课程章节安排

- 第一部分 软件工程概论
- 第二部分 软件项目开发过程与管理
- 第三部分 软件需求工程
- 第四部分 软件设计
- 第五部分 软件编码、测试与质量保障
- 第六部分 软件实施、维护与演化

第五部分 软件编码、测试与质量保障

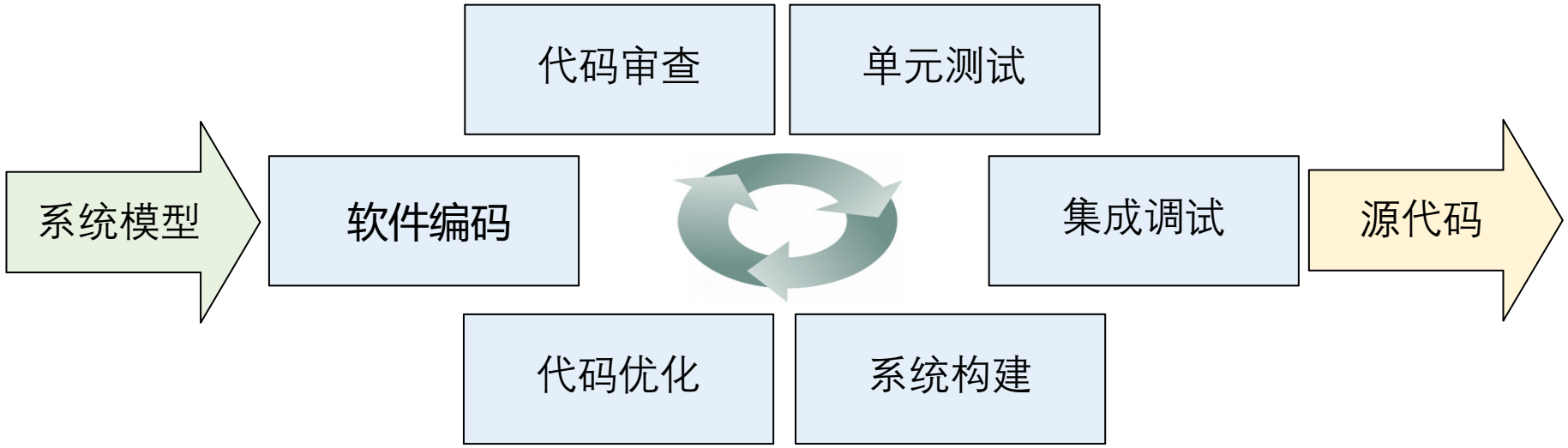
- 5.1 软件编程
- 5.2 软件测试
- 5.3 白盒测试
- 5.4 黑盒测试
- 5.5 变异测试
- 5.6 性能测试

5.1 软件编程

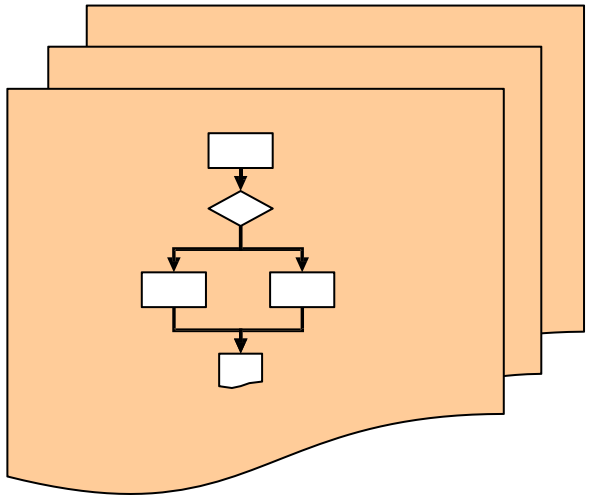
- 良好的编程实践
- 代码审查
- 代码重构

软件编程工作

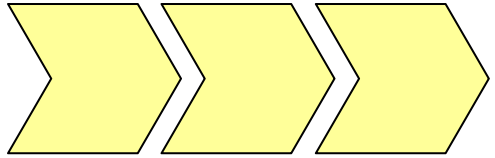
软件编程是一个复杂而迭代的过程，它不仅仅是编写代码，还应该包括代码审查、单元测试、代码优化、集成调试等一系列工作。

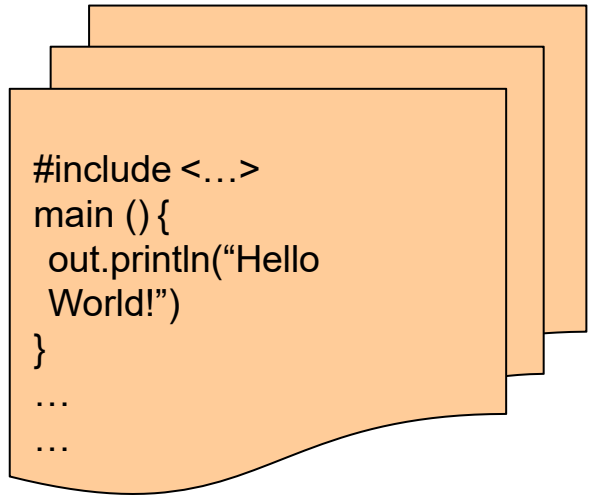


软件编程



使用程序设计语言
进行编码





软件设计描述(SDD) → 程序代码(Program Code)

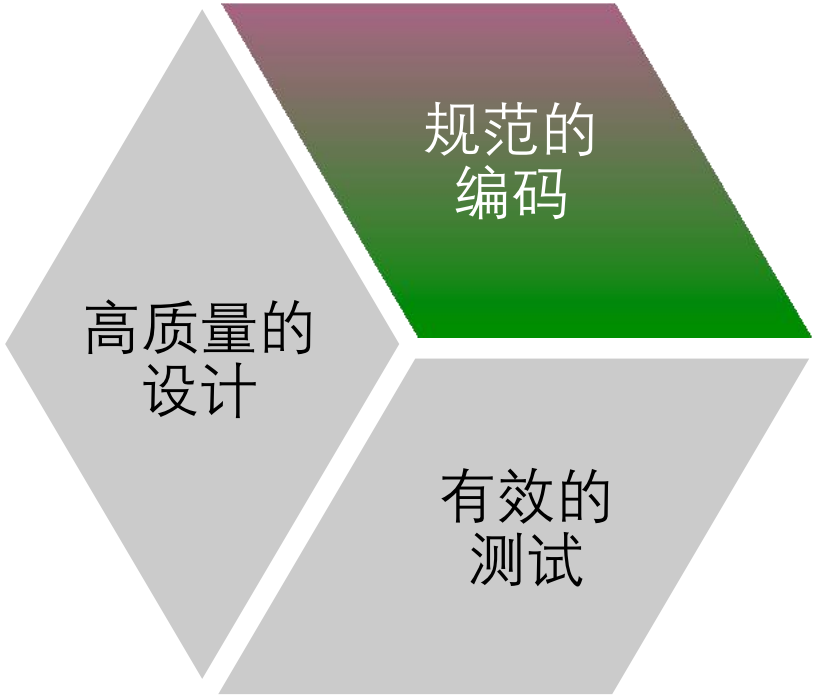
逻辑概念(不可执行) → 物理实体(可执行)

软件编码

- **软件编码是一个复杂而迭代的过程，包括程序设计(program design)和程序实现(program implementation)。**
- **软件编码要求**
  - 正确的理解用户需求和软件设计思想
  - 正确的根据设计模型进行程序设计
  - 正确而高效率的编写和测试源代码
- **软件编码是设计的继续，会影响软件质量和可维护性。**

软件编程工作

高质量软件
开发之道



软件编程规范的必要性

- **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。**

- **现实**
  - 软件全生命周期的 70% 成本是维护
  - 软件在其生命周期中很少由原编写人员进行维护
- **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

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？

说说你的做法。。。

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。

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：程序模版

```
# !/usr/bin/env python3
# -*- coding: utf-8 -*-
#
__author__ = 'YourName'


import sys


def main(argv):
    """Do something..."""
    return 0


if __name__ == '__main__':
    exit(main(sys.argv[1:]))
```

被直接执行的文件建议增加此行，被导入的模块请忽略文件的基本头部

模块的导入总应该放在文件顶部

主功能建议创建main函数来执行，函数、类等应遵循一定的注释规范

即使打算被直接执行的文件也应是可导入的，这样当模块被导入时主程序就不会被执行

良好的编程实践：注释

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：注释

- 形式1：由 # 开头的“真正的”注释，说明选择当前实现的原因以及这种实现的原理和难点；

```python
def gold_divide(n):
    # 黄金分割点比例为(√5---1)/2≈0.618034

    return n * 0.618 # 直接取0.618以加速
```

- 形式2：文档字符串，说明如何使用包、模块、类、函数（方法），甚至包括使用示例和单元测试。

```python
def gold_divide(n):
    """ Get gold divide value of n.

    Args:
        n: input number

    Returns:
        A float value
    """
    return n * 0.618
```

```
def main(argv):
    """
    函数main，返回0表示操作成功，返回-1表示失败
    """
    try:
        # 以二进制只读方式打开文件
        with open('cat.pic', 'rb') as fin:  # fin是输入文件
            # 将文件指针移动到文件尾
            if fin.seek(0, 2) != 400 * 400 * 3: # 判断文件尾位置是否等于400*400*3
                print("输入文件 cat.pic 不符合格式要求")    # 如果不等，显示错误信息
                return -1
            fin.seek(0) # 将文件指针移动到文件开头
            try:
                # 以二进制写方式打开文件
                with open('cat2.pic', 'wb') as fout:    # fout是输出文件
                    for i in range(400):    # 从图像的第1行到第400行循环
                        for j in range(400):    # 从每行的第1列到第400列循环
                            # 从输入文件中读入每像素的RGB值
                            b = ord(fin.read(1))
                            g = ord(fin.read(1))
                            r = ord(fin.read(1))
                            # 按照公式 Y=0.299R+0.587G+0.114B 计算灰度值
                            y = (299 * r + 587 * g + 114 * b) / 1000
                            fout.write(chr(y))  # 将计算出来的灰度值写到输出文件中去
            except IOError as e:
                print("打开文件 cat2.pic 时错误")  # 如果打开失败则显示错误信息
                return -1
    except IOError as e:
        print("打开文件 cat.pic 时错误")    # 如果打开失败则显示错误信息
        return -1
    return 0    # 返回0表示正确处理完毕
```

注释仅仅是语句的重复解释，没有任何价值

你明白这段代码的作用吗？

```
def main(argv):
    """
    主函数，返回0表示成功
    """
    try:
        with open('cat.pic', 'rb') as fin:
            if fin.seek(0, 2) != 400 * 400 * 3: # 判断文件长度是否符合格式要求
                print("输入文件 cat.pic 不符合格式要求")
                return -1
            fin.seek(0)
            try:
                with open('cat2.pic', 'wb') as fout:
                    # 下面是图像转换的算法实现。彩色图像到灰度图像的转换主要利用
                    # RGB色彩空间到YUV色彩空间的变换公式来取得灰度值Y，公式是
                    # Y = 0.299R + 0.587G + 0.114B
                    for i in range(400):
                        for j in range(400):
                            b = ord(fin.read(1))
                            g = ord(fin.read(1))
                            r = ord(fin.read(1))
                            y = (299 * r + 587 * g + 114 * b) / 1000
                            fout.write(chr(y))
            except IOError as e:
                print("打开文件 cat2.pic 时错误")
                return -1
    except IOError as e:
        print("打开文件 cat.pic 时错误")
        return -1
    return 0
```

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。

练习：编写一个产生素数的函数

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，永远不会存在舍入问题。

思考

在C中，写一个判断语句判断i是否等于7，你们怎么写？

```
if(7==i)
if(i==7)
```

软件编程规范

Google

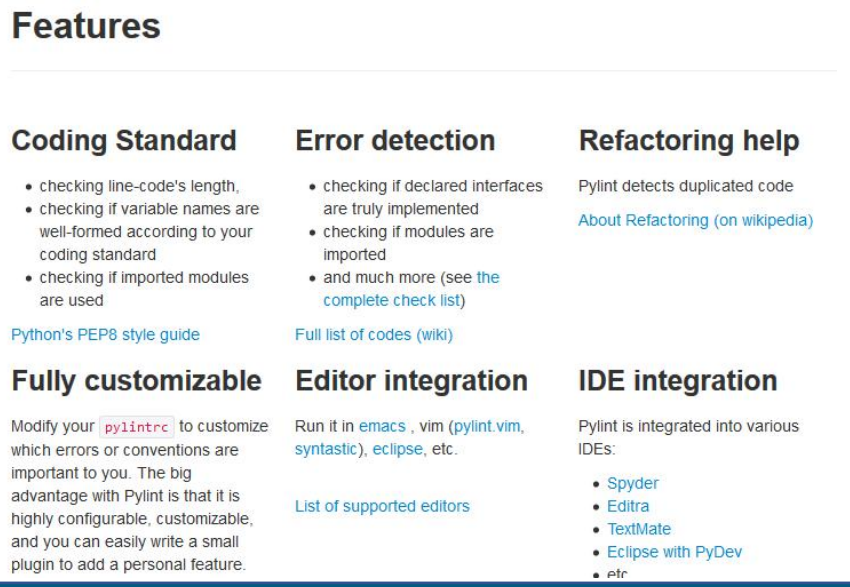
https://github.com/google/styleguide

Our C++ Style Guide, Objective-C Style Guide, Java Style Guide, Python Style Guide, Shell Style Guide, HTML/CSS Style Guide, JavaScript Style Guide, AngularJS Style Guide, Common Lisp Style Guide, and Vimscript Style Guide are now available. We have also released cpplint, a tool to assist with style guide compliance, and google-c-style.el, an Emacs settings file for Google style.

If your project requires that you create a new XML document format, our XML Document Format Style Guide may be helpful. In addition to actual style rules, it also contains advice on designing your own vs. adapting an existing format, on XML instance document formatting, and on elements vs. attributes.

Python代码分析工具

http://www.pylint.org/



Pylint 是一个 Python 代码分析工具，它用于分析 Python 代码的错误， 查找不符合代码风格标准（Pylint 默认使用的代码风格是 PEP 8）和有潜在问题的代码。

Python代码分析工具

Pylint功能

检查代码风格是否符合PEP8规范

检查代码是否存在常见的错误和违反最佳实践

http://pylint-messages.wikidot.com/all-messages

检查重复的代码

- C（Convention, 约定）
- R（Refactor, 重构）
- W（Warning, 警告）
- E （Error, 错误）

Pylint 执行结果

```
No config file found, using default configuration
************* Module simplecaeser
C:  1, 0: Missing module docstring (missing-docstring)
C:  4, 0: Invalid constant name "shift" (invalid-name)
C:  5, 0: Invalid constant name "choice" (invalid-name)
C:  6, 0: Invalid constant name "word" (invalid-name)
C:  7, 0: Invalid constant name "letters" (invalid-name)
C:  8, 0: Invalid constant name "encoded" (invalid-name)
```

错误类型：出现位置（行，列）：说明信息（问题标识）

```
Report
======
20 statements analysed.

Statistics by type
------------------
```

← 列出所查文件的模块、类、方法和函数的数量

type	number	old number	difference	%documented	%badname
module	1	1	=	0.00	0.00
class	0	0	=	0	0
method	0	0	=	0	0
function	0	0	=	0	0

Raw metrics

分析实际的代码、文档字符串、注释、空行等分别有多少行以及所占百分比

type	number	%	previous	difference
code	22	95.65	22	=
docstring	0	0.00	0	=
comment	1	4.35	1	=
empty	0	0.00	0	=

显示前一次结果，可以做对比

Duplication

显示是否有重复的行，由此可以判断是否有可以抽离出来的部分进行重构。

	now	previous	difference
nb duplicated lines	0	0	=
percent duplicated lines	0.000	0.000	=

Messages by category

对前面源代码分析信息进行汇总，分别列出R, W, E, C各是多少

type	number	previous	difference
convention	6	6	=
refactor	0	0	=
warning	0	0	=
error	0	0	=

Messages

根据问题标识进行汇总

message id	occurrences
invalid-name	5
missing-docstring	1

Global evaluation

```
Your code has been rated at 7.00/10 (previous run: 7.00/10, +0.00)
```

对分析文件的一个评分，10分为满分

代码静态分析工具

- Checkstyle：通过对代码编码格式、命名约定、Javadoc、类设计等方面进行代码规范和风格的检查。
- FindBugs：通过检查类文件或JAR文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。
- PMD：通过其内置的编码规则对Java代码进行静态检查，主要包括对潜在的Bug、未使用的代码、重复的代码、循环体创建新对象等问题的检验。
- Jtest：能够按照其内置的超过800条的Java编码规范自动检查并纠正这些隐蔽且难以修复的编码错误，同时还支持用户自定义编码规则，帮助用户预防一些特殊用法的错误。

代码静态分析工具

- 微软Visual Studio中的代码分析工具可以检查代码中是否存在一些常见缺陷和违反良好编程习惯的情况。
- Lint是一种静态程序分析工具，目前已形成了一系列工具。它侧重于代码的逻辑分析，发现代码中一些潜在的错误，如数组访问越界、内存泄漏、使用未初始化变量等。
- JSHint（www.jshint.com）是一款JavaScript代码验证工具，用于检测代码中的错误和潜在问题。
- CSSLint（csslint.net）是一款CSS代码检查工具，可以进行基本语法检查以及使用一套预设的规则来检查代码中的问题。
- HTMLHint（htmlhint.com）是一款基于JS开发的静态扫描组件，支持所有浏览器和Nodejs平台，可集成到IDE环境或编译系统中。

5.1 软件编程

- 良好的编程实践
- 代码审查
- 代码重构

代码审查

代码审查（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，它通过阅读代码来检查源代码与编码规范的符合性以及代码的质量。

代码审查的作用

- 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”

代码审查

非正式 ← → 正式

桌面检查　　代码走查　　代码审查

由一个程序员人工阅读代码，通过对源程序代码的分析和检验来发现程序中的错误。

设计或编程人员组成一个走查小组，通过阅读一段文档或代码并进行提问和讨论，发现可能存在的问题。

若干编程人员和测试人员组成一个审查小组，以会议形式通过阅读、讨论和争议对程序进行静态分析。

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

5.1 软件编程

- 良好的编程实践
- 代码审查
- 代码重构

代码重构

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。

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

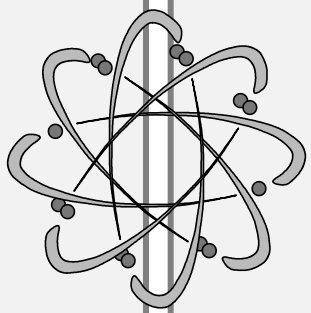
重构

- 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
过长的方法
过大的类
过长的参数列表
发散式变化
霰弹式修改
依恋情结
数据泥团
基本类型偏执
Switch语句
平行继承关系



多余的类
不确定的一般性
临时字段
消息链
二传手
过度亲密
异曲同工的类
不完整的库类
纯稚的数据类
拒收的馈赠
过多的注释

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

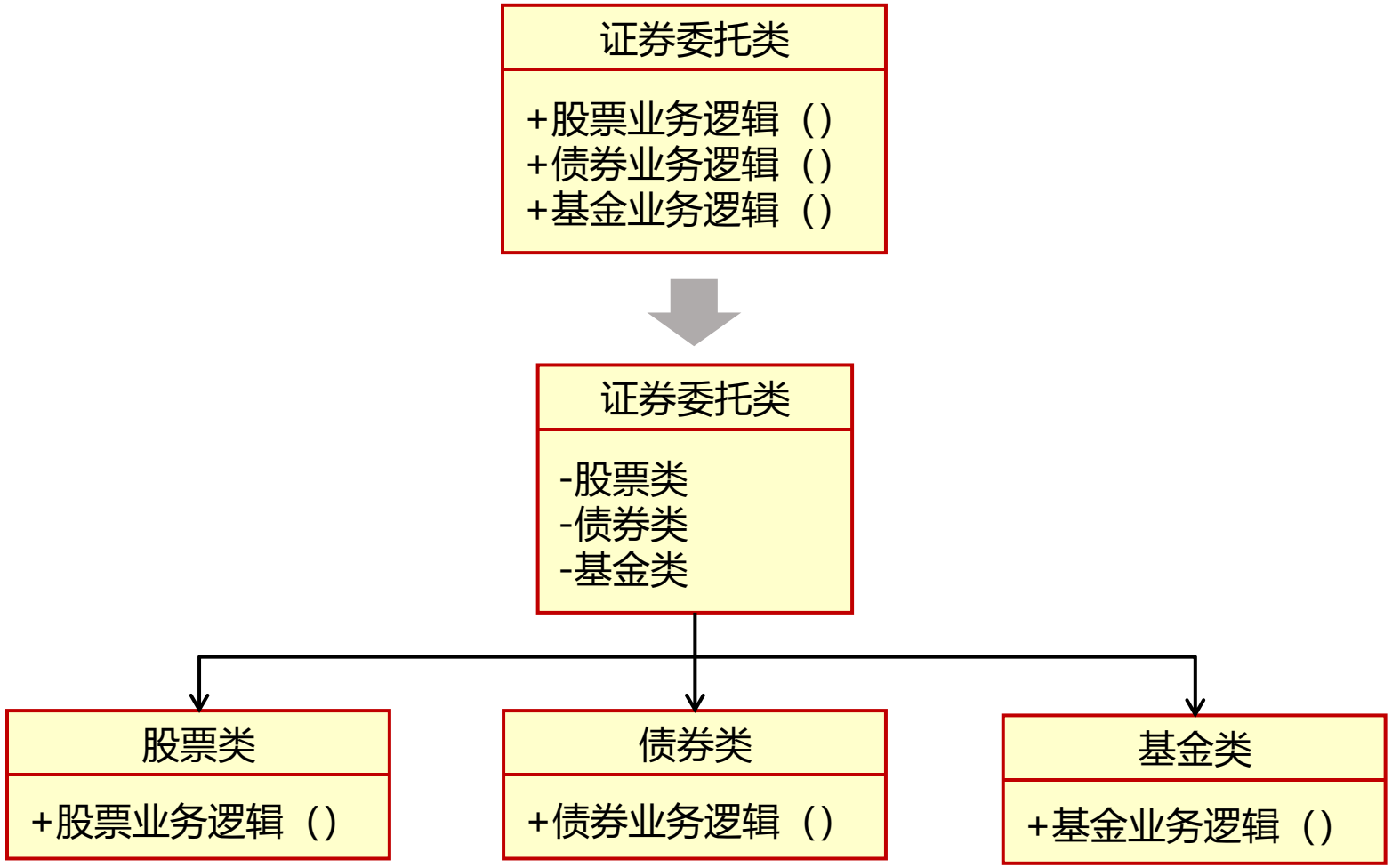
措施：

- 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


证券委托类
+股票业务逻辑（）
+债券业务逻辑（）
+基金业务逻辑（）
证券委托类
-股票类
-债券类
-基金类
股票类
+股票业务逻辑（）
债券类
+债券业务逻辑（）
基金类
+基金业务逻辑（）

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

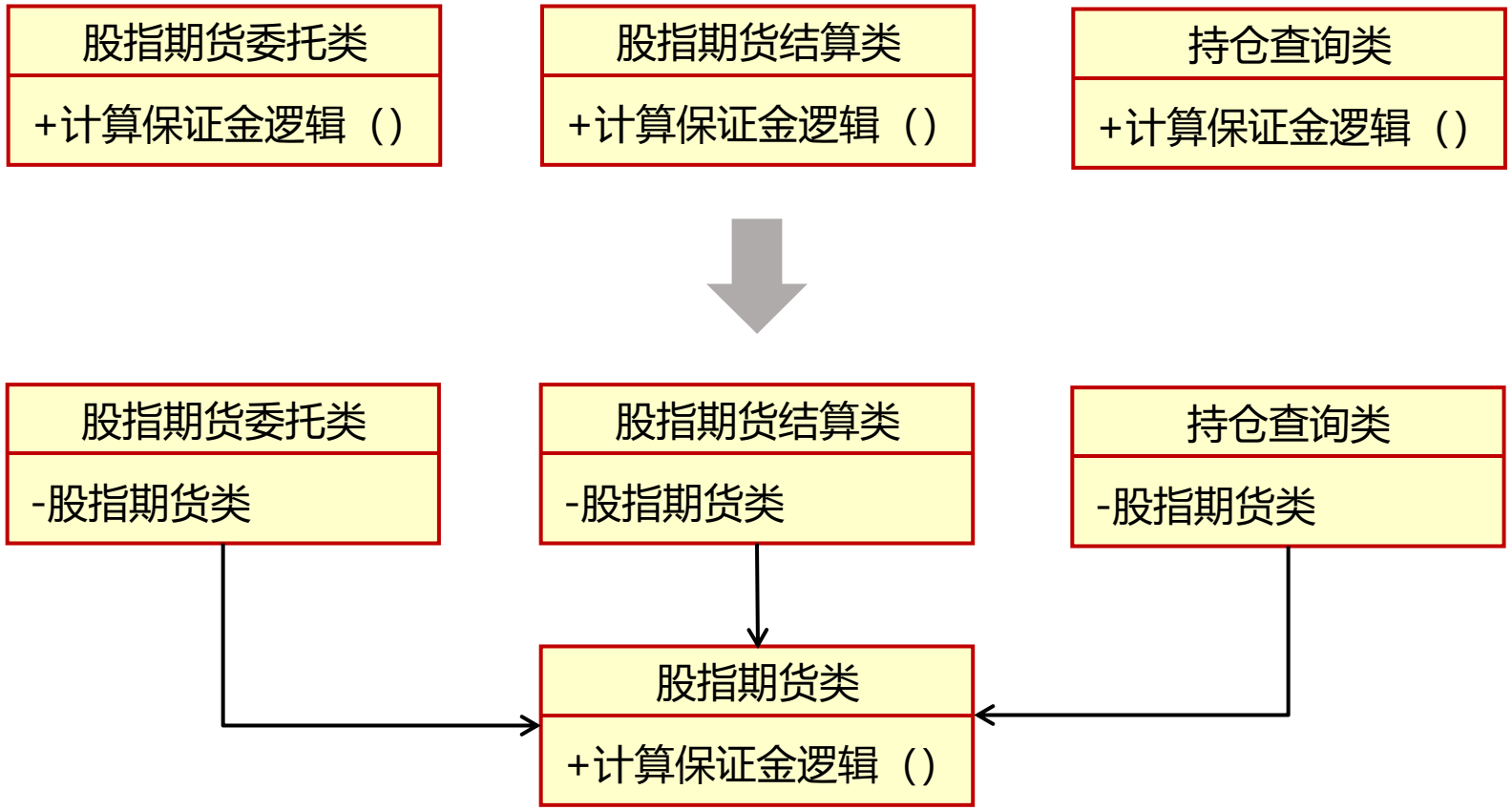
措施：

- 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


股指期货委托类
+计算保证金逻辑（）
股指期货结算类
+计算保证金逻辑（）
持仓查询类
+计算保证金逻辑（）
股指期货委托类
-股指期货类
股指期货结算类
-股指期货类
持仓查询类
-股指期货类
股指期货类
+计算保证金逻辑（）

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处相似的方法，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。

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 = _quatity * _itemPrice;
    double discountFactor;

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

    return basePrice * discountFactor;
}
```

练习：抽取方法

重构结果

```
double getPrice(){
    return basePrice () *discountFactor();
}

int basePrice(){
     return _quatity * _itemPrice;
}

Double discountFactor(){
       if (basePrice()>1000)  return 0.95;
       else return 0.98;
}
```

折扣这个方法可以单独分开

练习：引入解释性变量

下面的代码有什么不足？你认为应该如何重构？

```
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);
}
```

练习：引入解释性变量

重构结果

```
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：简化条件表达式

重构结果

```
double getPayAmount(){
if (_isDead) return deadAmount();
if (_isSeparated) return separateAmount();
if (_isRetired) return retiredAmount();
Return normalPayAmount();
}
```