

L5 质量管理(2)

——代码评审与高质量代码

大纲

- 代码评审概述
- 高质量代码的编码风格
- 静态代码分析及代码评审工具

大纲

- **代码评审概述**
 - 质量控制相关技术比较
 - 人工评审与自动分析
 - 评审的频度
 - 代码测量指标
- 高质量代码的编码风格
- 静态代码分析及代码评审工具

质量控制相关技术比较

- 评审

评审一般以静态方式（即不运行程序），对开发过程中的工作产品进行评估审查

用于确认编码或设计等方面的测量指标，包括编码标准符合性、架构合理性、重复代码量等。

- 测试

测试是动态的（即运行程序），需要通过以受控的方式运行软件来确认功能或性能（测试后续课程展开）

- 评审发现和修复缺陷的效率高于测试
- 测试和评审可以自动执行，也可以手工执行

质量控制相关技术比较

- 自动评审（自动分析）
 - 代码静态扫描工具
- 人工评审
 - 基于代码评审工具的过程评审；开发中的临时评审
- 自动化测试
 - 对测试用例进行编码，采用自动化测试工具自动运行并自动验证
 - 通常可以和自动化构建集成
- 手工测试
 - 人工运行测试过程、记录测试结果

人工评审与自动分析

代码评审可以是人工方式，也可以是由工具自动执行的方式。

自动工具（静态代码分析工具）的优点有：

1. 自动工具的运行成本很低，可反复自动执行，可以节省大量的时间。
2. 自动工具具有客观性，其结果不会受人与人之间的关系的影响。
3. 自动工具中的规则是可配置、可定制的，组织机构可选择适用的规则和参数。
4. 因为不需要面对面的讨论和沟通，自动工具对于地理上分布的开发团队同样有效。

人工评审与自动分析

工具可能能够完成80%的评审工作，但是依然有很多需要人来评审的问题。比如：

1. 架构相关
2. 领域抽象相关
3. 对于工具查出来过于复杂的部分，往往意味着需要不同的抽象或模型，这些需要人来介入修改

评审的频度

- 错误发现得越早，纠正的代价就越小，因此评审需要**保持较高的频度**。
- 通过持续进行评审，有助于：
 - 防止引入缺陷
 - 引入缺陷后尽早发现
 - 发现缺陷后较快和较易修复
- 在只执行人工评审的项目中，由于频度较低，缺陷可能在进入代码后很长时间才被发现。
- 定义清晰的、可度量的准则，尽可能使用自动化评审的工具，并且与CI工具和流水线集成。

代码测量指标：历史

- 软件开发效率和质量与代码之间有怎样的关系？
- 是否有一些仅靠对代码进行测量就能得出开发效率和质量指标？
 - 代码行？
 - 缺陷数量？缺陷率？
- 度量指标应被合理应用
 - “一抓就死，一放就乱”
- 因此，在开始度量代码之前，需要了解误用代码度量指标可能对软件开发产生的不良后果。

代码测量指标：历史

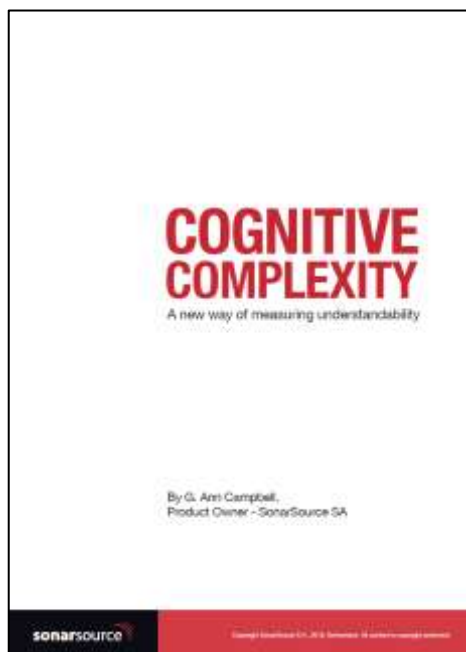
- 使用**代码行数**来测量**开发效率**事实上会导致对复制粘贴这种行为的鼓励
 - 复制粘贴产生的重复代码往往导致了系统内在质量恶化。
- 对**注释比例**的测量会导致无意义的、有干扰性的甚至是误导性的注释。
 - “注释最多也就是一种必须的恶，只是用来弥补用代码表达意图时遭遇的失败。”

代码测量指标：有价值的测量指标

- 复杂度是一个相对有价值的参考指标。
- 比如CCN（圈复杂度）是通过一个方法的执行路径数来衡量的复杂性。
 - 一般认为，CCN大于10的代码存在缺陷的风险更大。
 - CCN可以通过工具自动计算得出，其结果为评审和重构提供了重要的依据。
- 但也要注意不能过度依赖单一指标

代码测量指标：有价值的测量指标

- CCN也存在一些其他的问题，例如switch-case会导致CCN增大，但它本身却不一定很复杂
 - 引入其他复杂度测度，比如认知复杂度（Cognitive Complexity）在CCN的基础上增加了一些规则，使得复杂度度量更加接近于人对代码复杂性的认知



详情可参见：<https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html>

代码测量指标

- 我们在使用代码测量指标时，需要了解**每个指标背后的测量意图**。仅仅为了表面上满足指标而采取的行动往往适得其反。
- 可能的代码质量的测量指标
 - 采用自动化方法进行代码检查所发现的问题数量
 - 违反编码规范的次数
- 良好的编码风格是高质量代码的基础
- 下面列出一些在敏捷社区中推崇的、被实践所认可的高质量编码风格建议
 - 可以作为代码评审（人工或自动）的规则

大纲

- 代码评审概述
- **高质量代码的编码风格**
 - 1. 命名
 - 2. 函数
 - 3. 注释
 - 4. 格式
 - 5. 对象与数据结构
 - 6. 错误处理
 - 7. 类
- 静态代码分析及代码评审工具

高质量代码的编码风格

- 关于高质量代码的编码风格建议主要来自Robert C. Martin的《Clean Code》。
- 为便于讲授，略有取舍。



高质量代码的编码风格

- 1. 命名
- 2. 函数
- 3. 注释
- 4. 格式
- 5. 对象与数据结构
- 6. 错误处理
- 7. 类

命名

- 名字应该表达意图

- 选一个能够确切表达意图名字需要花时间，但之后省下来的时间比花掉的多。
- 一旦发现有更好的名称，就替换掉旧的。
- 如果需要使用注释来补充名字的含义，那就还没有达到表达意图的目标。

命名

- 名字应该表达意图

```
int d; //消逝的时间，以日记
```

作为比较：

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

命名

- 名字应易于沟通
 - 能够流畅地读出来

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
    /* ... */  
};
```

```
class Customer {  
    private Date generationTimestamp;  
    private Date modificationTimestamp;  
    private final String recordId = "102";  
    /* ... */  
}
```

- 不能太长

```
if(XYZControllerForEfficientHandlingOfStrings=="ABC")
```

与

```
if(XYZControllerForEfficientStorageOfStrings=="ABC")
```

命名

- **类名与方法名**

- 类名

类名通常为名词，比如

Customer WikiPage

- 方法名

方法名通常为动词，比如

createRealNumber placeOrder


命名

- 方法命名为 exhaleProcess

```
public Object exhaleProcess(DialogDTO vrDialog) {  
    CallOutUser callOutUser =  
        callOutUserService.queryByPhoneNum( ...)  
    .....  
    .....  
}
```

exhale 英[eks'heɪl]   美[eks'heɪl]  

v. 呼出, 吐出(肺中的空气、烟等); 呼气;

[例句] As the lungs **exhale** this waste, gas is expelled into the atmosphere. 
肺**呼出**这些废气被排到空气中。

命名

- 使用**问题域**中的术语
 - 在涉及到领域模型的部分。尽可能使用领域专家或客户使用的术语，包括名词或动词。
- 使用**解决方案域**中的术语
 - 对于涉及到解决方案的部分，尽可能使用计算机/软件领域熟悉的专业名称
 - 比如：如果使用了Visitor模式，就应该在相关代码中使用Visitor、accept等术语；如果使用了Observer模式，就应该使用Observer/Listener这类的名称。
- 汉语程序员的困难
 - 对关键的领域术语建立词汇表，在团队内部进行统一

高质量代码的编码风格

- 1. 命名
- **2. 函数**
- 3. 注释
- 4. 格式
- 5. 对象与数据结构
- 6. 错误处理
- 7. 类

函数

- 小的函数
- 只做一件事
- 每个函数一个抽象层级
- 消除switch语句
- 函数的参数
- 无副作用
- 区分命令和查询
- 避免返回错误代码
- 不要复制代码

短小的函数 (反例)

Listing 3-1

HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage(
                    SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
        WikiPage setup =
            PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName)
                .append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
            PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath teardownPath =
                wikiPage.getPageCrawler().getFullPath(teardown);
            String teardownPathName = PathParser.render(teardownPath);
            buffer.append("\n")
                .append("!include -teardown .")
                .append(teardownPathName)
                .append("\n");
        }
    }
}
```

代码清单3-1:

testableHtml: 输入PageData以及
是否需要包含SuiteSetup
处理PageData后返回PageData页面

需要Setup和Teardown

Listing 3-1 (continued)

HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
        WikiPage suiteTeardown =
            PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
            );
        if (suiteTeardown != null) {
            WikiPagePath pagePath =
                suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName = PathParser.render(pagePath);
            buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
}
```

短小的函数

- 函数到底应该多长?
 - 一屏？（早期的屏幕最多只有24行）
 - 100行？（高分辨率的大显示器确实能显示这么多）
 - 30行？
 - 3行？
- 有学者认为：越短越好，考虑20行封顶
 - 短小可以只聚焦于一件事情
- If/else/while语句其中的代码块应该只有一行
 - 一个函数调用；更具说明性

只做一件事

- 函数应该做一件事。
 - **做好**这件事，**只做**这一件事。

Listing 3-3

HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(  
    PageData pageData, boolean isSuite) throws Exception {  
    if (isTestPage(pageData))  
        includeSetupAndTeardownPages(pageData, isSuite);  
    return pageData.getHtml();  
}
```

- 如果函数只是做了该函数名下**同一抽象层次**上的步骤，则函数就是只做了一件事

每个函数一个抽象层级

- 向下规则
 - 代码应该有**自顶向下**的阅读顺序。
 - 这类似于树形结构的**广度优先**的遍历，处于同一层次的语句应该位于同样的抽象层级。
 - 代码清单3-1违反了这条规则：
 - 有getHtml等位于较高抽象层次的概念
 - 直接对输入参数PageData整体获取值
 - 也有pagePathName = PathParser.render(pagePath)中等抽象层次的概念
 - 还有append(“\n”)等较低抽象层次的概念

消除Switch语句

- 下面的代码给出了根据雇员类型来判断的一种计算操作。

Listing 3-4

Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

消除Switch语句

- 问题

- 太长，并且会越来越长
- 在其他地方可能有类似结构的函数
 - 比如，需要根据雇员类型来计算休假的时间

- 解决

- 可以考虑使用策略模式或者多态来解决
- 只在一处（比如工厂中）出现这样的结构
- 此类问题非常常见，有时候会很让人头痛，需要适当的机会进行整理

函数参数

- 函数参数的个数
 - 最理想的是无参数: niladic (对于OO来说这种情况实际上是一个this参数)
 - 其次是一个参数: monadic
 - 再其次是两个参数: dyadic
 - 应该避免三个以上的参数: triadic
 - 比较
 - **niladic** : task.isFinished()
 - **monadic** : writeField(name)
 - **dyadic** : writeField(outputStream, name)
 - **triadic** : assertEquals(message, expected, actual)

函数参数

更多个数的参数：

```
public EntryDataOutputListOutputDAOObject[] selectOutputDataEntryData(
    SepOutputFormSetupDAOObject[] _conditionDAOObject,
    String[] _strCondition1,
    String[] _strCondition2,
    String[] _strCondition3,
    String[] _strCondition4,
    String[] _strSearchOperatorDiv,
    SepOutputFormSetupDAOObject[] _outputDAOObject,
    SepOutputFormcmSetupDAOObject[] _tableDAOObject,
    SepOutputFormSetupDAOObject[] _sortDAOObject,
    String _strAccessAuthorityDiv,
    String _strUserId,
    ArrayList _listOrganization,
    ArrayList _aryWfStatus,
    ArrayList _aryDataEntryDiv,
    EntryDataOutputListOutputControllerObject _controllerObject
) throws ScmSystemException {
    .....
}
```

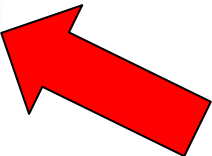

无副作用

- 副作用是一种谎言。
 - 副作用让函数变成：看上去只做了一件事，但背地里还做了另一件事。

Listing 3-6

UserValidator.java

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public boolean checkPassword(String userName, String password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase = user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase, password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```



区分命令和查询

- 如果一个函数是**命令**，那它会改变系统的状态
- 如果一个函数是**查询**，那它只获取系统当前的状态
- 函数要么做什么事，要么回答什么事，不应该同时做两件事。
- 比如，下面代码的含义就不够清楚

```
if (set("username", "unclebob"))...
```

- 应该用类似下面的形式

```
if (attributeExists("username")) {  
    setAttribute("username", "unclebob");  
    ...  
}
```

使用异常而不要使用错误代码返回

- 这样可以将正常的逻辑与异常的逻辑清晰地分离，实际上也保持了抽象层次的一致性。
- 下面的代码用的是返回值来表示错误：

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {  
            logger.log("page deleted");  
        } else {  
            logger.log("configKey not deleted");  
        }  
    } else {  
        logger.log("deleteReference from registry failed");  
    }  
} else {  
    logger.log("delete failed");  
    return E_ERROR;  
}
```

使用异常而不要使用错误代码返回

- 推荐的写法（使用异常）：

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

需要改进设计

不要复制代码

- 重复的代码总是意味着有什么地方设计不完善
- 重复的代码，或者模式相似的代码往往背后的逻辑代表了同一个功能。
 - 被复制到不同的地方后，一旦需要改变，就需要找到所有类似的位置同时修改。这是个很容易出错过程。
- “Copy and Paste is a design error”
(McConnell 1998b).
 - 一些软件系统的数据库连接和事务的管理就是一个典型的重复代码的案例。

高质量代码的编码风格

- 1. 命名
- 2. 函数
- **3. 注释**
- 4. 格式
- 5. 对象与数据结构
- 6. 错误处理
- 7. 类

注释

- 不能用于解释糟糕的代码
- 尽可能使用代码来代替注释
- 什么是好的注释
- 什么是差的注释

不用注释解释糟糕的代码

- “注释最多也就是一种必须的恶，只是用来弥补用代码表达意图时遭遇的失败。”
- 尽可能使用代码来代替注释
 - 下面的例子显示了代码可以很好地替代注释

```
// Check to see if the employee is eligible for full benefits  
if ((employee.flags & HOURLY_FLAG) &&  
    (employee.age > 65))
```

Or this?

```
if (employee.isEligibleForFullBenefits())
```


好的，或需要的注释

- 法律信息
- 提供代码无法直观表达的信息

```
// format matched kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile(  
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

- 对意图的解释
- 警示
- TODO

差的注释

- 无法让其他人理解的注释（喃喃自语）

```
,  
catch(IOException e)  
{  
    // No properties files means all defaults are loaded  
}
```

- 多余的注释

```
/**  
 * The lifecycle event support for this component.  
 */  
protected LifecycleSupport lifecycle =  
    new LifecycleSupport(this);
```

差的注释

- 误导性注释
 - 注释与代码实际行为并不相同
- 循规式注释

Listing 4-3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

差的注释

- 日志式注释

- 有了版本管理工具这种注释就完全没有价值了。

```
* Changes (from 11-Oct-2001)
* -----
* 11-Oct-2001 : Re-organised the class and moved it to new package
*               com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
*               class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
*               class is gone (DG); Changed getPreviousDayOfWeek(),
*               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
*               (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

差的注释

- 废话注释

```
/** The name. */  
private String name;
```

```
/** The version. */  
private String version;
```

```
/** The licenceName. */  
private String licenceName;
```

```
/** The version. */  
private String info;
```

高质量代码的编码风格

- 1. 命名
- 2. 函数
- 3. 注释
- **4. 格式**
- 5. 对象与数据结构
- 6. 错误处理
- 7. 类

代码格式和布局

- 布局概述
- 布局技术

布局概述

- 程序源代码的布局这一部分是关于计算机程序设计的美学方面。
- 对于有经验的程序员来说，对格式良好的代码会带来视觉和智力上的享受，这是其他人难以体会到的。
- 那些以自己的工作为荣的程序员，可以从他们代码的视觉结构中获得极大满足感。

布局概述

- 这里提到的代码布局技术不会影响程序的执行速度、内存使用或程序外部可见的其他方面。
- 它们只影响到**理解代码、评审代码**以及在编写代码数月后**修改代码的难易度**。

布局比较（计算机可理解）

```
/* Use the insertion sort technique to sort the "data" array in ascending order.
This routine assumes that data[ firstElement ] is not the first element in data and
that data[ firstElement-1 ] can be accessed. */ public void InsertionSort( int[]
data, int firstElement, int lastElement ) { /* Replace element at lower boundary
with an element guaranteed to be first in a sorted list. */ int lowerBoundary =
data[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* The elements in
positions firstElement through sortBoundary-1 are always sorted. In each pass
through the loop, sortBoundary is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the array, so it's inserted
into the proper place somewhere between firstElement and sortBoundary. */ for ( int
sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++ ) { int
insertVal = data[ sortBoundary ]; int insertPos = sortBoundary; while ( insertVal <
data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ]; insertPos =
insertPos-1; } data[ insertPos ] = insertVal; } /* Replace original lower-boundary
element */ data[ firstElement-1 ] = lowerBoundary; }
```

布局比较（人可理解）

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed.
*/

public void InsertionSort( int[] data, int firstElement, int lastElement ) {
    // Replace element at lower boundary with an element guaranteed to be
    // first in a sorted list.
    int lowerBoundary = data[ firstElement-1 ];
    data[ firstElement-1 ] = SORT_MIN;

    /* The elements in positions firstElement through sortBoundary-1 are
    always sorted. In each pass through the loop, sortBoundary
    is increased, and the element at the position of the
    new sortBoundary probably isn't in its sorted place in the
    array, so it's inserted into the proper place somewhere
    between firstElement and sortBoundary.
    */
    for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
        sortBoundary++ ) {
        int insertVal = data[ sortBoundary ];
        int insertPos = sortBoundary;
```

布局概述

- 布局和格式化的基本原则是：
 - 良好的视觉布局能够**直观地反映程序的逻辑结构**。
- The **smaller part** of the job of programming is writing a program so that the **computer can read it**; the **larger part** is writing it so that other **humans can read it**.
 - 编程工作只有一小部分内容是编写便于计算机能够理解的程序；绝大部分的内容是写人可以理解的程序。

软件代码写出来是给人看的，顺便可以用作机器执行

Harold Abelson等，计算机程序的构造与解释（原书第2版）

布局概述

人与计算机对程序的不同的理解（Java）

```
// swap left and right elements for whole array
for ( i = 0; i < MAX_ELEMENTS; i++ )
    leftElement = left[ i ];
    left[ i ]    = right[ i ];
    right[ i ]   = leftElement;
```

```
x = 3+4 * 2+7;
```

好的布局的准则

- 准确反映了代码的逻辑结构
- 以一致的方式表示代码的逻辑结构
- 改善可读性
- 经得起修改

布局技术

- 使用空格
 - 分组
 - 空行
 - 缩进
- 使用括号
- 使用工具自动格式化

避免成为“宗教信仰”

- 布局能体现美观和逻辑结构.
- 好的程序员应该对他们的布局实践持开放的态度，主动适应不同的团队需要。也许调整到一个新方法会导致一些最初的不适，但还是需要接受其他的规则和做法。

高质量代码的编码风格

- 1. 命名
- 2. 函数
- 3. 注释
- 4. 格式
- **5. 对象与数据结构**
- 6. 错误处理
- 7. 类

对象和数据结构

- 数据抽象
- 数据与对象的反对称性
- The Law of Demeter (得墨忒耳率)
- DTO(数据传送对象)

抽象

- 基于数据结构的抽象[过程式]
- 基于接口/类的抽象[OO]

Concrete Point

```
public class Point {  
    public double x;  
    public double y;  
}
```

Abstract Point

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

- 右边实现的优点是，从外部来看，没有限制点所处的坐标系
 - 既可以是极坐标，也可以是直角坐标系

数据、对象的反对称性

- 过程式或面向对象式：
 - 在不同的场景下，各自有各自的优势
 - 是否为面向对象风格不应成为判定结构优劣的依据
- 过程式代码（使用数据结构）
 - **便于**在不改动既有的数据结构的前提下添加新函数
 - **不便于**添加新的数据结构（涉及多个函数同时修改）
- 面向对象代码（接口、类）
 - **便于**在不增加新的函数的前提下增加或改变数据结构
 - **不便于**增加新的函数（涉及继承层次中的多个类同时修改）
- 需要根据应用的特点选择不同的范式

得墨忒耳律

- 得墨忒耳律又称为最少知识原则
 - 模块应该**尽可能少**地了解它操作对象内部的情形。
 - 具体来说，类C中的方法f只能调用以下对象的方法
 - C自己的方法
 - f中创建的对象的方法
 - f参数中对象的方法
 - C的成员中的方法
 - 根据得墨忒耳律，下面的代码是不合理的：

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- 得墨忒耳律应用于OO的场景，对于数据结构的场景，由于没有限制封装数据，得墨忒耳律并不适用

数据传送对象

- 数据传送对象是一个只有公共属性，没有函数的类。
- 这类对象被称为 DTO(Data Transfer Objects)
- 这类对象可以看做是纯数据结构，其中不应该有方法。

Listing 6-7

address.java

```
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;

    public Address(String street, String s
                    String city, String st
        this.street = street;
        this.streetExtra = streetExtra;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getStreet() {
        return street;
    }

    public String getStreetExtra() {
        return streetExtra;
    }

    public String getCity() {
        return city;
    }
}
```

高质量代码的编码风格

- 1. 命名
- 2. 函数
- 3. 注释
- 4. 格式
- 5. 对象与数据结构
- **6. 错误处理**
- 7. 类

错误处理

- 使用异常而不是返回错误代码
 - 在之前的函数部分已说明
- 使用非受检异常
- 自定义异常
- 异常中包含足够的信息
- 不要使用返回null表示异常
- 避免传入null

使用非受检异常

- 受检异常(`checked exception`)要求异常要么在函数中被处理掉，要么在函数的签名中声明该函数可能会抛出该异常。
- 受检异常(`checked exception`)看上去是一个好的设计，但不是必须。
 - 实际上只有Java用了`checked exception`，其他语言都没有使用，依然没有影响代码的质量。
- 受检异常破坏了封装，导致整个调用链上的函数都收到影响。

定义新的异常类

- 下面的异常处理代码中，catch块中包含了重复代码：

```
ACMEPort port = new ACMEPort(12);

try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```

定义新的异常类

- 定义一个新的异常类将这类异常包装起来:

Port的**open**方法:

```
public void open() {  
    try {  
        innerPort.open();  
    } catch (DeviceResponseException e) {  
        throw new PortDeviceFailure(e);  
    } catch (ATM1212UnlockedException e) {  
        throw new PortDeviceFailure(e);  
    } catch (GMXError e) {  
        throw new PortDeviceFailure(e);  
    }  
}
```

```
...  
}
```

调用方:

```
try {  
    port.open()  
} catch (PortDeviceFailure e) {  
    dealWith(e);  
}
```

定义常规流程

- 不要用异常来处理常规流程
- 例如：

```
try {  
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
    m_total += expenses.getTotal();  
} catch(MealExpensesNotFound e) {  
    m_total += getMealPerDiem();  
}
```

- 未找到MealExpense时expenseReportDAO抛出了一个异常。这就要求我们在代码中区别对待这两种情况

定义常规流程

- 还有一种解决方案是，定义一个MealExpense的子类，名为PerDiemMealExpense
- 未找到MealExpense时，expenseReportDAO返回一个PerDiemMealExpense，而不是抛出异常。
- 这样前面的代码可以改为更加清晰的形式：

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
m_total += expenses.getTotal();
```

不要返回null

- **Null References: The Billion Dollar Mistake ***

```
public void registerItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.getItemRegistry();  
        if (registry != null) {  
            Item existing = registry.getItem(item.getID());  
            if (existing.getBillingPeriod().hasRetailOwner()) {  
                existing.register(item);  
            }  
        }  
    }  
}
```

- 上面的代码的正常逻辑被掩埋在null check之中

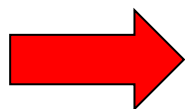
* <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/?ref=hackernoon.com>

不要传递null

- 一个函数可能接受空值的话，会增加这个函数校验参数合法性的负担。
- 下面是比较清晰的代码：

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

- 如果必须
校验参数
会让代码
变得复杂



```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x - p1.x) * 1.5;
    }
}
```

高质量代码的编码风格

- 1. 命名
- 2. 函数
- 3. 注释
- 4. 格式
- 5. 对象与数据结构
- 6. 错误处理
- **7. 类**

类

- 类应该短小
- 为演化而组织类

短小的类

- 短小是类的第一条规则

Listing 10-1

Too Many Responsibilities

```
public class SuperDashboard extends JFrame implements MetaDataUser
{
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
```

```

    public void setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public void setCurrentDir(String newDir)
    public void updateStatus(int dotPos, int markPos)
    public Class[] getDatabaseClasses()
    public MetadataFeeder getMetadataFeeder()
    public void addProject(Project project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
    public MetaProjectHeader getProgramMetadata()
    public void resetDashboard()
    public Project loadProject(String fileName, String projectName)
    public void setCanSaveMetadata(boolean canSave)
    public MetaObject getSelectedObject()
    public void deselectObjects()
    public void setProject(Project project)
    public void editorAction(String actionName, ActionEvent event)
    public void setMode(int mode)
    public FileManager getFileManager()
    public void setFileManager(FileManager fileManager)
    public ConfigManager getConfigManager()
    public void setConfigManager(ConfigManager configManager)
    public ClassLoader getClassLoader()
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
    public MetaObject pasting(
        MetaObject target, MetaObject pasted, MetaProject project)
    public void processMenuItems(MetaObject metaObject)
    public void processMenuSeparators(MetaObject metaObject)
    public void processTabPage(MetaObject metaObject)
    public void processPlacement(MetaObject object)
    public void processCreateLayout(MetaObject object)
    public void updateDisplayLayer(MetaObject object, int layerIndex)
    public void propertyEditedRepaint(MetaObject object)
    public void processDeleteObject(MetaObject object)
    public boolean getAttachedToDesigner()
```

短小的类

- 单一职责原则(SRP)
 - 类或者模块应该有且只有一条加以修改的理由。

Listing 10-2

Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser {
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```



Listing 10-3

A single-responsibility class

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

短小的类

- 内聚

- 类的成员（方法和属性）应该保持较少的规模。
- 类中的每个方法进尽可能多地用到了类中所有的属性。
 -
- 高内聚的类中的方法和属性相互依赖，相互结合为一个逻辑整体。
- 用以上面的准则确保类的内聚性，我们就会得到许多短小的类。

短小的类

Listing 10-4

Stack.java A cohesive class.

```
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();

    public int size() {
        return topOfStack;
    }

    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }

    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

Stack的三个方法中，只有size没用到所有的两个属性。这个类是高内聚的，同时也是短小的。

为演化而组织类

- 软件需要组织为这样的形式：
 - 在需要一个新的功能时，不是通过**修改**现有的代码而得到这个新的功能，而是通过**组合**现有的代码而得到这个新的功能。
- 通过设计接口和具体类来隔离修改

大纲

- 代码评审概述
- 高质量代码的编码风格
- **静态代码分析及代码评审工具**

静态代码分析工具

- 静态代码分析是指在不运行计算机程序的条件下进行程序分析的方法。
- 大部分的静态程序分析的对象是针对特定版本的源代码，也有些静态程序分析的对象是目标代码。
- 静态程序分析一词通常是指配合**静态程序分析工具**进行的分析
 - 人工进行的分析一般称为程序理解或代码评审。

静态代码分析工具

- 根据OMG发布的代码静态分析工具的类型清单，静态代码分析工具可分为下面几类：
 - 单元层面
 - 用于分析较小的单元，比如模块和子程序
 - 技术层面
 - 比如系统安全性分析
 - 关键的业务领域
 - 医疗软件：例如美国食品药品监督管理局确定在医疗设备软件上使用静态程序分析
 - 核能软件：例如英国的健康与安全委员会建议针对堆保护系统的软件进行静态程序分析。

静态代码分析工具

- 代码评审规则中的很多规则可以通过静态代码分析工具给出评价，例如：
 - 不够短小的函数、参数列表等
 - 重复的代码
 - 过于复杂的控制结构
 -

静态代码分析工具

- **Lint**

- Lint最初是作为Unix下为C/C++做静态检查的工具，现在已成为各种语言静态检查的通用术语，对大多数语言都有适配实现。

- **SonarQube**

- SonarQube 是一种很流行的静态分析工具，用于持续检查代码库的代码质量和安全性，并在代码评审期间指导开发团队。SonarQube 可与 CI/CD 集成，进行自动化代码检查。它还提供了质量管理工具帮助程序员主动纠正错误：IDE 集成、Jenkins 集成和代码评审工具集成。

静态代码分析工具

- **PMD**

- 经典且持续维护的代码静态分析工具
- 开源github.com/pmd/pmd、<https://pmd.github.io>
- 支持16种语言，主要聚焦于Java和Apex
- 集成在多个开源和商业产品中

- **Fortify**

- “快速、无摩擦的静态分析”
- 支持30+种语言和框架
- 支持多种部署方式
- 对代码组成成分、代码安全、漏洞分析、安全测试都有涵盖

静态代码分析工具

- **KlocWork**

- C/C++/Java/JS/C#代码质量静态分析工具
- 采用深度数据流分析，支持跨类、跨文件分析代码缺陷和安全漏洞
- 支持CWE、OWASP、CERT、ISO/IEC TS 17691（C安全编码规范）等多种标准规范
- 支持CI/CD集成，支持IDE集成
- 内建多种编码规则集，包括MISRA C/C++、AutoSAR C++14、JSF AV C++等

静态代码分析工具

- **库博 CoBOT**

- 北大软件自研的代码缺陷检测工具
- 通过美国CWE认证，检测语义缺陷、安全漏洞、编码规则问题
- 支持C、C++、Java、C#、Scala、Groovy、Kotlin、.net等多种语言混合检查
- 支持编译检测也支持编译不通过检测
- 支持国内编码标准和国产环境
- 支持多种代码度量、克隆检测、逆向工程等其他功能
- 提供二次开发SDK

静态代码分析工具

- **Codacy**

- Codacy是一个静态分析工具，可以帮助开发人员处理技术债务并提高代码质量。Codacy 监控每一次代码提交和 PR 的代码质量。你可以用它来加强代码质量标准，加强安全实践，并节省代码评审时间。

- **DeepScan**

- DeepScan 是一个支持 JavaScript、TypeScript、React 和 Vue.js 的静态分析工具。你可以使用 DeepScan 来查找部分运行时错误和质量问题，而不只是编码风格问题。将 DeepScan 与你的 GitHub 代码库集成起来，以此来发现项目的质量问题。

静态代码分析工具

- **Embold**

- Embold 是一个通用的静态分析器，可以帮助开发人员在关键代码问题成为障碍之前把它们找出来。它是一个有效诊断、转换和维护应用程序的得力工具。它集成了人工智能和机器学习技术，可以找出一级问题，提供最佳解决方案，并在必要时重构应用程序。你可以在已有的 DevOps 技术栈中使用它，可以在内部使用，也可以在私有云和公共云中使用它。

- **Veracode**

- Veracode 是一种流行的静态代码分析工具。它只针对安全问题，跨管道执行代码检查，以便发现安全漏洞，并将 IDE 扫描、管道扫描和策略扫描作为其服务的一部分。它会创建用于审计的代码评估，作为程序的一部分。

静态代码分析工具

- **DeepSource**

- DeepSource 可以帮你在代码评审期间自动发现并修复代码中的问题。它可以与 Bitbucket、GitHub 或 GitLab 帐户集成

- **Reshift**

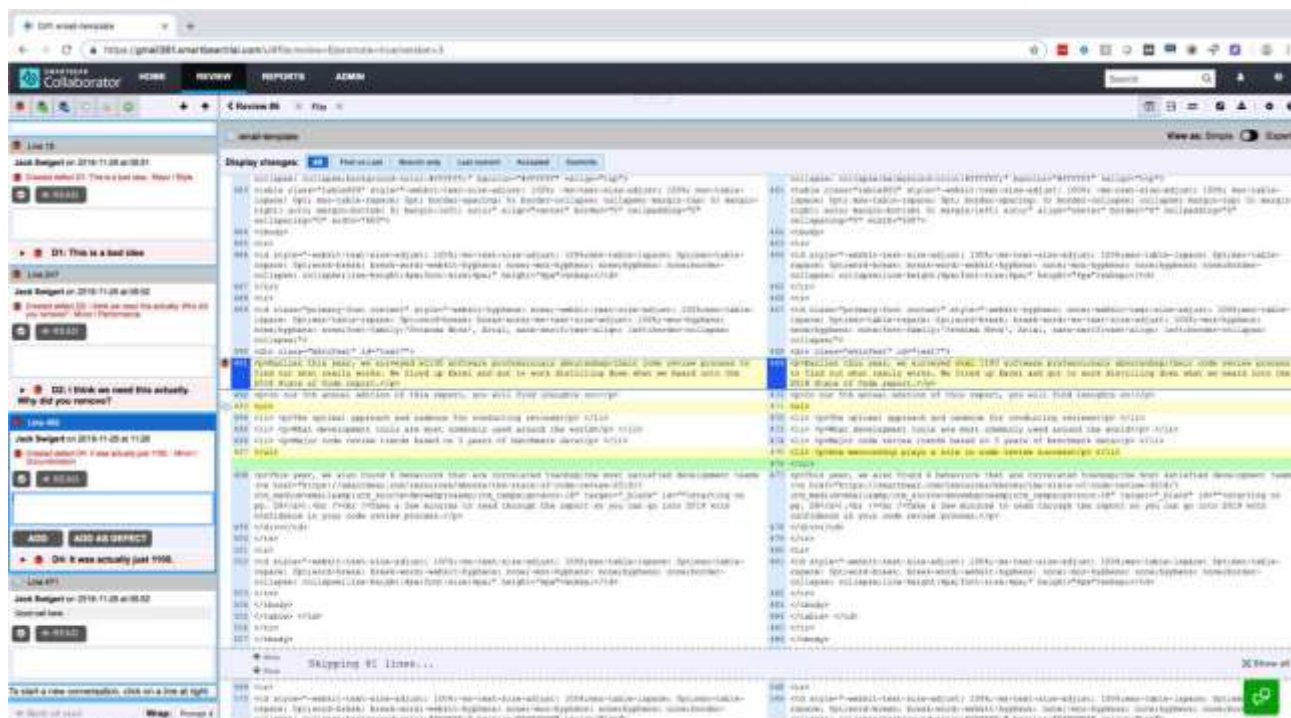
- Reshift 是一个基于 SaaS 的软件平台，它无缝地集成到软件开发工作流中，让企业可以持续地部署安全的软件产品，而不会减慢它们的速度。Reshift 减少了查找和修复漏洞、识别数据泄露的潜在风险以及帮助软件公司实现合规性和法规要求的成本和时间。

代码评审工具

- Collaborator

<https://smartbear.com/product/collaborator/>

是面向团队的代码和文档评审工具，支持主流软件配置管理和版本管理平台，集成了对GitHub中Pull Request的评审处理机制，还能对接Jira系统中的问题单。



代码评审工具

- Gerrit

<https://www.gerritcodereview.com/>

是一套开源的轻量级代码审查工具，与Git无缝集成，提供对IntelliJ IDEA的插件支持，能够在开发环境中对被评审的代码添加评论并打分，通过多人协同的方式判断代码能否通过评审并提交到目标分支上。

