Code Inspection Document

Marini Alberto
862838
alberto2.marini@mail.polimi.it

Marrone Matteo
810840
matteo.marrone@mail.polimi.it

Sabatelli Antonella
875666
antonella.sabatelli@mail.polimi.it

February 5th, 2017

Politecnico di Milano

Contents

Chapter 1

1 Classes assigned

The classes assigned to our group are:

- SetOperation.java
- CallSimpleMethod.java

The main path is: `/apache-ofbiz-16.11.01/framework/minilang/src/main/java/org/apache/ofbiz/minilang/method/`

SetOperation.java is included inside the *envops* folder. **CallSimpleMethod.java** is included inside the *callops* folder.

Chapter 2

2 Functional role

Other issues such as the improper handling of obsolete code, not explicitly mentioned in the dedicated section, are pointed out within the following description of the functional role of the code.

CallSimpleMethod is a class that when executed interacts with, as the name suggests, a Simple Method object . From the Mini-language reference provided by the developers at the beginning of the code we were able to infer that the object Simple Method is actually a java representation of an element of a xml resource whose content should be a block of code, more precisely, an executable method indeed. The constructor (lines 60-95) takes an element object and a simpleMethod object as parameters and after an inherited initialization whose details are not known it proceeds with the validation of the xml elements the element and simplemethod objects presumably refer to in accordance with the MiniLang standard.

```
    public CallSimpleMethod(Element element, SimpleMethod simpleMethod) throws MiniLangException {
        super(element, simpleMethod);
        if (MiniLangValidate.validationOn()) {
            MiniLangValidate.attributeNames(simpleMethod, element, "method-name", "xml-resource", "scope");
            MiniLangValidate.requiredAttributes(simpleMethod, element, "method-name");
            MiniLangValidate.constantAttributes(simpleMethod, element, "method-name", "xml-resource", "scope");
            MiniLangValidate.childElements(simpleMethod, element, "result-to-field");
        }
        this.methodName = element.getAttribute("method-name");
        String xmlResourceAttribute = element.getAttribute("xml-resource");
        if (xmlResourceAttribute.isEmpty()) {
            xmlResourceAttribute = simpleMethod.getFromLocation();
        }
        this.xmlResource = xmlResourceAttribute;
        URL xmlURL = null;
        try {
            xmlURL = FlexibleLocation.resolveLocation(this.xmlResource);
        } catch (MalformedURLException e) {
            MiniLangValidate.handleError("Could not find SimpleMethod XML document in resource: " + this.xmlResource + "; error was: " + e.toString(), simpleMethod, element);
        }
        this.xmlURL = xmlURL;
```

```java
        this.scope = element.getAttribute("scope");
        List<? extends Element> resultToFieldElements = UtilXml.childElementList(element, "result-to-field");
        if (UtilValidate.isNotEmpty(resultToFieldElements)) {
            if (!"function".equals(this.scope)) { //?
                MiniLangValidate.handleError("Inline scope cannot include <result-to-field> elements.", simpleMethod, element);
            }
            List<ResultToField> resultToFieldList = new ArrayList<ResultToField>(resultToFieldElements.size());
            for (Element resultToFieldElement : resultToFieldElements) {
                resultToFieldList.add(new ResultToField(resultToFieldElement, simpleMethod));
            }
            this.resultToFieldList = resultToFieldList;
        } else {
            this.resultToFieldList = null;
        }
    }
```

As the names imply, it is checked whether the objects passed to the constructor have the required attributes and subelements, and if the attributes have the correct names and are constant when required. Attributes of the CallSimpleMethod object such as xmlresource, xmlurl, xmlscope and resultToFieldList are subsequently initialized getting the needed values from the element parameter. The last one is initialized creating a new ResultToField object, whose purpose according to the documentation is to copy the called service's OUT attribute to the specified field, from any result to field element contained in the element passed.

The creation of a ResultToField object, as observable from the constructor below, simply involves the xml validation of the the couple element, simple method passed as parameters according to miniLang rules and the storage of two attributes result and field in two object of the custom type FlexibleMapAccessor. As for the exec, it begins initializing the simple method to be called from the method name attribute (initialized in the constructor), launching a runtime exception in case methodname was not initialized or no SimpleMethod is associable to such method name.

The localContext the SimpleMethod is going to be run in is then defined from the MethodContext object passed as the parameter as its equal save for two values of the hashmap envMap (presumably standing for environment map and defining almost on its own the method context), the event response code name and the service response message name, provided the scope if the method is function-wide.

```
        String returnVal = simpleMethodToCall.exec(localContext);
```

The simpleMethod is actually finally called at line 114 and asked to run in the local context defined as previously said; its return value is stored in the string returnVal.

```
        if (simpleMethodToCall.getDefaultErrorCode().equals(returnVal)) {
            if (methodContext.getMethodType() == MethodContext.EVENT) {
                methodContext.putEnv(simpleMethod.getEventResponseCodeName(), simpleMethod.getDefaultErrorCode());
            } else if (methodContext.getMethodType() == MethodContext.SERVICE) {
                methodContext.putEnv(simpleMethod.getServiceResponseMessageName(), simpleMethod.getDefaultErrorCode());
            }
            return false;}
        if (methodContext.getMethodType() == MethodContext.EVENT) {
```

In lines 117-124, if such string contains what is seen as an error return value, the methodContext environment map passed as a parameter is loaded with a new event response code name or a new service response message name equal to the default error message for the simple method called according to the methodtype chosen(remember that previous values were erased).

```
        if (methodContext.getMethodType() == MethodContext.EVENT) {
            // FIXME: This doesn't make sense. We are comparing the called method's response code with this method's
            // response code. Since response codes are configurable per method, this code will fail.
            String responseCode = (String) localContext.getEnv(this.simpleMethod.getEventResponseCodeName());
            if (this.simpleMethod.getDefaultErrorCode().equals(responseCode)) {
                Debug.logWarning("Got error [" + responseCode + "] calling inline simple-method named [" + this.methodName + "] in resource [" + this.xmlResource + "], message is " + methodContext.getEnv(this.simpleMethod.getEventErrorMessageName()), module);
                return false;
            }
        } else if (methodContext.getMethodType() == MethodContext.SERVICE) {
            // FIXME: This doesn't make sense. We are comparing the called method's response message with this method's
            // response message. Since response messages are configurable per method, this code will fail.
            String responseMessage = (String) localContext.getEnv(this.simpleMethod.getServiceResponseMessageName());
            if (this.simpleMethod.getDefaultErrorCode().equals(responseMessage)) {
                Debug.logWarning("Got error [" + responseMessage + "] calling inline simple-method named [" + this.methodName + "] in resource [" + this.xmlResource + "], message is " + methodContext.getEnv(this.simpleMethod.getServiceErrorMessageName()) + ", and the error message list is: "
                        + methodContext.getEnv(this.simpleMethod.getServiceErrorMessageListName()), module);
                return false;
            }
```

Lines 124-141 are apparently called to display on the terminal such event code name/service messages (as in the previous instance according to the method type), but if there was really an error which would create the need to display a corresponding appropriate response message then it would be intercepted from the analysis at returnVal at line 117, lines 117-124 would be executed and the exec would end returning false, so that the following ifs wouldn't really have a chance to be executed. Moreover, as the developers themselves state in FIXME comment, the code in those lines does not make sense as it compares *"the called method's response code with this method's response code. Since response codes are configurable per method, this code will fail"*.

Another possible issue would be with the usage of localContext, previously deprived of the event response code name and service response message name values, rather than methodcontext here: we have to assume that those values were replaced as a consequence of the execution of the method as such values would not exist otherwise.

```
        if ("function".equals(this.scope) && this.resultToFieldList != null) {
            Map<String, Object> results = localContext.getResults();
            if (results != null) {
                for (ResultToField resultToField : this.resultToFieldList) {
                    resultToField.exec(methodContext.getEnvMap(), results);
                }
            }
```

Before ending successfully, if the scope of the method called is function wide and the resultToFieldList defined in the class constructor is not null, we proceed with the at least partial execution of lines 145-151, where the values from the map results got from the local context the simple method was run in are made into values of the environment map of the method context passed as a parameter through the execution of the ResultToField private class.

The code that follows contains a factory method for the simple method element, not unlike any factory conceived according to the factory design pattern, the override of a toString obviously added for easily discernable formatting purposes, the already mentioned ResultToField private class and the overriding of a public void method called gatherArtifactInfo. As for the last one, it seems its only purpose is to add the instance od the simpleMethod we are calling to a list in the instance of the object ArtifactInfoContext passed as a parameter, as long as it is not already there.

SetOperation is a class whose execution leads to the addition to the environment map of the method context passed as a parameter at the moment of the execution of a newValue derived from the methodContext object itself, with a field derived from a field element in the element passed as parameter to the constructor as key.

More precisely, the class consists of five code blocks: the boolean method autocorrect, the constructor, the exec, a factory method for setOperation and an override of toString defined for formatting purposes. The last two won't be described in detail.

As it was the case for CallSimpleMethod, the setOperation constructor receives an element and simpleMethod parameters and proceeds with an inherited initialization and the validation of the xml elements the element and simplemethod objects presumably refer to in accordance with the MiniLang standard; this time though the validation consists of more passages, such as the one verifying the potential presence of deprecated attributes. The autocorrect method call follows: the aim of the method (according to the developers needed only during the transition from version 1 to version 2) is to substitute the three deprecated attributes possibly contained in the element , that is, from-field, default-value and value, with three new attributes from, default, from containing the same value. Key fields of the object, such as valueFse, formatFse and most importantly fieldFMa and targetClass are subsequently initialized from the corresponding attributes contained in the element parameter.

```
        boolean isConstant = false;
        Object newValue = null;
        if (this.scriptlet != null) {
            try {
                newValue = this.scriptlet.executeScript(
```

```java
methodContext.getEnvMap());
            } catch (Exception exc) {
                Debug.logWarning(exc, "Error evaluating scriptlet [" + this.scriptlet + "]: " + exc, module);
            }
        } else if (!this.fromFma.isEmpty()) {
            newValue = this.fromFma.get(methodContext.getEnvMap());
                Debug.logVerbose("In screen getting value for field from [" + this.fromFma.toString() + "]: " + newValue, module);
        } else if (!this.valueFse.isEmpty()) {
            newValue = this.valueFse.expand(methodContext.getEnvMap());
            isConstant = true;
        }

        if (ObjectType.isEmpty(newValue) && !this.defaultFse.isEmpty()) {
            newValue = this.defaultFse.expand(methodContext.getEnvMap());
            isConstant = true;
        }
        if (!setIfNull && newValue == null && !"NewMap".equals(this.type) && !"NewList".equals(this.type)) {
            if (Debug.verboseOn())
                Debug.logVerbose("Field value not found (null) with name [" + fromFma + "] and value [" + valueFse + "], and there was not default value, not setting field", module);
            return true;
        }
        if (!setIfEmpty && ObjectType.isEmpty(newValue)) {
            if (Debug.verboseOn())
                Debug.logVerbose("Field value not found (empty) with name [" + fromFma + "] and value [" + valueFse + "], and there was not default value, not setting field", module);
            return true;
        }
        if (this.type.length() > 0) {
```

As for the execution it can be divided in three parts: in the first one, spanning from line 137 to line 167, we decide upon the value to assign to newValue; in the second one we manipulate newValue in different ways according to the value of the object field methodtype; lastly the third part, lines 197-200, hosts the assignment described at the beginning.

```java
            Debug.logVerbose("Setting field [" + this.fieldFma.toString() + "] to value: " + newValue, module);
        this.fieldFma.put(methodContext.getEnvMap(), newValue);
        return true;
    }
```

As for the first part, the newValue can be obtained by executing a scriptlet on the method context parameter, or by extracting it from the envMap of the method context using either the from flexible map accessor or the value flexible string expander derived from the respective matching attributes contained in the element object passed to the constructor. If none of these ways succeeds, the newValue is initialized with the default value; and if after all this (for example due to the unavailability of a default value) newValue stays empty or null, the user is informed via terminal of this and the execution returns true.

In the second part it is first checked through the type field whether the newValue is meant to be a map or a list and, if that's the case, the value is initialized as either a hashmap or a LinkedList. If the type is defined but different than a list or a map newValue is a converted into an object of the type specified by the target class, a field that had been initialized in the constructor from the field type itself, potentially in the format given , if one was derived in the constructor.

Chapter 3

3 Issues

3.1 Naming Conventions

1. In SetOperation:

   - On line 46 the constant doesnt have a meaningful name. Instead of using the name module for the variable, a more meaningful name should be used.
   - On lines 78/81 even if the variable names (defaulFse, formatFse, fieldFma, fromFma) dont indicate their use, the meanings are clear when reading the variable types.
   - On line 87 even if the variable name (valueFse) does not indicate its use, the meaning is clear when reading the variable type.
   - On line 205 even if the variable name (sb) does not indicate its use, the meaning is clear when reading the variable type.

   In CallSimpleMethod:

   - On line 50 the name of the constant (module) is not meaningful.
   - On line 178 even if the variable name (sb) does not indicate its use, the meaning is clear when reading the variable type.
   - On lines 209 and 210 even if the variable names (fieldFma, resultNameFma) dont indicate their use, the meanings are clear when reading the variable types.

2. No use of one character variable.

3. In setOperation:

   - On line 44 the name of the class (SetOperation) doesnt follow the naming convention. The name of the class should be a noun not a verb.
   - On line 237 the name of the class (SetOperationFactory) doesn't follow the naming convention. It should be a noun not a verb.

   In CallSimpleMethod:

   - On line 48 the name of the class (CallSimpleMethod) doesn't follow the naming convention. It should be a noun not a verb.
   - On line 195 The name of the class (CallSimpleMethodFactory) doesn't follow the naming convention. It should be a noun not a verb.

4. No interface is present in the code.

5. In SetOperation:

   - On line 49 the name of the method (autoCorrect) doesnt follow the naming convention. It should be a verb.
   - On line 89 the name of the constructor (SetOperation) doesn't follow the naming convention. The first letter should be in lowercase.

   In CallSimpleMethod:

   - On line 78 the name of the method (CallSimpleMethod) doesn't follow the naming convention. The first letter should be in lowercase.
   - On line 212 the name of the method (ResultToField) doesn't follow the naming convention. It should be a verb and the first letter should be in lowercase.

6. All class variables are declared correctly.

7. In setOperation, on line 46 the name of the constant (module) is not in uppercase.
   In CallSimpleMethod, on line 50 the name of the constant (module) is not in uppercase.

3.2 Indention

8. Spaces are used correctly to indent.

9. No tabs are used to indent.

3.3 Braces

10. Bracing style is used adequately.

11. In SetOperation, on lines 160, 165 and 198 the if statement is not surrounded by curly braces.
    In CallSimpliMethod, on line 114 the if statement is not surrounded by curly braces.

3.4 File Organization

12. In SetOperation, on lines 101, 103, 105, 108, 114, 120, 126, 129, 136, 139, 204 and 206 a blank line is missing to separate variable declaration from the if condition.

- On lines, 190 and 191 a blank line is missing to separate variable declaration from the method invocation.
- On lines 90, 91, 198 and 200 A blank line is missing to separate the method invocation from the if condition.

In CallSimpleMethod, on lines 65/68, 77/81, 99, 101, 103, 105, 111, 113, 126, 177, 179, 219/221, 228 and 230 to separate the variable declaration from the if condition.

- On lines 58 and 60 a blank line is missing to separate the method invocation from the if condition.
- On lines 96 and 97 a blank line is missing to separate the class declaration from the if condition.
- On lines 70, 73, 153 and 155 a blank line is missing to separate the variable declaration from the try section.
- On lines 126 and 134 a blank lines is missing to separate the beginning of the comments.

13. In SetOperation, lines 92/99, 106, 109, 115/117, 120, 124, 129, 143, 158, 177, 181, 187 and 198 exceed 80 characters.
    In CallSimpleMethod, lines 61/64, 80, 85, 87, 98, 100, 110, 117, 119, 126, 134, 137, 156, 199, 212, 215/217, 219, 223, 228 and 232 exceed 80 characters.
14. In SetOperation, lines 94, 131, 148, 160, 165, 185, 187, 190 and 239 exceed 120 characters.
    In CallSimpleMethod, lines 76, 83, 102, 114, 128, 136, 164, and 197 exceed 120 characters. The exceeding of the 120 characters could be avoided using line breaks according the convention.

3.5 Wrapping Lines

15. In CallSimpleMethod, on lines 137 and 138 the line break doesnt occur after an operator but before.
16. No presence of high level breaks.
17. All the statement are correctly aligned with the beginning of the expression at the same level as the previous line.

3.6 Comments

18. No comment is present to adequately explain what the assigned code is doing. Should be useful to add some comments to help the reader to understand the purpose of the code.
19. No commented out code found.

3.7 Java Source Files

20. In each Java source file there is a single public class. At the same time, inside the CallSimpleMethod.java and SetOperation.java, there are two nested public static final classes named: CallSimpleMethodFactory and SetOperationFactory.

21. The two public classes CallSimpleMethod and SetOperation are the first one in the file.

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

23. The javadoc is complete and it covers all classes inside the code.

3.8 Package and Import Statements

A package statement is present in both segments of code, and in each case all the import statements are correctly immediately placed after it.

3.9 Class and Interface Declaration

25. Classes are declared in the right order.

26. Methods are grouped by functionality.

27. Code is free of duplicates, long methods, big classes and breaking encapsulation.

3.10 Initialization and Declaration

28. All the variables and class members are of correct type and they have right visibility.

29. Variables are declared in the proper scope.

30. Constructors are called when a new object is desired.

31. Object references are initialized before use.

32. Variables are initialized when they are declared.

33. Declarations appear at the beginning of the code.

3.11 Method Calls

34. Methods were checked and parameters were found to be listed in the correct order.

35. Used methods are mostly custom, and from the consultation of their library it seems they are adequate to their purpose.

36. Return values of not void methods are always used for assignments or condition checking in a proper manner.

3.12 Arrays

37. There are no arrays.

3.13 Object Comparison

40. In SetOperation, on lines 139, 158, 180, 212 the comparison with null is carried out using != or == instead of equals. The same goes for line 103, 118, 120, 125, 133, 144, 160, 224, 235 in CallSimpleMethod, where the comparison takes place not only between an object and null but also between an object and an enum value and between an object and an integer(224).

3.14 Output Format

41. All the output is free of spelling and grammatical errors.

42. Error messages are comprehensive and explain how to correct the problem.

43. The output is formatted correctly. Line stepping and spacing are right.

3.15 Computation, Comparisons and Assignments

44. No instances of redundant code or other example of brutish programming were found.

45. There seems to be no ambiguity with relation to execution order that calls for the need of parenthesis insertion.

47. No Mathematical operations are carried out in the code.

49. Comparisons and boolean operators seem to be correct indeed.

50. Three exceptions are called explicitly after verifying error conditions: an IllegalArgumentException and two MinilangRuntimeException. In all cases the condition is legitimate and evaluable.

51. The only conversion visible is located in setOperation and it is pretty explicit, happening only after a check of the existence of a target class in the document and utilizing a MiniLang custom method.

3.16 Exception

52. The relevant exception are caught.

53. The try-catch blocks in the CallSimpleMethod.java source code have been handled correctly. In the SetOperation.java file, at the line 142 e 189, gli errori vengono semplicemente segnalati come Exception.

3.17 Flow of Control

54. There are no switches nor loops.

3.18 Files

57. No files are opened throughout the code.

Chapter 4

4 Hours of work

- Alberto Marini: 12 hours
- Matteo Marrone: 12 hours
- Antonella Sabatelli: 12 hours