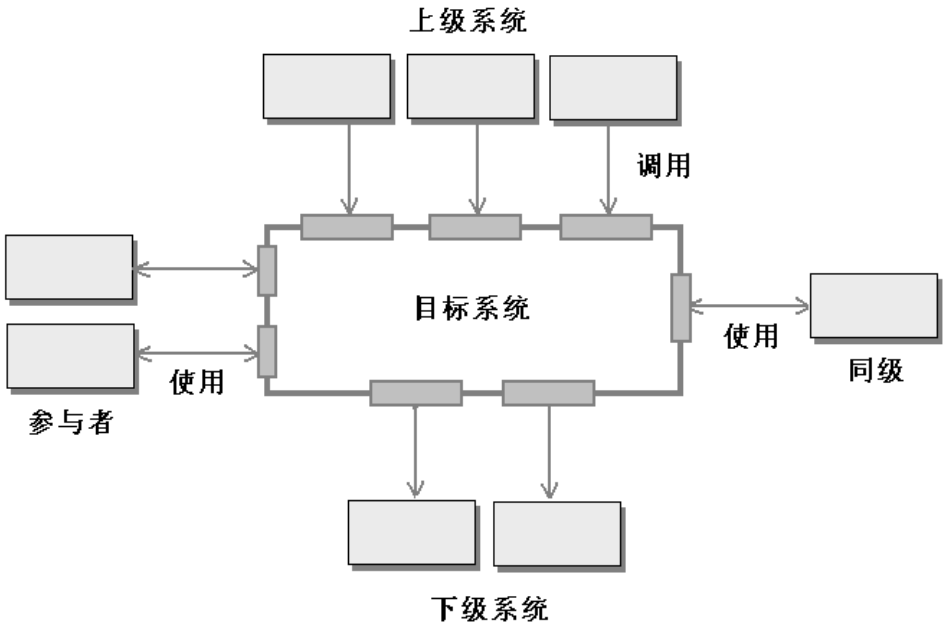
第8章 面向对象设计

- 面向对象设计过程与准则
- 体系结构模块及依赖性
- 系统分解
- 问题域部分的设计
- 人机交互部分的设计
- 任务管理部分的设计
- 数据管理部分的设计
- 对象设计

8.1 面向对象设计过程与准则

- **面向对象设计过程**

(1) 建立系统环境模型。在设计的初始阶段，系统设计师用系统环境图对软件与外部实体交互的方式进行建模。下图给出了系统环境图的一般的结构。



8.1 面向对象设计过程与准则

(2) 设计系统体系结构。体系结构设计可以自底向上进行，如将关系紧密的对象组织成子系统或层；也可以自顶向下进行，尤其是使用设计模式或遗产系统时，会从子系统的划分入手。

(3) 对各个子系统进行设计。对于面向对象的系统，典型的子系统有问题域子系统、人机交互子系统和任务管理子系统。

(4) 对象设计及优化。对象设计以问题领域的对象设计为核心，其结果是一个详细的对象模型。对象设计过程包括使用模式设计对象、接口规格说明、对象模型重构、对象模型优化4组活动。

8.1 面向对象设计过程与准则

- **面向对象设计准则**

(1) 模块化

- **传统的面向过程方法中的模块通常是函数、过程及子程序等，而面向对象方法中的模块则是类、对象、接口、构件等。**
- **在面向过程的方法中，数据及在数据上的处理是分离的；而在面向对象方法中，数据及其上的处理是封装在一起的，具有更好的独立性，也能够更好地支持复用。**

8.1 面向对象设计过程与准则

(2) 抽象

- **面向对象方法不仅支持过程抽象，而且支持数据抽象。类实际上就是一种抽象数据类型。可以将类的抽象分为规格说明抽象及参数化抽象。**
- **类对外开放的公共接口构成了类的规格说明，即协议。这种接口规定了外部可以使用的服务，使用者无需知道这些服务的具体实现算法。通常将这类抽象称为规格说明抽象。**
- **参数化抽象是指当描述类的规格说明时并不具体指定所要操作的数据类型，而是将数据类型作为参数。**

8.1 面向对象设计过程与准则

(3) 信息隐藏

- **在面向对象方法中，信息隐藏通过对象的封装性实现。对于类的用户来说，属性的表示方法和操作的实现算法都应该是隐藏的。**

(4) 弱耦合

- **耦合是指一个软件结构内不同模块之间互连的紧密程度。在面向对象方法中，对象是最基本的模块，因此，耦合主要指不同对象之间相互关联的紧密程度。**

8.1 面向对象设计过程与准则

(5) 强内聚

- **内聚衡量一个模块内各个元素彼此结合的紧密程度。在面向对象设计中存在以下3种内聚：**

(1) 服务内聚：一个服务应该完成一个且仅完成一个功能。

(2) 类内聚：设计类的原则是，一个类应该只有一个用途，它的属性和服务应该是高内聚的。类的属性和服务应该全都是完成该类对象的任务所必需的，其中不包含无用的属性或服务。如果某个类有多个用途，通常应该把它分解成多个专用的类。

(3) 一般—特殊内聚：设计出的一般—特殊结构，应该符合多数人的概念，更准确地说，这种结构应该是对相应的领域知识的正确抽取。

8.1 面向对象设计过程与准则

(6) 可重用

- **软件重用是提高软件开发生产率和目标系统质量的重要途径。**
- **重用基本上从设计阶段开始。重用有两方面的含义：**
  - **一是尽量使用已有的类(包括开发环境提供的类库，及以往开发类似系统时创建的类)，**
  - **二是如果确实需要创建新类，则在设计这些新类的协议时，应该考虑将来的可重复使用性。**

8.2 体系结构模块及依赖性

- 体系结构设计描述了建立计算机系统所需的数据结构和程序构件。一个好的体系结构设计要求软件模块的分层及编程标准的执行。
- 在面向对象软件中，常见的软件模块有类、接口、包和构件。
- 在设计阶段我们往往关注类、接口和包，在实现阶段关注构件，而在部署阶段则关注构件的部署，也就是将构件部署在哪些结点上。

类及其依赖性

1. 类

- 在面向对象的程序设计中，**类和接口是程序的基本组成单元**。
- 一个典型程序需要界面类专门负责表示用户界面信息，需要数据库类负责与数据库进行交互，需要有业务逻辑类负责算法计算等。
- 在计算机程序中，要设计和实现的所有类都具有唯一的名字，在不同的阶段或从不同的角度可以将它们称为**设计类**、**实现类**、**系统类**、**应用类**等。

类及其依赖性

2．继承依赖性

依赖性管理中最棘手的问题是由于继承所引起的依赖性。**继承是一种在父类和子类之间共享属性和行为的方式**，所以运行时可以用一个子类对象代替其父类对象。程序中凡是使用父类对象的地方，都可以用子类对象来代替。一个子类对象是一种特殊的父类对象，它继承父类的所有特征，同时它又可以覆盖父类的方法，从而改变从父类继承的一些特征，并可以在子类中增加一些新的功能。这样，从客户的角度看，在继承树中为请求提供服务的特定对象不同，系统的运行行为可能会有所不同。

类及其依赖性

（1）多态继承

根据为请求提供服务的对象不同可以得到不同的行为，这种现象称为**多态**。在运行时对类进行实例化，并调用与实例化对象相应的方法，称为**动态绑定**、后期绑定或运行时绑定。相应地，如果方法的调用是在编译时确定的，则称为是**静态绑定**、前期绑定或编译时绑定。

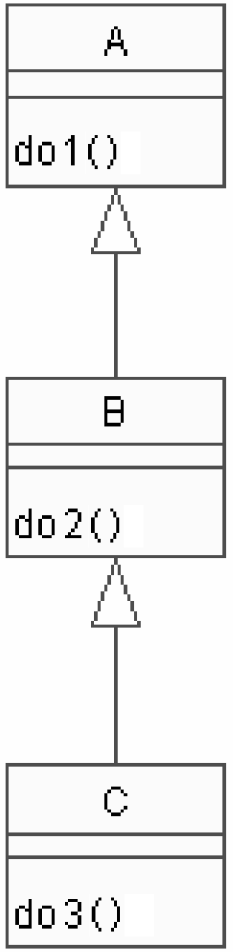
多态并不是伴随着继承而出现。如果在子类中不覆盖父类中的任何方法，就不会产生多态行为。

很明显，继承会带来类和方法之间的依赖性。继承带来的依赖性有**编译时继承依赖性**和**运行时继承依赖性**。

类及其依赖性

① 编译时继承依赖性

右图所示的例子说明了一棵树中类之间的编译时依赖性。在这个例子中，B继承A，但没有覆盖A中的方法do1( )。因此，B和A之间没有运行时继承依赖性。也就是说，由于编译时依赖性的存在，A中do1( )方法的任何变化，都会被B在编译时（静态地）继承。

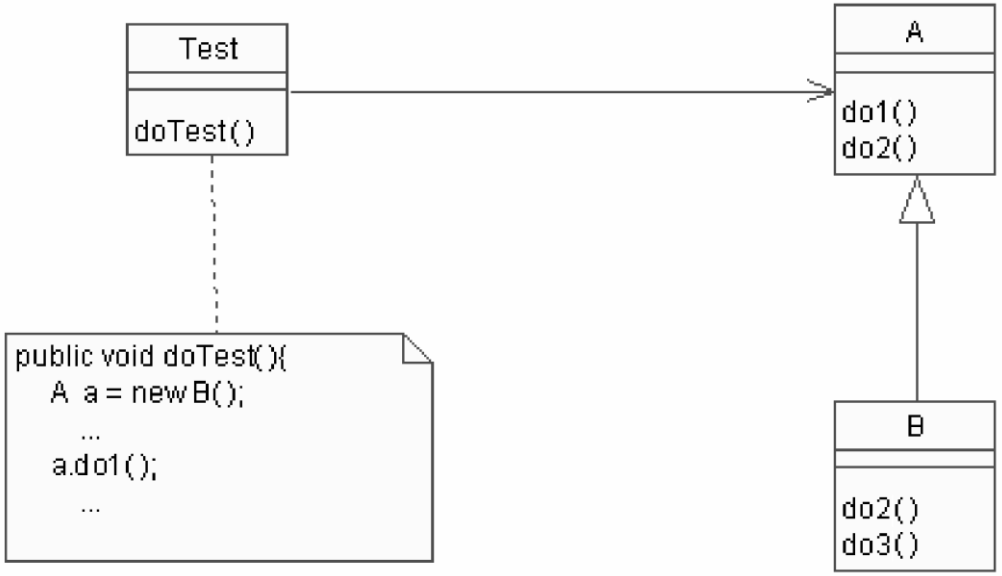


一般来说，**所有的继承都会引入编译时依赖性**。依赖性是**可传递的**，也就是说，如果C依赖B，B依赖A，那么C也依赖A。

类及其依赖性

② 运行时继承依赖性

下图举例说明了在一棵继承树中涉及客户对象访问类服务的运行时继承依赖性。图中类B的do1( )方法是从父类A继承来的，因此Test与B没有运行时继承依赖性，只是一个静态依赖性，通过从Test到A的关联来表明。如果在doTest方法中调用的是do2( )方法，或者在B中覆盖了A的do1( )方法，则从Test到A和B就会存在运行时依赖性。



类及其依赖性

（2）无多态继承

使用继承最简单的方式是子类不覆盖从父类继承来的方法，这样就不存在多态性继承问题。虽然无多态的继承有时并不是十分有用，但理解和管理起来是最容易的。

（3）扩展继承和约束继承

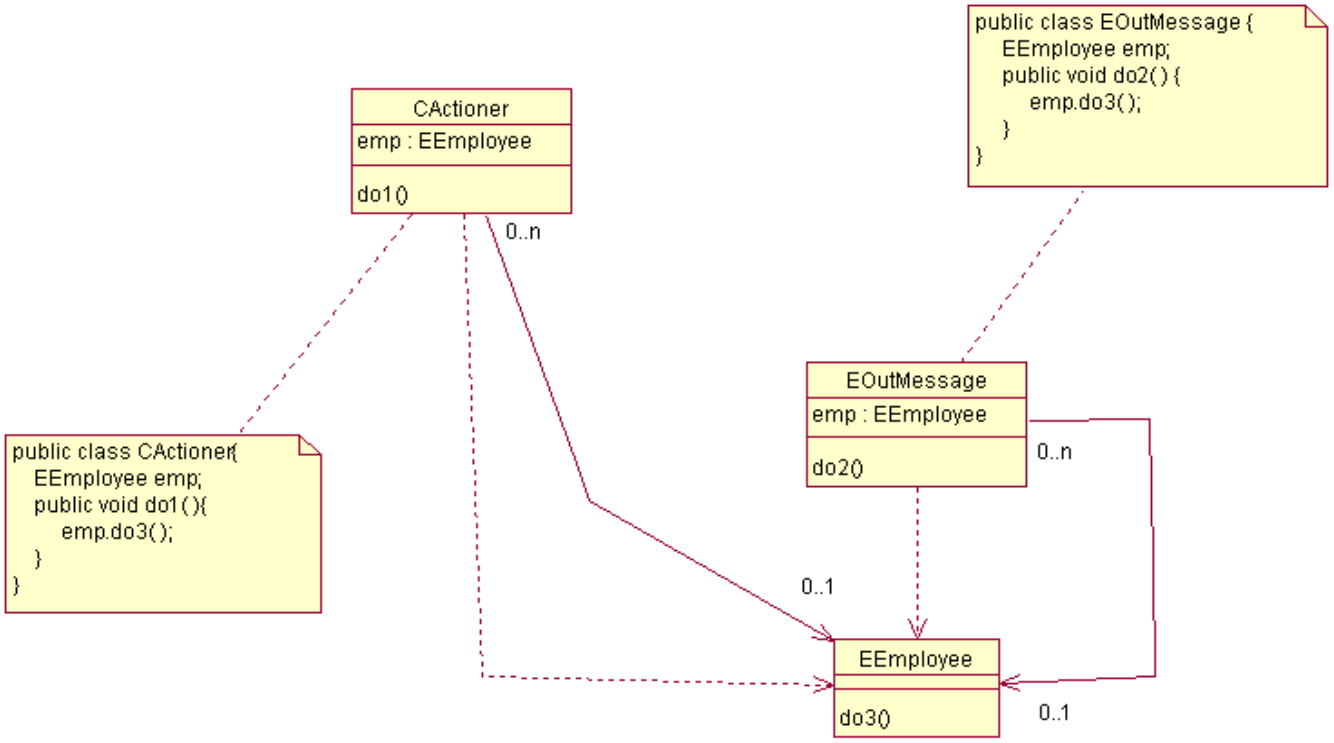
扩展继承是指子类继承父类的属性，并且提供额外属性来增强类定义。子类是父类的一种，如果子类覆盖了父类的方法，那么被覆盖的方法应该实现该方法的定义，并且能够在子类的语境中工作。

当一个类覆盖了继承来的方法，并对一些继承来的功能进行了限制，这时就产生了**约束继承**。这时，子类不再是父类的一种。有时，限制会造成继承方法的完全禁止。当方法的实现是空时，就会发生这种情况。

类及其依赖性

3．交互依赖性

交互依赖性也称为方法依赖性，是通过消息连接产生的。如下图所示。



类及其依赖性

图中，CActioner使用方法do1( )来发送一条消息do3( )给EEmployee，因此，do1( )依赖于do3( )。依赖性向上传递给所属的类，因此，CActioner依赖于EEmployee。类似地，EOutMessage的do2( )调用EEmployee的方法do3( )，因此，EOutMessage依赖于EEmployee。

接口及其依赖性

1．接口

在UML2.0中，接口是不可直接实例化的特性集合的声明，即其**对象不能直接实例化**，需要通过类来实现，实现接口的类需要实现接口中声明的方法。UML2.0对流行编程语言中的接口概念进行了扩展。接口中不仅可以声明操作，还可以声明属性。

由于允许在接口中存在属性，因此，在接口之间或者接口和类之间可能会产生关联。用另一个接口或类作为属性的类型可以表示关联。

接口及其依赖性

在UML2.0中，可以通过关联实现从接口到类的导航。但在Java中是无法实现的，因为Java规定接口中的数据元素必须是常量。

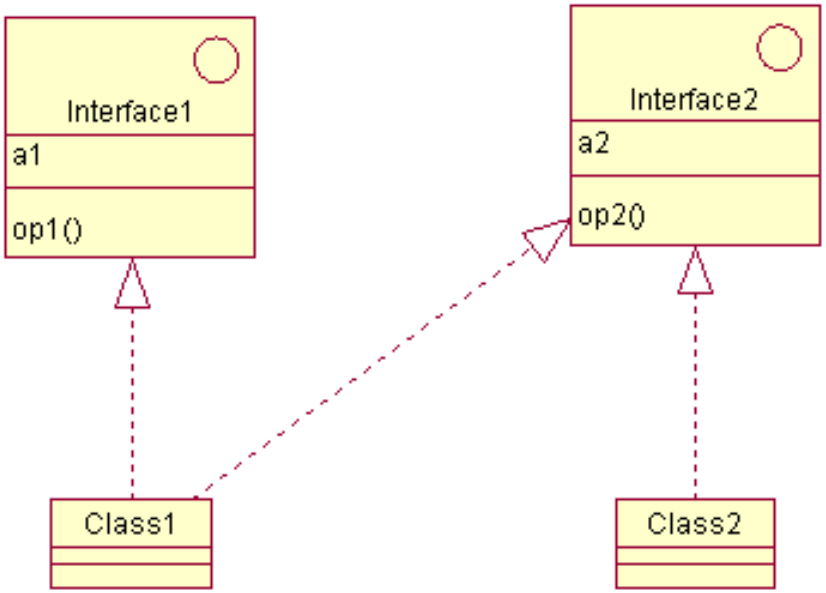
接口与抽象类有相似之处，抽象类是至少包含一个没有实现的方法的类，如果在一个抽象类中所有的方法都没有实现，则称其为**纯抽象类**，从这一点上，接口和纯抽象类似乎没有区别。但实际上，接口和抽象类还是有着本质的区别。在只支持单继承的语言中，**一个类只能有一个直接父类，但是却可以实现多个接口。**

接口及其依赖性

2．实现依赖性

一个类可以实现多个接口，由类实现的接口集合称为该类的**供给接口**。在UML2.0中，将一个类和该类实现的接口之间的依赖性称为**实现依赖性**。

右图所示为实现依赖性的UML符号，在箭头末端的类实现了箭头所指向的接口。从图中可以看到，Class1实现了Interface1接口和Interface2接口，而Class2只实现了Interface2接口。



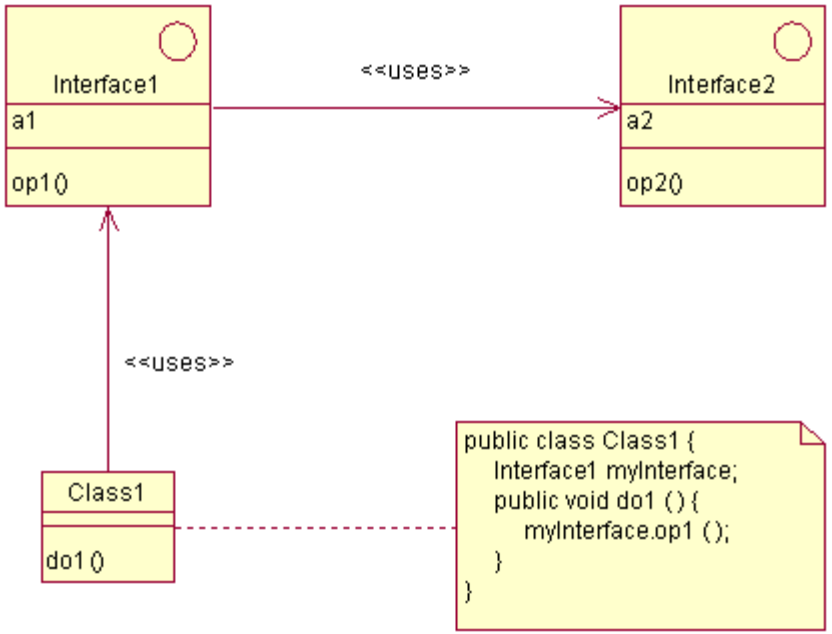
接口及其依赖性

3．使用依赖性

一个接口可以为其他类或接口提供服务，同时也可能需要其他接口的服务。一个接口所需要的其他接口所提供的服务称为这个类的**需求接口**。需求接口详细说明一个类或接口需要的服务，从而可以为其客户提供服务。在UML2.0中，通过类（接口）和它所需接口之间的依赖关系来说明需求接口，这称为**使用依赖性**。

下图所示为使用依赖性的UML符号，在箭头尾部的类或接口使用在箭头头部的接口。Class1使用Interface1，Interface1使用Interface2。在Java语言中，不允许接口之间的使用，只允许接口间的扩展继承。

接口及其依赖性



Class1包含方法do1( )，而do1( )调用操作op1( )。在静态代码中，并不清楚需求接口的哪个实现提供了所需的服务，可以是实现Interface1的任何一个类实例。当Class1的一个执行实例设置数据成员myInterface的值时，具体实例才能确定，从而可以引用具体类的一个具体对象。

包及其依赖性

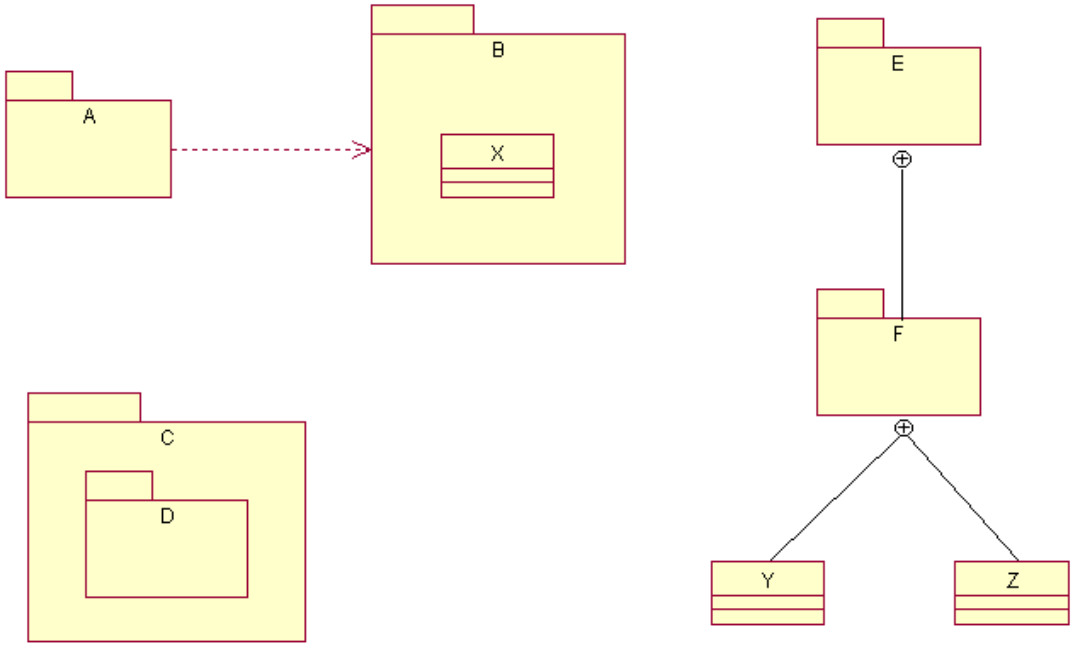
1. 包

包（package）又可称为层或子系统，是表示组织类的一种方式，用于划分应用程序的逻辑模型。包是高度相关的类的聚合，这些类本身是内聚的，但相对于其他聚合来说又是松散耦合的。

包可以嵌套。外层包可以直接访问包括在它的嵌套包中的任何类。**包还可以导入其他包**，例如，在包A中导入了包B，这意味着包A或者包A的元素可以引用包B或者包B的元素。因此，虽然一个类只属于一个包，但是它可以被导入其他包。包的导入操作会引入包之间的依赖性以及它们的元素之间的依赖性。

包及其依赖性

下图为UML包的例子。一个包可以不暴露任何成员，也可以明确标明它所包含的成员，或者用符号“⊕”来表示。图中，包 B拥有类 X，包C拥有包 D，包E拥有包 F，包 F拥有类Y和类Z。



包及其依赖性

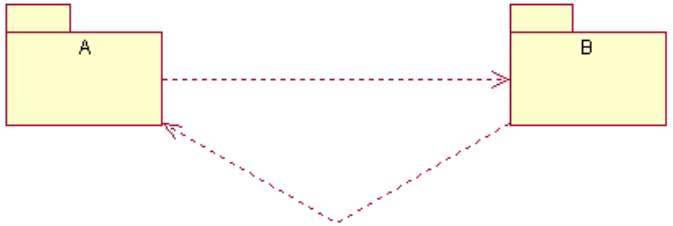
如果包A的一些成员在某种程度上引用了包B的某些成员（包A导入了包B的一些成员），这隐含着**双重含义**。

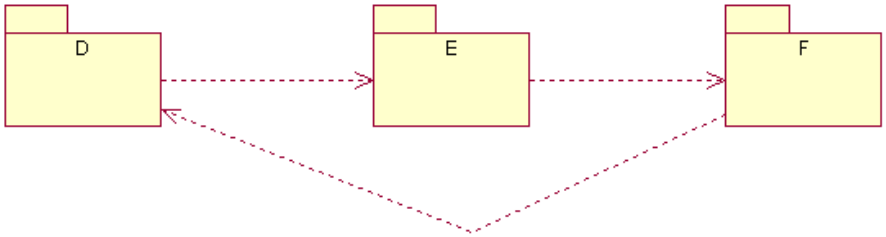
- 包B的变化可能会影响包A，通常需要对包A重新进行编译和测试。
- 包A只能和包B一起使用。

包及其依赖性

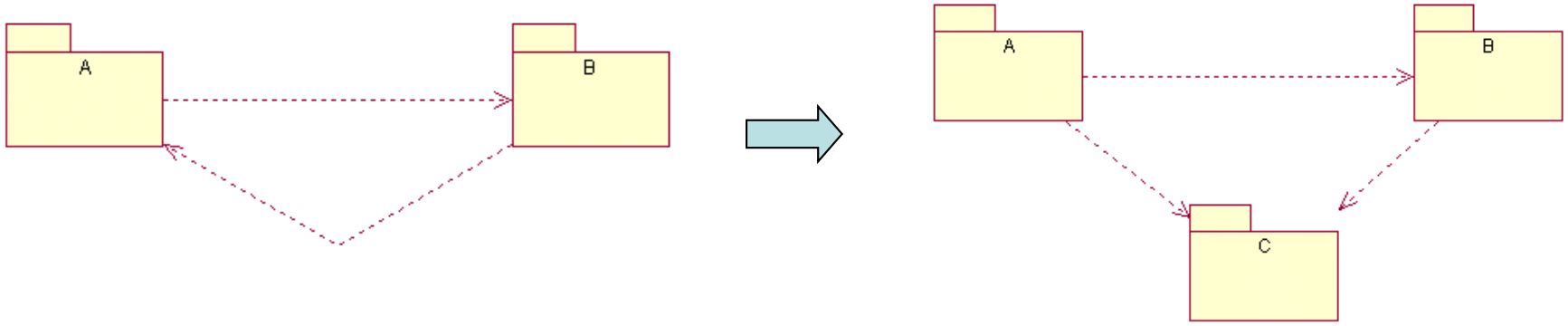
2．包依赖性

本质上，两个包之间的依赖性来自于两个包中类之间的依赖性。类之间的循环依赖性是个特别棘手的问题，好在大多数情况下可以通过重新设计避免循环依赖性。





包及其依赖性

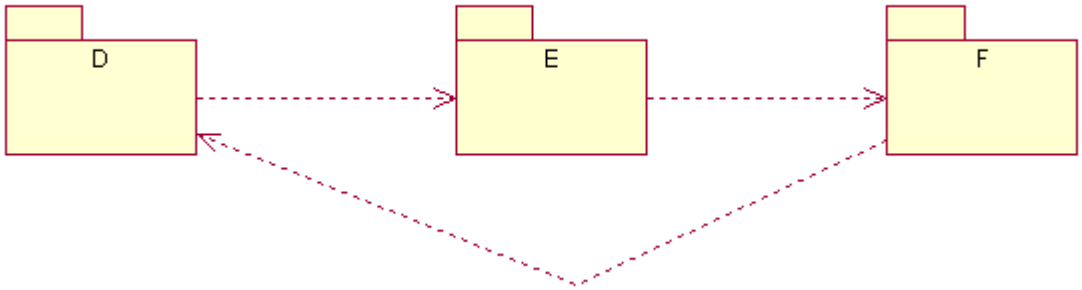


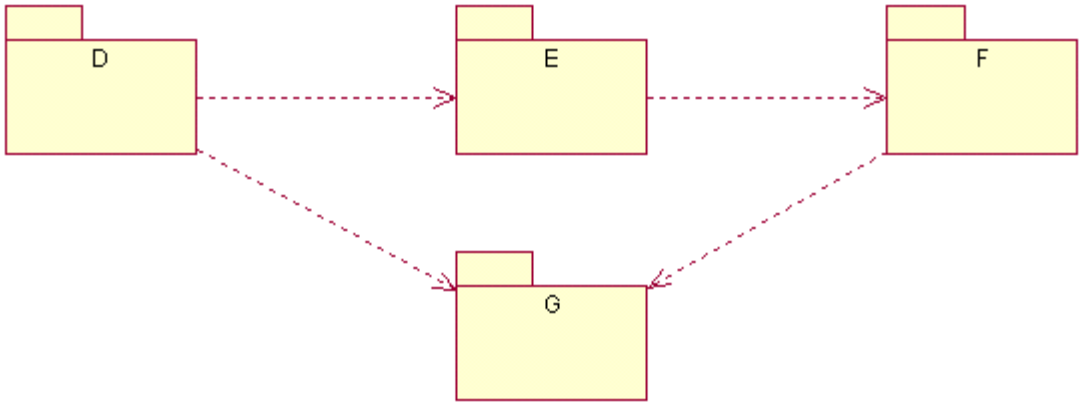
通过在上图中增加新包可以消除包之间的循环依赖性。

方法为：在第1个例子中将包B依赖的包A的元素从包A中分离出来，组成包C，使得包B不再依赖包A，而是依赖包C；

包及其依赖性

在第2个例子中，将包F所依赖的包D中的元素从包D中分离出来，组成包G。消除循环依赖性后如下图所示。





构件及其依赖性

在面向对象的软件工程环境中，面向对象技术已达到了类级复用，而构件级复用则是比类级复用更高一级的复用，它是对一组类的组合进行封装（当然，在某些情况下，一个构件可能只包含一个单独的类），并代表完成一个或多个功能的特定服务，也为用户提供了多个接口。

一个构件可以是一个编译的类，可以是一组编译的类，也可以是其他独立的部署单元，如一个文本文件、一个图片、一个数据文件、一个脚本等。

构件及其依赖性

从软件复用的角度，**构件**是指在软件开发过程中可以重复使用的软件元素，这些软件元素包括**程序代码、测试用例、设计文档、设计过程、需求分析文档、甚至领域知识**。可复用的软件元素越大，我们称复用的粒度就越大。为了能够支持复用，软件构件应具有以下特性：

（1）**独立部署单元**：一个构件是独立部署的，意味着它必须能与它所在的环境及其他构件完全分离。

（2）**作为第三方的组装单元**：构件必须具备很好的内聚性，必须封装它的实现，并且只通过良好定义的接口与外部环境进行交互。

构件及其依赖性

（3）一个构件不能有任何（外部的）可见状态：即构件不能与自己的拷贝有所区别。

根据上述特性可以得出以下的定义：

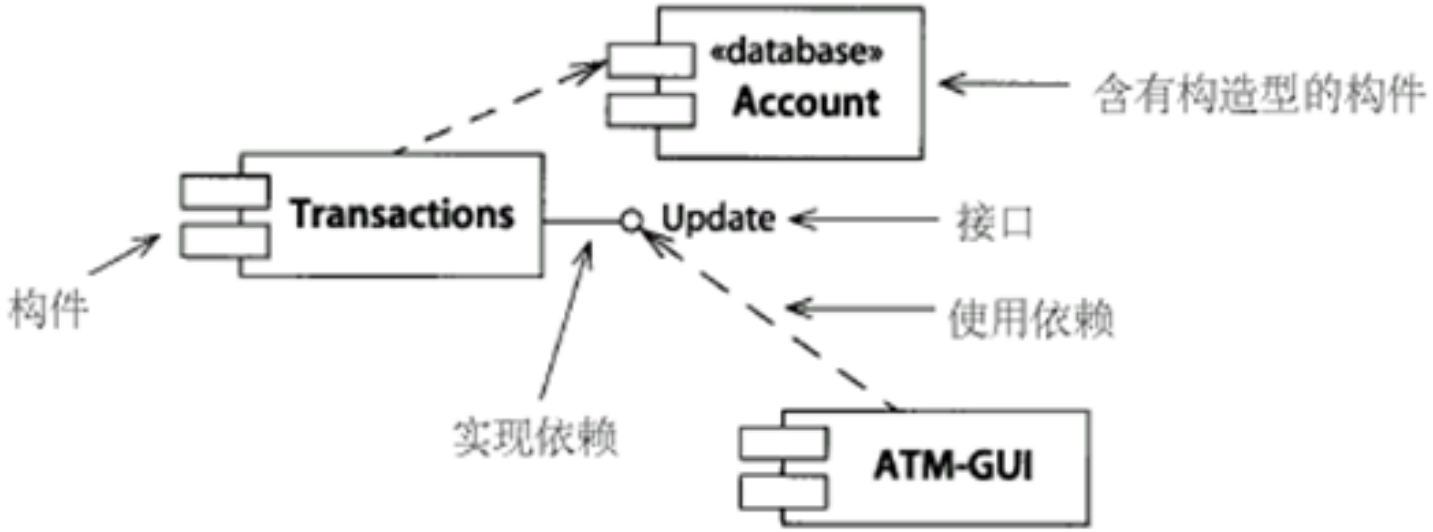
“软件构件是一种组装单元，它具有规范的接口规格说明

和显示的语境依赖。软件构件可以被独立部署，并由第三方任意组装。”

OMG UML规范中将构件定义为“系统中某一定型化的、可配置的和可替换的部件，该部件封装了实现并暴露一系列接口”。

构件及其依赖性

构件图表示构件之间的依赖关系，如下图所示。每个构件实现（支持）一些接口，并使用另一些接口。



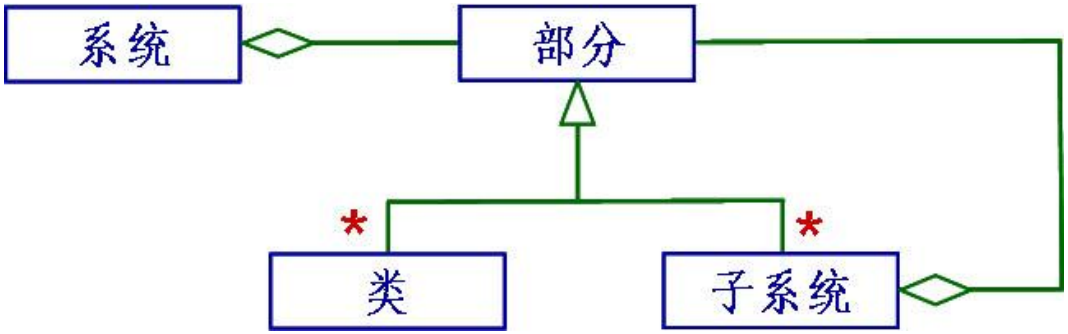
构件及其依赖性

构件与类的区别是双重的：

- **首先，构件是部署在某个计算机结点上的物理抽象。类表示逻辑事务，为了起到物理抽象的作用，不得不将其实现为构件。**
- **其次，构件只显示它所包含的类的某些接口，很多其他接口都被封装在构件中——它们只被协作的类在内部使用，对于其他构件是不可见的。**

8.3 系统分解

- 子系统和类
  - 在大型和复杂的软件系统情形，首先根据需求的功能模型（用例模型），将系统分解成若干个部分，每一部分又可分解为若干子系统或类，每个子系统还可以由更小的子系统或类组成，如图所示。



系统结构的类图

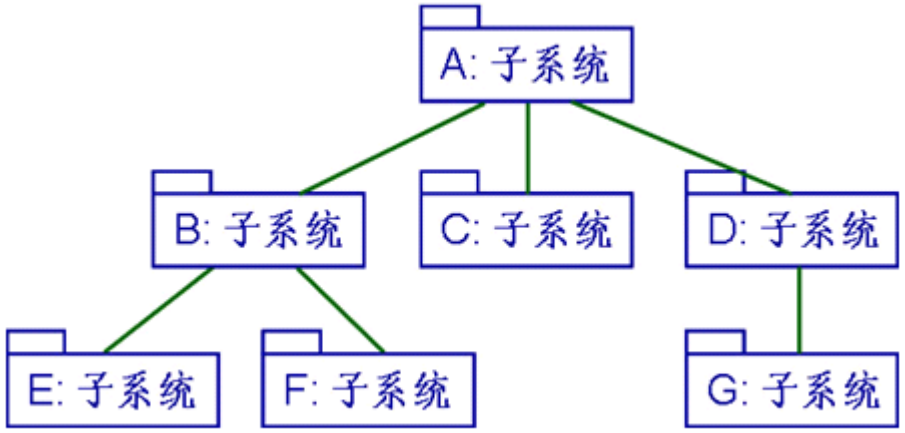
8.3 系统分解

- 服务和子系统接口

- 服务是一组有公共目的的相关操作。而子系统则通过给其他子系统提供服务来发挥自己的能力。与类不同的是，子系统不要求其他子系统为它提供服务。
- 供其他子系统调用的某个子系统的操作集合就是子系统的接口。
- 子系统的接口包括操作名、操作参数类型及返回值。
- 面向对象的系统设计主要关注每个子系统提供服务的定义，即枚举所有的操作、操作参数和行为。

8.3 系统分解

- 服务和子系统接口
  - **子系统分层的目的是建立系统的层次结构。每一层仅依赖于它下一层提供的服务，而对它的上一层可以一无所知。下图给出了一个三层的系统结构的示例。**

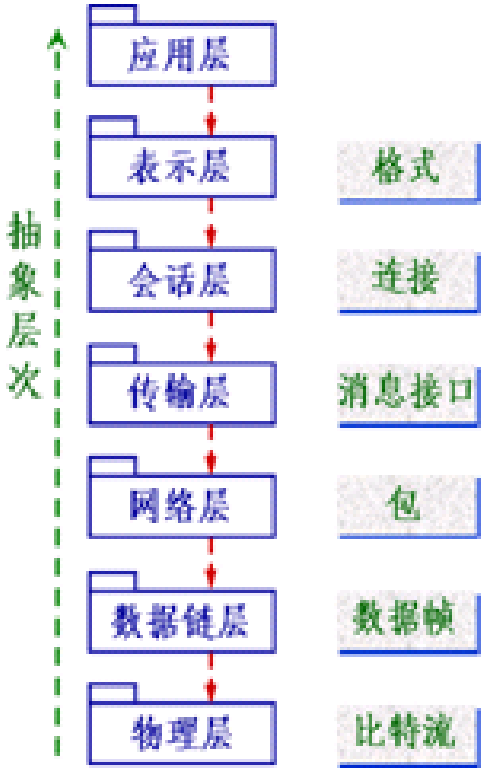


8.3 系统分解

- 服务和子系统接口

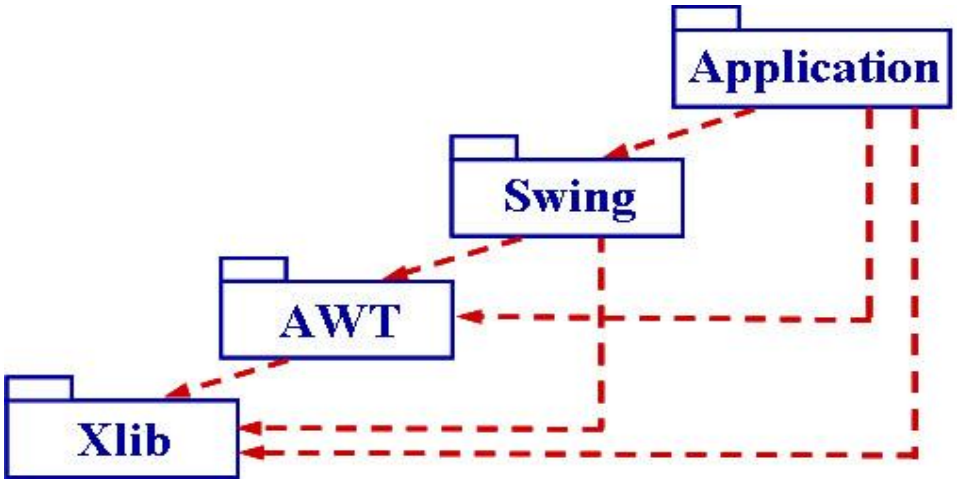
➢ **如果在一个系统的层次结构中，每一层只能访问与其相邻的下一层，则称之为封闭体系结构；如果每一层还可访问比其相邻下一层更低的层次，则称之为开放体系结构。**

➢ **典型的封闭体系结构的例子就是开放系统互联参考模型（OSI模型），如图所示。**



8.3 系统分解

- 服务和子系统接口
  - **开放体系结构的一个例子是Java的Swing用户接口包。它允许绕过高层直接访问低层接口以克服性能瓶颈。如图所示。**



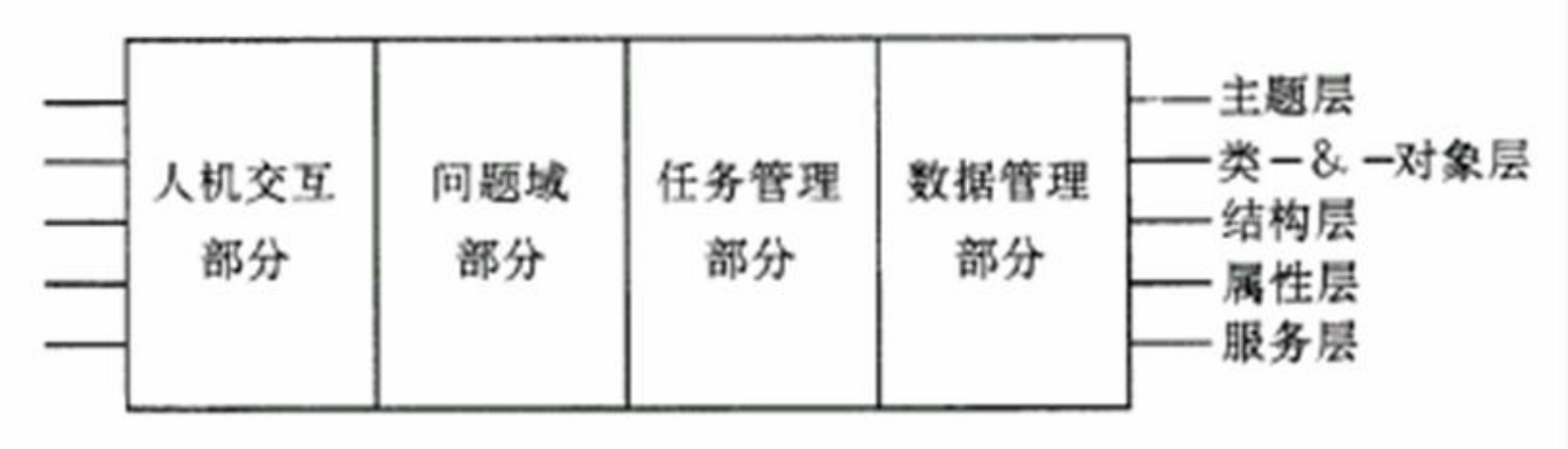
8.3 系统分解

- **Coad&Yourdon的面向对象设计模型**

- Coad & Yourdon基于MVC（Model-View-Controller）模型，在逻辑上将系统划分为4个部分，分别是问题域部分、人机交互部分、任务管理部分及数据管理部分，每一部分又可分为若干子系统。
- Coad 与 Yourdon 在设计阶段中继续采用了分析阶段中提到的5个层次，用于建立系统的4个组成成分。每一个子系统都由主题、类-&-对象、结构、属性和服务5个层次组成。这5个层次可以被当作整个模型的水平切片。

8.3 系统分解

- 典型的面向对象设计模型



8.3 系统分解

- 子系统之间的两种交互方式
  - **客户-供应商关系**：在这种关系中，客户子系统调用供应商子系统，后者完成某些服务工作并返回结果。使用这种交互方案，作为客户的子系统必须了解作为供应商的子系统的接口，而后者却无须了解前者的接口。
  - **平等伙伴关系**：在这种关系中，每个子系统都可能调用其他子系统，因此每个子系统都必须了解其他子系统的接口。与第一种方案相比，这种方案中，子系统间的交互更加复杂。

8.3 系统分解

- 组织系统的两种方案

- **分层组织**：这种组织方案把软件系统组织成一个层次系统，每层是一个子系统。上层在下层的基础上建立，下层为实现上层功能而提供必要的服务。每一层内所包含的对象，彼此间相互独立，而处于不同层次上的对象，彼此间往往有关联。
- **块状组织**：这种组织方案把软件系统垂直地分解成若干个相对独立的、弱耦合的子系统，一个子系统相当于一块，每块提供一种类型的服务。

混合使用层次结构和块状结构，可以成功地由多个子系统组成一个完整的软件系统。

8.4 问题域部分的设计

- 典型的面向对象系统一般由三层组成，即数据库层、业务逻辑层及用户界面层。那么，在这三层中，首先从哪一层开始设计呢？
- 实际上，面向对象的设计也是以面向对象分析的模型为基础的。
- 面向对象的分析模型包括有用例图、类图、顺序图和包图，主要是对问题领域进行描述，基本上不考虑技术实现，当然也不考虑数据库层和用户界面层。
- 面向对象分析所得到的问题域模型可以直接应用于系统的问题域部分的设计。

所以，面向对象设计应该从问题域部分的设计开始，也就是三层结构的中间层——应用逻辑层。

8.4 问题域部分的设计

- 在面向对象设计过程中，可能对面向对象分析所得出的问题域模型做以下方面的补充或调整。

(1) 调整需求。有两种情况会导致修改通过面向对象分析所确定的系统需求：

- **一是用户需求或外部环境发生变化；**
- **二是分析员对问题理解不透彻，导致分析模型不能完整、准确地反映用户的真实需求。**

8.4 问题域部分的设计

(2) 复用已有的类

- **从类库选择已有的类，从供应商那里购买商业外购构件，从网络、组织、小组或个人那里搜集适用的遗留软构件，把它们增加到问题域部分的设计中去。**
- **在被复用的已有类和问题域类之间添加泛化（一般化/特殊化）关系，继承被复用类或构件属性和方法。**
- **标出在问题域类中因继承被复用的已有类或构件而成为多余的属性和服务。**
- **修改与问题域类相关的关联。**

8.4 问题域部分的设计

(3) 把问题域类组合在一起

- **在进行面向对象设计时，通常需要先引入一个类，以便将问题域专用的类组合在一起，它起到“根”类的作用，将全部下层的类组合在一起。**
- **当没有一种更满意的组合机制可用时，可以从类库中引进一个根类，作为包容类，把所有与问题领域有关的类关联到一起，建立类的层次。**
- **之后，将同一问题领域的一些类集合起来，存于类库中。**

8.4 问题域部分的设计

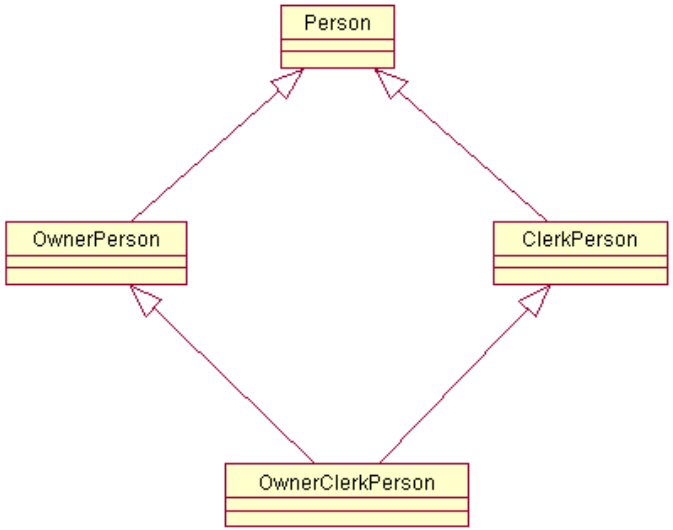
(4) 增添泛化类以建立类间的协议

- **有时某些问题域的类要求一组类似的服务（以及相应的属性）。此时，以这些问题域的类作为特化的类，定义一个泛化类。**
- **该泛化类定义了为所有这些特化类共用的一组服务名，作为公共的协议，用来与数据管理或其他外部系统部件通信。**
- **这些服务都是虚函数。在各个特化类中定义其实现。**

8.4 问题域部分的设计

(5) 调整继承的支持级别

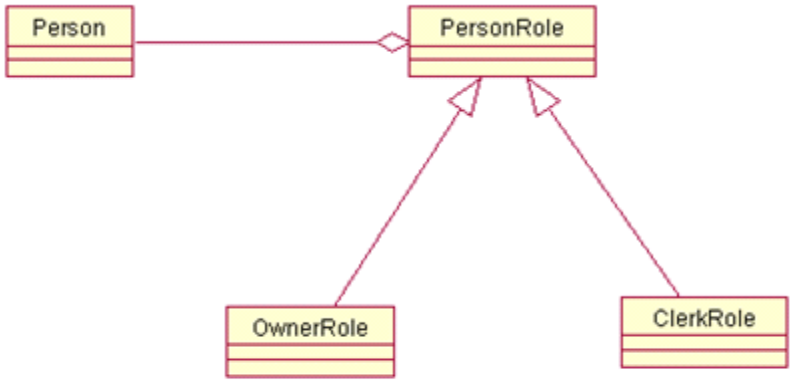
- **如果在分析模型中一个泛化关系中的特化类继承了多个类的属性或服务，就产生了多继承关系，如图所示。**



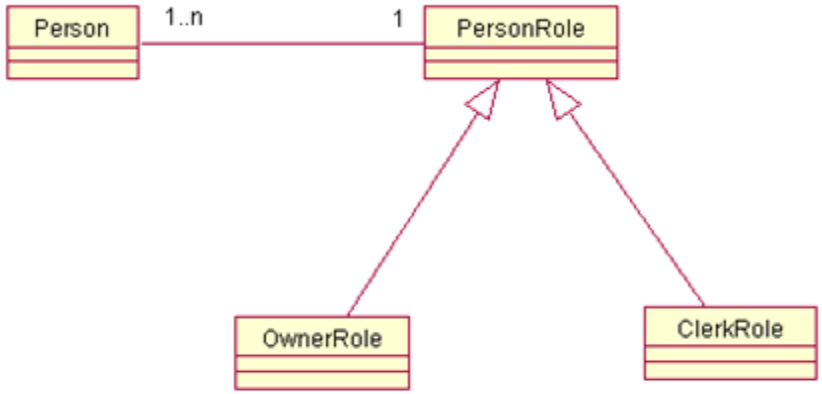
8.4 问题域部分的设计

1) 针对单继承语言的调整。对于只支持单继承关系的编程语言，可以使用两种方法将多继承结构转换为单继承结构。

- 把特化类看做是泛化类所扮演的角色，如图(a)和图(b)所示。



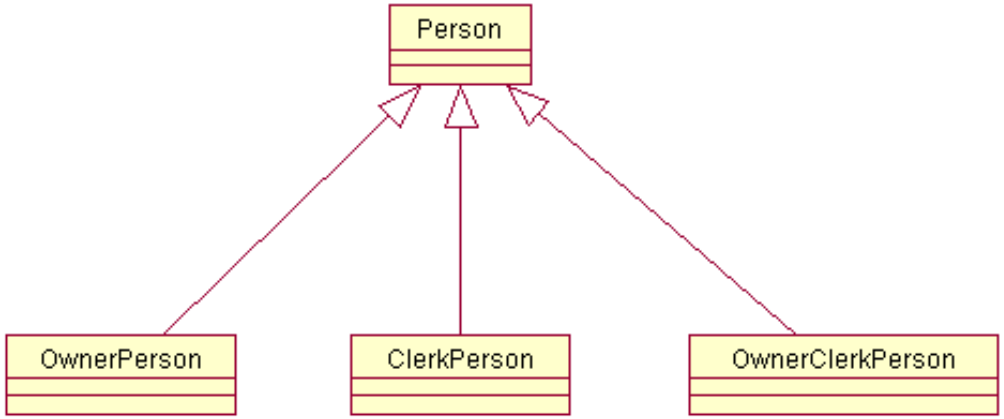
(a) 采用聚合将角色关联到人



(b) 采用实例连接将角色关联到人

8.4 问题域部分的设计

- **把多继承的层次结构平铺为单继承的层次结构，如图所示。这意味着该泛化关系在设计中就不再那么清晰了。同时某些属性和服务在特化类中重复出现，造成冗余。**



8.4 问题域部分的设计

2) 针对无继承语言的调整。

- **编程语言中的继承属性提供了表达问题域的一般化/特殊化语义的语法，它明确地表示了公共属性和服务，还为通过可扩展性而达到可复用性提供了基础。**
- **然而，由于开发组织方面的原因，有些项目最终选择了不支持继承性的编程语言。对于一个不支持继承的编程语言来说，只能将每一个泛化关系的层次展开，成为一组类及对象，之后再使用命名惯例将它们组合在一起。**

8.4 问题域部分的设计

(6) 改进性能

提高执行效率是系统设计的目标之一。为以提高效率有时必须改变问题域的结构。

1) 如果类之间经常需要传送大量消息，可合并相关的类，使得通信成为对象内的通信，而不是对象之间的通信，或者使用全局数据作用域，打破封装的原则，以减少消息传递引起的速度损失。

2) 增加某些属性到原来的类中，或增加低层的类，以保存暂时结果，避免每次都要重复计算造成速度损失。

8.4 问题域部分的设计

(7) 存储对象

通常的作法是，每个对象将自己传送给数据管理部分，让数据管理部分来存储对象本身。

8.5 人机交互部分的设计

- 用户界面设计步骤

(1) 从系统的输入、输出及与用户的交互中获得信息，定义界面对象和行为（操作）。

(2) 定义那些导致用户界面状态发生变化的事件，对事件建模。

(3) 描述最终向用户展示的每一个界面的状态，。

(4) 简要说明用户如何从界面提供的界面信息来解释系统状态。

Web应用系统的界面设计

- 界面设计目标

(1) 简单性：尽量做到适度和简单，不要在页面上提供太多的东西。

(2) 一致性：这一设计目标几乎适用于设计模型的每一个元素。

(3) 确定性：Web应用系统的美学、界面和导航设计必须与将要构造的应用系统所处的领域保持一致。

(4) 健壮性：在已经建立的确定性的基础上，Web应用系统通常会给用户明确的“承诺”。

Web应用系统的界面设计

- 界面设计目标

(5) 导航性：我们已经在前面提及了导航应该简单和一致，也应该以直观的和可预测的方式来设计。也就是说，用户不必搜索导航链接和帮助就知道如何使用Web应用系统。

(6) 视觉吸引：在所有类型的软件中，Web应用系统毫无疑问是最具有视觉效果的、最生动的、也是最具有审美感的。

(7) 兼容性：Web应用系统会应用于不同的环境(例如，不同的硬件、Internet连接类型、操作系统、浏览器)，并且必须互相兼容。

Web应用系统的界面设计

- 界面设计工作流

(1) 回顾那些在分析模型中的信息，并根据需要进行优化。

(2) 开发Web应用系统界面布局的草图。

(3) 将用户目标映射到特定的界面行为。

(4) 定义与每个行为相关的一组用户任务。

(5) 为每个界面行为设计情节串联图板屏像。

(6) 利用从美学设计中的输入来优化界面布局和情节串联图板。

Web应用系统的界面设计

- 界面设计工作流

(7) 明确实现界面功能的界面对象。

(8) 开发用户与界面交互的过程表示。

(9) 开发界面的行为表示法。

(10) 描述每种状态的界面布局。

(11) 优化和评审界面设计模型。

8.6 任务管理部分的设计

- **任务管理主要包括任务的选择和调整。常见的任务有事件驱动型任务、时钟驱动型任务、优先任务、关键任务和协调任务等。**
- **设计任务管理子系统时，需要确定各类任务，并将任务分配给适当的硬件或软件去执行。**

8.6 任务管理部分的设计

1. 识别事件驱动任务

- **有些任务是事件驱动的，这些任务可能是负责与设备、其他处理机或其他系统通信的。**
- **这类任务可以设计成由一个事件来触发，该事件常常针对一些数据的到达发出信号。**
- **数据可能来自数据行或者来自另一个任务写入的数据缓冲区。**

8.6 任务管理部分的设计

(1) 识别事件驱动任务

- **当系统运行时，这类任务的工作过程如下：**
  - ✓ **任务处于睡眠状态，等待来自数据行或其他数据源的中断；**
  - ✓ **一旦接收到中断就唤醒该任务，接收数据并将数据放入内存缓冲区或其他目的地，通知需要知道这件事的对象，然后该任务又回到睡眠状态。**

8.6 任务管理部分的设计

(2) 识别时钟驱动任务

- **以固定的时间间隔激发这种事件，以执行某些处理。某些人机界面、子系统、任务、处理机或其他系统可能需要周期性的通信，因此时钟驱动任务应运而生。**
- **当系统运行时，这类任务的工作过程如下：**
- ✓ **任务设置了唤醒时间后进入睡眠状态；**
- ✓ **等待来自系统的一个时钟中断，一旦接收到这种中断，任务就被唤醒，并做它的工作，通知有关的对象，然后该任务又回到睡眠状态。**

8.6 任务管理部分的设计

(3) 识别优先任务

- **根据处理的优先级别来安排各个任务。优先任务可以满足高优先级或低优先级的处理需求。**
- ✓ **高优先级：某些服务具有很高的优先级，为了在严格限定的时间内完成这种服务，可能需要把这类服务分离成独立的任务。**
- ✓ **低优先级：与高优先级相反，有些服务是低优先级的，属于低优先级处理（通常称为后台处理）。设计时可能用额外的任务把这样的处理分离出来。**

8.6 任务管理部分的设计

(4) 识别关键任务

- **关键任务是有关系统成功或失败的关键处理，这类处理通常都有严格的可靠性要求。**
- **在设计过程中可能用额外的任务把这样的关键处理分离出来，以满足高可靠性处理的要求。**
- **对高可靠性处理应该精心设计和编码，并且应该严格测试。**

8.6 任务管理部分的设计

(5) 识别协调任务

- **当有三个或更多的任务时，可考虑另外增加一个任务，这个任务起协调者的作用，将不同任务之间的协调控制封装在协调任务中。**
- **可以用状态转换矩阵来描述协调任务的行为。**

8.6 任务管理部分的设计

(6) 审查每个任务

- **要使任务数保持到最少。**
- **对每个任务要进行审查，确保它能满足一个或多个选择任务的工程标准——事件驱动、时钟驱动、优先任务/关键任务或协调者。**

8.6 任务管理部分的设计

(7) 定义每个任务

1) **它是什么任务**。首先要为任务命名，并对任务做简要描述。为面向对象设计部分的每个服务增加一个新的约束——任务名。如果一个服务被分裂，交叉在多个任务中，则要修改服务名及其描述，使每个服务能映射到一个任务。

2) **如何协调任务**。定义每个任务如何协调工作。指出它是事件驱动的，还是时钟驱动的；对于事件驱动的任务，描述触发该任务的事件；对时钟驱动的任务，描述在触发之前所经过的时间间隔，同时指出它是一次性的，还是重复的事件间隔。

3) **如何通信**。定义每个任务如何通信，任务从哪里取数据及往哪里送数据。

8.7 数据管理部分的设计

- 在传统的结构化设计方法中，很容易将实体-关系图映射到关系数据库中。
- 而在面向对象设计中，我们可以将UML类图看作是数据库的概念模型，但在UML类图中除了类之间的关联关系外，还有继承关系。
- 在映射时可以按下面的规则进行：

8.7 数据管理部分的设计

(1) 一个普通的类可以映射为一个表或多个表，当分解为多个表时，可以采用**横切和竖切**的方法。

- 竖切常用于实例较少而属性很多的对象，一般是现实中的事物，将不同分类的属性映射成不同的表。通常将经常使用的属性放在主表中，而将其他一些次要的属性放到其他表中。
- 横切常常用于记录与时间相关的对象，如成绩记录、运行记录等。由于一段时间后，这些对象很少被查看，所以往往在主表中只记录最近的对象，而将以前的记录转到对应的历史表中。

8.7 数据管理部分的设计

(2) 关联关系的映射

- **一对一关联的映射**：对于一对一关联，可以在两个表中都引入外键，这样两个表之间可以进行双向导航。也可以根据具体情况，将类组合成一张单独的表。
- **一对多关联的映射**：可以将关联中的“一”端毫无变化地映射到一张表，将关联中表示“多”的端上的类映射到带有外键的另一张表，使外键满足关系引用的完整性。
- **多对多关联的映射**：由于记录的一个外键最多只能引用另一条记录的一个主键值，因此关系数据库模型不能在表之间直接维护一个多对多联系。为了表示多对多关联，关系模型必须引入一个关联表，将两个类之间的多对多关联转换成表上的两个一对多关联。

8.7 数据管理部分的设计

(3) 继承关系的映射：通常使用以下两种方法来映射继承关系。

- **将基类映射到一张表，每个子类映射到一张表。在基类对应的表中定义主键，而在子类定义的表中定义外键。**
- **将每个子类映射到一张表，没有基类表。在每个子类的表中包括基类的所有属性。这种方法适用于子类的个数不多，基类属性比较少的情况。**

8.8 对象设计

- 对象设计过程包括使用模式设计对象、接口规格说明、对象模型重构、对象模型优化4组活动。

(1) 使用模式设计对象：设计者可以选择合适的设计模式，复用已有的解决方案，以提高系统的灵活性，并确保在系统开发过程中，特定类不会因要求的变化而被修改。

(2) 接口规格说明：在系统设计中所标识的子系统功能，都需要在类接口中详细说明，包括操作、参数、类型规格说明和异常情况等。

8.8 对象设计

(3) 对象模型重构：重构的目的是改进对象设计模型，提高该模型的可读性和扩展性。如将两个相似的类归并为一个类；将没有明显活动特征的类转为属性；将复杂的类分解为简单的类；重新组合类和操作来增进封装性和继承性等。

(4) 对象模型优化：优化活动是为了改进对象设计模型，以实现系统模型中的性能要求。包括选择更好的算法、提高系统执行的速度、更好地使用存储系统、减少连接中的重数来提高查询的速度、为了增加效率而增加额外的连接、改变执行的顺序等。

使用模式设计对象

- 设计模式包括4个要素：

(1) 名字：用来将一设计模式与其他设计模式区分开。

(2) 问题描述：用来描述该设计模式适用于何种情况。通常设计模式所解决的问题是对可更改性、可扩展性设计目标、以及非功能性需求的实现。

(3) 解决方案：描述解决该问题所需要的、结合在一起的类和接口的集合。

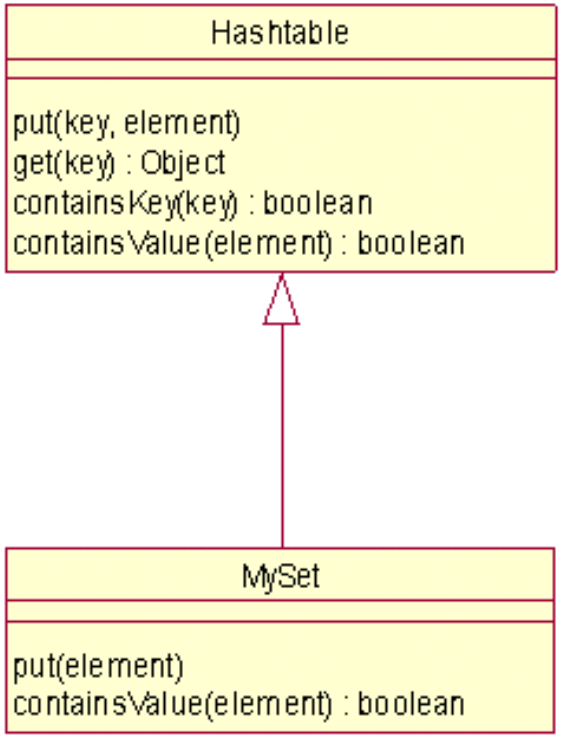
(4) 结果：描述将要解决设计目标的协议和可供选择的办法。

使用模式设计对象

- 设计模式中的继承
  - 在对象设计中，继承的核心是为了减少冗余和增加扩展性。
  - 通过将一些相似的类定义为子类，它们中的共有属性和操作集中放到继承结构的父类中，就能减少由于引入变化而引起的不一致。
  - 尽管继承能够让对象模型更加容易理解和修改，并具有更好的扩展性，但有些时候使用继承也会带来一些副作用，特别是，它增加了类之间的耦合性。

使用模式设计对象

- 例如，假设Java没有提供对集合的操作，现在编写Myset类来实现这些操作。
- 如果我们选择想继承java.util.Hashtable类，那么，在Myset类中插入一个新元素首先需要检查在表中是否存在一个表项，它的键值等于元素的键值，如果查找失败，新元素就可以插入到表中。
- 使用继承机制实现MySet类的类图如图所示。



使用继承机制实现MySet类

使用模式设计对象

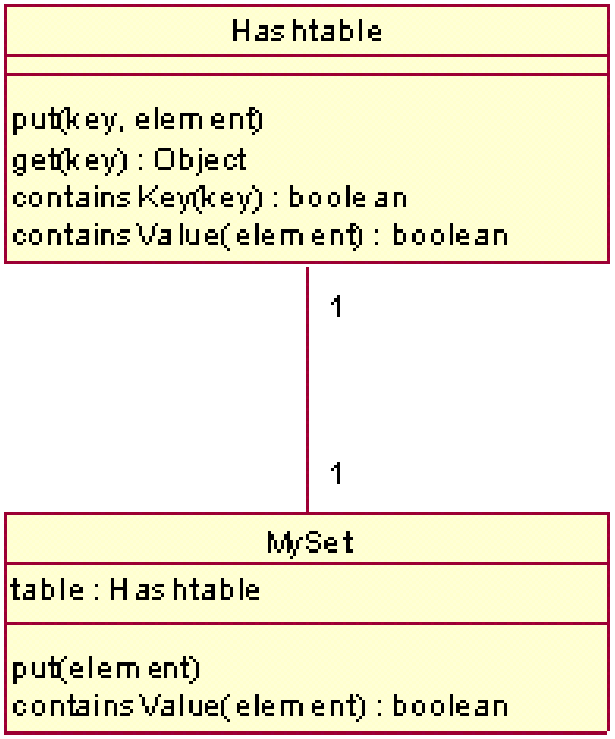
- 使用继承机制实现**MySet**的代码如下：

```
class MySet extends Hashtable {
        /*忽略构造方法*/
        MySet() {
        }
        void put(Object element) {
                if (!containsKey(element)) {
                put(element, this);
                }
        }
        boolean containsValue(Object element){
                return containsKey(element);
        }
        /*忽略其他方法*/
}
```

使用这种方式，让人们可以通过复用代码来实现他们所需的功能，但同样会提供一些他们并不需要的功能。

使用模式设计对象

- 授权
  - **授权是实现复用的另一种方法。A类授权B类，是指A类为了完成一个操作需要向B类发一个消息。**
  - **可以使用两个类之间的关联关系实现授权机制，在A类中增加一个类型为B的属性。**



使用授权机制实现 MySet 类

使用模式设计对象

- 使用授权机制实现**MySet**的代码如下：

```
class MySet {
        private Hashtable table;
        MySet() {
                table = Hashtable();
        }
        void put(Object element) {
                if (!containsKey(element)) {
                table.put(element, this);
                }
        }
        boolean containsValue(Object element) {
                return (table.containsKey(element));
        }
        /*忽略其他方法*/
}
```

使用模式设计对象

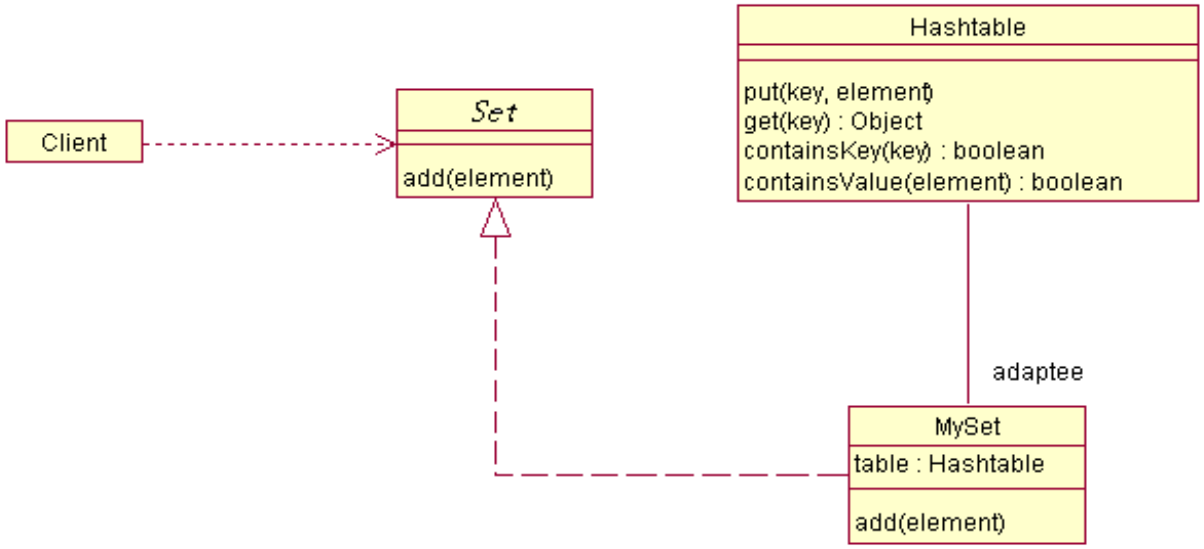
- 使用授权机制解决了我们在前面讨论的问题：

(1) 扩展性：使用授权机制实现的MySet类没有包含containsKey()操作，并且属性table是私有的。这样，如果MySet的内部实现使用的是链表而不是哈希表，也不会影响任何使用MySet类的地方。

(2) 子类型化：MySet类不是从Hashtable类继承来的，因此程序中的MySet对象不能被替换为Hashtable对象。以前使用Hashtable类的程序也不用改变。

使用模式设计对象

- 对于上面所讨论的问题，我们可以定义一个新类MySet，让MySet类遵从一个已存在的Set接口，并复用Hashtable类所提供的活动功能。
- 适配器（Adapter）设计模式就是解决这些问题的一个模板化方案。使用适配器设计模式解决MySet问题的方法如图所示。



接口规格说明设计

- 在对象设计过程中，要标识和求精解对象，以实现在概要设计期间定义的子系统。
- 对象设计的重点在于标识对象之间的界限。这个时期依然存在向设计中引入错误的可能性。
- 接口规格说明的目标是能够清晰地描述每一个对象的接口，这样开发者就能够独立地实现这些对象，以满足最小集成的需求。

接口规格说明设计

- 接口规格说明包括以下活动：

(1) 确定遗漏的属性和操作：在这个活动中，将检查每个子系统提供的服务及每一个分析对象，标识出被遗漏的操作和属性。需要对当前的对象设计模型求精，同时也对这些操作所用到的参数进行求精。

(2) 描述可见性和签名：在这个过程中，将决定哪一个操作对其他对象和子系统是可用的，哪一个操作仅仅是对本子系统是可用的。并说明操作的签名。这个活动的目标是减少子系统之间的偶合度，并提供一个较小且简单的接口，这个接口可被独立的开发者理解。

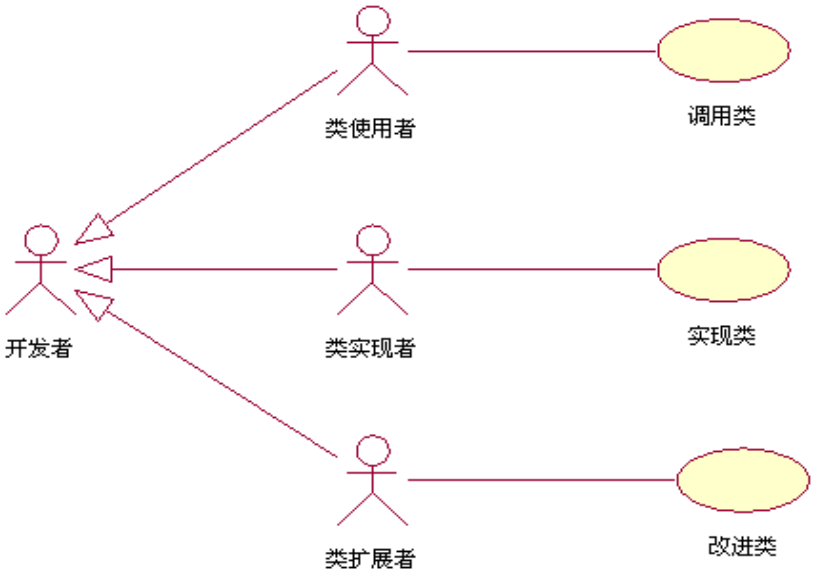
(3) 描述契约：描述每一个对象操作应该遵守的约束条件。契约包括不变式、前置条件和后置条件三种类型的约束。

接口规格说明设计

- 开发者角色的分类

➢ **在进一步研究对象设计和实现的细节时，则需要区分不同的开发者。**

➢ **当所有开发者都利用接口规格说明进行通信时，就需要从各自不同的观点去看待规格说明。如图所示。**



接口规格说明设计

- 契约。**是在一个类上定义的，确保有关该类的类实现者、类使用者、类扩展者都要遵守的假定条件。契约说明了类使用者在使用该类之前必须遵守的约束。**

➢ **契约包括3种类型的约束：**

(1) 不变式：不变式是对该类的所有实例而言都为真的谓词。不变式是和类或接口有关的约束。不变式通常用来说明类属性之间的一致性约束。

(2) 前置条件：是在调用操作之前，必须为真的谓词。前置条件和某个特定操作有关，用来说明类使用者在调用操作之前必须满足的约束。

(3) 后置条件：是在调用操作之后必须为真的谓词。后置条件与某个特定操作有关，用来说明类实现者和类扩展者在调用操作之后必须满足的约束。

重构对象设计模型

- 重构是对源代码的转换，在不影响系统行为的前提下，提高源代码的可读性和可修改性。
- 重构通过考虑类的属性和方法，达到改进系统设计的目的。

重构对象设计模型

- 典型的重构活动的例子包括：

- **将N 元关联转换成一组二元关联；**
- **将两个不同子系统中相似的类合并为一个通用的类；**
- **将没有明显活动特征的类转换为属性；**
- **将复杂类分解为几个相互关联的简单类；**
- **重新组合类和操作，增加封装性和继承性。**

8.9 对象设计

- 在对象设计期间，对象模型需要进行优化，以达到系统的设计目标，如最小化响应时间、执行时间或内存资源等。
- 一般情况下，系统的各项质量指标并不是同等重要的，设计人员必须确定各项质量指标的相对重要性（即确定优先级），以便在优化设计时制定折衷方案。
- 在进行优化时，设计者应该在效率与清晰度之间寻找平衡。
- 优化可以提高系统的效率，但同时会增加系统的复杂度。

优化对象设计模型

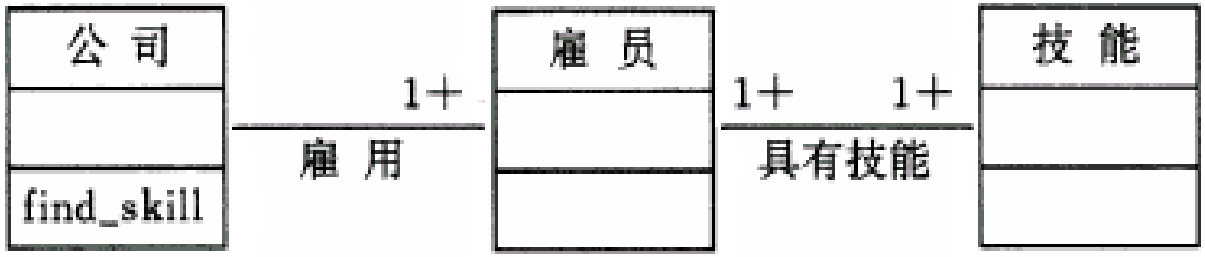
- 几种常用的优化方法包括：

(1) 增加冗余关联以提高访问效率

- **在面向对象设计过程中，当考虑用户的访问模式，及不同类型的访问彼此间的依赖关系时，就会发现，分析阶段确定的关联可能并没有构成效率最高的访问路径。**
- **下面用设计公司雇员技能数据库的例子，说明分析访问路径及提高访问效率的方法。**

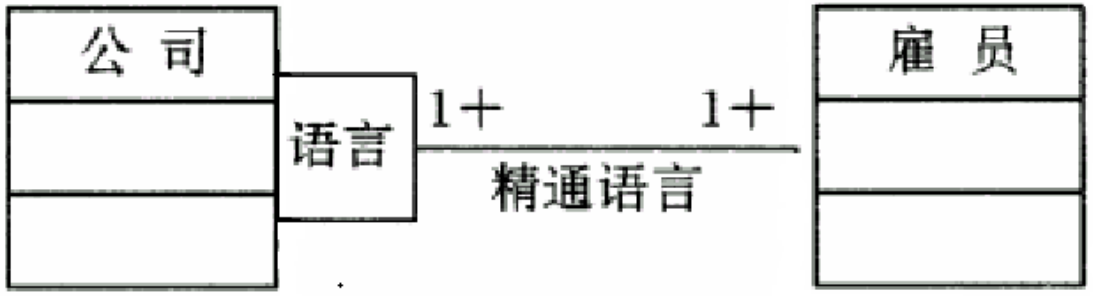
优化对象设计模型

- 公司、雇员及技能之间的关联链如图所示。
- 公司类中服务find_skill返回具有指定技能的雇员集合。例如，用户可能询问公司中会讲日语的雇员有哪些人。
- 假设某公司共有2000名雇员，平均每名雇员会10种技能，则简单的嵌套查询将遍历雇员对象2000次，针对每名雇员平均再遍历技能对象10次。如果全公司仅有5名雇员精通日语，则查询命中率仅有1/4000。



优化对象设计模型

- 针对上面的例子，我们可以增加一个额外的限定关联“精通语言”，用来联系公司与雇员这两类对象，如图所示。
- 利用冗余关联，可以立即查到精通某种具体语言的雇员，当然，索引也必然带来多余的内存开销。



优化对象设计模型

(2) 调整查询次序

- 例如，假设用户在使用上述的雇员技能数据库的过程中，希望找出既会讲日语，又会讲法语的所有雇员。
- 如果某公司只有5位雇员会讲日语，会讲法语的雇员却有200人，则应该先查找会讲日语的雇员，然后再从这些会讲日语的雇员中查找同时会讲法语的人。

(3) 保留派生属性

- 通过某种运算而从其他数据派生出来的数据，可以把这类冗余数据作为派生属性保存起来。

That's All!