【软件构造】第四章第一节 面向可理解性的构造

第四章第一节 面向可理解性的构造

Outline

- 代码可理解性
- 编码规范

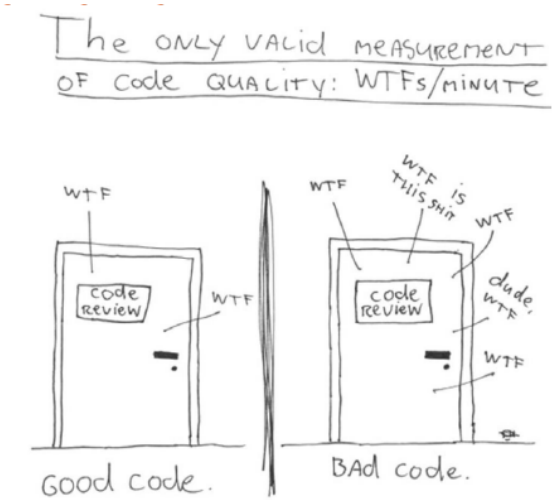
Notes

代码的可理解性

代码的可理解性可以理解为代码的可读性。具体来说，可从以下几个方面来看：

- 是否遵循特定的命名规范？
- 是否足够的注释/说明？
- 是否足够的内聚性？
- 方法是否太长或太短、是否容易理解？

【代码质量测量：**WTFs / min**】



简单来说，好的代码让人更容易理解，其他程序员在阅读时就会发出更少的WTF（What the fuck）来抱怨你的代码质量，而不好的代码则更加让人暴躁抓狂。

【标识符名称长度】

标识符的长度包含类名、变量名、方法名等的长度。主要看以下方面

- 是否具有自描述性？不看注释就可以理解其含义
- 是否足够简洁？太长的变量名降低效率

度量方式：所有标识符的平均长度。

【命名独特性比例（**UNIQ**）】

当两个实体名称相同时，它们可能会混合在一起。 UNIQ衡量所有名字的独特性。在许多地方使用相同的名称是可以接受的。然而，这个名字应该是指同样的逻辑事物。

【代码复杂度和代码行数】

复杂的代码不可能被理解。
一个方法越长，它可能越难理解。

【注释的密度 **MCOMM%**】

代码中的注释越多，阅读和理解越容易。

【如何编写易于理解的代码】

- 遵循命名规范
- 限制代码行的最大长度、文件的最大LoC
- 足够的注释
- 代码有好的布局：缩进、空行、对其、分块、等。
- 避免多层嵌套—增加复杂度
- 文件和包的组织

代码的可读性/可理解性很多时候比效率/性能更重要，不可读、不可理解代码可能蕴含更多的错误。 因此先写出可读易懂的代码，再去逐渐调优！

【可读性强的语句的栗子】

Example A：z = ((3*x^2) + (4*x) - 5) - ((2*y^2) - (7*y) + 11) / ((3*x^2) + (4*x) - 5)

Example B：a = (3*x^2) + (4*x) - 5; b = (2*y^2) - (7*y) + 11; z = (a - b) / a

B的代码可读性强于A

编码规范

- 编码规范：定义了一系列的规则，按这些规则进行编码，有助于提升代码可读性，例如—— 命名、代码布局/缩进、数据声明方式、文件组织方式 、etc。

【命名】

- 变量，函数或类的名称应告诉你，它为什么存在，它做了什么以及如何使用它。
- 如果名称需要注释，则名称不必显示意图：

```
int d; // elapsed time in days
int elapsedTimeInDays;
int daysSinceCreation;
```

- 需要注意一下内容：
  - 避免造假：避免留下模糊代码含义的虚假线索。例如仅当`accountList`实现`List`时才使用`accountList`; 其他情况下`accounts`更好。
  - 做出有意义的区别
  - 使用可发音名称

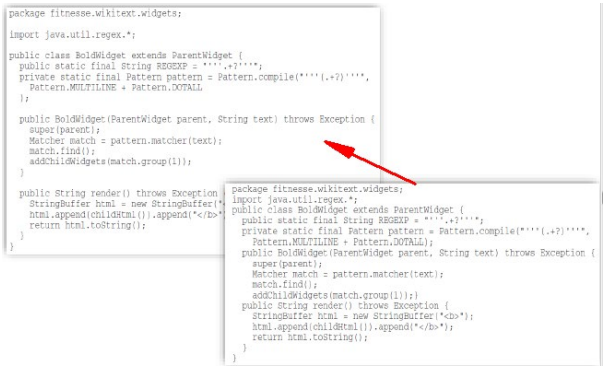
使用可搜索的名称

- 命名规范：
  - 包名称应该是小写的；
  - 类和接口名称应该是名词和大写；
  - 方法名称应该是动词并以小写开始；
  - 常量命名：所有大写字母之间带下划线；
  - 参数命名，确保你的参数意味着什么。

【行数限制】

- 限制每一行的长度；
- 每一个方法大概三十行，在一页内实现；
- 每一个文件大概200行，最多不超过500行。

【垂直格式化：空行】



- 单行空行
  - 在局部变量声明 和 方法中的第一个代码之间
  - 在块注释之前
  - 在代码的逻辑段之间提高可读性
- 双行空行
  - 在方法之间
  - 在类和接口声明之间
  - 在源文件的任何其他部分之间
- 垂直密度意味着密切关联，因此密切相关的代码行应该垂直密集。
- 密切相关的概念应该保持垂直相互靠近。
- 他们的垂直分离应该衡量每个人对另一个人的可理解性有多重要。

【横向格式化：空格】

我们使用水平空白区域来关联强烈关联的事物，并将与强调它们的关系更加微弱的事物分开。

【横向格式化：缩进】

代码必须根据其嵌套级别进行缩进。你可以选择缩进量，但应该保持一致。不好的缩进会使程序难以阅读，也可能成为一个难以理解的错误来源。

【横向格式化：换行】

当一个表达式不适合单独一行时，根据以下一般原则将它换行：

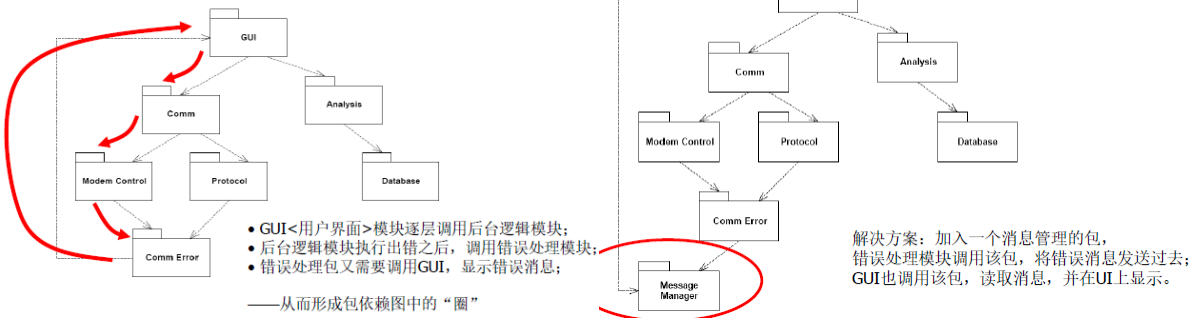
- 逗号后换行

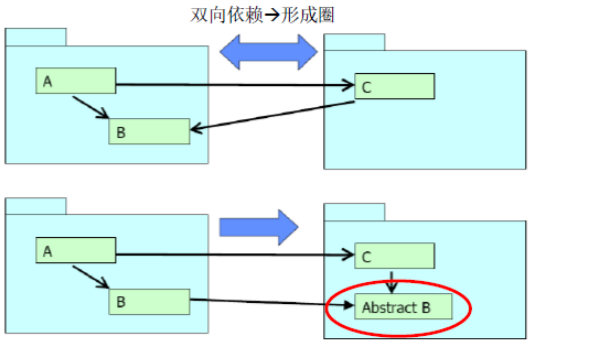
在操作符前换行

- 将新行与上一行中相同级别的表达式的开头对齐。

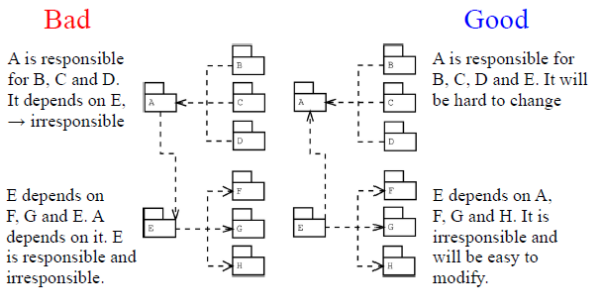
【文件组织】

- 在源文件中的顺序：
  - 包或文件级别的注释
  - 包和导入的说明
  - public的类和接口的声明
  - private的类和接口的声明
- 导入说明的顺序：
  - 标准包（`java.io, java.util, etc`）
  - 第三方包（例如`com.ibm.xml.parser`）
  - 你自己的包
- 类部分的顺序：
  - Javadoc 注释
  - 类声明的说明
  - 整个类的注释
  - 类静态变量声明（public，protected，package，private）
  - 类实例变量声明（public，protected，package，private）
  - 函数声明（构造函数优先）
- Principles of Package：
  - 复用/发布等价原则（REP）
    - 复用的粒度应等价于发布的粒度
    - 粒度：粒度大的复杂度大，粒度小的可以高度重用。
  - 共同封闭原则（CCP）
    - 一个包中的所有类针对同一种变化是封闭的；
    - 一个包的变化将会影响包里所有的类，而不会影响到其他的包；
    - 如果两个类紧密耦合在一 起，即二者总是同时发生变化，那么它们就应属于同一个包。
  - 共同复用原则（CRP）
    - 一个包里的所有类应 被一起复用；
    - 如果复用了其中一个类，那么就应复用所有的类
- Principles of Package Coupling：
  - 无圈依赖原则 （ADP）
    - 不允许在包依赖 图中出现任何圈/回路；
    - 无圈将容易进行测试、维护与理解；
    - 若存在回路依赖，很难预测该包的变化将会如何影响其他包。
    - 消除圈的两种方式：创建新包和利用DIP<依赖倒置原则>和ISP<接口隔离原则>。
      - 创建新包





    - 利用DIP<依赖倒置原则>和ISP<接口隔离原则>
- 稳定依赖原则（SDP）
    - 包之间的依赖关系只能指向稳定的方向；
    - 被依赖者应更稳定于依赖者；
    - 稳定的包较难发生改变；
    - 如果不稳定的包却被很多其他包依赖，会导致潜在的问题。



- 稳定抽象原则（SAP）
    - 在稳定性与抽象度之间建立关联；
    - 一个包是稳定的，那么它就应该尽可能抽象；
    - 一个完全稳定的包中只应包含抽象类；
    - 不稳定的包应是具体的，以便于容易的进行修改。

- SAP与SDP比较：
    - SAP和SDP共同构成了包之间的“ 依赖倒置原则DIP”；
    - SDP: 依赖应指向稳定的方向，SAP: 稳定性隐含着抽象；
    - 因此，依赖应指向抽象的方向。