返回总目录

# 目　　录

# 第 4 章　静态建模：类图和对象图

用面向对象的方法处理实际问题时，需要建立面向对象的模型。构成面向对象模型的基本元素有类（class）、对象（objects）、类与类之间的关系等等。用面向对象的思想描述问题，能够把复杂的系统简单化、直观化，而且易于用面向对象语言编程实现，还方便日后对系统的维护工作。本章主要讨论类、类图、对象、对象图、类与类之间的关系等基本概念。

## 4.1　类 和 对 象

首先考察一下日常生活中常见的交通工具——小汽车。比如，我们常见到的小汽车有桑塔拉、富康、丰田、夏利等，不管它是什么型号、哪个厂家生产的车，我们使用它的主要目的是以车代步，迅速便捷。因此，任何车一定要有启动、驱动车轮滚动（行驶）和制动的功能。小汽车除了应有基本的控制行驶的功能外，每种车的车身长度、颜色、底盘高低、汽缸容量等方面可能不太一致，汽车厂商可以根据用户的需要选定。于是，就小汽车而言，用户便有了多种选择，买桑塔拉也行，买丰田也行，只要价格用户能够承受（这里简化了汽车的某些因素）。桑塔拉是小汽车中的一种车型，桑塔拉与小汽车的关系我们可以抽象为对象与类的关系，即作为小汽车类产品，它应提供的功能至少有：启动、行驶、制动，对汽车的车身长度、汽缸容量等也有一定的要求。在符合小汽车类应有的功能和特征的前提下，具体生产出来的车型（比如，夏利、丰田等）则是该类的对象。其实，自然界中存在的事物大都具有类与对象的关系，于是，我们可以借用自然界中的类与对象的表示方法，在计算机的软件系统中实现类与对象，从而达到利用对象在计算机系统中表示事务、处理事务的目的。

所谓对象就是可以控制和操作的实体，它可以是一个设备，一个组织或一个商务。类是对象的抽象描述，它包括属性的描述和行为的描述二方面。属性描述类的基本特征（比如，车身的长度、颜色等）；行为描述类具有的功能（比如，汽车有启动、行驶、制动等功能），也就是对该类的对象可以进行哪些操作。就像程序设计语言中整型变量是整数类型的具体化，用户可以对整型变量进行操作（并不是对整数类型操作）一样，对象是类的实例化，所有的操作都是针对对象进行的。

在计算机系统中，我们用类表示系统，并把现实世界中我们能够识别的对象分类表示，这种处理方式称作面向对象。由于面向对象的思想与现实世界中的事物的表示方式相似，所以采用面向对象的思想建造模型会给建模者带来很多好处。

当建模者建造一个商务系统、信息系统或其它系统的时候，如果用于描述模型的一些概念与问题域中的概念一致，那么这个模型就易于理解，易于交流。比如，为保险公司的业务系统建模，那么一定要使用保险业务中的概念，否则，很难将对象之间的业务关系描述清楚。把系统建立在某个商务的基本概念之上的另一个重要原因是：易于维护系统。因为当新的法规、政策、条款等出台后，建模者必须重新建模使之适应新的变化，由于模型

是建立在对象基础上，所以修改模型并不费力，只要把新商务与旧商务之间的不同之处重新设计描述一下即可，其它地方并不需要修改或调整。由此也可以看出，以面向对象方式建造的模型，由于建造在真实世界的基本概念上，与真实世界非常接近，使得该模型易于交流，易于验证（检查功能需求），易于维护。反过来说，在各种不同的建模方法中（比如，面向对象、面向过程、面向数据），若模型需要反映真实世界中问题域的基本概念，那么面向对象的方法是最好的选择。

构建面向对象模型的基础是类、对象和它们之间的关系。可以在任何类型的系统中（比如，嵌入式实时系统、分布系统、商务软件等）应用面向对象技术。在不同的系统中描述的类可以各种各样。比如：在某个商务信息系统中，可以包含的类有：顾客、协议书、发票、债务、资产、报价单；在工程技术系统中，常常会用到一些设备，该系统中则可以有这些类：传感器、显示器、I/O 卡、发动机、按钮、控制类；在类似操作系统的软件系统中，类用来表示软件中的实体，可能有的类是：文件、执行程序、设备、图标、窗口、滚动条等。

## 4.2 类　　图

类图是用类和它们之间的关系描述系统的一种图示，是从静态角度表示系统的，因此类图属于一种静态模型。类图是构建其它图的基础，没有类图，就没有状态图、协作图等其它图，也就无法表示系统的其它各个方面。

类图中允许出现的模型元素只有类和它之间的关系。类用长方形表示，长方形分成上、中、下三个区域，每个区域用不同的名字标识，用以代表类的各个特征，上面的区域内用黑体字标识类的名字，中间的区域内标识类的属性，下面的区域内标识类的操作方法（即行为），这三部分作为一个整体描述某个类，如图 4-1 所示。当类图中存在多个类时，类与类之间的关系可以用表示某种关系的连线（比如直线、虚线等），把它们连接起来，类与类之间的关系将在后文详述。类图的另一种表示是用类的具体对象代替类，这种表示方法是类图的变种，称作对象图（后文详叙）。

图 4-1　类的图示

在类图中，类被图示为一个长方形，而在具体程序实现时，类可以用面向对象语言（比如，C++、JAVA 语言）中的类结构描述，对类结构的描述包括属性的描述和操作的描述。

### 4.2.1　定义类

在进行构造类图描述系统的工作时，首先要定义类，也就是将系统要处理的数据抽象成类的属性，将处理数据的方法抽象为操作。要准确地找到类不是一件容易的事，通常要获得对所解决的问题域很清楚的专家的帮助。对于建模者所定义的类通常要有这样二个特点：一是使用来自问题域的概念，二是类的名字用该类实际代表的涵义命名。

下面列出一些可以帮助建模者定义类的问题：

- 有没有一定要存储或分析的信息？如果存在需要存储，分析或处理的信息，那么

该信息有可能就是一个类。这里讲的信息可以是概念（该概念总在系统中出现）或事件或事务（它发生在某一时刻）。

- 有没有外部系统？如果有，外部系统可以看作类，该类可以是本系统所包含的类，也可以是本系统与之交互的类。
- 有没有模版、类库、组件等？如果手头上有这些东西，它们通常应作为类。模版、类库、组件可以来自原来的工程、或别人赠送或从厂家购买的。
- 系统中有被控制的设备吗？凡是与系统相连的任何设备都要有对应的类。通过这些类控制设备。
- 有无需要表示的组织机构？在计算机系统中表示组织机构通常用类，特别是构建商务模型时用得更多。
- 系统中有哪些角色？这些角色也可以看成类。比如，用户、系统操作员、客户等。

依照上述几条可以帮助建模者找到需要定义的类。需要说明的是，定义类的基础是系统的需求规格说明，通过分析需求说明文档，从中找到需要定义的类。

### 4.2.2 名字、属性和操作

大家知道，类的图形表示为长方形，长方形又分成三个部分，分别用来表示类的名字、属性和操作，下面详细介绍这三部分的具体表示方法和含义。

1．名字

类的名字用黑体字书写在长方形的最上面，给类命名时最好能够反映类所代表的问题域中的概念。比如，表示小汽车类产品，可以直接用“小汽车”作为类的名字。另外，类的名字含义要清楚准确，不能含糊不清。类通常表示为一个名词，既不带前缀，也不带后缀。

2．属性

类的属性放在类名字的下方，用来描述该类的对象所具有的特征。在图 4-2 中，“小汽车”是类的名字，注册号、生产日期、时速、方向是“小汽车”的属性。描述类的特征的属性可能很多，在系统建模时，我们只抽取那些系统中需要使用的特征作为类的属性。换句话说，只关心那些“有用”的特征，通过这些特征就可以识别该类的对象。从系统处理的角度讲，可能被改变值的特征，才作为类的属性。

正如变量有类型一样，属性也是有类型的，属性的类型反映属性的种类。比如，属性的类型可以是整型、实型、布尔型、枚举型等基本类型。除了基本类型外，属性的类型可以是程序设计语言能够提供的任何一种类型，包括类的类型。如图 4-3 所示。

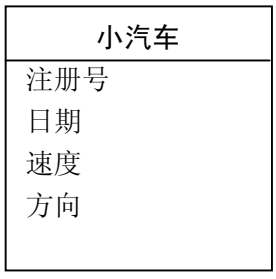



图 4-2　类的名字和属性示例


图 4-3　属性的类型示例

属性有不同的可见性（visibility）。利用可见性可以控制外部事物对类中属性的操作方

式。属性的可见性通常分为三种：公有的（public）、私有的（private）和保护的（protected）。公有属性能够被系统中其它任何操作查看和使用，当然也可以被修改；私有属性仅在类内部可见，只有类内部的操作才能存取该属性，并且该属性也不能被其子类使用；保护属性供类中的操作存取，并且该属性也能被其子类使用。比如，类与类之间可以存在继承关系（用已有的类定义新的类），被继承的类称为父类或基类，从父类中得到父类属性和操作的类，称为子类。一般情况下，有继承关系的父类和子类之间，如果希望父类的所有信息对子类都是公开的，也就是子类可以任意使用父类中的属性和操作，而与其没有继承关系的类不能使用父类中的属性和操作，那么为了达到此目的则必须将父类中的属性和操作定义为保护的；如果并不希望其它类（包括子类）能够存取该类的属性，则应将该类的属性定为私有的；如果对其它类（包括子类）没有任何约束，则可以使用公有的属性。

属性的可见性可以不限于上述的三种，某些具体的程序设计语言还可以定义其它的可见性类型，但是，在表示类图时，必须含有公有类型和私有类型。在类图中，公有类型表示为加号（+），私有类型表示为减号（–），它们标识在属性名称的左侧，如图 4-4 所示。如果属性名称旁没有标识任何符号，表示该属性的可见性尚未定义。注意，这里不存在缺省的可见性。类属性的缺省值可以表示在类图中，如图 4-5 所示。这样，当创建该类的对象时，该对象的属性值便自动被赋以图示中的缺省值。

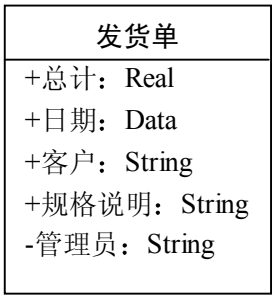



图 4-4　属性的可见性示例

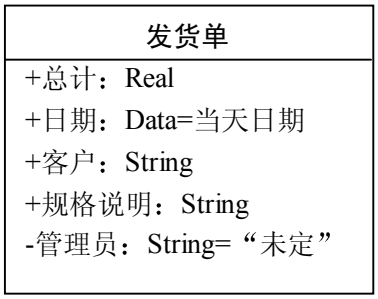



图 4-5　带有属性缺省值的类

类的属性中还可以有一种能被该类的所有对象共享的属性，称之为类的作用域属性（class-scope attribute），也称作类变量（class variable）。类变量在类图中表示为带下划线的形式，如图 4-6 所示。图中“货单个数”属性用来统计总的货单数，在该类的所有对象中，这个属性的值是一致的。

描述属性的语法格式为：

可见性　属性名：类型名 = 初值 {性质串}

其中属性名和类型名是一定要有的，其它部分可根据需要可有可无。属性名和类型名之间用冒号分隔，属性的缺省值用初值表示，类型名与初值之间用等号隔开。最后一个用花括号括起来的性质串，列出该属性所有可能的取值。枚举类型的属性经常使用性质串，性质串中的每个枚举值之间用逗号分隔。如图 4-7 所示。当然，性质串也可以用来说明该属性的其它信息，比如属性的持久性（persistent），关于这一点在第七章中详述。

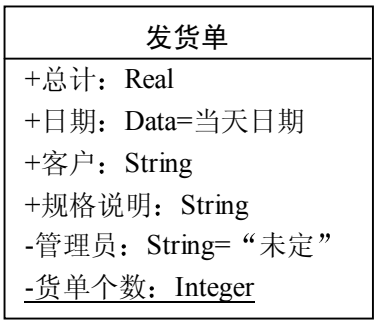



图 4-6　带有类作用域属性的类

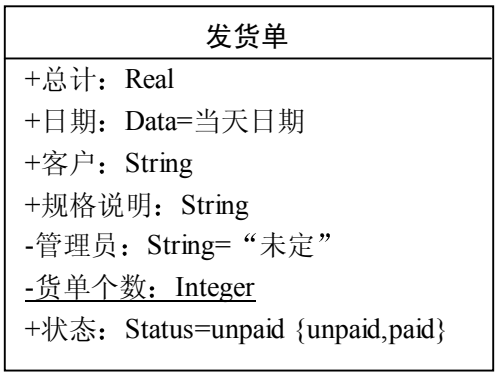



图 4-7　性质串示例

类可以使用面向对象语言的类结构描述平实现，下面以 JAVA 语言为例，描述图 4-6 中类的实现代码：

```
public  class  invoice
{ public double amount;
 public Date data= new Date( );
 public string customer;
 public string specification;
 public string administrator=”unspecified”;
 static private int number_of_invoices=0;

 public invoice()  //构造函数
 {  //部分初始化工作可在此进行
   number_of_invoices++;
 }
 //类的其它操作方法写在这里
 }
```

3．操作

属性仅仅表示了需要处理的数据，对数据的具体处理方法的描述则放在操作部分。存取或改变属性值或执行某个动作都是操作，操作说明了该类能做些什么工作。操作通常又称为函数，它是类的一个组成部分，只能作用于该类的对象上。从这一点也可以看出，类将数据和对数据进行处理的函数封装起来，形成一个完整的整体，这种机制非常符合问题本身的特性。

在类图中，操作部分位于长方形的最底部，一个类可以有多种操作，每种操作由操作名、参数表、返回值类型等几部分构成，标准语法格式为：

可见性 操作名（参数表）：返回值类型{性质串}

其中可见性和操作名是不可缺少的。操作名、参数表和返回值类型合在一起称为操作的标记（signature），操作标记描述了使用该操作的方式，操作标记必须是唯一的。注意，操作只能应用于该类的对象，比如，drive()只能作用于小汽车类的对象，如图 4-8 所示。

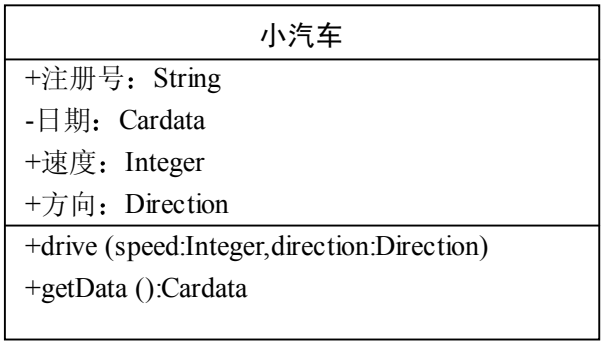



图 4-8　带有属性和操作的类

操作的可见性也分为公有（用加号表示）和私有（用减号表示）两种，其含义等同于属性的公有和私有可见性。

参数表由多个参数（用逗号分开）构成，参数的语法格式为：

参数名：参数类型名= 缺省值

其中省缺值的含义是，如果调用该操作时没有为操作中的参数提供实在参数，那么系统就自动将参数定义式中的缺省值赋给该参数。比如，图 4-9 描述了一个图画类，假设 figure 是图画类的对象，当执行 figure.resize（10，10）时，由于给定了实在参数（10，10），所以

percentX=10，percentY=10；

若执行 figure.resize（37），由于只给定了一个实在参数，按照参数位置、顺序一一对应的原则，percentX 的值被赋与了 37，而 percentY 没有对应的实在参数，故赋以省缺值 25，最后结果为：

percentX=37，percentY=25；

若执行 figure.resize（），由于两个参数都没给出实在参数，所以它们分别获得缺省值 15 和 25，最后结果为：

percentX=15，percentY=25；

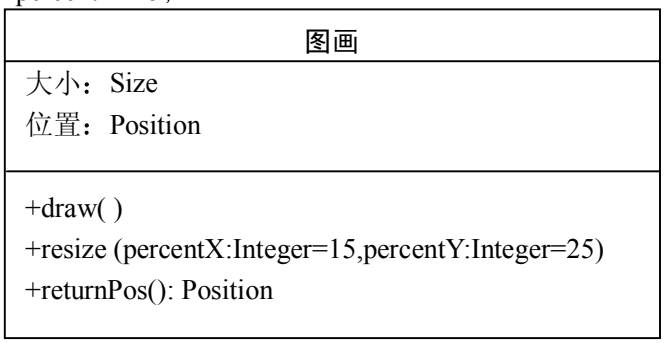



图 4-9　参数的缺省值示例

其实，操作的描述方式有点像程序设计语言（比如 C 语言）的函数定义，所起的作用也差不多。因此，操作可以看作是类的一个接口，通过该接口实现内、外信息的交互。操作的具体实现称作方法，它与实现该操作的算法有关。若算法设计得好，可以减少操作的时空开销，满足求解问题的要求。除了算法以外，一个操作能否执行还与前提条件和输

入参数有关系。所谓前提条件是指操作可以执行的基础，比如，执行放大图形操作的前提条件是被放大的图形一定要存在，否则，放大图形的操作不能执行。操作执行完毕后，被操作对象的形态会受到影响，这种受影响的状况，称为后置条件。比如，执行放大图形操作的结果是原来的图形被放大了。由于图形被放大，随之带来的是表示图形位置的属性的改变，因此，执行操作时，对象属性状态的改变情况也要记录下来。综上所述，要完整地描述一个操作，必须包括：操作的标识（返回值、名字、参数表）、执行操作的前提条件、操作执行完毕后的后置条件、实现操作的算法和对象状态的改变情况。在类图中，关于操作的这些规格说明称为操作的性质，操作的性质不会直接显示在类图中，它可以用文字说明。可以使用 CASE 工具为操作设置超链功能，单击操作便可获得操作的性质。

类也有类作用域操作，图示为带下划线的形式，如图 4-10 所示。引入类作用域操作的好处在于只有本类的对象才能使用该操作，一些通用的操作可以设计为类作用域操作，比如：产生对象和发现对象。需要注意的是，类作用域操作只能存取本类中的类作用域属性。

有一种特别的类，叫做持久类（persistent class）。如图 4-11 所示的类就是一个持久类。当产生对象的程序 draw（）运行结束时，所需的对象便生成了，同时生成的对象将自身存入数据库、文件或其它永久性的存储器中。通常该类还会有一个类作用域操作，用此操作完成对象的存储，通常这种操作表示为 STORE（）、LOAD（）或 CREATE（）等。在类图中，如果某个类需要定义为持久的类，那么需要在类的名字旁边附加上{persistent}标志（如图 4-11 所示）。

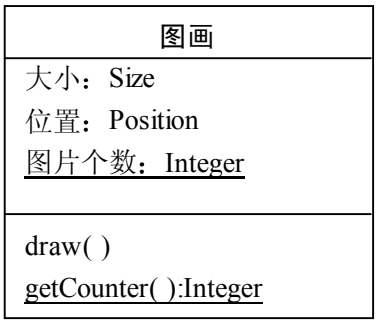



图 4-10　带有类作用域操作的类

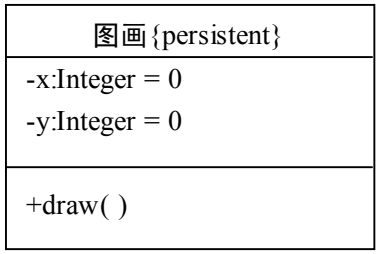



图 4-11　持久的类示例

图 4-11 的 JAVA 实现代码为：

```
public class Figure
{ private int x=0;
private int y = 0;
public void draw( )
{
   //实现画图的 JAVA 代码
}
};
```

产生图画对象的代码和调用 draw 操作的代码为：

```
Figure fig1 = new Figure();
Figure fig2 = new Figure();
```

```
fig1.draw();
fig2.draw();
```

还需说明的是，通常情况下，产生一个对象的操作和对该对象属性初始化的操作是分开进行的，第一步产生对象，第二步调用某个操作对类的属性初始化。为方便起见，我们可以像 C++或 JAVA 语言中定义构造函数那样定义一个操作，该操作具有创建对象并同时初始化对象的双重功能，操作名可与类名相同，如图 4-12 所示。

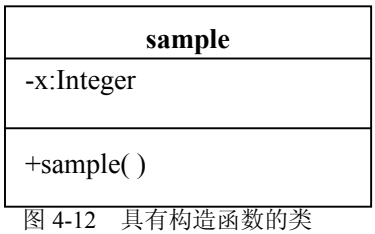


图 4-12　具有构造函数的类

图 4-12 所示类的 C++实现：

```
class sample
{
 protected:
int  x;
 public
sample( )  //定义构造函数
};
sample::sample( )  //实现构造函数
{
   x=0;
}
```

### 4.2.3　基本类型的使用

基本类型指的是像整型、实型、枚举型等这样的简单类型，它不是类。基本类型常被用来表示返回值的类型、参数的类型和属性的类型。UML 中没有预定义的基本类型，当用户在 CASE 工具中画 UML 中定义的各种图时，可以将 CASE 工具的工作环境配置为某一具体的编程语言，这样该语言本身所提供的基本类型就可以在 CASE 工具中使用了。如果不需要以某种具体语言为实现背景，或者不清楚用哪种语言，那么可以使用最简单的最常用的整型、字符串和浮点类型等，这些常用类型可以在 UML 语言中定义。以类图方式定义的类也可以用于定义属性类型、返回值类型和参数类型等等。

## 4.3　关　　系

前文已述，类图由类和它们之间的关系组成。类与类之间通常有关联、通用化（继承）、依赖和精化等四种关系。本节详细介绍这四种关系的含义和图示方法。

### 4.3.1　关联关系

关联用于描述类与类之间的连接。由于对象是类的实例，因此，类与类之间的关联也就是其对象之间的关联。类与类之间有多种连接方式，每种连接的含义各不相同（语义上的连接），但外部表示形式相似，故统称为关联。关联关系一般都是双向的，即关联的对

象双方彼此都能与对方通信。反过来说，如果某两个类的对象之间存在可以互相通信的关系，或者说对象双方能够感知另一方，那么这两个类之间就存在关联关系。描述这种关系常用的字句是：“彼此知道”、“互相连接”等。对于构建复杂系统的模型来说，能够从需求分析中抽象出类和类与类之间的关系是很重要的。

根据不同的含义，关联可分为普通关联、递归关联、限定关联、或关联、有序关联、三元关联和聚合等七种。

#### 4.3.1.1 普通关联

普通关联是最常见的一种关联，只要类与类之间存在连接关系就可以用普通关联表示。比如，张三使用计算机，计算机会将处理结果等信息返回给张三，那么，在其各自所对应的类之间就存在普通关联关系。普通关联的图示是连接二个类之间的直线，如图 4-13 所示。

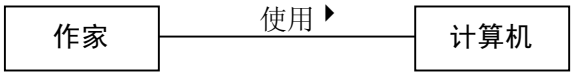



图 4-13　普通关联示例

由于关联是双向的，可以在关联的一个方向上为关联起一个名字，而在另一个方向上起另一个名字（也可不起名字），名字通常紧挨着直线书写。为了避免混淆，在名字的前面或后面带一个表示关联方向的黑三角，黑三角的尖角指明这个关联只能用在尖角所指的类上。通过直观的图示就可以很清楚地表达这种关联，比如图 4-13 的意思就是“某位作家使用计算机”。就像给类起的名字应能代表问题域本身的含义一样，给关联起的名字最好使用能够反映类之间关系的动词。

如果类与类之间的关联是单向的，则称为导航关联。导航关联采用实线箭头连接两个类。只有箭头所指的方向上才有这种关联关系，如图 4-14 所示，图中只表示某人可以拥有汽车，但汽车被人拥有的情况没有表示出来。其实，双向的普通关联可以看作导航关联的特例，只不过省略了表示两个关联方向的箭头罢了（类似于图的有向边和无向边）。

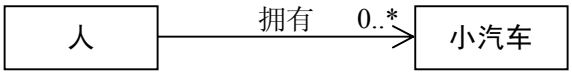



图 4-14　导航关联示例

除了上述的图示方式外，还可以在类图中图示关联中的数量关系——重数，比如，一个人可以拥有零辆车或多辆车。表示数量关系时，用重数说明数量或数量范围，也就是说，有多少个对象能被连接起来。

例：

0..1　　表示　零到 1 个对象

0..*或*　表示　零到多个对象

5..17　　表示　5 到 17 个对象

2　　表示　2 个对象

如果图中没有明确标识关联的重数，那就意味着是 1。类图中，重数标识在表示关联关系的某一方向上直线的末端。图 4-15 中关联的含义是：人可以拥有零到多辆车，车可

以被 1 至多个人拥有。而图 4-14 则只说明人可以拥有零至多辆车。

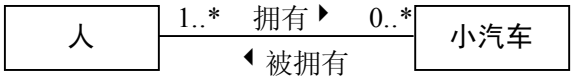



图 4-15　关联的重数示例

类图简单直观地描述了类、对象及它们之间的关系。通过类图可以反映出哪些对象之间有关系，系统是如何运作的。

对于一个复杂的系统来说，特别是信息系统、商务系统，应当利用构建的模型，模拟跟踪系统的工作状况，通过跟踪所建造的模型，验证系统的正确性和有效性，以便尽早发现系统模型的不足之处，及时更改、定型，加快后期开发工作。如果没进行跟踪验证，而在实现阶段发现问题，导致重新建模的话，将会增加系统开发成本和开发周期。所以，建模者在建模初期就要将问题考虑周全，使所建模型对各种情况都要有相应的处理结果，满足所解问题的需求。

类图虽小，但包含了很多信息。下面我们以图 4-16 为例，用自然语言表述该图的含义。保险公司有 0 或多个保险合同，这些合同与 1 个或多个客户有关；客户有 0 或多个保险合同，这些合同都与 1 个保险公司有关；保险合同位于一家保险公司和一个或多个客户之间，用它建立保险公司和客户之间的保险关系；保险合同用 0 或 1 个保险单表示，也就是合同的书面表示；一个保险单表示一份保险合同。

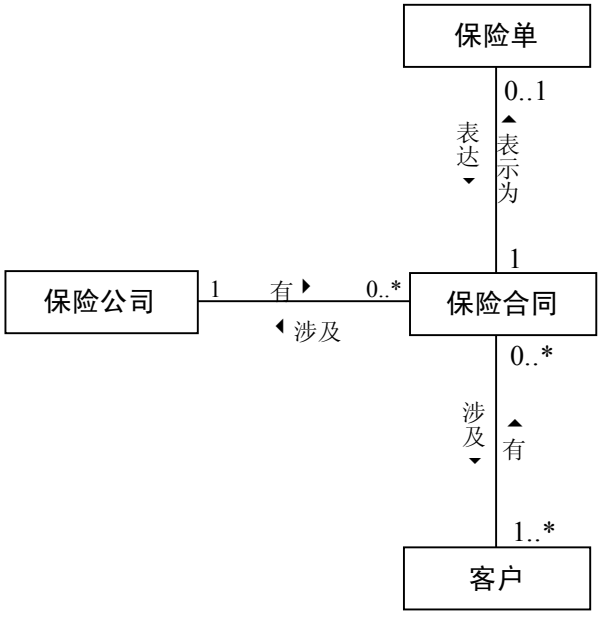



图 4-16　保险业务的类图

图 4-16 描述的保险业务类图中，保险合同用 0 或 1 个保险单表示，也就是保险合同可能有书面的保险单，可能没有书面的保险单。假如，某个客户打电话给某个保险公司请求保险他的汽车，若保险公司接受了他的请求，那么他与保险公司之间就有了保险合同，但是书面的保险单并不能马上得到，只能邮寄（或送达）。如果把这种模型当作真正的商务活动的描述，该模型是否可行呢？比如，某人打电话请求保险他的汽车，这时他与保险

公司之间就有了口头上的承诺。一分钟之后，它的汽车撞坏了，这时，他并没有保险单。如果保险业务以保险单为依据进行索赔，显然这种特殊情况发生时，该模型无法解决。从这个例子可以看出，建造一个真实业务的模型时，该模型一定要能够反映实际情况，而不能看起来“差不多”。一种解决上述问题的方法是开展网上业务，客户可以上网登记，获取另一种形式的保险单。对于建模者来说，只需在原来的类图上增加一个新的类——网上保险单，用该类完善系统的行为。从这个示例本身也可以看出，面向对象的建模方法，非常适合发展变化的问题，当需求发生变化时，只要修改或扩充一下原来的模型即可，并不需要推翻原始方案重新来，这也意味着原始的实现代码不必重新编写，只要相应地修改一下即可。

利用程序设计语言实现关联并不困难，比如，图 4-17 的 JAVA 实现大致为：


图 4-17　关联示例

```
public  class Insurance_company
{
    /*方法*/
    private  Insurance_contractVector  contracts;
         // Insurance_contractVector 是一个向量类的具体化，
           // Vector 是为动态数组提供的标准的类
}
public  class Insurance_contract
{
   /*方法*/
   private  Insurance_company  refer_to;
}
```

对于多对多的双向关联，可以转化为两个一对多的关联来实现，如图 4-18 所示。

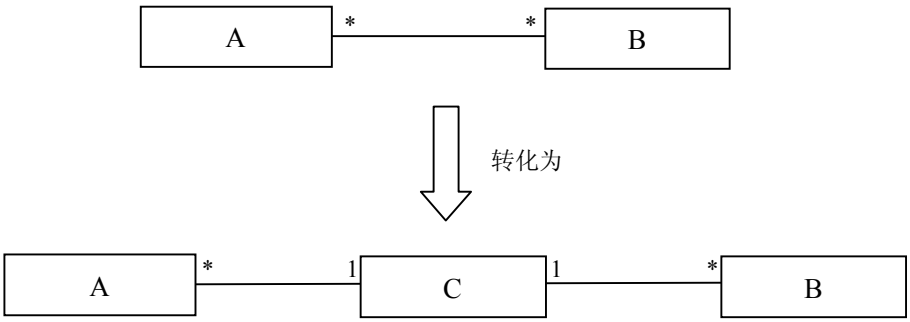



图 4-18　双向多对多关联的转化示例

#### 4.3.1.2　对象图

类图表示类和类与类之间的关系，对象图则表示在某一时刻这些类的具体实例和这些

实例之间的具体连接关系。由于对象是类的实例，所以，UML 对象图中的概念与类图中的概念完全一致，对象图可以看作类图的示例，帮助人们理解一个比较复杂的类图，对象图也可用于显示类图中的对象在某一点的连接关系。

对象的图示方式与类的图示方式几乎是一样的。主要差别在于对象的名字下面要加下划线。

对象名有下列三种表示格式。

第一种格式形如：

对象名：类名

即对象名在前，类名在后，中间用冒号连接。

第二种格式形如：

：类名

这种格式用于尚未给对象命名的情况，注意，类名前的冒号不能省略。

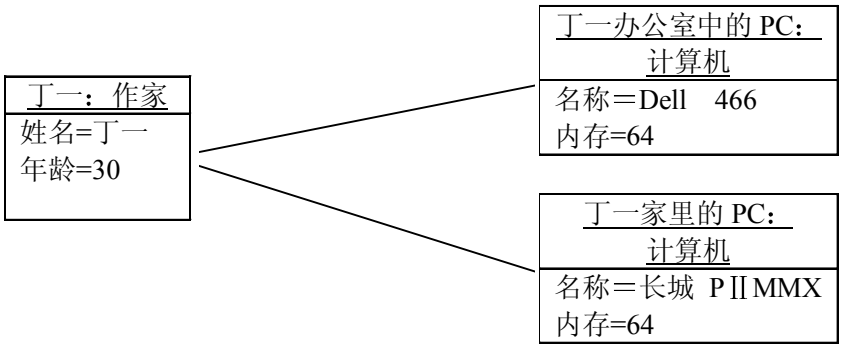
第三种格式形如：

对象名

这种格式不带类名（即省略类名）。

图 4-19 给出一个类图和一个对象图，其中对象图是类图的示例，读者可以比较一下两者之间有何异同。


（a） 类图




（b） 对象图

图 4-19 对象图示例

#### 4.3.1.3 递归关联

如果一个类与它本身有关联关系，那么这种关联称为递归关联（recursive association）。递归关联指的是同类的对象之间语义上的连接。比如，网络上的结点可以看作一个类，

那么用类图表示的网络结构如图 4-20 所示。显然，这是一个多对多的递归关联关系。假设网络中有 8 个结点（即 8 个对象），如果画出图 4-20 的对象图，则一种可能的连接方式如图 4-21 所示。图 4-21 才真正反映出对象之间的连接关系，而从其类图所看到的，只是类连接到该类本身。

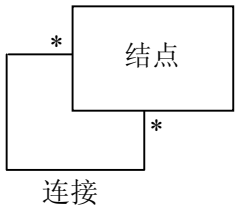



图 4-20　具有许多节点的网络的类图

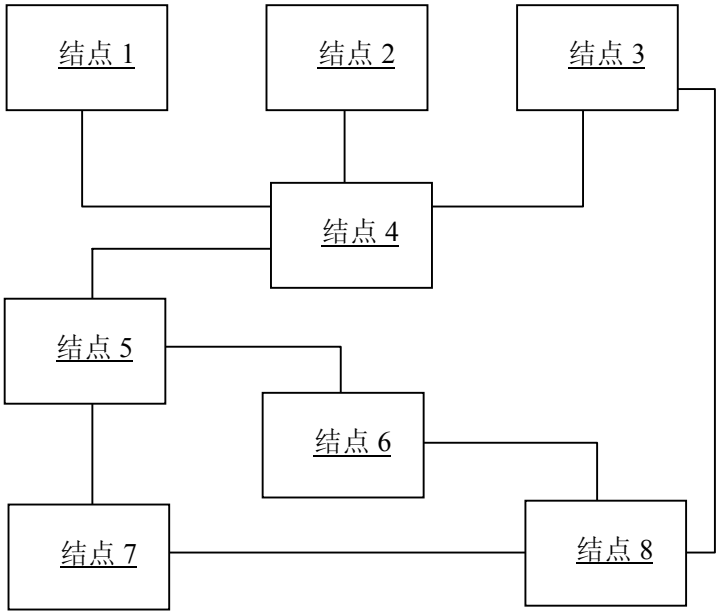


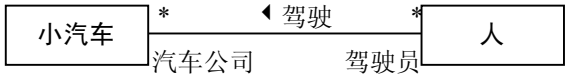

图 4-21　图 4-20 的一个对象图（8 个结点）

#### 4.3.1.4　关联中的角色

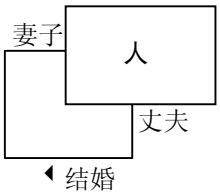
任何关联关系中都涉及到与此关联有关的角色，也就是与此关联相连的类中的对象所扮演的角色。比如，图 4-22（a）所示类图中的人和车，在人“驾驶”这个关联关系中，人扮演的是司机角色，车的所有权可由汽车公司扮演。又如图 4-22（b）中的“结婚”递归关联关系，一个人与另一个人结婚，必然一个扮演的是丈夫角色，另一个是妻子。如果一个人没有结婚，那么这个人不能作妻子或丈夫，因此，对这个人来说不能应用“结婚”关联关系。

关联中的角色通常用字符串命名。在类图中，把角色的名字放置在与此角色有关的关联关系（直线）的末端，并且紧挨着使用该角色的类。角色名是关联的一个组成部分，建模者可根据需要选用。引入角色的好处是：指明了类和类的对象之间的联系（CONTEXT）。注意，角色名不是类的组成部分，一个类可以在不同的关联中扮演不同的角色，比如图 4-

23 中的“人”就可以扮演“妻子”、“丈夫”、“保险客户”三个角色。




（a）




（b）

图 4-22　关联中的角色示例

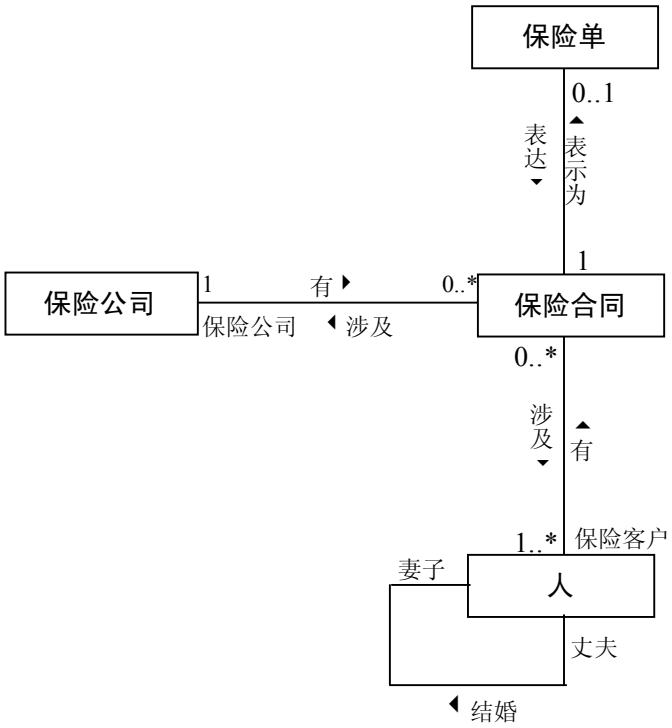



图 4-23　一个类在不同的关联中扮演不同的角色

#### 4.3.1.5　限定关联

限定关联用于一对多或多对多的关联关系中。在限定关联中，使用限定词将关联中多的那一端的具体对象分成对象集。限定词可以理解为一种关键词，用关键词把所有的对象分开。利用限定关联可以把模型中的重数从一对多变成一对一。类图中，限定词放置在关联关系末端的一个小方框内，紧挨着开始导航的类。如图 4-24 所示，图中从油画类到图画类的关联，图画用它的 ID 号表示。

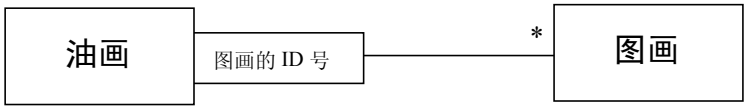



图 4-24　限定关联示例

#### 4.3.1.6　或关联

图 4-25 是保险业务的类图，从图中可以看出，人可以与保险公司建立保险合同，一般公司也可以与保险公司建立保险合同，显然人持有的合同与一般公司持有的合同应该不同，也就是说，人与保险合同的关联关系不能同公司与保险合同的关联关系同时发生。为了解决这类问题，人们引入了“或关联”。所谓或关联就是对二个或更多个关联附加的约束条件，使类中的对象一次只能应用于一个关联关系中。或关联的图示方法是，具有“或关系”的关联之间用虚线连接起来，虚线上方标注规格说明{或}，如图 4-26 所示。

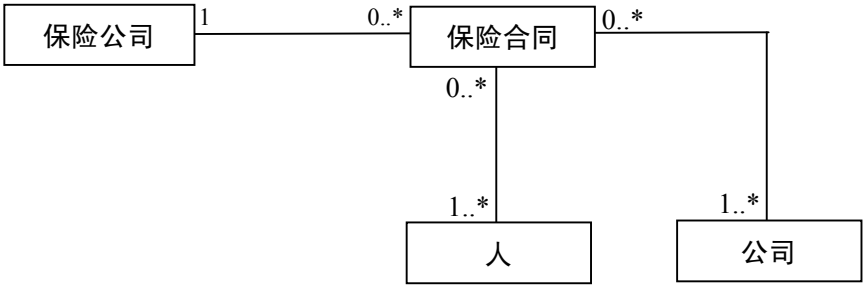



图 4-25　具有或关联的类图

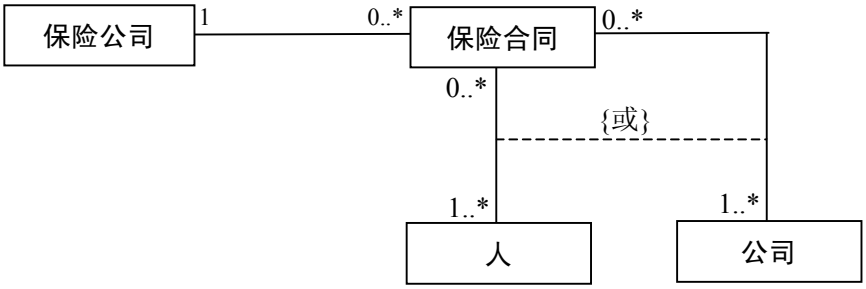



图 4-26　或关联的类图图示

#### 4.3.1.7　有序关联

对象与对象之间的连接可以具有一定的次序，就像应把窗口安排在屏幕之上一样。一般情况下，对象之间的关联都是无序的，如果要明确表示关联中的次序关系，一定要将规格说明{排序}放在表示关联的直线旁，且紧挨着对象被排序的类，如图 4-27 所示。将对象排序的具体方法可以写在关联的性质中，也可以标识在有序关联的花括号内，比如：{按时间递增顺序排序}。

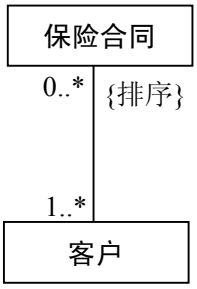



图 4-27　有序关联示例

#### 4.3.1.8　关联类

与一个关联关系相连的类，称作关联类。关联类并不位于表示关联关系的直线两端，而是对应一个实际的关联，用关联类表示该实际关联的一些附加信息。关联中的每个连接与关联类中的对象相联系。比如图 4-28 是一个电梯系统的模型，队列就是电梯控制器类与电梯类的关联关系上的关联类。从图中可以看出一个电梯控制器操纵着 4 台电梯，这样控制器和电梯之间的实际连接就有 4 个，每个连接都对应一个队列（对象），每个队列（对象）存储着来自控制器和电梯内部按钮的请求服务信息。电梯控制器通过读取队列信息，选择一个合适的电梯为乘客服务。关联类与一般的类一样，也有属性、操作和关联。

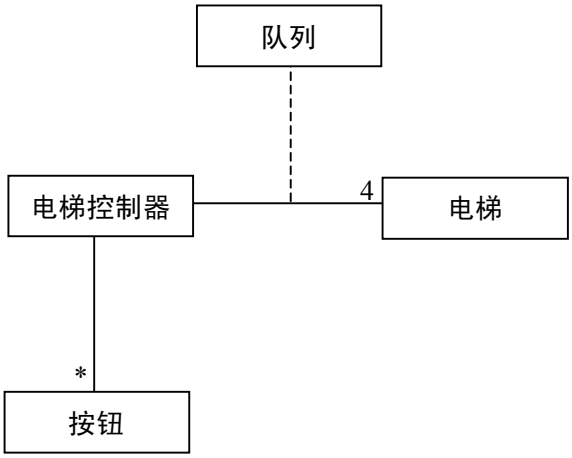



图 4-28　关联类示例

#### 4.3.1.9　三元关联

类与类之间的关联关系，不仅限于两个类之间，多个类之间也可以有关联关系。如果三个类之间有关联关系，则称之为三元关联。三元关联图示为一个大的菱形，菱形的角与关联的类之间用直线相连（也可以用虚线连接），如图 4-29 所示。注意，三元关联中可以出现角色和重数，但不能出现限定词和聚合（后文介绍）。

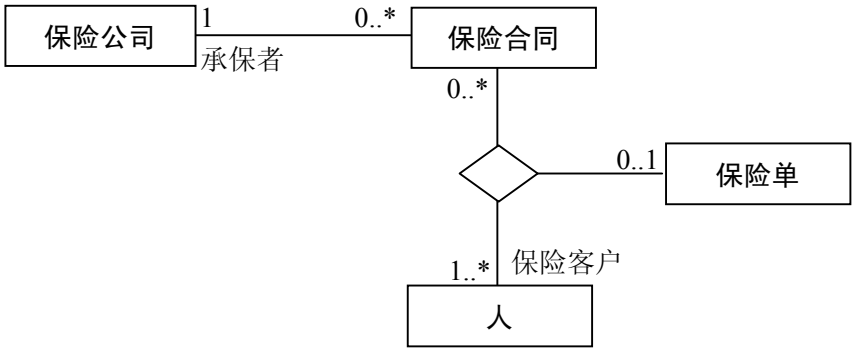



图 4-29　三元关联示例

图 4-29 的含义是，扮演保险客户角色的“人”可以有 0 至多个保险合同，每个保险合同都与一家保险公司有关，保险公司扮演着承保者的角色。顾客和保险合同通过 0 或 1 个保险单建立三元关联关系。

#### 4.3.1.10　聚合（aggregation）

聚合是关联的特例。如果类与类之间的关系具有“整体与部分”的特点，则把这样的关联称为聚合。例如，汽车由四个轮子、发动机、底盘等构成，则表示汽车的类与表示轮子的类、发动机的类、底盘的类之间的关系就具有“整体与部分”的特点，因此，这是一个聚合关系。识别聚合关系的常用方法是寻找“由……构成”、“包含”、“是……的一部分”等字句，这些字句很好地反映了相关类之间的“整体—一部分”关系。

聚合的图示方式为，在表示关联关系的直线末端加一个空心的小菱形，空心菱形紧挨着具有整体性质的类，如图 4-30 所示，聚合关系中可以出现重数、角色（仅用于表示部分的类）和限定词，也可以给聚合关系命名，图 4-30 所示的聚合关系表示海军由许多军舰组成。

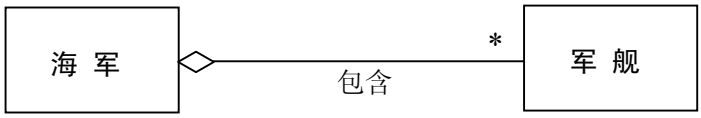



图 4-30　聚合的示例

除去上述的一般聚合外，聚合还有二种特殊的聚合方式，共享聚合和复合聚合。

如果聚合关系中的处于部分方的对象同时参与了多个处于整体方对象的构成，则该聚合称为共享聚合。比如，一个球队（整体方）由多个球员（部分方）组成，但是一个球员还可能加入了多个球队，球队和球员之间的这种关系就是共享聚合。共享聚合关系可以通过聚合的重数反映出来（不必引入另外的图示符号），如果作为整体方的类的重数不是 1，那么该聚合就是共享聚合，如图 4-31 所示。图中从球员到球队的关联的重数为多个（*号）。共享聚合是一个网状结构的关联关系。


图 4-31　共享聚合示例

如果构成整体类的部分类，完全隶属于整体类，则这样的聚合称为复合聚合。换句话说，如果没有整体类则部分类也没有存在的价值，部分类的存在是因为有整体类的存在。比如，窗口由文本框、列表框、按钮和菜单组成。整体方的重数必须是零或 1（0..1），部分方的重数可取任意范围值。复合聚合是一个树状结构的关联关系。复合聚合图示为一个带实心菱形的直线，实心菱形紧挨着表示整体方的类，如图 4-32 所示。

还可以将图 4-32 简化成图 4-33 的样子，将几个实心菱形合并为一个，用直线分支地连接到各个部分类，构成一个树状结构。这种图示方法适用于各种形式的聚合关系。此外，对于具有多个部分类的复合聚合（即图 4-32 所示的情况），还可以将部分类画在整体类的内部，如图 4-34 所示。

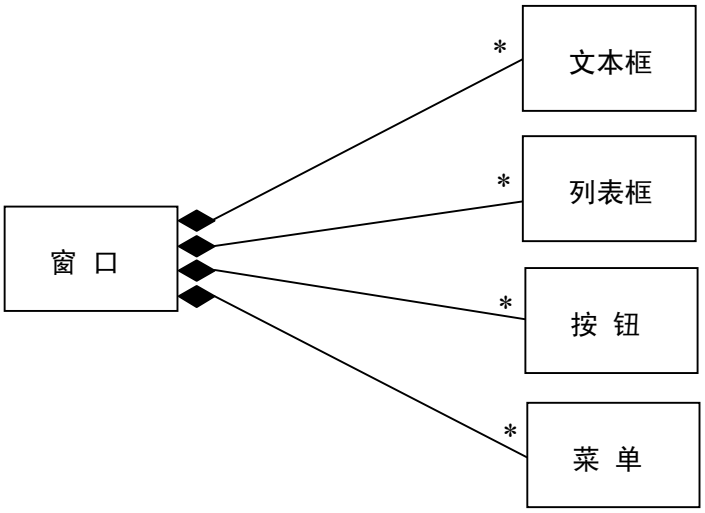



图 4-32　复合聚合图示一

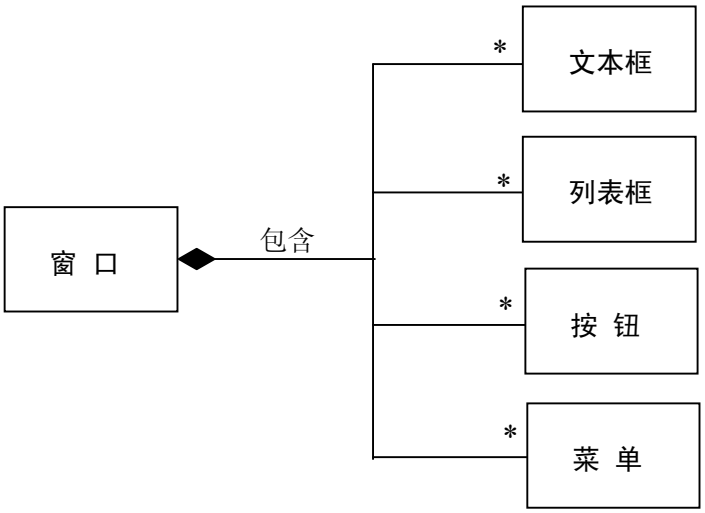



图 4-33　复合聚合图示二

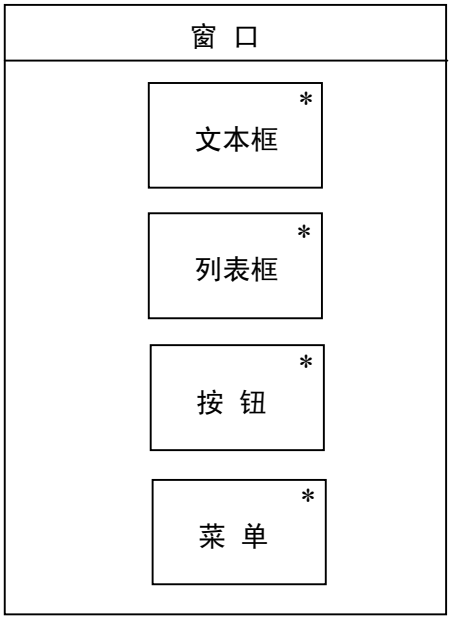



图 4-34　复合聚合图示三

如图 4-35 是一个带角色的复合聚合的示例。图中给出了三种图示方法，角色名位于部分类一方（按钮和图标），4.35（a）是带角色名的复合聚合图示；4.35（b）是带角色的复合聚合的标准语法形式；4.35（c）采用将属性名变成角色名，将属性的类型变成类的方法的形式图示复合聚合。

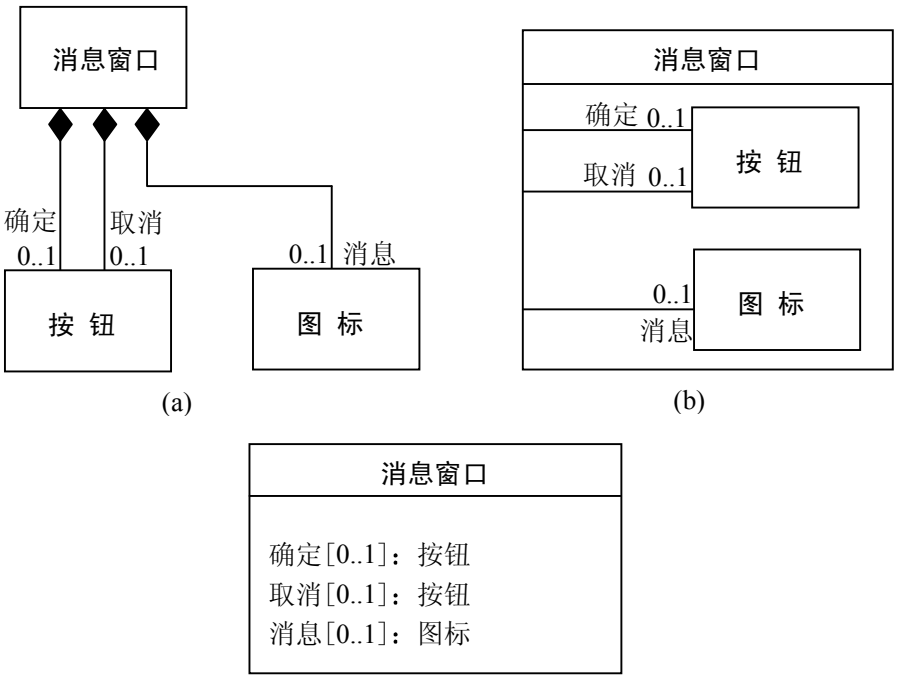



图 4-35　带角色的复合聚合的示例

复合聚合的实现方式一般是将部分类中的对象作为整体类的成员对象（member object），将部分类封装（encapsulating）在整体类中，如图 4-35（b）所示。

### 4.3.2 通用化

一个类（通用元素）的所有信息（属性或操作）能被另一个类（具体元素）继承，继承某个类的类中不仅可以有属于自己的信息，而且还拥有了被继承类中的信息，这种机制就是通用化。

通用化又称继承。UML 中的通用化是通用元素和具体元素之间的一种分类关系。具体元素完全拥有通用元素的信息，并且还可附加一些其它信息。比如，小汽车是交通工具，如果定义了一个交通工具类表示关于交通工具的抽象信息（发动、行驶等），那么这些信息（通用元素）可以包含在小汽车类（具体元素）中。引入通用化的好处在于由于把一般的公共信息放在通用元素中，处理某个具体特殊情况时只需定义该情况的个别信息，公共信息从通用元素中继承得来，增强了系统的灵活性、易维护性和可扩充性。程序员只要定义新扩充或更改的信息就可以了，旧的信息完全不必修改（仍可以继续使用），大大缩短了维护系统的时间。

通用化可用于类、用例等各种模型元素。注意，通用化针对类型，而不针对实例。比如，一个类继承另一个类，但一个对象不能继承另一个对象。

通用化分为普通通用化和受限通用化，下面详细介绍这两种通用化的具体内容和图示方法。

#### 4.3.2.1 普通通用化

具有通用化关系的两个类之间，继承通用类所有信息的具体类，称为子类，被继承类称为父类。可以从父类中继承的信息有属性、操作和所有的关联关系。

父类与子类的通用化关系图示为一个带空心三角形的直线。空心三角形紧挨着父类，如图 4-36 所示。图中交通工具是父类，其余三个类是从其派生出的子类。也可以像聚合关系的图示一样，把图 4-36 中的指向父类的三角形合成一个，其它的子类用带的分支的直线相连，如图 4-37 所示。

父类中的属性和操作又称作成员，不同可见性的成员在子类中用法不同。

父类中公有的成员在被继承的子类中仍然是公有的，而且可以在子类中随意使用；父类中的私有成员在子类中也是私有的，但是子类的对象不能存取父类中的私有成员。

一个类中的私有成员都不允许外界元素对其作任何操作，这就达到了保护数据的目的。

如果既需要保护父类的成员（相当于私有的），又需要让其子类也能存取父类的成员，那么父类的成员的可见性应设为保护的。拥有保护可见性的成员只能被具有继承关系的类存取和操作。具有保护可见性的成员名字前面通常加一个“#”号。类图中可以不表示该符号。

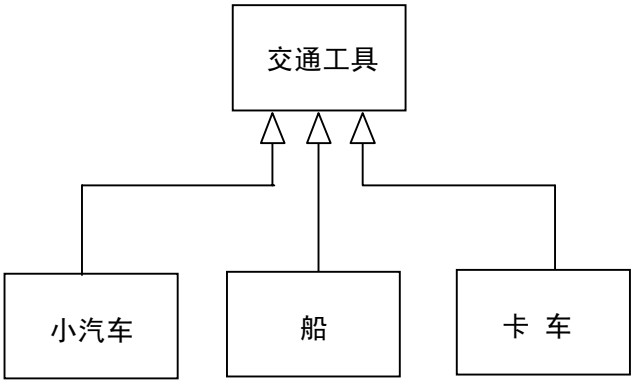



图 4-36　通用化关系示例一

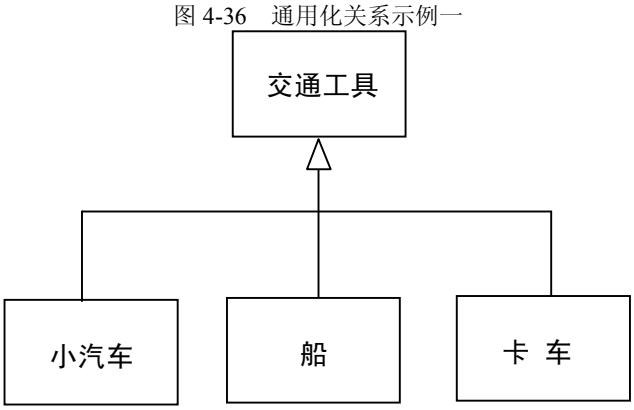



图 4-37　通用化关系示例二

类的继承关系可以是多层的。也就是说，一个子类本身还可以作另一个类的父类，层层继承下去。如图 4-38 所示。图中“车”是“交通工具”的子类，同时又是“卡车”的父类。

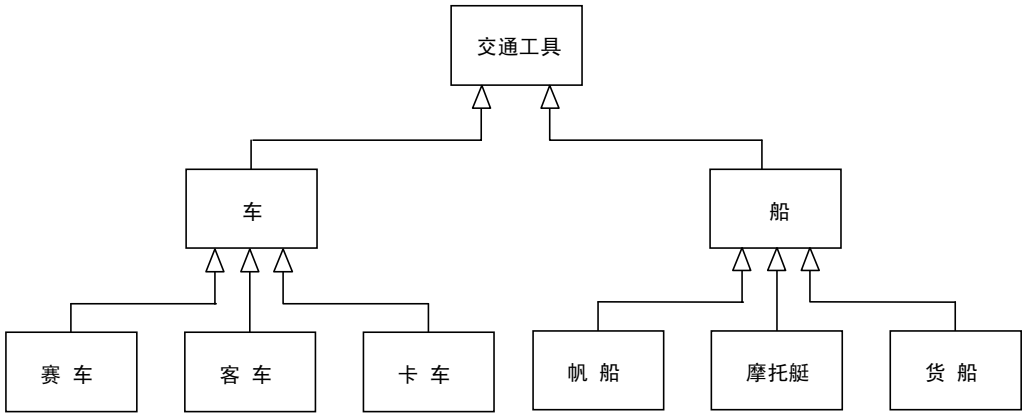



图 4-38　类的多层继承示例

没有具体对象的类称作抽象类。抽象类一般做为父类，用于描述其它类（子类）的公共属性和行为（操作）。比如，图 4-38 中的“交通工具”就是一个抽象类，很难想象该类的对象是什么样子？因为它不能是车，也不能是船，所以认为该类没有对象，但是它描述了交通工具的一般特征。图示抽象类时，在类的名字下方附加一个标记值{abstract}，如图 4-39 所示。

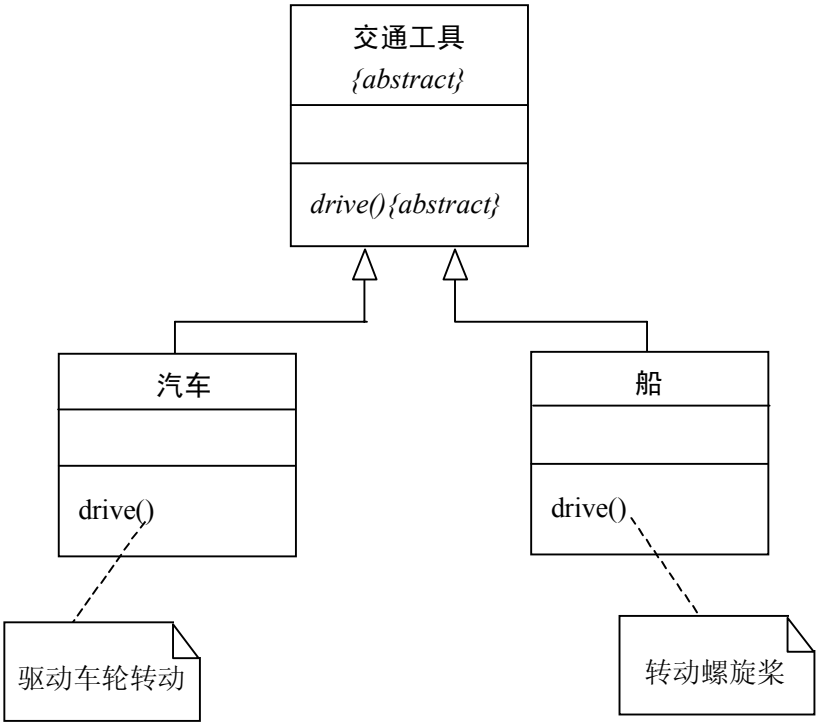



图 4-39　抽象类示例

抽象类中一般都带有抽象的操作。抽象操作仅仅用来描述该抽象类的所有子类应有什么样的行为，抽象操作只标记出返回值、操作的名称和参数表，关于操作的具体实现细节并不详细书写出来，抽象操作的具体实现细节由继承抽象类的子类实现。换句话说，抽象类的子类一定要实现抽象类中的抽象操作，为抽象操作提供方法（算法实现），否则该子类仍然还是抽象类。抽象操作的图示方法与抽象类相似，是在操作标记后面跟随一个性质串{abstract}。操作标记和性质串还可以用斜体字表示，如图 4-39 中的抽象操作 *drive（）{abstract}*，而子类“汽车”和“船”中分别实现了 drive（）的具体操作。当然，抽象类“交通工具”中还可以有其它抽象操作，比如 stop（）。

与抽象类恰好相反的类称为具体类。具体类有自己的对象，并且该类中的操作都有具体实现的方法。比如，图 4-39 中的“汽车”和“船”二个类就是具体类，“汽车”中的 drive（）具体实现为驱动车轮滚动，而“船”类中的 drive（）则具体实现为发动螺旋桨转动。比较一下抽象类与具体类，不难发现，子类继承了父类的操作，但是子类中对操作的实现方法却可以不一样，这种机制带来的好处是子类可以重新定义父类的操作。重新定义的操作的标记（返回值、名称和参数表）应和父类一样，同时该操作既可以是抽象操作，也可以是具体操作。当然，子类中还可以添加其它的属性、关联关系和操作。

如果图 4-39 中添加人“驾驶”交通工具这个关联关系，则如图 4-40 所示。当人执行（调用）drive（）操作时，如果当时可用的对象是汽车，那么汽车轮子将被转动；如果当时可用的对象是船，那么螺旋桨将会动起来，这种在运行时可能执行的多种功能，称为多态。多态技术利用抽象类定义操作，而用子类定义处理该操作的方法，达到单一接口，多种功能的目的，在 C++语言中，多态利用虚拟函数或纯虚拟函数实现。

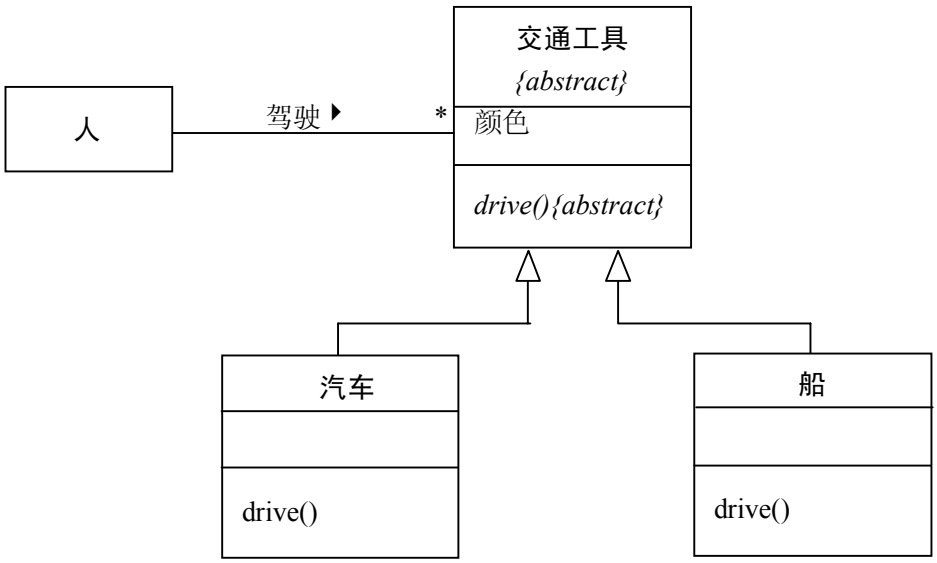



图 4-40　多态示例

图 4-41 给出一个比较复杂的类图示例。一幅油画由许多图形组成，图形可以有直线、圆、多边形和各种线型混合而成的组合图等。当客户要求画某幅油画时，油画便通过与图形之间的关联关系，由图形来完成作画任务，由于图形是抽象类，所以涉及到具体的某一图形时（比如：线、圆等）则使用其相应子类中具体实现的了 draw（）功能完成绘画。这些工作都是由系统自动完成的。

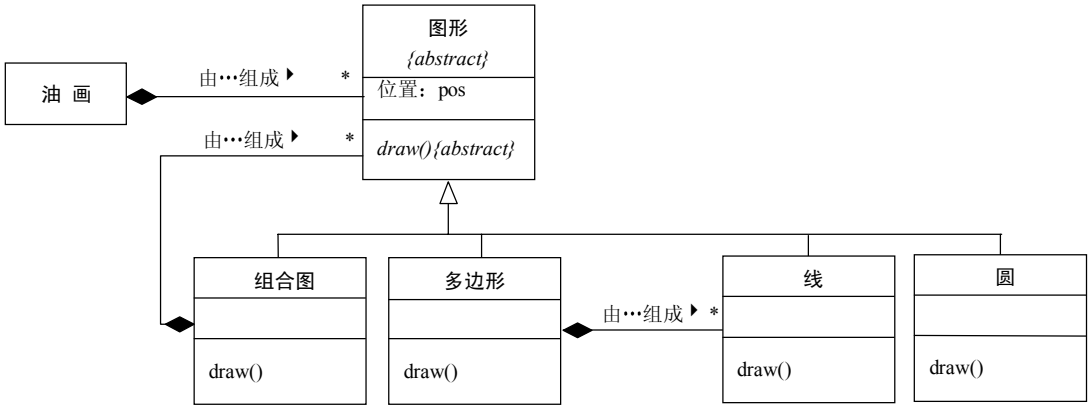



图 4-41　复杂类图的示例

可以在类图中标注把一般类和具体类划分开的分辨器。分辨器说明划分的依据。例如，把运输工具划分为汽车和船的依据是动力装置（推进器），不同的动力装置代表不同的类，如图 4-42 所示。

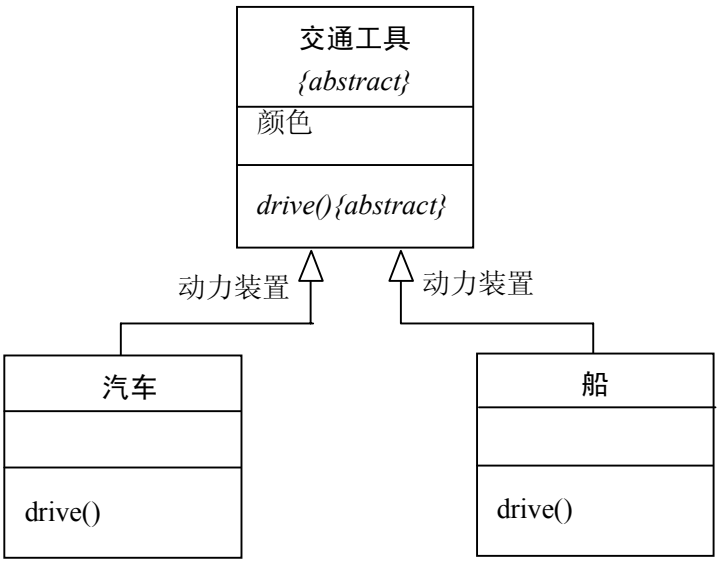



图 4-42　带分辨器的类图

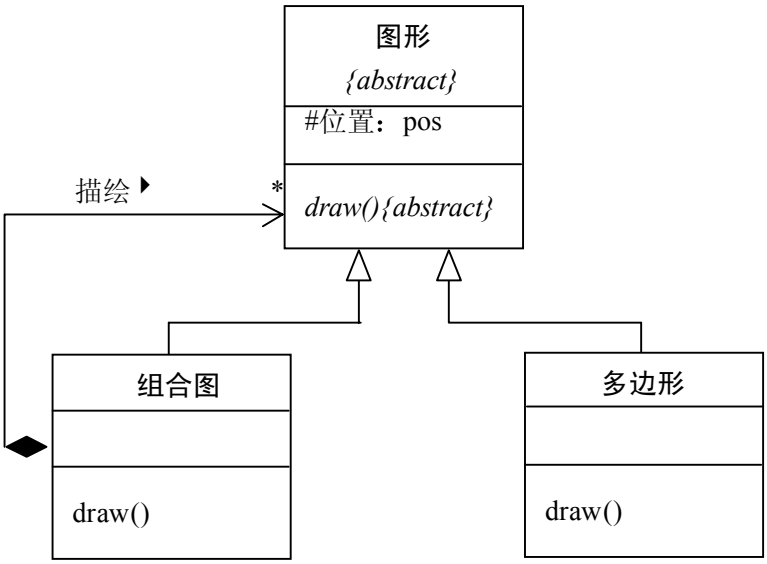



图 4-43　图形的组成

图 4-43 的 JAVA 实现的大体轮廓是：

```
abstract  public class Figure
{
  abstract public void draw();
  protected  Pos  position;
}

public class Group extends Figure
{
```

```
public void draw()
  {
    for  (int i = 0; i < consist_of.size(), i++)
     {
       consist_of[i].draw();
     }
  }
private FigureVector figures;
}

public class Polygon extends Figure
{
public void draw()
{
/*画多边形的代码*/
}
}
```

#### 4.3.2.2 受限通用化

给通用化关系附加一个约束条件，进一步说明该通用化关系的使用方法或扩充方法，这样的通用化关系称为受限通用化。预定义的约束有四种：多重、不相交、完全和不完全。这些约束都是语义上的约束。

图 4-44 图示了二种约束通用化的表示方法。图 4-44（a）是多个子类共用一个箭头指向父类，约束用花括号括起来放在直线旁边，多个约束之间用逗号分隔。图 4-44（b）中的继承关系是单独图示的，这种情况下，要另外附加一条虚线（穿越所有的继承关系）。

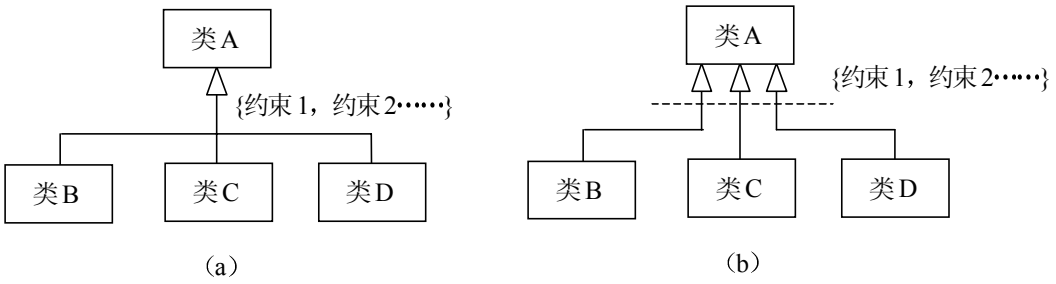



图 4-44　受限通用化的图示

多重继承指的是，子类的子类可以同时继承多个上一级子类。图 4-45 所示的“水陆两用”类就是通过多重继承得到的。由于“汽车”和“船”与“交通工具”之间的继承被指定为多重继承，所以子类“汽车”和“船”能被“水陆两用”类同时继承。这样，允许多重继承的父类（交通工具类）被“水陆两用”类继承了两次。

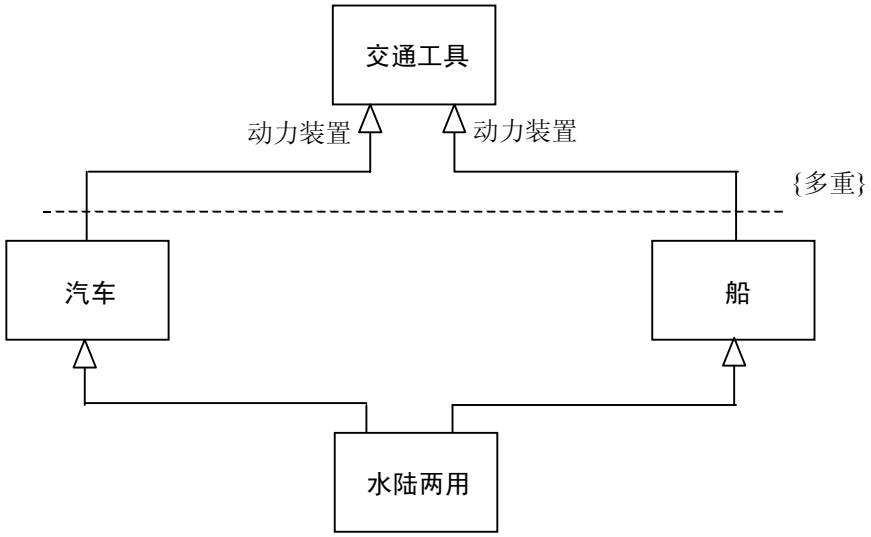



图 4-45　多重继承示例

与多重继承对立的是不相交继承，即一个子类不能同时继承多个上一级子类。比如图 4-45 中如果没有指定“多重”约束，则不允许“水陆两用”类如此继承的。如果不作特别声明，一般的继承都是不相交继承。

完全继承指的是父类的所有子类都被穷举完毕，不可能再有其它的未列出的子类存在，如图 4-46 所示。

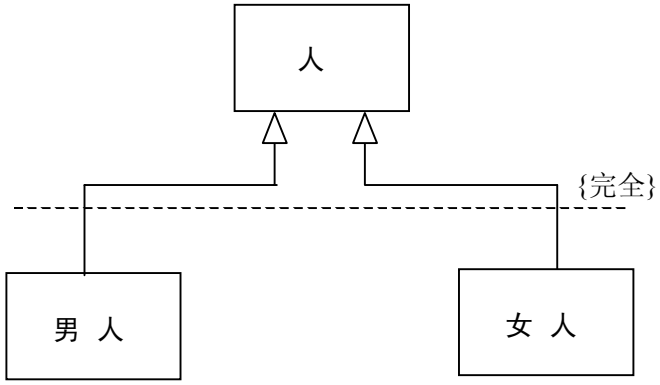



图 4-46　完全继承示例

非完全继承恰好与完全继承相反，父类的子类并不是无一遗漏地列出，而是随着问题不断地解决，不断地补充和完善，也正是这一点为日后系统的扩充和维护带来极大的方便，非完全继承是一般情况下默认的继承标准。

### 4.3.3　依赖和精化关系

依赖关系描述的是两个模型元素（类、组合、用例等）之间的语义上的连接关系。其中，一个模型元素是独立的，另一个模型元素是非独立的（依赖的），它依赖于独立的模型元素，如果独立的模型元素发生改变，将会影响依赖该模型元素的模型元素。比如，某个类中使用另一个类的对象作为操作中的参数，则这二个类之间就具有依赖关系，类似的依赖关系还有一个类存取另一个类中的全局对象，以及一个类调用另一个类中的类作用域

操作。图示具有依赖关系的二个模型元素时，用带箭头的虚线连接，箭头指向独立的类，箭头旁边还可以带一个版类（stereotype）标签，具体说明依赖的种类，比如图 3.10 中所示的版类《实现》。

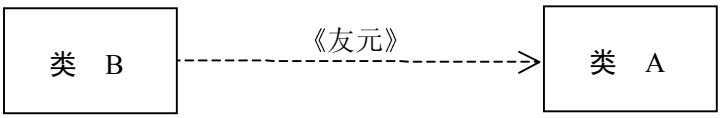



图 4-47　友元依赖关系示例

图 4-47 表示一个友元依赖（friend dependency）关系。友元依赖关系使得其它类中的操作可以存取该类中的私有或保护属性。在 C++语言中，可以通过定义友元类存取独立类中的成员，图 4-47 中的类 B 是类 A 的友元类，用 C++实现的示意性程序为：

```
#include  <iostream.h>
class A
{  int x;
public:
A ( )   //构造函数
 {  X=5;  }
friend class B  //定义 A 的友元类 B
};

class B
{
public:
void display(A  tmp)
{
cout<<" x="<<tmp.x<<endl;
}
};

void main()
{   A obj1;
B obj2;
obj2.display(obj1);
}
```

执行该程序，输出结果为：x=5

从该示例中看出类 B 存取了类 A 中的私有属性。

精化关系用于表示同一事物的两种描述之间的关系。对同一事物的两种描述建立在不同的抽象层上。比如，定义了某种数据类型，然后将其实现为某种语言中的类，那么抽象定义的类型与用语言实现的类之间就是精化关系，这种情况也被称为实现。

又如，对某事物的精确描述和粗糙描述。

精化关系常用于模型化表示同一个事物的不同实现。比如，一个是简单实现，一个是

比较复杂而高效的实现。精化关系的图示方法与继承关系相似，用带空心三角形的虚线表示。图 4-48 表示为同一事物建模时，构造的分析类和设计类之间的精化关系。

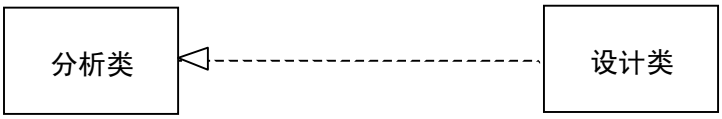



图 4-48　精化关系的图示

精化用于模型的协调。在对大型工程项目建模时，往往需要建立许多模型。然后用精化关系对这些模型进行协调。协调过程可以显示不同抽象层上的模型之间是怎样联系在一起的；显示来自不同建模阶段的模型之间具有什么样的关系；支持配置管理和模型之间的追溯。

## 4.4　约束和派生规则

前面几节中，我们已讨论了类、对象、类与类之间的关系，本节讨论 UML 的规则和语法表示方法。

UML 中的规则称为约束和派生。约束用于限制一个模型。我们已经讨论过的约束有或关联、有序关联和四种继承约束（多重、不相交、完全和不完全）。派生用于描述某种事物的产生规则，比如说，一个人的年龄可以由该人的生日和当前日期派生出来（两者之差）。一般说来，约束和派生能应用于任何模型元素，但最常用于属性、关联、继承、角色和时间（动态模型中的角色和时间约束在后面介绍）。图示中的约束和派生都用花括号括起来放在模型元素的附近，或者用圆括号括起来，以笔记的方式与模型元素相连。

关联关系可以被约束，也可以派生。如果一个关联是另一个关联的子集，则它们之间就会存在约束关联。如图 4-49 中的“领导”关联中的政治家是“组成”关联中成员的子集。派生关联由现有的关联关系衍生而来，图示时，派生关联的名称前加一条斜线，比如图 4-50 中的“VIP 客户”关联就是派生来的。因为出租车公司和很多客户有租车合同，这里面有些人是很重要的，于是在公司和客户之间直接派生出关联关系。


图 4-49　约束关联示例

属性也可以被约束或派生。对属性约束的方式表现为给某个属性定初值或确定取值范围，比如：

```
+status: Status=unpaid{unpaid, paid}
+color: Color=red {red, green, yellow}
-administrator: String=“wwcd”
```

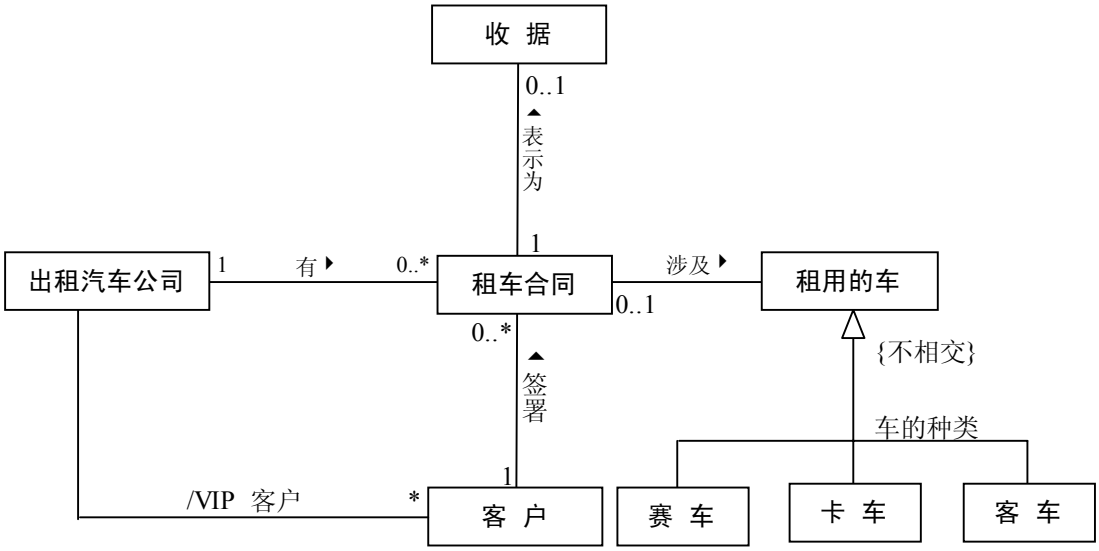
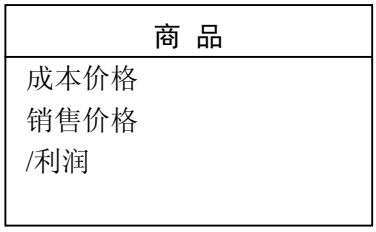



图4-50 派生关联示例

其中颜色这个属性的性质串还可以表示为{0<=Color<=255}。派生属性由其它属性通过某种方式计算得来。派生属性前面加一个斜线表示，它并不真正出现在类的对象中，派生属性的计算公式用括号括起来放在类的下方，如图 4-51 所示。




{利润=销售价格-成本价格}

图 4-51 派生属性示例

通用化关系只有约束，没有派生。通用化的四种约束（多重、不相交、完全和不完全）前文已述，这里不再重复。对角色的约束为了防止一个对象所扮演的多个角色连在一起。当然对时间也有约束。这些内容将在第五章和第八章中讨论。

对 UML 模型元素应用的约束和派生（规则），也可以用 UML 语言语法机制表示。表示规则的语法称为导航表达式，它构成说明一个具体规则的基本语句。根据需要，有时可以扩展导航表达式。

下面介绍五种常见语法的书写形式：

形式一 set.attribute

set 是一个表达式，代表一个对象或对象集，attribute 是 set 所代表对象的一个属性名，它们之间用“.”号相连接。形式一的结果是属性的值，结果值可能是单值，也可能是多值，具体结果依赖于关联的重数。

形式二 set.role

其中 set 的含义同形式一，role 代表关联关系中目的方角色名，它们之间用“.”相连。形式二的结果为一个或多个对象，对象的多少依赖于关联的重数。

形式三　　set.~role

形式三与形式二的含义差不多，不同之处在于 role 代表关联关系中的起始方角色名，role 前面多加一个“~”符号，表示对关联关系的逆转。具体结果仍然是与关联的重数有关的对象或对象集。

形式四　　set [布尔表达式]

set 是代表一个对象或多个对象的表达式，布尔表达式用 set 中的对象书写，并用方括号括起来。形式四的结果值是使布尔表达式为真的对象（是 set 的一个子集）。

形式五　　set.[限定词的值]

set 是代表一个对象或多个对象的表达式，限定词指明一个限定 set 的限定关联，限定词代表限定关联中的限定属性值。

例如：

```
Insurance_Contract.Policyholder>0
Person.~Policyholder.sum_insured>1000￥
Car.Driver.driving_license=True
Person[Supplier.Prospect]<Person[Supplier.Suspect]
```

## 4.5　接　　口

有一定编程经验的人或者熟悉计算机工作原理的人都知道，通过操作系统的接口可以实现人机交互和信息交流。UML 中的包、组件和类也可以定义接口，利用接口说明包、组件和类能够支持的行为。在建模时，接口起到非常重要的作用，因为模型元素之间的相互协作都是通过接口进行的。一个结构良好的系统，其接口必然也定义得非常规范。

接口通常被描述为抽象操作，也就是只用标识（返回值、操作名称、参数表）说明它的行为，而真正实现部分放在使用该接口的元素中。这样，应用该接口的不同元素就可以对接口采用不同的实现方法。在执行过程中，调用该接口的对象看到的仅仅是接口，而不管其它事情，比如，该接口是由哪个类实现的，怎样实现的，都有哪些类实现了该接口等。通俗地讲，接口的具体实现过程、方法，对调用该接口的对象是透明的。如果读者对 C++ 中的虚拟函数比较了解的话，就不难理解这种运行时的多态。

接口在类图中表示为一个带接口名称的小圆圈。接口与应用它的模型元素之间用一条直线相连（模型元素中包含了接口的具体实现方法），它们之间是一对一的关联关系。调用该接口的类与接口之间用带箭头的虚线连接，它们之间是依赖关系。如图 4-52 中的类 A 实现了二个接口“存储”和“运行”，类 C 实现了一个接口“运行”。显然，类 A 与类 C 中对“运行”的实现方法可能不同，当类 B 中的对象调用接口“运行”中的某个操作时，系统会自动选择一个合适的“运行”执行，而类 B 中的对象并不考虑究竟调用哪个“运行”。注意，类 B 只与接口之间有依赖关系，并不与实现该接口的类（例如，类 A）有依赖关系。如果类 B 与类 A 有依赖关系，那么类 A 和类 B 之间应直接画一条带箭头的虚线。

为了具体标识接口中的操作，接口也可以图示为带版类《接口》的类，如图 4-52 左下角所示。类图中反映不出接口中的操作，前面所说的调用“运行”，更确切地讲应为调

用该接口中的 run（）操作。接口之间也有继承关系，它与类图中类的继承描述方式是一样的。

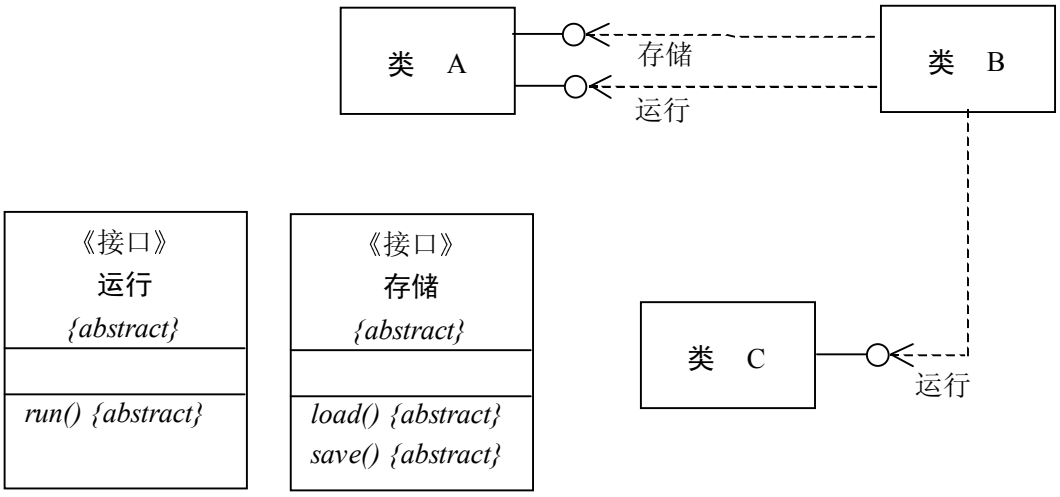



图 4-52 接口示例

接口可以采用 OLE/COM 或 JAVA 的接口实现，因为这二种编程方式允许把接口的说明和实现分开来，这样方便许多具体的类（或包或组件）选择实现某个接口。

图 4-52 中对类 C 和“运行”接口进行声明的 JAVA 代码大致为：

```
interface  Runnable
{
public  void  run ()
}

public  class  C  implements  Runnable
{
public  void  run ()
{
//  C 中 run () 操作的实现细节
}
}
```

## 4.6 包

包（package）是一种组合机制，把各种各样的模型元素通过内在的语义连在一起成为一个整体就叫做包。构成包的模型元素称为包的内容。包通常用于对模型的组织管理，因此有时又将包称为子系统（subsystem）。包拥有自己的模型元素，包与包之间不能共用一个相同的模型元素。包的实例没有任何语义（含义）。仅在模型执行期间，包才有意义。

包能够引用来自其它包的模型元素。当一个包从另一个包中引用模型元素时，这两个

包之间就建立了关系。包与包之间允许建立的关系有依赖、精化和通用化。注意，只能在类型之间建立关系，而不是在实例之间建立，因为包的实例没有语义。

包图示为类似书签卡片的形状，由二个长方形组成，小长方形（标签）位于大长方形的左上角。如果包的内容（比如类）没被图示出来，则包的名字可以写在大长方形内，否则包的名字写在小长方形内，如图 4-53 所示。

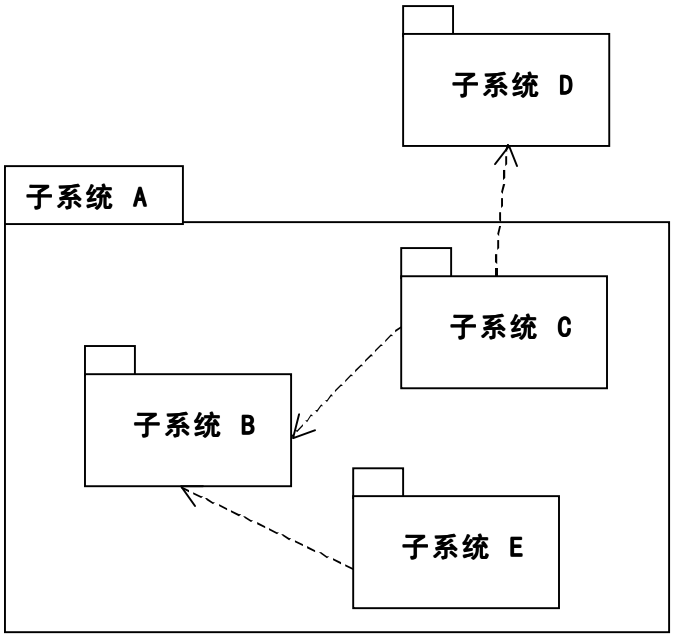



图 4-53　包的图示

包与聚合很相似，如果一个包是由模型元素构成的（拥有自己的内容），那么该包是复合聚合；反之，如果一个包从其它的包中引用模型元素，该包是共享聚合。

图 4-53 示意了包之间的依赖关系。图中 E 依赖 B（B 中元素被 E 引用），C 依赖 B 和 D（依赖关系使用版类《输出》），B，C，E 在 A 中。图 4-54 中 D，E 继承 C，而 B，C，D，E 构成 A。图 4-55 示意子系统 W 中的子系统 Z 被子系统 A 所引用。

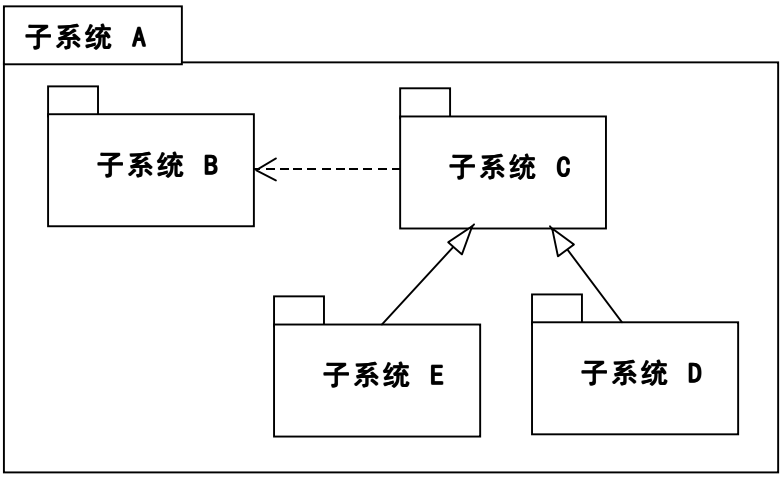
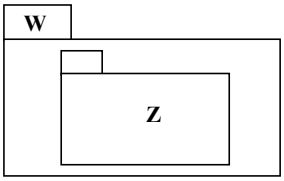


图 4-54　包之间的继承示例



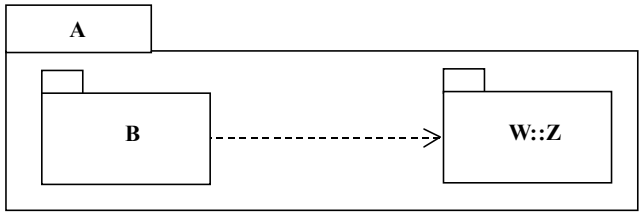




图 4-55　包之间的依赖示例

和类一样，包也有可见性，利用可见性控制外部包对包中内容的存取方式。UML 中对包定义了四种可见性：私有、保护、公有和实现。缺省的可见性为公有。公有可见性允许其它元素存取和使用包中的内容。私有可见性意味着包中的元素只能被拥有或引用该元素的包存取和使用。保护可见性除具有私有可见性的存取要求外，还允许有继承关系的包（一般的包和具体的包）中的具体包存取一般包中的元素。实现可见性与私有可见性很相似，但是有依赖关系的包之间，如果被引用（imported）的包定义为实现可见性，则不允许引用该包中的元素使用被引用包中的元素。换句话说，如果一个包有实现可见性，则不允许其它包引用该包（即无依赖关系）。实现可见性尚无特定的表示符号。

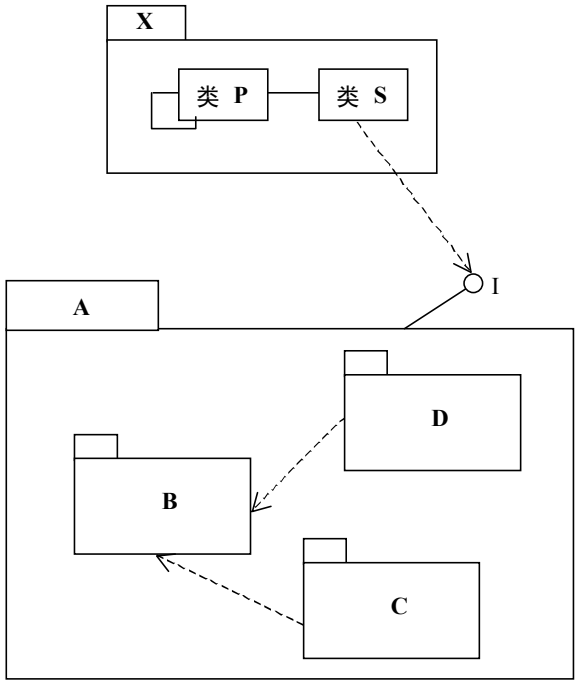


图 4-56　含有接口的包

包也可以有接口，接口与包之间用实线相连。接口通常由包中的一个或多个类实现，如图 4-56 中，包 A 具有接口 I，包 X 中的类 S 依赖 A 中的接口 I（即调用 A 中 I 的实现方法）。

## 4.7　模　　板

模板（templet）是一个尚未完全具体说明的类。模板中提供参数表，利用参数表向模板传递信息可最终形成用户需要的具体类。参数可以是类，也可以是整型、布尔型等基本类型。由于给定不同的参数便可确定不同的类，所以模板能够说明许多类，故又称模板是一个用参数表示的类，如图 4-57 所示。图中 Array 就是一个模板（用长方形表示），右上角的虚框表示一个参数表，参数表中的参数由参数名和参数类型构成，多个参数之间用逗号分隔。如果参数的类型是类，则可以省略该类的类型说明，比如[T：class，A：integer]与[T，A：integer]是一样的。

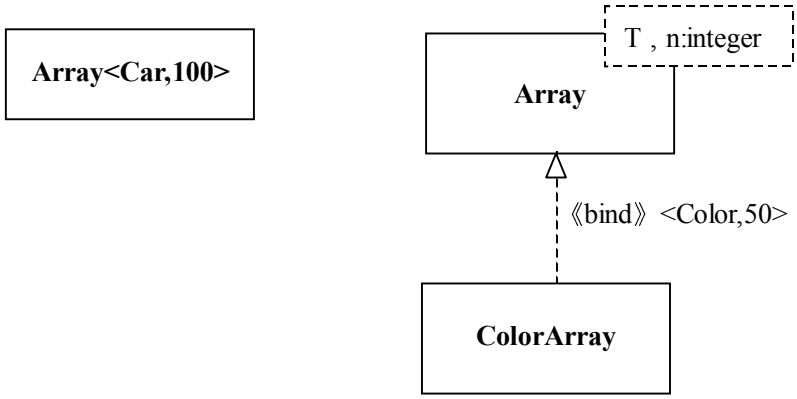



图 4-57　模板示例

模板可以是数组等各种类型。模板的实例是给定模板中参数值之后所得到的类的规格说明。如果模板是数组，那么它的实例可能是一组汽车、一组颜色等等。模板的实例图示为一个与类一样的矩形，但是类的名字中要反映出该类是哪个模板的实例，以及所给定的参数。比如，Array〈Car，100〉。也可以在类图中表示模板的实例与模板之间的关系，它们的关系是精化关系，用版类《bind》表示，《bind》后面紧跟着模板的实在参数。图 4-56 给出了一个模板的二个实例的图示方法。

C++语言提供模板机制，而 JAVA 中没有与此等价的语法成份。

## 4.8　模 型 质 量

通过系统建模可以抽象地表示一个系统。但是，建成的模型是不是最好的？或者说能否在时间上、空间上满足系统要求呢？这个问题的答案建模语言本身不能告诉我们。于是便产生了如何评价一个模型的质量这一重要课题。

设计模型时，所构建的模型要能够抓住问题的本质，准确地表达实际问题是非常重要的。比如说，在构造金融系统时，我们常常为发货单（invoice）建模，而不考虑债务（debt），而在其它系统中，可能只考虑债务，而忽视了发货单。其实，发货单也是债务的一种表示方式，因此建模时应当考虑到这种情况。又如，在十九世纪七十年代到八十年代期间，许多银行依据银行帐户为顾客提供服务。建模时，银行帐户作为一个类或一个实体，而顾客作为银行帐户类的一个属性。这种建模方式带来的第一个问题是银行不能处理许多人拥有一个帐户的情况，第二个问题是一些人员（比如，经商人员）没有固定的地址，所以无法获得银行帐户，也无法从银行得到资助（比如，贷款）。因此把握模型质量时，首先要保证模型抓住各个重要方面，其次，模型要易于通信，具有明确的目标，易于维护，保证一致性和完整性等等。可以为一个事物从不同的角度（不同的目的）建造多个模型，这些模型最终集成为一个模型。

另外，建模时还要考虑一些其它人为因素和社会因素。由于模型是人构建的，并且由人来使用，因此人在商务活动中的干涉和影响不能忽视。比如，政治立场观点、文化背景、社会关系和人的权力等因素都会影响商务活动的运作。如果建模时没有考虑到这些因素，那就不可能发现和捕捉到用户的全部需要，因为，有时某些人陈述的需求并不总是与用户的需求完全一致。特别是，与用户紧密相关的问题（国内政治、社会制度、非官方机构、权力）一定要考虑到建模的过程中。

### 4.8.1 什么是好的模型

能够满足用户需求，抓住了事物的本质且能与之交互的模型，才能算得上“好”。构建一个好的模型需要花费大量时间，并且需要团队合作。团队中的每个成员负责其专长方面的建模，达到发现需求，构建需求规格说明，执行分析和为系统描绘技术设计图的目的。团队中的成员可以由用户、建模专家或对商务或信息系统问题非常在行的专家构成。

能与模型方便地交互是很重要的，也是最基本的。任何工程无论大小都要提供交互功能，使用户能与系统对话，完成信息的双向流通。换而言之，对系统来说，从用户处获得需要提供何种服务的信息，对用户来说，从系统得到被服务的结果（比如文档、消息）。

交互使得利用强大的集成系统的各种功能达到提高生产力、工作效率和工作质量成为可能。没有交互能力的模型是没有任何意义的。

### 4.8.2 模型是否符合目标

任何模型都应具有一个明确目标。因为任何人都是通过识别特定的目标来使用系统的。如果系统目标含混不清，则该系统不易被理解和使用，更无法验证系统的正确性和有效性。

抓住事物的本质建模是保证模型符合目标的基础，同时还易于系统的更改和扩充。一般建模的方法是：提取事物的本质（内核），然后围绕内核建模，最后实现内核的具体表示。比如，金融系统模型中涉及到的文档有发货单、收据、保险单等，如果以这些具体文档建模，当（业务）发生改变时，该系统将无法运行。但是，若把债务抽象为系统的核心，那么，发货单可以看作债务的表示，当系统业务发生变化时，只要改变一下代表债务类（核心类）的类即可。

另外，给模型元素命名也有一定的讲究，这点我们在前面也重复过多次，就是“见名知义”，即名字反映了建模系统中的问题，并且名字不带前后缀。像类、关联、角色、属性和操作这些元素若赋予了与问题相关的名字，则模型理解起来就容易得多。

### 4.8.3 模型的协调性

为同一事物建造的多个模型之间要互相有关联，且能够集成为一个系统。模型协调性的一个方面是集成。集成的含义是把对同一事物从各个不同角度（动态、静态、功能性）描述的模型合成为一个整体，同时合成时不要产生不一致问题。另一个方面是各个模型（建立在不同的抽象层上）之间的关系能够用 UML 中的精化关系表示出来，有了这种精化关系才可能成功地追踪系统的工作状态。简而言之，模型必须在每个抽象层和不同的抽象层之间协调。

### 4.8.4 模型的复杂性

在前面所述的与系统交互，符合系统目标，抓住了事物的本质，具有协调性等工作都完成之后，还要考虑的一个问题那就是模型的复杂性。完成一个复杂模型的调研、验证、有效性证明和维护工作是困难的。通常的作法是先建立一个简单化了的模型，然后再利用模型的协调性逐步细化。如果要建模的问题相当复杂，则可以把该问题分成若干个子问题，分别为子问题建模，每个子问题构成原模型中的一个包（package），以降低建模的难度。

## 4.9 小　　结

所谓建模工作就是利用模型表达我们正在研究的事物的详细情况，要建造一个好的模型就必须抓住问题本质，抓住所研究的对象。在面向对象的设计方法中，利用对象体现并表达事物的本质和关系。所谓对象，简单地说就是我们可以谈论和控制的事，对象存在于现实世界中，它是我们对真实世界的认识和理解。对象是系统的一个组成部分。具有相同行为和特征的对象抽象描述为类。或者反过来说，对象是类的实例。类和对象是讨论系统时的基本元素。

建造系统模型需要使用建模语言，建模语言为创建模型工作提供基本的语法和语义上的支持，比如 UML 语言。建模语言可用于构造模型，但不能保证构建的模型是“好”的模型，因此，考察模型质量是很重要的步骤。如果所有的模型具有明确的清晰的构建思想，抓住了被建模对象的本质，这样的模型才是可靠的。从各个不同层面为系统构建的所有模型必须易于交互，易于验证（有效性和正确性）并易于维护。

UML 语言支持三种建模方式：静态、动态和功能性建模。类图属于静态建模。类图由类和类与类之间的关系构成。类之间的关系可以有关联、通用化、依赖和精化。关联关系是类之间的连接，同时也是该类的对象之间的连接。通用化描述的是通用元素和具体元素之间的继承关系，具体元素可以包含通用元素中没有的信息，可以使用通用元素实例的场合，也可以使用具体元素的实例。一个元素依赖于另一个元素时，则构成了依赖关系，在依赖关系中一个元素是独立的，另一个元素是依赖的。独立元素的变化将影响到依赖元

素。对某一事物在不同抽象层上的描述之间具有精化关系。