Глава 6

Основания проектирования систем

Данная глава является прямым продолжением идей и тем, рассмотренных в предыдущей главе. Различие состоит в том, что теперь предмет рассматривается с точки зрения проектирования. Вопросы, которые имели меньшее значение для анализа, такие как технические подробности кооперативных действий объектов, теперь излагаются более глубоко.

Системное проектирование заключает в себе два основных вопроса: архитектурное проектирование и детализированное проектирование. Архитектурное проектирование охватывает многоуровневую организацию классов и пакетов, распределение процессов по вычислительным средствам, повторное использование и управление компонентами. Детализированное проектирование обращено к моделям кооперации, необходимым для реализации функциональных возможностей системы, зафиксированных в прецедентах.

Следуя принятому во 2-й главе книге принципу изложения материала, объяснение концепций сопровождается примерами из области системного проектирования. Пояснения собраны вместе в виде наставления по проектированию для приложения *Internet-магазин*, которое служит продолжением наставления по анализу главы 2.

6.1. Архитектура программного обеспечения

Проект представляет собой низкоуровневую модель архитектуры системы и ее внутренних функций. Проектирование осуществляется в терминах программно-аппаратной платформы, на которой предстоит реализовать систему. При итеративной и наращиваемой разработке модели анализа непрерывно “обрастают” техническими подробностями. Как только технические подробности включают соображения, касающиеся программного и аппаратного обеспечения, модель анализа становится проектной моделью.

Между анализом и проектированием нельзя провести резкой границы. Можно углубляться в технические вопросы, не касаясь специфических программно-аппаратных решений. В этом смысле большую часть вопросов, обсуждавшихся в главе 5, можно отнести скорее к анализу, чем к проектированию.

Описание системы в терминах ее модулей называется *архитектурным проектированием* (*architectural design*). Архитектурный проект включает выбор стратегии решений в отношении клиентской и серверной компонент системы.

Описание внутренних функций каждого модуля (прецедента) называется *детализированным проектированием* (*detailed design*). Детализированный проект направлен на разработку завершенных алгоритмов и структур данных для каждого модуля. Эти алгоритмы и структуры данных приспосабливаются ко всем (усиливающим и навязываемым) ограничениям базовой платформы реализации.

6.1.1. Распределенная архитектура

Архитектурное проектирование связано с выбором *стратегии решений* и *модуляризацией* системы. Стратегия решения призвана разрешить проблемы, связанные с построением клиентской и серверной частей системы, а ПО промежуточного слоя (middleware) необходимо для "склеивания" клиента и сервера. Решение по основным строительным блокам (модулям) только отчасти зависит от выбранной стратегии решения.

Клиент и сервер — логические понятия [6]. *Клиент* (*client*) — это вычислительный процесс, который осуществляет запросы к процессу сервера. *Сервер* (*server*) — это вычислительный процесс, который обслуживает запросы сервера. Обычно процессы клиента и сервера выполняются на разных компьютерах, но вполне возможно реализовать систему клиент/сервер на одной машине.

В типичном сценарии *клиентский процесс* отвечает за управление отображением информации на экране пользователя и за обработку событий, инициированных пользователями. *Процесс сервера* — это любой компьютерный узел с базой данных, из которой данные могут быть запрошены клиентским процессом.

Архитектуру клиент/сервер можно расширить для представления произвольной распределенной системы. Любой компьютерный узел с базой данных может играть роль клиента в одних деловых операциях, а сервер — в других операциях. Соединение подобных узлов с помощью сети связи дает начало *архитектуре системы распределенной обработки*, как показано на рис. 6.1.

Рис. 6.1. Архитектура системы распределенной обработки

В системе распределенной обработки клиент может осуществлять доступ к любому количеству серверов. Однако клиенту может быть разрешен доступ одновременно только к одному серверу. Это значит, что он может быть не в состоянии объединить данные от двух или более серверов баз данных в одном запросе. Если это возможно, то архитектура поддерживает *систему распределенных баз данных*.

6.1.2. Трехзвенная архитектура

В разд. 5.2.4 был рассмотрен подход BCE (boundary–control–entity — граница–управление–сущность). Также утверждалось, что подход BCE хорошо увязан с *трехзвенной архитектурой*, в которой между клиентом (граница) и сервером (сущность) вводится отдельный промежуточный слой логики приложения (управление).

Аналогично клиентскому и серверному процессу *прикладной процесс* представляет собой логическое понятие, которое может поддерживаться или не поддерживаться специально выделенным для этой цели аппаратным обеспечением. Логика приложения может с равным успехом выполняться на клиентском или серверном узле, т.е. может быть встроена в клиентский или серверный процесс и реализована в виде библиотеки DLL (Dynamic Link Library — динамически компонуемая библиотека), API-интерфейса (Application Programming Interface — интерфейс прикладного программирования), RPC-вызовов (Remote Procedure Calls — удаленный вызов процедуры) и т.д. [57].

Если логика приложения скомпилирована с клиентом, говорят об *архитектуре толстого клиента* (*thick client architecture*) ("клиент на стероидах"). Если она скомпилирована с сервером, говорят об *архитектуре тонкого клиента* (*thin client architecture*) ("клиент "кожа да кости"). Возможны также промежуточные архитектуры, в которых логика приложения частично скомпилирована с клиентом, а частично — с сервером.

Логику приложения можно также развернуть на отдельных вычислительных узлах, как показано на рис. 6.2.

Рис. 6.2. Трехзвенная архитектура

Это трехзвенная архитектура в самом чистом виде. К ее лучшим сторонам относятся высокая гибкость, расширяемость, независимость пользователя, готовность и низкая стоимость обновления. Однако подобная архитектура может отличаться высокой начальной стоимостью, а кроме того может испытывать некоторые проблемы с производительностью [90].

6.1.3. Программирование баз данных

Независимо от того, где расположена логика приложения, программа (клиент) взаимодействует с базой данных (сервером), чтобы получить информацию для ее отображения и предоставления в распоряжение пользователя для дальнейших манипуляций. Однако современные базы данных также можно программировать. Говорят, что эти современные базы данных *активны*.

Программы баз данных называются *хранимыми процедурами* (*stored procedure*). Хранимые процедуры хранятся в самой базе данных (они являются *постоянными* или *персистентными*). Их можно вызвать из клиентской программы (или из другой хранимой процедуры) с помощью обычного оператора вызова процедуры/функции.

Существуют хранимые процедуры специального вида — *триггеры*, которые нельзя вызвать явно. Триггер срабатывает автоматически при попытке изменить содержимое базы данных. Триггеры используются для реализации бизнес-правил масштаба предприятия, которые должны быть проведены в жизнь способом, независимым от клиентских программ (или хранимых процедур). Триггеры усиливают целостность и непротиворечивость баз данных. Они не позволяют отдельным приложениям нарушать бизнес-правила, заложенные в базу данных.

6.1.3.1. Взаимодействие “приложение–база данных”

Нам необходимо решить, какая часть системы будет запрограммирована в клиенте, а какая — в базе данных. При этом рассматриваются следующие программируемые части системы.

- Пользовательский интерфейс.
- Презентационная логика.
- Прикладные функции.
- Интегральная логика.
- Функции доступа к данным.

Часть программы, которая называется *пользовательским интерфейсом*, отвечает за отображение информации на конкретный GUI-интерфейс, такой как GUI-интерфейс Microsoft Windows, GUI-интерфейс Unix Motif, GUI-интерфейс Macintosh. *Презентационная логика* (или логика представления) отвечает за обработку объектов GUI-интерфейса (форм, меню, кнопок действий и т.д.), как того требуют функции приложения.

Функции приложения содержат основную логику программы. Они фиксируют действия приложения и представляют собой связующее звено, соединяющее вместе клиента и базу данных. С точки зрения подхода *BCE* (разд. 5.2.4) функции приложения реализуются классами *управляющего пакета.*

Интегральная логика отвечает за бизнес-правила масштаба предприятия. Это правила, которые применяются ко всем прикладным программам, т.е. все программы

должны функционировать в соответствии с ними. *Функции доступа к данным* владеют вопросами доступа к постоянным объектам данных на диске.

На рис. 6.3 показан типичный сценарий. Пользовательский интерфейс и презентационная логика принадлежат клиенту. Функции доступа к данным и интегральная логика (триггеры) находятся в ведении базы данных. Прикладные функции зачастую программируются (в виде SQL-запросов) как часть клиента в начале этапа разработки, а затем переносятся в базу данных (в виде хранимых процедур) для окончательного развертывания программного продукта.

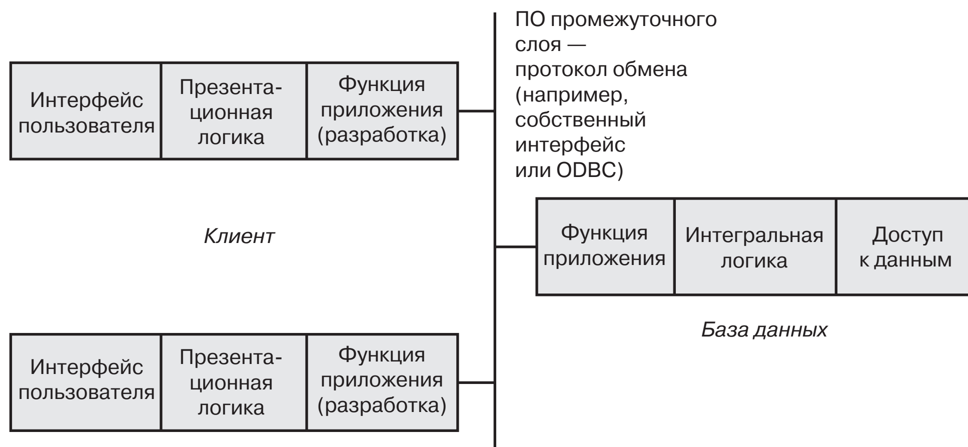

Рис. 6.3. Взаимодействие "приложение-база данных"

6.1.3.2. Подход *BCED*

Подход *BCED* (*Boundary–Control–Entity–Database— граница–управление–сущность–база данных*) является расширением подхода BCE (разд. 5.2.4). Он позволяет эффективно отделить `EntityPackage` (Пакет сущностей) от классов, ответственных за извлечение данных из базы данных. Эти классы помещаются в пакете `DatabasePackage` (Пакет базы данных) (рис. 6.4). (Пакет `DatabasePackage` в действительности является пакетом интерфейса с базой данных — `DatabaseInterfacePackage`, однако, мы предпочитаем короткие обозначения. Назвать пакет подобным образом — в любом случае не очень хорошая идея из-за многозначности понятия интерфейса.)

Пакет `EntityPackage` по-прежнему обеспечивает "исполняемую модель бизнес-процессов" [26]. Он хранит и помещает в буфер данные, полученные из базы данных и других источников; эффективно запрашивает две службы пакета `DatabasePackage`, а именно, службу загрузки объекта из базы данных `loadMe(anObject)` и службу сохранения объекта в базе данных `saveMe(anObject)` [26]. Реализация этих служб возможна в рамках классов пакета `DatabasePackage`, а именно, в виде классов `DatabaseReader` (Чтение базы данных) и `DatabaseUpdater` (Обновление базы данных) или классов с похожими именами.

Пакет `DatabasePackage` обеспечивает уровень взаимного обмена между приложением и базой данных. Логика приложения (`ControlPackage`) теперь отделена от изменений источника данных. Любые изменения структур баз данных и протоколов получения данных из базы данных влияют на классы пакета `DatabasePackage`. Хра-

нение и буферизация данных (пакет `EntityPackage`) для функций приложения остаются неизменными, поскольку об этом заботится пакет `ControlPackage`.

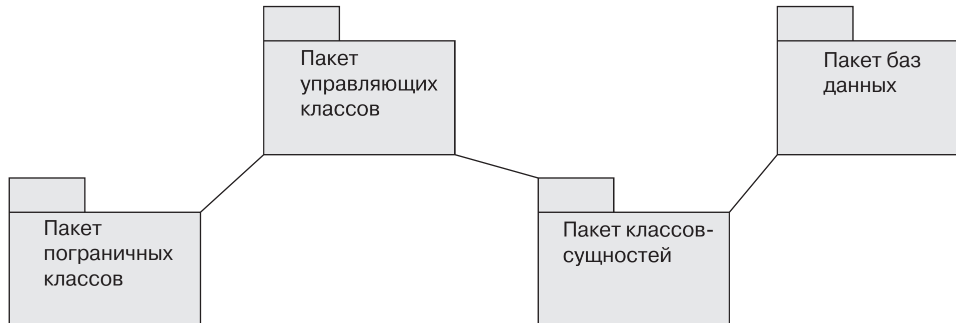

Рис. 6.4. Уровни BCED

В действительности, классы пакета `DatabasePackage` выполняют набор и других услуг. Эти дополнительные услуги могут состоять в открытии и закрытии соединения с базой данных, выдаче базе данных указания о подтверждении или откате транзакции, определении параметров времени выполнения для конфигурации базы данных, работе с авторизацией пользователей, извлечении и хранении метаданных об объектах базы данных (таких, как таблицы, столбцы, представления, хранимые процедуры, индексы и т.д.).

Разделение обязанностей, вводимое подходом *BCED*, можно "донести" до системных проектировщиков и программистов с помощью простого метода задания префиксов имен. Поскольку каждый класс может принадлежать только к одному пакету, его можно поименовать с помощью однобуквенного префикса (B, C, E или D). Например, класс, помеченный как `B_TreeBrowseView`, принадлежит пакету `BoundaryPackage`, а класс `D_DatabaseUpdater` — пакету `DatabasePackage`.

6.1.3.3. Системное ПО

Стратегия решения для *объектно-ориентированного клиента* обычно развивается вокруг программной среды со "строительными инструментами" в виде GUI-интерфейса. При этом решение, возможно, также учитывает используемый способ подключения к базе данных, при котором могут использоваться два основных подхода.

- *Собственный интерфейс базы данных*, предоставляемый многими графическими языками четвертого поколения — 4GL (например, PowerBuilder, Developer 2000, Delphi).
- Драйверы *ODBC* или *JDBC* для базы данных (например, в составе сред Visual C++, Power J++).

Стратегия решения для *сервера* определяет технологию баз данных. Поскольку "данные остаются навсегда, а приложения появляются и исчезают" (слова Боба Эпштейна (Bob Epstein) из *Sybase*, цитируются по памяти), стратегия решения для сервера, вероятно, оказывает значительное влияние на клиентскую стратегию. Часто выбирается та среда разработки клиента, которая предоставляется поставщиком СУБД.

К стратегическим решениям по серверу относятся следующие варианты СУБД.

- *Реляционные базы данных* (например, Sybase, Oracle, Informix).
- *Объектно-реляционные базы данных* (например, Oracle8, Informix/Illustra, UniSQL).
- *Объектные базы данных* (например, ObjectStore, Versant, Objectivity/DB).

В настоящее время перечисленные три технологии баз данных не конкурируют, они превосходят друг друга по разным вопросам и обращены к различным прикладным областям. В результате, решения по модели базы данных должны основываться на текущих и ожидаемых требованиях приложения.

6.1.4. Стратегия повторного использования

UML определяет *повторное использование* (*reuse*) как "использование ранее существовавших артефактов" [76]. Мы мимоходом уже обсуждали объектно-ориентированные методы для повторного использования ПО, такие как наследование и делегирование. В данном разделе обратимся к *стратегиям повторного использования ПО*. Оказывается, что стратегия также влияет на *степень детализации*, с которой осуществляется повторное использование. Могут применяться следующие *степени детализации повторного использования*.

- Класс.
- Компонента.
- Идея решения.

В связи со степенью детализации существуют три соответствующие стратегии повторного использования [13], [28], в основе которых лежат следующие программные сущности.

1. Инструментальные средства (библиотеки классов).
2. Каркасы.
3. Шаблоны анализа и проектирования.

6.1.4.1. Повторное использование инструментальных средств

Стратегия повторного использования *инструментальных средств* придает особое значение *многократному использованию программного кода* на уровне классов. При этой стратегии повторного использования программист "заполняет бреши" в программе за счет осуществления вызовов *конкретных классов* из некоторых библиотек классов. Основное тело (ядро) программы не подлежит повторному использованию — его пишет программист.

Существует два типа (уровня) инструментальных средств [60].

1. Базовые инструментальные средства.
2. Архитектурные инструментальные средства.

Базовые классы (*foundation classes*) широко представлены объектными программными средами. Они включают классы для реализации элементарных типов данных (такие как `String`), структурных типов данных (таких как `Date`) и коллекций (таких как `Set`, `List` или `Index`).

Архитектурные классы (*architecture classes*) обычно представлены как части системного ПО, такого как операционные системы, СУБД или ПО GUI-интерфейсов. Напри-

мер, когда мы приобретаем объектную СУБД, то фактически получаем архитектурные инструментальные средства, реализующие ожидаемые функциональные возможности системы, такие как поддержка хранения постоянных объектов, выполнение транзакций и обеспечение параллельности.

6.1.4.2. Повторное использование каркасов

Стратегия повторного использования *каркасов* (*framework*) придает особое значение *многократному использованию на уровне компонент* (компоненты рассматриваются в разд. 6.1.5). В противоположность повторному использованию инструментальных средств каркас предоставляет в распоряжение разработчиков скелет программы. Затем программист “наращивает” этот скелет (настраивает его) за счет написания программного кода, вызовы которого встроены в каркас. Помимо конкретных классов (для самого каркаса) каркас предоставляет в распоряжение программиста массу абстрактных классов, которые он должен реализовать (настроить).

Каркас — это настраиваемое прикладное ПО. Лучшими примерами каркасов служат ERP-системы (Enterprise Resource Planning Systems — системы планирования ресурсов предприятий), такие как SAP, PeopleSoft, Baan или J.D. Edwards. Однако повторное использование этих систем не основано на чистых объектно-ориентированных методах.

Объектно-ориентированные каркасы для разработки ИС предлагаются в рамках распределенных компонентных технологий, таких как CORBA, DCOM и EJB (см. разд. 1.1.1). Они известны как *“бизнес-объекты”* (*“business objects”*) — “поставляемые” программные продукты, призванные удовлетворить специфические деловые или прикладные потребности. Например, бизнес-объект может быть каркасом системы бухгалтерского учета с настраиваемыми классами, такими как `Invoice` (Счет-фактура) или `Customer` (Клиент).

Хотя каркасы и привлекательны с точки зрения повторного использования, они обладают рядом недостатков. Пожалуй, самым значительным из них является то, что исходное решение на основе “наименьшего общего знаменателя”, которое они предлагают, является условно оптимальным, а то и вовсе устарелым. В результате каркасы не дают конкурентного преимущества своим приверженцам и могут оказаться весьма обременительными в стремлении внедрить более современные решения.

6.1.4.3. Повторное использование шаблонов

Стратегия повторного использования *шаблонов* (*pattern*) придает особое значение *многократному использованию подходов*, при котором в распоряжение разработчиков предоставляются идеи и примеры *кооперативного взаимодействия* объектов, которые хорошо зарекомендовали себя в практике разработки и которые ведут к понятным и масштабируемым решениям (кооперативное взаимодействие рассматривается в разд. 6.2). Шаблоны могут применяться на этапе анализа или проектирования ЖЦ разработки ПО (отсюда — *шаблоны анализа* (*analysis patterns*) и *шаблоны проектирования* (*design patterns*)).

Шаблон — это документально оформленное решение, которое доказало свою работоспособность в ряде ситуаций. Данные ситуации четко обозначены и могут использоваться в качестве “статьи индекса” для поиска приемлемого решения в ходе разработки. Чтобы дать возможность разработчику принять обоснованное решение, необходимо привести перечень всех известных недостатков и побочных эффектов шаблона.

Повторное применение шаблонов во многом носит концептуальный характер, хотя многие проектные шаблоны содержат примеры программного кода для повторного ис-

пользования программистами. Рамки *проектного шаблона* (см., например, [28]) определяются кооперацией (разд. 6.2) — обычно они охватывают область более широкую, чем класс, но более узкую, чем компонента. Рамки *шаблона анализа* (см., например, [26]) зависят от уровня абстракции моделирования, к которой применяется шаблон.

6.1.5. Компоненты

Компонента (*component*) — это физическая часть системы, фрагмент реализации, программа [8], [48], [76], [88]. Компоненты зачатую воспринимаются как двоичные исполняемые (EXE-файлы) части системы. Но компонента может быть также частью системы, которая не является непосредственно исполняемым модулем (например, файлом исходного текста программы, файлом данных, динамически компонуемой библиотекой (Dynamic Link Library — DLL) или хранимой процедурой базы данных).

В настоящее время UML определяет пять стандартных стереотипов для компонент [8].

1. *Исполняемая* (т.е. непосредственно исполняемый модуль).
2. *Библиотека* (т.е. модуль статической или динамической объектной библиотеки).
3. *Таблица* (т.е. таблица базы данных).
4. *Файл* (т.е. файл исходного текста или данных).
5. *Документ* (т.е. документ, предназначенный для восприятия человеком).

Ниже приводится перечень характеристик компонент [76], [88].

- Компонента представляет собой независимо развертываемый программный блок (компонента никогда не развертывается частично).
- Компонента может служить строительным блоком для стороннего разработчика (т.е. компонента в достаточной мере документирована и самодостаточна, чтобы сторонний разработчик мог “встроить” ее в другие компоненты).
- Компонента не обладает тупиковым состоянием (т.е. компоненту невозможно отличить от ее собственных копий; в рамках любого данного приложения присутствует самое большее одна копия конкретной компоненты).
- Компонента — заменяемая часть системы, т.е. ее можно заменить другой компонентой, которая согласуется с тем же интерфейсом.
- Компонента выполняет четкую функцию и с логической и физической точки зрения образует единое целое.
- Компонента может быть вложена в другие компоненты.

6.1.5.1. Обозначение компонент

Графически компонента представляется как прямоугольник с двумя небольшими прямоугольниками-вставками с левой стороны. Уникальное имя компоненты помещается внутри большого прямоугольника. Между двумя любыми компонентами может быть задано отношение зависимости.

На рис. 6.5 показана компонента `InvoicingEXE`, которая относится к стереотипу исполняемых компонент.

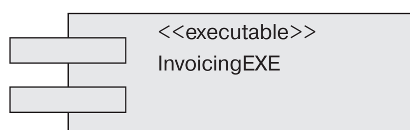

Рис. 6.5. Обозначение компоненты

6.1.5.2. Диаграмма компонент

Диаграмма компонент показывает компоненты и их взаимосвязь друг с другом. Компоненты могут быть связаны *отношениями зависимости*. Зависимая компонента запрашивает обслуживание у компоненты, на которую указывает отношение зависимости. Компоненты также могут быть связаны отношениями композиции, т.е. одна компонента может содержать другую.

На рис. 6.6 показаны три компоненты. Компонента `InvoicingEXE` зависит от двух других. Характер этих зависимостей не определен, поскольку интерфейсы компонент не показаны.

Язык UML допускает моделирование интерфейсов компонент с помощью символа "леденца на палочке" (разд. 5.3.3). Если зависимость между компонентами устанавливается в порядке договоренности через интерфейсы, то другая компонента, которая реализует тот же набор интерфейсов, может заменить одну из компонент, участвующих в отношении.

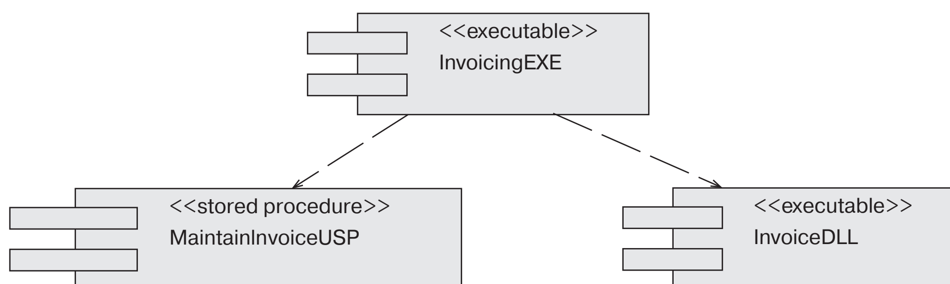

Рис. 6.6. Диаграмма компонент

6.1.5.3. Компоненты в сравнении с пакетами

Пакет (*package*) — логическая часть системы (разд. 5.2.3). На логическом уровне каждый класс принадлежит одному пакету. На физическом уровне каждый класс реализуется, по меньшей мере, одной компонентой, а компонента, возможно, реализует только один класс. Абстрактные классы, определяющие интерфейсы, зачастую реализуются более, чем одной компонентой.

Пакеты обычно представляют собой более крупные архитектурные элементы, чем компоненты. Они имеют тенденцию группирования классов *по горизонтали* — за счет статической близости классов, принадлежащих одной проблемной области. *Компоненты* — это *вертикальные* группы классов с близким поведением. Они могут принадлежать разным проблемным областям, однако, вносят вклад в один фрагмент деловой деятельности, возможно, представленный прецедентом.

Описанное выше свойство ортогональности пакетов и компонент затрудняет установление зависимостей между ними. Зачастую ситуация складывается так, что логический пакет зависит от нескольких физических компонент.

Пример 6.1. Запись на университетские курсы

Обратитесь к описанию пакетов, представленных в примере 5.10 (разд. 5.2.3.2). Рассмотрите пакет `Timetable`. Предположим, что пакет реализуется как программа на языке C++, которая включает логику распределения университетских аудиторий по группам. Программа получает доступ к базе данных по аудиториям и группам. Для предоставления программе этой услуги реализуются две хранимых процедуры.

Изобразите диаграмму компонент, которая показывает зависимости между пакетами и необходимыми компонентами.

На рис. 6.7 показана модель компонент. Она включает три компоненты: `RoomAllocEXE`, `RoomUSP` и `ClassUSP`. Пакет `Timetable` зависит от этих компонент.

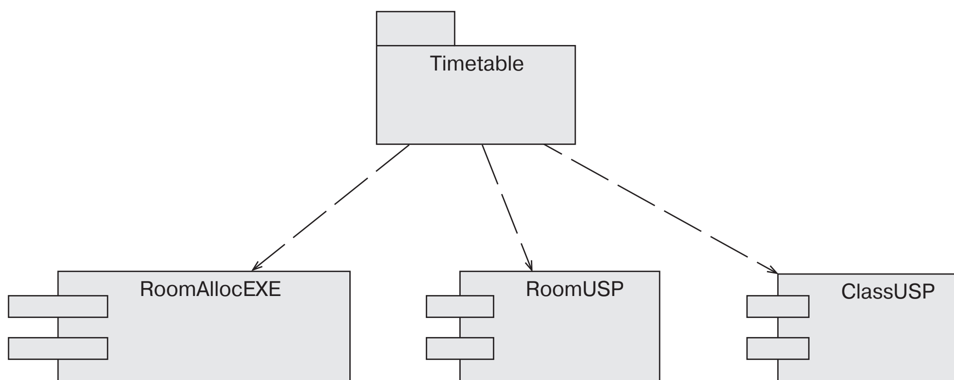

Рис. 6.7. Пакеты в сравнении с компонентами

6.1.5.4. Компоненты в сравнении с классами и интерфейсами

Подобно классам компоненты реализуют интерфейсы. Разница между ними двоякая. Во-первых, компонента — физическая абстракция, развертываемая на некотором компьютерном узле. Класс представляет логическую сущность, которая для того, чтобы действовать в качестве физической абстракции, должна быть реализована с помощью компоненты.

Во-вторых, компонента показывает только некоторые интерфейсы содержащихся в ней классов. Многие другие интерфейсы инкапсулированы компонентой — они используются только внутри кооперирующимися классами и не видимы другим компонентам.

Интерфейс, который конкретизируется компонентой, может быть реализован в отдельном классе. Подобный класс называется *доминантным классом* (*dominant class*) [76]. Поскольку доминантный класс представляет интерфейс компоненты, любой объект внутри компоненты достижим из доминантного класса через связи композиции. “Доминантный класс конкретизирует интерфейс компоненты” [76].

Пример 6.2. Запись на университетские курсы

Обратитесь к трем компонентам, представленным в примере 6.1 (разд. 6.1.1.3.3). Предположим, что компонента `RoomAllocEXE` инициирует процесс распределения аудиторий по курсам за счет обеспечения для компоненты `ClassUSP` идентификации классов. С этой целью реализует интерфейс под названием `Allocate`.
Компонента `ClassUSP` выполняет остальную работу, запрашивая подробную информацию об аудитории у компоненты `RoomUSP`. Для предоставления этой услуги `RoomUSP` реализует интерфейс под названием `Reserve`.

На рис. 6.8 показана диаграмма компонент, соответствующая сформулированным выше требованиям.

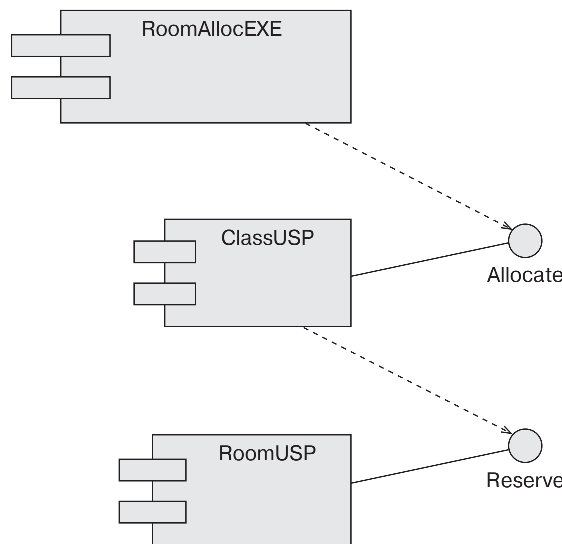

Рис. 6.8. Отображение интерфейсов на диаграмме компонент

6.1.6. Развертывание

В языке UML трехзвенная архитектура (наподобие той, что показана на рис. 6.2) или любая другая архитектура для системы выражается в виде *диаграммы развертывания* (*deployment diagram*). Фактически диаграмма, представленная на рис. 6.2, является допустимой диаграммой развертывания UML, на которой вычислительные ресурсы представлены в виде специальных пиктограмм.

Вычислительные ресурсы (физические объекты времени выполнения) называются *узлами* (*node*). Узел обладает, как минимум, памятью и некоторыми вычислительными возможностями (например, клиентский X-терминал на рис. 6.2). Узел может также быть сервером баз данных наподобие активного (программируемого) сервера.

6.1.6.1. Обозначение узлов

В UML узел графически представляется в виде куба. Куб может быть обозначен как стереотип. Подобный куб-стереотип может выглядеть как пиктограмма. Каждому узлу присваивается уникальное имя (текстовая строка).

На рис. 6.9 показан узел `CorporateDatabaseServer`. Стереотип «`Sybase`» говорит о том, что узел размещен на платформе СУБД Sybase.

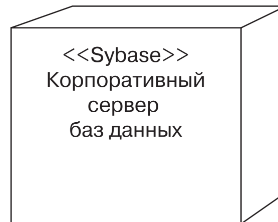

Рис. 6.9. Обозначение узла

6.1.6.2. Диаграмма развертывания

Диаграмма развертывания показывает узлы и взаимосвязи между ними. Узлы могут быть связаны *отношениями соединения* (*connection relationships*). Отношение соединения можно поименовать, чтобы указать используемый сетевой протокол (если это подходит) или описания характера соединения каким-либо другим способом. В общем случае отношение соединения представляет собой ассоциацию, и его можно моделировать с использованием типичных свойств ассоциации, таких как порядок, кратность, роли и т.д. (разд. 2.1.3).

На рис. 6.10 показаны два узла на диаграмме развертывания. Отношение соединения `download_nightly` говорит о том, что узел сервера хранилища данных `DataWarehouseServer` (функционирующий под управлением ПО IQ от Sybase) каждую ночь загружает данные с узла транзакционной базы данных `CorporateDatabaseServer`.

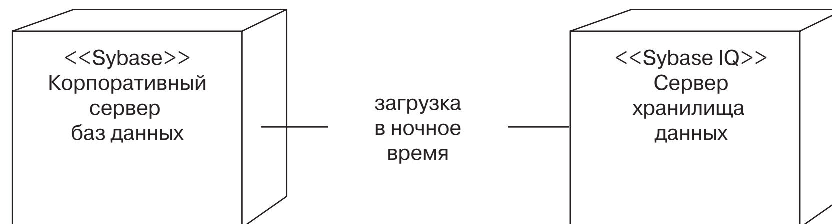

Рис. 6.10. Диаграмма развертывания

6.1.6.3. Узлы в сравнении с компонентами

Узлы представляют местоположение выполнения компонент. Компоненты функционируют на узлах. Компоненты развертываются на узлах. Узлы вместе с их компонентами иногда называются *элементами размещения* (*distribution unit*) [8].

На рис. 6.11 показан узел корпоративной базы данных `CorporateDatabaseServer`. На узле выполняются две хранимые процедуры, которые представлены как компоненты `CustomerUSP` и `InvoiceUSP`.

На рис. 6.12 показан альтернативный способ моделирования "включения" компонент в узел. Эту нотацию можно расширить таким образом, чтобы вся диаграмма компонент помещалась на диаграмме развертывания [25].

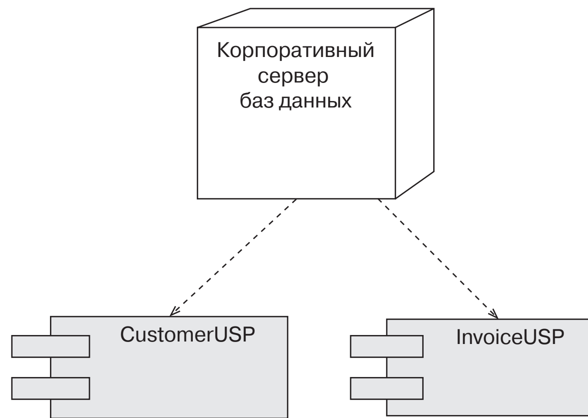

Рис. 6.11.Узел и его компоненты

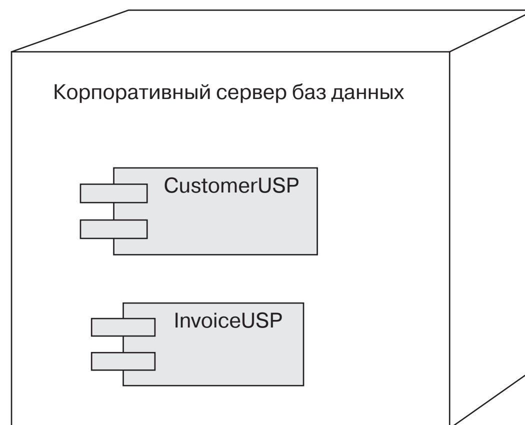

Рис. 6.12. Узел и его компоненты

6.2. Кооперация

Архитектурное проектирование оказывает влияние на детализированное проектирование в том смысле, что оно определяет целевую программно-аппаратную платформу, с которой должен быть согласован детализированный проект. Помимо этого, детализированное проектирование есть прямое продолжение анализа. Задача заключается в том, чтобы превратить модели анализа в документы детализированного проекта, на основе которых программисты могут реализовать систему.

В ходе анализа мы идем на упрощение моделей, абстрагируясь от деталей, которые служат помехой в представлении определенной точки зрения на систему. При проектировании мы стремимся действовать с точностью до наоборот. Берем одновременно одну из сторон представления архитектуры и добавляем в модель технические детали или создаем качественно новую проектную модель на более низком уровне абстракции.

Мы мимоходом довольно свободно использовали термин *кооперация* (*collaboration*), говоря о множествах объектов, которые кооперируются для выполнения задания (разд. 2.1.1.2 и 2.1.2.2.1). Язык UML использует термин кооперация в тех же целях и связывает с ним специфические методы моделирования. В частности, понятие кооперации используется в UML для задания реализации прецедентов или операций [8],[76].

6.2.1. Обозначение кооперации

Неудивительно, что нотация для *кооперации* похожа на нотацию для прецедентов. Кооперация обозначается в виде эллипса со штриховыми границами.

На рис. 6.13 показаны две кооперации `Browse Student List` (Показать список студентов) и `Add Student to Course Offering` (Занести студента в список [слушателей] курса). Две кооперации реализуют (через *отношение реализации* (*realization relationships*)) прецедент `Enter Program of Study` (Ввод программы обучения).

Заметим, что CASE-средства могут заменить явное моделирование отношения реализации на рис. 6.13 гиперссылками между прецедентом и моделью кооперации [8]. Другими словами, одна или более диаграмм UML, внутренне связанных с помощью CASE-средств, с их прецедентами могут представлять модель кооперации. Проектировщик может перемещаться между прецедентом и моделями кооперации, не прибегая к отношениям реализации.

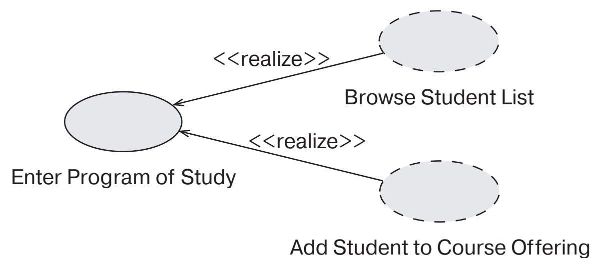

Рис. 6.13. Обозначение кооперации

6.2.2. Диаграмма кооперации

Диаграмма кооперации представляет собой один из видов модели взаимодействия UML (разд. 2.2.5). Второй вид — диаграммы последовательностей (разд. 2.2.5.3). Мы отдаем предпочтение диаграммам последовательностей при анализе и диаграммам кооперации при проектировании (по поводу сравнения этих двух типов диаграмм взаимодействия см. разд. 6.2.3).

Диаграмма кооперации на рис. 6.14 соответствует диаграмме последовательностей на рис. 2.33 (разд. 2.2.5.1). Номера последовательностей фиксируют временн*ы*е последовательности сообщений. Номера последовательностей необязательны. Для сложных алгоритмов бывает трудно приписать сообщениям и осмысленную временн*у*ю последовательность, поэтому для выражения временн*ы*х последовательностей могут потребоваться дополнительные модели (такие как диаграммы видов деятельности или псевдокод).

Заметим, что имена объектов на диаграмме кооперации в действительности представляют роли, которые играют объекты в кооперации. Соответствующая нотация UML выглядит так.

```
rolename:classname
```

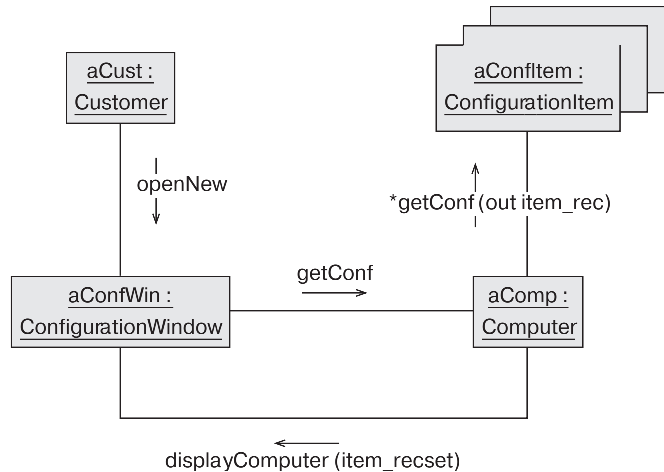

Рис. 6.14. Диаграмма кооперации "Display Current Configuration" (Internet-магазин)

Если быть более точными, то роль — это не объект. В языке UML роль — это "классификатор", представляющий любой объект, который может появиться в различных экземплярах кооперации. Кроме того, один и тот же объект может играть разные роли в последовательных кооперациях. В общем случае ролевое имя можно опустить, но перед именем класса необходимо поместить двоеточие, чтобы отличить его от обычного класса [76].

6.2.2.1. Обозначение сообщений

Структура сообщения соответствует сигнатуре целевого метода (разд. 2.1.2.2). Для успешной отправки сообщений объект-отправитель должен предоставить следующую информацию [60].

- Значение уникального идентификатора OID (дескриптор) для целевого объекта. Обычно оно хранится в одном из атрибутов отправителя (с точки зрения программирования — в переменных).
- Имя операции (метода) в целевом объекте.
- Дополнительно, фактические входные и выходные (возвращаемый результат) аргументы для подстановки вместо соответствующих формальных аргументов метода.

На рис. 6.15 показана альтернативная нотация UML для определения сообщения. Верхняя нотация, приведенная на рисунке, непосредственно отображается в сигнатуру метода (для краткости ключевые слова можно опустить). Средняя нотация использует возвращаемое значение и оператор присваивания вместо выходного аргумента. Нижняя нотация заменяет входные и выходные аргументы направленными маркерами данных.

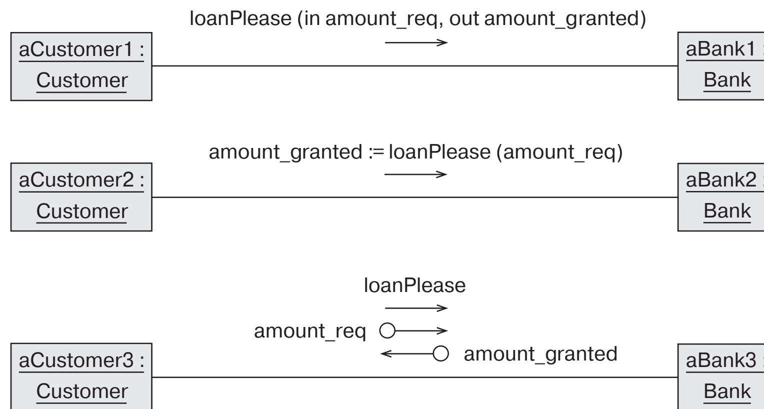

Рис. 6.15. Альтернативное обозначение сообщений

6.2.2.2. Структура сообщений

В программе отправителя сообщение на рис. 6.15 должно включать переменную, содержащую значение OID объекта-получателя. Переменная должна соответствовать имени объекта-получателя на диаграмме, например,

```
aBank1.loanPlease(in amount_req, out amount_granted)
```

Обратите внимание на порядок — первым идет объект, второй операция [60]. Порядок подчеркивает объектно-ориентированный способ мышления, при котором упор делается на кооперацию объектов, а не на процедурных вызовах. Он также указывает на возможный полиморфный характер сообщений, при котором одна и та же операция предоставления ссуды (`loanPlease`) может иметь разные реализации в разных классах (на объекты которых указывает значение OID переменной `aBank1`).

Аргументы сообщения на рис. 6.15 представляют собой маркеры данных. Это вполне приемлемо для популярных объектно-ориентированных сред, однако, неприемлемо с точки зрения чистого объектно-ориентированного подхода. В чистой реализации аргументы сообщения должны быть OID-переменными (*дескрипторами объектов*) [60]. Все, с чем мы имеем дело в чистых объектно-ориентированных системах, — это объекты!

Предположим на время, что аргументы сообщения `loanPlease` являются объектами класса `Money`. Следовательно, значения переменных `amount_req` (требуемое количество) и `amount_granted` (предоставляемое значение) должны быть OID-значениями, т.е. указателями на ячейки памяти, хранящие количество денежных средств для этих двух аргументов. Если это трудно для понимания, тогда вспомним, что класс для аргумента может быть любым классом, определенным пользователем, таким как `Employee` или `BankAccount`, а не просто элементарным типом, таким как `Money` или `Integer`. Теперь легче понять, почему аргумент сообщения представляет собой OID-дескриптор объекта.

6.2.2.3. Типы сообщений

Сообщение можно отправить объекту-классу (разд. 2.1.6) или объекту-экземпляру. Обычно сообщения, отправляемые объекту-классу, представляют собой *конструкторы* (*constructors*) (предназначенные для создания новых объектов-экземпляров) и *деструкторы* (*destructors*) (предназначенные для уничтожения существующих объектов-экземпляров). Остальные типы сообщений (направляемые как объектам-классам, так и объектам-

экземплярам) можно разделить на следующие три группы (альтернативные имена в скобках заимствованы из [60]).

- Сообщение чтения (вопросительное, адресованное настоящему).
- Сообщение обновления (информативное, адресованное прошлому).
- Кооперативное сообщение (повелительное, обращенное к будущему).

Сообщение чтения (*read message*) предписывает объекту-получателю предоставить некоторую информацию, возможно, значения его закрытых атрибутов. Оно “адресовано настоящему”, поскольку запрашивает текущую информацию. Вот пример подобного сообщения.

```
aBank1.openingHours(in weekday, out hours)
```

Сообщение обновления (*update message*) предписывает объекту-получателю обновить себя на основе информации, предоставляемой сообщением. Оно “адресовано прошлому”, поскольку обновляет объект с использованием информации, которая уже известна отправителю. Ниже приведен пример подобного сообщения.

```
aCustomer1.newCreditRating
            (in credit_rating, effective_date)
```

Кооперативное сообщение (*collaborative message*) предписывает объекту-получателю оказать помощь в выполнении запрашиваемого действия, которое является вкладом в кооперацию. Оно “обращено к будущему”, поскольку запрашивает со стороны отправителя объект о проведении действия как части более крупной задачи, которая должна быть завершена в ближайшем будущем. Примером кооперативного сообщения служит сообщение `loanPlease`. Для завершения запроса объект `aBank1` может запросить другие объекты о дальнейшем сотрудничестве.

```
aBank1.loanPlease(in amount_req, out amount_granted)
```

6.2.2.4. Сравнение замещения и перегрузки

При проектировании моделей взаимодействия следует учитывать, что имя метода может быть замещено или перегружено. Эти два термина обозначают не одно и то же.

Замещение метода (*overriding*) было рассмотрено в разд. 5.3.4.4.2. Замещение составляет базис полиморфизма. Оно означает, что существует несколько методов с одним и тем же именем в различных классах. Различные реализации одного метода должны выполняться в зависимости от того, какому классу принадлежит целевой объект.

Перегрузка метода (*overloading*) также означает, что существует несколько методов с одним и тем же именем, но в одном классе. Например, помимо ранее определенного метода `loanPlease` можно ввести еще один метод `loanPlease` в класс `Bank`. Этот метод должен включать дополнительный аргумент, задающий минимальный размер займа, который может получить клиент, как показано ниже

```
aBank1.loanPlease(in amount_req, minimum_amount, _out amount_granted)
```

Теперь метод `loanPlease` перегружен. Выполнение того или иного перегруженного метода зависит от сигнатуры сообщения. Отсюда следует, что установление сигнатуры сообщений — весьма важное мероприятие на этапе проектирования.

6.2.2.5. Итеративные сообщения и шаблоны

Итеративное сообщение (*iteration message*) — это сообщение, отправка которого повторяется для нескольких объектов класса. Для обозначения итерации в языке UML используется *маркер итерации* — звездочка перед меткой сообщения (разд. 2.2.5.1).

Множество объектов, на которые распространяется итеративное сообщение, называется *коллекцией* (*collection*). Коллекция может быть набором, списком (неупорядоченным набором) или массивом объектов. На диаграмме кооперации коллекция изображается как комплект ("штабель") объектов (рис. 6.14 в разд. 6.2.2).

В смысле объектно-ориентированного подхода коллекция (набор, список и т.д.) сама является классом. Естественно, многие среды объектно-ориентированного программирования по умолчанию включают соответствующие классы с именами `Set`, `List`, `FixedArray`, `VarryingArray`, `BtreeIndex` и т.д. Эти классы-коллекции можно затем использовать для включения в них объектов, определяемых пользователем классов. Типичным примером использования коллекции классов может служить реализация связей отношений ассоциации или агрегации с кратностью "многие" (разд. 2.1.3.2).

Рассмотрим рис. 6.14 и итеративное сообщение `*getConf(out item_rec)` от объекта класса `aComp` к объекту `aConfItem`. Сообщение распространяется на коллекцию объектов `ConfigurationItem`. Однако из диаграммы кооперации не ясно, каким образом реализована коллекция. Эту информацию можно вывести из диаграммы класса.

На рис. 6.16 приведена диаграмма классов, отображающая отношение агрегации между классами `Computer` и `ConfigurationItem`. Заметим, что ролевые имена предполагается в конечном счете реализовать как атрибуты классов. На рис. 6.17 приведена диаграмма классов уровня реализации, на которой ролевые имена преобразованы в имена атрибутов.

Как видно из рис. 6.17 , типом атрибута `has_conf_item`, описывающего выбранный элемент конфигурации компьютера, является список классов для этих элементов `List<ConfigurationItem>`. Класс `ConfigurationItem` является параметром класса-коллекции `List`. В общем случае класс `List` может заменить свой формальный параметр другими классами. Мы говорим, что тип `has_conf_item` является *шаблоном* (*template*) — *параметризованным типом* (*parameterized type*).

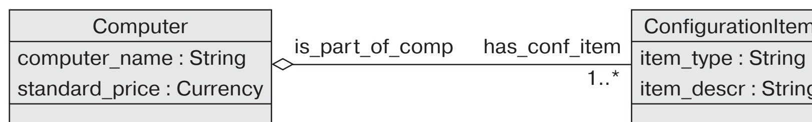

Рис. 6.16. Отношение агрегации с указанием ролей

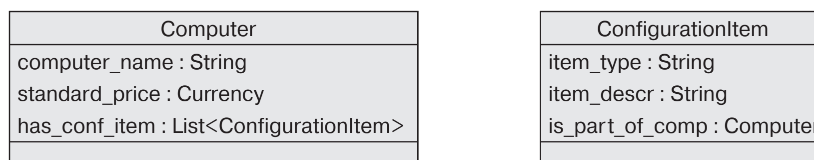

Рис. 6.17. Отношение агрегации, уже реализованное с использованием атрибутов

Определение шаблона играет важную роль в понимании области действия итеративного сообщения. Сообщение `*getConf(out item_rec)` распространяется на объекты класса `ConfigurationItem`, входящие в список `List`. Список образует текущее значение атрибута `has_conf_item`. Если быть более точным, список содержит OID-значения (дескрипторы) всех объектов `ConfigurationItem`, включенных в конкретный объект-компьютер `Computer`.

6.2.2.6. Автосообщения

Объект может отправить сообщение самому себе. *Автосообщение* (*self message*) означает локальный вызов — один метод вызывает другой метод одного и того же объекта. Название автосообщение заимствовано из языка Smalltalk. В языках C++ и Java автосообщению соответствует термин *this* применительно к объекту. Соответствующий объект представляет собой константу (не переменную), которая хранит идентификатор (OID) объекта [60].

Автосообщение может встретиться в последовательности кооперативных сообщений (разд. 6.2.2.3), которые воздействуют на тот же класс до передачи управления другому классу. Оно может также встретиться, если управление возвращается отправителю и следующее сообщение вызывает метод отправителя (отправитель и получатель — один и тот же объект).

На рис. 6.18 приведен пример двух автосообщений (`currentLeave` и `longServiceLeave`), которые активизируются в результате запроса на вычисление значения суммы, подлежащей выплате работнику за отпускной период — `leaveEntitlement`.

Автообъект может быть передан в некоторых редких случаях, как аргумент в сообщении, т.е.

```
messagename(self)
```

При передаче *синхронных сообщений* (*synchronous message*) (во время которой отправитель должен дождаться возврата управления от целевого объекта прежде, чем завершить выполнение) пересылка OID-отправителя, как правило, не требуется. Возврат управления выполнением происходит автоматически, по крайней мере в большинстве объектно-ориентированных сред программирования.

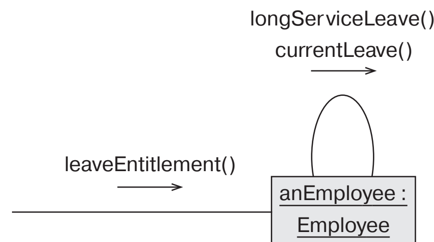

Рис. 6.18. Автосообщения

6.2.2.7. Асинхронные сообщения

При передаче *асинхронных сообщений* (*asynchronous message*) объект-отправитель не должен дожидаться, пока целевой объект закончит свою работу, а продолжать выполнение другого потока управления. Передача асинхронных сообщений предполагает существование множественных потоков управления (*параллелизм* (*concurrency*)) в программе.

Для обозначения асинхронных сообщений в UML используются "односторонние" стрелки. Объект `me` (я) на рис. 6.19 порождает два потока управления, отправляя два асинхронных сообщения объектам `myCoffeeMaker` (моя кофеварка) и `myRadio` (мое радио).

Передача асинхронных сообщений характерна для инженерных приложений и приложений реального времени. Типичные бизнес-приложения, выполняющие деловые операции, по большей части базируются на синхронных сообщениях.

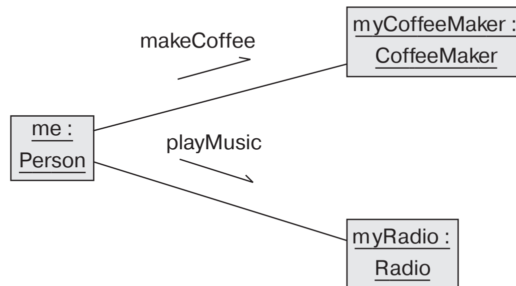

Рис. 6.19. Передача асинхронных сообщений

6.2.2.8. Обратные вызовы

Обратные вызовы упоминались ранее в этой книге в контексте наследования (разд. 5.3.4.4.2). *Обратный вызов* (*callback*) — это обращение "снизу-вверх" (если смотреть по связи отношения) к отправителю сообщения. Типичным примером использования обратного вызова служит передача асинхронного сообщения, при котором отправитель (*абонент* (*subscriber*)) заявляет целевому объекту (*получателю* (*listener*)), что ему необходимо получить информацию о завершении действия, выполняемого получателем. Целевой объект (тот, на который указывает связь отношения) должен отправить асинхронное сообщение назад отправителю.

На рис. 6.20 приведена расширенная диаграмма кооперации, показанная на рис. 6.19, включающая обратный вызов от объекта `myCoffeeMaker` к объекту `me`. Как-никак я (`me`) жду кофе, и мне совсем не хочется прерываться, чтобы посмотреть, готов ли он — об этом мне просигналит сама кофеварка (`myCoffeeMaker`).

Обратные вызовы — удобный механизм, однако, их очень трудно программировать. Чтобы обратный вызов работал надлежащим образом, получатель должен иметь возможность каким-либо образом наблюдать за состоянием и изменением состояния абонента прежде, чем отправить обратный вызов. Отправлять мне сообщение о том, что кофе готов (`coffeeReady`), когда я уже в пути, не очень умная затея!

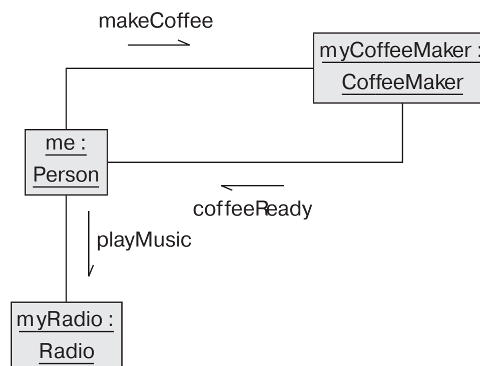

Рис. 6.20. Обратный вызов при передаче асинхронных сообщений

6.2.3. Диаграммы последовательностей и диаграммы кооперации

Диаграммы кооперации и диаграммы последовательностей эквивалентны в том смысле, что их можно автоматически преобразовать одну в другую. В то же время эти два типа диаграмм выделяют различные аспекты взаимодействия объектов (и иногда эти различные аспекты в процессе преобразования могут быть утеряны).

Диаграммы последовательностей придают особое значение временной последовательности сообщений между объектами. Они неудобны и не дают необходимой точности при представлении альтернативных путей сообщений — то, в чем отлично зарекомендовали себя диаграммы видов деятельности. Кроме того, они громоздки при представлении кооперативных действий многих объектов (хотя тщательное упорядочение жизненных линий объектов зачастую может улучшить удобство восприятия диаграммы в целом).

Диаграмма кооперации может явно отображать статические отношения между объектами, по которым может протекать поток сообщений. В результате, диаграммы кооперации обеспечивают большую точность при визуализации таких вещей, как полиморфное сообщение. Схема диаграмм кооперации позволяет отображать большее количество объектов на той же графической площади. Сообщения могут быть полностью определены и аннотированы.

Мы предпочитаем использовать диаграммы последовательностей для моделей анализа и диаграммы кооперации для проектных моделей. Оба типа диаграмм можно дополнить диаграммами видов деятельности. При изображении диаграмм видов деятельности можно использовать уровень абстракции, соответствующий уровню абстракции связанных с ними диаграмм последовательностей или кооперации.

6.2.4. Реализация прецедентов

Кооперация значит для проекта то же самое, что прецеденты для анализа. Если прецеденты направляют анализ, то кооперация объектов направляет проектную деятельность. Прецеденты реализуются с помощью кооперативных действий. Из-за различия в уровне абстракции каждый прецедент реализуется несколькими схемами кооперации.

Кооперативные взаимодействия имеют две стороны: структурную и поведенческую. *Структурная сторона* представляет статический аспект кооперации. Он представляется подмножеством диаграмм классов, соответствующих рамкам кооперации. Диаграмма классов *детализируется* (в сравнении со своей версией, полученной в результате анализа) за счет подробностей реализации. В частности, осуществляется идентификация сигнатур операций классов.

Поведенческая сторона представляет динамику, которая показывает, каким образом осуществляется кооперативное взаимодействие статических элементов. Поведенческий аспект моделируется именно в виде *взаимодействий* (разд. 2.2.5). Помимо диаграмм последовательностей (разд. 2.2.5.3), диаграммы кооперации часто используются для спецификации поведения объектов в ходе кооперативных действий. На практике структурный и поведенческий аспекты кооперации по возможности следует разрабатывать параллельно.

6.2.4.1. Структурный аспект кооперации

Для решения задачи необходимо рассмотреть все дополнительные классы и дополнительные атрибуты и операции, необходимые для поддержки кооперации. Нам требуется детализировать диаграмму классов, выработанную в ходе анализа. При этом следует руководствоваться диаграммой последовательностей для прецедента, разработанной в ходе анализа (рис. 4.14, разд. 4.3.3.3).

Как видно из рис. 6.21, к диаграмме классов добавлено всего несколько классов. Пограничный класс `ProgramEntryWindow` получен на основе диаграммы последовательностей, показанной на рис. 4.14. Класс `PrereqCourse` требуется для установления отношения “многие ко многим” проверки обязательных условий. Класс `Grade` (Оценка) добавлен для полноты, однако, в рассматриваемом прецеденте больше использоваться не будет. Прошлые оценки студентов хранятся в классе `AcademicRecord`. Класс `Grade` содержит отметки и оценки, которые студент получает при прохождении текущего курса.

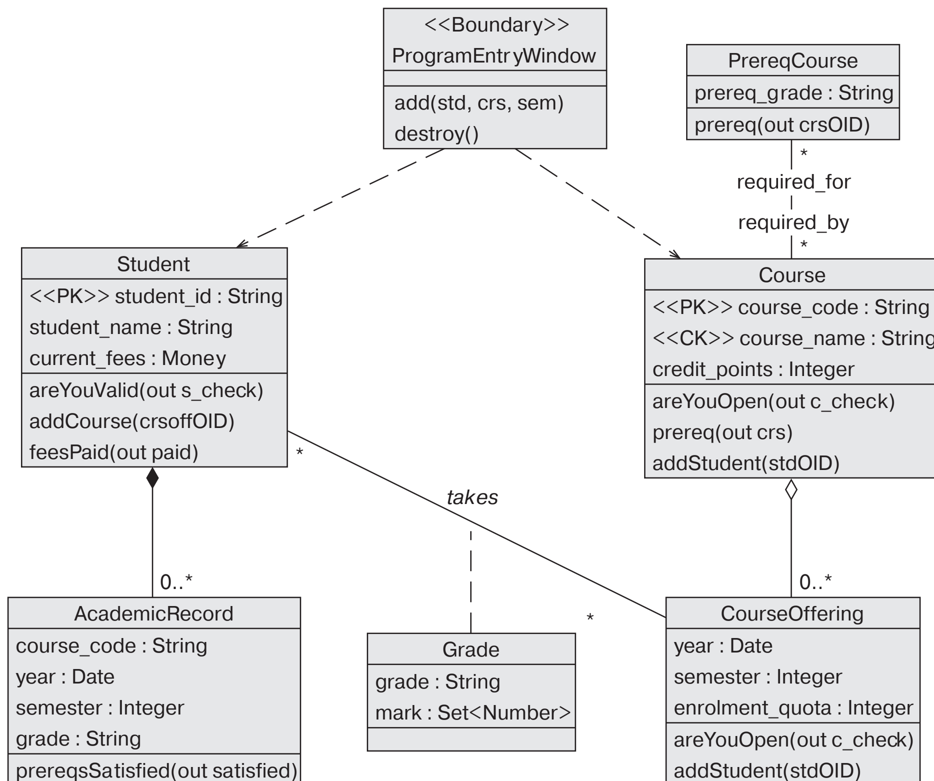

Рис. 6.21. Структурный аспект кооперации (Запись на университетские курсы)

Атрибуты ранее определенных классов не изменились. Новые классы содержат атрибуты, которые должны быть самоочевидными. Основное добавление в детализированную диаграмму классов состоит в определении операций. Операции непосредственно выводятся из диаграммы кооперации, представляющей поведенческий аспект кооперации (см. след. разд.).

Пример 6.3. Запись на университетские курсы

Обратитесь к примеру 4.9 (разд. 4.2.3.3) и примеру 4.17 (разд. 4.3.3.3). Мы рассматриваем прецедент “Enter Program of Study” (“Ввод программы обучения”). Прецедент управляет записью студентов на предлагаемые учебные курсы.

Для целей этого примера мы предполагаем, что прецедент проверяет (прежде, чем студент будет записан), внес ли студент плату и удовлетворяет ли студент предварительным условиям набора на курсы.

Наша задача состоит в том, чтобы расширить диаграмму классов на рис. 4.16 (пример 4.9) таким образом, чтобы смоделировать структурный аспект кооперации, необходимый для описанного сценария.

6.2.4.2. Поведенческий аспект кооперации

На рис. 6.22 представлено решение для примера. По сравнению с диаграммой последовательностей (рис. 4.14) в диаграмму кооперации введено два объекта `anAcademicRecord` и `aPrereqCourse`. Эти объекты необходимы для осуществления проверки оплаты студентом курса и выполнения обязательных условий прежде, чем студент может быть зарегистрирован на предлагаемом курсе (объект `CourseOffering`).

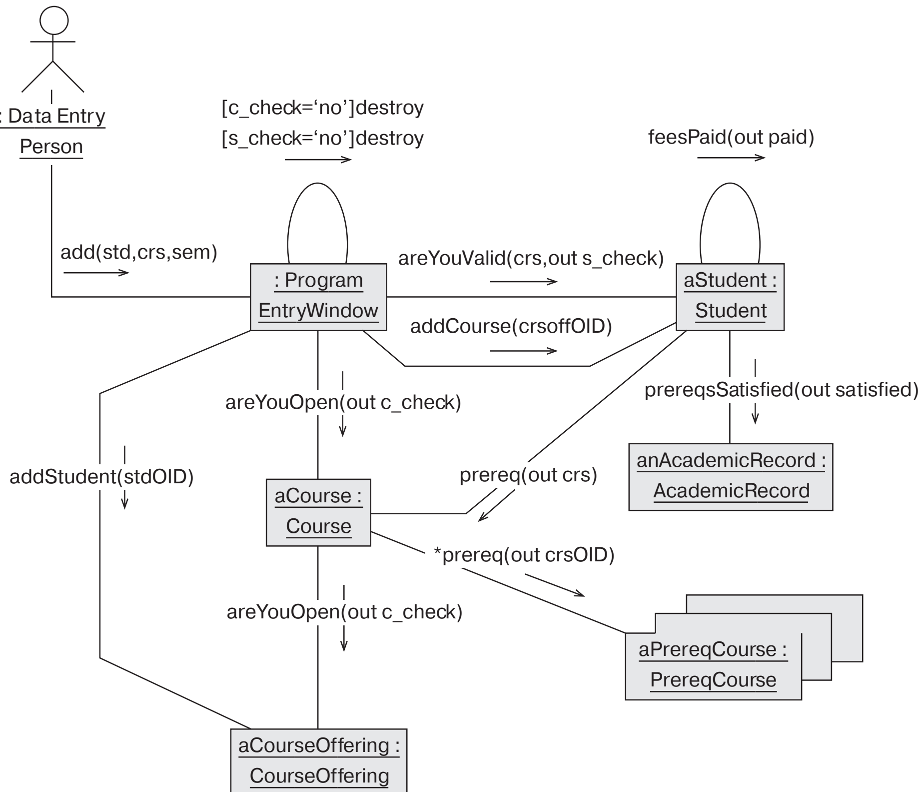

Рис. 6.22. Поведенческий аспект кооперации (Запись на университетские курсы)

Пример 6.4. Запись на университетские курсы

Обратитесь к примеру 4.9 (разд. 4.2.3.3) и примеру 4.17 (разд. 4.3.3.3). Наша задача состоит в создании диаграммы кооперации, представляющей поведенческий аспект кооперативного взаимодействия, как определено в примере 4.9. Диаграмма кооперации должна быть детализацией диаграммы последовательностей на рис. 4.14 (пример 4.17).

6.2.5. Реализация операций

Кооперацию можно также использовать для моделирования реализации более сложных операций [8]. С этой целью можно использовать как структурные, так и поведенческие аспекты.

Модели кооперации стоит применять только для сложных операций, требующих кооперативного взаимодействия нескольких объектов. Более простые операции — ограниченные одним-двумя классами — лучше моделировать с помощью диаграмм видов деятельности, в особенности, если эти операции сильны алгоритмически. Этот же принцип применим в отношении более простых операций, представляющих собой компоненты более сложных операций, которые моделируются с использованием кооперации.

Пример 6.5. Запись на университетские курсы

Обратитесь к примерам 4.18 (разд. 4.3.4.3), 6.3 (разд. 6.2.4.1) и 6.4 (разд. 6.2.4.2). Рассмотрите сообщение `addStudent(stdOID)` на рис 6.22 (пример 6.4).

Наша задача состоит в разработке диаграммы видов деятельности для реализации операции `CourseOffering.addStudent`. Операция должна добавлять студента в список студентов, проходящих данный курс обучения (атрибут `CourseOffering.std`). Атрибут вводится как *шаблон* `list<Student>` (рис. 4.15, пример 4.18).

Прежде, чем студент добавляется к списку слушателей курса, система осуществляет проверку, чтобы определить, по-прежнему ли открыт прием на курс. Это необходимо, даже если аналогичная проверка была выполнена непосредственно перед инициированием события `addStudent(stdOID)` (рис. 6.22). В системах баз данных с высоким уровнем распараллеливания операций статус предлагаемого курса (прием открыт или закрыт) мог за это время измениться.

После добавления студента в список слушателей курса `:CourseOffering` наш алгоритм должен зафиксировать противоположную связь в объекте `:Student`. Чтобы установить обратный указатель из объекта `:Student` на объект `:CourseOffering`, первый должен быть обновлен.

Решение для примера показано на рис. 6.23. Рамки диаграммы видов деятельности ограничены классом `CourseOffering` за исключением необходимости взаимодействия с классом `Student` для поддержания ссылочной целостности между связями.

Наконец, наиболее очевидные операции можно моделировать с использованием *псевдокода* или непосредственно реализовать программно. Программа, которая реализует подобную операцию, по-прежнему может быть включена в UML-модель в качестве примечания или присоединенного документа.

Событие (сообщение) `addStudent` активизирует *деятельность* (метод) `addStudent`. Внутреннее сообщение `areYouOpen` активизирует метод `areYouOpen`. Вид деятельности `areYouOpen` сравнивает квоту с количеством уже записавшихся студентов, чтобы прийти к решению. Проверка осуществляется с помощью внутреннего *исполнительного действия*.

Если решение отрицательно, объект `CourseOffering` переходит в *состояние* `Closed` ([Прием] закрыт). В противном случае он будет находиться в состоянии `Open`. Выполнение диаграммы деятельности должно завершиться при достижении состояния `Closed`.

При нахождении диаграммы (и объекта CourseOffering) в состоянии Open, деятельность addStudent продолжается. Действие add stdOID to CourseOffering.std приводит к новому состоянию Student Added (Студент добавлен [в список курса]). При нахождении в этом состоянии другое действие add crsoffOID to Student.crsoff вызывает деятельность Student.setCrsOff.

Последний вид деятельности идентифицирует новый метод, который должен быть включен в класс Student (рис. 6.21). Он также идентифицирует необходимость в атрибуте crsoff типа list<Course> для класса Student.

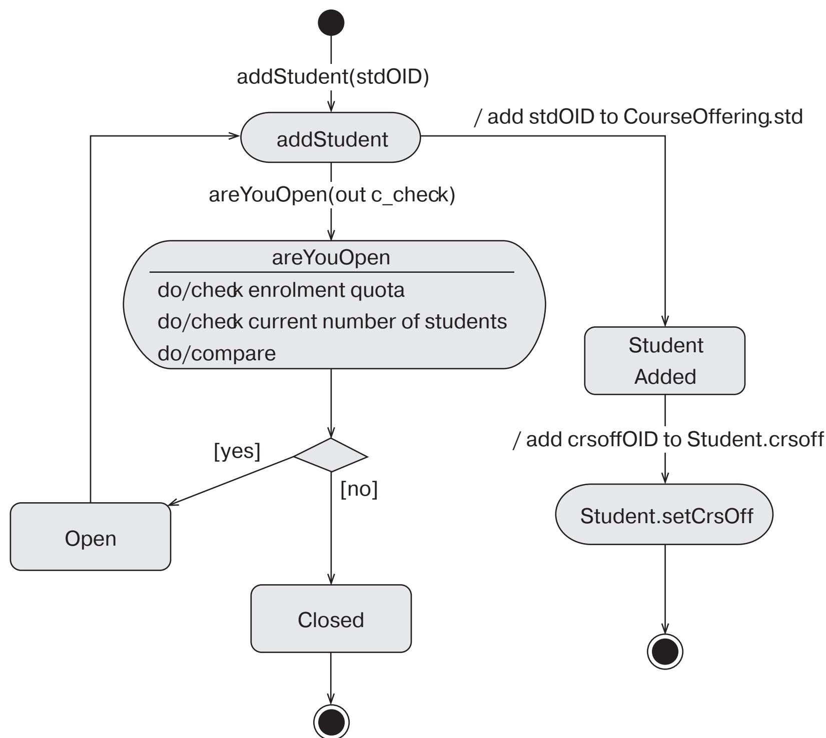

Рис. 6.23. Диаграмма видов деятельности для моделирования реализации операции (Запись на университетские курсы)

6.3. Наставление по проектному моделированию

В главе второй мы представили наставление с руководством по решению для приложения Internet-магазин. Целью этого наставления было помочь читателю освоить основные понятия анализа на основе разбора примеров. В данной главе мы расширим это же наставление, чтобы объединить вместе базовые понятия проектирования.

Наставление имеет двоякое назначение. Во-первых, мы *подкрепим новые концепции* (введенные в главах 5 и 6), используя их применительно к одной проблемной области. Мы обратимся как к архитектурным вопросам (пакеты, компоненты, узлы), так и к вопросам детализированного проектирования (кооперация). Во-вторых, детализиру-

ем модели анализа, разработанные в разд. 2.2. А именно, добавим проектные подробности и расширим существующие модели, чтобы эффективно превратить их в проектные документы, хотя и по-прежнему на относительно высоком уровне абстракции.

Рамки наставления ограничены понятиями, введенными до сих пор в книге. Детализированное проектирование пользовательского интерфейса (глава 7), баз данных (глава 8) и программной логики (глава 9) не рассматриваются.

6.3.1. Проектирование пакетов

В разд. 5.2.3 мы отметили, что пакеты могут объединять в группы классы или другие элементы моделирования, чаще всего — прецеденты. При использовании в *ходе анализа* пакеты обычно применяются для обозначения основных групп прецедентов. *При проектировании* пакеты используются наиболее характерным способом — для группирования классов. Следовательно, можно выделить две основные категории пакетов.

- Пакеты прецедентов.
- Пакеты классов.

Использование моделей пакетов оправдано только при структуризации *больших систем*. Небольшие системы можно понять и управлять ими, не прибегая к помощи пакетов. Для подобных систем обычно достаточно уровня модульности, который обеспечивают прецеденты. Иногда пакеты играют роль своеобразных “архитектурных объемов”, призванных дать возможность приспособиться к ожидаемому росту масштабов системы или зафиксировать тот факт, что первоначальное множество прецедентов и классов по мере продолжения разработки и продвижения детализированного проектирования будет расти.

Пакеты классов, в частности, эволюционируют в ходе проектирования по мере того, как к модели классов добавляются новые пограничные и управляющие классы, а также классы баз данных (напомним читателю, что диаграмма классов, построенная в результате анализа, идентифицирует и связывает между собой *классы-сущности* — к другим типам классов при проектировании фактически не обращаются). Соответственно, модель пакета класса играет ведущую роль при проектировании. Модель *пакета прецедентов* практически не используется за пределами этапов проектирования, поскольку модель пакета классов довольно эффективно заменяет ее.

6.3.1.1. Пакеты прецедентов

Наставление по проектированию: шаг 1 (Internet-магазин)

Обратитесь к наставлению по анализу для приложения *Internet-магазин* (разд. 2.2). Рассмотрите диаграмму прецедентов на рис. 2.24 (разд. 2.2.2.3). Более пристальный взгляд на спецификацию для приложения *Internet-магазин* приводит нас к заключению, что модель прецедентов неполна.

В процессе изучения и документирования уже выявленных прецедентов мы определенно должны обнаружить дополнительные прецеденты. Новые прецеденты могут быть расширением существующих прецедентов или входить в их состав (разд. 4.3.1.2), либо они могут означать прецеденты, которые были упущены из виду при первоначальном анализе требований.

В любом случае мы предполагаем, что новые прецеденты значительно усложнят систему, и поэтому нам кажется целесообразным структурировать существующие прецеденты в виде пакетов. Пакеты станут играть роль “функциональных объемов” для размещения в них существующих и новых прецедентов.

В данном примере мы конструируем диаграмму пакетов прецедентов, которые должны вобрать в себя существующие прецеденты и обеспечить адаптацию к их росту в будущем.

Наше решение для примера представлено на рис. 6.24. Из рисунка видно, что мы выделили пять пакетов и назначили им прецеденты (рис. 2.24). На рисунке также показаны отношения зависимости.

Наверное не удивительно, что пакеты на рис. 6.24 выстроились в соответствии с последовательностью выполнения бизнес-процессов для приложения *Internet-магазин.* Они отражают порядок, в котором клиенты, приобретающие компьютер, используют Web-страницы и формы, отображаемые броузером.

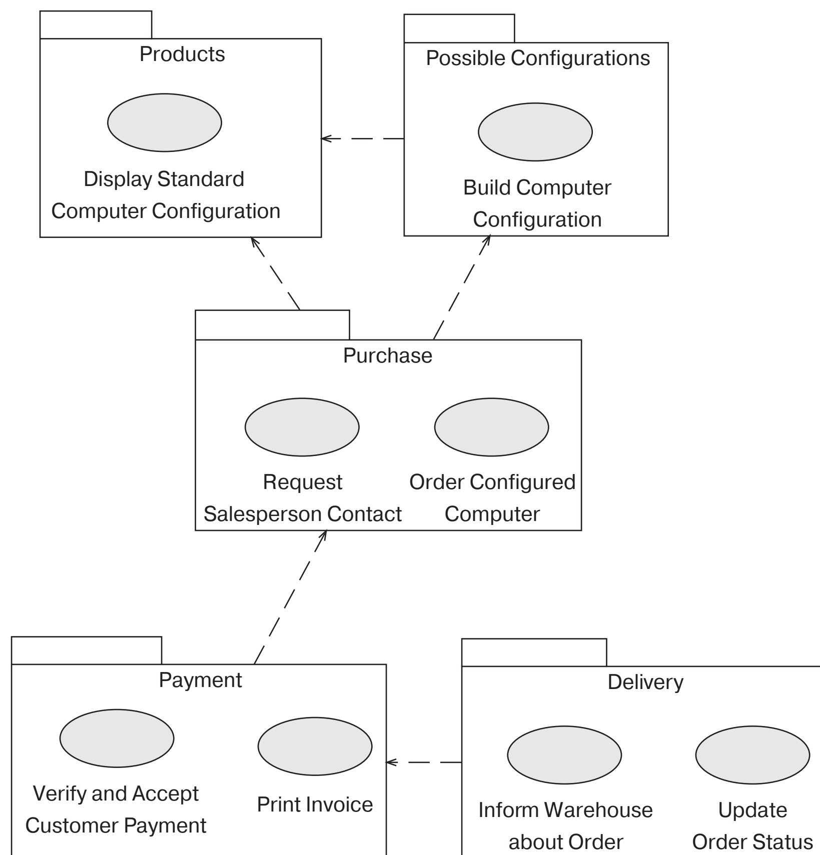

Рис. 6.24. Пакеты прецедентов (Internet-магазин)

6.3.1.2. Пакеты классов

Лучший способ "расправиться" с примером — "сымитировать систему" и представить, что необходимо сделать, чтобы принять заказ клиента на сконфигурированный компьютер. Первый очевидный вывод состоит в том, что система справляется с двумя отдельными функциями: конфигурирование компьютера и ввод заказа. Эти две функции требуют отдельных GUI-окон. В наставлении по анализу мы идентифицировали только два *пограничных класса*, однако, можно не сомневаться, что каждая функция может потребовать сво-

его набора GUI-объектов. Следовательно, можно создать два пакета: ConfigurationGUI (GUI-интерфейс для конфигурирования) и OrderGUI (GUI-интерфейс для заказа).

Наставление по проектированию: шаг 2 (Internet-магазин)

Обратитесь к наставлению по анализу для приложения *Internet-магазин* (разд. 2.2). Вполне естественно, что большинство классов, определенных нами в разд. 2.2, представляют *постоянные объекты базы данных* (“бизнес-объекты”). Более полная модель для системы может потребовать идентификации классов *прикладной программы*. Это можно выполнить во время проектирования кооперативных взаимодействий позже в этом наставлении.

Даже если мы еще не определили классы прикладной программы, можно сделать предположения относительно пакетов, которые должны группировать классы в связанные модули в соответствии с подходом *BCED* (разд. 6.1.3.2). Наша задача в этом примере заключается в том, чтобы продумать возможный состав и структуру пакетов для приложения “Internet-магазин” и основные зависимости между ними.

На “деловой стороне” спектра классов их номенклатура идентифицирована в рамках диаграммы классов (рис. 2.31). Эти *постоянные объекты базы данных* можно естественным образом сгруппировать в виде трех *пакетов сущностей*: Customers, Computers и Orders (последний может также включать классы Invoice и Payment).

Теперь все еще недостает пакетов, которые могли бы связать воедино пограничные классы и классы-сущности, т.е. управляющих пакетов. Нам требуются пакеты, содержащие *управляющие классы*, отвечающие за выполнение логики приложения, а также пакет для рационального конфигурирования компьютеров и подсчета цены конфигурации. Назовем такой пакет ConfigureProcess. Потребуется и пакет, отвечающий за ввод и учет заказов — пакет OrderPlacement.

Три пакета для сущностей (Customers, Computers, и Orders) представляют структуры для постоянных классов базы данных, находящиеся в памяти машины во время выполнения программы. Они обеспечивают временное представление в других обстоятельствах постоянных классов, хранимых в базе данных. В каждой точке выполнения программы объекты класса-сущности содержат только фрагмент содержимого базы данных, обеспечивают объектно-ориентированный образ структур данных, которые обычно хранятся в структурах баз данных, отличных от объектно-ориентированных, а именно, как реляционные таблицы. Следовательно, существует потребность в классах, устанавливающих интерфейс от классов-сущностей к базе данных — требуется предусмотреть один или несколько *пакетов баз данных*.

Основной пакет баз данных можно назвать пакетом CRUD — **C**reate–**R**ead–**U**pdate–**D**elete (создать–читать–обновить–удалить (разд. 4.3.4.1)). Пакет CRUD выступает посредником между классами сущностей и таблицами базы данных всякий раз, когда приложению требуется доступ или модификация содержимого базы данных.

Пакет CRUD зависит от двух других пакетов баз данных под названием Connection (Соединение) и Schema (Схема). Классы пакета Connection отвечают за установление соединения, авторизацию и транзакции. Пакет Schema содержит текущую информацию об объектах схемы базы данных — таблицах, столбцах, хранимых процедурах и т.д. Приложение может порождать объекты пакета Schema при запуске, так что оно в состоянии проверить, что объекты базы данных существуют в базе, прежде чем осуществить попытку фактического доступа к базе данных

(например, перед вызовом хранимой процедуры приложение может проверить, используя находящиеся в памяти объекты схемы, что хранимая процедура по-прежнему существует).

На рис. 6.25 представлены описанные выше пакеты. На нем также показаны основные зависимости. Зависимости носят первоначальный и небесспорный характер. В отсутствие знания всех классов и связей взаимодействия между ними между пакетами можно установить только гипотетические зависимости. Основной принцип состоит в том, что пограничные пакеты зависят от управляющих пакетов, а те, в свою очередь, — от пакетов сущностей. Наконец, пакеты сущностей зависят от пакетов баз данных.

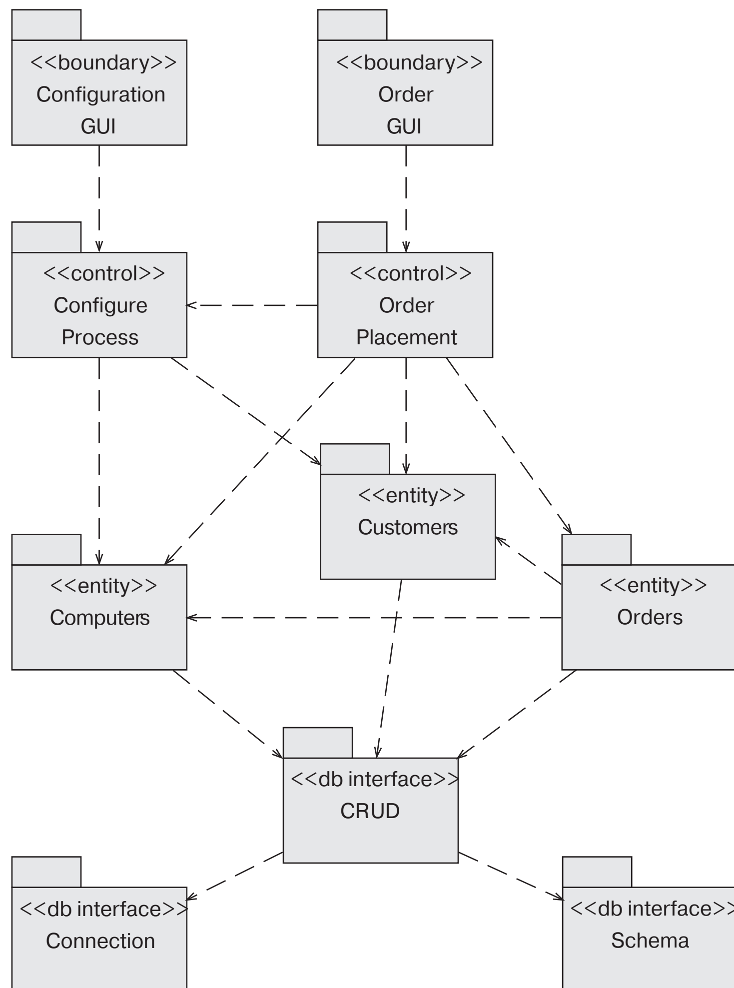

Рис. 6.25. Пакеты классов (Internet-магазин)

Заметим, что зависимости между пакетами *нетранзитивны* [25]. Например, на рис. 6.25 изменения в рамках пакета `Customers` могут влечь за собой изменения в `ConfigureProcess` и `OrderPlacement`, но не в `ConfigurationGUI` и `OrderGUI`. Нетранзитивность, конечно, необходима для достижения меньшей сложности в иерархических или квазииерархических структурах (разд. 5.2.2).

Еще одним фактором сложности являются *циклы* в структурах зависимости. В принципе, циклов следует избегать [48]. На практике иногда циклов очень трудно избежать, в особенности между пакетами на одном и том же уровне иерархии [25].

6.3.2. Проектирование компонент

Компоненты — это физические части системы. Следовательно, проектирование компонент нельзя отделить от платформы реализации. Приложение *Internet-магазин* — это Web-приложение с сервером баз данных. Другие аспекты платформы реализации нами на определялись.

Проектирование компонент требует знания платформы реализации. Поскольку мы не намерены в этой книге выступать в защиту каких-либо конкретных решений или поставщиков ПО, все предположения относительно платформы реализации будут строиться на достаточно общем уровне.

6.3.2.1. Реализация Web-приложений

"Web-приложение — это Web-система, позволяющая ее пользователям работать в соответствии с заложенной в нее бизнес-логикой с помощью Web-броузера [15]. Программа, поддерживающая бизнес-логику, может находиться на сервере и/или на клиенте. Следовательно, Web-приложение — не что иное, как разновидность системы клиент/сервер (разд. 6.1.1) с Web-узлом."

Броузер Internet-клиента отображает Web-страницы на экране компьютера. Web-сервер доставляет Web-страницы броузеру. Web-страницы могут быть статическими (неизменяемыми) или динамическими. Web-страница может представлять форму, заполняемую пользователем. Чтобы пользователь мог одновременно просматривать несколько Web-страниц, "драгоценное пространство" экрана разделяется приложением на *фреймы*.

Чтобы управлять логикой приложения и отслеживать состояние приложения, Web-приложение может включать *сервер приложений* (разд. 6.1.2). В приложении, подобном *Internet-магазину*, мониторинг состояния представляет собой важный вид деятельности, связанный с отслеживанием действий интерактивных пользователей, например, конфигураций компьютеров, которые они запрашивают. Клиент может решить приобрести определенную конфигурацию в любой момент в ходе *интерактивного сеанса*, и система должна связать заказ на покупку с конфигурацией.

Обычный метод отслеживания состояния состоит в хранении в броузере так называемых *cookie* — коротких символьных строк, представляющих состояние интерактивного пользователя. Поскольку количество интерактивных пользователей, за которыми необходимо следить с помощью Web-сервера или сервера приложений, произвольно велико, на *перерыв* в работе пользователей в *течение одного сеанса* может быть наложено ограничение. Если пользователь не активизировался в течение 15 минут (обычное время перерыва), сервер отсоединяется от клиента. Сами cookie могут удаляться или не удаляться с клиентской машины.

Чтобы придать страницам, отображаемым на клиентской машине, динамичность, используются сценарии или апплеты. *Сценарий* (или *скрипт* (*script*)) (например, написанный на языке JavaScript) представляет собой программу, выполняемую броузером в режиме интерпретации. *Апплет* (*applet*) — это скомпилированная компонента, которая выполняется в контексте броузера, однако, имеет лишь ограниченный доступ к другим ресурсам клиентской машины (из соображений безопасности).

Web-страница может также включать сценарии, выполняемые сервером. Подобная страница называется *серверной страницей* (*server page*). Серверная страница имеет доступ ко всем ресурсам сервера баз данных. Серверные страницы управляют клиентскими сеансами, размещают cookies в среде броузера и строят клиентские страницы (т.е. строят страничные документы из серверных бизнес-объектов и отправляют их назад клиенту).

Чтобы обеспечить доступ сценариев, содержащихся в серверных страницах, к базам данных, используются стандартные *библиотеки доступа к данным.* К типичным технологиям, позволяющим реализовать эту возможность, относятся *ODBC* (Open Database Connectivity — открытый интерфейс доступа к базам данных), *JDBC* (Java Database Connectivity — интерфейс доступа к базам данных Java-приложений), *RDO* (Remote Data Objects — интерфейс доступа к удаленным объектам), *ADO* (ActiveX Data Object — набор высокоуровневых интерфейсов, позволяющих разработчикам обращаться к данным на любом языке программирования на основе ActiveX). В ситуации, когда организация ориентируется на стандарты определенной СУБД, более непосредственный доступ к базам данных могут обеспечить вызовы низкоуровневых функций библиотек баз данных (DBLib — Database Library).

Технологией, дающей возможность функционировать *Web-серверу*, являются содержащие сценарии страницы, написанные на языке HTML (HyperText Markup Language — язык гипертекстовой разметки документов), — активные серверные страницы (Active Server Pages (ASP)) или серверные страницы Java (Java Server Pages (JSP)). Для создания *Web-страниц* можно использовать технологию написания клиентских сценариев (JavaScript или VBScript), документов XML (eXtensible Markup Language), Java-апплетов, управляющих элементов JavaBean или ActiveX

Для получения Web-страниц с Web-сервера клиенты используют протокол HTTP (Hypertext Transfer Protocol — протокол передачи гипертекстовых файлов). Страница может содержать сценарий или скомпилированные и непосредственно выполняемые модули DLL (Dynamic Link Library — динамически компонуемая библиотека), например, ISAPI (Internet Server Application Programming Interface — интерфейс прикладного программирования Internet-сервера), NSAPI (Netscape Server Application Programming Interface — интерфейс прикладного программирования сервера Netscape), CGI (Common Gateway Interface — Общий шлюзовый интерфейс) или Java-сервлеты [15].

Cookie играют роль примитивного механизма поддержки соединения между клиентом и сервером в системе, которая иначе называется *Internet-системой без установления соединения.* Более сложный механизм соединения клиента с сервером превращает Internet в *распределенную объектную систему.* В распределенной объектной системе объекты идентифицируются с помощью уникальных OID (разд. 2.1.1.3) и взаимодействуют за счет получения OID друг друга. Главными механизмами при этом выступают техно-

логии CORBA, DCOM и EJB (разд. 1.1.1). При использовании данных технологий объекты могут взаимодействовать без использования протокола HTTP или Web-сервера в качестве посредника [15].

6.3.2.2. Диаграмма компонент

Наставление по проектированию: шаг 3 (Internet-магазин)

Обратитесь к приведенному выше разделу 6.3.2.1 и предложите диаграмму компонент для приложения *Internet-магазин* (разд. 2.2). Исходите из предположения, что компонента — это цельный функциональный модуль с четким интерфейсом, так что она вполне может выступать в качестве заменяемой части системы. Поскольку платформа реализации для приложения *Internet-магазин* не определена, идентификация более мелких компонент (таких как библиотеки, хранимые процедуры и т.д.) на этом этапе невозможна.

Один из способов взяться за этот шаг наставления — рассмотреть типичную последовательность доступа к Web-страницам, интерактивного со стороны пользователя, желающего приобрести компьютер. Соответствующие руководящие указания можно получить из анализа прецедентов (разд. 2.2.1) и пакетов прецедентов (разд. 6.3.1.1).

Первая Web-страница, которую может посетить интерактивный пользователь, — это Web-страница поставщика, на которой перечислены группы изделий (такие как серверы, настольные системы, портативные компьютеры), выделены последние предложения и скидки, и приводятся ссылки на Web-страницы, на которых представлены перечни изделий и дано краткое описание каждого изделия. Краткое описание включает цену для стандартной конфигурации изделия. Эта часть системы посвящена рекламе товаров интерактивным покупателям. Это единый функциональный блок, который может образовать компоненту под названием `ProductList` (Перечень изделий).

На следующем шаге клиент может справиться относительно технических спецификаций для выбранного изделия. Сюда входит визуальное отображение изделия под разными углами зрения. Это функционально законченная Web-страница, которая является неплохим претендентом на следующую компоненту под названием `ProductDisplay` (Отображение изделия).

Если предположить, что описанные выше Web-страницы привлекли внимание клиента к продукту, он может сформулировать запрос в отношении различных конфигураций изделия, удовлетворяющих его специфические нужды и требования по расходам. Подобный запрос можно удовлетворить с помощью динамических Web-страниц, позволяющих в интерактивном режиме построить конфигурацию и отобразить готовое изделие вместе с рассчитанной ценой конфигурации. Это следующий неплохой претендент на компоненту. Назовем ее `Configuration` (Конфигурация).

Клиенту, решившему купить изделие, предоставляется форма заказа на покупку. В нее должны быть, в частности, введены такие детализированные данные, как имя клиента, адрес, по которому следует доставить изделие и оплатить счет-фактуру. Также выбирается способ оплаты, и соответствующие подробности пересылаются с использованием некоторого безопасного протокола передачи данных. Это четвертая компонента — `Purchase` (Покупка).

Последняя компонента, которую мы выделяем в нашем наставлении, должна обрабатывать выполнение и отслеживание заказа. С точки зрения клиента она должна

обеспечивать возможность просмотра состояния заказа с помощью Web-страницы (после ввода номера клиента и номера заказа). Эту компоненту можно назвать `OrderTracking` (Отслеживание заказа).

Рассмотренные выше пять компонент показаны на рис. 6.26. Интересно отметить сходство этой диаграммы с пакетами прецедентов на рис. 6.24, что неудивительно, поскольку пакеты прецедентов и компоненты представляют собой функциональные модули с четкими границами. Пакеты прецедентов —логические функциональные блоки — не оказывают влияния на способ окончательного построения системы, а если система проста, то нет даже нужды в их конструировании. Компоненты — это реальные независимо развертываемые физически блоки — требуют тщательного проектирования и реализации даже для небольших систем.

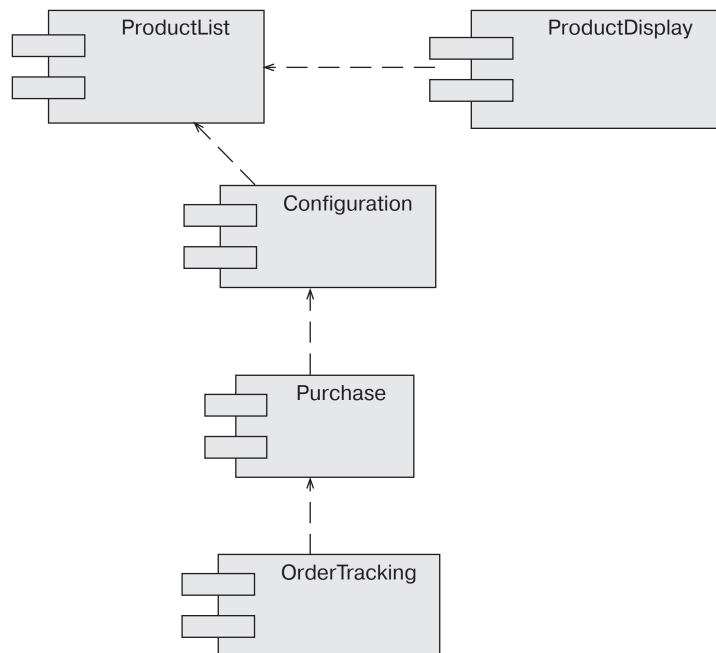

Рис. 6.26. Диаграмма компонент (Internet-магазин)

6.3.3. Проект развертывания

Характер Internet-систем без установления прямого соединения (разд. 6.3.2.1) делает *развертывание* (*deployment*) Web-приложений значительно более сложной задачей, чем развертывание приложений баз данных в архитектуре клиент/сервер. Чтобы приступить к развертыванию, требуется установить Web-сервер в качестве пункта маршрутизации между всеми броузерами клиентов и базой данных.

Если проблема управления сеансами не может быть удовлетворительно решена с помощью технологии *cookie*, необходимо привлекать на помощь технологию *распределенных объектов*. Развертывание распределенных объектов может потребовать размещения отдельных архитектурных элементов — сервера приложений — между Web-сервером и сервером баз данных.

Проект развертывания должен обращаться к вопросам безопасности. Безопасная передача данных и протоколы шифрования — еще один аспект требований со стороны проекта развертывания. Кроме того, необходимо тщательное планирование с учетом сетевой загрузки, Internet-соединений, резервных копий и т.д.

6.3.3.1. Развертывание Web-приложений

Архитектура развертывания, способная поддерживать более сложные Web-приложения, включает четыре звена вычислительных узлов.

1. Клиентский Web-броузер.
2. Web-сервер.
3. Сервер приложений.
4. Сервер баз данных.

Броузер *клиентского узла* можно использовать для отображения статических или динамических Web-страниц. Страницы, включающие сценарии и апплеты, можно загружать и выполнять в рамках броузера. Клиентский броузер можно оснастить дополнительными функциональными возможностями, такими как элементы управления ActiveX или JavaBeans. Выполнение программы приложения на клиентской машине, но вне броузера, может удовлетворить другие требования к GUI-интерфейсу.

Web-сервер обрабатывает запросы на страницы, поступающие от броузера, и динамически генерирует страницы и программный код для выполнения и отображения на клиенте. Web-сервер также обеспечивает настройку и параметризацию сеансов работы пользователя.

Сервер приложений необходим в том случае, когда в реализации используются распределенные объекты. Он управляет бизнес-логикой. Бизнес-компоненты публикуют свои интерфейсы для других узлов через интерфейсы компонент, такие как CORBA, DCOM или EJB.

Бизнес-компоненты инкапсулируют постоянные объекты, хранимые в базе данных, чаще всего в реляционной базе. Они взаимодействуют с сервером баз данных через протоколы связи с базами данных, например, такими как JDBC или ODBC. Узел базы данных обеспечивает масштабируемое хранилище данных и многопользовательский доступ к нему.

6.3.3.2. Развертывание диаграмм

Наставление по проектированию: шаг 4 (Internet-магазин)

Обратитесь к приведенному выше разделу 6.3.3.1 и предложите диаграмму развертывания для приложения *Internet-магазин*. В частности, рассмотрите вопрос о том, необходимо ли использование для приложения *Internet-магазин* сервера приложений.

Как показано на рис. 6.27, приложение *Internet-магазин* можно развернуть без отдельного сервера приложений. Программы, содержащиеся в Web-страницах, могут выполняться Web-сервером. Потенциальным преимуществом сервера приложений является то, что хранимые на нем компоненты приложения могут повторно использоваться другими Web-приложениями для вызова той же бизнес-логики. Однако *Internet-магазин* — автономная система и вряд ли можно указать другие Web-приложения, которые могли бы извлечь пользу из ее бизнес-логики.

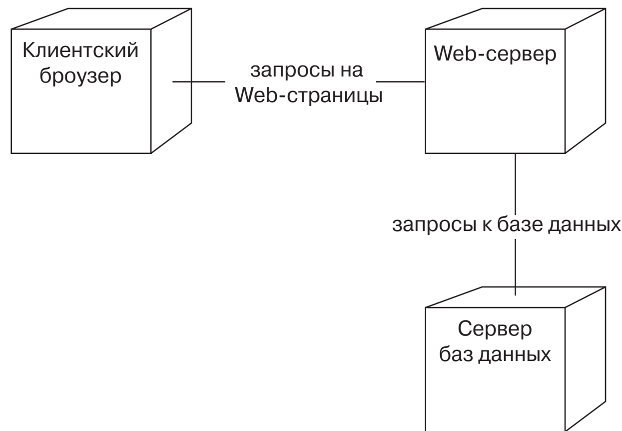

Рис. 6.27. Диаграмма развертывания (Internet-магазин)

6.3.4. Проектирование кооперативных взаимодействий

В начале этой главы объяснялось, что системное проектирование разделяется на архитектурное проектирование и детализированное проектирование. Проектирование пакетов, компонент и узлов относится к *архитектурному проектированию. Детализированное проектирование* сосредоточивается на кооперации (разд. 6.2).

Кооперативные взаимодействия определяют реализацию прецедентов и реализацию более сложных операций (простые операции не следует моделировать как кооперативные). Проектирование кооперативных взаимодействий неизменно приводит к *уточнению* (модификации и расширению) существующих диаграмм классов и выработке новых диаграмм кооперации (либо к детализации существующих диаграмм последовательностей). Другие типы диаграмм, в частности диаграммы состояний, также могут потребовать разработки или уточнения.

6.3.4.1. Детализация прецедентов

Важным побочным результатом (или даже предпосылкой) проектирования кооперации является *уточнение прецедентов.* Прецеденты, зафиксированные в виде документа при проведении анализа требований, как правило, не достаточно детализированы для проектирования кооперации. Спецификации прецедентов необходимо уточнить в проектной документации. Прецеденты, представленные более подробными спецификациями, должны включать требования системного уровня, оставаясь, в то же время, на точке зрения субъектов.

Если на этапе анализа на управлении требованиями (разд. 3.4) концентрировались не слишком большие усилия, то теперь наступает последняя возможность достичь большей формализации и строгости в их определении. Требования следует подвергнуть тщательной классификации и пронумеровать их. Чтобы обеспечить надлежащее управление изменениями и прослеживаемостью требований, их следует поместить в репозиторий CASE-системы. (Значение управления требованиями выходит далеко за рамки анализа требований на ранних фазах проектирования. Эти вопросы рассматриваются в главе 10.)

6.3.4.1.1. Нумерация и структурирование требований

Требования следует пронумеровать (разд. 3.4.1), структурировать (разд. 3.4.2). Названные виды деятельности лучше всего осуществлять с помощью CASE-средств. Попытки отслеживать изменения требований вручную обречены на провал. В то же время CASE-средства значительно облегчают процесс перенумерации и реструктуризации требований.

На рис. 6.28 показана часть документа-спецификации прецедентов с пронумерованными требованиями. Требования пронумерованы с использованием десятичной системы Девея (Dewey), при этом каждое требование обозначено префиксом UC (Use Case— прецедент). Префикс полезен в том случае, когда документ описания прецедентов содержит более одного типа требований. Заметим, что требования заключены в квадратные скобки, подчеркнуты и отображаются в зеленом цвете (в вопросе цвета вы уж нам поверьте).

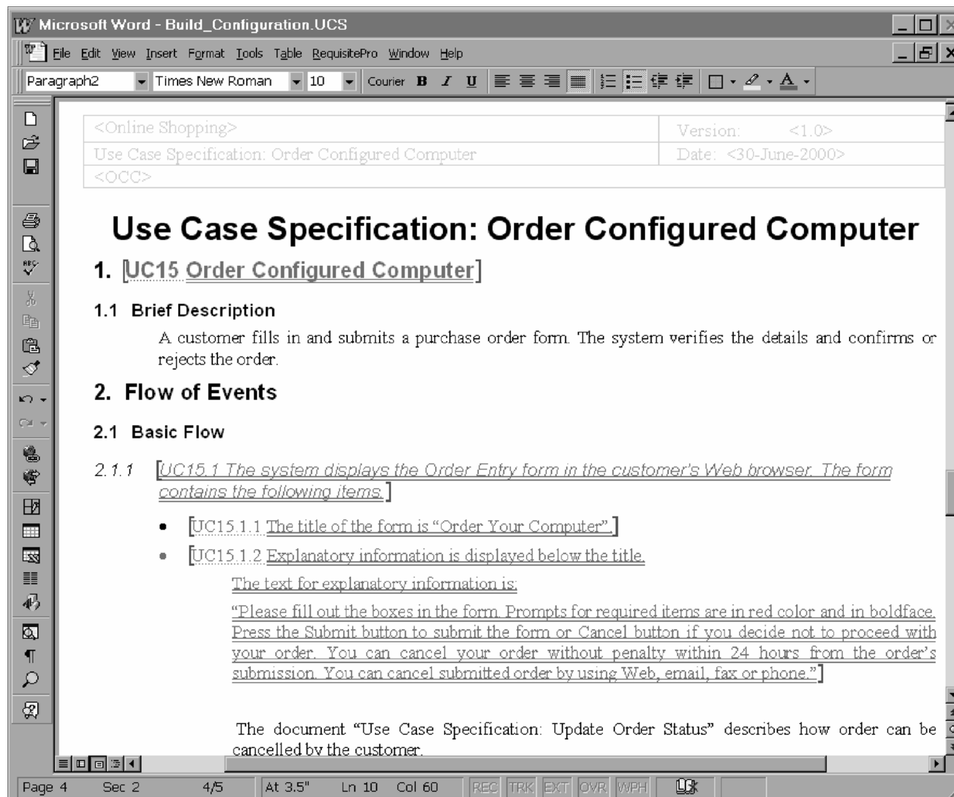

Рис. 6.28. Фрагмент документа описания прецедентов, поддерживаемого с помощью CASE-системы

Рис. 6.29. Управление требованиями с помощью CASE-системы

6.3.4.1.2. Проектная документация по прецедентам

Наставление по проектированию: шаг 5 (Internet-магазин)

Обратитесь к документу по анализу прецедентов для приложения *Internet-магазин*, приведенному в табл. 2.2 разд. 2.2.2.4. Этот документ описывает прецедент “Order Configured Computer” (“Заказ сконфигурированного компьютера”). Данный документ недостаточно подробный для разработки проекта кооперации объектов.

Наша цель на этом шаге наставления заключается в уточнении документа описания прецедентов таким образом, чтобы он представлял собой спецификацию прецедентов проектного уровня. Детализированный документ должен быть организован аналогично тому, как показано на рис. 6.28. Фактически, рис. 6.28 и 6.29 представляют собой часть решения для данного шага наставления.

Соответствующий текст документа описания прецедента приведен ниже. Заметим, что этот документ можно отпечатать в формате, отличном от приведенного здесь. Например, можно подавить отображение и печать номеров прецедентов.

(Поскольку приводимый ниже документ-спецификация носит иллюстративный характер и связь описанных в ней действий клиента и функций системы с табл. 2.2 разд. 2.2.2.4 легко пролеживается, для удобства читателей она полностью переведена на русский язык. *Прим. ред.*)

Спецификация прецедента: “Заказ сконфигурирова н-ного компьютера”

1. [UC15 Заказ сконфигурированного компь ютера]

1.1 Краткое описание

Клиент заполняет и отправляет форму заказа на покупку. Система проверяет детальную информацию и подтверждает или отвергает заказ.

2. Поток событий

2.1 Основной поток

2.1.1 [UC15.1 Система отображает форму ввода заказа в окне Ввод заказа броузера клиента. Форма состоит из следующих пунктов.]

- [UC15.1.1 Форма имеет заголовок `Заказывайте компьютер`.]
- [UC15.1.2 Под заголовком отображается поясняющая информация.]

  Приводим текст поясняющей информации:

  “Заполните, пожалуйста, поля формы. Приглашения для ввода требуемых ответов показаны красным полужирным шрифтом. Для отправки формы нажмите копку `Отправить`, если же вы решили не продолжать заполнение. Вы можете отказаться от своего заказа без всякого штрафа в течение 24-х часов с момента отправки заказа. Вы можете отказаться от отправленного заказа, прибегнув к помощи Web, электронной почты, факса или телефона.”]

  Описание того, каким образом клиент может отказаться от заказа, содержится в документе “Спецификация прецедента: Обновление статуса заказа”.

- [UC15.1.3 Реквизиты поставки].

  [UC15.1.3.1 К требуемым реквизитам доставки относятся: имя, страна, город, улица, курьерские направления.]

  [UC15.1.3.2 К дополнительным реквизитам доставки относятся: ригород, штат, почтовый код.]

- [UC15.1.4 Контактная информация, отличная от той, которая предоставляется как реквизиты доставки.]

  [UC15.1.4.1 Предпочтительные средства для контактов: электронная почта, телефон, факс, почта, курьерская почта.]

  [UC15.1.4.2 В качестве обязательной контактной информации следует указать один из следующих реквизитов: электронная почта, телефон, факс.]

  UC15.1.4.3 В качестве дополнительной контактной

информации можно указать два из трех контактных реквизита, перечисленные как обязательные, и почтовый адрес (если он отличается от указанного в реквизитах доставки).]

- [UC15.1.5 Адрес доставки счет-фактуры, если он отличается от реквизитов доставки.]
- [UC15.1.6 Способ оплаты.]

  [UC15.1.6.1 Клиент может выбрать в качестве метода оплаты чек или кредитную карточку.]

  [UC15.1.6.2 При оплате с помощью чека система предоставляет подробную информацию, кому должен быть оплачен чек и покакому адресу его следует доставить. Она также информирует клиента, что на клиринг чека после получения может потребоваться до трех дней.]

  [UC15.1.6.3 При оплате с помощью кредитной карточки система отображает реквизиты, которые должен заполнить клиент. К этим реквизитам относятся: перечень допустимых кредитных карточек, номер кредитной карточки, срок действия кредитной карточки.]
- [UC15.1.7 Имя торгового представителя, если оно известно клиенту по прошлым сделкам.]
- [UC15.1.8 Две командные кнопки: `Отправить` и `Отмена`.]

2.1.1 [UC15.2 Система предлагает клиенту ввести подробности, касающиеся заказа, поместив указатель мыши на первое редактируемое поле (реквизит имени).] [UC15.3 Система позволяет вводить информацию в произвольном порядке.]

- [UC15.4 Если клиент не отправляет и не аннулирует форму в течение 15 минут, выполняется альтернативный поток ”Клиент не активен”).]

2.1.3 [UC15.5 Если клиент нажимает кнопку **Отправить**, и вся необходимая информация предоставлена, форма заказа отправляется на Web-сервер. Web-сервер связывается с сервером баз данных для того, чтобы поместить заказ в базу данных.] [UC15.6 Сервер баз данных присваивает заказу на покупку уникальный номер заказа и учетный номер клиента.]

- [UC15.7 Если сервер баз данных не в состоянии создать и запомнить заказ, выполняется альтернативный поток “Исключительная ситуация для БД”.]
- [UC15.8 Если клиент отправляет заказ с неполной информацией, выполняется альтернативный поток “Неполная информация”.]

2.1.4 [UC15.9 Если клиент в качестве предпочтительного средства коммуникации предоставляет адрес электронной почты, система отправляет клиенту заказ и номера клиента вместе со всеми деталями, касающимися заказа, и подтверждением принятия заказа. Прецедент

завершается.] [UC15.10 В противном случае, детали, касающиеся заказа, отправляются клиенту по почте и прецедент также завершается.]

2.1.5 [UC15.11 Если клиент нажимает кнопку Отмена, выполняется альтернативный поток “Отмена”.]

2.2 *Альтернативные потоки*

2.2.1 Клиент неактивен

[UC15.4.1 Если клиент не проявляет активности в течение более чем 15 минут, система закрывает соединение с броузером. Прецедент завершается.]

2.2.2 Исключительная ситуация для БД

[UC15.7.1 Если база данных возбуждает исключительную ситуацию, система интерпретирует ее и информирует клиента о характере ошибки. Если клиент разорвал соединение, система отправляет сообщение об ошибке по электронной почте клиенту и продавцу. Прецедент завершается.]

Если с клиентом нет возможности связаться посредством Internet или электронной почты, продавец должен связаться с клиентом с помощью других средств.

2.2.3 Неполная информация

[UC15.8.1 Если клиент не заполнил все необходимые реквизиты, система предлагает клиенту предоставить пропущенную информацию. Список пропущенных реквизитов отображается. Прецедент продолжается.]

2.2.4 Отмена

[UC15.11.1 Если клиент нажал кнопку Отмена, заполненные поля формы очищаются. Прецедент продолжается.]

3. Предусловия

3.1 Клиент указывает броузеру Internet на Web-страницу системы. Страница отображает детальную информацию о конфигурации компьютера вместе с ее ценой. Клиент нажимает кнопку Покупка.

3.2 Клиент нажимает кнопку Покупка в пределах 15 минут с момента запроса на построение и отображение на странице броузера последней конфигурации компьютера.

4. Постусловия

4.1 Если отправка заказа клиента прошла успешно, заказ на покупку записывается в базу данных. В противном случае состояние системы не изменяется.

6.3.4.2. Структура кооперативного взаимодействия

Диаграмма классов, разработанная в ходе анализа, определяет *постоянные классы базы данных. Классы прикладной программы* фактически нельзя определить до начала проектирования. Следовательно, диаграмму классов, определяющую структуру не-

которой кооперации, необходимо разрабатывать практически "с нуля". Классы, участвующие в кооперации, разрабатываются для того, чтобы осуществить привязку к выбранной для приложения технологии, обеспечивающей его функционирование (разд. 6.3.2.1).

Web-страница может содержать сценарии, выполняемые на сервере или интерпретируемые на клиентской машине броузером, или же сценарии обоих типов. Проектирование кооперативного взаимодействия для Web-страницы — непростая проблема. Технология, обеспечивающая функционирование, может не поддерживать надлежащим образом объектно-ориентированного взгляда на мир. В некоторых случаях работа всего приложения может определяться одной Web-страницей, наполненной объектами и апплетами.

Что касается более сложных Web-приложений, для них должна существовать возможность разделения страниц на клиентские, серверные. С точки зрения подхода *BCED* (разделы 6.1.3.2 и 6.3.1.2), клиентские страницы, представленные в формате HTML, соответствуют пограничным классам. Серверные страницы могут, с одной стороны, обладать ассоциативными связями с клиентскими страницами, а, с другой стороны, с остальными ресурсами Web-сервера. Серверные страницы соответствуют управляющим классам в иерархии *BCED* [15].

Наставление по проектированию: шаг 6 (Internet-магазин)

Обратитесь к проектному документу по прецеденту "Order Configured Computer", представленному в разделе 6.3.4.1.2. Основываясь на потоке событий, представленном в этом документе, нам требуется создать проект кооперации, отражающий ее структурный аспект, который реализует прецедент "Заказ сконфигурированного компьютера".

Даже для такого простого прецедента детализированное проектирование кооперации выходит за рамки наставления. Поэтому мы сделаем некоторые упрощения. Нет необходимости обрабатывать все детали, касающиеся формы и схемы формы заказа (см. главу 7). Аналогично, с противоположной стороны, нет необходимости принимать решение в отношении интерфейса между приложением и базой данных (т.е. мы не касаемся вопросов получения и запоминания информации в базе данных) (см. главу 8).

В данном случае мы применяем подход *BCED* (разд. 6.3.1.2). Всем предложенным классам должны быть присвоены префиксы b, c, e, или d для обозначения их принадлежности соответствующему уровню пакета *BCED*. Классы также должны быть отнесены к одному из следующих стереотипов: «`client page`» (клиентская страница), «`form`» (форма), «`server page`» (серверная страница), «`entity`» (сущность), «`db interface`» (интерфейс с БД).

Следует определить все отношения *ассоциации* и *агрегации* между классами. Также следует показать *отношения материализации*, чтобы обозначить поток сообщений материализованным объектам (отношение материализации изображается в языке UML с помощью пунктирных линий со стрелками, направленными от отправителя сообщения к классу, который запрашивает материализацию нового объекта). Если объект материализован в результате инициированного пользователем события (например, пользователь нажимает командную кнопку), событие можно изобразить на линии материализации.

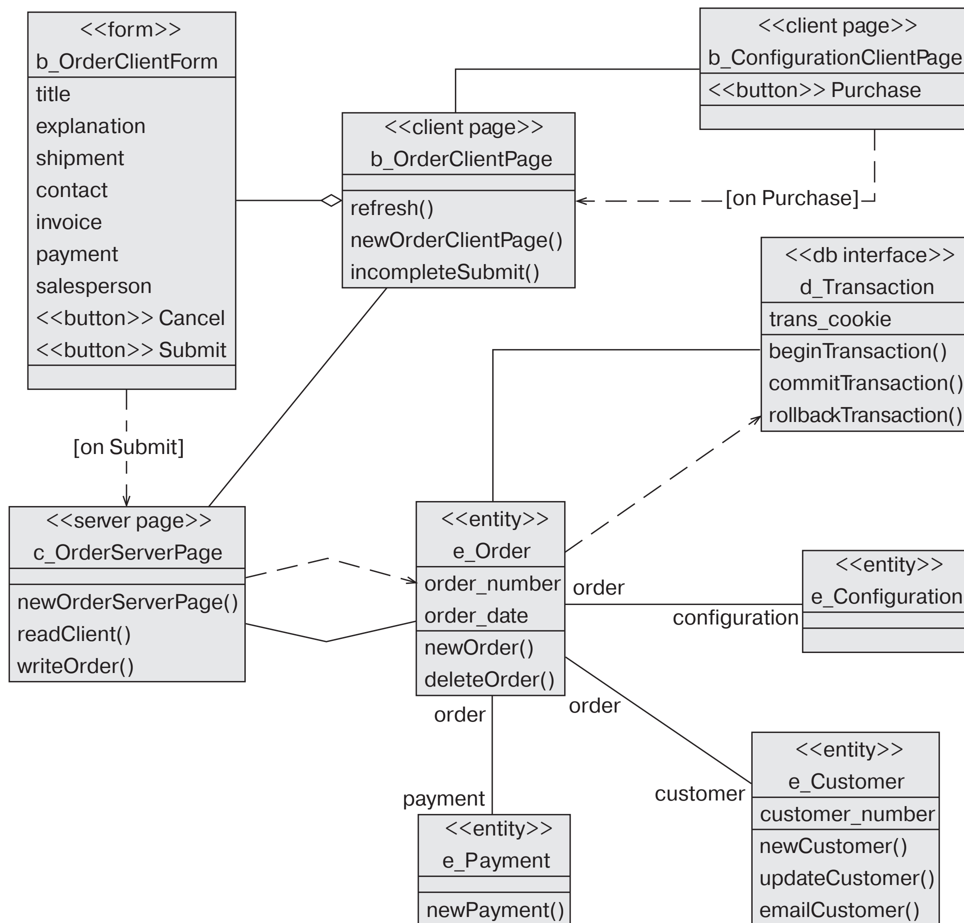

Рис. 6.30. Структурная сторона кооперации для прецедента "Order Configured Computer" (Internet-магазин)

На рис. 6.30 приведена диаграмма классов, представляющая структурную сторону кооперации для прецедента "Order Configured Computer". Поскольку этот прецедент составляет только часть приложения, а также с целью моделирования предусловий и постусловий прецедента на рисунке показаны два класса вместе с их связями, принадлежащие другому прецеденту "Build Computer Configuration" ("Составление конфигурации компьютера"). Это классы `b_ConfigurationClientPage` (Клиентская страница конфигурации) и `e_Configuration` (Конфигурация).

Заметим, что событие покупки `[on Purchase]`, инициируемое пользователем при работе со страницей `b_ConfigurationClientPage`, устанавливает окружение прецедента. Класс `b_OrderClientPage` отвечает за обработку содержимого клиентской формы заказа `b_OrderClientForm`. Форма представляет собой набор полей для ввода информации с экрана, которая принимает входные данные от пользователя. После того, как форма заполнена, она отправляется для обработки серверной странице (`c_OrderServerPage`).

Четыре объекта, относящиеся к стереотипу «entity» (e_Order, e_Configuration, e_Payment и e_Customer) хранят текущую информацию,относящуюся к заказу, независимо от Web-страниц. Следует, однако, напомнить, что это всего лишь *объекты памяти* — долговременное хранение заказа в базе данных выходит за рамки данного наставления.

С кооперацией связан объект d_Transaction (Транзакция), материализуемый классом c_OrderServerPage после отправки заказа. Хотя класс d_Transaction и не показан явно на диаграмме, он может взаимодействовать с базой данных с целью удостовериться в том, что передача заказа в базу данных осуществлена надлежащим образом. Этот класс также представляет механизм приложения для управления состоянием клиента — он вызывает размещение и удаление cookies с клиентской машины.

6.3.4.3. Поведенческая сторона кооперации

На рис. 6.31 приведена диаграмма классов, представляющая поведенческую сторону кооперации для прецедента "Order Configured Computer". Диаграмма выражает функциональные требования к прецеденту. Кооперативные действия начинаются в тот момент, когда удовлетворяется первое предусловие прецедента и клиент нажимает кнопку подтверждения покупки — Purchase. На диаграмме это событие моделируется как сообщение [on Purchase] newOrderClientPage.

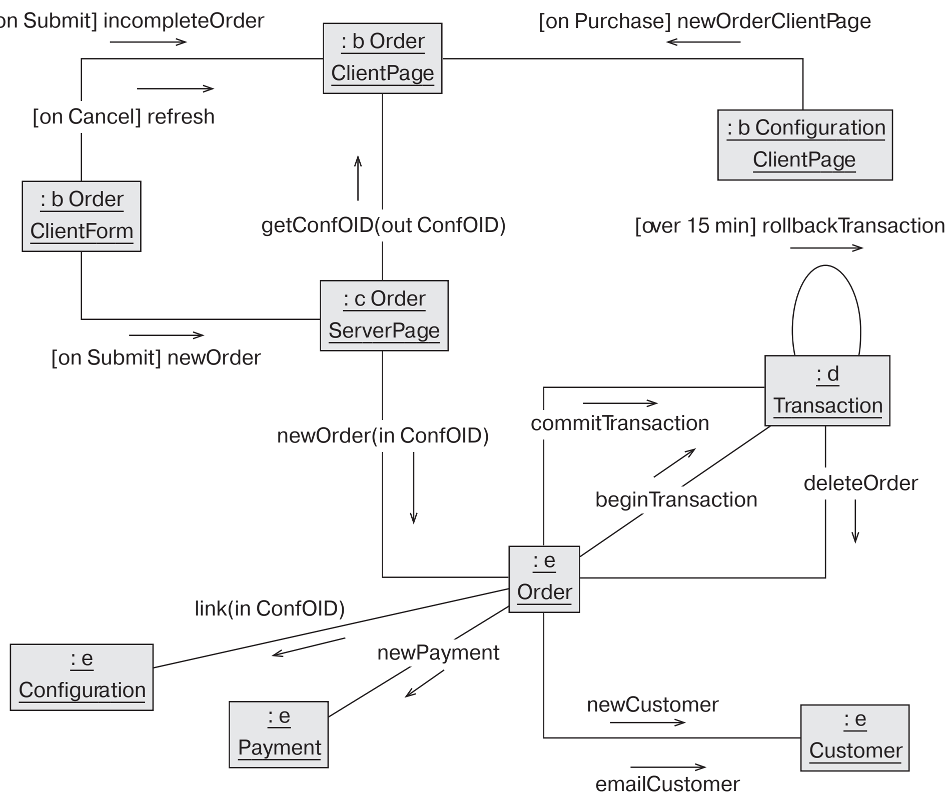

Рис. 6.31. Поведенческая сторона кооперации для прецедента "Order Configured Computer" (Internet-магазин)

Наставление по проектированию: шаг 7 (Internet-магазин)

Обратитесь к проектному документу по прецеденту “Order Configured Computer”, представленному в разделе 6.3.4.1.2, и предыдущему шагу наставления (разд. 6.3.4.2). Основываясь на потоке событий, представленном в этом проектном документе по прецеденту, и готовности классов,показанных на рис. 6.30, нам требуется создать проект кооперации, отражающий ее поведенческий аспект, который реализует прецедент “Order Configured Computer”.

Объект класса `b_OrderClientPage` обслуживает сообщение `[on Purchase] newOrderClientPage`. Эта представленная в формате HTML страница содержит три «`form`»-объекта класса `b_OrderClientForm` (рис. 6.30). Атрибуты этого класса представляют поля для ввода формы, включая две командные кнопки: `Cancel` (Отмена) и `Submit` (Отправить).

После того, как клиентская страница отображает форму на экране броузера (UC15.1), клиент может ввести информацию, касающуюся заказа (UC15.2 и UC15.3). Команда `Cancel` (UC15.11.1), обслуживается объектом класса `b_OrderClientPage` (с использованием метода, обновляющего содержимое страницы — `refresh`). В том случае, когда клиент не заполнил все необходимые поля (UC15.8.1), команда `Submit` также обслуживается объектом `b_OrderClientPage` (с использованием метода `incompleteOrder`).

Когда клиент нажимает кнопку `Submit` и вся необходимая информация предоставлена (UC15.5), управление передается классу `c_OrderServerPage`. Он создает объекты-сущности `e_Order`, `e_Customer`, `e_Payment` и `e_Configuration`. Первые три объекта-сущности хранят реквизиты формы заказа, введенные клиентом.

Объект-сущность `e_Configuration` был материализован прежде другим прецедентом (“Build Computer Configuration”). Теперь он связывается с объектом `e_Order` после того, как объект `c_OrderServerPage` получает его OID из атрибута связи, хранимого в объекте `b_OrderClientPage` (между классами `b_OrderClientPage` и `b_ConfigurationClientPage` определено отношение ассоциации).

После того, как объект `e_Order` материализован, начинается бизнес-транзакция. С этой целью создается новый объект `d_Transaction`. Он управляет состоянием клиента, включая интервалы, управляемые с помощью cookie. Если транзакция завершается успешно, `e_Order` требует от `e_Customer` подтвердить получение заказа клиентом с помощью электронной почты (UC15.9).

Если время хранения cookie истекает (UC15.4) или база данных возвращает ошибку (UC15.7.1), метод, реализующий откат — `rollbackTransaction`, требует от `e_Order` удалить самого себя, а объект `d_Transaction` переходит в распоряжение операционной системы. Чтобы продолжить попытку заказа компьютера, клиенту может потребоваться повторно ввести информацию в форму заказа. Модель взаимодействия между объектом `d_Transaction` и сервером баз данных на диаграмме отсутствует.

Резюме

Этой главой было открыто рассмотрение проблем системного проектирования — в ней представлены основания системного проектирования. Обсуждаемые здесь вопросы были прямым продолжением предыдущей главы, посвященной углубленному

анализу требований. Были прояснены два основных (и различных) аспекта проектирования — архитектурное проектирование и детализированное проектирование.

Типичные приложения ИС строятся на принципах архитектуры *клиент/сервер*. Специфические решения на базе этой архитектуры включают *распределенные системы обработки* и *распределенные системы баз данных*. *Трехзвенные системы* расширяют базовую клиент/серверную архитектуру за счет выделения логики приложения в отдельный логический и/или физический уровень.

Архитектурные решения оказывают влияние на проектирование интерфейсов между приложением и базой данных. Иерархия пакетов, построенная на основе подхода BCE (введенного в главе 5), была расширена за счет пакетов интерфейса с базами данных и образования *иерархии пакетов BCED*.

Повторное использование представляет собой один из важнейших проектных факторов, который влияет как на архитектурные решения, так и на вопросы детализированного проектирования. Решение проблемы повторного использования сводится к выбору между повторным использованием *инструментальных средств*, *каркасов* и *шаблонов*. Эти варианты выбора не являются взаимно исключающими — напротив, рекомендуется применять комплексную стратегию. Повторное использование за счет использования внешних источников следует увязывать с внутренним проектированием пакетов, *компонент*, классов и интерфейсов. В конечном итоге вычислительные ресурсы представляются в виде *диаграммы развертывания*.

Детализированное проектирование концентрируется на *кооперации*. Кооперация определяет реализацию прецедентов или операций. Кооперация дает модель передачи сообщений между объектами. При этом необходимо учитывать такие факторы, как замещение, перегрузка, итерация, шаблоны, автосообщения, асинхронные сообщения и обратные вызовы. *Структурные* аспекты кооперации моделируются в виде диаграмм классов, а *поведенческие* аспекты — в виде диаграмм кооперации.

Глава завершается наставлением с решениями для упражнений по проектному моделированию — продолжением наставления по анализу, представленному в главе 2 (при этом используется приложение *Internet-магазин*). Наставление служит иллюстрацией использования методов проектирования языка UML — проектирования пакетов, компонент, развертывания и кооперации.

Вопросы

В1. Объясните, в чем состоит различие между распределенной системой обработки и распределенной системой баз данных.

В2. Что такое трехзвенная архитектура? В чем ее преимущества и недостатки?

В3. Что мы понимаем под активной базой данных?

В4. Каково назначение пакета баз данных в подходе BCED?

В5. Что общего и каковы различия между повторным использованием инструментальных средств и каркасов?

В6. Каким образом компоненты и пакеты соотносятся между собой?

В7. Что такое доминантный класс?

В8. Что такое отношение соединения?

В9. Что такое отношение реализации, используемое в кооперации? Приведите пример (отличный, от приведенного в книге).

В10. Какими альтернативными способами можно представить аргументы сообщения на диаграммах кооперации?

В11. Какие типы сообщений можно отнести к основным (помимо конструкторов и деструкторов)?

В12. В чем заключается различие между замещением и перегрузкой?

В13. Каким образом итеративные сообщения связаны с коллекциями?

В14. Отправитель сообщения может послать или не посылать свой OID целевому объекту. Справедливо ли данное утверждение применительно к асинхронным сообщениям? Поясните свой ответ.

В15. Какие диаграммы языка UML используются для проектирования структурного аспекта кооперации? Объясните, насколько они подходят для выполнения этой задачи.

В16. Какие диаграммы языка UML используются для проектирования поведенческого аспекта кооперации? Сравните, насколько они подходят для выполнения этой задачи.

В17. Сравните, насколько пакеты прецедентов и пакеты классов применимы на этапах анализа и проектирования жизненного цикла разработки систем.

Упражнения

Дополнительные требования. Магазин видеопроката

Рассмотрите следующие дополнительные требования для приложения *Магазин видеопроката* (эти требования изложены в конце главы 4 и для удобства повторены здесь).

1. За кассеты и диски, возвращенные позже срока, взимается дополнительная плата за период, превышающий срок проката. Каждый видеоноситель обладает уникальным идентификационным номером.
2. Фильмы заказываются у поставщика, который в общем случае может поставить кассеты и диски в течение одной недели. Обычно один заказ делается на несколько фильмов.
3. Забронировать можно те фильмы, которые заказаны у поставщика и/или все копии которых находятся в прокате. Можно также забронировать те фильмы, которых нет в запасе и которые не заказаны у поставщика; при этом с клиента требуется задаток за один период проката.
4. Клиент может также сделать несколько предварительных заказов, однако, для каждого забронированного фильма готовится отдельный запрос на бронирование. Бронирование может быть отменено из-за отсутствия реакции со стороны клиента, более точно, в течение одной недели с момента, когда клиенту было сообщено о возможности взять фильм напрокат. Если за фильм был уплачен задаток, он записывается на счет клиента.
5. База данных хранит обычную информацию о поставщиках и клиентах, т.е. адреса, телефонные номера и т.д. В каждом заказе поставщику указываются заказываемые фильмы, их количество, форматы кассеты/диска, а также дата ожидаемой доставки, отпускная цена, возможные скидки и т.д.
6. Когда кассета возвращается клиентом или поступает от поставщика, вначале удовлетворяются предварительные заказы. Работники магазина устанавливают контакт с клиентами, сделавшими предварительный заказ. Для правильной обработки бронирования фильмов информация, связанная с бронированием, обновляется дважды: после установления контакта с клиентом, когда ему сообщается, что “забронированный фильм пришел”, и после сдачи фильма клиенту напрокат. Эти шаги гарантируют правильное проведение операции бронирования.
7. Клиент может взять несколько кассет или дисков, однако, каждому взятому видеоносителю ставится в соответствие отдельная запись. Для каждого выдаваемого напрокат фильма фиксируются дата и время выдачи, установленный и фактический

срок возврата. Позже запись о прокате обновляется, чтобы отразить факт возврата видеофильма и факт окончательного платежа (или возврата денег). Кроме того, запись хранит информацию о продавце, отвечающем за прокат фильма. Детальная информация о клиенте и по прокату хранится в течение года, чтобы можно было легко определить уровень доверия к клиенту. Старая информация по прокату сохраняется в течение года в целях проведения аудита.

8. Все операции выполняются с использованием наличности, электронного перевода денег или кредитных карточек. От клиентов требуется внести плату за прокат при выдаче кассет/дисков.
9. Если кассета/диск возвращены позже установленного срока (или не могут быть возвращены по каким-либо причинам), плата снимается либо со счета клиента, либо принимается непосредственно от клиента.
10. Если кассета/диск задержаны более, чем на два дня, клиенту отправляется уведомление о задержке. После отправки двух уведомлений о задержке одной и той же кассеты/диска клиент предупреждается о том, что он является “нарушителем”, и при следующем его обращении в магазин руководство рассматривает вопрос о снятии с него статуса “нарушителя”.

У1. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте структурную диаграмму кооперации для реализации прецедента “Резервирование видео” (“Reserve Video”).

У2. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте поведенческую диаграмму кооперации для реализации прецедента “Резервирование видео”.

У3. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте структурную диаграмму кооперации для реализации прецедента “Возврат видео” (“Return Video”).

У4. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте поведенческую диаграмму кооперации для реализации прецедента “Возврат видео”.

У5. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте структурную диаграмму кооперации для реализации прецедента “Заказ видео” (“Order Video”).

У6. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте поведенческую диаграмму кооперации для реализации прецедента “Заказ видео”.

У7. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте структурную диаграмму кооперации для реализации прецедента "Сопровождение клиентов" ("Maintain Customer").

У8. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.14 (разд. 4.3.1.3). За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Спроектируйте поведенческую диаграмму кооперации для реализации прецедента "Сопровождение клиентов".

У9. *Магазин видеопроката* — обратитесь к примеру 4.16 (разд. 4.3.2.3). Рассмотрите вид деятельности `Update Stock` (Обновление запаса), показанный на рис. 4.13. За помощью обратитесь также к решениям упражнений для приложения *Магазин видеопроката* в конце главы 4.

Изобразите диаграмму видов деятельности для реализации операции `Update Stock`.