2023年春季学期

软件设计实践

软件设计原则

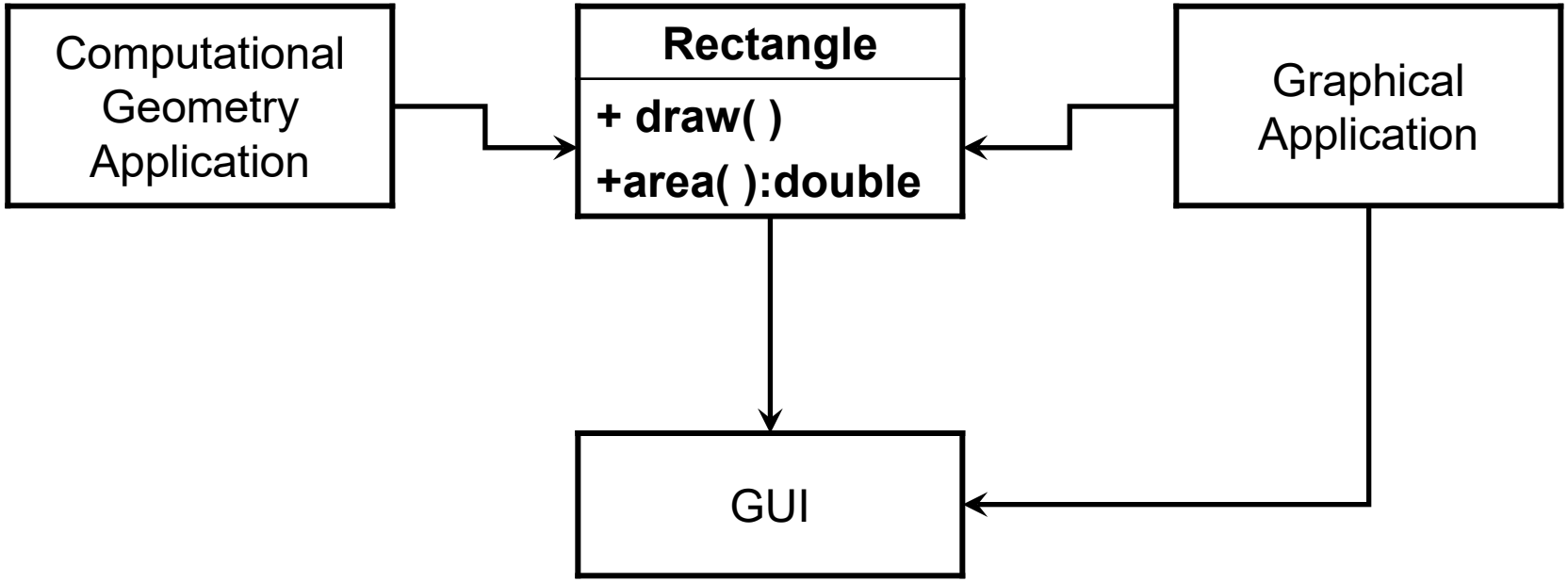
谢 涛　　马 郓

基于王亚沙老师的胶片

1. 单一职责原则（SRP）

- 陈述：
  - 就一个类而言，应该只有一个导致其变化的原因
- 分析：
  - 一个职责就是一个变化的轴线
  - 一个类如果承担的职责过多，就等于将这些职责耦合在一起。一个职责的变化可能会虚弱或者抑止这个类完成其它职责的能力
  - 多职责将导致脆弱性的臭味

- 示例1:

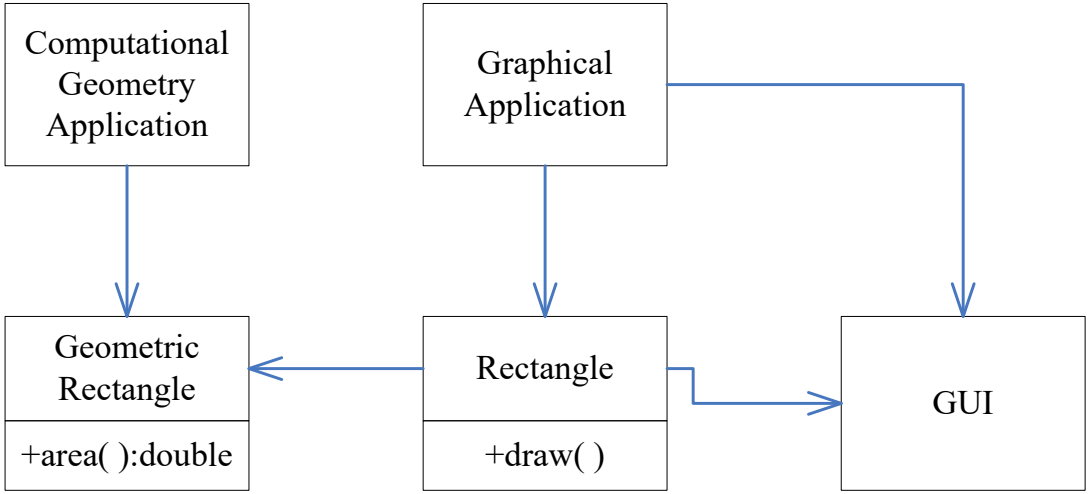


Rectangle类具有两个职责:

1. 计算矩形面积的数学模型
2. 将矩形在一个图形设备上描述出来

- Rectangle类违反了SRP，具有两个职能——计算面积和绘制矩形
- 这种对SRP的违反将导致两个方面的问题：
  - 包含不必要的代码
    - 一个应用可能希望使用Retangle类计算矩形的面积，但是却被迫将绘制矩形相关的代码也包含进来
  - 一些逻辑上毫无关联的原因可能导致应用失败
    - 如果GraphicalApplication的需求发生了变化，从而对Rectangle类进行了修改。但是这样的变化居然会要求我们重新构建、测试以及部署ComputationalGeometryApplication，否则其将莫名其妙的失败。

修改后的设计如下：


Computational Geometry Application
Graphical Application
Geometric Rectangle
+area( ):double
Rectangle
+draw( )
GUI

2. 开放封闭原则（OCP）

- 陈述：
  - 软件实体（类、模块、函数等）应该是可以扩展的，同时还可以是不必修改的，更确切的说，函数实体应该：

    （1）对扩展是开放的

    当应用的需求变化时，我们可以对模块进行扩展，使其具有满足改变的新的行为——即，我们可以改变模块的功能

    （2）对更改是封闭的

    对模块进行扩展时，不必改动已有的源代码或二进制代码。
- 分析：
  - 世界是变化的（而且变化很快），软件是对现实的抽象

    →软件必须能够扩展
  - 如果任何修改都需要改变已经存在的代码，那么可能导致牵一发动全身现象，进而导致雪崩效应，使软件质量显著下降

- 例子

C语言程序

```
---------shape.h-----------------
emum ShapeType{circle,square};
struct Shape{
        ShapeType itsType;
};
---------circle.h-----------------
struct Circle{
        ShapeType itsType;
        double itsRadius;
        CPoint itscenter;
};
---------square.h-----------------
struct Square{
        ShapeType itsType;
        double itsSide;
        CPoint itsTopLeft;
};
```

```
---------drawAllShapes.cpp----------
typedef struct Shape * ShapePointer;

void DrawAllShapes(ShapePointer list[], int n){
        int i;
        for(i=0;i<n;i++){
                struct Shape* s=list[i];
                switch (s->itsType){
                        case square:
                                s->Square();
                        break;
                        case circle:
                                s->DrawCircle();
                        break;
                }
}}
```

- 例子
  - 批判

    这个程序不符合OCP，如果需要处理的几何图形中再加入“三角形”将引发大量的修改

    - 僵化的

      增加Triangle会导致Shape、Square、Circle以及DrawAllShapes的重新编译和部署

    - 脆弱的

      因为存在大量的既难以查找又难以理解的Switch和If语句，修改稍有不慎，程序就会莫明其妙的出错

    - 牢固的

      想在一个程序中复用DrawAllShapes，都必须带上Circle、Square，即使那个程序不需要他们

例子

- 修改后的设计

```
class Shape{
public:
        virtual void Draw() const=0;
};

class Square:public Shape{
public:
        virtual void Draw() const;
};

class Circle:public Shape{
public:
        virtual void Draw() const;
};
```

```
void DrawAllShapes(Vector<Shape*>& list){
  vector<Shape*>::iterator i;
  for(i=list.begin();i!=list.end();i++)
        (*i)->Draw();
}
```

- 例子
  - 再看这些批判

    再加入“三角形”将变得十分简单:

    - ~~僵化的~~

      增加Triangle会导致Shape、Square、Circle以及DrawAllShapes的重新编译和部署

    - ~~脆弱的~~

      因为存在大量的既难以查找又难以理解的Switch和If语句，修改稍有不慎，程序就会莫明其妙的出错

    - ~~笨重的~~

      想在一个程序中复用DrawAllShapes，都必须带上Circle、Square，即使那个程序不需要他们

- 一般而言，无论模块多么“封闭”，都会存在一些无法对之封闭的变化

  没有对所有变化的情况都封闭的模型

- 我们怎么办?
  - 既然不可能完全封闭，我们必须有策略的对待此问题——对模型应该封闭那类变化作出选择，封闭最可能出现的变化

    → 这需要对领域的了解，丰富的经验和常识

    →错误的判断反而不美，因为OCP需要额外的开销（增加复杂度）

    → 敏捷的思想——我们预测他们，但是直到我们发现他们才行动

3. LisKov替换原则（LSP）

- 陈述：
  - 子类型（Subtype）必须能够替换他们的基类型(Basetype)

    Barbara Liskov对原则的陈述：

    若对每个类型S的对象o_1，都存在一个类型T的对象o_2，使得在所有针对T编写的程序P中，用o_1替换o_2后，程序P的行为功能不变，则S是T的子类型。
- 分析：
  - 违反这个职责将导致程序的脆弱性和对OCP的违反

    例如：基类Base，派生类Derived，派生类实例d，函数f(Base* p)；
    - f(&d) 会导致错误： 显然D对于f是脆弱的。
    - 如果我们试图编写一些if check，以保证把d传给f时可以使f具有正确的行为。那么这个if check违反了OCP——因为f无法对Base的所有派生类都是封闭的

示例1：

```
struct Point{double x,y;};

struct shape{
        enum ShapeType{square,circle}
        itsType;
        shape(ShapeType t):itsType(t){ }
};

struct Circle:public Shape{
        Circle():Shape(circle){ };
        void Draw()const;
        Point itsCenter;
        double itsRadius;
};

struct Square:public Shape{
        Square():Shape(square){ };
        void Draw()const;
        Point itsTopLeft;
        double itsSide;
};
```

```
void DrawShape(const Shape& s){
        if(s.itsType==Shape::square)
                static_cast<const Square&>(s).Draw();
        else if (s.itsType==Shape::circle)
                static_cast<const Circle&>(s).Draw();
}
```

显然，DrawShape违反了OCP，

为什么？

因为Circle和Square违反了LSP

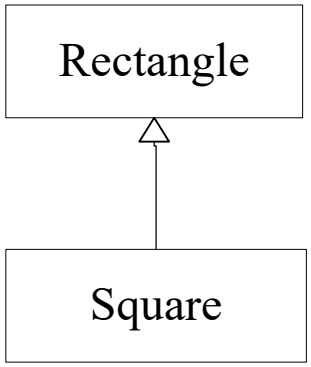
示例2：一次更加奇妙的违规

```
class Rectangle{
        Point topLeft;
        doulbe width;
        double height;
public:
        void setWidth(double w){width=w;}
        void setHeight(double h){height=h;}
        double getWidth() const{return width;}
        double getHeight() const{return
        height;}
};

class Square:public Rectangle{
public:
        void setWidth(double w);
        void setHeight(double h);
};
```

```
void Square::setWidth(double w){
        Rectangle::setWidth(w);
        Rectangle::setHeight(w);
};

void Square::setHeight(double h){
        Rectangle::setWidth(h);
        Rectangle::setHeight(h);
};
```



- 问题的第一步分析：
  - 看下面这个函数

```
void f(Rectangle& r){
    r.SetWidth(32);
}
```

问题：
显然，当我们将一个Square的实例传给f时，将可能导致其height与width不等，破坏了其完整性
——违反了LSP

要改正上述问题，很简单，我们只要将SetWidth和SetHeight两个函数设置成virtual函数即可

——添加派生类需要修改基类，通常意味着设计上的缺陷

但是并非所有人都同意上述的分析

反方：真正的设计缺陷是忘记把SetWidth和SetHeight两个函数作为virtual函数

正方：设置长方形的长宽是非常基本的操作，不是预见到有正方形这样的派生类，怎么会想到要将其设成虚函数?

- 放下这个争论，我们先将SetWidth和SetHeight改作虚函数看看

```
class Rectangle{
        Point topLeft;
        doulbe width;
        double height;
public:
        virtual  void setWidth(double w){width=w;}
        virtual void setHeight(double h){height=h;}
        double getWidth() const{return width;}
        double getHeight() const{return height;}
};

class Square:public Rectangle{
public:
        void setWidth(double w);
        void setHeight(double h);
};
```

```
void Square::setWidth(double w){
        Rectangle::setWidth(w);
        Rectangle::setHeight(w);
};

void Square::setHeight(double h){
        Rectangle::setWidth(h);
        Rectangle::setHeight(h);
};
```

看起来，
很不错！

- 真正的问题：

```
void g(Rectangle& r){
        r.setWidth(5);
        r.setHeight(4);
        assert(r.Area()==20);
}
```

函数g不能操作Square的实例，Square不能替换Rectangle，所以违反了LSP

LSP告诉我们：

孤立的看，我们无法判断模型的有效性

——考虑一个设计是否恰当时，不能孤立的看待并判断，应该从此设计的使用者所作出的假设来审视它！

事先的推测是困难的，我们采用敏捷的思想

→推迟这个判断——“一个模型是否违反LSP”。直到出现问题的时候我们才解决它。

- 更加深入的思索:

这个看似明显正确的模型怎么会出错呢?

“正方形是一种长方形”——地球人都知道

错在哪里?

对不是g函数的编写者而言，正方形可以是长方形，但是对g函数的编写者而言，Square绝对不是Rectangle！！

OOD中对象之间是否存在IS-A关系，应该从行为的角度来看待。

→而行为可以依赖客户程序做出合理的假设。

- 基于契约（和约）的设计——DBC（Deign by Contract）
  - “合理的假设”使人郁闷。

    ——我怎么知道是否合理呢？？

  - 使用DBC，类的编写者需要显示的规定针对该类的契约。客户代码的编写者可以通过契约获悉行为的依赖方式。
  - 契约通过为每一个方法规定前置条件（preconditions）和后置条件（postconditions）来指定的。要使一个方法执行，前置条件一定要为真（对客户的要求）；函数执行后要保证后置条件为真（对函数编写者的要求）。

基于契约（和约）的设计——DBC（Deign by Contract）（续）

- 例如：

  在上面的例子中，Rectangle::SetWidth(double w)的后置条件可以看作是：

  ```
  assert((itsWidth==w) && (itsHeight==old.itsHeight));
  ```

- 基类和派生类在前置条件和后置条件上的关系是：

  如果在派生类中重新申明了基类中已有的成员函数，这个函数只能使用相等或更弱的前置条件来替换原有的前置条件；并且，只能使用相等或更强的后置条件来替换原有的后置条件。

  →派生类必须接受基类已经接受的一切；并且，派生类不能违反基类已经确定的规则。

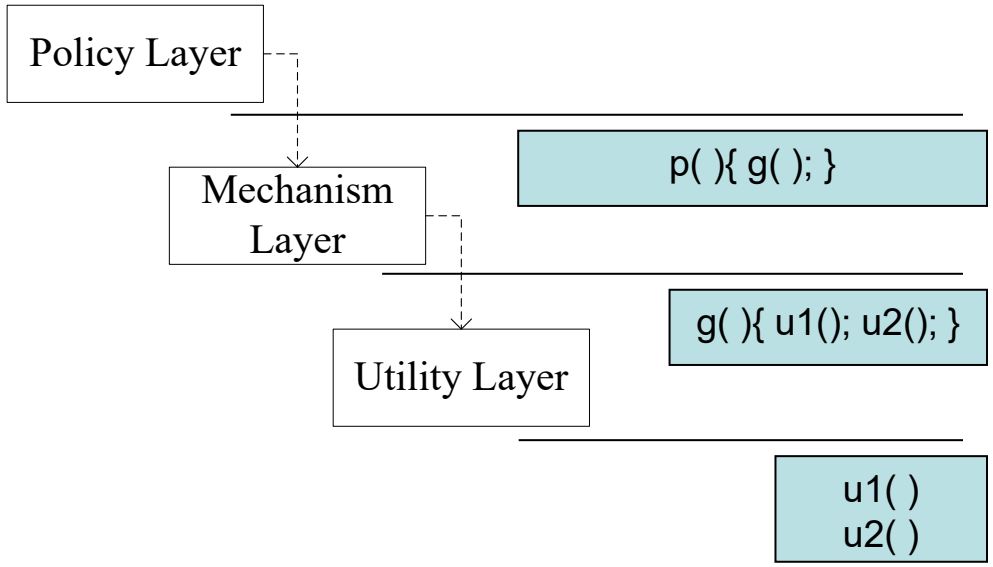
- 在一些语言中明确的支持契约，例如Eiffel，你申明它们，运行时系统会自动的检查。在Jave和C＋＋标准中尚未支持，我们必须自己考虑。

4. 依赖倒置原则（DIP）

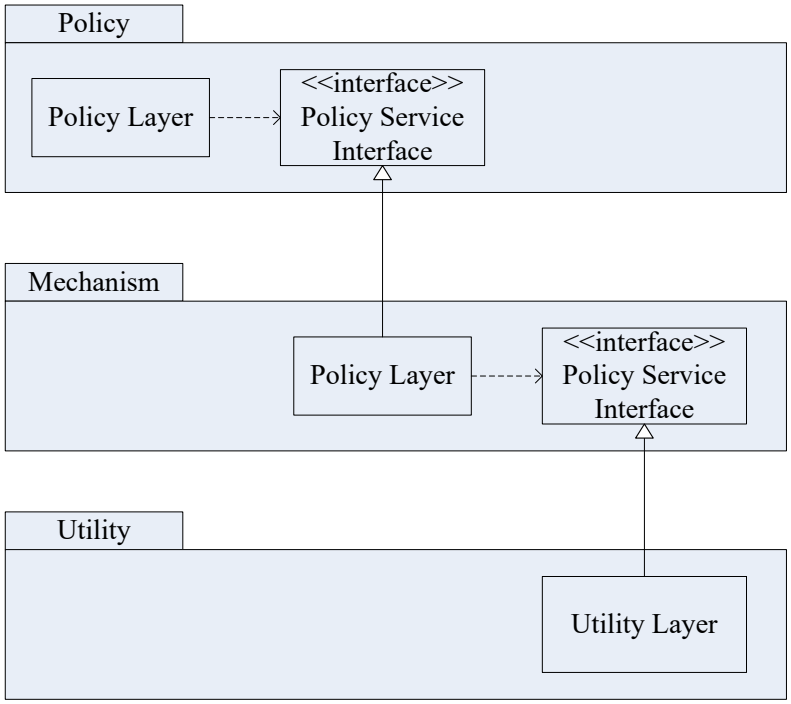
- 陈述：
  - 高层模块不应该依赖于低层模块。二者应该依赖于抽象。
  - 抽象不应该依赖于细节。细节应该依赖于抽象。
- 分析：
  - 所谓“倒置”是相对于传统的开发方法（例如结构化方法）中总是倾向于让高层模块依赖于低层模块而言的软件结构而言的。
  - 高层包含应用程序的策略和业务模型，而低层包含更多的实现细节，平台相关细节等。高层依赖低层将导致：
    - 难以复用。通常改变一个软硬件平台将导致一些具体的实现发生变化，如果高层依赖低层，这种变化将导致逐层的更改。
    - 难以维护。低层通常是易变的。

- 层次化：
  - “……所有良构的00体系结构都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务。”

    ——Booch
  - 对上述论述可能存在两种不同的理解：
    - 简单的理解



- 层次化(续)：
  - 更好的理解



- 依赖关系倒置

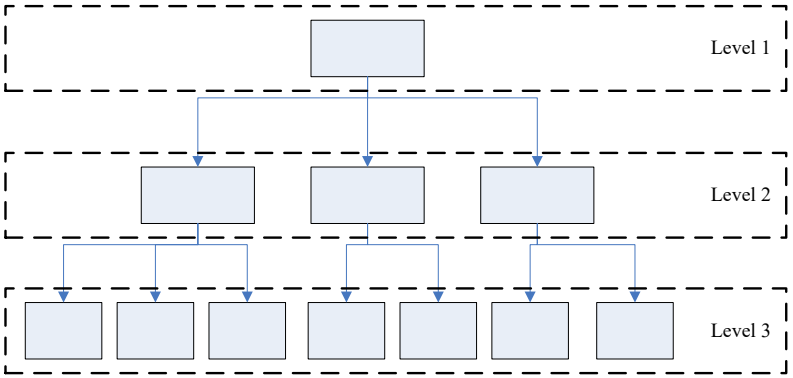
  下层的实现，依赖于上层的接口

- 接口所有权倒置

  客户拥有接口，而服务者则从这些接口派生

- 依赖不倒置的开发
  - 自顶向下首先设计整个软件的分解结构
  - 然后首先实现下层的功能
  - 再实现上层的功能，并使上层调用下层函数
- 依赖倒置的开发
  - 首先设计上层需要调用的接口，并实现上层
  - 然后低层类从上层接口派生，实现低层

  →接口属于上层



- 示例1（Button与Lamp）：
  - Button（开关）感知外界的变化。

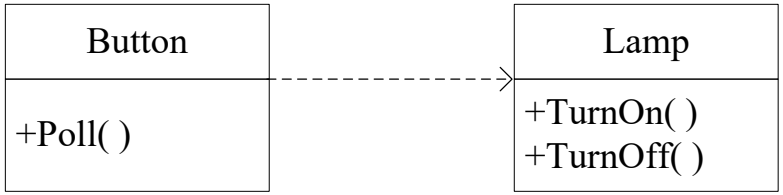
    当接受到Poll（轮询）消息时，判断其是否被“按下”。这个按下是抽象的（不关心通过什么样的机制去感知）：
    - 可能是GUI上的一个按钮被鼠标单击
    - 可能是一个真正的按钮被手指按下
    - 可能是一个防盗装置检测到了运动
    - ……
  - Lamp（灯）根据指示，收到turn on消息显示某种灯光，收到turn off消息关闭灯光
    - 可能是计算机控制台的LED
    - 可能是停车场的日光灯
    - 可能是激光打印机中的激光
    - ……

应该如何设计程序来用Button控制Lamp呢?

一个不成熟的设计

Button对象直接依赖Lamp对象，从而：

- Lamp的任何变化都会影响到Button，导致其改写或者重新编译
- 黑盒方式重用Button来控制一个Motor类变得不可能




```
class Button{
        Lamp * itsLamp;
public:
        void poll( ){
                if(/* some condition */)
                        itsLamp->turnOn( );
                ....
        }
};
```

一个依赖倒置的设计

依赖于抽象

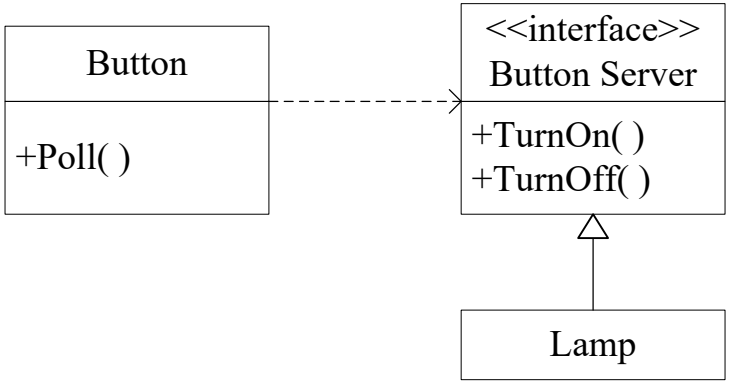
- 什么是高层策略？就是应用背后的抽象

  背后的抽象是检测用户的开/关指令

  - 用什么机制检测用户的指令？无关紧要
  - 目标对象是什么？无关紧要

  他们不会影响到抽象的具体细节

- 改进后的设计：



- Button依赖于抽象的接口ButtonServer（向该接口发消息）。ButtonServer提供一些抽象的方法，Button类通过这些接口可以开启或关掉一些东西。
- Lamp也依赖于ButtonServer接口（从此接口派生），提供具体的实现。

部分代码：

```
//Button.h
#include "ButtonServer.h"

class Button{
        ButtonServer * bs;
public:
        void poll( );
};

//Button.cpp
void Button::poll( ){
        if(/* mechanism for detecting
        turnOn command */)
                        bs->turnOn( );
        else if((/* mechanism for detecting
        turnOff command */)
                        bs->turnOff( );
}
```

```
//ButtonServer.h
class ButtonServer{
public:
        virtual void turnOn()=0;
        virtual void turnOff()=0;
};

//lamp.h
class Lamp:public ButtonServer{
public:
        void turnOn();
        void turnOff();
};

//lamp.cpp
void Lamp::turnOn(){
        /*codes for turn on a specific device*/
}

void Lamp::turnOff(){
        /*codes for turn off a specific device*/
}
```

分析：

- 上述设计使得Button可以控制所有愿意实现ButtonServer接口的设备，甚至是一个尚未开发出来的设备。

质疑：

- 这样的设计是不是强加了这样一个约束——所有需要被Button控制的对象一定要实现ButtonServer类。
- 如果我的设备还希望能够被另一个对象控制，比如Switch控制，怎么办?
- 这种设计是不是将Button对Lamp的依赖转嫁成了Lamp对Button的依赖呢？（毕竟Lamp只能被一种Button控制也是不好的）

抗辩：

- 上述质疑不成立。Button依赖于ButtonServer接口，但是接口并不依赖于Button，也就是说任何知道如何操作ButtonServer接口的对象都可以操作Lamp。
- 也许需要改进的仅仅是ButtonServer这样一个有些“误导性”的名字，我们可以将这个名字该得更加抽象一些，例如：SwitchableDevice

5. 接口隔离原则（ISP）

- 陈述：
  - 不应该强迫客户依赖于他们不用的方法
  - 一个类的不内聚的“胖接口”应该被分解成多组方法，每一组方法都服务于一组不同的客户程序。

先说一个例子：

```
class Door{
  public:
      virtual void Lock( );
      virtual void Unlock( );
      virtual bool IsDoorOpen( );
};
```

- Door可以加锁、解锁、而且可以感知自己是开还是关；
- Door是抽象基类，客户程序可以依赖于抽象而不是具体的实现

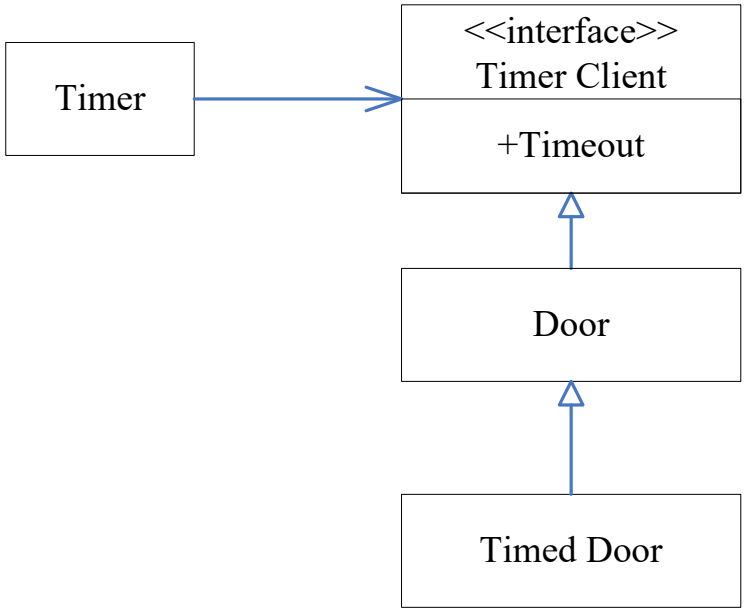
增加功能

如果门打开时间过长，它就会报警。
（比如宾馆客房的门）

一种常见的解决方案如下：

•问题——接口污染
在Door接口中加入新的方法（Timeout），而这个方法仅仅只为它的一个子类带来好处。

——如果每次子类需要一个新方法时它都被加到基类接口中，基类接口将很快变胖。



胖接口将导致SRP，LSP被违反，从而导致脆弱、僵化

- 客户的反作用力：
  - 通常接口的变化将导致client的改变
  - 但是很多时候，接口之所以变化是因为客户需要他们变化
  - → Client对interface具有反作用力！

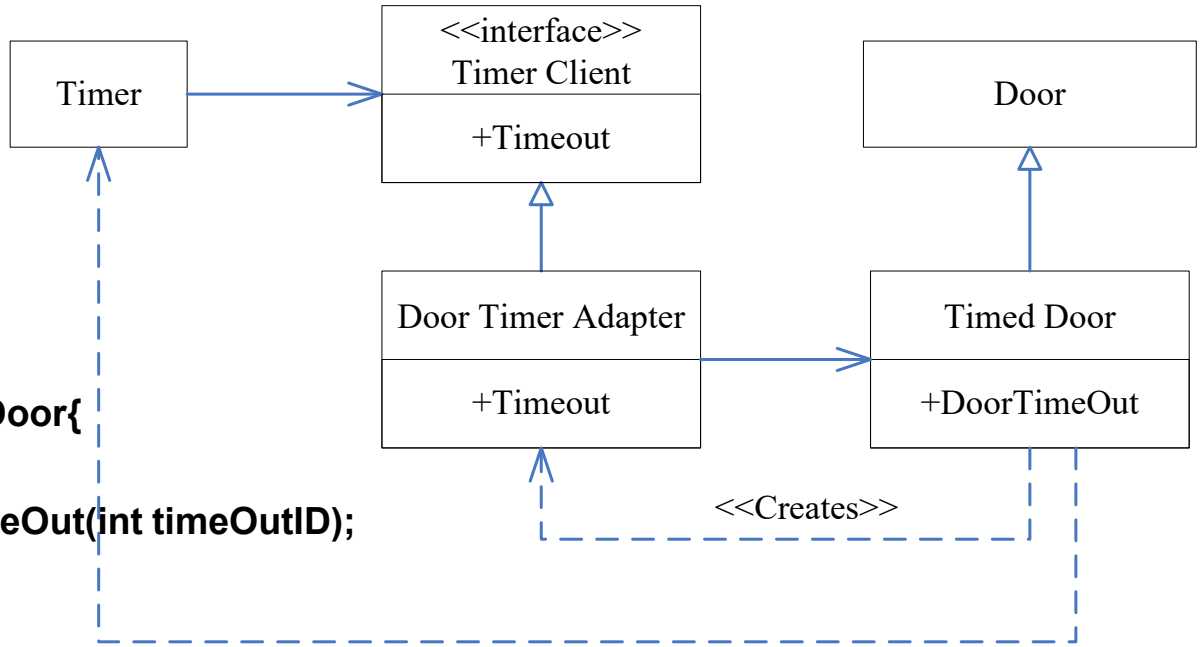
TimedDoor的多个超时请求问题，导致Timer接口做出下面的调整：

```
class Timer{
public:
        void Register(int timeout, int timeOutId, TimerClient* client);
};

class TimerClient{
public:
        virtual void TimerOut(int timeOutId );
};
```

TimedDoor对Timer接口的影响会传递到Door接口，从而导致所有Door都受到此影响，而且这一影响还会影响到Door的所有Clients——牵一发动全身

- 解决之道：
  - 使用委托




```
Class TimedDoor:public Door{
public:
        virtual void DoorTimeOut(int timeOutID);
};

class DoorTimeAdapter:public TimerClient{
        TimedDoor& itsTimedDoor;
public:
        DoorTimerAdapter(TimedDoor& theDoor):itsTimedDoor(theDoor){}
        vitual void TimeOut(int timeOutId){
                itsTimedDoor.DoorTimeOut(timeOutId);
        }
};
```