面向对象开发技术

林子童 徐卫霞
孙吉鹏 杜泽林
谷一滕 张晓敏
鲍伟

目录

第一章 面向对象的引入和发展

为什么会有面向对象？为什么面向对象有这么多东西？

我们都知道编程语言中都会有基本数据类型，比如 int，char 等等，它们作为描述一些常用的从事物中**抽象出来的符号**如数字，字符等的表达方式，被定义在程序语言中，数学可以证明这些符号配合上三种基本的逻辑结构（顺序，选择，循环）可以表示出任意大且复杂的程序思想。我们可以操作这些符号比如数字做加法，字符做拼接等等，来实现相应的表达目的。那么面向对象的设计初衷就来了，设计者想把自己定义的描述符号（小狗类这个符号来描述小狗），变成一种数据类型，即人人都能够像使用基本数据类型那样使用这种自己定义的数据类型。

Now we talk.

想想我们怎么描述一个事物呢，比如描述一只小狗，我们的直觉告诉我们应该把小狗的一些特征属性抽象出来，比如毛色，大小，叫声等等，把他们放在一起，包装成一个统一的东西，这样我们就可以使用它了。

你会发现历史真的就是这么过来的，跟你想的一模一样，在 C 语言里，这种数据类型叫做结构（Structure），它就是这样一种把所有属性都包装在一起的一种东西，它就是我们面向对象的雏形。

为了能让别人按照自己定义好的规则安全地使用这种数据类型不会出现危险操作（如把小狗的身长设成 3 米），作为定义这个数据结构的设计者，应该怎么做呢？我们应该规定好查询，修改这种数据类型属性的操作，只有通过这些我设计好的操作才能改变属性，比如只有通过我提供的 grow()方法才能改变小狗的长度，而 grow()里有长度不能超过 1.5 米的 if else 限制。

明白了么，类，属性，方法，对象，封装……

你是设计者，为了让你设计的数据类型功能丰富，使用安全，你必须得做这么些工作，为了完成更复杂的需求，还需要发展出继承，组合等设计手段。这就是为什么要学这么多内容，因为没有这些没法实现你提供的数据类型的稳定使用。

当我们明白我们所做之事是种必需和使命时，我们就不会迷茫。

以后的章节内容高效实用，没有废话。

第二章 面向对象的基本概念和程序设计

这一部分涉及到面向对象基本的概念，概念、规则多，是之后内容的基础，考试中主要以概念简述题出现，需要结合例子理解好每个概念。

编者解疑： 孙吉鹏 杜泽林 谷一滕 林子童 徐卫霞

2.1 基本概念

2.1.1 对象 Object

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。

2.1.2 类 Class

对象按照不同的性质划分为不同的类 。同类对象在数据和操作性质方面具有共性，把一组对象的共同特性加以抽象并存贮在一个类中。类是对象之上的抽象，有了类之后，对象则是类的具体化，是类的实例 。类是静态概念，对象是动态概念

2.1.3 方法 Method

定义于某一特定类上的操作与规则，具有同类的对象才可为该类的方法所操作。这组方法表达了该类对象的动态性质，而对于其它类的对象可能无意义，乃至非法规则，说明了对象的其他特征之间是怎样联系的，或者对象在什么条件下是可行的方法也称作行为（behavior）

就是之前提的操作啦。

2.1.4 消息 Message

对另一个对象的操作在于选择一对象并通知它要作什么，该对象“决定”如何完成这一任务。在其所属类的方法集合中选择合适的方法作用于自身 。所谓“操作一个对象”并不意味着直接将某个程序作用于该对象 ， 而是利用传递消息，通知对象自己去执行这一操作 ， 接收到消息的对象经过解释，然后予以响应 。发送消息的对象不需要知道接收消息的对象如何对请求予以响应 。

消息就是从外部告诉一个对象做什么操作的过程，对象名.方法名。

2.1.5 封装 Encapsulation

所有信息都存贮在对象中，即其数据及行为都封装在对象中。影响对象的唯一方式是执行它所属的类的方法即执行作用于其上的操作 。

信息隐藏（information hiding），将其内部结构从其环境中隐藏起来。要是对对象的数据进行读写，必须将消息传递给相应对象，得到消息的对象调用其相应的方法对其数据进行读写 。当使用对象时，不必知道对象的属性及行为在内部是如何表示和实现的，只须知道它提供了那些方法（操作）即可 。

封装绝对是一门学问，它涉及到一个设计者和一个使用者的本质区别，外部只用知道接口就可以而内部设计者却需要知道如何去实现。可以说没有封装就谈不上面向

对象，怎么让外部人员只看到应该或者需要看的东西，为了实现这个目标，面向对象类中都定义了诸如 public，private 等区别的可视性修饰符（visibility modifier）。

2.1.6 继承（Inheritance）

继承是一种使用户得以在一个类的基础上建立新的类的技术。新的类自动继承旧类的属性和行为特征，并可具备某些附加的特征或某些限制。新类称作旧类的子类，旧类称作新类的超类。继承机制的强有力之处还在于它允许程序设计人员可重用一个未必完全符合要求的类，允许对该类进行修改而不至于在该类的其它部分引起有害的副作用，是其它语言所没有的。

为什么会有继承这个概念？设计者为了让自己定义的类变得像基本数据类型那样功能强大而且适应现实问题的解决，就不得不定义比基本数据类型更多的类型来解决问题，这是必然的，因为一个类不能既对特殊问题针对性强又同时适用于很多场景，这就会导致解决一个问题会需要很多类，那么问题来了，设计者怎样能既省事又能做尽量少的工作呢？答案就是继承。继承可以重用代码，不影响父类并且扩展性还特别好，是实现把类变成“数据类型”的编程捷径。

2.1.7 多态（Polymorphism）与重载（Overriding）

在收到消息时对象要予以响应，不同的对象收到同一消息可以产生完全不同的结果。多态是指能够在不同上下文中对某一事物（变量、函数或对象）赋予不同含义或用法。多态一般分为继承多态、重载多态、模板多态。重载多态和模板多态是静态多态，即多态行为是在编译期决定的，而继承多态是一种动态多态的形式，即多态行为可以在运行时动态改变 。

多态具体内容在 2.6 节讨论。

2.1.8 动态联编（Dynamic Binding）

联编（binding）是把一个过程调用和响应这个调用而需要将执行的代码加以结合的过程。联编在编译时刻进行的叫静态联编（static binding），动态联编则是在运行时（run time）进行的，因此，一个给定的过程调用和代码的结合直到调用发生时才得以进行，因而也叫迟后联编（late binding）。

一般程序通过编译之后编译器就会把源代码翻译成机器代码，但是面向对象程序中因为继承的机制所以会导致父类的指针可以指向子类，而调用时两类都有相同的方法甚至签名（给函数传递参数的类型，数目，顺序），所以编译器也不知道实际在调用谁的函数，所以也就没法翻译成一成不变的机器码了。

2.2 对象

一个对象是一个实体，一个类的实例化形式，一个类作为一种数据结构，应该提供两方面功能，一方面是能保存当前自己的状态，就是自己的属性或者说数据，另一方面是给使用者规定定义好查询、修改这些属性的方法。

2.2.1 对象的特性（Property）

对象的属性和方法称作对象的特性。

2.2.2 对象标识（Object Identifier）

（1）缩写为 OID，是将一个对象和其它对象加以区别的标识符。一个对象标识和对象永久结合在一起，不管这个对象状态如何变化，一直到该对象消亡为止。

（2）面向对象程序设计语言中的 OID，强调对象标识的表达能力。用变量名充当标识 。可寻址性和标识这两个概念做了混合。强类型变量，像 C++,JAVA 中

Employ emp=new Employ();

非强类型的变量

var emp=new Employ()

（3）直接标识和间接标识：

- 直接标识就是变量的值即为要标识的对象
- 间接标识指变量的值不是要标识的对象而是该对象的指针

2.2.3 复合对象(composite object)

指一个对象的一个属性或多个属性引用了其他对象，复合对象也称作复杂对象。

说一个对象 O1 引用了另外一个对象 O2，意味着 O1 的一个属性的值是 O2，O1 中引用 O2 的属性的值是 O2 的对象标示

委托 delegation

是复合对象的一个特例，在委托方式下可有两个对象参与处理一个请求，接受请求的对象将责任委托给它的代理者。

实现的方法就是在当前类 A 中在声明一个类 B 的对象作为属性，类 A 中的方法 f（）的实现使用类 B 的方法来完成。比如

```
class A{
private:
B b  = new B();//声明一个类 B 的 b 对象作为属性
public :
int f (int m, int n){
return b.g(m,n);// 类 A 的 f（）方法的实现只是委托给了 B 的 g（）方法
}
}
class B{
public:
int g(int m, int n){
return m+n;
}
}
```

组合（composition）

一个对象可以由其它对象构造，如发动机，车轮，车身三个类组合成车这个类，具体实现就是车这个类里声明三个类的对象完成操作。

聚合（aggregation）

描述对象间具有相互关系的另一种方式，例如在家庭关系中，将男人、女人和孩子组合在一起建立起一个聚合关系称作“家庭”

组合和聚合的区别

组合，语义规则：a-part-of 是更强形式的聚合，零件和整体的感觉。

聚合是同质的，语义规则：has-a 部分可以脱离整体而存在，摄影协会和会员的感觉 。

现在我们已经很少区分这两种关系的区别，新版 uml 图标准中两者已合成一种，都是用一端为菱形的线段表示。

2.2.4 对象持久化

概念：要长久保存的对象，也就是持久对象（persistent object）

特点：①持久对象不随着创建它的进程结束而消亡

②在外存中存贮

2.3 类和实例

（一）两个角度理解类的概念

①类是具有相同或相似行为或数据结构的对象的共同描述。
②类是一组具有相同属性特征的对象的抽象描述。
第一个对类的描述强调了类在面向对象程序设计中的作用。第二个对类的描述是抽象的概念，是根据生活中的“种类”演化而来。例如，我们把货车、轿车和公交车都统称为汽车，是因为他们具有相同属性特征。

（二）类的性质

①类名：在同一个系统环境中，类的名能够唯一标识一个类。
②成员集合：属性、方法和方法操作的接口。

（三）类的实例

（注：面向对象方法与技术-Part1-OO 原理.ppt page77 学生类有助理解相关概念）
①一个实例是从一个类创建而来的对象。类描述了这个实例的行为（方法）及结构（属性）。每个实例可由该类上定义的操作（方法）来操纵。
②概括类与对象的关系：类是对象的抽象，对象是类的实例。
③同一个类的不同实例：

1. 具有相同的数据结构
2. 承受的是同一方法集合所定义的操作，因而具有相同的行为
3. 同一个类的不同实例可以持有不同的值，因而可以有不同的状态

2.4 类与面向对象的程序设计语言

2.4.1 类的定义

（一）实例：

```
Class <类名>   {
      private:
              <数据成员>
              <成员函数>
      protected:
              <数据成员>
              <成员函数>
      public
              <数据成员>
              <成员函数>  }
    [<对象名表>]
```

```
class Employee {
     private:
        char *Name;
        int  Age;
    public:
      void  Change (char
*name,int age);
      void  Retire();
      Emloyee (char
*name,int age);
      ~Employee ();
    } ;
```

（二）惯例和建议：

①惯例：类名首字母大写；
数据字段 private；
Accessor(Getter)/Setter/Is

②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

③类主题的变化：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

（三）可视性修饰符——public,private

封装性由程序员确定，外部用户不用关心具体实现。

（四）静态成员——类的数据字段/类属性

①定义：静态成员，声明为 static 的类成员或者成员函数便能在类的范围内共同享，我们把这样的成员称做静态成员和静态成员函数。

②性质：静态成员是属于类的而不是对象的；静态数据成员的值在函数退出时不消失，作为下一次调用时的初值，所以可在各对象之间传递数值。

③举例理解：汽车制造商为统计汽车的产量，可以在在汽车类 car 类中增加用于计数的静态数据成员变量，比如在 car 类中声明一个 static int number;初始化为 0。这个 number 就能被所有 car 的实例共用。在 car 类的构造函数里加上 number++，在 car 类的析构函数里加上 number--。那么每生成一个 car 的实例，number 就加一，每销毁一个 car 的实例（汽车报废），number 就减一，这样，number 就可以记录在市场上 car 的实例个数。

④静态成员函数的规范：不能访问非静态成员；无 this/不能使用 this 引用；构造和析构函数不能为静态成员。

2.4.2 类的使用

①两种方式： 允引和继承
允引：A 类是 B 类的用户
继承：A 类是 B 类的后代
②实现允引的两个方法：
用户和供应商（Client and supplier）：在 B 类中声明 A 类的实例。
功能调用（Feature call）：在 B 类中调用 A 类的方法。
注：类的继承作为重要概念，在以后会有详细的介绍。

2.4.3 对象的创建

（一）创建对象的语法

①C++：PlayingCard * aCard = new PlayingCard(Diamond, 3);
②Java,C#：PlayingCard aCard = new PlayingCard(Diamond, 3);
③Smalltalk：aCard <- PlayingCard new.

（二）以数组为例理解对象的创建与赋值

创建一个包含有效数据的数组分两部分：数组的分配和创建；数组所包含对象的分配和创建。
①数组的分配和创建：new 仅创建数组。数组包含的对象必须独立创建。相当于在内存中为数组申请一块空间，这个空间是数据的仓库，仓库中没有货物，是混乱无序的。
②数组所包含对象的分配和创建：为数组所包含的元素赋值。相当于在数组的仓库里放置有价值的货物。
实例：

```
PlayingCard cardArray[ ] = new PlayingCard[13];//数组的分配和创建
for (int i = 0; i < 13; i++)//数组所包含对象的分配和创建
      cardArray[i] = new PlayingCard(Spade,i+1);
```

（三）构造函数和析构函数

①构造函数的作用：初始化新创建对象
优点:确保初始化之前不会被使用，防多次调用
②构造函数重载
实例：

```
class PlayingCard {
public:
PlayingCard()
{ suit = Diamond; rank = 1; faceUp = true; }
PlayingCard(Suit is)
{ suit = is; rank = 1; faceUp = true; }
PlayingCard(Suit is, int ir)
{ suit = is; rank = ir; faceUp = true; }
};
```

类的构造函数可以有不同的签名，根据实际情况调用。
③缺省构造函数

在 C++的一个类中，如果构造函数没有参数，或者构造函数的所有参数都有默认值，就可以称其为缺省构造函数。一个类中，只能有一个缺省构造函数。
*缺省构造函数在何时被调用？比较常见的情况：
①对象被定义时无参数，形如：MyClass mc;
②在派生类的构造函数中未显示调用基类的构造函数，此时基类的缺省构造函数会被调用。

（四）NEW 语法

```
PlayingCard cardSeven = new PlayingCard(); // Java
PlayingCard *cardEight = new PlayingCard; // C++
```

*C++ new 创建对象和直接声明创建对象的区别
①程序内存：
编译器把内存分为三个部分：
1.静态存储区域：主要保存全局变量和静态变量。生存期：整个程序。
2.堆：存储动态生成的变量。生存期：自己来决定。
3.栈：存储调用函数相关的变量和地址等。生存期：所处的语句块（既{}的范围）
②声明创建对象——Myclass myclass;
此种创建方式，对象是被创建在栈上的，使用完后不需要手动释放，该类析构函数会自动执行。调用这个对象的成员变量和成员函数时用“.”操作符。例如
myclass.value;
③new 对象——Myclass *myclass = new Myclass();
通过 new 创建的实例返回的是对象指针（myclass 指向一个 Myclass 的对象），同时在堆上为它分配空间，并且需要显式的释放空间，delete 对象的时候才会调用对象的析构函数。
因为是指针的操作，所以调用这个对象的成员变量和函数时要用“->”例如 myclss->function();

（五）拷贝构造函数

当一个类没有自定义的拷贝构造函数的时候系统会自动提供一个默认的拷贝构造函数，来完成复制工作。
实例：

```
class Test
{
public:
    Test(int temp)
    {
        p1=temp;
    }
protected:
    int p1;

};
void main()
{
    Test a(99);
    Test b=a;
}
```

注：稍后会详细比较拷贝与克隆的区别。

（六）常数值初始化

①直接初始化：final int max = 100;
②在构造函数中初始化：

```
class PlayingCard {
public PlayingCard ( )
{ suit = Diamond; rank = 1; faceUp = true; }
```

```
public PlayingCard ( int is, int ir)
{ suit = is; rank = ir; faceUp = true; }
...
public final int suit; // suit and rank are
public final int rank; // immutable
private boolean faceUp; // faceUp is not
}
```

*const 与 final 的区别

①const：完全不允许改变。

②final：仅断言相关变量不会赋予新值，并不能阻止在对象内部对变量值进行改变。

实例：

```
class Box {
public void setValue (int v);
public int getValue () { return v; }
private int v = 0;
}
void main(){
final aBox = new Box(); // aBox 申明为 final
aBox.setValue(8); //可以调用方法改变其内部变量的值。
aBox.setValue(12);
}
```

Final 虽说也是常量，但是它有一定的灵活性，对象的“壳”不允许改变，但是它的“瓤”可以改变。

（七）析构函数

①内存回收

C++：使用 new 创建的对象在堆上为它分配空间，并且需要显式的释放空间，delete 对象的时候才会调用对象的析构函数。

Java,C#,Smalltalk：垃圾回收机制，时刻监控对象的操作，对象不再使用时，自动回收其所占内存。通常在内存将要耗尽时工作。

②实例

```
class Trace {
public:
Trace (string t){test = t;}//构造函数
~Trace ()//析构函数
private:
string text;
};
```

2.4.4 对象结构

（一）简单类型与引用类型

①简单类型：int a = 10;变量保存的是数据

②引用类型：int *a = 10;变量保存的是数据地址，a 指向 10 所在的地址。

（二）对象同一与对象相等

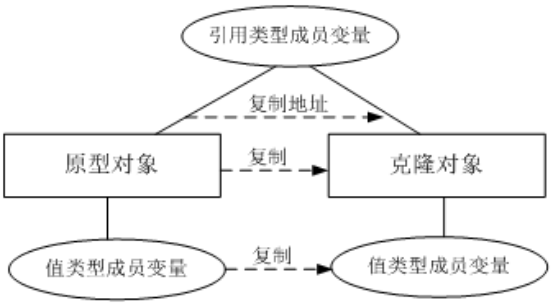
简单的讲：对象相等就是两个对象，它们的值相等。对象同一就是指两个对象引用的是否为同一个对象。
值相等是大家普遍理解的意义上的相等：它意味着两个对象包含相同值。例如，两个值为 2 的整数具有值相等性。
引用相等意味着要比较的不是两个对象，而是两个对象引用，这两个对象引用所引用的是同一个象。

（三）拷贝与克隆

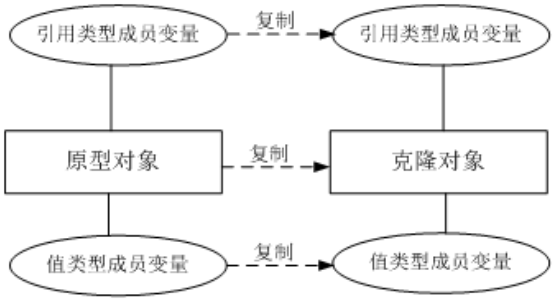
①克隆的语言环境支持：
Java 的 Object 类中包含一个 clone()方法，因为每个类直接或间接的父类都是 Object，因此它们都含有 clone()方法，但是因为该方法是 protected，所以都不能在类外进行访问。要想对一个对象进行复制，就需要对 clone 方法覆盖。
②克隆的优势：
我们常见的 Object a=new Object();Object b;b=a;这种形式的代码复制的是引用，即对象在内存中的地址，a 和 b 对象仍然指向了同一个对象。而通过 clone 方法赋值的对象跟原来的对象时同时独立存在的。

③浅克隆与深克隆

浅克隆：
在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。
简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制。



深克隆：
在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。
简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制。



实例：（这个实例比较长，请大家耐心阅读）

```
package clone;
class UnCloneA {
  private int i;
  public UnCloneA(int ii) { i = ii; }
  public void doublevalue() { i *= 2; }
  public String toString() {
    return Integer.toString(i);
  }
}
class CloneB implements Cloneable{
  public int aInt;
  public UnCloneA unCA = new UnCloneA(111);//new 创建返回地址
  public Object clone(){
    CloneB o = null;
    try{
      o = (CloneB)super.clone();
    }catch(CloneNotSupportedException e){
      e.printStackTrace();
    }
    return o;
  }
}
public class CloneMain {
  public static void main(String[] a){
    CloneB b1 = new CloneB();
    b1.aInt = 11;
    System.out.println("before clone,b1.aInt = "+ b1.aInt);
    System.out.println("before clone,b1.unCA = "+ b1.unCA);

    CloneB b2 = (CloneB)b1.clone();
    b2.aInt = 22;
    b2.unCA.doublevalue();
    System.out.println("================================");
    System.out.println("after clone,b1.aInt = "+ b1.aInt);
    System.out.println("after clone,b1.unCA = "+ b1.unCA);
    System.out.println("================================");
    System.out.println("after clone,b2.aInt = "+ b2.aInt);
    System.out.println("after clone,b2.unCA = "+ b2.unCA);
  }
}
/** RUN RESULT:
before clone,b1.aInt = 11
before clone,b1.unCA = 111
================================
after clone,b1.aInt = 11
after clone,b1.unCA = 222
================================
```

```
after clone,b2.aInt = 22
after clone,b2.unCA = 222
*/
```

输出的结果说明 int 类型的变量 aInt 和 UnCloneA 的实例对象 unCA 的 clone 结果不一致，int 类型是真正的被 clone 了，因为改变了 b2 中的 aInt 变量，对 b1 的 aInt 没有产生影响，也就是说，b2.aInt 与 b1.aInt 已经占据了不同的内存空间，b2.aInt 是 b1.aInt 的一个真正拷贝。相反，对 b2.unCA 的改变同时改变了 b1.unCA，很明显 b2.unCA 和 b1.unCA 是仅仅指向同一个对象的不同引用！从中可以看出，调用 Object 类中 clone()方法产生的效果是：先在内存中开辟一块和原始对象一样的空间，然后原样拷贝原始对象中的内容。对基本数据类型，这样的操作是没有问题的，但对非基本类型变量，我们知道它们保存的仅仅是对象的引用，这也导致 clone 后的非基本类型变量和原始对象中相应的变量指向的是同一对象。

大多时候，这种 clone 的结果往往不是我们所希望的结果，这种 clone 也被称为“影子 clone”。要想让 b2.unCA 指向与 b2.unCA 不同的对象，而且 b2.unCA 中还要包含 b1.unCA 中的信息作为初始信息，就要实现深度 clone。

默认的克隆方法为浅克隆，只克隆对象的非引用类型成员。

2.4.5 类对象

类本身也是一个**对象**，其属性和方法称之为**类属性**和**类方法**，是被称作为**元类**的特殊类的实例，元类是用来描述类的类。

元类的优点：

概念上一致：只用对象的概念，就可表述系统中的所有成分

使类成为运行时刻一部分，有助于改善程序设计环境

继承的规范化 ：类与元类的继承采用双轨制

对象类所具有的行为：

创建实例，返回类名称，返回类实例大小，返回类实例可识别消息列表。

类的操作：

获取类对象：

C++ typeinfo aClass = typeid(AVariable);

Delphi Pascal aClass := aVariable.ClassType;

Java Class aClass = aVariable.getClass();

Smalltalk aClass <- aVariable class

获取父类：

Class parentClass = aClass.getSuperclass(); // Java

parentClass <- aClass superclass //Smalltalk

获取类的字符串名称：

char * name = typeinfo(aVariable).name(); // C++

String internalName=aClass.getName();//Java

String descriptiveName=aClass.toString();

检测对象类:

Child *c_dynamic_cast<Child *>(aParentPtr);

if (c!=0){ … } //C++

if (aVariable instanceof Child) …

if (aCalss.isInstance(aVariable)) … //Java

如果下一步执行的行为要依赖于多态变量所包含的特定类型：

解决方案:建立一个新方法，子类特定行为移到子类中完成，缺省行为移到父类中完成。这样做不仅代码数量减少，而且也减少了潜在的出错机会。假如要增加一个新的子类，只需要保证这个类实现正确的接口即可。

类加载器示例:

ClassLoader 主要用于加载类文件，利用反射（newInstance())生成类实例

假设有类 A 和类 B，A 在方法 amethod 里需要实例化 B

其中一种方式就是使用 ClassLoader

```
ClassLoader cl=this.getClass.getClassLoader();//Step 1. Get ClassLoader
Class cls = cl.loadClass("com.rain.B");//Step 2. Load the class
B b = (B)cls.newInstance(); //Step 3. new instance
```

反射（内省）：

是指程序在运行过程中“了解”自身的能力。反射工具都开始于一个对象，该对象是关于一个类的动态（运行时）体现，其支持一个组件动态的加载和查询自身信息，因此反射为许多基于组件的编程工具建立了基础。

反射（内省）技术分类：

获取理解当前计算状态的特征。

用于修改的特征：增加新的行为。

反射（内省）特征的应用：

程序中的错误定位方法

灵活调用对象的方法

反射（内省）的操作：

得到某个对象的属性:

```
public Object getProperty(Object owner, String fieldName) throws
Exception {
    Class ownerClass = owner.getClass();
    Field field = ownerClass.getField(fieldName);
    Object property = field.get(owner);
    return property;
}
```

得到某个类的静态属性:

```
public Object getStaticProperty(String className, String fieldName)
throws Exception {
    Class ownerClass = Class.forName(className);
    Field field = ownerClass.getField(fieldName);
    Object property = field.get(ownerClass);
    return property;
}
```

执行某对象的方法:

```
public Object invokeMethod(Object owner, String    methodName,
Object[] args) throws Exception {
        Class ownerClass = owner.getClass();
        Class[] argsClass = new Class[args.length];
        for (int i = 0, j = args.length; i < j; i++) {
                argsClass[i] = args[i].getClass();
        }
        Method method = ownerClass.getMethod(methodName,
argsClass);
        return method.invoke(owner, args);
}
```

执行某个类的静态方法:

```
public Object invokeStaticMethod(String className, String
methodName,Object[] args) throws Exception {
        Class ownerClass = Class.forName(className);
        Class[] argsClass = new Class[args.length];
        for (int i = 0, j = args.length; i < j; i++) {
                argsClass[i] = args[i].getClass();
        }
        Method method = ownerClass.getMethod(methodName,
argsClass);
        return method.invoke(null, args);
}
```

新建实例:

```
public Object newInstance(String className, Object[] args) throws
Exception {
        Class newoneClass = Class.forName(className);
        Class[] argsClass = new Class[args.length];
        for (int i = 0, j = args.length; i < j; i++) {
                argsClass[i] = args[i].getClass();
        }
        Constructor cons = newoneClass.getConstructor(argsClass);
        return cons.newInstance(args);
}
```

判断是否为某个类的实例:

```
public boolean isInstance(Object obj, Class cls) {
        return cls.isInstance(obj);
}
```

得到数组中的某个元素:

```
public Object getByArray(Object array, int index) {
```

```
            return Array.get(array,index);
        }
```

2.4.6 消息传递

必要条件：（两个实体之间）

它们之间至少存在一条通道

遵守同一种通信协议

语法：消息-〉接收器

要求：

发送一条消息时，应指明信道或给出信道的决定方法，最常用的是用接收方的标识（如名字）来命名信道。

发送消息的流程：（A 向 B 发送）

对象 A 要明确知道对象 B 提供什么样的服务

根据请求服务的不同，对象 A 可能需要给对象 B 一些额外的信息，以使对象 B 明确知道如何处理该服务

对象 B 也应该知道对象 A 是否希望它将最终的执行结果以报告形式反馈回去

消息传递语法:

C++, C#, Java,Python, Ruby

```
aCard.flip ();
aCard.setFaceUp(true);
aGame.displayCard(aCard, 45, 56);
```

Pascal, Delphi,Eiffel, Oberon

```
aCard.flip;
aCard.setFaceUp(true);
aGame.displayCard(aCard, 45, 56);
```

Smalltalk

```
aCard flip.
aCard setFaceUp: true.
aGame display: aCard atLocation: 45 and: 56.
```

伪变量:

Java,C++:this（this 隐含指向调用成员函数的对象）

Eiffel:Current

Smalltalk,object-c:self

就好像在使用同类的实例。

参数传递:

```
lass QuitButton extends Button implements ActionListener{
    public QuitButton () {
        ...
        addActionListener(this);
    }
    ...
};
```

2.5 类和继承

概念：

在已有的类的基础上建立新的类的方法
重复使用和扩展那些经过测试的以有的类，实现重用
可以增强处理的一致性（是一种规范和约束）

作用：

代码复用
概念复用,共享方法的定义
‘is a’的检验（继承关系为‘is a’）

元类的继承层次：

类的继承关系与相应的元类的继承关系是平行的
如果 B 是 A 的子类，则 B 的元类也是 A 的元类的子类
object 是根类（无超类），而其相应的元类 objectclass 还有一个抽象超类 class

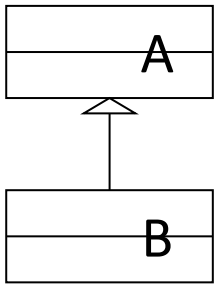
2.5.1 超类（superclass）和子类(subclass)

概念：

已存在的类通常称作超类
新的类通常称作子类
子类不仅可以继承超类的方法，也可以继承超类的属性
如果超类中的某些方法不适合与子类，则可以重置这些方法

定义：

如果类 B 能使用类 A 中的方法及属性，称 A 是 B 的超类，B 是 A 的子类，也称类 B 继承类 A



超类和子类的关系：

子类实例必须拥有父类的所有数据成员。
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。

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。

2.5.2 继承的传递性

定义:

如果类 C 是类 B 的派生类，而类 B 又是类 A 的派生类，则类 C 不仅继承了类 B 属性，同时也继承了类 A 的属性。称类 C 为类 B 的直接超类，类 C 为类 A 的间接超类。（派生类可以覆盖其超类的行为）

2.5.3 单继承

定义：

如果一个类只有一个直接超类。

特点：

构成类之间的关系是一棵树。

2.5.3 多继承

定义：

如果一个类有多于一个的直接超类。

特点：

构成类之间的关系是一个网格。

问题：

对于多个不同的超类的同名方法，同名属性会有二义性。

解决方案：

C++规定，多继承时，直接超类的构造函数的调用次序是:

（1）抽象超类。若有多个抽象超类，按继承说明次序从左到右。

（2）非抽象超类：若有多个非抽象超类，按继承顺序从左到右。

2.5.4 子类是否允许使用父类的属性和方案

访问权限修饰符：public private protected

区别：

public：可以被所有其他类所访问。

private：只能被自己访问和修改

protected：自身，子类可以访问。

2.5.5 泛化和特化

泛化定义：

通过将若干类的所共享的公共特征抽取出来，形成一个新类，并且将这个类放到类继承层次的上端以供更多的类所重用。抽取出的超类称作抽象类（抽象类一般没有实例）

特化定义：

新类作为旧类的子类。

抽象类特点：

抽象类不能创建实例

抽象类没有实例

接口和抽象类：

接口：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。

抽象类：创建实例前，子类必须实现父类的抽象方法。

抽象方法：介于类和接口之间的概念。定义方法但不实现。

2.5.6 替换原则

替换条件：子类实例必须拥有父类所有的数据成员，必须至少通过继承实现父类所定义的所有功能，这样在某种条件下才可以用子类实例来代替父类实例，子类可完全模仿父类行为，二者毫无差别。

可替换性：变量声明时指定的类型不必与它所容纳的值类型相一致。

对于类 A、B，若 B 是 A 的子类，则在任何条件下都可用类 B 来替换类 A。

2.5.7.重置（改写，overriding）：修正从超类继承下来的属性及方法。

实现：

语法上：子类定义一个和父类有相同名称且类型签名相同的方法（签名：方法名、方法的参数列表），而 C++中除了以上，还需要父类使用关键字 Virtual 来表明

运行时：变量声明为一个类，它所包含的值来自子类，与给定消息相对应的方法同时来自父类、子类（联想多态理解）。重置机制是基于动态联编实现的。

重定义：操作的表示和操作的实现体将被修改。

2.5.8.静态类和动态类

静态：用来表示在编译时绑定与对象并且不允许以后对其进行修改的属性或特征。

动态：用来表示直到运行时绑定与对象的属性或特征。

变量的静态类：用于声明变量的类，在编译时就确定下来，并且再也不会改变。

变量的动态类：与变量所表示的当前数值相关的类，在程序执行过程中对变量赋新值时可以改变

PS：对于静态类型，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。

向下造型（反多态）：做出数值是否属于指定类的决定之后，通常下一步就是讲这一数值的类型由父类转换为子类，这一过程叫做向下造型（也叫反多态，因为它所产生的效果与多态赋值恰好相反）。

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决定的

2.5.9.软件复用机制

常见：继承、组合

组合：利用已存在的软件组件来创建新的应用程序，委托，非替换，‘has-a’。

继承：‘is-a’

继承 VS 组合：

继承：is-a 关系。

优点：子类可以重写父类的方法来实现对父类的扩展；代码简洁。

缺点：①父类的内部细节对子类是可见的；② 子类从父类继承的方法在编译时就确定下来了，所以无法在运行时改变从父类继承的方法的行为；③父类和子类是一种高耦合，父类的方法修改后相应的子类的方法必须随之修改。

组合：has-a 关系，设计类的时候要把组合的类的对象加入到该类中作为自己的成员变量。

优点：①当前对象必须通过所包含的成员变量对象来调用成员对象的方法，其内部细节对当前对象不可见；②低耦合关系，若修改成员对象的类的代码不需要修改当前类的代码；③当前对象可以在运行时动态绑定所包含的对象。

缺点：①容易产生过多的对象②为了能组合多个对象，必须仔细对接口进行定义。

对继承和组合的使用：

例：已存在类

class Eye {void look()…}

Class Nose {void smell()…}

Class mouth {void speak()…}

Class ear {void listen()…}

若要创建类 Head，使其具有以上四种行为，选择继承？组合？

若选择继承，则 Head 类同时继承这四个类，则这四个类中的变量等也会被继承，而这些变量、方法等有一些是 Head 类用不到了，但是 head 类也会继承，这是没必要的。故继承不是一种优雅的实现。

若选择组合，head 类中定义这四个类的对象，分别调用它们的方法就可实现这四种行为，不会有不必要的变量、方法等，所以使用组合是合适的。

PS：选择继承，还是组合取决于具体情况，若 A 类中有一个 B 类，那么使用组合，若 A 类是一个 B 类，那么使用继承（is-a，has-a 的区别）。

2.5.10.继承的形式

特殊化继承：

很多情况下都是为了特殊化才使用继承的。在这种情况下，新类是基类的一种特定类型，能满足基类的所有规范。这种方式创建的总是子类型，并符合可替换性原则。

子类型：

强调新类具有父类一样的行为（未必是继承），是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；子类可以扩展父类功能，但不能改变父类原有功能；子类可实现父类的抽象方法，但不能覆盖父类的非抽象方法；子类可以增加自己特有的方法。

子类：

说明新类是继承自父类。

里氏替换原则：

新类继承父类时，除添加新的方法完成新功能外，尽量不要重写父类的方法，也尽量不要重载父类的方法。

规范化继承：

用于保证派生类和基类具有某个共同的接口（所有的派生类实现了具有相同方法界面的方法）

子类只是实现父类定义但未实现的方法，没有实现新的方法。

例：Java 中关键字 abstract 确保必须构建派生类，否则无法创建该类的实例。

特殊化继承和规范化继承构成了继承最理想的方式

构造继承：只是为了代码复用，新创建的子类通常都不是子类型，这称为构造子类化，它经常违反替换原则。新类和基类间可能不存在抽象概念上的相关性。

泛化继承：对基类已存在的功能进行修改或扩展，形成一种更泛化的抽象。与特化子类化相反。

扩展继承：只是往基类里添加新行为，不修改从基类那里继承的任何属性。

限制继承：派生类的行为比基类的少或者更严格。

本质：派生类先继承基类的所有方法，然后使一些方法无效。

例如堆栈继承自双向列表，将基类的一些方法屏蔽掉。

PS：限制继承违反了可替换性原则，创建的类已不再是派生类型，所有应该尽可能不用。

变体继承（变体子类化）：两个或多个类需要实现相似的功能，但是它们的抽象概念之间并无层次关系。（例如鼠标和触摸屏）可以选择其中一个类作为父类，子类中改写相关代码。

PS：更好的方法是将相似类中的公共代码提炼成一个抽象类，然后让其他类继承这个抽象类。但是基于已经存在的类不能使用这种方法了。

合并继承：可以通过合并两个或多个抽象特性来形成新的抽象。

2.5.11.继承和构造函数

子类在创建对象时父类的构造方法也会执行：

父类构造方法不需要参数时，父类和子类的构造方法都会自动执行。

父类需要参数时，子类必须显式提供参数，Java 中通过 super 关键字来实现。

例如：

```
class Base{
    public Base(){System.out.println("父类无参构造函数");}
    public Base(int x){System.out.println("父类有参构造函数");}
}
public class Test extends Base{
    public Test(){System.out.println("子类无参构造函数
");}
    public Test(int 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)
    }
}
```

运行结果：

```
父类无参构造函数
子类无参构造函数
父类无参构造函数
子类有参构造函数
```

子类构造函数默认调用父类的无参构造函数。

```
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)
    }
}
```

运行结果：

```
父类有参构造函数
子类无参构造函数
父类有参构造函数
子类有参构造函数
```

子类利用 super 指定调用父类的哪个构造函数

2.5.11 接口和抽象类

1) 抽象类

在定义方法时可以只给出方法头，而不必给出方法体、即方法实现的细节，这样的方法被称为抽象方法。

抽象方法必须使用关键字 abstract 修饰（Java 语言），包含抽象方法的类必须声明为抽象类，但是抽象类可以不包含抽象方法

抽象类不能被实例化

Java 语言规定，子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类

抽象类主要通过继承由子类发挥作用，其作用包括：代码重用和规划

抽象类中可以声明 static 属性和方法

定义抽象类与普通类相同，其继承方式也与普通类的继承相同，只是对于抽象方法要求非抽象子类必须实现其父类中的所有抽象方法，在此不加以例子描述

```
abstract class A{
      public static final String FLAG = "CHINA" ;
      private String name = "李兴华" ;
      public String getName() {         // 设置姓名
            return name;
      }
      public void setName(String name) {// 取得姓名
            this.name = name;
      }
      public abstract void print() ;// 定义抽象方法
}
```

2) 接口

类体里只包含静态最终变量和抽象方法，或者是只有抽象方法的时候，抽象类可以等同于一个接口。所以接口是特殊的抽象类。

用接口代替抽象类的好处（为什么单独区分出接口这一个定义）：

因为接口有更好的特性：

①可以被多继承
②设计和实现完全分离
③更自然的使用多态
④更容易搭建程序框架
⑤更容易更换实现
抽象类变成接口：
接口里的常量和方法的定义可以省略，因为所有定义在接口中的常量都默认为 public static 和 final。所有定义在接口中的方法默认为 public 和 abstract，所以可以不用修饰符限定它们。

```
//普通抽象类
public abstract class ClosedFigure {
    public abstract double area();
public abstract double perimeter();
}
//修改为接口以后
public interface ClosedFigure {
     double area();
     double perimeter();
}
```

接口的实现依旧是使用抽象类的方式，但是接口的实现具有以下几个特性：
①接口不可以被实例化
②实现类必须实现接口的所有方法
③实现类可以实现多个接口
④接口中的变量都是静态常量，方法和变量都是 public（JAVA 语法规定，接口中的变量默认自动隐含是 public static final）

2.6 多态 polymorphism

需要使用上下文来确定其确切含义

2.6.1 多态变量

定义：多态是指不同对象收到相同消息时，会产生不同行为。可以引用多种对象类型的变量。在程序执行过程中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，则是替换原则的具体表现

价值：优秀的软件可以即插即用，轻松替换，插入同等对象可以可动完成或者只需要极少量代码改动

2.6.2 多态的形式

1) 重载（专用多态）：具体在 2.6.3 讲述

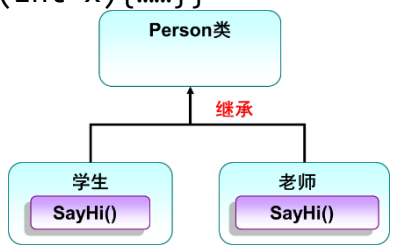
```
public void example (int x){……}
public void example (int x,double y){……}
```

2) 重置/改写（包含多态）：在层次关系中，相同的类型签名不同的执行方式，执行顺序从当前层次向高依次寻找，当找到该函数就近执行。

```
Class parent{public void example(int x){……}}
Class child extends parent{public void example(int x){……}}
```

3) 多态变量（赋值多态）：声明与包含不同 Parent

```
p=new child();
```



这种多态通常用于 list 中隶属于同一个父类的不同子类的处理

例如：对于右图所示的类，将所有“人”都存入 List<Person>中，执行方法 SayHi()

普通执行方法是：

```
for (int i = 0; i < person.Count; i++)
{
       if (person[i] is Student)
       {((Student)person[i]).StuSayHi();}
       else if (person[i] is Teacher)
       {((Teacher)person[i]).TeachSayHi();}
}
```

这种方法，当添加不同的子类的时候，就需要大量修改

但是现在我们使用多态变量，修改 Person 类

```
abstract class Person
{public abstract void SayHi();}
```

然后就能在真正的处理过程中取出子类类型判断，而在执行过程中动态的判断执行

```
abstract class Person
{public abstract void SayHi();}
```

4) 泛型（模型）：通用，T 可以用于任何的类型

```
Template <class T>
class Box{
      public:
       Box(T initial):value(initial){}
       T getValue(){return value;}
       setValue(T newValue){value=newValue;}
      private:
```

```
        T value;
};
Box<int> iBox(7);
Cout<<iBox.getValue();
iBox.setValue(12);
Cout<<iBox.getValue();
```

参数必须与接收器的类型相匹配

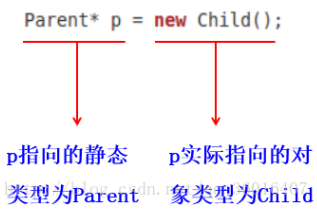
```
iBox.setValue(3.1415);//ERROR,invalid type
```

为什么需要泛型：我们希望容器能够同时持有多种类型对象，但是通常容器只会存储一种。泛型的主要目的之一就是指定容器要持有什么类型的对象，并由编译器来保证类型的正确性。那么与其使用 object，更好的选择是暂时不指定类型，在使用的时候再指定。

2.6.3 重载与类型转换

函数类型签名：关于函数参数类型、参数顺序和返回值类型的描述

1) 基于类型签名的重载：多个函数允许共享同一个名称，他们通过参数数目、顺序和类型来区分，在编译时就能基于参数值的静态类型完成解析




```
Class Parent {  };
Class Child : public Parent {  };
void Test(Parent *p) {  }
void Test(Child *c) {  }
Parent * value = new Child( );
Test (value);
```

执行哪个方法？

这里要注意，因为重载是编译时完成解析的，所以说对于上述的函数而言，value 是 Parent 类型对象。

2) 强制、转换和造型

- 强制是一种隐式类型转换，也就是所谓的小变大，无需显示引用，如：父类变子类
- 转换是程序员进行的显示类型转换，许多语言中成为造型，可以实现基本含义的改变，也可以实现类型的转换而保持含义不变，如：子类变父类、`x=((double)i)+x;`

在重载中，两个或更多方法具有相同名称和相同参数数目，编译器匹配方法：

①找精确匹配（形参实参精确匹配的同一类型）找到，则执行，找不到转第二步。

②找可行匹配（符合替换原则的匹配，即实参所属类是形参所属类的子类），没找到可行匹配，报错；只找到一个可行匹配，执行可行匹配对应的方法；如果有多于一个的可行匹配，转第三步。

③多个可行匹配两两比较，如果一个方法的各个形参，或者：与另一个方法对应位置形参所属类相同，或者：形参所属类是另一个方法对应位置形参所属类的子类，该方法淘汰另一个方法。也就是距离精准匹配越近越优先。

④如果只剩一个幸存者，执行；如果多于一个幸存者，报错。

例：对于下列继承关系有如下三个方法的参数

```
Void order (Dessert d, Cake c);
Void order (Pie p, Dessert d);
Void order (ApplePie a, Cake c);
```

```
order (aDessert, aCake);//精确匹配，执行方法一
order (anApplePie , aDessert);//可行匹配仅有方法二，执行方法二
order (aDessert , aDessert);//没有可行匹配错误
order (anApplePie , aChocolateCake);//三个方法都可行，其中方法三最近，执行方法三
order (aPie , aCake);//方法一二都可行而且距离相同，错误
```

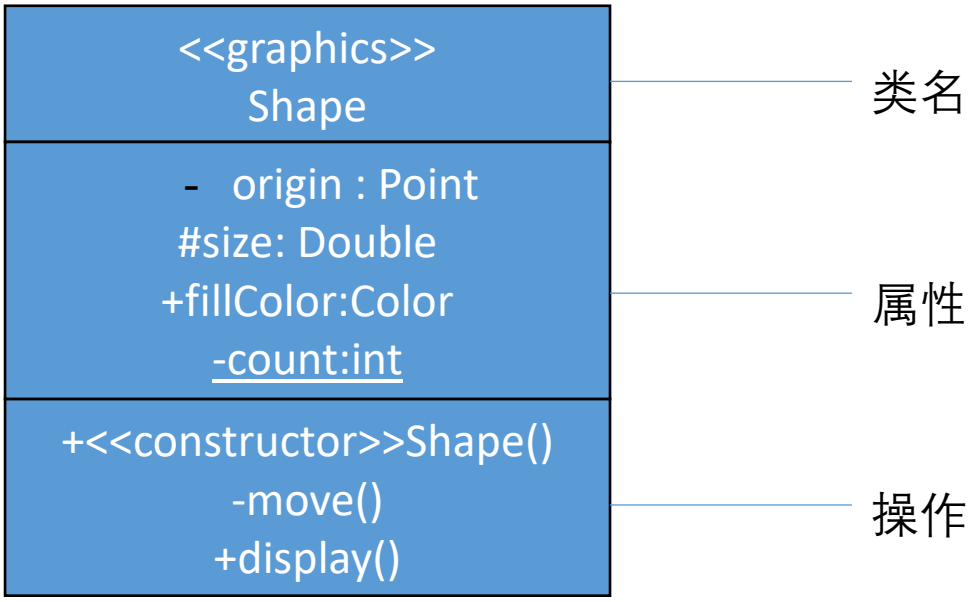
第三章 UML 类图

这一部分融合在重构与设计题的每一部分，是简约表示类图设计的重要方法，重点在类图部分。

编者解疑：鲍伟

3.1 类

1、在 UML 中使用划分为 3 行的格子的矩形来表示类和对象。
由上而下，第一块小矩形记录它的类名，所使用的版型；第二块小矩形记录类中的属性，有下划线的属性意味着属性是静态的（static）；第三块小矩形记录类中的行为方法，有的方法前可能会加版型说明，例如<<constructor>>构造器



2、类的属性

2.1 属性的语法格式：

[可见性] 属性名 [:类型] ['['多重性[次序]']'] [=初始值] [{约束条件}]

2.2 语法说明

可见性的表示符号：

+ ：public

- ： private

： protected

多重性是用来表明属性可能的取值个数，例如多重性标识 1…*表示这个属性可能有一个或多个属性值，同时这些属性值之间可能是有序的（用 ordered 指明），在多重性中*可以表示 0…n，即有 0 或者多个值。

约束条件：约束条件用来描述属性的可变性，UML 中预定义了三种属性可见性。（1）changeable；（2）addOnly ；（3）frozen

举个例子：（1）+size : Area = (100,100)

（2）#visibility：boolean = false

（3）points : Point[2…* ordered]

例子说明：（1）size 的可见性是 public，类型是 Area 类，初始值为（100,100）

（2）visibility 的可见性为 protected，类型是 boolean，初始值为 false

（3）points 的的类型是 Point，它的值可以是 2 个或者多个，而且是有序的。分析阶段它的可见性未定义，或许需要在以后的开发中进一步讨论。

3.类的操作

3.1 操作的语法格式：

[可见性] 属性名 [(参数列表)] [:返回值] [{约束条件}]

举个例子：

（1）+display() : Location

（2）-attachWindow(xwin : XwindowPtr)

例子说明：（1）display 方法，参数列表为空，返回值为一个 Location 对象，方法可见性为 public

（2）attachWindow 方法，参数列表为 xwin : XwindowPtr，xwin 为参数名，XwindowPtr 为参数类型，方法的可见性为 private。

4、类的职责

在声明类的职责时，可以非正式地在类图标符号的下方增加一栏，将该类的职责追条陈述出来。

3.2 类之间的关联

1、关联关系

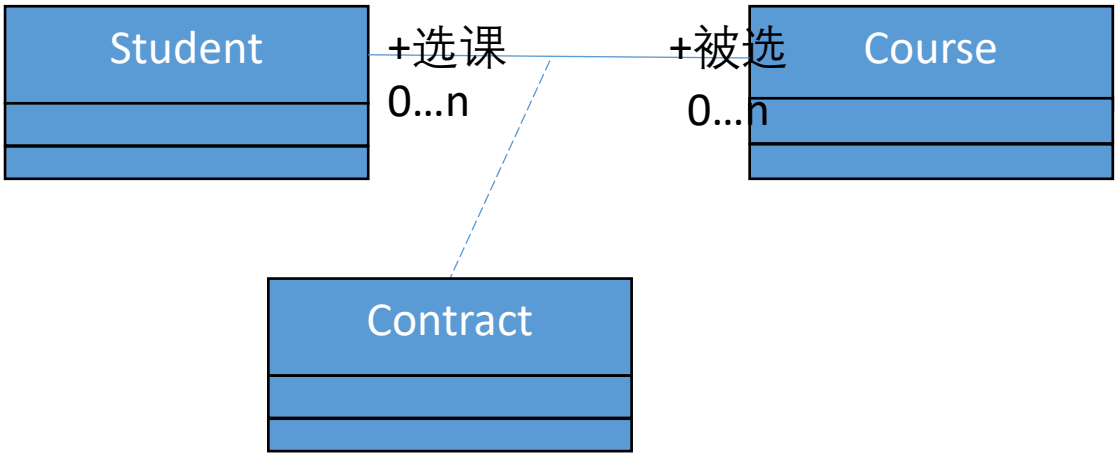
关联关系：“链”关系，关联表示的是类与类之间的语义关系，而链表示的是类的一个对象与另一个类的对象之间的语义关系。包括双向关联（连线）和单向关联（带箭头的连线）。可以理解为类的对象和另一个类的对象之间存在联系，但是并不是所有的类的实例都具备这种联系。在类图中使用连线表示。举例说明就是学生类和课程类之间存在关联关系，学生可以选课，课也可以被学生选，但是并不是所有课都会被选，也不是所有的学生都选课。

它和依赖关系是有区别的，换句话说，当考虑对象之间存在的连接时，即有些对象之间是不存在连接的，作为类之间的关联关系处理。当考虑类之间存在的连接时，所有对象之间都存在程序代码上的连接，作为类之间的依赖关系处理。

可以给关联加上关联名，来描述关联的作用。

关联的角色，关联关系两端的类的对象在对方的类里的标识符称为角色。角色有多重性。

关联类：当关联关系中存在一个属性，不适合放在任何一个类中，这个时候就需要关联类，比如在学生和课程关联关系中的成绩属性。关联类通过一条虚线和关联连接。（这和数据库设计很相似）



关联的约束：ordered，implicit，changeable，addonly，frozen，xor

关联的类型：

自返关联：同一个类的两个对象之间语义上的连接，如人之间的“结婚”关联。

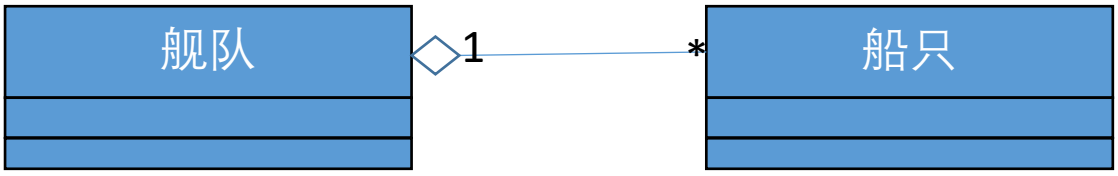
二元关联：两个类的对象之间的语义上的连接，如前面所说的学生和课程类。

N 元关联：对个类的对象之间语义上的连接 。

2、聚合关系（has-a）

聚合是一种特殊形式的关联。聚合表示类之间整体与部分的关系。聚合的图形表示方法是在挂念关系的直线末端添加一个空心小菱形，空心菱形指向具有整体性质的类。聚合分为共享聚合和组成（复用聚合），共享聚合即部分可以参与多个整体，例如律师可以为多个企业服务。组成是指整体与部分具有很强的“属于”关系，而且他们的生存期是一致的。整体不存在了，部分也会随之消失。组成的图形表示是实心菱形。

聚合关系

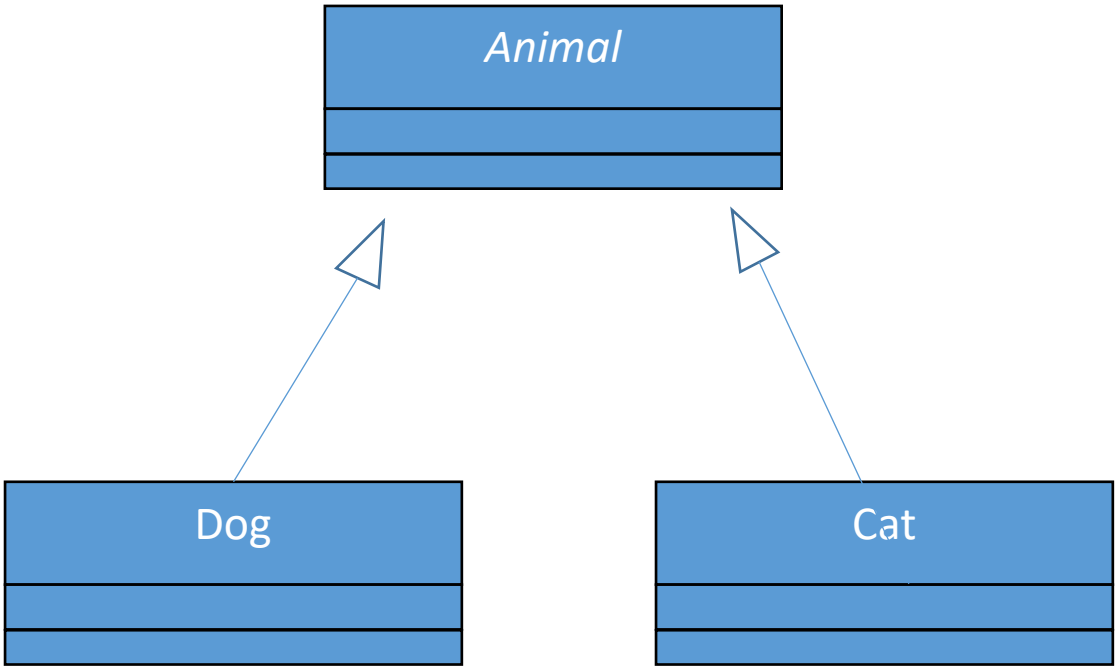


组合关系

3.泛化关系

泛化关系定义了一般元素和特殊元素之间的分类关系，在面向对象中就是类与类之间的继承关系。在 UML 中用空心三角形的连线表示泛化关系，空心三角形指向父类。

泛化关系



4、依赖关系

依赖关系定义了两个模型之间的语义连接。其中一个是独立的模型元素，另一个是依赖的模型元素。独立模型元素的变化会影响依赖模型元素的变化。在 UML 中使用虚线表示，箭头指向被依赖的类。前面对于它和关联关系的对比，应该有些概念了。

依赖关系

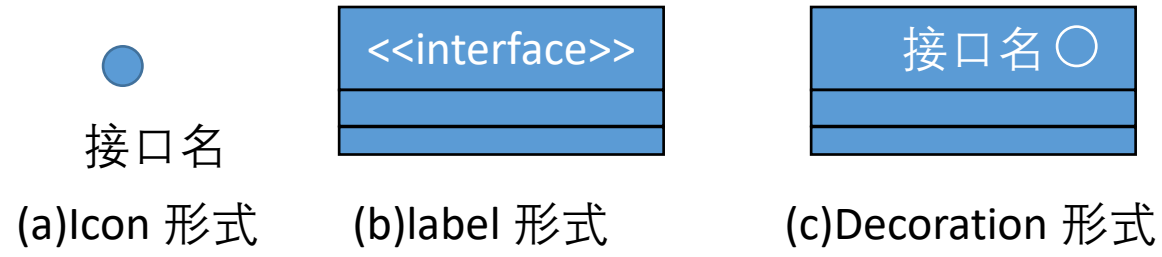
3.3 派生属性和派生关联

派生属性是指可以从其他属性推算出来的属性，如年龄可以有出生日期推算出来，派生属性的派生关联要在名字前面加上斜杠“/”

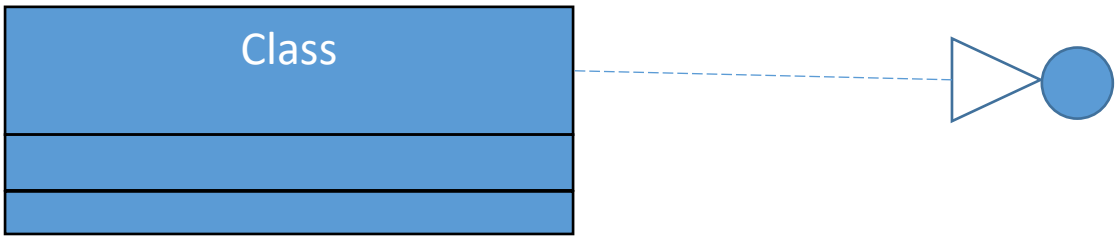
四、抽象类和接口

1、抽象类：在 UML 中用类名是斜体字类表示抽象类。如前面的 Animal 类就是使用斜体。

2、接口是类的<<interface>>版型，下图给出了接口的三种表示方法。在用接口的图标形式表示时，接口的操作不被列出。



一个类实现了接口成为实现关系，实现关系在 UML 中用带空心三角的虚线表示，空心三角指向接口。

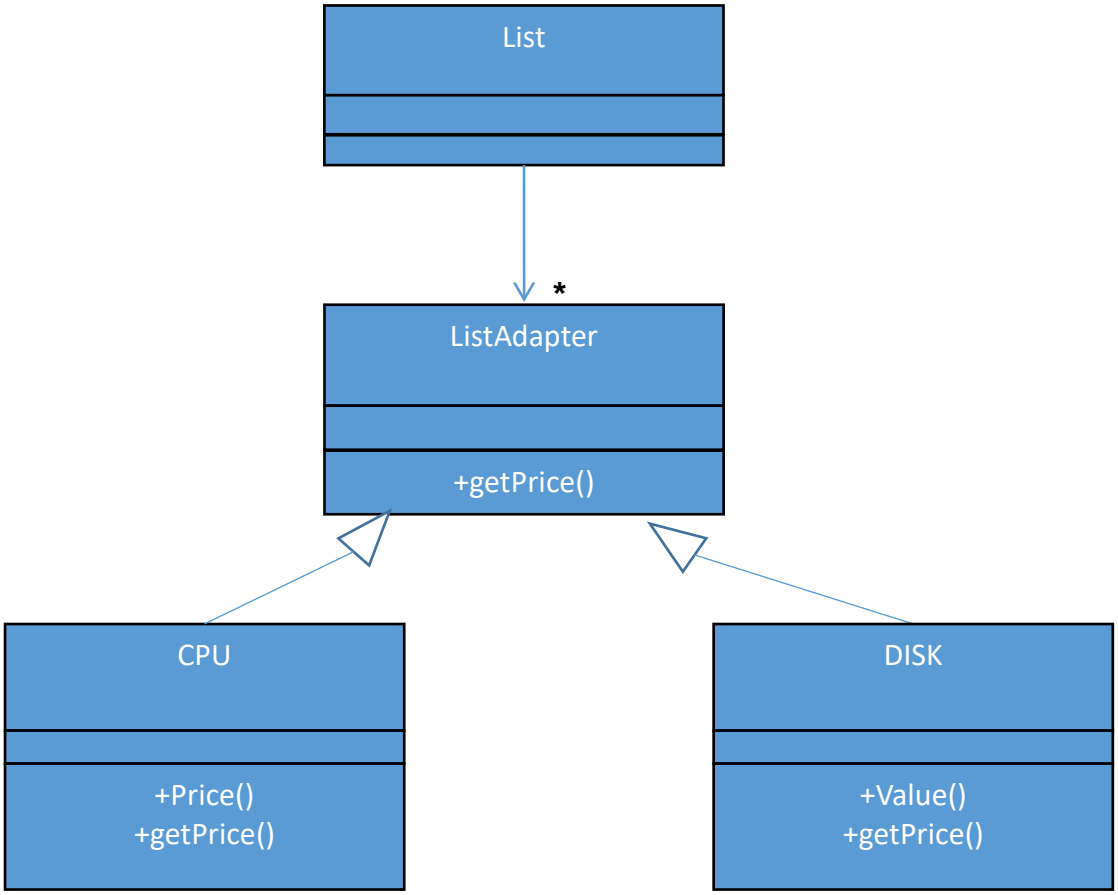


UML 类图关系一览表

往年例题解析：

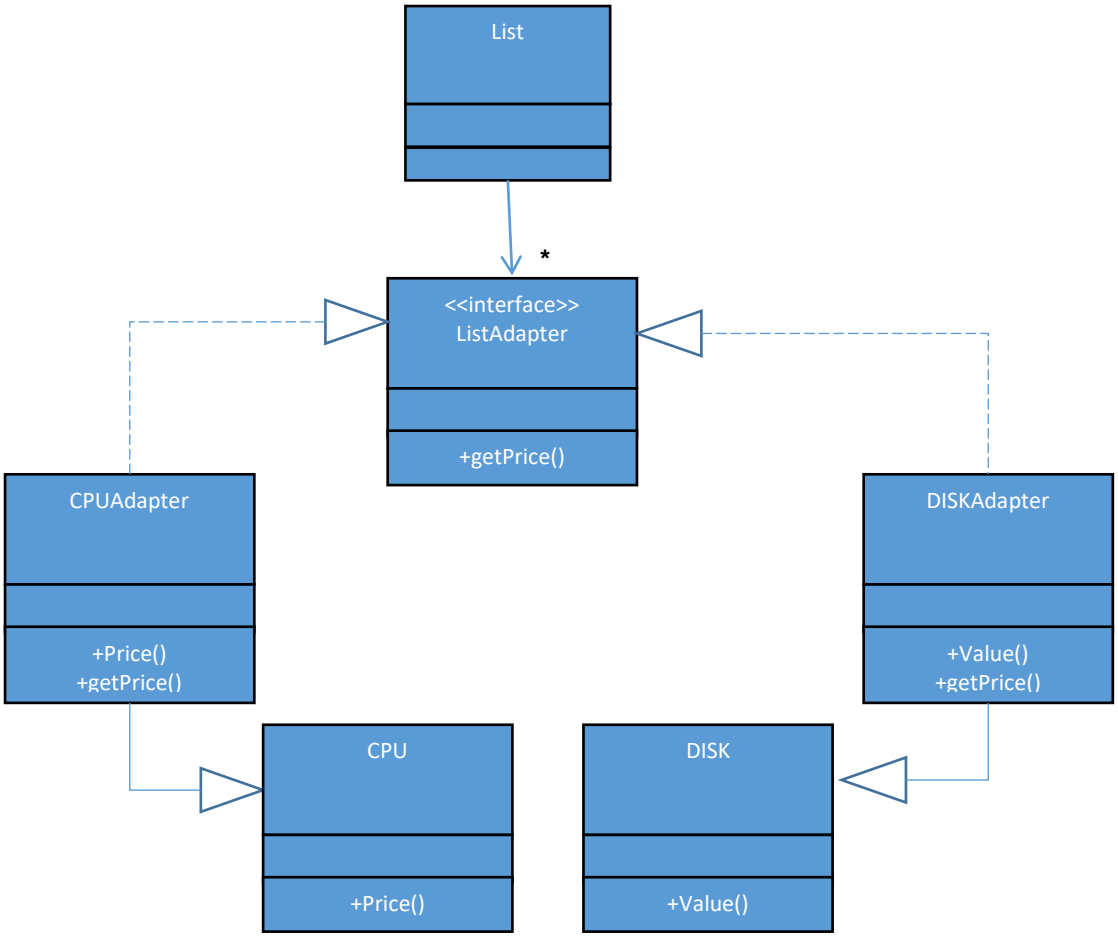
（设计类大题第五题）有两个零部件类 CPU 和 DISK，分别以二进制形式提供，即不允许更改 CPU 和 DISK 类。CPU 类中方法 Price()提供其价格，DISK 类中方法 Value()提供其价格。现在要求能将 CPU 和 DISK 的对象放入一个只能容纳同类型的列表 list 中，然后对 list 中的任何对象都能够使用 getprice()方法得到其价格。请设计实现方案，使用类图描述出来。

解：很显然两个类都提供了获得价格的方法，但是方法名不同。这个时候它们想要都放进一个 list 容器中，只能通过适配的形式来实现。这里用到适配器模式。下面是给出的类图。



定义了一个 ListAdapter 的一个适配器类，然后然 CPU 和 DISK 都继承自这个类，这样就需要覆写 ListAdapter 的 getPrice()方法，这个方法就是调用具体的子类中的 Value()或者 Price()方法。这样 List 类只要和 ListAdapter 一个类打交道就可以了。这应该是很多人的想法。

但是题目中明确说道不对 CPU 类和 DISK 类进行修改，那么这个时候我们就需要如下这样修改类图和实现。



定义一个 ListAdapter 的接口，接口中定义了 getPrice()方法，然后分别创建两个类，一个是 CPUAdapter 类，它继承 CPU 类，实现 ListAdapter 类，它其中的 getPrice()方法调用 Price()即可，另一个 DISKAdapter 是类似的实现，这样就满足了题目的要求，List 只对 ListAdapter 一个类型操作，而且 CPU 和 DISK 类都没有修改。

第四章 设计模式

这一部分是优化设计问题的模版，是重构与设计部分的参考方案，重点选取了 10 种设计模式，作为复习理解的重点。

编者答疑： 鲍伟 杜泽林 谷一滕 林子童 徐卫霞

4.1 工厂方法

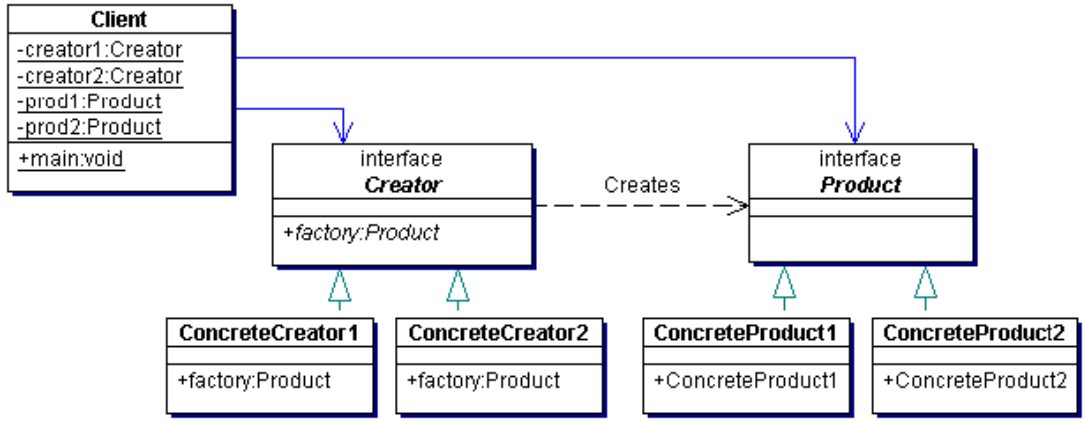
1.1 工厂方法使用场景

当你不想让客户代码决定实例化哪个类时，常常可以运用工厂方法模式。此外，工厂方法模式还可用于当客户代码不知道他需要创建哪个对象类的时候。

1.2 简单工厂和工厂方法

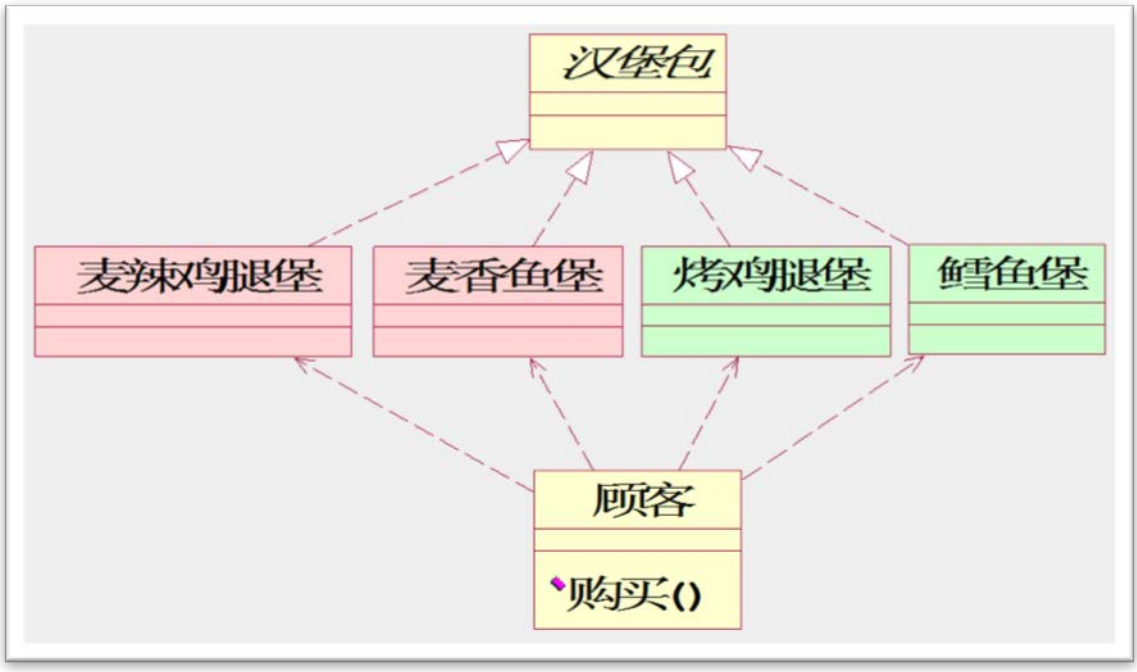
简单工厂使用一个工厂专门负责产品的生产，实现了责任的分离。但是没有完全做到“开闭原则”，新增产品时候必须修改工厂类。在面对需求小，变化不多的情况下是有效的。但是在需求变大，变化增多时简单工厂则显得很笨重，这个时候就可以使用工厂方法。工厂方法保持了简单工厂责任分离的优点，在一定程度上弥补了新增产品时修改工厂的缺陷。

1.3 类图和实现



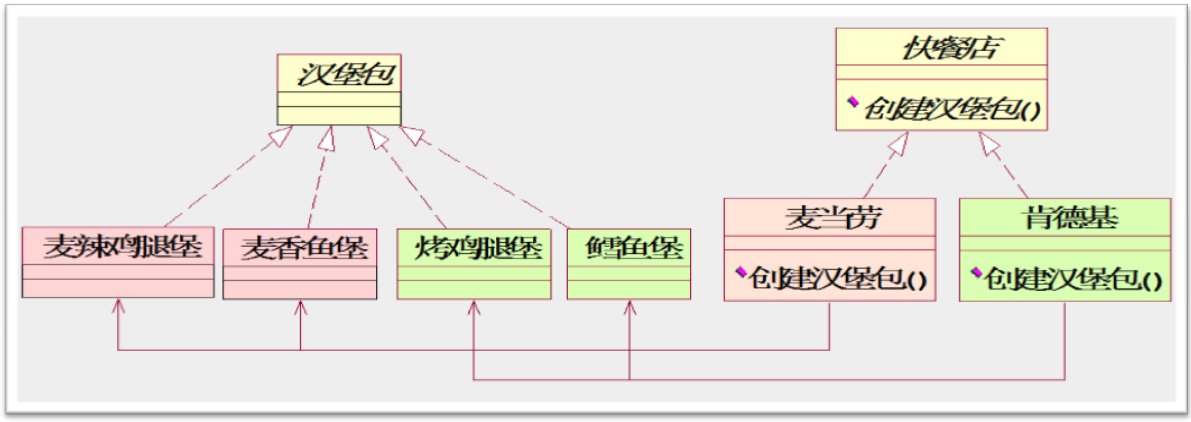
工厂方法是将工厂和产品都作为抽象的接口，不同的具体产品由不同的具体工厂生产。这个时候如果新增一个不同类别的具体产品，那么就需要对应的新增一个具体工厂来负责这个新增产品的生产。但是如果新增的是已有类别的具体产品，那么还是需要修改原有工厂的代码。

1.4 举例说明




```
void BuyFood (string 餐馆, string 食品) {
 if(餐馆 == "KFC") {
   if(食品 == "Chicken")
     hamburger = new KFCChickenHamburger;
   else if(food == "Fish")
     hamburger = new KFCFishHamburger;
   }
 else if(restaurant == "McDonald") {
   if(food == "Chicken")
     hamburger = new McDonaldChickenHamburger;
   else if(food == "Fish")
     hamburger = new McDonaldFishHamburger;
   }
}
```

以上所给的是简单工厂的实现，这个时候如果我们添加一个德克士鸡腿堡，那么我们肯定要修改这个代码，新增 ifelse。



这个时候创建两个快餐店

肯德基中 BuyFood 这样实现

```
void BuyFood (string 食品) {
    if(食品 == "Chicken")
      hamburger = new KFCChickenHamburger;
    else if(food == "Fish")
      hamburger = new KFCFishHamburger;
}
```

麦当劳中 BuyFood 这样实现

```
void BuyFood (string 食品) {
    if(food == "Chicken")
      hamburger = new McDonaldChickenHamburger;
    else if(food == "Fish")
      hamburger = new McDonaldFishHamburger;
}
```

以上是工厂方法的代码，如果这个时候新增一个德克士鸡腿堡，那么我们只需要再新增一个德克士快餐店的类负责生产这个德克士鸡腿堡即可，对上面的代码没有修改。但是如果新增一个肯德基巨无霸汉堡，那么肯德基快餐店的代码就需要修改。但是相比于简单工厂已经有很大提高了。

1.5 总结

优点

1、 在工厂方法中，用户只需要知道所要产品的具体工厂，无须关系具体的创建过程，甚至不需要具体产品类的类名。

2、 在系统增加新的产品时，我们只需要添加一个具体产品类和对应的实现工厂，无需对原工厂进行任何修改，很好地符合了“开闭原则”。

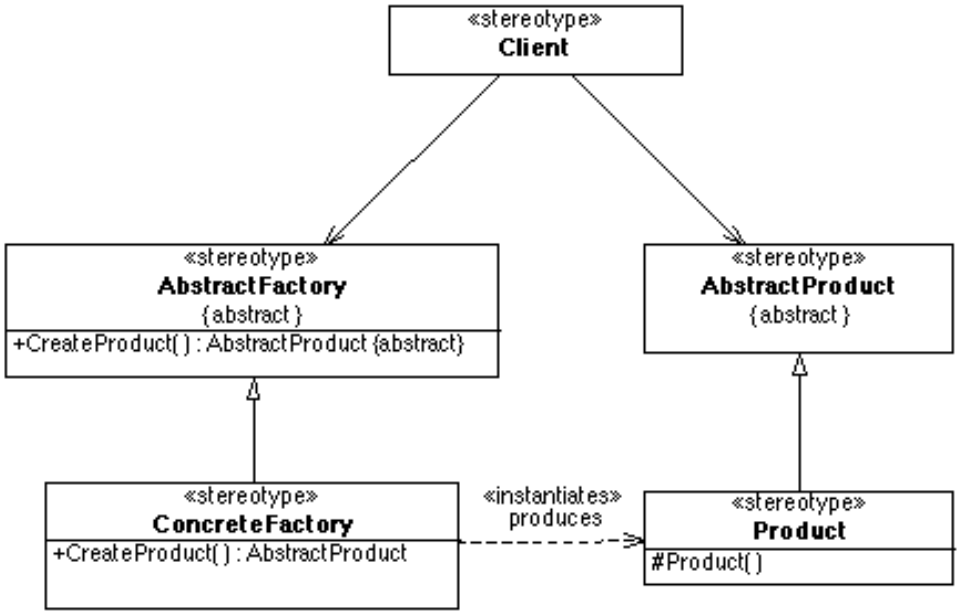
缺点

1、 每次增加一个产品时，都需要增加一个具体类和对象实现工厂，是的系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

4.2 抽象工厂（选看内容）

2.1 抽象工厂的定义和类图

定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。

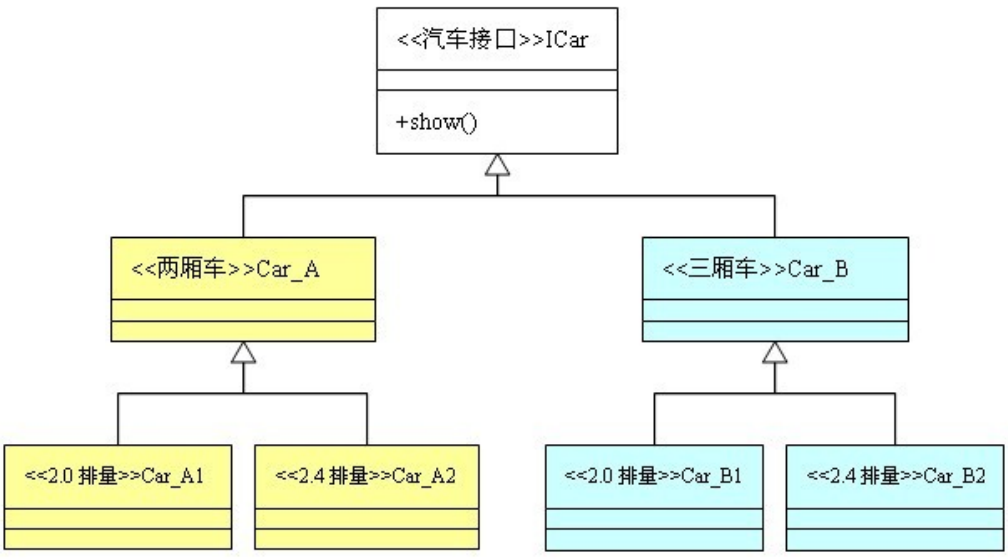


类图：

2.2 抽象工厂和工厂方法

抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象。他与工厂方法模式的区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则是针对的多个产品等级结构。在编程中，通常一个产品结构，表现为一个接口或者抽象类，也就是说，工厂方法模式提供的所有产品都是衍生自同一个接口或抽象类，而抽象工厂模式所提供的产品则是衍生自不同的接口或抽象类。

下面举例说明一下产品族



在上面的类图中，两厢车和三厢车称为两个不同的等级结构；而 2.0 排量车和 2.4 排量车则称为两个不同的产品族。再具体一点，2.0 排量两厢车和 2.4 排量两厢车属于同一个等级结构，2.0 排量三厢车和 2.4 排量三厢车属于另一个等级结构；而 2.0 排量两厢车和 2.0 排量三厢车属于同一个产品族，2.4 排量两厢车和 2.4 排量三厢车属于另一个产品族。

2.3 适用场景

当需要创建的对象是一系列相互关联或相互依赖的产品族时，便可以使用抽象工厂模式。说的更明白一点，就是一个继承体系中，如果存在着多个等级结构（即存在着多个抽象类），并且分属各个等级结构中的实现类之间存在着一定的关联或者约束，就可以使用抽象工厂模式。假如各个等级结构中的实现类之间不存在关联或约束，则使用多个独立的工厂来对产品进行创建，则更合适一点。

2.4 总结

无论是简单工厂模式，工厂方法模式，还是抽象工厂模式，他们都属于工厂模式，在形式和特点上也是极为相似的，他们的最终目的都是为了解耦。在使用时，我们不必去在意这个模式到底工厂方法模式还是抽象工厂模式，因为他们之间的演变常常是令人琢磨不透的。经常你会发现，明明使用的工厂方法模式，当新需求来临，稍加修改，加入了一个新方法后，由于类中的产品构成了不同等级结构中的产品族，它就变成抽象工厂模式了；而对于抽象工厂模式，当减少一个方法使的提供的产品不再构成产品族之后，它就演变成了工厂方法模式。

所以，在使用工厂模式时，只需要关心降低耦合度的目的是否达到了。

4.3 单例模式

3.1 单例模式的意图和特点

单例模式的意图是为了确保一个类有且仅有一个实例，并为它提供一个全局访问点。

单例模式的特点：

单例模式只可有一个实例

它必须自己实例化这一个实例

它必须向外部提供这个唯一的实例

3.2 单例模式的应用场景

先举几个大家常见的例子

比如 Windows 下的资源管理器就是使用单例模式的，你只可以打开一个资源管理器。

再比如 Windows 下的回收站也是使用单例模式的，整个系统运行中始终维持一个实例。

网站的计数器也是使用单例模式，否则很难保持同步。

一般来说单例模式的应用场景如下：

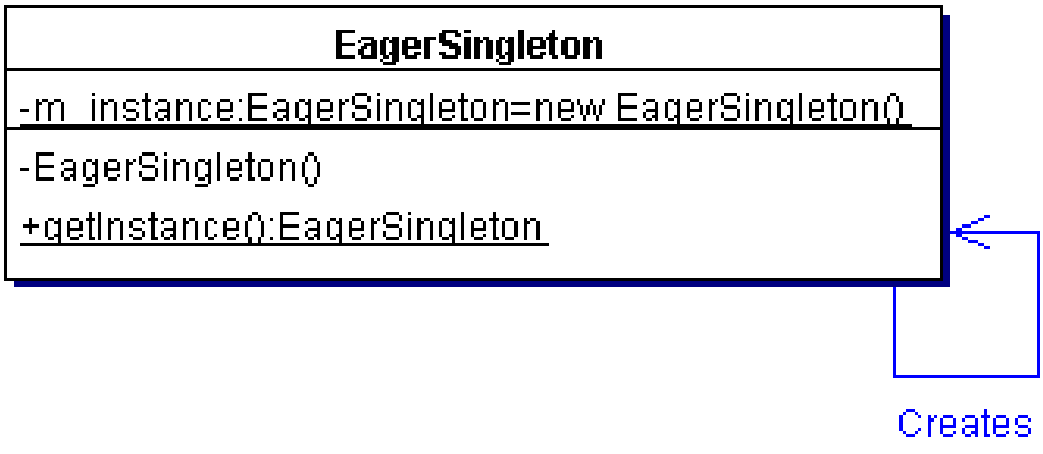
（1）资源共享的情况下，避免由于资源操作时导致的性能或损耗等。

（2）控制资源的情况下，方便资源之间的互相通信。

3.3 饿汉式和懒汉式

饿汉式和懒汉式的区别在于实例化对象的时间，饿汉式是在实例化的时候创建这个唯一的对象，也就是调用构造方法时创建；懒汉式是在需要的时候才开始创建，也就是在第一次调用 getInstance 时实例化对象。

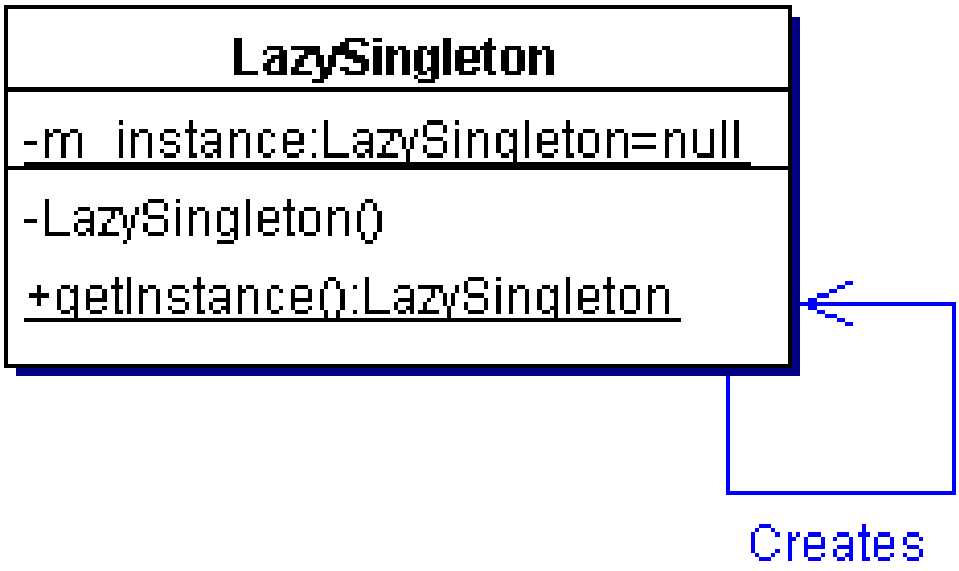
3.3.1 饿汉式



具体代码：

```
public class EagerSingleton {
        private static final EagerSingleton m_instance = new EagerSingleton();
        private EagerSingleton() { }
        public static EagerSingleton getInstance() {
                return m_instance;
        }
}
```

3.3.2 懒汉式



具体代码：

```
public class LazySingleton {
        private static LazySingleton m_instance = null;

        public static synchronized LazySingleton getInstance() {

        //这个方法比上面有所改进，不用每次都进行生成对象，只是第一次
        //使用时生成实例，提高了效率！
                if (m_instance==null)
                        m_instance = new LazySingleton();
                return m_instance;
        }
}
```

3.4 单例模式的总结

从上面提供的两种形式的单例模式可以看出，单例模式通过隐藏构造函数（构造函数定义为 private），提供对象创建的唯一个入口点（而这个入口点里的方法是类自己维护的），从而将类的职责集中在类的单个实例中。

4.4 Adapter(适配器)模式

模式概括：

将一个类的接口转换成客户希望的另一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

（一）Adapter 模式的应用场景

为了理解适配器在软件是设计里的应用，我们先看一下现实世界的具体情况，然后再去理解软件设计中的比较抽象的情况。

①现实生活中的情况：

世界上不同的地区插座的规格有一些差别，假设 A 地区的插座和 B 地区的插座规格不同，要让一台 A 地区的电器在 B 地区也能正常使用。显然不能采取将 B 地的所有插座替换为 A 地规格的方案，这样做的代价太大，对原有材料的改动成本太高。但是我们可以为电器配备插座转换器，这样就解决了电器在两个地区间的使用问题。

②软件设计中的情况：

在软件设计中，很可能会出现这种情况——类 A 实现了接口 A1,类 B 实现了接口 B1,这里 C 调用 A 和 B 希望 A 和 B 能提供相同方法的接口。这种情况就类似于电器插座问题，我们也想要在软件设计中提供一个“转换器”。

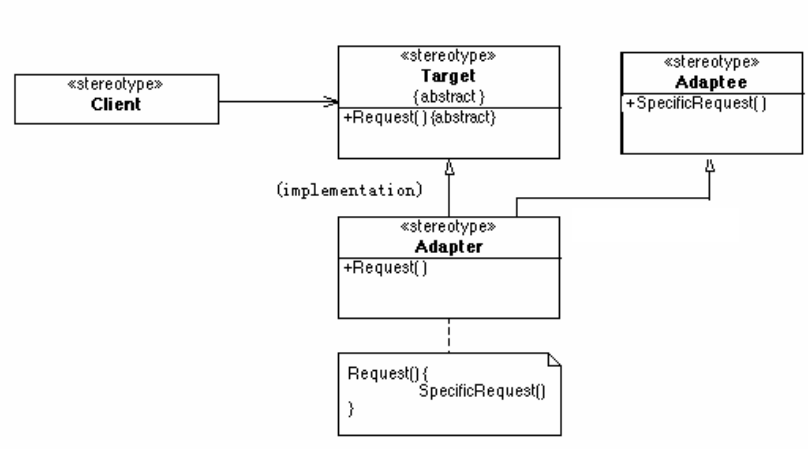
（二）结合实例说明 Adapter 模式的两种类型

（1）类适配

```
public interface Car {
    public double getSpeed();
    public void run();
}
public class Truck {
    private double speed = 100.00;
    public double getTruckSpeed() {
        return speed;
    }
    public void move() {
        System.out.println("Move move !!");
    }
}
public class Lorry extends Truck implements Car {
     //Lorry 继承自 Truck 并实现 Car 接口。（关于继承和接口实现稍后会有补充）
    @Override
    public double getSpeed() {
        return getTruckSpeed();
    }
    @Override
    public void run() {
        move();
    }
}
```

①实例解读：

首先定义 Car 接口，我们希望程序中所有的交通工具都实现 Car 接口。然而，Truck（卡车）类虽然实现了 Car 接口相似的功能，但是它与 Car 接口并不匹配。我们为了减少对原有代码的改动，添加适配器 Lorry，Lorry 继承自 Truck 并实现 Car 接口。
②UML 类图：

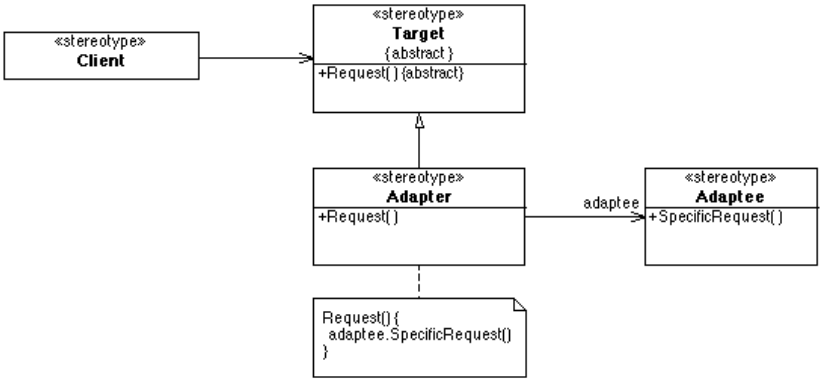


（2）对象适配

```
public abstract Car {
    public double getSpeed();
    public void run();
}
public class Truck {
    private double speed = 100.00;
    public double getTruckSpeed() {
        return speed;
    }
    public void move() {
        System.out.println("Move move !!");
    }
}
public class Lorry extends Car {
    private Truck truck;
    public Lorry() {
        truck = new Truck();
    }
    @Override
    public double getSpeed() {
        return Truck.getTruckSpeed();
    }
    @Override
    public void run() {
        super.run();
        truck.move();
    }
}
```

①实例解读：

首先定义 Car 抽象类，我们希望程序中所有的交通工具继承自 Car 类。然而，Truck（卡车）类虽然实现了 Car 类相似的功能，但是它与 Car 类并不匹配。我们为了减少对原有代码的改动，添加适配器 Lorry，Lorry 继承自 Car，并且 Lorry 适配器类中定义一个 Truck 类对象作为成员变量，通过 Truck 类对象中的方法来实现 Car 类方法的需求。
②UML 类图



③补充一下 extends 与 implements
1、在类的声明中，通过关键字 extends 来创建一个类的子类。
一个类通过关键字 implements 声明自己使用一个或者多个接口。
extends 是继承某个类，继承之后可以使用父类的方法，也可以重写父类的方法；
implements 是实现多个接口，接口的方法一般为空的，必须重写才能使用
2、extends 是继承父类，只要那个类不是声明为 final 或者那个类定义为 abstract 的就能继承
JAVA 中不支持多重继承，但是可以用接口来实现，这样就要用到 implements，继承只能继承一个类，
但 implements 可以实现多个接口，用逗号分开就行了，比如 ：
class A extends B implements C,D,E
3、接口实现的注意点：
a.实现一个接口就是要实现该接口的所有的方法(抽象类除外)。
b.接口中的方法都是抽象的。
c.多个无关的类可以实现同一个接口，一个类可以实现多个无关的接口。
④两种方式的比较
类适配:适配器类需要实现客户端接口，继承现有实体类；
对象适配:对象适配器采用了组合,并非是继承;创建一个对象适配器,继承客户端类,在类中维护一个现有类实例对象,满足客户端类需求方法；

4.5 Decorator(装饰者)模式

模式概括：

把基本功能和扩展功能区分开，并把扩展功能组件化，在使用者中按需组装成需要的扩展子类。

（一）Decorator 模式应用场景

在程序设计中，良好的设计应当遵守开闭原则。开闭原则要求我们：系统的扩展不影响已有的代码，开放是为了扩展，关闭是保持原有代码不变。所以当要对类实现新的功能时，我们不仅要实现拓展，还要维持原有部分不变。比如咖啡类，可能会改变很多咖啡配料的价格，如果所有种类的咖啡都继承自咖啡类，那么几乎要修改所有的子类，这样做显然不是良好的编程方式。所以我们将类可能拓展的部分从类中分离出来，独立成为拓展子类。

（二）结合实例说明 Decorator 模式

①实例

```
public interface Component {
    public void work();
}
public abstract class Decorator implements Component{
    protected Component component;
    protected Decorator(Component component){
        this.component = component;
    }
}
public class ConcreteComponent implements Component {
    @Override
    public void work() {
        System.out.println("主功能实现");
    }
}
public class ConcreteDecoratorA extends Decorator {
     public ConcreteDecoratorA(Component component){
        super(component);
    }
    @Override
    public void work() {
        System.out.println("装饰功能 A");
        component.work();
    }
}
public class ConcreteDecoratorB extends Decorator {
    public ConcreteDecoratorB(Component component){
        super(component);
    }
    @Override
```

```
    public void work() {
        System.out.println("装饰功能 B");
        component.work();
    }
}
public class test {
    public static void main(String[] args) {
        Component component = new ConcreteComponent();
        component = new ConcreteDecoratorA(component);
        component = new ConcreteDecoratorB(component);
        Component.work();
    }
}
```

输出结果是：装饰功能 B
装饰功能 A
主功能实现

实例分析：

1、基本类为 Component 和 Decorator，Decorator 继承自 Component，并且在 Decorator 类中包含一个 Component 对象。

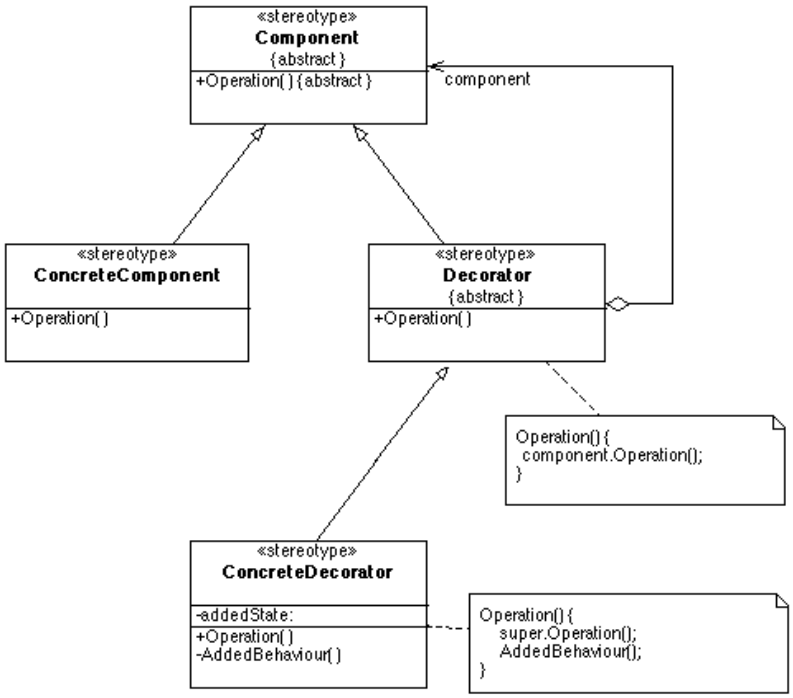
2、具体实现由 ConcreteComponent 和 ConcreteDecoratorA/B 完成。ConcreteComponent 是 Component 的子类，ConcreteDecoratorA/B 是 Decorator 的子类。

ConcreteComponent 是被装饰的对象。ConcreteDecoratorA/B 是装饰的提供者，它重写了 Decorator 的 work 方法——将内部维持的 Component 对象的 work 方法添加进去，并且添加了新的功能——以此实现功能的拓展。

3、因为 ConcreteDecoratorA/B 也是 Component 的子类，所以 ConcreteDecoratorA/B 可以多层修饰，比如

```
Decorator decoratedComponent = new ConcreteDecoratorB(new
ConcreteDecoratorA(component));
```

②UML 类图



③优缺点：

优点：

1、灵活性——核心功能与扩展功能分离，不仅扩展功能可选，使得核心功能也可以有多个实现版本，使用者按需选择。

2、高效性——如果扩展功能有 m 个，为了实现全部多态所需要的传统子类个数是一个排序数之合 A(m,m)+A(m-1,m)+...+A(1,m)，而使用装饰者模式，我们只需要 m 个扩展装饰类就行了。

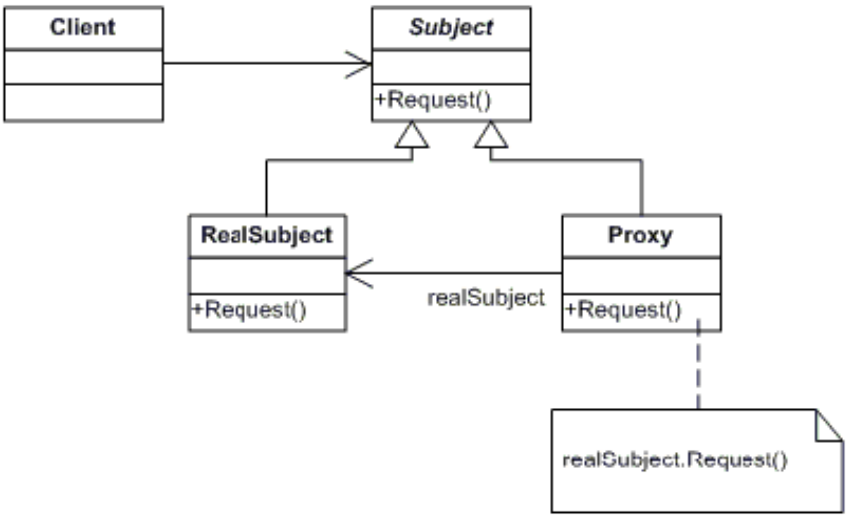
缺点：

1、组件必须分离——由于扩展功能必须能够被随意顺序添加，我们不仅要保证核心功能与扩展功能不相互依赖，更要保证扩展功能之间不相互依赖。

2、核心功能复杂——不难想象，核心功能类越复杂，在其上添加完全分离的扩展功能就越难。如果核心功能复杂到一定程度，装饰类将很难实现。

4.6 代理模式

描述：为其他对象提供一种代理以控制对这个对象的访问 。



模式构成：

抽象主题角色（Subject）：声明了真实主题和代理主题的共同接口，这样一来在任何使用真实主题的地方都可以使用代理主题

代理主题（Proxy）角色：代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；代理主题角色提供一个与真实主题角色相同的接口，以便可以在任何时候都可以替代真实主题；控制真实主题的应用，负责在需要的时候创建真实主题对象（和删除真实主题对象）

真实主题（RealSubject）角色：定义了代理角色所代表的真实对象

代码实现：

抽象主题角色：

```
public interface Subject{
    //各种具体操作方法
   void f();
   void g();
   void h();
}
```

真实主题角色：

```
public class RealSubject implements Subject {
    //真实主题角色的各种操作方法
    public RealSubject(){}//构造方法
   public void f() {
         System.out.println("RealSubject.f()");
   }
   public void g() {
         System.out.println("RealSubject.g()");
   }
   public void h() {
```

```
        System.out.println("RealSubject.h()");
    }
}
```

代理主题角色：

```
public class Proxy implements Subject {
    private Subject realSubject;
    public Proxy() { //构造方法，传入一个真实主题对象
        realSubject = new RealSubject ();
    }
    //代理调用真实主题角色的各种方法
    public void f() { realSubject.f(); }
    public void g() { realSubject.g(); }
    public void h() { realSubject.h(); }
}
```

场景类：

```
public class client {
    public static void main(String[] args){
        //定义一个真实主题角色
        Subject realSubject = new RealSubject();
        //定义一个代理主题角色
        Subject proxy = new Proxy();
        //通过代理主题角色来调用真实主题角色的各种方法
        proxy.f();
        proxy.g();
        proxy.h();
    }


}
```

具体描述：

代理（Proxy）模式给某一个对象提供一个替身或占位符，以控制对这个对象的访问。

所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

具体应用：

远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。可以隐藏一个对象存在于不同地址空间的事实。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使（Ambassador）。（可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的对象是局域的而不是远程的，而代理对象承担了大部分的网络通信工作。）

虚拟（Virtual）代理：代理对象可以在必要的时候才将被代理的对象加载。代理可以对加载的过程加以必要的优化。当一个模块的加载十分耗费资源的时候，虚拟代理的优点就非常明显。

保护（Protect or Access）代理（安全代理）：可以在运行时对用户的有关权限进行检查，然后在核实后决定将调用传递给被代理的对象。

智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。

模式要点：

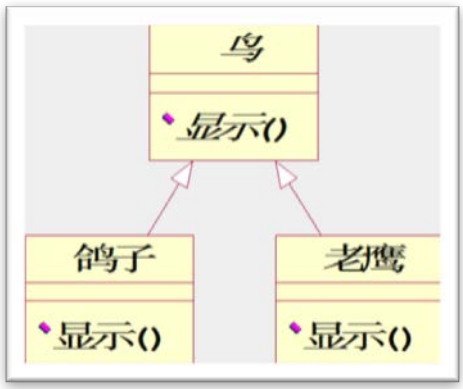
“增加一层间接层”是软件系统中对许多复杂问题的一种常见解决方法。在面向对象系统中，直接使用某些对象会带来很多问题，作为间接层的 proxy 对象便是解决这一问题的常用手段。

具体 proxy 设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象作细粒度的控制，有些可能对组件模块提供抽象代理层，在架构层次对对象作 proxy。

4.7 桥梁模式

1.背景

案例：在一个有各种不同鸟类的设计中，初始状态如下：



在这个基础上，需求不断变化，从初始的只需要显示，到后来的需要飞，以及增加了新的鸟类。对于初始的设置，如果将飞放在父类中实现，那么不会飞的鸟类将要重定义这个方法；如果只在父类中写一个抽象方法，那么会飞的鸟类又会出现代码的重复；

这个时候，我们想到继承，也就是将鸟类分为会飞的和不会飞的分别继承。但是可以想象，当再次增加游泳的方法以后，子类的继承关系有需要分别改变

设计原则中，有一个组合优先原则（组合复用原则），这是对于灵活系统而言的。继承复用的优点是可以很容易的修改或者扩展父类的实现，但相应的也会破坏封装，将父类细节全都暴露给子类，并且父类的变化势必会牵连子类。而且对于这个例子而言，继承是静态的，不能再运行时发生改变，缺少灵活性

组合恰恰相反，但是它也具有自己的缺点：对象数量增加、增加系统的复杂性

综合以上所讲，当系统需要应对变化的时候，应当首先使用组合，因为组合更加的灵活

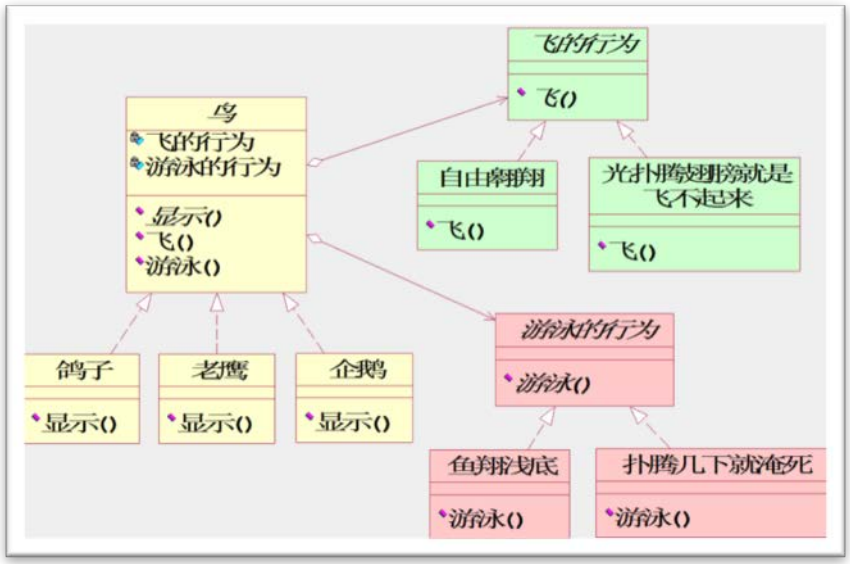
2.使用范围

当系统中有两种相关联的点易发生变化的时候，考虑桥梁模式。换种话说，桥梁模式就是将两端都变化的东西通过一座桥连接起来，两端发生变化的时候，桥梁不会改变，也就是，整体的架构不会改变。**“使变化点和不变点独立开来”**。

根据设计原则封装可变性，要发现代码中容易变化的部分封装，使它和不容易变化的部分独立开来。

接上面的例子，可以看到，系统中经常变化的两类分别是鸟的种类和鸟的行为，而且，鸟类*拥有*行为，鸟类行为的具体实现，委托行为类完成

据此，设计以下系统：



对于这个系统，无论增加鸟的类型还是鸟的行为都不需要改变原有的架构以及其他的所有类。接下来，将这个系统抽象出来。

3.系统构成

1)角色

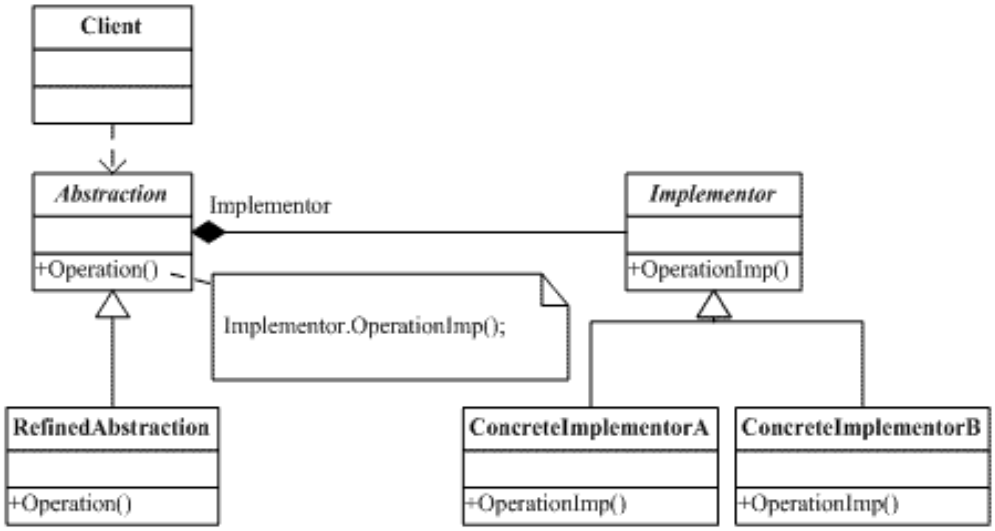
抽象化(Abstraction)角色：抽象化给出的定义，并保存一个对实现化对象的引用。

修正抽象化(Refined Abstraction)角色：扩展抽象化角色，改变和修正父类对抽象化的定义。

实现化(Implementor)角色：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。

具体实现化(Concrete Implementor)角色：这个角色给出实现化角色接口的具体实现。

2)UML 图



3)相关设计原则

组合优先原则（组合复用原则），封装可变性（上面已经讲到）

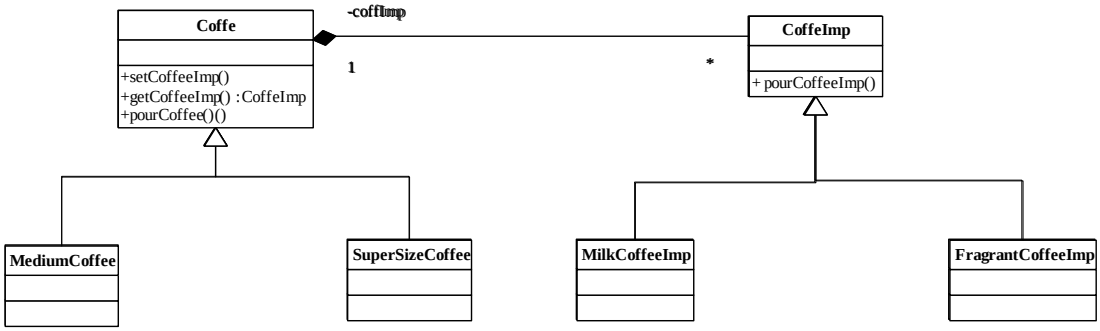
开闭原则：系统功能在不修改已有代码的情况下得到了扩展

4)意图

将抽象部分与实现部分分离，使它们都可以独立的变化，这里的抽象部分就是上面说的类型，实现部分则是指的行为，这种模式适用于抽象与实现部分分离的系统，大体而言就会有两种东西可能会增删改的系统

4.实现实例：

1)到咖啡厅喝咖啡，咖啡有中杯和大杯之分，同时还有加奶和不加奶之分



实现代码（主要学习应用上如何使用桥梁模式，因为类图中已经详细的说明了每个类要干什么，下面的代码仅供示例）：

抽象部分抽象类：

```
public abstract class Coffee
{
  CoffeeImp coffeeImp;
  public void setCoffeeImp()
  {
    this.CoffeeImp = CoffeeImpSingleton.getTheCoffeImp();
  }
  public CoffeeImp getCoffeeImp() {return this.CoffeeImp;}
  public abstract void pourCoffee();
}
```

行为实现部分的抽象类

```
public abstract class CoffeeImp
{
  public abstract void pourCoffeeImp();
}
```

中杯咖啡

```
public class MediumCoffee extends Coffee
{
  public MediumCoffee() {setCoffeeImp();}
  public void pourCoffee()
  {
    CoffeeImp coffeeImp = this.getCoffeeImp();
    //我们以重复次数来说明是冲中杯还是大杯 ,重复 2 次是中杯
    for (int i = 0; i < 2; i++)
    {
      coffeeImp.pourCoffeeImp();
    }

  }
}
```

大杯咖啡

```
public class SuperSizeCoffee extends Coffee
{
  public SuperSizeCoffee() {setCoffeeImp();}
  public void pourCoffee()
  {
    CoffeeImp coffeeImp = this.getCoffeeImp();
    //我们以重复次数来说明是冲中杯还是大杯 ,重复 5 次是大杯
    for (int i = 0; i < 5; i++)
    {
      coffeeImp.pourCoffeeImp();
    }
  }
}
```

行为的具体实现-加奶

```
public class MilkCoffeeImp extends CoffeeImp
{
  MilkCoffeeImp() {}
  public void pourCoffeeImp()
  {
    System.out.println("加了美味的牛奶");
  }
}
```

行为的实现-不加奶

```
public class FragrantCoffeeImp extends CoffeeImp
{
  FragrantCoffeeImp() {}
  public void pourCoffeeImp()
  {
```

```
    System.out.println("什么也没加,清香");
  }
}
```

应用：在这个系统中，我们设置了咖啡的大小杯为抽象的咖啡的种类，而加不加奶为实现类也就是行为，在实际的应用中，我们会新建一个咖啡类（确定是大小杯），然后向这个类传递加不加奶的行为信息（以参数的形式）。当有更多的行为的时候，，就会向咖啡传递更多的参数，也就是说下面这个类是会变化的“桥梁”

```
public class CoffeeImpSingleton
{
  private static CoffeeImp coffeeImp;
  public CoffeeImpSingleton(CoffeeImp coffeeImpIn)
  {this.coffeeImp = coffeeImpIn;}
  public static CoffeeImp getTheCoffeeImp()
  {
    return coffeeImp;
  }
}
```

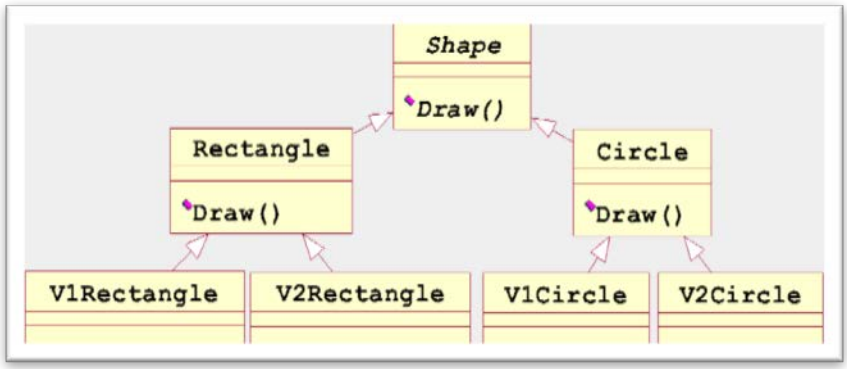
使用 main 方法：

```
//拿出牛奶
CoffeeImpSingleton coffeeImpSingleton = new CoffeeImpSingleton(new
MilkCoffeeImp());
//中杯加奶
MediumCoffee mediumCoffee = new MediumCoffee();
mediumCoffee.pourCoffee();
//大杯加奶
SuperSizeCoffee superSizeCoffee = new SuperSizeCoffee();
superSizeCoffee.pourCoffee();
```

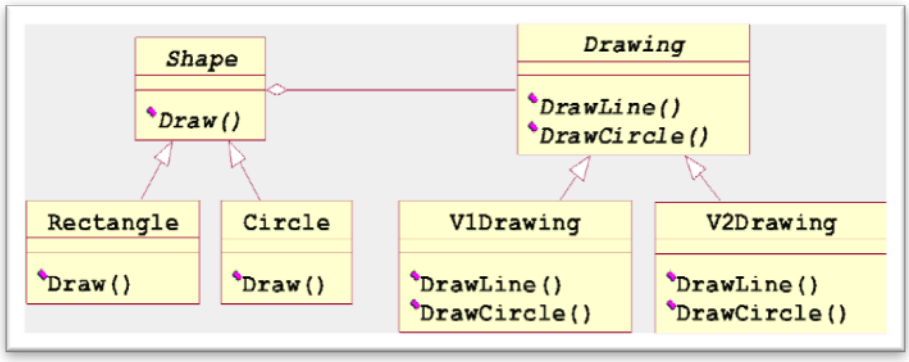
2)其他实例

一个绘图软件，可以画多种图形，并且每种图形都支持不同的绘图算法

传统设计：



桥梁模式：


Shape
Draw()
Drawing
DrawLine()
DrawCircle()
Rectangle
Draw()
Circle
Draw()
V1Drawing
DrawLine()
DrawCircle()
V2Drawing
DrawLine()
DrawCircle()

4.8 观察者模式

问题：

间谍监视敌人行动，如何在敌人行动时获得第一手消息？这里举一个战国时期秦国李斯通　　过间谍得知韩国韩非子一举一动的例子。

普通设计：

这个例子里，韩非子就是被观察者，李斯是观察者。

具体的被观察者

```
public class HanFeiZi implements IHanFeiZi{
    private ILiSi liSi =new LiSi();//把李斯声明出来
    public void haveBreakfast(){//韩非子要吃饭了
        System.out.println("韩非子:开始吃饭了...");
        this.liSi.update("韩非子在吃饭");//通知李斯
    }
}
```

具体的观察者

```
public class LiSi{
    public void update(String str){//韩非子有活动，他就知道，向老板汇报
    System.out.println("李斯:观察到韩非子活动，开始向老板汇报了...");
    this.reportToQinShiHuang(str);
    System.out.println("李斯：汇报完毕...\n");
}
//汇报给秦始皇
private void reportToQinShiHuang(String reportContext){
    System.out.println("李斯：报告，秦老板！韩非子有活动了--->"+reportContext);
    }
}
```

场景类：

```
public class Client {
    public static void main(String[] args) {
        HanFeiZi hanFeiZi = new HanFeiZi();//定义出韩非子
        hanFeiZi.haveBreakfast();//然后我们看看韩非子在干什么
    }
}
```

具体问题：

当有更多的观察者想要加入的时候，需要对被观察者代码进行更改，当被观察者有新的行动时，需要修改的代码就更多了。而且需要把所有的观察者都加入。严重违背开闭原则，此时需要引入观察者模式，对源代码进行更改。

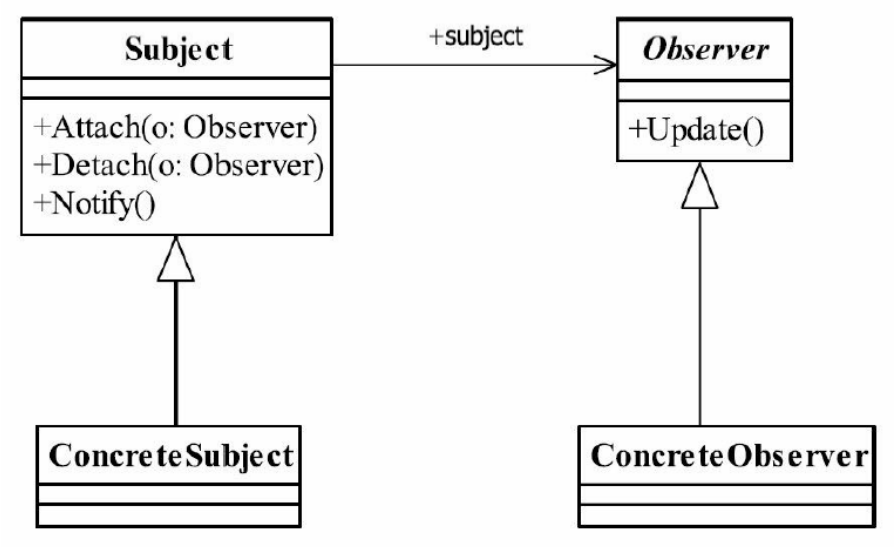
观察者模式：OBSERVER 模式 ， 发布订阅者模式

本质：

定义对象间的一种一对多的依赖关系

当一个对象的状态发生改变时，所有依赖于他的对象都得到通知并被自动更新

UML 图：



模式构成：

- Subject 被观察者

定义被观察者必须实现的职责，它必须能够动态地增加、取消观察者。它 一般是抽象 类或者是实现类，仅仅完成作为被观察者必须实现的职责：管理 观察者并通知观察者。

- Observer 观察者

观察者接收到消息后，即进行 update（更新方法）操作，对接收到的信息 进行处理。

- ConcreteSubject 具体的被观察者

定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知。

- ConcreteObserver 具体的观察者

每个观察在接收到消息后的处理反应是不同，各个观察者有自己的处理逻辑。

被观察者：

```
public abstract class Subject {
   //定义一个观察者数组
   private Vector<Observer>obsVector=new Vector<Observer>();
   //增加一个观察者
   public void addObserver(Observer o){
      this.obsVector.add(o);
   }
   //删除一个观察者
   public void delObserver(Observer o){
      this.obsVector.remove(o);
   }
```

```
    //通知所有观察者
    public void notifyObservers(){
        for(Observer o:this.obsVector){
            o.update();
        }
    }
}
```

观察者：

```
public interface Observer {
    //更新方法
    public void update();
}
```

场景类：

```
public class Client {
    public static void main(String[] args) {
        //创建一个被观察者
        ConcreteSubject subject = new ConcreteSubject();
        //定义一个观察者
        Observer obs= new ConcreteObserver();
        //观察者观察被观察者
        subject.addObserver(obs);
        //观察者开始活动了
        subject.doSomething();
    }
}
```

观察者模式的优点：

观察者和被观察者之间是抽象耦合

如此设计，则不管是增加观察者还是被观察者都非常容易扩展，而且在 Java 中都已经实现的抽象层级的定义，在系统扩展方面更是得心应手。

建立一套触发机制

根据单一职责原则，每个类的职责是单一的，那么怎么把各个单一的职责串联成真实世界的复杂的逻辑关系呢？比如，我们去打猎，打死了一只母鹿，母鹿有三个幼崽，因失去了母鹿而饿死，尸体又被两只秃鹰争抢，因分配不均，秃鹰开始斗殴，然后羸弱的秃鹰死掉，生存下来的秃鹰，则因此扩大了地盘……这就是一个触发机制，形成了一个触发链。观察者模式可以完美地实现这里的链条形式。

观察者模式的缺点：

观察者模式需要考虑一下开发效率和运行效率问题，一个被观察者，多个观察者，开发和调试就会比较复杂，而且在 Java 中消息的通知默认是顺序执行，一个观察者卡壳，会影响整体的执行效率。在这种情况下，一般考虑采用异步的方式。

多级触发时的效率更是让人担忧，大家在设计时注意考虑。

观察者模式的使用场景：

关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。

事件多级触发场景。

跨系统的消息交换场景，如消息队列的处理机制。

观察者注意问题：

减少广播链的长度，便于调试与维护。

设计逻辑，根据具体情况被观察者考虑是否通知观察者。防止没有意义的通知。

4.9 策略模式

问题：某网上商城举行促销活动，图书每本折扣 1 元，服装类八折，家具类九折，护肤品无折扣。顾客结算时计算购物车中商品的总金额。

普通设计：

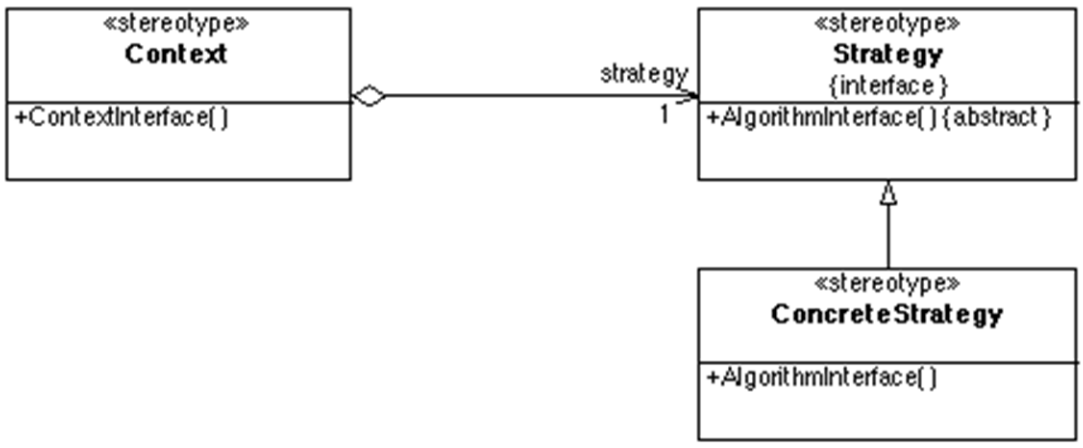
```
public void main(){
int price=0；
while(购物车里还有商品)
{取出商品；
if(商品==book)
  {price+=该商品价格-1；}
else
  if(商品==服装类)
  {price+=商品价格*0.8；}
  else
    If(商品==家具类)
    {Price+=商品价格*0.9；}
    else
    price+=商品价格；
  }
  }
```

具体问题：存在大量 if else 语句，编写代码复杂，易出错；代码的可读性差；若再增加一个新的促销项目，整体代码需要进行修改，代码的维护和修改变得复杂。鉴于以上方法的缺点，利用策略模式更加合适。

策略模式：Strategy 模式，又称 Policy 模式

本质：定义一系列的算法，并将它们分别封装起来，并且使他们可以相互替换（针对上述问题，定义一系列的促销算法）。该模式可以独立于使用它的客户而变化。

UML 图：



模式的构成：

环境（context）：持有一个 strategy 类的引用，可定义一个接口让 strategy 类访问它的数据。

抽象策略（strategy）：给出所有具体策略类所需接口，通常由一个接口或抽象类实现。

具体策略（ConcreteStrategy）：包装了相关算法或行为，实现 strategy 接口的某个具体类。

示意代码：

Context：

```
public class Context
{
   private Strategy strategy;
   public void contextInterface()
   {
            strategy.strategyInterface();
   }
Public void Setstrategy(strategy s){
Strategy=s;
}
}
```

Strategy：

```
abstract public class Strategy
{
   public abstract void strategyInterface();
}
```

Concretestrategy：

```
public class ConcreteStrategy extends Strategy
{
   public void strategyInterface()
   {
            //算法实现代码，上述问题中是某种促销的算法
   }
}
```

main 方法：

```
Public void main(){
//代码仅供参考。。。
context context=new context();
array；//购物车中的商品数组
length;//数组的元素个数
While(length>=0){
context. Setstrategy(array[length]);
context. contextInterface();
length--;
}
}
```

模式评价：

优点：

① 消除了 if else 条件语句，增强了代码的可读性、维护和修改更加的简单。

② 将具体策略封装，使得可以独立于 context 修改代码，易于扩展、切换、理解。
③ Strategy 类为 context 类定义了一系列的可供重用的算法或行为，继承有助于提取这些算法的公共功能。
④ 实现的选择：策略模式可以提供相同行为的不同实现，客户可以根据不同时间、不同空间的要求从不同策略中进行选择。

缺点：

① 客户端必须知道所有的策略类，并自行选择使用的策略类。
② 策略模式将造成产生很多策略类，增大了存储开销。
③ Context 类和 strategy 类之间的通信开销。

4.10 责任链模式

1 责任链的定义

定义：使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有个对象处理它为止。从定义上可以看出，责任链模式的提出是为了“解耦”，以应变系统需求的变更和不明确性 。

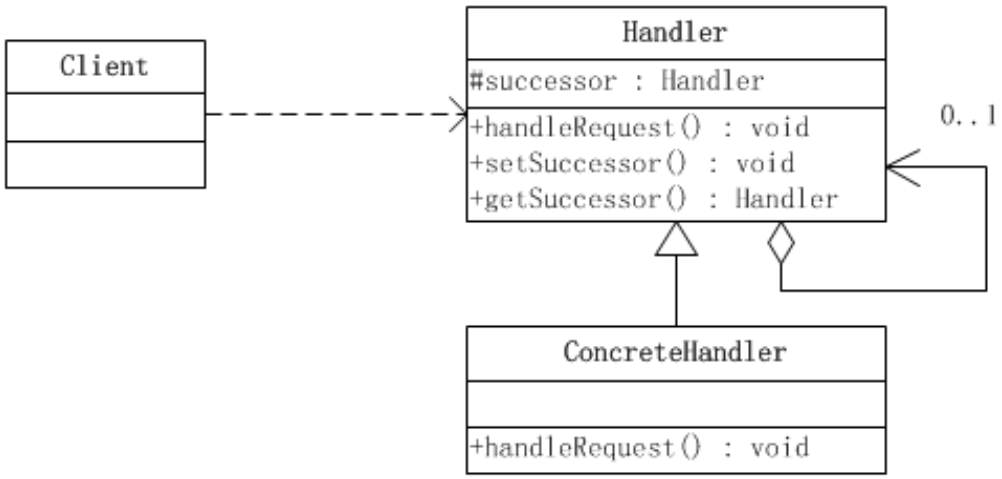
2 责任链的应用场景

1) 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定

2) 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

3) 可动态指定一组对象处理请求。

3 类图



抽象处理者(Handler)角色：定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个 Java 抽象类或者 Java 接口实现。上图中 Handler 类的聚合关系给出了具体子类对下家的引用，抽象方法 handleRequest()规范了子类处理请求的操作。

具体处理者(ConcreteHandler)角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

4．举例说明

我们就拿一个最简单最能说明问题的例子来说，那就是报销费用的问题。员工想要报销费用，低于 500 的项目经理就可以报销，超过 500 但是小于 1000 的部门经理可以报销，超过 1000 的只能找总经理。下面是给出的抽象处理类和具体处理类的代码

抽象的处理者类

```
public abstract class Handler {
   protected Handler successor = null;
```

```
    public Handler getSuccessor() {
        return successor;
    }
    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }
    public abstract String handleFeeRequest(double fee);
}
```

项目经理只能报销 500 以下的

```
public class ProjectManager extends Handler {
        String str = "";
    public String handleFeeRequest(double fee) {
        if(fee < 500){
          str = "成功：项目经理同意报销聚餐费用，金额为" + fee + "元"；
        }else{       //超过 500 只能交给后继来处理
            if(getSuccessor() != null){
                return getSuccessor().handleFeeRequest( fee);
        }
        return str;
    }
```

部门经理

```
public class DeptManager extends Handler {
    public String handleFeeRequest(double fee) {
        String str = "";
        if(fee < 1000){
          str = "成功：部门经理同意报销聚餐费用，金额为" + fee + "元";
        }else{       //超过 1000 只能交给后继来处理
            if(getSuccessor() != null){
                return getSuccessor().handleFeeRequest( fee);
            }
        }
        return str;
    }
}
```

总经理

```
public class GeneralManager extends Handler {
    public String handleFeeRequest(double fee) {
        String str = "";
        if(fee >= 1000){
          str = "成功：总经理同意报销聚餐费用，金额为" + fee + "元";
        }else{
            if(getSuccessor() != null){
                return getSuccessor().handleFeeRequest(fee);
            }
```

```
    }
    return str;
  }
}
```

很多人可能觉得这代码很多余，明明自己可以用几个 ifelse 在客户端之间实现，但是这种情况下，请求的发送者就是和这三个类都发生了耦合。当报销政策发生改变时，那么 ifelse 的代码就需要修改，这个时候就是请求的发送者和接受者都修改了代码，没有达到解耦合的效果。

5.优缺点

优点：在职责链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

缺点：职责链模式的主要优点在于可以降低系统的耦合度，简化对象的相互连接，同时增强给对象指派职责的灵活性，增加新的请求处理类也很方便；其主要缺点在于不能保证请求一定被接收，且对于比较长的职责链，请求的处理可能涉及到多个处理对象，系统性能将受到一定影响，而且在进行代码调试时不太方便。

第五章 面向对象的设计原则

这一章是基于之前内容的进一步抽象，设计原则在考试中主要在重构题和设计题里出现，一定要理解相应原则的具体含义，当前类的设计里是否违反了设计原则，应该如何应用设计原则进行修改。

编者答疑：孙吉鹏 林子童

5.1 开—闭原则（Open-Closed Principle）

5.1.1 定义

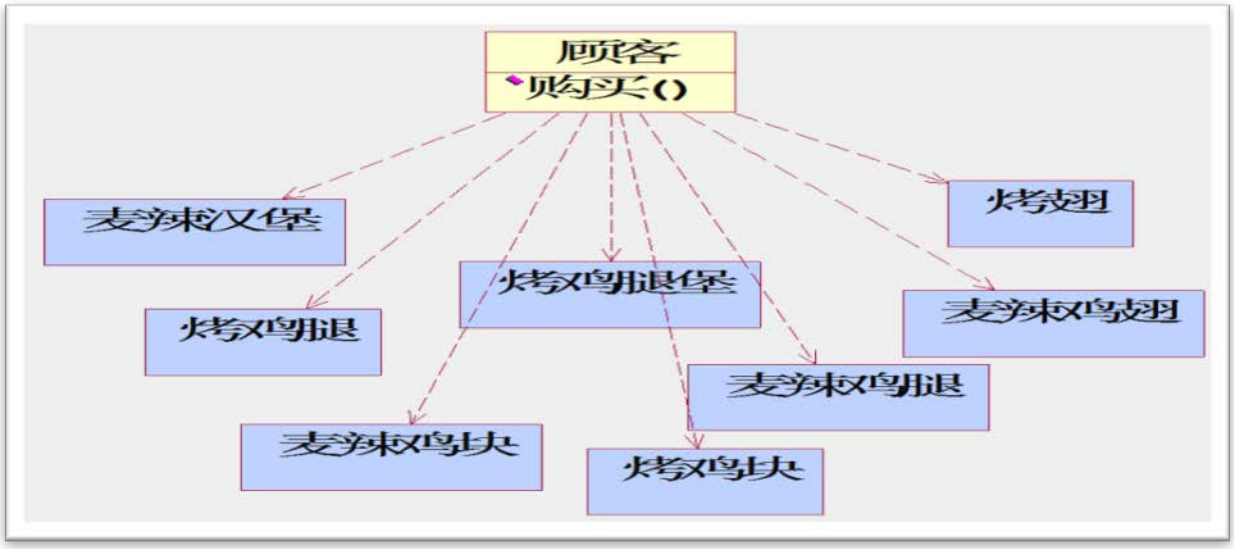
软件组成实体应该是可扩展的，但是不可修改的。

Software entities should be open for extension, but closed for modification.

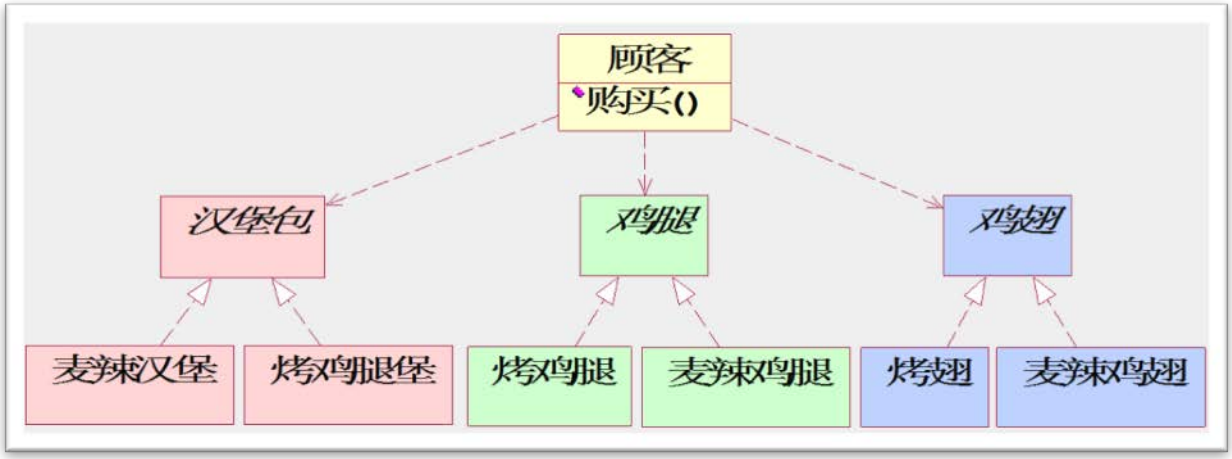
5.1.2 概念理解

开闭原则的提出是为了满足当系统需求发生了变化时，希望已经设计好的系统可以不修改原有类的代码，而是通过增加新的类来应对需求的变化。即在设计系统时，就要提前预知到可能发生变化的点，对这样的变化点要分离出一层抽象层，然后系统只针对这个抽象层进行编程，而不对变化点（实现层）编程，当有新的变化点时，只需要新写一个变化点的类，让它继承原有的抽象层的类，这样的变化不会对系统其它部分有影响。

举个例子，比如这个点餐情况，可能发生的点在于麦当劳每月都会推出一款新的小吃，如果像上面的类设计，比如增加麦香鱼汉堡时，我们不仅需要新加一个麦香鱼汉堡类，还需要修改顾客类里的代码，这会导致顾客里代码正确性需要重新验证。



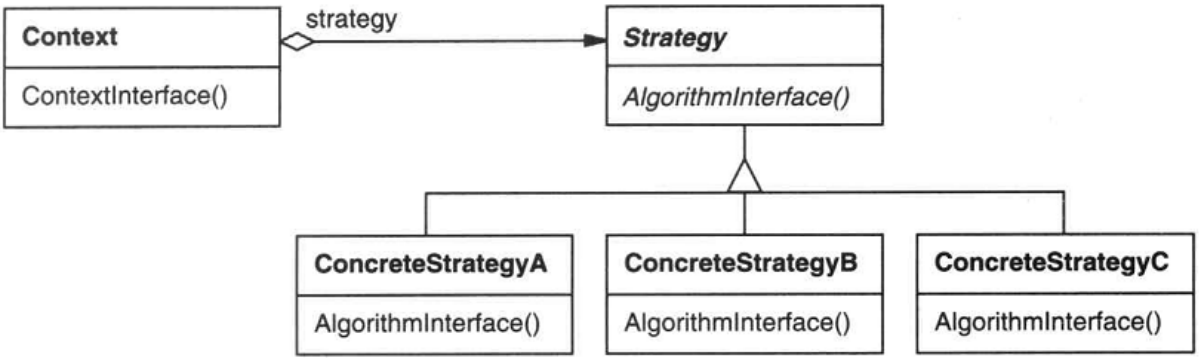
但如果我们采用下图的架构，我们发现只需要让麦香鱼汉堡类继承汉堡包类就可以实现功能的拓展而不需要修改顾客类，因为顾客类依赖的是汉堡包类啊，汉堡包类本身并没有发生任何改变。



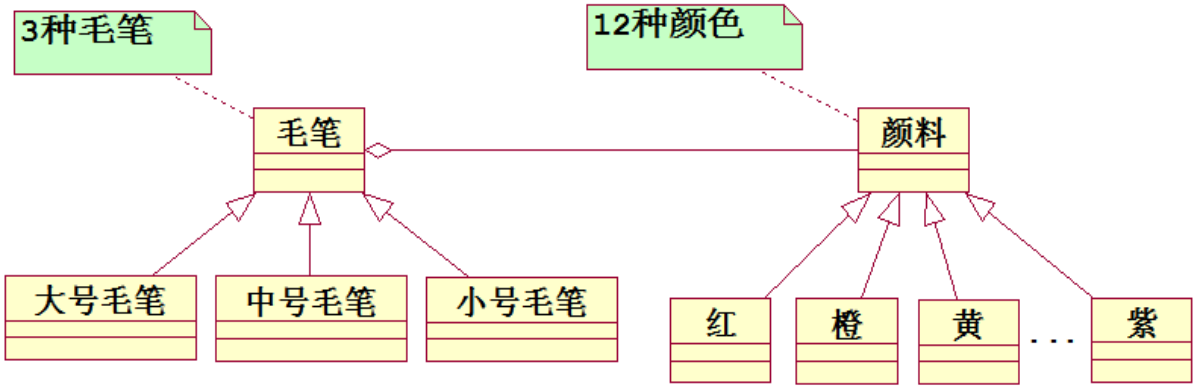
通过上述的例子我们发现，开闭的关键在于设计上的抽象化， 可给系统定义一个一劳永逸，不再更改的抽象设计，此设计允许有无穷无尽的行为在实现层被实现。抽象层预见所有扩展。上面这个类并没有预见所有改变，比如如果新增草莓奶昔小吃，这个设计就要修改顾客类了，因为草莓奶昔目前没有对应的抽象类。

5.1.3 设计模式实现例子

策略模式里的 Strategy 类（抽象层，Context 是基于这个类进行的架构）



桥梁模式里的毛笔类和颜料类（抽象层，新加类时只需扩展，不用修改原有类）



5.1.4 如何判断是否违反

假设出现新的变化，能否只通过扩展来满足：如果能，不违反；如果需要修改已有的类，违反。

5.1.5 如何修改使系统符合

将变化点抽象出一层，结合具体设计模式，进行改进。

5.2 里氏替换原则（Liskov Substitution Principle）

5.2.1 定义

使用指向基类(超类)的引用的函数，必须能够在不知道具体派生类(子类)对象类型的情况下使用它们。

Function That Use References To Base(Super) Classes Must Be Able To Use Objects Of Derived(Sub) Classes Without Knowing It.

5.2.2 概念理解

凡是父类适用的地方子类应当也适用，即父类应当是子类的充分条件，不能在某些情景下父类可以做到而子类做不到。形象的讲继承树的形状应当是正三角，而不是倒三角。如常举的例子“圆不是椭圆”，“企鹅不是鸟”等，都是因为子类做不到父类的某些行为而导致了在想使用父类实现多态时出现了问题。举个“正方形不是矩形”的例子说明一下为什么可能出现问题。

```
Public class Square extends Rectangle{//正方形类
  public Square(double s) {
    super(s,s);
  }
  public void setWidth(double w){ /*为了实现父类特有的方法而强行扩展了语义，函数名只是设置宽，但是这个类却一下子设置了长和宽* /
    super.setWidth ( w );
    super.setHeight( w );
  }
 public void setHeight(double h){
    super.setWidth ( h );
    super.setHeight( h );
  }
}
Public class TestRectangle{//测试类
  public static void testLSP(Rectangle r) {
    r.setWidth(4.0);
    r.setHeight(5.0);
    System.out.println(“Width is 4.0 and Height is 5.0 ,Area is ” + r.area()); / *测试人员以为是符合 LSP 的，所以认为所有长方形都应该具有长宽之间可以不同的特性，即长宽两个属性是独立的。* /
  }
public static void main(String args[]){
  Rectangle r=new Rectangle(1.0,1.0);
  Square s = new Square(1.0);

  testLSP ( r );
  testLSP ( s );//子类对象也可以在父类对象适用的情况下替换
  }
}
```

打注释的三个地方，修改哪一处地方都可以避免最后的结果错误，但修改的前提都是需要知道所有代码才能有效避免。出现问题的原因本质上就是设计时把面向对象的“is

a”继承概念和生活中语义上的“is a”搞混了。LSP 里的“A is a B”是指 B 的行为是 A 的子集的意思，只要是行为上完全包含一个类就可以是这个类的子类，并没有对生活上的“语义”进行过多限制。LSP 父类和子类的行为要一致，自然。不能为了成为某个类的子类而对这个类的行为强行限制。

所以，何时使用继承？符合语义上的“is a”+符合行为上的 LSP

5.2.3 如何修改不符合 LSP 的继承

通过组合的方式来使用某个类的部分功能而不是继承所有功能。如上例中 Rectangle 类和 Square 类，我们通过分析知道不能使用继承 Rectangle 类来完成 Square 类的架构，而是在 Square 类里声明一个 Rectangle 对象，委托 Rectangle 来完成相应的 setEdge（），getEdge（）等方法。

```
 Public class Square{
Public Square(double e){
 R = new Rectangle(e,e);
}
public void setEdge(double a){
R.setWidth(a);
R.setHeight(a);
}
public double  getEdge(){
return R.getHeight();
}
private:
Rectangle R;
}
```

5.3 依赖倒转原则（Dependency Inversion Principle）

5.3.1 定义

抽象不应当依赖于细节，细节应当依赖于抽象。

Abstractions should not depend upon details; details should depend upon abstractions.

5.3.2 概念理解

在没有面向对象之前，我们的功能用函数实现，传给函数的参数因为没有抽象所以都是具体的基本数据类型或者具体的指针，所以我们的函数高度依赖于具体的实现层，无法通过多态实现复用。产生面向对象后，思维很大程度上还有结构化编程的惯性，架构层依旧依赖于具体的实现类（如传入一个具体类类型当参数）来实现功能，导致无法充分发挥面向对象的优势，易变的实现层作为架构层的底层势必会导致系统频繁的改动（如之前的麦当劳例子），DIP 就是针对这种情况提出的。

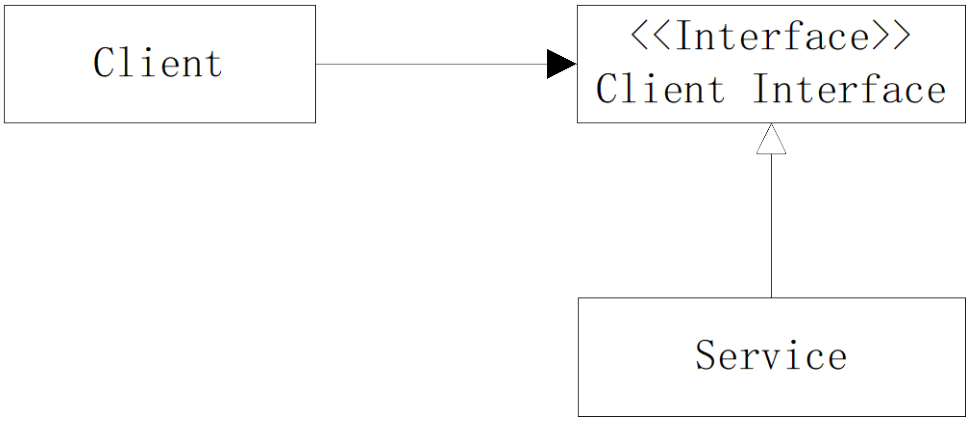
为了解决这个问题，让依赖关系变为依赖于抽象，方法就是针对接口编程，不要针对实现编程，“Program to an interface, not an implementation.” 具体实现建议就是变量、参数、返回值等应声明为抽象类；不要继承非抽象类；不要重载父类的非抽象方法。即不将变量声明为某个特定的具体类的实例对象，而让其遵从抽象类定义的接口。实现类仅实现接口，不添加方法。举个例子，设计 Draw（）方法传进来的参数类型时，要是 Draw(shape*p)， 不要设计成 Cricle*p Rectangle *p Triangle *p。

5.3.3 如何修改两个具体类之间存在的依赖关系

如果一个类的实例必须使用另一个对象，而这个对象又属于一个特定的类，那么复用性会受到损害。比如下图的架构

比如这是一个客户端获取观测站天气预报信息的系统，此时 Client 类高度依赖于具体的 Service 类来提供服务，一旦客户端需要更换使用另一个 Servic 类（更换不同地区的观测站）提供服务时，就要重新写 Client 类代码。

这时，我们采用让被使用的 Service 类抽象出一个借口来实现，这个借口里定义了所有气象站都需要提供的服务，比如 getWeather（），getRainfull（）等方法，而 Client 类获取服务的方法只需要传入一个 ClientInterface 类型的参数就可以，所有实现这个接口的 Service 类都可以利用多态传入 Client 类，从而实现 Client 类的实现不依赖于具体的 Service 类。



5.4 组合复用原则（Composition Reuse Principle）

5.4.1 定义

优先使用(对象)组合，而非(类)继承

Favor Composition Over Inheritance

5.4.2 概念理解

组合复用原则又叫组合优先原则，强调的是优先使用更加灵活的组合方式来实现功能。继承作为重要的重用方法，在面向对象早期被过度使用。回忆一下我们第一门面向对象语言学习时我们对接口的理解远远不如对继承这种具体易用的重用方式的理解深刻。

在没有理解设计原则时对继承的过度使用放大了继承的缺点：

（1）继承破坏了封装性，因为这会将父类的实现细节暴露给子类。

（2）当父类的实现更改时，子类也不得不会随之更改。

（3）从父类继承来的实现将不能在运行期间进行改变。

而组合的优点：

（1）容器类仅能通过被包含对象的接口来对其进行访问，所以这是“黑盒”复用，因为被包含对象的内部细节对外是不可见。

（2）实现上的相互依赖性比较小。

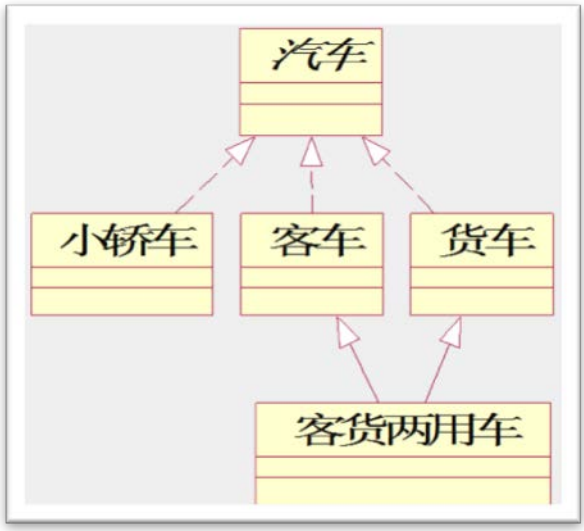
（3）通过获取指向其它的具有相同类型的对象引用，可以在运行期间动态地定义(对象的)组合。

恰恰解决了继承的缺点，所以使用组合可以实现几乎继承的所有功能，只不过在设计时更复杂，语义上更难理解。

所以，组合优先这条原则并不能拘泥，而是根据现实状况，符合语义上的“is a”关系和行为上的里氏替换原则，就使用更易理解和实现的继承；否则，使用组合。

5.4.3 如何将不合理继承转化为组合

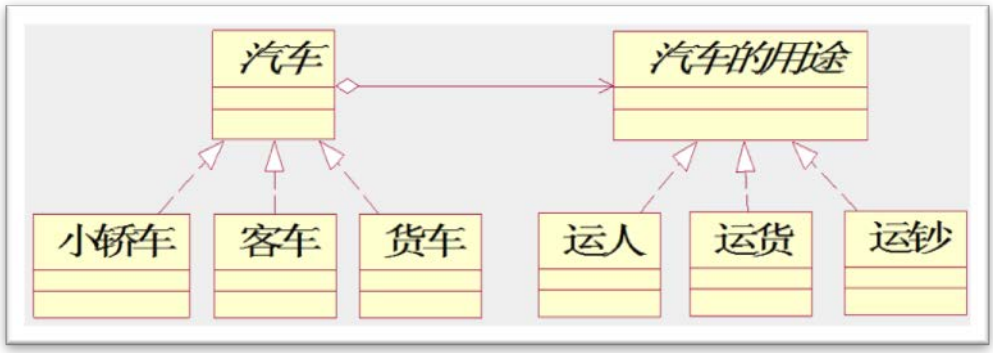
如下图这个例子，为了实现客货两用车这个功能，如果使用继承，不得不实现很多语言并不支持的充满风险的多继承技术，这时，考虑用组合来实现。



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



之前在里氏替换原则里的 Square 与 Rectangle 例子也是改造的例子。

5.5 迪米特法则（Low of Demeter）

5.5.1 定义

又称 最少知识原则，一个对象应该对其他对象尽可能少的了解。

5.5.2 概念理解

面向对象设计要求我们应当减少各个类之间的耦合，避免类之间的不必要依赖，一个类修改时尽可能地少影响其它类，这些目标都需要每个类之间要相对独立，而且之间的联系要精简。这就引出了最小知识法则，避免类间不必要依赖。

首先要定义好哪些对象是需要关联的（朋友）：

（1） 当前对象本身（this）
（2） 当作参数传入当前对象方法中的对象
（3） 当前对象的实例变量直接引用的对象
（4） 当前对象的实例变量如果是一个聚集，那么聚集中的元素也是朋友
（5） 当前对象所创建的对象
（6） 当前对象的组件

举个例子：

```
public  class Car {
        Engine engine;//这是类的一个组件，我们能够调用它的方法
        public Car () {
        }
        public void start (Key key) //被当作参数传进来的对象，其方法可以被调用。
                Door doors = new Doors();//在这里创建了一个新的对象，它的方法可以被调用。
                boolean authorized = key.turns();
                if (authorized)  {
                        engine.start();//可以调用对象组件的方法
                        updateDashboardDisplay();//可以调用 local method
                        doors.lock();//可以调用创建或实例化的对象的方法
                }
        }
        public void updateDashboardDisplay() {
        }
}
```

常见的出现问题的关联：

如果某对象是调用其他方法的返回结果，不要调用该对象的方法。（相当于向另一个对象的子部分发请求（增加了我们直接认识的对象数目）；原则要我们改为要求该对象为我们做出请求，这么一来，我们就不需要认识该对象的组件了。）

5.5.3 如何修改不符合迪米特法则的类关系

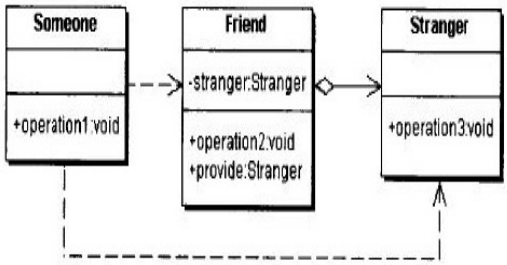
```
Void Someone::Operation1(Friend friend){
  Stranger stranger = friend.provide();
  stranger.Operation3();
}
```

```
Stranger Friend::provide(){
  return stranger;
}
```

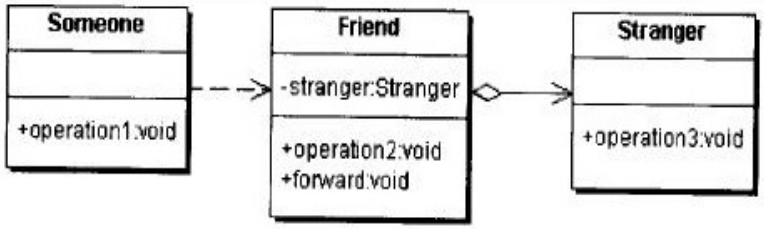
画出这个类的类图为：

不满足迪米特法则的系统

这里要讨论的系统由三个类组成，分别是 Someone，Friend 和 Stranger。其中 Someone 与 Friend 是朋友，而 Friend 与 Stranger 是朋友。系统的结构图如下图所示。



问题关键出在了 Someone 里的 operation1 通过 Friend 类返回 Stranger 调用了 Stranger 的 operation3，导致了多余的耦合，即 Someone 和 Stranger 产生了依赖关系，因为 Someone 在知道 Friend 类的条件下却还需要额外知道 Stranger 类才能实现功能，这显然是不合理的，因为稍作修改，Someone 就可以不需要了解 Stranger 类了，而是在 Friend 中加入一个委托 Stranger 进行 operation3 操作的 forward（）方法，而 Someone 的 operation1 只需要将 Friend 类当作参数传进来即可实现操作。



从上面的类图可以看出，与改造前相比，在 Someone 与 Stranger 之间的联系已经没有了。Someone 不需要知道 Stranger 的存在就可以做同样的事情。Someone 的源代码如代码清单 3 所示。

代码清单 3：Someone 类的源代码

```
public class Someone
{
    public void operation1(Friend friend)
    {
        friend.forward();
    }
}
```

也就是说，这样的调用返回对象方法的违反迪米特法则的设计，可以由一个以中介类为参数的函数和中介类组合一个目标类的方式来改造，即 A 关联 B 组合 C 模式。

5.6 接口隔离原则（Interface Segregation Principle）

5.6.1 定义

一个类对另一个类的依赖应当建立在最小的接口上

The dependency of one class to another one should depend on the smallest possible interface

使用多个专门的接口比使用单一的总接口好。

5.6.2 概念理解

客户端不应该依赖它不需要的接口，一个接口应当简单的只代表一个角色，如果一个接口中有多种没有关系的功能，那么接口的实现类就会被污染，也就会违反迪米特法则，并使得信息的隐藏、封装性变差，所以，接口隔离原则体现的是高内聚、低耦合的观点

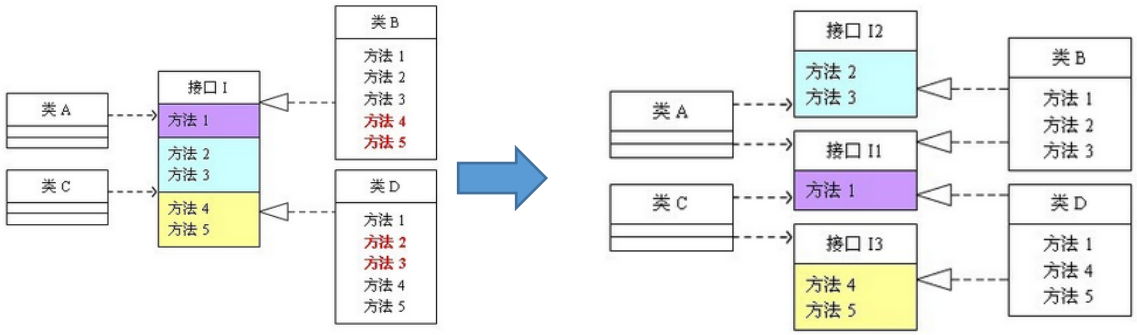
如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计

其中需要注意接口隔离原则和单一职责原则的区别：

其一，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口的方法尽量少。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。例如一个接口的职责可能包含 10 个方法，这 10 个方法都放在一个接口中，并且提供给多个模块访问，各个模块按照规定的权限来访问，在系统外通过文档约束“不使用的方法不要访问”，按照单一职责原则是允许的，按照接口隔离原则是不允许的，

5.6.3 接口隔离的实例

类 A 通过接口 I 依赖类 B，类 C 通过接口 I 依赖类 D，如果接口 I 对于类 A 和类 B 来说不是最小接口，则类 B 和类 D 必须去实现他们不需要的方法。所以，我们就要将臃肿的接口 I 拆分为独立的几个接口，类 A 和类 C 分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。



5.6.4 使用技巧和原则

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

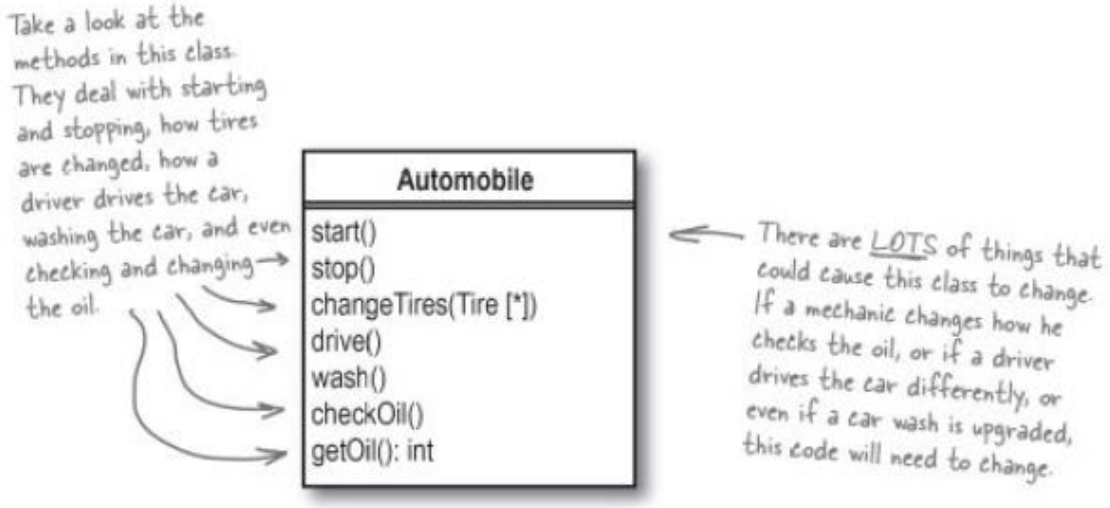
5.7 单一职责（Single Responsibility Principle）

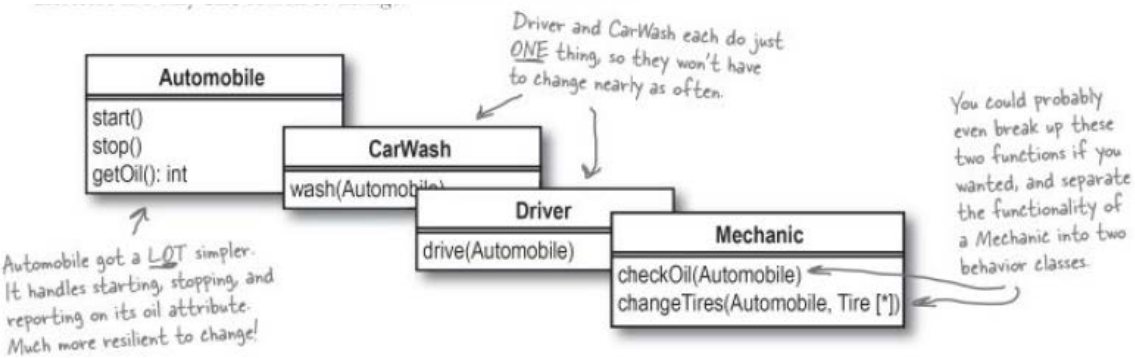
5.7.1 定义

不要存在多于一个导致类变更的原因。

5.7.2 概念解读

当一个类负责多个职责时，每个职责的变化都会导致这个类的代码的变化，这就导致修改时明明没有改动的所有其他职责方法却要重新因为某个职责的改变而重新进行测试，这不符合我们对类的功能要有高内聚的要求，即类内方法变化原因一致，因此引入了单一职责的设计原则。

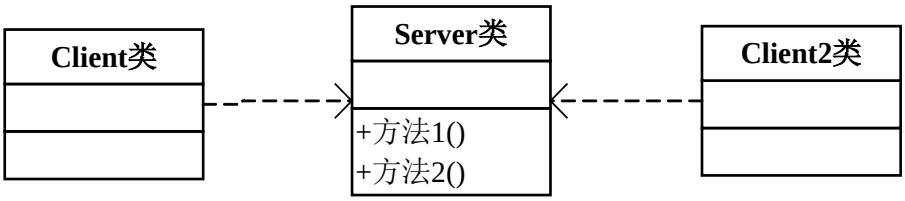




这两张类图的对比可以看出，第二种设计每个类的变化原因是一致的，所以修改时不会对其他类产生粘连

5.7.3 如何修改多职责类

如下所示类图结构，具体类 Client 调用了具体类 Server 中的方法，来实现业务逻辑。同时 Server 类还被具体类 Client2 类调用，这样如果 Client2 类需要 Server 类中的方法 1()和方法 2()的方法实现发生变动时，将影响 Client 类的业务逻辑。请设计一个好的方案，使 Server 类因 Client2 类要求发生变更的时候，不影响 Client 类的业务逻辑。请画出调整后的类图。



我们发现影响 Server 变化的原因同时有 Client 和 Client2 两个职责，而且这两个 Client 的变化是独立的，所以让一个 Server 类同时负责两个 Client 是违反了单一职责原则的，应当将 Server 类分成两个类 Server1，Server2，每个类中分别有方法 1（），方法 2（），这样 Client2 的需求变化导致的 Server2 的方法变化时不会影响 Server1 的方法，从而实现了两个 Client 类的独立。

1