Code Inspection

Politecnico di Milano **A.A. 2015-2016**

Software Engineering 2 Raffaela Mirandola

Salvatore Perillo 853349 Claudio Sesto 841623

Summary

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* and belongs to the package *org.glassfish.ejb.embedded*.
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.

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); actually is a “de facto” standard so is correct. 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. By searching in the oracle website (http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html) we have seen that this kind of truncation is allowed.

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).

All blocks in the methods we have to inspect, the Kernighan and Ritchie style is used consistently.

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();
      }
```

The methods we have to check, contains lot if, for, try-catch blocks with a single statemante to execute; all of them have curly braces so this point is fulfilled

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).

We have found some incongruity at lines **338, 346, 354, 380, 431, 438, 479**: all these blank lines separate pieces of code within the same method, mainly for “if” blocks but others (like line 380) are apparently blank without any meaning.

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

We have found some line that exceed the 80 characters but only if we count also the spaces for the identation. The lines are the following: **310,329,340,342,343,365,367**. The main motivation of this lenght threshold overcome is due declaration of methods with long (or many) variables, long variable names in “if” condition or logger message.

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

With the same considerations of the previous point the only lines that exceed the 120 characters are **415,446,463** and are messages for the logger**.**

Wrapping Lines

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

Lines **310, 342, 425,474** have a line break that occour after a closing parenthesis or before the string concatenation operator (not after).

16. Higher-level breaks are used.

A high-level break is a break that cut off part of expression without truncate the subexpressions; for example if we need to truncate x+(y+z) is preferable to have:

x+ (y+z)	than	x+(y + z)

Knowing this, the line breaks spotted in the previous point, that are the critical ones, have no problem at all (if we consider passing parameters in methods as one expression, the truncation at line 376 is correct due no subexpressions).

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

In all methods we have to check, we haven't found any problem

Comments

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

Some methods contains comments inside that explain at high level what we have in certain points to simplify the reading of the code and/or explain why some values are returned. Example from lines **413**, **414**:

```
if(containsMainClass(new Manifest(is))) {
        // Ignore dirs with a Manifest file with a Main-Class attribute
```

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.

No commented out code is present in the methods we are controlling.

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.

Points from 20 to 22 are correct.
We have only one public class EJBContainerProviderImpl that appear inside the file and is first declared: inside the file we have found a private class Locations written at line **514,** so after the public class.
No external program interfaces have been found.

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

The methods assigned to us have a simple javadoc that says what the method return, but not the meaning of the parameters that in some methods is not so immediate to understand: so actually the javadoc is incomplete.

Package and Import Statements

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

In our class the package statement at line **41** is the first non-comment line and is followed by all the import operations.

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

The only problem found here is at line **85** that contains a protected constant and then private ones, so the order is not respected: the other points are correct.

26. Methods are grouped by functionality rather than by scope or accessibility.

The methods that are assigned to us are grouped by functionality (addModule that calls the other two): by inspecting the other methods of the class, we have found different “get” methods declared in different points; usually all setter and getter methods are grouped at the end of the class.

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

We don't have duplication of code and the methods aren't too long (maybe skipJar can be considered a few long since it has about 80 lines of code). The class EjbContainerProviderIMpl can be considered quite long (about 600 lines) but is not the same for the private class Locations (80 lines more or less). All the important methods are private so we don't break encapsulation.
For what concerning the cohesion: since the methods of the class deal with different kind of actions and objects, we can consider a high level of cohesion. Finally, since our class is a provider of a specific object, there is also a high level of coupling so other classes may depend a lot from this one.

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)

The line **85** already spotted at point 25 contains a protected visibility constant: actually we don't know if is correct or not but in any case it should be declared after all the private constant.

29. Check that variables are declared in the proper scope

All declarations are stated in the correct scope: so declared and used inside a "for" loop or inside a method.

30. Check that constructors are called when a new object is desired

No problem here: even the variable "result" declared at line **312** as null, later at line **348** and **351** is initialized by calling the correct constructor.

31. Check that all object references are initialized before use

All object reference are initialized before the use.So we don't have any variable that is used without have a reference to a object.

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

At lines **321, 317, 318, 410, 439** we have found variables declared and initialized some lines after: the problem is that no computation is performed in order to change the initialization are just declared outside a try and initialized inside it.
eg.:

```
318 InputStream is = null;
319  try {
...
323         is = getDeploymentDescriptor(archive);
```

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.

Technically speaking, we have found lines **317, 319, 406, 410, 439, 450, 451** that declare variables after the beginning of the block but they are after a "if" block with inside a *return* or a logger operation: so if from one side is not the real "beginning of the block", from the other side we avoid potential useless declarations or perform a logger action immediatly.

Method Calls

34. Check that parameters are presented in the correct order

For this point we have checked that when our methods are called, the list of parameters is consistent with the header of the respective method.
eg:

```
400:
private boolean skipJar(File file, Locations l, boolean
skip_module_with_main_class) throws Exception {
379:
if (f.exists() && !skipJar(f, l, skip_module_with_main_class)) {
```

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

For what concerning our methods we are sure that when one of it is called, it refers to the correct method. For the other methods we don't have to check, we assume they are called properly.

36. Check that method returned values are used properly

Since our main method addModule calls the other two and the returned values are used in correct way, for this point we can say that all is correct:

```
379 if (f.exists() && !skipJar(f, l, skip_module_with_main_class)) {
381 DeploymentElement de = getRequestedEJBModuleOrLibrary(f, moduleNames);
```

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

We have a set of *DeploymentElement* and a constant array of string KNOWN_PACKAGES.
Due the fact our arrays are always passed through parameters or are a constant declared inside the class, no constructors are needed. When we have to perform a control of all values that belongs to the array, we access it by using this form:

```
line 453 for (String skipValue : KNOWN_PACKAGES) {
```

So all kind of controls (such as out of bounds) are automatically checked and also there is no access by using the index.

Object Comparison

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

In our piece of code the only comparison that requires the equals method is:

434 *l.modules_dir.equals(file.getAbsoluteFile().getParentFile().getAbsolutePath()).*

All the other comparisions are simply controls if the object is correctly instanciated or not; this is performed by checking if the variable is not null: some examples:

443 ***if*** *(m != null)*

452 ***if*** *(value != null)*

Output Format

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

All the output messages are free from spelling and grammatical errors.

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

In our methods, the error messages are simply the report of a thrown exception so actually there is no different guidance provided that says how to correct problems than the normal one.

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

No problem is found here.

Computation, Comparisons and Assignments

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

According to the guide provided by the URL above, our code respect all rules of good programming avoiding the brutish one. For example we don’t have “if” constructs in the form: *if(boolean variable ==true){*

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

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

In our methods, all the operator sequences are written in the way that all the precendence constraints are already fullfilled without any use of parenthesis.
eg.:
347 ***if*** *(!isEJBModule || moduleNames.isEmpty())*

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

We don’t have any arithmetic operation in our code ,so these points are correct.

49. Check that the comparison and Boolean operators are correct

All Boolean operators are used correctly according also to the motivations at point **44.**

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

All “throw-catch” conditions are correct, meaning that if a method (or a constructor) returns the error value, is properly checked and the exception is handle.

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

At line **410** we have spotted something unclear: (metto codice) the variable is immediatly instanciate as a *FileInputStream* object while the type in the declaration is *InputStream*. Due polymorphism this is correct but to avoid type conversion it’s preferable to declare the variable as *FileInputStream*.
The other methods are free from any implicit type conversions: whenever is needed, casting is correctly used.
329: EjbBundleDescriptor bundleDesc = (EjbBundleDescriptor) eddf.read(is);

Exceptions

52. Check that the relevant exceptions are caught

At line **390** we have identified a general exception thrown by the method *skipJar*; actually this exception, although of highest level, is handle into the method as a IOException and so correctly caught and handled.
At line **356** instead we have a “try” without “catch” but only with a “finally”: so the exception (if not managed in another class we don’t have to check) is not handled; actually this is allowed so there is no problem at all.
Also, if the methods called inside the “finally” throws exceptions, a “try-catch” block is missing.

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

All catch used calls a method of the class *Logger*, so assuming it is correct, the appropriate action is taken.

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

In our methods there is no “switch” statement so the previous two point are “fulfilled”.

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

For the reason already written at the points **37,38,39** we can say that all loop are correctly formed.

Files

57. Check that all files are properly declared and opened

The major part of the *File* variables used in put methods are passes as arguments, so we assume that they have been opened before our methods are called; also, the constructor of the class *File*, automatically open a file when initializated.
Example of already open file:
322 *archive = archiveFactory.openArchive(file);*

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

At line **406** a new file is created, and so open; the problem is that this file is not closed in any of the following lines of code of the method
406 *File m_file = new File(file, "META-INF/MANIFEST.MF");*

59. Check that EOF conditions are detected and handled correctly

In the three methods that we have to check, the access to the files is performed by simple methods of the class File, so there is no risk to encounter the EOF.

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

Due the only exceptions are thrown by methods of the class *File*, all file exceptions are taken and handled (see also points **52** and **53**).

Hours of Works

Perillo Salvatore---20
Sesto Claudio------20