

***Capstone Project***

**Report 2**

**Software Project Management Plan**

*Personal Finance Management*

**Author: PFM Team**

**Supervisor: Mr. Nguyen Van Hien**

**Hanoi, 2nd Oct, 2012**

Record of changes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Change Item** | **Description** | **By** | **Version** |
| 02/10/2012 | All | Create the document | GamNT01326 | 1.0 |
|  |  |  |  |  |
|  |  |  |  |  |

Table of Contents

[1. Problem Definition 2](#_Toc340562735)

[1.1. Name of this Capstone Project 2](#_Toc340562736)

[1.2. Problem Abstract 2](#_Toc340562737)

[1.3. Project Overview 2](#_Toc340562738)

[1.3.1. The current System 2](#_Toc340562739)

[1.3.2. The Proposed System 2](#_Toc340562740)

[1.3.3. Boundaries of the System 2](#_Toc340562741)

[1.3.4. Development Environment 2](#_Toc340562742)

[2. Problem Organization 3](#_Toc340562743)

[2.1. Software Process Model 3](#_Toc340562744)

[2.2. Roles and Responsibilities 3](#_Toc340562745)

[2.3. Tools and Techniques 5](#_Toc340562746)

[3. Project Management Plan 5](#_Toc340562747)

[3.1. Task Sheet: Assignments and Timetable 5](#_Toc340562748)

[3.2. Risks Management 9](#_Toc340562749)

[3.3. All Meeting Minutes 9](#_Toc340562750)

[4. Coding Convention 10](#_Toc340562751)

[4.1. Java Language Rules 10](#_Toc340562752)

[4.1.1. Don't Ignore Exceptions 10](#_Toc340562753)

[4.1.2. Fully Qualify Imports 12](#_Toc340562754)

[4.2. Java Style Rules 12](#_Toc340562755)

[4.2.1. Use Javadoc Standard Comments 12](#_Toc340562756)

[4.2.2. Limit Variable Scope 13](#_Toc340562757)

[4.2.3. Follow Field Naming Conventions 14](#_Toc340562758)

[4.2.4. Use Standard Java Annotations 15](#_Toc340562759)

[4.2.5. Treat Acronyms as Words 16](#_Toc340562760)

[4.3. Javatests Style Rules 16](#_Toc340562761)

[4.3.1. Follow Test Method Naming Conventions 16](#_Toc340562762)

[4.3.2. 16](#_Toc340562763)

# Problem Definition

## Name of this Capstone Project

As introduction in Report 1, name of this project is **Personal Finance Management**. Abbreviation **PFM**

## Problem Abstract

Our product helps people manage their personal finance in the way of the most quickly and usefully, without waste of time to remember and don't afraid forget any account.

## Project Overview

### The current System

It will be a new system, developed by ourselves. There is no current system.

### The Proposed System

There are some main functions of our product (when completed):

* Schedule: use to plan your future spending
* Revenue and expenditure management: helps user manage revenue and expenditure every month
* Borrowing and lending management: help user to easily manage borrowing and lending.
* Report: analyze your spending with reports and pie charts; help user has an overview of the expenses in a month, compare with plan or other month.
* Warning: alert you of upcoming bills or when expenditures exceed plan.
* Synchronize: help storage data

### Boundaries of the System

* Final product is an application on Android
* Application for Vietnamese
* The system will have to work Android 2.3 or higher

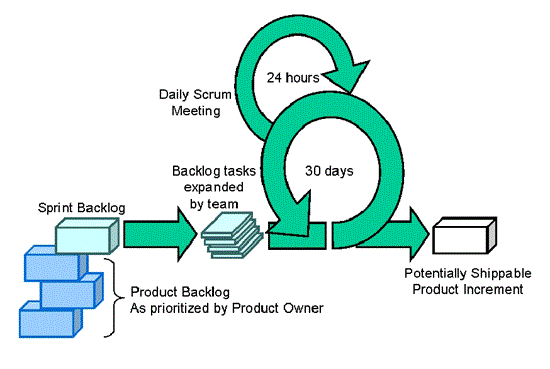
### Development Environment

* OS:
* Developing language
* IDE:
* Database:

# Problem Organization

## Software Process Model

We apply new process model named: Scrum. All phases of project will follow this model.



## Roles and Responsibilities

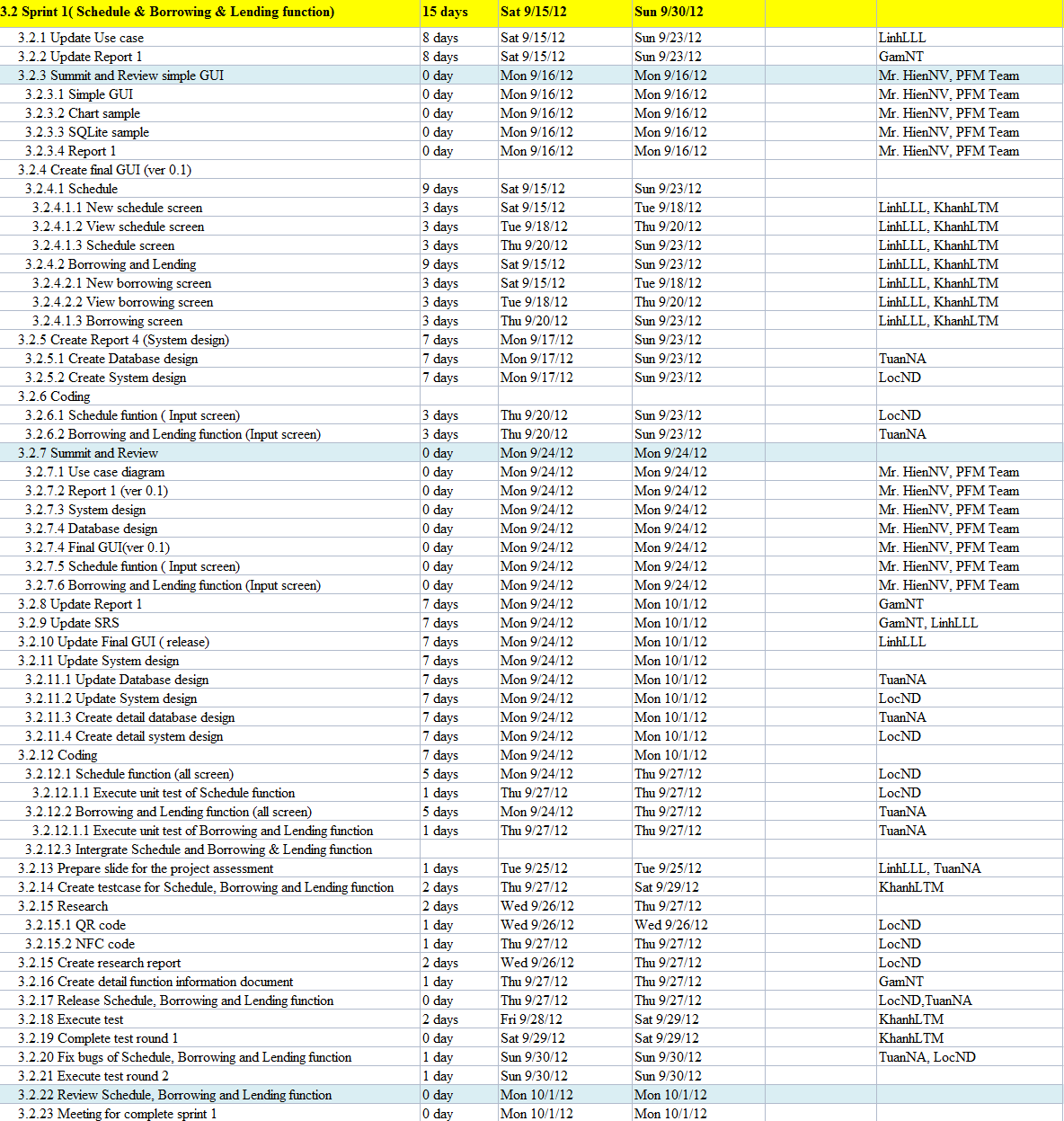
|  |  |  |  |
| --- | --- | --- | --- |
| STT | Full Name | Role in Group | Responsibilities |
| 1 | Lai Le Le Linh | Team Leader  Designer  Tester | - Create project plan, take meeting, check status, and monitor activities...of team members.  - Promote the project implementation and evaluate steps of project.  - Write meeting minutes  - GUI design  - Testing  - Create use-case diagram |
| 2 | Nguyen Duc Loc | Technical Leader  Developer | - Support other members about technical issues.  - Build framework and guide for developers.  - Coding  - Unit testing  - Create System Design |
| 3 | Nguyen Anh Tuan | Developer | - Create Database Design  - Coding  - Unit testing |
| 4 | Le Thi Minh Khanh | Tester  Designer | - Create test case  - Testing  - Design GUI  - Write meeting minutes |
| 5 | Nguyen Thi Gam | Tester | - Create SRS  - Testing  - Create documents, reports  - Write meeting minutes |

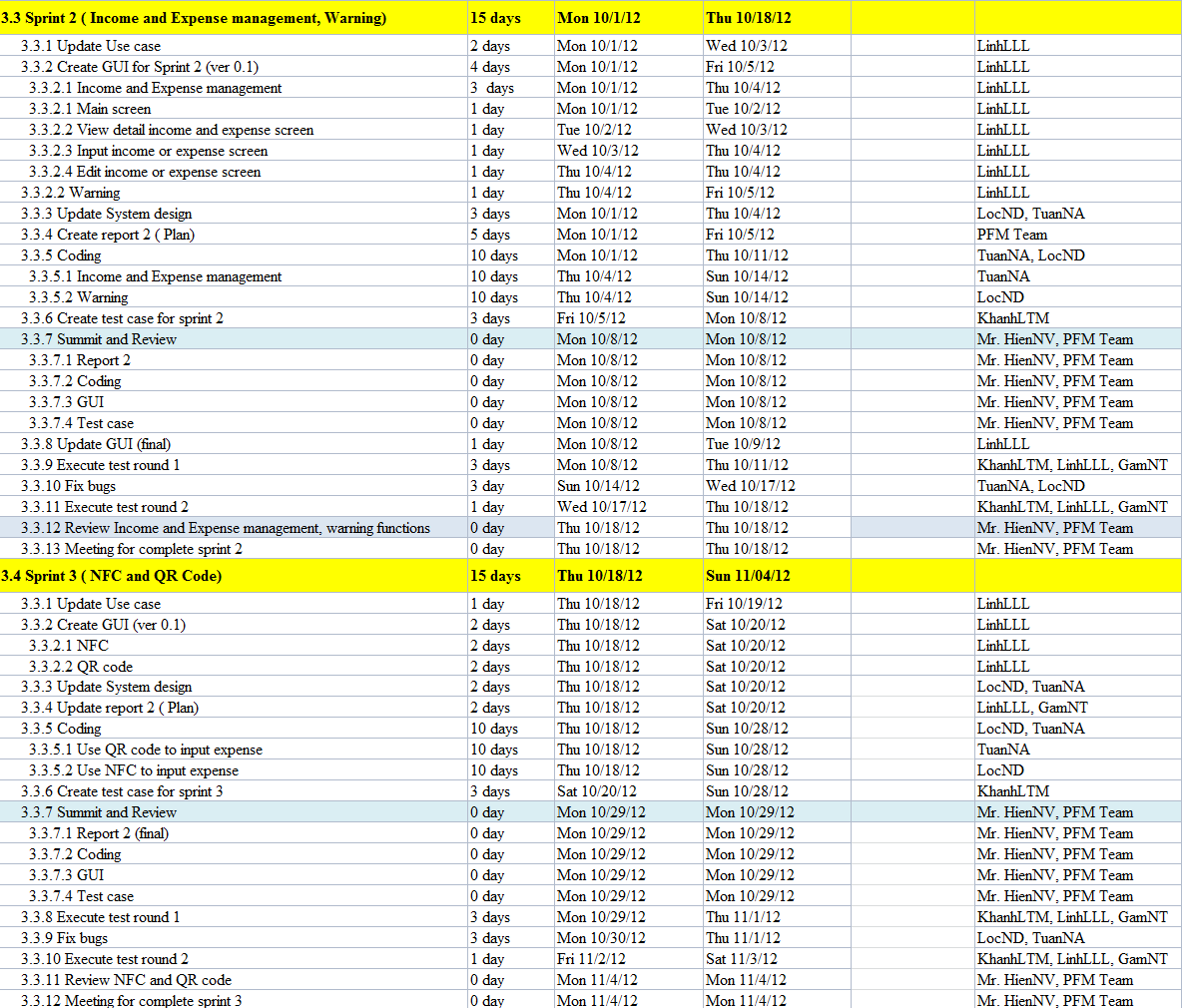
## Tools and Techniques

# Project Management Plan

## Task Sheet: Assignments and Timetable







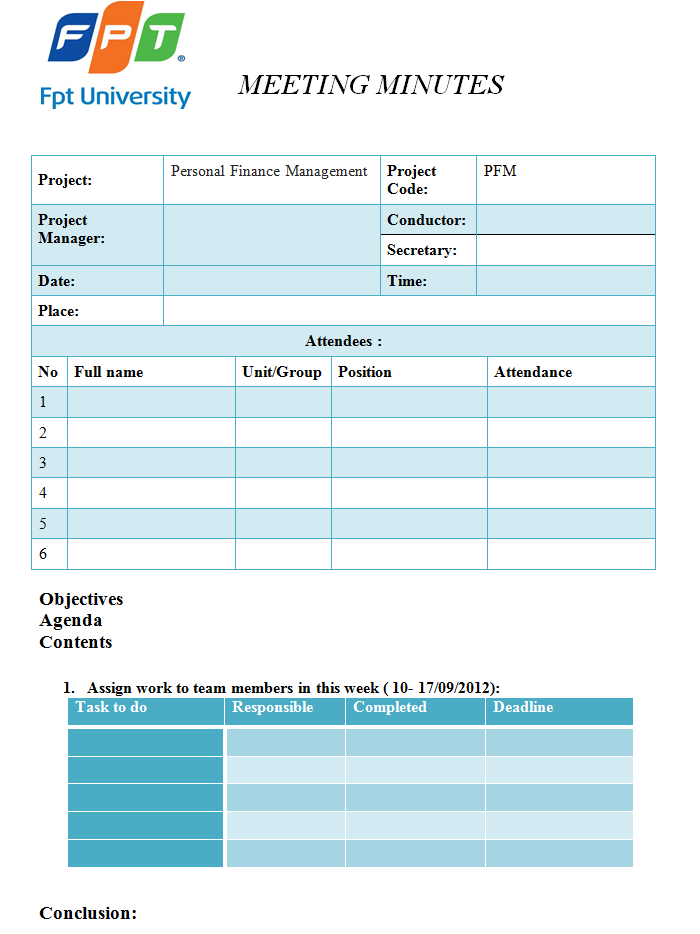


## Risks Management

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Risk Name | Probability | Impact | Description | Solution |
| 1 | Overlap |  |  |  | Modify plan and re-estimate tasks |
| 2 | Misunderstand requirements |  |  |  | Meeting to explain and discuss requirement |
| 3 | Un-adaptable design |  |  |  | Create survey, collect response and base on result to design |
| 4 | Conflict |  |  | Conflicts between team members. | Organize team-building |
| 5 | Illness or absence of team members |  |  |  |  |
| 6 | Experts not available when needed |  |  |  |  |
| 7 | New technology |  |  |  |  |
| 8 | Change in requirements |  |  |  |  |
| 9 | Equipment |  |  |  |  |

## All Meeting Minutes

Here is a template for meeting minute within the project’s life cycle.



# Coding Convention

We follow standard Java coding conventions. We add a few rules:

## Java Language Rules

### Don't Ignore Exceptions

Sometimes it is tempting to write code that completely ignores an exception like this:

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) { }

}

You must never do this. While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day. You must handle every Exception in your code in some principled way. The specific handling varies depending on the case.

Acceptable alternatives (in order of preference) are:

* Throw the exception up to the caller of your method.

*void setServerPort(String value) throws NumberFormatException {*

*serverPort = Integer.parseInt(value);*

*}*

* Throw a new exception that's appropriate to your level of abstraction.

*void setServerPort(String value) throws ConfigurationException {*

*try {*

*serverPort = Integer.parseInt(value);*

*} catch (NumberFormatException e) {*

*throw new ConfigurationException("Port " + value + " is not valid.");*

*}*

*}*

* Handle the error gracefully and substitute an appropriate value in the catch {} block.

*/\*\* Set port. If value is not a valid number, 80 is substituted. \*/*

*void setServerPort(String value) {*

*try {*

*serverPort = Integer.parseInt(value);*

*} catch (NumberFormatException e) {*

*serverPort = 80; // default port for server*

*}*

*}*

* Catch the Exception and throw a new RuntimeException. This is dangerous: only do it if you are positive that if this error occurs, the appropriate thing to do is crash.

*/\*\* Set port. If value is not a valid number, die. \*/*

*void setServerPort(String value) {*

*try {*

*serverPort = Integer.parseInt(value);*

*} catch (NumberFormatException e) {*

*throw new RuntimeException("port " + value " is invalid, ", e);*

*}*

*}*

Note that the original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you will need to omit the exception that is the cause.

* Last resort: if you are confident that actually ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason:

*/\*\* If value is not a valid number, original port number is used. \*/*

*void setServerPort(String value) {*

*try {*

*serverPort = Integer.parseInt(value);*

*} catch (NumberFormatException e) {*

*// Method is documented to just ignore invalid user input.*

*// serverPort will just be unchanged.*

*}*

*}*

### Fully Qualify Imports

When you want to use class Bar from package foo,there are two possible ways to import it:

import foo.\*;

Pros: Potentially reduces the number of import statements.

import foo.Bar;

Pros: Makes it obvious what classes are actually used. Makes code more readable for maintainers.

Decision: Use the latter for importing all Android code. An explicit exception is made for java standard libraries (java.util.\*, java.io.\*, etc.) and unit test code (junit.framework.\*)

## Java Style Rules

### Use Javadoc Standard Comments

Every file should have a copyright statement at the top. Then a package statement and import statements should follow, each block separated by a blank line. And then there is the class or interface declaration. In the Javadoc comments, describe what the class or interface does.

*/\**

*\* Copyright (C) 2010 The Android Open Source Project*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* http://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* Limitations under the License.*

*\*/*

*package com.android.internal.foo;*

*import android.os.Blah;*

*import android.view.Yada;*

*import java.sql.ResultSet;*

*import java.sql.SQLException;*

*/\*\**

*\* Does X and Y and provides an abstraction for Z.*

*\*/*

*public class Foo {*

*...*

*}*

Every class and nontrivial public method you write must contain a Javadoc comment with at least one sentence describing what the class or method does. This sentence should start with a 3rd person descriptive verb.

Examples:

*/\*\* Returns the correctly rounded positive square root of a double value. \*/*

*static double sqrt(double a) {*

*...*

*}*

Or

*/\*\**

*\* Constructs a new String by converting the specified array of*

*\* bytes using the platform's default character encoding.*

*\*/*

*public String(byte[] bytes) {*

*...*

*}*

### Limit Variable Scope

The scope of local variables should be kept to a minimum (Effective Java Item 29). By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

One exception to this rule concerns try-catch statements. If a variable is initialized with the return value of a method that throws a checked exception, it must be initialized inside a try block. If the value must be used outside of the try block, then it must be declared before the try block, where it cannot yet be sensibly initialized:

*// Instantiate class cl, which represents some sort of Set*

*Set s = null;*

*try {*

*s = (Set) cl.newInstance();*

*} catch(IllegalAccessException e) {*

*throw new IllegalArgumentException(cl + " not accessible");*

*} catch(InstantiationException e) {*

*throw new IllegalArgumentException(cl + " not instantiable");*

*}*

*// Exercise the set*

*s.addAll(Arrays.asList(args));*

But even this case can be avoided by encapsulating the try-catch block in a method:

*Set createSet(Class cl) {*

*// Instantiate class cl, which represents some sort of Set*

*try {*

*return (Set) cl.newInstance();*

*} catch(IllegalAccessException e) {*

*throw new IllegalArgumentException(cl + " not accessible");*

*} catch(InstantiationException e) {*

*throw new IllegalArgumentException(cl + " not instantiable");*

*}*

*}*

*...*

*// Exercise the set*

*Set s = createSet(cl);*

*s.addAll(Arrays.asList(args));*

Loop variables should be declared in the for statement itself unless there is a compelling reason to do otherwise:

*for (int i = 0; i n; i++) {*

*doSomething(i);*

*}*

and

*for (Iterator i = c.iterator(); i.hasNext(); ) {*

*doSomethingElse(i.next());*

*}*

### Follow Field Naming Conventions

* Non-public, non-static field names start with m.
* Static field names start with s.
* Other fields start with a lower case letter.
* Public static final fields (constants) are ALL\_CAPS\_WITH\_UNDERSCORES.

For example:

*public class MyClass {*

*public static final int SOME\_CONSTANT = 42;*

*public int publicField;*

*private static MyClass sSingleton;*

*int mPackagePrivate;*

*private int mPrivate;*

*protected int mProtected;*

*}*

### Use Standard Java Annotations

Annotations should precede other modifiers for the same language element. Simple marker annotations (e.g. @Override) can be listed on the same line with the language element. If there are multiple annotations, or parameterized annotations, they should each be listed one-per-line in alphabetical order.<

Android standard practices for the three predefined annotations in Java are:

* @Deprecated: The @Deprecated annotation must be used whenever the use of the annotated element is discouraged. If you use the @Deprecated annotation, you must also have a @deprecated Javadoc tag and it should name an alternate implementation. In addition, remember that a @Deprecated method is still supposed to work.

If you see old code that has a @deprecated Javadoc tag, please add the @Deprecated annotation.

* @Override: The @Override annotation must be used whenever a method overrides the declaration or implementation from a super-class.

For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method.

* @SuppressWarnings: The @SuppressWarnings annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation must be used, so as to ensure that all warnings reflect actual problems in the code.

When a @SuppressWarnings annotation is necessary, it must be prefixed with a TODO comment that explains the "impossible to eliminate" condition. This will normally identify an offending class that has an awkward interface. For example:

*// TODO: The third-party class com.third.useful.Utility.rotate() needs generics*

*@SuppressWarnings("generic-cast")*

*List<String> blix = Utility.rotate(blax);*

When a @SuppressWarnings annotation is required, the code should be refactored to isolate the software elements where the annotation applies.

### Treat Acronyms as Words

Treat acronyms and abbreviations as words in naming variables, methods, and classes. The names are much more readable:

|  |  |
| --- | --- |
| **Good** | **Bad** |
| XmlHttpRequest | XMLHTTPRequest |
| getCustomerId | getCustomerID |
| class Html | class HTML |
| String url | String URL |
| long id | long ID |

Both the JDK and the Android code bases are very inconsistent with regards to acronyms, therefore, it is virtually impossible to be consistent with the code around you. Bite the bullet, and treat acronyms as words.

## Javatests Style Rules

### Follow Test Method Naming Conventions

When naming test methods, you can use an underscore to seperate what is being tested from the specific case being tested. This style makes it easier to see exactly what cases are being tested.

For example:

*testMethod\_specificCase1 testMethod\_specificCase2*

*void testIsDistinguishable\_protanopia() {*

*ColorMatcher colorMatcher = new ColorMatcher(PROTANOPIA)*

*assertFalse(colorMatcher.isDistinguishable(Color.RED, Color.BLACK))*

*assertTrue(colorMatcher.isDistinguishable(Color.X, Color.Y))*

*}*

### 