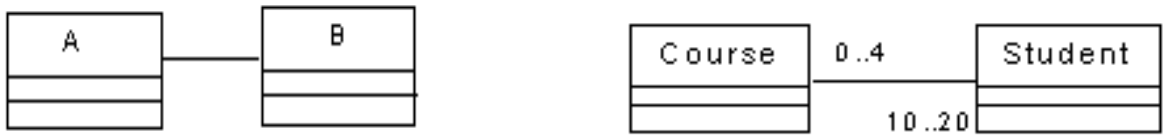
2.5.6. Связи

2.5.6.1. Типы связей

Между классами могут быть установлены следующие типы связей: ассоциация, зависимость, агрегация и обобщение. Рассмотрим каждую из них подробнее.

1. ***Ассоциация*** - это семантическая связь между классами (то есть можно указать только роли, которые классы играют друг для друга). Ассоциация дает классу возможность узнавать об общих атрибутах и операциях другого класса. Ассоциацию часто используют на ранней стадии анализа, чтобы зафиксировать участников отношения. В дальнейшем ассоциации обычно уточняются, становясь более специализированной связью.



а) б)

Рис. 2.21. Графическая нотация для изображения:
а) ассоциации; б) множественности связи

После того, как классы связали ассоциацией, они могут передавать друг другу сообщения на диаграммах Взаимодействия. Ассоциации могут быть двунаправленными или однонаправленными. На языке UML двунаправленные ассоциации изображают в виде простой линии без стрелок (рис. 2.21, а) или со стрелками с обеих сторон. На однонаправленной ассоциации ставят только одну стрелку, показывающую направление связи.

Множественность (multiplicity) показывает, сколько экземпляров одного класса взаимодействуют с одним экземпляром другого класса в данный момент времени. Приведем пример. При разработке системы регистрации курсов в университете можно определить классы Course (Курс) и Student (Студент). Вопросы, на которые должен ответить параметр множественности: «Сколько курсов студент может посещать в данный момент?» и «Сколько студентов могут посещать один курс?». Индикаторы множественности устанавливаются на обоих концах линии связи. В нашем примере можно решить, что одному студенту разрешается посещать от 0 до 4 курсов, а один курс могут слушать от 10 до 20 студентов (рис. 2.21, б).

Для того чтобы уменьшить область действия ассоциации применяются ***квалификаторы*** (qualifiers). Допустим, что между классами Person и Company установлена связь ассоциации и что для данного значения атрибута Person ID (Идентификационный номер) существуют две взаимодей-

ствующие с классом компании. Это можно показать на диаграмме с помощью квалификаторов (рис. 2.22).

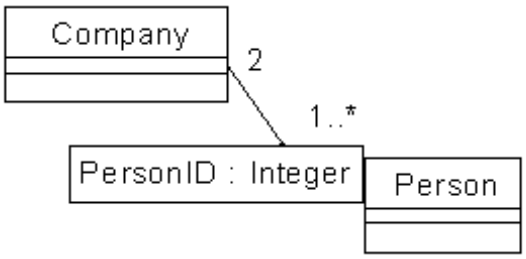


Рис. 2.22. Использование квалификаторов

2. ***Связь зависимости*** (использования) показывает, что один класс (клиент) ссылается на другой (сервер). Типичный случай проявления такого отношения - когда в реализации операции происходит объявление локального объекта используемого класса. Строгое отношение зависимости ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера. Зависимости всегда однонаправлены. Их изображают в виде стрелки, проведенной пунктирной линией (рис. 2.23).

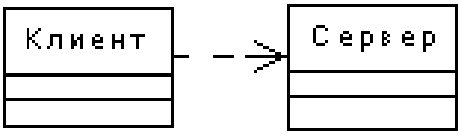


Рис. 2.23. Связь зависимости

3. ***Агрегация*** (aggregations) представляет собой более тесную форму ассоциации. Агрегация - это связь между целым и его частями. Например, такую связь можно установить между классом Автомобиль и классами для отдельных частей автомобиля, такими как Двигатель, Покрышки и т.п. Агрегацию изображают в виде линии с ромбиком у класса, являющегося целым (рис. 2.24, а).

а) б)

Рис. 2.24. Графическая нотация для изображения:
а) простой агрегации; б) рефлексивной агрегации

Как и ассоциации, агрегации могут быть рефлексивными (рис.2.24,б). Рефлексивные агрегации предполагают, что один экземпляр класса состоит из одного или нескольких экземпляров того же класса. Например, комбинируя ингредиенты при приготовлении пищи, можно получить ингредиенты для других блюд. Иначе говоря, каждый ингредиент состоит из других ингредиентов.

Один и тот же класс может быть агрегирован несколькими классами. Например, одна и та же книга в библиотеке может входить сразу в несколько каталогов.

Один класс может участвовать в нескольких отношениях агрегации с другими классами. Например, такую связь образует класс Car (Автомобиль) с классами для отдельных частей автомобиля, таких как Door (Дверь), Tire (Покрышка) и т.п. Автомобиль может иметь 4 двери, 4 покрышки и т.д. (рис. 2.25).

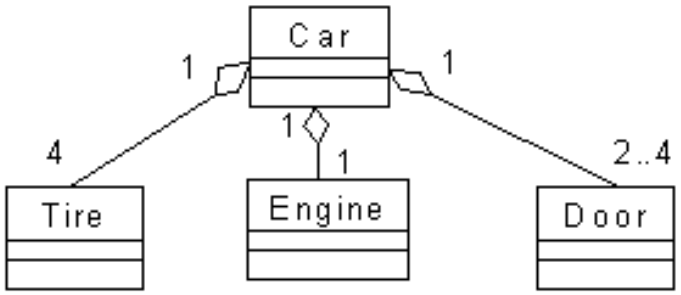


Рис. 2.25. Пример множественной агрегации

Пример агрегирования присутствует в простейшей файловой системе: каталоги могут содержать подкаталоги и файлы (рис. 2.26).

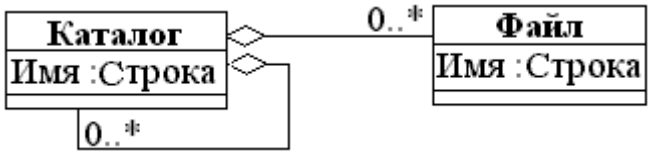
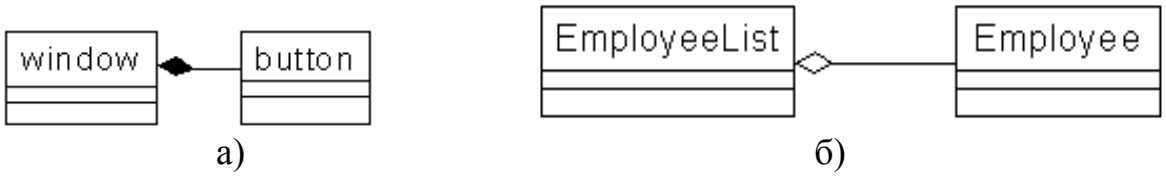


Рис. 2.26. Пример агрегирования

Если два класса связаны отношением агрегации, то в класс-целое войдут атрибуты для каждого класса-части. Можно устанавливать каким образом атрибуты будут включаться (метод включения): по значению или по ссылке. ***Агрегация по значению*** (By Value) предполагает, что целое и часть создаются и разрушаются одновременно. Например, если между классами Window (Окно) и Button (Кнопка) установлена агрегация по значению, соответствующие объекты создаются и разрушаются в одно и то же время. На языке UML агрегацию по значению помечают закрашенным

ромбиком (рис. 2.27, а). ***Агрегация по ссылке*** (By Reference) предполагает, что целое и часть могут создаваться и разрушаться в разное время. Если между классом EmployeeList (Список сотрудников) и Employee (Сотрудник) установлена агрегация по ссылке, это означает, что они могут создаваться и разрушаться в памяти независимо друг от друга, то есть в разное время. Агрегация по ссылке изображается в виде пустого ромбика (рис.2.27, б).



а) б)

Рис. 2.27. Графическая нотация для изображения:
а) агрегации по значению; б) агрегации по ссылке

Агрегация не требует обязательного физического включения ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть косвенной. Например, объект класса Shareholder (Акционер) может содержать ключ записи об этом акционере в базе данных акций. Это тоже агрегация, однако, без физического включения.

«Лакмусовая бумажка» для выявления агрегации такова: если налицо отношение «целое/часть» между объектами, их классы должны находиться друг с другом в отношении агрегации. Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Если Вы не уверены, что налицо отношение общего/частного («is-a»), лучше вместо наследования применить агрегацию или что-нибудь еще.

4. С помощью ***обобщений*** (generalization) показывают связи наследования между двумя классами. ***Наследование*** - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование). Наследование позволяет одному классу наследовать все атрибуты, операции и связи другого. Класс, структура и поведение которого наследуются, называется ***суперклассом*** (предком). Производный от суперкласса класс называется ***подклассом*** (потомком). Наследование устанавливает между классами иерархию «общего/частного». В подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Подкласс обычно расширяет или ограничивает существующую структуру

и поведение своего суперкласса. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.

Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. Хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем.

Классы, экземпляры которых не создаются, называются ***абстрактными***. Ожидается, что подклассы абстрактных классов (такие классы называют конкретными классами или листьями иерархического дерева) доопределят их до жизнеспособной абстракции, наполняя класс содержанием. В C++ существует возможность объявлять функции виртуальными. Метод объявленный виртуальным может быть в подклассе переопределен, а остальные - нет. Если виртуальные методы не переопределены, экземпляр такого класса невозможно создать.

Большинство объектно-ориентированных языков непосредственно поддерживает концепцию наследования.

Наследования изображают в виде стрелки от класса-потомка к классу-предку (рис. 2.28).

Рис. 2.28. Связь обобщения

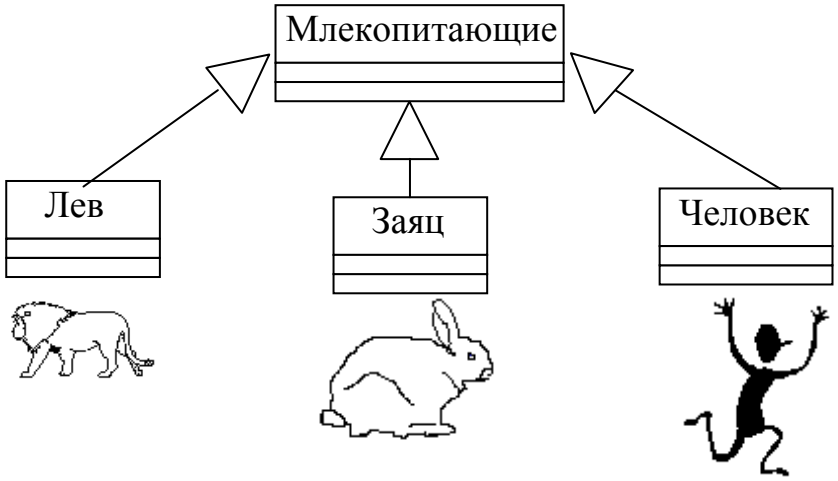


Рис. 2.29. Наследование: класс Млекопитающие

Примеры наследования можно обнаружить в природе. Существуют сотни различных типов млекопитающих: люди, львы, зайцы и т.д. У каждого из них имеются уникальные особенности, но есть и общие характеристики, такие, как принадлежность к теплокровным и воспитание потомства. Класс Млекопитающие имеет общие характеристики. Он порождает подклассы: Человек, Лев, Заяц и другие (рис.2.29). Подкласс наследует характеристики класса Млекопитающие, но в то же время имеет и свои собственные. Когда происходит событие, влияющее на всех млекопитающих, изменение надо внести только в родительский объект, а все потомки автоматически его наследуют. Если млекопитающие, например, внезапно станут холоднокровными, достаточно будет изменить класс Млекопитающие, подклассы Человек, Лев, Заяц и все другие автоматически наследуют новую характеристику.

В банковской системе наследование можно применять для работы с различными типами счетов. Допустим, банк обслуживает четыре типа счетов: до востребования (checking), сберегательный (savings), кредитный (credit) и депозитный сертификат. Все счета имеют следующие сходные характеристики: номер счёта, ставка процента и владелец. Можно создать родительский класс Account (Счёт) с общими характеристиками всех счетов. Подклассы наследуют характеристики суперкласса и имеют свои собственные уникальные характеристики. Например, кредитный счёт содержит лимит кредита и размер минимального взноса, а депозитный сертификат - срок платежа. Изменения в родительском классе повлияют на всех потомков, но потомки не влияют друг на друга и на своего предка.

Одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющийся код. Например, нельзя унаследовать графический элемент, который был бы одновременно окружностью и картинкой; приходится наследовать что-то одно и добавлять необходимое от второго. Механизм ***множественного наследования*** позволяет разрешить эти проблемы.

Если в структуре классов конечные классы (листья) могут быть сгруппированы во множества по разным ортогональным признакам, и эти множества перекрываются, тогда невозможно обойтись одной структурой наследования. В примере, описанном ниже, такими признаками являются способность приносить дивиденды и возможность страховки. В этой ситуации, чтобы соединить два нужных поведения, используют множественное наследование.

Рассмотрим пример множественного наследования. Допустим, что нужно организовать учет различных видов материального и нематериального имущества: банковские счета, недвижимость, акции и облигации. Банковские счета бывают текущие и сберегательные. Акции и облигации

можно отнести к ценным бумагам, управление ими отлично от банковских счетов. Но и счета и ценные бумаги - это разновидности имущества. Существуют другие классификации тех же видов имущества. Общим свойством банковских счетов и ценных бумаг является способность приносить дивиденды. Другим общим свойством является то, что их (недвижимость и сберегательные вклады) можно застраховать.

Одиночное наследование в данном случае не отражает реальности, поэтому будет использоваться множественное. Получившаяся структура классов показана на рис. 2.30. На нем класс Security (ценные бумаги) наследует одновременно от классов InterestBearingItem (источник дивидендов) и Asset (имущество). Аналогично, BankAccount (банковский счет) наследует сразу от трех классов: InsurableItem (страхуемое), Asset и InterestBearingItem.

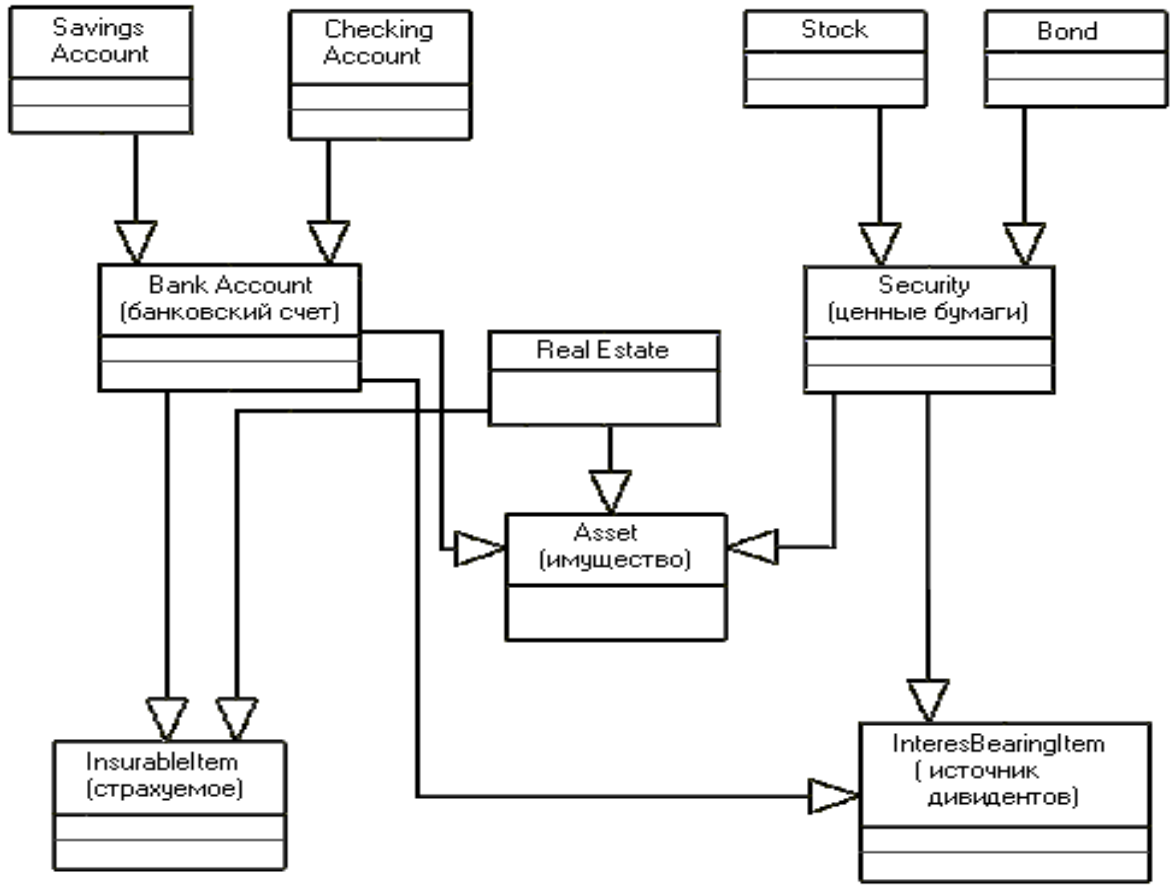


Рис. 2.30. Пример множественного наследования

Запишем, как это выражается на C++. Сначала базовые классы:

```
class Asset ...
class InsurableItem ...
class InterestBearingItem ...
```

Промежуточные классы (наследуют от нескольких суперклассов):

```
class BankAccount: public Asset,
                   public InsurableItem,
                   public InterestBearingItem ...
class RealEstate:  public Asset,
                   public InsurableItem ...
class Security:    public Asset,
                   public InterestBearingItem ...
// Листья:
class SavingsAccount: public BankAccount ...
class CheckingAccount: public BankAccount ...
class Stock: public Security ...
class Bond: public Security ...
```

Проектирование структур классов с множественным наследованием осуществляется путем последовательных приближений. Есть две специфические для множественного наследования проблемы: как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Допустим, классы Asset и InsurableItem содержат атрибут presentValue (текущая стоимость). Класс RealEstate потомок обоих классов, получается, что он наследует две операции с одним и тем же именем. Конфликт имен может ввести двусмысленность в поведение класса с несколькими предками. Для устранения этого к именам элементов добавляют префиксы, указывающие на имена классов-источников.

Вторая проблема множественного наследования возникает, когда один класс является наследником другого по нескольким линиям. Например, класс D наследует от B и C, которые, в свою очередь, наследуют от A. Эта ситуация называется ***повторным наследованием***. Рассмотрим следующий класс, который дважды наследует от класса Security:

```
class MutualFund: public Stock, public Bond ...
```

Проблема повторного наследования решается двумя способами. Во-первых, можно разделить две копии унаследованного элемента, добавив к именам префиксы в виде имени класса-источника. Во-вторых, можно рассматривать множественные ссылки на один и тот же класс как обозначающие один и тот же класс. Так поступают в C++, где повторяющийся суперкласс определяется как виртуальный базовый класс. Виртуальный базовый класс появляется, когда какой-либо подкласс именует другой класс своим суперклассом и отмечает этот суперкласс как виртуальный, чтобы показать, что это общий (shared) класс.

При множественном наследовании часто используют ***классы-примеси*** (mixin). В этом случае комбинируют (смешивают) небольшие классы, чтобы построить классы с более сложным поведением. Примесь синтаксически ничем не отличается от класса, но назначение их различно. Примесь не предназначена для порождения самостоятельно используемых экземпляров, она смешивается с другими классами. На рис. 2.30 классы InsurableItem и InterestBearingItem - это классы-примеси. Ни один из них не может существовать сам по себе, они используются для придания смысла другим классам. Таким образом, примесь - это класс, выражающий не поведение, а одно свойство, которое можно «привить» другим классам через наследование. Это свойство обычно ортогонально собственному поведению наследующего его класса. Классы, сконструированные только из примесей, называют агрегатными.

5. Наследование связано с явлением ***полиморфизма.***

Традиционные типизированные языки типа Pascal основаны на той идее, что функции и процедуры, а, следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом, то есть каждая переменная и каждое значение относятся к одному определенному типу. В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам.

Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Наследование без полиморфизма возможно, но не очень полезно. Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя.

Рассмотрим иерархию, в которой имеется базовый класс и три подкласса с именами Circle (Окружность), Triangle (Треугольник) и Rectangle (Прямоугольник) (рис. 2.31). Для класса Rectangle определен в свою очередь подкласс SolidRectangle (Окрашенный прямоугольник). Предположим, что в классе DisplayItem определена переменная экземпляра theCenter (координаты центра изображения), а также следующие операции: draw (нарисовать изображение), move (передвинуть изображение), location (вернуть координаты изображения). Операция location является общей для всех подклассов и не требует обязательного переопределения. Однако, поскольку только подклассы могут знать, как их изображать и передвигать, операции draw и move должны быть переопределены.

Класс Circle имеет переменную theRadius и, соответственно, операции для установки (set) и чтения значения этой переменной. Для этого

класса операция draw формирует изображение окружности заданного радиуса с центром в точке theCenter. В классе Rectangle есть переменные theHeight и theWidth и соответствующие операции установки и чтения их значений. Операция draw в данном случае формирует изображение прямоугольника заданной высоты и ширины с центром в заданной точке theCenter. Подкласс SolidRectangle наследует все особенности класса Rectangle, но операция draw в этом подклассе переопределена. Сначала вызывается draw вышестоящего класса, а затем полученный контур заполняется цветом.

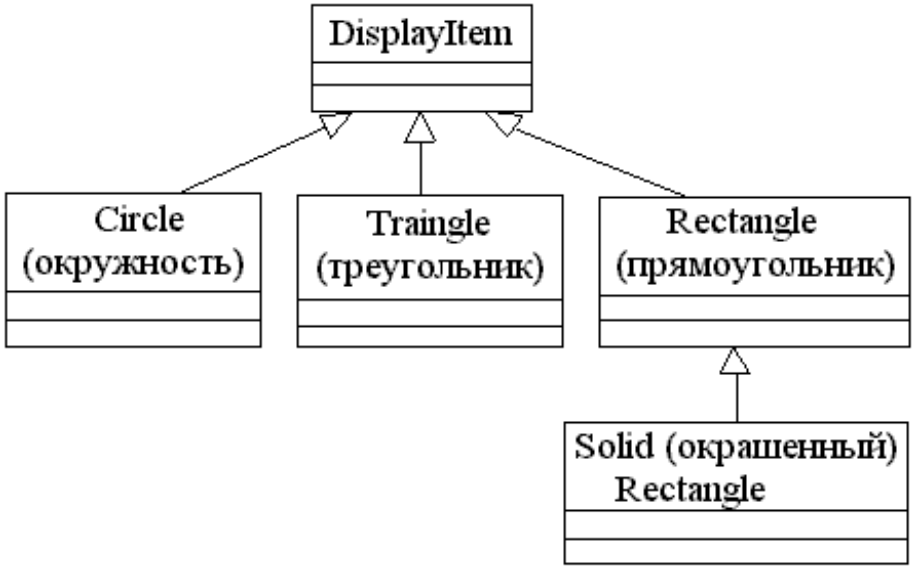


Рис. 2.31. Диаграмма класса DisplayItem

Рассмотрим следующий фрагмент программы:

```
DisplayItem* items[10]; ...
for (unsigned index=0;index<10;index++) items[index]->draw();
```

Вызов draw требует полиморфного поведения. Существует массив объектов, содержащий указатели на любые разновидности DisplayItem. Если нужно, чтобы все фигуры изобразили себя на экране, необходимо перебрать элементы массива и послать каждому указываемому объекту сообщение draw. Компилятор не может определить, какую функцию и откуда нужно при этом вызвать, так как невозможно предсказать, на что будут указывать элементы массива во время выполнения программы.

В C++ операции для позднего связывания объявляются виртуальными (virtual), а все остальные обрабатываются компилятором как обычные вызовы подпрограмм. В нашем примере draw - виртуальная функция, а location - обычная. Есть еще одно средство, используя которое можно выиграть в скорости. Невиртуальные методы могут быть объявлены подстав-

ляемыми (inline), при этом соответствующая подпрограмма не вызывается, а явно включается в код на манер макроподстановки.

Для управления виртуальными функциями в C++ используется концепция vtable (виртуальных таблиц), которые формируются для каждого объекта при его создании (то есть когда класс объекта уже известен). Такая таблица содержит список указателей на виртуальные функции. Например, при создании объекта класса Rectangle виртуальная таблица будет содержать запись для виртуальной функции draw, содержащую указатель на ближайшую в иерархии реализацию функции draw. Если в классе DisplayItem есть виртуальная функция rotate, которая в классе Rectangle не переопределена, то соответствующий указатель для rotate останется связан с классом DisplayItem. Во время исполнения программы происходит косвенное обращение через соответствующий указатель и сразу выполняется нужный код без всякого поиска.

Операция draw в классе SolidRectangle представляет собой особый случай. Чтобы вызвать метод draw из суперкласса, применяется специальный префикс, указывающий на место определения функции. Это выглядит следующим образом:

```
Rectangle::Draw();
```

Исследование Страуструпа показало, что вызов виртуальной функции по эффективности мало уступает вызову обычной функции [3]. Для одиночного наследования вызов виртуальной функции требует дополнительно выполнения трех-четырех операций доступа к памяти по отношению к обычному вызову; при множественном наследовании число таких дополнительных операций составляет пять или шесть.

Рассмотрим возможность ***множественного полиморфизма***. Вернемся к одной из функций-членов класса DisplayItem:

```
virtual void draw();
```

Эта операция изображает объект на экране в некотором контексте. Она объявлена виртуальной, то есть полиморфной, переопределяемой подклассами. Когда эту операцию вызывают для какого-то объекта, программа определяет, что, собственно, выполнять. Это одиночный полиморфизм в том плане, что смысл сообщения зависит только от одного параметра, а именно, объекта, для которого вызывается операция.

На самом деле операция draw должна бы зависеть от характеристик используемой системы отображения, в частности от графического режима. Например, в одном случае мы хотим получить изображение с высоким разрешением, а в другом - быстро получить черновое изображение. Можно ввести две различных операции, скажем, drawGraphic и drawText, но это не совсем то, что хотелось бы. Дело в том, что каждый раз, когда требуется

учесть новый вид устройства, его надо проводить по всей иерархии надклассов для класса DisplayItem.

В некоторых языках есть так называемые мультиметоды. Они полиморфны, то есть их смысл зависит от множества параметров (например, от графического режима и от объекта). В C++ мультиметодов нет, поэтому там используется идиома, так называемой, двойной диспетчеризации. Например, мы могли бы вести иерархию устройств отображения информации от базового класса DisplayDevice, а затем определить метод класса DisplayItem так:

```
virtual void draw(DisplayDevice&);
```

При реализации этого метода мы вызываем графические операции, которые полиморфны относительно переданного параметра типа DisplayItem. Таким образом, происходит двойная диспетчеризация: draw сначала демонстрирует полиморфное поведение в зависимости от того, к какому подклассу класса DisplayItem принадлежит объект, а затем полиморфизм проявляется в зависимости от того, к какому подклассу класса DisplayDevice принадлежит аргумент. Эту идиому можно продолжить до множественной диспетчеризации.

2.5.6.2. Дружественные связи

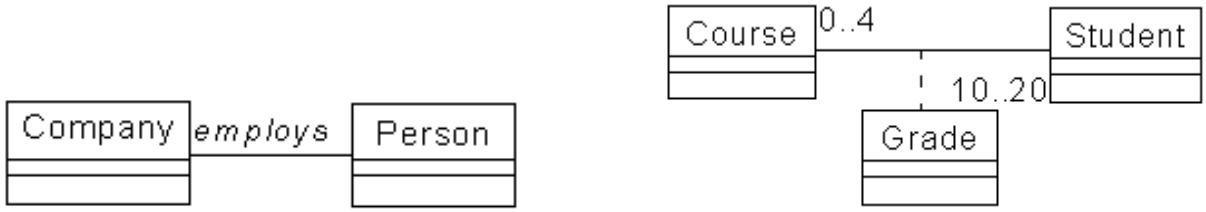
Дружественная (friend) связь предполагает, что класс-клиент имеет право доступа к атрибутам и операциям класса-сервера, которые не являются общими. Это свойство можно задать для ассоциаций, агрегаций, зависимостей и обобщений. В исходный код класса-сервера войдет логика, поддерживающая дружественную видимость для клиента. Допустим, что имеется двунаправленная ассоциация, связывающая классы Person и Company и между ними установлена дружественная связь. Тогда при генерации кода класс Person получит право доступа к частям класса Company, не являющимся общими.

2.5.6.3. Стереотипы, имена и элементы связей

Как и другим элементам модели, связям разрешается назначать ***стереотипы***. Они применяются для классификации связей. Например, Вы используете два типа ассоциаций, им можно задать стереотипы. Стереотипы пишут вдоль линии ассоциации в двойных угловых скобках (<< >>).

С помощью имен связей (ролевых имен) уточняют связи. ***Имя связи*** - это обычно глагол (глагольная фраза), описывающая, зачем нужна связь.

Например, между классом Person (Человек) и классом Company (Компания) существует ассоциация. Можно задать вопрос: «Зачем нужна эта связь? Является ли объект класса Person клиентом компании, ее сотрудником или владельцем?» Для прояснения ситуации можно назвать ассоциацию «employs» (нанимает). Имя показывают около линии соответствующей связи (рис. 2.32, а). Имена у связей определять необязательно. Обычно это делают, если причина создания связи неочевидна.



а) б)

Рис. 2.32. Графическая нотация для изображения:
а) имени связи; б) элемента связи (класса Ассоциаций)

Элементом связи (link element), известным также как класс Ассоциаций (Association class), называется место, где хранятся относящиеся к ассоциации атрибуты. Допустим, что имеются два класса: Student (Студент) и Course (Курс), и необходимо добавить на диаграмму атрибут Grade (Год обучения). В этом случае возникает вопрос, в какой класс добавить атрибут? Если мы поместим его в класс Student, то придется вводить атрибут для каждого посещаемого студентом курса, что значительно увеличит этот класс. Если же мы поместим его в класс Course, то придется задавать атрибут для каждого посещающего этот курс студента. Для решения этой проблемы можно создать класс Ассоциаций. В него следует поместить атрибут Grade, относящийся в большей степени к связи между классами Курс и Студент, чем к какому-то классу конкретно. Класс Ассоциаций изображается с помощью нотации показанной на рис. 2.32, б.

2.5.6.4. Выявление связей

Почти вся информация о связях находится на диаграммах Взаимодействия. Просмотрев их можно найти ассоциации и зависимости. Изучив имеющиеся классы можно обнаружить агрегации и обобщения. Для обнаружения связей рекомендуется выполнить следующие действия.

1. Начните с изучения диаграмм Взаимодействия. Если класс А посылает сообщение классу В, между этими классами может быть установлена связь. Как правило, обнаруживаемые таким способом связи - это ассоциации или зависимости.

2. Исследуйте классы на предмет наличия связей целое/часть. Любой класс, состоящий из других классов, может участвовать в связях агрегации.

3. Исследуйте классы на предмет связей обобщения. Ищите абстракции, которые бывают нескольких видов. Допустим, класс Employee (Сотрудник) содержит общие для всех сотрудников атрибуты, операции и связи. Однако в компании имеются два типа сотрудников: получающие почасовую оплату и оклад. Это означает, что можно создать классы HourlyEmp и SalariedEmp, наследующие от класса Employee, но содержащие элементы уникальные для сотрудников одного из типов.

4. Связи обобщения также можно обнаружить, исследуя классы, которые имеют много общего. Допустим, имеются классы CheckingAccount (Счет до востребования) и SavingsAccount (Сберегательный счет). Их данные и поведение сходны. Для общих элементов двух классов можно создать третий класс Account (Счет).

Будьте осторожны, работая с классами, у которых слишком много связей. Одной из особенностей хорошо спроектированного приложения является сравнительно небольшое количество связей в системе. Класс, у которого много связей, должен знать о большом числе других классов системы. В результате его трудно будет использовать во второй раз. Кроме того, сложно будет вносить изменения в готовое приложение. Если изменится любой из классов, это может повлиять на данный.

2.6. Качество классов и объектов

Классы и объекты являются ключевыми абстракциями системы. Опыт показывает, что процесс их выявления должен быть итеративным. За исключением самых простых задач, с первого раза не удается окончательно выделить и описать классы. Это влечет за собой дополнительные затраты на перекомпиляцию, согласование и внесение изменений в проект системы. Поэтому важно с самого начала максимально приблизиться к правильным решениям, чтобы сократить число последующих итераций. Для оценки качества классов и объектов используют пять критериев: сцепление, связность, достаточность, полнота, примитивность.

Термин «сцепление» взят из структурного проектирования. ***Сцепление*** является мерой внешней независимости между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения сцепления между отдельными модулями. При объектно-ориентированной разработке, кроме сцепления между модулями, существенную роль играет сцепление между классами и объектами. Существует определенное противоречие между сцеплением и наследованием. С одной стороны, желательно избегать сильного сцепления классов; с другой сто-

роны, механизм наследования, тесно связывающий подклассы с суперклассами, позволяет использовать сходство абстракций.

Понятие связности также заимствовано из структурного проектирования. ***Связность*** является мерой независимости внутренних частей модуля. При объектно-ориентированной разработке связность относится также к отдельным классам или объектам. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Примером может быть класс, объединяющий розы и космические аппараты. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля выполняют одну специфическую функцию. Эта функция может быть представлена набором элементарных функций, однако каждая из них не будет являться самостоятельной с учетом общей выполняемой работы. Так, например, класс Rose (Роза) будет функционально связным, если он описывает только поведение розы и ничего кроме этого.

К идеям сцепления и связности тесно примыкают понятия достаточности, полноты и примитивности.

Под ***достаточностью*** подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Компоненты должны быть полностью готовы к использованию. Допустим, в классе Set (множество) имеется операция удаления элемента из множества, будет ошибкой не включить в него также операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию.

Под ***полнотой*** подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями.

Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося наверх такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование примитивности. ***Примитивными*** являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом Set операция Add (добавление к множеству элемента) примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Примером примитивной операции также может служить операция добавления к множеству произвольного числа элементов (а не обязательно четырех).