Code Inspection

Politecnico di Milano A.A. 2015-2016

Software Engineering 2 Raffaela Mirandola

Salvatore Perillo 853349 Claudio Sesto 841623

Summary

Table of content: Include the table of content of your document

Functional role of assigned set of classes: <elaborate on the functional role you have identified for the class cluster that was assigned to you, also, elaborate on how you managed to understand this role and provide the necessary evidence, e.g., javadoc, diagrams, etc.>

List of issues found by applying the checklist: <report the classes/code fragments that do not fulfill some points in the check list. Explain which point is not fulfilled and why>.

Any other problem you have highlighted: <list here all the parts of code that you think create or may create a bug and explain why>.

Introduction

This document contains the results concerning the code inspection about the assigned methods: these methods are all inside a single class named *EJBContainerProviderImpl* that implements the interface *EJBContainerProvider* "***state the namespace pattern and name of the classes that were assigned to you***".
In order to perform a complete check of the methods, we followed the given checklist: thanks to the already existing (although a bit incomplete) javadoc of the class and look through the source code, we were able to understand the role of it.
The methods that we need to inspect are:
At line 310:
getRequestedEJBModuleOrLibrary(File file , Map < String , Boolean > moduleNames)
At line 364:
addModule(Locations l , Set < DeploymentElement > modules ,
  Map < String , Boolean > moduleNames ,File f ,
    boolean skip_module_with_main_class)
At line 396:
skipJar(File file , Locations l , boolean skip_module_with_main_class)

We have regrouped these methods in one cluster, this means that all methods we are inspecting are correlated one to each other; indeed the main method of this group is *addModule* calls the others.
The goal of this cluster is to add a DeploymentElement (in our case is an EJB module or a library) to a set of modules.
In order to add the module (passed as a file) we have in the class a list of known packages: the method skipJar (over other controls) checks if the module contains only known packages; if there exist at least one attribute value that is accepted (the list of values is also inside the class) we don't skip the file and is added.

In case of need to skip modules that contains a main class attribute (this list is outside the class we are in charge to inspect) the flag "*skip_module_with_main_class*" is set to true, otherwise is false. If all conditions are respected the new element is created with the *getRequestedEJBModuleOrLibrary* method: this methods returns **null** if the file is an EJB module which name is not present in the list of requested.
After the creation the module is added to the set.
As evidence of our deductions, we report below some high level diagrams (if you don't put diagrasm remember to delete the evidences)

Inspection Results

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.

We have encountered this problem at lines **318, 326, 410, 439, 406**:
in the first 4 lines the problem is that the name of the variable is the acronym of the class the object represent and this can bring to misundertandings (is!=null); the other line instead contain a variable named "*m_file*": by a more accurate inspection we've deduced that '*m*' means "module" but at the beginning wasn't so clear.

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

We have encountered this problem at lines **379, 400, 442**:
all these lines contains a one-character variable (respectively File f, Location l, Manifest m) that is not temporary but are used also to create the object Module to add and check if is a proper file to add (m.getMainAttributes(); if (f.exists() && !skipJar(f, l, skip_module_with_main_class))).

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized.

In the piece of code assigned to us, there are some references to classes and those names are correctly named as the point requires.

4. Interface names should be capitalized like classes.

The class of our methods implements the interface EJBContainerProvider and the name is write correctly.

5. Method names should be verbs, with the first letter of each addition word capitalized.Examples: getModuleDescriptor();setArchiveUri(fileName).

All methods we have to inspect plus all methods that are called inside are correclty named as the above examples show.

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.

On line 400 we have the variable *skip_module_with_main_class*: Although the name is meaningful, the words are separated by an underscore: also the variable *m_file* at line 406 (spotted at point 1 of this section) has this characteristic. One correct example is the variable _logger used multiple time.

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

All constant used in our methods are correctly declared, however some constants at line **105** and **108** don't respect the standard constant declaration (localStrings and lock)

Indention

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

To check this point we add as a preference the view of the space characters in the editor we've used for the inspection, after this we count the space for identation and we've notice that four spaces are used consistently. For what concern the truncation we've seen that a standard of eight spaces is used except for the line **335** where there are four spaces.

9. No tabs are used to indent.

In the code we have to inspect, no tabs are used: to check this we have used the same method of point 8; in this way it was much faster than check line per line for tabs by moving cursor.

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). NO PROBLEM

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();
        }
```

NO PROBLEM

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. NIE MA PROBLEMU
21. The public class is the first class or interface in the file.NIE MA PROBLEMU
22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.NIE MA PROBLEMU
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. NIE MA PROBLEMU

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
a. first public class variables
b. next protected class variables
c. next package level (no access modifier)
d. last private class variables
E. instance variables
a. first public instance variables
e. next protected instance variables
f. next package level (no access modifier)
g. 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 NIE MA PROBLEMU
30. Check that constructors are called when a new object is desired
31. Check that all object references are initialized before use NIE MA PROBLEMU
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 NIE MA PROBLEMU
35. Check that the correct method is being called, or should it be a different method with a similar name NIE MA PROBLEMU
36. Check that method returned values are used properly NIE MA PROBLEMU

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) NIE MA PROBLEMU

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds NIE MA PROBLEMU
39. Check that constructors are called when a new array item is desired NIE MA PROBLEMU

Object Comparison

40. Check that all objects (including Strings) are compared with "equals" and not with "==" NIE MA PROBLEMU

Output Format

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

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem NIE MA PROLEMU
43. Check that the output is formatted correctly in terms of line stepping and spacing NIE MA PROBLEMU

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
FIN QUI NIE MA PROBLEMU
50. Check throw-catch expressions, and check that the error condition is actually legitimate
NIE MA PROBLEMU
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 NIE MA PROBLEMU
55. Check that all switch statements have a default branch NIE MA PROBLEMU
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions NIE MA PROBLEMU

Files

57. Check that all files are properly declared and opened NIE MA PROBLEMU
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 NIE MA PROBLEMU
60. Check that all file exceptions are caught and dealt with accordingly SEE POINT 53