POLITECNICO
MILANO 1863

Computer Science and Engineering

SOFTWARE ENGINEERING 2

A.A. 2015 / 2016

Glassfish 4.1

Inspection Document

January 5th, 2016

REFERENCE PROFESSOR:

Mirandola Raffaela

AUTHORS:

Polidori Lucia

Regondi Chiara

Table of contents

1. **Introduction** 3

2. **Classes** 3

3. **Functional Role** 4

- ***3.1*** Preliminary considerations 4
- ***3.2*** Class: EjbDescriptor.java 5
- ***3.3*** Methods 6

4. **Issues** 15

5. **Used Tools** 22

6. **Working Hours** 22

1 Introduction

This document contains the summary of our code inspection activity, applied on a part of Java code, extracted from a release of the Glassfish 4.1 application server.
Glassfish is an open-source server for the Java EE platform: actually it is the reference implementation of Java EE (the standard). This allows developers to create enterprise applications that are portable and scalable.
In order to fulfill the code inspection assignment, we have been assigned six methods belonging to the same class that we have to check and inspect, with the purpose of evaluating the general quality of these pieces of code.
We have decided to structure this document as follows:

- a chapter dedicated to the description of the classes assigned to our group
- a chapter for the description of the functional role of those classes
- a chapter dedicated to the issues found during the inspection
- two final paragraph describing the tools we used to develop this document and the working hours we spent on it.

2 Classes

We describe in this section the classes assigned to us for the inspection activity.
In particular, there is only one java class which has been analyzed.
The class is **`EjbDescriptor.java`**, located in the following file: `appserver/ejb/ejb-container/src/main/java/org/glassfish/ejb/deployment/descriptor/`**`EjbDescriptor.java`**

The six methods under inspection are all contained in this class:

- `getContainerTransaction()`
  (*start line: 1367*)
- `addPermissionedMethod(MethodPermission mp, MethodDescriptor md)`
  (*start line: 1427*)
- `removePermissionedMethod(MethodPermission mp, MethodDescriptor md)`
  (*start line: 1502*)
- `getStyledPermissionedMethodsByPermission()`
  (*start line: 1544*)
- `convertMethodPermissions()`
  (*start line: 1596*)
- `updateMethodPermissionForMethod(MethodPermission mp, MethodDescriptor md)`
  (*start line: 1653*)

3 Functional Role

We start describing the functional role of the class **`EjbDescriptor.java`** and then we analyze each assigned method individually.

3.1 Preliminary considerations

Before starting to describe in details what our class and our methods do, we have decided to add some preliminary considerations about some notions that we consider quite important and necessary in order to understand properly the following descriptions.
As the piece of code that we have analyzed is part of a bigger project and much wider java code, we provide here some definitions of other classes, objects or methods of the whole code that can help to better understand the functional role of our methods.
So, the one that we provide here is a sort of small glossary that can be used in order to learn what is useful for our functions:

`addMethodPermissionForStyledMethodDescriptor(…,…)`: given a method permission and a method descriptor, this private method of the class `EjbDescriptor` adds a style (style1 or style2) to our tables.

`ContainerTransaction`: it's a final class representing a specification of a transactional behavior.

`EjbBundleDescriptor`: it's an abstract class representing all the configurable deployment information contained in an EJB JAR.

`getContainerTransactionFor(…)`: it's a public method of the class `EjbDescriptor` that, taking a specific method descriptor as parameter, returns a `ContainerTransaction` that represents the assigned container transaction object for the given method object.

`getMethodDescriptors()`: this public method of the class `EjbDescriptor` returns the full `Set` of method descriptors that we have (from all the methods on our home and remote interfaces).

`getMethodPermissions(…)`: given a method descriptor as parameter, this private method of the class `EjbDescriptor` returns the `Set` of method permissions for the specified method.

`getPermissionedMethodsByPermission()`: it's a public method of the class `EjbDescriptor` that returns a `Map` element of the method permissions (keys) that have been assigned to method descriptors (elements).

`getRole()`: it's a public method of the class `MethodPermission` that returns a `Role` object that represents the security role associated with the method permission.

`getTransactionMethodDescriptors()`: it's a public method of the class `EjbDescriptor` that returns a `Collection` containing `MethodDescriptors` for methods which may have an associated transaction attribute.

getUncheckedMethodPermission(): it's a public method of the class `MethodPermission` that returns a `MethodPermission` element that represents an unchecked method permission. Methods associated with such a method permission cannot be invoked by anyone.

IllegalArgumentException: it's a class describing a specific exception thrown to indicate that a method has been passed an illegal or inappropriate argument.

isRoleBased(): it's a public boolean method of the class `MethodPermission` that returns `true` if the method permission is based on a security role.

isExact(): it's a public boolean method of the class `MethodDescriptor` that returns `true` if there's enough information to specify a unique method on an ejb's home or remote interface unambiguously.

Map: java object that maps keys to values. The key and the corresponding value are objects of a specific type/class. A map cannot contain duplicate keys and each key can map to at most one value.

MethodPermission: the method permission definition defines access control based on security roles. Each method permission is an instance of the public class `MethodPermission`, it can be associated to a role and can be unchecked or excluded. Each method permission is constructed on the base of a security role: each `MethodPermission` element includes a list of one or more security roles and a list of one of more methods; all the listed security roles are allowed to invoke all the listed methods.

MethodDescriptor: it's a deployment object representing a single method or a collection of methods on Enterprise Bean classes.

saveMethodPermissionFromDD(…,…): given a method permission and a method descriptor as parameters, this private method of the class `EjbDescriptor` keeps a record of all the method permissions exactly as they were in the DD.

Set: a collection of java objects of a specific type that contains no duplicate elements.

3.2 Class: EjbDescriptor.java

As reported in the Javadoc, this is an abstract class containing all the information about the different types of Enterprise JavaBeans.

```
104  /**
105   * This abstract class encapsulates the meta-information describing
106   * Entity, Session and MessageDriven EJBs.
107   *
```

The Enterprise JavaBeans (EJBs) are server-side software components that implement the business logic of a web application. So, they represent software layers located on an application server in a multi-tier architecture.
As it's written in the Javadoc reported above, there are three different types of EJBs:

- SESSION BEANS: they usually represent an interface between clients and services offered by different server components.
- MESSAGE-DRIVEN BEANS: they are business objects whose execution is activated by messages instead of method calls. They are executed in an asynchronous mode. Through the JMS (Java Message Service), they subscribe to a topic or a queue and they are activated only when they receive a message from that topic or queue they are subscribed to.
- ENTITY BEANS: they represent persistent data maintained in a database.

3.3 Methods

Here the functional role of the methods under analysis:

`getContainerTransaction ()`

as reported in the Javadoc, this method returns a `ContainerTransaction` if all the transactional methods in the class `EjbDescriptor` have the same transaction type, otherwise it returns `null`.

```
1363  /**
1364   * returns a ContainerTransaction if all the transactional methods on
1365   * the ejb descriptor have the same transaction type else return null
1366   */
```

A new vector ("`transactionalMethods`"), that contains all the `MethodDescriptors` to which `ContainerTransactions` can be assigned, is created. The first element of this vector is stored in a `MethodDescriptor` variable ("`md`") and then an `if` clause checks this variable.
If "`md`" is not `null`, a `ContainerTransaction` variable ("`first`") is created and it becomes the container transaction that is associated to "`md`".

```
1367  public ContainerTransaction getContainerTransaction() {
1368      Vector transactionalMethods = new Vector(this.getTransactionMethodDescriptors());
1369      MethodDescriptor md = (MethodDescriptor) transactionalMethods.firstElement();
1370      if (md != null) {
1371          ContainerTransaction first = this.getContainerTransactionFor(md);
```

An iterator scans the "`transactionalMethods`" vector: at each iteration the method checks that the `ContainerTransactions` associated to all the method descriptors stored in the vector are equal.
If they are different, it returns `null`, otherwise the method returns "`first`", that is the `ContainerTransaction` considered.

```
1372          for (Enumeration e = transactionalMethods.elements(); e.hasMoreElements();) {
1373              MethodDescriptor next = (MethodDescriptor) e.nextElement();
1374              ContainerTransaction nextCt = this.getContainerTransactionFor(next);
1375              if (nextCt != null && !nextCt.equals(first)) {
1376                  return null;
1377              }
```

```
1378            }
1379            return first;
1380        }
1381        return null;
1382    }
```

addPermissionedMethod (MethodPermission mp, MethodDescriptor md)

as reported in the Javadoc, this method adds a new method permission to a method or to a set of methods.

```
1421    /**
1422     * Add a new method permission to a method or a set of methods
1423     *
1424     * @param mp is the new method permission to assign
1425     * @param md describe the method or set of methods this permission apply to
1426     */
```

The method accepts two parameters :

- a `MethodPermission` parameter that represents the new method permission to assign;
- a `MethodDescriptor` parameter that describes the method or the set of methods the permission apply to.

If the `bundleDescriptor` returned by the method `getEjbBundleDescriptor()` is `null`, then an exception is thrown with a specific detail message, because it's impossible to add roles when the descriptor is not contained in a bundle.

```
1427    public void addPermissionedMethod(MethodPermission mp, MethodDescriptor md) {
1428        if (getEjbBundleDescriptor() == null) {
1429            throw new IllegalArgumentException(localStrings.getLocalString(
1430                    "enterprise.deployment.exceptioncannotaddrolesdescriptor",
1431                    "Cannot add roles when the descriptor is not part of a bundle"));
1432        }
```

If the method permission taken as parameter is based on a security role and this role is not contained in the `bundleDescriptor` roles, then the name of the method permission's role to assign is checked: if there's any authenticated user role "**" that is equal to that role, then an exception is thrown because it's not possible to add roles when the bundle does not have them.

```
1433        if (mp.isRoleBased()) {
1434            if (!getEjbBundleDescriptor().getRoles().contains(mp.getRole())) {
1435                // Check for the any authenticated user role '**' as this role
1436                // will be implicitly defined when not listed as a security-role
1437                if (!"**".equals(mp.getRole().getName())) {
1438                    throw new IllegalArgumentException(localStrings.getLocalString(
1439                        "enterprise.deployment.exceptioncannotaddrolesbundle",
```

```
1440                     "Cannot add roles when the bundle does not have them"));
1441             }
1442         }
1443     }
```

Finally, the method descriptor taken as parameter is checked: if this is exact, so if there's enough information to specify a unique method on an EJB's home or remote interface unambiguously, then the `updateMethodPermissionForMethod()` is called and the new method permission is added to the list of existing method permissions; otherwise a new style is added to the tables.
The method permission is then saved.

```
1445     if (md.isExact()) {
1446         updateMethodPermissionForMethod(mp, md);
1447     } else {
1448         addMethodPermissionForStyledMethodDescriptor(mp, md);
1449     }
1450 
1451     saveMethodPermissionFromDD(mp, md);
1452 }
```

`removePermissionedMethod (MethodPermission mp, MethodDescriptor md)`
as reported in the Javadoc, this method removes a method permission from a method or from a set of methods.

```
1496 /**
1497  * Remove a method permission from a method or a set of methods
1498  *
1499  * @param mp is the method permission to remove
1500  * @param md describe the method or set of methods this permission apply to
1501  */
```

The parameters accepted as input are similar to the ones used for the previous method:

- a `MethodPermission` parameter that represents the method permission that has to be removed.
- a `MethodDescriptor` parameter that describes the method or the set of methods the permission apply to.

The first part of the method is similar to the one of the previous "add" method, so we are not going to describe it again.
After checking that the `bundleDescriptor` is not `null` and that the method permission taken as parameter is based on a security role which is not contained in the `bundleDescriptor` roles,
the method checks if the method permission ("`mp`") we want to remove is contained as key in the `Map` of `MethodPermission` (keys) that have been assigned to `MethodDescriptors` (elements).

This Map element is returned calling the getter getPermissionedMethodsByPermission() on the current element.
If so, a new set is created ("alreadyPermissionedMethodsForThisRole"): the method permission ("mp") assigned as parameter is extracted from the said Map and assigned to the new set created.
At this point the method descriptor ("md") is removed from the said set.
Finally, a new pair (key, element) is inserted in the Map, setting the method permission ("mp") as key and the set created as element.

```
1516      if (this.getPermissionedMethodsByPermission().containsKey(mp)) {
1517           Set alreadyPermissionedMethodsForThisRole = (Set)
                                        this.getPermissionedMethodsByPermission().get(mp);
1518           alreadyPermissionedMethodsForThisRole.remove(md);
1519           this.getPermissionedMethodsByPermission().put(mp,
                                                  alreadyPermissionedMethodsForThisRole);
1520            }
```

getStyledPermissionedMethodsByPermission ()

as reported in the Javadoc, this method returns a map of permission to style 1 or 2 method descriptors.

```
1541  /**
1542   * @return a map of permission to style 1 or 2 method descriptors
1543  */
```

If the HashMap variable "styledMethodDescriptors" is null, so there are not methods of style 1 or style 2, the whole method returns null. If this variable is not null, a new HashMap is created ("styledMethodDescriptorsByPermission") where the method descriptors (of style 1 or style 2) with permission will be inserted.
An iterator scans all the method descriptors of the HashMap "styledMethodDescriptors": the method fills, at each iteration, a new Set ("methodPermissions") with the permissions of each method descriptor considered (a method descriptor can have zero, one or more permissions); so, in this case, the methodDescriptors are the keys. Inside this iteration, there's a nested iteration that scans all the method permissions, for each method permission contained in the Set "methodPermissions": the method fills a new Set ("methodDescriptors") with the method descriptors permissioned with that permission (now, the "methodPermissions" are the keys).
So now, in the Set attribute "methodDescriptor" there are only the method descriptors permissioned.

```
1544  public Map getStyledPermissionedMethodsByPermission() {
1545      if (styledMethodDescriptors == null) {
1546          return null;
1547      }
1548
1549      // the current info is structured as MethodDescriptors as keys to
```

```
1550      // method permission, let's reverse this to make the Map using the
1551      // method permission as a key.

1552      Map styledMethodDescriptorsByPermission = new HashMap();
1553      for (Iterator mdIterator = styledMethodDescriptors.keySet().iterator();
                                                                  mdIterator.hasNext();) {
1554          MethodDescriptor md = (MethodDescriptor) mdIterator.next();
1555          Set methodPermissions = (Set) styledMethodDescriptors.get(md);
1556          for (Iterator mpIterator = methodPermissions.iterator();
                                                                  mpIterator.hasNext();) {
1557             MethodPermission mp = (MethodPermission) mpIterator.next();
1558
1559             Set methodDescriptors = (Set) styledMethodDescriptorsByPermission.get(mp);
```

If the "methodDescriptors" set is null, so if there are not permissioned methods with that permission, a new HashSet is created. Otherwise, the method descriptor considered is added to the set "methodDescriptors" and the HasMap "styledMethodDescriptorsByPermission" is updated (in fact, the method permission and the method descriptors permissioned with that permission are added to this HashMap).
The method returns this HashMap, where there are, on one side, the permissions and, on the other side, for every permission, there are the method descriptors with this permission.

```
1560             if (methodDescriptors == null) {
1561                 methodDescriptors = new HashSet();
1562             }
1563             methodDescriptors.add(md);
1564             styledMethodDescriptorsByPermission.put(mp, methodDescriptors);
1565          }
1566      }
1567       return styledMethodDescriptorsByPermission;
1568  }
```

convertMethodPermissions ()

as reported in the Javadoc, this method converts all style 1 and style 2 method descriptors contained in the tables into style 3 method descriptors.

```
1592  /**
1593   * convert all style 1 and style 2 method descriptors contained in
1594   * our tables into style 3 method descriptors.
1595   */
```

If the HashMap "styledMethodDescriptors" is null, the method returns null. This HashMap contains all the methods that need a conversion. The Set variable "allMethods" contains all the method descriptors of this class (EbjDescriptor). Initially, the Set variable "unpermissionedMethods" contains all the methods descriptors of this class, so it is equal to "allMethods".

```
1596  private void convertMethodPermissions() {
1597
1598      if (styledMethodDescriptors == null)
1599          return;
1600
1601      Set allMethods = getMethodDescriptors();
1602      Set unpermissionedMethods = getMethodDescriptors();
```

The `Set` variable "`methodDescriptors`" contains all the methods descriptors of style 1 or style 2, so they are the methods that need a conversion. An iterator scans all the method descriptors of the `Set` "`methodDescriptors`", that are the method descriptors that need a conversion : the method fills, at each iteration, the `Set` variable "`newPermissions`", with the permissions of the method descriptors we are scanning.

```
1604      Set methodDescriptors = styledMethodDescriptors.keySet();
1605      for (Iterator styledMdItr = methodDescriptors.iterator();
                                                    styledMdItr.hasNext();) {
1606          MethodDescriptor styledMd = (MethodDescriptor) styledMdItr.next();
1607
1608          // Get the new permissions we are trying to set for this
1609          // method(s)
1610          Set newPermissions = (Set) styledMethodDescriptors.get(styledMd);
```

A new vector "`mds`" converts all the methods of style 1 and style 2 to style 3.
An iterator scans all the method descriptors that have been converted and all these methods are removed from the set of not-permissioned methods and they are put in a set of new permissioned method ("`newMp`"). The table with method permissions and method descriptors is updated. All the remaining methods should now be defined as unchecked.

```
1612          // Convert to style 3 method descriptors
1613          Vector mds = styledMd.doStyleConversion(this, allMethods);
1614          for (Iterator mdItr = mds.iterator(); mdItr.hasNext();) {
1615              MethodDescriptor md = (MethodDescriptor) mdItr.next();
1616
1617              // remove it from the list of unpermissioned methods.
1618              // it will be used at the end to set all remaining methods
1619              // with the unchecked method permission
1620              unpermissionedMethods.remove(md);
1621
1622              // iterator over the new set of method permissions for that
1623              // method descriptor and update the table
1624              for (Iterator newPermissionsItr = newPermissions.iterator();
                                                  newPermissionsItr.hasNext();) {
1625                  MethodPermission newMp = (MethodPermission)
                                                      newPermissionsItr.next();
1626                  updateMethodPermissionForMethod(newMp, md);
1627              }
1628          }
1629      }
```

```
1630
1631        // All remaining methods should now be defined as unchecked...

1632        MethodPermission mp = MethodPermission.getUncheckedMethodPermission();
1633        Iterator iterator = unpermissionedMethods.iterator();
1634        while (iterator.hasNext()) {
1635            MethodDescriptor md = (MethodDescriptor) iterator.next();
1636            if (getMethodPermissions(md).isEmpty()) {
1637                addMethodPermissionForMethod(mp, md);
1638            }
1639        }
```

Finally the list of method descriptors that need a style conversion is reset.

```
1641        // finally we reset the list of method descriptors that need style conversion
1642        styledMethodDescriptors = null;
1643    }
```

updateMethodPermissionForMethod (MethodPermission mp, MethodDescriptor md)

as reported in the Javadoc, this method updates a method descriptor set of method permission with a new method permission given in input. This new method permission is added to the list of existing method permissions respecting the correct priorities.

```
1645    /**
1646     * Update a method descriptor set of method permission with a new method permission
1647     * The new method permission is added to the list of existing method permissions
1648     * given it respect the EJB 2.0 paragraph 21.3.2 on priorities of method permissions
1649     *
1650     * @param mp is the method permission to be added
1651     * @param md is the method descriptor (style3 only) to add the method permission to
1652     */
```

The parameters accepted as input are similar to the ones used for the previous methods:

- a `MethodPermission` parameter that represents the method permission that has to be added;
- a `MethodDescriptor` parameter that represents the method descriptor (rigorously of style3) to which the method permission is added.

First of all, a new set is created (“`oldPermissions`”) containing the current set of method permissions for the specific method.

If this set is empty, then the operation is easy because the new method permission is added.

```
1655        // Get the current set of method permissions for that method
1656        Set oldPermissions = getMethodPermissions(md);
1657
```

```
1658      if (oldPermissions.isEmpty()) {
1659          // this is easy, just add the new one

1660          addMethodPermissionForMethod(mp, md);
1661          return;
1662      }
```

If the set of method permissions is not empty, it's necessary to check the type of method permission: if the given method permission ("`mp`") is excluded, then it takes precedence on any other form of method permission, so it can be added after removing all the existing method permission.
An iterator on the "`oldPermission`" set is created and the set is scanned: all the elements contained in the set are removed and finally the excluded method permission is added.

```
1670     if (mp.isExcluded()) {
1671     // Excluded methods takes precedence on any other form of method permission
1672     // remove all existing method permission...
1673     for (Iterator oldPermissionsItr = oldPermissions.iterator(); oldPermissionsItr.hasNext();;) {
1674         MethodPermission oldMp = (MethodPermission) oldPermissionsItr.next();
1675         removePermissionedMethod(oldMp, md);
1676     }
1677     // add the excluded
1678     addMethodPermissionForMethod(mp, md);
1679     } else {
```

Otherwise, if the method permission is unchecked, it is considered like a role-based method permission and it's added to the list.
An iterator scans the "`oldPermission`" set and for each element taken into consideration at each iteration, the method checks if it is excluded.
If the method permission is not excluded, then it is removed and the new method permission is added.

```
1679      } else {
1680          if (mp.isUnchecked()) {
1681              // we are trying to add an unchecked method permisison, all role-based
1682              // method permission should be removed since unchecked is now used, if a
1683              // particular method has an excluded method permision, we do not add it
1684              for (Iterator oldPermissionsItr = oldPermissions.iterator();
                                                              oldPermissionsItr.hasNext();;) {
1685              MethodPermission oldMp = (MethodPermission) oldPermissionsItr.next();
1686              if (!oldMp.isExcluded()) {
1687                  removePermissionedMethod(oldMp, md);
1688                  addMethodPermissionForMethod(mp, md);
1689              }
1690          }
1691     } else {
```

Finally, if the method permission ("`mp`") is neither excluded nor unchecked, then it surely is a role-based method permission. An iterator scan the "`oldPermission`" set: the method checks, at each iteration, that unchecked or excluded method permissions have not been set and, if so, it adds the new method permission to the current list of role-based permission.

```
1691    } else {
1692        // we are trying to add a role based method permission. Check that
1693        // unchecked or excluded method permissions have not been set

1694        // and add it to the current list of role based permission
1695        for (Iterator oldPermissionsItr = oldPermissions.iterator();
                                                                    oldPermissionsItr.hasNext();) {
1696            MethodPermission oldMp = (MethodPermission) oldPermissionsItr.next();
1697            if (!oldMp.isExcluded()) {
1698                if (!oldMp.isUnchecked()) {
1699                    addMethodPermissionForMethod(mp, md);
1700                }
1701            }
1702        }
1703    }
```

4 Issues

In this chapter we are going to describe the issues and the critical aspects listed in the checklist that we have found during our inspection activity.

First of all, here are some ***issues concerning*** those points in the checklist that are about ***the whole class***.

- **Java Source Files** (*Checklist: point 23*)
  The Javadoc of our class `EjbDescriptor.java` is not complete: not all the methods contained in this class are properly described through the Javadoc; some of them do not even have any Javadoc.
  It would be better to write a complete and clarifying Javadoc for the class and for all the methods, in order to make the code easier to understand.
- **Class and Interface Declaration** (*Checklist: points 25, 26*)
  As for the class and interface declarations, the class variables and the instance variables of the class are not ordered properly: it's quite confusing the way in which they are declared. They should follow a specific order of declaration, according to their access level modifiers (`public`, `protected`, `private`).
  Moreover, the order given to the methods described in this class is not correct.
  The constructor should be the first method declared, followed by all the other methods: here, we have the declaration of a method (`getIASEjbExtraDescriptors()`) before the default constructor.
  Finally, all the other methods described are not properly grouped: it should be better to group them by functionality (for example: all the getters, all the setters, and so on…) in order to improve the readability of the code.

If all these critical aspects are corrected, the code will result clearer and the quality of the software will improve.

Now, we will list the issues found in the methods we have been assigned:

1) **`getContainerTransaction ()`**

There are no comments in this method: it would be better to insert some comments that properly explain what the blocks of code are doing.
The Javadoc describing the method is not complete: it simply describes what the method does, but as this method does not return `void`, but a typed object, the clause "`@return`" must be insert in the Javadoc.
At lines 1372-1378, there's a `for` iteration that can be replaced with a `while` iteration, like this:

```
Enumeration e = transactionalMethods.elements();
while (e.hasMoreElements()) {
    MethodDescriptor next = (MethodDescriptor) e.nextElement();

}
```

The method `firstElement()` at line 1369 and the method `nextElement()` at line 1373 throw a `NoSuchElementException` that is not properly managed.

2) **`addPermissionedMethod (MethodPermission mp, MethodDescriptor md)`**

There are few poor comments in this method: it would be better to insert some other comments in order to better explain what the blocks of code are doing.

We also have some issues of naming conventions: it would be better to substitute the insignificant names "`mp`" and "`md`" with some meaningful names, such as "`methodDscrp`" and "`methodPerms`".

As for the file organization, we have noticed some blank lines missing or overused: it's needed a blank line to highlight the beginning of the comment between line 1434 and line 1435, while the blank line at line 1444 should be removed.

In the `if` clause starting at line 1437, we have some indention and wrapping lines issues: the argument of the `IllegalArgumentException` is too long to be written on the same line, so it has to be divided properly using line breaks; in particular we have to indent the lines 1439 and 1440 using 8 spaces, starting from the position of the "`throw`" instruction in the previous line, in this way:

```
if (!"**".equals(mp.getRole().getName())) {
    throw new IllegalArgumentException(localStrings.getLocalString(
        "enterprise.deployment.exceptioncannotaddrolesbundle",
        "Cannot add roles when the bundle does not have them"));
}
```

8 indention spaces

This is needed, because the other rules about breaks (applied when an expression doesn't fit on a single line) lead to code that is squished up against the right margin.

3) **`removePermissionedMethod (MethodPermission mp, MethodDescriptor md)`**

Just like in the previous method, there are few poor comments in the code: it would be better to insert some other comments in order to better explain what the blocks of code are doing.

We also have here some issues of naming conventions: it would be better to substitute the insignificant names "`mp`" and "`md`" with some meaningful names, such as "`methodDscrp`" and "`methodPerms`".

As for the file organization, there are some useless blank spaces: at line 1515 and at line 1521.

Some methods called inside this one throw exceptions that are not properly managed; in particular:

- containsKey(mp) at line 1516 :
  ClassCastException, NullPointerException
- get(mp) at line 1517 :
  ClassCastException, NullPointerException
- remove(md) at line 1518 :
  ClassCastException, NullPointerException, UnsupportedOperationException
- put(mp, alreadyPermissionedMethodsByPermission) at line 1519 :
  ClassCastException, NullPointerException, UnsupportedOperationException, IllegalArgumentException

4) **getStyledPermissionedMethodsByPermission ()**

The Javadoc provided doesn't describe the method properly.

It would be better to replace the variable names "md" and "mp" with some more consistent and significant names: for examples "methodDscrp" and "methodPerms".

After line number 1557, there's an useless blank line that can be removed.

The comments inserted inside the method start without capital letters; it's necessary to replace them with capital letters.

There are some for iterations that has to be replaced with while iteration, like we have already stated before:

Lines 1552-1556

```
Iterator mdIterator = styledMethodDescriptors.keySet().iterator();
while (mdIterator.hasNext()){
    MethodDescriptor md = (MethodDescriptor) mdIterator.next();
…

}
```

Lines 1556-1565

```
Iterator mpIterator = methodPermissions.iterator();
while (mpIterator.hasNext()){
    MethodPermission mp = (MethodPermission) mpIterator.next();
…

}
```

Some methods called inside this one throw exceptions that are not properly managed; in particular:

- next() at lines 1554 and 1557 :
  NoSuchElementExceptionn
- get(md) and get(mp) at lines 1555 and 1559:
  ClassCastException, NullPointerException

- `add(md)` at line 1563 :

  `ClassCastException, NullPointerException, UnsupportedOperationException, IllegalArgumentException`
- `put(mp, methodDescriptors)` at line 1564 :

  `ClassCastException, NullPointerException, UnsupportedOperationException, IllegalArgumentException`

5) **`convertMethodPermissions ()`**

It would be better to replace the variable names "`md`" and "`mp`" with some more consistent and significant names: for example "`methodDscrp`" and "`methodPerms`".

At lines 1598 and 1599 a not consistent bracing styled is used: it should be replaced with something like:

```
if (styledMethodDescriptors == null){
     return;
}
```

After line number 1602, there's an useless blank line that can be removed.

The comments begin without capital letters; it's necessary to replace them with capital letters.

There are some `for` iteration that has to be replaced with `while` iteration, like we have already stated before:

Line 1605-1629

```
Iterator styledMdItr = methodDescriptors.iterator();
while(styledMdItr.hasNext()){
    MethodDescriptor styledMd = (MethodDescriptor) styledMdItr.next();
…

}
```

Line 1614-1628

```
Iterator mdItr = mds.iterator();
while (mdItr.hasNext()) {
    MethodDescriptor md = (MethodDescriptor) mdItr.next();
…

}
```

Line 1624-1627

```
Iterator newPermissionsItr = newPermissions.iterator();
while (newPermissionsItr.hasNext()){
    MethodPermission newMp = (MethodPermission) newPermissionsItr.next();
…

}
```

Some methods called inside this one throw exceptions that are not properly managed; in particular:

- `get(styledMd)` at line1610 :
  `ClassCastException, NullPointerException`
- `remove(md)` at line 1620 :
  `ClassCastException, NullPointerException, UnsupportedOperationException`
- `next()` at lines 1625 and 1635 :
  NoSuchElementException

6) **`updateMethodPermissionForMethod (MethodPermission mp, MethodDescriptor md)`**

We also have here naming conventions issues: the variable names "`md`" and "`mp`" should be substituted with some more consistent and significant names, such as "`methodDscrp`" and "`methodPerms`". Also the variable name "`oldMp`" should be substituted with a meaningful name such as "`oldMethodPerm`".

As for the file organization, there are some missing blank spaces useful to highlight the beginning comments: between lines 1658 and 1659, between lines 1670 and 1671, between lines 1676 and 1677, between lines 1680 and 1681 and between lines 1691 and 1692.

There's also an useless blank space at line 1669.

Some comments begin without capital letters; it's necessary to replace them with capital letters.

The method `next()` at lines 1674, 1685 and 1696 throws a `NoSuchElementException` that is not properly managed.

Finally, some general considerations concerning all the methods analyzed: all the lines of code that exceed 80 characters are necessary, because it's not possible to insert a line break to separate them into two lines. No lines exceed 120 characters.

All the object comparisons use the keyword "`equals`" (as it should be), except the ones that compare an object to `null`: if we substitute an expression like "`object == null`" with the expression using the method `equals()`, that is "`object.equals(null)`", the latter will always return `false` (due to the standard implementation of the method `equals()`) or throw a `NullPointerException` if the `object` itself is `null`. So, we haven't highlighted as errors the object comparisons like "`if(object == null)`" in any of our methods.

In the following pages a schematic summary table containing all the issues found during the inspection is reported.

METHOD	ISSUES
1)	**Comments:** there are no comments in the method. **Java Source Files:** the Javadoc at the beginning of the method is inaccurate. **Computation, Comparisons and Assignments:** there is a `for` iteration that should be replaced with a `while` iteration. **Exceptions:** some exceptions thrown by methods invoked are not properly managed.
2)	**Naming Conventions:** the names "`md`" and "`mp`" are not meaningful. **File Organization / Wrapping Lines:** useless blank line (line 1444); blank line needed (between lines 1434 and 1435). **Indention / Wrapping Lines:** lines 1439 and 1440 are not properly indented: 8 spaces are needed, starting from the position of the "`throw`" instruction in the previous line. **Comments:** there are no comments in the method.
3)	**Naming Conventions:** the names "`md`" and "`mp`" are not meaningful. **File Organization / Wrapping Lines:** useless blank lines (line 1515 and line 1521). **Comments:** there are no comments in the method. **Exceptions:** some exceptions thrown by methods invoked are not properly managed.

4)	**Naming Conventions:** the names “`md`” and “`mp`” are not meaningful. **File Organization / Wrapping Lines:** useless blank line (line 1558). **Comments:** the comments begin without capital letters. **Java Source Files:** the Javadoc at the beginning of the method is inaccurate. **Computation, Comparisons and Assignments:** there are some `for` iterations that should be replaced with `while` iterations. **Exceptions:** some exceptions thrown by methods invoked are not properly managed.
5)	**Naming Conventions:** the names “`md`” and “`mp`” are not meaningful. **Braces:** No consistent bracing styled is used. **File Organization / Wrapping Lines:** useless blank line (line 1603). **Comments:** the comments begin without capital letters. **Computation, Comparisons and Assignments:** there are some `for` iterations that should be replaced with `while` iterations. **Exceptions:** some exceptions thrown by methods invoked are not properly managed.
6)	**Naming Conventions:** the names “`md`”, “`mp`” and “`oldMp`” are not meaningful. **File Organization / Wrapping Lines:** useless blank line (line 1669); blank lines needed (between lines 1658 and 1659, between lines 1670, and 1671, between lines 1676 and 1677, between line 1680 and 1681 and between lines 1691 and 1692). **Exceptions:** some exceptions thrown by methods invoked are not properly managed.

5 Used Tools

In this paragraph we are going to list all the tools we have used to create this document:

- *Microsoft Office Word 2010* : to redact and format this document
- *Eclipse* : to analyze the code we had to inspect

6 Working Hours

In order to analyze the code, redact and write this document we spent 20 hours per person.