Analysis Versus Design

PIM → PSM

- Analysis

- Focus on understanding the problem
- Idealized design
- Behavior
- System structure
- Functional requirements
- A small model

- Design

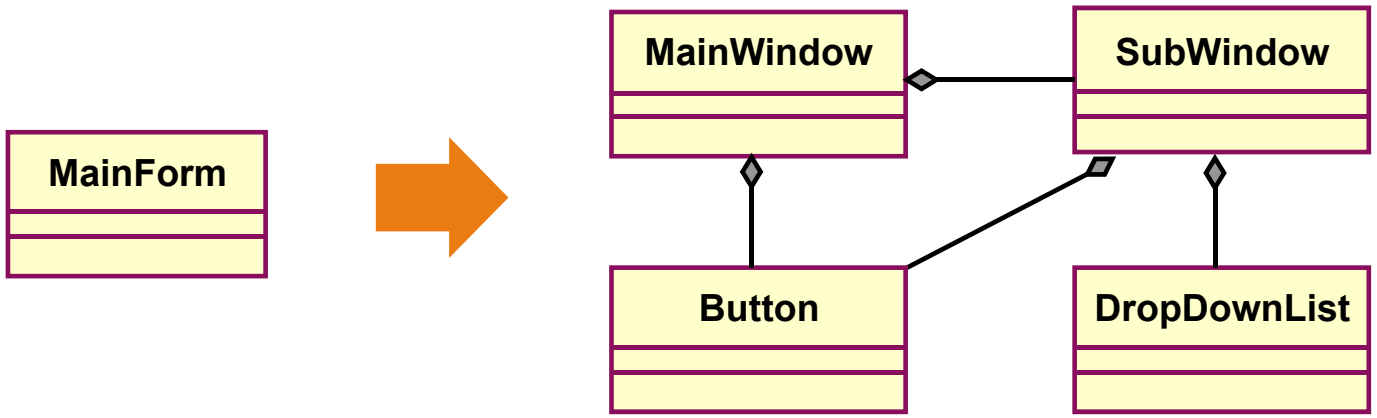
- Focus on understanding the solution
- Operations and attributes
- Performance
- Close to real code
- Object lifecycles
- Nonfunctional requirements
- A large model

面向对象设计Object Oriented Design

- 在面向对象分析与架构设计的基础上，针对特定平台，进行不断的细化与优化.
  - 针对特定平台（编程语言、操作系统、数据库、框架、中间件、库等），进行specialization
  - 应用面向对象设计的原则与模式进行优化

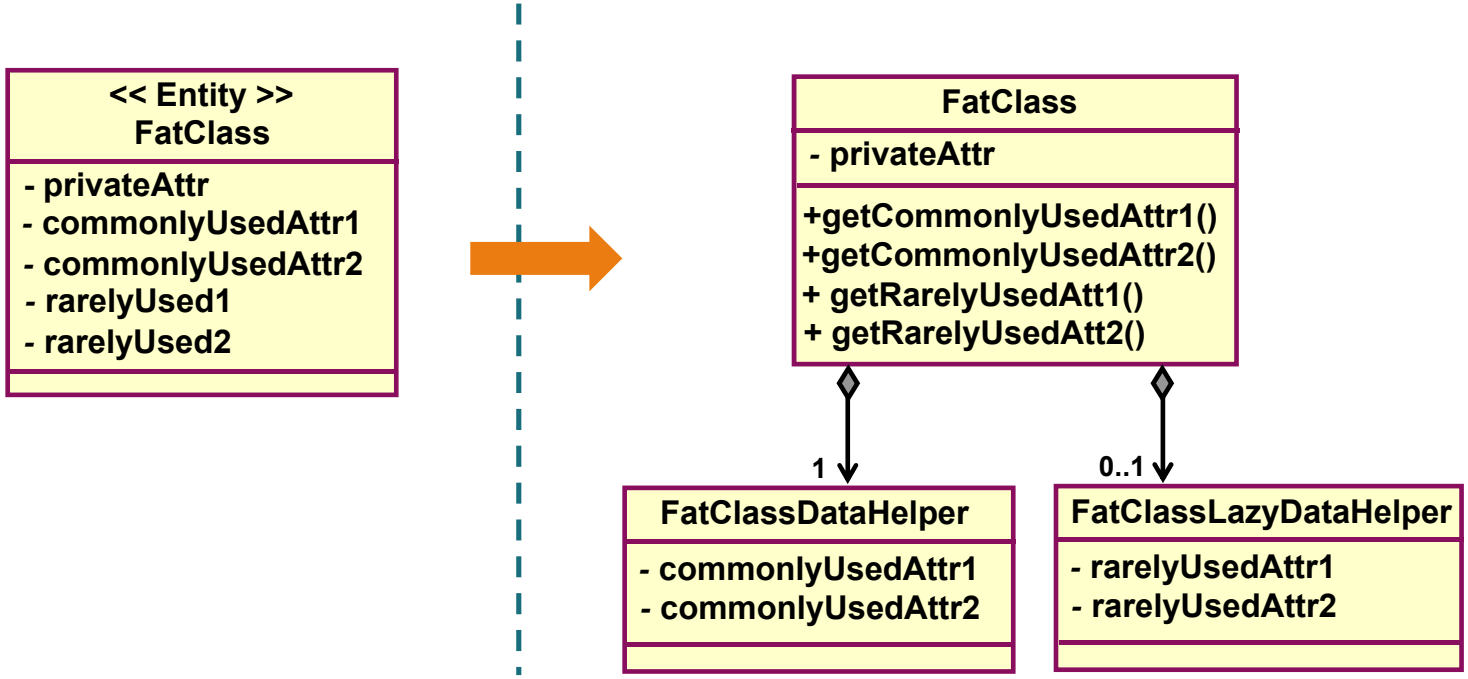
细化示例：细化UI类

- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?



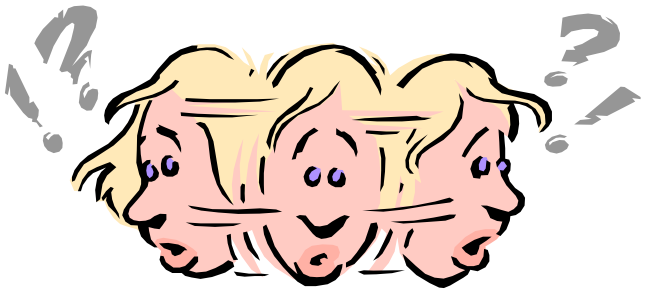
优化示例：优化Entity类

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring
- See the Identify Persistent Classes step

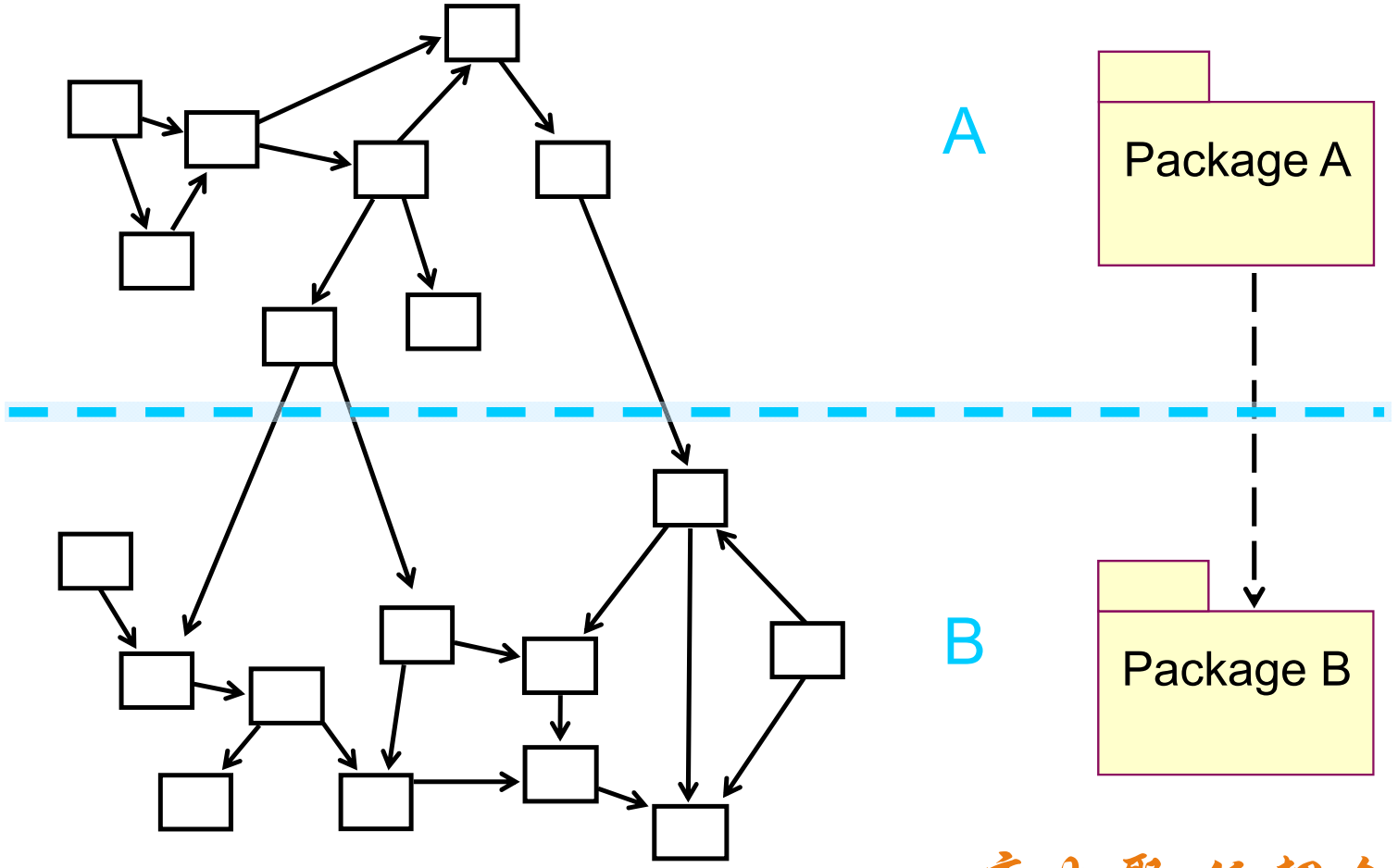


优化示例：优化Control类

- What happens to Control Classes?
  - Are they really needed?
  - Should they be split?
- How do you decide?
  - Complexity
  - Change probability
  - Distribution and performance
  - Transaction management

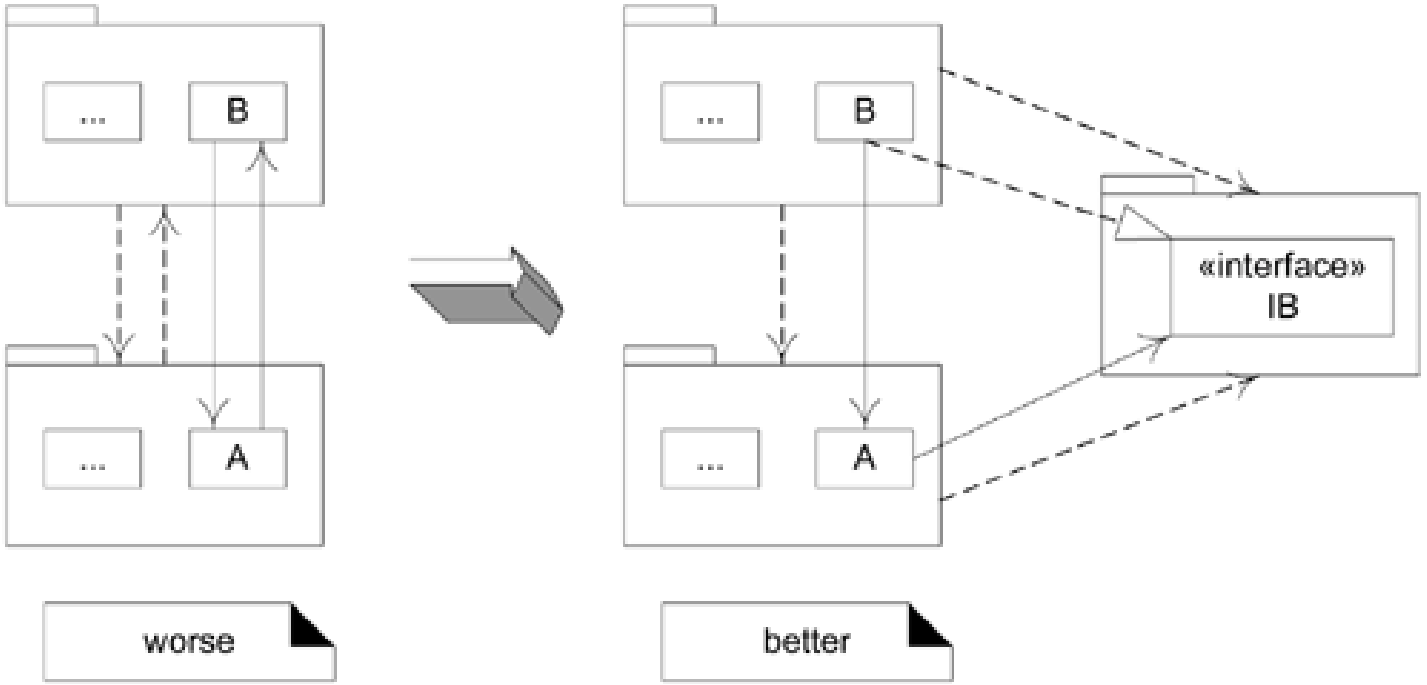


优化示例：Partitioning



高内聚 低耦合

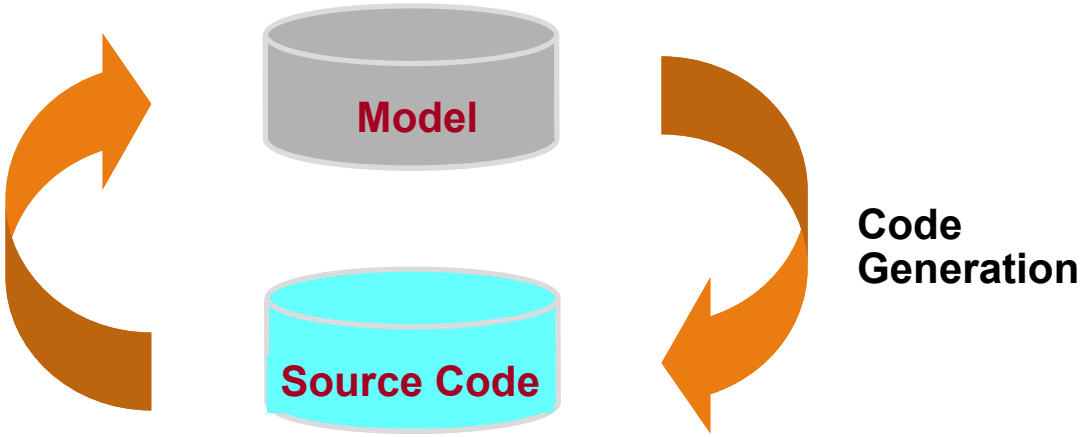
优化示例：无环依赖



低耦合

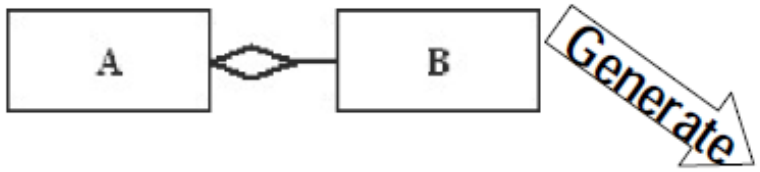
From Design to Implementation

- 正向工程Forward engineering
- 逆向工程Reverse engineering
- 双向工程Round-trip engineering



Forward Engineering

- Forward engineering means the generation of code from UML diagrams
- Many of the tools can only do the static models:
  - They can generate class diagrams from code, but can't generate interaction diagrams.
  - For forward engineering, they can generate the basic (e.g., Java) class definition from a class diagram, but not the method bodies from interaction diagrams.
- Example:




```
public class A {
    private B instancesOfB[];

    public A()
    {
    }
}
```

```
public class B {

    public B()
    {
    }
}
```

Reverse Engineering

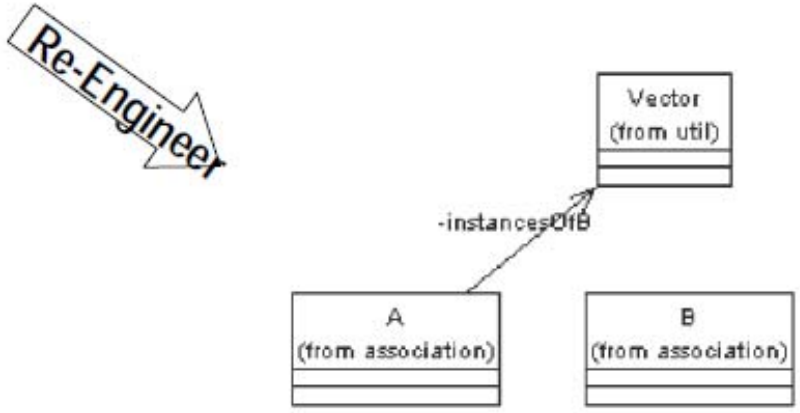
- Reverse engineering means generation of UML diagrams from code
- Example:

```
public class A {
    private B instancesOfB[];

    public A()
    {
    }
}
```

```
public class B {

    public B()
    {
    }
}
```



Round-Trip Engineering

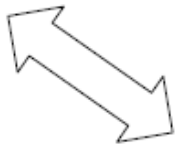
- Round-trip engineering closes the loop
  - the tool supports generation in either direction and can synchronize between UML diagrams and code, ideally automatically and immediately as either is changed.
- Example:

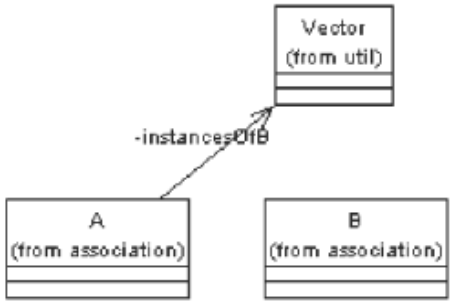
```
public class A {
    private B instancesOfB[];

    public A()
    {
    }
}
```

```
public class B {

    public B()
    {
    }
}
```





大 纲

01-面向对象设计的方法

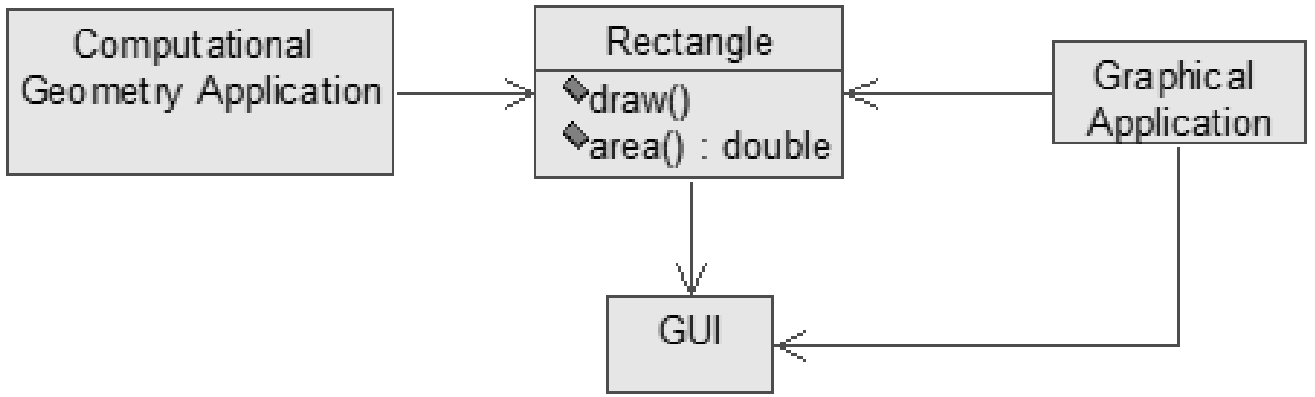
02-面向对象设计的原则

1. 单一职责原则SRP
2. 里氏替换原则LSP
3. 依赖倒置原则DIP
4. 接口隔离原则ISP
5. 开-闭原则OCP
6. 组合/聚合复用原则CARP

03-GoF 设计模式

1. SRP: The Single – Responsibility Principle

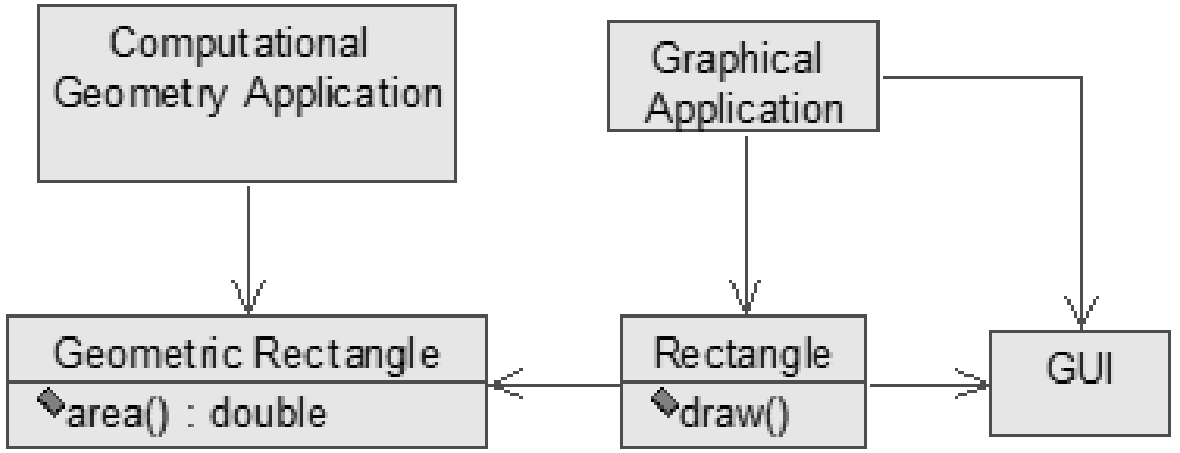
- 单一职责原则
  - 就一个类而言，应该只有一个导致其变更的原因
  - 在SRP中，“职责”是“变更的原因”
- 举例



  - Rectangle 类有两个职责，违反了SRP
    - 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

SRP: The Single – Responsibility Principle

- Separating Coupled Responsibilities



√

高内聚
低耦合

- 多职责将导致脆弱性的臭味：一个类如果承担的职责过多，一个职责的变化可能会削弱或者抑止这个类完成其它职责的能力。
- 一个类承担的职责越多，它被复用的可能性越小。

2. LSP: The Liskov Substitution Principle

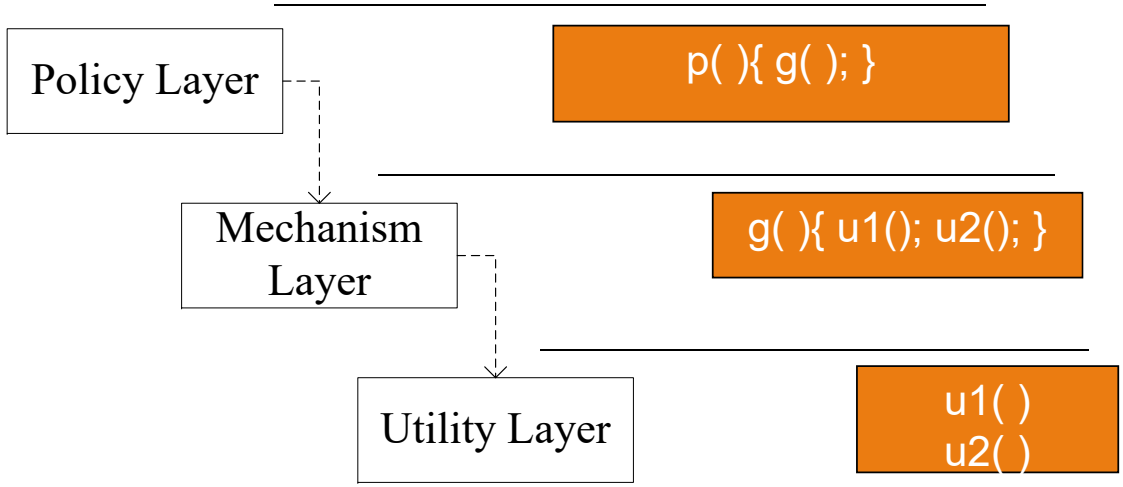
- 里氏替换原则
  - 子类型（Subtype）必须能够替换它们的基类型(Basetype)
  - Barbara Liskov对原则的陈述： 若对每个类型S的对象o1,都存在一个类型T的对象o2，使得在所有针对T编写的程序P中，用o1替换o2后，程序P的行为功能不变，则S是T的子类型。
- 举例:
  - 在现实世界中，企鹅和鸟之间是“is-a” 关系。如果我们把企鹅设计成鸟的子类，就违反了LSP原则，因此鸟会飞，但企鹅不会。
- 里氏替换原则为良好的继承定义了一个规范

3. DIP: The Dependency Inversion Principle

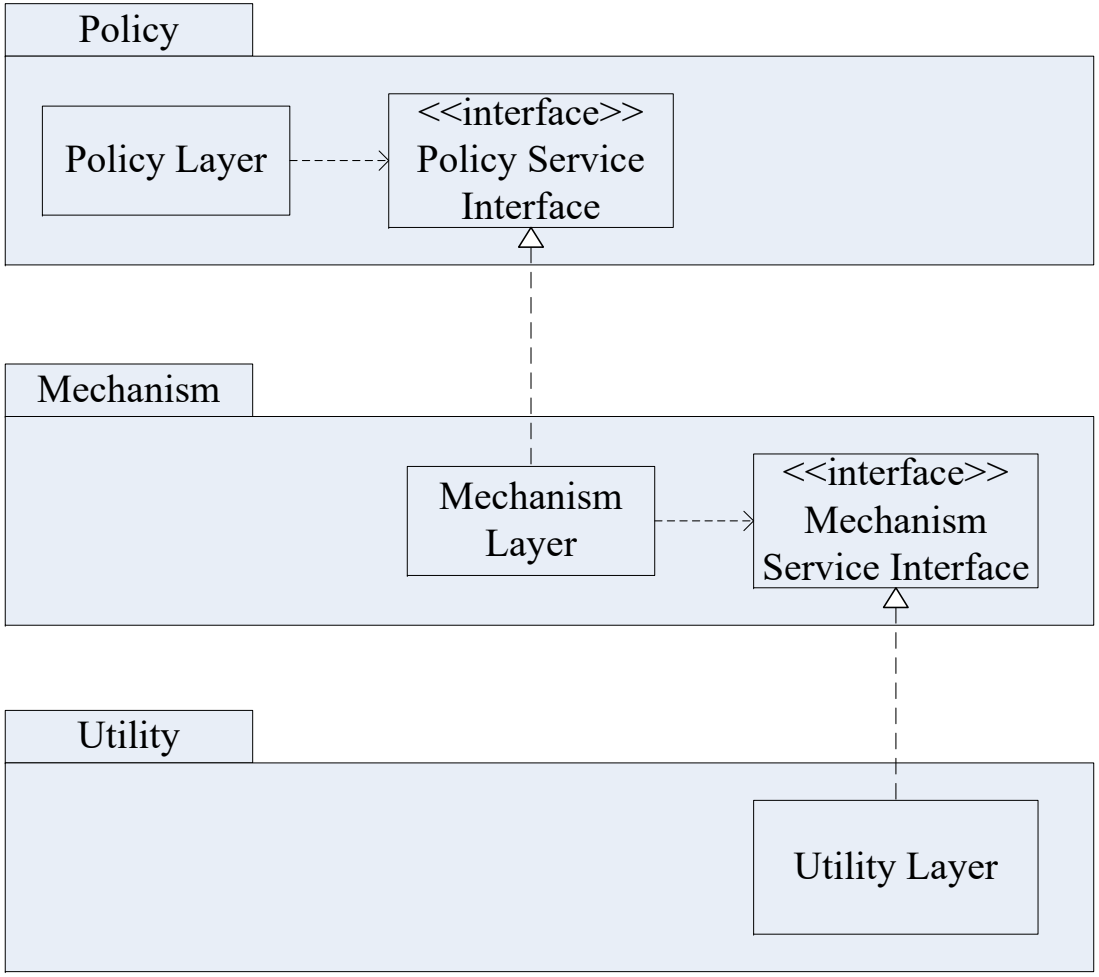
依赖倒置原则：

- 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。

举例：



DIP: The Dependency Inversion Principle



- 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

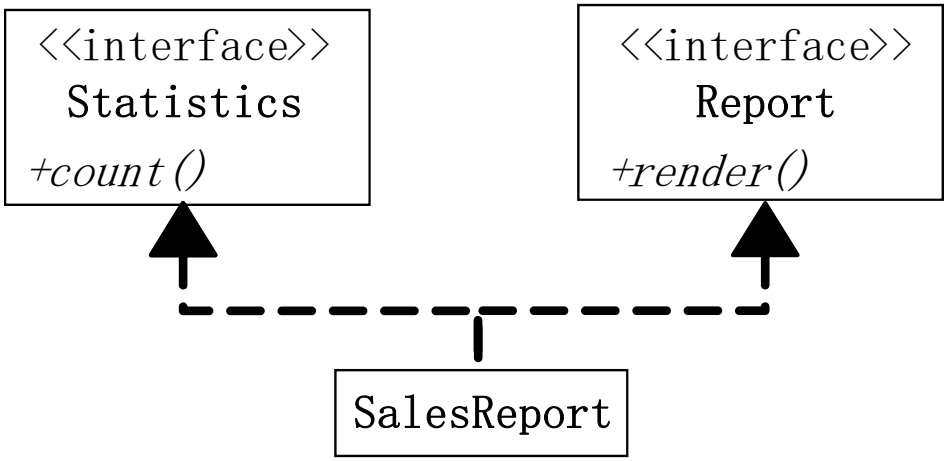
  客户拥有接口，而服务者则从这些接口派生

√

要针对接口编程，
不要针对实现编程

4. ISP: The Interface Segregation Principle

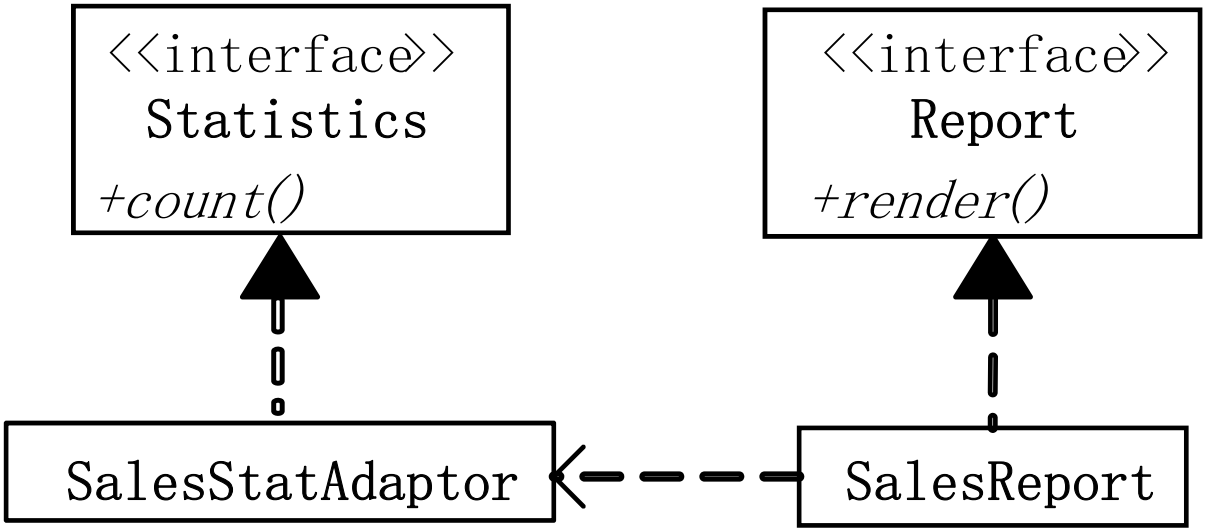
- 接口隔离原则
  - 不应该强迫客户代码依赖于它们不用的方法
  - 一个类的不内聚的“胖接口”应该被分解成多组方法，每一组方法都服务于一组不同的客户程序。
- 举例



使用多个专门的接口
取代使用单一的总接口

ISP: The Interface Segregation Principle

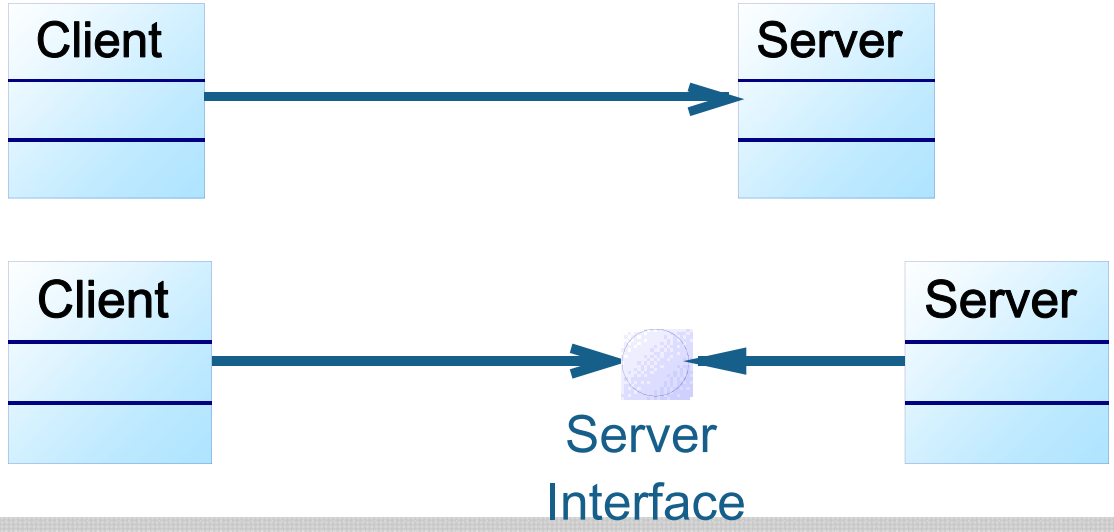
- 举例



5. OCP: The Open-Closed Principle

- 开-闭原则：
  - 模块对扩展open （只需添加新代码，不修改现有代码）
  - 模块对修改closed（只需修改本代码，不修改客户代码）
- 模块可以是方法、类、子系统、应用等
- 开闭原则的关键：找到系统的可变因素，进行封装与抽象。

举例1:

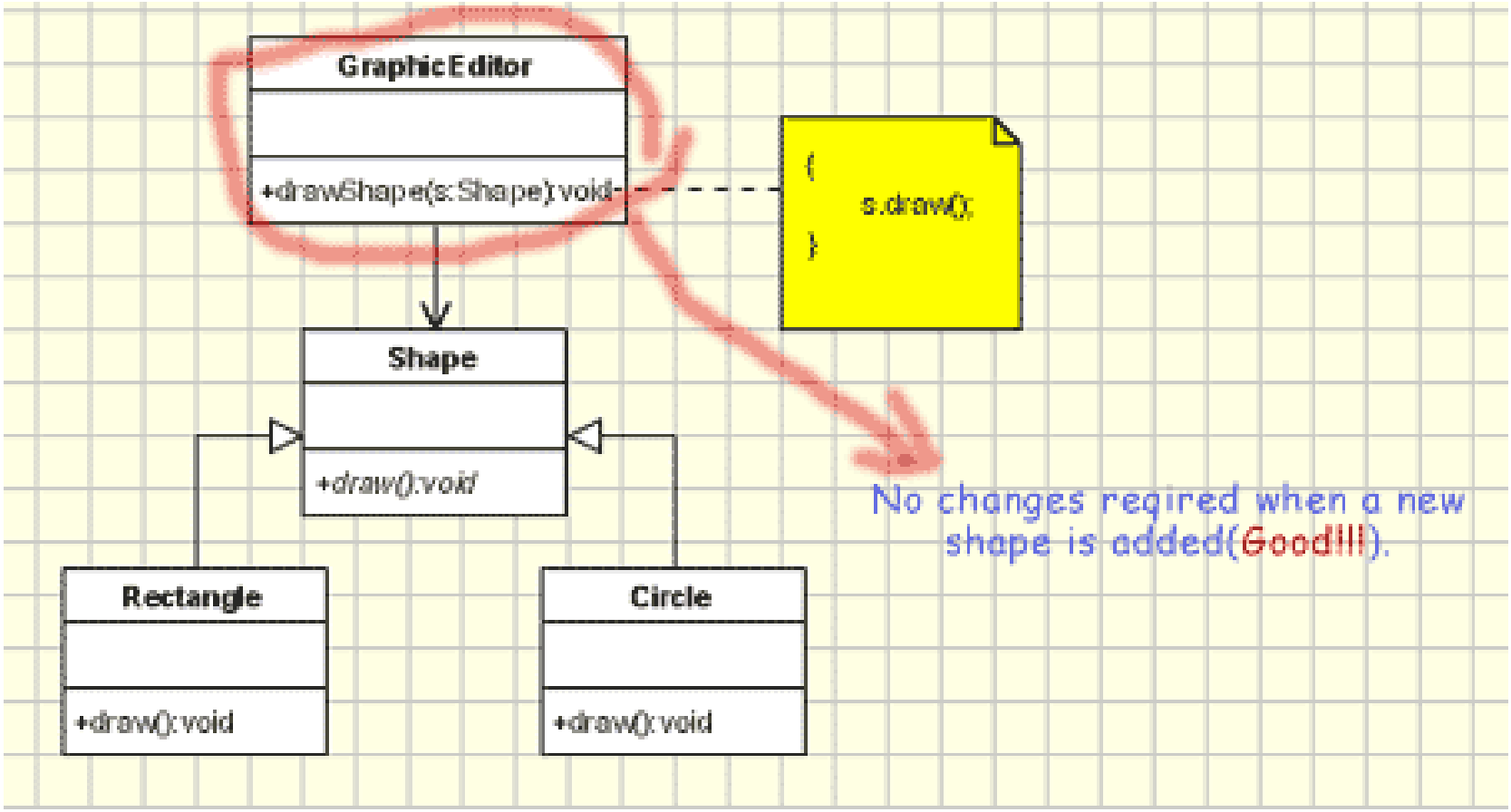


√

封装可变点

OCP: The Open-Closed Principle

举例2:



6. CARP: The Composite/Aggregate Reuse Principle

- 组合/聚合复用原则
  - 要尽量使用组合/聚合，尽量不要使用继承。
  - Favor composition of objects over inheritance as a reuse mechanism.
- 面向对象设计里，复用已有设计和实现的两种基本方法：
  - 继承
  - 组合/聚合

继承复用

- 继承复用通过扩展一个已有对象的实现来得到新的功能，基类明显地捕获共同的属性和方法，而子类通过增加新的属性和方法来扩展超类的实现。继承是类型的复用。
- 继承复用的优点：
  - 新的实现较为容易，因为超类的大部分功能可通过继承关系自动进入子类；
  - 修改或扩展继承而来的实现较为容易。
- 继承复用的缺点：
  - 继承复用破坏封装，因为继承将超类的实现细节 暴露给子类。**“白箱”复用**。
  - 如果超类的实现发生改变，那么子类的实现也不得不发生改变。
  - 从超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，因此没有足够的灵活性。

组合/聚合复用

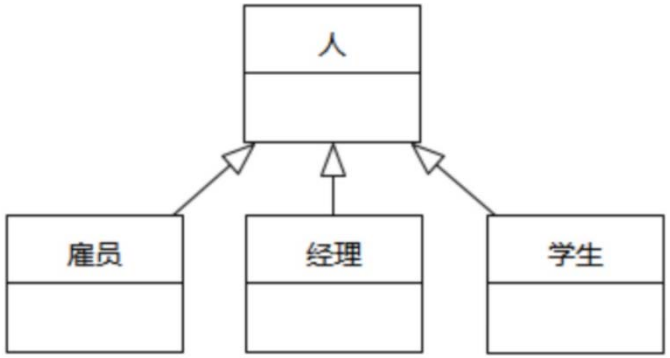
- 由于组合/聚合可以将已有的对象纳入到新对象中，使之成为新对象的一部分，因此新的对象可以调用已有对象的功能
- 优点：
  - 新对象存取成分对象的唯一方法是通过成分对象的接口。
  - 成分对象的内部细节对新对象不可见。 **“黑箱”复用**。
  - 该复用支持封装。
  - 该复用所需的依赖较少。
  - 每一个新的类可将焦点集中在一个任务上。
  - 该复用可在运行时间内动态进行，新对象可动态引用于 成分对象类型相同的对象。
- 缺点：
  - 通过这种复用建造的系统会有较多的对象需要管理。
  - 为了能将多个不同的对象作为组合块（composition block）来使用，必须仔细地对接口进行定义。

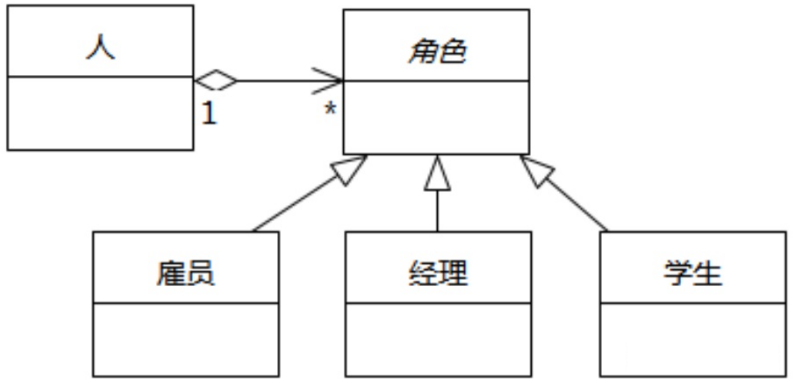
Coad法则

- 要正确选择组合/复用和继承，必须透彻地理解里氏替换原则和Coad法则。
- Coad法则: 只有当以下Coad条件全部被满足时，才应当使用继承关系：
  - 子类是超类的一个特殊种类，而不是超类的一个角色。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。

错误使用

- 错误地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。





√

CARP原则小结

- 组合与继承都是重要的复用方法
- 在OO开发的早期，继承被过度地使用
- 随着时间的发展，人们发现优先使用组合可以获得复用性与简单性更佳的设计
- 可以通过继承，扩充（enlarge）可用的组合类集（the set of composable classes）。
- 组合与继承可以一起工作
- 基本法则是：

优先使用对象组合，而非（类）继承

Summary:设计原则

设计原则名称	设计原则简介	重要性
单一职责原则(SRP)	类的职责要单一，不能将太多的职责放在一 个类中	★★★★☆
开闭原则 (OCP)	软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能	★★★★★
里氏代换原则(LSP)	在软件系统中，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★☆
依赖倒置原则(DIP)	要针对抽象层编程，而不要针对具体类编程	★★★★★
接口隔离原则(ISP)	使用多个专门的接口来取代一个统一的接口	★★☆☆☆
组合复用原则(CRP)	在系统中应该尽量多使用组合和聚合关联关系，尽量少使用甚至不使用继承关系	★★★★☆