**International School, Duy Tan University**



CAPSTONE PROJECT 2

An Phu Farm System

**CODE STANDARD DOCUMENT**

Version 1.0

28/04/2016

**Mentor : MSc. Nguyen Minh Nhat**

**Group name : Divine Team**

**PROJECT OVERVIEW**

|  |  |
| --- | --- |
| Project Title | An Phu Farm System |
| Start Date – End Date | 20/01/2016 to 22/05/2016 |
| Lead Institution | International School, Duy Tan University |
| Project Mentor | Msc Nguyen Minh Nhat |
| Project Manager & Contact detail | Project lead, Doan Hai Dang  K18CMUTPM1, International School, Duy Tan University  Mail: [doanhaidang32@gmail.com](mailto:doanhaidang32@gmail.com)  Tel: +84 1202348323 |
| Team member | Nguyen Ba Chinh, [nguyenbachinh94@gmail.com](mailto:nguyenbachinh94@gmail.com)  Le Hoang Phuc, [lehoangphucdtu@gmail.com](mailto:lehoangphucdtu@gmail.com)  Le Dinh Doan, ledinhdoan.0083@gmail.com |

**DOCUMENT**

|  |  |
| --- | --- |
| Document Title | Code Standard document |
| Reporting period | April, 2016 |
| Author | Team Divine |
| Role | Team member |
| Filename | CodeStandardDocument \_v1.0 |

document approval

The following signatures are required for approval of this document. Note, however, that since some role assignments will change during the course of the project, the perspective names should be update if needed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. | F. Name | L. Name | Role | Date | Signature |
| 01 | Nguyen Minh | Nhat | P.Owner |  |  |
| 02 | Doan Hai | Dang | Team Leader |  |  |
| 03 | Nguyen Ba | Chinh | Member |  |  |
| 04 | Le Hoang | Phuc | Member |  |  |
| 05 | Le Dinh | Doan | Member |  |  |

Contents

[1. INTRODUCTION 5](#_Toc435801288)

[2. PURPOSE OF CODING STANDARDS AND BEST PRACTICES 5](#_Toc435801289)

[3. HOW TO FOLLOW THE STANDARDS ACROSS THE TEAM 6](#_Toc435801290)

[4. CONVENTION 7](#_Toc435801291)

[5. NAMING CONVENTIONS AND STANDARDS 10](#_Toc435801292)

[6. COMMENT FORMATS 14](#_Toc435801293)

[7. BE CONSISTENT 16](#_Toc435801294)

[8. CONFIGURATION 16](#_Toc435801294)

[9. SERVICES 16](#_Toc435801294)

[10. AOP 19](#_Toc435801294)

[11. SPRING MVC 21](#_Toc435801294)

12[. REFERENCE 24](#_Toc435801295)

1. INTRODUCTION

Anybody can write code. With a few months of programming experience, you can write 'working applications'. Making it work is easy, but doing it the right way requires more work, than just making it work. Believe it, majority of the programmers write 'working code', but not ‘good code'. Writing 'good code' is an art and you must learn and practice it.

Everyone may have different definitions for the term ‘good code’. In my definition, the following are the characteristics of good code.

* Reliable
* Maintainable
* Efficient

Most of the developers are inclined towards writing code for higher performance, compromising reliability and maintainability. But considering the long term ROI (Return On Investment), efficiency and performance comes below reliability and maintainability. If your code is not reliable and maintainable, you (and your company) will be spending lot of time to identify issues, trying to understand code etc. throughout the life of your application.

1. PURPOSE OF CODING STANDARDS AND BEST PRACTICES

To develop reliable and maintainable applications, you must follow coding standards and best practices.

The naming conventions, coding standards and best practices described in this document are compiled from our own experience and by referring to various Microsoft and non-Microsoft guidelines.

There are several standards exists in the programming industry. None of them are wrong or bad and you may follow any of them. What is more important is, selecting one standard approach and ensuring that everyone is following it.

1. HOW TO FOLLOW THE STANDARDS ACROSS THE TEAM

If you have a team of different skills and tastes, you are going to have a tough time convincing everyone to follow the same standards. The best approach is to have a team meeting and developing your own standards document. You may use this document as a template to prepare your own document.

Distribute a copy of this document (or your own coding standard document) well ahead of the coding standards meeting. All members should come to the meeting prepared to discuss pros and cons of the various points in the document. Make sure you have a manager present in the meeting to resolve conflicts.

Discuss all points in the document. Everyone may have a different opinion about each point, but at the end of the discussion, all members must agree upon the standard you are going to follow. Prepare a new standards document with appropriate changes based on the suggestions from all of the team members. Print copies of it and post it in all workstations.

After you start the development, you must schedule code review meetings to ensure that everyone is following the rules. 3 types of code reviews are recommended:

1. Peer review – another team member review the code to ensure that the code follows the coding standards and meets requirements. This level of review can include some unit testing also. Every file in the project must go through this process.
2. Architect review – the architect of the team must review the core modules of the project to ensure that they adhere to the design and there is no “big” mistakes that can affect the project in the long run.
3. Group review – randomly select one or more files and conduct a group review once in a week. Distribute a printed copy of the files to all team members 30 minutes before the meeting. Let them read and come up with points for discussion. In the group review meeting, use a projector to display the file content in the screen. Go through every sections of the code and let every member give their suggestions on how could that piece of code can be written in a better way. (Don’t forget to appreciate the developer for the good work and also make sure he does not get offended by the “group attack”!)
4. CONVENTION

We follow standard Java coding conventions. We add a few Android specific rules.

Package and Import Statements .

The first non-comment line of most Java source files is a package statement. After that,

import statements can follow. For example:

package java.awt;

import java.awt.peer.CanvasPeer;

Order Import Statements

The ordering of import statements is:

* + - 1. Android imports
      2. Imports from third parties (com, junit, net, org)
      3. Java and javax

To exactly match the IDE settings, the imports should be:

Alphabetical within each grouping, with capital letters before lower case letters (e.g. Z before a).

There should be a blank line between each major grouping (android, com, junit, net, org, java, javax).

Fully Qualify Imports

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

import foo.\*;

import foo.Bar;

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.\*)

Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

int size; // size of table

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

* No space between a method name and the parenthesis “(“ starting its parameter list
* Open brace “{” appears at the end of the same line as the declaration statement
* Closing brace “}” starts a line by itself indented to match its corresponding opening

statement, except when it is a null statement the “}” should appear immediately after:

“{“ 2

class Sample extends Object {

int ivar1;

int ivar2;

Sample(int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

...

}

Methods are separated by a blank line

We use 4 space indents for blocks. We never use tabs. When in doubt, be consistent with code around you.

We use 8 space indents for line wraps, including function calls and assignments. For example, this is correct:

Instrument i =

someLongExpression(that, wouldNotFit, on, one, line);

and this is not correct:

Instrument i =

someLongExpression(that, wouldNotFit, on, one, line);

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;

}

1. NAMING CONVENTIONS AND STANDARDS

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

|  |  |  |
| --- | --- | --- |
| **Identifier Type** | **Rules for Naming** | **Examples** |
| Packages | The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.  Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names. | com.sun.eng  com.apple.quicktime.v2  edu.cmu.cs.bovik.cheese |
| Classes | Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). | class Raster; class ImageSprite; |
| Interfaces | Interface names should be capitalized like class names and always prefix with capital “I”. | interface IDelegate; interface IStoring; |
| Methods | Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized. | run(); runFast(); getBackground(); |
| Variables | Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore \_ or dollar sign $ characters, even though both are allowed.  Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters. | int i;  char c;  float myWidth; |
| Constants | The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("\_"). (ANSI constants should be avoided, for ease of debugging.) | static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 999; |

**Use Standard Brace Style**

Braces do not go on their own line; they go on the same line as the code before them. So:

class MyClass {

int func() {

if (something) {

// ...

} else if (somethingElse) {

// ...

} else {

//

}

We require braces around the statements for a conditional.

if (condition) {

body();

}

while Statements

A while statement should have the following form:

while (condition) {

statements;

}

do-while Statements

A do-while statement should have the following form:

do {

statements;

} while (condition);

switch Statements

A switch statement should have the following form:

switch (condition) {

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

break;

}

1. COMMENT FORMATS

Block comments are used to provide descriptions of files, methods, data structures and algorithms.

* Short comments can appear on a single line indented to the level of the code that follows.
* Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.
* The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string TODO in all caps, followed by a colon:

// TODO: Remove this code after the UrlTable2 has been checked in.

and

// TODO: Change this to use a flag instead of a constant.

If your TODO is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code after all production mixers understand protocol V7.").

1. BE CONSISTENT

* Our parting thought: BE CONSISTENT. If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.
* The point of having style guidelines is to have a common vocabulary of coding, so people can concentrate on what you're saying, rather than on how you're saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it.

1. CONFIGURATION

Spring is the de facto IOC container used at *47 Degrees* for Java server side applications. Both XML and annotation based configurations are being used. [XML configuration](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html) is favored for those services and beans that have not originally been annotated with either [@Component](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/stereotype/Component.html), [@Service](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/stereotype/Service.html), [@Controller](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/stereotype/Controller.html)... or that require certain depedencies that are easier to wire via xml. [Annotations](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-annotation-config) are preffered for Services in local packages and default Service interface implementations for applications that use package scanning to wire and discover the application context.

1. SERVICES

Spring services should always be implementations of a Service interface that hides the implementations details when being used in other services dependencies. Consider the following requirements.

* The application needs a UserService to manipulate User model beans.
* The User service should hide any dependencies it has in other services or 3rd party technologies.
* The service implementation could be replaced at runtime via Abstract Factories or Injection through service id qualifiers.

**Correct:**

/\*\*

\* Defines the contract for the user related operations

\* @see User

\*/

public interface UserService {

/\*\*

\* Persists a user

\*

\* @param user the user to be persisted

\*/

void save(User user);

}

/\*\*

\* Implements the UserService utilizing the Persistence Adapter for storage persistence

\* @see UserService

\*/

@Service

public class LocalUserServiceImpl implements UserService {

/\*\*

\* A storage independent façade for object persistence

\*/

@Autowired

private PersistenceAdapter persistenceAdapter;

/\*\*

\* Persists a user

\*

\* @param user the user to be persisted

\*/

@Override

public void save(User user) {

persistenceAdapter.persist(user);

}

}

**Incorrect:**

/\*\*

\* Utilizes the Persistence Adapter for storage persistence

\* @see UserService

\*/

@Service

public class UserService {

/\*\*

\* A storage independent façade for object persistence

\*/

@Autowired

private PersistenceAdapter persistenceAdapter;

/\*\*

\* Persists a user

\*

\* @param user the user to be persisted

\*/

public void save(User user) {

persistenceAdapter.persist(user);

}

}

1. AOP

[Spring AOP](http://static.springsource.org/spring/docs/3.0.x/reference/aop.html) is extensively used in server side Java based projects at *47 Degrees*. Developers should be aware that their method classes and method invokations may be decorated, intercepted, validated and that their service runtime instances may be [proxied](http://static.springsource.org/spring/docs/3.0.x/reference/aop.html" \l "aop-understanding-aop-proxies). This is necessary to implement some of the AOP design patterns that are provided by Spring to such as[Security](http://static.springsource.org/spring-security/site/docs/3.0.x/apidocs/org/springframework/security/access/annotation/Secured.html), [Transactions](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/transaction.html#transaction-declarative-annotations), Logging, Events, etc.

The [AspectJ Annotation](http://static.springsource.org/spring/docs/2.5.5/reference/aop.html#aop-ataspectj) style is preffered over other AOP flavors.

You can find AOP patterns in use across many layers.

**Security:**

@Override

@Secured(SecureRoles.ADMIN) // enforces method access to only admins

public void save(User user) {

...

}

}

**Transactions:**

@Override

@Transactional // creates, opens and closes a transaction around this method invokation

public void save(User user) {

...

}

}

**Events**

@Override

@EventListener(Events.ON\_USER\_SAVE\_REQUEST) // notifies this method whenever other service invokes eventService.publish(Events.ON\_USER\_SAVE\_REQUEST, user);

public void save(User user) {

...

}

}

**Logging**

@Logger //injects the application logger into this service implementation

private Log logger;

1. SPRING MVC

Spring MVC is utilized at *47 Degrees* for both building API's and Webapps. All classes exposed as Web or API controllers should include the [@Controller](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/stereotype/Controller.html)

annotation and be configured by using the [@RequestMapping](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/bind/annotation/RequestMapping.html) annotations that determine which paths and variables are mapped to methods.

11.1. API's

All API's Services should follow the same pattern described in the [Services](https://github.com/47deg/coding-guidelines/tree/master/java/spring#2-services) section. Below is an example of an API service implementation that exposes a method at http(s)://host/api/vi1/users/{accessToken}/user/{userId}.

/\*\*

\* Default impl for the UserAPI that maps url requests to methods

\*/

@Controller

@RequestMapping("/api/v1/users")

public class UserAPIImpl implements UserAPI {

@Autowired

private UserService userService;

/\*\*

\* Fetches a user by id

\* This handler maps path variables to method arguments and returns a serialized representation of a UserResponse

\*

\* @param accessToken the requesting user accessToken

\* @param userId the id for the user being fetched

\*/

@Override

@RequestMapping(value = "/{accessToken:.+}/user/{userId}", method = RequestMethod.GET)

public @ResponseBody UserResponse getAuthenticatedProfile(@PathVariable("accessToken") String accessToken, @PathVariable("userId") String userId) {

User requester = userService.getAccountForToken(accessToken);

User foundUser = userService.getUser(requester, userId);

return new UserResponse(foundUser);

}

}

11.2. Web apps

Webapps controllers do not require to extend from a service interface as they are tightly coupled to the views they handle. Below is an example of an Web page implementation with its corresponding view

**Controller**

/\*\*

\* User page

\*/

@Controller

@RequestMapping("/users")

public class UserAPIImpl implements UserAPI {

@Autowired

private UserService userService;

/\*\*

\* Fetches a user by id

\* This handler maps path variables to method arguments and dispatch to the appropiate view setting the user model variable used to render the page

\*

\* @param accessToken the requesting user accessToken

\* @param userId the id for the user being fetched

\*/

@Override

@RequestMapping(value = "/{userId}", method = RequestMethod.GET)

public ModelAndView getAuthenticatedProfile(ModelAndView mav, @PathVariable("userId") String userId) {

mav.setViewName("users");

User foundUser = userService.getUser(userId);

UserResponse userResponse = new UserResponse(foundUser);

mav.addObject("user", userResponse);

return mav;

}

}

**View**

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<!DOCTYPE html>

<html lang="en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

<title>Users</title>

</head>

<body>

<div>${user.firstName}</div>

</body>

</html>

1. REFERENCE

[1]. http://source.android.com/source/code-style.html

[2]. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[3] . https://github.com/47deg/coding-guidelines/tree/master/java/spring

**Divine Team**

|  |  |  |
| --- | --- | --- |
| Le Dinh Doan  Mr    K18CMUTM1 | | |
| Ngyen Ba Chinh  Mr    K18CMUTM1 | |  |
| Le Hoang Phuc  Mr    K18CMUTPM1 | |  |
| Doan Hai Dang  Mr    K18CMUTPM1 | |  |
| ***Approved by*** | ***Signature*** | ***Date*** |
| Nguyen Minh Nhat  Msc, Mentor  Duy Tan University | |  |