第5章

类　图

类图用来描述系统内各种实体的类型及不同的实体之间是如何彼此关联的，显示系统的内部静态结构，因此类图的描述对于系统的整个生命周期都是有效的。如果说用例图是系统的“面子”，那么类图就是系统的“里子”。类图不仅包含了系统定义的各种类，还包含了各种关系，如关联、泛化和依赖等。类图大部分涉及对系统的词汇建模、对协作建模或对模式建模。作为面向对象系统的建模中最常见的图，类图是组件图与部署图的基础，它不仅对结构模型的可视化、详述和文档化很重要，而且对通过正向与逆向工程构造可执行的系统也很重要。

本章学习目标

- 掌握类图中所包含元素的语义及表示法；
- 理解并掌握类图中的关系（关联关系、泛化关系、依赖关系和实现关系）；
- 了解类的高级概念（抽象类、模板类、关联类和分析类）；
- 了解类图建模技术（对系统的词汇建模、对简单协作建模和对逻辑数据库模式建模）；
- 了解对类图进行正向工程和逆向工程需要遵循的策略；
- 了解面向对象的设计原则。

5.1　什么是类图

类图（class diagram）是显示一组类、接口、协作以及它们之间关系的图。一个类图主要通过系统中的类及各个类之间的关系来描述系统的静态结构。

类图与数据模型有许多相似之处，区别就在于类不仅描述了系统内部信息的结构，也包含了系统的内部行为，系统通过自身行为与外部事物进行交互。

类图主要包含七种元素：类、接口、协作、依赖关系、泛化关系、实现关系和关联关系。类图中还可以含有包或子系统，用来把模型元素聚集成更大的组块。与其他 UML 图类似，类图同样可以创建约束和注释等。图 5-1 显示了一个类图，读者可以在学习完本章全部内容之后再来尝试阅读这个类图所描述的情境。

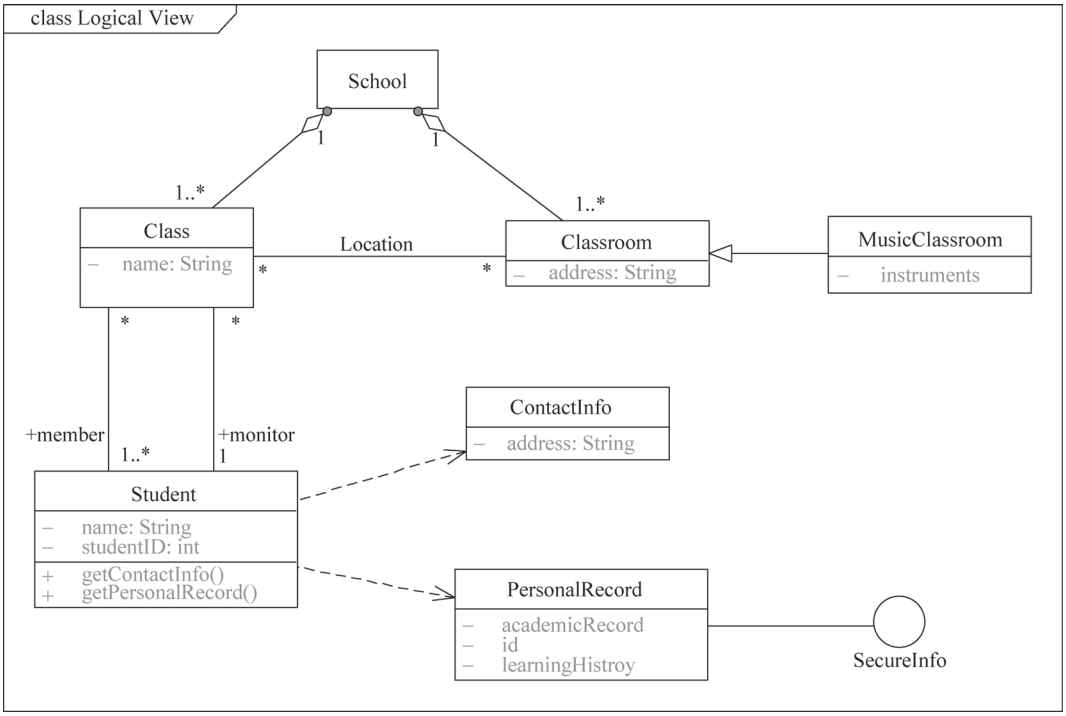


图 5-1 类图

5.2 类图的组成元素

本节将重点介绍类图的组成元素：类、接口以及类图中的四种关系。

5.2.1 类

类(class)是一组拥有相同的属性、方法(操作)、关系和行为的对象描述符。一个类代表了被建模系统中的一个概念。根据模型种类的不同，此概念可能是现实世界中的(对于分析模型)，也可能是包括算法和计算机实现的概念(对于设计模型)。类是面向对象系统组织结构的核心。

类定义了一组有着状态与行为的对象。类的状态由属性和关联来描述，个体行为由操作来描述，对象的生命周期则由附加给类的状态机来描述。

在 UML 中，类表达成一个有三个分隔区的矩形。其中顶端显示类名(name)，中间显示类的属性(attribute)，尾端显示类的操作(operation)，如图 5-2 所示。其中，可选择显示属性和操作的可见性、属性类型、属性初始值、操作的参数列表和操作的返回值等信息。此外，也可以选择隐藏类的属性或操作部分，隐藏了这两部分的类简化为一个只显示类名的矩形，如图 5-3 所示。

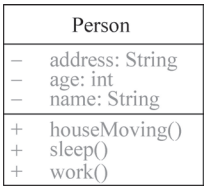


图 5-2 类

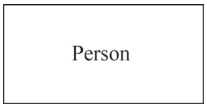


图 5-3 隐藏属性和操作的类

1. 类名

每个类都必须有一个区别于其他类的名称。类名是一个文本串,在实际应用中,类名应该来自系统的问题域,选择从系统的词汇表中提取出来的名词或名词短语,明确而无歧义,便于理解交流。

类名有两种表示方法:使用单独的名称叫作简单名(simple name),如图 5-2 中的 Person;在类名前边加上包的名称,如 java::awt::Rectangle,叫作路径名(path name),表示 Rectangle 类属于 awt 包,而 awt 包又属于 java 包。

按照一般约定,类名一般采用 UpperCamelCase 格式,即以大写字母开头,大小写混合,每个单词首字母大写,避免使用特殊符号。

注意:关于类的路径名,请参考第 7 章有关包的内容。

2. 属性

属性是已被命名的类的特性,它描述了该特性的实例可以取值的范围。类可以有任意数量的属性,也可以没有任何属性。属性描述了类的所有对象所共有的一些特性。例如,每一面墙都有高度、宽度和厚度三个属性。因此,一个属性是对类的一个对象可能包含的一种数据或状态的抽象。在一个给定的时刻,类的一个对象将对该类属性的每个属性具有特定值。

在 UML 中,描述一个属性的语法格式如下:

$$可见性_{opt}\ 属性名\lfloor : 类型 \rfloor_{opt}\ 多重性_{opt}\ \lfloor = 初始值 \rfloor_{opt}\ \lfloor \{特性\} \rfloor_{opt}$$

注意:下标 opt 在这里表示“可选”,即可以省略下标前的项。括号“⌊⌋”表示括号内部的短语是一个整体。本书在下文中说明语法格式时同样会用到这些符号,请读者注意。

属性名是属性的标识符。在描述属性时,属性名是必需的,其他部分可选。按照一般约定,属性名采用 lowerCamelCase 格式,即以小写字母开头,非首单词的首字母大写。用下画线标识的属性名,说明该属性是静态(static)属性,即该类的所有对象之间共享该属性。

可见性描述了该属性在哪些范围内可以被使用。属性的可见性有公有、私有和保护三种,如表 5-1 所示。例如,-attr 就表示一个私有属性 attr。

表 5-1 属性可见性

可 见 性	英文限定符	UML 标准图示	说 明
公有	public	+	其他类可以访问
私有	private	−	只对本类可见,不能被其他类访问
保护	protected	#	对本类及其派生类可见

类型即属性的数据类型，可以是系统固有的类型，如整型、字符型等，也可以是用户自定义的类型。属性的类型决定了该属性的所有可能取值的集合。例如，length：double 即表示一个 double 类型的属性 length。对于用于生成代码的类图，要求类的属性类型必须限制在由编程语言提供的类型或包含于系统中实现的模型类型之中。

属性的多重性表示为一个包含于方括号中的数字表达式，位于类型名后，相当于编程语言中的数组概念。例如，nums：int［10］表示此属性是一个大小为 10 的 int 数组。当然，如果多重性为 1，则可以省略。

初始值作为创建该类对象时这个属性的默认值。例如，num：int＝3 就表示了 int 类型的 num 属性的初始值是 3。设定初始值有两个好处，即保护系统完整性，防止漏掉取值或被非法值破坏系统完整性，以及为用户提供易用性。

特性即对属性性质的约束，UML 定义了三种可以用于属性的特性：可变（changeable）表示属性可以随便修改，没有约束；只增（addOnly）表示该属性修改时可以增加附加值，但不允许对值进行消除或进行减的改变；冻结（frozen）表示在初始化对象后，就不允许改变属性值，对应于 C++中的常量（const）。除非另行指定，否则属性总是可变（changeable）的。例如，PI：double＝3.1415926{frozen}就表示一个不可修改的属性 PI。

3. 操作

操作是一个可以由类的对象请求以影响其行为的服务的实现，也即是对一个对象所做的事情的抽象，并且由这个类的所有对象共享。操作是类的行为特征或动态特征。一个类可以有任意数量的操作，也可以没有操作。调用对象的操作会改变该对象的数据或状态或者为服务的请求者以返回值为承载提供某些信息。

UML 对操作和方法做了区分。操作详述了一个可以由类的任何一个对象请求以影响行为的服务；方法是操作的实现。类的每个非抽象操作必须有一个方法，这个方法的主体是一个可执行的算法（一般用某种编程语言或结构化文本描述）。在一个继承网格结构中，对于同一个操作可能有很多方法，并在运行时多态地选择层次结构中的哪一个方法被调用。

在 UML 中，描述一个操作的语法格式为：

$$\text{可见性}_{opt}\ \text{操作名}\ \lfloor(\text{参数列表})\rfloor_{opt}\ \lfloor:\text{返回类型}\rfloor_{opt}\ \lfloor\{\text{特性}\}\rfloor_{opt}$$

操作名是操作的标识符。在描述操作时，操作名是必需的，其他部分可选。在实际建模中，操作名一般是用来描述该操作行为的动词或动词短语，命名规则与属性相同。同样地，用下画线标识的操作名，说明该操作是静态操作，即外部只需要通过类就可以调用该操作，不需要事先生成对象。而操作名是斜体则表示操作是抽象的。

可见性同样描述该操作在哪些范围内可以使用，与属性的可见性相同。例如，＋oper()就表示此操作是一个公有操作。

参数列表是一些按照顺序排列的属性，定义了操作的输入。参数列表的表示方式与 C++、Java 等编程语言相同，可以有零到多个参数，多个参数之间以逗号隔开。参数的定义方式使用“［方向］参数名：类型［＝默认值］”的方式，方向可以取 in（输入参数，不能对其进行修改）、out（输出参数，为了与调用者通信可以对其进行修改）和 input（输入参数，可以对其进行修改）三个可选值。参数可以具有默认值，这意味着如果操作的调用者没有提供某个具

有默认值的参数的值,该参数将使用指定的默认值。例如,oper(out arg1:int, arg2:double=3.2)表示操作 oper 有两个参数,其中第一个参数是 int 类型的输出参数,第二个参数类型为 double,默认值 3.2。

返回类型即回送调用对象消息的类型。无返回值时,一般的编程语言会添加 void 关键字表示无返回值。例如,oper():String 表示该操作的返回类型是 String 类型。

特性是对操作性质的约束说明。在 UML 中,定义了以下几种可用于操作的特性。

- 叶子(leaf):代表操作不是多态的,即不能被重写,对应于 C++中的非虚函数。
- 查询(isQuery):代表操作的执行不会改变系统的状态。换句话说,这样的操作是完全没有副作用的纯函数,对应于 C++的函数的 const 限定符。
- 顺序(sequential):调用者必须协调好外部的对象,以保证在一个对象中一次仅有一个流。在多控制流的情况下,不能保证对象的语义和完整性。
- 监护(guarded):在多控制流的情况下,通过将对象的各监护操作的所有调用进行顺序化来保证对象的语义和完整性。其效果是一次只能调用对象的一个操作,这又回到了顺序语义。
- 并发(concurrent):在多控制流的情况下,通过把操作作为原子来保证对象的语义和完整性。对任何并发操作,来自并发控制流的多个调用可能同时作用于一个对象,而且所有操作都可以用正确的语义并发运行。并发操作必须设计为:在对同一个对象同时进行顺序的或监护的操作的情况下,它们仍能正确地执行。

以上的顺序、监护与并发三个特性表达的是操作的并发语义,是一些仅与主动对象、进程或线程的存在有关的特性。

4. 职责

职责(responsibility)是类的契约或责任。当创建一个类时,就声明了这个类的所有对象具有相同种类的状态和相同种类的行为。在较高的抽象层次上,这些相应的属性和操作正是要完成类的职责的特征。

类可以有任意数目的职责,当精化模型时,要把这些职责转换成能很好地完成这些职责的一组属性和操作。类的职责是自由形式的文本,在非正式的类图中,可以将职责列在类图操作下的另一分割栏中,如图 5-4 所示。

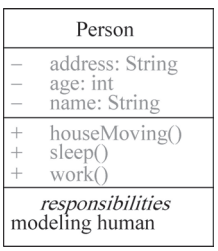


图 5-4 职责

5.2.2 接口

接口(interface)是一个被命名的操作集合,用于描述类或组件的一个服务。接口不同于任何类或类型,它不描述任何结构,因此不包含任何属性;也不描述任何实现,因此不包含任何实现操作的方法。像类一样,接口可以有一些操作。一个类可以支持多个接口,多个接口可以是互斥的,也可以是重叠的。接口中没有对自身内部结构的描述,因此,接口没有私有特性,它的所有内容都是公共的。

接口代表了一份契约，实现该接口的类元必须履行它。例如，当设计一个窗口界面时，可能窗口中有一些控件是允许用户拖动的，这时可以定义接口 draggable，所有可拖动的控件都需要实现这一接口。

与类相似，接口可以有泛化。子接口包含其祖先的全部内容，并且可以添加额外的内容。与类不同的是，接口没有直接实例。也就是说，不存在属于某个接口的对象。

在 UML 中，接口由一个带名称的小圆圈表示，如图 5-5 所示。接口名与类名相似，同样存在简单名和路径名两种表示法。为了显示接口中的操作，接口可以表示为带有<< interface >>构造型的类，如图 5-6 所示。

图 5-5 接口的圆圈表示法

图 5-6 接口的“构造型的类”表示法

5.2.3 类图中的关系

在类图中，很少有类是独立为系统发挥作用的，大部分的类以某些方式彼此协作进行工作。因此，在进行系统建模时，不仅要抽象出形成系统词汇的事物，还必须对这些事物之间的关系进行建模。

类图中涉及了 UML 中最常用的四种关系，即关联关系、泛化关系、依赖关系和实现关系。

1. 关联关系

关联关系是两个或多个类元之间的关系，它描述了这些类元的实例间的连接。关联的实例被称为链(link)，每个链由一组有序或无序的对象组成；也就是说，如果一些对象之间存在链，那么这些对象所属的类之间必定存在关联关系。关联关系靠近被关联元素的部分称为关联端，关联的大部分描述都包含在一组关联端的列表里，每个端用来描述关联中类的对象的参与。关联将一个系统模型组织在一起，如果没有关联，便只有一个由孤立的类组成的集合。

最普通也是最常用的关联关系是二元关联，二元关联即有两个关联端的关联关系。图 5-7 显示了一个二元关联。二元关联使用一条连接两个类边框的实线段表示(通常是直线段，但也允许使用弧线或其他曲线)，这条实线段称为关联路径。对于一个二元关联，除关联路径外，其他描述该关系的内容都是可选的。

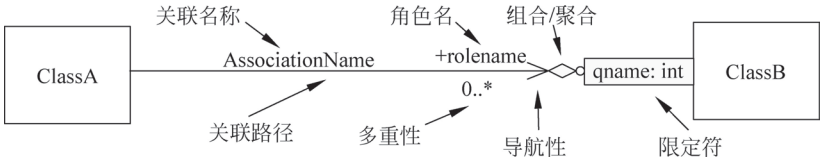


图 5-7 二元关联

特别地，一个类与自身的关联称为自关联。自关联不代表类的实例与其自身相关，而是类的实例与其他实例之间有关。自关联同样拥有两个关联端，因此可以看作二元关联的一种特例。如图 5-8 所示，Student 类存在一个自关联，表示某些学生是其他学生的班长。描述自关联的内容的表示法与二元关联相同。

当三个或以上的类之间存在关联关系时，便无法使用二元关联的表示法了，此时称之为 N 元关联。N 元关联表示为一个菱形，从菱形向外引出通向各个参与类的路径。图 5-9 显示了一个三元关联关系，表达了学生在学期内选修某个教师的课程的关系。

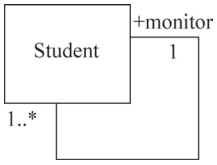


图 5-8　自关联

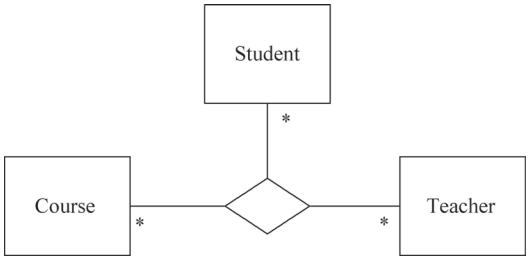


图 5-9　三元关联

建议：*N 元关联在理解上不如二元关联直观，初学者也容易误用，并且绝大部分 N 元关联都可以被重新建模成为多个二元关联。因此不建议在建模中使用 N 元关联，以免引起错误。*

除连接类元素的关联路径外，还有很多可选内容可以对关联关系的语义进行进一步的细化和精确化，下面将对这些内容进行详细介绍。

1）关联名称

关联可以有一个名称(关联名称并不是必需的)，关联名称放在关联路径的旁边，但远离关联端，以免引起视觉上的混淆。关联关系上的构造型同样是将构造型名称放在双尖括号(<< >>)内表示，可以放在关联名称前或替代关联名称。

2）角色

角色名放在靠近关联端的部分，表示该关联端连接的类在这一关联关系中担任的角色。具体来说，角色就是关联关系中一个类对另一个类所表现的职责。比如，Person 类和 Room 类建立了一个用来表示人在室内工作所产生的关联，那么 Room 类就可以用 Office(办公室)作为角色名，Person 类可以用 Worker(工作者)作为角色名，如图 5-10 所示。另外，角色名也可使用可见性修饰符号＋、＃和－来表示角色的可见性。

图 5-10　角色名

3）多重性

多重性(multiplicity)同样放在靠近关联端的部分，表示在关联关系中源端的一个对象可以与目标类的多少个对象之间有关联。在 UML 中，多重性的格式为“min..max”，其中，min 和 max 分别表示该端最少和最多可以有多少对象与另一端关联。常用的多重性有 0，

1,0..1(0 或 1)、0.. *(0 或更多)、1.. *(1 或更多)、*(0 或更多)等。图 5-11 显示了 Student 类与 School 类的关联关系的多重性,即一个学校可以有一个或更多个学生,而一个学生可能在 0 所或更多所学校中学习。

图 5-11 多重性

4) 导航性

导航性(navigation)是一个布尔值,用来说明运行时刻是否可能穿越一个关联。对于二元关联,当对一个关联端(目标端)设置了导航性就意味着可以从另一端(源端)指定类型的一个值得到目标端的一个或一组值(取决于关联端的多重性)。对于二元关联,只有一个关联端上具有导航性的关联关系称为单向关联(unidirectional association),通过在关联路径的一侧添加箭头来表示;在两个关联端上都具有导航性的关联关系称为双向关联(bidirectional association),关联路径上不加箭头。使用导航性可以降低类间的耦合度,这也是好的面向对象分析与设计的目标之一。图 5-12 显示了一种导航性的使用场景,这代表一个订单可以获取到该订单的一份产品列表,但一个产品却无法获取到哪些订单包括了该产品。

图 5-12 导航性

5) 限定符

限定符(qualifier)是二元关联上的属性组成的列表的插槽,其中的属性值用来从整个对象集合里选择一个唯一的关联对象或者关联对象的集合。存在限定符的关联称为限定关联(qualified association)。一个对象连同一个限定符一起,决定一个唯一的关联对象或对象的子集(比较少见)。被限定符选中的对象称为目标对象。限定符总是作用于目标方向上多重性为“多”的关联中。在最简单也是最常见的情况下,每个限定符只从目标关联对象集合中选择一个对象,这样就将关联对象方向上的多重性从 0.. * 降到了 0..1。也就是说,一个受限定对象和一个限定值映射到一个唯一的关联对象。举个简单的例子,数组就可以被建模成一个受限定的关联。数组是受限定对象,数组下标就是限定符,而数组元素就是目标对象,如图 5-13 所示。

图 5-13 限定符

6) 关联的约束

两个关联关系之间也可以有约束,约束用连接两个关联关系路径的虚线表示,并带有花括号包围的约束字符串。如果约束有方向性,可在通过虚线上添加箭头来表示。图 5-14 显

示了两个关联关系之间存在的异或约束,表示个人与公司不允许拥有同一个银行账户,即同一个 Account 类对象不能同时与 Corporation 类对象和 Person 类对象都有关联。

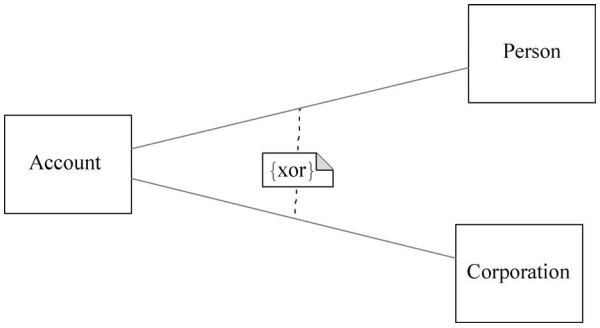


图 5-14　二元约束

7) 特殊的关联——聚合与组合

在一个简单的二元关联中,两个类的"地位"是同等的,即在概念上是同级别的。有时候我们需要对"整体-部分"的关系建模,即一个描述整体的对象由一些描述部分的对象组成,这种关系称为聚合(aggregation)。聚合关系是一种特殊形式的关联关系,用来表示一个"整体-部分"的关系。需要注意的是,聚合关系没有改变整体与部分之间整个关联的导航含义,也与整体和部分的生命周期无关。也就是说,在聚合关系中,"部分"可以独立于"整体"存在。在 UML 中,通过在关联路径上靠近表示"整体"的类的一端上使用一个小空心菱形表示。如图 5-15 所示,Classroom(教室)类与 Desk(课桌)类之间构成一个聚合关系,即教室中有许多课桌,当教室对象不存在时,课桌同样可以作为其他用途,二者是独立存在的。

另外,聚合关系的实例应该具有传递性与反对称性。

- 传递性:如果对象 A 与对象 B 之间有一条链,B 与 C 之间有一条链,并且两条链是同一个聚合关系的实例,那么 A 与 C 之间也存在一条链。
- 反对称性:聚合关系的链不能将对象连接到自己,即聚合关系的实例不能成环。需要注意的是,这里说的是聚合关系的实例,而非聚合关系本身。事实上,聚合关系可以成环。

组合关系(composition)描述的也是整体与部分的关系,它是一种更强形式的聚合关系,又被称为强聚合。与聚合关系的区别在于,在组合关系中的部分要完全依赖于整体。这种依赖性主要表现在两个方面:部分对象在某一特定时刻只能属于一个组合(整体)对象;组合对象与部分对象具有重合的生命周期,组合对象销毁的时候,所有从属部分必须同时销毁。如图 5-16 所示,Window(窗口)类与 Frame(框架)类之间构成组合关系,Frame 必须附加在 Window 中存在,当一个 Window 被删除时,其中的 Frame 部分也必须被删除。

图 5-15　聚合关系

图 5-16　组合关系

在实际建模过程中，使用聚合还是组合，需要根据应用场景和需求分析描述的上下文来灵活确定。其实，聚合和组合对于绝大多数面向对象的编程语言来说，并没有实质的区别，因此不必过于执着于此。

8）派生关联

派生关联(derived association)属于一种派生元素(derived element)。它不增加语义信息，只是一种可以由两个或两个以上的基础关联推算出来的虚拟关联。如图 5-17 所示，WorksForCompany 关联可以根据另外两个关联推算得出。该关联关系表示如果一个人为某个部门工作，而此部门又属于某公司，则可以得出这个人是在为此公司工作，表示为派生关联关系 Person. employer=Person. department. employer。

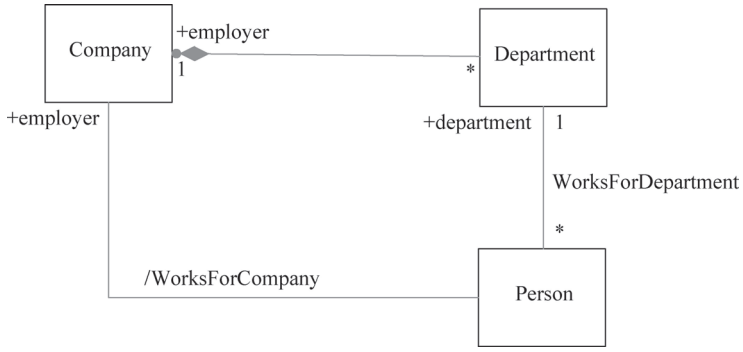


图 5-17 派生关联

建议：由于派生关联在模型中不增加任何语义关系，大量的派生关联在视觉上容易导致混淆从而降低可读性，因此不要在建模过程中大量使用。建议只有对于在建模过程中频繁使用的少数派生关联才在模型上显式画出，并最好给予注释说明以避免混淆。

2. 泛化关系

泛化关系定义为一个较普通的元素与一个较特殊的元素之间的类元关系。其中描述一般的元素称为父，描述特殊的元素称为子。对于类而言，即为超类(或称父类)与子类之间的关系。例如，猫和狗都是宠物的一种，因此对其建模时，宠物类作为父类，猫和狗作为子类。另外，接口之间也可以存在泛化关系，即子接口与父接口的关系。

泛化关系描述了一种“is-a-kind-of”(是……的一种)关系，它的使用有利于简化有些类的描述，可以不必重复添加大量相同的属性和操作等特性，而是通过泛化对应的继承机制使子类共享父类的属性和操作。继承机制减小了模型的规模，同时也防止了模型的更新所导致的定义不一致的意外。需要注意的是，泛化与继承是类级别上的概念，因此不能说对象之间存在泛化关系。

在 UML 中，泛化关系通过一个由子类指向父类的空心三角形箭头表示，如图 5-18 所示。图中 Tiger 类和 Bird 类继承了 Animal 类的属性和操作，还添加了属于自己的某些属性和操作。当存在多个泛化关系时，可以表示为一个树结构，每个分支指向一个子类。

泛化是一种传递的、反对称的关系。泛化的传递性表现在一个类的子类同样继承了这个类的特性。在父方向上经过了一个或几个泛化的元素被称为祖先，在子方向上则被称为后代。泛化的反对称性表现在泛化关系不能成环，即一个类不可能是自己的祖先和自己的后代。

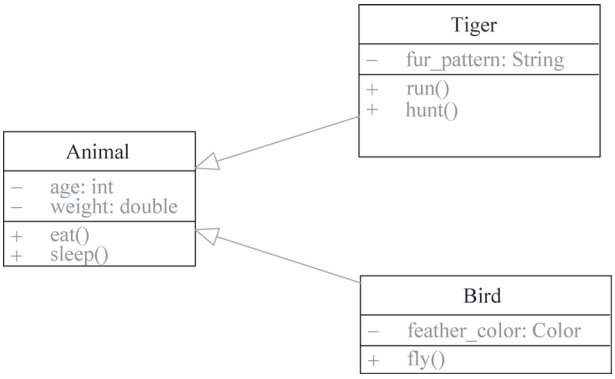


图 5-18 泛化关系

泛化关系有两种情况。在最简单的情况下,每个类最多能拥有一个父类,这称为单继承。而在更复杂的情况中,子类可以有多个父类并继承了所有父类的结构、行为和约束。这被称为多重继承(或多重泛化),其表示法如图 5-19 所示。

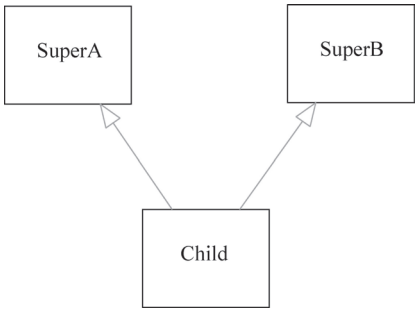


图 5-19 多重继承

建议:不同的编程语言对于多重继承的支持性各有不同。例如,C++支持多重继承,而Java、C#则不支持。并且,使用多重继承容易出现子类中某些属性或操作的二义性问题,因此不建议使用多重继承。读者可以参考 Java 或 C#中的解决方法,即子类继承于唯一的父类,并可以实现多个接口,来达到多重继承的效果。

3. 依赖关系

依赖关系表示的是两个元素之间语义上的连接关系。对于两个元素 X 和 Y,如果元素 X 的变化会引起对另一个元素 Y 的变化,则称元素 Y 依赖于 X。其中,X 被称为提供者,Y 被称为客户。依赖关系使用一个指向提供者的虚线箭头来表示,如图 5-20 所示。

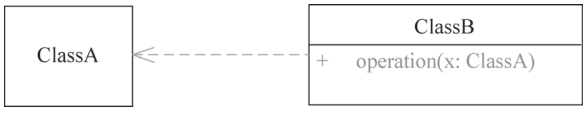


图 5-20 依赖关系

对于类图而言,主要有以下需要使用依赖的情况:

- 客户类向提供者类发送消息。

- 提供者类是客户类的属性类型。
- 提供者类是客户类操作的参数类型。

建议：由于依赖关系语义的宽泛性，在类图中要标记出所有的依赖关系是一件费时费力的事情，并且会降低模型的可读性。因此建议在类图中尽量不使用依赖关系。

4. 实现关系

实现关系用来表示规格说明与实现之间的关系。在类图中，实现关系主要用于接口与实现该接口的类之间。一个类可以实现多个接口，一个接口也可以被多个类实现。

在 UML 中，实现关系表示为一条指向提供规格说明的元素的虚线＋三角形箭头，如图 5-21 所示。图中表示了 Wall 类实现了 Measurable 接口，即在 Wall 类中要实现接口中三个操作的声明。当接口元素使用小圆圈的形式表示时，实现关系也可以被简化成一条简单的实线，如图 5-22 所示。

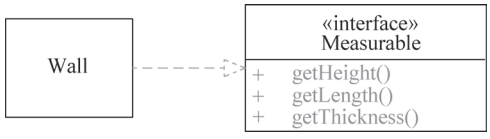


图 5-21 实现关系

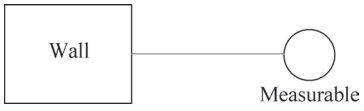


图 5-22 实现关系的简化表示

5.2.4 类的高级概念

1. 抽象类

抽象类(abstract class)即不可实例化的类，也就是说，抽象类没有直接的实例。当某些类有一些共同的方法或属性时，可以定义一个抽象类来抽取这些共性，然后将包含这些共性方法和属性的具体类作为该抽象类的继承。

操作也有类似的特性。通常操作是多态的，这意味着，在类的层次中，可以用相同的特征标记在层次的不同位置上描述操作。子类的操作重写父类的操作的行为。当运行中要发送消息时，在这个层次中被调用的操作就被多态地选择，即在运行时按照对象的类型决定匹配的操作。

在 UML 中，抽象类和抽象操作的表示方法是将类名和操作名用斜体表示。如图 5-23 所示，类 Drawing 和 Shape 是抽象类，Drawing 类下的 draw()操作和 Shape 类下的 getArea()操作是抽象操作。

2. 模板类

模板(template)又称为参数化元素(parameterized element)，是对一类带有一个或者多个未绑定的形式参数的元素的描述。C++中的模板概念和 Java 中的泛型(generic)概念都表达了这种元素。模板参数化可以应用于类元(如类和协作)、包和操作，模板应用在类上是最为常用的，称为模板类。

模板类可以解决这样的问题：根据参数来定义类，而不用说明属性和操作参数及返回值的具体类型，使用时通过实际值代替参数即可创建新的类，这样就可以避免建立大量功能相似的类。在 UML 中使用带有<< bind >>构造型的依赖关系表示从模板类创建新的类。图 5-24 中的 Array 类就是带有 size 和 T 两个参数的模板类，并通过依赖关系创建了三个新的类。

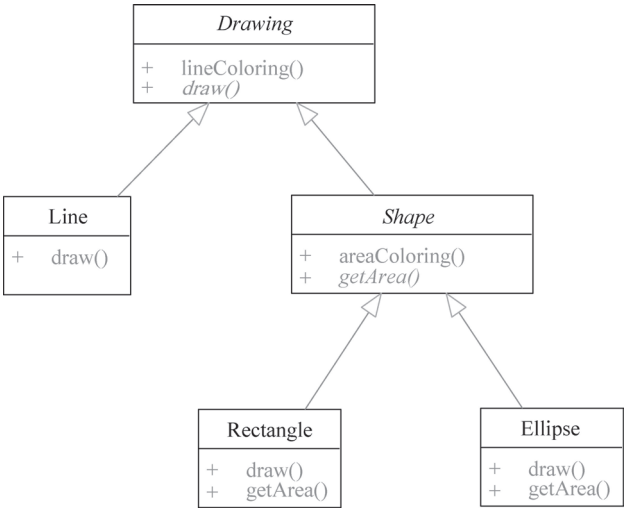


图 5-23 抽象类

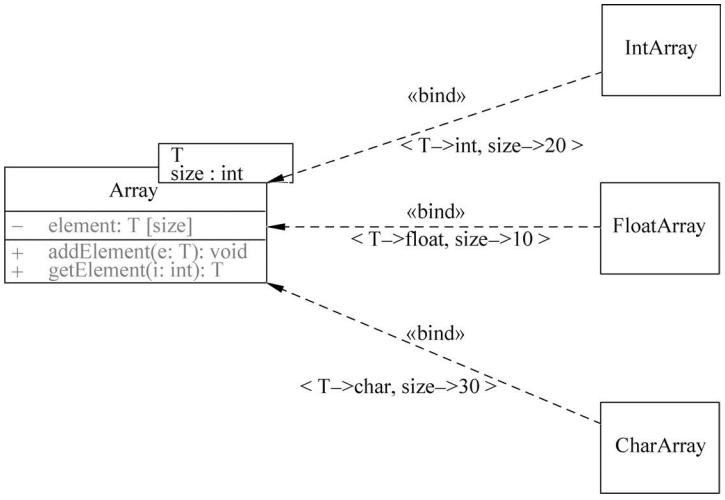


图 5-24 模板类

3. 关联类

前面介绍了关联关系，关联关系本身也可以有特性。比如对于一个描述公司和员工的雇佣关系中，存在着薪资和雇佣合同的开始及结束时间等属性，由于雇佣关系是多对多的，所以这些属性既不属于公司类，也不属于员工类，而属于雇佣关系。这时可以创建一个具有类的特性的关联关系，UML 中称为关联类(association class)。关联类具有关联和类二者

的特性，它既可以关联类元素，也可以拥有属性和操作。关联类在 UML 中被表示为一个类符号，并通过一条虚线连接到关联路径。对本段开始时描述的场景设计类图即如图 5-25 所示。对于 N 元关联（见图 5-9），关联类通过虚线连接到菱形上。虚线上没有修饰内容。

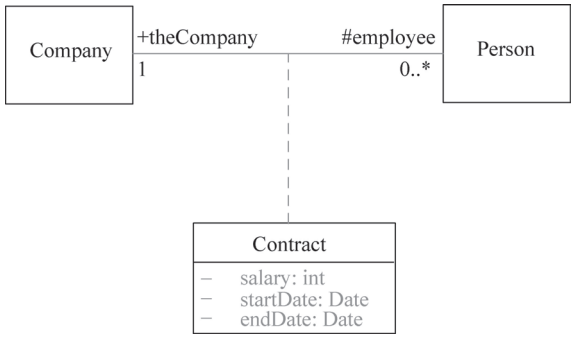


图 5-25 关联类

4. 分析类

分析类是一个主要用于开发过程中的概念，用来获取系统中主要的“职责簇”，代表系统的原型类，是带有某些构造型的类元素，包括边界类（boundary）、控制类（control）和实体类（entity）三种，表示法如表 5-2 所示。分析类在从业务需求向系统设计的转化过程中起到重要的作用，它们在高层次抽象出系统实现业务需求的原型，业务需求通过分析类逻辑化，被计算机所理解。

表 5-2 分析类

分析类类型	边界类	控制类	实体类
Icon 形式	NewClass	NewClass	NewClass
Label 形式	<<boundary>> NewClass	<<control>> NewClass	<<entity>> NewClass

边界类是一种用于对系统外部环境与其内部运作之间的交互进行建模的类。这种交互包括转换事件，并记录系统表示方式中的变更。一般来说，边界类的实例可以是窗口、通信协议、外部设备接口、传感器、终端等。总之，在两个有交互的关键对象之间都应当考虑建立边界类。在建模过程中，边界类有下列几种常用场景：

- 参与者与用例之间应当建立边界类。用例可以提供给参与者完成业务目标的操作只能通过边界类表现出来。例如，参与者通过一组网页、一系列窗口、一个命令行终端等才能使用用例的功能。

- 用例与用例之间如果有交互,应当为其建立边界类。如果一个用例需要访问另一个用例,直接访问用例内部对象不是一个良好的做法,因为这将导致紧耦合的发生。使用边界类可以使这种直接访问变为间接访问。在实现时,这种边界类可以表现为一组 API、一组 JMS 或一组代理类。
- 如果用例与系统边界之外的第三方系统等对象有交互,应当为其建立边界类,以起到中介的作用。

控制类是一种对一个或多个用例所特有的控制行为进行建模的类。控制类的实例称为控制对象,用来控制其他对象,体现出应用程序的执行逻辑。在 UML 中,控制类被认为主要起到协调对象的作用,例如,从边界类通过控制类访问实体类,协调两个对象之间的行为。在建模过程中,一般由一个用例拥有一个控制类或者多个用例共享同一个控制类,由控制类向其他类发送消息,进而协调各个类的行为来完成整个用例的功能。

实体类是用于对必须存储的信息和相关行为建模的类。简单来说,实体类就是对来自现实世界的具体事物的抽象。实体类的主要职责是存储和管理系统内部的信息,它也可以有行为,甚至很复杂的行为,但这些行为必须与它所代表的实体对象密切相关。实体类的实例称为实体对象,用于保存和更新一些现象的有关信息。实体类具有的属性和关系一般都是被长期需要的,有时甚至在系统整个生存周期内都需要。

5.3 应用类图建模

5.3.1 类图建模技术

类图用于对系统的静态设计视图建模。这种视图主要用于支持系统的功能需求,即系统要提供给最终用户的服务。当对系统的静态设计视图建模时,通常用以下三种方式使用类图。

1. 对系统的词汇建模

类可以对从试图解决的问题中或从用于解决该问题的技术中得到的抽象进行建模。这样的抽象是系统词汇表中的一部分,它们在整体上的描述对用户和系统开发人员来说都是很重要的。对系统的词汇建模主要需要考虑哪些抽象是系统的一部分,哪些抽象处于系统边界之外,并用类图详述这些抽象。为了对系统进行词汇建模,需做以下工作:

- 识别用户或系统开发人员用于描述问题或解决问题的那些实体。可以使用基于用例分析的技术帮助用户发现这些抽象。
- 对于每个抽象,识别一个职责集。要明确地定义每个类,而且这些职责要在所有的类之间进行很好的均衡。
- 提供为实现每个类的职责所需的属性和操作。

2. 对简单协作建模

类不是单独存在的,而是要和其他类一起协同工作,以便表达出单个类无法表达的语义。因此,除了捕获系统的词汇外,也需要将注意力转移到对词汇中的这些事物协同工作的各

种方式当中来，并用类图描述这些协作。为了对系统进行简单协作建模，需遵循以下策略：

- 识别要建模的机制。一个机制描述了正在建模的部分系统的一些功能和行为，这些功能起因于类、接口以及其他一些事物所组成的群体的相互作用。
- 识别元素及其关系。对于每种机制，分别识别参与协作的类、接口和其他协作，并识别这些事物之间的关系。
- 用脚本排演这些事物。通过这种方法，可发现模型的哪些部分被遗漏以及哪些部分有明显语义错误。
- 把元素和其包含的内容聚集在一起。对于类而言，要做好职责的平均分配，然后逐渐把它们转换成具体的属性和操作。

3. 对逻辑数据库模式建模

实体关系图(E-R 图)是一种用于逻辑数据库设计的通用建模工具，UML 的类图是 E-R 图的超集。传统的 E-R 图只针对数据，类图则更进了一步，也允许对行为建模。所以，类图显示的细节足以用来构造一个数据库。二者的区别在于，类图中的数据只能在系统的生存时间之内存在，而数据库中存储的则是永久数据。

对数据库模式建模，要遵循以下策略：

- 识别模型中那些必须超过应用程序生存时间的类作为永久数据存储。
- 创建一个包含这些类的类图。针对数据库中的特定细节，可以自己定义相关的构造型和标记值组合。
- 对类的结构细节进行细化。主要包括明确属性的细节、之间的关联及其多重性。
- 注意那些增加数据库设计复杂化的公共模式并尽量简化，如循环关联、一对一的关联和 N 元关联等。
- 考虑类的行为。这些行为主要包括对数据存取和数据完整性约束重要的操作。一般情况下，这些业务规则应该被封装在这些永久类的上一层中。

5.3.2 使用类图进行正向工程与逆向工程

扫一扫

视频讲解

建模是重要的，但最终要交付的产品是软件而不是图。因此，要让模型与软件实现相匹配，并使二者保持同步的代价减到最小。

正向工程是通过到实现语言的映射而把模型转换为代码的过程。由于 UML 所描述的模型在语义上要比当前任何的程序设计语言都要丰富，因此使用正向工程将模型转换为代码将导致一些信息丢失。

对类图进行正向工程，要遵循以下策略：

- 确定映射到实现语言的规则。
- 根据所选择的语言，限制某些 UML 的特性。例如，当所选择的语言是 Java 时，要禁止多重继承的使用。
- 使用标记值来帮助实现目标语言的细节。
- 使用工具生成代码。

逆向工程是通过从特定语言的映射而把代码转换为模型的过程。一方面，逆向工程会产生大量的冗余信息，其中一些对模型而言属于细节层次；另一方面，逆向工程是不完整

的,因为在正向工程的过程中已经丢失了一部分模型信息,因此难以在代码中创建一个完整的模型。

对类图进行逆向工程,要遵循以下策略:

- 确定实现语言进行映射的规则。
- 指定要进行逆向工程的代码。期望从一大块代码中逆向生成一个简明的模型是不切实际的。应该选择部分代码,从底部建造模型。
- 使用工具,通过查询模型来创建类图。

人工为模型增加在逆向工程中丢失或隐藏的相关信息。

5.3.3 面向对象的设计原则

对于面向对象设计,要使系统的设计结果能适应系统的需求变化,使软件既灵活又便于维护。在面向对象方法的发展过程中,逐渐形成了几条公认的设计原则。在面向对象设计过程中,要遵循这几条原则。

注意:本部分内容是属于面向对象设计的范畴,本书在这里讲解是为了便于结合类图进行实例的说明,并且提醒读者在设计模型的类图时要能够应用这几条原则从而设计出优秀的类图。

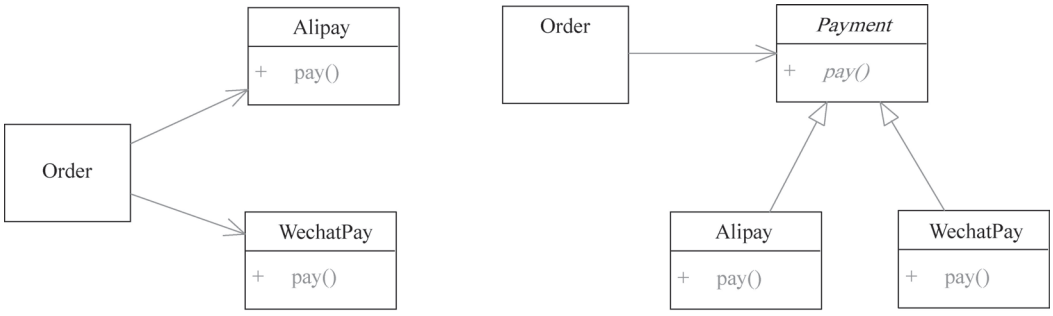
1. 开闭原则

开闭原则(Open-Closed Principle,OCP)是由 Bertrand Meyer 于 1988 年在其著作《面向对象软件构造》中提出的,原文是"Software entities should be open for extension, but closed for modification."翻译过来就是"软件实体应当对扩展开放,对修改关闭。"这个原则关注的是系统内部改变的影响,最大限度地使模块免受其他模块改变带来的影响。

通俗地讲,开闭原则就是软件系统中的各组件,应该能够在不修改现有内容的基础上,引入新功能。其中的"开",指的是对扩展开放,即允许对系统的功能进行扩展;其中的"闭",指的是对原有内容的修改是封闭的,即不应当修改原有的内容。由于系统需求很少是稳定不变的,开放模块的扩展可以降低系统维护的成本。封闭模块的修改可以让用户在系统扩展后可以放心使用原有的模块,而不必担心扩展会修改原有模块的源代码或降低稳定性。

为了遵循开闭原则,对于类图的设计应该尽可能地使用接口或泛化进行封装,并且通过使用多态机制进行调用。接口和泛化的使用可以使操作的定义与实现分离,使得新添加的模块依赖于原有模块的接口。多态的使用可以通过创建父类的间接实例来调用子类的方法,从而避免对调用者的修改。

图 5-26 显示了一个应用开闭原则前后的简单实例。在图 5-26(a)中,Order(订单)类与两个表示支付方式的类存在关联关系。在这种设计下,如果系统需要新增加一种支付方式,不仅要新增加一个类,而且需要改动 Order 类中的代码才能适应新模块的出现。而在图 5-26(b)中,Order 类与抽象类 Payment(支付方式)之间建立了唯一一个关联关系,所有表示支付方式的类通过实现 Payment 类来接入模块,在 Order 类中创建 Payment 类型的间接实例,通过多态完成调用,这样在新增加支付方式时,只需要将新的支付方式的类作为 Payment 的子类来进行扩展即可。



(a) 示例一　　(b) 示例二

图 5-26　开闭原则

2. 里氏替换原则

里氏替换原则(Liskov Substitution Principle，LSP)是由 Barbara Liskov 于 1987 年在 OOPSLA 会议上做的题为“数据抽象与层次”的演讲中首次提出的。其内容是：子类对于父类应该是完全可替换的。具体来说，如果 S 是 T 的子类，则 T 类的对象可以被 S 类的对象所替代，而不会改变该程序的任何理想特性。

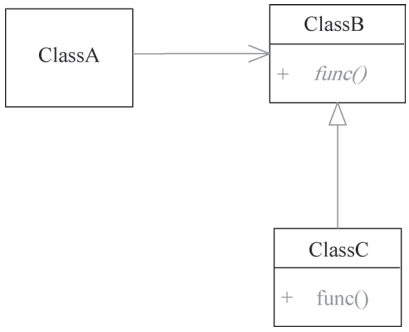


图 5-27　里氏替换原则

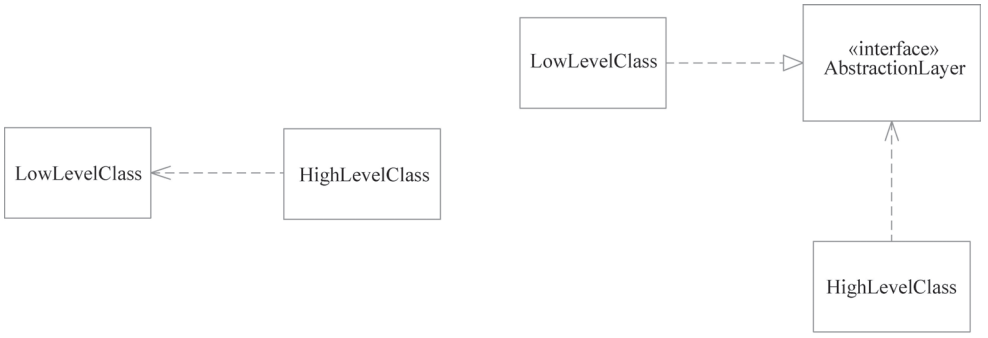
我们都知道，子类的实例是父类的间接实例。根据多态原则，当父类创建一个间接实例并调用操作时，将根据实际类型调用子类的操作实现。如图 5-27 所示，当 ClassA 类中创建一个 ClassB 的间接实例(实际类型为 ClassC)并调用 func()操作时，系统将选择 ClassC 中的操作实现进行调用。因此在设计类时，需要保持 ClassB 与 ClassC 两个类的 func()操作功能上是可替换的，否则会得到违背直觉的结果。

3. 依赖倒置原则

依赖倒置原则(Dependency Inversion Principle，DIP)是由 Robert C. Martin 提出的，其内容包括两条：高层次模块不应该依赖于低层次模块，二者都应该依赖于抽象；抽象不应该依赖于具体，具体应该依赖于抽象。

在传统的设计模式中，高层次的模块直接依赖于低层次的模块，这导致当低层次模块剧烈变动时，需要对上层模块进行大量变动才能使系统稳定运行。为了防止这种问题的出现，可以在高层次模块与低层次模块之间引入一个抽象层。因为高层次模块包含有复杂的逻辑结构而不能依赖于低层次模块，这个新的抽象层不应该根据低层次创建，而是低层次模块要根据抽象层而创建。根据依赖倒置原则，从高层次到低层次之间设计类结构的方式应该是：高层次类→抽象层→低层次类。

在设计时，可以使用接口作为抽象层。图 5-28(a)给出了没有应用依赖倒置原则的设计，即高层次类直接依赖于低层次类。图 5-28(b)则使用接口作为抽象层，让低层次类来实现接口，高层次类依赖接口。



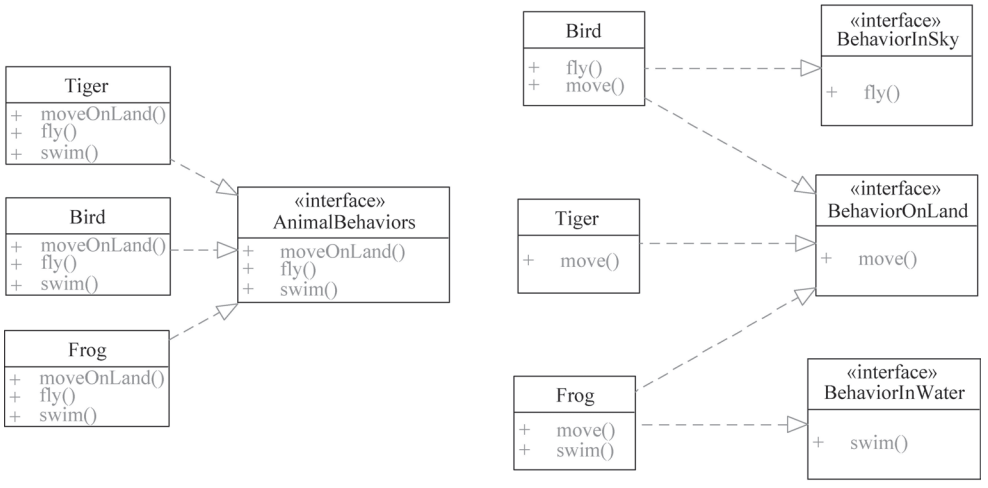
(a) 示例一　　(b) 示例二

图 5-28　依赖倒置原则

4. 接口分离原则

接口分离原则(Interface Segregation Principle,ISP)是由 Robert C. Martin 首次使用并详细阐述的。它阐述了在系统中任何客户类都不应该依赖于它们不使用的接口。这意味着,当系统中需要接入许多个子模块时,相比于只使用一个接口,将其分成许多规模更小的接口是一种更好的选择,其中每一个接口服务于一个子模块。

在图 5-29(a)中,三个类实现了一个共同的接口。可以看到,每个类都不应该拥有该接口中的所有行为,比如,Tiger 类显然不应该有 fly()与 swim()的操作,却必须要实现这两个行为。这样的接口称为臃肿的接口(fat interface)或污染的接口(polluted interface),这可能导致不恰当的操作调用。图 5-29(b)是应用了接口分离原则进行修改之后的设计。新设计将其分解为三个小接口,三个客户类只实现自身具有的行为所属的接口。应用接口分离原则降低了系统的耦合度,从而使系统更容易重构、改变并重新部署。



(a) 示例一　　(b) 示例二

图 5-29　接口分离原则

5. 单一职责原则

单一职责原则(Single Responsibility Principle,SRP)是由 Robert C. Martin 在他的《敏捷软件开发：原则、模式和实践》一书中提出。单一职责原则规定每个类都应该只含有单一的职责,并且该职责要由这个类完全封装起来。Martin 将职责定义为“改变的原因”,因此这个原则也可以被描述为“一个类应该只有一个可以引起它变化的原因”。每个职责都是变化的一个中轴线,如果类有多个职责或职责被封装在了多个类里,就会导致系统的高耦合,当系统发生变化时,这种设计会产生破坏性的后果。

上述内容是面向对象设计的五大原则,根据其首字母,这五个原则也被合称为 SOLID。这些原则不是独立存在的,而是相辅相成的。这些原则的应用可以产生一个灵活的设计,但也需要花费时间和精力去应用并且会增加代码的复杂度。在使用时,我们需要根据系统的规模和是否经常变更需求来适时应用这些原则,从而得出一个更优秀的设计。

扫一扫

视频讲解

5.4 实验：绘制“机票预订系统”的类图

为了使读者更好地理解类图,我们仍然假设了一个具体情境,展示项目分析阶段的类图的主要创建过程。系统的具体情景说明请参考 4.5 节。

1. 确定类元素

根据情境描述,我们应该确定出系统主要可以包括哪些类。这里可以归结出用户、管理员、机场、航班与机票几个实体类,还应该包括有一个系统控制类来控制整个系统。由于分析阶段尚未进行用户界面设计,因此类图中暂时不涉及边界类,需要在设计阶段再对类图进一步完善。将确定好的类添加到类图中,如图 5-30 所示。

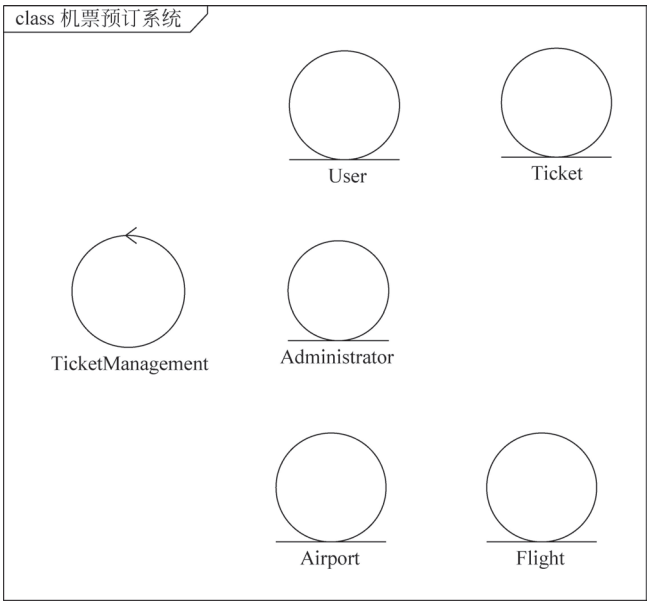


图 5-30 确定类元素

2. 添加类的属性与操作

在确定了系统中包括的类之后，我们需要根据类的职责来确定类的属性与操作。在实际开发过程中，这往往是一个需要多次迭代的过程，即需要多次明确其语义和添加新内容。在最初的分析阶段，只要能大致描述类在整个系统中的作用即可。添加了属性与操作的类图如图 5-31 所示。

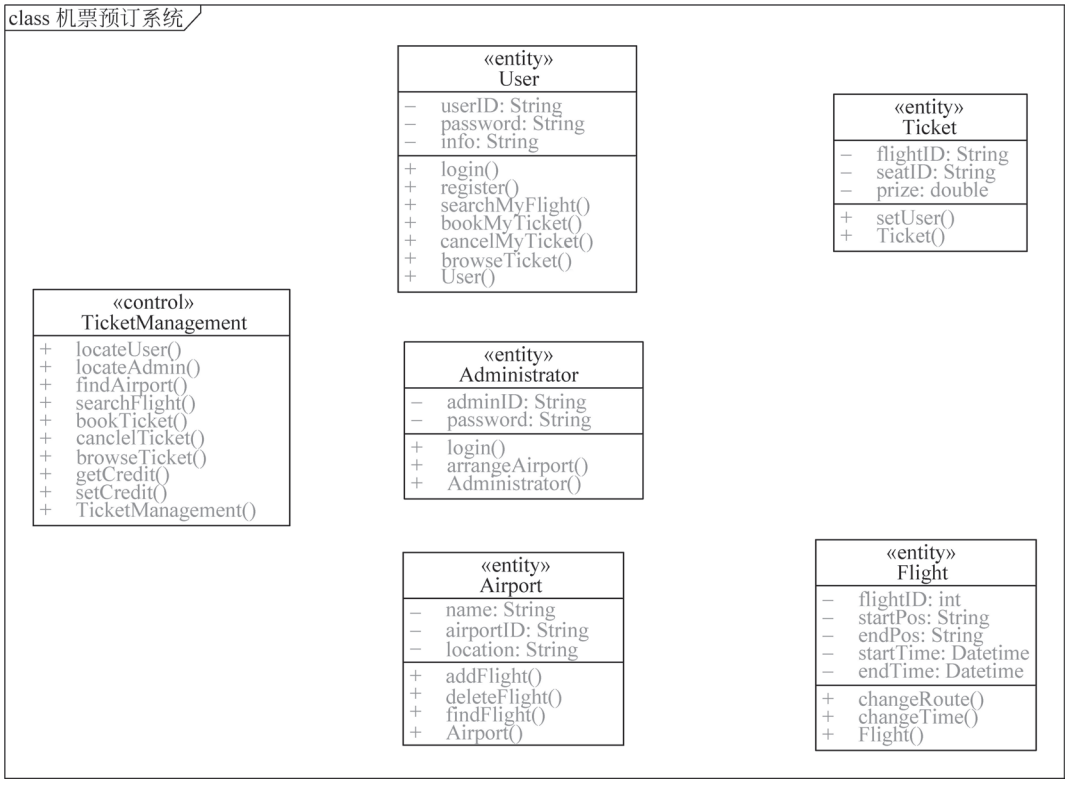


图 5-31　添加类的属性与操作

3. 确定类图中的关系

在确定了类的基本内容之后，我们需要添加类图中的关系来完善类图的内容。类图中的类需要通过关系的联系才能互相协作，发挥完整的作用。

在本节的情境中，类之间主要通过关联关系相互联系。TicketManagement 类与 User 类及 Administrator 类之间相关联，表示系统中包含的用户及管理员账户。TicketManagement 类与 Airport 类之间的关联表示系统中包括的机场。Airport 类与 Flight 类之间的关联表示机场中运行的航班。Flight 类与 Ticket 类之间的关联表示一趟航班中包含的机票。Ticket 类与 User 类之间的关联表示用户所购买的机票。此外，还要注意这些关联关系两端的多重性和导航性。图 5-32 显示了添加了关联关系的类图。至此，类图已基本创建完毕。在实际开发过程中，类图作为指导编程实现最重要的图，还需要不断地修改来完善和丰富其中的内容。

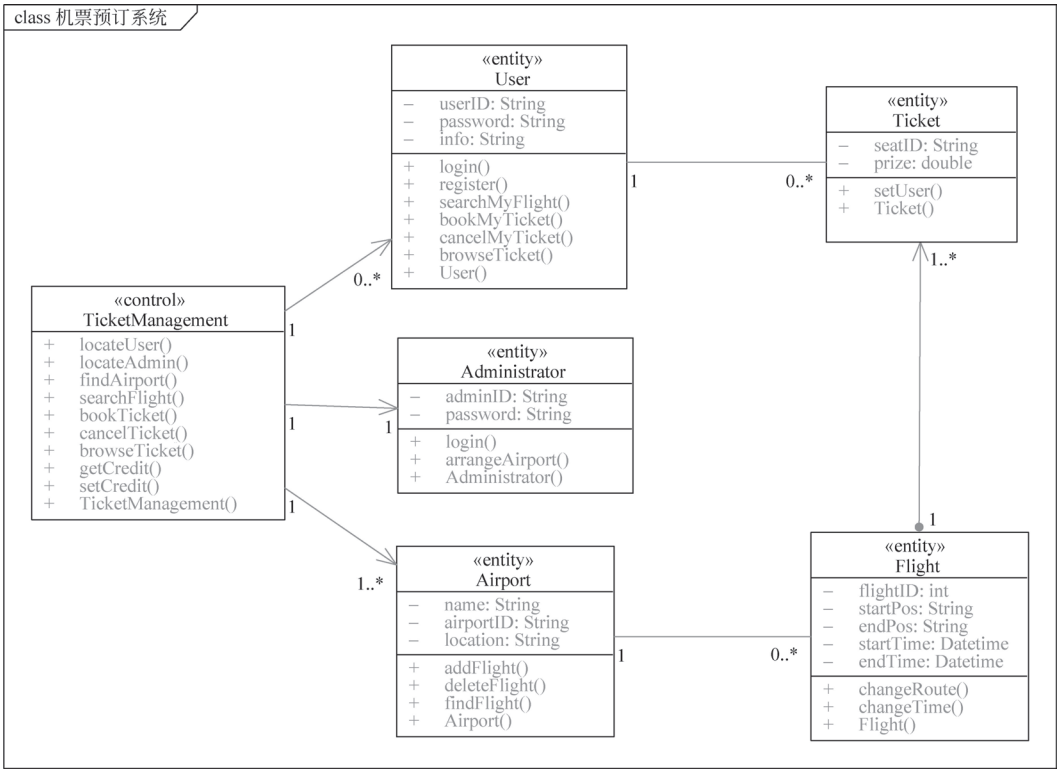


图 5-32 最终类图

扫一扫

视频讲解

4. 使用类图生成“机票预订系统”“Airport”类和“Flight”类的代码

在类图绘制完毕后，可以利用正向工程来生成对应的代码。由于篇幅限制，这里只给出 Airport 类与 Flight 类的生成代码。读者也可以通过对比类图与生成的代码来加深对类图的理解。

Airport 类：

```
//Source file: C:\\Program Files\\Java\\jdk1.8.0_05\\jre\\lib\\Airport.java
public class Airport
{
   private String name;
   private int airportNO;
   private String location;
   public Flight theFlight[];

   /**
    * @roseuid 561BE50C01A2
    */
   public Airport()
   {

   }
```

```
    /**
     * @param flight
     * @return Void
     * @roseuid 561B178F025D
     */
    public Void addFlight(Flight flight)
    {
     return null;
    }

    /**
     * @param flight
     * @return Void
     * @roseuid 561B1797000C
     */
    public Void deleteFlight(Flight flight)
    {
     return null;
    }

    /**
     * @param flightNO
     * @return Flight
     * @roseuid 561B179B0338
     */
    public Flight findFlight(int flightNO)
    {
     return null;
    }
}
```

Flight 类：

```
//Source file: C:\\Program Files\\Java\\jdk1.8.0_05\\jre\\lib\\Flight.java
public class Flight
{
    private int flightNO;
    private String startPos;
    private String endPos;
    private String startTime;
    private String endTime;
    public Airport theAirport;
    public Ticket theTicket[];

    /**
     * @roseuid 561BE50C0208
     */
    public Flight()
    {

    }
```

```
    /**
     * @param s
     * @param e
     * @return Void
     * @roseuid 561B036A0391
     */
    public Void changeRoute(String s, String e)
    {
     return null;
    }

    /**
     * @param s
     * @param e
     * @return Void
     * @roseuid 561B036F0045
     */
    public Void changeTime(String s, String e)
    {
     return null;
    }
}
```

从两个类生成的代码中可以看到，类的属性与操作分别对应于类图中的声明。而且，在Airport 类中添加了 theFlight[]数组，在 Flight 类中添加了 theAirport 与 theTicket[]数组作为表示类间关联关系的属性。

小结

本章主要介绍类图的概念，读者应当重点掌握类图中所包含元素的语义及表示法。在实际建模过程中，类图是整个系统中最重要的图之一，也是与编程实现过程关系最密切的图。本章还介绍了类图的建模技术，以及面向对象的 SOLID 原则。最后通过介绍创建类图的操作与过程使读者对类图的创建过程有一个比较完整的认识。

习题 5

扫一扫

习题

扫一扫

自测题