Java 编码规范及实践

第1章 概述

1.1 前言

代码之于程序员，就像零件之于机械工，庄稼之于农民，它是软件的基石，一行行代码都是程序员的心血经过日日夜夜凝结成的。做为一个程序员，应该像母亲呵护孩子一样呵护自己的代码，它不仅仅是一行一行的文字，它是一个程序员的尊严和价值所在；它是活的，你甚至能感受到它的心跳。编码规范只是大家达成一致的约定，这样大家的代码就可以互相看懂，维护起来更加容易，思想更畅快的交流，经验更快的得到传播。代码规范不是束缚程序员的桎梏，应该知道，不遵守规范的个性的代码并不代表程序员的性格，并不能张扬个性。个性应该体现在用更简单、更优雅、更易读、更易理解以及算法实现效率更高等方面。

可读性，可理解性是代码的重要方面，本规范主要围绕如何去产生规范易读的代码。另外，它也保证了大家有共同的先验知识。

1.2 术语

- **Pascal case** － 所有单词第一个字母大写，其它字母小写。

  例：Person, OrderDetail, OilTank。

- **Camel case** － 除了第一个单词，所有单词第一个字母大写，其他字母小写。

  例：oilLevel, customerName

在后面的描述中我们使用 Pascal 代表第一种表示方式， Camel 代表第二种表示方式。

1.3 约束

没有一个规范可以到处适用，也不可能无所不包。

1. 本规范并非完全强制性规范，对于任何违背本规范但能提高代码可读性的措施，都可以采用。
2. 本规范第 3，4，5 章为强制性规范，其它章节为建议规范。如违背本条，请参考第一条。

第2章 一般规则

1) 所有包，类，接口，方法，属性，变量，参数均使用英文单词进行命名，具体细节请参见命名规范一章。
2) 命名包，类，接口，方法以及变量时，尽量使用贴近问题域的表意丰富的名称。
3) 修改源代码时，应尽量保持与所修改系统的编码风格保持一致。
4) 所有包名使用必须使用 com.[company]前缀，所有项目使用 com.[company].projects.[project name], company 是公司简称，project name 是项目的缩写。

第3章 格式规范

5) 包的导入应该按照相关性进行分组。

```
import java.io.IOException;
import java.net.URL;
```

```
import java.rmi.RmiServer;
import java.rmi.server.Server;

import javax.swing.JPanel;
import javax.swing.event.ActionEvent;

import org.linux.apache.server.SoapServer;
```

6) 只倒入明确需要的类，这样只要看导入列表，就可以知道该类依赖于哪些类和接口，保证可读性。

```
import java.util.List; // 避免: import java.util.*
import java.util.Arraylist;
import java.util.HashSet;
```

7) 类和接口中元素的布局顺序。

1. 类和接口的文档描述
2. 类和接口的声明
3. 类的静态变量，按照 public，protected，package，private 的顺序。
4. 实例变量，按照 public，protected，package，private 的顺序。
5. 类的方法，无固定顺序。

8) 类的声明，基类和实现的接口应该独立成行，保证可读性。

```
class UserManagerImpl
   extends AbstractManager
      implements IUserManager{
   …
}
```

9) 方法修饰关键字定义顺序。

<public, protected, private > static abstract synchronized

unuaual final native methodName

注意访问标示符一定要在最前面。

```
public static double square(double a);
```

```
//避免: static public double square(double a);
```

10) 变量声明，采用 Camel 表示法不要在一行声明多个变量。

```
//推荐
int level;
int size;

//避免
int level, size;
```

11) 保证明确的类型转换，不要默认进行隐式类型转换

```
intValue = (int) floadValue; //避免 intValue = floatValue
```

12) 数组指示符紧跟类型变量

```
int[] a = new int[20]; // 避免: int a[] = new int[20]
```

一个变量要代表独立的意思，不要在其生命周期赋予它不同的概念。

```
int tempValue;
tempValue = maxValue;
…
…
tempValue = minValue;
…
tempValue = anotherValue;
```

tempValue 在生命周期内表示了各种各样的意图，增加理解代码的难度。

应该为每个独立概念定义单独的变量：

```
int tempMaxValue;
int tempMinValue;
int tempAnotherValue;
```

13) 仅仅循环控制变量才能出现在for()循环中

```
sum = 0;
for (i = 0; i < 100; i++) {
    sum += value[i];
}

//避免:
for (i = 0, sum = 0; i < 100; i++){
    sum += value[i];
}
```

14) 循环变量应靠近循环体初始化

```
isDone = false
while(!isDone){
    …
}

//避免
isDone = false;
…
…
while(!isDone){
    …
}
```

15) 避免长的布尔表达式，应换成多个更容易理解的表达式。

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
    …
}
// 避免
if ((elementNo < 0) || (elementNo > maxElement)|| elementNo ==
lastElement) {
    …
 }
```

16) 不要在条件语句中执行方法，以提高可读性

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
    …
}

//避免
if (File.open(fileName, "w") != null)) {
    …
}
```

17) 代码缩进，应该使用 4 个空格为一个单位进行缩进。

```
public String invoke() throws Exception {
....String profileKey = "invoke: ";
    try {
    ....UtilTimerStack.push(profileKey);
        if (executed) {
        ....test = true;
    }
```

```
        catch{
        }
    }
```

18) 条件语句的主要形式，即使单条语句，也要使用括号括起来。

```
    if (condition) {
        statements;
    }

    if (condition) {
        statements;
    } else {
        statements;
    }

    if (condition) {
        statements;
    } else if (condition) {
        statements;
    } else {
        statements;
    }
```

19) 空循环体也要使用完整的{}块

```
    for (initialization; condition; update) {
        ;
    }
```

20) switch 语句的使用格式

```
switch (condition) {
   case ABC :
      statements;
      //穿透，一定要做出注释
   case DEF :
      statements;
      break;
   case XYZ :
      statements;
      break;
   default :
      statements;
      break;
}
```

21) try-catch 使用格式

```
try {
   statements;
}
catch (Exception exception) {
   statements;
}

try {
   statements;
}
catch (Exception exception) {
   statements;
```

```
}
finally {
   statements;
}
```

22) 空格的使用

1. 运算符两边应该各有一个空格。
2. Java 保留字后面应跟随一个空格。
3. 逗号后面应跟随一个空格。
4. 冒号两个应各有一个空格。
5. 分号后面应跟随一个空格。

```
a  = (b + c) * d;          // NOT: a=(b+c)*d
while (true) {             // NOT: while(true){ ...
doSomething(a, b, c, d);   // NOT: doSomething(a,b,c,d);
case 100 :                 // NOT: case 100:
for (i = 0; i < 10; i++) { // NOT: for(i=0;i<10;i++){…
```

23) 空行的使用

1. 文件头部注释、package 语句和 import 语句之间。
2. class 之间
3. 方法之间
4. 方法中，变量的申明和具体代码之间。
5. 逻辑上相关的语句段之间
6. 块注释和行注释的前面

▽ --- 代表空行

```
/**
```

```
 *
 */
▽
package XXX.XXX;
▽
import  XXX.XXX.XXX.XXX;
▽
/**
 *
 */
public class UserFileAccess {
▽
   //
   private int myObjId;
▽
   /**
    *
    */
   public UserFileAccess() {
      ...
   }
▽
   /**
    *
    */
   public void getCtlInfo() {
      int     count;
      String  msg;
▽
      count = 100;
      ...
▽

      //实现代码注释前空行
      msg = “MESSAGE”;
▽
      count = dataCount;
      if (count == 0) {
         ...
      }
   }
▽
/**
 *
```

```
 */
private class UserFileRead {
```

24)　逻辑上紧密相关的代码块应该用一个空行分开。

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

25)　为了保证可读性，变量名应该左对齐。

```
TextFile  file;
int       nPoints;
double    x, y;

//避免
TextFile  file;
int nPoints;
```

```
double x, y;
```

26) 像前面一般规则里说的那样，任何提高代码可读性的排版都可以去尝试，下面是一些例子。

```
if      (a == lowValue)     compueSomething();
else if (a == mediumValue)  computeSomethingElse();
else if (a == highValue)    computeSomethingElseYet();

value = (potential         * oilDensity)   / constant1 +
        (depth             * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition =       computeDistance(min,     x, y, z);
averagePosition =   computeDistance(average, x, y, z);

switch (phase) {
   case PHASE_OIL   :  text  = "Oil";    break;
   case PHASE_WATER :  text  = "Water";  break;
   case PHASE_GAS   :  text  = "Gas";    break;
}
```

27) 当对 if 语句中的条件进行折行时，应该使折行的条件语句相对主功能语句再行缩进 4 个空格，以突出主要功能语句。

```
//使用这种缩进，突出主要功能语句。
if ((condition1 && condition2)
       || (condition3 && condition4)
```

```
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//避免使用这种缩进，主功能语句不突出。
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

28) 三元条件运算符

可以使用如下三种表达方式，条件要用括号括起来。

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha =  (aLongBooleanExpression)
         ? beta
         : gamma
```

第4章 命名规范

4.1 一般命名规范

29) 包名应该用小写字母，不要出现下划线等符号，名词用有意义的缩写或者英文单词。

示例：

```
//推荐
com.esse.business
java.lang.util
```

```
//避免
com.Esse-tech.buSiness
```

30) 所有类命名使用 Pascal 表示方式，使用名词组合。

```
UserManager, ClassLoader, HttpHeaderResult
```

31) 接口命名使用字母“I”加上 Pascal 形式的表示方式。

```
IQuery, IDataAccess,IReportBuilder
```

32) 使用名词组合或形容词去命名一个接口，接口声明了一个对象能提供的服务，也描述了一个对象的能力。一般以“able”和“ible”作为后缀，代表了一种能力。

```
public interface Runnable{
   public void run();
}

public interface Accessible{
   public Context getContext();
}
```

33) 变量名和参数名使用 Camel 表示方式。

```
userName, objectFactory, entrys, list
```

34) 对于常量名，使用大写字母，并使用下划线做间隔。

```
MAX_TIMES, DEFAULT_NAME
```

程序中应该使用常量代替“25”，“100”等实际的数字，如：

```
//推荐
```

```
if(times == MAX_TIMES){
    …
}

//避免
if(times == 25){
    …
}
```

这样做的好处是，当因需要修改实际的数字时，比如修改 25 为 30，只需要修改一处。

35) 方法名应该使用动词开头，使用 Camel 表示方式,一般由动词+名词组成。

```
getName, initialize, addParameter, deleteUser
```

36) 缩写字母也应该保持首字母大写

```
exportHtmlSource(); // 避免: exportHTMLSource();
openDvdPlayer();    // 避免: openDVDPlayer();
```

37) 变量的名字应该和类型名称一致

```
void setTopic(Topic topic) // 避免: void setTopic(Topic value)
                           // 避免: void setTopic(Topic aTopic)
                           // 避免: void setTopic(Topic t)
void connect(Database database)
                         // 避免: void connect(Database db)
                         // 避免: void connect(Database oracleDB)
```

当同时定义多个属于同一个类的变量时，把类型作为实例的后缀，如：

```
Point startPoint;
Point centerPoint;
```

这样做是为了从实例名就可以推断它的类型名称。

38) 根据变量的作用范围，作用范围大的应该使用长名称，作用范围大，表明变量的生命周期比较长，为了有助于理解，应尽量用长名称以表达变量的真实意图。反之，对于作用范围小，可以使用一些简化的名称，比如 i，j，k 等，提高编程效率。

```
for(int i =0;i < times; i++){
   …
}
```

4.2 特殊命名规范

39) 使用 get/set 对类属性进行访问，这是 Java 社区的核心编码规范。

40) 使用 *is* 前缀表示一个布尔变量和方法。

```
isUsed, isEmpty,isVisible,isFinished
```

有时也可以使用 has，can，should：

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

41) 在查询方法中应使用 *find* 作为前缀

```
vertex.findNearestVertex();
matrix.findSmallestElement();
```

```
node.findShortestPath(Node destinationNode);
```

42) 使用 initialize 做为对象初始化的方法前缀，也可以简写为 init

```
initializeFiles();
init();
initFontSet();
```

43) 对于对象集合， 变量名称应使用复数。

```
Collection<Point>  points;
int[]              values;
```

44) 对于抽象类，应该使用 Abstract 前缀。

```
AbstractReportBuilder，AbstractBeanFactory
```

45) 对于表示编号的变量，应加 No 后缀。

```
tableNo, userNo,employeeNo
```

46) 常在一起使用的对称词汇，这些词汇一起使用，方法的表达意图自然可以互相推测和演绎。

```
get/set,  add/remove,  create/destroy,  start/stop,
insert/delete, increment/decrement, begin/end, first/last,
up/down,  min/max,  next/previous,  old/new,  open/close,
show/hide, suspend/resume
```

47) 避免使用否定布尔变量

```
bool isError;  // 避免: isNoError
bool isFound;  // 避免: isNotFound
```

48) 异常类应该使用 Exception 做为后缀。

AccessException, RuntimeException

49) 缺省接口实现应该使用 Default 前缀

```
class DefaultTableCellRenderer
    implements TableCellRenderer {
    …
}
```

50) 对于单例类（Singleton），应该使用 getInstance 方法得到单例。

```
class UnitManager {
    private final static UnitManager instance = new UnitManager();
    private UnitManager() {
        ...
    }
    public static UnitManager getInstance(){
        return instance_;
    }
}
```

51) 对于工厂类，进行创建对象的方法，应该使用 new 前缀

```
class PointFactory {
    public Point newPoint(...) {
        ...
    }
```

```
}
```

第5章 注释规范

5.1 概述

代码中为什么要包含注释？

1. 别人要调用你的程序中的公共接口，对这部分进行文档描述，使别人能够正确而有效的使用它。
2. 除了自己，别人要阅读和维护你的代码。为了使代码更容易维护，首先要使代码更易于理解，才能在理解的基础上进行维护。对这些代码进行文档描述，将使这个过程变得更加容易。

对代码进行注释，是在代码可读性的基础上，使用自然语言对代码所表达的意思进行阐述。并不是说代码可以写的很烂，注释写的很详细，这不是好的方式。如果代码可读性很好，命名表意丰富，清晰，一般不需要特别多的注释。对于类，主要着重要描述它的职责，即它能干什么，对于复杂的算法实现，应该使用内部实现注释，说明算法的主要思路，对于长方法，要让阅读代码的人比较容易的明白方法实现的主要流程。反之，对于一看就懂的方法，则不需要进行注释，比如 `get/set` 方法。

5.2 一般原则

1. 代码应该和注释保持同步，如果代码和注释不同步，则阅读代码的人会想，“到底是代码准确，还是注释准确啊”，换谁都会糊涂。
2. 注释尽量简洁，尺度没有准确的定义，大部分人能明白即可，可以将自己的代码给同事看看。太简单的方法就不要注释了，比如上面提到的 `get/set` 方法。

5.3 注释内容

```
/*
 * Copyright (c) 2002-2006 by OpenSymphony
 * All rights reserved.
 */
package com.opensymphony.xwork2;

import com.opensymphony.xwork2.interceptor.PreResultListener;
import com.opensymphony.xwork2.util.ValueStack;

import java.io.Serializable;


/**
 *  类职责简要描述
 *  。。。
 *
 * @author Jason Carreira
 * @see com.opensymphony.xwork2.ActionProxy
 */
public class ActionInvocation implements Serializable {

    /**
     * 方法简要描述
     *
     * 方法详细描述
     * 。。。
     *
     * JavaDoc tags,比如
     * @author
     * @version
     * @see 。。。
     * @return a Result instance
     */
    Result getResult() throws Exception{
        //内部实现注释
        //多行内部实现注释
        String name = this.getName();
    }

    /**
     * Get the Action associated with this ActionInvocation
     */
```

```
    Object getAction();

    /**
     * @return whether this ActionInvocation has executed before.
     *         executed.
     */
    boolean isExecuted();

    /**
     * Invokes the next step in processing this ActionInvocation.
     * one. If Interceptors choose not to
     * they will call invoke() to allow the next Interceptor to execute
     * the Action is executed. If the ActionProxy getExecuteResult
     */
    String invoke() throws Exception;

}
```

① 代码的版权信息。

② 类描述信息，描述类的主要职责和用处。

③ 方法描述信息，描述方法是做什么的，如何调用，最好给出调用代码示例。

④ JavaDoc tags ,用来生成 Html 形式的 API 文档

⑤ 内部实现注释，用于描述复杂的算法，长方法，从为什么要这么做角度去描述

52) 尽可能在类描述中加入代码调用示例，使用<pre></pre>标记，提示 JavaDoc 工具不要改变格式.

```
/**
 * DateFormat is an abstract class for date/time formatting
 * formats and parses dates or time in a language-independent manner.
 *
 * <pre>
```

```
 *  myString = DateFormat.getDateInstance().format(myDate);
 * </pre>
 * <pre>
 *  DateFormat df = DateFormat.getDateInstance();
 *  for (int i = 0; i < myDate.length; ++i) {
 *    output.println(df.format(myDate[i]) + "; ");
 *  }
 * </pre>
 *
 * @see         Format
 * @see         java.util.TimeZone
 * @version     1.51 04/12/04
 * @author      Mark Davis, Chen-Lieh Huang, Alan Liu
 */
public abstract class DateFormat extends Format {
```

包含代码调用示例

53)　使用**@deprecated**废弃方法，不要删掉它。

```
   /**
    * ...
    * @deprecated
    */
   /*
     Get a default date/time formatter that uses the SHORT
     date and the time.

       public final static DateFormat getInstance() {
           return getDateTimeInstance(SHORT, SHORT);
       }
   */
```

54) 使用行末注释对深层嵌套代码进行注释

```
for(i…){
    for(j…){
        while(…){
            if(…){
                switch(…){
                    …
                }// end switch
            }//end if
        }//end while
    }//end for i
}//end for j
```

第6章 编程实践

55) 对于静态方法，应该使用类名去使用，不应该用实例去引用，主要是为了体现更多的语义。

```
Thread.sleep(1000);

//避免,无法体现 sleep 是静态方法还是实例方法
thread.sleep(1000);
```

56) 对一些基本数据类型和不太可能通过继承进行扩展的类，应声明为 final，提高效率。

57) 类和方法的粒度保持适中，保持类的规模尽量短小，职责单一。小类有很多好处，易于设计，易于测试，易于理解。同样方法也要尽量的小，每个方法尽量不要超出 25 行。

58) 开闭原则，软件应该对扩展开发，对修改开放。也就是说，应该在不修

改以前源代码的基础上，改变程序的行为以适应新的需求。

59) 里氏代换原则：假设有两个类，一个是基类 Base，一个是派生类 Derived，如果一个方法可以接受基类对象 b 的话：method1(Base b)，同样，这个方法也应该接受派生类 Derived 的对象 d，而不影响方法的行为。里氏代换原则是继承复用的基石。

60) 抽象依赖原则（稳定依赖原则）。应该依赖于抽象而不依赖与具体类，抽象的类和接口是稳定的，而具体类是易变的，如果依赖于具体类，代码就会非常脆弱，失去了灵活性。

61) 接口隔离原则，一个类对另外一个类的依赖应该建立在最小的接口之上的。

62) 单一职责原则，如果一个类有多于一种的职责，当需求变化时，类的职责就要发生变化，而因此就会引起引用该类的代码发生改变，职责越多，这个类就容易跟更多的类产生耦合关系，而且改变一个职责，可能会影响到另外一个职责的履行。

63) 编写代码前，先编写注释（可以认为是伪代码），先想后写。

```
/**
 * 报表构建器，主要职责：
 * 1.创建拷贝报表模版
 * 2.填充报表数据
 * 3.构建报表
 * 4.画图处理
 */
```

通过编写这些伪代码，可以起到理清思路的作用，这时候再编写代码，过程就非常流畅了，不会编一会儿，想一会儿，删掉代码，再重新编。

第7章 设计模式快速参考

本章描述常见的设计模式。描述主要由两部分组成，一部分是模式代码，一部分时调用示例。

7.1 工厂模式

7.1.1 简单工厂

```
abstract class Fruit{
}

class Apple extends Fruit{
}

class Orange extends Fruit{

}

class FruitFactory{

   pulic static Fruit getFruit(String fruitType){
      if (“apple” == fruitType){
         return new Apple();
      } else if (“orange” == fruitType) {
         return new Orange();
      }
   }
}
```

Client：

```
Apple apple = FruitFactory.getFruit(“apple”);
...
```

7.1.2 工厂方法

```
interface IFruitFactory{
   puhlic Fruit getFruit();
}
```

```
class AppleFactory implements IFruitFactory{
   public Fruit getFruit(){
      //生产苹果
      return new Apple();
   }
}
```

Client：

```
IFruitFactory factory = new AppleFactory();
Fruit fruit = new factory.getFruit();
```

7.2 单例模式

```
class Singleton{
   private static Singleton singleton = null;
   public static Singleton getInstance(){
      if(null == singleton){
         singleton = new Singleton();
      }

      return singleton;
   }

   public String otherOperation(){
      //方法实现
   }
}
```

Client:

```
String str = Singleton.getInstance().otherOperation();
```

多线程时使用 double-check 模式确保线程安全：

```
class Singleton{
   private static Singleton singleton = null;
   public static Singleton getInstance(){
      if(null == singleton)
         sychronized (Singleton.class){
```

```
            if(null == singleton){
                singleton = new Singleton();
            }
        }
    }

    return singleton;
  }
}
```

7.3 适配器模式

```
interface Powerable{
   110v provide();
}

class 110v {
}

class 220v {
}


class 110vPower implements Powerable{
   public 110v provide(){

      //提供 110v 电压

   }
}

class 220vPower {
   public 220v provide(){

      //提供 220v 电压

   }
}


class 220vAdapter implements Powerable{
   public 110v provide(){
      110v voltage = null;
      220vPower power = new 220vPower();
```

```
        //转换过程，@!@#$%^
        return voltage;
    }
}

Client:

Powerable provider = new 220vAdapter();
provider.provide();
```

7.4 组合模式

```
abstract class Hardware{
}
class Mainboard extends Hardware{
}
class Memory extends Hardware{
}
class Display extends Hardware(
}
class NetworkAdapter extends Hardware{
}

class Computer extends Hardware[
    private List parts = new Arraylist();
    public List add(Hardware hardware){
        parts.add(hardware);
        return parts;
    }
}

Client:
Computer computer = new Computer();
Mainboard mainboard = new Mainboard();
NetworkAdapter networkAdapter = new NetworkAdapter();
Display display = new Display();
Memory memory = new Memory();
computer .add(mainboad)
         .add(networkAdapter)
         .add(display)
         .add(memory);
```

7.5 外观模式

为一组类提供简单的外部接口，使外部调用者不需要和所有内部干系人打交道，就能让调用者满意。

```
class CallCenter{
   public void solve(Customer customer){
      //接受客户提出的问题
      operator.acceptProblem(customer.getProblem());
      boolean  canSolved = operator.solve();
      if (!canSolved) {
         //如果不能解决，则请求其它人帮助。
         operator.askHelp();
      }
   }
}

class Customer{
   public void call(CallCenter callCenter){
      callCenter.solve(this);
   }
}
class Operator{
}
```

Client:

```
CallCenter callCenter = new CallCenter();
Custom aCustomer = new Customer();
aCustomer.call(callCenter);
```

这里对客户来讲，与他接触的只有一个接口，就是接线员，最后的结果是解决他的问题。接线员可以直接解决，如果他不能解决，它可以选择请求其它人的帮助去解决这个问题。客户是不关心接线员在内部做了什么。

7.6 代理模式

```
class FileDownloader(
   public download(File file);
}
class File{
}

//一般代理类使用 Proxy 后缀
class FileDownloaderProxy{
   FileDownloaderProxy(FileDownloader downloader){
      this.downloader = downloader;
   }
   private FileDownloader downloader;

   public download(File file){
      //这里可以添加通知，通知用户开始下载文件
      notifyDownloadWillStart();

      //调用代理目标类的方法，进行下载文件
      this.downloader.download(file);

      //这里可以添加通知，通知用户文件下载完成
      notifyDownloadIsComplete();

   }
   private void notifyDownloadWillStart(){
      System.out.println(“下载开始…”);
   }

   private void notifyDownloadIsComplete(){
      System.out.println(“下载完成!”);
   }

}
```

```
Client:

FileDownloaderProxy proxy =
      new FileDownloaderProxy( new FileDownloader());
proxy.downlaod(file);
```

代理模式提供了一种间接性，可以做一些附加工作，比如记录日志，触发一些事件等，Spring 框架中大量使用了这个模式来进行 AOP 编程。

7.7 命令模式

```
interface ICommand{
   void execute(IReceiver receiver);
}

//发送传真命令
class SendFaxCommand implenments ICommand{
   void execute(IReceiver receiver){
      receiver.do(this);
   }

}

class AttackCommand implements ICommand{
   void execute(IReceiver receiver){
      receiver.do(this);
   }

}

class General{
   public ICommand createCommand(String commandType){
      if(“发传真” == commandType){
         return new SendFaxCommand();
      }else if (“打仗” == commandType){
         return new AttackCommand();
      }
   }
```

```
}

interface IReceiver{
   void do(ICommand command);
}

class Secretary implements IReceiver{
   //将具体命令信息传入，如传真文件内容等。
   public void do(ICommand command){
      //发送传真
   }
}

class Soldier implements IReceiver {
   //将具体命令传入，如作战地点，作战目标等等。
   public void do(ICommand command){
      //打仗
   }
}

Client:
General pengDeHuai = new General();
Secretary pengDeHuaiSecretary = new Secretary();
Soldier zhangSan = new Soldier(“张三”);
ICommand sendFaxCommand =
                  pendDeHuai.creatCommand(“发传真”);
sendFaxCommand.execute(pengDehuaiSecretary);

ICommand sendFaxCommand =
                  pendDeHuai.creatCommand(“打仗”);
sendFaxCommand.execute(zhangSan);
```

7.8 观察者模式

```
//主题,这里是快餐店
class SnackShop{
```

```
    private List<Customer> customers = new Arraylist();
    public void add(Customer customer){
        customers.add(customer);
    }

    //通知，对订阅主题的客户发布通知，比如“外卖已好”
    public void notify(){

        for(Customer c: customers){
            c.getFood();
        }
    }

}

//订阅者，这里是客户
class Customer{
    Customer (SnackShop shop){
        //将客户加入到快餐店列表
        shop.add(this);
    }

    //回调函数，当接到通知后，客户的动作
    public void getFood(){
        //取得外卖
    }

}
```

Client：

```
SnackShop snackShop = new SnackShop();
Customer zhangSan = new Customer(snackShop);
snackShop.notify();
```

7.9 策略模式

```
//密钥对生成接口
interface IKeyPairGenerable{
   KeyPair create();
}
class KeyPair{
}
class DesKeyPairGenerator  implements IKeyPairGenerable{
}

class IdeaKeyPairGenerator implements IKeyPairGenerable{
}

class IdeaKeyPairGenerator implements IKeyPairGenerable{
}



class KeyPairManager{
   private IKeyPairGenerable generator;
   private List keyPairList = new Arraylist();

   public void setGenerator(IKeyPairGenerable generator){
      this.generator = generator;
   }

   public KeyPair create(){
      KeyPair keyPair = null;
      if(null != generator){
         keyPair = generator.create();
         keyPairList.add(keyPair);
      }

      return keyPair;
   }
}
```

Client:

```
IKeyPairGenerable desGenerator =
```

```
                        new DesKeyPairGenerator();

IKeyPairGenerable rsaGenerator =
                        new RsaKeyPairGenerator();

IKeyPairGenerable ideaGenerator =
                        new IdeaKeyPairGenerator();

KeyPairManager manager = new KeyPairManager();


//使用 DES 算法生成密钥
manager.setGenerator(desGenerator);
KeyPair keyPair = manager.create();


//使用 RSA 算法生成密钥
manager.setGenerator(rsaGenerator);
KeyPair keyPair = manager.create();
```

可以看出，KeyPairManager 仅仅依赖于接口 IKeyPairGeneratble，改变密钥对生成算法不改变 KeyPairManager 类。策略模式一般用来封装算法的不同实现。

7.10 模版方法模式

```
abstract class TravelTemplate{
   public void travel(){


      //上车
      getOnBus();


      //去目的地

      goto(“目的地”)

      //吃饭
      eat();
      //下车
```

```
        getOffBus();
        //如果想拍照，就拍照
        if(wantToTakePhoto){
            takePhoto();
        }

        //如果想上厕所，就上厕所
        if(wantToGotoRestroom){
            gotoRestRoom();
        }


        //上车回家
        getOnBus();
        goHome();

    }

    protected void goto(String str){
    }

    protected void getOnBus(){
    }

    protected void eat{
    }

    protected void takePhoto(){
    }

    protected void gotoRestRoom(){
    }

    protected void getOffBus(){
    }

    protected void goHome(){
    }

}
```

```
class HangzhouTravelTemplate extends TravelTemplate{
}

class BeijingTravelTemplate extends TravelTemplate{
}

class Tourist{
   private TravelTemplate template;
   pulic void setTravelTemplate(TravelTemplate template){
   }

   public travel(){
      template.travel();
   }
}

TravelTemplate hangzhouTemplate =

              new HangzhouTravelTemplate(“杭州一日游”)

TravelTemplate beijingTemplate =

              new HangzhouTravelTemplate(“北京奥运两日游”)


Tourist zhangSan = new Tourist(“张三”);

//杭州一日游

zhangSan.setTemplate(travelTemplate);
zhangSan.travel();


//北京两日游

zhangSan.setTemplate(beijingTemplate);
zhangSan.travel();
```

7.11 参观者模式

参观者模式主要对一组固定结构的对象进行访问，一般和组合模式一起使用。

```
abstract class Hardware{
   protected double price;
   protected double getPrice();
   void accept(IComputerVisitor visitor);
}

class Mainboard extends Hardware{
   void accept(IComputerVisitor visitor){
      visitor.visitMainboard(this);
   }
}

class Memory extends Hardware{
   void accept(IComputerVisitor visitor){
      visitor.visitMemory(this);
   }

}

class Display extends Hardware(
   void accept(IComputerVisitor visitor){
      visitor.visitDisplay(this);
   }

}

class NetworkAdapter extends Hardware{
   void accept(IComputerVisitor visitor){
      visitor.visitNetworkAdapter(this);
   }

}

class Computer extends Hardware[
   private List<Hardware> parts = new Arraylist();
   public List<Hardware> add(Hardware hardware){
      parts.add(hardware);
      return parts;
   }

   public void accept(IComputerVisitor visitor){
      for(Hardware h: parts){
         h.accept(visitor);
```

```
        }
    }
}


interface IComputerVisitor{
    void visitMainboard(Mainboard mainboard);
    void visitNetworkAdapter(NetworkAdapter adapter);
    void visitDisplay(Display display);
    void visitMemory(Memory memory);
}


//遍历 computer 的每个部件，汇总价格
class PriceVisitor implements IComputerVisitor{
    //总价格
    private double amountPrice;

    void visitMainboard(Mainboard mainboard){
        amountPrice += mainboard.getPrice();
    }
    void visitNetworkAdapter(NetworkAdapter adapter) {
        amountPrice += adapter.getPrice();
    }

    void visitDisplay(Display display) {
        amountPrice += display.getPrice();
    }

    void visitMemory(Memory memory) {
        amountPrice += memory.getPrice();
    }


}
```

Client:

```
Computer computer = new Computer();
Mainboard mainboard = new Mainboard();
NetworkAdapter networkAdapter = new NetworkAdapter();
Display display = new Display();
Memory memory = new Memory();
```

```
computer .add(mainboad)
         .add(networkAdapter)
         .add(display)
         .add(memory);
IComputerVisitor visitor = new PriceVisitor();
visitor.accept(computer);
```

第8章 单元测试

8.1 为什么要单元测试

8.2 单元测试工具

第9章 代码重构

正在编写中。。。

9.1 长方法

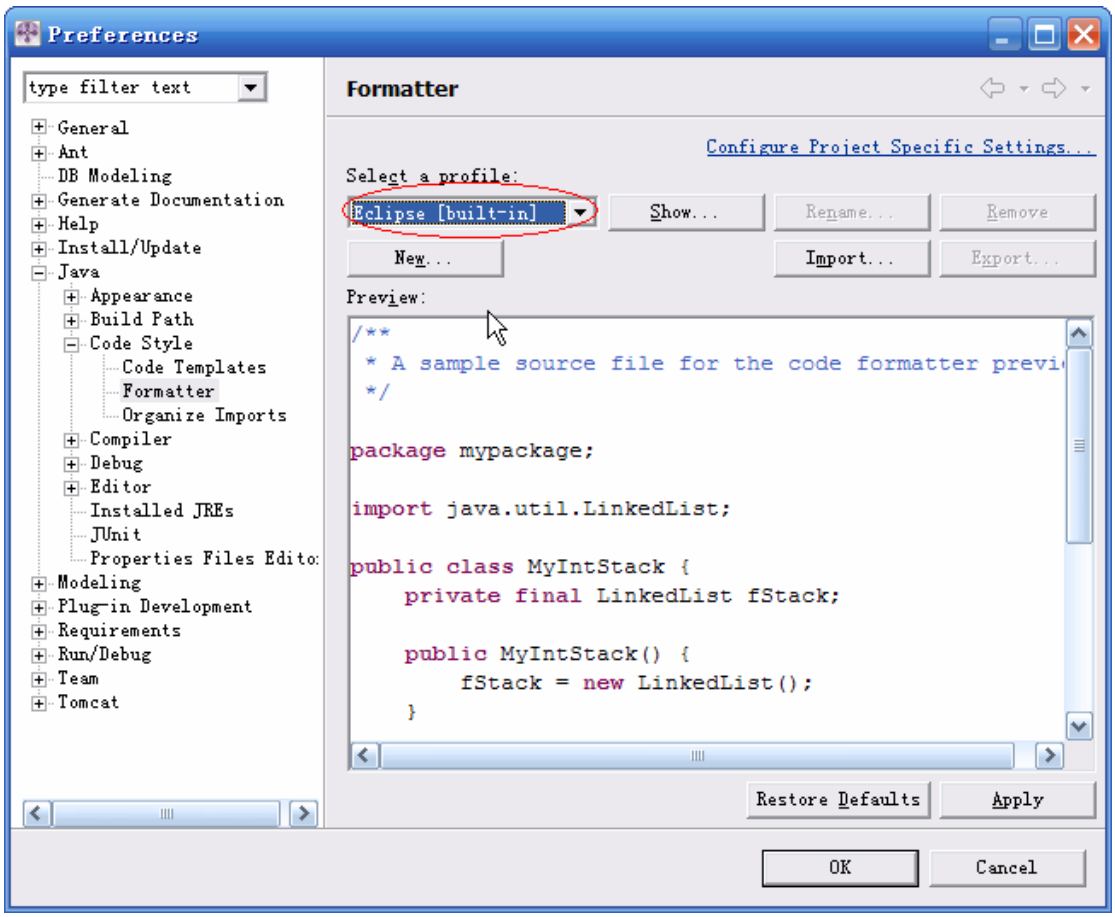
9.2 巨大类

第10章 工具支持

本规范中所描述的格式规范，均可通过 eclipse 自带的 Format 功能实现。

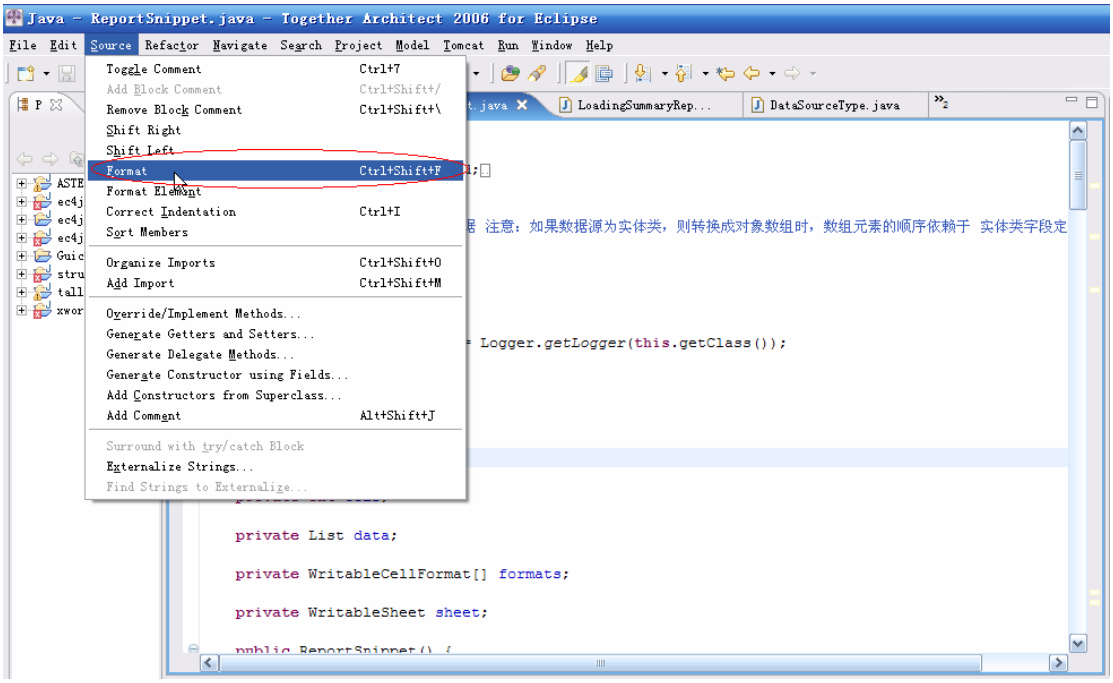
10.1 eclipse 配置

如下图，配置使用默认配置，不需要改动。



10.2 使用方式

使用起来非常简单，菜单 `Source -> Format` 或者直接按 `Ctrl+Shift+F`


Java - ReportSnippet.java - Together Architect 2006 for Eclipse
File Edit Source Refactor Navigate Search Project Model Tomcat Run Window Help
Toggle Comment Ctrl+7
Add Block Comment Ctrl+Shift+/
Remove Block Comment Ctrl+Shift+\
Shift Right
Shift Left
Format Ctrl+Shift+F
Format Element
Correct Indentation Ctrl+I
Sort Members
Organize Imports Ctrl+Shift+O
Add Import Ctrl+Shift+M
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate Constructor using Fields...
Add Constructors from Superclass...
Add Comment Alt+Shift+J
Surround with try/catch Block
Externalize Strings...
Find Strings to Externalize...
LoadingSummaryRep...
DataSourceType.java
注意：如果数据源为实体类，则转换成对象数组时，数组元素的顺序依赖于 实体类字段定
Logger.getLogger(this.getClass());
private List data;
private WritableCellFormat[] formats;
private WritableSheet sheet;

第11章　参考资料

1. http://geosoft.no/development/javastyle.html
2. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
3. http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
4. http://www.refactoring.com/
5. 设计模式-可复用面向对象软件的基础 机械工业出版社
6. Java 与模式　闫宏著
7. Java 编码规范 人民邮电出版社
8. 重构-改善既有代码的设计 Martin Fowler 2003 电力出版社

第12章　致谢