POLITECNICO
MILANO 1863

CODE INSPECTION

Software Engineering 2 – AA 2016/2017

Daniel Botta
806545

Agnese Bruschi
810762

Adriana Cano
812233

Summary

1 Introduction

1.1 Purpose and scope

The purpose of this document is to report the result of the code inspection. Following the lines of the code inspection document who was assigned to perform the task, the team run an analysis of the class, and here reported the conclusions.

For every possible semantic or syntactic fault, the lines in which the problem appears were reported, with an explanation regarding why that particular aspect is a problem or an error, and how may be updated or improved.

1.2 Assigned classes

The assigned class was *ModelMenu.java*, a class whose scope is to create an XML file.

1.3 Related Documents

- ModelMenu.java
- Code Inspection Assignment Task Description

2 Code Inspection

2.1 Naming Conventions

#1 Issue: Line 58, Global variables

Constants should be declared using all uppercase with words separated by an underscore. The name of the variable “module” should have all characters uppercase.

```
57
58      public static final String module = ModelMenu.class.getName();
59
```

2.2 Indentation

No issues were found on indentation: four spaces were used and consistently and tabs were not found in the class.

2.3 Braces

Consistent braced style is used throughout the whole class. The “Kernighan and Ritchie” style was used. Although it was consistent, parts of code were missing them. A large number of if statements, with one instruction, were missing braces. This could cause confusion and can make the mistakes easier to commit.

#1 Issue: Lines 195 - 248, Constructor

```
195         if (!menuElement.getAttribute("type").isEmpty())
196             type = menuElement.getAttribute("type");
197         if (!menuElement.getAttribute("target").isEmpty())
198             target = menuElement.getAttribute("target");
199         if (!menuElement.getAttribute("id").isEmpty())
200             id = menuElement.getAttribute("id");
201         if (!menuElement.getAttribute("title").isEmpty())
202             title = FlexibleStringExpander.getInstance(menuElement.getAttribute("title"));
203         if (!menuElement.getAttribute("tooltip").isEmpty())
204             tooltip = menuElement.getAttribute("tooltip");
205         if (!menuElement.getAttribute("default-entity-name").isEmpty())
206             defaultEntityName = menuElement.getAttribute("default-entity-name");
207         if (!menuElement.getAttribute("default-title-style").isEmpty())
208             defaultTitleStyle = menuElement.getAttribute("default-title-style");
209         if (!menuElement.getAttribute("default-selected-style").isEmpty())
210             defaultSelectedStyle = menuElement.getAttribute("default-selected-style");
211         if (!menuElement.getAttribute("default-widget-style").isEmpty())
212             defaultWidgetStyle = menuElement.getAttribute("default-widget-style");
213         if (!menuElement.getAttribute("default-tooltip-style").isEmpty())
214             defaultTooltipStyle = menuElement.getAttribute("default-tooltip-style");
215         if (!menuElement.getAttribute("default-menu-item-name").isEmpty())
216             defaultMenuItemName = menuElement.getAttribute("default-menu-item-name");
217         if (!menuElement.getAttribute("default-permission-operation").isEmpty())
218             defaultPermissionOperation = menuElement.getAttribute("default-permission-operation");
219         if (!menuElement.getAttribute("default-permission-entity-action").isEmpty())
220             defaultPermissionEntityAction = menuElement.getAttribute("default-permission-entity-action");
221         if (!menuElement.getAttribute("default-associated-content-id").isEmpty())
222             defaultAssociatedContentId = FlexibleStringExpander.getInstance(menuElement
223                     .getAttribute("default-associated-content-id"));
```

Image 2.3.A

```
224         if (!menuElement.getAttribute("orientation").isEmpty())
225             orientation = menuElement.getAttribute("orientation");
226         if (!menuElement.getAttribute("menu-width").isEmpty())
227             menuWidth = menuElement.getAttribute("menu-width");
228         if (!menuElement.getAttribute("default-cell-width").isEmpty())
229             defaultCellWidth = menuElement.getAttribute("default-cell-width");
230         if (!menuElement.getAttribute("default-hide-if-selected").isEmpty())
231             defaultHideIfSelected = "true".equals(menuElement.getAttribute("default-hide-if-selected").isEmpty());
232         if (!menuElement.getAttribute("default-disabled-title-style").isEmpty())
233             defaultDisabledTitleStyle = menuElement.getAttribute("default-disabled-title-style");
234         if (!menuElement.getAttribute("selected-menuitem-context-field-name").isEmpty())
235             selectedMenuItemContextFieldName = FlexibleMapAccessor.getInstance(menuElement
236                     .getAttribute("selected-menuitem-context-field-name"));
237         if (!menuElement.getAttribute("menu-container-style").isEmpty())
238             menuContainerStyleExdr = FlexibleStringExpander.getInstance(menuElement.getAttribute("menu-container-style"));
239         if (!menuElement.getAttribute("default-align").isEmpty())
240             defaultAlign = menuElement.getAttribute("default-align");
241         if (!menuElement.getAttribute("default-align-style").isEmpty())
242             defaultAlignStyle = menuElement.getAttribute("default-align-style");
243         if (!menuElement.getAttribute("fill-style").isEmpty())
244             fillStyle = menuElement.getAttribute("fill-style");
245         if (!menuElement.getAttribute("extra-index").isEmpty())
246             extraIndex = FlexibleStringExpander.getInstance(menuElement.getAttribute("extra-index"));
```

Image 2.3.B

Braces are needed even for one instruction. Better safe than sorry, it's easy to avoid this kind of mistakes, just add a brace.

#2 Issue: Line 480, *renderedMenuItemCount*(...) function:

```
479         for (ModelMenuItem item : this.menuItemList) {
480             if (item.shouldBeRendered(context))
481                 count++;
482         }
```

The same as what has been said for #1 Issue.

2.4 File Organization

As a standard it's good to keep 80 characters as limit. This increases readability of a given class. Although this might be true, sometimes going over 80 characters has more advantages than making a new line. In this class the 80 char limit has been surpassed quite often, but with good reasons, like giving complete and unambiguous names to variables. In the constructor there are many objects with long names, thus it's hard to stay in the limit of 80 during assignments.

In spite of the fact that sometimes it might be a good practice to surpass the limit, in some occasions it's just better to make a new line.

A few examples of acceptable exceeded limits will follow:

Line 110, 82 char

```
110         FlexibleStringExpander defaultAssociatedContentId = FlexibleStringExpander.getInstance("");
```

Line 127, 87 char

```
127         Map<String, ModelMenuItem> menuItemMap = new HashMap<String, ModelMenuItem>();
```

Line 231, 115 char

```
230         if (!menuElement.getAttribute("default-hide-if-selected").isEmpty())
231             defaultHideIfSelected = "true".equals(menuElement.getAttribute("default-hide-if-selected").isEmpty());
```

As seen in these examples, it is more clear to keep it on one line. Inserting another line to just write "getInstance("") would make it harder to read. No further examples like this will be shown (because they aren't real issues), but they do exist in the class.

On the contrary these are the kind of limit exceeded that should not be committed:

#1 Issue, Line 231, Constructor:

```
300     private void addUpdateMenuItem(ModelMenuItem modelMenuItem, List<ModelMenuItem> menuItemList,
301             Map<String, ModelMenuItem> menuItemMap) {
```

The line 231 has 98 chars. While it is under the 120 char limit, there is a new line over the 80 char limit. If a new line was going to be made anyways, might as well have put it *before* the 80 char limit.

#2 Issue, Line 144, Constructor:

```
143                 } catch (Exception e) {
144                     Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                             + "'", module);
```

This line has 130 char. The problem here is that not only exceeds the limit, but it also makes a new line. A new line could have been made sooner. Another mistake is made here but I will go into it in a following section.

#3 Issue, Line 159, Constructor:

```
158                 if (parent == null) {
159                     Debug.logError("Failed to find parent menu definition '" + parentMenu + "' in same document.", module);
160                 }
```

124 char line. A new line should have been made.

#4 Issue, Line 238, Constructor:

```
237         if (!menuElement.getAttribute("menu-container-style").isEmpty())
238             menuContainerStyleExdr = FlexibleStringExpander.getInstance(menuElement.getAttribute("menu-container-style"));
```

123 char line. A new line should have been made

#5 Issue, Line 512, *renderSimpleMenuString*(...) method:

```
512     public void renderSimpleMenuString(Appendable writer, Map<String, Object> context, MenuStringRenderer menuStringRenderer)
```

123 char line. A new line should have been made

2.5 Wrapping Lines

The wrapping lines guidelines are necessary to correct the “File Organization” issues. They give rules so it’s clear when a new line is required. In general, a line break occurs *after* a comma or an operator, and high level breaks are used.

#1 Issue, Line 144, Constructor:

```
144                     Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                             + "'", module);
```

It is bad practice to conclude a line without an operator or a comma. Here the line break was done *before* an operator.

#2 Issue, Line 507, renderMenuString(...) method:

```
507                 throw new IllegalArgumentException("The type " + this.getType() + " is not supported for menu with name "
508                         + this.getName());
```

Same issue, line break *before* operator.

2.6 Comments

Comments are essential to understand quickly what a snippet of code should do. This class is surely **lacking of comments**. There are many parts of code without any explanation, though no commented out code was found in the document.

Some parts of code that need comments will be shown.

```
136         ModelMenu parent = null;
137         String parentResource = menuElement.getAttribute("extends-resource");
138         String parentMenu = menuElement.getAttribute("extends");
139         if (!parentMenu.isEmpty()) {
140             if (!parentResource.isEmpty()) {
141                 try {
142                     parent = MenuFactory.getMenuFromLocation(parentResource, parentMenu);
143                 } catch (Exception e) {
144                     Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                             + "'", module);
146                 }
147             } else {
148                 parentResource = menuLocation;
149                 // try to find a menu definition in the same file
150                 Element rootElement = menuElement.getOwnerDocument().getDocumentElement();
151                 List<? extends Element> menuElements = UtilXml.childElementList(rootElement, "menu");
152                 for (Element menuElementEntry : menuElements) {
153                     if (menuElementEntry.getAttribute("name").equals(parentMenu)) {
154                         parent = new ModelMenu(menuElementEntry, parentResource);
155                         break;
156                     }
157                 }
158                 if (parent == null) {
159                     Debug.logError("Failed to find parent menu definition '" + parentMenu + "' in same document.", module);
160                 }
161             }
```

Only one section is explained here, and it's done poorly. Had the programmer taken more time to comment he would have made it easier to read.

```
162             if (parent != null) {
163                 type = parent.type;
164                 target = parent.target;
165                 id = parent.id;
166                 title = parent.title;
167                 tooltip = parent.tooltip;
168                 defaultEntityName = parent.defaultEntityName;
169                 defaultTitleStyle = parent.defaultTitleStyle;
170                 defaultSelectedStyle = parent.defaultSelectedStyle;
171                 defaultWidgetStyle = parent.defaultWidgetStyle;
172                 defaultTooltipStyle = parent.defaultTooltipStyle;
173                 defaultMenuItemName = parent.defaultMenuItemName;
174                 menuItemList.addAll(parent.menuItemList);
175                 menuItemMap.putAll(parent.menuItemMap);
176                 defaultPermissionOperation = parent.defaultPermissionOperation;
177                 defaultPermissionEntityAction = parent.defaultPermissionEntityAction;
178                 defaultAssociatedContentId = parent.defaultAssociatedContentId;
179                 defaultHideIfSelected = parent.defaultHideIfSelected;
180                 orientation = parent.orientation;
181                 menuWidth = parent.menuWidth;
182                 defaultCellWidth = parent.defaultCellWidth;
183                 defaultDisabledTitleStyle = parent.defaultDisabledTitleStyle;
184                 defaultAlign = parent.defaultAlign;
185                 defaultAlignStyle = parent.defaultAlignStyle;
186                 fillStyle = parent.fillStyle;
187                 extraIndex = parent.extraIndex;
188                 selectedMenuItemContextFieldName = parent.selectedMenuItemContextFieldName;
189                 menuContainerStyleExdr = parent.menuContainerStyleExdr;
190                 if (parent.actions != null) {
191                     actions.addAll(parent.actions);
192                 }
193             }
```

The same goes here. Nearly 50 full lines of code without explaining with comments what is going on. It takes time to fully comprehend what is happening here.

```
195         if (!menuElement.getAttribute("type").isEmpty())
196             type = menuElement.getAttribute("type");
197         if (!menuElement.getAttribute("target").isEmpty())
198             target = menuElement.getAttribute("target");
199         if (!menuElement.getAttribute("id").isEmpty())
200             id = menuElement.getAttribute("id");
201         if (!menuElement.getAttribute("title").isEmpty())
202             title = FlexibleStringExpander.getInstance(menuElement.getAttribute("title"));
203         if (!menuElement.getAttribute("tooltip").isEmpty())
204             tooltip = menuElement.getAttribute("tooltip");
205         if (!menuElement.getAttribute("default-entity-name").isEmpty())
206             defaultEntityName = menuElement.getAttribute("default-entity-name");
207         if (!menuElement.getAttribute("default-title-style").isEmpty())
208             defaultTitleStyle = menuElement.getAttribute("default-title-style");
209         if (!menuElement.getAttribute("default-selected-style").isEmpty())
210             defaultSelectedStyle = menuElement.getAttribute("default-selected-style");
211         if (!menuElement.getAttribute("default-widget-style").isEmpty())
212             defaultWidgetStyle = menuElement.getAttribute("default-widget-style");
213         if (!menuElement.getAttribute("default-tooltip-style").isEmpty())
214             defaultTooltipStyle = menuElement.getAttribute("default-tooltip-style");
215         if (!menuElement.getAttribute("default-menu-item-name").isEmpty())
216             defaultMenuItemName = menuElement.getAttribute("default-menu-item-name");
217         if (!menuElement.getAttribute("default-permission-operation").isEmpty())
218             defaultPermissionOperation = menuElement.getAttribute("default-permission-operation");
219         if (!menuElement.getAttribute("default-permission-entity-action").isEmpty())
220             defaultPermissionEntityAction = menuElement.getAttribute("default-permission-entity-action");
221         if (!menuElement.getAttribute("default-associated-content-id").isEmpty())
222             defaultAssociatedContentId = FlexibleStringExpander.getInstance(menuElement
223                     .getAttribute("default-associated-content-id"));
224         if (!menuElement.getAttribute("orientation").isEmpty())
225             orientation = menuElement.getAttribute("orientation");
226         if (!menuElement.getAttribute("menu-width").isEmpty())
227             menuWidth = menuElement.getAttribute("menu-width");
228         if (!menuElement.getAttribute("default-cell-width").isEmpty())
229             defaultCellWidth = menuElement.getAttribute("default-cell-width");
```

This chain of *if* statements is indeed hard to read. The absence of comments takes away clarity.

2.7 Java Source Files

Each Java source should contain a single public class or interface to increase readability.

Not a lot of JavaDoc is given for the class: more is needed. There is only one method with JavaDoc, and the class is longer than 500 lines of code. More should be provided.

Some examples of locations where JavaDoc should exist:

```
512     public void renderSimpleMenuString(Appendable writer, Map<String, Object> context, MenuStringRenderer menuStringRenderer)
513             throws IOException {
514         // render menu open
515         menuStringRenderer.renderMenuOpen(writer, context, this);
516
517         // render formatting wrapper open
518         menuStringRenderer.renderFormatSimpleWrapperOpen(writer, context, this);
519
520         // render each menuItem row, except hidden & ignored rows
521         for (ModelMenuItem item : this.menuItemList) {
522             item.renderMenuItemString(writer, context, menuStringRenderer);
523         }
524         // render formatting wrapper close
525         menuStringRenderer.renderFormatSimpleWrapperClose(writer, context, this);
526
527         // render menu close
528         menuStringRenderer.renderMenuClose(writer, context, this);
529     }
```

While an attempt for comments is being made, javadoc should be there instead. It would make everything more clear.

```
477     public int renderedMenuItemCount(Map<String, Object> context) {
478         int count = 0;
479         for (ModelMenuItem item : this.menuItemList) {
480             if (item.shouldBeRendered(context))
481                 count++;
482         }
483         return count;
484     }
```

There are many small functions like this that do not present any comment or javadoc.

```
449     public String getSelectedMenuItemContextFieldName(Map<String, Object> context) {
450         String menuItemName = this.selectedMenuItemContextFieldName.get(context);
451         if (UtilValidate.isEmpty(menuItemName)) {
452             return this.defaultMenuItemName;
453         }
454         return menuItemName;
455     }
```

Like this one:

```
296     /**
297      * add/override modelMenuItem using the menuItemList and menuItemMap
298      *
299      */
300     private void addUpdateMenuItem(ModelMenuItem modelMenuItem, List<ModelMenuItem> menuItemList,
301             Map<String, ModelMenuItem> menuItemMap) {
302         ModelMenuItem existingMenuItem = menuItemMap.get(modelMenuItem.getName());
303         if (existingMenuItem != null) {
304             // does exist, update the item by doing a merge/override
305             ModelMenuItem mergedMenuItem = existingMenuItem.mergeOverrideModelMenuItem(modelMenuItem);
306             int existingItemIndex = menuItemList.indexOf(existingMenuItem);
307             menuItemList.set(existingItemIndex, mergedMenuItem);
308             menuItemMap.put(modelMenuItem.getName(), mergedMenuItem);
309         } else {
310             // does not exist, add to Map
311             menuItemList.add(modelMenuItem);
312             menuItemMap.put(modelMenuItem.getName(), modelMenuItem);
313         }
314     }
```

Here space for javadoc was even made, but not utilized. A few comments are present but not clear enough for readers.

2.8 Package Import

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

In the ModelMenu class this rule was followed impeccably. No issues found here.

2.9 Class and Interface Declarations

Correct code:

The class or interface declarations are **well ordered**, following this list:

1. Class/interface documentation comment [Lines 1-40]]
   First, we have the licence comment, and then the description of the package to which the class belongs. The imports are organized by type, and ordered well.
2. Class or interface statement
   ```
   42  public class ModelMenu extends ModelWidget {
   ```
3. Class/interface implementation comment [Lines 44-56]
   This class is a MODEL DATA STRUCTURE that represents an XML document, thus it must be immutable.
4. Class (static) variables
5. Instance variables [Lines 58-102]
   There are no static variables, (which should go first), and, regarding instance variables: there is only one public variable, and multiple private final variables, which are declared after the public one.

6. Constructors [Lines 105-290]
   XML Constructor is in the right position, after all the variables are listed.
7. Methods. [Lines 291-533]

Methods are correctly grouped by **functionality**: in fact, all the getters are clustered together, [Lines 316-473], and the *rendering* functions are grouped [Lines 489-529*, renderMenuString, renderSimpleMenuString*].

Problems:

- The class is very **big**. Counting comments, this class is longer than 500 lines of code (534): another object should have been created to simplify this class.

- Another issue is that many parts of the code are long and complicated to read. An example from before can be seen:

```
162         if (parent != null) {
163             type = parent.type;
164             target = parent.target;
165             id = parent.id;
166             title = parent.title;
167             tooltip = parent.tooltip;
168             defaultEntityName = parent.defaultEntityName;
169             defaultTitleStyle = parent.defaultTitleStyle;
170             defaultSelectedStyle = parent.defaultSelectedStyle;
171             defaultWidgetStyle = parent.defaultWidgetStyle;
172             defaultTooltipStyle = parent.defaultTooltipStyle;
173             defaultMenuItemName = parent.defaultMenuItemName;
174             menuItemList.addAll(parent.menuItemList);
175             menuItemMap.putAll(parent.menuItemMap);
176             defaultPermissionOperation = parent.defaultPermissionOperation;
177             defaultPermissionEntityAction = parent.defaultPermissionEntityAction;
178             defaultAssociatedContentId = parent.defaultAssociatedContentId;
179             defaultHideIfSelected = parent.defaultHideIfSelected;
180             orientation = parent.orientation;
181             menuWidth = parent.menuWidth;
182             defaultCellWidth = parent.defaultCellWidth;
183             defaultDisabledTitleStyle = parent.defaultDisabledTitleStyle;
184             defaultAlign = parent.defaultAlign;
185             defaultAlignStyle = parent.defaultAlignStyle;
186             fillStyle = parent.fillStyle;
187             extraIndex = parent.extraIndex;
188             selectedMenuItemContextFieldName = parent.selectedMenuItemContextFieldName;
189             menuContainerStyleExdr = parent.menuContainerStyleExdr;
190             if (parent.actions != null) {
191                 actions.addAll(parent.actions);
192             }
193         }
```

  This kind of writing style goes on for quite a while. First, as we already stated, there aren't any comments, but beside that, even if there were comments, it would have been equally hard to read the class. This is because it's a cluster of lines of code without a structure.
  More methods should have been created to increase the readability.

- The *XML Constructor* is a huge method, too **long** to be readable: it takes more than a hundred lines of code, and the actions taken in it may be better off split into new functions, sub-functions of the constructor.

- All the temporary variables declared in the constructor are **duplicates** of the private final attributes of the class. It is preferable to find a more efficient way to control the values before assigning them to the private final attributes of the class.

```
108  String defaultAlign = "";
109  String defaultAlignStyle = "";
110  FlexibleStringExpander defaultAssociatedContentId = FlexibleStringExpander.getInstance("");
111  String defaultCellWidth = "";
112  String defaultDisabledTitleStyle = "";
113  String defaultEntityName = "";
114  Boolean defaultHideIfSelected = Boolean.FALSE;
115  String defaultMenuItemName = "";
116  String defaultPermissionEntityAction = "";
117  String defaultPermissionOperation = "";
118  String defaultSelectedStyle = "";
119  String defaultTitleStyle = "";
120  String defaultTooltipStyle = "";
121  String defaultWidgetStyle = "";
122  FlexibleStringExpander extraIndex = FlexibleStringExpander.getInstance("");
123  String fillStyle = "";
124  String id = "";
125  FlexibleStringExpander menuContainerStyleExdr = FlexibleStringExpander.getInstance("");
126  ArrayList<ModelMenuItem> menuItemList = new ArrayList<ModelMenuItem>();
127  Map<String, ModelMenuItem> menuItemMap = new HashMap<String, ModelMenuItem>();
128  String menuWidth = "";
129  String orientation = "horizontal";
130  FlexibleMapAccessor<String> selectedMenuItemContextFieldName = FlexibleMapAccessor.getInstance("")
131  String target = "";
132  FlexibleStringExpander title = FlexibleStringExpander.getInstance("");
133  String tooltip = "";
134  String type = "";
135  // check if there is a parent menu to inherit from
136  ModelMenu parent = null;
```

Image 2.9.A

- The constructor part in which it is required to check if there is a parent menu to inherit from [Image 2.9.A] could easily be removed and exported in a **new function** (for instance, *checkParentExistence*()).

  In fact, this function could return the parent itself if it exists, *null* otherwise.

  After checking if the parent menu does exist or not, if the parent menu exists, it is possible to assign directly the values to the private final attributes, without using the temporary ones.

```
135   // check if there is a parent menu to inherit from
136   ModelMenu parent = null;
137   String parentResource = menuElement.getAttribute("extends-resource");
138   String parentMenu = menuElement.getAttribute("extends");
139   if (!parentMenu.isEmpty()) {
140       if (!parentResource.isEmpty()) {
141           try {
142               parent = MenuFactory.getMenuFromLocation(parentResource, parentMenu);
143           } catch (Exception e) {
144               Debug.logError(e, "Failed to load parent menu definition '"
145
146               + parentMenu + "' at resource '" + parentResource
147                       + "'", module);
148           }
149       } else {
150           parentResource = menuLocation;
151           // try to find a menu definition in the same file
152           Element rootElement = menuElement.getOwnerDocument().getDocumentElement();
153           List<? extends Element> menuElements = UtilXml.childElementList(rootElement, "menu");
154           for (Element menuElementEntry : menuElements) {
155               if (menuElementEntry.getAttribute("name").equals(parentMenu)) {
156                   parent = new ModelMenu(menuElementEntry, parentResource);
157                   break;
158               }
159           }
160           if (parent == null) {
161               Debug.logError("Failed to find parent menu definition '" + parentMenu +
162               "' in same document.", module);
163           }
164       }
```

Image 2.9.B

- From line 198 to line 249 the constructor checks if the parameters are empty and, if they are, they're filled in. This should be done only after the parent menu is confirmed to not exist, so the assignments could be done directly to the private final attributes. Implementing the constructor in this way will help avoiding **redundancy** and useless creation of new variables, saving **memory**.

```
198   if (!menuElement.getAttribute("type").isEmpty())
199       type = menuElement.getAttribute("type");
200   if (!menuElement.getAttribute("target").isEmpty())
201       target = menuElement.getAttribute("target");
202   if (!menuElement.getAttribute("id").isEmpty())
203       id = menuElement.getAttribute("id");
204   if (!menuElement.getAttribute("title").isEmpty())
205       title = FlexibleStringExpander.getInstance(menuElement.getAttribute("title"));
206   if (!menuElement.getAttribute("tooltip").isEmpty())
207       tooltip = menuElement.getAttribute("tooltip");
208   if (!menuElement.getAttribute("default-entity-name").isEmpty())
209       defaultEntityName = menuElement.getAttribute("default-entity-name");
210   if (!menuElement.getAttribute("default-title-style").isEmpty())
211       defaultTitleStyle = menuElement.getAttribute("default-title-style");
212   if (!menuElement.getAttribute("default-selected-style").isEmpty())
213       defaultSelectedStyle = menuElement.getAttribute("default-selected-style");
214   if (!menuElement.getAttribute("default-widget-style").isEmpty())
215       defaultWidgetStyle = menuElement.getAttribute("default-widget-style");
216   if (!menuElement.getAttribute("default-tooltip-style").isEmpty())
217       defaultTooltipStyle = menuElement.getAttribute("default-tooltip-style");
218   if (!menuElement.getAttribute("default-menu-item-name").isEmpty())
219       defaultMenuItemName = menuElement.getAttribute("default-menu-item-name");
220   if (!menuElement.getAttribute("default-permission-operation").isEmpty())
221       defaultPermissionOperation = menuElement.getAttribute("default-permission-operation");
222   if (!menuElement.getAttribute("default-permission-entity-action").isEmpty())
223       defaultPermissionEntityAction = menuElement.getAttribute("default-permission-entity-action");
224   if (!menuElement.getAttribute("default-associated-content-id").isEmpty())
225       defaultAssociatedContentId = FlexibleStringExpander.getInstance(menuElement
226               .getAttribute("default-associated-content-id"));
```

```
226                 .getAttribute("default-associated-content-id"));
227         if (!menuElement.getAttribute("orientation").isEmpty())
228             orientation = menuElement.getAttribute("orientation");
229         if (!menuElement.getAttribute("menu-width").isEmpty())
230             menuWidth = menuElement.getAttribute("menu-width");
231         if (!menuElement.getAttribute("default-cell-width").isEmpty())
232             defaultCellWidth = menuElement.getAttribute("default-cell-width");
233         if (!menuElement.getAttribute("default-hide-if-selected").isEmpty())
234             defaultHideIfSelected = "true".equals(menuElement.getAttribute("default-hide-if-selected").isEmpty());
235         if (!menuElement.getAttribute("default-disabled-title-style").isEmpty())
236             defaultDisabledTitleStyle = menuElement.getAttribute("default-disabled-title-style");
237         if (!menuElement.getAttribute("selected-menuitem-context-field-name").isEmpty())
238             selectedMenuItemContextFieldName = FlexibleMapAccessor.getInstance(menuElement
239                     .getAttribute("selected-menuitem-context-field-name"));
240         if (!menuElement.getAttribute("menu-container-style").isEmpty())
241             menuContainerStyleExdr = FlexibleStringExpander.getInstance(menuElement.getAttribute("menu-container-style"));
242         if (!menuElement.getAttribute("default-align").isEmpty())
243             defaultAlign = menuElement.getAttribute("default-align");
244         if (!menuElement.getAttribute("default-align-style").isEmpty())
245             defaultAlignStyle = menuElement.getAttribute("default-align-style");
246         if (!menuElement.getAttribute("fill-style").isEmpty())
247             fillStyle = menuElement.getAttribute("fill-style");
248         if (!menuElement.getAttribute("extra-index").isEmpty())
249             extraIndex = FlexibleStringExpander.getInstance(menuElement.getAttribute("extra-index"));
```

- **Cohesion** is adequate, since the class purpose is to create an XML and that is exactly what it does. However, **coupling** may be an issue because every field in the constructor (and thus every private final attribute) is strictly linked to the information carried by the *menuElement* and *menuLocation* classes (which are the fields required by the constructor).

```
105   public ModelMenu(Element menuElement, String menuLocation) {
```

- The last two functions grouped [Lines 489-529, *renderMenuString, renderSimpleMenuString*], create strings starting from the menu. Since all the fields can be accessed through getters and **the scope of the function** is not fully in line with the scope of the class, these two functions might be exported in a different class.
  Furthermore, the first class just check the type of the class and, if it's *simple*, it calls the second class, otherwise sends an error message. That would be easily managed even using only the second class, throwing an exception in case the type is different from the expected one.

2.10 Initialization and Declarations

In this section we look at the declaration of the variables as well as the initialization. We check the order in which they are declared, their visibility, scope and if they are correctly used inside the class.

```
58      public static final String module = ModelMenu.class.getName();
59
60      private final List<ModelAction> actions;
61      private final String defaultAlign;
62      private final String defaultAlignStyle;
63      private final FlexibleStringExpander defaultAssociatedContentId;
64      private final String defaultCellWidth;
65      private final String defaultDisabledTitleStyle;
66      private final String defaultEntityName;
67      private final Boolean defaultHideIfSelected;
68      private final String defaultMenuItemName;
69      private final String defaultPermissionEntityAction;
70      private final String defaultPermissionOperation;
71      private final String defaultSelectedStyle;
72      private final String defaultTitleStyle;
73      private final String defaultTooltipStyle;
74      private final String defaultWidgetStyle;
75      private final FlexibleStringExpander extraIndex;
76      private final String fillStyle;
77      private final String id;
78      private final FlexibleStringExpander menuContainerStyleExdr;
```

Figure 2.10 A: Variables declaration part 1

```
88      private final List<ModelMenuItem> menuItemList;
89      /** This Map is keyed with the item name and has a ModelMenuItem for the value; items
90       * with conditions will not be put in this Map so item definition overrides for items
91       * with conditions is not possible.
92       */
93      private final Map<String, ModelMenuItem> menuItemMap;
94      private final String menuLocation;
95      private final String menuWidth;
96      private final String orientation;
97      private final ModelMenu parentMenu;
98      private final FlexibleMapAccessor<String> selectedMenuItemContextFieldName;
99      private final String target;
100     private final FlexibleStringExpander title;
101     private final String tooltip;
102     private final String type;
```

Figure 1.10 B: Variables declaration part 2

In the class all variables are declared **private**. Seeing that they are all also final their visibility could be declared *public* since the values cannot be changed once there are assigned in the constructor upon creation. However, getters are available for them to be read from outside the class.

```
104     /** XML Constructor */
105     public ModelMenu(Element menuElement, String menuLocation) {
106         super(menuElement);
107         ArrayList<ModelAction> actions = new ArrayList<ModelAction>();
108         String defaultAlign = "";
109         String defaultAlignStyle = "";
110         FlexibleStringExpander defaultAssociatedContentId = FlexibleStringExpander.getInstance("");
111         String defaultCellWidth = "";
112         String defaultDisabledTitleStyle = "";
113         String defaultEntityName = "";
114         Boolean defaultHideIfSelected = Boolean.FALSE;
115         String defaultMenuItemName = "";
116         String defaultPermissionEntityAction = "";
117         String defaultPermissionOperation = "";
118         String defaultSelectedStyle = "";
119         String defaultTitleStyle = "";
120         String defaultTooltipStyle = "";
121         String defaultWidgetStyle = "";
122         FlexibleStringExpander extraIndex = FlexibleStringExpander.getInstance("");
123         String fillStyle = "";
124         String id = "";
125         FlexibleStringExpander menuContainerStyleExdr = FlexibleStringExpander.getInstance("");
126         ArrayList<ModelMenuItem> menuItemList = new ArrayList<ModelMenuItem>();
127         Map<String, ModelMenuItem> menuItemMap = new HashMap<String, ModelMenuItem>();
128         String menuWidth = "";
129         String orientation = "horizontal";
130         FlexibleMapAccessor<String> selectedMenuItemContextFieldName = FlexibleMapAccessor.getInstance("");
131         String target = "";
132         FlexibleStringExpander title = FlexibleStringExpander.getInstance("");
133         String tooltip = "";
134         String type = "";
```

Figure 2.10 C: Local variables in the constructor

Inside the constructor, as it can be seen from the image, we have the declaration of **local variables**, one for each of the variables of the class for later on to be used for their assignment. However, the declaration of these local variables can be avoided and we can do a straight assignment of the values.

```
122         FlexibleStringExpander extraIndex = FlexibleStringExpander.getInstance("");
```

When declaring the FlexibleStringExpander variables, a constructor is not called. The variables of this type are initialized through the method getInstance as seen above.

```
136         ModelMenu parent = null;
```

```
141             try {
142                 parent = MenuFactory.getMenuFromLocation(parentResource, parentMenu);
143             } catch (Exception e) {
144                 Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                         + "'", module);
146             }
```

Figure 2.10 D: Parent variable not initialized

The parent variable is not initialized before use, which could cause problems when used.

Initialization of the arrays should be done at the moment of declaration instead of inside the constructor.

The rest of the variables declared at the beginning of the class are not initialized in that moment but that's due to the fact that they are all dependent upon a computation so that is correctly done.

Also, the declaration of the variables is done at the beginning of the class and at the beginning of blocks of code as it should be.

2.11 Method Calls

In this section we check the use of the methods called inside the class.

In the methods called, parameters are passed in the correct order. The return values of the methods are correctly used.

```
110         FlexibleStringExpander defaultAssociatedContentId = FlexibleStringExpander.getInstance("");
```

There are several calls of the method shown in the image above, where the empty string is given as input to the getInstance method. This is done with the intention of later on assigning the empty string to these variables, but this is not the correct way to call the method given that it could return undesired content.

2.12 Arrays

The array indexing is handled correctly.

Arrays are also trimmed to sizes to save memory space.

Constructors are called when a new array element is needed.

2.13 Object Comparison

There is only one comparison that requires **==**, and it is correctly inserted:

```
160     if (parent == null) {
161         Debug.logError("Failed to find parent menu definition '" + parentMenu +
162         "' in same document.", module);
```

2.14 Output Format

There is no displayed output, and the error messages are comprehensible, grammatically correct and explain what is the problem. They do not provide how to correct the problem, but that is easily understandable by reading the error message.

2.15 Computation, Comparisons and Assignments

The implementation is not completely free from **brutish programming**. In fact, in the constructor, the creation and assignment of new temporary variables is unnecessary: if the parent exists, the assignment is made between lines 165-197, while if it doesn't, the assignment is done from line 198 to 249. Then, the static final values are assigned with the values their temporary counterparts have. Those assignments could have been done directly, since they are done exactly once. Thus, the assignment of variables done inside the if statements could have been done directly assigning the private final attributes of the class.

Parenthesis are generally correctly put, all the *if* and *else* statements and *for* loops brackets are aligned and set correctly.

One exception is the list o *if* statements from line 198 until line 249. An example below:

```
198     if (!menuElement.getAttribute("type").isEmpty())
199         type = menuElement.getAttribute("type");
```

The if statements consequences are not enveloped in brackets, which is not a fault in this cases because, when the if clause is found to be true, the program executes the next line. Problems may rise afterward, though, in the prospect that the class will be modified: a careless fix may ignore the necessity to put brackets causing faults in the performance.

All the expressions that result in a **Boolean** (for instance, !parentMenu.isEmpty()), are rightly put.

Example: if (!parentMenu.isEmpty(), line 139, correctly states that the if statement is entered in the case that the parentMenu instance is not empty.

Between line 396-402 a **try-catch** does not specify which kind of exception is found, does not specify what has happened, what might be done to solve it. Moreover, since it returns " ", it may not even be clear if an error occurred or not.

```
396     public String getExtraIndex(Map<String, Object> context) {
397         try {
398             return extraIndex.expandString(context);
399         } catch (Exception ex) {
400             return "";
401         }
402     }
```

The code does not have any implicit type conversions.

2.16 Exceptions

In this section we check the exceptions thrown in the class: we check if the correct exceptions are thrown and if the they are handled correctly.

```
157                 if (parent == null) {
158                     Debug.logError("Failed to find parent menu definition '" + parentMenu + "' in same document.", module);
159                 }
160             }
161             if (parent != null) {
162                 type = parent.type;
163                 target = parent.target;
164                 id = parent.id;
165                 title = parent.title;
166                 tooltip = parent.tooltip;
167                 defaultEntityName = parent.defaultEntityName;
168                 defaultTitleStyle = parent.defaultTitleStyle;
169                 defaultSelectedStyle = parent.defaultSelectedStyle;
170                 defaultWidgetStyle = parent.defaultWidgetStyle;
171                 defaultTooltipStyle = parent.defaultTooltipStyle;
172                 defaultMenuItemName = parent.defaultMenuItemName;
173                 menuItemList.addAll(parent.menuItemList);
174                 menuItemMap.putAll(parent.menuItemMap);
175                 defaultPermissionOperation = parent.defaultPermissionOperation;
176                 defaultPermissionEntityAction = parent.defaultPermissionEntityAction;
177                 defaultAssociatedContentId = parent.defaultAssociatedContentId;
178                 defaultHideIfSelected = parent.defaultHideIfSelected;
179                 orientation = parent.orientation;
180                 menuWidth = parent.menuWidth;
181                 defaultCellWidth = parent.defaultCellWidth;
182                 defaultDisabledTitleStyle = parent.defaultDisabledTitleStyle;
```

Figure 2.16 A: Missing Exception

A *NullPointerException* should be thrown instead of doing the two check of the object parent.

```
141             try {
142                 parent = MenuFactory.getMenuFromLocation(parentResource, parentMenu);
143             } catch (Exception e) {
144                 Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                         + "'", module);
146             }
```

Figure 2.16 B: Generic Exception

The exception thrown is very **generic**, something more specific is needed.

```
500    public void renderMenuString(Appendable writer, Map<String, Object> context, MenuStringRenderer menuStringRenderer)
501            throws IOException {
502        AbstractModelAction.runSubActions(this.actions, context);
503        if ("simple".equals(this.type)) {
504            this.renderSimpleMenuString(writer, context, menuStringRenderer);
505        } else {
506            throw new IllegalArgumentException("The type " + this.getType() + " is not supported for menu with name "
507                    + this.getName());
508        }
509    }
```

Figure 2.16 C: Mismatching exceptions

Different exception is thrown (*IllegalArguemntException*) from the one declared in the method declaration (*IOException*) and they are not compatible.

```
511    public void renderSimpleMenuString(Appendable writer, Map<String, Object> context, MenuStringRenderer menuStringRenderer)
512            throws IOException {
513        // render menu open
514        menuStringRenderer.renderMenuOpen(writer, context, this);
515
516        // render formatting wrapper open
517        menuStringRenderer.renderFormatSimpleWrapperOpen(writer, context, this);
518
519        // render each menuItem row, except hidden & ignored rows
520        for (ModelMenuItem item : this.menuItemList) {
521            item.renderMenuItemString(writer, context, menuStringRenderer);
522        }
523        // render formatting wrapper close
524        menuStringRenderer.renderFormatSimpleWrapperClose(writer, context, this);
525
526        // render menu close
527        menuStringRenderer.renderMenuClose(writer, context, this);
528    }
```

Figure 2.16 D: Exceptioin not handled

The IOException is declared but not handled.

```
141                try {
142                    parent = MenuFactory.getMenuFromLocation(parentResource, parentMenu);
143                } catch (Exception e) {
144                    Debug.logError(e, "Failed to load parent menu definition '" + parentMenu + "' at resource '" + parentResource
145                            + "'", module);
146                }
```

Figure 2.16 E: Exception not handled well

Nothing is done in the catch block apart from the message with the cause of the exception.

```
290    @Override
291    public void accept(ModelWidgetVisitor visitor) throws Exception {
292        visitor.visit(this);
293    }
```

In this case we have a very generic exception declared in the method (it could be due to the method that it's overriding, so the fact that it's not handled here could not be a problem if the parent handles it).

2.17 Flow of Control

There are no switch statements, and all loops are correctly formed and terminated.

2.18 Files

There are no files used in the class, so there are no issues created related to files.

3 Individual Effort

Agnese Bruschi – chapters 2.9, 2.13, 2.14, 2.15, 2.17, Layout, Introduction 7 hours

Daniel Botta – from chapters 2.1 to 2.9, Others, 6 hours

Adriana Cano - chapters 2.10, 2.11, 2.12, 2.16, 2.18: 6 hours