廣東技術師範大學
Guangdong Polytechnic Normal University

《系统分析与设计》课程

Instructor: Wen Jianfeng
Email: wjfgdin@qq.com

系统分析与设计

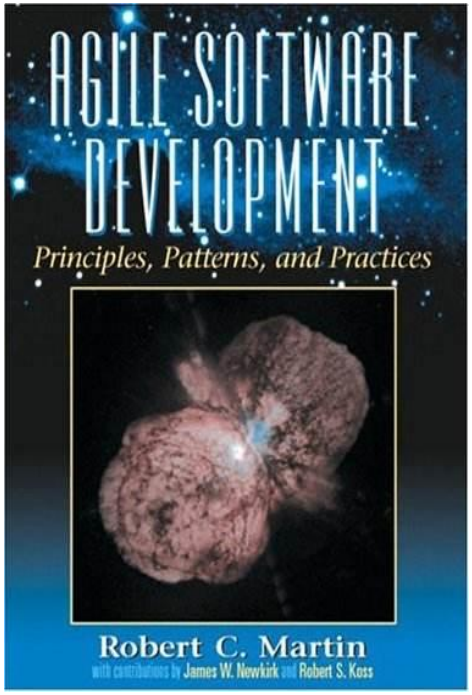
第4讲：面向对象设计原则

本讲内容

- SOLID原则
  - 单一职责原则（Single Responsibility Principle）
  - 开闭原则（Open-Closed Principle）
  - 里氏替换原则（Liskov Substitution Principle）
  - 接口隔离原则（Interface Segregation Principle）
  - 依赖倒置原则（Dependency Inversion Principle）
- 其他原则
  - 组合复用原则（Composite Reuse Principle）
  - 迪米特法则（Law of Demeter）
  - KISS原则（Keep It Simple and Stupid）
  - DRY原则（Don’t Repeat Yourself）

本讲主要参考书及资源

- 书名：Agile Software Development: Principles, Patterns, and Practices (2002)
  中译版【敏捷软件开发：原则、模式与实践】
  - 作者：Robert C. Martin
  - 出版社：Pearson



- 专栏：设计模式之美
  - 王争
  - 极客时间

本讲主要参考书及资源

- Java Design Patterns
  - https://java-design-patterns.com/principles/
- Programming Principles
  - https://github.com/webpro/programming-principles
- Principles Wiki
  - http://principles-wiki.net/principles:start

见参考书P95-98（英）、P88-91（中）

单一职责原则（SRP）

Single Responsibility Principle

单一职责原则（SRP）

- A class should have only one reason to change
  就一个类而言，应该仅有一个引起它变化的原因
  - Responsibility = Reason to change
    什么是职责？(职责 = 变化的原因)
  - Separate coupled responsibilities into separate classes
    (一个类只负责一项职责)

- Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class
  一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中

单一职责原则（SRP）

- SRP的变种/别名
  - One Responsibility Rule
  - Separation of Concerns
  - Curly’s Law（Do One Thing）

单一职责原则（SRP）

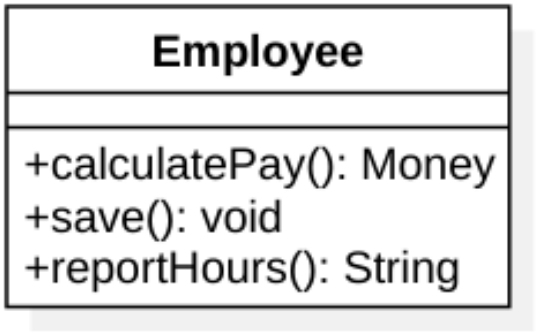
- 如何理解单一职责原则？
  - 一个类只负责完成一个职责或者功能
  - 不要设计大而全的类，要设计粒度小、功能单一的类
  - 类的职责包括两方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现
  - 单一职责原则是为了实现代码高内聚、低耦合，提高代码的复用性、可读性、可维护性

单一职责原则（SRP）

- 如何判断类的职责是否足够单一？
  - 类中的代码行数、方法或者属性过多
  - 类依赖的其他类过多，或者依赖该类的其他类过多
  - 私有方法过多
  - 比较难给类起一个合适的名字，比较难用简短的语句描述一个类
  - 类中大量的方法都是集中操作类中的某几个属性

单一职责原则（SRP）

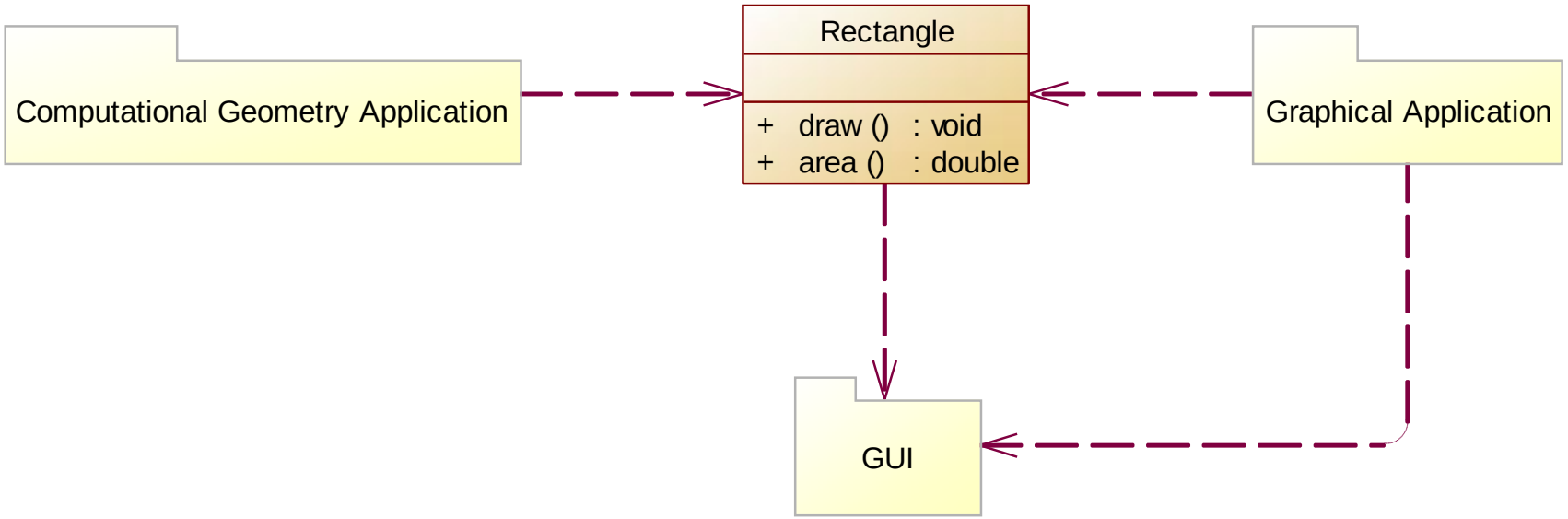
- 例如：不符合SRP的设计



  - 有3个原因会导致这个类的修改：
    - The business rules having to do with calculating pay.
    - The database schema.
    - The format of the string that reports hours.

单一职责原则（SRP）

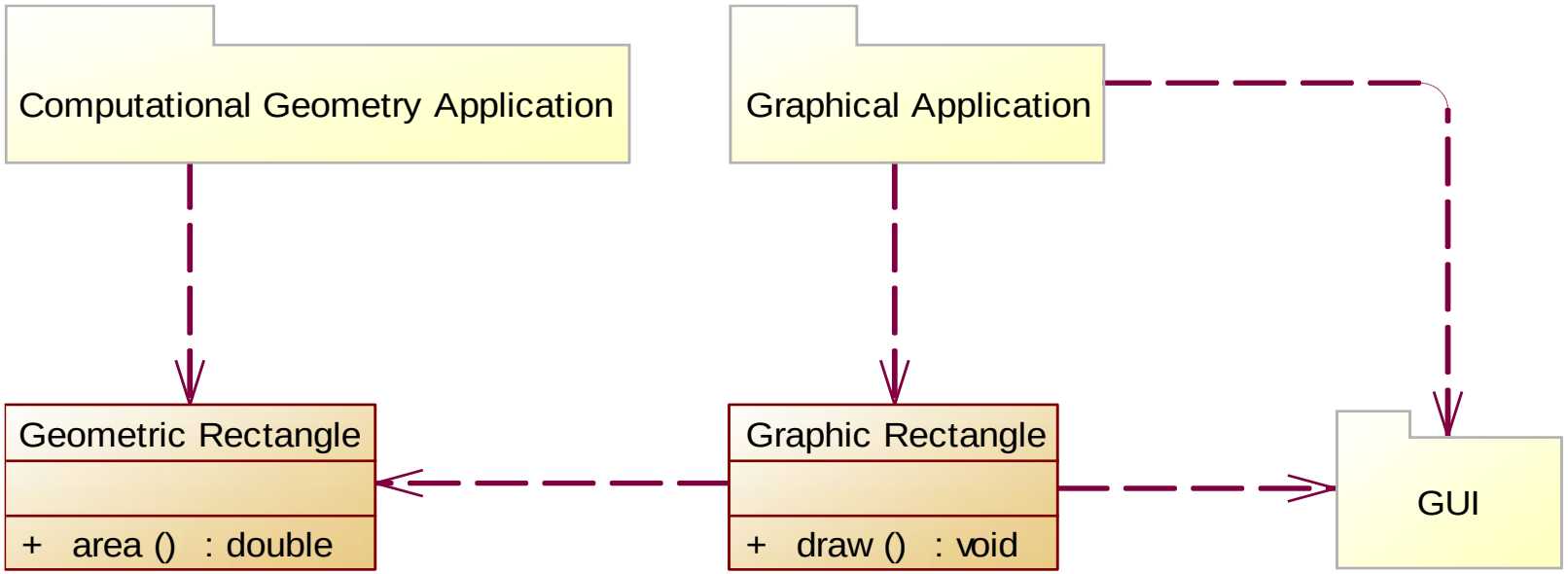
- 不符合SRP的设计：



- Rectangle类实现了两个互不相关的职责
- Rectangle类可能会受到两个应用的更改，当一个应用更改Rectangle时，会导致另一个应用受到影响

单一职责原则（SRP）

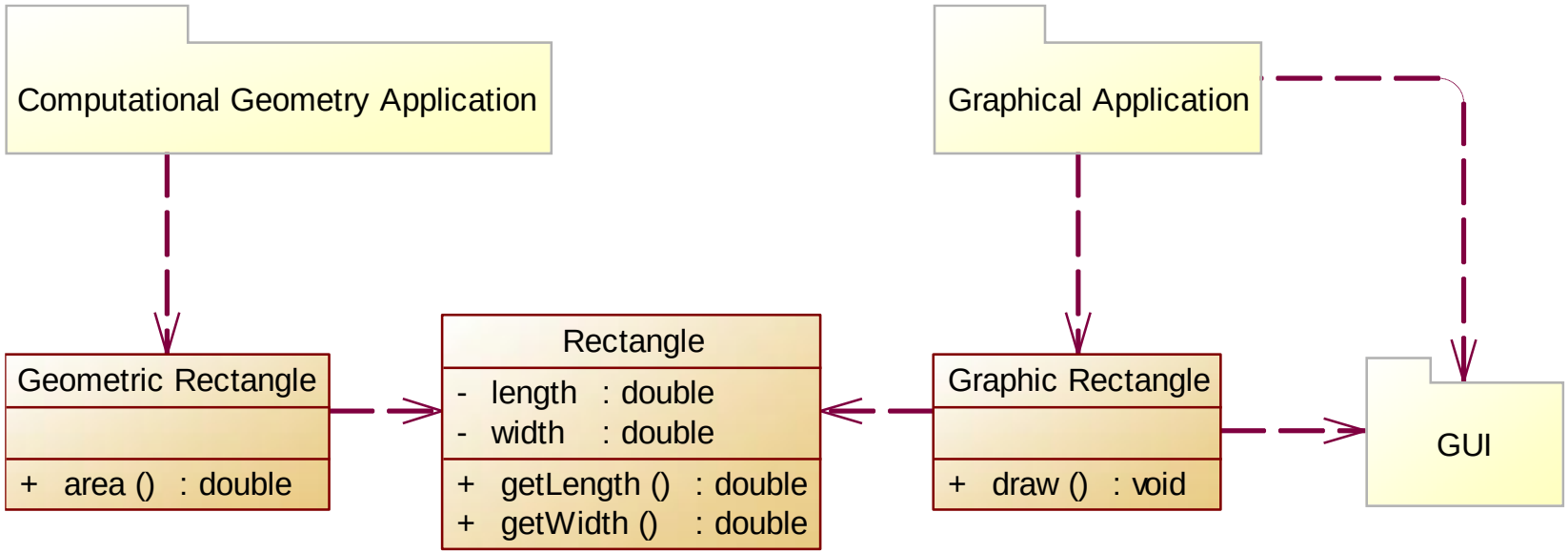
- 重构后…



- GA对Graphic Rectangle类的更改不会影响到CGA
- CGA对Geometric Rectangle类的更改仍会影响到GA

单一职责原则（SRP）

- 进一步改进设计…



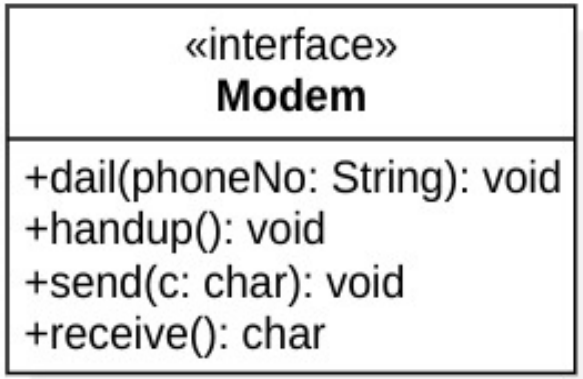
- Rectangle类包含最原始的属性和方法
- Geometric Rectangle类和Graphic Rectangle类相互独立，CGA和GA对它们的更改不会彼此影响

单一职责原则（SRP）

- 类的职责是否设计得越单一越好？
  - 单一职责原则通过避免设计大而全的类，避免将不相关的功能耦合在一起，来提高类的内聚性。同时，类职责单一，类依赖的和被依赖的其他类也会变少，减少了代码的耦合性，以此来实现代码的高内聚、低耦合。
  - 但是，如果拆分得过细，实际上会适得其反，反倒会降低内聚性，也会影响代码的可维护性。

单一职责原则（SRP）

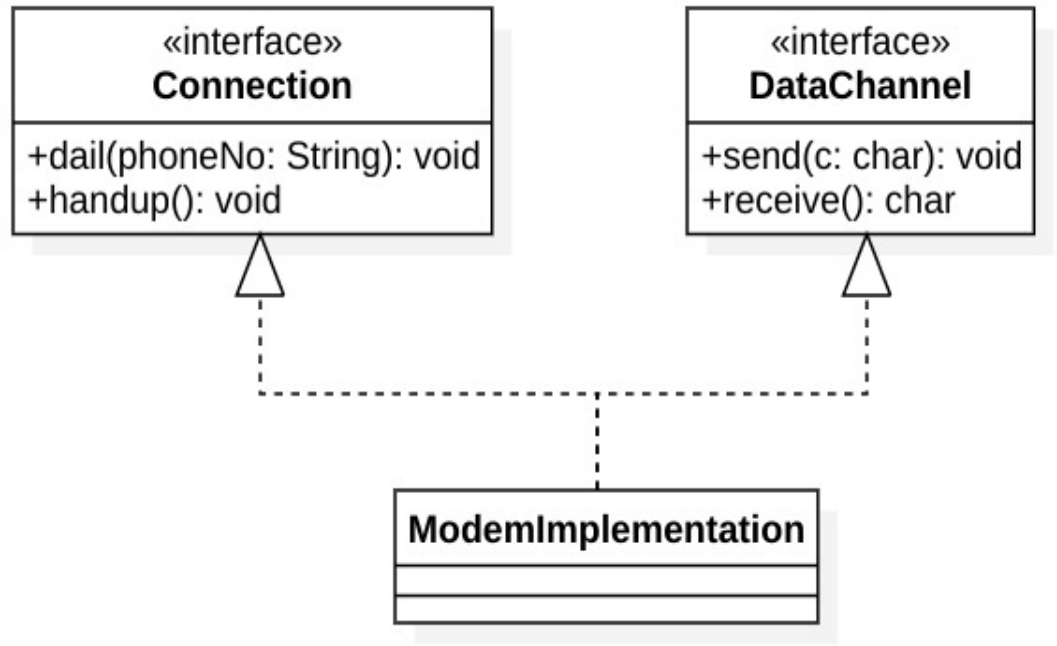
- 职责分离与否也要看具体的情形
  - 例如：Modem接口有两个职责
    - 连接管理（dail, handup）
    - 数据通信（send, recv）
  - 这两个职责要不要分离？
    - 取决于应用程序的变化方式



    - 如果应用程序的变化只单独影响到连接管理方法（dail, handup），或只单独影响到数据通信方法（send, recv），则应该分离以解耦两个职责
    - 如果应用程序的变化会同时影响到连接管理方法（dail, handup）和数据通信方法（send, recv），即这两个职责不可能会发生单独变化的情形，则不需要分离

单一职责原则（SRP）

- 如果需要分离职责，则可以这样设计：



单一职责原则（SRP）

- 遵循单一职责原则的好处
  - 可以降低类的复杂度
  - 提高类的可读性
  - 提高类的重用性
  - 提高系统的可维护性
  - 变更引起的风险降低

见参考书P99-109（英）、P92-101（中）

开闭原则 (OCP)
Open-Closed Principle

开放/封闭原则 (OCP)

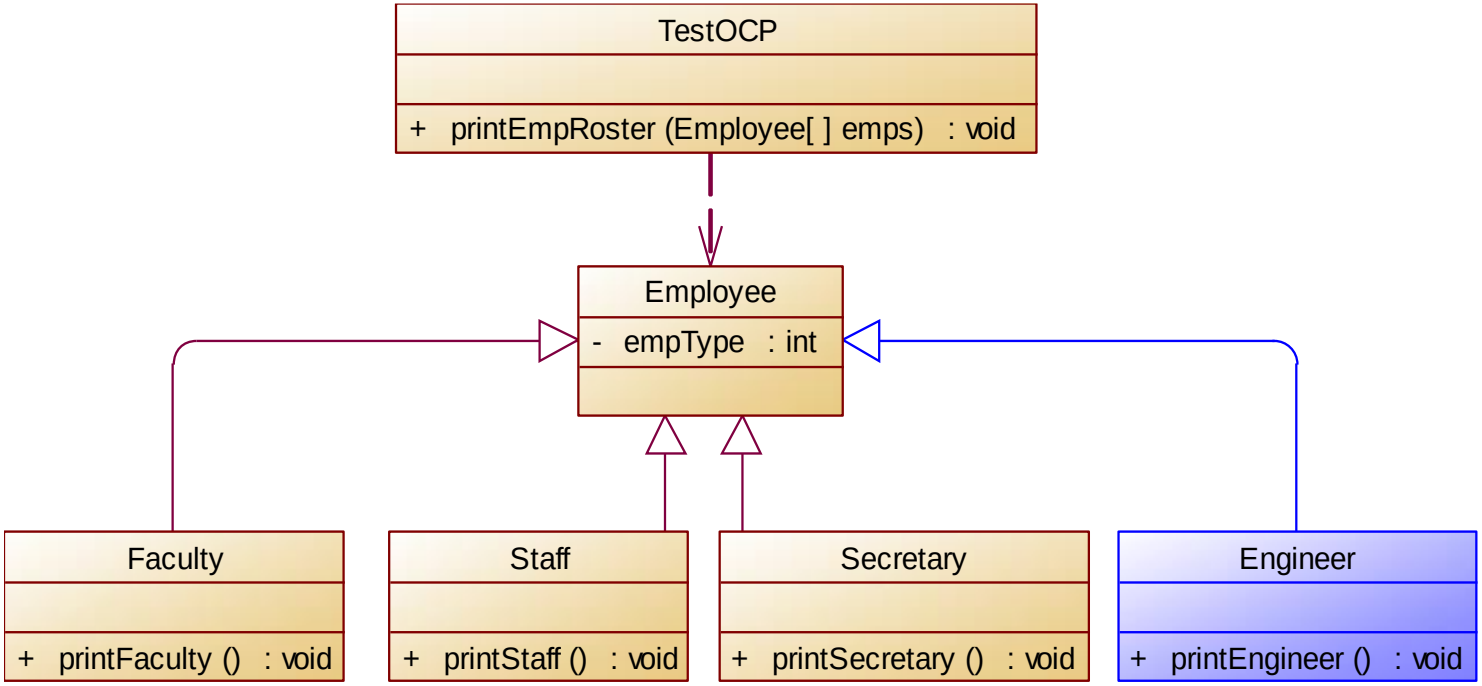
- Software entities (classes, modules, functions, etc.,) should be open for extension, but closed for modification

  软件实体（类、模块、方法等）应该对扩展开放，对修改关闭；也就是说在设计一个类/模块的时候，应当使这个类/模块可以在不被修改的前提下被扩展

- 理解开闭原则
  - 添加一个新的功能应该是在已有代码基础上扩展代码（新增模块、类、方法等），而不是修改已有代码（修改模块、类、方法等）
  - 并不是说完全杜绝修改，而是以最小的修改代价来完成新功能的开发
  - 试想一下插件式架构的系统…（plugin system）

开放/封闭原则 (OCP)

- 不符合OCP的设计：



- printEmpRoster(Employee[ ] emps) 打印大学职工信息
- 不同类型的职工具有不同的信息，各自实现打印方法，如：printFaculty( ), printStaff( ), printSecretary( )等

开放/封闭原则 (OCP)

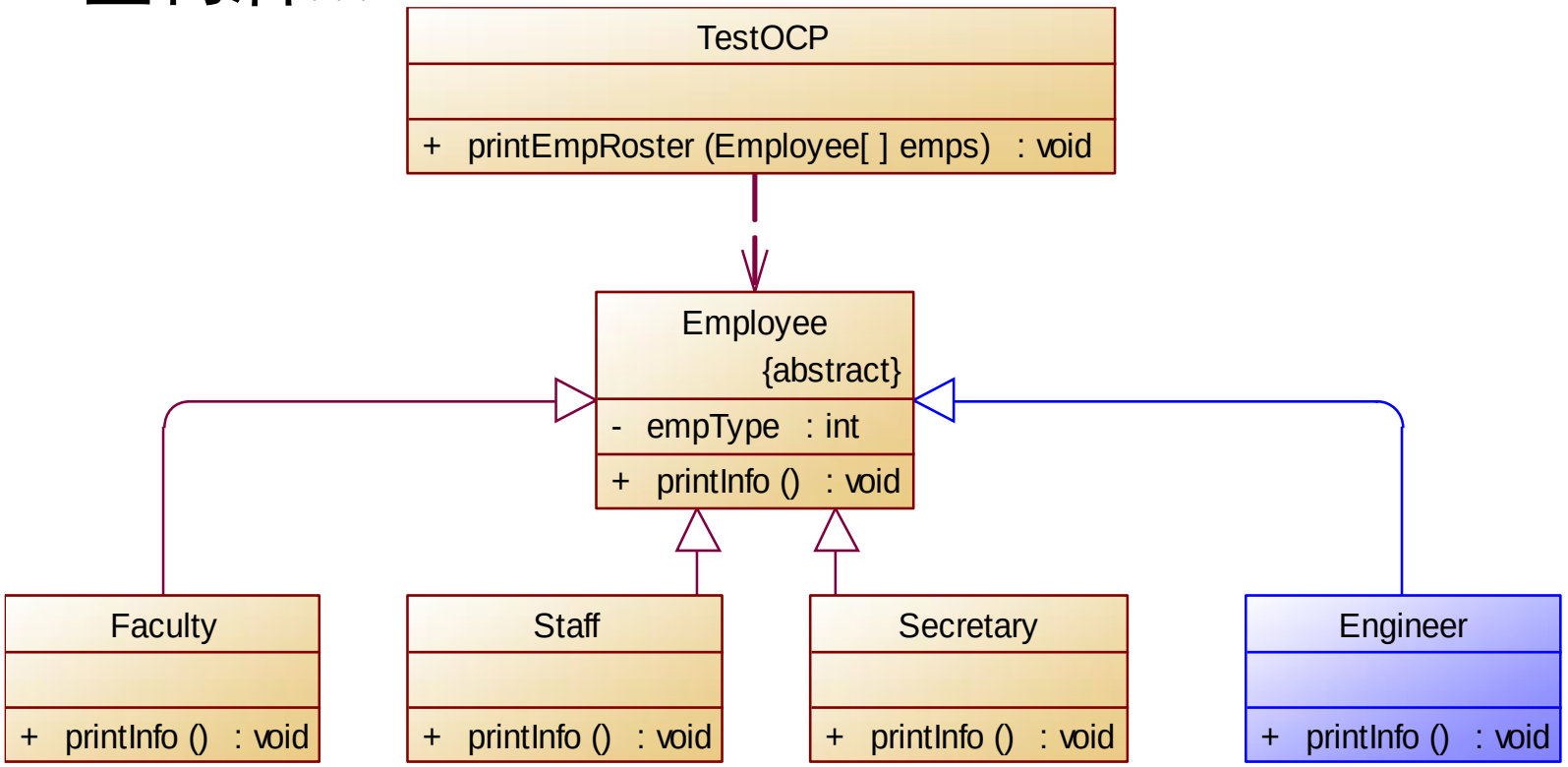
- 不符合OCP的设计：
  - 若要添加一个新的职工类别，如Engineer，则如何？

```
private static void printEmpRoster(Employee[] emps) {
    for (int i = 0; i < emps.length; i++) {
        if (emps[i].getEmpType() == FACULTY)
                ((Faculty) emps[i]).printFaculty();
        else if (emps[i].getEmpType() == STAFF)
                ((Staff) emps[i]).printStaff();
        else if (emps[i].getEmpType() == SECRETARY)
                ((Secretary) emps[i]).printSecretary();
    }
}
```

  - 客户程序需要进行更改，添加 else if 语句，重新编译；其他模块用到这个功能的也都需要更改；printEmpRoster()功能难于重用

开放/封闭原则 (OCP)

- 重构后…



- Employee类定义抽象方法printInfo()，由各子类实现

开放/封闭原则 (OCP)

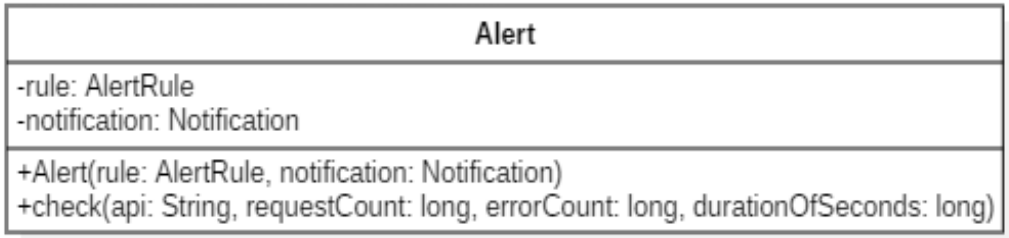
- 重构后…

```
private static void
printEmpRoster(Employee[] emps) {
    for (int i = 0; i < emps.length; i++) {
       emps[i].printInfo();
    }
}
```

  - 客户程序不需要进行更改，不需重新编译
  - Employee.printInfo( )对于扩展开放，客户程序printEmpRoster( )对于修改关闭

开放/封闭原则 (OCP)

- 不符合OCP的设计：
  - 举例：API 接口监控告警功能



    - AlertRule 存储告警规则，可以自由设置
    - Notification 是告警通知类，支持邮件、短信、微信、手机等多种通知渠道
    - NotificationEmergencyLevel 表示通知的紧急程度，包括 SEVERE、URGENCY、NORMAL、TRIVIAL，不同的紧急程度对应不同的发送渠道
    - 当接口的 TPS （每秒请求数，吞吐量指标之一）超过某个预先设置的最大值时，以及当接口请求出错数大于某个最大允许值时，就会触发告警，通知接口的相关负责人或者团队

开放/封闭原则 (OCP)

- 不符合OCP的设计：

```
public void check(String api, long requestCount, long errorCount,
long durationOfSeconds) {

  long tps = requestCount / durationOfSeconds;

  //每秒请求数告警
  if (tps > rule.getMatchedRule(api).getMaxTps()) {
    notification.notify(NotificationEmergencyLevel.URGENCY, "...");
  }

  //请求出错数告警
  if (errorCount > rule.getMatchedRule(api).getMaxErrorCount()) {
    notification.notify(NotificationEmergencyLevel.SEVERE, "...");
  }
}
```

开放/封闭原则 (OCP)

- 不符合OCP的设计：
  - 现在，如果我们需要添加一个功能，当每秒钟接口超时请求个数，超过某个预先设置的最大阈值时，我们也要触发告警发送通知

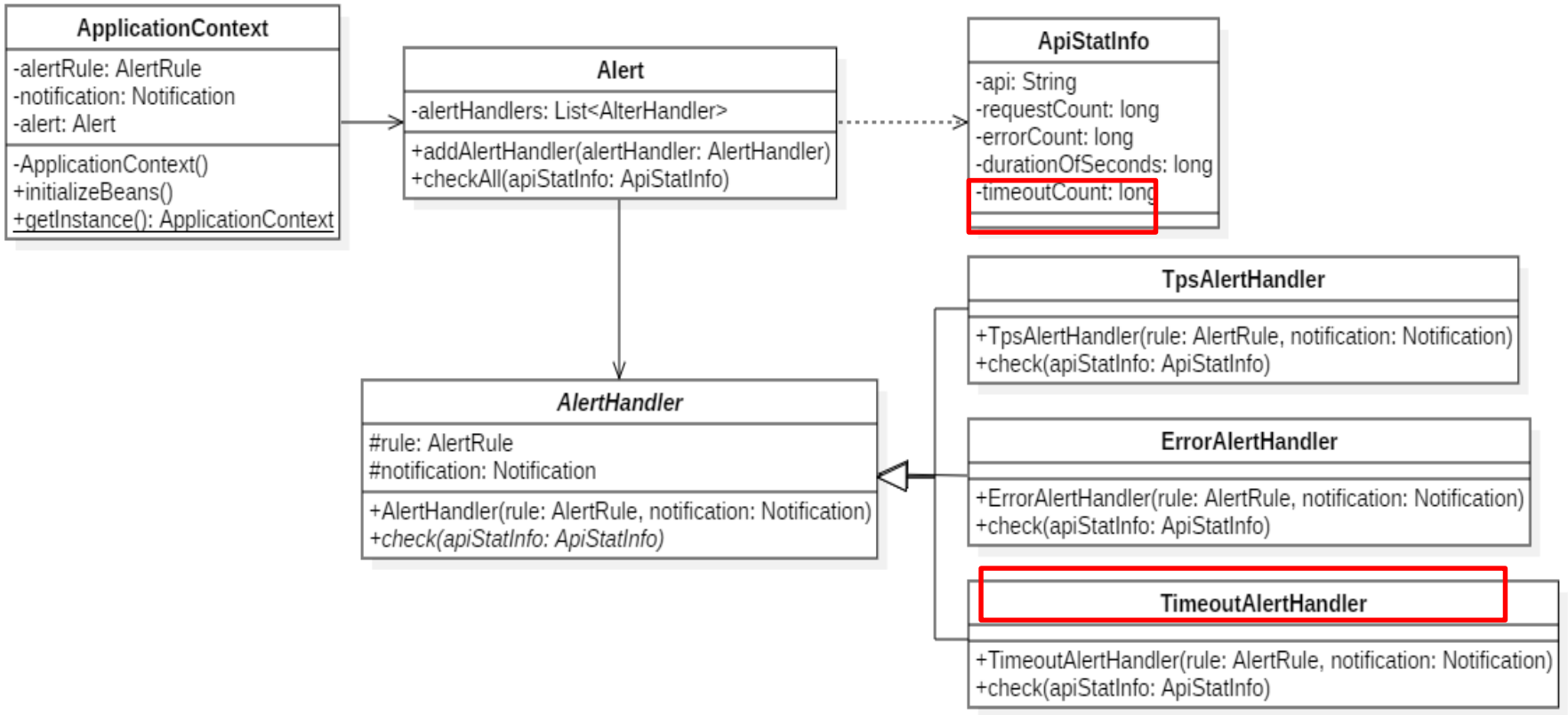
开放/封闭原则 (OCP)

- 不符合OCP的设计：

```
// 改动一：添加参数timeoutCount
public void check(String api, long requestCount, long errorCount,
long timeoutCount, long durationOfSeconds) {
  long tps = requestCount / durationOfSeconds;
  if (tps > rule.getMatchedRule(api).getMaxTps()) {
    notification.notify(NotificationEmergencyLevel.URGENCY, "...");
  }
  if (errorCount > rule.getMatchedRule(api).getMaxErrorCount()) {
    notification.notify(NotificationEmergencyLevel.SEVERE, "...");
  }
  // 改动二：添加接口超时处理逻辑
  long timeoutTps = timeoutCount / durationOfSeconds;
  //请求出错数告警
  if (timeoutTps > rule.getMatchedRule(api).getMaxTimeoutTps()) {
    notification.notify(NotificationEmergencyLevel.URGENCY, "...");
  }
}
```

开放/封闭原则 (OCP)

- 重构后…
  - 将 check() 函数的多个入参封装成 ApiStatInfo 类
  - 引入 handler 的概念，将 if 判断逻辑分散在各个 handler



开放/封闭原则 (OCP)

- In many ways, the OCP is at the heart of object-oriented design
  OCP是面向对象设计的核心原则

- OCP原则背后的机制是抽象(abstraction)和多态(polymorphism)，通过继承(inheritance)方式实现

- 遵循OCP原则可获得面向对象技术的最大好处
  - 可重用性 (reusability)
  - 可维护性 (maintainability)

开放/封闭原则 (OCP)

- 如何做到“对扩展开放、修改关闭”？
  - 时刻保持扩展意识、抽象意识、封装意识
  - 花点时间往前多思考一下，未来可能有哪些需求变更，事先留好扩展点，当需求变更的时候，不需要改动整体结构
  - 在识别出代码可变部分和不可变部分之后，将可变部分封装起来，隔离变化，提供抽象化的不可变接口，给上层系统使用
  - 提高扩展性的方法有：多态、依赖注入、基于接口而非实现编程

Barbara Liskov, 1939 -

美国计算机科学家，2008年图灵奖得主，2004年约翰.冯诺依曼奖得主，美国工程院院士，美国艺术与科学院院士，美国计算机协会会士（ACM Fellow）。现任麻省理工学院电子电气与计算机科学系教授。她是美国首批计算机科学女博士之一。

见参考书P111-125（英）、P102-115（中）

里氏替换原则 (LSP)

Liskov Substitution Principle

里氏替换原则 (LSP)

- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it

  所有引用基类的地方必须能透明地使用其子类的对象

  - Barbara Liskov的原话：
    - What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T

      如果对每一个类型为S的对象 o_1，都有一个类型为T的对象 o_2，使得以T定义的所有程序P，当用对象o_1替换o_2时，程序P的行为不会发生变化，那么类型S是类型T的子类型
  - 即：Subtypes must be substitutable for their base types

    可以用子类型替换基类型

里氏替换原则 (LSP)

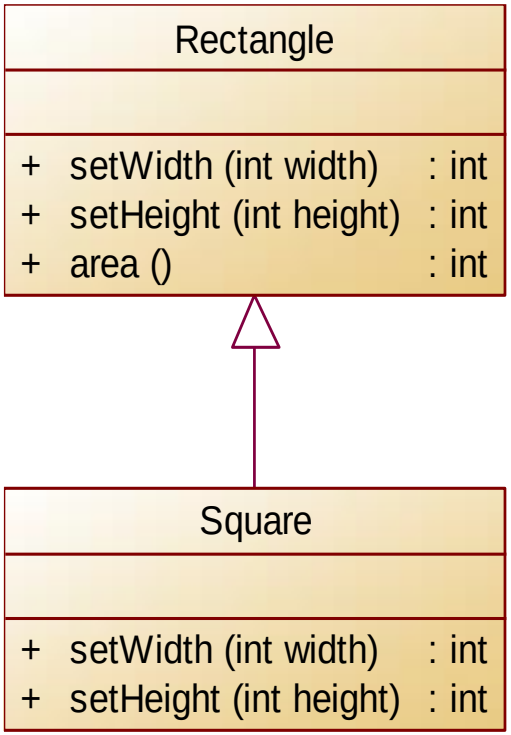
- 理解里氏替换原则
  - 是用来指导继承关系中子类该如何设计的一个原则
  - 可描述为“Design by Contract”（按照契约来设计）
    - 父类定义了函数的“约定”，子类可以改变函数的内部实现逻辑，但不能改变函数原有的“约定”。这里的“约定”包括：函数声明要实现的功能；对输入、输出、异常的约定；注释中所罗列的任何特殊说明
  - 里式替换和多态有点类似，但它们关注的角度是不一样的
    - 多态是面向对象编程的特性，是一种代码实现的思路
    - 里式替换是一种设计原则

里氏替换原则 (LSP)

- 不符合LSP原则的设计：

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

  public void setHeight(int height) {
       super.setWidth(height);
       super.setHeight(height);
  }
} // 破坏了Rectangle的width-height独立性
```



  - 子类Square重写了父类的方法
    - 看上去好像也没什么问题啊…
    - 但是看看客户代码就会有问题了

里氏替换原则 (LSP)

- 不符合LSP原则的设计：
  - 客户代码

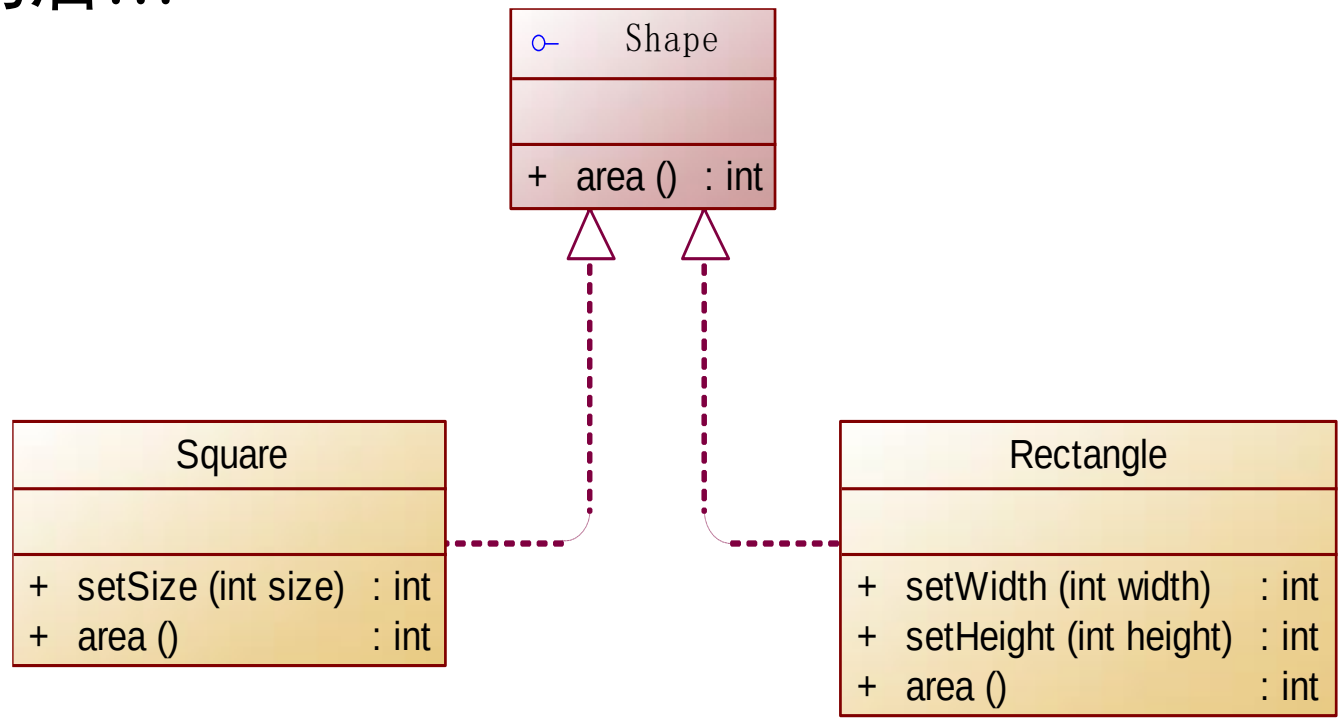
```
public class TestLSP {
  public static void main(String[] args) {
        Rectangle rec = new Square();
        clientOfRectangle(rec); // rec是子类Square的对象
  }

  private static void clientOfRectangle(Rectangle rec) {
        rec.setWidth(4);
        rec.setHeight(5);
        System.out.println(rec.area());
  }
}
```

  - 在clientOfRectangle( )中用Square对象替换Rectangle对象后出问题了！

里氏替换原则 (LSP)

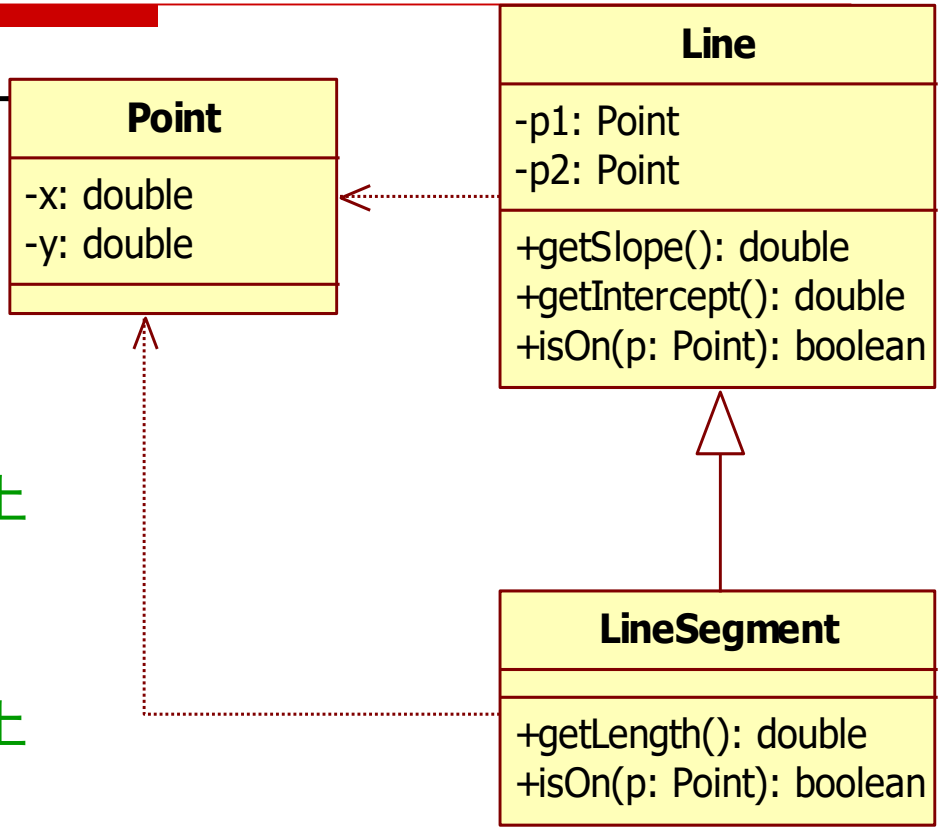
- 重构后…



- 符合LSP原则
- 当用Square对象或Rectangle对象替换Shape对象时，客户程序也不会出现问题

里氏替换原则 (LSP)

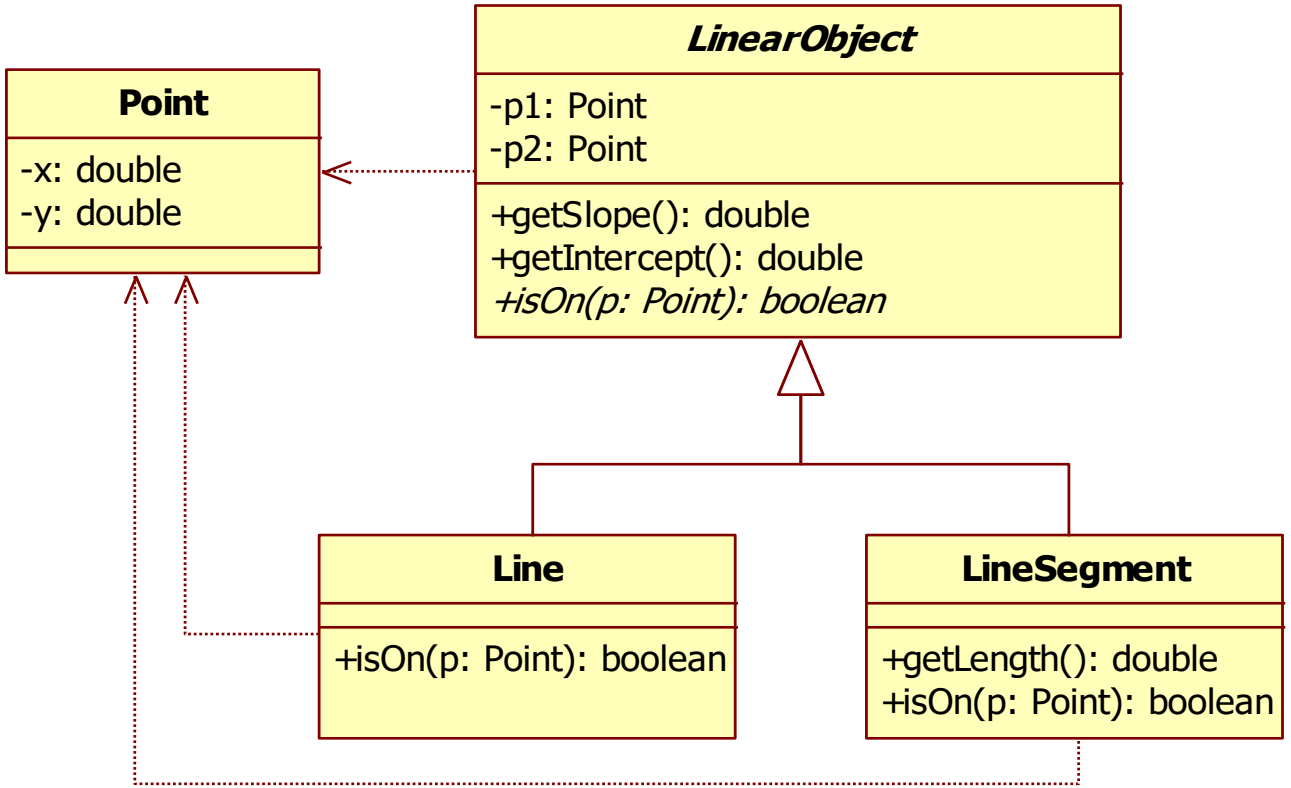
- 不符合LSP原则的设计
  - Point：坐标点
  - Line：直线
    - getSlope()：斜率
    - getIntercept()：截距
    - isOn()：某点是否在线上
  - LineSegment：线段
    - getLength()：长度
    - isOn()：某点是否在线上



  - 注意：isOn( )在子类LineSegment中被重写了，以判断某点是否在线段上
    - 试想一下：若在使用Line对象的客户程序中，用LineSegment对象替换，isOn()的结果总是一样吗？

里氏替换原则 (LSP)

- 重构后…



- 引入抽象基类LinearObject，其派生Line和LineSegment，从而符合LSP
- 当需要扩展时也很方便，如加入子类“射线Ray”

里氏替换原则 (LSP)

- 当使用继承时，要遵循LSP原则
  - 类B继承类A时，除添加新的方法完成新增功能外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法
  - 父类中已经实现好的方法，实际上是在设定一系列的契约(Contract)，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏
  - 继承在给程序设计带来巨大便利的同时，也带来了弊端
    - 比如使用继承增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障

里氏替换原则 (LSP)

- 违反LSP原则的一些线索
  - 如果你发现在子类中把父类的某些功能去掉了，即子类能做的事情比父类还少，说明有违反LSP原则的可能，需仔细进行分析

```
public class Base
{
  public void f() {/*some code*/}
}

public class Derived extends Base
{
  public void f() {}
}
```

  - 如果子类的方法中增加了异常的抛出，也可能会导致子类对象变得不可替换（因为调用者并没有处理异常的代码），从而违反LSP原则

里氏替换原则 (LSP)

- 避免违反LSP的一些策略
  - Only strengthen invariants in subclasses; never weaken them
    在子类中只可增强不变要素，而不可减弱
  - Only weaken preconditions when overriding methods
    当重写方法时只减弱前置条件
  - Only strengthen postconditions when overriding methods
    当重写方法时只增强后置条件
  - Use Delegation instead of Inheritance
    使用委派（聚合/组合）代替继承
  - Figure out better abstractions
    设计更好的抽象方案

见参考书P135-145（英）、P123-132（中）

接口隔离原则 (ISP)
Interface Segregation Principle

接口隔离原则 (ISP)

- Clients should not be forced to depend upon interfaces that they do not use.

  客户程序不应该被强迫依赖它不使用的接口

  - 这里说的接口可指：一组API接口集合；单个API接口或函数；OOP中的接口概念

- When we bundle functions for different clients into one interface/class, we create unnecessary coupling among the clients.

  将不同客户所使用的函数捆绑在一个接口/类中，相当于在这些客户间增加了不必要的耦合

接口隔离原则 (ISP)

- Once an interface has gotten too 'fat' it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them.

  一旦一个接口太大，则需要将它分割成一些更细小的接口，这样使用该接口的客户程序仅需知道与之相关的方法即可

接口隔离原则 (ISP)

- 理解ISP原则

  - 使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色
    - 一个接口就只代表一个角色，每个角色都有它特定的一个接口
    - 接口仅仅提供客户程序需要的行为，即所需的方法，客户程序不需要的行为则隐藏起来，应当为客户程序提供尽可能小的单独的接口，而不要提供大的总接口

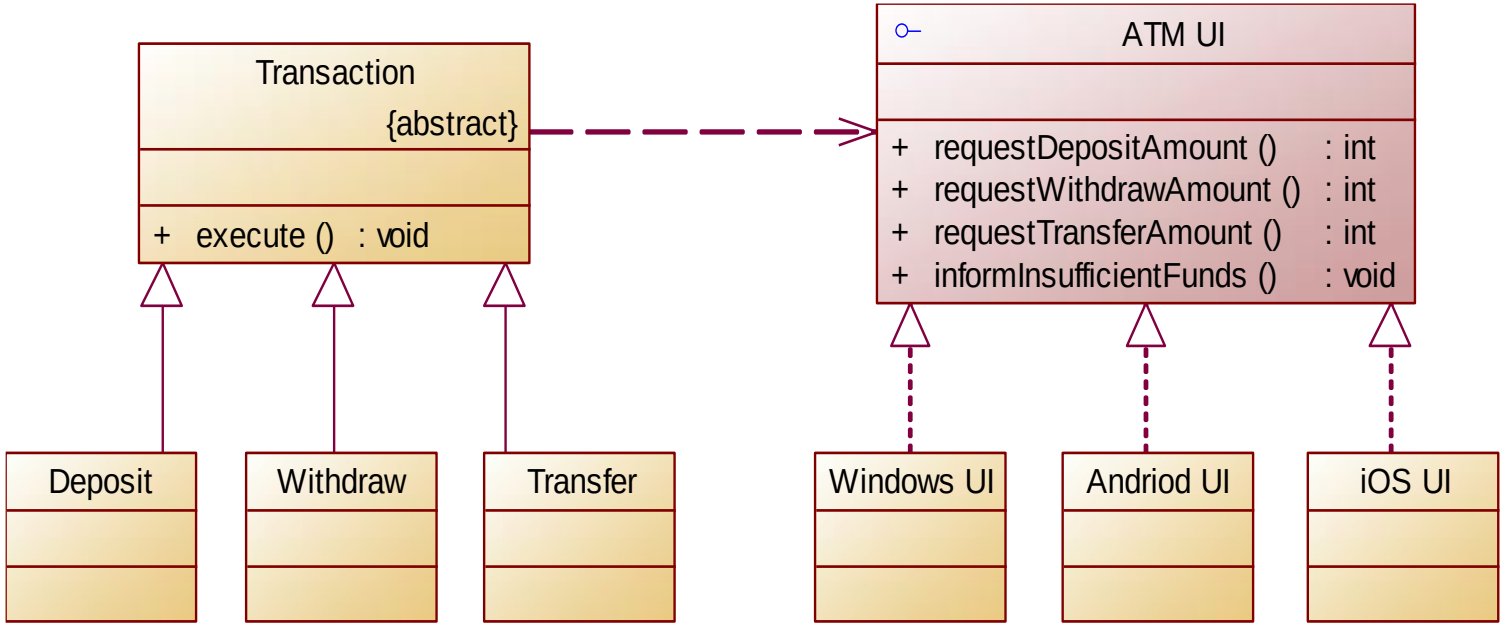
  - 拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好

接口隔离原则 (ISP)

- （1）把“接口”理解为一组 API 接口集合
  - 可以是某个微服务的接口，也可以是某个类库的接口
  - 例子：设计ATM
  - ATM要实现的功能：
    - 支持的交易类型：取款（Withdraw）、存款（Deposit）、转账（transfer）
    - 支持不同类型的用户界面（UI）
    - 每种交易类型需要调用 UI 的方法，如请求存款、取款或转账的金额

接口隔离原则 (ISP)

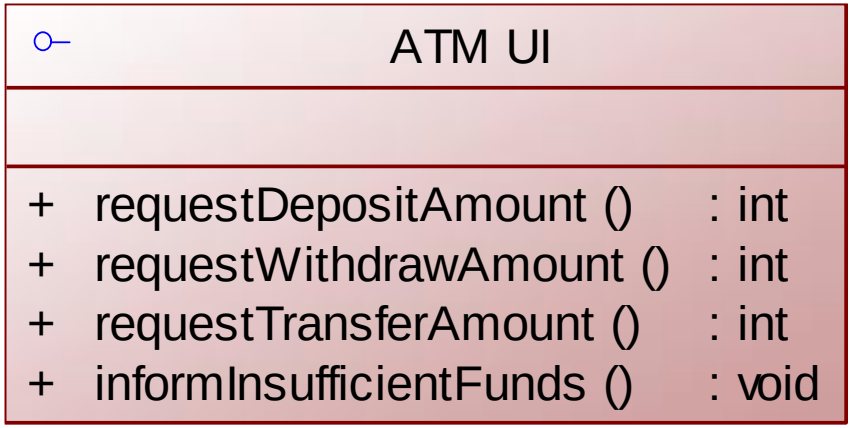
- 不符合ISP原则的设计：



- 每种交易类使用了接口 ATM UI 中的一部分方法，但却依赖其所有方法，故 ATM UI 的任何改变均会影响所有的交易类型

接口隔离原则 (ISP)

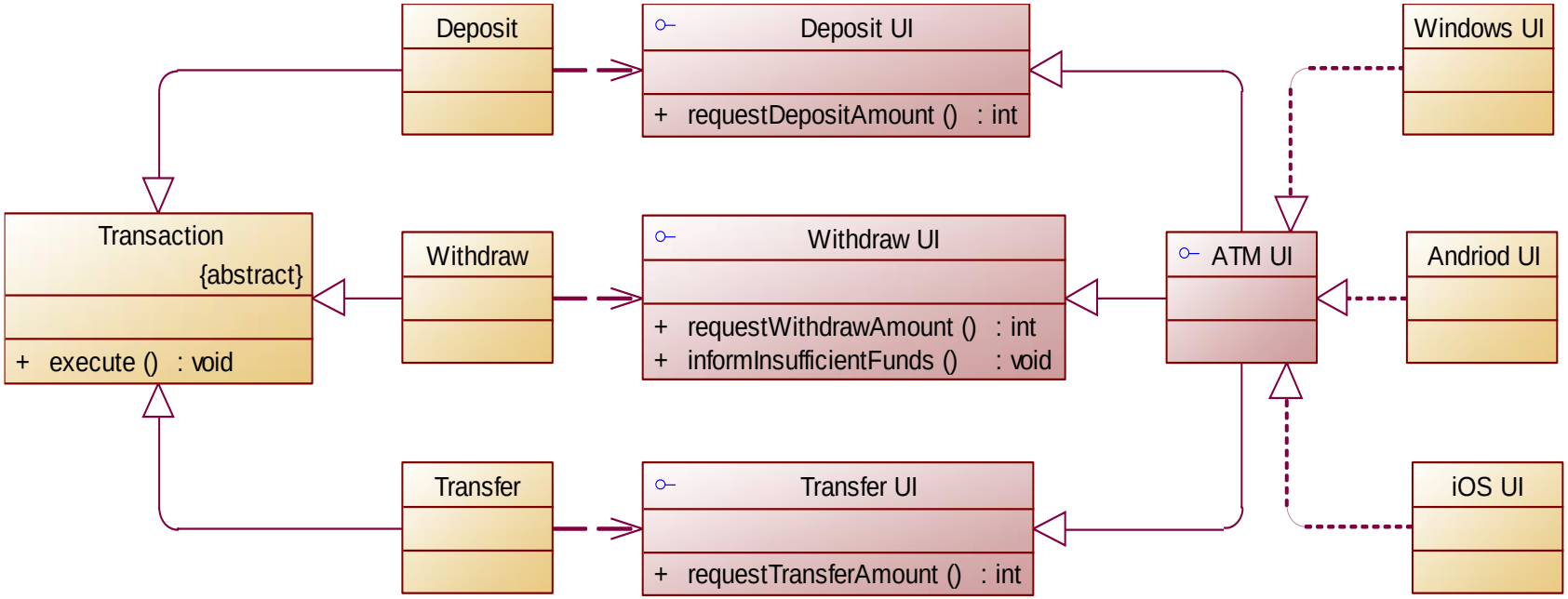
- ATM UI是一个被污染(polluted)了的接口



- 它声明的方法并不是属于一起实现某个功能的
- 它使类依赖于不需用到的方法，因此受与该类无关的修改所影响
  - 例如：如果要加一个“支付电费”的Transaction子类，则不得不又在UI接口中添加只为它所使用的方法，因其他Transaction子类也依赖ATM UI接口，故它们也必须重新编译
- ISP要求这样的接口应该分隔

接口隔离原则 (ISP)

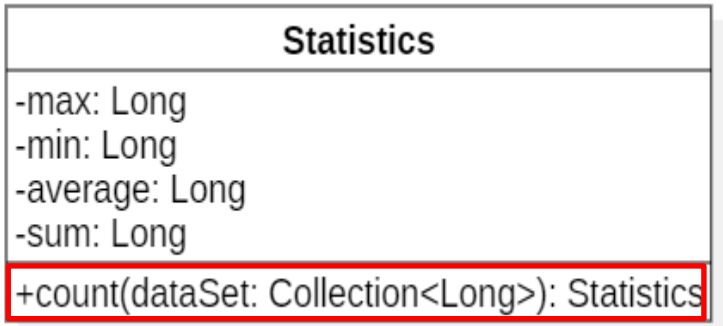
- 重构后…



- 将原 ATM UI 拆分为3个接口为：Deposit UI, Withdraw UI, Transfer UI。让Deposit、Withdraw、Transfer 依赖各自相应的接口

接口隔离原则 (ISP)

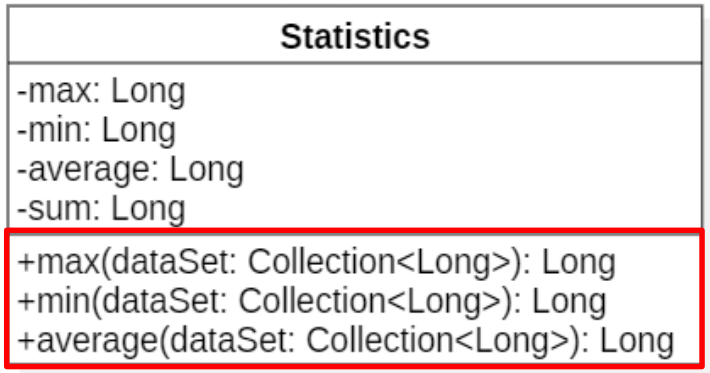
- （2）把“接口”理解为单个 API 接口或函数
  - 函数的设计要功能单一，不要将多个不同的功能逻辑在一个函数中实现
  - 不符合ISP原则的设计：



  - count( ) 函数的功能不够单一，包含很多不同的统计功能，比如，求最大值、最小值、平均值等等
    - 注意：如果对于每个统计需求，Statistics 定义的那几个统计信息都有涉及，那么count() 函数的设计就是合理的，符合ISP原则；如果有的只需要用到其中的一部分，则不符合ISP原则

接口隔离原则 (ISP)

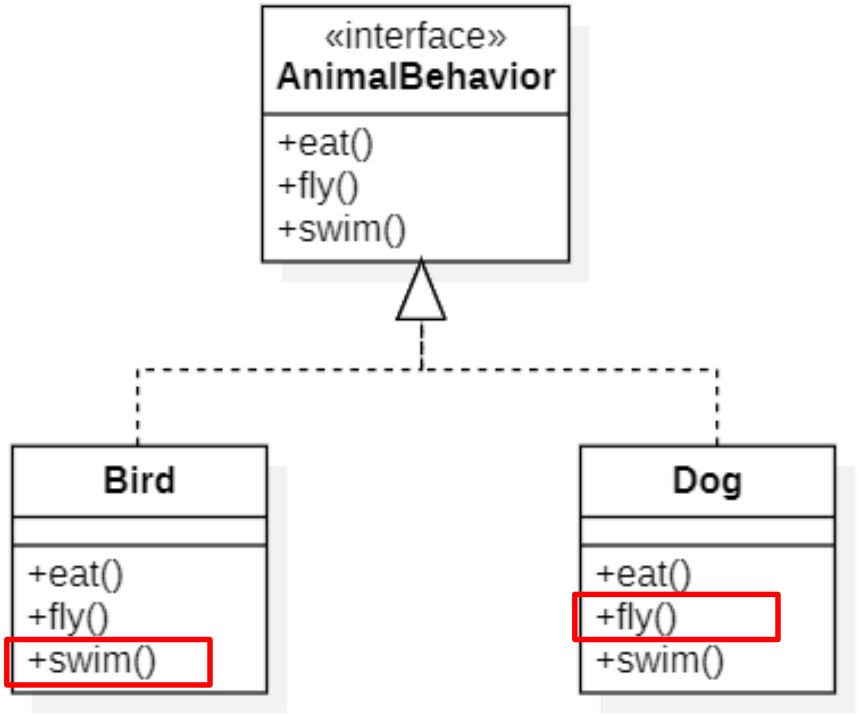
- 重构后…



- 把 count( ) 函数拆成几个更小粒度的函数，每个函数负责一个独立的统计功能
- 判定功能是否单一，除了很强的主观性，还需要结合具体的场景
- 接口隔离原则跟单一职责原则有点类似，不过稍微还是有点区别
  - 单一职责原则针对的是模块、类、接口的设计
  - 接口隔离原则一方面它更侧重于接口的设计，另一方面它思考的角度不同。它提供了一种判断接口是否职责单一的标准：通过调用者如何使用接口来间接地判定

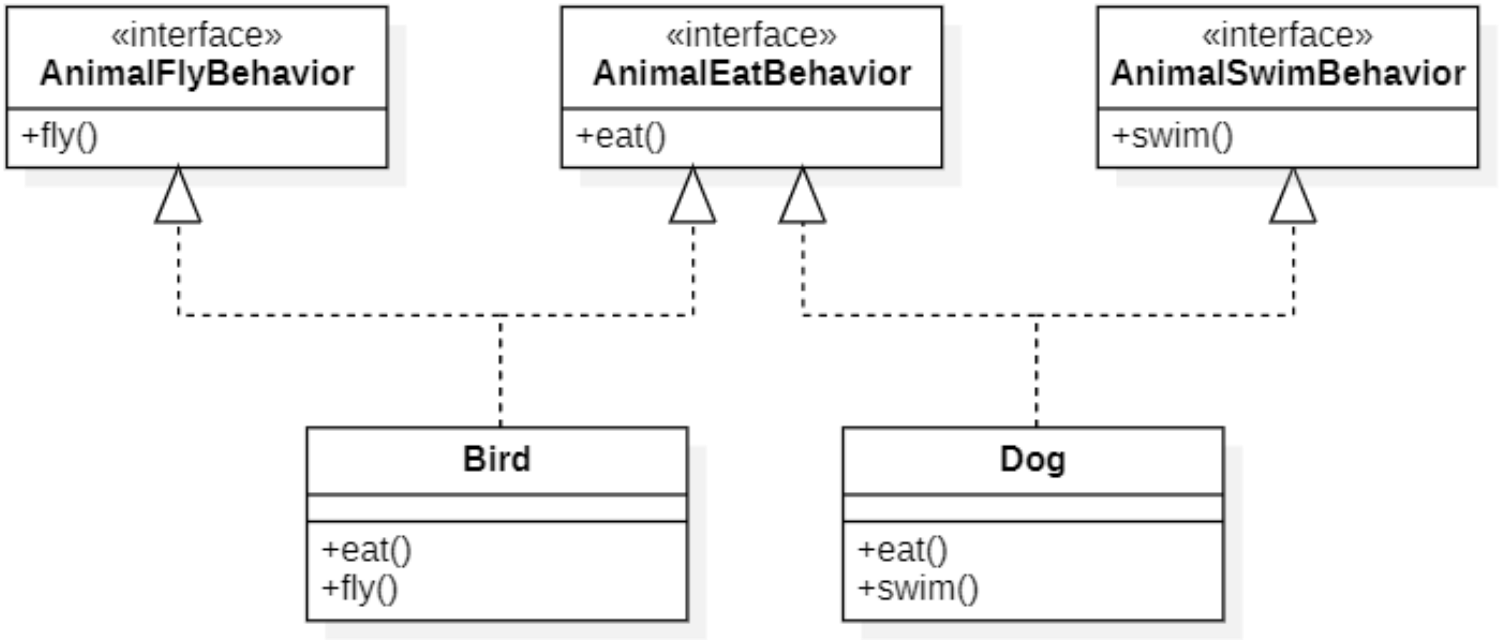
接口隔离原则 (ISP)

- 不符合ISP原则的设计：
  - 接口包含了较多的功能，会导致一些不必要的依赖
  - 如：Bird并不需要实现swim()；Dog并不需要实现fly()



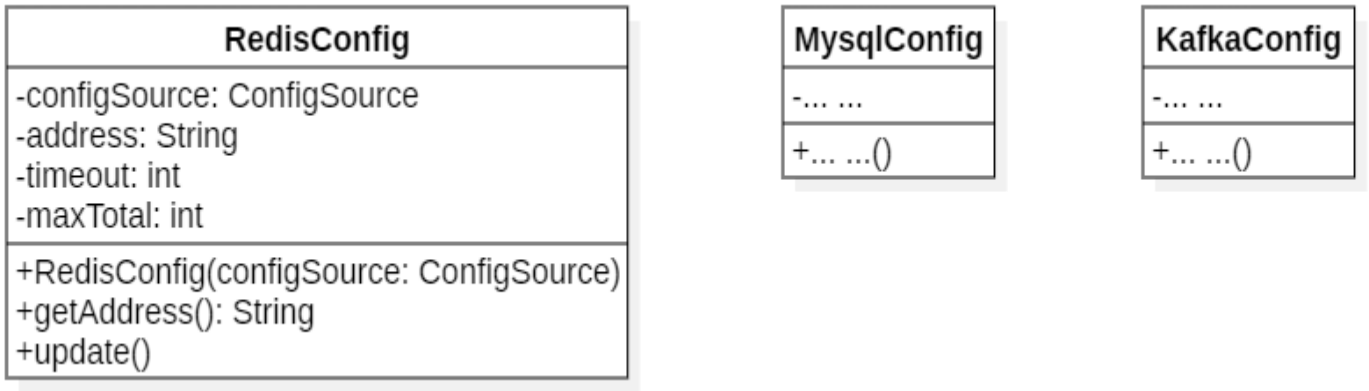
接口隔离原则 (ISP)

- 重构后…
  - 注：接口的粒度大小要取决于具体业务，也不是越小越好



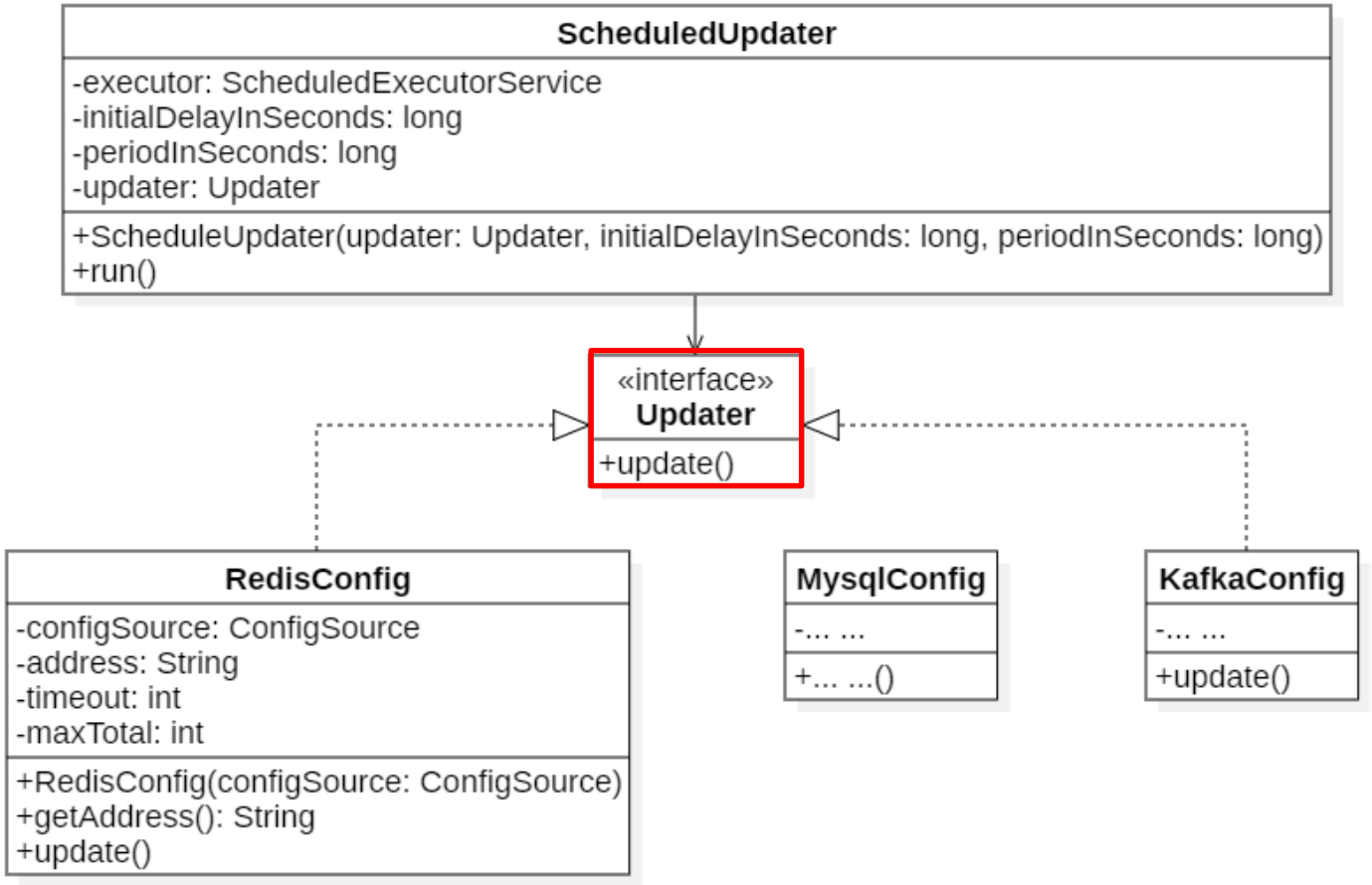
接口隔离原则 (ISP)

- （3）把“接口”理解为 OOP 中的接口概念
  - 接口的设计要尽量单一，不要让接口的实现类和调用者，依赖不需要的接口函数
  - 举例：假设项目中用到了三个外部系统：Redis、MySQL、Kafka。每个系统都对应一系列配置信息，比如地址、端口、访问超时时间等。这些配置信息需在内存中存储，供项目中的其他模块使用。



接口隔离原则 (ISP)

- 现在，有一个新的功能需求，希望支持 Redis 和 Kafka 配置信息的“热更新”（Hot Update），下面的设计符合ISP原则：



接口隔离原则 (ISP)

- ISP的策略：
  - 建立单一的接口，不要建立庞大臃肿的接口，接口中的方法尽量少
    - 为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用
  - 接口的大小要适度，设计得过大或过小都不好
- 遵循ISP的好处：
  - 通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性

见参考书P127-134（英）、P116-122（中）

依赖倒置原则 (DIP)

【也称依赖反转原则】

Dependency Inversion Principle

依赖倒置原则 (DIP)

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
  高层模块不应该依赖低层模块，二者都应该依赖其抽象
  或者说：
- Abstractions should not depend upon details. Details should depend upon abstractions.
  抽象不应该依赖具体实现细节；具体实现细节应该依赖抽象

  另一种表述：
- Program to an interface, not an implementation
  要针对接口编程，不要针对实现编程

依赖倒置原则 (DIP)

- Traditional structural programming creates a dependency structure in which policies depend on details.

  传统的结构化编程的依赖结构：Policy依赖Detail

  - Policies become vulnerable to changes in the details.

- Object-orientation enables to invert the dependency.

  面向对象编程将依赖反转了：Policy和Detail都依赖Abstraction

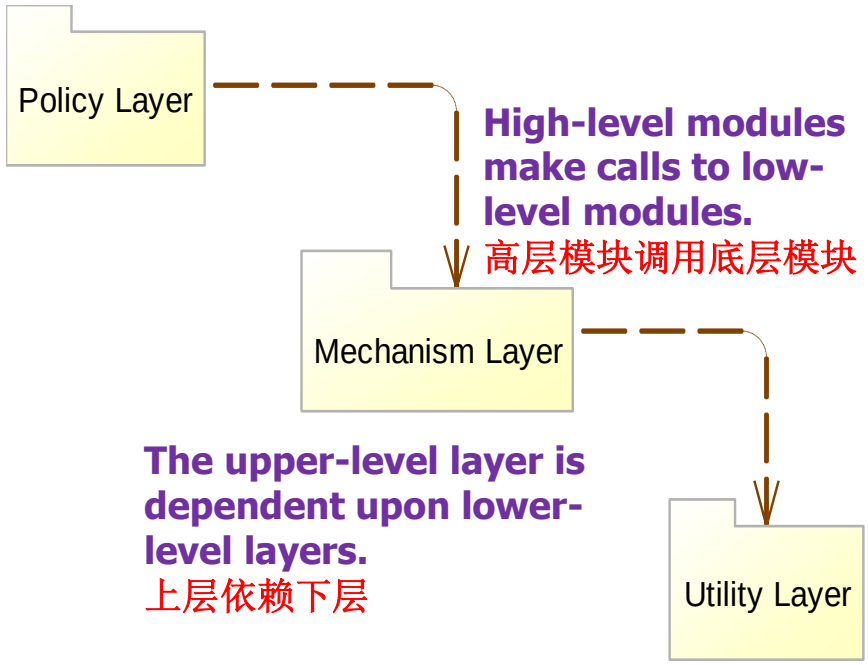
  - Policy and details depend on abstractions.
  - Service interfaces are owned by their clients.
  - Inversion of dependency is the hallmark of good object-oriented design.

依赖倒置原则 (DIP)

- 分层【Layering】
  - "…all well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface…" ---- Grady Booch

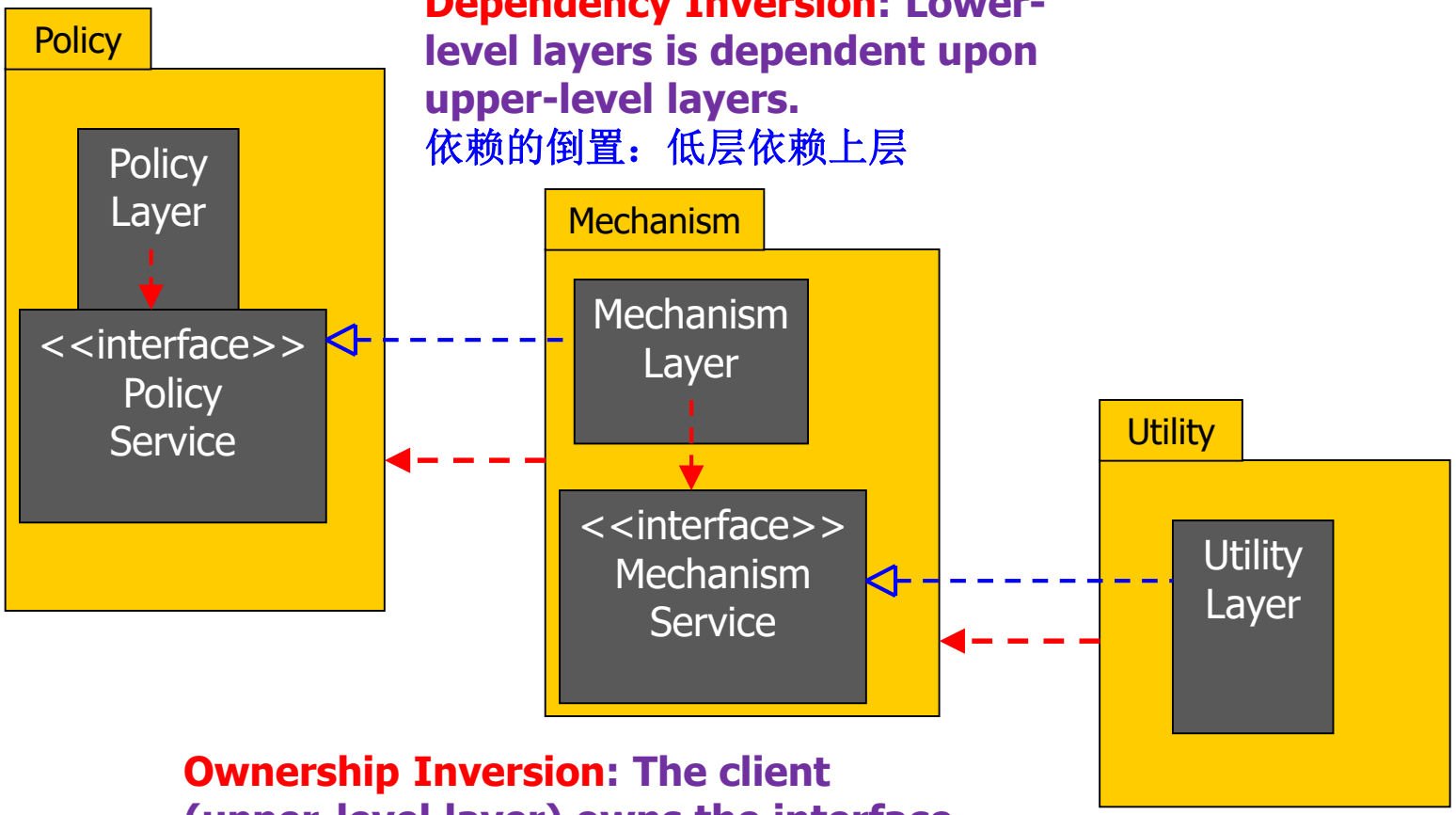
The higher the module is positioned in a layered architecture, the more general the function it implements.
越高层的模块，它实现的功能就越抽象越宽泛

The lower the module, the more detailed the function it implements.
越底层的模块，它实现的功能就越具体越细节



依赖倒置原则 (DIP)

- 分层【Layering】（更合适的模型）



Dependency Inversion: Lower-level layers is dependent upon upper-level layers.
依赖的倒置：低层依赖上层

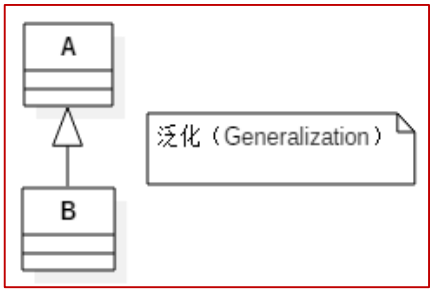
Ownership Inversion: The client (upper-level layer) owns the interface, not the lower-level layers
所有权的倒置：上层拥有接口

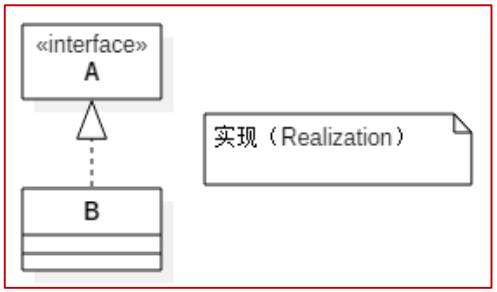
依赖倒置原则 (DIP)

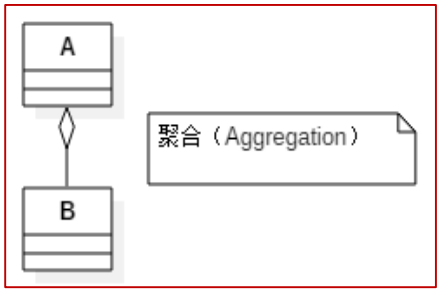
- 遵循DIP原则的启发式做法（Heuristic）：
  - Depend on abstractions
    依赖抽象
  - 即：Do not depend on a concrete class – that all relationships in a program should terminate on an abstract class or an interface
    - No variable should hold a pointer or reference to a concrete class.
      没有任何变量通过指针或引用指向一个具体类
    - No class should derive from a concrete class.
      没有任何类派生自一个具体类
    - No method should override an implemented method of any of its base classes.
      没有任何方法重写在基类中已实现的方法

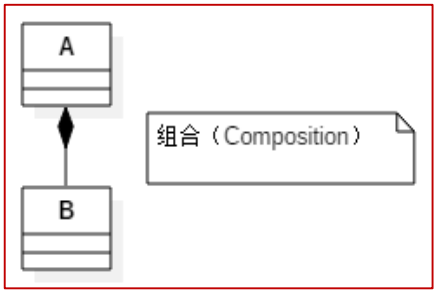
依赖倒置原则 (DIP)

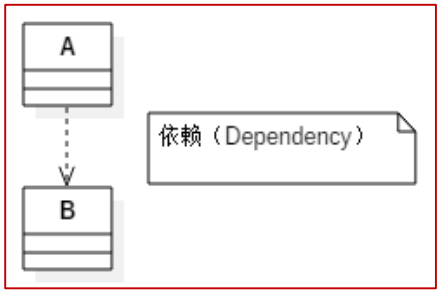
- 什么是依赖？
  - 重温类之间的关系
    - 泛化、实现、聚合、组合、关联、依赖
      - 关联关系包含：聚合、组合
      - 依赖关系包括：聚合、组合、使用（参数、返回值、局部变量）
    - 各关系的强弱顺序：泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖











依赖倒置原则 (DIP)

- 类间关系的代码实现（Java语言为例）

```
/* 泛化（Generalization）*/
public class A { ... }
public class B extends A { ... }
```

```
/* 实现（Realization）*/
public interface A {...}
public class B implements A { ... }
```

```
/* 聚合（Aggregation）*/
public class A {
  private B b;

  public A(B b) {
    this.b = b;
  }
}
```

```
/* 组合（Composition）*/
public class A {
  private B b;

  public A() {
    this.b = new B();
  }
}
```

```
/* 关联（Association）*/
public class A {
  private B b;

  public A(B b) {
    this.b = b;
  }
}
// 或者
public class A {
  private B b;

  public A() {
    this.b = new B();
  }
}
```

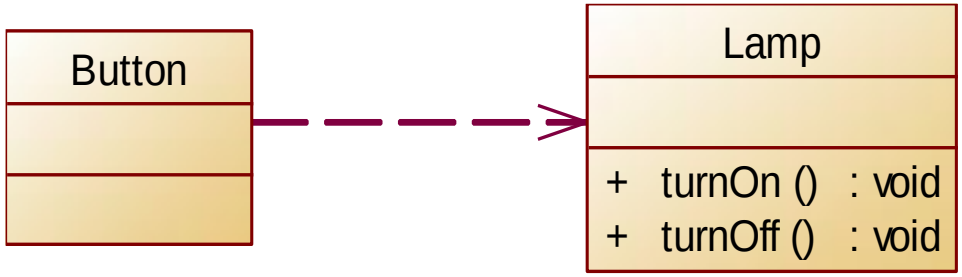
```
/* 依赖（Dependency）*/
public class A {
  private B b;

  public A(B b) {
    this.b = b;
  }
}
// 或者
public class A {
  private B b;

  public A() {
    this.b = new B();
  }
}
// 或者
public class A {
  public void func(B b) { ... }
}
```

依赖倒置原则 (DIP)

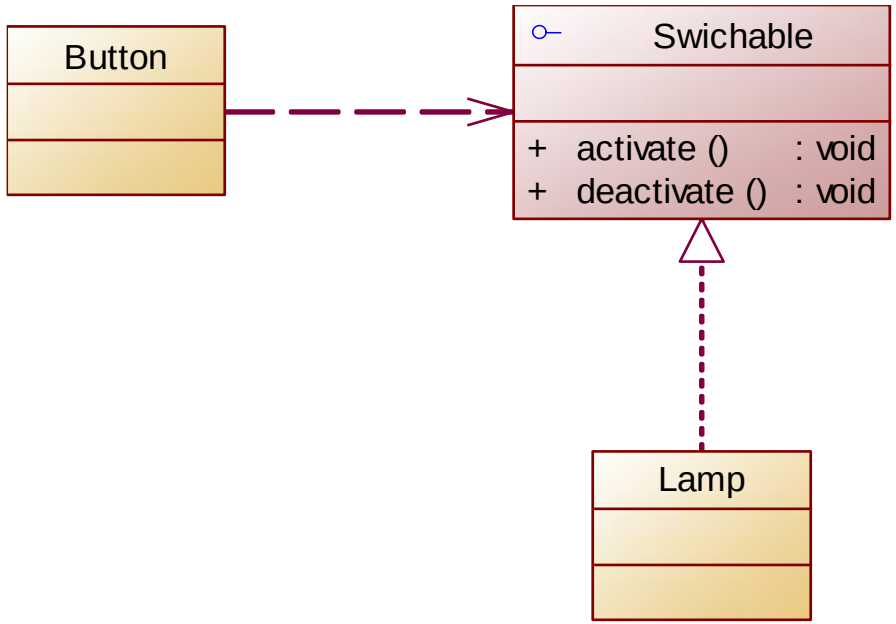
- 不符合DIP原则的设计：



- Button知道自己当前的状态是activated还是deactivated，当用户按它时，它切换到另一个状态
- 在此设计中，Button对象控制且只控制Lamp对象
- 问题：如果将来我要让Button去控制一个Fan，而不是Lamp，咋办？

依赖倒置原则 (DIP)

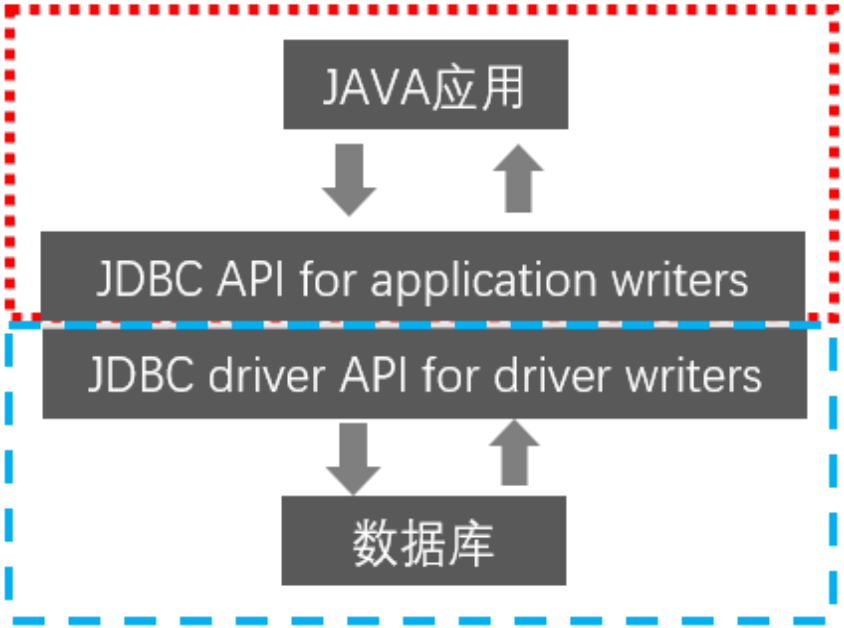
- 重构后…



- 接口Switchable提供抽象方法，Lamp实现Switchable接口，Lamp依赖Switchable而不是被依赖
- 若要Button控制Fan时，不用修改Button类，只需让Fan类实现Switchable接口即可，不会带来风险

依赖倒置原则 (DIP)

- 例如：JDBC架构的设计符合依赖倒置原则
  - JDBC主要包括两类接口：
    - 提供了一套 Java API 给应用程序开发者，用于开发访问数据库的程序
    - 提供了一套 JDBC Driver API 给数据库驱动开发者，用于提供服务



依赖倒置原则 (DIP)

- 传递依赖关系的方式（即：依赖注入，DI，Dependency Injection）

  （即：不通过 new() 的方式在类内部创建依赖类对象）

  - 构造方法传递
  - setter方法传递

  - 依赖注入的对象可由“依赖注入框架”来自动创建和管理，如：Google Guice、Java Spring、Pico Container 等

```
/* 通过构造方法注入的例子 */
public class Notification {
  private MessageSender messageSender;

  public Notification(MessageSender messageSender) {
    this.messageSender = messageSender;
  }

  public void sendMessage(String cellphone, String message) {
    this.messageSender.send(cellphone, message);
  }
}

public interface MessageSender {
  void send(String cellphone, String message);
}
```

依赖倒置原则 (DIP)

- DIP原则背后的原理：
  - Good software designs are structured into modules
    好的软件设计是模块化的
    - High-level modules contain the important policy decisions and business models of an application – The identity of the application.
      高层模块包含一个应用的重要规范和业务模型 — 等同于该应用
    - Low-level modules contain detailed implementations of individual mechanisms needed to realize the policy.
      低层模块包含为实现这些规范所需要的个体机制的具体实现

依赖倒置原则 (DIP)

- High-level Policy
  高层的规范（可理解为业务逻辑Business Logic）

  - The abstraction that underlies the application;
  - The truth that does not vary when details are changed;
  - The system inside the system;
  - The metaphor.

- 相对于实现细节的多变性，抽象的东西要稳定得多

组合复用原则 (CRP)
Composite Reuse Principle

组合复用原则 (CRP)

- Favor composition of objects over inheritance as a reuse mechanism

  尽量使用对象组合，而不是继承来达到复用的目的

  - 组合复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的成员；新对象通过委派调用已有对象的方法达到复用其已有功能的目的
  - 简言之：要尽量使用组合/聚合关系，少用继承

- 也称为：组合/聚合复用原则

  Composition / Aggregate Reuse Principle, CARP

组合复用原则 (CRP)

- 理解CRP原则

  - 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承

    - 继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用 ）

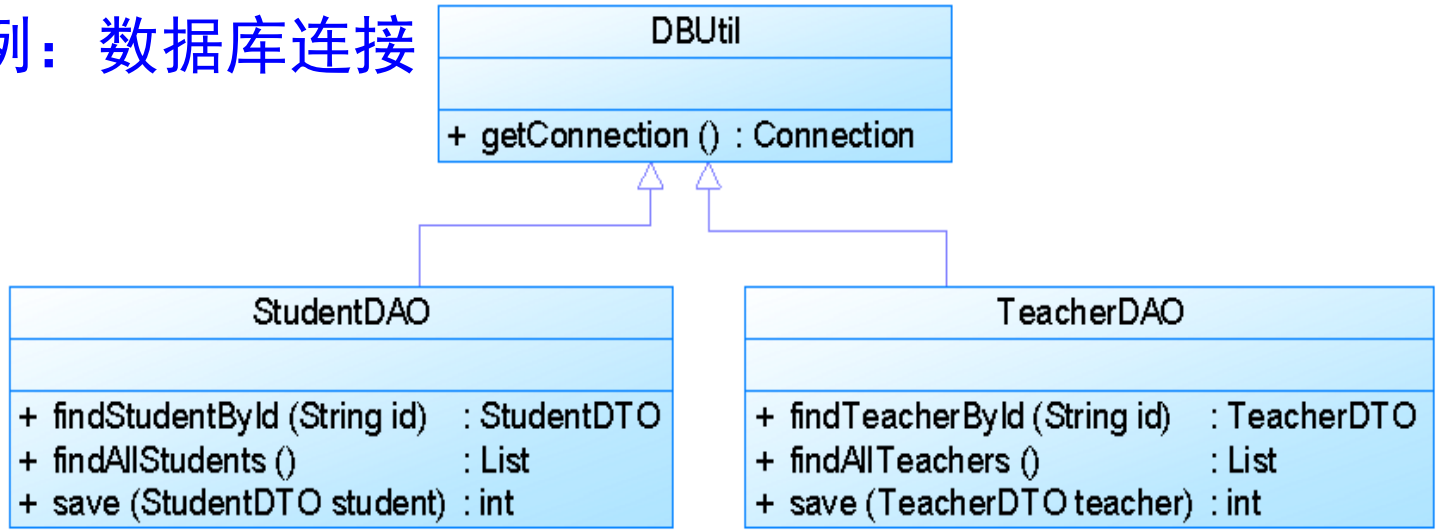
    - 组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用 ）

组合复用原则 (CRP)

- 理解CRP原则
  - 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低
    - 一个类的变化对其他类造成的影响相对较少
  - 一般首选使用组合/聚合来实现复用，其次才考虑继承；“has a”关系使用组合，“is a”关系使用继承
  - 在使用继承时，要严格遵循里氏替换原则
  - 要慎重使用继承复用
    - 有效使用继承有助于对问题的理解，降低复杂度；而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度

组合复用原则 (CRP)

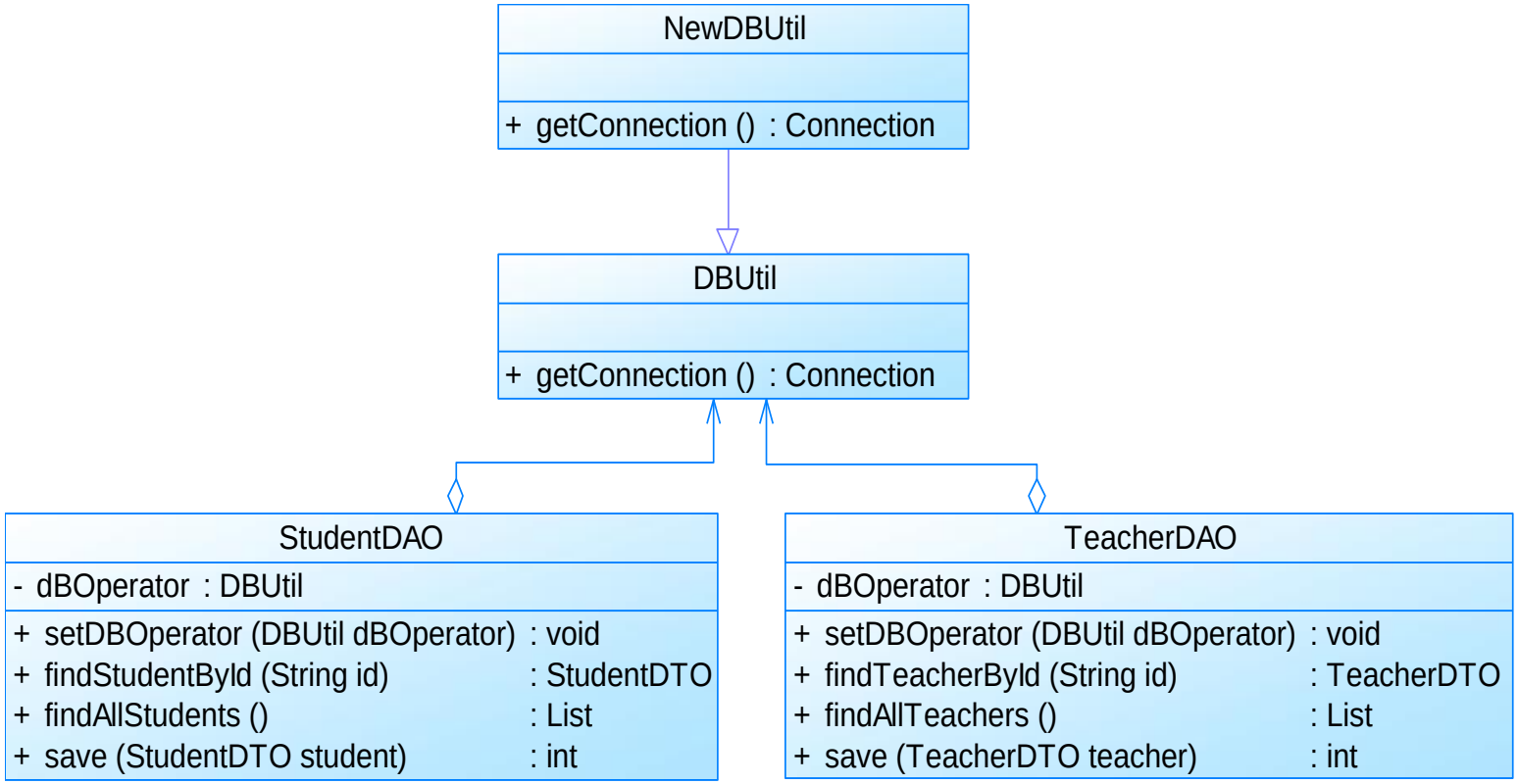
- 不符合CRP原则的设计：
  - 例：数据库连接



- 如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需修改DBUtil类源代码
- 如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类
- 违背开闭原则，系统扩展性较差

组合复用原则 (CRP)

- 重构后…
  - 不使用继承，而是通过聚合的方式复用DBUtil类的数据库连接功能



组合复用原则 (CRP)

- 不符合CRP原则的设计：
  - 例如：假设我们想实现一个HashSet的变种，以跟踪往其中添加元素的次数
    - 输出的值是几？
    - Why？
    - 看看父类的addAll()里面干了什么

```java
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

```java
public class InstrumentedHashSet extends HashSet<Object> {
    private static final long serialVersionUID = 1L;
    // 记录往集合中尝试插入元素的次数
    private int addCount = 0;

    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection<? extends Object> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

```java
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[] { "Snap", "Crackle", "Pop" }));
    System.out.println(s.getAddCount());
}
```

组合复用原则 (CRP)

- 重构后…
  - 组合了Set对象s
  - Set是接口
    - 所有方法均在实现类中实现，不用担心副作用

```
public class InstrumentedSet2 implements Set<Object> {
    private final Set<Object> s;
    private int addCount = 0;

    public InstrumentedSet2(Set<Object> s) {
        this.s = s;
    }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends Object> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
    ... ...
}
```

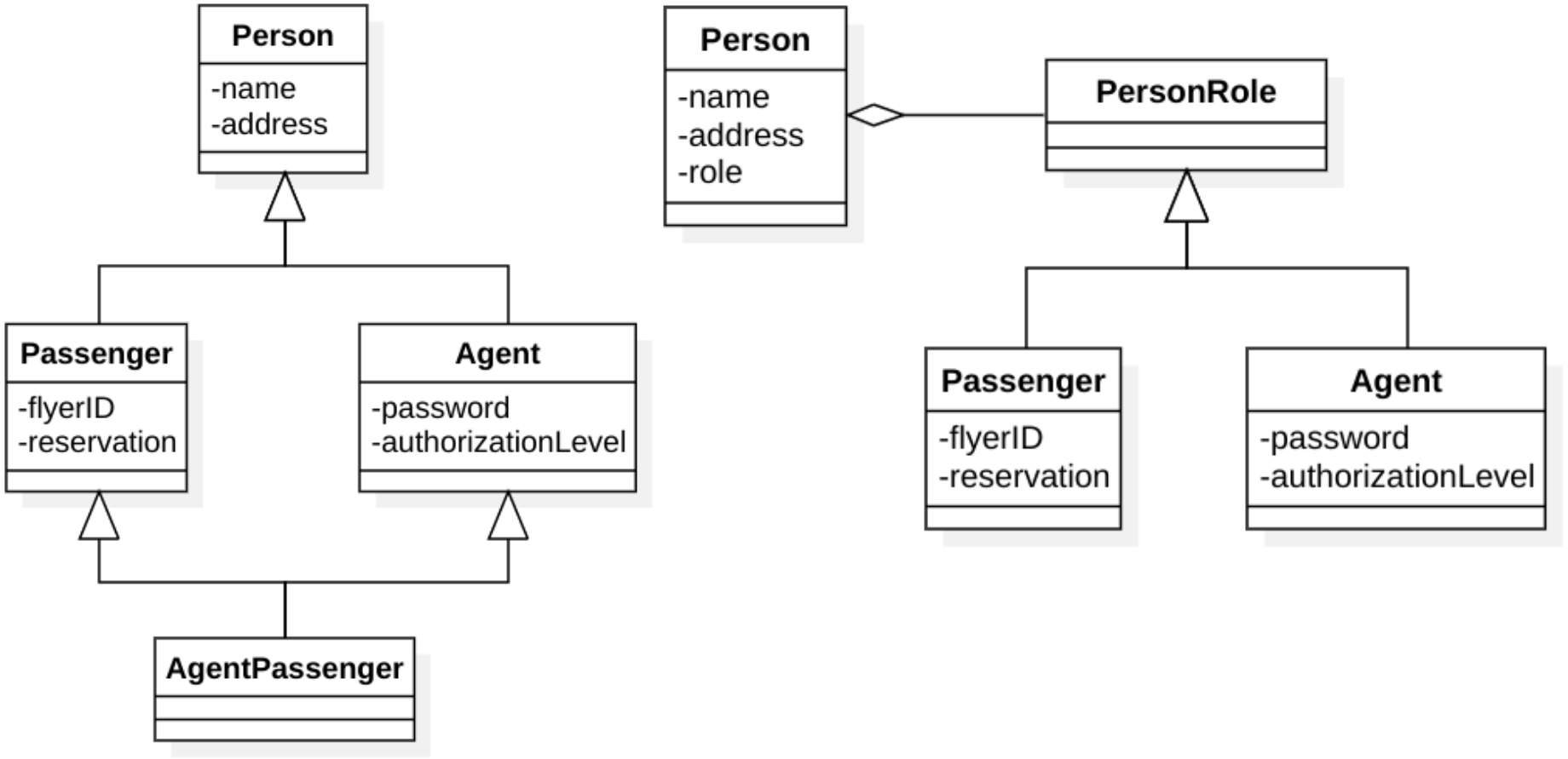
```
public static void main(String[] args) {
    List<Object> list = new ArrayList<Object>();
    InstrumentedSet2 s2 = new InstrumentedSet2(new TreeSet<Object>(list));
    s2.addAll(Arrays.asList(new String[] { "Snap", "Crackle", "Pop" }));
    System.out.println(s2.getAddCount());
}
```

组合复用原则 (CRP)

- 只有当满足下面的条件时才考虑使用继承
  - A subclass expresses “is a special kind of” and not “is a role played by a”
    子类表示“一个特定的种类”而不是“一个扮演的角色”
  - An instance of a subclass never needs to become an object of another class
    子类的实例永不需要成为另一个类的对象
  - A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
    子类扩展而不是重写（或去掉）父类的功能
  - A subclass does not extend the capabilities of what is merely a utility class
    子类不是扩展仅作为工具类的功能

组合复用原则 (CRP)

- 下面哪种设计中使用继承是合适的？



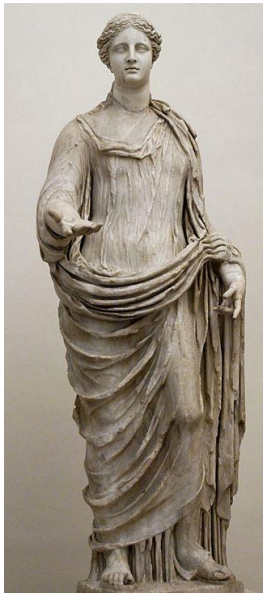
Demeter法则 (Law of Demeter, LoD)
迪米特法则

Demeter法则 (LoD)

- Each unit should only talk to its friends; Don't talk to strangers
  每个模块只和自己的朋友“说话”；不和陌生人“说话”
- Only talk to your immediate friends
  只与你的直接朋友通信
- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
  每个模块只应了解那些与它关系密切的模块的有限知识
  - 也称为：最少知识原则（Least Knowledge Principle）

Demeter法则 (LoD)

- Demeter法则来自于1987年美国东北大学（Northeastern University）一个名为“Demeter”的研究项目，由Ian Holland提出



- Demeter项目
  - 项目主页：https://www2.ccs.neu.edu/research/demeter/
  - 一个关于自适应编程（Adaptive Programming）和面向切面编程（Aspect-Oriented Programming）的项目，以使得软件更易于维护和演化
  - Demeter: 古希腊的收成和农业女神

Demeter法则 (LoD)

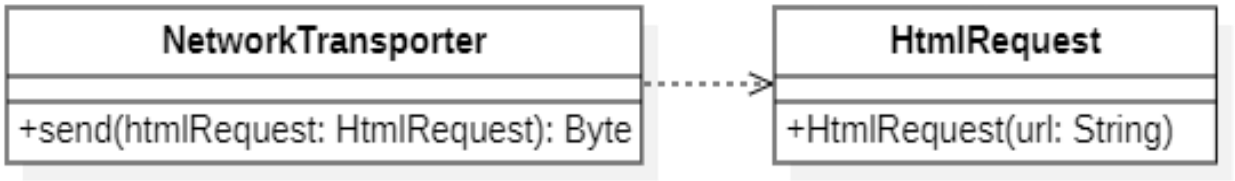
- 理解Demeter法则
  - “高内聚、松耦合”是一个非常重要的设计思想
    - “高内聚”用来指导类本身的设计，相近的功能应该放到同一个类中
    - “松耦合”用来指导类与类之间依赖关系的设计，类之间的依赖关系要简单清晰
  - 不该有直接依赖关系的类之间，不要有依赖
  - 有依赖关系的类之间，尽量只依赖必要的接口（也即“有限知识”）
  - Demeter法则目的是减少类之间的耦合，每个类都应该减少对其他类的了解。这样，一旦其他类发生变化时，受到的影响就会比较小

Demeter法则 (LoD)

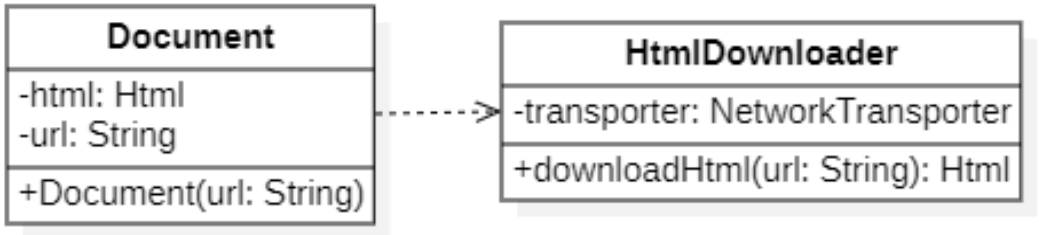
- 理解Demeter法则
  - 怎样做到“不和陌生人说话”？
  - A method of an object may only call methods of:
    一个对象的方法只能调用如下对象的方法
    - The object itself.
      这个对象本身的方法
    - An argument of the method.
      这个方法的参数中引入的对象的方法
    - Any object created within the method.
      这个方法体内创建的对象的方法
    - Any direct properties/fields of the object.
      这个对象的成员属性对象的方法

Demeter法则 (LoD)

- 不符合LoD法则的设计…
  - 不该有直接依赖关系的类之间，不要有依赖
  - 例：搜索引擎爬取网页
    - NetworkTransporter 是底层网络通信类，它应尽可能通用，而不只是服务于下载 HTML，故不该直接依赖HtmlRequest



    - Document 是网页文档，没必要依赖 HtmlDownloader



Demeter法则 (LoD)

- 不符合LoD法则的设计…

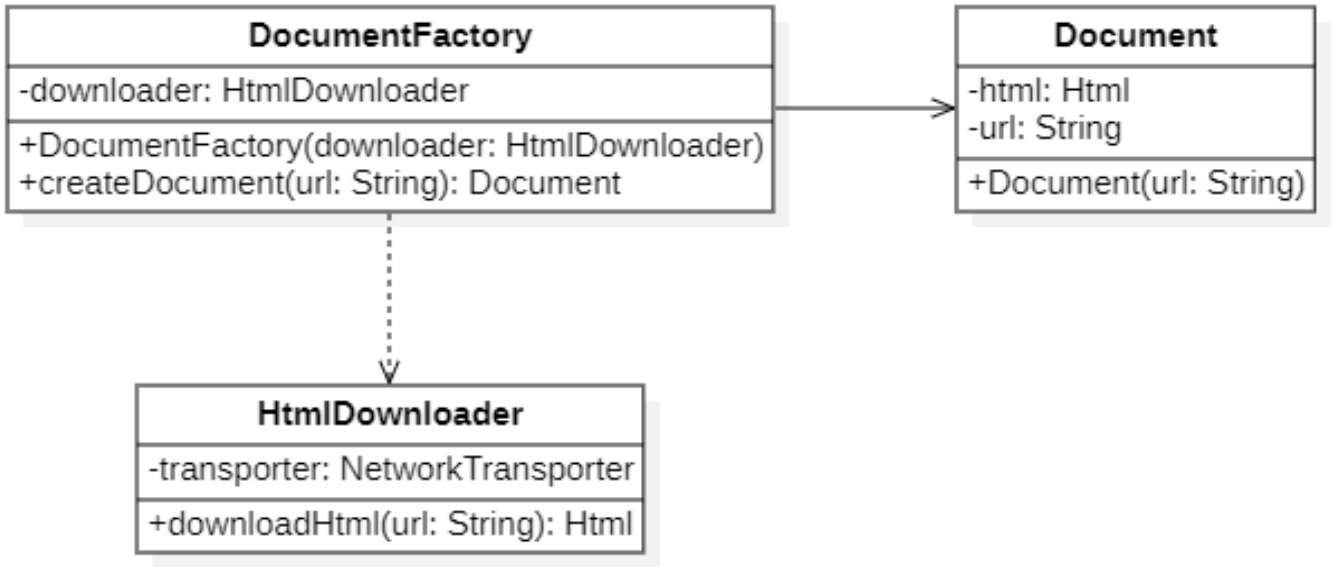
```
public class NetworkTransporter {
    ... ...
    public Byte[] send(HtmlRequest htmlRequest) {
    }
}

public class HtmlDownloader {
  private NetworkTransporter transporter;
  public Html downloadHtml(String url) {
    Byte[] rawHtml = transporter.send(new HtmlRequest(url));
    return new Html(rawHtml);
  }
}

public class Document {
  private Html html;
  private String url;
  public Document(String url) {
    this.url = url;
    HtmlDownloader downloader = new HtmlDownloader();
    this.html = downloader.downloadHtml(url);
  }
}
```

Demeter法则 (LoD)

- 重构后…
  - 把 address 和 content 传给 NetworkTransporter，而不要直接把 HtmlRequest 交给它
  - 引入简单工厂 DocumentFactory 来创建 Document，而不要让 Document 依赖其他类



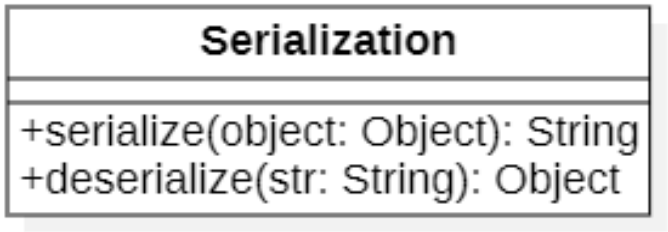
Demeter法则 (LoD)

- 重构后…

```
public class NetworkTransporter {
    ... ...
    public Byte[] send(String address, Byte[] data) {
    }
}
public class Document {
  private Html html;
  private String url;
  public Document(String url, Html html) {
    this.html = html;
    this.url = url;
  }
}
public class DocumentFactory { // 通过一个简单工厂来创建
  private HtmlDownloader downloader;
  public DocumentFactory(HtmlDownloader downloader) {
    this.downloader = downloader;
  }
  public Document createDocument(String url) {
    Html html = downloader.downloadHtml(url);
    return new Document(url, html);
  }
}
```

Demeter法则 (LoD)

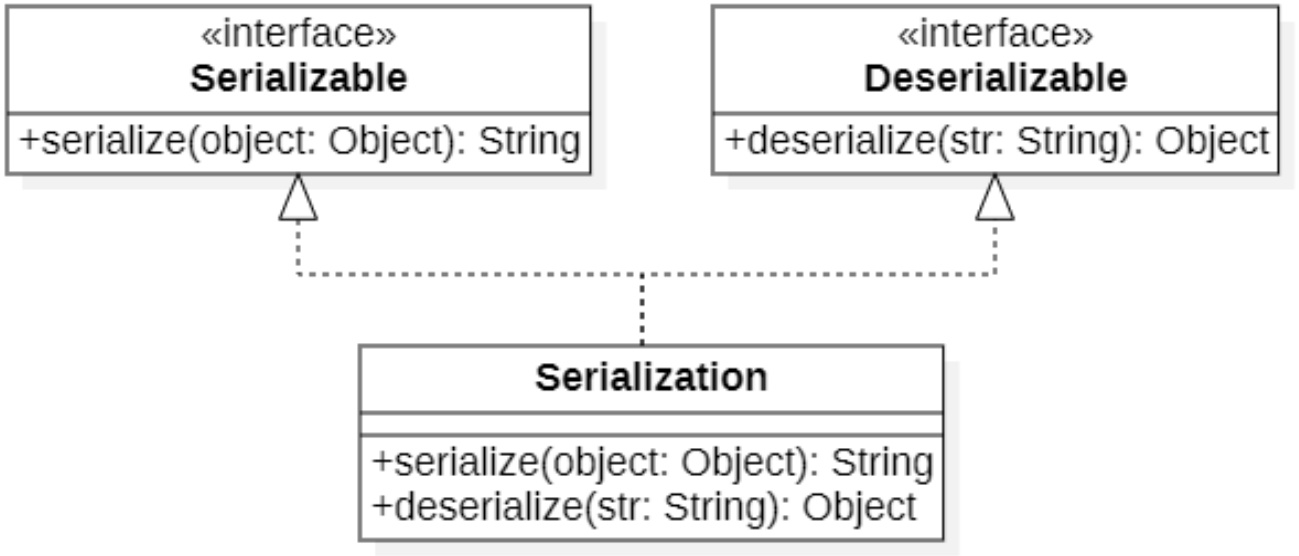
- 不符合LoD法则的设计…
  - 有依赖关系的类之间，尽量只依赖必要的接口
  - 例：序列化



    - 只用到序列化操作的类不应该依赖反序列化接口
    - 只用到反序列化操作的类不应该依赖序列化接口
    - 若序列化和反序列化不是一定要一起使用的话，该设计也是不符合“接口隔离原则”的

Demeter法则 (LoD)

- 重构后…
  - 将 Serialization 类拆分为两个接口，Serializable 只负责序列化，Deserializable 只负责反序列化
    - 虽然用户使用的实现类包含序列化和反序列化实现，但因传入的是其中一个接口，另一个对其透明



KISS原则
Keep It Simple and Stupid

KISS原则

- 理解KISS原则
  - Keep It Simple and Stupid.
  - Keep It Short and Simple.
  - Keep It Simple and Straightforward.

  - 是保持代码可读和可维护的重要手段
  - 代码足够简单，意味着容易读懂，bug 比较难隐藏，即便出现 bug，修复起来也比较简单

KISS原则

- 代码行数越少就越“简单”吗？
  - 例：检查字符串是否是合法的 IP 地址

```
// 第一种实现方式：使用正则表达式
public boolean isValidIpAddressV1(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String regex = "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|[1-9])\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$";
  return ipAddress.matches(regex);
}
```

```
// 第二种实现方式：使用现成的工具类
public boolean isValidIpAddressV2(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String[] ipUnits = StringUtils.split(ipAddress, '.');
  if (ipUnits.length != 4) {
    return false;
  }
  for (int i = 0; i < 4; ++i) {
    int ipUnitIntValue;
    try {
      ipUnitIntValue = Integer.parseInt(ipUnits[i]);
    } catch (NumberFormatException e) {
      return false;
    }
    if (ipUnitIntValue < 0 || ipUnitIntValue > 255) {
      return false;
    }
    if (i == 0 && ipUnitIntValue == 0) {
      return false;
    }
  }
  return true;
}
```

KISS原则

- 代码逻辑复杂就违背 KISS 原则吗？
  - 例：字符串匹配算法--KMP算法

```
// KMP algorithm: a, b分别是主串和模式串；n, m分别是主串和模式串的长度。
public static int kmp(char[] a, int n, char[] b, int m) {
  int[] next = getNexts(b, m);
  int j = 0;
  for (int i = 0; i < n; ++i) {
    while (j > 0 && a[i] != b[j]) { // 一直找到a[i]和b[j]
      j = next[j - 1] + 1;
    }
    if (a[i] == b[j]) {
      ++j;
    }
    if (j == m) { // 找到匹配模式串的了
      return i - m + 1;
    }
  }
  return -1;
}
```

```
// b表示模式串，m表示模式串的长度
private static int[] getNexts(char[] b, int m) {
  int[] next = new int[m];
  next[0] = -1;
  int k = -1;
  for (int i = 1; i < m; ++i) {
    while (k != -1 && b[k + 1] != b[i]) {
      k = next[k];
    }
    if (b[k + 1] == b[i]) {
      ++k;
    }
    next[i] = k;
  }
  return next;
}
```

KISS原则

- 如何写出满足 KISS 原则的代码？
  - 不要使用较为难懂的技术来实现代码
    - 比如复杂的正则表达式、编程语言中过于高级的语法等
  - 要善于使用已经有的工具类库
    - 自己去实现这些类库，出 bug 的概率会更高，维护的成本也比较高
  - 不要过度优化
    - 过度优化代码会牺牲代码的可读性

KISS原则

- YAGNI 跟 KISS 说的是一回事吗？
  - YAGNI: You Aren’t Gonna Need It
  - YAGNI 的含义：不要去设计当前用不到的功能，即不要做过度设计
  - KISS 原则说的是“如何做”的问题（尽量保持简单）
  - YAGNI 原则说的是“要不要做”的问题（当前不需要的就不要做）

DRY原则
Don’t Repeat Yourself

DRY原则

- DRY: Don't Repeat Yourself
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
    在一个系统中，每一项知识必须具有单一、无歧义、权威的表示

- 发生重复的情形
  - 实现逻辑重复
    - 实现逻辑重复，但语义不重复
  - 功能语义重复
    - 实现逻辑不重复，但语义重复（即功能重复）
  - 代码执行重复
    - 既没有逻辑重复，也没有语义重复，但代码中存在“执行重复”

DRY原则

- 实现逻辑重复
  - 没有违反DRY原则
  - 例：用户注册功能
    - 重复的代码片段
      - isValidUserName()
      - isValidPassword()
    - 可否将重复代码合并？
      - isValidUserNameOrPassword()
      - 不可以！！！
      - 因为将来有可能逻辑会变
    - 将某些共性的的东西抽象出来是可以的
      - 如：只允许小写字母和数字

```
public class UserAuthenticator {
  public void authenticate(String username, String password) {
    if (!isValidUsername(username)) {
      // throw InvalidUsernameException
    }
    if (!isValidPassword(password)) {
      // throw InvalidPasswordException
    }
    ... ...
  }

  private boolean isValidUsername(String username) {
    // 用户名非空、长度大于4、只允许小写字母和数字
    ... ...
  }

  private boolean isValidPassword(String password) {
    // 密码非空、长度大于4、只允许小写字母和数字
    ... ...
  }
}
```

DRY原则

- 功能语义重复
  - 违反DRY原则
  - 例：验证IP地址格式合法性
    - 功能相同的两个函数
      - isValidIp(String ipAddress)
      - checkIfIpValid(String ipAddress)
    - 需统一使用同一个函数
      - 试想当验证规则改变时…

```
public boolean isValidIp(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String regex = "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|[1-9])\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$";
  return ipAddress.matches(regex);
}

public boolean checkIfIpValid(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String[] ipUnits = StringUtils.split(ipAddress, '.');
  if (ipUnits.length != 4) {
    return false;
  }
  for (int i = 0; i < 4; ++i) {
    // 验证IP地址的每段数字是否合法
    ... ...
  }
  return true;
}
```

DRY原则

- 代码执行重复
  - 违反DRY原则
  - 例：校验用户登录是否成功
    - 在login()中重复执行的代码
      - EmailValidation.validate(email)
    - 需要将email校验逻辑从UserRepo中移除，统一放到UserService 中

```
public class UserService {
  private UserRepo userRepo;//通过依赖注入或者IOC框架注入

  public User login(String email, String password) {
    boolean existed = userRepo.checkIfUserExisted(email, password);
    ... ...
    User user = userRepo.getUserByEmail(email);
    return user;
  }
}

public class UserRepo {
  public boolean checkIfUserExisted(String email, String password) {
    if (!EmailValidation.validate(email)) { ... ... }
    ... ...
  }
  public User getUserByEmail(String email) {
    if (!EmailValidation.validate(email)) { ... ... }
    ... ...
  }
}
```

DRY原则

- 怎么提高代码复用性？
  - 减少代码耦合
  - 满足单一职责原则
  - 模块化
  - 业务与非业务逻辑分离
  - 通用代码下沉
  - 继承、多态、抽象、封装
  - 应用模板等设计模式

小结

小结

- 单一职责原则
  - 一个类只负责一个功能
- 开闭原则
  - 对扩展开放，对修改关闭
- 里氏替换原则
  - 能使用基类对象地方一定能使用其子类对象
- 接口隔离原则
  - 不应该依赖不相关的接口
- 依赖倒置原则
  - 抽象不应该依赖具体而应该反过来；
  - 针对接口而不是实现编程

小结

- 组合复用原则
  - 复用时尽量使用聚合/组合而不是继承
- Demeter法则
  - 尽可能少地与其他软件实体（类/模块/子系统等）发生依赖关系
- KISS原则
  - 保持简单的设计
- DRY原则
  - 在一个系统中，每一项知识必须具有单一、无歧义、权威的表示

Thank you!