**课程主要内容**

面向对象基本概念和原理

UML（ UnifiedModeling Language ）基础

面向对象设计原则

面向对象设计模式

**什么是面向对象**

面向对象的方法是一种分析方法、设计方法和思维方法。

面向对象方法学的出发点和所追求的基本目标是使人们分析、设计与实现一个系统的方法尽可能接近人们认识一个系统的方法。

使描述问题的问题空间和解决问题的方法空间在结构上尽可能一致

**第1章   面向对象基本思想**

对问题空间进行自然分割，以更接近人类思维的方式建立问题域模型，以便对客观实体进行结构模拟和行为模拟，从而使设计出的软件尽可能直接地描述现实世界。

构造出模块化的、可重用的、维护性好的软件，同时限制软件的复杂性和降低开发维护费用。

**什么是面向对象**

从程序设计方法的角度看，面向对象是一种新的程序设计范型(paradigm)，其基本思想是使用对象、类、继承、封装、聚合、关联、消息、多态性等基本概念来进行程序设计。

自八十年代以来，面向对象方法已深入到计算机软件领域的几乎所有分支。它不仅是一些具体的软件开发技术与策略，而且是一整套关于如何看待软件系统与现实世界的关系，用什么观点来研究问题并进行问题求解，以及如何进行系统构造的软件方法学。从这个意义上讲：

**面向对象方法是一种运用对象、类、继承、封装、聚合、关联、消息、多态性等概念来构造系统的软件开发方法。**

**基本思想**

**从现实世界中客观存在的事物出发来建立软件系统**

强调直接以问题域（现实世界）中的事物为中心来思考问题、认识问题，并根据这些事物的本质特征，把它们抽象地表示为系统中的对象，作为系统的基本构成单位。这可以使系统直接映射问题域，保持问题域中事物及其相互关系的本来面貌。

**充分运用人类日常的思维方法**

强调运用人类在日常的逻辑思维中经常采用的思想方法与原则，例如抽象、分类、继承、聚合、封装、关联等等。这使得软件开发者能更有效地思考问题，并以其他人也能看得懂的方式把自己的认识表达出来。

**Ø  主要特点：**

Ø  从问题域中客观存在的事物出发来构造软件系统，用对象作为对这些事物的抽象表示，并作为系统的基本构成单位。（对象）

Ø  用对象的属性表示事物的静态特征；用对象的服务（操作）表示事物的动态特征。（属性与服务）

Ø  对象的属性与服务结合为一体，成为一个独立的、不可分的实体，对外屏蔽其内部细节。（封装）

Ø  对事物进行分类。把具有相同属性和相同服务的对象归为一类，类是这些对象的抽象描述，每个对象是它的类的一个实例。（分类）

Ø  通过在不同程度上运用抽象的原则可以得到较一般的类和较特殊的类。特殊类继承一般类的属性与服务，从而简化系统的构造过程及其文档。（继承）

Ø  复杂的对象可以用简单的对象作为其构成部分。（聚合）

Ø  通过关联表达对象之间的静态关系。（关联）

总结：

用类和对象作为系统的基本构成单位。对象对应问题域中的事物，其属性与服务刻画了事物的静态特征和动态特征，它们之间的继承关系、聚合关系、消息和关联如实地表达了问题域中事物之间实际存在的各种关系。

因此，无论系统的构成成分，还是通过这些成分之间的关系而体现的系统结构，都可直接地映射问题域。

**UML**

图形化的建模语言

开发者用来为面向对象系统建立模型

具有灵活性与可扩展性

UML结构型图:识别对象和类、描述对象和类之间的关系(静态结构)

UML行为型图:对象如何协作产生满足要求的系统行为(动态性质)

**设计模式**

面向对象思想的发展使得在软件开发中使用模式化的方法受到了重视，模式化的思想来源于建筑业。

建筑大师Alexander:每一个模式描述了一个在我们周围不断重复发生的问题以及该问题解决方案的核心，这样你就可以一次又一次的使用该方案而不必做重复劳动。

就像建筑业用预制的墙和窗来构筑房屋，在面向对象设计中我们使用对象和接口代替墙和窗来构筑系统，它们的核心都在于提供了相关问题的解决方案。

模式化的设计使系统更稳定、易修改、易扩展、易理解、易测试。

90年代末，被称为四人帮的GoF发表了在面向对象编程中使用模式化方法研究的开创性著作-《设计模式-可复用面向对象软件的基础》(《Design Patterns-Elements of Reusable Object-Oriented Software》)。

其中提出了23种模式，这23种模式又可以被划分为三类：

创建型：与对象创建有关

结构型：处理类或对象组合

行为型：对象或类怎样交互和怎样分配职责

**第1章.   面向对象思想**

一种观察世界的方式

现实世界问题例。(济南男孩A送花给北京女孩B)

How?

**代理和团体**

解决问题的方法是找到一个合适的代理C，并把要求告诉他。

代理有责任完成你的需求。

A没必要理解C使用什么方法来完成任务，这些细节通常是隐蔽的。

一个面向对象的程序是由一个相互作用的代理团体组成，这些代理被称作对象。

每一个对象承担一个角色。

每一个对象都提供一种服务或者执行一种动作，以便为团体中其他对象服务。

**对象**

对象是独立存在的客观事物，它由一组属性和一组操作构成。

属性和操作是对象的两大要素。属性是对象静态特征的描述，操作是对象动态特征的描述。

属性一般只能通过执行对象的操作来改变。

操作又称为方法或服务，它描述了对象执行的功能。通过消息传递，还可以为其它对象使用。

**复合对象**

划分(partition)

聚合(aggregation)

**部分/整体**

部分/整体关系中有两种方式：组合和聚合。

PC机是一个组合的例子，一个部分对象（CPU）只能属于一个唯一的整体对象（PC机）。

组合关系中部分和整体的关系很紧密。

聚合关系中则比较松散，一个部分对象可以属于几个整体对象。

**消息和方法**

送花例

团体的成员通过传达要求来相互合作。

在面向对象编程中，行为的启动是通过将“消息”传递给对此行为负责的代理(对象)来完成的。

消息对行为的要求进行编码，并且随着执行要求所需的附加信息（参数）来一起传递。

“接收器”就是消息发送的对象。如果接收器接受了消息，那么同时它也接受了消息所包含的行为责任。然后，接受器响应消息，执行相应的“方法”以实现要求。

**消息传递与过程调用**

每一条消息都有一个指定的接收器（对象）相对应；接收器就是消息发送的对象。过程调用没有指定的接收器。

消息的解释由接收器决定，并且随着接收器的不同而不同。

动态绑定。 （动态决定 消息的解释是由父类还是子类的方法）

**信息隐藏**

作为某对象提供的服务的一个用户，只需要知道对象将接受的消息的名字。

不需要知道怎么完成要求，需要执行哪些动作。

在接收到一条消息后，对象会负责将该项任务完成。

**责任？？？**

用责任来描述行为。A对行为的要求仅表明他所期望的结果，C可随意选择使用的方法来实现所期待的目标，并在此过程中不受A的干扰。

提高了抽象水平，对象更加独立。（A想要排序，调用排序方法。C可以使用各种排序函数）

**不干预原则**

允许对象以任何 它认为合适的 不干涉其他对象的方式来完成任务，而不要干预它。

**类和实例**

C是花商的一个特例。

C是花商（Florist）类（class）的一个实例（instance）

**类**

根据抽象的原则对客观事物进行归纳和划分，只关注与当前目标相关的特征，把具有相同特征的事物归为一个类。它是一个抽象的概念。

类是具有相同属性和相同操作（服务）的对象的集合。它包括属性和操作。

**类和实例**

所有对象都是类的实例。

在响应消息时调用何种方法由类的接收器（对象/实例）来决定。

一个特定类的所有对象使用相同的方法来响应类似的消息。

每一个对象都是某个类的实例。类是一组相似的对象 。

类是对象相关行为的储存库(repository)。即同一个类的所有对象都能执行同样的动作。、

**类的层次**

除了知道C是花商外，还知道他是商人、人类、哺乳动物、物质对象。

在每一层次上，都可以了解特定的信息，这些信息适用于所有较低层次。

付款适用其他店主。

**继承**

类被组织成有单个根节点的树状结构，称为继承层次结构。与类实例相关的数据和行为都会被树结构中的后代自动继承。

在类层次结构中与某层相联系的信息（数据、行为）都会自动地提供给该层次结构的较低层次中。

继承表达了对象的一般与特殊的关系。

特殊类的对象具有一般类的全部属性和服务。

类可以组织成一个有层次的继承机构。

一个子类继承层次树中更高一层的父类的属性。

抽象父类是指没有具体实例的类，他只是用来产生子类。

**一般/特殊**

对象之间存在着一般和特殊的结构关系，也就是说它们存在继承关系。很多时候也称作泛化和特化关系。

**方法绑定与改写**

鸭嘴兽？

处理一般规则外的特例？

**改写**

将子类中某一方法取与父类方法相同的名称，结合寻找方法的规则（当响应特定消息时）实现改写。

**方法绑定与改写**

接收器搜索并执行相应的方法以响应给定的消息。

如果没有找到匹配的方法，搜索就会传导到此类的父类。搜索会在父类链上一直进行下去，直到找到匹配的方法，或者父类链结束。

如果能在更高类层次找到相同名称的方法，所执行的方法就称为改写了继承的行为。

**多态**

多态性是指一般类中定义的属性和服务，在特殊类中不改变其名字，但通过各自不同的实现后，可以具有不同的数据类型或具有不同的行为。

如不同花商送花方式。

多态可通过重载和类等级不同层次共享同一方法名字等方式来实现。

**JAVA中的重写/改写和重载**

重写一般是指父类和子类之间，子类重写了父类的一个方法，当然方法名是一样的，而且不能改变父类方法的返回值，比如说父类是返回String，子类重写了这个方法，想返回一个int，那是不行的，也得返回String。

重载是一个类里面，写了多了同名的方法，各个方法的返回值类型可以不一样。区分重载方法可以通过参数列表的个数，类型和顺序。

**第2章   抽象**

**抽象**

抽象是指对于一个过程或者一件制品的某些细节有目的的隐藏，以便把其他方面、细节或者结构表达得更加清楚。(信息隐藏)

抽象，是控制复杂性时最重要的工具。

老虎、狮子（猛兽）

**抽象实例：地图集**

如果打开一本地图集，一般看到的常是一幅世界地图。该地图只显示了一些最主要的特征，如主要的山脉、海洋等等，但细节基本上都被忽略了。

随后的一系列地图将覆盖小一些的地理区域，也能处理更多的细节。例如，一块大陆（如各大洲）的地图将包括国家的边界和主要的国家。更小的区域（如国家）地图，将包括城市、各山峰的名称。一个城市的地图可能会包括进出该城市的主要道路，再小一些的地图甚至还会画出一些建筑物。

**抽象技术**

每个抽象层次，包括了某些信息，也忽略了某些信息。

人们通常使用一些简单的工具来建立、理解和管理复杂的系统。其中最重要的技术称为“抽象”（abstraction）。

**抽象的层次**

在典型的OOP程序中，有许多级抽象。

更高层次的抽象部分地体现了面向对象程序面向对象的特征。

**确定正确的抽象级别**

在软件开发的早期，关键的问题就是确定合适层次的抽象。

既不要忽略太多的细节，也不要包括太多的细节。

**第3章   类和方法**

**封装**

利用数据抽象进行编程。

数据类型两个方面：外部用户/实现者

避免重复的代码

保护类受到不必要的修改

**概念**

实例

实例变量/数据成员/数据字段

对象=状态(实例变量)+行为(方法)

  对象外部看，客户只能看到对象的行为；对象内部看，方法通过修改对象的状态，以及和其他对象的相互作用，提供了适当的行为。

**类的定义**

可视性修饰符public,private

Java和C++中，封装性由程序员确定。

**惯例**

类名首字母大写。

数据字段private

Accessor(Getter)/Setter（访问器）

**类中声明次序建议-可读性**

先列出主要特征，次要的列在后面。

私有数据字段列在后面。

构造函数列在前面。

**定义和实现分离？**

Java,C#:方法主体直接放在类定义中。

C++:分离

**接口**

不提供实现

接口定义新类型，可以声明变量

类的实例可以赋值给接口类型变量

**向前定义**

多个互相引用的类(相互递归)

Java全文扫描、C++向前定义

Java 语言在生成代码之前会扫描整个文件，这样，如果程序文件的前面部分引用在文件后面声明的类，就不会产生任何冲突。其他的语言，例如 C++，会从前到后地依次处理程序文件中的类和方法去。名称在使用之前必须至少进行部分定义。

**类的数据字段（类属性）**

被一个类的所有实例共享的公共数据字段。

如何对该字段初始化？

每个实例都执行对公共数据字段的初始化？

没有实例执行初始化任务？

对象本身不对共享字段初始化。内存管理器自动将共享数据初始化为某特定值，每个实例去测试该特定值。第一个进行测试的做初始化。

Java和C++使用修饰符static创建共享数据字段。

Java:静态数据字段的初始化是在加载类时，执行静态块来完成

**静态成员**

实现同类多个对象间的数据共享。

**静态成员函数**

成员函数。不能访问非静态成员。（函数是静态的则访问的变量须是静态的）

无this（因为所有的变量都是共有的不存在特指）

构造和析构函数不能为静态成员。（对于每一个对象都有自己的构造函数和析构函数而不是公用同一个）

**第4章   消息、实例和初始化**

如何实例化？如何初始化？如何通过消息传递来联系？

**消息**

**消息传递**

使用消息传递（message passing）这一术语来表示请求对象执行一项特定行为的动态过程

消息总是传递给某个称为接收器的对象

响应消息所执行的行为不是固定不变的，它们根据接收器类的不同而不同

广播，可响应可不响应？？？

对象间相互请求或相互协作的途径。

**消息传递语法**

任何消息传递表达式都有 3 个确定的部分，包括接收器（receiver，消息传递的目的对象）消息选择器(message  selector，表示待传递的特定的消息文本)和用于响应消息的参数(argument)。

响应行为随接收器不同而不同。

**静态类型语言/动态类型语言**

静态：类型和变量联系在一起(Java,C++,C#,Pascal)

动态：变量看作名称标识，类型和数值联系在一起。(SmatllTalk,Python)

动态类型语言与静态类型语言之间的差异在于变量或数值是否具备类型这种特性。

焦点：高效性与灵活性

在消息传递这方面，静态类型语言和动态类型语言之间存在显著的差异。一方面，静态类型语言在编译时使用接收器的类型来检查选择器，以理解它所接收的消息。另一方面，动态类型语言在编译时则没有办法去核实这一消息。

**静态类型语言**

编译时作出内存分配决定。不必运行时刻重新分配。

控制类型错误。

PlayingCard aCard;

 aCard.someFunction();

**动态类型语言**

Function max(a,b){

       if(a<b)

              thenreturn b;

 return a;

}

**伪变量？？**

大多数面向对象语言中，接收器并不出现在方法的参数列表中，而是隐藏于方法的定义之中。只有当必须从方法体内部去存取接收器的数值时，才会使用伪变量

Java,C++:this

Eiffel:Current

Smalltalk,object-c:self

它不需要被声明，也不能被更改。

**对象自身引用**

this隐含指向调用成员函数的对象

**默认隐藏**

class PlayingCard {

...

       publicvoid flip () { setFaceUp( ! faceUp ); }

...

}

**等价**

class PlayingCard {

...

       publicvoid flip(){this.setFaceUp(!this.faceUp); }

...

}

**参数传递**

class QuitButton extends Button implementsActionListener {

public QuitButton () {

...

// install ourselves as a listener forbutton events

       addActionListener(this);//为这个Button注册ActionListener

}

...

};

**建议**

Java:构造函数中，使用this区分参数和数据成员。

**对象的创建**

变量声明与初始化结合

变量声明与创建分离

**创建对象的语法**

Java, C#

       PlayingCardaCard = new PlayingCard(Diamond, 3);

**对象数组的创建**

数组的分配和创建

数组所包含对象的分配和创建

Java:new仅创建数组。数组包含的对象必须独立创建。

PlayingCard cardArray[ ] = newPlayingCard[13];

for (int i = 0; i < 13; i++)

       cardArray[i]= new PlayingCard(Spade,i+1);

**构造函数**

初始化新创建对象。不论是基本类型，还是其他的类。java默认的初始化是最先发生的，位于一切方法之前。

优点:确保初始化之前不会被使用，防多次调用

Java/C++:名称，返回值

   可以通过检查与类显示的名称是否相同来识别构造函数与普通方法的区别。

   构造函数与普通方法的另外一个细微的差异就是构造函数不声明返回值的数据类型

Java和C#中数据字段可以初始化为特定的数值。

构造函数重载

**New语法**

PlayingCard cardSeven = new PlayingCard();// Java

**第5章   继承和替换**

花商中的收费行为。对其他商人都适用

**扩展与简化**

继承在程序语言中的含义如下：子类所具有的数据和行为总是作为与其相关的父类的属性的扩展( extension) （即更大的集合）。子类具有父类的所有属性以及其他属性。另一方面，由于子类是父类的更加特殊（或受限制）的形式，在某种程度上，子类也是父类的收缩( contraction)。

这种把继承作为一种扩展同时也作为一种收缩的思想，正是面向对象技术强大的原因，同时也会在正常的部署中引起混淆 。

**继承是向下传递的**

继承总是向下传递的，因此一个类可以从它上面的多个超类中继承各种属性。

如果Dog是Mammal的派生类，而Mammal又是Animal的派生类，则Dog不仅继承了Mammal的属性，同时也继承了Animal的属性。

派生类可以覆盖从基类继承来的行为。（重写/改写）

**继承的作用**

代码复用。

概念复用，共享方法的定义

**“是一个”检验**

通过(is-a)检验规则检验两个概念是否为继承关系。

如果检验概念 A 与概念 B 是否为继承关系，那么就尝试着套用这个英语语句：“A (n) A is a (n) B”，如果这个语句“听起来是对的”，那么这个继承关系很可能就是正确的。

**Public, Private and Protected**

父类中public和protected类型的成员，其子类能访问；父类中private类型的成员，其子类不能访问，只能自己类内部使用。

JAVA：

作用域 当前类 同包 子孙类 其他  
public    √     √    √    √  
protected √    √    √    X  
default   √    √    X    X   
private   √     X   X     X

在静态类型语言中

父类和子类数据类型的关系？

1.       子类实例必须拥有父类的所有数据成员。

2.       子类的实例必须至少通过继承实现父类所定义的所有功能。

3.       这样，在某种条件下，如果用子类实例来替换父类实例，那么将会发现子类实例可以完全模拟父类的行为，二者毫无差异。

**替换原则**

指如果类B是类A的子类，那么在任何情况下都可以用类B来替换类A，而外界毫无察觉。

**子类型**

指符合替换原则的子类关系。

区别于一般的可能不符合替换原则的子类关系

**改写**

子类有时为了避免继承父类的行为，需要对其进行改写

语法上：子类定义一个与父类有着相同名称且类型签名相同的方法。

运行时：变量声明为一个类，它所包含的值来自于子类，与给定消息相对应的方法同时出现于父类和子类。

改写与替换结合时，想要执行的一般都是子类的方法。

**改写机制**

Java、Smalltalk等面向对象语言，只要子类通过同一类型签名改写父类的方法，自然便会发生所期望的行为

**接口**

Java接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为（功能）。

与类一样，接口可以继承于其他接口，甚至可以继承于多个父接口。

虽然继承类和实现接口并不完全相同，但他们非常相似，因此使用继承这一术语来描述这两种行为。

**第6章   静态行为和动态行为**

面向对象语言的强大之处在于对象可以在运行时动态地改变其行为。

术语静态总是用来表示在编译时绑定于对象并且不允许以后对其进行修改的属性或特征。

术语动态用来表示直到运行时绑定于对象的属性或特征。

**静态类和动态类**

变量的静态类是指用于**声明变量**的类。静态类在编译时就确定下来，并且再也不会改变

变量的动态类指与变量所表示的当前数值相关的类。动态类在程序的执行过程中，当对变量赋新值时可以改变。

**静态类型和动态类型的区别**

对于静态类型面向对象编程语言，在编译时消息传递表达式的合法性不是基于接收器的当前动态数值，而是基于接收器的静态类来决定的。

**public static void** main(String args[])

    {

        Dog fido;

        fido = **new** Dog();

        fido.speak();  //will barkwoof

        fido.bark();   //will barkwoof

        Animal pet;

        pet=fido;      //legalto assign Dog to Animal

                       //但是必须要有 fido的实例化申明，reason：

        pet.Speak();   //will speakwoof!

        //pet.bark();      //The method bark() is undefined for the type Animal

                       //因为父类中没有子类的bark()方法

        //消息传递表达式的合法性不是基于接收器的当前动态数值，

        //而是基于接收器的静态类来决定的

    }

        Test a =**new** Test();

        Animal animal=**new** Animal();

        animal = **new** Dog();

        //animal.bark();//animal是Animal对象，没有bark()方法

        ((Dog)animal).bark();//将父类对象强制转化为子类就可以

**运行时类型决定**

替换原则可以通过提升数值在继承层次上的位置来体现。例如将 Dog 类型的数值赋值给类型为 Animal 的变量的同时，也将数值的类型从子类提升到父类。

有时则相反，还需要判断一种变量目前所包含的数值是否为类层次中的低层次类。例如，判断类型为 Animal 的变量所包含的数值是否为 Dog。

**向下造型（反多态）**

做出数值是否属于指定类的决定之后，通常下一步就是将这一数值的类型由父类转换为子类。这一过程称为向下造型，或者反多态，因为这一操作所产生的效果恰好与多态赋值的效果相反。（需要强制）

声明方式有问题

animal的静态类是Animal，

将其静态类强制转化为Dog，d中才可以访问bark()方法.

方案一：

Animalanimal = **new** Dog();

Dog d= (Dog) animal;

d.bark();

将animal动态绑定为子类；将animal的静态类强制转化为子类。

方案二：

Animalanimal;

Dog d= **new** Dog();

animal= d;

((Dog)animal).bark();

**方法绑定：**

静态方法绑定/动态方法绑定

响应消息时对哪个方法进行绑定是由接收器当前所包含的动态数值来决定的。

**多态变量**

如果方法所执行的消息绑定是由最近赋值给变量的数值的类型来决定，那么就称这个变量是多态的。

Java,Smalltalk等变量都是多态的。

**第7章   多态及软件复用**

Polymorphism

“多种形式”

**编程语言中的多态**

**可以有多种不同含义的名称**

**多态的形式**

f  重载：(也称为专用多态(ad hoc  polymorphism))用于描述一个函数名称（或者方法名称）有多种不同实现方式的情况。通常可以在编译时基于类型签名（方法参数类型、个数）来区分各个重载函数的名称。

f  改写：(或者包含多态(inclusion polymorphism))是重载的一种特殊情况。只发生在有关父类与子类之间关系的上下文中。改写方法具有相同名称、相同的类型签名。

f  多态变量(polymorphic  variable)：（或者赋值多态（assignment polymorphism））是指声明为一种类型，但实际上却可以包含另一种类型数值的变量。当多态变量用作参数时建立的函数将展现纯多态(pure polymorphism)。

f  泛型（或模板）提供了一种创建通用工具的方法，可以在特定的场合将其特化。

**重载：类型签名区分**

**改写:层次关系中，相同类型签名**

是重载的一种特殊情况，但是只发生在有父类和子类关系的上下文中

**多态变量（赋值多态）:声明与包含不同**

Parent p=new child();//declared asparent,holding child value

**软件复用机制**

通过通用的可复用的组件来构建软件？

最常用的软件复用机制：继承和组合。

为了说明这两项技术，我们使用已经存在的List类来构造一系列抽象。

Class List {

       List();

       voidadd (int);

       intfirstElement ( );

       intsize ( );

       intincludes (int);

       voidremove (int);

};

**“是一个”和“有一个”?**

如果A类有一个B类，那么自然地，B类的数据字段也应该是一个A类实例的一部分。另一方面，如果A类是一个B类，那么使用继承是正确的编码方式。

**组合**

提供了一种利用已存在的软件组件来创建新的应用程序的方法。

对象是数据（数据值）和行为的封装。在通过使用组合复用已存在的数据抽象来创建新数据类型时，新数据结构的部分状态只是已存在的数据结构的一个实例。

例：数据类型 Set 包含一个名称为theData且类型声明为 List 的实例字段。

**继承**

通过继承，新的类可以声明为已存在类的子类。通过这种方式，与初始类相关的所有数据字段和函数都自动地与新的数据抽象建立联系。

Class set:public List{

       public:

              //constructor

              Set();

              //operations

              voidadd(int);

              intsize();

};

FirstElement方法？

**组合和继承的比较**

组合是较为简单的一种技术。优点是清楚地指示出在特定的数据结构中需要执行哪些操作。无需考虑列表类所定义的所有操作。

继承无法知道一个方法是否可以合法地应用于集合。

使用继承构建数据抽象的代码的简洁性是继承的一个优点

*继承无法防止用户使用父类的方法来操纵新的数据结构*：FirstElement？？？

**重载**

Overloaded

语言中很多单词都是重载的，需要使用上下文来决定其确切含义。

重载是在编译时执行的，而改写是在运行时选择的。

重载是多态的一种很强大的形式。

非面向对象语言也支持。

**类型签名**

函数类型签名是关于函数参数类型、参数顺序和返回值类型的描述

多个过程（或函数、方法）允许共享同一名称，且通过该过程所需的参数数目、顺序和类型来对它们进行区分。即使函数处于同一上下文，这也是合法的。

class Example{

       //samename,three different methods

       intsum(int a){return a;}

       intsum(int a,int b){return a+b;}

       intsum(int a,int b,int c){return a+b+c;}

}

关于重载的解析，是在编译时基于参数值的静态类型完成的。

不涉及运行时机制

**造型：自动类型转换和强制类型转换**

**自动类型转换**

自动类型转换是一种隐式的类型转换，它发生在无需显式引用的程序中。

double x=2.8;

       inti=3;

       x=i+x;//integeri will be  converted to real

**强制类型转换**

强制类型转换表示程序员所进行的显式类型转换。

       x=((double)i)+x;

**x是y的父类**

上溯造型

**X a=new X();**

**Y b=new Y();**

**a=b;  //将子类对象造型成父类对象，相当做了个隐式造型：a = (X)b;**

下塑造型

**Xa=new X();**

**Yb=new Y();**

**Xa1=b；//先转化为一个子类**

**Yb1=(Y)a1 ；//将父类对象a1转化为子类对象**

**作为转换的替换**

类型转换：“类型的改变”，替换原则将引入一种传统语言所不存在的另外一种形式的类型转换。

发生在类型为子类的数值作为实参用于使用父类类型定义对应的形参的方法中时。

**函数的参数**

形参、实参

如果两个或更多的方法具有相同的名称和相同的参数数目，编译器如何匹配?

**匹配原则**

找到所有可能进行调用的方法，即各个参数可以合法地赋值给各个参数类型的所有方法。如果只找到一个在调用时可以完全匹配所使用的参数类型的方法，那么就执行这个方法。

如果第一步所产生的集合中，某个方法的所有参数类型都可以赋值给集合中的另一个方法的所有参数类型，那么就将第二个方法从集合中移走。重复以上操作，直至无法实现进一步的缩减为止。

如果只剩下一

个方法，那么这个方法就非常明确了，调用这个方法即可。如果剩余的方法不止一个，那么调用就产生歧义了，此时编译器将报告错误。

**Void order (Dessert d, Cake c);**

**Void order (Pie p, Dessert d);**

**Void order (ApplePie a, Cake c);**

**order (aDessert, aCake);//执行方法一**

**order (anApplePie , aDessert);//执行方法二**

**order (aDessert , aDessert);//错误**

**order (anApplePie , aChocolateCake);//执行方法三**

**order (aPie , aCake);//错误**

**order (aChocolateCake, anApplePie );//错误**

**order (aChocolateCake,aChocolateCake);//**

**order (aPie , aChocolateCake);//错误**

**改写**

如果子类的方法具有与父类的方法相同的名称和类型签名，称子类的方法改写了父类的方法。

与替换原则结合使用。

改写可看成是重载的一种特殊情况。

由于重载也涉及一个方法名称和两个或更多个方法体

**改写和重载的差异**

对于改写来说，方法所在的类之间必须符合父类/子类继承关系，而对于简单的重载来说，并无此要求。

如果发生改写，两个方法的类型签名必须匹配。（子类和父类必须具有相同的东西）

重载方法总是独立的，而对于改写的两个方法，有时会结合起来一起实现某种行为。

重载通常是在编译时解析的，而改写则是一种运行时机制。对于任何给定的消息，都无法预言将会执行何种行为，而只有到程序实际运行的时候才能对其进行确定。（编译时，各个变量的类型已确定，调用函数的参数、参数类型、顺序都已确定；但是是子类还是父类则有不同）

改写并不能改变方法的可存取性。

如果一个方法在父类中为public，那么不允许在子类中将该方法声明为private。反之亦然。

**代替与改进**

两种不同的关于改写的解释方式：

1.       代替（replacement）：在程序执行时，实现代替的方法完全覆盖父类的方法。即，当操作子类实例时，父类的代码完全不会执行。（普通函数）

2.       改进(refinement)：实现改进的方法将继承自父类的方法的执行作为其行为的一部分。这样父类的行为得以保留且扩充。（构造函数）

几乎所有的语言在构造函数中都使用改进语义。即，子类构造函数总是调用父类的构造函数，来保证父类的数据字段和子类的数据字段都能够正确地初始化化。

**class** Base{

**public** Base(**int** x){System.out.println("父类有参构造函数");}

}

**public class** Test **extends** Base{

**public** Test(){**super**(2);System.out.println("子类无参构造函数");}

**public** Test(**int** x){**super**(x);System.out.println("子类有参构造函数");}

**public static void** main(String args[]){

        Test t = **new** Test();//(1)

         Test t2 = **new** Test(2);//(2)

    }

}

由于改进的使用保证了父类的行为得以保留，使得父类所执行的行为也必然是子类行为的一部分，因此，通过这种机制所创建的子类几乎不可能不是子类型。由此，支持使用改进语义语言的人们认为这种机制非常优雅。而对于使用代替语义的语言来说则无法保证这一点。

**第8章   多态变量**

多态变量是指可以引用多种对象类型的变量。

这种变量在程序执行过程可以包含不同类型的数值。

对于动态类型语言，所有的变量都可能是多态的。

对于静态类型语言，多态变量则是替换原则的具体表现。

Parent variable=new Child();

**实际用法？？？**

很少使用赋值，通常是伴随着函数或方法调用，通过数值和参数之间的绑定来实现的。

**多态变量形式**

简单变量

接收器变量

纯多态(多态方法)

**简单多态变量**

布局管理器LayoutManager是一个接口

标准库为这个接口提供了几种不同的实现

通过调用继承自Component类的setLayoutManager方法，将参数赋值给本地多态变量

**接收器变量**

多态变量最常用的场合是作为一个数值，用来表示正在执行的方法内部的接收器。

隐藏

伪变量 smalltalk:self，C++,Java,C#:this

**多态变量在框架中的作用**

多态接收器功能的强大之处表现在消息传递与改写相结合时。这种结合是软件框架开发的关键。

一般框架系统中的方法分为两大类：

f  在父类中定义基础方法，被多个子类所继承，但不被改写；

f  父类定义了关于多种活动的方法，但要延迟到子类才实现。

由于基础方法被子类所继承，因此它们可以用于各个子类实例。

接收器变量多态性的展现。

f  当执行基础方法时，接收器实际上保存的是一个子类实例的数值。

f  当执行一个改写方法时，执行的是子类的方法，而不是父类的方法。

**第9章   框架**

对于一类相似问题的骨架解决方案。

通过类的集合形成，类之间紧密结合，共同实现对问题的可复用解决方案

继承和改写的强大能力体现

最常见的框架

f  Java中的GUI框架

f  Web开发中的Struts框架

**复用和特化**

框架开发的一个重要基础是以完全不同的两种方式来使用继承—复用、特化

类的方法可以分为两大类：

f  基本方法：它定义于父类，通过继承但却不对其改写的方式成为子类的一部分。体现了对问题的现存的解决方案。代码复用。

f  特化方法：为了适合于子类的特定环境，改变了其在父类中的行为。用于特定应用的解决方案。概念复用。

这些方法通常都是延迟的方法：定义于父类但在子类中实现的方法。

**高级抽象和低级抽象**

**例：雇员排序。**

雇员类                                      插入排序-根据工作年份

f  按照薪水排序？

f  按照姓名排序?

f  不再对雇员记录排序,对一个浮点数组排序?

源代码级的修改。

复用的是排序的思想，不是真正的实现。

封装改变

OO方案-排序框架

00方案-特化

基类不再需要改变。

特化子类满足不同的需求。

f  改变为对收入进行排序只需改变子类，无需改变父类

f  对浮点数进行排序也只需创建一个新的子类，而无需改变父类

**继承**

继承允许进行高级别算法细节的封装

还允许在不改变原始代码的情况下修改或特化这些细节。

**第10章        UML**

**UML简介**

UML 统一建模语言  (Unified Modeling Language)

**UML产生与发展**

70年代中期，公认的面向对象建模语言出现。

从1989年到1994年，其数量从不到十种增加到了五十多种。

90年代中，一批新方法出现了，其中最引人注目的是Booch 1993、OOSE和OMT-2等。

f  面对众多的建模语言，用户由于没有能力区别不同语言之间的差别，因此很难找到一种比较适合其应用特点的语言；

f  众多的建模语言实际上各有千秋；

f  虽然不同的建模语言大多类同，但仍存在某些细微的差别，极大地妨碍了用户之间的交流。

因此在客观上，极有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础上，取其精华，去其糟粕，求同存异，统一建模语言。

1994年10月，Grady Booch和Jim Rumbaugh开始致力于这一工作。他们首先将Booch 93和OMT-2 统一起来，并于1995年10月发布了第一个公开版本，称之为统一方法UM 0.8（United Method）。

1995年秋，OOSE 的创始人Ivar Jacobson加盟到这一工作。经过Booch、Rumbaugh和Jacobson三人的共同努力，于1996年6月和10月分别发布了两个新的版本，即UML 0.9和UML 0.91，并将UM重新命名为UML（Unified Modeling Language）

**UML内容**

**UML是一种可视化的建模语言，它能让系统构造者用标准的、易于理解的方式建立起能够表达出他们想象力的系统蓝图，并且提供一种机制，以以便于不同的人之间有效地共享和交流设计结果。**

**UML多种视图**

UML可以通过称为4+1视图模型的软件系统结构来了解。

五个视图并不对应于 UML 中描述的特定的形式构造或图

f  每个视图对应一个特定的研究系统的观点。

f  不同的视图突出特定的参与群体所关心的系统的不同方面，通过合并所有五个视图中得到的信息就可以形成系统的完整描述

f    
为了特殊的目的，只考虑这些视图的子集中包含的信息可能就足够了

用例视图(use case view)定义了系统的外部行为，是最终用户、分析人员和测试人员所关心的。该视图定义了系统的需求，因此约束了描述系统设计和构造的某些方面的所有其他视图。这正是用例视图具有中心作用的原因，也是通常所说的用例驱动开发过程。

设计视图（ design view）描述的是支持用例视图图中规定的功能需求的逻辑结构。它由程序构件的定义，主要是类、类所持有的数据、类的行为以及类之间交互的说明组成。

实现视图（ implementation view）描述构造系统的物理构件。这些构件不同于设计视图中描述的逻辑构件，这些构件包括如可执行文件、代码库和数据库等内容。这个视图中包含的信息与配置管理和系统集成这类活动  
有关。

进程视图（process view），涉及系统中并发性的问题。

部署视图（deployment view）描述物理构件如何在系统运行的实际环境（如计算机网络）中分布。

**UML表示法**

UML表示法定义UML符号的表示法。它为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。

模型通常作为一组图呈现给设计人员。图(Diagrams)是一组模型元素的图形化表示。不同类型的图表示不同的信息，一般是它们描述的模型元素的结构或行为。

各种图都有一些规则，规定哪些模型元素能够出现在这种图中以及如何表示这些模型型元素。

**符号与图形**

在UML元模型中定义了很多模型元素，如：Use Case、类、接口、组件等，为了模型的可视化，UML为每个模型元素规定的特定的图形符号来表示。

**活动者（Actor）**

活动者是作用于系统的一个角色或者说是一个外部用户。活动者可以是一个人，也可以是使用本系统的外部系统。

**用例（Use Case）**

用例是对活动者使用系统的一项功能的交互过程的陈述。

**用户进行登录的用例图**

**类（class）**

类（类）是具有相同属性和相同操作的对象的集合。

**接口（interface）**

接口是一种抽象类，它对外提供一组操作，但自己没有属性和方法（操作的实现），它是在没有给出对象实现的情况下对对象行为的描述。

接口使用对象类的图形表示方法，接口名前面加构造型《interface》。

**包（Package）**

包也是一种模型元素，可以把语义相近的模型元素组织在一个包里，增加对模型元素的可维护性。

**消息（Message）**

一般消息

返回消息

**注释（Comment）**

注释没有特定的语义，它用于对其他模型元素的补充说明。

**UML模型图**

客观世界是一个复杂的系统，需要从不同的角度来考察，才能真正理解这个系统。

为了能支持从不同角度来考察系统，标准建模语言UML定义了下列5类、共9种模型图：

f  用例图

f  静态图

对象图

类图

f  行为图

状态图

活动图

f  交互图

顺序图

协作图

f  实现图

构件图

部署图

**用例图**

从用户角度描述系统功能，并指出各功能的操作者。

**静态图**

包括类图、对象图和包图。

类图描述系统中类的静态结构。不仅定义系统中的类,表示类之间的联系如关联、依赖、聚合等,也包括类的内部结构(类的属性和操作)。

类图描述的是一种静态关系,在系统的整个生命周期都是有效的。

**类图**

类图是用类和它们之间的关系描述系统的一种图示。

类、对象和它们之间的关联是面向对象技术中最基本的元素。类模型和对象模型揭示了系统的结构

**类**

类（class）是具有相同属性和相同操作的对象的集合。

**类的名称**

类的命名应尽量用应用领域中的术语,应明确、无歧义,以利于开发人员与用户之间的相互理解和交流。

类的获取是一个依赖于人的创造力的过程,必须与领域专家合作,对研究领域仔细地分析,抽象出领域中的概念,定义其含义及相互关系,分析出系统类,并用领域中的术语为类命名。

一般而言,类的名字是名词。

**类的属性**

原则上来说,类的属性应能描述并区分每个特定的对象;

只有系统感兴趣的特征才包含在类的属性中;

系统建模的目的也会影响到属性的选取。

**属性**

+（public）,-(private),#(protected)

**操作**

+（public）,-(private),#(protected)

**类之间的关系**

事物之间相互联系的方式，无论是逻辑上的还是物理上的，都被建模为关系。

f  依赖（Dependency）

f  关联（Association）

f  聚合（Aggregation）

f  组合（Composition）

f  泛化（Generalization）

f  实现（Realization）

**泛化**

泛化用于表示对象之间一般和特殊的结构关系。

通常所说的继承（特殊个体 iskind of 一般个体）关系；指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力。

UML中用带空心箭头的实线线表示Generalization关系，箭头指向一般个体。

**泛化关系**

是一般事物和该事物的较为特殊的种类之间的关系。

一个类可以有0-多个父类。

把没有父类且至少有一个子类的类称为根类或基类，把没有子类的类称为叶子类

**实现（Realization）**

实现是指一个模型元素（如：类）是另一个模型元素（如：接口）的实现。

具体来说，可以指一个class类实现interface接口（可以是多个）的功能；实现是类与接口之间最常见的关系；在Java中此类关系通过关键字implements明确标识

UML中用空心箭头和虚线表示Realize关系，箭头指向定义约定的元素。

**关联（Association）**

关联就是类或对象之类链接的描述。

元素间的结构化关系，是一种弱关系，被关联的元素间通常可以被独立的考虑。

关系表现为变量(has a )。类与类之间的联接，它使一个类知道另一个类的属性和方法。

UML中用实线表示Association关系，箭头指向被依赖元素

**关联关系**

是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。

恰好连接两个类的关联称为二元关联。

多于两个类的关联称为n元关联。

关联在Java中是用实例变量实现的。

关联可以有方向，可以是单向关联，也可以是双向关联。

l  双向关联：两个类都知道另一个类的公共属性和操作。

l  单向关联：只有一个类知道另外一个类的公共属性和操作。大多数关联应该是单向的，单向关系更容易建立和维护，有助于寻找可服用的类。

关联表现在代码层面，为被关联类B以 类属性 的形式出现在关联类A中，

也可能是关联类A引用了一个类型为被关联类B的全局变量

在使用类图表示关联关系时可以在关联线上标注角色名，一般使用一个表示两者之间关系的动词或者名词表示角色名（有时该名词为实例对象名），关系的两端代表两种不同的角色，因此在一个关联关系中可以包含两个角色名，角色名不是必须的，可以根据需要增加，其目的是使类之间的关系更加明确。

**单向关联**

在一个登录界面类LoginForm中包含一个JButton类型的注册按钮loginButton，它们之间可以表示为关联关系，代码实现时可以在LoginForm中定义一个名为loginButton的属性对象，其类型为JButton。

 此时的loginButton（JButton）是LoginForm的成员变量

**双向关联**

顾客(Customer)购买商品(Product)并拥有商品，反之，卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系。

此时，两个类中，都含有彼此的成员变量。

**依赖（Dependency）**

依赖表示两个或多个模型元素之间语义上的关系 。

在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。

依赖(Dependency)关系是一种使用关系，表现为函数中的参数(use a)。是类与类之间的连接，表示一个类依赖于另一个类的定义，其中一个类的变化将影响另外一个类。例如如果A依赖于B，则B多体现方法的参数，也可以表现为局部变量，或静态方法的调用。

是一种使用关系，它说明一个事物规格说明的变化可能影响到使用它的另一事物，但反之未必。

**整体/部分关联**

**聚合（Aggregation）**

聚合关系用于表示对象之间部分和整体关系，但关系比较松散。

关联关系的一种特例，表示部分和整体（整体 has a 部分）的关系。

UML中用带空心菱形头的实线表示Aggregation关系，菱形头指向整体。

在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。

例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系。

在代码实现聚合关系时，成员对象通常作为构造方法、Setter方法或业务方法的参数注入到整体对象中。构造方法注入

聚合是关联的特例。如果类与类之间的关系具有部分与整体（part-whole）的的特点，则把这样的关联成为聚合

聚合在Java中是用实例变量实现的。

关联与聚合仅从Java语法分辨不出。

组合（Composition）

组合关系用于表示对象之间部分和整体关系，关系很紧密。

在UML中，组合关系用带实心菱形的直线表示。

在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。例如：人的头(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系

在代码实现组合关系时，通常在整理类的构造方法中直接实例化成员类。

**关联、组合、聚合**

f  关联和聚合的区别主要在语义上

关联的两个对象之间一般是平等的，例如你是我的朋友

聚合则一般不是平等的，例如一个公司包含了很多员工，其实现上是差不多的。

f  聚合和组合的区别则在语义和实现上都有差别

组合的两个对象之间其生命期有很大的关联，被组合的对象是在组合对象创建的同时或者创建之后创建，在组合对象销毁之前销毁。一般来说被组合对象不能脱离组合对象独立存在，而且也只能属于一个组合对象，例如一个文档的版本，必须依赖于文档的存在，也只能属于一个文档。

聚合则不一样，被聚合的对象可以属于多个聚合对象，例如一个员工可能可以属于多个公司。

**UML类图**

**用例图**

用例( use case)是从用户的观点对系统行为的一个描述。

用例图展现了一组用例、参与者以及它们间的关系。

可以用用例图描述系统的静态使用情况。

在对系统行为组织和建模方面，用例图是相当重要的。

用例图中包含三种模型元素：系统、角色、用例。

**用例图—系统**

系统是用例模型的一个组成部分。它可以代表一台机器或者一个业务活动。

系统的边界用来说明构建的用例模型的应用范围。

**系统边界**

例如一台自助式售货机(被看作系统)应提供售货、供货、提取销售款等功能。这些功能在自动售货机之内的区域起作用，自动售货机之外的情况不考虑。

**用例图—角色(Actor)**

角色是与系统交互的人或者事物。角色代表一个群体，而不具体的指某一个体。

**用例图—用例(Use Case)**

用例代表一个完整的功能。是对一组动作序列的描述，显示了系统是如何被使用。

用例的特征：

  1. 用例总由角色初始化。

  2. 用例为角色提供值。

  3. 用例具有完整性。

**第11章        面向对象得设计原则**

提高面向对象设计复用性的设计原则

**设计目标**

f  可扩展性(Extensibility):新功能易加入系统。

f  灵活性(Flexibility):允许代码修改平稳发生，不会涉及很多其他模块。

f  可插入性(Pluggability):容易将一个类换为另一个具有同样接口的类。

**软件复用重要性**

f  较高的生产率

f  较高的软件质量

f  恰当使用复用,可改善系统的可维护性

**面向对象设计**

f  使一个系统可在更高的层次上提供了可复用性

f  抽象化和继承:使概念和定义可复用

f  多态:使实现和应用可复用

f  抽象化和封装:可保持和促进系统的可维护性

**复用**

抽象层次是一个应用系统作战略性判断和决定的地方，那么抽象层次就应当是较为稳定的，应当是复用的重点。

复用的焦点不再集中在函数和算法等具体实现细节上，而是集中在最重要的含有宏观商业逻辑的抽象层次上。

既然如果抽象层次的模块相对独立于具体层次的模块的话，那么具体层次内部的变化就不会影响到抽象层次的结构，所以抽象层次的复用就会较为容易。

**面向对象复用**

面向对象设计中，可维护性复用是以设计原则和设计模式为基础的。

**面向对象设计原则**

1.开闭原则OCP:Open-Closed Principle

2.里氏替换原则LSP:LiskovSubstitution Principle

3.依赖倒转原则DIP:Dependency InversionPrinciple

4.接口隔离原则ISP:InterfaceSegregation Principle

5.组合复用原则CRP:Compositoin ResusePrinciple

6.迪米特法则LoD:Law ofDemeter

7.单一职责原则（SRP)

**开-闭原则OCP**

软件组成实体应该是对扩展可扩展的，但是对修改是关闭的。( Software Entities Should BeOpen For Extension， But Closed For Modification)

开放-封闭法则认为应该试图去设计出永远也不需要改变的模块。

关键在于抽象化:可给系统定义一个一劳永逸，不再更改的抽象设计，此设计允许有无穷无尽的行为在实现层被实现。抽象层预见所有扩展。

一个软件系统的所有模块不可能都满足OCP，但是应该努力最小化这些不满足OCP的模块数量。

**OCP例-类**

内存折扣？

**依赖倒转原则（DIP）**

抽象不应当依赖于细节，细节应当依赖于抽象。

针对接口编程，而非实现

**使用接口的优点**

Client不必知道其使用对象的具体所属类。

一个对象可以很容易地被(实现了相同接口的)的另一个对象所替换。

对象间的连接不必硬绑定(hardwire)到一个具体类的对象上，因此增加了灵活性。

松散耦合 (loosens coupling)。

增加了重用的可能性。

提高了(对象)组合的机率，因为被包含对象可以是任何实现了一个指定接口的类。

**针对接口编程**

不将变量声明为某个特定的具体类的实例对象，而让其遵从抽象类定义的接口。实现类仅实现接口，不添加方法。

(Dra w(shape\*p)  不要Cricle\*p  Rectangle \*p Triangle \*p)

DIP可应用于任何存在一个类向另一个类发送消息的地方。

**组合复用原则**

组合优先原则：优先使用(对象)组合，而非(类)继承

**组合优点**

f  容器类仅能通过被包含对象的接口来对其进行访问。

f  “黑盒”复用，因为被包含对象的内部细节对外是不可见。

f  封装性好。

f  实现上的相互依赖性比较小。

f  每一个类只专注于一项任务。

f  通过获取指向其它的具有相同类型的对象引用，可以在运行期间动态地定义(对象的)组合。

**组合缺点**

从而导致系统中的对象过多。

为了能将多个不同的对象作为组合块(compositionblock)来使用，必须仔细地对接口进行定义。

**继承**

(类)继承是一种通过扩展一个已有对象的实现，从而获得新功能的复用方法。

泛化类(超类)可以显式地捕获那些公共的属性和方法。

特殊类(子类)则通过附加属性和方法来进行实现的扩展

**继承优点**

容易进行新的实现，因为其大多数可继承而来。

易于修改或扩展那些被复用的实现。

**继承缺点**

破坏封装性，因为这会将父类的实现细节暴露给子类。

“白盒”复用，因为父类的内部细节对于子类而言通常是可见的。

当父类的实现更改时，子类也不得不会随之更改。

从父类继承来的实现将不能在运行期间进行改变。

**小结**

组合与继承都是重要的重用方法

在OO开发的早期，继承被过度地使用

随着时间的发展，我们发现优先使用组合可以获得重用性与简单性更佳的设计

当然可以通过继承，以扩充可用的组合类集。

因此组合与继承可以一起工作

但是我们的基本法则是：优先使用对象组合，而非(类)继承

**第12章        设计模式**

**模式**

设计模式的思想根源是基本原则的宏观运用,本质上是没有任何模式的

发现模式的人永远是大师,而死守模式的人,最多只能是一个工匠

模式就是对以解决问题的记录，使处理后面与之类似的问题时更容易。

设计模式使人们更加简单方便地复用成功的设计和体系结构。

**设计模式**

每一个模式描述了在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该解决方案而不必重复劳动”

尽管软件技术发展非常快，但是仍然有非常多的设计模式可以让我们套用

设计模式可以帮助人们简便地复用以前成功的设计方案，提高工作效率式

**设计模式要素**

**模式名称（pattern name）**

一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一

**问题(problem)**

描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

**解决方案(solution)**

描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

**效果(consequences)**

描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

**设计模式**

创建型：对象创建

结构型：处理类或对象组合

行为型：对类或对象怎样交互和怎样分配职责进行描述

**设计模式－工厂模式**

定义简单的工厂

**Bridge(桥梁)模式**

**设计原则：组合优先**

f  继承复用的优点：

       可以很容易的修改或扩展父类的实现

f  继承复用的缺点：

       继承破坏封装，因为父类的实现细节完全暴露给子类(白盒复用)

父类的实现发生改变，则子类必受牵连

继承是静态的，不能在运行时发生改变，不灵活

f  组合复用的优点

不破坏封装，这种复用是黑盒复用，因为成员对象的内部细节对新对象保密

所需依赖少(只依赖接口)

动态的，可以把成员对象动态替换为另一个类型相同的对象

f  组合复用的缺点

对象数量会增加

使用委托（delegation）会使系统复杂

当需要应对变化的时候，应该首先使用组合的方式，而不是继承。因为组合更加灵活

例1：

汽车有很多种，小轿车、货车、客车，有的车是客货两用，有的车水陆两用

如果使用继承来描述：

一旦增加新的汽车种类或用途，都需要大量改动原有代码

使用“组合”思路考虑问题

“汽车”拥有某种或某些“用途”

“汽车”和“用途”独立变化，互不影响

区分“Is-A”与“Has-A”

有一个系统需要描述经理、雇员和学生

它们都是人，所以：

*换一个角度看*

*雇员、经理、学生都是角色的一种*

*人 拥有自己的角色*

***设计原则：封装可变性***

发现代码容易变化的部分，封装之，使它和不容易变化的部分独立开来。

设计原则：开-闭原则

在设计一个软件的时候，应当是这个软件可以在不被修改的前提下扩展

使用桥梁模式的效果

       当需求改变的时候（增加动物或行为），只需要简单添加几个类。对原有代码不需要改动。保证了代码的稳定，提高了可维护性。

结构

意图

将抽象部分与它的实现部分分离，使它们都可以独立地变化

适用性

抽象和它的实现部分可以独立变化

类的抽象以及它的实现都可以通过生成子类的方法加以扩充

实现部分的修改不会对客户产生影响

**Strategy（策略）模式**

策略模式：封装了一系列算法，使得它们可以相互替换

效果：算法可以独立变化