Code Inspection Document

POWER Enjoy

Authors:
Cannas, Castiglioni, Loiacono

POLITECNICO
MILANO 1863

1.SUMMARY

1.1 REFERENCES

- Apache OFBiz Framework Introduction (https://cwiki.apache.org/confluence/display/OFBIZ/Framework+Introduction+Videos+and+Diagrams)
- “Apache OFBiz Cookbook”, by Ruth Hoffman
- “Apache OFBiz Development: the Beginner’s tutorial”, by Jonathon Wong, Rupert Howell
- “Apache OFBiz Advanced Framework Training”, by David E. Jones
- “Building Parsers with Java book”, by Steven John Metsker

2.CLASSES ASSIGNED

The task of our group was to analyze the ***ModelWidgetCondition*** class of the open source project ***Apache OfBiz***. This class is located in the ***org.apache.ofbiz.widget.model*** package.

3.FUNCTIONAL ROLE OF THE CLASS ASSIGNED

This part of the document aims to provide an high-level view of the class functionalities, in the context of the Apache OfBiz project. In order to do this, an important premise has to be done: the class ***ModelWidgetCondition*** is never used, nor expanded, and in general never referenced outside the file ***ModelWidgetCondition.java*** itself. This statement is confirmed by the following reasonings:

- even if the class is declared *Abstract* and consequently could not be instantiated, but has to be extended by other non abstract classes, we found out, using a simple search tool -provided by explorer.exe (part of Windows operative system)- , that the class is never extended throughout the whole project (i.e. doesn't exist any reference like ***public class xxx extends ModelWidgetCondition***) .
- It is true the inside the class, many Static Classes are declared, but, according to the java 8 tutorials about Nested Classes (https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html), Static Classes are always accessed using the enclosing class name (i.e. ***OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass()***). Throughout the whole project no reference could be found about these classes.
- Also the Inner interfaces are never implemented by any of the classes of the project (i.e. ***public class xxx implements ModelWidgetCondition.interfacexxx***)
- Finally, a class that is very similar in structure and usage to ***ModelWidgetCondition***, but instead used outside the context of the class itself, is ***AbstractModelCondition***, which presents the same Inner Static Classes, the same methods (with the exception of the ***toString*** method, overridden in the second class), and the same Inner Interfaces.

After the analysis of the structure of the project, and the similarities between the two classes ***ModelWidgetCondition***, and ***AbstractModelCondition***, we can state, with an high degree of certainty, that ***ModelWidgetCondition*** was a class written before ***AbstractModelCondition***, and then abandoned in favor of the second one.
The project managers never annotated it as Deprecated, or deleted it from Apache OfBiz probably due to an oversight.

With this important premise in mind we can analyze the class, pretending that it is the one used inside the project, and that is not considered obsolete.
We will start analyzing its structure, and then we will move on with the analysis of its functionality and behaviour in the context of the Apache OfBiz project.

3.1 CLASS STRUCTURE

ModelWidgetCondition is an abstract class inside which are defined many different Inner Static Classes, most of them extends the outer one, and implements an Interface:

```
public static interface Condition {
   boolean eval(Map<String, Object> context);
}
```

NOTE: the abstract class also implement an eval method:

```
public boolean eval(Map<String, Object> context) {
   return rootCondition.eval(context);
}
```

This circumstance, although possible (every Inner Static Class has a @override annotation for the eval method), is considered a bad practice in Java programming: it would have been much better to just implement the interface when **ModelWidgetCondition** is defined.

Another Inner Static class is the ***DefaultConditionFactory***, which implements the ***ConditionFactory*** (also defined inside ***ModelWidgetCondition***).
Finally, the methods provided by the class (and indeed inherited by the Static Classes), are the following:

- ***protected ModelWigetCondition (ConditionFactory factory, ModelWidget modelWidget, Element conditionElement)***(the constructor method)
- ***public getModelWidget ()***(which return the ModelWidget object stored inside the class)
- **public static List<Condition> readSubConditions(ConditionFactory factory, ModelWidget modelWidget, Element conditionElement).**

3.2 FUNCTIONAL ROLE

With the high level structure of the program analyzed and clear in our mind, we can provide a brief explanation of what functionality is provided.
When one of the extension of ***ModelWidgetCondition*** is instantiated, an ***XML Element*** (see https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Element.html) is passed as one of the arguments, as well as a ConditionFactory (in order to follow the Factory Pattern, and not directly expose the construction logic of extended ***ModelWidgetCondition*** objects).

> NOTE: a ***ModelWidget*** is also passed in as an argument, but has no internal use inside the class. We deduced that it is only used as an ID-like argument, to track down ModelWidgetConditions objects.

Before going into further details about the functional role of our class inside the Apache OFBiz framework, we need to briefly talk about the architecture of it, and about its basic functioning.
The Apache OFBiz framework works primarily with a client-server architecture, following a Model-View-Controller pattern.
A ***Controller*** manages the Client's requests, calling the right service for each incoming request. The implementation of the service exploits the data from the **Model** (a collection of data entities in relationship within each other), and when necessary, creates a **View** to display some information to the User.
All of this work is done by the Controller, which acts as a wire between the Model and the View: through its ***request-map***, it calls the appropriate services, and through its ***view-map***, displays the correct View to the User with the right information taken from the Model.

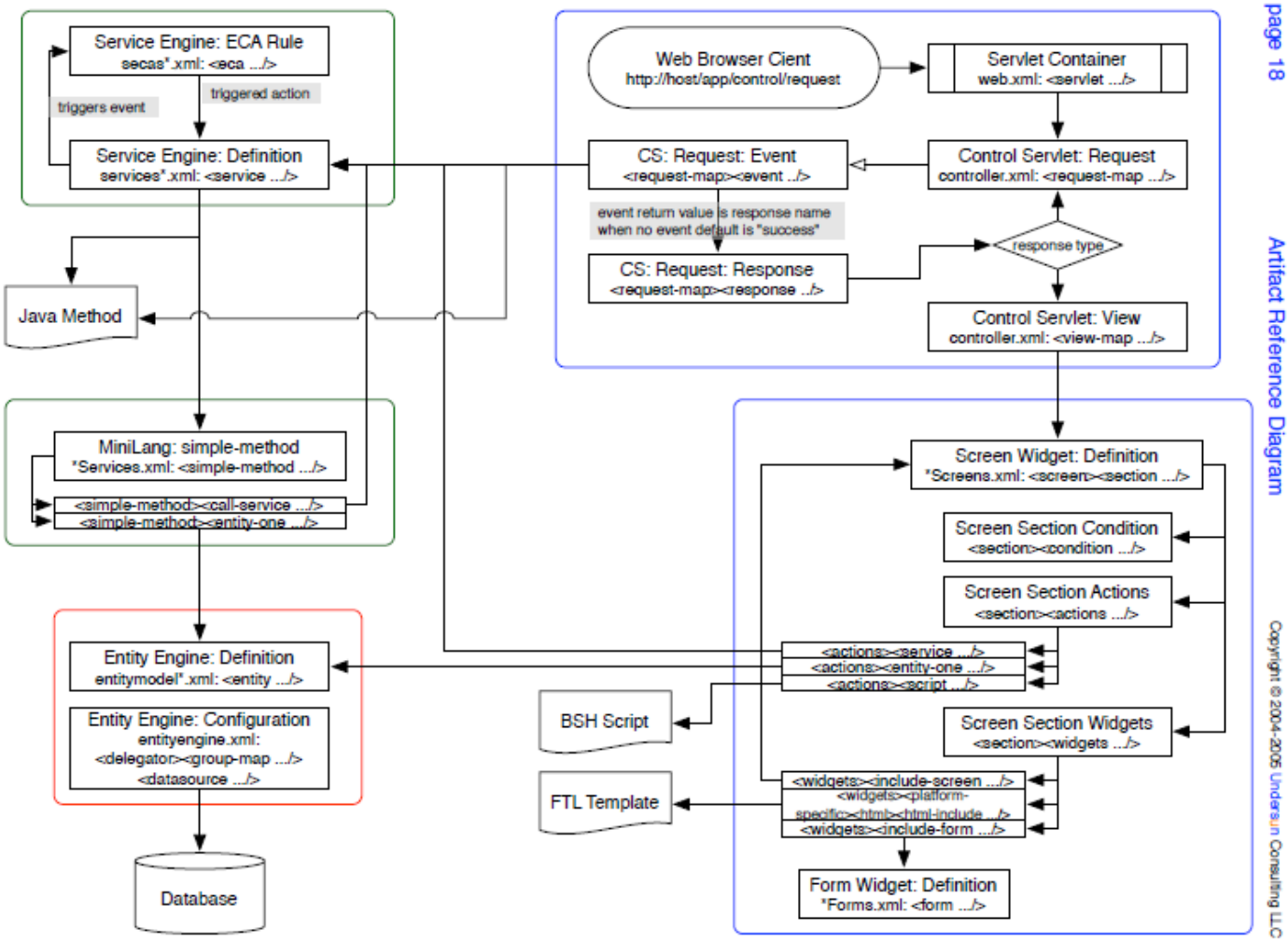

Figura 1: Apache OFBiz artifact diagram

As we can see from the artifact diagram (https://cwiki.apache.org/confluence/download/attachments/7045155/18ArtRefDia.pdf?version=1&modificationDate=1267053493000&api=v2), Controllers are implemented through specific components called ***Servlets***, which are deployed inside the environment created by a ***Container*** (in a similar fashion to J2EE).

Both the ***request*** and ***view-maps*** are declared inside an XML file specifying the requests and the Views that the Controller can handle (***controller.xml***).

XML is widely used throughout the whole framework: **Entities**, for example, representing the actual entities in the Model of our architecture, are defined through XML files.

While this operation is quite common for the portability of data throughout different DBMS in the web-services environment, the OFBiz framework goes further allowing the definition of services too, through means of **MiniLang**.

MiniLanguage is a simple programming language based on the concepts expressed in the book **"Building Parsers with Java book"**, by Steven John Metsker, often referred to as the **"Gang Of Four Interpreter"** pattern.
The main idea is to evaluate the sentences in a language assigning each symbol of this to a class in another object-oriented specialized language.
In the MiniLang context, the specific language is XML, the symbols are the XML tags, and the highly specialized language is Java.
Through means of this tool, the programmer can easily specify services and express instructions using an easier and higher-level syntax (theoretically), without caring of the implementation details of the actual specialized language (in this case, Java, being the whole OFBiz framework being based on a JVM) on which MiniLang is going to be interpreted.
All of this permits the programmer to flexibly define, for example, **XML content** for a View to be displayed, depending on the **actual environment condition** and on the type and characteristic of the Client's request.
That is the reason behind the definition of the ***<condition>*** and ***<actions>*** tags inside the screen and widget XML definition files.

```
<screen name="ConditionalScreen">
  <section>
    <condition>
      <if-compare field-name="parameters.show" operator="equals"
                  value="widgets"/>
    </condition>
    <actions>
      <set field="blah" value="blih"/>
    </actions>
   <widgets>
      <label text="Condition passed. Showing widgets element. Blah
                   is: ${blah}"/>
    </widgets>
    <fail-widgets>
      <label text="Condition failed! Showing fail-widgets element.
                   Blah is: ${blah}"/>
    </fail-widgets>
  </section>
</screen>
```

Figure 2: a View's screen XML definition file

As we can see from the image above, every screen and widget (that is, a particular visual element to be collocated inside a screen) definition in XML, comprehends a certain numbers of tags.
While the ***<action>*** elements are optional and represent some operations needed for preparing the data for the actual screen rendering, the ***<condition>*** element express a series of requirements that have to be encountered to permit the rendering of the ***<widget>*** element in case of success, or, in case of failure, of the ***<fail-widget>*** element.

This operation is done recursively: thus, though a screen may have a certain number of conditions in order to display a widget, the widget itself may contain a certain number of conditions in order to or not display specific elements that compose it.

```
<widgets>
    <decorator-screen name="CommonPartyDecorator" location="${parameters.mainDecoratorLocation}">
        <decorator-section name="body">
            <section>
                <condition>
                    <or>
                        <if-compare field="hasViewPermission" operator="equals" value="true" type="Boolean"/>
                        <if-compare field="hasPcmCreatePermission" operator="equals" value="true" type="Boolean"/>
                        <if-compare field="hasPcmUpdatePermission" operator="equals" value="true" type="Boolean"/>
                        <if-compare-field field="parameters.partyId" operator="equals" to-field="userLogin.partyId"/>
                        <not><if-empty field="mechMap.partyContactMech"/></not>
                    </or>
                </condition>
```

Figure 3: a widget XML definition inside a screen XML definition file

The ***<condition>*** elements can be quite articulated: they can have logical operators (joiners, *and, or, xor, nor*), permission related checks (*<if-service-permission>, <if-has-permission>, <if-entity-permission>*) and comparators of various form (*<if-validate-method>, <if-compare>, <if-compare-field>, <ifregexp>, <if-empty>*).

The ***XML Element*** passed as a parameter to the constructor of ***ModelWidgetCondition***, is an XML object representing the list of conditions for the evaluation of a screen widget: that is, in poor words, ***ModelWidgetCondition*** is the class corresponding to the ***<condition>*** symbol **in the interpreter pattern** inside the MiniLang context.

In the following part of this section, we will analyse how the class' object construction follows this model.

In fact, the ***XML Element*** is read starting from its root node, and for every sub-element, using a cycle in the constructor, a new object extending and implementing ***ModelWigetCondition*** is created, following the logic suggested by the name of the node accessed (the condition tag is representing):

```
public static List<Condition> readSubConditions(ConditionFactory factory, ModelWidget
modelWidget, Element conditionElement) {
    List<? extends Element> subElementList = UtilXml.childElementList(conditionElement);
    List<Condition> condList = new ArrayList<Condition>(subElementList.size());
    for (Element subElement : subElementList) {
      condList.add(factory.newInstance(modelWidget, subElement));
    }
    return Collections.unmodifiableList(condList);
  }

public Condition newInstance(ModelWidget modelWidget, Element conditionElement) {
  if (conditionElement == null) {
    return TRUE;
  }
```

```
if ("and".equals(conditionElement.getNodeName())) {
   return new And(this, modelWidget, conditionElement);
} else if ("xor".equals(conditionElement.getNodeName())) {
   return new Xor(this, modelWidget, conditionElement);
  ...
```

It is possible to notice a clear bipartition in the classes that extends ***ModelWidgetCondition*** and implements ***Condition*** (all of which refers directly to the constructor of their parents, using ***super(...)*** inside their constructor): if the condition they are referred to is logical (i.e. ***And, Or, Xor, Not***), they always try to read sub-conditions from the current node; instead, if the condition they are referred to is not a logical one (i.e. ***If-Compare, IfValidateMethod***…) the are considered as leaves, and because of this, no sub-condition is read from the node they take into account.
In summary, each one of the objects created during the constructor cycle represent a different XML ***<condition>*** tag (logical, permission related, or comparative).
For everyone of them, a different class implementation is provided, with an appropriate method ***eval()*** (defined in the ***Condition*** interface, which every class extending ***ModelWidgetCondition*** implements) returning a boolean value on whether the conditions expressed in the tags are met are or not.
Now, we can elaborate on the context and the behaviour of the class: we will address which methods are called outside ***ModelWidgetCondition***, where they are called, and why. We will not address every single Static Class nested inside ***ModelWidgetCondition***, because they behave in similar way, calling similar methods, and because of this, it won't be useful to understand what the Class does inside its context.
It is clear that the original functionality of the class was to provide the actual implementation for the evaluation methods of a list of condition read from an XML document, inside the MiniLang context of the Gang Of Four Interpreter pattern.
It is a class designated to provide information about when a Widget of a particular View screen has to be rendered.
This becomes evident when we analyze ***AbstractModelCondition***, the class that replaced ***ModelWidgetCondition*** in its functionalities.
Inside every extension of the abstract class, we can find ***get*** methods (which returns the values read from XML nodes and saved inside the instances of the classes), and a ***public boolean eval(Map<String, Object> context)*** overridden method, provided with a context Object to be evaluated.
The context is estimated starting from the variables of the class extending ***AbstractModelCondition***: the comparison is performed in different ways, due to the different implementations of the method ***eval()***, which in turn depends on the different types of the ***<condition>*** XML tag, for which a different static class implementation is provided (again, remaining inside the Gang of Four Interpreter pattern model).
It becomes clear that that the original usage of ***ModelWidgetCondition*** was similar to the one of ***AbstractModelCondition***, although no ***get*** method is implemented inside the class, and thus

accessing the values of the Classes is impossible, and they could only be used as an evaluation base (using the method ***eval()***).

However, often ***AbstractModelCondition*** static classes' implementations are used to evaluate different widget conditions, while those of ***ModelWidgetCondition()*** are not.

In particular, we focused our attention on the ***ModelScreenWidget*** class.

This particular piece of code, which implements the list of instructions for a screen and its all subwidgets to be rendered by a different class (see ***XMLWidgetConditionVisitor*** and ***XMLWidgetVisitor***, in the same package), never refers to the ***<condition>*** tags using the ***ModelWidgetCondition eval()*** methods, but to the ***eval()*** methods of the ***AbstractModelCondition*** static implementations instead.

For all these reasons, we are likely convinced of our previous statements: our inspected class was an earlier implementation for the ***<condition>*** tags of a generic XML widget element, which was after deprecated for a more generic ***AbstractModelCondition***, which could be used recursively in the rendering of a screen (as it is used in the ***ModelScreenWidget*** class).

4.LIST OF ISSUES

4.1 NAMING CONVENTIONS

Errors are only listed when first found within a Class for the first time, i.e. when variables/methods/classes are named for the first time.

Line75:'module' is not a proper name for a constant,it should be uppercase.
Line 87: 'eval' is not a proper name for a method.
Line 118:'eval' is not a proper name for a method.
Line 130:'eval' is not a proper name for a method.
Line 163: 'newInstance' is not a proper name for a method, it should be a verb.
line 211-214:'fieldAcsr' is not a proper name for a variable.
line 215:'valueExdr' is not a proper name for a variable.
line 253: IfCompareField is not a proper name for a class.
line 307: IfEmpty is not a proper name for a class.
line 388: IfRegexp is not a proper name for a class.
Line 434-437: '*Exdr' is not a proper subname for a variable.
Line 440: 'condElement' is not a proper name for a parameter.
Line 448: 'eval' is not a proper name for a method, it should be a verb.
Line 473: 'dctx' is not a proper name for a variable.
Line 475: 'permService' is not a proper name for a variable.
Line 484: 'svcCtx' is not a proper name for a variable.
Line 490: 'resp' is not a proper name for a variable.
Line 517: 'classExdr' is not a proper name for a variable.
Line 518: 'fieldAcsr' is not a proper name for a variable.
Line 519: 'methodAcsr' is not a proper name for a variable.
Line 521: 'condElement' is not a proper name for a variable.
Line 532: 'eval' is not a proper name for a method, it should be a verb.
Line 535: 'fieldVal' is not a proper name for a variable.
Line 548: 'paramTypes' is not a proper name for a variable.
Line 550: 'valClass' is not a proper name for a variable.
Line 557: 'valMethod' is not a proper name for a variable.
Line 564: 'resultBool' is not a proper name for a variable.
Line 583: 'condElement' is not a proper name for a variable.
Line 590: 'eval' is not a proper name for a method, it should be a verb.
Line 603: 'condElement' is not a proper name for a variable.
Line 609: 'eval' is not a proper name for a method, it should be a verb.
Line 628: 'condElement' is not a proper name for a variable.

Line 634: ‘eval’ is not a proper name for a method, it should be a verb.
Line 636: ‘foundOneTrue’ is not a proper name for a variable.

4.2 INDENTION

Line 234: Indention of 8 spaces instead of 4.
Line 285: Indention of 8 spaces instead of 4.
Line 288: Indention of 8 spaces instead of 4.
Line 289: Indention of 8 spaces instead of 4.
Line 416: Indention of 8 spaces instead of 4.
Line 540: Indention of 8 spaces instead of 4.
Line 569: Indention of 8 spaces instead of 4.

In all the 6 cases the indention happens to be in the list of parameters insertion of a method call: even though it is not present in the check-list, it should be noted that it might be a convention used by the development team.

4.3 BRACES

Line 212-213: No braces are used to surround the if-statement. Even though it consists of a single statement, it should be surrounded by curly braces.
Line 262-263: No braces are used to surround the if-statement. Even though it consists of a single statement, it should be surrounded by curly braces.
Line 313-314: No braces are used to surround the if-statement. Even though it consists of a single statement, it should be surrounded by curly braces.
Line 395-396: No braces are used to surround the if-statement. Even though it consists of a single statement, it should be surrounded by curly braces.
Line 421-422: No braces are used to surround the if-statement. Even though it consists of a single statement, it should be surrounded by curly braces.
Line 524-525: No braces are used to surround the statement.
Line 546-547: No braces are used to surround the statement.

4.4 FILE ORGANIZATION

Several lines inside the exceed the 80 character length limit, but they are always performing a method call, or a class signature declaration, so are not considered ‘practical’.

Line 18-19:Line separator missing between sections.
Line 76:Line exceed average length (comment).
Line 81:Line exceed average length (comment).

Line 95:Line exceed average length (comment).
Line 96:Line exceed average length (comment).
Line 97:Line exceed average length (comment).
Line 108-109:Line separator missing between sections.
Line 112:Line exceed average length (comment).
Line 146:Line exceed average length (comment).
Line 149-150:Line separator missing between sections.
Line 155-156:Line separator missing between sections.
Line 163:Line exceed average length (comment).
Line 175:Line exceed average length (comment).
Line 192:Line exceed 120 characters.
Line 201-202:Line separator missing between sections.
Line 202-203:Line separator missing between sections.
Line 209:Line exceed average length (comment).
Line 215:Line exceed average length (comment).
Line 218:Line exceed average length (comment).
Line 231:Line exceed 120 characters.
Line 233:Line exceed 120 characters.
Line 234:Line exceed 120 characters.
Line 251-252:Line separator missing between sections.
Line 252-253:Line separator missing between sections.
Line 284: Line exceeds 80 characters (method call).
Line 287: Line exceeds 80 characters (method call with a manually inserted String parameter).
Line 288: Line exceeds 80 characters (method call with a manually inserted String parameter).
Line 306-307: Line separator missing between sections.
Line 307-308: Line separator missing between sections.
Line 329-330: Line separator missing between sections.
Line 330-331: Line separator missing between sections.
Line 348-349: Line separator missing between sections.
Line 349-350: Line separator missing between sections.
Line 387-388: Line separator missing between sections.
Line 388-389: Line separator missing between sections.
Line 432-432: Line separator missing between sections.
Line 433-434: Line separator missing between sections.
Line 449: Line exceed average length (comment).
Line 515-516: Line separator missing between sections.
Line 579-580: Line separator missing between sections.
Line 580-581: Line separator missing between sections.
Line 599-600: Line separator missing between sections.
Line 600-601: Line separator missing between sections.
Line 610: Line exceed average length (comment).

Line 624-625: Line separator missing between sections.
Line 625-626: Line separator missing between sections.
Line 635: Line exceed average length (comment).

4.5 WRAPPING LINES

Line 233-234: Line break before the operator '+'.
Line 287-288-289: Line break occurs before the operator '+'.
Line 449: missing comma at the end of the comment.
Line 465: missing comma at the end of the comment.
Line 471: missing comma at the end of the comment.
Line 483: missing comma at the end of the comment.
Line 489: missing comma at the end of the comment.
Line 545: missing comma at the end of the comment.
Line 568-569: Line break before the operator '+'.
Line 610: missing comma at the end of the comment.
Line 635: missing comma at the end of the comment.

4.6 COMMENTS

IfCompareField (Class)
IfEmpty
IfEntityPermission
IfHasPermission
IfRegExp

All these classes are briefly commented before their declaration: however, the comments do not explain their function inside the main class and direct the reader to the project documentation for further information. All of them, implementing a different evaluation condition for the correct rendering of the widget, implement the method eval, for which sometimes there are some other comments explaining its behaviour depending on the widget evaluation condition.

IfServicePermission (Class): The class is briefly commented to explain its function, although the comment itself is not explanatory but redirect elsewhere. Inside the class there are some comments to explain the 'eval' method pieces of code, but not the method itself.

IfValidateMethod (Class): The class is briefly commented to explain its function, although the comment itself is not explanatory but redirect elsewhere. Inside the class there are only one comment to explain a piece of code of the 'eval method; the method itself is not explained.

Not/Or/Xor (Classes): The classes are briefly commented to explain their function, although the comment itself is not explanatory but redirect elsewhere. Inside the classes there are only one comment to explain a piece of code of the ‘eval method; the method itself is not explained.

4.7 JAVA SOURCE FILE

Due to a different organization of the class, in order to achieve the Nested Class paradigm, many of the points in the checklist given regarding the Source File could not be respected. The class presents many nested public classes, which doesn’t seem to have any resemblance of order, because for example, classes inheriting from ***ModelWidgetCondition*** are written before and after ***DefaultModelFactory*** (implementing the ***ModelFactory*** interface).
The javadoc is also incomplete, because there is no mention of the class being unused inside the framework.

4.8 PACKAGES

No issues have been found.

4.9 CLASS AND INTERFACES DECLARATIONS

All classes and interfaces are coded using the correct practice described in the checklist (name, class static attributes (public, protected, package, private), instance attributes (same order as the static ones), constructor and then methods.

4.10 INITIALIZATION AND DECLARATION

Line 230: Variable is not declared at the beginning of a block.
Line 236: Variable is not declared at the beginning of a block.
Line 283: Variable is not declared at the beginning of a block.
Line 405: Variable is not declared at the beginning of a block; actually, its initialization depends on a following computation. However, depending on the execution flow the variable could possibly be never initialized.
Line 423: Variable is not declared at the beginning of a block.
Line 463: Variable is not declared at the beginning of a block.
Line 472: Variable is not declared at the beginning of a block.
Line 473: Variable is not declared at the beginning of a block.
Line 475: Variable is not declared at the beginning of a block.
Line 484: Variable is not declared at the beginning of a block.
Line 490: Variable is not declared at the beginning of a block.
Line 548: Variable is not declared at the beginning of a block.

Line 549: Variable is not declared at the beginning of a block.
Line 550: Variable is not declared at the beginning of a block.
Line 557: Variable is not declared at the beginning of a block.
Line 564: Variable is not declared at the beginning of a block.
Line 463: Variable is not declared at the beginning of a block.
Line 463: Variable is not declared at the beginning of a block.

4.11 METHOD CALLS

No issues have been found.

4.12 ARRAYS

Line 548-549: Two arrays are created, but they are not prevented from going out of bounds.

4.13 OBJECT COMPARISON

Line 164: two object are compared with '=='.
Line 227: two object are compared with '=='.

In general, no correct policy of object comparison is used throughout the code. Several examples of comparison using '==' instead of method ***equals*** can be found in line 164,227,280,421,545. Similar use of incorrect use of object comparison is the use of "!=", instead of the boolean evaluation of the result of the method ***equals*** (***if (!a.equals(b))*** instead of ***if(a!=b)***) can be found in lines 363,451,464,482 and 537.

4.14 OUTPUT FORMAT

No issues have been found

4.15 COMPUTATION, COMPARISON, AND ASSIGNMENTS

There are not arithmetic expression evaluation and all the boolean expressions have only one term, for this reason there are not such issues.
All the comparison are correctly handle with the equals() operator.

4.16 EXCEPTIONS

All the catch block log the exception and/or throws another exception that will be handled by another class.

4.17 FLOW OF CONTROL

No ***switch*** statements are used inside the code of the class under inspection. Concerning the loops, only ***for*** statements are used, and the following observations can be formulated about their use:

- Line 98: the ***for*** loop is correctly initialized inside the implementation of the ***eval*** method. However, the dimension of the initialization is based on the dimension of a method parameter (***subElementList*** is obtained from the ***conditionElement*** parameter), for which there is no control on its actual content.

- Line 120, line 611, line 637: the ***for*** loop is correctly initialized inside the implementation of the ***eval*** method. However, the dimension of the initialization is based on the dimension of a class attribute, and there is no control on whether this attribute has been correctly initialized (even though, it should be properly done during the object construction).

- Line 237, line 292: the ***for*** loop is correctly initialized inside the implementation of the ***eval*** method. This time instead, the initialization parameter is based on the size of a previously checked parameter. The flow of control is correctly handled.

No other control flow issues have been found.

4.18 FILES

No external files are managed by the methods of this class.

6. EFFORT

	Cannas	Castiglioni	Loiacono
19.01	2	0	0
20.01	4	0	3
21.01	2	0	3
22.01	0	0	4
23.01	1	4	1
24.01	1	2	0
25.01	1	1	0
26.01	0	1	1
27.01	3	1	3
28.01	0	2	1
29.01	2	0	2
30.01	3	2	5
31.01	5	5	1
01.02	1	0	0
02.02	0	3	0
03.02	2	4	0
04.02	0	0	2
SUM	27	25	26