии с ISO 9001);
- восприятие ТС ПО как фактора, способствующего достижению стратегических целей организации;
- наличие у организации собственной программы совершенствования процесса разработки ПО;
- восприятие инициативы внедрения ТС ПО как части более широкомасштабного проекта по созданию среды разработки ПО.

Определение потребностей организации, связанных с использованием ТС ПО, включает анализ целей и существующих возможностей. После того как все потребности организации определены, каждой из них должен быть присвоен определенный приоритет, отражающий ее значимость для успешной деятельности организации. Если потребности, связанные с технологией, не обладают

высшим приоритетом, имеет смысл отказаться от ее внедрения и сосредоточиться на потребностях с наивысшим приоритетом.

Определению потребностей организации могут помочь ответы на следующие вопросы.

- Каким образом продуктивность и качество деятельности организации сравниваются с аналогичными показателями подобных организаций (к сожалению, многие организации не располагают данными для такого сравнения)?
- Какие процессы ЖЦ ПО дают наилучшую (наихудшую) отдачу, существуют ли конкретные процессы, которые могут быть усовершенствованы путем использования новых методов и средств?

С внедрением ТС ПО обычно связывают большие ожидания. В ряде случаев эти ожидания оказываются нереалистичными и приводят к неудаче при внедрении.

Составление реалистичного перечня ожидаемых результатов является трудной задачей, поскольку он может зависеть от таких факторов, как тип внедряемых средств и характеристики внедряющей организации. Кроме того, достижение некоторых результатов может противоречить другим результатам.

Ряд потенциально реалистичных и нереалистичных ожидаемых результатов, связанных с организацией в целом, пользователями, планированием, анализом, проектированием, разработкой и затратами, приведен ниже. Практически невозможно, чтобы в процессе одного внедрения ТС ПО были достигнуты все положительные результаты. Тем не менее, любая организация может выработать собственные идеи относительно ожидаемых результатов, имея в виду, что данный перечень является всего лишь примером.

Реалистичные ожидания:

- повышение внимания к планированию деятельности, связанной с информационной технологией;
- долговременное повышение продуктивности и качества деятельности организации;

- ускорение и повышение согласованности разработки приложений;
- снижение доли ручного труда в процессе разработки и (или) эксплуатации;
- более точное соответствие приложений требованиям пользователей;
- отсутствие необходимости большой переделки приложений для повышения их эффективности;
- улучшение реакции службы эксплуатации на требования внесения изменений и усовершенствований;
- лучшее документирование;
- улучшение коммуникации между пользователями и разработчиками;
- последовательное и постоянное повышение качества проектирования;
- более высокие возможности повторного использования разработок;
- кратковременное возрастание затрат, связанное с деятельностью по внедрению ТС ПО;
- последовательное снижение общих затрат;
- лучшая прогнозируемость затрат.

Нереалистичные ожидания:

- отсутствие воздействия на общую культуру и распределение ролей в организации;
- понимание проектных спецификаций неподготовленными пользователями;
- сокращение персонала, связанного с информационной технологией;
- уменьшение степени участия в проектах высшего руководства и менеджеров, а также экспертов предметной области, уменьшение степени участия пользователей в процессе разработки приложений;
- немедленное повышение продуктивности деятельности организации;
- достижение абсолютной полноты и непротиворечивости спецификаций;
- автоматическая генерация прикладных систем из проектных спецификаций;

- немедленное снижение затрат, связанных с информационной технологией;
- снижение затрат на обучение.

Реализм в оценке ожидаемых затрат имеет особенно важное значение, поскольку он позволяет правильно оценить отдачу от инвестиций. Затраты на внедрение ТС ПО обычно недооцениваются. Среди конкретных статей затрат на внедрение можно выделить следующие:

- специалисты по планированию внедрения ТС ПО;
- выбор и установка инструментальных средств;
- учет специфических требований персонала;
- приобретение ТС ПО и обучение;
- настройка инструментальных средств;
- подготовка документации, стандартов и процедур использования ТС ПО;
- интеграция с другими технологиями и существующими данными;
- освоение ТС ПО разработчиками;
- технические средства;
- обновление версий.

Важно также осознавать, что улучшение деятельности организации, являющееся следствием использования ТС ПО, может быть неочевидным в течение самого первого проекта, использующего новую технологию. Продуктивность и другие характеристики деятельности организации могут первоначально даже ухудшиться, поскольку на освоение новых средств и внесение необходимых изменений в процесс разработки требуется некоторое время. Таким образом, ожидаемые результаты должны рассматриваться с учетом вероятной отсрочки в улучшении проектных характеристик.

Каждая потребность должна иметь определенный приоритет, зависящий от того, насколько критической она является для достижения успеха в организации. В конечном счете должно четко прослеживаться воздействие каждой функции или возможности приобретаемых средств на удовлетворение каких-

либо потребностей.

Результатом данного действия является формулировка потребностей с их приоритетами, которая используется на этапе оценки и выбора в качестве «пользовательских потребностей».

Определение критериев успешного внедрения

Определяемые критерии должны позволять количественно оценивать степень удовлетворения каждой из потребностей, связанных с внедрением. Кроме того, по каждому критерию должно быть установлено его конкретное оптимальное значение. На отдельных этапах внедрения эти критерии должны анализироваться для того, чтобы оценить текущую степень удовлетворения потребностей.

Как правило, большинство организаций осуществляет внедрение ТС ПО для повышения продуктивности процессов разработки и сопровождения ПО, а также качества результатов разработки. Однако ряд организаций не занимается и не занимался ранее сбором количественных данных по указанным параметрам. Отсутствие таких данных затрудняет количественную оценку воздействия, оказываемого внедрением ТС ПО. Для таких организаций рекомендуется разработка соответствующих метрик.

Если базовые метрические данные отсутствуют, организация зачастую может извлечь полезную информацию из своих проектных архивов.

Помимо продуктивности и качества, полезную информацию о состоянии внедрения ТС ПО также могут дать и другие характеристики организационных процессов и персонала. Например, оценка степени успешности внедрения может включать процент проектов, использующих технологию, рейтинговые оценки уровня квалификации специалистов, связанные с использованием ТС ПО, и результаты опросов персонала по поводу отношения к использованию ТС ПО. Приведем другие проектные характеристики, которые могут быть оценены количественно:

- согласованность проектных результатов;
- точность стоимостных и плановых оценок;

- изменчивость внешних требований;
- соблюдение стандартов организации;
- степень повторного использования существующих компонентов ПО;
- объем и виды необходимого обучения;
- типы и моменты обнаружения проектных ошибок.

Разработка стратегии внедрения ТС ПО

Стратегия должна обеспечивать удовлетворение определенных ранее потребностей и критериев. Данная стратегия определяет:

- организационные потребности;
- базовые метрики, необходимые для последующего сравнения результатов;
- критерии успешного внедрения, связанные с удовлетворением организационных потребностей, включая ожидаемые результаты последовательных этапов процесса внедрения;
- подразделения организации, в которых должно выполняться внедрение ТС ПО;
- влияние, оказываемое на другие подразделения организации;
- стратегии и планы оценки и выбора, пилотного проектирования и перехода к полномасштабному внедрению;
- основные факторы риска;
- ориентировочный уровень и источники финансирования процесса внедрения ТС ПО;
- ключевой персонал и другие ресурсы.

Необходимо отметить, что внедрение новой ТС ПО может включать важные и трудные изменения в культуре организации. Большое внимание должно уделяться ролям различных групп, вовлеченных в процесс таких изменений. Наиболее существенными являются следующие роли:

- спонсор (обычно из числа менеджеров высшего уровня). Данная роль является критической для поддержки проекта и обеспечения необходимого

финансирования. Спонсор должен обладать четким пониманием необходимости серьезных усилий, связанных с внедрением ТС ПО, и быть готов к длительному периоду ожидания осязаемых результатов;

- исполнитель — обычно лицо (или группа лиц), осознающее потенциальные возможности новой ТС ПО, пользующееся авторитетом среди технического персонала и способное возглавить процесс внедрения новой ТС ПО;
- целевая группа — обычно включает менеджеров и технический персонал, которые будут привлечены к непосредственному использованию ТС ПО, а также специалистов, которые будут привлечены косвенно, таких, как специалисты по документированию, персонал поддержки сети и заказчики.

Должны быть определены потребности каждой такой группы и план их эффективного удовлетворения.

В общем случае внедрение ТС ПО должно управляться и финансироваться таким же образом, как и любой проект разработки ПО. Стратегия внедрения может быть пересмотрена в случае появления дополнительной информации.

5.3.3.

ОЦЕНКА И ВЫБОР ТС ПО

Входной информацией для процесса оценки и выбора являются:

- требования к ТС ПО;
- цели и ограничения проекта;
- данные о доступных технологиях;
- список критериев, используемых в процессе оценки.

Процесс оценки и (или) выбора может быть начат только тогда, когда лицо, группа или организация полностью определили для себя конкретные потребности и формализовали их в виде количественных и качественных требований в

заданной предметной области. Термин «требования» далее означает именно такие формализованные требования.

Пользователь должен определить конкретный порядок действий и принятия решений с любыми необходимыми итерациями. Например, процесс может быть представлен в виде дерева решений с его последовательным обходом и выбором подмножеств кандидатов для более детальной оценки. Описание последовательности действий должно определять поток данных между ними.

Определение списка критериев основано на требованиях и включает:

- выбор критериев для использования из приведенного далее перечня;
- определение дополнительных критериев;
- определение области использования каждого критерия (оценка, выбор или оба процесса);
- определение одной или более метрики для каждого критерия для использования при оценке;
- назначение веса каждому критерию при выборе.

Цель процесса оценки — определение функциональности и качества ТС ПО для последующего выбора. Оценка выполняется в соответствии с конкретными критериями, ее результаты включают как объективные, так и субъективные данные по каждой ТС ПО.

Процесс оценки включает следующие действия:

- формулировка задачи оценки, включая информацию о цели и масштабах оценки;
- определение критериев оценки, вытекающее из определения задачи;
- определение технологий-кандидатов путем просмотра списка кандидатов и анализа информации о конкретных технологиях;
- оценка технологий-кандидатов в контексте выбранных критериев. Необходимые для этого данные могут быть получены путем анализа самих технологий и их документации, опроса пользователей, работы с демо-версиями, выполнения тестовых примеров, экспериментального применения и анализа результатов предшествующих оценок;

- подготовка отчета по результатам оценки.

Одним из важнейших критериев в процессе оценки может быть потенциальная возможность интеграции между каждой из технологий-кандидатов и другими технологиями, уже находящимися в эксплуатации или планируемыми к использованию в данной организации.

Список ТС ПО — возможных кандидатов формируется из различных источников: обзоров рынка ПО, информации поставщиков, обзоров технологий и др.

Следующим шагом является получение информации о технологиях или получение их самих, или и то, и другое. Эта информация может состоять из оценок независимых экспертов, сообщений и отчетов поставщиков технологий, результатов демонстрации их возможностей со стороны поставщиков и информации, полученной непосредственно от реальных пользователей. Сами технологии могут быть получены путем приобретения, в виде оценочной копии или другими способами.

Оценка и накопление соответствующих данных могут выполняться следующими способами:

- анализ технологий и документации поставщика;
- опрос реальных пользователей;
- анализ результатов проектов, использовавших данные технологии;
- просмотр демонстраций и опрос демонстраторов;
- выполнение тестовых примеров;
- применение технологий в пилотных проектах;
- анализ любых доступных результатов предыдущих оценок.

Существуют как объективные, так и субъективные критерии. Результаты оценки в соответствии с конкретным критерием могут быть двоичными, находиться в некотором числовом диапазоне, представлять собой просто числовое значение или иметь какую-либо другую форму.

Для **объективных критериев** оценка должна выполняться путем воспроизводимой процедуры, чтобы любой другой специалист, выполняющий оценку,

мог получить такие же результаты. Если используются тестовые примеры, их набор должен быть заранее определен, унифицирован и документирован.

Для *субъективных критериев* ТС ПО должна оцениваться более чем одним специалистом или группой с использованием одних и тех же критериев. Необходимый уровень опыта специалистов или групп должен быть заранее определен.

Результаты оценки должны быть стандартным образом документированы (для облегчения последующего использования) и при необходимости утверждены.

Отчет по результатам оценки должен содержать следующую информацию:

- введение — общий обзор процесса и перечень основных результатов;
- предпосылки — цель оценки и желаемые результаты, период времени, в течение которого выполнялась оценка, определение ролей и соответствующего опыта специалистов, выполнявших оценку;
- подход к оценке - описание общего подхода, информацию, определяющую контекст и масштаб оценки, а также любые предположения и ограничения;
- информацию о технологиях, содержащую: 1) наименование; 2) версию; 3) данные о поставщике, включая контактный адрес и телефон; 4) конфигурацию технических средств; 5) стоимостные данные; 6) описание, включающее поддерживаемые данной технологией процессы создания и сопровождения ПО и область применения;
- этапы оценки — выполняемые в процессе оценки конкретные действия, описанные со степенью детализации, необходимой как для понимания масштаба и глубины оценки, так и для ее повторения при необходимости;
- конкретные результаты — результаты оценки, представленные в терминах критериев оценки. В тех случаях, когда отчет охватывает целый ряд технологий или результаты данной оценки будут сопоставляться с аналогичными результатами других оценок, необходимо обратить особое внимание на формат представления результатов, способствующий такому

сравнению. Субъективные результаты должны быть отделены от объективных и сопровождаться необходимыми пояснениями;

- выводы и заключения;
- приложения — формулировка задачи оценки и уточненный список критериев.

Процессы оценки и выбора тесно взаимосвязаны. По результатам оценки цели выбора и (или) критерии выбора и их веса могут потребовать модификации. В таких случаях может понадобиться повторная оценка. Когда анализируются окончательные результаты оценки и к ним применяются критерии выбора, может быть рекомендовано приобретение технологии. Альтернативой может быть отсутствие адекватных технологий, в этом случае рекомендуется разработать новую технологию, модифицировать существующую или отказаться от внедрения.

Процесс выбора включает в себя следующие действия:

- формулировка задач выбора, включая цели, предположения и ограничения;
- выполнение всех необходимых действий по выбору, включая определение и ранжирование критериев, определение технологий-кандидатов, сбор необходимых данных и применение ранжированных критериев к результатам оценки для определения средств с наилучшими показателями;
- выполнение необходимого количества итераций с тем, чтобы выбрать (или отвергнуть) технологии, имеющие сходные показатели;
- подготовка отчета по результатам выбора.

В процессе выбора возможно получение двух результатов:

- рекомендаций по выбору конкретной технологии;
- запроса на получение дополнительной информации к процессу оценки.

В том случае, если предыдущие оценки выполнялись с использованием различных наборов критериев или с использованием конкретных критериев, но различными способами, результаты оценок должны быть представлены в согласованной форме. После завершения данного шага оценка каждой технологии

должна быть представлена в рамках единого набора критериев и сопоставима с другими оценками.

Алгоритмы, обычно используемые для выбора, могут быть основаны на масштабе или ранге. Алгоритмы, основанные на масштабе, вычисляют единственное значение для каждой технологии путем умножения веса каждого критерия на его значение (с учетом масштаба) и сложения всех произведений. ТС ПО с наивысшим результатом получает первый ранг. Алгоритмы, основанные на ранге, используют ранжирование ТС ПО — кандидатов по отдельным критериям или группам критериев в соответствии со значениями критериев в заданном масштабе. Затем аналогично предыдущему ранги сводятся вместе и вычисляются общие значения рангов.

При анализе результатов выбора предполагается, что процесс выбора завершен, ТС ПО выбрана и рекомендована к использованию. Тем не менее, может потребоваться более точный анализ для определения степени зависимости значений ключевых критериев от различий в значениях характеристик ТС ПО — кандидатов. Такой анализ позволит установить, насколько результат ранжирования ТС ПО зависит от оптимальности выбора весовых коэффициентов критериев. Он также может использоваться для определения существенных различий между ТС ПО с очень близкими значениями критериев или рангами.

Если ни одна из ТС ПО не удовлетворяет минимальным критериям, выбор (возможно, вместе с оценкой) может быть повторен для других ТС ПО — кандидатов.

Если различия между самыми предпочтительными кандидатами несущественны, дополнительная информация может быть получена путем повторного выбора (возможно, вместе с оценкой) с использованием дополнительных или других критериев.

Рекомендации по выбору должны быть строго обоснованы. В случае отсутствия адекватных ТС ПО, как было отмечено выше, рекомендуется разработать новую технологию, модифицировать существующую или отказаться от внедрения.

КРИТЕРИИ ОЦЕНКИ И ВЫБОРА ТС ПО

Критерии формируют базис для процессов оценки и выбора и могут принимать различные формы:

- числовые меры в широком диапазоне значений, например, объем требуемых ресурсов;
- числовые меры в ограниченном диапазоне значений, например простота освоения, выраженная в баллах от 1 до 5;
- двоичные меры (истина/ложь, да/нет), например способность генерации документации в заданном формате;
- меры, которые могут принимать одно значение или более из конечных множеств значений, например платформы, для которых поддерживается ТС ПО.

Типичный процесс оценки и (или) выбора может использовать набор критериев различных типов. Каждый критерий должен быть выбран и адаптирован экспертом с учетом особенностей конкретного процесса. В большинстве случаев только некоторые из множества описанных ниже критериев оказываются приемлемыми для использования, при этом также добавляются дополнительные критерии. Выбор и уточнение набора используемых критериев является критическим шагом в процессе оценки и (или) выбора.

Факторы выбора ТС ПО:

- характеристики объекта внедрения, определяющие требования, предъявляемые к ТС ПО;
- параметры доступных ТС ПО;
- ресурсы проекта (финансовые, кадровые и технические).

Исходные данные для выбора и оценки применимости — набор параметров (техничко-экономических характеристик) ТС ПО (рис. 5.4).

1. Функциональные характеристики, ориентированные на процессы

жизненного цикла ПО.

1.1. Управление проектом:

- управление работами и ресурсами (контроль и управление процессом проектирования ПО в терминах структуры заданий и назначения исполнителей, последовательности их выполнения, завершенности отдельных этапов проекта и проекта в целом; возможность поддержки плановых данных, фактических данных и их анализа. Типичные данные включают графики (с учетом календаря, рабочих часов, выходных и др.), компьютерные ресурсы, распределение персонала, бюджет и др.);
- оценка затрат, трудоемкости и времени;
- планирование;
- мониторинг проекта.

1.2. Разработка ПО

1.2.1. Моделирование:



Рис. 5.4. Совокупность параметров (технико-экономических характеристик) ТС ПО

- построение и анализ диаграмм (возможность создания и редактирования диаграмм различных типов, возможность анализа графических объектов, а также хранения и представления проектной информации в графическом виде);
- спецификация требований и проектных решений (включая возможность импорта, экспорта и редактирования спецификаций с использованием формального языка);
- моделирование данных и процессов;
- проектирование архитектуры ПО;

- имитационное моделирование (возможность динамического моделирования различных аспектов функционирования системы на основе спецификаций требований и (или) проектных спецификаций, включая внешний интерфейс и производительность, например, время отклика, коэффициент использования ресурсов и пропускную способность);
- прототипирование (возможность проектирования и генерации предварительного варианта всей системы или ее отдельных компонентов на основе спецификаций требований и (или) проектных спецификаций);
- трассировка (возможность сквозного анализа функционирования системы от спецификации требований до конечных результатов (установления и отслеживания соответствий и связей между функциональными и другими внешними требованиями к ПО и техническими решениями и результатами проектирования); прямая трассировка (проверка учета всех требований) и обратная трассировка (поиск проектных решений, не связанных ни с какими внешними требованиями)).

1.2.2. Программирование:

- генерация кода (программного кода, схемы базы данных, запросов, экранов/меню);
- синтаксически управляемое редактирование (возможность ввода и редактирования исходных кодов на одном или нескольких языках с одновременным синтаксическим контролем);
- компиляция кода;
- конвертирование кода;
- отладка (трассировка программ, выделение узких мест и наиболее часто используемых фрагментов кода и т.д.).

1.3. Сопровождение:

- идентификация и локализация проблем;
- реверсный инжиниринг (возможность анализа существующих исходных кодов и формирования на их основе проектных спецификаций);
- реструктуризация исходного кода (возможность модификации формата и

(или) структуры существующего исходного кода).

1.4. Документирование:

- редактирование текстов и графики;
- редактирование с помощью форм;
- возможности издательских систем;
- поддержка функций и форматов гипертекста;
- автоматическое извлечение данных из репозитория и генерация документации.

1.5. Управление конфигурацией:

- контроль доступа и изменений (возможность контроля доступа на физическом уровне к элементам данных и контроля изменений. Контроль доступа включает возможности определения прав доступа к компонентам, а также извлечения элементов данных для модификации, блокировки доступа к ним на время модификации и помещения обратно в репозиторий);
- отслеживание модификаций (фиксация и ведение журнала всех модификаций, внесенных в систему в процессе разработки или сопровождения);
- управление версиями (ведение и контроль данных о версиях системы и всех ее коллективно используемых компонентах);
- учет состояния объектов управления конфигурацией (возможность получения отчетов о всех последовательных версиях, содержанием и состоянии различных объектов конфигурационного управления);
- генерация версий и модификаций (поддержка пользовательского описания последовательности действий, требуемых для формирования версий и модификаций, и автоматическое выполнение этих действий);
- архивирование (возможность автоматического архивирования элементов данных для последующего использования).

1.6. Обеспечение качества:

- управление данными о качестве;
- управление рисками.

1.7. Верификация:

- анализ трассировки спецификаций;
- анализ спецификаций (синтаксический и семантический контроль проектных спецификаций - контроль синтаксиса диаграмм и типов их элементов, контроль декомпозиции функций, проверка спецификаций на полноту и непротиворечивость);
- анализ исходного кода (определение размера кода, вычисление показателей сложности, генерация перекрестных ссылок и проверка на соответствие стандартам).

1.8. Аттестация:

- описание тестов (генерация тестовых данных, алгоритмов тестирования, требуемых результатов и т.д.);
- фиксация и повторение действий оператора (возможность фиксировать данные, вводимые оператором с помощью клавиатуры, мыши и т.д., редактировать их и воспроизводить в тестовых примерах);
- автоматический запуск тестовых примеров;
- регрессионное тестирование (возможность повторения и модификации ранее выполненных тестов для определения различий в системе и (или) среде);
- статистический анализ результатов тестирования;
- анализ тестового покрытия (оснащенность средствами контроля исходного кода и анализ тестового покрытия. Проверяются, в частности, исполняемые и вызываемые (или нет) операторы, процедуры и переменные);
- анализ производительности (возможность анализа производительности программ. Анализируемые параметры производительности могут включать степень использования ресурсов центрального процессора и памяти, количество обращений к определенным элементам данных и (или) сегментам кода, временные характеристики и т.д.);
- анализ исключительных ситуаций;

- динамическое моделирование среды;
 - интеграционное тестирование.
2. **Функциональные характеристики применения (инструментальных средств).**
- 2.1. Среда функционирования:
- требуемые технические средства;
 - требуемое ПО.
- 2.2. Совместимость:
- совместимость с различными средами функционирования;
 - совместимость с другими средствами по данным, представлению и управлению (способность к взаимодействию с другими средствами, включая непосредственный обмен данными (примерами таких средств являются текстовые процессоры и другие средства документирования, базы данных и другие CASE-средства) и возможность преобразования репозитория или его части в стандартный формат для обработки другими средствами).
- 2.3. Аспекты применения:
- соответствие технологическим стандартам (касающимся языков, баз данных, репозитория, коммуникаций, графического интерфейса пользователя, документации, безопасности, стандартов обмена информацией);
 - область применения (системы обработки транзакций, системы реального времени, экономические информационные системы и др.);
 - размер поддерживаемых приложений (определяет ограничения на такие величины, как количество строк кода, уровней вложенности, размер базы данных, количество элементов данных, количество объектов управления конфигурацией);
 - поддерживаемые методы (набор методов и методик, поддерживаемых CASE-средством. Примерами являются структурный или объектно-ориентированный анализ и проектирование);
 - поддерживаемые языки (все языки, используемые CASE-средством: язы-

ки программирования, языки баз данных, графические языки, языки спецификации проектных требований и интерфейсы операционных систем);

- локализация.

3. Характеристики качества.

3.1. Надежность:

- контроль и обеспечение целостности проектных данных;
- автоматическое резервирование;
- безопасность (защита от несанкционированного доступа);
- обработка ошибок (обнаружение ошибок в работе системы, извещение пользователя, корректное завершение работы или сохранение состояния к моменту прерывания);
- анализ отказов в критических приложениях.

3.2. Удобство использования:

- удобство пользовательского интерфейса (удобство расположения и представления часто используемых элементов экрана, способов ввода данных и др.);
- простота освоения (трудовые и временные затраты на освоение средств);
- адаптируемость к конкретным требованиям пользователя: различным алфавитам, режимам текстового и графического представления (слева направо, сверху вниз), различным форматам даты, способам ввода-вывода (экраным формам и форматам), изменениям в методологии (изменениям графических нотаций, правил, свойств и состава predetermined объектов) и др.;
- качество документации и учебных материалов (полнота, понятность, удобочитаемость, полезность компьютерных учебных материалов, учебных пособий, курсов);
- качество диагностики (понятность и полезность диагностических сообщений для пользователя);
- простота установки и обновления версий.

3.3. Эффективность:

- требования к оптимальному размеру внешней и оперативной памяти, типу и производительности процессора, обеспечивающим приемлемый уровень производительности;
- эффективность рабочей нагрузки: эффективность выполнения CASE-средством своих функций в зависимости от интенсивности работы пользователя (например, количество нажатий клавиш или кнопки мыши, требуемое для выполнения определенных функций);
- производительность: время, затрачиваемое CASE-средством для выполнения конкретных задач (например, время ответа на запрос, время анализа 100 000 строк кода).

3.4. Сопровождаемость:

- уровень поддержки со стороны поставщика - скорость разрешения проблем, поставки новых версий, обеспечение дополнительных возможностей;
- трассируемость обновлений — простота освоения отличий новых версий от существующих;
- совместимость обновлений — совместимость новых версий с существующими, включая, например, совместимость по входным или выходным данным;
- адаптируемость к изменениям в методах;
- сопровождаемость конечного продукта.

3.5. Переносимость:

- совместимость с версиями операционной системы;
- переносимость данных между различными версиями средств;
- соответствие стандартам переносимости, включающим документацию, коммуникации и пользовательский интерфейс, оконный интерфейс, языки программирования, языки запросов и др.

4. Общие характеристики.

4.1. Приобретение:

- затраты на технологию, включающие стоимость приобретения, установки, начального сопровождения и обучения. Следует учитывать цену для всех необходимых конфигураций, включая единственную копию, много копий, локальную лицензию, лицензию для предприятия, сетевую лицензию;
- лицензионная политика — доступные возможности лицензирования, право копирования (носителей и документации), любые ограничения и (или) штрафные санкции за вторичное использование (подразумевается продажа пользователем ТС ПО продуктов, в состав которых входят некоторые компоненты ТС ПО, использовавшиеся при разработке продуктов);
- экспортные ограничения.

4.2. Реализация:

- оценочный эффект от внедрения ТС ПО — уровень продуктивности, качества и т.д. Такая оценка может потребовать экономического анализа;
- инфраструктура, требуемая для внедрения ТС ПО.

4.3. Поддержка:

- профиль поставщика — общие показатели возможностей поставщика. Профиль поставщика может включать масштаб его организации, стаж в бизнесе, финансовое положение, список любых дополнительных продуктов, деловые связи (в частности, с другими поставщиками технологии), планируемую стратегию развития;
- профиль продукта — общая информация о продукте, включая срок его существования, количество проданных копий, наличие, размер и уровень деятельности пользовательской группы, систему отчетов о проблемах, программу развития продукта, совокупность применений, наличие ошибок и др.;
- доступность и качество обучения (обучение может предоставляться на площади поставщика, пользователя или где-либо в другом месте).

4.4. Сертификация:

- сертификация поставщика — сертификаты, полученные от

специализированных организаций в области создания ПО (например, SEI и ISO), удостоверяющие, что квалификация поставщика в области создания и сопровождения ПО удовлетворяет некоторым минимально необходимым или вполне определенным требованиям. Сертификация может быть неформальной, например на основе анализа качества работы поставщика;

- сертификация продукта;
- поддержка поставщика — доступность, реактивность и качество услуг, предоставляемых поставщиком для пользователей ТС ПО. Такие услуги могут включать телефонную «горячую линию», местную техническую поддержку, поддержку в самой организации.

На основе данного набора параметров анализируются и классифицируются существующие ТС ПО. Общий набор критериев, применяемых для оценки ТС ПО, приведен ниже.

Критерии, применяемые для оценки ТС ПО

Критерий	Определение
Минимум времени обучения	Количество человеко-часов, затрачиваемых на обучение ТС ПО
Минимум затрат на обучение	Стоимость процесса обучения плюс все накладные расходы, связанные с обучением
Минимум трудоемкости создания ПО	Количество человеко-месяцев, затрачиваемых на создание ПО с использованием ТС ПО
Минимум трудоемкости сопровождения ПО	Количество человеко-месяцев, затрачиваемых на сопровождение ПО с использованием ТС ПО
Минимум времени создания ПО	Временной интервал от начала до завершения проекта (сдачи ПО в эксплуатацию) при использовании данной ТС ПО

Максимум про- дуктивности	Объем работы (измеряемый в количестве строк кода или функциональных точек), приходящийся на единицу трудоемкости (человеко-месяц) при использовании данной ТС ПО
Максимум каче- ства создавае- мого ПО	Количество дефектов в рабочих продуктах при использовании данной ТС ПО
Возврат инвести- ций	1. Доход от использования ПО — Затраты на создание и сопровождение ПО / Затраты на создание и сопровождение ПО 2. Трудоемкость создания и сопровождения ПО без использования ТС ПО — Трудоемкость создания и сопровождения ПО с использованием ТС ПО / Трудоемкость создания и сопровождения ПО с использованием ТС ПО
Минимум затрат на создание ПО	Стоимость выполнения проекта (до сдачи ПО в эксплуатацию) с использованием ТС ПО
Минимум затрат на сопровожде- ние ПО	Отношение стоимости сопровождения ПО при использовании данной ТС ПО к совокупным затратам на информационные ТС ПО в организации
Минимум време- ни внедрения ТС ПО	Временной интервал от начала внедрения ТС ПО до выхода на безубыточный уровень (начало возврата инвестиций в ТС ПО)
Минимум затрат на внедрение ТС ПО	Суммарная стоимость приобретения, обучения и сопровождения ТС ПО
Минимальный срок окупаемости затрат на внедре-	Временной интервал от начала внедрения ТС ПО до полной окупаемости затрат на ее внедрение

ние ТС ПО

В результате выполненной оценки может оказаться, что ни одна доступная технология не удовлетворяет в нужной мере всем критериям и не покрывает все потребности проекта. В этом случае может применяться набор средств, позволяющий построить на их базе единую технологическую среду.

5.3.5

ВЫПОЛНЕНИЕ ПИЛОТНОГО ПРОЕКТА

Перед полномасштабным внедрением выбранной ТС ПО в организации выполняется пилотный проект, целью которого является экспериментальная проверка правильности решений, принятых на предыдущих этапах, и подготовка к внедрению.

Пилотный проект представляет собой первоначальное реальное использование ТС ПО в предназначенной для этого среде и обычно подразумевает более широкий масштаб использования ТС ПО по отношению к тому, который был достигнут во время оценки. Пилотный проект должен обладать многими характеристиками реальных проектов, для которых предназначено данное средство. Он преследует следующие цели:

- подтвердить достоверность результатов оценки и выбора;
- установить, действительно ли ТС ПО годится для использования в данной организации, и если да, то определить наиболее подходящую область ее применения;
- собрать информацию, необходимую для разработки плана практического внедрения;
- приобрести собственный опыт использования ТС ПО.

Пилотный проект позволяет получить важную информацию, необходимую для оценки ТС ПО и его поддержки со стороны поставщика после того, как средство установлено.

Важной функцией пилотного проекта является принятие решения относительно приобретения или отказа от использования ТС ПО. Провал пилотного проекта позволяет избежать более значительных и дорогостоящих неудач в дальнейшем, поскольку пилотный проект обычно требует приобретения относительно небольшого количества лицензий и обучения узкого круга специалистов.

Первоначальное использование новой ТС ПО в пилотном проекте должно тщательно планироваться и контролироваться. Пилотный проект включает следующие шаги (рис. 5.5).

Определение характеристик пилотного проекта

Пилотный проект должен обладать следующими характеристиками.

Типичность предметной области. Чтобы облегчить окончательное определение области применения ТС ПО, предметная область пилотного проекта должна быть типичной для обычной деятельности организации. Пилотный проект должен помочь определить любую дополнительную технологию, обучение или поддержку, которые необходимы для перехода от пилотного проекта к широкомасштабному использованию ТС ПО. В рамках этих ограничений пилотный проект должен иметь небольшой, но значимый размер.

Масштабируемость. Результаты, полученные в пилотном проекте, должны показать масштабируемость ТС ПО. Цель — получить четкое представление о масштабах проектов, для которых данная ТС ПО применима.

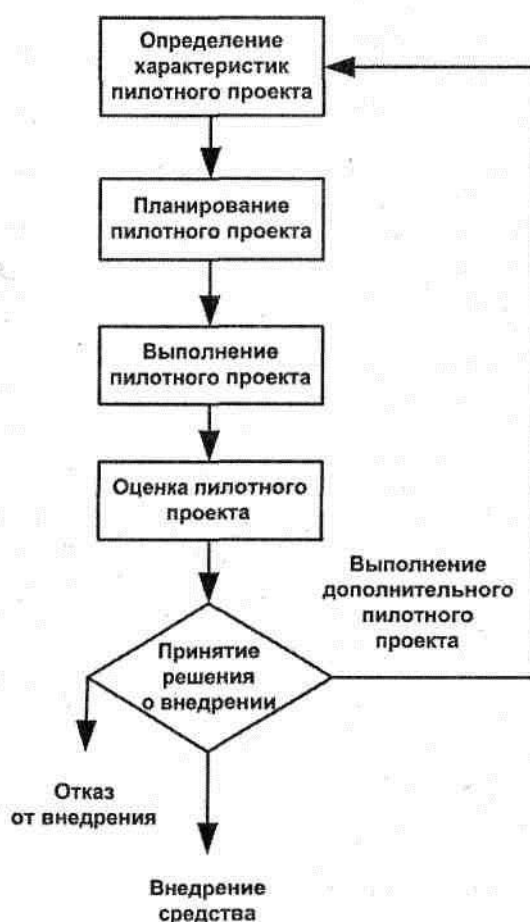


Рис. 5.5. Шаги пилотного проекта

Представительность. Пилотный проект не должен быть необычным или уникальным для организации. ТС ПО должна использоваться для решения задач, относящихся к предметной области, хорошо понимаемой всей организацией.

Критичность. Пилотный проект должен иметь существенную значимость, чтобы оказаться в центре внимания, но не должен быть критичным для успешной деятельности организации в целом. Необходимо осознавать, что первоначальное внедрение новой ТС ПО подразумевает определенный риск. При выборе пилотного проекта приходится решать следующую дилемму: успех незначительного проекта может остаться незамеченным, с другой стороны, провал значимого проекта может вызвать чрезмерную критику.

Авторитетность. Группа специалистов, участвующих в проекте, должна обладать высоким авторитетом, при этом результаты проекта будут всерьез восприняты остальными сотрудниками организации.

Готовность проектной группы. Проектная группа должна обладать готовностью к нововведениям, технической зрелостью и приемлемым уровнем опыта и знаний в данной ТС ПО и предметной области. С другой стороны, группа должна отражать в миниатюре характеристики организации в целом.

В большинстве случаев существует баланс между желанием реализовать идеальный пилотный проект и реальными ограничениями организации. Организация должна выбрать пилотный проект таким образом, чтобы, во-первых, способ использования ТС ПО в нем совпадал с дальнейшими планами и, во-вторых, перечисленные выше характеристики были сбалансированы с реальными условиями организации.

Кроме того, организация должна учитывать продолжительность пилотного проекта (и в целом процесса внедрения). Слишком продолжительный проект связан с риском потери интереса к нему со стороны руководства.

Планирование пилотного проекта

Планирование пилотного проекта должно по возможности вписываться в обычный процесс планирования проектов в организации. План должен содержать следующую информацию:

- цели, задачи и критерии оценки;
- персонал;
- процедуры и соглашения;
- обучение;
- график и ресурсы.

Цели, задачи и критерии оценки

Ожидаемые результаты пилотного проекта должны быть четко определены. Степень соответствия этим результатам представляет собой основу для последующей оценки проекта. Для формирования такой основы необходимо выполнить следующие действия:

- описать проект в терминах ожидаемых результатов (т.е. конечного продукта). Описание должно включать форму представления и содержание результатов. Должны быть четко определены договорные требования и

соответствующие стандарты;

- определить общие цели проекта, а именно: насколько хорошо и до какой степени ТС ПО планируется использовать в среде данного проекта. Примером цели может быть определение степени улучшения качества проектной документации в результате применения ТС ПО;
- определить конкретные задачи, реализующие поставленные цели. Каждой цели можно поставить в соответствие одну или несколько конкретных задач с количественно оцениваемыми результатами. Примером такой задачи может быть сравнительный анализ качества документации, полученной с помощью ТС ПО и без нее. Документация может включать спецификацию требований к ПО, высокоуровневые и детальные проектные спецификации;
- установить критерии оценки результатов. Чтобы определить степень успеха пилотного проекта, необходимо использовать набор критериев, основанных на упомянутых выше задачах. Примером критерия может быть степень непротиворечивости проектной документации и контролируемости выполнения требований к ПО. Значения критериев должны сравниваться с базовыми значениями, полученными до выполнения пилотного проекта.

Персонал

Специалисты, выбранные для участия в пилотном проекте, должны иметь соответствующий авторитет и влияние и быть сторонниками новой ТС ПО. Группа должна включать как технических специалистов, так и менеджеров, заинтересованных в новой ТС ПО и разбирающихся в ее использовании. Группа должна обладать высокими способностями к коммуникации, знанием особенностей организационных процессов и процедур, а также предметной области. Группа не должна, тем не менее, состоять полностью из специалистов высшего звена, она должна представлять средний уровень организации.

После завершения пилотного проекта группа должна быть открыта для обмена информацией с остальными специалистами организации относительно

возможностей нового средства и опыта, полученного при его использовании. Может оказаться желательным рассредоточить проектную группу по всей организации в целях распространения их опыта и знаний.

Процедуры и соглашения

Необходимо четко определить процедуры и соглашения, регулирующие использование ТС ПО в пилотном проекте. Эта задача, скорее всего, может оказаться более долгой и сложной, чем ожидается, при этом может оказаться необходимым привлечение сторонних экспертов. Примерами процедур и соглашений, которые могут повлиять на успех пилотного проекта, являются методы, технические соглашения (в частности, соглашения по наименованиям и структуре каталогов, стандарты проектирования и программирования) и организационные соглашения (в частности, правила учета использования ресурсов, авторизации и контроля изменений, а также процедуры экспертизы, проверки качества и подготовки отчетов).

В пилотном проекте по возможности должны использоваться существующие в организации процедуры и соглашения. С другой стороны, в течение пилотного проекта неэффективные или чересчур ограничивающие процедуры и соглашения могут развиваться и совершенствоваться по мере накопления опыта применения средства. При этом те изменения, которые предлагается в них вносить, должны документироваться.

Обучение

Должны быть определены виды и объем обучения, необходимого для пилотного проекта. Планируемое обучение должно обеспечивать три вида потребностей: технические, управленческие и мотивационные. Ресурсы, требуемые для обучения (учебные аудитории и оборудование, преподаватели и учебные материалы), должны соответствовать плану пилотного проекта.

График обучения должен определять как специалистов, подлежащих обучению, так и виды обучения, которое они должны пройти. Обучение, которое проводится в период выполнения проекта, должно начинаться как можно быстрее после начала проекта. Обучение средствам, процессам или методам, кото-

рые не будут использоваться в течение нескольких месяцев после начала проекта, должно планироваться в то время, когда в них возникнет реальная потребность.

Поставщики ТС ПО обычно предлагают обучение использованию поставляемых ими средств. Помимо этого для некоторых средств может быть необходимо обучение методологии. Некоторые виды обучения могут быть недоступны со стороны коммерческих организаций, они должны выполняться собственными силами. Такие виды обучения включают использование ТС ПО в контексте процессов, происходящих в организации, а также в совокупности с другими средствами в данной среде. Часть плана пилотного проекта, связанная с обучением, должна использоваться в качестве входа для плана практического внедрения.

При выборе необходимого обучения должны приниматься во внимание следующие факторы:

- квалификация преподавателей;
- соответствие обучения характеристикам конкретных групп специалистов (например, обзорные курсы для менеджеров, подробные курсы для разработчиков);
- возможность проведения курсов непосредственно на рабочих местах;
- возможность проведения расширенных курсов;
- возможность подготовки самих преподавателей.

График и ресурсы

Должен быть разработан график, включающий ресурсы и сроки (этапы) проведения работ. Ресурсы включают персонал, технические средства, ПО и финансирование. Данные о персонале могут определять конкретных специалистов или требования к квалификации, необходимой для успешного выполнения пилотного проекта. Финансирование должно определяться отдельно по каждому виду работ: приобретение ТС ПО, установка, обучение, отдельные этапы проектирования.

Выполнение пилотного проекта

Пилотный проект должен выполняться в соответствии с планом. Организационная деятельность, связанная с выполнением пилотного проекта и подготовкой отчетов, должна осуществляться в установленном порядке. Пилотная природа проекта требует специального внимания к вопросам приобретения, поддержки, экспертизы и обновления версий.

Приобретение, установка и интеграция

После того как ТС ПО выбрана, она должна быть приобретена, интегрирована в проектную среду и настроена в соответствии с требованиями пилотного проекта. Границы этой деятельности зависят от тех действий, которые имели место в процессе оценки и выбора, а также от степени модификации средства, необходимой для его использования в проекте.

Процесс приобретения может включать подготовку контракта, переговоры, лицензирование и другую деятельность, которая выходит за рамки данных рекомендаций. Эта деятельность требует затрат времени и человеческих ресурсов, которые должны быть учтены при планировании. План должен предусматривать отказ от выбранной ТС ПО на данном этапе из-за договорных разногласий.

После приобретения средства должны быть установлены, оттестированы и приняты в эксплуатацию. Тестирование позволяет убедиться, что поставленный продукт соответствует требованиям контракта, обладает необходимой полнотой и корректностью. Этап приемки может быть предусмотрен контрактом, его реальный срок может отличаться от того, который был предусмотрен первоначально в плане пилотного проекта. Особое внимание необходимо уделить соблюдению всех требований поставщика к параметрам среды функционирования ТС ПО.

После завершения приемки может потребоваться некоторая настройка и интеграция. Настройка может включать модификацию интерфейсов, связанную с требованиями специалистов проектной группы, а также с установкой прав доступа и привилегий. Настройка должна оставаться в рамках тех возможностей, которые предоставляет сама ТС ПО.

Если новые средства должны использоваться в совокупности с некоторыми

другими средствами, необходимо определить взаимодействие средств и требуемую интеграцию. Для интеграции новых средств с существующими может потребоваться построение специальных оболочек. Сложная интеграция может потребовать привлечения сторонних экспертов.

Поддержка

Доступная поддержка должна включать (по соглашению) «горячую линию» поставщика и поддержку местного поставщика, поддержку в самой организации, контакты с опытными пользователями в других организациях и участие в работе групп пользователей.

Внутренняя поддержка должна включать доступ к специалистам, знакомым с установкой средств и работой с ними. Существует несколько возможных вариантов получения такой поддержки (например, от специалиста данной организации, имеющего опыт предшествующей работы со средством; участников процесса оценки и выбора или опытного консультанта). Такой тип поддержки должен специальным образом планироваться и администрироваться. Особое внимание должно быть уделено средствам, работающим в сетях или обладающих репозиториями, поддерживающими многопользовательскую работу.

Периодические экспертизы

Обычные процедуры экспертизы проектов, существующие в организации, должны выполняться и для пилотного проекта, при этом особое внимание должно уделяться именно пилотным аспектам проекта. Помимо этого результаты экспертиз должны служить мерой успешного использования ТС ПО.

Обновление версий

Пользователи ТС ПО могут ожидать периодического обновления версий со стороны поставщика в течение выполнения пилотного проекта. При этом необходимо тщательное отношение к интеграции этих версий. Следует заранее оценить влияние этих обновлений на ход проекта. Новые версии могут как обеспечить новые возможности, так и породить новые проблемы. В то же время новая версия может потребовать видоизмененного или дополнительного обучения, а также может оказать отрицательное воздействие на уже выполненную к этому

моменту работу.

Оценка пилотного проекта

После завершения пилотного проекта его результаты необходимо оценить и сопоставить с начальными потребностями организации, критериями успешного внедрения ТС ПО, базовыми метриками и критериями успеха пилотного проекта. Такая оценка должна установить возможные проблемы и важнейшие характеристики пилотного проекта, которые могут повлиять на пригодность ТС ПО для организации. Она должна также указать проекты или структурные подразделения внутри организации, для которых данная ТС ПО является подходящей. Помимо этого оценка может дать информацию относительно совершенствования процесса внедрения в дальнейшем.

В процессе оценки пилотного проекта организация должна определить свою позицию по следующим трем вопросам.

- Целесообразно ли внедрять ТС ПО?
- Какие конкретные особенности пилотного проекта привели к его успеху (или неудаче)?
- Какие проекты или подразделения в организации могли бы получить выгоду от использования ТС ПО?

Принятие решения о внедрении

На данном этапе процесса внедрения организация должна сделать существенные инвестиции в ТС ПО. Если ТС ПО удовлетворила или даже превысила ожидания организации, то решение об ее внедрении может быть принято достаточно просто и быстро.

С другой стороны, может оказаться, что в рамках пилотного проекта ТС ПО не оправдала тех ожиданий, которые на нее возлагались, или же в пилотном проекте она использовалась удовлетворительно, однако опыт показал, что дальнейшие вложения в ТС ПО не гарантируют успеха.

Возможны четыре варианта результатов и соответствующих действий.

- Пилотный проект потерпел неудачу, и его анализ показал неадекватность ожиданий организации. В этом случае организация может пересмотреть

результаты проекта в контексте более реалистичных ожиданий.

- Пилотный проект потерпел неудачу, и его анализ показал, что выбранные средства не удовлетворяют потребности организации. В этом случае организация может принять решение не внедрять данные средства, однако при этом также пересмотреть свои потребности и подход к оценке и выбору ТС ПО.
- Пилотный проект потерпел неудачу, и его анализ показал наличие таких проблем, как неудачный выбор пилотного проекта, неадекватное обучение и недостаток ресурсов. В этом случае может оказаться достаточно сложным принять решение о том, следует ли вновь выполнить пилотный проект, продолжить работу по внедрению или отказаться от ТС ПО. Однако независимо от принятого решения процесс внедрения нуждается в пересмотре и повышенном внимании.
- Пилотный проект завершился успешно, и признано целесообразным внедрять ТС ПО в некоторых подразделениях или, возможно, во всей организации. В этом случае следующим шагом является определение наиболее подходящего масштаба внедрения.

В ряде случаев анализ пилотного проекта может показать, что причиной неудачи явился более чем один фактор. Последующие попытки внедрения ТС ПО должны четко выявить все причины неудачи. В экстремальных случаях тщательный анализ может показать, что в настоящий момент организация просто не готова к успешному внедрению сложных ТС ПО. В такой ситуации организация может попытаться решить свои проблемы другими средствами.

Особенности пилотного проекта

Очень важно провести анализ пилотного проекта с тем, чтобы определить его элементы, являющиеся критическими для успеха, и степень отражения этими элементами организации в целом. Например, если в пилотном проекте участвуют самые лучшие специалисты организации, он может закончиться успешно даже вопреки использованию ТС ПО, а не благодаря ей. С другой стороны, ТС ПО может быть применена для разработки приложений, для которых

она явно не подходит по своим характеристикам. Тем не менее, такое использование могло бы указать на область наиболее рационального применения ТС ПО в данной организации.

Отметим важнейшие характеристики пилотного проекта, не являющиеся представительными для организации в целом:

- процессы в пилотном проекте в чем-либо отличаются от процессов во всей организации;
- квалификация группы пилотного проекта не отражает квалификацию остальных специалистов организации;
- ресурсы, выделенные на выполнение проекта, могут отличаться от тех, которые выделяются для обычных проектов;
- предметная область или масштаб проекта могут отличаться от других проектов.

Выгода от использования ТС ПО

Результаты пилотного проекта следует сопоставить с возможностями организации в целом. Например, если наиболее заинтересованные и квалифицированные участники проекта столкнулись с серьезными трудностями в освоении ТС ПО, то менее заинтересованным и квалифицированным программистам из других подразделений потребуется существенно большее обучение.

Пилотный проект может также показать, что ТС ПО целесообразно использовать для некоторых классов проектов и нецелесообразно — для других. Например, средство формальной верификации может подходить для жизненно важных приложений и не подходить для менее критических приложений.

Перед разработкой плана перехода организация должна оценить ожидаемый эффект для различных подразделений или классов проектов. При этом следует учитывать, что некоторые подразделения могут не обладать необходимой квалификацией или ресурсами для использования ТС ПО.

Варианты решения о внедрении

Возможным решением должно быть одно из следующих.

- Внедрить ТС ПО. В этом случае рекомендуемый масштаб внедрения

должен быть определен в терминах структурных Подразделений и предметной области.

- Выполнить дополнительный пилотный проект. Такой вариант должен рассматриваться только в том случае, если остались конкретные неразрешенные вопросы относительно внедрения ТС ПО в организации. Новый пилотный проект должен быть таким, чтобы ответить на эти вопросы.
- Отказаться от ТС ПО. В этом случае причины отказа от конкретной ТС ПО должны быть определены в терминах потребностей организации или критериев, которые остались неудовлетворенными. Перед тем как продолжить деятельность по внедрению ТС ПО, потребности организации должны быть пересмотрены на предмет своей обоснованности.
- Отказаться от использования ТС ПО вообще. Пилотный проект может показать, что организация либо не готова к внедрению ТС ПО, либо автоматизация данного аспекта процесса создания и сопровождения ПО не дает никакого эффекта для организации. В этом случае причины отказа от ТС ПО должны быть также определены в терминах потребностей организации или критериев, которые остались неудовлетворенными. При этом необходимо понимать отличие этого варианта от предыдущего, связанного с недостатками конкретной ТС ПО.

Результатом данного этапа является документ, в котором обсуждаются результаты пилотного проекта и детализируются решения по внедрению.

5.3.6.

ПРАКТИЧЕСКОЕ ВНЕДРЕНИЕ ТС ПО

Процесс перехода к практическому использованию ТС ПО начинается с разработки и последующей реализации плана перехода. Этот план может отражать поэтапный подход к переходу — от тщательно выбранного пилотного проекта до проектов с существенно возросшим разнообразием характеристик.

Разработка плана перехода

План перехода должен охватывать:

- информацию относительно целей, критериев оценки, графика и возможных рисков, связанных с реализацией плана;
- информацию относительно приобретения, установки и настройки ТС ПО;
- информацию относительно интеграции с существующими средствами, включая как интеграцию средств друг с другом, так и их интеграцию в процессы разработки и эксплуатации ПО, существующие в организации;
- ожидаемые потребности в обучении и ресурсы, используемые в течение и после завершения процесса перехода;
- определение стандартных процедур использования ТС ПО.

Цели, критерии оценки, график и риски, связанные с планом перехода

Данная информация должна включать:

- типы проектов, в которых в конечном счете будет использоваться ТС ПО;
- график перехода к практическому использованию ТС ПО в отдельных проектах;
- график внедрения ТС ПО в терминах количества пользователей, включая необходимое обучение;
- возможные риски и непредвиденные обстоятельства;
- источники существующих (базовых) данных и метрики для оценки изменений, вызванных использованием ТС ПО.

В дополнение к сказанному следует уделить особое внимание вопросам контроля изменений. Роли высшего руководства, субъектов и объектов изменений должны быть уточнены по сравнению с пилотным проектом, поскольку технология подлежит широкому распространению в организации.

Подразумевается, что план перехода успешно выполнен, когда не требуется больше специального планирования поддержки использования ТС ПО. В этот момент использование ТС ПО согласуется с тем, что от нее ожидалось, и план работы с ней включается в общий план текущей поддержки ПО, существующий в организации.

Приобретение, установка и настройка ТС ПО

Приобретение, установка и настройка, выполненные в рамках пилотного проекта, могут потребоваться в более широком масштабе. При этом необходима следующая информация:

- совокупность программных компонентов и документации, которые следует приобретать для каждой отдельной платформы;
- необходимое обучение;
- механизм получения новых версий;
- настройка средств для выполнения существующих в организации процедур и соглашений;
- наличие лица или подразделения, ответственного за установку, интеграцию, настройку и эксплуатацию средств;
- план конвертирования данных и снятия старых средств с эксплуатации.

Задачи приобретения, установки и настройки должны быть как можно быстрее переданы из группы пилотного проекта в существующую службу системной поддержки ПО организации.

Интеграция средств ТС ПО с существующими средствами и процессами.

Интеграция новых средств с существующими средствами и процессами является важным шагом в полномасштабном внедрении ТС ПО. В большинстве случаев такая интеграция в процессе пилотного проектирования не осуществляется, однако накопленная при этом информация может помочь в разработке планов интеграции. Для планирования интеграции необходима следующая информация:

- наименования и версии существующих средств, с которыми должны интегрироваться новые средства;
- описания данных, которые должны совместно использоваться новыми и существующими средствами, а также предварительная информация об источниках этих данных;
- описания других взаимосвязей между новыми и существующими сред-

ствами (таких, как связи по передаче управления и порядку использования), а также предварительная информация о механизмах поддержки этих взаимосвязей;

- оценки затрат, сроков и рисков, связанных с интеграцией (и, возможно, с переходом от существующих средств и данных);
- способы внедрения данных средств в деятельность по совершенствованию существующих процессов;
- ожидаемые изменения в существующих процессах и продуктах, являющиеся следствием использования новых средств и оцениваемые по возможности количественно.

Риск, связанный с интеграцией новых средств с существующими средствами и процессами, снижается, если потребности в интеграции учитываются в процессе оценки и выбора ТС ПО.

Обучение и ресурсы, используемые в течение и после завершения процесса перехода

Данная информация должна охватывать:

- персонал (включая пользователей, администраторов и интеграторов), нуждающийся в обучении использованию ТС ПО;
- вид обучения, необходимого для каждой категории пользователей и обслуживающего персонала, с учетом особой важности обучения совместному использованию различных средств, а также методам и процессам, связанным с данными средствами;
- вид обучения, необходимого для различных специалистов (например, для группы тестирования и независимой службы сертификации);
- частоту обучения;
- виды и доступность поддержки.

Определение стандартных процедур использования средств

План перехода должен определять начальную практику применения и процедуры использования средств. Реальное применение любой ТС ПО в конкретной организации и конкретном проекте невозможно без выработки ряда стан-

дартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта (это особенно актуально при коллективной разработке ПО большим количеством групп специалистов). К таким стандартам относятся следующие:

- руководства по моделированию и проектированию;
- соглашения по присвоению имен;
- процедуры контроля качества и процессов приемки, включая расписание экспертиз и используемые методологии;
- процедуры резервного копирования, защиты мастер-копий и конфигурирования базы данных проекта;
- процедуры интеграции с существующими средствами и базами данных;
- процедуры совместного использования данных и контроля целостности базы данных;
- стандарты и процедуры обеспечения секретности; в стандарты документирования;
- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт интерфейса пользователя. Стандарт проектирования должен устанавливать:
- набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;
- правила фиксации проектных решений на диаграммах, в том числе правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм (включая требования к форме и размерам объектов) и т. д.;
- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств и т. д.;
- механизм обеспечения совместной работы над проектом, в том числе правила интеграции подсистем проекта, правила поддержания проекта в

одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т.д.), правила анализа проектных решений на непротиворечивость и т. д.

Стандарт оформления проектной документации должен устанавливать:

- комплектность, состав и структуру документации на каждой стадии проектирования (в соответствии со стандартом ГОСТ Р ИСО 9127-94 «Системы обработки информации. Документация пользователя и информация на упаковке потребительских программных пакетов»);
- требования к оформлению документации (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т.д.);
- правила подготовки, рассмотрения, согласования и утверждения документации с указанием предельных сроков для каждой стадии;
- требования к настройке издательской системы, используемой в качестве встроенного средства подготовки документации;
- требования к настройке CASE-средств для обеспечения подготовки документации в соответствии с установленными правилами.

Стандарт интерфейса пользователя должен регламентировать:

- правила оформления экранов (шрифты и цветовая палитра), состав и расположение окон и элементов управления;
- правила использования клавиатуры и мыши;
- правила оформления текстов помощи;
- перечень стандартных сообщений;
- правила обработки реакции пользователя.

Стандарты использования ТС ПО, выработанные во время пилотного проекта, должны использоваться в качестве отправной точки для разработки более полного набора стандартов использования средств в данной организации. При этом должен учитываться опыт участников пилотного проекта.

Реализация плана перехода

Реализация плана перехода требует постоянного мониторинга использова-

ния ТС ПО, обеспечения текущей поддержки, сопровождения и обновления ее средств по мере необходимости.

Периодические экспертизы

Достигнутые результаты должны периодически подвергаться экспертизе в соответствии с графиком. План перехода должен при необходимости корректироваться. Постоянное внимание должно уделяться степени удовлетворения потребностей организации и критериев успешного внедрения ТС ПО.

Периодические экспертизы должны продолжаться и после завершения процесса внедрения. Такие экспертизы могут анализировать метрики и другую информацию, получаемую в процессе работы с ТС ПО, чтобы определять, насколько хорошо она продолжает выполнять требуемые функции. Экспертизы могут также указать на необходимость дополнительной модификации процессов.

Текущая поддержка

Необходимо определить источники для текущей поддержки ТС ПО. Такая поддержка должна обеспечивать:

- получение ответов на вопросы, связанные с использованием ТС ПО;
- передачу информации о достигнутых успехах и полученных уроках другим специалистам организации;
- модификацию и совершенствование стандартов, соглашений и процедур, связанных с использованием ТС ПО;
- интеграцию новых средств с существующими и сопровождение интегрированных средств по мере появления новых версий;
- помощь новым сотрудникам в освоении ТС ПО и связанных с ней процедур;
- планирование и контроль обновления версий;
- планирование внедрения новых возможностей ТС ПО в организационные процессы.

Действия, выполняемые в процессе перехода

Для поддержки процесса перехода к практическому использованию ТС ПО

желательно выполнение следующих действий:

- поддержка текущего обучения. Потребность в обучении может возникать периодически вследствие появления новых версий ТС ПО или вовлечения в проект новых сотрудников;
- поддержка ролевых функций, связанных с процессом внедрения. Поскольку внедрение ТС ПО приводит к изменениям в культуре организации, необходимо в процессе внедрения выделить ряд ключевых ролей (руководство высшего уровня, проектная группа и целевые группы);
- управление обновлением версий. Управление может быть связано с такими вопросами, как обновление версий, процедуры установки и ответственные за установку, процедуры контроля качества для оценки новых версий, процедуры обновления базы данных, конфигурация версий и среда поддержки (другие средства, операционная система и т.д.);
- свободный доступ к информации. Должны быть определены механизмы, обеспечивающие свободный доступ к информации об опыте внедрения и извлеченных из этого уроках, включая доски объявлений, информационные бюллетени, пользовательские группы, семинары и публикации;
- налаживание тесного рабочего взаимодействия с поставщиком, позволяющего организации быть в курсе планов поставщика и обеспечивать нормальную обратную связь в соответствии со своими требованиями.

Для успешного внедрения ТС ПО в организации существенной является последовательность в ее применении. Поскольку большинство систем разрабатываются коллективно, необходимо определить характер будущего использования ТС ПО как отдельными разработчиками, так и группами. Использование стандартных процедур позволит обеспечить плавный переход между отдельными стадиями ЖЦ ПО.

Как правило, все понимают, что обучение является центральным звеном, обеспечивающим нормальное использование ТС ПО в организации. Тем не менее, довольно распространенная ошибка заключается в том, что производится начальное обучение для группы неподготовленных пользователей, а затем все

ограничивается минимальным текущим обучением. Участники пилотного проекта, получившие начальное обучение, могут быть высококвалифицированными энтузиастами новой ТС ПО, стремящимися использовать ее во что бы то ни стало. С другой стороны, для разработчиков, которые будут участвовать в проекте в дальнейшем, могут потребоваться более интенсивное и глубокое обучение, а также текущая поддержка в использовании ТС ПО.

В дополнение к этому следует отметить, что каждая категория персонала (например, администраторы средств, служба поддержки рабочих мест, интеграторы средств, служба сопровождения и разработчики приложений) нуждается в различном обучении.

Обучение не должно замыкаться только на пользователях средств, обучаться должны также те сотрудники организации, на деятельность которых оказывает влияние использование ТС ПО.

В процессе дальнейшего использования ТС ПО в организации обучение должно стать частью процесса ориентации при найме новых сотрудников и привлечении сотрудников к проектам. Обучение должно стать неотъемлемой составной частью нормативных материалов, касающихся деятельности организации, которые предлагаются новым сотрудникам.

Одна общая ошибка, которая делается в процессе перехода, заключается в недооценке ресурсов, необходимых для поддержки постоянного использования сложных ТС ПО. Рост необходимых ресурсов вызывается тремя причинами:

- сложностью ТС ПО;
- частотой появления новых версий;
- взаимодействием между ТС ПО и внешней средой.

Сложность ТС ПО приводит к возрастанию потребностей в тщательном и продуманном обучении. Кроме того, многие ТС ПО нуждаются в опытных специалистах, умеющих сопровождать проектные базы данных и оперативно реагировать на возникающие проблемы. Частота обновления версий ТС ПО может привести к возникновению нетривиальных проблем, которые зачастую упускаются из виду. Такие обновления обычно пагубно отражаются на жестких

планах и графиках работы. Взаимодействие между средствами и внешней по отношению к ним средой также может иногда порождать некоторые проблемы, так как, хотя многие средства достигли уровня минимальной несовместимости данных между отдельными версиями, проблемы обеспечения совместимости с другими элементами внешней среды остаются в силе.

Оценка результатов перехода

Программа постоянной оценки качества и продуктивности ПО преследует следующие цели:

- определение степени совершенствования процессов;
- упреждение возможных стратегических просчетов;
- своевременный отказ от использования устаревшей ТС ПО.

Чтобы определить, насколько эффективно новая ТС ПО повышает продуктивность и (или) качество, организация должна опираться на некоторые базовые данные. К сожалению, лишь немногие организации в настоящее время накапливают данные для реализации программы текущей количественной оценки и усовершенствования процессов. Для доказательства эффективности ТС ПО и ее возможностей улучшить продуктивность необходимы следующие базовые метрические данные:

- использованное время;
- время, выделенное персонально для конкретных специалистов;
- размер, сложность и качество ПО;
- удобство сопровождения.

Метрическая оценка должна начинаться с реальной оценки текущего состояния среды до начала внедрения ТС ПО и поддерживать процедуры постоянного накопления данных.

Период времени, в течение которого выполняется количественная оценка воздействия, оказываемого внедрением ТС ПО, является весьма значимой величиной относительно определения степени успешности перехода. Некоторые организации, успешно внедрившие в конечном счете ТС ПО, столкнулись с кратковременными негативными эффектами в начале процесса. Другие, успеш-

но начав, недооценили долговременные затраты на сопровождение и обучение. Вследствие этого наиболее приемлемый временной интервал для оценки степени успешности внедрения должен быть достаточно большим, чтобы преодолеть любые негативные эффекты на начальном этапе, а также смоделировать будущие долговременные затраты. С другой стороны, данный интервал должен соответствовать целям организации и ожидаемым результатам.

В конечном счете опыт, полученный при внедрении ТС ПО, может отчасти изменить цели организации и ожидания, возлагаемые на ТС ПО. Например, организация может сделать вывод, что средства целесообразно использовать для большего или меньшего круга пользователей и процессов в цикле создания и сопровождения ПО. Такие изменения в ожиданиях зачастую могут дать положительные результаты, но могут также привести к внесению соответствующих корректив в определение степени успешного внедрения ТС ПО в данной организации.

Результатом данного этапа является внедрение ТС ПО в повседневную практику организации, при этом больше не требуется какого-либо специального планирования. Кроме того, поддержка ТС ПО включается в план текущей поддержки ПО в данной организации.

Дополнительную информацию по материалу данного подраздела можно найти в американских стандартах IEEE Std 1348-1995. IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools и IEEE Std 1209-1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools (IEEE — Institute of Electrical and Electronics Engineers — Институт Инженеров по электротехнике и электронике). Временной разрыв между их утверждением составляет четыре года (первый стандарт был утвержден в декабре 1996 г., а второй — в декабре 1992 г.), однако они достаточно тесно взаимосвязаны, поскольку первый стандарт содержит целый ряд ссылок на второй (помимо упомянутых стандартов, существует также упомянутый выше международный стандарт ISO/IEC 14102:1995(E). Information technology — Guideline for the evaluation and selection of CASE Tools, основные положения которого во

многим совпадают с положениями IEEE Std 1209—1992). Цель приведенных в стандартах рекомендаций — предоставить руководство, позволяющее повысить вероятность успешного внедрения ТС ПО. Эти рекомендации достаточно актуальны и ценны, поскольку отражают опыт, накопленный многими зарубежными пользователями и разработчиками в течение длительного периода.

5.4.

ПРИМЕРЫ ТС ПО

Многие организации-разработчики программных продуктов год за годом накапливали профессиональные знания в области ТС ПО, которые материализовались в виде практических рекомендаций, документации, обучающих программ и книг. Поскольку документация и книги быстро устаревают, со временем эти технологии стали приобретать электронную форму, превращаясь, таким образом, в программный продукт. Как правило, они поставляются вместе с CASE-средствами и включают библиотеки процессов, шаблонов, методов, моделей и других компонентов, предназначенных для построения ПО того класса систем, на который ориентирована технология. Электронные технологии включают также средства, которые должны обеспечивать их адаптацию для конкретных пользователей и развитие по результатам выполнения конкретных проектов.

На сегодняшний день практически все ведущие компании — разработчики технологий и программных продуктов (IBM, Microsoft, Oracle, Borland, Computer Associates, Sybase и др.) располагают развитыми технологиями создания ПО, которые создавались как собственными силами, так и за счет приобретения продуктов и технологий, созданных небольшими специализированными компаниями. Выбор в качестве примера четырех перечисленных ниже технологий объясняется их ведущими позициями на мировом рынке ТС ПО и присутствием на российском рынке.

ТЕХНОЛОГИЯ RUP (RATIONAL UNIFIED PROCESS)

Одна из наиболее совершенных технологий, претендующих на роль мирового корпоративного стандарта — Rational Unified Process. RUP представляет собой программный продукт, разработанный компанией Rational Software (www.rational.com), которая в настоящее время входит в состав IBM. RUP является результатом объединения подходов Rational Approach и Objectory Process, происшедшего после слияния в 1995 году компаний Rational Software и Objectory AB (созданной Иваром Якобсоном).

RUP в значительной степени соответствует стандартам и нормативным документам, связанным с процессами ЖЦ ПО и оценкой технологической зрелости организаций-разработчиков (ISO 12207, ISO 9000, CMM и др.). Ее основными принципами являются:

1. Итерационный и инкрементный (наращиваемый) подход к созданию ПО.
2. Планирование и управление проектом на основе функциональных требований к системе — вариантов использования.
3. Построение системы на базе архитектуры ПО.

Первый принцип является определяющим. В соответствии с ним разработка системы выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности (от 2 до 6 недель), называемых итерациями. Каждая итерация включает свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием работающей системы.

Итерационный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру системы. Система постоянно разрастается шаг за шагом, поэтому такой подход называют *итерационным и инкрементным*.

При таком подходе исключено и слишком быстрое написание кода (без детальной проработки) и чрезмерно длительный этап детального проектирования и построения моделей без обратной связи.

На рис. 5.6 показано общее представление RUP в двух измерениях:

1. горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как стадии, итерации и контрольные точки;
2. вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как виды деятельности (технологические операции), рабочие продукты, исполнители и дисциплины (технологические процессы).

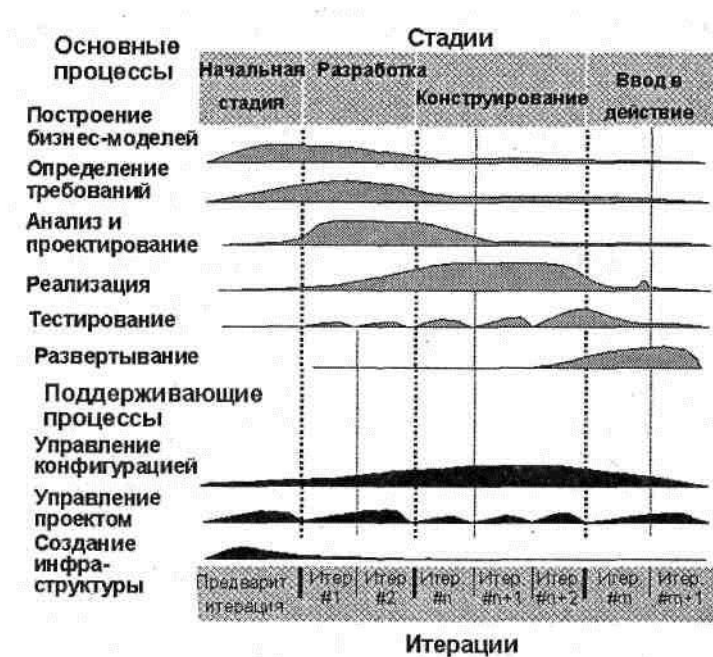


Рис. 5.6. Общее представление RUP

Динамический аспект

Согласно технологии RUP ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

- начальная стадия (inception);
- стадия разработки (elaboration);

- стадия конструирования (construction);
- стадия ввода в действие (transition).

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

Первыми двумя стадиями являются начальная стадия проекта и разработка. **Начальная стадия** может принимать множество разных форм. Для крупных проектов начальная стадия может вылиться во всестороннее изучение всех возможностей реализации проекта, которое займет месяцы. Во время начальной стадии вырабатывается бизнес-план проекта — определяется, сколько приблизительно он будет стоить и какой доход принесет. Определяются также границы проекта, и выполняется некоторый начальный анализ для оценки размеров проекта.

Для того чтобы выполнить такую работу, необходимо идентифицировать все внешние сущности (действующие лица), с которыми система будет взаимодействовать, и определить в самом общем виде природу этого взаимодействия. Это подразумевает идентификацию всех вариантов использования и описание наиболее важных из них. Бизнес-план включает критерии успеха, оценку риска, оценку необходимых ресурсов и общий план по стадиям, включающий даты основных контрольных точек.

Результатами начальной стадии являются:

- общее описание системы: основные требования к проекту, его характеристики и ограничения;
- начальная модель вариантов использования (степень готовности - 10-20%);
- начальный проектный глоссарий (словарь терминов);
- начальный бизнес-план;
- план проекта, отражающий стадии и итерации;
- один или несколько прототипов.

На стадии **разработки** выявляются более детальные требования к системе,

выполняется высокоуровневый анализ предметной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Результатами стадии разработки являются:

- модель вариантов использования (завершенная по крайней мере на 80%), определяющая функциональные требования к системе;
- перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;
- описание базовой архитектуры будущей системы;
- работающий прототип;
- уточненный бизнес-план;
- план разработки всего проекта, отражающий итерации и критерии оценки для каждой итерации.

Самым важным результатом стадии разработки является описание базовой архитектуры будущей системы. Эта архитектура включает:

- модель предметной области, которая отражает понимание бизнеса и служит отправным пунктом для формирования основных классов предметной области;
- технологическую платформу, определяющую основные элементы технологии реализации системы и их взаимодействие.

Эта архитектура является основой всей дальнейшей разработки, она служит своего рода проектом для последующих стадий. В дальнейшем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

Стадия разработки занимает около пятой части общей продолжительности проекта. Основными признаками завершения стадии разработки являются два события:

- разработчики в состоянии оценить с достаточно высокой точностью, сколько времени потребуется на реализацию каждого варианта использо-

вания;

- идентифицированы все наиболее серьезные риски, и степень понимания наиболее важных из них такова, что известно, как справиться с ними.

Сущность планирования заключается в определении последовательности итераций конструирования и вариантов использования, реализуемых на каждой итерации.

Планирование завершается, когда определены место каждого варианта использования в некоторой итерации и дата начала каждой итерации. На данном этапе более детальное планирование не делается.

RUP представляет собой итерационный и пошаговый процесс разработки, в котором программное обеспечение разрабатывается и реализуется по частям. На стадии *конструирования* построение системы выполняется путем серии итераций. Каждая итерация является своего рода мини-проектом. На каждой итерации для конкретных вариантов использования выполняются анализ, проектирование, кодирование, тестирование и интеграция. Итерация завершается демонстрацией результатов пользователям и выполнением системных тестов с целью контроля корректности реализации вариантов использования.

Назначением этого процесса является снижение степени риска. Причиной появления риска часто является откладывание решения сложных проблем на самый конец проекта. Тестирование и интеграция — это достаточно крупные задачи, они всегда занимают больше времени, чем ожидается. Чем позже выполнять тестирование и интеграцию, тем более трудными задачами они становятся и тем более способны дезорганизовать весь проект. При итеративной разработке на каждой итерации выполняется весь процесс, что позволяет оперативно справляться со всеми возникающими проблемами.

Итерации на стадии конструирования являются одновременно инкрементными и повторяющимися:

- итерации являются инкрементными в соответствии с той функцией, которую они выполняют. Каждая итерация добавляет очередные конструкции к вариантам использования, реализованным во время предыдущих итера-

ций;

- итерации являются повторяющимися по отношению к разрабатываемому коду. На каждой итерации некоторая часть существующего кода перепиывается с целью сделать, его более гибким.

Процесс интеграции должен быть непрерывным. В конце каждой итерации должна выполняться полная интеграция. С другой стороны, интеграция может и должна выполняться еще чаще. Приложения следует интегрировать после выполнения каждой сколько-нибудь значительной части работы. Во время каждой интеграции должен выполняться полный набор тестов.

Главная особенность итерационной разработки заключается в том, что она жестко ограничена временными рамками, и сдвигать сроки недопустимо. Исключением может быть перенос реализации каких-либо вариантов использования на более позднюю итерацию по соглашению с заказчиком. Смысл таких ограничений — поддерживать строгую дисциплину разработки и не допускать переноса сроков.

При этом если на более поздний срок перенесено слишком много вариантов использования, то необходимо корректировать план, пересмотрев при этом оценку трудоемкости реализации вариантов использования. На данной стадии разработчики должны иметь более глубокое представление о такой оценке.

Помимо конструирования, итерации могут присутствовать во всех стадиях, однако конструирование - это ключевая стадия. Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям. Как минимум, он содержит следующее:

- ПО, интегрированное на требуемых платформах;
- руководства пользователя;
- описание текущей реализации.

Назначением стадии *ввода в действие* является передача готового продукта в распоряжение пользователей. Данная стадия включает:

- бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;

- параллельное функционирование с существующей (legacy) системой, которая подлежит постепенной замене;
- конвертирование баз данных;
- оптимизацию производительности;
- обучение пользователей и специалистов службы сопровождения.

Главная идея итерационной разработки — поставить весь процесс разработки на регулярную основу с тем, чтобы команда разработчиков смогла получить конечный продукт. Однако есть некоторые вещи, которые не следует выполнять слишком рано, например оптимизация.

Оптимизация снижает прозрачность и расширяемость системы, однако повышает ее производительность. В этой ситуации необходимо принятие компромиссного решения, поскольку система должна быть достаточно производительной, чтобы удовлетворять пользовательским требованиям. Слишком ранняя оптимизация затруднит последующую разработку, поэтому ее следует выполнять в последнюю очередь.

На стадии ввода в действие продукт не дополняется никакой новой функциональностью (кроме самой минимальной и абсолютно необходимой). Здесь только вылавливаются ошибки. Хорошим примером для стадии ввода в действие может служить период времени между выпуском бета-версии и окончательной версии продукта.

Статический аспект

Статический аспект RUP представлен четырьмя основными элементами:

- роли;
- виды деятельности;
- рабочие продукты;
- дисциплины.

Понятие «роль» (role) определяет поведение и ответственность личности или группы личностей, составляющих проектную команду. Одна личность может играть в проекте много различных ролей.

Под видом деятельности конкретного исполнителя понимается единица

выполняемой им работы. **Вид деятельности** (activity) соответствует понятию технологической операции. Он имеет четко определенную цель, обычно выражаемую в терминах получения Или модификации некоторых **рабочих продуктов** (artifacts), таких, как модель, элемент модели, документ, исходный код или план. Каждый вид деятельности связан с конкретной ролью. Продолжительность вида деятельности составляет от нескольких часов до нескольких дней, он обычно выполняется одним исполнителем и порождает только один или весьма небольшое количество рабочих продуктов. Любой вид деятельности должен являться элементом процесса планирования. Примерами видов деятельности могут быть планирование итерации, определение вариантов использования и действующих лиц, выполнение теста на производительность. Каждый вид деятельности сопровождается набором **руководств** (guidelines), представляющих собой методики выполнения технологических операций.

Дисциплина (discipline) соответствует понятию технологического процесса и представляет собой последовательность действий, приводящую к получению значимого результата.

В рамках RUP определены шесть основных дисциплин:

- построение бизнес-моделей;
- определение требований;
- анализ и проектирование;
- реализация;
- тестирование; *
- развертывание;
- и три вспомогательных:
- управление конфигурацией и изменениями;
- управление проектом;
- создание инфраструктуры.

Основное содержание дисциплин построения бизнес-моделей, определения требований, анализа и проектирования было изложено в главах 3 и 4.

RUP как продукт входит в состав комплекса Rational Suite, причем каждая

из перечисленных выше дисциплин поддерживается определенным инструментальным средством комплекса. Физическая реализация RUP представляет собой Web-сайт (рис. 5.7), включающий следующие компоненты:

- описание всех элементов динамического и статического аспекта RUP;
- навигатор по всем элементам RUP, глоссарий и средство быстрого обучения технологии;
- руководства для всех участников проектной команды, охватывающие весь жизненный цикл ПО. Руководства представлены в двух видах: для осмысления процесса на верхнем уровне и в виде подробных инструкций по отдельным видам деятельности;
- руководства по использованию инструментальных средств, входящих в состав Rational Suite;
- примеры и шаблоны проектных решений для Rational Rose;
- шаблоны проектной документации для SoDa;
- шаблоны в формате Microsoft Word, предназначенные для поддержки документации по всем процессам и действиям жизненного цикла ПО;
- планы в формате Microsoft Project, отражающие итерационный характер разработки ПО.

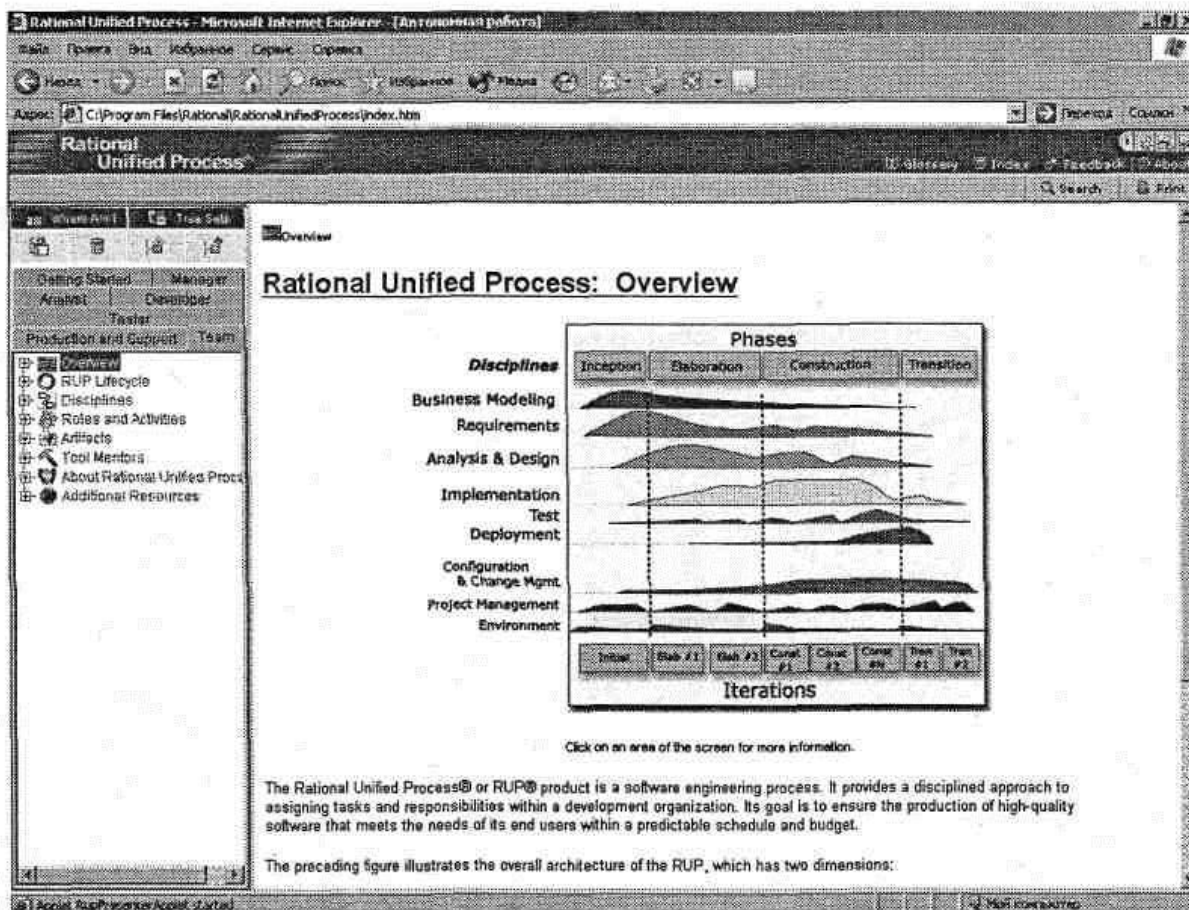


Рис. 5.7. Электронная версия RUP

Адаптация RUP к потребностям конкретной организации или проекта обеспечивается с помощью Rational Process Workbench (RPW) - специального набора инструментов и шаблонов для настройки и публикации Web-сайтов на основе RUP. RPW поддерживает три основные функции моделирования технологических процессов:

- определение процесса;
- описание процесса;
- представление процесса.

Основой для определения процесса является модель RUP в среде CASE-средства Rational Rose, подобная модели на рис. 5.1.

Библиотека элементов процесса содержит текстовую информацию о каждом элементе в модели процесса, все текстовые страницы RUP, а RPW - необходимые шаблоны для создания новых страниц описания. RPW генерирует

описание процессов, включающее текст и графику, в виде Web-сайта, соединяя модели процессов и библиотеку описаний в единое целое.

Компания Rational Software поддерживает портал Rational Developer Network (www.rational.net), содержащий:

- статьи по инструментальным средствам и RUP;
- обзоры книг;
- курсы по обучению специалистов;
- инструкции по эксплуатации инструментальных средств;
- архивы дополнений и расширений к инструментальным средствам;
- базу знаний, основанную на реальных проектах.

RUP опирается на интегрированный комплекс инструментальных средств Rational Suite. Он существует в вариантах:

- Rational Suite AnalystStudio - предназначен для определения и управления полным набором требований к разрабатываемой системе;
- Rational Suite DevelopmentStudio - предназначен для проектирования и реализации ПО;
- Rational Suite TestStudio — набор продуктов, предназначенных для автоматического тестирования приложений;
- Rational Suite Enterprise — обеспечивает поддержку полного жизненного цикла ПО и предназначен как для менеджеров проекта, так и отдельных разработчиков, выполняющих несколько функциональных ролей в команде разработчиков.

В состав Rational Suite, кроме самой технологии RUP как продукта, входят следующие компоненты:

- Rational Rose — средство визуального моделирования (анализа и проектирования), использующее язык UML;
- Rational XDE — средство анализа и проектирования, интегрируемое с платформами MS Visual Studio .NET и IBM WebSphere Studio Application Developer;

- Rational Requisite Pro — средство управления требованиями, предназначенное для организации совместной работы группы разработчиков. Оно позволяет команде разработчиков создавать, структурировать, устанавливать приоритеты, отслеживать, контролировать изменения требований, возникающих на любом этапе разработки компонентов приложения;
- Rational Rapid Developer — средство быстрой разработки приложений на платформе Java 2 Enterprise Edition;
- Rational ClearCase — средство управления конфигурацией ПО;
- Rational SoDA — средство автоматической генерации проектной документации;
- Rational ClearQuest — средство для управления изменениями и отслеживания дефектов в проекте на основе средств e-mail и Web;
- Rational Quantify — средство количественного определения узких мест, влияющих на общую эффективность работы программы;
- Rational Purify — средство для локализации трудно обнаруживаемых ошибок времени выполнения программы;
- Rational PureCoverage — средство идентификации участков кода, пропущенных при тестировании;
- Rational TestManager — средство планирования функционального и нагрузочного тестирования;
- Rational Robot — средство записи и воспроизведения тестовых сценариев;
- Rational TestFactory — средство тестирования надежности;
- Rational Quality Architect — средство генерации кода для тестирования.

Одно из основных инструментальных средств комплекса Rational Rose представляет собой семейство объектно-ориентированных CASE-средств; оно предназначено для автоматизации процессов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Rational Rose реализует процесс объектно-ориентированного анализа и проектирования ПО, описанный в RUR В основе работы Rational Rose лежит

построение диаграмм и спецификаций UML, определяющих архитектуру системы, ее статические и динамические аспекты. В составе Rational Rose можно выделить шесть основных структурных компонентов: репозиторий, графический интерфейс пользователя, средства просмотра проекта (браузер), средства контроля проекта, средства сбора статистики и генератор документов. К ним добавляются генераторы кодов для каждого поддерживаемого языка, состав которых меняется от версии к версии.

Репозиторий представляет собой базу данных проекта. Браузер обеспечивает «навигацию» по проекту, в том числе перемещение по иерархиям классов и подсистем, переключение от одного вида диаграмм к другому и т. д. Средства контроля и сбора статистики дают возможность находить и устранять ошибки по мере развития проекта, а не после завершения его описания. Генератор отчетов формирует тексты выходных документов на основе содержащейся в репозиторий информации.

Средства автоматической генерации кода, используя информацию, содержащуюся в диаграммах классов и компонентов, формируют файлы описаний классов. Создаваемый таким образом скелет программы может быть уточнен путем прямого программирования на соответствующем языке (основные языки, поддерживаемые Rational Rose — C++ и Java).

В результате разработки проекта с помощью Rational Rose формируются следующие документы:

- диаграммы UML, в совокупности представляющие собой модель разрабатываемой программной системы;
- спецификации классов, объектов, атрибутов и операций;
- заготовки текстов программ.

Тексты программ являются заготовками для последующей работы программистов. Состав информации, включаемой в программные файлы, определяется либо по умолчанию, либо по усмотрению пользователя. В дальнейшем эти исходные тексты развиваются программистами в полноценные программы.

Инструментальное средство Rational XDE представляет собой развитие

возможностей Rational Rose в части синхронизации модели и кода (исключающей необходимость прямой и обратной генерации кода). Rational XDE обеспечивает:

- синхронизацию между кодом и моделью;
- отображение элементов кода Java и C# в UML;
- автоматическую синхронизацию с настраиваемым разрешением конфликтов;
- одновременное отображение кода и модели;
- постоянную синхронизацию модели UML как части проекта Java или C#.

5.4.2.

ТЕХНОЛОГИЯ ORACLE

Методическую основу ТС ПО корпорации Oracle (www.oracle.com) составляет метод Oracle (Oracle Method) - комплекс методов, охватывающий большинство процессов ЖЦ ПО. В состав комплекса входят:

- CDM (Custom Development Method) — разработка прикладного ПО;
- PJM (Project Management Method) — управление проектом;
- AIM (Application Implementation Method) — внедрение прикладного ПО;
- BPR (Business Process Reengineering) — реинжиниринг бизнес-процессов;
- OCM (Organizational Change Management) — управление изменениями, и др.

Метод CDM оформлен в виде консалтингового продукта CDM Advantage - библиотеки стандартов и руководств (включающего также PJM). Он представляет собой развитие достаточно давно созданного Oracle CASE-Method, известного по использованию CASE-средств фирмы Oracle и книгам Ричарда Баркера. По существу CDM является методическим руководством по разработке прикладного ПО с использованием инструментального комплекса Oracle Developer Suite, а сам процесс проектирования и разработки тесно связан с Oracle Designer

и Oracle Forms.

В соответствии с CDM ЖЦ ПО формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов (рис. 5.8):

- стратегия (определение требований);
- анализ (формулирование детальных требований к системе);
- проектирование (преобразование требований в детальные спецификации системы);
- реализация (написание и тестирование приложений);
- внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);
- эксплуатация.

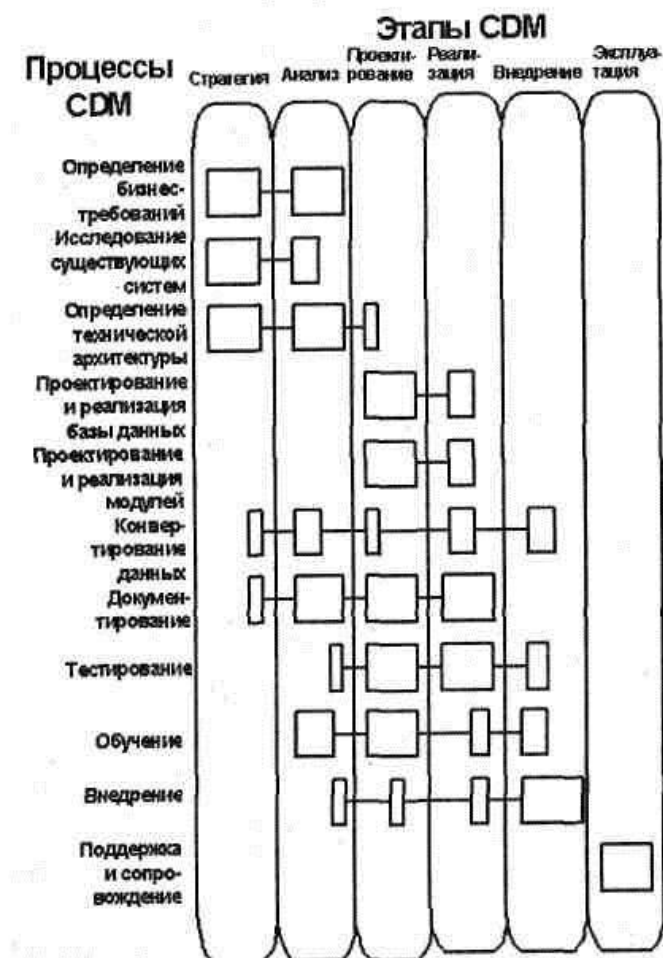


Рис. 5.8. Этапы и процессы CDM

На этапе стратегии определяются цели создания системы, приоритеты и

ограничения, разрабатывается системная архитектура и составляется план разработки. На этапе анализа строятся модель информационных потребностей (диаграмма «сущность-связь»), диаграмма функциональной иерархии (на основе функциональной декомпозиции системы), матрица перекрестных ссылок и диаграмма потоков данных.

На этапе проектирования разрабатывается подробная архитектура системы, проектируются схема реляционной БД и программные модули, устанавливаются перекрестные ссылки между компонентами системы для анализа их взаимного влияния и контроля за изменениями.

На этапе реализации создается БД, строятся прикладные системы, производится их тестирование, проверка качества и соответствия требованиям пользователей. Создается системная документация, материалы для обучения и руководства пользователей.

На этапах внедрения и эксплуатации анализируются производительность и целостность системы, выполняется поддержка и, при необходимости, модификация системы.

Процессы CDM:

- определение бизнес-требований, или постановка задачи (Business Requirements Definition);
- исследование существующих систем (Existing Systems Examination). Выполнение этого процесса должно обеспечить понимание состояния существующего технического и программного обеспечения для планирования необходимых изменений;
- определение технической архитектуры (Technical Architecture);
- проектирование и реализация базы данных (Database Design and Build). Процесс предусматривает проектирование и реализацию реляционной базы данных, включая создание индексов и других объектов БД;
- проектирование и реализация модулей (Module Design and Build). Этот процесс является основным в проекте. Он включает непосредственное проектирование приложения и создание кода прикладной программы;

- конвертирование данных (Data Conversion). Цель этого процесса — преобразовывать, перенести и проверить согласованность и непротиворечивость данных, оставшихся в наследство от «старой» системы и необходимых для работы в новой системе;
- документирование (Documentation);
- тестирование (Testing);
- обучение (Training);
- внедрение, или переход к новой системе (Transition). Этот процесс включает решение задач установки, ввода новой системы в эксплуатацию, прекращения эксплуатации старых систем;
- поддержка и сопровождение (Post-System Support).

Процессы состоят из последовательностей взаимосвязанных задач.

CDM предоставляет возможность выбрать требуемый подход к разработке. Это возможно, поскольку каждый процесс базируется на известных зависимостях между задачами одного типа и не зависит от того, на какие этапы будет разбит проект.

При определении подхода к разработке оценивается масштаб, степень сложности и критичность будущей системы. При этом учитываются стабильность требований, сложность и количество бизнес-правил, количество автоматически выполняемых функций, разнообразие и количество пользователей, степень взаимодействия с другими системами, критичность приложения для основного бизнес-процесса компании и целый ряд других.

В соответствии с этими факторами в CDM выделяются два основных подхода к разработке:

Классический подход (Classic). Этапы данного подхода представлены на рис 5.7. Классический подход применяется для наиболее сложных и масштабных проектов, он предусматривает последовательный и детерминированный порядок выполнения задач. Для таких проектов характерно большое количество реализуемых бизнес-правил, распределенная архитектура, критичность приложения. Применение классического подхода также рекомендуется при нехватке

опыта у разработчиков, неподготовленности пользователей, нечетко определенной задаче. Продолжительность таких проектов от 8 до 36 месяцев.

Подход быстрой разработки (Fast Track). Данный подход, в отличие от каскадного классического, является итерационным и основан на методе DSDM (Dynamic Systems Development Method). В этом подходе четыре этапа — стратегия, моделирование требований, проектирование и генерация системы и внедрение в эксплуатацию. Подход используется для реализации небольших и средних проектов с несложной архитектурой системы, гибкими сроками и четкой постановкой задач. Продолжительность проекта от 4 до 16 месяцев.

RJM— это определенная дисциплина ведения проекта, позволяющая гарантировать, что цели проекта, четко определенные в его начале, остаются в центре внимания на протяжении всего проекта. В основе RJM лежит метод, ориентированный на выполнение самостоятельных процессов (под процессом понимается набор связанных задач, выполнением которых достигается определенная цель проекта). Так же, как и CDM, метод руководства проектом представляется в виде четко определенной операционной схемы, в которой выделяются процессы, этапы, задачи, результаты решения задач и зависимости между задачами:

- Управление проектом и предоставление отчетности (Control and Reporting). Этот процесс содержит задачи, в результате решения которых определяются границы проекта и подход к разработке, происходит управление изменениями и контролируется возможный риск;
- Управление работой (Work Management). Процесс содержит задачи, помогающие контролировать работы, выполняемые в проекте;
- Управление ресурсами (Resource Management). Здесь решаются задачи, связанные с обеспечением каждого этапа исполнителями;
- Управление качеством (Quality Management). Процесс управления качеством гарантирует, что проект отвечает требованиям пользователя в течение всего процесса разработки;
- Управление конфигурацией (Configuration Management).

Цикл решения задач PJM состоит из отдельных этапов. Количество этапов зависит от выбранного подхода к разработке. Задачи PJM можно распределить внутри каждого процесса по трем группам — задачи планирования, управления и завершения, и по уровням — отнести задачу на уровень проекта или на уровень отдельного этапа.

По аналогии с CDM, в PJM предусмотрено широкое использование шаблонов разрабатываемых документов.

Комплекс Oracle Developer Suite содержит набор интегрированных средств разработки для быстрого создания приложений. Он включает средства моделирования, программирования на Java, разработки компонентов, бизнес-анализа и составления отчетов. Все эти средства используют общие ресурсы, что позволяет совместно работать над одним проектом группе разработчиков. Oracle Developer Suite интегрирован с Oracle Database и Oracle Application Server, образуя единую платформу для создания и установки приложений.

Oracle Developer Suite поддерживает стандарты J2EE: Enterprise Java Beans (EJB), сервлеты и страницы JavaServer (JSP). В него также входят анализатор XML, процессор XSLT, процессор схем XML и XSQL-сервлет для разработки XML-приложений.

В Oracle Developer Suite встроена поддержка языка UML для разработки приложений на основе моделей. Модели хранятся в общем репозитории Oracle, который предназначен для поддержки больших коллективов разработчиков.

Oracle Developer Suite включает в себя:

- Oracle Designer — средство моделирования и генерации приложений;
- Oracle Forms — средство быстрой разработки приложений;
- Oracle Reports - визуальное средство разработки отчетов;
- Oracle JDeveloper — средство визуального программирования на языке Java;
- Oracle Discoverer — средство для разработки аналитических приложений;
- Oracle Warehouse Builder - система для построения хранилищ данных;
- Oracle Portal — средство разработки информационного портала организа-

ции.

CASE-средство Oracle Designer является интегрированным средством, обеспечивающим в совокупности со средствами разработки приложений поддержку ЖЦ ПО.

Oracle Designer представляет собой семейство методов и поддерживающих их программных продуктов. Базовый метод Oracle Designer (CDM) — структурный метод проектирования систем, охватывающий полностью все стадии ЖЦ ПО. Версия Oracle Designer для объектно-реляционной СУБД Oracle содержит также расширение в виде средств объектного моделирования, базирующихся на стандарте UML.

Oracle Designer обеспечивает графический интерфейс при разработке различных моделей (диаграмм) предметной области. В процессе построения моделей информация о них заносится в репозитории. В состав Oracle Designer входят следующие компоненты:

- Repository Administrator — средства управления репозитори-ем (создание и удаление приложений, управление доступом к данным со стороны различных пользователей, экспорт и импорт данных);
- Repository Object Navigator — средство доступа к репозиторию, обеспечивающее многооконный объектно-ориентированный интерфейс доступа ко всем элементам репозитория;
- Process Modeler - средство анализа и моделирования бизнес-процессов;
- Systems Modeler — набор средств построения функциональных и информационных моделей проектируемой системы, включающий средства для построения диаграмм «сущность-связь» (Entity-Relationship Diagrammer), диаграмм функциональных иерархий (Function Hierarchy Diagrammer), диаграмм потоков данных (Data Flow Diagrammer) и средство анализа и модификации связей объектов репозитория различных типов (Matrix Diagrammer);
- Systems Designer — набор средств проектирования ПО, включающий средство построения структуры реляционной базы данных (Data

Diagrammer), а также средства построения диаграмм, отображающих взаимодействие с данными, иерархию, структуру и логику приложений, реализуемую хранимыми процедурами на языке PL/SQL (Module Data Diagrammer, Module Structure Diagrammer и Module Logic Navigator);

- Server Generator - генератор описаний объектов БД Oracle (таблиц, индексов, ключей, последовательностей и т.д.);
- Forms Generator — генератор приложений для Oracle Forms. Генерируемые приложения включают в себя различные экраны, ранние формы, средства контроля данных, проверки Ограничений целостности и автоматические подсказки;
- Repository Reports — генератор стандартных отчетов, интегрированный с Oracle Reports.

Репозиторий Oracle Designer представляет собой хранилище всех проектных данных и может работать в многопользовательском режиме, обеспечивая параллельное обновление информации несколькими разработчиками. В процессе проектирования автоматически поддерживаются перекрестные ссылки между объектами словаря и могут генерироваться более 70 стандартных отчетов о моделируемой предметной области. Физическая среда хранения репозитория — база данных Oracle.

5.4.3.

ТЕХНОЛОГИЯ BORLAND

Компания Borland (www.borland.com) в результате развития собственных разработок и приобретения ряда компаний представила интегрированный комплекс инструментальных средств, реализующих управление полным жизненным циклом приложений (Application Life Cycle Management, ALM). В соответствии с технологией Borland процесс создания ПО включает в себя пять основных этапов:

- определение требований;

- анализ и проектирование;
- разработка;
- тестирование и профилирование;
- развертывание.

Выполнение всех этапов координируется процессом управления конфигурацией и изменениями.

Определение требований реализуется с помощью системы управления требованиями CaliberRM, которая стала частью семейства продуктов Borland в результате покупки компании Starbase. CaliberRM сохраняет требования в базе данных, документы с их описанием создаются с помощью встроенного механизма генерации документов MS Word на базе заданных шаблонов. Система обеспечивает экспорт данных в таблицы MS Access и импорт из MS Word. CaliberRM поддерживает различные методы визуализации зависимостей между требованиями, с помощью которых пользователь может ограничить область анализа, необходимого в случае изменения того или иного требования. Имеется модуль, который использует данные требования для оценки трудозатрат, рисков и расходов, связанных с реализацией требований.

Средство анализа и проектирования Together ControlCenter разработано компанией TogetherSoft. В основе его применения лежит один из вариантов подхода «Быстрой разработки ПО» под названием Feature Driven Development (FDD)¹ [Палмер-02].

Together ControlCenter — интегрированная среда проектирования и разработки, поддерживающая визуальное моделирование на UML с последующим написанием приложений для платформ J2EE (Java) и .Net (C#, C++ и Visual Basic). Кроме базовой версии, имеется уменьшенный вариант системы для индивидуальных разработчиков и небольших групп (Together Solo), а также редакции для платформы IBM WebSphere и среды разработки Jbuilder.

¹ Палмер СР., Фелсинг Дж.М. Практическое руководство по функционально-ориентированной разработке ПО: Пер. с англ. - М.: Вильямс, 2002;

В системе реализована технология LiveSource, которая обеспечивает синхронизацию между проектом приложения и изменениями — при внесении изменений в исходные тексты меняется модель программы, а при изменении модели надлежащим образом изменяется текст на языке программирования. Это исключает необходимость вручную модифицировать модель или переписывать код. Контроль версий осуществляется благодаря функциональной интеграции Together и системы StarTeam. Поддерживается также интеграция с системой управления конфигурацией Rational ClearCase.

Инструментальные средства тестирования появились в составе комплекса Borland в результате покупки компании Optimizeit. К ним относятся Optimizeit Suite 5, Optimizeit Profiler for .NET и Optimizeit ServerTrace. Первые две системы позволяют выявить потенциальные проблемы использования аппаратных ресурсов — памяти и процессорных мощностей на платформах J2EE и .Net соответственно. Интеграция Optimizeit Suite 5 в среду разработки Jbuilder, а Optimizeit Profiler — в C#Builder и Visual Basic .Net позволяет проводить контрольные испытания приложений по мере разработки и ликвидировать узкие места производительности. Система Optimizeit ServerTrace предназначена для управления производительностью серверных J2EE-приложений с точки зрения достижения заданного уровня обслуживания и сбора контрольных данных по виртуальным Java-машинам.

Сущность концепции ALM сосредоточена в системе управления конфигурацией и изменениями: именно она объединяет основные фазы ЖЦ ПО. Такой системой является StarTeam, разработанная компанией Starbase. Она выполняет функции контроля версий, управления изменениями, отслеживания дефектов, управления требованиями (в интеграции с CaliberRM), управления потоком задач и управления проектом.

StarTeam совместима с интерфейсом Microsoft Source Code Control и интегрируется с любой системой разработки, которая поддерживает этот API. Кроме того, в системе реализованы средства интеграции со средствами разработки и моделирования Together, JBuilder, Delphi, C++Builder и C#Builder.

В технологии Borland выделяются три уровня интеграции. **Функциональная (touch-point) интеграция** позволяет обратиться из одной системы к функциям другой, выбрав соответствующий пункт меню. Например, интерфейс управления изменениями StarTeam непосредственно отображается в системах Together, C#Builder и Visual Studio .Net. Такая интеграция дает возможность разделять информацию между системами, но не обеспечивает единого рабочего пространства, вынуждает пользователя переключать окна и приводит к дублированию процессов управления структурой проекта. **Встроенная (embedded) интеграция** обеспечивает работу с одной системой непосредственно в среде другой. Например, не выходя из среды разработки Jbuilder, можно просматривать графики производительности, которые создает система Optimizeit. Самый высокий уровень интеграции — **синергетический (synergistic)**, позволяющий сочетать функции двух различных продуктов незаметно для разработчиков. Для большинства продуктов Borland и других поставщиков синергетическая интеграция пока остается делом будущего, однако ее принципы уже начинают реализовываться.

5.4.4.

ТЕХНОЛОГИЯ COMPUTER ASSOCIATES

Компания Computer Associates (www.ca.com) предлагает комплексы инструментальных средств поддержки различных процессов ЖЦ ПО:

AllFusion Modeling Suite — интегрированный комплекс CASE-средств¹, включающий следующие продукты:

- AllFusion Process Modeler (BPwin) — функциональное моделирование;
- AllFusion ERwin Data Modeler (ERwin) — моделирование данных;
- AllFusion Component Modeler (Paradigm Plus) — объектно-, ориентиро-

¹ Маклаков СВ. Создание информационных систем с AllFusion Modeling Suite. - М.: Диалог-МИФИ, 2003.

ванный анализ и проектирование с использованием UML и возможностью генерации кода;

- AllFusion Model Manager (Model Mart) — организация совместной работы команды разработчиков;
- AllFusion Data Model Validator (ERwin Examiner) — проверка структуры и качества моделей данных.

AllFusion Change Management Suite — комплекс средств управления конфигурацией и изменениями.

AllFusion Process Management Suite — средства управления процессами и проектами для различных типов приложений.

CASE-средства ERwin и BPwin были разработаны фирмой Logic Works, которая в 1998 г. вошла в состав PLATINUM Technology, а затем Computer Associates.

BPwin — средство моделирования бизнес-процессов, реализующее метод IDEF0, а также поддерживающее диаграммы потоков данных и IDEF3. В процессе моделирования BPwin позволяет переключиться с нотации IDEF0 на любой ветви модели на нотацию IDEF3 или DFD и создать смешанную модель. BPwin поддерживает функционально-стоимостной анализ (ABC).

Семейство продуктов ERwin представляет собой набор средств концептуального моделирования данных, использующих метод IDEF1X. ERwin реализует проектирование схемы БД, генерацию ее описания на языке целевой СУБД (Oracle, Sybase, DB2, Microsoft SQL Server и др.) и реверсный инжиниринг существующей БД. ERwin выпускается в нескольких конфигурациях, ориентированных на наиболее распространенные средства разработки приложений.

Для управления групповой разработкой используется средство Model Mart, обеспечивающее многопользовательский доступ к моделям, созданным с помощью ERwin и BPwin. Модели хранятся на центральном сервере и доступны для всех участников группы проектирования.

Model Mart удовлетворяет ряду требований, предъявляемых к средствам управления разработкой крупных систем, а именно:

- Совместное моделирование. Каждый участник проекта имеет инструмент поиска и доступа к интересующей его модели в любое время. При совместной работе используются три режима: незащищенный, защищенный и режим просмотра. В режиме просмотра запрещается любое изменение моделей. В защищенном режиме модель, с которой работает один пользователь, не может быть изменена другими пользователями. В незащищенном режиме пользователи могут работать с общими моделями в реальном масштабе времени.
- Создание библиотек решений. Model Mart позволяет формировать библиотеки стандартных решений, включающие наиболее удачные фрагменты реализованных проектов, накапливать и использовать типовые модели, объединяя их при необходимости «сборки» больших систем. На основе существующих баз данных с помощью ERwin возможно восстановление моделей (реверсный инжиниринг), которые в процессе анализа пригодности их для новой системы могут объединяться с типовыми моделями из библиотек моделей.
- Управление доступом. Для каждого участника проекта определяются права доступа, в соответствии с которыми он получит возможность работать только с определенными моделями. Права доступа могут быть определены как для групп, так и для отдельных участников проекта. Роль специалистов, участвующих в различных проектах может меняться, поэтому в Model Mart можно определять и управлять правами доступа участников проекта к библиотекам, моделям и даже к специфическим областям модели.

! Следует запомнить

1. Основным требованием, предъявляемым к современным ТС ПО, является их соответствие стандартам и нормативным документам, связанным с процессами ЖЦ ПО и оценкой технологической зрелости организаций-разработчиков (ISO 12207, ISO 9000, CMM и др.).

2. Процесс внедрения ТС ПО охватывает планирование и реализацию множества технических, организационных, структурных процессов, изменений в общей культуре организации и основан на четком понимании возможностей ТС ПО.

Основные понятия

Технология создания ПО, технологический процесс, технологическая операция, рабочий продукт, роль, руководство, инструментальное средство (CASE-средство).

? Вопросы для самоконтроля

1. Охарактеризуйте систему понятий, описывающих ТС ПО. Какие понятия являются наиболее важными?
2. Какие из требований, предъявляемых к современным ТС ПО, представляются наиболее важными и почему?
3. Поясните смысл реалистичных и нереалистичных ожиданий от внедрения ТС ПО.
4. Какие из критериев, приведенных в табл. 5.1, представляются наиболее значимыми?
5. Охарактеризуйте принципы и область применения методики анализа и проектирования Rational Unified Process.

ОЦЕНКА ТРУДОЕМКОСТИ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой оценка трудоемкости создания ПО.*
- *Какие существуют методы оценки и в чем они заключаются.*
- *Какие факторы и каким образом влияют на трудоемкость создания ПО.*

Оценка трудоемкости создания ПО является одним из наиболее важных видов деятельности в процессе создания ПО, хотя она и не выделена в стандарте ISO 12207 как отдельный процесс (тем не менее, в новую версию стандарта предполагается включить процесс «Измерение» (measurement)).

Модели и методы оценки трудоемкости используются для решения многих задач, среди которых можно выделить следующие:

- разработка бюджета проекта (здесь прежде всего требуется точность общей оценки);
- анализ степени риска и выбор компромиссного решения (решение данной задачи позволяет уточнить такие характеристики проекта, как масштабы, возможность повторного использования, количество разработчиков, используемое оборудование и т. д.);
- планирование и управление проектом (полученные результаты обеспечивают распределение и классификацию расходов по компонентам, этапам и операциям);
- анализ затрат на улучшение качества ПО (это позволяет оценить затраты и прибыль от стратегии инвестирования в совершенствование аппаратных средств, технологий и возможности повторного использования).

При отсутствии адекватной и достоверной оценки невозможно обеспечить

четкое планирование и управление проектом. В целом ситуация в данной области, сложившаяся в индустрии информационных технологий, выглядит далеко не блестящей.

Недооценка стоимости, времени и ресурсов, требуемых для создания ПО, влечет за собой недостаточную численность проектной команды, чрезмерно сжатые сроки разработки и, как результат, утрату доверия к разработчикам в случае нарушения графика. С другой стороны, перестраховка и переоценка могут оказаться ничуть не лучше. Если для проекта выделено больше ресурсов, чем реально необходимо, причем без должного контроля за их использованием, то ни о какой экономии ресурсов говорить не приходится. Такой проект окажется более дорогостоящим, чем должен был быть при грамотной оценке, и приведет к запаздыванию с началом следующего проекта.

6.1.

МЕТОДЫ ОЦЕНКИ И ИХ КЛАССИФИКАЦИЯ

1. **Алгоритмическое моделирование.** Метод основан на анализе статистических данных о ранее выполненных проектах, при этом определяется зависимость трудоемкости проекта от какого-нибудь количественного показателя программного продукта (обычно это размер программного кода). Проводится оценка этого показателя для данного проекта, после чего с помощью модели прогнозируются будущие затраты.

2. **Экспертные оценки.** Проводится опрос нескольких экспертов по технологии разработки ПО, знающих область применения создаваемого программного продукта. Каждый из них дает свою оценку трудоемкости проекта. Потом все оценки сравниваются и обсуждаются. Этот процесс повторяется до тех пор, пока не будет достигнуто согласие по окончательному варианту предварительной трудоемкости.

3. **Оценка по аналогии.** Этот метод используется в том случае, если в данной области применения создаваемого ПО уже реализованы аналогичные про-

екты. Метод основан на сравнении планируемого проекта с предыдущими проектами, имеющими подобные характеристики. Он использует экспертные данные или сохраненные данные о проекте. Эксперты вычисляют высокую, низкую и наиболее вероятную оценку трудоемкости, основываясь на различиях между новым и предыдущими проектами. Оценка может быть достаточно детальной в зависимости от глубины аналогий. Слабость модели заключается в том, что степень подобия нового проекта и предыдущих, как правило, не слишком велика.

Самый лучший вариант — это использование накопленных в организации исторических данных, позволяющих сопоставить трудоемкость вашего проекта с трудоемкостью предыдущих проектов аналогичного размера. Однако это возможно только при следующих условиях:

- в организации аккуратно документируются реальные результаты предыдущих проектов;
- по крайней мере, один из предыдущих проектов (а лучше несколько) имеет аналогичный характер и размер;
- жизненный цикл, используемые методы и средства разработки, квалификация и опыт проектной команды нового проекта также подобны тем, которые имели место в предыдущих проектах.

4. **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 работают пять человек, должен быть закончен в течение 12 месяцев, то затраты на его выполнение исчисляются в 60 человеко-месяцев.

5. **Оценка с целью выиграть контракт.** Затраты на проект определяются наличием тех средств, которые имеются у заказчика. Поэтому трудоемкость проекта зависит от бюджета заказчика, а не от функциональных характеристик создаваемого продукта. Требования приходится изменить так, чтобы не выходить за рамки принятого бюджета.

Каждый метод оценки имеет слабые и сильные стороны. Для работы над большими проектами необходимо применить несколько методов оценки 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 оценку, и так до тех пор, пока результаты разных методов не станут достаточно близкими.

Описанные методы оценки применимы, если документированы требования будущей системы. В таком случае существует возможность определить функциональные характеристики разрабатываемой системы. Однако во многих проектах оценка затрат проводится только на основе предварительных требований к системе. В этом случае лица, участвующие в оценке стоимости проекта, будут иметь минимум информации для работы. Процедуры анализа требований и создания спецификаций весьма дорогостоящи. Поэтому менеджерам следует составить смету на их выполнение еще до утверждения бюджета для всего проекта.

Практика в этой области такова, что независимые оценки (т.е. выполненные людьми, которые никак не зависят от команды разработчиков) обычно неточны. Единственный способ, позволяющий получить заслуживающую доверия оценку, — это когда компетентная команда — менеджер проекта вместе с ведущими специалистами по созданию архитектуры, разработке и тестированию — выполняют несколько итераций по оценке трудоемкости и анализу чувствительности модели. Для того чтобы проект мог быть успешно выполнен, эта команда должна затем признать свое авторство произведенной оценки трудоемкости.

Хорошая оценка трудоемкости разработки ПО:

- создается и поддерживается менеджером проекта и командами архитекторов, разработчиков и тестировщиков, ответственными за выполнение работы;
- воспринимается всеми исполнителями как амбициозная, но выполняемая;

- основывается на подробно описанной и обоснованной модели оценки;
- основывается на данных по аналогичным проектам, которые включают в себя аналогичные процессы, технологии, среду, требования к качеству и квалификации работников;
- подробно описывается таким образом, чтобы все ключевые области риска были хорошо видны, а вероятность успеха оценивалась объективно.

Идеальную оценку можно получить путем экстраполяции хорошей оценки, полученной на основе устоявшейся модели трудоемкости и использующей опыт выполнения множества аналогичных проектов, подготовленных той же командой, которая использовала те же зрелые процессы и инструментарий. Хотя такая ситуация на практике встречается редко, когда команда приступает к осуществлению нового проекта, хорошие оценки могут быть получены обычным путем на более поздних этапах жизненного цикла проекта.

В алгоритмическом моделировании трудоемкости разработки ПО существует в основном два подхода к моделированию: теоретические модели и статистические модели, которые будут рассмотрены ниже. Большинство моделей для определения трудоемкости разработки ПО могут быть сведены к функции пяти основных параметров:

- размера конечного продукта (для компонентов, написанных вручную), который обычно измеряется числом строк исходного кода или количеством функциональных точек, необходимых для реализации данной функциональности;
- особенностей процесса, используемого для получения конечного продукта, в частности его способность избегать непроизводительных видов деятельности (переделок, бюрократических проволочек, затрат на взаимодействие);
- возможностей персонала, участвующего в разработке ПО, в особенности его профессионального опыта и знания предметной области проекта;
- среды, которая состоит из инструментов и методов, используемых для эффективного выполнения разработки ПО и автоматизации процесса;

- требуемого качества продукта, включающего в себя его функциональные возможности, производительность, надежность и адаптируемость.

Соотношение между этими параметрами, с одной стороны, и рассчитываемой трудоемкостью с другой, может быть записано следующим образом:

$$\text{Трудоемкость} = (\text{Персонал}) \cdot (\text{Среда}) \cdot (\text{Качество}) \cdot (\text{Размер}^{\text{объём}}).$$

Наиболее влиятельный фактор оценки трудоемкости в этих моделях — размер программного продукта. Процедура оценки трудоемкости разработки ПО состоит из следующих действий:

1. оценка размера разрабатываемого продукта;
2. оценка трудоемкости в человеко-месяцах или человеко-часах;
3. оценка продолжительности проекта в календарных месяцах;
4. оценка стоимости проекта.

Оценка размера продукта базируется на знании требований к системе. Для такой оценки существуют два основных способа.

- По аналогии. Если в прошлом приходилось иметь дело с подобным проектом, и его оценки известны, то можно, отталкиваясь от них, приблизительно оценить свой проект.
- Путем подсчета размера по определенным алгоритмам на основе исходных данных — требований к системе.
- Проблемы оценки размера ПО.
- Проблема может быть недостаточно хорошо понята разработчиками и (или) заказчиками из-за того, что некоторые факты были упущены или искажены.
- Недостаток или полное отсутствие исторических данных не позволяет создать базу для оценок в будущем.
- Проектирующая организация не располагает стандартами, с помощью которых можно выполнять процесс оценивания (либо в случае наличия стандартов их никто не придерживается); в результате наблюдается недо-

статок совместимости при выполнении процесса оценивания.

- Менеджеры проектов полагают, что было бы неплохо фиксировать требования в начале проекта, заказчики же считают, что не стоит тратить время на разработку спецификации требований.
- Ошибки, как правило, скрываются, вместо того, чтобы оцениваться и отображаться, в результате чего создается ложное впечатление о фактической производительности.
- Возможности оценивания существенно зависят от субъектов, вовлеченных в процесс оценивания.
- Менеджеры, аналитики, разработчики, тестеры и те, кто внедряет продукт, могут иметь разные представления о процессах оценивания и о возможностях совершенствования продукта.

Точность оценки стоимости и размера ПО в зависимости от стадии проекта определяется следующим графиком (рис. 6.1). Основными единицами измерения размера ПО являются:

- количество строк кода (LOC - Lines of Code);
- функциональные точки (FP — Function Points).

Количество строк кода — исторически самая известная и до недавнего времени распространенная единица измерения. Однако при ее использовании возникает ряд вопросов:

- Каким образом можно определить количество строк кода до того, как они фактически будут написаны либо просто спроектированы?
- Как показатель количества строк кода может отражать величину трудозатрат, если не будет учитываться сложность продукта, способности (стиль) программиста либо возможности применяемого языка программирования?
- Каким образом разница в количестве строк кода может быть трансформирована в объем эквивалентной работы?

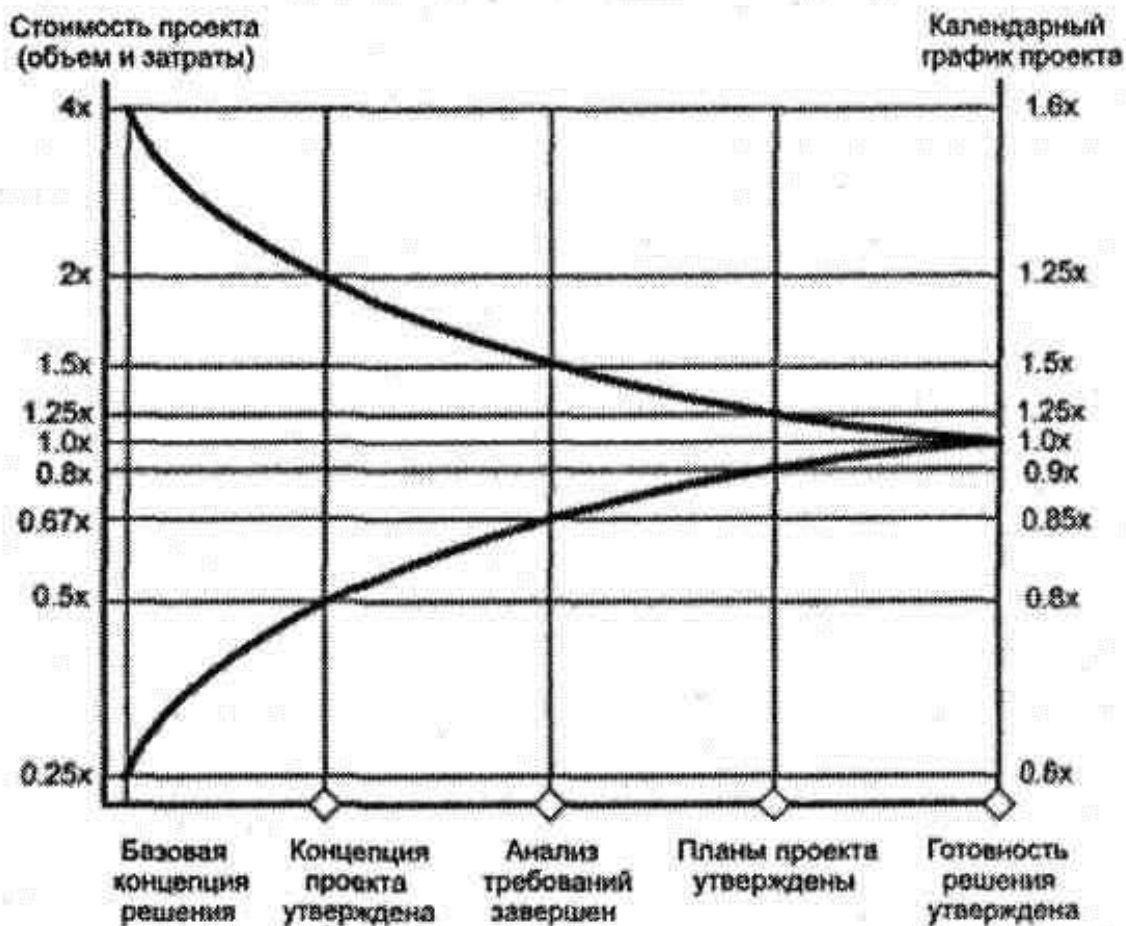


Рис. 6.1. Точность оценки стоимости и размера ПО в зависимости от стадии проекта

Эти и другие вопросы привели к тому, что строки кода как единицы измерения получили «дурную репутацию», хотя они по-прежнему остаются наиболее широко используемыми. Взаимосвязь между LOC и затрачиваемыми усилиями не является линейной. Несмотря на появление новых языков программирования, средняя производительность работы программистов за двадцать лет осталась неизменной и составляет около 3000 строк кода на одного программиста в год. Это говорит о том, что уменьшение времени, затрачиваемого на цикл разработки, не может быть достигнуто за счет значительного повышения производительности труда программистов. Причем это не зависит от усовершенствований языка программирования, усилий со стороны менеджеров или сверхурочных работ. На самом деле первостепенное значение имеет набор функциональных свойств и качество ПО, а не количество строк кода.

Преимущества использования LOC в качестве единиц измерения:

- широкое распространение и легкая адаптируемость;
- возможность сопоставления методов измерения размеров и производительности в различных группах разработчиков;
- непосредственная связь с конечным продуктом;
- легкая оценка до завершения проекта;
- оценка размеров ПО на основе точки зрения разработчика — физическая оценка созданного продукта (количество написанных строк кода).
- Недостатки применения LOC:
- LOC затруднительны в применении при оценке размера ПО на ранних стадиях разработки;
- строки исходного кода могут различаться в зависимости от типов языков программирования, методов проектирования, стиля и способностей программиста;
- применение методов оценки с помощью подсчета количества строк кода не регламентируется промышленными стандартами (например, ISO);
- разработка ПО может быть связана с большими затратами, которые прямо не зависят от размеров программного кода — «фиксированными затратами», такими, как спецификации требований и пользовательские документы, не включенными в прямые затраты на кодирование;
- программисты могут быть незаслуженно премированы за достижение высоких показателей LOC в случае, если руководство по ошибке считает это признаком высокой продуктивности, но при этом будет отсутствовать тщательно разработанный проект; исходный код не является самоцелью при создании продукта — главную роль играют функциональные свойства и показатели производительности;
- при подсчете количества LOC следует различать автоматически и вручную созданный код - эта задача является более сложной, чем простой подсчет, который может быть выполнен на основе листинга, сгенериро-

ванного компилятором, либо с помощью утилиты, выполняющей подсчет строк кода;

- показатели LOC не могут применяться при осуществлении нормализации в случае, если применяемые платформы разработки или языки являются различными;
- единственный способ учета с помощью LOC по отношению к разрабатываемому ПО заключается в использовании метода аналогии на основе сравнения функциональных свойств у подобных программных продуктов, либо в использовании мнений экспертов (однако эти методы не относятся к числу точных);
- генераторы кода зачастую продуцируют чрезмерный объем кода, в результате чего искажаются показатели LOC.

Результатом этих соображений явилось осознание необходимости другой единицы измерения, в качестве которой стали выступать функциональные точки.

6.2.

МЕТОДИКА ОЦЕНКИ ТРУДОЕМКОСТИ

РАЗРАБОТКИ ПО НА ОСНОВЕ

ФУНКЦИОНАЛЬНЫХ ТОЧЕК

Определение числа функциональных точек является методом количественной оценки ПО, применяемым для измерения функциональных характеристик процессов его разработки и сопровождения независимо от технологии, использованной для его реализации.

Подсчет функциональных точек, помимо средства для объективной оценки ресурсов, необходимых для разработки и сопровождения ПО, применяется также в качестве средства для определения сложности приобретаемого продук-

та с целью принятия решения о покупке или собственной разработке.

Метод разработан на основе опыта реализации множества проектов создания ПО и поддерживается международной организацией IFPUG (International Function Point User Group). Рассматриваемый в данном разделе сокращенный вариант методики оценки трудоемкости разработки ПО основан на материалах IFPUG и компании SPR (Software Productivity Research), которая является одним из лидеров в области методов и средств оценки характеристик ПО.

Согласно данной методике трудоемкость вычисляется на основе функциональности разрабатываемой системы, которая, в свою очередь, определяется на основе выявления **функциональных типов** — логических групп взаимосвязанных данных, используемых и поддерживаемых приложением, а также элементарных процессов, связанных с вводом и выводом информации (рис. 6.2).

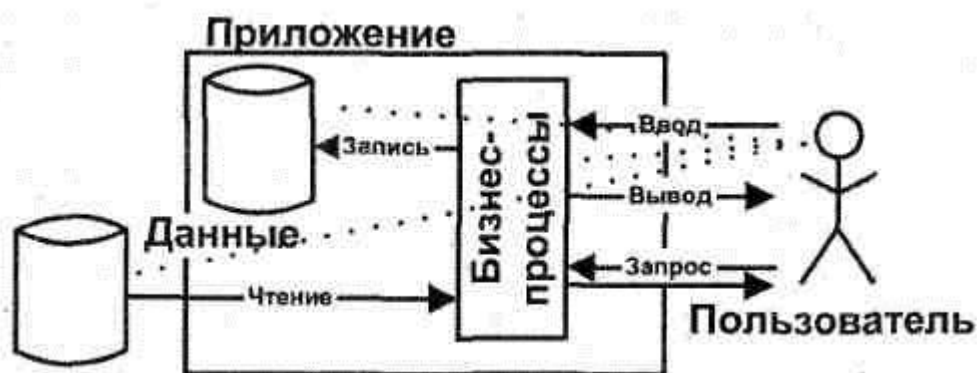


Рис. 6.2. Выявление функциональных типов

Порядок расчета трудоемкости разработки ПО:

- определение количества и сложности функциональных типов приложения;
- определение количества связанных с каждым функциональным типом элементарных данных (DET), элементарных записей (RET) и файлов типа ссылок (FTR);
- определение сложности (в зависимости от количества DET, RET и FTR);
- подсчет количества функциональных точек приложения;
- подсчет количества функциональных точек с учетом общих характери-

стик системы (рис.6.3);

- оценка трудоемкости разработки (с использованием различных статистических данных).

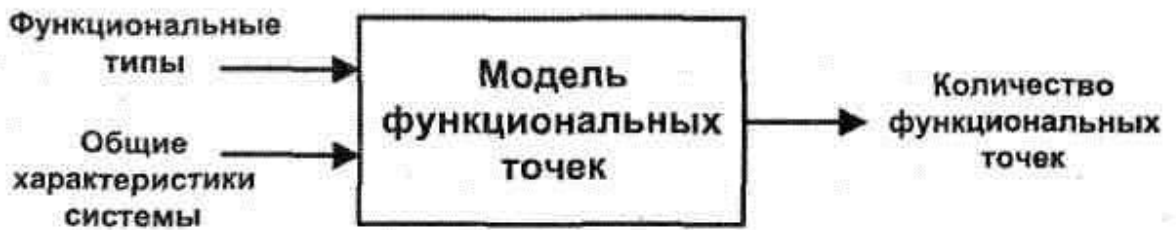


Рис. 6.3. Определение количества функциональных точек

6.2.1.

ОПРЕДЕЛЕНИЕ ФУНКЦИОНАЛЬНЫХ ТИПОВ

В состав функциональных типов (function type) включаются следующие элементы приложений разрабатываемой системы.

1. *Внутренний логический файл (internal logical file, ILF)* — идентифицируемая совокупность логически взаимосвязанных записей данных, поддерживаемая внутри приложения посредством элементарного процесса (рис. 6.4).

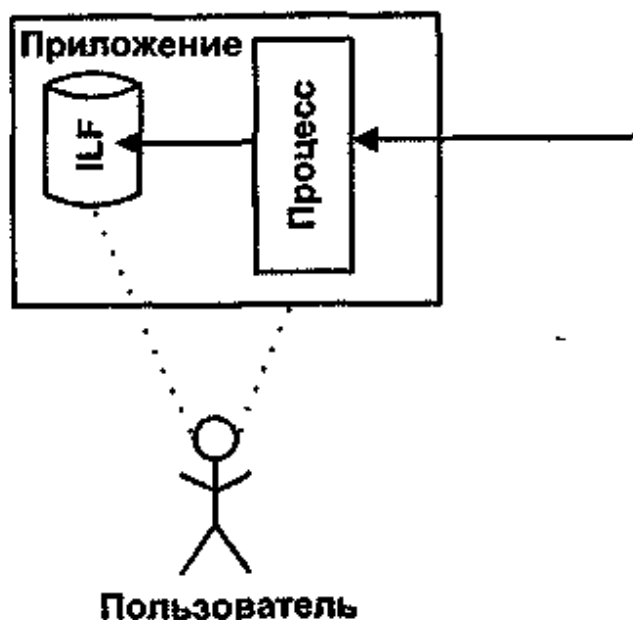


Рис. 6.4. Внутренний логический файл

2. **Внешний интерфейсный файл (external interface file, EIF)** — идентифицируемая совокупность логически взаимосвязанных записей данных, передаваемых другому приложению или получаемых от него и поддерживаемых вне данного приложения (рис. 6.5).

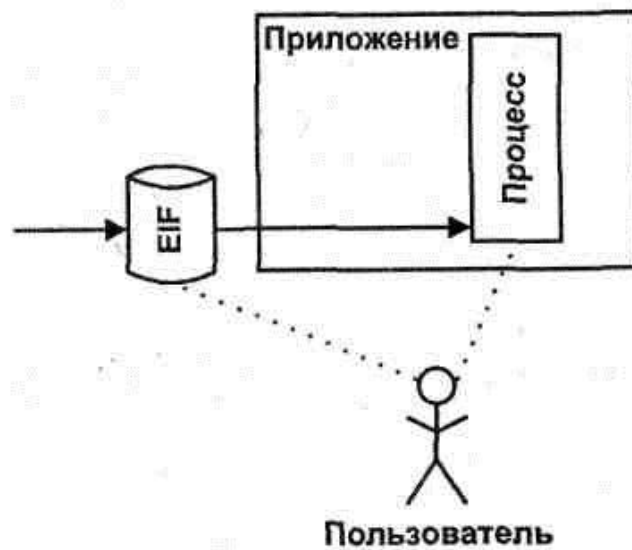


Рис. 6.5. Внешний интерфейсный файл

3. **Входной элемент приложения (external input, EI)** — элементарный процесс, связанный с обработкой входной информации приложения - входного документа или экранной формы. Обработываемые данные могут соответствовать одному или более ILF (рис. 6.6).



Рис. 6.6. Входной элемент приложения

4. **Выходной элемент приложения (external output, EO)** — элементарный процесс, связанный с обработкой выходной информации приложения — выходного отчета, документа, экранной формы (рис. 6.7).

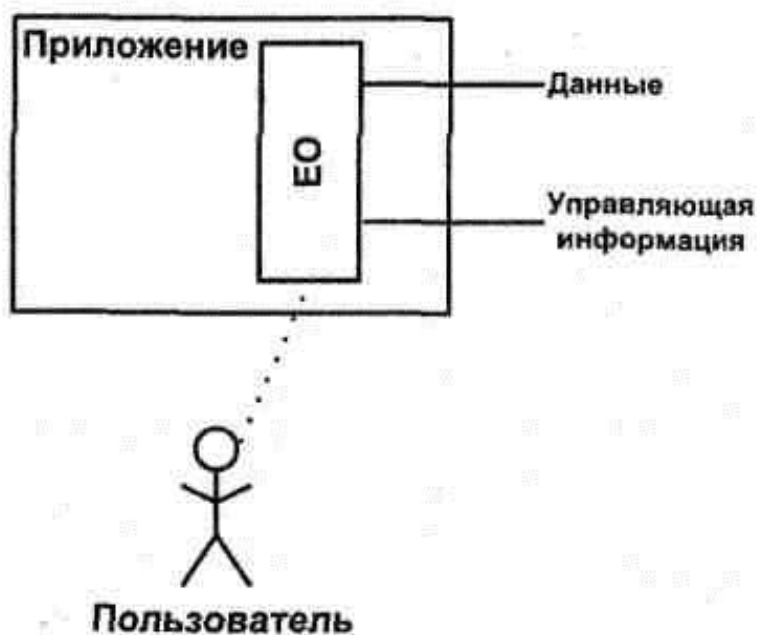


Рис. 6.7. Выходной элемент приложения

5. **Внешний запрос (external query, EQ)** — элементарный процесс, состоящий из комбинации «запрос/ответ», не связанный с вычислением производных данных или обновлением ILF (базы данных) (рис. 6.8).

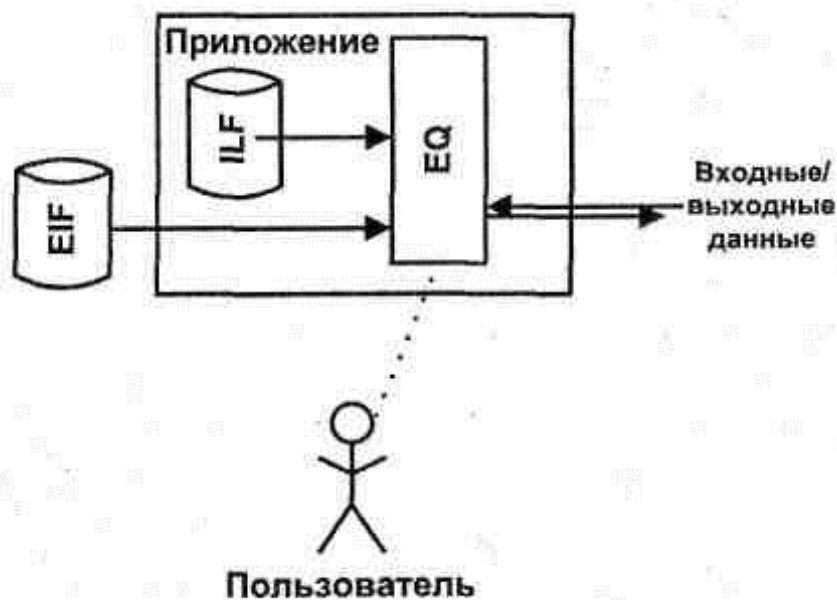


Рис. 6.8. Внешний запрос

ОПРЕДЕЛЕНИЕ КОЛИЧЕСТВА И СЛОЖНОСТИ ФУНКЦИОНАЛЬНЫХ ТИПОВ ПО ДАННЫМ

Количество функциональных типов по данным (внутренних логических файлов и внешних интерфейсных файлов) определяется на основе диаграмм «сущность-связь» (для структурного подхода) и диаграмм классов (для объектно-ориентированного подхода). В последнем случае в расчете участвуют только устойчивые (persistent) классы, или классы-сущности.

Устойчивый класс соответствует ILF (если его объекты обязательно создаются внутри самого приложения) или EIF (если его объекты не создаются внутри самого приложения, а получаются в результате запросов к базе данных).

Примечание. Если операции класса являются операциями-запросами, то это характеризует его принадлежность к EIF.

Для каждого выявленного функционального типа (ILF и EIF) определяется его сложность (низкая, средняя или высокая). Она зависит от количества связанных с этим функциональным типом элементарных данных (data element types, DET) и элементарных записей (record element types, RET), которые в свою очередь определяются следующим образом:

- DET — уникальный идентифицируемый нерекурсивный элемент данных (включая внешние ключи), входящий в ILF или EIF;
- RET — идентифицируемая подгруппа элементов данных, входящая в ILF или EIF. На диаграммах «сущность-связь» такая подгруппа обычно представляется в виде сущности-подтипа в связи «супертип-подтип».

Один DET соответствует отдельному атрибуту или связи класса. Количество DET не зависит от количества объектов класса или количества связанных объектов. Если данный класс связан с некоторым другим классом, который обладает явно заданным идентификатором, состоящим более чем из одного атрибута, то для каждого такого атрибута определяется один отдельный DET (а не

один DET на всю связь). Производные атрибуты могут игнорироваться. Повторяющиеся атрибуты одинакового формата рассматриваются как один DET

Одна RET на диаграмме устойчивых классов соответствует либо абстрактному классу в связи обобщения (generalization), либо классу — «части целого» в композиции, либо классу с рекурсивной связью «родитель-потомок» (агрегацией).

Зависимость сложности функциональных типов от количества DET и RET определяется следующей таблицей (табл. 6.1).

Таблица 6.1

Сложность ILF и EIF

Количество RET	Количество DET		
	1-19	20-50	51 +
1	Низкая	Низкая	Средняя
2-5	Низкая	Средняя	Высокая
6 +	Средняя	Высокая	Высокая

6.2.3.

ОПРЕДЕЛЕНИЕ КОЛИЧЕСТВА И СЛОЖНОСТИ

ТРАНЗАКЦИОННЫХ ФУНКЦИОНАЛЬНЫХ ТИПОВ

Количество транзакционных функциональных типов (входных элементов приложения, выходных элементов приложения и внешних запросов) определяется на основе выявления входных и выходных документов, экранных форм, отчетов, а также по диаграммам классов (в расчете участвуют граничные классы).

Далее для каждого выявленного функционального типа (EI, EO или EQ) определяется его сложность (низкая, средняя или высокая). Она зависит от количества связанных с этим функциональным типом DET, RET и файлов типа ссылок (file type referenced, FTR) - ILF или EIF, читаемых или модифицируемых

функциональным типом.

Правила расчета DET для EI:

- каждое нерекурсивное поле, принадлежащее (поддерживаемое) ILF и обрабатываемое во вводе;
- каждое поле, которое пользователь хотя и не вызывает, но оно через процесс ввода поддерживается в ILF;
- логическое поле, которое физически представляет собой множество полей, но воспринимается пользователем как единый блок информации;
- группа полей, которые появляются в ILF более одного раза, но в связи с особенностями алгоритма их использования воспринимаются как один DET;
- группа полей, которые фиксируют ошибки в процессе обработки или подтверждают, что обработка закончилась успешно;
- действие, которое может быть выполнено во вводе.

Правила расчета DET для EO:

- каждое распознаваемое пользователем нерекурсивное поле, участвующее в процессе вывода;
- поле, которое физически отображается в виде нескольких полей его составляющих, но используемое как единый информационный элемент;
- каждый тип метки и каждое значение числового эквивалента при графическом выводе;
- текстовая информация, которая может содержать одно слово, предложение или фразу;
- литералы не могут считаться элементами данных;
- переменные, определяющие номера страниц или генерируемые системой логотипы не являются элементами данных.

Таблица 6.2

Сложность EI

Количество FTR	Количество DET		
	1-4	5-15	16 +
0-1	Низкая	Низкая	Средняя
2	Низкая	Средняя	Высокая
3 +	Средняя	Высокая	Высокая

Таблица 6.3

Сложность EO

Количество FTR	Количество DET		
	1-5	6-19	20 +
0-1	Низкая	Низкая	Средняя
2-3	Низкая	Средняя	Высокая
4 +	Средняя	Высокая	Высокая

Сложность EQ определяется как максимальная из сложностей EI и EO, связанных с данным запросом.

6.2.4.**ПОДСЧЕТ КОЛИЧЕСТВА ФУНКЦИОНАЛЬНЫХ ТОЧЕК**

Для каждого функционального типа подсчитывается количество входящих в его состав функциональных точек (function point, FP) — условных элементарных единиц. Этот подсчет выполняется в соответствии с табл. 6.4.

Таблица 6.4

Зависимость количества FP от сложности функционального типа

Функциональный тип	Сложность		
	Низкая	Средняя	Высокая
ILF	7	10	15
EIF	5	7	10
EI	3	4	6
EO	4	5	7
EQ	3	4	6

В результате суммирования количества FP по всем функциональным типам получается общее количество FP (UFP, Unadjusted Function Points) без учета поправочного коэффициента. Значение поправочного коэффициента (VAF, Value Adjustment Factor) определяется набором из 14 общих характеристик системы (GSC, General System Characteristics) и вычисляется по следующей формуле:

$$VAF = (0,65 + (sumGSC * 0,01)).$$

Значения GSC варьируются в диапазоне от 0 до 5 и определяются по данным, приведенным ниже.

Коммуникации данных

0. Полностью пакетная обработка на локальном ПК
1. Пакетная обработка, удаленный ввод данных или удаленная печать
2. Пакетная обработка, удаленный ввод данных и удаленная печать
3. Сбор данных в режиме «онлайн» или дистанционная обработка, связанная с пакетным процессом
4. Несколько внешних интерфейсов, один тип коммуникационного протокола
5. Несколько внешних интерфейсов, более одного типа коммуникационного протокола

Распределенная обработка данных

0. Передача данных или процессов между компонентами системы отсутствует
1. Приложение готовит данные для обработки на ПК конечного пользователя
2. Данные готовятся для передачи, затем передаются и обрабатываются на другом компоненте системы (не на ПК конечного пользователя)
3. Распределенная обработка и передача данных в режиме «онлайн» только в одном направлении
4. Распределенная обработка и передача данных в режиме «онлайн» в обоих направлениях

5. Динамическое выполнение процессов в любом подходящем компоненте системы

Производительность

0. К системе не предъявляются специальных требований, касающихся производительности
1. Требования к производительности определены, но не требуется никаких специальных действий
2. Время реакции или пропускная способность являются критическими в пиковые периоды. Не требуется никаких специальных решений относительно использования ресурсов процессора. Обработка может быть завершена в течение следующего рабочего дня
3. Время реакции или пропускная способность являются критическими в обычное рабочее время. Не требуется никаких специальных решений относительно использования ресурсов процессора. Время обработки ограничено взаимодействующими системами
4. То же, кроме того, пользовательские требования к производительности достаточно серьезны, чтобы ее необходимо было анализировать на стадии проектирования
5. То же, кроме того, на стадиях проектирования, разработки и (или) реализации для удовлетворения пользовательских требований к производительности используются специальные средства анализа

Эксплуатационные ограничения

0. Какие-либо явные или неявные ограничения отсутствуют
1. Эксплуатационные ограничения присутствуют, но не требуют никаких специальных усилий
2. Должны учитываться некоторые ограничения, связанные с безопасностью или временем реакции
3. Должны учитываться конкретные требования к процессору со стороны конкретных компонентов приложения
4. Заданные эксплуатационные ограничения требуют специальных ог-

раничений на выполнение приложения в центральном или выделенном процессоре

5. То же, кроме того, специальные ограничения затрагивают распределенные компоненты системы

Частота транзакций

0. Пиковых периодов не ожидается
1. Ожидаются пиковые периоды (ежемесячные, ежеквартальные, ежегодные)
2. Ожидаются еженедельные пиковые периоды
3. Ожидаются ежедневные пиковые периоды
4. Высокая частота транзакций требует анализа производительности на стадии проектирования
5. То же, кроме того, на стадиях проектирования, разработки и (или) внедрения необходимо использовать специальные средства анализа производительности

Ввод данных в режиме «онлайн»

0. Все транзакции обрабатываются в пакетном режиме
1. От 1% до 7% транзакций требуют интерактивного ввода данных
2. От 8% до 15% транзакций требуют интерактивного ввода данных
3. От 16% до 23% транзакций требуют интерактивного ввода данных
4. От 24% до 30% транзакций требуют интерактивного ввода данных
5. Более 30% транзакций требуют интерактивного ввода данных

Эффективность работы конечных пользователей¹

0. Ни одной из перечисленных функциональных возможностей¹
1. От одной до трех функциональных возможностей
2. От четырех до пяти функциональных возможностей

¹ Эффективность работы конечных пользователей определяется наличием следующих функциональных возможностей.

3. Шесть или более функциональных возможностей при отсутствии конкретных пользовательских требований к эффективности
4. То же, кроме того, пользовательские требования к эффективности требуют специальных проектных решений для учета эргономических факторов (например, минимизации нажатий клавиш, максимизации значений по умолчанию, использования шаблонов)
5. То же, кроме того, пользовательские требования к эффективности требуют применения специальных средств и процессов, демонстрирующих их выполнение
 - Средства навигации (например, функциональные клавиши, динамически генерируемые меню).
 - Меню.
 - Онлайн-подсказки и документация.
 - Автоматическое перемещение курсора.
 - Скроллинг.
 - Удаленная печать.
 - Предварительно назначенные функциональные клавиши.
 - Выбор данных на экране с помощью курсора.
 - Использование видеоэффектов, цветового выделения, подчеркивания и других индикаторов.
 - Всплывающие окна.
 - Минимизация количества экранов, необходимых для выполнения бизнес-функций.
 - Поддержка двух и более языков.

Онлайновое обновление

0. Отсутствует
1. Онлайновое обновление от одного до трех управляющих файлов Объем обновлений незначителен, восстановление несложно
2. Онлайновое обновление четырех или более управляющих файлов Объем обновлений незначителен, восстановление несложно

3. Онлайнное обновление основных внутренних логических файлов
4. То же, плюс необходимость специальной защиты от потери данных
5. То же, кроме того, большой объем данных требует учета затрат на процесс восстановления. Требуются автоматизированные процедуры восстановления с минимальным вмешательством оператора

Сложная обработка¹

0. Ни одной из перечисленных функциональных возможностей¹
1. Любая одна из возможностей
2. Любые две из возможностей
3. Любые три из возможностей
4. Любые четыре из возможностей
5. Все пять возможностей
 - повышенная реакция на внешние воздействия и (или) специальная защита от внешних воздействий;
 - экстенсивная логическая обработка;
 - экстенсивная математическая обработка;
 - обработка большого количества исключительных ситуаций;
 - поддержка разнородных типов входных/выходных данных.

Повторное использование

0. Отсутствует
1. Повторное использование кода внутри одного приложения
2. Не более 10% приложений будут использоваться более чем одним пользователем
3. Более 10% приложений будут использоваться более чем одним пользователем
4. Приложение оформляется как продукт и (или) документируется для облегчения повторного использования. Настройка приложения выполняется пользователем на уровне исходного кода.

¹ Сложная обработка характеризуется наличием у приложения следующих функциональных возможностей:

5. То же, с возможностью параметрической настройки приложений

Простота установки

0. К установке не предъявляется никаких специальных требований.
1. Для установки требуется специальная процедура
2. Заданы пользовательские требования к конвертированию (переносу существующих данных и приложений в новую систему) и установке, должны быть обеспечены и проверены соответствующие руководства. Конвертированию не придается важное значение
3. То же, однако конвертированию придается важное значение.
4. То же, что и в случае 2, плюс наличие автоматизированных средств конвертирования и установки
5. То же, что и в случае 3, плюс наличие автоматизированных средств конвертирования и установки

Простота эксплуатации

0. К эксплуатации не предъявляется никаких специальных требований, за исключением обычных процедур резервного копирования
- 1-4. Приложение обладает одной, несколькими или всеми из перечисленных далее возможностей. Каждая возможность, за исключением второй, обладает единичным весом: 1) наличие процедур запуска, копирования и восстановления с участием оператора; 2) то же, без участия оператора; 3) минимизируется необходимость в монтировании носителей для резервного копирования; 4) минимизируется необходимость в средствах подачи и укладки бумаги при печати
5. Вмешательство оператора требуется только при запуске и завершении работы системы. Обеспечивается автоматическое восстановление работоспособности приложения после сбоев и ошибок

Количество возможных установок на различных платформах

0. Приложение рассчитано на установку у одного пользователя
1. Приложение рассчитано на много установок для строго стандартной платформы (технические средства + программное обеспечение)

2. Приложение рассчитано на много установок для платформ с близкими характеристиками
3. Приложение рассчитано на много установок для различных платформ
4. То же, что в случаях 1 или 2, плюс наличие документации и планов поддержки всех установленных копий приложения
5. То же, что в случае 3, плюс наличие документации и планов поддержки всех установленных копий приложения

Гибкость¹

0. Ни одной из перечисленных возможностей¹
 1. Любая одна из возможностей
 2. Любые две из возможностей
 3. Любые три из возможностей
 4. Любые четыре из возможностей
 5. Все пять возможностей
 - поддержка простых запросов, например, логики и (или) в применении только к одному ILF (вес - 1);
 - поддержка запросов средней сложности, например, логики и (или) в применении более чем к одному ILF (вес - 2);
 - поддержка сложных запросов, например, комбинации логических связок и (или) в применении к одному или более ILF (вес — 3);
 - управляющая информация хранится в таблицах, поддерживаемых пользователем в интерактивном режиме, однако эффект от ее изменений проявляется на следующий рабочий день;
 - то же, но эффект проявляется немедленно (вес - 2).

После определения всех значений GSC и вычисления поправочного коэффициента VAF вычисляется итоговая оценка количества функциональных точек (Adjusted Function Points, AFP):

$$AFP = UFP * VAF.$$

¹ Гибкость характеризуется наличием у приложения следующих возможностей:

ОЦЕНКА ТРУДОЕМКОСТИ РАЗРАБОТКИ

Ниже в соответствии с данными SPR определяется количество строк кода (SLOC) на одну функциональную точку в зависимости от используемого языка программирования.

Количество строк кода на одну функциональную точку

Язык (средство)	Количество SLOC на FP
ABAP/4	16
Access	38
ANSI SQL	13
C++	53
Clarion	58
Data base default	40
Delphi 5	18
Excel 5	6
FoxPro 2.5	34
Oracle Developer	23
PowerBuilder	16
Smalltalk	21
Visual Basic 6	24
Visual C++	34
HTML 4	14
Java 2	46

Умножая AFP на (количество SLOC на FP), получаем количество SLOC в приложении.

Далее для оценки трудоемкости и времени разработки может использоваться один из вариантов известной модели оценки трудоемкости разработки ПО под названием COCOMO и ее современной версии COCOMO II (см. под-

разд. 6.3).

В табл. 6.5 приведены усредненные статистические данные размера ПО по некоторым видам приложений.

Таблица 6.5

Размер программного обеспечения в FP и LOC

Тип ПО	Размер, FP	Размер, LOC	Количество LOC на одну
Текстовые процессоры	3500	437500	125
Электронные таблицы	3500	437500	125
Клиент-серверные приложения	7500	675000	90
ПО баз данных	7500	937500	125
Производственные приложения	7500	937500	125
Крупные бизнес-приложения	10000	1050000	105
Крупные корпоративные приложения	25000	2625000	105
Крупные приложения в госучреждениях	50000	5250000	105
Операционные системы	75000	11250000	150
Системы масштаба предприятий	150000	18750000	125
Крупные оборонные системы	250000	25000000	100

На реализацию проектов размером в одну функциональную точку требуется один день, и они практически всегда заканчиваются успешно. Таковы, как правило, небольшие утилиты для временных нужд.

Объем в 10 функциональных точек — это типичный объем небольших приложений и дополнений, вносимых в готовые системы. Такие проекты требуют до 1 месяца работ и тоже всегда успешны.

Объем в 100 функциональных точек близок к пределам возможностей программиста-одиночки. Проект доводится до конца за 6 месяцев в 85% случаев. Эти цифры верны для современных средств разработки: 15 лет назад 15 тысяч

строк занимал профессиональный интерпретатор Бейсика для MS-DOS, создать который одному человеку (среднему программисту) было не под силу.

Объем проекта в 1000 функциональных точек характерен для большинства сегодняшних коммерческих настольных и небольших клиент-серверных приложений. Заметную долю общего объема работы начинает занимать документация. Для разработки проекта необходима группа примерно из 10 программистов, проектировщиков, специалистов по управлению качеством и около года работы. Проваливается 15% подобных групповых проектов и 65% попыток программистов-одиночек. В 20% случаев не удается уложиться в срок. Перерасход средств отмечается в 25% проектов.

Для проекта объемом в 10000 функциональных точек требуется около 100 разработчиков. Остро встают вопросы организации совместной работы нескольких групп сотрудников. Работы делятся от 1,5 до 5 лет, при этом запланированные сроки чаще всего не выдерживаются. Хорошее качество системы невозможно обеспечить без использования формальной технологии тестирования. До 50% проектов заканчивается неудачей, а реализация таких проектов в одиночку вообще невозможна.

Объем в 100000 функциональных точек — пока что предел для большинства сегодняшних проектов. Это объем современных операционных систем, таких как MS Windows или IBM MVS, и крупнейших военных систем. На их создание уходит 5-8 лет. Над проектом трудятся сотни разработчиков иногда из разных стран, и эффективно координировать их работу не удастся. В законченных системах много ошибок. До 65% проектов завершается провалом. Во всех успешных проектах используются системы управления конфигурацией.

Табл. 6.6 иллюстрирует различия в распределении временных затрат по стадиям жизненного цикла в случае большого и маленького проекта. Маленький проект (например, приложение Windows объемом 50000 строк исходного кода на Visual Basic, создаваемое командой из 5 человек) может потребовать 1 месяц на начальную стадию, 2 месяца на проектирование, 5 месяцев на разработку и 2 месяца на ввод в действие. Большой, сложный проект (например, бор-

товая программа для летательного аппарата объемом 300000 строк исходного кода, создаваемая командой из 40 человек) может потребовать 8 месяцев на начальную стадию, 14 месяцев на проектирование, 20 месяцев на разработку и 8 месяцев на ввод в действие. Сравнение долей жизненного цикла, приходящихся на каждую стадию, позволяет обнаружить очевидные различия.

Таблица 6.6

Распределение временных затрат по стадиям для маленьких и больших проектов

Масштаб проекта	Начальная стадия, %	Проектирование, %	Разработка, %	Ввод в действие, %
Маленький коммерческий проект	10	20	50	20
Большой, сложный проект	15	30	40	15

Самым значительным отличием является относительное время, за которое достигается контрольная точка жизненного цикла, связанная с созданием архитектуры (завершение проектирования). Для маленьких проектов это отношение составляет 30/70; для большого проекта оно ближе к 45/55.

Время разработки может быть изменено с учетом статистических данных, накопленных в реальных проектах и отраженных в табл. 6.7, где сопоставлены планируемый и реальный сроки выполнения проекта в зависимости от его размера, выраженного в количестве функциональных точек. Частично это отставание объясняется неточной оценкой, частично — ростом количества требований к системе после того, как выполнена начальная оценка.

Таблица 6.7

Статистические данные

Размер проекта	< 100 FP	100 - 1000 FP	1000 - 10000 FP	> 10000 FP
Планируемый	6	12	18	24
Реальный срок	8	16	24	36
Отставание	2	4	6	12

АЛГОРИТМИЧЕСКОЕ МОДЕЛИРОВАНИЕ

ТРУДОЕМКОСТИ РАЗРАБОТКИ

ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

6.3.1.

ТЕОРЕТИЧЕСКИЕ (МАТЕМАТИЧЕСКИЕ) МОДЕЛИ

Математическое моделирование трудоемкости разработки ПО основано на сопоставлении экспериментальных данных с формой существующей математической функции. В начале 1960-х годов Питер Норден из фирмы IBM пришел к выводу, что в проектах по исследованию и разработке может применяться хорошо прогнозируемое распределение трудовых ресурсов, основанное на распределении вероятности, называемом кривой Рэля (Rayleigh distribution). Позднее, в 1970-х годах Лоуренс Патнэм¹ из компании Quantitative Systems Management применил результаты Нордена к разработке ПО. Используя статистический анализ проектов, Патнэм обнаружил, что взаимосвязь между тремя основными параметрами проекта (размером, временем и трудоемкостью) напоминает функцию Нордена-Рэля (рис. 6.9), отражающую распределение трудовых ресурсов проекта в зависимости от времени.

¹ Putnam L.H. A General Empirical Solution to the Macro Software Sizing and Estimating Problem // IEEE Transactions on Software Engineering, 1978, July. — P. 345-361.

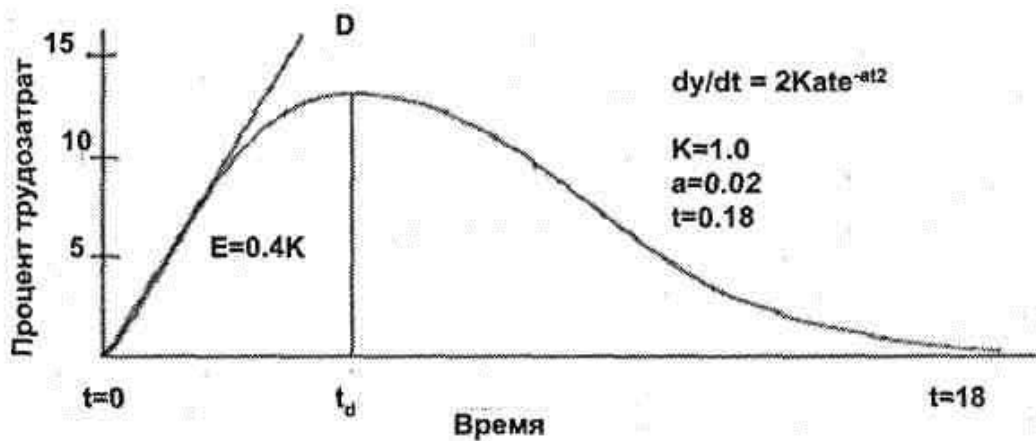


Рис. 6.9. Функция Рэля

Функция Рэля моделируется дифференциальным уравнением:

$$\frac{dy}{dt} = 2 * K * a * t * \exp(-at^2).$$

где $\frac{dy}{dt}$ — скорость роста персонала проекта; t — время, прошедшее от начала проекта до изъятия продукта из эксплуатации; K — область под кривой — представляет полную трудоемкость в течение всего жизненного цикла (включая сопровождение), выраженную в человеко-годах; a — константа, которая определяет форму кривой (фактор ускорения) и вычисляется по формуле

$$a = \frac{1}{2t_d^2},$$

где t_d - время разработки.

Приняв ряд допущений, Патнэм получил следующее уравнение:

$$E = 0,4 * \left(\frac{S}{C}\right)^3 * \frac{1}{(t_d)^4},$$

где E — трудоемкость разработки ПО, S — размер ПО в LOC, t_d — планируемый срок разработки, C — технологический фактор, учитывающий различные аппаратные ограничения, опыт персонала и характеристики среды программирования. Он определяется на основе хронологических данных по прошлым проектам и, согласно рекомендациям Патнэма определяется для различных

типов проектов следующим образом:

- проект, внедренный в сжатые сроки без детальной проработки, -1500;
- проект, выполненный в соответствии с четким планом, — 5000;
- проект, предусматривающий оптимальную организацию и поддержку, — 10000.

Оптимальный срок разработки определяется как

$$t_d = 2,4 * E^{\frac{1}{3}},$$

что хорошо согласуется с большинством статистических моделей.

Более подробное описание модели Патнэма приведено в книге Фатрелл Р., Шафер Д., Шафер Л. Управление программными проектами: достижение оптимального качества при минимуме затрат: Пер. с англ. — М.: Вильямс, 2003.

6.3.2.

СТАТИСТИЧЕСКИЕ (РЕГРЕССИОННЫЕ) МОДЕЛИ

Статистические модели используют накопленные хронологические данные, чтобы получить значения для коэффициентов модели. Для определения соотношений между параметрами модели и трудоемкостью разработки ПО используется регрессионный анализ. Существуют две формы статистических моделей: линейные и нелинейные.

Линейные статистические модели имеют следующий вид:

$$\text{Трудоёмкость} = b_0 + \sum_{i=1}^n b_i * x_i$$

где x_i — факторы, влияющие на трудоемкость, b_i — коэффициенты модели. Линейные модели работают не слишком хорошо, поскольку практика показывает, что соотношения между трудоемкостью и размером ПО нелинейно. По мере роста размера ПО возникает экспоненциальный отрицательный эффект масштаба.

Нелинейные, статистические модели имеют следующий вид:

$$\text{Трудоёмкость} = A * (\text{РазмерПО})^b$$

где A — комбинация факторов, влияющих на трудоемкость; b — экспоненциальный коэффициент масштаба.

Статистические модели просты для понимания, но имеют следующий недостаток: результаты справедливы в основном только для конкретной ситуации. Другой недостаток — при увеличении количества входных параметров количество данных, необходимых для калибровки модели, также возрастает.

Статистическая модель COSOMO II

Модель COSOMO¹ (Constructive COst Model — конструктивная модель стоимости), разработанная Барри Боэмом, является одной из самых известных и хорошо документированных моделей оценки трудоемкости разработки ПО. Исходная модель COSOMO основывалась на базе данных по 56 выполненным проектам, а ее различные варианты отражали различия между процессами в различных областях ПО.

В модели COSOMO используется ряд допущений.

- Исходный код конечного продукта включает в себя все (кроме комментариев) строки кода.
- Начало цикла разработки совпадает с началом разработки продукта, окончание совпадает с окончанием приемочного тестирования, завершающим стадию интеграции и тестирования (работа и время, затрачиваемые на анализ требований, оцениваются отдельно как дополнительный процент от оценки разработки в целом).
- Виды деятельности включают в себя только непосредственно направленные на выполнение проекта работы, в них не входят обычные вспомогательные виды деятельности, такие, как административная поддержка, техническое обеспечение и капитальное оборудование.
- Человеко-месяц состоит из 152 ч.
- Проект управляется надлежащим образом, в нем используются стабиль-

¹ Боэм Б.У. Инженерное проектирование программного обеспечения: Пер. с англ. — М.: Радио и связь, 1985.

ные требования.

Проект COSOMO II (современный вариант модели COSO-MO) был выполнен в Центре по разработке ПО Южно-Калифорнийского университета (USC Centre for Software Engineering). Этот проект преследовал следующие цели.

- Разработать модель для оценки трудоемкости и сроков создания ПО для итерационной модели жизненного цикла ПО, которая будет применяться в 1990-х и 2000-х годах.
- Создать базу данных по трудоемкости ПО.
- Разработать инструментальную поддержку для усовершенствования модели.
- Создать количественную аналитическую схему для оценки технологий создания ПО и их экономического эффекта.

Уравнения COSOMO II для оценки номинальных значений трудоемкости и времени имеют следующий вид. Трудоемкость (в человеко-месяцах):

$$PM_{NS} = A \times Size^E \times \prod_{i=1}^n EM_i,$$

где

$$E = B + 0,01 \times \prod_{j=1}^5 SF_j,$$

Календарное время:

$$TDEV_{NS} = C \times (PM_{NS})^F,$$

где

$$F = D + 0,2 \times 0,01 \times \sum_{j=1}^5 SF_j = D + 0,2 \times (E - B),$$

EM_i — мультипликативные коэффициенты трудоемкости;

SF_j — экспоненциальные коэффициенты масштаба;

$Size$ — размер ПО, выраженный в тысячах строк исходного кода или количестве функциональных точек без учета поправочных коэффициентов (UFP), определенном по методике IFPUG, с последующим преобразованием в количе-

ство строк кода.

Калибровочные переменные A, B, C и D в модели СОСОМО II версии 2000 г. принимают следующие значения: A = 2.94, B — 0.91, C = 3.67, D = 0.28.

Коэффициенты E_{Mt} отражают совместное влияние многих параметров. Они позволяют характеризовать и нормировать среду разработки по параметрам, содержащимся в базе данных проектов модели СОСОМО II (в настоящее время более 160 проектов). Каждый коэффициент в зависимости от установленного значения (очень низкое, низкое, номинальное, высокое, очень высокое) вносит свой вклад в виде множителя с определенным диапазоном значений. Результат учета этих 17 коэффициентов используется при вычислении в уравнении трудоемкости. Состав коэффициентов приведен в табл. 6.8.

Таблица 6.8

Мультипликативные коэффициенты трудоемкости

Идентифика-	Описание коэффициента	Диапазон
RELY	Требуемая надежность	0.82-1.26
DATA	Размер базы данных	0.90-1.28
CPLX	Сложность продукта	0.73 - 1.74
RUSE	Требуемый уровень повторного исполь-	0.95- 1.24
DOCU	Соответствие документации требовани-	0.81- 1.23
TIME	Ограничение времени выполнения	1.00- 1.63
STOR	Ограничение по объему основной памя-	1.00-1.46
PVOL	Изменчивость платформы	0.87- 1.30
ACAP	Способности аналитика	1.42-0.71
PCAP	Способности программиста	1.34-0.76
APEX	Знание приложений	1.22-0.81
PLEX	Знание платформы	1.19-0.85
PCON	Преемственность персонала	1.29-0.81
LTEX	Знание языка/инструментальных средств	1.20-0.84
TOOL	Использование инструментальных	1.17 - 0.78
SCED	Требуемые сроки разработки	1.43-1.00
SITE	Рассредоточенность команды разработ-	1.22-0.80

Показатель экспоненты процесса E может изменяться в диапазоне от 0.91 до 1.23 и определяется как сочетание следующих параметров.

1. Наличие прецедентов у приложения: степень опытности организационно-разработчика в данной области.
2. Гибкость процесса: степень строгости контракта, порядок его выполне-

ния, присущая контракту свобода внесения изменений, виды деятельности в течение всего жизненного цикла и взаимодействие между заинтересованными сторонами.

3. Разрешение рисков, присущих архитектуре: степень технической осуществимости, продемонстрированной до перехода к полномасштабному внедрению.

4. Сплоченность команды: степень сотрудничества и того, насколько все заинтересованные стороны (заказчики, разработчики, пользователи, ответственные за сопровождение и другие) разделяют общую концепцию.

5. Зрелость процесса: уровень зрелости организации-разработчика, определяемый в соответствии с моделью СММ.

В табл. 6.9 приведены возможные значения экспоненциальных коэффициентов масштаба, составляющих в сумме показатель E. Суммарное влияние этих коэффициентов может оказаться весьма существенным.

Таблица 6.9

Экспоненциальные коэффициенты масштаба модели СОСОМО II

Параметр	Характеристика	Значение
PREC - наличие прецедентов	Полное отсутствие прецедентов, полностью непредсказуемый проект	6.20
	Почти полное отсутствие прецедентов, в значительной степени непредсказуемый проект	4.96
	Наличие некоторого количества прецедентов	3.72
	Общее знакомство с проектом	2.48
	Значительное знакомство с проектом	1.24
	Полное знакомство с проектом	0.00
FLEX - гибкость процесса разработки	Точный, строгий процесс разработки	5.07
	Случайные послабления в процессе	4.05
	Некоторые послабления в процессе	3.04
	Большой частью согласованный процесс	2.03
	Некоторое согласование процесса	1.01
	Заказчик определил только общие цели	0.00

RESL - разрешение рисков в архитектуре	Малое (20%)	7.07
	Некоторое (40%)	5.65
	Частое (60%)	4.24
	В целом (75%)	2.83
	Почти полное (90%)	1.41
	Полное (100%)	0.00
ТЕАМ - сплоченность команды	Сильно затрудненное взаимодействие	5.48
	Несколько затрудненное взаимодействие	4.38
	Некоторая согласованность	3.29
	Повышенная согласованность	2.19
	Высокая согласованность	1.10
	Взаимодействия как в едином целом	0.00
РМАТ - зрелость процессов	Уровень 1 СММ	7.80
	Уровень 1+ СММ	6.24
	Уровень 2 СММ	4.68
	Уровень 3 СММ	3.12
	Уровень 4 СММ	1.56
	Уровень 5 СММ	0.00

Пример экспоненциального коэффициента масштаба — коэффициент зрелости процессов (РМАТ— Process Maturity).

Значение коэффициента РМАТ зависит в основном от уровня зрелости процессов в соответствии с моделью СММ. Процедура определения значения РМАТ основана на определении процента соответствия для каждой из 18 основных групп процессов (key process areas — КРА), определенных в СММ. Процент соответствия вычисляется на основе групп процессов (подробно описано 8 групп из 18).

Группа процессов

КРА 1 — Управление требованиями

- Требования к системе контролируются и служат основой для разработки ПО и управления проектом

- Планы создания ПО, продукты и виды деятельности согласованы с требованиями к ПО

КРА 2 — Планирование проекта

- Оценки ПО документируются и используются для планирования и отслеживания проекта
- Проектные виды деятельности и обязанности планируются и документируются
- Участники проекта (группы и индивидуумы) придерживаются своих обязанностей по отношению к проекту

КРА 3 — Отслеживание и контроль проекта

- Текущие результаты и продуктивность отслеживаются на предмет соответствия планам
- В случае существенного отклонения от планов предпринимаются корректирующие действия
- Изменения в обязанностях согласовываются с соответствующими группами и индивидуумами

КРА 4 — Управление контрактами

КРА 5 — Обеспечение качества продукта

КРА 6 — Управление конфигурацией ПО

- Управление конфигурацией планируется
- Обеспечивается идентификация, контроль и доступ к выбранным рабочим продуктам
- Изменения, вносимые в конкретные рабочие продукты, контролируются
- Участники проекта информируются о состоянии содержания базовых версий ПО

КРА 7 — Координация процессов организации

КРА 8 — Стандартизация процессов организации

КРА 9 - Обучение

КРА 10 — Интегрированное управление созданием ПО

- Процессы создания ПО в рамках конкретного проекта являются адап-

тированной версией стандартных процессов, принятых в организации

- Проект планируется и управляется в соответствии с установленным в проекте процессом создания ПО

КРА 11 — Разработка программного продукта

- Задачи разработки продукта четко определены, интегрированы и последовательно реализуются
- Рабочие продукты согласованы друг с другом

КРА 12 — Межгрупповая координация

КРА 13 — Экспертные оценки

КРА 14 — Количественное управление проектом

КРА 15 — Управление качеством продукта

КРА 16 — Предотвращение дефектов

КРА 17 — Управление изменениями в технологии

- Изменения в технологии планируются
- Новые технологии оцениваются на предмет их воздействия на качество и продуктивность
- Подходящие новые технологии внедряются в обычную практику организации

КРА 18 — Управление изменениями в процессах

- Планируется постоянное совершенствование процессов
- В совершенствовании процессов участвует вся организация
- Стандартные процессы организации и процессы конкретных проектов постоянно совершенствуются

Значение рейтинга по каждой КРА определяется в соответствии с табл.

6.10.

Значения рейтинга КРА

Характеристика состояния КРА в организации	Вариант ответа в таблице рейтингов	Значения КРА, %
Задачи хорошо определены в стандартных процедурах и последовательно реализуются (более 90% времени)	Почти всегда	100
Задачи реализуются относительно часто, однако при определенных затруднениях не выполняются (от 60 до 90% времени)	Часто	75
Задачи реализуются от 40 до 60% времени	Наполовину	50
Задачи реализуются не слишком часто (от 10 до 40% времени)	Случайно	25
Задачи реализуются редко (менее 10% времени)	Очень редко	1
Данные процессы вообще не реализуются	Не применяется	0
Невозможно сказать ничего определенного относительно данных процессов	Неизвестно	0

Оценочный уровень зрелости процессов (EPML) вычисляется следующим образом:

$$EMPL = 5 \times \left(\sum_{i=1}^n \frac{KPA\%_i}{100} \right) \times \frac{1}{n},$$

где значение КРА% определяется по табл. 6.10.

Значения коэффициента РМАТ определяются в соответствии с табл. 6.11.

Таблица 6.11

Значения коэффициента РМАТ

Оценочный уровень зрелости процессов (EPML)	Уровень зрелости процессов	Значение РМАТ
0	Уровень 1 СММ	7.8
1	Уровень 1+ СММ	6.24
2	Уровень 2 СММ	4.68
3	Уровень 3 СММ	3.12
4	Уровень 4 СММ	1.56
5	Уровень 5 СММ	0

Пример мультипликативного коэффициента трудоемкости — коэффициент использования инструментальных средств (TOOL)

Значения коэффициента TOOL вычисляются в соответствии с табл. 6.12.

В целом модель СОСОМО II является хорошим усовершенствованием традиционных и устаревших моделей трудоемкости. Она вполне соответствует принципам итерационной разработки и современным технологиям создания ПО. В частности, СОСОМО II активно используется в технологии Rational Unified Process. Вместе с тем она постоянно развивается, поскольку ее база данных пополняется сведениями о разнообразных проектах¹.

¹ Более подробное описание модели СОСОМО II (на русском языке) приведено в книгах: Орлов С.А. Технологии разработки программного обеспечения. — СПб.: Питер, 2002; Фатрелл Р., Шафер Д., Шафер Л. Управление программными проектами: достижение оптимального качества при минимуме затрат: Пер. с англ. — М.: Вильямс, 2003.

Значения коэффициента TOOL

Дескрипторы TOOL	Уровни рейтинга	Значение TOOL
Редакторы кода, отладчики	Очень низкий	1.17
Простые CASE-средства с минимальной интеграцией	Низкий	1.09
Средства поддержки основных процессов ЖЦ, средняя степень интеграции	Номинальный	1.00
Мощные, развитые средства поддержки ЖЦ, средняя степень интеграции	Высокий	0.90
Мощные, развитые средства поддержки ЖЦ, хорошо интегрированные с процессами и методами, повторное использование	Очень высокий	0.78

6.4.

**МЕТОДИКА ОЦЕНКИ ТРУДОЕМКОСТИ
РАЗРАБОТКИ ПО НА ОСНОВЕ ВАРИАНТОВ
ИСПОЛЬЗОВАНИЯ (ПО МАТЕРИАЛАМ
КОМПАНИИ RATIONAL SOFTWARE)**

6.4.1.

**ОПРЕДЕЛЕНИЕ ВЕСОВЫХ ПОКАЗАТЕЛЕЙ
ДЕЙСТВУЮЩИХ ЛИЦ**

Все действующие лица системы делятся на три типа: простые, средние и

сложные.

Простое действующее лицо представляет внешнюю систему с четко определенным программным интерфейсом (API).

Среднее действующее лицо представляет либо внешнюю систему, взаимодействующую с данной системой посредством протокола наподобие TCP/IP, либо личность, пользующуюся текстовым интерфейсом (например, ASCII-терминалом).

Сложное действующее лицо представляет личность, пользующуюся графическим интерфейсом (GUI).

Подсчитанное количество действующих лиц каждого типа умножается на соответствующий весовой коэффициент, затем вычисляется общий весовой показатель A .

Весовые коэффициенты действующих лиц

Тип действующего лица	Весовой коэффициент
простое	1
среднее	2
сложное	3

В качестве примера рассмотрим систему регистрации для учебного заведения, описанную в главе 3:

Типы действующих лиц

Действующее лицо	Тип
Студент	Сложное
Профессор	Сложное
Регистратор	Сложное
Расчетная система	Простое
Каталог курсов	Простое

Таким образом, общий весовой показатель равен:

$$A = 2 * 1 + 3 * 3 = 11,$$

ОПРЕДЕЛЕНИЕ ВЕСОВЫХ ПОКАЗАТЕЛЕЙ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Все варианты использования делятся на три типа: простые, средние и сложные, в зависимости от количества транзакций в потоках событий (основных и альтернативных). В данном случае под транзакцией понимается атомарная последовательность действий, которая выполняется полностью или отменяется.

Подсчитанное количество вариантов использования каждого типа умножается на соответствующий весовой коэффициент, затем вычисляется общий весовой показатель UCP (табл. 6.13).

Другой способ определения сложности вариантов использования заключается в подсчете количества классов анализа, участвующих в их реализации (табл. 6.14).

Таблица 6.13

Весовые коэффициенты вариантов использования

Тип варианта использования	Описание	Весовой коэффициент
Простой	3 или менее транзакций	5
Средний	От 4 до 7 транзакций	10
Сложный	Более 7 транзакций	15

Таблица 6.14

Весовые коэффициенты вариантов использования

Тип варианта использования	Описание	Весовой коэффициент
Простой	Менее 5 классов	5
Средний	От 5 до 10 классов	10
Сложный	Более 10 классов	15

Для системы регистрации сложность вариантов использования определяет-

ся следующим образом:

Сложность вариантов использования

Вариант использования	Тип
Войти в систему	Простой
Зарегистрироваться на курсы	Средний
Просмотреть таблицу успеваемости	Простой
Выбрать курсы для преподавания	Средний
Проставить оценки	Простой
Вести информацию о профессорах	Простой
Вести информацию о студентах	Простой
Закрыть регистрацию	Средний

Таким образом, общий весовой показатель равен:

$$UCP = 5*5+10*3 = 45.$$

В результате получаем показатель UUCP (unadjusted use case points):

$$UUCP=4 + UC = 56.$$

6.4.3.

ОПРЕДЕЛЕНИЕ ТЕХНИЧЕСКОЙ

СЛОЖНОСТИ ПРОЕКТА

Техническая сложность проекта (TCF — technical complexity factor) вычисляется с учетом показателей технической сложности (табл. 6.15).

Таблица 6.15

Показатели технической сложности

Показатель	Описание	Вес
T1	Распределенная система	2
T2	Высокая производительность (пропускная способность)	1

T3	Работа конечных пользователей в режиме он-лайн	1
T4	Сложная обработка данных	1
T5	Повторное использование кода	1
T6	Простота установки	0,5
T7	Простота использования	0,5
T8	Переносимость	2
T9	Простота внесения изменений	1
T10	Параллелизм	1
T11	Специальные требования к безопасности	1
T12	Непосредственный доступ к системе со стороны внешних пользователей	1
T13	Специальные требования к обучению пользователей	1

Каждому показателю присваивается значение T_i в диапазоне от 0 до 5 (0 означает отсутствие значимости показателя для данного проекта, 5 — высокую значимость). Значение TCF вычисляется по следующей формуле:

$$TCF = 0,6 + (0,01 * (\sum T_i \cdot Вес_i))$$

Вычислим TCF для системы регистрации (табл. 6.16).

Таблица 6.16

Показатели технической сложности системы регистрации

Показа-	Вес	Значение	Значение с учетом
T1	2	4	8
T2	1	3	3
T3	1	5	5
T4	1	1	1
T5	1	0	0
T6	0,5	5	2,5
T7	0,5	5	2,5
T8	2	0	0
T9	1	4	4
T10	1	5	5
T11	1	3	3
T12	1	5	5
T13	1	1	1
Σ			40

$$TCF = 0,6 + (0,01 * 40) = 1,0.$$

ОПРЕДЕЛЕНИЕ УРОВНЯ КВАЛИФИКАЦИИ РАЗРАБОТЧИКОВ

Уровень квалификации разработчиков (EF — environmental factor) вычисляется с учетом следующих показателей (табл. 6.17).

Таблица 6.17

Показатели уровня квалификации разработчиков

Показатель	Описание	Вес
F1	Знакомство с технологией	1,5
F2	Опыт разработки приложений	0,5
F3	Опыт использования объектно-ориентированного подхода	1
F4	Наличие ведущего аналитика	0,5
F5	Мотивация	1
F6	Стабильность требований	2
F7	Частичная занятость	-1
F8	Сложные языки программирования	-1

Каждому показателю присваивается значение в диапазоне от 0 до 5. Для показателей F1 — F4 0 означает отсутствие, 3 — средний уровень, 5 — высокий уровень. Для показателя F5 0 означает отсутствие мотивации, 3 — средний уровень, 5 — высокий уровень мотивации. Для F6 0 означает высокую нестабильность требований, 3 — среднюю, 5 — стабильные требования. Для F7 0 означает отсутствие специалистов с частичной занятостью, 3 — средний уровень, 5 — все специалисты с частичной занятостью. Для показателя F8 0 означает простой язык программирования, 3 — среднюю сложность, 5 — высокую сложность.

Значение EF вычисляется по следующей формуле:

$$EF = 1,4 + (-0,03 * (\sum F_i \text{Вес}_i))$$

Вычислим EF для системы регистрации (табл. 6.18).

Таблица 6.18

Показатели уровня квалификации разработчиков системы регистрации

Показа-	Вес	Значение	Значение с учетом
F1	1.5	1	1,5
F2	0,5.	1	0,5
F3	1	1	1
F4	0,5	4	2
F5	1	5	5
F6	?	3	6
F7	-1	0	0
F8	-1	3	-3
S		1	13

$$EF = 1,4 + (-0,03 * 13) = 1,01.$$

В результате получаем окончательное значение UCP (use case points):

$$UCP = UUCP * TCF * EF = 56 * 1,0 * 1,01 = 56,56.$$

6.4.5.

ОЦЕНКА ТРУДОЕМКОСТИ ПРОЕКТА

В качестве начального значения предлагается использовать 20 человеко-часов на одну UCP. Эта величина может уточняться с учетом опыта разработчиков. Приведем пример возможного уточнения.

Рассмотрим показатели F1—F8 и определим, сколько показателей F1—F6 имеют значение меньше 3 и сколько показателей F7—F8 имеют значение больше 3. Если общее количество меньше или равно 2, следует использовать 20 чел.-ч. на одну UCP, если 3 или 4—28. Если общее количество равно 5 или более, следует внести изменения в сам проект, в противном случае риск провала

слишком высок.

Для системы регистрации получаем 28 чел.-ч. на одну UCP, таким образом, общее количество человеко-часов на весь проект равно $56,56 \cdot 28 = 1583,68$, что составляет 40 недель при 40-часовой рабочей неделе. Допустим, что команда разработчиков состоит из четырех человек, и добавим 3 недели на различные непредвиденные ситуации, тогда в итоге получим 13 недель на весь проект.

Опытные данные компании Rational. Проект среднего размера (приблизительно 10 разработчиков, более чем 6-8 месяцев) может включать приблизительно 30 вариантов использования. Это соответствует тому, что средний вариант использования содержит 12 UCP, и каждая UCP требует 20-30 ч. Это означает общую трудоемкость 240-360 чел.-ч. на вариант использования. Таким образом, 30 вариантов использования потребуют приблизительно 9000 чел.-ч. (10 разработчиков в течение 6 месяцев). Однако прямой пропорции не существует: очень большой проект со 100 разработчиками и сроком 20 месяцев не начнется с 1000 вариантов использования из-за проблем размерности.

Использование описанной выше методики для простых и сложных систем хорошо согласуется с опытными данными компании Rational (приблизительно 150-350 ч. на один вариант использования). Самая простая система (весовой показатель $UC = 5$, $A = 2$, $UUCP = 7$) дает (при 20 чел.-ч. на UCP) приблизительно 140 чел.-ч. Сложная система (весовой показатель $UC = 15$, $A = 3$, $UUCP = 18$) дает приблизительно 360 чел.-ч.

6.5.

МЕТОДЫ, ОСНОВАННЫЕ НА ЭКСПЕРТНЫХ

ОЦЕНКАХ

Подходы, основанные на экспертных оценках, применяются при отсутствии дискретных эмпирических данных. Они используют опыт и знания экспертов-практиков в различных областях. Оценки, получаемые при этом, пред-

ставляют собой синтез известных результатов прошлых проектов, в которых принимал участие эксперт.

Очевидными недостатками таких методов являются зависимость оценки от мнения эксперта и тот факт, что, как правило, не существует способов проверить правильность этого мнения до тех пор, когда уже сложно что-либо исправить в том случае, если мнение эксперта окажется ошибочным. Известно, что годы практической работы не всегда переходят в высокое качество профессиональных знаний. Даже признанные эксперты иногда делают неверные догадки и предположения. На основе экспертных оценок были разработаны два метода, допускающие возможность ошибки экспертов: метод Дельфи и метод декомпозиции работ.

6.5.1.

МЕТОД ДЕЛЬФИ

Метод Дельфи был разработан в корпорации «Рэнд» в конце 1940-х гг. и использовался первоначально для прогнозирования будущих событий (отсюда метод и получил свое название по сходству с предсказаниями Дельфийского оракула в Древней Греции). Позднее метод использовался для принятия решений по спорным вопросам. На предварительном этапе участники дискуссии должны без обсуждения с другими ответить на ряд вопросов, относительно их мнения по спорному вопросу. Затем ответы обобщаются, табулируются и возвращаются каждому участнику дискуссии для проведения второго этапа, на котором участникам снова предстоит дать свою оценку спорного вопроса, но на этот раз, располагая мнениями других участников, полученными на первом этапе. Второй этап завершается сужением и выделением круга мнений, отражающих некоторую общую оценку проблемы. Изначально в методе Дельфи коллективное обсуждение не использовалось; обсуждение между этапами метода было впервые применено в обобщенном методе Дельфи. Метод достаточно эффективен в том случае, если необходимо сделать заключение по некото-

рой проблеме, а доступная информация состоит больше из «мнений экспертов», чем из строго определенных эмпирических данных.

6.5.2.

МЕТОД ДЕКОМПОЗИЦИИ РАБОТ

Долгое время являясь фактическим стандартом в практике разработки как программного, так и аппаратного обеспечения, метод декомпозиции работ (МДР) представляет собой способ иерархической организации элементов проекта, упрощающий задачу составления бюджета проекта и контроля за расходованием средств. Он позволяет определить, на что именно расходуются средства. Если с каждой категорией расходов, связанной с тем или иным элементом иерархии проекта, сопоставить некоторую вероятность, можно определить ожидаемую сумму расходов на разработку, начиная с некоторых структурных элементов проекта и заканчивая совокупными затратами на выполнение всего проекта. В рамках данного метода экспертные оценки применяются для составления наиболее нужных спецификаций элементов структуры проекта и для сопоставления каждому элементу определенной степени вероятности категории расходов, связанной с ним. Методы, основанные на экспертных оценках, пригодны для проектов, связанных с разработкой принципиально нового ПО, и для совокупной оценки проектов, но содержат ряд «узких мест», упомянутых выше. Кроме того, методы, основанные на экспертных оценках, плохо масштабируемы, что затрудняет масштабный анализ чувствительности. Подходы, в основу которых положен МДР, хорошо пригодны для планирования и управления.

Метод декомпозиции работ для ПО предполагает существование двух иерархий элементов проекта. Одна из них отражает структуру ПО, другая представляет собой упорядоченные стадии разработки ПО. Иерархия структуры ПО отражает фундаментальную структуру ПО и показывает функции и местоположение каждого элемента в рамках ПО. Иерархия стадий разработки показывает основные этапы, связанные с разработкой данного компонента ПО.

Наряду с процессом оценки МДР широко применяется для расчета себестоимости разработки ПО. Каждому элементу иерархий МДР можно приписать свой бюджет и уровень издержек, что позволяет персоналу составлять отчеты о количестве рабочего времени, которое было затрачено на выполнение заданной задачи в рамках проекта или какого-либо компонента проекта, которые затем могут быть обобщены для административного контроля над использованием средств.

Если организация применяет МДР как стандарт для всех проектов, то с течением времени формируется ценная база данных, отражающая распределение расходов на разработку ПО. На основе этих данных может быть разработана собственная модель оценки расходов, подстроенная под практический опыт организации.

6.6.

СРЕДСТВА ОЦЕНКИ ТРУДОЕМКОСТИ

Ряд средств оценки является коммерческими продуктами — SLIM (Quantitative Systems Management), ESTIMACS (Computer Associates), KnowledgePLAN и CHECKPOINT (SPR). Эти продукты нельзя назвать совершенными, и все они требуют от пользователя высокого уровня квалификации (здесь, как и в других областях деятельности, действует принцип «что заложишь, то и получишь»). В лучшем случае с помощью таких продуктов можно получить оценку с точностью 10%. Даже если точность будет 50%, это все равно лучше, чем брать данные «с потолка».

Существуют специальные программные средства, автоматизирующие проведение оценок по методу функциональных точек и позволяющие оценить, насколько быстро и с какими затратами в действительности удастся реализовать проект. Одним из таких средств является KnowledgePLAN - продукт фирмы SPR.

KnowledgePLAN разработан на основе исследований, проведенных в фир-

ме SPR, в области оценок сложности, трудоемкости и производительности при разработке ПО. Оценка и планирование в пакете KnowledgePLAN ведется на основе статистических закономерностей, выведенных на основе анализа более чем 8000 успешно завершённых проектов из различных областей применения. Исходные данные для вычислений находятся в специальной репозитории, который обновляется по результатам выполнения реальных проектов. В качестве метрик для оценки размеров ПО используются методика подсчета функциональных точек и метод оценки сложности программного продукта — собственная разработка фирмы SPR - метрика, позволяющая учесть алгоритмическую сложность разрабатываемых программ.

KnowledgePLAN обладает следующими возможностями:

- формирование близкого к реальному плана работ по проекту;
- определение трудоемкости и стоимости планируемых проектов;
- учет влияния условий разработки, применяемых инструментальных средств и используемых технологий на прогнозируемую трудоемкость, сроки и стоимость разработки;
- проведение анализа «what — if» («что-если») для поиска лучших решений;
- проведение сравнительного анализа качества и производительности разработки разнотипных проектов, или однотипных проектов, при выполнении которых использовались различные технологии;
- накопление статистической многомерной информации о проекте и его участниках;
- классификация проектов для принятия решения о структуре управления проектом;
- анализ плановой и реальной оценки сложности и величины разработанного ПО и трудоемкости выполнения проекта.

ПЛАНИРОВАНИЕ ИТЕРАЦИОННОГО

ПРОЦЕССА СОЗДАНИЯ ПО

При планировании процесса создания ПО важно не только оценить длительность разработки (как это, например, предлагалось сделать в подразд. 6.3.2), но и представлять себе соотношение длительности различных стадий процесса создания ПО. Так, например, поданным компании Rational Software, распределение времени и объема работ по стадиям представлено в табл. 6.19.

Такое распределение справедливо для проекта, имеющего следующие характеристики:

- средний размер проекта и средний объем работ;
- незначительное число рисков и нововведений;
- нахождение в начальном цикле разработки;
- отсутствие predetermined архитектуры.

Таблица 6.19

Распределение времени и объема работ по стадиям

Стадия	Время работы,	Объем работы,
Начальная ста-	10	5
Разработка	30	20
Конструирова-	50	65
Ввод в действие	10	10

Данное распределение может быть достаточно гибким, оно подчиняется указанным ниже эвристическим правилам.

- Если для определения области действия проекта, изыскания финансирования, изучения рынка или создания первоначального прототипа нужно значительное время, — увеличивается начальная стадия.
- Если отсутствует архитектура, предусматривается применение новых

(или незнакомых) технологий и (или) имеются жесткие ограничения производительности, значительное число технических рисков, а персонал большей частью состоит из новичков, — увеличивается стадия разработки.

- Если проект представляет собой разработку второго поколения существующего продукта и в архитектуру не вносятся значительные изменения — уменьшаются стадии разработки и конструирования.
- Если нужно быстро выпустить продукт на рынок (из-за высокой конкуренции) и спланировать постепенное завершение разработки — уменьшается стадия конструирования и увеличивается стадия ввода в действие.
- Если затруднено внедрение, например, при замещении одной системы другой без прерывания эксплуатации или при необходимости сертификации (в таких областях, как медицинское оборудование, ядерная промышленность или авиационная электроника), — увеличивается стадия ввода в действие.

Следующий важный вопрос «Сколько итераций будет содержать каждая стадия?». Прежде чем принять решение по этому вопросу, нужно рассмотреть продолжительность каждой итерации.

В идеальной ситуации итерация длится от двух до шести недель, хотя в действительности это зависит от характера проекта и размера организации-разработчика. Приведем несколько примеров.

- Группа из пяти человек может выполнить некоторое планирование в понедельник утром, наблюдать за ходом проекта и перераспределять задания во время ежедневных совместных обедов, начать конструирование во вторник, а завершить итерацию в пятницу вечером.
- Если попытаться применить этот сценарий для группы из 20 человек, то это окажется затруднительным. Распределение работы, синхронизация подгрупп и интеграция будут занимать больше времени. В этом случае итерация может занять три—четыре недели.
- Если же группа будет состоять из 40 человек, то только неделя уйдет на

прохождение указаний от руководства к исполнителям. Нужны промежуточные уровни управления; кроме того, достижение общего понимания целей потребует большей формальности и документирования. В этом случае разумной продолжительностью итерации будет уже три месяца.

Кроме того, влияние оказывают и другие факторы: степень знакомства организации с итерационным подходом, стабильность и уровень развития организации, а также уровень технологической зрелости процессов создания ПО. В табл. 6.20 приведена приблизительная оценка длительности итерации, обобщающая результаты, полученные в реальных проектах.

Таблица 6.20

Длительность итерации

Количество строк	Число сотруд-	Длительность итера-
5000	4	2 недели
20000	10	1 месяц
100000	40	3 месяца
1000000	150	8 месяцев

После того как определена приемлемая длительность итерации, следует определить число итераций на каждой стадии.

На начальной стадии может вообще не быть итераций, поскольку не производится никакого ПО, все действия связаны только с планированием и маркетингом. Одна итерация все же может потребоваться для создания прототипа, предназначенного смягчить последствия от реализации основных технических решений.

На стадии разработки нужна, как минимум, одна итерация. Если отсутствует начальная архитектура и нужно адаптироваться к новым инструментальным средствам, технологиям, платформе или языку программирования, то следует планировать две—три итерации. Возможно, потребуются продемонстрировать прототип заказчику или конечным пользователям, чтобы уточнить требования. Кроме того, дополнительная итерация может потребоваться для исправления ошибок, допущенных при разработке архитектуры.

На стадии конструирования также нужно запланировать не менее одной

итерации, при этом количество итераций зависит в основном от реализуемой функциональности (количества вариантов использования).

На стадии ввода в действие нужна, как минимум, одна итерация, например, для выпуска окончательной версии после бета-версии.

В итоге можно определить три уровня проведения полного цикла разработки (табл. 6.21).

Таблица 6.21

Количество итераций по стадиям

Уровень	Начальная стадия	Разработка	Конструирование	Ввод в действие
Низкий	0	1	1	1
Типичный	1	2	2	1
Высокий	1	3	3	2

Таким образом, можно принять в качестве эмпирического правила, что средний итерационный проект включает $6 + 3$ итерации.

! Следует запомнить

1. Оценка трудоемкости создания ПО является одним из наиболее важных видов деятельности в процессе создания ПО.

2. Модели и методы оценки трудоемкости необходимы для разработки бюджета проекта, анализа степени риска и выбора компромиссного решения, планирования и управления проектом, анализа затрат на улучшение качества ПО.

3. Большинство моделей для определения трудоемкости разработки ПО могут быть сведены к функции пяти основных параметров: размера конечного продукта, особенностей процесса, возможностей персонала, среды и требуемого качества продукта.

Основные понятия

Функциональный тип, функциональная точка.

? Вопросы для самоконтроля

1. К каким последствиям могут привести недооценка и переоценка стоимости, времени и ресурсов, требуемых для создания ПО?
2. На какой стадии проекта и с какой точностью может выполняться оценка трудоемкости создания ПО?
3. Охарактеризуйте и сравните методы оценки трудоемкости создания ПО.
4. Что означает хорошая оценка трудоемкости?
5. В каких единицах измеряется размер ПО?
6. Какие факторы наиболее значительно влияют на трудоемкость создания ПО?

ГЛАВА 7

ОСОБЕННОСТИ СОВРЕМЕННЫХ ПРОЕКТОВ

Прочитав эту главу, вы узнаете:

- *В каких условиях выполняется большинство современных проектов создания ПО.*
- *Какую роль играет человеческий фактор в «безнадежных» проектах.*
- *Какие процессы, средства и технологии являются предпочтительными в «безнадежных» проектах.*

В последнее время множество проектов создания ПО выполняется в экстремальных условиях. Независимо от причин такие «смертельные марши» (по определению Эдварда Йордона, одного из ведущих мировых консультантов), или «безнадежные» проекты, предъявляют особые требования к используемым методам управления, технологиям и средствам. Эти требования наиболее полно (и не без юмора) изложены в книге Эдварда Йордона «Death March»¹, выпу-

¹ Йордон Эд. Путь камикадзе: Пер. с англ. — М.: ЛОРИ, 2001 (в настоящее время готовится перевод второго издания).

щенной издательством Prentice-Hall в 1997 г. и в 2003 г. (второе издание).

7.1.

КАТЕГОРИИ «БЕЗНАДЕЖНЫХ» ПРОЕКТОВ

Далеко не все «безнадежные» проекты одинаковы; помимо ограничений в планах, штатах, бюджете и функциональности, они могут иметь различные масштабы, формы и другие особенности.

Наиболее важной отличительной характеристикой «безнадежного» проекта является его масштаб. В зависимости от масштаба можно выделить четыре категории проектов:

- небольшие проекты — проектная команда включает менее 10 человек, работает в исключительно неблагоприятных условиях и должна завершить проект в срок от 3 до 6 месяцев;
- средние проекты - проектная команда включает от 20 до 30 человек, протяженность проекта составляет 1—2 года;
- крупномасштабные проекты - проектная команда включает от 100 до 300 человек, протяженность проекта составляет 3-5 лет;
- гигантские проекты — в проекте участвует армия разработчиков от 1000 до 2000 человек и более (включающая, как правило, консультантов и соисполнителей), протяженность проекта составляет 7—10 лет.

Небольшие проекты являются наиболее распространенной категорией, и шансы на их успешное завершение наиболее высоки. Для средних проектов они заметно ниже, а для крупномасштабных проектов почти равны нулю. В больших проектных командах трудно поддерживать дух сплоченности. Причем решающее значение имеет не только число участников проекта, но и временная протяженность: 80-часовую рабочую неделю в течение 6 месяцев еще можно вытерпеть, но если работать так в течение двух лет, могут возникнуть проблемы.

До сих пор остались без ответа вопросы, почему вполне разумные организации предпринимая подобные проекты и почему разумные менеджеры и технические специалисты соглашаются в них участвовать. Эти вопросы будут рассмотрены ниже.

7.2.

ПРИЧИНЫ, ПОРОЖДАЮЩИЕ

«БЕЗНАДЕЖНЫЕ» ПРОЕКТЫ

Такие проекты порождаются самыми различными причинами:

- высокая конкуренция, вызванная появлением новых компаний на рынке или новых технологий;
- сильное воздействие неожиданных правительственных решений;
- неожиданный и (или) незапланированный кризис;
- политические «игры» высшего руководства;
- наивный оптимизм и менталитет первопроходцев у неопытных разработчиков.

Многие «безнадежные» проекты связаны с применением в первый раз новых технологий. Достаточно вспомнить первые проекты, связанные с объектно-ориентированной технологией, технологией «клиент-сервер», реляционными базами данных или Internet. Некоторые из них носили экспериментальный характер и имели целью извлечение потенциальных выгод из новой технологии, однако другие, вероятно, были ответом конкурентам, внедрившим такую же технологию. В последнем случае такие проекты могут быть непомерно большими, а также отягощенными чрезвычайно агрессивными планами и бюджетами.

В то же время самым серьезным фактором «безнадежности» проекта, помимо его масштаба, плана и бюджета, является попытка использовать непробанованную технологию в критически важных приложениях. Даже если техноло-

гия в принципе применима, она может не годиться для крупномасштабного применения; никто не знает, как использовать ее преимущества и избежать ее недостатков; у поставщика нет опыта ее поддержки и т.п.

Наиболее распространенные причины, побуждающие разработчиков участвовать в «безнадежных» проектах, можно охарактеризовать следующим образом:

- высокое вознаграждение;
- синдром покорителей Эвереста;
- наивность и оптимизм молодости;
- угроза безработицы;
- возможность будущей карьеры;
- возможность победить бюрократию.

Наилучший пример данной ситуации — это работа в начинающей компании. Если заверить участников проекта, что его успех принесет компании всеобщую известность, а им поможет стать миллионерами, то они с радостью будут работать до изнеможения. Конечно, они могут отдавать себе отчет в рискованности этой затеи, но, поскольку многие верят, что они всемогущи и бессмертны, они не обратят особого внимания на риск.

Разумеется, самоуверенность участников проекта выручает в ситуациях, когда обычные проектные команды терпят поражение. Тот факт, что наиболее успешные продукты - от Lotus 1-2-3 до Netscape Navigator — были разработаны небольшими командами в условиях, неприемлемых для нормальных людей, уже стал фольклором в индустрии ПО. Такие проекты, заканчиваясь успешно, приносят проектной команде известность и славу; даже когда они проваливаются, то зачастую позволяют всем участникам извлечь для себя важные уроки (хотя для организации в целом последствия могут быть катастрофическими).

Важно отметить, что наивность и оптимизм молодости обычно сочетаются с огромной энергией, целеустремленностью и отсутствием таких помех, как семейные отношения. Гораздо чаще можно встретить 22-летнего программиста, который готов и жаждет участвовать в «безнадежном» проекте со 100-часовой

рабочей неделей в течение года или двух, чем 35-летнего женатого программиста с двумя детьми и весьма умеренной склонностью к покорению горных вершин. Молодой программист, желающий участвовать в «безнадежном» проекте, а также относительно молодой менеджер проекта, дающий оптимистические обещания начальству, как бы утверждают: «Разумеется, мы обеспечим успех этого проекта и сокрушим все препятствия на своем пути!»

Индустрия ПО привлекает в основном молодых, и вряд ли ситуация изменится в ближайшие несколько лет, а молодежь утратит оптимизм, энергию и способности сосредотачиваться на решении проблем. Что касается наивности, то эта болезнь проходит только с возрастом.

Технические специалисты и менеджеры проектов часто жалуются, что их корпоративные бюрократы мешают продуктивно работать и задерживают разработки ПО. Чем крупнее организация, тем сильнее бюрократия, особенно в тех организациях, где служба стандартов требует строгого соблюдения требований SEI-CMM или ISO-9000. Аналогично департамент по управлению персоналом может использовать процедуры скрупулезной проверки каждого вновь принимаемого на работу сотрудника или стороннего разработчика, привлекаемого к участию в проекте.

«Безнадежные» проекты нередко предоставляют возможность обойти некоторые, если не все, бюрократические рогадки, и этого достаточно, чтобы раздраженные бюрократией разработчики ПО принимали участие в таких проектах. В крайнем случае проектная команда перебирается в отдельное здание, где ничто не мешает им выполнять свою работу. Даже в менее экстремальной ситуации «безнадежный» проект зачастую дает возможность использовать собственные средства и языки программирования, осваивать новые технологии наподобие объектно-ориентированного программирования, а также сокращать большинство громоздких процедур и объем документации, которые в обычных условиях требуются в полном объеме. Не менее важно и то, что менеджер проекта получает большую свободу действий в подборе участников проектной команды, чем в обычных условиях.

Следует отметить, что некоторые аналитики смотрят на проблему оптимистично; они полагают, что количество «безнадежных» проектов в последующие годы будет уменьшаться. Причины такого уменьшения: а) рост корпоративной культуры IT-компаний; б) обучение разработчиков различным методам обеспечения качества в сочетании с «быстрой» разработкой ПО.

7.3.

ПРИЧИНЫ РАЗНОГЛАСИЙ МЕЖДУ УЧАСТНИКАМИ ПРОЕКТА

Ни один проект не может обойтись без политики; она стоит за каждым взаимоотношением между двумя или более индивидуумами. В лучшем случае она может играть незначительную роль, если участники проекта договорятся между собой и достигнут компромисса или один участник (или группа) одержит верх над всеми остальными, и будет диктовать свои решения.

В условиях «безнадежного» проекта наиболее очевидным источником политических споров и дискуссий являются временные, бюджетные и ресурсные ограничения. Если бы с ресурсами было все в порядке, то не было бы и предмета для обсуждения.

Однако существуют и другие источники разногласий. Некоторые из них очевидны для любых разработчиков, например споры по поводу спецификации требований: она действительно отражает реальные требования пользователей или является неадекватной и неполной. Это особенно актуально для «безнадежных» проектов, поскольку не хватает времени на интервьюирование конечных пользователей, формальное документирование, согласование и утверждение их требований. Большинство профессионалов понимает, что это может стать серьезной проблемой и в нормальном проекте, поэтому растет интерес к прототипированию, итерационной разработке и другим подобным методам.

Еще более серьезная политическая дискуссия, результаты которой могут

оказаться полной неожиданностью для менеджера проекта, связана с вопросом о целесообразности создания новой системы. В некоторых случаях участники проекта действуют под давлением различных правительственных постановлений, хотя это вызывает у них раздражение и возмущение. В других случаях новая система может создаваться под давлением руководителя организации, несмотря на явное несогласие его коллег и подчиненных. Менеджер проекта может оказаться активным участником политических баталий, предшествовавших его официальному утверждению на данную роль, и последствия этого он будет ощущать на себе в течение всего процесса разработки и внедрения системы. А если менеджер проекта не участвовал ни в каких дебатах (например, его наняли на работу уже после того, как были приняты все судьбоносные решения по проекту), то он никак не может понять, почему у него оказалось столько врагов среди основных заинтересованных лиц. Разумеется, простого решения этой проблемы не существует, но, по крайней мере, лучше знать о ней заранее.

Кроме этих проблем существует еще один источник политических дебатов: разногласия по поводу статуса проекта, вероятности его успешного завершения и действий, которые необходимо предпринять, если проект выбьется из графика и/или бюджета.

В любом случае политические разногласия могут привести к существенным задержкам в проекте, что само по себе является серьезной проблемой для «безнадежного» проекта. Одним из возможных решений особенно в случае разногласий относительно функциональности и других требований к системе является проведение JAD-сессий (JAD — Joint Application Development), в которых участвуют все заинтересованные лица, пытаясь выработать общее решение в серии коротких, но интенсивных совещаний.

Если это невозможно, менеджеру проекта следует добиться согласия основных заинтересованных лиц на то, что все решения, требующие их согласования или утверждения, будут рассматриваться в течение 24 часов. Хорошим примером такой стратегии является обсуждение «экстремальное» программирование.

ПЕРЕГОВОРЫ В «БЕЗНАДЕЖНОМ» ПРОЕКТЕ

Предположим, что проектная команда подготовила «разумную» оценку плана, бюджета и персонала, требуемых для «безнадежного» проекта, и руководство готово пойти на переговоры перед тем, как принять окончательное решение. Наиболее вероятно, что руководство объявит первоначальную оценку «неприемлемой» и выдвинет свои требования, которые окажутся более жесткими. Как в этом случае поступить менеджеру проекта?

Если контрпредложение со стороны высшего руководства или заказчика содержит только одну «переменную», менеджер проекта может оценить влияние остальных переменных. Например, если первоначальная оценка менеджера проекта заключается в том, что проект потребует 12 месяцев на реализацию, трех разработчиков и бюджет \$200 000, вполне вероятно, что первой реакцией руководства будет: «Вздор! Нам нужно, чтобы система была готова и работала через шесть месяцев!» На первый взгляд, очевидный способ сделать это заключается в увеличении штата и/или бюджета (например, увеличить зарплату, чтобы привлечь более продуктивных программистов).

С другой стороны, Фредерик Брукс уже более 30 лет назад отмечал, что зависимость между количеством разработчиков и временем разработки носит нелинейный характер, поэтому термин «человеко-месяц» был объявлен мифом. Практически все зависимости между основными переменными проекта являются нелинейными и зависящими от времени. Вследствие эффекта «обратной связи», возникающего в результате многих решений руководства, изменение одной переменной (например, увеличение штата) повлияет со временем не только на другие переменные (продуктивность), но и на собственное первоначальное значение. Например, увеличение штата может отрицательно сказаться на моральном состоянии команды, что повлечет за собой текучесть кадров. В результате уменьшится численность персонала.

Переговоры - это игра, которая имеет место во всех проектах. Переговоры в «безнадежных» проектах характеризуются тем, что ставки гораздо выше, эмоции перехлестывают через край, а запросы противоположной стороны (в терминах плана, бюджета и т.д.) обычно доходят до такой крайности, что могут превысить любой «запас прочности». В обычном проекте самый очевидный запас прочности — это сверхурочная работа. Даже если менеджер проекта оказался вынужден принять под давлением жесткий график и урезанный бюджет, успеха все равно можно будет добиться. Для этого надо уговорить проектную команду работать сверхурочно от 10 до 20 ч. в неделю в течение нескольких заключительных месяцев проекта. Эти дополнительные усилия никак не отражаются в официальной статистике, поскольку программисты ничего не получают за сверхурочную работу, поэтому в конце проекта менеджер выглядит героем.

Однако в «безнадежном» проекте небольшого количества сверхурочного времени обычно недостаточно, чтобы достичь тех результатов, которые требует руководство. Кроме того, пользователи и высшее руководство не столь наивны — они знают, что может потребоваться сверхурочная работа и учитывают ее в своих собственных оценках графика выполнения проекта. Таким образом, они лишают менеджера возможности использовать этот свободный ресурс. Правда, менеджеры-ветераны подобных переговоров должны иметь запасные доводы, которые могут пригодиться уже в начале переговоров.

Консультант в области менеджмента Роб Томсет определил общие варианты переговорных игр. Наиболее распространенные из них приведены ниже:

«Удвой и добавь еще» — эта уловка используется в проектах, начиная с египетских пирамид, если не раньше. Используются любые методы оценки, оказавшиеся под рукой, затем полученная «разумная» оценка удваивается и для большей надежности добавляются еще три месяца (или три недели, или три года, в зависимости от масштаба проекта). Главная проблема такой стратегии заключается в том, что она ведет к самому жесткому ограничению, связанному с «безнадежными» проектами: сжатым срокам.

«Обратное удвоение» — руководство может быть осведомлено о попыт-

ках менеджеров проектов «раздуть» свои оценки, используя предыдущую стратегию. Одна из причин такой проницательности заключается в том, что высшее руководство во многих организациях — это бывшие менеджеры проектов в области информационных технологий, поэтому они хорошо разбираются в таких играх. В результате они берут те начальные оценки, которые им дают менеджеры проектов, и автоматически урезают их наполовину. В незавидном положении оказывается неопытный менеджер, который даже не знает, что его подозревают в удвоении своих начальных оценок.

«Угадай число, которое я задумал» — у пользователя или руководителя высокого уровня есть своя «приемлемая» оценка для сроков, бюджета и/или других аспектов переговоров, но они отказываются четко ее сформулировать. Когда менеджер проекта предлагает свою оценку сроков и бюджета, пользователь или руководитель попросту качает головой и говорит «нет». Такой ответ подразумевает: «Это слишком много — попробуй угадать еще раз». Незадачливый менеджер в конце концов (иногда после полудюжины попыток) приходит с нужной оценкой, но поскольку теперь это его оценка, ему впоследствии и придется отвечать за нее.

«Двойной плевок пустышкой» — «пустышкой» (dummy) называется детская соска, а «выплюнуть пустышку» означает, что ребенок настолько расстроен и рассержен, что выплевывает свою соску. Томсет использует это как метафору, чтобы описать такую ситуацию в процессе переговоров: руководитель высокого уровня впадает в буйное неистовство, когда менеджер проекта в первый раз докладывает свои предложения по плану и бюджету. Дисциплинированный менеджер поспешно удаляется, затем снова возвращается с пересмотренной оценкой, а руководитель снова закатывает истерику. Получается «двойной плевок пустышкой». Идея заключается в следующем: запугать и затерроризировать менеджера до такой степени, чтобы он согласился с чем угодно, лишь бы избежать еще одной вспышки гнева.

«Испанская инквизиция» — такое случается, когда менеджер проекта приходит на совещание руководителей высшего уровня, не зная, что от него

потребуется дать «немедленную оценку» проекту. Представьте себе комнату, полную ворчливых вице-президентов, которые пристально смотрят на вас, в то время, как директор грозным тоном спрашивает: «Итак, когда же, по вашему мнению, мы получим эту систему? Я уже доложил всему руководству, что она будет готова к середине марта, вы же не собираетесь подвести меня, не так ли?» Если у вас хватит смелости возразить, что более реальный срок — середина ноября, тогда инквизиторы обрушатся на вас с вопросами относительно вашего интеллекта, полномочий, лояльности и, возможно, даже религиозных убеждений.

«Игра на понижение» - часто имеет место в ситуации, когда организация — разработчик ПО проигрывает своим конкурентам борьбу за право разработки системы для организации-клиента. Эта игра очевидна: заказчик (или в некоторых случаях представитель службы маркетинга организации-разработчика) говорит менеджеру проекта, что один из претендентов внес свои предложения о коротком сроке разработки и/или более скромном бюджете. Это вынуждает менеджера проекта не только ответить на вызов конкурента (который может быть вполне реальным, а может, и нет), но и превзойти его, чтобы повысить свои шансы на заключение контракта. Один из вариантов такой игры — клиент дает понять, что он не исключает возможность, что проект вообще не состоится; при этом организация-разработчик, которая во что бы то ни стало стремится заполнить этот проект, постарается выдвинуть такие заманчивые для клиента предложения, от которых он не сможет отказаться.

«Китайская пытка водой» — данная игра заключается в том, что плохие новости преподносятся заказчику и (или) высшему руководству небольшими порциями. Допустим, разумная оценка срока выполнения проекта, сделанная менеджером, составляет 12 месяцев. По его мнению, при помощи сверхурочной работы и разного рода чудес проект можно выполнить за 6 месяцев, однако руководство настаивает на 4 месяцах. Менеджер нехотя уступает и устанавливает для проекта последовательность «контрольных точек». Например, новая версия прототипа системы должна предъявляться заказчику каждую неделю. Первое

предъявление окажется опоздавшим на один день, однако менеджер докажет, что для данной работы задержка может составлять от 14 до 20% (в зависимости от того, сколько дней в неделю работает команда — 5 или 7); отсюда, по его мнению, следует, что срок разработки заключительной версии системы также может быть отодвинут на 14—20%. На такой ранней стадии проекта руководство отказывается пойти на уступки, но когда вторая контрольная точка также окажется сдвинутой на день (что означает общую задержку в два дня в течение двух недель), менеджер вновь повторит свои аргументы. Это похоже на китайскую пытку водой — одной плохой новости недостаточно, но все вместе могут оказаться роковыми.

«Спрятанное качество» — это одна из наиболее хитрых игр. В ней могут участвовать (в конструктивной или деструктивной манере) хорошо информированные менеджеры проектов, менеджеры высокого уровня по информационным технологиям и/или заказчики. Менеджер проекта может предоставить пользователю бесконечно большое число программ за нулевое время, пока их не нужно реально эксплуатировать. Разумеется, было бы глупо предлагать такой экстремальный сценарий. Однако суть заключается в том, что качество ПО (выражаемое в количестве ошибок, переносимости, сопровождаемое™ и т.д.) — это одно из «измерений» проекта, которое нужно учитывать во время переговоров по срокам, затратам, персоналу и другим ресурсам. Некоторые заказчики слишком неопытны, чтобы понимать это, а другие не собираются принимать в расчет длительную перспективу. В самом лучшем варианте такая «игра» отражает стратегию «достаточно хорошего» (good-enough) ПО, которая описана ниже. В худшем виде она носит такой же жульнический характер, как и некоторые другие упомянутые выше политические игры.

Самый важный совет Томсета - не попасть в ловушку «скоропалительных» оценок проекта. Наихудшую разновидность такой ловушки представляет собой игра «испанская инквизиция». Однако существуют и не такие заметные ловушки. Преднамеренно или нет, но менеджера проекта часто ставят в положение, когда от него требуется быстро дать «грубую оценку» времени и количеству

персонала, требуемым для реализации каких-либо частей проекта, и эти оценки могут превратиться в жесткие, не подлежащие изменению требования. В подобной ситуации, когда имеется некоторое время на выполнение формальной оценки, очень важно выразить оценки в терминах «уровней достоверности» или в некотором диапазоне «плюс-минус». Если абсолютно отсутствуют данные для детальной оценки и если в «безнадежном» проекте используется совершенно новая технология и участвует персонал неизвестной квалификации, то будет благоразумным сказать: «Проект, вероятно, потребует от трех до шести месяцев» или «Я думаю, что мы закончим проект через шесть месяцев плюс-минус 50%».

Если менеджер проекта оказался не в состоянии добиться от заказчика или высшего руководства понимания той неопределенности, которая связана с планом или бюджетом «безнадежного» проекта, он должен всерьез задуматься об отставке; то же самое касается и технических специалистов проектной команды. Но это только один аспект неудачных переговоров; как следует поступить менеджеру, если он на 100% уверен, что продиктованный политическими соображениями срок в шесть месяцев явно недостаточен? Как следует ему поступить, если он на 100% уверен, что проектная команда должна состоять как минимум из трех человек, а руководство дает только двух?

Если высшее руководство угрожает уволить менеджера в случае провала «безнадежного проекта» или он (что практически одно и то же) категорически не согласен с нереальными планами, следует в ответ проявить такое же хладнокровие и настойчивость в своих требованиях. Может быть, и не стоит сильно настаивать на изменении плана, однако следует проявить гораздо большую требовательность, когда речь пойдет о формировании проектной команды. И, определенно, нужно быть настойчивым в том, что касается игнорирования или отмены административных и бюрократических правил и процедур, которые могут гарантировать провал «безнадежного» проекта.

Лидер проекта, который заботится о своих сотрудниках, не должен обещать им золотые горы и скрывать истинное положение вещей. Он честно ска-

жет о том, какие усилия от них потребуются и каковы шансы на успех. Программисты вовсе не так уж глупы. Самые опытные из них прекрасно чувствуют, когда им «вешают лапшу на уши». Большинство из них не желают участвовать в разных играх вокруг проекта, зная, что в случае кризиса вся его тяжесть ляжет именно на них.

Если же менеджер проекта убедился, что цели проекта недостижимы, но проект в любом случае должен продолжаться, то для него очень важно донести до остальных участников команды, что они оказались в «безнадежном» проекте. Некоторые могут согласиться с любым вариантом, и менеджеру важно понимать причины, толкнувшие их на это; а другие могут отказаться от участия в проекте.

7.5.

ЧЕЛОВЕЧЕСКИЙ ФАКТОР

В «БЕЗНАДЕЖНЫХ» ПРОЕКТАХ

Джеральд Вейнберг, автор книги по психологии программирования, сказал, что у каждого проекта есть три проблемы: люди, люди и люди. Это как нельзя лучше характеризует «безнадежные» проекты: если имеются ограниченные средства, то большинство менеджеров проекта скажет, что для повышения шансов на успех в проекте их надо потратить на «человеческие ресурсы». Это не означает, что сплоченная команда талантливых людей всегда сумеет справиться с несовершенными процессами, устаревшими инструментами, нерасположенными к сотрудничеству пользователями, враждебно настроенными заинтересованными лицами, недостаточным бюджетом и крайне сжатыми сроками. Но скорее следует сделать ставку на такую команду, чем рассчитывать на то, что команда среднего уровня, вооруженная мощными средствами программирования и передовой технологией, сможет справиться со всеми этими проблемами.

В рекомендациях менеджеру проекта нет ничего нового. Будьте настойчивы в отстаивании права формировать собственную команду. Можно ожидать, что команда будет работать сверхурочно. Однако при этом надо помнить, что они участвуют в марафоне и могут пробежать в спринтерском темпе только последние 100 метров. Если работа над проектом закончится успешно, добейтесь для них щедрого вознаграждения. Но не дразните их обещаниями будущих экстраординарных наград, потому что это только собьет их с толку. Приложите максимум усилий для создания спаянной и дружной команды, готовой к сотрудничеству. Важно, чтобы все члены команды имели необходимую квалификацию, а также умели общаться друг с другом. В основном это все, что касается человеческого фактора в «безнадежном» проекте.

К сожалению, этого явно недостаточно для большинства менеджеров «безнадежных» проектов, поскольку они работают в организациях, пренебрегающих человеческим фактором даже в нормальных проектах. Может показаться, что в таких условиях «безнадежные» проекты обречены на провал, но иногда получается как раз наоборот. Как отмечалось выше, менеджер может быть вынужден согласиться с нереальными сроками или бюджетом, но в качестве компенсации настаивать на принятии предложенных им решений по вопросам персонала (на праве нанимать нужных исполнителей, соответствующим образом вознаграждать их и обеспечивать им нормальные условия работы).

«Безнадежный» проект может восприниматься как угроза тем, кто желает сохранить бюрократическое статус-кво. Менеджер проекта, обходя бюрократические ограничения с помощью непосредственных распоряжений высшего руководства, должен быть готов к тому, что приобретет себе вечных врагов в лице кадровых работников и других администраторов. Тем не менее, если «безнадежный» проект будет иметь громкий успех, он может послужить катализатором к изменению кадровой политики в последующих проектах.

Первая отличительная особенность «безнадежного проекта» - умение правильно сформировать проектную команду. Существуют четыре наиболее распространенных стратегии формирования команды «безнадежного проекта»:

- нанять суперпрограммистов и предоставить им свободу действий;
- настаивать на привлечении команды, которая готова к «невыполнимой миссии» и имеет опыт совместной работы;
- набрать команду из «простых смертных», но при условии, что они будут знать, на что идут;
- взять любых сотрудников, которых дадут, и сделать из них команду «невыполнимой миссии».

Первая стратегия выглядит весьма заманчиво. Предполагается, что суперпрограммисты будут невероятно изобретательны, чтобы предложить для «безнадежного» проекта новые решения. С другой стороны, это риск, поскольку суперпрограммисты обычно являются эгоистами и могут не ужиться друг с другом. Кроме того, для многих организаций такая стратегия проблематична, поскольку руководство не желает платить суперпрограммистам такую высокую зарплату, какую они требуют.

Вторая стратегия почти наверняка будет идеальной для большинства организаций, поскольку она не требует привлечения суперпрограммистов. Однако, если организация предпринимает свой первый «безнадежный» проект, такой команды просто не существует. Если же такие проекты ранее имели место, то их команды в дальнейшем, скорее всего, распались и прекратили свое существование. Таким образом, стратегия сохранения в целости проектной команды успешного «безнадежного» проекта должна, как правило, планироваться заранее как корпоративная стратегия, исходя из предположения, что такие проекты могут появиться в будущем.

Третья стратегия по вполне очевидным причинам является наиболее распространенной. В большинстве организаций нет суперпрограммистов и нет тех, кто уцелел в предыдущих «безнадежных» проектах. Следовательно, команда каждого нового проекта комплектуется заново. Участники команды вполне компетентны и, возможно, выше среднего уровня разработчиков в данной организации, однако от них нельзя ожидать сотворения чудес. Для данной стратегии жизненно важно, чтобы участники команды понимали, на что они идут;

знали, что им придется совершать необычайные подвиги в разработке ПО.

Последней стратегии следует избегать при любых затратах. Если проект превращается в «свалку» для сотрудников, которых не хотят брать ни в какие другие проекты, он почти наверняка будет самоубийственным.

Центральный вопрос формирования команды «безнадежного» проекта: до какой степени менеджеру проекта следует настаивать на своем праве принимать кадровые решения? Как было отмечено выше, большинство менеджеров вынуждены примириться с фактом, что они не получают карт-бланш, чтобы нанимать самых талантливых программистов в мире. Кроме того, существующая в организации политика может не позволить менеджеру по собственной воле, не спрашивая разрешения, привлечь в проект лучших сотрудников организации, поскольку они либо уже участвуют в других важных проектах, либо их отстаивают другие менеджеры. Но есть один аспект, который менеджер должен отстаивать как свое абсолютное право: это право наложить вето на попытку других руководителей навязать проектной команде неудобного им сотрудника. В противном случае уровень риска в проекте может повыситься до недопустимых пределов.

Одной из ключевых характеристик команды, которой менеджер проекта должен уделять максимальное внимание, является ее отношение к проекту. Существуют уровни или степени мотивации. Можно рассчитывать на то, что разработчик проявит определенную степень мотивации в нормальном проекте, но «безнадежные» проекты требуют более высокой степени, поскольку участникам команды месяцами придется выдерживать изнурительную работу, политическое давление и технические трудности.

Было бы хорошо решить проблему мотивации, посулив большие суммы денег каждому участнику проектной команды. Деньги, выгода, комфорт и тому подобное являются факторами «гигиены» — их отсутствие вызывает неудовлетворенность, однако они не могут заставить людей полюбить свою работу и дать им необходимые внутренние стимулы. Что действительно может стать стимулами, так это ощущение значительности достигнутых результатов, гор-

дость за хорошо выполненную работу, более высокая ответственность, продвижение по службе и профессиональный рост, все то, что обогащает труд.

Эта оценка достаточно точно характеризует нормальные проекты. В большинстве «безнадежных» проектов деньги все же играют важную роль. Многие начинающие компании предпринимают безумные и бесперспективные проекты в надежде на то, что им удастся разработать какое-нибудь совершенно новое приложение для нового технического устройства и продать миллион его копий на рынке. Если участники проектной команды являются акционерами и собираются участвовать в распределении прибыли, то денежное вознаграждение, очевидно, составляет весьма существенную часть мотивации их участия в проекте. Действительно, многие компании намеренно придерживают зарплату на 20—30% ниже преобладающего на рынке уровня, заинтересовывая своих специалистов возможностью серьезного участия в акционировании и других формах распределения будущей прибыли. Эта стратегия направлена не только на повышение мотивации, но и на снижение расхода наличности, поскольку зарплаты сотрудников зачастую составляют единственные самые крупные затраты начинающей компании.

Если «безнадежный» проект достаточно важен для предприятия, оно найдет способы зарезервировать значительный премиальный фонд для поощрения проектной команды при условии удачного завершения проекта в срок. Допуская возможность такого вознаграждения, не следует забывать, что 20%-ное повышение зарплаты имеет гораздо большее значение для молодого программиста, чем для опытного и квалифицированного программиста.

Размер премии напрямую не связан с количеством времени, затрачиваемым проектной командой на выполнение проекта. В некоторых организациях высшее руководство пытается соблазнить проектную команду двойной премией (при отставании проекта от графика), поскольку руководство, очевидно, верит, что это заставит людей работать вдвое больше. Однако, если участники команды уже работают по 18 часов в день, законы физики не позволят работать вдвое больше даже самым рьяным из них.

Что касается проектов, за выполнение которых невозможно получить большие премии, менеджеру проекта важно не забывать о том, что существуют разнообразные виды нематериального вознаграждения (дополнительный отпуск, свобода действий, создание полностью оснащенной домашней компьютерной системы и др.), и с их помощью можно стимулировать участников проекта. Это также справедливо и для нормальных проектов, но для «безнадежных» особенно важно.

В то время как премии и дополнительные отпуска являются положительным стимулом, необходимость сверхурочной работы на протяжении проекта считается отрицательным. Тем не менее, в «безнадежных» проектах она почти всегда неизбежна; это единственный способ, дающий менеджеру проекта хоть какой-то шанс уложиться в жесткий график. Сверхурочным временем необходимо правильно распоряжаться, чтобы избежать отрицательного воздействия на команду и не поставить под угрозу успех проекта. Один из способов это сделать — убедиться, что высшее руководство знает реальную стоимость сверхурочной работы.

До тех пор, пока участники проектной команды не будут иметь такие же хорошие возможности участия в акционерном капитале компании, как высшее руководство, любые другие формы компенсации за участие в «безнадежном» проекте нельзя всерьез считать вознаграждением (в положительном смысле). В то время как менеджер проекта редко распоряжается такого рода компенсацией, то, что реально можно сделать — это немедленно оплачивать сверхурочную работу. При этом люди, почти полностью отдающие себя проекту, получают хоть какую-то отдачу, а тех, кому не мешало бы знать точную стоимость проекта (высшее руководство и др.), можно будет заставить ее узнать (посредством бюджета).

Независимо от того, получают участники команды компенсацию за сверхурочную работу или нет, большой ошибкой будет отсутствие учета этого времени. Даже если с точки зрения бухгалтерии именно так оно и есть, менеджер не должен рассматривать сверхурочное время как свободное. Даже если участ-

ники команды способны всю жизнь без усталости работать по 18 часов в день, для менеджера важно фиксировать, сколько таких сверхурочных часов затрачивается в течение работы над проектом. Это единственный способ точно измерить производительность команды и вероятность своевременного достижения каждой промежуточной контрольной точки в графике проекта.

Каждому известно, что люди не в состоянии работать всю жизнь по 18 часов в день; даже если они будут очень стараться, все равно скоро устанут. А когда они устают, то становятся раздражительными и вспыльчивыми, работают менее продуктивно и делают гораздо больше ошибок. Это может отрицательно повлиять на работу над проектом. Поэтому менеджер должен четко понимать, когда следует попросить команду поработать сверхурочно, а когда не следует этого делать.

Одна из опасностей, которые менеджер проекта должен предвидеть — это чрезмерная добровольная сверхурочная работа той части молодых энтузиастов, которые не представляют пределов своих собственных возможностей и не задумываются о побочных эффектах работы до изнеможения. Производительность труда действительно может расти в первые 20 часов сверхурочной работы (благодаря повышенному содержанию адреналина в крови, концентрации внимания и т.д.). Но рано или поздно каждый достигнет критической точки, после чего начнется спад, поскольку возрастет количество ошибок и ослабнет концентрация внимания. Таким образом, наступит момент, когда участник команды будет демонстрировать «отрицательную» продуктивность, поскольку усилия, затрачиваемые на исправление ошибок и дефектов в программах и их переписывание, будут сводить к нулю всю выполненную работу. Менеджер может относительно безболезненно настаивать на 60-часовой рабочей неделе. При работе от 60 до 80 ч следует четко установить пределы индивидуальных возможностей каждого разработчика; при 80—90-часовой рабочей неделе менеджер должен отправлять разработчиков домой и заставлять их отдыхать.

В «безнадежных» проектах очень важны вопросы, связанные с человеческими ресурсами — природа и степень взаимодействия между менеджером

проекта и остальной командой. Идеальная ситуация — когда у менеджера проекта нет секретов от команды, и каждый участник команды знает о проекте все. Это означает, что каждый участник команды располагает актуальной информацией относительно состояния проекта, первоочередных приоритетов, рисков, ограничений, политики и т.д.

При таких условиях в команде может быть создана атмосфера взаимного доверия и доброжелательности. Если участники команды прилагают экстраординарные усилия к работе над проектом и приносят в жертву личную жизнь, для них было бы крайним разочарованием обнаружить, что менеджер проекта скрывает от них важную информацию или играет за их спиной в политические игры. Поскольку в «безнадежном» проекте все процессы протекают интенсивнее и быстрее, чем в нормальном, выше вероятность, что участники команды обнаружат утаивание информации или непристойные политические игры вокруг них.

Очевидный контраргумент против такой философии заключается в том, что менеджер должен оберегать команду от посторонних помех, особенно от каждодневных мелочных политических игр, которые происходят вокруг проекта. В большинстве случаев участники команды по достоинству оценивают возможность освободиться от политики; однако они также нуждаются в уверенности, что в ответ на прямой вопрос менеджер не станет темнить или лгать им. В большинстве проектов, нормальных или «безнадежных», регулярно проводятся совещания, на которых говорится о состоянии проекта и поднимаются острые вопросы; если участники команды всегда могут узнать о том, что происходит в проекте, тогда они спокойно смогут на 99% сконцентрироваться на своей непосредственной работе.

Уровень коммуникации между отдельными участниками команды весьма важен, особенно если они ранее не работали вместе. Нужно, чтобы не было постороннего вмешательства во внутрикомандное взаимодействие. В этом случае можно будет поддерживать честный и откровенный обмен информацией. Для большинства сегодняшних проектов это обязательно подразумевает наличие

электронной почты и различных средств групповой работы.

Открытые и честные взаимоотношения являются важной составляющей процесса формирования эффективной команды. Подбор психологически совместимых исполнителей — другая ключевая составляющая. Крайне важно, чтобы менеджер проекта имел свободу действий при выборе участников команды.

Еще одна составляющая заключается в концепции командных ролей. Многие менеджеры проекта сосредотачиваются на чисто «технических» ролях, таких, как проектировщики баз данных, специалисты по сетям, эксперты по пользовательскому интерфейсу и т.д. Все они важны, но нужно подумать и о ролях «психологического» плана, которые могут играть один или более участников команды. Эти роли присутствуют и в нормальных проектах, однако в «безнадежных» они приобретают особую важность. Роб Томсет определил восемь ключевых ролей в проекте следующим образом.

Председатель (chairperson) - выбирает путь, по которому команда движется вперед к общим целям, обеспечивая наилучшее использование ее ресурсов; умеет обнаружить сильные и слабые стороны команды и обеспечить наибольшее применение потенциала каждого участника команды. Можно думать, что таким человеком является, как правило, официальный руководитель проекта; однако в самоуправляемых командах им может быть любой человек.

Оформитель (shaper) — придает законченную форму действиям команды, направляет внимание и пытается придать определенные рамки групповым обсуждениям и результатам совместной деятельности. Такой человек может иметь официальную должность «архитектора» или «ведущего проектировщика», но главное то, что эта роль «воображаемая». В безнадежном проекте особенно важно иметь единое и четкое представление о проблеме и ее возможном решении.

Генератор идей (plant) — выдвигает новые идеи и стратегии, уделяя особое внимание главным проблемам, занимается поиском новых подходов к решению проблем, с которыми сталкивается группа. Это человек, который пытается внедрять в команде радикальные технологии, искать новые решения тех-

нических задач.

Критик (monitor-evaluator) — анализирует проблемы с прагматической точки зрения, оценивает идеи и предложения таким образом, чтобы команда могла принять сбалансированные решения. В большинстве случаев такой человек поступает как скептик, уравнивая оптимистические предложения оформителя и генератора идей. Критик хорошо знает, что новые технологии отнюдь не всегда работают, обещания поставщиков о возможностях новых средств и языков программирования иногда не сбываются и все может пойти не так, как было задумано.

Рабочая пчелка (company worker) — превращает планы и концепции в практические рабочие процедуры, систематически и эффективно выполняет принятые обязательства. Другими словами, в то время как оформитель придает законченную форму крупным технологическим решениям, генератор идей предлагает радикальные новые решения, а критик занимается поиском изъянов и недостатков в этих предложениях, рабочая пчелка — это тот человек, который работает, не привлекая внимания, и выдает «на гора» тонны кода. Очевидно, любой «безнадежный» проект нуждается по крайней мере в паре таких человек, но сами по себе они не способны принести успех проекту, поскольку не обладают необходимой широтой кругозора.

Опора команды (team worker) — поддерживает силу духа в участниках проекта, оказывает им помощь в трудных положениях, пытается улучшить взаимоотношения между ними и в целом способствует поднятию командного настроения. Другими словами, такой человек выполняет в команде роль дипломата. Им может быть и менеджер проекта, однако им может быть также любой из участников команды, относящийся более внимательно к своим коллегам. Эта роль особенно важна в «безнадежных» проектах, поскольку команда зачастую испытывает сильный стресс.

Добытчик (resource investigator) — обнаруживает и сообщает о новых идеях, разработках и ресурсах, имеющихся за пределами проектной группы, налаживает внешние контакты, которые могут оказаться полезными для команды, и

проводит все последующие переговоры. Он всегда знает, где отыскать бесхозный ПК, свободный конференц-зал, дополнительный рабочий стол или любой другой ресурс, в котором нуждается команда. Такие ресурсы могут быть добыты по официальным каналам или нет; но даже если их можно достать нормальным способом, приходится ждать выполнения всех бюрократических процедур. «Безнадежный» проект не может ждать долго и не может позволить, чтобы вся работа застопорилась из-за того, что какой-то помощник вице-президента не разрешает воспользоваться единственным в организации свободным конференц-залом. Командный добытчик имеет много друзей и связей в своей организации, с помощью которых можно выпросить или одолжить необходимые ресурсы.

Завершающий (completer) — поддерживает в команде настойчивость в достижении цели, активно стремится отыскать работу, которая требует повышенного внимания, и старается, насколько возможно, избавить команду от ошибок, связанных как с деятельностью, так и с бездеятельностью. Такой человек играет доминирующую роль во время тестирования системы на завершающей стадии жизненного цикла проекта, однако его роль на более ранних стадиях тоже важна.

Если процесс формирования такой команды протекает успешно, это бывает заметно по некоторым внешним признакам. В преуспевающих командах обычно бывает сильное ощущение общности интересов и гордости за свою команду, а также чувство, что они в состоянии хорошо выполнять свою работу и получать от этого удовольствие. Если организация не в состоянии обеспечить создание такой сплоченной команды, это может привести к тому, что принимается сознательное или бессознательное решение плыть по течению и не совершать никаких действий для создания сплоченной и целостной команды.

Проблемы создания нормальных условий для работы уже много лет обсуждаются среди разработчиков ПО. Если разработчики работают в достаточно тихом помещении, вероятность появления у них ошибок уменьшается на одну треть по сравнению с теми, кто работает в шумном офисе и постоянно отвлека-

ется на посторонние дела. Например, условия работы в Microsoft и в большинстве других компаний Кремниевой Долины достаточно цивилизованные - отдельные помещения с закрытыми дверями, наличие содовой, сока и других напитков, «постоянный» телефонный номер, который остается за программистом даже в том случае, если он перебирается в другое помещение.

Многие разработчики ПО работают в банках, страховых компаниях, правительственных учреждениях, промышленных предприятиях и сотнях других компаний, где до сих пор смотрят на ПО, как на «накладные» расходы. Поэтому им приходится работать не в нормальных офисах, а в отгороженных клетушках, где практически нет возможности сконцентрировать свои умственные усилия на решении какой-либо проблемы. Звучит музыка, не прекращаются телефонные звонки, кричат люди и нет никакого спасения от кого угодно (от курьера до директора), кто может засунуть голову в твою комнату и отвлечь от работы. Пока сотрудники будут тесниться в шумных и непригодных помещениях, нельзя говорить о сколько-нибудь продуктивной работе.

Если менеджер проекта достаточно изобретателен, стоит попытаться обеспечить подходящие условия для работы таким образом, чтобы хозяйственная служба вообще не знала о том, что происходит. Для этого есть несколько возможностей.

Лобовая атака — если у проекта есть защитник и/или владелец, которые крайне заинтересованы в его успехе, нужно объяснить им, насколько важно, чтобы проектная команда работала в хороших условиях. Если защитник проекта является руководителем высокого уровня, то организовать временный переезд команды в более подходящее помещение будет несложно.

Самовольный захват — не спрашивая ни у кого разрешения, занять, какое-нибудь пустое помещение. Такой захват обеспечит 90% успеха в борьбе за условия работы; пока бюрократы будут ругаться, спорить и отправлять в разные стороны гневные послания, может быть, уже удастся закончить проект и незаметно удалиться на прежнее место.

Дистанционный доступ — разрешить всем работать дома и организовать

еженедельные рабочие совещания в ближайшем «Макдональдсе» (в 9 часов утра, когда почти нет посетителей). Пока кто-нибудь обнаружит исчезновение команды, может пройти не одна неделя. Для дополнительного отвлечения внимания можно посадить чучела за столы, которые обычно занимала проектная команда; руководству понадобится достаточно много времени, чтобы отличить их от других сотрудников, сидящих в офисе.

Переход на работу в ночную смену — это более радикальный вариант, однако он может оказаться достаточно эффективным, если большая часть работы может выполняться без взаимодействия с пользователями.

Преграды и заслоны — если команда работает в обычном «открытом» офисе и упомянутые выше стратегии неприменимы, тогда надо постараться сделать все возможное, чтобы собрать команду в смежных помещениях и забаррикадироваться от остальной толпы в офисе.

Некоторые из этих акций могут спровоцировать более резкую ответную реакцию корпоративной бюрократии, чем другие; команда и ее менеджер должны решить, какая стратегия будет наиболее эффективной. Борьба с бюрократией таким способом - не для робкого десятка; но ведь и сами «безнадежные» проекты тоже не для робкого десятка. Если менеджер «безнадежного» проекта не проявляет желания бороться и отстаивать право на нормальные условия работы, то с какой стати проектная команда должна идти на экстраординарные жертвы ради организации и менеджера проекта?

Талантливых исполнителей, сплоченной команды и хороших условий для работы все же недостаточно, чтобы гарантировать успех «безнадежного» проекта. С другой стороны, их отсутствие приведет к провалу проекта. Хорошо организованные процессы разработки и хорошая технология также являются важными составляющими успеха; однако самая главная составляющая — это люди.

7.6.

ПРОЦЕССЫ В «БЕЗНАДЕЖНЫХ» ПРОЕКТАХ

В ситуации «безнадежного» проекта применима концепция приоритетности — при нехватке времени и ресурсов команда откажется от тех методов, которые она сочтет бесполезными или несущественными (например, детальные спецификации в структурном анализе), и примет только подходящие для себя. Распределение дефицитных ресурсов (самым дефицитным из которых обычно является время) должно осуществляться таким образом, чтобы извлечь из этого наибольшую выгоду.

Почти во всех «безнадежных» проектах приходится устанавливать приоритеты, разделяя все требования к системе на различные категории (см. подразд. 1.5). При этом все акционеры и заинтересованные лица должны принять согласованное решение относительно того, какие требования следует отнести к категориям «необходимо», «желательно», «хотелось бы» и «хотелось бы, но не в этот раз». Разумеется, если владелец проекта категорически настаивает на том, чтобы все требования были обязательно выполнены, дальнейшее обсуждение будет пустой тратой времени. Если акционеры и заинтересованные лица не могут достичь консенсуса по поводу отнесения требований к той или иной категории, то проектная команда, пытаясь удовлетворить всех, в результате окажется парализованной из-за отсутствия необходимых ресурсов.

К сожалению, в большинстве организаций нет дисциплины, опыта и политической силы, чтобы определить приоритет требований в самом начале проекта. Политические баталии вокруг «безнадежного» проекта могут сделать почти невозможным достижение консенсуса по приемлемым для всех приоритетам. Только когда станет понятно, что проект «безнадежен», противоборствующие стороны придут к соглашению, которое им надо было бы достичь в самом начале проекта;

В «безнадежном» проекте необходимо уделить особое внимание такому аспекту жизненного цикла ПО, как управление требованиями. В самом деле, в экстремальной ситуации проектная команда даже не будет документировать ни одно из пользовательских требований. В свое оправдание они говорят, что для документирования требуется слишком много времени, требования часто меня-

ются и, кроме того, пользователи сами точно не знают, что им нужно. Таким образом, команда обычно полагается на методы и средства прототипирования, с помощью которых можно наглядно продемонстрировать всю важную проектную работу, а также выявить реальные требования к системе. С точки зрения «приоритетности» это порождает невозможность организованно управлять требованиями.

Чтобы достичь успеха, проектная команда должна прийти к соглашению, какие процессы будут формализованы (например, контроль исходного кода, управление изменениями и управление требованиями) и какие будут неформальными (например, проектирование пользовательского интерфейса). Бессмысленно требовать в обязательном порядке выполнения какого-либо процесса, если никто не собирается ему следовать.

Менеджер «безнадежного» проекта должен безоговорочно настаивать на выполнении тех процессов, которые он считает принципиально важными, например, процесса управления изменениями.

Формальные подходы (типа SEI-CMM, ISO-9000 или внедрение новых технологий анализа и проектирования) должны иметь место где-нибудь за пределами «безнадежного» проекта. Внедрение таких процессов имеет смысл как часть долговременной корпоративной стратегии. Оно должно начинаться с выполнения пилотного проекта (но не «безнадежного»), сопровождаясь организацией необходимого обучения. Если все это уже сделано, и другие разработки ПО в организации уже выполняются на третьем уровне SEI-CMM, то официально принятые в организации процессы следует усовершенствовать, чтобы сделать их пригодными для использования в «безнадежном» проекте.

Существует еще один аспект в разработке ПО, который порождает трудности в «безнадежных» проектах: неявно подразумеваемое требование достижения идеального качества. Обычно оно выражается в терминах количества дефектов, независимости от платформы, гибкости и др. Даже в нормальных проектах достаточно трудно удовлетворить всем этим требованиям, а в «безнадежных» сделать это просто невозможно. Вместо этого проектная команда

должна прийти к решению (и по возможности согласовать его с акционерами и заинтересованными лицами) относительно того, какое качество является достаточно хорошим.

Важность такого решения объясняется тем, что достижение абсолютного качества съедает все ресурсы проекта, особенно время. Если нужно разработать сертифицированную, не содержащую ошибок программу и математически доказать ее корректность, на это потребуется время. Для этого нужны более талантливые и способные специалисты, а их недостаточно в проектной команде. Кроме того, одному или более участникам команды придется расходовать дополнительную энергию, следовательно, они не смогут работать над другими задачами. Выполнение таких требований, как надежность, переносимость и сопровождаемость, невозможно без компромиссов, и это необходимо учитывать в процессе определения приоритетов требований.

Эти требования не являются необходимыми, практичными или желательными в большинстве «безнадежных» проектов. Достичь такого совершенства невозможно даже в нормальных проектах, хотя в них нет таких жестких ограничений на время, бюджет и людские ресурсы. Что же касается «безнадежных» проектов, то пользователям в действительности нужна система, которая достаточно дешева, достаточно производительна, обладает необходимыми возможностями, достаточно устойчива и будет готова достаточно скоро — вот в чем заключается их определение «достаточно хорошего» ПО.

Однако отсутствие стандартов и технологии само по себе может превратить проект в «безнадежный». Не следует воспринимать сказанное выше как предлог для отказа от каких-либо процессов или методов вообще. Проблема заключается в том, чтобы отыскать те процессы, которые действительно работают и которым команда будет следовать естественным образом и бессознательно. Последнее особенно важно: команда испытывает такой стресс и давление, что должна многое делать чисто инстинктивно. Следует запомнить только одно — приоритетность.

ДИНАМИКА ПРОЦЕССОВ

Во многих «безнадежных» проектах наиболее серьезные проблемы носят не столько технический характер, сколько политический, социальный, культурный и человеческий. И хотя существует ряд хороших подходов, ориентированных на учет человеческого фактора (набор лучших специалистов, их мотивация и организация продуктивных коллективов), проблема в том, что все это должно работать в рамках некоторой реальной организации, которая может оказаться настолько сложной, что непонятно, как она вообще работает. На самом деле, зачастую она и не работает, при этом деятельность проектной команды оказывается парализованной из-за непонятных и неожиданных решений и политики руководства.

Это происходит, несмотря на героические усилия, предпринимаемые многими организациями для упорядочения и усовершенствования своих процессов, используя RAD, XP, SEI-CMM и т.д. Многие из этих усилий затрачиваются впустую, поскольку не понимается динамика процессов (особенно временные задержки и циклы обратной связи) и игнорируются неформальные процессы, играющие главную роль в проектах. По этой причине в последние несколько лет моделирование и имитация динамики таких процессов вызывают особый интерес.

Динамика систем — это не новое понятие, ему уже несколько десятков лет. В США одни из первых работ в этой области выполнил в начале 1960-х годов в Массачусетс ком технологическом институте Джей Форрестер. С тех пор многое изменилось: есть ПК, позволяющие работать с имитационной моделью в интерактивном режиме, в отличие от пакетных систем с недельным циклом обработки; имеются языки визуального моделирования, и применяются общие принципы динамики систем для решения небольших, конкретных задач.

Один из примеров таких конкретных задач — процессы создания ПО.

Начиная с первых работ Тарика Абдель-Хамида¹ в Массачусетском технологическом институте в начале 1990-х годов, исследователи и специалисты в области совершенствования процессов экспериментируют с динамическими моделями, пытаясь глубже разобраться в процессах создания ПО. Если это поможет менеджеру безнадёжного проекта лучше понять, что происходит в его проекте, то вероятность успеха может повыситься.

Многие воспринимают слово «модель» в контексте процессов разработки ПО, как набор графических абстракций ПО — например, диаграммы UML, блок-схемы, диаграммы «сущность-связь» и др. Однако в реальном мире менеджеры проектов мало что используют из всей этой кухни в своей повседневной деятельности. Такие модели используются скорее для планирования и согласования общей стратегии, а для принятия ежедневных оперативных решений используются другие модели, наиболее важными из которых являются мысленные (mental) и табличные (spreadsheet) модели.

Мысленную модель следует буквально понимать как модель, которая существует только в голове. Она может основываться на опыте или интуиции или на мифах и легендах, на нее оказывают влияние политика и весь спектр человеческих эмоций. Однако главная особенность мысленной модели заключается в том, что ее невозможно выразить в письменной форме; во многих случаях ее даже невозможно озвучить, это глубоко личная практика менеджера, которой он руководствуется в своей работе.

Представьте себе такую ситуацию: половина проекта позади, утром программисты приходят в офис к менеджеру и хмуро заявляют: «Не знаем, как это случилось, но когда мы проснулись сегодня утром, то обнаружили, что отстаем от графика на полгода!» Простая мысленная модель неопытного менеджера проекта предлагает следующий план действий — немедленно нанять побольше людей. Почему? — Потому что его мысленная модель говорит примерно следующее: «Нам необходимо выполнить больше работы в ограниченное время,

¹ Tarek Abdel-Hamid, Stuart E. Madnick. Software Project Dynamics: An Integrated Approach. Englewood Cliffs, NJ: Prentice Hall, 1991.

поскольку мы не можем отодвинуть срок окончания проекта. Больше людей сделают больше работы, поэтому, чтобы выполнить всю требуемую работу, надо нанять дополнительных сотрудников».

Разумеется, у опытных менеджеров совершенно другая мысленная модель. Она опирается на их собственный опыт и Закон Брукса, который он сформулировал в своей знаменитой книге «Мифический человеко-месяц»: если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше. Поэтому реакция многих менеджеров, особенно в безнадежных проектах, будет иной: «В нашем распоряжении неограниченное и бесплатное сверхурочное время. Если проект отстает от графика, попросим команду «добровольно» поработать сверхурочно некоторое время, пока проект опять не войдет в график. Если с самого начала проекта ясно, что график чересчур оптимистичен, то следует заранее объявить, что сверхурочная работа будет обязательной, и довести до сведения каждого, что все продвижения по службе и поощрения будут зависеть от их энтузиазма в выполнении своих обязанностей».

Исследования показали, что менеджеры зачастую справляются с проблемами, используя мысленные модели, не учитывающие всех аспектов ситуации. Например, менеджеры с техническим образованием склонны недооценивать влияние социальных факторов.

Чем же еще пользуются менеджеры проектов помимо мысленных моделей, основанных на опыте, интуиции и суевериях? Разумеется, многие менеджеры используют сетевые графики PERT (Program Evaluation-and- Review Technique — метод оценки и пересмотра планов) или графики Ганта. Однако не будет преувеличением предположить, что Microsoft Excel — наиболее широко используемое средство управления проектами в IT-организациях. Табличная модель дает полезную информацию относительно процесса, проекта или организации, и большинство менеджеров с этим согласятся.

Тем не менее таблица статична, она не показывает динамику процессов, во всяком случае, она не наглядна. С таблицами связана еще одна проблема. Поскольку они зачастую связаны с финансовым планированием, их стремятся

наполнить только «осязаемыми» показателями, которые можно измерить — количество сотрудников, денег, рабочих станций, строк кода и т.д. Такой подход уводит в сторону от оценки неформальных параметров проекта — морального состояния, мотивации, качества и уровня понимания технологии разработки ПО.

Поскольку такие параметры являются неформальными по своей природе, нужно относиться к ним с некоторой предосторожностью и использовать их в основном для качественной оценки порядка изменений и прогнозирования тенденций. Например, менеджер может задать такой практический вопрос: «Что произойдет с бюджетом и графиком проекта, если я направлю участников проектной команды на учебные курсы, и в результате они потратят вдвое меньше времени на освоение объектно-ориентированной технологии?»

Большое значение имеет также природа взаимодействий между различными компонентами процесса. Если изменить какой-либо компонент процесса, то это, вероятно, скажется на каком-нибудь другом. Кроме того, при взаимодействии могут иметь место временные задержки и циклы обратной связи. Эти взаимодействия можно отразить в табличной модели, но они не будут видны постороннему наблюдателю. Взаимодействия выражаются в виде связей между ячейками таблицы за счет использования различных математических формул, встроенных в ячейки. Если пользоваться табличным процессором, то можно видеть эти формулы, но если просто смотреть на таблицу, то не видно никаких связей между ее ячейками.

Сочетание нескольких взаимодействующих компонентов может привести к появлению «волнового эффекта» в организации. Допустим, организация-разработчик делает существенные инвестиции в CASE-средства, но с каждым кварталом тратит все меньше и меньше денег на обучение. Если работа с CASE-средствами требует использования формальных и строгих методов, а персонал не освоил их в достаточной степени, не удивительно, что продуктивность разработки в начальном периоде снижается. Вместе с падением продуктивности и разочарованием в новых CASE-средствах ухудшается и моральное

состояние команды, от которого невозможно отмахнуться. Если оно ухудшается, растет текучесть кадров, и в первую очередь уходят самые продуктивные. Во-первых, у них самые лучшие приглашения на работу от других организаций и, во-вторых, они испытывают наибольшее разочарование в нынешней ситуации. В результате их ухода средняя продуктивность организации падает еще больше, что вызывает дальнейшее ухудшение морального состояния ... которое повышает текучесть кадров ... которая заставляет увольняться оставшихся продуктивных сотрудников ... и так до тех пор, пока в организации не останется ни одного нормального специалиста.

Если этот пример выглядит слишком надуманным (хотя эта ситуация вполне реальна), можно рассмотреть более приземленную ситуацию. При оценке различных параметров проекта почти все менеджеры учитывают фактор «безопасности» (известный также, как фактор случайности, подгона под нужный результат, и под другими красивыми названиями), и они это делают без учета какой-либо динамики. Различные факторы безопасности порождают различные проекты. Если проектная команда знает, что ее менеджер опубликовал официальный график проекта с нулевым запасом прочности, участники проекта немедленно скорректируют свое поведение, исключив любую деятельность, которую они считают несущественной. В зависимости от характера проекта это может означать отказ от документирования, тестирования, обеспечения качества, присутствия на еженедельных совещаниях или ответов на сообщения электронной почты. И наоборот, если график проекта составлен с большим запасом, то начинает действовать Закон Паркинсона: работа занимает все отведенное для нее время.

Некоторые организации, достигшие третьего, четвертого или пятого уровня СММ, осознали важность проблем, связанных с динамикой процессов. Даже если исключить неформальные факторы, временные задержки и циклы обратной связи все равно могут оказать значительное воздействие на успех проекта. В качестве простого примера рассмотрим влияние дефектов проектирования, допущенных на стадии анализа в классическом «каскадном» процессе и

обнаруженных только в начале стадии программирования или тестирования. В формальном, строгом процессе разработки это означает, что нужно снова вернуться на стадию анализа для исправления ошибок и затем провести все изменения через проектирование и кодирование, пока не дойти снова до тестирования. Разумеется, возможно, что в процессе исправления ошибок что-то сделано не так, и тестирование выявит новые ошибки. В итоге может потребоваться несколько циклов исправлений, пока тестирование не даст удовлетворительных результатов. Все это может существенно повлиять на общее время процесса, который организация пытается усовершенствовать.

Если динамика процесса разработки ПО настолько важна, то каким образом в ней можно разобраться? Таблицы для этого не очень подходят, поскольку они не отражают взаимозависимости и обратные связи между компонентами процесса. Напротив, визуальные модели лучше подходят для иллюстрации «целостной» природы процесса, такие модели вызывают дискуссии и критику.

Здесь имеется только одна проблема: графические модели статичны. За небольшим исключением они были изобретены до появления современных CASE-средств, поэтому их привыкли рисовать на бумаге или отображать на пассивном дисплее. Если построить диаграмму потоков данных с помощью типичного CASE-средства, то она не «движется», не показывает динамику моделируемого процесса. Для этого требуется анимация, которой большинство поставщиков CASE-средств никогда не занималось.

В результате организации-разработчики, пытающиеся достичь более глубокого понимания своих процессов, обращаются к средствам имитационного моделирования, которые могут отобразить динамику системы. Одна из наиболее известных динамических моделей процесса создания ПО была разработана Тариком Абдель-Хамидом. Она отражает виды работ в процессе управления проектом среднего размера, использующего традиционные средства разработки и классическую «каскадную» модель процесса разработки. Хотя она не учитывает особенности сегодняшних проектов, использующих быстрое прототипирование, средства визуальной разработки, библиотеки повторно используемых

компонентов и т.д., тем не менее, она может служить в качестве отправной точки для организаций, желающих более глубоко разобраться во взаимодействиях между различными компонентами их процесса разработки ПО.

В идеальной ситуации менеджеру «безнадежного» проекта желательно располагать временем, финансами, настойчивостью и техническими ресурсами для создания динамической модели своего проекта. Основная причина для построения такой модели заключается в том, что ее можно использовать для ответов на вопросы типа «что-если», анализируя «нелинейные» аспекты динамики проекта. Например, может оказаться, что существенное увеличение некоторого параметра (такого, как количество новых сотрудников, нанимаемых за месяц) окажет совершенно незначительное воздействие на какой-либо из основных параметров, например, на «ожидаемую дату завершения проекта». В других случаях небольшое изменение незначительного, на первый взгляд, параметра (например, «частоты ухода» профессионалов) может оказать на проект такое существенное воздействие, которого никто не ожидал. Такое происходит из-за наличия множества взаимодействующих факторов, взаимное влияние которых друг на друга достаточно сложно количественно оценить.

К сожалению, в реальном мире очень немногие менеджеры «безнадежных» проектов располагают временем или ресурсами для создания моделей. Однако даже у менеджеров, находящихся в самых стрессовых ситуациях, все же есть некоторое время и ресурсы, и если они потратят хотя бы час или два на анализ временных задержек и обратных связей, то уже и это сможет существенно повлиять на динамику проекта.

7.8.

КОНТРОЛЬ НАД ПРОДВИЖЕНИЕМ ПРОЕКТА

Безусловно, очень важно в самом начале проекта договориться с требованиями заказчиками относительно реалистичных сроков и бюджета, сформировав

ровать проектную команду, определить необходимые рабочие процессы и процедуры. К сожалению, некоторые менеджеры проектов полагают, что это самая трудная часть работы и после того, как она сделана, все пойдет гораздо легче. На самом деле контроль за продвижением (иначе говоря — прогрессом) безнадежного проекта — это срочная и важная работа, отнимающая много времени; ее нельзя игнорировать или заниматься ею от случая к случаю. Именно это самая важная обязанность менеджера проекта после завершения всех тяжелых переговоров в начале проекта.

Чтобы преуспеть в этой деятельности, менеджер проекта должен уметь отличать реальный прогресс от «кажущегося» прогресса. Многие менеджеры проектов давно усвоили, что синдром «выполнено на 90%» является опасной иллюзией, и они знают, что в безнадежном проекте не менее опасно утверждать «мы выполнили анализ и проектирование на 100%, но еще не закончили написание кода и не тестировали его». Практичной и эффективной альтернативой это дилемме является принцип «ежедневной сборки» («daily build»).

Неявно подразумевается, что очередные результаты, получаемые проектной командой, появляются через интервалы, измеряемые месяцами или неделями. К этому приучает опыт нормальных проектов, и это согласуется с обычным темпом деловой жизни — например еженедельными совещаниями персонала, ежемесячными отчетами о состоянии работ, ежеквартальными презентациями для высшего руководства и т.д.

Однако безнадежные проекты нуждаются в другом подходе. Когда проект приходит к прототипированию и пошаговой разработке, работу над ним надо организовать на основе принципа «ежедневной сборки». Под этим подразумевается следующее: компиляция, сборка, установка и тестирование всей совокупности разработанного командой кода должны выполняться каждый день, как если бы этот день был последним перед завершением проекта, и на следующее утро было бы необходимо сдать законченную систему пользователям.

Разумеется, с первого дня невозможно приступить к ежедневной сборке. Правда, уже на второй день работы можно написать подпрограмму типа «Hello,

World», и трудно сегодня удивить кого-то совершенно новыми технологиями. Однако существуют определенные требования, которым должна удовлетворять версия прототипа системы при первой «официальной» демонстрации: помимо того, что она включает необходимую совокупность компонентов, процедур или модулей и несколько сотен, а может быть и тысяч строк кода, она должна выполнять реальный ввод данных, производить реальную обработку или вычисления и формировать реальный выход. Именно с этого момента следует начинать ежедневную сборку и формировать каждый день новую (желательно улучшенную) версию системы.

Для менеджера проекта подобная стратегия может быть приоритетом номер один. Если в течение недели никто не сможет сообщить менеджеру проекта, что разрабатываемое приложение не хочет правильно взаимодействовать с новой объектно-ориентированной базой данных, то в результате проект может безнадежно отстать от графика. До тех пор, пока менеджер судит о состоянии проекта по устным отчетам, запискам или диаграммам потоков данных, будет слишком легко перепутать движение с прогрессом и усилия с результатами. Однако, если менеджер проекта настаивает на ежедневной физической демонстрации результатов, будет нелегко скрыть возникающие трудности, которые в конечном счете могут способствовать провалу проекта.

В то время как вряд ли кто-нибудь вправе претендовать на «изобретение» данного подхода, известно, что он впервые стал популярным во время разработки операционной системы Windows NT. Можно также отметить, что при разработке Windows 95 использовался принцип «ежедневной сборки»; заключительная бета-версия перед выпуском конечного продукта была реализована в августе 1995 г. и называлась «Проект 951».

Важно осознавать, что подобный подход становится неотъемлемой составляющей процесса разработки системы, которому следует проектная команда. Кроме того, процесс ежедневного завершения работы над проектом должен быть автоматизированным и выполняться ночью без участия программистов, тогда он будет эффективным. Такой подход подразумевает наличие авто-

матризованного управления конфигурацией ПО и механизмов управления исходными кодами, а также разнообразных «сценариев» для выполнения компиляции и сборки приложений. Но что еще более важно, он подразумевает наличие системы автоматизированного тестирования, которая может работать всю ночь, выполняя множество тестов для проверки работоспособности системы. Таким образом, чтобы реализовать на практике принцип «ежедневной сборки», необходимо иметь в своем распоряжении адекватный набор средств и технологий.

Хотя ежедневная сборка весьма эффективна с точки зрения мониторинга продвижения безнадежного проекта, она не слишком помогает справиться со многими серьезными проблемами. Важно понимать семантическое различие между тем, что некоторые менеджеры называют «issue» (вопрос, незначительная проблема), и рисками. В ходе работы над проектом ситуация может выйти из-под контроля. Это происходит потому, что управление рисками строится в основном на эмоциях и инстинктах, а не на формальных процессах, и менеджер в процессе работы зачастую может вовремя не заметить появление новых рисков. В лучшем случае риски, очевидные в самом начале проекта, будут устраняться. В нормальной ситуации они являются поводом для беспокойства в течение всей работы над проектом (например, риск ухода ключевого разработчика). Однако неожиданно могут возникнуть совершенно новые риски, которые окажутся убийственными для проекта.

Проектная команда должна различать оценку риска, управление риском и ликвидацию риска. В худшем случае проектная команда реагирует на риск только по мере его возникновения — например, выделяет дополнительные ресурсы для проведения очередного тестирования, чтобы смягчить последствия ошибки. В этом случае проблеме уделяется внимание только после ее проявления. При подобном подходе работа строится в стиле «тушения пожара». Для проектной команды такая ситуация может быть катастрофой. Гораздо лучше предупреждать риск заранее. Это означает, что команда согласна соблюдать выполнение формальных процессов оценки и управления с целью предотвра-

щения потенциальных рисков.

Профилактическое управление рисками направлено на устранение самих причин, приводящих к неудачам. Оно нередко является центральным звеном всех начинаний, связанных с управлением качеством в организации. При таком подходе проявляется тенденция к значительному расширению границ оценки рисков и появлению возможности их предотвращения. Это может привести к весьма агрессивному стилю управления, основанному на полном контроле над степенью риска в соответствии с его допустимостью для организации. Эта проблема в большей степени стратегическая, она должна обсуждаться и реализовываться за пределами «безнадежного» проекта. Команда «безнадежного» проекта преследует в основном тактические цели; она пытается не изменить культуру организации, а всего лишь выжить и нормально закончить проект.

Оценка риска выполняется обычно путем оценок сложности разрабатываемой системы или продукта, а также клиентской среды и среды проектной команды. Сложность продукта можно оценить в терминах объема (например, количества функциональных точек), ограничений производительности, технической сложности и т.д. Риск, связанный с клиентской средой, определяется в основном такими факторами, как число пользователей, вовлеченных в проект, их уровень квалификации, значение разработки для пользовательского бизнеса, вероятность того, что внедрение новой системы (если оно произойдет) приведет к реорганизации, и т.д. Наконец, риск, связанный со средой проектной команды, зависит от ее способностей, опыта, морального состояния и физического/эмоционального здоровья.

Как правило, достаточно полная модель риска может включать сто или более факторов риска. Некоторые из рисков можно оценить количественно, например требования к производительности (скорости реакции системы) или объем системы, выраженный в количестве функциональных точек. Другие факторы (например, степень расположенности или враждебности пользователей) могут быть оценены только качественно. Такие факторы принято характеризовать значениями «высокий», «низкий» или «средний».

После того как риски подверглись идентификации и оценке, менеджер и команда могут выбрать подходящую стратегию минимизации или исключения по возможности большего числа рисков. Эта деятельность носит, конечно, общий характер. Однако не следует забывать: сама природа «безнадежного» проекта такова, что количество рисков превышает обычное, они более серьезны, и от них нельзя просто так избавиться. С другой стороны, если риски являются экстраординарными, то и решения должны быть адекватными: в то время как проектная команда нормального проекта может никогда не набраться смелости, чтобы обратиться к исполнительному директору или первому вице-президенту с просьбой уменьшить риск путем существенного увеличения бюджета или снятия серьезных бюрократических ограничений, участникам безнадежного проекта будет вполне разумно обратиться с такой просьбой.

В любом случае, если существуют серьезные факторы риска, воздействие которых исключить невозможно (а в «безнадежных» проектах почти всегда так оно и есть), их следует зафиксировать в специальном документе, идентифицировав для каждого риска возможные последствия и разработав план действий в непредвиденных ситуациях. Документирование рисков является важной практической деятельностью, подталкивающей пользователей и руководство к пониманию того, что они предпочитали не замечать и игнорировать.

Менеджеры проектов традиционно использовали сетевые графики и графики Ганта для планирования продвижения своих проектов. Они пытались разбить весь проект на наибольшие задачи с «двоичным» результатом (т.е. либо «выполнено», либо «не выполнено») и контролем качества на выходе, который позволял убедиться, что результаты выполненных задач можно включить в разрабатываемую систему. В такой подход очень хорошо вписывается принцип ежедневной сборки: периодические демонстрации либо показывают работоспособность промежуточного продукта, либо нет. Помимо этого менеджеру проекта нужно оценивать текущее состояние, продвижение, риски и проблемы проекта путем проведения периодических совещаний. На самом деле все, что нужно менеджеру проекта, — это точная и реалистичная оценка перспектив проекта.

Он должен сообщить своей команде, когда следует ожидать поставку системы заказчиком и какие проблемы ждут их впереди.

Это достаточно трудно сделать, даже если в совещании участвуют только сам менеджер проекта и участники проектной команды. Если же в совещании участвуют сотрудники отдела качества, аудиторы, высшее руководство, конечные пользователи и другие заинтересованные лица, то оно зачастую превращается в политическое мероприятие, на котором реальные проблемы и состояние проекта скрываются, чтобы избежать политических последствий.

В то же время менеджер проекта должен понимать, что периодические оценки действительно важны и могут оказаться очень эффективными с точки зрения разрешения проблем и планирования будущей деятельности. При этом нужно обратить особое внимание на неформальные экспертные оценки и доступность их результатов для всех активных участников проекта.

Нужно проводить небольшие разборы в конце по завершении каждой стадии проекта или каждого прототипа, или каждой очередной версии системы, поставляемой заказчику. В зависимости от конкретного проекта это означает, что разбору подвергается работа, выполненная в течение пары недель или месяцев, поэтому большинство основных игроков еще участвует в проекте и, скорее всего, помнит, что они делали и почему. Такой разбор можно провести на одном совещании за несколько часов, вместо привычного «посмертного» разбора в конце проекта, который продолжается несколько дней или недель.

7.9.

ТЕХНОЛОГИЯ И ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА «БЕЗНАДЕЖНЫХ» ПРОЕКТОВ

Летом 1992 г. Эдварду Йордону довелось обедать с группой менеджеров среднего уровня Microsoft. Во время беседы он спросил, является ли для них обычным делом использование таких методологий, как структурный анализ или объектно-ориентированное проектирование. Ответы были примерно следующими: «иногда», «вроде бы да», «от случая к случаю» и «а что это такое?». Когда же он поинтересовался относительно использования CASE-средств (которые в то время все еще были довольно популярными в индустрии ПО), то из их ответов понял, в чем заключается общее мнение майкрософтовцев: эти средства годятся для «людей с улицы», т.е. «невежественных дикарей, которые только что вылезли из своего первобытного леса и начали обучаться программированию, в отличие от настоящих программистов, не нуждающихся во всяких финтифлюшках».

Будучи слегка уязвленным, он любопытствовал, используют ли их проектные команды хоть какие-нибудь средства, и в ответ услышал, что каждая команда Microsoft может выбрать любые средства, которые сочтет подходящими для своего проекта. Ухватившись за такой ответ, Йордон спросил, какое средство считает наиболее важным типичная проектная команда?

«На днях я задал одной из проектных команд такой же вопрос», - ответил один из менеджеров. «Как вы думаете, что они ответили?»

«Какой-нибудь высокопроизводительный компилятор C++? Ассемблер? Или мощное средство отладки для устранения множества ошибок в их коде (хи-хи-хи)?»

«Ничего подобного», — ответил менеджер, игнорируя иронию. «Они ответили: электронная почта. Средний разработчик Microsoft получает сотню сообщений в день; он живет в электронной почте. Уберите электронную почту, и проект умрет».

Эти события происходили до начала эры Internet и World Wide Web, когда сотня почтовых сообщений в день могла потрясти воображение. Однако можно представить себе, что если бы такой же вопрос о «наиболее важном средстве» был задан в 1996 г., ответом могло быть «World Wide Web»; по аналогии,

«факс» в 1987 г., «ПК» в 1983 г., «онлайновый терминал» в 1976 г. и «телефон на рабочем столе» в 1964 г.

Очевидно, не следует ожидать, что команда «безнадежного» проекта сможет ограничиться только одним средством. Большинство команд (даже в нормальных проектах) пользуется в своей повседневной работе самыми разнообразными средствами и технологиями. Правда, подчас количество средств становится чересчур большим, технологии — слишком новыми, а иногда нежелательные средства навязываются некомпетентными менеджерами.

Ранее настоятельно рекомендовалось устанавливать приоритеты для пользовательских требований. Такой же подход используется по отношению к средствам и технологии, и его разумно применить в самом начале проекта. Наиболее очевидная причина — экономия средств. Даже если средства хорошо работают и все знакомы с ними, их приобретение может стоить слишком дорого. Кроме того, на их получение может уйти много времени — процесс приобретения их в условиях обычной корпоративной бюрократии может завершиться уже после окончания работы над проектом. Для большинства «безнадежных» проектов следует выбрать небольшое количество критически важных средств и убедить высшее руководство (или соответствующую службу) в необходимости их приобретения.

С другой стороны, предположим, что команда работает в крупной корпорации, имеющей в своем распоряжении сотни различных средств, приобретаемых в течение ряда лет. Следует ли их все использовать? Конечно, нет! Даже если все они работают, те умственные усилия, которые необходимо затратить, чтобы запомнить, как ими пользоваться, а также дополнительные усилия для обеспечения их совместной работы обычно сводят на нет всю выгоду. Можно провести аналогию с командой альпинистов, которые собираются штурмовать вершину и пытаются решить, какое снаряжение им использовать. Существуют необходимые вещи (палатки, питьевая вода и т.д.); и, если маршрут не слишком сложный, можно взять с собой некоторые новомодные приспособления, о которых написано в альпинистском журнале. Однако, если они собираются штурмовать Эверест, им не обойтись без помощи ослов или носильщиков из местных жителей, иначе они будут не в состоянии тащить на спине по

300 фунтов снаряжения на человека.

Команда «безнадежного» проекта должна самостоятельно, независимо от принятых в организации стандартов, решить, какие средства являются необходимыми, а без каких можно обойтись. Очень важно участникам команды прийти к единому мнению относительно используемых в проекте средств, иначе наступит хаос. Разумеется, это утверждение не следует понимать буквально; оно не означает, что все участники команды должны обязательно использовать один и тот же текстовый процессор для подготовки своих документов, хотя, скорее всего, важен один и тот же компилятор C++. Одна из проблем, связанных с «безнадежными» проектами, заключается в том, что разработчики ПО считают допустимой полную анархию на индивидуальном уровне (например, если им хочется использовать никому не известный компилятор C++, который они переписали с университетского Web-сайта, то они считают это своим неотъемлемым правом). Но это не так: неотъемлемым правом обладает команда, и менеджер проекта должен неуклонно проводить его в жизнь во всех ситуациях, когда несовместимые средства могут привести к значительным разногласиям.

Следовательно, пока участники команды не поработают вместе над несколькими «безнадежными» проектами, они не придут к единому мнению относительно «минимального» набора средств. Определив набор средств, команда выбирает те, которыми следует пользоваться. При этом возникает еще одна проблема — добиться согласия в команде и получить разрешение руководства на приобретение новых средств.

Менеджер проекта должен настаивать на достижении консенсуса; в самом деле, это может быть одним из критериев, используемых менеджером для выбора потенциальных участников команды. То же самое можно сказать относительно процессов, которые обсуждались ранее.

Вот перечень средств, которые рекомендуются для «безнадежных» проектов:

Электронная почта, ПО для групповой работы, средства Internet/Web, видеоконференции и т.д. Так же, как и в эпизоде с Microsoft, эти средства

находятся в начале списка. Причина заключается в следующем: электронные средства общения и взаимодействия являются не только более эффективным средством коммуникации, чем записки и факсы, но они также способствуют координации и сотрудничеству. Безразлично, какие именно средства использовать: Microsoft Outlook или Lotus Notes; важно только, чтобы вся команда работала в сети и хранила там общие проектные данные.

Средства прототипирования/быстрой разработки приложений (RAD). Почти все «безнадежные» проекты используют в той или иной степени прототипирование и пошаговую разработку; следовательно, им необходимы соответствующие инструментальные средства. Сегодня не так просто отыскать популярную среду разработки приложений, которая заявляла бы о себе иначе, чем среда RAD. Большинство таких средств обладают визуальным пользовательским интерфейсом, выполненным в стиле «drag and drop», облегчающим и ускоряющим процесс разработки. Важно, чтобы вся команда использовала один и тот же набор средств от одного и того же поставщика. Если часть команды использует среду разработки Java компании Sun, а другие — Microsoft Visual J++, то это явно глупо, хотя и допустимо с точки зрения технологии.

Средства управления конфигурацией/управления версиями. Некоторые полагают, что они должны быть на первом месте в списке. Очевидно, использование средств управления конфигурацией может принести больше пользы, если они будут интегрированы со средствами разработки приложений. Например, SourceSafe от Microsoft может и не быть самым лучшим средством управления версиями ПО, однако тот факт, что оно тесно интегрировано с Visual Basic, является весомым аргументом в его пользу. Аналогично, многие другие средства разработки приложений интегрированы с PVCS или другими подобными средствами управления конфигурацией.

Средства тестирования и отладки. Многие автоматически включают эти средства в «базовый» набор средств разработки приложений, позволяющих создавать, компилировать и выполнять код.

Средства управления проектом (оценка, планирование, PERT/GANTT

и т.д.). Обычно их считают средствами менеджера проекта. К этой же категории следует отнести такие средства оценки, как ESTIMACS (Computer Associates), CHECKPOINT (Software Productivity Research) и SLIM (Quantitative Software Management). Они, я считаю, являются важными, поскольку позволяют в ходе выполнения проекта динамически пересматривать планы и сроки.

Наборы повторно используемых компонентов. Если проектная команда знакома с концепцией повторного использования ПО и рассматривает ее как стратегическое оружие, позволяющее достичь высокого уровня продуктивности разработки, то набор повторно используемых компонентов должен быть в списке тех средств, которые необходимо использовать. Это может быть набор компонентов VBX для Visual Basic, компоненты Java компании Sun или библиотека классов STL для C++. Разумеется, можно использовать и компоненты, разработанные другими проектными командами в организации. Выбор их обычно зависит от используемого языка программирования, и это еще одна проблема, нуждающаяся в выработке единого подхода со стороны проектной команды.

CASE-средства для анализа и проектирования. Некоторые проектные команды рассматривают CASE-средства как «костыли» для новичков, а другие считают их не менее важными, чем текстовые процессоры. Самая большая проблема, связанная с CASE-средствами, заключается в том, что они поддерживают (а иногда навязывают) определенную методологию, которую проектная команда не понимает и не желает использовать.

Упомянутая проблема CASE-средств, вероятно, представляет собой наиболее очевидный пример трюизма: средства и процессы связаны друг с другом сложным образом. Бессмысленно браться за объектно-ориентированное CASE-средство, поддерживающее анализ, если разработчики никогда не слышали об иерархии классов или диаграммах вариантов использования. Использование такого CASE-средства будет не только бесполезным, но и чрезвычайно обременительным, если проектная команда искренне полагает, что диаграммы классов представляют собой лишние формы бюрократических документов.

Но ситуация не всегда бывает черно-белой. Например, проектная команда считает, что диаграммы потоков данных полезны, но только как «неформальное» средство моделирования. Таким образом, «гибкое» CASE-средство может рассматриваться как нужное и полезное, в то время как «жесткое» CASE-средство может быть отвергнуто.

Все это означает, что команда «безнадежного» проекта должна в первую очередь нормально воспринимать те процессы и методы, которым она собирается следовать. Кроме того, она должна решить, каким из них надо подчиняться беспрекословно, а каким — следовать духу, но не букве закона. После принятия такого решения можно соответственно выбрать (или отвергнуть) средства и технологию. Таким же образом менеджер проекта может решить использовать какое-либо средство для усиления процесса, необходимость которого все понимают, но на практике следуют ему достаточно небрежно; примеры таких процессов — контроль версий и управление конфигурацией.

Один из величайших мифов, касающихся использования инструментальных средств в любых проектах (и особенно опасных в безнадежных проектах) заключается в отношении к средству как к «серебряной пуле», которая позволит творить чудеса. Разумеется, поиском чудес занимается в основном высшее руководство. Однако даже менеджера проекта могут соблазнить рекламные заявления поставщика, уверяющего, что с помощью его гениальных средств можно в десять раз повысить производительность программирования, тестирования или какой-нибудь другой деятельности.

Помимо проблемы, заключающейся в новизне таких средств и в неумении их использовать, существует более важный момент: средство станет подобным «серебряной пуле» только в том случае, если оно позволит или заставит разработчиков изменить свои процессы. Едва ли кто-нибудь станет возражать против использования усовершенствованных технологий, позволяющих избавляться от рутинных и утомительных процессов. Труднее внедрить новую технологию, требующую введения новых процессов или модификации существующих процессов, к которым привыкли.

Ирония заключается в том, что большинство организаций в своих провалах винит технологию. Они приобретут дорогостоящую библиотеку классов или поменяют свою старую технологию разработки ПО на объектно-ориентированную, исходя из предположения, что объекты и повторное использование — это одно и то же. Обнаружив, что не добились сколько-нибудь ощутимых результатов, они будут винить во всем объектную технологию, библиотеку классов, поставщика и др. Между тем все процессы остались в точности такими же, какими были до внедрения новой технологии. Культура такой организации может быть выражена следующей фразой: «Только бездари пользуются чужим кодом; настоящие программисты, черт возьми, пишут свой!»

С точки зрения работы над «безнадежным» проектом мораль проста: если внедрение новых средств потребует серьезного изменения стандартных процессов команды, то это значительно увеличит риск и, возможно, будет способствовать провалу проекта. Необходимость обучения и освоения практического использования новых средств создает дополнительные проблемы. Однако наиболее серьезной из них является изменение режима работы, который целиком определяется процессом. Это трудно сделать и в нормальных условиях, когда у нас достаточно времени, чтобы относительно безболезненно перейти к новому процессу. Для «безнадежного» проекта такой переход будет просто катастрофическим.

При работе над «безнадежными» проектами некоторые часто хватаются за новые средства и технологии для достижения более высокой продуктивности. При этом возникают два наиболее вероятных риска — технология и обучение. Во многих случаях новое средство даже не является законченным коммерческим продуктом; обычно кто-нибудь из проектной команды переписывает из Интернета бета-версию. Или же данное средство невозможно интегрировать с любыми другими средствами, используемыми проектной командой; поставщик давал на этот счет неопределенные обещания, однако в результате оказалось, что возможности экспорта-импорта изобилуют ошибками. Или, средство никем не поддерживается — оно разработано студентом из Ирака или (что еще хуже)

создано в домашних условиях одним из разработчиков ПО, который не видит ничего странного в том, что банк разрабатывает свое собственное CASE-средство, а страховая компания — свою СУБД.

Допустим, что средство является достаточно надежным, а его поставщик пользуется устойчивой репутацией и поддержкой на высоком уровне. В этом случае проблемы будут связаны с освоением, поскольку даже если это средство прежде широко использовалось в организации, никто не воспринимал его как «серебряную пулю», которая сможет чудесным образом спасти проектную команду от гарантированной катастрофы. Иногда можно встретить проектную команду, добивающуюся разрешения использовать какое-либо мощное средство, с которым она уже имела дело в предыдущей работе. Однако это редкое явление. В большинстве случаев никто из участников проектной команды и вообще никто в организации никогда прежде не видел или не использовал подобное средство. Если предположить, что не существует никаких проблем, связанных с данным средством, тогда все, что остается — это обучение и практика.

Как много времени на это потребуется? Очевидно, это зависит от характера и сложности средства, а также от его пользовательского интерфейса, возможностей онлайн-подсказки и др. В лучшем случае разработчики могут самостоятельно разобраться в использовании средства. В такую возможность очень хочется верить менеджеру проекта и разным другим руководителям, поскольку они считают любое обучение потерей времени и отвлечением от работы над проектом. Более реалистичная оценка заключается в том, что на освоение средства потребуется час, день или неделя. Независимо от формы обучения (занятия в классе, чтение книги или просто «игры» со средством), на это все равно потребуется какое-то время.

Однако в результате недельного обучения не получится опытного пользователя, в совершенстве владеющего средством. Это должно быть очевидным, однако нарушает планы высшего руководства, которое склонно возмущаться: «Мы потратили кучу денег на этих высокооплачиваемых преподавателей и напрасно потеряли столько времени в классах, чтобы эти ленивые бездельники-

программисты могли научиться кодировать. Теперь мы хотим увидеть реальную отдачу от этого «замечательного» средства, за которое вы так агитировали!» Наверное, в такой наивности высшего руководства нет ничего удивительного, поскольку они сами практически не сталкивались с инструментальными средствами. Однако, к сожалению, приходится наблюдать похожую реакцию со стороны многих менеджеров «безнадежных» проектов, гораздо лучше разбирающихся в технических вопросах.

Если бы внедрение новой технологии не оказывало никакого влияния на процессы, не требуя специального обучения и практики, то можно было бы принять решение, основываясь всего лишь на сопоставлении затрат и выгод. Поскольку инстинкт многих руководителей высокого уровня подсказывает им, что любую проблему можно решить с помощью простого финансового вливания, то существует тенденция к гораздо большему использованию совершенно новых технологий в «безнадежных» проектах, чем в нормальных. Ирония заключается в том, что новое средство может оказаться последней каплей, переполнившей чашу терпения. Таким образом, именно на средство будет возложена ответственность за неудачу проекта.

Итак, нужно использовать любые средства, которые подходят для «безнадежного» проекта, не обращая внимания на то, какими их считает весь остальной мир: современными или устаревшими. Но не следует забывать, что новые средства в «безнадежном» проекте окажут воздействие и на людей, и на процессы. Как сказал 150 лет назад Генри Дэвид Торо, люди становятся орудиями в руках собственных средств.

! Следует запомнить

1. Во многих проектах наиболее серьезные проблемы носят не столько технический характер, сколько политический, социальный, культурный и человеческий.
2. Проекты, выполняющиеся в экстремальных условиях, предъявляют особые требования к используемым методам, средствам и технологиям.

Основные понятия

«Безнадежный» проект, «достаточно хорошее» ПО, принцип «ежедневной сборки».

? Вопросы для самоконтроля

1. Каким образом менеджер проекта может оценить вероятность его успешного завершения?
2. Чем вызываются разногласия между участниками «безнадежных» проектов?
3. Какие технологии и инструментальные средства являются наиболее предпочтительными в «безнадежных» проектах?
4. Какие меры следует предпринимать менеджеру проекта, чтобы снизить текучесть кадров?

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. *Бек К.* Экстремальное программирование: Пер. с англ. — СПб.: Питер, 2002.

2. *Боггс У., Боггс М.* UML и Rational Rose: Пер. с англ. — М.: ЛОРИ, 2000.
3. *Бозм Б.У.* Инженерное проектирование программного обеспечения : Пер. с англ. — М.: Радио и связь, 1985.
4. *Брауде Э. Дж.* Технология разработки программного обеспечения: Пер. с англ. - СПб: Питер, 2004.
5. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы: Пер. с англ. — СПб.: Символ-Плюс, 1999.
6. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. - 2-е изд.: Пер. с англ. — М.: Издательство Бином, СПб.: Невский диалект, 1999.
7. *Буч Г. и др.* Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Джекобсон: Пер. с англ. - М.: ДМК, 2000.
8. *Вендров А.М.* CASE-технологии. Современные методы и средства проектирования информационных систем. - М.: Финансы и статистика, 1998.
9. *Вендров А.М.* Практикум по проектированию программного обеспечения экономических информационных систем: Учеб. пособие. - М.: Финансы и статистика, 2002.
10. *Вигерс К.* Разработка требований к программному обеспечению: Пер. с англ. - М.: Русская редакция, 2004.
11. *Гамма Э. и др.* Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес: Пер. с англ. — СПб: Питер, 2001.
12. *Тома Х.* UML. Проектирование систем реального времени, распределенных и параллельных приложений: Пер. с англ. — М.: ДМК, 2002.
13. *Йордан Эд.* Путь камикадзе: Пер. с англ. — М.: ЛОРИ, 2001.
14. *Калашиян А.Н., Каляное Г.Н.* Структурные модели бизнеса: DFD-технологии. — М.: Финансы и статистика, 2003.
15. *Каляное Г.Н.* Консалтинг при автоматизации предприятий. — М.: СИНТЕГ, 1997. (Серия «Информатизация России на пороге XXI века»)
16. *Коберн А.* Быстрая разработка программного обеспечения: Пер. с англ. -

- М.: ЛОРИ, 2002.
17. *Кватрани Т.* Визуальное моделирование с помощью Rational Rose 2002 и UML: Пер. с англ. - М.: Вильяме, 2003.
 18. *Коберн А.* Современные методы описания функциональных требований к системам: Пер. с англ. — М.: ЛОРИ, 2002.
 19. *Конноли Т., Бегг К.* Базы данных: проектирование, реализация и сопровождение. Теория и практика. — 3-е изд.: Пер. с англ. — М.: Вильяме, 2003.
 20. *Крачтен Ф.* Введение в Rational Unified Process: Пер. с англ. -М.: Вильяме, 2002.
 21. *Ларман К.* Применение UML и шаблонов проектирования. — 2-е изд.: Пер. с англ. — М.: Вильяме, 2002.
 22. *Леффингуэлл Д., Уидриг Д.* Принципы работы с требованиями к программному обеспечению. Унифицированный подход: Пер. с англ. — М.: Вильяме, 2002.
 23. *Липаев В.В.* Документирование и управление конфигурацией программных средств. Методы и стандарты. — М.: СИНТЕГ, 1998.
 24. *Липаев В.В.* Системное проектирование сложных программных средств для информационных систем. — 2-е изд. — М.: СИНТЕГ, 2002.
 25. *Маклаков СВ.* VPwin и ERwin. CASE-средства разработки информационных систем. — М.: Диалог-МИФИ, 1999.
 26. *Маклаков СВ.* Моделирование бизнес процессов с VPwin 4.0. - М.: Диалог-МИФИ, 2002.
 27. *Марка Д.А., МакГоуэн К.* Методология структурного анализа и проектирования. - М.: МетаТехнология, 1993.
 28. *Мацяшек Л.* Анализ требований и проектирование систем. Разработка информационных систем с использованием UML: Пер. с англ. — М.: Вильяме, 2002.
 29. *Мюллер Р.* Базы данных и UML. Проектирование: Пер. с англ. - М.: ЛОРИ, 2002.

30. *Нейбург Э. Дж., Максимчук Р.А.* Проектирование баз данных с помощью UML: Пер. с англ. — М.: Вильяме, 2002.
31. *Одинцов И.* Профессиональное программирование. Системный подход. — СПб.: БХВ-Петербург, 2002.
32. *Орлов С.А.* Технологии разработки программного обеспечения. - СПб.: Питер, 2002.
33. Оценка и аттестация зрелости процессов создания и сопровождения программных средств и информационных систем (ISO/IEC TR 15504-CMM): Пер. с англ. А.С. Агапова и др. - М.: Книга и бизнес, 2001.
34. *Палмер С.Р., Фелсинг Дж.М.* Практическое руководство по функционально-ориентированной разработке ПО: Пер. с англ. — М.: Вильяме, 2002.
35. Принципы проектирования и разработки программного обеспечения. Учебный курс MCSD. - 2-е изд.: Пер. с англ. - М.: Русская редакция, 2002.
36. *Рамбо Дж. и др.* UML. Специальный справочник / Дж. Рамбо, Г. Буч, А. Якобсон: Пер. с англ. — СПб: Питер, 2002.
37. *Розенберг Д., Скотт К.* Применение объектно-ориентированного моделирования с использованием UML и анализ прецедентов: Пер. с англ. - М.: ДМК, 2002.
38. *Ройс У.* Управление проектами по созданию программного обеспечения: Пер. с англ. — М.: ЛОРИ, 2002.
39. *Соммервилл И.* Инженерия программного обеспечения. — 6-е изд.: Пер. с англ. - М.: Вильяме, 2002.
40. *Фатрелл Р. и др.* Управление программными проектами: достижение оптимального качества при минимуме затрат / Р. Фатрелл, Д. Шафер, Л. Шафер: Пер. с англ. - М.: Вильяме, 2003.
41. *Фаулер М., Скотт К.* UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. - М.: Мир, 1999.
42. *Черемных С.В. и др.* Структурный анализ систем: IDEF-технологии / С.В.Черемных, И.О. Семенов, В.С. Ручкин. - М.: Финансы и статистика,

2001.

43. *Черемных СВ. и др.* Моделирование и анализ систем. IDEF-технологии: практикум/С.В.Черемных, И.О. Семенов, В.С. Ручкин. - М.: Финансы и статистика, 2002.
44. *Элиенс А.* Принципы объектно-ориентированной разработки программ. - 2-е изд.: Пер. с англ. - М.: Вильяме, 2002.
45. *Якобсон А. и др.* Унифицированный процесс разработки программного обеспечения / А. Якобсон, Г. Буч, Дж. Рамбо: Пер. с англ. - СПб.: Питер, 2002.

КРАТКИЙ СЛОВАРЬ ТЕРМИНОВ

А

Абстрагирование — выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа.

Агрегация (форма ассоциации) - связь между целым (составным) объектом и его частями (компонентными объектами).

Ассоциация - семантическая связь между классами. Ассоциация отражает структурные связи между объектами различных классов.

Атрибут — любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.

Архитектура ПО — описание системы ПО, включающее совокупность структурных элементов системы и связей между ними; поведение элементов системы в процессе их взаимодействия и иерархию подсистем, объединяющих структурные элементы.

Бизнес-модель — формализованное описание процессов, связанных с ре-

сурсами и отражающих существующую или предполагаемую деятельность предприятия.

Бизнес-процесс — логически завершенный набор взаимосвязанных и взаимодействующих видов деятельности, поддерживающий деятельность организации и реализующий ее политику, направленную на достижение поставленных целей.

Вариант использования (use case) - последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом).

Внешняя сущность — материальный предмет или физическое лицо, представляющие собой источник или приемник информации.

Действующее лицо (actor) — роль, которую пользователь играет по отношению к системе.

Ж-3

Жизненный цикл программного обеспечения - период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

Зрелость процессов (software process maturity) - степень управляемости, контролируемости и эффективности процессов создания ПО.

И

Иерархия - ранжированная или упорядоченная система абстракций, расположение их по уровням.

Индивидуальность - набор свойств объекта, отличающих его от всех других объектов.

Инкапсуляция — физическая локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающая их реализацию за общедоступным интерфейсом.

Инструментальное средство (CASE-средство) - программное средство, поддерживающее процессы жизненного цикла ПО, определенные в стандарте ISO/IEC 12207:1995.

Интерфейс — совокупность операций, определяющих набор услуг класса, подсистемы или компонента.

Информационная система — совокупность функциональных и информационных процессов конкретной предметной области; средств и методов сбора, хранения, анализа, обработки и передачи информации, зависящих от специфики области применения; методов управления процессами решения функциональных задач, а также информационными, материальными и денежными потоками в предметной области.

К

Качество ПО — совокупность свойств, которые характеризуют способность ПО удовлетворять заданным требованиям.

Класс — множество объектов, связанных общностью свойств, поведения, связей и семантики. Класс инкапсулирует (объединяет) в себе данные (атрибуты) и поведение (операции).

Класс принадлежности — характеристика обязательности участия экземпляра сущности в связи.

Компонент - относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры. Компонент представляет собой физическую реализацию проектной абстракции.

Конфигурация ПО — совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО.

М

Моделирование - процесс создания формализованного описания системы в виде совокупности моделей.

Модель ПО - формализованное описание системы ПО на определенном уровне абстракции.

Модель ЖЦ ПО — структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ.

Модульность — свойство системы, связанное с возможностью ее деком-

позиции на ряд внутренне связанных, но слабо связанных между собой подсистем (модулей).

Мощность связи — максимальное число экземпляров сущности, которое может быть связано с одним экземпляром данной сущности.

Н

Накопитель данных — абстрактное устройство для хранения информации.

Наследование — построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций).

Нотация (языка моделирования) — совокупность графических объектов, которые используются в моделях.

Нормативно-методическое обеспечение (НМО) - комплекс документов, регламентирующих порядок разработки, внедрения и сопровождения ПО; общие требования к составу ПО и связям между его компонентами, а также к его качеству; виды, состав и содержание проектной и программной документации.

О

Образец — общее решение некоторой проблемной ситуации в заданном контексте. Образец состоит из четырех основных элементов: имя, проблема, решение и следствия.

Объект — осязаемая сущность (tangible entity) - предмет или явление, имеющие четко определяемое поведение.

Объектная декомпозиция - описание структуры системы в терминах объектов и связей между ними, а поведения системы — в терминах обмена сообщениями между объектами.

Операция (метод) — определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Операция - это реализация услуги, которую можно запросить у любого объекта данного класса.

П

Поведение - набор действий объекта и его реакций на запросы от других

объектов. Поведение характеризует воздействие объекта на другие объекты и, наоборот, с точки зрения изменения состояния этих объектов и передачи сообщений. Поведение представляется с помощью набора сообщений, воспринимаемых объектом (операций, которые может выполнять объект).

Полиморфизм — способность скрывать множество различных реализаций под единственным общим интерфейсом. Понятие полиморфизма может быть интерпретировано, как способность класса принадлежать более чем одному типу.

Поток данных - информация, передаваемая через некоторое соединение от источника к приемнику.

Программная инженерия 1. Совокупность инженерных методов и средств создания ПО. 2. Дисциплина, изучающая применение строгого систематического количественного (т.е. инженерного) подхода к разработке, эксплуатации и сопровождению ПО.

Программное обеспечение (программный продукт) — совокупность компьютерных программ, процедур и, возможно, связанной с ними документации и данных.

Проект — временное предприятие, осуществляемое с целью создания уникального продукта или услуги.

Проект ПО — совокупность спецификаций программного обеспечения (включающих модели и проектную документацию), обеспечивающих создание ПО в конкретной программно-технической среде.

Проектирование ПО — процесс создания спецификаций программного обеспечения на основе исходных требований к нему. Проектирование ПО сводится к последовательному уточнению его спецификаций на различных стадиях процесса создания ПО.

Прототип — действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО.

Процесс (ЖЦ ПО) — совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные.

Процесс создания ПО — совокупность упорядоченных во времени, взаимосвязанных и объединенных в стадии работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям.

Процесс (на диаграмме потоков данных) — преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Р

Рабочий продукт — информационная или материальная сущность, которая создается, модифицируется или используется в некоторой технологической операции (модель, документ, код, тест и т.п.). Рабочий продукт определяет область ответственности роли и является объектом управления конфигурацией.

Разработка ПО - комплекс работ по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, требуемых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т.д.

Репозиторий - база данных, предназначенная для хранения проектных данных (версий проекта и его отдельных компонентов), синхронизации поступления информации от различных разработчиков при групповой разработке, контроля данных на полноту и непротиворечивость.

Роль - определение поведения и обязанностей отдельного лица или группы лиц в среде организации-разработчика ПО, осуществляющих деятельность в рамках некоторого технологического процесса и ответственных за определенные рабочие продукты.

Руководство - практическое руководство по выполнению одной операции или совокупности технологических операций. Руководства включают методические материалы, инструкции, нормативы, стандарты и критерии оценки качества рабочих продуктов.

С

Связь - поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области (в модели «сущность-связь»).

Сообщение (message) — средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

Сопровождение ПО — внесение изменений в программное обеспечение в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Состояние объекта — одно из возможных условий, в которых он может существовать. Оно характеризуется перечнем всех возможных (статических) свойств данного объекта и текущими (динамическими) значениями каждого из этих свойств. Состояние объекта определяется значениями его свойств (атрибутов) и связями с другими объектами.

Стадия процесса создания ПО - часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для данной стадии требованиями.

Степень связи — количество сущностей, участвующих в связи.

Стереотип (UML) — новый тип элемента модели, который определяется на основе уже существующего элемента. Стереотипы расширяют нотацию модели, могут применяться к любым элементам модели и представляются в виде текстовой метки или пиктограммы.

Сущность - реальный либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Т

Технология создания ПО — упорядоченная совокупность взаимосвязанных технологических процессов в рамках жизненного цикла программного обеспечения.

Технологический процесс — совокупность взаимосвязанных технологических операций.

Технологическая операция — основная единица работы, выполняемая определенной ролью, которая подразумевает четко определенную ответствен-

ность роли; дает четко определенный результат (набор рабочих продуктов), базирующийся на определенных исходных данных (другом наборе рабочих продуктов); представляет собой единицу работы с жестко определенными границами, которые устанавливаются при планировании проекта.

Трассировка требований — установка и отслеживание связей требований с другими требованиями или проектными решениями

Требование — условие, которому должна удовлетворять система, или свойство, которым она должна обладать, чтобы удовлетворить потребность пользователя в решении некоторой задачи, а также удовлетворить требования контракта, стандарта или спецификации.

У

Уникальный идентификатор — избыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности.

Управление конфигурацией — применение административных и технических процедур для определения состояния компонентов ПО в системе, управления модификациями ПО, описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПО, управления хранением и поставкой ПО.

Управление требованиями 1. Систематический подход к выявлению, организации и документированию требований к системе. 2. Процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к системе и обеспечивающий его выполнение.

Ф-Я

Функциональная декомпозиция — описание структуры системы в терминах иерархии ее функций и передачи информации между отдельными функциональными элементами.

Функциональный тип — логическая группа взаимосвязанных данных, используемых и поддерживаемых приложением, а также элементарный про-

цесс, связанный с вводом и выводом информации.

Язык моделирования — совокупность элементов модели — фундаментальных концепций моделирования и их семантики; нотации (системы обозначений) — визуального представления элементов моделирования; руководства по использованию — правил применения элементов в рамках построения тех или иных типов моделей ПО.

СПИСОК ОСНОВНЫХ СОКРАЩЕНИЙ

АС — автоматизированная система

БД - база данных

ГНИ — Государственная налоговая инспекция

ЕСПД — Единая система программной документации

ЖЦ - жизненный цикл

ИС - информационная система

МДР - метод декомпозиции работ

НМО — нормативно-методическое обеспечение

ООП — объектно-ориентированный подход

ПК — персональный компьютер

ПО — программное обеспечение

СУБД — система управления базами данных

ТС ПО — технология создания программного обеспечения

AFP (Adjusted Function Points) - итоговая оценка количества функциональных точек

API (Application Programming Interface) — интерфейс прикладного программирования

ARIS (Architecture of Integrated Information System) - архитектура интегрированной информационной системы

ATM (Automated Teller Machine) - банкомат

CASE (Computer Aided Software Engineering) - автоматизированная разра-

ботка программного обеспечения

CBP (Critical Best Practices) - критически важные практические навыки

CDM (Custom Development Method) — метод разработки приложений пользователя

CMM (Capability Maturity Model) — модель оценки зрелости технологических процессов в организации ч

CMMI (Capability Maturity Model Integrated) - интеграционная версия модели CMM

COCOMO (Constructive COst Model) — конструктивная модель стоимости

DET (Data Element Type) — тип элементарных данных

DFD (Data Flow Diagram) — диаграмма потоков данных

eEPC (extended Event Driven Process Chain) - расширенная модель цепочки процессов, управляемых событиями

EF (Environmental Factor) - уровень квалификации разработчиков

EI (External Input) - входной элемент приложения

EIF (External Interface File) — внешний интерфейсный файл

EO (External Output) - выходной элемент приложения

EQ (External Query) — внешний запрос

ERM (Entity-Relationship Model) - модель «сущность-связь»

FP (Function Point) - функциональная точка

FTR (File Type Referenced) - файл типа ссылки

GSC (General System Characteristics) — общие характеристики системы

GUI (Graphical User Interface) — графический интерфейс пользователя

JAD (Joint Application Development) — совместная разработка приложений

JDBC (Java Database Connectivity) - интерфейс Java для реляционных баз данных

ICAM (Integrated Computer Aided Manufacturing) - интегрированная компьютеризация производства

IDEF (Icam DEFinition) — методология моделирования программы ICAM

IEC (International Electrotechnical Commission) - Международная комиссия

по электротехнике

IEEE (Institute of Electrical and Electronics Engineers) - Институт инженеров

по электротехнике и электронике

IFPUG (International Function Point User Group) - Международная организация по стандартизации методов оценки ПО

ILF (Internal Logical File) - внутренний логический файл

ISO (International Organization for Standardization) - Международная организация по стандартизации

KPA (Key Process Area) — основная группа процессов

LOC (Lines of Code) — количество строк кода

MSF (Microsoft Solutions Framework) — технология компании Microsoft

NATO (North-Atlantic Treaty Organization) - НАТО, Североатлантический союз

OCL (Object Constraint Language) - язык объектных ограничений

OMG (Object Management Group) - Организация по стандартизации в области объектно-ориентированных методов и технологий

OMT (Object Modeling Technique) - метод объектного моделирования

OOSE (Object-Oriented Software Engineering) - объектно-ориентированная разработка программного обеспечения

PDSA (Plan-Do-Study-Act) - планирование, реализация, изучение и действие

PERT (Program Evaluation-and-Review Technique) - метод оценки и пересмотра планов

PIN (Personal Identification Number) - личный код в банковской системе

PMI (Project Management Institute) - Институт управления проектами

RAD (Rapid Application Development) - быстрая разработка приложений

RET (Record Element Type) — элементарная запись RPW (Rational Process Workbench) - набор инструментов и шаблонов для настройки и публикации Web-сайтов на основе RUP

RUP (Rational Unified Process) - унифицированный процесс SADT (Struc-

Structured Analysis and Design Technique) - метод структурного анализа и проектирования

SEI (Software Engineering Institute) - Институт программной инженерии

SEPG (Software Engineering Process Group) - группа по разработке процессов создания ПО

SLOC (Source Lines of Code) - количество строк исходного кода

SoDA (Software Document Automation) — автоматизированное документирование ПО

SPMN (Software Program Managers Network) - сеть Министерства обороны США для менеджеров проектов

SPR (Software Productivity Research) — название компании

SQL (Structured Query Language) - структурированный язык запросов

TCP/IP (Transmission Control Protocol/Internet Protocol) - протокол управления передачей/протокол Интернет

TCF (Technical Complexity Factor) — техническая сложность проекта

UCP (Use Case Points) — количество вариантов использования

UFP (Unadjusted Function Points) — общее количество функциональных точек без учета поправочного коэффициента

UML (Unified Modeling Language) — унифицированный язык моделирования

UOW (Unit of Work) - единица работы

UUCP (Unadjusted Use Case Points) - количество вариантов использования без учета поправочного коэффициента

VAF (Value Adjustment Factor) - поправочный коэффициент

XP (Extreme Programming) — экстремальное программирование

SOFTWARE DESIGN

Second Edition A. Vendrov

Moscow, «Finansy i statistika» Publishing House, 2004

The textbook examines «state-of-the-art» in software design methods and tools. It bases on the international standards, first of all on ISO 12207 «Software life cycle processes». Special attention is paid to the structural and object-oriented approaches to the business modeling, software requirements specification, analysis and design. Key features;

- Application of a standard modeling language UML.
- Practical introduction to object-oriented analysis and design using the Unified Software Development Process and showing how it can be applied in a relatively simple case study.

The structure and emphasis in this book are based on years of experience in training and teaching hundreds of students and software developers.

Intended Audience: students in computer science or software engineering courses, system analysts, software developers and project managers.

About the author:

Alexander Vendrov is a consultant and lecturer in Moscow State University. He is known as an expert in software engineering methods and tools, including CASE, structured and object-oriented analysis and design as well as UML modeling.

He is Ph.D. (technical sciences) the author of monograph «CASE-technology* (1998), textbooks «Software Design* (2000), «A practical work on Software Design* (2002), and coauthor of «Database and knowledge base management systems* (1991).

He is also the interpreter of two books: «UML Distilled* by M. Fowler (1997) and «Death March* by Ed. Yourdon (1997, 2003), editor of some well-known books (Russian versions), including «Mastering UML with Rational Rose* by W Boggs, M. Boggs (1999), «Writing Effective Use Cases* by A. Cockburn (2000), «Software Project Management. A Unified Framework* by W Royce (1998) and «Agile Software Development* by A. Cockburn (2001), and technical editor of «UML: A Be-

ginner's Guide* (J. T. Roff, McGraw-Hill/Osborne, 2003).