Глава 5

Углубленный анализ

В предыдущей главе мы нарисовали картину визуального объектно-ориентированного моделирования 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 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.

5.1. Углубленное моделирование классов

Концепции моделирования анализа, которые рассматривались до сих пор, были достаточны для выработки завершенных моделей анализа. Однако обеспечиваемый этими концепциями уровень абстракции не исчерпывает всех возможных деталей, допустимых 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, ограничения, производную информацию, видимость, квалифицированные ассоциации, ассоциативные классы, параметризованные классы и некоторые другие. Эти концепции необязательны. Многие модели могут быть вполне приемлемы без них. Их применение требует осторожности и точности, так чтобы любой, кто попробует разобраться с моделями впоследствии, мог без труда понять намерения разработчика.

5.1.1. Стереотипы

Стереотип (*stereotype*) расширяет существующий элемент моделирования UML. Вносит некоторые изменения в семантику существующего элемента. *По существу* не является новым элементом моделирования и также не изменяет структуру UML, а только обогащает содержание существующей нотации. Он позволяет расширить и настроить метод. Существуют различные способы применения стереотипов.

Обычно стереотипы *помечены* в модели с помощью имени, заключенного в особые скобки (которые называются во французском языке *guillemets* и выглядят как двойные угловые кавычки), например, «`global`», «`PK`», «`include`» (Следует иметь в виду, что эти кавычки — один символ, а не два. *Прим. ред.*). Возможно также представлять стереотип с помощью *пиктограммы*.

Некоторые распространенные стереотипы являются *встроенными* — они заранее определены в UML. Некоторые CASE-средства включают готовые пиктограммы для встроенных стереотипов. Большинство CASE-средств обеспечивают возможность создания новых пиктограмм по желанию аналитика. На рис. 5.1 показан пример класса, обозначенного как стереотип с помощью пиктограммы, метки, и тот же класс, не являющийся стереотипом.

Рис. 5.1. Пиктограммы и метки стереотипов

Ограничение, гласящее, что стереотип расширяет семантику, но не структуру UML, довольно несущественно. Главная отличительная черта любой объектно-ориентированной системы заключается в том, что она *полностью* состоит из *объектов* — класс является объектом, атрибут также является объектом, метод — также объект и т.д. Следовательно, превращение класса в стереотип может привести в итоге к созданию нового элемента моделирования, который вводит новую категорию объектов.

Например, в разд. 7.6.1 и 9.2.1 показано, как можно использовать стереотипы для создания нового метода моделирования — диаграмм навигации по окнам, диаграмм навигации по программе. При использовании подобного способа стереотипы расширяют UML для конкретной цели. А цель эта зачастую состоит в том, чтобы работать моделированием проектирования, а не моделированием анализа. Модели проектирования согласуются с платформой реализации. Следовательно, они должны работать с элементами моделирования, надлежащим образом представляя эту платформу.

Целевой набор стереотипов, направленный на решение вопросов моделирования проектных решений, называется *профилем* (*profile*). В будущем стандарты UML могут включать профили для распространенных проектных решений, таких как базы данных или CORBA (Common Object Request Broker Architecture — Общая архитектура брокера объектных запросов).

5.1.2. Ограничения

Стереотипы часто путают с ограничениями. Конечно, иногда различие между этими двумя понятиями провести довольно трудно. Зачастую *стереотип* используется для введения в модель нового *ограничения* – того, которое имеет смысл для моделирования, но не поддерживается непосредственно в UML.

С каждым элементом моделирования может быть связано ограничение, либо он может быть превращен в стереотип. На диаграмме модели показывают только простые ограничения. Их показывают в виде текста, заключенного в фигурные скобки (или в виде символа примечания – см. разд. 5.1.3), и они графически и семантически ограничены модельным элементом. Более доскональные ограничения (слишком большие для графической модели) хранятся в репозитории CASE-системы обычно как документ, созданный некоторым текстовым процессором.

Пример 5.1. Телемаркетинг

Обратитесь к примеру 4.7 (разд. 4.2.1.2.3). Вернитесь к той части требования 2, которая гласит: “Все билеты пронумерованы. В рамках кампании все билеты обладают уникальными номерами.”

В решении к примеру 4.7 (рис. 4.4) мы не зафиксировали приведенное выше ограничение (недостает включения атрибута `campaign_code` (вместе с `ticket_number`) в класс `CampaignTicket` как часть составного первичного ключа).

Наша задача состоит в том чтобы, смоделировать ограничение на диаграмме классов.

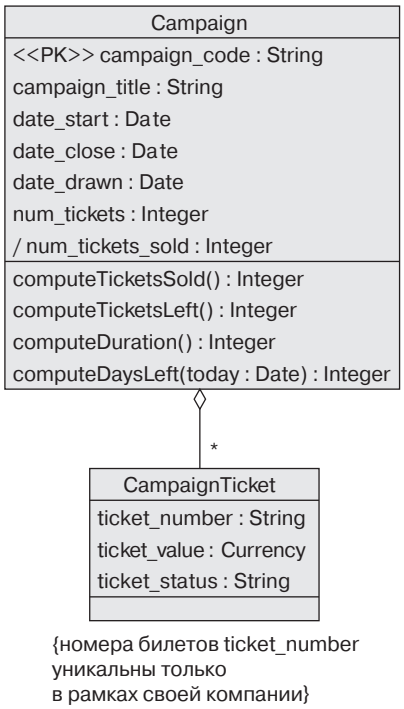

Рис. 5.2. Ограничение для класса (Телемаркетинг)

На рис. 5.2 представлено простое расширение модели классов, позволяющее показать ограничение. Ограничение связано с классом `CampaignTicket`. В отчете, выработанном любым CASE-средством, ограничение будет приведено вместе с другими свойствами класса.

Пример 5.2. Управление контактами с клиентами

Обратитесь к примеру 4.8 (разд. 4.2.2.3). Предположим, что было выявлено новое требование, которое не было предусмотрено в системе: планирование мероприятий сотрудниками для себя. Это значит, что сотрудник, который создал мероприятие (`Employee.created`), не должен быть тем же сотрудником, который отвечает за выполнение мероприятия (`Employee.due`).

Расширьте соответствующую часть модели классов (рис. 4.5) таким образом, чтобы включить новое требование.

Решение для этого примера (рис. 5.3) имеет своим результатом ограничение на ассоциативные связи, что показано в виде прерывистой стрелки-указателя зависимости, проведенной от ассоциации `created` к ассоциации `due`. Ограничение *привязано* к стрелке.

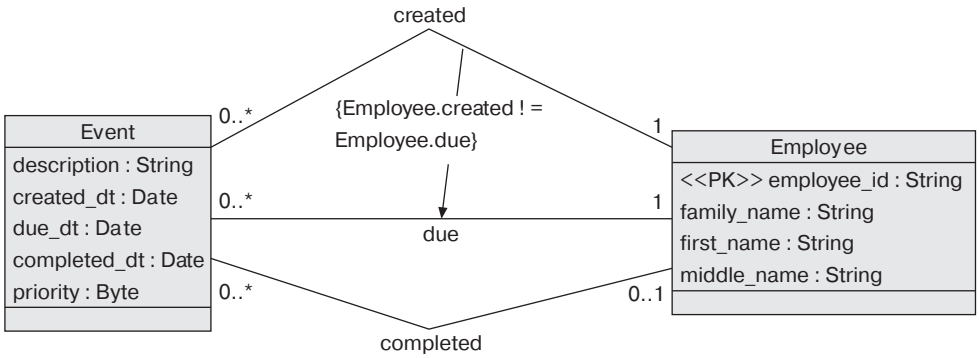

Рис. 5.3. Ограничение на ассоциации (Управление контактами с клиентами)

5.1.3. Примечания и дескрипторы

Если описание ограничения слишком длинно или же ограничение должно быть выражено с использованием трех или более графических символов, можно воспользоваться таким понятием UML, как *примечание* (*note*). Графически "примечание представляет собой прямоугольник с загнутым правым верхним краем" [76].

Символ примечания может содержать текст для выражения ограничения, однако, в общем случае, он может содержать любую информацию. Чтобы показать, что примечание является ограничением, его необходимо дополнительно обозначить как стереотип с помощью ключевого слова «`constraint`».

Подобно примечаниям, *дескрипторы* (*tag*) представляют произвольную текстовую информацию в модели, в том числе, ограничения. Аналогично ограничениям дескрипторы записываются внутри фигурных скобок и принимают следующий вид

`tag = значение`, например:

`{author = les, status = 2nd iteration}`.

Аналогично стереотипам и ограничениям некоторые дескрипторы предопределены в UML. Обычно дескрипторы используются для отображения информации по управлению проектом.

Пример 5.3. Управление контактами с клиентами

Обратитесь к примеру 4.8 (разд. 4.2.2.3) и приведенному выше примеру 5.2. Предположим, что было выявлено новое требование, которое не было предусмотрено в системе: сотрудник, который создал задание, должен также создать — в этой же транзакции — первое мероприятие для этого задания.

Расширьте соответствующую часть модели классов (рис. 4.5 и 5.3) таким образом, чтобы включить новое требование. Используйте примечание ограничения. Также отобразите на диаграмме информацию о том, что аналитик, разработавший диаграмму — les (вероятно, подразумевается автор — Лешек), а диаграмма представляет собой вторую итерацию.

На рис. 5.4 показана расширенная модель. Для представления информации по проекту используется дескриптор в левом верхнем углу диаграммы. Для представления ограничения на трех ассоциациях используется примечание. Примечание обозначено как стереотип с помощью слова `«constraint»`, чтобы подтвердить, что примечание — это ограничение.

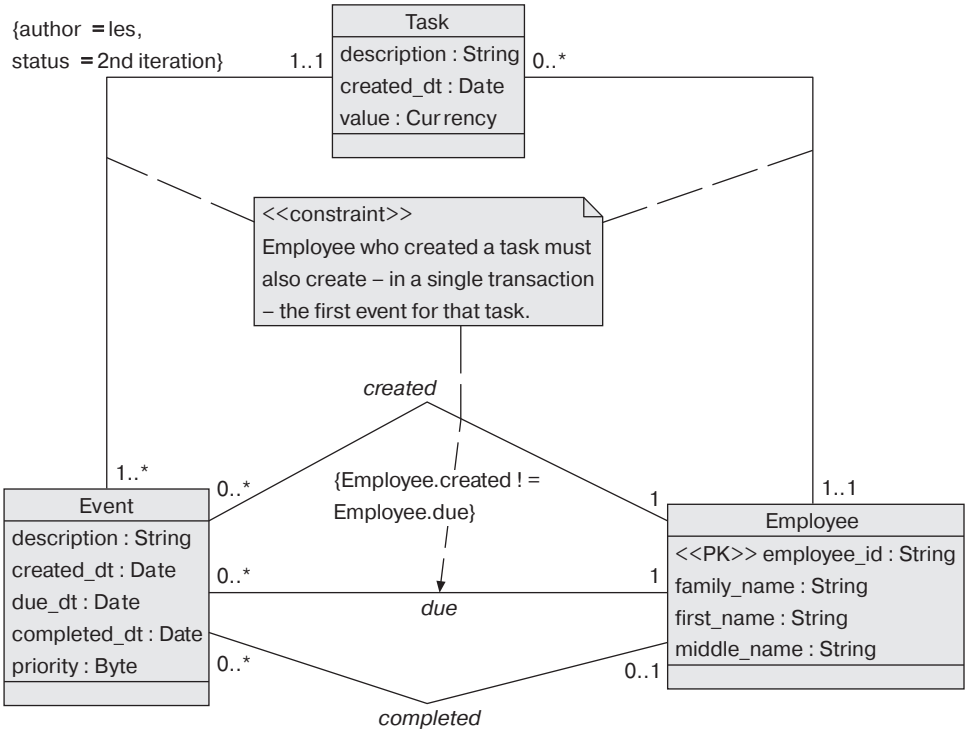

Рис. 5.4. Примечание и дескриптор как ограничения (Управление контактами с клиентами)

5.1.4. Видимость и инкапсуляция

Концепция *видимости* (*visibility*) и связанное с ней понятие *инкапсуляции* (*encapsulation*) были впервые рассмотрены в разд. 2.1.2.1.2 и 2.1.2.2.2. При этом было только обозначено различие между открытой и закрытой видимостью атрибутов и операций. Третий предопределенный в UML тип видимости — *защищенная* (*protected*) — не рассматривался.

Точно так же не рассматривались различные тонкие взаимные влияния между видимостью и наследованием. Кроме того, не были рассмотрены популярные и удобные (кое-кто может возразить — неудобные) методы преодоления инкапсуляции объектов с помощью *дружественных* операций.

В UML в качестве стандартных обозначений видимости приняты следующие знаки: + (для открытой), # (для защищенной) и – (для закрытой) видимости. Знаки помещаются перед именем свойства. CASE-средства часто заменяют эти довольно тусклые обозначения. На рис. 5.5 приведен пример более привлекательной нотации.

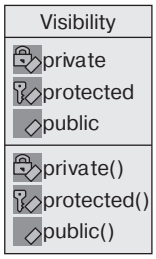

Рис. 5.5. Обозначение видимости с помощью CASE-средства

5.1.4.1. Защищенная видимость

Ситуация, при которой *закрытые свойства* (атрибуты и операции) базового класса доступны только объектам базового класса, иногда связана с неудобствами. Во многих случаях объектам производного класса (подкласса базового класса) необходимо разрешить доступ к закрытым свойствам базового класса.

Рассмотрим иерархию классов, где `Person` — это 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 такому как дата рождения — `date_of_birth`).

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* (*protected*). (Вспомним из разд. 2.1.2.1.2, что видимость применяется по отношению к классам. Если `Betty` — еще один объект класса `Employee`, то он должен иметь доступ к любому свойству `Joe`: открытому, защищенному или закрытому.)

Подобный сценарий справедлив для большинства известных объектно-ориентированных сред программирования, примерами могут служить C++ и Java [25]. Язык UML не составляет исключения в этом вопросе. Здесь просто утверждается: “Видимость является частью отношения между элементом и содержащим его контейнером. Контейнер может быть пакетом, классом или другим представителем пространства имен” [76].

Пример 5.4. Телемаркетинг

Обратитесь к постановке задачи 4 (разд. 2.3.4) и примеру 4.7 (разд. 4.2.1.2.3). Постановка задачи содержит следующее замечание: “Эти схемы включают специальные *призовые 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.

Наша задача состоит в выполнении следующих действий.

- Обновить модель класса, включив в нее класс BonusCampaign.
- Изменить видимость атрибутов класса Campaign (рис. 5.2) таким образом, чтобы они были видимы классу BonusCampaign за исключением атрибута date_start. Сделать атрибуты campaign_code и campaign_title видимыми для остальных классов модели.
- Добавить в класс Campaign следующие вычислительные операции: computeTicketsSold (проданные билеты), computeTicketsLeft (остаток билетов), computeDuration, (продолжительность [кампании]), computeDaysLeft (оставшиеся дни).
- Осознать, что у внешних классов нет нужды в атрибуте computeTicketsSold. Им требуется знать только атрибут computeTicketsLeft. Кроме того, computeDuration используется только операцией Campaign.computeDaysLeft.
- Класс BonusCampaign хранит атрибут ticket_book_size (величина билетной книжки), а операция доступа к нему называется `bookSize`.

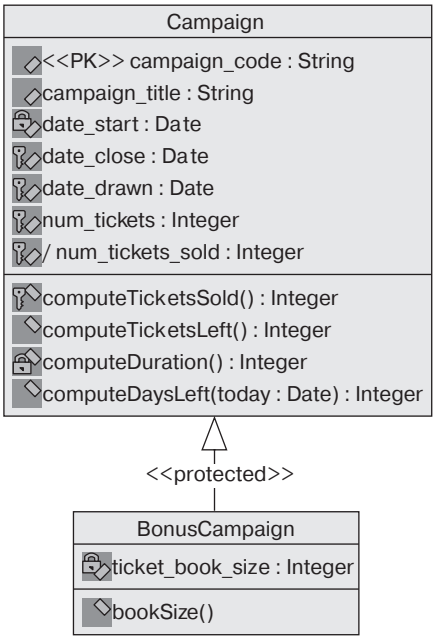

Рис. 5.6. Видимость свойств класса (Телемаркетинг)

На рис. 5.6 показана модель классов, соответствующая приведенному выше пояснению. Стереотип «protected» на линии обобщения рассматривается в следующем разделе. Операция computeTicketsSold в классе Campaign является закрытой. Операция computeDuration в классе Campaign является защищенной. Остальные две операции в классе Campaign — открытые. Операция bookSize специфична для класса BonusCampaign и является открытой.

5.1.4.2. Видимость унаследованных свойств классов

Как декларируется в UML, характеристика *видимости* применяется к объектам на различных уровнях детализации. Обычно имеется в виду, что видимость применяется к *элементарным объектам* — атрибутам и операциям. Однако видимость можно задать и по отношению к другим "контейнерам". Это приводит к полной неразберихе с правилами замещения.

Рассмотрим, к примеру, распространенную ситуацию, когда видимость определяется в иерархии наследования на уровне базового класса *и* на уровне свойств базового класса. Пусть, скажем, класс B — подкласс класса A. Класс A содержит смесь атрибутов и операций — некоторые из них являются открытыми, другие — закрытыми, а остальные — 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).

Типичное разрешение приведенного выше сценария выглядит следующим образом [38].

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

Пример 5.5 Телемаркетинг

Обратитесь к приведенному выше примеру 5.4. Предположим, что стереотип «protected» на линии обобщения означает, что класс Campaign определен в BonusCampaign как защищенный:

```
class BonusCampaign: protected Campaign
```

Определите уровень видимости свойств унаследованного класса в классе BonusCampaign. Поясните ваше решение.

Если базовый класс определен как защищенный (`protected`), его открытые (`public`) свойства в производном классе изменятся на защищенные. Защищенные свойства останутся защищенными. Закрытые свойства не будут доступны производному классу (они никогда не доступны).

На рис. 5.7 и 5.8 представлены диалоговые окна CASE-репозитория, которые, соответственно, отображают атрибуты и операции класса `BonusCampaign` после вступления в силу эффекта наследования.

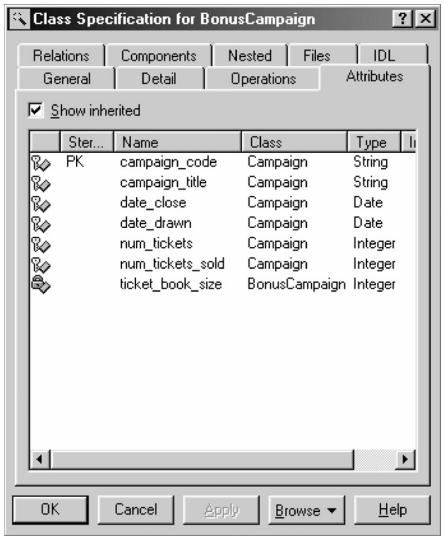

Рис. 5.7. Унаследованные атрибуты в классе BonusCampaign (Телемаркетинг)

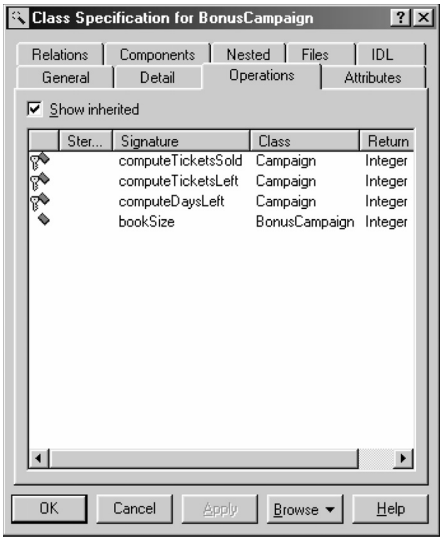

Рис. 5.8. Унаследованные операции в классе BonusCampaign (Телемаркетинг)

5.1.4.3. Дружественные классы и операции

В некоторых случаях функции (или даже классу в целом) требуется предоставить непосредственный доступ к свойствам другого класса независимо от уровней видимости этих свойств. Подобная ситуация возникает, когда два класса тесно связаны и операции одного из классов зависят от свойств другого класса. В качестве типичного примера можно привести два класса: `Book` (Книга) и `BookShelf` (Книжная полка), и операцию класса `Book` под названием `putOnBookShelf` (Поставить на полку).

Один из способов разрешения ситуации, подобной описанной выше, состоит в том, чтобы объявить операцию `putOnBookShelf` *дружественной* (*friend*) в пределах класса `BookShelf`; это может выглядеть, примерно, так `friend void Book::putOnBookShelf()`.

Дружественным может быть другой класс или операция класса. Дружественность не взаимна. Класс, дружественный по отношению к другому классу, может и не пользоваться дружественностью со стороны последнего.

Дружественная операция или класс объявляются *внутри* класса, который наделяет дружбой другой класс. Однако дружественная операция — это не свойство класса, поэтому к ней неприменимы атрибуты видимости. Это также означает, что в определении дружественного отношения нельзя упоминать атрибуты класса даже по имени — все они должны квалифицироваться по имени класса (как будто дружественная операция — это обычная внешняя операция).

В языке UML дружественные отношения показываются с помощью прерывистой линии *отношения зависимости* (*dependency relationship*), исходящей из класса или операции к классу, который "дарит" дружбу. Стереотип «`friend`» привязан к стрелке зависимости. По общему признанию, нотация UML не полностью воспринимает и поддерживает семантику дружественности.

Пример 5.6. Телемаркетинг

Обратитесь к примеру 4.7 (разд. 4.2.1.2.3). Рассмотрите отношение между классами `Campaign` и `CallScheduled`.

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

Наша задача состоит в том, чтобы объявить операцию `getTicketsLeft` дружественной к классу `Campaign`.

Решение в терминах UML для нашего примера показано на рис. 5.9. Операция `getTicketsLeft` расположена внутри класса `CallScheduled`. Напомним, что в языке программирования, используемом для реализации, операция `getTicketsLeft` объявляется дружественной внутри класса `Campaign` (поскольку именно класс `Campaign` определяет, кто будет другом), а затем полностью определяется в классе `CallScheduled`

Отношение зависимости фиксирует тот факт, что класс `Campaign` предоставляет некоторый дружественный статус классу `CallScheduled`. Из нотации не совсем ясно, распространяется ли дружественный статус на весь класс или на некоторые операции класса (и на какие именно).

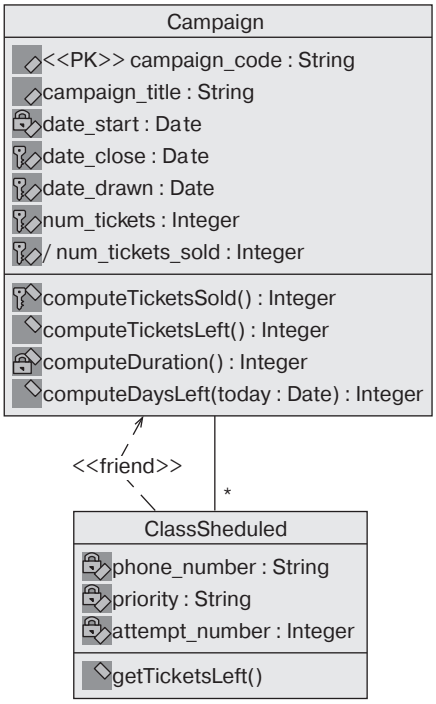

Рис. 5.9. Дружественность (Телемаркетинг)

5.1.5. Производная информация

Производная информация (*derived information*) представляет собой разновидность ограничения, которое наиболее часто применяется к атрибуту или ассоциации. Производная информация вычисляется на основе других элементов модели. Строго говоря, производная информация является избыточной — при необходимости ее можно вычислять.

Хотя производная информация не обогащает семантику *модели анализа*, она придает модели большую четкость (поскольку в модели явно находит отражение факт возможности вычисления некоторой величины). Решение о том, приводить или нет в модели анализа производную информацию, довольно произвольно и должно последовательно распространяться на модель в целом.

Владение производной информацией имеет большее значение для *проектных моделей*, в которых требуется учет возможностей оптимизации доступа к информации. Применительно к проектным моделям можно также принять решение о том, хранить ли производную информацию (после ее вычисления) или же динамически вычислять ее каждый раз, когда в ней возникает необходимость. Это не новое свойство — в прежних сетевых базах данных оно было известно как *актуальные* (т.е. хранимые) или *виртуальные* данные.

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

5.1.5.1. Производный атрибут

Мы уже использовали производные атрибуты ранее в нескольких примерах диаграмм, правда, не приводя никаких объяснений. Например, атрибут `num_tickets_sold` на рис. 5.2 — производный. Расширенная диаграмма для этого рисунка показана ниже (рис. 5.10).

Значение атрибута `num_tickets_sold` вычисляется с помощью операции `computeTicketsSold`. Выполнение операции следует агрегативной связи с классом `CampaignTicket` (Билеты, используемые в кампании) и проверяет статус каждого билета — `ticket_status`. Если статус билета — `ticket_status` — принимает значение "продан", то к счетчику проданных билетов прибавляется единица. После обработки всех билетов выводится текущее значение количества проданных билетов `num_tickets_sold`.

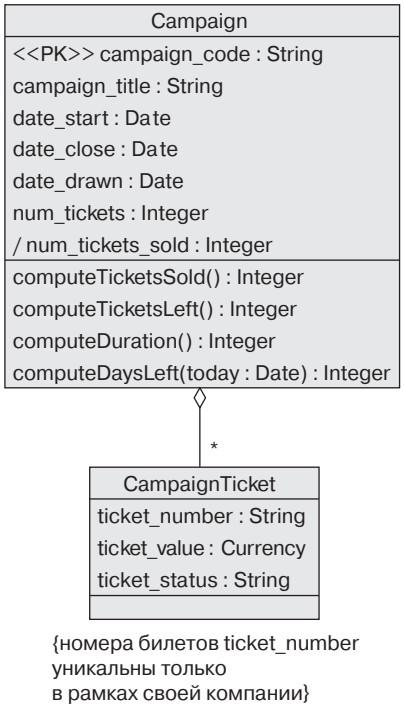

Рис. 5.10. Производный атрибут (Телемаркетинг)

5.1.5.2. Производная ассоциация

Производная ассоциация — более спорная тема. Типичным случаем производной ассоциации можно считать ассоциацию, образованную тремя классами, уже соединенными двумя ассоциациями, при этом третья ассоциативная связь замыкает контур. Зачастую необходимость в третьей ассоциации возникает в связи с требованием семантической корректности модели (это известно как *коммутативность контура*). Если

третья ассоциация не представлена в модели явно, она может быть выведена из двух других ассоциативных связей.

Пример 5.7. Internet-магазин

Обратитесь к наставлению по Internet-магазину в главе 2 и к диаграмме классов на рис. 2.32 (разд. 2.2.4.6). Рассмотрите часть модели, включающую классы `Customer`, `Order` и `Invoice`.

Существует ли возможность ввести производную ассоциацию в модель? Если да, то добавьте ее к диаграмме.

Да, между классами `Customer` и `Invoice` возможно ввести производную ассоциацию. На приведенном ниже рис. 5.11 она обозначена именем `/CustInv`. Эта ассоциация выводится, благодаря слегка необычному бизнес-правилу, по которому кратность ассоциации между классами `Order` и `Invoice` равна "один к одному".

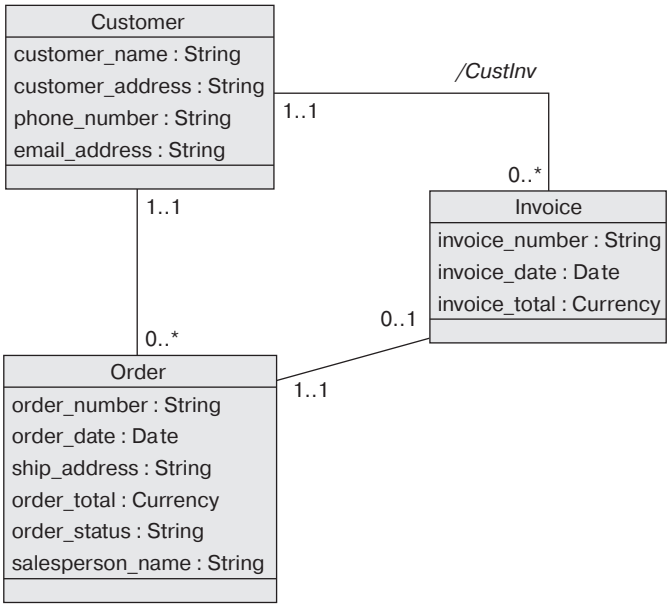

Рис. 5.11. Производная ассоциация (Internet-магазин)

Производная ассоциация не вносит в модель какой-либо новой информации. Всегда можно связать клиента (`Customer`) с определенным заказом (`Order`), отыскав один заказ для каждой счет-фактуры (`Invoice`), а затем одного клиента для каждого заказа.

5.1.6. Квалифицированная ассоциация

К концепции *квалифицированной ассоциации* (*qualified association*) в среде специалистов по моделированию относятся по-разному. Некоторые охотно пользуются этим понятием, другие не воспринимают его вовсе. Можно ли построить полную и достаточно выразительную модель классов без использования квалифицированной ассо-

циации — вопрос спорный. Однако если уж квалифицированная ассоциация используется, то использовать ее при построении моделей классов следует последовательно.

По одну сторону линии бинарной квалифицированной ассоциативной связи имеется "отделение" для атрибута (*квалификатора* (*qualifier*)) (ассоциация может быть квалифицирована по обоим концам связи, но это довольно редкий случай). Это "отделение" содержит один или более атрибутов, которые служат в качестве ключа индекса для прослеживания ассоциативной связи от *квалифицирующего класса* (*qualified class*) посредством квалификатора до *целевого класса* (*target class*) на противоположном конце ассоциации.

Например, ассоциация между классами `Flight` (Рейс) и `Passenger` (Пассажир) имеет кратность "многие ко многим". Однако, если класс `Flight` квалифицировать с помощью атрибутов `seat_number` (место) и `departure` (отправление), то кратность ассоциации снижается до значения "один к одному" (рис. 5.12). Составной ключ индекса, введенный с помощью квалификатора (`flight_number + seat_number + departure`), может быть связан только с одним объектом `Passenger` либо не связан вообще ни с одним объектом этого класса.

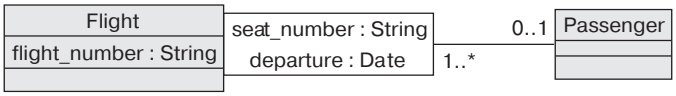

Рис. 5.12. Квалифицированная ассоциация

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 [76].

Уникальность в идентификации объектов, вводимая за счет использования квалификаторов, зачастую представляет собой важную семантическую информацию, которую невозможно эффективно получить другими способами (такими как ограничение или добавление дополнительных атрибутов в целевой класс). В общем случае дублировать атрибут квалификации в целевом классе нежелательно.

Пример 5.8. Internet-магазин

Обратитесь к наставлению по Internet-магазину в главе 2 и к диаграмме классов на рис. 2.32 (разд. 2.2.4.6). Рассмотрите часть модели, включающую классы `Order` и `Computer`.

Измените ассоциацию между классами `Order` и `Computer` на составную ассоциацию для явного представления ограничения: "в каждом заказе на компьютер отводится одна позиция заказа".

Как мы помним, компьютер заказывается после того, как клиент составит спецификацию на него. Каждая такая конфигурация может быть обозначена уникальным номером. Стандартной конфигурации также можно присвоить уникальный номер. Поэтому атрибут номера конфигурации `configuration_number` представляет собой квалификатор для объекта-заказа `Order`. Составной ключ (`order_number + configuration_number`) связывает заказ с одним объектом `Computer` (рис. 5.13).

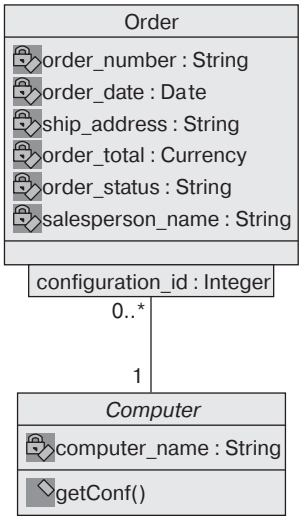

Рис. 5.13. Квалифицированная ассоциация (Internet-магазин)

5.1.7. Ассоциативный класс или материализованный класс

В разделе 2.1.3.4 мы рассмотрели и привели пример *ассоциативного класса* — 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. Чтобы обеспечить возможность хранить эти атрибуты, требуется класс — 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, специалист по моделированию должен материализовать ассоциацию, заменив класс `C` обычным классом `D` [76]. *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`.

5.1.7.1. Постановка задачи для примера с базой данных сотрудников

Для объяснения различий между ассоциативным классом и материализованным классом нам потребуется специальный пример. Пример касается ведения учета текущей и прошлой зарплаты сотрудников. Конечно, различия между ассоциативным классом и материализованным классом чаще всего проявляются в контексте моделирования временно*й* (хронологической) информации.

Постановка задачи для примера. База данных сотрудников

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

Каждому сотруднику в штатном расписании определен уровень его заработной платы. Для каждого уровня существуют вилки окладов, т.е. минимальная и максимальная зарплата. Вилки окладов для данного уровня никогда не изменяются. Если возникает необходимость изменить минимальную или максимальную зарплату, 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. Любые изменения зарплаты сотрудника в рамках одного и того же уровня также фиксируются.

5.1.7.2. Модель, использующая ассоциативный класс

При порождении объектов ассоциативного класса им – подобно объектам любого обычного класса – присваиваются уникальные идентификаторы (OID) (разд. 2.1.1.3). Помимо системных OID объекты можно также идентифицировать по значениям их атрибутов. В случае ассоциативного класса источником идентичности объекта являются атрибуты, которые обозначают ассоциированные классы (разд. 2.1.2.1.1). Значения других атрибутов не способствуют идентификации объекта.

Пример 5.9. База данных сотрудников

Обратитесь к приведенной выше постановке задачи для примера *База данных сотрудников*. Постройте для нее модель классов. Используйте при этом ассоциативный класс.

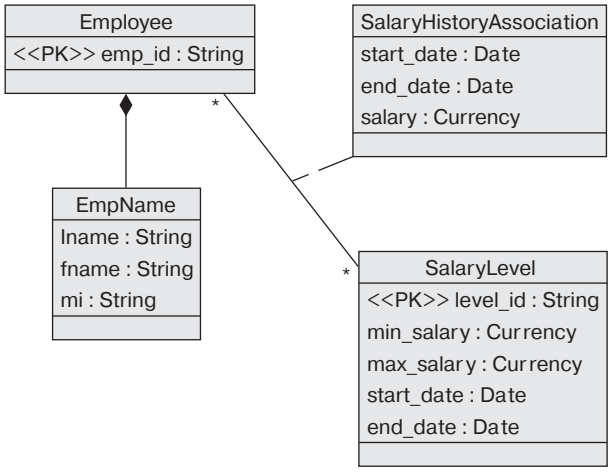

Рис. 5.14. Неадекватное использование ассоциативного класса (База данных сотрудников)

Постановка задачи для примера создает некоторые проблемы. Мы знаем, что нам требуется класс для хранения подробной информации о сотруднике (`Employee`), а также класс для хранения информации об уровнях зарплаты (`SalaryLevel`). Про-

блема состоит в моделировании текущих назначений зарплаты работникам, а также хронологии этих назначений. На первый взгляд кажется естественным использовать ассоциативный класс `SalaryHistoryAssociation`.

На рис. 5.14 представлена модель классов, в которой использован ассоциативный класс `SalaryHistoryAssociation`. Подобное решение неадекватно. Идентичность объектов `SalaryHistoryAssociation` выводится на основании составного ключа, созданного с использованием ссылок на первичные ключи классов `Employee` и `SalaryLevel` (т.е. `emp_id` и `level_id`).

Никакие два объекта `SalaryHistoryAssociation` не могут иметь одинаковых составных ключей (т.е. одинаковых связей с объектами `Employee` и `SalaryLevel`). Это также означает, что проект диаграммы на рис. 5.14 не обеспечивает выполнение требования "Любые изменения зарплаты сотрудника в рамках одного и того же уровня также фиксируются". Решение, показанное на рис. 5.14, не может рассматриваться как удовлетворительное— требуется более корректная модель.

5.1.7.3. Модель, использующая материализованный класс

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

Пример 5.10. База данных сотрудников

Обратитесь к постановке задачи для примера *База данных сотрудников* (разд. 5.1.7.1). Постройте для нее модель классов. Используйте при этом материализованный класс.

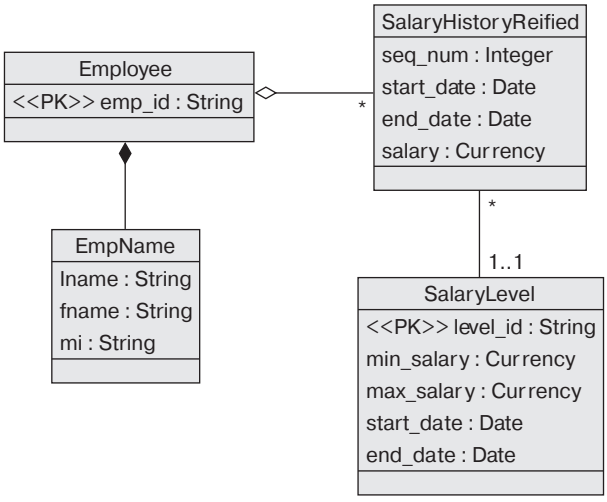

Рис. 5.15. Лучшее решение на основе материализованного класса (База данных сотрудников)

5.2. Иерархия классов

Разработчики ПО знают, что трудность разработки небольшой системы не идет ни в какое сравнение с трудностью поставки крупномасштабного решения. Небольшие объектные системы зачастую легче понять, реализовать и развернуть. Для масштабных объектных решений характерны сложные сетевые структуры объектов, реагирующих на случайные события, которые вызывают переплетающиеся цепочки взаимосвязанных операций (методов). В отсутствие четкого архитектурного проекта и строго определенного процесса разработки крупные программные проекты вполне могут завершиться провалом.

Хорошо известный принцип когнитивной психологии — *правило 7±2* — гласит, что кратковременная память человеческого мозга может одновременно справиться приблизительно с девятью (7±2) предметами (графическими элементами, идеями, понятиями и т.д.). Нижняя граница, равная пяти (7-2), указывает на то, что работа, менее чем с пятью предметами, составляет тривиальную проблему.

Правила когнитивной психологии не в состоянии повлиять на тот факт, что нам необходимы большие системы, а большие системы сложны. Значительная часть этой сложности привнесена человеком, а не присуща самой системе (разд. 1.1.1).

Главным виновником тут выступает системное моделирование, которое допускает неограниченное взаимодействие объектов. В подобных системах объекты образуют *сетевые структуры* — "паутину" объектов, образованную перекрестными ссылками. Эти ссылки образуют пути передачи сообщений. Возможны обращения к нижележащим и вышележащим объектам. В сетевых структурах количество возможных путей взаимодействия между объектами с появлением в системе новых объектов растет экспоненциально. Как замечает Шиперски (Szyperski): "...объектные ссылки порождают связи над произвольными областями абстракции. Поэтому верный выбор уровней применительно к системной архитектуре становится проблематичным, если вообще возможным" [88].

Удачные системы организованы по иерархическому принципу — при этом сетевые структуры оказываются ограниченными и тщательно контролируются. Применение *иерархических структур* позволяет снизить степень сложности с экспоненциальной до полиномиальной. Они вносят в систему уровни (или слои) объектов и ограничивают взаимодействие между уровнями. В типичной иерархии непосредственно взаимодействуют только объекты соседних уровней. Таким образом удается скрыть сложность и разнести ее по разным уровням.

5.2.1. Сложность сетевых структур

Прежде чем приступить к рассмотрению *сложности* объектных систем, необходимо договориться о мере этой сложности. Как измерить сложность? Сложность разнообразна и принимает разные формы. Простой, но очень показательной мерой сложности может выступать количество соединений между классами. *Соединение* (*connection*) определяется как существование постоянной или временной связи между классами (разд. 2.1.1.3).

Каждое соединение обычно допускает двунаправленное взаимодействие классов, т.е. от `A` к `B` и от `B` к `A`. Рис. 5.16 иллюстрирует сложность сети, состоящей из семи классов. Между *n* классами существует `n(n-1)/2` возможных соединений. Применение этой формулы к нашему случаю дает в итоге 21 соединение (42 пути взаимодействия).

Обратите внимание, что сложность измерялась по отношению к классам, а не объектам. В программах именно объекты, а не классы, пересылают сообщения другим объектам того же или другого класса. Это создает для программиста, ответственного за разработку логики программы и управление программными переменными и другими структурами, дополнительные трудности. Однако основная проблема заключается не в сложности отдельной программы, а в сложности всей системы программ.

Объект может отправить сообщение другому объекту лишь при наличии между ними постоянной или временной связи. Временные связи разрешаются в пределах однократного вызова программы, и поэтому они не усложняют систему в целом. С другой стороны, постоянные связи существуют только при наличии соединения (например, ассоциации), определенного в модели классов. Классы могут совместно и многократно использоваться во многих программах.

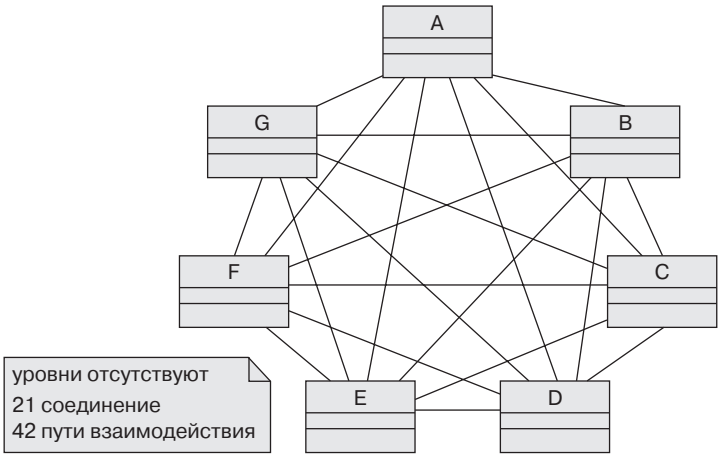

Рис. 5.16. Сложность сетевых структур

5.2.2. Сложность иерархических структур

Решение проблемы управления сложностью состоит в редукции сетевой структуры за счет группирования классов в виде *иерархии классов*. При этом подходе классы могут естественным образом упорядочиваться в виде уровней, которые подчеркивают иерархическое разбиение между уровнями, делая в то же время возможным взаимодействие внутри уровней наподобие сетевого.

Разделение на уровни иерархии дает возможность снизить сложность за счет ограничения количества возможных путей взаимодействия между объектами. Это снижение сложности достигается посредством разделения классов на уровни и разрешения непосредственного взаимодействия классов внутри уровня и между соседними уровнями.

Рис. 5.17 иллюстрирует сложность иерархии, составленной из семи классов, при этом классы сгруппированы в четыре уровня. По сравнению с сетевой структурой, показанной на рис. 5.16, сложность понизилась с 42 до 26 путей взаимодействия.

Пример на рис. 5.17 придуман в соответствии с четырьмя уровнями, предлагаемыми в нашем подходе *BCDE*, который рассматривается в разд.6.1.3.2. В иерархической структуре с большим количеством классов количество *межуровневых соединений* может вызывать затруднения. Для уменьшения сложности межуровневых соединений межу-

ровневые взаимодействия могут быть направлены через несколько специально предназначенных для этого *классов-посредников*. Это в еще большей степени снижает общую сложность системы.

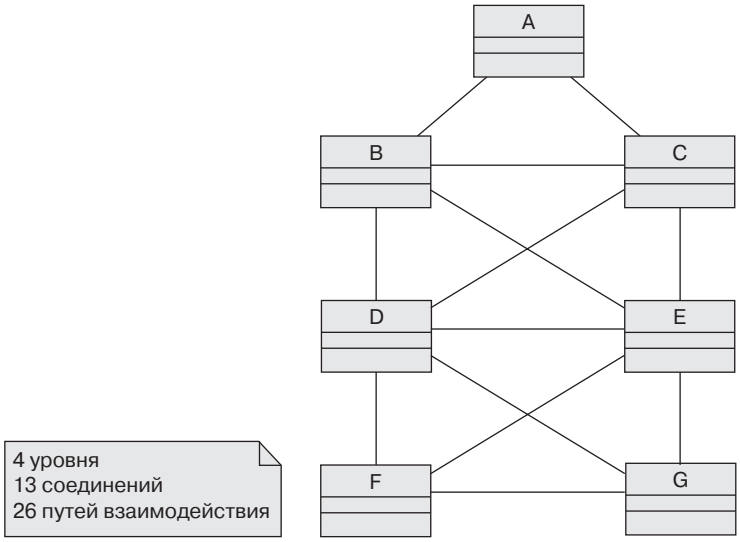

Рис. 5.17. Сложность иерархических структур

5.2.3. Пакеты

Для представления групп классов (или других элементов моделирования, например, прецедентов) в языке UML предусмотрено понятие *пакета* (*package*) [76]. Пакеты служат для разделения логической модели прикладной программы. Они представляют собой кластеры (блоки) сильносвязанных классов, которые сами образуют некое единое целое, но слабо связаны с другими подобными кластерами [48].

Пакеты могут быть *вложенными*. Это позволяет осуществлять декомпозицию больших систем на подсистемы, модули и т.д. Например, пакет, обозначенный как стереотип «`system`», может содержать пакеты, обозначенные в виде стереотипа «`subsystem`». Внешний пакет обладает доступом к любому классу, который непосредственно содержится в его вложенных пакетах.

Классом может владеть только один пакет. Это не препятствует появлению класса в других пакетах или взаимодействию с классами в других пакетах. Используя объявление класса внутри пакета как закрытого, защищенного или открытого, можно контролировать взаимодействие и зависимости между классами в разных пакетах (разд.5.1.4).

5.2.3.1. Обозначение пакетов

Пакет изображается в виде пиктограммы папки (рис. 5.18). Вложенные пакеты изображаются внутри внешнего пакета. Для каждого пакета должна быть построена собственная диаграмма классов, определяющая все классы, принадлежащие пакету.

Пакеты могут быть связаны с двумя типами отношений: *обобщением* и *зависимостью*. В качестве отношения на рис. 5.18 показана зависимость. Из рисунка видно, что пакет `NestedPackage` зависит от пакета `Package2`. Точный характер зависимости не задается, за исключением, скажем, ситуации, когда изменения для `Package2` могут потре-

бовать внесения изменений в пакет NestedPackage. Пожалуй, наиболее часто зависимость между пакетами обусловлена тем, что класс одного из пакетов отправляет сообщение классу другого пакета.

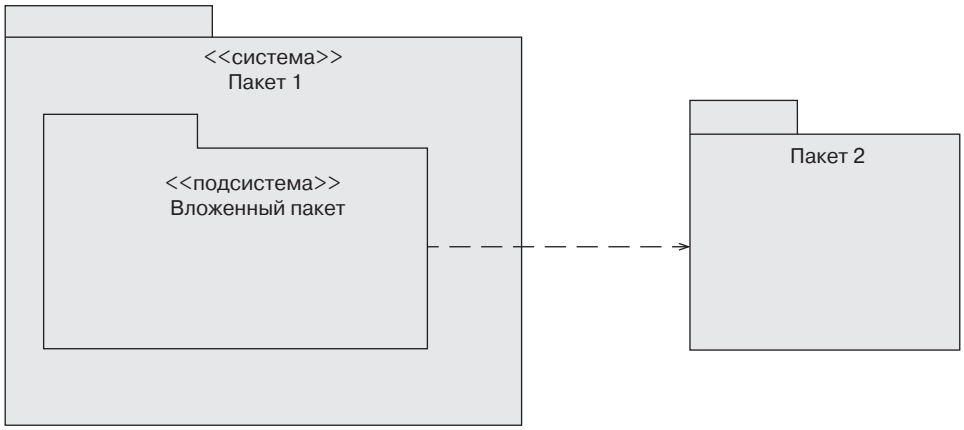

Рис. 5.18. Пакеты и зависимости

В языке UML определено несколько разных категорий отношения зависимости (например, зависимость по использованию, зависимость по доступу, зависимость по видимости). Мы считаем, что определение категории зависимости не приносит большой пользы для диаграмм анализа. Истинный характер каждой зависимости должен быть зафиксирован как описательное ограничение в CASE-репозитории для использования в проектных моделях.

Обратите внимание, что отношение обобщения между пакетами предполагает также зависимость. Эта зависимость направлена от пакета подтипа к пакету супертипа. Изменения в пакете супертипа вызывают изменения в пакете подтипа.

5.2.3.2. Диаграммы пакетов

Понятия диаграммы пакетов как таковой в UML не существует, тем не менее этот термин удобен для использования. Пакеты создаются в диаграмме классов или диаграмме прецедентов. В первом случае диаграмма пакетов представляет модель системы с точки зрения ее состояний. Во втором случае — с точки зрения поведения системы. После того, как пакет создан, назначение классов (или прецедентов) пакету должно быть предусмотрено в отдельной диаграмме классов (или диаграмме прецедентов).

Без сомнения, диаграмма пакетов для состояний описывает архитектурную основу системы. Поведенческая диаграмма пакетов служит только в качестве описания высокоуровневой функциональной структуры [15]. Диаграмма пакетов для состояний помогает в управлении масштабами и сложностью системы. Она также играет важную роль в качестве механизма доступа и конфигурационного управления, способствующего сотрудничеству разработчиков.

На рис. 5.19 показаны три пакета и отношения зависимости между ними. Пакет Enrolment зависит от пакетов Marking и Timetable.

Пример 5.11. Запись на университетские курсы

Более пристальный взгляд на приложение *Запись на университетские курсы* показывает, что система должна быть "осведомлена" о расписании занятий и степенях студентов, чтобы иметь возможность проверить обоснованность записи студентов в группы.

Нам неизвестно, существуют ли "расписание" ("marking") и "выставление оценок" ("timetable") в качестве отдельных программных модулей, в которые может быть помещена наша система записи на курсы. Если это не так, то система записи на курсы должна включать подобные модули.

Наша задача состоит в том, чтобы построить модель пакетов для приложения *Запись на университетские курсы*.

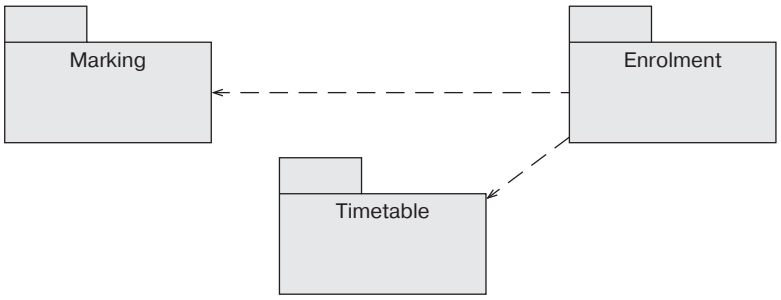

Рис. 5.19. Пакеты (Запись на университетские курсы)

5.2.4. Подход BCE

Подход *BCE* (**B**oundary–**C**ontrol–**E**ntity — граница–управление–сущность) представляет собой подход к объектному моделированию, основанный на трехфакторном представлении классов. Подходу *BCE* до некоторой степени можно сопоставить хорошо известный подход *MVC* (**M**odel–**V**iew–**C**ontroller — модель–представление–контроллер) [14], [11], [59]. В языке UML на классах предопределены три стереотипа: boundary (граница), control (управление) и entity (сущность) [76]. Это и определило выбор аббревиатуры BCE.

Пограничные классы (*boundary class*) описывают объекты, которые представляют интерфейс между субъектом и системой. Они выделяют часть состояния системы и представляют ее пользователю в форме визуального отображения или звукового эффекта. Пограничные объекты часто сохраняются после однократного выполнения программы.

Управляющие классы (*control class*) описывают объекты, которые перехватывают входные события, инициированные пользователем, и контролируют выполнение бизнес-процесса. Управляющий класс представляет действия и виды деятельности прецедентов. Управляющие объекты зачастую не сохраняются после выполнения программы.

Классы-сущности (*entity class*) описывают объекты, которые представляют семантику сущностей, принадлежащих проблемной области. Они соотносятся со структурами данных системной базы данных. Объекты-сущности всегда сохраняются после выполнения программы и участвуют во многих прецедентах.

Как показано на рис. 5.20, в верно спроектированной иерархии пакетов субъект может взаимодействовать только с пограничными объектами из пакета `BoundaryPackage`, объекты-сущности из пакета `EntityPackage` могут взаимодействовать только с управляющими объектами из `ControlPackage` и управляющие объекты из `ControlPackage` могут взаимодействовать со объектами любого типа [15].

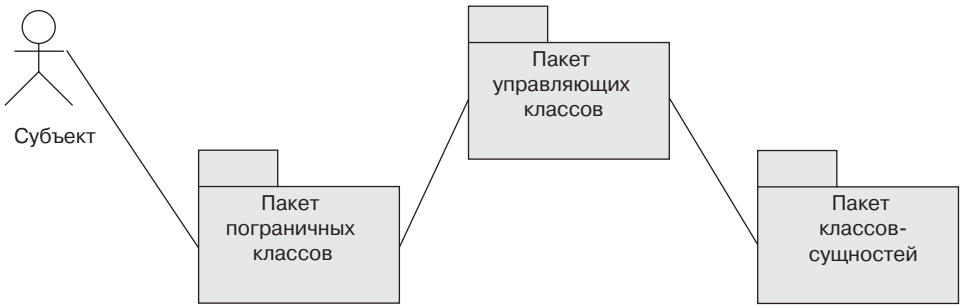

***Рис. 5.20. Соединение объектов в иерархии пакетов* BCE**

Пограничные классы соответствуют классам, представленным в разработанном GUI-интерфейсе. Пограничные объекты зависят от объектов-сущностей (через некоторые управляющие объекты). Объекты-сущности распространяют изменения на свои состояния, так что пограничные объекты могут обновить отображение GUI-интерфейса. Говорят, что "пограничный слой" (экран) *поврежден*, если он временно не синхронизирован с состоянием соответствующих сущностей проблемой области.

Управляющие объекты манипулируют взаимодействиями между событиями, инициированными пользователями, и воздействуют на объекты-сущности и пограничные объекты. В частности, каждому пограничному объекту, который допускает взаимодействие, должен быть поставлен в соответствие связанный с ним управляющий объект. Поэтому некоторые типы архитектуры ПО объединяют две функции в одном классе. В качестве примера можно привести библиотеку MFC (Microsoft Foundation Classes— библиотека базовых классов Microsoft).

Подход *BCE*— это "способ мышления", который воплощает в себе лучшие достижения практики разработки ПО. Поэтому он должен быть составной частью любого объектно-ориентированного метода разработки приложений вне зависимости от того, подкрепляется ли он базовой платформой реализации.

Основным преимуществом подхода *BCE* является группирование классов в виде иерархических уровней. Это способствует лучшему пониманию модели и уменьшает ее сложность. Это также позволяет снизить риск создания классов, которые берут на себя слишком много функций, *классов-"монстров"*, контролирующих работу всей системы. Хороший объектно-ориентированный проект отличает равномерное распределение "системного интеллекта" по всем классам.

Дополнительным преимуществом подхода BCE является его увязка с *трехзвенной моделью клиент/сервер*, которая отделяет управление данными (объекты-сущности) от представления (пограничные объекты) посредством промежуточного слоя логики приложения (управляющие объекты). Сервер баз данных может даже быть реализован не на объектно-ориентированной, а на реляционной СУБД.

Следует, однако, подчеркнуть, что пакет `EntityPackage` описывает классы-сущности, которые располагаются в памяти клиентской программы, а не на сервере ба-

зы данных. Затем постоянные бизнес-объекты запоминаются в виде записей реляционных таблиц. Чтобы осуществить отображение классов-сущностей в реляционную память СУБД и для хранения другой релевантной информации о структурах баз данных, требуется отдельный уровень (слой) классов, которые организованы в виде пакета, предназначенного для работы с базой данных — `DatabasePackage` (разд.6.1.3.2).

5.3. Углубленное моделирование обобщения и наследования

Существует три основных вида отношений между классами: ассоциация, агрегация и обобщение. Внимательный читатель мог заметить, что полезность обобщения для моделей анализа была подвергнута сомнению. *Обобщение* — полезная и мощная концепция, однако, 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 и защищенные) свойства суперкласса. *Наследование* — это "механизм, с помощью которого более специфические элементы вбирают в себя структуру и поведение, определенные более общими элементами" [76].

5.3.1. Обобщение и подставимость

С точки зрения семантики моделирования *обобщение* вводит в модель дополнительные классы, разделяет их на общие и более специфические классы и устанавливает отношения суперкласс-подкласс. Хотя обобщение и вводит новые классы, оно может уменьшить общее количество отношений *ассоциации* и *агрегации* в модели.

Исходя из требуемой семантики, ассоциативная или агрегативная связь класса может указывать на наиболее общий класс иерархии обобщения (см. диаграммы классов на рис. 2.31 и 4.7). Поскольку подкласс может быть *подставлен* вместо его родительского класса, объекты подкласса связаны 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. Источником преимуществ обобщения служит принцип *подставимости* (*substantiability*) — объект подкласса может быть использован вместо объекта суперкласса в любом месте программы, где объект подкласса имеет доступ к объекту суперкласса. Однако, к сожалению, существуют такие способы использования механизма наследования, которые могут свести на нет все преимущества принципа подставимости.

5.3.2. Наследование или инкапсуляция

Инкапсуляция требует, чтобы доступ к атрибутам объекта 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 необходимо достижение определенного компромисса между инкапсуляцией и последними двумя свойствами. На практике, объявить все данные как закрытые невозможно.

Наследование подрывает основы инкапсуляции, разрешая подклассам осуществлять непосредственный доступ к защищенным атрибутам. Для вычислений, охватывающих объекты, принадлежащие различным классам, может потребоваться, чтобы различные классы были дружественными друг другу, что приводит к еще большему нарушению инкапсуляции. Иногда должны использоваться статические атрибуты, являющиеся глобальными по отношению ко всем объектам класса. И, поскольку всем ясно, что инкапсуляция касается понятия класса, а не объекта — 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 или SQL-подобных *языков незапланированных запросов*, вполне справедливо ожидают, что им придется обращаться в запросах непосредственно к атрибутам, а не испытывать необходимость работать с методами доступа к данным, которые делают формирование запросов более трудным и более подверженным ошибкам. Это требование особенно справедливо в отношении приложений хранилищ данных, связанных с интерактивной аналитической обработкой запросов (OLAP — OnLine Analytical Processing).

Приложения должны разрабатываться таким образом, чтобы достичь требуемого уровня инкапсуляции и при том соблюсти компромисс в отношении наследования, незапланированных запросов и вычислительных требований.

5.3.3. Наследование интерфейса

Когда обобщение используется с целью обеспечения подставимости, оно может служить синонимом понятия *наследование интерфейса* (*порождения подтипа*, *наследования типа*). Это, так сказать, "безопасный" вид наследования. 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.

Обычно *интерфейсы* объявляются посредством *абстрактного класса*. Можно сказать, с одной оговоркой, что интерфейс является абстрактным классом. Оговорка состоит в том, что абстрактный класс может обеспечить частичную реализацию для некоторых операций, в то время как чистый интерфейс откладывает определение всех операций.

На рис. 5.21 представлен фрагмент модели классов для приложения *Телемаркетинг*, который демонстрирует наследование интерфейса. `Computer` — это абстрактный класс. Объекты этого класса не могут быть объектами-экземплярами. Операция `getConf()` получает две различные реализации: одну в классе `ConfiguredComputer`, а другую в `StandardComputer`.

На рис. 5.22 показан альтернативный способ представления наследования интерфейса с использованием символа "леденца на палочке" для обозначения поддерживаемого интерфейса. Это отношение "леденец на палочке", исходящее из подкласса (конкретизированного класса) к абстрактному классу, формально называется *отношением реализации* (*realization relationship*).

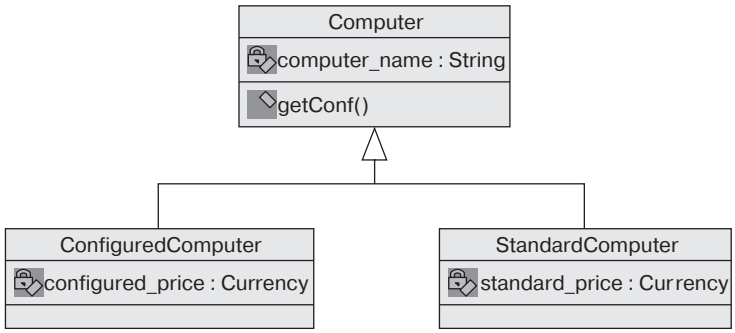

Рис. 5.21. Наследование интерфейса

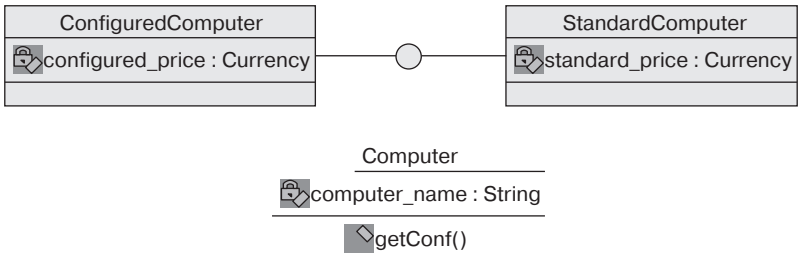

Рис. 5.22. Использование отношения реализации для представления наследования интерфейса

5.3.4. Наследование реализации

Как было отмечено в предыдущем разделе, использование обобщения может подразумевать *подставимость* с последующей ее реализацией за счет *наследования интерфейса*. Однако использование обобщения может также подразумевать (сознательно или нет) *повторное использование программного кода* (*code reuse*) с последующей его реализацией за счет *наследования реализации*. Это очень мощная, иногда даже опасная по силе, интерпретация обобщения. Кроме того, эта интерпретация обобщения принята "по умолчанию".

Наследование реализации — называемое также *выведением подкласса*, *наследованием кода* или *наследованием класса* — 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 свойства, *повторное использование программного кода* и *полиморфизм*.

При использовании в процессе моделирования 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 [88], [34]. Если его тщательно не контролировать и не ограничивать, наследование реализации может принести больше вреда, чем пользы. Теперь мы приступим к обсуждению доводов за и против наследования реализации.

5.3.4.1. Правильный способ использования наследования реализации — наследование посредством расширения

Что касается увязки наследования с обобщением и надлежащего использования наследования реализации, язык UML довольно определенно трактует эти вопросы [76]. Единственным правильным способом использования наследования является наращиваемое определение класса. Подкласс обладает большим количеством свойств (атрибутов и/или методов), чем его суперкласс. Подкласс — "*это нечто вроде*" суперкласса. Подобное наследование известно также как *наследование посредством расширения* или *расширяющее наследование* (*extension inheritance*).

На рис. 2.17 (повторенном здесь для удобства как рис. 5.23) представлен пример расширяющего наследования. Каждый объект `Employee` (Сотрудник) — *это нечто вроде* объекта `Person` (Личность). Это не означает, что объект `Employee` одновременно является экземпляром двух классов: `Person` и `Employee` (см. обсуждение вопросов множественного наследования в разд. 2.1.5.2.2). Объект `Employee` — это экземпляр класса `Employee`.

Класс `Person` на рис. 5.23 — это не абстрактный класс. Могут существовать некоторые объекты `Person`, которые представляют просто некую личность (в том смысле, что они не являются сотрудниками, т.е. 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, объект подкласса не может быть больше подставлен вместо объекта суперкласса.

5.3.4.2. 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 *наследованием посредством ограничения* или *ограничивающим наследованием* (*restriction inheritance*) [74].

На рис. 5.24 приведены два примера ограничивающего наследования. Поскольку наследование нельзя блокировать выборочно, класс окружностей `Circle` должен унаследовать свойства главной и дополнительной осей — `minor_axis` и `major_axis` — от эллипса `Ellipse` и заменить их атрибутом диаметра `diameter`. Аналогично, подкласс `Penguin` (Пингвин) должен унаследовать способность летать (операцию `fly`) от класса птиц `Bird` и заменить ее на операцию плавания `swim`

(пожалуй, полет на "отрицательной" высоте можно с некоторой натяжкой трактовать как плавание).

Ограничивающее наследование проблематично. С точки зрения обобщения подкласс не включает все свойства суперкласса. Объект суперкласса по-прежнему может быть заменен на объект 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, ограничивающее наследование может принести определенные выгоды. Однако, в общем случае ограничивающее наследование вызывает проблемы при сопровождении.

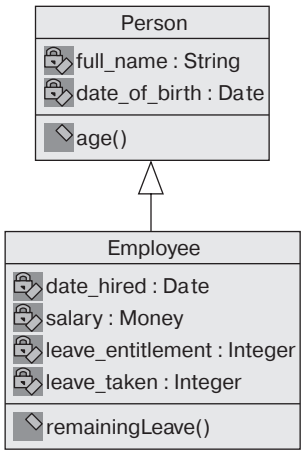

Рис. 5.23. Расширяющее наследование

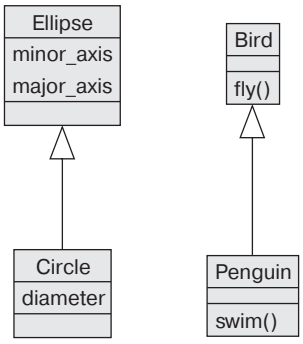

Рис. 5.24. Ограничивающее наследование

5.3.4.3. Неверный способ использования наследования реализации — наследование по удобству

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 классов 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 *наследование по удобству* (*convenience inheritance*) [54], [74].

На рис. 5.25 приведено два примера наследования по удобству. Класс дуг `LineSegment` определен как подкласс класса точек `Point`. Ясно, что дуга — 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, для класса `Point` можно определить атрибуты координат `x_coordinate` и `y_coordinate` и операцию перемещения `Move`. Класс `LineSegment` может унаследовать эти свойства и может определить дополнительную операцию изменения размеров `Resize`. Операцию `Move` необходимо заместить. Аналогично, во втором примере класс автомобилей `Car` наследует свойства класса мопедов `Motorbike` и добавляет некоторые новые свойства.

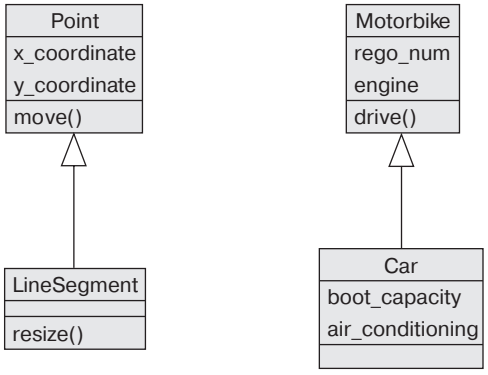

Рис. 5.25. Наследование по удобству

Наследование по удобству неприемлемо. Оно семантически некорректно. И приводит к масштабному замещению. Принцип подставимости, как правило, не работает, поскольку объекты не принадлежат к подобным типам (дуга (`LineSegment`) — не точка (`Point`); автомобиль (`Car`) — не мотоцикл (`Motorbike`)).

На практике, и к большому сожалению, разработчики часто используют наследование по удобству, 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 (в особенности агрегации) отсутствует.

5.3.4.4. Недостатки наследования реализации

Сказанное выше вовсе не означает, что запрещение использования наследования по удобству 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 особенно 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 со следующими перечисленными ниже концептуальными трудностями [88].

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

5.3.4.4.1. 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, а в особенности в ситуации, когда суперкласс можно получить из внешних источников, находящихся вне контроля бригады разработчиков системы.

Рассмотрим ситуацию, при которой суперклассы, которым наследует ваше приложение, образуют часть операционной системы, СУБД или GUI-интерфейса. Если для разработки вашего 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 выказать предусмотрительности, то это несомненно так и есть).

С проблемой изменчивости базового класса трудно справиться, не объявив открытые интерфейсы неизменяемыми или, по меньшей мере, не установив контроля 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. Вот примеры подобных ситуаций [88].

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

Практический вывод из сказанного можно сформулировать следующим образом. 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 гадалок, нельзя. Как гласит шутка, начертанная в виде лозунга на бампере автомобиля: "сумасшествие не наследуется, оно достается вам от ваших детей" [29]. В разд.5.4 рассматриваются некоторые альтернативные методы объектной разработки, которые, не будучи основаны на наследовании, все же обеспечивают требуемые функциональные возможности объектов.

5.3.4.4.2. Замещение и обратные вызовы

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

1. Подкласс может наследовать интерфейс и реализацию метода без внесения каких-либо изменений в реализацию.
2. Подкласс может наследовать код и включить его (вызвать его) в свой собственный метод с той же сигнатурой.
3. Подкласс может наследовать код и затем полностью заместить его новой реализацией с той же сигнатурой.
4. Подкласс может наследовать пустой код (т.е. декларация метода отсутствует), а затем ввести реализацию для метода.
5. Подкласс может наследовать только интерфейс метода (т.е. мы имеем случай наследования интерфейса), а затем ввести реализацию для 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 5.12. Телемаркетинг

Обратитесь к приложению *Телемаркетинг* и, в частности, к примеру 5.4 (разд. 5.1.4.1). Модифицируйте отношение обобщения между классами `Campaign` и `BonusCampaign` таким образом, чтобы включить операции, которые служат примером первых трех методов повторного использования, перечисленных выше.

На рис. 5.26 приведен особый вариант отношения обобщения между классами `Campaign` и `BonusCampaign`, призванный продемонстрировать три метода повторного использования. Этим методам соответствуют следующие операции: `getDateClose` (получить дату завершения [кампании]), `printTicketDetails` (отпечатать подробную информацию о билете) и `computeTicketsLeft` (подсчитать количество оставшихся билетов).

Реализация операции `getDateClose()` наследуется классом `BonusCampaign` и повторно используется им без изменений (это первый метод из приведенного выше списка).

Две оставшиеся операции замещаются в классе BonusCampaign. На модели этот факт отражается за счет дублирования имен операций в BonusCampaign. Предполагается, что операция printTicketDetails представляет второй метод, т.е. метод BonusCampaign.printTicketDetails() вызывает унаследованный метод Campaign.printTicketDetails(), скажем, для печати общего количества билетов, участвующих в кампании. Затем он выводит на печать специфическую информацию о призовых билетах, такую как количество билетных книжек и объем книжек.

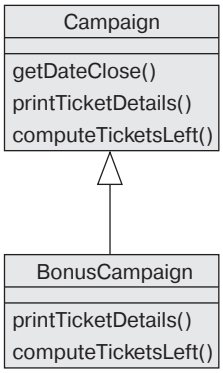

Рис. 5.26. Наследование и замещение (Телемаркетинг)

Теперь предположим, что операция computeTicketsLeft() представляет третий метод повторного использования. Метод BonusCampaign.computeTicketsLeft() является наследником (поскольку это предусмотрено) метода Campaign.computeTicketsLeft(), а затем полностью замещает унаследованный код.

Пример 5.12 демонстрирует влияние замещения на проблему изменчивости базового класса. Он также показывает, что наследование реализации приводит к возникновению сетеобразных структур для путей взаимодействия между объектами, которые, как было показано ранее, отличаются неустойчивостью в больших системах (разд.5.2). При использовании наследования реализации передача сообщений может возникать повсеместно. Помимо простых *прямых вызовов сверху-вниз* (*down-calls*) по иерархии наследования, имеют место также и *обратные вызовы* (*вызовы снизу-вверх*) (*callbacks* (*up-calls*)) (разд.6.2.2.8).

Как замечает Шиперский [88]: "В соединении с наблюдаемым состоянием это приводит к семантике замкнутых вызовов, близкой той, которая характерна для параллельных систем. Произвольный граф вызовов, формируемый сетью взаимодействующих объектов, разрушает классическую схему иерархии уровней и делает замкнутые вызовы нормой."

Ради справедливости следует заметить, что обратные вызовы возможны не только между объектами, связанными отношениями наследования, а всюду, где существуют ссылки между объектами. Вновь приведем замечание Шиперского [88]: "При наличии объектных ссылок ... всякий вызов метода может рассматриваться как потенциальный вызов снизу-вверх, каждый метод может быть потенциально связан с обратным вызовом". Наследование только вносит свой вклад в общую проблему, но, надо заметить, вклад существенный.

5.3.4.4.3. Множественное наследование реализации

Множественное наследование было введено в разд.2.1.5.2.1, однако, до сих пор мы не проводили различия между *множественным наследованием интерфейса* (множественным наследованием от супертипов) 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 присущим системам росте сложности из-за множественного наследования — росте, вызванном отсутствием поддержки *множественной классификации* в объектных системах (разд.2.1.5.2.2). 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. пример в разд.2.1.5.2.2).

5.4. Углубленное моделирование агрегации и делегирования

Агрегация представляет собой третий метод связывания классов в моделях анализа (разд.2.1.4). 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 нам методов управления сложностью больших систем с помощью распределения классов по иерархическим уровням абстракции.

Агрегация (и ее боле сильный вариант — *композиция*) — отношение включения. Составной класс содержит один или более компонентных классов. Компонентный класс является элементом одного или более составных классов. Компонентные классы являются элементами их составных классов (хотя элементы ведут свое собственное существование). Хотя 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. Хотя структура времени компиляции для агрегации аналогична структуре ассоциации, во время выполнения они ведут себя поразному. Агрегация обладает более строгой семантикой, и ответственность за то, чтобы структуры времени выполнения удовлетворяли этой семантике, лежит (к сожалению) на программисте.

5.4.1. Расширение семантики агрегации

Хотя 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 агрегативным конструкциям строгую семантическую интерпретацию, зачастую трактуя их просто как особую форму ассоциации.

Как рассматривалось в разд.4.2.3, можно выделить четыре возможных семантики для агрегации [55].

1. Агрегация типа "Безраздельно обладает".
2. Агрегация типа "Обладает".
3. Агрегация типа "Включает".
4. Агрегация типа "Член".

UML признает только две семантики агрегации, а именно *агрегацию* (ссылочная семантика) и *композицию* (семантика значений) (разд.2.1.4). Теперь будет показано, каким образом можно использовать существующую нотацию UML для представления четырех видов агрегации, указанных выше.

5.4.1.1. Агрегация типа "Безраздельно обладает"

Агрегацию типа *Безраздельно обладает* можно представить в UML как композицию-стереотип, обозначенную с помощью ключевого слова «ExclusiveOwns» и дополнительно ограниченную с помощью ключевого слова (замороженный) [25]. Ограничение *frozen* применяется к компонентному классу. Оно констатирует, что объект компонентного класса не может быть *заново соединен* (в течение своего ЖЦ) с другим составным объектом. Компонентный объект может быть удален вовсе, но не может переключиться на другого владельца.

На рис. 5.27 показано два примера агрегации типа *Безраздельно обладает.* Левая часть рисунка представляет пример моделирования агрегации в UML с использованием семантики значений (заполненный ромб), а правая часть – пример моделирования агрегации в UML с использованием ссылочной семантики (пустой ромб).

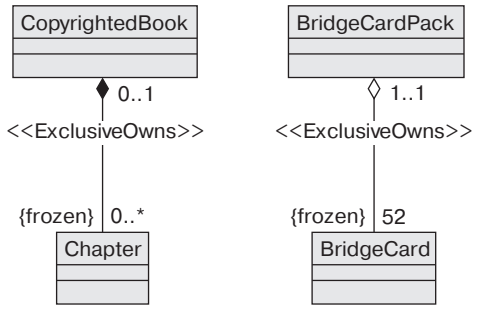

***Рис. 5.27. Агрегация типа* Безраздельно обладает**

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. Соединение заморожено (за счет ограничения *frozen*).

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

5.4.1.2. Агрегация типа "Обладает"

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

На рис. 5.28 показано два примера агрегации типа *Обладает*. Вода (объект Water) может быть перелита из одного кувшина (объект Jug) в другой. Аналогично, шина (объект Tire) может быть переставлена с одного велосипеда (Bicycle) на другой. Благодаря зависимости по существованию, разрушение объекта Jug или Bicycle распространяется вниз на их компонентные объекты.

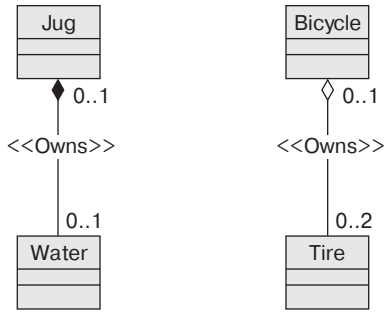

***Рис. 5.28. Агрегация типа* Обладает**

5.4.1.3. Агрегация типа "Включает"

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

Пример агрегации типа *Включает* показан на рис. 5.29. Если тележка (объект Trolley) включает несколько ящиков с пивом (BeerCrate), а каждый ящик с пивом включает несколько бутылок с пивом (BeerBottle), то тележка также включает бутылки с пивом (транзитивность). Если тележка включает ящики с пивом, то ящики с пивом не могут включать тележку (асимметричность).

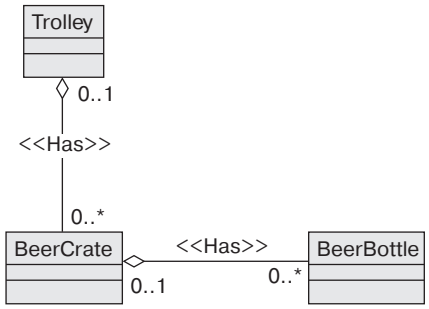

Рис. 5.29. Агрегация типа **Включает**

5.4.1.4. Агрегация типа "Участник"

Агрегация типа *Участник* 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. Благодаря кратности "многие ко многим", агрегацию типа *Участник* можно моделировать в UML только с помощью ссылочной семантики агрегации (пустой ромб).

На рис. 5.30 показано четыре отношения агрегации типа *Участник*. JAD-совещание (объект `JADSession`) (см. разд. 3.2.2.2) состоит из одного модератора и одного или нескольких секретарей (`Scribe`), пользователей (`User`) и разработчиков (`Developer`). Каждый компонентный "объект" может принимать участие, более чем в одном JAD-совещании.

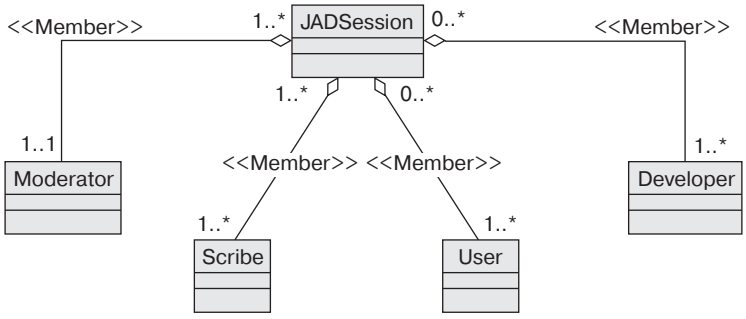

Рис. 5.30. Агрегация типа **Участник**

5.4.2. Агрегация как альтернатива обобщению

Обобщение — это отношение суперкласс–подкласс. Агрегация больше напоминает отношение супермножество–подмножество. Вопреки этому различию обобщение можно представить как агрегацию.

Рассмотрим рис. 5.31. Заказы клиентов, которые не выполнены, могут ожидать некоторых дальнейших действий. Заказ в состоянии ожидания может быть отложенным заказом, который должен быть выполнен после пополнения запаса. Заказ в состоянии

ожидания может быть заказом наперед, если он должен быть выполнен позднее, в сроки, определенные клиентом.

Модель в левой части рисунка представляет собой обобщение заказов клиентов. Класс `Gorder` (Заказ) может быть классом `GPendingOrder` (Ожидающий заказ). Класс `GPendingOrder` может быть классом `GBackOrder` (Невыполненный заказ) или классом `GFutureOrder` (Заказ наперед). Наследование гарантирует разделение атрибутов и операций вниз по дереву обобщения.

Аналогичную семантику можно смоделировать с помощью агрегации, показанной в правой части рисунка 5.31. Классы `ABackOrder` и `AFutureOrder` включают атрибуты и операции класса `APendingOrder`, которые, в свою очередь, включает класс `AOrder`.

Хотя две модели, показанные на рис. 5.31, отражают аналогичную 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 `GBackOrder` является также объектом класса и `GPendingOrder` и `Gorder`. Для объекта `GBackOrder` существует один *идентификатор объекта* (OID). С другой стороны, конкретный объект `ABackOrder` состоит из трех отдельных объектов, каждый из которых обладает собственным идентификатором объекта, — собственно объект `ABackOrder`, содержащийся в нем объект `APendingOrder` и содержащийся в нем объект `AOrder`.

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

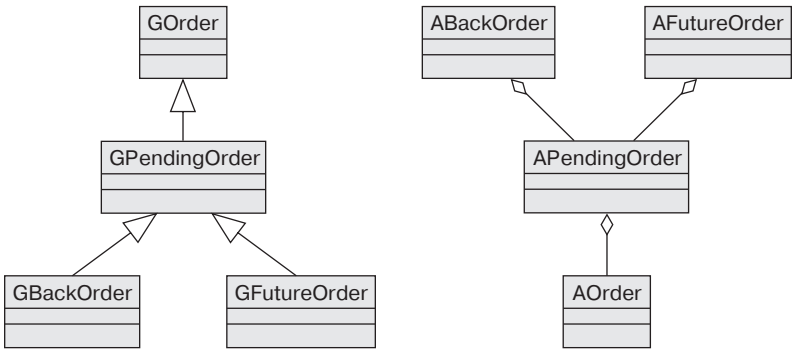

Рис. 5.31. Альтернатива: обобщение или агрегация

5.4.2.1. Делегирование и системы-прототипы

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

При подходе, основанном на делегировании, функциональные возможности системы реализуются с помощью включения (клонирования) функций существующих объек-

тов 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*. Системы, построенные подобным способом из существующих объектов-прототипов, называются *системами-прототипами*.

Объект может быть связан отношением делегирования с любым другим идентифицируемым и видимым объектом в системе [49]. Когда внешний объект получает сообщение и не в состоянии выполнить обслуживание самостоятельно, он делегирует выполнение обслуживания внутреннему объекту. При необходимости внутренний объект может переслать сообщение любого из своих собственных внутренних объектов.

Интерфейсы внутреннего объекта могут быть видимы или невидимы для объектов, отличных от внешнего объекта. Для контроля уровня видимости внутренних объектов можно использовать четыре типа агрегации, определенные в разд.5.4.1. Например, внешний объект может раскрыть интерфейс внутреннего объекта как свой собственный в более слабой форме агрегации (например, в такой как агрегация типа *Включает* или *Участник*). При более сильной форме агрегации внешний объект может скрыть интерфейс своего внутреннего объекта от внешнего мира (вводя, таким образом, уровень *инкапсуляции*).

5.4.2.2. Делегирование или наследование

Покажем, что с помощью делегирования можно моделировать наследование и наоборот. Это значит, что одни и те же функциональные возможности можно реализовать как с помощью наследования, так и делегирования. Согласие в этом вопросе впервые было достигнуто на конференции в США в городе Орландо, штат Флорида, в 1987 году и теперь известно как Орландское соглашение [81].

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 [88].

С точки зрения *повторного использования* 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 обслуживание).

В случае делегирования после того, как управление передано от внешнего объекта внутреннему объекту, оно остается у последнего. Любая авторекурсия должна быть явно запланирована и спроектирована в рамках делегирования. При использовании наследования реализации авторекурсия всегда случайна — она не запланирована и наложена как программная "заплата" [88]. 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 определяется 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 [49]. Объекты объединяются естественным способом для формирования масштабных решений и могут эволюционировать непредвиденным образом. В следующем разделе приводится другая точка зрения на эти же вопросы.

5.4.3. Агрегация и голоны — интеллектуальное орудие

В работах [54] и [55] с целью обуздания сложности объектных моделей был предложен новый подход для описания архитектуры ПО, основанный на интерпретации естественных систем Артура Кёстлера (Arthur Koestler) [44], [45]. Центральной концепцией является идея так называемых "*голонов*" ("*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 являют как независимые свойства целого, так и взаимозависимые свойства части. По Артуру Кёстлеру они представляют собой агрегации *голонов* от греческого слова *holos*, обозначающего целое (англ. *whole*) с суффиксом, измененным на *-он* (*on*), означающим частицу или часть (как в словах протон или нейтрон) [44].

Части и целое в абсолютном смысле не существуют в живых организмах и даже в социальных системах. Голоны образуют иерархические уровни соответствующей сложности, например, в биологических организмах можно различать иерархию атомов, молекул, органоидов, клеток, тканей, органов и систем органов. Подобные иерархии голонов называются *голократиями*.

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

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

Отдельные голоны, таким образом, представляются четырьмя характеристиками.

1. Его внутренние правила (взаимодействия между ними могут формировать уникальные шаблоны).
2. Самоутверждающаяся агрегация подчиненных голонов.
3. Тенденция к интеграции по отношению к высшим голонам.
4. Отношения с соседними голонами.

Удачные системы упорядочены в виде голократий, которые скрывают 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 реализаций 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 (голонов) достигается за счет требования, согласно которому объекты должны "признавать интерфейсы друг друга" [28]. 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 (голонов) сверху-вниз и снизу-вверх.

Однако агрегация не позволяет моделировать необходимые возможности взаимодействия между голонами 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 [28], в особенности поддержки подхода на основе голонов и применения четырех видов агрегации. Мы надеемся, что рассмотренный выше подход послужит читателям интеллектуальным орудием в их собственных изысканиях.

Резюме

В этой главе было завершено рассмотрение анализа требований. Была тщательно исследована поддержка объектной технологии для разработки крупномасштабных систем. 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*. Они позволяют моделировать свойства, выходящие за пределы предопределенных в UML. За счет изобретательности в этой главе удалось широко использовать стереотипы. Стереотипы не следует путать с *ограничениями*, *примечаниями* и *дескрипторами* UML.

Открытая и закрытая *видимость*, рассмотренная в предыдущей главе, обеспечивает только базовую поддержку важного понятия *инкапсуляции*. *Защищенная видимость* позволяет управлять инкапсуляцией в рамках структур наследования. *Видимость уровня класса* (в противоположность видимости отдельных атрибутов и операций) представляет собой еще одну важную концепцию, касающуюся наследования. Понятие друже-

ственности позволяет "прорваться" сквозь инкапсуляцию для обработки специфических ситуаций.

Язык UML включает некоторые 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 классом* и *материализованным классом*.

Современные программные системы отличаются большой сложностью. Поэтому важно, что решения по моделированию упрощают разработку систем и позволяют уменьшить присущую им сложность там, где это возможно. Пожалуй, наиболее важным механизмом, позволяющим обуздать сложность ПО, является представление архитектуры системы в виде *многоуровневой иерархии*. Надлежащее структурирование классов в виде *пакетов*, организованных в соответствии с *подходом ВСЕ*, является важной архитектурной задачей.

Концепция *обобщения и наследования* представляет собой в моделировании "палку о двух концах". С одной стороны, она способствует повторному использованию ПО 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 *голона* дает интересный взгляд на способ построения сложных систем.

Вопросы

В1. Что называется в UML "профилем"? Приведите пример.

В2. 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?

В3. Объясните различие между ограничением и примечанием.

В4. Обозначают ли термины "инкапсуляция" и "видимость" одно и то же понятие? Объясните вашу точку зрения.

В5. 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.

В6. 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.

В7. В чем заключаются преимущества использования производной информации для моделирования?

В8. Когда материализованный класс должен заменять ассоциативный класс? Приведите пример (отличный от приведенного в этой книге).

В9. Какова сложность сети из девяти классов (измеряемая как количество возможных соединений между этими классами)? Изобразите иерархию для этих девяти классов, состоящую из четырех уровней. Какого понижения сложности можно достичь за счет этой четырехуровневой иерархии?

В10. Предположим, что модель классов для банковского приложения содержит класс `InterestCalculation` (Подсчет процентов). Какому пакету *ВСЕ* должен принадлежать этот класс? Объясните свою точку зрения.

В11. В чем состоит принцип подставимости? Дайте пояснения.

В12. Объясните различия между наследованием интерфейса и наследованим реализации.

В13. В чем состоит проблема изменчивости базового класса? Каковы основные причины изменчивости базовых классов?

В14. Объясните различия между агрегациями типа *Безраздельно владеет* и *Владеет*. Какие преимущества при моделировании дает разделение агрегации на два этих вида?

В15. Сравните наследование и делегирование. В чем их схожесть? В чем различие между ними?

Упражнения

У1. Обратитесь к рис. 2.13 (разд.2.1.3.2). Предположим, что преподаватель, который руководит курсом, должен также вести этот курс.

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

У2. Обратитесь к рис. 2.17 (разд.2.1.3.2) и 2.18 (разд.2.1.5.1). Объедините оба рисунка в виде одной модели классов.

Разработайте схему видимости для модели классов. Дайте к ней пояснения.

Каким уровнем видимости должны обладать унаследованные атрибуты класса Employee и класса Manager? Объясните ваше решение.

У3. *Запись на университетские курсы* — обратитесь к примеру 4.9 (разд.4.2.3.3).

Возможно ли введение производной информации в модель, показанную на рис. 4.6? Если это так, модифицируйте диаграмму.

У4. Обратитесь к рис. 2.15 (разд.2.1.3.4). 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. 2.15 таким образом, чтобы учесть в модели это ограничение. Используйте при этом материализованный класс. Введите в модель любые предположения и поясните их.

У5. *Измерение затрат на рекламу* — обратитесь к постановке задачи, приведенной в конце главы 3.

Изобразите диаграмму пакетов для АЕМ-системы. Объясните введенные вами предположения.

У6. *Магазин видеопродукции* — обратитесь к примеру 4.10 (разд.4.2.4.3).

Перерисуйте диаграмму на рис. 4.7 с использованием отношений реализации.

У7. *Магазин видеопродукции* — обратитесь к примеру 4.10 (разд.4.2.4.3).

Перерисуйте диаграмму на рис. 4.7, используя агрегацию вместо обобщения. Приведите доводы за и против новой модели.