Code Inspection Document
“MyTaxiService” Application
A.Y. 2015 - 2016

Ennio Visconti (mat. 790382)
Simone Zocchi (mat. 852910)
Khanh Huy Paolo Tran (mat. 852496)

January 5, 2016

POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Assignment

Class assigned:

appserver/resources/resources-connector/src/main/java/org/glassfish/
resources/admin/cli/**ResourcesXMLParser.java**

Methods:

- generateWorkSecurityMap(Node node , String scope)
- generateSecurityMap(String poolName , Node mapNode , String scope)
- generateResourceAdapterConfig(Node nextKid , String scope)
- resolveEntity(String publicId , String systemId)

1.2 Document Structure

The document divides into three parts:

- Section 1: **Introduction**. List the class and the methods analyzed in this assignment and the references used for the redaction of this document.
- Section 2: **Functional Description**. Contains the descriptions of the behaviour and the functionality implemented in the class and the methods analyzed.
- Section 3: **Issues Analysis**. Contains the problems found in the code according to the checklist given.

1.3 References

- Assignment document: Software Engineering 2 Project, AA 2015-2016, Assignment 3: Code Inspection
- Oracle GlassFish Server 3.1 Application Deployment Guide

2 Functional Description

The following information were taken from the JavaDoc of the interface EntityResolver in the package org.xml.sax, from ”Oracle GlassFish Server 3.1 ”Application Deployment Guide” and from a step by step reading of the code.

2.1 Context Overview

This procedure is used during the deployment of a web application. It extracts the configurations of application-scoped resources from an xml document named *glassfish-resources.xml*, which has the following structure (only relevant elements has been considered):

```
resources
[. . .]
.   resource-adapter-config
.   .   property (with attributes)
.   .   .   Description
.   connector-connection-pool
.   .   Description
.   .   security-map
.   .   .   Principal
.   .   .   user-group
.   .   .   backend-principal
.   .   property (with attributes)
.   .   .   Description
.   work-security-map
.   .   Description
.   .   principal-map
.   .   group-map
```

Application-scoped resources are all the modules and other applications which are required at deployment time, they are created during the deployment phase and destroyed at its completion, they are available only to the module or application which defines it.

2.2 Class Overview

The class we have inspected is an implementation of the interface *EntityResolver* provided by the SAX java package.
It should be used by an XML parser to intercept any external entities before including them. The *resolveEntity* method is the implementation of this aspect.

When the constructor of this class is invoked with a source file as a parameter, which must be an XML document, the file is parsed and all the nodes are copied into a collection of objects called *vResource*, with all the attributes that the nodes have, and in a Map with all the their child nodes' references. The three generate* methods are an implementation of what just said, for three specific type of nodes. For further information see Figure 1.

2.3 Method: resolveEntity

In the resolveEntity method, firstly, the DTD version is identified, comparing the publicId and systemId parameters with a preset list of names, and then if the DTD file is present, it is loaded as an InputStream otherwise an Exception is thrown.

2.4 Method: generateWorkSecurityMap

The method generateWorkSecurityMap is called by generateResourceObjects if the name of the node in exam is CONNECTOR_WORK_SECURITY_MAP. It creates a Resource object and sets two attributes of the object with the value of the relevant attributes of the node. Then for all the children nodes that are
WORK_SECURITY_MAP_GROUP_MAP or
WORK_SECURITY_MAP_PRINCIPAL_MAP it add other attributes to the resource with the value of the relevant nodes.
Finally the resource is added to the collection of resources and to the map.

2.5 Method: generateSecurityMap

The method generateSecurityMap is called by generateConnectorConnectionPoolResource if the name of the node in exam is SECURITY_MAP. A resource for the node is created and few basic attributes of the resource are added. Then the children nodes are extracted and if their names are SECURITY_MAP_PRINCIPAL or SECURITY_MAP_USER_GROUP the first child is extracted and the value inserted in a String array that are then inserted as attributes of the Resource. If the child node is, instead, a SECURITY_MAP_BACKEND_PRINCIPAL, the values of its attributes USER_NAME and PASSWORD are directly added as attributes of the resource. The resource is finally added to the collection of resources and to the map.

2.6 Method: generateResourceAdapterConfig

The method generateResourceAdapterConfig is called by generateResourceObjects if the name of the node in exam is RESOURCE_ADAPTER_CONFIG. As before, a Resource element is created and the attributes of the nodes are added to the resource. The resource is then added to the the collection of resources and to the map.

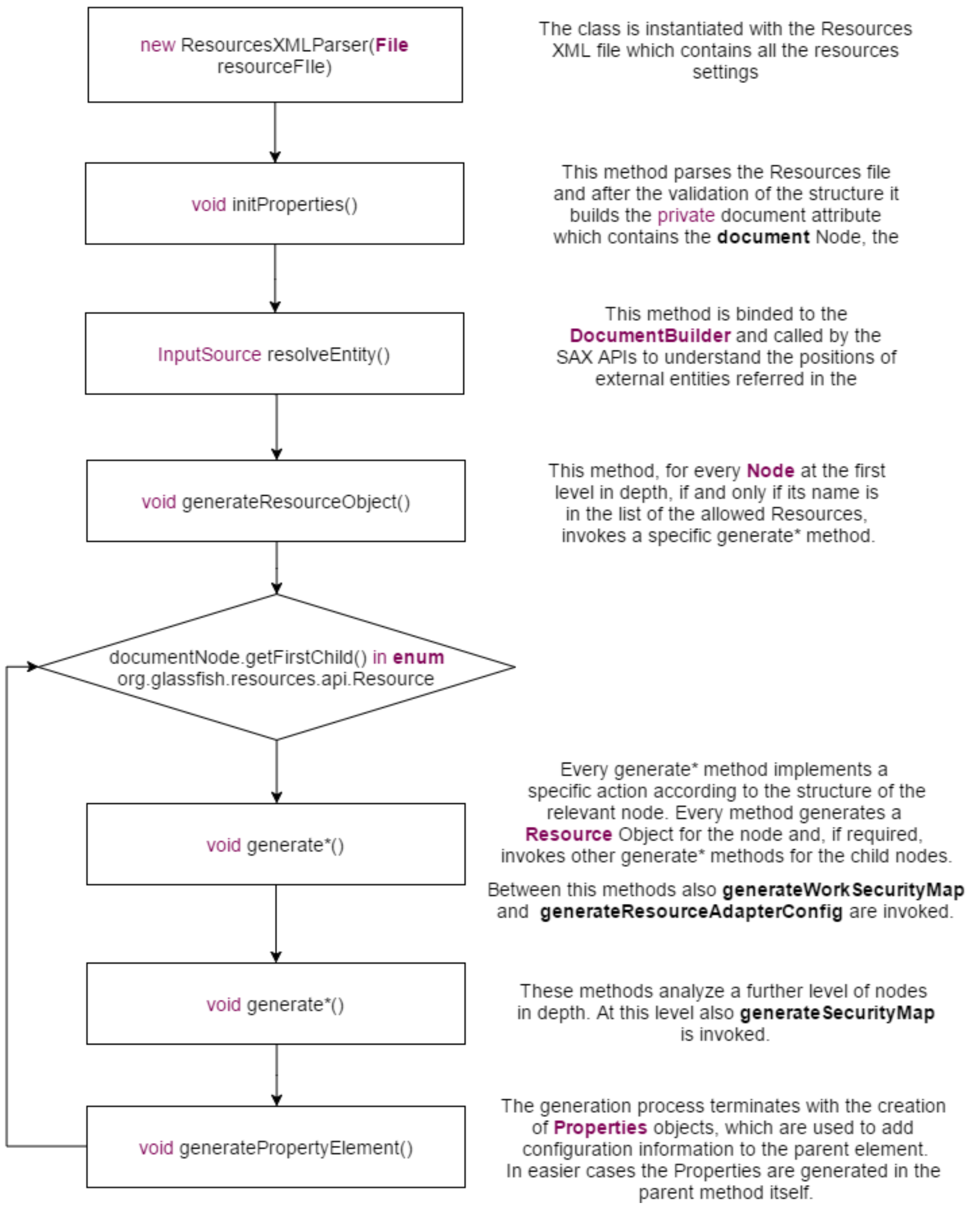
new ResourcesXMLParser(File resourceFIle)
void initProperties()
InputSource resolveEntity()
void generateResourceObject()
documentNode.getFirstChild() in enum org.glassfish.resources.api.Resource
void generate*()
void generate*()
void generatePropertyElement()
The class is instantiated with the Resources XML file which contains all the resources settings
This method parses the Resources file and after the validation of the structure it builds the private document attribute which contains the document Node, the
This method is binded to the DocumentBuilder and called by the SAX APIs to understand the positions of external entities referred in the
This method, for every Node at the first level in depth, if and only if its name is in the list of the allowed Resources, invokes a specific generate* method.
Every generate* method implements a specific action according to the structure of the relevant node. Every method generates a Resource Object for the node and, if required, invokes other generate* methods for the child nodes.
Between this methods also generateWorkSecurityMap and generateResourceAdapterConfig are invoked.
These methods analyze a further level of nodes in depth. At this level also generateSecurityMap is invoked.
The generation process terminates with the creation of Properties objects, which are used to add configuration information to the parent element.
In easier cases the Properties are generated in the parent method itself.

Figure 1: High level flow chart of class execution

3 Issues Analysis

3.1 Issue Categories

3.1.1 Naming Conventions

1-7. Naming convention has been correctly taken into account. In fact constants are always in upper case and the classic camel case standard is used (upper camel case for class and interface names and lower camel case for variable names and parameters) with the only exception of a variable at *line 1403*. Furthermore all the constants, variables, classes and interfaces used or defined have a name that clearly explains their meaning.

3.1.2 Indentation

8. Indentations are done with 4 spaces.

9. No tabs are used (besides *line 234*, which isn't in our assignment).

3.1.3 Braces

10-11. "Kernighan and Ritchie" style is used but is not really consistent, like on method definitions at *lines 1448, 1533*. Furthermore in a few cases control blocks have no curly braces because they are executing a single statements, like at *lines 1389, 1400, 1453*.

3.1.4 File Organization

12. Use of blank lines and comments is not very frequent in the class analyzed. When used, they properly separate sections in the class.

13. Practical lines in a few cases exceed the maximum length of 80 characters, for example at *lines 1328, 1336, 1349, 1363, 1364, 1372, 1383, 1394, 1455, 1626*

14. In a few cases, for instance in method declaration, they also exceed the 120 char maximum length. *Lines 964, 1191, 1524*

3.1.5 Wrapping Lines

15. Line breaks are always used in the correct way, after '+' *at lines 1598-1600*, or ',' *at lines 1227, 1232, 1506, 1524...*

16. When too long, parenthesized expressions are wrapped accordingly to the parenthesis hierarchy.

17. Not always new statements are aligned with the beginning of the expression at the same level as the previous lines, like at *lines 1260, 1436, 1437, 1439.*

3.1.6 Comments

18. There are no comments explaining what the code is doing. *Lines 1319, 1385, 1449.*

19. Outside our code block there is commented out code. It doesn't contain neither reasons for being eliminated nor date for the definitive elimination. *Lines 243, 1107.*

3.1.7 Java Source Files

20-21. The only file of our block contains just three classes and no interfaces of which the only public class is the first.

22. The method resolveEntity is consistent with the javadoc, besides it doesn't throw *IOException*.

23. The javadoc doesn't cover *generateWorkSecurityMap*, *generateSecurityMap* and *generateResourceAdapterConfig* probably because they are private methods.

3.1.8 Package and Import Statements

24. In the given class the package statement is correctly placed at the beginning of the document, only after the introductory comment. It is correctly followed by all the import statements.

3.1.9 Class and Interface Declarations

25. A first comment with the explanation of the class is present, followed by the package location and the import statements. The documentation comment is present but is very short and not complete.
    As the class is an implementation of EntityResolver interface, the implementation comment should be provided but it is not present.
    Class (static) variables and instance variables are grouped but placed in the wrong order, firstly are placed the instance variables and then the class ones.
    After that, methods are placed but, at the end of the document, they are mixed with the declarations of two inner classes.

26. The methods are not properly grouped by accessibility, for instance the most of the public methods are placed at the beginning of the class and a few of them are at the end. They seems to be grouped by functionality, in fact there is a set of methods for generate resources, followed by some getter methods.

27. Methods *generateWorkSecurityMap*, *generateSecurityMap* and *resolveEntity* are a little bit too long.
    Encapsulation is preserved.

generateWorkSecurityMap has an *else if* which does basically the same thing as the precondition, but with a different node, so there are duplicated code lines. *Lines 1344-1374*
In addition inside *resolveEntity* there is a checking command repeated many times (*if(_logger.isLoggable(Level.FINEST))*). *Lines 1601, 1606, 1611, 1619, 1625.*

3.1.10 Initialization and Declarations

28. All class members (of our code block) have the correct visibility (they are all private except the one defined in the interface implemented). All variables and class members are of the type required by their usage and so they can be considered as correct.

29. Variable scoping is done really well. Every variable is declared exactly when it is needed and never before.

30. Every new object is instantiated through the constructor.

31-32. All the variables are initialized immediately after the declaration. Furthermore the methods are always invoked with correct parameters and therefore parameters references are correctly initialized.

33. Some variables aren't declared at the beginning of a block. *Lines 1339, 1405, 1472, 1609, 1616-1617, 1622*

3.1.11 Method Calls

34. All parameters are presented in the correct order.

35. All the correct methods are called.

36. Methods invoked in *generateWorkSecurityMap*, *generateSecurityMap* and *generateResourceAdapterConfig*, whose returned value is then used, are only methods that extract Nodes, List of nodes or the value of a node as String. The values are then copied in local resources and maps, matching node types with their value.
In *resolveEntity* are used methods to get items from a list, compare strings and create InputStream from URL.
The values are always used in a proper way.

3.1.12 Arrays

37. The collections used are always entirely scanned and the items of the collections are always accessed through the *get(i)* or *item(i)* methods. *Lines 1342, 1409, 1420, 1579.*

38. The collections used are always scanned in for statements that use the exit condition $i <$ *array.getLength()* (or *array.size()*) with i=0 and incremented by one at every iteration. So no index goes out of bound. *1341, 1408, 1578.*

39. No array is created and no elements are added to Collections used.

3.1.13 Object Comparison

40. "==" is always and only used to check whether pointers to object are null or different from null *at lines 1322, 1329, 1335, 1340, 1388*, only in one case it is used to compare an integer *at line 418.*

3.1.14 Output Format

41. Displayed output is free of spelling or grammatical mistakes.

42. *generateWorkSecurityMap*, *generateSecurityMap* and *generateResourceAdapterConfig* just throws exceptions to the upper level so errors aren't handled there. *resovleEntity* has comprehensive error messages, but not always provides guidance as to how to solve the problem. *1631.*

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

3.1.15 Computation, Comparisons and Assignments

44. "The term *brute force* refers to solutions where someone applies an obvious but excessive technique when more refined and effective alternatives are available. ".
Given this definition, the analyzed code has generated many concerns.
From a very basic analysis the code looks not so bad, because it strongly relies on framework methods and therefore avoids to redefine methods to handle already solved problems.
However some design problmes will be discussed later.

45. Computations order is correct, so is parenthesizing.

46. There are really few calculations. So there are no operator precedence issues, neither divisions, nor integer arithmetic in general is used.

49. Boolean operators are always used in either *if* or *for* conditions. In most cases comparison operator are used, such as == or != to check if an extracted node is whether null or not. *Lines 1322,1329,1335,1340,1347,1350,1353.*
Other comparison operators, such as <, are used in *for* statements as termination conditions to scan the whole collection in object. *Lines 1342,1408,1578.*
Other Boolean operators, such as &&, are used in *lines 1350,1353,1365,1368,1587.*
All the operators are used in the correct way and with a correct behaviour.

50. At *line 1630* in the catch field a generic SAXException is thrown and the real exception is hidden. This is not good practice.

51. At *lines 1354,1369* String object are automatically converted to Object and passed as parameters to the put method.

3.1.16 Exceptions

52-53. Not only exceptions are never handled, as it can be seen by the fact that the general unchecked *Exception* is thrown by method definitions at *lines 1317, 1383, 1447.* Furthermore the only try/catch block in the code, handles the exception in the really worst way possible: it hides the exception and generates a new one which is the only allowed by the interface specification.

3.1.17 Flow of Control

54-56. Not applicable. There is no switch statement in our code block. All loops are correctly formed with appropriate initializations, increments and terminations.

3.1.18 Files

57-59. In the methods analyzed only one file is used, at *line 1604*. It is then opened in an InputSource but never closed because the InputSource element is returned as the result of the method. The file is not scanned so the EOF condition is not used.

60. At *line 1630* a generic exception is caught for every exception in reading a file. It is re-thrown by the method as a SAXException.

3.2 Other problems

3.2.1 Design

An in-depth look at the code has shown a different problem: in many cases the code goes up to 5 levels of if clauses as in lines *1340-1355* and in general in our class there are 169 if clauses out of 1662 lines of code (counted with comments and empty lines). That's about 10% of the total lines, and gives a clear idea of the problem: the project seems to have quite a good high level architecture, but quite a bad code-level design. Furthermore, the only comments are used to enforce a really bad practice: they tell the reader that a parameter should not be used (symptom of a really bad design).

3.2.2 Implementation

TODO comments are still present *486, 1615* in the code and also a FIX ME *1100.*