Глава 28

Теоретически обоснованные формальные методы спецификации требований

Основные положения

- Формальные методы спецификации требований применяются тогда, когда описание требований слишком сложно для естественного языка или вам не удается создать недвусмысленную спецификацию.
- К формальным методам относится использование псевдокода, конечных автоматов, деревьев решений, диаграмм деятельности, моделей сущность-связь, объектно-ориентированных моделей и схем потоков данных.

До сих пор мы исходили из предположения, что большинство требований будет написано на естественном языке команды разработчиков (в форме традиционных утверждений или прецедентов). Кроме того, предлагалось сопроводить требования диаграммами, таблицами, схемами и т.п., чтобы прояснить значение требований пользователя. Но иногда присущая естественному языку неоднозначность просто неприемлема, особенно когда требования касаются жизненно важных вопросов или когда неправильное поведение системы может привести к чрезвычайным экономическим или юридическим последствиям. Если определение требования сложно сформулировать на естественном языке и невозможно предотвратить неправильное понимание спецификации, следует попытаться написать эту часть требований с помощью теоретически обоснованных формальных методов.

Можно выбрать одно из перечисленных ниже формальных средств спецификации

- Псевдокод
- Конечные автоматы
- Деревья решений
- Диаграммы деятельности(блок-схемы)
- Модели сущность-связь
- Объектно-ориентированные модели
- Схемы потоков данных

Мы не будем подробно описывать применение какого-либо из этих средств, так как каждое из них достойно отдельной книги. Мы просто предлагаем краткий их обзор.

В Modern SRS Package формальные методы следует использовать экономно и последовательно. При выборе конкретного формального метода необходимо руководствоваться здравым смыслом. При создании системы управления ядерным реактором, возможно, каждый аспект системы является критическим; однако в большинстве систем не более 10 процентов требований требуют такой степени формальности.

Если это возможно, следует использовать только один из этих формальных методов для всех требований определенной системы. Это упростит не очень подготовленному читателю задачу прочтения и понимания элементов пакета. Если все разрабатываемые организацией системы относятся к одной прикладной области, такой, например, как телефонные системы коммутации, то можно использовать один и тот же формальный метод для всех систем. Но в большинстве организаций нереально придерживаться единственного метода для всех требований во всех системах; тот, кто пишет требования, должен выбрать подход, наиболее соответствующий конкретной ситуации.

Псевдокод

Как и подразумевает его название, псевдокод — это квазиязык программирования; попытка соединить неформальный естественный язык со строгими синтаксическими и управляющими структурами языка программирования. В чистом виде псевдокод состоит из комбинации следующих элементов.

- Императивных предложений с одним глаголом и одним объектом.
- Ограниченного множества (как правило, не более 40–50) "ориентированных на действия" глаголов, из которых должны конструироваться предложения.
- Решений, представленных формальной структурой `IF-ELSE-ENDIF`.
- Итеративных действий, представленных структурами `DO-WHILE` и `FOR-NEXT`.

На рис. 28.1 представлен пример спецификации с помощью псевдокода алгоритма вычисления отложенного дохода от услуг в течение данного месяца в бизнес-приложении. Фрагменты текста на псевдокоде расположены уступами; такой формат используется для того, чтобы выделить логические "блоки". Сочетание синтаксических ограничений, формата и разбивки существенно уменьшает неоднозначность требования, которое в противном случае было бы очень сложным и расплывчатым. (Так оно и было до написания псевдокода!) В то же время такое представление требования вполне понятно для человека, не являющегося программистом. Не нужно быть выдающимся ученым, чтобы понимать псевдокод, и нет необходимости знать C++ или Java.

```
Àëãîðèòì âû÷èñëåíèÿ îòëîæåííîãî äîõîäà îò óñëóã â òå÷åíèå ìåñÿöà:
Set Sum(X)=0
FOR êàæäîãî êëèåíòà X
  IF êëèåíò îïëàòèë óñëóãè âîâðåä
    AND ((Òåêóùèé ìåñÿö)>=(2 ìåñ. ïîñëå äàòû ïðèîáðåòåíèÿ))
    AND ((Òåêóùèé ìåñÿö)<=(14 ìåñ. ïîñëå äàòû ïðèîáðåòåíèÿ))
  THEN Sum(X) = Sum(X) + (ñóììà, çàïëà÷åííàÿ êëèåíòîì)/12
```

Рис. 28.1. Пример спецификации с использованием псевдокода

Конечные автоматы

В некоторых ситуациях систему или ее дискретное подмножество удобно рассматривать как "гипотетическую машину, которая в конкретный момент времени может находиться только в одном из указанных состояний" (Дэвис (Davis, 1993)). В ответ на ввод, будь то данные, вводимые пользователем, или информация, поступившая от внешнего устройства, машина изменяет свое состояние и генерирует выводимую информацию или выполняет некое действие. Как вывод, так и следующее состояние можно определить, основываясь исключительно на знании текущего состояния и события, вызывающего транзакцию. Таким образом, поведение системы можно назвать детерминированным; можно математическим путем определить все возможные состояния и, как следствие, выводы системы, основываясь на множестве предлагаемых вводов.

Разработчики аппаратного обеспечения десятилетиями использовали конечные автоматы (Finite state machines, FSMs). Существует огромное количество литературы, описывающей создание и анализ таких автоматов. Математическая природа FSMs располагает к проведению строго формального анализа, поэтому описанные ранее в данной части проблемы непротиворечивости, полноты и неоднозначности значительно уменьшаются при их использовании.

Конечные автоматы зачастую представляются в виде диаграмм перехода состояний, как показано на рис. 28.2. В этой системе обозначений прямоугольники представляют состояние, в котором находится устройство, а стрелки – действия, переводящие устройство в другие состояния. Рисунок иллюстрирует переходы состояний "лампового ящика", рассматривавшегося в главе 26. В том примере выражение на естественном языке "лампа будет мигать каждую секунду" было несколько неоднозначным. Представленная на рис. 28.2 диаграмма перехода состояний не является неоднозначной и иллюстрирует, что цикл B действительно был выбран правильно. Если лампа перегорает, прибор чередует попытки включить четную и нечетную лампы; каждую в течение 1 секунды.

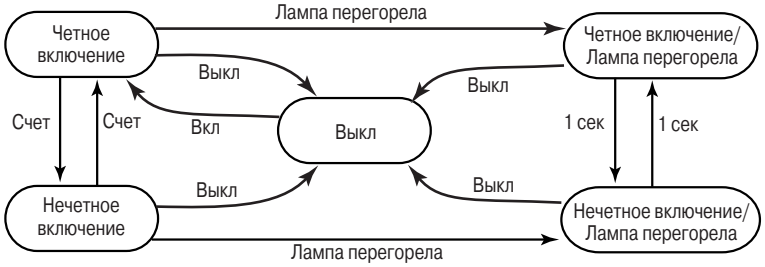

Рис. 28.2. Диаграмма перехода состояний

Предлагаем вам применить FSM для повторного определения прецедента "Управление освещением" системы HOLIS. Вы сразу заметите, что альтернативный поток Dim (изменение яркости) прекрасно подходит для представления с помощью FSM.

В более строгой форме конечный автомат представляется в виде таблицы или матрицы, где показаны все состояния, в которых может находиться устройство, вывод системы для каждого состояния и воздействия всех возможных событий на каждое возможное состояние. Это приводит к более высокому уровню детализации, так как каждое состояние и воздействие каждого события должны быть представлены в таблице. Например, табл. 28.1 определяет поведение осветительной коробки в виде матрицы перехода состояний.

Таблица 28.1. Матрица перехода состояний для прибора подсчитывающего четное/нечетное число нажатий

	Событие					
Состояние	**Нажатие *Вкл***	**Нажатие *Выкл***	**Нажатие *Счет***	**Лампа перегорает**	**Каждая секунда**	**Вывод**
Выключен	Четная горит	—	—	—	—	Обе лампы выключены
Четная горит	—	Выключен	Нечетное включение	Лампа перегорела/Четная горит	—	Четная горит
Нечетная горит	—	Выключен	Четное включение	Лампа перегорела/Нечетная горит	—	Нечетная горит
Лампа перегорела/Четная горит	—	Выключен	—	Выключен	Лампа перегорела/ Нечетная горит	Четная горит
Лампа перегорела/Нечетная горит	—	Выключен	—	Выключен	Лампа перегорела/ Четная горит	Нечетная горит

В таком представлении можно разрешить дополнительные неоднозначности, которые могут возникнуть при попытке понять поведение прибора.

- Что происходит, если пользователь нажимает кнопку *Вкл*, когда прибор уже включен? *Ответ:* ничего.
- Что происходит, если обе лампы перегорают? *Ответ:* прибор выключается.

Конечные автоматы широко применяются для некоторых категорий системных программных приложений, таких как системы обмена сообщениями, операционные системы и системы управления процессами. Они также являются удобным средством описания взаимодействия между внешним пользователем-человеком и системой, например взаимодействия клиента банка и банкомата, когда клиент хочет снять деньги со счета. Но конечные автоматы становятся слишком громоздкими, если нужно представить поведение системы как функцию *нескольких* вводов. В таких случаях требуемое поведение системы, как правило, является функцией всех текущих условий и событий, а не одного текущего события или некой цепочки прежних событий.

Таблицы решений

Достаточно часто встречаются требования, связанные с комбинацией вводов; различные комбинации вводов приводят к различным вариантам поведения или вывода. Предположим, имеется система с пятью возможными вводами (A, B, C, D, E) и требование, заключающееся в утверждении, напоминающем оператор псевдокода: “Если A истинно, то, если B и C также истинны, генерировать вывод X при условии, что E не является истинным, иначе требуемый

вывод — Y". Комбинация предложений IF-THEN-ELSE (если-то-иначе) быстро становится запутанной, особенно если, как и в этом примере, имеются вложенные предложения IF. Как правило, обычные пользователи не могут их понять, и невозможно гарантировать, что все возможные комбинации A, B, C, D и E рассмотрены.

Решением в данном случае является перечисление всех комбинаций вводов и описание каждой из них в явном виде в таблице. В нашем примере, когда допустимых значений вводов только два ("истинно" и "ложно"), получается 2^5 или 32 комбинации. Их можно представить с помощью таблицы, содержащей 5 строк (по одной для каждой вводимой переменной) и 32 столбца.

Графические деревья решений

Дерево решений применяется для графического отображения информации. Мы использовали это представление в главе 15, когда нужно было принять решение, какой прототип создавать. На рис. 28.3 показано дерево решений, используемое для описания последовательности действий системыHOLIS в чрезвычайной ситуации.

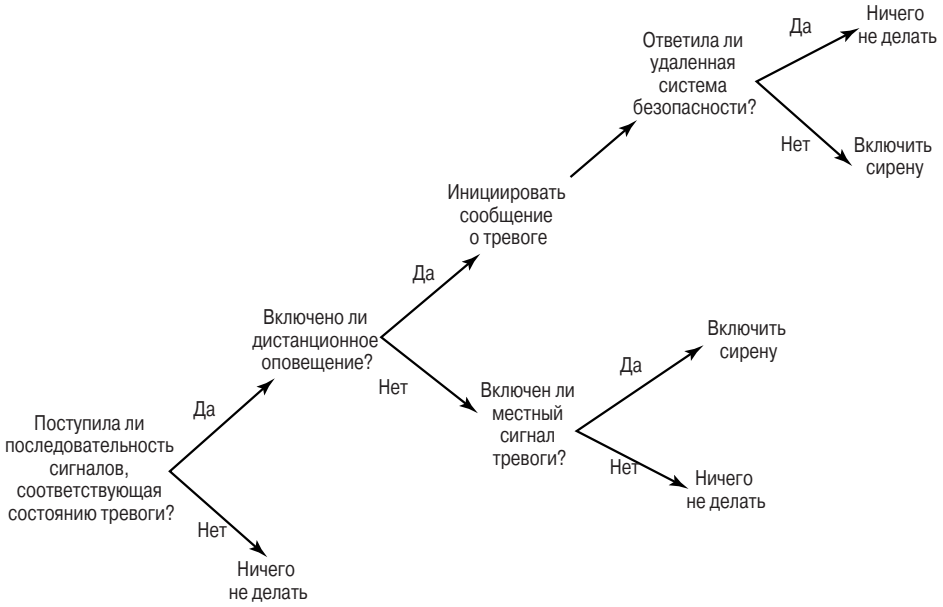

Рис. 28.3. Графическое дерево решений

Диаграммы деятельности

Блок-схемы (и их разновидность, UML-диаграммы деятельности) имеют несомненное преимущество — они достаточно известны. Даже люди, далекие от всего, что связано с компьютерами, знают, что такое блок-схема. Например, местная газета недавно поместила блок-схему, описывающую алгоритм, с помощью которого человеческий мозг принимает решение о покупке кабриолета SAAB. По понятным причинам все пути в этой блок-схеме заканчивались одним и тем же решением: "Купить SAAB". Где-то должна была быть логическая ошибка, но мы не смогли ее найти. Нам действительно нравится эта машина!

На рис. 28.4 показана типичная диаграмма деятельности в принятых в UML обозначениях. Хотя ту же информацию можно представить в форме псевдокода, UML обеспечивает визуальное представление, которое проще понять.

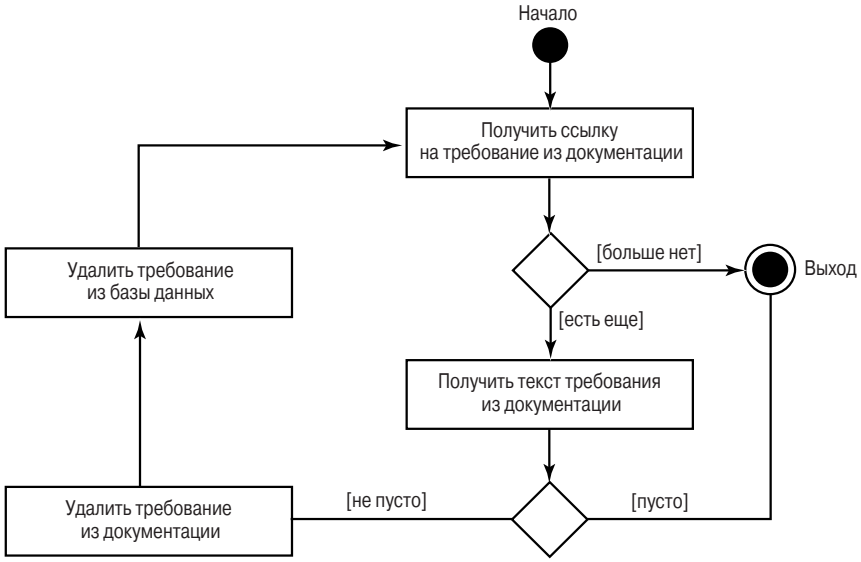

Рис. 28.4. Диаграмма деятельности

Единственная проблема при использовании диаграмм деятельности, как выяснили программисты за последние 30 лет, состоит в том, что достаточно скучно поддерживать их соответствие современному состоянию разработки. Такая проблема существует для всех графических представлений при отсутствии автоматических средств; никому не хочется перерисовывать диаграмму перехода состояний или дерево решений.

Модели сущность-связь

Если набор требований описывает структуру *данных* системы и связи между ними, удобно представить эту информацию в виде диаграммы сущность-связь (Entity-relationship diagram, ERD). На рис. 28.5 представлена типичная ERD.

Диаграмма сущность-связь обеспечивает высокоуровневое "архитектурное" представление данных (на рисунке – это заказчики, счета-фактуры, заказы и т.д.). Ее можно затем дополнить соответствующими подробностями, содержащими необходимую информацию (например, описание заказчика). В ERD основное внимание уделяется внешнему поведению системы, что позволяет ответить на такие вопросы, как "Можно ли в счет-фактуре указать более одного адреса выставления счета?" *Ответ:* нет.

Хотя модели сущность-связь являются мощным инструментом моделирования, они имеют существенный недостаток, который заключается в том, что далекому от техники читателю трудно их понять. Как видно из рис. 28.5, линии, соединяющие *заказчика* и *заказ*, а также *заказ* и *счет-фактуру*, отмечены кружками и "птичьими лапками". Возникает вопрос: *что это все означает?* Попытка ответить на него в рам-

ках данной книги была бы значительным отклонением от темы, чего мы решили избежать. Но если уклониться от ответа на этот вопрос при описании множества требований, это может привести к тому, что некоторые пользователи просто не поймут, что происходит. Можно организовать для пользователей двухдневные курсы по принятой в ERD-моделях системе обозначений, но неизвестно, воспримут ли они ее. Можно использовать ERD в качестве "формальной" формы документации в среде разработчиков.

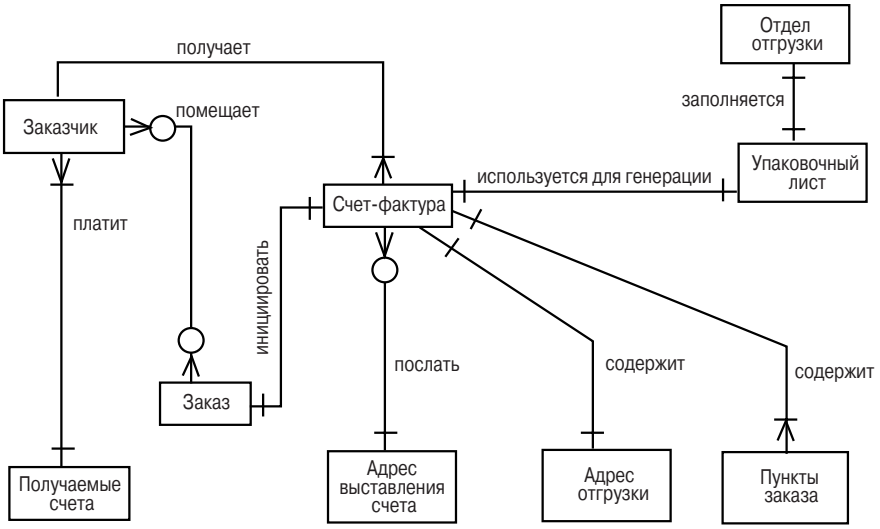

Рис. 28.5. Диаграммасущность-связь

Объектно-ориентированные модели

Если детально разрабатываемые требования должны содержать описание структуры и связей *сущностей* системы (например, расчетных листов, служащих, сотрудников отдела заработной платы и т.д), для более полного описания поведения системы можно использовать объектно-ориентированные модели. При современном уровне популярности объектно-ориентированных методов и быстром внедрении языка UML эти модели постепенно превращаются в спецификации и, более того, в реализационные модели, используемые при реализации функциональных возможностей системы.

Удобство заключается в том, что применение стандартов UML обеспечивает всем общее понимание того, что означает данное представление, и тем самым уменьшает неоднозначность, заставляя всех "говорить на одном языке", пусть и техническом.

Например, объект "Сотрудник" на рис. 28.6 будет описываться с помощью содержащихся в нем ориентированных на данные *атрибутов*, таких как название подразделения и должность, а также предоставляемых им *услуг*, таких как добавление нового сотрудника, удаление сотрудника и поиск конкретного экземпляра сотрудника.

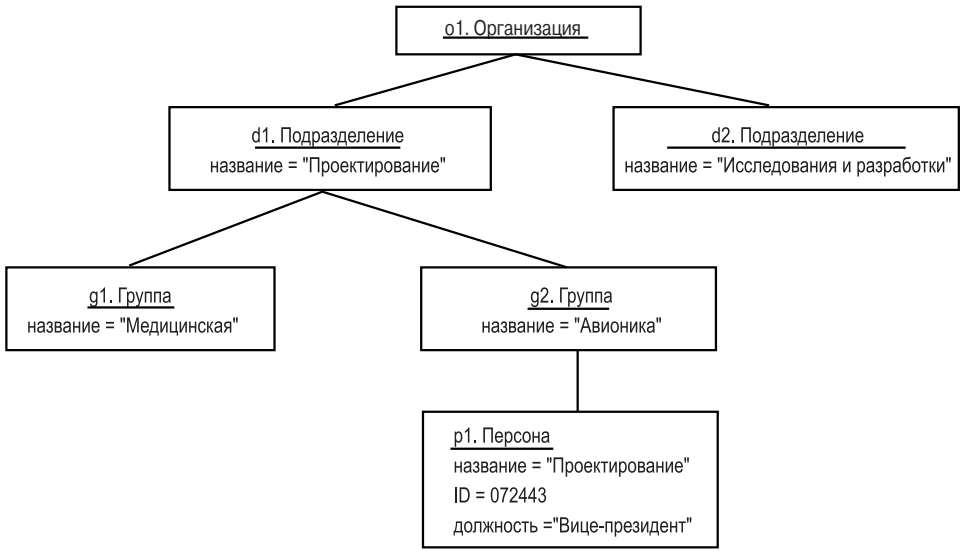

Рис. 28.6. Объектно-ориентированная модель

Схемы потоков данных

При обсуждении требований в данной главе предполагалось, что мы имеем дело с требованиями "атомарного уровня". Как правило, так и бывает в типичном документе, но часто полезно иметь визуальное представление, иллюстрирующее структуру и организацию этих атомарных требований, а также связей ввода-вывода между ними. Для этого широко используется представление подобной информации в виде схемы потоков данных (рис. 28.7).

Модели, использующие схемы потоков данных (DFD), испытывают те же трудности, что и модели сущность-связь (ERD), хотя далекому от техники читателю обычно несколько проще понять значение DFD без специальной подготовки. Некоторые организации добились значительного успеха, используя DFD в качестве основы общения нетехнически ориентированных пользователей и разработчиков; в то время как другие обнаружили, что их пользователи блокируют любые попытки использовать столь "формальные" обозначения. Если схему потоков данных *в принципе удается* использовать, то возможно провести декомпозицию каждого из "кружков" (рис. 28.7) на DFD более низкого уровня. Так требования для кружка 5 ("закупаемые ресурсы") можно описать подробнее с помощью схемы более низкого уровня, которая иллюстрирует соответствующие детали. Процесс декомпозиции продолжается до тех пор, пока кружки действительно станут "атомарными"; на этом уровне связанные с кружком требования уже можно описать с помощью псевдокода, конечного автомата, дерева решений или блок-схем. (На самом деле сегодня существует более серьезная опасность при использовании DFD; приверженцы объектно-ориентированного программирования могут посчитать, что вы занимаетесь функциональной декомпозицией данных и, следовательно, являетесь ретроградом, и в дальнейшем будут игнорировать все, что вы скажете по любому поводу.)

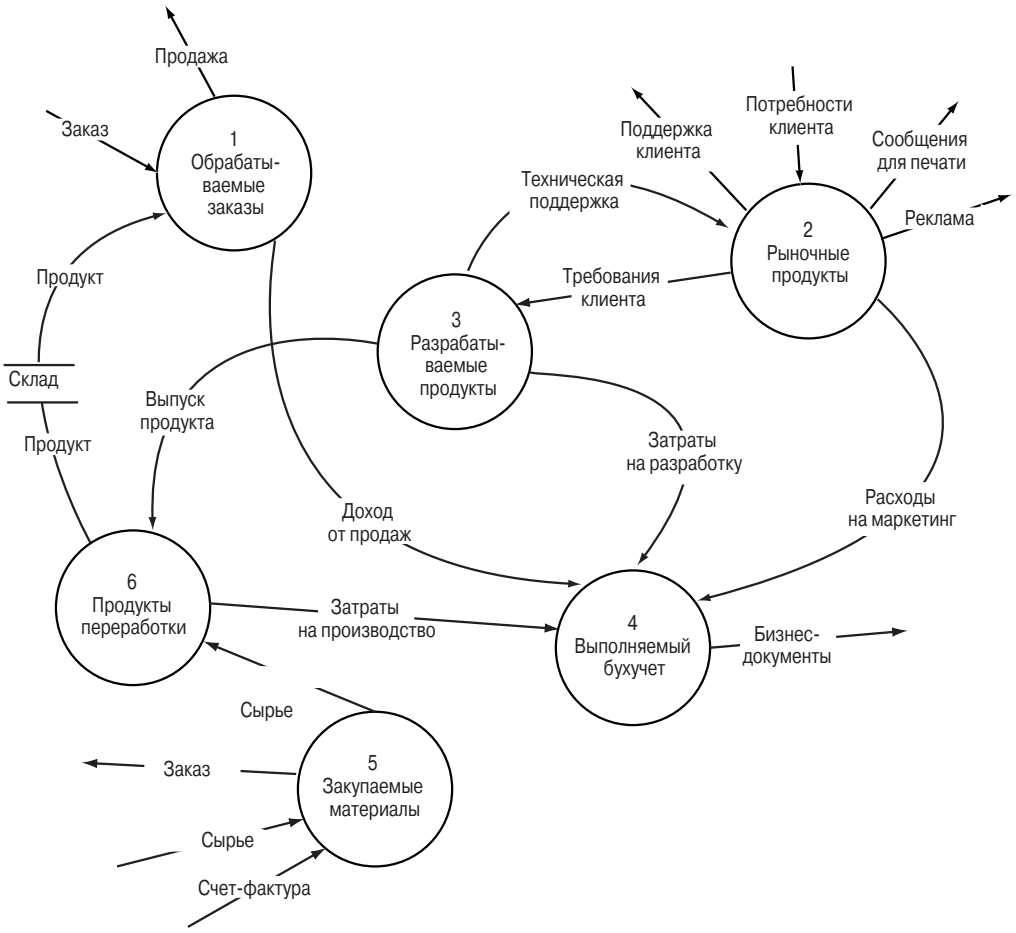

Рис. 28.7. Схема потоков данных

Ведение спецификаций

Работая с требованиями, мы не злоупотребляем этими формальными методами и используем их только для того, чтобы проиллюстрировать поведение системы. Это значительно уменьшает головную боль от необходимости их сопровождения. Кроме того, новое поколение автоматических средств разработки программ обеспечивает существенную поддержку прямого и обратного проектирования, т.е. они предоставляют возможность автоматически поддерживать синхронизацию кода и представления в модели. По мере развития этих средств появится возможность рассматривать изменения требований как принятые в процессе кодирования решения, которые влияют на внешнее поведение системы.

> Когда дело касается сопровождения, на первом плане должен быть здравый смысл.

Типичная проблема сопровождения возникает, когда код или спецификации пересматриваются, а соответствующие формальные спецификации одновременно не исправляются. Теория гласит: “код — это спецификация”. Мы не утверждаем, что абсолютным правилом является *обязательное* обновление формальных спецификаций по мере развития проекта. Но тех, кто случайно обратится к устаревшей документации, подстерегает множество ловушек. С нами это также случалось в процессе написания данной книги, когда мы говорили, что “не нужно обновлять этот рисунок”, а затем по ошибке использовали устаревшую версию.

В том, что касается обновления, следует руководствоваться здравым смыслом. Обновляйте то, что необходимо, основываясь на важности данной информации. Если это не является жизненно важным, может быть, можно отказаться от обновления. Один из методов, который мы в свое время использовали, состоит в том, чтобы при принятии решения не обновлять некий документ, а делать на нем пометку “Устаревший”.

Мы считаем устаревшую модель меньшим из двух зол. Лучше иметь устаревшую модель, чем не иметь модели спецификаций вовсе!

Рабочий пример

Все эти методы рассматривались командой проекта HOLIS при подготовке пакета HOLIS SRS Package. Первая версия данного пакета представлена среди артефактов системы HOLIS в приложенииA.

Заключение части 5

В части 5 мы выяснили, что требования должны полно и сжато фиксировать потребности пользователей в таком виде, чтобы разработчик мог построить удовлетворяющее их приложение. Кроме того, требования должны быть достаточно конкретными, чтобы можно было определить, когда они удовлетворены. Зачастую именно команда обеспечивает эту конкретизацию; это еще одна возможность убедиться, что определяется правильная система.

Существуют различные возможности организации и документирования этих требований. Мы остановились на пакете Modern SRS Package, логической конструкции, которая позволяет документировать требования в виде прецедентов, документов, форм баз данных и т.д. Хотя мы внесли несколько предложений относительно того, как организовать такой пакет, мы считаем, что форма не так важна, лишь бы пакет содержал все, что необходимо.

Вся разработка должна вытекать из требований, зафиксированных в пакете Modern SRS Package, а все спецификации пакета должны найти отражение в действиях разработки. Таким образом, все виды деятельности являются отражением пакета и наоборот. Пакет Modern SRS Package является "живой" сущностью; его следует пересматривать и обновлять на протяжении жизненного цикла проекта. В пакете указывается, *какие* функции должны осуществляться, а не то, *как* они осуществляются. Он используется для задания функциональных требований, нефункциональных требований и ограничений проектирования.

Мы также предложили набор показателей, которые можно использовать для оценки качества пакета и содержащихся в нем элементов. Если необходимо, документация требований может дополняться одним или несколькими более формальными либо более структурированными методами спецификации.

Пакет Modern SRS Package содержит детали, которые необходимы *для построения (реализации)* правильной системы. Эту часть работы над проектом мы обсудим в части 6, "Построение правильной системы".