# Checklist Table

*execute*

|  |  |
| --- | --- |
| **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 doesn’t 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;   * Lines 242, 243   // we use the property mechanism to provide  // support for JSR88 client  we won’t go in details   * 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 DA CONTINUARE |
| **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’s 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’s 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*). |