POLITECNICO
DI MILANO

Code Inspection Document (CID)

Computer Science and Engineering (CSE)

Software Engineering 2 Project

Year 2015/16

Date: 05/01/2016 – Version: 1.0

STUDENTS:

*Martino Andrea (**788701**)*

*Marchesani Francesco (**852444**)*

PROFESSOR:

Mirandola Raffaela

1. Table of Contents

2. Introduction

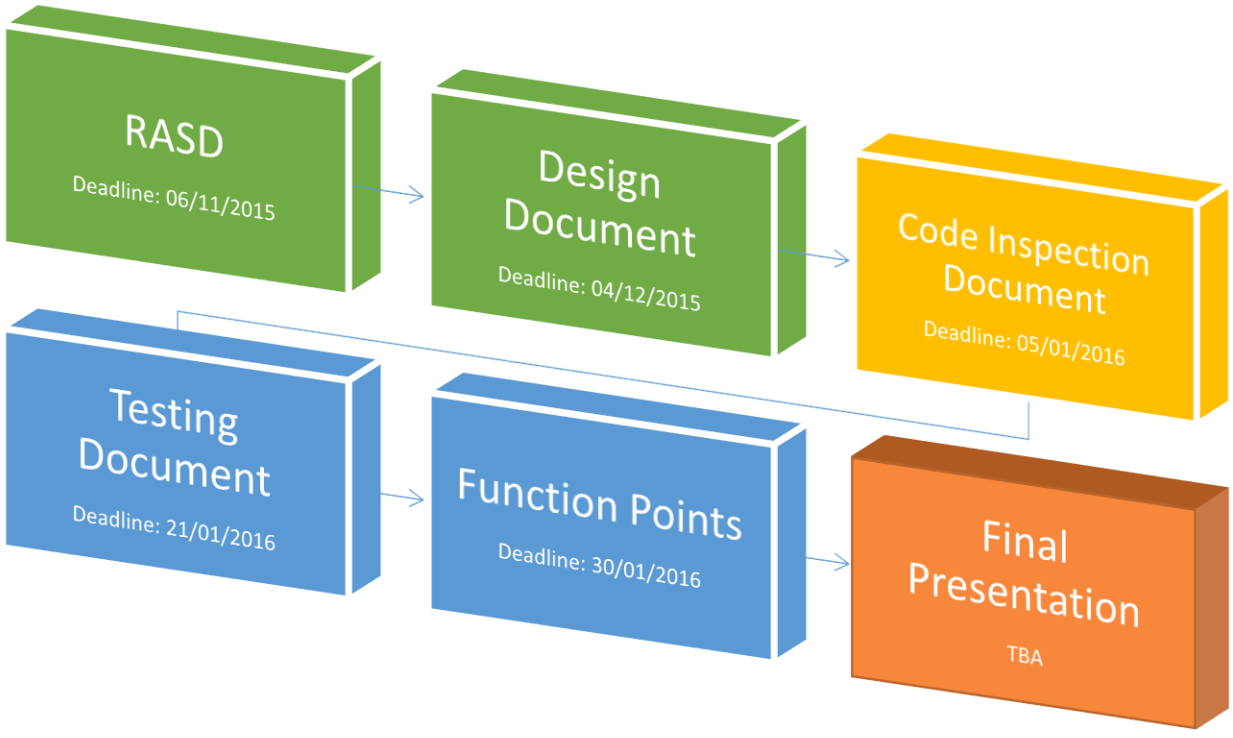
2.1. Purpose

The **Code Inspection Document** is the third document of the **Software Engineering 2 Project** (*Politecnico di Milano – 2015/2016*). This document concerns the systematic examination of code about the release of the **Glassfish 4.1** Application Server.

Our main reference for the code inspection is the *Code Inspection Checklist* delivered with the *Assignment 3 pdf*.

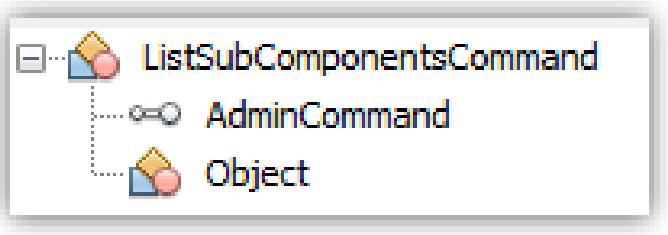
Of course, we will try to follow the guidelines of the reference as much as possible.

Here is a resume of the steps of the project, with the related deadlines (in green documents already delivered, in yellow the current document):



3. Classes that were assigned to the group

3.1. ListSubComponentsCommand



Class name: *ListSubComponentsCommand*

Full class path:
appserver/deployment/javaee-core/src/main/java/org/glassfish/javaee/core/deployment/ListSubComponentsCommand.java

Implemented Interface: *AdminCommand* (`org.glassfish.api.admin.AdminCommand`)

Online Javadoc:
http://glassfish.pompel.me/org/glassfish/javaee/core/deployment/ListSubComponentsCommand.html

Description: We focus only on one class (*ListSubComponentsCommand*) and in particular on two methods (*execute [line 133]* and *getSubModulesListForEar [line 321]*). For the functional role of the class, see paragraph *4. Functional role of assigned set of classes*.

Note: Only one class was assigned to this group.

4. Functional role of assigned set of classes

4.1. ListSubComponentsCommand.java functional role

ListSubCompontentsCommand is an implementation of *AdminCommand,* an interface to provide the command *list-sub-components* to *asadmin* utility of GlassFish server. A brief description of this command is given by **GlassFish** documentation:

" The list-sub-commands *subcommand lists EJBs or servlets in a deployed module or in a module of a deployed application. If a module is not specified, all modules are listed. The* **--appname** *option functions only when the specified module is standalone. To display a specific module in an application, you must specify the module name with the* **--appname** *option. "*

Here you can see the synopsis of the command, to understand how to use parameters:

list-sub-components [--help] [--type *type*] [--appname *appname*] *modulename*

Thanks to the comment in *AdminCommand.java* we understand that an implementation has to be *stateless*[1] and needs to have a *Scope* of value *PerLookup*[2]. Command parameters are handled using *@Param* signatures.

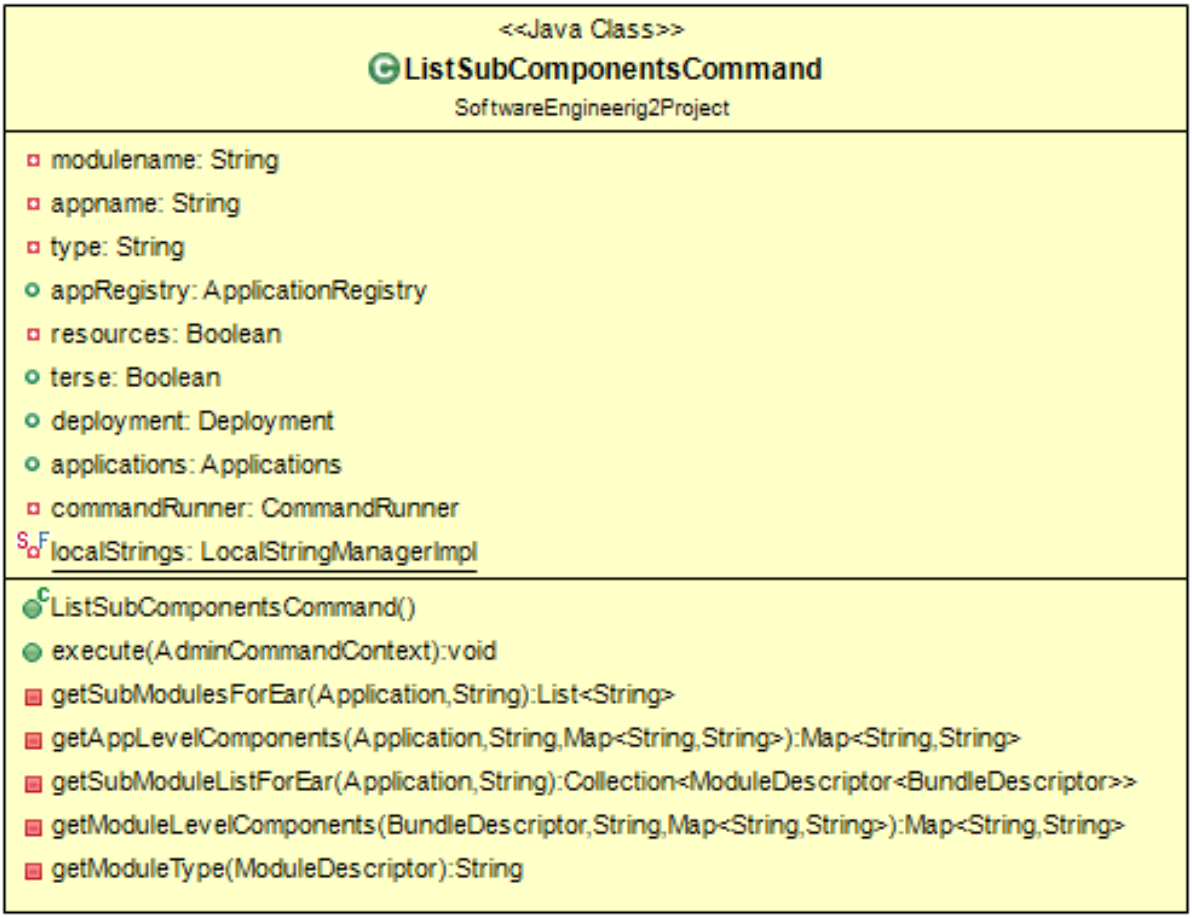
Class can give information about standalone modules (*servlets* or *ejbs*) or *EAR* packages (for more info check the explanation in *getSubmoduleListForEar*).

[1] From Oracle documentation: *A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation.*

[2] From HK2 documentation: PerLookup *is the scope for objects that are created every time they are looked up.* This document will not explain how dependency injection/HK2 framework works.

4.2. UML Class Diagram (Reverse Engineering)

This is the **UML Class Diagram** of *ListSubComponentsCommand.java*.
We obtained this diagram with basilar *Reverse Engineering*:



5. List of issues found by applying the checklist

5.1. Method 1 Checklist

Method	*private List<String> getSubModulesForEar(com.sun.enterprise.deployment.Application application, String type) {…}*
Naming Conventions	• The name of the method is self-explicative. In fact, it collect the list of the sub-modules in the application grouped in the Ear (Enterprise Archive). The other methods involved have meaningful and coherent names. The only suggestion is to change the returned variable name *moduleInfoList* with something like *subModuleEarInfoList* (more meaningful, but also longer in practice). • No one-character variables are used in this method. • The classes involved in the method are nouns in mixed case with the first letter capitalized. Only the *application* one has a path-structure: *com.sun.enterprise.deployment.Application* (probably for path-readability purpose). • The names of the involved methods follow the standard convention (verbs, with the first letter of each addition word capitalized). • The attributes are all in *CamelCase*. • There are not constants involved in this method.
Indentions	• The indentation is always a four spaces indentation (**note**: four spaces equals a tab, as a main standard in programming).
Braces	• The braces follow coherently the Kernighan and Ritchie style. • There are only two blocks in the method (one for, one if). Both of them have the suggested structure: • ***for/if*** *<condition> {* *<< lines of code >>* *}*

Files Organization	• No blank lines and optional comments are used to separate the different parts of the code. For example, the insertion of blank lines between lines 278-279, 279-280, 282-283 may be a good idea (in order to separate lines of code different from a conceptual point of view). • There is only a very long line (279) with 115 characters (over 80, but still less than 120). We can do better with a compression of the application class path -> from "*com.sun.enterprise.deployment.Application*" to just "*Application*".
Wrapping Lines	• Line breaks always occur after a comma or an operator. • No high level breaks are used, because they are not requested in this specific part of code. • The statements are always aligned with respect to the nesting level.
Comments	• There is only a short comment line (278) at the top of the method. It says: // list sub components for ear The method in fact gets all the subcomponents for ear. Probably, the name of the method is self-explicative, so the comment is a little bit redundant. • There are not out-date comments for this method.
Java Source Files	• There is not Javadoc in the method (and in the class). • There is only one class declared in *ListSubComponentsCommand.java*, that implements one interface (*AdminCommand*). • There are not inner classes.
Package and Import Statements	• There are many "*import*" commands, on the top of the class.
Class and Interface Declarations	• We are analyzing the *getSubModuleForEar* method in detail. By the way, we have analyzed the full class in paragraph 5.3.
Initialization and Declarations	• The method is private because it is not used in other classes. • There is only one variable in the method (the list of submodules information). It is clearly nested in the right way. • The list is initialized to empty by default. • The variable *moduleInfoList* is clearly declared at the beginning of the block.
Method Calls	• The parameters are presented in the correct order.

	• Every method called in the *getSubModulesForEar* is the right one. There not exist wrong method calls.
Arrays	• There are not arrays in this code. Therefore, we will not have problem with indexes or array overflow. Anyway, there is an ArrayList (*moduleInfoList*), without problems to highlight.
Object Comparison	• There is only one comparison (line 286) and it is done in the right way (with *equals()*).
Output Format	• The method is a getter. There is not a displayed output, but just a returned list of elements. • No error messages are necessary.
Computation, Comparisons and Assignments	• The implementation of the method is elegant. It avoids "brutish programming". • The order of computation/evaluation, operator precedence and parenthesizing is correct. • No additional parentheses are added to clarify the operator precedence. By the way, we suggest to add parentheses after the '=' in lines 284, 287 to clarify the precedence of the assignment over the string concatenation with '+'. • There are no divisions, so there are not denominators that can assume zero as value. • There are no divisions, so we cannot have problem with divisions between integer numbers. • There is only a comparison between objects (line 286). It is done in the right way, using *equals().* • Try-Catch blocks are absent in this part of the code. • There are not implicit type conversions in the analyzed part of code.
Exceptions	• No try-catch blocks are in the code. • There are not relevant exceptions to catch in the method.
Flow of Control	• There are not switch statements in the code. • After the execution of the *for statement*, the method always returns the list of sub-modules. Note that there is always convergence because the sub-modules are in a finite number. • The only one loop (for, lines 283~290) is well formed.

Files	• No Files are directly involved in the method.

5.2. Method 2 Checklist

Method	*public void execute(AdminCommandContext context) {…}*
Naming Conventions	• The name of the method actually represent the core functionality of the class. In fact, execute(…) contains code that is executed during the subcommand call. • One-character integer variables j and i are respectively declared in line 225 and 235. j scope is limited to the for block between lines 225 and 230 and it is used as a counter variable to scan an array. i is used as a throwaway variable (counter) in the for block 236-267; • The classes involved in the method are nouns in mixed case with the first letter capitalized. • The names of the involved methods follow the standard convention (verbs, with the first letter of each addition word capitalized). • The attributes are all in *CamelCase*. • There are not constants involved in this method.
Indentions	• The indentation is always a four spaces indentation. No tab character is used.

Braces	• The braces follow coherently the Kernighan and Ritchie style.
	• There are different blocks in the method (if, for, try/catch) with different levels of nesting. Every block have the suggested structure:
	
	for/if <condition> {
	
	<< lines of code >>
	
	}

Files Organization	• There are different cases in which blank lines are used to separate pieces of code. 134: Separates method declaration from the initialization of report; 136: separates part initialization (135) from report initialization (137);

	138: separates part initialization (137) from applicationName initialization (139); 143, 151, 156: respectively separate code blocks at lines 140-142, 144-150, 151-157; 156: separates return statement (155) from the closing bracket (157) of the code block 152-157; 158: separates closing bracket (157) and application initialization (159); 160: separates application initialization (159) from code block 161-166; 167: separates appInfo initialization (168) from code block 161-167; 173: separates app initialization (174) from the closing bracket of the code block 168-172; 175: separates app initialization (174) from the the code block 176-181; 182: separates subComponents declaration (183) from the closing bracket of code block 176-181; 185: separates subComponentsMap initialization 198: separates code block 186-197 from the comment at line 199; 207: separates subModuleInfos initialization (208) from the closing bracket of code block 200-206; 211: separates closing bracket of code block 209-211 from longestValue initialization; 258: separates the method execution at line 257 from subpart initialization (259); 268: separates the closing bracket of code block 236-267 from the comment at line 269; • There does not seem to be a proper "style" in the use of empty lines. Sometimes are used to separate different code blocks, sometimes to separate variable declaration. • Length of lines 153, 163, 170, 178, 191, 202 exceeds 120. For readability purposes, it would be better to replace some strings with some static final variables. For example at line 202 the first two string parameters passed to getLocalString("listsubcomponents.invalidtype", "The type option has invalid value {0}. It should have a value of servlets or ejbs.", type) can be replaced with getLocalString(INV_TYPE_PROP, INV_TYPE_DEFAULT_STR, type) . In this way, length is reduced from 189 to 107. • Length of line 253 exceeds 120. In this case, we think that would be better to reduce the level of nesting and cyclomatic complexity, maybe delegating some operations to a new method.

Wrapping Lines	• Line breaks always occur after a comma or an operator. • No high level breaks are used, because they are not requested in this specific part of code. • The statements are always aligned with respect to the nesting level.

Comments	• Line 199
	// the type param can only have values "ejbs" and "servlets"
	better explains why and how type value is checked in lines 200 and 201;
	• Line 269
	// add the properties for GUI to display
	explains what the code below actually does, prepare the output message.
	• Line 274
	// now this is the normal output for the list-sub-components command
	simply states that if code execution has come to this point, then no problem was found and a normal output could be given to the originator of the action.
	• There are not outdated comments for this method.
	• This method lacks a proper documentation and meaningful comments. In fact, every comments seems to be only a reminder for the developer who wrote this class. In general, this is not a good practice because code must be written to be as readable as possible by anyone, but in this case it's not a big deal.

Initialization and Declarations	• The method is public because it will be called by other software components, maybe belonging to different packages. • subComponents is declared and not initialized at line 184. The initialization depends upon following code lines. • subModuleInfos is declared and initialized at line 208. This object is only used at lines 210, 244, 246 if app.isVirtual() is false. If app.isVirtual() is false then subModuleInfos points to getSubModulesForEar(app, type). It seems that subModuleInfos initialization is only a dirty trick to avoid compile time error at 242 (subModuleInfos might not have been initialized). • Some variables used in execute are not declared at the beginning of the method. In general, this is not a good practice but in this case it could be a wise choice because in some blocks return statement is present. Doing in this way the number of declared not used variables is reduced. For example, variable subComponents is declared at line 183. A return statement is at line 180. If some conditions are verified execute can terminate before declaring subComponents. But subComponents variable is actually used only from line 187.

Method Calls	• The parameters are presented in the correct order. • Every method called in the execute() is the right one. There not exist wrong method calls.
Arrays	• At line 213 longestValue is declared as an array of two elements. At lines 216, 217, 220, 221 and 226 longestValue valid position are accessed. In fact on lines 216 and 217 longestValue[0] is accessed (that is, the position with index 0) and on lines 220 and 221 longestValue[1] is accessed. During the for block at lines 225-230 j assumes integer value of 0 and 1. So statement longestValue[j] doesn't cause overflow/underflow. • No problems were highlighted during analysis for what regards iteration/access on collections. There is only one case of manual collection access at code blocks 244-246.
Object Comparison	• At lines 140, 169, 176, 190, 200, 241 and 244 NULL checks are performed. In those cases, "==" comparison (that is, a comparison between references) is needed because it's necessary to understand whether a variable is a reference to an object or not. • At line 201 type string value is checked. In this case "equals.()" comparison is needed because a "==" would always return false as the strings checked are stored in different memory areas.
Output Format	• No output is directly handled by this method. However execute is able to call methods of the injected implementation of ActionReport context.getActionReport(). Every possible error is then communicated to the originator of the action (using report.setMessage and report.setActionMethod methods).
Computation, Comparisons and Assignments	• The order of computation/evaluation, operator precedence and parenthesizing is correct. • There is no need to add parenthesis to avoid operators' precedence issues. • There are no divisions, so there are not denominators that can assume zero as value. • There are no divisions, so we cannot have problem with divisions between integer numbers. • Objects comparison are correctly done. Check Object Comparison section. • There are not implicit type conversions in the analyzed part of code. • No problem found in exception handling. See Exceptions section for more.

Exceptions	• There is only one try/catch block (144-150). applicationName should be composed by the application name and the version identifier, separated by a chosen character. So checkIdentifier(applicationName) method of VersioningUtils class is called to check if applicationName is syntactically correct. If not VersioningSyntaxException will be thrown. In this case the exception is catch by catch block at line 146, the output is populated with information about the exception, the exit code is properly set and the method is then terminated.
Flow of Control	• There are not switch statements in the code. • Every loops in the method is then analyzed: Lines 214-223 subComponents.entrySet() is a finite set. So the iterator will scan a limited number of elements Map.Entry. Lines 225-230 No problem on this loop. In fact, as introduced in Array section, j can only assume value 0 and 1 (j initial value is 0 and is incremented by 1 unit every iteration) and then the loop will terminate (when condition j<2 is not valid anymore). Lines 236-267 Similar to the block 214-223, subComponents.keySet() is a finite set.
Files	• No Files are directly involved in this method, even though there could be references to methods/classes that handle files (for example LocalStringManagerImpl class looks for LocalString.properties).

5.3. Assigned Class in detail

As we said in the checklists of the methods, in this chapter we will analyze the full class *ListSubComponentsCommand.java* following the specific points of the reference checklist.

We will focus on points **24~33** of the checklist. In fact, these points reflect the entire class and not just the two assigned methods.

- The package statement is the first of the non-comment statements.
- Then there are all the import statements, as suggested in the checklist.
- The class order follows the indications of the point 25 of the checklist.
- Methods are grouped by functionality as suggested.
- The code is free of duplicates and the class is not huge. Anyway, there is a very long methods: *execute(),* with 136 lines of code. We strongly suggest splitting the method in sub-methods in order to respect the "*Divide et Impera*" principle of software engineering.
- All the variables and class member are of the correct type.
- No 'standard' constructor of the class exists.
- All parameters are initialized before the use with a value, also with *null* in some cases, see:

@Param(primary=true)

private String modulename = null;

@Param(optional=true)

private String appname = null;

@Param(optional=true)

private String type = null;

[lines 104~111]

- Declarations of variables are always at the beginning of the blocks {…}.

6. Other highlighted problems

We want to remark two problems about the *Code Documentation.*

- First, we do not have an appropriate documentation with **comments**. In fact, comments are rare in the code and they are probably too much concise. We know that it is not a good fact to write too many comments, but in this case there is a lack of comments.
  Note that the methods are not always easy to understand from an external point of view. A good comprehension takes several hours of *reverse engineering*.

- Second, the **JavaDoc** of the class is very basilar. There are only the header of the class [lines 85 ~ 101] and the indications of Parameters (*@Param*) and Injections (*@Inject*). Our suggestion is to improve the *JavaDoc* (for example with pre-conditions/post-conditions) in order to give the software a better documentation (and to make it more professional and precise).

7. Hours of work

- **Andrea Martino** ~ 25 Hours
- **Francesco Marchesani** ~ 25 Hours