International
Virtual
Observatory
Alliance

Implementing Note of the Framework built around PDL
Version 0.1.1

IVOA Working Draft December 18, 2012

This version:
0.1.1: December 18, 2012 (SVN Revision 123)

Latest version:
N/A

Previous versions:

Working Group:
GWS / Theory

Editor(s):

Authors:
Carlo Maria Zwölf, Paul Harrison

Contents

1 Introduction

This document is a detailed and structured description (by this I means it is more user friendly than a JavaDoc) of the framework around PDL. In particular I explain how, starting from a PDL description, are implemented the automatic generation of validator algorithms and the dynamic Graphic User Interface (GUI).

2 The package structure

I'm looking for a tool for generating a nice package dependency graph starting from the Eclipse project. If you have the solution, it is welcome...

3 Java Objects corresponding to the PDL data Model

These object

are all generated using JAX-B on the PDL XML schema. The generated classes corresponds to the PDL

4 The CommonsObject package

This package contains the classes

- GeneralParameter.java
- GeneralPatameterAlgebra.java

4.1 The GeneralParameter.java class

This class is the core PDL and is used for handling all the parameters. It is a way for encapsulating the usual basic types (integer, double, boolean, string) into an object. Moreover the modular design of this class allow users to easily define new types.

This attributes of this class are

- a field *value* of String type,
- a field *type* of String type,
- a field *description* of String type,
- a field *visitor* of Ivisitor type (cf. paragraph 5 for the definition of this interface).

The constructor of this class takes as arguments the quadruplet *value, type, description, visitor* and invoke the method *visit(GeneralParameter)* on the object we are

creating. If all the test contained in the visitor (typically this methods verify if the value provided could be cast to the type contained into the *type* field) pass with no problem, then the object is created. If a problem is encountered, the constructor throws an *InvalidParameterException* exception explaining the reasons of the problem.

With this mechanisms, the validation of a GeneralParameter is automatically performed at its own construction. All the existing instances of GeneralParameter are natively validated. Moreover, just by modifying the content of the visitor class passed as arguments, developers could easily add support for new types.

4.2 The GeneralParameterAlgebra.java class

This class is implemented using the singleton pattern. In the following, let g_i be a family of GeneralParameter instances. This class contains the methods for computing:

- the sum $g_r = g_i + g_j$;
- the multiplication $g_r = g_i \cdot g_j$;
- the difference of $g_r = g_i - g_j$. ;
  For these last three items, if the both g_i and g_j are integer, the resulting GeneralParameter will be of integer type too. If one of the two is a real, then the result will be a real (this is internally hold using double types).
- the power $g_r = g_i^{g_j}$. Since the native Java Math.pow method provide a double, the result will always be a GeneralParameter of real type;
- the absolute value $|g_r|$. Since the native Math.abs method provide a double, the result will always be a GeneralParameter of real type;
- the function $f(g_i)$, $f() \in \{\sin(), \cos(), \tan(), \sin^{-1}(), \cos^{-1}(), \tan^{-1}(), \exp(), \log()\}$. The result wil be a GeneralParamter of real type;
- the sum of all the components $g_{i,j}$ of a vector of general parameters $\vec{g}_i$: $g_r = \sum_j g_{i,j}$. By the choice of our implementation, the result is a GeneralParameter of real type;
- the product of all the components $g_{i,j}$ of a vector of general parameters $\vec{g}_i$: $g_r = \prod_j g_{i,j}$. By the choice of our implementation, the result is a GeneralParameter of real type;
- the size of a vector $\vec{g}_i$. In this case the result is a generalParameter of type Intger and the incapsulated value will be the size of the vector.

All these operations (excepted the last one) are performed only if g_i (and g_j too when it appears) is (are) numerical. In the other cases, this method will throw an InvalidEx-

pression exception explaining the reasons of the problem;

Moreover this class contains methods for characterizing instances of GeneralParameter:

- *areGeneralParameterEqual* takes g_i and g_j and return a boolean. This last is true if
  - both g_i, g_j are numerical and the difference of the values $v(g_i)$ and $v(g_j)$ encapsulated in these objects is smaller than $\epsilon = 0.0000001$ ($v(g_i) - v(g_j) < \epsilon$), or
  - the type of g_i is equal (in the Java equalsIgnoreCase String sense) to the type of g_j and the value if g_i is equal (again, in the Java equalsIgnoreCase String sense) to the value of g_j.
- *isFirstGreaterThanSecond* takes g_i, g_j and a boolean r. The result is true if
  - r is true and $v(g_i) \geq v(g_j)$, or
  - r is false and $v(g_i) > v(g_j)$.
- *isFirstSmallerThanSecond* takes g_i, g_j and a boolean r. The result is true if
  - r is true and $v(g_i) \leq v(g_j)$, or
  - r is false and $v(g_i) < v(g_j)$.
- *isGeneralParameterInteger* takes g_i and returns a boolean. This last is true if $v(g_i) \in \mathbb{N}$.
- *isGeneralParameterReal* takes g_i and returns a boolean. This last is true if $v(g_i) \in \mathbb{R}$.

5 The visitors package

This package contains the visitors classes used by the *GeneralParameter* class constructor (cf. paragraph 4.1).

5.1 The visitors objects

The interface *Ivisitor* describe the method *visit* which takes as argument the *GeneralParameter* instance to inspect.
The abstract *AbstractVisitor* class implements the *Ivisitor* interface. It defines

- the abstract method *buildCriteriaList* which return a list of *Icriteria* (cf. par. 5.2). Developer has to implement this function in order to define his own list of criteria. By default, we provide the *GeneralParameterVisitor* class (cf. the end of this paragraph).
- the *visit* function: the previous method is invoked and, for every Icriteria object returned, we verify if the couple (type, value) of the *GeneralParameter* instance passed as argument verify the criterion. If at least one criterion is satisfied, than

the visit methods end positevely. If no criterion is satisfied, the methods throws an *InvalidParameterException* explaining the reasons of the problem.

The *GeneralParameterVisitor* class is the concrete implementation of *AbstractVisitor*. It defines the method *buildCriteriaList*. The list returned contains a *RealCriteria*, an *IntegerCriteria*, a *BooleanCriteria* and a *StringCriteria* (cf. par. 5.2).

5.2 The criteria objects

All the criteria we are going to define implements the *Icriteria* interface. This last describe the methods:

- *getAuthorizedCriteriaType*, which returns the String containing the type authorized by the concrete criterion implementing the interface.
- *verifyCriteria*, which takes as argument a couple (type,value) characterizing a *GeneralParameter*. It returns the boolean true if the criterion is satisfied and false in the other case.

This interface is implemented by *RealCriteria*, an *IntegerCriteria*, a *BooleanCriteria* and a *StringCriteria*.

5.2.1 The RealCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('real') **used in the PDL grammar** for specifying that a parameter is of real type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'real' and if the value of the same *GeneralParameter* could be cast to a double type with no errors. If the type of the *GeneralParameter* is not 'real', this methods return false. Finally if the type of the *GeneralParameter* is 'real' and the value cannot be casted to a double, this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.2 The IntegerCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('Integer') **used in the PDL grammar** for specifying that a parameter is of Integer type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'Integer' and if the value of the same *GeneralParameter* could be cast to an Integer type with no errors. If the type of the *GeneralParameter* is not 'Integer', this methods return false. Finally if the type of the *GeneralParameter* is 'Integer' and the value cannot be casted to an Integer, this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.3 The BooleanCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('Boolean') **used in the PDL grammar** for specifying that a parameter is of Boolean type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'Boolean' and if the value of the same *GeneralParameter* is equal to 'true' or 'false'. If the type of the *GeneralParameter* is not 'Boolean', this methods return false. Finally if the type of the *GeneralParameter* is 'Boolean' and the value is not 'true' or 'false', this method throws an *InvalidParameterException* explaining the reasons of the problem.

5.2.4 The StringCriteria

In this class, the method *getAuthorizedCriteriaType* returns the String ('String') **used in the PDL grammar** for specifying that a parameter is of String type.
The *VerifyCriteria* method returns the boolean true if the type of the *GeneralParameter* is 'String' and the boolean false in the other cases.

6 The pdl.interpreter.expression package

In this package are contained all the classes for interpreting and parsing PDL expressions.
The entry point for understanding the expression interpreting mechanism is the abstract class *ExpressionParser*. It describes the abstract method *parse*, which interprets the expression invoking the method. The result of this method is a list of *GeneralParameter* instances. Since in PDL expressions are vectorial, the i-th element of that list corresponds to the result of the interpretation of i-th component of the vector expression. The abstract class *ExpressionParser* is used, jointly with the polymorphism mechanism for building the *ExpressionParserFactory*

6.1 The ExpressionParserFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildParser* takes as argument an instance of the PDL *Expression* object and returns an *ExpressionParser*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *Expression* is an instance of *AtomicConstantExpression* then the methods returns a *AtomicConstantExpressionParser*;
- if the *Expression* is an instance of *AtomicParameterExpression* then the methods returns a *AtomicParameterExpressionParser*;
- if the *Expression* is an instance of *FunctionExpression* then the methods returns a *FunctionExpressionParser*;
- if the *Expression* is an instance of *Function* then the methods returns a *Function-Parser*;

- if the *Expression* is an instance of *ParenthesisContent* then the methods returns a *ParenthesisContentParser*;
- if the *Expression* is not an instance of what seen in the previous items, the function throws an *InvalidExpression* exception.

It is important to note that, since in PDL expressions are recursive, this factory will be invoked inside every concrete class implementing *ExpressionParser*.
In what follows, we are going to describe all these concrete classes returned by the *buildParser* method (and implementing the *ExpressionParser*).

6.2 The ExpressionWithPowerParser class

This abstract class inherits from *ExpressionParser* and is used for factoring the code of all the classes interpreting PDL expressions involving powers. It specifies the method *evaluatePower* which takes as arguments two lists of *GeneralParameter*, one list for the base (let us note g_i^{base} its elements) and the other for the exponent (let us note g_i^{exp} its elements). This method returns:

- the base list of g_i^{base}, if the expos ant list is null;
- the list whose elements are $\left(g_i^{base}\right)^{g_i^{exp}}$, if both lists have the same size. The power operation is performed in the *GeneralParameter* sense, using the methods described in paragraph 4.2.
- the list whose elements are $\left(g_i^{base}\right)^{g_0^{exp}}$, if the exponent list contains only one element g_0^{exp}.

In the other cases, the *evaluatePower* method throws and *InvalidExpression* exception, specifying the reasons of the problem.

6.3 The AtomicParameterExpressionParser class

This class extends the *ExpressionWithPowerParser* and is used for interpreting the PDL AtomicParameterExpression instances. Indeed this class has an attribute of type *AtomicParameterExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- the instance of the SingleParameter referenced by the expression, is retrieved using the *Utilities* class methods.
- if the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: Using the *Utilities* class we get the list of *GeneralParameter* corresponding to the user provided input vector for the SingleParameter contained into the AtomicConstantExpression. We verify that the dimension of that list corresponds to the value put in the dimension field

of the SingleParameter. If this test is negative we throws an *InvalidExpression* exception. If the test is positive, we invoke the method *evaluatePower* inherited from the *ExpressionWithPowerParser* superclass.
- If the *Operation* contained into the *AtomicParameterExpression* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case our first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.4 The AtomicConstantExpressionParser class

This class extends the *ExpressionWithPowerParser* and is used for interpreting the PDL AtomicConstantExpression instances. Indeed this class has an attribute of type *AtomicConstantExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: first we recover the set of constant values provided in the description of the AtomicConstantExpression. Then we try to instantiate a list of *GeneralParameter* whose values are the previously retrieved and the type is the one contained into the ConstantType attribute (of type ParameterType) contained into the AtomicConstantExpression. If no exception is thrown during this process, the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked.
- If the *Operation* contained into the *AtomicConstantExpression* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.5 The ParenthesisContentParser class

This class extends the *ExpressionWithPowerParser* and is used for interpreting the PDL *ParenthesisContent* expression instances. This class has an attribute of type *ParenthesisContent* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The expression 'contained into the parenthesis', i.e. with the highest priority, is first evaluated by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*.
- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.

- The expression (without the operation part) is evaluated: the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked using as base the list of *GeneralParameter* corresponding to the expression contained into the parenthesis and as exponent the list of *GeneralParameter* corresponding to the power.
- If the *Operation* contained into the *ParenthesisContent* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.6 The FunctionParser class

This class extends the *ExpressionParser* and is used for interpreting PDL *Function* expressions.
This class has an attribute of type *Function* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The argument of the function is interpreted by invoking by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- According to the function type contained in the attribute *functionName* of the *Function* object, we invoke the good correspondent method contained into the GeneralParameterAlgebra class (cf. 4.2) passing the list of *GeneralParameter* (built during the previous step) as argument. The result of this invocation is returned by the function.

6.7 The FunctionExpressionParser class

This class extends the *ExpressionWithPowerParser* and is used for interpreting the PDL *FunctionExpression* expression instances. This class has indeed an attribute of type *FunctionExpression* whose value is initialized by the class constructor.
In the overridden method *parse*, the following actions are performed:

- The *Function* contained into the FunctionExpression is interpreted by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- If the power of the expression is not null, then we convert the power expression into a List of *GeneralParameter* by building (using the *ExpressionParserFactory*) the had hoc parser and invoking its *parse* method.
- The expression (without the operation part) is evaluated: the method *evaluatePower* (inherited from the *ExpressionWithPowerParser* superclass) is invoked using as base the list of *GeneralParameter* corresponding to the *Function* expression contained into the *FunctionExpression* expression and as exponent the list of *GeneralParameter* corresponding to the power.

- If the *Operation* contained into the *ParenthesisContent* is not null, we evaluate the results of this operation (cf 6.8, the result of the expression without the operation is in this case the first operand).
- Finally, the methods returns the list of *GeneralParameter* resulting from the previous stages.

6.8 The OperationParser class

This class is used for interpreting the results of operations expressed in PDL syntax. This class has indeed an attribute of *Operation* type, whose value is initialized by the class constructor. The only method of this class is *processOperation*, which takes as argument a list of *GeneralParameter* representing the first operand of the operation. In this last method, the following operations are performed:

- The second operand of the operation (i.e. the *Expression* instance contained into the *Operation* object) is interpreted by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*;
- We get the *OperationType* from the *Operation* object and invoke the good corresponding method contained into the GeneralParameterAlgebra class (cf. 4.2) passing the *GeneralParameter* list corresponding to the first and second operands as arguments. The function returns the result of this last invokation.

7 The pdl.interpreter.condition package

In this package we put all the classes used for interpreting the instances of all the PDL objects implementing *AbstractCondition*.
The entry point for understanding the mechanisms of this package is the abstract class *ConditionInterpreter*. This last describe the abstract method *isConditionVerified* which a boolean value. True id the condition is verified, false in the opposite case. This method could throws three kind of exceptions: *InvalidExpression, InvalidParameterException, InvalidCondition*.
The abstract class *ConditionInterpreter* is used, jointly with the polymorphism mechanism for building the *ConditionInterpreterFactory*

7.1 The ConditionInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildConditionInterpreter* takes as argument an instance of the PDL *AbstractCondition* object and returns a *ConditionInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *AbstractCondition* is an instance of *ValueLargerThan* then the methods returns a *ValueLargerThanInterpreter*;

- if the *AbstractCondition* is an instance of *ValueSmallerThan* then the methods returns a *ValueSmallerThanInterpreter*;
- if the *AbstractCondition* is an instance of *ValueInRange* then the methods returns a *ValueInRangeInterpreter*;
- if the *AbstractCondition* is an instance of *ValueDifferentFrom* then the methods returns a *ValueDifferentFromInterpreter*;
- if the *AbstractCondition* is an instance of *BelongToSet* then the methods returns a *BelongToSetInterpreter*;
- if the *AbstractCondition* is an instance of *DefaultValue* then the methods returns a *DefaultValueInterpreter*;
- if the *AbstractCondition* is an instance of *IsInteger* then the methods returns a *IsIntegerInterpreter*;
- if the *AbstractCondition* is an instance of *IsReal* then the methods returns a *IsRealInterpreter*;
- if the *AbstractCondition* is an instance of *IsNull* then the methods returns a *IsNullInterpreter*;
- if the *AbstractCondition* is not an instance of what seen in the previous items, the function throws an *InvalidCondition* exception, explaining the origin of the problem.

In what follows, we are going to describe all these concrete classes returned by the *buildConditionInterpreter* method (and implementing the *ConditionInterpreter*).

7.2 The ValueLargerThanInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *ValueLargerThan* PDL condition. Indeed this class has an attribute of type *ValueLargerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{cond}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueLargerThan* condition.
- If the lists built during the two previous steps has different sizes, we throw a *InvalidCondition* expression expelling the problem origin.
- For every couple (g_i^{exp}, g_i^{cond}) we call the method *isFirstGreaterThanSecond* of *GeneralParameterAlgebra* (by setting the boolean field reached according to the value contained in the attribute *reached* in the *ValueLargerThan* condition, cf. paragraph 4.2). The method returns the true boolean value if and only if all the results of this calls are true for all the couples. It returns false otherwise.

7.3 The ValueSmallerThanInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *ValueSmallerThan* PDL condition. Indeed this class has an attribute of type *ValueSmallerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{cond}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueLargerThan* condition.
- If the lists built during the two previous steps has different sizes, we throw a *InvalidCondition* expression expelling the problem origin.
- For every couple (g_i^{exp}, g_i^{cond}) we call the method *isFirstSmallerThanSecond* of *GeneralParameterAlgebra* (by setting the boolean field reached according to the value contained in the attribute *reached* in the *ValueSmallerThan* condition, cf. paragraph 4.2). The method returns the true boolean value if and only if all the results of this calls are true for all the couples. It returns false otherwise.

7.4 The ValueInRangeInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *ValueInRange* PDL condition. Indeed this class has an attribute of type *ValueSmallerThan* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We interpret the *Sup* field of the *ValueInRange*, which is of type *ValueSmallerThan* (cf. par. 7.3)
- We interpret the *Inf* field of the *ValueInRange*, which is of type *ValueLargerThan* (cf. par. 7.2)
- We returns true if both the previous interpretations returned true, false otherwise.

7.5 The BelongToSetInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *BelongToSet* PDL condition. Indeed this class has an attribute of type *BelongToSet* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- For every expression $value_j$ appearing in the field *Value* of the *BelongToSet* object
  - We interpret $value_j$, by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*. Let $\vec{g}^{value_j}$ be this result.

– If the list $\vec{g}^{value_j}$ resulting from this interpretation is equal to $\vec{g}^{exp}$ (i.e. $\forall i$, $g_i^{exp} = g_i^{value_j}$ in the sense defined by the method *areGeneralParameterEqual* of *GeneralParameterAlgebra*, cf. par. 4.2), then the function returns true. False otherwise.

7.6 The ValueDifferentFromInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *ValueDifferentFrom* PDL condition. Indeed this class has an attribute of type *ValueDifferentFrom* whose value is initialized by the class constructor.
In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- We build $\vec{g}^{value}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* contained into the field *Value* of the *ValueDifferentFrom* object.
- The methods return the boolean true if, at least for a given of i, we have $g_i^{exp} \neq g_i^{value}$.

7.7 The IsRealInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *IsReal* PDL condition. In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- The method returns the boolean true if for all i, g_i^{exp} is real (the test is performed by calling the method *isGeneralParameterReal* of the *GeneralParameterAlgebra* class, cf. par. 4.2).

7.8 The IsIntegerInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *IsInteger* PDL condition. In the overridden *isConditionVerified* method, the following actions are performed:

- We build $\vec{g}^{exp}$, the list of *GeneralParameter* resulting from the interpretation of the *Expression* passed as argument to the function (by calling the *ExpressionParserFactory* and invoking on its returned object the method *parse*).
- The method returns the boolean true if for all i, g_i^{exp} is integer (the test is performed by calling the method *isGeneralParameterInteger* of the *GeneralParameterAlgebra* class, cf. par. 4.2).

7.9 The IsNullInterpreter class

This class extends the *ConditionInterpreter* and is used for interpreting the *IsNull* PDL condition.
It is important to remark that, due to the signification of the *IsNull* condition, this last could be applied only on *AtomicParameterExpression* (because it has sense to say parameter p_1 is null, but has no sense to say $(p_1 \cdot p_2/p_3)$ is null.
In the overridden *isConditionVerified* method, the following actions are performed:

- If the *Expression* passed as parameter is not an *AtomicParameterExpression* we throw an *InvalidCondition* exception, explaining the problem source.
- We get the *SingleParameter* instance from the *ParameterReference* contained into the *Expression*.
- We look[1] for the value (or values for vector Parameter cases) provided by user for the *SingleParameter*. If no value is provided, the method return true. Otherwise, it returns false.

7.10 The DefaultValueInterpreter class

This class extends the *ConditionInterpreter* and is used in association *DefaultValue* PDL condition.
Since this particular condition don't need to be interpreted as the previous ones (it is used only for saying that the default value of a parameter is a given value) the overridden method *isConditionVerified* always returns true.

8 The pdl.interpreter.criterion package

In this package we put all the classes used for interpreting instances of all the PDL objects implementing *AbstractCriterion*. The entry point for understanding the mechanism of this package is the abstract class *AbstractCriterionInterpreter*. This last describe the abstract method *isCriterionSatisfied* which returns a boolean value: true if the criterion is satisfied, false in the opposite case. This method could throw four kind of exceptions: *InvalidExpression, InvalidParameterException, InvalidCondition, InvalidCriterion*.
The class *AbstractCriterionInterpreter* is used, jointly with the polymorphism mechanism for building the *CriterionInterpreterFactory*.

8.1 The CriterionInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildCriterrionInterpreter* takes as argument an instance of the PDL *AbstractCriterion* object and returns a *AbstractCriterionInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

[1] by calling the *getuserProvidedValuesForParameter* of the *Utilities* class

- if the *AbstractCriterion* is an instance of *Criterion* then the methods returns a *CriterionInterpreter*;
- if the *AbstractCriterion* is an instance of *ParenthesisCriterion* then the methods returns a *ParenthesisCriterion*;
- if the *AbstractCriterion* is not an instance of what seen in the previous items, the function throws an *InvalidCriterion* exception, explaining the origin of the problem.

In what follows, we are going to describe these two concrete classes returned by the *buildCriterrionInterpreter* method (and implementing the *AbstractCriterionInterpreter*).

8.2 The CriterionInterpreter class

This class extends the *AbstractCriterionInterpreter* and is used for interpreting the *Criterion* PDL criterion.
Indeed this class has an attribute of type *Criterion* whose value is initialized by the class constructor.
In the overridden *isCriterionSatisfied* method, the following actions are performed:

- We interpret the condition *ConditionType* contained into the *Criterion* object. For this we call the method *buildConditionInterpreter* of *ConditionInterpreterFactory* (cf. par 7.1). We call the method *isConditionVerified* (cf. par. 7) contained in the object returned by *buildConditionInterpreter* by passing as argument the *Expression* contained into the *Criterion* object. This returns the boolean value b_1.
- If the *Criterion* object has no *LogicalConnector*, we return the boolean obtained at the previous item.
- If the *Criterion* object has a *LogicalConnector*, we interpret it (as explained in paragraph 8.4), by passing to the method *interpret* of *LogicalConnectorInterpreter* b_1 as parameter. Then the method returns the result of this last interpretation.

8.3 The ParenthesisCriterionInterpreter class

This class extends the *AbstractCriterionInterpreter* and is used for interpreting the *ParenthesisCriterion* PDL object. Indeed it contains an attribute of type *ParenthesisCriterion* whose value is initialized by the class constructor. We recall that in PDL syntax, the *ParenthesisCriterion* are used for defining complex criteria, with arbitrary evaluation priority fixed by the user.
In the overridden method *isCriterionSatisfied* the following actions are performed:

- The *AbstractCriterion* contained into the *ParenthesisCriterion* is interpreted by calling the method *isCriterionSatisfied* contained into the object returned by the method *buildCriterionInterpreter* (of the class *CriterionInterpreterFactory*, cf. par. 8.1). Let b_1 be the boolean result of this interpretation.
- If the *ParenthesisCriterion* has no *ExternalLogicalCriterion*, the method returns b_1.

- If the *ParenthesisCriterion* has no *ExternalLogicalCriterion*, the methods return the boolean resulting from the call of the method

8.4 The LogicalConnectorInterpreter class

This class is used for interpreting the abstract *LogicalConnector* objects. Indeed, it contains an attribute of *LogicalConnector* type. We recall that the two concrete classes implementing *LogicalConnector* are *And* and *Or*.
The method *interpret* takes as argument a boolean value b_1 (which is typically the result of the interpretation of a first Criterion, which contains the *LogicalConnector* we are trying to interpret) and returns a boolean value. The following actions are performed inside this method:

- The second criterion (i.e. the criterion contained into the LogicalConnector instance) is interpreted by calling the method *isCriterionSatisfied* contained into the object returned by the method *buildCriterionInterpreter* (of the class *CriterionInterpreterFactory*, cf. par. 8.1). Let b_2 be the result of this interpretation.
- If the instance of *LogicalConnector* is of type *And*, the boolean (b_1 and b_2) is returned.
- If the instance of *LogicalConnector* is of type *Or*, the boolean (b_1 or b_2) is returned.

9 The pdl.interpreter.conditionalStatement package

In this package we put all the classes used for interpreting the instances of all the PDL objects implementing *ConditionalStatement*.
The entry point for understanding the mechanisms of this package is the abstract class *ConditionalStatementInterpreter*. This last describe two abstract methods

- *isStatementSwitched* which returns a boolean value: true if the statement is switched (i.e. if we are in a case where the statement has to be considered), false otherwise.
- *isValidStatement* which returns a boolean value: true if the statement is valid, false otherwise.

The abstract class *ConditionalStatementInterpreter* is used, jointly with the polymorphism mechanism for building the *ConditionalStatementInterpreterFactory*.

9.1 The ConditionalStatementInterpreterFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The method *buildInterpreter* takes as argument an instance of the PDL *ConditionalStatement* object and returns a *ConditionalStatementInterpreter*. More precisely, using introspection, this function analyze the instance of the PDL Expression:

- if the *ConditionalStatement* is an instance of *AlwaysConditionalStatement* then the methods returns a *AlwaysConditionalStatementInterpreter*;

- if the *ConditionalStatement* is an instance of *IfThenConditionalStatement* then the methods returns a *IfThenConditionalStatementInterpreter*;
- if the *ConditionalStatement* is not an instance of what seen in the previous items, the function throws an *InvalidConditionalStatement* exception, explaining the origin of the problem.

In what follows, we are going to describe these two concrete classes returned by the *buildInterpreter* method (and implementing the *ConditionalStatementInterpreter*).

9.2 The AlwaysConditionalStatementInterpreter class

This class extends the *ConditionalStatementInterpreter* and is used for interpreting the *AlwaysConditionalStatement* PDL objects. Indeed this class has an attribute of *AlwaysConditionalStatement* type, whose value is initialized by the class constructor. Moreover it contains a boolean attribute *isStatementSwitched* which is set to true by the class constructor, since as the name indicates an *AlwaysConditionalStatement* is always valid and has no conditions to be verified.

- The overridden method *isStatementSwitched* is a getter on the *isStatementSwitched* and always returns true.
- The overridden method *isValidStatement* calls the method *buildCriterionInterpreter* of the factory class *CriterionInterpreterFactory* (cf. par. 8.1) by passing as argument the PDL *Criterion* object contained into the *Always* PDL object[2], which is in turn contained into the *AlwaysConditionalStatement* class attribute. The function returns the result of the method *isCriterionSatisfied* owned by the object returned by the *buildCriterionInterpreter* method.

9.3 The IfThenConditionalStatementInterpreter class

This class extends the *ConditionalStatementInterpreter* and is used for interpreting the *IfThenConditionalStatement* PDL objects. Indeed this class has an attribute of *IfThenConditionalStatement* type, whose value is initialized by the class constructor. Moreover it contains a boolean attribute *isStatementSwitched* whose value is computed by the method *isStatementSwitched*.

- In the overridden method *isStatementSwitched* the following actions are performed:
  - We call the method *buildCriterionInterpreter* of the factory class *CriterionInterpreterFactory* (cf. par. 8.1) by passing as argument the PDL *Criterion* object contained into the *If* PDL object (which is in turn contained into the *IfThenConditionalStatement* class attribute). If no exception is detected, the function returns the result of the method *isCriterionSatisfied* owned by the object returned by the *buildCriterionInterpreter* method.

[2]implementing the PDL abstract class *ConditionalClause*

  - In case of exception, we are not able to evaluate the condition, this means that the statement is not switched: the function return the boolean false.

- In the overridden method the following actions are performed:
  - The class attribute *isStatementSwitched* is set with the value returned by the *isStatementSwitched* method.
  - If the statement is not switched, the function returns true: indeed if the statement is not switched it is always valid, since the contained condition has no matter.
  - If the the statement is switched, it is valid only if the criterion contained into the *Then* PDL condition (which is in turn contained into the class attribute of type *IfThenConditionalStatement*) is valid. Indeed the function calls the method *buildCriterionInterpreter* of the factory class *CriterionInterpreterFactory* (cf. par. 8.1) by passing as argument the PDL *Criterion* object contained into the *Then* PDL object . The function returns the result of the method *isCriterionSatisfied* owned by the object returned by the *buildCriterionInterpreter* method.

9.4 The WhenConditionalStatementInterpreter class

This class extends the *ConditionalStatementInterpreter* and is used for interpreting the *WhenConditionalStatement* PDL objects. Indeed this class has an attribute of *IfThenConditionalStatement* type, whose value is initialized by the class constructor. Moreover it contains a boolean attribute *isStatementSwitched* whose value is computed by the method *isStatementSwitched*.

- In the overridden method *isStatementSwitched* the following actions are performed:
  - We call the method *buildCriterionInterpreter* of the factory class *CriterionInterpreterFactory* (cf. par. 8.1) by passing as argument the PDL *Criterion* object contained into the *When* PDL object (which is in turn contained into the *WhenConditionalStatement* class attribute). If no exception is detected, the function returns the result of the method *isCriterionSatisfied* owned by the object returned by the *buildCriterionInterpreter* method.
  - In case of exception, we are not able to evaluate the condition, this means that the statement is not switched: the function return the boolean false.

9.5 The StatementHelperContanier class

This class is a Java Bean used for easily handling the properties of the *ConditionalStatement* objects. It is used for facilitating communications between the logical layers we explained into the previous paragraphs and the IHM layer. It contains the

- *StatementComment* String attribute. It is used for storing the comments contained into the attribute *comment* of all *ConditionalStatement* objects.
- *statementSwitched* boolean attribute,
- *statementValid* boolean attribute,
- *statement* attribute of type *ConditionalStatement*,

and their related getters and setters.

10 The pdl.interpreter.groupInterpreter package

In this package we put all the classes used for interpreting the *ParameterGroup* objects. This package contains two classes

- *GroupHandlerHelper* which plays a role analogous to the *StatementHelperContainer* class (cf. par. 9.5).
- *GroupProcessor* which implements the business logic for processing all the group of a given PDL service.

10.1 The GroupHandlerHelper class

This class is used for easily handling properties of *ParameterGroup* objects. It contains the following attributes

- the String *fatherName* which indicates the name of the direct ancestor of the current group,
- the List of String *sonNames* which indicates the name of the direct offsprings of the current group,
- the String *groupName* which indicates the name of the current group,
- the *ParameterGroup group* which is the current group,
- the List of *SingleParameter* objects *singleParamsIntoThisGroup* which indicates the *SingleParameter* instances contained into the current group,
- the boolean *groupValid* which indicates if all the *ConditionalStatement* contained into the *ConstraintOnGroup* object of the current group are valid,
- the List of *StatementHelperContainer* objects *statementHelperList*, which associates to any *ConditionalStatement* contained into the *ConstraintOnGroup* object of the current group, the corresponding *StatementHelperContainer*.

With the exception of *setGroup* method, all the getters and setters associated to these attributes are usual.
In *setGroup*, first we set the attribute *group* according to the value passed as argument. Then we set the *groupName* and, by invoking the method *getParameterForTheGroup* of the *Utilities* class, the attribute *singleParamsIntoThisGroup*.

10.2 The GroupProcessor class

As already mentioned, this class implements the logic for processing the groups of a given PDL service.
This class has two attributes

- *service* of type the PDL *Service*,
- *groupsHandler* which is a List of *GroupHandlerHelper* objects (cf. par. 10.1), one for every *ParameterGroup* of the *service*,

two public methods

- *getGroupsHandler*, an usual getter on the *groupsHandler* attribute,
- *process* which process the groups of parameters composing the service,

and three private methods

- *buildGroupListFromService*, which builds the group structure from the PDL *Service*,
- *addGroups*, which adds a particular group to the *groupsHandler* List,
- *processStatementsOfGroups*, which process and interoperate all the *ConditionalStatment* contained into the *ConstraintOnGroup* object all the groups composing the service.

In the following we are going to explain in details the actions performed in these methods.

10.2.1 The process method

This void method, which takes no argument, initializes the value of the attribute *groupsHandler* by calling the method *buildGroupListFromService* (cf. par. 10.2.2) then call the method *processStatementsOfGroups* (cf. par. 10.2.4).

10.2.2 The buildGroupListFromService method

This method, which takes no argument, returns a List of *GroupHandlerHelper* objects. After initializing the list to be returned, it calls the *addGroups* method[3] (cf. par. 10.2.3), by passing as arguments

- the list that will be returned,
- the *Input* object of type *ParameterGroup* contained into the PDL *Service*,
- the String 'root'.

[3]Pay attention: the method *addGroups* will modify the list it receives as argument

10.2.3 The addGroups method

This void method has a recursive structure. It takes as arguments

- a List of *GroupHandlerHelper* (this list will be modified by the function),
- a *ParameterGroup* PDL object,
- a String containing the name of the father of this *ParameterGroup*.

In this method, the following actions are performed:

- A temporary *GroupHandlerHelper* is created. Its attribute *group* is set to the value passed as second argument and its attribute *fatherName* is set to the value passed as third argument.
- If the current group (i.e. the *ParameterGroup* passed as first argument) does not contain other *ParameterGroup* instances, we set the attribute *sonNames* of the temporary object to 'null' and we add the temporary object to the List of *GroupHandlerHelper* received as first argument. Then the method ends.
- If the current group contains other *ParameterGroup* instances, we build the List containing the names of all the direct offsprings of the current group and we copy this List into the attribute *sonNames* of the temporary object. Then for every 'son' of the current group, we call the method *addGroups* by passing as arguments
  - the list that will be returned (i.e. the list received as first argument),
  - the current 'son' group,
  - the name of the current group, which is the 'father' of the current 'son' group.

10.2.4 The processStatementsOfGroups method

This method, which takes no argument and returns no value, is always called by *process* (cf. par. 10.2.1) and its invocation is always preceded by the call of *buildGroupList-FromService* method (cf. par. 10.2.2). Indeed when one call *processStatementsOfGroups* the attribute *groupsHandler* of the *GroupProcessor* class is always well initialized and defined.
In this method, the following actions are performed:

- If the attribute List *groupsHandler* has no elements, the method does nothing.
- If the attribute List *groupsHandler* has elements, we loop on them and for every *GroupHandlerHelper* object of the list
  - We get the current *ParameterGroup* instance associated to the *GroupHandlerHelper* object,
  - If the *ParameterGroup* instance contains no constraint, we set the attribute *groupValid* of the current *GroupHandlerHelper* to true,
  - We build the List *statementList* of the *ConditionalStatement* PDL objects contained into the *ConstraintOnGroup* field of the current *ParameterGroup*.
  - If the List *statementList* is empty, we set the attribute *groupValid* of the current *GroupHandlerHelper* to true,

- If the List *statementList* is not empty, we initialize a List of *StatementHelperContainer* and we loop on the elements of *statementList*. For every statement in the list
  - We build a *ConditionalStatementInterpreter* by calling the method buildInterpreter (with the current statement as argument) of the *ConditionalStatementInterpreterFactory* (cf. par. 9.1).
  - We define a temporary *StatementHelperContainer* objects and
  - We set its *statement* attribute to the current statement,
  - We set its attribute *setStatementComment* to the comment contained into the current statement,
  - We set its *statementSwitched* attribute to the value returned by method *isStatementSwitched* owned by the previously built *ConditionalStatementInterpreter*.
  - We set its *statementValid* attribute to the value returned by the method *isStatementValid* owned by the previously built *ConditionalStatementInterpreter*.
  - If an exception occurs during the previous procedures, the *statementValid* attribute is set to the 'null' value, because we cannot determine the statement validity.
  - We add the temporary *StatementHelperContainer* to the previously created List of *StatementHelperContainer*
- We set the attribute *statementHelperList* of the current *GroupHandlerHelper* to the List of *StatementHelperContainer* previously built.

11 The gui.dynamicLabel package

With the mechanisms we exposed int the previous paragraphs, the framework can automatically check (by invoking the *process* method of the *GroupProcessor* class, cf. respectively par. 10.2.1 and 10.2) if all the parameters provided by the user for a given PDL Service are consistent with the expressed description and related constraints. But this is not sufficient for building a dynamic GUI.
Let explore this point with a little example: consider the following description/constraint (where p_i are parameters):

$$\text{if } p_1 > 0 \text{ then } p_2 \in \{2; 3; 4; 5; 8\}$$

The program is indeed able to verify if a couple (p_1, p_2) conforms the constraint: e.g. $(-1, 1)$, $(0, -3)$, $(1, 3)$, $(1, 5)$ will be accepted whereas $(1, 1)$, $(1, 7)$, $(1, 10)$ will be refused. But we still don't have a mechanism for tell to user, in case he/she/it (in case user is another computer system) provide only $p_1 > 0$ (and no value for P_2) that this parameter p_2 must be in the set of values $\{2; 3; 4; 5; 8\}$.
In this package we put all the classes that will provide this missing functionality.
The entry point for understanding the mechanism of this package, is the abstract class

PDLBaseParamPanel. The class PDLBaseParamPanel is used, jointly with the polymorphism mechanism for building the *PDLParamPanelFactory* (cf. par. 11.1.1).

11.1 The PDLBaseParamPanel class

This abstract class extends *Jpanel* and extends *FocusListener*. It is used for

- displaying in the GUI the information of a given parameter
- set/modify the value(s) of this parameter.

The class contains the attributes

- *paramLabel* of type *Jlabel*, which is the label used in the GUI for describing the parameter,
- *paramName* of type String, used for storing the name of the parameter,
- *paramUnit* of type String, used for storing the unit of the parameter,
- *paramType* of type String, used for storing the type of the parameter,
- *paramDimension* of type String, used for storing the dimension (we recall PDL handle vector parameters) of the parameter,
- *skossConcept* of type String, used for storing the SKOS concept of the parameter.

has three public methods

- *getParamName* an usual getter on the *paramLabel* attribute.
- *PDLBaseParamPanel* a constructor which will be invoked (with the super() syntax) by all the classes implementing this abstract one.
- *verify* which performs some basic validation on the values provided by the user for the parameter,

five protected methods

- *getComponent* which is abstract and returns a *JComponent* instances,
- *setComponentValue* which is abstract and sets the value of the *JComponent* returned by the previous method,
- *initializeComponent* which initializes the component handled by the two previous methods,
- *getUserProvidedValue* which is abstract and returns the String corresponding to the values entered by the user for the parameter,
- *convertToStringProvidedValues* which convert the parameter already provided by the user into a String that could be displayed into the GUI.

and one private method *buildLabelText* which builds the text to put into the label used for describing the parameter.
Remark: All the abstract methods appearing in the previous items will be detailed in paragraphs 11.3, 11.4 and 11.5 , where we will describe the concrete classes extending the abstract *PDLBaseParamPanel*.

11.1.1 The PDLBaseParamPanel method

This constructor, which takes as argument a *SingleParameter* instance, will be invoked (with the super() syntax) by all the classes implementing this abstract one. The following actions are performed:

- The attributes *paramName*, *paramUnit*, *paramType*, *skossConcept* and *paramDimension* are initialized starting from the *SingleParameter* received as argument. We pay attention to the *paramDimension*: since this last is a PDL *Expression* it interpretation could leas to exception. In this case this attribute is set to 'null'.
- We set the Layout of this component to a GridLayout.
- We affect to the attribute *paramLabel* the value of a new JLabel whose text is initialized by calling the private method *buildLabelText* (cf. par. 11.1.5).

11.1.2 The verify method

This method performs an atomic validation of the parameter (i.e. only on the parameter, without considering the constraints where it is involved). It takes no arguments and returns no value. The following actions are performed

- By calling the method *getUserProvidedValue* we get the set of values provided by the user for the parameter (by convention, for providing a N-size vector $\vec{v}$, user will provide the String '$v_1; v_2; ...; v_N$').
- If the dimension of the current parameter could be evaluated with no exception, we compare the dimension of vector $\vec{v}$ with the expected dimension. If these two number are different, a message dialog window is shown notifying this error,
- We loop on every element of $\vec{v}$: for every value v_i we try to build the corresponding *GeneralParameter* g_i (by providing to the *GeneralParameter* constructor the triplet (v_i, the type of the current parameter, an instance of *GeneralParameterVisitor*)). If an exemption is intercepted during this procedure, we show a message dialog window notifying the origin of the problem. If no exemption is detected, every g_i is added to a vector $\vec{g}$ which is the *GeneralParameter* representation of the values entered by the user.
- The vector $\vec{g}$ associated to the parameter is updated into the internal representation system (the Mapper, cf. 12) using the Utilities class (cf. 12).

11.1.3 The initializeComponent method

This method is used for initializing the graphic component represented by this class (which we recall extends JPanel). It takes no arguments and returns no value. The following actions are performed:

- We set the attribute *skossConcept* as a tooltip for the instance of *JComponent* returned by *getComponent*. This component is added to the current instance of *PDLBaseParamPanel*,

- We add to the the current instance of *PDLBaseParamPanel* the attribute *paramLabel*.
- We call the *setComponentValue* method
- We add the the current instance of *PDLBaseParamPanel* as a FocusListener to the the instance of *JComponent* returned by *getComponent*.

11.1.4 The convertToStringProvidedValues method

This method takes as argument a N-size vector $\vec{g}$ of *GeneralParameter* corresponding to a given parameter and returns a String that will be displayed into the GUI. This String will be equal to 'v_1; v_2; ... ; v_N', where v_i is the value contained into the *GeneralParameter* g_i, $\forall i \in [1, N]$.

11.1.5 The buildLabelText method

This method builds the String label that will inform the user about the nature of the current parameter. It takes no argument and returns the built String. This last is built as follows: the attribute *paramName* is followed by a parenthesis containing the attributes *paramUnit*, *paramType* and *paramDimension*.

11.2 The PDLParamPanelFactory class

This class is built by implementing the *singleton* pattern and is, as its name indicates, an implementation of the *factory* pattern.
The public method *paramBuilder* takes as arguments an instance of *GroupHandlerHelper* and a *SingleParameter* instance. It returns a *PDLBaseParamPanel*. Four private methods are used (directly or indirectly) by *paramBuilder*: *buildBasicPanel*, *getCriterionWhereParamIsInvolved*, *getCriterionFromStatement* and *test*.

11.2.1 The buildBasicPanel method

This private method takes as argument an instance of *SingleParameter* s and returns a *PDLBaseParamPanel* (cf. par. 11.1.1). If s is a boolean the function returns a *PDLBooleanParamPanel* (cf. par PDLBooleanParamPanel). It returns a *PDLTextParamPanel* (cf. par PDLTextParamPanel) otherwise.

11.2.2 The getCriterionFromStatement method

This private method takes as argument an instance of *StatementHelperContainer* (cf. par. 9.5) and returns a PDL *AbstractCriterion*. If the *Statement* into the *StatementHelperContainer* is a *AlwaysConditionalStatement* the function returns the *Criterion* contained into the *Always* object (which, we recall, is in turn contained into the *AlwaysConditionalStatement*). If the *Statement* into the *StatementHelperContainer* is a *IfThenConditionalStatement* the function returns the *Criterion* contained into the *Then*

object (which, we recall, is in turn contained into the *IfThenConditionalStatement*). In all the other cases, the function return 'null'.

11.2.3 The getCriterionWhereParamIsInvolved method

The aim of this private method is to return the *Criterion* related to a given *SingleParameter* (i.e. the *Criterion* involving the specific *SingleParameter*). It takes indeed as argument a *SingleParameter* s and an *AbstractCriterion* $\mathcal{A}$ and returns an *AbstractCriterion*. But things are not so trivial as their could appear, since the *Criterion* involving the *SingleParameter* could be deeply located into the tree structure of the *AbstractCriterion* passed as argument. For this, this method has an recursive bahaviour. The following actions are performed:

- If $\mathcal{A}$ is 'null' the method returns 'null',
- If the *Expression* contained into $\mathcal{A}$ is an *AtomicParameterExpression* then
  - If s is the parameter referenced into the *AtomicParameterExpression* contained in $\mathcal{A}$, we returns $\mathcal{A}$.
  - if s is not the parameter referenced into the *AtomicParameterExpression* contained in $\mathcal{A}$ and this last contains a *LogicalConnector*, we call the method *getCriterionWhereParamIsInvolved* by passing as arguments the parameter s and the *AbstractCriterion* contained into the *LogicalConnector* of $\mathcal{A}$.
- If the *Expression* contained into $\mathcal{A}$ is not an *AtomicParameterExpression*, then we call the method *getCriterionWhereParamIsInvolved* by passing as arguments the parameter s and the *AbstractCriterion* contained into the *LogicalConnector* of $\mathcal{A}$.

Remark: The test of the second item is very important, because we handle only *AtomicParameterExpression*. It is straightforward to understand the reasons of this conditions while building a dynamic GUI: one could represent on the GUI the condition $p_1 \in \{2; 3; 4; 5; 8\}$, but what would mean (and how to represent with standard graphic components) a condition such $(p_1 + p_2) \in \{2; 3; 4; 5; 8\}$? (This last condition has a PDL sense and could be verified by our framework, but not represented in the dynamic GUI).

11.2.4 The test method

This private method takes as arguments a *StatementHelperContainer* $\mathcal{S}$, a *SingleParameter* s and returns a *PDLBaseParamPanel*. In this method the following actions are performed:

- If the *ConditionalStatement* contained into $\mathcal{S}$ is switched and involves the parameter s (these two aspects are respectively verified by calling the *getCriterionWhereParamIsInvolved* and *getCriterionFromStatement* methods), let $\mathcal{A}$ be the *AbstractCriterion* involving s and $\mathcal{C}$ the *AbstractCondition* contained into $\mathcal{A}$,

- If $\mathcal{C}$ is a *DefaultValue* instance, then we put into the internal representation system (the Mapper, cf. par 12) the vector $\vec{g}$ of *GeneralParameter* built by interpreting (cf. section. 6) the *Expression* contained into the *DefaultValue* object which, we recall, explicits this vector default value for the vector parameter s. The function calls the *buildBasicPanel* method by passing s as argument and returns the value it receives from *buildBasicPanel* .
- If $\mathcal{C}$ is a *BelongToSet* instance, then we build a List of vectors $\vec{g}_j$, where every element j is a vector of *GeneralParameter* and comes from the interpretation (cf. section. 6) of the j^{th} *Expression* contained into the *BelongToSet* object. The function return a *PDLChoseBoxParamPanel* build by passing as argument to the constructor the parameter s and the List of vectors $\vec{g}_j$.
- If $\mathcal{C}$ is a *IsNull* instance, we create a new *PDLTextParamPanel* by passing as argument to the constructor the parameter s, we set the visibility of this *PDLTextParamPanel* graphic component to false, and return it.

- If the method reaches this point, it returns "null'.

11.2.5 The paramBuilder method

This public class takes as argument a *GroupHandlerHelper* $\mathcal{G}$ and a *SingleParameter* s. It returns a *PDLBaseParamPanel*. In this method the following actions are performed:

- if the List of *StatementHelperContainer* contained into $\mathcal{G}$ is 'null' we call *buildBasicPanel* by passing s as parameter and return its returned value.
- if the List of *StatementHelperContainer* contained into $\mathcal{G}$ is not 'null', we loop on every *StatementHelperContainer* $\mathcal{S}_i$ element and
  - we call the method *test* by passing as argument the couple$(\mathcal{S}, s)$. If the returned panel is not null, we return it.
- if the method reaches this point, we call *buildBasicPanel* by passing s as parameter and return its returned value.

11.3 The PDLTextParamPanel class

This class extends *PDLBaseParamPanel* (cf.par 11.1.1). It is used for interacting with the values of a given parameter through an editable *JTextField*. This class has a *JtextField* attributes *textField*, which comes with usual getter and setter.

- The class constructor, which takes as argument an instance s of *SingleParameter* call the constructor of the super class by passing s as argument, initialize the attribute and call the inherited *initializeComponent* method.
- The overridden method *getComponent* returns the *textField* attribute.
- The overridden method *setComponentValue* sets the text contained into *textField* to the value returned by the private method *buildParamValue*.
- The private method *buildParamValue* build the List of *GeneralParameter* representing the values already provided for the parameter s. Then it return the String

returned by the inherited method *convertToStringProvidedValues* invoked passing the previously built List as parameter.
- The overridden method *getUserProvidedValue* returns the text contained into *textField* (which has previously been set by *setComponentValue* .
- The method *focusGained* does nothing.
- The method *focusLost* call the inherited method *verify*.

11.4 The PDLBooleanParamPanel class

This class extends *PDLBaseParamPanel* (cf.par 11.1.1). It is used for interacting with boolean parameter using an *JCheckBoxes*. This class has three attributes:

- *boxList* a List of *JCheckBox*, that will appears into the GUI.
- *checkPanel* of type *JPanel*, that will contains all the JCheckBoxes.
- *parameter* of type *SingleParameter*.

Moreover,

- the class constructor, which takes as argument an instance s of *SingleParameter* call the constructor of the super class by passing s as argument, initialize the attributes, set the *parameter* attribute equal to s and call the inherited *initializeComponent* method.
- The *getComponent* method is a usual getter on the attribute *checkPanel*.
- The private method *getPreEnteredValues* takes no argument and return a List of Booleans. This method first get (by calling the method *getuserProvidedValuesForParame* of the *Utilities* class, cf. par 12) the vector $\vec{g}$ of *GeneralParameter* representing the values already provided for the actual parameter s, which we recall is to type boolean. We return the vector of boolean $\vec{b}$ defined $\forall i,\ b_i = value(g_i)$.
- The overridden method *setComponentValue* try to get the dimension of the *parameter* attribute. If an error occurs during this phase, the dimension is set to 1. Then we will generate a number equal to dimension of *JCheckBox*. The set of the boolean values of this vector of *JCheckBox* is initialized with the vector returned by the call of *getPreEnteredValues* method and the actual instance of *PDLBooleanParamPanel* is added to every component of the *JCheckBox* vector as *ActionListener*. Finally we add every element of this *JCheckBox* vector to the attribute *boxList* and to the attribute *checkPanel*.
- The overridden *getUserProvidedValue* takes no argument and return a String. From the attribute *boxList* we build a vector of boolean values $\vec{b}$, where the component b_i is equal to 'true' if the i^{th} *JCheckBox* is selected and is equal to 'false' otherwise. The String returned by the method is '$b_1; b_2; ...; b_N$', where N is the size of vector $\vec{b}$.
- The *focusGained* method does nothing.
- The *focusLost* and *actionPerformed* calls the inherited *verify* method.

11.5 The PDLChoseBoxParamPanel class

This class extends the *PDLBaseParamPanel* (cf.par 11.1.1). It is used for interacting with parameter (appearing into *BelongToSet*) through a *JComboBox*.
This class has two attributes

- *comboBox* of type *JComboBox*. This JComboBox will be displayed in the GUI.
- *setOfValues*: a List of List of GeneralParameter. In other words this attribute is a set of vector $\{\vec{g}_i\}$, where $i = 1, ..., N$, this last being the number of values contained into the set. Every element $\vec{g}_i$ is indeed the result of the interpretation of the i^{th} *Expression* contained into a given *BelongToSet* object.

Moreover,

- the class constructor takes as arguments an instance of *SingleParameter* s and a List of List of *GeneralParameter* $\{\vec{g}_i\}$. It sets the two attribute with the values received as parameters and calls the inherited method *initializeComponent*.
- The overridden method *getComponent* is an usual getter on the attribute *comboBox*.
- In the overridden method *setComponentValue* the following actions are performed:
  - First we try to get the List of *GeneralParameter* corresponding to an already selected value of the set and convert it into a String by calling the inherited method *convertToStringProvidedValues*. This String will be our pre-selected value label.
  - If no particular value of the set has been pre-selected, we convert into a String the List of *GeneralParameter* corresponding to the first element of the *setOfValues*. This String will be our pre-selected value label.
  - We loop on the element of *setOfValues* and for every element e_j
    - We convert e_j into a String by calling the inherited method *convertToStringProvidedValues*.
    - We add this last String to the items of the *comboBox*.
    - If this String is equal to the pre-selected value label, then we store the index i.
  - We set the selected index of the attribute *comboBox* to the index value i previously retained.
- In the *focusGained* method no action is performed.
- In the *focusLost* method, the inherited *verify* method is called.

12 The pdl.interpreter.utilities

This package contains three utilities classes used transversally through the code.

- The *ConstantUtils* class is only used for building the instances of PDL descriptions using JAX-B: it is not necessary to explore it for understudying how works the PDL framework starting from an existing description.

- The *UserMapper* is the class used for the local internal representation of the parameter whose values are already provided by the user. It contains the attribute *map* of type `HashMap<String, List<GeneralParameter>>` . The key of this map is the String name of the parameter and the value is the List of *GeneralParameter* constituting the values provided by the user for this parameter.
  Moreover,
  - the class constructor takes no argument and initialize the unique attribute by defining a new *HashMap*.
  - The method *getuserProvidedValuesForParameter* takes as argument a *SingleParameter* instance *s* and returns the List of *GeneralParameter* associated to that parameter (of course the *map* is used for this operation).
  - The method *getUserProvidedValuesForParam* takes as argument the String name of a parameter and returns the List of *GeneralParameter* associated to that parameter (again the *map* mechanism is used).
  - The *makeListFromSingle* method is a facility method used for easily handling scalar parameters. This method automatically create a List of *GeneralParameter* containing only a single element from the *GeneralParameter* it receive as argument.
- The *Utilities* class is implemented using the singleton pattern. It contains to attributes *service* of PDL *Service* type and *mapper* of type *UserMapper*. These two attributes come with usual getters and setters. Moreover
  - the method *getParameterFromReference* takes a *ParameterReference* (i.e. the reference on a parameter) and returns the referenced *SingleParameter* instance.
  - The method *getuserProvidedValuesForParameter* takes as argument an instance of *SingleParameter s* and returns the result of the invocation of the method *getuserProvidedValuesForParameter* owned by the attribute *mapper*.
  - The method *getuserProvidedValuesForParame* takes as argument the String name of a *SingleParameter* and returns the result of the invocation of the method *getUserProvidedValuesForParam* owned by the attribute *mapper*.
  - The method *getParameterForTheGroup* takes as argument an instance of *ParameterGroup* and returns the List of all the *SingleParameer* contained into this group.
  - The void method *callService* (which takes no argument) is used for passing to the remote service the values entered by the user through the dynamic GUI. The following actions are performed:
    * An instance of *IserviceCaller* (cf. par. 13) is created by calling the *buildCaller* method of the *ServiceCallerFactory* and passing the *service* attribute as argument. The method *callService* owned by the *IserviceCaller* instance is then called.

13 The pdl.serviceCaller package

In this package we define the interface classes for communicating with the remote processing engine and services (which are exposed by providing a PDL description). This package contains the interface *IserviceCaller* which describe the method void *callService* which takes no arguments.
In their method *callService* all the classes implementing the interface has to pass the set of value entered by the used (and stored into the *mapper* attribute of the *Utilities* class (cf. par. 12) to the remote service.
With the actual design one has to implement a class for every service and integrate its construction/particular case in the *ServiceCallerFactory* class[4]. At the moment we have three classes implementing *IserviceCaller*, one for every service we exposed (this service are exposed via usual Java servlet, so we invoke the service pointing to the ad hoc URL, built using the user provided values).

The *ServiceCallerFactory* implements the singleton patterns. The method *buildCaller* takes as argument an instance of *PDL* service and, according to the service name return, if it known, the *ad hoc* class implementing *IserviceCaller*.

14 The gui package

Every class in this package corresponds to a different (macro) part of the GUI, as it is shown if figure 1.
Since all the complexity deriving from PDL is handled in the previously exposed packages, the code contained into this one is less PDL-tricky and is really an usual Java Swing interface, coming with its set of (heavy!) listeners and actions. For this reason, the presentation of this package is less detailed than the previous ones.
The classes composing this package are:

- *PDLTree*, which is used for presenting into the GUI the hierarchy of groups composing the input of a given service using a tree structure (a JTree object). The label of every node is the name of the group (constituting the node itself) and a change in the selected node will force the refresh of the *GroupPanel*.
- *PDLSummaryPanel*, which is used for presenting a quick summary of the service. In this panel are indeed displayed the groups to complete (i.e. all the the groups where at least a parameter is not provided by user), the groups in error (i.e. all the groups where at least a constraint is not satisfied) and the valid groups (i.e. all the groups where all the parameter are provided by user and all the constraints are satisfied). When all the groups composing the services belongs to this last category, in this panel is shown a button for launching the computation on the remote system.

[4]This part should quickly evolve and I think I'm going to introduce a dispatcher which will works for all the service calls

- *PDLStatementPanel* represent informations about a given statement and inform user about the statement validity, the switched/off statement character and display the String comment of the statement.
  A set of *PDLStatementPanel* is contained into *GroupPanel*: we have a *PDLStatementPanel* for every statement attached to the current group (i.e. the group whose node is selected into the *PDLTree* and which is current displayed in the *GroupPanel*.
- *GroupPanel* which is used for presenting into the GUI all the parameters attached to the current group and the *PDLStatementPanel* of all the statement attached to the current group.
  For every *SingleParameter* in the current group this panel build and display a *PDLBaseParamPanel* (cf. par. 11.1.1) by invoking the method *buildBasicPanel* of *PDLParamPanelFactory* class (cf. par. 11.2).
  This panel also display the set of *PDLStatementPanel* associated to all the statement attached to the current group.
- *IntelligentGUI* which coordinates and build all the previous graphic elements and handle their internal communications based on the action and listener mechanism.

15 The test package

The class *BaseExample* and the classing inheriting from it (BroadeningExample, Example02, ExampleInteropPune02, OpacityExample, PDRExample, PDRService, Principale) are used only for generating using the Jax-B framework the instances of the PDL descriptions that will be interpreted by the dynamic GUI. Indeed they are *a priori* with respect to the mechanisms we have exposed in this document.

Starting from an instance of PDL service description (contained into the file named PDL-Description.xml), the *buildService* method of the class *GUITest* (which takes no argument) uses the Jax-B framework and return a*Service* object containing the Java object structure corresponding to the transposition of the XML into Java objects (cf. par. 3). This method is called by the *main* function of *GUITest* which, after this invocation, will initialize the instance of UserMapper that will be used in all the application and will build an instance of *IntelligentGUI* that will display the graphical interface.

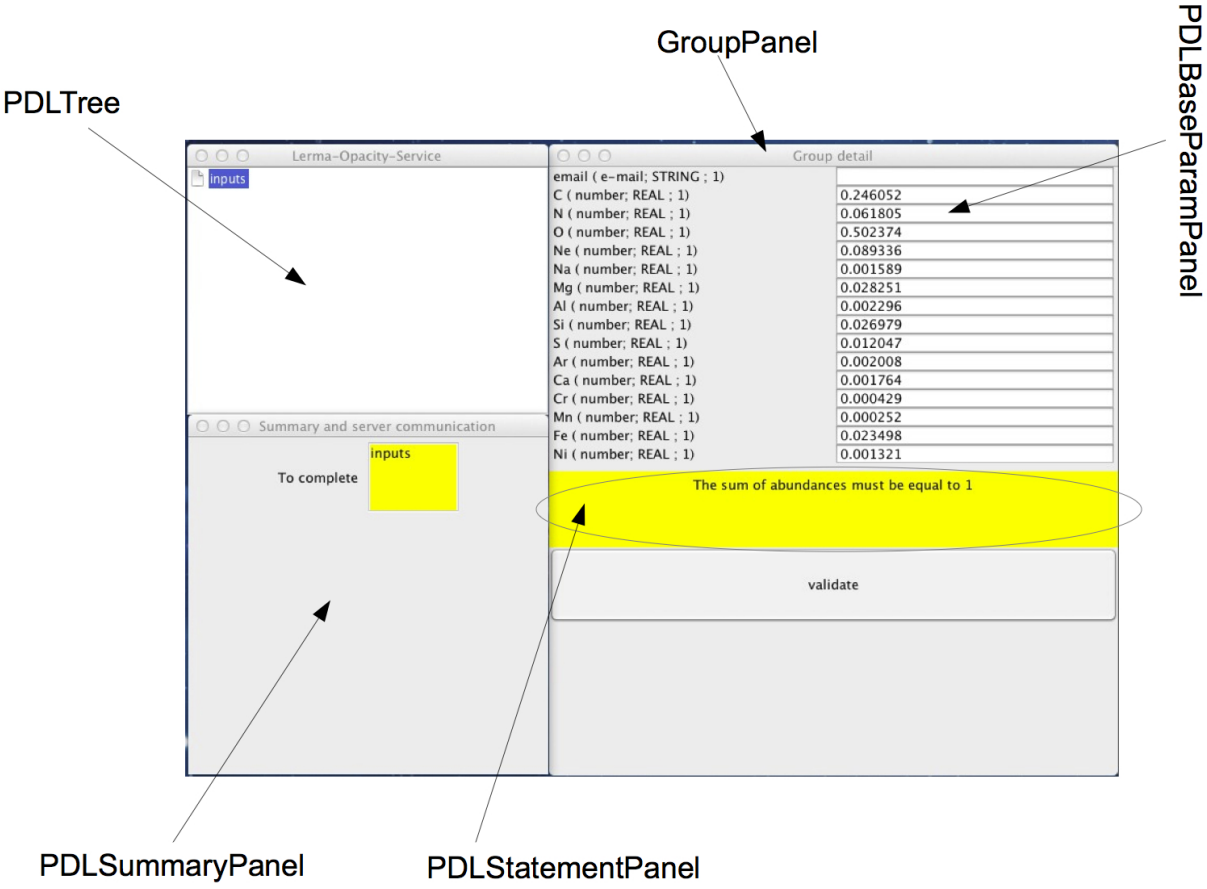

Figure 1: PDL GUI with highlights of the macro graphical components corresponding to the classes of the present package