5

Диаграммы классов: дополнительные понятия

Описанные ранее в главе 4 понятия соответствуют основной нотации диаграмм классов. Именно эти понятия нужно постичь и освоить прежде всего, поскольку они на 90% удовлетворят ваши потребности при построении диаграмм классов.

Однако диаграммы классов могут содержать множество обозначений для представления различных дополнительных понятий. Я сам обращаюсь к ним не слишком часто, но в отдельных случаях они оказываются весьма удобными. Рассмотрим последовательно эти дополнительные понятия, обращая внимание на особенности их применения.

Возможно, при чтении этой главы вы столкнетесь с некоторыми трудностями. Порадую вас: эту главу можно без всякого ущерба пропустить при первом чтении книги и вернуться к ней позже.

Ключевые слова

Одна из трудностей, сопряженных с графическими языками, состоит в необходимости запоминать значения символов. **Когда символов слишком много, пользователям трудно запомнить, что означает каждый из них. Поэтому в UML нередко предпринимаются попытки уменьшить количество символов, заменяя их ключевыми словами.** Когда требуется смоделировать конструкцию, отсутствующую в UML, но похожую на один из его элементов, возьмите символ существующей конструкции UML, пометив его ключевым словом, чтобы показать, что используется нечто другое.

Примером может служить интерфейс. Интерфейс (interface) в UML (*стр. 96*) означает класс, в котором все операции открытые и не имеют тел методов. Это соответствует интерфейсам в Java, COM (Component Object Module) и CORBA. **Поскольку это специальный вид класса, то он изображается с помощью пиктограммы с ключевым словом** «inter-

face». **Обычно ключевые слова представляются в виде текста, заключенного во французские кавычки («елочки»).** Вместо ключевых слов можно использовать специальные значки, но тем самым вы заставляете всех запоминать их значения.

Некоторые ключевые слова, такие как {abstract}**, заключаются в фигурные скобки. В действительности никогда не понятно, что формально должно быть в кавычках, а что в фигурных скобках.** К счастью, если вы ошибетесь, то заметят это только настоящие знатоки UML. Но лучше быть внимательными.

Некоторые ключевые слова настолько общеупотребительны, что часто заменяются сокращениями: «interface» **часто сокращается до** «I»**, а** {abstract} **— до** {A}**. Такие сокращения очень полезны, особенно на белых досках, однако их применение не стандартизовано. Поэтому если вы их употребляете, то не забудьте найти место для расшифровки этих обозначений.**

В UML 1 кавычки применялись в основном для стереотипов. В UML версии 2 стереотипы определены очень кратко, и разговор о том, что является стереотипом, а что нет, выходит за рамки этой книги. Однако из-за UML 1 многие разработчики употребляют термин «стереотип» в качестве синонима ключевого слова, хотя теперь это неверно.

Стереотипы используются как части профилей. **Профиль** (profile) берет часть UML и расширяет его с помощью связанной группы стереотипов для определенной цели, например для бизнес-моделирования. Полное описание семантики профилей выходит за рамки этой книги. Пока вы не займетесь разработкой серьезной метамодели, вам вряд ли понадобится создавать профиль самому. Скорее всего, вы возьмете профиль, ранее созданный для конкретного варианта моделирования, и, к счастью, применение профиля не требует знания чудовищного количества подробностей, связанных с метамоделью.

Ответственности

Часто бывает удобным показывать ответственности класса (*стр. 90*) на диаграмме классов. Лучший способ показать их состоит в том, чтобы располагать строки комментария в их собственной ячейке (рис. 5.1). При желании ячейке можно присвоить имя, но я обычно этого не делаю, поскольку вероятность возникновения путаницы невелика.

Статические операции и атрибуты

Если в UML ссылаются на операции и атрибуты, принадлежащие классу, а не экземпляру класса, то они называются статическими. Это эквивалентно статическим членам в C-подобных языках. **На диаграмме класса статические элементы подчеркиваются (рис. 5.2).**

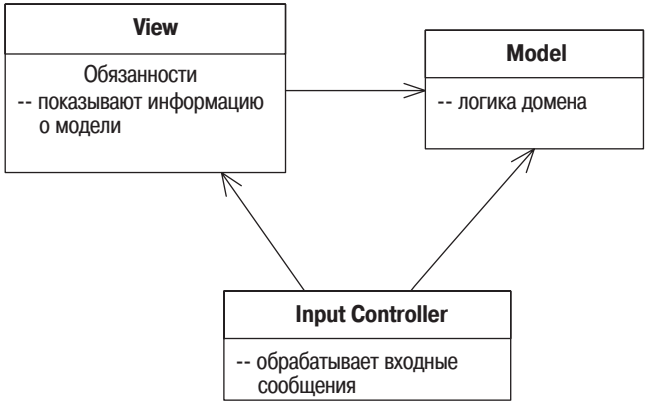

Рис. 5.1. Представление обязанностей на диаграмме классов

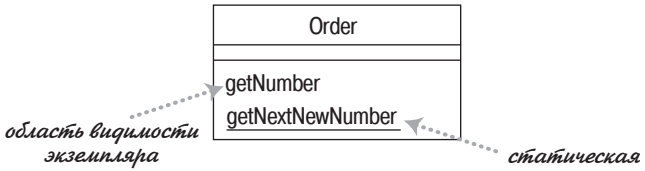

Рис. 5.2. Статическая нотация

Агрегация и композиция

К одним из наиболее частых источников недоразумений в UML –можно отнести агрегацию и композицию. В нескольких словах это можно объяснить так: **Агрегация (aggregation) – это отношение типа «часть целого».** Точно так же можно сказать, что двигатель и колеса представляют собой части автомобиля. Звучит вроде бы просто, однако при рассмотрении разницы между агрегацией и композицией возникают определенные трудности.

До появления языка UML вопрос о различии между агрегацией и композицией у аналитиков просто не возникал. Осознавалась подобная неопределенность или нет, но свои работы в этом вопросе аналитики совсем не согласовывали между собой. В результате многие разработчики считают агрегацию важной, но по совершенно другой причине. Язык UML включает агрегацию (рис. 5.3) но семантика ее очень расплывчата. Как говорит Джим Рамбо (Jim Rumbaugh): «Можно представить себе агрегацию как плацебо для моделирования» [40].

Наряду с агрегацией в языке UML есть более определенное свойство – композиция (composition). На рис. 5.4 экземпляр класса `Point` (Точка) может быть частью многоугольника, а может представлять центр окружности, но он не может быть и тем и другим одновременно. Главное

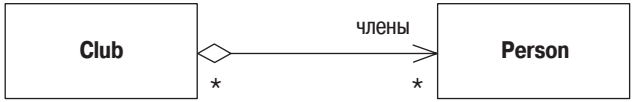

Рис. 5.3. *Агрегация*

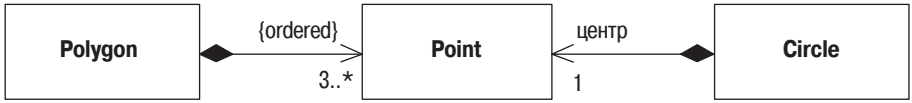

Рис. 5.4. *Композиция*

правило состоит в том, что хотя класс может быть частью нескольких других классов, но любой экземпляр может принадлежать только одному владельцу. На диаграмме классов можно показать несколько классов потенциальных владельцев, но у любого экземпляра класса есть только один объект-владелец.

Вы заметите, что на рис 5.4 **я не показываю обратные кратности. В боль- шинстве случаев, как и здесь, они равны 0..1. Единственной альтерна- тивой является значение 1, когда класс-компонент разработан таким образом, что у него только один класс-владелец.**

Правило «нет совместного владения» является ключевым в композиции. Другое допущение состоит в том, что если удаляется многоугольник (Polygon), то автоматически должны удалиться все точки (Points), которыми он владеет.

Композиция – это хороший способ показать свойства, которыми владеют по значению, свойства объектов-значений (*стр. 100*) или свойства, которые имеют определенные и до некоторой степени исключительные права владения другими компонентами. Агрегация совершенно не имеет смысла; поэтому я не рекомендовал бы применять ее в диаграммах. Если вы встретите ее в диаграммах других разработчиков, то вам придется покопаться, чтобы понять их значение. Разные авторы и команды разработчиков используют их в совершенно разных целях.

Производные свойства

Производные свойства (derived properties) могут вычисляться на основе других значений. Говоря об интервале дат (рис. 5.5), мы можем рассуждать о трех свойствах: начальной дате, конечной дате и количестве дней за данный период. Эти значения связаны, поэтому мы можем сказать, что длина является производной двух других значений.

С точки зрения программного обеспечения образование производных можно интерпретировать двумя различными путями. Можно использовать образование производных для обозначения различия между вычисляемым и хранимым значениями. В этом случае, глядя на рис. 5.5,

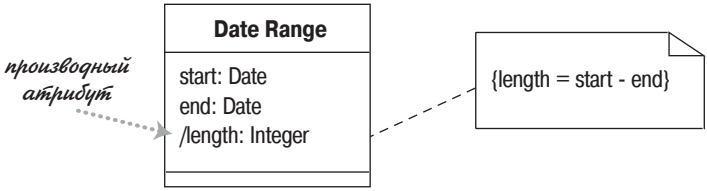

Рис. 5.5. Производный атрибут для временного интервала

мы скажем, что начальная (`start`) и конечная (`end`) даты хранятся, а длина (`length`) вычисляется. И хотя это наиболее распространенное применение, меня это не очень привлекает, поскольку слишком раскрывает внутреннее устройство класса `DateRange` (Интервал дат).

Я предпочитаю рассматривать это как связь между значениями. В данном случае мы говорим, что между тремя значениями существует связь, но не важно, какое из трех значений вычисляется. При этом можно произвольно выбирать, какой атрибут отмечать как производный, а можно и вовсе этого не делать, но все же полезно напомнить разработчикам о связи. Такое применение имеет смысл в концептуальных диаграммах.

Образование производных может быть применено к свойствам с помощью ассоциаций. В этом случае вы просто отмечаете имя символом «/».

Интерфейсы и абстрактные классы

Абстрактный класс (abstract class) – это класс, который нельзя реализовать непосредственно. Вместо этого создается экземпляр подкласса. Обычно абстрактный класс имеет одну или более абстрактных операций. У абстрактной операции (abstract operation) нет реализации; это чистое объявление, которое клиенты могут привязать к абстрактному классу.

Наиболее распространенным способом обозначения абстрактного класса или операции в языке UML является написание их имен курсивом. Можно также сделать свойства абстрактными, определяя абстрактное свойство или методы доступа. Курсив сложно изобразить на доске, поэтому **можно прибегнуть к метке:** `{abstract}`.

Интерфейс – это класс, не имеющий реализации, то есть вся его функциональность абстрактна. Интерфейсы прямо соответствуют интерфейсам в C# и Java и являются общей идиомой в других типизированных языках. Интерфейс обозначается ключевым словом `«interface»`.

Классы обладают двумя типами отношений с интерфейсами: предоставление или требование. Класс **предоставляет интерфейс**, если его можно заменить на интерфейс. В Java и .NET класс может сделать это, реализуя интерфейс или подтип интерфейса. В C++ создается подкласс класса, являющегося интерфейсом.

Класс **требует интерфейс**, если для работы ему нужен экземпляр данного интерфейса. По сути дела, это зависимость от интерфейса.

На рис. 5.6 эти отношения демонстрируются в действии на базе небольшого набора классов, заимствованных из Java. Я мог бы написать класс `Order` (Заказ), содержащий список позиций заказа (`Line Items`). Поскольку я использую список, то класс `Order` зависит от интерфейса `List` (Список). Предположим, что он вызывает методы `equals`, `add` и `get`. При выполнении связывания объект `Order` действительно будет использовать экземпляр класса `ArrayList`, но ему не нужно знать, что необходимо вызывать эти три метода, поскольку они входят в состав интерфейса `List`.

Класс `ArrayList` – это подкласс класса `AbstractList`. Класс `AbstractList` предоставляет некоторую, но не всю реализацию поведения интерфейса `List`. В частности, метод `get` – абстрактный. В результате `ArrayList` реализует метод `get`, а также переопределяет некоторые другие операции класса `AbstractList`. В данном случае он переопределяет метод `add`, но вполне удовлетворен наследованием реализации метода `equals`.

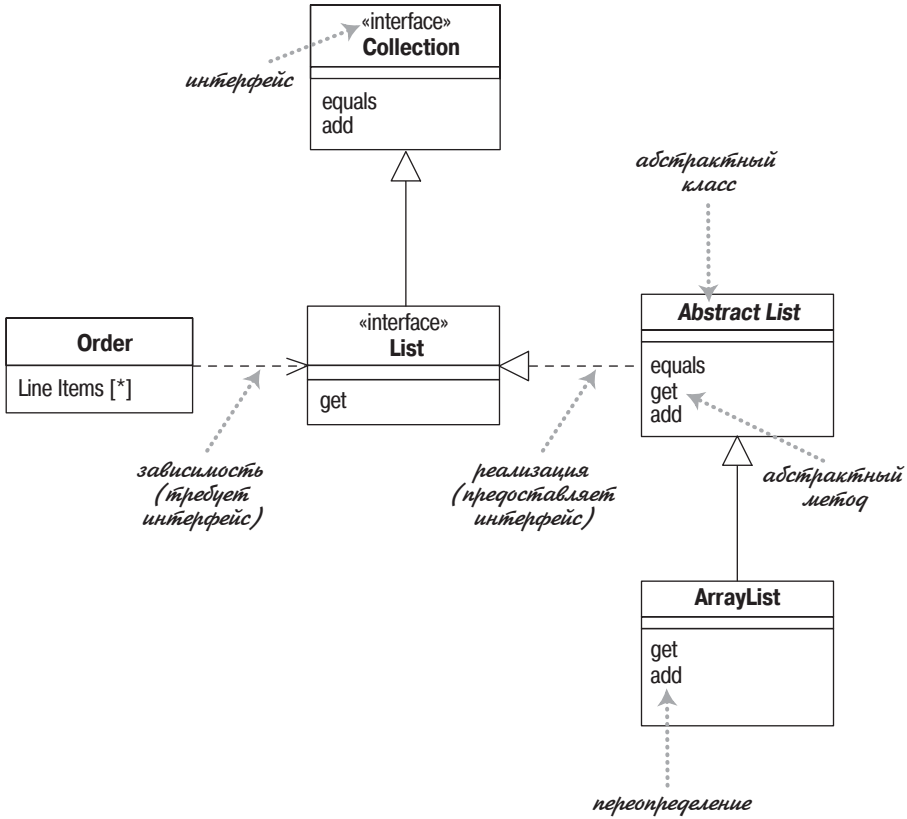

Рис. 5.6. Пример интерфейсов и абстрактного класса на языке Java

Почему бы мне просто не отказаться от этого и не заставить `Order` прямо использовать `ArrayList`? Применение интерфейса позволяет мне получить преимущество при последующем изменении реализации, если потребуется. Другой способ реализации может оказаться более производительным – он может предоставить функции работы с базой данных или другие возможности. Программируя интерфейс, а не реализацию, я избегаю необходимости переделывать весь код, когда достаточно изменить реализацию класса `List`. Следует всегда стараться программировать интерфейс так, как показано выше, то есть всегда использовать наиболее общий тип.

Относительно вышесказанного приведу один практический совет. Когда программисты применяют коллекцию, подобную приведенной здесь, они обычно инициализируют ее при объявлении, например:

```
private List lineItems = new ArrayList();
```

Обратите внимание, что это определенно приводит к зависимости `Order` от конкретного `ArrayList`. С точки зрения теории это проблема, но на практике разработчиков это не беспокоит. Поскольку `lineItems` объявлен как `List`, то никакая другая часть класса `Order` не зависит от `ArrayList`. При необходимости изменить реализацию нужно побеспокоиться лишь об одной строке кода инициализации. Общепринято ссылаться на конкретный класс единожды – при создании, а впоследствии использовать только интерфейс.

Полная нотация на рис. 5.6 – это один из способов обозначения интерфейса. На рис. 5.7 показана более компактная нотация. Тот факт, что `ArrayList` реализует `List` и `Collection`, показан с помощью кружков, называемых часто «леденцами на палочках». То, что `Order` требует интерфейс `List`, показано с помощью значка «гнездо». Связь совершенно очевидна.

В UML уже применялась нотация «леденцов на палочках», но гнездовая нотация – это новинка UML 2. (Мне кажется, это моя любимая нотация из добавленных.) Возможно, вы встретите более старые диаграммы, использующие стиль, представленный на рис. 5.8, где зависимость основана на нотации леденцов.

Любой класс – это сочетание интерфейса и реализации. Поэтому мы часто можем видеть, что объект используется посредством интерфейса одного из его суперклассов. Определенно, было бы допустимо исполь-

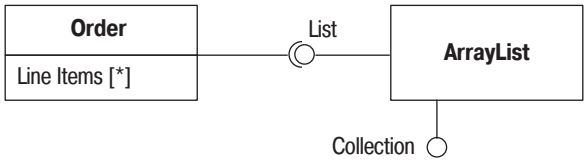

Рис. 5.7. Шарово-гнездовая нотация

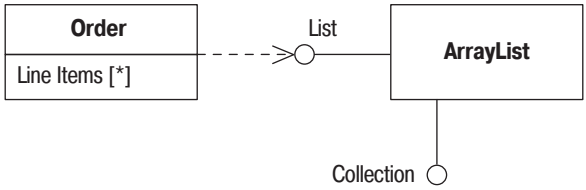

Рис. 5.8. Более старое обозначение зависимостей с помощью «леденцов на палочке»

зовать для суперкласса нотацию леденцов, поскольку суперкласс – это класс, а не чистый интерфейс. Но я обхожу эти правила для ясности.

Разработчики сочли, что нотация леденцов полезна не только для диаграмм классов, но и в других местах. Одна из вечных проблем диаграмм взаимодействий заключается в том, что они не обеспечивают хорошую визуализацию полиморфного поведения. Хотя это нормативное применение, вы можете обозначить такое поведение вдоль линий, как на рис. 5.9. Здесь, как вы можете видеть, **хотя у нас есть экземпляр класса** `Salesman`**, который используется объектом** `Bonus Calculator` **как таковой, но объект** `Pay Period` **использует** `Salesman` **только через его интерфейс** `Employee`. (Тот же самый прием может применяться и в случае коммуникационных диаграмм.)

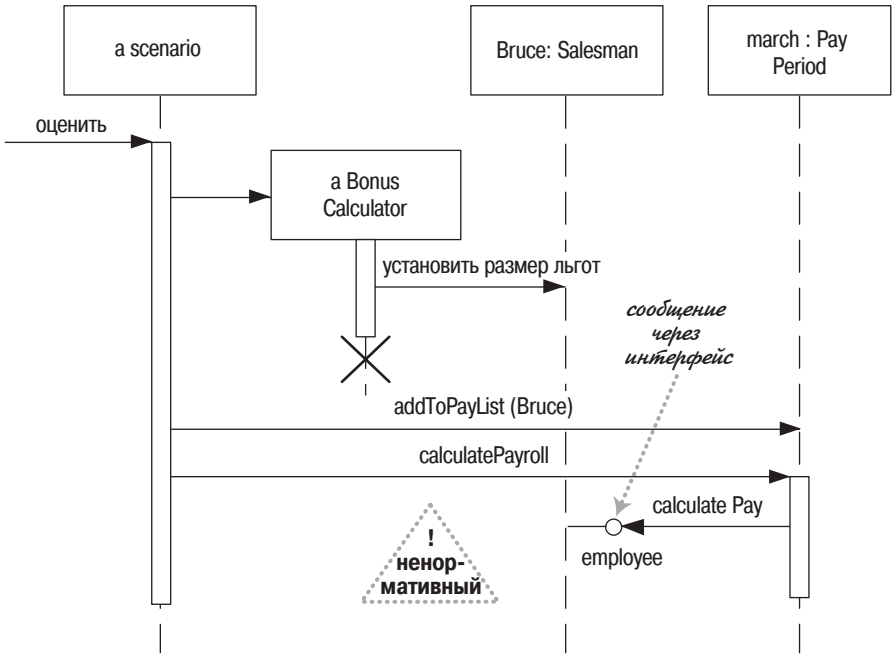

Рис. 5.9. Представление полиморфизма на диаграммах последовательности с помощью нотации леденцов

Read-Only и Frozen

На *стр. 64* я описал ключевое слово {readOnly} (только для чтения). Этим ключевым словом обозначается свойство, которое клиенты могут только читать, но не могут обновлять. Подобное, но несколько отличающееся ключевое слово {frozen} (замороженный) было в UML 1. Свойство находится в состоянии frozen, если оно не может быть изменено в течение жизни объекта; такие свойства часто называются неизменными (immutable). Оно было исключено из UML 2, но понятие frozen очень полезное, поэтому я буду применять его по-прежнему. Наряду с обозначением отдельных свойств, ключевое слово frozen можно применять к классу для указания того, что все свойства всех экземпляров класса находятся в состоянии frozen. (До меня дошли слухи, что ключевое слово frozen скоро будет восстановлено.)

Объекты-ссылки и объекты-значения

Одна из наиболее общих черт объектов заключается в том, что они обладают индивидуальностью (identity). Это правда, но все обстоит не столь просто, как может показаться. На практике оказывается, что индивидуальность важна для объектов-ссылок, но она не так важна для объектов-значений.

Объекты-ссылки (reference objects) – это такие объекты, как Customer (Клиент). В данном случае индивидуальность очень важна, поскольку в реальном мире конкретному классу обычно должен соответствовать только один программный объект. Любой объект, который обращается к объекту Customer, может воспользоваться соответствующей ссылкой или указателем. В результате все объекты, обращающиеся к данному объекту Customer, получат доступ к одному и тому же программному объекту. Таким образом, изменения, вносимые в объект Customer, будут доступны всем пользователям данного объекта.

Если имеются две ссылки на объект Customer и требуется установить их тождественность, то обычно сравниваются индивидуальности тех объектов, на которые указывают эти ссылки. Создание копий может быть запрещено; если же оно разрешено, то, как правило, применяется редко – возможно, для архивирования или репликации через компьютерную сеть. Если копии созданы, то необходимо обеспечить синхронизацию вносимых в них изменений.

Объекты-значения (value objects) – это такие объекты, как Date (Дата). Как правило, один и тот же объект в реальном мире может быть представлен целым множеством объектов значений. Например, вполне нормально, когда имеются сотни объектов со значением «1 января 2004 года». Все эти объекты являются взаимозаменяемыми копиями. При этом новые даты создаются и уничтожаются достаточно часто.

Если имеются две даты и надо установить, тождественны ли они, то вполне достаточно просто посмотреть на их значения, а не устанавливать их индивидуальность. Обычно это означает, что в программе необходимо определить оператор проверки равенства, который бы проверял в датах год, месяц и день (каким бы ни было их внутреннее представление). Обычно каждый объект, ссылающийся на 1 января 2004 года, имеет собственный специальный объект, однако иногда даты могут быть объектами общего пользования.

Объекты-значения должны быть постоянными. Другими словами, не должно допускаться изменение значения объекта-даты «1 января 2004 года» на «2 января 2004 года». Вместо этого следует создать новый объект «2 января 2004 года» и использовать его вместо первого объекта. Причина запрета подобного изменения заключается в следующем: если бы эта дата была объектом общего пользования, то ее обновление могло бы повлиять на другие объекты непредсказуемым образом. Данная проблема известна как **совмещение имен** (aliasing).

В прежнее время различие между объектами-ссылками и объектами-значениями было более четким. Объекты-значения являлись встроенными элементами системы типов. В настоящее время можно расширить систему типов с помощью собственных классов, поэтому данный аспект требует более внимательного отношения.

В языке UML используется концепция **типа данных**, который представляется ключевым словом на символе класса. Строго говоря, тип данных не идентичен объекту-значению, поскольку типы данных не могут иметь индивидуальности. Объекты-значения могут иметь индивидуальность, но она не используется для проверки равенства. В языке Java примитивы могут быть типами данных, но не даты, хотя они могут быть объектами-значениями. Если требуется их выделить, то при создании взаимосвязи с объектом-значением я использую композицию. Можно также применить ключевое слово для типа значения; на мой взгляд, стандартными являются слова «`value`» и «`struct`».

Квалифицированные ассоциации

Квалифицированная ассоциация в языке UML эквивалентна таким известным понятиям в языках программирования, как ассоциативные массивы (associative arrays), проекции (maps), хеши (hashes) и словари (dictionaries). Рисунок 5.10 иллюстрирует способ представления ассоциации между классами `Order` (Заказ) и `Order Line` (Строка заказа), в котором используется квалификатор. Квалификатор указывает, что в соответствии с заказом для каждого экземпляра продукта (`Product`) может существовать только одна строка заказа.

С точки зрения программного обеспечения такая квалифицированная ассоциация может повлечь создание интерфейса следующего вида:

***Рис. 5.10.** Квалифицированная ассоциация*

```
class Order ...
public OrderLine getLineItem(Product aProduct);
public void addLineItem(Number amount, Product forProduct);
```

Таким образом, любой доступ к определенной строке заказа требует подстановки некоторого продукта в качестве аргумента, в предположении, что это структура данных, состоящая из ключа и значения.

Разработчиков часто ставит в тупик кратность квалифицированных ассоциаций. На рис. 5.10 заказ может иметь несколько позиций заказа (`Line Items`), но кратность квалифицированной ассоциации – это кратность в контексте квалификатора. Поэтому диаграмма говорит, что в заказе имеется 0..1 позиций заказа на продукт. Кратность, равная 1, означает, что в заказе должна быть одна позиция заказа для каждого продукта. Кратность * означает, что для любого продукта может существовать несколько позиций заказа, но доступ к позициям заказа индексируется по продукту.

В ходе концептуального моделирования я использую конструкцию квалификатора только для того, чтобы показать ограничения относительно отдельных позиций – «единственная строка заказа для каждого продукта в заказе».

Классификация и обобщение

Мне часто приходится слышать суждения разработчиков о механизме подтипов как об отношении *является* (*это* [*есть*]). Я настоятельно рекомендую держаться подальше от такого представления. Проблема заключается в том, что выражение *является* может иметь самый разный смысл.

Рассмотрим следующие предложения.

1. Шеп – это бордер-колли.
2. Бордер-колли – это собака.
3. Собаки являются животными.
4. Бордер-колли – это порода собак.
5. Собака – это биологический вид.

Теперь попытаемся скомбинировать эти фразы. При объединении первого и второго предложений получаем «Шеп – это собака»; второе и третье предложения в результате дают «бордер-колли – это живот-

ные». Объединение первых трех фраз дает «Шеп – это животное». Чем дальше, тем лучше. Теперь попробуем первое и четвертое предложения: «Шеп – это порода собак». В результате объединения второго и пятого предложений получим «бордер-колли – это биологический вид». Это уже не так хорошо.

Почему некоторые из этих фраз можно комбинировать, а другие нельзя? Причина в том, что некоторые предложения представляют собой **классификацию** – объект Шеп (Shep) является экземпляром типа Бордер-Колли (Border Collie), в то время как другие предложения представляют собой **обобщение** – тип Бордер-Колли является подтипом типа Собака (Dog). Обобщение транзитивно, а классификация – нет. Если обобщение следует за классификацией, то их можно объединить, а если наоборот – классификация следует за обобщением, то нельзя.

Смысл сказанного в том, что с отношением *является* следует обращаться весьма осторожно. Его использование может привести к неверному применению подклассов и к ошибочному распределению ответственностей. В приведенном примере хорошими тестами для проверки подтипов могут служить следующие фразы: «Собаки являются разновидностью Животных» и «Каждый экземпляр Бордер-Колли является экземпляром Собаки».

В языке UML обобщение обозначается соответствующим символом. Для того чтобы показать классификацию, применяется зависимость с ключевым словом «instantiate».

Множественная и динамическая классификация

Классификация служит для обозначения отношения между некоторым объектом и его типом. В основных языках программирования предполагается, что объект относится к единственному классу. Но в UML имеется больше возможностей для классификации.

При **однозначной классификации** (single classification) любой объект принадлежит единственному типу, который может быть унаследован от супертипов. Во **множественной классификации** (multiple classification) объект может быть описан несколькими типами, которые не обязательно должны быть связаны наследованием.

Множественная классификация отличается от множественного наследования. При множественном наследовании тип может иметь несколько супертипов, но для каждого объекта должен быть только один тип. Множественная классификация допускает принадлежность объекта нескольким типам, при этом не требуется определять специальный тип.

В качестве примера рассмотрим тип `Person` (Личность), подтипами которого являются `Male` (Мужчина) или `Female` (Женщина), `Doctor` (Доктор) или `Nurse` (Медсестра), `Patient` (Пациент) или вообще никто (рис. 5.11). Множественная классификация позволяет некоторому

объекту иметь любой из этих типов в любом допустимом сочетании, при этом нет необходимости определять отдельные типы для всех возможных комбинаций.

Если вы используете множественную классификацию, то должны быть уверены в том, что четко определили, какие комбинации являются допустимыми. В языке UML версии 2 это осуществляется помещением каждого обобщающего отношения в **множество обобщения**. На диаграмме классов вы помечаете линию обобщения с помощью имени множества обобщения, которое в UML 1 называется дискриминатором. Единственная классификация соответствует одному безымянному множеству обобщения.

По определению множества обобщения не пересекаются: каждый экземпляр супертипа может быть экземпляром только одного подтипа из данного множества. Если вы соединяете линии обобщений с одной стрелкой, то они должны входить в одно и то же множество обобщения, как показано на рис. 5.11. Альтернативный способ – изобразить несколько стрелок с одинаковой текстовой меткой.

В качестве иллюстрации отметим на диаграмме следующие допустимые комбинации подтипов: (`Female`, `Patient`, `Nurse`); (`Male`, `Physiotherapist` (Физиотерапевт)); (`Female`, `Patient`) и (`Female`, `Doctor`, `Surgeon` (Хирург)). Комбинация (`Patient`, `Doctor`, `Nurse`) является недопустимой, поскольку она содержит два типа из множества обобщения `role` (роль).

Возникает еще один вопрос: может ли объект изменить свой класс? Например, когда банковский счет клиента становится пустым, он существенно меняет свое поведение. В частности, отклоняются некоторые операции, такие как «снять со счета» и «закрыть счет».

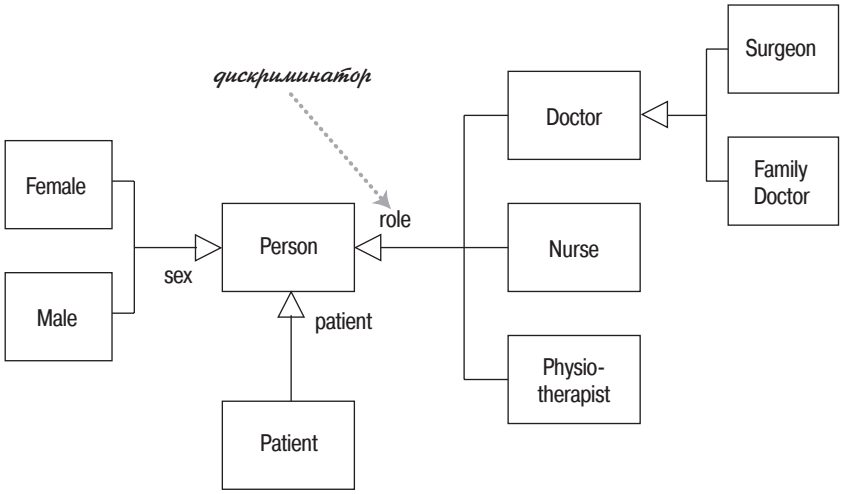

Рис. 5.11. *Множественная классификация*

Динамическая классификация (dynamic classification) разрешает объектам изменять свой тип в рамках структуры подтипов, а **статическая классификация** (static classification) этого не допускает. Статическая классификация проводит границу между типами и состояниями, а динамическая классификация объединяет эти понятия.

Следует ли использовать множественную динамическую классификацию? Я полагаю, что она полезна для концептуального моделирования. Однако с точки зрения программного обеспечения на пути ее реализации слишком много препятствий. В подавляющем большинстве диаграмм UML вы встретите только однозначную статическую классификацию, поэтому она должна стать вашим обычным инструментом.

Класс-ассоциация

Классы-ассоциации (association classes) позволяют дополнительно определять для ассоциаций атрибуты, операции и другие свойства, как показано на рис. 5.12. Из данной диаграммы видно, что `Person` (Личность) может принимать участие в нескольких совещаниях (`Meeting`). При этом необходимо каким-то образом хранить информацию о том, насколько внимательной была данная личность; это можно сделать, добавив к ассоциации атрибут `attentiveness` (внимательность).

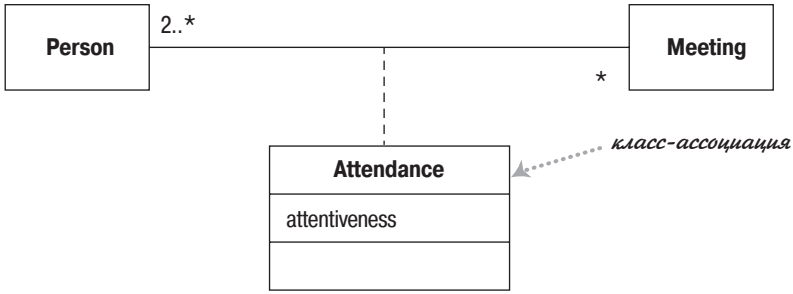

Рис. 5.12. *Класс-ассоциация*

На рис. 5.13 показан другой способ представления данной информации: образование самостоятельного класса `Attendance` (Присутствие). Обратите внимание, как при этом изменили свои значения кратности.

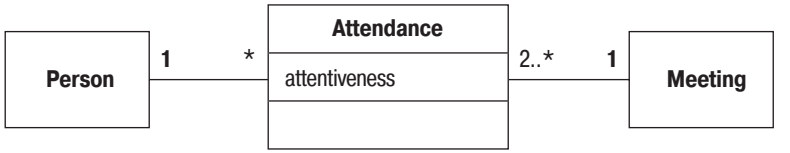

Рис. 5.13. *Развитие класса-ассоциации до обычного класса*

Какие же преимущества может дать класс-ассоциация в качестве компенсации за необходимость помнить еще один вариант уже описанной нотации? Класс-ассоциация дает возможность определить дополнительное ограничение, согласно которому двум участвующим в ассоциации объектам может соответствовать только один экземпляр класса-ассоциации. Мне кажется, необходимо привести еще один пример.

Посмотрим на две диаграммы, изображенные на рис. 5.14. Форма этих диаграмм практически одинакова. Хотя можно себе представить компанию (`Company`), играющую различные роли (`Role`) в одном и том же контракте (`Contract`), но трудно вообразить личность (`Person`), имеющую различные уровни компетенции в одном и том же навыке (`Skill`); действительно, скорее всего, это можно считать ошибкой.

В языке UML допустим только последний вариант. Может существовать только один уровень компетенции для каждой комбинации личности и навыка. Верхняя диаграмма на рис. 5.14 не допускает участия компании более чем в одной роли в одном и том же контракте. Если без этого не обойтись, надо превратить `Role` в полный класс, как это сделано на рис. 5.13.

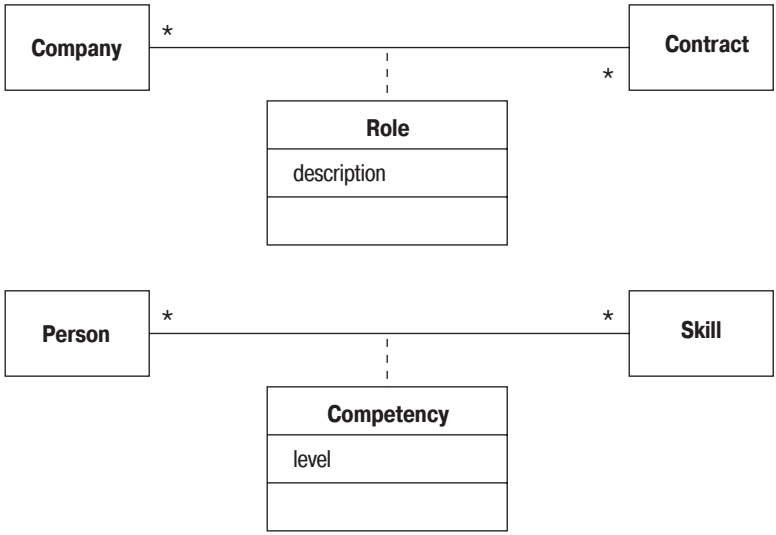

Рис. 5.14. *Хитрости класса-ассоциации (класс Role, возможно, не должен быть классом-ассоциацией)*

Реализация классов-ассоциаций не слишком очевидна. Мой совет: реализовывать класс-ассоциацию так, как будто это обычный класс, но методы, предоставляющие информацию связанным классам, должны принадлежать классу-ассоциации. Поэтому для случая, изображенного на рис. 5.12, я бы представил следующие методы класса `Person`:

```
class Person
List getAttendances()
List getMeetings()
```

Таким образом, клиенты объекта Person могут обнаружить сотрудников на совещании; если им требуются детали, то они могут получить собственно часы работы (Attendance). Если вы так делаете, не забудьте об ограничении, при котором для любой пары объектов Person (Личность) и Meeting (Совещание) может существовать только один объект Attendance (Присутствие).

Часто этот вид конструкции можно встретить там, где речь идет о временных изменениях (см., например, рис. 5.15). Однако я считаю, что создание дополнительных классов или классов-ассоциаций может сделать модель сложной для понимания, а также направить реализацию в неправильное русло.

Рис. 5.15. *Использование класса для временного отношения*

Если я встречаю временную информацию такого типа, то использую для ассоциации ключевое слово «temporal» (временной) (рис. 5.16). Модель означает, что некоторое время личность может работать только в одной компании. Однако по прошествии времени личность сможет работать в нескольких компаниях. Это предполагает интерфейс, описываемый следующими строками:

```
class Person ...
Company getEmployer();         // определение текущего работодателя
Company getEmployer(Date);     // определение работодателя на указанный момент
void changeEmployer(Company newEmployer,Date changeDate);
void leaveEmployer (Date changeDate);
```

Ключевое слово «temporal» не входит в состав языка UML, но я упомянул его здесь по двум причинам. Во-первых, это понятие часто оказывалось полезным для меня как проектировщика. Во-вторых, это демонстрирует, как можно применять ключевые слова для расширения языка UML. Дополнительную информацию по данному вопросу можно найти на *http://martinfowler.com/ap2/timeNarrative.html*.

Рис. 5.16. *Ключевое слово «temporal» для ассоциаций*

Шаблон класса (параметризованный класс)

Некоторые языки, в особенности C++, включают в себя понятие **параметризованного класса** (parameterized class) или **шаблона** (template). (Шаблоны могут быть включены в языки Java и C# в ближайшем будущем.)

Наиболее очевидная польза от применения этого понятия проявляется при работе с коллекциями в строго типизированных языках. Таким образом, в общем случае поведение для множества можно определить с помощью шаблона класса `Set` (Множество).

```
class Set <T> {
void insert (T newElement);
void remove (T anElement);
```

После этого можно использовать данное общее определение для задания более конкретных классов-множеств:

```
Set <Employee> employeeSet;
```

Для объявления класса-шаблона в языке UML используется нотация, показанная на рис. 5.17. Прямоугольник с буквой «T» на диаграмме служит для указания параметра типа. (Можно указать более одного параметра.)

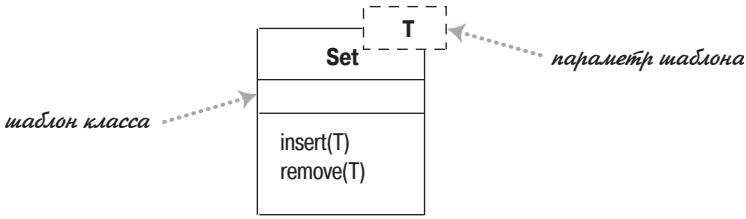

Рис. 5.17. *Класс-шаблон*

Применение параметризованного класса, такого как `Set<Employee>`, называется **образованием производных** (derivation). Образование производных можно изобразить двумя способами. Первый способ отражает синтаксис языка C++ (рис. 5.18). Выражение образования производных описывается в угловых скобках в виде `<parameter-name::parameter-value>`. Если указывается только один параметр, то обычно имя параметра опускают. Альтернативная нотация (рис. 5.19) усиливает связь с шаблоном и допускает переименование связанного элемента.

Рис. 5.18. *Связанный элемент (вариант 1)*

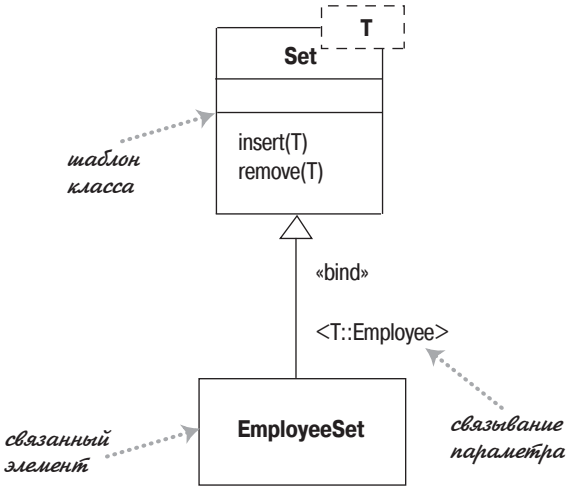

Рис. 5.19. *Связанный элемент (вариант 2)*

Ключевое слово «bind» (связать) является стереотипом отношения уточнения. Это отношение означает, что класс EmployeeSet (Множество служащих) будет согласован с интерфейсом класса Set. Можете считать EmployeeSet подтипом типа Set. Это соответствует другому способу реализации типизированных коллекций, который заключается в объявлении всех соответствующих подтипов.

Однако использование образования производных не эквивалентно определению подтипа. В связанный элемент нельзя добавлять новые возможности – он полностью определен своим шаблоном; можно только добавить информацию, ограничивающую его тип. Если же вы хотите добавить возможности, то должны создать некоторый подтип.

Перечисления

Перечисления (рис. 5.20) используются для представления фиксированного набора значений, у которых нет других свойств кроме их символических значений. Они изображаются в виде класса с ключевым словом «enumeration».

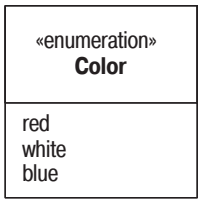

Рис. 5.20. *Перечисление*

Активный класс

Активный класс (active class) имеет экземпляры, каждый из которых выполняет и управляет собственным потоком управления. Вызовы методов могут выполняться в клиентском потоке или в потоке активного объекта. Удачным примером может служить командный процессор, который принимает извне командные объекты, а затем исполняет команды в контексте собственного потока управления.

Как видно из рис. 5.21, при переходе от UML 1 к UML 2 нотация активных классов изменилась. В UML 2 активный класс обозначен дополнительными вертикальными линиями по краям; в UML 1 он имел толстую границу и назывался активным объектом.

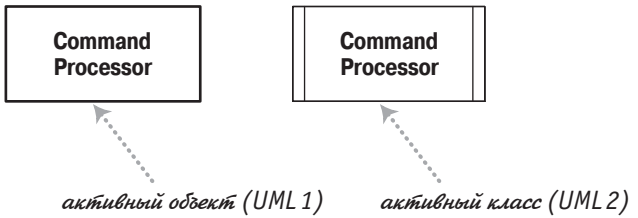

***Рис. 5.21.** Активный класс*

Видимость

Видимость (visibility) – это понятие простое по существу, но содержащее сложные нюансы. Идея заключается в том, что у любого класса имеются открытые (public) и закрытые (private) элементы. Открытые элементы могут быть использованы любым другим классом, а закрытые элементы – только классом-владельцем. Однако в каждом языке действуют свои правила. Несмотря на то что во многих языках употребляются такие термины, как *public* (открытый), *private* (закрытый) и *protected* (защищенный), в разных языках они имеют различные значения. Эти различия невелики, но они приводят к недоразумениям, особенно у тех, кто использует в своей работе более одного языка программирования.

В языке UML делается попытка решить эту задачу, не устраивая жуткую путаницу. По существу, в рамках UML для любого атрибута или операции можно указать индикатор видимости. Для этой цели можно использовать любую подходящую метку, смысл которой определяется тем или иным языком программирования. Тем не менее в UML предлагается четыре аббревиатуры для обозначения видимости: + (public – открытый), - (private – закрытый), ~ (package – пакетный) и # (protected – защищенный). Эти четыре уровня определены и используются в рамках метамодели языка UML, но их определения очень незначительно отличаются от соответствующих определений в других языках программирования.

При использовании видимости применяйте правила того языка программирования, на котором вы работаете. Рассматривая модель в UML с какой-нибудь точки зрения, аккуратно расшифровывайте значения маркеров видимости и старайтесь понять, как эти значения могут измениться при переходе от одного языка программирования к другому.

В подавляющем большинстве случаев я не рисую на диаграммах маркеры видимости; я их использую, только если необходимо подчеркнуть различия в видимости определенных свойств. И даже в таких случаях я могу, как правило, обойтись без + и -, которые, по крайней мере, достаточно легко запомнить.

Сообщения

В стандартном языке UML не отображается информация о сообщениях на диаграммах классов. Однако иногда я встречал диаграммы, подобные той, которая приведена на рис. 5.22.

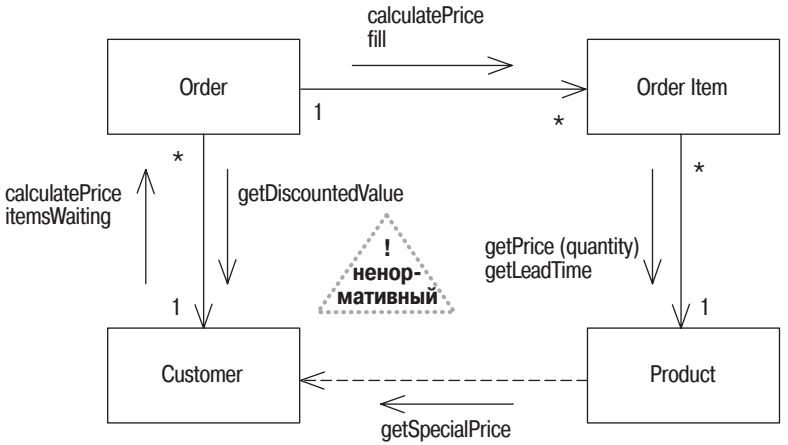

Рис. 5.22. *Классы с сообщениями*

При этом добавляются стрелки со стороны ассоциаций. Стрелки помечаются сообщениями, которые один объект посылает другому. Поскольку для посылки сообщения классу наличие ассоциации с ним не обязательно, то может потребоваться дополнительная стрелка зависимости, чтобы отобразить сообщения между классами, не связанными ассоциацией.

Информация о сообщениях охватывает несколько прецедентов, поэтому, в отличие от коммуникационных диаграмм, они не нумеруются, чтобы показать их последовательность.

6

Диаграммы объектов

Диаграмма объектов (object diagram) – это снимок объектов системы в какой-то момент времени. Поскольку она показывает экземпляры, а не классы, то диаграмму объектов часто называют диаграммой экземпляров.

Диаграмму объектов можно использовать для отображения одного из вариантов конфигурации объектов. (На рис. 6.1 показано множество классов, а на рис. 6.2 представлено множество связанных объектов.) Последний вариант очень полезен, когда допустимые связи между объектами могут быть сложными.

Можно определить, что элементы, показанные на рис. 6.2, являются экземплярами, поскольку их имена подчеркнуты. Каждое имя представляется в виде: `имя экземпляра : имя класса`. Обе части имени не являются обязательными, поэтому имена `John`, `:Person` и `aPerson` являются допустимыми. Если указано только имя класса, то необходимо поставить двоеточие. Можно также задать значения и атрибуты, как показано на рис. 6.2.

Строго говоря, элементы диаграммы объектов – это спецификации экземпляров, а не сами экземпляры. Причина в том, что разрешается оставлять обязательные атрибуты пустыми или показывать спецификации экземпляров абстрактных классов. Можно рассматривать **спе-**

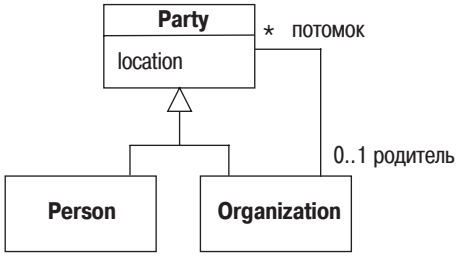

Рис. 6.1. *Диаграмма классов, показывающая структуру класса Party (вечеринка)*

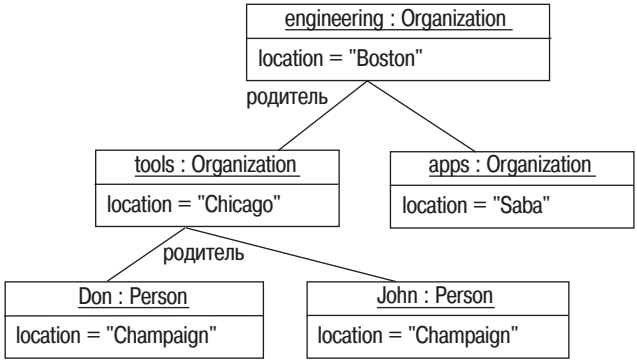

Рис. 6.2. *Диаграмма объектов с примером экземпляра класса Party*

цификации экземпляров (instance specifications**)** как частично определенные экземпляры.

С другой стороны, диаграмму объектов можно считать коммуникационной диаграммой **(***стр. 152***)** без сообщений.

Когда применяются диаграммы объектов

Диаграммы объектов удобны для показа примеров связанных друг с другом объектов. Во многих ситуациях точную структуру можно определить с помощью диаграммы классов, но при этом структура остается трудной для понимания. В таких случаях пара примеров диаграммы объектов может прояснить ситуацию.