POLITECNICO
MILANO 1863

Code Inspection Document for Apache OFBiz® during the PowerEnJoy project

Daniele Riva[*] Marco Sartini[†]

February 5, 2017

version 1.0

[*]matr. 875154
[†]matr. 877979

Contents

1. Introduction

1.1. Purpose

This document reports the code analysis performed on the `ScrumService` class, part of the Apache OFBiz project, as explained in details through the chapters. The code inspection has the scope of review the code to find out possible mistakes.

1.2. Scope

This document is a part of the Software Engineering II project, which main purpose was to design a platform based on mobile and web application thought to offer a car sharing service with electrical powered cars called *Power EnJoy*.

1.3. Definitions, acronyms, abbreviations

EOF end of file;

Revision history

Name	Date	Reason For Changes	Version
Marco e Daniele	05/02/2017	Initial	1.0

2. Code description

2.1. Assigned class and methods

Our group was assigned the `ScrumService.java` in the `org.apache.ofbiz.scrum` package.

The path to the class in the provided wrapper is: `apache-ofbiz-16.11.01/specialpurpose/scrum/src/main/java/org/apache/ofbiz/scrum/ScrumServices.java`

2.2. Functional role

2.2.1. Apache OFBiz®

Apache OFBiz (contraction for *Open For Business*) is a large software, mainly for enterprises, created to support them in the automation of their processes, from ERP to CRM, SCM and similar, basically in the information systems section. It is open source (under the *Apache License Version 2.0*) and it is a customizable, really flexible framework to solve business needs.

2.2.2. Scrum

Scrum is a framework from Ken Schwaber and Jeff Sutherland, conceived by Hirotaka Takeuchi and Ikujiro Nonaka in the 1986. It represents a new approach in software (and widely, products) development, standing on the empiric evidence of requirements volatility.

To adapt the developing processes to the unavoidable and unforeseen changes, the *Scrum* approach acts as a sort of a forerunner of the *agile* method. The approach is based on an iterative, incremental, flexible and holistic development strategy, where a development team works as a unit to reach a common goal.

To manging the file reviews and the multi-developer editing during the whole process, the framework rely on a *version control system*, with the aim of having a coordinated, automated and consistent project.

2.2.3. Scrum in OFBiz

The product includes, along with many business solutions, also an implementation of the Scrum framework.

Of course as previously stated about a version controller, this implementation relies (without surprise) on the Apache *Subversion* system. The methods of the analyzed class

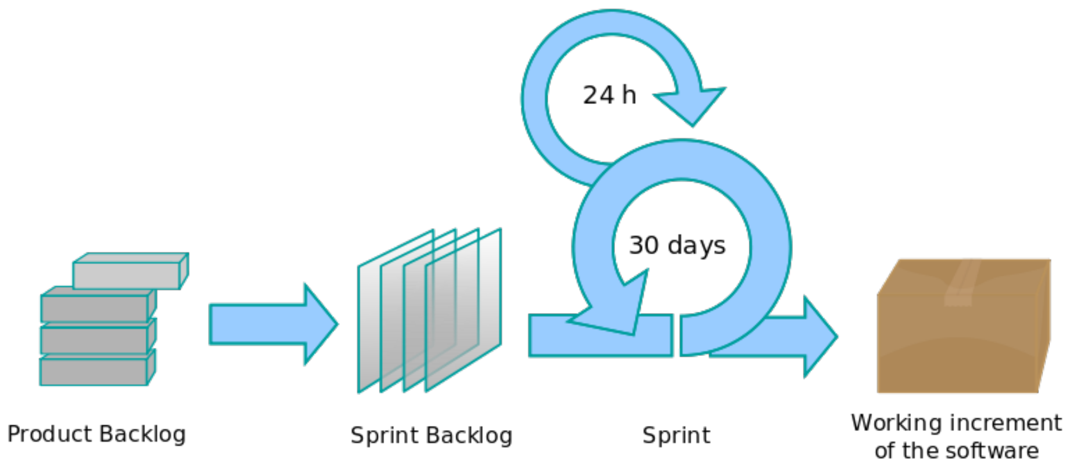

Figure 2.1.: The Scrum process.
Image by Lakeworks (Own work) [GFDL or CC BY-SA 4.0-3.0-2.5-2.0-1.0], via Wikimedia Commons

manage the interaction between OFBiz and the version controller. In particular, they interact with the `svn` client to mainly synchronize the internal database of OFBiz with the revisions in the Subversion.

linkToProduct this method receives as parameters a `DispatchContext` object and a `Map` of objects to strings; it returns a `Map` of objects to strings.

The aim of this method is to produce a link between a particular subject received by the caller context and a product in the database; if the product is found, it will be connected to a special `CommunicationEvent` entity, also associated to the login of the user responsible for the call;

viewScrumRevision this method receives as parameters a `DispatchContext` object and a `Map` of objects to strings; it returns a `Map` of objects to strings.

The aim of this method is to acquire the log of the revision requested and the differences between the last revision and the previous one.

This method bring the path of the repository and the revision number from the parameter Map, it builds the proper command to be invoked by the console to get the commits log and it concatenates all those logs into a String; the same operation is executed with the *diff* specification to get the differences between the two revisions. It returns an object containing the results of the operations and a bunch of supplementary information.

retrieveMissingScrumRevision this method receives as parameters a `DispatchContext` object and a `Map` of objects to strings; it returns a `Map` of objects to strings.

The aim of this method is to maintain the OFBiz database updated with respect to the revisions on Subversion: this method looks for revisions in the svn but not yet present in the database.

If the task ends successfully, it is returned a success entity.

removeDuplicateScrumRevision this method receives as parameters a `DispatchContext` object and a `Map` of objects to strings; it returns a `Map` of objects to strings.

The aim of this method is to remove from the OFBiz database duplicate revisions.

If the task ends successfully, it is returned a success entity.

3. Code issues

in this chapter there is a description of the bad and good practices adopted by the Apache OFBiz developers in the writing of the `ScrumService` class.

3.1. Checklist issues

In the following list we will stress the discrepancy among the checklist in the appendix and the analyzed code.

7 Constant variables (declared as `static final`) are not described with upper cases: at line 51 `module` should be `MODULE` and at line 52 `resource` should be `RESOURCE`;

12 At line 19, the `package` statement is not separate from the comment at the beginning; at line 53, the class variables declaration is not separated from the method signature;

13 The following lines are longer than 80 characters: 53, 57, 58, 62, 70, 73, 74, 75, 77, 78, 79, 83, 84, 85, 86, 87, 90, 91, 94, 101, 107, 121, 130, 134, 139, 141, 166, 174, 181, 186, 205, 206, 209, 210, 211, 213, 214, 215, 216, 217, 218, 220, 221, 222, 225, 229, 230, 259, 266, 267, 268, 269, 270, 272, 284, 287, 288, 289;

14 The following lines are longer than 120 characters: 62, 70, 74, 75, 77, 79, 83, 87, 91, 94, 101, 107, 139, 166, 205, 209, 214, 215, 216, 217, 220, 259, 266, 267, 270, 272, 287, 288, 289;

19 There are commented lines of code without expiration data, precisely at 58 and 74;

23 *Javadoc* is poor, it is brief and not exhaustive; the `linkToProduct` method at line 53 has no *Javadoc* at all, and not even the class variables;

27 The class itself is not very huge, but the methods inside, which mainly carry out a single task, are long and contain many nested loops and conditional blocks;

28 All the variables and methods are declared as `public`, but especially for internal variables, it is better to restrict the access through at least `protected` or even `private` clause;

42 Error messages are not so clear nor help very much to solve the error. The errors should have been explained more clearly. In particular, see lines 58, 74, 90;

52 Only `IOException` is caught among the most relevant ones. Possible exceptions raised by the `Integer.parseInt()` are not caught, see line 175;

53 Message errors are simply printed to the error console and a return action to a specific entity is taken, but no effective resolution nor recovery actions are supposed;

The other clauses are correct, in fact:

1 Variables and methods in the class have meaningful names; it is noticeable that some methods of others class, used by `ScrumServices`, have *ambiguous* meanings (see section 3.2 on page 11);

2 All one-character variables are used in temporary contexts;

3 The `ScrumServices` class is correctly written;

4 No *interfaces* to analyze;

5 All the methods declared in the class comply to the name conventions;

6 All the variables comply to the name convention;

8 Always four spaces are used to indent the source code;

9 No *tabs* are used to indent the code;

10 The *Kernighan and Ritchie* style is adopted to delimit blocks by braces;

11 All the blocks, included the ones with only one statement, are surrounded by curly braces;

15 There are no wrapped lines, and it is bad due to the huge amount of very long lines (refer to points *12* and *13*);

16 Unfortunately there are not wrapped lines;

17 Again, no wrapped lines to check;

18 Comments are used sparingly, however in key points they are present;

20 The Java source file of the class contains rightly only the `ScrumServices` class;

21 The `ScrumServices` class is the first appearing in the file;

22 No external *interfaces* are implemented;

24 The `package` statement is the first appearing in the code after the `imports`;

25 The declarations order complies to the convention;

26 The class only contains four methods, thus the functional/scope/accessibility order is not so distinguishably different;

29 All the variables are declared in their proper scope;

30 Only constructors are used when a new object is needed;

31 All the objects are initialized before using them;

32 All the variables are initialized;

33 All the declarations appear at the beginning of blocks;

34 All the parameters are presented in the correct order;

35 No erroneous method is called due to misleading similar names;

36 Values returned by methods are used accordingly to the call;

37 No pure arrays are used in the code;

38 No problems regarding bounds in any of the collection;

39 Of course constructors are called when a new array item is needed;

40 There are no direct object comparisons;

41 Outputs are free of spelling and grammatical errors;

43 Outputs are correctly formatted;

44 No evidence of *brutish programming*;

45 Operator precedences, parenthesizing and computations are correct;

46 No ambiguous precedence expressions are found, so no parenthesis problem;

47 No arithmetical divisions in the code;

48 Integer arithmetic operations are used properly;

49 All the comparison and boolean operators are correct;

50 Error conditions are legitimate;

51 The code is free of implicit type conversions;

54 No switch in the code;

55 No switch in the code;

56 All loops are correctly formed;

57 No files are employed;

58 No files are opened, then no file are closed;

59 No files are used, then no EOFs conditions;

60 No files presence, then no file exceptions to catch.

3.2. Other issues

Some strings are used more than three times, thus it should be better to declare them as constant. Many arithmetic operators and punctuation are not surrounded by spaces on both sides.

Some methods exploited by our class coming from other classes (e.g. `get(..)`, `use(..)`) do not specify what they do clearly in their names; in fact, names are very general: the only helpful hints to understand the meaning of the method come from the `String` parameters passed. The issue is probably due to the design approach, which claims to manage all the data through generic classes and this implies you to use different strings to specify what field you want to employ in the method.

4. Hours of work

Document	**Marco [h]**	**Daniele [h]**	**Total [h]**
Requirements and Specifications Document	48	49.5	97.5
Design Document	35	40	75
Integration Test Plan Document	26	22	48
Project Plan Document	20	20	40
Inspection Document	10	7	17
Overall revision	–	–	–
Total	139	138.5	**277.5**

5. References

To draw up this document, we refer to the assignment about the Code Inspection Document provided in the lectures.

A. Checklist

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.
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized.
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.
7. Constants are declared using all uppercase with words separated by an underscore.

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.

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. External program interfaces are implemented consistently with what is described in the javadoc.
23. 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. Variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).
29. Variables are declared in the proper scope.
30. Constructors are called when a new object is desired.
31. 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 (block: any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. 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. Method returned values are used properly.

Arrays

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

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

39. Constructors are called when a new array item is desired.

Object Comparison

40. All objects (including `String`s) are compared with `equals` and not with `==`.

Output Format

41. Displayed output is free of spelling and grammatical errors.

42. Error messages are comprehensive and provide guidance as to how to correct the problem.

43. The output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Implementation avoids "brutish programming"[1].

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

46. Liberal use of parenthesis is used to avoid operator precedence problems.

47. All denominators of a division are prevented from being zero.

48. Integer arithmetic, especially division, is used appropriately to avoid causing unexpected truncation/rounding.

[1]See `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html` for more information.

49. Comparison and Boolean operators are correct.

50. Check `try-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 most relevant exceptions are caught.

53. 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. All switch statements have a default branch.

56. All loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. All files are properly declared and opened.

58. All files are closed properly, even in the case of an error.

59. EOF conditions are detected and handled correctly.

60. All file exceptions are caught and dealt with accordingly.