我们在进行面向对象设时应该怎样进行，遵循什么原则呢？面向对象的设计从提出到现在经过很多人的经验和实践，总结出了很多原则。这些被大师们总结出来的基本原则包括了**类的设计原则**和**包的设计原则**，后者又分为包的内部关系方面（聚合性）的原则以及包之间的关系方面（耦合性）的原则。这里我们只讨论类的设计原则。

类的设计原则有七个，包括：开闭原则、里氏代换原则、迪米特原则、单一职责原则、接口分隔原则、依赖倒置原则、组合/聚合复用原则。

七大原则之间并不是相互孤立的，彼此间存在着一定关联，一个可以是另一个原则的加强或是基础，违反其中的某一个，可能同时违反了其余的原则。正如牛顿三大定律在经典力学中的位置一样，开闭原则是面向对象的可复用设计的基石，其他设计原则是实现开闭原则的手段和工具。

一般地，可以把这七个原则分成了以下两个部分：

设计目标：开闭原则、里氏代换原则、迪米特原则

设计方法：单一职责原则、接口分隔原则、依赖倒置原则、组合/聚合复用原则

一、 开闭原则（The Open-Closed Principle ，OCP）

软件实体（模块，类，方法等）应该对扩展开放，对修改关闭。

开闭原则是指在进行面向对象设计中，设计类或其他软件实体时，应该遵循：

- 对扩展开放（open）
- 对修改关闭（closed）

的设计原则。

开闭原则是判断面向对象设计是否正确的最基本的原理之一。

根据开闭原则，在设计一个软件模块（类，方法）的时候，应该可以在不修改原有的模块（修改关闭）的基础上，能扩展其功能（扩展开放）。

- 扩展开放：某模块的功能是可扩展的，则该模块是扩展开放的。**软件系统的功能上的可扩展性**要求模块是扩展开放的。
- 修改关闭：某模块被其他模块调用，如果该模块的源代码不允许修改，则该模块修改关闭的。**软件系统的功能上的稳定性，持续性要求**是修改关闭的。

这也是**系统设计需要遵循开闭原则的原因：**

1. 稳定性。开闭原则要求扩展功能不修改原来的代码，这可以让软件系统在变化中保持稳定。
2. 扩展性。开闭原则要求对扩展开放，通过扩展提供新的或改变原有的功能，

让软件系统具有灵活的可扩展性。

遵循开闭原则的系统设计，可以让软件系统可复用，并且易于维护。

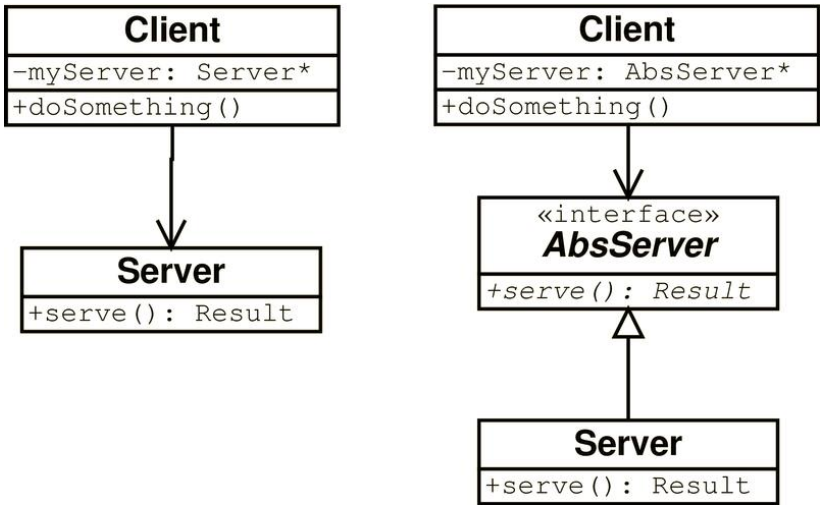
开闭原则的实现方法

为了满足开闭原则的对修改关闭原则以及扩展开放原则，应该对软件系统中的不变的部分加以抽象，在面向对象的设计中，

- 可以把这些不变的部分加以抽象成不变的接口，这些不变的接口可以应对未来的扩展；
- 接口的最小功能设计原则。根据这个原则，原有的接口要么可以应对未来的扩展；不足的部分可以通过定义新的接口来实现；
- 模块之间的调用通过抽象接口进行，这样即使实现层发生变化，也无需修改调用方的代码。

接口可以被复用，但接口的实现却不一定能被复用。接口是稳定的，关闭的，但接口的实现是可变的，开放的。可以通过对接口的不同实现以及类的继承行为等为系统增加新的或改变系统原来的功能，实现软件系统的柔性扩展。

简单地说，软件系统是否有良好的接口（抽象）设计是判断软件系统是否满足开闭原则的一种重要的判断基准。现在多把开闭原则等同于面向接口的软件设计。



Client 对于 Server 提供的接口是修改关闭的

Client 对于 Server 的新的接口实现方法是扩展开放的

Shape 例子－过程化

Shape.h

```
enum ShapeType{ isCircle, isSquare};
```

```
typedef struct Shape {
        enumShapeType type
} shape;
```

Circle.h

```
typedef struct Circle {
        enum ShapeType type;
        double radius;
        Point center;
} circle;
void drawCircle( circle* );
```

Square.h

```
typedef struct Square {
        enum ShapeType type;
        double side;
        Point topleft;
} square;
void drawSquare( square* );
```

drawShapes.cpp

```
#include "Shape.h"
#include "Circle.h"
#include "Square.h"
void drawShapes( shape* list[], int n ) {
        int i;
        for( int i=0; i<n; i++ ) {
                shape* s= list[i];
                switch( s->type ) {
                case isSquare:
```

```
            drawSquare( (square*)s );
            break;
        case isCircle:
            drawCircle( (circle*)s );
            break;
        }
    }
}
```

增加 1 个新的图形需要修改哪些地方?

•drawShapes 不是封闭的

–switch/case 可能需要出现在多个地方

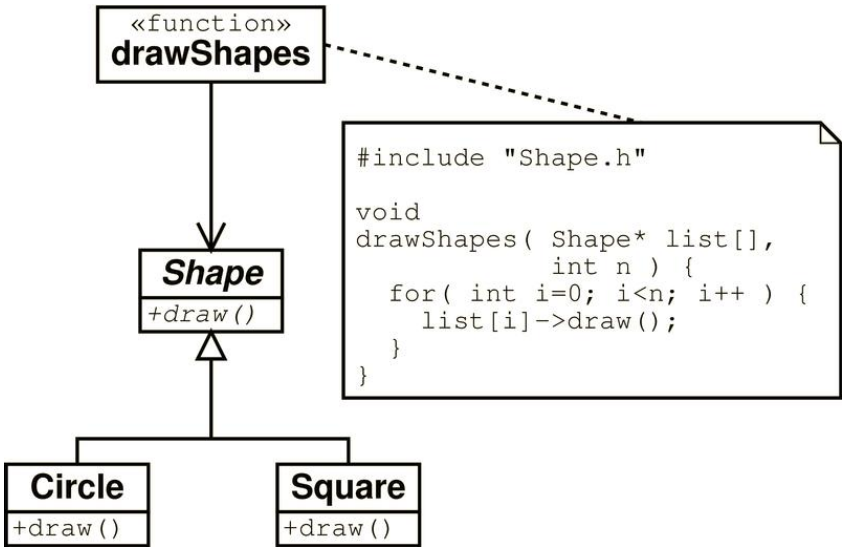
–增加一个图形→修改 switch/case

–逻辑复杂

–扩展枚举类型 ShapeType→重新编译所有的程序

•这是一个僵化的、脆弱的、具有很高的牢固性的设计

良好的设计：



开闭原则的相对性

软件系统的构建是一个需要不断重构的过程，在这个过程中，模块的功能抽象，模块与模块间的关系，都不会从一开始就非常清晰明了，所以构建 100%满足开闭原则的软件系统是相当困难的，这就是开闭原则的相对性。但在设计过程中，通过对模块功能的抽象（接口定义），模块之间的关系的抽象（通过接口调用），抽象与实现的分离（面向接口的程序设计）等，可以尽量接近满足开闭原则。

参考资料：

Martin, Robert C.（1996 January）. “The Open-Closed Principle”

二、 里氏替换原则（Liskov Substitution Principle ，LSP）

所有引用基类的地方必须能透明地使用其派生类的对象。

也就是说，只有满足以下 2 个条件的 OO 设计才可被认为是满足了 LSP 原则：

- **不应该在代码中出现 if/else 之类对派生类类型进行判断的条件。**
  以下代码就违反了 LSP 定义。

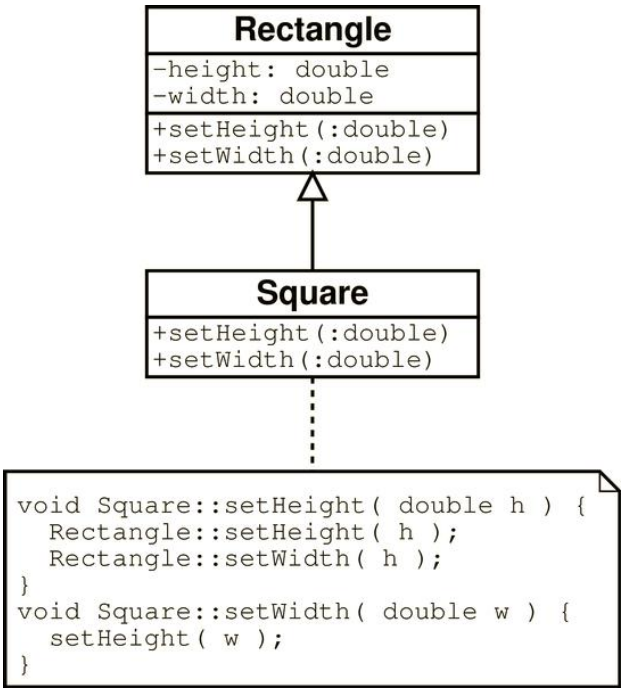
```
void DrawShape(const Shape& s) {
    if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```

- **派生类对象应当可以替换基类对象并出现在基类对象能够出现的任何地方，或者说如果我们把代码中使用基类对象的地方用它的派生类对象来代替，代码还能正常工作。**

里氏替换原则 LSP 是使代码符合开闭原则的一个重要保证。同时 LSP 体现了：

- 类的继承原则：如果一个派生类对象可能会在替换基类对象的地方出现运行错误，则该派生类不应该从该基类继承，或者说，应该重新设计它们之间的关系。
- 动作正确性保证：从另一个侧面上保证了符合 LSP 设计原则的类的扩展不会给已有的系统引入新的错误。

示例：



这里 Rectangle 是基类，Square 从 Rectangle 继承。

这种继承关系有什么问题吗？

假如已有的系统中存在以下既有的业务逻辑代码：

```
void g(Rectangle& r) {
   r.SetWidth(5);
   r.SetHeight(4);
   assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

则对应于扩展类 Square，在调用既有业务逻辑时：

```
Rectangle* square = new Square();
g(*square);
```

时会抛出一个异常。这显然违反了 LSP 原则。

例如鲸鱼和鱼，应该属于什么关系？从生物学的角度看，鲸鱼应该属于哺乳动物，而不是鱼类。没错，在程序世界中我们可以得出同样的结论。如果让鲸鱼类去继承鱼类，就完全违背了 Liskov 替换原则。因为鱼作为基类，很多特性是鲸鱼所不具备的，例如通过腮呼吸，以及卵生繁殖。那么，二者是否具有共性呢？ 有，

那就是它们都可以在水中"游泳"，从程序设计的角度来说，它们都共同实现了一个支持"游泳"行为的接口。

例如运动员和自行车例子，每个运动员都有一辆自行车，如果按照下面设计，很显然违反了 LSP 原则。

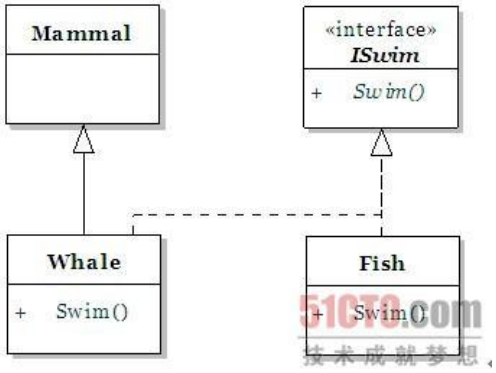
```
class Bike {
public:
        void Move( );
        void Stop( );
        void Repair( );
protected:
        int     ChangeColor(int );
private:
        int     mColor;

};

class Player : private Bike
{
public:
       void   StartRace( );
       void   EndRace( );
protected:
        int     CurStrength ( );

private:
         int    mMaxStrength;
         int    mAge;
};
```

如果两个具体的类 A，B 之间的关系违反了 LSP 的设计，那么根据具体的情况可以在下面的两种重构方案中选择一种：

1．创建一个新的抽象类 C，作为两个具体类的基类，将 A，B 的共同行为移动到 C 中来解决问题。
2．从 B 到 A 的继承关系改为关联关系。

对于长方形和正方形例子，可以构造一个抽象的四边形类，把长方形和正方形共同的行为放到这个四边形类里面，让长方形和正方形都是它的派生类，问题就解决了。对于长方形和正方形，取 width 和 height 是它们共同的行为，但是给 width 和 height 赋值，两者行为不同，因此，这个抽象的四边形的类只有取值方法，没有赋值方法。

对于鱼和鲸鱼例子，可以按下图重新设计：



对于运动员和自行车例子，可以采用关联关系来重构：

```
class Player

{
public:
        void    StartRace( );
        void    EndRace( );
protected:
         int       CurStrength ( );

private:
          int     mMaxStrength;
          int     mAge;

          Bike * abike;
};
```

在进行设计的时候，我们**尽量从抽象类继承，而不是从具体类继承**。如果从继承等级树来看，所有叶子节点应当是具体类，而所有的树枝节点应当是抽象类或者接口。当然这只是一个一般性的指导原则，使用的时候还要具体情况具体分析。

在很多情况下，在设计初期我们类之间的关系不是很明确，LSP 则给了我们一个判断和设计类之间关系的基准：需不需要继承，以及怎样设计继承关系。

参考资料：

Martin, Robert C. “Liskov Substitution Principle”

三、 迪米特原则（最少知道原则）（Law of Demeter，LoD）

迪米特原则（Law of Demeter）又叫最少知道原则（Least Knowledge Principle），1987 年秋天由美国 Northeastern University 的 Ian Holland 提出，在 OOPSLA '88 Proceedings 发表（程序语言领域国际顶级学术会议，也是中国计算机学会推荐 A 类国际学术会议），被 UML 的创始者之一 Booch 等普及。后来，因为在经典著作《The Pragmatic Programmer（程序员修炼之道）》中提出而广为人知。

迪米特原则可以简单说成：talk only to your immediate friends，只与你直接的朋友们通信，不要跟”陌生人”说话。进一步可理解为两个层次：

1）一个软件实体应当尽可能少地与其他软件实体发生相互作用（只和你的”朋友”通信）。

2）每一个软件实体对其他软件实体都只有最少的知识，而且局限于那些与本软件实体密切相关的软件实体（跟”朋友”通信越少越好，具体来说就是一个类对自己依赖的其它类知道的越少越好）。

朋友圈的确定，”朋友”条件：

1）当前对象本身（this）

2）以参数形式传入到当前对象方法中的对象（依赖）

3）当前对象的实例变量直接引用的对象（关联）

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都是朋友（聚集）

5）当前对象所创建的对象（组合）

任何一个对象，如果满足上面条件之一，就是当前对象的”朋友”，否则就是”陌生人”。

解析一：出现在成员变量，方法参数，方法返回值中的类为本类的朋友，而出现在局部变量中的类则不是朋友。此观点就是告诉我们尽量不要在本类中通过局部变量的形式使用其它陌生类。

解析二：尽量不暴露独属于类本身的方法和属性，可以简单理解为，每个类都是要有属于自己的秘密，这样就可以使其它类对自己知道的更少。

迪米特原则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

迪米特原则不希望类之间建立直接的接触。如果真的有需要建立联系，也希望能建立较少的联系，或者通过中介类来转达。因此，应用迪米特原则有可能造成的一个后果就是：系统中存在大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系，这在一定程度上增加了系统的复杂度。

例如，购房者要购买楼盘 A、B、C 中的楼，他不必直接到楼盘去买楼，而是可以通过一个售楼处去了解情况，这样就减少了购房者与楼盘之间的耦合，如图所示。

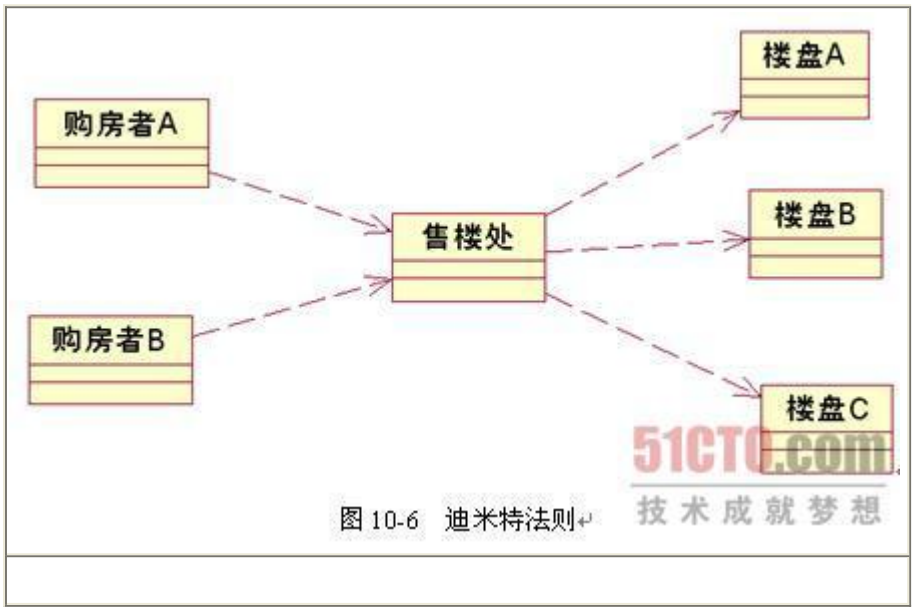


图 10-6 迪米特法则

示例一，下面的代码在方法体内部依赖了其他类，违反了迪米特原则

```
class Teacher {

public:

  void command(GroupLeader groupLeader) {

        list<Student> listStudents = new list<Student>;

        for (int i = 0; i < 20; i++) {

              listStudents.add(new Student());

        }

        groupLeader.countStudents(listStudents);

  }

};
```

方法是类的一个行为，类竟然不知道自己的行为与其他类产生了依赖关系，这是不允许的。

调整后的代码是：

```
class Teacher {
public:
   void command(GroupLeader groupLeader) {
                groupLeader.countStudents();
   }
};
class GroupLeader {
private:
   list<Student> listStudents;
public:
   GroupLeader(list<Student> _listStudents) {
                this.listStudents = _listStudents;
   }
   void countStudents() {
           cout<<"女生数量是：" <<listStudents.size() <<endl;
    }
};
```

示例二，下面的代码对”朋友”知道的太多，也违反了迪米特原则

```
class WashingMachine{
public:
   void receiveClothes();
   void wash();
   void drying();
```

```
};

class Person{

public:

   void washClothes(WashingMachine& wm){

      wm.receiveClothes();

      wm.wash();

      wm.drying();

   }

};
```

对于 Person 类来说，要实现洗衣服的功能，具体洗衣机如何洗是不需要知道的。但是上面 Person 类中的 washClothes 方法，却一连调用了 WashingMachine 类的三个方法，并且这三个方法都是洗衣机需要做的，跟 Person 类没有什么关系。这就造成了 Person 类对 WashingMachine 类知道的太多了。

调整后的代码是：

```
class WashingMachine{

public:

   void automatic();

private:

   void receiveClothes();

   void wash();

   void drying();

};

class Person{

public:

   void washClothes(WashingMachine& wm){
```

```
        wm.automatic();
    }
};
```

总结：

1. 朋友间也是有距离的

一个类公开的 public 属性或方法越多，修改时涉及的面也就越大，变更引起的风险扩散也就越大。因此，为了保持朋友类间的距离，在设计时需要反复衡量：是否还可以再减少 public 方法和属性，是否可以修改为 private 等。

注意：迪米特原则要求类“羞涩”一点，尽量不要对外公布太多的 public 方法和非静态的 public 变量，尽量内敛，多使用 private、protected 等访问权限。

2. 是自己的就是自己的，不要给别人

如果一个方法放在本类中，既不增加类间关系，也对本类不产生负面影响，就放置在本类中。如果放在其他类中，反而可能增加类间关系。

四、 单一职责原则（Single Responsibility Principle，SRP）

永远不要让一个类存在多个改变的理由。

换句话说，如果一个类需要改变，改变它的理由永远只有一个。如果存在多个改变它的理由，就需要重新设计该类。

单一职责原则的核心含意是：只能让一个类有且仅有一个职责。这也是单一职责原则的命名含义。

为什么一个类不能有多于一个以上的职责呢？

如果一个类具有一个以上的职责，那么就会有多个不同的原因引起该类变化，而这种变化将影响到该类不同职责的使用者（不同用户）：

1， 一方面，如果一个职责使用了外部类库，则使用另外一个职责的用户却也不得不包含这个未被使用的外部类库。
2， 另一方面，某个用户由于某个原因需要修改其中一个职责，另外一个职责的用户也将受到影响，他将不得不重新编译和配置。

这违反了设计的开闭原则，也不是我们所期望的。

职责的划分

既然一个类不能有多个职责，那么怎么划分职责呢？

Robert.C Martin 给出了一个著名的定义：所谓一个类的一个职责是指引起该类变化的一个原因。

如果你能想到一个类存在多个使其改变的原因，那么这个类就存在多个职责。

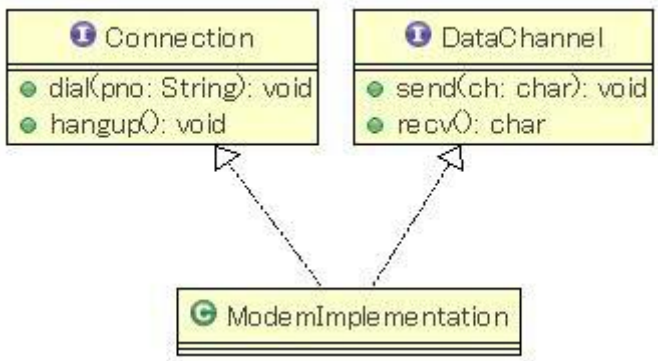
Single Responsibility Principle (SRP)的原文里举了一个 Modem 的例子来说明怎么样进行职责的划分，这里我们也沿用这个例子来说明一下：

SRP 违反例：
Modem.cpp

```
class Modem {
   public：void dial(String pno);   //拨号
           void hangup();       //挂断
           void send(char c);   //发送数据
           char recv();       //接收数据
};
```

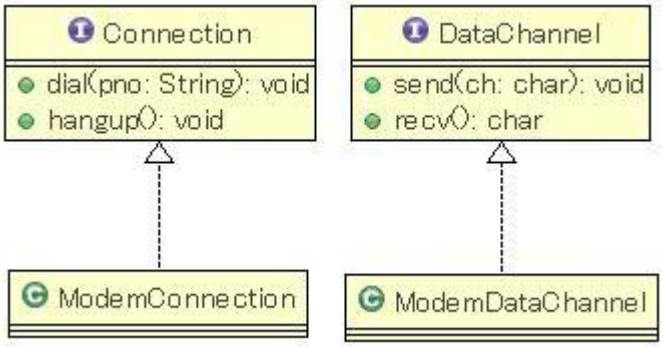
咋一看，这是一个没有任何问题的接口设计。但事实上，这个接口包含了 2 个职责：第一个是连接管理（dial，hangup）；另一个是数据通信（send，recv）。很多情况下，这 2 个职责没有任何共通的部分，它们因为不同的理由而改变，被不同部分的程序调用。所以它违反了 SRP 原则。

下面的类图将它的 2 个不同职责分成 2 个不同的接口，这样至少可以让客户端应用程序使用具有单一职责的接口：

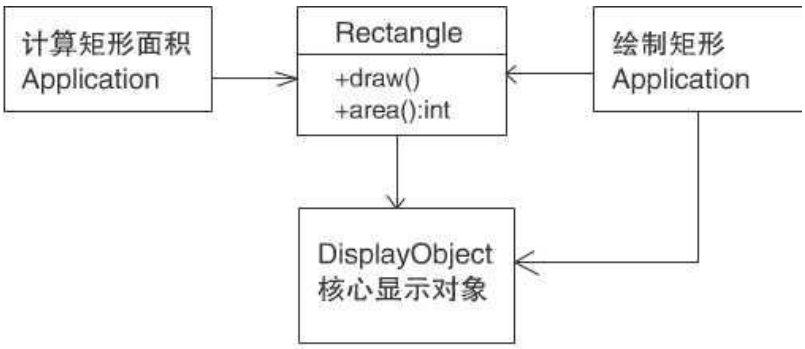


让 ModemImplementation 实现这两个接口。我们注意到，ModemImplementation 又组合了 2 个职责，这不是我们希望的，但有时这又是必须的。通常由于某些原因，迫使我们不得不绑定多个职责到一个类中，但我们至少可以通过接口的分割来分离应用程序关心的概念。

事实上，这个例子一个更好的设计应该是这样的，如图：



例如，考虑下图的设计。Retangle 类具有两个方法，如图。一个方法把矩形绘制在屏幕上，另一个方法计算矩形的面积。

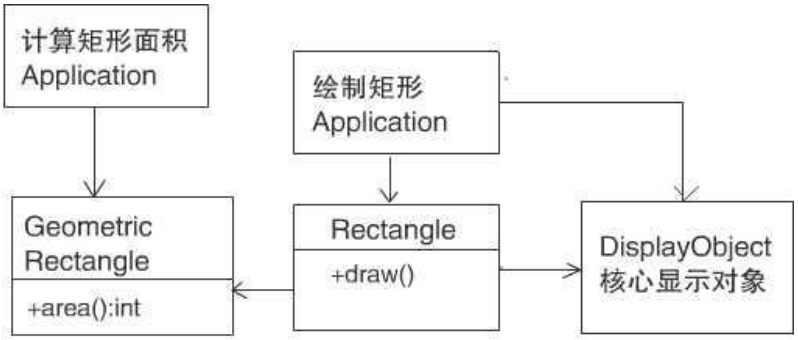


多于一个的职责

有两个不同的 Application 使用 Rectangle 类，如上图。一个是计算几何面积的，Rectangle 类会在几何形状计算方面给予它帮助。另一个 Application 实质上是绘制一个在舞台上显示的矩形。

这一设计违反了单一职责原则。Rectangle 类具有了两个职责，第一个职责是提供一个矩形形状几何数据模型；第二个职责是把矩形显示在屏幕上。

对于 SRP 的违反导致了一些严重的问题。首先，我们必须在计算几何应用程序中包含核心显示对象的模块。其次，如果绘制矩形 Application 发生改变，也可能导致计算矩形面积 Application 发生改变，导致不必要的重新编译，和不可预测的失败。



分离的职责

一个较好的设计是把这两个职责分离到下图所示的两个完全不同的类中。这个设计把 Rectangle 类中进行计算的部分一道 GeometryRectangle 类中。现在矩形绘制方式的改变不会对计算矩形面积的应用产生影响了。

单一职责原则从职责（改变理由）的侧面上为我们对类（接口）的抽象的颗粒度建立了判断基准：在为系统设计类（接口）的时候应该保证它们的单一职责性。

Robert C. Martin. The Single Responsibility Principle (SRP)

五、 接口分隔原则（Interface Segregation Principle，ISP）

不能强迫用户去依赖那些他们不使用的接口。换句话说，使用多个专门的接口比使用单一的总接口总要好。它包含了 2 层意思：

- 接口的设计原则：接口的设计应该遵循最小接口原则，不要把用户不使用的方法塞进同一个接口里。 如果一个接口的方法没有被使用到，则说明该接口过胖，应该将其分割成几个功能专一的接口。

- 接口的依赖（继承）原则：如果一个接口 a 继承另一个接口 b，则接口 a 相当于继承了接口 b 的方法，那么继承了接口 b 后的接口 a 也应该遵循上述原则：不应该包含用户不使用的方法。 反之，则说明接口 a 被 b 给污染了，应该重新设计它们的关系。

如果用户被迫依赖他们不使用的接口，当接口发生改变时，他们也不得不跟着改变。换而言之，一个用户依赖了未使用但被其他用户使用的接口，当其他用户修改该接口时，依赖该接口的所有用户都将受到影响。这显然违反了开闭原则，也不是我们所期望的。

下面我们举例说明怎么设计接口或类之间的关系，使其不违反 ISP 原则。

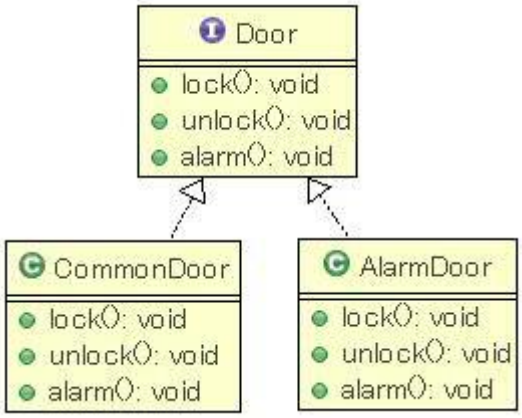
假如有一个 Door，有 lock，unlock 功能，另外，可以在 Door 上安装一个 Alarm 而使其具有报警功能。用户可以选择一般的 Door，也可以选择具有报警功能的

Door。有以下几种设计方法：

ISP 原则的违反例：

方法一：

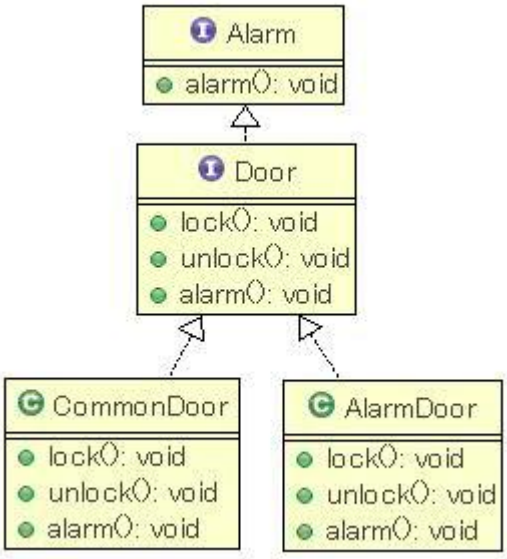
在 Door 接口里定义所有的方法。图：



但这样一来，依赖 Door 接口的 CommonDoor 却不得不实现未使用的 alarm()方法。违反了 ISP 原则。

方法二：

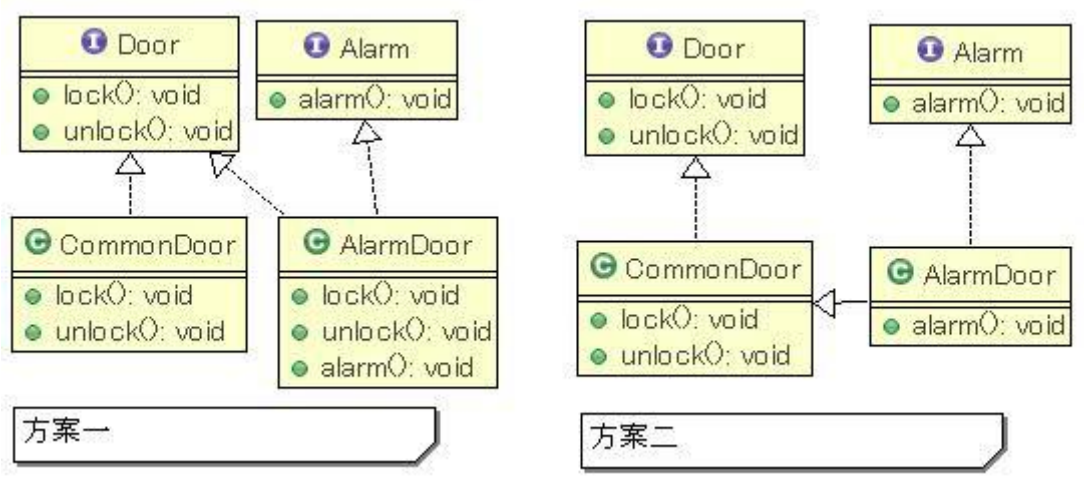
在 Alarm 接口定义 alarm 方法，在 Door 接口定义 lock，unlock 方法，Door 接口继承 Alarm 接口。



跟方法一一样，依赖 Door 接口的 CommonDoor 却不得不实现未使用的 alarm()方法。违反了 ISP 原则。

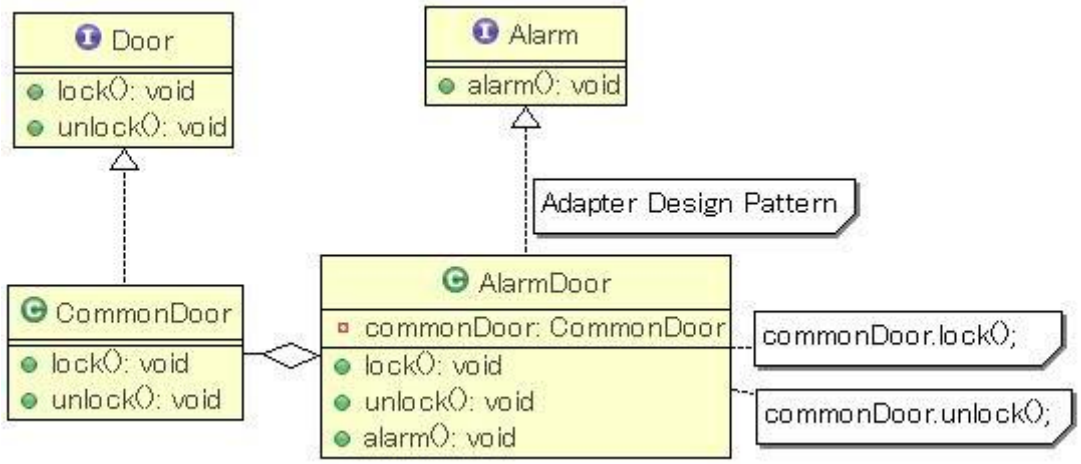
遵循 ISP 原则的例：

方法三：通过多重继承实现



在 Alarm 接口定义 alarm 方法，在 Door 接口定义 lock，unlock 方法。接口之间无继承关系。CommonDoor 实现 Door 接口，AlarmDoor 有 2 种实现方案：
1）同时实现 Door 和 Alarm 接口。
2）继承 CommonDoor，并实现 Alarm 接口。
第 2）种方案更具有实用性。
这种设计遵循了 ISP 设计原则。

方法四：通过关联实现



在这种方法里，AlarmDoor 实现了 Alarm 接口，同时把功能 lock 和 unlock 委让给 CommonDoor 对象完成。
这种设计遵循了 ISP 设计原则。

接口分隔原则从对接口的使用上为我们对接口抽象的颗粒度建立了判断基准：在为系统设计接口的时候，使用多个专门的接口代替单一的胖接口。

Robert C. Martin. The Interface Segregation Principle (ISP)

单一职责原则与接口隔离原则的共同点

1. 两者都是为了提高内聚性、降低耦合性，体现了封装的思想；
2. 最终表现出来的都是将接口约束到最小功能。

单一职责原则与接口隔离原则的区别

1. 针对内容不同：单一职责原则针对的是模块、类、接口的设计，接口隔离原则**更侧重于接口的设计**；
2. 思考角度不同：单一职责原则是从软件实体本身的职责/功能是否单一来考虑，接口隔离原则主要是从用户的角度来考虑接口约束问题，它同时也提供了一种判断接口职责是否单一的标准：通过调用者如何使用接口来间接地判定，如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。
3. 例如一个接口的职责可能包含多个方法，这多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，因为它要求“尽量使用多个专门的接口”。专门的接口就是指提供给每个模块的都应该是其需要调用的接口，而不是建立一个庞大的臃肿的接口，容纳所有的客户端访问。

六、 依赖倒置原则（Dependency Inversion Principle ， DIP）

A. 高层模块不应该依赖于低层模块，二者都应该依赖于抽象

B. 抽象不应该依赖于细节，细节应该依赖于抽象

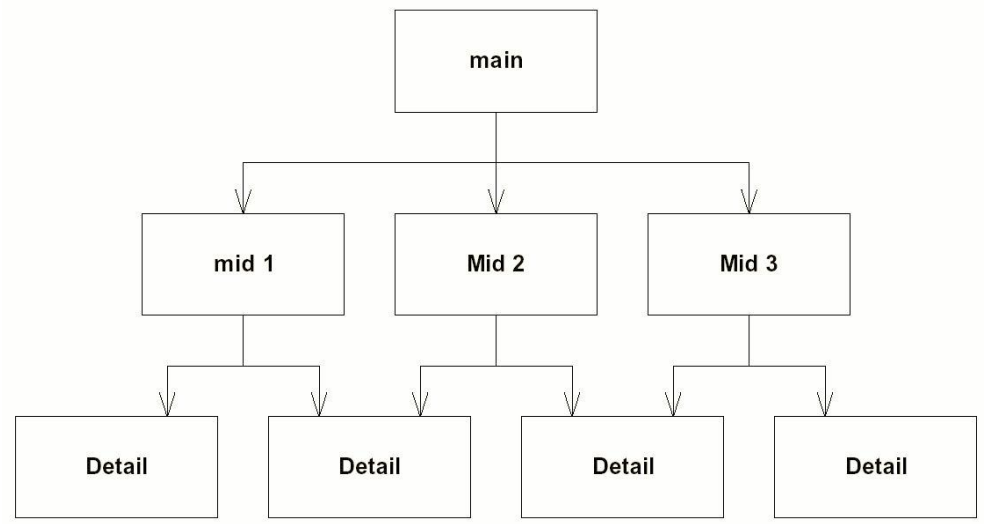
概念解说：

依赖：在程序设计中，如果一个模块 a 使用/调用了另一个模块 b，我们称模块 a 依赖模块 b。

高层模块与低层模块：往往在一个应用程序中，我们有一些低层次的类，这些类实现了一些基本的或初级的操作，我们称之为低层模块；另外有一些高层次的类，这些类封装了某些复杂的逻辑，并且依赖于低层次的类，这些类我们称之为高层模块。

为什么叫做依赖倒置（Dependency Inversion）呢？

面向对象程序设计相对于面向过程（结构化）程序设计而言，依赖关系被倒置了。因为传统的结构化程序设计中，高层模块总是依赖于低层模块。



问题的提出：

Robert C. Martin 在原文中给出了“Bad Design”的定义：

1. 系统很难改变，因为每个改变都会影响其他很多部分。

2. 当你对某地方做一修改，系统的看似无关的其他部分都不工作了。

3. 系统很难被另外一个应用重用，因为很难将要重用的部分从系统中分离开来。

导致“Bad Design”的很大原因是“高层模块”过分依赖“低层模块”。

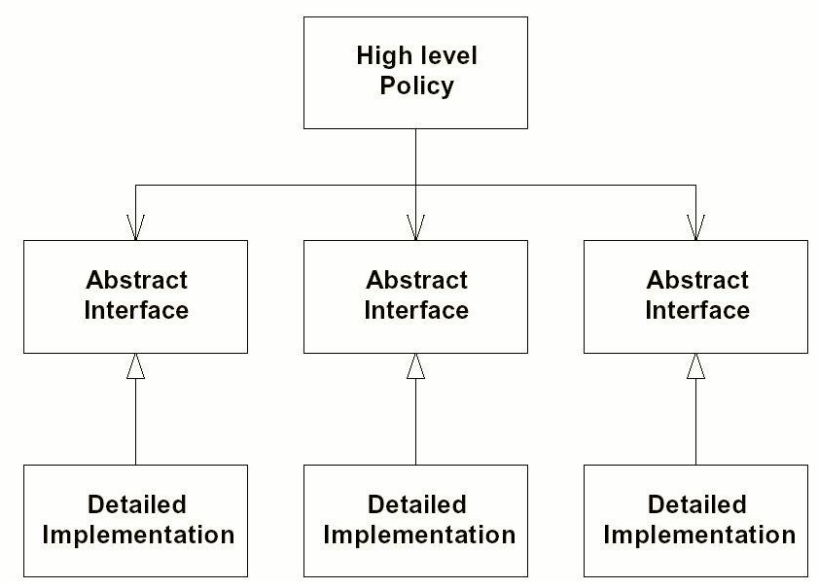
一个良好的设计应该是系统的每一部分都是可替换的。

如果“高层模块”过分依赖“低层模块”，一方面一旦“低层模块”需要替换或者修改，“高层模块”将受到影响；另一方面，高层模块很难可以重用。

问题的解决：

为了解决上述问题，Robert C. Martin 氏提出了 OO 设计的 Dependency Inversion Principle (DIP) 原则。

DIP 给出了一个解决方案：在高层模块与低层模块之间，引入一个抽象接口层。



抽象接口是对低层模块的抽象，低层模块继承或实现该抽象接口。

这样，高层模块不直接依赖低层模块，而是依赖抽象接口层。抽象接口也不依赖低层模块的实现细节，而是低层模块依赖（继承或实现）抽象接口。

类与类之间都通过抽象接口层来建立关系。

熔炉示例：考虑一个控制熔炉调节器的软件。该软件从一个 IO 通道中读取当前的温度，并通过向另一个 IO 通道发送命令来指示熔炉的开或者关。

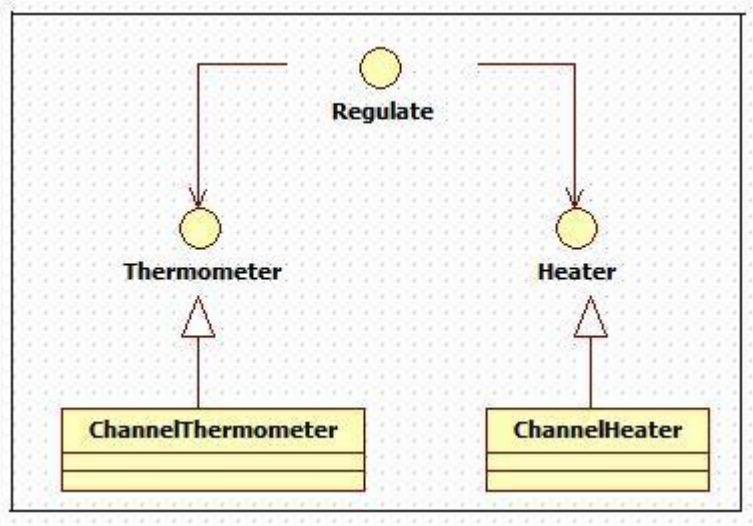
温度调节器的简单算法

```
const byte THERMONETER=0x86;
const byte FURNACE=0x87;
const byte ENGAGE=1;
const byte DISENGAGE=0;

void Regulate(double minTemp,double maxTemp)
{
  for(;;)
  {
    while (in(THERMONETER) > minTemp)
      wait(1);
    out(FURNACE,ENGAGE);

    while (in(THERMONETER) < maxTemp)
      wait(1);
    out(FURNACE,DISENGAGE);
```

```
  }
}
```

算法的高层意图是清楚的，但是实现代码中却夹杂着许多低层细节。这段代码根本不能重用于不同的控制硬件。

由于代码很少，所以这样做不会造成太大的损害。但是，即使是这样，使算法失去重用性也是可惜的。我们更愿意倒置这种依赖关系，

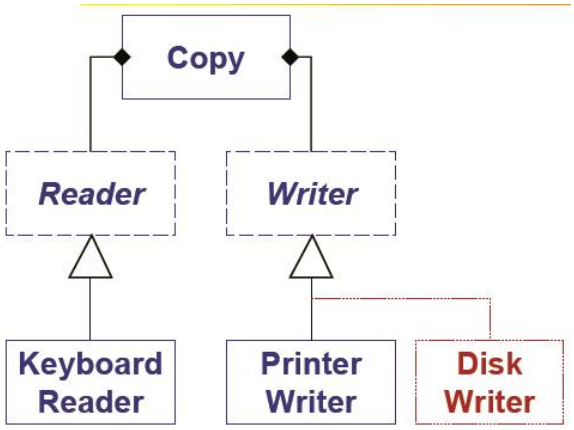


图中显示了 Regulate 函数接受了两个接口参数。Thermometer 接口可以读取，而 Heater 接口可以启动和停止。Regulate 算法需要的就是这些。这就倒置了依赖关系，使得高层的调节策略不再依赖于任何温度计或者熔炉的特定细节。该算法具有很好的可重用性。

通用的调节器

```
void Regulate(Thermometer t, Heater h, double minTemp,
  double maxTemp)
{
 for(;;)
 {
   while (t.Read() > minTemp)
     wait(1);
   h.Engate();

   while (t.Read() < maxTemp)
     wait(1);
   h.Disengage();
 }
}
```

例如一个 Copy 模块，需要把来自 Keyboard 的输入复制到 Print，即使对 Keyboard 和 Print 的封装已经做得非常好，但如果 Copy 模块里直接使用 Keyboard 与 Print，Copy 任很难被其他应用环境（比如需要输出到磁盘时）重用。




```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

启发 1：依赖于抽象

✓ 任何对象都不应该持有一个指向具体类的指针或引用

```
class class1{

    class2* cls2 = new class2();

}

class class2{
```

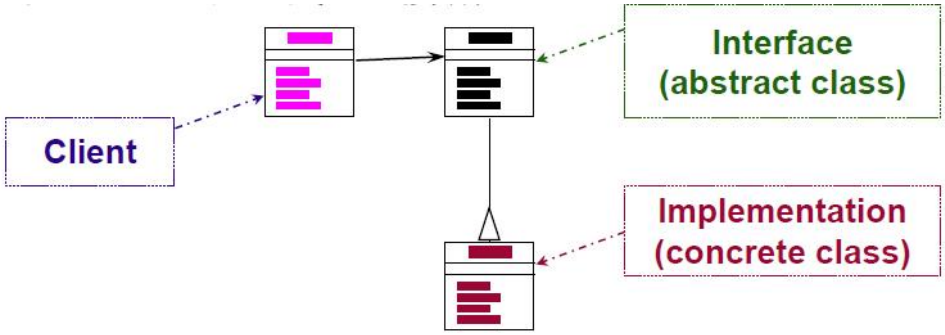
.....

}

✓ 任何类都不应该从具体类派生

启发 2：设计接口而非设计实现

✓ 使用继承/实现避免对具体类的直接绑定



•抽象类/接口：

倾向于较少的变化

抽象是关键点，它易于修改和扩展

不要强制修改那些抽象接口/类

•例外

有些类不可能变化

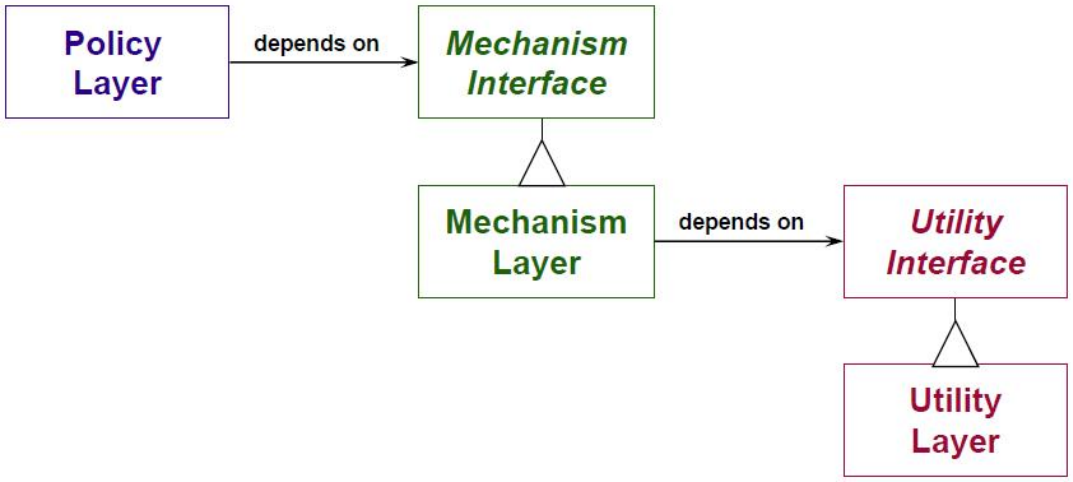
在可以直接使用具体类的情况下，不需要插入抽象层

如：字符串类

启发 3：避免传递依赖

✓ 避免高层依赖于低层

✓ 使用抽象类/接口和继承/实现来有效地消除传递依赖



Robert C. Martin. Dependency Inversion Principle (DIP)

七、 组合/聚合复用原则（Composite/Aggregate Reuse Principle ， CARP）

也称作合成复用原则（Composite Reuse Principle，CRP）。

尽量使用组合/聚合而非继承来达到复用目的；或者说，是在一个新的对象中使用一些已有的对象，使之成为新对象的一部分，然后新的对象通过向这些对象委托功能达到复用这些对象的目的。

组合和聚合都是关联的特殊种类。聚合表示整体和部分的关系，表示“拥有”。组合则是一种更强的“拥有”。组合的新的对象完全支配其组成部分，包括它们的创建和湮灭等。一个组合关系的成员对象是不能与另一个组合关系共享的。组合是值的聚合（Aggregation by Value），而一般说的聚合是引用的聚合（Aggregation by Reference）。

在面向对象设计中，有两种基本的办法可以实现复用：第一种是通过组合/聚合，第二种就是通过继承。

要正确的选择组合/聚合和继承，必须透彻的理解里氏替换原则和 Coad 法则。里氏替换原则前面学习过，Coad 法则由 Peter Coad 提出，总结了一些什么时候使用继承作为复用工具的条件。只有当以下的 Coad 条件全部被满足时，才应当使用继承关系：：

1）派生类是基类的一个特殊种类，而不是基类的一个角色，也就是区分"has a"和"is a"。只有"is a"关系才符合继承关系，"has a"关系应当用聚合来描述。

2）永远不会出现需要将派生类换成另外一个类的派生类的情况。如果不能肯定将来是否会变成另外一个派生类的话，就不要使用继承。

3）派生类具有扩展基类的责任，而不是具有置换掉（override）或注销掉（nullify）基类的责任。如果一个派生类需要大量的置换掉基类的行为，那么这个类就不应该是这个基类的派生类。

4）只有在分类学角度上有意义时，才可以使用继承。

如果语义上存在着明确的"is a"关系，并且这种关系是稳定的、不变的，则考虑使用继承；如果没有"is a"关系，或者这种关系是可变的，使用组合/聚合。

错误的使用继承而不是组合/聚合的一个常见原因是错误的把"has a"当成了"is a" 。"is a"代表一个类是另外一个类的一种；"has a"代表一个类是另外一个类的一个角色，而不是另外一个类的特殊种类。

示例：如果把“人”当成一个类，然后把“雇员”，“经理”，“学生”当成是“人”的派生类。这个的错误在于把 “角色” 的等级结构和 “人” 的等级结构混淆了。“经理”，“雇员”，“学生”是一个人的角色，一个人可以同时拥有上述角色。如果按继承来设计，那么如果一个人是雇员的话，就不可能是学生，这显然不合理。正确的设计是有个抽象类 “角色”，“人”可以拥有多个“角色”（聚合），“雇员”，“经理”，“学生”是“角色”的派生类。

另外一个就是只有两个类满足里氏替换原则的时候，才可能是"is a" 关系。也就是说，如果两个类是"has a"关系，但是设计成了继承，那么肯定违反里氏替换原则。

通过组合/聚合复用的优缺点

优点：

1) 新对象存取成员对象的唯一方法是通过成员对象的接口；

2) 这种复用是黑箱复用，因为成员对象的内部细节是新对象所看不见的；

3) 这种复用更好地支持封装性；

4) 这种复用实现上的相互依赖性比较小；

5) 每一个新的类可以将焦点集中在一个任务上；

6) 这种复用可以在运行时间内动态进行，新对象可以动态的引用与子对象类型相同的对象。

7) 作为复用手段可以应用到几乎任何环境中去。

缺点: 就是系统中会有较多的对象需要管理。

通过继承来进行复用的优缺点

优点：

新的实现较为容易，因为基类的大部分功能可以通过继承的关系自动进入派生类。

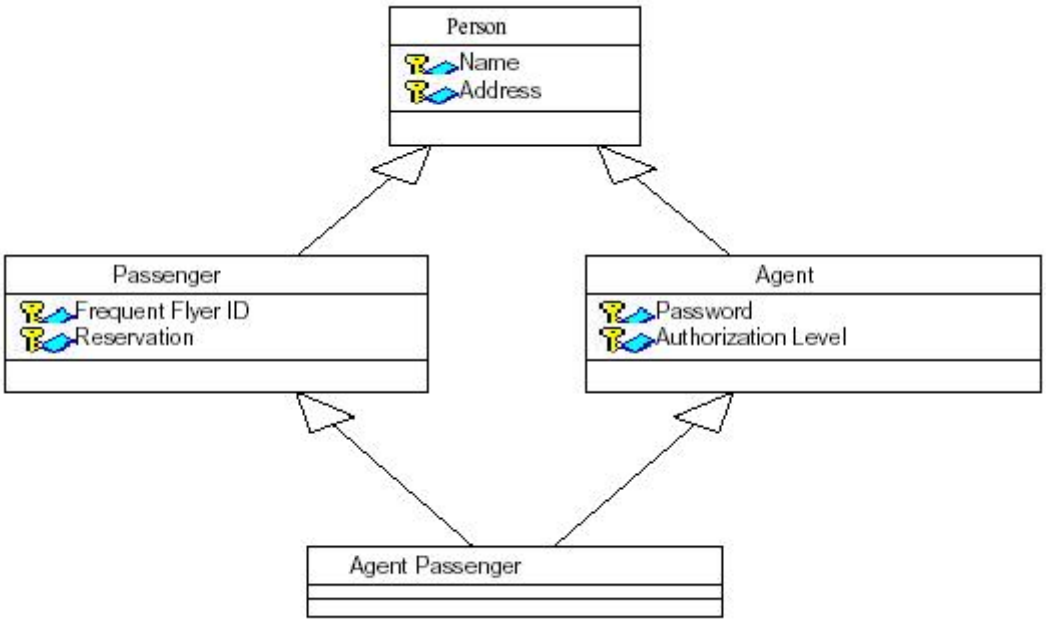
修改和扩展继承而来的实现较为容易。

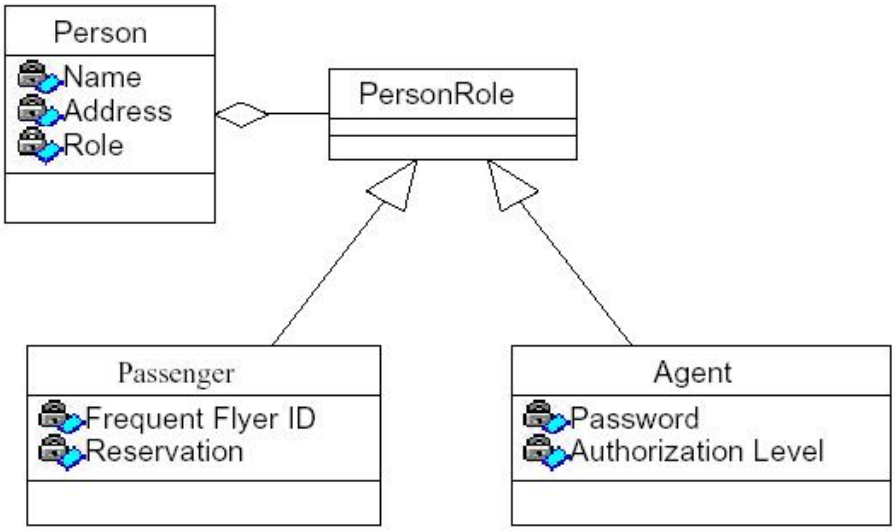
缺点：

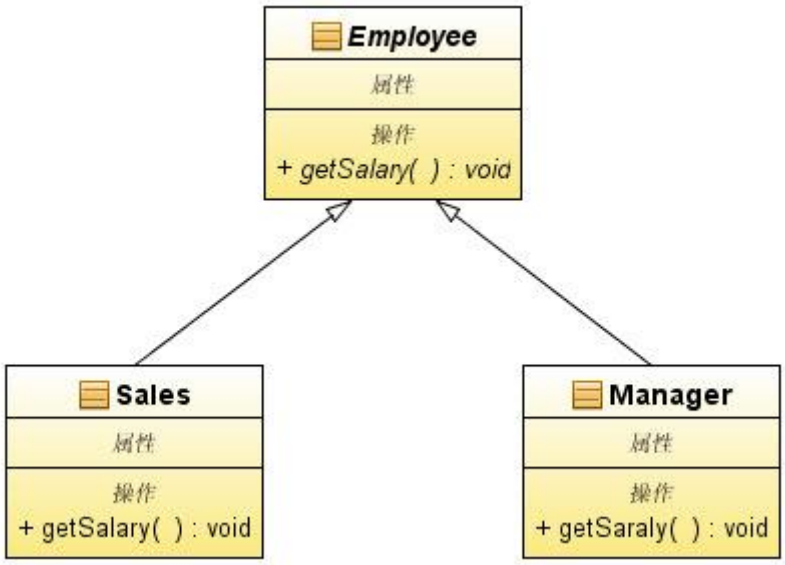
继承复用破坏封装性，因为继承将基类的实现细节暴露给派生类。由于基类的内部细节常常是对于派生类透明的，所以这种复用是透明的复用，又称“白箱”复用。

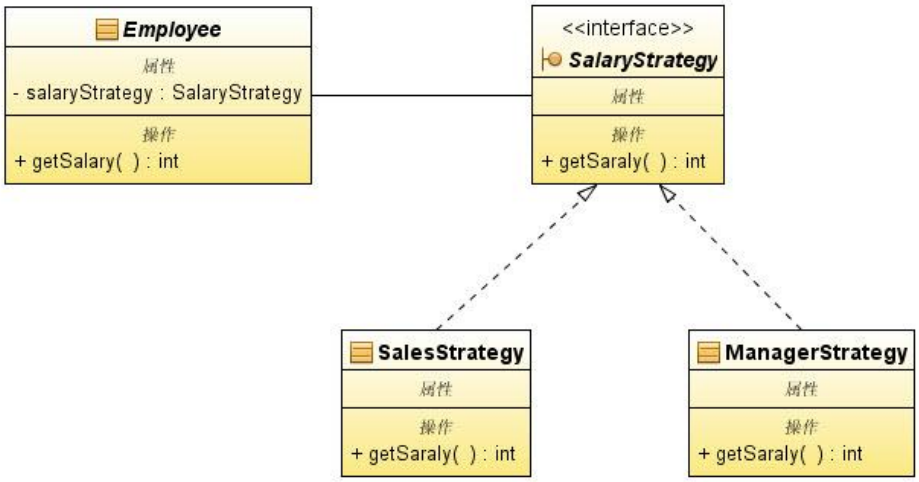
如果基类发生改变，那么派生类的实现也不得不发生改变。

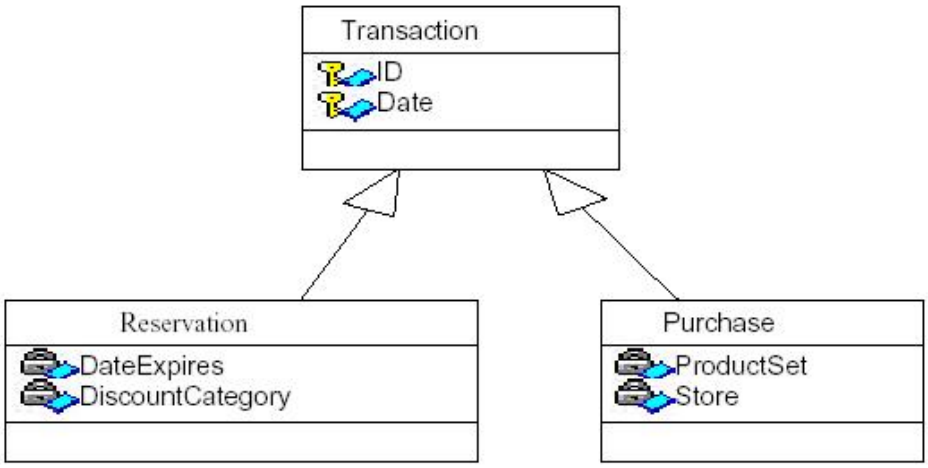
从基类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。




Person
Name
Address
Role
PersonRole
Passenger
Frequent Flyer ID
Reservation
Agent
Password
Authorization Level


Employee
属性
操作
+ getSalary( ) : void
Sales
属性
操作
+ getSalary( ) : void
Manager
属性
操作
+ getSaraly( ) : void


Employee
属性
- salaryStrategy : SalaryStrategy
操作
+ getSalary( ) : int
<<interface>>
SalaryStrategy
属性
操作
+ getSaraly( ) : int
SalesStrategy
属性
操作
+ getSaraly( ) : int
ManagerStrategy
属性
操作
+ getSaraly( ) : int



GOF，Design Pattern.