第3章 面向对象的设计原则

软件设计中存在的问题

- 过于僵硬（Rigidity）
  - 很难加入新功能
- 过于脆弱（Fragility）
  - 很难修改
- 复用率低（Immobility）
  - 高层模块无法重用
- 黏度过高（Viscosity）
  - 破坏原始设计框架

什么是好的设计？

- 一个好的系统设计应该有如下性质：
  - 可扩展性
  - 灵活性
  - 可插入性

—Peter Code [CODE99]

设计目标

- 可扩展性（**Extensibility**）
  - 容易添加新的功能
- 灵活性（**Flexibility**）
  - 代码修改平稳地发生
- 可插入性（**Pluggability**）
  - 容易将一个类抽出去，同时将另一个有同样接口的类加入进来

面向对象的设计原则

- OCP：开-闭原则:
  - 对可变性封装
- SRP：单一职责原则
  - 如何划分职责
- LSP：里氏代换原则
  - 如何进行继承
- DIP：依赖倒转原则
  - 针对接口编程
- ISP：接口隔离原则
  - 恰当的划分角色和接口
- CRP：合成复用原则
  - 尽量使用合成/聚合而不使用继承复用
- LoD：迪米特原则
  - 不要跟陌生人说话
- 其他设计原则

目标与原则的关系

- 可扩展性（Extensibility）
  - “开—闭”原则、里氏替换原则、依赖倒转原则、合成/聚合复用原则
- 灵活性（Flexibility）
  - “开—闭”原则、Demeter法则、接口隔离原则
- 可插入性（Pluggability）
  - “开—闭”原则、里氏替换原则、依赖倒转原则、合成/聚合复用原则

开放—封闭原则

Open-Closed Principle

定义

- 软件实体（类，模块，函数，等等）应该尽可能允许扩展，同时尽可能避免被更改。
- **SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.**

—*Bertrand Meyer 1988*

任何软件系统都会发生变化

- 任何软件系统在其生命周期中都会发生变化。如果我们不希望开发出的系统第一版本后就被抛弃，那么我们就必须牢牢记住这一点。

—IvarJacobson

OCP特征

- 可扩展（对扩展是开放的）
  - 模块的行为功能可以被扩展，在应用需求改变或需要满足新的应用需求时，我们可以让模块以不同的方式工作
- 不可更改（对更改是封闭的）
  - 这些模块的源代码是不可改动的
  - 任何人都不许修改模块的源代码
- 自相矛盾？

太玄经

知固而不知革，物失其则
知革而不知固，物失其均

- 固：close for modification
- 革：open for extension

太玄经

- 一个系统对修改关闭，就是”固”
- 而一个系统对扩展开放，就是”革”
- 一个系统不可拓展，就会“物则失则”，或者说系统无法发展
- 而一个系统动则需要修改，便会“物失其均”，也就是失去其重心

OCP的关键

- **OCP的关键是**

抽象！

- **由抽象可以预见所有可能的扩展（闭）:模块可以操作一个抽象体，由于模块依赖于一个固定的抽象体，因此它对修改是封闭的(closed for modification)**
- **由抽象可以随时导出新的类（开）:同时，通过从这个抽象体派生，又可扩展此模块的行为和功能(open for extension)**

OCP的关键

- 符合**OCP**原则的程序只通过增加代码来变化而不是通过更改现有代码来变化，因此这样的程序就不会引起象非开放—封闭的程序那样对变化的连锁反应

OCP实例(1)

- 思考：如何在程序中模拟用手去开门和关门？
- 行为：
  - 开门
  - 关门
  - 判断门的状态

门

```
public class Door1 {
    private boolean _isOpen=false;
    public boolean isOpen(){
        return _isOpen;
    }
    public void open(){
        _isOpen = true;
    }
    public void close(){
        _isOpen = false;
    }
}
```

手

```
public class Hand1 {
    public Door1 door;
    void act() {
        if (door.isOpen())
            door.close();
        else
            door.open();
    }
}
```

主程序

```
public class Main1 {
    public static void main(String[] args) {
        Hand1 myHand = new Hand1();
        myHand.door = new Door1();
        myHand.act();
    }
}
```

新的问题

- 需要手去开关抽屉，冰箱……？
- 我们只好去修改程序……

抽屉

```
public class Drawer1 {
    private boolean _isOpen=false;
    public boolean isOpen(){
        return _isOpen;
    }
    public void open(){
        _isOpen = true;
    }
    public void close(){
        _isOpen = false;
    }
}
```

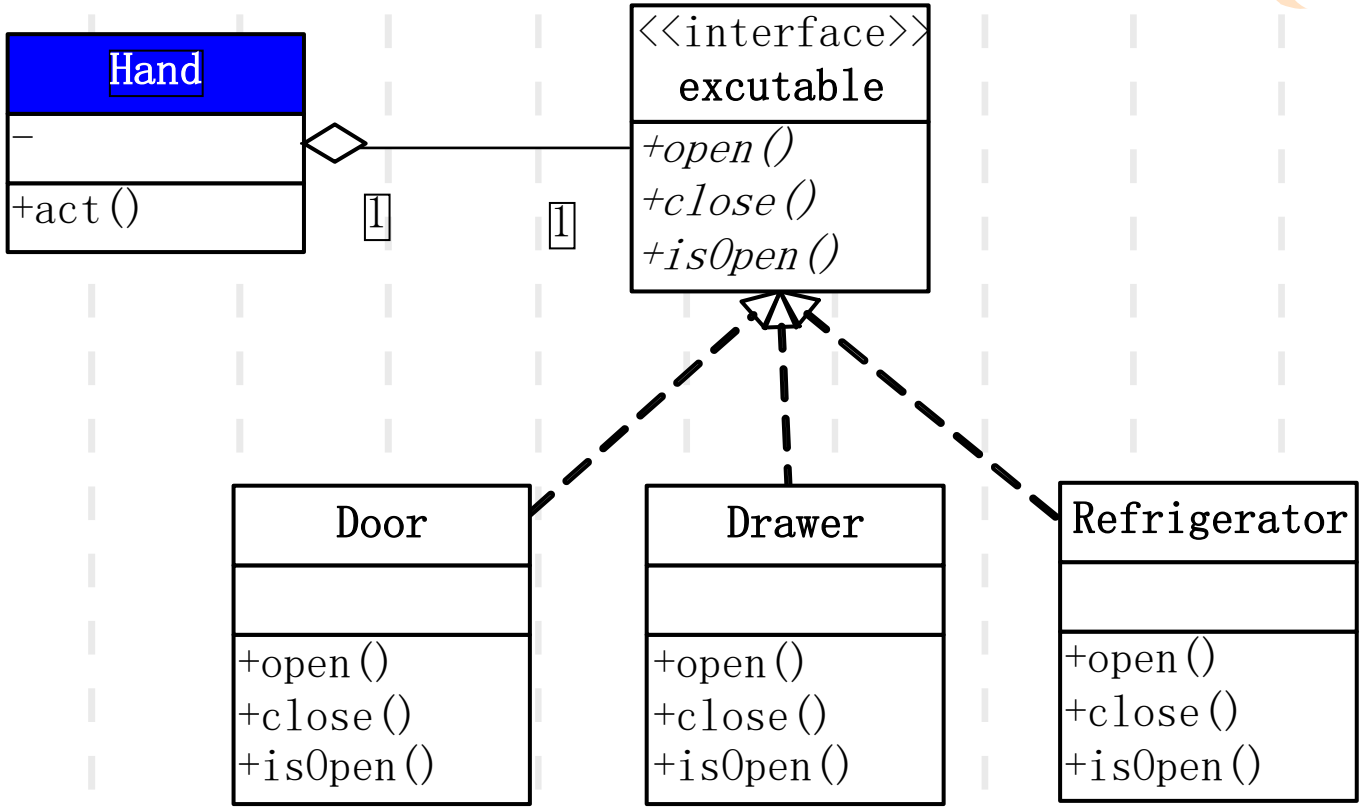
手（注意：被改变了！）

```
public class Hand1 {
    public Door1 door;
    public Drawer1 drawer;
    public int item=1;
    void act() {
        switch (item){
            case 1:
                if (door.isOpen())
                    door.close();
                else
                    door.open();
                break;
            case 2:
                if (drawer.isOpen())
                    drawer.close();
                else
                    drawer.open();
                break;
        }
    }
```

主程序

```
public class Main1 {
    public static void main(String[] args) {
        Hand1 myHand = new Hand1();
        myHand.door = new Door1();
        myHand.item = 1;
        myHand.act();
    }
}
```

符合OCP的设计


Hand
-
+act()
1
1
<<interface>>
excutable
+open()
+close()
+isOpen()
Door
+open()
+close()
+isOpen()
Drawer
+open()
+close()
+isOpen()
Refrigerator
+open()
+close()
+isOpen()

“可开关的”接口

```
public interface Excutable {
    public boolean isOpen();

    public void open();

    public void close();
}
```

门（新的）

```
public class Door2 implements Excutable {
    private boolean _isOpen = false;
    public boolean isOpen() {
        return _isOpen;
    }
    public void open() {
        _isOpen = true;
    }
    public void close() {
        _isOpen = false;
    }
}
```

抽屉（新的）

```
public class Drawer2 implements Excutable {
    private boolean _isOpen = false;
    public boolean isOpen() {
        return _isOpen;
    }
    public void open() {
        _isOpen = true;
    }
    public void close() {
        _isOpen = false;
    }
}
```

手（新的）

```
public class Hand2 {
    public Excutable item;

    void act() {
        if (item.isOpen())
            item.close();
        else
            item.open();
    }
}
```

主程序（新的）

```
public class Main2 {
    public static void main(String[] args) {
        Hand2 myHand = new Hand2();
        myHand.item = new Door2();
        myHand.act();
    }
}
```

选择性的封闭（Strategic Closure）

- 通常情况下，没有任何一个大的程序能够做到100%的封闭
- 一般来讲，无论模块是多么的“封闭”，都会存在一些无法对之封闭的变化
- 既然不可能完全封闭，因此就必须选择性地对待这个问题
- 也就是说，设计者必须对于他（她）设计的模块应该对何种变化封闭做出选择

对可变性的封装原则(EVP)

- 从另一方面看**OCP**就是所谓的“对可变性的封装原则”**(Principle of Encapsulation of Variation,**缩写为**EVP)**
- **[GOF95]**：”考虑你的设计中什么可能会发生变化，与通常将焦点放在什么会导致设计发生变化的思考方式相反，这一思路考虑的不是什么会导致设计变化，而是考虑你应当允许什么发生变化，而不让这一变化导致重新进行设计“
- **[Shall]**将这一思想总结为：“找出一个系统中的可变因素，并将其封装起来。”

单一选则原则(SCP)

- OCP的一个直接推论是单一选则原则(the Single Choice Principle)
- 即,如果一个软件系统必须支持一组备选项，理想情况下，在系统中应只有一个类知道整个备选项集合

OCP与设计模式

- 几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到“开—闭”原则的要求

单一职责原则

Single Responsibility Principle

定义

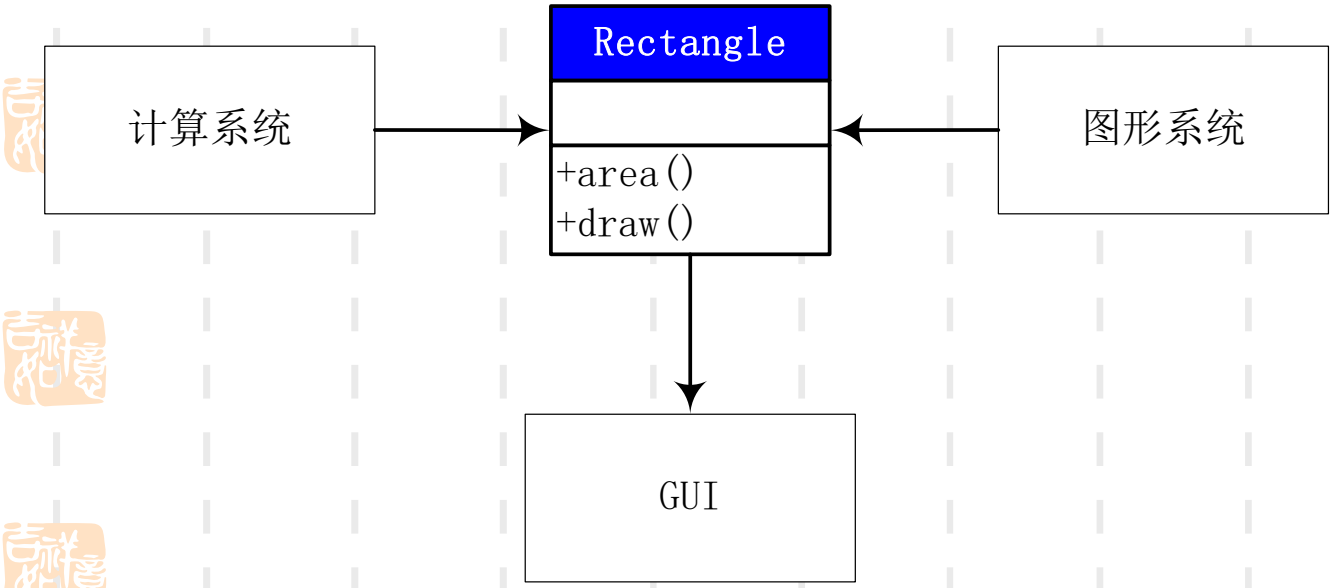
- 就一个类而言，应该仅有一个引起它变化的原因

实例：矩形系统

- 一个GUI系统的设计
- 要求：
  - 可以计算矩形的面积
  - 并在屏幕上显示

脆弱的设计

- 几何计算系统调用Rectangle计算面积
- GUI系统调用Rectangle绘制在屏幕上



出现的问题

- 编译几何计算系统时还需要编译进图形代码…
- 需要更换显示系统时还需要重新测试所有几何计算系统…
- 解决方法：将计算和绘制的职责分别放入**CulRectangle**和**GraphRectangle**中

结论

- 所谓一个类的职责是指引起该类变化的原因，如果一个类具有一个以上的职责，那么就会有多个不同的原因引起该类变化，其实就是耦合了多个互不相关的职责，将会降低这个类的内聚性

里氏替换原则

Liskov Substitution Principle

定义

- 一个软件实体如果使用的是父类的话，一定适用于子类型
- 软件设计时，子类应该设计成为子类型
- 子类型(sub type) vs. 子类(sub class)
- 反过来的替换不成立

LSP

- OCP原则背后的主要机制是抽象和多态
- 支持抽象和多态的关键机制是继承
- 当前存在的普遍的现象：

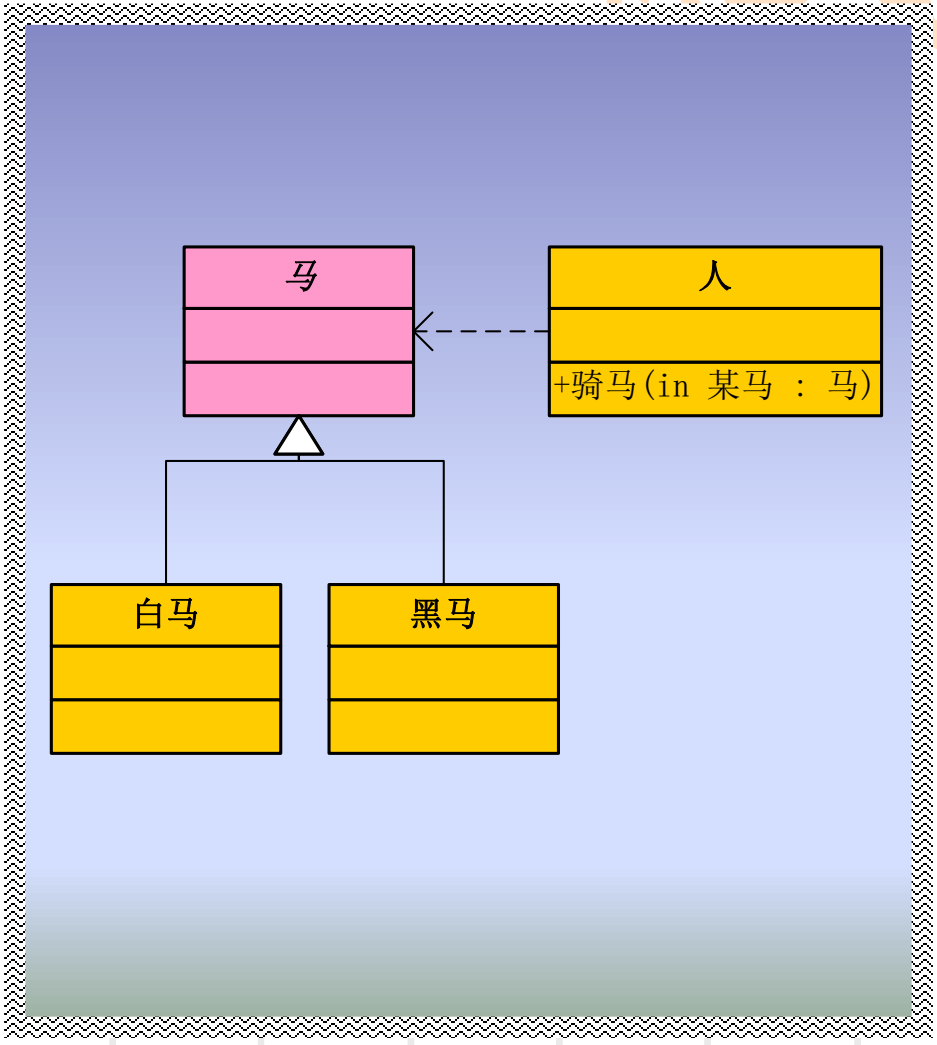
  滥用继承!!!

LSP的严格表达

- 如果对每一类型为T1的对象o1，都有类型为T2的对象o2，使得以T1定义的所有程序P在所有对象o1都替换为o2时，程序P的行为没有变化，那么类型T2是类型T1的子类型
- 换言之，一个软件实体如果适用于一个基类，那么也一定使用于其子类，而且它根本察觉不到基类对象与子类对象的区别
- LSP是继承复用的基石：只有当衍生类可以在软件单元的功能不受影响的前提下替换基类，基类才谈的上是真正被复用，而衍生类也才能在基类的基础上增加新的行为

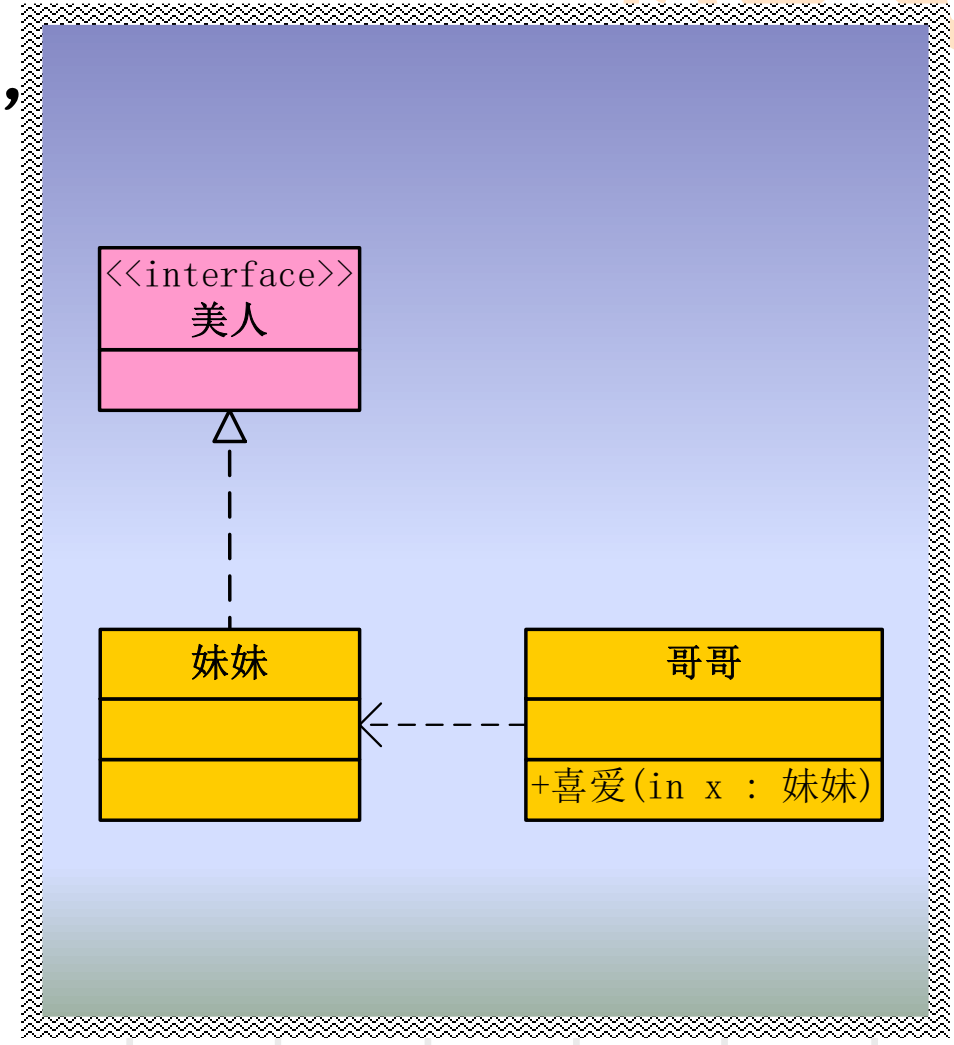
墨子论马

- 《墨子·小取》：白马马也；乘白马，乘马也。骊马，马也；乘骊马，乘马也。



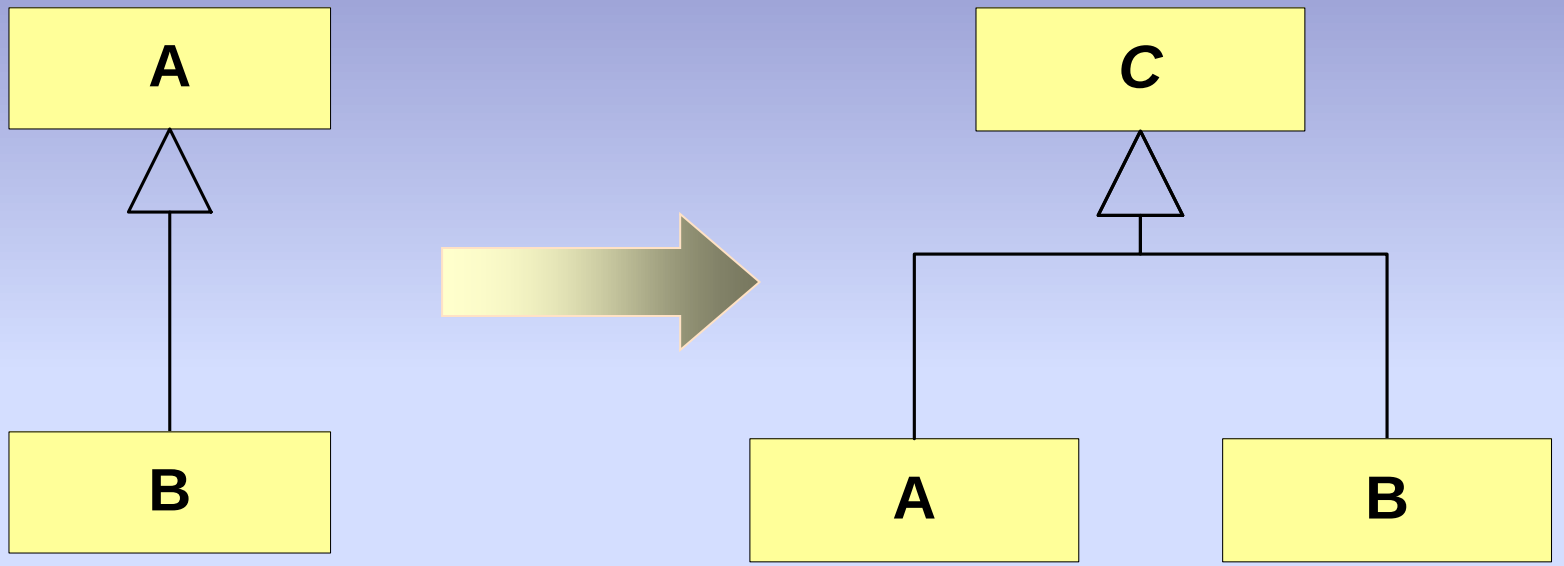
反过来的替换不成立

- 《墨子·小取》：“娣，美人也，爱娣，非爱美人也……盗，人也；恶盗，非恶人也。”

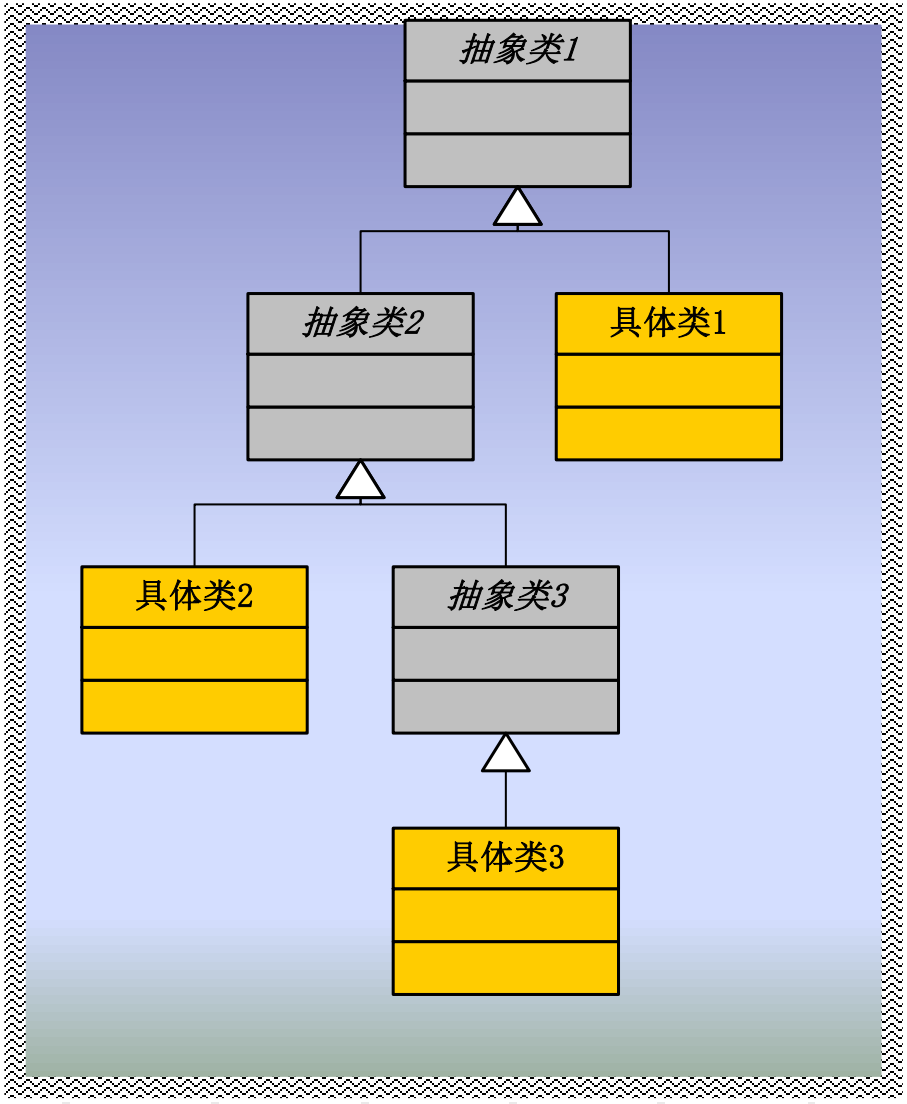


结论

- 在可能的情况下，由抽象类继承



再论抽象类与具体类



- 只要有可能，不要从具体类继承
- 代码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）

IS-A关系的再思考

- 鸵鸟是鸟吗？
- 是
  - 鸟有翅膀，鸵鸟也有翅膀
  - 鸟有喙，鸵鸟也有喙...
- 但是...

  鸟.getFlySpeed()

  鸵鸟.getRunSpeed()
- 有着不同

IS-A关系的再思考

- 对于动物学家

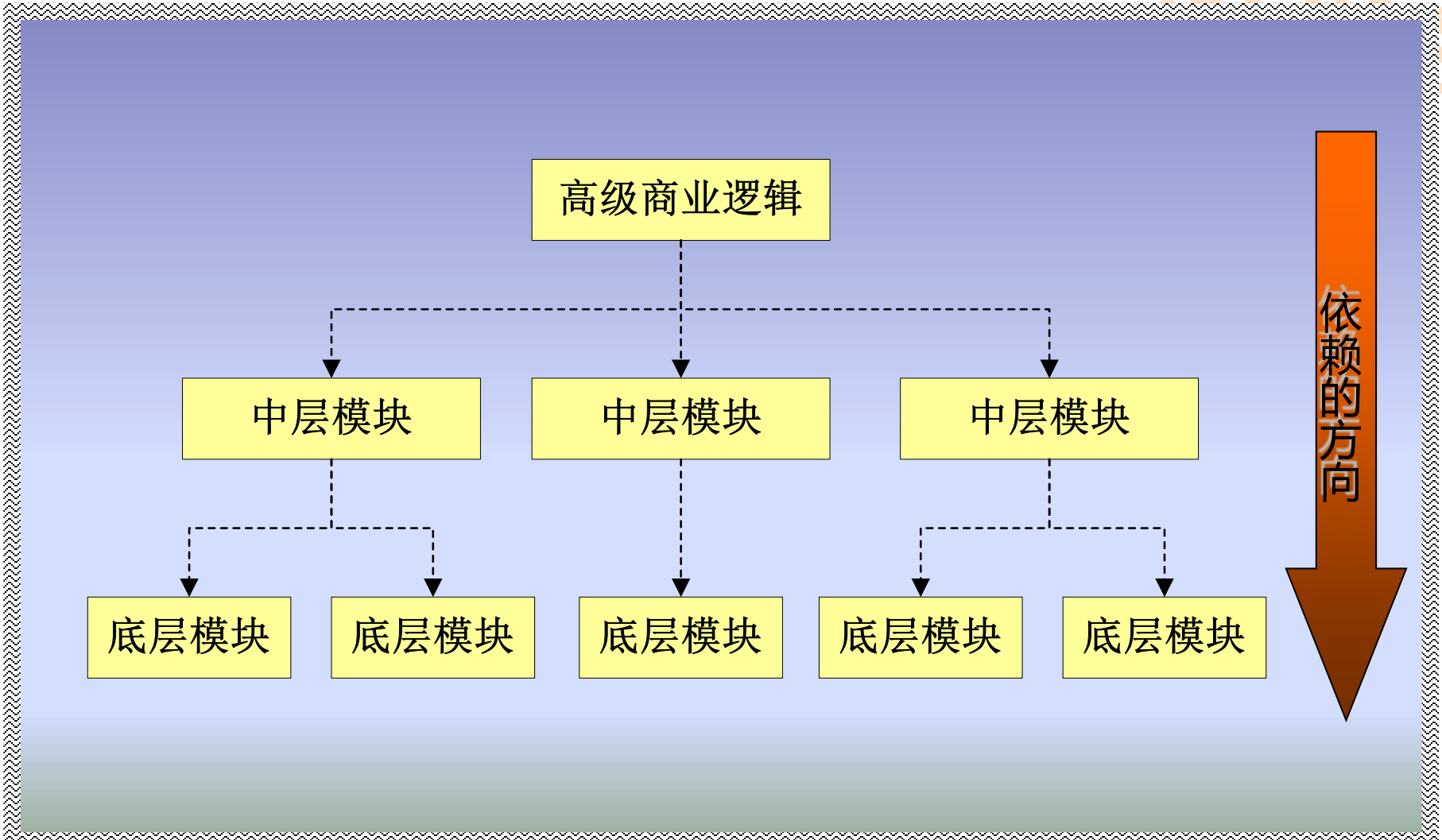
  只关心鸟的生理特征，对他们来说，鸵鸟就是鸟
- 对于养鸟人

  关心鸟的行为特征，鸵鸟不是鸟
- 他们都正确
- 考虑一个特定设计是否恰当时，不能完全孤立地看这个解决方案，应该根据设计的使用者提出的合理假设来审视。

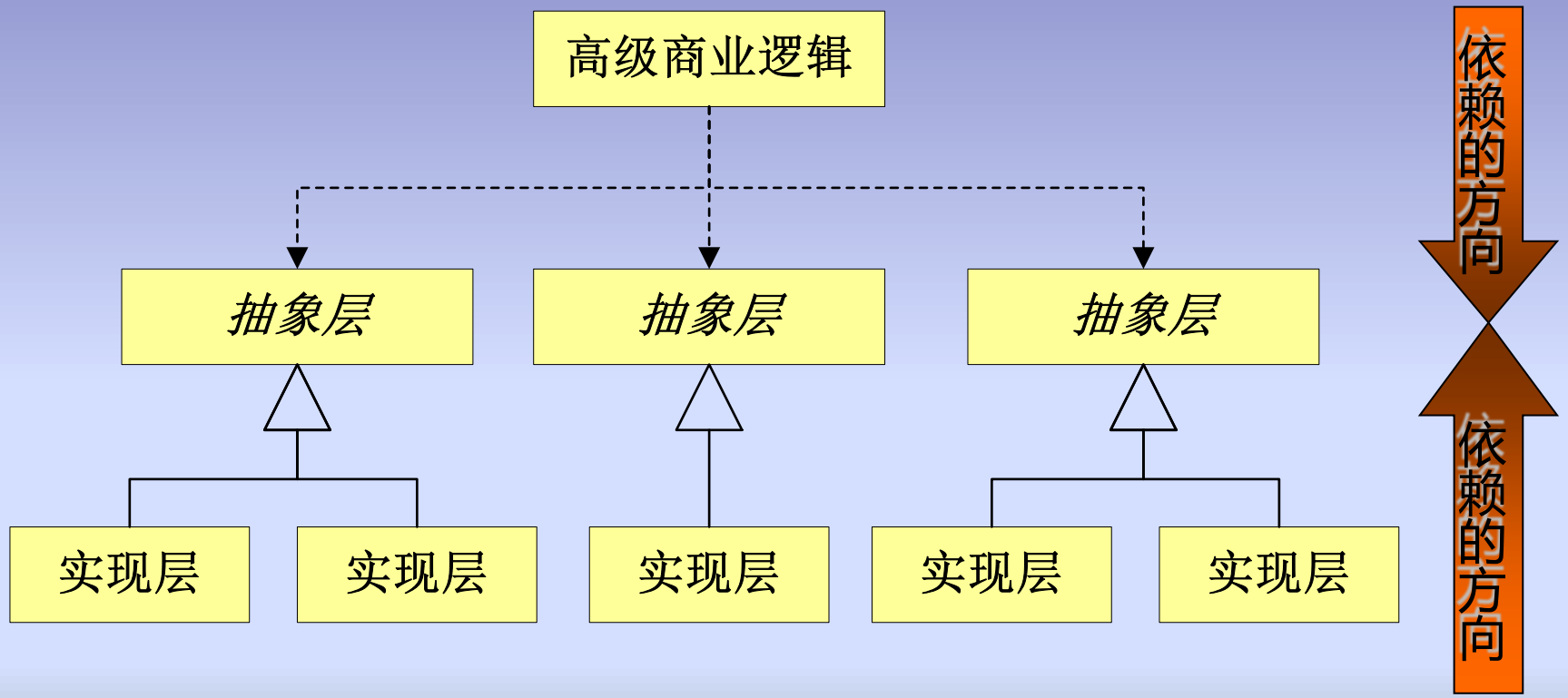
依赖倒转原则

Dependence Inversion Principle

传统的过程性系统



符合DIP的系统



为何“倒转”？

- 传统的过程化设计倾向于使高层次的模块依赖于低层次的模块：即抽象层依赖于具体实现层
- 抽象层包含的是应用系统的商务逻辑和宏观的、对整个系统来说具有全局重要性的战略决定，是必然性的体现
- 具体层包含的是次要的与实现有关的算法和逻辑，是战术性的决定，具有相当大的偶然性，具体层的代码经常变化，并难以避免错误

为何“倒转”？

- 抽象层依赖于具体层，将使许多具体层实现细节的变化会立即影响到抽象层的宏观逻辑，导致
  ”微观决定宏观，战术决定战略，偶然决定必然”
- 依赖倒转原则就是要将这个不合理的依赖关系倒转过来
- 依赖倒转原则也是隐藏在COM、CORBA、JavaBeans、EJB等构件设计模型背后的基本原理

复用与可维护性的“倒转”

- 从软件复用的角度看，高层模块才是设计者真正应该复用的对象。但在传统的过程化设计中，却侧重于具体层模块的复用，如算法、数据结构和函数库的复用等
- 这样较高层的结构依赖于较低的结构，较低的结构又依赖于更低层的结构，直到依赖于每一行代码。因此，较低层上的修改，会造成较高层的修改，直到高层逻辑的修改
- 同样，传统方法也强调具体层上可维护性，包括函数、数据结构的维护性，而不是高层模块的可维护性

复用与可维护性的“倒转”

- 从复用的意义而言，既然抽象层含有一个系统中最重要的宏观商务逻辑，是做战略性判断和决定的地方，因此，抽象层应当是相对稳定的，也应当是复用的重点
- 由于现有的复用侧重于具体层模块和细节的复用，因此，“倒转”也是指复用的重点放在抽象层上
- 同样，最重要的宏观商务逻辑也应当是维护的重点，而不是相反。因此，依赖倒转原则也可以带来软件系统复用和可维护性的“倒转”

DIP的具体表述

- DIP原则要求客户端尽量依赖于抽象偶合
- DIP的具体表述是：抽象不应当依赖于细节；细节应当依赖于抽象
- Abstractions should not depend upon details，Details should depend upon abstractions

DIP的另一种表述

- DIP另一种表述：要针对接口编程，而不要针对实现编程。
- Pragram to an interface,not to an implementation
- 倒转依赖关系强调一个系统之中实体之间关系的灵活性
- 如果设计者希望遵守“开放-封闭”原则，那么依赖倒转便是达到这一目标的基本途径

接口

- 接口是一个对象在对其它的对象进行调用时所知道的方法集合
- 一个对象可以实现多个接口(实际上，接口是对象所有方法的一个子集)
- 类型(Type)是对象的一个特定的接口
- 不同的对象可以具有相同的类型，而且一个对象可以具有多个不同的类型
- 一个对象仅能通过其接口才会被其它对象所了解
- 接口是实现插件化（pluggability）的关键

实现继承和接口继承

- 实现继承（类继承）：一个对象的实现是根据另一个对象的实现来定义的
- 接口继承（子类型化）：描述了一个对象可在什么时候被用来替代另一个对象
- C++的继承机制既指类继承，又指接口继承
- C++通过继承纯虚(抽象)类来实现接口继承
- Java对接口继承具有单独的语言构造方式—Java接口
- Java接口构造方式更加易于表达和实现那些专注于对象接口的设计

接口的优点

- 对象间的连接不必硬绑定(hardwire)到一个具体类的对象上，增加了灵活性
- Client不必知道其使用对象的具体所属类:一个对象可以很容易地被（实现了相同接口的）的另一个对象所替换
- 松散藕合(loosens coupling): 提高了(对象)复用的机率，因为被包含对象可以是任何一个实现了指定接口的类

变量的静态类型和真实类型(Java)

- 变量被声明时的类型称为变量的静态类型(Static Type),有时静态类型又称明显类型(Apparent Type)，变量所引用的对象的实际类型则称为变量的真实类型(Actual Type)，其源代码如下：

```
List employees = new Vector();
```

- 在上面代码中，变量employee的静态类型是List，而它的真实类型是Vector

接口隔离原则

Interface Segregation Principle

ISP定义

- 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
- 不应该强迫客户程序依赖于它们不用的方法
- 应该避免接口污染（**Interface ontamination**）

接口污染

- 过分臃肿的接口就是对接口的污染(contamination)

接口污染

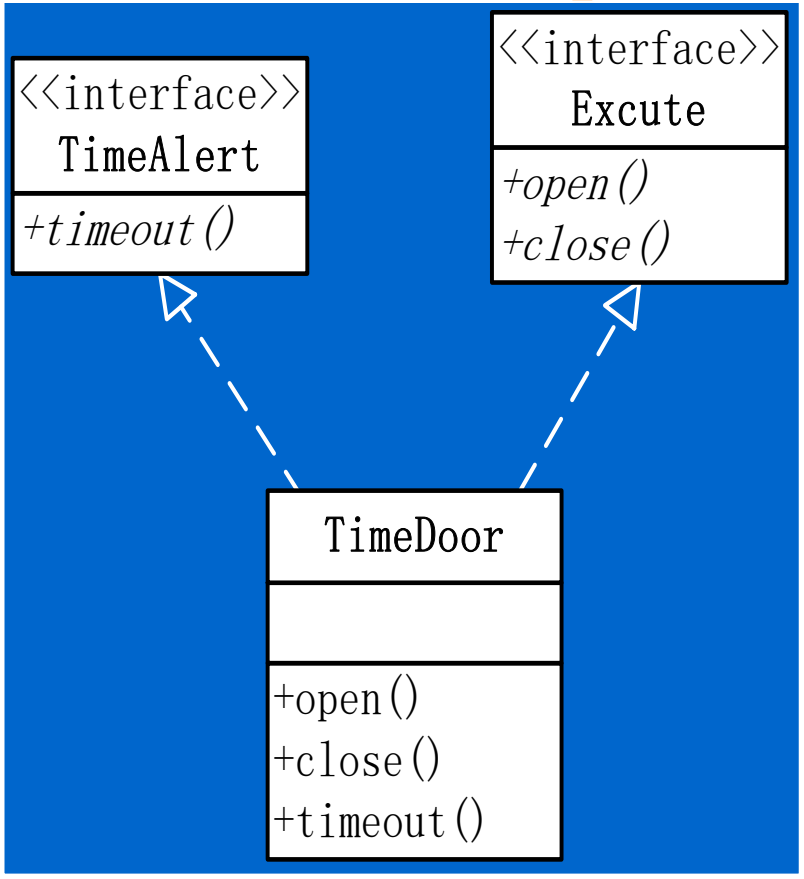
- 需求：一扇能超时报警的门
- Door
  - Open()
  - Close()
  - Timeout()
- 当需要其他的门时习惯性从Door中继承
- 问题在哪儿？
  - 所有的门都拥有timeout操作，即便它并不需要

接口污染

- 缺乏经验的设计者往往想节省接口的数目，因此，将一些看起来差不多的接口合并
- 有些人认为这是一种代码优化，这是不对的
- 准确而恰当地划分角色以及角色所对应的接口，是面向对象设计的一个重要组成部分
- 将没有关系的接口合并在一起，形成一个臃肿庞大的接口就是对角色和接口的污染

解决方法-分离接口

- 使用委托分离接口
  Adapter模式
- 使用多重继承分离接口



角色的合理划分

- 一个接口就相当于剧本中的一个角色，而该角色在舞台上由哪个演员来演出则相当于接口的实现
- 因此，一个接口应简单地代表一个角色，而不是多个角色
- 如果系统涉及多个角色，那么每一个角色都应当由一个特定的接口代表。
- 为避免混淆，我们将这种角色划分的原则称为“角色隔离原则”

角色的合理划分

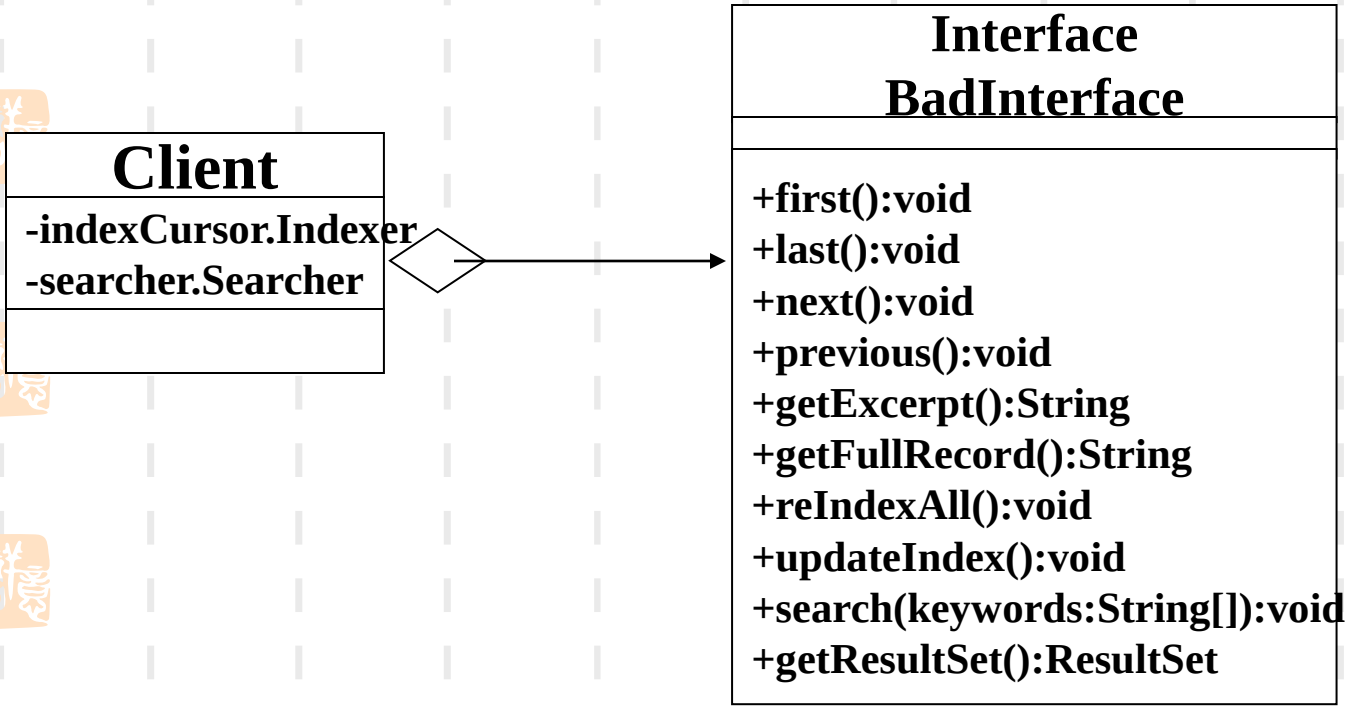
- 由于每一个接口都代表一个角色，实现一个接口的对象，在其整个生命周期中都扮演这个角色，因此将角色划分清楚是系统设计的一项重要工作
- 一个符合逻辑的推论是，不要将几个不同角色交给同一个接口，而应当交给不同的接口
- 将接口理解为一个类所提供的全部方法特征的集合，是一种逻辑上的概念
- 这样，接口的划分就直接带来类型的划分

实际案例

- 一个动态资料网站将大量的文本资料存储在文件或关系数据库中，用户可以通过输入一个或数个关键字对网站进行全文搜索。
- **AltaVista** 的网站搜索引擎
  - 索引库
  - 搜索器
  - 搜索结果
  - 客户端程序

一个不好的设计

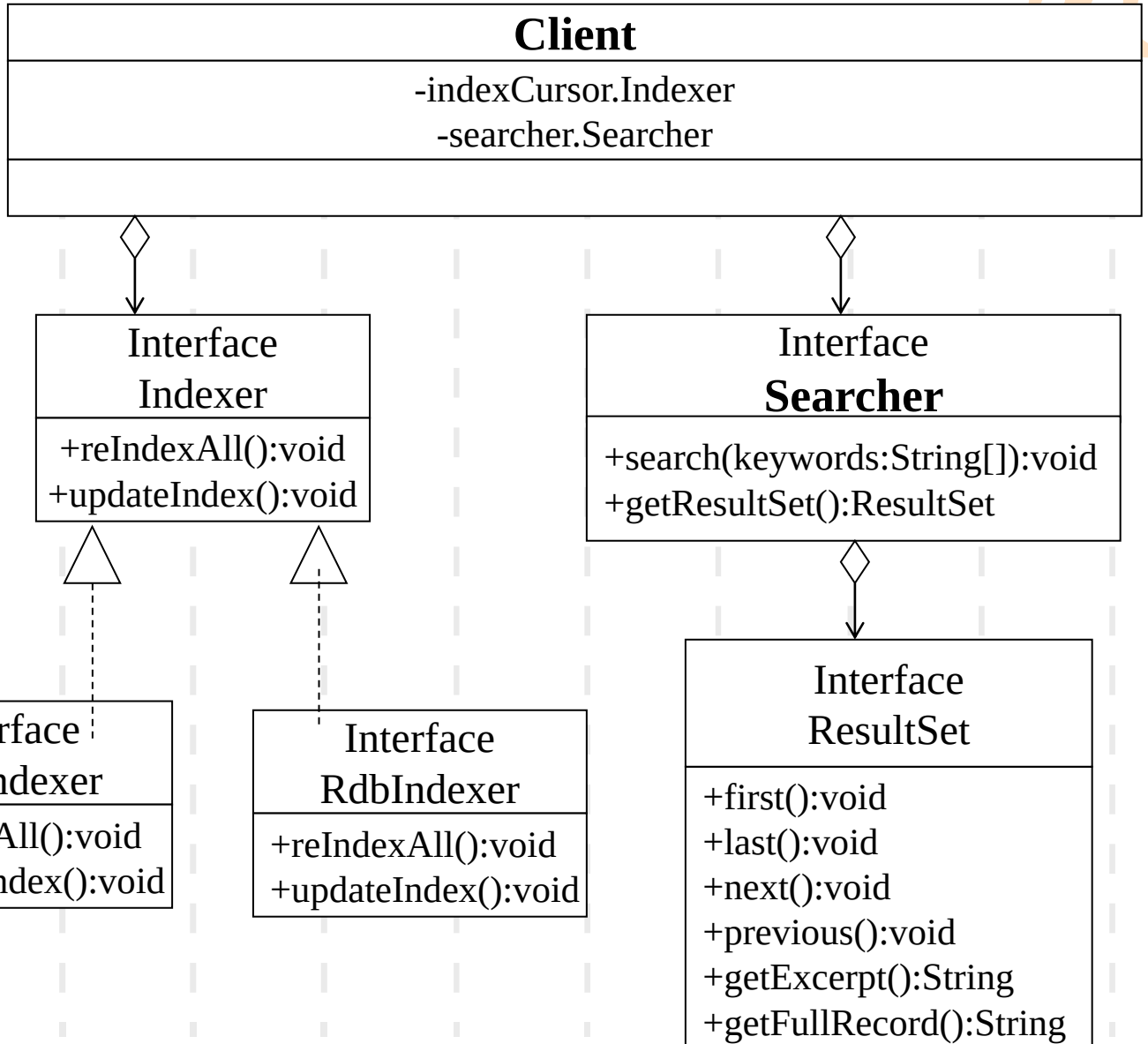
- 如下图所示，一个称为BadInterface的接口负责所有操作，从提供搜索功能到建立索引功能，甚至包括搜索结果集的功能均包含在着一个接口中



一个不好的设计

- 这个设计违反了角色分割原则，将不同功能的接口放在一起，由一个接口扮演了包括搜索器、索引生成器以及搜索结果集角色在内的所有角色。
- 那么，符合接口隔离原则的设计如图所示。

符合接口隔离原则的设计



复合复用原则

Composite Reuse Principle

定义

- 复合复用原则(Composite Reuse Principle,CRP)又称为复合/聚合复用原则(Composite/Aggregate Reuse Principle, CARP)
- CRP就是在一个新的对象当中使用一些已有的对象，使之成为新对象的一个组成部分；新对象通过向这些对象的委托而达到复用已有功能的目的
- 复合/聚合复用原则还有另一个更加明确的表述：优先使用(对象)复合，而非(类)继承(Favor Composition Over Inheritance)

复合与聚合

- 复合(Composite)和聚合(Aggregate)均为特殊类型的关联(Association)
- 聚合用来表示“拥有”关系或整体与部分的关系。而复合则用来表示一种强得多的“拥有”关系
- 在复合关系中，部分和整体的生命周期是一样的。一个复合而成的新对象完全拥有对其组成部分的支配权，包括他们的创建和销毁。从程序设计语言的角度，复合而成的新对象对组成部分的内存分配和释放负有绝对责任

复合与聚合

- 更进一步讲，复合的多重性(Multiplicity)不能超过1，即一个复合关系中的成分对象在同一时间只能属于复合关系，即不能被另一个复合关系共享
- 如果一个复合对象要被销毁，那么所有的成分对象要么也被销毁，要么将其责任交给另一个复合对象
- 对C++语言程序员而言，复合就是值的聚合(Aggregation by values),而聚合则是引用的聚合(Aggregation by Reference)

复合复用

- 复合复用是一种通过创建一个复合了其它对象的对象，并将功能委托给所复合的一个对象，从而获得新功能的复用方法
- 复合有时也称为“聚合”(aggregation)或“包容”(containment)，尽管有些作者对这些术语赋予了专门的含义

复合复用的优点

- “黑盒”复用:容器类仅能通过被包含对象的接口来对其进行访问,因为被包含对象的内部细节对外不可见
- 装性好:实现上的相互依赖性比较小，被包含对象与容器对象之间的依赖关系比较少
- 每一个类只专注于一项任务:通过获取指向其它的具有相同类型的对象引用，可以在运行期间动态地定义(对象的)组合

复合复用的缺点

- 导致系统中的对象过多
- 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

继承复用

- (类)继承是一种通过扩展一个已有对象的实现，从而获得新功能的复用方法
- 泛化类（超类）可以显式地捕获那些公共的属性和方法
- 特殊类（子类）则通过附加属性和方法来进行实现的扩展

继承复用的优点

- 容易进行新的实现，因为其大多数可继承而来
- 易于修改或扩展那些被复用的实现

继承复用的缺点

- “白盒”复用:因为父类的内部细节对于子类而言通常是可见的
- 破坏封装性:因为会将父类的实现细节暴露给子类，当父类的实现更改时，子类也不得不会随之更改
- 僵硬：从父类继承来的实现将不能在运行期间进行改变

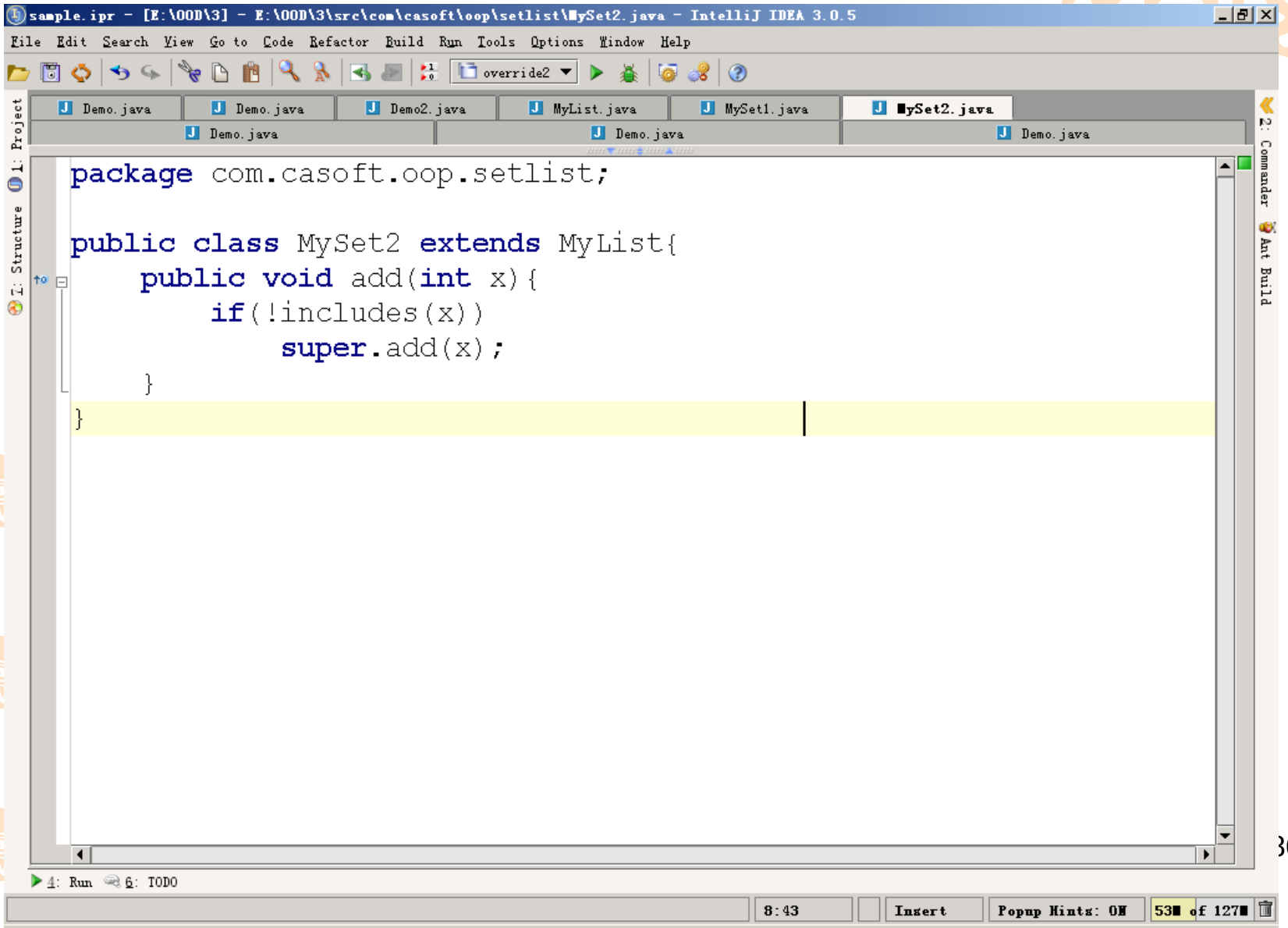
Coad规则

- 仅当下列的所有标准被满足时，方可使用继承：
  1）子类表达了“是一个...的特殊类型”，而非“是一个由...所扮演的角色”
  2）子类的一个实例永远不需要转化（transmute）为其它类的一个对象
  3）子类是对其父类的职责（responsibility）进行扩展，而非重写或废除（nullify）
  4）子类没有对那些仅作为一个工具类(utility class)的功能进行扩展
  5）对于一个位于实际的问题域(Problem Domain)的类而言，其子类特指一种角色(role)、交易(transaction)或设备(device)

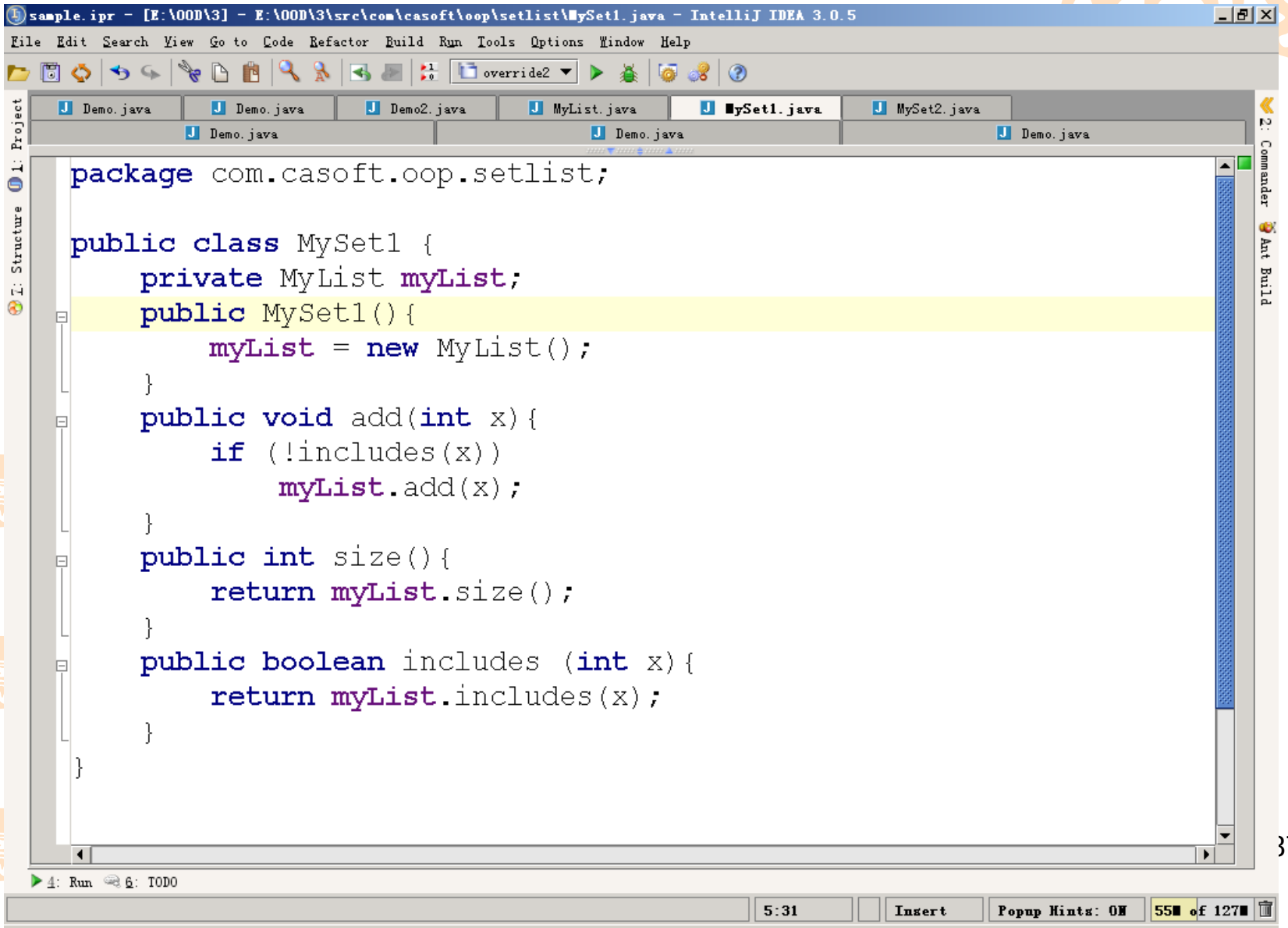
案例

- MyList: 包含整数值的一个列表
  - void add(int);
  - int firstElement();
  - int size();
  - boolean includes(int);
  - void remove(int);
- MySet:包含整数值的一个集合（禁止重复）
  - void add(int);
  - int size();
  - boolean includes(int);

使用继承


sample.ipr - [E:\OOD\3] - E:\OOD\3\src\com\casoft\oop\setlist\MySet2.java - IntelliJ IDEA 3.0.5
File Edit Search View Go to Code Refactor Build Run Tools Options Window Help
override2
Demo.java
Demo.java
Demo2.java
MyList.java
MySet1.java
MySet2.java
Demo.java
Demo.java
Demo.java
1: Project
7: Structure
2: Commander
Ant Build
package com.casoft.oop.setlist;
public class MySet2 extends MyList{
public void add(int x){
if(!includes(x))
super.add(x);
}
}
4: Run
6: TODO
8:43
Insert
Popup Hints: ON
53 of 127

使用组合


sample.ipr - [E:\OOD\3] - E:\OOD\3\src\com\casoft\oop\setlist\MySet1.java - IntelliJ IDEA 3.0.5
File Edit Search View Go to Code Refactor Build Run Tools Options Window Help
override2
Demo.java
Demo.java
Demo2.java
MyList.java
MySet1.java
MySet2.java
Demo.java
Demo.java
Demo.java
1: Project
7: Structure
2: Commander
Ant Build
package com.casoft.oop.setlist;
public class MySet1 {
private MyList myList;
public MySet1(){
myList = new MyList();
}
public void add(int x){
if (!includes(x))
myList.add(x);
}
public int size(){
return myList.size();
}
public boolean includes (int x){
return myList.includes(x);
}
}
4: Run
6: TODO
5:31
Insert
Popup Hints: ON
55M of 127M

组合 vs. 继承

	组合	继承
代码长度	长	短
提供功能	少	多
屏蔽功能	可以	不可以
支持多态	否	是
效率	略低	略高
规则复杂度	简练	复杂(yo-yo)

思考题

- **java.util.Stack**
- 继承了 **Vector**
- 思考：这样对吗？

继承/组合总结

- 组合与继承都是重要的重用方法
- 在OO开发的早期，继承被过度地使用
- 随着时间的发展，我们发现优先使用组合可以获得重用性与简单性更佳的设计
- 当然可以通过继承，以扩充(enlarge)可用的组合类集(the set of composable classes)
- 因此组合与继承可以一起工作
- 但是我们的基本原则是：优先使用对象组合，而非（类）继承

迪米特原则

Law of Demeter
Least Knowledge Principle

表述

- 软件实体要尽可能的只与和它最近的实体进行通讯**(Each unit should only have limited knowledge about other units: only about units “closely” related to the current unit)**
- 只与你的朋友讲话(Only talk to your immediate friends)
- 不要与陌生人说话(Don’t talk to strangers)

老子论圣人之治

- 降低统治成本的最少知识原则
- 最大程度减少对象之间的通信

狭义迪米特法则

- 如果两个类不必彼此直接通信，则不应当相互调用
- 可以通过第三者转发消息

对象间方法的调用

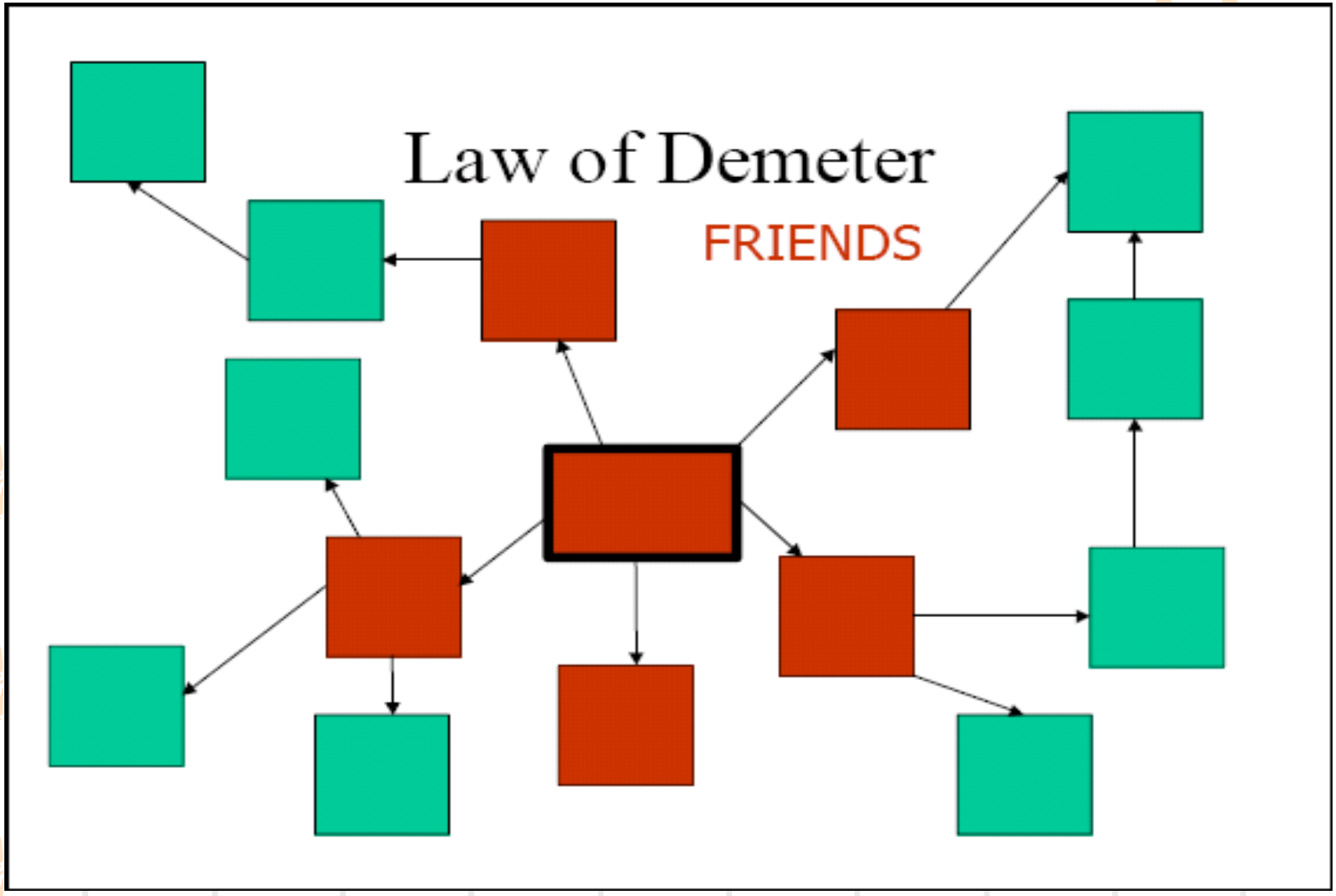
- “talk”就是对象间方法的调用
- 迪米特法则表明了对象间方法调用的原则：
  （1） 调用对象本身的方法；
  （2）调用通过参数传入的对象的方法；
  （3）调用在方法中创建的对象的方法
  （4）调用所包含对象的方法

Law of Demeter

- A method M of an object O should invoke only the methods of the following kinds of objects:
  1. O itself
  2. parameters of M
  3. any object M creates /instantiates
  4. direct component objects of O

http://en.wikipedia.org/wiki/Law_of_Demeter

对象间方法的调用



符合迪米特法则的例子

```
1 public class Car {
2   Engine engine;
3
4   public Car() {
5     //initialize engine,etc.
6 }
7
8 public void start(Key key) {
9   Doors doors = new Doors();
10   boolean authorized = key.turns();
11
12   if(authorized) {
13     engine.start();
14     updateDashboardDisplay();
15     doors.lock();
16 }
17
18 public void updateDashboardDisplay() {
19   //update display
20}
21
22 }
```

符合迪米特法则的例子

- 对start()方法中的语句进行分析：
  - 第10行－key.turns()：符合上述的第（2）条，key对象是通过参数传入start()方法的
  - 第13行－engine.start()：符合上述的第（4）条，engine对象是包含在Car的对象之中的
  - 第14行－UpdateDashboardDisplay()：符合上述的第（1）条UpdateDashboardDisplay()方法是Car对像自身的方法
  - 第15行－doors.lock()：符合上述的第（3）条，doors对象是在start()方法中创建的对象

违反迪米特法则的例子

```
1 public float getTemp(station) {
2   Thermometer thermometer = station.getThermometer();
3   return thermometer.getTemperature();
4 }
```

- 上面的方法中**station**对象是**immediate friends**
- 但是上面的代码却从**station**对象中返回了一个**Thermometer**对象，然后调用了**thermometer**对象的**getTemperature()**方法，违反了**Principle of Least Knowledge!**

违反迪米特法则的例子

- 下面对上面的方法作出符合**Principle of Least Knowledge**的改进：

```
1  public float getTemp(station) {
2    return station. getTemperature();
3  }
```

- 要点:
  - 在**Station**类中添加一个方法**getTemperature()**
  - 这个方法将调用**Station**类中含有的**Thermometer**对象的**getTemperature()**
  - 这样**getTemp()**方法就只知道**Station**对象而不知道**Thermometer**对象

Example Quiz Question

- What code represents the better design, a or b?

```
// a.
dog.body.tail.wag();
// b.
dog.expressHappiness();
```

- **The bad example couples dog to two indirect classes DogBody, and DogTail**
- **The good design couples dog only to the direct class DogAnimal**

广义迪米特法则

- 不变类(例如: String)，只要有可能，类应当设计为不变类
- 尽量降低类的访问权限
- 类之间的耦合越弱越好

Immutable

不变模式*

不变模式

- 满足下列条件：
  - 不提供任何方法修改对象的状态
  - 所有属性都是私有的
  - 类中所有的方法都是 **final** 的
  - 类本身就是**final**的

总 结

- 笛米特法则告诉我们要尽量使对象只和离自己最近的对象进行交互
- 离自己最近的对象包括：
  - 自身包含的对象
  - 方法中创建的对象
  - 通过参数传进的对象
  - 还有自己本身

其他设计原则

- **REP** 重用发布等价原则：重用的粒度就是发布的粒度
- **CCP** 共同重用原则：一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么就要重用包中的所有类。相互之间没有紧密联系的类不应该在同一个包中
- **CRP** 共同封闭原则：包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包影响，则将对包中的所有类产生影响，而对其他的包不造成任何影响

其他设计原则

- **ADP** 无环依赖原则：在包的依赖关系中不允许存在环，细节不应该被依赖
- **SDP** 稳定依赖原则：朝着稳定的方向进行依赖.应该把封装系统高层设计的软件（比如抽象类）放进稳定的包中，不稳定的包中应该只包含那些很可能会改变的软件(比如具体类)
- **SAP** 稳定抽象原则：包的抽象程度应该和其他稳定程度一致.一个稳定的包应该也是抽象的，一个不稳定的包应该是具体的

其他设计原则

- **DAP(Default Abstraction Principle)缺省抽象原则：**在接口和实现接口的类之间引入一个抽象类,这个类实现了接口的大部分操作
- **IDP(Interface Design Principle)接口设计原则：**规划一个接口而不是实现一个接口
- **BBP(Black Box Principle)黑盒原则：**多用类的聚合，少用类的继承
- **DCSP(Don‘t Concrete Supperclass Principle)不要构造具体的超类原则：**避免维护具体的超类