POLITECNICO
MILANO 1863

PROJECT for SOFTWARE ENGINEERING 2

Code Inspection

Politecnico di Milano
A.A. 2016-2017
Prof.ssa Elisabetta Di Nitto

Students:
Diego Gaboardi
Giorgio Giardini
Riccardo Giol

INDEX

- 1 Assigned Classes pag 3
- 2 Functional Role of Assigned Classes pag 4
  - 2.1 ScrumServices pag 4
    - 2.1.1 ViewScrumRevision pag 5
    - 2.1.2 RetriveMissingRevision pag 6
    - 2.1.3 RemoveDuplicateScrumRevision pag 6
    - 2.1.4 LinkToProduct pag 7
  - 2.2 ValidateMethodCondition pag 9
- 3 List of Issues pag 11
  - 3.1 Class ScrumServices pag 11
    - 3.1.1 ViewScrumRevision pag 12
    - 3.1.2 RetrieveMissingRevision pag 13
    - 3.1.3 RemoveDuplicateScrumRevision pag 14
    - 3.1.4 LinkToProduct pag 15
  - 3.2 ValidateMethodCondition pag 17
- 4 Hours of work pag 19
- 5 Used tools pag 19

1 ASSIGNED CLASSES

The two assigned classes are:

ScrumServices:

../apache-ofbiz-16.11.01/specialpurpose/scrum/src/main/java/org/apache/ofbiz/scrum/ScrumServices.java

ValidateMethodCondition:

../apache-ofbiz-16.11.01/framework/minilang/src/main/java/org/apache/ofbiz/minilang/method/conditional/ValidateMethodCondition.java

2 FUNCTIONAL ROLE OF ASSIGNED CLASSES

2.1 SCRUM SERVICES

This class belongs to the Special Purpose stack which comprises several components and applications that extend the base functionalities offered by Apache OFBiz.

In particular it is situated in the scrum package that, as written in the documentation, *enables organisations to manage their product backlog and agile development projects* offering the following features:

- Resource assignment;
- Product backlog;
- Sprint management;
- Version management.

With agile software development, we refer to a group of software development methodologies based on iterative development in which requirement and solutions evolve through collaboration between self-organizing cross-functional teams.

Scrum is a subset of Agile that is often used to manage complex software and product development, using iterative and incremental practices. In particular, work is confined to a regular work cycle, known as a sprint or iteration.

In Scrum, each sprint is required to deliver a potentially shippable product increment. This means that at the end of each sprint, the team has produced a coded, tested and usable piece of software and a sprint review meeting is held in which the Scrum team shows what they accomplished during the sprint.

The ScrumServices class is composed by 4 static method that offer important functionalities for managing Scrum. In particular 3 of them are used for managing scrum revision:

- `public static Map<String, Object> viewScrumRevision(DispatchContext ctx, Map<String, ? extends Object> context)`

- `public static Map<String, Object> retrieveMissingScrumRevision(DispatchContext ctx, Map<String, ? extends Object> context)`

- ```
  public static Map<String, Object>
  removeDuplicateScrumRevision(DispatchContext ctx, Map<String, ?
  extends Object> context)
  ```

Then there is the last method which manages communication events, relative to specific products.

```
public static Map<String, Object> linkToProduct(DispatchContext
ctx, Map<String, ? extends Object> context)
```

In the following paragraph these method will be analysed more in detail.

2.1.1 ViewScrumRevision

The function of this method is clearly explained in the Javadoc: *Use for view Scrum Revision.*

In particular it takes from the context (in which there are the input parameter) the revision and repository names as Strings:

```
String revision = (String) context.get("revision");
String repository = (String) context.get("repository");
```

Then it creates a process with the command to call the “log” relative to the revision:

```
String logCommand = "svn log -r" + revision + " " + repository;
Process logProcess = Runtime.getRuntime().exec(logCommand);
```

And stamp the result in a buffered reader:

```
BufferedReader logIn = new BufferedReader(new
InputStreamReader(logProcess.getInputStream()));
```

With a while cycle save all the result of the command in a string:

```
while ((logline = logIn.readLine()) != null) {
            logMessage.append(logline).append("\n");
```

The same commands are executed for the extraction of the “diff” relative to the revision. Then the two resulted strings are putted in the result with the revision and the repository reference:

```
result.put("revision", revision);
result.put("repository", repository);
result.put("logMessage", logMessage.toString());
result.put("diffMessage", diffMessage.toString());
```
```

2.1.2 RetrieveMissingRevision

The function of this method is clearly explained in the javadoc: *use for retrieve the missing data of the Revision.*
In particular it takes from the context (in which there are the input parameters) the number of revision:

```
String latestRevision = (String) context.get("latestRevision");
Integer revision = Integer.parseInt(latestRevision.trim());
```

Then it obtains all the information about the user and the task and so it performs the following queries retrieving the missing data of the revision:

```
List <GenericValue> workeffContentList =
EntityQuery.use(delegator).
from("WorkEffortAndContentDataResource").where("contentName",versio
n.trim() ,"drObjectInfo", revisionLink.trim()).queryList();

List<GenericValue> workEffortList = EntityQuery.use(delegator)
.from("WorkEffort").where(exprsAnd).queryList();
```

At the end it calls the runSync method of the dispatcher in order to update the scrum revisions passing as a parameter a Map in which there are all the information necessary.

```
dispatcher.runSync("updateScrumRevision", inputMap);
```

2.1.3 RemoveDuplicateScrumRevision

The function of this document is immediately understandable by its name and by the javadoc: *use for remove duplicate scrum revision.*

In particular with the following queries it retrieve the revision:

```
List<GenericValue> workEffortDataResourceList =
EntityQuery.use(delegator).from("WorkEffortAndContentDataResource")
.where(exprsAnd).queryList();
```

Then the results are distributed over two data structures:

- keys in which there are univocal values;
- exclusion in which there are the duplicates.

```
for (GenericValue workEffort : workEffortDataResourceList)
   {
      String drObjectInfo =workEffort.getString ("drObjectInfo");
         if (keys.contains(drObjectInfo)) {
            exclusions.add(workEffort);
         } else {
                     keys.add(drObjectInfo);
                  }
   }
```

At the end all the elements in exclusion are removed.

2.1.4 LinkToProduct

For this method there is not the Javadoc that explains its function. Reading the code we can see that the method manages the communication events relative to a product. The product is contained in the context as the communication event.

Firstly it extract from the context the communication event ID, and so the communication event itself with a query:

```
String communicationEventId = (String)
context.get("communicationEventId");


GenericValue communicationEvent =
EntityQuery.use(delegator).from("CommunicationEvent").where(
"communicationEventId", communicationEventId).queryOne();
```

Then it extracts the subject of the communication event, and from this the ID of the product at issue. The id is localized and extracted like a substring starting from after the “PD#” characters until the last digit:

```
String subject = communicationEvent.getString("subject");

int pdLocation = subject.indexOf("PD#");
 if (pdLocation > 0) {
     int nonDigitLocation = pdLocation + 3;
    while (nonDigitLocation < subject.length() &&
    Character.isDigit(subject.charAt(nonDigitLocation))) {
          nonDigitLocation++;
     }
    String productId = subject.substring(pdLocation + 3,
    nonDigitLocation);

}
```

Then, after it has extracted the product, it creates the CommunicationEventProduct mapping with the delegator:

```
GenericValue communicationEventProduct = delegator.makeValue(
"CommunicationEventProduct", UtilMisc.toMap("productId",
productId, "communicationEventId", communicationEventId));


communicationEventProduct.create();
```

Then it extracts the Owner of the product and the User Login:

```
GenericValue productRoleMap =
EntityQuery.use(delegator).from("ProductRole").where("productId",
productId, "partyId",
communicationEvent.getString("partyIdFrom"),
"roleTypeId","PRODUCT_OWNER").queryFirst();


 GenericValue userLogin = (GenericValue)
context.get("userLogin");
```

And it uses these to commit the communication:

```
 dispatcher.runSync("setCommunicationEventStatus",
 UtilMisc.<String, Object>toMap("communicationEventId",
 communicationEvent.getString("communicationEventId"),
 "statusId", "COM_COMPLETE", "userLogin", userLogin));
```

So the communication events are registered.

2.2 VALIDATE METHOD CONDITION

This class is part of the minilang package. It is a subclass of MiniLangElements which, as said in Javadoc documentation is a *superclass for all XML element models*. In particular, it is a subclass of MethodOperation, *an abstract class for Mini-language element models*, which extends in turn MiniLangElement class. Moreover, it implements Conditional interface, *an interface for all conditional element*.

```
public final class ValidateMethodCondition extends MethodOperation
implements Conditional
```

The constructor of the class takes fields (thanks to FlexibleMapAccessor), method name, class name and a list of sub operations, divided in subOps and elseSubOps, from the xml element, and set them as private final attributes.

The class has 2 main methods:

- CheckCondition (override from Conditional interface)

```
public boolean checkCondition(MethodContext methodContext) throws
MiniLangException
```

Although Javadoc is not present neither in the interface nor in the class itself, this method seems to invoke the static method that this class aims to check, with fields taken as parameters, and returns a Boolean value which represents the checking of the method condition.

```
Exec (override from MiniLangElements superclass)
public boolean exec(MethodContext methodContext) throws
MiniLangException
```

As specified in Javadoc of the superclass this method generally "*Executes the operation. Returns <code>true</code> if script execution should continue, or <code>false</code> if script execution should stop*". In this particular case, if the condition is checked, it executes all subOps in the method context (*A container for the Mini-language script engine state*), else it runs all elseSubOps.

Then there is another method that "*Updates <code>aic</code> with this element's artifact information*" (in particular with suboperation's artifact information), and two methods for the printing most relevant information about the object, which are helpful for a complete understanding of the structure of the class itself.

Finally, another smaller class is defined in the same file.

```
public static final class ValidateMethodConditionFactory extends
ConditionalFactory<ValidateMethodCondition> implements
Factory<ValidateMethodCondition>
```

It is a static final factory class with the role of generating new ValidateMethodCondition objects.

3 LIST OF ISSUES

3.1 Class ScrumServices

[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.

- Lines 53, 121, 166, 259
  The parameter “ctx” can be named “dispatchContext” in order to specify where the service is operating.
- Lines 53, 121, 166, 259
  The parameter “context” can be named “inputParameter” that underlines better its functionalities.

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

There are two class attributes which are constants and are all in lower case, at lines 51 and 52:

```
public static final String module = ScrumServices.class.getName();
public static final String resource = "scrumUiLabels";
```

They need to be changed in this way:

```
public static final String MODULE = ScrumServices.class.getName();
public static final String RESOURCE = "scrumUiLabels";
```

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

Blank lines should be inserted at line 18 to separate the beginning comment from the package statements, at line 46 to separate import statements from the class declaration and at line 52 to separate attribute declarations from the methods. With these changes the section will be distinguished more clearly.

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

The Javadoc placed before the class declarations contains only the name of the class. It must be expanded for example inserting the services offered by the static method of the class.

3.1.1 ViewScrumRevision

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

It is used the “Kernighan and Ritchie" style.

[12] Blank lines and optional comments are used to separate sections.

No blank lines nor optional comments are used. They can be used in the lines 129, 138 and 145.

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

Line length exceed 80 character in lines 130, 134, 139 and 141. Line 130 cannot be changed, instead line 134, 139 and 141 can be divided in 2 or more expressions.

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

The comment that explain the class is not exhaustive. It is general with the specification of the result. There are no comments for the blocks of code.

[33] Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces ‘{‘ and ‘}’). The exception is a variable can be declared in a for loop.

The lines from 138 to 144 form a conceptual block. They are not surrounded by braces but it is the same a block, and has the declarations at the beginning.

[other problems]

“dispatcher” and “delegator” at line 122 and 123 are declared, initialized but never used.

3.1.2 RetrieveMissingRevision

[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.

At line 175 the integer variable “revision” should be “numberOfRevisions” to underline better its function.

```
Integer revision = Integer.parseInt(latestRevision.trim());
```

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

Many lines exceed the length of 80: 174, 181, 205, 209, 214, 215, 216, 217, 211 and 220. In particular the lines 205, 209 and 211 should be divided in 2 lines to make the code more readable.

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

Some lines exceed the length of 120: 205 and 209.

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

Comments are absolutely not exhaustive. In particular they can be added to lines 209 and 220 in order to explain clearly the meaning of the queries. Than a comment is useful also in line 195 in order to explain how taskId is extracted from taskInfoList.

[33] Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces ‘{‘ and ‘}’). The exception is a variable can be declared in a for loop.

At line “revisionLink” is not declared at the beginning of the block to which it belongs.

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

In line 190 there is no check that the second element of the array “versinInfoTemp” effectively exists.

```
String[] versionInfoTemp = userInfo.split(",");
String user = versionInfoTemp[1];
```

[other problems]

The variable “result”, defined at line 172, is used only in the return statement. So instead of using the following code

```
Map<String, Object> result = ServiceUtil.returnSuccess();
return result;
```

it can be substituted by the single line

```
return ServiceUtil.returnSuccess();
```

3.1.3 RemoveDuplicateScrumRevision

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

Many lines exceed the length of 80: 266, 267, 268, 269, 270, 272, 284, 287, 288 and 289. In particular the lines 266, 270, 287 and 288 should be divided in 2 lines to make the code more readable.

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

Some lines exceed the length of 120: 266, 270, 287 and 288.

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

Comments are absolutely not exhaustive. In particular they can be added to lines 270 and 288 in order to explain clearly the meaning of the queries. Then a comment

can be necessary also at line for explain how the results of the query are distributed between the two sets “keys” and “exclusions”.

[other problems]

“dispatcher” at line 261 is declared, initialized but never used.

The variable “result”, defined at line 263, is used only in the return statement. So instead of using the following code

```
Map<String, Object> result = ServiceUtil.returnSuccess();
return result;
```

it can be substituted by the single line

```
return ServiceUtil.returnSuccess();
```

3.1.4 LinkToProduct

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

It is used the “Kernighan and Ritchie" style.

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

Many lines exceed the length of 80: 62, 70, 75, 77, 79, 83, 87, 91, 94, 101 and 107. Many of these lines can be divided in two or more parts.

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

Many lines exceed the length of 120: 62 with 151 characters, 77 with 192 characters, 79 with 176 characters, 83 with 205 characters, 87 with 212 characters, 91 with 136 and 101 with 121 characters.

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

The method function is not explained at all. The comments at the block are not exhaustive to understand the function of the block.

[33] Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a for loop.

Many declarations appear in the code, not only at the beginning of the blocks, as at line 73.

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

Often use the class "Object" and the class "GenericValue" for save some result, without an explicit casting.

[other problems]

The method has a high cyclomatic complexity. It can be avoided creating a function with the inner instructions, and invoking it.

There are some debugging instructions commented (lines 58 and 74). It would be better to be removed.

```
// Debug.logInfo("==== Processing Commevent: " +

communicationEventId, module);

// Debug.logInfo("======================Product id found in subject: >>"
+ custRequestId + "<<", module);
```

3.2 ValidateMethodCondition

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

There are two class attributes which are constants and are all in lower case, at lines 44 and 45:

```
public static final String module =
ValidateMethodCondition.class.getName();

private static final Class<?>[] paramTypes = new Class<?>[] {String.class
};
```

They need to be changed in this way:

```
public static final String MODULE =
ValidateMethodCondition.class.getName();

private static final Class<?>[] PARAMTYPES = new Class<?>[] {String.class
};
```

[12] Blank lines and optional comments are used to separate sections.

No blank lines between end of beginning comments (line 18) and package import (line 19)

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

Many lines exceed the length of 120: 64 with 132 characters, 86 with 139 characters, 161 with 159 characters, 163 with 126 characters, 168 with 132 characters.

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

Code is not well commented, difficult to understand what methods do.

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

The Java source file ValidateMethodCondition.java contains two public classes: ValidateMethodCondition and ValidateMethodConditionFactory.

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

Javadoc is only on classes, not on methods.

[33] Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a for loop.

Many declarations appear in the code, not only at the beginning of the blocks, as at line 65, 71 and 91.

[44] Check that the implementation avoids "brutish programming".

```
messageBuffer.append("validate-method[");
messageBuffer.append(className);
messageBuffer.append(".");
messageBuffer.append(methodName);
messageBuffer.append("(");
messageBuffer.append(this.fieldFma);
```

Line from 131 to 136: append invocations can be grouped with '+' operator between string, from the moment that there aren't iteration or condition between these instructions, for example in this way:

```
messageBuffer.append("validate-method[" + className + "." + methodName +
"(" + this.fieldFma);
```

From line 93 to line 96, besides, there are multiple useless variable declaration.

```
Class<?> valClass = methodContext.getLoader().loadClass(className);
Method valMethod = valClass.getMethod(methodName, paramTypes);
Boolean resultBool = (Boolean) valMethod.invoke(null, params);
return resultBool.booleanValue();
```

They can be changed in this way:

```
return ((Boolean) (
(methodContext.getLoader().loadClass(className)).getMethod(methodName,
paramTypes)).invoke(null, params)).booleanValue();
```

4 HOURS OF WORK

The writing of this document took overall about 14 hours of head-work.

In particular we used 4 ours in order to divide the work equally and to discuss about the most critical issues. Then we spent 10 hours individually in order to complete our tasks.

At the end we reread the complete document in the last 2 hours.

3 USED TOOLS

The tools we used to create this document are:

- Microsoft Word 2016: to write and assemble the document
- Dropbox e GitHub: to share work
- Eclipse: to read the code of the classes