POLITECNICO
DI MILANO

Computer Science and Engineering
A.A. 2016/2017
Software Engineering 2 Project:
Power&Joy

Code Inspection Document

February 5, 2017

Prof. Luca Mottola
Joshua Nicolay Ortiz Osorio Matr: 806568
Michelangelo Medori Matr: 878025

Index

1 Introduction

1.1 Purpose and Scope

This is the Code Inspection Document for the Software Engineering 2 course project. The aim of this document is to report a list of all the issues that the members of our project group managed to find into some source code that has been assigned to us. As described in section 2, in our case the source code consists of a single class, taken from the Apache ofbiz project. Section 2 contains also contains a brief description of the functional role of this class. Section 3 contains the list of the issues found; they are listed following the order that has been given to us on the assignments documents.

1.2 Reference Documents

- Assignments AA 2016-2017.pdf
- Code Inspection Assignment Task Description.pdf
- IterateSectionWidget.java
- apache-ofbiz-16.11.01 source code

2 Classes assigned to the group

We have being assigned a single class , taken from the Apache ofbiz source code. Follows name and namespace of the class, and a description of its functional role

2.1 Namespace patterns and names

namespace
../apache-ofbiz-16.11.01/framework/widget/src/main/java/org/apache/ofbiz/widget/model/IterateSectionWidget.java

name of the class
IterateSectionWidget.java

2.2 Functional role of the assigned classes

The selected class is used to manage the widget sections of the Apache Ofbiz Application. A widget is a graphical element that provides immediate access to some of the main functionalities of the application, commonly used as shortcuts. The Apache Ofbitz application provides a list of widgets that users can customize as they prefer. The IterateSectionWidget.java class provides methods to iterate through the widgets sections and display them, by means of writing an XML file with data accessed via a Map structure.

3 Issues found applying the checklist

3.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

The method "accept " declared at lines 361 to 363 should have a more meaningful name:

```
public void accept(ModelWidgetVisitor visitor) throws Exception {
        visitor.visit(this);
    }
```

2. If one-character variables are used, they are used only for temporary ?throwaway? variables, such as those used in for loops.

There is only a one-character array variable declared at line 135:

```
Object [] a = entrySet.toArray();
```

but is soon converted to a List at line 136:

```
theList = Arrays.asList(a);
```

which means it's a throwaway variable, and does not represent an issue.

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized.

IterateSectionWidget is the name of a class and contains the verb Iterate. Anyway it could be considered as a noun and does not represent an issue.

4. Interface names should be capitalized like classes.

The only intrface name that appears is ModelScreenWidget, and it respect the constrain

5. Method names should be verbs, with the first letter of each addition word capitalized

All methods names are verbs

6. Class variables, also called attributes, are mixed case, but might begin with an underscore (? ?) followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized.

There are not variables that begin with an underscore. They all begin with a lowercase letter, and do not repreesnt an issue

7. Constants are declared using all uppercase with words separated by an underscore

All constants names are capitalized and contain underscores to separate the words

3.2 Indention

8. Three or four spaces are used for indentation and done so consistently.

Identation is used correctly

9. No tabs are used to indent.

No tabs are used

3.3 Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

The "Allman" style is used in the whole document

11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces.

No curly braces for single 'if' instructions are used at the following lines:

- lines 318-319 :

```
if (linkText.indexOf("?") < 0)  linkText.append("?");
            else linkText.append("&amp;");
```

- lines 340-341:

```
if (linkText.indexOf("?") < 0)  linkText.append("?");
            else linkText.append("&amp;");
```

3.4 File Organization

12. Blank lines and optional comments are used to separate sections (be- ginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

blank lines are consistently used to separate sections.

13. Where practical, line length does not exceed 80 characters.

These are the lines that exceed the length of 80 characters

?Lines: 70, 87, 88, 95, 99, 101, 113, 134, 151, 152, 232, 243, 249, 250, 257, 324, 327, 328, 333, 334, 335, 345, 348, 349.

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

The following lines exceed the length of 120 characters: 113, 324, 333, 334, 345

3.5 Wrapping Lines

15. Line break occurs after a comma or an operator

Line breaks are never used after a comma or an operator

16. Higher-level breaks are used.

Higher level breaks are used to allow readability

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

New statements are always aligned with the beginning of the expression at the same level as the previous line.

3.6 Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

There are not enough comments to explain the function of the class, variables, attributes, methods , blocks of code and method calls.

The only two useful comments are found at lines 121, 300.

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

There are two blocks of commented out code, and none of them contains a date or an explanation for it's presence in the file.

In particular, the following commented out code block at lines 282-296

```
/*
      int highIndex = -1;
      try {
          highIndex = modelForm.getHighIndex();
      } catch (Exception e) {
          highIndex = 0;
      }

      int lowIndex = -1;
      try {
          lowIndex = modelForm.getLowIndex();
      } catch (Exception e) {
          lowIndex = 0;
      }
       */
```

which is clearly an alternative to the non-commentd-out block of code it

follows (lines 267-289)

```
int viewIndex = -1;
      try {
          viewIndex = ((Integer) context.get("viewIndex")).intValue();
      } catch (Exception e) {
          viewIndex = 0;
      }

      int viewSize = -1;
      try {
          viewSize = ((Integer) context.get("viewSize")).intValue();
      } catch (Exception e) {
          viewSize = this.getViewSize();
      }
```

looks quite inappropriate.

An other commented out code block are found at lines 320

```
//if (queryString != null && !queryString.equals("null"))
linkText += queryString + "&";
```

3.7 Java Source Files

20. Each Java source file contains a single public class or interface.

Th java IterateSectionWidget.java file contains only a public class

21. The public class is the first class or interface in the file.

There is only one public class

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

Javadoc is not complete

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Javadoc for the class ItrateSectionWidget.java is not complete

3.8 Package and Import Statements

24. If any package statements are needed, they should be the first non comment statements. Import statements follow.

Package and import statements are in the correct order

3.9 Class and Interface Declarations

25. Check the order of the class and interface declarations

The order is respected.

26. Methods are grouped by functionality rather than by scope or accessibility.

Mthods are grouped by functionality.

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 code at lines 324-329

```
 linkText.append("VIEW_SIZE_"+ paginatorNumber + "=").
 append(viewSize).append("&amp;VIEW_INDEX_" +
 paginatorNumber + "=").append(viewIndex - 1).append("\"");

// make the link
writer.append(rh.makeLink(request, response,
linkText.toString(), false, false, false));
String previous = UtilProperties.
getMessage("CommonUiLabels", "CommonPrevious",
(Locale) context.get("locale"));
writer.append(" class=\"buttontext\">[").append(previous).
append("]</a>\n");
```

are duplicated at lines 345-350

3.10 Initialisation and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

All variables are of correct type and have the right visibility

29. Check that variables are declared in the proper scope.

All variables are declared in the proper scope

30. Check that constructors are called when a new object is desired.

The constructors for the inputFields and queryStringMap, request, response, ctx and rh variables are missing :

- line 249:

```
Map<String, Object> inputFields =
UtilGenerics.checkMap(context.get("requestParameters"));
```

- line 250:

```
Map<String, Object> queryStringMap =
UtilGenerics.toMap(context.get("queryStringMap"));
```

- line 305:

```
HttpServletRequest request =
 (HttpServletRequest) context.get("request");
```

- line 306:

```
HttpServletResponse response =
(HttpServletResponse) context.get("response");
```

- line 308 and 309:

```
ServletContext ctx =
(ServletContext) request.getAttribute("servletContext");

RequestHandler rh =
(RequestHandler) ctx.getAttribute("_REQUEST_HANDLER_");
```

All other constructors are correctly called.
Sometimes the constructor, instead of being called directly, is called by a method of another class. For example the constructors for the following object parameters (declared at lines 48-52)

```
private final FlexibleMapAccessor<Object> listNameExdr;
private final FlexibleStringExpander entryNameExdr;
private final FlexibleStringExpander keyNameExdr;
private final FlexibleStringExpander paginateTarget;
private final FlexibleStringExpander paginate;
```

are called inside the relative getIstance() methods as follows:

- line 76:

```
this.listNameExdr =
FlexibleMapAccessor.getInstance(listName);
```

- line 81 :

```
this.entryNameExdr =
FlexibleStringExpander.getInstance(entryName);
```

- line 86:

```
this.keyNameExdr =
FlexibleStringExpander.getInstance(keyName);
```

- line 87:

```
this.paginateTarget = FlexibleStringExpander.
getInstance(iterateSectionElement.getAttribute("paginate-target"));
```

- line 88:

```
this.paginate = FlexibleStringExpander.
getInstance(iterateSectionElement.getAttribute("paginate"));
```

The getIstance() method checks if existing istances for these objects already exist, otherwise creates and returns a new instance for each object (that's because they are declared as final). The getInstance method for the class FlexibleMapAccessor is reported below (lines 87-97)

```
public static <T> FlexibleMapAccessor<T> getInstance(String original) {
      if (UtilValidate.isEmpty(original) || "null".equals(original)) {
          return nullFma;
      }
      FlexibleMapAccessor fma = fmaCache.get(original);
      if (fma == null) {
          fmaCache.putIfAbsent(original, new FlexibleMapAccessor(original));
          fma = fmaCache.get(original);
      }
      return fma;
  }
```

The parameter

```
private final List<ModelScreenWidget.Section> sectionList;
```

is constructed at line 101:

```
ModelScreenWidget.Section section =
new ModelScreenWidget.Section(modelScreen, sectionElement, false);
```

The childElementList is a linked list which is created and returned by the childElementList() method of the UtilXml class

```
List<? extends Element> childElementList =
UtilXml.childElementList(iterateSectionElement);
```

the contextMs object is created as follows:
line 123:

```
MapStack<String> contextMs = MapStack.create(context);
```

31:Check that all object references are initialized before use.

All object references are correctly initialized before use.

32: Variables are initialized where they are declared, unless dependent upon a computation.

All variables are correctly initialized before being used.

33 Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces . The exception is a variable can be declared in a for loop. Not all variables are declared at the beginning of

the block they belong.

For the variables declared inside the constructor of the class:

- line 72:

  ```
  String listName = iterateSectionElement.getAttribute("list");
  ```

- line 77:

  ```
  String entryName = iterateSectionElement.getAttribute("entry");
  ```

- line 82:

  ```
  String keyName = iterateSectionElement.getAttribute("key");
  ```

- line 89:

```
int viewSize = DEFAULT_PAGE_SIZE;
```

- line 90:

```
String viewSizeStr = iterateSectionElement.getAttribute("view-size");
```

- line 95:

```
List<? extends Element> childElementList =
UtilXml.childElementList(iterateSectionElement);
```

For the renderWidgetString method() :

- line 146:

```
int startPageNumber = WidgetWorker.getPaginatorNumber(context);
```

- line 171:

```
Iterator<?> iter = theList.iterator();
```

- line 172:

```
int itemIndex = -1;
```

- line 173:

```
int iterateIndex = 0;
```

3.11 Method Calls

34. Check that parameters are presented in the correct order.

All parameters are sent by methods in the right order

35. Check that the correct method is being called, or should it be a dif- ferent method with a similar name.

All methods calls are correct

36. Check that method returned values are used properly.

All the items returned by method calls are coherent with the expected values

3.12 Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

Arrays are not scanned by loops. The only array instance is found at line 135:

```
Object [] a = entrySet.toArray();
```

created from a collection of elements (entrySet in this case) and returned by the toArray method of Java, which, as the documentation reports, prevents from bad indexing

"[The returned array] will be "safe" in that no references to it are maintained by this set. (In other words, this method must allocate a new array even if this set is backed by an array). The caller is thus free to modify the returned array."

Eventually, it is transformed into a List and is not directly accessed

```
theList = Arrays.asList(a);
```

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

Every time a collection is created or accessed, there are instructions that check if it is null.

For example the listNameExdr is initialized at line 76 calling the getInstance() method of the FlexibleMapAccessor class:

```
this.listNameExdr = FlexibleMapAccessor.getInstance(listName);
```

the getInstence method is reported below (lines 87-97), and contains all the needed instruction to prevent a bad use for the collection:

```
public static <T> FlexibleMapAccessor<T> getInstance(String original) {
     if (UtilValidate.isEmpty(original) || "null".equals(original)) {
         return nullFma;
     }
    FlexibleMapAccessor fma = fmaCache.get(original);
     if (fma == null) {
         fmaCache.putIfAbsent(original, new FlexibleMapAccessor(original));
         fma = fmaCache.get(original);
     }
     return fma;
}
```

All Iterators correctly access collections and sets defined.

39. Check that constructors are called when a new array item is desired.

Every time a collection or array is desired, the right constructor is called.

3.13 Object Comparison

40. Check that all objects (including Strings) are compared with equals and not with ==.

Some issues of bad comparison are to be found as follows:

- line 128:

  ```
  if (obj == null) {
  ```

bad comparison for object obj declared at line 127 as

```
Object obj = listNameExdr.get(context);
```

- line 209:

```
if (globalCtx != null) {
```

- line 214:

```
if (globalCtx != null) {
```

bad comparison for objectglobalCtx objct declared at line 208:

```
Map<String, Object> globalCtx =
UtilGenerics.checkMap(context.get("globalContext"));
```

- line 245:

```
if (targetService == null) {
```

bad comparison for String targetService, declared at line 244 as:

```
String targetService = this.getPaginateTarget(context);
```

3.14 Output Format

41. Check that displayed output is free of spelling and grammatical errors.

There is no displayed output

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.

All error messages written in the log are comprehensive and clear, for example:

- line 129:

```
Debug.logError("No object found for listName:" +
 listNameExdr.toString(), module);
```

- line 263:

```
Debug.logWarning("TargetService is empty.", module);
```

except for line 218:

```
Debug.logError(e, module);
```

where no meaningful message is written

43. Check that the output is formatted correctly in terms of line stepping and spacing.

All output format is formatted correctly

3.15 Computation, Comparisons and Assignments

44. Check that the implementation avoids brutish programming:

There is no brutish programming, for instance, when managing the page numbering (lines 201-205), instructions are self-explanatory:

```
if ((itemIndex + 1) < highIndex) {
        highIndex = itemIndex + 1;
    }
    actualPageSize = highIndex - lowIndex;
```

45. Check order of computation/evaluation, operator precedence and parenthesizing.

All operators precedences are respected, parenthesys are correctly used when concatenations of method calls are done

46: Check the liberal use of parenthesis is used to avoid operator precedence problems.

All needed parenthesys are correctly used to avoid issues that come from operator precedences

47. Check that all denominators of a division are prevented from being zero.

There are no divisions.

48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.

No arithmetic operation cause problems of truncation or unexpected rounding.

49. Check that the comparison and Boolean operators are correct.

All boolean comparisons are correct

50: Check throw-catch expressions, and check that the error condition is actually legitimate.

All throw and catch expressions are used properly (only if needed, and the error conditions are legitimate)

51. Check that the code is free of any implicit type conversions.

All types conversion are correctly done, explicitly using built in java methods such as toString(), parseInt,() toArray(), asList(). For example:

- line153:

```
viewIndex = Integer.parseInt(viewIndexString);
```

- line 135:

```
Object [] a = entrySet.toArray();
```

- line 136:

```
theList = Arrays.asList(a);
```

Anyway some bad casting is inside mthods that belong to the FlexiblMapAccessor class:

- line142

```
Map<String, Object> writableMap =
(Map<String, Object>) base;
```

(see also the comment at line 141)

```
// This method is a hot spot, so placing the cast here
instead of in another class.
```

3.16 Exceptions

52. Check that the relevant exceptions are caught.

All relevant exceptions are caught.

53. Check that the appropriate action are taken for each catch block.

The catch block at lines 217-220 propagates the exception without writing any relevant message on the log

```
} catch (IOException e) {
                Debug.logError(e, module);
                throw new RuntimeException(e.getMessage());
            }
```

3.17 Flow of Control

54. In a switch statement, check that all cases are addressed by break or return.

There are no switch statements.

55. Check that all switch statements have a default branch

There are no switch statements.

56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

All loops are formed correctly, for example the for loop at lines 100-103 that cycles among the elements of the list childElementList:

```
for (Element sectionElement: childElementList) {
  ModelScreenWidget.Section section =
  new ModelScreenWidget.Section(modelScreen, sectionElement, false);
   sectionList.add(section);

}
```

or the while at lines 174-199 that cycles over the elements of the list theList with the use of an iterator:

```
Iterator<?> iter = theList.iterator();
[..]
while (iter.hasNext()) {
[..]
}
```

3.18 Files

57. Check that all files are properly declared and opened.

The method renderNextPrev of the IterateSectionWidget class modifies an HTML file, but it receives the appendable writer object as input, and it does not have to open or close the file itself, since these actions are performed elsewhere in the source code. Thus there are no issues concerning files opening, closing, EOF conditions or exceptions.

58. Check that all files are closed properly, even in the case of an error.

See (57)

59. Check that EOF conditions are detected and handled correctly.

See (57)

60. Check that all file exceptions are caught and dealt with accordingly.
See (57)

4 Other Issues

There are no other issues in addition to the ones encountered and reported in section 3

5 Effort Spent

The following table shows the total number of hours spent working on the whole project for both members of our group.

Document	**Michelangelo Medori**	**Joshua Nicolay Ortiz Osorio**
RASD	28	30
DD	22	20
ITPD	30	30
PP	15	15
CID	10	10
TOTAL	**105**	**105**