

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
M.Sc. in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria



Software Engineering 2 - Project

CI

Code Inspection

version 1.0

5th January 2016

Authors:
Alberto Maria METELLI Matr. 850141
Riccardo MOLOGNI Matr. 852416

Academic Year 2015–2016

Contents

1	Introduction	1
1.1	Purpose	1
1.2	References	1
1.3	Code inspection checklist	1
1.3.1	Naming Conventions	1
1.3.2	Indentation	2
1.3.3	Braces	2
1.3.4	File Organization	2
1.3.5	Wrapping Lines	2
1.3.6	Comments	3
1.3.7	Java Source Files	3
1.3.8	Package and Import Statements	3
1.3.9	Class and Interface Declarations	3
1.3.10	Initialization and Declarations	4
1.3.11	Method Calls	4
1.3.12	Arrays	4
1.3.13	Object Comparison	4
1.3.14	Output Format	4
1.3.15	Computation, Comparisons and Assignments	5
1.3.16	Exceptions	5
1.3.17	Flow of Control	5
1.3.18	Files	5
1.4	Our code inspection process	5
1.5	Document Structure	6
2	Assigned classes	7
2.1	Classes	7
2.1.1	ComponentEnvManagerImpl	7
2.2	Methods	7
2.2.1	getJndiNameEnvironment	7
2.2.2	bindToComponentNamespace	7
2.2.3	addAllDescriptorBindings	7
2.2.4	unbindFromComponentNamespace	8

3	Functional roles of classes	9
3.1	About JNDI (from Oracle doc)	9
3.1.1	Java EE Naming Environment	9
3.1.2	How the Naming Environment and the Container Work Together	9
3.2	Diagrams	11
3.3	Classes	12
3.3.1	ComponentEnvManagerImpl	12
3.4	Methods	12
3.4.1	getJndiNameEnvironment	12
3.4.2	bindToComponentNamespace	12
3.4.3	addAllDescriptorBindings	12
3.4.4	unbindFromComponentNamespace	12
4	Issues	13
4.1	Issue list	13
4.1.1	Naiming conventions	13
4.1.2	Indentation	13
4.1.3	Braces	14
4.1.4	File Organization	14
4.1.5	Wrapping Lines	16
4.1.6	Comments	16
4.1.7	Java Source Files	18
4.1.8	Package and Import Statements	19
4.1.9	Class and interface declaration	19
4.1.10	Initialization and declarations	21
4.1.11	Method calls	23
4.1.12	Arrays	23
4.1.13	Object comparision	23
4.1.14	Output format	23
4.1.15	Computation, Comparitions and Assignments	24
4.1.16	Exceptions	24
4.1.17	Flow of control	24
4.1.18	Files	24
4.2	Cumulative data	25
4.2.1	Issues per category and class/method - table	25
4.2.2	Issues per category - hystogram	26
4.2.3	Issues per class/method per KSLOC - hystogram	27
4.2.4	Issues per class/method per KSLOC - pie chart	28

5	Other problems	29
5.1	Automatic code review	29
5.2	Other issues from manual inspection	33
A	Appendix	35

List of Figures

1	UML Class Diagram of the main methods and dependencies	11
2	Hystogram showing the distribution of issues in categories.	26
3	Hystogram showing the distribution of issues in method/class.	27
4	Pie chart showing the distribution of issues in method/class.	28

1 Introduction

1.1 Purpose

The term CI (*Code Inspection*) refers to peer review of any work product, involving code, by trained individuals who look for mistakes (also known as defects or issues) using a well defined process. A mistake is an incorrect feature of the code that can result in the introduction of faults into a system which in turn can become errors and failures. Therefore the main purpose of code inspection is to identify defects and, if necessary, improve the quality of the code. Code inspection is a specific case of *static analysis* as a part of the V&V (*Verification* and *Validation*) process.

Code revision can be classified into *walkthroughs* and *inspection*, those are general approaches to revision of products that can be exploited in any stage of the software lifecycle. Even if they share the same goal the revision process is quite different:

- *walkthroughs* are typically informal reviews that involve experts of domain which are in charge of verifying the correctness of the product with respect to their viewpoint. The producer, in case of code walkthrough the developer, presents the code and the attached documentation, if any, and the reviewers discuss the correctness of the product.
- *inspections* are based on formal evaluation techniques proposed by Fagan in which code is exterminated by a group of professional inspectors to check its correctness, typically on the bases of a set of quality standards defined as *checklist*. Contrary to what happens in walkthroughs, here people taking part to the inspection have well-defined roles and the process is composed of several steps: the moderator chooses participants (readers, testers and inspectors) and schedules the meeting that takes place analysing code line-by-line with the support of the explanation of the author.

This document is intended to be a track for the inspection process of an extract of the Glassfish 4.1 application server code we performed. This document is mainly addressed to developers in order to fix possible mistakes highlighted during the inspection process and for possible further inspection sessions.

1.2 References

- [1] Software Engineering 2 course slides.
- [2] Glassfish reference <http://glassfish.pompe1.me/>.
- [3] Oracle documentation https://docs.oracle.com/cd/E18930_01/html/821-2416/ggjue.html#ab11b.

1.3 Code inspection checklist

In this subsection we present the checklist followed to perform code inspection of an extract of the Glassfish 4.1 application server code.

1.3.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary “throwaway” variables, such as those used in for loops.

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`.
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.
5. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter.
6. All the remaining words in the variable name have their first letter capitalized. Examples: `_windowHeight`, `timeSeriesData`.
7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

1.3.2 Indentation

8. Three or four spaces are used for indentation and done so consistently.
9. No tabs are used to indent.

1.3.3 Braces

10. Consistent bracing style is used, either the preferred “Allman” style (first brace goes underneath the opening block) or the “Kernighan and Ritchie” style (first brace is on the same line of the instruction that opens the new block).
11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example, avoid this:

```
if ( condition )
    doThis();
```

Instad do this:

```
if ( condition )
{
    doThis();
}
```

1.3.4 File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).
13. Where practical, line length does not exceed 80 characters.
14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

1.3.5 Wrapping Lines

15. Line break occurs after a comma or an operator.
16. Higher-level breaks are used.
17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

1.3.6 Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.
19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

1.3.7 Java Source Files

20. Each Java source file contains a single public class or interface.
21. The public class is the first class or interface in the file.
22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.
23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

1.3.8 Package and Import Statements

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

1.3.9 Class and Interface Declarations

25. The class or interface declarations shall be in the following order:
 1. A. class/interface documentation comment
 2. class or interface statement
 3. class/interface implementation comment, if necessary
 4. 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
 5. instance variables
 - (a) first public instance variables
 - (b) next protected instance variables
 - (c) next package level (no access modifier)
 - (d) last private instance variables
 6. constructors
 7. methods
26. Methods are grouped by functionality rather than by scope or accessibility.
27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

1.3.10 Initialization and Declarations

- 28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)
- 29. Check that variables are declared in the proper scope
- 30. Check that constructors are called when a new object is desired
- 31. Check that all object references are initialized before use
- 32. Variables are initialized where they are declared, unless dependent upon a computation
- 33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces “{“ and “}”). The exception is a variable can be declared in a ‘for’ loop.

1.3.11 Method Calls

- 34. Check that parameters are presented in the correct order
- 35. Check that the correct method is being called, or should it be a different method with a similar name
- 36. Check that method returned values are used properly

1.3.12 Arrays

- 37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)
- 38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds
- 39. Check that constructors are called when a new array item is desired

1.3.13 Object Comparison

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

1.3.14 Output Format

- 41. Check that displayed output is free of spelling and grammatical errors.
- 42. Check that error messages are comprehensive and provide guidance as to how to correct the problem
- 43. Check that the output is formatted correctly in terms of line stepping and spacing

1.3.15 Computation, Comparisons and Assignments

44. Check that the implementation avoids “brutish programming: (see <http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html>)
45. Check order of computation/evaluation, operator precedence and parenthesizing
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding
49. Check that the comparison and Boolean operators are correct
50. Check throw-catch expressions, and check that the error condition is actually legitimate
51. Check that the code is free of any implicit type conversions

1.3.16 Exceptions

52. Check that the relevant exceptions are caught
53. Check that the appropriate action are taken for each catch block

1.3.17 Flow of Control

54. In a switch statement, check that all cases are addressed by break or return
55. Check that all switch statements have a default branch
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

1.3.18 Files

57. Check that all files are properly declared and opened
58. Check that all files are closed properly, even in the case of an error
59. Check that EOF conditions are detected and handled correctly
60. Check that all file exceptions are caught and dealt with accordingly

1.4 Our code inspection process

Fagan inspection, as already mentioned, being a formal analysis technique, has a specific process and roles associated to the participants; considered the academic nature of this document we think that such an approach should be inappropriate, therefore beside the manual inspection either based on the given checklist or exploiting other considerations about the code, we leveraged on automatic tools in order to discover further defects. The following list, shows the techniques and tools adopted.

- *Manual inspection*: revision of the code line by line performed by the group members together in order to discover defects according to:

- the *checklist* proposed in the assignment,
- other considerations based on the experience of the group members.
- *Automatic code review*: revision of the code performed by static code analyzers, in particular the following has been used¹:
 - *SonarQube*: an open platform to manage code quality (duplications, complexity, potential bugs, coding rules, comments, architecture and design) <http://www.sonarqube.org/>,
 - *PMD*: a Java source code analyzer aimed to finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation <https://pmd.github.io/>,
 - *FindBugs*: a program which uses static analysis to look for bugs in Java code <http://findbugs.sourceforge.net/>.

1.5 Document Structure

This document is composed of five sections and an appendix.

- The first section, this one, provides an overall description of the review processes focusing on the specific assignment consisting in the inspection of an extract of the Glassfish 4.1 application server code. It also presents the checklist used in the following sections.
- The second section is devoted to the description of the extract of code. Classes and method names will be stated with their location in the source code.
- The third section provides the illustration of the functional role of those classes and methods both in informal language with respect to the JEE architecture and in semiformal way by means of UML class diagram to highlight dependencies between classes. We will focus on the reverse engineering approach we adopted.
- The fourth section is devoted to the inspection. For each category of defects and for each defect description, issues identified will be stated with reference to the line of the code involved, the motivation and a possible solution when appropriate.
- The fifth section describes other possible problems identified during the inspection that do not conform to the points presented in the checklist.
- The appendix contains a brief description of the tools used to produce this documents and the number of hours each group member has worked towards the fulfillment of this deadline.

¹They are all very similar but differ in the importance given to different aspects of code quality.

2 Assigned classes

2.1 Classes

2.1.1 ComponentEnvManagerImpl

- Signature: `public class` ComponentEnvManagerImpl `implements` ComponentEnvManager
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2 Methods

2.2.1 getJndiNameEnvironment

- Signature: `public` JndiNameEnvironment getJndiNameEnvironment(String componentId)
- Start line: 160
- End line: 168
- SLOC: 9
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.2 bindToComponentNamespace

- Signature: `public` String bindToComponentNamespace(JndiNameEnvironment env)`throws` NamingException
- Start line: 188
- End line: 272
- SLOC: 85
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.3 addAllDescriptorBindings

- Signature: `private void` addAllDescriptorBindings addAllDescriptorBindings(JndiNameEnvironment env , ScopeType scope , Collection < JNDIBinding > jndiBindings)
- Start Line: 312
- End line: 366
- SLOC: 55
- Location: appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java

2.2.4 unbindFromComponentNamespace

- Signature: `public void unbindFromComponentNamespace(JndiNameEnvironment env) throws NamingException`
- Start line: 373
- End line: 418
- SLOC: 46
- Location: `appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.java`

3 Functional roles of classes

This section is devoted to the description of the overall functionalities of the class focusing in particular on the role of the methods. Understanding the role of a module or some methods of that module is a non trivial *reverse engineering* task, even more complex when the reference and the documentation is entirely lacking. Therefore we tried, as best as we could, to exploit the code, the few comments present and a variety of online resources to extract at least the general characteristics. The description we provide below is a very high level explanation intended to show the conceptual functionalities of the code assigned without going into details that actually we were not be able to understand because of the complexity and the bad documentation of this part of the project.

The first subsection gives a general description of what JNDI and JNDI environment are in the context of Glassfish, this is necessary to understand the role of the methods; if the reader already knows how they work, subsection 3.1 can be skipped.

3.1 About JNDI (from Oracle doc)

Mainly taken from [3].

By making calls to the JNDI API, applications locate resources and other program objects. A resource is a program object that provides connections to systems, such as database servers and messaging systems. Each resource object is identified by a unique, people-friendly name, called the JNDI name. A resource object and its JNDI name are bound together by the naming and directory service, which is included with the GlassFish Server. When a new name-object binding is entered into the JNDI, a new resource is created.

3.1.1 Java EE Naming Environment

JNDI names are bound to their objects by the naming and directory service that is provided by a Java EE server. Because Java EE components access this service through the JNDI API, the object usually uses its JNDI name.

Java EE application clients, enterprise beans, and web components must have access to a JNDI naming environment.

The *application component's naming environment* is the mechanism that allows customization of the application component's business logic during deployment or assembly. This environment allows you to customize the application component without needing to access or change the source code of the component. A Java EE container implements the provides the environment to the application component instance as a JNDI naming context.

3.1.2 How the Naming Environment and the Container Work Together

The application component's environment is used as follows:

- The application component's business methods access the environment using the JNDI interfaces. In the deployment descriptor, the application component provider declares all the environment entries that the application component expects to be provided in its environment at runtime.
- The container provides an implementation of the JNDI naming context that stores the application component environment. The container also provides the tools that allow the deployer to create and manage the environment of each application component.

- A deployer uses the tools provided by the container to initialize the environment entries that are declared in the application component's deployment descriptor. The deployer sets and modifies the values of the environment entries.
- The container makes the JNDI context available to the application component instances at runtime. These instances use the JNDI interfaces to obtain the values of the environment entries.

Each application component defines its own set of environment entries. All instances of an application component within the same container share the same environment entries. Application component instances are not allowed to modify the environment at runtime.

3.2 Diagrams

A class diagram showing the main methods and dependencies of the assigned class is now provided.

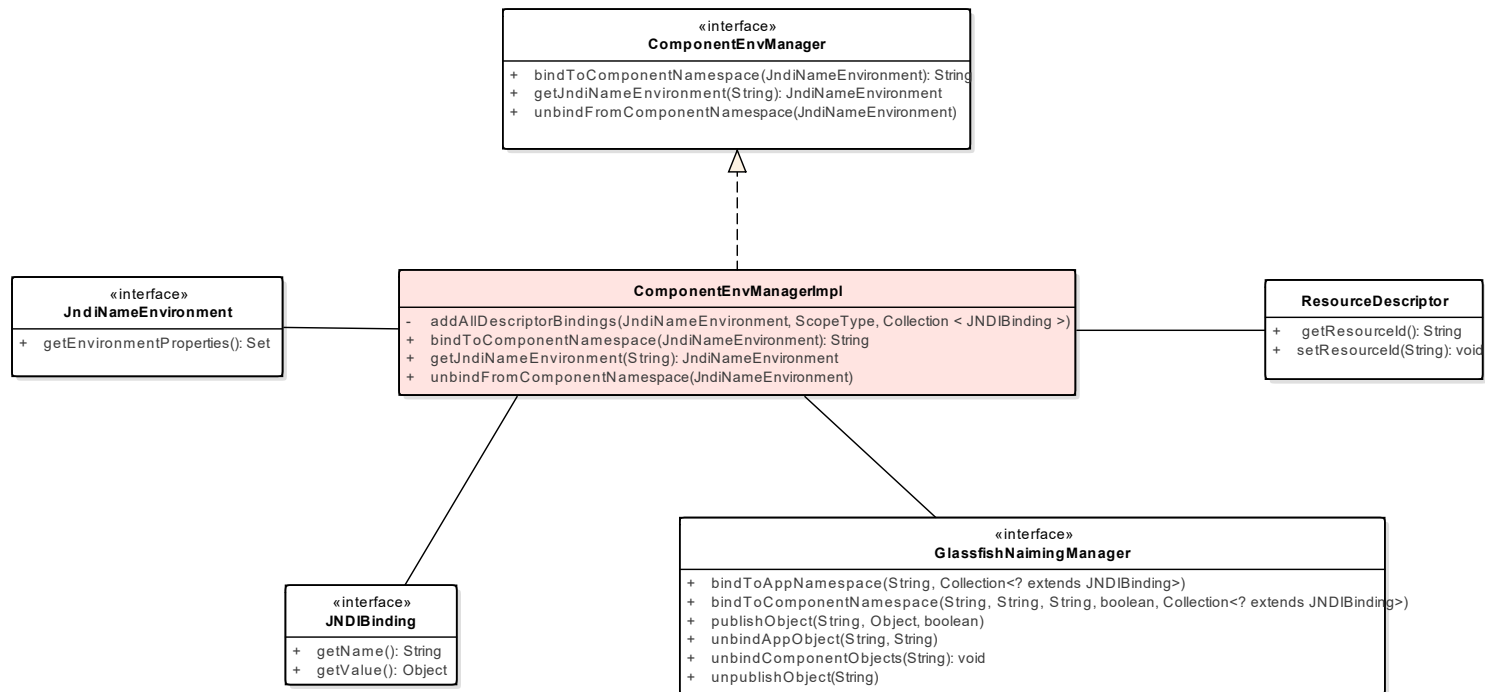


Figure 1: UML Class Diagram of the main methods and dependencies

3.3 Classes

3.3.1 ComponentEnvManagerImpl

The class `ComponentEnvManagerImpl` implements the interface *ComponentEnvManager* that defines its public interface and is in charge of all the main operations concerning the management of the application component's naming environment. In particular it provides methods to bind/unbind a JNDI environment of a component or an application to the JNDI service interacting with the naming manager (these methods will be discussed further later), keeping a local track of those bindings by means of the register/unregister methods. It also provides interface methods to get the JNDI environment associated to one component, the current JNDI environment, the current Application environment and the string name of a JNDI environment. Eventually it allows to modify properties of already bound JNDI environments.

3.4 Methods

3.4.1 getJndiNameEnvironment

The *getJndiNameEnvironment* method returns the JNDI environment associated to the component whose string name is passed as a parameter, if the name of the component is not bound in any JNDI environment the method returns null.

3.4.2 bindToComponentNamespace

The *bindToComponentNamespace* method is in charge of publishing in the JNDI server the name of the JNDI environment passed as parameter. It collects all JNDI bindings associated to the JNDI environment for both component, module and application scope; then according to the type of the JNDI environment (application or application client descriptor) uses the methods provided by the naming manager to bind it and rebind all the JNDI binding collected before within the context of the JNDI environment. Eventually it registers locally the binding. It propagates the possible `NamingException` generated by the naming manager.

3.4.3 addAllDescriptorBindings

The *addAllDescriptorBindings* method is in charge of converting the resource descriptors associated to the JNDI environment within a specific scope (application, component or module) to JNDI bindings. It receives as parameters the JNDI environment, the scope and the collection of JNDI bindings and augments the latter with the newly generated JNDI bindings.

3.4.4 unbindFromComponentNamespace

The *unbindFromComponentNamespace* method plays the opposite role with respect to *bindToComponentNamespace*. It receives as parameter a JNDI environment and it undeploys all resources descriptors associated; then uses the naming manager to unpublish all the JNDI bindings associated to the JNDI environment. Eventually it unregisters locally the binding. It propagates the possible `NamingException` generated by the naming manager.

4 Issues

4.1 Issue list

In the following we present the list of all issues found during the code inspection divided into the categories defined in the checklist. The field `#Issue` is used to number progressively the issues, few numbers may be missing because of successive revisions.

4.1.1 Naming conventions

#Issue	1
<i>Class/Method</i>	ComponentEnvManagerImpl/getJndiNameEnvironment
<i>#Line</i>	161
<i>Code fragment</i>	RefCountJndiNameEnvironment rj
<i>Issue category</i>	Naming conventions
<i>Issue ref</i>	1
<i>Motivation</i>	Method variable name rj non meaningful.
<i>Comment</i>	Variable name should refer to the object referenced.

#Issue	2
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	236
<i>Code fragment</i>	JndiNameEnvironment next
<i>Issue category</i>	Naming conventions
<i>Issue ref</i>	1
<i>Motivation</i>	Method variable name next non corresponding to meaning.
<i>Comment</i>	Variable name should refer to the object referenced.

#Issue	3
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	241, 253
<i>Code fragment</i>	JNDIBinding next
<i>Issue category</i>	Naming conventions
<i>Issue ref</i>	1
<i>Motivation</i>	Method variable name next non corresponding to meaning.
<i>Comment</i>	Variable name should refer to the object referenced.

#Issue	4
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	383, 399
<i>Code fragment</i>	JNDIBinding next
<i>Issue category</i>	Naming conventions
<i>Issue ref</i>	1
<i>Motivation</i>	Method variable name next non corresponding to meaning.
<i>Comment</i>	Variable name should refer to the object referenced.

4.1.2 Indentation

No issues found.

4.1.3 Braces

#Issue	5
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	265
<i>Code fragment</i>	<code>if (_logger.isLoggable(Level.FINEST))</code>
<i>Issue category</i>	Braces
<i>Issue ref</i>	11
<i>Motivation</i>	Avoid using if statements without curly braces.
<i>Comment</i>	

4.1.4 File Organization

#Issue	6
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	71, 85
<i>Code fragment</i>	blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Blank line between groups of imports.
<i>Comment</i>	Acceptable because separates different types of imports.

#Issue	7
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	186, 273, 371, 767, 780, 781, 800, 801, 802, 803, 804, 805
<i>Code fragment</i>	<code>double</code> blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Double blank line between methods.
<i>Comment</i>	Just one line is preferable.

#Issue	8
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	1056
<i>Code fragment</i>	blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Blank line between closing curly brackets.
<i>Comment</i>	No blank line between curly brackets is preferred.

#Issue	9
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	1058
<i>Code fragment</i>	blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Blank line at the end of the file.
<i>Comment</i>	No blank line at the end of the file is preferred.

#Issue	10
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	262
<i>Code fragment</i>	<code>double</code> blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Double blank line within a method.
<i>Comment</i>	Just one line to separate different sections within a method is preferred.

#Issue	11
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	319
<i>Code fragment</i>	no blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Missing blank line between declaration and code.
<i>Comment</i>	One blank line to separate declarations and code is preferred.

#Issue	12
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	417
<i>Code fragment</i>	blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Blank line between closing curly brackets.
<i>Comment</i>	No blank line between curly brackets is preferred.

#Issue	13
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	405
<i>Code fragment</i>	blank line
<i>Issue category</i>	File organization
<i>Issue ref</i>	12
<i>Motivation</i>	Blank line before closing curly bracket
<i>Comment</i>	No blank line between curly brackets is preferred.

#Issue	14
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	213, 216, 219, 242, 256, 258
<i>Code fragment</i>	
<i>Issue category</i>	File organization
<i>Issue ref</i>	13
<i>Motivation</i>	Line exceeds 80 characters.
<i>Comment</i>	In those case line wrap should be possible. ²

<i>#Issue</i>	15
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	390, 402
<i>Code fragment</i>	
<i>Issue category</i>	File organization
<i>Issue ref</i>	13
<i>Motivation</i>	Line exceeds 80 characters
<i>Comment</i>	In those case line wrap should be possible.

<i>#Issue</i>	16
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	290
<i>Code fragment</i>	
<i>Issue category</i>	File organization
<i>Issue ref</i>	13
<i>Motivation</i>	Line exceeds 80 characters
<i>Comment</i>	In those case line wrap should be possible.

<i>#Issue</i>	17
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	312, 362
<i>Code fragment</i>	
<i>Issue category</i>	File organization
<i>Issue ref</i>	14
<i>Motivation</i>	Line exceeds 120 characters
<i>Comment</i>	In those cases line wrap should be possible.

4.1.5 Wrapping Lines

No issues found.

4.1.6 Comments

<i>#Issue</i>	18
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	89
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Class behaviour explanation missing.
<i>Comment</i>	Every class should be provided with a comment explaining its main functionalities.

#Issue	19
<i>Class/Method</i>	ComponentEnvManagerImpl/getJndiNameEnvironment
<i>#Line</i>	160
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Method behaviour explanation missing, the corresponding comment in the interface declaration is inconsistent.
<i>Comment</i>	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	21
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	188
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Method behaviour explanation missing
<i>Comment</i>	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	22
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	200, 204, 225, 251
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Block behaviour explanation missing
<i>Comment</i>	Every significant block should be provided with a comment explaining its main functionalities.

#Issue	23
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	312
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Method behaviour explanation missing
<i>Comment</i>	Every method should be provided with a comment explaining its main functionalities.

#Issue	24
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	319, 336
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Block behaviour explanation missing
<i>Comment</i>	Every significant block should be provided with a comment explaining its main functionalities.

#Issue	25
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	373
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Method behaviour explanation missing
<i>Comment</i>	Every public method should be provided with a comment explaining its main functionalities, preferably in the interface declaration.

#Issue	26
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	395, 409
<i>Code fragment</i>	
<i>Issue category</i>	Comments
<i>Issue ref</i>	18
<i>Motivation</i>	Block behaviour explanation missing
<i>Comment</i>	Every significant block should be provided with a comment explaining its main functionalities.

4.1.7 Java Source Files

#Issue	27
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	88
<i>Code fragment</i>	
<i>Issue category</i>	Java Source Files
<i>Issue ref</i>	23
<i>Motivation</i>	Incomplete javadoc of the class.
<i>Comment</i>	Every class should be documented explaining its role in the whole project.

#Issue	28
<i>Class/Method</i>	ComponentEnvManagerImpl/getJndiNameEnvironment
<i>#Line</i>	160
<i>Code fragment</i>	
<i>Issue category</i>	Java Source Files
<i>Issue ref</i>	23
<i>Motivation</i>	Incomplete javadoc for the method.
<i>Comment</i>	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	29
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	188
<i>Code fragment</i>	
<i>Issue category</i>	Java Source Files
<i>Issue ref</i>	23
<i>Motivation</i>	Incomplete javadoc for the method.
<i>Comment</i>	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	30
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	380
<i>Code fragment</i>	
<i>Issue category</i>	Java Source Files
<i>Issue ref</i>	23
<i>Motivation</i>	Incomplete javadoc for the method.
<i>Comment</i>	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

#Issue	31
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	312
<i>Code fragment</i>	
<i>Issue category</i>	Java Source Files
<i>Issue ref</i>	23
<i>Motivation</i>	Javadoc missing for the method.
<i>Comment</i>	Every method should be documented explaining the role of its parameters, the returned value and its functionality.

4.1.8 Package and Import Statements

No issues found.

4.1.9 Class and interface declaration

#Issue	34
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	100
<i>Code fragment</i>	<code>private ServiceLocator locator</code>
<i>Issue category</i>	Class and interface declaration
<i>Issue ref</i>	25, d, d
<i>Motivation</i>	Private instance variable before package level variable.
<i>Comment</i>	Private instance variables should appear as last.

#Issue	35
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	103
<i>Code fragment</i>	<code>private</code> Logger _logger
<i>Issue category</i>	Class and interface declaration
<i>Issue ref</i>	25, d, d
<i>Motivation</i>	Private instance variable before package level variable.
<i>Comment</i>	Private instance variables should appear as last.

#Issue	36
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	188
<i>Code fragment</i>	
<i>Issue category</i>	Class and interface declaration
<i>Issue ref</i>	26
<i>Motivation</i>	Methods are not presented in the order in which they are declared in the interface.
<i>Comment</i>	Methods implementation should appear in the same order in which they are defined in the corresponding interface.

#Issue	37
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	188
<i>Code fragment</i>	
<i>Issue category</i>	Class and interface declaration
<i>Issue ref</i>	27
<i>Motivation</i>	Method is too long, the Cyclomatic Complexity of this method is 14.
<i>Comment</i>	Method should not be too much long, maximum Cyclomatic Complexity is 10.

#Issue	38
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	221
<i>Code fragment</i>	<code>compEnvId = getComponentEnvId(env)</code>
<i>Issue category</i>	Class and interface declaration
<i>Issue ref</i>	27
<i>Motivation</i>	Duplicated code at line 191.
<i>Comment</i>	Since variable compEnvId is not modified in between, this declaration is useless.

4.1.10 Initialization and declarations

#Issue	39
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	106
<i>Code fragment</i>	GlassfishNamingManager namingManager
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	28
<i>Motivation</i>	Friendly instance variable not used by same package classes, should be private.
<i>Comment</i>	Keeping the lowest visibility as possible is preferred for information hiding.

#Issue	40
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	109
<i>Code fragment</i>	ComponentNamingUtil componentNamingUtil
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	28
<i>Motivation</i>	Friendly instance variable not used by same package classes, should be private.
<i>Comment</i>	Keeping the lowest visibility as possible is preferred for information hiding.

#Issue	41
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	112
<i>Code fragment</i>	<code>transient private</code> CallFlowAgent callFlowAgent
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	29
<i>Motivation</i>	Instance variable used only at line 820, could be declared as static field of the inner class FactoryForEntityManagerWrapper.
<i>Comment</i>	Keeping the smallest scope as possible is preferred. for information hiding.

#Issue	42
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	115
<i>Code fragment</i>	<code>transient private</code> TransactionManager txManager
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	29
<i>Motivation</i>	Instance variable used only at line 820, could be declared as static field of the inner class FactoryForEntityManagerWrapper
<i>Comment</i>	Keeping the smallest scope as possible is preferred. for information hiding.

#Issue	43
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	250
<i>Code fragment</i>	Application app = DOLUtils.getApplicationFromEnv(env)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks.
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	44
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	348
<i>Code fragment</i>	String resourceId = getResourceId(env, descriptor)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks.
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	45
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	351
<i>Code fragment</i>	CommonResourceProxy proxy = locator.getService(CommonResourceProxy.class)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks.
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	46
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	354
<i>Code fragment</i>	String logicalJndiName = descriptorToLogicalJndiName(descriptor)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks.
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	47
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	355
<i>Code fragment</i>	CompEnvBinding envBinding = new CompEnvBinding(logicalJndiName, proxy)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	48
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	362
<i>Code fragment</i>	CompEnvBinding jmscfEnvBinding = new CompEnvBinding(ConnectorsUtil.getPMJndiName(logicalJndiName), jmscfProxy)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	49
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	380
<i>Code fragment</i>	Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	50
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	387
<i>Code fragment</i>	Application app = DOLUtils.getApplicationFromEnv(env)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

#Issue	51
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	390
<i>Code fragment</i>	Set<ApplicationClientDescriptor> appClientDescs = app.getBundleDescriptors(ApplicationClientDescriptor.class)
<i>Issue category</i>	Initialization and Declarations
<i>Issue ref</i>	33
<i>Motivation</i>	Declarations do not appear at the beginning of blocks
<i>Comment</i>	Variable declaration at the beginning of a block is preferred for readability.

4.1.11 Method calls

No issues found.

4.1.12 Arrays

No issues found.

4.1.13 Object comparison

No issues found.

4.1.14 Output format

No issues found.

4.1.15 Computation, Comparitions and Assignments

<i>#Issue</i>	52
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	342
<i>Code fragment</i>	<code>if(descriptor.getResourceType().equals(DSD)){if (((DataSourceDefinitionDescriptor)descriptor).isDeployed())</code>
<i>Issue category</i>	Computation, Comparitions and Assignments
<i>Issue ref</i>	45
<i>Motivation</i>	Enclosed if should be merged.
<i>Comment</i>	If not needed netsed if should be merged in one if with the congiunction of the conditions.

<i>#Issue</i>	53
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	395
<i>Code fragment</i>	<code>if(!(env instanceof ApplicationClientDescriptor)&& (app.getBundleDescriptors(ApplicationClientDescriptor.class).size())> 0))</code>
<i>Issue category</i>	Computation, Comparitions and Assignments
<i>Issue ref</i>	46
<i>Motivation</i>	Useless parenthesis around second operand.
<i>Comment</i>	Non necessary parenthesis are acceptable only to highlight precedence.

4.1.16 Exceptions

No issues found.

4.1.17 Flow of control

No issues found.

4.1.18 Files

No issues found.

4.2 Cumulative data

In this subsection we present some aggregate data derived from the process of code inspection that could be useful to understand the distribution of issues in category and in methods. We will use both tables and diagrams.

4.2.1 Issues per category and class/method - table

Class/Method - Issue category	SLOC	Naming conventions	Indentation	Braces	File organization	Wrapping lines	Comments	Java Source Files	Package and import statements	Class and interface declaration	Initialization and declarations	Method calls	Arrays	Object comparison	Output format	Computation, Comparisons and Assignments	Exceptions	Flow of control	Files	Total for class/method	Total per class/method per KSLOC
ComponentEnvManagerImpl	121	0	0	0	15	0	1	0	0	2	2	0	0	0	0	0	0	0	0	20	165,3
ComponentEnvManagerImpl/ getJndiNameEnvironment	9	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2	222,2
ComponentEnvManagerImpl/ addAllDescriptorBindings	55	0	0	0	4	0	3	1	0	0	1	0	0	0	0	1	0	0	0	10	181,8
ComponentEnvManagerImpl/ unbindFromComponentNames- pace	46	2	0	0	4	0	3	0	0	0	3	0	0	0	0	1	0	0	0	13	282,6
ComponentEnvManagerImpl/ bindToComponentNamespace	85	3	0	1	7	0	5	0	0	2	1	0	0	0	0	0	0	0	0	19	223,5
Total per category	316	6	0	1	30	0	13	1	0	4	7	0	0	0	0	2	0	0	0	64	215,1
Total per category (%)		9,38	0	1,6	46,9	0	20,3	1,6	0	6,3	10,9	0	0	0	0	3,1	0	0	0	100	

4.2.2 Issues per category - hystogram

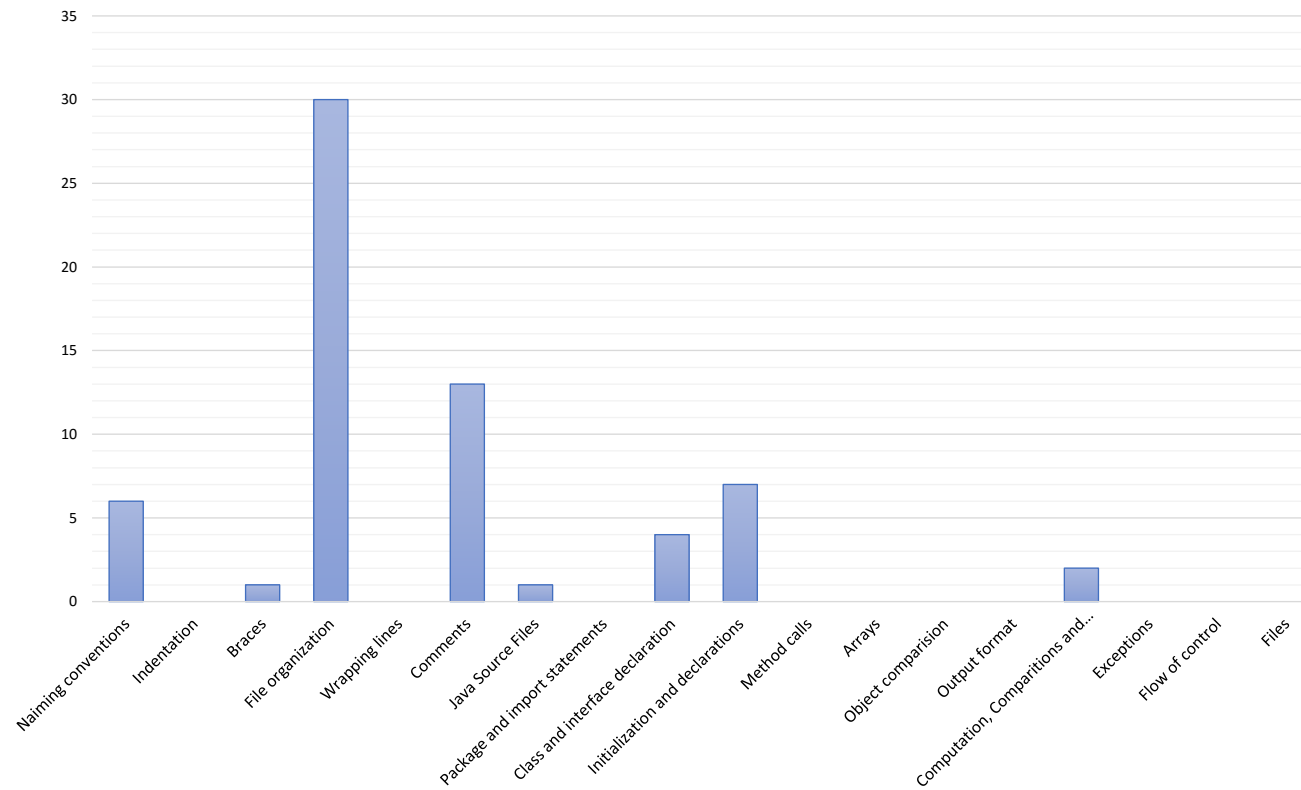


Figure 2: Hystogram showing the distribution of issues in categories.

4.2.3 Issues per class/method per KSLOC - hystogram

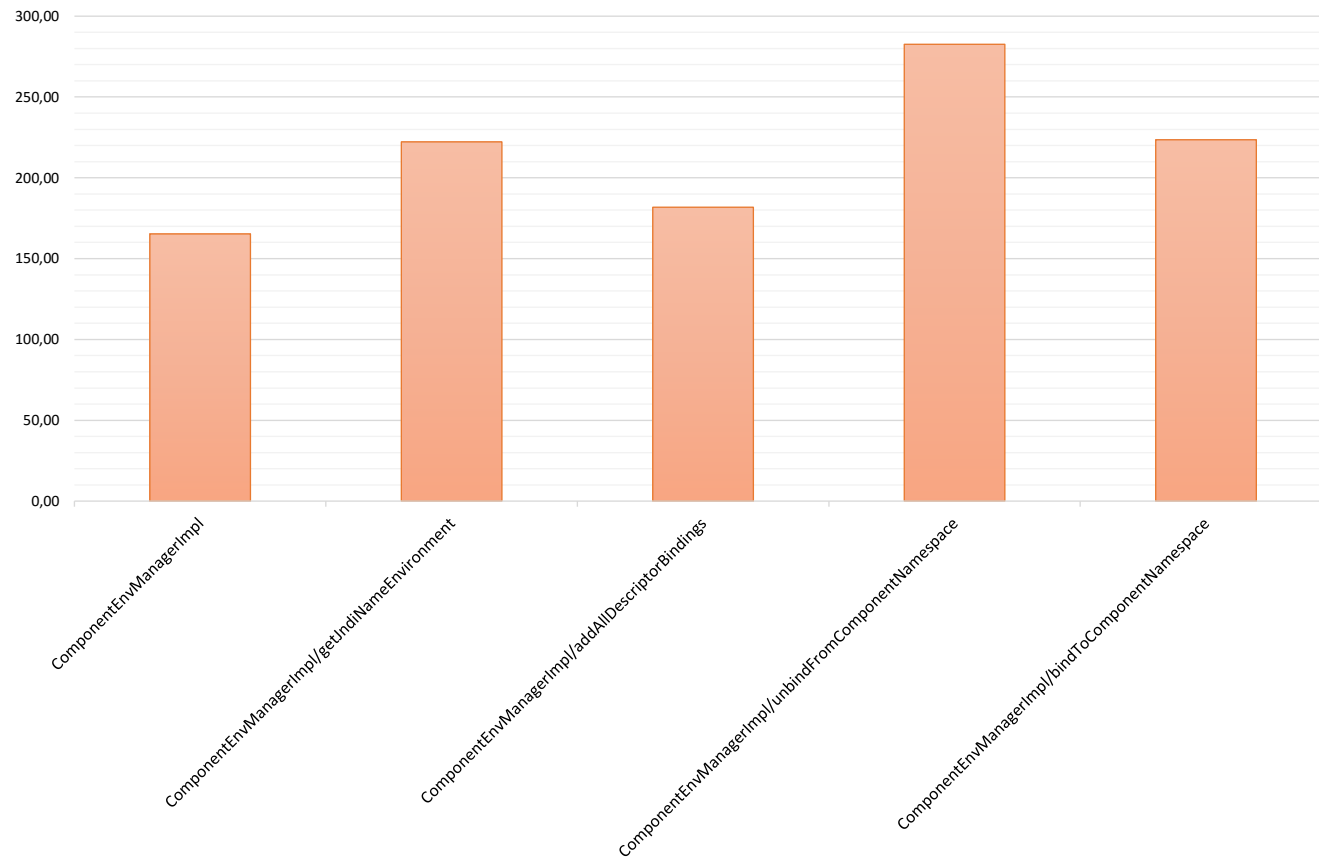


Figure 3: Hystogram showing the distribution of issues in method/class.

4.2.4 Issues per class/method per KSLOC - pie chart

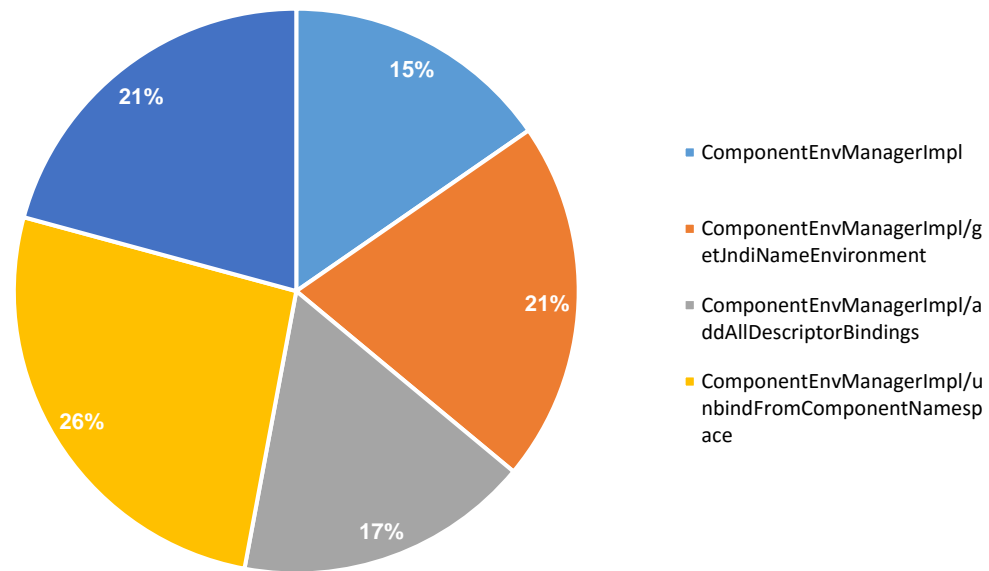


Figure 4: Pie chart showing the distribution of issues in method/class.

5 Other problems

In this section we expose some issues, not contained in the checklist, highlighted by other automatic tools for code revision or identified during the manual revision that, in our opinion, can lead to bugs.

5.1 Automatic code review

We don't present all issues reported by the tools, since many of them has already been identified during the manual inspection driven by the checklist and some of them we think are not appropriate (for instance, PMD reports all variables local to methods to be declared final if not modifies, we think this is exaggerated while this policy should be adopted for instance variable of classes.

#Issue	1
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	61
<i>Code fragment</i>	<code>import org.glassfish.hk2.api.ActiveDescriptor</code>
<i>Issuer</i>	FindBugs
<i>Message</i>	Unused import.
<i>Comment</i>	

#Issue	2
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	63
<i>Code fragment</i>	<code>import org.glassfish.hk2.utilities.BuilderHelper</code>
<i>Issuer</i>	FindBugs
<i>Message</i>	Unused import.
<i>Comment</i>	

#Issue	3
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	103
<i>Code fragment</i>	<code>private Logger _logger</code>
<i>Issuer</i>	FindBugs, PMD
<i>Message</i>	The logger declaration field _logger should be static and final.
<i>Comment</i>	

#Issue	4
<i>Class/Method</i>	ComponentEnvManagerImpl
<i>#Line</i>	128
<i>Code fragment</i>	<code>private ConcurrentMap<String, RefCountJndiNameEnvironment> compId2Env = new ConcurrentHashMap<String, RefCountJndiNameEnvironment>()</code>
<i>Issuer</i>	FindBugs
<i>Message</i>	Field compId2Env should be final.
<i>Comment</i>	Instance variable non modified after instansiation should be declared final.

#Issue	5
Class/Method	ComponentEnvManagerImpl
#Line	129
Code fragment	<code>private ConcurrentMap<String, RefCountJndiNameEnvironment> compId2Env = new ConcurrentHashMap<String, RefCountJndiNameEnvironment>()</code>
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad).
Comment	

#Issue	6
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	159
Code fragment	<code>public JndiNameEnvironment getJndiNameEnvironment(String componentId)</code>
Issuer	FindBugs
Message	Add @Override annotation.
Comment	All methods inherited by superclasses or interfaces should be marked with @Override annotation.

#Issue	7
Class/Method	ComponentEnvManagerImpl/getJndiNameEnvironment
#Line	163
Code fragment	<code>_logger.finest("ComponentEnvManagerImpl: "+ "getCurrentJndiNameEnvironment "+ inv.componentId + "is "+ desc.getClass())</code>
Issuer	FindBugs
Message	Inefficient use of string concatenation in logger.
Comment	A unique string should be used instad of concatenation.

#Issue	8
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	188
Code fragment	<code>public String bindToComponentNamespace(JndiNameEnvironment env)</code>
Issuer	FindBugs
Message	Add @Override annotation
Comment	All methods inherited by superclasses or interfaces should be marked with @Override annotation.

#Issue	9
Class/Method	ComponentEnvManagerImpl/bindToComponentNamespace
#Line	193
Code fragment	<code>Collection<JNDIBinding> bindings = new ArrayList<JNDIBinding>()</code>
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instad).
Comment	

#Issue	10
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	228
<i>Code fragment</i>	Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()
<i>Issuer</i>	FindBugs
<i>Message</i>	Redundant type arguments in a new expression (use diamond operator instead).
<i>Comment</i>	

#Issue	11
<i>Class/Method</i>	ComponentEnvManagerImpl/bindToComponentNamespace
<i>#Line</i>	266
<i>Code fragment</i>	_logger.finest("ComponentEnvManagerImpl: "+ "register "+ compEnvId + "is "+ env.getClass())
<i>Issuer</i>	FindBugs
<i>Message</i>	Inefficient use of string concatenation in logger
<i>Comment</i>	A unique string should be used instead of concatenation.

#Issue	12
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	314
<i>Code fragment</i>	Set<ResourceDescriptor> allDescriptors = new HashSet<ResourceDescriptor>()
<i>Issuer</i>	FindBugs
<i>Message</i>	Redundant type arguments in a new expression (use diamond operator instead)
<i>Comment</i>	

#Issue	13
<i>Class/Method</i>	ComponentEnvManagerImpl/addAllDescriptorBindings
<i>#Line</i>	343
<i>Code fragment</i>	((DataSourceDefinitionDescriptor)descriptor)
<i>Issuer</i>	FindBugs
<i>Message</i>	Unchecked/unconfirmed cast of return value from method.
<i>Comment</i>	Casting an instance of superclass to an instance of subclass, not necessarily wrong but maybe deserved an explicative comment.

#Issue	14
<i>Class/Method</i>	ComponentEnvManagerImpl/unbindFromComponentNamespace
<i>#Line</i>	373
<i>Code fragment</i>	public void unbindFromComponentNamespace(JndiNameEnvironment env)
<i>Issuer</i>	FindBugs
<i>Message</i>	Add @Override annotation
<i>Comment</i>	All methods inherited by superclasses or interfaces should be marked with @Override annotation.

#Issue	15
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	380
Code fragment	Collection<JNDIBinding> globalBindings = new ArrayList<JNDIBinding>()
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instead)
Comment	

#Issue	16
Class/Method	ComponentEnvManagerImpl/unbindFromComponentNamespace
#Line	397
Code fragment	Collection<JNDIBinding> appBindings = new ArrayList<JNDIBinding>()
Issuer	FindBugs
Message	Redundant type arguments in a new expression (use diamond operator instead)
Comment	

#Issue	17
Class/Method	ComponentEnvManagerImpl
#Line	112
Code fragment	transient private CallFlowAgent callFlowAgent
Issuer	SonarQube
Message	Reorder the modifiers to comply with the Java Language Specification.
Comment	private should precede transient.

#Issue	18
Class/Method	ComponentEnvManagerImpl
#Line	115
Code fragment	transient private TransactionManager txManager
Issuer	SonarQube
Message	Reorder the modifiers to comply with the Java Language Specification.
Comment	private should precede transient.

#Issue	19
Class/Method	ComponentEnvManagerImpl
#Line	120
Code fragment	// TODO: container-common shouldn't depend on EJB stuff, right?
Issuer	SonarQube
Message	Complete the task associated to this TODO comment.
Comment	

#Issue	20
Class/Method	ComponentEnvManagerImpl
#Line	89
Code fragment	
Issuer	PMD
Message	The class 'ComponentEnvManagerImpl' has a Cydomatic Complexity of 6 (Highest = 16). Rule: CydomaticComplexity Rule set Code Size
Comment	

#Issue	21
Class/Method	ComponentEnvManagerImpl
#Line	90
Code fragment	
Issuer	PMD
Message	This class has too many methods, consider refactoring it.
Comment	

#Issue	22
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	319
Code fragment	<code>if(!(env instanceof ApplicationClientDescriptor)){ [...] } else { [...] }</code>
Issuer	PMD
Message	Avoid if (x != y) .. else .. ConfusingTernary
Comment	It is preferable to use if(x==y) .. else ..

#Issue	23
Class/Method	ComponentEnvManagerImpl/addAllDescriptorBindings
#Line	325
Code fragment	<code>if(!(env instanceof ApplicationClientDescriptor)){ [...] } else { [...] }</code>
Issuer	PMD
Message	Avoid if (x != y) .. else .. ConfusingTernary
Comment	It is preferable to use if(x==y) .. else ..

5.2 Other issues from manual inspection

1. The usage of continue statements should be avoided since it modifies the flow of execution in a brutal way, making maintainance harder since the code following the continue statement seems to be non conditioned.

```

for (ResourceDescriptor descriptor : allDescriptors) {

    if (!dependencyAppliesToScope(descriptor, scope)) {
        continue;
    }

    if(descriptor.getResourceType().equals(DSD)) {
        if
            (((DataSourceDefinitionDescriptor)descriptor).isDeployed())
            {

```

```

        continue;
    }
}

[...]
```

The code above should be rephrased as follows

```

for (ResourceDescriptor descriptor : allDescriptors) {

    if (dependencyAppliesToScope(descriptor, scope) &&
        !(descriptor.getResourceType().equals(DSD) &&
            ((DataSourceDefinitionDescriptor)descriptor).isDeployed())
        ) {

        [...]

    }

}
```

2. This inner class, even if it does not belong to the fragment of code we were assigned, is good example of “conscientious” usage of public variables. The class `RefCountJndiNameEnvironment` is a private inner class of `ComponentEnvManagerImpl` which is used exactly as a struct in C language. Its instance variable are public, but since the class is a private inner class its visibility is restricted to the outer class, therefore having those public variable does not break encapsulation. We must report, however, that since the variable `env` is never modified after creation it should be declared `final`.

```

private static class RefCountJndiNameEnvironment {
    public RefCountJndiNameEnvironment(JndiNameEnvironment
        env) {
        this.env = env;
        this.refcnt = new AtomicInteger(1);
    }
    public JndiNameEnvironment env;
    public AtomicInteger refcnt;
}
```

3. There is no coherent useage of spacing at the beginning and at the end of the if conditions. All styles are valid but it's preferable to use always the same.

```

if (componentId != null && _logger.isLoggable(Level.FINEST))
if( env instanceof Application)
if( env instanceof Application )
if (wsRefMgr != null )
```


A Appendix

Used tools

1. L^AT_EX visual editor for L^AT_EX (<http://www.lyx.org/>) to write this document.
2. Enterprise Architect 11 (<http://www.sparxsystems.com.au/products/ea/>) for UML diagrams.
3. NetBeans IDE 8.1 (<https://netbeans.org/>) for code inspection.
4. The automatic analysis tools listed in section 1.5.

Hours of works

Time spent by each group member:

- Alberto Maria Metelli: 18 h
- Riccardo Mologni: h

Revision history

<i>Version</i>	<i>Date</i>	<i>Revision description</i>	<i>Revision notes</i>
0.1	1-1-2016	Initial draft	-
1.0	5-1-2016	Final draft	-