



软件工程

第六章 OO分析与设计

6-5 面向NFR的OO设计原则

徐汉川

xhc@hit.edu.cn

2017年11月6日

主要内容

- 1. 内聚度和耦合度
- 2. SOLID: 类的设计原则
- 3. 模块化: 包的设计原则
- *4. 基于模块化和SOLOD的OO设计模式



1. 内聚度和耦合度



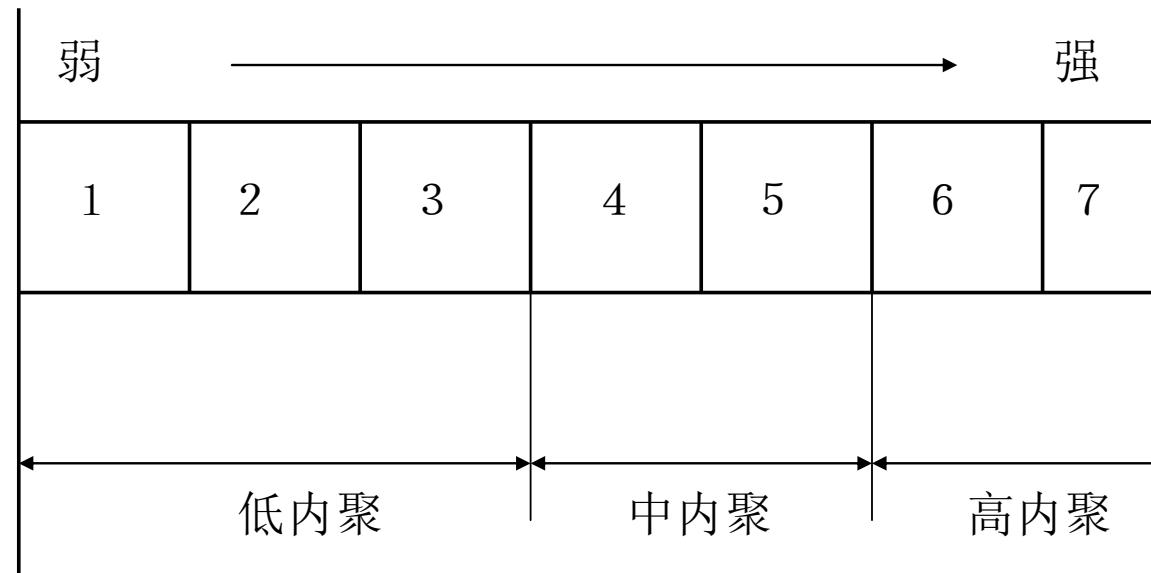
内聚度和耦合度

- 构件或类的独立性：用来判断构件或类的构造是否合理的标准。
- 从两个方面来度量：
 - 聚合度/内聚度 (Cohesion)：某一构件或类内部包含的各功能之间相互关联的紧密程度；
 - 耦合度 (Coupling)：多个构件或类之间相互关联的紧密程度；
- 设计的目标：
 - 高内聚、低耦合 (high cohesion and low coupling)

内聚度

- 聚合度/内聚度(Cohesion): 某一构件或类内部包含的各功能之间相互关联的紧密程度;
- 传统观点
 - 构件的专诚性
- 面向对象观点
 - 构件或者类只封装那些相互关系密切, 以及与构件或类自身有密切关系的属性或操作

内聚强度的划分



1. 偶然性内聚 Coincidental Cohesion
2. 逻辑性内聚 Logical Cohesion
3. 时间性内聚 Temporal Cohesion
4. 过程性内聚 Procedural Cohesion
5. 通讯性内聚 Communicational Cohesion
6. 顺序性内聚 Sequential Cohesion
7. 功能性内聚 Functional Cohesion

7. 功能性内聚

- 功能性内聚：模块中各个部分都是为完成一项单一的功能而协同工作
 - 模块只执行单一的计算并返回结果
 - 无副作用(执行前后系统状态相同)，对其他模块无影响；
 - 这类模块通常粒度最小，且不可分解；
 - 通常以“动宾短语”来命名。
- 例如：
 - 根据输入的角度，计算其正弦值；
 - 求一组数中的最大值；
- “集中精力做一件事情”

优点：

- 模块的功能只是生成特定的输出而没有副作用，容易理解该模块。
- 因功能内聚的模块没有副作用，所以模块的可复用性高。
- 功能单一、易修改、易替换、易维护

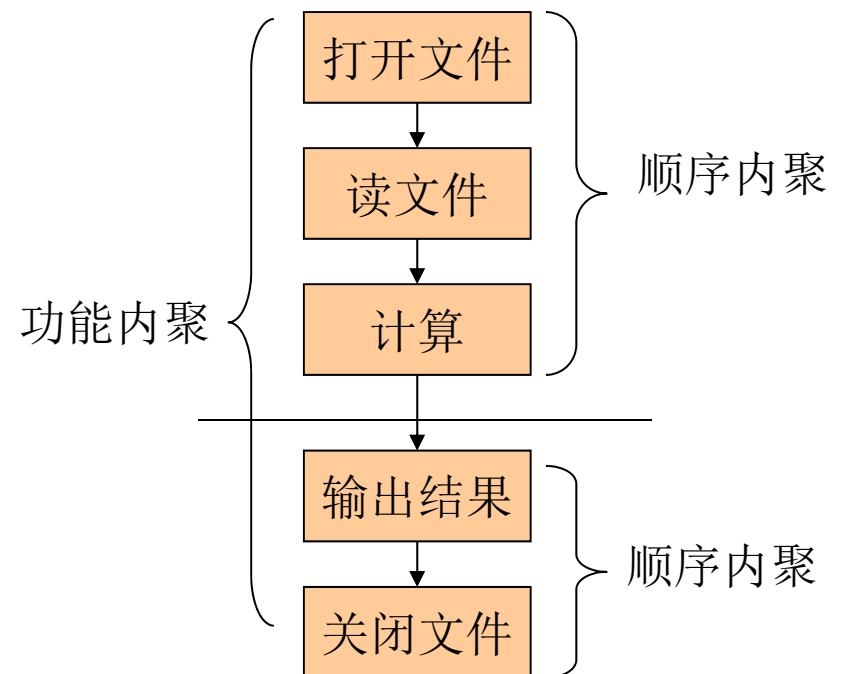
6. 顺序性内聚

■ 顺序性内聚：

- 模块完成多个功能，它们无法实现一个完整的功能；
- 各功能之间按顺序执行，形成操作序列；
- 上一个功能的输出是下一个功能的输入；
- 各功能都在同一数据结构上操作。

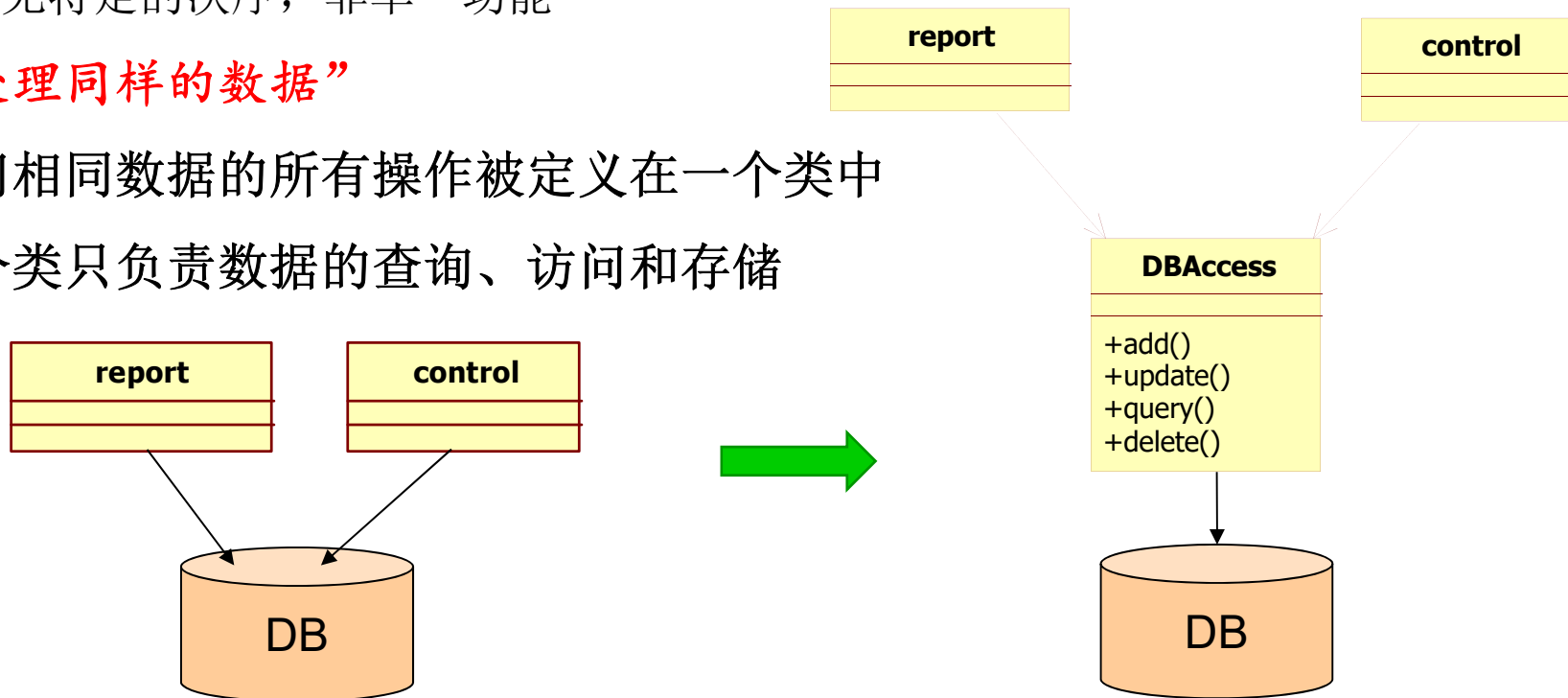
- 判断标准：能否用“动宾短语”命名？

■ “先后次序，放置在一起”



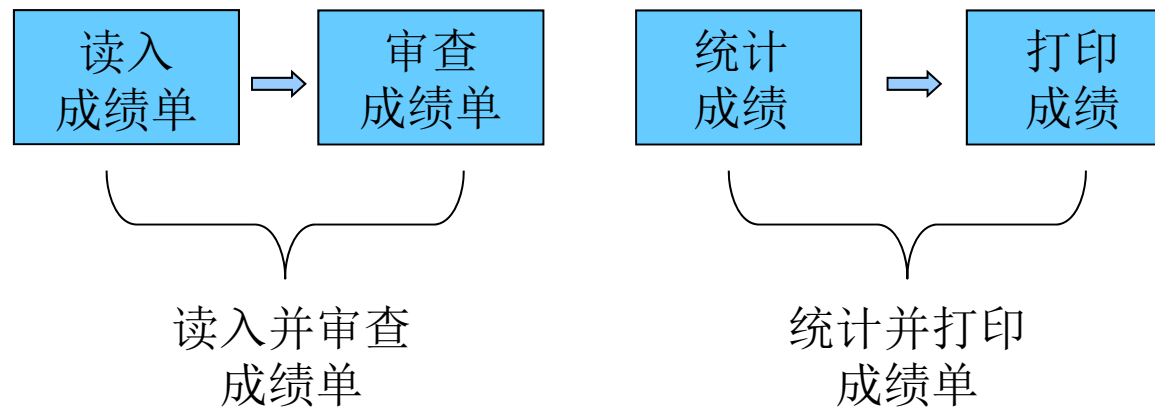
5. 通信性内聚

- 通信性内聚：
 - 模块内各部分操作访问相同的数据
 - 除此之外，再无任何关系
 - “无特定的次序，非单一功能”
- “处理同样的数据”
- 访问相同数据的所有操作被定义在一个类中
- 这个类只负责数据的查询、访问和存储



4. 过程性内聚

- 过程性内聚：
 - 模块中的操作遵循某一特定的顺序
 - 但这些顺序操作并不使用相同的数据
 - 给这类模块命名是非常困难的

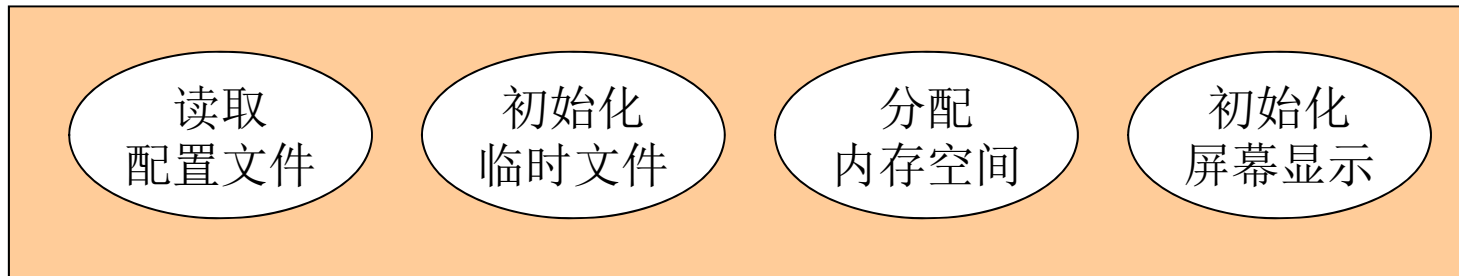


3. 时间性内聚

- 时间性内聚/暂时内聚:

- 模块的各个成分必须在同一时间段执行，但各个成分之间无必然的联系

Startup()



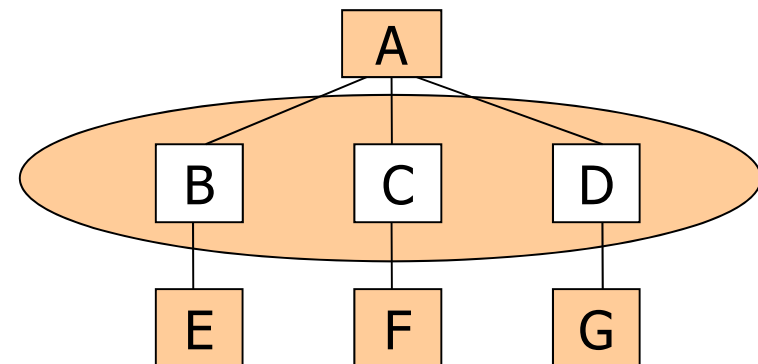
问题：不同的功能混在一个模块中，有时共用部分编码，
使局部功能的修改牵动全局。

2. 逻辑性内聚

■ 逻辑性内聚：

- 一个模块中同时含有几个操作，这些操作之间既无顺序关系，也无数据共享关系；
- 它们的执行与否由外面传进来的控制标志所决定
- 之所以称之为逻辑内聚性，是因为这些操作仅仅是因为控制流，或者说“逻辑”的原因才联系到一起的，它们都被包括在一个很大的if或者case语句中，彼此之间并没有任何其它逻辑上的联系。

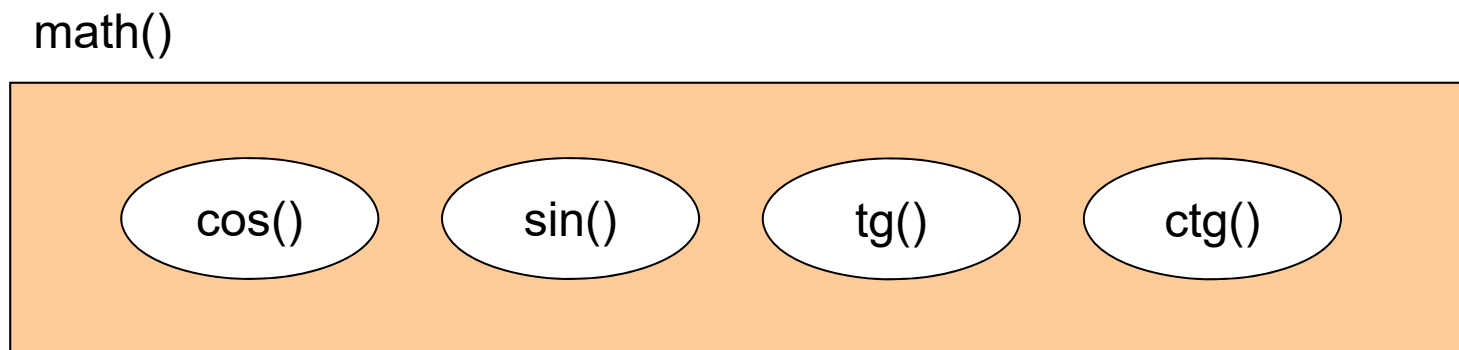
问题：接口难于理解；完成多个操作的代码互相纠缠在一起，导致严重的维护问题。



1. 偶然性内聚

- 偶然性内聚/实用内聚:

- 构成模块的各组成部分无任何关联。
- 通常用于库函数管理，将多个相互无关但功能比较类似的模块放置在同一个模块内。



缺点：产品的可维护性退化；模块不可复用，增加软件成本。

解决：将模块分成更小的模块，每个小模块执行一个操作。

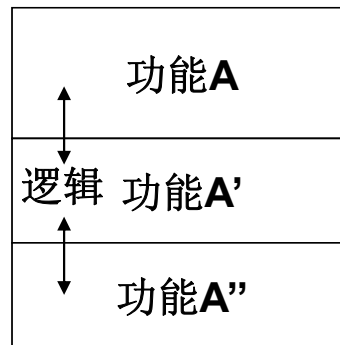
内聚强度的划分

内聚类型	说 明
偶然性内聚	各组成部分在功能上互不相关
逻辑性内聚	各组成部分逻辑功能相似
时间性内聚	各组成部分需要在同一时间内执行
过程性内聚	各组成部分必须按照某一特定的次序执行
通信性内聚	各组成部分处理公共的数据
顺序性内聚	各组成部分顺序执行，前一个的输出数据为后一个的输入数据
功能性内聚	内部所有活动均完成单一功能

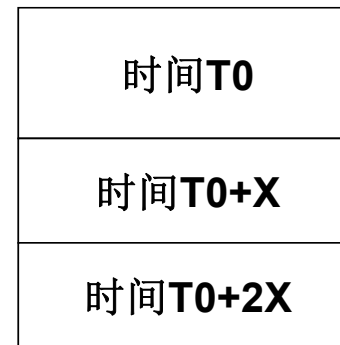
内聚强度的划分



偶然内聚
各部分不相关



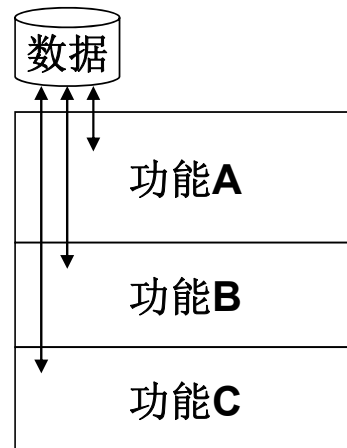
逻辑内聚
类似的功能



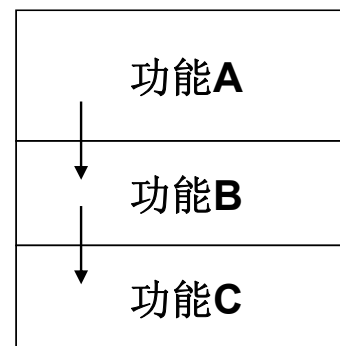
时间内聚
按时间相关



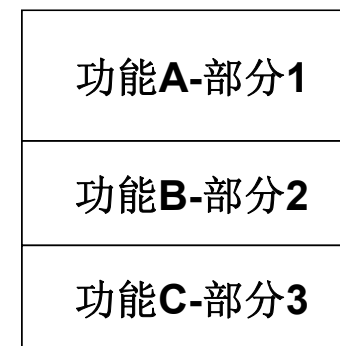
过程内聚
按功能的顺序相关



通信内聚
访问同样的数据

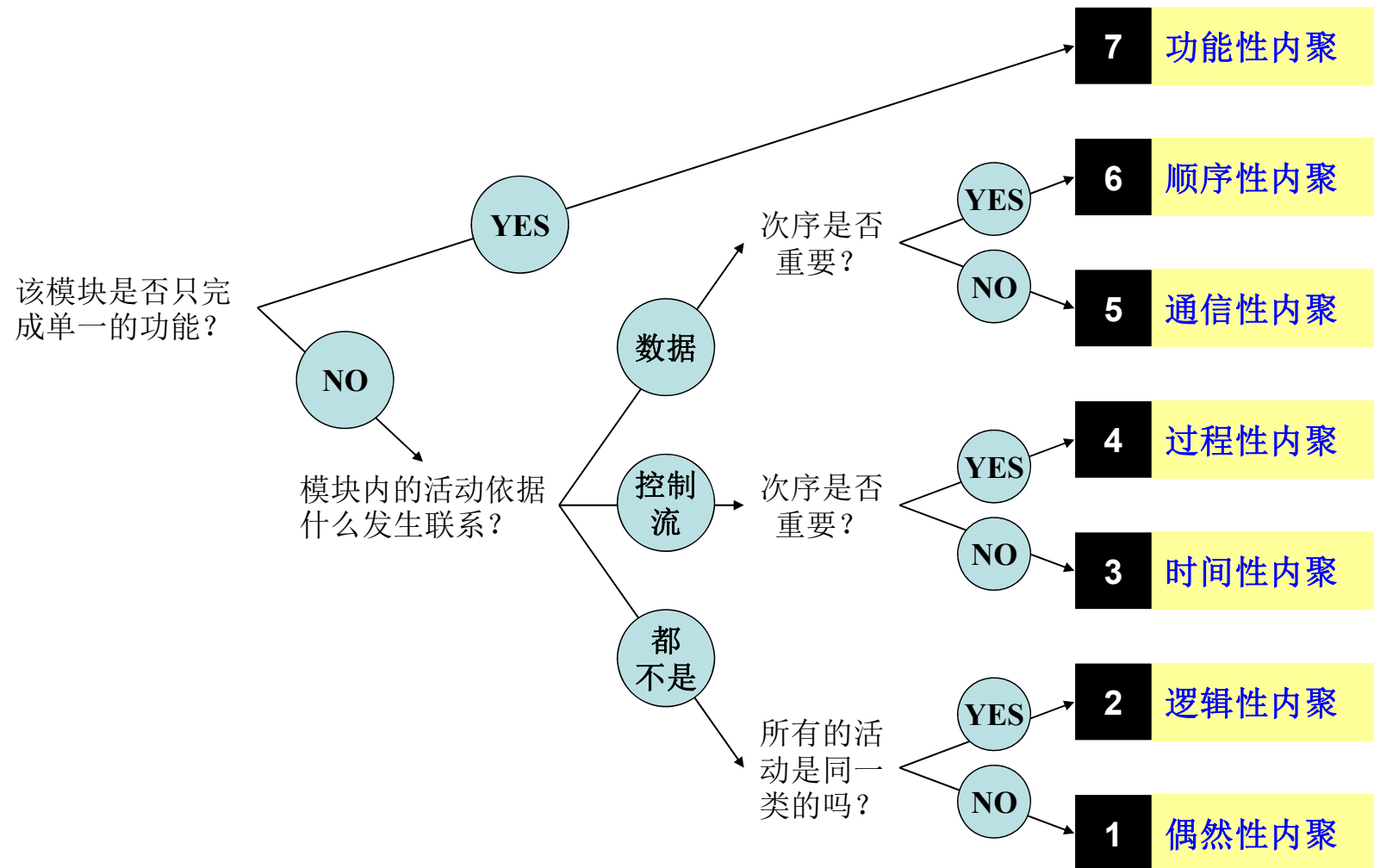


顺序内聚
某一部分的输出
是下一部分的输入



功能内聚
完备的、相关的功能

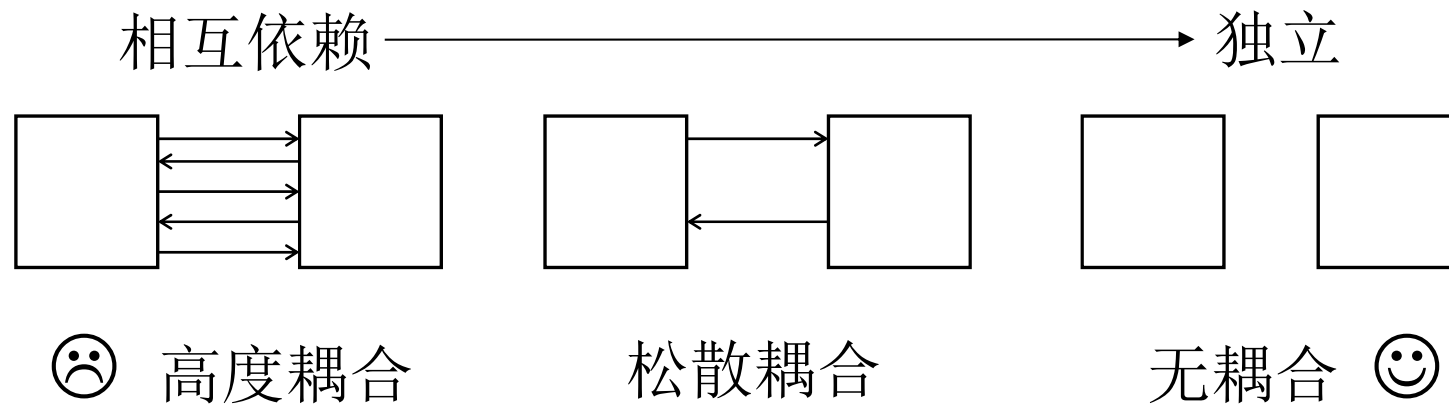
内聚强度的划分




耦合度

- **耦合度(Coupling)**: 多个构件或类之间相互关联的紧密程度
- 传统观点
 - 构件和其他构件或外部世界的连接程度
- 面向对象观点
 - 耦合性是构件或类之间彼此联系程度的一种定性度量

模块独立性之耦合度



影响模块之间发生耦合的因素

- 
- 一个构件引用另一个构件；
 - 一个构件传递给另一个构件数据；
 - 某个构件控制或调用其他构件；
 - 构件之间接口的复杂程度；

耦合度

■ 多个模块之间相互关联的紧密程度

1. 例程调用耦合
2. 数据耦合
3. 印记耦合
4. 类型使用耦合
5. 控制耦合
6. 外部耦合
7. 共用耦合
8. 内容耦合

弱耦合

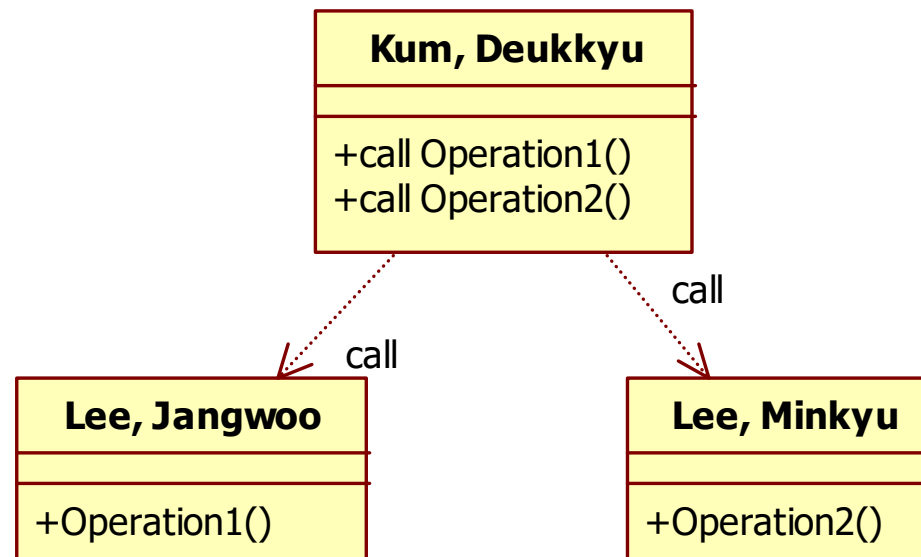
中耦合

强耦合

1. 例程调用耦合

■ 例程调用耦合

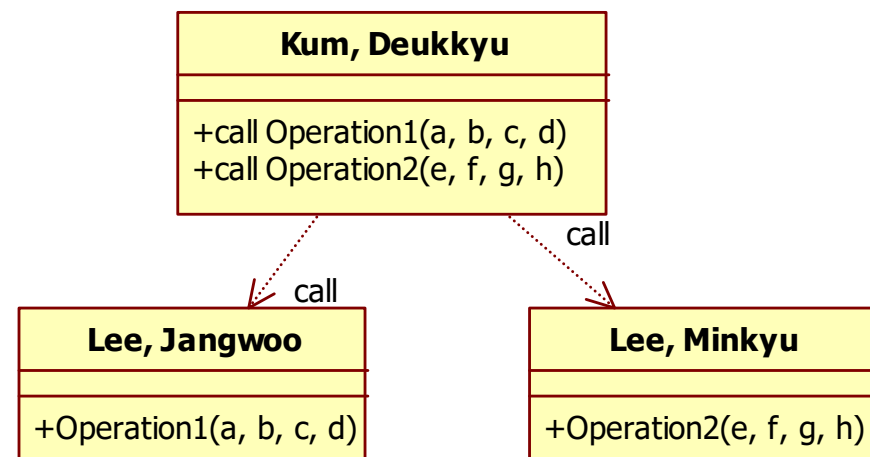
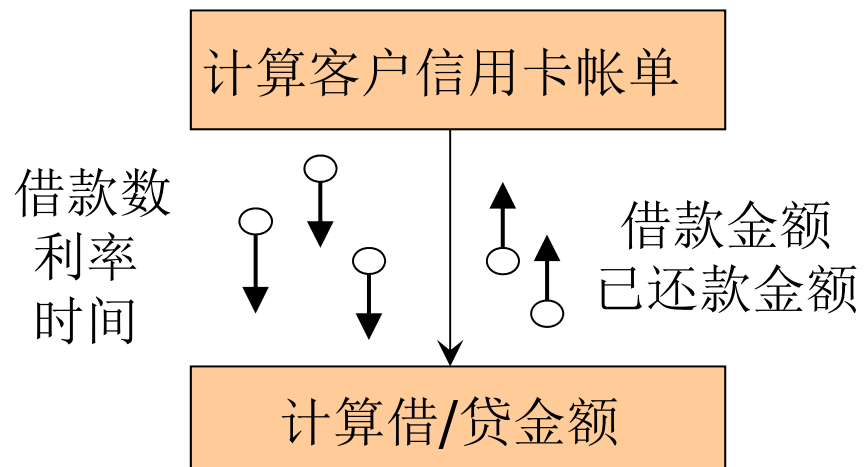
- 当一个操作调用另外一个操作时就会发生此种耦合
- 简单的、常见的、必要的
- 增加了系统的连通性



2. 数据耦合

■ 数据耦合

- 当操作需要传递较长或较复杂的数据参数时就会发生此种耦合
- 过多的参数为测试、维护带来困难



3. 印记耦合

- 印记耦合

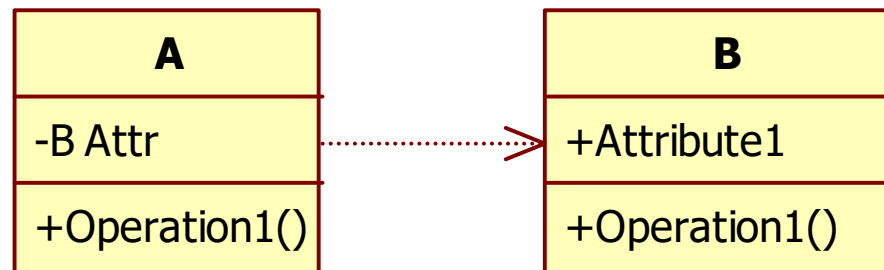
- 当类B被声明为类A的某一操作中的参数类型时发生的耦合
- 类B作为类A定义的一部分，类B的修改影响到类A



4. 类型使用耦合

■ 类型使用耦合

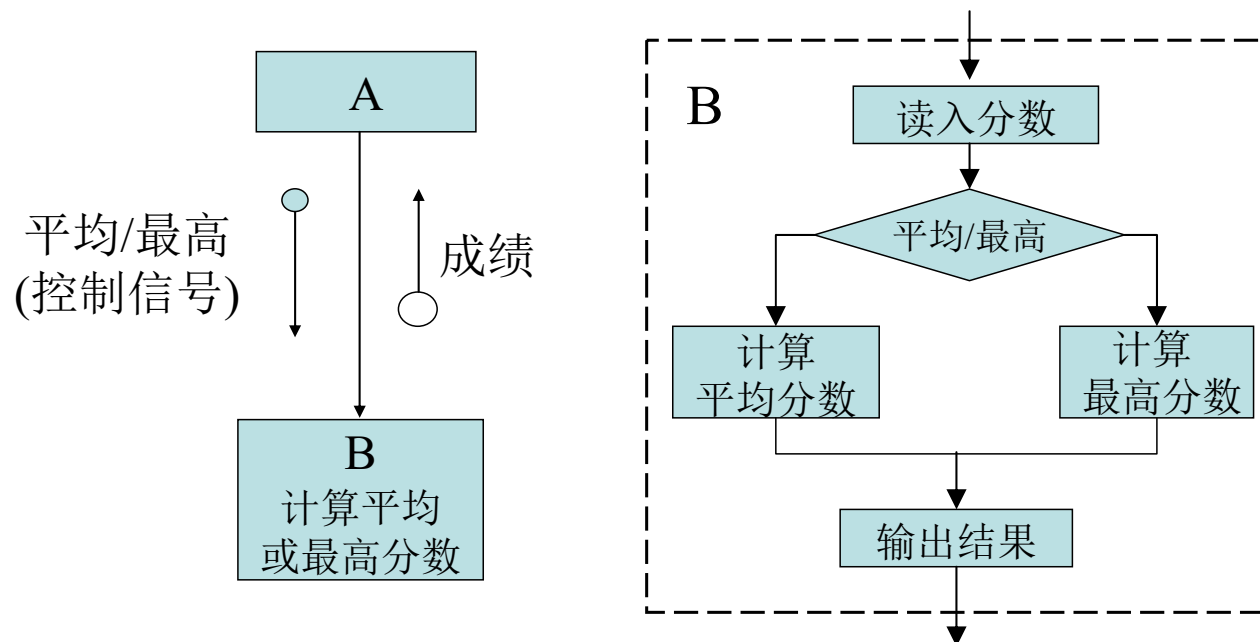
- 当构件A使用了在构件B中定义的一个数据类型时会发生此种耦合
- 一个类将某个变量声明为另一个类的类型
- 类型定义发生变化时，使用该定义的构件必须随之改变



5. 控制耦合

■ 控制耦合

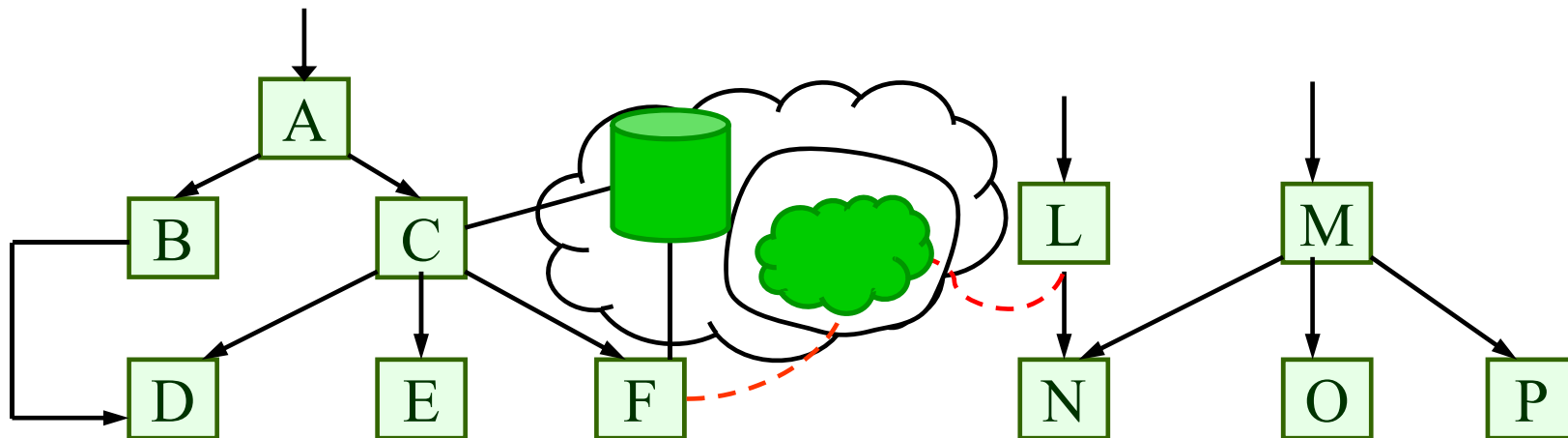
- 当操作A调用操作B，并向B传递了一个控制标记时，就会发生此种耦合。
- 控制标记会指引B中的逻辑流程
- B中的一个不相关变更可能导致A传递的控制标记失去意义，A就必须变更



6. 外部耦合

■ 外部耦合

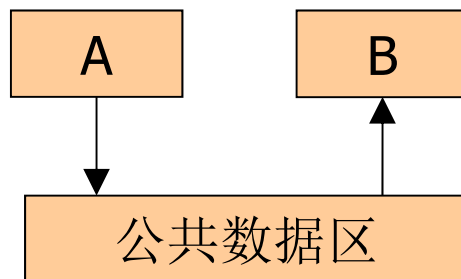
- 当一个构件和基础设施构件（如操作系统功能、数据库功能、网络通信功能等）进行通信和协作时会发生此种耦合



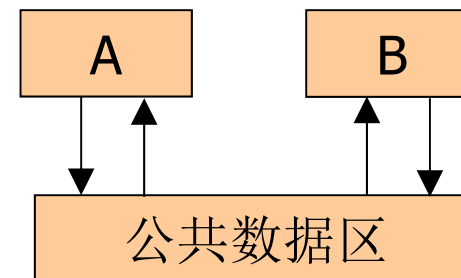
7. 共用耦合

■ 共用耦合

- 若一组模块都访问同一个公共数据环境，则它们之间的耦合就称为共用耦合。
 -
- 公共的数据环境可以是全局变量、共享的通信区、内存的公共覆盖区等。



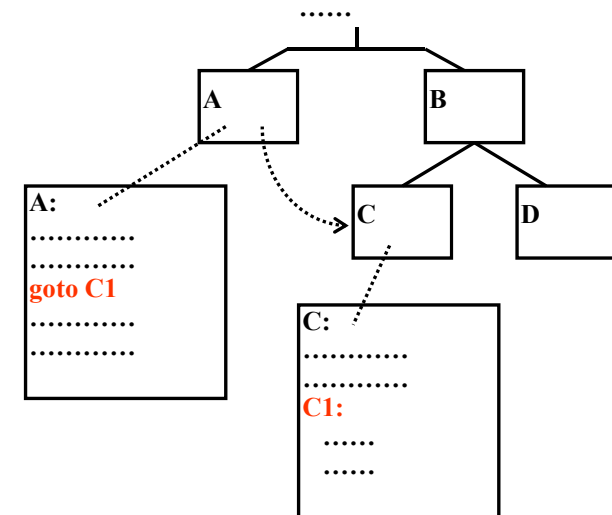
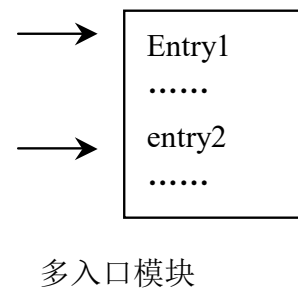
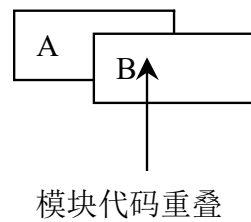
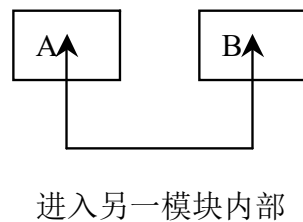
松散의公共耦合



紧密的公共耦合

8. 内容耦合

- 内容耦合：如果发生下列情形，两个模块之间就发生了内容耦合
 - 一个模块直接访问或修改另一个模块的内部数据;
 - 一个模块不通过正常入口转到另一模块内部;
 - 两个模块有一部分程序代码重迭(只可能出现在汇编语言中);
 - 一个模块有多个入口。
- 内容耦合违反了“信息隐藏”的原则。



关于耦合度的小结

- 耦合是影响软件复杂程度和设计质量的重要因素，应建立模块间耦合度尽可能松散的系统；
- 降低模块间耦合度：
 - 尽量使用数据、印记、类型使用耦合
 - 限制共用、外部耦合的范围
 - 坚决避免使用内容耦合

 - 尽量少的交换数据(few interfaces)
 - 尽量小的接口(small interfaces)
 - 尽量简单的数据交换(simple interfaces)
 - 尽量公开的接口(implicit interfaces)



2. SOLID: 类的设计原则



面向对象设计的原则

- 设计是一个反复、权衡、优化的过程
 - 迭代的设计过程
 - 考虑设计原则
 - 应用设计模式

缺乏良好设计系统的表现

- **僵化性 (Rigidity)** : 很难对系统进行改动, 因为每个改动都会迫使许多对系统其他部分的其他改动;
- **脆弱性 (Fragility)** : 对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题;
- **牢固性 (Immobility)** : 很难解开系统的纠结, 使之成为一些可在其他系统中重用的组件;
- **粘滞性 (Viscosity)** : 做正确的事情比做错误的事情要困难;
- **不必要的复杂性 (Needless Complexity)** : 设计中包含有不具任何直接好处的基础结构;
- **不必要的重复 (Needless Repetition)** : 设计中包含有重复的结构, 而该结构的本可以使用单一的抽象进行统一;
- **晦涩性 (Opacity)** : 很难阅读、理解。没有很好地表现出意图。

这些设计原则追求的NFR是什么？

- 不管是OO的五大设计原则，还是模块化设计原则，追求的目标是—

可维护性、可扩展性
(Maintainability, Extensibility)

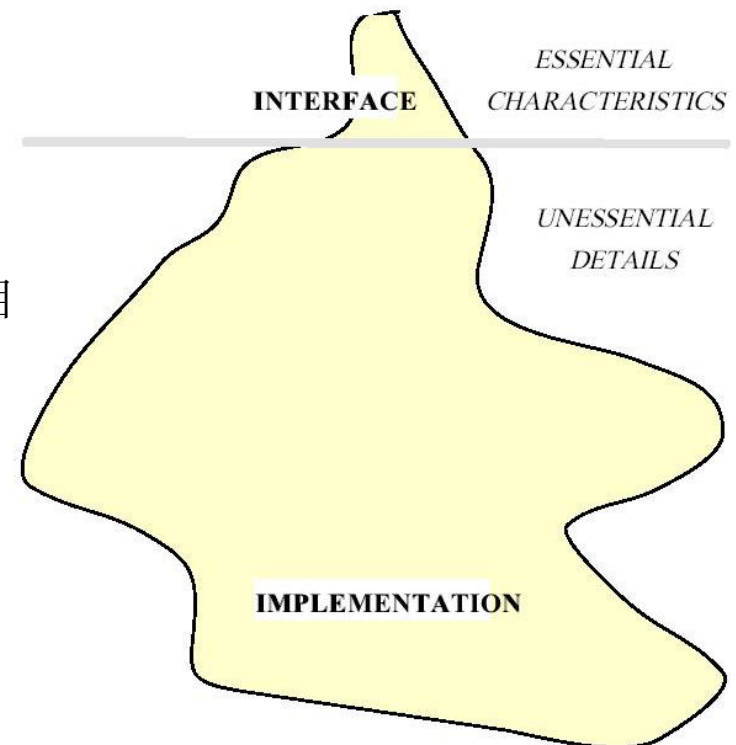
- 实现机制：降低耦合度(coupling)——隔离变化，让模块之间的依赖关系尽可能降低，当一方发生变化时，不会影响到另一方。
 - 关联(association): 两个类之间需要持久维持与对方的联系；
 - 依赖(dependency): 在某些特定时刻，两个类发生调用关系；
- 基本手段：封装(Encapsulation)、继承(Inheritance)、多态(Polymorphism)

封装(Encapsulation)

- 使对象形成两个部分：接口(可见)和实现(不可见)，将对象所声明的功能(行为)与内部实现(细节)分离

——信息隐藏(Information Hiding)

- “封装”的作用是什么？
 - 保护对象，避免用户误用；
 - 保护客户端，其实现过程的改变不会影响到相应客户端的改变。

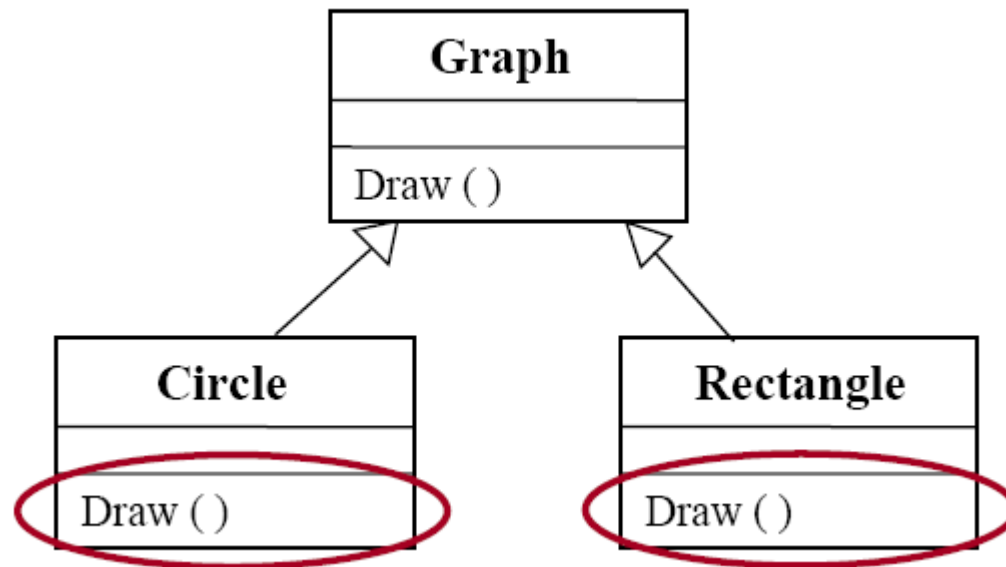


继承(Inheritance)

- **单一继承**：一个子类只有唯一的一个父类
- **多重继承**：一个子类有一个以上的父类
- **抽象类**：把一些类组织起来，提供一些公共的行为，但不能使用这个类的实例(即从该类中派生出具体的对象)，而仅仅使用其子类的实例。称不能建立实例的类为抽象类。
 - 抽象类中至少有一个方法被定义为“abstract”类型的。
 - “abstract”类型的方法：只有方法定义，没有方法的具体实现。

多态(Polymorphism)

- **多态性(Polymorphism):** 在父类中定义的属性或服务被子类继承后，可以具有不同的数据类型或表现出不同的行为。



OO的五大设计原则

- (SRP) The Single Responsibility Principle 单一责任原则
- (OCP) The Open-Closed Principle 开放封闭原则
- (LSP) The Liskov Substitution Principle Liskov替换原则
- (ISP) The Interface Segregation Principle 接口隔离原则
- (DIP) The Dependency Inversion Principle 依赖转置原则

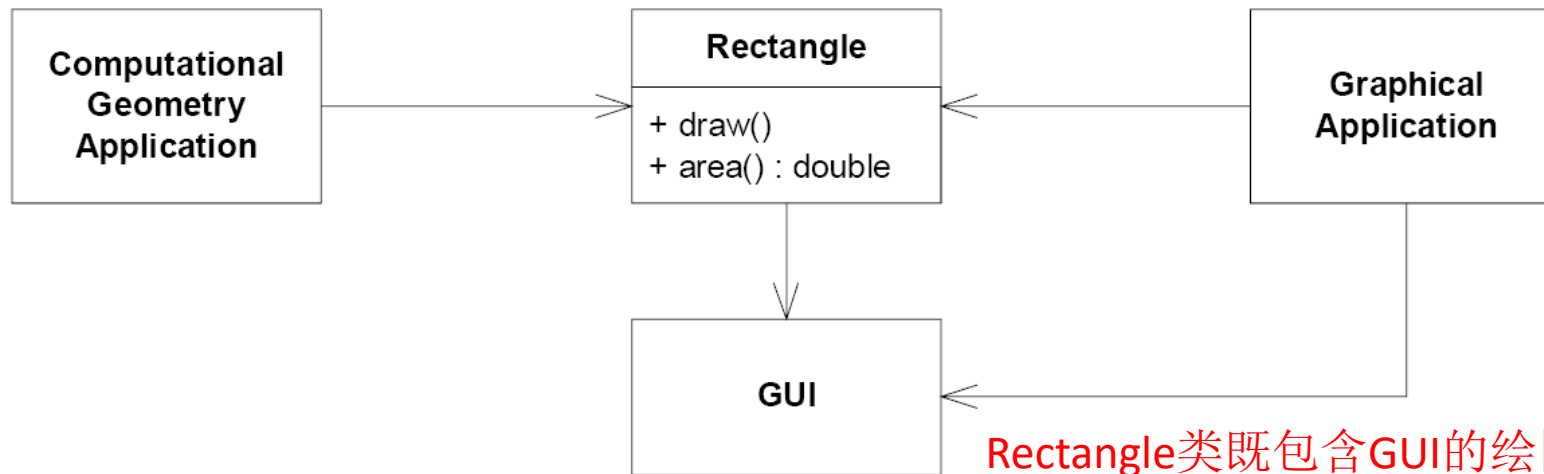
■ SOLID

- 参考资料《敏捷软件开发:原则、模式与实践》，Robert C. Martin，人民邮电出版社

1.(SRP) Single Responsibility Principle 单一责任原则

- 责任：“变化的原因” (a reason for change)
- 就一个类而言, 应该仅有一个引起它变化的原因(一个类应该有且仅有一个责任)。
- SRP的核心思想:
 - 不应有多于1个的原因使得一个类发生变化;
 - 一个类, 一个责任, 把“不变”与“变”分离开来。
 - 当一个类需要承担多个责任的时候, 就需要分解这个类。
- 如果一个类包含了多个责任, 那么将引起不良后果:
 - 功能之间就形成了关联, 改变其中一个功能, 有可能影响另一个功能;
 - 导致频繁的测试、重新配置、部署等。
- “身兼数职”的问题
- 类的细粒度——便于复用
- 类的单一性——利于稳定

SRP的一个反例

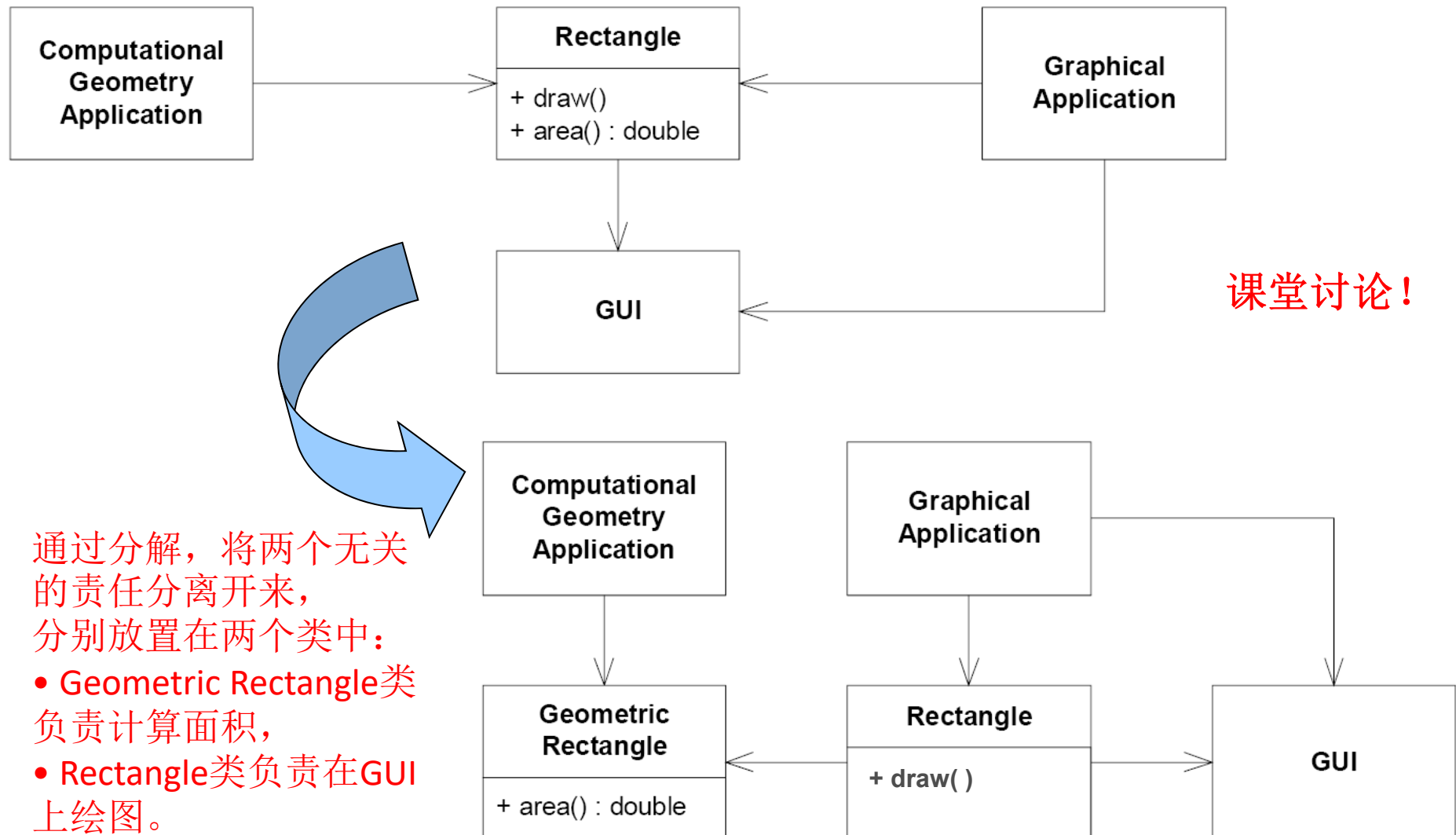


Rectangle类既包含GUI的绘图功能`draw()`--在GUI上显示矩形，又包括几何计算功能`area()`--计算面积。该设计违反了SRP。

问题1：计算应用ComputeAPP会包含同其无关的GUI代码，如：C++会把GUI代码链接进来，Java则要求GUI的Class文件必须被部署到目标平台上。

问题2：如果GraphicalAPP的改变导致Rectangle需要变化，则ComputerAPP也需要重新构建、测试和部署。

SRP的一个反例



通过分解，将两个无关的责任分离开来，分别放置在两个类中：

- **Geometric Rectangle**类负责计算面积，
- **Rectangle**类负责在GUI上绘图。

单一责任原则SRP

- 分离接口的责任

- 很多时候设计人员习惯于以组的形式考虑职责，往往会违反SRP

```
interface Modem
```

```
{
```

```
    //调制解调器的连接、挂断处理
```

```
    public void dial (string pno);
```

```
    public void hangup ();
```

```
    //发送、接收函数的数据通信
```

```
    public void send (char c);
```

```
    public void recv ();
```

```
}
```

该接口声明的4个函数都是调制解调器具有的功能。

有问题吗？

单一责任原则SRP

- 该接口程序有两个责任：

调制解调器的连接处理
两个数据通信的函数

要不要分离？

- 若程序变化导致**两个责任同时变化**，就不要分离。
- 若程序变化**仅影响一个责任**，导致调用和类要重新编译，增加了部署的次数，则这两个责任应分离。
- 注意：变化**实际发生**了应用SRP原则才有意义，若无征兆地刻意去应用单一责任，是不明智的。

集中式原则：实现效率高，变化时不易维护

单一责任原则：发生变化时易维护

单一责任原则SRP

■ SRP总结:

- SRP是所有原则中最简单，也是最难正确应用的一个。
- 我们经常会自然地把职责结合在一起，软件设计真正要做的许多内容，就是发现责任并把责任分离。
- 后续讨论的其它原则均同SRP相关。
- 再次强调：**变化实际发生**了应用SRP原则才有意义，若无征兆地刻意去应用单一职责，是不明智的。

2.(OCP) Open-Closed Principle 开放封闭原则

- 开闭原则(The Open-Close Principle,OCP): 软件实体（类、模块、函数等）应该是可以扩展的，但是不可修改的。
- OCP：一对矛盾观点的融合
- 对于扩展是开放的(Open for extension) 适应需求的变化
 - 当需求变更时，软件实体(类、模块、函数等)应该是可以扩展的，可以改变模块的功能，使其具有满足那些改变的新行为。
- 对于更改是封闭的(Closed for modification) 不影响已有的程序
 - 对模块进行扩展时，不必改动模块的源代码。

2.(OCP) Open-Closed Principle 开放封闭原则

- 那么当发生变化时，如何修改？
 - 通过增加代码来扩展功能，而不是修改已经存在的类的代码
 - 客户端代码不能被修改
- 解决方案：抽象、继承、多态机制
 - 把类中稳定的部分分离出来形成抽象类，具体的实现机制派生为子类，不同子类中的操作具有不同的实现形态，但其接口保持一致。
- OCP是OO的终极理想，很难实现，作为设计者要不断逼近之。
- **By Bertrand Meyer (1988)**

2.(OCP) Open-Closed Principle 开放封闭原则

- 一个既不开放也不封闭的例子



Client类使用Server类，若Client类使用另一个替代的Server类怎么办？

方案：要修改Client类中所有使用Server类的地方为新的Server类

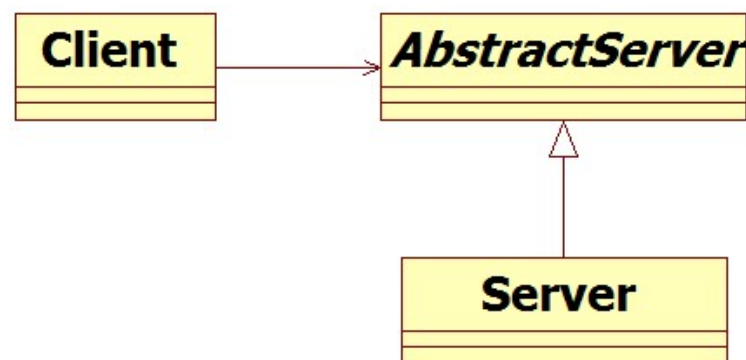
违背OCP原则

```
class Client
{
    Server server;
    void GetMessage()
    {
        server.Message();
    }
}
```

```
class Server
{
    void Message();
}
```

2.(OCP) Open-Closed Principle 开放封闭原则

- C++, Java等OOP语言具有**抽象类/接口**，可以创建出固定却能够描述一组任意个可能行为的抽象体，其任意行为可能出现在派生类中。

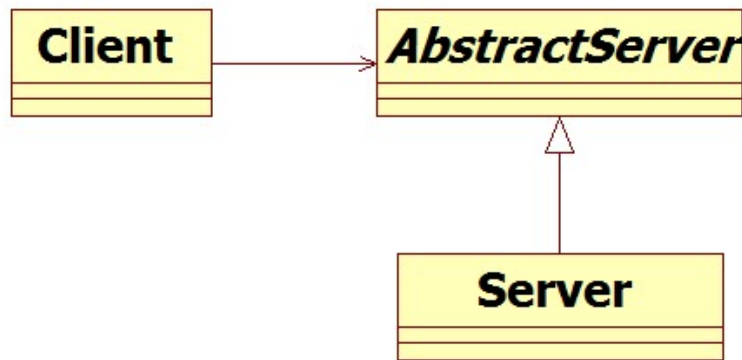


通过构造一个抽象的**Server**类：**AbstractServer**，该抽象类中包含针对所有类型的**Server**都通用的代码，从而实现了**对修改的封闭**；

当出现新的**Server**类型时，只需从该抽象类中派生出具体的子类**ConcreteServer**即可，从而支持了**对扩展的开放**。

“**接口+实现**”
“**抽象类+继承**”

2.(OCP) Open-Closed Principle 开放封闭原则



“接口+实现”
“抽象类+继承”

```
abstract class AbstractServer
{
    public void Message();
    //Other functions
}
```

```
class Server extends AbstractServer
{
    public void Message();
}
```

```
class Client
{
    AbstractServer ci;
    public void GetMessage()
    {
        ci.Message();
    }
    public void Client(ClientInterface paramCi)
    {
        ci=paramCi; //构造函数
    }
}
```


2.(OCP) Open-Closed Principle 开放封闭原则

- 主函数(或主控端)的处理

```
public static void Main()
{
    AbstractServer ci = new Server();
    //在上面如果有新的Server类只要替换Server()即可.

    Client client = new Client(ci);
    client.GetMessage();
}
```

- 上述方式中，**AbstractServer**的子类型可以以任何它们所选择的方式去实现这个抽象类，通过创建**AbstractServer**的子类型去扩展，而不需要改动**Client**。
- **AbstractServer**通过“**对接口编程**”和“**继承机制**”实现了**OCP**原则

2.(OCP) Open-Closed Principle 开放封闭原则

- 例: **Shape(形状)**应用程序, 其主要功能包括:
 - 在标准GUI上按照特定顺序绘制圆形和正方形。
 - 创建一个列表, 列表由按适当顺序排列的圆形和正方形组成, 程序遍历该列表, 依次绘制每个圆形和正方形。

shape.h

```
enum ShapeType { circle,square };
Struct Shape
{
    ShapeType itsType;
}
```

circle.h

Struct Circle

```
{
    ShapeType itsType;
    double itsRadius(半径);
    Point itsCenter(中心点);
};
```

square.h

Struct Square

```
{
    ShapeType itsType;
    double    itsSide; (边长)
    point     itsTopLeft; (左边的顶点)
};
```

DrawAllShapes.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:
                DrawCircle ((struct Circle*)s);
                break;
        }
    }
}

```

DrawAllShapes函数不符合**OCP**，它对于新的形状类型添加是不封闭的。每增加一个新的形状类型，都必须改变这个函数，要对新类型判断。

问题1： 实际应用中，所有类似于DrawAllShapes的函数中**switch**语句都要重复出现，但完成工作略有差异（绘图/拉伸/移动等）。增加一个新的形状，需要找到所有函数中的**switch**语句进行修改。

问题2： 不同形状依赖于**enum**声明。增加一个新成员都要重新编译、部署（DLL、共享库、二进制组件），一个简单行为导致连锁改动，是僵化的设计。

问题3： 复用时，需要额外附带很多无用的形状

2.(OCP) Open-Closed Principle 开放封闭原则

- 遵循OCP规则：定义一个抽象类**Shape**及抽象方法**Draw**。**Circle**和**Square**都从**Shape**类派生。

```
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;
```

```
//扩展时不需要改动DrawAllShapes方法
```

```
//可通过增加Shape类的派生类，扩展其行为.
```

```
Void DrawAllShapes (vector<Shape*>& list)
```

```
{
```

```
    vector<Shape*>::iterator i;
```

```
    for (i=list.begin () ; i != list.end(); i++)
```

```
        (*i)->Draw();
```

```
}
```

2.OCP总结

- 建立一个抽象类，使其他类依赖这个抽象类，这样它对于更改可以是封闭的。从这个抽象类派生的类(扩展类)的行为是开放的。

一般而言：无论模块是多么的封闭，都会存在无法对所有情况都适用的模型。

一般做法：预测或找出一个模块**易变化**的部分，甚至等到变化发生时再采取行动（遭受一次损失，但避免了再次的损失），构造**抽象类**，**隔离其变化**。

- **OCP是面向对象的核心**，遵守这个原则会使设计具有灵活性、可重用性、可维护性。但是，并不意味着对应用程序的每个部分都要进行抽象，过多的抽象会增加设计的复杂性。正确的做法是开发人员应该仅仅对程序中呈现出**频繁变化的那些部分**做出抽象。

拒绝不成熟的抽象和抽象本身一样重要。

3.(LSP) Liskov Substitution Principle (Liskov替换原则)

- **LSP: Subtypes must be substitutable for their base types. (子类型必须能够替换其基类型)**
 - 关注的是“操作”的可替换性
 - 派生类必须能够通过其基类的接口使用，客户端无需了解二者之间的差异。
 - 开闭原则OCP背后的主要机制是**抽象和多态**，**继承**是支持抽象和多态的关键机制之一。
 - LSP是在构建类的继承结构的过程中需要遵循的基本原则，LSP和OCP是相关联的，是OCP的基本保证，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。
- **LSP**可用作验证继承关系设计是否正确的准则，体现了多个子类在接口的一致性。
- 如果违背该原则，通常的做法是引入一个新的超类，将父子关系修改为兄弟关系。

3.Liskov替换原则

■ 例子：经典的“正方形与矩形”

- 在数学里，正方形当然是矩形。用OO的观点，正方形和矩形之间是“IS-A”的关系——这个关系正好是OO初级教程里说的确定继承关系的依据。因此，理所当然的，正方形类Square应该继承长方形类Rectangle。

```
public class Rectangle
```

```
{
```

```
    private long width;  
    private long height;
```

```
    public void setWidth(long width){  
        this.width=width;
```

```
    }
```

```
    public void setHeight(long height){  
        this.height=height;
```

```
    }
```

```
    public long getWidth(){  
        return width;
```

```
    }
```

```
    public long getHeight(){  
        return height;
```

```
    }
```

```
};
```

```
public class Square : Rectangle
```

```
{
```

```
    public void setWidth(long width){  
        this.width=width;  
        this.height=width;
```

```
    }
```

```
    public void setHeight(long height){  
        this.width=width;  
        this.height=width;
```

```
    }
```

```
};
```

```
public Rectangle IncreaseHeight(Rectangle r)
```

```
{
```

```
    while(r.getHeight()<r.getWidth())  
    {  
        r.setHeight(r.getHeight()+)
```

```
    }
```

```
    return r;
```

```
}
```

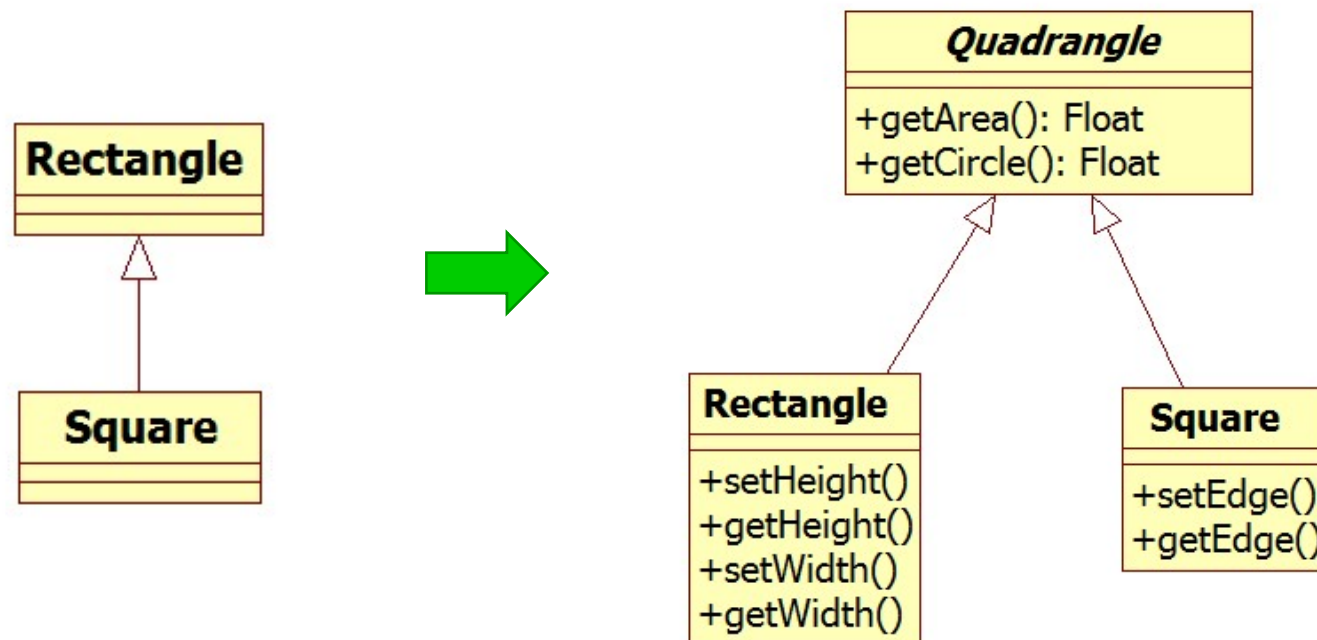
如果传递给IncreaseHeight的是一个Rectangle（长宽不同）的对象的话，没问题；如果为Square对象则会出现问题。

原因是这个继承结构违反了LSP原则：Square类对Height和Weight的处理和Rectangle逻辑不同，Rectangle单独改变Width和Height，而Square必须同时改变Width和Height。所以，Square和Rectangle之间的继承关系是不能成立的。

3.Liskov替换原则

- 例子：经典的“正方形与矩形”

- 定义一个抽象类Quadrangle(四边形)，正方形和矩形均从其继承，既保证了继承四边形的共有属性和行为，也保证了其独特性。



3.Liskov替换原则

- 有效性的问题

- LSP的一个重要结论：一个模型，如果孤立地看，并不具有真正意义上的有效性，有效性只能通过它的客户程序来表现。如：单独看“正方形-矩形”模型，最初的设计是正确的，但是站在程序员（用户程序）角度看，是有问题的。

- 重新认识继承的判断方法“IS-A”

- “IS-A”不仅仅指属性，也要满足行为方式的要求。
- 行为方式是可以进行合理假设的，是客户程序所依赖的。

3.Liskov替换原则

■ 一些经验原则

- 尽量从抽象类继承，不要从实体类继承。因为实体类会具有和特定实体相关的方法，这些方法在子类中可能不会有用，甚至产生问题。
- 使用**契约式编程方法**DBC（Design by Contract），把类和其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。在父类里定义好子类需要实现的功能，而子类只要实现这些功能即可。
 - 在重新声明派生类的例程时，只能使用相等或者更弱的前置条件来替换原始的前置条件，只能使用相等或者更强的后置条件来替换原始的后置条件。即，派生类必需接收基类可以接收的一切，同时基类的用户不应该被派生类的输出扰乱。
- 所有派生类的行为功能必须和客户程序对其基类所期望的保持一致，这是OO在继承和重写（override）时的基本要求。

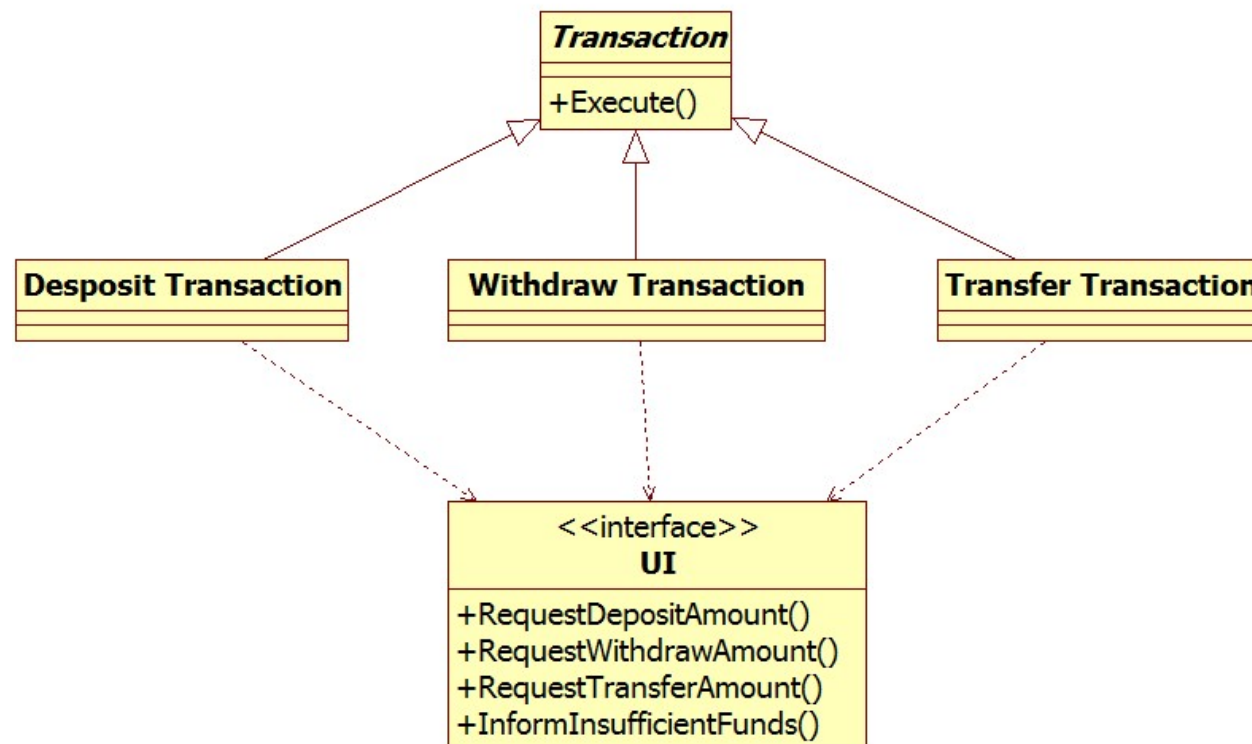
4.(ISP) Interface Segregation Principle 接口隔离原则

- **ISP:** 客户端不应依赖于它们不需要的方法。
- 同一个类可以同时实现多个接口，站在调用者的角度，不同的接口代表不同的关注点，不同的职责，甚至是不同的角色。
- **LSP的思想:**
 - 把胖接口分解为多个小的接口
 - 不同接口向不同客户端提供不同的服务，多个用户专用接口优于一个通用接口
 - 客户端只访问自己所需要的端口
- **所体现的思想:**
 - 接口若要稳定，就应承担较少责任(SRP)
 - 客户端应依赖稳定的接口(DIP)。

4接口隔离原则ISP

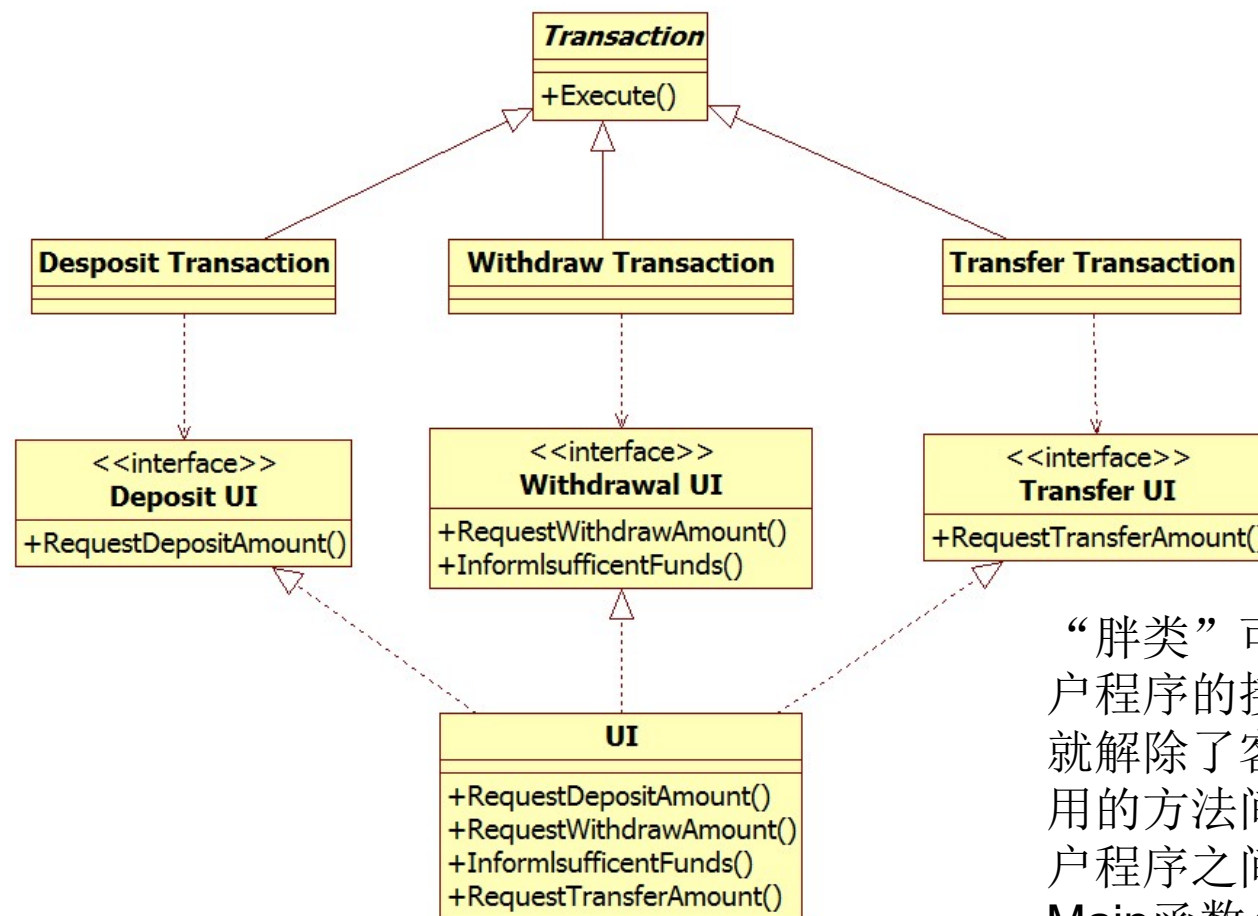
■ 例子：ATM显示的问题

- ATM中的存款、取款、转帐类都依赖UI接口（都只使用了接口中的部分功能），增加新操作时（如：缴费），需要修改接口，会影响到其它依赖于接口的类。



4.接口隔离原则ISP

■ 解决方案:



“胖类”可以继承所有特定于客户程序的接口，并实现它们。这就解除了客户程序和它们没有调用的方法间的依赖关系，并使客户程序之间互不依赖。（只有Main函数会调用类UI）

4.接口隔离原则ISP

■ 总结:

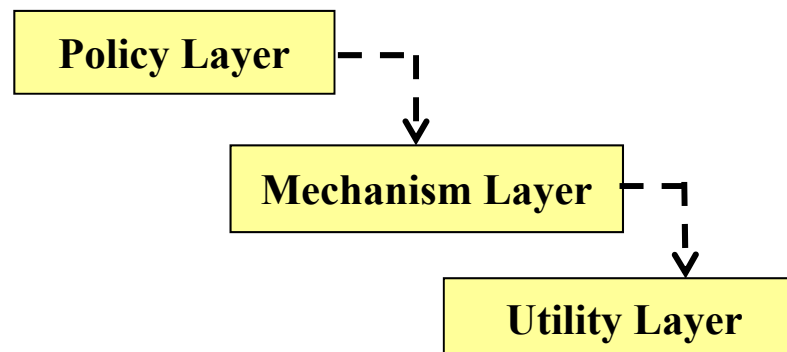
- 胖类(fat class)导致客户程序间不正常且有害的耦合关系，为解除客户程序和它们没有调用的方法间的依赖关系，并使客户程序间互不依赖考虑如下几点：
 - 1) 把胖类接口分解为多个特定于客户程序的接口，可以实现客户程序仅仅依赖于它们实际调用的方法；
 - 2) 每个特定于客户程序的接口仅仅声明它的特定客户或者客户组调用的那些方法；
 - 3) 胖类可以继承所有特定于客户程序的接口，并实现它们。这就解除了客户程序和它们没有调用的方法间的依赖关系，并使客户程序之间互不依赖。

5.(DIP) Dependency Inversion Principle 依赖倒置原则

- Abstractions should not depend upon details (抽象的模块不应依赖于具体的模块)
- Details should depend upon abstractions (具体应依赖于抽象)
- 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) 抽象不应该依赖于实现细节。实现细节应该依赖于抽象。
- 实现手段：面向接口编程
 - 不管是在高层模块和低层模块之间，还是在客户端模块和服务端模块之间，都应依赖于接口，而不是具体实现。
- 核心思想：“抽象”，把直接依赖关系断开，

5. 依赖倒置原则DIP

- 典型层次化(策略-机制-效用)方案：高层依赖于低层，并且这种依赖关系是传递的。
- 典型层次化设计方案的问题：
 - **结构化分析和设计**：创建一些高层依赖于低层的模块，策略（policy）依赖于细节的软件结构，为了便于描述高层模块如何调用低层模块。
 - **问题1：低层模块的变化会影响到高层**。高层模块包含了策略选择和业务模型，应该是高层模块去影响低层的细节实现模块，高层模块应优先并独立于包含实现细节的模块；
 - **问题2：无法做到高层模块的重用**。低层模块易于通过如程序库、函数库等形式进行重用，如果高层依赖于低层，则很难被重用。框架（framework）设计（体系结构级重用）的核心原则是要求高层独立于低层。

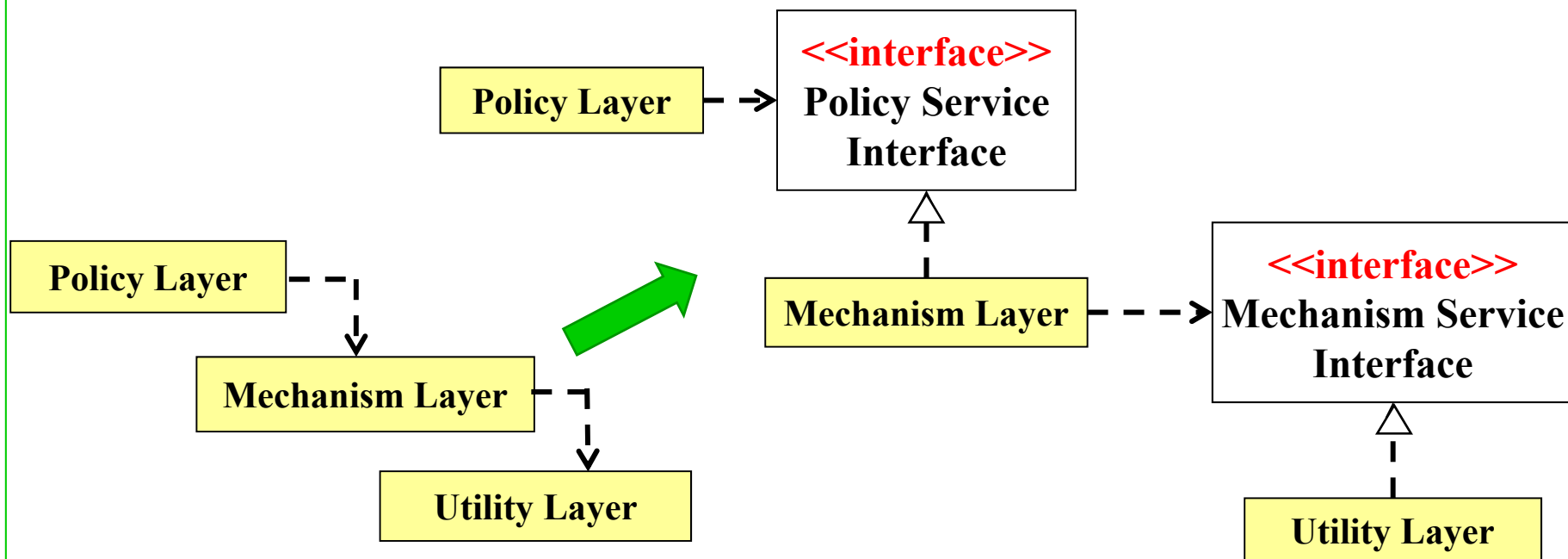


5. 依赖倒置原则DIP

- “...所有结构良好的面向对象架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

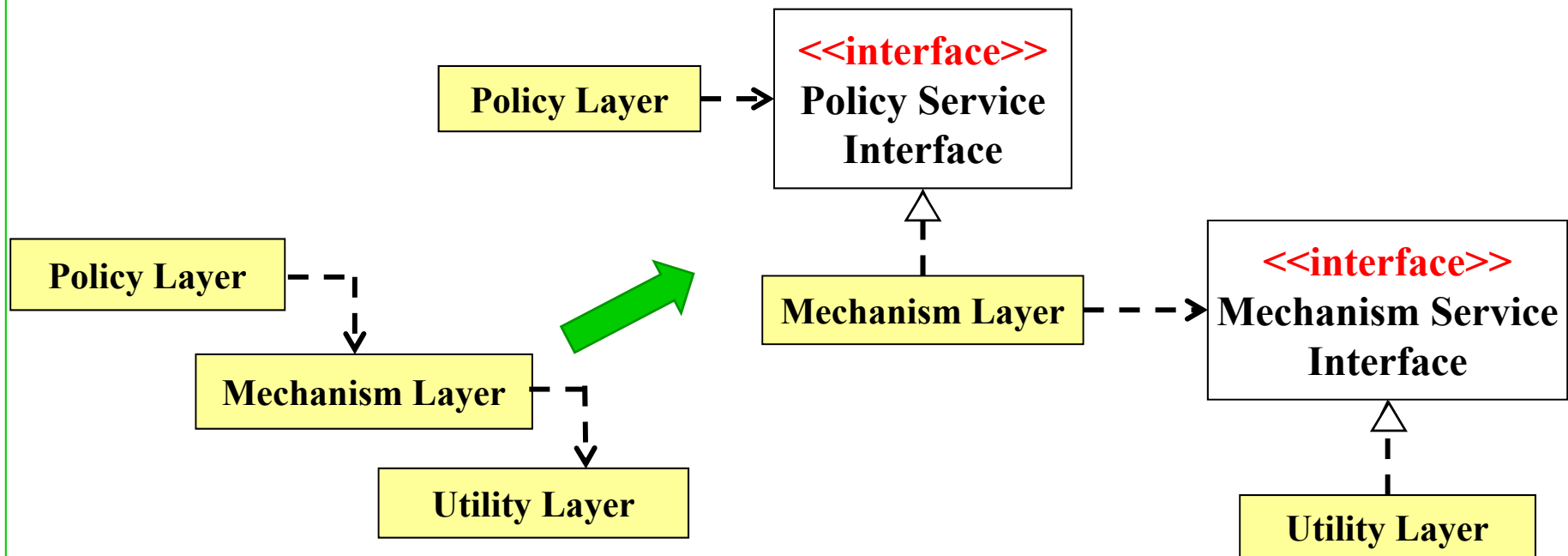
- 分层设计是必需的和提倡的，但是不应使高层依赖于低层。
- 更为合适的模型：每个较高层次都为它的服务声明一个抽象接口，较低层次实现抽象接口，每个高层类都通过该抽象接口使用下一层。



5. 依赖倒置原则DIP

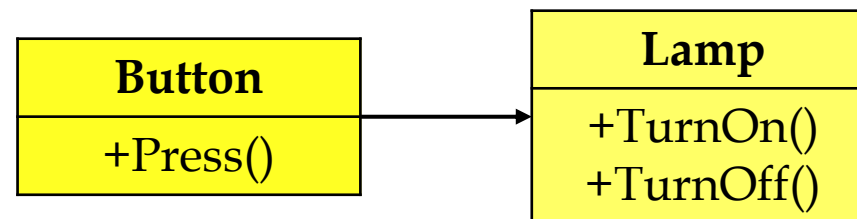
■ 优点:

- 增加各层模块的独立性，高层不依赖于低层，低层反而依赖于高层中声明的抽象服务接口（**依赖倒置**），低层模块的改动不会影响到高层模块。
- PolicyLayer（高层）可以在定义了符合PolicyServiceInterface的任何上下文中重用。



5. 依赖倒置原则DIP

- 一个例子：Button对象控制Lamp对象的一个模型
 - Button对象感知外部环境变化，收到Press消息，判断是否被用户按下。它并不关心是通过何种机制去感知的。
 - Lamp对象会影响外部环境



Button.Java 代码

```
Public class Button
{
    private Lamp itsLamp;
    public void Press()
    {
        if ( /* some condition */ )
            itsLamp.turnOn();
    }
}
```

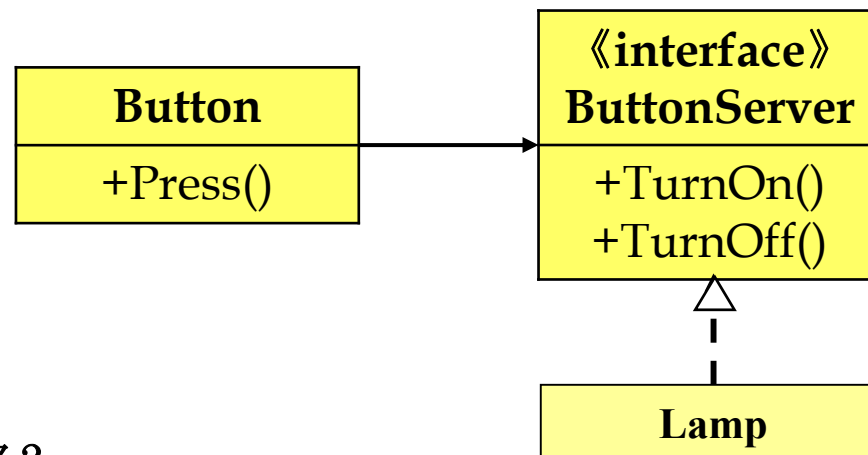
问题：高层策略依赖于低层模块，抽象依赖于具体细节。

1. Lamp改变时，会影响到Button；
2. 重用Button控制其他设备的开关时，不可行。（控制电视、冰箱等）

5. 依赖倒置原则DIP

- 一个解决方案：

- 该设计可以使Button控制任何愿意实现ButtonServer接口的设备。

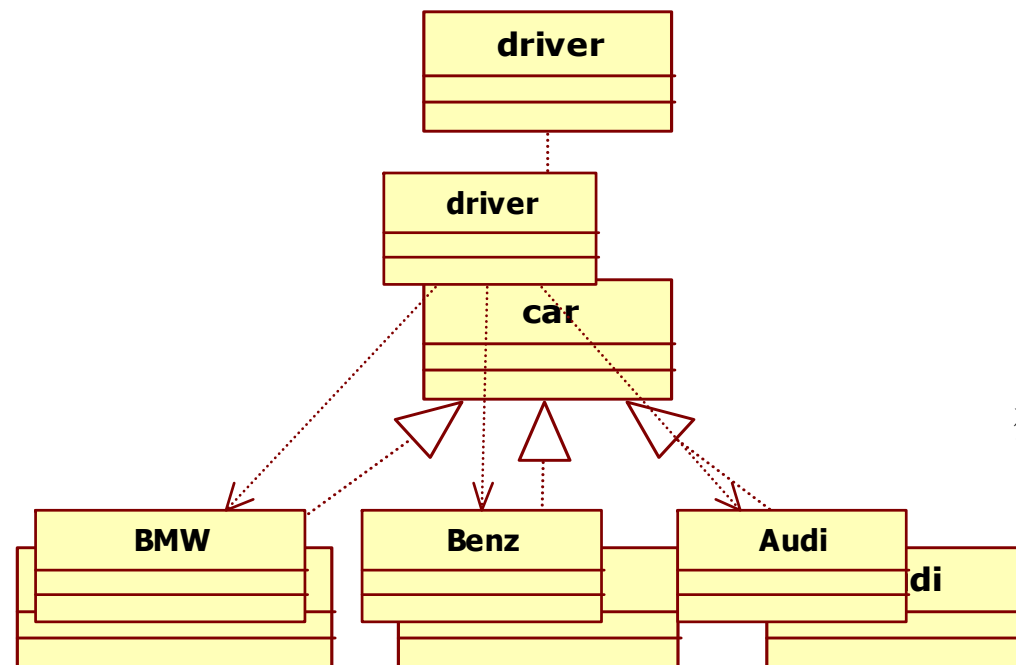


- 什么是高层策略？

- 应用背后的抽象，是那些不随具体细节的改变而改变的真理。
- 例子中背后的抽象是：检测用户的开/关指令并将指令传给目标对象。何种机制检测？目标对象是什么？怎么工作？均无关紧要。

5. 依赖倒置原则DIP

■ 例子：



想开QQ怎么办？

5. 依赖倒置原则DIP

■ 另外一个例子：熔炉控制示例

- 一个控制(Regulate)熔炉调节器软件，从I/O通道中读取当前的温度，并通过向另一个通道发送命令来指示熔炉的开或关。

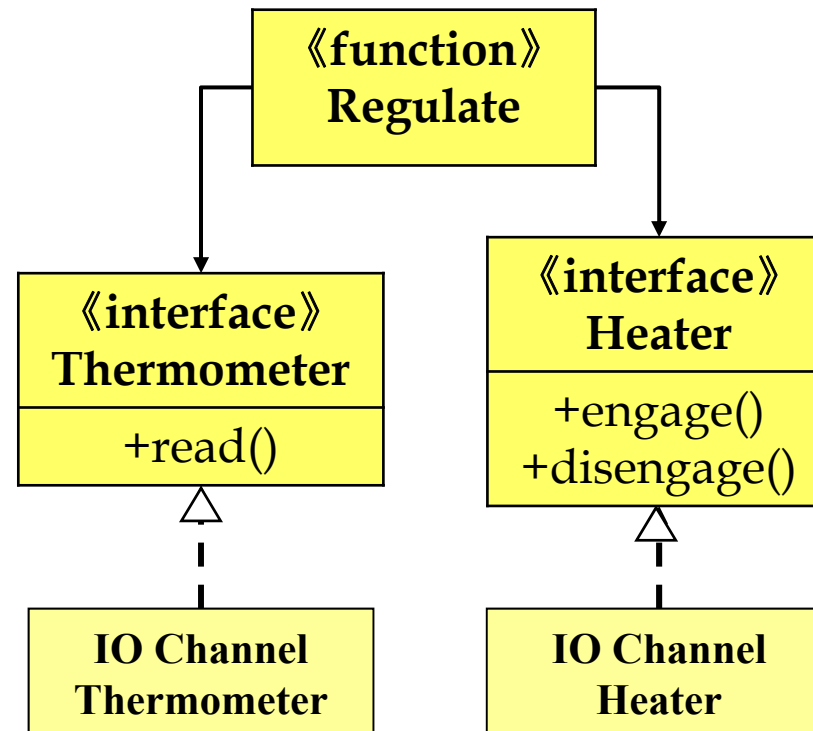
```
#define TERMOMETER 0x86 //炉子两个通道,从温度计上读取温度
#define FURNACE    0x87 //熔炉
#define ENGAGE     1    //启动
#define DISENGAGE  0    //停止
Void Regulate(double minTemp, double maxTemp)
{
    for (;;) {
        while (in(TERMOMETER) > minTemp)
            wait(1);
        out (FURNACE, ENGAGE); //熔炉启动
        while (in(TERMOMETER) < maxTemp)
            wait(1);
        out (FURNACE, DISENGAGE); //熔炉停止
    }
}
```

算法高层意图是清楚的，但代码包含了许多低层细节，不能重用

5. 依赖倒置原则DIP

■ 修改后的方案:

- 调节器函数Regulate接受两个接口参数, 温度计(Thermometer)接口可以读取, 加热器(Heater)接口可以启动和停止。
- 此时, 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, 具有很好的可重用性。成为了通用的调节器。



5. 依赖倒置原则DIP

■ 修改后的方案:

- 此时，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，具有很好的可重用性。成为了通用的调节器。

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


5.依赖倒置原则DIP

- DIP可概括为一个启发式规则：“依赖于抽象”。它建议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接口。
- 具体而言（理想状况下）：
 - 任何变量都不应该持有一个指向具体类的指针或者引用；
 - 任何类都不应当从具体类中派生；
 - 任何方法都不应该覆写它的任何基类中已经实现了的方法。
- 上述启发式规则要灵活运用：
 - 完全依据上述启发式规则会增加设计的复杂度
 - 该启发式规则适用于那些不稳定的具体类。
 - 若一个具体类(如描述字符串的类String)是稳定的，也不会创建其他类似的派生类，直接依赖它不会造成损害。

5.依赖倒置原则DIP

■ 总结:

- 过程化程序设计所创建出来的依赖关系结构，高层策略是依赖于低层细节的，会使高层受到细节改变的影响。
- 程序的依赖关系没倒置就是过程化的设计，程序的依赖关系倒置了，就是面向对象的设计。
- 正确应用依赖关系的倒置对于创建可重用的框架是必须的。
- 对于构建在变化方面富有弹性的代码也是非常重要的，抽象和细节分离，代码易维护。

小结：OO设计的两大武器

- **抽象(abstraction):** 模块之间通过抽象隔离开来，将稳定部分和容易变化部分分开
 - LSP: 对外界看来，父类和子类是“一样”的；
 - DIP: 对接口编程，而不是对实现编程，通过抽象接口隔离变化；
 - OCP: 当需要变化时，通过扩展隐藏在接口之后的子类加以完成，而不要修改接口本身。
- **分离(Separation): Keep It Simple, Stupid (KISS)**
 - SRP: 按责任将大类拆分为多个小类，每个类完成单一职责，规避变化，提高复用度；
 - ISP: 将接口拆分为多个小接口，规避不必要的耦合。
- **归纳起来:** 让类保持责任单一、接口稳定。



3. 包的设计原则：模块化

Principles of Package Cohesion (包聚合设计原则)

- **(REP) The Reuse/Release Equivalency Principle**

复用/发布等价原则：复用的粒度应等价于发布的粒度

- **(CCP) The Common Closure Principle**

共同封闭原则：一个包中的所有类针对同一种变化是封闭的；一个包的变化将会影响包里所有的类，而不会影响到其他的包；如果两个类紧密耦合在一起，即二者总是同时发生变化，那么它们就应属于同一个包。

- **(CRP) The Common Reuse Principle**

共同复用原则：一个包里的所有类应被一起复用；如果复用了其中一个类，那么就应复用所有的类。

Principles of Package Coupling (包耦合设计原则)

- (ADP) The Acyclic Dependencies Principle

无圈依赖原则：不允许在包依赖图中出现任何圈/回路；容易进行测试、维护与理解；否则将很难预测该包的变化将会如何影响其他包。

- (SDP) The Stable Dependencies Principle

稳定依赖原则：包之间的依赖关系只能指向稳定的方向；被依赖者应更稳定于依赖者；如果不稳定的包却被很多其他包依赖，会导致潜在的问题。

- (SAP) The Stable Abstraction Principle

稳定抽象原则：一个包是稳定的，那么它就应该尽可能抽象；一个完全稳定的包中只应包含抽象类。

SDP： 依赖应指向稳定的方向， **SAP：** 稳定性隐含着抽象



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

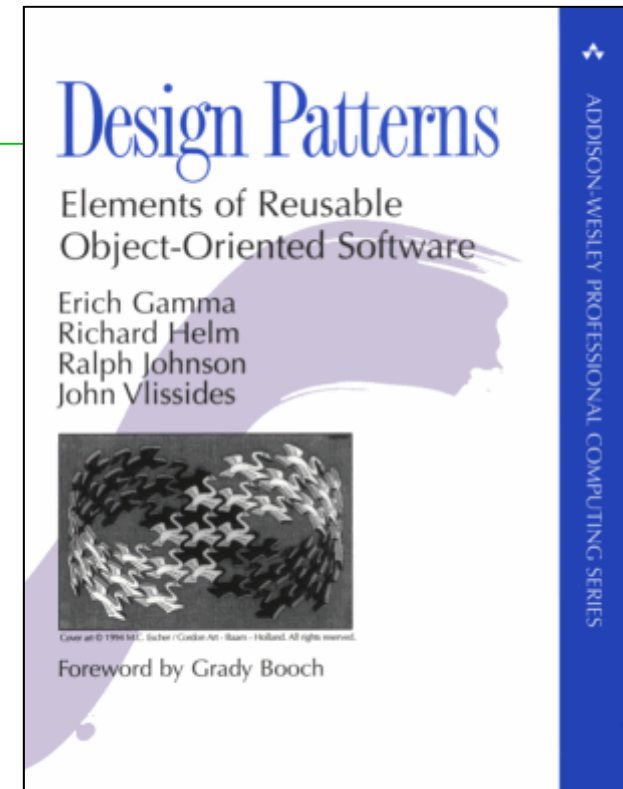
软件工程

4. 设计模式



OO中的设计模式

- 20多种设计模式
 - Abstract Server (抽象服务器)
 - Adapter (适配器)
 - Observer (观察者)
 - Bridge (桥接器)
 - Abstract Factory (抽象工厂)
 - ...
- 作者: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GoF, Gang of Four, 四人帮)
- 书名: Design Patterns: Elements of Reusable Object-Oriented Software





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

結束

2017年11月6日