**宏天软件X5 编程规范**

# 介绍

编码规范对于程序员而言尤为重要，有以下几个原因：

* 一个软件的生命周期中，80%的花费在于维护
* 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护。
* 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
* 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

为了执行规范，每个软件开发人员必须一致遵守编码规范。

# 注释

Java程序有两类注释：文档注释(document comments)和实现注释(implementation comments)。

**文档注释：**由/\*\* .../界定。文档注释可以通过Javadoc工具转换成HTML文件。

**实现注释**：也称**普通注释**。用以注释代码或者实现细节。普通注释是为了帮助开发者和阅读者更好地理解程序，不会出现在API文档中。其中，单行注释以“//”开头；多行注释以“/\*”开头，以“\*/”结束。

注：要求每个类、方法、变量都要有相应的注释，而且方法的注释必须把这个方法的逻辑思路讲清楚。

## 文档注释

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过2000行的程序难以阅读，应该尽量避免。

在实际开发中，我建议对超过1000行的类需要进行重构，包括合并重复代码，将不属于该文件的方法分离到其它的文件中等。

每个Java源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

### 开头注释

/\*\*

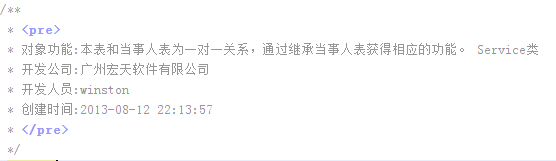
\* 版权声明

\*/

这里基于公司的授权条款统一制定，各个文件相同，代码生成时统一生成。

### 类注释

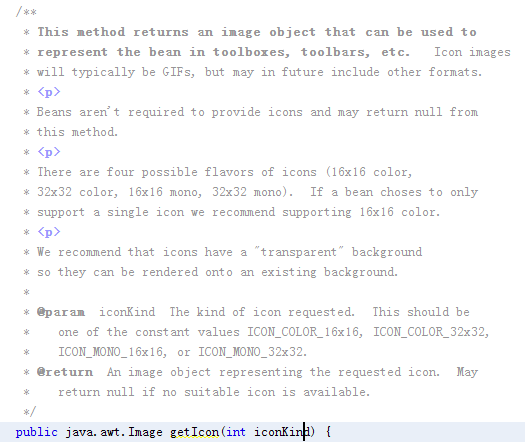
**示例：**



通过代码生成器生成的代码，默认包含这样的注释，人工创建的类需要自行添加。

### 方法的注释

**原则：**以“/\*\*”开始，“\*/”结尾注释。写明程序作用，以及输入输出参数，和异常。**例如：**



1. 程序作用：多行用<p>分隔；
2. @param 参数名，需补充的说明：它的含义和使用要点。
3. @return 返回值。

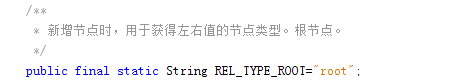
需补充的说明：它的含义和某些特殊情况。

### 属性的注释（多用于常量）

**原则：**以“/\*\*”开始，“\*/”结尾注释。写明该属性的含义。

这类属性将会出现在Javadoc中。

**例如：**



### 修改程序时不必注释

由于现在已经使用了SVN等源码版本管理工具，所以在修改代码时，不需要在代码中增加日志类的注释，免得代码中存留大量的日志和废弃代码，影响阅读。

**注**：这里要求在提交文件到svn服务器时，在提交消息中务必将“**修改的内容**”完善。

关于svn使用规范，详见第8章。

## 实现注释

程序可以有4种实现注释的风格：

* 块(block)
* 单行(single-line)
* 尾端(trailing)
* 行末(end-of-line)。

实现注释采用/\*…\*/或者//，该类注释不会出现在javadoc中。

### 块注释

块注释通常用于提供对文件，方法，数据结构和算法的描述。

块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释的首位应该有一个空行，用于把块注释和代码分割开来，比如：

/\*  
   \* Here is a block comment.  
   \*/

### 单行注释

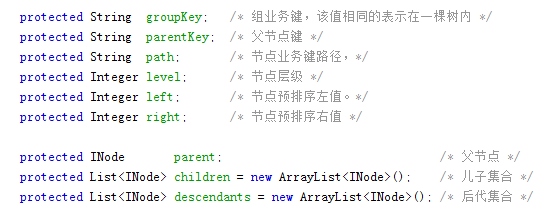
短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见"块注释")。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

if (condition) {  
 /\* Handle the condition. \*/  
 ...  
}

### 尾端注释

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

示例：



**要点**：

1. 每个/\*\*/的内容部分的左边和右边留一个空格，避免文字和 \* 太近影响阅读。
2. 同一组内的 /\*\*/ 左侧对齐，方便视线垂直往下阅读。

### 行末注释

注释界定符"//"，可以注释掉整行或者一行中的一部分。

它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

【注】对于废弃的代码，建议删除，而不是注释掉。

这些代码可以通过svn来找回以前的版本进行对比。

# 格式

## 格式的目的

或许你认为“让代码能工作”才是专业开发者的头等大事，但我希望我们的团队成员能够抛掉这种想法。

我们今天实现的所有功能，都有可能在交付客户后，给客户的开发人员进行修改和扩展，代码的可读性会对以后可能发生的修改行为产生深远的影响。

代码的整洁、一致性能够在第一次被阅读时就表现出我们对细节的关注，从而让客户感受到我们的专业。而反过来，如果他们看到的是一堆鬼画符般的代码，那他们多半会认为我们对项目的其它方面的细节也漠不关心。

细节决定成败。

## 垂直格式

### 垂直方向的间隔

在包声明、导入声明和每个方法之间，都用空白行隔开。

这条极其简单的规则极大的影响到代码的视觉外观。每个空白行都是一条线索，标识出新的独立概念。往下读代码时，你的目光总会停留在空白行之后的那一行。

### 垂直方向的靠近

如果说空白行隔开了概念，那么靠近的代码则暗示了它们之间的紧密关系。

紧密相关的代码应该互相靠近，更容易“一览无遗”。

概念相关的代码应该放在一起，或者距离相近。相关性越强，彼此之间的距离就该越短。

## 横向格式

### 缩进

4个空格作为缩进排版的一个单位。

### 行长度

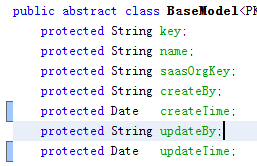
一般情况避免一行的长度超过80个字符。

但实际开发中，也不需要限制在这个数量，超出几个字符问题不大，但如果超过90或100，则应该使用换行了。

### 水平对齐

对于一组声明中的变量名，或者一组赋值语句的右值。请尽量对齐。如：

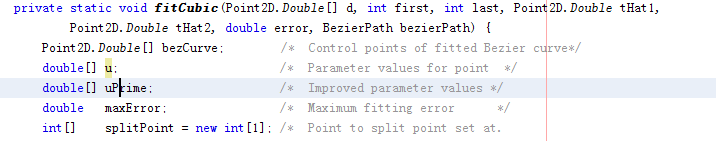
例子1：



但这个规则在变量增加了文档注释后就变得作用不大了，因为文档注释增加了垂直距离，让各个变量的关系变得不那么紧密了。

我建议只针对常量使用文档注释，变量使用实现注释。

例子2：



对于上例，如果第2～5个变量和第一个变量对齐，则变量名和类型距离太远，反而影响阅读，所以只处理2～5变量的对齐。

对于注释则全部对齐。

### 换行

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开：

* 在一个逗号后面断开

如3.3.4.1例子1

* 在一个操作符前面断开

如3.3.4.2的 ||

* 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开

即按分组的关系断开。

如3.3.4.1例子2的Double(

* 新的一行应该与上一行同一级别表达式的开头处对齐

如3.3.4.1例子2的Math.\*

* 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进8个空格。

如3.3.4.1例子2的Rectangle2D.Double

以下是断开方法调用的一些例子：

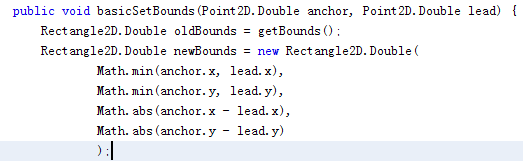
#### 多参数换行

* 例子1：



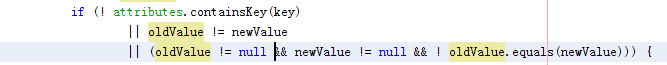
将逗号留在上一行的末端。

* 例子2：

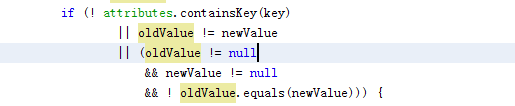


一行一个参数的处理。

#### 条件的换行

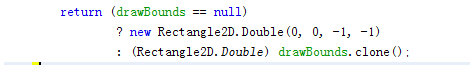


* 每一个或条件段独占一行。
* 第三行为一个组合条件，独占一行。如果过长，则进行二级分行。如：



#### 三元运算的分行

一般而言，三元运算不需要分行。但当该行过长时，可以考虑分行，分行的示例如下：

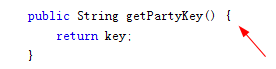


## 代码块

一个代码块是指任何被包含在大括号"{"和"}"中间的代码。

代码块的起始{ 有两种放置方式：1}行尾；2）另起一行并独占一行。

为了保持整体风格统一，并且**减少不必要的行数**，我们统一将起始{放在行尾，如：



# 声明

## 一行一个变量

推荐一行一个声明，因为这样以利于写注释。亦即，

int level; // indentation level  
  int size; // size of table

尽量在声明局部变量的同时初始化。

唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

## 在代码块首声明变量

只在代码块的开始处声明变量，不要在首次用到该变量时才声明之，因为这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

Public void myMethod() {  
   int int1 = 0; // 在方法开始处声明

  if (condition) {  
     int int2 = 0; // 在if代码块开始处声明  
           ...;  
     }  
 }

该规则的一个例外是for循环的索引变量

for (int i = 0; i < maxLoops; i++) { ... }

## 避免同名变量覆盖

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

int count;  
   ...  
  myMethod() {  
   if (condition) {  
   int count = 0; // 不要这样做!  
      ...  
    }  
    ...  
  }

# 语句

## 一行一条语句

每行至多包含一条语句，例如：

argv++; // 正确  
  argc--; // 正确

argv++; argc--; // 错误!

## 返回语句

一个带返回值的return语句一般不使用小括号"()"，除非它们以某种方式使返回值更为显见。

下面这些例子是被允许的：

return size;

return myDisk.size();

return (size ? size : defaultSize);

## if语句代码块

注意：if语句总是用"{"和"}"括起来，避免使用如下容易引起错误的格式：

if (condition) //内部的代码只有一行，语法正确，但容易误解，避免使用。  
   statement;

## switch语句注释

一个switch语句应该具有如下格式：

（case的值建议使用常量，这样可读性会更好）

switch (condition) {  
 case ABC: /\* 说明该case的含义 \*/  
     statements;  
 /\* 需要穿越到下一个case时（不break），这里需要写注释说明原因\*/  
    case DEF:  
      statements;  
      break;

  case XYZ:  
      statements;  
      break;

  default:  
      statements;  
      break;  
  }

# 命名

## 基本规范

系统中主要使用驼峰式命名。

即多个单词构成的名称，第一个单词第一个字母小写，其它的单词第一个字母大写。

比如orgName，processCmd等等。

但对于一些方法的传入参数或局部变量则可以根据实际需要来选择驼峰式命名或下划线分隔。

比如weight\_g（表示重量，按克计算），start\_ms（开始时间，用毫秒ms表示）等。

## 变量命名

变量的命名有两个关键思想：

1. 把信息装到名字里

让变量的名字本身就具备准确的信息，而不需要反复查看上下文代码。

1. 选择不会引起误解的名字

这是对第一点的补充，有时信息很多，但是却带来歧义，就过犹不及了。

### 把信息装到名字里

1. 选择专业的词

下面的6.6章提供了常用的单词和说明。单词的选择关键在于动词，名词比较容易选。

1. 避免泛泛的名字

避免使用如abc、a1、a2、tmp、temp、retval、flag、foo等等没有明显含义的名字。

除了某些特定的情况，如for循环中使用i，冒泡算法中的temp等，但是绝大多数情况，请起一个有意义的名字。

1. 用具体的名字代替抽象的名字

比如fruit（水果）对比apple、orange；vehicle（交通工具）对比car、bike等；

除非抽象的名字用在抽象类或者接口中。

1. 为名字附加更多的信息

诸如id、name、number、key等等变量，如果不是处在实体类中，而是某个方法内的局部变量，那么这些名字的可读性很有限。增加一些附加信息会避免误解。

如orgId、orgName、repeat\_number、orgKey等等。

1. 增加内容的限定

有些名字具备一些默认的含义，比如password，默认就是加密的密码，但是如果你希望传递的是明文的密码，那么最好加上一些限定会避免误解，比如

plaintext\_password 或者 plaintextPassword。

1. 增加单位描述

这里主要针对数值，比如某个数值表示时、分、秒、毫秒、千克、克或者其它计量单位，那么单纯一个weight、start、end等等，不足以描述清楚，而且如果弄错了单位，可能导致其它的错误。这时候可以加上单位：

如：weight\_kg、weight\_g、start\_ms、end\_s等等。

1. 关于缩写

不要随便缩写，很容易造成理解困难。

在6.6.1中提到缩写规则，在6.6.2/6.6.3等表格的常用单词有些提供了缩写，请参照使用。如果自行做了缩写，请在方法的注释中说明缩写对应的单词。

1. 关于长名字

长名字带来丰富信息的同时，某些开发人员会认为它难以输入。但现在的IDE已经提供了很多快捷方式。比如NetBeans的：

* 代码模板（输入sout，按空格，自动转成System.out.println(“”);
* 补全或提示：输入前几位字母，输入ctrl+空格，会自动补全或提示。

### 选择不会引起误解的名字

1. 模棱两可的一些名字

避免使用有多重含义的名字。如：

* Filter：是将满足条件的过滤去掉，还是过滤后留下？

除非在方法里面有更明确的表示。

* Length：长度。单纯的length不足以表示足够的信息，增加一些补充信息才可以。如max\_chars\_length（这个名字可以去掉length，用max\_chars）。

1. Min和Max表示极限
2. First和last表示包含的范围，多用于数值。
3. Begin和end表示时间范围。
4. 给布尔值命名。

比如某个方法的返回值是boolean，那么在方法体内return true/false 比不上将布尔值换成有意义的名字。如return canVisit; return isAdmin;等等。

1. 避免采用反义。

比如isNotXXX。

反义的反义是更加要避免的，如: !isNotEmpty()，这种情况请务必改为isEmpty();

## 包命名

格式如下：

* 前缀

com.[company].[system].[module]

如：com.hotent.core.db

com.hotent.user.party

前缀由项目经理或架构师制订。

* 细分

com.[company].[system].[module].dao

com.[company].[system].[module].mapping

com.[company].[system].[module].model

com.[company].[system].[module].service

com.[company].[system].[module].controller

com.[company].[system].[module].util

即每一个module（模块）下面，按不同的功能层级划分子包。

上面是常用的包名称，按实际情况，经过讨论可以增加。

## 类命名

类的名字必须由大写字母开头而单词中的其他字母均为小写；

如果类名称由多个单词组成，则每个单词的首字母均应为大写例如PartyService；

如果类名称中包含单词缩写，则这个所写词的每个字母均应大写，如：XMLUtils。

还有一点命名技巧就是由于类是设计用来代表对象的，所以在命名类时应尽量选择名词。长度不超过18位

### 类名

按照类的作用和所属架构层次进行不同的命名方式：

Dao：[实体名]Dao.java

Service: [实体名]Service.java

Model: [实体名].java

Controller: [实体名]Controller.java

等。

对于不是通过表直接生成的类的命名方式，则采用：

[实体/业务名称][操作/处理][层次后缀(Service/Controller等)].java

的格式进行命名

### 抽象类

系统的抽象类基本在hotent\_core项目中定义，使用范围并不广，为了避免命名过长，并没有在命名中体现。

### 接口类

为了特别注明接口类，通常给类加上一个前缀I。

但如果该项目或者模块下面全部是接口类，那么为了简化命名，也可以不增加I前缀。

### 实现类

接口类本身是不能被实例化的，其包含的操作方法是在其子类中实现，常称为

"实现类"。

实现类分为两大类：

* 继承方式的：

如通过代码生成器生成的Dao、Service等类，不继续划分接口和实现类，所以无需加Impl后缀。

如PartyDao、OrgDao等。

* 实现接口方式的：

如定义某个接口，然后实现该接口，则给该实现类加上一个后缀Impl。

如：MyBtaisDaoImpl、BaseTreeDaoImpl等。

### 数据库实体类

为了减少实体类的数据，在X5中，将数据库实体类和业务实体类合并处理。详见业务实体类。

### 业务实体类

一般将表名去掉模块前缀，然后转换为驼峰式命名。

如party\_org\_ 对应的实体类为：Org.java

对于某些表去掉前缀后会导致理解困难的，则保留前缀，如

party\_type\_，对应的实体类为：PartyType.java（而不是Type.java）

## 类方法

### 动词+名词

类操作方法的名字的第一个单词应以小写字母作为开头（动词），后面的单词则用大写字母开头。

一般为：

[动词][名词][s]

如

sendMessage();

sendMessages(); /\* 表示复数 \*/

### 是否方法

* is
* has
* allow

等单词开头的方法，返回值一般为布尔型。

“是否”的事务控制为只读。

### 获取

get 开头，表示获取单个实体。

“获取”的事务控制为只读。

### 查询

* query：查询。
* find：查找。

两者均可以作为查询方法的前缀，区别不大，在基础代码中采用query，所以开发人员在自定义的方法中可以用find，以便和框架的方法进行区分。

“查询”的事务控制为只读。

## 常用命名的中英文对照表

### 缩写规则

下面的缩写规则非英语专业使用的规则，而是为了编程规范和方便设置的规则。

待缩写的单词：

1. 有通用缩写：采用；
2. 无通用缩写：
   1. 单词小于或等于6位，无需缩写。
   2. 当整个名称的单词小于或等于3个，不使用缩写；
   3. 当整个名称的单词超出3个，使用单词的前四位字母作为缩写代替。然后在注释中说明。

### 动词的中英文对照表

考虑到可读性，方法命名中，动词一般不做缩写，除非特别长的单词（对于这种情况，一般采用其它同义的较短单词代替）

* **首选动词**

|  |  |  |  |
| --- | --- | --- | --- |
| 英文 | 缩写 | 中文 | 使用说明 |
| **数据库查询** | | | |
| query |  | 查询 | 标准的查询方法前缀 |
| find |  | 查找 | 也可以用于查询方法。开发人员自定义的查询方法可以以此开头以示区分。 |
| Count |  | 计算数量 |  |
| get |  | 获取 | 【注】这并不是一个专业的单词。  只适合两种情况使用：  1、getter  2、dao和service层提供的获取单个实体的方法。 |
| **页面呈现和处理** | | | |
| display |  | 显示 | 用于复杂页面的呈现 |
| initialize | init | 初始化 | 一般针对复杂对象。将初始化处理封装到init方法中，在工厂或者构造函数中调用。  也可以用于复杂的业务流程的初始化页面。 |
| Manage |  | 管理 | 用于呈现管理界面 |
| add |  | 增加 | 新增页面的呈现，多用于控制层。 |
| edit |  | 编辑 | 编辑页面的呈现，多用于控制层。 |
| validate |  | 验证 |  |
| print |  | 页面打印 |  |
| **持久化处理** | | | |
| create |  | 创建 | 创建单个实体 |
| update |  | 更新 | 更新单个实体 |
| save |  | 保存 | 复杂业务的保存处理。在里面可以涉及创建和更新等操作。 |
| remove |  | 删除 | 将某个实体从持久化层中移除。 |
| rebuild |  | 重建 | 一般针对树或索引。涉及若干条记录的计算和更新。 |
| do |  | 操作 | 对于没有更合适的单词，可以选择用do。但一般不建议。 |
| import |  | 导入 | 分为导入到数据库或文件系统。 |
| export |  | 导出 | 导出为指定文件格式。 |
| **内存处理** | | | |
| set |  | 设置 | 更改对象的某个变量的值。 |
| clear |  | 清空 | 将集合清空，将某个对象恢复到初始化阶段，但该集合或对象不为null。 |
| dump |  | 写入 | 把内容从内存写入文件中 |
| refresh |  | 刷新 | 一般为在内存中刷新，涉及重新读取数据。  方法为只读。 |
| **信息处理** | | | |
| send |  | 发送 | 消息的发送 |
| receive | rec | 接收 | 消息的接受 |
| retry |  | 重试、重启 | 重发消息 |
| format |  | 格式化 | 将指定内容进行某些方式的格式化处理。  如format[指定内容](格式参数) |
|  |  |  |  |
| **规则计算** | | | |
| calculate | calc | 计算 | 数值、金额等的计算，偏向数字。 |
| sum |  | 总计 | 统计总数。Cal的具体化。 |
| evaluate | eval | 评估 | 规则的计算，偏向业务。 |
| execute | exec | 执行 | 适合执行脚本或某些规则。和eval的区别在于前者重在分析处理，exec在于执行。 |
| **业务流程** | | | |
| start |  | 开启 |  |
| launch/begin/open |  | 发动/开始/开启 | 作为start的补充，再更适合的场景中可以选用。 |
| stop |  | 停止 |  |
| run |  | 运行 |  |
| fork |  | 分支 |  |
| complete |  | 完成 |  |
|  |  |  |  |

* **备选动词**

|  |  |  |  |
| --- | --- | --- | --- |
| 英文 | 缩写 | 中文 | 使用说明 |
| show |  | 显示 | Display的备选 |
| modify |  | 修改 | edit的备选 |
| delete |  | 删除 | remove的备选 |
| finish |  | 完成 | complete的备选。统一用complete。 |
|  |  |  |  |

### 名词的中英文对照表

|  |  |  |
| --- | --- | --- |
| 英文 | 缩写 | 中文 |
| file |  | 文件 |
| dir |  | 目录 |
| host |  | 主机 |
| website |  | 网站 |
| configuration | conf | 配置 |
| server |  | 服务器 |
| service |  | 服务 |
| user |  | 用户 |
| role |  | 角色 |
| resource | res | 资源 |
| organization | org | 组织、单位 |
| operation |  | 操作 |
| permission | perm | 权限 |
| security |  | 安全 |
| exception |  | 异常 |
| table |  | 表格 |
| error |  | 错误 |
| message |  | 消息、通知 |
| sum |  | 总数 |
| context |  | 上下文。  如登录、应用或者某个特定功能的上下文封装处理。 |

### 助词的中英文对照表

|  |  |  |
| --- | --- | --- |
| 英文 | 缩写 | 中文 |
| abstract |  | 抽象的 |
| first |  | 第一个 |
| last |  | 最后一个 |
| next |  | 下一个 |
| previous | prev | 前一个 |
| current | curr | 当前 |
|  |  |  |

# 单元测试

## 重要性和责任

程序员的价值在于和他人合作，开发出高质量的代码，而不是开发出一堆虫件（bugware）。

程序员必须对自己的代码质量负责，单元测试是对自己代码质量的基本承诺。

程序 = 单元测试（UT） + 代码（CODE）

对于组件化的项目，除了1~2个子项目是可以通过界面进行测试之外，其它的项目均是java代码。如果一个bug不能在单元测试阶段被发现，而是在集成测试，甚至是系统测试阶段才被发现，那么排查和修复成本将是非常高昂的。

不做好单元测试，就会影响团队其他人员的工作。

测试人员有权利对没有做过单元测试的代码说**不**。

其他的开发人员也有权利对没有做过单元测试的代码说**不**。

项目经理在日常管理工作中，单元测试是重要的考察部分，也是界定出错责任的关键依据。

## 两种UT方式

1. 传统方式。编程，测试和改良。

流程如下：

编写程序单元 🡪 编写UT代码 🡪 测试 🡪 改良程序 🡪 调整UT代码 🡪 继续测试 🡪 【反复的过程】 🡪 所有业务全部实现和相应的单元测试都通过。

1. TDD 测试驱动开发

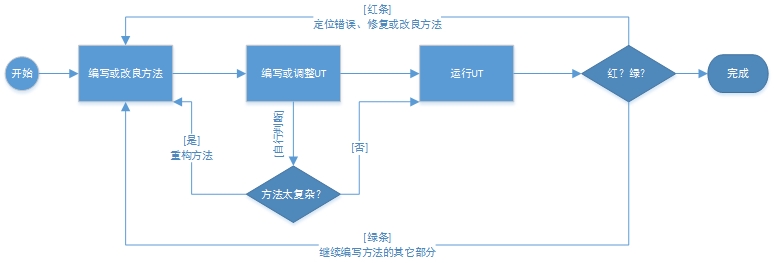
流程如下：

根据需求先写UT 🡪 编写程序单元满足UT的通过 🡪 调整UT，增加更多的业务逻辑测试 🡪 丰富程序单元再次满足UT 🡪 【反复的过程】 🡪 单元测试覆盖所有业务并全部测试通过。

注意，两种测试方式都必须保证public和protected的方法都有相应的UT方法。

对于底层框架的公共方法，则不需要在每一个子类都重复测试。

## 传统方式



1. 写一点方法、写一点UT

不要等待方法全部完成后才写UT，而是在该方法的每一部分完成后就进行UT，让UT贯穿方法编写的全过程。

1. 编写UT

程序员每编写完一个程序单元，就编写UT代码。

1. 运行UT

通过运行UT代码来定位程序的错误，而不是直接调试代码，节约了大量调试的时间。

1. 修复和改良方法

修复和改良方法，以通过UT。

1. 方法太复杂？——重构方法

对代码较多(注)的方法进行重构，将其分解为若干个较小的方法，并独立的进行UT。

每个小方法只处理一个特定的业务，这样比UT一个大的方法要简单和更容易覆盖全面。

【注】什么是代码较多没有硬性标准，一般超过40行就需要考虑是否可以进行重构了。

按照上面的方式，方法的错误能尽早的发现，不需要等到Code Review或者集成测试才被发现和修复，在这个阶段修复错误成本很低。

## TDD 测试驱动流程

### 原理

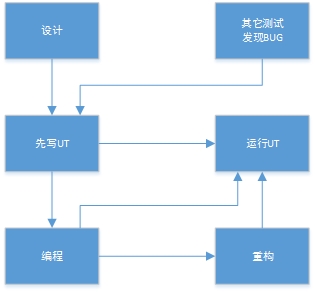
测试先行会确保所写的代码就是功能所需要的代码。（由于之前的业务需要中已经对验收测试进行了定义，我们使用单元测试的断言就可以确保实现的功能是符合要求的）

如果代码通过了单元测试，我们就可以放心地使用这部分的代码。

使用测试先行，可以更好地帮助你考虑怎样设计和编写类，因为你会考虑类或方法将被 怎样调用，以及怎样才能方便进行单元测试。

测试先行，它可以使你自信地进行重构，因为你可以很快地通过JUnit的单元测试对重构过的代码进行测试，确保重构的代码在功能上与原来的版本是一致的。

### 流程



1. 设计

设计接口方法或者在类设计空的方法。

初步制订输入参数和输出参数。

1. 先写UT

调用该接口，构造输入参数、测试输出参数等。

依照需求设计，编写UT的内容，通过UT反过来决定生产代码的编写。

注意步子要小。

1. 根据UT编写生产代码
2. 运行UT

测试新编写的生产代码，并增加新的测试代码。

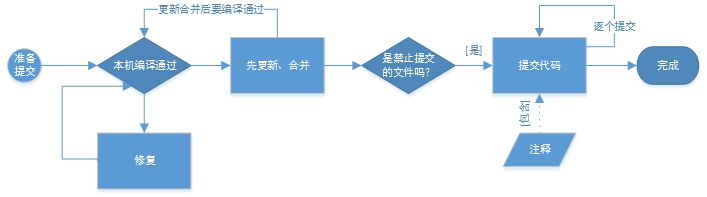
1. 重构

重复2~4的过程，如果这个过程中发现：

1. 第一步设计不够理想，则可以返回对设计进行重构。
2. 发现生产代码不够理想，则对生产代码进行重构。

重构后相应的调整UT。

# SVN使用规范



1. 严禁使用其他人的账号提交代码。
2. 准备阶段
3. 本机是否编译通过？

本机不能编译通过，不允许做提交操作。

1. 这些代码自己是否能够看明白？

提交的代码必须是原子性的，如完成了某个功能，修复了某个bug，做了某个页面调整等。

对于只是自己使用的代码，比如private方法，可以阶段性完成即提交，但最起码自己看得明白，不是垃圾代码。

1. 先更新、合并代码

每一次要提交前，先查看要提交的文件在SVN服务器上是否有修改。

如NetBeans就是右键点击要提交的目录，选择Subversion 🡪 显示更改。

1. 更新合并后，是否编译通过？

如编译不通过，则调整代码或更新编译依赖的其它的文件。

1. 禁止提交的文件
2. 本地IDE自动生成的文件；
3. 本地测试的文件，如uploads的图片、附件等。（这些目录应该设置为svn ignore）
4. 配置文件的本地化修改。比如app.properties，使用了本机的数据库。
5. 提交代码 —— 逐个文件提交！

除了下面这些例外，其它的文件请逐个提交：

1. 第三方组件的新增或者升级。（注：我方做个性化重构时需要逐一提交）
2. 某个目录的文件全部是新增的。
3. 样式设计和设计图片。
4. 每次提交必须书写明晰的标注

逐个文件提交，单独写标注，记录调整历史，方便其他开发人员更新和理解。

1. 描述本次提交的文件变更内容。
2. 如果修复了JIRA的bug，请附上JIRA的BUG ID。

（注：需要配置JIRA和SVN的整合）

1. 发布版本周期要短 —— 管理员打Tag

这要求项目经理将每个小版本的待开发功能控制在一定数量内，保证周期要短，这样才可以经常打tag发布版本。

步子要小，这样项目才更加可控，每个版本的稳定性才更能保证。

# 代码质量审查

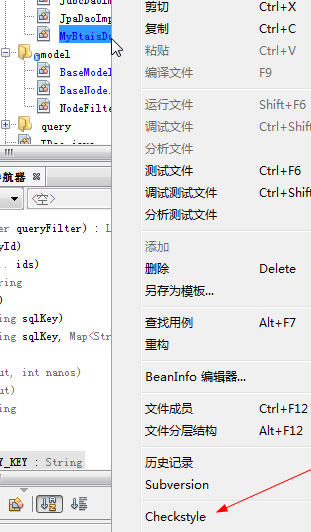
## Checkstyle

<http://plugins.netbeans.org/PluginPortal/faces/PluginDetailPage.jsp?pluginid=4552>

点击download下载，然后：

NetBeans 🡪 工具 🡪 插件 🡪 已下载，选择该nbm，安装即可。

然后右键点击Java文件，出现：



Checkstyle的检查结果有些过犹不及，但还是有不少参考价值的。

## Java代码审查表

TODO