
南京大学

# 设计原则

# Design Principle

# 面向对象设计原则概述

- 可维护性(Maintainability)：指软件能够被理解、改正、适应及扩展的难易程度
- 可复用性(Reusability)：指软件能够被重复使用的难易程度
- 面向对象设计的目标之一在于支持可维护性复用，一方面需要实现设计方案或者源代码的复用，另一方面要确保系统能够易于扩展和修改，具有良好的可维护性

# 面向对象设计原则概述

- 面向对象设计原则为支持可维护性复用而诞生
- 指导性原则，非强制性原则
- 每一个设计模式都符合一个或多个面向对象设计原则，面向对象设计原则是用于评价一个设计模式的使用效果的重要指标之一

# 面向对象设计原则概述



- SRP (The Single-Responsibility Principle) 单一职责原则
- OCP (The Open-Closed Principle) 开放-封闭原则
- LSP (The Liskov Substitution Principle) Liskov替换原则
- DIP (The Dependency-Inversion Principle) 依赖倒置原则
- ISP (The Interface-Segregation Principle) 接口隔离原则
- CARP (Composition/Aggregation Reuse Principle ) 合成/聚合复用原则
- LoD (Law of Demeter) 迪米特法则

# 面向对象设计原则概述

| 设计原则名称 | 定 义 | 使用频率 |
| --- | --- | --- |
| 单一职责原则 (Single Responsibility Principle, SRP) | 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中 | ★★★★☆ |
| 开放-封闭原则 (Open-Closed Principle, OCP) | 软件实体应当对扩展开放，对修改关闭 | ★★★★★ |
| 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子类的对象 | ★★★★★ |
| 依赖倒置原则 (Dependence Inversion Principle, DIP) | 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象 | ★★★★★ |
| 接口隔离原则 (Interface Segregation Principle, ISP) | 客户端不应该依赖那些它不需要的接口 | ★★☆☆☆ |
| 合成/聚合复用原则 (Composition/Aggregation Reuse Principle , CARP) | 优先使用对象组合，而不是继承来达到复用的目的 | ★★★★☆ |
| 迪米特法则 (Law of Demeter, LoD) | 每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位 | ★★★☆☆ |

# SRP 单一职责原则



- 单一职责原则定义
  - 单一职责原则是最简单的面向对象设计原则，用于控制类的粒度大小

**单一职责原则：**一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。

**Single Responsibility Principle (SRP):** Every object should have **a single responsibility**, and that responsibility should be entirely encapsulated by the class.

- 就一个类而言，应该仅有一个引起它变化的原因
- There should never be more than one reason for a class to change.

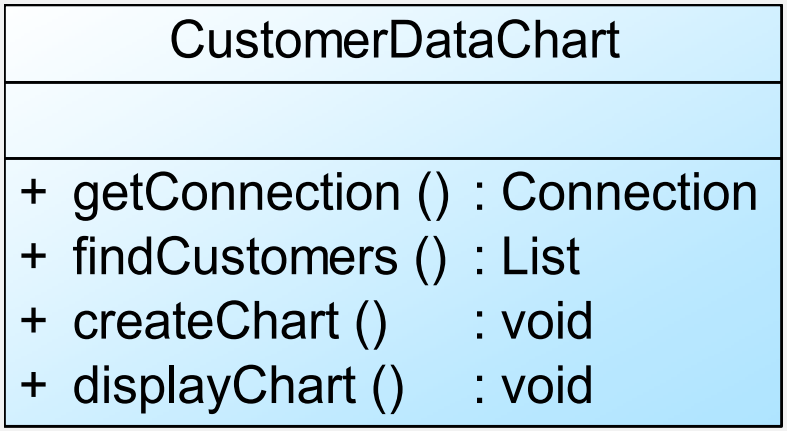
# SRP 单一职责原则

- 单一职责原则分析
  - 一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小
  - 当一个职责变化时，可能会影响其他职责的运作
  - 将这些职责进行分离，将不同的职责封装在不同的类中
  - 将不同的变化原因封装在不同的类中
  - 单一职责原则是实现高内聚、低耦合的指导方针

# SRP 单一职责原则

- 单一职责原则实例
  - 实例说明

某软件公司开发人员针对CRM（Customer Relationship Management，客户关系管理）系统中的客户信息图表统计模块提出了如下图所示的初始设计方案。



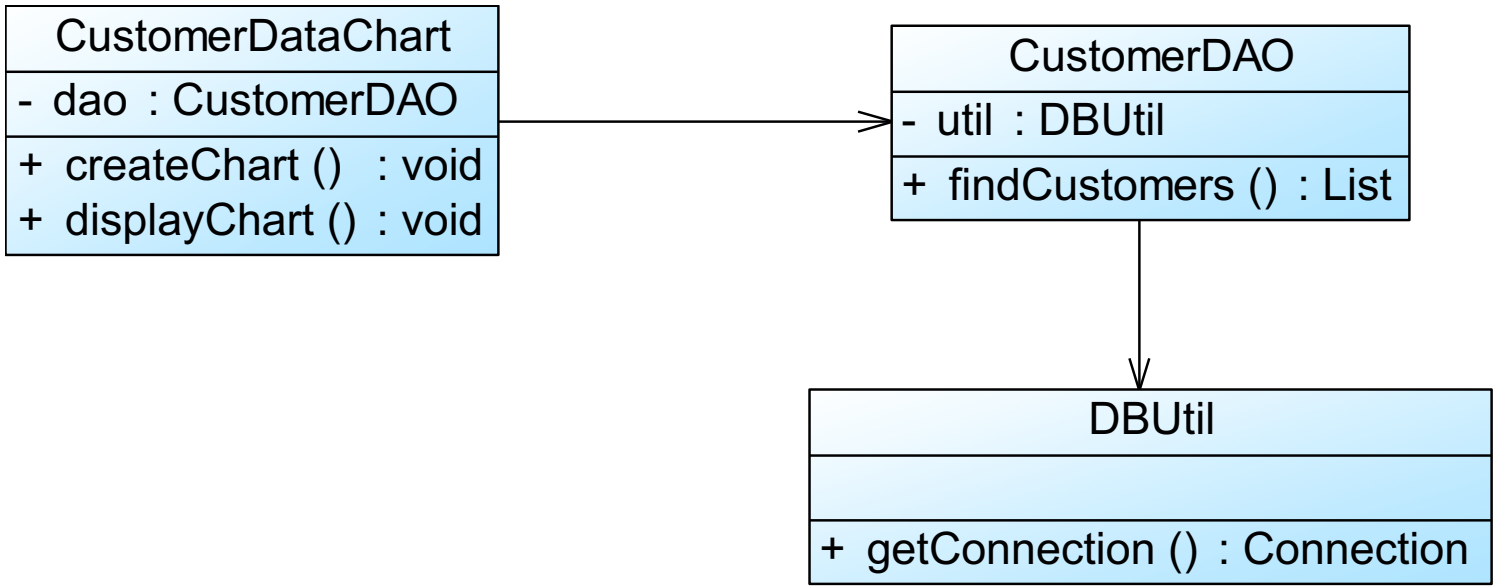

初始设计方案结构图

在上图中，getConnection()方法用于连接数据库，findCustomers()用于查询所有的客户信息，createChart()用于创建图表，displayChart()用于显示图表。

现使用单一职责原则对其进行重构。

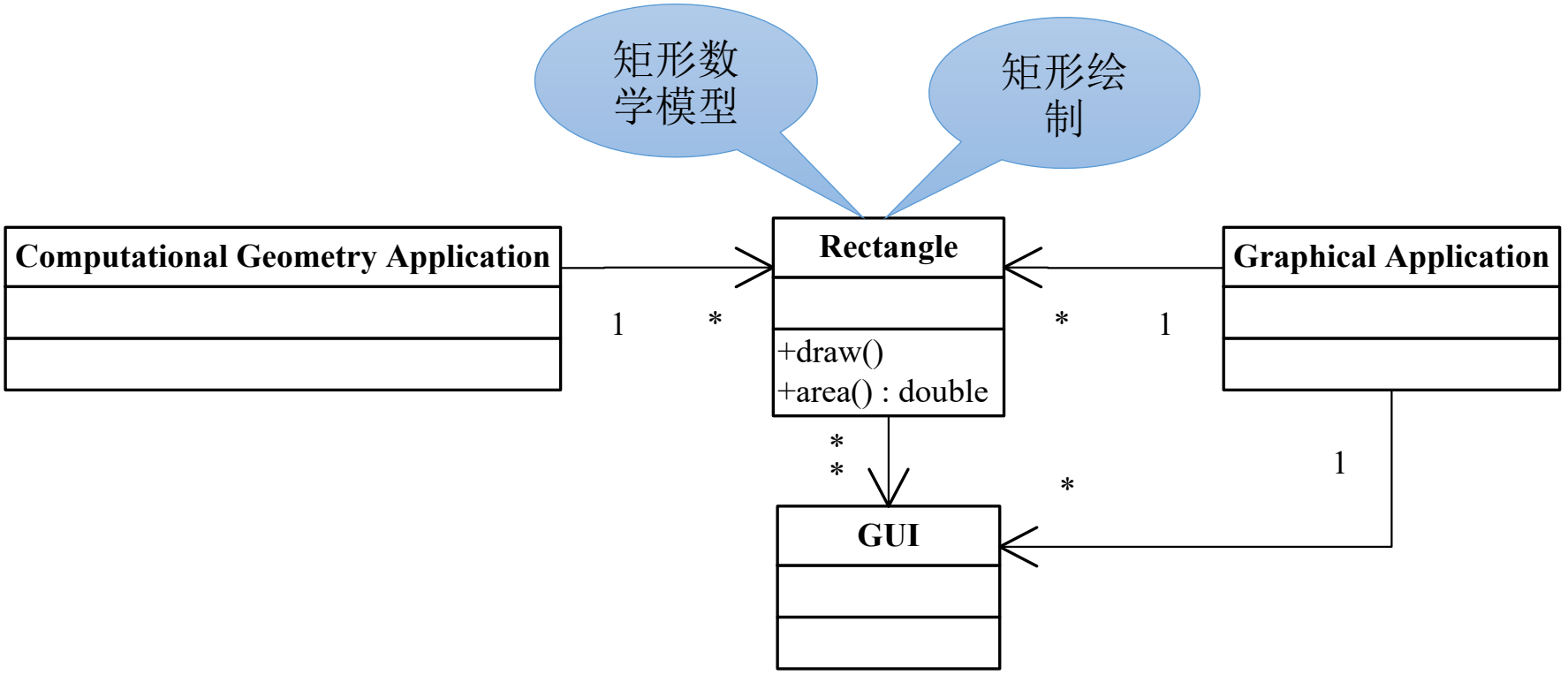
# SRP 单一职责原则

- 单一职责原则实例
  - 实例解析



# SRP 单一职责原则



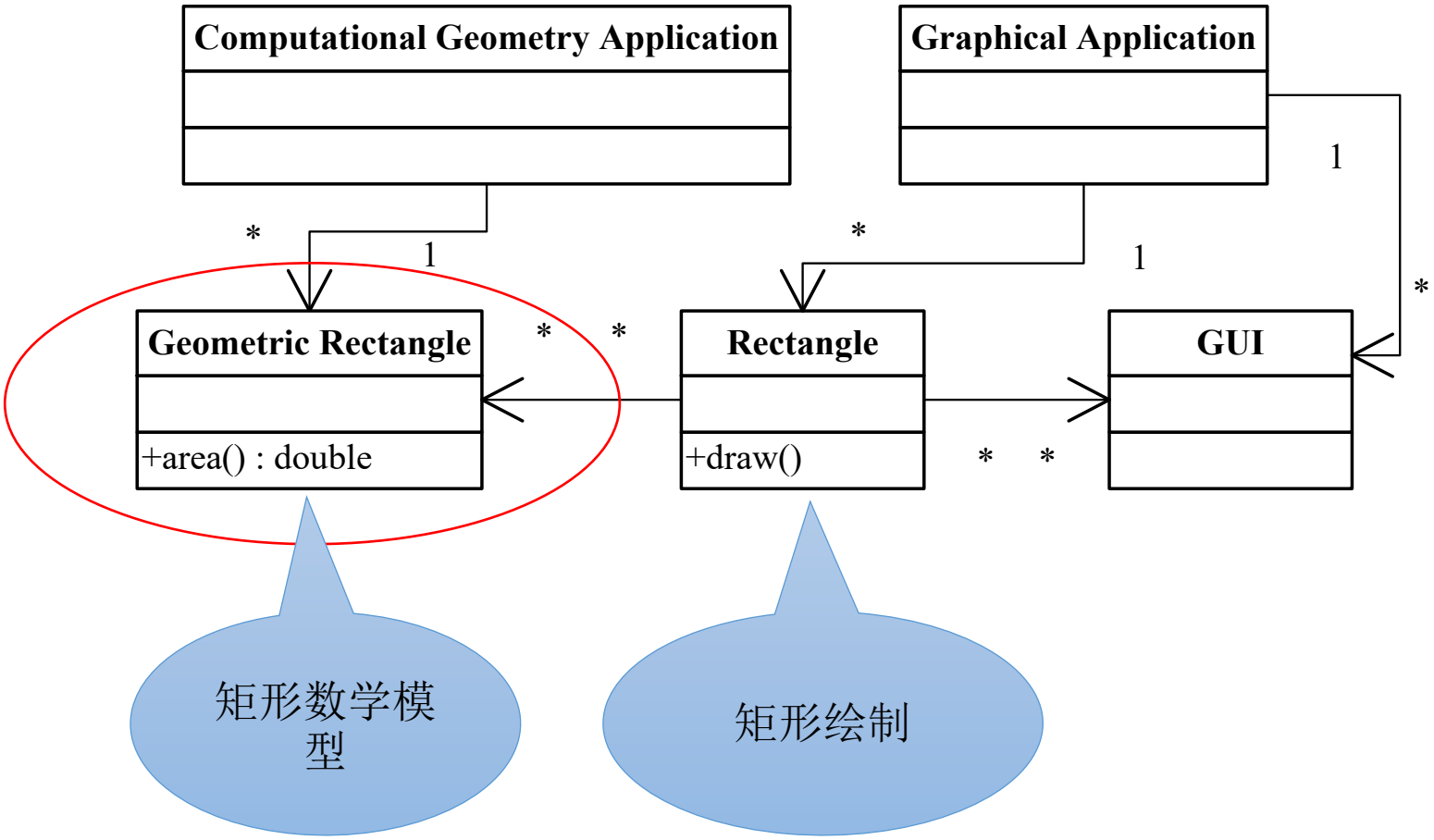



在绘制图形的程序中，既有图形作为数学模型的操作功能，又有图形的绘制功能。上图中，Rectangle类中的area()是图形的数学模型功能，draw()是图形的绘制功能。

现使用单一职责原则对其进行重构。

# SRP 单一职责原则




Computational Geometry Application
Graphical Application
Geometric Rectangle
+area() : double
Rectangle
+draw()
GUI
1
*
矩形数学模型
矩形绘制

# OCP 开放-封闭原则

- 开闭原则定义
  - 开闭原则是面向对象的可复用设计的第一块基石，是最重要的面向对象设计原则

开闭原则：软件实体应当对扩展开放，对修改关闭。

**Open-Closed Principle (OCP):** Software entities should be **open for extension**, but **closed for modification**.

OCP is the heart of OO design!

# OCP 开放-封闭原则

- 开闭原则分析
  - 开闭原则由Bertrand Meyer于1988年提出
  - 在开闭原则的定义中，软件实体可以是一个软件模块、一个由多个类组成的局部结构或一个独立的类
  - 开闭原则是指软件实体应尽量在不修改原有代码的情况下进行扩展

# OCP 开放-封闭原则



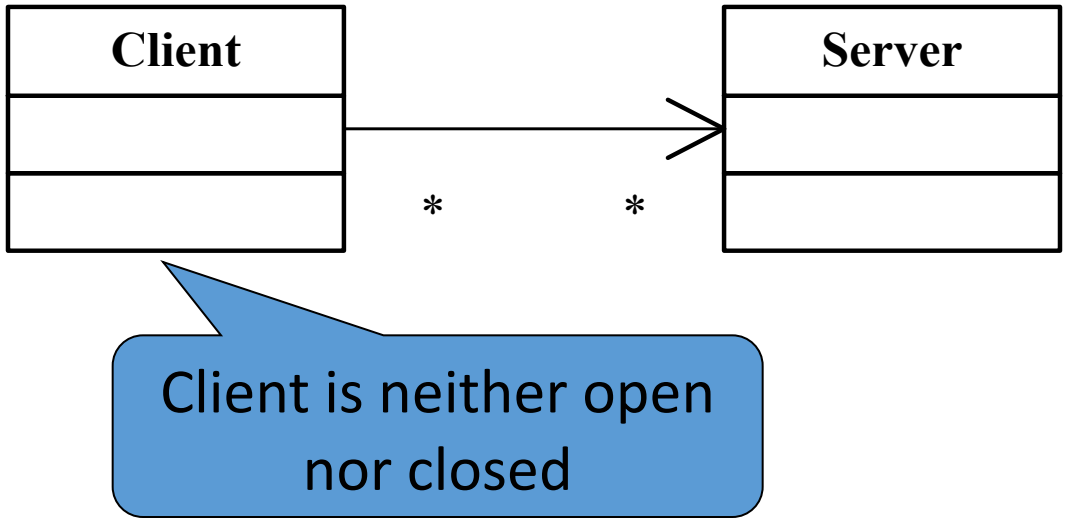
- Two primary attributes:
  - Open for extension（对扩展开放）: the behavior of the module can be extended
  - Closed for modification（对修改封闭）: extending the behavior of a module does not result in changes to the source or binary code of the module.
- 本质：不改源码，改行为
- Is it possible?

# OCP 开放-封闭原则

- 开闭原则分析
  - 抽象化是开闭原则的关键
  - 相对稳定的抽象层 + 灵活的具体层
  - 对可变性封装原则(Principle of Encapsulation of Variation, EVP)：找到系统的可变因素并将其封装起来

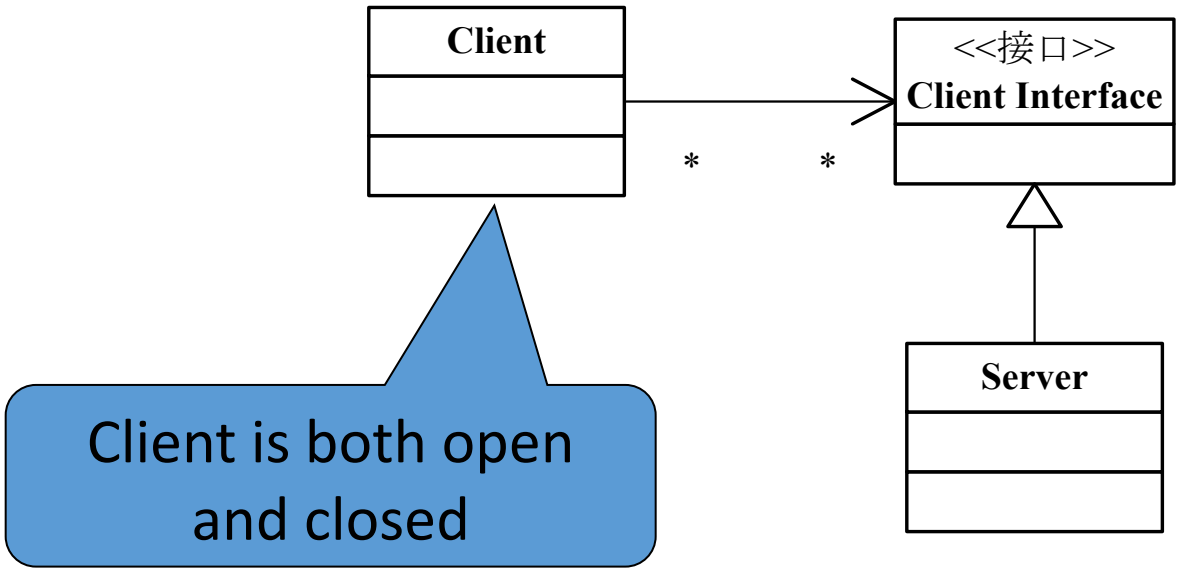
# 抽象是关键





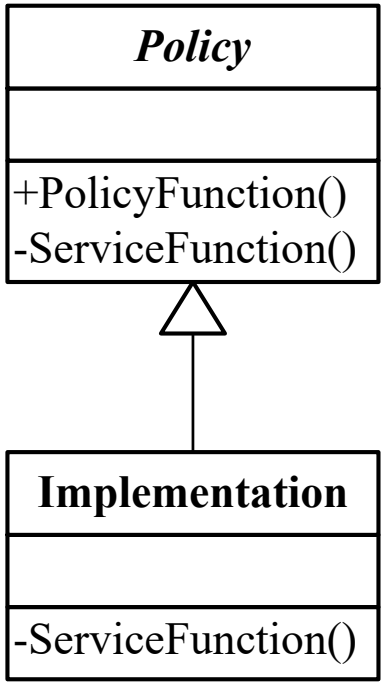
# 策略模式




Client
<<接口>>
Client Interface
*
*
Server
Client is both open and closed

# 模板方法模式






*A clear separation of generic functionality from the detailed implementation of that functionality*

# Shape应用程序



- 过程化解决方案

--------shape.h----------
```
enum ShapeType{circle,square};
struct Shape{
   ShapeType itsType;
};
```

--------circle.h---------
```
struct Circle{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
```

--------square.h---------
```
struct Square{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
```

--------drawAllShape.cc---
```
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: DrawSquare((struct
Square*)s);
        break;
        case circle: DrawSquare((struct
Circle*)s);
        break;
     }
  }
}
```

# 遵循OCP



- OOD解决方案

```
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();
}
```

# LSP: Liskov替换原则




- ✓ 里氏替换原则由2008年图灵奖得主、美国第一位计算机科学女博士、麻省理工学院教授Barbara Liskov和卡内基.梅隆大学Jeannette Wing教授于1994年提出

# LSP: Liskov替换原则

- 里氏替换原则定义

**里氏代换原则：**如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1都代换o2时，程序P的行为没有变化，那么类型S是类型T的子类型。

**Liskov Substitution Principle (LSP):** If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

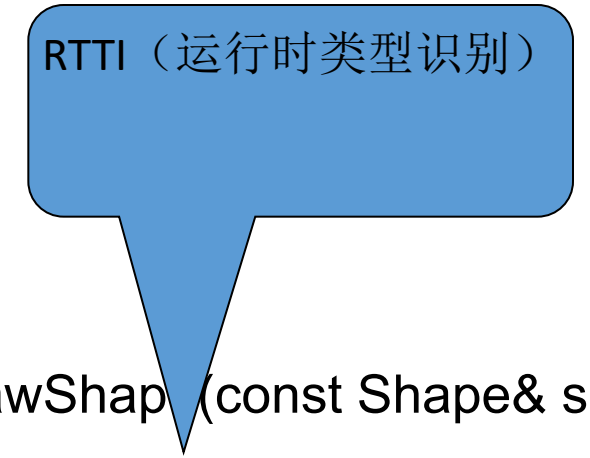
**里氏替换原则：**所有引用基类的地方必须能透明地使用其子类的对象。

**Liskov Substitution Principle (LSP):** Functions that use pointers or references to **base classes** must be able to use objects of **derived classes** without knowing it.

# LSP Violation (I)



```
struct Shape{
   enum ShapeType {square, circle} itsType;
   Shape(ShapeType t):itsType(t){ }
};
struct Circle: public Shape{
   Circle():Shape(circle){ };
   void Draw() const;
};
struct Square: public Shape{
   Square():Shape(square){ };
   void Draw() const;
};
```

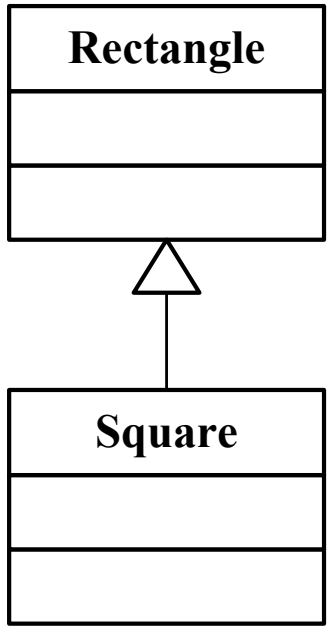



```
void DrawShape(const Shape& s ){
    if(s.itsType == Shape::square){
     static_cast<const Square&>(s).draw()
    else if (s.itsType == Shape::circle)
     static_cast<const Circle&>(s).draw()
}
```

# LSP Violation (II)



```
class Rectangle{
  public:
    void SetWidth(double w) {itsWidth=w;}
     void SetHeight(double h) {itsHeight=h;}
      …
  private:
     double itsWidth;
     double itsHeight;
     …
}
```

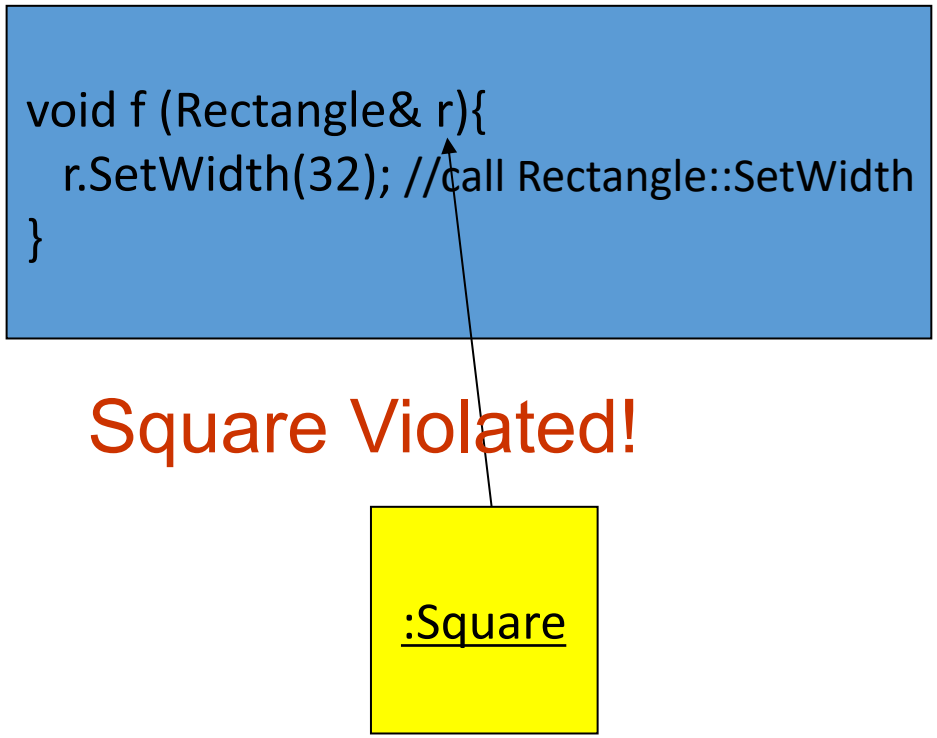



IS-A Relationship

# LSP Violation (II)



```
void Square::SetWidth(double w){
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w):
}

void Square::SetHeight(double w){
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w):
}
```



# LSP Violation (II)



```
class Rectangle{
  public:
    virtual void SetWidth(double w) {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
      …
  private:
    double itsWidth;
    double itsHeight;
    …
}
```

Are Rectangle and Square self-consistent?

:Square

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

True or false?

Violated!

# The real problem



- Validity is not intrinsic! 有效性并非本质属性
  - 模型的有效性只能通过它的客户程序来表现。
- IS-A is about Behavior
  - Behaviorally, a Square is not a Rectangle.
- Recall DbC principle about inheritance
  - Precondition 更弱；Postcondition 更强

# The real problem



- Rectangle::SetWidth(double w)的后置条件
  - Postcond: (itsWidth==w)&&(itsHeight==old.itsHeight)
  - 但是Square::SetWidth(double w)不能满足该条件
- 可以通过编写单元测试的方法来指定契约

# Heuristics and Conventions



- Violation 1:
  - Degenerate functions in derivatives 派生类中的退化函数

```
public class Base{
    public void f() { /*some code*/ }
}
Public class Derived extends Base{
    public void f() { }
}
```

- Violation 2:
  - Throwing exceptions from derivatives 从派生类中抛出异常

# LSP



- One of the enablers of the OCP。

  *LSP是使OCP成为可能的主要原则之一。*

- It is the substitutability of subtypes that allows a module, expressed in terms of a base type, to be extensible without modification. *正是子类型的可替换性才使得使用基类类型的模块在无需修改的情况下就可以扩展。*

# DIP 依赖倒置原则

- 依赖倒置原则定义

**依赖倒置原则：**高层模块不应该依赖低层模块，它们都应该依赖抽象。**抽象不应该依赖于细节，细节应该依赖于抽象。**

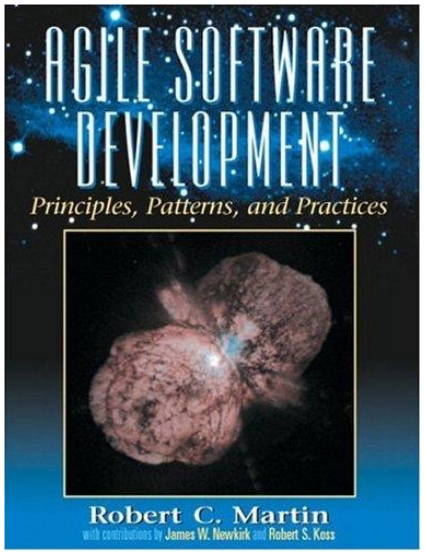
**Dependency Inversion Principle (DIP):** High level modules should not depend upon low level modules, both should depend upon abstractions. **Abstractions should not depend upon details, details should depend upon abstractions**.

✓要针对接口编程，不要针对实现编程

✓Program to an interface, not an implementation.

# DIP 依赖倒置原则

- 依赖倒置原则分析
  - DIP 依赖倒置原则是Robert C. Martin在1996年为“C++ Reporter”所写的专栏Engineering Notebook的第三篇，后来加入到他在2002年出版的经典著作《Agile Software Development, Principles, Patterns, and Practices》一书中

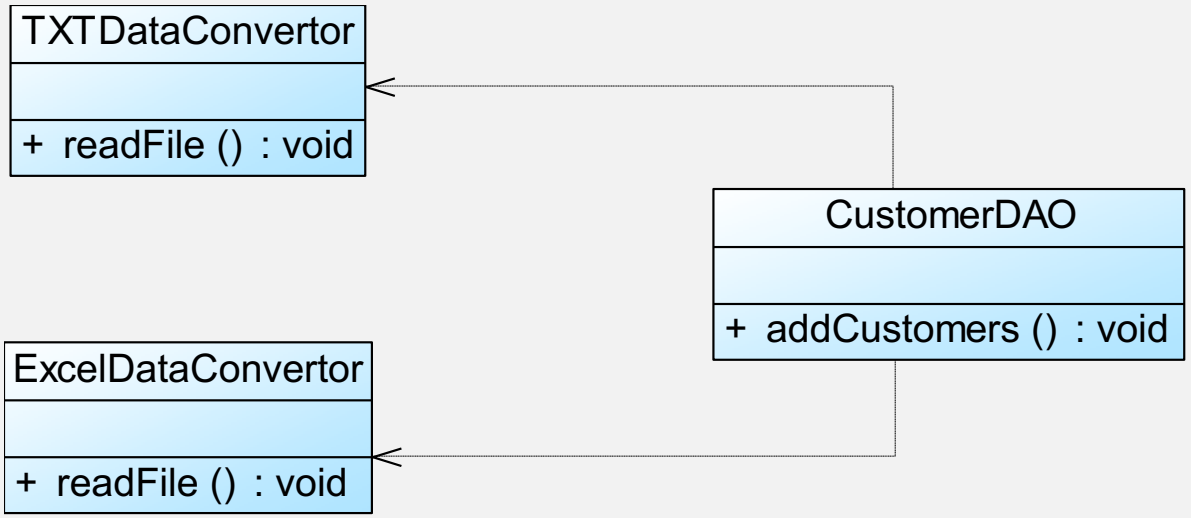


# DIP 依赖倒置原则

- 依赖倒置原则分析
  - 在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等
  - 在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中

# DIP 依赖倒置原则

某软件公司开发人员在开发CRM系统时发现：该系统经常需要将存储在TXT或Excel文件中的客户信息转存到数据库中，因此需要进行数据格式转换。在客户数据操作类CustomerDAO中将调用数据格式转换类的方法来实现格式转换，初始设计方案结构如图所示：

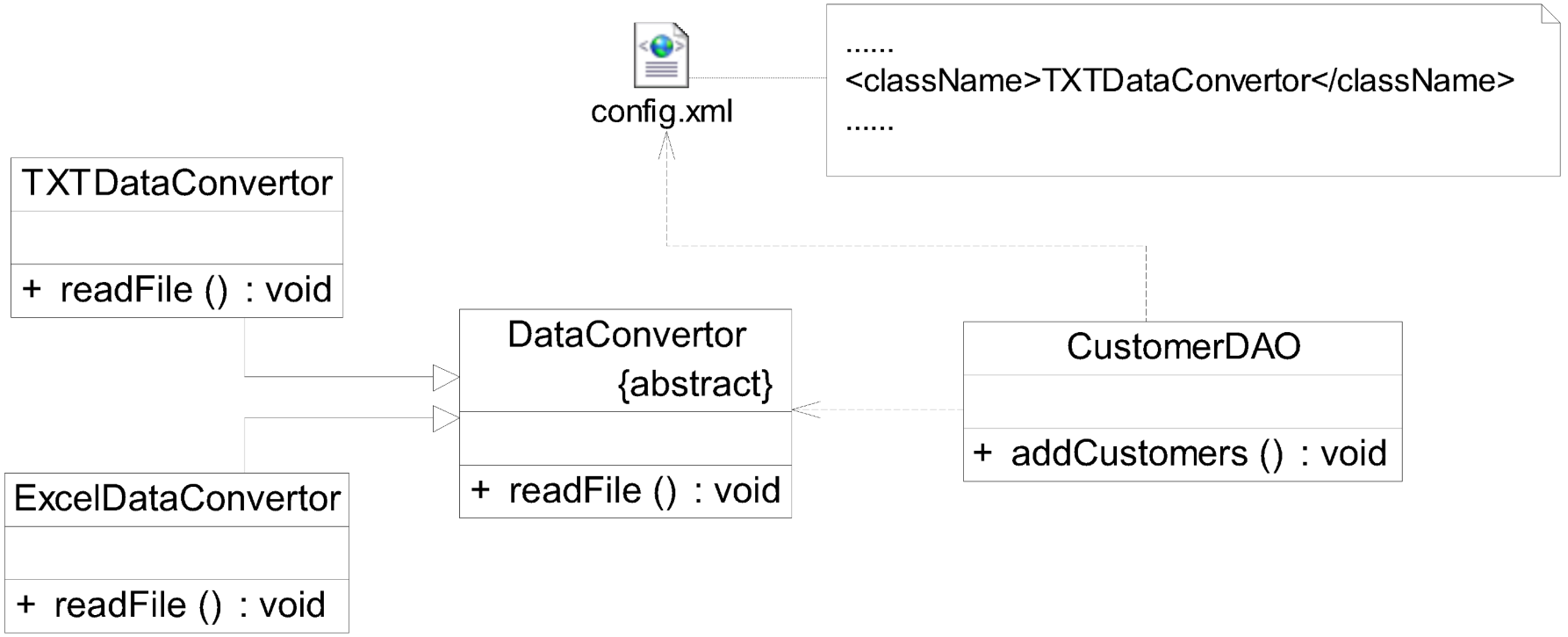



**初始设计方案结构图**

在编码实现图所示结构时，该软件公司开发人员发现该设计方案存在一个非常严重的问题，由于每次转换数据时数据来源不一定相同，因此需要经常更换数据转换类，例如有时候需要将TXTDataConvertor改为ExcelDataConvertor，此时需要修改CustomerDAO的源代码，而且在引入并使用新的数据转换类时也不得不修改CustomerDAO的源代码，系统扩展性较差，违反了开闭原则，现需要对该方案进行重构。

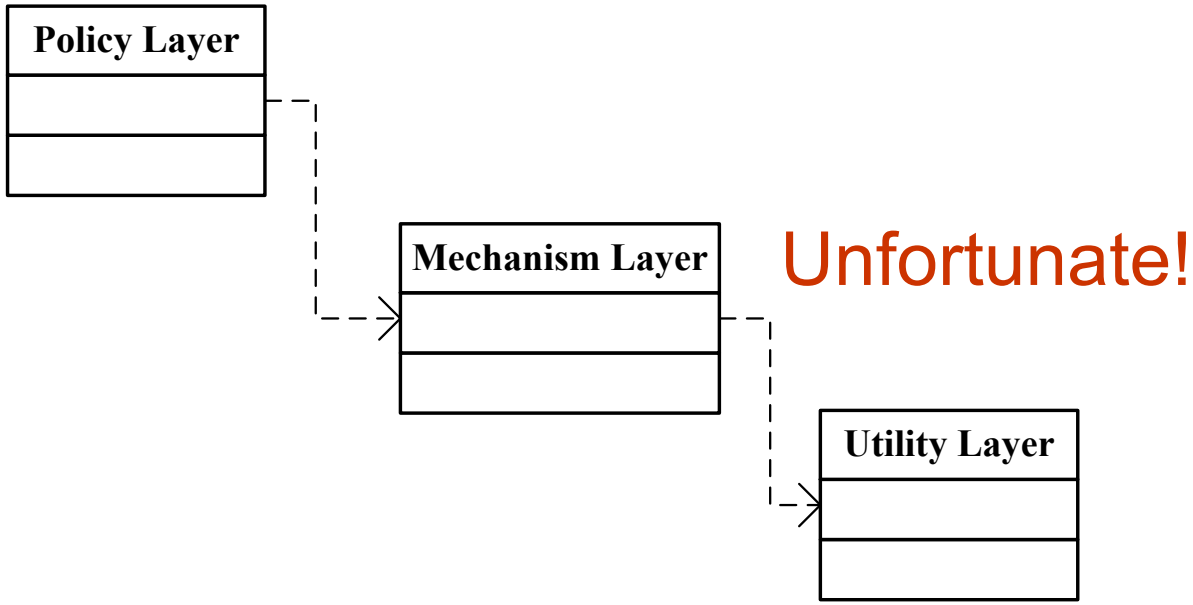
# DIP 依赖倒置原则

- OCP/LSP/DIP综合实例
  - 实例解析



# Laying

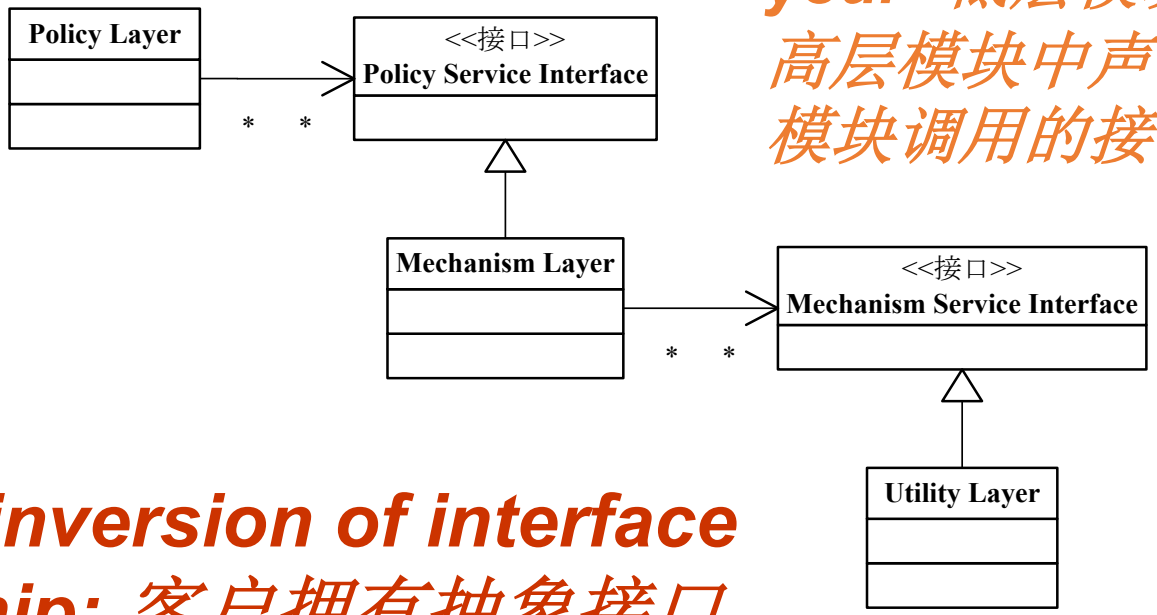
- Booch: *"… all well structured OO architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface."*



# Laying



- Inverted layers



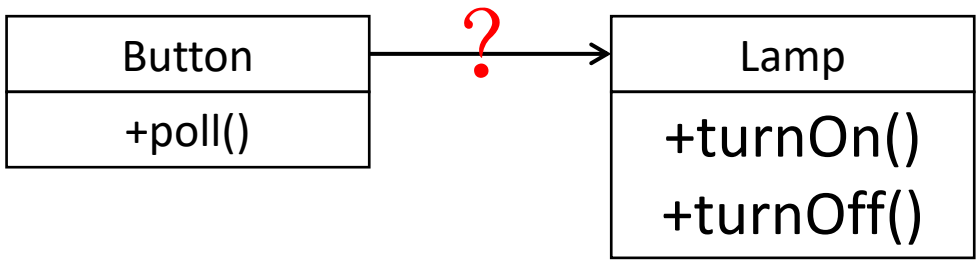

***Hollywood principle: "don't call us, we'll call you." 低层模块实现了在高层模块中声明并被高层模块调用的接口。***

***also an inversion of interface ownership: 客户拥有抽象接口，服务者则从这些抽象接口派生。***

# Example



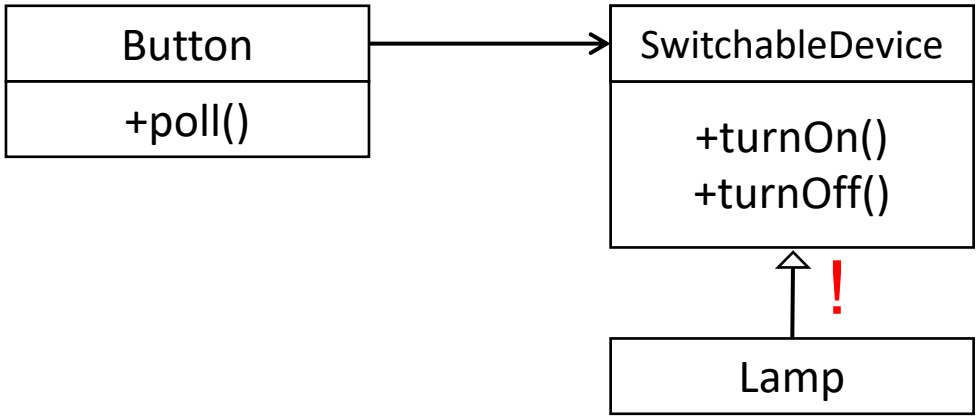
```
public class Button{

    private Lamp itsLamp;

    public void poll(){

        if (/* some condition */)

            itsLamp.turnOn();

     }

}
```



# Example



```
public class Button{

    private ButtonServer bs;

    public void poll(){

        if (/* some condition */)

            bs.turnOn();

     }

}
```



# Heuristic



- “Depend on abstractions”依赖于抽象
  - 不应该依赖于具体类——程序中所有的依赖关系都应该终止于抽象类或者接口
- According to it:
  - 任何变量都不应该持有一个指向具体类的指针或引用
  - 任何类都不应该从具体类派生
  - 任何方法都不应该覆写它的任何基类中的已经实现了的方法
  - 例外：可以依赖稳定的具体类，比如String

# DIP



- 依赖关系的倒置正是好的面向对象设计的标志所在。
- 如果程序的依赖关系是倒置的，它就是面向对象的设计，否则就是过程化的设计。
- DIP是实现许多OO技术所宣称的好处的基本低层机制。它的正确应用对于创建可重用的框架来说是必须的。

# 接口隔离原则

- 接口隔离原则定义

**接口隔离原则：**客户端不应该依赖那些它不需要的接口。

**Interface Segregation Principle (ISP):** Clients **should not be forced to depend upon interfaces that they do not use**.

# 接口隔离原则

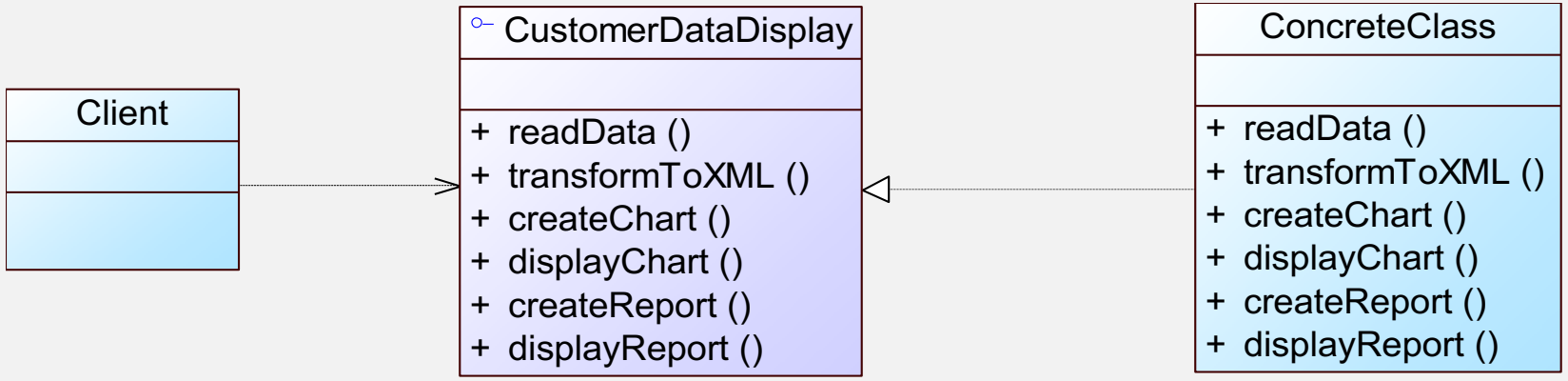
- 接口隔离原则分析
  - 当一个接口太大时，需要将它分割成一些更细小的接口
  - 使用该接口的客户端仅需知道与之相关的方法即可
  - 每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干

# 接口隔离原则

- 接口隔离原则分析
  - “接口”定义(1)：一个类型所提供的所有方法特征的集合。一个接口代表一个角色，每个角色都有它特定的一个接口，“角色隔离原则”
  - “接口”定义(2)：狭义的特定语言的接口。接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口，每个接口中只包含一个客户端所需的方法，“定制服务”

# 接口隔离原则

某软件公司开发人员针对CRM系统的客户数据显示模块设计了如图所示CustomerDataDisplay接口，其中方法readData()用于从文件中读取数据，方法transformToXML()用于将数据转换成XML格式，方法createChart()用于创建图表，方法displayChart()用于显示图表，方法createReport()用于创建文字报表，方法displayReport()用于显示文字报表。



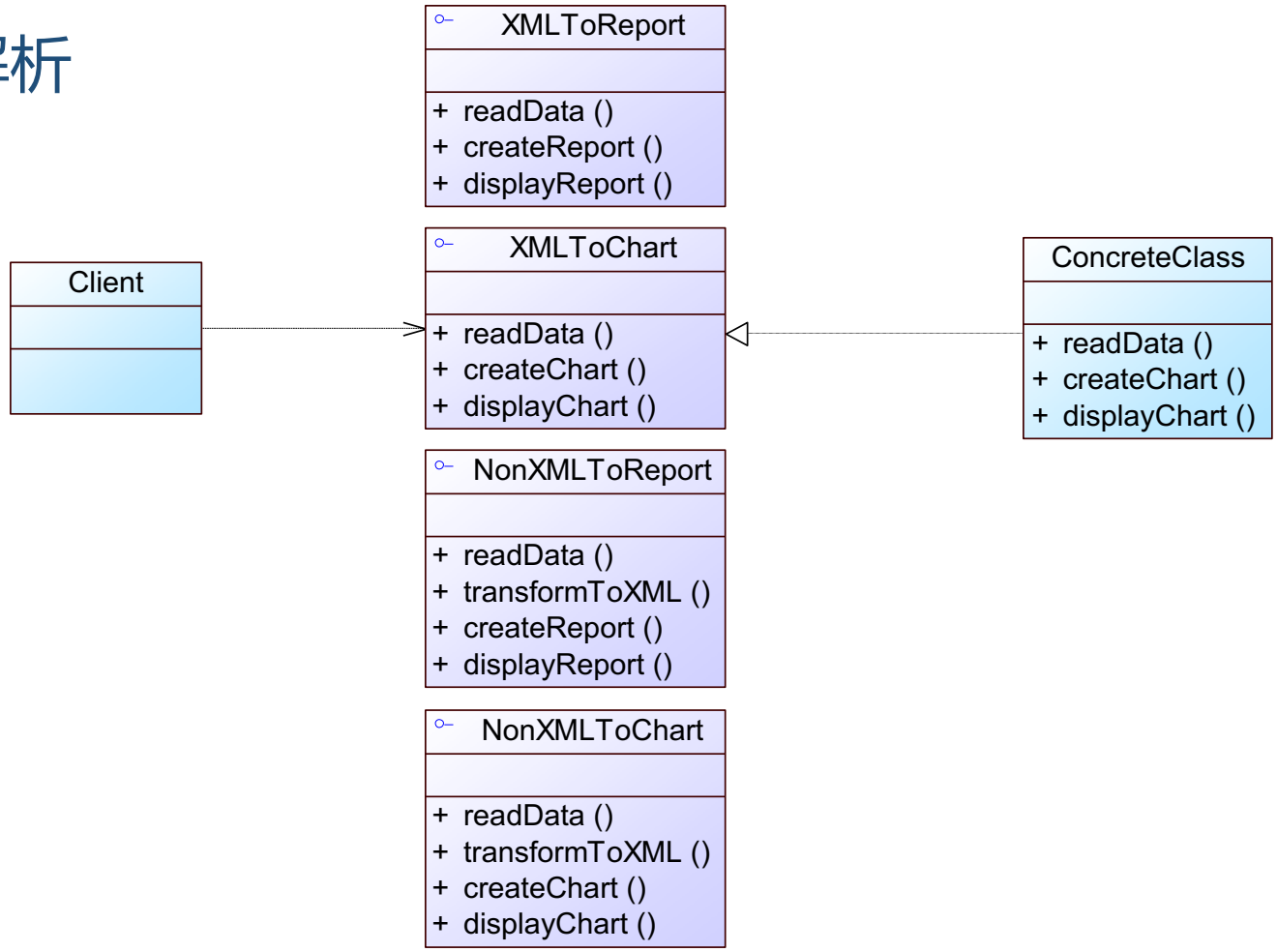

初始设计方案结构图

在实际使用过程中开发人员发现该接口很不灵活，例如：如果一个具体的数据显示类无须进行数据转换（源文件本身就是XML格式），但由于实现了该接口，不得不实现其中声明的transformToXML()方法（至少需要提供一个空实现）；如果需要创建和显示图表，除了需要实现与图表相关的方法外，还需要实现创建和显示文字报表的方法，否则程序在编译时将报错。

现使用接口隔离原则对其进行重构。

# 接口隔离原则

- 接口隔离原则实例
  - 实例解析



# ISP 接口隔离原则



- Clients should not be forced to depend on methods that they do not use. 不应该强迫客户依赖于它们不用的方法。
- Deals with the disadvantage of “fat” interfaces – whose interfaces are not cohesive. 处理**接口不是内聚的胖接口的**缺点

# Example



## common door!

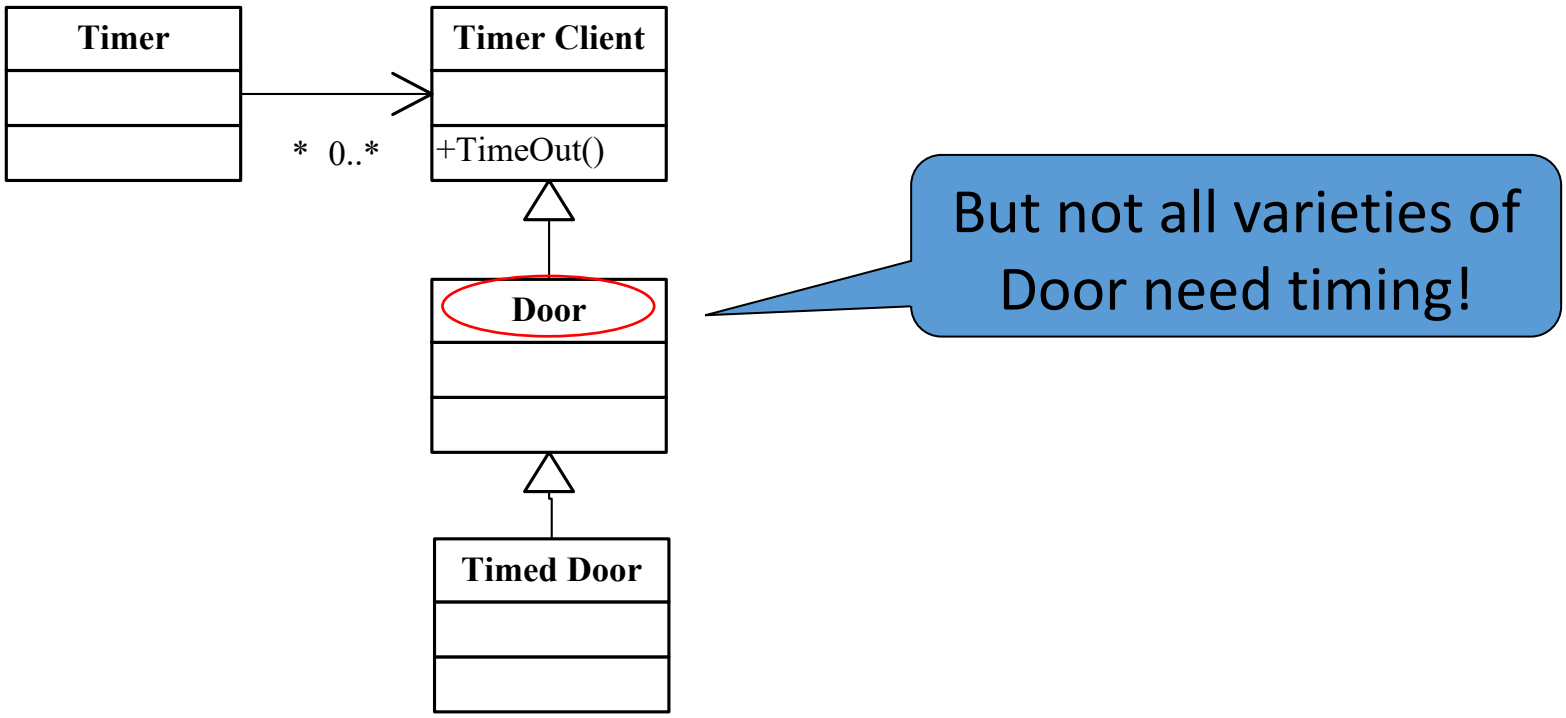
```
class Door {

   public:

      virtual void Lock() = 0;

      virtual void Unlock() = 0;

      virtual bool IsDoorOpen() = 0;

}
```

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

class TimerClient {
   public:
     virtual void TimeOut () = 0;
}
```

**How about a timed door?**

# Interface Pollution

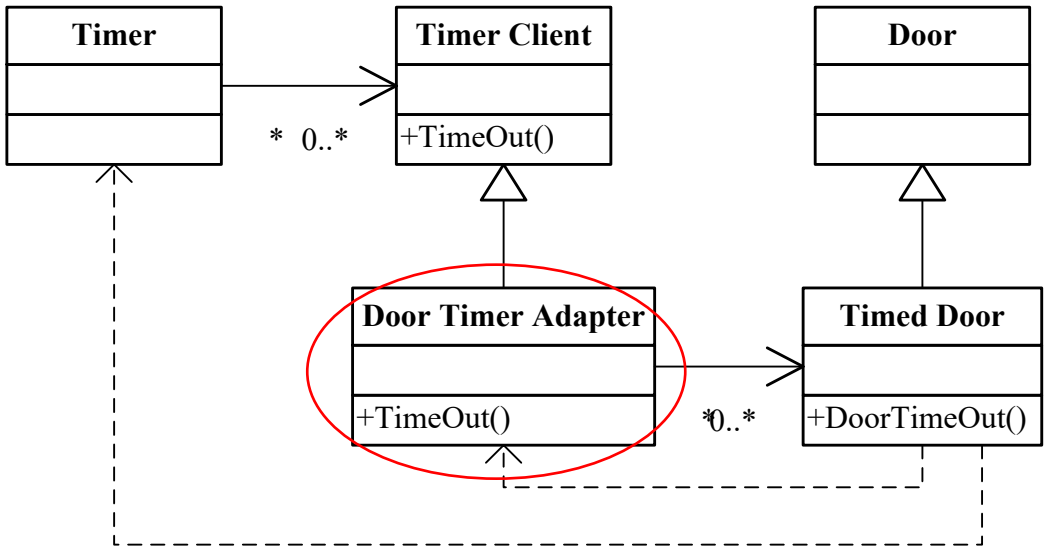




# Separate Interfaces



- Solution 1: adapter 使用委托分离接口



# Separate Interfaces

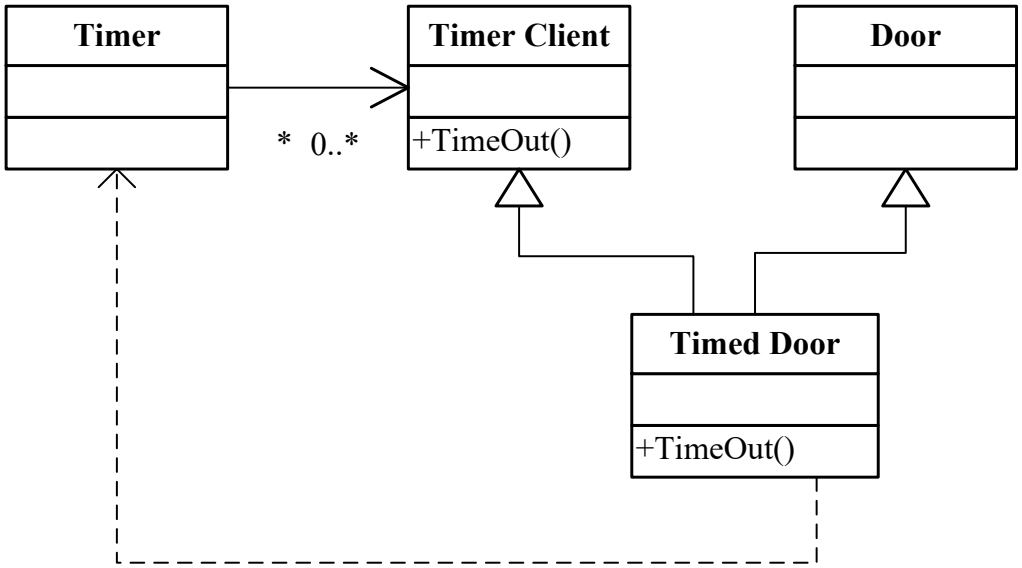


```
class TimedDoor: public Door{
  public: virtual void DoorTimeOut(int timeoutId);
}

Class DoorTimerAdapter: public TimeClient {
   pubic:
      DoorTimerAdapter(TimedDoor& theDoor): itsTimedDoor(theDoor){}
      virtual void TimeOut(int timeoutId){
            itsTimedDoor.DoorTimeOut(timeoutId);}
   private:
       TimedDoor& itsTimedDoor;
}
```

# Separate Interfaces



- Solution 2: multiple inheritance




```
class TimedDoor: public Door, public TimerClient{
  public: virtual void TimeOut(int timeoutId);
}
```

# Heuristic



- 客户程序应该仅仅依赖于它们实际调用的方法！
- 方法：把胖类的接口分解为多个特定于客户程序的接口
- 目标：高内聚，低耦合！

# CARP 合成/聚合复用原则

- 合成复用原则定义
  - 合成复用原则又称为组合/聚合复用原则 (Composition/ Aggregate Reuse Principle, CARP)

**合成复用原则：**优先使用对象组合，而不是继承来达到复用的目的。

**Composite Reuse Principle (CRP)：** **Favor composition** of objects **over inheritance** as a **reuse** mechanism.

# CARP 合成/聚合复用原则



- Composition(合成) vs. Aggregation(聚合)
  - 聚合表示“拥有”关系或者整体与部分的关系
  - 合成是一种强得多的“拥有”关系——部分和整体的生命周期是一样的。
  - 换句话说：合成是值的聚合（Aggregation by Value），而一般说的聚合是引用的聚合（Aggregation by Reference）

# 复用



- 复用的基本种类
  - 合成/聚合复用：将已有对象纳入到新对象中，使之成为新对象的一部分。
  - 继承

# 复用



- 继承的优点：
  - 新类易实现
  - 易修改或扩展
- 继承的缺点：
  - 继承复用破环包装，白箱复用.
  - 超类发生变化，子类不得不改变.
  - 继承的实现是静态的，不能在运行时改变.

# 复用



- 合成/聚合的优点
  - 新对象存取成分对象的唯一方法是通过成分对象的接口。
  - 黑箱复用，因为成分对象的内部细节是新对象所看不见的。
  - 支持包装。
  - 所需的依赖较少。
  - 每一个新的类可以将焦点集中在一个任务上。
  - 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成分对象类型相同的对象。
  - 作为复用手段可以应用到几乎任何环境中去。
- 缺点：系统中会有较多的对象需要管理

# 复用



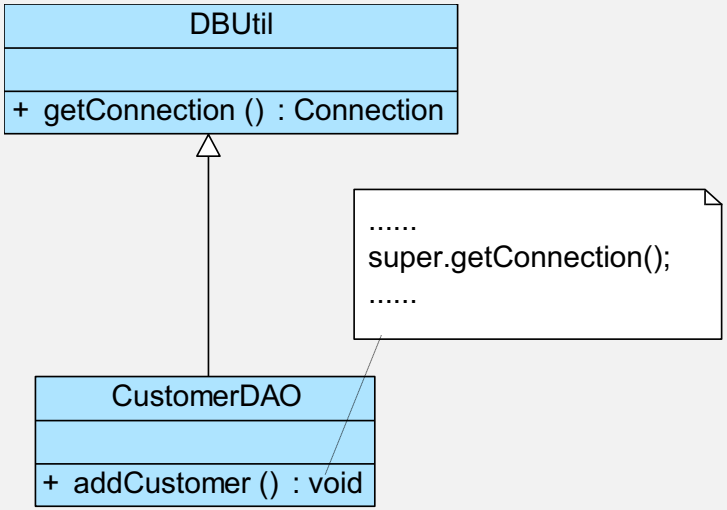
- 优先使用对象合成/聚合，而不是继承
- 利用合成/聚合可以在运行时动态配置组件的功能，并防止类层次规模的爆炸性增长
- **区分HAS-A 和 IS-A**

# 复用



- Coad法则：什么时候使用继承作为复用的工具
- 只有当以下Coad条件都满足时才应当使用继承
  - 子类是超类的一个特殊种类，而不是超类的一个角色，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。**
  - 永远不会出现需要将子类换成另外一个类的子类的情况。**如果不能肯定将来是否会变成另外一个子类的话，就不要使用继承。**
  - 子类具有扩展超类的责任，而不是具有置换掉（override）或注销掉（Nullify）超类的责任。**如果一个子类需要大量的置换掉超类的行为，那么这个类就不应该是这个超类的子类。**
  - 只有在分类学角度上有意义时，才可以使用继承。不要从工具类继承。

某软件公司开发人员在初期的CRM系统设计中，考虑到客户数量不多，系统采用Access作为数据库，与数据库操作有关的类，例如CustomerDAO类等都需要连接数据库，连接数据库的方法getConnection()封装在DBUtil类中，由于需要重用DBUtil类的getConnection()方法，设计人员将CustomerDAO作为DBUtil类的子类，初始设计方案结构如图所示。



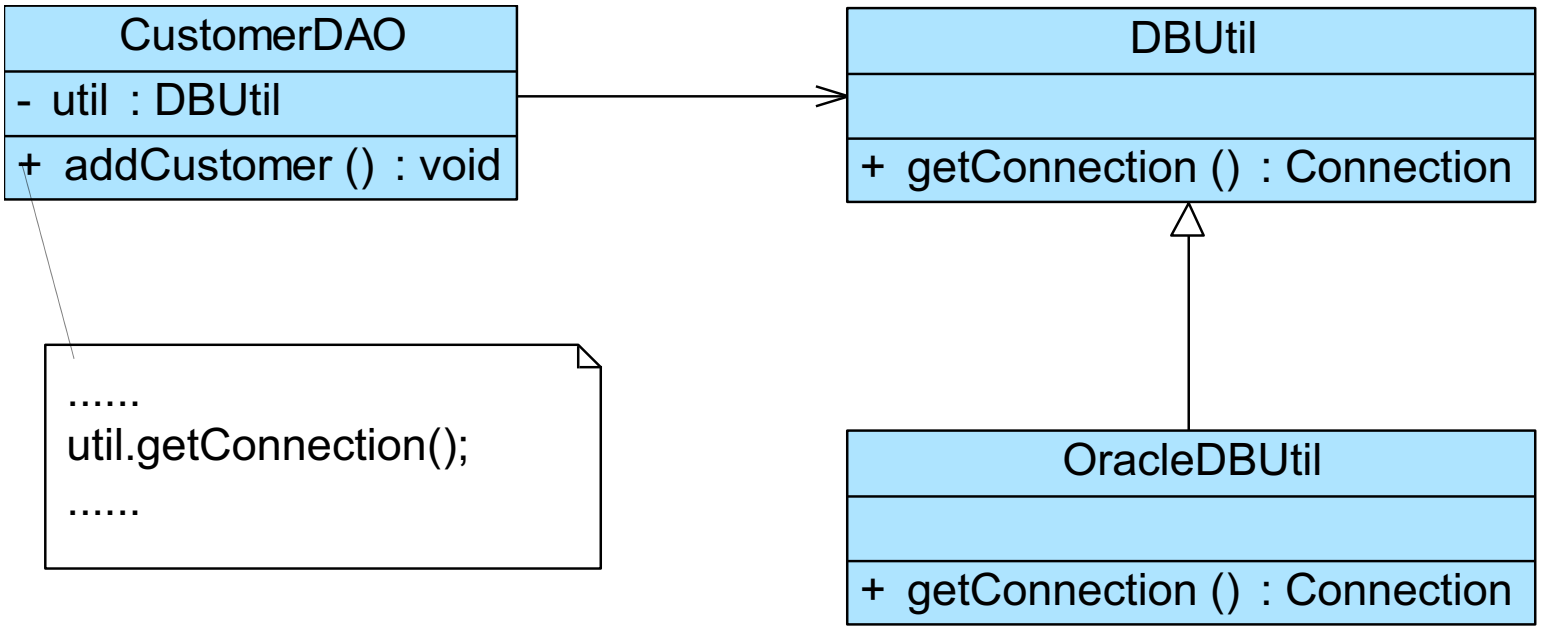

**初始设计方案结构图**

随着客户数量的增加，系统决定升级为Oracle数据库，因此需要增加一个新的OracleDBUtil类来连接Oracle数据库，由于在初始设计方案中CustomerDAO和DBUtil之间是继承关系，因此在更换数据库连接方式时需要修改CustomerDAO类的源代码，将CustomerDAO作为OracleDBUtil的子类，这将违背开闭原则。当然也可以直接修改DBUtil类的源代码，这同样也违背了开闭原则。

现使用合成复用原则对其进行重构。

# 合成复用原则

- 合成复用原则实例
  - 实例解析



# LoD 迪米特法则

- 迪米特法则定义
  - 迪米特法则又称为最少知识原则(Least Knowledge Principle, LKP)

**迪米特法则：**每一个软件单位对其他的单位都只有**最少的知识**，而且局限于那些与本单位密切相关的软件单位。

**Law of Demeter (LoD):** Each unit should have **only limited knowledge** about other units: only units "closely" related to the current unit.

# LoD



- LoD的实质是控制对象之间的信息流量，流向及信息的影响 -- **信息隐藏**
  - 在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。
  - 在类的结构设计上，每一个类都应当尽量降低成员的访问权限。一个类不应当public自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。
  - 在类的设计上，只要有可能，一个类应当设计成不变类。
  - 在对其它对象的引用上，一个类对其它对象的引用应该降到最低。

某软件公司所开发CRM系统包含很多业务操作窗口，在这些窗口中，某些界面控件之间存在复杂的交互关系，一个控件事件的触发将导致多个其他界面控件产生响应。例如，当一个按钮(Button)被单击时，对应的列表框(List)、组合框(ComboBox)、文本框(TextBox)、文本标签(Label)等都将发生改变，在初始设计方案中，界面控件之间的交互关系可以简化为如图2-9所示的结构。

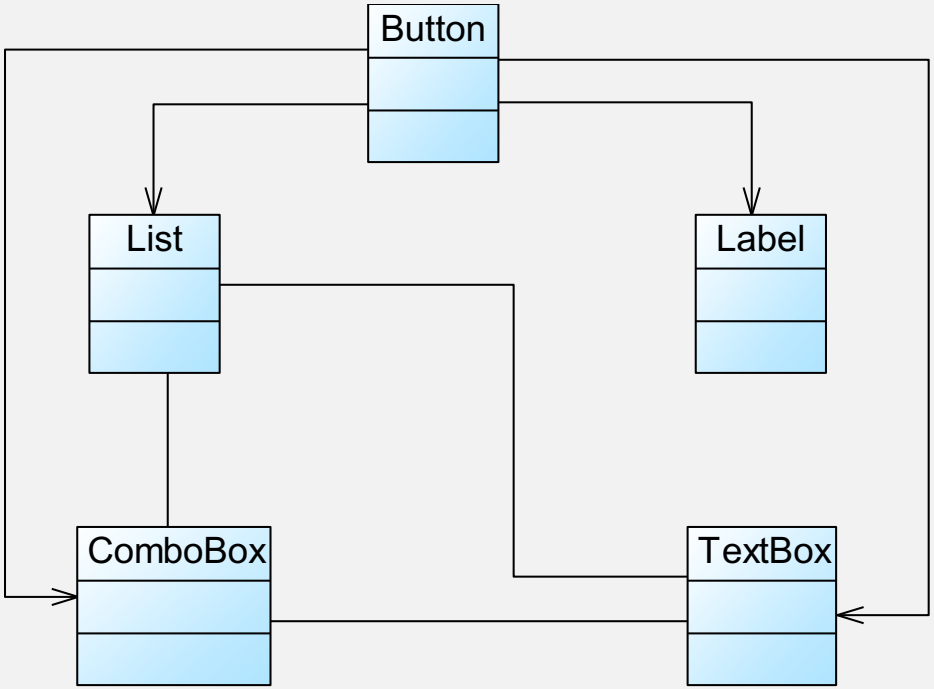



**图2-9 初始设计方案结构图**

在图2-9中，由于界面控件之间的交互关系复杂，导致在该窗口中增加新的界面控件时需要修改与之交互的其他控件的源代码，系统扩展性较差，也不便于增加和删除控件。

现使用迪米特法则对其进行重构。

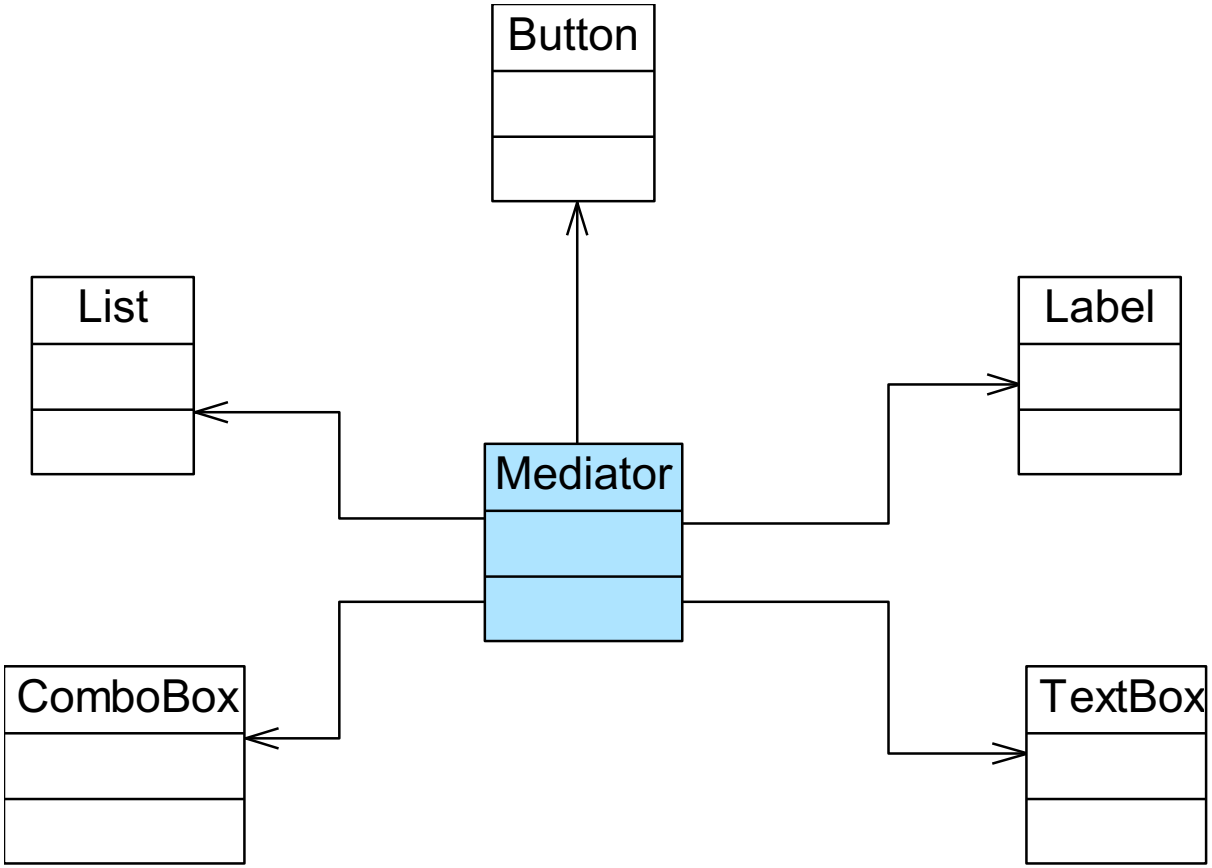
# 迪米特法则




客户信息管理窗口
客户信息管理
张无忌
请输入查询关键字：
张无忌
查询
张无忌
杨过
小龙女
令狐冲
段誉
王语嫣
黄蓉
郭靖
姓名：
张无忌
性别：
男
女
出生日期：
1980
年
10
月
2
日
联系电话：
13000001111
电子邮箱：
wuji_zhang@dp.com
增加
删除
修改

# 迪米特法则

- 迪米特法则实例
  - 实例解析



# Design Principles



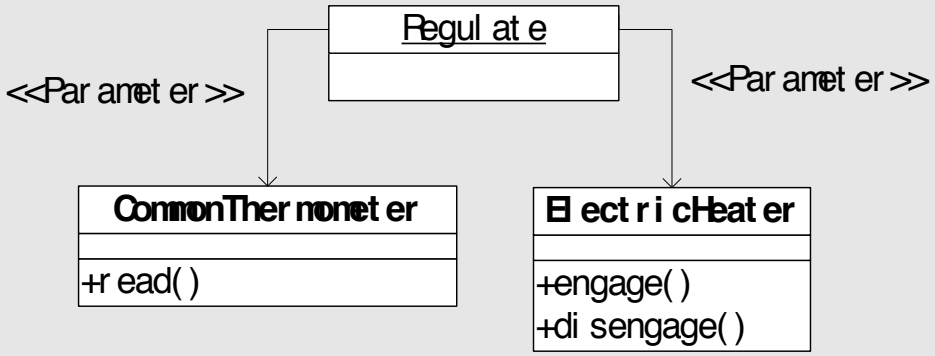
- SRP (The Single-Responsibility Principle) 单一职责原则
- OCP (The Open-Closed Principle) 开放-封闭原则
- LSP (The Liskov Substitution Principle) Liskov替换原则
- DIP (The Dependency-Inversion Principle) 依赖倒置原则
- ISP (The Interface-Segregation Principle) 接口隔离原则
- CARP (Composition/Aggregation Reuse Principle ) 合成/聚合复用原则
- LoD (Law of Demeter) 迪米特法则

# 作业（一）



- 考虑以下一个软件开发问题的设计，有一个锅炉的炉温调节软件模块，该软件模块可通过一个温度计获得锅炉的当前炉温，此外可根据当前炉温发送打开或者关闭的命令给一个加热器来控制炉温。
- 有一个软件公司的软件设计人员设计了如下的一个方案：

```
void Regulate(CommonThermometer& t,
ElectricHeater & h, double minTemp, double maxTemp)
{
    for(;;)
    {
        while(t.read()>minTemp)
            wait(1);
        h.engage();
        while(t.read()<maxTemp)
            wait(1);
        h.disengage();
    }
}
```




- 请分析一下上述的设计违背了哪些设计原则，并给出你的改进设计。（考虑后续该调节软件有可能会使用其它类型的温度计或者其它类型的加热器。）

# 作业（二）



- 结合面向对象设计原则分析正方形是否为长方形的子类？
- 有人将面向对象设计原则简单归为3条，1. 封装变化点；2. 对接口进行编程；3. 多使用组合，而不是编程。请结合本章所学内容，谈谈对这三条归纳的理解。

**提交作业到教学立方（3月31号24点截止）**