CS2001
软件工程

9. 软件设计
—面向对象软件设计

课件来源　　复旦大学软件学院

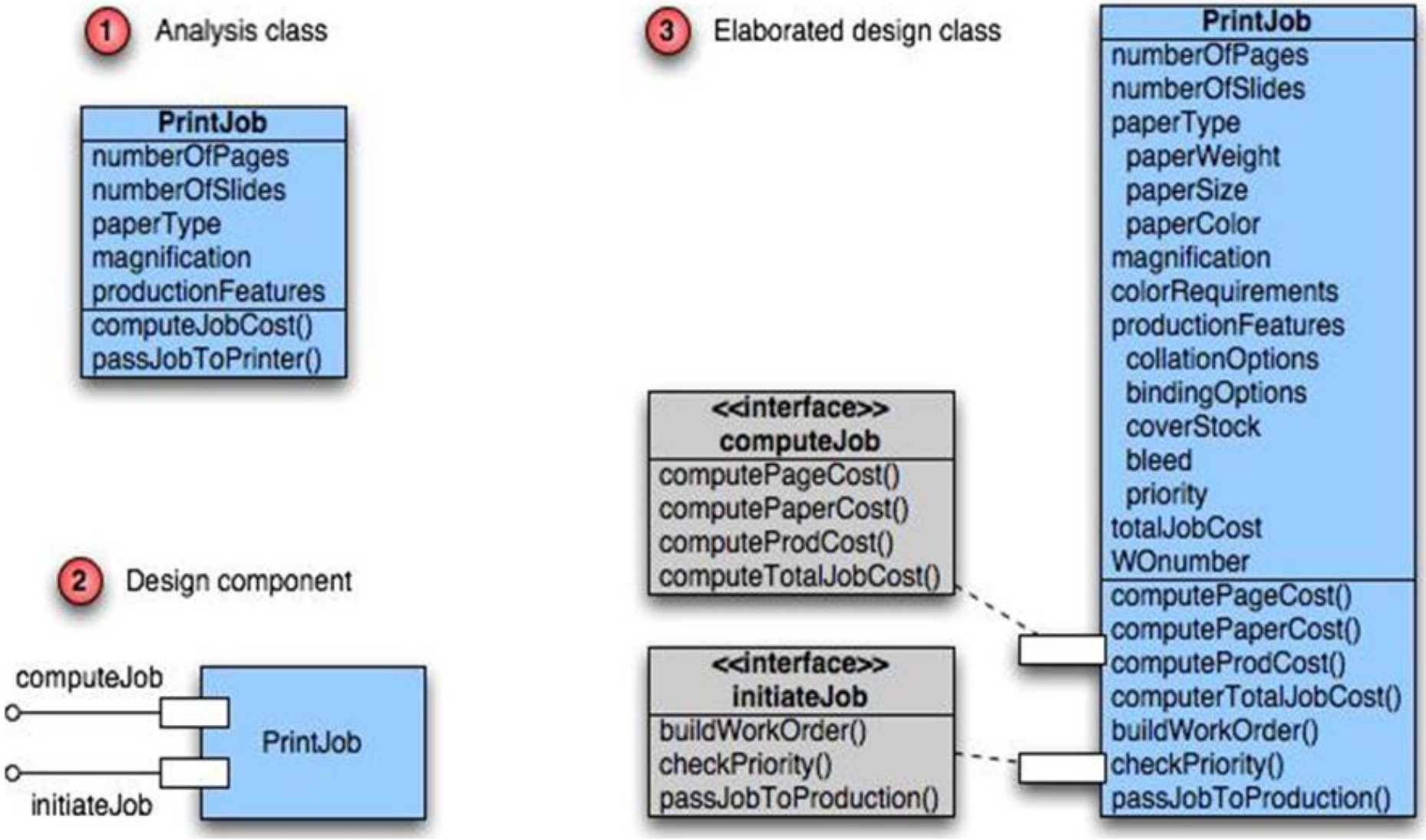
组件级详细设计

- 组件（Component）*OMG UML Specification*
  - ✓ 一种系统组成部分
  - ✓ 模块化、可部署、可替换
  - ✓ 内部封装了实现
  - ✓ 对外暴露一组接口
- 面向对象视角的组件：由一组相互协作的类组成

组件级详细设计的任务

- 通过细化的组件设计满足体系结构设计中对组件的要求
- 例如，一个“订单处理”组件要求
  - ✓ 提供“订单处理”接口：接受订单请求，返回处理结果
  - ✓ 接口标准：通过标准的Web Service提供服务
  - ✓ 交互协议：首先发送订单号，核对通过发送详细信息…
  - ✓ 质量要求：具有每秒处理100笔以上订单的能力
- 详细设计任务
  - ✓ 细化接口定义：操作（方法）、参数、返回值等
  - ✓ 明确内部结构：类、职责分配、交互关系及行为
  - ✓ 关键算法和数据结构：确保满足质量要求

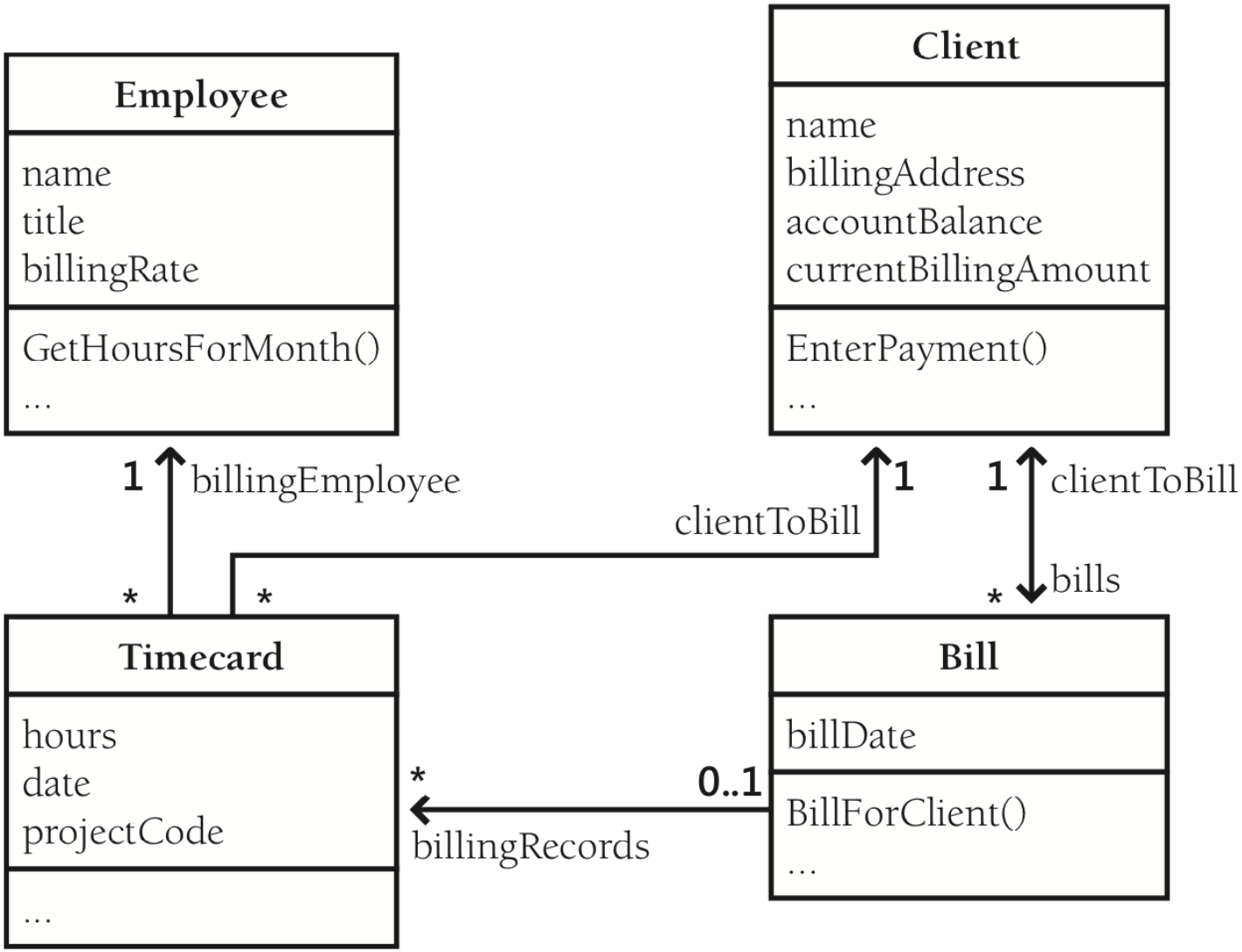
设计类的细化


1 Analysis class
PrintJob
numberOfPages
numberOfSlides
paperType
magnification
productionFeatures
computeJobCost()
passJobToPrinter()
2 Design component
computeJob
initiateJob
PrintJob
3 Elaborated design class
<<interface>>
computeJob
computePageCost()
computePaperCost()
computeProdCost()
computeTotalJobCost()
<<interface>>
initiateJob
buildWorkOrder()
checkPriority()
passJobToProduction()
PrintJob
numberOfPages
numberOfSlides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
productionFeatures
collationOptions
bindingOptions
coverStock
bleed
priority
totalJobCost
WOnumber
computePageCost()
computePaperCost()
computeProdCost()
computerTotalJobCost()
buildWorkOrder()
checkPriority()
passJobToProduction()

面向对象设计步骤

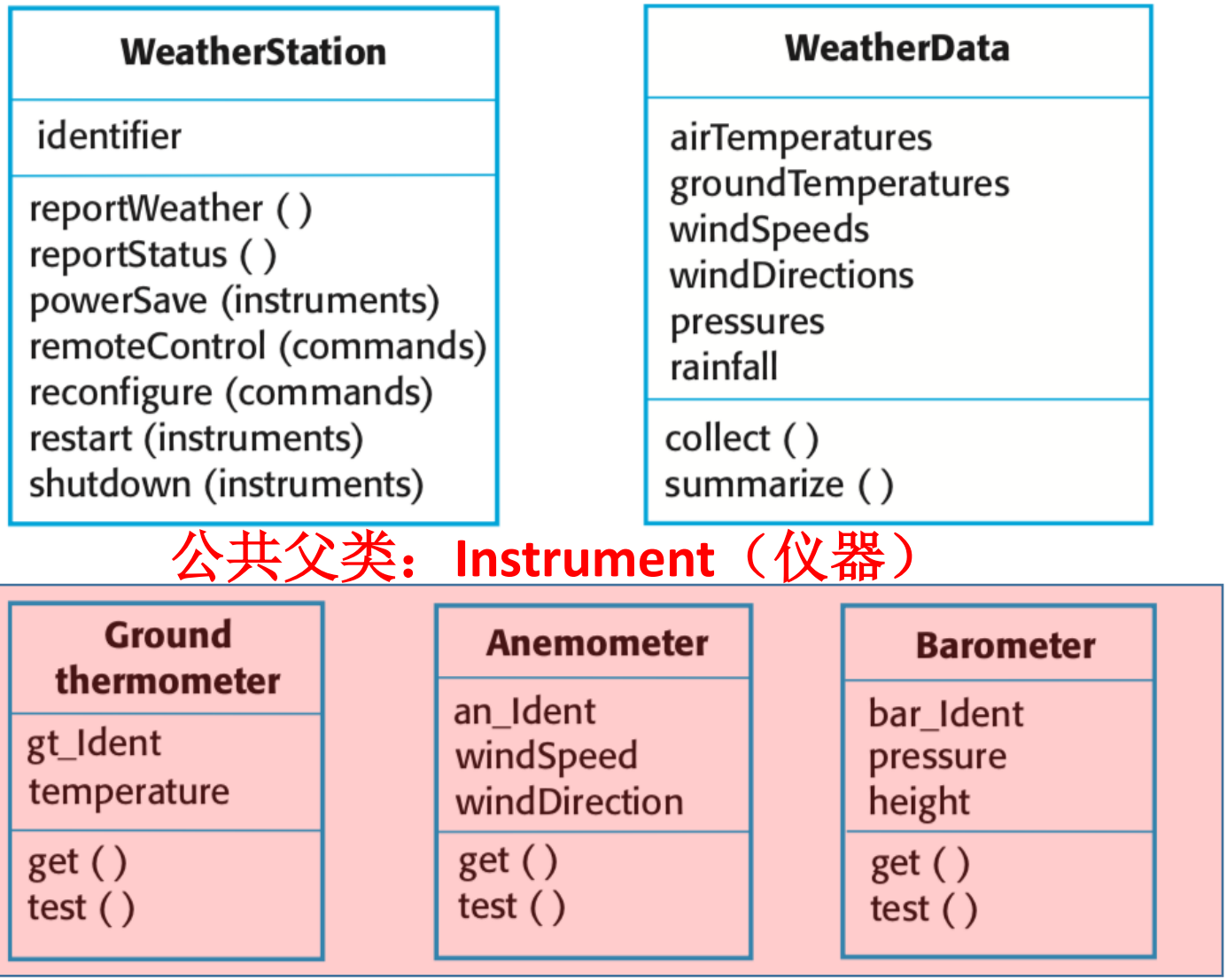
- 识别设计类
  - ✓ 识别来自问题域中的设计类
  - ✓ 识别位于接口（用户、硬件等）上的设计类
  - ✓ 识别基础设施设计类
- 明确设计类职责和协作
  - ✓ 明确每个类承担的职责
  - ✓ 明确类与类之间的交互和协作关系
  - ✓ 大致确定每个类的属性和操作
- 细化设计类内部细节
  - ✓ 考虑详细的类的属性和操作描述
  - ✓ 设计关键的内部数据结构和算法

一个服务收费系统中的类



《代码大全》5.3

一个气象站系统中的类

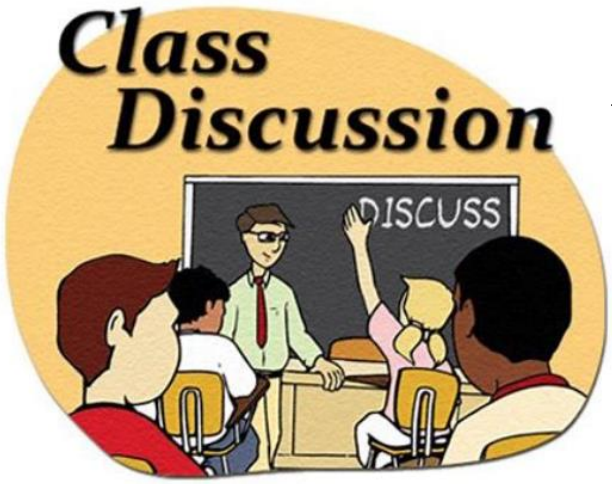


《软件工程》7.1.3

不同类型的类

- 来自问题领域的类
  - ✓ 业务对象：具有业务含义的名词性实体
  - ✓ 业务过程：具有业务含义的处理过程
- 位于接口上的类
  - ✓ 用户界面：负责与用户交互的类
  - ✓ 硬件接口：负责与硬件交互的类
  - ✓ 软件接口：负责与第三方软件系统交互的类
  - ✓ 一些业务对象也会在接口上传递业务信息
- 来自基础设施的类
  - ✓ 资源管理：对底层计算和存储等资源进行管理
  - ✓ 进程通信：负责跨网络或跨进程的通信

课堂讨论：设计类的识别



课堂讨论：学校所使用的图书馆管理系统（借/还书部分）应该包含哪些设计类？

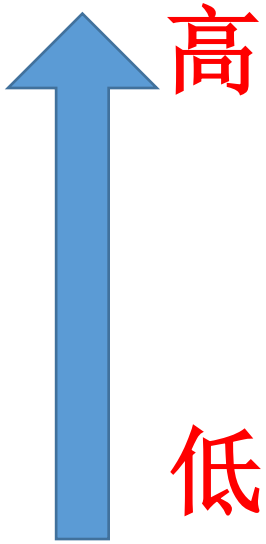
图书类、图书副本类
管理人员类
用户类（学生类、老师类、校外人员）
借阅记录类（借还书日期、备注类）
用户界面类
借还书操作类
数据库访问类
扫描接口类

类的职责分配

- 关注点分离：每个类有自己明确的职责
  - ✓ 纵向划分：按照业务或功能单元分解职责，如成绩管理、学分管理、学费管理等不同业务子领域
  - ✓ 横向划分：按照系统技术层次分解职责，如界面层、业务层、数据层、工具层等
  - ✓ 衡量标准：所对应的业务主题明确、技术领域聚焦
- 相对独立性：高内聚、低耦合
- 粒度合理：避免过于集中或过于分散
- 考虑未来可能的扩展
  - ✓ 共性抽象：通过抽象类和接口等手段进行共性抽象
  - ✓ 保持灵活性：依赖于抽象而非具体、松耦合

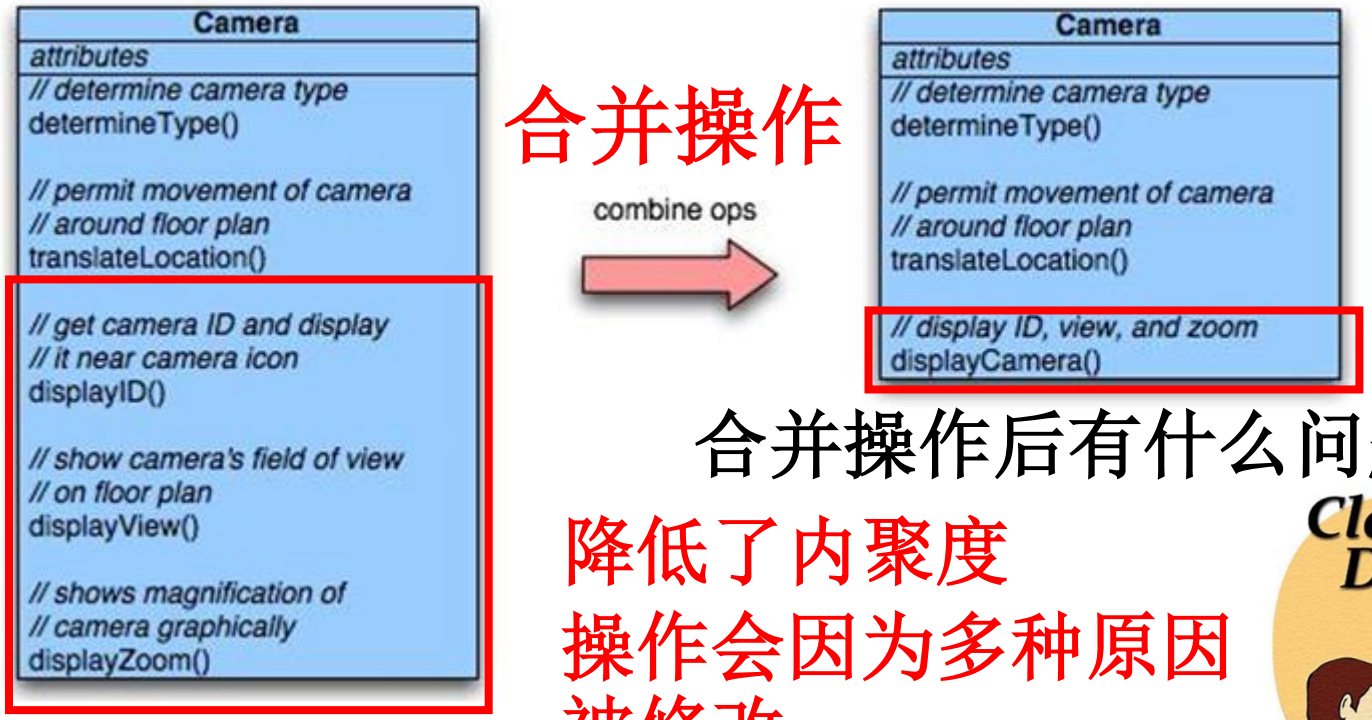
内聚

- 一个模块保持“单一意图”的程度
- 内聚意味着单个组件或类
  - ✓ 封装了与组件或类的职责密切相关的属性和操作
  - ✓ 这些属性和操作相互之间密切相关
  - ✓ 除此之外没有其他属性和操作
- 内聚的类型
  - ✓ Functional：功能内聚
  - ✓ Layer：层次内聚
  - ✓ Communicational：通信内聚
  - ✓ Sequential：顺序内聚
  - ✓ Procedural：过程内聚
  - ✓ Temporal：时间内聚
  - ✓ Utility：功用内聚



功能内聚

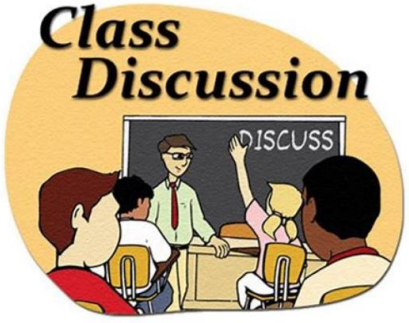
- 通常适用于单个操作（方法、函数）
- 表示一个模块仅执行一个操作然后返回结果



合并操作后有什么问题吗？

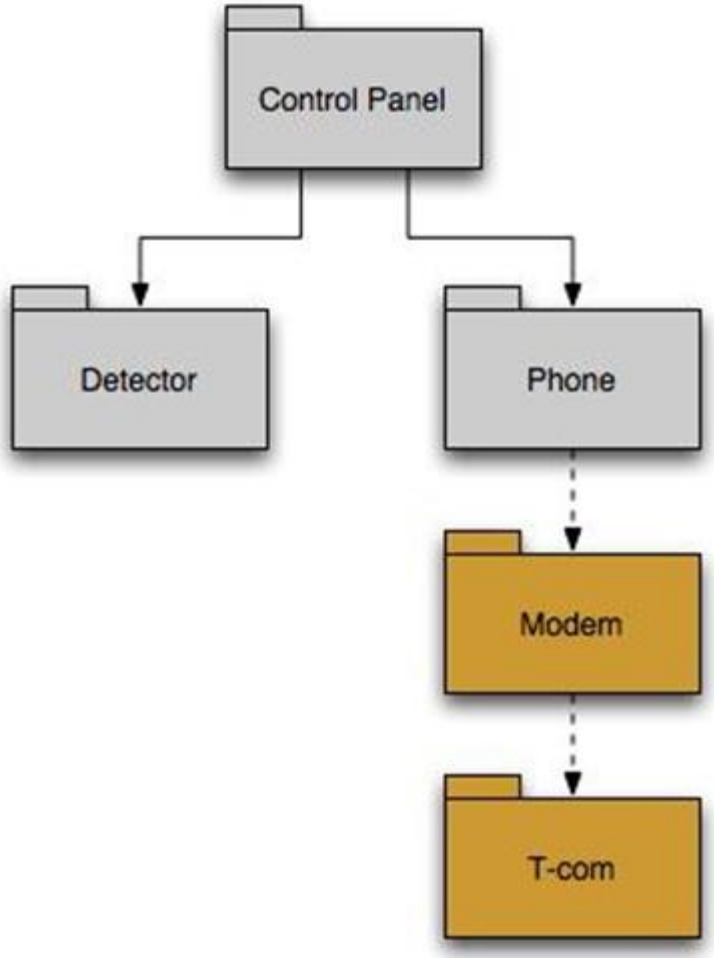
降低了内聚度

操作会因为多种原因被修改



层次内聚

- 通常适用于包、组件、类
- 内部的类或属性和操作等形成层次结构：上层访问下层，但下层不会访问上层

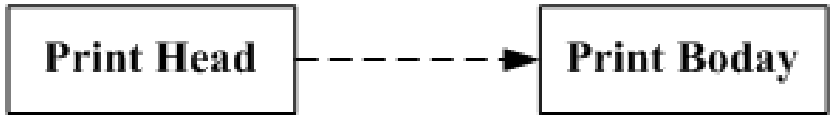


通信内聚

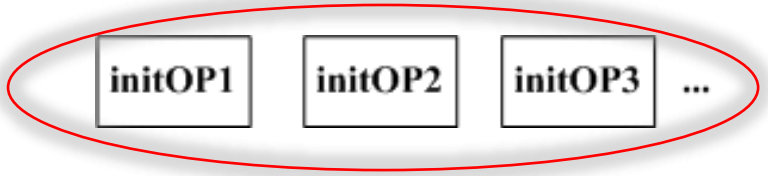
- 通常适用于类
- 访问所有数据的操作都被定义在一个类中
- 这种类通常只关心所考虑的数据，对其进行访问和存储
- 例如，ShoppingCart类只提供针对购物项的增删改查和保存操作

其他内聚

- 顺序内聚
- 过程内聚
- 时间内聚
- 功用内聚



初始化时执行的一组操作



Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
SimpleTimeZone	SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use
Stack	The Stack class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thre
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	TimeZone represents a time zone offset, and also figures out daylight savings.
TreeMap	Red-Black tree based implementation of the SortedMap interface.
TreeSet	This class implements the Set interface, backed by a TreeMap instance.
Vector	The Vector class implements a growable array of objects.
WeakHashMap	A hashtable-based Map implementation with *weak keys*.

classes in java.util package

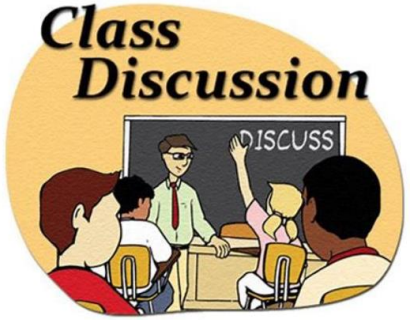
耦合

- 组件或类之间相互联系的程度
- 耦合的类型
  - ✓ 避免
    - ◆ 内容耦合（Content Coupling）
  - ✓ 小心
    - ◆ 共用耦合（Common Coupling）
    - ◆ 控制耦合（Control coupling）
  - ✓ 知晓
    - ◆ 调用耦合（Routine call coupling）
    - ◆ 类型使用耦合（Type use coupling）
    - ◆ 包含或引入耦合（Inclusion or import coupling）

内容耦合（需要避免）

- 某个组件或类“偷偷”修改其他组件或类的内部数据
- 违反了信息隐藏原则

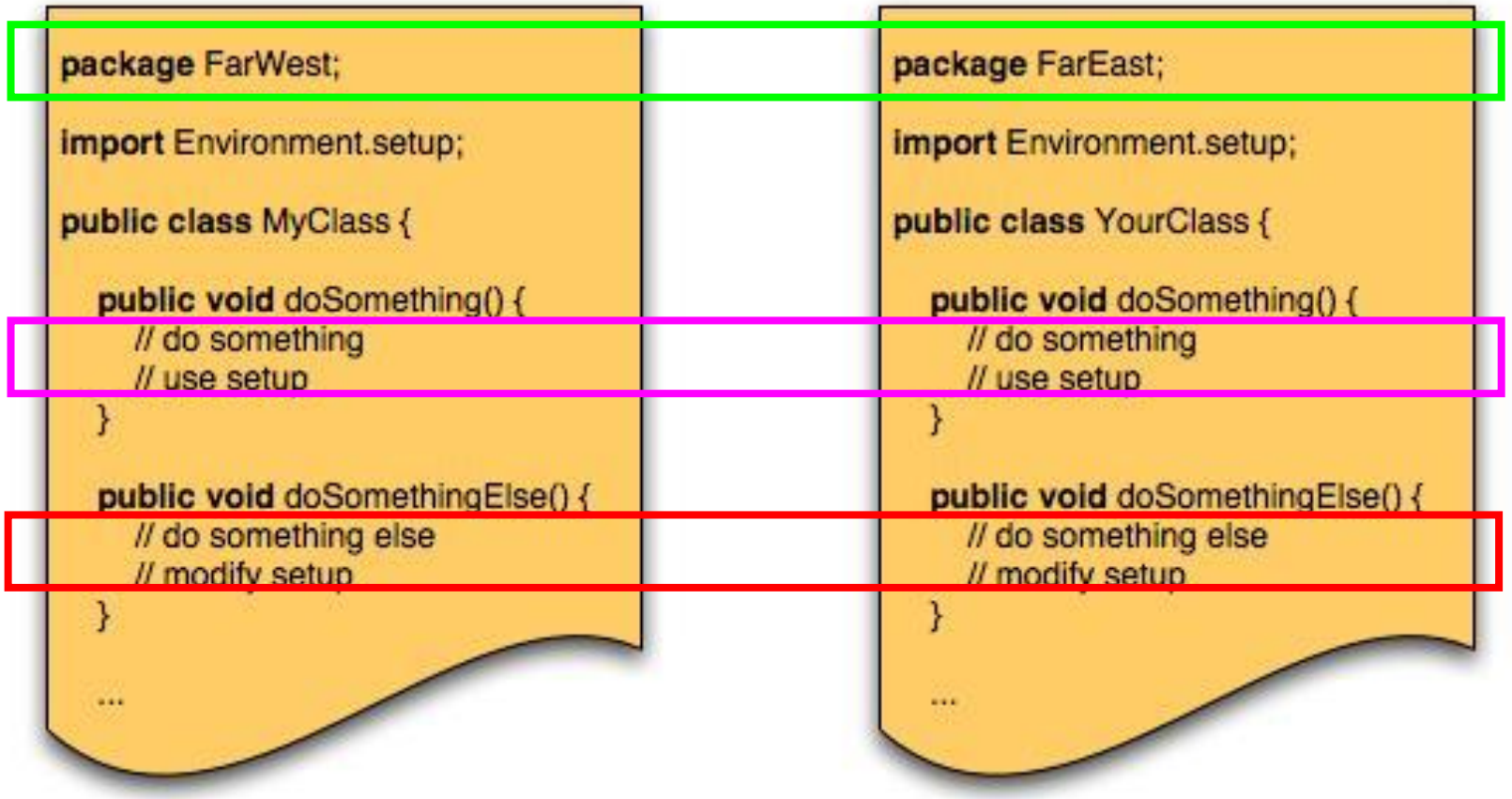
```
public class StudentRecord {

    private String name;
    private int[ ] quizScores;

    public String getName() {
        return name;
    }
    public int getQuizScore(int n) {
        return quizScores[n];
    }
    public int[ ] getAllQuizScores() {
        return quizScores;
    }

    ....
```



这个设计哪里有问题？

共用耦合（需要小心）

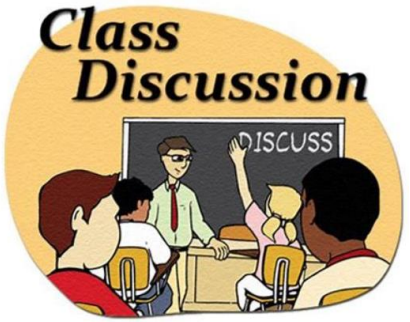
- 多个组件或类共用全局变量



控制耦合（需要小心）

- 一个操作调用另一个操作时传入影响其执行流的控制标志

```
public void studentRegister(Student stu) {
      …
      if (stu.isGraduate==true) // graduate
         …

      else  // undergraduate student
         …
   }
   ….
```

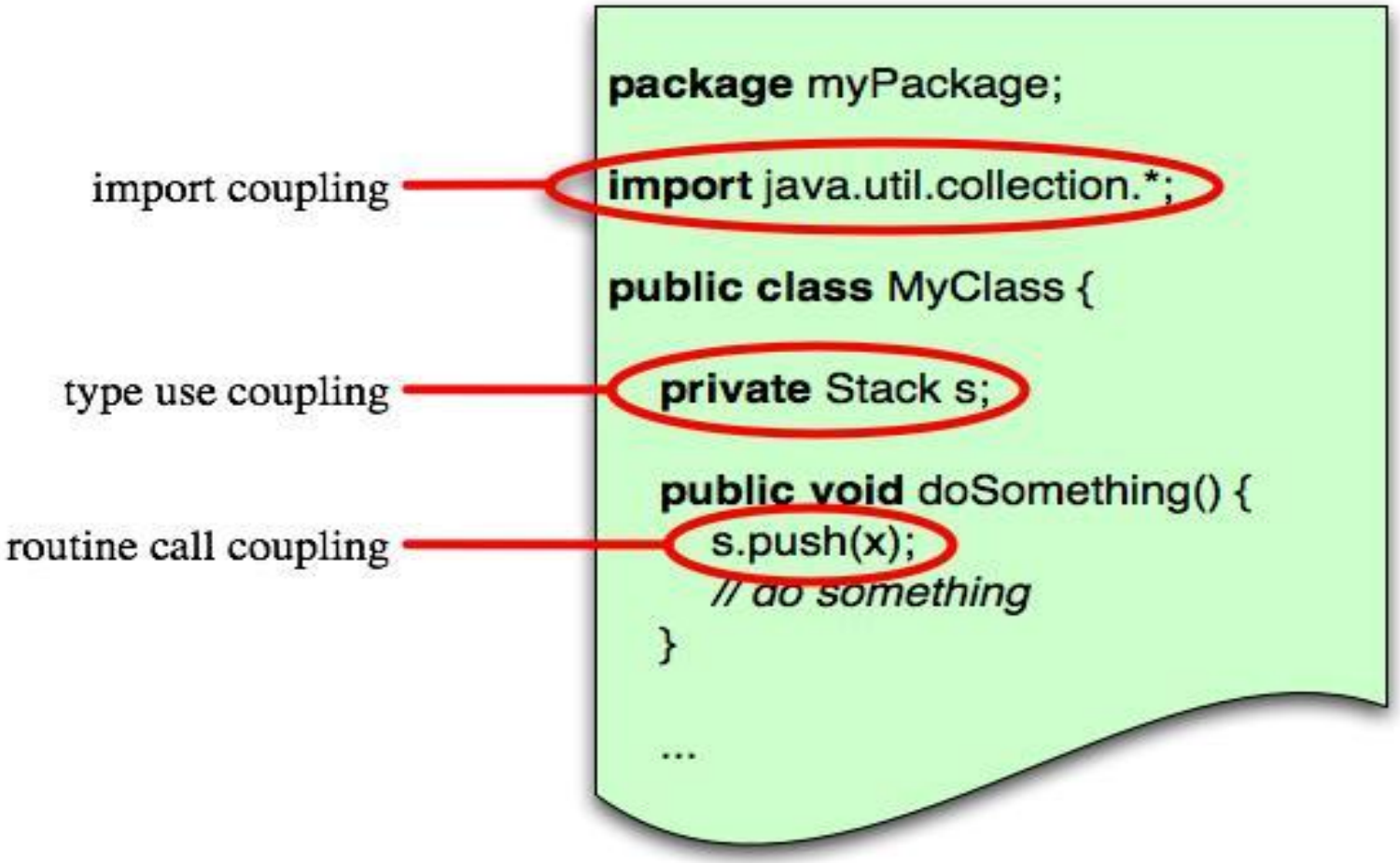


这种控制耦合有什么危害？

当增加新的学生类型时需要修改该方法（违反开闭原则）

常规耦合（需要知晓）

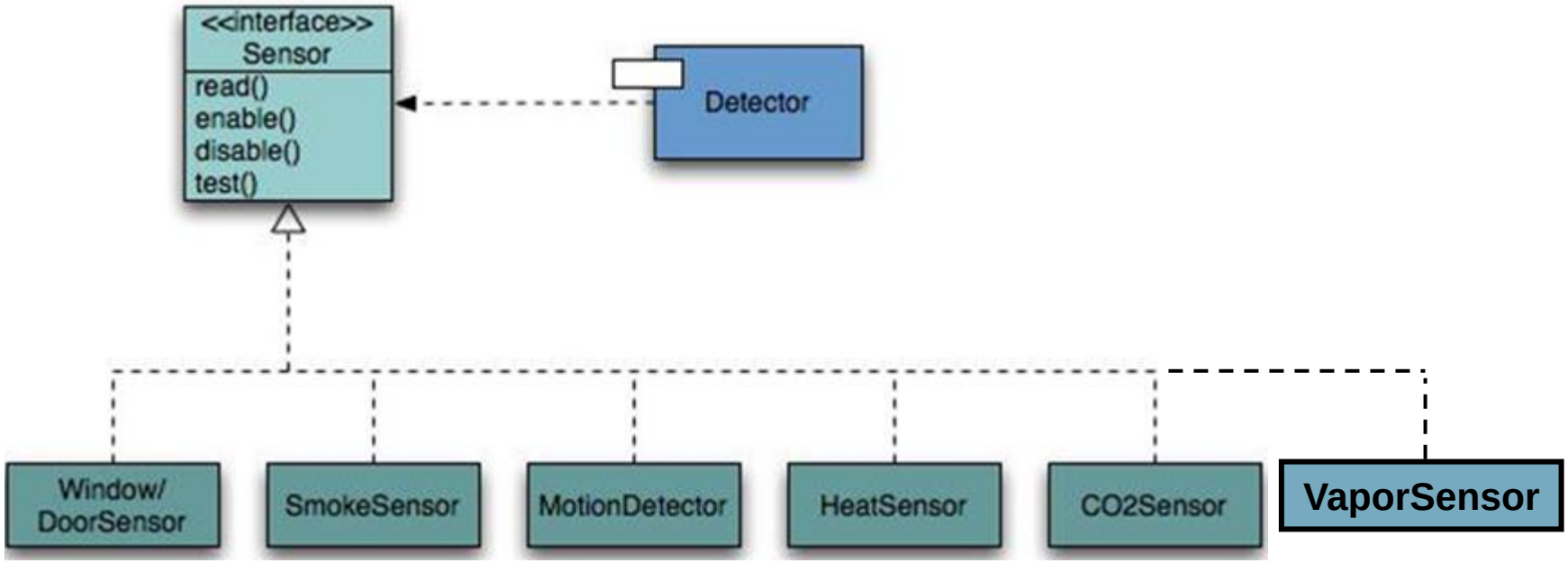
- 面向对象设计中经常出现



面向对象设计原则

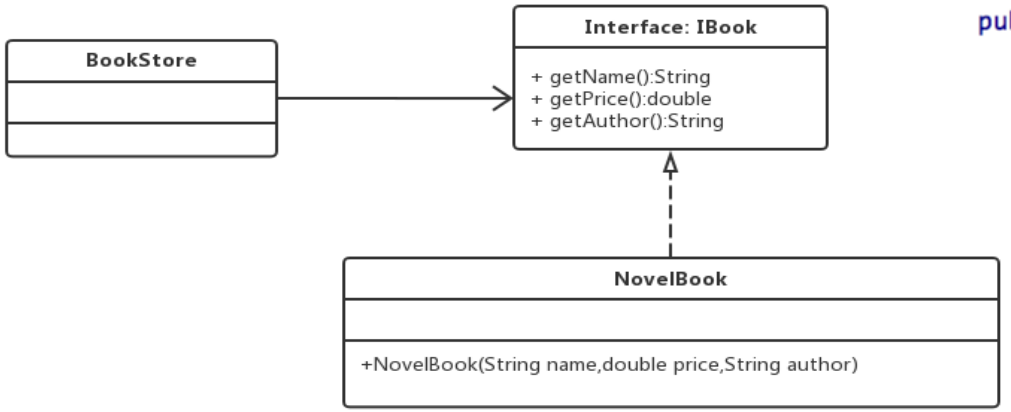
- 开闭原则
- Liskov替换原则
- 依赖反转原则
- 接口隔离原则

开闭原则



一个模块应该对扩展开放对修改封闭

开闭原则实例分析-1



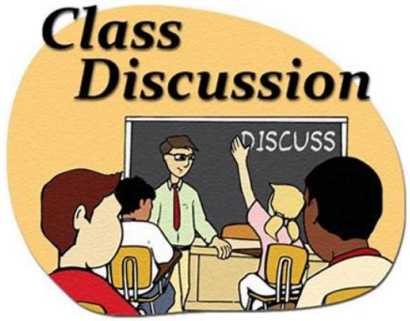

```
public interface IBook{
  public String getName();
  public double getPrice();
  public String getAuthor();
}
```

```
public class NovelBook implements IBook{
  private String name;
  private double price;
  private String author;

  public NovelBook(String name,int price,String author){
    this.name = name;
    this.price = price;
    this.author = author;
  }

  public String getAutor(){
    return this.author;
  }

  public String getName(){
    return this.name;
  }

  public double getPrice(){
    return this.price;
  }
}
```

```
public class Client{
  public static void main(Strings[] args){
    IBook novel = new NovelBook("笑傲江湖",100.0,"金庸");
    System.out.println("书籍名字："+novel.getName()+"书籍作者：
"+novel.getAuthor()+"书籍价格："+novel.getPrice());
  }
}
```

开闭原则实例分析-2

客户提出有些小说在销售时要进行打折，因此需要计算一个折扣价。为了实现改需求，应当如何修改现有的设计方案？



方案1：在Ibook接口中增加getOffPrice()方法

需修改接口和实现类，同时导致作为契约的接口的不稳定

方案2：修改NovelBook类的getPrice()方法处理打折

需修改实现类，同时导致无法读取原价

方案3：在NovelBook类中增加getOffPrice()方法

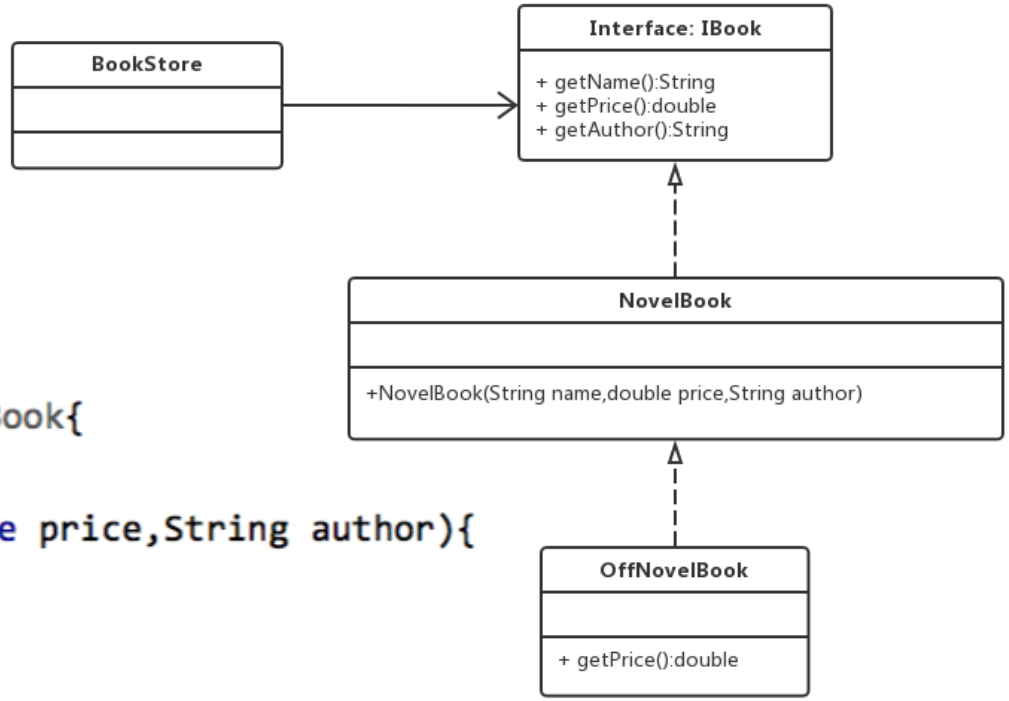
需修改实现类，同时导致有两个获取价格的方法

方案4：为NovelBook类增加一个OffNovelBook子类并覆写其中的getPrice()方法

不修改已有的类或接口，符合开闭原则

开闭原则实例分析-3

修改后的设计




```
public class OffNovelBook extends NovelBook{

   public OffNovelBook(String name,double price,String author){
      super(name,price,author);
   }

   //覆写价格方法，当价格大于 40，就打 8 析，其他价格就打 9 析
   public double getPrice(){
     if(this.price > 40){
        return this.price * 0.8;
     }else{
        return this.price * 0.9;
     }
   }
}
```

Liskov替换原则

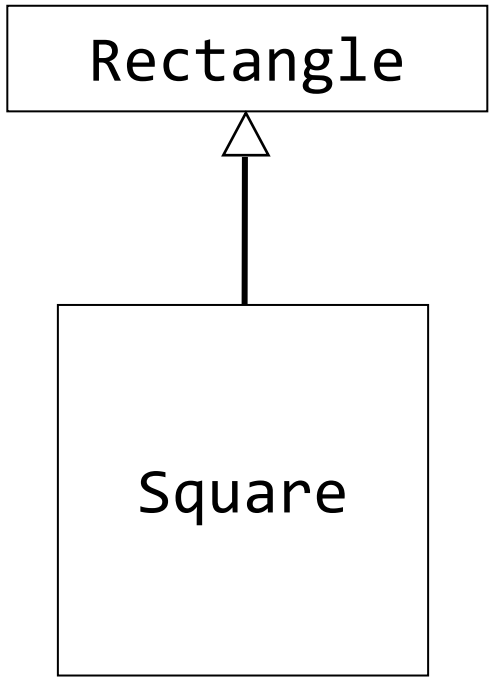
Is a circle a kind of ellipse?

public void setSize(int x, int y);
requires nothing
ensures after the call, the ellipse is
x units wide and y units high

public void setSize(int x, int y);
requires x = y
ensures after the call, the ellipse is
x units wide and y units high

子类对象应该可以替换父类对象出现的地方

Liskov替换原则实例分析-1



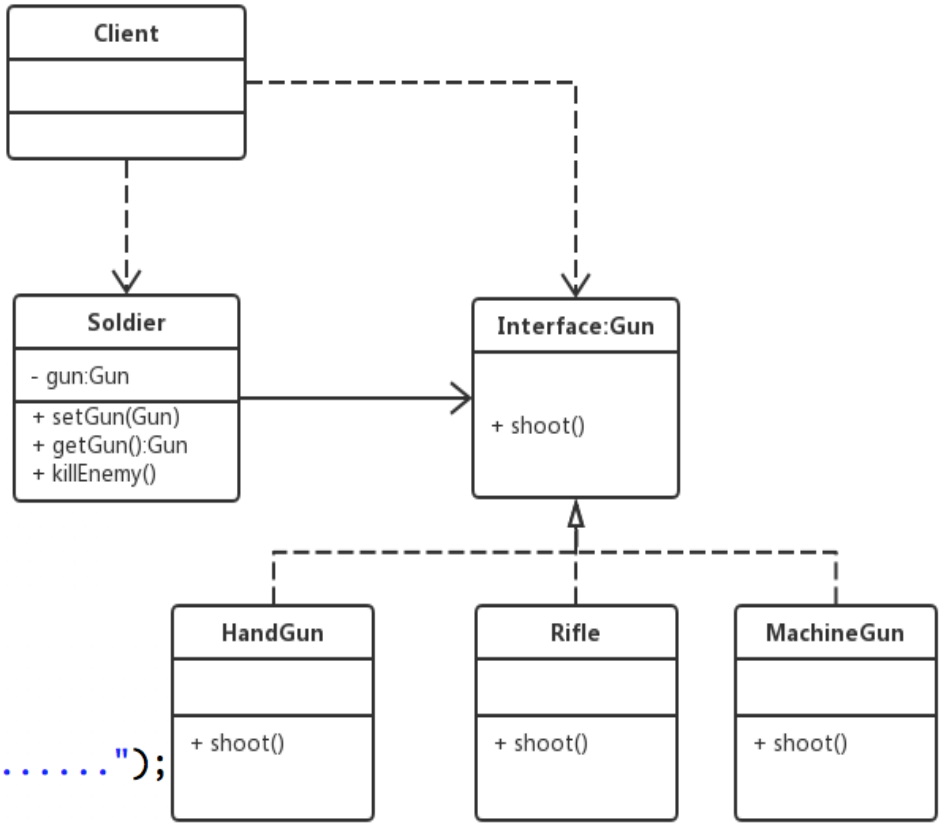

```
class Rectangle{
public void setWidth(double width);
public void setHeight(double height);
public double getArea();
…
};
```

```
Square square = new Square();
Rectangle rect = square;
rect.setWidth(3.0);
rect.setHeight(5.0);
```

有问题的设计：getArea的返回值是多少？

Liskov替换原则实例分析-2

```
public interface Gun {
    void shoot();
}

public class HandGun implements Gun {
    public void shoot() {
        System.out.println("手枪 bang...");
    }
}

public class MachineGun implements Gun {
    public void shoot() {
        System.out.println("机关枪 bang bang bang......");
    }
}

public class Rifle implements Gun {
    public void shoot() {
        System.out.println("步枪 bang bang ...");
    }
}
```



合理的设计

Liskov替换原则实例分析-3

```java
public class Soldier {

    private Gun gun;

    public Gun getGun() {
        return gun;
    }

    public void setGun(Gun gun) {
        this.gun = gun;
    }

    void killEnemy() {
        this.gun.shoot();
    }

}
```

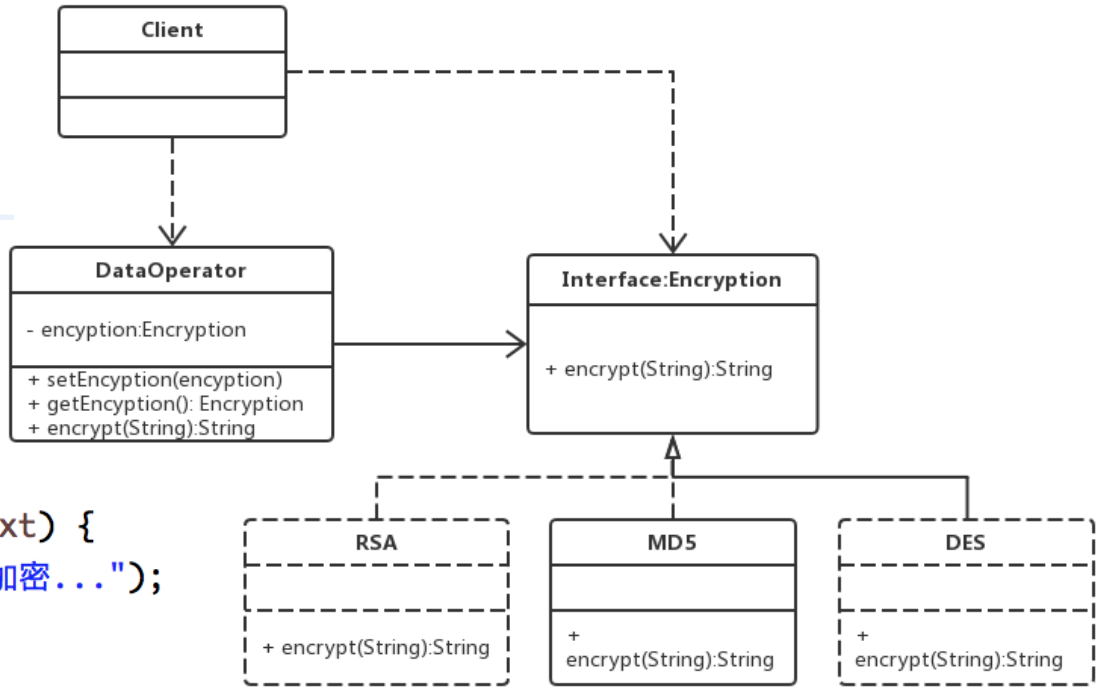
```java
public class Client {
    public static void main(String[] args) {
        Gun m416 = new Rifle();
        Soldier soldier = new Soldier();
        soldier.setGun(m416);//用子类代替父类
        soldier.killEnemy();
    }
}
```

运行结果

```
步枪 bang bang ...
```

Liskov替换原则实例分析-4

```java
public interface Encryption {
    String encrypt(String plainText);
}
```




```java
public class MD5 implements Encryption {
    public String encrypt(String plainText) {
        System.out.println("已使用MD5算法加密...");
        return plainText;
    }
}
```

```java
public class DES implements Encryption{
    public String encrypt(String plainText) {
        System.out.println("已使用DES算法加密...");
        return plainText;
    }
}
```

```java
public class RSA implements Encryption{
    public String encrypt(String plainText) {
        System.out.println("已使用RSA算法加密...");
        return plainText;
    }
}
```

合理的设计

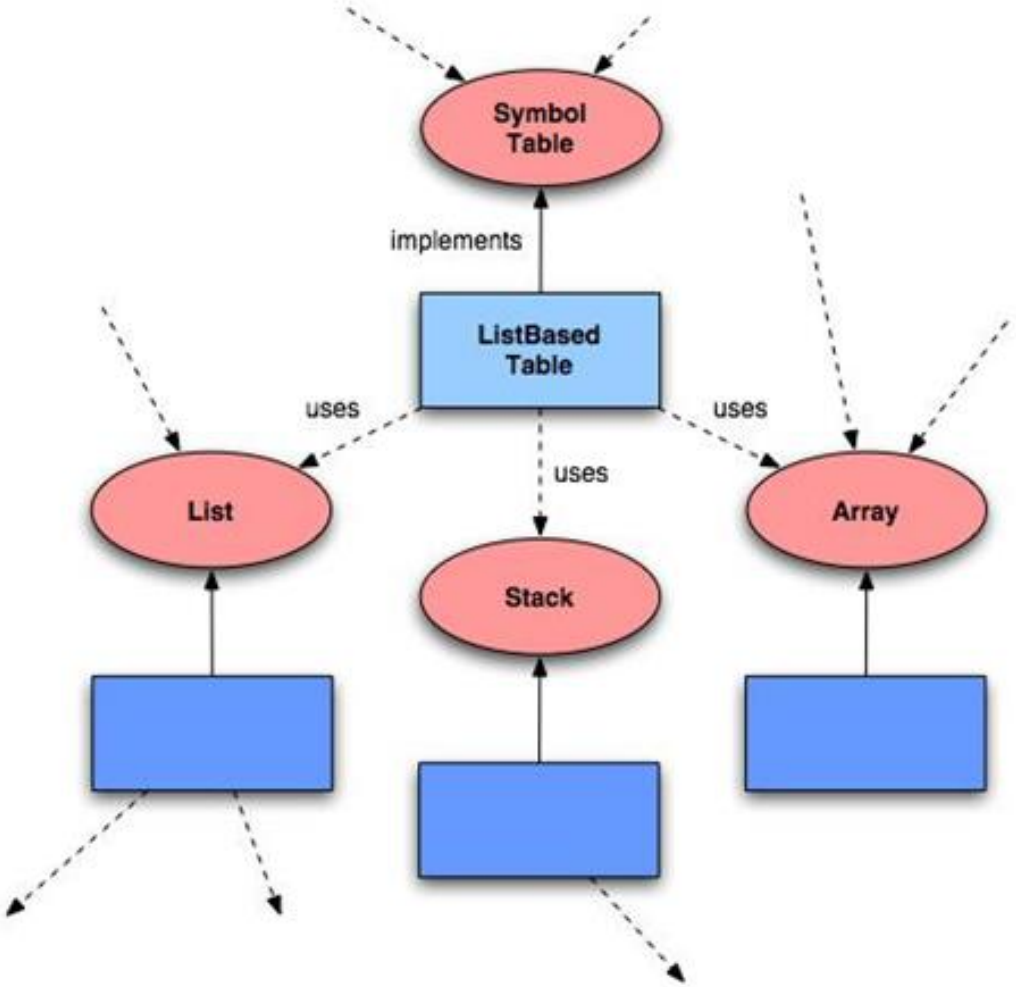
Liskov替换原则实例分析-5

```
public class DataOperator {
    private Encryption encyption;
    public Encryption getEncyption() {
        return encyption;
    }
    public void setEncyption(Encryption encyption) {
        this.encyption = encyption;
    }
    public String encrypt(String plainText) {
        return this.encyption.encrypt(plainText);
    }
}
public class Client {
    public static void main(String[] args) {
        DataOperator da = new DataOperator();
        Encryption e = new RSA();
        da.setEncyption(e);
        da.encrypt("hello world");
    }
}
```

运行结果

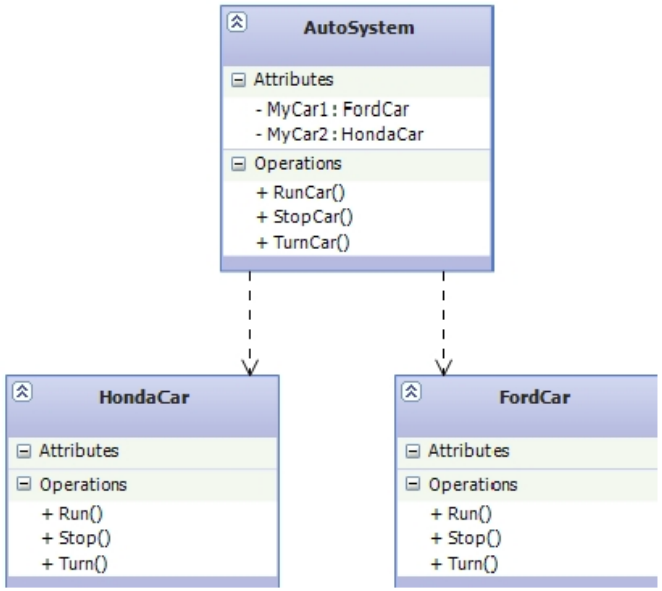
已使用RSA算法加密...

依赖转置原则



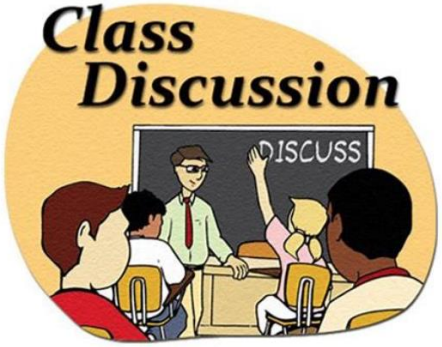
依赖于抽象（抽象类、接口）而非具体

依赖转置原则实例分析-1



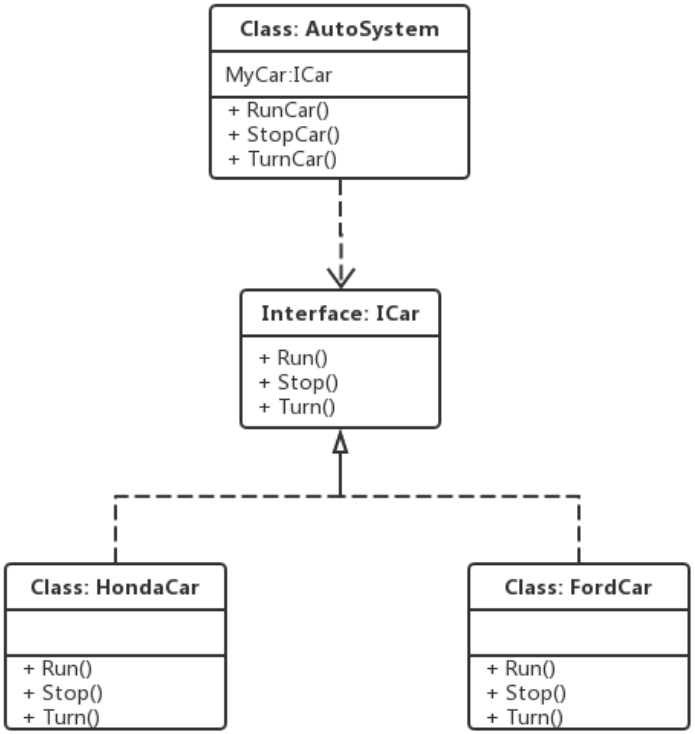
一个自动驾驶系统需要支持不同的车型，因此建立了对这些车型的依赖。

这个设计有什么问题吗？



脆弱、僵硬的设计：随着该系统不断支持新的车型，自动驾驶系统类将不断修改并依赖大量的车型实现类，任何车型发生修改都有可能导致自动驾驶系统的修改。

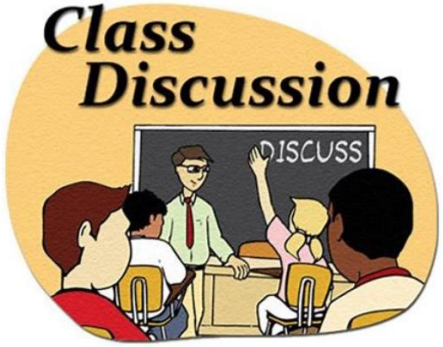
依赖转置原则实例分析-2



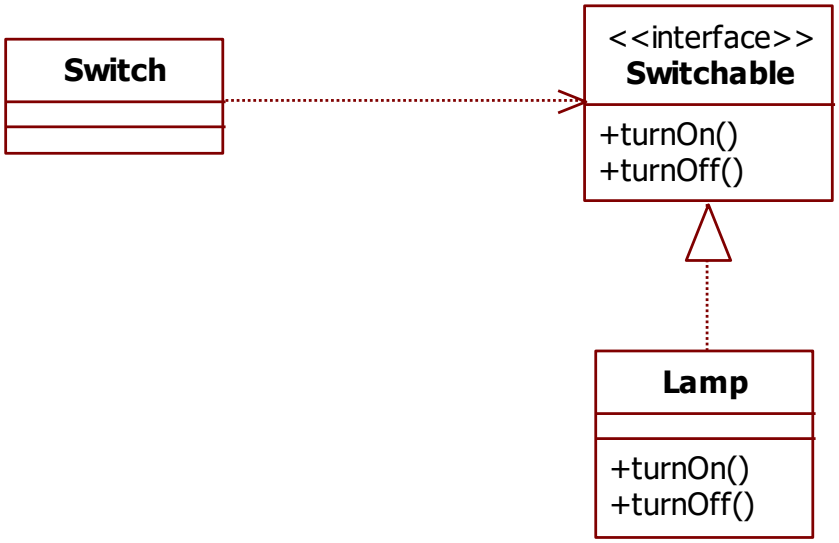
改进后的设计：自动驾驶系统依赖于抽象的汽车接口而不是具体的车型。汽车接口抽象出了自动驾驶所需的共性汽车控制功能。

依赖转置原则实例分析-3

如何改进设计，让同样的Switch可以被用于控制风扇？



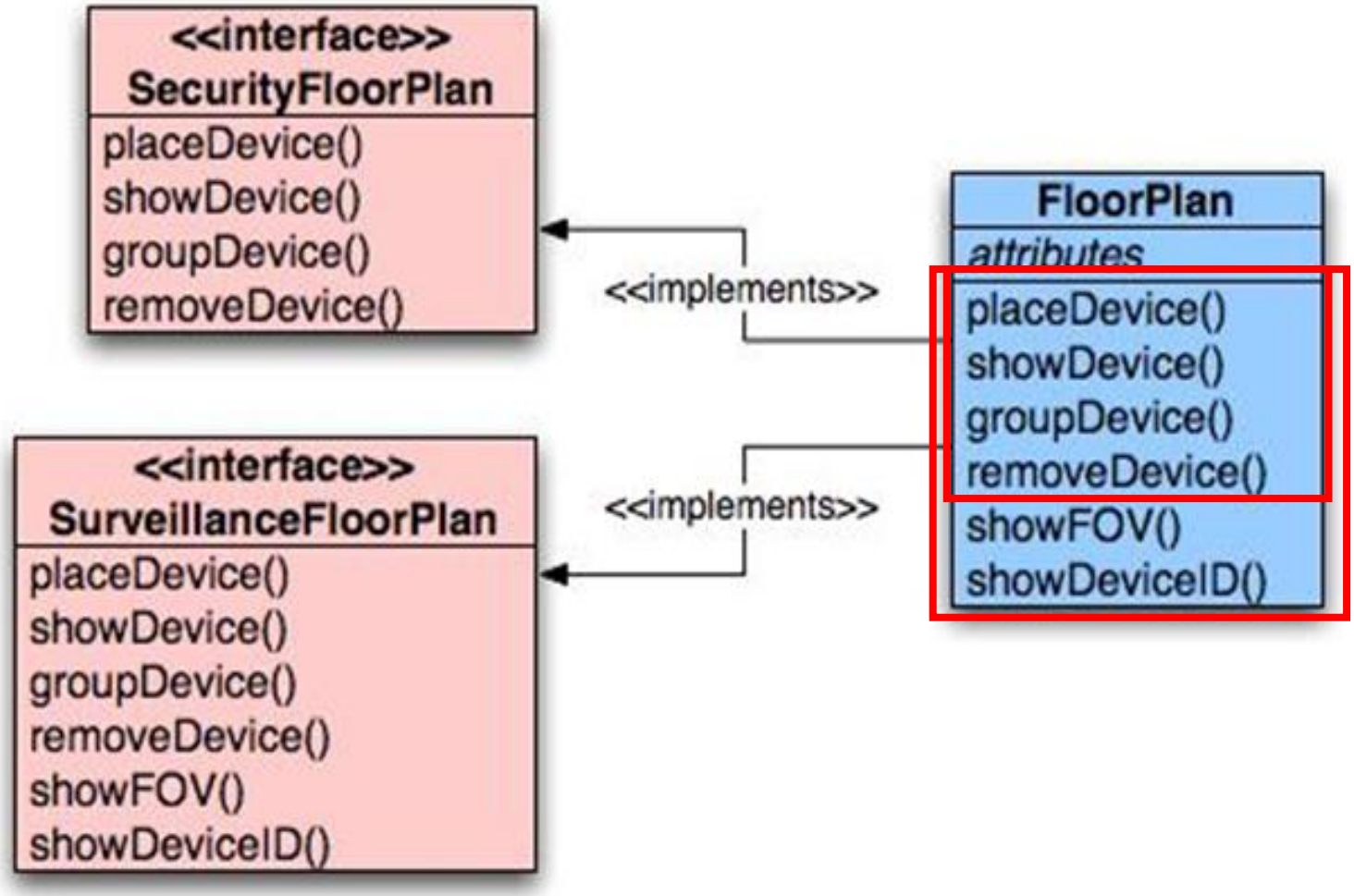
依赖转置原则实例分析-4




```
public class Switch {
    public Switch (Switchable switchable) {
        this. switchable = switchable;
    }
    ...
    private Switchable switchable;
}
public class DependencyInjectionExample {
    public static void main(String[] args) {
        Switchable lamp = new Lamp();
        Switch switch = new Switch(lamp);
    ...
    }
}
```

依赖转置+依赖注入

接口隔离原则

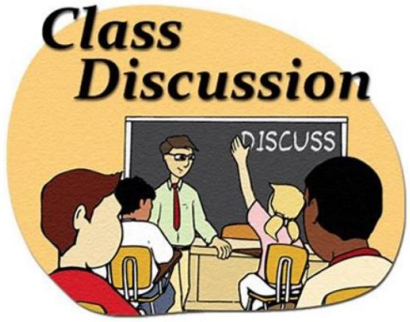


不要强迫客户模块依赖它们不会使用的接口

多个服务于特定请求方的接口好过一个通用接口

接口隔离原则实例分析-1

```
Interface Rectangle{
   public void draw();
   public double area();
};
```

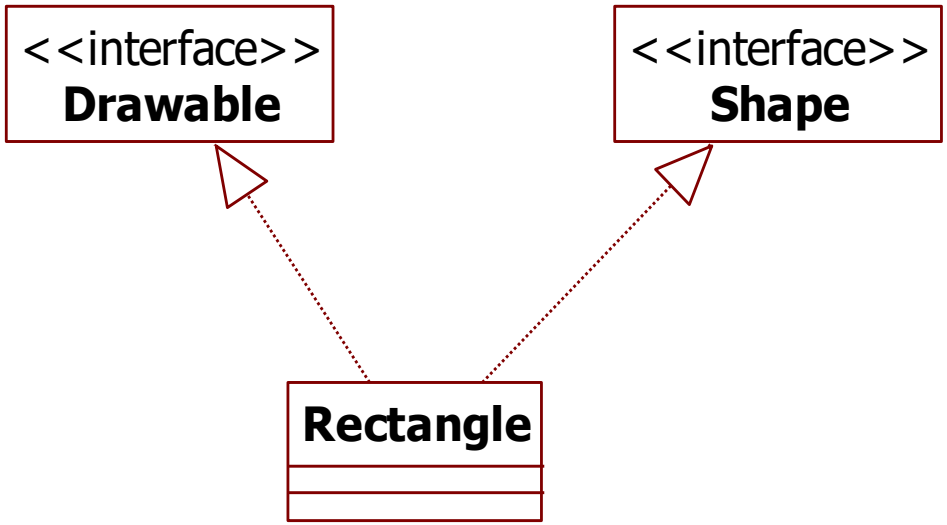


这个矩形接口的设计有什么问题吗？

问题：内聚性不足，同时违背接口隔离原则

假如客户端代码只需要Rectangle的面积计算，它是否需要了解和绘图相关的内容？如果绘图本身依赖于一个很复杂的库呢？如果绘图库是平台相关的呢？

接口隔离原则实例分析-2



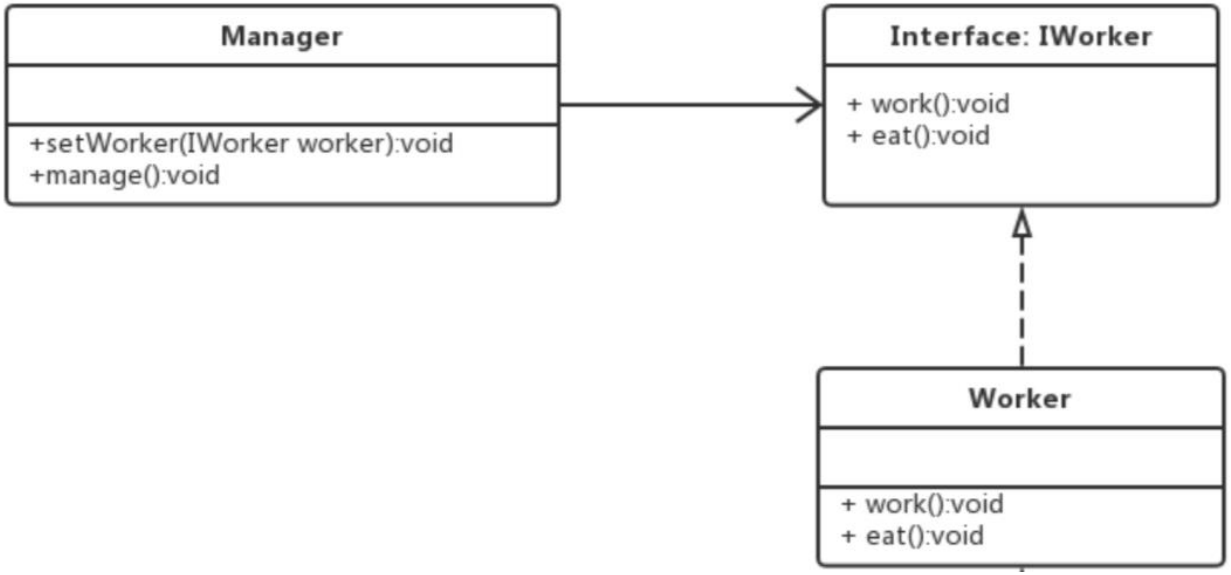
改进的矩形接口设计：将绘图与几何形状相关的职责相隔离

（1）简化了调用方需要理解的概念

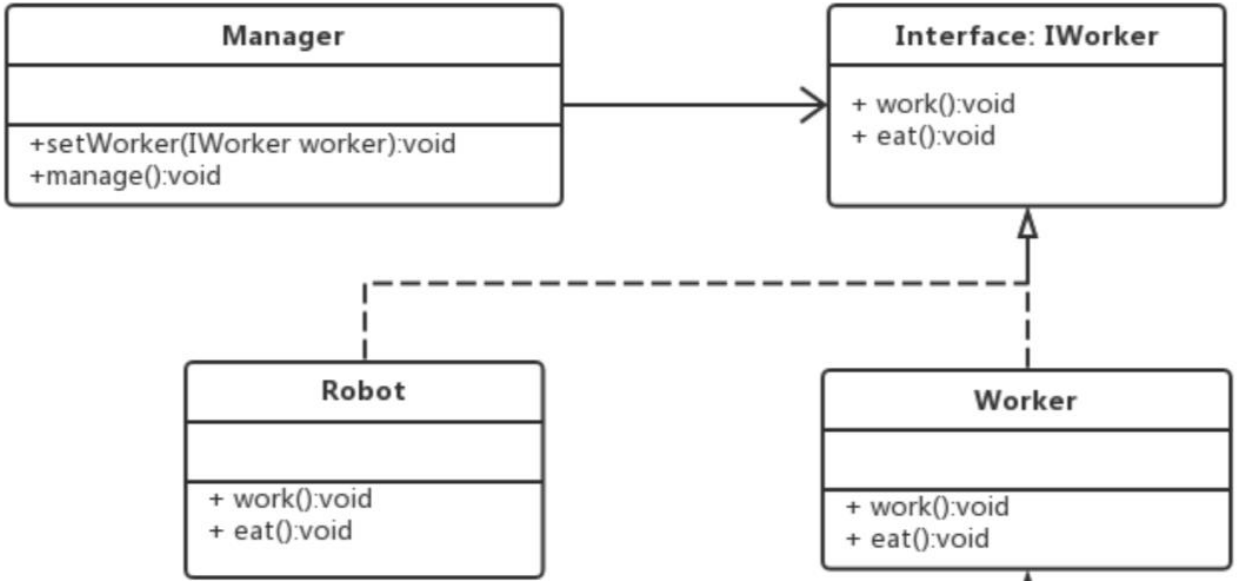
（2）降低了类间的耦合

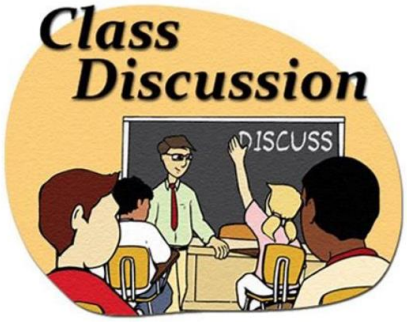
（3）提高了对变化的适应能力

接口隔离原则实例分析-3

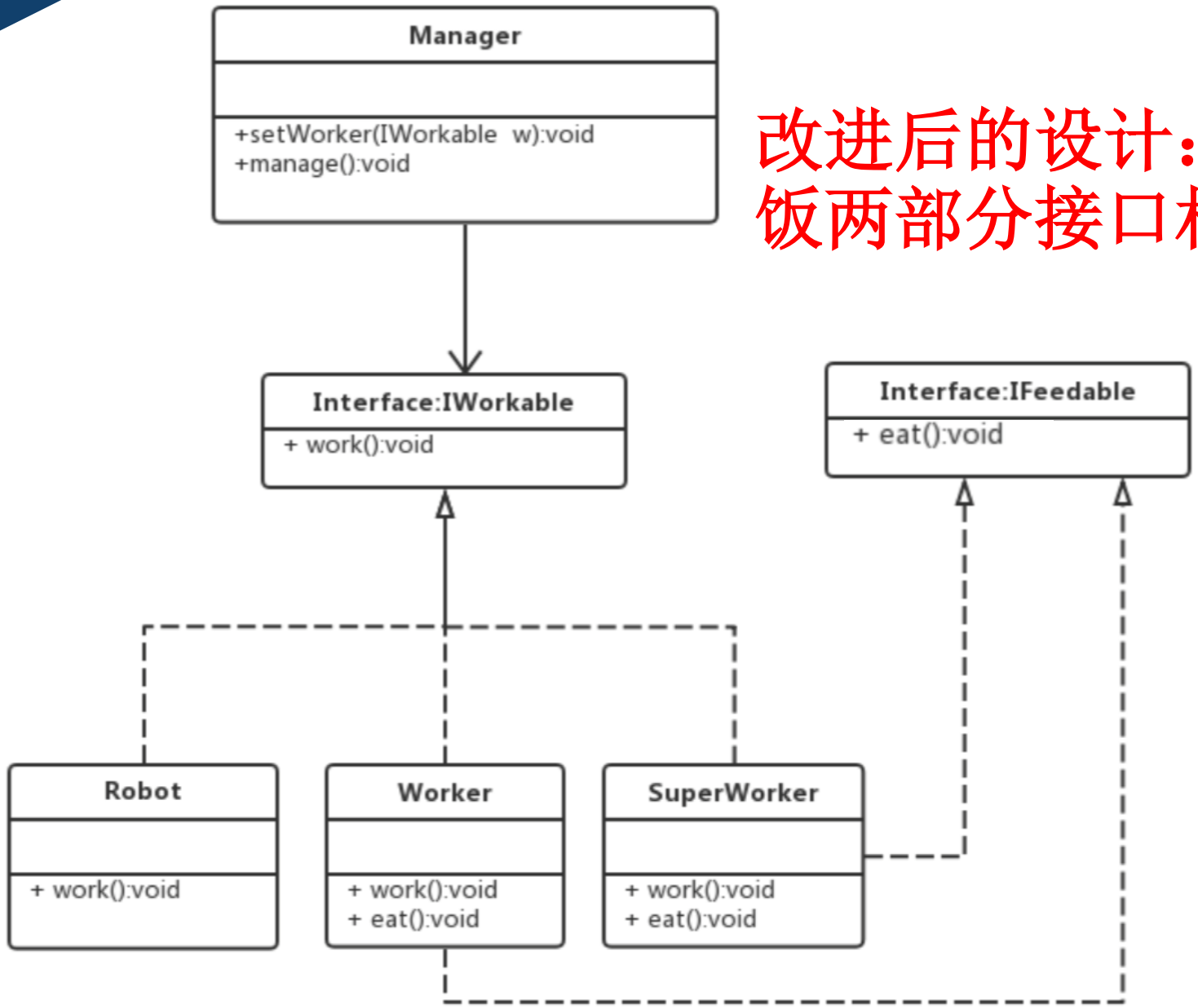


管理者管理工人，后来增加了机器人来承担工人的任务。这个设计暴露出什么问题？





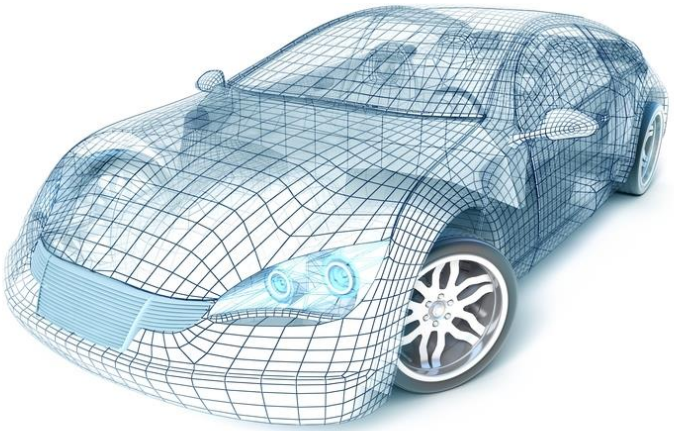
接口隔离原则实例分析-4



改进后的设计：工作与吃饭两部分接口相互隔离

模型

- 按照需要对现实世界对象物进行抽象
- 可用于模拟和分析对象物的各种特性
- 降低了整体成本
  - ✓ 修改模型要比修改一个真实的系统要容易的多
  - ✓ 同时对多种可能的解决方案进行对比和选择
  - ✓ 方便相关人员建立一致的理解



面向对象类设计的描述

- 静态结构模型
  - ✓ 使用静态类及其关系描述系统的静态结构
  - ✓ 需要描述的关系包括：泛化（继承）、使用/被使用、组合等
- 动态交互模型
  - ✓ 描述系统的动态结构并展示系统对象之间的运行时交互
  - ✓ 需要描述的交互包括：对象发出的服务请求的序列、由这些对象交互所触发状态变化

统一建模语言UML（Unified Modeling Language）

- 由13种不同的图形类型组成，可用于建模软件系统
- 在1990年代的面向对象建模方面的工作基础上出现的
- 目前流行的UML 2.0于2004年定稿
- 一种广泛采用的软件建模方法

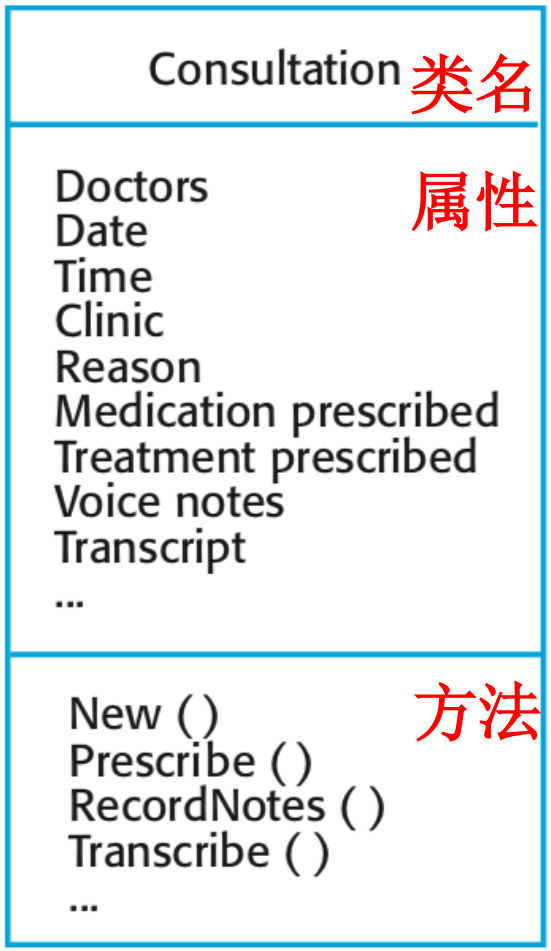
http://software-engineering-book.com/web/uml/

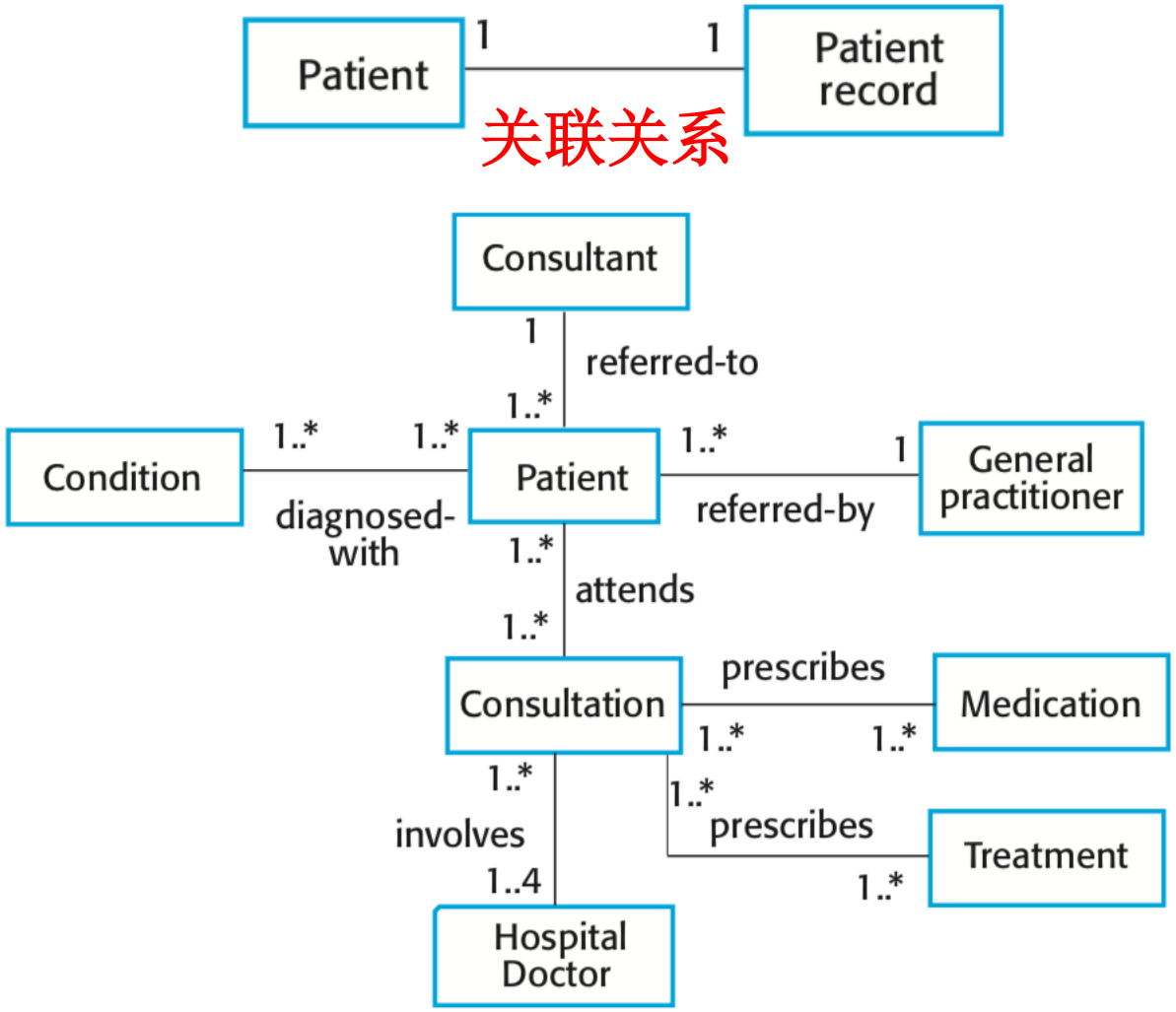
UML类模型

- 类：类名、属性、方法
- 类间关系
  - ✓ 关联（Association）
  - ✓ 依赖（Dependency）
  - ✓ 整体/部分：组合（Composition）
    聚合（Aggregation）
  - ✓ 泛化/特殊：继承（Inheritance）
  - ✓ 实现（Implementation）

类间关系的强度：Generalization = Implementation > Composition > Aggregation > Association > Dependency

类与关联



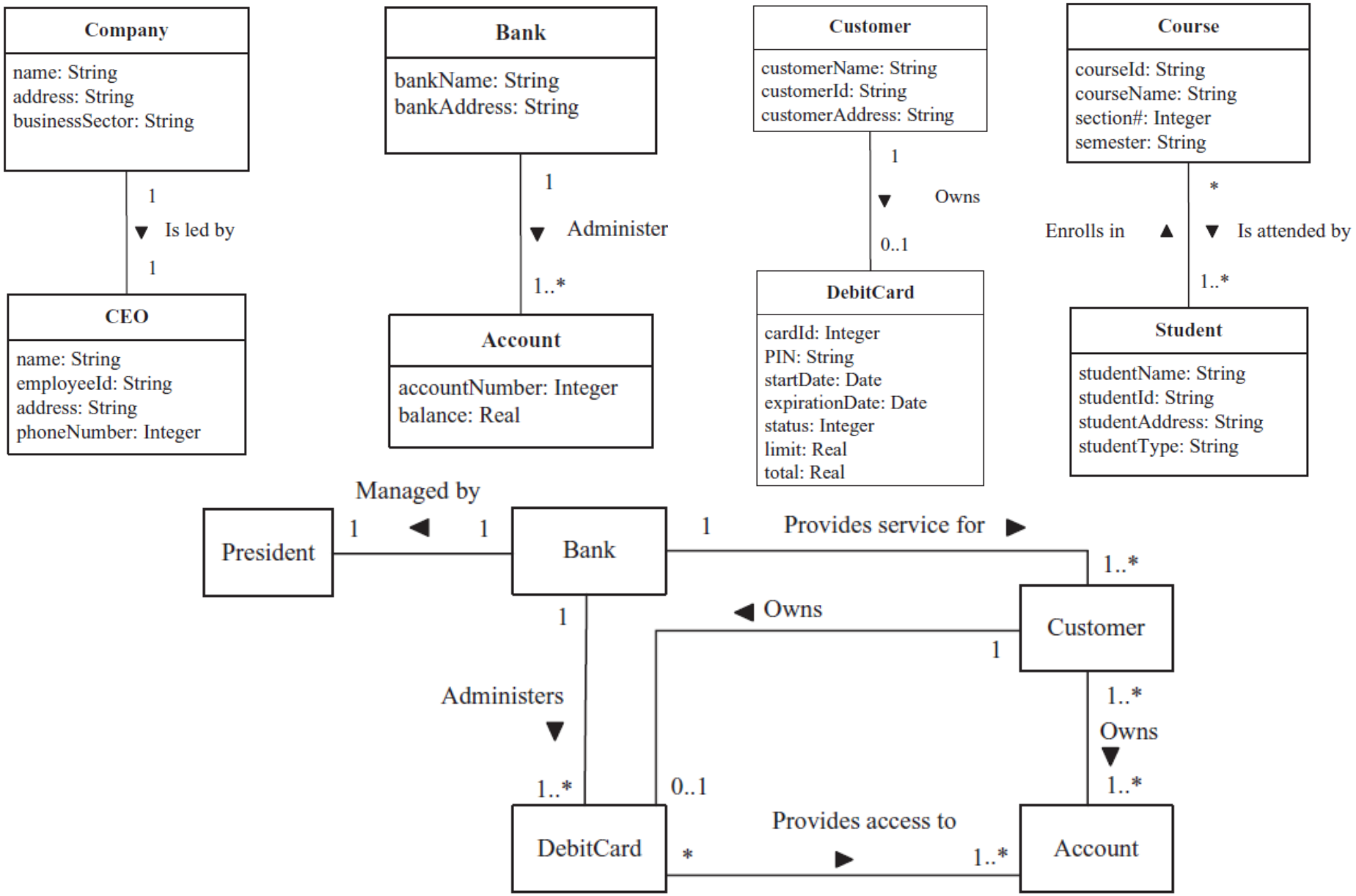


《软件工程》5.3（Mentcare系统）

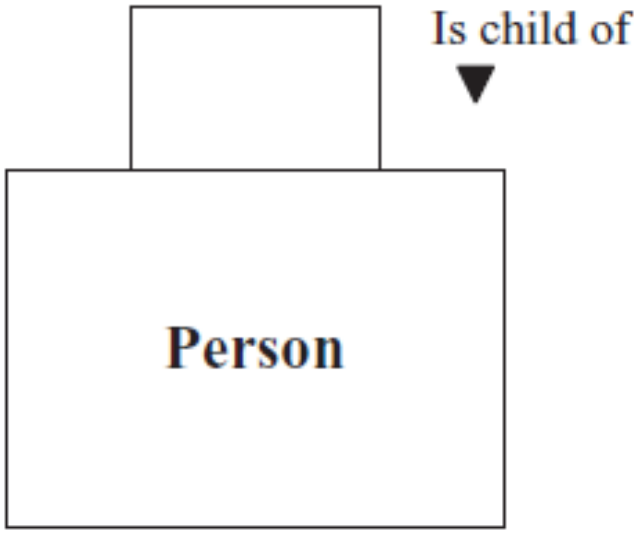
关联关系

- 关联定义了两个或多个类之间的一种静态结构关系
- 关联本质上是双向的
  - ✓ 正向与反向均可解读，例如
    Employee *Works in* Department
    Department *Employs* Employee
- 在类图中一般是从上到下、从左到右进行解读

关联关系示例


Company
name: String
address: String
businessSector: String
1
Is led by
1
CEO
name: String
employeeId: String
address: String
phoneNumber: Integer
Bank
bankName: String
bankAddress: String
1
Administer
1..*
Account
accountNumber: Integer
balance: Real
Customer
customerName: String
customerId: String
customerAddress: String
1
Owns
0..1
DebitCard
cardId: Integer
PIN: String
startDate: Date
expirationDate: Date
status: Integer
limit: Real
total: Real
Course
courseId: String
courseName: String
section#: Integer
semester: String
*
Enrolls in
Is attended by
1..*
Student
studentName: String
studentId: String
studentAddress: String
studentType: String
Managed by
President
1
1
Bank
1
Provides service for
1..*
Customer
Owns
1
1..*
Owns
1..*
1
Administers
1..*
0..1
DebitCard
Provides access to
*
1..*
Account

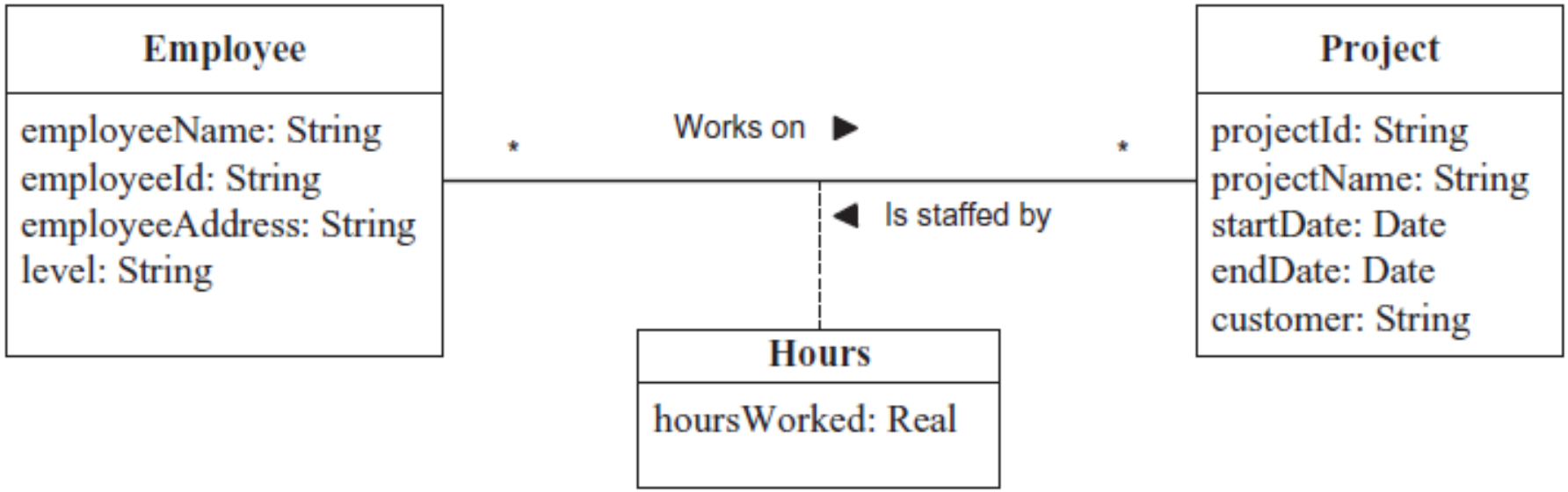
关联关系示例



一元（自体）关联

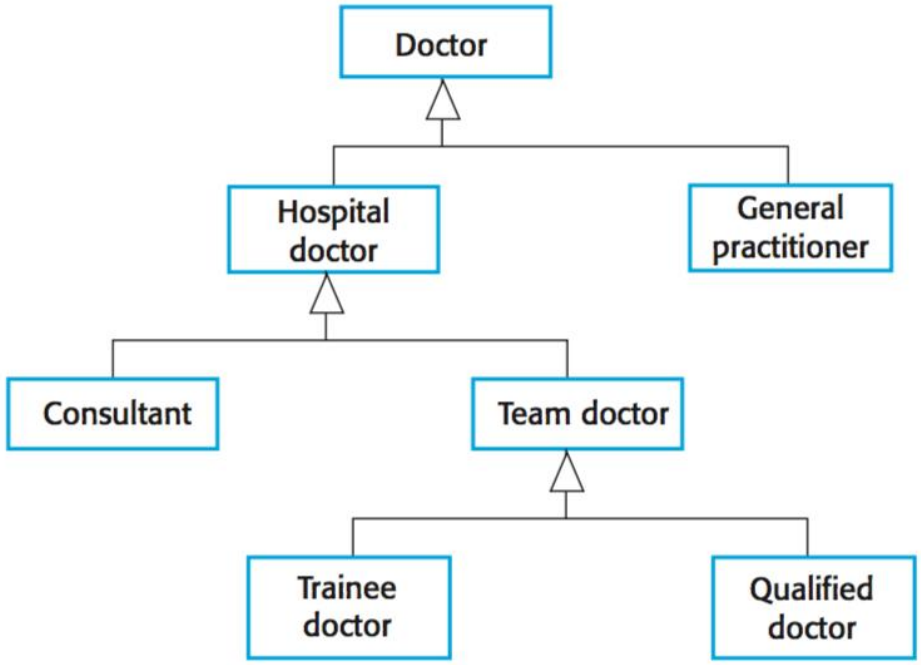
关联类

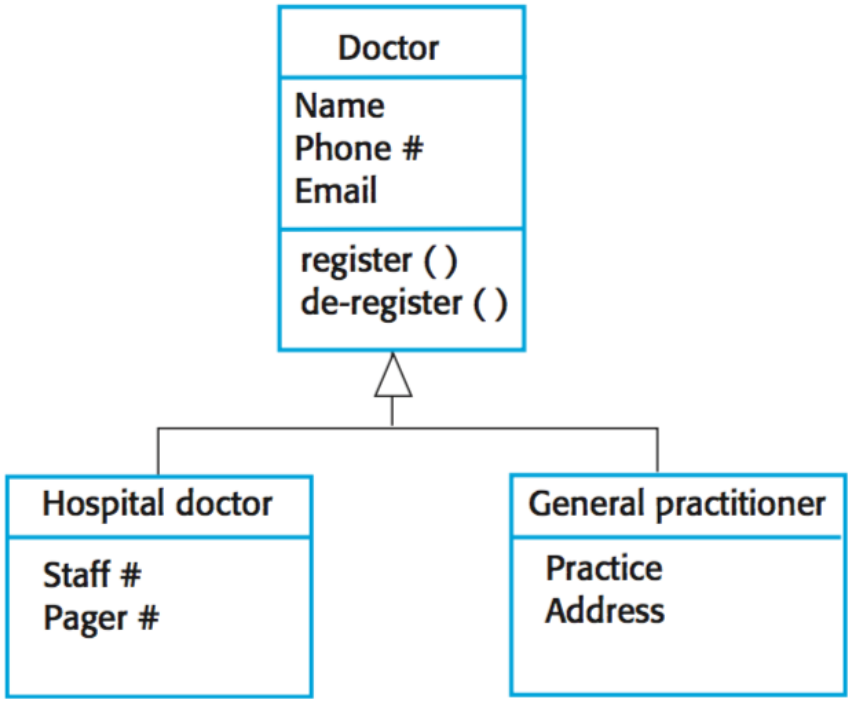
- 两个类之间的关联关系上本身包含属性
- 三个或三个以上类之间的关联关系
- 关联类的属性属于关联关系本身



类的泛化（继承）

代表从特殊到一般（isa）的关系



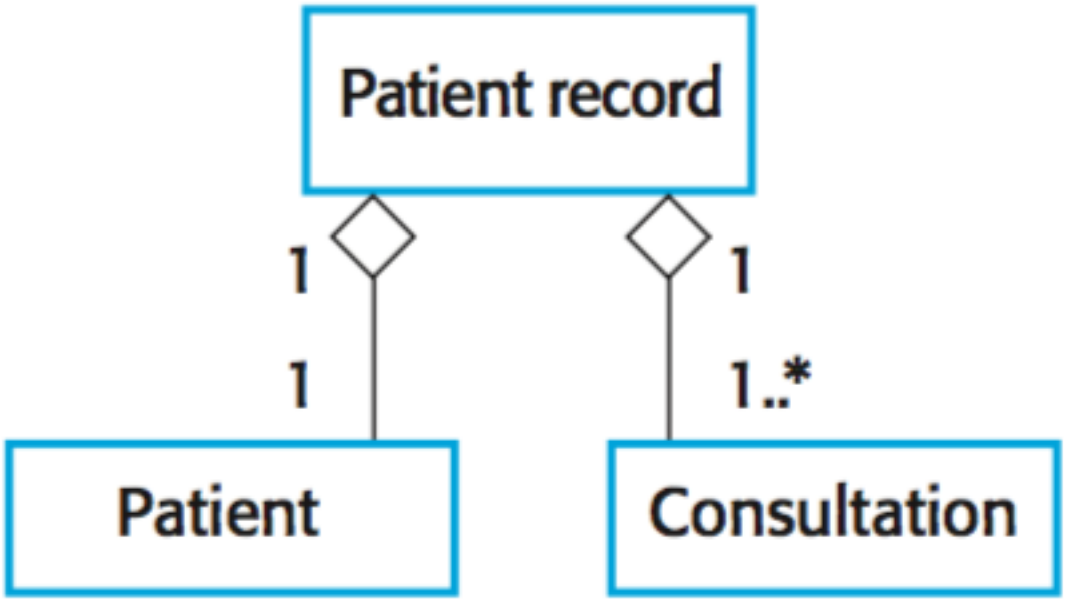


子类可增加更多细节

《软件工程》5.3（Mentcare系统）

类的聚集

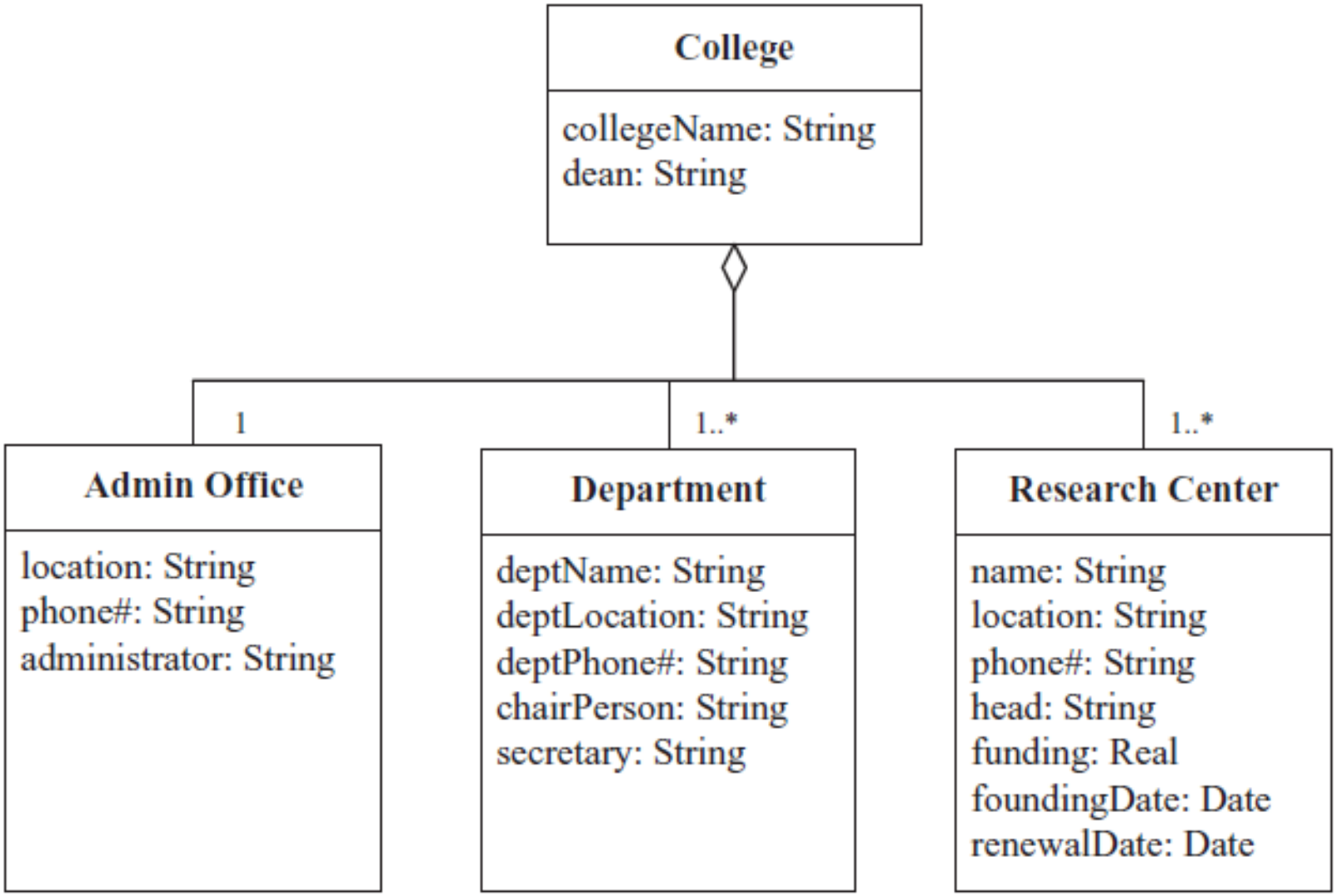
代表整体与部分的关系



类的聚集：组合与聚合

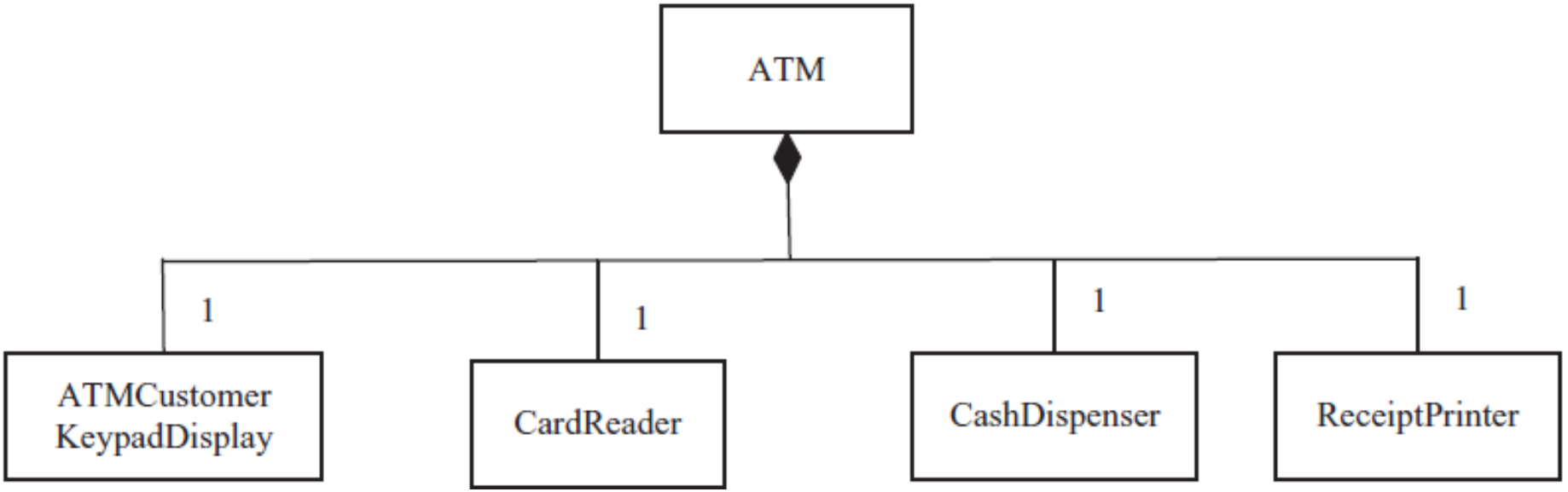
- 组合与聚合都代表整体与部分之间的关系
- 区别在于关系的强弱
  - ✓ 组合：整体与部分之间往往存在物理的包含关系，整体完全拥有部分
  - ✓ 聚合：整体与部分之间往往不存在物理的包含关系，部分可以加入或退出整体

类的聚合示例



一种较弱的整体部分关系

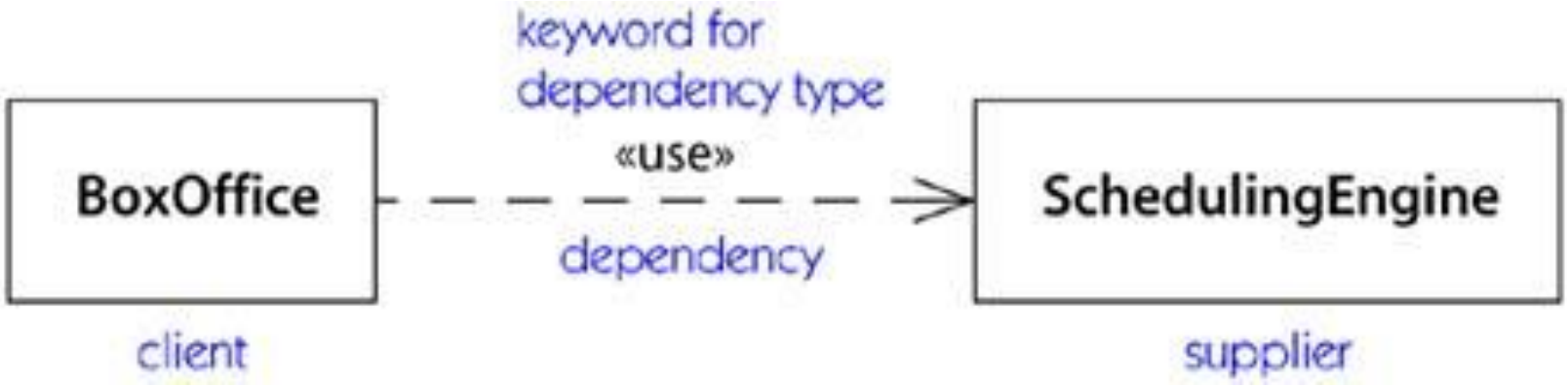
类的组合示例



一种较强的整体部分关系

依赖关系

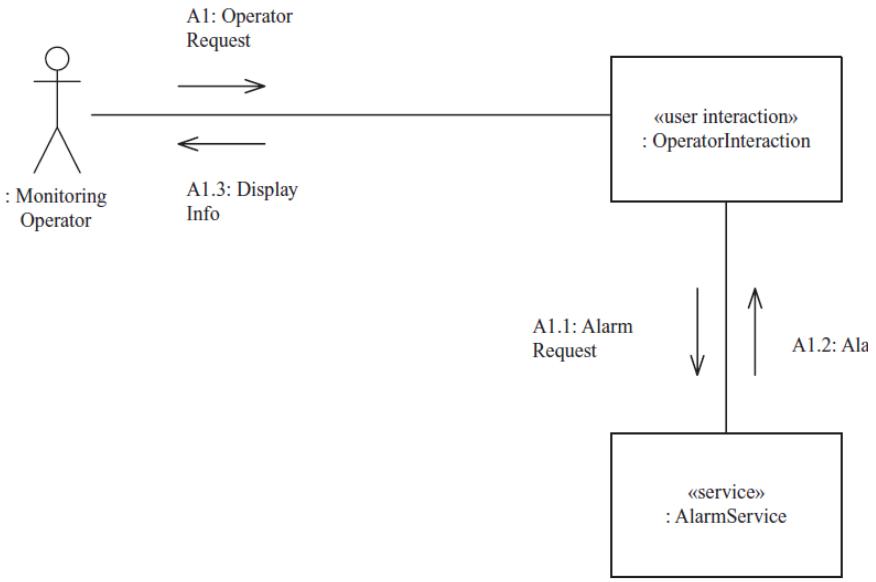
- 依赖是一种较弱的关联关系
- 一个类依赖于另一个类协作完成任务，关联关系只在一定时间段内存在

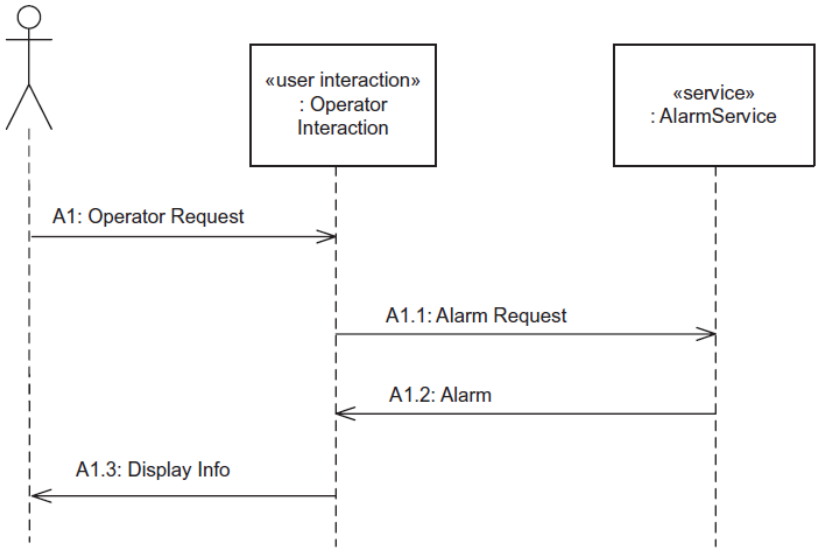


动态交互模型

- 描述相关对象如何通过相互协作实现特定场景的执行
- 可以使用UML顺序图或通信图来表示
  - ✓ 顺序图：按照时间线展示消息发送的序列，突出时间顺序
  - ✓ 通信图：展示一组对象之间的消息交互，突出交互拓扑结构
- UML顺序图使用更为广泛

UML顺序图与通信图对比

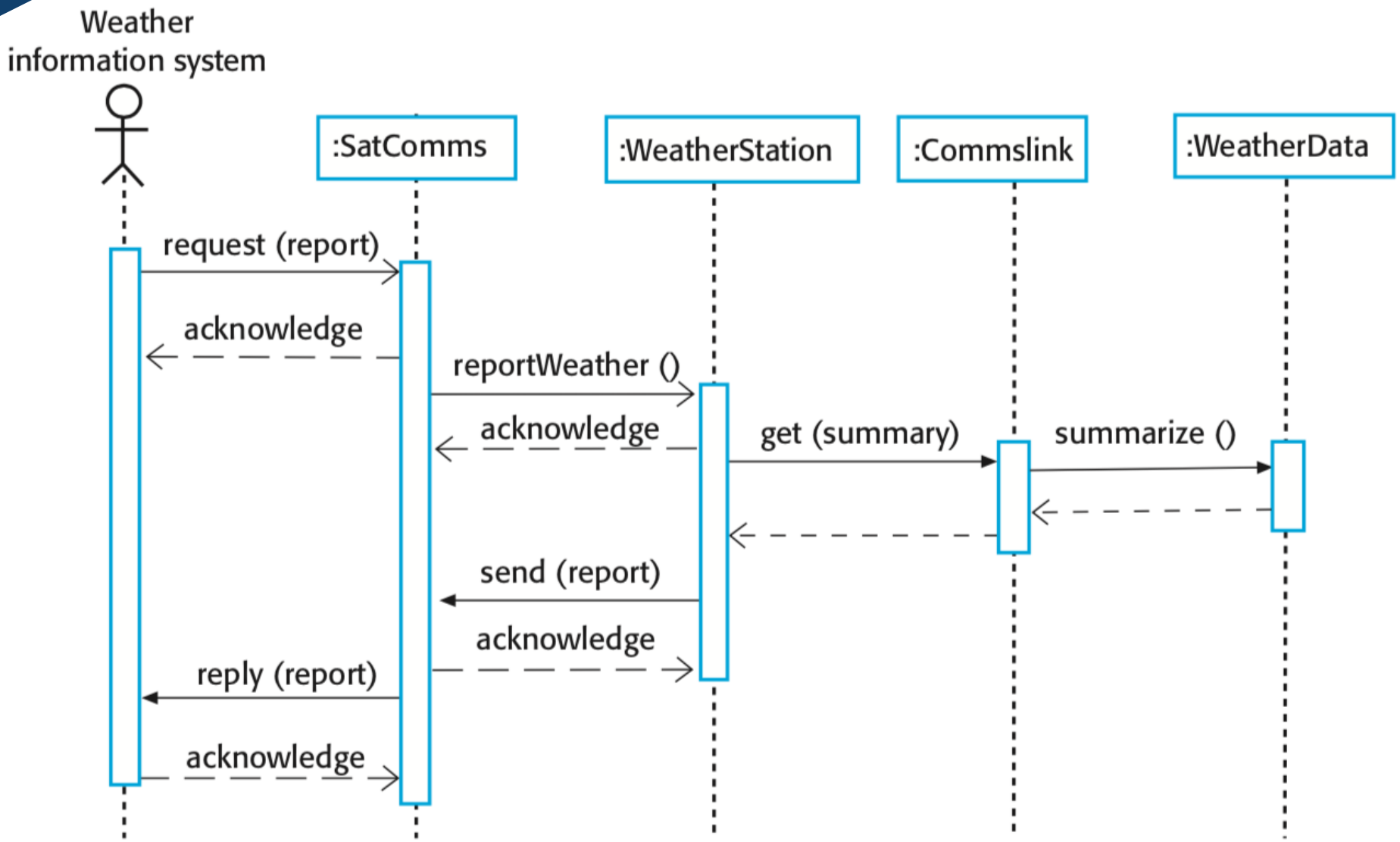




通信图

顺序图

UML顺序图示例



《软件工程》7.1.4（WeatherStation系统）

阅读建议

- 《软件工程》5.3、7.1
- 《代码大全》第5章

快速阅读后整理问题
在QQ群中提出并讨论

CS2001
软件工程

End

9. 软件设计
—面向对象软件设计