ГЛАВА 5

Переход от анализа к проектированию

Цели

Цели

В предыдущих двух главах картина визуального объектно-ориентированного моделирования нарисована в розовом свете. Примеры были нетребовательными, язык визуального проектирования — простым и привлекательным для использования, зависимости между моделями — очевидными. Мы вольно обращались с методами моделирования и не слишком заботились об альтернативных решениях.

Реалии разработки программного обеспечения куда сложнее. Как мы заметили в начале книги, простых решений для сложных проблем не существует. Объекты составляют основу современной технологии решения сложных проблем. А раз так, то объекты обязаны обеспечить и техническую глубину, соответствующую уровню сложности, к которой они обращены.

Эта глава призвана дать критическую оценку объектной технологии и ее пригодности к решению сложных проблем. Мы вводим передовые концепции в моделирование классов, иерархию классов, наследование и делегирование. На протяжении всей главы мы сравниваем, выносим решения, высказываем мнение и предлагаем альтернативные решения. Из-за своего технического характера многие темы, рассматриваемые в этой главе, переносятся прямо на системное проектирование. Это согласуется с универсальным характером UML, а также итеративным и наращиваемым процессом объектно-ориентированной разработки ПО.

Прочитав главу, читатели будут

- уметь настраивать язык UML для удовлетворения конкретных потребностей и технологических требований проекта;
- знать свойства языка UML для моделирования на более низких абстрактных уровнях;
- понимать, что не следует применять слепо некоторые мощные объектные технологические концепции;
- осознавать компромиссы моделирования, особенно в отношении обобщения и агрегации;
- владеть практическими навыками создания высокоуровневых моделей взаимодействия.

5.1. Углубленное моделирование классов

Концепции моделирования анализа, рассмотренные ранее, были достаточны для выработки завершенных моделей анализа. Однако обеспечиваемый этими концепциями уровень абстракции не исчерпывает всех возможных деталей, допу-

стимых в рамках моделирования анализа (т.е. деталей, которые не касаются аппаратных/программных решений, но обогащают семантику модели). UML включает нотацию для ряда дополнительных концепций, о которых мы упоминаем только вскользь или которых не касаемся вообще.

Дополнительные концепции моделирования включают **стереотипы** (stereotypes), **ограничения** (constraints), **производную информацию** (derived information), **видимость** (visibility), *квалифицированные ассоциации* (qualified associations), *ассоциативные классы* (association classes), *параметризованные классы* (parametrized class) и некоторые другие. Эти концепции необязательны. Многие модели могут быть вполне приемлемы без них. Их применение требует осторожности и точности, чтобы любой, кто попробует разобраться с моделями впоследствии, мог без труда понять намерения разработчика.

5.1.1. Механизмы расширения

Стандарт UML содержит набор концепций моделирования и обозначений, которые могут применяться в любых проектах, связанных с разработкой программного обеспечения. Однако эта универсальность означает, что для некоторых специфических и неортодоксальных проектов стандартных средств языка UML может оказаться недостаточно. Стандарт UML, как и любой стандарт, представляет собой "наименьший общий знаменатель". Было бы нецелесообразно усложнять язык UML экзотическими свойствами, предназначенными для узкоспециализированных предметных областей, инфраструктур программного обеспечения или языков программирования.

В то же время существует необходимость выйти за пределы встроенных возможностей языка UML. Для этой цели в языке UML предусмотрены механизмы расширения, такие как стереотипы, ограничения, **определения дескрипторов** и **меченые значения**. Согласованный набор расширений материализует в языке UML концепцию **профиля**, которая, в свою очередь, расширяет ссылочную **метамодель** (т.е. язык UML), ориентируясь на разные прикладные области или технологические платформы. Например, можно создавать профили языка UML для моделирования баз данных, разработки хранилищ данных или Web-систем.

5.1.1.1. Стереотипы

Стереотип (stereotype) расширяет существующий элемент моделирования в языке UML, изменяя его семантику. По существу стереотип не является новым элементом модели и не влияет на структуру UML. Он лишь обогащает содержание существующей нотации, расширяя и уточняя модель. Существуют различные способы применения стереотипов.

Обычно стереотипы *метятся* в моделях с помощью имен, заключенных в угловые кавычки, например «`global`», «`PK`», «`include`». Кроме того, стереотип можно изобразить с помощью *пиктограммы*.

Некоторые распространенные стереотипы являются встроенными — они заранее определены в UML. Некоторые CASE-инструменты включают готовые пиктограммы для встроенных стереотипов. Большинство CASE-инструментов обеспечивают возможность создания новых пиктограмм по желанию аналитика. На рис. 5.1 показаны примеры классов, стереотипизированных с помощью пиктограммы и метки соответственно, а также пример класса, не являющегося стереотипом.

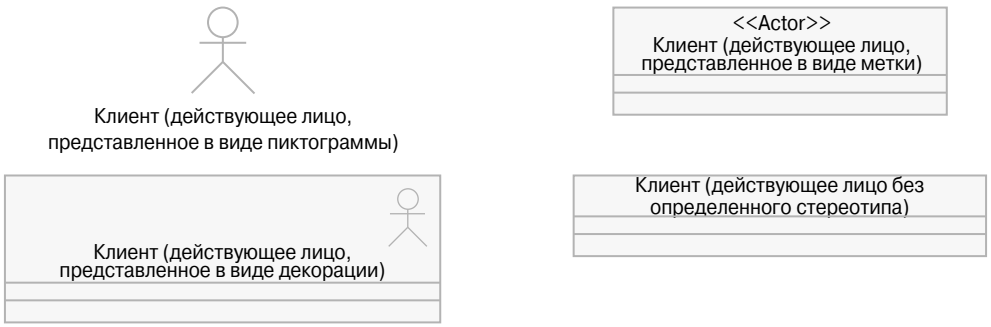


Рис. 5.1. Графическое представление стереотипов

Ограничение, гласящее, что стереотип расширяет семантику, но не структуру UML, не имеет большого значения. Главная отличительная черта любой объектно-ориентированной системы заключается в том, что она *полностью состоит из объектов* (объектами являются классы, атрибуты, методы и т.д.). Следовательно, стереотипизирование класса может привести в итоге к созданию нового элемента моделирования, который вводит новую категорию объектов.

5.1.1.2. Комментарии и ограничения

"**Комментарий** (comment) — это текстуальная аннотация, сопровождающая набор элементов... Компонент дает возможность сопровождать элементы разными примечаниями. Комментарии не имеют семантического значения, но могут содержать информацию, полезную для моделирования... Комментарии указываются в прямоугольнике, правый верхний угол которого загнут (так называемый "символ примечания"). Этот прямоугольник содержит текст комментария. Связь с каждым из аннотированных элементов указывается отдельной пунктирной линией" (UML, 2005).

"*Ограничение* — это условие, выраженное на естественном или машинном языке для объявления семантики элемента... Ограничение выражает дополнительную семантическую информацию, связанную с ограничиваемыми элементами. Ограничение — это условие, которое должно выполняться для корректного проектирования системы... Ограничение представляет собой текстовую строку в фигурных скобках ({}). Для элементов, примечание к которым выражается текстовой строкой (например, атрибутов и т.д.), строка ограничения может следовать вслед за тексто-

вой строкой в скобках. Если ограничение относится к отдельному элементу (классу, ассоциативному пути и т.п.), то оно указывается возле него, желательно ниже имени... Если ограничение относится к двум элементам (двум классам или двум ассоциациям), то оно указывается в виде пунктирной линии, проведенной между элементами, помеченными строкой ограничения (в скобках)" (UML, 2005).

Поскольку ограничения также могут выражаться в виде текстовых строк, разница между комментарием и ограничением заключается не столько в виде представления, сколько в семантических последствиях. Комментарий не имеет никакого семантического значения для модели. *Ограничение* имеет семантическое значение для модели и (в идеале) должно записываться на формальном языке. Фактически язык UML предлагает специализированный язык для этой цели — *Object Constraint Language* (OCL).

На диаграмме модели указываются только простые комментарии и ограничения. Более сложные комментарии и ограничения (с более длинными пояснениями) хранятся в CASE-хранилище как текстовые документы.

Некоторые виды ограничений в языке UML предусмотрены заранее. На рис. 4.8 (см. раздел 4.2.2.2 главы 4) показан пример использования встроенного *XOR-ограничения*. Ограничения, введенные проектировщиками, образуют *механизм расширения*.

Стереотипы часто путают с ограничениями. Действительно, разница между ними со временем стерлась. *Стереотип* часто используется для введения нового *ограничения* в модель — иногда представляющего ценность для проектировщика, но не имеющего непосредственной поддержки в языке UML.

Пример 5.1. Прямой маркетинг по телефону

Обратитесь к примеру 4.7 (см. разд. 4.2.1.2.3 главы 4). Рассмотрите классы `ECampaign и ECampaignTicket` (см. рис. 4.7) и установите отношение между ними. Добавьте в класс `ECampaign` операции для вычисления количества проданных и оставшихся билетов, а также для определения продолжительности кампании и количества дней, оставшихся до ее закрытия.

Добавьте в диаграмму информацию о том, что класс `ECampaign` отличается от бонусной кампании. Кроме того, напишите на диаграмме напоминание об операциях, которые следует добавить в класс `ECampaignTicket`.

Напомним, что часть второго требования, указанного в примере 4.7, гласит: "Все билеты пронумерованы. На протяжении отдельной кампании каждый билет имеет уникальный номер". Включите это требование в список ограничений. (Решая пример 4.7 (рис. 4.7), мы не стали фиксировать указанное выше ограничение, а лишь включили атрибуты `campaignCode` и `ticketNumber` как часть составного первичного ключа для класса `CampaihnTicket`.)

Простое расширение модели класса, удовлетворяющее требованиям примера 5.1, приведено на рис. 5.2. На рисунке показаны два комментария и одно ограничение.

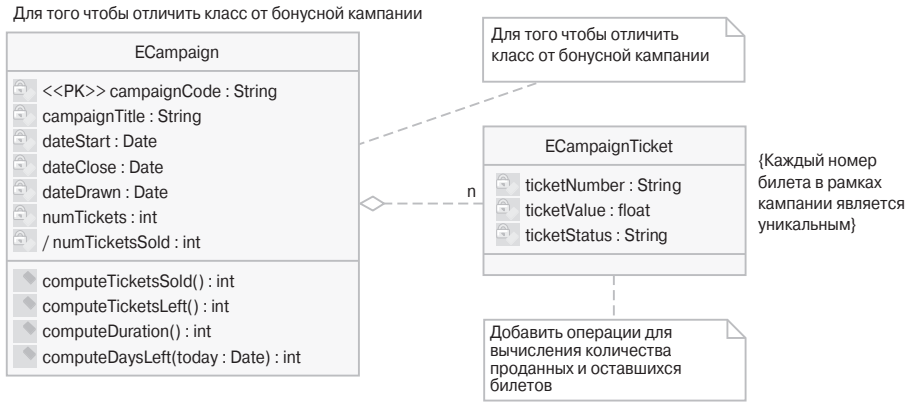


Рис. 5.2. Два комментария и ограничение для системы прямого маркетинга по телефону

Пример 5.2. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.8 (см. разд. 4.2.2.2.1 главы 4). Предположим, что в систему введено новое ограничение: сотрудники не должны планировать мероприятия для себя. Это значит, что сотрудник, запланировавший мероприятие, должен отличаться от сотрудника, отвечающего за проведение мероприятия.

Расширьте соответствующую часть модели классов (см. рис. 4.8) таким образом, чтобы включить новое требование.

Решение примера 5.2 (рис. 5.3) выражается в виде ограничения, наложенного на ассоциативные линии. Они изображаются в виде прерывистой стрелки-указателя зависимости, проведенной от ассоциации `Создано` к ассоциации `Подлежит выполнению`. Это ограничение *привязано* к стрелке.

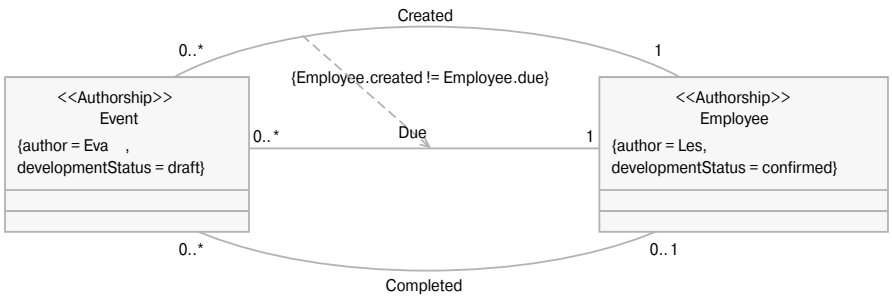


Рис. 5.3. Ограничения, наложенные на ассоциацию в системе управления взаимоотношениями с заказчиками

5.1.1.3. Примечания и дескрипторы

Понимание концепции дескрипторов требует различать *определение дескриптора* и *меченое значение*. *Определение дескриптора* (tag definition) — это свойство стереотипа. Оно показывается в виде прямоугольника, символизирующего атрибут класса, содержащего определение стереотипа. "*Меченое значение* (tagged value) — это пара "имя–значение", которую можно присоединить к элементу модели, использующему стереотип, содержащий определение дескриптора" (Rumbaugh et al., 2005).

Подобно примечаниям, *дескрипторы* (tag) представляют произвольную текстовую информацию в модели и записываются в фигурных скобках.

`tag = значение` — например:

```
{analyst = Les, developmentStatus = confirmed}
```

Поскольку дескриптор можно представить лишь в виде атрибута, определенного в стереотипе, стереотип с определениями дескрипторов должен быть определен в элементе модели (например, пакета) до того, как меченое значение будет приписано к конкретному экземпляру данного элемента модели.

Аналогично стереотипам и ограничениям, некоторые дескрипторы в UML определены заранее. Обычно дескрипторы используются для отображения информации об управлении проектом.

Пример 5.3. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.8 (см. разд. 4.2.2.2.1 главы 4) и приведенному выше примеру 5.2. Определите стереотип `Authorship` и сделайте так, чтобы его можно было применить к любому элементу модели. Объявите дескрипторы `author` и `developmentStatus`. Примените определенные стереотипы к классам `Event` и `Employee`. Укажите некоторые меченые значения.

Предположим, что появилось новое требование: сотрудник, запланировавший задание, должен также создать первое событие для этого задания в рамках той же самой транзакции. Расширьте соответствующую часть модели класса (см. рис. 4.8 и 5.3) так, чтобы учесть новое требование. Используйте для этого примечание об ограничении, определенное в классе `Task`.

Расширенная модель для примера 5.3 показана на рис. 5.4. Стереотип `Authorship` (Авторство) с определениями дескрипторов показан в левом верхнем углу. Для выражения ограничения, присоединенного к элементу `Task` (Задание), используется примечание. (В идеале примечание об ограничении должно быть присоединено ко всем трем ассоциациям, но это требование трудно реализовать с помощью CASE-инструментов. Причина заключается в том, что ограничение относится не к примечанию, а к элементу модели, например, к классу или ассоциации.)

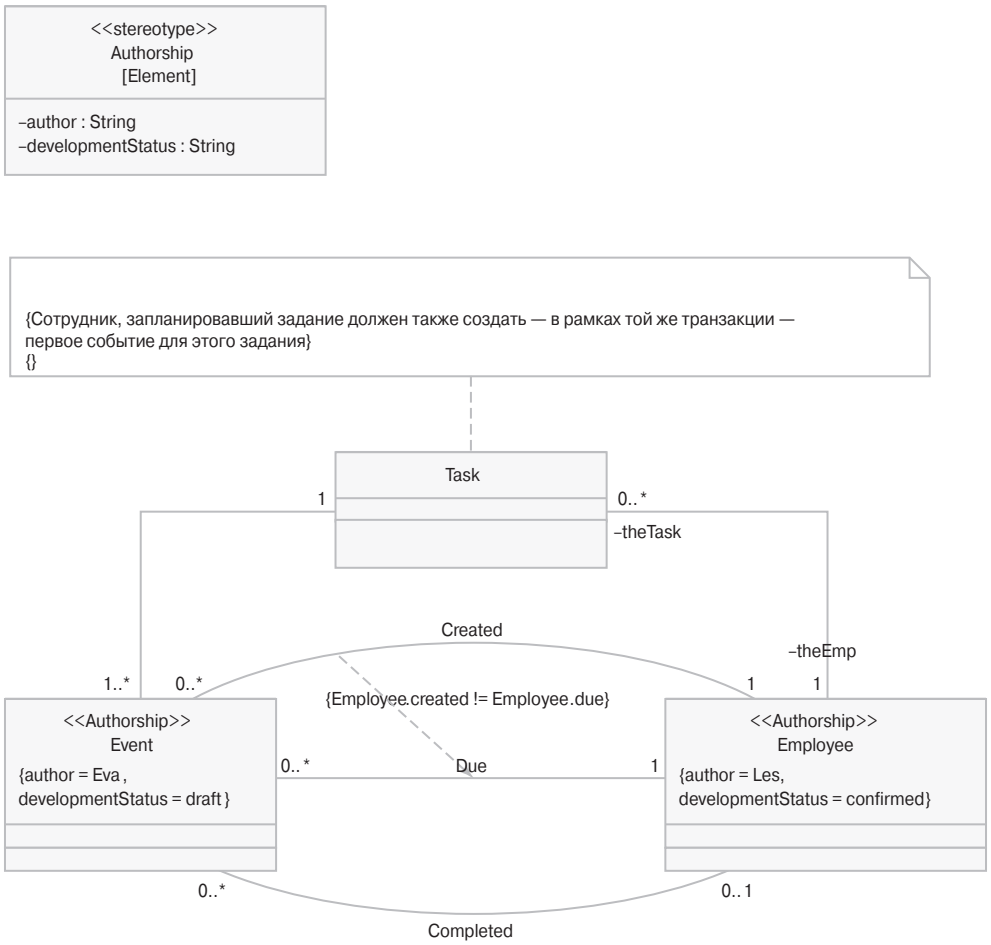


Рис. 5.4. Дескриптор и ограничения, наложенные на систему управления взаимоотношениями с заказчиками

5.1.2. Видимость и инкапсуляция

Концепция *видимости* (visibility) и связанное с ней понятие *инкапсуляции* (encapsulation) рассматриваются в приложении А в ракурсе видимости классов, т.е. *видимости атрибутов* (см. раздел А.3.1.2 приложения А) и *видимости операций* (см. раздел А.3.2.2 приложения А). Видимость класса означает, что другой класс имеет доступ к его элементам. В приложении А описывается *открытая* (public) и *закрытая* (private) видимость *атрибутов* и *операций*. В стандарте UML предусмотрены еще два вида видимости — *защищенная* (protected) и *пакетная* (package).

Точно так же можно определить видимость *класса*, *интерфейса*, *ассоциации* или *пакета* (в противоположность видимости элементов класса, т.е. *атрибутов* и *операций*). В *ассоциации* видимость применяется к *именам ролей* (см. раздел А.3.1.1

приложения А). Поскольку имена ролей (называемые в языке UML 2.0 *именами полюсов ассоциации*) реализуются как атрибуты классов, видимость имен ролей означает, что результирующие атрибуты можно использовать в выражениях доступа для перехода вдоль ассоциации. В *пакете* понятие видимости применяется к элементам, содержащимся в пакете, например к *классам*, *ассоциациям* и *вложенным пакетам*. Это значит, что другие пакеты имеют доступ к элементам данного пакета.

Перечислим полный набор маркеров видимости для атрибутов и операций классов:

+ — для открытой видимости;
– — для закрытой видимости;
— для защищенной видимости;
~ — для пакетной видимости.

CASE-инструменты часто заменяют эти довольно невыразительные обозначения более яркими и привлекательными графическими маркерами. Пример альтернативных обозначений приведен на рис. 5.5.

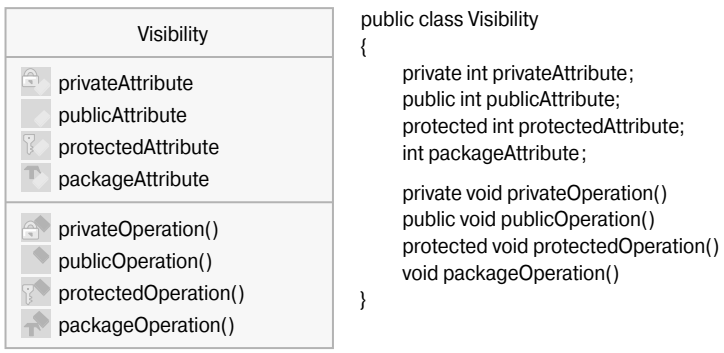


Рис. 5.5. Обозначения видимости в CASE-инструменте и соответствующий код на языке Java

5.1.2.1. Защищенная видимость

Защищенная видимость применяется в контексте наследования. *Закрытые свойства* (атрибуты и операции) базового класса, доступные лишь объектам самого класса, не всегда удобны. Часто необходимо открыть доступ к закрытым свойствам базового класса *объектам производного класса* (подкласса базового класса).

Рассмотрим иерархию классов, где `Person` — это (не абстрактный) базовый класс, а `Employee` — производный класс. Если `Joe` — объект класса `Employee`, то — по определению обобщения — `Joe` должен иметь доступ к свойствам (по меньшей мере к некоторым атрибутам) класса `Person` (например, к операции `getBirthDate()`).

Для того чтобы дать возможность производному классу осуществлять свободный доступ к свойствам его базового класса, эти (в противном случае — закрытые)

свойства необходимо определить в базовом классе как *защищенные*. (Напомним, что понятие видимости относится к классам. Если `Betty` — еще один объект класса `Employee`, то он должен иметь доступ к любому свойству объекта `Joe`: открытому, защищенному или закрытому.)

Пример 5.4. Прямой маркетинг по телефону

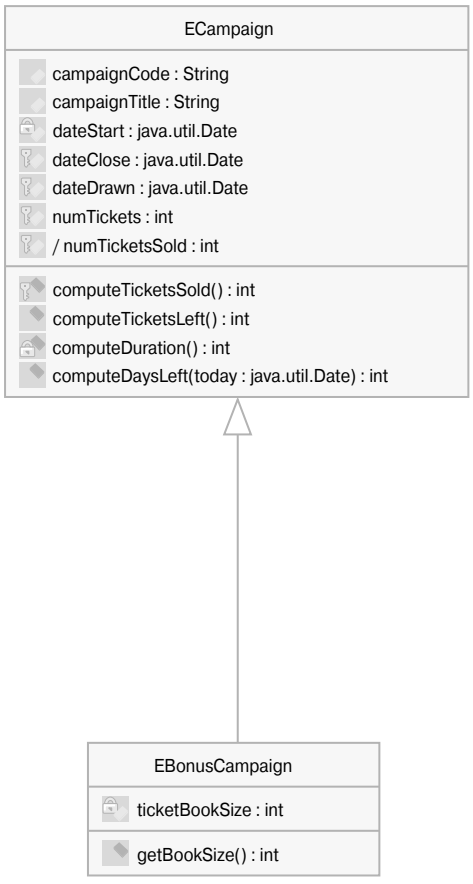
Обратитесь к постановке задачи 4 (см. раздел 1.6.4 главы 1) и примеру 4.7 (см. раздел 4.2.1.2.3 главы 4). Постановка задачи содержит следующее замечание: "Эти схемы включают специальные *бонусные кампании* для вознаграждения приверженцев, закупающих лотерейные билеты в массовом количестве, для привлечения новых жертвователей и т.д.". Это замечание еще не нашло отражения в модели.

Предположим, что одна из призовых кампаний включает "билетные книжки": если благотворитель покупает целую книжку билетов, ему бесплатно вручается дополнительный билет основной кампании.

Задача заключается в следующем.

- Обновить модель класса, включив в нее класс `EBonusCampaign`.
- Изменить видимость атрибутов класса `ECampaign` (см. рис. 5.2) таким образом, чтобы они были видимы классу `EBonusCampaign` за исключением атрибута `dateStart`. Сделать атрибуты `campaignCode` и `campaignTitle` видимыми для остальных классов модели.
- Добавить в класс `ECampaign` следующие вычислительные операции: `computeTicketsSold()` (вычислить количество проданных билетов), `computeTicketsLeft()` (вычислить количество оставшихся билетов), `computeDuration()` (вычислить продолжительность кампании), `computeDaysLeft()` (вычислить количество оставшихся дней кампании).
- Выяснить, что у внешних классов нет нужды в операции `computeTicketsSold()`, — им требуется только выполнять операцию `computeTicketsLeft()`. Кроме того, операция `computeDuration()` используется только операцией `ECampaign.computeDaysLeft()`.
- Класс `EBonusCampaign` хранит атрибут `ticketBookSize` (размер билетной книжки), а операция доступа к нему называется `getbookSize()`.

На рис. 5.6 показана модель классов и код на языке Java, соответствующие примеру 5.4. Операция `computeDuration()` в классе `ECampaign` является закрытой. Операция `computeDuration()` — защищенной, а остальные две — открытыми. Операция `getBookSize()` специфична для класса `EBonusCampaign` и является открытой.




```
public class ECampaign
{
    public String campaignCode;
    public String campaignTitle;
    private Date dateStart;
    protected Date dateClose;
    protected Date dateDrawn;
    protected int numTickets;

    protected int computeTicketsSold()
    {
    return 0;
    }

    public int compute Tickets Left()
    {
    return 0;
    }

    private int computeDuration()
    {
    return 0;
    }

    public int computeDaysLeft(Date today)
    {
    return 0;
    }
}

public class EBonusCampaign extends
ECampaign
{
    private int ticketBookSize;

    public int getBookSize()
    {
    return 0;
    }
}
```

Рис. 5.6. Использование защищенной видимости в системе прямого маркетинга по телефону

5.1.2.2. Видимость унаследованных свойств классов

Как указывалось в предыдущих разделах, понятие *видимости* применяется к объектам на различных уровнях детализации. Обычно имеется в виду, что видимость применяется к *элементарным объектам* (primitive objects) — атрибутам и операциям. Однако видимость можно задать и по отношению к другим "контейнерам". Это приводит к полной неразберихе с правилами замещения.

Рассмотрим, к примеру, ситуацию, когда видимость определяется в *иерархии наследования* на уровне базового класса и на уровне *свойств* базового класса. Пусть класс `B` — подкласс класса `A`. Класс `A` содержит смесь атрибутов и операций; одни из них являются открытыми, другие — закрытыми, а некоторые — защищенными. Вопрос звучит так: "К какому типу видимости относятся унаследованные свойства в классе `B`?"

Ответ на этот вопрос зависит от уровня видимости, установленного базовому классу A при объявлении его в производном классе B. Базовый класс можно объявить открытым (`class B: public A`), защищенным (`class B: protected A`) или закрытым (`class B: private A`).

Типичная реализация приведенного выше сценария выглядит следующим образом (Horton, 1997).

- Закрытые свойства (атрибуты и операции) базового класса A не видимы для объектов класса B, независимо от того, как базовый класс определен в классе B.
- Если базовый класс A определен как `public`, видимость унаследованных свойств в производном классе B не изменяется (открытые свойства остаются открытыми, а защищенные — защищенными).
- Если базовый класс A определен как `protected`, видимость унаследованных свойств в производном классе B изменяется на `protected`.
- Если базовый класс A определен как `private`, видимость унаследованных свойств с типом видимости `public` и `protected` в производном классе B изменяется на `private`.

Отметим, что в контексте проведенного выше обсуждения понятие *наследования реализации* означает следующее: если в базовом классе A существует свойство x, то оно существует и во всех классах, производных от класса A. Однако *наследование* свойства не обязательно означает, что это свойство *доступно* в производных классах. В частности, закрытые свойства базового класса остаются закрытыми в базовом и не доступными в производном классе. Это продемонстрировано на рис. 5.7, на котором показаны (доступные) свойства, унаследованные от класса `EBonusCompany`, представленного на рис. 5.6.

5.1.2.3. Видимость в пакетах и дружественных классах

Во многих ситуациях некоторые классы должны иметь прямой доступ к свойствам другого класса, в то время как остальные классы в системе остаются закрытыми. Язык Java поддерживает такие возможности с помощью *пакетов*. Язык C++ решает эту задачу, определяя *дружественные операции и классы*.

Пакетная видимость в языке Java предусмотрена по умолчанию. Если перед атрибутом или операцией в программе на языке Java не указаны ключевые слова `private`, `protected` или `public` (или ключевое слово `public` перед всем классом), то их видимость по умолчанию считается пакетной. Пакетная видимость означает, что все остальные классы в пакете имеют прямой доступ к этому атрибуту и операции. Однако для всех классов в *других* пакетах этот атрибут, операция или класс остаются закрытыми.

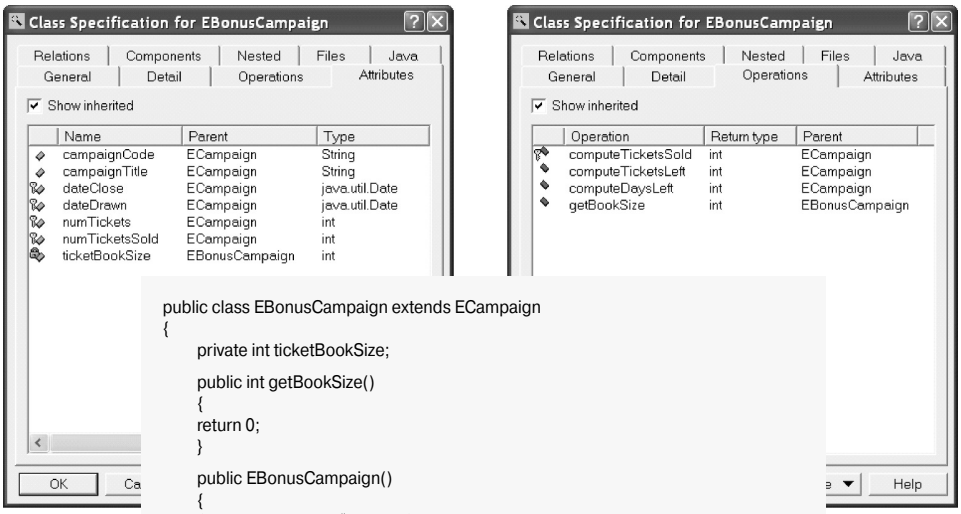


Рис. 5.7. Доступность свойств производного класса в системе прямого маркетинга по телефону

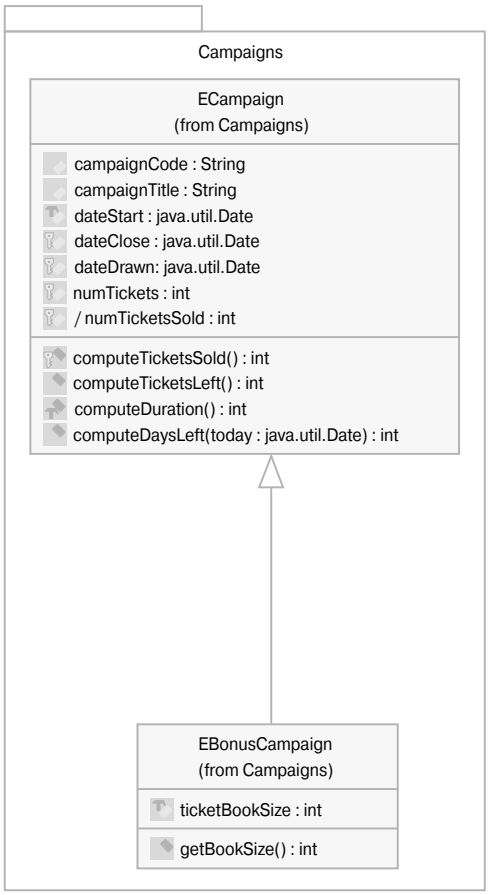
Защищенная (и открытая) видимость также является пакетной, но не наоборот. Это значит, что другие классы в том же самом пакете могут иметь доступ к защищенным свойствам, но *производные* классы не имеют доступа к свойствам, имеющим пакетную видимость, если производные и базовый классы относятся к разным пакетам.

На рис. 5.8 показан вариант модели, представленной на рис. 5.6. В этом варианте закрытый метод `computeDuration()` и закрытые данные-члены `dateStart` и `ticketBookSize` имеют пакетную видимость. Более того, класс `EBonusCampaign` также имеет пакетную, а не открытую видимость.

Как и пакетная видимость в языке Java, концепция *дружественных операций и классов* в языке C++ позволяет решить проблему, когда существует несколько связанных классов и одному из них необходим доступ к закрытым свойствам другого класса. Типичным примером являются два класса, `Book` (Книга) и `BookShelf` (Книжная полка), а также операция `putOnBookShelf()` в классе `Book`.

Для того чтобы решить эту проблему, можно объявить операцию `putOnBookShelf()` *дружественной* по отношению к классу `BookShelf`.

```
friend void Book:putOnBookShelf()
```




```
package Campaigns;

import java.util.Date;

public class ECampaign
{
      public String campaignCode;
      public String campaignTitle;
      Date dateStart;
      protected Date dateClose;
      protected Date dateDrawn;
      protected int numTickets;

      protected int computeTicketsSold()
      {
      return 0;
      }

      public int compute TicketsLeft()
      {
      return 0;
      }

      int computeDuration()
      {
      return 0;
      }

      public int computeDaysLeft(Date today)
      {
      return 0;
      }
}

package Campaigns;

class EBonusCampaign extends ECampaign
{
      int ticketBookSize;

      public int getBookSize()
      {
      return 0;
      }
}
```

Рис. 5.8. Использование пакетной видимости в системе прямого маркетинга по телефону

Дружественными могут быть как *класс*, так и *операция* другого класса. Дружественность не носит взаимного характера. Класс, объявивший другой класс дружественным, не обязательно является дружественным по отношению к нему.

Дружественная операция или класс объявляется *внутри* класса, предоставляющего другому классу права друга. Однако дружественная операция не является свойством класса, поэтому к ней неприменимы атрибуты видимости. Это также означает, что в определении дружественного класса нельзя упоминать атрибуты класса просто по имени, — все они должны уточняться именем класса (как если бы дружественная операция была обычной внешней операцией).

В языке UML дружественные отношения показаны с помощью пунктирной линии *отношения зависимости* (dependency relationship), исходящей из класса или операции и направленной к классу, который предоставляет права друга. Стереотип «`friend`» привязан к стрелке зависимости. По общему признанию, нотация UML не полностью воспринимает и поддерживает семантику дружественности.

Пример 5.5. Прямой маркетинг по телефону

Обратитесь к примеру 4.7 (см. раздел 4.2.1.2.3 главы 4). Рассмотрите отношение между классами `ECampaign` и `ECallScheduled`.

Объекты класса `ECallScheduled` очень активны, и при выполнении операций им требуются специальные права. В частности, они выполняют операцию `getTicketsLeft()` (рис. 5.9), которая устанавливает, остались ли какие-то билеты, чтобы заказ благотворителя мог быть удовлетворен. Существенно, что эта операция имеет прямой доступ к свойствам класса `ECampaign` (таким как `numTickets` и `numTicketsSold`).

Наша задача — объявить операцию `getTicketsLeft()` дружественной классу `ECampaign`.

Обозначения языка UML для отношения дружественности в системе прямого маркетинга по телефону, описанной в примере 5.5, показаны на рис. 5.9. Отношение зависимости, стереотипизированное ключевым словом «`friend`», означает, что класс `ECallScheduled` зависит от класса `ECampaign`, являясь дружественным по отношению к нему. Это отражает тот факт, что класс `ECampaign` предоставляет статус дружественного классу `ECallScheduled` (операция `getTicketsLeft()` объявлена дружественной в классе `ECampaign`, поскольку класс `ECampaign` должен сам называть своих друзей).

```
class ECampaign{
public:
    friend void ECallScheduled::getTicketsLeft();
};
```

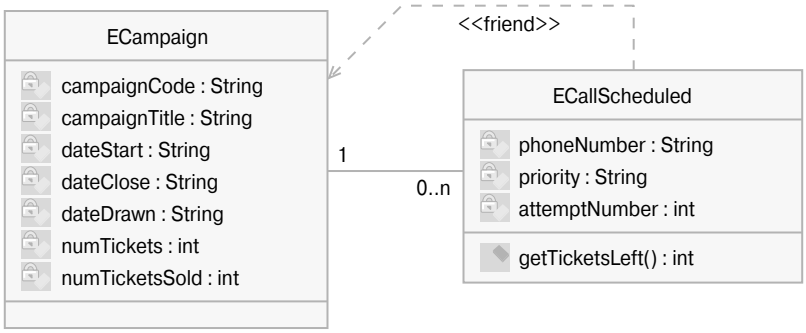


Рис. 5.9. Отношение дружественности в системе прямого маркетинга по телефону

5.1.3. Производная информация

Производная информация (derived information) представляет собой разновидность *ограничения*, которое чаще всего накладывается на *атрибуты* или *ассоциации*. Производная информация вычисляется на основе других элементов модели. Строго говоря, производная информация является избыточной — при необходимости ее можно вычислять.

Несмотря на то что производная информация не обогащает семантику *модели анализа* (analysis model), она придает ей большую четкость (поскольку в модели явно указывается, что некая величина вычисляется). Решение о том, приводить ли в модели анализа производную информацию, разработчик должен принимать самостоятельно, поскольку эта информация последовательно учитывается во всей модели.

Знание, что определенная информация является производной, имеет большее значение в *проектной модели* (design model), в которой необходимо оптимизировать доступ к информации. В проектных моделях можно также решить, хранить ли производную информацию (после ее вычисления) или динамически вычислять каждый раз, когда в ней возникает необходимость. Это не новое свойство — в прежних сетевых базах данных оно было известно как концепция *актуальных* (хранимых) и *виртуальных* данных.

В языке UML производная информация обозначается с помощью косой черты (`/`) перед именем производного атрибута или ассоциации.

5.1.3.1. Производный атрибут

Мы уже использовали производные атрибуты ранее в нескольких диаграммах, правда, не приводя никаких объяснений. Например, атрибут `/numTicketsSold` на рис. 5.2 — производный в классе `ECampaign`. Значение атрибута `/numTicketsSold` вычисляется операцией `computeTicketsSold()`. Эта операция прослеживает связи агрегации, идущие от объекта класса `ECampaign` к объектам класса `ECampaignTicket`, и проверяет каждый атрибут `ticketStatus`. Если атрибут `ticketStatus` принимает значение "продан", то к счетчику проданных билетов добавляется единица. После обработки всех билетов вычисляется текущее значение количества проданных билетов `numTicketsSold`.

5.1.3.2. Производная ассоциация

Производная ассоциация — более спорная тема. Типичным случаем производной ассоциации можно считать ассоциацию, образованную тремя классами, уже соединенными двумя ассоциациями, но не имеющими третьей ассоциации, замыкающей цикл. Зачастую необходимость в третьей ассоциации возникает в связи с требованием семантической корректности модели (*коммутативность цикла*). Если третья ассоциация не представлена в модели явно, она может быть выведена из двух других ассоциативных связей.

Пример 5.6. База данных о заказах

Рассмотрите простую базу данных о заказах с классами `Customer`, `Order` и `Invoice`. Предположим, что заказ всегда формируется одним клиентом, а каждый счет-фактура генерируется для отдельного заказа.

Выведите ассоциацию между этими тремя классами. Существует ли возможность ввести производную ассоциацию в модель?

Между классами `Customer` и `Invoice` возможно ввести производную ассоциацию. На рис. 5.10 она обозначена именем `/CustInv`. Эта ассоциация выводится благодаря несколько необычному бизнес-правилу, по которому кратность ассоциации между классами `Order` и `Invoice` равна "один к одному".

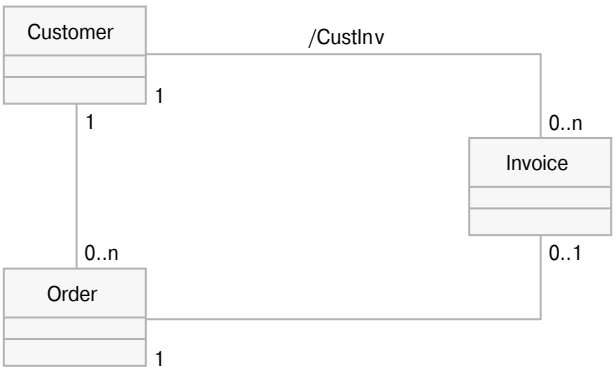


Рис. 5.10. Производная ассоциация

Производная ассоциация не вносит в модель какую-либо новую информацию. Всегда можно связать клиента с определенным счетом-фактурой, отыскав единственный заказ для каждого счета-фактуры, а затем одного клиента для каждого заказа.

5.1.4. Квалифицированная ассоциация

Концепция *квалифицированной ассоциации* (qualified association) вызывает споры среди специалистов. Некоторые охотно пользуются этим понятием, другие не воспринимают его совсем. Неизвестно, можно ли построить полную и достаточно выразительную модель классов без использования квалифицированной ассоциации. Однако если уж использовать квалифицированную ассоциацию, то делать это следует последовательно.

На одном полюсе бинарной квалифицированной ассоциации имеется "отделение" для атрибута (*квалификатора*). Ассоциация редко квалифицируется на обоих полюсах. Это "отделение" содержит один или несколько атрибутов, служа-

щих ключами индексирования для прослеживания ассоциативной связи от квалифицированного *класса-источника* (source class) к *целевому классу* (target class) на противоположном полюсе ассоциации.

Например, ассоциация между классами `Flight` (Рейс) и `Passenger` (Пассажир) имеет кратность "многие ко многим". Однако, если класс `Flight` квалифицировать с помощью атрибутов `seatNumber` (место) и `departure` (пункт отправления), то кратность ассоциации снижается до значения "один к одному" (рис. 5.11). Составной ключ индекса, введенный с помощью квалификатора (`flightNumber` + `seatNumber` + `departure`), может быть связан только с одним объектом `Passenger` либо не связан вообще ни с одним объектом этого класса.

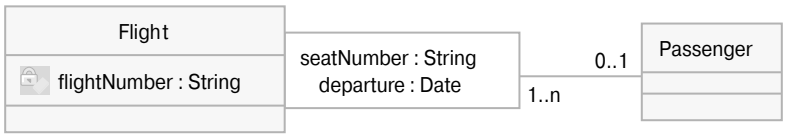


Рис. 5.11. Квалифицированная ассоциация

При прослеживании связи в прямом направлении кратность ассоциации представляет количество целевых объектов, связанных с составным ключом (квалифицированный объект + значение квалификатора). При прослеживании связи в обратном направлении кратность ассоциации описывает количество объектов, обозначенных составным ключом (квалифицированный объект + значение квалификатора) и связанных с каждым целевым объектом (Rumbaugh et al., 2005).

Уникальность в идентификации объектов, введенная с помощью квалификатора, часто представляет собой важную семантическую информацию, которую невозможно эффективно получить другими способами (такими как ограничения или включение дополнительных атрибутов в целевой класс). В общем случае дублировать атрибут квалификации в целевом классе нежелательно.

5.1.5. Ассоциативный или материализованный класс

В приложении А (см. раздел А.5.4) рассматривается пример *ассоциативного класса* (association class) — ассоциации, которая сама является классом. Ассоциативный класс обычно используется в тех случаях, когда между двумя классами существует ассоциация "многие ко многим" и каждый экземпляр ассоциации (связь) обладает собственными значениями атрибутов. Для того чтобы обеспечить возможность хранить эти атрибуты, требуется *ассоциативный класс*.

На первый взгляд простая концепция ассоциативного класса таит в себе хитрое ограничение. Рассмотрим ассоциативный класс *C* между классами *A* и *B*. Ограничение состоит в том, что для каждой пары связанных экземпляров классов *A* и *B* может существовать только один экземпляр класса *C*.

Если подобное ограничение неприемлемо, то специалист по моделированию должен *материализовать* ассоциацию, заменив класс *C* обычным классом *D* (Rumbaugh et al., 2005). *Материализованный класс* (reified class) *D* допускает две бинарные ассоциации с классами *A* и *B*. Класс *D* не зависит от классов *A* и *B*. Каждый экземпляр *D* обладает своей собственной идентичностью, так что при необходимости можно создать несколько экземпляров этого класса для того, чтобы установить связь между одними и теми же экземплярами классов *A* и *B*.

Различия между *ассоциативным и материализованным классами* чаще всего возникают в контексте моделирования временной (исторической) информации. Примером такого приложения является база данных сотрудников, в которой записаны их предыдущие и нынешние ставки.

5.1.5.1. Модель с ассоциативным классом

Пример 5.7. База данных о сотрудниках

Каждому сотруднику в организации присвоен уникальный идентификатор `empId`. Имя сотрудника хранится в виде имени, фамилии и инициалов.

Каждому сотруднику в штатном расписании установлен оклад. Для каждого уровня существует диапазон окладов, т.е. минимальная и максимальная зарплата. Диапазоны окладов для данного уровня никогда не изменяются. Если возникает необходимость изменить минимальный или максимальный оклад, создается новый уровень зарплаты. Кроме того, в организации хранятся начальная и конечная даты введения каждого уровня зарплаты.

В организации хранятся все предыдущие значения зарплаты сотрудника, включая начальную и конечную даты установления ему определенного уровня зарплаты. Также фиксируются любые изменения зарплаты сотрудника в рамках одного и того же уровня.

Разработайте модель классов для базы данных о сотрудниках, используя ассоциативный класс.

Пример 5.7 порождает проблемы. Нам требуется класс для хранения подробной информации о сотруднике (`Employee`), а также класс для хранения информации об уровнях зарплаты (`SalaryLevel`). Проблема состоит в моделировании текущих назначений зарплаты работникам, а также хронологии этих назначений. На первый взгляд кажется естественным использовать ассоциативный класс `SalaryHistoryAssociation`.

На рис. 5.12 представлена модель классов, в которой использован ассоциативный класс `SalaryHistoryAssociation`. Это решение неверно. Идентичность

объектов SalaryHistoryAssociation выводится на основании составного ключа, созданного с использованием ссылок на первичные ключи классов Employee и SalaryLevel (т.е. empId и levelId).

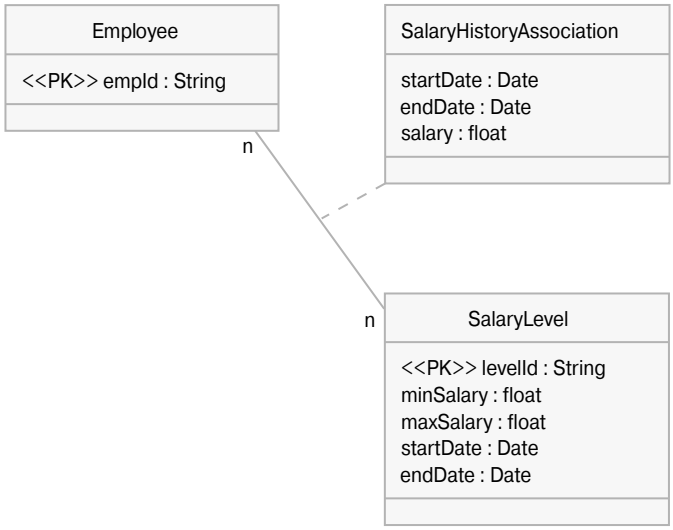


Рис. 5.12. Неправильное использование ассоциативного класса

Никакие два объекта класса SalaryHistoryAssociation не могут иметь одинаковых составных ключей (т.е. одинаковых связей с объектами класса Employee и SalaryLevel). Это также означает, что проект диаграммы на рис. 5.12 не обеспечивает выполнение следующего требования: любые изменения зарплаты сотрудника в рамках одного и того же уровня также фиксируются. Решение, показанное на рис. 5.12, не может рассматриваться как удовлетворительное — требуется более корректная модель.

5.1.5.2. Модель, использующая материализованный класс

Ассоциативный класс не может иметь дублирующихся ссылок на объекты ассоциированных классов. *Материализованный класс* не зависит от ассоциированных классов, и подобное ограничение на него не распространяется. Первичный ключ материализованного класса не использует атрибутов, обозначающих связанные классы.

Пример 5.8. База данных о сотрудниках

Обратитесь к примеру 5.7 (см. раздел 5.1.5.1). Создайте модель классов для базы данных о сотрудниках, используя материализованный класс.

Решение примера 5.8 продемонстрировано на рис. 5.13. Оно основано на использовании материализованного класса `SalaryHistoryReified`. Этот класс не имеет явно помеченного первичного ключа. Однако можно предположить, что этот ключ содержит атрибуты `empId` и `segNum`. Атрибут `segNum` хранит последовательные номера изменений оклада для каждого сотрудника. Каждый объект класса `SalaryHistoryReified` принадлежит отдельному объекту класса `Employee` и связан с единственным объектом класса `SalaryLevel`. Теперь модель учитывает изменения зарплаты сотрудников, относящихся к одному и тому же уровню.

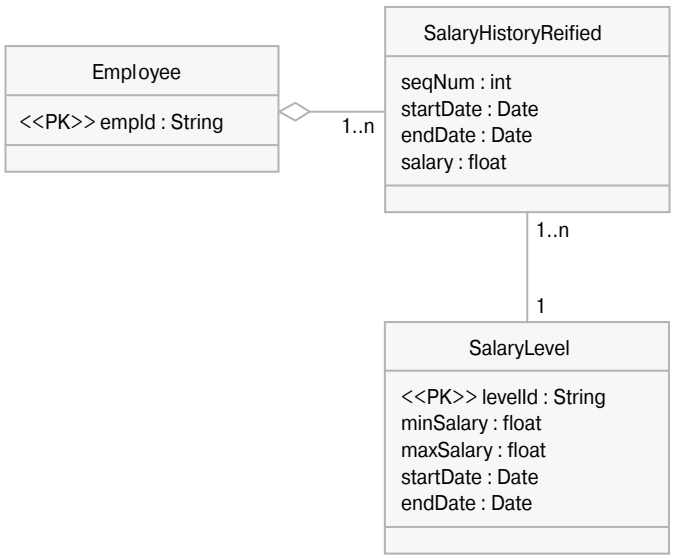


Рис. 5.13. Правильное решение с помощью материализованного класса

Обратите внимание на то, что модель, приведенную на рис. 5.13, следует уточнить. В частности, весьма вероятно, что предположение "диапазоны окладов для заданного уровня никогда не изменяются" в будущем будет ослаблено.

Контрольные вопросы 5.1

- Назовите наиболее важный механизм расширения в языке UML.
- Как называются имена ролей в языке UML 2.0?
- Какой уровень видимости предусмотрен в языке Java по умолчанию, т.е. когда уровень видимости явно не указан?
- Может ли материализованный класс заменить ассоциативный без потери семантики?

5.2. Углубленное моделирование обобщения и наследования

Существуют три основных вида отношений между классами: ассоциация, агрегация и обобщение. Обобщение и наследование обсуждаются в приложении А (см. раздел А.7), но внимательный читатель мог заметить, что полезность обобщения для моделей анализа подвергается сомнению. *Обобщение* (generalization) — полезная и мощная концепция, но она также может стать источником множества проблем из-за сложности механизмов наследования, в частности в крупных программных проектах.

Понятия обобщения и наследования связаны, но не идентичны. Важно понимать разницу между ними. Ценой неточности в определении этих понятий является часто явно демонстрируемое в литературе недопонимание их различия. Конечно, до тех пор, пока это различие не осознано, легко впасть в бессмысленную и безосновательную дискуссию, касающуюся доводов за и против обобщения и наследования.

Обобщение — это семантическое отношение между классами. Оно устанавливает, что *интерфейс* подкласса должен включать все (открытые, пакетные и защищенные) свойства суперкласса. *Наследование* — это "механизм, с помощью которого более специфические элементы вбирают в себя структуру и поведение, определенные более общими элементами" (Rumbaugh et al., 2005).

5.2.1. Обобщение и заменимость

С точки зрения семантики моделирования обобщение вводит в модель дополнительные классы, разделяет их на общие и более специфические классы и устанавливает отношения "суперкласс–подкласс". Несмотря на то что обобщение вводит новые классы, оно может уменьшить общее количество отношений *ассоциации* и *агрегации* в модели (поскольку ассоциации и агрегации характерны для более общих классов и подразумевают существование связей с объектами более специфичных классов).

Исходя из требуемой семантики, ассоциация или агрегация может связывать класс с наиболее общим классом иерархии обобщения (см. диаграммы классов на рис. 3.11 и 4.10). Поскольку подкласс может *заменять* родительский класс, объекты подкласса обладают всеми отношениями ассоциации и агрегации суперкласса. Это позволяет зафиксировать ту же семантику модели с помощью меньшего количества отношений ассоциации/агрегации. Хорошая модель подразумевает верный выбор компромисса между глубиной обобщения и уменьшением количества отношений ассоциации/агрегации, являющегося следствием обобщения.

При продуманном использовании обобщение способствует повышению уровня выразительности, понятности и абстрактности системных моделей. Источни-

ком преимуществ обобщения служит *принцип заменимости* (см. раздел 4.2.4 главы 4) — объект подкласса может быть использован вместо объекта суперкласса в любом месте программы, где объект подкласса имеет доступ к объекту суперкласса. К сожалению, существуют такие способы использования механизма наследования, которые могут свести на нет все преимущества принципа заменимости.

5.2.2. Наследование или инкапсуляция

Инкапсуляция (encapsulation) требует, чтобы доступ к состоянию объекта (т.е. его атрибутам) был возможен только через операции интерфейса объекта. Результатом применения инкапсуляции является высокая степень независимости данных, так что изменения, затрагивающие инкапсулированные структуры данных, не требуют внесения изменений в существующие программы. Но в какой мере идея инкапсуляции осуществима в приложениях?

В действительности инкапсуляция *ортогональна* наследованию и возможностям запросов, и поэтому необходим определенный компромисс между инкапсуляцией и последними двумя свойствами. На практике объявить все данные как *закрытые* невозможно.

Наследование подрывает основы инкапсуляции, открывая подклассам непосредственный доступ к *защищенным* атрибутам. Для вычислений, охватывающих объекты, принадлежащие разным классам, может потребоваться, чтобы они были *дружественными* друг другу или содержали элементы, имеющие пакетную видимость. Это еще больше нарушает инкапсуляцию. Следует также помнить, что инкапсуляция касается понятия класса, а не объекта, и в большинстве программных сред (за исключением языка Smalltalk) объект не может скрыть ничего от другого объекта того же класса.

Наконец, пользователи, осуществляющие доступ к базам данных с помощью средств языка SQL, вполне справедливо желают обращаться непосредственно к атрибутам, а не работать с методами доступа к данным, затрудняющими формирование запросов. Это требование особенно справедливо в отношении приложений хранилищ данных, связанных с интерактивной аналитической обработкой запросов (OnLine Analytical Processing — OLAP).

Итак, приложения должны разрабатываться таким образом, чтобы был достигнут требуемый уровень инкапсуляции и при этом достигался компромисс в отношении наследования, незапланированных запросов и вычислительных требований.

5.2.3. Наследование интерфейса

Когда *обобщение* используется с целью обеспечения заменимости, оно может служить синонимом *наследования интерфейса* (*выделения подтипа*, *наследования типа*). Это не только “безопасный”, но и самый желательный вид наследования

(см. раздел А.9 приложения А). Помимо других преимуществ, наследование интерфейса позволяет реализовать множественное наследование в языках, не поддерживающих его непосредственно (например, в языке Java).

Подкласс наследует типы атрибутов и сигнатуру операций (имя операции плюс формальные аргументы). Говорят, что подкласс *поддерживает* интерфейс суперкласса. Реализация унаследованных операций откладывается на более позднее время.

Существует разница между понятиями *интерфейса* и *абстрактного класса* (см. раздел А.9.1 приложения А). Она заключается в том, что абстрактный класс может обеспечить частичную реализацию некоторых операций, в то время как чистый интерфейс откладывает определение всех операций.

Рис. 5.14 соответствует рис. А.26 (см. раздел А.9.2 приложения А). Этот рисунок демонстрирует альтернативную визуализацию интерфейса "леденец на палочке" и код на языке Java.

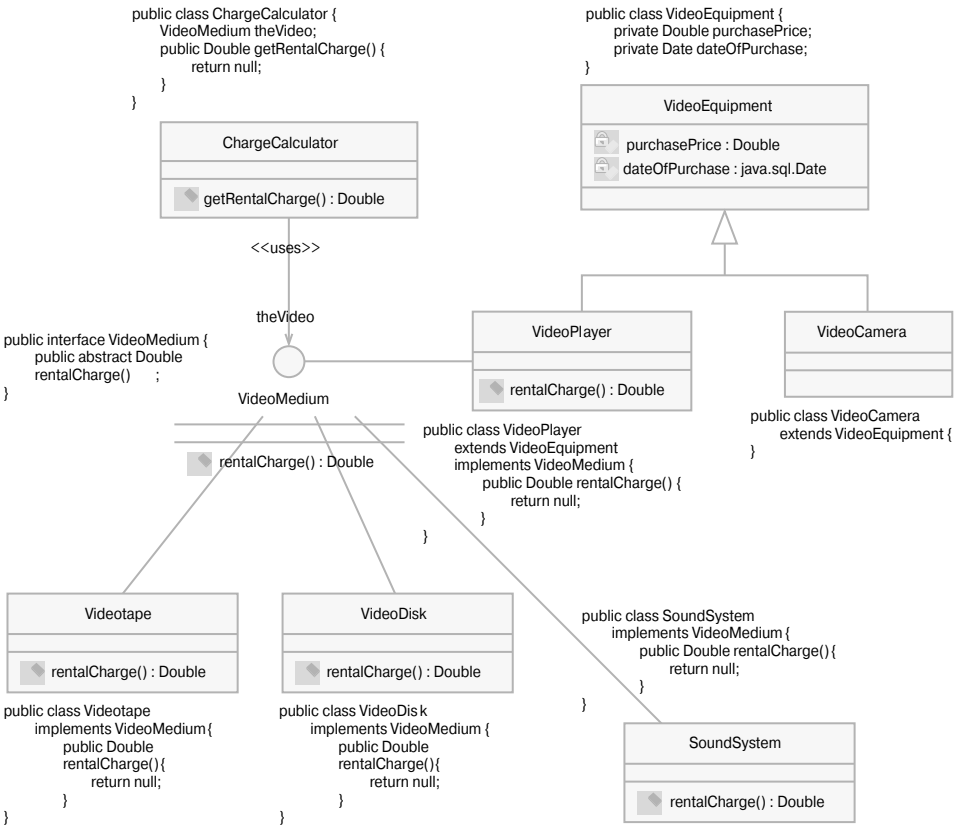


Рис. 5.14. Наследование интерфейса и реализации

5.2.4. Наследование реализации

Как отмечалось в предыдущем разделе, *обобщение* может подразумевать *заменимость* и реализовываться путем *наследования интерфейса*. Однако обобщение может поддерживать (сознательно или нет) *повторное использование кода* (code reuse) с помощью *наследования реализации*. Это очень мощная, иногда даже опасная по силе, интерпретация обобщения. Кроме того, эта интерпретация обобщения принята "по умолчанию".

Наследование реализации (implementation inheritance) — называемое также *выделением подтипа* (subclassing), *наследованием кода* (code inheritance) или *наследованием класса* (class inheritance), — объединяет свойства суперкласса в подклассах и позволяет при необходимости *замещать* их новыми реализациями. *Замещение* (overriding) может означать включение (вызов) метода суперкласса в метод подкласса и расширение его за счет введения новых функциональных возможностей. Оно также может означать полную замену метода суперкласса методом подкласса. Наследование реализации допускает совместное использование *описаний свойств* (property descriptions), *повторное использование кода* и *полиморфизм* (polymorphism).

При моделировании с помощью обобщения необходимо четко отдавать себе отчет в том, какой именно вид наследования подразумевается. Использование *наследования интерфейса* безопасно только в том случае, если оно касается наследования *фрагментов контракта — сигнатуры операций*. Наследование реализации касается наследования кода — наследования *фрагментов реализации* (Harmon and Watson, 1998; Szypersky, 1998). Если тщательно не контролировать и не ограничивать наследование реализации, оно может принести больше вреда, чем пользы. Теперь мы приступим к обсуждению доводов за и против наследования реализации.

5.2.4.1. Правильный способ использования наследования реализации — наследование посредством расширения

Язык UML довольно жестко регламентирует увязку наследования с обобщением и надлежащее использование наследования реализации (Rumbaugh et al., 2005). Единственным правильным способом наследования является инкрементное определение класса. Подкласс обладает бо́льшим количеством свойств (атрибутов и/или методов), чем его суперкласс. Подкласс — это *разновидность* суперкласса. Подобное наследование называется *расширяющим* (extension inheritance).

На рис. 5.23 представлен пример расширяющего наследования. Каждый объект `Employee` (Сотрудник) — это *разновидность* объекта `Person` (Личность), а объект `Manager` — это разновидность объекта `Employee`. Это не означает, что объект `Manager` одновременно является экземпляром трех классов (см. обсуждение множественного наследования в разделе А.7.4 приложения А). Объект `Manager` — это экземпляр класса `Manager`.

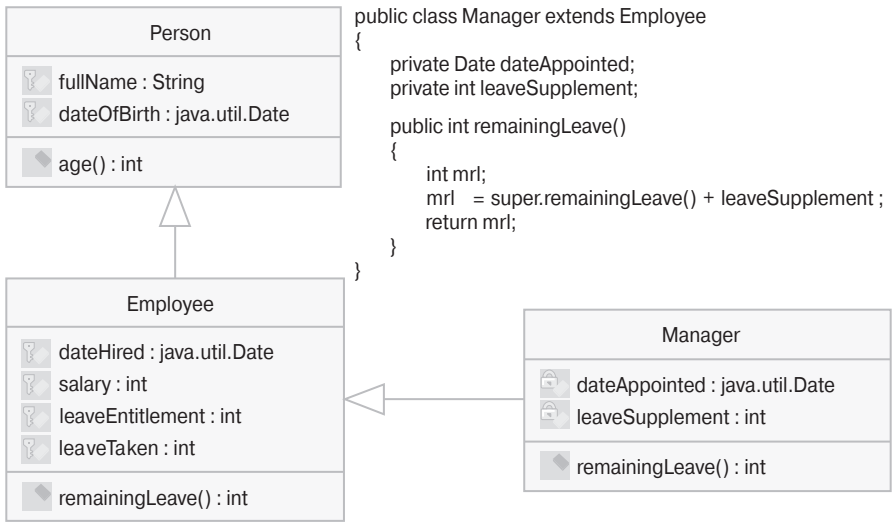


Рис. 5.15. Расширяющее наследование

Метод `remainingLeave()` может быть вызван из объекта `Manager` или `Employee`. Его выполнение зависит от того, из какого объекта он вызван.

Обратите внимание на то, что класс `Person` на рис. 5.15 не является абстрактным. Могут существовать некоторые объекты класса `Person`, которые представляют собой просто некую личность, т.е. не являющиеся сотрудниками (объектами класса `Employee`).

Расширяющее наследование требует внимательного отношения к использованию *замещения* свойств. Допустимым считается только придание свойствам большей определенности (например, ограничение допустимых значений или более эффективная реализация операции), но никак не изменение значения свойства. Если замещение приводит к изменению значения свойства, то объект подкласса нельзя больше подставить вместо объекта суперкласса.

5.2.4.2. Проблематичный способ использования наследования реализации — наследование посредством ограничения

При использовании расширяющего наследования определение подкласса расширяется за счет введения новых свойств. Однако наследование можно также использовать в качестве ограничивающего механизма, посредством которого некоторые унаследованные свойства подавляются (замещаются) в подклассе. Подобное наследование называется *ограничивающим* (restriction inheritance) (Rumbaugh et al., 1991).

На рис. 5.16 приведены два примера ограничивающего наследования. Поскольку наследование нельзя блокировать выборочно, класс `Circle` (Окружность) должен унаследовать свойства главной и дополнительной осей — `minor_axis` и `ma-`

jor_axis — от эллипса Ellipse (Эллипс) и заменить их атрибутом diameter (диаметр). Аналогично, класс Penguin (Пингвин) должен унаследовать операцию fly (летать) у класса Bird (Птица) и заменить ее операцией swim (плавать).

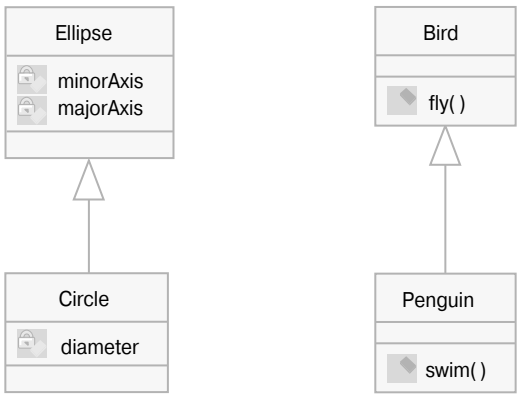


Рис. 5.16. Ограничивающее наследование

Ограничивающее наследование проблематично. С точки зрения обобщения подкласс не включает в себя все свойства суперкласса. Объект суперкласса попрежнему может быть заменен объектом подкласса при условии, что тот, кто использует объект, знает о замещенных (подавленных) свойствах.

При ограничивающем наследовании свойства класса используются (с помощью наследования) для реализации другого класса. Если замещение не носит всеобъемлющего характера, то ограничивающее наследование может принести определенные выгоды. Однако в общем случае ограничивающее наследование вызывает проблемы при сопровождении. Возможна даже ситуация, когда ограничивающее наследование подавляет наследуемые методы, заменяя их пустыми.

5.2.4.3. Неверный способ использования наследования реализации — удобное наследование

Если в процессе системного моделирования оказывается, что наследование нельзя отнести к расширяющему или ограничивающему, оно воспринимается как "плохая новость". Подобный тип наследования встречается в тех случаях, когда два или более класса обладают аналогичными реализациями, но при этом отсутствует отношение таксономии между понятиями, представленными этими классами. Один из классов произвольно выбирается в качестве прообраза другого. Этот вид наследования называется *удобным* (convenience inheritance) (Maciazhek et. al, 1996a; Rumbaugh et al., 1991).

На рис. 5.17 приведены два примера удобного наследования. Класс LineSegment (Отрезок) определен как подкласс класса Point (Точка). Очевидно, что *отрезок* — это не *точка*, и поэтому в данном случае обобщение в том виде, как

оно было определено ранее, неприменимо. Однако наследование все же можно использовать. Конечно, в классе `Point` можно определить атрибуты координат `xCoordinate` и `yCoordinate` и операцию перемещения `move()`. Класс `LineSegment` может унаследовать эти свойства и определить дополнительную операцию изменения размеров `resize()`. Операцию `move()` необходимо заместить. Аналогично, во втором примере класс `Car` (Автомобиль) наследует свойства класса `Motorbike` (Мопед) и добавляет некоторые новые свойства.

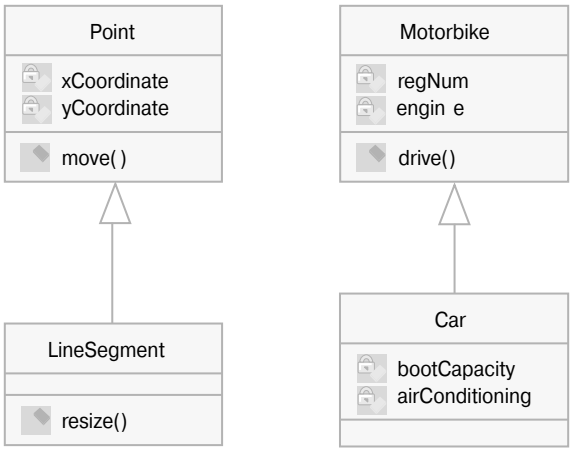


Рис. 5.17. Удобное наследование

Удобное наследование неприемлемо. Оно семантически некорректно и приводит к масштабному замещению. Принцип заменимости, как правило, не работает, поскольку объекты не принадлежат к подобным типам (класс `LineSegment` не является разновидностью класса `Point`, а класс `Car` не является разновидностью класса `Motorbike`).

На практике, к большому сожалению, разработчики часто используют удобное наследование, поскольку многие объектные среды программирования поощряют неразборчивость при использовании наследования реализации. Многие языки оснащены неисчислимыми средствами "усиления программирования" с помощью наследования, в то время как поддержка других свойств объектов (особенно агрегации) отсутствует.

5.2.4.4. Недостатки наследования реализации

Сказанное выше вовсе не означает, что запрещение использования *удобного наследования* служит гарантией успеха. Использование *наследования реализации* — рискованное дело по многим меркам. При отсутствии надлежащего контроля и управления наследование может использоваться чрезмерно или неверно и может создать проблемы, которые требуют первоочередного решения. Это осо-

бенно справедливо для разработки крупномасштабных систем, которые включают сотни классов и тысячи объектов, отличаются динамизмом изменения состояний объектов и эволюционным характером структур классов (как, например, в случае типичных бизнес-приложений).

Основные факторы риска связаны с перечисленными ниже концептуальными трудностями (Szyperski, 1998).

- Изменчивость базового класса.
- Замещение и обратные вызовы.
- Множественное наследование реализации.

5.2.4.4.1. Изменчивый базовый класс

Проблема *изменчивости базового класса* (*суперкласса*) касается ситуации, при которой подклассы достигли определенного уровня зрелости и надежности, в то время как *реализация* их суперкласса (или суперклассов при множественном наследовании) продолжается. Это серьезная проблема в любом случае, особенно в ситуации, когда суперкласс можно получить из внешних источников, находящихся вне контроля коллектива разработчиков системы.

Рассмотрим ситуацию, при которой суперклассы приложения образуют часть операционной системы, СУБД или графического пользовательского интерфейса. Если для разработки своего приложения вы приобретаете объектную СУБД, в действительности вы покупаете библиотеку для реализации типичных функций СУБД, таких как сохранение постоянных объектов, управление транзакциями, обеспечение параллельности, восстановление после сбоев и т.д. Если разрабатываемые классы являются потомками библиотеки классов, то последствия от внедрения новой версии библиотеки непредсказуемы (если при разработке модели наследования для приложения не проявить осторожность, то это несомненно так и есть).

С проблемой изменчивости базового класса трудно справиться, не объявив открытые интерфейсы неизменяемыми или, по меньшей мере, не установив контроль над наследованием реализации из суперклассов. Изменения в реализации суперклассов (для которых может даже отсутствовать исходный текст программ) непредсказуемым образом воздействуют на подклассы прикладной системы. Это справедливо даже тогда, когда интерфейс суперкласса остается неизменным. Ситуация может еще более усугубиться, если изменения также сказываются на интерфейсах. Ниже приведены примеры подобных ситуаций (Szyperski, 1998).

- Изменение сигнатуры метода.
- Разделение метода на два или более новых метода.
- Объединение существующих методов в виде более крупного метода.

Практический вывод из сказанного можно сформулировать следующим образом. Для того чтобы справиться с проблемой изменчивости базового класса, разработчики, проектирующие суперкласс, должны заранее иметь представление о том,

как будут действовать люди при повторном использовании суперкласса сегодня и в будущем. Конечно, узнать это, не прибегая к помощи волшебного шара, нельзя. Как гласит известная шутка: "сумасшествие наследуется — оно достается вам от ваших детей" (Gray, 1994). В разделе 5.3 рассматриваются некоторые альтернативные методы объектной разработки, которые, не будучи основаны на наследовании, все же обеспечивают требуемые функциональные возможности объектов.

5.2.4.4.2. Замещение, нисходящие и восходящие вызовы

Наследование реализации допускает выборочное замещение унаследованного программного кода. Ниже перечислены пять методов, с помощью которых метод подкласса может повторно использовать код его суперкласса.

- Подкласс может наследовать интерфейс и реализацию метода без внесения каких-либо изменений в реализацию.
- Подкласс может наследовать код и включить его (вызвать его) в свой собственный метод с той же сигнатурой.
- Подкласс может наследовать код и затем полностью заместить его новой реализацией с той же сигнатурой.
- Подкласс может наследовать пустой код (т.е. декларация метода отсутствует), а затем ввести реализацию для метода.
- Подкласс может наследовать только интерфейс метода (т.е. мы имеем случай наследования интерфейса), а затем ввести реализацию метода.

Из этих пяти методов первые два доставляют наибольшие трудности в случае склонности программного кода базового класса к эволюции. Пятый метод выказывает полное безразличие к наследованию. Последние два метода представляют особые случаи — четвертый случай тривиален, а пятый не касается наследования реализации.

Пример 5.9. Прямой маркетинг по телефону

Вернитесь к задаче 4, в которой описана система прямого маркетинга по телефону (см. раздел 1.6.4 главы 1), и к примеру 5.4 (см. раздел 5.1.2.1). Модифицируйте модель классов и отношение обобщения между классами `ECampaign` и `EBonusCampaign` (см. рис. 5.6), чтобы включить операции, иллюстрирующие два первых способа повторного использования кода, и покажите нисходящие и восходящие вызовы в отношении обобщения.

Для того чтобы продемонстрировать первый способ повторного использования кода, рассмотрите операцию `computeTicketSold()` в классе `ECampaign`, наследуемую без модификации классом `EBonusCampaign`. Реализация операции `computeTicket()` содержит вызов операции `computeTicketLeft()`. Операция `computeTicketLeft()` существует в классе `ECampaign`, а ее замещенная версия — в классе `EBonusCampaign`.

Для того чтобы продемонстрировать второй способ повторного использования кода и восходящий вызов, рассмотрите операцию getDateClose() в классе ECampaign и ее замещенную версию в классе EBonusCampaign. При вызове из класса ECampaign операция getDateClose() возвращает дату завершения кампании, т.е. объекта класса ECampaign. Однако при вызове из классе EBonusCampaign она возвращает наиболее позднюю дату из двух возможных для классов ECampaign и EBonusCampaign.

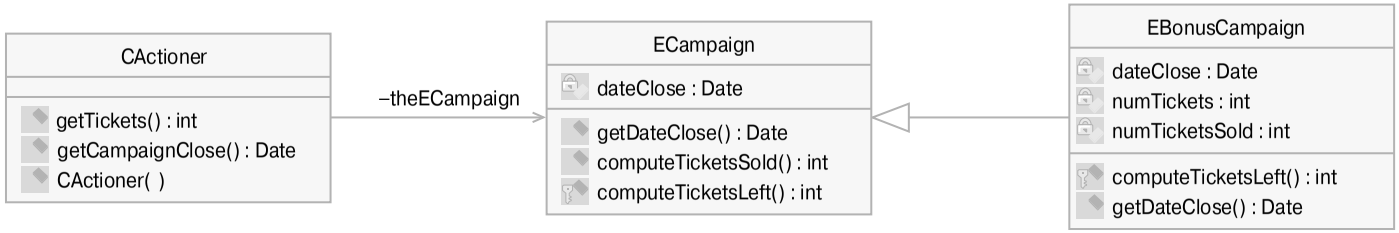
На рис. 5.18 показаны модель и код, описывающие решение примера 5.9. Класс CActioner (Акционер) вызывает операции из классов ECampaign и EBonusCampaign. Он имеет ссылку (-theECampaign) на суперкласс EBonusCampaign, но хранит ее в виде ссылки на тип ECampaign. В реальности присваивание объекта класса EBonusCampaign ссылке на класс ECampaign выполнятся на этапе выполнения программы (Maciaszek and Liong, 2005), а не на этапе компиляции, как показано на рис. 5.18.

Когда операция getTickets() вызывается объектом класса CActioner, она вызывает операцию computeTicketsSold() из класса ECampaign. Здесь используется принцип заменимости — объект класса EBonusCampaign, на который ссылается ссылка -theECampaign, заменяется объектом класса ECampaign (поскольку класс ECampaign не содержит свою собственную замещенную версию операции computeTicketsSold()).

Далее, операция computeTicketsSold() вызывает операцию computeTicketsLeft(). Однако операция computeTicketsLeft() в классе EBonusCampaign замещается. Следовательно, вместо нее будет вызвана ее замещенная версия. Этот пример демонстрирует *нисходящий вызов* (down-call) подкласса из суперкласса. В результате вычисляется количество билетов, оставшихся в объекте класса EBounsCampaign, которое возвращается объекту CActioner.

Обратите внимание на то, что инициализация ссылки theECampaign и последующие нисходящие вызовы образуют зависимость класса CActioner от класса EBonusCampaign на этапе выполнения программы. На этапе компиляции эта зависимость не фиксируется. Класс CActioner имеет ассоциацию с классом ECampaign, но не с классом EBonusCampaign. Такие динамические зависимости трудно отслеживать. Они противоречат принципу EAP модели PCBMER (см. раздел 4.1.3.2 главы 4).

Когда операция getCampaignClose() вызывается объектом класса CActioner, она вызывает операцию getDateClose() из объекта, на который ссылается переменная -theECampaign. Этот объект принадлежит классу EBonusCampaign. Соответственно, вызывается операция getDateClose() из класса EBonusCampaign. Интересно, что операция getDateClose() обеспе-




```
public class CActioner
{
    private ECampaign theECampaign;

    public CActioner()
    {
        theECampaign    = new EBonusCampaign();
    }

    public void getTickets()
    {
        int tnum;
        tnum   = theECampaign.computeTicketsSold();
        return tnum;
    }

    public void getCampaignClose()
    {
        Date cdate;
        cdate    = theECampaign.getDateClose();
        return cdate
    }
}
```

```
public class ECampaign
{
    public Date getDateClose()
    {
        Date cdate      = dateClose;
        return cdate;
    }

    public int computeTicketsSold()
    {
        int tnum;
        tnum   = computeTicketsLeft();
        return tnum;
    }

    protected int computeTicketsLeft()
    {
        return 0; //not executed in the example
    }
}
```

```
public class EBonusCampaign
        extends ECampaign
{
    protected int computeTicketsLeft()
    {
        int tnum;
        tnum   = numTickets - numTicketsSold;
        return tnum;
    }

    public Date getDateClose()
    {
        if (super.getDateClose() < dateClose)
            Date cdate      = dateClose;
        else
            Date cdate      = super.getDateClose();
        return cdate;
    }
}
```

Рис. 5.18. Замещение, нисходящие и восходящие вызовы

чивает расширение метода, определенного в классе `Ecampaign`, — он содержит обращение к суперклассу. Этот пример демонстрирует *восходящий вызов* (*обратный вызов*) суперкласса из подкласса.

Несмотря на простоту обратного вызова, комбинация восходящих и нисходящих вызовов создает запутанные циклические зависимости между классами. Кроме того, эти зависимости возникают на этапе выполнения программ, и, следовательно, их сложно отслеживать.

Пример 5.9 демонстрирует влияние замещения на проблему изменчивости базового класса. Он также показывает, что наследование реализации приводит к возникновению сетей, образованных линиями взаимодействия между объектами, которые, как было показано ранее, в больших системах отличаются неустойчивостью (см. раздел 4.1 главы 4). При наследовании реализации передача сообщений может возникать повсеместно, порождая как *нисходящие*, так и *восходящие вызовы* по иерархии наследования.

Как заметил Шиперский (Szyperswki, 1998): "В сочетании с наблюдаемым состоянием это приводит к циклической семантике, характерной для параллельных систем. Произвольный граф вызовов, формируемый сетью взаимодействующих объектов, разрушает классическую схему иерархии уровней и делает циклические вызовы нормой".

Справедливости ради, следует заметить, что обратные вызовы возможны не только между объектами, связанными отношениями наследования, а всюду, где существуют ссылки между объектами. Вновь приведем замечание Шиперского (Szyperswki, 1998): "При наличии ссылок на объекты… всякий вызов метода может рассматриваться как потенциальный восходящий вызов, и каждый метод может быть потенциально связан с обратным вызовом". Наследование только вносит свой вклад в общую проблему, но, надо заметить, вклад существенный.

5.2.4.4.3. Множественное наследование реализации

Множественное наследование описывается в приложении А (см. раздел А.7.3). В разделе А.9 проведено различие между *множественным наследованием интерфейса* (множественным выделением подтипов) и *множественным наследованием реализации* (множественным наследованием от суперклассов). Множественное наследование интерфейса допускает слияние контрактов интерфейсов. Множественное наследование реализации позволяет объединить *фрагменты реализации*.

Множественное наследование реализации в действительности не создает каких-либо дополнительных проблем наследования реализации. Оно скорее усугубляет проблемы, вызванные изменчивостью базового класса, замещением и обратными вызовами. Помимо требования блокировать наследование любых дублирующихся фрагментов (когда два или более суперкласса определяют одну и ту же операцию), оно может также вызвать необходимость переименовать операции всякий раз, когда повторяющиеся имена совпадают (а в действительности должны обозначать разные операции).

В этом контексте стоит напомнить о присущем системам росте сложности из-за множественного наследования — росте, вызванном отсутствием поддержки *множественной классификации* в объектных системах (см. раздел А.7.4 приложения А). Любые ортогональные ветви наследования, сходящиеся в одном суперклассе, должны объединяться в дереве наследования на более низком уровне иерархии с помощью специально созданных "объединяющих" классов (см. рис. А.22 приложения А).

Проблемы, связанные с множественным наследованием реализации, привели к тому, что в некоторых языках программирования этот механизм был исключен (например, в языке Java). Вместо него язык Java рекомендует использовать *множественное наследование интерфейса* (см. раздел А.9 приложения А).

Контрольные вопросы 5.2

- Какие принципы определяют полезность обобщения?
- Как наследование нарушает инкапсуляцию?
- Какую концепцию можно использовать вместо множественного наследования реализации?

5.3. Углубленное моделирование агрегации и делегирования

Агрегация представляет собой третий метод связывания классов в моделях анализа (см. раздел А.6 приложения А). В сравнении с двумя другими методами (*традиционной ассоциацией* и *обобщением*) агрегации уделялось меньше всего внимания. Тем не менее агрегация представляет собой наиболее мощный из известных методов управления сложностью больших систем с помощью распределения классов по иерархическим уровням абстракции.

Агрегация (и ее более сильный вариант — *композиция*) — это отношение включения. *Составной класс* (composite class) содержит один или несколько *компонентных классов* (component classes). Компонентный класс является элементом одного или более составного класса (хотя они могут существовать самостоятельно). Несмотря на то что агрегация получила признание как фундаментальная концепция моделирования, по меньшей мере одновременно с обобщением, в объектно-ориентированном анализе и проектировании ей уделяется лишь незначительное внимание (за исключением областей приложений наподобие систем мультимедиа).

В средах программирования (включая большинство объектных СУБД) агрегация реализуется так же, как традиционная ассоциация, — с помощью запроса ссылок между составными и компонентными объектами. Несмотря на то что структура времени компиляции для агрегации аналогична структуре ассоциации,

во время выполнения они ведут себя по-разному. Агрегация обладает более строгой семантикой, и ответственность за то, чтобы структуры времени выполнения удовлетворяли этой семантике, лежит (к сожалению) на программисте.

5.3.1. Расширение семантики агрегации

Несмотря на то что современные среды программирования игнорируют агрегацию, методы разработки объектных приложений включают агрегацию среди прочих возможностей моделирования, однако отводят ей последнее место. Кроме того (или вследствие недостаточной поддержки в средах программирования), методы разработки объектных приложений не стремятся придать агрегатным конструкциям строгую семантическую интерпретацию, зачастую трактуя их просто как особую форму ассоциации.

Как указывалось в разделе 4.2.3 главы 4, можно выделить четыре возможных семантики для агрегации (Maciaszek et. al, 1996).

1. Агрегация *ExclusiveOwns* (Безраздельное владение).
2. Агрегация *Owns* (Владение).
3. Агрегация *Has* (Содержит).
4. Агрегация *Member* (Участник).

Язык UML признает только две семантики агрегации, а именно *агрегацию* (*ссылочную семантику*) и *композицию* (*семантику значений*) (см. раздел A.6 приложения A). Теперь будет показано, каким образом можно использовать стереотипы и ограничения, чтобы расширить существующую нотацию UML для представления четырех видов агрегации, указанных выше.

5.3.1.1. Агрегация *ExclusiveOwns*

Агрегацию *ExclusiveOwns* в языке UML можно представить как композицию, стереотипизированную ключевым словом «`ExclusiveOwns`» и дополнительно ограниченную с помощью ключевого слова *frozen* (Fowler, 2003). Ограничение *frozen* (заморожен) применяется к *компонентному классу*. Оно констатирует, что объект *компонентного класса* не может быть *заново соединен* (в течение своего жизненного цикла) с другим составным объектом. Компонентный объект может быть удален вовсе, но не может переключиться на другого владельца.

На рис. 5.19 показаны два примера агрегации *ExclusiveOwns*. Левая часть рисунка представляет пример моделирования агрегации в UML с использованием семантики значений (закрашенный ромб), а правая часть — пример моделирования агрегации в UML с использованием ссылочной семантики (пустой ромб).

Объект `Chapter` (Глава) является частью по меньшей мере одного объекта `CopyrightedBook` (Книга, защищенная авторским правом). Будучи включенным (по значению) в составной объект, он не может быть повторно соединен с другим объектом `CopyrightedBook`. Это соединение заморожено.

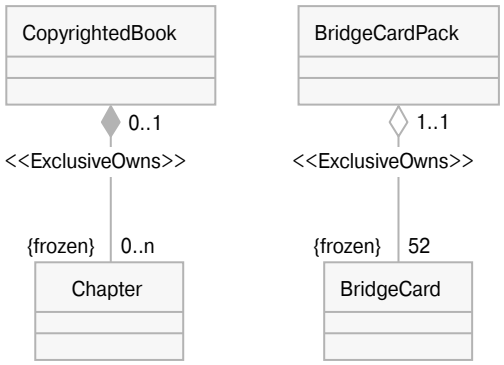


Рис. 5.19. Агрегация ExclusiveOwns

Объект `BridgeCardPack` (Колода карт для игры в бридж) содержит пятьдесят две карты. Для моделирования принадлежности в UML используется ссылочная семантика. Каждая карта для бриджа (объект `BridgeCard`) принадлежит одной колоде карт (`BridgeCardPack`) и не может быть повторно соединена с другой колодой.

5.3.1.2. Агрегация *Owns*

Аналогично агрегации *ExclusiveOwns*, агрегацию *Owns* можно выразить в языке UML с помощью семантики значений композиции (закрашенный ромб) или ссылочной семантики агрегации (пустой ромб). В каждый момент времени компонентный объект принадлежит одному составному объекту, однако он может быть *заново соединен* с другим составным объектом. При удалении составного объекта его компонентные объекты также удаляются.

На рис. 5.20 показаны два примера агрегации *Owns*. Объект класса `Water` (Вода) может быть соединен с другим объектом класса `Jug` (Кувшин). Аналогично, объект класса `Tire` (Шина) может быть соединен с другим объектом класса `Bicycle` (Велосипед). Благодаря зависимости по существованию разрушение объекта класса `Jug` или `Bicycle` распространяется вниз на их компонентные объекты.

5.3.1.3. Агрегация *Has*

Для моделирования агрегации *Has* в языке UML обычно используется ссылочная семантика агрегации (пустой ромб). Агрегация *Has* не содержит зависимости по существованию — удаление составного объекта не распространяется автоматически вниз на компонентные объекты. Агрегацию *Has* отличают такие свойства, как транзитивность и асимметричность.

Пример агрегации *Has* показан на рис. 5.21. Если объект класса `Trolley` (Тележка) *содержит* несколько объектов класса `BeerCrate` (Ящик пива), а каждый объект класса `BeerCrate` *содержит* несколько объектов класса `BeerBottle` (Бутылка пива), то объект класса `Trolley` содержит объекты класса `Beer-`

`Bottle` (*транзитивность*). В то же время объект класса `Trolley` тележка содержит объекты класса `BeerCrate`, но не наоборот (*асимметричность*).

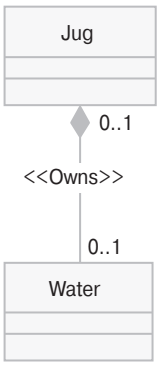


Рис. 5.20. Агрегация Owns

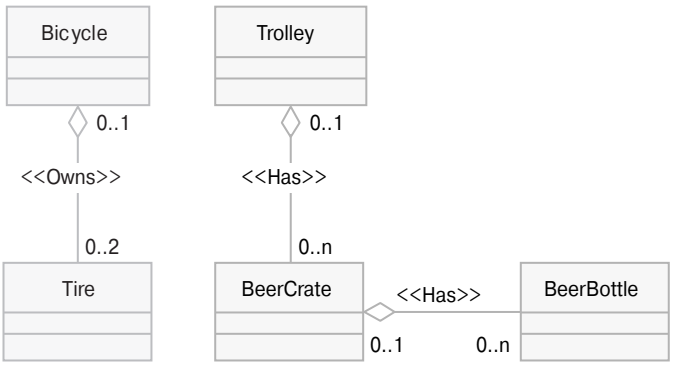


Рис. 5.21. Агрегация Has

5.3.1.4. Агрегация Member

Агрегация *Member* допускает отношения с кратностью "многие ко многим". В отношении свойств зависимости по существованию, транзитивности, асимметрии и ограничения `frozen` не делается никаких специальных предположений. При необходимости любое из этих четырех свойств можно выразить в UML с помощью ограничения. Из-за кратности "многие ко многим" агрегацию *Member* можно моделировать в UML только с помощью ссылочной семантики агрегации (пустой ромб).

На рис. 5.22 показаны четыре отношения агрегации *Member*. Объект класса `JADSession` (см. раздел 2.2.3.3 главы 2), описывающий совещание JAD, состоит из одного объекта класса `Moderator` (Модератора) и одного или нескольких объектов класса `Scribe` (Секретарь), `User` (Пользователь) и `Developer` (Разработчик). Каждый компонентный объект может быть членом нескольких объектов класса `JADSession`.

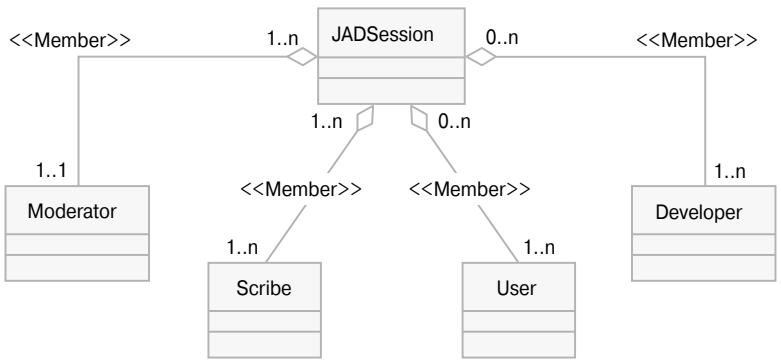


Рис. 5.22. Агрегация Member

5.3.2. Агрегация как альтернатива обобщению

Обобщение (generalization) — это отношение "суперкласс–подкласс". *Агрегация* (aggregation) больше напоминает отношение "супермножество–подмножество". Вопреки этому различию, обобщение можно представить как агрегацию.

Рассмотрим рис. 5.23. Невыполненные заказы клиентов могут быть отложены в ожидании дальнейших действий. Например, заказ может быть выполнен после пополнения запаса или в конкретные сроки, определенные клиентом.

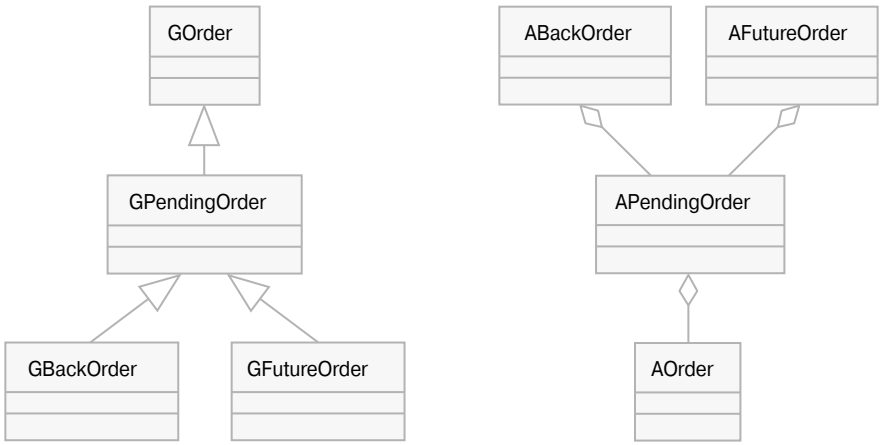


Рис. 5.23. Сравнение обобщения и агрегации

Модель, показанная на рис. 5.23, *слева*, представляет собой обобщение заказов клиентов. Класс `GOrder` (Заказ) может быть классом `GPendingOrder` (Отложенный заказ). Класс `GPendingOrder` может быть классом `GBackOrder` (Невыполненный заказ) или классом `GFutureOrder` (Заказ на будущее). Наследование гарантирует разделение атрибутов и операций вниз по дереву обобщения.

Аналогичную семантику можно смоделировать с помощью агрегации, показанной на рис. 5.23, *справа*. Классы `ABackOrder` и `AFutureOrder` включают в себя атрибуты и операции класса `APendingOrder`, которые в свою очередь включают класс `AOrder`.

Несмотря на то что две модели, показанные на рис. 5.23, отражают одну и ту же семантику, между ними существуют определенные различия. Одно из них вытекает из замечания, что модель обобщения основана на понятии класса, в то время как модель агрегации фактически сконцентрирована на понятии объекта.

Конкретный объект класса `GBackOrder` является также объектом классов `GPendingOrder` и `GOrder`. Для объекта класса `GBackOrder` существует один *идентификатор объекта* (OID). С другой стороны, конкретный объект класса `ABackOrder` состоит из трех отдельных объектов, каждый из которых обладает собственным идентификатором объекта, — собственно объект класса `ABack–`

`Order`, содержащийся в нем объект класса `APendingOrder` и содержащийся в нем объект класса `AOrder`.

Обобщение использует *наследование* для реализации своей семантики. Агрегация использует *делегирование* для повторного использования реализации компонентных объектов. Использование делегирования рассматривается в следующем разделе.

5.3.2.1. Делегирование и системы-прототипы

Вычислительная модель наследования основана на понятии *класса*. Однако в основу вычислительной модели можно положить понятие объекта. Объектно-ориентированная вычислительная модель структурирует объекты в виде *иерархий агрегации* (aggregation hierarchies). Всякий раз, когда *составной объект* (*внешний объект*) не в состоянии выполнить задание самостоятельно, он может вызвать методы одного из его *компонентных объектов* (*внутренних объектов*), — это и называется *делегированием* (delegation).

При подходе, основанном на делегировании, функциональные возможности системы реализуются с помощью включения (*клонирования*) функций существующих объектов во вновь требуемые функции. Существующие объекты трактуются как *прототипы* для создания новых *объектов*. Идея состоит в том, чтобы сначала отыскать требуемые функции в существующих объектах (*внутренних объектах*), а затем реализовать необходимые функции во *внешних объектах*. Внешние объекты запрашивают услуги внутренних объектов по мере необходимости. Системы, построенные подобным способом из существующих объектов-прототипов, называются *прототипами* (prototypical systems).

Объект может быть связан отношением *делегирования* с любым другим идентифицируемым и видимым объектом в системе (Lee and Tepfenhart, 1997). Когда внешний объект получает сообщение и не в состоянии выполнить задачу самостоятельно, он делегирует ее выполнение внутреннему объекту. При необходимости внутренний объект может переслать сообщение любому из своих внутренних объектов.

Интерфейсы внутреннего объекта могут быть видимы или невидимы объектам, отличным от внешнего объекта. Для контроля уровня видимости внутренних объектов можно использовать четыре типа агрегации (см. раздел 5.3.1). Например, внешний объект может раскрыть интерфейс внутреннего объекта как свой собственный в более слабой форме агрегации (например, в такой, как агрегация *Has* или *Member*). При более сильной форме агрегации внешний объект может скрыть интерфейс своего внутреннего объекта от внешнего мира (вводя, таким образом, более строгую форму *инкапсуляции*).

5.3.2.2. Сравнение делегирования и наследования

С помощью *делегирования* можно моделировать *наследование* и наоборот. Это значит, что одни и те же функциональные возможности системы можно реализовать как с помощью *наследования*, так и *делегирования*. Согласие в этом вопро-

се впервые было достигнуто на конференции в США (Орландо, штат Флорида), в 1987 году и теперь известно как Орландское соглашение (Stein et al., 1989).

Мы уже касались изъянов *наследования реализации*. В связи с этим возникает настоятельный вопрос: позволяет ли делегирование избежать недостатков, присущих наследованию реализации. Ответ на этот вопрос не очевиден (Szyperski, 1998).

С точки зрения *повторного использования* кода делегирование сильно приближено к наследованию. Внешний объект повторно использует реализацию внутреннего объекта. Разница состоит в том, что — в случае наследования — после завершения обслуживания управление всегда возвращается объекту, который получает исходное сообщение (запрос на выполнение задачи).

В случае делегирования, после того как управление передано от внешнего объекта внутреннему объекту, оно остается у последнего. Любая *авторекурсия* (self-recursion) должна быть явно запланирована и спроектирована в рамках делегирования. При использовании наследования реализации авторекурсия всегда случайна — она не запланирована и наложена как программная "заплата" (Szyperski, 1998). Одним из нежелательных последствий незапланированного/"заплатанного" повторного использования является проблема *изменчивости базового класса*.

Еще одним потенциальным преимуществом делегирования является то, что разделение и повторное использование можно определить динамически во время выполнения приложения. В системах, ориентированных на использование наследования, разделение и повторное использование обычно определяются статически при создании объекта. При этом достигается компромисс между безопасностью и скоростью выполнения *предусмотренного разделения* наследования и гибкостью *непредусмотренного разделения* делегирования.

В пользу делегирования можно привести тот аргумент, что непредусмотренное разделение более естественно и ближе к человеческому способу мышления (Lee and Tepfenhart, 1997). Объекты объединяются естественным способом для формирования масштабных решений и могут эволюционировать непредвиденным образом. В следующем разделе приводится другая точка зрения на эти же вопросы.

5.3.3. Агрегация и холоны — интеллектуальное орудие

В работах Мацияшека (Maciaszek et al., 1996a, 1996b) с целью преодоления сложности объектных моделей был предложен новый подход для описания архитектуры программного обеспечения, основанный на интерпретации естественных систем, предложенной Артуром Кёстлером (Koestler, 1967, 1978). Центральной концепцией является идея так называемых *холонов* ("holons"), которые интерпретируются как объекты, являющиеся одновременно и частью и целым. Более точно они рассматриваются как саморегулируемые сущности, которые проявляют одновременно взаимозависимые свойства части и независимые свойства целого.

Живые системы обладают иерархической организацией. Структурно они представляют собой агрегации полуавтономных элементов, которые обладают как не-

зависимыми свойствами целого, так и взаимозависимыми свойствами части. По Артуру Кёстлеру, они представляют собой агрегации холонов (от греческого слова *holos* — целое). Суффикс "-он" означает частицу или часть (как в словах протон или нейтрон) (Koestler, 1967).

Части и целое в абсолютном смысле не существуют в живых организмах и даже в социальных системах. Холоны образуют иерархические уровни соответствующей сложности. Например, в биологических организмах различаются иерархии атомов, молекул, органоидов, клеток, тканей, органов и систем органов. Подобные иерархии холонов называются *холархиями* (holarchies).

Каждый уровень холархии скрывает свою сложность от вышележащего уровня. Если смотреть *сверху вниз*, то холон представляет собой нечто законченное и уникальное, целое. Если смотреть *снизу вверх*, то холон представляет собой элементарную компоненту, часть. Каждый уровень холархии содержит множество холонов, например атомы (водород, углерод, кислород и т.д.), клетки (нервные волокна, клетки крови и т.д.).

Если смотреть *изнутри*, то холон предоставляет услуги другому холону. Если смотреть *снаружи*, то холон запрашивает услуги у других холонов. Холархии отличаются незавершенностью. Не существует абсолютных холонов-"листьев" или холонов-"вершин", за исключением тех, которые мы специально обозначаем таким образом для удобства интерпретации. Благодаря этим характеристикам сложные системы могут эволюционировать из простых систем.

Отдельные холоны, таким образом, представляются четырьмя характеристиками.

- Его внутренние правила (взаимодействия между ними могут формировать уникальные шаблоны).
- Самоутверждающаяся агрегация подчиненных холонов.
- Тенденция к интеграции по отношению к высшим холонам.
- Отношения с соседними холонами.

Удачные системы упорядочены в виде холархий, скрывающих сложность в последовательных нижних уровнях, и в то же время обеспечивают больший уровень абстракции в рамках более высоких уровней ее структур. Эта концепция соответствует семантике агрегации.

Агрегация предусматривает разделение — позволяет каждому классу оставаться инкапсулированным и концентрироваться на специфическом поведении (кооперация и услуги) класса способом, который не связан с реализацией его родительских классов (как это имеет место для обобщения). В то же время агрегация позволяет свободное перемещение между стратифицированными уровнями во время выполнения.

Баланс между интеграцией и самоутверждением объектов (холонов) достигается за счет требования, согласно которому объекты должны "признавать интерфейсы друг друга" (Gamma et al., 1995). Инкапсуляция не нарушается, поскольку

объекты взаимодействуют только посредством своих интерфейсов. Эволюция системы облегчается, поскольку взаимодействие объектов не запрограммировано жестко в реализации с помощью механизмов, аналогичных наследованию.

Со структурной точки зрения агрегация дает возможность моделировать большие сообщества объектов с помощью группирования их в виде различных множеств и установления между ними отношения "часть–целое". С функциональной точки зрения агрегация позволяет просматривать иерархию объектов (холонов) сверху вниз и снизу вверх.

Однако агрегация не позволяет моделировать необходимые возможности взаимодействия между холонами одного уровня так, чтобы они могли просматривать структуру изнутри и извне. Этот структурный и функциональный разрыв можно преодолеть с помощью отношений обобщения и ассоциации.

В рамках рекомендуемого подхода агрегация обеспечивает "вертикальное" решение и обобщение "горизонтального" решения для разработки объектных приложений. Агрегация становится преобладающей концепцией моделирования, которая определяет общую структуру системы. Эта структура может быть формализована за счет использования множества проектных шаблонов (Gamma et al., 1995), в особенности поддержки подхода на основе холонов и применения четырех видов агрегации. Мы надеемся, что рассмотренный выше подход послужит читателям интеллектуальным орудием в их собственных изысканиях (Maciaszek, 2006).

Контрольные вопросы 5.3

- Как агрегация реализуется в типичной программной среде?
- Какой вид агрегации должен указываться с ограничением frozen?
- Какая агрегация используется для реализации компонентных объектов?

5.4. Углубленное моделирование взаимодействий

Основы моделирования взаимодействий изложены в главе 3 (см. раздел 3.4) и главе 4 (см. раздел 4.3.3). Рассмотрим более сложные свойства диаграмм взаимодействия и покажем, как их гибкость и точность позволяют гладко перейти на уровень абстракции, требуемый детализированным проектом.

Из двух диаграмм взаимодействий *диаграмма последовательностей* более популярна, чем *диаграмма коммуникаций*. Они выражают одинаковую информацию, но диаграммы последовательностей отражают последовательность сообщений, а диаграммы коммуникаций — отношения между объектами. CASE-инструменты лучше приспособлены для визуализации диаграмм последовательностей и лучше

поддерживают тонкости моделирования взаимодействий в диаграммах последовательностей. По отношению к диаграммам коммуникации это не всегда правда. Замечательным преимуществом диаграмм коммуникации является то, что их можно использовать для презентаций на совещаниях. В таких ситуациях диаграммы коммуникации требуют намного меньше памяти, и их легче исправлять, чем диаграммы последовательностей.

5.4.1. Линии жизни и сообщения

Две из наиболее заметных концепций в диаграммах взаимодействия — *линии жизни* и *сообщения*. *Линия жизни* (lifeline) представляет участника взаимодействия: он символизирует существование объекта в конкретный момент на протяжении взаимодействия. *Сообщение* отражает существование связи между жизненными линиями в ходе взаимодействия. Линия жизни или объект, получающий сообщение, активизирует соответствующую операцию/метод. Момент, в который поток управления фокусируется на объекте, в языке UML 2.0 называется *выполнением спецификации* (ранее он назывался *активацией*).

Обозначения языка UML 2.0 для линий жизни и сообщений продемонстрированы на рис. 5.24. *Линии жизни* изображаются в виде именованных прямоугольников, расширенных вертикальными линиями (как правило, пунктирными). Прямоугольник, символизирующий линию жизни, может иметь имя, чтобы отражать следующие элементы:

- неименованный экземпляр класса — `:Class1`;
- именованный экземпляр класса — `:Class2`;
- класс, т.е. экземпляр метакласса, — `:Class3`, чтобы показать вызовы статических методов внутри классов;
- интерфейс — `:Interface`.

На рис. 5.24 показаны разные типы сообщений, разрешенных при моделировании взаимодействий.

- *Синхронные сообщения*, в которых вызывающие блоки, т.е. блоки, ожидающие ответа, представлены закрашенными стрелками — например, `doX`, `doA`, `doC`, `doD`.
- *Асинхронные сообщения* (asynchronous messages), в которых вызывающей стороной является не блок и, следовательно, допускаются многопоточные вычисления. Они представлены в виде открытых стрелок — например, `doB`.
- *Сообщения о создании объекта* (object creation messages), часто (но не всегда) сопровождаемые ключевыми словами, например `new` или `create`. Они представлены в виде открытой стрелки — например, `new(a,b)`, где `a` и `b` — параметры, передаваемые конструктору класса `Class2`.

- *Ответные сообщения* (reply messages), передающие результаты взаимодействия вызывающему модулю, активизировавшему действие. Они представляются в виде пунктирной линии с открытой стрелкой и часто сопровождаются описанием возвращаемого значения — например, `dValue`.

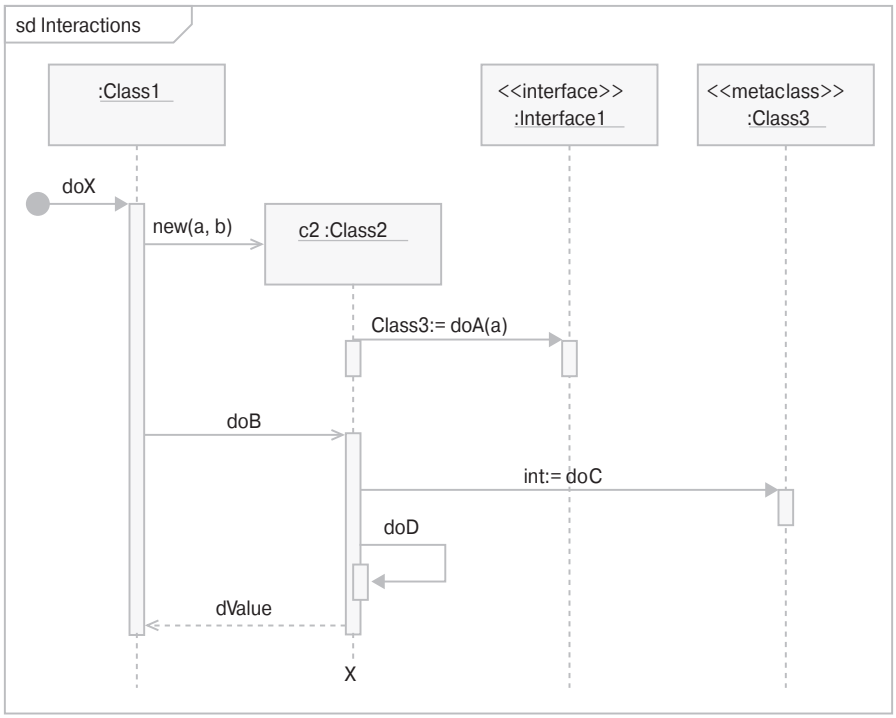


Рис. 5.24. Линии жизни и сообщения

Обратите внимание на то, что *ответное сообщение* — это один из двух способов представления возвращаемого значения. Второй способ — указание возвращаемой переменной в синтаксисе сообщения, например `Class3 = doA()` или `int = doC` (Larman, 2005). В зависимости от уровня абстракции, на котором формируется диаграмма, возвращаемые значения и ответы могут указываться или нет.

На рис. 5.24 также показано *найденное сообщение* (found message) — `doX`, — представляющее собой сообщение без указания отправителя. Иначе говоря, источник найденного сообщения находится за пределами видимости модели.

Разрушение объекта обозначается на диаграмме прописной буквой `X`. Обычно предполагается, что объекты, созданные в модели взаимодействия, уничтожаются в рамках той же самой модели (например, `c2:Class2`). Для языков, в которых не предусмотрена автоматическая сборка мусора (таких, как C++), уничтожение объекта должно инициироваться другим объектом с помощью отдельного сообщения «`destroy`».

Если линия жизни представлена с помощью интерфейса (`:Interface1`) или абстрактного класса, возникает естественное усложнение: вызванный метод выполняется в классе, реализующем интерфейс, или в конкретном классе, производном от абстрактного класса. В обоих вариантах рекомендуется создавать отдельные диаграммы взаимодействия для соответствующих реализаций интерфейса или для каждого полиморфного конкретного класса (Larman, 2005).

5.4.1.1. Учет базовой технологии

Даже если моделирование взаимодействий производится на достаточно высоком уровне абстракции, необходимо учитывать *программную технологию*, выбранную для разработки проекта. Современные языки программирования представляют собой скорее *среды программирования* (programming environment) с готовыми компонентами, библиотеками классов, файлами конфигураций XML, пользовательскими дескрипторами, соединениями с базами данных и т.п. Следовательно, б*о*льшая часть работы, выполняемой приложением, связана не с созданием нового кода, а с повторным использованием программ.

Оказывается, что детали взаимодействия пользовательского кода и среды трудно описать с помощью системы обозначения языка UML, поэтому нужны некоторые элементы, которые могли бы "заполнить пробел". Как минимум, необходимо указать используемую технологию с помощью стереотипов, сопровождающих линии жизни в диаграммах последовательности. (Предполагается, что читатель знаком с основами технологий программирования. В любом случае в примерах, сценариях и упражнениях приводится краткое описание технологий.)

В языке Java основными технологиями создания Web-приложений являются Java Server Pages (JSP), сервлеты и JavaBeans. *Сервлет* (servlet) — это программа на языке Java, которая разворачивается и выполняется на Web-сервере. Как правило, сервлет не имеет графического пользовательского интерфейса, и его можно отнести к уровню контроллера. Графический пользовательский интерфейс поставляется клиентами сервлета, например страницей сервера или аплетом.

Java Server Pages (JSP) — это страницы на языке HTML с фрагментами кода на языке Java. Для того чтобы выполнить приложение, действующее лицо запрашивает страницу JSP, набирая на клавиатуре адрес URL, например `www.myserver.com/myJSP.jsp`. Страницы JSP относятся к уровню презентации.

Компоненты *JavaBeans* — это классы на языке Java, способные хранить данные и выполнять определенные правила, позволяющие пользователю получать и отправлять данные с помощью методов `get()` и `set()` соответственно. Язык Java имеет механизм компонентов, позволяющих страницам JSP формировать значения, автоматически получаемые от компонента и отправляемые компоненты. Компоненты JavaBeans относятся к слою компонента.

Диаграмма последовательностей, соответствующая сценарию, описанному в примере 5.10, показана на рис. 5.25. Проект использует принцип CNP модели

PCBMER, в соответствии с которым перед каждым именем класса указывается первая буква пакета или подсистемы. Например, имя `PRequest` означает, что класс принадлежит пакету или подсистеме презентации.

Пример 5.10. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1). Предположим, что приложение состоит из двух Web-страниц: одна — для ввода данных и другая — для вывода результатов. Приложение должно получать текущий валютный курс из базы данных.

Постройте диаграмму последовательностей для приложения, конвертирующего валюту. Разработайте модель для технологии Java, состоящую из страниц JSP, сервлетов и компонентов JavaBeans. Следуйте принципам модели PCBMER, но можете не использовать уровень сущностей (см. раздел 4.1.3.1 главы 4). Показывать параметры и типы возвращаемых значений не обязательно.

Опишем диаграмму последовательностей, представленную на рис. 5.25. Страница `PRequest.jsp` посылает сервлету — классу `CCalculator` — запросы и получает от него ответы. Класс `CCalculator` просит класс `MMediator` выполнить операцию `getRate()`, а класс `MMediator` делегирует этот запрос классу `RQuery`. Класс `RQuery` получает валютный курс из базы данных и возвращает его классу `CCalculator`. Класс `CCalculator` создает и заполняет экземпляр класса `BBean`. Кроме того, класс `CCalculator` объявляет страницу JSP (в нашем примере — `PResult`), на которую должны выводиться результаты. Страница `PResult` обращается к данным, хранящимся в компоненте `BBean`, и передает их Web-браузеру. В итоге класс `CCalculator` получает от страницы `PResult` сообщения `startOver()` и передает их странице `PRequest`.

Представление результатов и другие детали, например доступ пользователя и оформление запроса, относятся к сфере ответственности Web-контейнера. Таким образом, класс `CurrencyConverter` просто сообщает Web-контейнеру, что страница `PResult` запрашивает ответ, и просит передать управление странице `PRequest`, когда пользователь хочет начать новые вычисления.

5.4.1.2. Визуализация информации о технологии в моделях взаимодействия

Линии жизни, снабженные стереотипами и ссылками на программную технологию, полезны, но не могут ясно показать, как пользовательский и технологический код взаимодействуют при выполнении приложения. Как указывалось ранее, создание моделей взаимодействия для пользовательского кода без учета технологических

деталей оставляет значительные пробелы в моделях и может оказаться бесполезным. С другой стороны, отражение технологических деталей в моделях взаимодействия может оказаться проблемой, поскольку язык UML не зависит от технологии. Даже если существует технологический профиль языка UML (см. раздел 5.1.1) или он разработан в рамках проекта, мощность и постоянные изменения технологии не позволяют точно показать, что именно происходит в ходе выполнения кода.

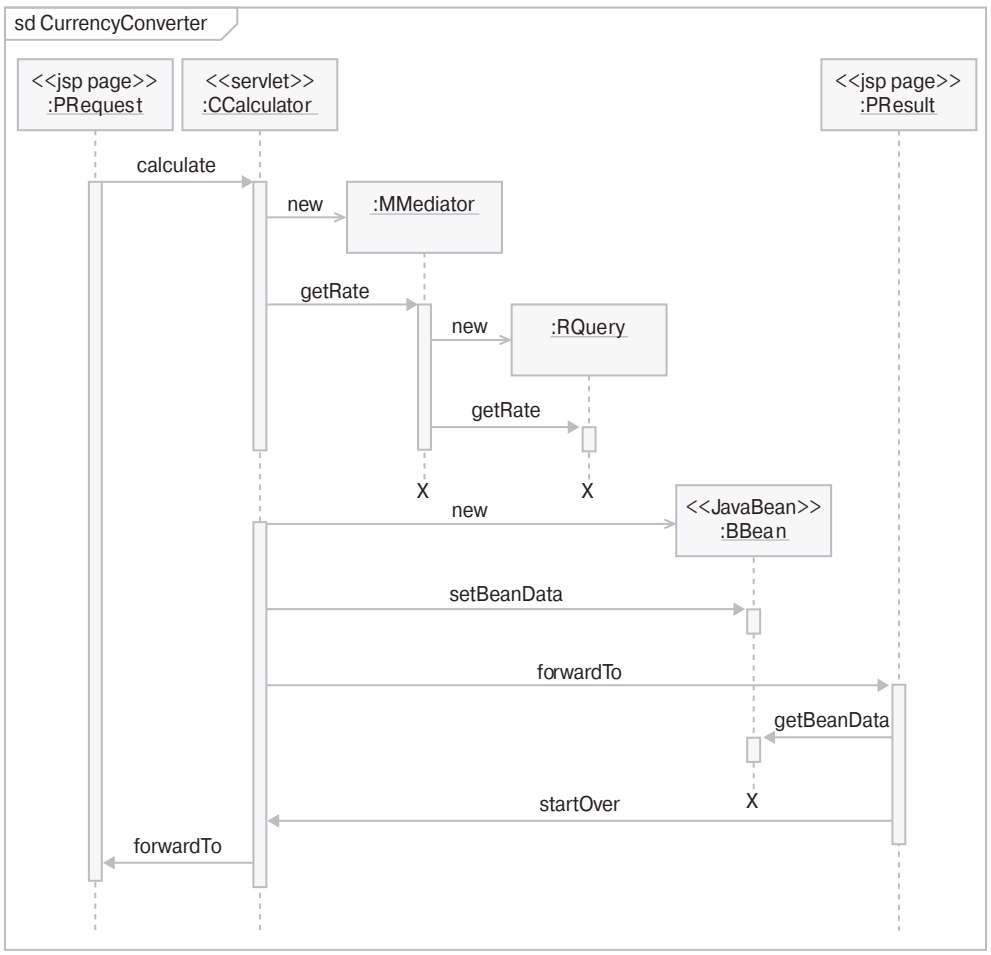


Рис. 5.25. Диаграмма последовательностей, использующая технологию JSP/servlet/JavaBeans для описания системы конвертации валют

С учетом этой оговорки рассмотрим простой пример приложения, состоящего из страницы JSP, создающей форму HTML, и сервлета, реализующего логику приложения и отражающего результаты вычислений с помощью Web-браузера. Технология JSP/сервлет позволяет использовать преимущества библиотеки классов

javax.servlet.*. Кроме того, она использует файл *дескриптора развертывания* (deployment descriptor) web.xml, содержащий инструкции о развертывании приложения. Помимо прочего, файл web.xml содержит имя сервлета, который должен быть вызван в соответствии с адресом URL. На этот адрес пользователи должны сослаться или непосредственно ввести его с клавиатуры в окне Web-браузера, чтобы отобразить начальную страницу JSP. Следовательно, когда пользователи укажут эту страницу, Web-сервер (*контейнер*) будет знать, какой сервлет следует вызвать.

Таким образом, сервлет связан с вводом названия страницы JSP и вычислением результатов. Выполняя эти задачи, сервлет основывается на реализации интерфейсов javax.servlet.* с именами HttpServletRequest (для ввода) и HttpServletResponse (для вывода). Во многих случаях класс сервлета состоит лишь из одного метода — doGet() или doPost(), — способного принять ввод, провести вычисления и сгенерировать вывод. Метод doGet() используется для получения запроса GET, в котором информация посылается как часть адреса URL, а метод doPost() используется для получения запроса POST, в котором информация посылается как часть сообщения, а не адреса URL. Запросы POST обычно используются, когда необходимо передать сервлету значения полей в форме HTML.

Диаграмма последовательностей, реализующая сценарий, описанный в примере 5.11, показана на рис. 5.26. Несмотря на то что эта диаграмма вполне проста и очевидна, некоторые элементы моделирования необходимо описать дополнительно. Например, взаимодействия с действующим лицом The User моделируются с помощью асинхронных сообщений, хотя, за исключением сообщения submitForm, другие сообщения представляют собой лишь вводимые данные. Аналогично, визуализация передачи сообщений от страницы JSP к сервлету, закодированная в файле web.xml, несколько произвольна. Вызовы интерфейсов HttpServletRequest и HttpServletResponse демонстрируют, как интерфейс Java/J2EE API (или, скорее, его реализация с помощью Web-сервера или сервера приложения) облегчает программирование Web-приложений.

Пример 5.11. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1), а также к примеру 5.10 из предыдущего раздела. Рассмотрим упрощенную реализацию системы, конвертирующей деньги из одной валюты (например, австралийские доллары) в другую (например, американские доллары). Пользователь должен ввести с клавиатуры сумму денег, подлежащих конвертации, и текущий курс этих валют. Получив эту информацию, приложение вычисляет сумму в целевой валюте и отражает результаты в трех полях: сумма конвертируемых денег, обменный курс и сумма денег, вычисленных в результате конвертации.

Постройте диаграмму последовательностей, демонстрирующую описанный сценарий. Разработайте модель для технологии Java, состоящей только из страниц JSP и сервлетов. Для того чтобы показать взаимодействие с пользователем, создайте действующее лицо `The User`. Для моделирования отображения между страницей JSP и сервлетом используйте действующее лицо `web.xml`. Кроме того, покажите линии жизни интерфейсов `HttpServletRequest` и `HttpServletResponse`. Показывать параметры методов и типы возвращаемых значений не обязательно.

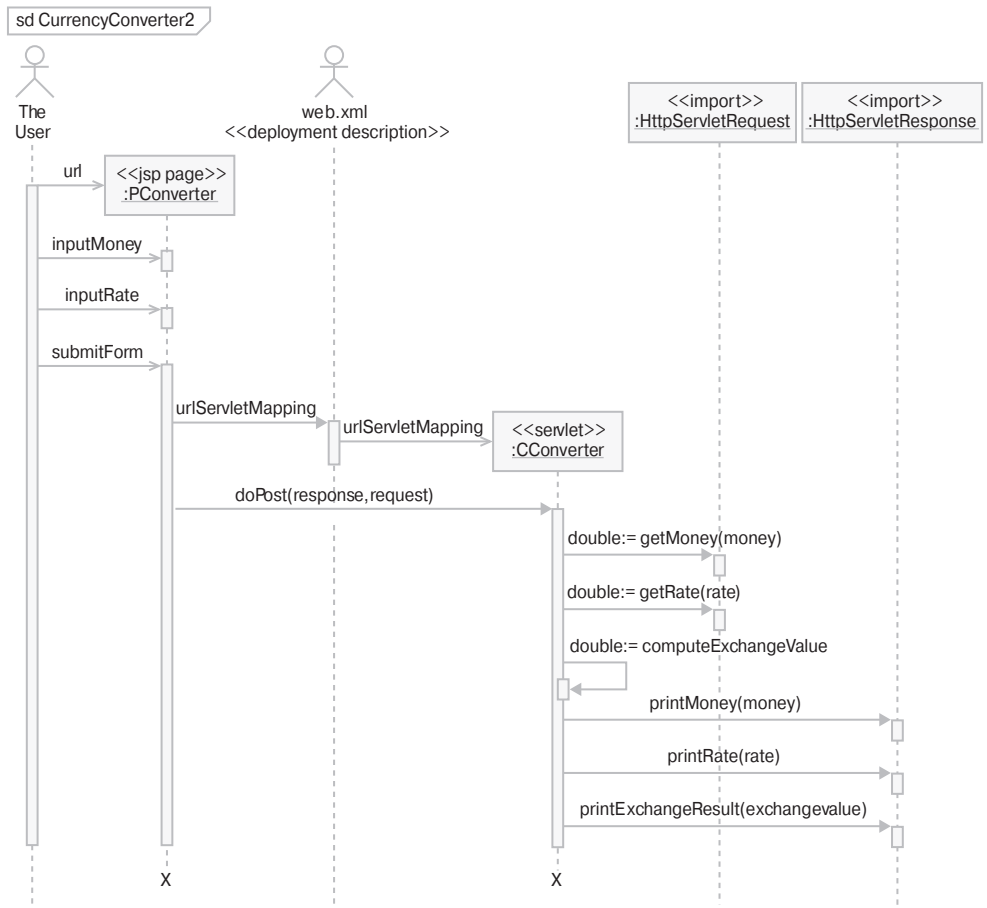


Рис. 5.26. Подробная диаграмма последовательностей, использующая технологию JSP/servlet для описания системы конвертации валют

5.4.2. Фрагменты

Часть взаимодействия называется **фрагментом взаимодействия** (interaction fragment). Взаимодействия могут содержать более мелкие фрагменты взаимодействия, называемые *комбинированными фрагментами* (combined fragments). Семантика комбинированного фрагмента определяется *оператором взаимодействия* (interaction operator). В языке UML 2.0 предусмотрено большое количество операторов, наиболее важные из которых перечислены ниже (Larman, 2005; UML, 2005).

- `alt`. Альтернативный фрагмент логического выражения многовариантного ветвления, выраженный в виде сторожевого условия.
- `opt`. Фрагмент, выполняемый, когда сторожевое условие является истинным.
- `loop`. Фрагмент цикла, повторяемый много раз в зависимости от условия цикла.
- `break`. Фрагмент прерывания, который выполняется вместо оставшегося вложенного фрагмента, если условие выхода является истинным.
- `parallel`. Параллельный фрагмент, допускающий перемежающееся выполнение функций.

На рис. 5.27 показано, как представляются фрагменты на диаграмме последовательностей. В модели показан *альтернативный фрагмент* — `alt`, который содержит *опциональный фрагмент* — `opt`. *Опциональный фрагмент* выполняется только в условии `else` альтернативного фрагмента и только если сторожевое условие `y<=0` является истинным.

Пример 5.12. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1). Рассмотрим настольную реализацию системы, конвертирующей деньги из одной валюты (например, австралийские доллары) в другую (например, американские доллары). Приложение состоит из одного языка класса Java — `CurrencyConvertor`.

Фрейм содержит три поля: для суммы в австралийских долларах, для суммы в американских долларах и для обменного курса. Кроме того, он содержит три кнопки (`btn`): `USD to AUD`, `AUD to USD` и `Close` (для выхода из приложения). Класс содержит метод `actionPerformed()`. Когда пользователь щелкает на одной из трех кнопок, происходит вызов этого метода из объекта `ActionEvent`. Метод `getSource()`, вызываемый из объекта `ActionEvent`, позволяет системе определить, на какой из кнопок щелкнул пользователь и какие вычисления следует выполнить.

Постройте диаграмму последовательностей, демонстрирующую описанный сценарий. При определении кнопки, на которой щелкнул пользователь, используйте альтернативный фрагмент условной логики.

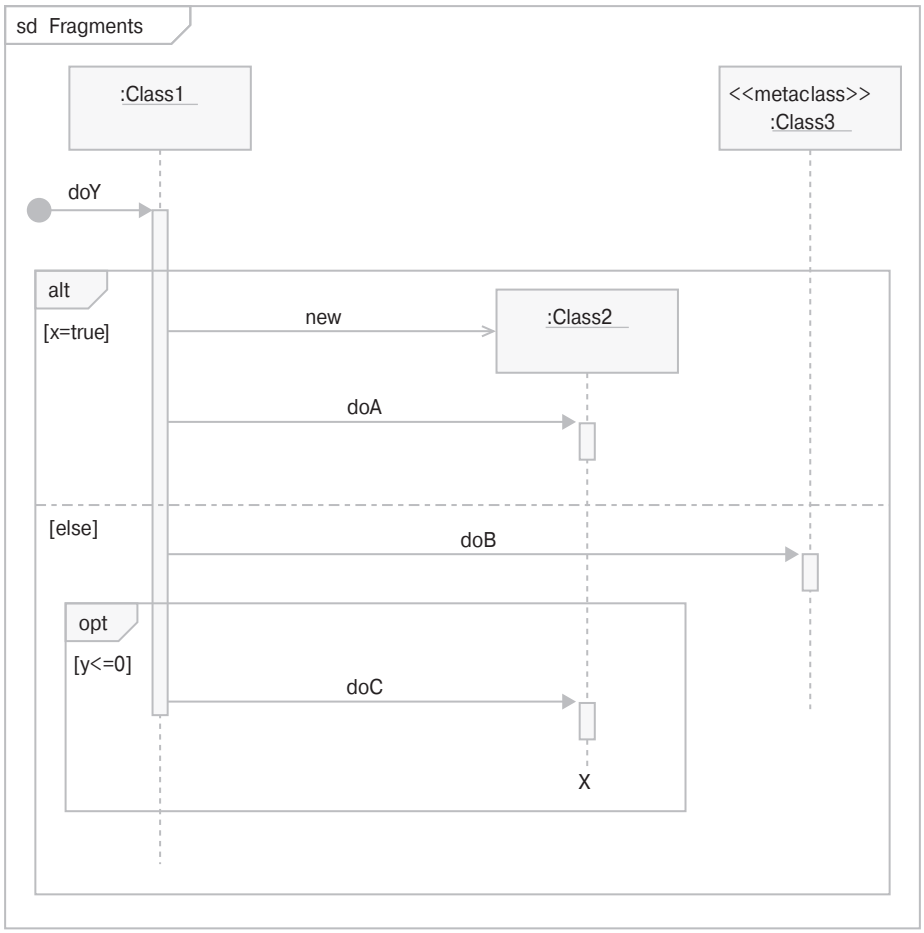


Рис. 5.27. Фрагменты

Диаграмма последовательностей, реализующая сценарий, описанный в примере 5.12, показана на рис. 5.28. Для обработки взаимоисключающих условий, соответствующих трем кнопкам, используется трехвариантный альтернативный фрагмент (условие `[else]` реагирует на событие `Close` выполнением метода `exit`). Все приложение состоит из одного класса — `CurrencyConverter`. Объект `ActionEvent` предоставляется библиотекой Swing.

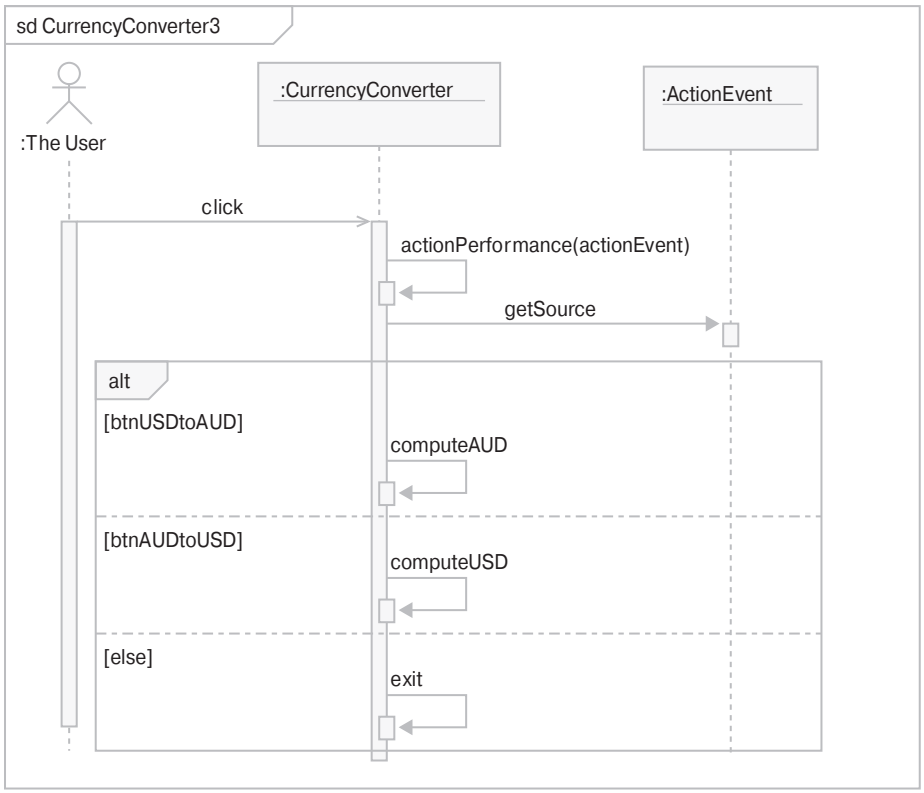


Рис. 5.28. Диаграмма последовательностей, использующая технологию Swing для описания системы конвертации валют

5.4.3. Использование взаимодействия

Кроме комбинированных фрагментов, взаимодействия могут содержать другие взаимодействия. Такие ссылки на взаимодействие называются *использованием взаимодействия* (interaction use). Объемлющее взаимодействие помечается дескриптором `sd` (sequence diagram — диаграмма последовательностей), как показано на рис. 5.24–5.28. Использование взаимодействия помечается дескриптором `ref` (reference — ссылка) и ссылается на другое взаимодействие `sd`, созданное отдельно.

Понятие использования взаимодействия позволяет извлекать и разделять общее поведение. Благодаря этому обеспечивается возможность повторного использования кода. Кроме того, полезно упростить сложную модель взаимодействий, разделив ее на множество отдельно определенных вариантов использования взаимодействия. Простой пример моделирования взаимодействий приведен на рис. 5.29.

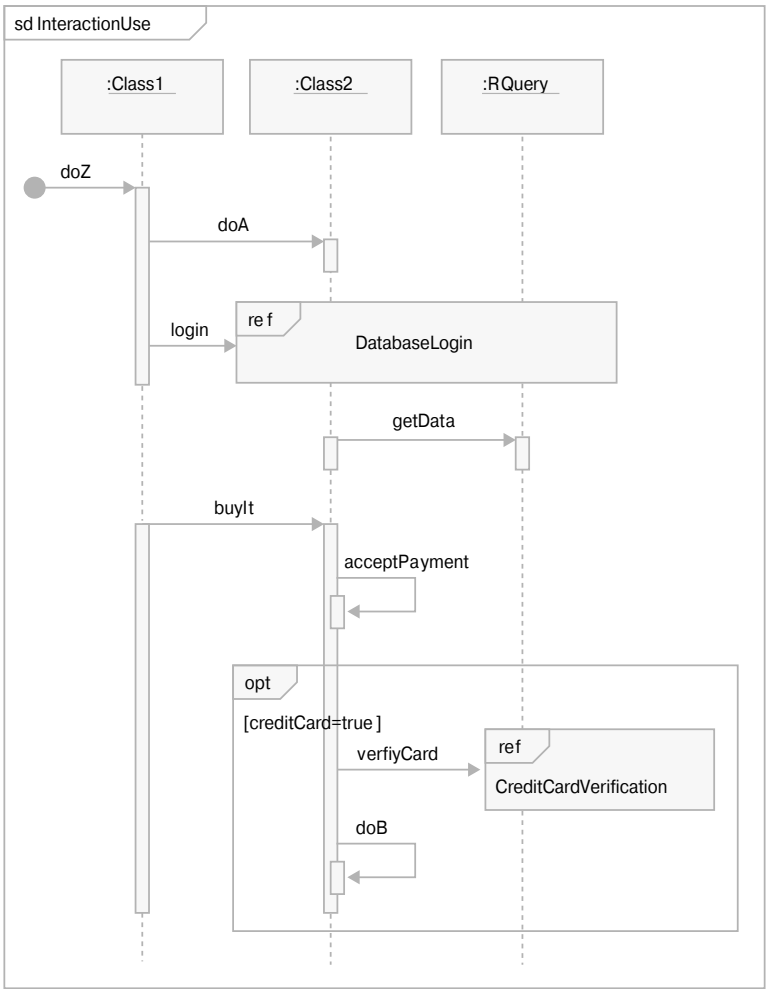


Рис. 5.29. Использование взаимодействий

Контрольные вопросы 5.4

- Какая концепция моделирования взаимодействий используется для определения многопоточного выполнения?
- Какая концепция моделирования взаимодействий используется для определения взаимодействия с неизвестным отправителем?
- Какому архитектурному слою принадлежит сервлет?
- Какой дескриптор используется для обозначения использования взаимодействия?

Резюме

В этой главе было завершено рассмотрение анализа требований и перекинут мостик к анализу и проектированию систем. Была тщательно исследована поддержка объектной технологии для разработки крупномасштабных систем. Местами глава представляется технически сложной, но обеспечивает проникновение в сущность объектной технологии, которое не так просто найти в книгах по анализу и проектированию систем. Это глубокое проникновение во многом помогает раскрыть слабые стороны и недостатки объектной технологии.

Основным методом расширения языка UML является использование *стереотипов*, которые дополняются *ограничениями* и *дескрипторами*. Они позволяют моделировать свойства, выходящие за пределы предопределенных в UML. За счет изобретательности в этой главе удалось широко использовать стереотипы.

Открытая и *закрытая* видимость, рассмотренные в предыдущей главе, обеспечивают только базовую поддержку *инкапсуляции*. *Защищенная* видимость позволяет управлять инкапсуляцией в рамках структур наследования. Понятие *дружественности* и *пакетная видимость* позволяют "прорваться" сквозь инкапсуляцию для обработки специфических ситуаций. *Видимость класса* (в противоположность видимости отдельных атрибутов и операций) представляет собой еще одну важную концепцию, касающуюся наследования.

Язык UML включает дополнительные концепции моделирования, способные повысить выразительность моделей классов. К ним относятся *производные атрибуты*, *производные ассоциации* и *квалифицированные ассоциации*. Одним из наиболее интригующих моментов в моделировании классов является выбор между *ассоциативным* и *материализованным классами*.

Концепция *обобщения* и *наследования* представляет собой в моделировании "палку о двух концах". С одной стороны, она способствует повторному использованию программного обеспечения и повышает выразительность, понятность и уровень абстракции моделей системы. С другой стороны, при неверном использовании всех перечисленных преимуществ она потенциально склонна к саморазрушению.

Концепция *агрегации* и *делегирования* представляет собой важную альтернативу обобщению и наследованию в моделировании. *Делегирование* и *системы-прототипы* обладают дополнительными преимуществами поддержки иерархических архитектурных структур. Абстрактное понятие *холона* дает интересный взгляд на способ построения сложных систем.

Диаграммы последовательностей являются предпочтительным средством моделирования взаимодействия и обеспечивают хорошую поддержку сложных задач моделирования. *Комбинированные фрагменты* были добавлены в модели взаимодействия в языке UML 2.0 для того, чтобы учесть детальную логику программирования. Сложные модели можно создавать с помощью *использования взаимодействий*.

Ключевые термины

Видимость (visibility). “Перечисление, значения которого (открытая, защищенная, закрытая или пакетная) означают, может ли элемент модели, к которой они относятся, быть доступным в рамках объемлющего пространства имен” (Rumbaugh et al., 2005).

Использование взаимодействия (interaction use). “Ссылка на взаимодействие в определении другого взаимодействия” (Rumbaugh et al., 2005).

Комментарий (comment). Аннотация, сопровождающая элемент или набор элементов модели. Комментарий не влияет на семантику модели.

Метакласс (metaclass). “Класс, экземплярами которого являются классы... обычно используемые для создания метамоделей” (Rumbaugh et al., 2005).

Метамодель (metamodel). “Модель, определяющая язык для выражения других моделей” (Rumbaugh et al., 2005).

Меченое значение (tagged value). “Пара “имя–значение”, которую можно присоединить к элементу модели, использующему стереотип, содержащий определение дескриптора” (Rumbaugh et al., 2005).

Ограничение (constraint). “Условие, выраженное на естественном или машинном языке для объявления семантики элемента” (UML, 2005).

Определение дескриптора (tag descriptor). Свойство стереотипа, выраженное в виде атрибута в прямоугольнике класса, содержащем объявление стереотипа.

Производная информация (derived information). “Элемент, который вычисляется на основе других элементов и включается в модель для ее прояснения или для упрощения проектирования, даже если он не имеет дополнительной семантической ценности” (Rumbaugh et al., 2005).

Профиль (profile). “Определяет ограниченные расширения ссылочной метамодели для адаптации метамодели к конкретной платформе или предметной области” (UML, 2005).

Стереотип (stereotype). “Определяет способ расширения существующего метакласса, облегчает использование платформы или специальной терминологии, ориентированной на предметную область, или системы обозначений вместо или в дополнение к используемой в метаклассе” (UML, 2005).

Фрагмент взаимодействия (interaction fragment). “Структурная часть взаимодействия” (Rumbaugh et al., 2005).

Многовариантные тесты

MT1. Что из перечисленного ниже не является механизмом расширения в языке UML?

а. Ограничение.

б. Стереотип.

в. Производный атрибут.

г. Меченое значение.

MT2. Что из перечисленного ниже является синонимом наследования интерфейса?

а. Выделение подтипа.

б. Заменимость.

в. Полиморфизм.

г. Ничто из перечисленного выше.

MT3. Наследование, при котором происходит замещение унаследованных свойств в подклассе, называется...

а. Расширяющим наследованием.

б. Удобным наследованием

в. Ограничивающим наследованием.

г. Все перечисленные варианты не верные.

MT4. В рамках какой концепции всегда появляется авторекурсия?

а. Делегирование.

б. Наследование интерфейса.

в. Пересылка.

г. Наследование реализации.

MT5. Как в языке UML 2.0 называется время, в течение которого поток управления фокусируется на объекте?

а. Использование взаимодействия.

б. Спецификация выполнения.

в. Линия жизни.

г. Все перечисленные варианты не верные.

MT6. Какой из следующих операторов определяет параллельный фрагмент, позволяющий перемежающееся выполнение вложенных функциональных свойств?

а. `Opt`.

б. `Loop`.

в. `Alt`.

г. Ни один из перечисленных выше.

Вопросы

B1. Что в языке UML называется “профилем”? Приведите пример.

B2. Иногда класс позволяет порождать только неизменяемые объекты, т.е. объекты, которые не могут изменяться после реализации. Каким образом подобное требование можно представить в UML-модели?

B3. Объясните различие между ограничением и примечанием.

B4. Обозначают ли термины “инкапсуляция” и “видимость” одно и то же понятие? Объясните вашу точку зрения.

B5. Видимость унаследованных свойств в производном классе зависит от уровня видимости, который предоставлен базовому классу в определении этого производного класса. Какова будет видимость, если базовый класс объявлен как закрытый? Каковы последствия этого факта для остальной модели? Приведите пример.

B6. Понятие дружественности применимо к классу или операции. Объясните, в чем различие этих случаев. Приведите пример (отличный от изложенного в этой книге), когда требуется использование свойства дружественности.

B7. В чем заключаются преимущества использования производной информации для моделирования?

B8. Когда материализованный класс должен заменять ассоциативный? Приведите пример (отличный от приведенного в этой книге).

B9. В чем заключается принцип заменимости? Обоснуйте свой ответ.

B10. Объясните разницу между наследованием интерфейса и наследованием реализации. Используйте в ответе понятия наследования от интерфейсного, абстрактного или конкретного класса.

B11. В чем состоит проблема изменчивости базового класса? Каковы основные причины изменчивости базовых классов?

B12. Объясните различия между агрегациями *ExclusiveOwns* и *Owns*. Какие преимущества при моделировании дает разделение агрегации на два этих вида?

B13. Сравните наследование и делегирование. В чем их схожесть и различие?

B14. Как правило, обработка сводится к перебору всех объектов в коллекции (например, массиве или списке) и посылке каждому из них одного и того же сообщения. Как это можно смоделировать с помощью диаграмм последовательностей? Приведите пример.

B15. Можно ли с помощью сочетания диаграмм деятельности и последовательностей создать полезное обозначение для моделирования? Обоснуйте свой ответ.

Упражнения

У1. Обратитесь к рис. А.14 (см. раздел А.5.2 приложения А). Предположим, что преподаватель, который *руководит* курсом, должен также *вести* этот курс. Измените диаграмму на рис. А.14, чтобы отразить этот факт.

У2. Обратитесь к рис. А.20 (см. раздел А.7 приложения А) и рис. А.21 (см. раздел А.7.1 приложения А). Объедините оба рисунка в виде одной модели классов. Разработайте схему видимости для модели классов. Обоснуйте свой ответ.

У3. Обратитесь к рис. А.16 (см. раздел А.5.4 приложения А). Предположим, что система должна отслеживать успеваемость студентов по изучению нескольких курсов одной и той же дисциплины. Это связано с ограничением, которое гласит, что студент может провалить один и тот же курс не более трех раз (запись на этот же курс в четвертый раз не разрешается). Расширьте диаграмму на рис. А.16 таким образом, чтобы учесть в модели это ограничение. Используйте при этом материализованный класс. Введите в модель любые предположения и поясните их.

У4. Обратитесь к примеру 4.10 (см. раздел 4.2.4.3 главы 4). Перерисуйте диаграмму на рис. 4.10, используя агрегацию вместо обобщения. Приведите аргументы за и против новой модели.

У5. Обратитесь к рис. 4.18 (см. раздел 4.3.3.3 главы 4). Уточните диаграмму последовательностей на рис. 4.18 с помощью комбинированных фрагментов и других усовершенствованных понятий, связанных с моделированием взаимодействий.

У6. Обратитесь к рис. 4.18 (см. раздел 4.3.3.3 главы 4). Уточните диаграмму последовательностей на рис. 4.19 с помощью комбинированных фрагментов и других усовершенствованных понятий, связанных с моделированием взаимодействий.

Упражнение. Регистрация времени

Дополнительная информация

Вернитесь к задаче 6, в которой описывается система регистрации времени (см. раздел 1.6.6 главы 1). Рассмотрим функцию, с помощью которой сотрудник использует возможность запуска секундомера в программе Time Logger, чтобы начать новую запись. Эта функция возлагается на прецедент использования "*Создать запись — запуск секундомера*". Окно графического интерфейса, поддерживающее этот подпоток, показан на рис. 5.30.

Окно секундомера представляет собой немодальное диалоговое окно. Это позволяет пользователю получить доступ к записям о времени в первичном окне браузера, а также к другим свойствам программы Time Logger. Экран дисплея, показанный на рис. 5.30, показывает секундомер в состоянии запуска, в которое он переходит после старта из меню `Stopwatch`.

Окно имеет кнопки для запуска/остановки секундомера. Если секундомер запущен, пользователь может использовать записи списка для выбора и поля `Description` (Описание) для заполнения информации о своих действиях. Если пользователь щелкнул на кнопке `Stop`, программа Time Logger добавляет новую запись в строку браузера.

Параметр `Duration` (Продолжительность) вычисляется по содержимому полей `Start` (Начало), `Now` (Текущее время) и `Pause Duration` (Продолжительность паузы). Поле `Now` (Текущее время) не редактируется. Продолжительность паузы управляется кнопками `Pause` (Пауза) и `End Pause` (Конец паузы).

Кнопка `Reset` (Сброс) сбрасывает показания секундомера без сохранения записи в базе данных. Кнопка `Hide` скрывает секундомер и переводит его в фоновый режим работы. Скрытый секундомер можно вновь показать, используя меню `Stopwatch` (Секундомер).

Кнопки с пиктограммами, на которых изображены знак "плюс", карандаш и знак "минус", позволяют создавать, обновлять и удалять соответствующие элементы списка.

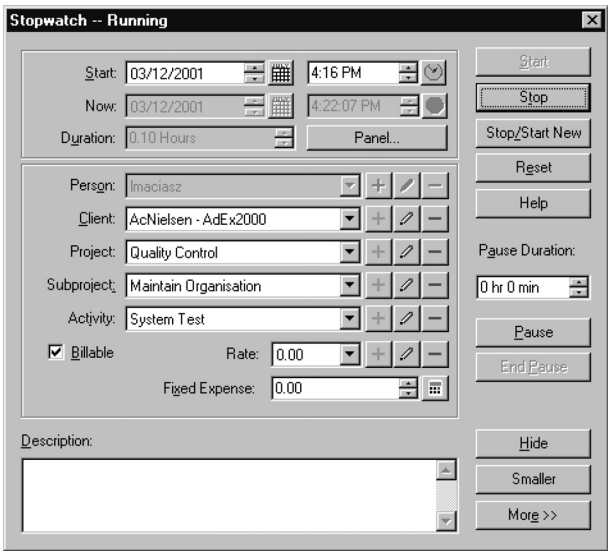


Рис. 5.30. Окно остановки отсчета в системе регистрации времени

PB1. Разработайте высокоуровневую диаграмму коммуникации для подпотока "Создать запись — запуск секундомера". На диаграмме следует показать только основные классы и сообщения, которыми они обмениваются. Нумеровать сообщения не обязательно. Сигнатуру сообщений указывать не обязательно. Проект должен соответствовать модели PCBMER, но классы медиатора и компонентов не являются обязательными. Объясните предположения и потенциальные недоразумения, а также неоднозначные сообщения.

PB2. Основываясь на решении упражнения PB1, разработайте диаграмму классов для подпотока "Создать запись — запуск секундомера". На диаграмме следует показать операции, но не обязательно показывать атрибуты. Классы должны быть связаны необходимыми статическими отношениями. Если между классами нет статических отношений, но они взаимодействуют между собой на этапе выполнения программы, следует использовать отношения зависимости. Объясните предположения и потенциальные недоразумения, а также неоднозначные части модели.

Упражнение. Затраты на рекламу

Дополнительная информация

Вернитесь к задаче 5, в которой описывается система учета затрат на рекламу (см. раздел 1.6.5 главы 1). Вспомните также решения упражнений, приведенных в конце главы 2. Рассмотрите функцию, с помощью которой пользователь поддерживает списки категорий и соответствующих рекламируемых товаров. Эта функция относится к сфере ответственности прецедента использования "Поддержка связи категория–товар". Часть окна графического интерфейса, поддерживающего этот подпоток, показана на рис. 5.31.

Окно `Maintain Category-Product Links` (Поддержка связи категория–товар) состоит из двух панелей: `Categories` (Категории) и `Products for [Active Category]` (Товары для [активной категории]), в данном случае — `Products for consolidated loans`. В окне `Categories` содержится браузер дерева. Категорию можно раскрыть или свернуть, щелкнув на знаке "плюс" или "минус" соответственно. Категории, содержащие подкатегории, идентифицируются пиктограммой папки. Категории на листах дерева (не имеющих подкатегорий) показаны с помощью пиктограммы заметки.

Выбор (подсветка) категории, не имеющей подкатегорий, а также список товаров в категории показаны в правой части окна.

Двойной щелчок на любой категории открывает окно `Update Category` (Обновить категорию). Выпадающее меню содержит пункт `Reassign From` (Перенести), который используется для переноса товара в другую категорию.

ЗР1. Разработайте высокоуровневую диаграмму коммуникации для подпотока "Поддержка связи категория–товар". На диаграмме следует показать только основные классы и потоки сообщений между ними. Нумеровать сообщения не обязательно. Сигнатуру сообщений указывать не обязательно. Проект должен соответствовать модели PCBMER, но классы медиатора и компонентов не являются обязательными. Объясните предположения и потенциальные недоразумения, а также неоднозначные сообщения.

ЗР2. Основываясь на решении упражнения ЗР1, разработайте диаграмму классов для подпотока "Поддержка связи категория–товар". На диаграмме следует показать операции, но не обязательно показывать атрибуты. Классы должны быть связаны необходимыми статическими отношениями. Если между классами нет статических отношений, но они взаимодействуют

между собой на этапе выполнения программы, следует использовать отношения зависимости. Объясните предположения и потенциальные недоразумения, а также неоднозначные части модели.

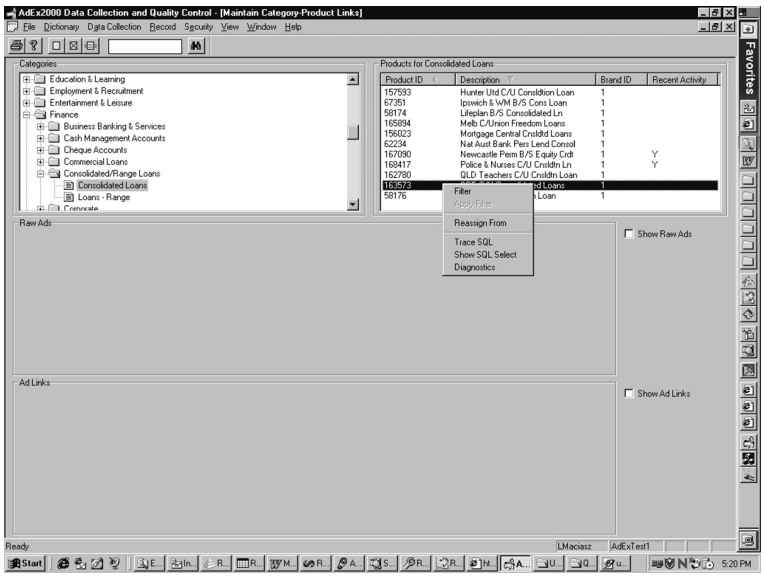


Рис. 5.31. Окно `Maintain Category-Product Links` *в системе учета расходов на рекламу. Источник: компания Nielsen Media Research, Sydney, Australia*

Ответы на контрольные вопросы

Контрольные вопросы 5.1

КВ1. Стереотип.

КВ2. Имена полюсов ассоциации.

КВ3. Пакетная видимость.

КВ4. Да, материализованные классы всегда могут заменить класс ассоциации без потери семантики (но не наоборот).

Контрольные вопросы 5.2

КВ1. Принцип заменимости.

КВ2. Открывая подклассам прямой доступ к защищенным атрибутам.

КВ3. Наследование интерфейса.

Контрольные вопросы 5.3

КВ1. Реализуется как обычная ассоциация с помощью запроса ссылок между составными и компонентными объектами.

КВ2. Агрегация *ExclusiveOwns*.

КВ3. Делегирование.

Контрольные вопросы 5.4

КВ1. Асинхронные сообщения.

КВ2. Найденное сообщение.

КВ3. Уровень контроллера.

КВ4. Дескриптор `ref` (ссылка).

Ответы к многовариантным тестам

МТ1. в

МТ2. а

МТ3. в

МТ4. г

МТ5. б

МТ6. г (этот оператор называется параллельным)

Ответы на вопросы с нечетными номерами

В1

"*Профиль* охватывает часть языка UML и расширяет ее с помощью согласованной группы специализированных стереотипов, например, для бизнес-моделирования" (Fowler, 2003). Для того чтобы отобразить специфику предметной области, стереотипы профиля должны обозначаться новыми графическими пиктограммами. Профили в основном необходимы для *проектного моделирования* и реже — для *аналитического моделирования*. Яркий пример — профиль для создания Web-приложений, опубликованный в работе Коналлена (Conallen, 2000).

Для разработки программного обеспечения изобретены специальные профили. Некоторые из них можно найти по адресу `www.jeckle.de/uml_spec.html#profiles`.

Беглый поиск в Интернете привел к следующими результатам.

- Моделированные данных
  `www.agiledata.org/essays/umlDataModelingProfile.html`
- CORBA (Common Object Request Broker Architecture) — технология создания распределенных объектных приложений, предложенная фирмой IBM.
  `www.omg.org/technology/documents/formal/profile_corba.htm`
- Web-моделирование, схема XSD, моделирование бизнес-процессов
  `www.sparxsystems.com.ua/uml_profiles.htm`
- MOF (Meta-Object Facility — механизм бизнес-объектов) — стандарт группы OMG (Object Management Group) для модельного проектирования
  `mdr.netbeans.org/uml2mof/profile.html`
- Профили для бизнес-моделирования и программирования
  `www-128.ibm.com/developerworks/rational/library/5167.html`
  `www-128.ibm.com/developerworks/rational/library/05/419_soa/`
- Оценка рисков на основе моделирования
  `coras.sourceforge.net/uml_profile.html`
- Встроенные системы реального времени
  `www-omega.imag.fr/profile.php`

B3

Ограничение — это семантическое условие или оговорка, размещенная на элементе моделирования UML. Ограничение может быть изображено графически как текстовая строка, заключенная в фигурные скобки, или как символ заметки (прямоугольник с загнутым правым верхним углом).

Как правило, более сложные ограничения представляются в виде замечания. Поскольку замечание — это отдельный графический элемент, его можно визуально — с помощью отношений — связать с другими элементами моделирования UML.

Заметка является лишь графическим средством выражения ограничения. Символ замечания не имеет самостоятельного семантического значения. Заметку можно использовать для представления информации, не имеющей семантического смысла. Например, замечание может содержать комментарий.

B5

Реализация *видимости* в разных объектно-ориентированных языках может значительно отличаться друг от друга (Fowler, 2004; Page-Jones, 2000). Что касается вопроса, то закрытые свойства данного класса остаются закрытыми и не доступны для объектов производного класса (независимо от того, какой вид наследования используется — закрытый, защищенный или открытый).

Если базовый класс *A* объявлен как закрытый в подклассе *B* (`class B: private A`), то видимость свойств, унаследованных от класса *A*, в классе *B* становится закрытой. Иначе говоря, открытые или закрытые свойства класса *A* становятся закрытыми свойствами класса *B*. В результате дальнейшая специализация класса *B* (например, `class C: public B`) закроет классу *C* доступ к каким-либо свойствам класса *A*. Таким образом, несмотря на то что класс *C* наследует свойства класса *A*, объекты класса *C* не имеют доступа к свойствам класса *A*. Свойства класса *A* становятся невидимыми в классе *C*, хотя определения некоторых свойств в классе *C* могут быть выведены из родительских свойств класса *A*.

Пример, показанный на рис. 5.32, отдаленно напоминает пример из книги Мейерса (Meyers, 1998). Защищенный атрибут `tax_file_number` в классе `Person` становится закрытым атрибутом в классе `Student`. Если `std` — объект класса `Student`, то присваивание `x = tax_file_number` запрещено. Аналогично, вызов `std.sing()` является ошибкой.

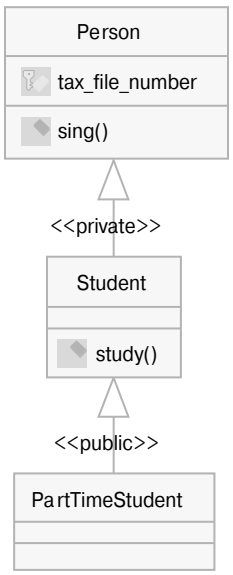


Рис. 5.32. Закрытая видимость класса

Соответствующий код на языке C++ показан ниже.

```
class Person
{
protected:
 int tax_file_number;
public:
 void sing(){};
};
```

```
class Student : private Person
{
};
void main()
{
 Student std;
 int x = std.tax_file_number; // Ошибка компиляции
 std.sing(); // Ошибка компиляции
}
```

Как указано в книге Мейерса (Meyers, 1998): "В противоположность открытому наследованию, компиляторы обычно не конвертируют производный класс (например, `Student`) в базовый класс (например, `Person`), если отношение наследования между классами является закрытым".

К сожалению, *закрытое наследование* является антиподом наследования "разновидность", пропагандируемого в данной книге (например, в разделе 4.2.4.1). Объект класса `Student` больше не является объектом класса `Person`. "Закрытое наследование означает, что класс "реализован посредством" базового класса. Класс *D* объявляется закрытым наследником класса *B*, если программист хочет воспользоваться готовым фрагментом кода, содержащимся в классе *B*, а не потому, что между объектами классов *B* и *D* существует концептуальное отношение" (Meyers, 1998).

Закрытое наследование во многих отношениях является синонимом *удобного наследования*, рассмотренного в разделе 5.2.4.3 и названного неприемлемым вариантом наследования реализации.

B7

Производная информация на самом деле не вносит в модель UML никакой новой информации, а лишь обогащает ее семантику. Она проясняет модель, выявляя информацию, которая в ином случае была бы скрыта от непосредственного наблюдения.

В модели *анализа* производную информацию можно использовать для именования или определения концепции или требования пользователя. В *проектной* модели производную информацию можно использовать для указания на то, что данное значение следует вычислить заново, поскольку величины, от которых она зависит, могут изменяться.

В большинстве практических ситуаций производная информация описывает *ограничение* на существующие свойства, например, что некоторое значение может быть вычислено по существующим величинам. Строго говоря, производная информация может упросить модель, даже если она является излишней.

B9

Принцип заменимости утверждает: "Если переменная или параметр объявлены с типом *X*, любой экземпляр класса, производного от класса *X*, может быть использован как фактический параметр без нарушения семантики его объявления и использования. Иначе говоря, экземпляр производного класса может быть замещен экземпляром базового класса" (Rumbaugh et al., 2005).

Отношение обобщения "является разновидностью" (см. раздел 4.2.4.1 главы 4) и поддерживает *заменимость*. Поскольку отношение "является разновидностью" реализуется с помощью *открытого наследования классов* (см. вопрос B5), принцип заменимости требует использования открытого наследования. Открытое наследование утверждает, что все, что применимо к объектам суперкласса, применимо и к объектам подкласса (подкласс не имеет права отвергать или модифицировать свойства своего суперкласса).

B11

Проблема изменчивости классов состоит в нежелательном влиянии эволюции суперклассов (базовых классов) на всю прикладную программу, содержащую подклассы, производные от суперклассов. Влияние таких изменений практически непредсказуемо, поскольку разработчики суперклассов не знают, как подклассы планируют использовать их свойства.

Если разработчик базового класса не ясновидец, то проблема изменчивости базового класса в объектно-ориентированных приложениях неизбежна. Любые изменения открытых интерфейсов в базовом классе неизбежно повлияют на подклассы. Изменения в реализации операций, наследуемых подклассами, могут иметь значительные последствия и трудно распознаются (особенно в случае реализаций по умолчанию, произвольно переопределяемых в подклассах).

Особый вид проблемы изменчивости класса возникает при *множественном наследовании* (см. раздел 5.2.4.4.3). По существу, любой конфликт, порожденный множественным наследованием, является вариантом проблемы изменчивости базового класса, которая возникает еще до того, как произойдет модификация подкласса.

B13

Наследование — это способ *повторного использования кода* с помощью отношений обобщения. *Делегирование* — это способ *повторного использования кода* с помощью отношений агрегации. В большинстве случаев выбор наследования (обобщения) или делегирования (агрегации) очевиден — семантика "является разновидностью" и требует обобщения, а семантика "содержит" требует агрегации.

Однако, как показано в сложном примере (см. рис. 5.23), обобщение можно реализовать с помощью агрегации. За исключением этой ситуации, наследование следует использовать в сочетании с семантикой "является разновидностью", а делегирование — в сочетании с семантикой "содержит".

Сходство между этими методами заключается в том, что оба они относятся к *повторному использования кода*. Разница между ними состоит в том, что наследование — это способ повторного использования кода в *классах*, а делегирование — способ повторного использования кода в *объектах*. Таким образом, делегирование — более мощный механизм, чем наследование.

Во-первых, делегирование может имитировать наследование, но не наоборот. Во-вторых, делегирование относится к этапу выполнения программы и поддерживает динамическую эволюцию системы, а наследование — статическое наследование, относящееся к этапу компиляции. В-третьих, делегирующий (внешний) объект может использовать как функциональные свойства (реализации операций), так и состояние (значения атрибутов) делегированных (внутренних) объектов, в то время как производный объект не наследует состояние.

B15

Комбинация диаграмм деятельности и последовательностей может повысить выразительность логики более сложных программ. Необходимость такого комбинированного моделирования в языке UML 2.0 была осознана с появлением диаграмм обзора взаимодействий.

Диаграмма обзора взаимодействий разделяет проблемную область на использование взаимодействия и комбинированные фрагменты (из диаграмм последовательностей), а также на конструкции потоков управления (обозначение ветвления и разделения из диаграмм деятельности). Управляющие конструкции позволяют делать обзор потока управления, а более специфичные вычислительные узлы моделируются с помощью использования взаимодействия и комбинированных фрагментов. Использование взаимодействий и фрагменты допускают анализ деталей взаимодействия.

Объяснение упражнений с нечетными номерами

У1

Легкое решение этого вопроса можно получить, наложив *ограничение* на две ассоциации (рис. 5.33). Ограничение `{subset}` означает, что преподавателем, ответственным за курс, должен быть один из преподавателей, читающих лекции на этом курсе.

У3

Решение, представленное на рис. 5.34, получено путем превращения класса ассоциации, показанного на рис. A.16 (см. раздел A.5.4 приложения A), в независимый класс `Assessment`. Замечание к ограничению (про правило “не более трех ассоциаций”) было добавлено в модель и связано с классом `Assessment` и его ассоциациями с двумя другими классами.

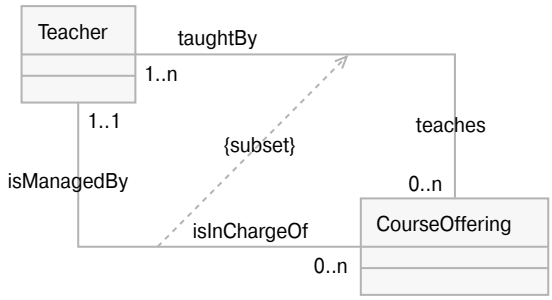


Рис. 5.33. Ограничение, наложенное на ассоциации

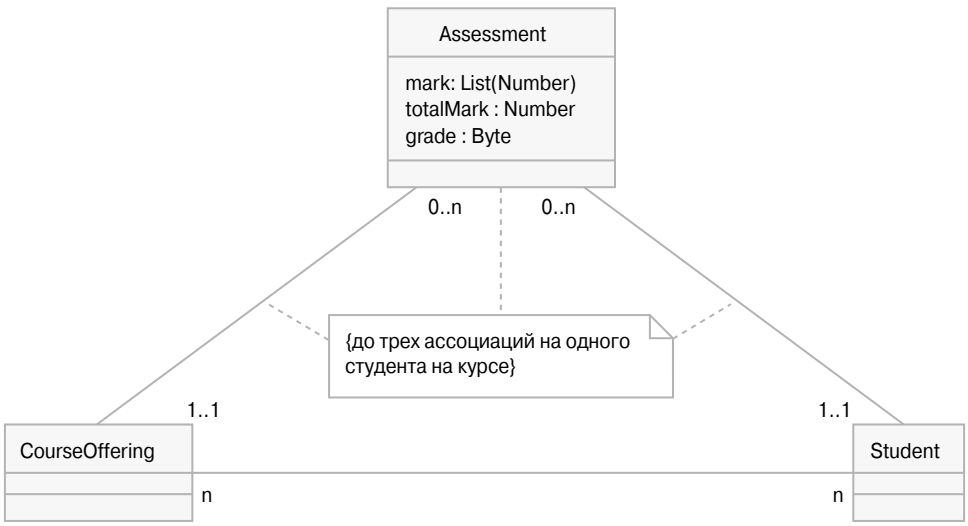


Рис. 5.34. Моделирование ограничения

У5

На рис. 5.35 показана улучшенная диаграмма последовательностей с централизованным решением примера, касающегося системы управления зачислением в университет. Эта модель использует три комбинированных фрагмента — альтернативу, цикл и опцию.

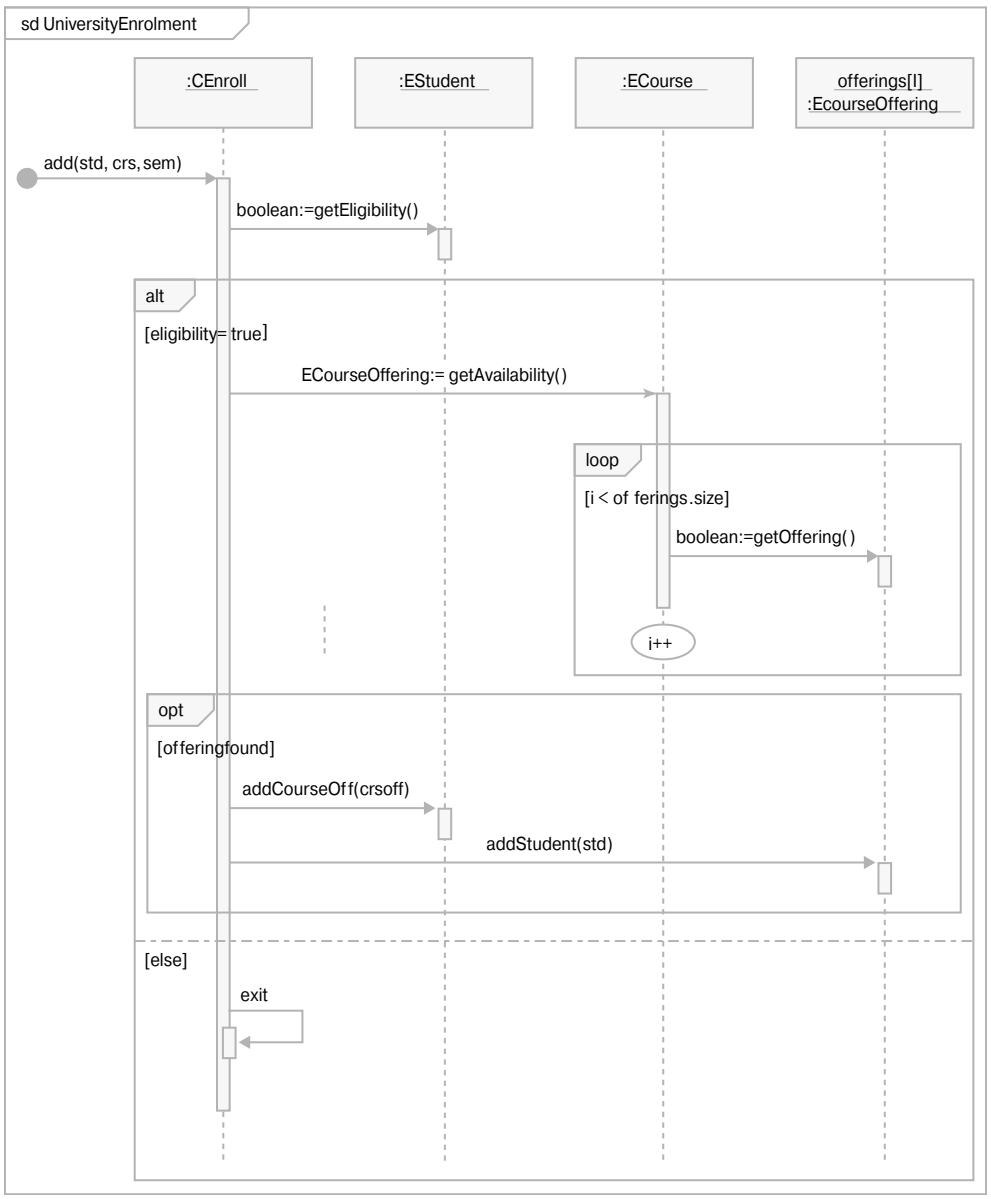


Рис. 5.35. Диаграмма последовательностей с комбинированными фрагментами

Объяснение упражнений. Регистрация времени

PB1

На рис. 5.36 приведена модель коммуникации для подпотока, управляющего секундомером. Выполнение подпотока начинается, когда пользователь активизирует секундомер из меню :CMenuItem. В этот момент открывается диалоговое окно :PStopwatch, предлагающее ввести данные. Для этого оно инициирует экземпляр :CStopwatchInitializer, который должен выполнить операцию refreshView().

Для того чтобы объект сущности :ETimeRecord появился на экране, он должен выполнить операцию getTimeRecord(). Это действие порождает большое количество сообщений другим объектам сущностей, чтобы они выполнили операции getDate(), getTime(), getPerson(), getClients(), getProjects() и getActivates().

Объекты — получатели этих сообщений — обращаются к классу :RReader, чтобы извлечь информацию из базы данных. Наша модель носит упрощенный характер и показывает, что услугами класса :RReader пользуется только класс :ETimeRecord.

Работая в диалоговом окне, пользователь может редактировать многие поля. Для того чтобы редактировать пункты списка, пользователь обращается к объектам :CMouseEvent. Выбранный пункт передается методу pickCurrent() в соответствующем контейнерном объекте (:EClientList, :EProjectList или :EActivityList). Как и ранее, доступ к базе данных инкапсулирован в объекте :RReader.

В заключение запрос stop(), поступающий от класса :CButton к классу :CStopwatchInitializer, порождает передачу сообщения saveTimeRecord() классу :ETimeRecord. Генерирование SQL-запроса для модификации базы данных возложено на класс :RUpdater.

PB2

На рис. 5.37 показана модель классов для подпотока, управляющего секундомером. Эта модель построена на основе решения, полученного в разделе PB1. Диаграмма упрощает модель PCBMER. В центре проекта находится управляющий класс CStopwatchInitializer, но основные задачи, связанные с установкой и определением времени, координируются классом сущности ETimeRecord. Этот класс выполняет множество операций с помощью своих компонентных классов. Для связи с базой данных класс ETimeRecords использует классы RReader и RUpdater.

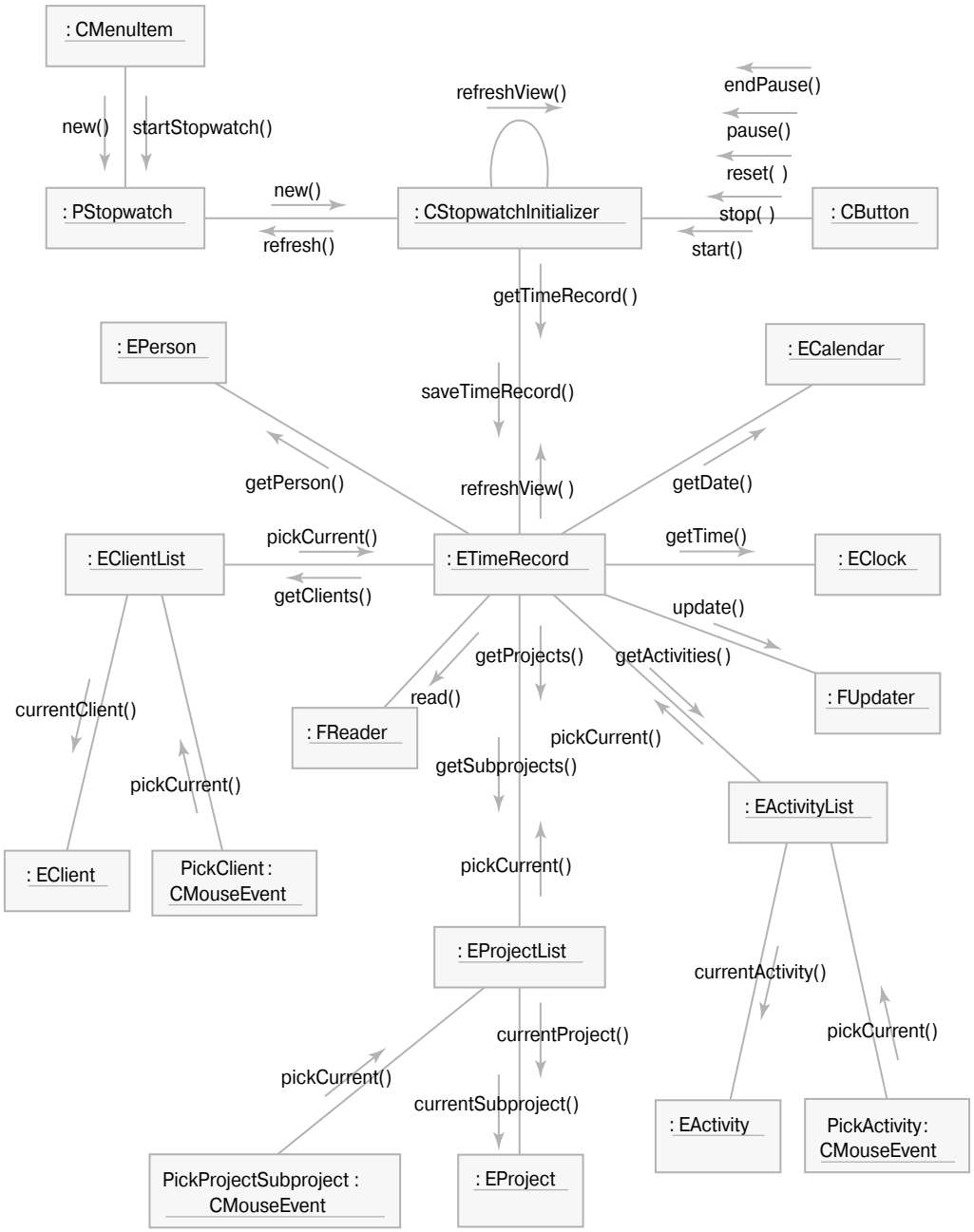


Рис. 5.36. Модель коммуникации для секундомера в системе регистрации времени

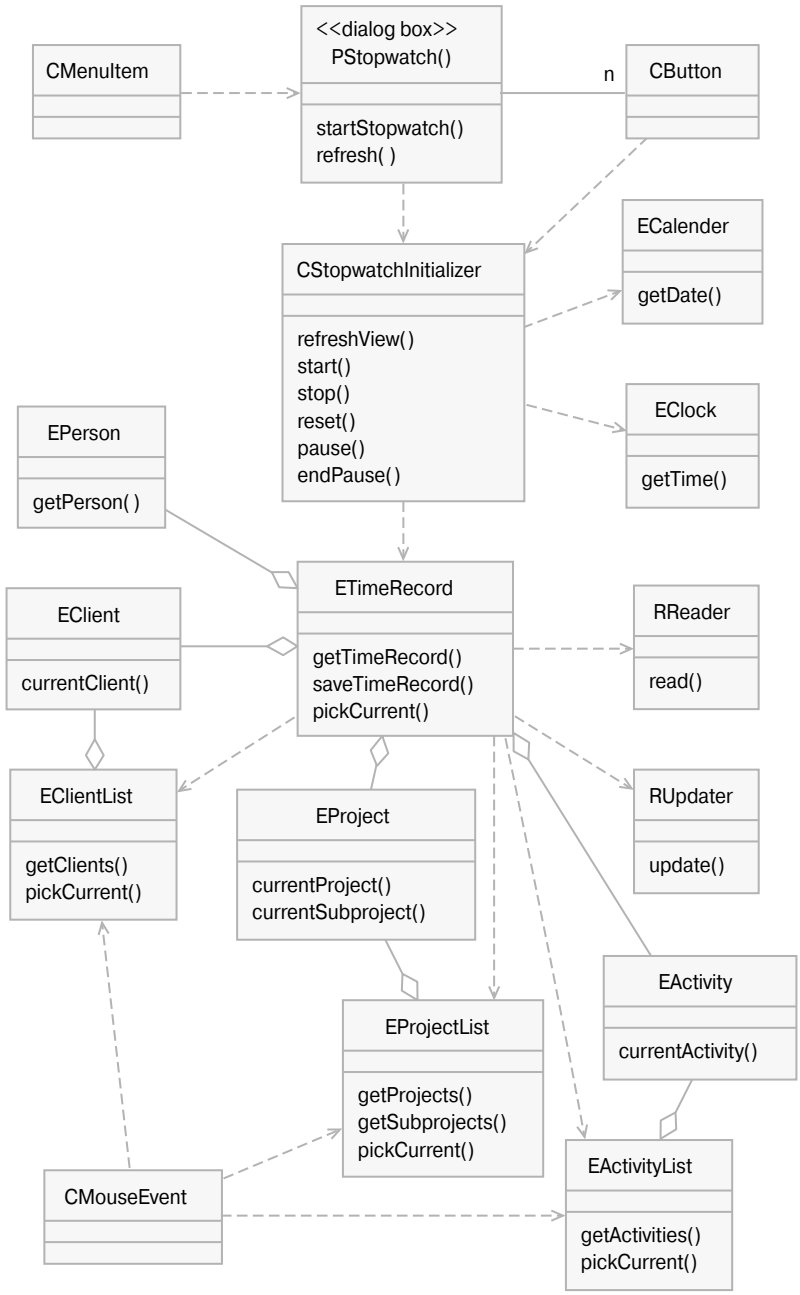


Рис. 5.37. Модель классов для секундомера в системе регистрации времени