Politecnico di Milano

Software Engineering II

---

myTaxiService
Code Inspection

Authors
Giovanni Beri 852984
Biagio Festa 841988

Monday 4th January, 2016

Contents

https://github.com/BiagioFesta/SE2-PoliMi-myTaxiService

Professor: Di Nitto
Students: Biagio Festa, Giovanni Beri

Methods:

- **Name:**addImportsAndIncludes(NodeList list , Collection result , String namespace , String location)
  Start Line:535
  Location:appserver/webservices/jsr109-impl/src/main/java/org/glassfish/webservices/WebServicesDeployer.java
- **Name:**doWebServicesDeployment(Application app , DeploymentContext dc)
  Start Line:574
  Location:appserver/webservices/jsr109-impl/src/main/java/org/glassfish/webservices/WebServicesDeployer.java
- **Name:**doWebServiceDeployment(WebBundleDescriptor webBunDesc)
  Start Line:686
  Location:appserver/webservices/jsr109-impl/src/main/java/org/glassfish/webservices/WebServicesDeployer.java

1 WebServicesDeployer

1.1 Functional Role

The class is used for the deployment of a web application. There a principal public method called "prepare" which invokes all the other ones.

1.1.1 `addImportsAndIncludes`

As the code is quite short, we can display it below:

```
1     private void addImportsAndIncludes(NodeList list, Collection result,
2                         String namespace, String location) throws
                              SAXException,
3                         ParserConfigurationException, IOException {
4         for(int i=0; i<list.getLength(); i++) {
5             String givenLocation = null;
6             Node element = list.item(i);
7             NamedNodeMap attrs = element.getAttributes();
8             Node n= attrs.getNamedItem(location);
9             if(n != null) {
10                givenLocation = n.getNodeValue();
11            }
12            if(givenLocation == null || givenLocation.startsWith("http")) {
13                continue;
14            }
15            Import imp = new Import();
16            imp.setLocation(givenLocation);
17            if(namespace != null) {
18                n = attrs.getNamedItem(namespace);
19                if(n != null) {
20                    imp.setNamespace(n.getNodeValue());
21                }
22            }
23            result.add(imp);
24        }
25
26    }
```

Figure 1: Snippet from lines 535 to 560.

The following class method simply adds a set of *import object* to the collection. That method is private so the *javadoc* documentation is missing. Through an analysis of the code and the class with an *reverse engineering* approach, we can deduct that:

- `NodeList list` is an input parameter. It represents a list of nodes in which each node is an element of *xml-like* document. We can deduct

that because of the method `parseRelativeImports` which constructs the document object and then it calls other four methods in order to process that document (*see code in figure 2*).

```
1              DocumentBuilder builder = factory.newDocumentBuilder();
2              is = wsdlFileUrl.openStream();
3              Document document = builder.parse(is);
4              procesSchemaImports(document, schemaRelativeImports);
5              procesWsdlImports(document, wsdlRelativeImports);
6              procesSchemaIncludes(document, schemaIncludes);
7              procesWsdlIncludes(document, wsdlIncludes);
```

Figure 2: Snippet from lines 457 to 481.

The methods dedicated to the document processing are very similar among them, let's see at least one (*figure 3*):

```
1      private void procesSchemaImports(Document document, Collection
           schemaImportCollection) throws SAXException,
2              ParserConfigurationException, IOException {
3          NodeList schemaImports =
4                  document.getElementsByTagNameNS("http://www.w3.org/2001/XMLSchema",
                       "import");
5          addImportsAndIncludes(schemaImports, schemaImportCollection,
               "namespace", "schemaLocation");
6      }
```

Figure 3: Snippet from lines 507 to 512.

As we can see the *NodeList* is extract from the document through a *tag*: "*import*". After that the *our* method is called.

- From each node its attributes are extracted and analyzed. Whether the node satisfies some conditions, then a new *import object* will be created and filled with appropriate attribute's information data previously extracted. The object is added to the collection and so on for every node of the list.

1.1.2 `doWebServicesDeployment`

The method is dedicated to the preparation of the deployment of the web application services. In particular that method is needed in order to generate the *WSDL* file of the application.

The *Web Services Description Language* file (called WSDL) is an XML-based interface definition language that is used for describing the functionality offered by a web service.

First of all the procedure retrieves the services which have to be deployed by the context of the application. To do so, it prepares a collection of *Web Service* objects.

```
Collection<WebService> webServices = new HashSet<WebService>();
```

Figure 4: Snippet from lines 577 to 577.

```
1          WebServicesDescriptor wsDesc =
               dc.getModuleMetaData(WebServicesDescriptor.class);
2          if (wsDesc != null && wsDesc.getWebServices().size() > 0) {
3              if (logger.isLoggable(Level.FINE)) {
4                  logger.log(Level.FINE, LogUtils.WS_LOCAL,
5                          new Object[] {wsDesc.getWebServices().size(),
                               getWebServiceDescriptors(app).size()});
6              }
7              webServices.addAll(wsDesc.getWebServices());
8          }
```

Figure 5: Snippet from lines 582 to 589.

In order to fill that collection of services, it gets the web services associated with *module descriptor*, we can see that in the snippet 5.

Then it gets the web services associated with *extension descriptors*, as we can see in the snippet 6.

```
1          WebBundleDescriptor webBundleDesc =
               dc.getModuleMetaData(WebBundleDescriptor.class);
2          if (webBundleDesc != null) {
3              Collection<EjbBundleDescriptor> ejbBundleDescriptors =
                   webBundleDesc.getExtensionsDescriptors(EjbBundleDescriptor.class);
4              for (EjbBundleDescriptor ejbBundleDescriptor :
                   ejbBundleDescriptors) {
5                  Collection wsInExtnDesc =
                       ejbBundleDescriptor.getWebServices().getWebServices();
6                  webServices.addAll(wsInExtnDesc);
```

Figure 6: Snippet from lines 591 to 596.

The role of the *deployment descriptor* is to capture the declarative information (i.e information that is not included directly in the enterprise beans' code) that is intended for the deployment unit [1]. We can read more in the figure 7.

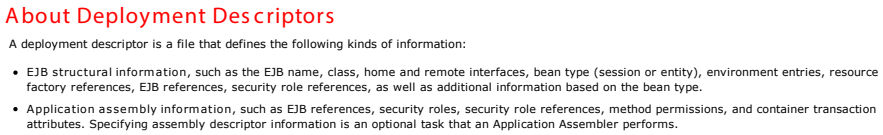

Figure 7: About Deployment Descriptor.
https://docs.oracle.com/cd/E13211_01/wle/dd/ddref.htm

Finally the output file will be created in the application directory, that because, as the comments suggest us: "*Even if deployer specified a wsdl file publish location, server can't assume it can access that file system*".

1.1.3 `doWebServiceDeployment`

This procedure is an overload of the method we've discussed previously. Actually we can see this method as a completion of the previous one as it is invoked in the end of that (*see figure 9*).

Essentially this procedure swaps the servlets name and implementation for ones which can be processed by a SOAP request.

Finally it gets a URL for the root of the webserver where the host portion is a canonical host name (*see snippet 10*).

```
1                   File genXmlDir =  dc.getScratchDir("xml");
2
3                   String wsdlFileDir = next.getWsdlFileUri().substring(0,
                        next.getWsdlFileUri().lastIndexOf('/'));
4                   mkDirs(new File(genXmlDir, wsdlFileDir));
5                   File genWsdlFile = new File(genXmlDir,
                        next.getWsdlFileUri());
6                   wsUtil.generateFinalWsdl(url, next,
                        wsUtil.getWebServerInfoForDAS(), genWsdlFile);
```

Figure 8: Snippet from lines 664 to 669.

```
1           if (webBundleDesc != null) {
2               doWebServiceDeployment(webBundleDesc);
3           }
```

Figure 9: Snippet from lines 673 to 675.

```
1               WebServerInfo wsi = new WsUtil().getWebServerInfoForDAS();
2               URL rootURL = wsi.getWebServerRootURL(nextEndpoint.isSecure());
3               String contextRoot = webBunDesc.getContextRoot();
4               URL actualAddress = nextEndpoint.composeEndpointAddress(rootURL,
                    contextRoot);
5               if (wsi.getHttpVS() != null && wsi.getHttpVS().getPort()!=0) {
6                  logger.log(Level.INFO, LogUtils.ENDPOINT_REGISTRATION,
7                           new Object[] {nextEndpoint.getEndpointName(),
                                actualAddress});
8               }
```

Figure 10: Snippet from lines 752 to 759.

1.2 Application of the checklist

In this paragraph we are going to report the result of the application of the points of the checklist that must be applied to the whole *WebServicesDeployer* class, not only to assigned methods.

2. At the line 542, the variable of type *Node* is used with one single letter. But that variable is not temporary even if it is used as store operation in the loop.

```
Node n= attrs.getNamedItem(location);
```

15. At the line 694 there is a superfluous line break, on the other hand the line break falls after the assignment operator.

```
Collection<WebServiceEndpoint> endpoints =
    webBunDesc.getWebServices().getEndpoints();
```

17. At the line 731 there is a bad indention: the assignment is on the same level.

```
SingleThreadModel.class.isAssignableFrom(servletImplClazz)
     ?
"org.glassfish.webservices.SingleThreadJAXRPCServlet"
     :
    "org.glassfish.webservices.JAXRPCServlet";
```

18. Although the class is public in the package its preliminar documentation is quite poor. There is no explanation about what the class is supposed to do or how it should work.

```
/**
 * Webservices module deployer. This is loaded from
     WebservicesContainer
 *
 * @author Bhakti Mehta
 * @author Rama Pulavarthi
 */
```

23. Java doc is not complete here. There is a public method of which is not explained what the return object is supposed to do. An input parameter is missing.

```
/**
 * Loads the meta date associated with the application.
 *
 * @parameters type type of metadata that this deployer has
     declared providing.
 */
@Override
public Object loadMetaData(Class type, DeploymentContext dc) {
```

23. Java doc is missing on this public class method.

```
public URL resolveCatalog(File catalogFile, String wsdlFile,
     WebService ws) throws DeploymentException {
```

23. Although the meaning is quite easy to understand, Java doc is missing on this public class methods:

```
public void downloadFile(URL httpUrl, File toFile) throws
    Exception {

@Override
public void unload(WebServicesApplication container,
    DeploymentContext context) {

public void clean(DeploymentContext dc) {

public WebServicesApplication load(WebServicesContainer container,
    DeploymentContext context) {
```

33. At the line 549, the import object is initialized at the end of the scope. Despite of that, it is fine because of performance reason. Indeed the loop could be enter in a conditional branch and skip that cycle. In that case it would be a waste of memory and computation initializing an object that will be not used.

```
if(givenLocation == null ||
    givenLocation.startsWith("http")) {
    continue;
}
Import imp = new Import();
```

2 JDOMetaDataProperties

2.1 Functional Role

The JDOMetaDataProperties class has a quite detailed JavaDoc documentation, so we have been able to get a first, high-level idea of the functional role of JDOMetaDataProperties class only by reading it. From the JavaDoc of the class, we discovered that this class is a parser of properties containing metadata information about classes and fields. In particular, the stored information about a class can be:

- Persistence Specification:
  - persistent
  - transactional
- Name of the superclass
- Access level:
  - public
  - private
  - protected
  - package

The stored information about the field of a class can be:

- Persistence Specification:
  - persistent
  - transactional
  - transient
- Class type
- Access level:
  - public
  - private
  - protected
  - package
- JDO annotation:
  - primary key
  - default fetch group
  - mediated

The presence of *JDO annotation* of a field implies that the field is going to be processed for persistence. The annotation provides information about how the annotated fields are processed for persistence (e.g. the annotated field is part of the *Default Fetch Group*).

The *JavaDoc* documentation provides also some information about the syntax used to store metadata, but this aspect is irrelevant for our purpose.

After the reading of the class *JavaDoc*, we started analyzing the public methods in order to understand which functions are provided by JDOMetaDataProperties class. We are now not considering the public constructor *JDOMetaDataProperties (Properties props)* because it doesn't actually offer some functionalities.

JDOClass getJDOClass (String classname) Get the information about the class with the name passed as a parameter

JDOField getJDOField (String fieldname, String classname) Get the information about the field with the name passed as a parameter

String[] getKnownClassNames () Get all the metadata about classes and fields that have already been retrieved

The public methods documentation and implementation are consistent with the description of the class in the JavaDoc. Analyzing the private methods, we understood that the mode of operation of the class is roughly the following:

- every time a metadata is requested through the *getJDOClass getJDOField* method, the class checks if the requested class has already been stored in the HashMap *cachedJDOClasses*.
- if the requested metadata has already been searched in the past, the result is stored in a JDOClass and returned; otherwise, the result is searched in the *properties file* and it's stored in the HashMap, in order to get retrieved if the same properties will be searched in the future.

In order to achieve its goals, the *JDOMetaDataProperties* class uses 3 static nested class:

JDOClass
: used to hold the parsed result of the search for metadata concerning a class; when *getJDOClass* is invoked, a *JDOClass* object is returned

JDOField
: used to hold the parsed result of the search for metadata concerning a field of a class; when *getJDOField* is invoked, a *JDOField* object is returned

Property
: used to hold the name and the value of a property

It's also used a static nested Interface **IErrorMessages** that stores the needed error messages as *static variables*.

2.2 Application of the checklist

We report below the result of the application of the assigned checklist; we are going to report only the unrespected points, and the respected ones that we thought deserves some explanation. We are going to omit the trivially respected ones.

JDOMetaDataProperties class In this paragraph we are going to report the result of the application of the points of the checklist that must be applied to the whole *JDOMetaDataProperties* class, not only to assigned methods.

19. from line 965 to 995, a whole method has been commented out; the JavaDoc specifies that this is a test method, but there is no reason to maintain it.

23. The Javadoc of the class is almost complete; the only undocumented methods are *getModifiers* method at line 945, and *getModifiers* method at line 1168; however, they are only getter methods so the JavaDoc documentation is generally good.

25.D. static variable *NULL*, declared at line 184, is not declared in the correct order, because the static variables block ends at line 163.

29. all class fields have a *private* visibility, that is the lowest. So this constraints are respected.

44.3. in the class there is a great amount of *static final* fields, exactly 20. The creation of a suitable *enum* could lead to a cleaner solution; in particular, the code of *parseJDOField* method, from line 31 to 47, could improve its readability and compactness.

51. at line 249, an implicit cast from JDOClass type to String type appears.

parseJDOField method

11. There are some parenthesisation errors in the if-else block between lines 34 and 49:
    - the body of *if* instruction at line 39 contains a single instruction and it's not surrounded by curly braces.
    - the bodies of the *else* instructions at line 39, 42 and 46 contain a single instruction, and they are not surrounded by curly braces.
    - the couple of curly braces at lines 38 and 41 is useless.

13. line 27 is 93 characters long, but the length is due to the length of the method and parameter names; so the constraint is not violated.

14. line 27, analyzed at point 13, doesn't exceed 120 characters, so the constraint is not violated.

19. line 53 contains an useless comment containing the method name

33. the variable *field* is declared at line 30; it should have been declared at line 18 and only initialized at line 30. It couldn't have been also initialized at line 18 because it depends from the computation at line 23 and 27.

validateFieldProperty method

11. the bodies of *else* statements at line 30 and 38 contain a single instruction, and they are not surrounded by curly braces

18. line 98 contains an useless comment containing the method name

validateDependencies method

11. at line 23, the body of the *else* instruction contains a single instruction but it's not surrounded by curly braces

13. lines 17, 20, 43 exceed the 80 characters of length;but only line 17, that is a comment line could have been shortened.

14. lines 28, 29, 34, 35, 42 exceed the 120 characters

18. line 59 contains an useless comment containing the method name.

References

[1] Vlada Matena & Mark Hapner, *Enterprise JavaBeans^{TM} Specification, v1.1*, Sun Microsystems, Inc., Final Release, 1999.

3 Hours of Work

- Beri Giovanni: ~ 15 hours
- Festa Biagio: ~ 15 hours