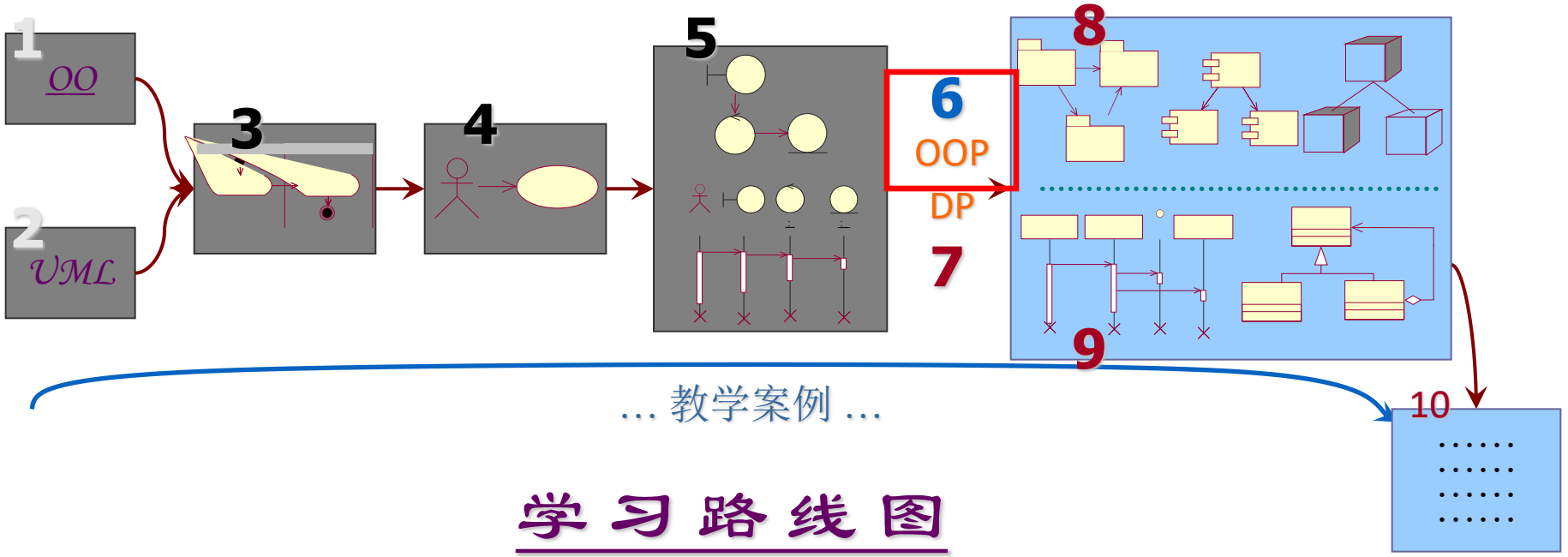
Lecture 11. Object-Oriented Design Principles
(面向对象的设计原则)

Object Oriented Modeling Technology
面向对象建模技术

Professor: Yushan Sun
Fall 2022

学习路线图



Contents

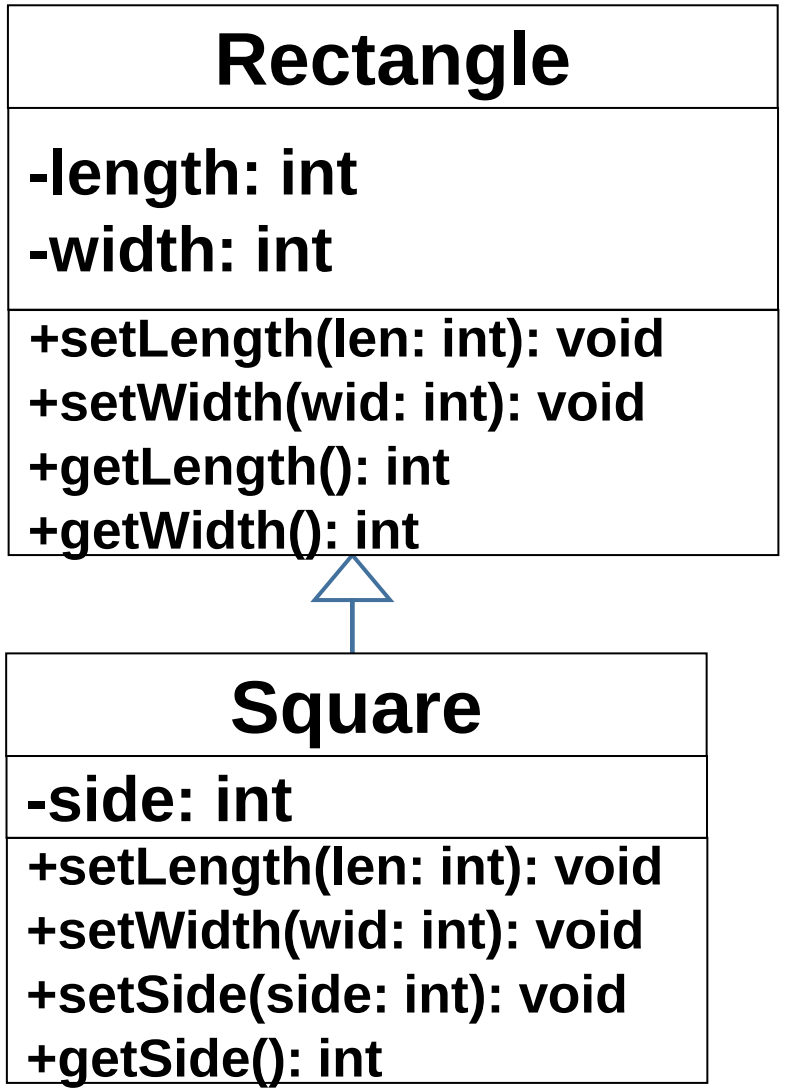
引言 (Introduction)

从问题开始！

- **例1：矩形与正方形**
  - 假如我们有一个类：矩形（Rectangle）
  - 我们需要一个新的类，正方形（Square）
  - 问：可否直接继承矩形？

回答：没问题呀，因为数学上正方形就是矩形的子类！

开始设计：正方形




```
public class Rectangle {
    private int length;
    private int width;
    public void setLength(int l) {
        length = l;
    }
    public int getLength() {
        return length;
    }
    public void setWidth(int w) {
        width = w;
    }
    public int getWidth() {
        return width;
    }
}
```

```
public class Square
    extends Rectangle {
        public void setSide(int side){
            super.setLength(side);
            super.setWidth(side);
        }
        public void setLength(int len)
        {setSide(len);}
        public void setWidth(int wid)
        {setSide(wid);}
}
```

某程序员写了一个客户程序中，有一个resize方法，如下：

```
public static void resize(Rectangle r) {
      while (r.getLength() <= r.getWidth()) {
         r.setLength(r.getLength() + 1);
      }
      System.out.println( “It’ s OK.");
}

    Rectangle r1 = new Rectangle();
    r1.setLength(5);
    r1.setWidth(15);
    resize(r1);

    Rectangle r2 = new Square();
    r2.setLength(5);
    r2.setWidth(15);
    resize(r2);
```

出问题了：一个正方形竟然出现了4条边不等的情况。

- ✓使用父类（矩形）时，程序正常运行
- ✓使用子类（正方形）时，程序出错
- ✓设计出问题了？继承出问题了？

为什么会出现问题呢?

违背了面向对象的设计原则!

面向对象的设计原则

- 面向对象的设计原则
  - 是面向对象设计的基本指导思想
  - 是评价面向对象设计的价值观体系
  - 是设计模式的出发点和归宿
- 面向对象的设计原则是构造高质量软件的出发点

设计质量：培养灵敏的嗅觉

- 糟糕的设计总是散发出臭味，让人不悦
  - 判断一个设计的好坏，主观上能否让你的合作方感到心情愉悦，是最直观的标准
- 设计开发人员要培养嗅觉，当你看到UML图或者代码，感到杂乱、繁琐、郁闷的时候，你可能正面对一个糟糕的设计
- 这种嗅觉是在实践开发中培养起来的，而面向对象设计原则对此加以归纳和总结

设计质量：坏的设计

- 什么是坏的设计?
  - 僵硬性（Rigidity）：刚性，难以扩展
  - 脆弱性（Fragility）：易碎，难以修改，一处改动，程序的很多地方就碎了
  - 牢固性（Immobility）：无法分解成可移植、复用的组件
  - 粘滞性（Viscosity）：设计粘滞性（简单的修改即破坏已有设计方案）和环境粘滞性(开发环境迟钝、低效)
  - 不必要的复杂性(Needless complexity)：包含了很多当前无用的组成部分
  - 不必要的重复性(needless repetition): 设计中包含重复结构，而这些重复的结构本可通过复用的方式进行统一的管理
  - 晦涩性（Opacity）：不透明，很难看清设计者的真实意图

设计质量：好的设计

- 什么是好的设计?
  - 容易理解
  - 容易修改和扩展
  - 容易复用
  - 容易实现与应用
  - 简单、紧凑、经济适用
- 让人工作起来心情愉快的设计
- 设计原则是提高设计质量的基本原则

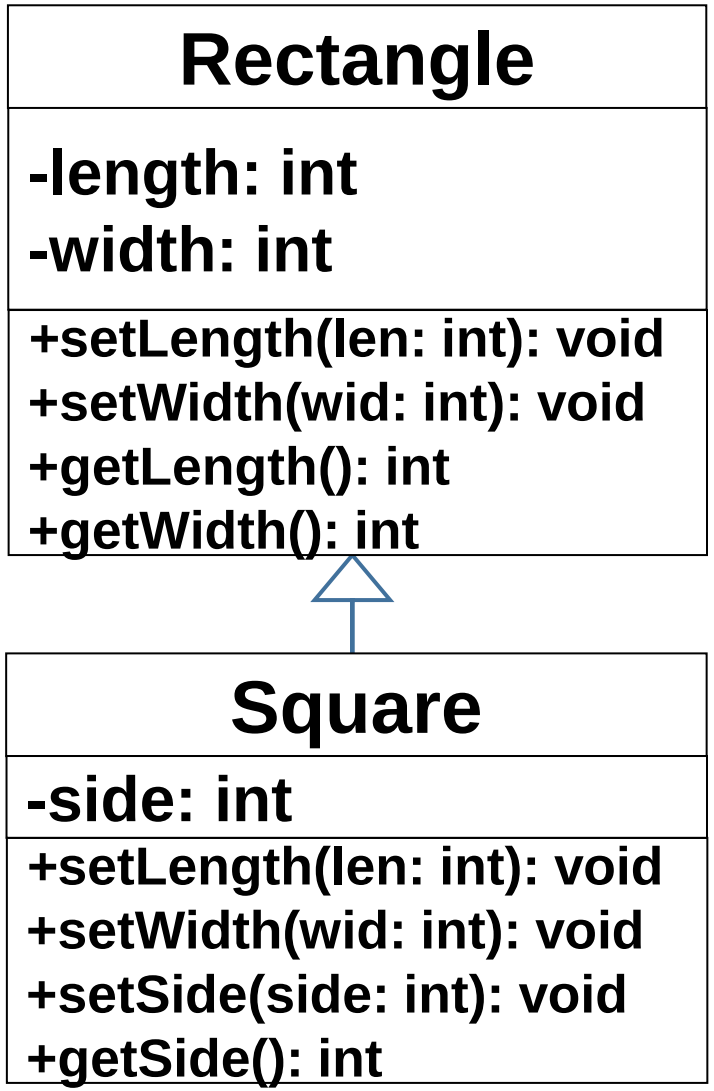
Return

LSP：Liskov替换原则

LSP

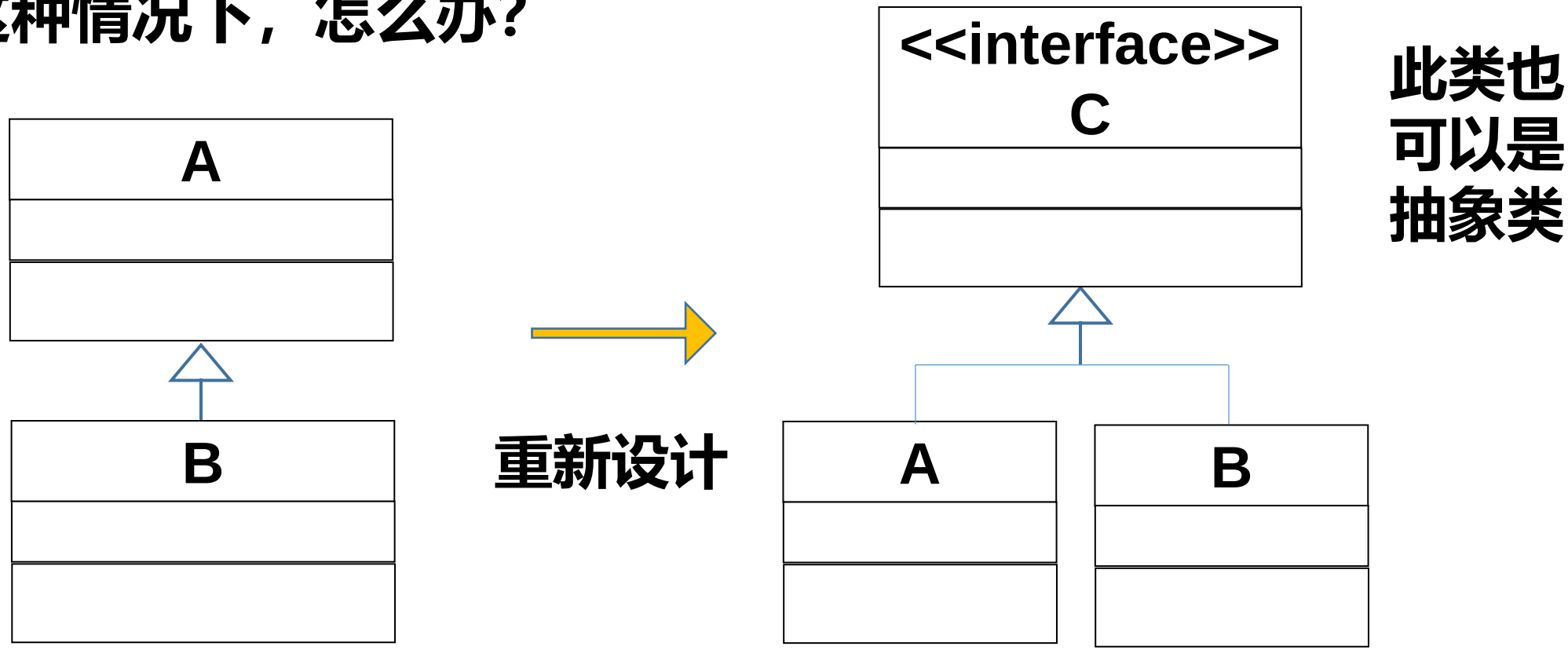
- LSP(The Liskov Substitution Principle, Liskov替换原则)
  - “若对于类型S的任一对象o_1，均有类型T的对象o_2存在，使得在T定义的所有程序P中，用o_1替换o_2之后，程序的行为不变，则S是T的子类型”
  - 如果在任何情况下，子类（或子类型）或实现类与超类对象都是可以互换的，那么继承的使用就是合适的。为了达到这一目标，子类不能添加任何父类没有的附加约束
  - “子类对象必须可以替换超类对象”

违背LSP原则的例子



- Square类针对length、width添加了Rectangle所没有的附加的约束（即要求length=width），这样Square类（子类）不能完全替换Rectangle（父类）
- 违背了LSP原则
- 带来潜在的设计问题（使用resize方法时，子类出错！）

在这种情况下，怎么办?

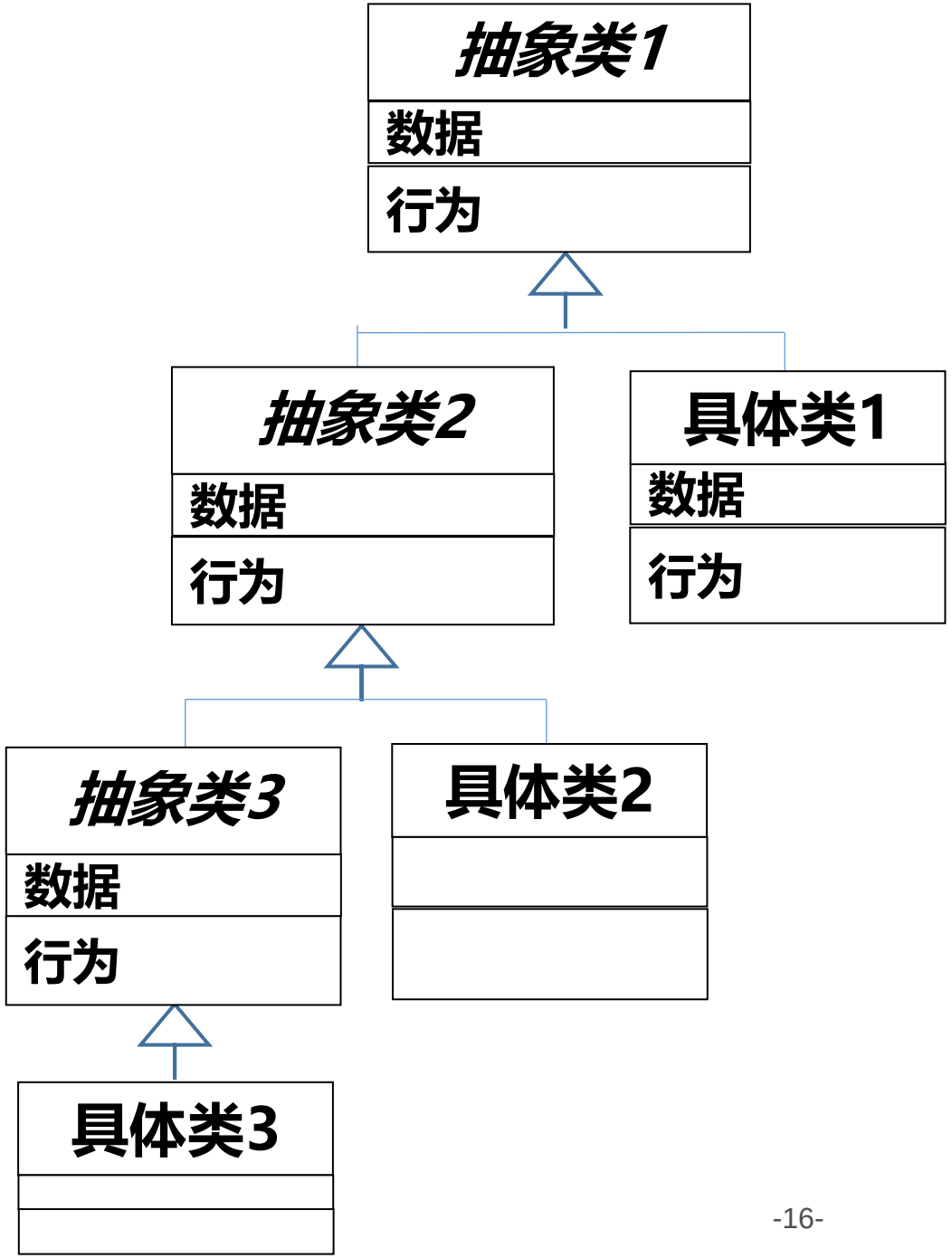


在可能的情况下，重新设计为另外一个层次类，让A与B都作为子类

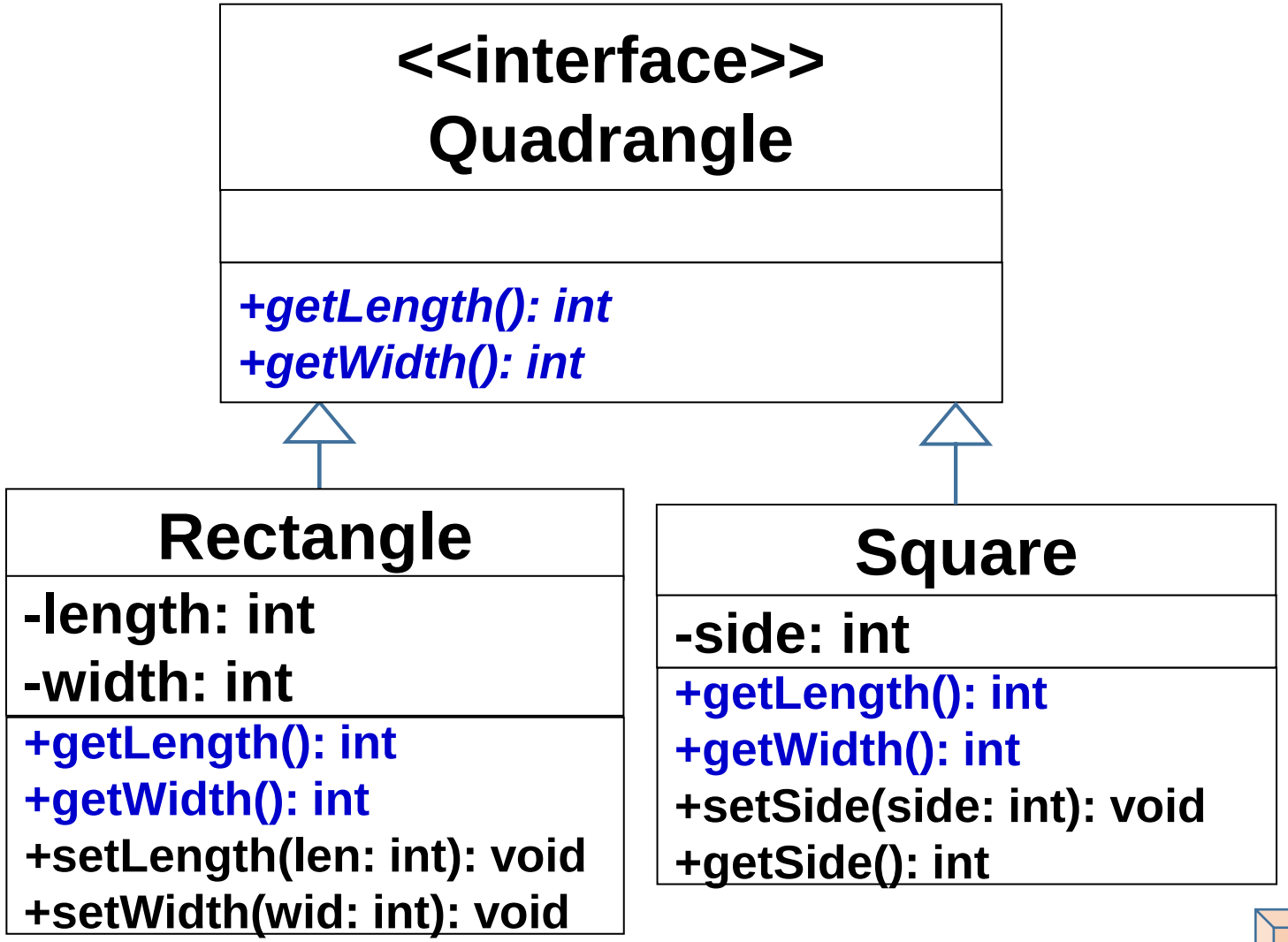
- 子类实现接口类
- 子类继承抽象类，或

抽象类与具体类

- 只要有可能，不要从**具体类**继承，而建议从抽象类继承
- 层次类的设计经验
  - 行为集中的方向是向上的（抽象类）
  - 数据集中的方向是向下的（具体类）



解决方案


<<interface>>
Quadrangle
+getLength(): int
+getWidth(): int
Rectangle
-length: int
-width: int
+getLength(): int
+getWidth(): int
+setLength(len: int): void
+setWidth(wid: int): void
Square
-side: int
+getLength(): int
+getWidth(): int
+setSide(side: int): void
+getSide(): int

Return

开-闭原则
(OCP-The Open-Close Principle)

Open/closed Principle（开闭原则）

- **开闭原则的定义：开闭原则说明软件实体应该对扩展开放，对修改关闭；即在不修改已经存在的源代码的情况下，修改其行为**
- **In object-oriented programming, the open-closed principle states software entities (classes, modules, functions, etc.) should be**
  - **open for extension, but**
  - **closed for modification**

that is, such an entity can allow its behavior to be modified without altering its source code.

开闭原则的好处

- 这在软件生产环境中尤其有价值，因为在生产环境下可能需要更改源代码。然后，极有可能对源代码进行
  - 代码评审；
  - 单元测试，
  - 集成测试等

- 符合开闭原则的代码可以做到在不修改源代码的情况下扩展功能，因此从经济方面非常节省。

- **关于实现开闭原则的两种策略**

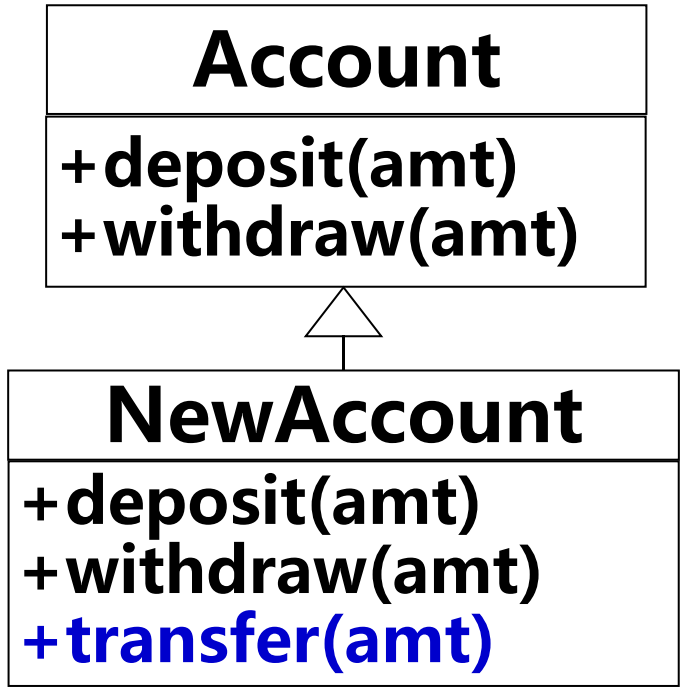
策略1：Bertrand Meyer的策略(老方法)

- 只有在更正代码错误的情况下才能修改一个类；
- 新增或改变功能需要创建不同的类。

a)新类通过继承的方式复用原来的类的代码

b)子类可能与超类可能有不同的接口

例2：银行账户类

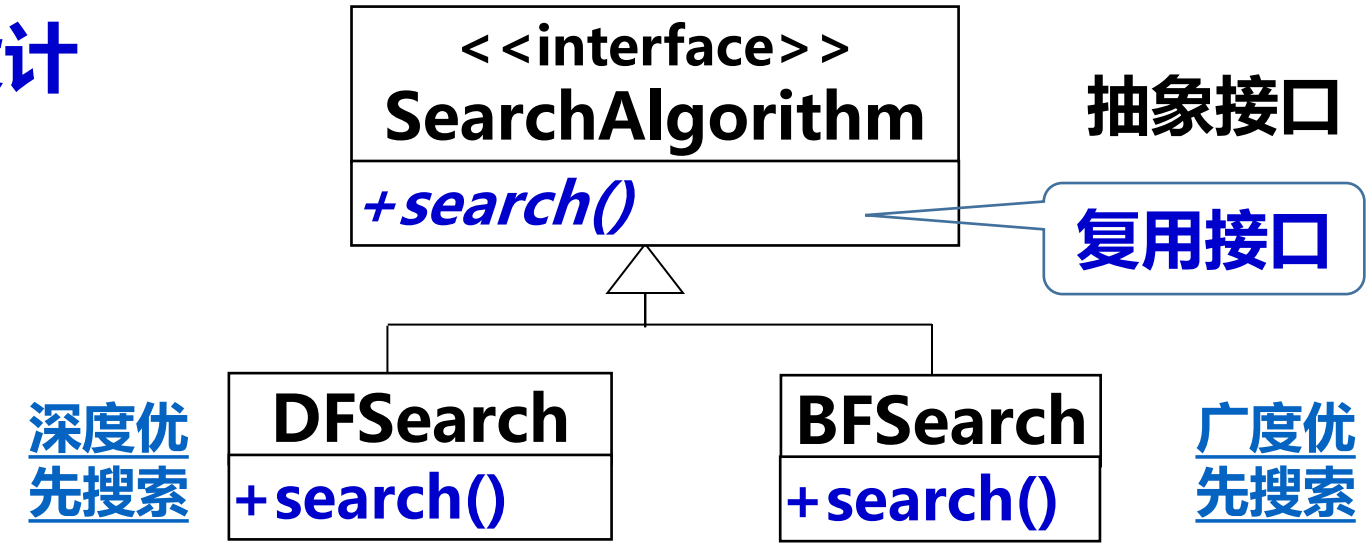


接口改变了

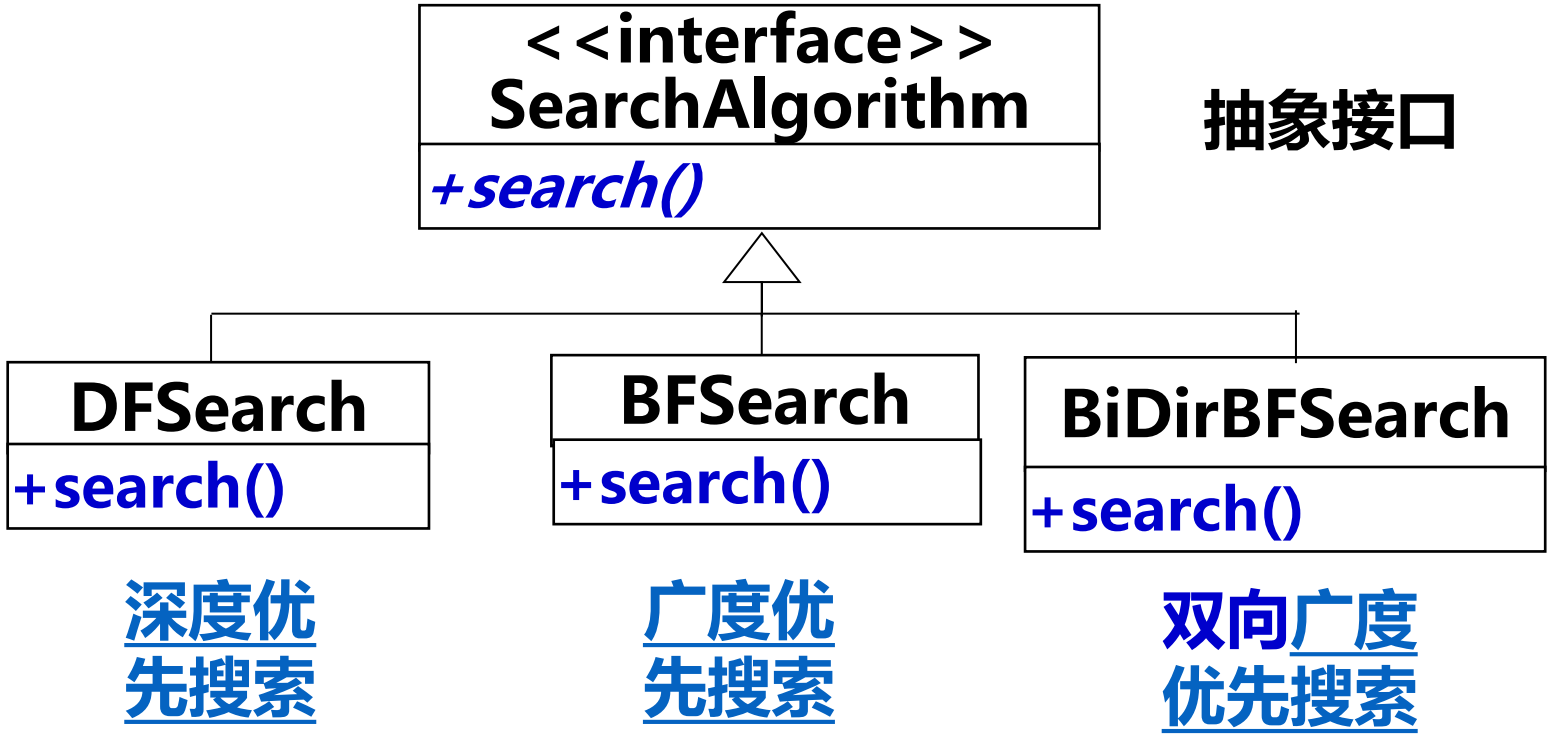
- **Meyer的主要思想是：复用实现（代码），而不复用接口**
- 现有实现代码不允许修改，新实现不需要实现现有接口。

- **策略2： 1990年代新思维：复用抽象接口，而不是复用实现（代码）**
- **在20世纪90年代，开-闭原则流行，并且开-毕原则被重新定义为使用抽象接口，**
  - **其中实现代码可以改变；**
  - **可以创建多个实现并以多态方式彼此替换。**

例3. 搜索算法层次类的设计



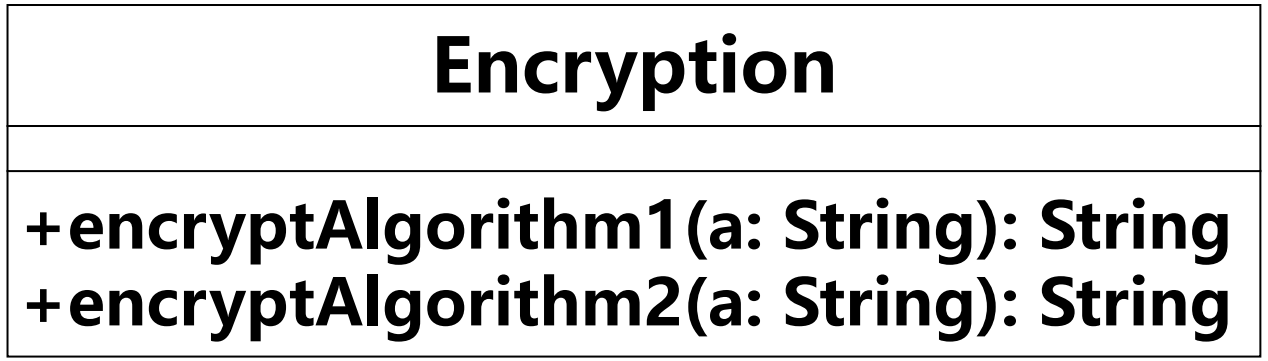
机制: 重用抽象接口，而不是重用实现（代码）



接口不变，通过新增加子类扩展系统功能

- **复用抽象接口的好处**
- 20世纪90年代的新定义提倡从抽象基类继承。接口规范可以通过继承重用，但实现不需要。
- 利用继承获得抽象基类的接口的复用
- 已存在的接口对修改是关闭的；即不允许修改接口
- 但是子类、实现类至少必须实现该接口，各个子类或实现子类都可以有不同的行为
- 即在不必修改接口的情况下即可修改行为

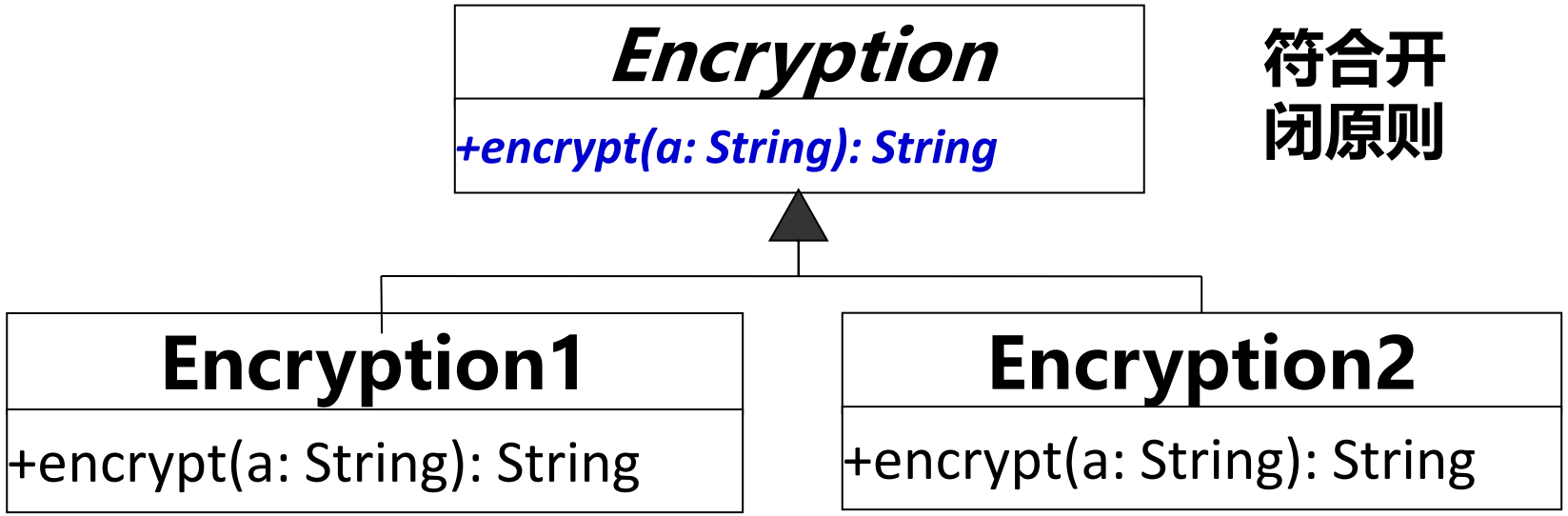
- **例4. 如果您需要设计一个程序来对文本字符串进行加密。目前，有两种算法加密Algorithm1与Algorithm2，以不同的方式加密文本字符串。你初始设计如下:**



用一个类封装两个算法

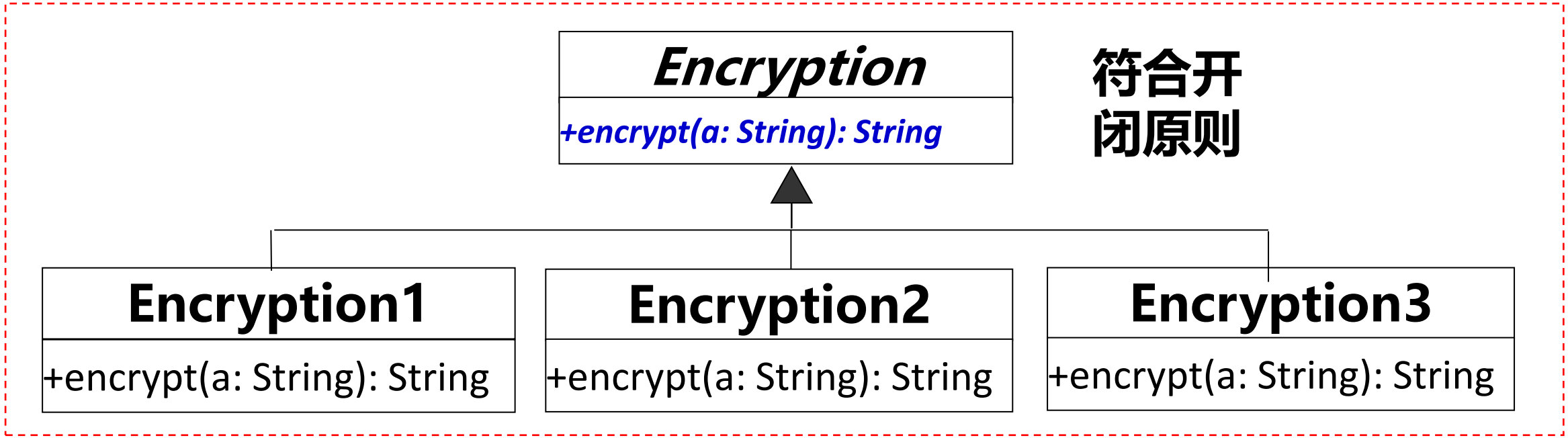
- **评论（Comment）：**
  - **若修改一个算法，需要修改该类，并且需要重新编译整个类**
  - **若添加一个新算法，需要修改该类，并且需要重新编译整个类**
  - **该设计不遵循开闭原则**

改善设计: use an abstract super class with two subclasses, each does encryption the way its class name suggests.



符合开闭原则

- **新设计的优点:**
- **如需要修改一个算法，只需修改一个子类，而不影响已经存在的类。**
- **如需要增加一个新算法，只需增加一个新子类，而不影响已经存在的类。**
- **即，在不修改源代码的情况下，修改了行为。此设计符合开闭原则。**



Return

单一职责原则

SRP(The Single Responsibility Principle)

SRP单一职责原则

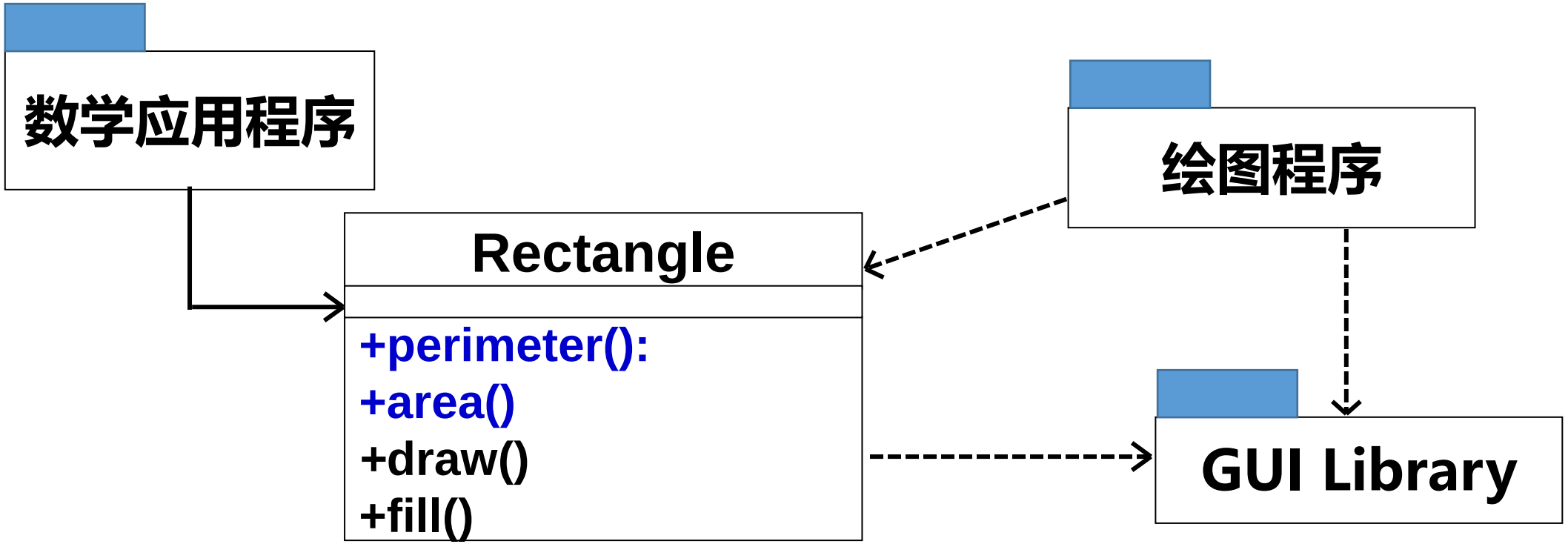
- SRP（The Single Responsibility Principle, 单一职责原则）
  - 就一个类而言，应该仅有一个引起它变化的原因
- 有关类的职责分配问题，是面向对象设计中最重要的基本原则

> “A critical, fundamental ability in OOA/D is to skillfully assign responsibility to software components.”
> Craig Larman

SRP本质

- SRP体现了内聚性（Cohesion）
  - 内聚性：一个模块的组成元素之间的功能相关性
- 类的职责定义为“变化的原因”，每个职责都是变化的一个轴线；
  - 当需求变化时，该变化会反映为类的职责的变化
  - 如果一个类承担了多于一个的职责，那么引起它变化的原因就会有多个

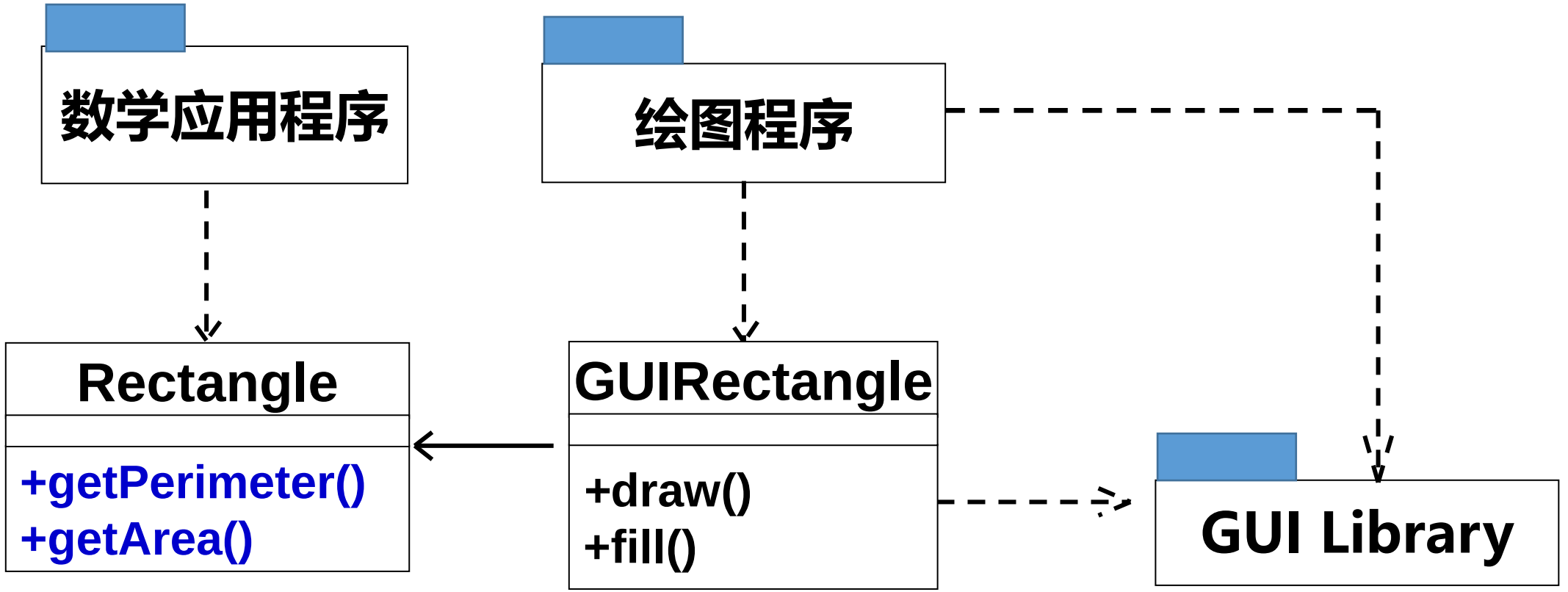
例5：违反SRP的案例



- Rectangle类会因为两方面的原因而变化：计算几何方面的原因和用户界面设计方面的原因。
- 其中一个发生变化后，必须修改Rectangle类，而这种修改则可能导致另一个应用程序出错；
- 除此之外，违反SRP还会带来物理依赖的缺点。

解决方案

- 增加新的类，使得每个类仅有一个职责



只有计算Rectangle的各种计算几何方面的责任

只有绘制Rectangle的方面的责任

Return

接口隔离原则

ISP(The Interface Segregation Principle)

Interface segregation principle (接口分离原则)

- **接口分离原则的定义：客户类不应该被迫依赖于它不需要的方法。**
  The interface segregation principle (ISP) states that no client (class) should be forced to depend on methods it does not use.

例6. 宠物类的设计

Pet
-dog: String -cat: String -parrot: String
+bark(): void +dogJump(): void +mew(): void +climbTree(): void +parrotTalk(): void +parrotFly(): void

接口污染

```
Public class Client{
    private static Pet pet = new Pet(null, “cat”, null);
    public static void main(String[] argsv){
        pet.mew();
        pet.climbTree();
    }
}
```

这个Client类只是关心Cat的行为，而不关心其它的宠物的行为。程序员要从Pet类中挑选出Cat的行为;将三种宠物都封装在一个类中不合情理。

- **接口分离原则的意义：根据ISP原则，将很大的接口拆分成较小的，更具体的接口；使得客户类只需要知道它所感兴趣的方法。**
- **ISP原则的意图是使得一个系统保持较低的耦合，以便于更容易重构，修改与部署。**

改善设计1

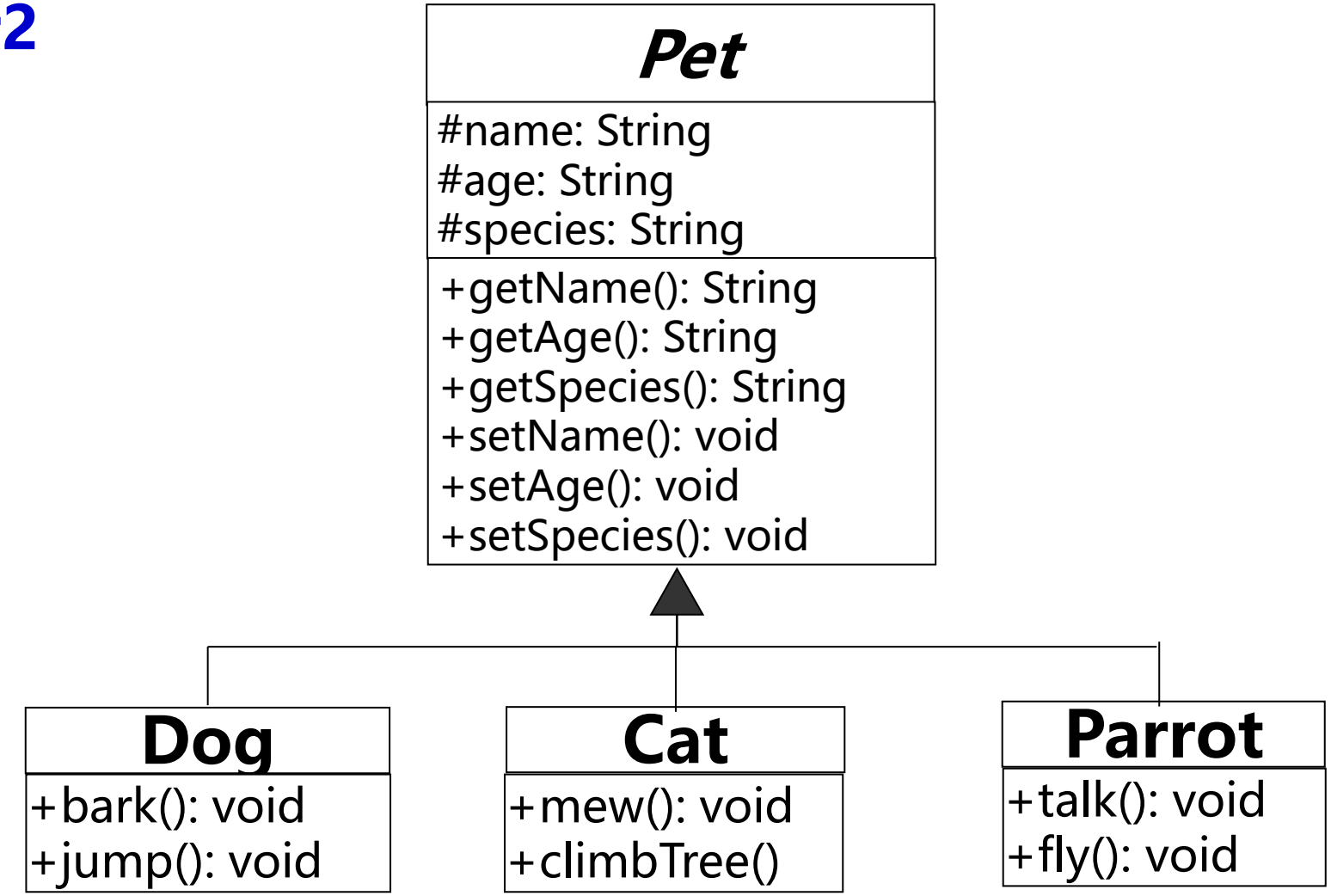
Dog
#name: String #age: String #species: String
+getName(): String +getAge(): String +getSpecies(): String +setName(): void +setAge(): void +setSpecies(): void +bark(): void +jump(): void

Cat
#name: String #age: String #species: String
+getName(): String +getAge(): String +getSpecies(): String +setName(): void +setAge(): void +setSpecies(): void +mew(): void +climbTree(): void

Parrot
#name: String #age: String #species: String
+getName(): String +getAge(): String +getSpecies(): String +setName(): void +setAge(): void +setSpecies(): void +talk(): void +fly(): void

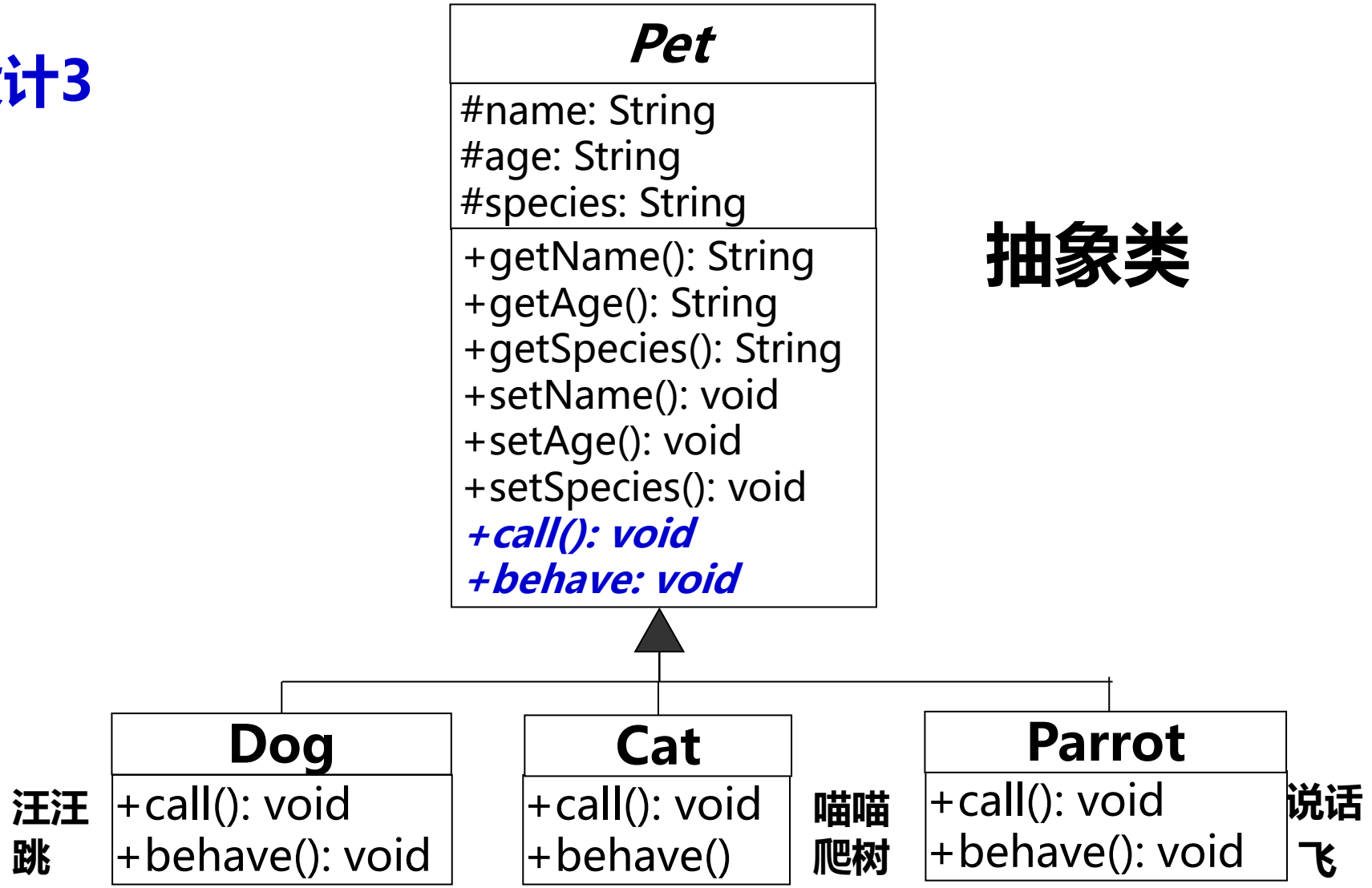
利用接口隔离原则，将原来的大接口分成3个具体的小接口：Dog，Cat与Parrot

改善设计2



设计成层次类，将原来的大接口分成三个小接口。符合接口隔离原则。

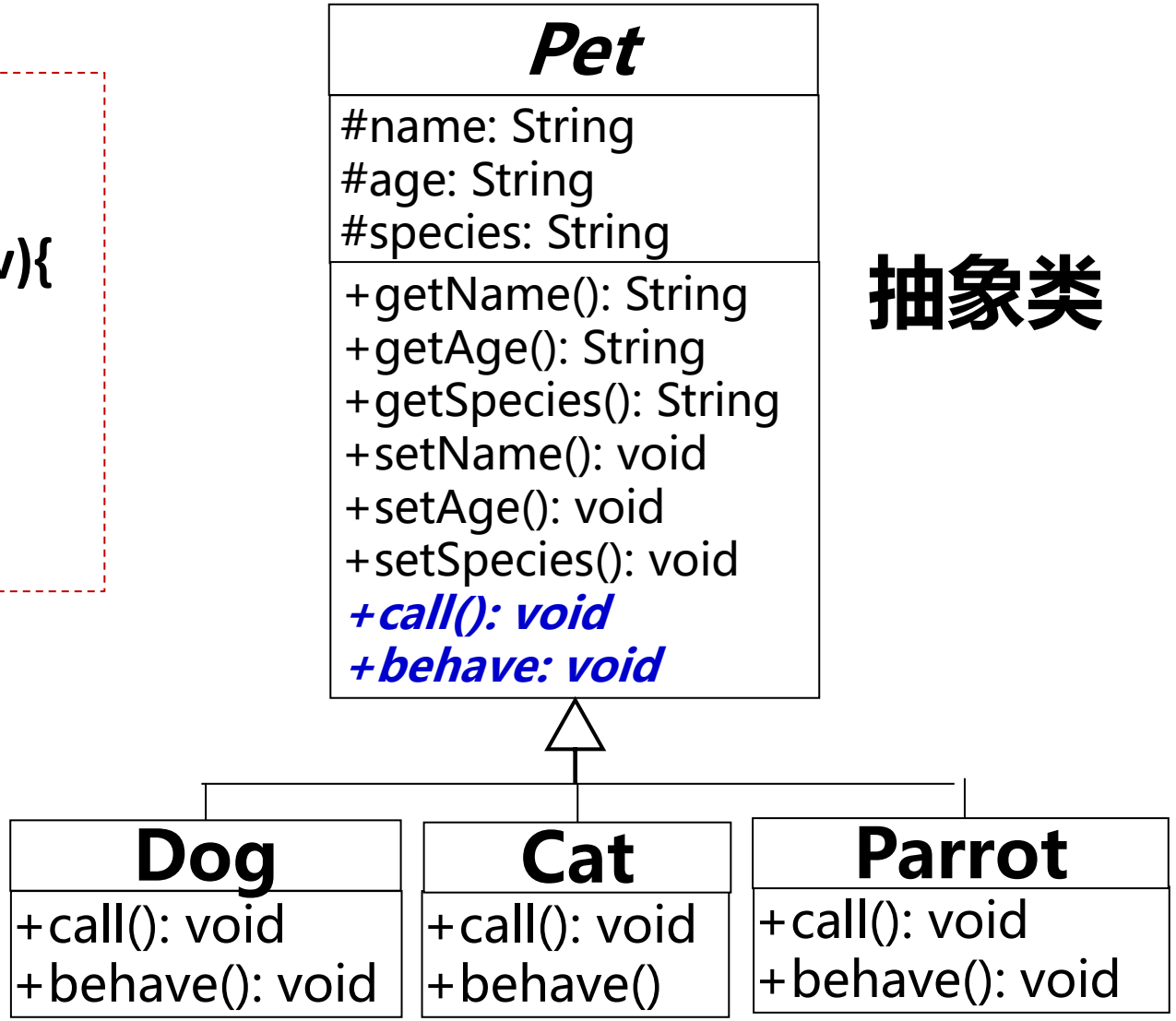
改善设计3



抽象类

设计成层次类。强迫几个子类有共同行为； call() （（鸣）叫），behave()（表现）是共同行为；几个子类对它们的实现各不相同。

```
Public class Clent{
    private static Pet pet = new Cat();
    public static void main(String[] argsv){
        pet.call();
        pet.behave();
    }
}
```

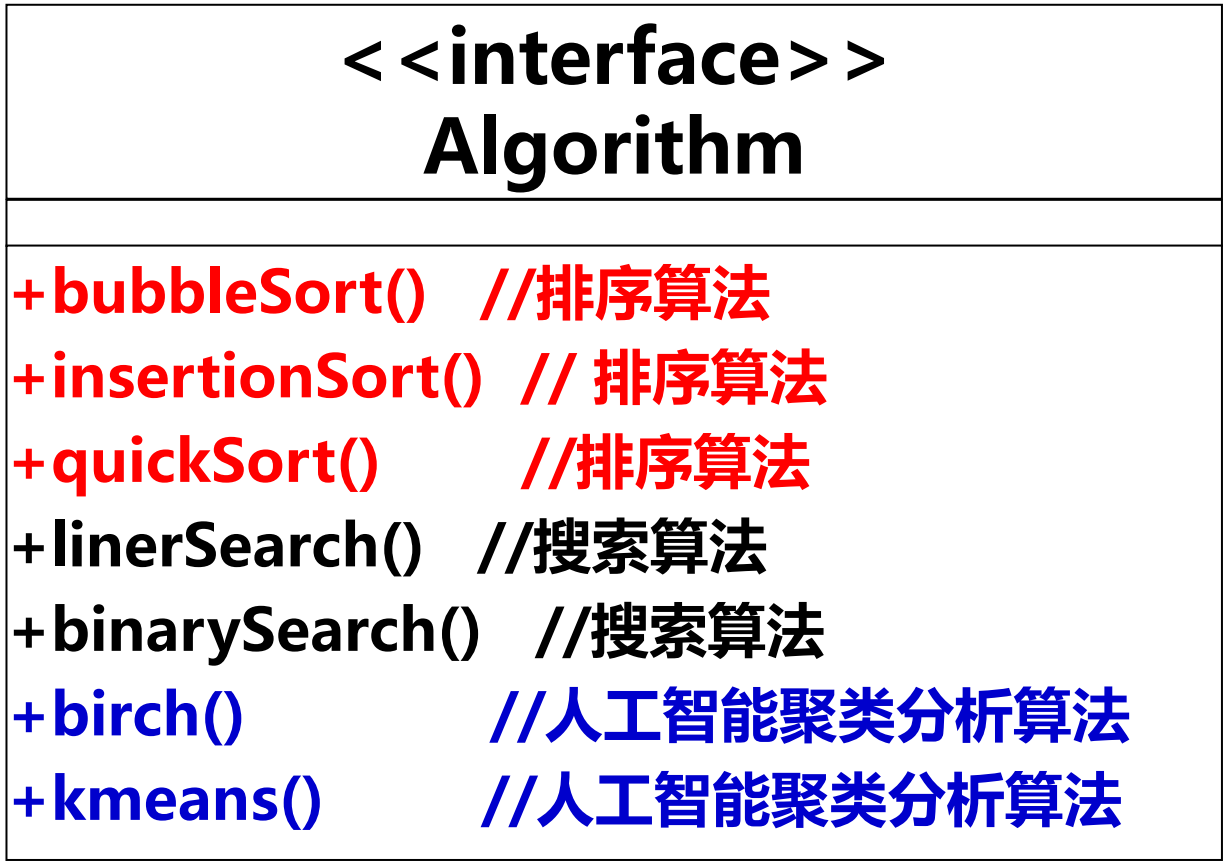


抽象类

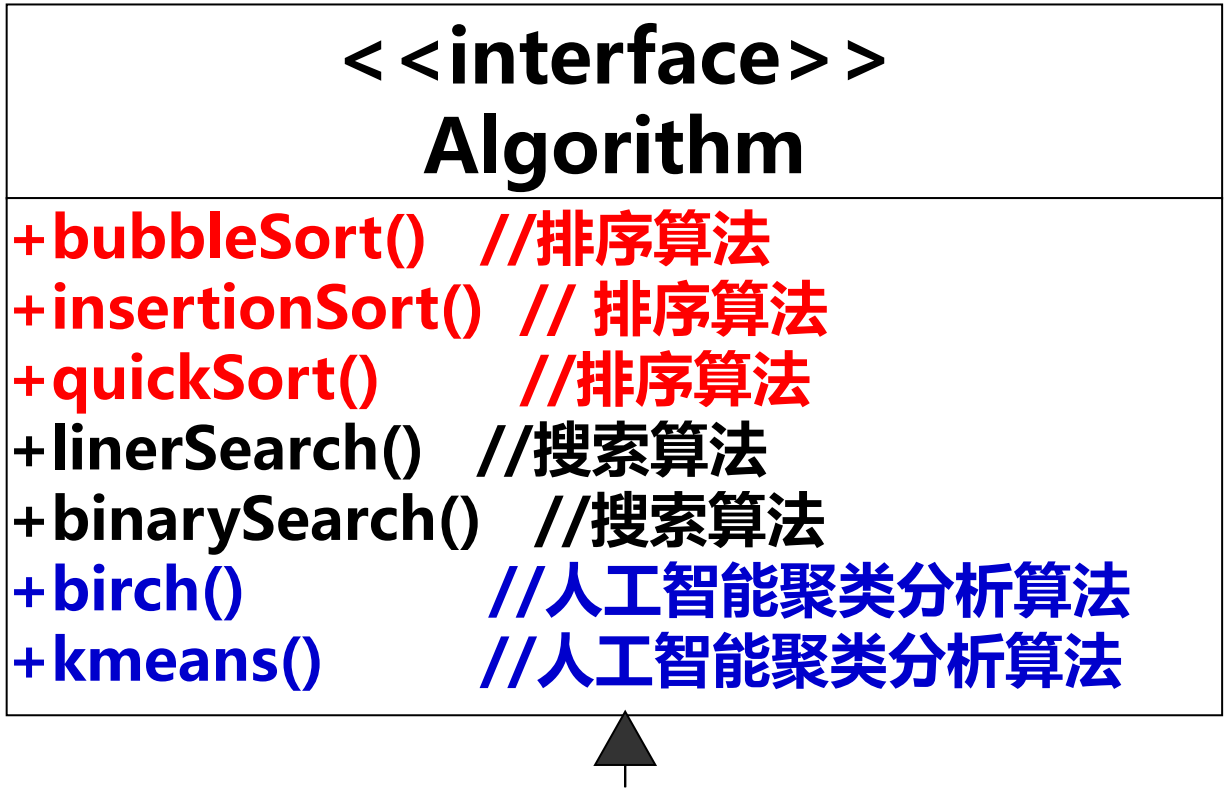
优点：

1. **符合接口隔离原则；**
2. **Pet层次类符合开-闭原则，若要增加一种宠物，只需增加一个相应的子类即可，即新增功能不必修改原有的代码**
3. **运用多态，不同的动物对方法call(),behave()有不同的实现**

例7：西门吹雪先生设计的Java接口类。包含了一些算法。



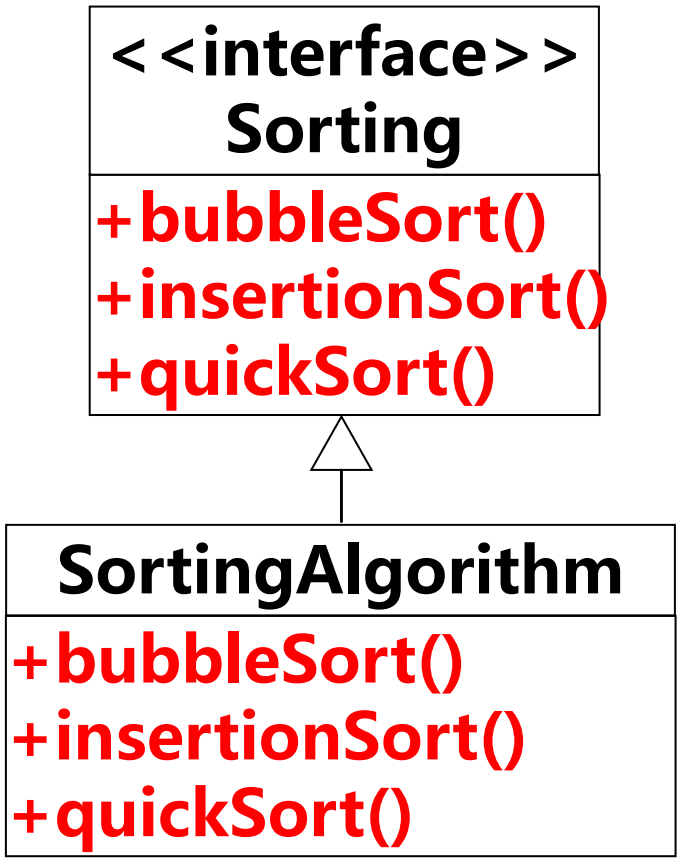
开发团队要
实现这个接口类。



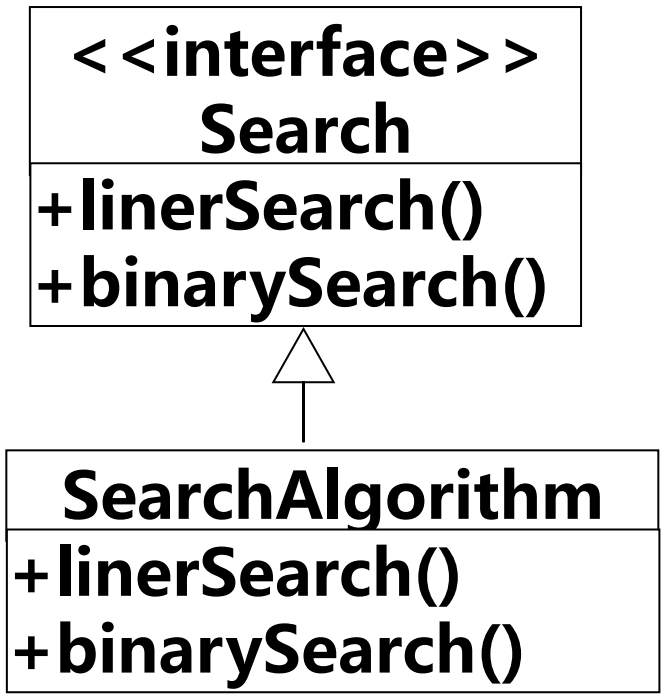
- ClientA类要使用到排序算法，因此要写一个Sorting类来实现超类接口。
- 语法要求7个算法都要在Sorting类里面实现。不合理。

欧阳小明的设计：将那个大接口拆分成三个小接口。

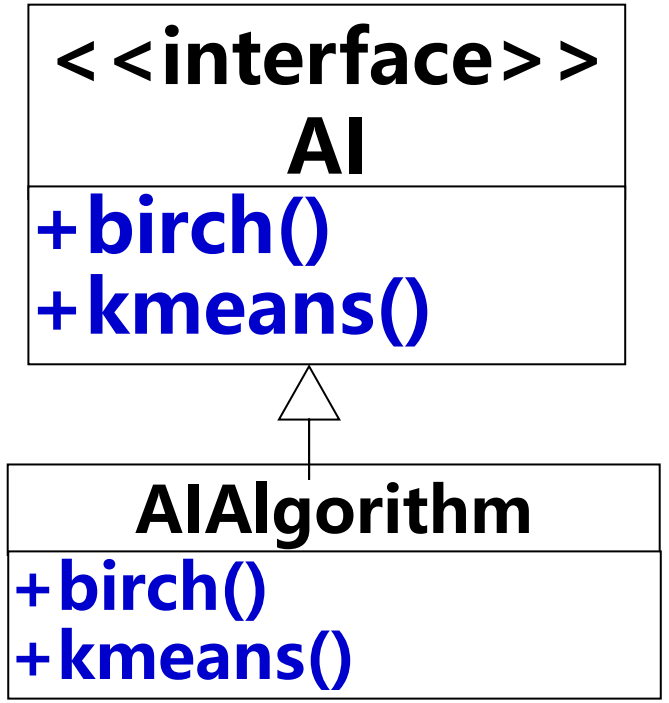
可以换为Java抽象类



可以换为Java抽象类



可以换为Java抽象类



ISP本质

- 使用多个专门的接口比使用单一的接口好
- 一个类对另一个类的依赖性应当是建立在最小的接口上的
- 避免接口污染(Interface Pollution)

Return

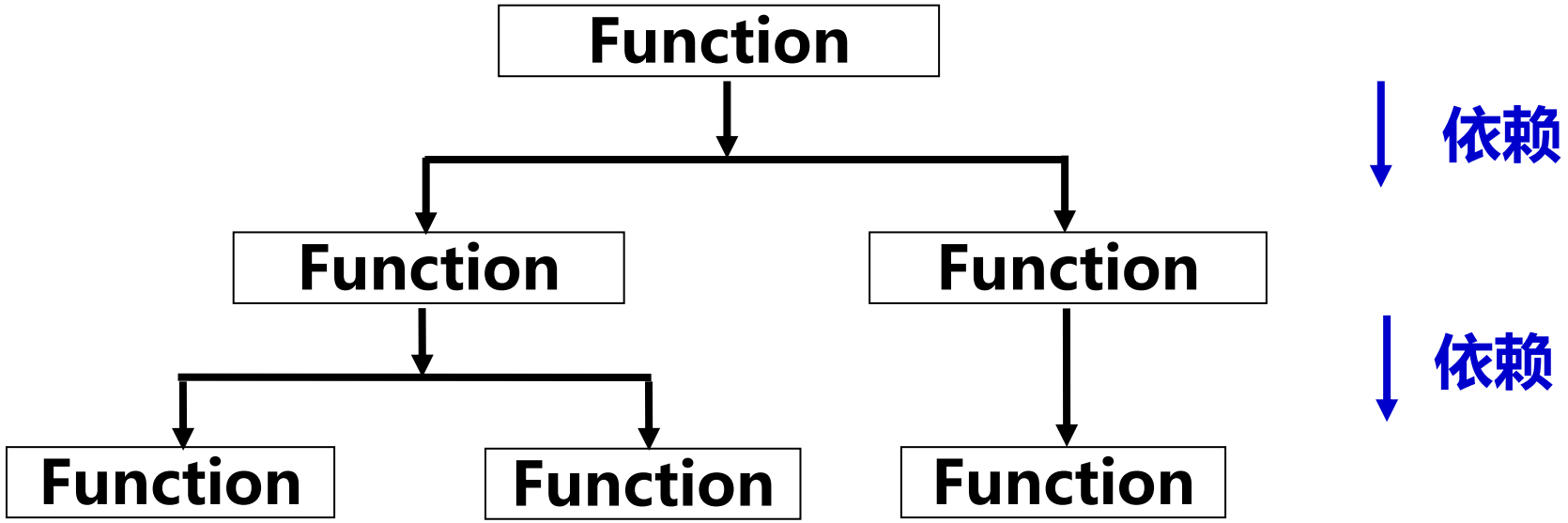
依赖倒置原则

(DIP-The Dependency Inversion Principle)

DIP(依赖倒置原则)

- DIP(依赖倒置原则，The Dependency Inversion Principle)
  - 高层模块不依赖于低层模块，二者都依赖于抽象
  - 抽象不依赖于细节，细节依赖于抽象
  - 针对接口编程，不要针对实现编程
  - 又称控制反转(IoC，Inversion of Control)、依赖注入

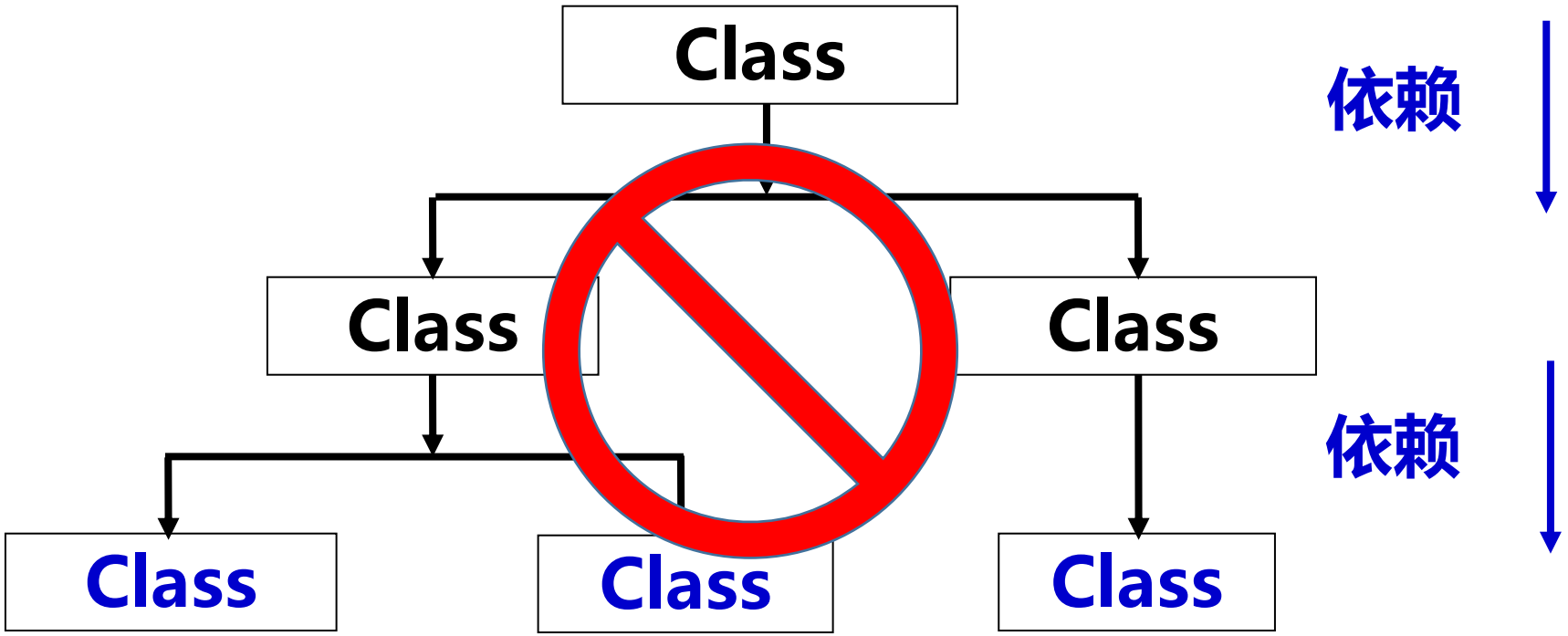
- **结构化设计中的依赖关系**
- 在结构化设计中，较低级别的组件被设计为被较高级别的组件调用，从而能够构建越来越复杂的系统。
- 高层组件直接依赖于低级组件来完成某些任务。
- 这种对低级组件的依赖性限制了高级组件的重用机会。



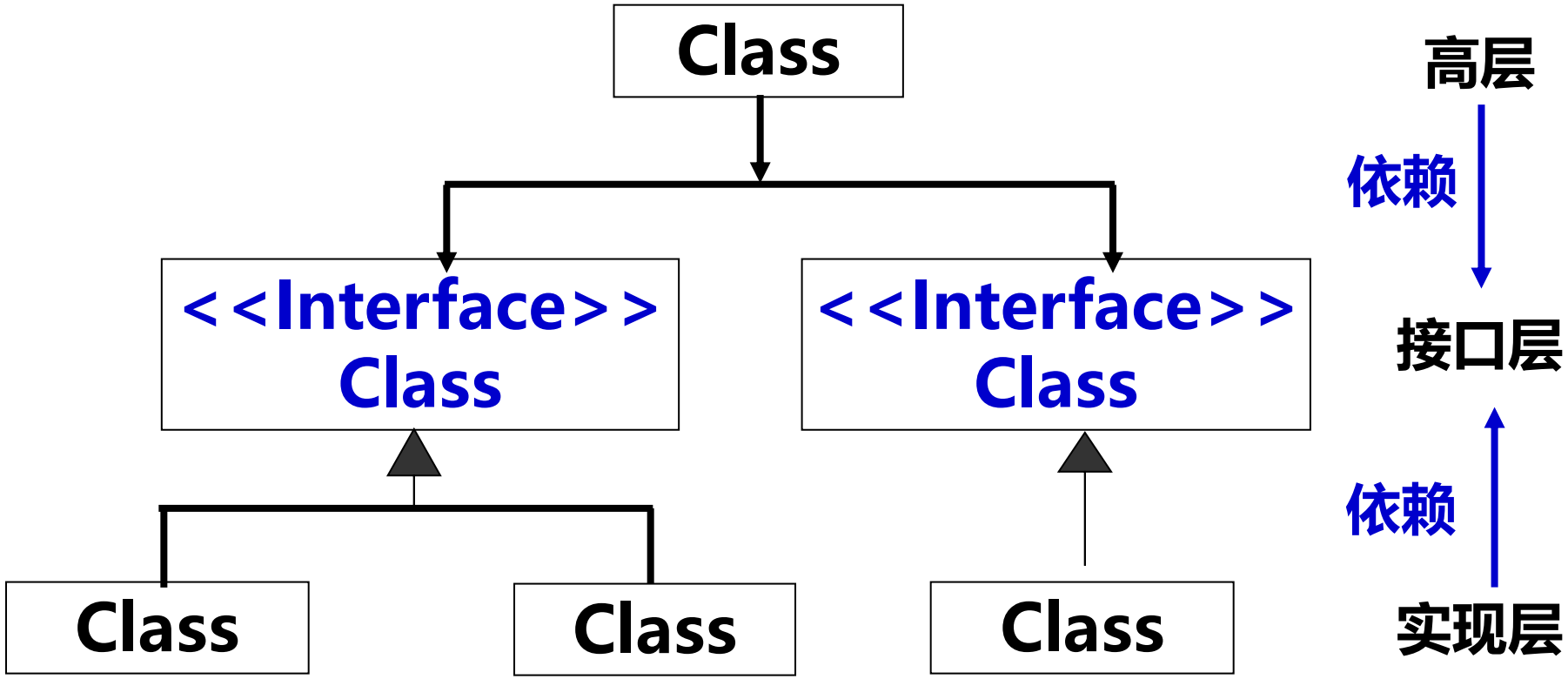
结构化设计-高层函数依赖于低层函数

面向对象设计中-某些人不良的设计习惯：

- 在设计中包含的全是 “独类”；
- 高层类依赖于低层类



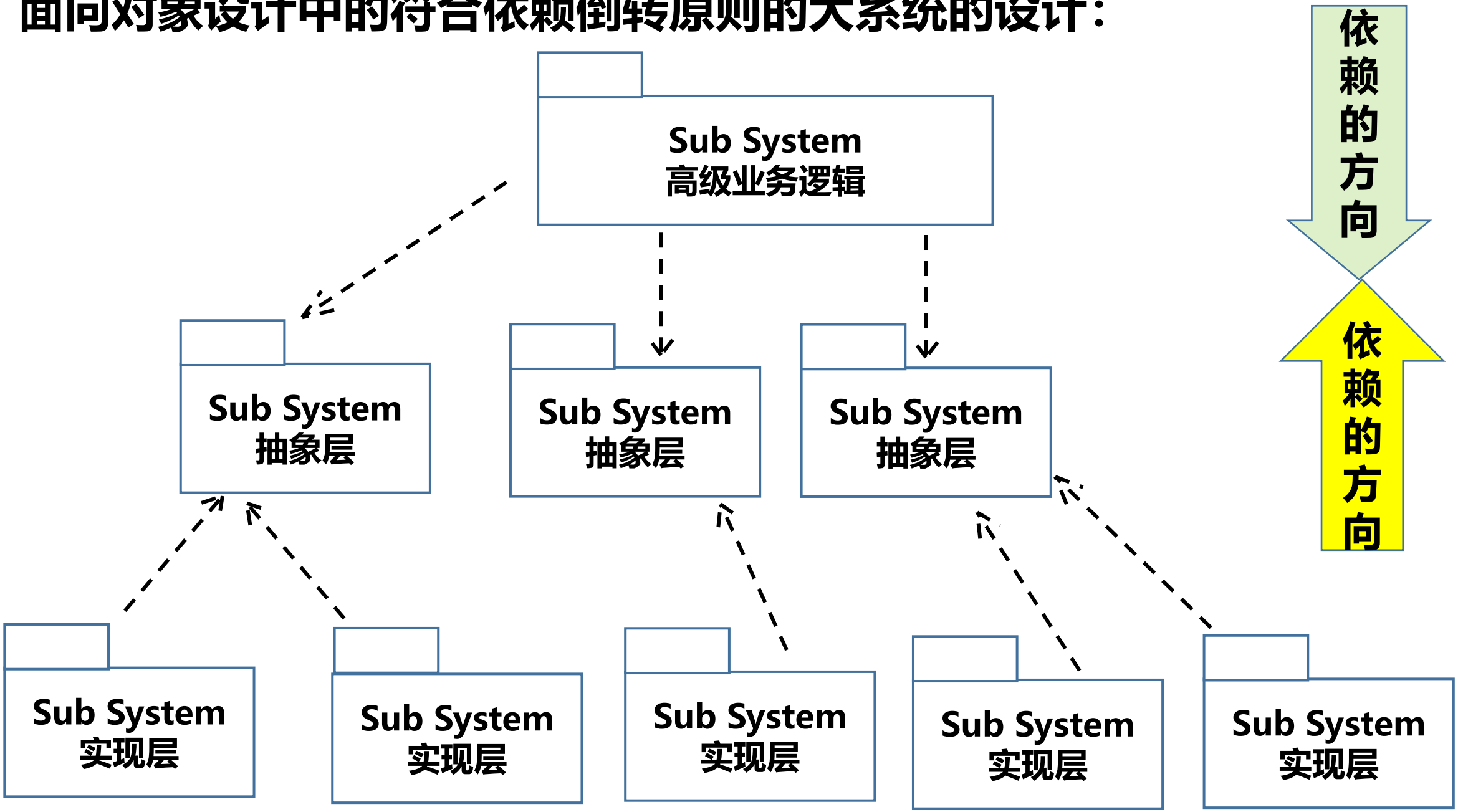
面向对象设计中的符合依赖倒转原则的小程序的设计：



在你的设计中，应该尽量包含这样的设计。但是，不可能将所有的类都设计成这个样子。

高层逻辑与低层都依赖于接口-依赖倒转

面向对象设计中的符合依赖倒转原则的大系统的设计：



启发式原则

- “依赖于抽象”—程序中所有依赖关系都应该终止于抽象类或者接口
- 启发式原则：
  - 任何变量都不应该拥有指向具体类的指针或者引用
  - 任何类都不应该从具体类派生 (可以继承抽象类，实现接口类)
  - 任何方法都不应该改写其任何基类中已经实现的方法

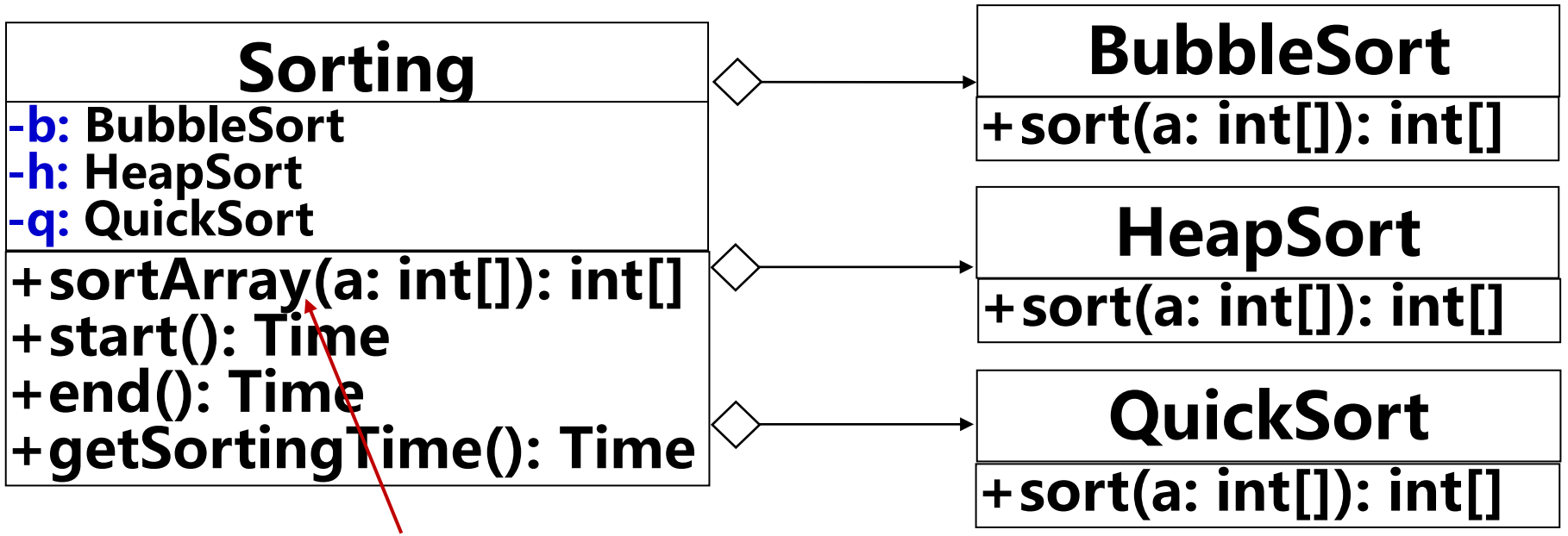
- **哪种好的设计思路导致依赖倒转?**
  - **OO设计鼓励使用抽象接口来定义方法，并让其子类来实现方法**
  - **不同的子类可以以不同的方式实现接口。**
  - **增加通过让子类实现中间层接口，使下层依赖于中间层**
  - **实现类(子类)的复用变得更加容易。**

结构化设计与面向对象设计的区别：

- **结构化设计**，上层调用下层，上层依赖于下层，当下层剧烈变动时上层也要跟着变动，这就会导致模块的复用性降低而且大大提高了开发的成本。
- **面向对象设计中的依赖倒转**：一般情况下抽象的变化概率很小，让客户类依赖于抽象，实现的细节也依赖于抽象。
- **即使实现细节不断变动，只要抽象不变，客户程序也不需要变化。**这大大降低了客户程序与实现细节的耦合度。

- **例8. 排序算法类的设计**
- **假设你们组为教授数据结构的老师设计了一款排序软件。对排序算法的速度进行比较。首先实现以下算法**
  - **冒泡法（bubble sort）排序**
  - **堆排序法（heap sort）排序**
  - **快速（quick sort）排序**
- **西门吹雪同学给出了如下的设计。**

西门吹雪 先生的设计



sortArray方法中含有很多条件语句，判断到底是BubbleSort，HeapSort还是 QuickSort？然后再调用相应的类的sort方法。

问题：Sorting直接依赖于3个具有同一接口的具体的类BubbleSort，HeapSort和QuickSort，违反了面向对象的依赖倒转原则。

设计缺点：

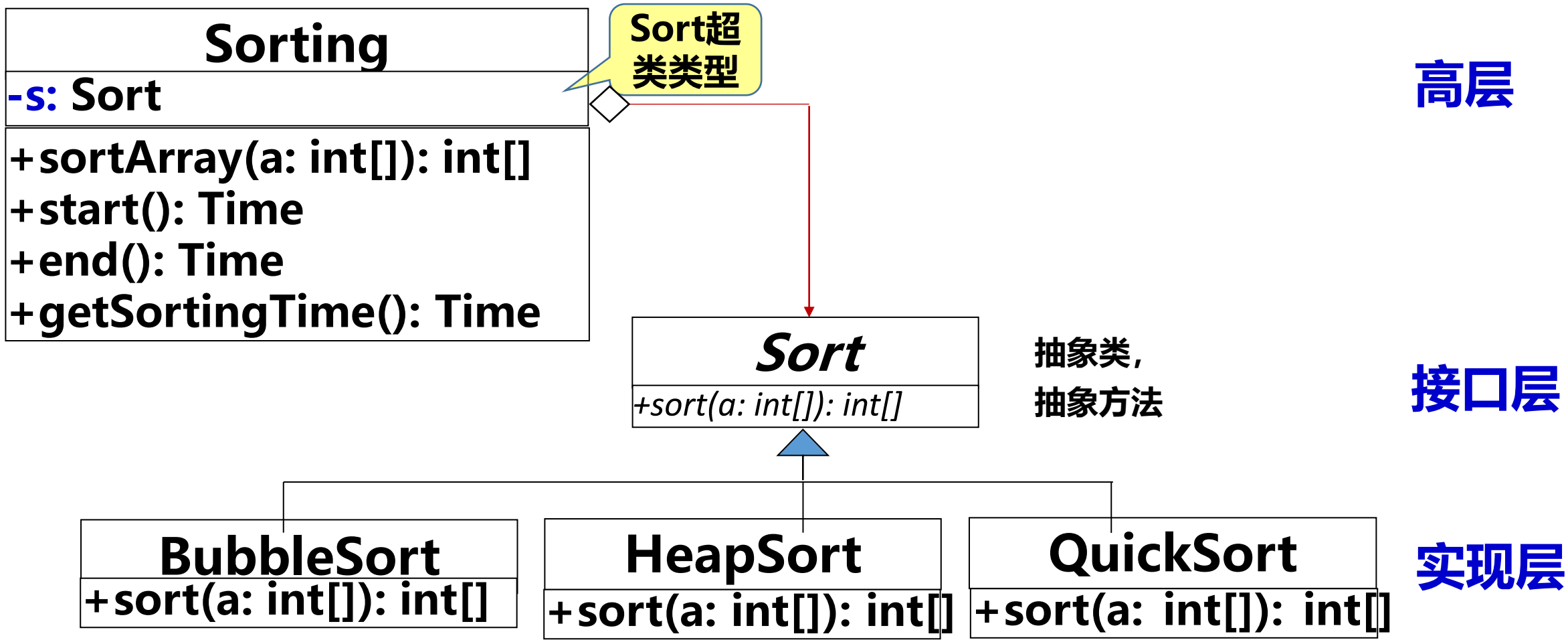
a) 随着时间推移，会有更多的新型排序算法加入到Sorting中，该类将频繁增加条件语句；

b) 可扩展性不好，增加一种新排序算法需要修改Sorting类，难以扩展与维护。

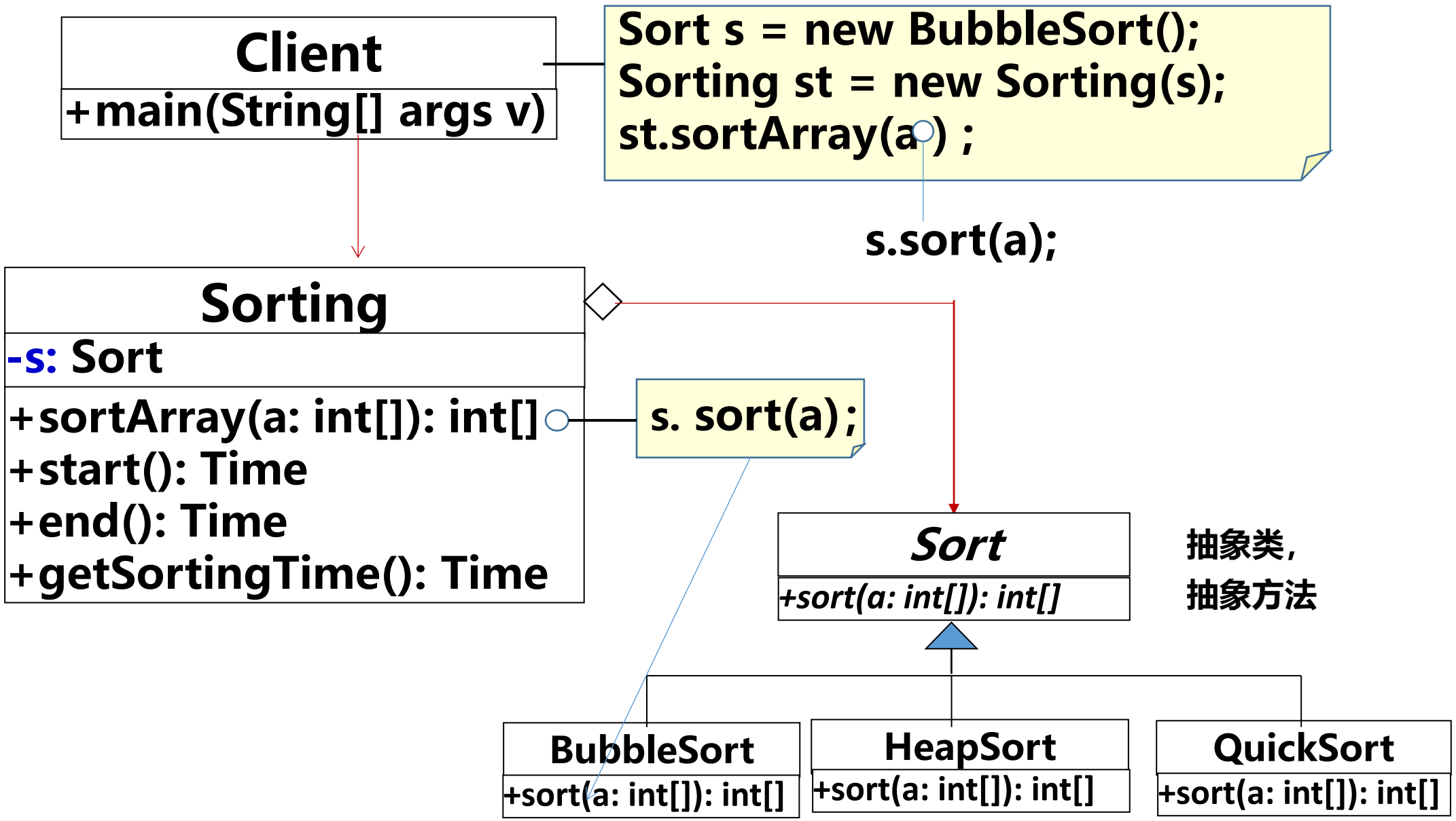
原因： Sorting类依赖于它所控制的低层的具体细节的类 BubbleSort，HeapSort与 QuickSort

重新设计如下

大师哥 欧阳智慧 的重新设计如下：

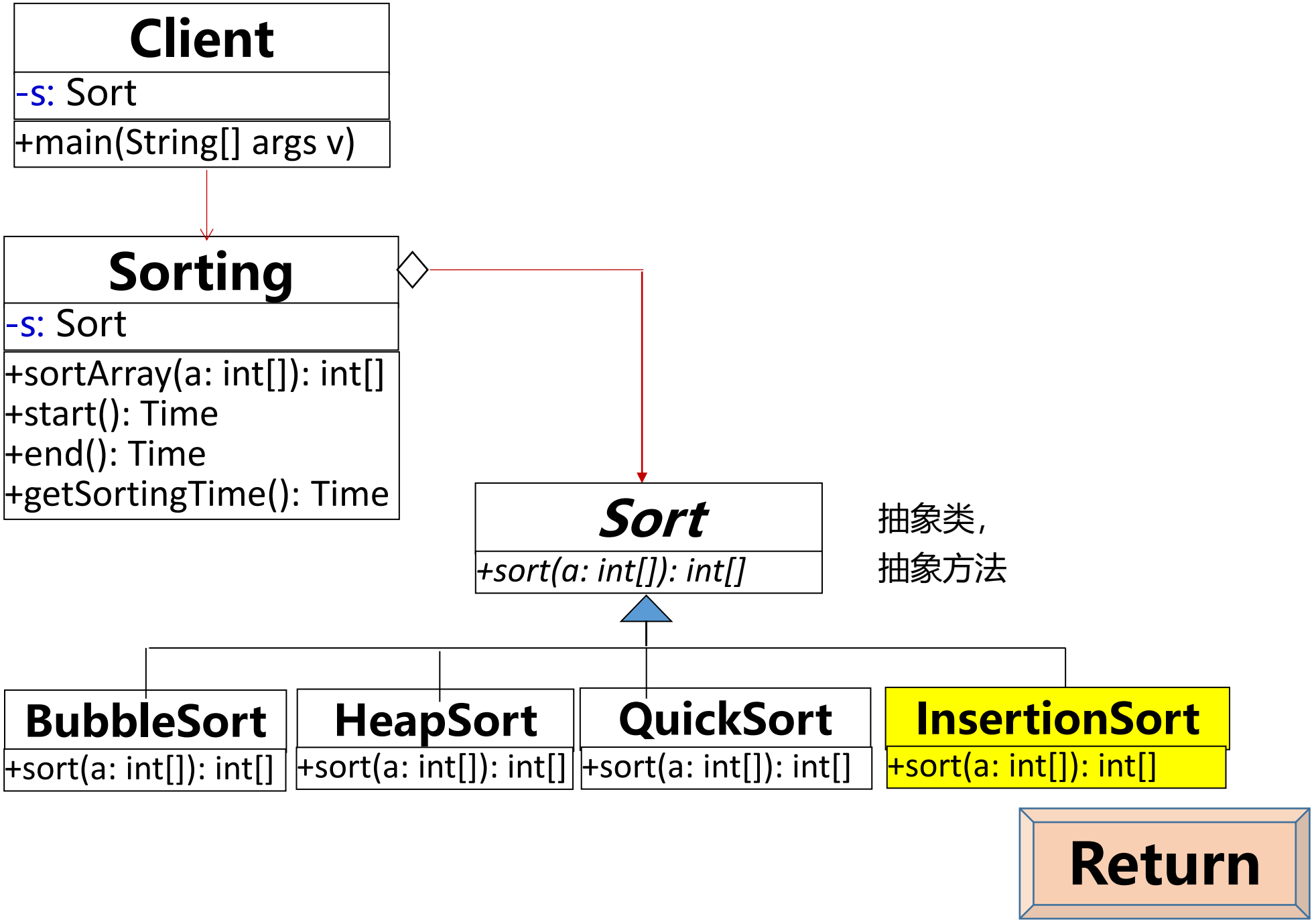


新设计的典型交互情况：


Client
+main(String[] args v)
Sort s = new BubbleSort();
Sorting st = new Sorting(s);
st.sortArray(a) ;
s.sort(a);
Sorting
-s: Sort
+sortArray(a: int[]): int[]
+start(): Time
+end(): Time
+getSortingTime(): Time
s. sort(a);
Sort
+sort(a: int[]): int[]
抽象类，
抽象方法
BubbleSort
+sort(a: int[]): int[]
HeapSort
+sort(a: int[]): int[]
QuickSort
+sort(a: int[]): int[]

新设计说明：将排序类Sort设计成一个层次类；超类有三个具有同一接口的具体的子类BubbleSort，HeapSort和QuickSort。子类实现超类的抽象方法sort.

- **优点**:
- Sorting仅仅依赖于**Sort接口**；符合面向对象的依赖倒转原则
- 可扩展性好：增加一个新算法， Sorting 类不需要修改代码；该设计还符合开闭原则
- 可维护性好：哪个算法需要修改代码，只需要修改相应的子类即可


Client
-s: Sort
+main(String[] args v)
Sorting
-s: Sort
+sortArray(a: int[]): int[]
+start(): Time
+end(): Time
+getSortingTime(): Time
Sort
+sort(a: int[]): int[]
抽象类，
抽象方法
BubbleSort
+sort(a: int[]): int[]
HeapSort
+sort(a: int[]): int[]
QuickSort
+sort(a: int[]): int[]
InsertionSort
+sort(a: int[]): int[]
Return

The end