COMP130015.02

软件工程

4. 高质量编码

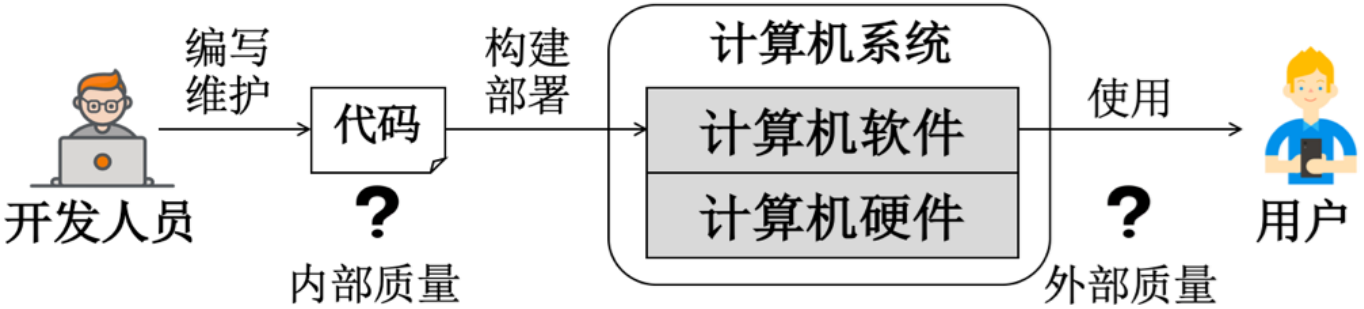
复旦大学计算机科学技术学院

沈立炜

shenliwei@fudan.edu.cn

代码质量

- 外部质量：面向用户以及软件运行和使用
  - ✓ 逻辑严密、尽量杜绝缺陷和漏洞
  - ✓ 具有良好的性能、可靠性和安全性
- 内部质量：面向开发者及持续演化和维护
  - ✓ 代码应当容易理解、修改和扩展



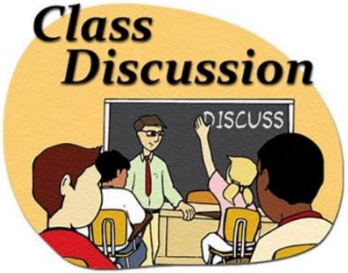
开发人员不仅仅是代码的“编写者”而且还是代码的“使用者”：在长期的软件生存周期内，代码需要持续进行维护和扩展，开发人员经常需要阅读并理解代码并在其基础上进行修改或实现新功能

整洁代码（Clean Code）

- 软件工程师应当具有质量意识，而这种意识的培养需要从严谨、认真写好每一行代码开始
  - ✓ 好的代码不仅逻辑严密、不容易出问题，而且读起来让人感觉干净、整洁、易懂
  - ✓ 不好的代码除了运行起来问题层出不穷之外，读起来可能让人感觉晦涩难懂、像一团乱麻
- 很多企业都将“Clean Code”（“整洁代码”）作为软件工程能力提升所追求的目标

案例：低质量的代码（图书借阅功能）

```
1   public void borrowBook(String id1, String id2) {
2       if (! id1.equals(""))) {
3           if (checkBooks(id1) < 5) {
4               if (!(hasOverdueBooks(id1))) {
5                   if (! id2.equals(""))) {
6                   if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                       updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                   throw new IllegalArgumentException("This book can't be borrowed");
9                   } else
10                      throw new IllegalArgumentException("Book id can't be empty");
11              }
12              else
13                  throw new IllegalArgumentException("Overdue books exist");
14          } else
15              throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16      } else
17          throw new IllegalArgumentException("Student id can't be empty");
18  }
```



这段代码有哪些问题？

案例：低质量的代码（图书借阅功能）

```
1   public void borrowBook(String id1, String id2) {
2       if (! id1.equals(""))) {
3           if (checkBooks(id1) < 5) {
4               if (!(hasOverdueBooks(id1))) {
5                   if (! id2.equals(""))) {
6                   if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                       updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                   throw new IllegalArgumentException("This book can't be borrowed");
9                   } else
10                      throw new IllegalArgumentException("Book id can't be empty");
11              }
12              else
13                  throw new IllegalArgumentException("Overdue books exist");
14          } else
15              throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16      } else
17          throw new IllegalArgumentException("Student id can't be empty");
18  }
```

标识符命名不规范，难以理解其含义

未考虑id1、id2为null的情况，存在缺陷

“魔法”数字

五层条件语句嵌套，逻辑难以理解

语句缩进排版不规范，容易造成误解

缺少else，存在缺陷

if条件第二部分少了“非”操作，同时两个更新操作没有整体加括号

异常抛出不当，需要使用自定义业务异常

案例：改进后的代码（图书借阅功能）

自定义业务异常类并根据具体情况建立异常类继承层次

```
1       public void borrowBook(String studentID, String bookID) throw ServiceException
    {
2           if (studentID == null || "".equals(studentID))
3               throw new StudentIDEmptyException();
4           if (getBorrowedBookCount(studentID) >= BOOK_BORROW_LIMIT)
5               throw new BookBorrowExeedLimitException();
6           if (hasOverdueBooks(studentID))
7               throw new OverdueBooksException();
8           if (bookID == null || "".equals(bookID))
9               throw new BookIDEmptyException();
10          if (getBook(bookID).isAllOut() || getBook(bookID).blocked())
11              throw new BookUnavailableException();
12          updateStudentStatus(studentID, bookID);
13          updateBookStatus(bookID, studentID);
14      }
```

标识符规范命名，含义自解释

对可能为null的参数进行判断，消除潜在缺陷

通过常量定义消除了魔法数字

通过针对各种规则检查的异常抛出消除了深层嵌套的条件语句

代码结构变清晰之后各种缺陷也更容易被发现和纠正

案例：进一步改进（图书借阅功能）

通过通用方法notEmpty判断字符串是否为空（如果为null或空字符串则抛出异常）

```
1     public void borrowBook(String studentID, String bookID) throw ServiceException {
2         notEmpty(studentID, "Student ID can't be empty");
3         notEmpty(bookID, "Book ID can't be empty");
4
5         checkStudentCanBorrowBook(studentID);
6         checkBookCanBeBorrowed(bookID);
7         updateBorrowingStatus(studentID, bookID);
…         ……
```

通过专门的check方法和update方法分别执行相应的条件检查和最终的借书信息更新

整个代码逻辑非常清楚，代码本身的标识符（参数名、方法名）基本都是自解释的

代码质量要求

- 可理解性（understandability）
- 可维护性（maintainability）
- 可靠性（reliability）
- 信息安全性（security）
- 高效性（high-efficiency）
- 可移植性（portability ）

代码质量除了功能逻辑正确、不包含功能缺陷外，还需要满足容易理解和维护、确保系统可靠性和信息安全性、能够高效运行、容易在不同环境之间移植等多个方面的要求

可理解性和可维护性-1

- 写好的代码经常需要阅读并进行维护
  - ✓ 代码由于各种原因（比如需求变化、环境变化、缺陷），需要被开发人员不时地修改维护
  - ✓ 开发人员经常需要理解然后修改他人所编写的代码
  - ✓ 即使是自己所写的代码经常也需要回忆甚至重新理解
- 编程语言的发展在一定程度上也是为了提高程序的可读性和可理解性
  - ✓ 汇编语言让开发人员从一大串的0、1编码中脱离出来，可以更好地理解程序如何操作计算机的寄存器、内存
  - ✓ C语言让开发人员不用纠结于如何使用寄存器这样的底层硬件，可以更好地理解程序的逻辑和算法
  - ✓ Java、Go、Python等语言将开发人员从复杂的内存等资源管理工作中解放出来，从而聚焦于业务逻辑的开发

从一定的意义上说，软件代码写出来是给人看的，只是顺便作为机器执行之用

可理解性和可维护性-2

- 代码的可理解性：又称代码的可读性，是指代码能易于阅读和理解的程度
  - ✓ 具有良好的可理解性的代码更容易被开发人员理解，使他们能够快速阅读并正确理解代码的逻辑
  - ✓ 代码理解的范围并不局限于某个局部（例如某个文件或方法），而是可能涉及整个软件系统（与设计相关）
- 代码的可维护性：指软件代码易于修改、扩展和复用的程度
  - ✓ 修改：由于需求变化、错误修复、性能改进、设计或代码质量提升等原因而对代码进行的改动
  - ✓ 扩展：在原有软件功能和能力基础上扩展新的功能和能力
  - ✓ 复用：代码被重复使用（可能需要修改）实现新的需求

代码的可理解性是可维护性的基础，只有理解了代码才有可能正确做出并实施修改决策

可理解性和可维护性-3

- 代码的可理解性和可维护性通常通过代码逻辑和代码风格来提升
  - ✓ 代码风格：采用统一的命名规范、符合逻辑结构的排版以及适当的注释等
  - ✓ 代码逻辑：降低代码的复杂度、对代码的复杂性进行封装、优化代码的设计结构等

课堂讨论：代码的可理解性和可维护性

课堂讨论：吐槽一下，你遇到过哪些难懂的、令人迷惑的、难改的代码？

可靠性和信息安全性-1

- 软件的可靠性
  - ✓ 通常是指软件在给定的时间区间和环境条件下，按设计要求正常运行的概率
  - ✓ 系统必须具备预防错误、容错、故障恢复（自愈）等方面的能力
  - ✓ 主要关注于应对系统内部和外部的一些异常情况（例如偶然的硬件失效、用户错误输入等）
- 代码的信息安全性
  - ✓ 软件对恶意威胁（如未授权访问/使用、泄露、破坏、篡改、毁灭）的防护能力，从而保证系统信息的机密性、完整性和可用性
  - ✓ 主要关注于应对外部的安全威胁（例如黑客的恶意攻击）

首先取决于整体的体系结构设计，

但同时也与各个部分具体的代码编写方式和代码质量密切相关

可靠性和信息安全性-2

- 对于高质量编码的要求
  - ✓ 代码运行本身不容易出错：开发人员在根据给定的需求编写代码时，需要以一种负责任的态度，认真思考所需解决问题的逻辑，并在代码中将这种逻辑完整地表达出来，来确保代码不容易出错
  - ✓ 代码的正常运行不容易被外部影响或破坏：高质量的代码必须通过一些额外的检查逻辑或者采用特定的写法，来确保在出现异常的输入或恶意的访问时，软件仍然正常运行或不产生错误

可靠性和信息安全性对于高质量编码的要求很大程度上是相通的，只是可靠性主要关注于应对系统内部和外部的一些异常情况，而安全性则主要关注于应对外部的安全威胁

课堂讨论：代码的可靠性和信息安全性

课堂讨论：你写过或见过哪些存在可靠性缺陷或安全漏洞的代码，存在哪些隐患？

高效性-1

- 软件的高效性：软件的运行效率高，能够在运行中表现出较高的性能
- 不仅取决于编码质量，也与软件的体系结构设计相关（如组件间的通信协议）
- 从高质量编码的角度看
  - ✓ 性能问题主要体现为代码运行过程中所花费的计算时间以及所占用的存储空间
  - ✓ 主要取决于代码对于系统资源（CPU、内存和网络等）的利用效率

高效性-2

- 代码的性能主要取决于两个方面
  - ✓ 算法的选择，例如
    - 不同的排序算法（例如冒泡排序、快速排序）的时间和空间复杂度各不相同，性能表现上也不同
  - ✓ 资源使用策略的选择，例如
    - 安卓应用中的长时间操作（例如后台下载）如果放在界面主线程中实现那么可能造成界面卡顿的问题
    - 循环语句中所涉及的磁盘或数据库访问如果能提取到循环以外执行将提高代码性能
- 要对资源使用的成本效益有着全面的了解，并慎重作出相关的实现决策，例如
  - ✓ 将一些属性在很多对象间复制，如果软件运行时大量创建对象那么会占据很多内存从而影响性能

高效性-3

- 编写代码时要对性能问题保持敏感，特别是一些涉及大量磁盘读写、数据库访问、网络传输等耗时的操作
- 同时注意软件工程的权衡取舍问题
  - ✓ 代码的性能问题包含时间和空间性能两方面
  - ✓ 代码的性能问题与代码的可理解性与可维护性等其他方面存在一定冲突
  - ✓ 对于性能敏感的软件使用场景，追求代码性能是第一位的，哪怕对代码的可理解性和可维护性有所影响
  - ✓ 对于性能一定程度上不太敏感的软件使用场景，需要更关注代码的可理解性和可维护性的提升

课堂讨论：代码的高效性

课堂讨论：你写过或见过哪些存在运行性能和效率问题的程序?

可移植性-1

- 软件的运行环境也经常会不断变化
  - ✓ 计算机和网络技术一直处于快速的发展之中
  - ✓ 软件企业以及客户企业的商业策略也在不断变化之中，例如商业与开源、自建机房与云平台
- 软件运行环境变化示例
  - ✓ 某个通用软件产品的客户中有一些希望在Linux上部署和运行，而其他客户则希望使用Windows操作系统
  - ✓ 一些客户会要求将软件迁移到国产软硬件体系结构上运行
  - ✓ 随着云计算技术的发展，越来越多的客户可能会要求软件能够迁移到公有云或私有云平台上运行
- 为了适应运行环境的变化，相应的软件代码可能会需要做一些修改

可移植性-2

- 可移植性：代码能在多大程度上适应不同运行环境，需要多少修改代价才能实现迁移
- 很大程度上取决于代码对特定平台的技术依赖性，包括编程语言、API、指令集等
- 可移植性也需要与其他代码质量属性一起进行综合权衡
  - ✓ 为了提高性能或者实现功能需要，一个Java程序可能使用本地化（native）接口，即使为此带来可移植性问题
  - ✓ 通常，一个程序通常应该尽量避免依赖于特定软硬件平台的API、指令或其他特性

可移植性-3

- 例1：Java程序
  - ✓ 通过虚拟机（JVM）解释执行字节码的方式运行，屏蔽了不同软硬件之间的差异，实现了“一处编译，到处运行”，因此具有良好的可移植性
  - ✓ 如果Java程序中使用了本地化接口调用Windows系统的相关功能，那么这个程序无法在Linux环境中直接运行，其可移植性就降低了
- 例2：C++程序
  - ✓ 如果只使用C++标准库中提供的API，那么可移植性会比较好
  - ✓ 如果使用了MFC（即微软基础类库，其中包含一个微软的应用程序框架以及以C++类的形式封装的Windows API）那么可移植性就比较差了

课堂讨论：代码的可移植性

课堂讨论：你写过或见过哪些存在可移植性问题的程序？

代码风格

- 好的代码风格应该让代码具有自解释性
  - ✓ 标识符命名：使得包、类、方法、属性、变量等代码元素的含义能够不言自明
  - ✓ 代码排版：使得代码的逻辑结构能够直观呈现出来，例如代码排版能够直接体现循环和条件判断语句的嵌套结构
  - ✓ 代码注释：为代码的逻辑考虑和技术原理（特别是难以通过代码领会到的）提供简洁的说明
- 代码的一致性也很重要
  - ✓ 同一项目甚至同一产品系列采用统一的代码风格约定
  - ✓ 有利于企业内的开发人员建立相应的代码编写的习惯，方便所有人按照同样的约定阅读和理解代码

清晰第一，简洁为美，风格一致

标识符命名

- 代码标识符包括：包名、文件名、类名、方法/函数名、属性名、变量名、常量名等
- 选择什么样的标识符不会影响程序的执行，但会在很大程度上影响代码的理解
- 标识符命名的出发点
  - ✓ 清晰、准确地表达它所代表的功能、操作、实体或数据的含义或作用
  - ✓ 符合所约定的统一格式

含义不明的标识符命名和魔法数字

```
1   public List<int[]> getList() {
2       List<int[]> list = new ArrayList<int[]>();
3       for (int[] x: theList) {
4           if (x[0] == 4) {
5               list.add(x);
6           }
7       }
8       return list;
9   }
```

theList列表到底是什么意思?

这里的0和4代表什么含义？

魔法数字:代码中含义不明的数字或字符串

消除魔法数字以及改进后的标识符命名

这个方法的功能就是把所有这些有标记的单元格取出来

```
1       public List<Cell> getFlaggedCells() {
2           List<Cell> flaggedCells = new ArrayList<Cell>();
3           for (Cell cell: gameBoard) {
4               if (cell.isFlagged()) {
5                   flaggedCells.add(cell);
6               }
7           }
8           return flaggedCells;
9       }
```

游戏有一个游戏面板（gameBoard）

面板上面有很多个单元格（cell）

每个单元格都有可能会做上标记（flagged）

修改后的代码不需要任何的注释，具有极强的自解释能力，更易于理解

标识符命名的含义

- 标识符的命名应当有意义
  - ✓ 准确表达它所代表的业务和逻辑含义、不会造成误解
  - ✓ 除非是非常简单的临时变量，否则尽量避免使用缺少明确的含义的命名，例如“x”、“f1”、“string1”
  - ✓ 避免不必要的硬编码，不要使用无法理解的魔法数字，可以通过定义有意义的常量或方法、属性来消除魔法数字
- 标识符命名应当尽量做到自解释
  - ✓ 如果标识符能够自解释，那么就不必再写冗长的注释去解释每个方法、属性和变量的含义了
  - ✓ 副作用是标识符较长，例如“checkStudentCanBorrowBook”
  - ✓ IDE可以帮助开发人员补全完整的标识符名称
- 可以针对特定业务领域建立规范化的术语表，从而便于采用统一、规范的标识符命名

标识符命名的形式

- 最常用的标识符命名方式是驼峰命名法
  - ✓ 大驼峰：类名、接口名、注解、枚举类型等
  - ✓ 小驼峰：局部变量名、方法名、属性名等
- 布尔型变量
  - ✓ 建议用那些隐含了“真/假”含义的词（例如done、success、fileFound）
  - ✓ 同时避免用否定意义的词（因为if (! fileNotFound)要比if (fileFound)难懂

命名形式	形式特点	例子	常用情况
大驼峰	大写字母开头;多个单词时，每个单词首字母大写，其他字母小写	CourseOffered	类名、接口名、注解、枚举类型
小驼峰	小写字母开头;多个单词时，除了第一个单词全小写，其他每个单词首字母大写、其余字母小写	selectedCourse	类的属性名、局部变量名、方法名、方法参数
全大写	所有字母大写	MANDATORY、IN_PROGRESS	枚举值，静态常量

排版格式

- 良好的排版格式可以使得代码看上去干净、整洁，同时能够直观体现代码的逻辑结构
- 纵向（多行代码之间）代码排版
  - ✓ 应当体现代码内容的分组以及分组之间的分隔
  - ✓ 关系紧密的代码行应该挨在一起并与其他代码行分隔开
  - ✓ 类或文件中的方法或函数顺序反映其重要程度或抽象层次
- 横向（单行代码之内）代码排版
  - ✓ 每行代码不宜太长，以免增加代码理解的难度
  - ✓ 一行一般只放一条语句，从而使得代码更加清晰，一行多条语句很容易让人在阅读时漏掉其中的语句
  - ✓ 还需要考虑代码块缩进、单词或符号间的空格分隔、括号的位置等问题
  - ✓ 让每行代码显得不拥挤，并且逻辑紧密的部分靠近，逻辑松散的部分分开，同时整行不会过于松散零碎

针对Java程序的排版格式推荐

- 纵向排版建议
  - ✓ 按照下列顺序排列且各部分之间用空行分隔：版权声明信息、包名（package）、引用包和类（import）、顶层类或接口
  - ✓ 类或接口的内部声明建议按照下列顺序排列且各部分之间用空行分隔：类属性、静态初始化块、实例属性、实例初始化块、构造方法、其他方法
- 横向排版建议
  - ✓ 使用空格缩进，每层次缩进4个半角空格
  - ✓ 每行不超过一个语句
  - ✓ 一行代码不超过120个字符
  - ✓ 块注释的缩进应该与该注释的上下文相同

代码注释

- 注释是开发人员之间的一种交流沟通手段
- 注释的目的是给代码提供额外的说明
  - ✓ 特别是对于代码中一些深层次的原理性的知识，例如业务规则或设计思考
  - ✓ 还可以对一些容易被忽略的细节进行说明和提醒
- 注释并非多多益善
  - ✓ 希望通过良好的标识符命名、复杂性控制、单一职责原则等手段来提升代码的自解释性
  - ✓ 有意义的注释就需要能够表达代码本身无法表达的内容，而不仅仅是对代码逻辑的简单翻译
  - ✓ 例如，在一个变量赋值语句后增加“将变量x初始化为0”这样的注释就有些多余

高质量编码并不推荐用大量的注释来说明程序的逻辑，而是建议通过提升代码本身的自解释能力，尽量减少无谓的注释

代码注释的基本原则

- 注释应当使用团队最擅长、沟通效率最高的语言（例如中文）进行编写，并且应当在团队内统一
- 通过良好的标识符命名、复杂性控制、单一职责原则等手段来提升代码的自解释性，对于晦涩难懂的标识符命名和语句应该进行重构而不是添加注释
- 注释应当为阅读和理解代码提供额外的信息，而非简单翻译代码本身
- 在必要的情况下，可以明确文档生成规则，方便通过工具将注释导出为文档，从而保持代码与文档的一致性
- 修改代码后检查注释是否仍然是有效的，过时且错误的注释会影响对代码的理解，甚至误导开发人员

常见的有意义的注释类型

- 补充解释代码的意图
  - ✓ 有时候会出现即使代码逻辑结构清楚、标识符命名规范但代码的深层意图仍然难以被直观领会到
  - ✓ 注释可以用来补充描述这种代码无法表达的作用或意图
- 给出必要的警示或预告
  - ✓ 代码中存在一些有特殊目的和考虑的实现、可能存在问题的临时性解决方案、未完成的任务及计划未来进行的修改
  - ✓ 为了提醒其他人和自己注意相关问题并在此后开发过程中进行规避或解决，可以用注释给出必要的警示或预告
- 给出代码无法描述的附加信息
  - ✓ 一些其他的附加描述信息，例如版权声明或许可证信息
  - ✓ 除非与版权相关，否则不需要为了说明代码作者而添加“added by…”这样的标注，在现代化的软件开发环境中这些信息应当由版本管理系统记录

需要通过注释补充解释代码意图

```
1     while (! finished) {
2         try {
3             Thread.sleep(10);
4         }
5         catch (InterruptedException e) {
…             ……
…         }
…     }
```

为何需要在循环里面加上sleep(10)，即让当前线程睡眠10毫秒？

若有一个注释则会更好地表达作者当时的意图：sleep(10)是为了让运行进程有时间处理其他逻辑代码，否则单线程的程序会假死

不需要“将变量x初始化为0”、“循环访问list数组”这样对显而易见的代码逻辑进行“直译”的注释

我们应该考虑的是有哪些深层次的代码意图、决策考虑或其他思考需要通过注释传达给代码的阅读者

通过注释对代码中的特殊问题给出警示

```
1  //WARNING：本测试用例需要耗费大约15分钟时间，因此日常集成时不运行
2  @Ignore
3  @Test
4  public void testHighThroughput() {
5  //模拟十万个线程同时进行访问
6  }
```

这段用来测试程序吞吐量的代码需要耗费大量的资源，不能在日常提交时频繁地运行，因此程序员将这个测试方法关闭，警告开发人员不要随意开启这个测试方法，以免降低持续集成效率

通过注释对自承认技术债进行提示

```
1   //Notice：  下列数据库连接参数需要改为从外部配置文件中读取
2   String driver = "com.mysql.jdbc.Driver";
3   String url = "jdbc:mysql://localhost:3306/sqltestdb";
4   String user = "root";
5   String password = "123456";
6   Class.forName(driver);
7   con = DriverManager.getConnection(url,user,password);
```

自承认技术债（Self-admitted Technical Debt）：为了尽快完成当前交付任务而采取一些方便实现但存在问题的解决方案，例如仅实现了部分需求、采用了不好的设计和实现方式等，相当于暂时欠下了软件质量上的“债务”

开发人员自身已经意识到问题的存在，只是由于时间或技术上的限制暂时无法解决，因此通过注释的方式提示其他开发人员或自己未来加以解决

通过注释说明未完成的工作

```
1  public int[] sortNumbers() {
2      //TODO: 需在这里补充真实的排序代码
3      return new int[]{1,2,3,4,5};
4  }
```

当前方法直接返回一个数组是因为真实的排序代码还没有写

对于这些注释，现代代码编辑器一般能给出特殊的标识，从而辅助开发人员快速找到它们并且及时进行处理

需要注意的是，作为未完工的标志，TODO只能在开发阶段存在，在正式交付的代码中不允许有TODO注释

高质量编码对代码逻辑的要求

- 代码逻辑严密，尽量避免错误
- 代码逻辑清晰、易于理解和维护
  - ✓ 控制代码复杂度
  - ✓ 减少代码重复
  - ✓ 编写高质量的子程序

代码编写的基本要求

- 变量声明和初始化
  - ✓ 显式地声明变量同时控制变量作用域
  - ✓ 进行必要的初始化
  - ✓ 在靠近首次使用的地方声明并初始化一个变量
- 数据类型的选择和使用
  - ✓ 避免数据溢出
  - ✓ 小心浮点数比较
  - ✓ 避免除零问题
  - ✓ 避免使用魔法数字
- 代码控制结构
  - ✓ 分支语句：注意逻辑完备性、避免复杂的条件判断、避免在条件判断中进行变量赋值等其他操作
  - ✓ 循环语句：避免不恰当的循环语句

控制变量作用域

- 避免作用域（即作用范围）过大的变量，如全局变量
  - ✓ 优点：开发人员方便地在不同的地方直接使用变量
  - ✓ 缺点：作用范围大大超出了局部的代码单元（例如方法和类），难以准确理解其含义
  - ✓ 缺点：可以在很多地方被使用或修改，变量取值不易受控，容易带来副作用
- 考虑变量作用域需要平衡以上优缺点
- 一般而言，确保变量容易理解且对变量的修改进行控制以保障代码质量无疑更加重要

避免在一行中声明多个变量

```
1  public int countNumbers() {
2      int []values, num;
.. ...
.. }
```

num到底是一个整型变量还是整型数组变量？

这样的变量声明方式降低了代码的可读性

如果改为分两行分别声明两个变量则没有这个问题

进行必要的变量初始化

```
1  public class Demo {
2      private static int secretValue;
3      private static String name;
4      public static void main(String[] args) {
5          System.out.println(name + ":" + secretValue);
6      }
7  }
```

如果直接访问name的方法或属性，那么会产生空指针异常

并不会报错，将输出“null:0”

进行变量初始化能让代码逻辑更加清晰，而且不会受到编程语言本身变量初始化方式或随机性的影响

Java程序中的变量初始化习惯

- 方法局部变量一般在声明时或者首次使用时进行初始化
- 类的静态成员变量一般在声明的时候直接进行初始化
- 类的非静态成员变量则在构造方法中进行初始化

避免数据溢出

数值类型的变量都可能存在溢出的问题，例如：
如果编程语言用32个二进制位来表示有符号整数，那么可表示的数值范围为-2147483648到2147483647

```
1  int result = 2147483647;
2  System.out.println("result+1= " + (result+1));
```

该程序将输出“result+1= -2147483648”
出现问题的原因是变量的期望值已经超出了变量类型能够表示的最大值，即发生了上溢（overflow）

对于浮点数（如float）类型的变量，如果变量值过于接近0（但仍然大于0），那么可能由于精度不足而意外变为0，即发生了下溢（underflow）

避免直接使用等于操作进行浮点数比较

```
1       float v1 = 1.0f - 0.9f;
2       float v2 = 2.9f - 2.8f;
3       if (v1 == v2) {
4           System.out.println("equal");
5       }
6       else {
7           System.out.println("not equal");
8       }
9       System.out.println("v1: " + v1);
10      System.out.println("v2: " + v2);
```

程序运行结果：

not equal
v1: 0.100000024
v2: 0.10000014

受限于浮点数的内部表示精度，浮点数之间不能直接用等于操作来比较，而应该在一定的精度下去判断值相等

按照预定义的精度进行浮点数比较

```
1  final float epsilon= 1e-6f;
2  if (Math.abs(v1 - v2) < epsilon) {
3      // equal
4  }
5  else {
6      // not equal
7  }
```

如果两个浮点数的差的绝对值小于预定义精度(例如表示以元为单位的金额的变量精确到两位小数)，那么就可以认为它们是相等的

需要注意:受限于浮点数的精度，当需要进行精确计算时，不能使用浮点数

避免除零问题

```
1   int totalStudents = 2000;
2   java.util.Scanner scanner = new java.util.Scanner(System.in);
3   System.out.println("Please input the rooms:");
4   String inputLine = scanner.nextLine();
5   int rooms = 0;
6   try {
7       rooms = Integer.parseInt(inputLine);
8   }
9   catch (NumberFormatException e) {
10                  // …
11  }
12  int studentsPerRoom = totalStudents / rooms;
13  System.out.println("There are " + studentsPerRoom + " students in one
    room.");
```

rooms取值可能为0

一些代码静态检查工具可以扫描出潜在的除零问题
但解决这类问题或避免这类问题造成代码缺陷，还需要开发人员用一种负责任的态度增加一些逻辑代码来确保除数不会为零

避免使用魔法数字（magic number）

```
1   switch (cardState) {
2       case 1: // valid
3           // ...
4           break;
5       case 2: // freezed
6           // ...
7           break;
8       case 3: // expired
9           // ...
10          break;
11      default: ...
12  }
```

表示不同的校园卡状态

魔法数字：在代码中出现的具有特殊含义但又没有经过解释的数字
类似的还有“魔法字符串”

这些数字或字符串具有某种神秘未知的含义和作用，往往会给开发人员带来不小的困惑

魔法数字的出现往往隐含了一些重要的常量概念，而开发人员却没有对它们进行适当的抽象
主要问题：难以理解、难以同步更新、不同的人产生不同的理解

通过常量定义消除代码中的魔法数字

```
1  switch (cardState) {
2      case CARD_STATE_VALID:
3          // ...
4          break;
5      case CARD_STATE_FREEZED:
6          // ...
7          break;
8      case CARD_STATE_EXPIRED:
9          // ...
1          break;
0
1      default:
1
1  }
2
```

通过自解释的常量
定义消除魔法数字

这些常量可以放在一个文件中集中定义，并在所有需要的地方进行引用，不仅直观易懂而且可以统一维护

分支语句的逻辑完备性

- 分支语句需要全面考虑并准确表达所有可能发生的情况
- if-else语句：考虑整个分支语句嵌套结构是否覆盖了所有的可能性，是否存在逻辑漏洞，例如未覆盖的特殊情况
- switch-case语句：确定所有的case条件是否包含了所需要枚举的所有情况，并且尽量使用default子句提供缺省的处理逻辑

避免复杂的条件判断

```
1  if (isStudent) {
2      if (cardState == CARD_STATE_VALID) {
3          if (hasMoney) {
4              // pay ...
5              return payMoney(...);
6          }
7      }
8  }
9  return PAY_FAILED;
```

使用了三层嵌套的if条件进行逻辑判断，使得payMoney操作隐藏得比较深，不便于阅读和理解

这种复杂条件语句增加了代码的复杂度和理解的难度

通过特殊情况预处理简化嵌套if-else语句

```
1  if (!isStudent) {
2      return PAY_FAILED_NOT_STUDENT;
3  }
4  if (cardState != CARD_STATE_VALID) {
5      return PAY_FAILED_CARD_STATE;
6  }
7  if (!hasMoney) {
8      return PAY_FAILED_INSUFFICIENT_BALANCE;
9  }
10 return payMoney(...);
```

预先针对一些特殊情况进行处理，通过守卫条件对可能失败的情况进行预先判断

使得后续处理需要考虑的情况大大减少，因此可以有效减少复杂的嵌套逻辑判断，使得代码逻辑结构更清晰、更容易理解

避免在条件判断中进行变量赋值等操作

当用户是老师(即student变量取值为false)时if条件中获取余额的语句不会被执行，从而引起逻辑错误

```
1    int remainMoney = 0;
2    if (student && ((remainMoney = getRemainMoney()) > 0)) {
3        // student card
4        System.out.println("Student's remain money:" + remainMoney);
5    }
6    else {
7        // teacher card
8        System.out.println("Teacher's remain money:" + remainMoney);
9    }
```

条件语句应该执行纯粹的逻辑判断，而不应该执行赋值和计算等其他操作

否则将会使得代码逻辑变得更加复杂难懂，同时可能造成额外的副作用并引发各种问题

在循环语句头上用条件判断进行循环控制

```
1 for (int i = 0; i < MAX; ) {
2     // do something
3     if (boo[i++] == 0) break;
4 }
```

通过循环体中的break语句实现循环控制
虽然可以达到循环控制的目的，但不容易理解

```
1    for (int i = 0; i < MAX && boo[i] != 0; i ++) {
2        // do something
3    }
```

明确在循环语句头上用条件判断进行循环控制，因此更加容易理解

永真的外层循环结合循环体中的退出条件

```
1   while (true) {
2       // do something
3       if(conditionA) {
4           break;
5       }
6       // do something
7       if(conditionB) {
8           break;
9       }
10  }
```

如果遇到循环条件复杂且难以表达的情况，那么可以使用永真的外层循环，同时在循环体内部根据各种条件判断退出循环

避免空循环体的“死循环”

```
1 while(!isDone) {
2     // 循环体中没有任何语句
3 }
```

编写者希望通过循环不断检查某项处理是否完成(isDone为true)，然而程序可能并不能如愿执行

1)CPU会全功率运行，从而影响其他部分的运行速度

2)由于计算机指令调度等问题，对isDone进行赋值代码可能执行不到，导致循环无法退出，系统也会呈现“卡死”的状态

这种循环体为空的while“死循环”必须避免

如果要实现对某个状态参数的轮询判断，那么应该将while循环体中加上放弃CPU处理时间的语句(例如调用当前线程的sleep方法)，并且把这样的循环体放在独立的线程中运行

重复代码问题

- 代码复制粘贴是常见的软件复用手段
  - ✓ 来自软件开发问答网站、技术博客和教程的代码
  - ✓ 来自开源或企业软件项目、过去所开发的软件中的代码
  - ✓ 在不同模块中加入同样或相似的功能实现，例如权限检查、日志记录、数据库或文件访问等
- 不同粒度的重复代码：代码片段、方法或函数、类或文件、模块和项目
- 重复代码被称为代码克隆（Code Clone），一般都被认为是一种需要注意的问题
  - ✓ 增加了代码长度和复杂度
  - ✓ 增加了代码理解的负担
  - ✓ 带来额外的代码修改工作量和缺陷风险

重复代码（代码克隆）的危害

- 试想一段权限检查代码通过复制粘贴的方式出现在了几十个不同的模块之中
  - ✓ 阅读每个模块的代码时都需要重复理解这段代码的含义
  - ✓ 如果由于某种原因要修改权限检查方式那么开发人员需要将这些地方一一找出来修改，任何遗漏都有可能带来问题
  - ✓ 如果代码中包含缺陷或漏洞，那么代码复制粘贴也会将这些问题传播到其他地方
- 代码克隆被认为是良好软件设计的大敌以及头号的代码坏味道或异味（bad smell）
- 消除代码克隆的常用手段
  - ✓ 使用代码模板
  - ✓ 将重复代码提取为公共方法/函数
  - ✓ 建立继承层次并将重复代码提取到父类中

重复代码（克隆代码）

```
1   int lunchFee = getFeeAmount(stuNo, “lunch”);
2   if(lunchFee <= 0)
3       return “no unpaid lunch fee”;
4   int result = pay(lunchFee);
5   if(result == 0){
6       sendMessage(stuNo, “receive:” + lunchFee);
7       return “lunch fee paid successfully”;
8   }
9   else
10      return “lunch fee not paid”;
```

```
1   int textbookFee = getFeeAmount(stuNo, “textbook”);
2   if(textbookFee <= 0)
3       return “no unpaid textbook fee”;
4   int result = pay(textbookFee);
5   if(result == 0)
6       return “textbook fee paid successfully”;
7   else
8       return “textbook fee not paid”;
```

整体代码逻辑相似，局部参数和逻辑不同

根据学号查询学生所欠的两种不同类型的费用（午餐费和书本费）然后进行支付并返回结果，其中午餐费支付成功后还要发送一个消息

通过提取公共代码消除重复

```
1    String payFee(String stuNo, String feeType, boolean doSendMsg){
2        int fee = getFeeAmount(stuNo, feeType);
3        if(fee <= 0)
4            return “no unpaid ” + feeType + ” fee”;
5        int result = pay(fee);
6        if(result == 0) {
7            if(doSendMsg) {
8                sendMessage(stuNo, “receive:” + fee);
9            }
10           return feeType + “ fee paid successfully”;
11       }
12       else {
13           return feeType + “ fee not paid”;
14       }
15   }
```

```
payFee(stuNo, “lunch”, true);
payFee(stuNo, “textbook”, false);
```

将常量替换为参数
用条件语句控制差异化的语句和计算逻辑

通过参数抽取的方式将重复代码提取为公共方法或函数

通过不同的调用参数实现差异化的功能

树立软件复用的思想

- 尽量将通用功能实现为可复用的代码单元（例如类、方法/函数）甚至封装为可复用的开发库（library）及其API
- 利用各种软件开发框架来消除重复代码，例如
  - ✓ 使用O/R Mapping（实体/关系映射）框架（如myBATIS、Hibernate等）消除与数据库访问相关的重复代码
  - ✓ 使用日志框架（如Log4j、Logback等）消除与日志记录相关的重复代码
  - ✓ 使用Java开发框架Spring的注解（annotation）消除与权限检查、读取参数配置等相关的重复代码

并非一切重复代码都是可以消除的

- 避免教条主义，即认为一切重复代码都可以消除
- 牢记工程化的基本原则，权衡利弊，根据实际情况做出合理决策：综合权衡消除重复代码的难度与成本以及可能获得的收益
  - ✓ 有些重复代码很难消除，例如重复代码与不同的部分混在在一起并且难以通过参数化或代码模板的方式进行提炼
  - ✓ 有些重复代码并没有太大的危害，例如非常稳定不会发生修改、实现的是广为人知的通用逻辑（例如排序算法）等
- 即使有些重复代码通过权衡利弊决定保留，也还是要对其潜在危害加以了解和注意

代码复杂度问题

- 人类认知能力存在限制，对过于复杂的事物和问题总是感觉难以理解和掌握，同时也容易因为理解的偏差或不完整而犯错
- 代码的复杂度与长度以及逻辑组合的数量相关，与问题本身的复杂度也相关
- 控制代码复杂度的主要思路就是充分利用分解和抽象
  - ✓ 一个人需要同时关注的东西越多，就越容易犯错误
  - ✓ 如果一个人可以在同一时刻只关注问题的一个特定的部分，那么这个人所面临的认知复杂度就可以降低
  - ✓ 对复杂的代码进行拆分，还能便于针对不同的部分单独进行单元测试，从而提升代码的质量

过长的代码片段

- 过长的代码会带来心理压力和认知负担，因此控制代码长度是很有必要的
- 很多企业代码规范中都会限制文件/类以及函数/方法的最大长度
  - ✓ 例如一个类最长不超过200行、一个方法最长不超过50行
  - ✓ 这种限制并没有一个绝对的合理值，不同项目情况不一样
- 过长的类和方法一般都存在内聚不足的问题，即职责不单一、内部逻辑不够紧致、实现了多个关联不高的任务
- 控制代码长度的基本手段是分解：模块、文件、类，然后进一步分解为函数或方法

课堂讨论：代码长度

课堂讨论：大家写过的程序中最长的类或方法/函数有多长？有没有遇到过相关困难？感觉是否能改进？

合理的代码分拆

- 代码拆分应该以体现代码逻辑、符合人的理解习惯为前提
- 错误的拆分：为了达到代码复杂度指标要求，将方法按照代码行的顺序拆分为多个方法并按照“m1”、“m2”这样的顺序编号进行命名
- 合理的拆分：认真分析代码逻辑，将内部密切相关且与其他部分相对独立的代码抽取出来作为独立的方法，按照其功能和职责确定一个有意义的的方法名

复杂的代码嵌套结构

```
1   boolean treatment() {
2       if (condition) {
3           if (conditionA) {
4               // do something if A is true
5               if (conditionB) {
6                   // do something is B is true
7                   if (conditionC) {
8                       // do something is C is true
9                       return true;
10                  }
11                  else {
12                      // do something if C is false
13                      return false;
14                  }
15              }
16              else {
17                  // do something if B is false
18                  return false;
19              }
20          }
21          else {
22              // do something if A is false
23              return false;
24          }
25      }
26  }
```

在代码长度一定的情况下，复杂的嵌套条件分支或循环语句也是复杂性的一个来源，因为开发人员在理解这种语句时需要考虑复杂的条件组合

一般而言，超过3层的嵌套条件分支或循环语句对于大多数人而言已经有些难以理解了，如果不同层次上还会对一些条件或循环变量进行各种操作那么就更加复杂了

减少嵌套条件或循环语句层数的方法

- 合并条件
  - ✓ 通过“与”、“或”等逻辑操作将不同条件组合成复合条件，将原本嵌套的逻辑提升到同一层次上
  - ✓ 尽量考虑将嵌套条件语句改造成Switch/Case语句中并列的分支，这种多选一的并列选择不存在复杂的逻辑组合
- 按分支逻辑提取函数/方法
  - ✓ 包含深层嵌套的函数或方法经常也存在代码过长的问题，因此也可以通过分拆函数或方法的方式来缓解由于深层嵌套带来的复杂性问题
  - ✓ 通过对每一层循环或者嵌套分支抽取单一职责的函数或方法，可以将嵌套结构中的内层与外层代码拆分到不同的地方，从而降低嵌套层次
  - ✓ 通过这样的拆分，不但代码的嵌套层数降低而且原来的函数或方法代码也变短了

重构后的深层嵌套条件分支语句

```
1   treatment() {
2       if (condition)
3           return doConditionLevelA();
4       return false;
5   }
6
7   doConditionLevelA() {
8       if (conditionA) {
9           // do something if A is true
10          return doConditionLevelB();
11      }
12      else {
13          // do something if A is false
14          return false;
15      }
16  }
17
```

```
18  doConditionLevelB() {
19      if (conditionB) {
20          // do something if B is true
21          return doConditionLevelC();
22      }
23      else {
24          // do something if B is false
25          return false;
26      }
27  }
28
29  doConditionLevelC () {
30      if (conditionC) {
31          // do something if C is true
32          return true;
33      }
34      else {
35          return false;
36      }
37  }
```

如果各个判断条件的抽象层次不同，那么可以按照嵌套结构的层次进行这样的子程序抽取

开发人员每次在理解一个子程序时可以专注于某一层的逻辑，而不被复杂嵌套结构打断

高质量的子程序

- 子程序：表示一种功能和逻辑分解，可以是函数、过程或方法
- 采用子程序的重要原因
  - ✓ 通过分解降低程序对于人的认知复杂度
  - ✓ 减少代码重复以及由此造成的代码逻辑不一致问题
- 高质量子程序的几个方面
  - ✓ 控制子程序复杂度
  - ✓ 子程序参数的选择
  - ✓ 子程序返回值的选择

子程序的复杂度

- 子程序应当职责单一（ 只做一件事），体现功能内聚性
- 子程序命名需要准确体现职责并保持简洁，否则可能说明承担了多个职责
- 命令与查询分离
  - ✓ 命令：完成一些操作并返回结果，同时可能会改变状态（例如对象状态或数据库、文件中的数据）
  - ✓ 查询：仅仅执行所要求的查询并返回结果，不改变状态
  - ✓ 子程序要么是命令要么是查询，不应该同时兼具两种职责
- 简短的公共代码（例如三四行）也有必要抽取为子程序，便于理解和统一维护

子程序的参数

- 子程序应当没有或只有很少量的参数
  - ✓ 过多的参数要求增加了程序的复杂度和理解的难度
  - ✓ 如果某个参数本质上是表示类的状态的一部分，那么应该将其作为类的成员属性
  - ✓ 如果有大量的参数必须一起传递，那么应该考虑用一个类（或者结构体）把这些参数组织和封装起来
- 在不同的子程序中，类似参数的排列顺序要保持一致
- 不要在子程序中把参数用作工作变量
- 对参数的假设或约定应当加以明确（契约式设计）

子程序的返回值

- 子程序的返回值应当考虑让调用方得到一个明确结果并且不会导致误解
  - ✓ 例如，一个获取堆栈当前元素个数的方法如果想表明出现异常情况那么可以返回-1而不是0，因为0可能会是一个正常的返回值（当堆栈为空时）
- 如果一个子程序的返回值类型是数组或者列表
  - ✓ 当没有数据返回时返回一个长度为0的数组或列表而不是空值（null），确保数据类型的一致性
  - ✓ 在内部数据可信的前提下，调用子程序的代码就无需对返回值做空值判断，提升代码的执行效率

安全和可靠性编码

- 包含安全和可靠性在内的可信性已经成为软件（特别是关键性和基础设施软件）的基本要求
- 不仅与整体体系结构设计相关，而且也取决于每一处代码的严密思考和规范化实现
- 安全和可靠性编码包含一系列实践准则
  - ✓ 很重要的一条是防御式编程，即：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据
  - ✓ 为此，开发人员需要特别注意数据验证以及代码逻辑的严密性，同时合理使用错误处理、断言和异常处理
  - ✓ 此外，开发人员还需要了解一些容易出现安全性和可靠性问题的函数，并尽量使用不容易出错的安全函数

防御式编程

- 基本思想：子程序不应该由于传入错误的数据（由外部接口或其他子程序传入）而被破坏
- 程序员可以决定自己的程序逻辑但无法决定用户或其他子程序提供什么样的输入
- 但仍要竭尽所能保护自身程序的正确运行

防御性驾驶

预估风险
放眼远方
顾全大局
留有余地
引人注意

你永远不知道别人要做什么！

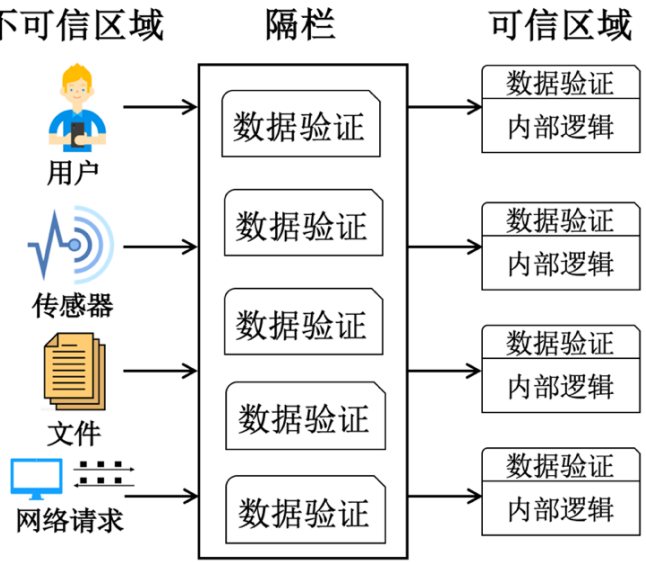
数据验证

- 开发人员可以决定自己的程序逻辑，但是无法决定用户或者其他程序的输入
- 例如，对于一个表示手机号的字符串参数
  - ✓ 传入的可能是空指针（null）或空字符串
  - ✓ 传入的可能是超长字符串、包含字母和其他非法字符（甚至恶意注入的引号等特殊字符）的字符串等
- 必须要对各种可能的输入有所考虑，并在代码中进行必要的数据验证
- 为了确定数据验证的方式和严格程度，需要首先明确程序的可信区域和不可信区域

程序的可信区域与不可信区域

不在开发人员的控制范围之内，可能是非法甚至恶意的输入

开发人员自己可以掌控



隔栏通过各种数据验证手段对来自外部的不可靠和“不干净”的数据进行处理，然后将符合要求的数据交给系统内部使用

可信区域内的程序无需直接面对不确定甚至恶意的外部输入

数据不确定性更高并且可能存在恶意性，因此相应的数据验证需要考虑的情况更多、验证要求也更高

数据一般不会存在恶意，出现问题要么属于可预期的异常情况要么源自于程序中的缺陷

SQL注入的检测和防护

- 使用外部数据构造的SQL语句所代表的数据库操作与预期不符，导致信息泄露或篡改
- 根本原因是使用外部数据直接拼接SQL语句

```
query.open("select id, name, phone from student where id ='" + studentID + "'")
```

如果外部输入的studentID取值为"' or '1'='1"，执行该语句将获取所有学生信息

- SQL注入防护措施
  - ✓ 参数化查询：在SQL语句中用参数代替外部输入值然后在执行前进行参数赋值，这是一种最有效的防护手段，但对SQL语句中的表名和字段名等不适用
  - ✓ 白名单校验：验证外部数据是否在允许的范围内，适用于拼接SQL语句中的表名和字段名
  - ✓ 特殊字符转义：对外部数据中的与SQL注入相关的特殊字符（例如单引号、注释符号等）进行转义

可信区域上的数据验证

- 内部数据验证需要沿着子程序调用链在各个层次上进行
- 这种方式可能带来性能和资源上的浪费，因为不同层次上可能存在重复验证
- 为了避免浪费并明确数据验证要求，应当明确子程序之间的契约，即调用方和被调用方分别应该满足的条件
- 例如，界面方法调用四则运算方法进行计算
  - ✓ 如果按照契约，界面方法需要除法运算时不能传出除数0，那么除数不为0的验证应当由界面方法来承担
  - ✓ 否则，除数不能为0的验证应当由四则运算方法来承担

代码逻辑问题：资源使用

- 文件流、数据库连接、网络连接等资源来自系统管理的各种资源池
- 一旦耗尽则相关操作（如文件访问、数据库访问、网络访问）都将无法实现了
- 因此，资源使用完毕之后应当确保安全关闭和释放所使用的资源

```
1    File file = new File(filePath);
2    InputStreamReader reader = new InputStreamReader(new FileInputStream(file));
3    BufferedReader bufferedReader = new BufferedReader(reader);
…    ...
…    bufferedReader.close();
```

关闭方式不安全：一旦代码发生异常，那么关闭语句将被跳过不被执行，从而导致资源泄漏

一般需要将关闭语句放入finally代码块以确保被执行或者采用try-with-resources语法糖（对于Java而言）

代码逻辑问题：判断条件和返回值检测

- 逻辑判断条件考虑上的不严密可能导致程序逻辑错误、循环无法终止等严重问题
  - ✓ 循环没有终止条件或终止条件不可达：可能导致循环在某些情况下无法终止，大量消耗资源甚至导致服务不可用
  - ✓ 不可达的条件分支：可能意味着分支条件存在逻辑上的错误，或者不可达分支没有意义可以去掉
  - ✓ 遗漏重要的条件分支：一些可能出现的情况没有考虑，一旦出现则可能发生逻辑错误
  - ✓ Switch语句缺少Default分支：即使确信所有的Case分支已经覆盖了所有情况，但理论上仍然可能存在例外情况
- 返回值检测
  - ✓ 对于有返回值的调用，一般都要对返回值进行读取和处理
  - ✓ 对于有可能为空（null）的返回值，要进行非空检测
  - ✓ 对于表示多种不同情况或执行结果的返回值，需要通过条件语句对返回值进行判断并根据情况提供不同的处理逻辑

错误处理

- 程序运行过程中可能出现的各种错误：功能调用失败、数据读取失败、超范围的非法值
- 适当的错误处理对于提升软件的可靠性和可维护性都有着重要的作用
  - ✓ 对于可靠性：通过识别错误并进行适当处理，程序可以恢复到正常状态从而确保系统可靠性
  - ✓ 对于可维护性：通过错误处理机制记录错误信息，开发人员可以更易定位并解决问题
- 常用的错误处理手段
  - ✓ 尝试正常服务：重试、使用中立值替换、使用与前次相同的值、换用最接近的合法值
  - ✓ 进行错误处理：将警告信息记录到日志文件中、返回错误码或显示出错消息、调用统一的错误处理子程序、关闭程序

课堂讨论：常用的错误处理手段

常用的错误处理手段

重试
使用中立值替换
使用与前次相同的值
换用最接近的合法值
将警告信息记录到日志文件中
返回错误码或显示出错消息
调用统一的错误处理子程序
关闭程序

课堂讨论：下列出错场景适合于使用何种错误处理手段?

1）系统启动时读取用户界面配色方案失败
2）ATM机读取顾客银行卡号失败
3）手机应用为用户提供基于位置的服务时读取当前位置失败
4）医疗仪器分析诊断时读取病人医学影像失败
5）医疗仪器在进行病人照射治疗时发现计算出来的照射量异常
6）飞行中的飞机发现速度检测信号不正常

错误处理决策的两种倾向

- 正确性（Correctness）：永远不要返回不正确的结果，哪怕不返回结果
  - ✓ 典型代表：人身安全攸关的软件
  - ✓ 例如，放射线治疗仪宁愿停止工作也不能按照一个错误的放射量对病人进行治疗
- 健壮性（Robustness）：保证软件可以持续运转，哪怕偶尔返回一些不够准确的结果
  - ✓ 典型代表：面向终端消费者的软件
  - ✓ 例如，用户在玩游戏时对于软件因为读取错误而缺少部分内容或出现卡顿有一定的容忍度，但如果经常突然关闭甚至造成数据丢失那么就无法接受了

软件整体错误处理机制的设计

- 需要制定统一的错误处理策略以协调不同层次上的子程序在错误处理过程中的角色和任务，例如
  - ✓ 底层的子程序负责检测和发现错误并返回错误码
  - ✓ 中间层的子程序负责进行进一步的处理，例如决定取什么值或采取什么备用方案
  - ✓ 高层的子程序负责通过界面展示错误信息及处理情况
- 对于子程序返回的错误信息（例如错误码、错误提示等）需要进行检查和处理
  - ✓ 根据下层子程序的错误信息执行相应的处理
  - ✓ 错误信息中可能包含涉及安全的敏感信息（例如内部结构和行为信息），在进一步处理和展示之前可能需要进行特殊处理以确保信息安全

断言

- 开发人员对于自己所编写的程序的执行逻辑都有所掌握，因此对于程序运行到某些特定位置时的状态都会有相应的判断和假设
- 如果程序在运行时可以对这些假设进行检测，那么就可以及时发现程序运行逻辑与预期不一致的地方，从而更快发现并定位问题
- 断言（assertion）就是支持这种检测的机制
  - ✓ 允许开发人员在程序的一些特定位置上声明假设条件并在程序运行到对应的位置时对条件进行判断
  - ✓ 一般在开发期间使用，以一种调试的方式让程序在运行过程中不断进行自检

程序状态检测：以堆栈为例

- 堆栈完成初始化创建后，元素个数为0
- 执行入栈操作前，堆栈元素个数小于容量
- 执行入栈操作后，堆栈元素个数比原来多1
- 执行出栈操作前，堆栈元素个数大于0
- 执行出栈操作后，堆栈元素个数比原来少1个，而返回值等于此前堆栈的栈顶元素
- 执行返回栈顶元素操作后，堆栈元素个数不变，而返回值等于此前堆栈的栈顶元素

断言的使用场合和检测内容

- 常见的使用断言的地方
  - ✓ 对子程序输入值的合法性进行判断
  - ✓ 对子程序输出结果是否符合预期进行检测
  - ✓ 在子程序执行过程中的一些关键点（例如循环结束后）上对程序状态进行检测
  - ✓ 在测试代码中对测试结果是否符合预期进行判断
- 断言包含一个可以在运行时自动检查的条件表达式，其中检测的内容一般包括
  - ✓ 输入、输出或状态值是否在预期范围内或等于特定值
  - ✓ 指针或对象是有效的
  - ✓ 相关资源处于合理的状态
  - ✓ 变量的值没有被意外修改
  - ✓ 多种计算方式的结果相同

断言的表达和使用方式

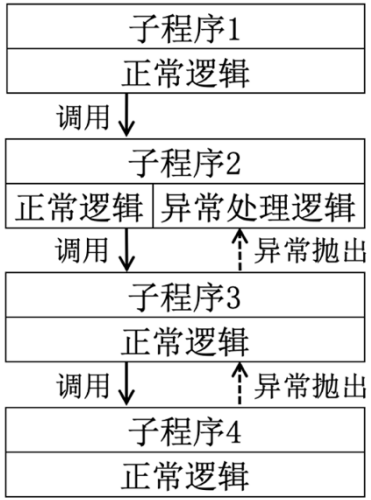
- Java断言示例：`assert payment > 0 :` “支付金额应大于0”
  - ✓ 明确假设变量payment的值大于0
  - ✓ 冒号后的文字说明则解释了这句断言的意图
  - ✓ 程序运行到这条语句时，如果payment的值小于或等于0，那么断言失败，程序会终止运行
- 不正确的使用：`ASSERT(performAction())`
  - ✓ performAction方法执行某个操作并返回一个布尔值表示执行是否成功
  - ✓ 在非调试环境中可能并不会执行，导致正式运行时缺少相应操作而出错
  - ✓ 断言应当只做判断，而不应该包含需要执行的代码
- 断言会终止程序运行，一般不会在正式版本中使用
  - ✓ Java程序中的断言一般仅用于测试代码中
  - ✓ C++程序可能会在功能实现代码中使用断言用于调试

异常处理

- 程序运行过程中难免会发生各种异常情况，例如空指针、文件读写失败、网络通信故障等
- 这些异常情况难以通过错误处理机制或断言来处理
  - ✓ 这些异常并不都是可以预期的，无法通过预设条件进行检查，而且无法保证开发人员总是考虑到所有情况
  - ✓ 有些异常是由于外部原因造成的，例如文件或网络访问，难以进行检查
- 许多语言都提供异常（Exception）的处理机制
  - ✓ 将错误或异常情况包装成一种事件并传递给调用方，通知它们发生了不可忽略的错误并要求它们进行处理
  - ✓ 调用方发现异常后可以进行针对性的处理，如果不知如何处理也可以继续向上抛出（throw）给更上层的调用方

异常处理中的传递

- 异常会中断子程序的正常执行，以无法被忽略的方式逐层传递
- 每个异常最终都会在某个层次上进行处理
- 在一般的应用程序中最高层次的用户界面如果收到异常并且没有进行处理，那么用户将会从界面上看到异常报告



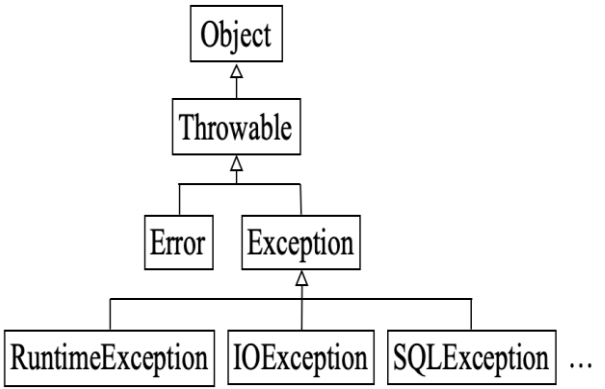
影响用户友好性
和系统安全性

Java中的异常处理手段

- try-catch语句：按照所声明的异常类型捕获try代码块中发生的异常并进行处理
- try-catch-finally语句：finally代码块定义了一些即使try语句块中止也必须执行的一些操作（JDK1.7版本开始提供try-with-resources机制，由程序运行环境自动维护相关资源的释放工作）
- throw语句：当前执行代码出现问题需要产生一个异常对象并向上层程序抛出，同时交出控制权
- throws语句：Java方法声明当前方法并不对某些类型的异常进行处理

Java中的异常类继承层次

- 运行时异常（Runtime Exception）
  - ✓ RuntimeException及其子类，如NullPointerException、IndexOutOfBoundsException等
  - ✓ 这些异常编译器不会检查程序是否有处理
  - ✓ 程序运行时如果这种异常没有捕获处理，那么会一直往上抛出，直至某个子程序处理或者导致整个程序中止运行
- 受检异常（Checked Exceptio`
  - ✓ 其他Exception类的子类，如输入输出IOException和SQLException
  - ✓ 编译器会对这类异常的处理进行检查
  - ✓ 必须明确被捕获处理或明确声明抛出，否则编译不通过



异常抛出的规范

- 只在真正例外的情况下抛出异常，不用异常来推卸责任
  - ✓ 增加程序复杂性：调用方需要了解子程序可能抛出的异常
  - ✓ 弱化程序的封装性：异常信息中包含关于子程序实现细节的信息，例如进行的操作、出现的例外情况等
- 抛出的异常类型应与所处的抽象层次相符
  - ✓ 例如，负责选课处理的子程序如果抛出文件未找到的异常将会很奇怪，因为二者不在一个抽象层次上
  - ✓ 如果收到底层抛出的文件未找到的异常且无法处理，那么应该向上抛出自定义的业务异常，例如课程信息读取失败
- 所抛出的异常消息中应当包含理解异常抛出的原因所需要的全部信息

异常捕获和处理的规范

- 不要直接捕获异常基类Exception，否则会导致难以进行对应异常的恢复
- 不要直接捕获可通过预先的检查消除的运行时异常（例如空指针、数组越界等）
- 避免使用空的catch语句
- 不能处理的异常要继续向上抛出，抛出时可以选择原样抛出还是重新包装后再抛出
- 防止通过异常信息泄露系统内部敏感细节
- 异常处理策略也是一个需要通盘考虑的“设计”问题（不同层次的处理职责和协作等）

泄露系统内部敏感细节的异常信息

- 异常信息中泄漏的敏感信息：服务器端软件使用Spring框架、调用REST服务使用RestTemplate类、内部敏感服务的URL等
- 让攻击者了解软件内部细节和逻辑，从而有利于他们找到系统的漏洞并开展攻击
- 对外提供的异常信息需要对敏感信息进行过滤
- 可以建立集中的异常处理机制

安全编程函数

- 各种编程语言提供的实现内存分配、访问和释放等敏感操作的库函数存在安全风险
  - ✓ 需要特别关注内存访问边界检查的问题，例如程序运行时是否只访问了它该访问的内存、是否越界访问了它不该访问的内存区域等
  - ✓ 很多系统安全漏洞都与内存访问的边界检查与控制有关
- 如果编程语言没有提供内置检查，那么开发人员可能会忽视相关问题从而导致安全隐患
- 一些开发库及厂商在标准库函数基础上增加问题检验和规避处理，形成相应的安全函数版本
  - ✓ 例如C语言的内存复制安全函数memcpy_s(dstBuf, dstMax, src, srcLen)将src指向的内容复制到dstBuf指向的区域时，会检查dstMax与srcLen的大小以确保不产生内存区域越界的问题
  - ✓ 类似的安全函数还包括memset_s、strcpy_s、sprintf_s等

代码质量控制

- 包含编码在内的软件构造过程除了编码之外还包括本地编译构建、本地个人测试和调试、代码提交与合并、项目级集成与测试等
- 贯穿着从个人到团队、自动化工具与人工手段相结合的质量控制过程以保障代码质量
  - ✓ 本地编译构建：需要解决程序编译错误及依赖环境配置等问题
  - ✓ 本地个人测试和调试：通过单元测试验证所编写的代码单元是否满足预定义的开发要求并对测试发现的问题进行调试
  - ✓ 代码提交与合并：通过代码质量门禁和代码评审等环节进行代码质量检查，然后才能将自己编写的代码合并到主干分支中
  - ✓ 代码度量：软件开发组织和个人还经常使用代码度量工具对代码进行量化度量，从而及时发现各种问题

个人测试与调试

- 开发者测试（Testing）
  - ✓ 验证所开发的代码单元（例如一个模块、文件或者类）是否符合开发任务要求，属于单元测试
  - ✓ 单元测试用例一般是根据代码单元的开发要求来编写的，例如需要提供的接口、实现的功能等
  - ✓ 测试驱动的开发：开发人员在编码之前先编写好测试用例并将通过这些测试用例作为完成开发任务的检验标准
- 开发者调试（Debug）
  - ✓ 测试发现问题（例如程序异常退出、运行结果不符合预期等）之后对问题进行分析和解决的过程
  - ✓ 问题定位：重现失败的测试用例并确定问题（缺陷）在代码中的什么位置
  - ✓ 问题修复：通过修改代码对问题进行修复
  - ✓ 修复验证：重新运行原有测试用例（即进行回归测试）验证问题是否已经得到修复，有时还会增加新的测试用例以进行确认

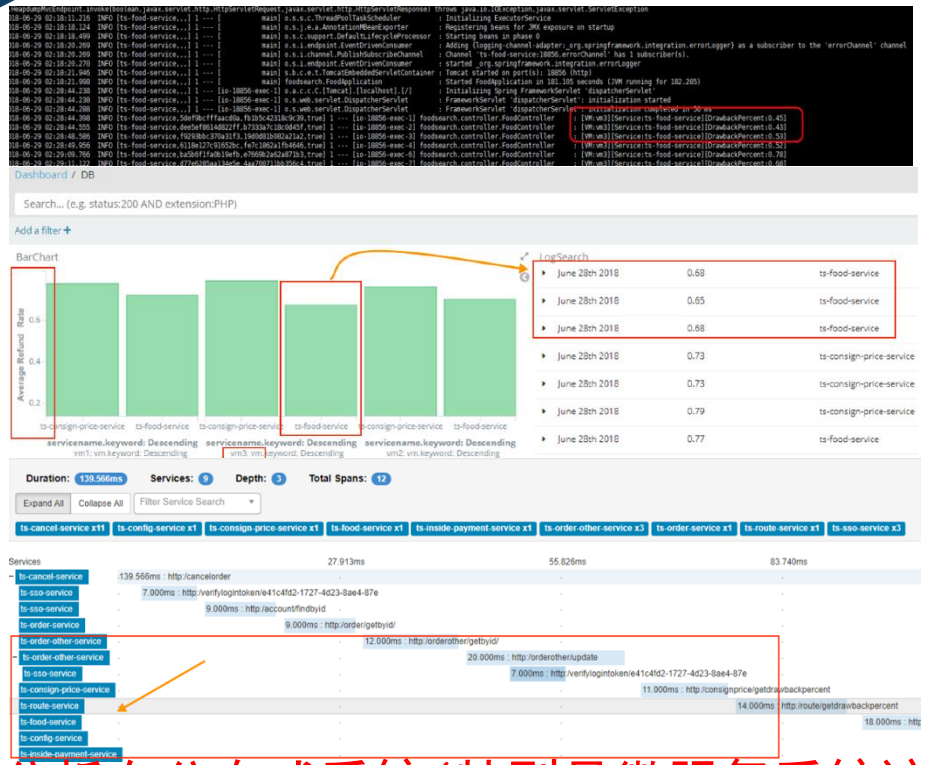
调试中的问题定位

- 调试过程中最困难的是问题定位
  - ✓ 问题的表面现象（例如界面报错或不正确的结果）与问题的根源（例如底层代码中的一处逻辑错误）可能相距很远
  - ✓ 问题的表面现象受到多种因素影响，存在一定的不确定性
- 调试中的问题定位是一种经验性的工作
  - ✓ 一种常用的策略是采用二分法，不断作出假设并进行验证，根据结果确认或否定假设
  - ✓ 例如一个字符串处理程序出错，开发人员根据经验判断可能是因为传入的字符串参数中包含特殊符号，那么可以尝试去掉特殊符号，接下来如果测试通过那么很可能就是因为这个原因，否则可以初步排除这个原因
- 在具体调试过程中，开发人员可以利用一些常用的方法和工具来提升调试的效率

常用的问题定位方法和工具

- 断点和单步运行
  - ✓ 可以观察程序的执行路径（例如所经过的分支和语句）以及在此过程中的状态（例如相关变量的取值）变化情况，从而深入理解程序的实际行为并判断与预期是否相符
  - ✓ 一旦发现程序运行过程与预期不相符，那么就可以回溯问题的来源，定位到问题的根源
  - ✓ 难点：难以决定断点位置的设置和需要观察的程序状态
  - ✓ 缺点：侵入式手段，可能改变程序多线程或异常抛出行为
- 日志
  - ✓ 日志以记录程序执行过程、中间状态和出错信息，便于还原并分析出错过程
  - ✓ 通过多轮迭代的日志查看和日志输出内容调整来进行问题定位
  - ✓ 复杂分布式系统（例如微服务系统）调试的重要手段
  - ✓ 日志输出的详细程度需要恰到好处，既能帮助发现关键的程序错误，又不会由于日志过多而影响到错误定位和程序运行性能

日志分析在分布式微服务系统中的应用



通过命令行工具搜索日志中的特定关键词，查找故障定位的线索

通过从日志中抽取特定的事件（例如某种异常抛出、某个操作执行等）并进行统计分析

通过调用链路日志获得服务请求的执行链路信息（即服务调用链）并进行可视化分析

日志分析在分布式系统（特别是微服务系统这种大规模分布式系统）故障分析和调试中起着重要的作用

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, Dan Ding: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Trans. Software Eng. 47(2): 243-260 (2021)

课堂讨论：程序运行日志

课堂讨论：有没有在自己的程序中进行过日志输出？这些日志发挥了什么样的作用？

代码静态检查

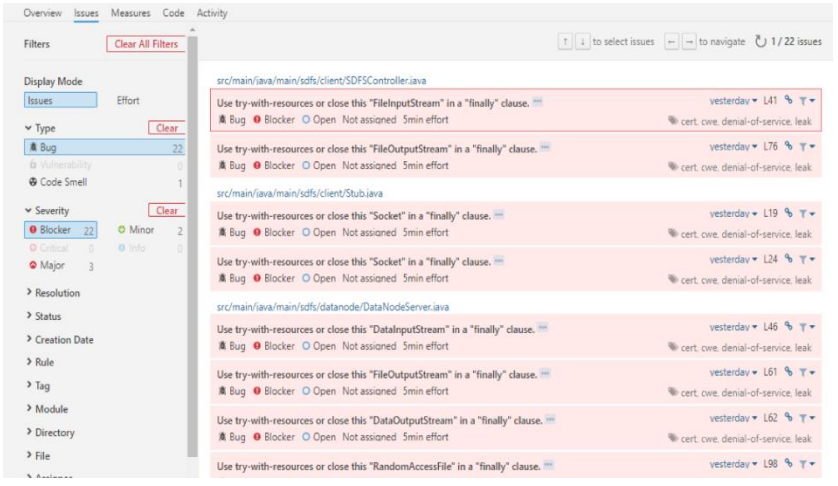
- 开发人员在完成本地的开发工作之后可以提交代码并将其合并到主干分支
- 虽然这些代码可能已经通过本地测试，但仍然可能存在多种潜在的问题，包括逻辑错误、代码坏味道、不好的代码风格等
- 在企业开发实践中，代码静态检查工具被广泛应用于各种代码质量问题的自动化检测
  - ✓ 常用的代码静态检查工具包括SonarQube 、SpotBugs （前身是FindBugs ）、CheckStyle 、P3C 等
  - ✓ 普遍都利用代码静态分析技术来发现潜在的代码质量问题
  - ✓ 一般都使用基于规则的检测方法来发现各种代码质量问题

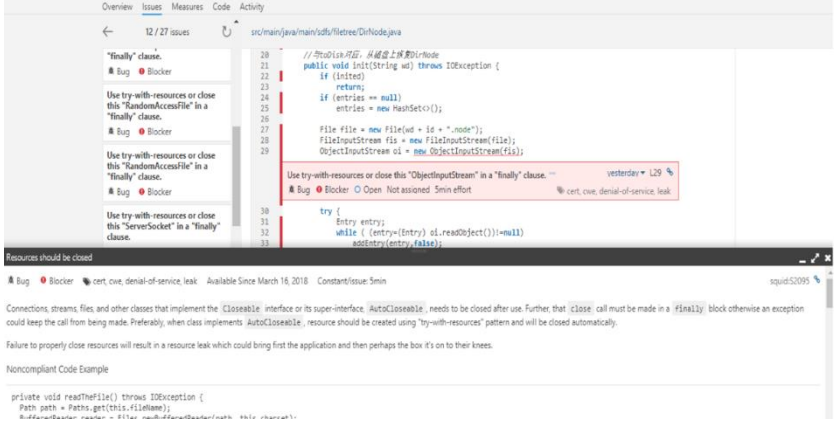
代码质量检测工具SonarCube

问题类型：
缺陷（Bug）
漏洞（Vulnerability）
代码坏味道（Code Smell）

严重程度（从高到低）：
Blocker
Critical
Major
Minor
Info等

问题详细信息：所在位置、问题类型、严重程度、问题解释





基于代码静态检测的质量门禁

- 个人本地检测
  - ✓ 开发人员在进行编码活动时利用IDE中的代码静态扫描插件实时扫描代码中可能存在的质量问题并及时进行改进
  - ✓ 对于提升个人编写代码的质量有重要的作用
- 作为持续集成（Continuous Integration，简称CI）流水线的一部分
  - ✓ 自动执行检测，作为持续集成结果的一部分进行反馈
- 作为代码提交或合并之前的质量门禁，例如
  - ✓ 要求提交或合并的代码中不允许存在严重的代码质量问题（即严重程度为Critical或更高）
  - ✓ 要求所发现的代码质量问题的数量不能增加

课堂讨论：代码质量静态检测

课堂讨论：你们在课程项目中通过代码质量静态检测发现了自己小组的代码中的哪些质量问题？如何理解这种静态检测工具的作用？

代码评审

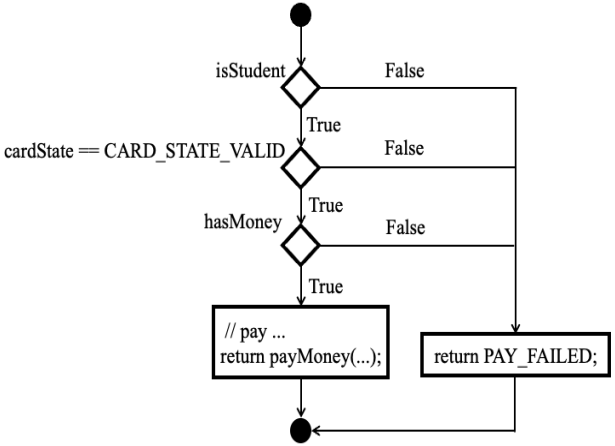
- 有些代码质量问题很难通过测试或静态检查来发现，但有经验的开发人员很容易看出来
- 代码评审：通过代码阅读的形式来判断源代码是否符合指定的代码质量标准，同时发现潜在的代码质量问题
- 正式的代码评审：在代码提交、合并的流程中进行控制，内容包括编码规范、实现逻辑、异常处理等，评审不通过的代码不能进入代码库
- 代码评审工具
  - ✓ Phabricator、Collaborator、Gerrit等工具
  - ✓ 目的是集成配置管理和版本管理工具、便于阅读和理解代码修改（例如高亮差异部分）及添加评审反馈（打分、备注等）

代码质量度量

- 工程化管理的一个基本原则就是量化度量，例如汽车发动机的转速、燃油消耗率等指标
- 类似的，我们也希望针对代码质量定义相应的量化指标，从而帮助我们针对代码质量进行量化分析同时及时发现代码质量问题
  - ✓ 代码规模度量：代码行数、方法数、文件数、类数、编译后二进制码字节数
  - ✓ 代码复杂度度量：体现代码中逻辑组合带来的复杂性，常用的指标包括圈复杂度（cyclomatic complexity）和认知复杂度（cognitive complexity）
  - ✓ 代码重复度度量：当前代码中有多大比例的代码属于重复代码，可用于发现代码重构和改进机会、发现可复用的公共模块、衡量开发团队和开发人员的有效工作量

代码复杂度度量

- 代码圈复杂度
  - ✓ 衡量代码的程序流图表示中圈的数量
  - ✓ 基于程序流图表示：箭头表示执行流程，菱形表示简单条件判断（包含AND、OR操作的复合条件需要分解），方框表示执行的语句块
  - ✓ 圈复杂度的直观定义就是程序流图所分割的平面区域的数量，另一种算法是简单条件判断的数量加1



- 代码圈复杂度的合理范围
  - ✓ 1-5之间比较理想
  - ✓ 6-10需要注意并考虑进行简化
  - ✓ 超过10需要高度重视并尽量简化

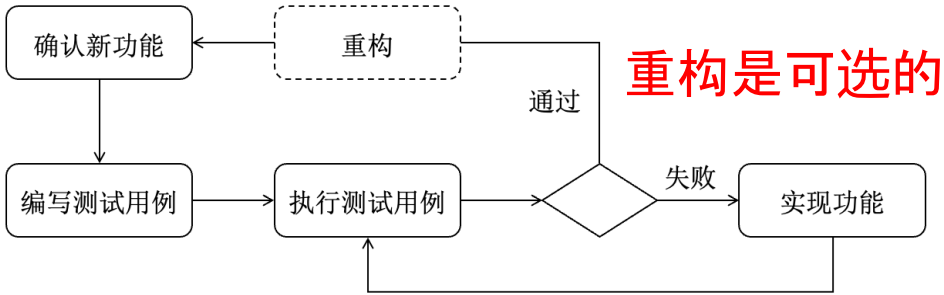
圈复杂度4:
3个封闭区域+1个开放区域
3个简单判断+1

测试驱动开发（TDD）

- 测试驱动开发（Test-driven development）是Kent Beck所提出的极限编程方法论中的最佳实践之一
- 倡导测试先行，编码之前先编写测试用例并将通过测试作为开发任务完成的标志
- 主要优势
  - ✓ 测试用例作为一种“可执行”的需求强化了需求和开发任务理解，同时使开发任务有了一个相对客观的评判标准
  - ✓ 强化了对待实现的软件模块、文件或类的对外接口的要求，从而强化了软件设计方面的思考
  - ✓ 使得软件测试无法被忽略，测试用例能更好反映开发要求
  - ✓ 测试用例成为一种衡量开发进度的客观手段
  - ✓ 测试用例可以积累形成可复用的软件开发资产并在后续代码修改后被重复执行进行回归测试

TDD的基本过程

- 基本过程
  - ✓ 确认一个新功能后编写测试用例
  - ✓ 直接运行新测试用例的结果应该是期望中的失败
  - ✓ 进行代码编写或修正，随后不断运行测试直至测试都通过
  - ✓ 在测试通过后也可以对代码进行内部优化（即进行重构）
- 三个阶段
  - ✓ “红”：测试失败，即测试用例对应的实现代码还没有完成
  - ✓ “绿”：测试通过，即实现了功能代码从而使测试得以通过
  - ✓ “重构”：测试通过后，在不改变程序外部行为的前提下对代码进行内部优化（例如去除重复代码等）



TDD的基本原则

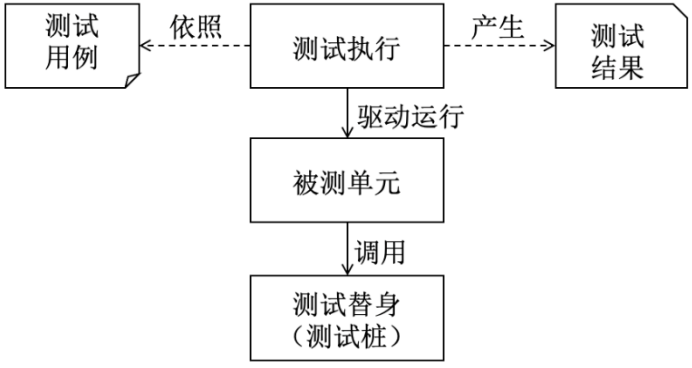
- 测试先行：最根本的原则，在开始编码之前首先针对开发任务编写测试用例
- 从用户的角度出发编写测试用例：体现系统相关应用场景的要求，不要站在编码实现的角度
- 小步前进：每次实现的功能粒度尽量小
- 测试自动化：实现测试用例执行的自动化并加入持续集成
- 小步提交：在代码出错时通过版本控制系统实现快速回滚，有效减少出错处理的代价
- 及时小规模重构：不要等到所有功能都完成后再重构，否则容易导致问题累积、重构难度大

TDD中的单元测试

- TDD中的开发者测试一般都是单元测试级别
- 单元测试是针对程序基本组成单元的一种测试方法，包括子过程（方法或函数）、类或模块
  - ✓ 针对子程序：验证其针对不同输入组合的行为是否符合预期，包括程序的输出结果、状态变化、异常抛出、外部数据变化等
  - ✓ 针对类：验证类及对象从初始化到执行一系列操作的过程中的行为是否符合预期，包括方法的输出结果、行为及状态变化
- 通过单元测试用例作为完成任务的主要标志
  - ✓ 单元测试用例定义需要综合考虑多种不同的情形
  - ✓ 例如，堆栈类的单元测试用例需考虑：连续入栈多个元素后再连续出栈、入栈和出栈交替进行、栈空时尝试出栈、栈满时尝试入栈、查询栈元素个数和栈顶元素
- TDD中的单元测试的主要目的是定义一个满足开发要求的程序是什么样，而非发现缺陷

单元测试框架

- 单元测试应当尽可能地自动化，采用工具而非人工的方式来执行
- 被测单元经常依赖于其他单元，此时需要将被测单元与所依赖的部分进行隔离，以保证测试结果不受所依赖的部分的影响
- 为了实现以上目标需要搭建一套单元测试框架
  - ✓ 测试用例本身以可执行代码的形式编写，驱动被测单元（如一个新编写的类）运行并产生测试结果
  - ✓ 测试用例中的断言对被测单元的实际行为与测试用例的预期行为的一致性进行判断，如果所有断言成立则通过否则失败
  - ✓ 需要满足依赖关系以使得被测单元可以运行，同时又要隔离依赖，因此经常使用测试替身



单元测试工具

- 使用最广泛的单元测试框架是xUnit系列：Junit（Java）、CppUnit（C++）、PyUnit（Python）
- JUnit5于2017年正式发布，作为Java程序的单元测试框架已被广泛使用
  - ✓ 通过@Test、@ParameterizedTest、@BeforeAll、@AfterAll等Java注解（Annotation）定义测试的行为
  - ✓ 通过Assertions类下的各类断言（包含assertEquals、assertTrue、assertSame等）对测试结果进行判定
- JMockit 是一款针对Java类、接口、对象的Mock工具，被广泛应用于Java单元测试
  - ✓ 两种模拟方式：一种是基于行为的，另一种是基于状态的
  - ✓ 其他常见的开源Mock工具有JMock、EasyMock、Mockito等，JMockit被看作对JMock做了进一步的封装

单元测试用例代码示例

```
1   public class testCalFee {
2       @Test
3       public void testCalFeeByCity() {
4           MockUp<IGetPostcode> stub = new MockUp<IGetPostcode>() {
5               @Mock
6               public String getPCodeByCity(String city) {
7                   if ("Shanghai".equals(city)) {
8                       return "200000";
9                   }
10                  else {
11                      return "000000";
12                  }
13              }
14          };
15          IGetPostcode mockInstance = stub.getMockInstance();
16          CalFee calFee = new CalFee(mockInstance);
17          assertEquals(20,calFee.calFeeByCity("Shanghai"));
18      }
19  }
```

测试类

测试方法

根据预设的城市名称返回对应的邮编

定义并构造一个代表IGetPostcode接口的测试桩sub

实例化测试桩

实例化被测方法的实例化对象

利用断言对返回结果进行判断

本章小结-1

- 软件代码质量包括外部和内部两方面
  - ✓ 外部质量：面向软件的运行和使用，强调代码应当杜绝缺陷和漏洞，同时具有良好的性能、可靠性和安全性
  - ✓ 内部质量：面向软件长期演化和维护，强调代码的易理解性、易修改性和易扩展性
- 代码风格和代码逻辑
  - ✓ 代码风格：涉及标识符命名、代码排版和注释等方面，影响代码的可理解性和可维护性
  - ✓ 代码逻辑：要求代码逻辑严密、尽量避免错误，同时要控制代码复杂度、减少代码重复以及编写高质量的子程序

本章小结-2

- 安全和可靠性编码
  - ✓ 遵循包含防御式编程在内的一系列编程实践准则
  - ✓ 尽量使用不容易出错的安全函数
- 代码质量控制
  - ✓ 从个人到团队、自动化工具与人工手段相结合的质量控制过程
  - ✓ 包括个人测试与调试、代码静态检查与质量门禁、代码评审、代码质量度量等
- 测试驱动开发（TDD）
  - ✓ 倡导在编码之前先编写测试用例并将通过测试作为开发任务完成的标志
  - ✓ 强化了编码活动的测试保障
  - ✓ 对于高质量编码具有重要的作用

COMP130015.02
软件工程

End

4. 高质量编码