POLITECNICO
MILANO 1863

Code Inspection

Version 1.0

28/01/2017

Software Engineering 2 (A.A. 2016/2017)

- Simone Boglio (mat. 772263)
- Lorenzo Croce (mat. 807833)

Summary

1 Classes Assigned

We are inspecting a piece of code from Apache OFBiz, version 16.11.01.
We were assigned the following class:

ProductFeatureServices.Java

The class is located in the following path:

/applications/product/src/main/java/org/apache/ofbiz/product/feature/

2 Functional Role of Assigned Set of Classes

The ProductFeatureServices are a static class that offers methods to search and retrieve information about similar products.

All four methods return the same type of result, Map<String, Object>, whereas String is used a key value string and the object is the result of the computation.
The four methods analysed are:

- getProductFeaturesByType(DispatchContext, Map<String, ? extends Object>)
  This method allows to search product features by three different parameters: the productFeatureCategoryId, the productFeatureGroupId, the productId (with the latter it is possible to use the optional parameter productFeatureApplTypeId to apply an additional filter).
  If the parameter were productFeatureCategoryId, the results are from ProductFeatures.
  If productFeatureCategoryId were null and there were a productFeatureGroupId, the results are from ProductFeatureGroupAndAppl.
  If there is a productId, the results are from ProductFeatureAndAppl.
  The method returns as object a list of productFeature.

- getAllExistingVariants(DispatchContext, Map<String, ? extends Object>)
  This method receives two parameters: productId and a list of productFeatureAppls.
  The method returns as object list of ProductId of products that are variants of the productId with at least all the productFeatureAppls.

- getVariantCombinations(DispatchContext, Map<String, ? extends Object>)
  This method receives one parameter: productId.
  The method returns as object a list of maps containing as key each possible variant of the productID.
  For each variant, there are a list with its productFeaturesAndAppls and a list of id of the products that are its variant.

- getCategoryVariantProducts(DispatchContext, Map<String, ? extends Object>)
  This method receives two parameters: productCategoryId and a list of productFeatures.
  The method returns as object a list of variant products of the products that have the categoryId and the features in the ProductFeatures list.

All this method retrieve data from a database using the EntityQuery component, after this, they further manipulate the data and return the requested information to the caller.

3 List of Issues

3.1 Naming Conventions

7. Constants are declared using all uppercase with words separated by an underscore.
The following scratch of code shows that two constants doesn't respect the previous convention.

```
52        public static final String module = ProductFeatureServices.class.getName();
53        public static final String resource = "ProductUiLabels";
```

3.2 Indention

No problems found about this section.

3.3 Braces

10. The bracing style used is the "Kernighan and Ritchie" style.
The following scratch of code is an example of the previous convention.

```
108     if (features == null) {
109         features = new LinkedList<GenericValue>();
110         featuresByType.put(featureType, features);
111     }
```

11. Al if statements that have only one statement to execute are surrounded by curly braces.
The following scratch of code shows that this if statement doesn't respect the previous convention.

```
97      if (entityToSearch.equals("ProductFeatureAndAppl") && productFeatureApplTypeId != null)
98          allFeatures = EntityUtil.filterByAnd(allFeatures, UtilMisc.toMap("productFeatureApplTypeId", productFeatureApplTypeId));
```

3.4 File Organization

12. Blank lines and optional comments are used to separate sections.
Blank lines are not used in the correct way at lines: *29, 49, 72, 76, 88, 92, 96, 99, 114, 136, 152, 161, 184, 186, 190, 264, 287, 298, 311, 319, 323, 330, 338, 345, 349*.

13. Where practical, line length does not exceed 80 characters.
In many cases line length exceeds 80 characters

14. When line length must exceed 80 characters, it does not exceed 120 characters.
At the following lines the line length exceeds 120 characters: *57, 58, 59, 60, 62, 67, 86, 95, 98, 126, 129, 139, 140, 143, 153, 179, 181, 188, 197, 226, 244, 245, 247, 249, 266, 273, 295, 308, 317, 325, 327, 335, 340*.

3.5 Wrapping Lines

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.
The following scratch of code shows that the previous condition is violated.

```
172             }
173         return results;
174         }
```

3.6 Comments

19. Commented out code contains reason for begin commented out and a date it can be removed from the source file if determined it is no longer needed.
The following scratch of code shows a comment that can be removed because ".clone()" is never used in the rest of the code.

```
242       // .clone() is important, or you'll keep adding to the same List for all the variants
243       // have to cast twice: once from get() and once from clone()
```

Others:
If a comment is written over multiple lines, it is best to use this annotation:
/ line1*
** line2*
**/*

The following scratches shows two blocks of comments that can be written with the correct annotation.

```
197     // need to keep 2 lists, oldCombinations and newCombinations, and keep swapping them after each looping.  Otherwise, you'll get a
198     // concurrent modification exception
```

```
215     // in both cases, use each feature of current feature type's idCode and
216     // product feature and add it to the id code and product feature applications
217     // of the next variant.  just a matter of whether we're starting with an
218     // existing list of features and id code or from scratch.
```

3.7 Java Source Files

23. Check that the Javadoc is complete.
The following scratch of code shows the only Javadoc annotation is the class description but it is incomplete.

```
46    /**
47     * Services for product features
48     */
```

The class methods are all commented but not in a Javadoc format and this doesn't respect the convention.

3.8 Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.
The following scratch of code shows that there is a useless import statement “LinkedHashMap”

```
23    import java.util.LinkedHashMap;
```

3.9 Class and Interface Declarations

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.
The methods in this class are rather long. In particular, the third method called “getVariantCombinations” (Line 181) is too long and complex. Analysing this method with sonar we obtained a cognitive complexity equals to 50.

```
181       public static Map<String, Object> getVariantCombinations(DispatchContext dctx, Map<String, ? extends Object> context) {
```

3.10 Initialization and Declarations

30. Check that constructors are called when a new object is desired.
All the strings are not initialized by the constructor “new String()” but the compiler accept also an initialization like the following:
String stringName = “string”;

33. Declaration appear at the beginning of blocks. The exception is a variable can be declared in a for loop.
The following variables aren’t declared at the beginning of blocks and this can cause problem during the read of the code.

```
100               List<String> featureTypes = new LinkedList<String>();
101               Map<String, List<GenericValue>> featuresByType = new LinkedHashMap<String, List<GenericValue>>();
```

```
107      List<GenericValue> features = featuresByType.get(featureType);
```

```
199     List<Map<String, Object>> oldCombinations = new LinkedList<Map<String,Object>>();
```

```
265      int defaultCodeCounter = 1;
266      Set<String> defaultVariantProductIds = new HashSet<String>();
```

```
312     List<GenericValue> memberProducts = UtilGenerics.checkList(result.get("categoryMembers"));
```

```
320    List<GenericValue> products = new LinkedList<GenericValue>();
```

```
331     List<GenericValue> variantProducts = UtilGenerics.checkList(result.get("products"));
```

3.11 Method Calls
No problems found about this section.

3.12 Arrays
No problems found about this section.

3.13 Object Comparison
No problems found about this section.

3.14 Output Format
41. Check that displayed output is free of spelling and grammatical errors.
All the output messages are managed by logger in this code.
The following scratch of code shows that the only grammatical error is “memebers” instead of “members”.

```
308     Debug.logError("Cannot get category memebers for " + productCategoryId + " due to error: " + ex.getMessage(), module);
```

3.15 Computation, Comparison and Assignments
No problems found about this section.

3.16 Exception
No problems found about this section.

3.17 Flow of Control
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expression.
All the loops of our code are generalized for and only one can be reported.
It is better to check the condition “hasAllFeatures” into the for declaration and avoid the break instruction to exit from the loop.
The following scratch of code shows this for cycle.

```
146    //for each associated product, if it has all standard features, display it's productId
147    boolean hasAllFeatures = true;
148    for (String productFeatureAndAppl: curProductFeatureAndAppls) {
149        Map<String, String> findByMap = UtilMisc.toMap("productId", productAssoc.getString("productIdTo"),
150                "productFeatureId", productFeatureAndAppl,
151                "productFeatureApplTypeId", "STANDARD_FEATURE");
152
153        List<GenericValue> standardProductFeatureAndAppls = EntityQuery.use(delegator).from("ProductFeatureAppl").where(findByMap).filterByDate().queryList();
154        if (UtilValidate.isEmpty(standardProductFeatureAndAppls)) {
155            hasAllFeatures = false;
156            break;
157        } else {
158
159        }
160    }
```

3.18 Files
No problems found about this section.

4 Other Issues

4.1 Relevant Issues

In the following points, there are relevant issues found in the code of the class:

- There is not a private constructor in the static class.
  The class ProductFeatureServices is a static class so is a good practice to declare the construct as private, otherwise the compiler uses the default Java constructor with no parameters, that lack permit to instantiate objects of this class in runtime.
  Insert these lines of code to solve the problem:
  private ProductFeatureServices(){}

- The code is full of string in the format "stringName" that are repeated many times. It's a good practice to create, for every of these string, a constant and use it where it is necessary. In this way if the value need to change there is to edit only one value and there is not risk to miss some values inside the whole code.

Variable	**Occurrences**	**Lines**
productId	*13*	*82, 84, 85, 133, 143, 149, 185, 188, 277, 325, 325, 327, 335*
defaultVariantProductId	*10*	*226, 228, 247, 247, 249, 249, 272, 273, 273, 276*
productFeatureApplTypeId	*6*	*71, 86, 98, 151, 221, 240*
idCode	*4*	*225, 226, 246, 247*
productFeatureTypeId	*4*	*75, 86, 103, 317*
curProductFeatureIds	*4*	*233, 245, 254, 278*
productFeatureGroupId	*3*	*77, 79, 80*
curProductFeatureAndAppls	*3*	*232, 244, 253*

- Useless assignment to a variable: the value assigned is never used and overwritten afterwards with a computation.
  Line with this problem: *63, 130, 182.*

4.2 Less Relevant Issues

In the following points, there are other “bad practice” issues found in the code of the class:

- After the declaration of a Collection (List for example) it is useless rewrite the element type of the collection (unless before Java 7, but this project is built on Java 8).
  List<String> stringList = new List<String>(); NO!
  List<String> stringList = new List<>(); OK!
  Lines whit this problem: *63, 97, 100, 101, 109, 130, 135, 182, 189, 199, 205, 212, 222, 223, 224, 241, 266, 296, 315, 320*.

- When two string must be compared using the equal method, if one of the two string is in the format “StringaToCompare” it is best to write the instruction in the following way:
  “StringaToCompare”.equals(String);
  Instead of:
  String.equal(“StringToCompare”);
  Lines with this problem: *97, 221, 240*.

- There are some lines of code where is preferable to use the method “.isEmpty()” in the if condition instead of:
  “.size() == 0” or “.size() > 0”
  Lines with this problem: *209, 219, 313, 332, 339.*

- Concatenate the empty string “” with primitive type like in the following scratch of code.

```
273    combination.put("defaultVariantProductId", combination.get("defaultVariantProductId") + "-" +
       (defaultCodeCounter < 10? "0" + defaultCodeCounter: "" + defaultCodeCounter));
```

  Instead, it is better to use:
  Integer.ToString(defaultCodeCounter)

5 Appendix

5.1 External References

- Official Java code convention: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

5.2 Software and Tools Used

- IntelliJ IDEA CE: to inspect the class.
- SonarLint (IntelliJ plug-in): tool for code analysis and code quality.
- GitHub: use the differential to find space and tab indentation problems.

5.3 Hours of Work

- Boglio Simone: 18 hours of work.
- Croce Lorenzo: 15 hours of work.