Politecnico di Milano

A.A. 2015-2016

Software Engineering 2

Code inspection

Version 0.1

Ivana Salerno

Alexis Rougnant

Daniel Vacca

January 5th 2016

Table of content

Table of Content**¡Error! Marcador no definido.**

1. Introduction3
2. Functional role of assigned set of classes3
3. Issues8
   - 3.1. Issues found by applying the checklist8
     - 3.1.1. Naming conventions10
     - 3.1.2. Indentation11
     - 3.1.3. Braces11
     - 3.1.4. File Organization11
     - 3.1.5. Wrapping Lines12
     - 3.1.6. Comments12
     - 3.1.7. Java Source Files13
     - 3.1.8. Package and Import Statements13
     - 3.1.9. Class and Interface Declarations13
     - 3.1.10. Initialization and Declarations14
     - 3.1.11. Method Calls14
     - 3.1.12. Arrays15
     - 3.1.13. Object Comparison15
     - 3.1.14. Output Format15
     - 3.1.15. Comparisons and Assignments16
     - 3.1.16. Exceptions16
     - 3.1.17. Flow of Control17
     - 3.1.18. Files17
   - 3.2. Other problems17

1. Introduction

The present document is intended to show the result of the inspection of some fragments of the Glassfish source code. Such inspection included an analysis of the functional role of the code fragment, and the verification of several syntactic issues by using a checklist.

In the release 4.1 of Glassfish, inside the namespace appserver/common/container-common/src/main/java/com/sun/enterprise/container/common/impl, we were assigned the following four methods which belong to the class *ComponentEnvManagerImpl*:

- addJNDIBindings, in line 557
- getCompEnvBinding, in line 686
- dependencyAppliesToScope, in line 744
- create, in line 874

2. Functional role of assigned set of classes

In this section we provide a description of the purpose of the methods that were assigned to us. In order to do so, we had to explore as well in the functional role of the class as a whole.

The following terms are recurrently used, and it is worth it to have precise definitions of them:

- **Binding:** is a pair *<name, object>* that associates a name with an object.
- **Component:** self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.
- **Dependency:** any component to which a specific component will communicate in order to work.
- **Environment:** set of all the dependencies of a component.
- **Naming context:** directory in the naming server where all the dependencies from a naming environment are stored.
- **Naming environment:** entity that names the dependencies in the environment of a component in order to manage them.
- **Naming service:** service that provides services to register, unregister and lookup for objects given their names.

According to Oracle's web site: "A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components."[1] Those other components are called *dependencies*, and the set of all the dependencies of a component is called its *environment*.

As part of the flexibility that Glassfish containers (in which components are deployed) provide to the developers of applications, the environment of a component can be customized without

[1] https://docs.oracle.com/javaee/7/tutorial/overview003.htm#BNABB

modifying the source code of the related component. Such capability is possible through the abstraction of the environment into an entity that manages the dependencies by naming them, which is implemented by the container; this new entity is a *naming environment*. Whenever a component needs to access its naming environment, the container provides such access in the form of a *naming context*. The naming environment uses a *naming service* to register the dependencies and allow them to be looked up.[2] Not all the dependencies might be reachable from any other component; some could be globally usable, and some others only locally. In order to prevent access conflicts, scopes are associated to dependencies to indicate their reachability.

In particular, Glassfish implements a JNDI server to support the naming service. Thus the provided context is actually a *JNDI naming context*, and the environment is a *JNDI Naming Environment*.

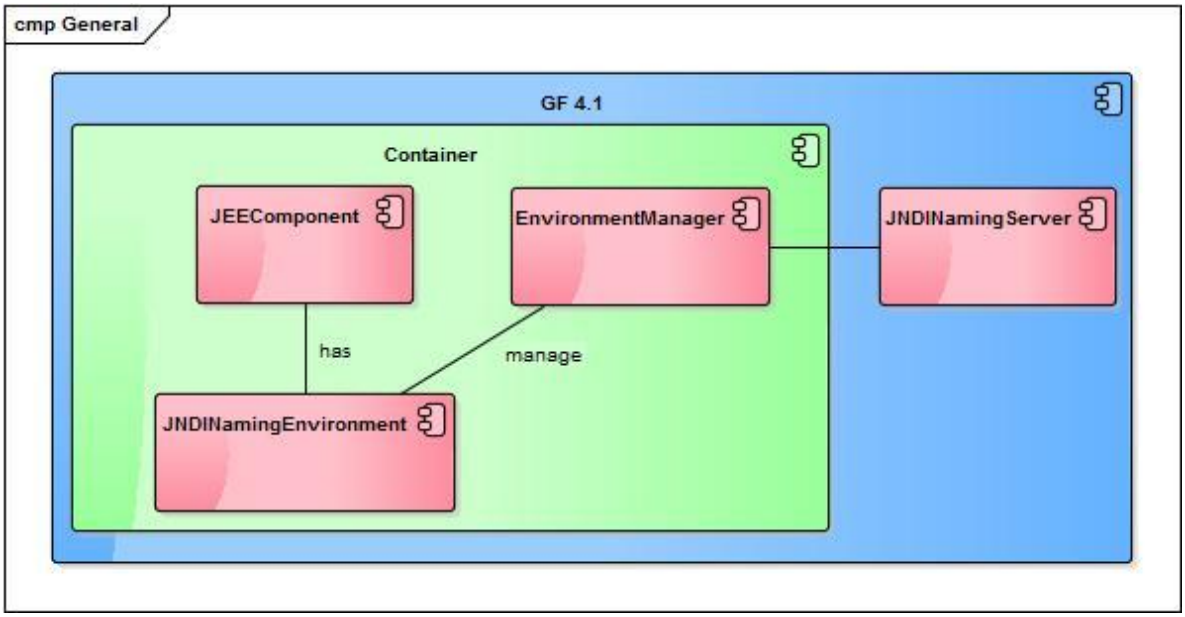

Figure 1: Environment manager in Glassfish

The class *ComponentEnvManagerImpl* is responsible for managing JNDI naming environments; these are represented with the class *JNDINameEnvironment*. In order to give a more meaningful explanation of the four assigned methods, we will roughly describe the most important methods. Most of the following information concerning the ComponentEnvManagerImpl was obtained by directly inspecting the source code since the documentation is almost empty; comments inside the code were also helpful.

As suggested by its name, and evident in the source code, ComponentEnvManagerImpl is an implementation of the interface *ComponentEnvManager*. We will explain the functional role of the class in terms of the operations that such interface provides:

---

2 https://docs.oracle.com/javaee/7/tutorial/overview008.htm#GIRDR

- Get a naming environment given its ID. This method receives the ID of a JNDINamingEnvironment and returns the corresponding instance, or null if the name is not found.
- Get the naming environment on which the current invocation (from the container) is being executed. This method returns the corresponding instance, or null if no naming environment is being invoked.
- Get the ID of a naming environment given the instance. This method returns the unique ID of the received instance of JNDINameEnvironment.
- Bind a given naming environment of a component to its namespace. This method receives a JNDINameEnviroment and returns the ID of the component, after having bound all the dependencies; it might throw a *NamingException*.
- Add dependencies to the namespace of a component, given its naming environment. This method receives a JNDINameEnvironment, and collections of *EnvironmentProperty* and *ResourceReferenceDescriptor*, and adds these to the namespace of the received naming environment; it might throw a NamingException.
- Unbind a given naming environment of a component from its namespace. This method receives a JNDINameEnvironment and unbinds all the dependencies; it might throw a NamingException.
- Get the current application environment. This method returns the current application environment (as an *ApplicationEnvironment*) if not running in a specified container.[3]

Besides the previous operations, the class ComponentEnvManagerImpl has two additional public methods. These methods concern the maintenance of the instances of the naming environments (only the instance, without the actual dependencies):

- Register a naming environment, given the corresponding instance and its ID. It adds the instance to a map that links the ID with it.
- Unregister a naming environment, given the ID of the naming environment. Receives the ID and decreases a counter related to the instance in the map; if this gets to zero, the entry is deleted.

The methods bindToComponentNamespace, addToComponentNamespace and unbindFromComponentNamespace use the method addJNDIBindings() to get the bindings of the dependencies associated to each scope. Such bindings are used either to be published or unpublished.

---

[3] http://glassfish.pompel.me/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.html

After this overall explanation of the class as a whole, we are able to expose the functional role of the four methods that were assigned to us:

- õ private void addJNDIBindings(JndiNameEnvironment env, ScopeType scope, Collection<JNDIBinding> jndiBindings):
  This method has the responsibility of creating and adding the bindings for the dependencies in the naming environment, provided that they apply to the given scope.
  The *env* parameter represents the naming environment to be explored; the *scope* is the scope type for which the method will add the bindings; the *jndiBindings* is the collection of the bindings to which the created ones will be added. A binding is a pair *<name, object>* that will be used by the name manager to publish the dependency.

  The method will go through each set of possible dependencies and will create the corresponding binding, if it applies to the scope. Some of the bindings are created directly in the method and some other created by additional invoked methods. Every binding is created with a name obtained by the descriptorToLogicalJndiName method, but the bound object is not the actual dependency obtained from the environment; instead, an appropriate implementation of NamingObjectProxy is used. Some of this implementations are inner classes.

  Two of the assigned methods are used here:
  - getCompEnvBinding is called to create the bindings for some of the dependencies.
  - dependencyAppliesToScope is called to check whether a dependency applies to a scope.
- õ private CompEnvBinding getCompEnvBinding(final ResourceEnvReferenceDescriptor next):
  This method is in charge of creating the appropriate binding for the received (dependency) descriptor.
  The *next* parameter represents the dependency for which the CompEnvBinding object will be created.
  The method creates the name of the dependency through descriptorToLogicalJndiName. A series of conditions are checked in order to instantiate the appropriate proxy object to be bound; if no specific case is matched, a generic one is used. In any case, an object of an implementation of the interface NamingObjectProxy is created.
  With the name and the object, the instance of ComEnvBinding to be returned is created.
- õ private boolean dependencyAppliesToScope(String name, ScopeType scope):
  This method checks if a dependency applies to a scope, given the name of the dependency and the scope.
  The *name* parameter represents the name of the dependency, and the *scope* parameter represents the scope.
  The method compares the prefix of the name with the scope and determines in this way the applicability of the dependency to the scope.
- õ public Object create(Context ctx) throws NamingException:
  This method corresponds to the specific implementation that the inner class *ValidatorProxy* gives for the interface NamingObjectProxy. It is intended to create and return an instance of

Validator (the “proxied” object), based on the information of the context *ct;* it might throw a NamingExeption.

The class ValidatorProxy is the proxy object that represents a Validator. Whenever a components need the actual instance, this method is invoked in order to get it. Such creation is supported by the attributes *validator* and *validatorFactory*, and the received *ct*.

This method is not used in the class ComponentEnvManagerImpl, but it will be probably done when any component looks up for this dependency in the naming manager.

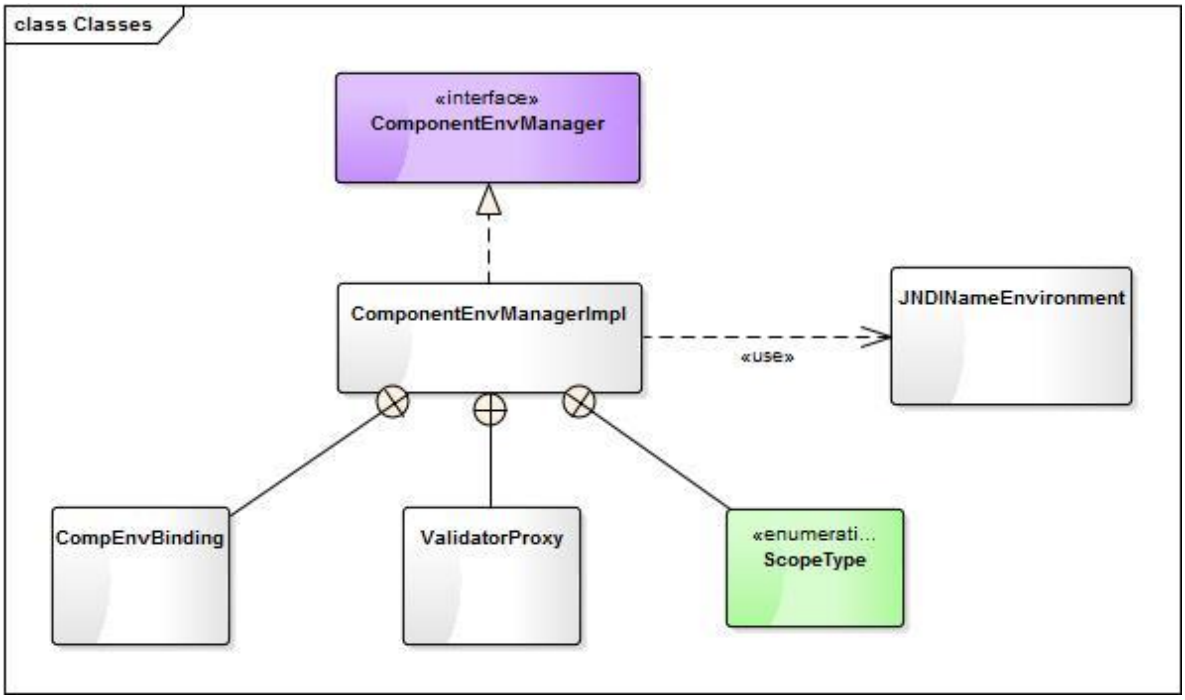

Figure 2: ComponentEnvManagerImpl class diagram

3. Issues

In this section we describe the issues found in the assigned methods, supported by the checklist that was provided. Additional potential problems that were found during the use of the list are enumerated later in this section.

3.1. Issues found by applying the checklist

The problems found by directly using the checklist are pointed out here. They are separated by subsection and by method. Some of the points in the list refer to aspects that belong to the entire class and not only to the specific methods. The issues found in such points are:

Point 5:

Methods dependencyAppliesToScope and descriptorToLogicalJndiName in lines 736, 744 and 774 have verb-less names.

Point 6:

In line 940, wsRedMgr should be written wSRefMgr, wSrvRefMgr or webSrvRefMgr.

In line 143, refcnt should be written refCnt.

Point 22:

Three different interfaces are used in this file. They are implemented consistently with the following classes:

- Interface ComponentEnvMangager; class ComponentEnvManagerImpl.
- Interface JNDIBinding; class CompEnvBinding.
- Interface NamingObjectProxy; classEjbReferenceProxy, WebServiceRefProxy, ValidatorFactoryProxy, ValidatorProxy, EjbContextProxy.

NamingObjectProxy is also implemented with the class *FactoryForEntityManagerWrapper* but the method *create* doesn't throw any exception in this class whereas it has to do it according to the javadoc of the interface.

Point 23:

The Java documentation available is not complete. Only a few methods are briefly described (getComponentEnvId and getCurrentApplicationEnvironment). Meaningful documentation is absent in both the interface and implementing class.

Point 25:

Neither the big class nor the inner classes have documentation comments to describe them, except for *RefCountJndiNameEnvironment*.

In lines 106 and 109, the attribute have package visibility but are declared after private attributes.

In line 137, the class presents the constructor first and then the attributes.

Point 26:

In general, there is no evident functional ordering of the methods in the class. They seem to be placed as they were created or needed. A possible organization could be:

1) getJndiNameEnvironment; getCurrentJndiNameEnvironment
2) bindToComponentNamespace; addToComponentNamespace; addEnvironmentProperties; addResourceReferences; addJNDIBindings; addAllDescriptorBindings
3) unbindFromComponentNamespace; undeployAllDescriptors; undepoyResource
4) getResourceId; getResourceDeployer; getCompEnvBinding; getCompEnvBinding; descriptorToLogicalJndiName; getComponentEnvId; getCurrentApplicationEnvironment
5) dependencyAppliesToScope; dependencyAppliesToScope

Point 27:

The following table presents some metrics related to the assigned class (obtained through Netbeans, Source Code Metrics[4] and LocMetrics[5]):

	Class	**Average in package**	**Maximum in package**	**Minimum in package**
Amount of methods	22	13,7	64	1
Lines of code (without comments and blank lines)	721	209,9	948	12

Based on this, we can say that the class is big with respect to the package where it is implemented. The amount of methods is almost the double of the average but is far away

[4] http://plugins.netbeans.org/plugin/42970/sourcecodemetrics

[5] http://www.locmetrics.com/

from the maximum; the lines of code are also much higher than the average and are closer to the maximum.

The average method size is 28 lines. Only 6 of the 22 methods have a size over this average. The method addJNDIBindings has 128 lines, what makes it a long method.

Class is also too coupled, it refers 51 other classes.

Point 28:

In lines 106 and 109, attributes have package visibility instead of the usual private one, and there are no comments that explain such decision. The visibility might be erroneous.

In lines 142 and 143, attributes have public visibility instead of the usual private one, and there are no comments that explain such decision. The visibility can be erroneous even if such attributes can be only accessible from the class ComponentEnvManagerImpl, since other inner classes might access them directly.

Point 44:

The constants in lines 92 to 97 could be modeled as an enumeration, such that each one has associated the String prefix they refer to.

3.1.1. Naming conventions

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

Point 1: In line 714, the variable *ic* should be more explicit, like *ctx*.

Method dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.2. Indentation

Method addJNDIBindings:

Point 8: there is one space in excess before '}' at line 618. The block of code which starts at line 670 should go under the previous curly brace. There is one missing space before code at line 680 and line 683.

Method getCompEnvBinding:

No issues found.

Method dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.3. Braces

Method addJNDIBindings:

Point 10: in line 663 there is an extra brace (in the commented out code).

Method getCompEnvBinding:

No issues found.

Method dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.4. File Organization

Method addJNDIBindings:

Point 12: in line 618, maybe too many blank lines.

Point 13: only for calling methods.

Method getCompEnvBinding:

Point 13: lines 696, 700, 702 and 709 should have been broken after a comma/operator. Line 686 is impossible to break.

Method dependencyAppliesToScope:

Point 12: in line 745, comment is not separated from the code with a blank line. In line 752, blank line could be used to separate cases.

Method create:

No issues found.

3.1.5. Wrapping Lines

Method addJNDIBindings:

Point 16: in line 593, higher-level breaks not used.

Method getCompEnvBinding:

Point 17: in line 688, indentation not justified. In line 692, missing an indentation.

Method dependencyAppliesToScope:

No issues found.

Method create:

Point 15: line 874 is short enough to not have been broken.

3.1.6. Comments

Method addJNDIBindings:

Point 19: in line 637 is not specified the reason and there is not a date.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

Point 18: in line 744, the method (as the class and most of the other methods) lacks of comments that explain what they are doing.

Method create:

Point 18: in line 883, comment is not very clear.

3.1.7. Java Source Files

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.8. Package and Import Statements

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.9. Class and Interface Declarations

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.10. Initialization and Declarations

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.11. Method Calls

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.12. Arrays

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.13. Object Comparison

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.14. Output Format

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.15. Comparisons and Assignments

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

Point 44: A switch statement should be used in place of several if/else statements.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.16. Exceptions

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.17. Flow of Control

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.1.18. Files

Method addJNDIBindings:

No issues found.

Method getCompEnvBinding:

No issues found.

Method: dependencyAppliesToScope:

No issues found.

Method create:

No issues found.

3.2. Other problems

In this section we list some additional issues that were discovered while inspecting the code but that are not included in the previous section.

- Each occurrence of “JNDI” should be written in the same way (either JNDI or Jndi).
- There is a typing error in line 458: the method is undepoyResource, should be called undeployResource.
- It would be a good practice to group all the inner class declarations into one section (e. g. at the end of the file).

- In the line 761, the *default* branch is included but it does not have any instruction. It is not an immediate problem but a double check ca be done in order to be sure that no actions need to be taken.
- The method dependencyAppliesToScope could be improved by delegating to the Scope the responsibility to check whether a given name applies to it. In this way, it will only be necessary to invoke the method in the received scope instance. In such method it will be used the prefix constants (or the enumeration of prefixes).

The following are based on the suggestions of Netbeans and FindBugs' static analysis.

- Imports in lines 48, 77 and 79 use the star (*).
- Imports in lines 61 and 63 are not used.
- Class complexity is 97, it is too complex.
- The attribute *_logger* should be static and final. Every instance should use the same logger to write information.
- Return statement in line 683 is unnecessary.
- The method addJNDIBindings is too complex, has a cyclomatic complexity of 19.

4. Conclusions

The following graphs show the relation of the amount of lines with issues found per method and per section in the checklist.

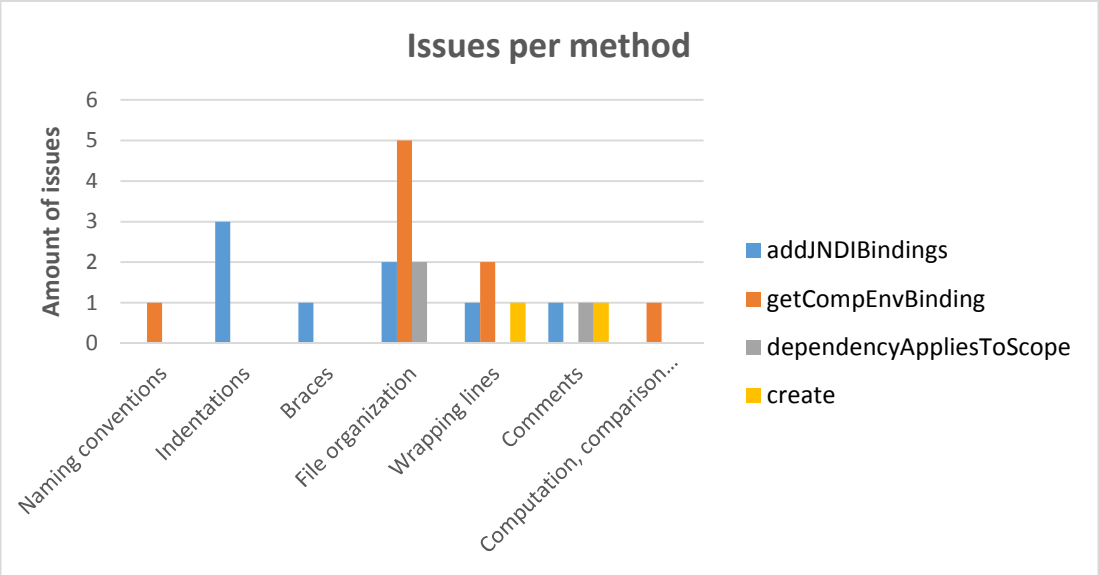

Figure 3: Total issues per method per section

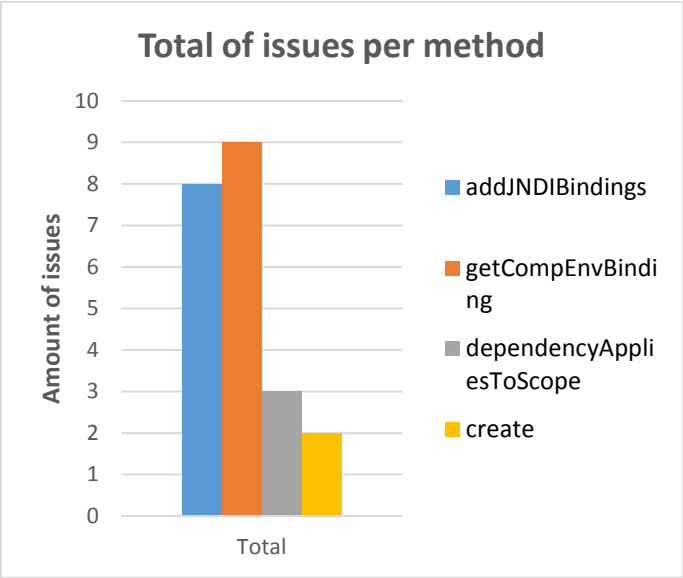

Figure 4: Total issues per method

The graph below shows the amount of issues found per section which are relative to the class in general (not to the four assigned methods)

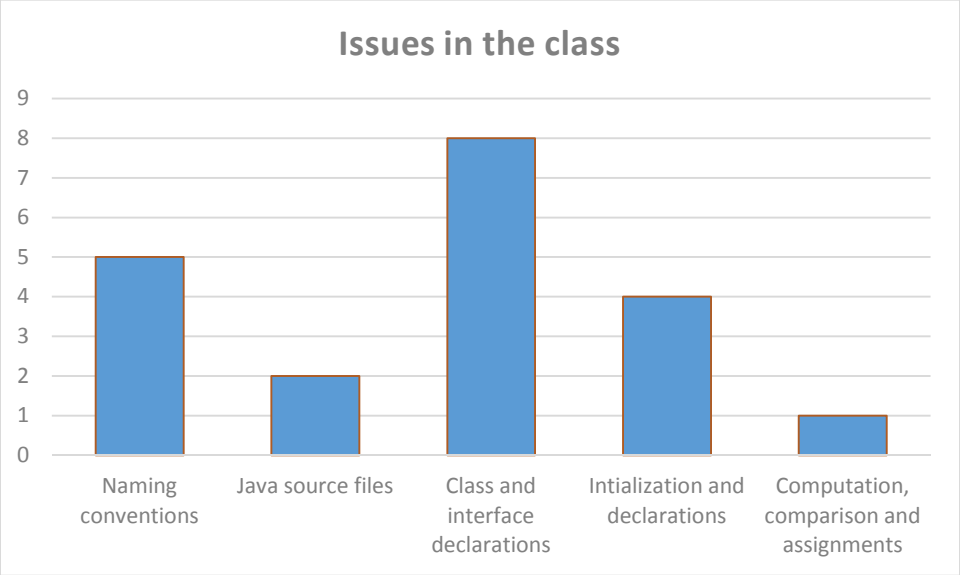

Figure 5: Issues in the class