POLYTECHNIC UNIVERSITY OF MILAN

School of Industrial and Information Engineering

Computer Science and Engineering

Project of Software Engineering 2: Glassfish 4.1 Application Server Code Inspection

Course Professor: Prof. Elisabetta DI NITTO

Authors:

Mattia CRIPPA 854126

Francesca GALLUZZI 788328

Marco LATTARULO 841399

Academic Year 2015–2016

Contents

List of Tables

Chapter 1

Introduction

Code review is probably the single-most effective technique for identifying security flaws. When used together with automated tools and manual penetration testing, code review can significantly increase the cost effectiveness of an application security verification effort. Manual security code review provides insight into the "real risk" associated with insecure code. This is the single most important value from a manual approach. A human reviewer can understand the context for certain coding practices, and make a serious risk estimate that accounts for both the likelihood of attack and the business impact of a breach.

1.1 Purpose

Code reviews have two purposes. Their first purpose is to make sure that the code that is being produced has sufficient quality to be released. Code reviews are very effective at finding errors of all types, including those caused by poor structure, those that don't match business process, and also those caused by simple omissions. The second purpose is as a teaching tool to help developers learn when and how to apply techniques to improve code quality, consistency, and maintainability.

1.2 Coding Conventions

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of

a program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc. These are guidelines for software structural quality. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project. We decided to follow the *"Oracle Java Code Conventions"* for our code inspection.

1.3 Issues Checklist

This is a comprehensive tabular checklist of possible issues that we used during the inspection process.

Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;

4. Interface names should be capitalized like classes.

5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.

6. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: `_windowHeight`, `timeSeriesData`.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

8. Three or four spaces are used for indentation and done so consistently.

9. No tabs are used to indent.

Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

11. All `if`, `while`, `do-while`, `try-catch`, and `for` statements that have only one statement to execute are surrounded by curly braces. Example: avoid this:

```
if ( condition )
    doThis();
```

instead do this:

```
if ( condition )
{
    doThis();
}
```

File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

13. Where practical, line length does not exceed 80 characters.

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.

16. Higher-level breaks are used.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

Java Source Files

20. Each Java source file contains a single public class or interface.

21. The public class is the first class or interface in the file.

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

Class and Interface Declarations

25. The class or interface declarations shall be in the following order:
    (a) class/interface documentation comment;
    (b) class or interface statement;
    (c) class/interface implementation comment, if necessary;
    (d) class (static) variables;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);
        iv. last private instance variables.
    (f) constructors;
    (g) methods.
26. Methods are grouped by functionality rather than by scope or accessibility.
27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).
29. Check that variables are declared in the proper scope.
30. Check that constructors are called when a new object is desired.

31. Check that all object references are initialized before use.

32. Variables are initialized where they are declared, unless dependent upon a computation.

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '`{`' and '`}`'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. Check that parameters are presented in the correct order.

35. Check that the correct method is being called, or should it be a different method with a similar name.

36. Check that method returned values are used properly.

Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

39. Check that constructors are called when a new array item is desired.

Object Comparison

40. Check that all objects (including Strings) are compared with `equals` and not with `==`.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.

43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).

45. Check order of computation/evaluation, operator precedence and parenthesizing.

46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

47. Check that all denominators of a division are prevented from being zero.

48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.

49. Check that the comparison and Boolean operators are correct.

50. Check throw-catch expressions, and check that the error condition is actually legitimate.

51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.

53. Check that the appropriate action are taken for each catch block.

Flow of Control

54. In a `switch` statement, check that all cases are addressed by break or return.

55. Check that all switch statements have a default branch.

56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.

58. Check that all files are closed properly, even in the case of an error.

59. Check that EOF conditions are detected and handled correctly.

60. Check that all file exceptions are caught and dealt with accordingly.

1.4 References

- Assignment 3 - Code Inspection.pdf.
- Link referenced to *"Oracle Java Code Conventions"*: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

1.5 Document Overview

This document is essentially structured in three parts:

- **Section 1** → Introduction: it gives a description of the document and gives general information about "Coding Conventions" and the "Checklist" used in order to detect issues in the code.
- **Section 2** → Code Inspection Process : it gives a description of the classes assigned and contains all the problems detected during the code inspection.
- **Section 3** → Other Information: this part contains information about the total working hours of each group member.

Chapter 2

Code Inspection Process

2.1 Assigned Methods

This section contains the namespace pattern and the name of the methods that were assigned to us:

- **Name**: *getResourceAsStream(**String** path)*
  Location: *appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java*

- **Name**: *getResource(**String** path)*
  Location: *appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java*

- **Name**: *getResourcePaths(**String** path)*
  Location: *appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java*

- **Name**: *getResourcePathsInternal(**DirContext** resources, **String** path)*
  Location: *appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java*

- **Name**: *getRequestDispatcher(**String** path)*
  Location: *appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java*

2.2 Functional Role of Methods

This section contains the functional role of the methods that were assigned to us:

- **Name**: *getResourceAsStream(**String** path)*
  Role: Return the requested resource as an **InputStream**. The path must be specified according to the rules described under *getResource*. If no such resource can be identified, return **null**.

- **Name**: *getResource(**String** path)*
  Role: Return the URL to the resource that is mapped to a specified path. The path must begin with a "/" and is interpreted as relative to the current context root.

- **Name**: *getResourcePaths(**String** path)*
  Role: Return a Set containing the resource paths of resources member of the specified collection. Each path will be a **String** starting with a "/" character. The returned set is immutable.

- **Name**: *getResourcePathsInternal(**DirContext** resources, **String** path)*
  Role: Internal implementation of *getResourcesPath()* logic.

- **Name**: *getRequestDispatcher(**String** path)*
  Role: Return a **RequestDispatcher** instance that acts as a wrapper for the resource at the given path. The path must begin with a "/" and is interpreted as relative to the current context root.

2.3 Issues

In this section are presented all the issues detected during our code inspection process for each method assigned to us:

getResourceAsStream(String path)

Checklist Point	Issue
11	Every time an *if* statements contains only a *return()* statement in his body, this is not surrounded by braces
15	At line 7500 the line is broken before the OR operator
18	Even though all the main classes, methods and interfaces are well commented, there is a large number of blocks of code not commented at all
25	The Constructor is placed between Static Variables and Instance Variables. Private and protected Instance Variables alternates (i.e. line 630)
27	The entire class consists of more than 8000 lines of code, that are too many. There is not a fixed maximum size for a class, but 8000 lines are surely too many for every style convention commonly used
42	There is no error message associated with the only one exception present in the method
53	The only one exception present in the method has an empty body

Table 2.1: Issues for *getResourceAsStream(**String** path)*

getResource(String path)

Checklist Point	Issue
09	At line 7586, two tabs are used
11	Every time an *if* statements contains only a *return()* statement in his body, this is not surrounded by braces
13	At lines 7535 and 7567 line length exceeds 80 characters
15	At lines 7560 and 7579 the line is broken respectively before the OR operator and before the bracket
18	Even though all the main classes, methods and interfaces are well commented, there is a large number of blocks of code not commented at all
19	The commented out code at lines from 7580 to 7582 does not contains the reason for being commented out and does not contain the date it can be removed
25	The Constructor is placed between Static Variables and Instance Variables. Private and protected Instance Variables alternates (i.e. line 630)
27	The entire class consists of more than 8000 lines of code, that are too many. There is not a fixed maximum size for a class, but 8000 lines are surely too many for every style convention commonly used
42	There is no error message associated with the exception at line 7587
53	The exception at line 7587 has an empty body

Table 2.2: Issues for *getResource(**String** path)*

getResourcePaths(String path)

Checklist Point	**Issue**
11	Every time an *if* statements contains only a *return()* statement in his body, this is not surrounded by braces
13	At line 7607 line length exceeds 80 characters
15	At line 7617 the line is broken before the OR operator
18	Even though all the main classes, methods and interfaces are well commented, there is a large number of blocks of code not commented at all
25	The Constructor is placed between Static Variables and Instance Variables. Private and protected Instance Variables alternates (i.e. line 630)
27	The entire class consists of more than 8000 lines of code, that are too many. There is not a fixed maximum size for a class, but 8000 lines are surely too many for every style convention commonly used

Table 2.3: Issues for *getPaths(**String** path)*

getResourcePathsInternal(DirContext resources, String path)

Checklist Point	Issue
23	This method is not present at all in the Javadoc
25	The Constructor is placed between Static Variables and Instance Variables. Private and protected Instance Variables alternates (i.e. line 630)
27	The entire class consists of more than 8000 lines of code, that are too many. There is not a fixed maximum size for a class, but 8000 lines are surely too many for every style convention commonly used
39	At line 7657 the array *File[]* is filled without the use of a constructor
42	Both the exceptions at lines 7649 and 7671 do not contain error messages
53	Both the exceptions at lines 7649 and 7671 have an empty body and thus they do nothing

Table 2.4: Issues for *getResourcePathsInternal(**DirContext** resources, **String** path)*

getRequestDispatcher(String path)

Checklist Point	Issue
11	Every time an *if* statements contains only a *return()* statement in his body, this is not surrounded by braces
13	At line 7690 line length exceeds 80 characters
15	At line 7759 the line is broken before the bracket
25	The Constructor is placed between Static Variables and Instance Variables. Private and protected Instance Variables alternates (i.e. line 630)
27	The entire class consists of more than 8000 lines of code, that are too many. There is not a fixed maximum size for a class, but 8000 lines are surely too many for every style convention commonly used

Table 2.5: Issues for *getRequestDispatcher(**String** path)*

Chapter 3

Other Information

3.1 Working Hours

First Name	Last Name	Total Hours
Mattia	Crippa	18h
Francesca	Galluzzi	18h
Marco	Lattarulo	18h