# Code Inspection Document

Software Engineering 2

Fioratto Raffaele, Longoni Nicolò
Politecnico di Milano

A.Y. 2015/2016

Version 1.0

January 5, 2016

# Contents

# List of Figures

# Listings

# 1 Introduction

## 1.1 Purpose

Code Inspection document collects all results obtained by doing Code Inspection. The overall goal is to assess quality of a software by means of formal methods called Code Inspection techniques and more informal procedures.
Probably the most important one is to apply a set of rules, which can be seen as a sort of checklist, to the code and to find whether it follows or not such rules. This procedure will be described more deeply in next sections and it is the key concept that will be used throughout the document.
Doing Code Inspection has several benefits: to find coding mistakes is the most important and oversights that might be arisen during initial development phase in a systematic way. By reporting these issues in a document like this one, developers can acknowledge this kind of problems and then fix them. This will take to a more robust code and also it can potentially increase skills of developers themselves, so they hopefully will not repeating errors during following development phases in the same project or even in other ones, eventually this will reduce costs, the time needed to create a software in each of its phases and it will optimize the overall process.

## 1.2 Scope

As said before, Code Inspection is applied to an existing software project which is being developed. In this case, we are going to inspect "Glassfish": an open source implementation of a JEE server. This project is written mainly in Java using Maven as a build tool, for this reason, its structure is highly modularized.
Since there are a lot of contributors in this project, the entire code base is managed using a VCS, specifically SVN. A VCS is a centralized service that keeps track and provides control of all modifications made by contributors. Every time the source code of the software is modified, a new revision is uploaded to servers so that every contributor can check, discuss and improve changes. Furthermore as long as servers are reliable, it is possible to eliminate local backups by developers.
This document focuses on a particular version and revision of that software which is Glassfish v4.1.1 at revision 64219. The entire code base consists of $\sim 1,000,000$ (Physical) SLoC[1], which $\approx 84\%$ are written in Java and $\approx 14\%$ in XML, the rest is written in other programming languages such as Python, Pascal and Ruby. Some important things to note are Development Effort Estimate which is $\approx 280$ Person-Years, (i.e. it requires almost 300 years to an average working person to develop this project on his own), and also Total Estimated Cost to Develop this software which is $\approx$ \$38.1M, so it is trivial to understand that Glassfish is a very big piece of software and it is worth spending time and effort focusing on code inspection in order to improve quality and reliability of this software.

[1]Code statistics generated using David A. Wheeler's "SLOCCount".

## 1.3 Acronyms and Abbreviations

### 1.3.1 Acronyms

- JEE: **J**ava **E**nterprise **E**dition
- VCS: **V**ersion **C**ontrol **S**ystem
- SVN: Apache **S**ub**V**ersio**N**
- API: **A**pplication **P**rogramming **I**nterface
- XML: e**X**tensible **M**arkup **L**anguage
- DTD: **D**ocument **T**ype **D**escription
- SAX: **S**imple **A**PI for **X**ML
- DOM: **D**ocument **O**bject **M**odel
- DOL: **D**eployment **O**bject **L**ibrary
- SLoC: **S**ource **L**ines of **C**ode

### 1.3.2 Abbreviations

- Cn: $n^{th}$ checklist element
- Ln: $n^{th}$ line of code
- Li-j: lines of code in the interval i-j

## 1.4 Reference Documents

- Code Inspection assignment document
- Glassfish Javadoc Documentation[2]
- Oracle JEE Documentation[3]

## 1.5 Document Structure

- Section 1: Introduction, it gives a description of this document, some basic information in order to clearly understand subsequent sections.
- Section 2: Classes assigned, it briefly lists a set of classes that will be inspected throughout next sections.
- Section 3: Functional role, it describes what assigned classes do and how we determined this with the respect to some evidences such as Javadoc, diagrams and so on.

[2]`http://glassfish.pompel.me/`

[3]`http://docs.oracle.com/javaee/7/index.html`

- Section 4: Issues found, it collects all problems found in analyzed code. For each item is stated which rules defined in the checklist mentioned before are violated and why. Snippets of violated code are also provided.
- Section 5: Other problems, it includes additional issues found during inspection that are not covered in the checklist, but worthy to be mentioned, so that potential software defects can be corrected.
- Section 6: Appendices, other extra information regarding this document.

## 2 Classes assigned

The following list includes a set of classes of the software's source code assigned to us and their respective packages in which they reside. Actually, in our assignment, there is only one class to inspect and it is declared as follows:

```java
public abstract class DeploymentDescriptorNode<T> implements
  ↪ XMLNode<T>  {
```

Listing 1: `DeploymentDescriptorNode` declaration.

This class resides in a package declared at the beginning of the source Java file:

```java
package com.sun.enterprise.deployment.node;
```

Listing 2: `DeploymentDescriptorNode` package membership declaration.

That package is inside a module called `Deployment Object Library`.
Its pathname is
`appserver/deployment/dol/src/main/java/com/sun/enterprise/deployment/node`, and the filename of the source code is `DeploymentDescriptorNode.java`. Below there is a diagram which describes a hierarchy for this class:

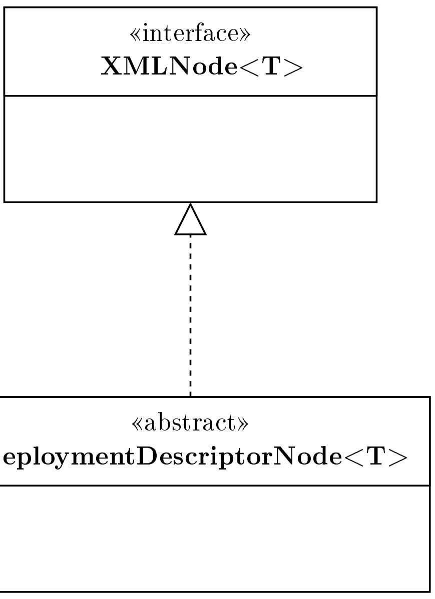

Figure 1: Class hierarchy for `DeploymentDescriptorNode`.

## 3 Functional Role

DeploymentDescriptorNode is a base class responsible for handling a XML deployment descriptor, it implements the interface XMLNode.
The XML Deployment Descriptor describes how a web application should be deployed. Each node reimplements this base class and overrides a bunch of methods for handling specific tag names in a XML file. For instance, we show here the code of a subclass of DeploymentDescriptorNode which is called EjbReferenceNode, it is responsible for handling ejb-reference XML node tags. We briefly analyze methods that are overridden.
The first method is getDescriptor() which returns a descriptor of a XML Node being handled and encapsulates data regarding its XML node such as attributes and sub-tag names.

```
@Override
public EjbReference getDescriptor() {
    if (descriptor==null) {
        descriptor = new EjbReferenceDescriptor();
        descriptor.setLocal(false);
    }
    return descriptor;
```

Listing 3: EjbReferenceNode.getDescriptor() method implementation.

Notice that, this method is implemented using Singleton design pattern.
Second method is called getDispatchTable() which returns a Map associating each tag name to an appropriate function implemented in a Descriptor object returned by getDescriptor(). These functions will be called at runtime using reflection. With this approach, it is possible to achieve a high degree of flexibility since a parent class doesn't need to know anything about its children, they just supply to it how to handle at runtime each subtag and populate correctly its descriptor. getDispatchTable() method is implemented as follows:

```
@Override
protected Map getDispatchTable() {
    Map table = super.getDispatchTable();
    table.put(TagNames.EJB_REFERENCE_NAME, "setName");
    table.put(TagNames.EJB_REFERENCE_TYPE, "setType");
    table.put(TagNames.HOME, "setEjbHomeInterface");
    table.put(TagNames.REMOTE, "setEjbInterface");
    table.put(TagNames.LOCAL_HOME, "setEjbHomeInterface");
    table.put(TagNames.LOCAL, "setEjbInterface");
    table.put(TagNames.EJB_LINK, "setLinkName");
    table.put(TagNames.MAPPED_NAME, "setMappedName");
    table.put(TagNames.LOOKUP_NAME, "setLookupName");
    return table;
}
```

Listing 4: EjbReferenceNode.getDispatchTable() method implementation.

At last, `writeDescriptor()` method is responsible to perform the actual construction of a DOM tree node taking values from a `Descriptor` object whose attributes were populated previously. Eventually the DOM node is returned so that it will be inserted in the overall tree, this function is implemented by this code:

```java
@Override
public Node writeDescriptor(Node parent, String nodeName, EjbReference descriptor) {
    Node ejbRefNode = appendChild(parent, nodeName);
    if (descriptor instanceof Descriptor) {
        Descriptor ejbRefDesc = (Descriptor)descriptor;
        writeLocalizedDescriptions(ejbRefNode, ejbRefDesc);
    }
    appendTextChild(ejbRefNode, TagNames.EJB_REFERENCE_NAME, descriptor.getName());
    appendTextChild(ejbRefNode, TagNames.EJB_REFERENCE_TYPE, descriptor.getType());
    if (descriptor.isLocal()) {
        appendTextChild(ejbRefNode, TagNames.LOCAL_HOME, descriptor.getEjbHomeInterface());
        appendTextChild(ejbRefNode, TagNames.LOCAL, descriptor.getEjbInterface());
    } else {
        appendTextChild(ejbRefNode, TagNames.HOME, descriptor.getEjbHomeInterface());
        appendTextChild(ejbRefNode, TagNames.REMOTE, descriptor.getEjbInterface());
    }
    appendTextChild(ejbRefNode, TagNames.EJB_LINK, descriptor.getLinkName());

    if( descriptor instanceof EnvironmentProperty) {
        EnvironmentProperty envProp = (EnvironmentProperty)descriptor;
        appendTextChild(ejbRefNode, TagNames.MAPPED_NAME, envProp.getMappedName());
    }
    if( descriptor.isInjectable() ) {
        InjectionTargetNode ijNode = new InjectionTargetNode();
        for (InjectionTarget target : descriptor.getInjectionTargets()) {
            ijNode.writeDescriptor(ejbRefNode, TagNames.INJECTION_TARGET, target);
        }
    }

    if( descriptor.hasLookupName() ) {
        appendTextChild(ejbRefNode, TagNames.LOOKUP_NAME,
            ↪ descriptor.getLookupName());
    }

    return ejbRefNode;
}
```

Listing 5: `EjbReferenceNode.writeDescriptor()` method implementation.

As explained before, a subclass is responsible for writing a DOM representation of a handled tag, each tag is composed by attributes and possibly subtags. These last ones can be managed by a same class or delegated to other subclasses which are registered to the parent class.
This implies that all classes inheriting from `DeploymentDescriptorNode` are shaped like a tree with one root `DeploymentDescriptorNode` and they are arranged according to DTD.
SAX parser is responsible of interacting with correct subclasses depending on which tag is encountered while parsing a XML file by calling the method
`XMLNode.getHandlerFor(element)`. `element` parameter is an instance of `XMLElement` class which contains simple attributes to encapsulate tag names and (possible) XML namespaces. This function returns an object whose static type is `XMLNode` but its runtime type is one of possible subclasses of
`DeploymentDescriptorNode` which will handle a tag as described previously.
Eventually a full DOM tree of the XML is created in memory which will be manipulated later by Glassfish in order to deploy the web application.
This information has been gathered directly by the documentation of the source code written in Javadoc and it is placed within the code itself. Documentation is also publicly available from Glassfish official site. Another important source of information comes from Oracle, which points out in its documentation, what a XML Deployment Descriptor is and its role in the whole JEE ecosystem.

# 4 Issues found

This section comprises all problems found by applying the checklist provided in the Code Inspection assignment document, only violated rules are reported here: we are assuming that if inspected code is consistent with respect to a particular rule, it will be not listed here. Issues are grouped by method.

## 4.1 Issues related to `DeploymentDescriptorNode` class

1. C7 L120-121. The following class attribute is declared as `static final` and therefore its name should be in uppercase.

```
120 protected static final LocalStringManagerImpl
      ↪ localStrings=
121     new LocalStringManagerImpl(DeploymentDescriptorNode.
          ↪ class);
```

Listing 6: C7 violation at L120-121

2. C25D and C25E L88-121. Instance and class variables are mixed together and also they are not grouped by scope visibility.

```
 88 protected ServiceLocator habitat = Globals.
      ↪ getDefaultHabitat();
 89
 90 private static final String QNAME_SEPARATOR = ":";
 91
 92 // handlers is the list of XMLNodes  registered for
      ↪ handling sub xml tags of the current
 93 // XMLNode
 94 protected Hashtable<String, Class> handlers = null;
 95
 96 // list of add methods declared on the descriptor class
      ↪ to add sub descriptors extracted
 97 // by the handlers registered above. The key for the
      ↪ table is the xml root tag for the
 98 // descriptor to be added, the value is the method name
      ↪ to add such descriptor to the
 99 // current descriptor.
100 private Hashtable addMethods = null;
101
102 // Each node is associated with a XML tag it is handling
103 private XMLElement xmlTag;
104
105 // Parent node in the XML Nodes implementation tree we
      ↪ create to map to the XML
106 // tags of the XML document
107 protected XMLNode parentNode = null;
108
```

```
109 // The root xml node in the XML Node implementation tree
110 protected XMLNode rootNode = null;
111
112 // default descriptor associated with this node, some sub nodes which
113 // relies on the dispatch table don't really need to know the actual
114 // type of the descriptor they deal with since they are populated through
115 // reflection method calls
116 protected Object abstractDescriptor;
117
118
119 // for i18N
120 protected static final LocalStringManagerImpl localStrings=
121     new LocalStringManagerImpl(DeploymentDescriptorNode.class);
```
Listing 7: C25D and C25E violations at L88-121

3. C27 The overall class is 585 lines long and contains many methods, so it is better to split it in order to improve maintainability and to increase cohesion.

## 4.2 Issues affecting method `handlesElement()`

This method begins at L394 and ends at L426, below there is a list of violations found between this line range.

1. C13 L399. Line length exceeds 80 characters and can be broken at ";" just before the condition statement.

```
399 for (Enumeration handlersIterator = handlers.keys(); handlersIterator.hasMoreElements();) {
```
Listing 8: C13 violation at L399

2. C31 L401, L403, L411, L416, L419 and L423. These statements don't check properly if either `element` or `element.getQName()` is not `null` before using only one of them or both.

```
401 if (element.getQName().equals(subElement)) {
```
Listing 9: C31 violation at L401

```
403 recordNodeMapping(element.getQName(), handlers.get(subElement));
```
Listing 10: C31 violation at L403

```
411 Class extHandler = getExtensionHandler(element);
```
Listing 11: C31 violation at L411

```
416 registerElementHandler(new XMLElement(element.getQName())
      ↪ ,
```
Listing 12: C31 violation at L416

```
419 recordNodeMapping(element.getQName(), extHandler);
```
Listing 13: C31 violation at L419

```
423 recordNodeMapping(element.getQName(), this.getClass());
```
Listing 14: C31 violation at L423

3. C33 L411. Move this statement to the beginning of the block.

```
411 Class extHandler = getExtensionHandler(element);
```
Listing 15: C33 violation at L411

4. C56 L399-400. Loop isn't well formed since increment statement is inside its block of code rather than in initialization.

```
399 for (Enumeration handlersIterator = handlers.keys();
      ↪ handlersIterator.hasMoreElements();) {
400     String subElement  = (String) handlersIterator.
          ↪ nextElement();
```
Listing 16: C56 violation at L399-400

## 4.3 Issues related to method `setElementValue()`

This method begins at L442 and ends at L491. Here it's a list of violations found within this method:

1. C2 L468. Avoid using variables with short names.

```
468         Throwable t = e.getTargetException();
```
Listing 17: C2 violation at L468

2. C9 L443 and L468. These lines are indented using tabs, consider replacing them with spaces.

```
443     //DOLUtils.getDefaultLogger().finer("SETELEMENTVALUE
          ↪ : " + "in " + getXMLRootTag() + "  Node,
          ↪ startElement " + element.getQName());
```
Listing 18: C9 violation at L443

```
468         Throwable t = e.getTargetException();
```
Listing 19: C9 violation at L468

3. C13 L470-471. Consider wrapping this comment in order not to exceed 80 characters in a single line.

```
470 // We report the error but we continue loading, this will
    ↪  allow the verifier to catch these errors or to
    ↪ register
471 // an error handler for notification
```

Listing 20: C13 violation at L470-471

4. C13 L458. Consider splitting this function call at “,”.

```
458 setDescriptorInfo(descriptor, (String) dispatchTable.get(
    ↪ element.getQName()), (String) value);
```

Listing 21: C13 violation at L458

5. C14 L443. Commented statement length exceeds 120 characters, it is advisable to split it before 3rd “+”.

```
443     //DOLUtils.getDefaultLogger().finer("SETELEMENTVALUE
        ↪ : " + "in " + getXMLRootTag() + "  Node,
        ↪ startElement " + element.getQName());
```

Listing 22: C14 violation at L443

6. C14 L451. String concatenation statement length inside this function call exceeds 120 characters, consider wrapping it at 1st “+”.

```
451 DOLUtils.getDefaultLogger().finer("Deprecated element " +
    ↪  element.getQName() + " with value " + value + " is
    ↪ ignored");
```

Listing 23: C14 violation at L451

7. C19 L443. Consider either to remove this comment from statement or add a reason why it is there and optionally add a date when it can be removed.

```
443     //DOLUtils.getDefaultLogger().finer("SETELEMENTVALUE
        ↪ : " + "in " + getXMLRootTag() + "  Node,
        ↪ startElement " + element.getQName());
```

Listing 24: C19 violation at L443

8. C31 L447, L448, L458, L460-461, L466-467, L472-473 and L487-488. These statements don’t check properly if either `element` or `element.getQName()` is not `null` before using only one of them or both.

```
447 if (dispatchTable.containsKey(element.getQName())) {
```

Listing 25: C31 violation at L447

```
448 if (dispatchTable.get(element.getQName())==null) {
```

Listing 26: C31 violation at L448

```
458 setDescriptorInfo(descriptor, (String) dispatchTable.get(
    ↪ element.getQName()), (String) value);
```

Listing 27: C31 violation at L458

```
460 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
461     new Object[] {element.getQName() , value });
```
Listing 28: C31 violation at L460-461

```
466 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
467     new Object[] {dispatchTable.get(element.getQName()) , getDescriptor().getClass()});
```
Listing 29: C31 violation at L466-467

```
472 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
473     new Object[] {element , value});
```
Listing 30: C31 violation at L472-473

```
487 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
488     new Object[] {element.getQName() , value });
```
Listing 31: C31 violation at L487-488

9. C31 L458, L460-461, L472-473, L486 and L487-488. These statements don't check properly if `value` is not `null` before using it.

```
458 setDescriptorInfo(descriptor, (String) dispatchTable.get(element.getQName()), (String) value);
```
Listing 32: C31 violation at L458

```
460 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
461     new Object[] {element.getQName() , value });
```
Listing 33: C31 violation at L460-461

```
472 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
473     new Object[] {element , value});
```
Listing 34: C31 violation at L472-473

```
486 if (value.trim().length()!=0) {
```
Listing 35: C31 violation at L486

```
487 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
488     new Object[] {element.getQName() , value });
```
Listing 36: C31 violation at L487-488

10. C31 L458, L460-461, L472-473, L486 and L487-488. The following statements don't properly check if the returned value by `getDescriptor()` is not `null` before using it.

```java
467 new Object[] {dispatchTable.get(element.getQName()) , getDescriptor().getClass()});
468         Throwable t = e.getTargetException();
```

Listing 37: C31 violation at L467-468

11. C33 L468. Move this statement to the beginning of `catch` block.

```java
465 } catch (InvocationTargetException e) {
466     DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.INVALID_DESC_MAPPING,
467         new Object[] {dispatchTable.get(element.getQName()) , getDescriptor().getClass()});
468         Throwable t = e.getTargetException();
```

Listing 38: C33 violation at L468

12. C42 L477 and L482. Generic "`Error occurred`" message, consider to change it with a more explicative one.

```java
477 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error occurred", t);
```

Listing 39: C42 violation at L477

```java
482 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error occurred", t);
```

Listing 40: C42 violation at L482

13. C44 L486. Consider using `String.isEmpty()` which is already provided by Java standard library.

```java
486 if (value.trim().length()!=0) {
```

Listing 41: C44 violation at L486

14. C50 L479. Catching a `Throwable` can lead to errors, consider to use at least `Exception` or more specific subclasses, this can also improve code readability.

```java
479 } catch(Throwable t) {
```

Listing 42: C50 violation at L479

15. C52 L447, L448, L458 and L467. These statements should be surrounded with a proper `try-catch` block for`NullPointerException` runtime exception. This exception is raised if either `Map.containsKey()` or `Map.get()` is equal to `null`. Note that if this problem is fixed also issues described at point 8 are automatically corrected. It is possible either to check manually if preconditions of these methods are met as described in point 8 or to catch the relevant exception as pointed out in this issue.

```
447 if (dispatchTable.containsKey(element.getQName())) {
```
Listing 43: C52 violation at L447

```
448 if (dispatchTable.get(element.getQName())==null) {
```
Listing 44: C52 violation at L448

```
458 setDescriptorInfo(descriptor, (String) dispatchTable.get(
      ↪ element.getQName()), (String) value);
```
Listing 45: C52 violation at L458

```
467 new Object[] {dispatchTable.get(element.getQName()) ,
      ↪ getDescriptor().getClass()});
```
Listing 46: C52 violation at L467

## 4.4 Issues related to method `setDescriptorInfo()`

This method begins at L512 and ends at L537 and a list of violations regarding this method is the following:

1. C9 L519-523, L530-531 and L536. These lines are indented totally or partially using tabs.

```
519
520         try {
521             Method toInvoke = target.getClass().getMethod(
                  ↪ methodName, new Class[] { String.class });
522             toInvoke.invoke(target, new Object[] {value});
523         } catch(NoSuchMethodException e1) {
```
Listing 47: C9 violation at L519-523

```
530                 new Object []{ getXMLPath() , nfe.toString()
                      ↪ });
531             } catch(NoSuchMethodException e2) {
```
Listing 48: C9 violation at L530-531

```
536         }
```
Listing 49: C9 violation at L536

2. C13 L521, L526 and L533. Consider splitting this function calls at “,”.

```
521         Method toInvoke = target.getClass().getMethod(
              ↪ methodName, new Class[] { String.class });
```
Listing 50: C13 violation at L521

```
526 Method toInvoke = target.getClass().getMethod(methodName,
      ↪  new Class[] { int.class });
```
Listing 51: C13 violation at L526

```
533 Method toInvoke = target.getClass().getMethod(methodName, new Class[] { boolean.class });
```

Listing 52: C13 violation at L533

3. C14 L517. String concatenation statement length inside function call exceeds 120 characters, consider to wrap it at 2nd "+".

```
517 DOLUtils.getDefaultLogger().fine("in " + target.getClass() + " method " + methodName + " with " + value);
```

Listing 53: C14 violation at L517

4. C31 L522, L527 and L534. These statements don't check properly if `value` is not `null` before using it.

```
522     toInvoke.invoke(target, new Object[] {value});
```

Listing 54: C31 violation at L522

```
527 toInvoke.invoke(target, new Object[] {Integer.valueOf(value)});
```

Listing 55: C31 violation at L527

```
534 toInvoke.invoke(target, new Object[] {Boolean.valueOf(value)});
```

Listing 56: C31 violation at L534

5. C31 L521-522, L526-527 and L533-534. These statements don't check properly if `target` is not `null` before using it.

```
521     Method toInvoke = target.getClass().getMethod(methodName, new Class[] { String.class });
522     toInvoke.invoke(target, new Object[] {value});
```

Listing 57: C31 violation at L521-522

```
526 Method toInvoke = target.getClass().getMethod(methodName, new Class[] { int.class });
527 toInvoke.invoke(target, new Object[] {Integer.valueOf(value)});
```

Listing 58: C31 violation at L526-527

```
533 Method toInvoke = target.getClass().getMethod(methodName, new Class[] { boolean.class });
534 toInvoke.invoke(target, new Object[] {Boolean.valueOf(value)});
```

Listing 59: C31 violation at L533-534

6. C31 L521, L526 and L533. These statements don't check properly if `methodName` is not `null` before using it.

```
521     Method toInvoke = target.getClass().getMethod(
          ↪ methodName, new Class[] { String.class });
```

Listing 60: C31 violation at L521

```
526 Method toInvoke = target.getClass().getMethod(methodName,
      ↪  new Class[] { int.class });
```

Listing 61: C31 violation at L526

```
533 Method toInvoke = target.getClass().getMethod(methodName,
      ↪  new Class[] { boolean.class });
```

Listing 62: C31 violation at L533

7. C42 L517. This message is misleading. It says that it's already in its target method of class but its invocation hasn't occurred yet. This could potentially lead to confusion if there are problems invoking this method at runtime using reflection, i.e. if at least one of the exceptions is caught.

```
517 DOLUtils.getDefaultLogger().fine("in " + target.getClass
      ↪ () + "  method  " + methodName +  " with  " + value
      ↪ );
```

Listing 63: C42 violation at L517

8. C53 L524-535. Consider catching all checked exceptions within the method rather than propagating the last one to caller in case all tries fail. If an error is unrecoverable, a good solution is to throw a dedicated exception with a meaningful message containing an explanation about the exceptional event.

```
524 try {
525     // try with int as a parameter
526     Method toInvoke = target.getClass().getMethod(
          ↪ methodName, new Class[] { int.class });
527     toInvoke.invoke(target, new Object[] {Integer.valueOf
          ↪ (value)});
528 } catch (NumberFormatException nfe) {
529     DOLUtils.getDefaultLogger().log(Level.WARNING,
          ↪ DOLUtils.INVALID_DESC_MAPPING,
530             new Object []{ getXMLPath() , nfe.toString()
                  ↪ });
531     } catch(NoSuchMethodException e2) {
532     // try with boolean as a parameter
533     Method toInvoke = target.getClass().getMethod(
          ↪ methodName, new Class[] { boolean.class });
534     toInvoke.invoke(target, new Object[] {Boolean.valueOf
          ↪ (value)});
535 }
```

Listing 64: C53 violation at L524-535

## 4.5 Issues related to method `writeSubDescriptors()`

This method begins at line L626 and ends at line L664. This is a list of issues found in it:

1. C1 L626. Parameters passed `name`, `nodeName` and `descriptor` don't describe precisely their function in this method.

```
626 public Node writeSubDescriptors(Node node, String
      ↪ nodeName, Descriptor descriptor) {
```

Listing 65: C1 violation at L626

2. C13 L626, L630 and L653. These lines exceeded 80 characters, it can be inserted a break after = in last two cases. In first case it's suggested to begin a new line after a comma.

```
626 public Node writeSubDescriptors(Node node, String
      ↪ nodeName, Descriptor descriptor) {
```

Listing 66: C13 violation at L626

```
630 LinkedHashMap<String, Class> elementToNodeMappings = ((
      ↪ RuntimeBundleNode)rootNode).getNodeMappings(nodeName
      ↪ );
```

Listing 67: C13 violation at L630

```
653 DeploymentDescriptorNode subNode = (
      ↪ DeploymentDescriptorNode)handlerClass.newInstance();
```

Listing 68: C13 violation at L653

3. C17 L639. Wrong identation with respect to L637.

```
637 String subElementName = entry.getKey();
638   // skip if it's the element itself and not the
        ↪ subelement
639   if (subElementName.equals(nodeName)) {
```

Listing 69: C17 violation at L639

## 4.6 Issues related to method `writeEjbReferenceDescriptors()`

This method begins at line L799 and ends at line L. Issues in this method are:

1. C11 L801-802. `If` followed by only one statement has to stay between curly brackets.

```
801 if (refs==null || !refs.hasNext())
802     return;
```

Listing 70: C11 violation at L801-802

2. C18 L805 and L815. These comments aren't useful to explain their blocks of code.

```
805 // ejb-ref*
806 Set localRefDescs = new HashSet();
807 for (;refs.hasNext();) {
808     EjbReference ejbRef = (EjbReference) refs.next();
809     if (ejbRef.isLocal()) {
810         localRefDescs.add(ejbRef);
811     } else {
812         subNode.writeDescriptor(parentNode, TagNames.
             ↪ EJB_REFERENCE, ejbRef);
813     }
814 }
```

Listing 71: C18 violation at L805-814

```
815 // ejb-local-ref*
816 for (Iterator e=localRefDescs.iterator(); e.hasNext();) {
817     EjbReference ejbRef = (EjbReference) e.next();
818     subNode.writeDescriptor(parentNode, TagNames.
         ↪ EJB_LOCAL_REFERENCE,ejbRef);
819 }
```

Listing 72: C18 violation at L815-819

3. C29 L804. This attribute is used only to invoke `writeDescriptor` method, it is better if `subNode` is created when that method has to be invoked.

```
804 EjbReferenceNode subNode = new EjbReferenceNode();
```

Listing 73: C29 violation at L804

4. C40 L801. Comparison has to be done with `equals()` method.

```
801 if (refs==null || !refs.hasNext())
```

Listing 74: C40 violation at L801

5. C56 L816-829. Looking this `for` cycle, it is better to iterate `localRefDescs` with a variable which will use directly as parameter in `writeDescriptor()` method.

```
816 for (Iterator e=localRefDescs.iterator(); e.hasNext();) {
817     EjbReference ejbRef = (EjbReference) e.next();
818     subNode.writeDescriptor(parentNode, TagNames.
         ↪ EJB_LOCAL_REFERENCE,ejbRef);
819 }
```

Listing 75: C56 violation at L816-819

## 5 Other problems

Although the checklist is composed by many rules, we believed that there are some checks that can be made in this code, in order to prevent software defects or misbehaviours. This section encloses additional issues that we think that are important to highlight so that when corrected, the overall code will be more robust and readable.

### 5.1 Other problems related to `DeploymentDescriptorNode` class

1. L85. Abstract classes should be named `AbstractXXX`.

```
85 public abstract class DeploymentDescriptorNode<T>
     ↪ implements XMLNode<T>  {
```

Listing 76: Class name not starting with `Abstract`

2. This class has too many methods, consider refactoring it.
3. L94 and L100. Avoid using implementation types like “`Hashtable`”; instead use an interface.

```
94 protected Hashtable<String, Class> handlers = null;
```

Listing 77: Use of “`Hashtable`” instead of its base interface (1/2)

```
100 private Hashtable addMethods = null;
```

Listing 78: Use of “`Hashtable`” instead of its base interface (2/2)

4. The String literal “`enterprise.deployment.backend.addDescriptorFailure`” appears 5 times in this file (L152, L194, L198, L202 and L204); consider declaring it as a constant.

```
152 DOLUtils.getDefaultLogger().log(Level.SEVERE, "enterprise
     ↪ .deployment.backend.addDescriptorFailure",
```

Listing 79: Multiple usage of “`enterprise.deployment.backend.addDescriptorFailure`” (1/5)

```
194 DOLUtils.getDefaultLogger().log(Level.SEVERE, "enterprise
     ↪ .deployment.backend.addDescriptorFailure",
```

Listing 80: Multiple usage of “`enterprise.deployment.backend.addDescriptorFailure`” (2/5)

```
198 DOLUtils.getDefaultLogger().log(Level.SEVERE, "enterprise
     ↪ .deployment.backend.addDescriptorFailure",
```

Listing 81: Multiple usage of “`enterprise.deployment.backend.addDescriptorFailure`” (3/5)

```
202 DOLUtils.getDefaultLogger().log(Level.SEVERE, "enterprise
     ↪ .deployment.backend.addDescriptorFailure",
```

Listing 82: Multiple usage of “`enterprise.deployment.backend.addDescriptorFailure`” (4/5)

```
204 DOLUtils.getDefaultLogger().log(Level.SEVERE, "enterprise
    ↪ .deployment.backend.addDescriptorFailure",
```
Listing 83: Multiple usage of "enterprise.deployment.backend.addDescriptorFailure" (5/5)

5. The String literal "Error occurred" appears 5 times in this file (L197, L206, L309, L477 and L482); consider declaring it as a constant.

```
197 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
    ↪ occurred", t);
```
Listing 84: Multiple usage of "Error occurred" (1/5)

```
206 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
    ↪ occurred", t);
```
Listing 85: Multiple usage of "Error occurred" (2/5)

```
309 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
    ↪ occurred", e);
```
Listing 86: Multiple usage of "Error occurred" (3/5)

```
477 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
    ↪ occurred", t);
```
Listing 87: Multiple usage of "Error occurred" (4/5)

```
482 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
    ↪ occurred", t);
```
Listing 88: Multiple usage of "Error occurred" (5/5)

## 5.2 Other problems affecting method `handlesElement()`

1. L394. Add "@Override" annotation above this method signature.

```
394 public boolean handlesElement(XMLElement element) {
```
Listing 89: Missing "@Override" annotation in `handlesElement()`

2. L401. Consider using a local variable with value returned by `element.getQName()`, instead of chaining function calls.

```
401 if (element.getQName().equals(subElement)) {
```
Listing 90: `element.getQName().equals(...)` methods chaining

3. L399. Consider declaring a variable using type `Enumeration<String>` in order to avoid casts in L400. This improves readability.

```
399 for (Enumeration handlersIterator = handlers.keys();
    ↪ handlersIterator.hasMoreElements();) {
400     String subElement = (String) handlersIterator.
        ↪ nextElement();
```
Listing 91: Add diamond operators to "Enumeration" with enclosing type "String" in variable declaration

## 5.3 Other problems related to method `setElementValue()`

1. L442. Add “`@Override`” annotation above this method signature.

```
442 public void setElementValue(XMLElement element, String
      ↪ value) {
```

Listing 92: Missing “`@Override`” annotation in `setElementValue()`

2. L447. Merge this `if` statement with the enclosing one.

```
446 if (dispatchTable != null) {
447     if (dispatchTable.containsKey(element.getQName())) {
```

Listing 93: Nested `if`s

3. L450 and L457. Refactor this code not to nest more than 3 `if`/`for`/`while`/`switch`/`try` statements.

4. L450, L451, L460, L466, L472, L475, L477, L480, L482 and L487. Consider using a local variable with value returned by `DOLUtils.getDefaultLogger()`, instead of chaining function calls.

```
450 if (DOLUtils.getDefaultLogger().isLoggable(Level.FINER))
      ↪ {
```

Listing 94: `DOLUtils.getDefaultLogger().isLoggable(...)` methods chaining

```
451 DOLUtils.getDefaultLogger().finer("Deprecated element " +
      ↪  element.getQName() + " with value " + value + " is
      ↪ ignored");
```

Listing 95: `DOLUtils.getDefaultLogger().finer(...)` methods chaining

```
460 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```

Listing 96: `DOLUtils.getDefaultLogger().log(...)` methods chaining (1/8)

```
466 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```

Listing 97: `DOLUtils.getDefaultLogger().log(...)` methods chaining (2/8)

```
472 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```

Listing 98: `DOLUtils.getDefaultLogger().log(...)` methods chaining (3/8)

```
475 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```

Listing 99: `DOLUtils.getDefaultLogger().log(...)` methods chaining (4/8)

```
477 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
      ↪ occurred", t);
```

Listing 100: `DOLUtils.getDefaultLogger().log(...)` methods chaining (5/8)

```
480 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```
Listing 101: `DOLUtils.getDefaultLogger().log(...)` methods chaining (6/8)

```
482 DOLUtils.getDefaultLogger().log(Level.WARNING, "Error
      ↪ occurred", t);
```
Listing 102: `DOLUtils.getDefaultLogger().log(...)` methods chaining (7/8)

```
487 DOLUtils.getDefaultLogger().log(Level.WARNING, DOLUtils.
      ↪ INVALID_DESC_MAPPING,
```
Listing 103: `DOLUtils.getDefaultLogger().log(...)` methods chaining (8/8)

5. L490. Avoid unnecessary `return` statements.

```
490 return;
```
Listing 104: Unnecessary `return` statement at end of method

6. L444. Consider declaring a variable using type `Map<String, String>` in order to avoid casts in L458. This improves readability.

```
444 Map dispatchTable = getDispatchTable();
```
Listing 105: Add diamond operators to “`Map`” with enclosing type “`String, String`” in variable declaration

7. L458. Useless cast to `String` of parameter `value` which is already declared with type `String`.

```
458 setDescriptorInfo(descriptor, (String) dispatchTable.get(
      ↪ element.getQName()), (String) value);
```
Listing 106: Useless cast to `String` in variable declaration

## 5.4 Other problems related to method `setDescriptorInfo()`

1. L516 and L517. Consider using a local variable with value returned by `DOLUtils.getDefaultLogger()`, instead of chaining function calls.

```
516 if (DOLUtils.getDefaultLogger().isLoggable(Level.FINE)) {
```
Listing 107: `DOLUtils.getDefaultLogger().isLoggable(...)` methods chaining

```
517 DOLUtils.getDefaultLogger().fine("in " + target.getClass
      ↪ () + " method " + methodName + " with " + value
      ↪ );
```
Listing 108: `DOLUtils.getDefaultLogger().finer(...)` methods chaining

## 5.5 Other problems related to method `writeSubDescriptors()`

1. L643. Avoid to use a chain of methods

```
643 if (handlerClass.getName().equals(this.getClass().getName())) {
```

Listing 109: Concatenation of methods

2. L657. Consider using a local variable with value returned by `DOLUtils.getDefaultLogger()` method, instead of chaining function calls.

```
657 DOLUtils.getDefaultLogger().log(Level.WARNING, e.getMessage(), e);
```

Listing 110: `DOLUtils.getDefaultLogger().log(...)` methods chaining

## 5.6 Other problems related to method `writeEjbReferenceDescriptors()`

1. L806. Consider using type `<EjbReference>` when a `Set` and a `HashSet` are declared to avoid errors caused by insertion of different type elements.

```
806 Set localRefDescs = new HashSet();
```

Listing 111: Add diamond operator `<EjbReference>`

# Appendices

## A Tools

- Document written in LaTeX
- Basic MiKTeX 2.9.5721 64-bit – `http://miktex.org/`
- Texmaker 4.5 – `http://www.xm1math.net/texmaker/index.html`
- draw.io – `https://www.draw.io/`

## B Hours of Work

- Raffaele Fioratto: 18 hours
- Nicolò Longoni: 18 hours

## C Code inspection checklist

Below there is a checklist used to find violations in the code reported in the Issues found section. It is provided as a reference to readers to better understand all reported problems.

### Naming Conventions

C1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

C2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.

C3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;

C4. Interface names should be capitalized like classes.

C5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.

C6. 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`.

C7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

### Indention

C8. Three or four spaces are used for indentation and done so consistently.

C9. No tabs are used to indent.

### Braces

C10. 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).

C11. 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();
}
```

## File Organization

C12. 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).

C13. Where practical, line length does not exceed 80 characters.

C14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

## Wrapping Lines

C15. Line break occurs after a comma or an operator.

C16. Higher-level breaks are used.

C17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

## Comments

C18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

C19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

## Java Source Files

C20. Each Java source file contains a single public class or interface.

C21. The public class is the first class or interface in the file.

C22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

C23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

## Package and Import Statements

C24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

## Class and Interface Declarations

C25. 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;
  - b. next protected instance variables;
  - c. next package level (no access modifier);
  - d. last private instance variables.
- F. constructors;
- G. methods.

C26. Methods are grouped by functionality rather than by scope or accessibility.

C27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

## Initialization and Declarations

C28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

C29. Check that variables are declared in the proper scope.

C30. Check that constructors are called when a new object is desired.

C31. Check that all object references are initialized before use.

C32. Variables are initialized where they are declared, unless dependent upon a computation.

C33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '`{`' and '`}`'). The exception is a variable can be declared in a `for` loop.

## Method Calls

C34. Check that parameters are presented in the correct order.

C35. Check that the correct method is being called, or should it be a different method with a similar name.

C36. Check that method returned values are used properly.

## Arrays

C37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

C38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

C39. Check that constructors are called when a new array item is desired.

## Object Comparison

C40. Check that all objects (including Strings) are compared with `equals` and not with `==`.

## Output Format

C41. Check that displayed output is free of spelling and grammatical errors.

C42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.

C43. Check that the output is formatted correctly in terms of line stepping and spacing.

## Computation, Comparisons and Assignments

C44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).

C45. Check order of computation/evaluation, operator precedence and parenthesizing.

C46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

C47. Check that all denominators of a division are prevented from being zero.

C48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.

C49. Check that the comparison and Boolean operators are correct.

C50. Check throw-catch expressions, and check that the error condition is actually legitimate.

C51. Check that the code is free of any implicit type conversions.

## Exceptions

C52. Check that the relevant exceptions are caught.

C53. Check that the appropriate action are taken for each catch block.

## Flow of Control

C54. In a `switch` statement, check that all cases are addressed by break or return.

C55. Check that all switch statements have a default branch.

C56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

## Files

C57. Check that all files are properly declared and opened.

C58. Check that all files are closed properly, even in the case of an error.

C59. Check that EOF conditions are detected and handled correctly.

C60. Check that all file exceptions are caught and dealt with accordingly.