《面向对象程序先导》
Lec8-代码风格与代码重构

北京航空航天大学计算机学院

吴际

2023.11.3

代码风格

- 代码风格强调规范性
  - 代码的可读性，添加新功能的难易程度，和维护成本
- 代码风格检查是工业界开发的普遍实践
  - 课程使用的代码风格检查规则来自于阿里公司
- 目前我们主要从可读性和降低耦合复杂度角度来部署检查规则

目前 Checkstyle 所采用的主要规则

- 1. 限制文件行数（500行），限制方法长度（100行），限制单行长度（100字符）：引导大家多做封装和逻辑拆分，降低复杂性和提高代码**可读性**
- 2. 不使用 import *：明确指出 import 内容，提高代码**可读性**，避免污染命名空间
- 3. 运算符周围添加空格、单行单语句、强制4个空格缩进、不使用 Tab：提高代码**可读性**
- 4. 限制嵌套层数：鼓励简化逻辑，鼓励封装，提高**可读性**
- 5. Switch语句的fall through 和 default分支：避免遗漏情况和写出容易造成误会的代码
- 6. 可见性约束：鼓励封装，减少不必要的数据耦合
- 7. 不要使用缩写命名：鼓励明确的命名方式，晦涩难懂的命名会极大降低**可读性**
- 8. 驼峰命名规范（变量使用小驼峰命名 camelCase，类名使用帕斯卡命名 PascalCase）：保持命名统一性，提高**可读性**
- 9. 不要对参数重新赋值：Java 中对参数重新赋值并不会改变实参的值，反而可能造成误解
- 10. 不要使用空 block：空 block 没有作用，给人的理解造成迷惑

阿里使用的其他风格检查

- 方法参数类型必须一致，不要出现自动装箱拆箱操作，反例：

```
public static int handel(Integer value) {
    return value;
}
```

- 容易导致 Null Pointer Exception 且难以排查

阿里使用的其他风格检查

- 使用 `equals` 方法小心空指针问题：`obj.equals(other)`
  - `other`或`obj` 为 `null` 都会导致空指针异常问题
- 建议
  - 使用常量或确定值在前，如`"test".equals(obj);`
  - 或改用`Objects.equals`，如`Objects.equals(obj, other)`

阿里使用的其他风格检查

- 不要在异常处理的finally 块中使用 return

```
public static boolean getValue(String text) {
    try {
        return text.equals("123");
    } finally {
        return true;
    }
}
```

- 此写法始终返回 true，因为 finally 块总是会执行

阿里使用的其他风格检查

- 单元测试原则：
  - 单元测试需要全自动执行，采用非交互方式
  - 不要使用 `System.out` 输出，否则需要人手动检查单元测试结果
  - 应使用 `assertXXX`来自动检查是否有错
- 单元测试需要多次重复执行
  - 实现代码修改，进行回归
- 每次代码变更应当首先运行单元测试，确保通过所有用例再提交到git库

代码坏味

- 代码坏味：代码的腐化现象
- 主要原因
  - 对需求易变性的估计不足
  - 未能在代码编写过程中对结构进行持续的追踪分析
- 掌握识别和改善代码坏味道的方法，能让重构变得高效，是重构的重中之重

神秘命名（Mysterious Name）

- 类名、函数名和变量名等带有丰富的含义，如果名称与所表达的语义一致，程序理解就会简单很多

- 好的命名能让代码自解释，可以减少不必要的注释
- 命名是任何开发和重构都需要重视的工作
  - 没有严格的规则和指南
  - 名字有意义、和逻辑相关联，相互可区分

例子展示

```
public void print(long a){

   DecimalFormat b = new DecimalFormat("0000000000");
   String c = b.format(a);

   java.text.MessageFormat d =
        new java.text.MessageFormat("({0})-{1}-{2}");

   String[] e ={c.substring(0, 3),
                c.substring(3, 6),
                c.substring(6)};
   System.out.println(d.format(e));
}
```

例子展示

```
public void print(long phoneFmt){

   DecimalFormat phoneDecimalFmt = new DecimalFormat("0000000000");
   String phoneRawString = phoneDecimalFmt.format(phoneFmt);

   java.text.MessageFormat phoneMsgFmt =
        new java.text.MessageFormat("({0})-{1}-{2}");

   String[] phoneNumArr = { phoneRawString.substring(0, 3),
                            phoneRawString.substring(3, 6),
                            phoneRawString.substring(6)};
   System.out.println(phoneMsgFmt.format(phoneNumArr));
}
```

重复代码

- 重复代码会带来很高的缺陷修复和维护成本
  - 需要人力找出所有相似的代码，容易出现疏漏
- 提炼重复代码是经典的重构办法
  - 如果重复代码在同一个类中，提炼重复代码形成一个新方法
  - 如果重复代码在兄弟类中，提炼重复代码到父类
  - 如果重复代码在无关联类中，提炼重复代码到第三者或两个类中的一个

过长函数

- 规模是构成程序复杂性的核心要素
  - 代码行数、控制流、数据流
- 函数的代码行数越大，就越难理解其功能实现
- 过长函数也易违反**单一职责原则**，易导致产生重复代码
  - checkstyle 要求 100 行
  - 航空领域要求不超过30行
- 常见的处理方法是按照函数的步骤或功能进行拆分

过大的类

- 与过长函数类似，过大的类也是典型的代码坏味道
  - 一个类做太多的事情，维护太多的功能，导致难以理解和维护
  - 典型特征：代码行数过长、有过多属性和方法、与其他类的关联过多等
- 常见处理方法就是拆分
  - 按照数据拆分是常用的策略
  - 水平拆分：形成多个相互独立的类
  - 垂直拆分：形成层次，一个公共父类和若干个不同子类

数据泥团（Data Clumps）

- 程序中有些数据总是一起出现，处理时不能出现遗漏
  - 根本原因是这些数据具有内在的关联性
- 应对这些数据进行封装和集中处理
  - 提供数据访问和修改方法
  - 建立类之间的关联关系
  - ➔形成层次

数据泥团（Data Clumps）

- 考虑下面的 User 类：

```
public class User {
    private String firstName;
    private String midName;
    private String lastName;
    // other attributes
    public String getName() {
        return firstName + " " + midName + " " + lastName;
    }
    // other methods
}
```

- User 中的三个name属性具有内在的关联，会经常一起出现，应构造一个 UserName 类进行统一处理

数据泥团（Data Clumps）

```
public class User {

    private UserName name;
    // other attributes

    public String getName() {
        return name.toString();
    }
    // other methods
}
```

```
public class UserName {
    private String firstName;
    private String midName;
    private String lastName;
    // some getter and setter
@Override
    public String toString() {
        return firstName + " " +
                midName + " " +
                lastName;
    }
}
```

发散式变化（Divergent Change）

- 如果某个类经常需要因为不同的原因在不同的方向上发生变化，发散式变化就出现了
  - 如增加属性、改写方法中的计算逻辑等
- 发散式变化往往是违反单一职责原则的特征
  - 一个类做的事情有点散，不专注
- 建议：按照数据和方法功能进行拆分

发散式变化（Divergent Change）

- 针对下属工厂模式的实现：

```
public class Factory {
   public static Product create(String type) {
      if (type.equalsIgnoreCase("A")) {
         return new ProductA(100);
      } else if (type.equalsIgnoreCase("B")) {
         return new ProductB(200);
      } else {
         System.out.println("Wrong kind!");
         return null;
      }
   }
}
```

- 新增产品或者ProductA或ProductB的内容发生变化，该工厂类都需要修改
- 把工厂类的方法抽象出来，然后对每种产品新增一个工厂

霰弹式修改（Shotgun Surgery）

- 霰弹式修改：某个变化导致一个类或多个类中的多处代码进行细小修改
- 霰弹式修改是一种高耦合的表现
  - 关注点分离做的不够
  - 违背单一职责
  - 未识别出公共行为，导致多处重复
- 引入封装，建立层次关联

霰弹式修改（Shotgun Surgery）

- 有一个Account类
  - 提供取款、转账、生活费用代扣功能
  - 要求账户余额必须不能少于某个数额，如300（代扣额度）
- Account类的方法中很容易出现这样的语句
  - if(balance < 300){…}
- 如果代扣额度发生变化，则多处这样的判断语句都需要修改
- 该判断语句的核心是确认账号对象的某种状态
  - 应增加一个方法boolean IsAccountUnderflow(threshold)

依恋情结（Feature Envy）

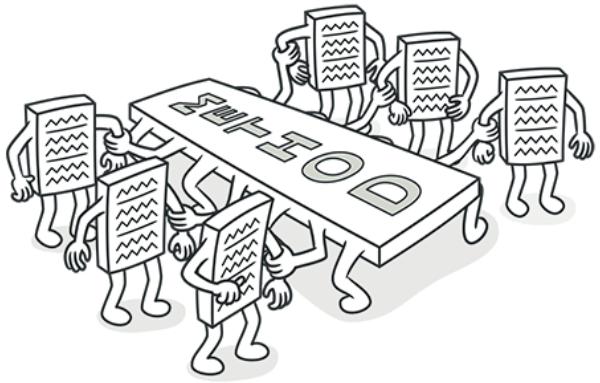
- 依恋情结：如果类 A 的某个方法，大量的使用类 B 的方法

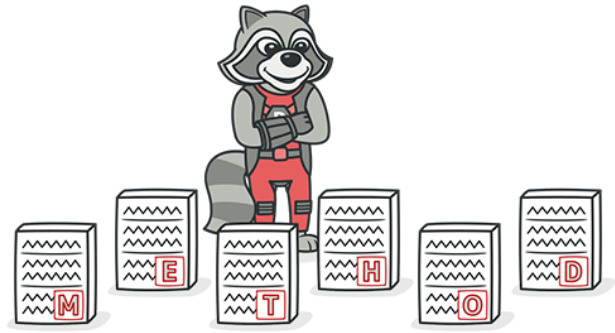
```
public class User {
    private Phone phone;
    public User(Phone phone) {
        this.phone = phone;
    }
    public void getFullPhoneNumber() {
        System.out.println("areaCode: " + phone.getAreaCode());
        System.out.println("prefix: " + phone.getPrefix());
        System.out.println("number: " + phone.getNumber());
    }
}
```

密集使用B类方法的A类方法更应该放在类 B 中，在类 A 中直接调用即可

狎昵关系（Inappropriate Intimacy）

- 如果两个类互相调用对方的方法来获取对方的私有化数据，就形成了过分的耦合关系
  - 本质是业务逻辑上某些数据具有内在的关联性，但却被封装在不同的类中
- 解决办法：识别紧密关联的属性数据，形成单独的类，并把相应的处理一并封装起来
  - 降低耦合





中间人（Middle Man）

- 面向对象设计会在类之间形成关联关系和行为代理关系(delegation)
  - 方法调用
  - 单层方法调用是一种很好的封装
  - 多层方法调用则意味着某种全局控制
- 多层调用案例
  - `a.getB().getC().getD()`
  - getB返回的对象实际充当**中间人**，把调用getC转为被代理的getD请求
- 解决方案1：把对getD的调用封装进getC方法中
- 解决方案2：取消getB返回的中间人职责，使得通过对象a可以快速调用到getD方法

被拒绝的馈赠（Refused Bequest）

- 子类应当继承父类的数据和方法
- 如果子类只使用了父类很少的行为，甚至是直接覆盖父类继承得到的行为，意味着继承体系设计不当
- 案例分析：
  - 举办活动和卖票，活动包含属性：日期、主题、基础价格
  - 票有两种：普通票（Ticket）、VIP票（VIPTicket）
  - 普通票（Ticket）
    - 查询票价：如果周一到周四➔票价=原价，如果周五➔票价=原价x2
    - 退款：活动开始前可以进行退款
  - VIP票（VIPTicket）
    - 查询票价：如果周一到周四➔票价=原价+100，如果周五➔票价=原价x2+100
    - 是否有附加活动：如果有则返回true，否则返回false

被拒绝的馈赠（Refused Bequest）

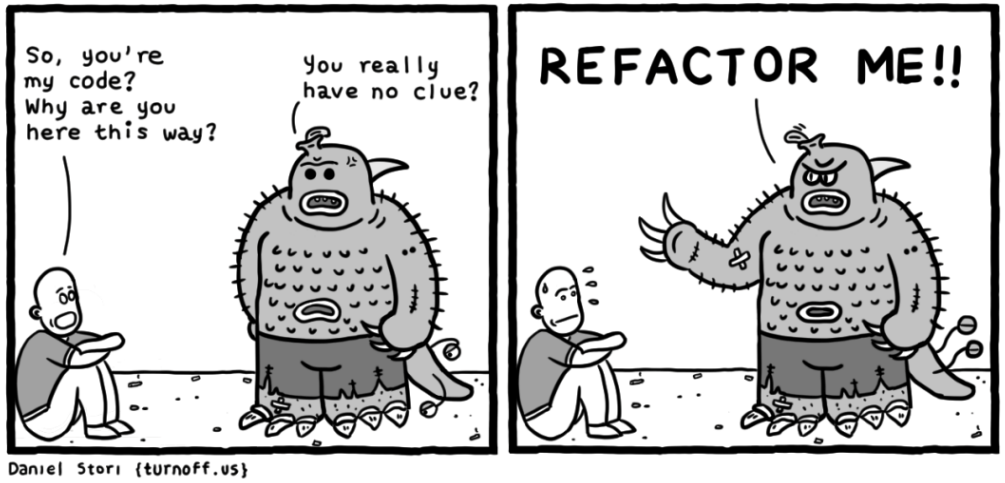
- Activity类
  - 活动主题session、活动日期date、基础票价price
- 把Ticket作为父类，VIPTicket作为子类
  - Ticket类关联到Activity类
  - Ticket类提供getPrice方法、refund方法和getSession方法
  - VIPTicket重新实现getPrice方法，增加hasAdditionalActivity方法
    - getPrice(){super.getPrice() + 100;}
- 问题
  - 如果Ticket的定价需求有变更，而VIP无变更，则出现bug
  - VIPTicket是不允许退票的，但却提供了refund方法

被拒绝的馈赠（Refused Bequest）

- 解决方案1
  - 建立一个BaseTicket类，让Ticket和VIPTicket均为其子类
  - 各自实现不同的getPrice，没有逻辑关联
- 解决方案2
  - 把票所蕴含的权利和义乌等抽象出来形成独立的接口
    - RefundableTicket只提供refund方法
    - BasicTicket只提供getPrice和getSession方法
    - ExtensionalTicket提供附件活动安排，提供hasAdditionalActivity方法
  - Ticket实现RefundableTicket和BasicTicket
  - VIPTicket实现ExtensionalTicket和BasicTicket
- 解决方案3
  - VIPTicket关联到Ticket（封装Ticket），形成代理结构

为什么重构

- 需求变化和缺陷修复都会导致设计发生变化
- 如果不及时进行重构，就会出现代码坏味道
- 重构可以改善程序的设计复杂性，使得代码更容易理解
- 设计以实现需求为第一目标，“老”设计成分不适应“新”需求是正常现象
- 重构不可或缺



什么时候重构

- 在添加新功能之前和在完成新功能后都是理想的重构时机
- 设计规划：OO 课程的项目作业是迭代式的，重构很重要
  - 在拿到新需求后，首先检查已有的代码设计，通过重构来尽可能减少新功能所需的新增代码量
- 结构复杂：如果往程序中添加新功能会触发多处代码修改，甚至是多处代码都需要丢掉重写，必须要进行重构

OO 正课介绍

- 课程目标：运用面向对象思维开展**层次化**架构设计，并构造**高质量**复杂程序
- 四个主题单元
  - 层次化设计
  - 线程安全设计
  - 规格化设计
  - 模型化设计

OO 正课介绍

- 每个主题包括 4 次授课、4次作业、2 次实验和 2 次研讨
  - 前三次作业为迭代式开发，最后一次博客总结
  - 研讨课分组开展，并有主题报告
  - 实验online方式，限时完成
- 迭代式开发作业
  - 每次新增一些需求，有可能会对之前的需求产生一些影响
  - 中测、强测、互测和代码修复
  - 互测：根据强测成绩把同学们匿名分配到不同的房间，通过提交符合规定的测试样例 hack 其他同学以获得加分，并努力避免自己被 hack 扣分

OO 正课介绍

- 博客作业
  - 建议同学们提前读一读往届同学的博客
  - 认真对待，用心总结
- 相对于OOPre，OO代码作业的难度会有一定程度的提升
  - 当然，你的能力也在同步提升
  - 学会搭建评测机，开展自动化测试是个重要技能
  - 单元测试需要继续执行下去
- 和OOPre一样，我们希望OO课程也是一个愉快的学习过程
  - 师生共建

OO 正课的必要准备

- 技术准备
  - Java语言和编程环境
  - Java程序的测试和调试
  - Gitlab系统的使用
- 知识准备
  - 学习和了解设计模式
  - 自学关于多线程的基础概念
  - 自学关于程序规格的基础概念
  - 课程组会在合适时间提前发一些素材

最后一次作业

- 博客总结
  - 1. 作业最终的架构设计, 在迭代中的架构调整及考虑
  - 2. 使用junit的心得体会
  - 3. 学习oopre的心得体会(包括但不限于从面向过程编程过渡到面向对象编程的体会)
  - 4. 对oopre课程的简单建议（不多于两条)
- 在CSDN上发布博客
  - 按照要求来发布