关于对象设计

你愿或者不愿，**需求**就在那里，日新月异

你想或者不想，设计创新就在那里，**始终持续**

你测或者不测，**Bug**都在那里，**多多少少**

结束或者不结束，干系人在那里，**决策**终不由己

来，面向对象的阵营里，

或者，让**对象思想驻**进你的心里，

默然，领会。由衷，欢喜 ……

关于对象设计

其实，真相是:

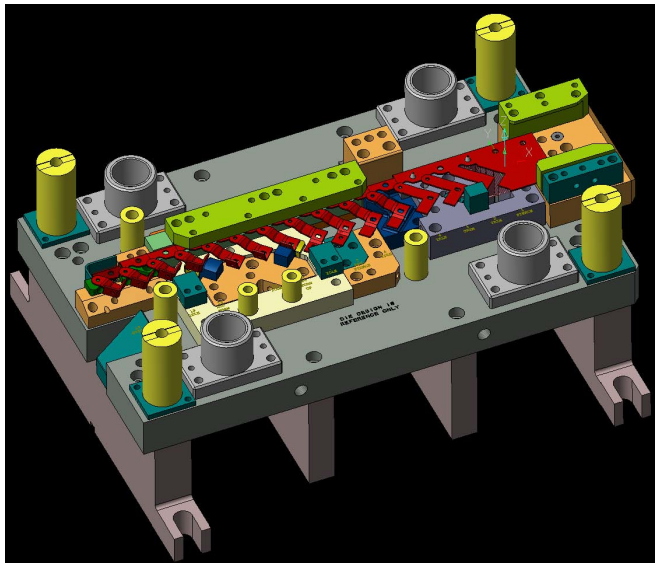
面向对象方法本身并不能保证你的设计成为优秀的设计

You can create a very bad OO design

just as easily as you can create a very bad non-OO design...

面向对象设计过程

- 进行适当的领域分析
- 撰写问题描述，确定系统的开发任务
- 基于问题描述抽取需求
- 开发用户界面原型
- 识别对象类
- 定义每个类的职责
- 确定类之间的交互关系
- 建立系统的设计模型



面向对象思维方式的核心理念

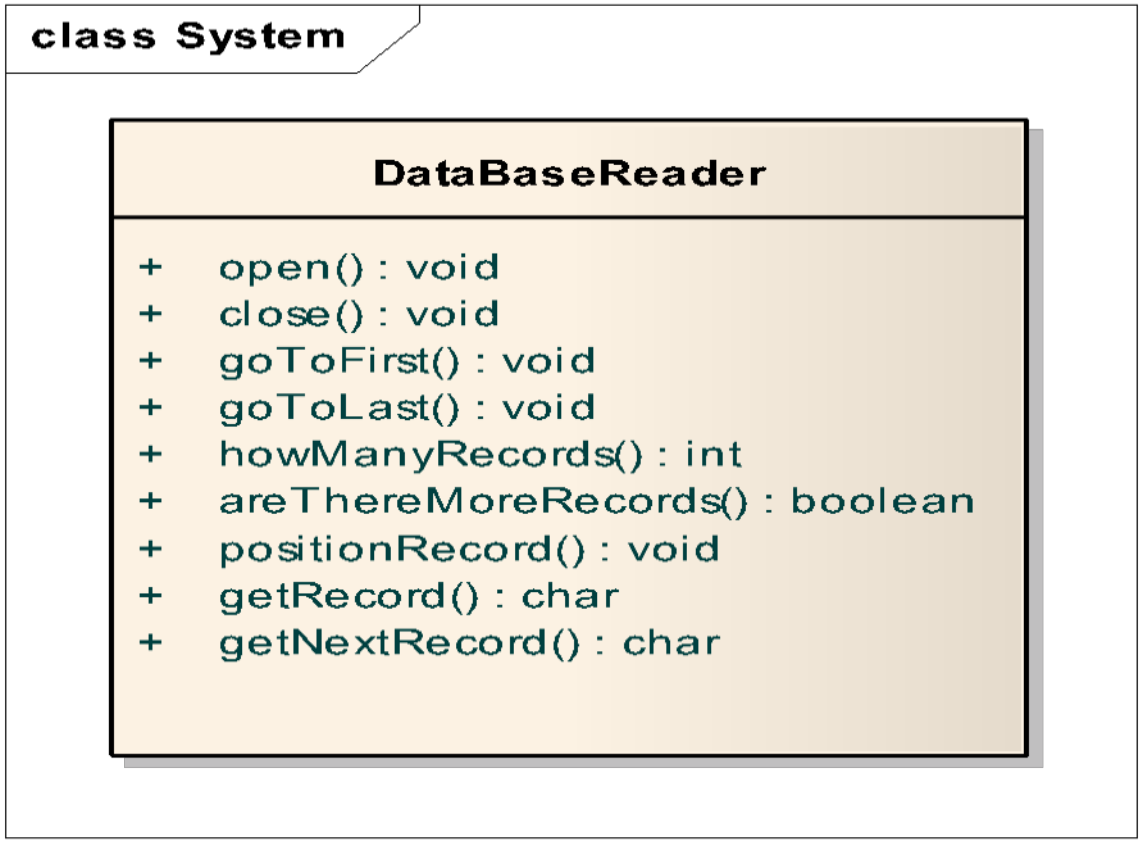
✧区分接口与实现

✧从具体到抽象

✧最小接口原则

区分接口与实现

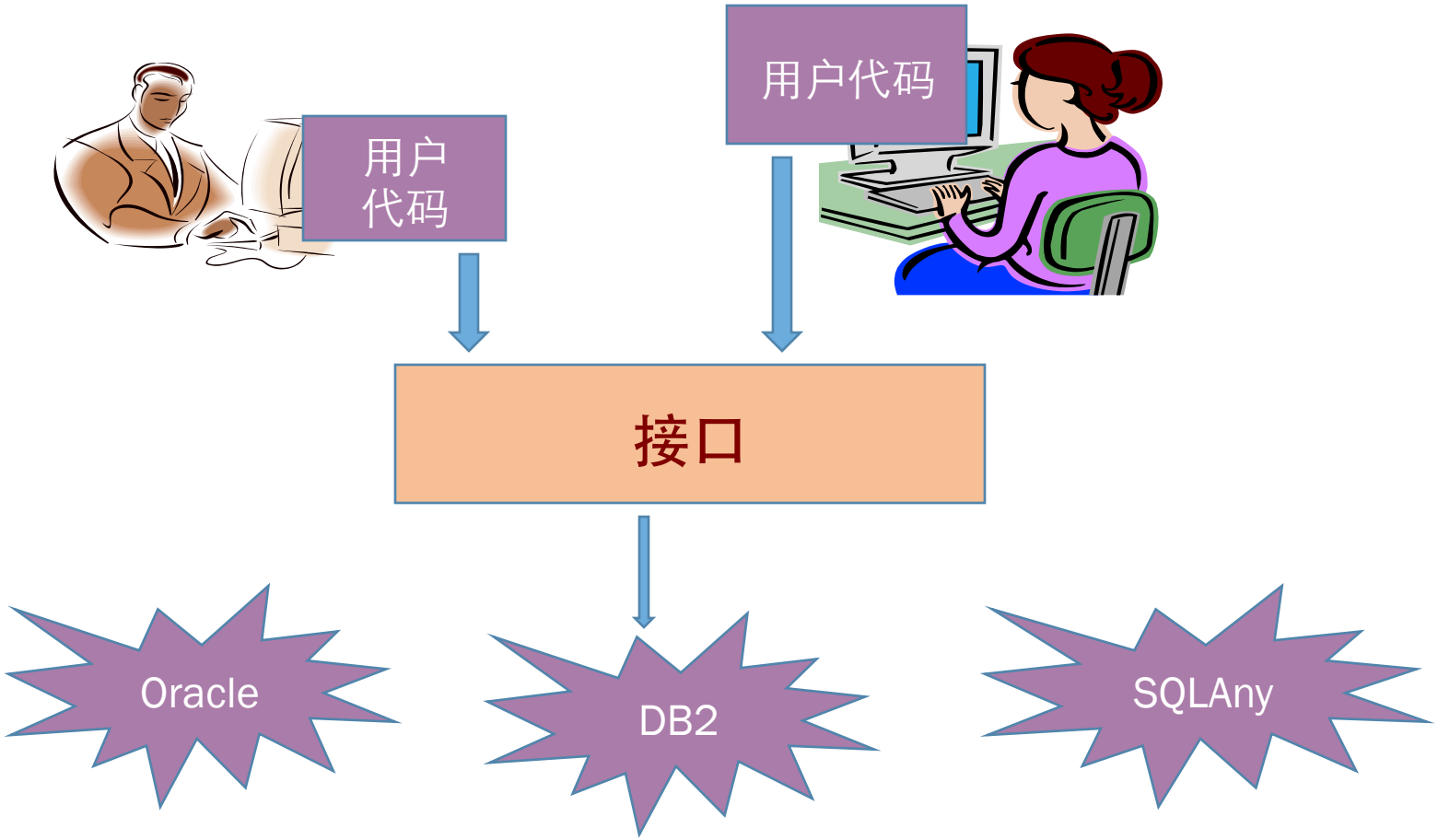
- 接口的标准化 vs. 实现的演化




```
public void open(string name) {
/* some application-specific processing */
/* call the Oracle API to open the DB*/
/* more application specific processing */
}
```

```
public void open(string name) {
/* some application-specific processing */
/* call the SQLAnywhere to open the DB*/
/* more application specific processing */
}
```

接口

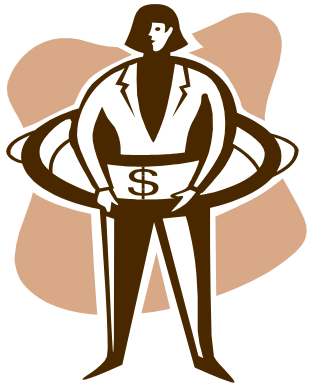


设计抽象的接口

Using Abstract Thinking When Designing Interfaces

抽象的接口

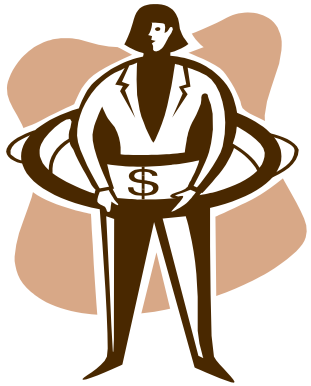
抽象的接口



师傅，请送我去机场

不太抽象的接口

不够抽象的接口



右转

右转

左转

左转

抽象的接口：向用户暴露尽可能少的实现细节

- 让用户知道的关于类的内部实现细节越少越好:
  - 只给看必须的
  - 只看公开的
  - 只为用户的业务需求考虑

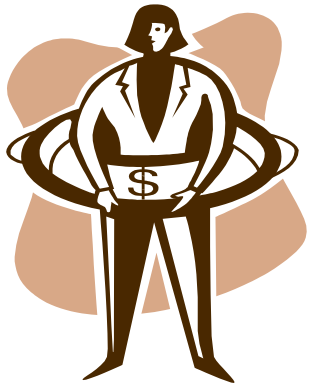
最小用户负担原则

确定用户

- 用户是谁 ？ 重要程度高达50%
- **面向服务的原则 （Services Principle）**

提供服务:
只要能赚钱就好

使用服务:
不要太贵喔



确定对象行为

用例!!!

以往的设计决策在我们定义抽象接口时候会发生变化

识别环境约束

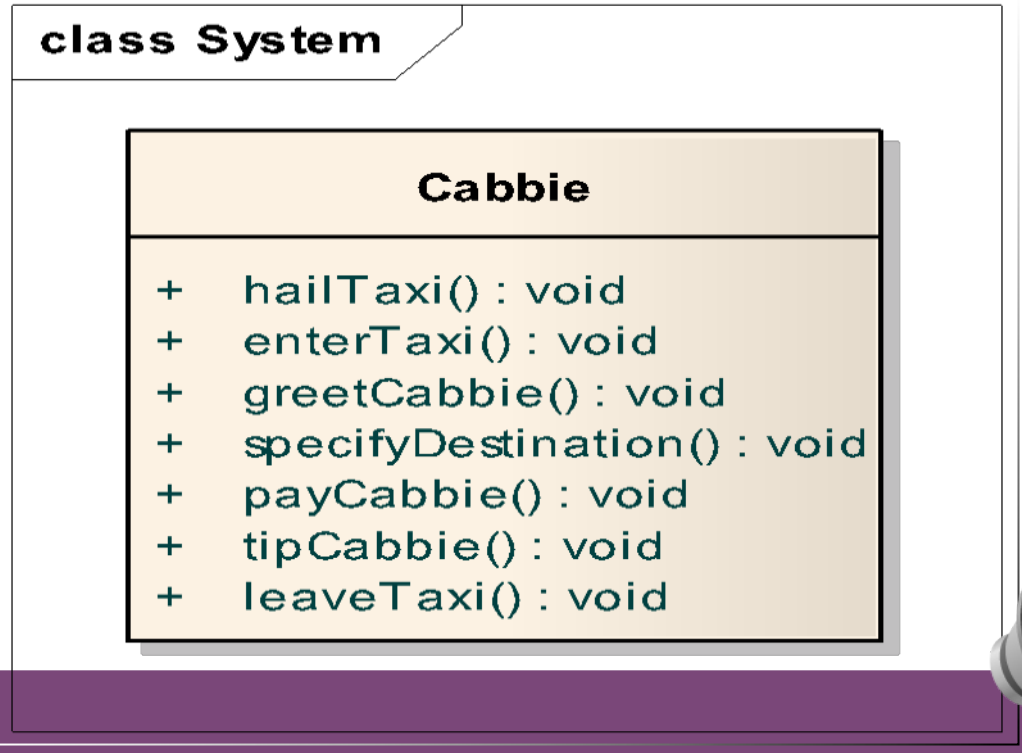
环境对对象的行为施加约束限制条件

前置条件/后置条件/例外条件...

公共接口的识别

- 用户使用出租车对象的时候，需要以下功能:
  - 上车
  - 告知司机终点
  - 付钱
  - 下车

- 用户要用出租车的时候:
  - 有出行地点
  - 召唤出租车
  - 付钱

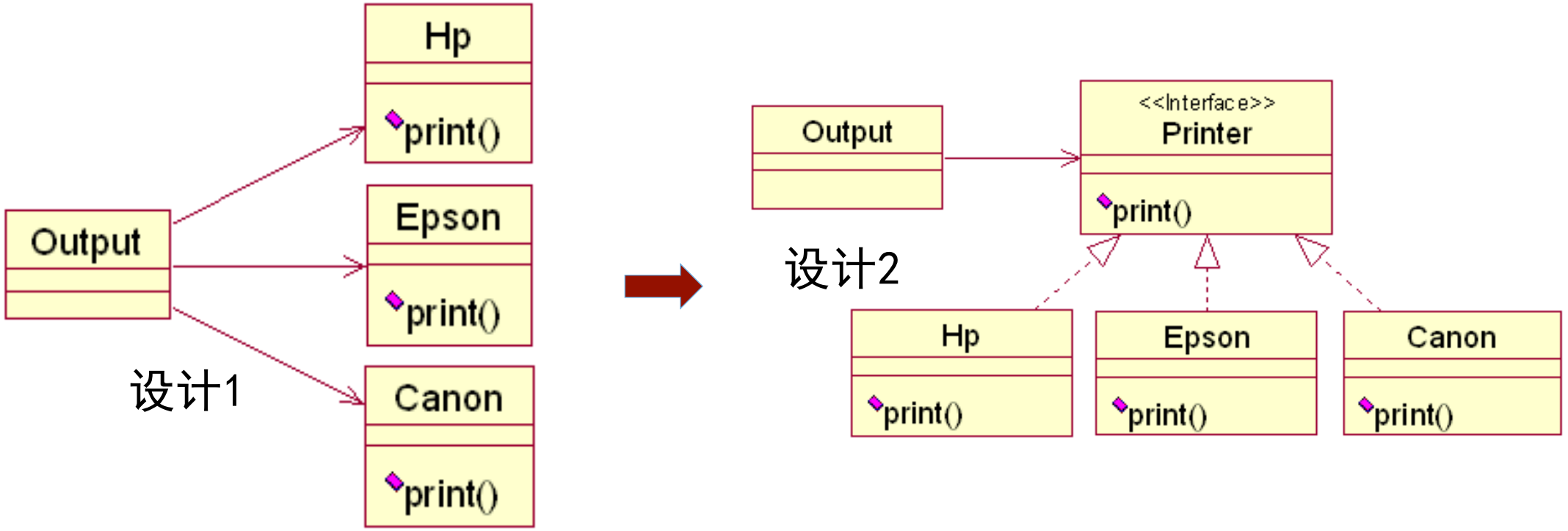


确定实现细节

- 公共接口以外的内容都可以看做是实现相关的
- 用户永远无需关注实现细节
  - 方法的命名和参数定义 (name and parameter list)
  - 编码实现
- 对实现的修改无须牵涉接口
- 实现为用户的期望提供解决方案
- 接口从用户的角度看待对象，实现则是对象的果核和果肉
- 实现中包含有描述对象状态的代码

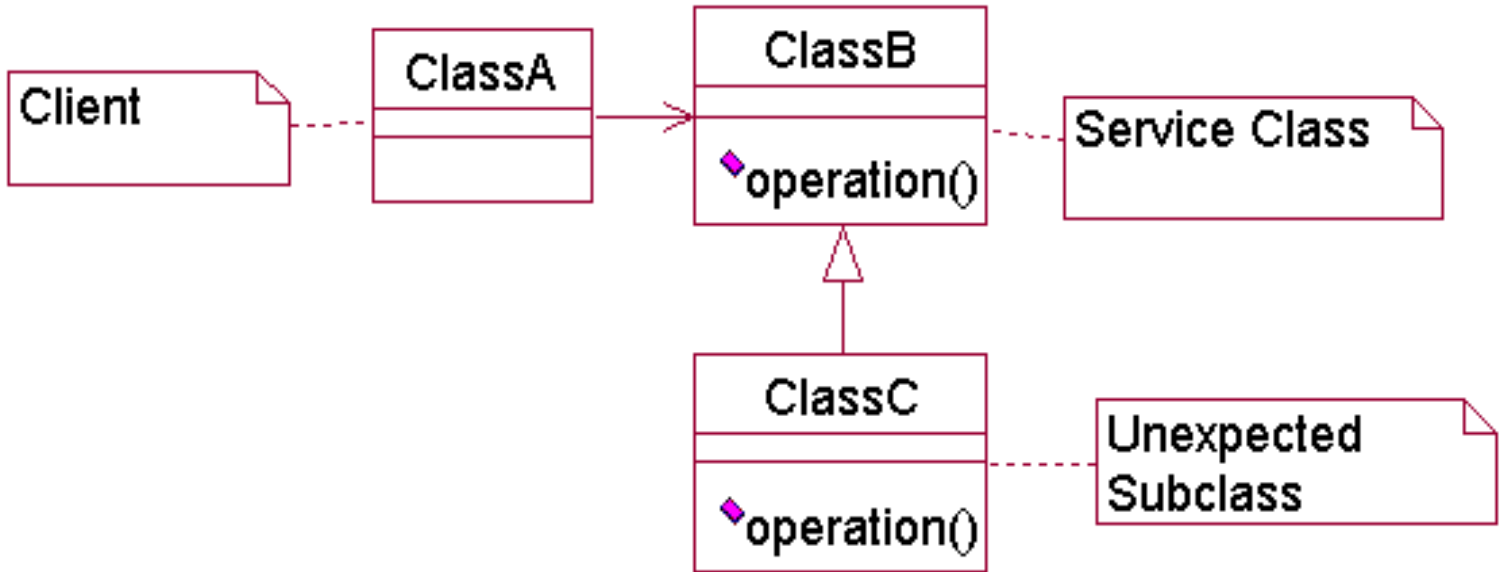
开闭原则(Open/Closed Principle, OCP)

- 最初由Bertrand Meyer提出
- 软件实体在扩展性方面应该是开放的，而在更改性方面应该是封闭的。
- 例：打印输出设计



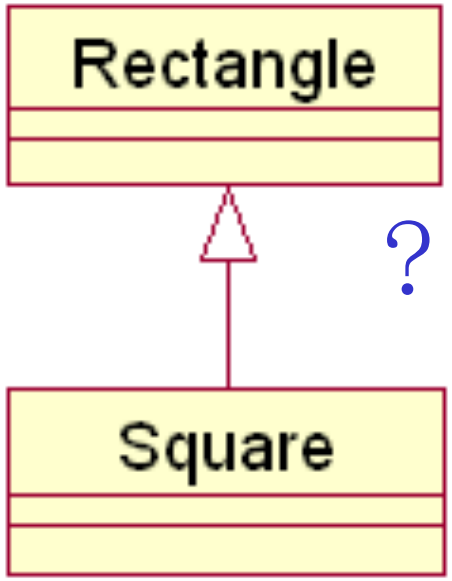
Liskov替换原则 (Liskov Substitution Principle, LSP)

- 最早由**Liskov**于1987年在OOPSLA会议上提出
- 子类可以替换父类出现在父类能出现的任何地方



违反LSP的例子

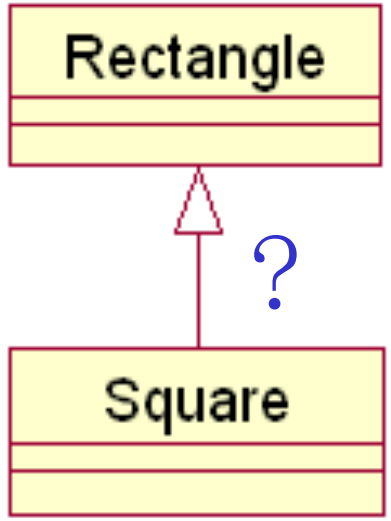
```
public class Rectangle {
  private int topLeftX;
  private int topLeftY;
  int width;
  int height;

  public void setWidth(int width) {
    this.width = width;
  }
  public void setHeight(int height) {
    this.height = height;
  }
  public int getWidth() {
    return width;
  }
  public int getHeight() {
    return height;
  }
}
```



问题：如果将Square作为Rectangle的子类，则如何定义Square类的setWidth和setHeight方法？

解决方法：

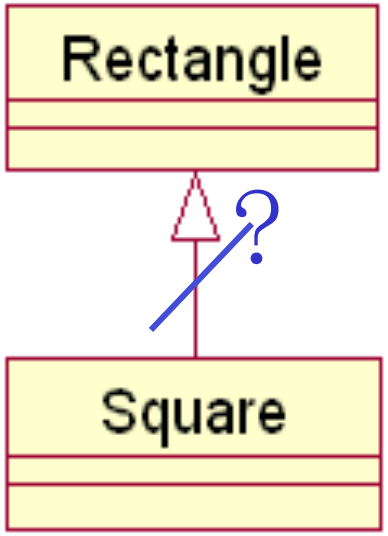
```
public class Square extends Rectangle {
  public void setWidth(int width) {
    super.setWidth(width);
    super.setHeight(width);
  }
  public void setHeight(int height) {
    super.setHeight(height);
    super.setWidth(height);
  }
}
```



这种解决方法是否可行？

考虑下面的代码：

```
public class Test {
  public static void main(String[] args) {
    Test t = new Test();
    Rectangle r = new Rectangle();
    Square s = new Square();
    t.g(r);
//    t.g(s);
  }

  private void g(Rectangle r) {
    r.setWidth(10);
    r.setHeight(20);
    assert (r.getWidth()*r.getHeight() == 200);
  }
```



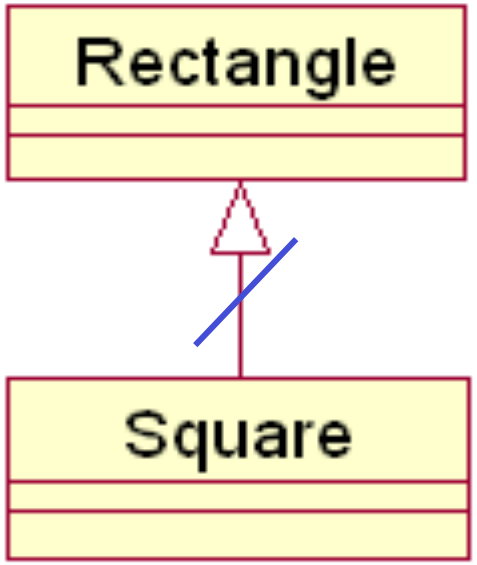
如果传给方法g的参数是Rectangle类型的对象，则没有问题，如果是Square类型的对象，则出错

思考题：如何知道子类的行为符合父类的要求?

- 契约式设计 (Design by Contract)

为了满足Liskov替换原则，设计时要求：

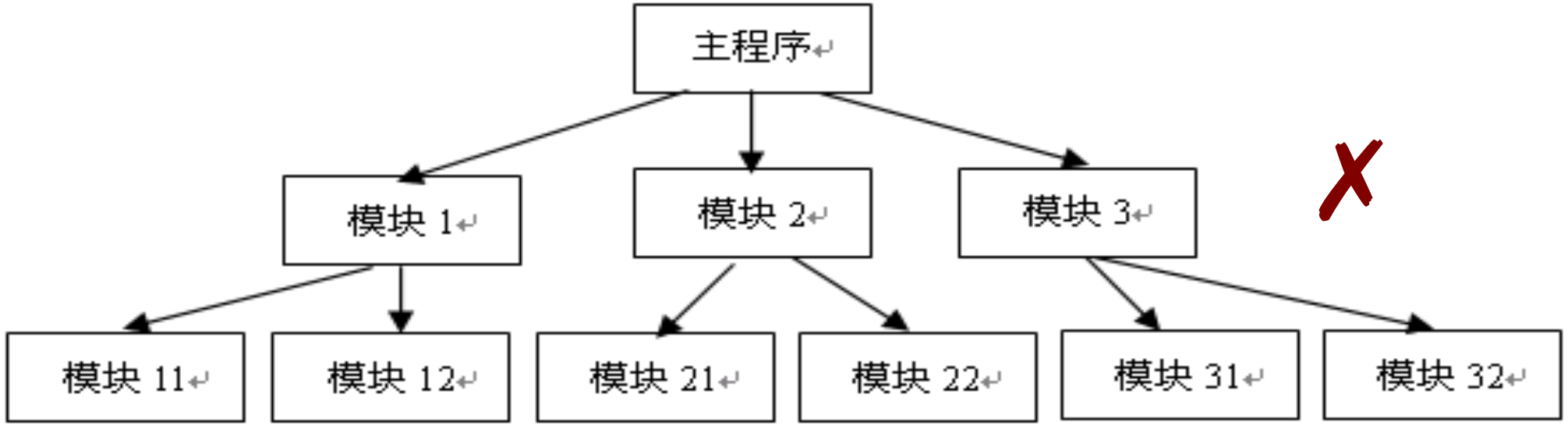
- 子类中方法的前置条件不能强于父类中相应方法的前置条件。
- 子类中方法的后置条件不能弱于父类中相应方法的后置条件。



Liskov 替换原则要求子类宽入严出！

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

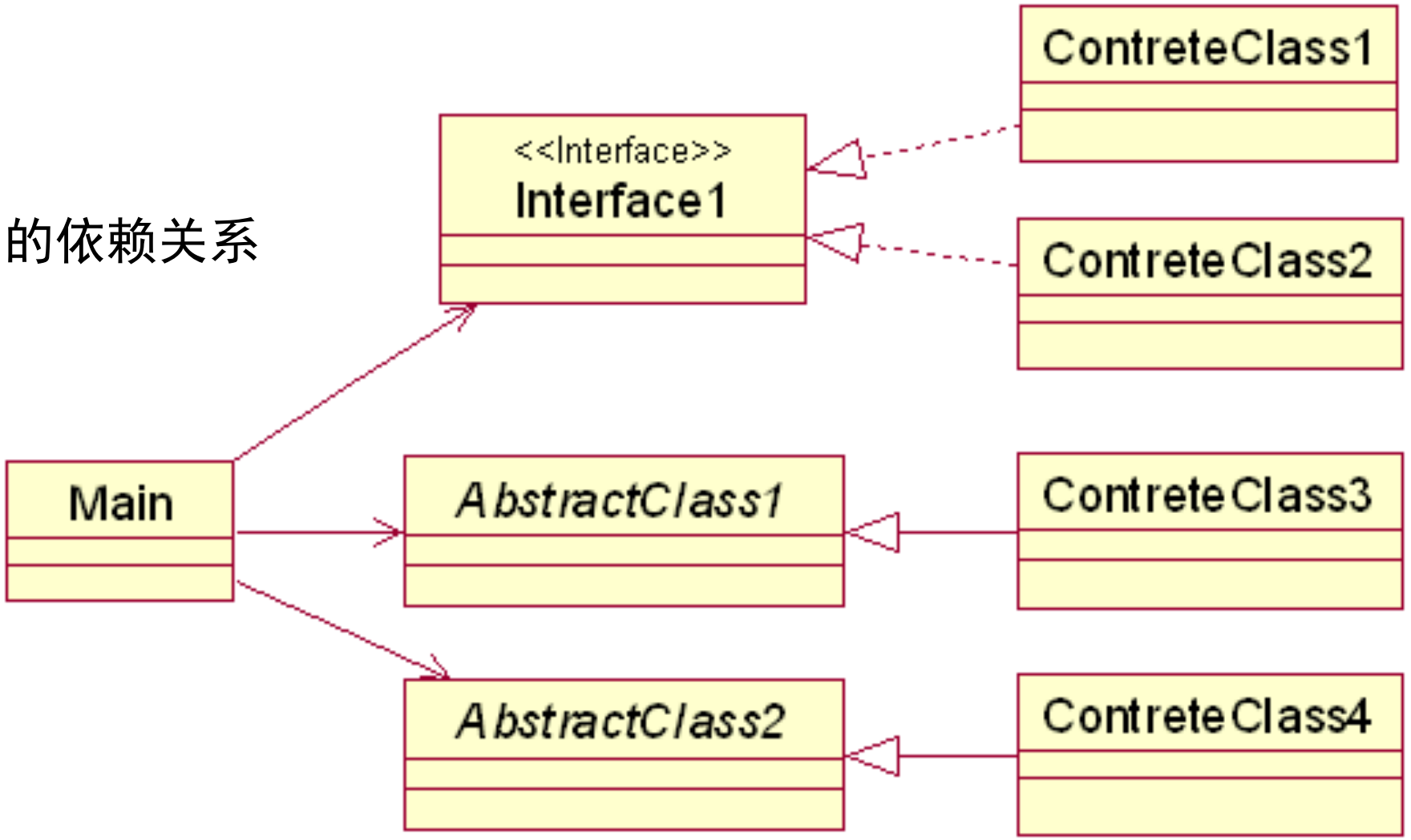
- 依赖倒置原则指的是依赖关系应该是尽量依赖接口(或抽象类)，而不是依赖于具体类。



结构化设计中模块间的依赖关系

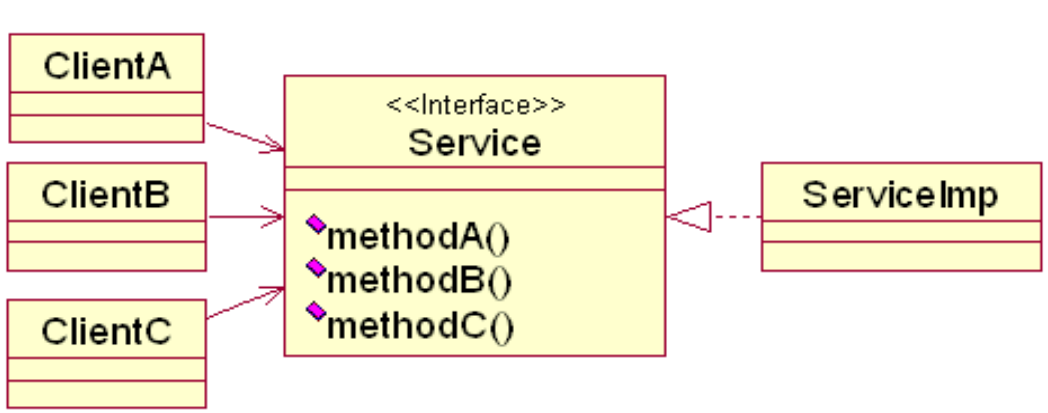
面向对象设计中的依赖关系

OOD中的依赖关系

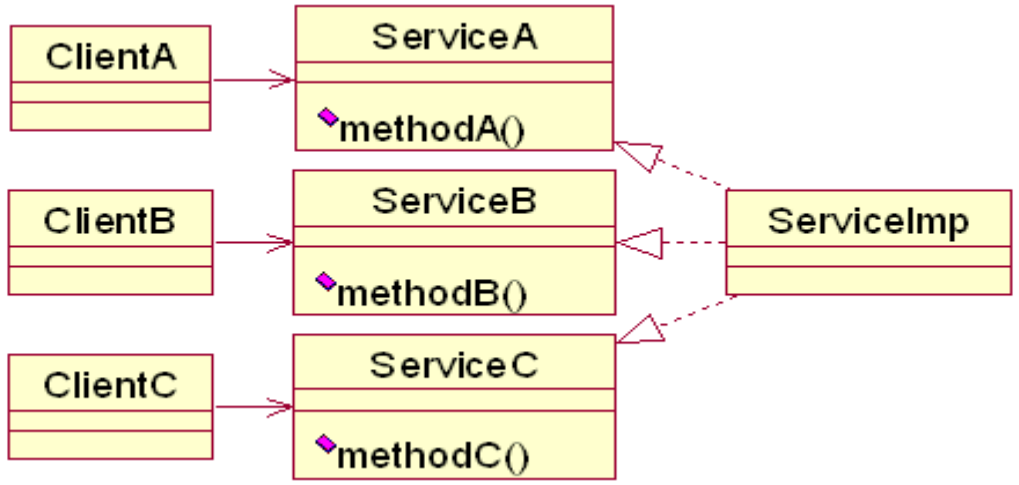


接口分离原则(Interface Segregation Principle, ISP)

- 在设计时采用多个和特定客户类(client)有关的接口要比采用一个通用的接口要好。



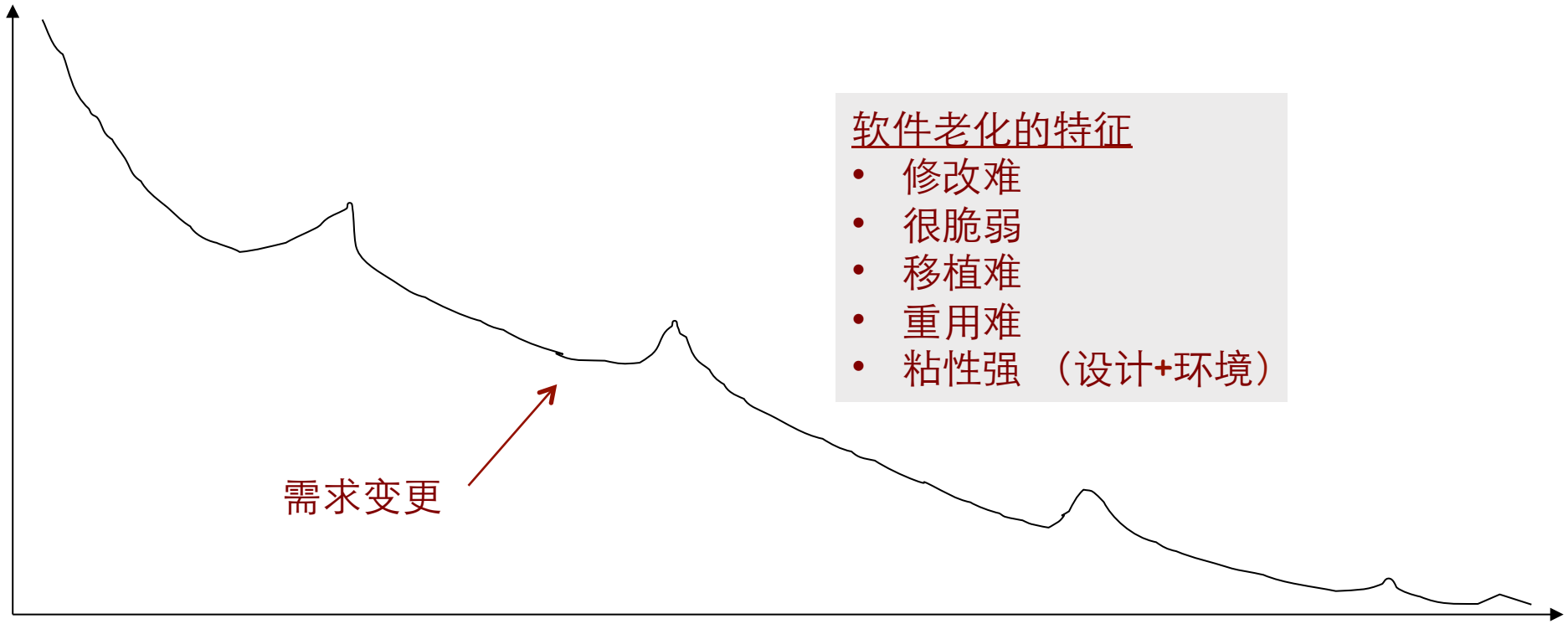
使用通用接口的设计



使用分离接口的设计

好的系统设计的特征

- 用户友好
- 易理解
- 可靠
- 可扩展
- 可移植
- 可伸缩
- 可重用
- ...



软件老化的特征
- 修改难
- 很脆弱
- 移植难
- 重用难
- 粘性强 （设计+环境）

- 简单性：实现简单，使用简单，理解简单，维护简单

OO设计时要注意的一些问题

1. 不同类中相似方法的名字应该相同
2. 遵守已有的约定俗成的习惯
3. 尽量减少消息模式的数目。只要可能，就使消息具有一致的模式，以利于理解。
4. 设计简单的类。类的职责要明确，应该从类名就可以较容易地推断出类的用途。
5. 定义简单的操作、方法
6. 定义简单的交互协议
7. 泛化结构的深度要适当
8. 把设计变动的副作用减至最少