Глава 2

Основания анализа требований

В качестве основного метода создания ПО в данной книге принят объектно-ориентированный подход. Этот подход не дается легко, поскольку объектно-ориентированная разработка требует хорошего знания объектной технологии. Без глубокого понимания тонкостей объектной технологии разработчик не сможет правильно применять язык UML в качестве универсального и привычного средства моделирования.

Основная трудность изучения объектной технологии связана с отсутствием очевидной отправной точки и ясного направления исследования. Это не хорошо известные нам подходы к изучению систем "сверху-вниз" или "снизу-вверх". По существу, рассматриваемый подход представляет собой что-то наподобие "все время со средины". Не зависимо от того, насколько мы продвинулись в процессе изучения, кажется, что мы всегда находимся посередине этого процесса (поскольку все время возникают новые вопросы). Можно считать, что в процессе изучения достигнут первый успех, когда читатель осознает глубинный смысл того факта, что в объектно-ориентированной системе "все есть объект".

Данная глава направлена на то, чтобы дать читателю необходимый базовый уровень знаний об изучаемом предмете прежде, чем приступить к его более глубокому изучению. В этой главе разъясняется, выполнение каких предварительных условий необходимо для анализа требований, при этом одновременно используются два способа подачи материала. Во-первых, здесь объясняются основы объектной технологии. Во-вторых, проводится "обучение на примерах" и дается наставление по моделированию анализа с использованием приложения *Internet-магазин*, относящегося к известной прикладной области — "Электронная торговля".

2.1. Основы объектной технологии

Для объяснения сути объектной ориентации информационных систем воспользуемся аналогией с объектами реального мира. Окружающий мир состоит из *объектов* (*object*), пребывающих в *состоянии* (*state*), которое определяется текущими значениями атрибутов объекта.

Например, кружка на моем столе находится в *состоянии* `filled` (наполнена), поскольку она приспособлена для хранения жидкостей и в ней все еще есть кофе. Когда в ней нет больше кофе, состояние кружки можно определить как `empty` (пуста). Если она упадет на пол и разобьется, она перейдет в состояние `broken` (разбита).

Моя кофейная чашка, конечно, пассивна — она не обладает собственным *поведением* (*behavior*). Однако, этого нельзя сказать о моей собаке или эвкалиптовом дереве за моим окном. Моя собака лает, дерево растет и т.д. Итак, некоторые объекты реального мира обладают поведением.

Все объекты реального мира обладают также *идентичностью* (*identity*)— постоянным свойством, с помощью которого мы отличаем один объект от другого. Если на моем столе стоят две чашки из одного набора, я могу сказать, что они *одинаковые*, но *не идентичные*. Чашки одинаковые, потому что значения их свойств совпадают (они одинакового размера и формы, черного цвета и пустые). Однако, на объектно-ориентированном языке они не идентичны, поскольку их две, и у меня есть выбор, которую из них использовать.

Реальные объекты, обладающие тремя свойствами (состояние, поведение, идентичность), образуют *системы с естественным поведением*. Естественные системы безусловно являются самыми *сложными системами* из всех известных. Никакая компьютерная система не может сравниться по сложности с животным или заводом.

Несмотря на сложность, естественные системы способны работать: они демонстрируют интересное поведение, могут приспосабливаться к внешним и внутренним изменениям, могут эволюционировать со временем и т.д. Вывод очевиден. Наверное, мы должны конструировать искусственные системы с помощью моделирования структуры и поведения естественных систем (сравн. [55]).

Искусственные системы являются моделью реальности. Кофейная чашка на экране моего компьютера всего лишь модель реальной "сущности" так же, как собака или эвкалиптовое дерево на моем экране. Кофейная чашка может быть, таким образом, смоделирована с помощью поведенческих свойств. Она может, к примеру, упасть на пол, если ее уронить. Действие "падения" можно смоделировать как поведенческую *операцию* (*operation*) чашки. Еще одним логическим "действием" чашки может быть операция "разбиться" при ударе об пол. Большинство, если не все, объекты в компьютерной системе "оживают" — они обладают поведением.

2.1.1. Объект-экземпляр

Объект — это *экземпляр* (*instance*) некоей "сущности". Он может быть одним из множества экземпляров одной и той же "сущности". Моя чашка — экземпляр множества всевозможных чашек.

Общее описание "сущности" называется *классом* (*class*). Поэтому объект является экземпляром класса. Однако, как мы увидим в разделе 2.1.6, класс также может нуждаться в конкретизации — он может быть объектом. Поэтому нам необходимо различать *объект-экземпляр* (*instance object*) и *объект-класс* (*class object*).

Для краткости объект-экземпляр часто называют *объектом* или *экземпляром*. Название "экземпляр объекта" сбивает с толку. Объектно-ориентированная система состоит из взаимодействующих объектов. В объектно-ориентированной системе нет ничего, кроме объектов, будь-то объект экземпляра (*объект-экземпляр*) или объект класса (*объект-класс*).

В качестве отступления и в продолжение темы заметим, что мы не настаиваем на использовании термина "класс объекта". Конечно, класс представляет собой шаблон для объектов с аналогичными атрибутами и операциями, но сам класс также может быть конкретизирован в виде объекта (и мы не хотим называть такое образование "объект класса объекта").

2.1.1.1. Объектная нотация

В языке UML объект обозначается прямоугольником с двумя отделениями. Верхнее отделение содержит имя объекта и имя класса, которому принадлежит объект. Синтаксис этой конструкции выглядит так:

```
objectname: classname.
```

Нижнее отделение содержит список имен атрибутов и значений. С помощью этого синтаксиса можно также показать типы атрибутов:

```
attributename: type = value.
```

На рис. 2.1 показан объект `Course` (Дисциплина) с именем `c1`. Объект обладает двумя атрибутами. Типы атрибутов не показаны — они заданы в определении класса.

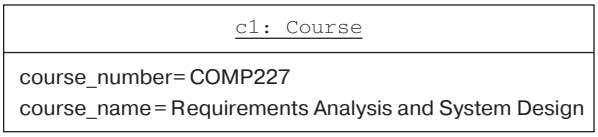

Рис. 2.1. Объект-экземпляр

Важно иметь в виду, что объектная нотация не предусматривает в обозначении объекта особого "отделения" для перечня *операций*, которые может выполнять объект-экземпляр. Это связано с тем, что операции, выполняемые всеми объектами-экземплярами, идентичны, и поэтому хранить их в каждом объекте-экземпляре было бы накладно. Операции можно хранить в объекте-классе или же можно связать их с объектами-экземплярами другими средствами (реализованными в ПО базовой объектно-ориентированной системы).

2.1.1.2. Как объекты кооперируются

Количество объектов определенного класса может быть очень большим. Показать большое количество объектов на диаграмме нереально, а иногда просто невозможно. Объекты изображаются только для того, чтобы привести пример системы в определенный момент времени или проиллюстрировать, каким образом они *кооперируются* (*collaborate*) с течением времени при выполнении определенных задач. Например, чтобы упорядочить товары, может потребоваться установить *кооперацию* между объектом `Stock` (Склад) и объектом `Purchase` (Покупка).

Системные задачи выполняются объектами, которые активизируют *операции* (поведение) друг друга. Мы говорим, что они обмениваются *сообщениями* (*message*). Сообщения запускают операции на объектах, которые могут приводить к изменению состояний объектов и вызывать другие операции.

Рис. 2.2 иллюстрирует поток сообщений между четырьмя объектами. Скобки после имени сообщения указывают на то, что сообщение может принимать параметры (аналогично вызову функции в традиционном программировании). Объект `Order` (Заказ) предписывает объекту `Shipment` (Поставка) доставить заказ. Для этого объект `Shipment` инструктирует объект `Stock` о необходимости вычесть соответствующее количество товаров. Затем объект `Stock` анализирует новый уровень запаса и, если он оказывается ниже определенного значения, предписывает объекту `Purchase` сделать повторный заказ дополнительного объема товаров.

Хотя приведенное объяснение кооперации объектов выглядит как последовательность видов деятельности, а сообщения даже пронумерованы, в общем случае на порядок, в котором активируются объекты, не накладывается строгих ограничений. Например, сообщения `analyzeStockLevels` (проанализировать уровни запаса) и `reorderProducts` (повторить заказ товара) могут выполняться в любой последовательности, возможно, независимо от сообщений `shipOrder` (доставить заказ) и `subtractProduct` (уменьшить количество товара). Поэтому в дальнейшем при рассмотрении кооперации объектов на уровне реализации мы откажемся от нумерации сообщений (раздел 6.2).

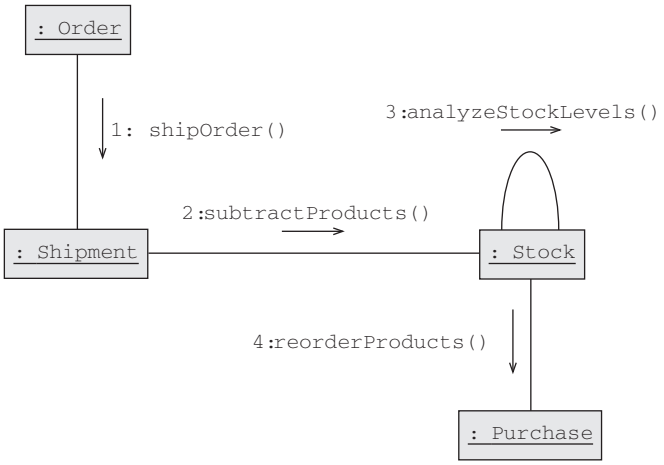

Рис. 2.2. Кооперация объектов

2.1.1.3. Как объекты идентифицируют друг друга

Вопрос состоит в том, каким образом объект узнает об *идентичности* другого объекта, которому требуется отправить сообщение. Каким образом объект `Order` узнает свой объект `Shipment` так, чтобы сообщение `shipOrder` попало к своему адресату?

Ответ заключается в том, что каждому объекту при создании присваивается *идентификатор объекта* (*object identifier — OID*). Идентификатор объекта (OID) представляет собой *дескриптор* (*handle*) объекта — уникальный номер, который остается с объектом на протяжении всего времени его существования. Если объекту `X` необходимо отправить сообщение объекту `Y`, объект `X` каким-либо образом должен узнать OID объекта `Y`.

На практике для установления связи по OID между объектами существует два подхода, каждому из которых соответствует определенный тип связи.

- Постоянная связь по OID.
- Временная связь по OID.

Различие между этими видами связи определяется продолжительностью существования объекта. Время жизни некоторых объектов не превышает времени выполнения программы — они создаются программой и уничтожаются во время выполнения программы или по ее завершении. Это так называемые *временные объекты* (*transient object*). Другие объекты "переживают" выполнение программы — после завершения программы они запоминаются в долговременной дисковой памяти и доступны при следующем выполнении программы. Это так называемые *постоянные объекты* (*persistent object*).

(Временные объекты называют также *короткоживущими*, а постоянные — *долгоживущими*, *постоянного хранения* или *энергонезависимыми*. Последнее название связано с тем, что объекты этого типа не исчезают при отключении питания компьютера, сохраняясь в энергонезависимой памяти, как правило, на дисковых устройствах. *Прим. ред.*)

2.1.1.3.1. Постоянная связь

Постоянная связь (*persistent link*) — это ссылка на объект (или множество ссылок на объекты), принадлежащая одному из объектов, хранимых в долговременной памяти, которая связывает этот объект с другим объектом в долговременной памяти (или со множеством других объектов). Поэтому, для установления постоянной связи объекта `Course` (Курс) с его объектом `Teacher` (Преподаватель) объект `Course` должен содержать атрибут связи, значение которого равно OID объекта `Teacher`. Описанная связь постоянна, поскольку OID физически хранится в объекте `Course`, как показано на рис. 2.3.

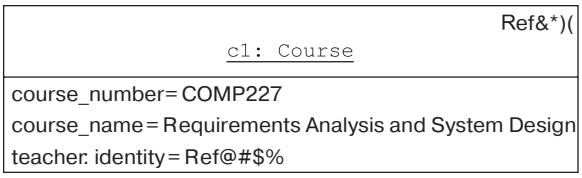

Рис. 2.3. Реализация постоянной связи

OID объекта `c1` помечен здесь как `Ref&*)(`. Объект содержит атрибут связи под именем `teacher`. Данный атрибут принадлежит типу `identity`. Его значение равно магическому числу `Ref#$%` —адресу дисковой памяти, где *постоянно* хранится объект `Teacher`.

После того, как объекты `Course` и `Teacher` перемещаются в память программы, значение атрибута `teacher` будет "втянуто" в указатель памяти, в результате устанавливается кооперация между объектами на уровне памяти. (Выражение "втягивать" (swizzling, по англ. букв. "тянуть коктейль") — это не термин языка UML, он используется в объектных базах данных (см. раздел 6.2), где перемещение объектов между долговременной и оперативной памятью довольно частое явление.)

На рис. 2.3 показана типичная реализация постоянной связи. Однако, связи между объектами для случая моделирования с помощью языка UML показаны на рис. 2.4. Связи представляются как *экземпляры ассоциации* (*instances of association*) между объектами `Course` и `Teacher`.

Обычно кооперативные связи допускают *передвижение* (*navigation*) в обоих направлениях. Каждый объект `Course` связан со своим объектом `Teacher`, а объект `Teacher` может привести к объектам `Course`. Изредка допускается передвижение только в одном направлении.

После того как объект связан с другим объектом на постоянной основе, он может отправить по этой связи сообщение, чтобы запросить у другого объекта *сервис* (*service*). Это значит, что объект вызывает операцию на другом объекте за счет отправки ему *сообщения*. При типичном сценарии для указания объекта отправитель использует программную переменную, содержащую значение связи (значение OID) этого объекта.

Например, сообщение, отправленное объектом `Teacher` для того, чтобы определить имя объекта `Course`, может иметь следующий вид:

```
crs-ref.getCourseName(out crs_name)
```

В приведенном примере определенный объект класса Course, который будет выполнять операцию getCourseName, указывается текущим значением переменной связи crs_ref. Результирующий (out) аргумент (crs_name) представляет собой переменную, значение которой инициализируется с помощью значения, возвращаемого операцией getCourseName, реализованной в классе Course.

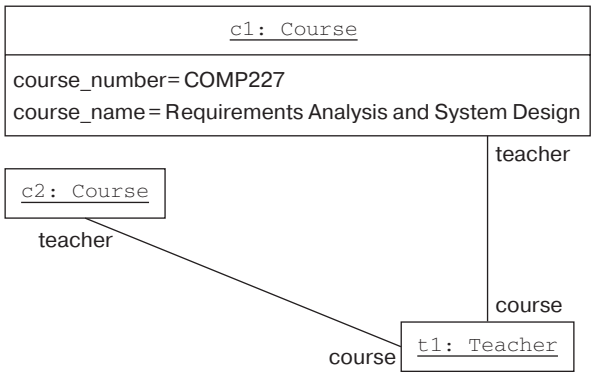

Рис. 2.4. Постоянные связи в объектной UML-модели

2.1.1.3.2. Временная связь

Что если мы не определили постоянные связи между объектами Course и Teacher, но нам по-прежнему требуется отправить сообщение от объекта t1 к объекту c1 для вызова операции getCourseName? Прикладная программа должна обладать другими средствами для установления идентичности объекта c1 и создать *временную связь* от объекта t1 к объекту c1 [70].

К счастью, в распоряжении программиста имеется много средств, способных обеспечить инициализацию переменной crs_name с помощью OID объекта c1, резидентно находящегося в памяти. Для начала заметим, что, возможно, связь между объектами c1 и t1 была установлена в программе раньше, и переменная crs_name хранит правильный OID. Например, программа выполнила операцию поиска с использованием атрибута занятости преподавателя t1 и расписания курсов и определила, что преподаватель t1 должен вести курс c1.

Альтернативная возможность состоит в том, что программа имеет доступ к постоянно хранимой таблице, которая отображает номера курсов на имена преподавателей. Затем программа может выполнить поиск на объектах Course, чтобы отыскать все курсы, читаемые преподавателем t1, и попросить пользователя определить курс (т.е. объект Course), которому должно быть отправлено сообщение getCourseName.

Возможно также, что задача программы как раз и состоит в создании объектов для курсов и преподавателей перед тем, как запомнить их в базе данных. Между соответствующими объектами нет постоянных связей, но пользователь вводит информацию таким образом, что каждый курс четко определяет ответственного за него преподавателя. Затем программа может запомнить временную связь в программной переменной (такой как crs-ref), а эту переменную можно позднее использовать (во время выполнения этой же программы) для отправки сообщений между объектами Teacher и Course.

Короче говоря, существуют как программные, так и управляемые пользователем методы установления временных связей между объектами, которые не связаны постоянно с помощью ассоциации между соответствующими классами. *Временные связи* — это программные переменные, которые содержат значения OID объектов, находящихся в текущий момент в оперативной памяти. Отображение ("втягивание") между временными и постоянными OID должно быть прерогативой базовой программной среды, например, такой как система управления объектной базой данных.

2.1.2. Класс

Класс (*class*) — это дескриптор множества объектов, обладающих одинаковым набором атрибутов и операций. Он служит в качестве *шаблона* (*template*) для создания объектов. Каждый объект, созданный по шаблону, содержит *значения* атрибута, соответствующие *типу* атрибута, определенному в классе, и может вызвать операции, определенные в классе.

Графически класс представляется в виде прямоугольника с тремя отделениями, разделенными горизонтальными линиями, как показано на рис. 2.5. Верхнее отделение хранит имя класса. Среднее отделение содержит объявления всех атрибутов класса. Нижнее отделение содержит определения операций.

2.1.2.1. Атрибуты

Атрибут представляет собой пару *тип-значение*. Класс определяет *типы атрибутов*. Объекты содержат *значения атрибутов*. Рис. 2.6 иллюстрирует два класса с определениями имен и типов принадлежащих им атрибутов.

Тип атрибута может быть встроенным *элементарным типом* (*primitive type*) или другим *классом*. Элементарный тип — это непосредственно распознаваемый и поддерживаемый базовой объектно-ориентированной средой тип данных. Все типы атрибутов, показанных на рис. 2.6, обозначают элементарные типы.

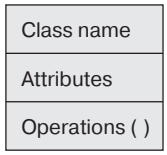

Рис. 2.5. "Отделения" в представлении класса

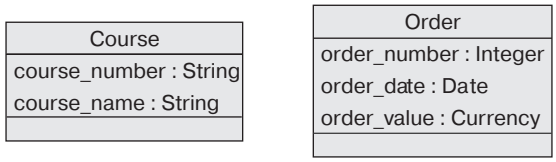

Рис. 2.6. Атрибуты

2.1.2.1.1. Тип атрибута, обозначающий класс

Тип атрибута также может обозначать класс. Применительно к конкретному объекту некоторого класса подобный атрибут содержит значение идентификатора объекта (OID), указывающее на объект другого класса. В UML-моделях анализа атрибуты,

типы которых обозначают классы, не приводятся в среднем "отделении" представления класса (в отличие от примитивных типов). Вместо этого они представляются с помощью *ассоциации* (*association*) между классами. На рис. 2.7 показана подобная ассоциация между двумя классами.

Два имени на ассоциативной линии (`the_shipment` ([конкретная] поставка) и `the_order` ([конкретный] заказ)) представляют так называемые ролевые имена. Ролевое имя (*rolename*) определяет значение для одного из концов ассоциации и используется для перемещения к объекту другого класса в ассоциации.

В реализованной системе ролевое имя (на противоположном конце ассоциации) становится атрибутом класса, тип которого является классом, на который указывает ролевое имя. На рис. 2.8 показано, как выглядят два класса с рис. 2.7 после того, как они были реализованы.

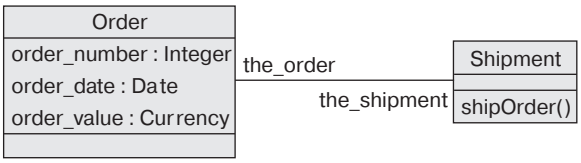

Рис. 2.7. Ролевые имена, обозначающие классы (модель анализа)

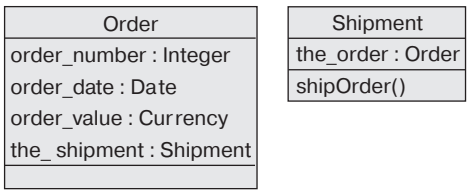

Рис. 2.8. Атрибуты, обозначающие классы (модель реализации)

2.1.2.1.2. Видимость атрибутов

Как отмечалось в разделе 2.1.1.2, объекты взаимодействуют с помощью отправки друг другу сообщений. Сообщение вызывает операцию класса. Операция обслуживает запрос вызывающего объекта с помощью доступа к значениям атрибутов в своем собственном объекте. Чтобы подобный сценарий стал возможен, операции должны быть *видимы* (*visible*) внешним объектам (сообщения должны "увидеть" операции). Говорят, что подобные операции обладают открытой видимостью (*public visibility*).

В чисто объектно-ориентированной системе (примером которой может служить среда программирования языка Smalltalk) большинство операций относится к *открытым* (*public*), а большинство атрибутов — к закрытым (*private*). Значения атрибутов скрыты от других классов. Объекты одного класса могут только затребовать услугу (операцию), опубликованную в открытом интерфейсе другого класса. Им не разрешается непосредственно манипулировать атрибутами другого объекта.

Мы говорим, что операции *инкапсулируют* (*encapsulate*) атрибуты. Заметим, однако, что инкапсуляция применима к классам. Один объект не может скрыть (инкапсулировать) ничего, кроме другого объекта того же класса.

Видимость обычно обозначается с помощью символов плюс и минус:

+ для открытой видимости;
– для закрытой видимости.

В некоторых CASE-средствах эти символы заменены на графические пиктограммы. На рис. 2.9 продемонстрировано два графических представления для обозначения видимости атрибутов.

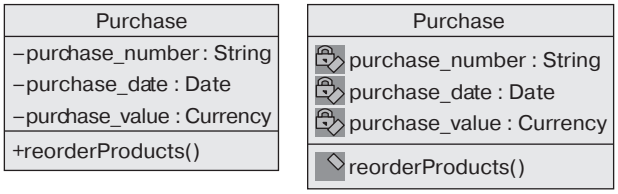

Рис. 2.9. Закрытые атрибуты и открытые операции

2.1.2.2. Операции

Объект содержит данные (атрибуты) и алгоритмы (операции) для работы с этими данными. Операция объявляется в классе. Процедура, реализующая операцию, называется *методом* (*method*).

Операция (или метод, если быть точным) вызывается с помощью отправленного ей сообщения. Имя сообщения и имя операции совпадают. Операция может содержать список параметров, которым в вызове сообщения могут быть присвоены определенные значения, и может возвращать значение вызывающему объекту.

Имя операции вместе со списком типов формальных аргументов называется *сигнатурой* (*signature*) операции. Сигнатура в пределах класса должна быть уникальной. Это значит, что класс может обладать множеством операций с одним и тем же именем, при условии, что списки типов параметров этих операций отличаются.

2.1.2.2.1. Операции — основа совместной работы объектов

Объектно-ориентированная программа выполняется, реагируя на случайные события, которые инициирует пользователь. Источником событий выступает клавиатура, щелчки мышью, пункты меню, кнопки действий и любые другие входные устройства. Инициированное пользователем *событие* преобразуется в *сообщение*, отправляемое *объекту*. Для выполнения задания многим объектам требуется действовать совместно. Объекты "сотрудничают" с помощью вызова *операций* других объектов (см. разд. 2.1.1.2).

На рис. 2.10 показаны операции классов, необходимые для поддержки совместных действий объектов, приведенных на рис. 2.2. Каждое сообщение на рис. 2.2 запрашивает операцию класса, обозначенного как адресат сообщения. В данном простом примере у класса `Order` операции отсутствуют. Объект класса `Order` инициирует совместную работу объектов. Объект `Order` (Заказ) предписывает объекту `Shipment` (Поставка) осуществить его "поставку". В результате поставки (т.е. выполнения заказа за счет товаров со склада) может потребоваться пополнение запаса новыми товарами.

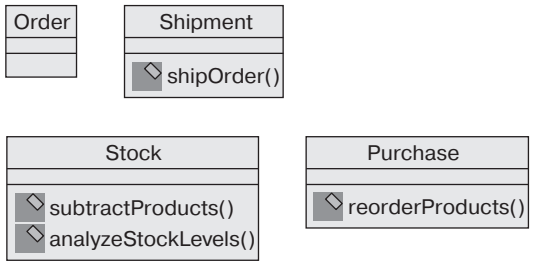

Рис. 2.10. Операции служат основой совместных действий объектов

2.1.2.2.2. Видимость операций

Принципы *видимости операций* не отличаются от принципов видимости атрибутов (см. разд. 2.1.2.1.2). Видимость операции определяет, является ли операция видимой объектам классов, отличных от класса, который определяет операцию. Если она видима, то ее видимость является *открытой*. В противном случае она будет *закрытой*. Пиктограммы перед именами операций на рис. 2.10 обозначают открытую видимость.

Как правило, большинство операций объектно-ориентированной системы обладают открытой видимостью. Чтобы объект мог предоставить сервис внешнему миру, операция "обслуживания" должна быть видима. Однако, большинство объектов имеют также несколько внутренних операций "местного" значения. Видимость этих операций должна быть закрыта. Они доступны только объектам класса, в котором они определены.

Следует различать видимость операции и *область действия* (*scope*) *операции*. Операцию можно вызвать на объекте-экземпляре (см. разд. 2.1.1) или же на объекте-классе (см. разд. 2.1.6). В первом случае говорят, что операция обладает *областью действия экземпляра*. Во втором случае — *областью действия класса*. Например, операция, предназначенная для отыскания возраста работника, обладает областью действия экземпляра, а операция вычисления среднего возраста всех работников обладает областью действия класса.

2.1.3. Ассоциации

Ассоциация (*association*) представляет собой один из видов отношений между классами. Помимо ассоциации существуют такие виды отношений, как обобщение (generalization), агрегация (aggregation), зависимость (dependency) и некоторые другие.

Отношение ассоциации устанавливает связь между объектами данных классов. Объекты, которым требуется взаимодействовать друг с другом, могут использовать установленную связь. Обычно сообщения между объектами отправляются по ассоциативной связи.

На рис. 2.11 показано отношение между классами `Order` и `Shipment` под именем `OrdShip`. Это отношение дает возможность "отправить" (или связать) объект `Order` (Заказ) более, чем одному объекту `Shipment` (Поставка) (обозначенному звездочкой *). Также и объект `Shipment` может осуществить операцию "поставки" (быть связанным с) более, чем одного объекта `Order`.

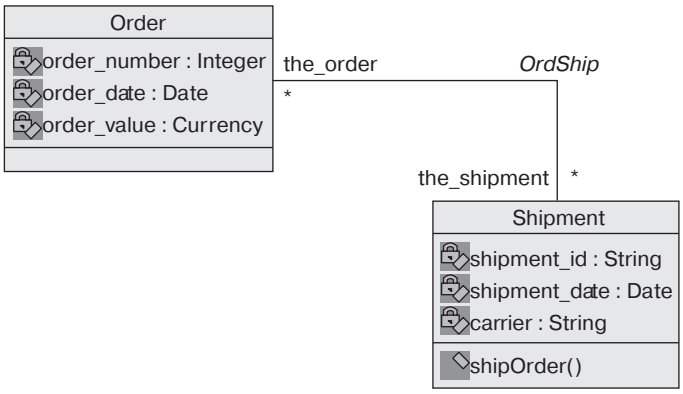

Рис. 2.11. Пример ассоциации

2.1.3.1. Порядок ассоциации

Порядок ассоциации (*association degree*) определяет количество классов, соединенных с помощью ассоциации. Наиболее часто встречаются ассоциации *второго порядка*. Такая ассоциация называется *бинарной ассоциацией*. Ассоциация, показанная на рис. 2.11, является бинарной.

Ее можно также определить на единственном классе. Тогда она называется *унарной* (*unary*) (или *сингулярной* (*singular*)) *ассоциацией* [51]. Унарная ассоциация устанавливает связь между объектами одного класса.

На рис. 2.12 показан типичный пример унарной ассоциации. Этот пример фиксирует иерархическую организационную структуру. Объект `Employee` (Сотрудник) "работает" "под руководством" (`managed_by`) другого объекта `Employee` или в отсутствие руководства с чьей-либо стороны (это может быть, к примеру, генеральный директор, которым никто не руководит). Объект `Employee` "является руководителем" (`manager_of`) для других сотрудников до тех пор, пока не оказывается в самом низу служебной лестницы и никем не "руководит".

Возможны также ассоциации третьего порядка (*тернарные* (*ternary*) *ассоциации*), хотя их применение не рекомендуется [51].

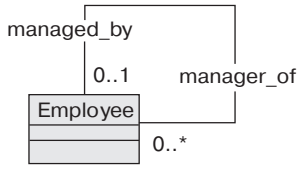

Рис. 2.12. Пример унарной ассоциации

2.1.3.2. Кратность ассоциации

Кратность ассоциации (*association multiplicity*) определяет, сколько объектов могут занимать позицию, указанную *ролевым именем*. Кратность говорит о том, сколько объектов целевого класса (указываемых ролевым именем) может быть ассоциировано с одним объектом исходного класса.

Кратность обозначается в виде диапазона целых чисел `n1..n2`. Число `n1` определяет минимальное количество связываемых объектов, а `n2` — максимальное количество (если мы не знаем точного максимального целочисленного значения, то вместо максимального количества может быть подставлена звездочка *). Если точное минимальное количество заранее неизвестно, но мы знаем, что в отношении может участвовать несколько объектов, этот параметр можно вообще не указывать (подобно тому, как показано на рис.2.11).

Наиболее часто встречаются следующие значения кратности:

```
0..1

0..*

1..1

1..*

*
```

На рис. 2.13 показаны две ассоциации на объектах классов `Teacher` (Преподаватель) и `CourseOffering` (Предлагаемый курс). Одна из ассоциаций фиксирует закрепление преподавателей за текущими учебными курсами. Другая определяет, кто из преподавателей отвечает за учебный курс. Преподаватель может вести несколько учебных курсов или не вести ни одного (например, если преподаватель в отъезде). Учебный курс ведут один или несколько преподавателей. Один из этих преподавателей отвечает за курс. В общем случае преподаватель может отвечать за несколько курсов или не отвечать вовсе. Один и только один преподаватель руководит учебным курсом.

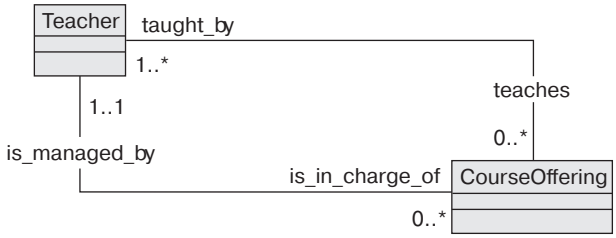

Рис. 2.13. Кратность ассоциации

В языке UML термин “кратность” является очень емким. Минимальная кратность, равная “нулю” или “единице”, может рассматриваться как еще одно семантическое понятие, называемое *принадлежностью* (*membership*) или *участием* (*participation*) [51]. “Нулевая” минимальная кратность означает *необязательную принадлежность* объекта ассоциации. “Единичная” кратность означает *обязательную принадлежность*. Например, предлагаемый курс обучения (объект `CourseOffering`) должен проводиться под руководством преподавателя (объект `Teacher`).

Свойство принадлежности само по себе обладает любопытными семантическими особенностями. Например, данная обязательная принадлежность может дополнительно означать, что принадлежность является *фиксированной*, т.е. если объект связан с целевым объектом в ассоциации, он не может быть повторно связан с другим целевым объектом в той же ассоциации.

2.1.3.3. Ассоциативная связь и объем ассоциации

Ассоциативная *связь* представляет собой экземпляр ассоциации. Это *кортеж* (*tuple*) ссылок на объекты. Кортеж может быть *набором* (*set*) ссылок или *списком* (*list*) (упорядоченным множеством) ссылок. В общем случае кортеж может содержать только одну ссылку. Как рассматривалось выше, связь также представляет *ролевое имя*. Объем (*extent*) ассоциации — это количество связей в наборе.

На рис. 2.14 представлена конкретная реализация ассоциации `OrdShip`, показанной на рис. 2.11. На рис. 2.14 представлено пять связей. Следовательно, объем ассоциации равен пяти.

Понимание сущности ассоциативной связи и объема ассоциации важно для общего представления об ассоциации, однако, ассоциативные связи и объемы ассоциаций не предназначены для моделирования, их нельзя получить во время выполнения программы или каким-либо образом получить к ним явный доступ.

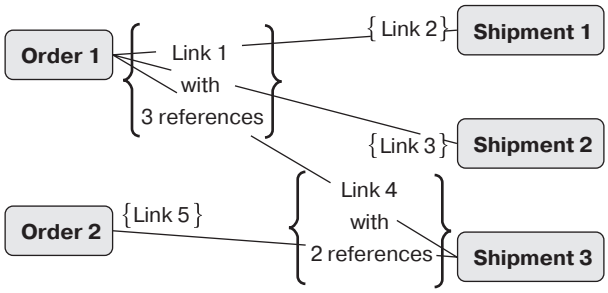

Рис. 2.14. Кратность ассоциации

2.1.3.4. Ассоциативный класс

Иногда ассоциация обладает своими собственными атрибутами (и/или операциями). В качестве модели подобной ассоциации необходимо использовать класс, поскольку атрибуты могут быть определены только в классе. Каждый объект *ассоциативного класса* обладает значениями атрибутов и связями с ассоциированными классами. Поскольку ассоциативный класс является классом, он может быть ассоциирован с другими классами модели обычным способом.

На рис. 2.15 показан ассоциативный класс `Assessment` (Оценка). Объект класса `Assessment` хранит список баллов, общий балл и оценку, полученную студентом (`Student`) по предлагаемому курсу (`CourseOffering`).

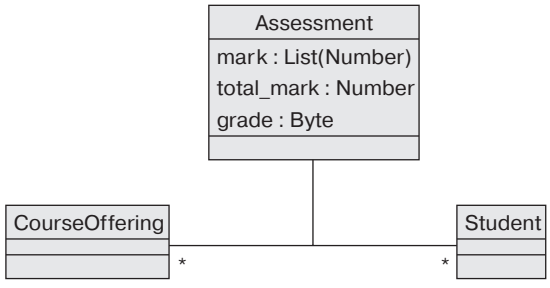

Рис. 2.15. Ассоциативный класс

Атрибут `mark` (балл) имеет тип `List (Number)`. Это так называемый *параметризованный тип* (*parameterized type*). `Number` представляет собой параметр класса `List`, где `List` (Список) определяет упорядоченное множество значений. Атрибут `mark` содержит список всех баллов, полученных студентом по данному курсу. То есть, если студент "Фрэд" проходит курс обучения по дисциплине "COMP227", со временем заполняется список (упорядоченное множество) баллов, полученных им в ходе обучения по этому курсу. Этот список баллов запоминается в объекте `Assessment`, который представляет собой ассоциацию между объектами "Фрэд" и "COMP227".

2.1.4. Агрегация и композиция

Агрегация (*aggregation*) — это отношение вида *часть-целое* между классом, который представляет собрание компонент (*класс-супермножество* (*superset class*)), и классами, представляющими компоненты (*классы-подмножества* (*subset class*)). Класс-супермножество содержит один или более классов-подмножеств. Свойство включения может быть сильным (агрегация по значению (*aggregation by value*)) или слабым (агрегация по ссылке (*aggregation by reference*)). В языке UML агрегация по значению называется композицией (*composition*), а агрегация по ссылке называется просто агрегацией.

С точки зрения системного моделирования агрегация представляет собой особый случай ассоциации, обладающей дополнительной семантикой. В частности, агрегация обладает свойствами транзитивности и асимметрии. *Транзитивность* означает, что если класс A содержит класс B, а класс B содержит класс C, то класс A содержит класс C. *Асимметрия* означает, что если A содержит B, то B не может содержать A.

Композиция обладает дополнительным свойством *зависимость по существованию* (*existence dependency*). Объект класса-подмножества не может существовать в отсутствие связи с объектом класса-супермножества. Отсюда следует, что если объект супермножества удален (уничтожен), объекты его подмножеств также удаляются.

Композиция обозначается заполненным ромбовидным "украшением" на конце ассоциативной связи, присоединенной к классу-супермножеству. *Агрегация*, не являющаяся композицией, помечается незаполненным ромбом. Заметим, однако, что пустой ромб можно также использовать, если в ходе моделирования вопрос о том, является ли агрегация композицией или нет, следует оставить открытым.

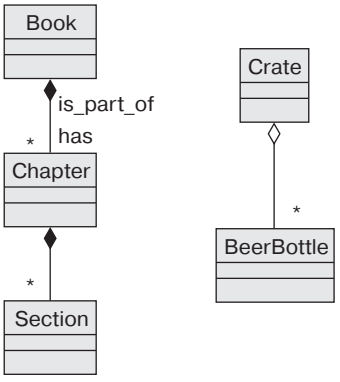

Рис. 2.16. Композиция и агрегация

На рис. 2.16 слева показана композиция, а справа — обычная агрегация. Всякий объект Book (Книга) является композицией объектов Chapter (Глава), а всякая глава (объект Book) — композицией разделов (объектов Section). Объект Chapter не обладает собственной независимой жизнью; он существует только в пределах объекта Book. Этого же нельзя сказать об объектах BeerBottle (Пивная бутылка). Объект может существовать вне контейнера — объекта Crate (Ящик).

2.1.5. Обобщение

Обобщение (*generalization*) представляет собой видовое отношение между более общим классом (*суперкласс* или *родительский класс*) и более специфическим видом класса (*подкласс* или *дочерний класс*). Подкласс является видом суперкласса. Там, где допустимо использование суперкласса, может использоваться и объект подкласса.

Обобщение делает невозможным переопределение уже заданных свойств. Атрибуты и операции, уже определенные для суперкласса, могут повторно использоваться в подклассе. Говорят, что подкласс *наследует* (*inherit*) атрибуты и методы его родительского класса. Обобщение способствует пошаговой спецификации, использованию общих свойств разными классами и лучшей локализации изменений.

Обобщение изображается в виде незаполненного треугольника на конце линии отношения, присоединенной к родительскому классу. На рис. 2.17 Person (Личность) является суперклассом, а Employee (Сотрудник) — подклассом. Класс Employee наследует все атрибуты и операции класса Person. Наследуемые свойства явно не показаны в прямоугольнике, обозначающем подкласс, — отношение обобщения отодвигает наследование на задний план.

Обратите внимание, что наследование применимо к классам, а не объектам. Оно применимо по отношению к типам, а не значениям. Класс Employee наследует определения атрибутов full_name (полное имя) и date_of_birth (дата рождения). Объект класса Employee может быть позже реализован с использованием значений этих атрибутов (поскольку атрибуты существуют в объекте наряду с четырьмя другими атрибутами: date_hired (дата приема на работу), salary (зарплата), leave_entitlement (положенный отпуск) и leave_taken (использованный отпуск)).

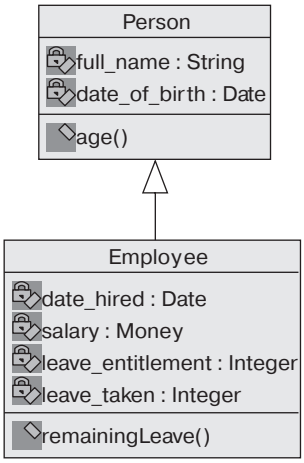

Рис. 2.17. Обобщение

2.1.5.1. Полиморфизм

Зачастую метод, унаследованный подклассом, напрямую используется этим подклассом. Операция `age()` (возраст) работает аналогично в объектах классов `Person` и `Employee`. Однако, иногда необходимо, чтобы операции в подклассе были *замещены* (*overridden*) в соответствии с семантическими вариациями подкласса.

Например, операция `Employee.remainingLeave()` (остаток отпуска сотрудника) вычисляется посредством вычитания значения атрибута `leave_taken` из значения атрибута `leave_entitlement` (рис. 2.17). Однако сотрудник, являющийся менеджером, имеет право на ежегодное получение дополнительного отпуска (`leave_supplement`). Теперь, если добавить в обобщенную иерархию класс `Manager` (как показано на рис. 2.18), операция `Manager.remainingLeave()` должна заместить операцию `Employee.remainingLeave()`. Это показано на рис. 2.18 с помощью дублирования имени операции в подклассе.

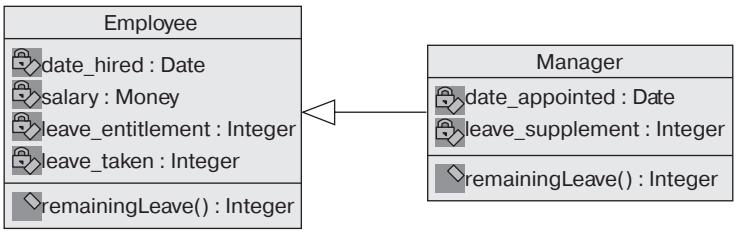

Рис. 2.18. Полиморфизм

Теперь операция `remainingLeave()` *замещена* (модифицирована). Существует две реализации (два *метода*) операции. Можно отправить сообщение `remainingLeave()` объекту `Employee` или объекту `Manager`, и при этом будут выполняться различные методы. Мы можем даже не знать и не заботиться о том, кто выступает в качестве адресата: объект `Employee` или объект `Manager` — будет выполнен подходящий метод. Говорят, что операция `remainingLeave()` *полиморфна* (*polymorphic*).

2.1.5.2. Наследование

В отсутствие наследования полиморфизм находит лишь ограниченное применение. *Наследование* (*inheritance*) открывает возможности для определения подкласса методом последовательного наращивания с использованием и последующим расширением описаний суперкласса. Вполне возможно, операция `Manager.remainingLeave()` реализована с помощью обращения к функциональным возможностям метода `Employee.remainingLeave()` с последующим добавлением к значению, возвращаемому этим методом, величины `leave_supplement`.

Несколько классов в рамках одной иерархии наследования могут объявлять одну и ту же операцию. Подобная операция имеет несколько реализаций (*методов*), но одну и ту же *сигнатуру* — имя, а также количество и типы параметров, если, конечно, они определены для операции. Полиморфное поведение зависит от наследования.

2.1.5.2.1. Множественное наследование

Подкласс может наследовать более, чем одному суперклассу. *Множественное наследование* может стать источником конфликтов наследования, которые должны явно разрешаться программистом.

На рис. 2.19 класс Tutor (Наставник) наследует классам Teacher (Преподаватель) и PostgraduateStudent (Аспирант). Класс Teacher в свою очередь наследует классу Person точно так же, как класс PostgraduateStudent (через класс Student). В результате класс Tutor дважды унаследует атрибуты и операции класса Person, если только программист не укажет программной среде на необходимость однократного наследования за счет использования левого либо правого "пути" наследования (или если программная среда не применит некоторое правило умолчания, приемлемое для программиста, которое ликвидирует двойное наследование).

2.1.5.2.2. Множественная классификация

В большинстве современных объектно-ориентированных программных сред объект может принадлежать только одному классу. Это жесткое ограничение, поскольку в реальности объект может принадлежать одновременно нескольким классам.

Множественная классификация отличается от множественного наследования. При множественной классификации объект одновременно является экземпляром двух или более классов. При множественном наследовании класс может иметь множество суперклассов, но для каждого объекта должен быть определен единственный класс.

В примере множественного наследования на рис. 2.19 каждый объект Person (такой, как Мери или Питер) принадлежит одному классу (наиболее *конкретизированному*, который к нему применим). Если Мери — аспирантка (PostgraduateStudent), но не наставник (Tutor), то Мери принадлежит классу PostgraduateStudent.

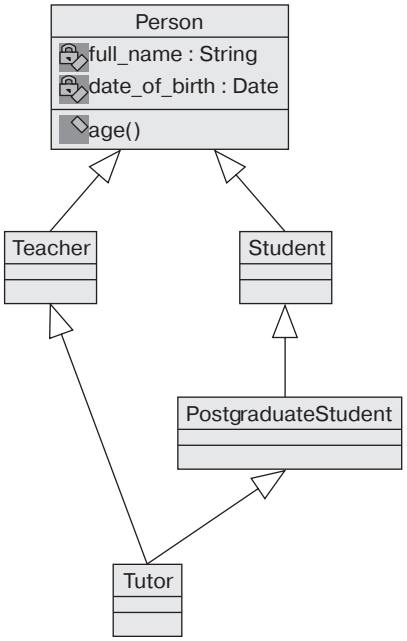

Рис. 2.19. Множественное наследование

Проблема возникает при попытке уточнить значение объекта Person для нескольких ортогональных иерархий. Например, личность (Person) может быть со-

трудником (Employee) или студентом (Student), мужчиной (Male) или женщиной (Female), ребенком (Child) или взрослым (Adult) и т.д. При отсутствии возможности множественной классификации нам необходимо определить классы для каждой разрешенной комбинации иерархий, чтобы получить, к примеру, объект Person, который является ребенком (Child), женщиной (Female) и студентом (Student) (т.е. класс, который можно было бы назвать ChildFemaleStuden) [25].

2.1.5.2.3. Динамическая классификация

В большинстве современных объектно-ориентированных программных сред объект не может сменить класс после того, как он реализован (создан). Это еще одно жесткое ограничение, поскольку в реальности объекты могут динамически изменять класс.

Динамическая классификация является прямым следствием множественной классификации. Объект может не только принадлежать нескольким классам, но также может в процессе своего жизненного цикла приобретать или утрачивать принадлежность к классу.

В случае схемы динамической классификации объект Person в один день может быть просто сотрудником, а в другой — менеджером (и сотрудником). В отсутствие динамической классификации деловые перемены, такие как продвижение сотрудников, трудно (или даже невозможно) *декларативно* моделировать в диаграмме классов. Их необходимо моделировать процедурно в диаграмме состояний или с помощью аналогичного метода моделирования.

К сожалению, язык UML не поддерживает моделирования динамической или множественной классификации. Это ставит его по отсутствию поддержки вровень с программными средами. Следовательно, наши пояснения и примеры не обогащены графическими моделями динамической и множественной классификации.

2.1.5.3. Абстрактный класс

Абстрактный класс (*abstract class*) — важная концепция моделирования, которая логически вытекает из понятия наследования. Абстрактный класс — это родительский класс, который не имеет непосредственных объектов-экземпляров. Только подклассы абстрактного родительского класса могут быть материализованы как экземпляры.

В типичном случае класс становится абстрактным, если, по меньшей мере, одна из его операций является абстрактной. *Абстрактная операция* обладает *сигнатурой* (именем и списком формальных аргументов), определенной в абстрактном родительском классе, однако реализации операции (метод) отличаются для *конкретных* дочерних классов.

Причина, по которой абстрактный класс не может порождать объекты-экземпляры, заключается в том, что он обладает, по меньшей мере, одной абстрактной операцией. Если допустить, что абстрактный класс создает объекты, то сообщение абстрактной операции этого объекта приведет к ошибке при выполнении программы (поскольку для абстрактной операции в классе этого объекта отсутствовала бы реализация).

Класс может быть абстрактным только в том случае, если он является суперклассом, который полностью подразделяется на подклассы. Разделение называется *полным*, если подклассы содержат все возможные объекты, которые порождены в рамках наследственной иерархии. В этом случае не существует никаких “отбившихся” объектов [60]. Класс Person на рис. 2.19 не является абстрактным, нам может потребоваться реализовать объекты класса Person, которые не являются преподавателями (Teacher) и студентами (Student). Возможно также, что нам может потребоваться добавить в будущем к классу Person другие классы (например, такие как AdminEmployee).

На рис. 2.20 показан абстрактный класс `Video` (в языке UML имя абстрактного класса выделяется курсивом). Этот класс содержит абстрактную операцию `rentalCharge` (арендная плата). Ясно, что арендная плата вычисляется по-разному для видеокассет и видеодисков. Поэтому вводятся две различных реализации операции `rentalCharge` — для классов `VideoTape` и `VideoDisk`.

Абстрактные классы могут не иметь объектов, но очень полезны при моделировании. Они создают высокоуровневый "словарь" моделирования, без которого язык моделирования будет неполным.

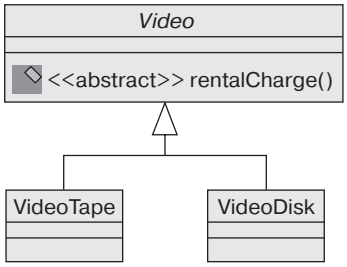

Рис. 2.20. Абстрактный класс с абстрактной операцией

2.1.6. Объект-класс

Попутно мы провели различие между *объектом-экземпляром* и *объектом-классом*. Объект-класс — это объект, область действия атрибутов и/или операций которого совпадает с данным классом. Область действия-класс здесь означает глобальный атрибут или операцию, которые применимы к самому классу и не применимы ни к одному из объектов-экземпляров.

Поскольку в чистой объектно-ориентированной системе в качестве хранилищ выступают объекты, необходимо, чтобы объект-класс хранил глобальные атрибуты и операции. Наиболее распространенными *атрибутами с областью действия-класс* являются атрибуты, которые хранят значения, принятые по умолчанию, или агрегированные значения (такие, как суммы, итоги, средние значения). Наиболее распространенной *операцией с областью действия-класс* является операция создания и уничтожения объектов-экземпляров и операции, вычисляющие агрегированные значения.

На рис. 2.21 показан класс `Student` с атрибутом с областью действия-класс (`max_courses_per_semester`) (максимальное количество курсов за семестр) и аналогичной операцией `averageStudentAge()` (средний возраст студентов). Каждый студент может прослушать одинаковое максимальное количество курсов за семестр, поэтому данное количество должно храниться в объекте-классе. Операция, вычисляющая средний возраст студентов, обладает глобальной областью действия, поскольку ей для того, чтобы определить средний возраст всех студентов, необходим доступ к индивидуальному возрасту каждого студента (в объекте-экземпляре `Student`).

При работе с языком UML в прямоугольнике описания класса рекомендуется выделять атрибуты и операции с областью действия-класс. На рис. 2.21 символ $ перед именем атрибута обозначает атрибут с областью действия-класс (или *статический* атрибут). Для обозначения операции с областью действия-класс (или *статической* операции) в UML в качестве стандартной записи используется глобальное имя, заключенное внутрь скобок (<< >>).

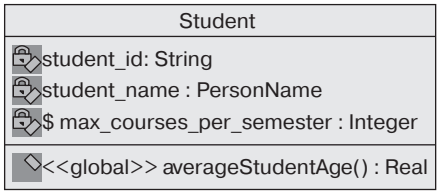

Рис. 2.21. Класс, содержащий атрибуты и операции с областью действия-класс

2.2. Наставление по моделированию анализа

В данном разделе представлено краткое наставление по визуальному моделированию на языке UML с использованием простого примера. Цель состоит в том, чтобы продемонстрировать различные виды диаграмм UML и показать, как они согласуются друг с другом. Каждая из диаграмм UML символизирует определенный взгляд на систему. Чтобы понять систему в целом, необходима разработка и интеграция нескольких видов диаграмм, представляющих разные взгляды.

На самом общем уровне можно выделить три типа UML-моделей— каждая со своим собственным набором диаграмм и связанных с ними конструкций.

1. *Модель состояний* (*state model*), которая представляет *статический взгляд* на систему, — это модель *требований к данным*. Модель состояния представляет структуры данных и отношения на них. Основной метод визуализации модели состояний состоит в использовании диаграммы классов.
2. *Модель поведения* (*behavior model*), которая представляет *операционный взгляд* на систему, — это модель *функциональных требований*. Модель поведения представляет бизнес-транзакции, операции и алгоритмы над данными. Для визуализации модели поведения существует несколько способов — диаграммы прецедентов, диаграммы последовательностей, диаграммы кооперации и диаграммы видов деятельности.
3. *Модель изменения состояний* (*state change model*), которая представляет *динамический взгляд* на систему, — это модель эволюции объектов со временем. Модель изменения состояний представляет возможные изменения состояний объекта (где под *состоянием* понимаются текущие значения атрибутов и ассоциативных связей с другими объектами). Основной метод визуализации модели изменения состояний заключается в использовании диаграммы состояний.

2.2.1. Internet-магазин

Internet вызвал революцию в способах ведения бизнеса. Чтобы оставаться конкурентоспособными организациям, необходимо "присутствовать" в Internet и расширить круг своих бизнес-приложений, включив в них средства электронной коммерции (e-commerce). Изменения касаются прежде всего *клиентской части* (*front end*) приложений, однако, основная *серверная часть* (*back end*) систем управления базами данных (СУБД) по-прежнему должна выполнять вполне традиционную обработку деловых операций.

Наше краткое наставление опирается на приложение *Internet-магазин*. В основном мы будем иметь дело с серверной частью приложения — приемом электронных заказов от клиентов, обработкой и выполнением заказов, выставлением счетов и отправкой товаров потребителю.

Содержание наставления. Internet-магазин
Обработка заказов клиентов

Производитель компьютеров предлагает возможность приобретения своей продукции через Internet. Клиент может выбрать компьютер на Web-странице производителя. Компьютеры подразделяются на серверы, настольные и портативные. Заказчик может выбрать стандартную конфигурацию или построить требуемую конфигурацию в диалоговом режиме. Компоненты конфигурации (такие, как оперативная память) представляются как список для выбора из доступных альтернатив. Для каждой новой конфигурации система может подсчитать цену.

Чтобы оформить заказ, клиент должен заполнить информацию по доставке и оплате. В качестве платежных средств допускается использование кредитных карточек или чеков. После ввода заказа система отправляет клиенту по электронной почте сообщение с подтверждением получения заказа вместе с относящимися к нему деталями. Пока клиент ожидает прибытия компьютера, он может проверить состояние заказа в любое время в диалоговом режиме.

Серверная часть обработки заказа состоит из заданий, необходимых для проверки кредитоспособности и способа расчета клиента за покупку, истребования заказанной конфигурации со склада, печати счета и подачи заявки на склад о доставке компьютера клиенту.

Наставление задумано с целью продемонстрировать методы визуального моделирования языка UML. Представленная последовательность видов деятельности по моделированию также задумана таким образом, чтобы показать зависимости между диаграммами UML.

Последовательность видов деятельности не должна рассматриваться как рекомендуемый *процесс*. В действительности процесс является *итеративным процессом с наращиванием возможностей* (раздел 1.1.3.1). Слишком сильная опора в наставлении на моделирование прецедентов в действительности не отражает сложившейся практики разработки ПО. Обычно наряду с моделированием прецедентов должно проводиться моделирование классов.

2.2.2. Моделирование прецедентов

Поведение системы — это ее реакция в ответ на внешние события. В языке UML внешне наблюдаемое и допускающее тестирование поведение фиксируется в виде прецедентов. *Прецедент* (*use case*) выполняет бизнес-функцию, которую может наблюдать внешний субъект и которая может быть впоследствии отдельно *протестирована* в процессе разработки.

Субъект (*actor*) — это некто или нечто (человек, машина и т.д.), взаимодействующее с прецедентом. Субъект взаимодействует с прецедентом, ожидая получить некий полезный результат.

Диаграмма прецедентов — это наглядное представление субъектов и прецедентов вместе с любыми дополнительными определениями и спецификациями. Диаграмма прецедентов представляет собой не просто некую схему, а является полностью документированной моделью предполагаемого поведения системы. Такое же понимание применимо в отношении других диаграмм языка UML. Если только не оговорено противное, то *UML-диаграмма* используется как синоним *UML-модели*.

2.2.2.1. Субъекты

Субъекты и прецеденты определяются в результате анализа функциональных требований. Функциональные требования воплощаются в прецедентах. Прецеденты

удовлетворяют функциональные требования за счет предоставления субъекту полезного результата. При этом не имеет значения, в какой последовательности решает бизнес-аналитик свои задачи: сначала обозначает субъектов, а затем прецеденты, или наоборот. В нашем наставлении сначала выбираются субъекты.

Типичным графическим изображением субъекта является "штриховой человечек" (см. ниже). В общем случае субъект может быть показан в виде прямоугольного символа класса. Подобно обычному классу субъект может обладать атрибутами и операциями (связанными с событиями, сообщения о которых он отправляет и получает).

На рис. 2.22 показаны три субъекта, которые явно представлены в спецификации. Это субъекты `Customer` (Клиент), `Salesperson` (Продавец) и `Warehouse` (Склад).

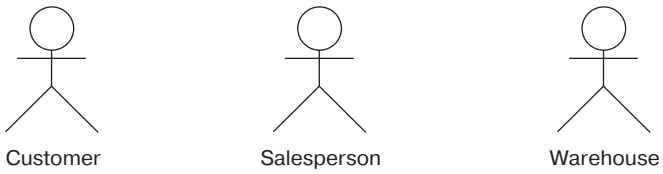

Рис. 2.22. Субъекты (Internet-магазин)

Наставление по анализу: задание 1 (Internet-магазин)

Обратитесь к приведенному выше содержанию наставления (с.54) и рассмотрите следующие расширенные требования для установления субъектов в приложении *Internet-магазин*.

1. Для знакомства со стандартной конфигурацией выбираемого сервера, настольного или портативного компьютера клиент использует Web-страницу Internet-магазина. При этом также приводится цена конфигурации.
2. Клиент выбирает детали конфигурации, с которыми он хочет познакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя.
3. Клиент может выбрать вариант заказа компьютера по Internet либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен.
4. Для размещения заказа клиент должен заполнить электронную форму с адресами для доставки товара и отправки счет-фактуры, а также деталями, касающимися оплаты (кредитная карточка или чек).
5. После ввода заказа клиента в систему продавец отправляет на склад электронное требование, содержащее детали заказанной конфигурации.
6. Детали сделки, включая номер заказа, номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Internet.
7. Склад получает счет-фактуру от продавца и отгружает компьютер клиенту.

2.2.2.2. Прецеденты

Прецедент (*use case*) представляет собой некий целостный набор функций, имеющих определенную ценность для субъекта. Субъект, который не общается с прецедентом, не имеет смысла, однако обратное утверждение не всегда верно (т.е. прецедент, который не общается с субъектом — вещь допустимая). Могут существовать некоторые прецеденты, которые обобщают или уточняют основной прецедент и не взаимодействуют непосредственно с субъектами. Они используются как внутренние в модели прецедентов и помогают основному прецеденту выработать результат, предоставляемый субъекту.

Прецеденты можно вывести в результате идентификации задач для субъекта. Для этого следует задаться вопросом: "Каковы обязанности субъекта по отношению к системе и чего он ожидает от системы?" Прецеденты также можно определить в результате непосредственного анализа функциональных требований. Во многих случаях *функциональное требование* отображается непосредственно в *прецедент*.

Наставление по анализу: задание 2 (Internet-магазин)

Обратитесь к шагу 1 наставления и выберите *прецеденты* для приложения *Internet-магазин*.

Для выполнения этого задания наставления мы можем построить таблицу, которая распределяет функциональные требования по субъектам и прецедентам. Обратите внимание, что некоторые потенциальные бизнес-функции могут выходить за рамки приложения — они не подлежат преобразованию в прецеденты.

В табл. 2.1 функциональные требования, приведенные на шаге 1 наставления, распределены по субъектами и прецедентам. Складские задачи сборки конфигурации компьютера и отправки его клиенту относятся к функциям, которые в данном случае *выходят за рамки* приложения.

На рис. 2.23 показано графическое обозначение прецедентов. Прецедент изображается в виде эллипса, внутри или ниже которого помещается имя прецедента. Можно использовать также другие представления, включая обозначение класса в виде прямоугольника, внутри которого можно привести перечень всех необходимых атрибутов и операций прецедента.

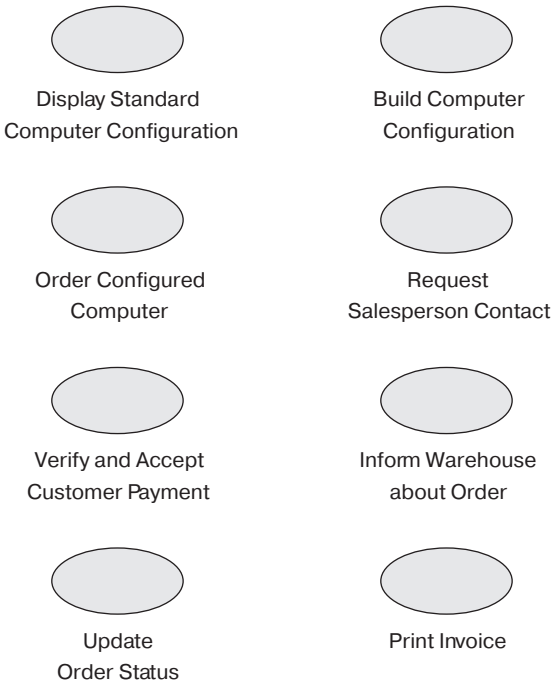

Рис. 2.23. Прецеденты (Internet-магазин)

Таблица 2.1. Распределение требований по субъектам и прецедентам

№ п/п	*Требование*	*Субъект*	*Прецедент*
1.	Для знакомства со стандартной конфигурацией выбираемого сервера, настольного или портативного компьютера клиент использует Web-страницу Internet-магазина. При этом также приводится цена конфигурации	`Customer` (Клиент)	`Display Standard Computer Configuration` (Отображение стандартной конфигурации компьютера)
2.	Клиент выбирает детали конфигурации, с которыми он хочет познакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя	`Customer`	`Build Computer Configuration` (Составление конфигурации компьютера)
3.	Клиент может выбрать вариант заказа компьютера по Internet либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен	`Customer`, `Salesperson` (Продавец)	`Order Configured Computer` (Заказ сконфигурированного компьютера), `Request Salesperson Contact` (Обращение с просьбой к продавцу)
4.	Для размещения заказа клиент должен заполнить электронную форму с адресами для доставки товара и отправки счет-фактуры, а также деталями, касающимися оплаты (кредитная карточка или чек)	`Customer`	`Order Configured Computer`, `Verify and Accept Customer Payment` (Проверка и прием платежа от клиента)
5.	После ввода заказа клиента в систему продавец отправляет на склад электронное требование, содержащее детали, касающиеся заказанной конфигурации	`Salesperson`, `Warehouse` (Склад)	`Inform Warehouse About Order` (Информирование склада о заказе)
6.	Детали сделки, включая номер заказа, номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Internet	`Salesperson`, `Customer`	`Order Configured Computer`, `Update Order States` (Обновление состояния заказа)
7.	Склад получает счет-фактуру от продавца и отправляет компьютер клиенту	`Salesperson`, `Warehouse`	`Print Invoice` (Печать счет-фактуры)

2.2.2.3. Диаграммы прецедентов

Диаграмма прецедентов приписывает прецеденты к субъектам. Она также позволяет пользователю установить отношения между прецедентами, конечно, если такие отношения существуют. Эти отношения рассматриваются в главе 4 (см. разд. 4.3.1.2).

Диаграммы прецедентов — основной метод визуализации для модели поведения системы. Чтобы представить полную *модель прецедентов* (см. разд. 2.2.2.4), необходимо более подробное описание элементов диаграммы (прецедентов и субъектов).

Наставление по анализу: задание 3 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления и изобразите *диаграмму прецедентов* для приложения *Internet-магазин*.

Решение для этого задания наставления можно прямо получить, воспользовавшись информацией предыдущих заданий. Единственное, что необходимо при этом учесть, — это отношения между прецедентами. Диаграмма прецедентов представлена на рис. 2.24. Смысл отношения `<<extend>>` (расширяет) состоит в том, что прецедент `Order Configured Computer` может быть расширен субъектом `Customer` с помощью прецедента `Request Salesperson Contact`.

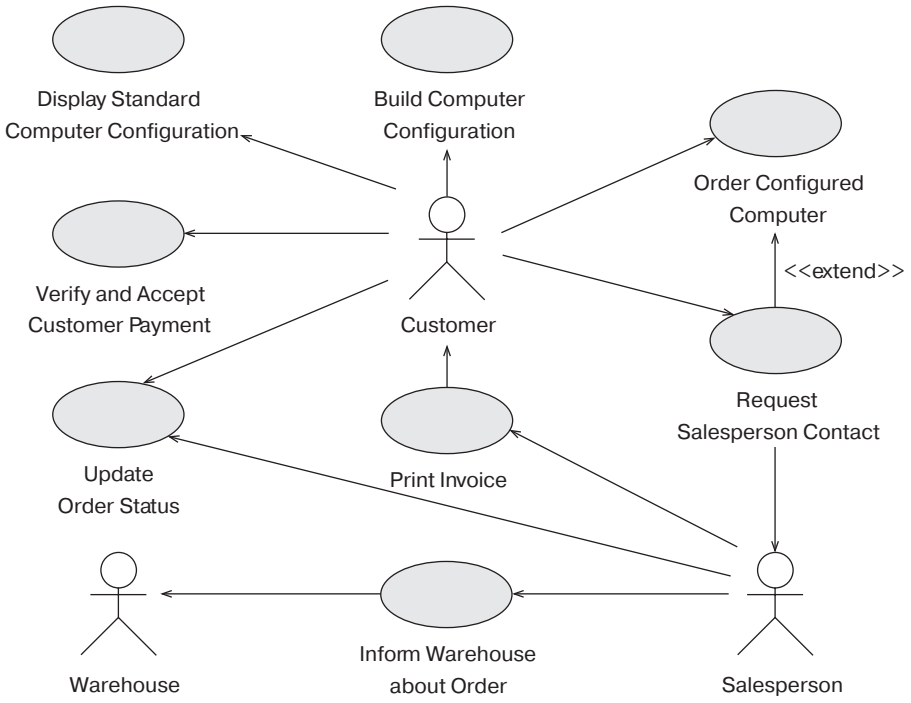

Рис. 2.24. Диаграмма прецедентов (Internet-магазин)

2.2.2.4. Документирование прецедентов

Каждый прецедент должен быть описан с помощью документально зафиксированного *потока событий* (*flow of events*). Соответствующий текстовый документ определяет, что должна делать система, когда субъект инициирует прецедент. Структура *документа, описывающего прецеденты*, может варьироваться, однако типичное описание должно содержать следующие разделы [66].

- *Краткое описание.*
- Участвующие *субъекты.*
- *Предусловия*, необходимые для инициирования прецедента.
- *Детализированное описание* потока событий, которое включает:
  - *основной поток*, который можно разбить для того, чтобы показать *подчиненные потоки* событий (подчиненные потоки могут быть разделены дальше на еще более мелкие потоки, с целью улучшить удобочитаемость документа);
  - *альтернативные потоки* для определения исключительных ситуаций.
- *Постусловия*, определяющие состояние системы, по достижении которого прецедент завершается.

Документ, содержащий описания прецедента, развивается по ходу разработки. На ранней стадии определения требований составляется только краткое описание. Остальные части документа создаются постепенно и итеративно. Полный документ возникает в конце этапа спецификации требований. На этой стадии документ может быть дополнен прототипами GUI-экранов. Позднее документ по прецедентам используется для создания пользовательской документации для реализуемой системы.

Решение для этого задания наставления представлено в табличной форме (табл. 2.2). Не следует рассматривать этот способ документирования прецедента как общепринятый. Документы с описанием прецедентов могут быть многостраничными (в среднем около десятка страниц) и обладать стандартной для документов структурой, и включать, например, такие элементы, как оглавление.

Наставление по анализу: задание 4 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления и подготовьте *документ, содержащий описание прецедента* `Order Configured Computer`. Чтобы установить детали, которые явно не сформулированы в требованиях, используйте ваши общие знания или информацию о типичной задаче обработки заказов.

Таблица 2.2. Описательная спецификация прецедента `Order Configured Computer` (Internet-магазин)

Прецедент	*Заказ сконфигурированного компьютера*
Краткое описание	Прецедент дает возможность клиенту (`Client`) ввести заказ на покупку. Заказ включает адреса доставки товара и оплаты счета, а также детали условий оплаты
Субъекты	`Client` (Клиент)
Предусловия	Клиент с помощью Internet-броузера выбирает страницу производителя компьютеров для ввода заказа. На Web-странице отображается подробная информация о сконфигурированном компьютере вместе с его ценой

Окончание табл. 2.2

Прецедент	*Заказ сконфигурированного компьютера*
Основной поток	Начало прецедента совпадает с решением клиента заказать сконфигурированный компьютер с помощью выбора функции `Continue` (Продолжить) (или аналогичной функции) при отображении на экране детализированной информации, относящейся к заказу.
	Система просит клиента ввести детали покупки, в том числе: имя продавца (если оно известно); детали, касающиеся доставки (имя и адрес клиента); детальную информацию по оплате (если она отличается от информации по доставке); способ оплаты (кредитная карточка или чек) и произвольные комментарии.
	Клиент выбирает функцию `Purchase` (Покупка) (или аналогичную функцию) для отправки заказа производителю.
	Система присваивает уникальный номер заказа и клиентский учетный номер заказу на покупку и запоминает информацию о заказе в базе данных.
	Система отправляет клиенту по электронной почте номер заказа и клиентский номер клиенту вместе со всеми деталями, относящимися к заказу, в качестве подтверждения принятия заказа
Альтернативные потоки	Клиент инициирует функцию `Purchase` до того, как введет всю обязательную информацию. Система отображает на экране сообщение об ошибке и просит ввести пропущенную информацию.
	Клиент выбирает функцию `Reset` (Сброс) (или аналогичную) для того, чтобы вернуться к исходной форме заказа на покупку. Система дает возможность клиенту вновь ввести информацию
Постусловия	Если прецедент был успешным, заказ на покупку записывается в базу данных. В противном случае состояние системы остается неизменным

2.2.3. Моделирование видов деятельности

Модель видов деятельности (*activity model*) может представлять в графической форме поток событий для прецедента. Этот тип модели был введен только в более поздние версии UML и позволил преодолеть разрыв между высокоуровневым представлением поведения системы с помощью *моделей прецедентов* и намного более низким уровнем представления поведения с помощью *моделей взаимодействий* (диаграмм последовательностей и диаграмм кооперации).

Диаграмма видов деятельности показывает шаги вычисления. Каждый шаг соответствует *состоянию* (*state*), в котором что-либо выполняется. Поэтому шаги выполнения называются *состояниями вида деятельности*. Диаграмма описывает, какие шаги выполняются последовательно, а какие — параллельно. Передача управления от одного состояния вида деятельности к другому называется *переходом* (*transition*).

После завершения документа с описанием прецедента состояния вида деятельности можно установить по описанию *основного* и *альтернативных потоков*. Однако, между описанием прецедентов и моделью видов деятельности существует важное различие. Описание прецедента создается *с точки зрения внешнего субъекта*. Модель видов деятельности отражает *внутрисистемную точку зрения*.

Модели видов деятельности могут находить и другое применение при разработке систем помимо моделирования прецедентов [25]. Они могут использоваться для анализа бизнес-процессов на высоком уровне абстракции до выработки прецедентов. И наоборот, их можно использовать на более низком уровне абстракции для разработки сложных последовательных алгоритмов или средств распараллеливания в многопоточных приложениях.

2.2.3.1. Виды деятельности

Если моделирование видов деятельности используется для визуализации последовательности видов деятельности, связанных с прецедентом, то состояния вида деятельности можно установить на основе документа описания прецедента. Как было отмечено выше, имена действиям следует присваивать, исходя из системных соображений, а не с точки зрения субъекта.

Состояние вида деятельности представляется в UML в виде прямоугольника с закругленными углами. Следует сразу уточнить, что один и тот же графический символ используется для визуализации *состояния вида деятельности* (*activity state*) и *состояния действия* (*action state*). Различие между деятельностью и действием реакцией заключается в их временном масштабе. Для осуществления *деятельности* требуется определенное время; *действие* же завершается столь быстро, что — в масштабах нашей временной шкалы — может считаться происходящим мгновенно. (Следовательно, в модели состояний (раздел 2.2.6) виды деятельности могут быть определены только в рамках *состояния объекта*, а *действия* могут появляться также при переходе между состояниями объекта.)

В табл. 2.3 изложены события, относящиеся к основному и альтернативным потокам, заимствованным из документа описания прецедента, а также обозначены состояния видов деятельности. Обратите внимание, что имя каждому виду деятельности присвоено, исходя из системной точки зрения, а не с точки зрения субъекта.

Виды деятельности, приведенные в табл. 2.3, показаны на рис. 2.25.

Таблица 2.3. Установление действий в основном и альтернативных потоках

№ п/п	*Формулировка прецедента*	*Состояние вида деятельности*
1.	Начало прецедента совпадает с решением клиента заказать сконфигурированный компьютер с помощью выбора функции `Continue` (или аналогичной функции) при отображении на экране детализированной информации, относящейся к заказу	`Display Current Configuration` (Отображение текущей конфигурации); `Get Order Request` (Получение запроса на заказ)
2.	Система просит клиента ввести детализированную информацию о покупке, в том числе: имя продавца (если оно известно); детали, касающиеся доставки (имя и адрес клиента); детальную информацию по оплате (если она отличается от информации по доставке); способ оплаты (кредитная карточка или чек) и произвольные комментарии	`Display Purchase Form` (Отображение закупочной формы)

Окончание табл. 2.3

№ п/п	*Формулировка прецедента*	*Состояние вида деятельности*
3.	Клиент выбирает функцию `Purchase` (или аналогичную функцию) для отправки заказа производителю	`Get Purchase Details` (Детализировать информацию о покупке)
4.	Система присваивает уникальный номер заказа и клиентский учетный номер заказу на покупку и запоминает информацию о заказе в базе данных	`Store Order` (Запомнить заказ)
5.	Система отправляет клиенту по электронной почте номер заказа и клиентский номер клиенту вместе со всеми деталями, относящимися к заказу, в качестве подтверждения принятия заказа	`Email Order Details` (Отправить детальную информацию по заказу)
6.	Клиент инициирует функцию `Purchase` до того, как введет всю обязательную информацию. Система отображает на экране сообщение об ошибке и просит ввести пропущенную информацию	`Get Purchase Details`; `Display Purchase Form`
7.	Клиент выбирает функцию `Reset` (или аналогичную) для того, чтобы вернуться к исходной форме заказа на покупку. Система дает возможность клиенту вновь ввести информацию	`Display Purchase Form`

Наставление по анализу: задание 5 (Internet-магазин)

Обратитесь к заданию 4 наставления. Проанализируйте основной и альтернативный потоки, представленные в документе описания прецедента. Установите, какие виды деятельности свойственны прецеденту `Order Configured Computer`.

2.2.3.2. Диаграмма видов деятельности

Диаграмма видов деятельности (*activity diagram*) показывает переходы между видами деятельности. Если вид деятельности не представляет собой замкнутого цикла, то диаграмма содержит начальное состояние вида деятельности и одно или более конечных состояний вида деятельности. Полностью закрашенная окружность представляет начальное состояние. Конечное состояние изображается в виде окружности с закрашенной центральной частью (автор образно называет этот символ “бычьим глазом”).

Переходы могут *разветвляться по условию* и *объединяться*. В результате возникают альтернативные (*alternative*) вычислительные потоки (*thread*). Условие ветвления обозначается ромбом.

Переходы могут также *разделяться* и *сливаться*. В результате возникают *параллельные* (одновременно выполняемые) (*concurrent*) потоки. Распараллеливание и воссоединение переходов представляется в виде жирной линии или полосы. Заметим, что диаграмма вида деятельности, в которой отсутствуют параллельные процессы, похожа на обычную блок-схему. (Поведение с параллелизмом в наставлении не используется. Его демонстрации посвящен подраздел 4.3.2).

Рис. 2.25. Виды деятельности для прецедента Order Configured Computer (Internet-магазин)

Наставление по анализу: задание 6 (Internet-магазин)

Обратитесь к заданию 4 и заданию 5 наставления и изобразите диаграмму видов деятельности для прецедента `Order Configured Computer`, относящегося к приложению *Internet-магазин*.

Чтобы изобразить диаграмму, виды деятельности, обозначенные в задании 5 наставления, следует соединить линиями перехода, как показано на рис. 2.26.

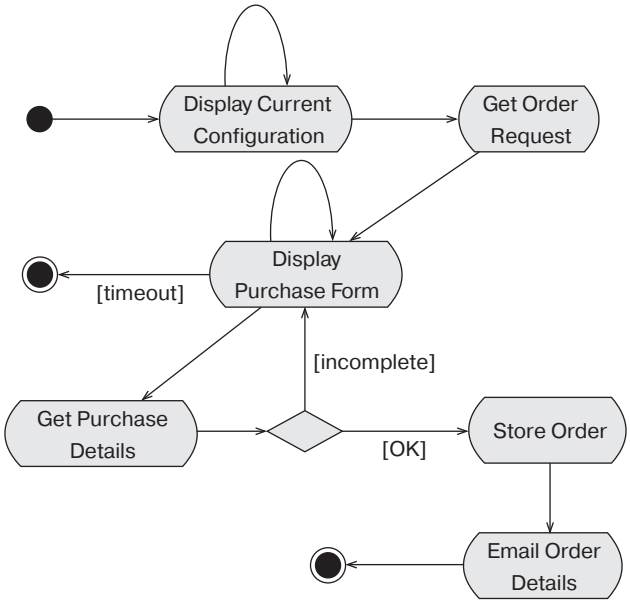

Рис. 2.26. Диаграмма видов деятельности для прецедента Order Configured Computer (Internet-магазин)

Начальное состояние деятельности — `Display Current Configuration`. *Рекурсивный переход* на этом состоянии служит признанием того факта, что отображение непрерывно обновляется до тех пор, пока не сработает следующий переход (переход в состояние `Get Order Request`). Этот факт может интерпретироваться как осознание того, что это состояние является *деятельностью*, а не *действием*.

Если при нахождении модели в состоянии `Display Purchase Form` сработает условие `timeout` (истечение времени ожидания), то выполнение модели видов деятель-

ности завершится. Иначе, активизируется состояние `Get Purchase Details`. Если детальные данные относительно покупки неполны, система вновь переходит в состояние `Display Purchase Form`. В противном случае система переходит в состояние `Store Order`, а затем — в состояние `Email Order Details` (конечное состояние).

Обратите внимание, что на диаграмме показаны только те условные переходы, которые (всегда) появляются на выходах из состояния вида деятельности. Условные переходы, которые являются внутренними для состояния вида деятельности, не показаны явно на диаграмме. Об их существовании можно догадаться по наличию нескольких исходящих переходов, которые, возможно, сопровождаются именем условия в квадратных скобках (например, таким как `[timeout]` на выходе из состояния `Display Purchase Form`).

2.2.4. Моделирование классов

Систему образует *системное состояние*. Состояние является функцией содержимого системной информации в заданный момент времени — это функция системного текущего набора объектов-экземпляров (см. разд. 2.1.1).

Определение внутреннего состояния системы дается в *модели классов* (*class model*). К элементам, принимающим участие в моделировании классов, относятся сами классы, атрибуты и операции классов, ассоциации, агрегации и композиции, а также обобщения (см. разделы 2.1.2, 2.1.3, 2.1.4 и 2.1.5). *Диаграмма классов* (*class diagram*) дает обобщенное визуальное представление обо всех этих элементах модели.

Еще раз напоминаем, что хотя в наставлении моделирование классов рассматривается после моделирования прецедентов, на практике эти два вида деятельности проводятся в параллель (см. разд. 3.2). Две модели взаимно дополняют одна другую вспомогательной информацией. Прецеденты облегчают выбор классов, и наоборот — модели классов могут помочь выявить упущенные прецеденты.

2.2.4.1. Классы

До сих пор мы использовали классы для определения *бизнес-объектов*. Все приведенные в книге примеры классов относились к долгоживущим (постоянным или перманентным) сущностям бизнес-процессов, таким как `Order` (Заказ), `Shipment` (Доставка), `Customer` (Клиент), `Student` (Студент) и т.д. Это классы, которые определяют модель базы данных для прикладной области. По этой причине подобные классы часто называют *классами-сущностями* (*entity class*) или модельными классами. Они представляют постоянно хранимые объекты базы данных.

Классы-сущности определяют существо любой информационной системы. Анализ требований направлен преимущественно на выявление классов-сущностей. Однако, для функционирования системы требуются также классы другого типа. Пользователям системы необходимы классы, которые определяют GUI-объекты (например, такие как экранные формы), называемые *пограничными классами* (*boundary classes*) (классами представления (view classes)). Чтобы функционировать надлежащим образом, системе также необходимы классы, которые управляют программной логикой — *управляющие классы* (*control classes*) (см. разд. 5.2.3).

В зависимости от конкретного подхода к моделированию пограничные и управляющие классы могут рассматриваться или не рассматриваться на некотором уровне представления в ходе анализа требований. Моделирование классов этого типа может быть отложено до этапа проектирования системы.

Наставление по анализу: задание 7 (Internet-магазин)

Обратитесь к требованиям, определенным в содержании наставления (разд. 2.2.1) и в задании 1 наставления (разд. 2.2.2.1). Укажите классы, которые могут служить прототипами *классов-сущностей* для приложения *Internet-магазин*.

Следуя подходу, принятому при установлении субъектов и прецедентов (см. табл. 2.1), можно построить таблицу, которая поможет выявить классы в результате анализа функциональных требований. В табл. 2.4 функциональным требованиям, перечисленным в задании 1 наставления, поставлены в соответствие классы-сущности.

Таблица 2.4. Соответствие функциональных требований и классов-сущностей (Internet-магазин)

№ п/п	*Требование*	*Класс-сущность*
1.	Для знакомства со стандартной конфигурацией выбираемого сервера, настольного или портативного компьютера клиент использует Web-страницу Internet-магазина. При этом также приводится цена конфигурации	`Customer` (Клиент); `Computer` (Компьютер); `StandardConfiguration` (Стандартная конфигурация); `Product` (Товар)
2.	Клиент выбирает детали конфигурации, с которыми хочет познакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя	`Customer`, `ConfiguredComputer` (Сконфигурированный компьютер); `ConfiguredProduct` (Укомплектованный товар); `ConfigurationItem` (Элемент конфигурации)
3.	Клиент может выбрать вариант заказа компьютера по Internet либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен	`Customer`, `ConfiguredComputer`, `Order` (Заказ), `Salesperson` (Продавец)
4.	Для размещения заказа клиент должен заполнить электронную форму с адресами для доставки товара и отправки счет-фактуры, а также деталями, касающимися оплаты (кредитная карточка или чек)	`Customer`; `Order`; `Shipment` (Поставка); `Invoice` (Счет-фактура); `Payment` (Платеж)
5.	После ввода заказа клиента в систему продавец отправляет на склад электронное требование, содержащее детали, касающиеся заказанной конфигурации	`Customer`; `Order`; `Salesperson`; `ConfiguredComputer`; `ConfigurationItem`

Окончание табл. 2.4

№ п/п	*Требование*	*Класс-сущность*
6.	Детали сделки, включая номер заказа, номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Internet	`Order`; `Customer`; `OrderStatus` (Состояние заказа)
7.	Склад получает счет-фактуру от продавца и отправляет компьютер клиенту	`Invoice`; `Shipment`

Выделение классов представляет собой итеративную задачу, и первоначальный перечень предполагаемых классов, как правило, претерпевает изменения. При определении того, являются ли понятия, присутствующие в требованиях, искомыми классами, могут помочь ответы на некоторые вопросы. Вот эти вопросы.

1. Является ли понятие "вместилищем" данных?
2. Обладает ли оно отдельными атрибутами, способными принимать разные значения?
3. Можно ли создать для него множество объектов-экземпляров?
4. Входит ли оно в границы прикладной области?

Перечень классов, приведенный в табл. 2.4, кроме того, вызывает много вопросов. К примеру, следует задаться такими вопросами.

1. В чем различие между классами `ConfiguredComputer` и `Order`? Помимо прочего, мы не собираемся хранить информацию о сконфигурированном компьютере (`ConfiguredComputer`), до тех пор пока заказ (объект `Order`) не размещен, или это не так?
2. Совпадает ли смысл понятия `Shipment`, приведенный в требованиях №4 и №7? Скорее всего, нет. Необходим ли нам класс `Shipment`, если нам известно, что поставка является обязанностью склада и, таким образом, выходит за рамки приложения? (Разд. 2.2.2.2).
3. Не является ли понятие элемента конфигурации (`ConfigurationItem`) просто атрибутом понятия сконфигурированного компьютера (`ConfiguredComputer`)?
4. Является понятие состояние заказа (`OrderStatus`) самостоятельным классом или же атрибутом понятия заказа (`Order`)?
5. Является ли понятие продавца (`Salesperson`) самостоятельным классом или же атрибутом понятий (`Order`) и (`Invoice`)?

Дать ответы на эти и аналогичные вопросы не легко, для этого требуется глубокое знание требований прикладной области. В целях дальнейшей работы с данным наставлением мы выбрали классы, перечень которых приведен на рис. 2.27. Обратите внимание, что класс `Customer` (Клиент) уже появился в качестве *субъекта* на диаграмме прецедента – отсюда примечание "с точки зрения прецедента".

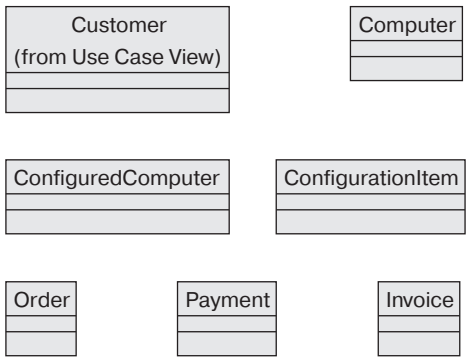

Рис. 2.27. Классы (Internet-магазин)

2.2.4.2. Атрибуты

Структура класса определяется его атрибутами (разд. 2.1.2.1). При первоначальном объявлении класса аналитик должен иметь некоторое представление о структуре атрибутов. На практике основные атрибуты обычно назначаются классу сразу после его добавления к модели.

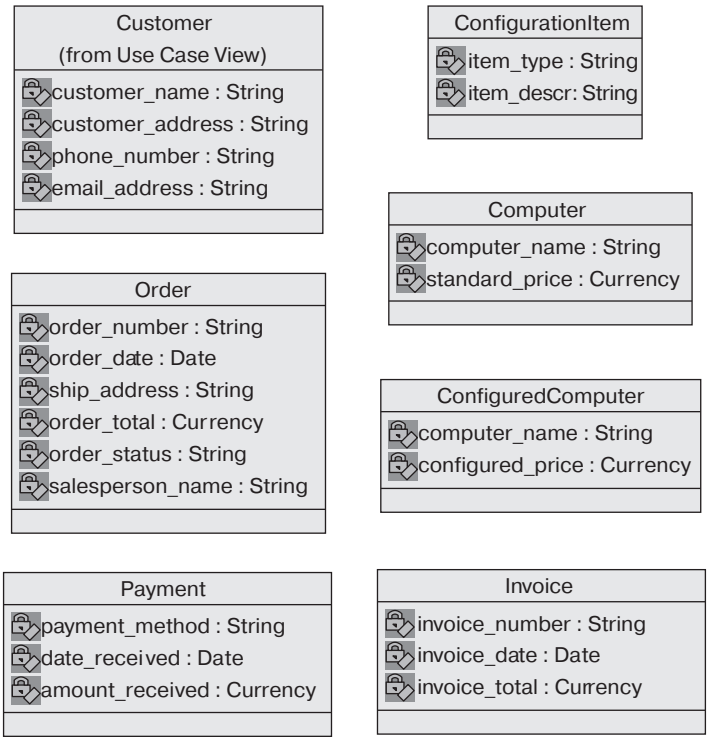

Рис. 2.28. Элементарные атрибуты (Internet-магазин)

Наставление по анализу: задание 8 (Internet-магазин)

Обратитесь к заданиям 5, 6 и 7 наставления. Подумайте над атрибутами для классов, показанных на рис. 2.27. Рассмотрите только атрибуты, *тип которых принадлежит к элементарным* (разд. 2.1.2.1).

На рис. 2.28 показаны классы с элементарными атрибутами. При этом приведены только наиболее интересные атрибуты. Атрибуты класса `ConfigurationItem` требуют некоторых пояснений. Значениями атрибута `item_type` являются типы элементов конфигурации, такие как процессор, память, монитор, жесткий диск и т.д. Атрибут `item_descr` содержит более подробное описание типа элемента. Например, процессор, входящий в конфигурацию, может представлять собой процессор марки Intel с частотой 600 МГц, имеющий кэш объемом 256 Кбайт.

Совершенно очевидно, что при определении атрибутов, приведенных на рис. 2.28, существуют широкие возможности для выбора произвольных вариантов. Если читатель попытается предложить свое собственное решение для этого задания наставления, то другие, отличные от предложенных в книге, интерпретации возможны и конечно допустимы.

2.2.4.3. Ассоциации

Ассоциации, связывающие классы, устанавливают пути, облегчающие кооперацию объектов (разд. 2.1.3). В реализуемой системе ассоциации представляются типами атрибутов, которые обозначают ассоциированные классы (разд. 2.1.2.1.1). В модели анализа ассоциации представлены соединительными линиями.

Наставление по анализу: задание 9 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления. Рассмотрите классы, показанные на рис. 2.28. Подумайте, какие пути доступа между этими классами необходимы для прецедентов. Добавьте к модели классов *ассоциации*.

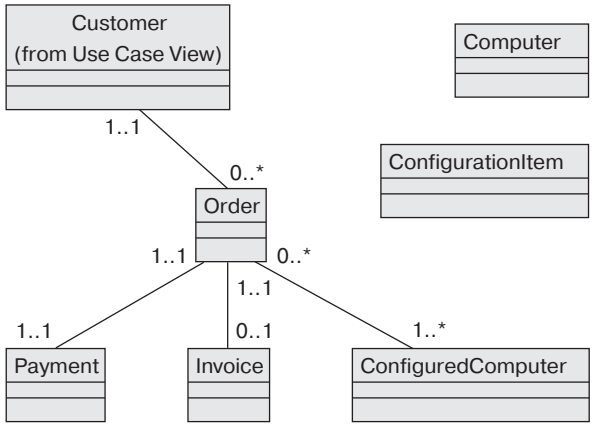

Рис. 2.29. Ассоциации (Internet-магазин)

На рис. 2.29 показаны наиболее очевидные ассоциации между классами. При определении *кратности* ассоциаций был сделан ряд предположений (разд. 2.1.3.2). Заказ (`Order`) поступает от одного клиента (`Customer`), однако клиент может разместить несколько заказов. Заказ не принимается до тех пор, пока не определены реквизиты платежа (`Payment`) (отсюда, ассоциация типа "один к одному"). Заказ не должен обладать связанной с ним счетом-фактурой (`Invoice`), однако счет-фактура всегда связана с единственным заказом. Заказ делается на один или несколько сконфигурированных компьютеров (`ConfiguredComputer`). А компьютер с данной конфигурацией может заказываться многократно или не заказываться вовсе.

2.2.4.4. Агрегации

Агрегация и *композиция* являются более сильной формой ассоциативной связи, которой присуща семантика принадлежности (разд. 2.1.4). В типичной коммерческой программной среде агрегация и композиция, скорее всего, должны быть реализованы подобно ассоциации — с помощью типов атрибутов, которые обозначают ассоциированные классы (разд. 2.1.2.1.1).

Наставление по анализу: задание 10 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления. Рассмотрите модели, представленные на рис. 2.28 и 2.29. Добавьте к модели классов *агрегации*.

На рис. 2.30 к модели добавлены отношения агрегации. Компьютер (`Computer`) *обладает* (как вы помните, выше говорилось о свойстве *принадлежности*!) одним или более элементами конфигурации (`ConfigurationItems`). Подобно этому, сконфигурированный компьютер (`ConfiguredComputer`) *состоит из* одного или нескольких элементов (`ConfigurationItems`).

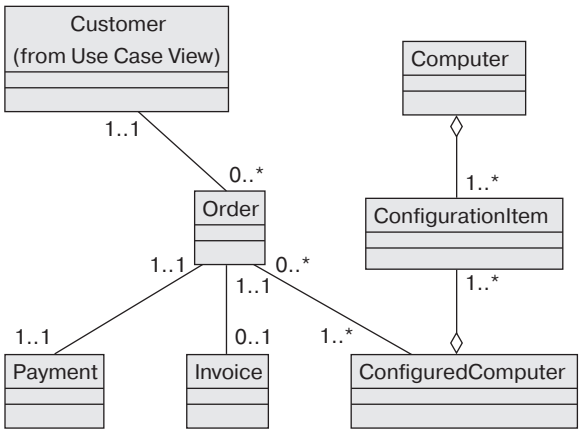

Рис. 2.30. Агрегации (Internet-магазин)

2.2.4.5. Обобщения

Обобщение представляет собой мощное средство повторного использования ПО, однако, оно также способно в значительной мере упростить и прояснить модель

(разд. 2.1.5). Упрощение модели семантически не находит выражения в ее графическом изображении. Обобщение обычно используют как метод создания дополнительных родовых классов (как правило, *абстрактных*). Упрощение достигается за счет увеличения точности, с которой существующий класс может быть ассоциирован с наиболее подходящими (т.е. на наиболее удобном уровне абстракции) классами в иерархии обобщения.

На рис. 3.31 показана модифицированная модель, в которой класс изменен на абстрактный класс `Computer`, являющийся родовым для двух конкретных подклассов — `Standard Computer` и `ConfiguredComputer`. Теперь классы `Order` и `ConfigurationItem` связаны с классом `Computer`, который может быть либо классом `StandardComputer`, либо классом `ConfiguredComputer`.

Наставление по анализу: задание 11 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления. Рассмотрите модели на рис. 2.28 и 2.30. Подумайте, каким образом можно вычленить какие-либо общие атрибуты в существующих классах и перенести их в класс более высокого уровня. Введите в модель классов *обобщение*.

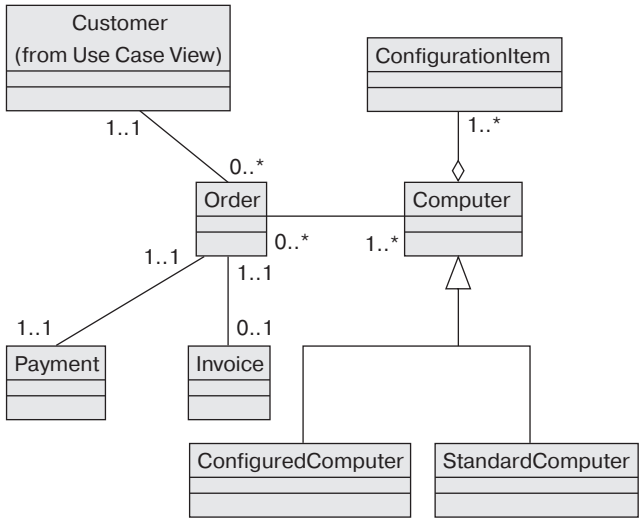

Рис. 2.31. Обобщение (Internet-магазин)

2.2.4.6. Диаграмма классов

Диаграмма классов составляет, так сказать, "сердце" и "душу" объектно-ориентированной системы. В данном наставлении ставится цель только продемонстрировать возможности *статического моделирования* применительно к модели классов.

В классы пока что не введено ни одной новой операции. Операции принадлежат скорее к сфере проектирования, чем анализа. После того, как операции в конечном итоге вводятся в классы, модель классов в явном виде определяет *поведение* системы.

Наставление по анализу: задание 12 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления. Скомбинируйте модели, представленные на рис. 2.28 и 2.30, таким образом, чтобы показать полную диаграмму классов. Измените содержимое атрибутов классов, как того требует введение обобщающей иерархии.

На рис. 2.32 показана диаграмма классов для приложения *Internet-магазин*. Это еще не законченное решение, поскольку, например, для практического решения может потребоваться введение дополнительных атрибутов.

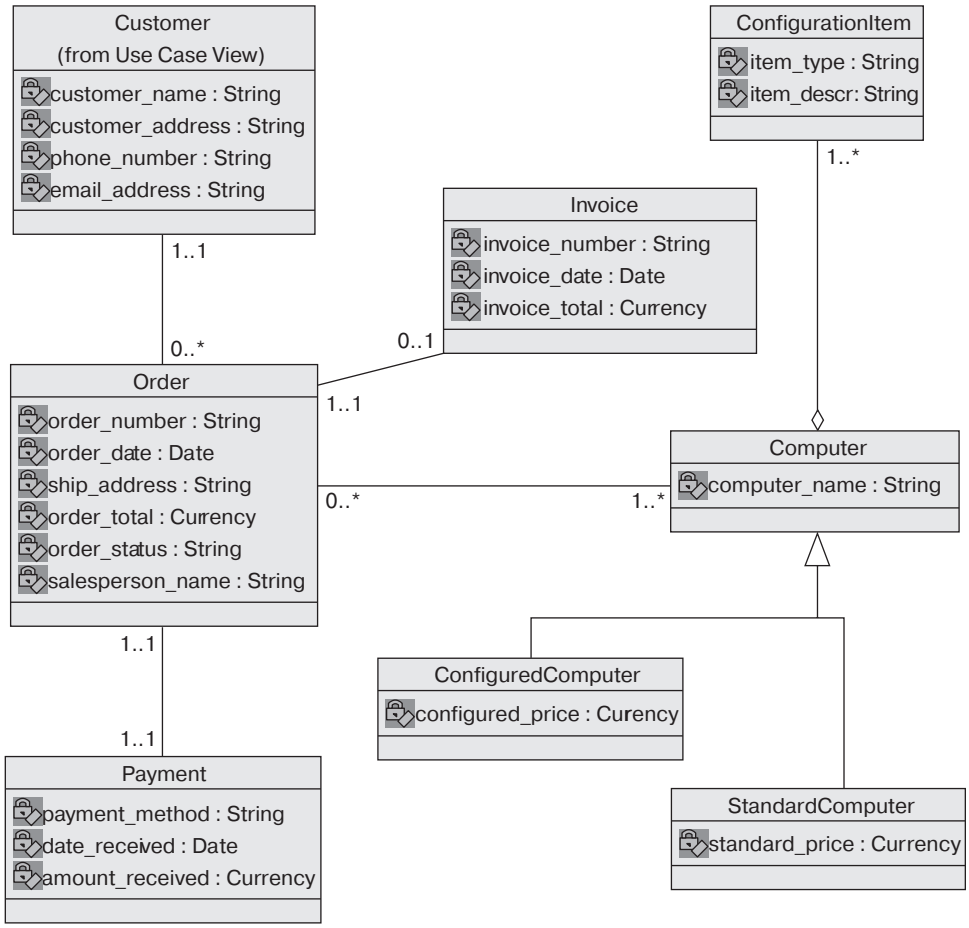

Рис. 2.32. Диаграмма классов (Internet-магазин)

2.2.5. Моделирование взаимодействий

Моделирование взаимодействий (*interaction modeling*) охватывает вопросы взаимодействия между объектами, необходимыми для выполнения прецедента. Модели взаимодей-

ствия используются на более развитых стадиях анализа требований, когда становится известной модель классов, так что ссылки на объекты опираются на модель классов.

Приведенное выше наблюдение служит опорой для установления основного различия между моделированием видов деятельности (см. разд. 2.2.3) и моделированием взаимодействий. Модели обоих типов описывают поведение одного прецедента (как правило). Однако, *моделирование видов деятельности* осуществляется на более высоком уровне абстракции — оно отражает последовательность событий вне связи событий с объектами. *Моделирование взаимодействий* отображает последовательность событий (сообщений) в их связи с действующими в кооперации объектами.

Диаграммы взаимодействий разделяются на два вида — диаграммы последовательностей и диаграммы кооперации. Они могут использоваться как взаимозаменяемые, и, конечно, многие CASE-средства поддерживают автоматическое преобразование одной модели в другую. Разница между моделями заключается в акцентах. Модели последовательностей концентрируются на временны*х* последовательностях событий, а в моделях кооперации основное внимание уделяется отношениям между объектами [76].

В настоящей книге для этих типов моделей выбрано следующее применение: диаграммы последовательностей используются на этапе анализа требований, а диаграммы кооперации — системного проектирования. Этот выбор соответствует общепринятой практике разработки ИС.

2.2.5.1. Взаимодействия

Взаимодействие (*interaction*) представляет собой набор сообщений, свойственных поведению некоторой системы, которыми обмениваются *объекты* в соответствии с установленными между ними *связями* (последние могут быть постоянными или временными (разд. 2.1.1.3)). Диаграмма последовательностей представляется двумерным графом. Объекты располагаются по горизонтали. Последовательности сообщений располагаются сверху вниз по вертикали. Каждая вертикальная линия называется *линией жизни* (*lifeline*) объекта (см. рис. 2.33).

Стрелки представляют каждое *сообщение*, направляемое от вызывающего объекта (*отправителя*) к операции (методу) вызываемого объекта (*получателя*). Для каждого сообщения, как минимум, указывается его имя. Кроме того, для сообщения могут быть указаны *фактические аргументы* сообщения и другая управляющая информация. Фактические аргументы соответствуют *формальным аргументам* метода объекта-получателя.

Фактические аргументы могут быть *входными аргументами* (передаются от отправителя к получателю) или *выходными аргументами* (передаются от получателя назад к отправителю). Входные аргументы могут быть обозначены ключевым словом `in` (если ключевое слово отсутствует, то предполагается, что аргумент — входной). Выходные аргументы обозначаются ключевым словом `out`. Допускаются также аргументы типа `inout` ("входные-выходные"), однако, для объектно-ориентированного подхода они не характерны. Сообщение `getCourseName` (см. разд. 2.1.1.3.1), отправленное объекту, обозначенному переменной `crs_ref`, имеет один выходной аргумент и ни одного входного:

```
crs_ref.getCourseName(out crs_name)
```

Показывать на диаграмме *возврат* управления от объекта-получателя объекту-отправителю не обязательно. Стрелка, указывающая на объект-получатель, предполагает автоматический возврат управления отправителю. Получатель знает уникальный идентификатор объекта (OID) отправителя.

Сообщение может быть отправлено *коллекции* (*collection*) объектов (коллекция может быть набором, списком, массивом объектов и т.д.). Довольно частой является ситуация, когда вызывающий объект связан с несколькими объектами-получателями (поскольку кратность ассоциации указана как "один ко многим" или "многие ко многим"). *Итеративный маркер* — звездочка перед обозначением сообщения — указывает на процесс итерации сообщения по всей коллекции.

Наставление по анализу: задание 13 (Internet-магазин)

Обратитесь к диаграмме видов деятельности на рис. 2.26. Рассмотрите первый шаг диаграммы `Display Current Configuration`. Постройте диаграмму последовательностей для этого шага.

Диаграмма последовательностей для "отображения текущей конфигурации " показана на рис. 2.33. Внешний субъект — клиент — (`Customer`) принимает решение об отображении конфигурации компьютера. Сообщение `openNew` (открыть новое [окно]) отправляется объекту `ConfWin` класса `ConfigurationWindow` ([диалоговое] окно конфигурации). В результате *создается* ("*материализуется как экземпляр*") новый объект `ConfWin`. (Класс `ConfigurationWindow` — *пограничный класс* (разд. 2.2.4.1).)

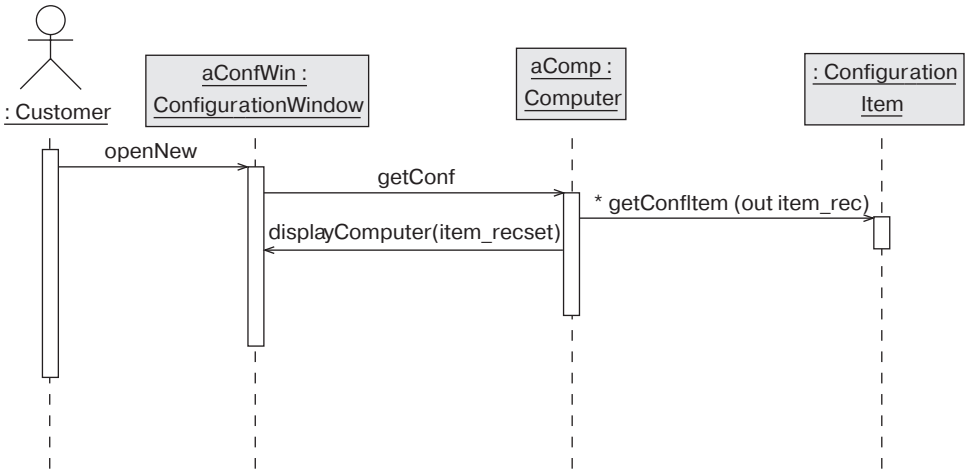

Рис. 2.33. Диаграмма последовательностей для вида деятельности Display Current Configuration (Internet-магазин)

Объекту `ConfWin` необходимо "отобразить себя" вместе с данными, относящимися к конфигурации. С этой целью он отправляет сообщение объекту `aComp:Computer`. В действительности, `aComp` — это объект класса `StandardComputer` или `ConfiguredComputer`. Класс `Computer` — абстрактный класс (рис. 2.31 в разд. 2.2.4.5).

Объект `aComp` использует выходной аргумент для того, чтобы "собрать себя" из элементов конфигурации — объектов `ConfigurationItem`. Затем он "оптом" отсылает элементы конфигурации объекту `aConfWin` в качестве аргумента `i_recset` сообщения `displayComputer`. Теперь объект `aConfWin` может отобразить себя. На экране компьютера выводится изображение, подобное показанному на рис. 2.34.

Обратите внимание, что поля для выбора на рис. 2.34 в начале отображают элементы стандартной конфигурации. Однако, эти поля для выбора заполнены другими не стандартными элементами, которые клиент может выбрать для создания новой допустимой конфигурации. Диаграмма последовательностей на рис. 2.33 не моделирует способ и время заполнения полей выбора данными.

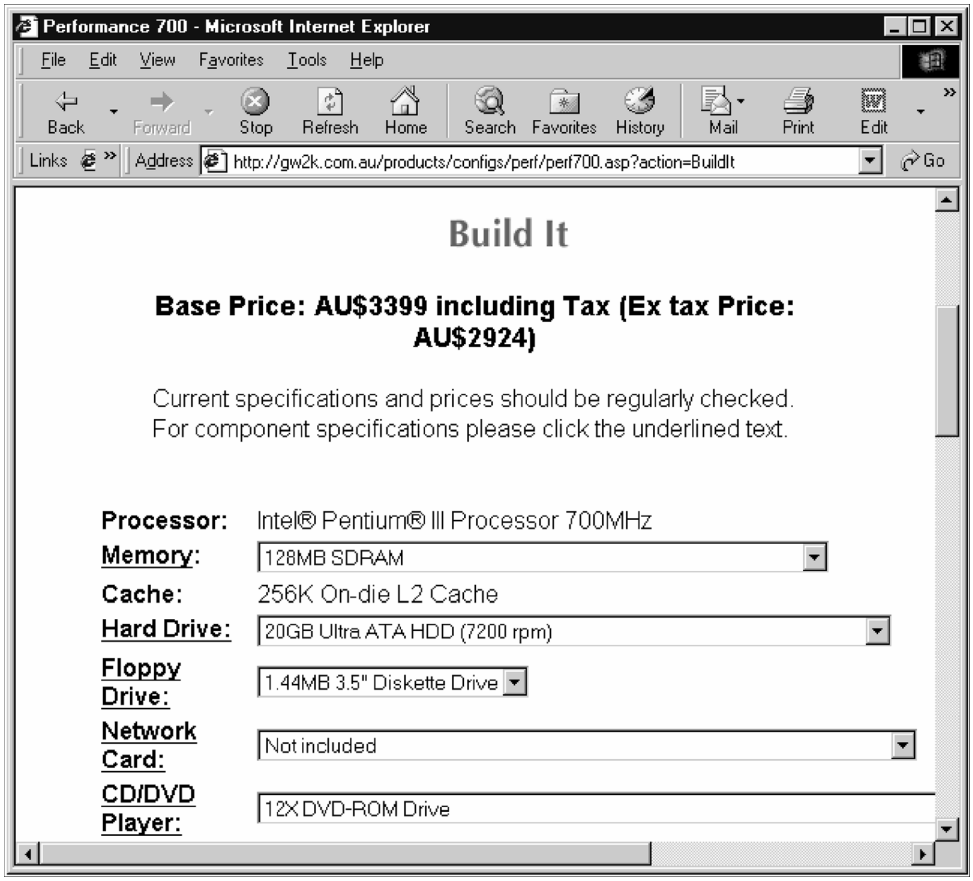

Рис. 2.34. Пример окна выбора конфигурации (Internet-магазин)

2.2.5.2. Операции

Хотя введение *операций* в классы зачастую откладывается до этапа проектирования, в данном наставлении будет показано, что исследование взаимодействий между классами может привести к выявлению операций. Между взаимодействиями и операциями существует прямая зависимость. Каждое *сообщение* обращается к операции в вызываемом объекте. Имена операции и сообщения совпадают.

Конечно, это однозначное отображение между *сообщениями* модели взаимодействий и *методами* реализуемых классов может иметь смысл только в том случае, если модель взаимодействий образует детализированный технический проект, что вряд ли возможно и может потребоваться на этапе анализа.

Попутно заметим, что подобное однозначное соответствие существует между *сообщениями* и *ассоциациями*, когда сообщение пересылается между *постоянными* (*модельными*) объектами. Эти сообщения должны поддерживаться постоянными связями (разд. 2.1.1.3.1). Таким образом наличие сообщения в диаграмме последовательностей обусловливает необходимость в ассоциации в диаграмме классов.

Наставление по анализу: задание 14 (Internet-магазин)

Обратитесь к диаграмме видов деятельности на рис. 2.32 и диаграмме последовательностей на рис. 2.33. Для каждого сообщения о взаимодействии на диаграмме последовательностей введите операцию в соответствующий класс на диаграмме классов. Не перерисовывайте всю диаграмму классов — только отобразите классы, расширенные за счет операций.

Решение для этого задания наставления показано на рис. 2.35. Изменениям подверглись три класса. Класс `ConfigurationWindow` — это пограничный класс. Два других класса — это классы-сущности, представляющие постоянные объекты базы данных. Операция `openNew` выбрана в качестве шаблона *операции конструктора*. Это означает, что операция `openNew` будет реализована в качестве метода конструктора, который порождает новые объекты класса.

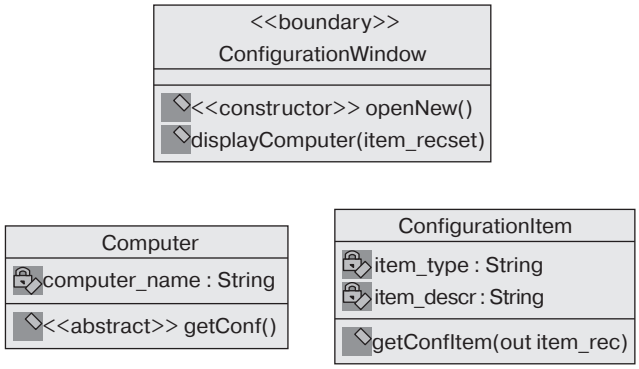

Рис. 2.35. Использование взаимодействий для введения операций в классы (Internet-магазин)

Класс `Computer` — *абстрактный класс*. Операция `getConf` — абстрактная операция, наследуемая подклассами (`ConfiguredComputer` и `StandardComputer`). Эти подклассы обеспечивают собственную реализацию операции `getConf`.

2.2.5.3. Диаграмма последовательностей

Как уже упоминалось, для каждого прецедента зачастую строится отдельная диаграмма последовательностей. Поскольку каждый прецедент вероятно должен быть выражен через диаграмму видов деятельности, диаграмма последовательностей строится для каждой диаграммы видов деятельности. Использование нескольких взаимоувязанных точек зрения на одну и ту же систему является краеугольным камнем надлежащего моделирования.

Решение для сформулированных выше вопросов показано на рис. 2.36. Для удобства диаграмма последовательностей разделена на две части (а именно, линии жизни объектов `Order` и `OrderWindow` разделены на две части).

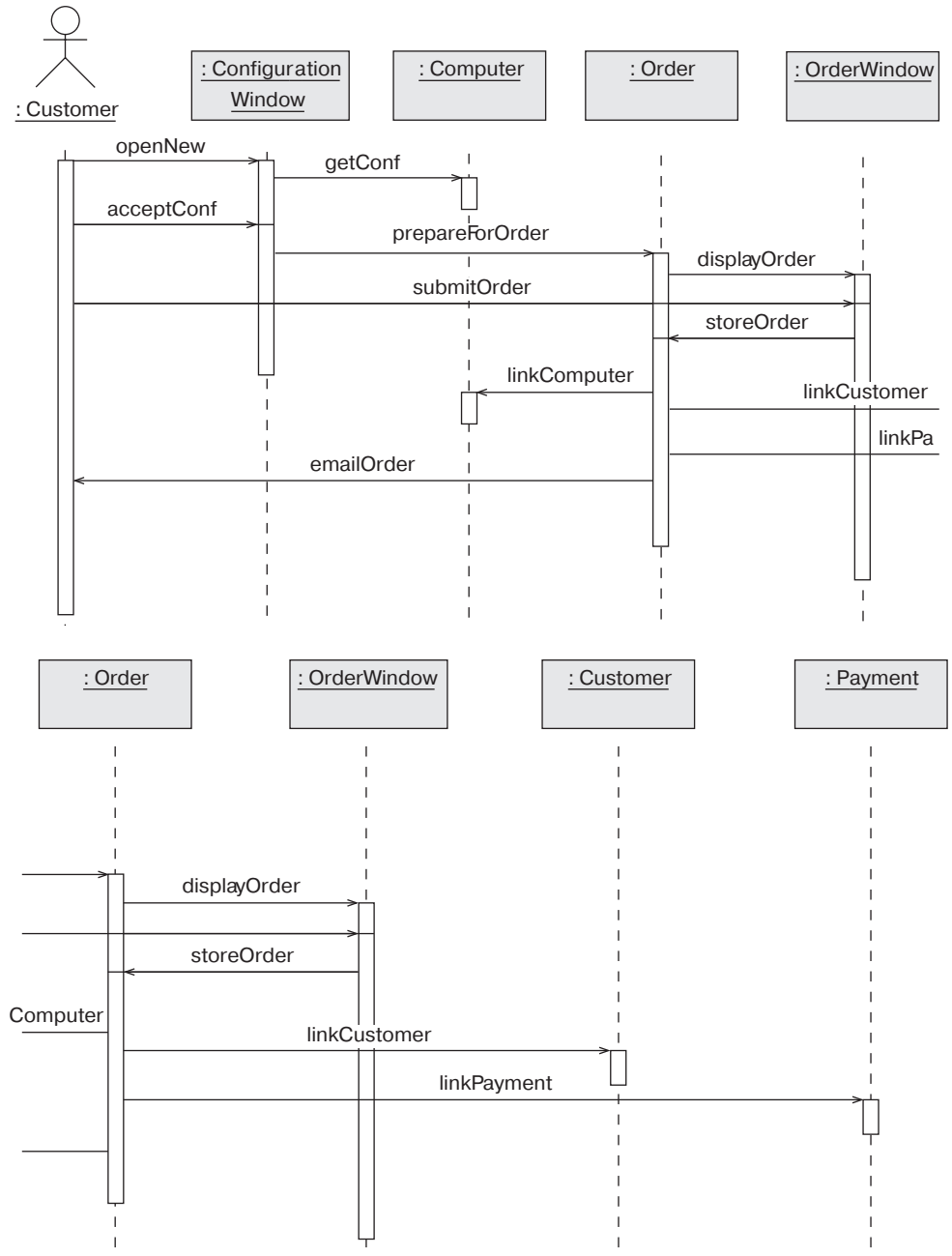

Рис. 2.36. Диаграмма последовательностей для диаграммы видов деятельности Order Configured Computer (Internet-магазин)

Наставление по анализу: задание 15 (Internet-магазин)

Обратитесь к диаграмме видов деятельности на рис. 2.26 и постройте для нее диаграмму последовательностей. Чтобы упростить диаграмму, не показывайте обмен сообщениями между объектами `Computer` и `ConfigurationItem`.

Кроме того, не изображайте объектов подклассов — считайте, что объект `Computer` принадлежит либо классу `StandardComputer`, либо классу `ConfiguredComputer`.

Покажите только сообщения активизации. Возвраты управления подразумеваются. Задавать аргументы операций или другую управляющую информацию нет необходимости.

Диаграмма последовательностей во многом является самоочевидной. Пояснения к первым двум сообщением были сделаны в задании 13 наставления. Сообщение `acceptConf` вызывает отправку сообщения `prepareForOrder` объекту `:Order`. Это приводит к созданию временного объекта `:Order`, отображаемого в "объекте-окне" `:OrderWindow`.

В ответ на принятие клиентом детализированных условий заказа (`submitOrder`) объект `:OrderWindow` инициирует (`storeOrder`) создание постоянного объекта `:Order`. После этого объект-заказ `:Order` устанавливает связь с заказанным компьютером (`:Computer`), а также соответствующими клиентом (`:Customer`) и платежом `:Payment`. После того, как эти объекты постоянно связаны в базе данных, объект `:Order` отправляет электронное сообщение `emailOrder` внешнему субъекту-клиенту (`Customer`).

Обратите внимание на двойное использование сущности `:Customer` — в качестве внешнего субъекта, представленного объектом, и внутреннего объекта-класса. Подобная двойственность довольно часто встречается в моделировании. Клиент (`Customer`) является по отношению к системе одновременно и внешней, и внутренней сущностью. Он представляет собой внешнюю сущность, поскольку взаимодействует с системой извне. Однако информация о клиенте должна храниться в системе, чтобы иметь возможность установить идентичность внешнего клиента и внутренней сущности, о которой знает система.

2.2.6. Моделирование состояний

Модель взаимодействий (*interaction model*) является источником детализированной спецификации прецедента. *Модель состояний* (*statechart model*) служит детализированным описанием класса или, более точно, динамических изменений состояний класса. Эти динамические изменения обычно описывают поведение объекта в рамках нескольких прецедентов.

Состояние (*state*) объекта обозначается текущими значениями его атрибутов (как элементарных атрибутов, так и атрибутов, обозначающих другие классы). *Модель состояний* (*statechart model*) фиксирует возможные состояния, в которых может находиться класс, и эффективно фиксирует "жизненный путь" класса. На протяжение своего жизненно цикла объект остается одним и тем же — его идентичность никогда не изменяется (разд. 2.1.1.3). Однако состояние объекта изменяется.

Диаграмма состояний представляет собой двудольный граф состояний (прямоугольников с закругленными углами) и переходов (стрелки), вызванных событиями. Основная концепция состояний и событий совпадает с концепцией, которая известна нам по диаграмме видов деятельности. Различие же заключается в том, что "состояния графа видов деятельности представляют состояния выполнения вычисления, а не состояния обычного объекта" [76].

2.2.6.1. Состояния и переходы

Значения атрибутов объекта изменяются, однако не все подобные изменения приводят к *переходу между состояниями*. Рассмотрим объект `BankAccount` (банковский счет) и связанное с ним бизнес-правило, по которому банк отказывается от взимания платы за услуги по ведению счета, когда остаток на счету (`balance`) превышает 100000 долларов. Можно сказать, что объект `BankAccount` переходит в привилегированное (`priviledged`) состояние. В противном случае объект пребывает в обычном состоянии. Остаток на счету (`balance`) изменяется после каждого перехода, соответствующего операции по снятию/вкладу денег, однако, состояние изменяется только тогда, когда баланс превышает или становится ниже порога в 100000 долларов.

Наставление по анализу: задание 16 (Internet-магазин)

Рассмотрите класс `Invoice` (Счет-фактура), относящийся к приложению *Internet-магазин*. Из модели прецедентов нам известно, что клиент определяет способ оплаты (кредитная карточка или чек) за компьютер, когда закупочная форма заполнена и отослана производителю. Это приводит к генерации заказа и последующей подготовке счет-фактуры. Однако, диаграмма прецедентов не проясняет вопрос о том, когда же производится фактическая оплата в соответствии со счет-фактурой. Можно, к примеру, предположить, что оплата может осуществляться до или после того, как счет-фактура выдана, а также предположить возможность частичной оплаты.

Из модели класса нам известно, что счет-фактура для заказа подготавливается продавцом, однако, в конечном итоге она передается на склад. Склад отправляет счет-фактуру клиенту одновременно с доставкой компьютера. Важно отметить, что состояние оплаты счет-фактуры отслеживается в системе, так что счет-фактуры снабжены надлежащими комментариями.

Изобразите диаграмму состояний, которая фиксирует возможные состояния счет-фактуры в той мере, в которой это касается платежей.

Приведенный выше пример схватывает сущность моделирования состояний. Модели состояний строятся для классов, которые характеризуются не просто изменениями состояний, а изменениями состояний, представляющими определенный интерес с точки зрения предметной области. Решение о том, что представляет интерес, а что нет, является прерогативой моделирования бизнес-процессов. Диаграмма состояний представляет собой модель бизнес-правил. В течение некоторого времени бизнес-правила остаются неизменными. Они относительно независимы от конкретных прецедентов. В действительности прецеденты должны соответствовать бизнес-правилам.

На рис. 2.37 показана модель состояний для класса `Invoice`. Начальным состоянием объекта `Invoice` является состояние `Unpaid` (не оплачено). Из состояния `Unpaid` возможны два перехода. При наступлении события `partial payment` (частичная оплата) объект `Invoice` переходит в состояние `Partly Paid` (частично оплачено). Допускается только одна частичная оплата. Наступление события `final payment` (окончательная оплата), когда объект `Invoice` находится в состоянии `Unpaid` или `Partly Paid`, инициирует переход в состояние `Fully Paid` (полностью оплачено). Это конечное состояние.

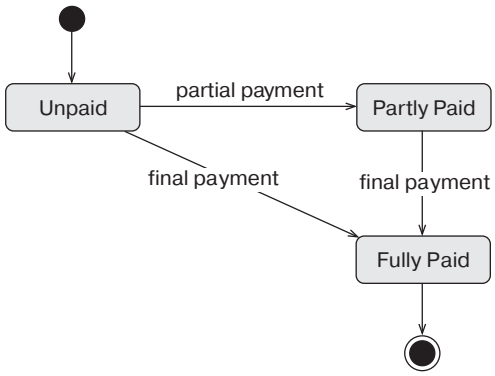

Рис. 2.37. Состояния и события для класса Invoice (Internet-магазин)

2.2.6.2. Диаграмма состояний

Диаграмма состояний обычно присоединяется к классу, но, в общем случае, она может присоединяться к другим модельным представлениям, например, прецедентам. Диаграмма состояний, присоединенная к классу, определяет способ реагирования объектов класса на события. Более точно, диаграмма определяет — для каждого состояния объекта — *действие* (*action*), выполняемое объектом при получении им сигнала о событии. Один и тот же объект может выполнять различные действия в ответ на одно и то же событие в зависимости от состояния объекта. Выполнение действия, как правило, вызывает изменение состояния.

Полное описание *перехода* (*transition*) состоит из трех частей:

```
event (parameters) [guard] / action.
```

Каждая из частей является необязательной. Если строка перехода самоочевидна, допустимо опустить все компоненты. Событие — это кратковременное явление, которое воздействует на объект. Оно может обладать параметрами, например, `кнопка мыши нажата (правая_кнопка)`. Событие может быть защищено *ограждающим условием* (*guard*), например, `кнопка мыши нажата (правая_кнопка) [внутри окна]`. Только в том случае, если условие принимает значение "истина", событие срабатывает и воздействует на объект.

Различие между событием и защитой не всегда очевидно. Это различие состоит в том, что *событие* "происходит" и оно может быть даже сохранено до тех пор, пока объект не приготовится к его обработке. С этих позиций защитное условие проверяется на истинность, чтобы определить, должен ли сработать переход.

Действие (*action*) представляет собой короткое атомическое вычисление, которое выполняется при срабатывании перехода. Действие также может быть связано с состоянием. В общем случае действие — это отклик объекта на обнаруженное событие. Состояние может содержать более продолжительное вычисление — *вид деятельности* (*activity*).

Состояние может состоять из других состояний, которые называются *вложенными* (*nested state*). *Составное состояние* (*composite state*) является абстрактным — это всего лишь родовое имя для всех вложенных состояний. Переход, который берет начало из-за границы составного состояния, означает, что он может сработать от любого из вложенных состояний. Это делает диаграмму более ясной и лаконичной. Конечно, пере-

ход, который берет начало вне границы составного состояния, может сработать от вложенного состояния.

Диаграмма состояний для класса `Order` показана на рис. 2.38. Начальное состояние объекта класса — `New Order` (новый заказ). Это одно из вложенных состояний составного состояния `Pending` (ожидающий [заказ]) — к другим относятся состояния `Back Order` (невыполненный заказ) и `Future Order` (будущий заказ). Из каждого из этих трех состояний, вложенных в состояние `Pending`, возможны два перехода.

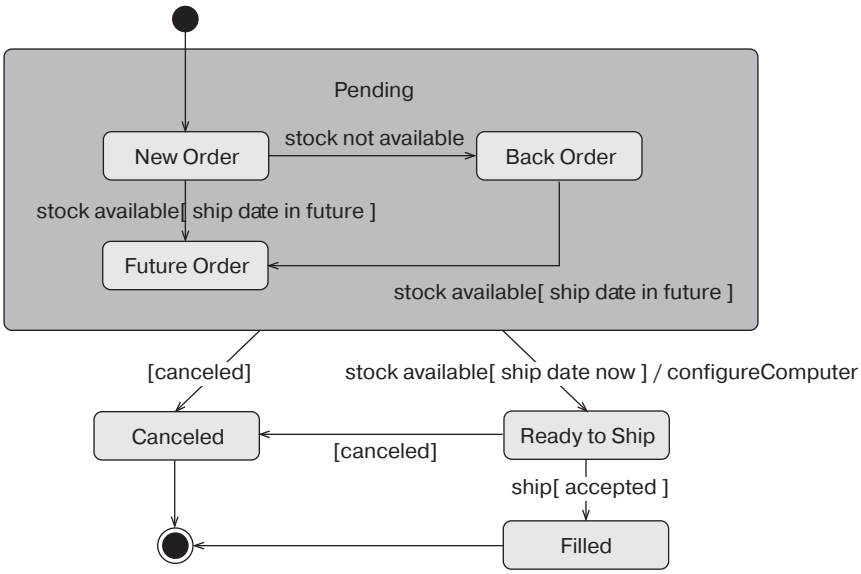

Рис. 2.38. Диаграмма состояний для класса Order (Internet-магазин)

Наставление по анализу: задание 17 (Internet-магазин)

Обратитесь к предыдущим заданиям наставления. Рассмотрите класс `Order` (заказ), относящийся к приложению *Internet-магазин*. Обдумайте, в каких состояниях может находиться класс `Order`, начиная от отправки заказа в систему и заканчивая его заполнением.

Рассмотрите ситуации, когда заказанный компьютер находится на складе в готовом виде или же его необходимо сконфигурировать в отдельности, чтобы удовлетворить ваши требования. Вы можете также задать определенный день, в который вы бы хотели получить компьютер, даже если он имеется на складе.

Вы можете отменить ваш заказ в любое время прежде, чем компьютер будет доставлен. Вам не требуется моделировать ситуацию взимания штрафа, связанную со слишком поздним отказом. Изобразите диаграмму состояний для класса `Order`.

Переход в состояние `Canceled` (отменен) защищено условием `[canceled]`. Должна существовать возможность — без изменения правил моделирования состояний — заменить защитное условие событием `cancel`. Переход в состояние `Ready to Ship` (готов к доставке) помечен полным описанием, содержащим событие, защиту и действие.

2.3. Постановки задач для исследуемых примеров

Чтобы проиллюстрировать различные виды деятельности по моделированию, на протяжении всей книги разбираются четыре примера. Разбор примеров представлен с использованием того же подхода, который использовался для наставления "Internet-магазин" – сначала формулируются вопросы, а затем даются решения. Это дает возможность читателям предложить собственные решения, а затем сравнить их с предложенными в книге. Итак, мы разберем следующие четыре примера.

1. Запись на университетские курсы.
2. Магазин видеопроката.
3. Управление контактами с клиентами.
4. Телемаркетинг.

2.3.1. Запись на университетские курсы

Запись на университетские курсы – классический пример из учебников [66], [85]. Это удивительно сложная прикладная область с богатым набором бизнес-правил, которые относительно часто изменяются, при этом данные за прошлые периоды должны тщательно храниться вместе с действовавшими на то время бизнес-правилами.

Постановка задачи 1. Запись на университетские курсы

Средний по размерам университет проводит набор студентов и аспирантов дневной и вечерней формы обучения для подготовки по ряду специальностей. Учебная структура университета состоит из факультетов. Факультет имеет в своем составе несколько кафедр. Хотя обучение и присвоение степени по определенной специальности является прерогативой факультета, обучение по специальности может включать дисциплины, преподаваемы на других факультетах. В действительности, университет гордится свободой, предоставляемой им студентам в выборе дисциплин, изучаемых для получения специальности.

Гибкость выбора учебных курсов оказывает дополнительную нагрузку на университетскую систему набора. Индивидуально подобранным программам обучения должны быть противопоставлены правила, регулирующие получение степени по выбранной специальности. Такие правила можно сформулировать, например, в виде структуры дисциплин, изучение которых является обязательной предпосылкой получения диплома, так, чтобы студент мог прослушать обязательные для данной специальности курсы. Выбор студентами дисциплин может быть ограничен несоответствием расписаний, максимальной вместимостью аудиторий и т.д.

Гибкость в получении образования, предлагаемого университетом, стала одной из причин устойчивого роста количества студентов. Однако, для сохранения своих традиционно сильных сторон система набора — по-прежнему, частично ручная — должна быть заменена новым программным решением. Предварительный поиск готовых программных пакетов не дал результатов. Университетская система набора достаточно уникальна, чтобы оправдать разработку ПО собственными силами.

От системы требуется оказывать помощь в предварительной деятельности по записи на курсы и управляться с процедурами набора. Предварительная деятельность по записи должна включать рассылку оценок последнего семестра студентам наряду с инструкциями по записи на лекции. В период записи на курсы система должна принимать заявленные студентами программы обучения и проверять их на предмет обязательности, конфликтов расписания, вместимости аудиторий, специальных санкций и т.д. Решения по некоторым проблемам могут требовать консультаций с научными руководителями или профессорами, ответственными за преподавание дисциплин.

Двух одинаковых университетов не существует. У каждого есть собственные любопытные особенности, которые можно использовать при разборе примеров, чтобы уделить основное внимание определенным аспектам системного анализа и проектирования. Основное внимание при разборе данного примера посвящается сложностям, которые возникают при моделировании состояний, связанным с обработкой временны*х* факторов (временно*й* информации) и добыванием бизнес-правил в структуры данных.

2.3.2. Магазин видеопроката

Наш второй пример — обычное бизнес-приложение, характерное для малого бизнеса. Это приложение предназначено для поддержки работы небольшого магазина видеопроката. В видеомагазине имеется широкий выбор последних и наиболее популярных фильмов на видеокассетах и компакт-дисках. Основным видом деятельности магазина является прокат видеофильмов.

Типичную компьютерную систему для поддержки работы небольшого видеомагазина можно получить посредством настройки готового ПО или настройки какого-либо другого патентованного решения. Система может базироваться на одной из популярных систем управления базами данных (СУБД), способной функционировать на компьютере, ресурсов которого хватает для решения задач малого бизнеса.

Несмотря на то, что базис системы составляет ПО баз данных, она может быть поначалу развернута на одной машине. GUI-интерфейс должен, вероятно, разрабатывться с использованием простого языка четвертого поколения (4GL), с возможностями создания экранных изображений, генерации программного кода и простым подключением к базе данных.

Отличительной чертой магазина видеопроката как примера для анализа является развитая цепочка видов деятельности — начиная с заказа видеопродукции с помощью управления ассортиментом и до расчетов с пользователями за прокат видеофильмов. В известном смысле — это модель цепочки ценностей, функционирующей в ограниченных масштабах (см. 1.2.2).

Постановка задачи 2. Магазин видеопроката

Вновь открывшийся видеомагазин намерен предложить прокат видеокассет и дисков широкой публике. Директор магазина полон решимости начать свою деятельность, опираясь на поддержку компьютерной системы. Директор уже навел справки по некоторым программным пакетам для малого бизнеса, которые могли бы пригодиться для настройки и последующего развития. Для оказания помощи в выборе пакета магазин нанял бизнес-аналитика, который занимается определением и спецификацией требований.

Поначалу ассортимент магазина составляет около тысячи видеокассет и пятьсот видеодисков. Запас уже заказан у одного поставщика, однако, для будущих заказов директор намерен прибегать к услугам большего числа поставщиков. Все видеокассеты и диски снабжены штрих-кодом, так что сканер, интегрированный в систему, может поддерживать операции выдачи напрокат и возврата видеофильмов. Членские карточки клиентов также снабжены штрих-кодом.

Существующие клиенты имеют возможность резервировать видео таким образом, чтобы комплект видеофильмов был собран к определенной дате. Система должна обладать гибким поисковым механизмом для ответов на запросы клиентов, включая вопросы, касающиеся фильмов, которых нет в ассортименте магазина (но которые он может заказать по просьбе клиента).

2.3.3. Управление контактами с клиентами

Управление контактами с клиентами — это "горячая" проблемная область. Больше известное под аббревиатурой CMS (Customer Management System — система управления работой с клиентами) управление контактами представляет собой важную составляющую ERP-систем (Enterprise Resource Planning — планирование ресурсов предприятия). ERP-системы автоматизируют так называемые *конторские* (*back office*) приложения обработки деловых операций. К трем наиболее типичным компонентам ERP-систем относятся: бухгалтерский учет, производство и управление людскими ресурсами. CMS-системы принадлежат к компонентам управления людскими ресурсами.

ERP-система — очень масштабное настраиваемое решение. Некоторые специалисты называют их мега-пакетами. Вполне естественно, что CMS-компонета ERP-решения может быть очень сложной. В нашем примере мы касаемся только небольшой части проблемной области CMS.

Приложения по управлению контактами с клиентами отличаются весьма интересными решениями в отношении GUI-интерфейса, с помощью которого работники службы связей с клиентами (Customer Relations) или аналогичного подразделения могут планировать свою деятельность, касающуюся клиентов. По существу GUI-интерфейс системы управления контактами работает как ежедневник по записи заданий и событий, связанных с клиентами, и позволяет следить за их ходом.

Подобный электронный ежедневник должен быть ориентирован на использование базы данных, позволяющей осуществлять динамическое планирование и отслеживать задания и события в отношении большого количества работников фирмы. Подобно большинству систем управления людскими ресурсами приложения по управлению контактами с клиентами требуют изощренной схемы авторизации для управления доступом к закрытой информации.

Постановка задачи 3. Управление контактами с клиентами

Компания, занимающаяся иследованием рынка, обладает стабильной клиентской базой организаций, которые приобретают отчеты по анализу рынка. Некоторые крупные клиенты приобретают у компании также специализированное ПО, предназначенное для создания отчетов. Этих клиентов компания также обеспечивает необработанной и агрегированной информацией для генерации их собственных отчетов.

Компания постоянно ищет новых клиентов, даже если новые клиенты заинтересованы только в получении однократных, узкоспециализированных отчетов по рынку. Хотя перспективные клиенты — это в полном смысле еще клиенты, компания стремится установить с ними контакт — отсюда, система управления контактами с клиентами (контакты бывают с перспективными, реальными и прошлыми клиентами).

Новая система управления контактами должна разрабатываться своими силами и находиться в распоряжении всех работников компании, но с предоставлением различного уровня доступа. Сотрудники отдела обслуживания клиентов (Customer Services Department) берут шефство над системой. Система должна обеспечить гибкое планирование и перепланирование видов деятельности, связанных с контактами, так чтобы сотрудники могли успешно кооперировать свои усилия в завоевании новых клиентов и стимулировании существующих взаимоотношений с клиентами.

2.3.4. Телемаркетинг

Многие организации продают свои товары и услуги с использованием телемаркетинга. Телемаркетинг – это прямое обращение к потребителям с помощью телефона. Система телемаркетинга требует поддержки детально продуманного процесса планирования телефонных звонков, осуществляемых продавцами, оказания помощи в поддержании разговора и фиксирования результатов разговора.

Одной из характерных особенностей систем телемаркетинга является сильная зависимость от способности базы данных активно планировать и динамически перепланировать телефонные звонки при поддержке параллельных разговоров. Другим интересным аспектом таких систем является возможность автоматического набора запланированного телефонного номера.

Постановка задачи 4. Телемаркетинг

Благотворительное общество продает лотерейные билеты для пополнения своего бюджета. Пополнение бюджета осуществляется в рамках *кампании*, направленной на поддержку текущих важных благотворительных мероприятий. Общество хранит список благотворителей (жертвователей). Для каждой новой кампании часть этих благотворителей заранее выбирается для обращения к ним с использованием телемаркетинга, а также, возможно, с помощью прямых обращений по почте.

Чтобы завоевать новых приверженцев, общество использует некоторые новейшие схемы. Эти схемы включают специальные *призовые кампании* для вознаграждения благотворителей, закупающих лотерейные билеты в массовом количестве, для привлечения новых жертвователей и т.д. Общество не обращается к случайным потенциальным благотворителям с использованием телефонных справочников или других подобных средств.

Для поддержки своей деятельности общество решило заключить контракт на разработку нового телемаркетингового приложения. От новой системы требуется поддерживать около пятидесяти сотрудников службы телемаркетинга, работающих одновременно. Система должна давать возможность планировать телефонные звонки в соответствии с заранее заданными приоритетами и другими известными ограничениями.

От системы требуется устанавливать соединения в соответствии с запланированными телефонными звонками. В случае неудачных попыток соединения звонки должны быть перепланированы так, чтобы можно было попытаться дозвониться позже. Обратные звонки от приверженцев также должны упорядочиваться. Результаты разговора, включая заказ лотерейных билетов и какие-либо изменения, касающиеся данных о благотворителях, должны сохраняться.

Резюме

В данной главе нам удалось охватить значительную часть вопросов, связанных с основаниями анализа требований. Были введены основополагающие термины и понятия, касающиеся объектной технологии. При изложении принципов объектного моделирования были использованы многочисленные поясняющие примеры, включая детальное наставление. Для новичков задания, должно быть, показались пугающими или унылыми. Награда за настойчивость не заставит себя ждать – она придет в следующих главах.

Объектная система состоит из взаимодействующих *объектов-экземпляров*. Каждый объект обладает состоянием, поведением и идентичностью. Последнее понятие является, наверное, самым существенным для правильного понимания объектных систем;

в то же время его также труднее всего оценить людям, имеющим за плечами определенный опыт в отношении традиционных компьютерных приложений. Перемещение по *связям* представляет собой modus operandi объектной технологии — нечто, что воспитанному на технологии реляционных баз данных читателю в действительности трудно переварить.

Класс — это шаблон, по которому создаются объекты. Он определяет *атрибуты*, которые объект может содержать, и *операции*, которые объект может вызывать. Атрибуты могут относиться к элементарному типу или обозначать другие классы. Атрибуты, которые обозначают другие классы, служат объявлением *ассоциаций*. Ассоциации — это один из видов отношений между объектами. В качестве других типов отношений выступают *агрегации* и *обобщения*.

Отношение обобщения обеспечивает основу для *полиморфизма* и *наследования*. Коммерческие программные среды могут поддерживать *множественное наследование*, но маловероятно, чтобы они поддерживали *множественную* или *динамическую классификацию*. С наследованием связано понятие *абстрактного класса*. Класс может обладать атрибутами и операциями, которые применимы только к нему, но не применимы ни к одному из его объектов-экземпляров. Подобные атрибуты и операции требуют введения понятия *объекта-класса*.

Понятия объектной технологии были закреплены в *наставлении*, сконцентрированном на разработке приложения для Internet-магазина. В наставлении был введен полный набор методов моделирования языка UML — моделирование прецедентов, видов деятельности, классов, взаимодействий и моделирование состояний. Перечисленные методы представлены на относительно высоком уровне абстракции, при этом многие детали моделирования были опущены. В последующих главах эти детали будут уточнены.

Наконец, мы сформулировали постановки задач для четырех *конкретных разбираемых примеров*, которые должны использоваться позже (наряду с наставлением) для иллюстрации и объяснения более сложных методов моделирования анализа и проектирования. Разбираемые примеры относятся к таким областям, как запись на университетские курсы, магазин видеопроката, управление контактами с клиентами и телемаркетинг.

Вопросы

В1. В чем состоит необходимость различения объектов-экземпляров и объектов-классов?

В2. Что такое идентификатор объекта?

В3. В чем заключается различие между временными и постоянными объектами?

В4. Что такое временная и постоянная связь? Каким образом они используются во время выполнения программы?

В5. Что подразумевают, когда говорят, что тип атрибута обозначает класс? Приведите пример.

В6. Хорошая объектная модель отличается тем, что большинство атрибутов в ней закрытых, а большинство операций — открытых. Почему?

В7. В чем различие между понятиями видимости и области действия операции?

В8. В какой ситуации при моделировании должен использоваться ассоциативный класс? Приведите пример.

В9. В чем различие между композицией и агрегацией?

В10. Объясните смысл замечания о том, что в типичной объектной среде программирования наследование применяется к классам, а не к объектам?

В11. Какова связь между переопределением и полиморфизмом?

В12. Что такое сигнатура?

В13. В чем различие между множественной классификацией и множественным наследованием?

В14. В чем преимущества использования абстрактных классов для моделирования?

В15. Объясните, в чем состоят основные отличительные черты и принципиальные отличия между моделью состояний, моделью поведения и моделью изменения состояний?

В16. Может ли субъект обладать атрибутами и операциями? Объясните свой ответ.

В17. Объясните, какова роль и место диаграммы видов деятельности в системном моделировании?

В18. В чем различие между ветвлением по условию и разделением потоков управления на диаграммах видов деятельности? Приведите пример.

В19. Что такое классы-сущности?

В20. Что такое фактический аргумент и что такое формальный аргумент?

В21. В чем состоит различие между действием и деятельностью на диаграмме состояний? Приведите пример.

Упражнения

У1. Обратитесь к рисунку 2.13 (разд. 2.1.3.2). Предположим, что предлагаемые учебные курсы включают лекции и практические занятия. При этом возможно, что один преподаватель отвечает за лекционную часть курса, а другой — за практическую.

Модифицируйте диаграмму на рис. 2.13 таким образом, чтобы учесть изложений выше факт.

У2. Обратитесь к рисунку 2.15 (разд. 2.1.3.4).

Постройте альтернативную модель, которая не использует ассоциативного класса и не использует тернарную ассоциацию (поскольку мы вообще не рекомендуем использование тернарных ассоциаций). Если между моделью, представленной на рис. 2.15, и вновь построенной моделью существуют семантические различия, опишите их.

У3. Обратитесь к рисунку 2.19 (разд. 2.1.5.2.1).

Расширьте пример за счет введения новых атрибутов в классы Teacher (Преподаватель), Student (Студент), PostgraduateStudent (Аспирант) и Tutor (Наставник).

У4. *Internet-магазин* — обратитесь к заданию 2 (разд. 2.2.2.2).

Пункт 6 в табл. 2.1 говорит о том, что клиенту должно быть отправлено сообщение по электронной почте, так что он может проверить состояние заказа через Internet. Прецедент, представляющий это событие, отсутствует. Можно ли его ввести? Объясните ваш ответ.

У5. *Internet-магазин* — обратитесь (в качестве примера) к заданию 2 (разд. 2.2.2.2) и заданию 3 (разд. 2.2.2.4).

Подготовьте документ описания прецедента для прецедента Update Order Status (Обновить состояние заказа). Прецедент дает возможность клиенту проверить состояние заказа на компьютер.

Отражает ли имя прецедента его функциональное назначение? Можно ли его изменить на другое имя?

У6. *Internet-магазин* — обратитесь к заданию 9 (разд. 2.2.4.3).

На рис. 2.29 класс Customer (Клиент) не связан непосредственно с классами Payment (Платеж), Invoice (Счет-фактура) и ConfiguredComputer (Сконфигурированный компьютер). Можно ли связать их в ассоциацию. Если это возможно, модифицируйте соответствующим образом диаграмму. Объясните ваше решение.

У7. *Internet-магазин* — обратитесь к заданию 11 (разд. 2.2.4.5).

Рис. 2.31 представляет собой относительно простой пример, иллюстрирующий обобщение. Какие сложности могут возникнуть, если потребуется ввести различия в информацию, вносимую в счет-фактуру для продажи стандартного компьютера (StandardComputer) и продажи сконфигурированного компьютера (`ConfiguredComputer`)? Например, может быть предусмотрена дополнительная плата, зависящая от модификации, требуемой для конфигурируемой системы (`ConfiguredComputer`), и, напротив, может быть предусмотрена скидка за оптовую покупку стандартной системы (`StandardComputer`).

Модифицируйте диаграмму таким образом, чтобы отразить подобное усложнение. Кратко поясните свое решение.

У8. *Internet-магазин* — обратитесь (в качестве примера) к заданию 13 (разд. 2.2.5.1) и заданию 6 (разд. 2.2.3.2).

Изобразите диаграмму последовательностей для вида деятельности `Display Purchase Form` (Отобразить закупочную форму) (рис. 2.26).

У9. *Internet-магазин* — обратитесь к решению упражнения 8, приведенного выше.

Введите дополнительные операции в классы, обозначенные объектами в диаграмме последовательностей.

У10. *Internet-магазин* — обратитесь к заданию 16 (разд. 2.2.6.1).

Диаграмма состояний, показанная на рис. 2.37, удовлетворяет ограничению, в соответствии с которым допускается только один частичный платеж. Предположим, что это не так, и разрешается осуществлять большее количество частичных платежей. Модифицируйте соответствующим образом указанную диаграмму.