Extending Model Analysis in Rational Software Architect

Part 2: Extending the UML Model Metrics feature

Author: Celso Gonzalez (cgonzale@ca.ibm.com), Model Analysis Lead, Rational Software Architect

Level: Intermediate/Advanced

Model analysis is a capability in Rational Software Architect that simplifies model automation and improves design quality. You can use model analysis to access capabilities such as automated model review, model metrics, and structural analysis that you can use daily in the development process.

Part 1 of this information introduces the model analysis framework and shows you how to create a new analysis review rule. Part 2 explores how to create new metrics rules, and Part 3 explores more advanced capabilities of the model analysis API, including the help, rule template, and quick fix capabilities.

1. Introduction

Part 1 introduced the model analysis framework and how to create a review rule. In this part, you write UML metrics rules to extend the current rule set for model analysis metrics.

The following information provides an overview of these items:
- The main parts of the metrics class rule
- The process to create your own metrics rules by implementing a simple rule
- A specific extension point called Measurement that you use to gather measurements

2. UML Model Metrics rules

The main extension points for metrics rules are the same as for review rules; therefore, the first half of the rule definition is an Eclipse extension. The second half is a Java class that uses the rule API and parts of the UML 2 API to query the content of a class and produce results for any items that match the rule criteria. The rule class typically
contains a single method. Because rules usually extend a default rule class that is provided by the model analysis framework, most of the required functionality is implemented already.

The following code shows a template for a metric rule:

```java
public class NumberOfVisibilityAttributesMetric extends CollectableMetric {
    // Identifies the elements you want to check.
    protected boolean canCheck(AnalysisHistory history, EObject object) {
    }

    // Reads the feature associated with object.
    protected Object collectData(AnalysisHistory history, Object object) {
    }

    // Measures the data collected by collectData()
    protected double measureData(AnalysisHistory history, Object data) {
    }

    // Defines how results should be combined.
    protected int getResultsCombinationMethod() {
    }

    // Defines objects that can be measured.
    protected boolean canMeasure(AnalysisHistory history, EObject object) {
        return UMLPackage.Literals.NAMESPACE.isInstance(object);
    }
}
```

The rule class extends `com.ibm.xtools.analysis.uml.metrics.internal.CollectableMetric` from the model analysis framework and, because this class is abstract, the derived rule must implement the following methods:

- **canCheck**: Used to decide if a given element is applicable to the metric.
- **collectData**: Used to gather the elements that match the rule to measure.
- **measureData**: Implements the measure algorithm based on the data collected by collectData.
- **getResultsCombinationMethod**: Defines how metrics must be aggregated when navigating the model hierarchy.
- **canMeasure**: Used to define the elements that you want to see. It is a superset of the elements filtered in the canCheck method and must include its possible parents. For example, if you want to calculate a metric about classes, the canMeasure method must also include packages and models. Typically, you use the `NAMESPACE` type because it is a superclass of the main UML elements.

### 3. Writing new rules
After you understand the different parts of a metrics rule, you can write a complete metrics rule.

### 3.1. Create a plug-in

Before you create a rule, you must create a plug-in.

1. In Eclipse, open the New Project wizard.
2. On the Plug-in Project page, create a new plug-in project in your workspace named `com.ibm.xtools.analysis.uml.metrics.myexample` as shown in Figure 1.

![Figure 1](image)

3. On the Content page, ensure that the **This plug-in will make contributions to the UI** check box is selected, as in Figure 2.
4. To create the plug-in, click **Finish**.
5. To successfully build rules, you must add dependencies to the plug-in. Open the plug-in manifest files and click the **Dependencies** tab.
6. Add plug-ins as shown in Figure 3.
The `com.ibm.xtexts.analysis.model` plug-in contains the basic model analysis framework API and extension points.

The `com.ibm.xtexts.analysis.uml.metrics` plug-in contains the API for writing UML metrics rules.

The `org.eclipse.uml2.uml` plug-in contains the UML API, including the parser.

Typically, you need all three of these plug-in dependencies when you create new metrics rules.

### 3.2. Create a rule class

In object-oriented languages, it is common practice for attributes to be private. However, as an exception, attributes occasionally require other visibilities, such as protected, package, or public. To ensure that developers do not create too many attributes with other visibilities, you can create a rule to measure how many non-private attributes exist in models. To make the rule useful, you can measure class by class, but also aggregate the information by package and by model. With this rule, you can view the model to determine whether too many non-private attributes exist at the model level, and then drill down to find the responsible classes.
The rule must perform these actions:

1. The rule must generate a list of the classes in the specified models. The rule only has to examine models that have at least one attribute.

2. After the list is generated, the rule must examine all the attributes and count those that do not have a public visibility.

3. As the rule proceeds through the model hierarchy, it must add the results so that when it reaches a package node, it provides an aggregate number of the non-private attributes in the package and its subpackages.

To create the rule, you complete these steps:

1. In the `metrics.myexample` plug-in, create a new class in the package named `com.ibm.xtools.analysis.uml.metrics.myexample`.

2. Name the class `NumberOfVisibilityAttributesMetric`, and ensure that the rule class extends `com.ibm.xtools.analysis.uml.metrics.internal.CollectableMetric`.

3. Add the previously discussed methods to the class, so that the class looks like the following example:

   ```java
   import org.eclipse.emf.ecore.EObject;
   import com.ibm.rsaz.analysis.core.history.AnalysisHistory;
   import com.ibm.xtools.analysis.uml.metrics.internal.CollectableMetric;
   
   public class NumberOfVisibilityAttributesMetric extends CollectableMetric {
   
   // Identifies the elements you want to check.
   protected boolean canCheck(AnalysisHistory history, EObject object) {
   
   // Reads the feature associated with object.
   protected Object collectData(AnalysisHistory history, Object object) {
   
   // Measures the data collected by collectData()
   protected double measureData(AnalysisHistory history, Object data) {
   
   // defines how results should be combined
   protected int getResultsCombinationMethod() {
   
   // defines objects that can be measured
   protected boolean canMeasure(AnalysisHistory history, EObject object) {
       return UMLPackage.Literals.NAMESPACE.isInstance(object);
   }
   }
   }
   }
   }
   }
   ```

4. To collect a list of classes from the models that have at least one attribute, add the following lines of code to the `canCheck` method:
```java
boolean status = false;
if (object instanceof Class ){
    status = (((Class)object).getAttributes() != null);
}
return status;
```

5. After you filter the elements of interest, add the logic to retrieve the non-private attributes by adding the following code to the collectData method:

```java
if(object instanceof EObject && canCheck(history, ((EObject)object))) {
    Set<Property> styleAttr = new HashSet<Property>();
    Class clazz = (Class)object;
    for (Iterator<Property> i = clazz.getAttributes().iterator(); i.hasNext();)
    {
        Property attr = (Property) i.next();
        if ((param != null) &&
            (VisibilityKind.PRIVATE != attr.getVisibility().getValue())){
            styleAttr.add(attr);
        }
    }
    return styleAttr;
} else {
    return NO_DATA;
}
```

To ensure that the element matches the defined criteria, the code calls the canCheck method. Then, the code retrieves the private attributes and adds them to the property set. If the object is a class with attributes, the set is returned; otherwise, a NO_DATA constant is returned, which is an empty collection. The data is now collected.

6. To measure the data, add the following code to the measureData method so that 0 is returned for a result of NO_DATA, the size is returned for a collection, and 1 is returned for all other results:

```java
if(data == NO_DATA){
    return 0;
} else if (data instanceof Collection){
    return(((Collection)data).size());
} else {
    return 1;
}
```

7. To aggregate the results, add one of the following three SimpleRollupMetric constants to the getResultsCombinationMethod method:
   - RESULTS_SUM
   - RESULTS_AVERAGE
   - RESULTS_NOROLLUP

To understand the constants, assume that you have a package with three classes and that each class has two non-private attributes.

If you choose RESULTS_SUM, the metric for each class is 2 and the metric for the package is 6 (2+2+2). If you choose RESULTS_AVERAGE, the metric for the classes is
the same, but the package metric is 2 \((2+2+2)/3\). With RESULTS_NOROLLUP, the classes have metrics, but the metric for the package is 0.

In this example, you want to use RESULTS_SUM, so add the following code to the getResultsCombinationMethod method:

```java
return SimpleRollupMetric.RESULTS_SUM;
```

The following code shows the completed rule:

```java
package com.ibm.xtools.analysis.uml.metrics.example.internal.rules.size;

import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.uml2.uml.Class;
import org.eclipse.uml2.uml.Property;
import org.eclipse.uml2.uml.UMLPackage;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;
import com.ibm.xtools.analysis.uml.metrics.internal.CollectableMetric;
import com.ibm.xtools.analysis.uml.metrics.internal.SimpleRollupMetric;

public class NumberOfVisibilityAttributesMetric extends CollectableMetric {

    // Identifies the elements you want to check.
    protected boolean canCheck(AnalysisHistory history, EObject object) {
        boolean status = false;
        if (object instanceof Class) {
            status = ((Class) object).getAttributes() != null;
        }
        return status;
    }

    // Reads the feature associated with object.
    protected Object collectData(AnalysisHistory history, Object object) {
        if (object instanceof EObject && canCheck(history, ((EObject) object))) {
            Set<Property> styleAttr = new HashSet<Property>;;
            Class clazz = (Class) object;
            for (Iterator<Property> i = clazz.getAttributes().iterator();
                 i.hasNext();) {
                Property attr = (Property) i.next();
                if ((param != null) && 
                    (getParamVisibility(param) != attr.getVisibility().getValue())) {
                    styleAttr.add(attr);
                }
            }
            return styleAttr;
        } else 
            return NO_DATA;
    }

    // Measures the data collected by collectData()
    protected double measureData(AnalysisHistory history, Object data) {
        if (data == NO_DATA) 
            return 0;
        else if (data instanceof Collection) 
            return (((Collection) data).size());
        else
```
return 1;
}

// defines how results should be combined
protected int getResultsCombinationMethod() {
    return SimpleRollupMetric.RESULTS_SUM;
}

// defines objects that can be measured
protected boolean canMeasure(AnalysisHistory history, EObject object) {
    return UMLPackage.Literals.NAMESPACE.isInstance(object);
}

private int getParamVisibility(AnalysisParameter param) {
    return Integer.valueOf(param.getValue()).intValue();
}

3.3. Create extension points

After you complete the rule class, you must describe it through the provided extension points in the model analysis framework so that the review engine can discover it. In this example, you associate the rule with the existing Size category.

To associate the rule with a category, complete these steps:
1. On the Extensions tab, right-click the rule extension and add a new analysisRule rule.
2. Populate the rule details as shown in Figure 4.

Figure 4
Extension Element Details
Set the properties of "analysisRule". Required fields are denoted by "*".

| id*:* | com.ibm.xtext.grammar.uml.metrics.example.rule.NumberVisibilityAttributes |
| label*:* | Number of non private attributes |
| category*:* | com.ibm.xtext.grammar.uml.metrics.category.size |
| icon: | |
| class: | com.ibm.xtext.grammar.uml.metrics.example.internal.rules.size.NumberOfVisibilityAttributesMetric |
| viewer: | com.ibm.xtext.grammar.uml.metrics.viewer |
| severity: | 0 |
| help: | |
| quickFixIcon: | |
| quickfixAll: | |

3. Enter a label, which is displayed in the user interface for the rule
The class field contains the qualified name of the rule class that you created earlier. The category field contains the identifier string for the Example category. After you provide the details, the plugin.xml file contains the following text:

```xml
<extension
    point="com.ibm.rsaz.analysis.core.analysisRule">
    <analysisRule
        category="com.ibm.xtools.analysis.uml.metrics.category.size"
        class="com.ibm.xtools.analysis.uml.metrics.example.internal.rules.size.NumberOfVisibilityAttributesMetric"
        id="com.ibm.xtools.analysis.uml.metrics.example.rule.NumberOfVisibilityAttributes"
        label="Number of non private attributes"
        severity="0"
        viewer="com.ibm.xtools.analysis.uml.metrics.viewer">
    </analysisRule>
</extension>
```

4. To make the metric more usable, you can define a lower and upper bound to more easily identify problems. To define the boundaries, you add two analysis parameters to the rule extension.

To add a new parameter, use the Eclipse extension editor to add a new analysisParameter parameter as shown in Figure 5.

**Figure 5**

![FIGURE 5](image)

Parameters are name/value pairs with some additional information. The label field contains the text that users see when they view the parameters field in the user interface.

The type field determines the data type for the value and is used to provide the correct field editor and field validation. Valid values for this field are string or integer.

The style field controls the type of UI control for the parameter. For example, you can represent data as standard text, a combination box, or a check box.

In this example, you only need standard text so that users can enter the lower and upper bound. If the value of the metric is not between the lower and upper bounds, the metric is displayed in red, which makes it easy to locate. Figure 6 shows an example of the analysis parameter for the lower bound.
You also need to add a similar parameter for the upper bound. The following code shows the rule with the parameters:

```xml
<extension
    point="com.ibm.rsaz.analysis.core.analysisRule">
  <analysisRule
    category="com.ibm.xtools.analysis.uml.metrics.category.size"
    class="com.ibm.xtools.analysis.uml.metrics.example.internal.rules.size.NumberOfVisibilityAttributesMetric"
    id="com.ibm.xtools.analysis.uml.metrics.example.rule.NumberOfVisibilityAttributes"
    label="%metrics.example.rule.NumberOfVisibilityAttributes"
    severity="0"
    viewer="com.ibm.xtools.analysis.uml.metrics.viewer">
    <analysisParameter
      label="The lowest allowed value"
      name="LOWER BOUND"
      type="string"
      value=""/>
    <analysisParameter
      label="The highest allowed value"
      name="UPPER BOUND"
      type="string"
      value=""/>
  </analysisRule>
</extension>
```

### 3.4. Run the rule

To run the rule that you created, complete these steps:

1. From the Eclipse “Run” menu select the “Run…” option and create a new “Eclipse Application”.
2. In the configuration pane, click Run to invoke a new runtime workbench to test the rule.
3. In the new workbench
a. Import a UML model that contains at least some classes that have public or protected attributes, so the metric will return some results.
b. From the menu, click Run > Analysis.
c. In the dialog box that is displayed, create a new analysis configuration and click the Rules tab.
d. In the hierarchy, expand Java Naming until the Example subcategory is visible, as shown in Figure 7.

e. Select the new rule and click Analyze to start the code review process. If the UML model in the workbench has classes or interfaces with a name that fits the rule, the Analysis Results view displays the results, as shown in Figure 8.
3.5. **Add flexibility to the metric**

With the new metrics rule, you can now measure the number of non-private attributes. However, if you are interested in metrics for non-protected or non-public attributes, you do not have to create a new rule. You can add a parameter to the rule that enables the user to select which visibility to measure.

To add flexibility to the metric, complete these steps:
1. Create a new parameter as you did before, but select the combo style as shown in Figure 9.

   **Figure 9**
   
   ![Extension Element Details](image)

   ```
   name*: VISIBILITY
   value*: 0
   label: Visibility:
   type: string
   style: combo
   ```

2. To provide the values for the combination box, add a new comboValue extension to the analysisParameter parameter for each visibility value (public, private, protected, and package) as shown in Figure 10. ComboValue extensions have a value field.
that contains the text that users see when they expand the combination box. The value is an integer based on the order of the ComboValue values: the first value is 0, the second value is 1, and so on. As in UML, the VisibilityKind value is also identified by an integer. The order that you use for the comboValue values is important for simplifying the rule. You should follow the order provided above so that the ComboValue value is the same as the corresponding VisibilityKind value.

Figure 10

3. After you add the custom data parameters to the rule extension, modify the code to retrieve the parameter value by replacing the content of the measureData method with the following text:

protected Object collectData(AnalysisHistory history, Object object) {
    if (object instanceof EObject && canCheck(history, ((EObject)object))) {
        Set<Property> styleAttr = new HashSet<Property>();
        Class clazz = (Class)object;
        for (Iterator<Property> i = clazz.getAttributes().iterator(); i.hasNext();) {
            Property attr = (Property) i.next();
            AnalysisParameter param = getParameter(VISIBILITY);
            if ((param != null) && ((Integer.valueOf(param.getValue()).intValue()!=
            attr.getVisibility().getValue())){
                styleAttr.add(attr);
            }
        }
        return styleAttr;
    } else
        return NO_DATA;
}

4. Add the following lines to the attribute section of the class:

    // Name of the AnalysisParameter used to define the considered visibility
    protected final static String VISIBILITY = "VISIBILITY";

Now, when you select the rule in the analysis configuration, you can select the visibility as shown in Figure 11.
4. Measurements

In the metrics rule that you created, the rule retrieves the data and performs the computation and calculation. However, you might want to share measures between rules or, more precisely, base different rules on the same base measure. Having multiple rules redo the computation and calculation can be time consuming, especially if you require large computations such as for inheritance. Measurements mitigate potential performance problems. A measurement performs the collection and computation and then stores the result in an in-memory object, so that rules that are interested in the measures that it contains can retrieve them instead of redoing the measurement operations.

In the following exercises, you implement a rule that measures the abstractness of elements based on the ratio of interfaces over other classes. For other extensions, a measurement consists of a Java class and an extension. In the following procedures, you write the measurement Java class, add the corresponding extension information, and then refer to the measurement from a model metrics rule.

4.1. Measurement class
Similar to rules, the measurement class extends an existing class so that it can leverage the model analysis framework.

The following code shows a template for a measurement:

```java
public class InterfaceMeasurement extends ModelAnalysisMeasurement {
    public static final String ID = ""

    protected boolean canMeasure(AnalysisHistory history, EObject object) {
        return UMLPackage.Literals.NAMESPACE.isInstance(object);
    }

    protected boolean canCheck(AnalysisHistory history, EObject object) {
    }

    protected Collection<NamedElement> getChildren(AnalysisHistory history, EObject eObject) {
    }

    protected Object getMeasurement(AnalysisHistory history, Object object, Collection childValues) {
        <Data> data = new <Data>();
    }

    class <Data> {
    }
}
```

The rule class extends `com.ibm.xtools.analysis.model.internal.metric.ModelAnalysisMeasurement` from the model analysis framework. The class must contain the following elements to make it a valid measurement:

- `<Data>`: This inner class persists the measurements in memory so that the metrics rules can access them.
- `ID`: Identifies the measurement.
- `canMeasure, canCheck`: These methods are the same as for the metrics rule.
- `getChildren`: Represents the elements to measure. The elements are accessed as children of the elements that comply with the canCheck method.
- `getMeasurement`: Performs the measurements and places the results in instances of the `<Data>` class.

### 4.2. Create the measurement class

To create the measurement class, complete these steps:

1. Provide the following ID for the measurement:
```
   public static final String ID =
        "com.ibm.xtools.analysis.uml.metrics.example.measurements.interface";
```
2. To measure the ratio between interfaces and other classes, which are typically stored in UML packages, the canCheck method must return UML packages and the getChildren must return the owned elements of the package. The following code shows the completed canCheck and getChildren methods:

```java
protected boolean canCheck(AnalysisHistory history, EObject object) {
    return UMLPackage.Literals.PACKAGE.isInstance(object);
}

protected Collection<NamedElement> getChildren(AnalysisHistory history,
                                              EObject eObject) {
    return ((Namespace)eObject).getOwnedMembers();
}
```

3. Define the name and content of the <Data> class, which is what persists in memory. In this example, persist the abstractness (the ratio between interfaces and other classes), the sum of interfaces, and the sum of other classes, so that the results are aggregated in parent packages. The following code shows the resulting AbstractnessData class:

```java
class AbstractnessData {
    double abstractness;
    double sumOfOthersSize;
    double sumOfInterfacesSize;
}
```

In the measurement class, the getMeasurement method checks whether the object is in scope of the measurement (it can pass the cancheck method), iterates through the owned elements, and separates the interfaces from the other elements.

Then, the following code retrieves the size of each set of elements and divides them to determine the abstractness:

```java
AbstractnessData abstractData = new AbstractnessData();

if (object instanceof EObject && canCheck(history, (EObject)object)) {
    Package pack = (Package) object;
    Set<Interface> abstracts = new HashSet<Interface>();
    Set<Class> others = new HashSet<Class>();

    for (Iterator<Type> i = pack.getOwnedTypes().iterator(); i.hasNext();) {
        Type type = i.next();
        if (UMLPackage.Literals.INTERFACE.isInstance(type)){
            abstracts.add((Interface)type);
        } else if (UMLPackage.Literals.CLASS.isInstance(type)){
            others.add((Class) type);
    }
}
abstractData.sumOfOthersSize += others.size();
abstractData.sumOfInterfacesSize += abstracts.size();
}
if (abstractData.sumOfInterfacesSize == 0) {
    abstractData.abstractness = 0;
} else {
```
abstractData.abstractness =
   (abstractData.sumOfInterfacesSize/(abstractData.sumOfOthersSize +
   abstractData.sumOfInterfacesSize));
}

At this point in the code, you can measure the abstractness of a package, but you cannot aggregate measurements, such as including subpackage results in the calculation of a package measurement. To get these values, you must iterate through the childValues collection to initialize the \texttt{sumOfOthersSize} and \texttt{sumOfInterfacesSize} attributes.

The following code shows the modified content of the \texttt{getMeasurement} method. The modifications are highlighted in bold font.

AbstractnessData abstractData = new AbstractnessData();

for (Iterator<AbstractnessData> i = childValues.iterator(); i.hasNext();) {
   AbstractnessData childData = i.next();
   abstractData.sumOfOthersSize += childData.sumOfOthersSize;
   abstractData.sumOfInterfacesSize += childData.sumOfInterfacesSize;
}

if (object instanceof EObject && canCheck(history, (EObject)object)) {
   Package pack = (Package) object;
   Set<Interface> abstracts = new HashSet<Interface>();
   Set<Class> others = new HashSet<Class>();

   for (Iterator<Type> i = pack.getOwnedTypes().iterator(); i.hasNext();) {
      Type type = i.next();
      if (UMLPackage.Literals.INTERFACE.isInstance(type)){
         abstracts.add((Interface)type);
      } else if (UMLPackage.Literals.CLASS.isInstance(type)){
         others.add((Class) type);
      }
   }

   abstractData.sumOfOthersSize += others.size();
   abstractData.sumOfInterfacesSize += abstracts.size();
}

if (abstractData.sumOfInterfacesSize == 0){
   abstractData.abstractness = 0;
} else {
   abstractData.abstractness =
   (abstractData.sumOfInterfacesSize/(abstractData.sumOfOthersSize +
   abstractData.sumOfInterfacesSize));
}

return abstractData;

4.3. **Create the measurement extension**

After you complete the measurement class, you must describe it through the provided extension points in the model analysis framework so that the review engine can discover it.
To create the measurement extension, complete these steps:

1. On the **Extensions** tab, add an extension named `com.ibm.xtools.analysis.model.measurement` and populate the details as shown in Figure 12.

   **Figure 12**
   
<table>
<thead>
<tr>
<th>Extension Element Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the properties of &quot;measurement&quot;. Required fields are denoted by &quot;*&quot;.</td>
</tr>
<tr>
<td>id*: com.ibm.xtools.analysis.uml.metrics.example.measurements.interface</td>
</tr>
<tr>
<td>class*: com.ibm.xtools.analysis.uml.metrics.example.internal.rules.abstractness.InterfaceMeasurement</td>
</tr>
</tbody>
</table>

2. Specify the ID of the measurement, ensuring that it matches the ID attribute on the measurement class and points to the class that implements the measurement.

   After you specify the details, the `plugin.xml` file contains the following text:

   ```xml
   <extension point="com.ibm.xtools.analysis.model.measurement">
     <measurement class="com.ibm.xtools.analysis.uml.metrics.example.internal.rules.abstractness.InterfaceMeasurement"
                   id="com.ibm.xtools.analysis.uml.metrics.example.measurements.interface">
   </measurement>
</extension>
```

### 4.4. Apply the measurement in a metrics rule

You can now use the measurement in a metric class. The following example shows a `getMeasurement` method that uses the `InterfaceMeasurement` measurement. To access the `AbstractnessData` class, the `getMeasurementManager` method retrieves the measurement by specifying its ID (InterfaceMeasurement.ID).

```java
protected Object getMeasurement(AnalysisHistory history, Object object,
                                 Collection childResults) {

    AbstractnessData data = (AbstractnessData) getMeasurementManager(history).
                             getMeasurementValue(object, InterfaceMeasurement.ID);
    double result = 0;

    if (data != null) {
        result = data.abstractness;
    }
    return result;
}
```
5. Conclusion

This information showed how to use the model analysis API to write new UML model metrics rules. After learning about metrics rule specification in the `plugin.xml` file, you wrote a class to create a basic metrics rule. You discovered how to use analysis parameters and measurements. You should now have enough knowledge to write your own metrics rules.

In Part 3 we will explore more advanced capabilities of the model analysis API, including the help, rule template, and quick fix capabilities.