infohold-logo.png

智控-平台-规范

JAVA编码规范

密 级：□ 普通 ■ 秘密 □ 机密 □ 绝密

文档编号：IH\_CP\_SPC\_001

| 版本号 | 版本说明/变更理由/变更内容 | 作者/日期 | 审批人/日期 | 备注 |
| --- | --- | --- | --- | --- |
| 0.5 | 初始创建/C | 杨明/20110607 |  |  |
| 0.6 | 移除框架要求目录/D | 张玉双/20160508 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

变更说明：C：Create，初始创建；A：Add，增加内容；M：Mod，修改；D：Del，删除

目录

[1. 概述 5](#_Toc427932352)

[1.1. 简介 5](#_Toc427932353)

[1.2. 目的 5](#_Toc427932354)

[1.3. 适用范围 5](#_Toc427932355)

[1.4. 文档读者 5](#_Toc427932356)

[1.5. 词汇表 5](#_Toc427932357)

[1.6. 参考资料 5](#_Toc427932358)

[2. 一般规则 6](#_Toc427932359)

[3. 格式规范 8](#_Toc427932360)

[4. 命名规范 11](#_Toc427932361)

[4.1. 一般命名规范 11](#_Toc427932362)

[4.2. 特殊命名规范 12](#_Toc427932363)

[5. 注释规范 13](#_Toc427932364)

[5.1. 概述 13](#_Toc427932365)

[5.2. 一般原则 14](#_Toc427932366)

[5.3. 注释内容 14](#_Toc427932367)

[6. 框架要求 17](#_Toc427932368)

[6.1. Icore(SpringBoot) 17](#_Toc427932369)

[6.1.1. 目录格式 17](#_Toc427932370)

[6.1.2. 文件命名 18](#_Toc427932371)

[6.1.3. Model定义 19](#_Toc427932372)

[6.1.4. 方法定义 19](#_Toc427932373)

[6.1.5. 配置文件 20](#_Toc427932374)

[7. 编程实践 20](#_Toc427932375)

[8. 设计模式快速参考 21](#_Toc427932376)

[8.1. 工厂模式 21](#_Toc427932377)

[8.1.2 工厂方法 21](#_Toc427932378)

[7.1.1 简单工厂 21](#_Toc427932379)

[8.2. 单例模式 22](#_Toc427932380)

[8.3. 适配器模式 23](#_Toc427932381)

[8.4. 组合模式 24](#_Toc427932382)

[8.5. 外观模式 24](#_Toc427932383)

[8.6. 代理模式 25](#_Toc427932384)

[8.7. 命令模式 26](#_Toc427932385)

[8.8. 观察者模式 27](#_Toc427932386)

[8.9. 策略模式 28](#_Toc427932387)

[8.10. 模版方法模式 29](#_Toc427932388)

[8.11. 参观者模式 31](#_Toc427932389)

[9. 单元测试 33](#_Toc427932390)

[9.1. 必要性 33](#_Toc427932391)

[9.2. 单元测试工具 33](#_Toc427932392)

[10. 代码重构 33](#_Toc427932393)

[10.1. 长方法 33](#_Toc427932394)

[10.2. 巨大类 33](#_Toc427932395)

[11. 工具支持 33](#_Toc427932396)

[11.1. eclipse 配置 33](#_Toc427932397)

[11.2. 使用方式 33](#_Toc427932398)

# 概述

## 简介

## 目的

本文档指导前端平台初级开发人员开展产品开发活动，围绕如何去产生规范的代码，使代码更简单，更易懂，更易理解以及算法实现效率更高。

## 适用范围

本规范的适用范围包括产品研发前端平台的全部工作产品以及相关评审记录。

本规范只在本部门范围内有效，对其他部门没有参考作用。

## 文档读者

本规范的预期读者为智控美信相关领导和参与本部门的所有管理、设计、开发人员，未经智控美信书面允许，请勿扩散。

## 词汇表

* 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.

## 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

# 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, company 是公司简称(infohold)，project name 是项目的缩写(ibpp)。
5. 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。

1. 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){

・・・

}

}

1. 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);

1. 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**{

}

}

# 格式规范

1. 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;

1. 只导入明确需要的类，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;**

1. 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。

1. 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);*

1. 变量声明，采用 Camel 表示法尽量不要在一行声明多个变量。

//推荐

int level;

int size;

//避免

int level, size;

1. 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

1. 数组指示符紧跟类型变量

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

1. 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;

1. 条件语句的主要形式，即使单条语句，也尽量要使用括号括起来。

if (condition){

statements;

}

if (condition){

statements;

} else {

statements;

}

if (condition){

statements;

} elseif(condition){

statements;

} else {

statements;

}

1. 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();

}

1. switch 语句的使用格式

switch (condition){

case ABC :

statements;

case DEF :

statements;

break;

case XYZ :

statements;

break;

default :

statements;

break;

}

1. try-catch使用格式

try{

statements;

}catch(Exception exception){

statements;

}

try{

statements;

}catch(Exception exception){

statements;

}finally{

statements;

}

1. 三元条件运算符
2. 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。

# 命名规范

## 一般命名规范

1. 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.infohold.ibpp

java.lang.util

//避免

com.Infohold.ibpp

1. 所有类命名使用Pascal表示方式，使用名词组合。

UserManager, ClassLoader, HttpHeaderResult

1. 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();

}

1. 变量名和参数名使用Camel表示方式。

userName, objectFactory, entrys, list

1. 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。

1. 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

1. 缩写字母也应该保持首字母大写

exportHtmlSource(); *//* 避免*: exportHTMLSource();*

openDvdPlayer(); *//* 避免*: openDVDPlayer();*

1. 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。

1. 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++){

…

}

## 特殊命名规范

1. 使用get/set对类属性进行访问，这是Java社区的核心编码规范。
2. 使用is前缀表示一个布尔变量和方法。

isUsed, isEmpty,isVisible,isFinished

1. 在查询方法中应使尽量用find作为前缀

vertex.findNearestVertex();

matrix.findSmallestElement();

node.findShortestPath(Node destinationNode);

1. 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();

1. 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

1. 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

1. 避免使用否定布尔变量

bool isError; *//* 避免: isNoError

bool isFound; *//* 避免*: isNotFound*

1. 异常类应该使用Exception做为后缀。

AccessException, RuntimeException

1. 缺省接口实现应该使用Default前缀

class DefaultTableCellRenderer implements TableCellRenderer {

…

}

1. 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;

}

}

1. 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(...){

...

}

}

# 注释规范

## 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。

3、整理设计思路，可以将设计思路进行传递

## 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。

## 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。

④JavaDoc 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 {}

1)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);

}

\*/

2)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

# 编程实践

1)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);

2)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。

3)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。

4)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。

5)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。

6)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。

7)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。

8)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。

9)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。

# 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，一部分是调用示例。

## 工厂模式

### 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();

### 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();

} **elseif**(“orange” == fruitType){

**return** new Orange();

}

}

}

**Client**：

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

...

## 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;

}

}

## 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();

## 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);

## 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。

## 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。

## 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();

}**elseif**(“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);

## 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();

## 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。

## 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();

## 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);

# 代码重构

## 长方法

## 巨大类

# 单元测试

## 必要性

## 单元测试工具

单元测试工具使用JUnit（4+）

# 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。

## eclipse 配置

## 常用快捷操作

使用快捷键Ctrl+Shift+F