Implementing a binary validation rule

Implementing Custom Validation Behaviour

If you want to control when the validation rule has to be executed and you want to control what must be validated and how, then you have to create a Java class and implement the interface *com.nomagic.magicdraw.validation.ValidationRule*. The class must have the public *no-arg* constructor. A value of the implementation property of the validation rule must be a qualified name of the implemented class.

This case is not recommended for validating model elements because the validation rule provider must implement not only validating of model elements but also other details that are provided by the modeling tool validation engine. In this case, the validation rule provider must correctly implement validating of model elements that are from the validation scope defined in project options, should take care that implementation would respect the project property *Exclude element from used read-only project*, and is responsible for implementing when the validation has to be executed.

Implementing Rule to Validate Model Elements

If you want to concentrate on implementation of validating model elements and delegate a task of detecting model element changes and project property changes to the validation engine then you have to create a Java class and implement *com.nomagic.magicdraw.validation.ElementValidationRuleImpl* and the interface. The class must have the public *no-arg* constructor. The value of the implementation property of the validation rule must be a qualified name of the implemented class. Note that If this rule is used in active validation, then it will be repeatedly executed after every single change in the model.

This example shows how to add custom solver action and set dynamic error message to validated element:

```
public Set<Annotation> run(Project project, Constraint constraint,
Collection<? extends Element> elements)
{
   Set<Annotation> result = new HashSet<>();

   for (Element element : elements)
   {
      if (!isValid(element))
      {
         List<NMAction> actions = new ArrayList<>();
         actions.add(new MyAction("MY_ACTION_ID", "Test", null));
         result.add(new Annotation(element, constraint, getErrorText
(element), actions));
      }
   }

   return result;
}

private static String getErrorText(Element element)
{
   return "Name for [" + element.getID() + "] should be longer than " +
MIN_NAME_LENGTH + " symbols.";
}

private static boolean isValid(Element element)
{
   NamedElement el = (NamedElement) element;
   return el.getName().length() > MIN_NAME_LENGTH;
}
```

Optimizing Rule Execution for Active Validation

If the validation rule is designed for the active validation then the validation performance becomes very important. In this case we want that the validation rule would be executed only when it is actually required, so the implementation of the validation rule must somehow inform the active validation engine which model element property changes are important to the validation rule. In order to create such validation rule you have implement *ElementValidationRuleImpl* and *com.nomagic.uml2.ext.jmi.smartlistener.SmartListenerConfigProvider* interfaces. The class must have the public *no-arg* constructor. The value of the implementation property of the validation rule must be a qualified name of the implemented class.

The *SmartListenerConfigProvider* interface defines a single method:

Related pages

- Creating validation rules

```
    Map<Class<? extends Element>, Collection<SmartListenerConfig>>
getListenerConfigurations()
```

The implementation of this method should return a map of classes derived from the *com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Element* class to collection(s) of *com.nomagic.uml2.ext.jmi.smartlistener.SmartListenerConfig* objects. Another way would be to extend the *com.nomagic.magicdraw.validation.DefaultValidationRuleImpl* class and override the following method:

```
    Map<Class<? extends Element>, Collection<SmartListenerConfig>>
getListenerConfigurations()
```


In the Constraint Specification dialog box, the Constrained Element property should be specified. A constrained element is the element for which the validation rule is created.

Example of listener configuration for the validation rule that is interested in name changes of Class and Interface model elements:

```
    public Map<Class<? extends Element>, Collection<SmartListenerConfig>>
getListenerConfigurations()
    {
        Map<Class<? extends Element>, Collection<SmartListenerConfig>>
configMap =
                                        new HashMap<Class<? extends
Element>, Collection<SmartListenerConfig>>();
        Collection<SmartListenerConfig> configs = new
ArrayList<SmartListenerConfig>();
        configs.add(SmartListenerConfig.NAME_CONFIG);
        configMap.put(com.nomagic.uml2.ext.magicdraw.classes.mdkernel.
Class.class, configs);
        configMap.put(com.nomagic.uml2.ext.magicdraw.classes.mdinterfaces.
Interface.class, configs);
        return configMap;
    }
```


For the given example, In the specification dialog of the **Constraint** element the Constrained Element property should be set to Class and Interface metaclasses from the UML Standard Profile.

Example of listener configuration for the validation rule that is interested in Activity parameters and Activity parameter node changes

```
    public Map<Class<? extends Element>, Collection<SmartListenerConfig>>
getListenerConfigurations()
    {
        Map<Class<? extends Element>, Collection<SmartListenerConfig>>
configMap =
                                        new HashMap<Class<? extends
Element>, Collection<SmartListenerConfig>>();
        Collection<SmartListenerConfig> configs = new
ArrayList<SmartListenerConfig>();
        SmartListenerConfig parameterConfig = new SmartListenerConfig();
        Collection<String> parameterPropertiesList = new
ArrayList<String>();
        parameterPropertiesList.add(PropertyNames.DIRECTION);
        parameterPropertiesList.add(PropertyNames.TYPE);
        parameterPropertiesList.add(PropertyNames.OWNED_TYPE);
        parameterConfig.listenToNested(PropertyNames.OWNED_PARAMETER).
listenTo(parameterPr opertiesList);
        SmartListenerConfig cfg = new SmartListenerConfig();
        Collection<String> argumentCftList = new ArrayList<String>();
        argumentCftList.add(PropertyNames.PARAMETER);
        argumentCftList.add(PropertyNames.TYPE);
        argumentCftList.add(PropertyNames.OWNED_TYPE);
        cfg.listenTo(argumentCftList);
        SmartListenerConfig argumentConfig = new SmartListenerConfig();
        argumentConfig.listenTo(PropertyNames.NODE, cfg);
        configs.add(parameterConfig);
        configs.add(argumentConfig); configMap.put(Activity.class,
configs); return configMap;
    }
```

See the *JavaConstantNameValidationRuleImpl.java* example in *<program installation directory>\openapi\examples\validation*.

Case D

If you want to create a validation rule that will not be used in the active validation and you do not need to provide solving actions then you can create a Java class that contains the static method with the signature:

```
     public static Boolean <method_name>(<Element_type> param)
```

and specify *<qualifiedClassName>.<method_name>* as a body value of the validation rule's specification.

The referred java method must be a static method and it must take exactly one parameter. Several validating methods can be placed into the one Java class or use the one class per validating method – this is irrelevant.

The type of the parameter MUST match the type of the constrained element of the validation rule. For validation rules on metaclasses, use the appropriate class from the *com.nomagic.uml2.ext.magicdraw.** package. For example, to write the rule for the metaclass *DataType*, use *com.nomagic.uml2.ext.magicdraw.classes.mdkernel.DataType* as a type of the parameter.

For the validation rules on stereotypes, use the same class as you would use when specifying the validation rule for a metaclass of that stereotype. For validation rules on the classifiers of the model, use *com.nomagic.uml2.ext.magicdraw.classes.mdkernel.InstanceSpecification* as the type of the parameter.

The Java code example of a trivial validation rule, which always returns *true* (that is, all elements are valid):

```
package com.nomagic.magicdraw.validation;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.DataType;
public class TestRule
{
    public static Boolean testMeta(DataType exp)
    {
        return Boolean.TRUE;
    }
}
```

Compile such code into the Java bytecode, locate it where the modeling tool can load it (a classpath, plugin), and then you can use it for the validation:

1. Create the validation rule in our model.
2. Select the *DataType* metaclass as the constrained element of the rule.
3. Select the *Binary* language for the expression of the rule.
4. Specify *com.nomagic.magicdraw.validation.TestRule.testMeta* as the body of the expression of the rule.

Run the validation suite with the validation rule included. The method will be invoked for each datatype model element that is found in the scope of the validation. For each element, where this method returns *false*, an entry will be placed in the validation results table.

Implementing Rule to Validate Presentation Elements (Symbols)

In case you need a validation rule that would check and annotate individual presentation elements (symbols) in diagrams, then a Java class extending *com.nomagic.magicdraw.validation.DiagramValidator* should be implemented. This rule can be used for example to annotate symbols with overlapping bounds, or to detect multiple representations of the same model element in the same diagram in case only a single representation is desired.

Instead of receiving model element to validate as a parameter, use *DiagramValidator.getDiagrams()* method to get diagrams that are in the currently validated scope. To annotate individual symbols, create *com.nomagic.magicdraw.annotation.Annotation* object with the target being the *com.nomagic.magicdraw.uml.symbols.PresentationElement* which does not pass your validation conditions.

Extend a subclass *com.nomagic.magicdraw.validation.OpenedDiagramValidator* instead for your rule to always restrict the validated diagrams to the currently opened ones only.

Also see: Presentation elements (symbols)