Extending Model Analysis in Rational Software Architect

Part 1: Extending the UML Model Review 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

You can write UML review rules to extend the current set of model analysis rules. The model analysis framework includes several extension points and an API for adding new review and metrics rules. The framework also supplies a fully functional analysis provider to perform automated model analysis. The model analysis framework is an extension of the more generic Software Analyzer analysis framework. If you understand a certain amount of the framework, you can write any UML model review rule that you need.

The following information provides an overview of these items:

- The components that make up a rule
- The category and rule extension points
- The complete process to create review rules, demonstrated by implementing a simple rule

2. Main extension points for UML model analysis

Although many extension points are available in the analysis framework, you need only a few to create new categories and rules.
To see the available categories, you can open the Analysis Configuration dialog box and expand the **UML Model Metrics** and **UML Model Review** providers. Each category is contributed by an extension in a plugin.xml file such as the one in the com.ibm.xtools.analysis.uml.review.example plug-in that is available for you to download. For example, consider the following category from the plugin.xml file:

```xml
<analysisCategory
    class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
    id="com.ibm.xtools.analysis.uml.review.example.category.example"
    category="com.ibm.xtools.analysis.uml.review.category.naming"
    label="%example.category.example"/>
```

The class attribute defines a class that manages the category. Although you can write a custom category class, this code uses the default category that the API supplies, which is effective in most cases. Custom categories must handle common characteristics of a category such as managing a list of contained children, a label, an icon name, and other elements. This concept is more advanced; for now, the default category can provide all the services that are required for any category. In almost all situations, you use the supplied class for a category.

A category also has a unique ID, which is used by other categories and rules to determine containment. A category must also have a label and a description, which are displayed in the user interface.

In this example, the extension also has a category attribute, which indicates that this category is nested in another category whose unique ID is com.ibm.xtools.analysis.uml.review.category.naming. To create a top-level category, you replace the category attribute with provider=provider.id, as in the following example:

```xml
<analysisCategory
    class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
    id="com.ibm.xtools.analysis.uml.review.category.dependency"
    label="%review.category.dependency"
    provider="com.ibm.xtools.analysis.uml.review.provider"/>
</analysisCategory>
```

In this example, com.ibm.xtools.analysis.uml.review.provider is the unique identifier of the Java code review provider.

Rule extensions work in a similar way, but they include more detail. The following code shows a sample extension for a rule:

```xml
<analysisRule
    category="com.ibm.xtools.analysis.uml.review.example.category.example"
    class="com.ibm.xtools.analysis.uml.review.example.internal.rule.example.ClassNameStartingWithUpperCase"
```

---

1 For a quick reminder on Software Analyzer basics you can consult the following developerWorks article: *Static analysis IBM Rational Software Analyzer: Getting started*
The first part of the rule extension is almost identical to the category extension previously shown. In this case, the class attribute contains a fully qualified class path for the rule.

The rule also contains additional information. The help tag supplies an ID for context-sensitive help in a related help.xml file. This information allows a rule to provide a nicely formatted example and solution information to the user.

### 3. UML Model Review rules

The first half of a rule definition is an Eclipse extension. The second half is a Java class that uses the rule API or 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 is a complete rule that creates results for any line of code that uses the Java “==” operator to compare two objects:

```java
public class ClassNameStartingWithUpperCase extends ModelAnalysisRule {
    protected void doAnalyze(AnalysisHistory history,
                              Collection resourceContents) {

        // The selection criteria
        EObjectCondition classCondition = new EObjectCondition() {
            public boolean isSatisfied(EObject object) {
                ...
            }
        };

        // The selected UML elements
        Set selectedElements = new SELECT(new FROM(resourceContents),
                                            new WHERE(classCondition)).getEObjects();

        // List for elements non complying with the rule
        Set problemElements = new HashSet();

        // Rule logic
        ...

        // Creating Analysis results for the elements complying with the rule
        for (Iterator i = problemElements.iterator();
             i.hasNext();)
        {
```
The rule class extends the `com.ibm.xtools.analysis.model.internal.rule.ModelAnalysisRule` rule from the model analysis framework and, because this class is abstract, the derived rule must implement the `doAnalyze()` method. This method takes two parameters:

- **history**: This parameter points to the analysis session for the rules that are run. Every time you perform an analysis, a new history is registered to collect results. In addition to results, the history element also tracks which providers, categories, and rules were selected.
- **resourceContents**: This parameter is the collection of elements that is retrieved from parsing the model file.

The new `EObjectCondition` object defines the criteria that is used to filter the content of the `resourceContents` parameter, based on the information that you want the rule to extract. The `SELECT` statement performs the filtering.

You then need to encode the rule logic that identifies elements in the selection that do not comply with the rule.

The last step consists of iterating over the noncompliant elements and creating a model analysis result (`ModelAnalysisResult`) for each of them.

The `getProblemDescription()` method gathers the message to associate with each result. This message is displayed in the Software Analyzer Result view. This will allow you to have the message more easily translatable.

### 4. Writing new rules

After you have a basic understanding of the model analysis framework, you can use it to write a complete rule plug-in with a new category and rule.

#### 4.1. 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.review.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.xtools.analysis.model` plug-in contains the basic model analysis framework API and extension points.

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

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

The `org.eclipse.emf.query` plug-in contains the EMF query framework that retrieves and filters the elements that are checked against the different model review rules.

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

### 4.2. Create a rule class

In Java, all classes and interfaces start with an uppercase letter. In this example, you create a rule that detects the classes and interfaces in a model that start with a lowercase letter. This simple rule shows the main elements of the framework. Also,
because this rule is provided in Rational Software Architect in the UML Model Analysis feature, you can verify that the rule works as planned.

This rule must perform these actions:

1. Create a list of the classes and interfaces in the model to analyze.

2. Examine the names of the elements to determine whether they start with a lowercase letter or an uppercase letter.

To create the rule, you complete these steps:

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

2. Name the class ClassNameStartingWithUpperCase and ensure that the rule class extends the ibm.xtools.analysis.model.internal.rule.ModelAnalysisRule class.

3. Add a doAnalyze method and a getProblemDescription method to the class, as in the following example.

```java
import java.util.Collection;
import com.ibm.xtools.analysis.model.internal.rule.ModelAnalysisRule;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;

public class ClassNameStartingWithUpperCase extends ModelAnalysisRule {
    // Analyze the resources
    protected void doAnalyze(AnalysisHistory history, Collection resourceContents) {
    }

    // Retrieve the corresponding result description
    protected String getProblemDescription(Classifier problemClass) {
    }
}
```

4. To collect a list of classes and interfaces from the model, add the following lines of code to the doAnalyze() method:

```java
// The selection criteria
EOBJECTCondition classCondition = new EObjectCondition() {
    public boolean isSatisfied(EOBJECT object) {
        public boolean isSatisfied(EOBJECT object) {
            return (UMLPackage.Literals.CLASS.isInstance(object) |
                    UMLPackage.Literals.INTERFACE.isInstance(object));
        }
    }
};

// The selected UML elements
Set classes = new SELECT(new FROM(resourceContents), new WHERE(classCondition)).getEObjects();
```
The line of code that contains EObjectCondition defines the filter that retrieves only the elements of interest for the rule. In this example, the filter retrieves classes (UMLPackage.Literals.CLASS) and interfaces (UMLPackage.Literals INTERFACE).

The line of code that contains SELECT performs the filtering of the resources.

5. To analyze the selected resources to identify which element names start with a lowercase letter, create a list iterator to go over the elements that the filter returns by adding these lines to the end of the doAnalyze() method:

```java
for (Object classObj : classes) {
    Classifier classifier = (Classifier) classObj;
}
```

The name of the class is a string, so the easiest way to check if it starts with a lowercase letter is to use the java.util.regex.Pattern class.

6. To define the pattern, add the following line at the beginning of the class:

```java
private static final Pattern PATTERN = Pattern.compile("[pP]Lower\.*");
```

And update the for loop of the doAnalyze() method with the following code. This code will check that the name of the classifier (classifier.getName()) matches the pattern we defined (PATTERN).

```java
for (Object classObj : classes) {
    Classifier classifier = (Classifier) classObj;
    if (PATTERN.matcher(classifier.getName()).matches()){
    }
}
```

If the name matches the pattern, you need to create a new model analysis result to display in the Results view. Before you create the code that is specific to this rule, you should understand the setModelAnalysisResult method that you use to create the result.

The setModelAnalysisResult method takes five parameters:

- **history**: This parameter is the history of the object.
- **label**: This parameter is the description that is displayed in the Results view.
- **icon**: This parameter is not used in the current implementation, so set it to null.
- **modelElements**: This parameter is the list of URIs for the model elements that are associated with this rule. There is usually only one element, but for rules that are related to relationships between elements, such as the coupling review rules, you can store all the related elements in this list.
- **isVisualizable**: This parameter is a Boolean operator that controls whether to visualize the results. Setting this parameter to true has two effects. The first effect is that when you double-click the result, the result is displayed in a read-
only diagram. The second effect is the enabling of the **Browse Result** action, which you can use to see the associated model element in a Browse diagram. If you set this parameter to **false**, you navigate to the element in the model instead of opening a read-only diagram.

After you know how the `setModelAnalysisResult()` method works, you can finish creating the rule by creating results for the classes and interfaces that match the defined pattern. Because you do not need to visualize the results in the diagram, you can set the `isVisualizable` parameter to **false**.

Add the following lines of code to the if block.

```java
List<URI> uris = new ArrayList<URI>();
uris.add(EcoreUtil.getURI(classifier));
setModelAnalysisResult(history, getProblemDescription(classifier),
null, uris, false);
```

First, you gather the URI of the classifier; then, you create a new model analysis result. The label is gathered by calling the `getProblemDescription` method, which you need to populate. You use this function to allow the label to be easily translatable.

Add the following line to the `getProblemDescription` method:

```java
return ICUUtil.format(Messages.example_rule_ClassNameStartingWithUpperCase,
new Object[]{problemClass.eClass().getName(),
EMFCoreUtil.getQualifiedName(problemClass, true)});
```

For the code to work, you need to create a `Messages.java` class with the corresponding `Messages.properties` in a `com.ibm.xtools.analysis.uml.review.examples.myexample.l10n` package.

The following code shows the content of the `Messages.java` file:

```java
package com.ibm.xtools.analysis.uml.review.myexample.l10n;

import org.eclipse.osgi.util.NLS;
import com.ibm.xtools.analysis.uml.review.myexample.Activator;

public class Messages
extends NLS {

private static final String BUNDLE_NAME =
Activator.getDefault().getBundle().getSymbolicName() +
".internal.l10n.messages"; //SNON-NLS-1$;

// Strings for the description column of model report view

// Example:
public static String example_rule_ClassNameStartingWithUpperCase;

static {
NLS.initializeMessages(BUNDLE_NAME, Messages.class);
}
```
The following code shows the content of the Messages.class ___________:

```java
package com.ibm.xtools.analysis.uml.review.examples.myexample;
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;
import java.util.regex.Pattern;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.query.conditions.eobjects.EObjectCondition;
import org.eclipse.emf.query.statements.FROM;
import org.eclipse.emf.query.statements.SELECT;
import org.eclipse.emf.query.statements.WHERE;
import org.eclipse.gmf.runtime.emf.core.util.EMFCoreUtil;
import org.eclipse.uml2.uml.Classifier;
import org.eclipse.uml2.uml.UMLPackage;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;
import com.ibm.xtools.analysis.model.internal.rule.ModelAnalysisRule;
import com.ibm.xtools.analysis.model.internal.util.ICUUtil;
import com.ibm.xtools.analysis.uml.review.example.myexample.l10n.Messages;
public class ClassNameStartingWithUpperCase extends ModelAnalysisRule {
    private static final Pattern PATTERN = Pattern.compile("\\p{Lower}.*");
    // Analyze the resources
    protected void doAnalyze(AnalysisHistory history,
        Collection resourceContents) {
        // The selection criteria
        EObjectCondition classCondition = new EObjectCondition() {
            public boolean isSatisfied(EObject object) {
                return (UMLPackage.Literals.CLASS.isInstance(object)
                        || UMLPackage.Literals.INTERFACE.isInstance(object));
            }
        };
        // The selected UML elements
        Set classes = new SELECT(new FROM(resourceContents), new WHERE(classCondition)).getEObjects();
        // Elements not complying with the rule
        Set<Classifier> problemClassifiers = new HashSet<Classifier>();
        // Rule logic
        for (Object classObj : classes) {
            Classifier classifier = (Classifier) classObj;
            if (PATTERN.matcher(classifier.getName()).matches()){  
                List<URI> uris = new ArrayList<URI>();
                uris.add(EcoreUtil.getURI(classifier));
                setModelAnalysisResult(history, getProblemDescription(classifier),
```
4.3. Create extension points

After you complete the rule class, you need to describe it through the provided extension points in the model analysis framework so that the review engine can discover it. In this example, you first create your own category and then associate the rule with it.

To create a new category, complete these steps:
1. Open the example plug-in manifest file, click the Extensions tab, and click Add.
2. From the list, select the com.ibm.rsaz.analysis.core.analysisCategory extension point as shown in Figure 4.
3. On the Extension page, you should now see a new extension point entry. Right-click the extension point and add a new analysis category.
4. Provide the extension details as illustrated in Figure 5.
This category has a parent category called com.ibm.xtools.analysis.uml.review.category.naming. So in the user interface, our Example category will be nested under the existing Java Naming category.

You can use the DefaultAnalysisCategory class to supply a predefined category that provides all the functionality that is required for most categories that you create.

If you provide the details listed above, the following text is inserted into the plugin.xml file:

```xml
<extension point="com.ibm.rsaz.analysis.core.analysisCategory">
  <analysisCategory
    class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
    id="com.ibm.xtools.analysis.uml.review.example.category.naming.example"
    category="com.ibm.xtools.analysis.uml.review.category.naming"
    label="Example"/>
</extension>
```

Next, you need to add the rule to the new category.

5. Add a new extension to the plug-in as you did for the category. This time, select the `com.ibm.rsaz.analysis.core.analysisRule` extension point.

6. On the Extensions page, right-click the rule extension and add a new analysis rule.

7. Populate the rule details as shown in Figure 6.
8. Enter a label, which will be 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.review.example.category.naming.example" class="com.ibm.xtools.analysis.uml.review.example.internal.rule.example.ClassNameStartingWithUpperCase" id="com.ibm.xtools.analysis.uml.review.example.rule.ClassNameStartingWithUpperCase">
    <label>Class names begin with an uppercase letter</label>
    <severity>0</severity>
  </analysisRule>
</extension>
```

### 4.4. 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 workbench, import a UML model that contains at least one class that starts with a lowercase letter, so that there are model elements to review.
4. From the menu, click Run > Analysis.
5. In the dialog box that is displayed, create a new analysis configuration and click the Rules tab.
3. In the hierarchy, expand **Java Naming** until the **Example** subcategory is visible, as shown in Figure 7.

**Figure 7**

![Software Analyzer Configurations](image)

4. 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.
5. Conclusion

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

The second installment explores the model analysis API further and shows how to use it to create new a UML metrics rule.