

CS10102302

软件重构

赵君峤 副教授

计算机科学与技术系 电子与信息工程学院

同济大学

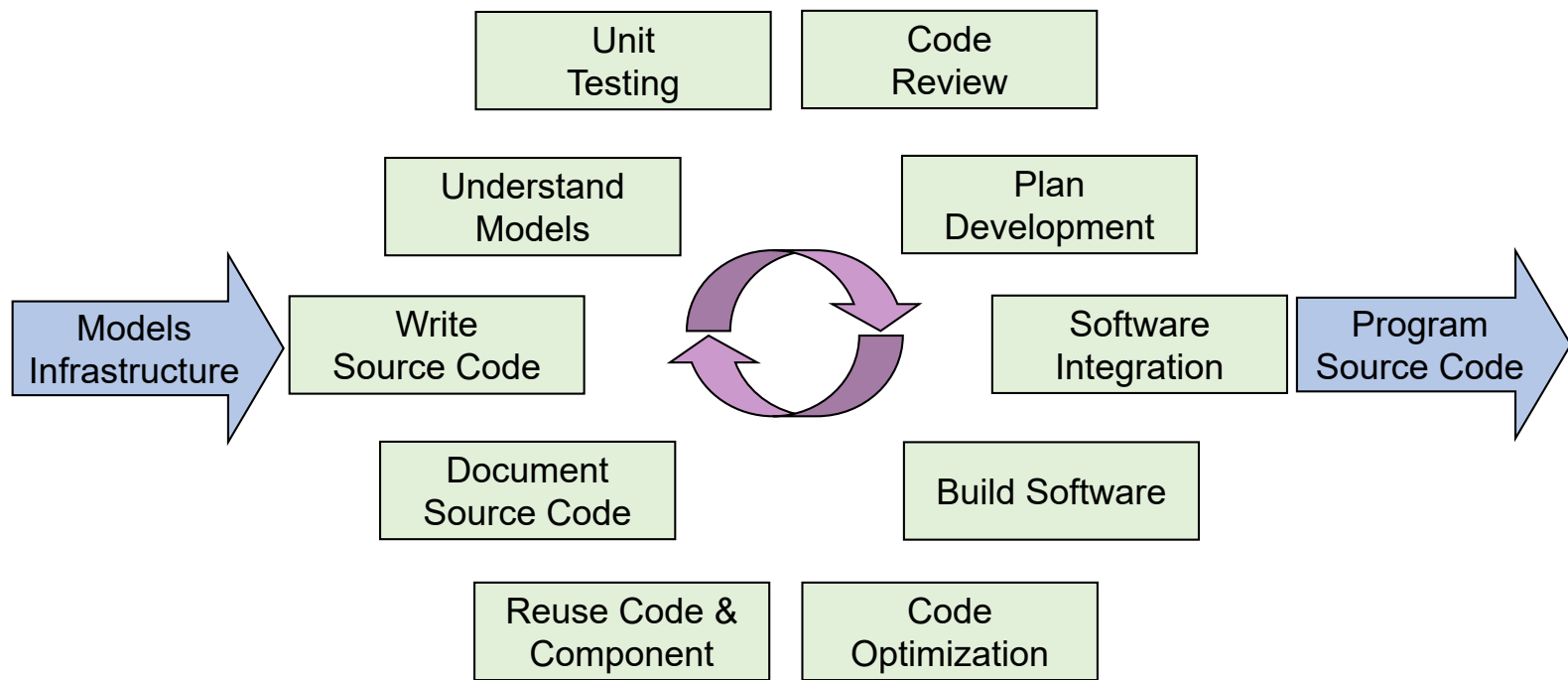
教学提纲

1

编程原则与实践

- 软件编码工作
- 软件编码规范
- 代码走查
- 良好的编码实践

软件编码工作



软件编码工作

程序设计

- 理解软件的需求说明和设计模型
- 补充遗漏的或剩余的详细设计
- 设计程序代码的结构

设计审查

- 检查设计结果
- 记录发现的设计缺陷（类型、来源、严重性）

编写代码

- 应用编码规范进行代码编写
- 所编写代码应该是易验证的



软件编码工作

代码走查

- 确认所写代码完成了所要求的工作
- 记录发现的代码缺陷（类型、来源、严重性）

编译代码

- 修改代码的语法错误

测试所写代码

- 对代码进行单元测试
- 调试代码修改错误



软件编码规范

软件编码规范是与特定语言相关的描写如何编写代码的规则集合。

目的

- 提高编码质量，避免不必要的程序错误
- 增强程序代码的可读性、可重用性和可移植性

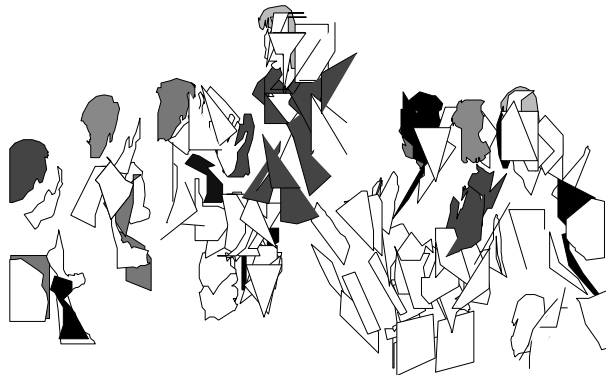
软件编码规范 <https://code.google.com/p/google-styleguide/>

代码检查

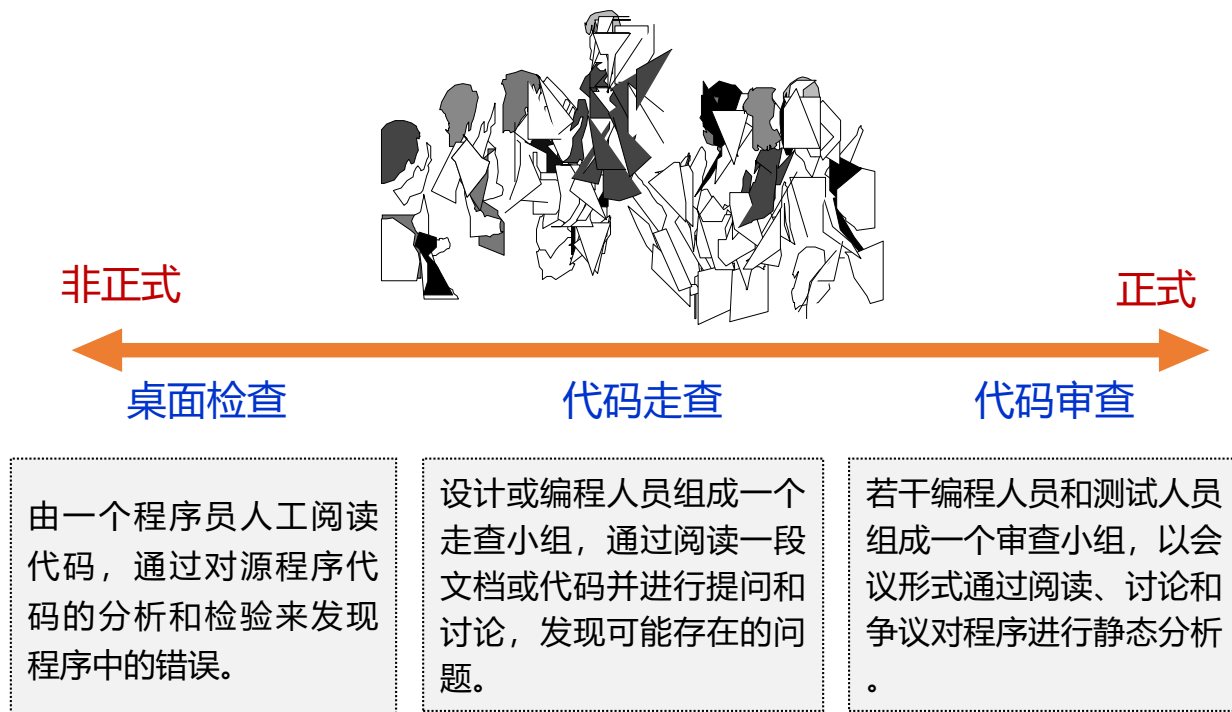
代码检查 (Code Review) 是一种用来确认方案设计和代码实现的质量保证机制, 用于在开发过程中改进代码质量。

代码检查的作用

- 检查设计的合理性
- 互为 Backup
- 分享知识、设计、技术
- 增加代码可读性
- 处理代码中的“地雷区”



代码检查



缺陷检查表

编程规范

- 按照具体编程语言的编码规范进行检查，包括命名规则、程序注释、缩进排版、声明与初始化、语句格式等。

面向对象设计

- 类的设计和抽象是否合适
- 是否符合面向接口编程的思想
- 是否使用合适的设计模式

性能方面

- 在出现海量数据时，队列、表、文件在传输、上载等方面是否会出现问题，是否控制如分配的内存块大小、队列长度等
- 对Hashtable、Vector等集合类数据结构的选择和设置是否合适
- 有无滥用 String 对象的现象
- 是否采用通用的线程池、对象池等高速缓存技术以提高性能
- 类的接口是否定义良好，如参数类型等应避免内部转换

性能方面（续）

- 是否采用内存或硬盘缓冲机制以提高效率？
- 并发访问时的应对策略
- I/O方面是否使用了合适的类或采用良好的方法以提高性能（如减少序列化、使用 buffer 类封装流等）
- 同步方法的使用是否得当，是否过度使用？
- 递归方法中的迭代次数是否合适（应保证在合理的栈空间范围内）
- 如果调用了阻塞方法，是否考虑了保证性能的措施
- 避免过度优化，对性能要求高的代码是否使用profile工具

资源释放处理

- 分配的内存是否释放，尤其在错误处理路径上（如 C/C++）
- 错误发生时是否所有对象被释放，如数据库连接、Socket、文件等
- 是否同一个对象被释放多次（如 C/C++）
- 代码是否保存准确的对象引用计数

缺陷检查表

程序流程

- 循环结束条件是否准确
- 是否避免了死循环的产生
- 对循环的处理是否合适，应考虑到性能方面的影响

线程安全

- 代码中所有的全局变量是否是线程安全的
- 需要被多个线程访问的对象是否线程安全，检查有无通过同步方法保护
- 同步对象上的锁是否按相同的顺序获得和释放以避免死锁，注意错误处理代码
- 是否存在可能的死锁或是竞争，当用到多个锁时，避免出现类似情况：线程A获得锁1，然后锁2，线程B获得锁2，然后锁1
- 在保证线程安全的同时，注意避免过度使用同步，导致性能降低

数据库处理

- 数据库设计或SQL语句是否便于移植（注意与性能会存在冲突）
- 数据库资源是否正常关闭和释放

数据库处理（续）

- 数据库访问模块是否正确封装，便于管理和提高性能
- 是否采用合适的事务隔离级别
- 是否采用存储过程以提高性能
- 是否采用 PreparedStatement 以提高性能

通讯方面

- Socket 通讯是否存在长期阻塞问题
- 发送接收的数据流是否采用缓冲机制
- Socket 超时处理和异常处理
- 数据传输的流量控制问题

JAVA对象处理

- 对象生命周期的处理，是否对象引用已失效可设置null并被回收
- 在对象传值和传参方面有无问题，对象的 clone 方法使用是否过度
- 是否大量经常地创建临时对象
- 是否尽量使用局部对象（堆栈对象）
- 在只需要对象引用的地方是否创建了新的对象实例

缺陷检查表

异常处理

- 每次当方法返回时是否正确处理了异常，如最简单的处理是记录日志到日志文件中
- 是否对数据的值和范围是否合法进行校验，包括使用断言
- 在出错路径上是否所有的资源和内存都已经释放
- 所有抛出的异常是否都得到正确的处理，特别是对子方法抛出的异常，在整个调用栈中必须能够被捕捉并处理
- 当调用导致错误发生时，方法的调用者应该得到一个通知
- 不要忘了对错误处理部分的代码进行测试，很多代码在正常情况下执行良好，而一旦出错整个系统就崩溃了？

方法（函数）

- 方法的参数是否都做了校验
- 数组类结构是否做了边界校验

方法（续）

- 变量在使用前是否做了初始化
- 返回堆对象的引用，不要返回栈对象的引用
- 方法的 API 是否被良好定义，即是否尽量面向接口编程，以便于维护和重构

安全方面

- 对命令行执行的代码，需要详细检查命令行参数
- WEB 类程序检查是否对访问参数进行合法性验证
- 重要信息的保存是否选用合适的加密算法
- 通讯时考虑是否选用安全的通讯方式

其他

- 日志是否正常输出和控制
- 配置信息如何获得，是否有硬编码

良好的编程实践

不要编写需要外部文档支持的代码，这样的代码是脆弱的，要确保你的代码本身读起来就很清晰。



应该如何做到这一点？

良好的编程实践

编写自文档化的代码

- 唯一能完整并正确地描述代码的文档是代码本身
- 编写可以阅读的代码，其本身简单易懂



良好的编程实践

学会只编写够用的注释，过犹不及，重视质量而不是数量。应该把时间花在编写不需要大量注释支持的代码上，即让代码自文档化。

- 好的注释解释为什么，而不是怎么样
- 不要在注释中重复描述代码
- 当你发现自己在编写密密麻麻的注释来解释代码时，需要停下来看是否存在更大的问题
- 想一想在注释中写什么，不要不动脑筋就输入，写完之后还要在代码的上下文中回顾一下这些注释，它们是否包含正确的信息？
- 当修改代码时，维护代码周围的所有注释

良好的编程实践

函数编写的第一条规则是短小，第二条规则是更短小。
函数应该做一件事，做好这件事，并且只做这件事。




编写短小的函数

良好的编程实践

软件系统应将起始过程和起始过程之后的运行时逻辑分离开，在起始过程中构建应用对象，也会存在互相缠结的依赖关系。

- **方法一：** main函数创建系统所需的对象，再传递给应用程序，应用程序只管使用。
- **方法二：** 使用抽象工厂模式让应用自行控制何时创建对象，但构造的细节却隔离于应用程序之外。
- **方法三：** 使用依赖注入（或称控制反转），即通过第三方框架将需要依赖的类主动注入进来。



依赖注入

良好的编程实践

浮点运算

- 一个数值的**准确度**反映了这个值与它所代表的数量之间的接近程度，即衡量我们得到这个数值过程中的误差。
- 数字的**精度**表示其规格的紧密程度，即能够区分所表达的数与其邻近数的程度。
- 数字应该使用它们的准确度相当的精度来存储，但是应该使用底层硬件所能有效支持的最大精度来操作。
- 大多数有尽十进制小数不能被有尽二进制小数准确表示。

$$0.1_{10} = 0.1000000000000000055511151231257827021181583404541015625$$

- 在正常的表示范围内乘以或除以2的幂（2、4、8、16、...）永远都是精确运算，永远不存在舍入问题。

教学提纲

2

软件重构

- 软件重构概述
- 代码的坏味道
- 软件重构方法

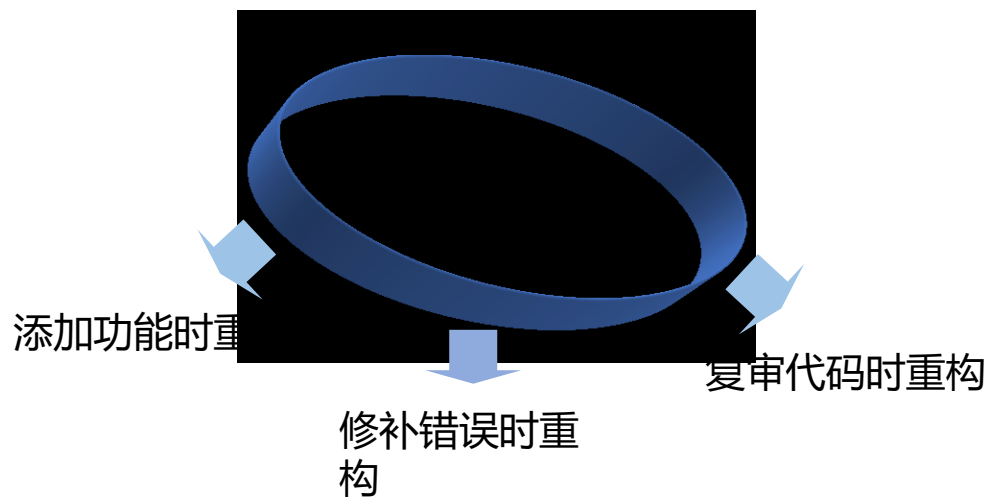
重构的概念

重构 (Refactoring) 是对软件内部结构的一种调整，其目的是在不改变软件功能和外部行为的前提下，提高其可理解性、可扩展性和可重用性。



何时重构?

三次法则： 第一次做某件事时只管去做；
第二次做类似的事会产生反感，但无论如何还是去做；
第三次再做类似的事就该重构。



什么时候不适合重构？

代码太混乱，设计完全错误

- 与其重构不如重新开始

明天是Deadline

- 永远不要做Last-Minute-Change；应推迟重构但不可忽略，即使进入Production的代码都正确地运行。

重构的工作量显著地影响估算

- 一个任务的估算时间是3天，如果为了重构，就需要更多的时间。
- 推迟重构但不忽略，可以把重构作为一个新任务，或者安排在重构的迭代周期中完成。

重构与添加新功能



添加新功能

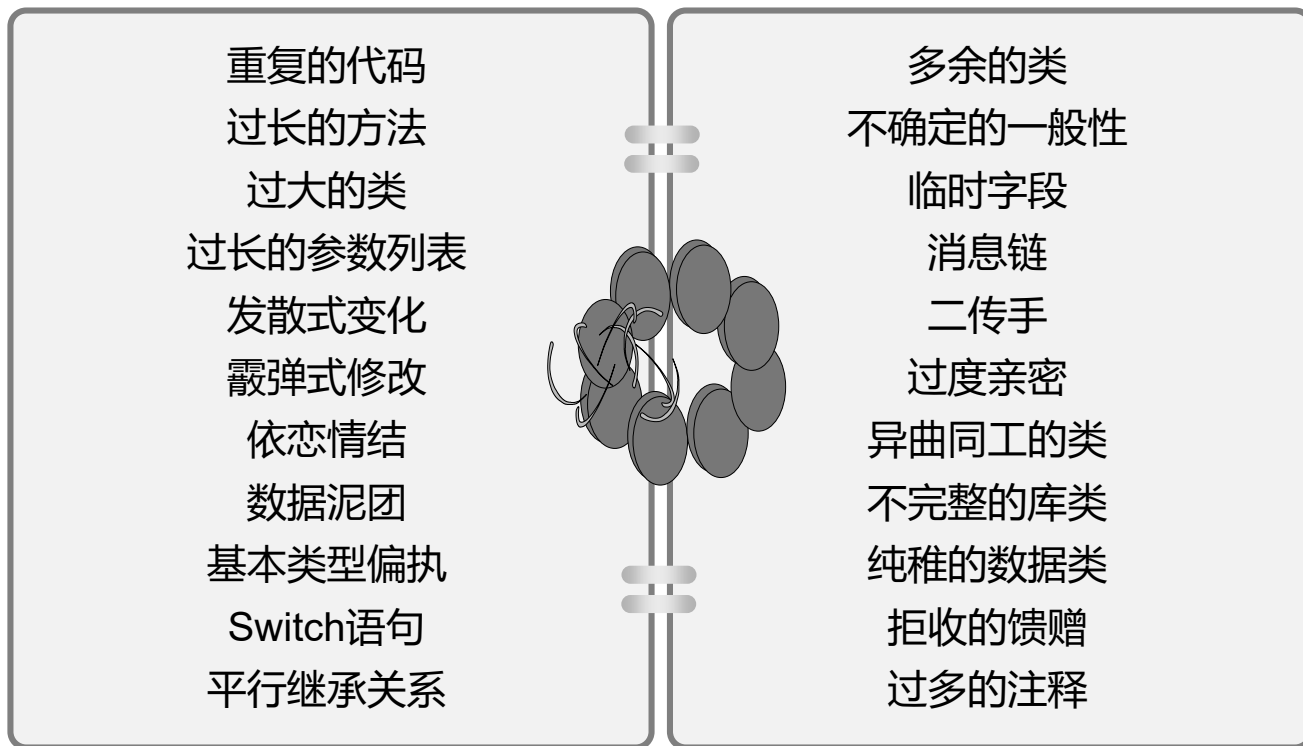
- 添加新功能时，不应该修改既有代码，只管添加新功能

重构

- 重构时不再添加功能，只管改进程序结构

重构与添加新功能可交替进行

代码的坏味道



重复的代码

症状：

- 容易形式：两个代码段看上去几乎相同
- 困难形式：两个代码段都拥有几乎相同的作用

措施：

- 抽取方法：如果同一个类的两个函数存在重复代码，则将重复代码抽取成一个函数，在原有函数的对应处调用所抽取的函数。
- 抽取类：如果两个毫不相关的类存在重复代码，则将重复代码抽取成到一个独立的类中或者某个类中，其他类引用这个类。
- 替换算法：如果有些函数以不同的算法做相同的事，则选择一个比较清晰的函数，而将其其他函数的算法替换掉。

重复的代码

```
protected void queryBtn(object sender, EventArgs e) {  
    //如果项目编号不为空时  
    if (this.xmbhText.Text.ToString().Trim().Equals("") == false) {  
        //对输入的查询条件——项目编号是否合法进行校验  
        Byte[] MyBytes = System.Text.Encoding.Default.GetBytes(  
            this.xmbhText.Text.ToString().Trim());  
        if (MyBytes.Length > 10) {  
            MessageBox.Alert("项目编号不能超过10个字节! ");  
            this.xmbhText.Focus();  
            return;  
        }  
    }  
    //如果项目名称不为空时  
    if (this.xmmcText.Text.ToString().Trim().Equals("") == false) {  
        //对输入的查询条件——项目名称是否合法进行校验  
        Byte[] xmmccheck = System.Text.Encoding.Default.GetBytes(  
            this.xmmcText.Text.ToString().Trim());  
        if (xmmccheck.Length > 40) {  
            MessageBox.Alert("项目名称不能超过40个字节! ");  
            this.xmmcText.Focus();  
            return;  
        }  
    }  
    //实现数据库绑定  
    GridView1_DataBind();  
}
```

过长的函数

症状：

- 只要看到超过N（如10）行代码的方法，立即检查是否可以重构之。
- 长度是一个警告信号，并不表示一定有问题。

措施：

- 绝大多数场合下，可以采取抽取方法的重构手法来把方法变小。
- 每当需要以注释来说明点什么时，就可以把需要说明的东西写进一个独立方法中。
- 以其用途（而非实现方法）对抽取的方法命名。

说明：

- 对于现代开发工具，方法调用增多不会影响性能，但长方法难以理解。

发散式变化

症状：

- 某个类因为不同的原因在不同的方向上发生变化

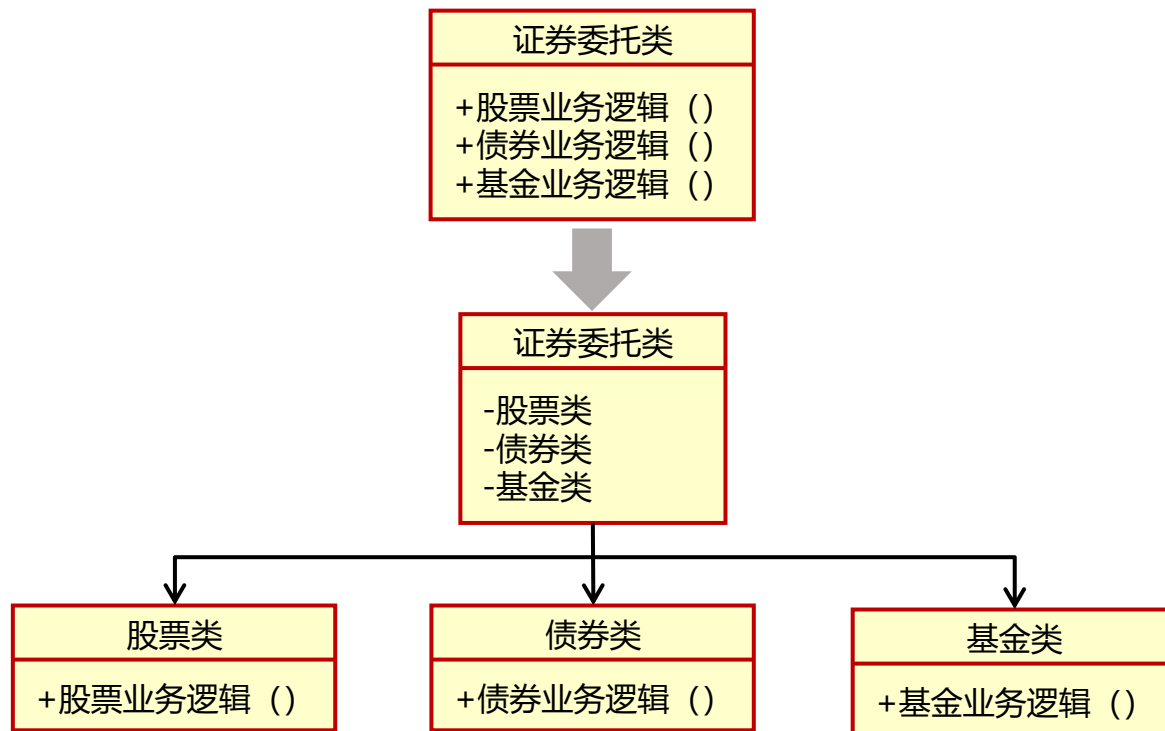
措施：

- 如果类既要找到对象，又要对其做某些处理，则让调用者查找对象，并将该对象传入，或者让类返回调用者所用的值。
- 采用抽取类的方法，为不同的决策抽取不同的类。
- 如果多个类共享相同类型的决策，则可以合并这些新类。至少这些类可以构成一层。

说明：

- 发散式变化指的是“一个类受多个外界变化的影响”，其基本思想是把相对不变的和相对变化相隔离，即封装变化。

发散式变化



霰弹式修改

症状：

- 发生一次改变时，需要修改多个类的多个地方

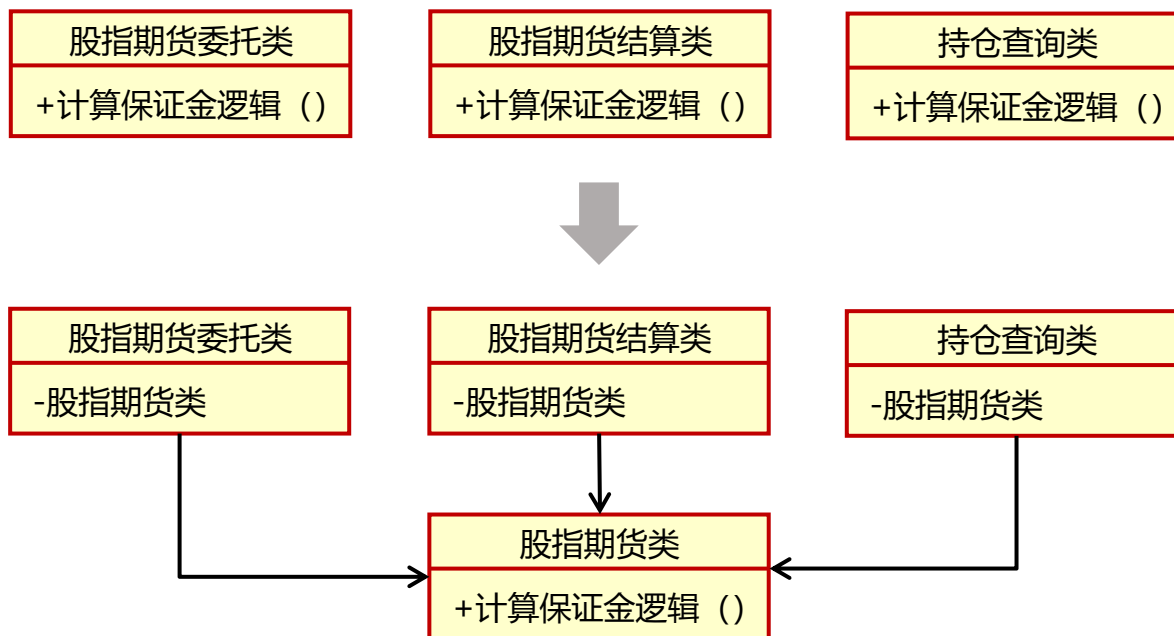
措施：

- 找出一个应对这些修改负责的类，这可能是一个现有的类，也可能需要通过抽取类来创建一个新类。
- 使用移动字段和移动方法等措施将功能置于所选的类中，如果未选中类足够简单，则可以使用内联类将该类除去。

说明：

- 散弹式修改指的是“一种变化引发多个类的修改”，其基本思想是将变化率和变化内容相似的状态和行为放在同一个类中，即集中变化。

霰弹式修改



数据泥团

症状：

- 同样的两至三项数据频繁地一起出现在类和参数表中。
- 代码声明了某些字段，并声明了处理这些字段的方法，然后又声明了更多的字段和更多的方法，如此继续。
- 各组字段名以类似的子串开头或结束。

措施：

- 如果项是类中的字段，则使用抽取类将其取至一个新类中；
- 如果值共同出现在方法的签名中，则使用引入参数对象的重构方法以抽取新对象；
- 查看这些项的使用：通常可以利用移动方法等重构技术，从而将这些使用移至新的对象中。

纯稚的数据类

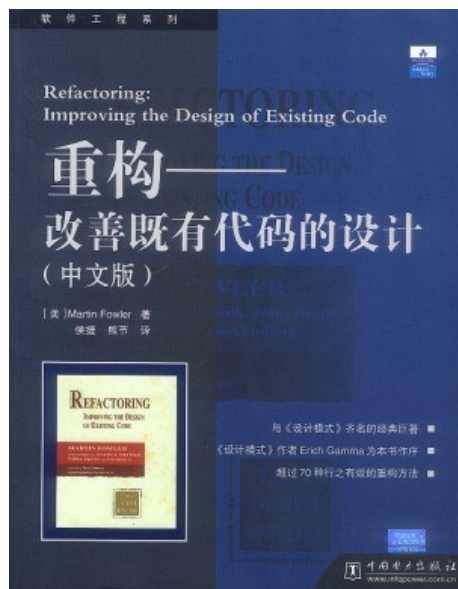
症状：

- 类仅有字段构成，或者只有简单的赋值方法和取值方法构成。

措施：

- 采用封装字段阻止对字段直接访问（仅允许通过赋值方法和取值方法）
- 对可能的方法尽量采用移除设置方法进行重构。
- 采用封装集合方法去除对所有集合类型字段的直接访问。
- 查看对象的各个客户，如果客户试图对数据做同样的工作，则对客户采用抽取方法，然后将方法移到该类中。
- 在完成上述工作后，可能发现类中存在多出相似的方法，使用相应的重构技术，以协调签名并消除重复。
- 对字段的大多数访问都不再需要，因为所移动的方法涵盖了其实际应用，因此可以使用隐藏方法来消除对赋值方法和取值方法的访问。

软件重构方法



- 重新组织函数
- 在对象之间移动特性
- 重新组织数据
- 简化条件表达式
- 简化函数调用
- 处理概括关系
- 大型重构

练习：抽取方法

下面的代码有什么不足？你认为应该如何重构？

```
double getPrice()
{
    int basePrice = _quantity * _itemPrice;
    double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

练习：方法内联化

下面的代码有什么不足？你认为应该如何重构？

```
int getRating()
{
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
```

```
boolean moreThanFiveLateDeliveries()
{
    return _numberOfLateDeliveries > 5;
}
```

练习：引入解释性变量

重构前

```
double price()
{
    //price is base Price - quantity Discount + shipping
    return _quantity*_itemPrice -
        Math.max(0,_quantity-500)*_itemPrice*0.05 +
        Math.min(_quantity*_itemPrice*0.1,100.0);
}
```

重构后

```
double price()
{
    final double basePrice = _quantity*_itemPrice;
    final double quantityDiscount =
        Math.max(0,_quantity-500)*_itemPrice*0.05;
    final double shipping = Math.min(basePrice*0.1,100.0);
    return basePrice-quantityDiscount+shipping;
}
```

练习：简化条件表达式

下面的代码有什么不足？你认为应该如何重构？

```
double getPayAmount()
{
    double result;

    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isRetired) result = retiredAmount();
            else result = normalPayAmount();
        }
    }
    return result;
}
```

练习：移动字段

下面的代码有什么不足？你认为应该如何重构？

```
class Account {  
    ...  
    private AccountType _type;  
    private double _interestRate;  
  
    double interestForAmount_days (double amount, int days) {  
        return _interestRate * amount * days / 365;  
    }  
    ...  
}
```

练习：移动方法

下面的代码有什么不足？你认为应该如何重构？

```
class Account...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result=10;
            if (_daysOverdrawn>7) result+=(_daysOverdrawn-7)*0.85;
            return result;
        }
        else return _daysOverdrawn*1.75;
    }
    double bankCharge() {
        double result=4.5;
        if(_daysOverdrawn>0) result+=overdraftCharge();
        return result;
    }
    private AccountType _type;
    private int _daysOverdrawn;
```

说明：假设有几种账户，每一种都有自己的“透支金额计算规则”。

练习：提炼类

下面的代码有什么不足？你认为应该如何重构？

```
Class Person {  
    private string _name;  
    private string _officeAreaCode;  
    private string _officeNumber;  
    public string getName() {  
        return _name;  
    }  
    public string getTelephoneNumber() {  
        return "("+_officeAreaCode+")"+_officeNumber; //格式为：(区号)电话号码  
    }  
    string getOfficeAreaCode() {  
        return _officeAreaCode;  
    }  
    void setOfficeAreaCode(string arg) {  
        _officeAreaCode=arg;  
    }  
    string getOfficeNumber() {  
        return _officeNumber;  
    }  
    void setOfficeNumber(string arg) {  
        _officeNumber=arg;  
    }  
}
```


Reference

- 清华大学国家级精品课程 《软件工程》 主讲人 刘强 副教授 刘璘 副教授
 - https://www.icourses.cn/sCourse/course_3016.html
 - https://www.xuetangx.com/course/THU08091000367/5883555?channel=learn_title
- Maurício Aniche, An Introduction to Software Architecture and Design, TUDelft, 2019



谢谢大家！

THANKS

