
POLITECNICO DI MILANO
2016-2017

# SOFTWARE ENGINEERING 2

# CODE INSPECTION
VERSION 1.0

PEVERELLI FRANCESCO
REPPUCCI FEDERICO

RELEASED ON: FEBRUARY 4, 2017

**TABLE OF CONTENT**

## 1. ASSIGNED CLASSES

*Class path:* apache-ofbiz-16.11.01/framework/minilang/src/main/java/org/apache/ofbiz/minilang/method/envops/SetCalendar.java

*Class name:* SetCalendar

## 2. FUNCTIONAL ROLE OF ASSIGNED SET OF CLASSES

### 2.1. INTRODUCTION

The SetCalendar class is part of the minilang section of the ofbiz framework. Minilang is a language defined as an XML schema, used specifically to code events and services to use within the framework. It aims to ease the development process by allowing whoever is writing the code to operate at a higher level, without needing to worry about declaring variables and managing data structures. The SetCalendar class defines an operation which allows to create a new timestamp by adding the desired amount of time to another timestamp ("from" attribute) or to a specified constant ("value" attribute), and storing it in a specified destination ("field" attribute). For more detail on the syntax see:

https://cwiki.apache.org/confluence/display/OFBADMIN/Mini+Language+-+minilang+-+simple-method+-+Reference

### 2.2 CONSTRUCTOR

The class constructor starts with the validation of the syntax of the element, if the validation option is on, and notifies if any problem arises. Then, the autoCorrect method of the class is called, to modify any deprecated syntax in the name of the attributes and correct if any expression belonging to the "from" attribute has been placed in the "value" attribute. In the following section it is established whether the "from" attribute will be specified as an object returned by a script or as an object present in a map structure, and auxiliary objects are instanced accordingly. If no "from" value is specified, the value of the "value" attribute is retrieved. If both are present, an exception is thrown. Afterwards, the value of the attributes representing the time to add to the timestamp is retrieved, as well as the settings to align the resulting value either to the start or the end of the date's year, month, week or day, the local calendar and the local time zone.

### 2.3 THE EXEC METHOD

The exec method is the method which executes the operation described in the minilang command. It starts by retrieving the value of the "from" attribute, either by executing a script or by retrieving the object from a map structure, and assigning it to a "newValue" attribute. If the "from" attribute is not present, the value of the "value" attribute is assinged newValue. If newValue is still empty but a default value is available, the default value is assigned. If no default value is available and the attribute "setIfNull" is false, the method returns. After that, the variables representing time constants are declared and

initialized to ‘0’, and the local calendar and time zone to null. Moreover, a timestamp variable “fromStamp” is also initialized to null. After the initialization, the values specified in the set calendar element are assigned to all the variables declared above, and the value of fromStamp is used to create a calendar object, add the specified time constants to it, taking into account the local calendar and time zone, and the returned value is assigned to a “toStamp” variable. Finally, the desired period alignments are performed, and the value of toStamp is put in a map structure, and the method returns.

### 2.4 OTHERS

The toString method and the SetCalendarFactory are both well-known java constructs and behave as expected.

## 3. LIST OF ISSUES FOUND BY 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.

- classes: no problems detected.
- interfaces: no problems detected.
- methods: no problems detected.
- class variables: no problems detected.
- method variables: no problems detected.
- constants: no problems detected.

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

No one-character variables are used

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

The name of the class is SetCalendar. It is not composed of nouns in mixed case, but it represents the XML element model "set-calendar". Thus, its name is appropriate.

4) Interface names should be capitalized like classes.

Interface names are correctly capitalized

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

All the methods are verbs with the first letter of each addition word capitalized.

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. Examples: _windowHeight, timeSeriesData.

Variable names are correctly capitalized according to convention

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

The following constant is in lowercase instead of being in uppercase.

line 50: public static final String module = SetCalendar.class.getName();

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

Four spaces are consistently used to indent. At line 111, eight spaces are used instead of four

9) No tabs are used to indent

All indentations are done using tabs.

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 "Kernighan and Ritchie" bracing style is used consistently

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

- if statement: each if statement has curly braces.
- while statement: there are not while statements.
- do-while statement: there are not do-while statements.
- try-catch statement: each try-catch statement has curly braces.
- for statement: there are not for statements.

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 are correctly used to separate sections

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

line 46: 166 characters (it is a comment)

line 110: 99 characters
line 112: 105 characters
line 113: 137 characters
line 113: 133 characters
line 117: 82 characters
line 119: 92 characters
line 130: 87 characters
line 136: 82 characters
line 138: 126 characters
line 141: 86 characters
line 143: 82 characters
line 144: 84 characters
line 146: 82 characters
line 147: 86 characters
line 148: 86 characters
line 149: 84 characters
line 150: 103 characters
line 151: 99 characters
line 152: 84 characters
line 153: 89 characters
line 165: 93 characters
line 192: 118 characters
line 201: 127 characters
line 232: 105 characters
line 254: 98 characters
line 267: 97 characters
line 335: 111 characters

14) When line length must exceed 80 characters, it does NOT exceed 120 characters.
Multiple lines exceed 120 characters:

- lines 110 and 111: MinilangValidate.attributeNames(...
- line 134: throw new IllegalArgumentException(...
- line 183: locale = (Locale) ObjectType.simpleTypeConvert(...
- line 192: timeZone = (TimeZone) ObjectType.simpleTypeConvert(...

- line 200: fromStamp = (Timestamp) MiniLangUtil.convertType(...

15) Line break occurs after a comma or an operator

No problems detected.

16) Higher-level breaks are used.

There are no nested-expressions breaks

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

No problems detected.

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

The comments explain clearly what a section of code does, even though they may not be so understandable without knowledge of the context in which the class operates.

The reference link for the mini-language documentation is outdated, but the page has a link which redirects to the new documentation page.

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 is not commented out code;

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

The java souce files contains more than one public class, since it contains both the SetCalendar class and the SetCalendarFactory class.

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

Yes, it is.

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

The class does not implement any interface. It extends the MethodOperation abstract class, and does so consistently with the class documentation.

23) Check that the Javadoc is complete.

The Javadoc miss for the following:

Constructors

public SetCalendar(Element element, SimpleMethod simpleMethod) throws MiniLangException

Methods

private static boolean autoCorrect(Element element)

private static int parseInt(String intStr)

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

Package and import statements follow conventions

25) The class or interface declarations shall be in the following order:

(a) class/interface documentation comment;

(b) class or interface statement;

(c) class/interface implementation comment, if necessary;

(d) class (static) variables;

i. first public class variables;

ii. next protected class variables;

iii. next package level (no access modifier);

iv. last private class variables.

(e) instance variables;

i. first public instance variables;

ii. next protected instance variables;

iii. next package level (no access modifier);

iv. last private instance variables.

(f) constructors;

(g) methods.

There is the "autoCorrect" method in the attribute declaration section. By the way, the order of attributes declaration is respected.2

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

Methods seem grouped by accessibility in this way:

- private static methods first

- constructors after the class attributes

- public methods right after rather than by functionality. That being said, there are only two public static methods, which are both used to address compatibility issues, and one public method which fulfils the class' main functionality, other than a constructor and a toString method. Therefore, the methods result also grouped by functionality in this case. A more extensive analysis should be conducted on other source files in order to clarify this point.

27) Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate

duplicates: the class is free of duplicated code.

long method: the longest method (exec) has around 100 rows. I think that is not too much long since it is written to perform just one purpose and there are not pieces of code useful in different context.

big classes: the class has 335 rows. It has a medium size.

breaking encapsulation: there are no breaking encapsulation, since the class doesn't implement interfaces.

coupling and cohesion: the cohesion is high, especially in the exec method.

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

Variables and class members are of the correct type and have the right visibility

29) Check that variables are declared in the proper scope

Each variable is declared in the correct scope.

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

When a new object is desired, either constructor or appropriate getInstance methods are called.

31) Check that all object references are initialized before use

• the "element" object, passed to "SetCalendar" constructor, can be null during the execution.

• the "element" object, passed to "autoCorrect" method, can be null during the execution.

• the "intStr" object, passed to "parseInt" method, can be null during the execution.

• the "methodContext" object, passed to "exec" method, can be null during the execution.

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

Variables are initialized where declared, unless they depend on a computation. In this case all the private static variables of the class depend on a computation carried out in the constructor

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.

- class attributes are not at the beginning of the class.
- in the SetCalendar constructor, the attributes are not at the beginning.
- in the following methods, the attributes are not at the beginning:
  - exec
  - autoCorrect

34) Check that parameters are presented in the correct order

Parameters are presented in the correct order.

35) Check that the correct method is being called, or should it be a different method with a similar name

No problems detected.

36) Check that method returned values are used properly.

Methods returned values are used properly

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)

No arrays are used.

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

There is no array/collection access by indexes.

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

No collections or arrays are used.

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

'==' operator is used in the following instances:

- line 168:  if (!setIfNull && newValue == null)
- line 185, 188: if (locale == null)
- line 194, 197: if (timeZone == null)

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

The output text I well written and without errors.

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

At line 158:

```
Debug.logWarning(exc, "Error evaluating scriptlet [" + this.scriptlet + "]: " + exc, module);
```

the message is rather generic, but further information is provided by the methods throwing the exception. The same reasoning applies for line 223:

```
throw new MiniLangRuntimeException("Exception thrown while parsing attributes: " + e.getMessage(), this);
```

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

All output texts are error messages. They are always well formatted and easily readable.

44) Check that the implementation avoids “brutish programming”

No "brute force" solutions found.

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

No operations are computed.

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

There is no liberal use of parenthesis.

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

No divisions are computed.

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

No integer arithmetic is used.

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

No problems detected.

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

All error conditions are legitimate.

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

No problems detected.

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.

For each catch of the class (2 in total), the adopted action seems appropriate.

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

No 'switch' statements used.

55) Check that all switch statements have a default branch.

There is no switch statement.

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

No loops in the code.

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

No files are used in the class.

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

No files are used in the class

59) Check that EOF conditions are detected and handled correctly

No files are used in the class

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

No files are used in the class.

**4. WORKING HOURS**

| | | |
|---|---|---|
| Together: | 2h | [2h] |
| Peverelli: | 5h + 2h + 1h | [8h] |
| Reppucci: | 5h + 2h | [7h] |