

Static analysis IBM Rational Software Analyzer: Part 3. Enhancing rules for Java code review

Creating severity indicators, templates, user Help, generic variables, and configurable parameters

Skill Level: Advanced

[Steve Gutz \(sgutz@ca.ibm.com\)](mailto:sgutz@ca.ibm.com)
Manager, Rational Software Analyzer
IBM Corporation

29 Apr 2008

This is the third article in a four-part series about IBM® Rational® Software Analyzer and its related capabilities in Rational Application Developer and Rational Software Architect. It walks you through the process of creating your own custom rules and Rule Sets and using other advanced capabilities that extend Java code review. Rational Software Analyzer encourages and simplifies the process of automated code quality improvements. It was initially designed as an API and user interface to create and integrate static analysis tools into other Rational products, such as Rational Application Developer and Rational Software Architect. It has evolved into a complete, standalone offering that enables software developers to easily use automated code review and structural analysis in their daily development process.

In Part 1 of this four-part series of articles, we examined static analysis from a high level and introduced the common user interface for analysis available in IBM® Rational® Software Analyzer (also in the Rational® Software Architect and Rational® Application Developer) and the process of creating an analysis configuration by selecting providers, categories and rules. We also discussed the results view that shows the outcome of the static analysis, as well as some of the basic reporting capabilities built into the software.

In Part 2, you learned how to write rules to extend the current Java code review Rule Set. The Rational Software Analyzer analysis framework (referred to in code as

`rsar`) includes several extension points and a complete API for adding new forms of analysis and new rules, but more importantly for this purpose, it also supplies a fully functional analysis provider to perform automated code reviews for Java source code.

The rule that you created in the Part 2 is fairly uncomplicated, but it ignores many capabilities because of its simplicity. This article describes some of the more advanced features of rule creation. It describes how to add a severity indicator and generic rule variables that enable users to add configurable parameters. It also explains detail provider tools that allow rule authors to provide more detailed information about a rule, such as examples and solutions.

This and the previous article describe enough of the framework for you to understand and write any rule that you need.

Creating user Help for rules

After you have finished building your rule, you need to tell the user why it exists and how to fix the problems that it reports. You do this by adding an entry to the Eclipse Help system to show code examples that cause the rule to react, along with solutions. Adding this kind of rule help is actually a relatively simple task if you follow these basic steps:

1. Create a Help document by using an HTML editor.
2. Add a `help="my_help_id"` tag to your rule extension.
3. Create or update your plug-in's `help.xml` file to add a Help context for your rule.

When creating a Help document, take the easy route. For consistency, it is best to make the Help for your rule look the same as the Help for other rules. The best way to do this is to clone the HTML file from one of the rules included with Rational Software Analyzer.

1. By convention, create a folder in your plug-in named `ruleinfo` and put your HTML file inside of that, as Listing 1 shows.

Listing 1 is an example of the Help code for a rule in Rational Software Analyzer.

Listing 1. Example of HTML code for Help

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!--
*****
```

```

* Licensed Materials - Property of IBM *
* (C) Copyright IBM Corp. 2006. All Rights Reserved. *
* *
* US Government Users Restricted Rights - Use, duplication or disclosure *
* restricted by GSA ADP Schedule Contract with IBM Corp. *
*****
-->
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<head>
<meta HTTP-EQUIV="Content-Type"
CONTENT="text/html; charset=iso-8859-1"></meta>
<link rel="stylesheet" type="text/css" href="../ruleinfo.css" >
</head>
<body>
<span class="Text">
<table border="0">
<tr><td><span class="SubHeader">Example</span></td></tr>

<tr><td>
<table bgcolor="#F7F8F9" border="1">
<tr><td>
<span class="JavaKeyword">package</span> default;<br>
<br>
<span class="
"indent"></span><span class="Javadoc">/**<br>
<span class="indent"></span><span class="indent"></span> *Javadoc<br>
<span class="indent"></span> */<br></span>
<span class="
"indent"></span><span class="JavaComment">/* <br>
<span class="indent"></span><span class="indent"></span>*Non-Javadoc<br>
<span class="indent"></span><span class="indent"></span>*/<br></span>
<span class="JavaKeyword">public class</span> ClassA {<br>
<span class="indent2"></span>
<span class="JavaKeyword">public</span> ClassA () { <br>
<span class="indent2"></span>}<br>
<span class="indent"></span>}<br>
<br>

</td></tr>
</table>

</td></tr>
</table>

<br>
<table border="0">
<tr><td>
<span class="SubHeader">Solution</span></td></tr>
<tr><td>
Place the non-Javadoc in a proper spot, preferably inside the
body of the declaration.
<br>

<table bgcolor="#F7F8F9" border="1">
<tr><td>
<span class="JavaKeyword">package</span> default;<br>
<br>
<span class="
"indent"></span><span class="Javadoc">/**<br>
<span class="indent"></span><span class="indent"></span> * Javadoc<br>
<span class="indent"></span> */<br></span>

<span class="indent"></span>
<span class="JavaKeyword">public class</span> ClassA {<br>
<span class="indent2"></span>
"indent2

```

```

        "></span><span class="JavaComment"> /*<br>
        <span class="indent2"></span>*Non-Javadoc<br>
        <span class="indent2"></span>*/<br></span>

        <span class="indent2"></span>
        <span class="JavaKeyword">public </span> ClassA () { <br>
        <span class="indent2"></span></span>}<br>
        <span class="indent"></span>}
    </td></tr>
</table>
</td></tr>
</table>
</span>
</body>
</html>

```

Notice that this rule example references the cascading style sheet file named `ruleinfo.css`. To simplify the HTML in your rule Help, you will probably want one of these CSS files in your `ruleinfo` folder, as well. It looks like Listing 2.

Listing 2. RuleInfo cascading style sheet (ruleinfo.css)

```

A:link
{ color: #000099; text-decoration: underline; font-size: 11px; font-family: tahoma }
A:active
{ color: #000099; text-decoration: underline; font-size: 11px; font-family: tahoma }
A:visited
{ color: #000099; text-decoration: underline; font-size: 11px; font-family: tahoma }
A:hover
{ color: #000099; text-decoration: underline; font-size: 11px; font-family: tahoma }

.Text
{font-family: tahoma; font-size: 11px; color: black}
.Italic
{font-family: tahoma; font-size: 11px; font-style: italic}
.Strong
{font-family: tahoma; font-size: 11px; font-weight: bold}
.Header
{font-family: tahoma; font-size: 16px; color: black; font-weight: bold; }
.SubHeader
{font-family: tahoma; font-size: 11px; color: black; font-weight: bold; }
.Code
{font-family: tahoma; font-size: 11px;
color: black; font-weight: bold; background: #F7F8F9; }
.JavaType
{font-family: tahoma; font-size: 11px; color: black; font-weight: bold; }
.JavaKeyword
{font-family: tahoma; font-size: 11px; color: #7F0055; font-weight: bold; }
.JavaConst
{font-family: tahoma; font-size: 11px; color: #2A00FF; font-weight: bold; }
.JavaComment
{font-family: tahoma; font-size: 11px; color: #3F7F5F }
.XmlDefinition {font-family: tahoma; font-size: 11px; color: black;
text-decoration: italic }
.indent {padding-left: 10px;}
.indent2 {padding-left: 30px;}
.indent3 {padding-left: 50px;}
.indent4 {padding-left: 70px;}

```

2. The next step required to add Help your rules is to create unique Help IDs for them. After you decide on your IDs, you can add a Help attribute to

your rule. For example:

```
<analysisRule
  category="analysis.codereview.java.examples"
  class="com.ibm.rsar.codereview.java.examples.RuleFinalizerSuper"
  help="com.ibm.rsar.codereview.java.examples.finalSuper"
  id="codereview.java.example.finalSuper"
  label="Avoid finalize() methods that call only super">
</analysisRule>
```

3. Finally, you need to associate this ID with a Help context. Do this by creating or updating the help.xml file for your plug-in.

Listing 3 is an example of a simple help.xml file.

Listing 3. Simple help.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<contexts>
  <context id="examples" title="Examples">
    <description>This category contains example rules for Java code review.
    </description>
  </context>
  <context id="finalSuper" title="Avoid finalize() methods that call only super">
    <description>Providing finalize() methods that call only super is a waste of time
    </description>
    <topic href=
      ".\ruleinfo\examples\finalsuper.html" label="Examples and Solutions"/>
  </context>
</contexts>
```

There are a few noteworthy features in this file.

- First, notice that the first context describes a category. Yes, categories can have Help too, and it is implemented the same way: by using a Help attribute in the extension.
- Second, notice that all Help context IDs are named relative to the plug-in. This is why the ID in the help.xml file is `finalSuper`, but the fully qualified name is used in the rule definition.
- Finally, ensure that the topic includes a relative path to the HTML file that will be displayed when the user clicks the link on the top-level Help page (Examples and Solutions in this case).

The HTML Help file for this example here is shown in Listing 4. Notice that, in this case, the only solution is to remove the `finalize()` method; therefore, no solution example code is included. However, most rules will include a corrected code

fragment to show the solution.

Listing 4. HTML Help file

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!--
*****
* Licensed Materials - Property of IBM
* (C) Copyright IBM Corp. 2006. All Rights Reserved.
*
* US Government Users Restricted Rights - Use, duplication or disclosure
* restricted by GSA ADP Schedule Contract with IBM Corp.
*****
-->
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<head>
<meta HTTP-EQUIV="Content-Type"
CONTENT="text/html; charset=iso-8859-1"></meta>
<link rel="stylesheet"
type="text/css" href="../ruleinfo.css" >
</head>
<body>
<span class="Text">
<table border="0">
<tr><td><span class=
"SubHeader">Example</span></td></tr>

<tr><td>
<table bgcolor="#F7F8F9" border="1">
<tr><td>
<span class="indent"></span>
<span class="JavaKeyword">public class</span> ClassA {<br>
<span class="indent2"></span>
<span class="JavaKeyword">public</span> finalize() { <br>
<span class="indent3"></span>super.finalize();<br>
<span class="indent2"></span>}<br>
<span class="indent"></span>}<br>
<br>

</td></tr>
</table>

</td></tr></table></span>

<br>
<span class="Text">
<table border="0">
<tr><td>
<span class=
"SubHeader">Solution</span></td></tr>
<tr><td>
Remove the superfluous finalize method
</td></tr>
</table>
</span>
</body>
</html>

```

There is one final step needed to enable context-sensitive Help for your rule. You need to link the help.xml file to the Eclipse engine through an extension point. Add the following to your plugin.xml file:

```

<extension
  name="Context-Sensitive Help"
  point="org.eclipse.help.contexts">
  <contexts
    file="help.xml" />
</extension>

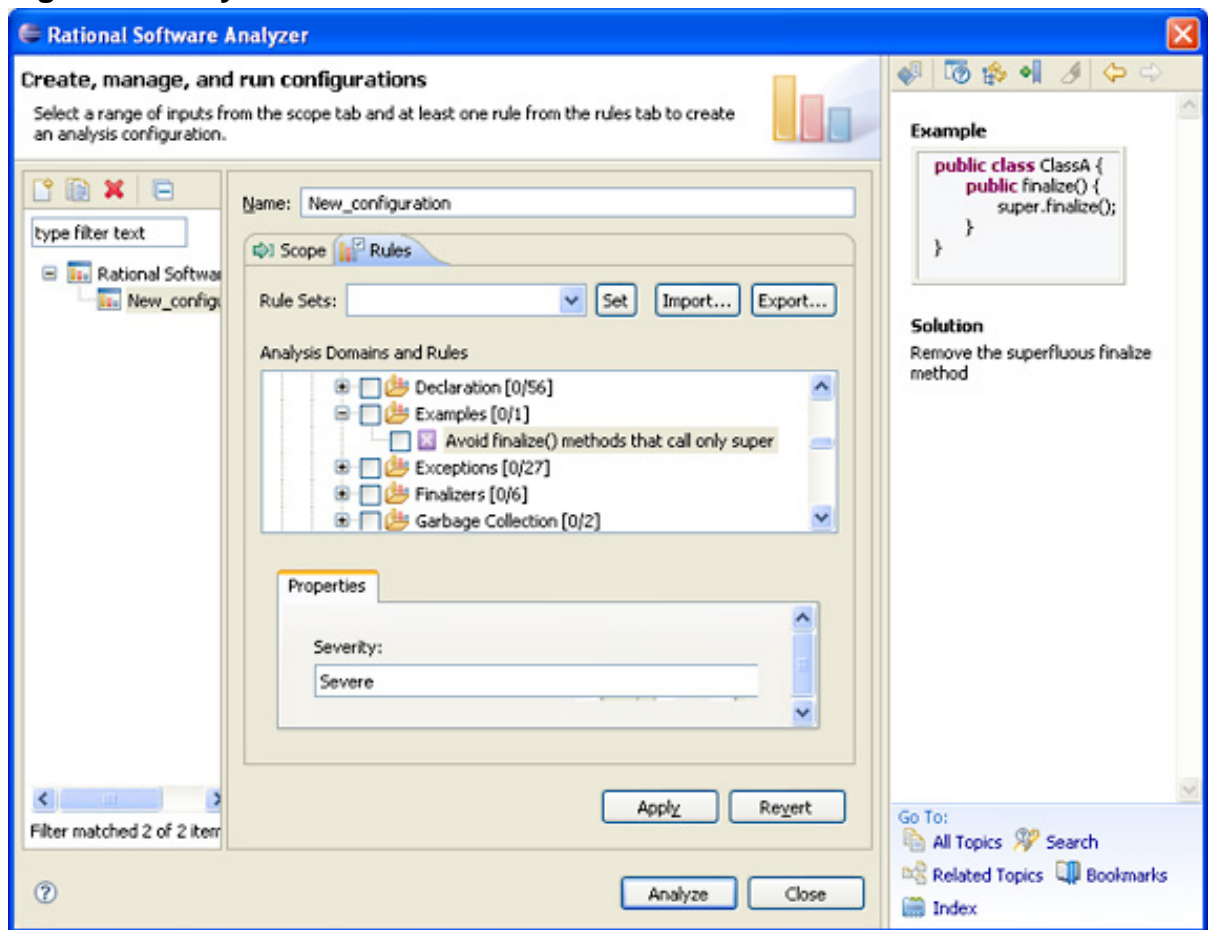
```

Tip:

Don't forget to export the ruleinfo folder and the help.xml file with your plug-in. Open the plugin.xml file and select the **Build** tab, and then select these resources in the UI (user interface).

When your Help is completed and correctly integrated, the user will be able to use the F1 key to display it in either the Analysis Configuration dialog or in the analysis results view (Figure 1).

Figure 1. Analysis results view



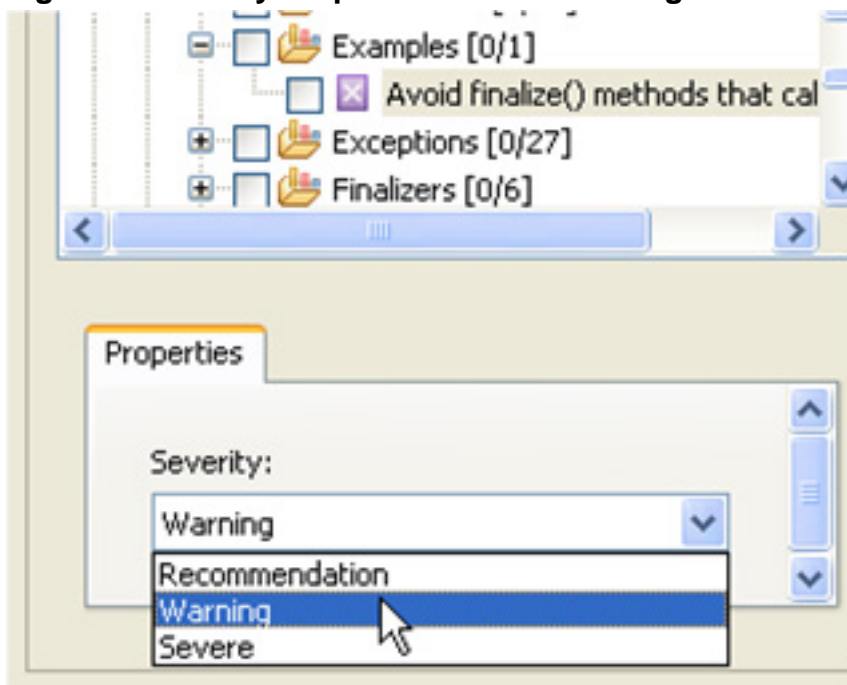
Indicating rule severity

The Rational Software Analyzer configuration dialog provides a reserved area to allow users to display and change any properties associated with the rule. In addition to any custom details defined by the rule author, this property panel provides a drop-down list so that the user can change the default severity for the rule, as shown in Figure 2. The value of this property controls the icon associated with the rule and its results, to indicate importance. There are three basic modes of severity:

- recommendation
- warning
- severe

You can use these levels to indicate to the user how important it is to address the results produced. It is not mandatory to use them, but you should specify a default severity for any rules that you create. This will help guide the user when results are reported for your rules (see Figure 2).

Figure 2. Severity drop-down menu to change the severity of a rule



Use the wizard in Eclipse to edit the `analysisRule` and set the severity field by using the drop-down list:

- 0 = recommendation
- 1 = warning

- 2 = severe

This code snippet shows the resulting XML text:

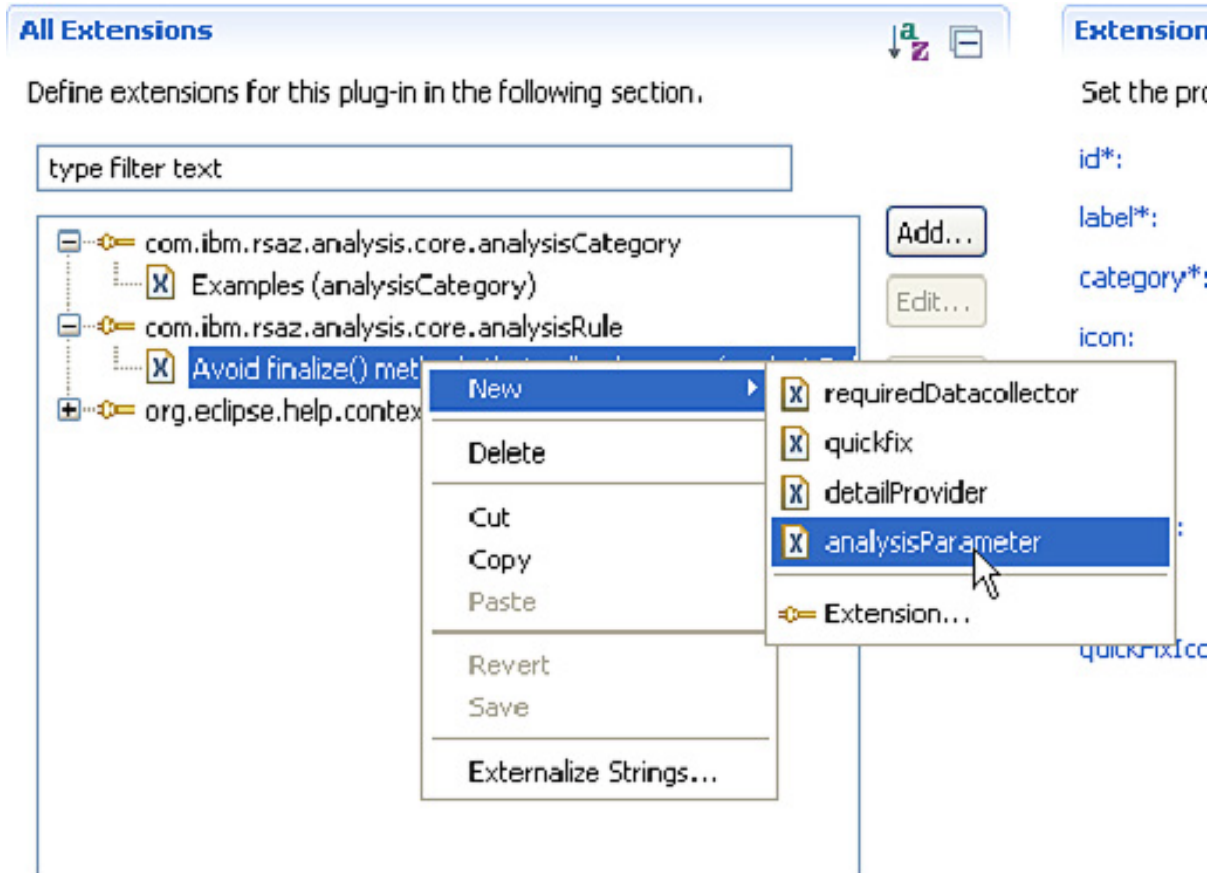
```
<analysisRule
  category="analysis.codereview.java.examples"
  class="com.ibm.rsar.codereview.java.examples.RuleFinalizerSuper"
  help="com.ibm.rsar.codereview.java.examples.finalSuper"
  id="codereview.java.example.finalSuper"
  label="Avoid finalize() methods that call only super"
  severity="2">
</analysisRule>
```

Including generic analysis element parameters

There will be situations when you will need to allow the user to store custom data for a rule, category, or provider (analysis elements). The analysis rule API enables you to create custom variables and present them on the Properties tab when the analysis element is selected.

1. Use the Eclipse extension editor to add a new `analysisParameter`, as shown in Figure 3.

Figure 3. Adding a new parameter



Parameters are **name** (`testParameter` in this example) and **value** (here: `Hello`) with this additional information (see Figure 4):

- The **Label** field contains the text that the user will see when viewing the parameters field in the user interface (here: `Test`).
- The **Type** field determines the data type for the value and is used to provide the correct field editor, as well as field validation. Valid values for this field are `string` or `integer`.
- The **Style** field controls the type of UI control used to display the parameter. For example, data can be represented as standard text, a combo box, or a check box.

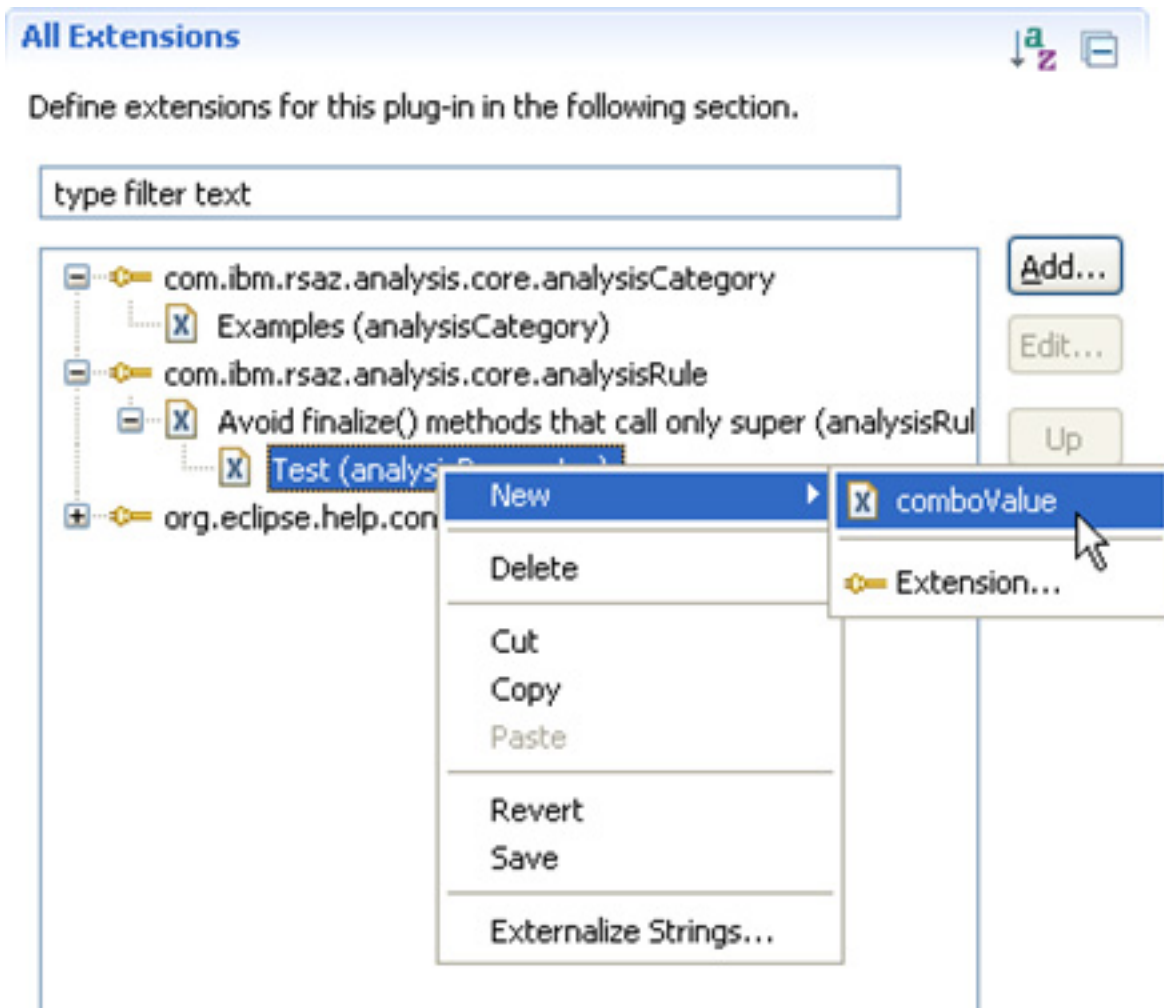
Figure 4. Extension element details

Extension Element Details	
Set the properties of "analysisParameter"	
name*:	<input type="text" value="testParameter"/>
value*:	<input type="text" value="Hello"/>
label:	<input type="text" value="Test"/>
type:	<input type="text" value="string"/> ▼
style:	<input type="text" value="combo"/> ▼

If the style is a combo box, as previously in Figure 4, then its values must also be added.

2. Add a new `comboValue` extension to the `analysisParameter` for each value that you want to include in the list, as shown in Figure 5. `ComboValue` extensions have only a value field that contains the text that the user will see when the combo box is expanded.

Figure 5. Adding a `comboValue` extension



Now that you know how to add custom data parameters to a rule extension, you're ready to learn how to use those values in your code.

3. To access a rule parameter in your rule's `analyze()` method, you can use the following code fragment:

```
// Extract required parameters
AnalysisParameter rv = this.getParameter( "parameterName" );
String value = rv.getValue();
```

Creating rule templates

As you begin writing rules, you will quickly realize that many rules are very similar. For example, you may want rules that generate results when code extends various Java™ classes. Given that the only fundamental difference between them is the

name of the class, it would help to have a simple way to create a template for a single rule that handles all cases. Fortunately, the analysis API provides a tool so that you can do this.

By exploiting rule parameters, you can create simple, flexible templates for rules. If you examine the Java code review rules, you will see several templates. Listing 5 shows a sample template that generates results for extending classes.

Listing 5. Template example

```

/*
 *-----+
 * Licensed Materials - Property of IBM
 * (C) Copyright IBM Corp. 2005,2007. All Rights Reserved.
 *
 * US Government Users Restricted Rights - Use, duplication or disclosure
 * restricted by GSA ADP Schedule Contract with IBM Corp.
 *-----+
 */
package com.ibm.rsaz.analysis.codereview.java.rules.base.templates;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.TypeDeclaration;

import com.ibm.rsaz.analysis.codereview.java.AbstractCodeReviewRule;
import com.ibm.rsaz.analysis.codereview.java.CodeReviewResource;
import com.ibm.rsaz.analysis.codereview.java.ast.ASTHelper;
import com.ibm.rsaz.analysis.codereview.java.rulefilter.SuperClassRuleFilter;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;

public class Template_Avoid_Extending_Class
    extends AbstractCodeReviewRule
{
    private static final String CLASS = "CLASS"; //$NON-NLS-1$

    private String className;

    /**
     * Analyze this rule
     *
     * @param history A reference to the history record for this analysis
     */
    public void analyze(AnalysisHistory history, CodeReviewResource resource ) {
        // Extract required variables
        className = getParameter( CLASS ).getValue();

        // Obtain a list of string literals
        List list = resource.getTypedNodeList( resource.getResourceCompUnit(),
            ASTNode.TYPE_DECLARATION );
        ASTHelper.satisfy( list, new SuperClassRuleFilter( className, true ) );
        for( Iterator it = list.iterator(); it.hasNext(); ) {
            TypeDeclaration td = (TypeDeclaration)it.next();
            resource.generateResultsForASTNode( this, history.getHistoryId(),
                td.getSuperclassType() );
        }
    }
}

```

Notice that the code uses a rule parameter called `CLASS` and uses its value to check the superclass. The extension of an `analysisRule` using this template looks like this code:

```
<analysisRule
  category="codereview.java.category.exceptions"
  class="com.ibm.rsaz.analysis.templates.Template_Avoid_Extending_Class"
  id="codereview.java.rules.exceptions.RuleExceptionsExtendError"
  label="%label.template.avoid.extending.class"
  severity="0"
  <analysisParameter name="CLASS" value="java.lang.Error" />
</analysisRule>
```

Notice the `analysisParameter` line, which defines the `CLASS` parameter and assigns a value, `java.lang.Error`, to it. Because this parameter does not have a label, it will not appear in the rule's Properties tab. This is how invisible parameters are created.

The other notable line in this test is the class definition, which points to the template class, rather than one that you might create.

Writing custom rules

Templates can also be exploited by the end user to define new rules without writing any code. If you have previously defined a rule class that can be used as a template, you can use the `analysisRuleTemplate` extension point to expose it to the end user. Consider the extension example in Listing 6.

Listing 6. Extension example

```
<extension
  point="com.ibm.rsaz.analysis.core.analysisRuleTemplate">
  <ruleTemplate
    provider="codereview.java.analysisProvider"
    class="com.ibm.rsaz.analysis.templates.Template_Avoid_Extending_Class"
    id="codereview.java.templates.Template_Avoid_Extending_Class"
    label="Avoid extending <CLASS>">
    <ruleParameter
      label="Class Name:"
      description="Enter a qualified Class Name"
      name="CLASS"
      style="text"
      type="string"/>
    </ruleTemplate>
  </extension>
```

This example creates a new rule template that allows the user to define new rules to report and code that extends a given class. It uses a rule parameter to declare and describe a field where the user can type a qualified class name.

As described in Part 1 of this series, any rule templates that you define will appear in

the **New Rule** wizard in Eclipse Preferences. To find these:

1. Select **Window > Preferences** from the main Eclipse menu
2. Then, in the Preferences tree, select **Analysis > Custom Rules and Categories**.

Adding QuickFix support

Sometimes, the results of a rule are so easy to fix that you can provide a simple repair procedure for the user. This article does not cover the intricacies of using Java tools to manipulate a source file, but does discuss the basic interface requirements. The `analysisRule` extension offers a `quickfix` sub-extension point where you can place the unique identifier for your QuickFix, as in this example:

```
<analysisRule
  category="analysis.codereview.java.examples"
  class="com.ibm.RSAR.codereview.java.examples.RuleFinalizeSuper"
  icon="analysis.codereview.java.examples"
  id="codereview.java.example.finalSuper"
  label="Avoid finalize() methods that call only super"
  severity="2">
  <quickfix id=
    "com.ibm.rsar.codereview.java.examples.quickfix.FixFinalSuper" />
</analysisRule>
```

Note:

If violations of your rule can be resolved in more than one way, you can place more than one `quickfix` extension in the rule definition. In the results view, each quick fix that you specify will appear in the result drop-down menu.

A QuickFix is, of course, an extension point available from the analysis API. To define a new one, use the `analysisRuleQuickFix` extension point. This extension requires that you define a unique ID, a class to implement the QuickFix, and a label. After it is defined, it will resemble this plugin.xml file fragment:

```
<extension
  point="com.ibm.rsaz.analysis.core.analysisRuleQuickFix">
  <analysisRuleQuickFix
    class="com.ibm.rsar.codereview.java.examples.quickfix.FixFinalSuper"
    label="%label.stringLiteral.quickfix"
    id=
      " com.ibm.RSAR.codereview.java.examples.quickfix.FixFinalSuper" />
```

The label field in this extension is optional. If you omit this value, the analysis framework will automatically set the QuickFix label to "QuickFix." The only time that you actually need to specify a label is in cases where you have more than one

QuickFix for the rule.

The `quickfix` class that you implement must extend the `AbstractAnalysisQuickFix` class supplied by the API. Strictly speaking, this is required only to implement a `quickfix()` method that receives the analysis result being fixed. Within this `quickfix` method, you can use Java tools to perform any required modifications. To assist with the implementation of the Java code review QuickFix, the analysis API provides the `JavaCodeReviewQuickFix` base class. This class does all of the heavy lifting needed to fix a problem, and you should extend it.

The code in Listing 7 implements a simple QuickFix for one of the rules supplied with Rational Software Analyzer:

Listing 7. QuickFix for one of the rules included with Rational Software Analyzer

```
public class RuleComparisonReferenceEqualityQuickFix extends
    JavaCodeReviewQuickFix {

    private static final String NOT = "&quot;!&quot;; //NON-NLS-1$
    private static final String CLOSE_PAREN = "&quot;)&quot;; //NON-NLS-1$
    private static final String EQUALS = "&quot;.equals( &quot;; //NON-NLS-1$
    private InfixExpression infixExpr = null;

    public TextEdit fixCodeReviewResult ( ASTNode theNode, IDocument docToChange ) {

        // reset
        infixExpr = null;

        if(theNode instanceof InfixExpression) {
            infixExpr = (InfixExpression)theNode;
        }

        if(infixExpr != null) {
            ASTNode enclosingClass = this.getEnclosingClass(infixExpr);
            AST ast = enclosingClass.getAST();
            ASTRewrite rewriter = ASTRewrite.create( ast );

            Expression left = infixExpr.getLeftOperand();
            Expression right = infixExpr.getRightOperand();
            Operator op = infixExpr.getOperator();

            StringBuffer replaceString = new StringBuffer(left.toString());
            replaceString.append(EQUALS);
            replaceString.append(right.toString());
            replaceString.append(CLOSE_PAREN);

            ASTNode methodInv = (Operator.NOT_EQUALS.equals( op )) ?
                rewriter.createStringPlaceholder(NOT + replaceString.toString(),
                    ASTNode.METHOD_INVOCATION) :
                rewriter.createStringPlaceholder(replaceString.toString(),
                    ASTNode.METHOD_INVOCATION);

            rewriter.replace(infixExpr, methodInv, null);

            TextEdit edits = new MultiTextEdit();
            edits.addChild(rewriter.rewriteAST(docToChange, null));

            return edits;
        }
    }
}
```

```

        return null;
    }
    /*
    * Returns the ASTNode of type
    * TYPE_DECLARATION or ANONYMOUS_CLASS_DECLARATION
    * that encloses the given node.
    * It returns null if there is not enclosing node of those types
    */
    private ASTNode getEnclosingClass( ASTNode node ) {

        ASTNode currentNode = node;

        while ( currentNode != null &&
            currentNode.getNodeType() != ASTNode.TYPE_DECLARATION &&
            currentNode.getNodeType() != ASTNode.ANONYMOUS_CLASS_DECLARATION ) {
            currentNode = currentNode.getParent();
        }

        return currentNode;
    }
}

```

Adding Rule Set support

Suppose that you have created many rules of your own and have scattered them over several different categories. If your rules fit into a particular theme (security, for example) it may be desirable to allow the user to quickly select them without sifting through the rule tree.

Fortunately, you can achieve this simply and without writing any code at all by using the extension points provided by the static analysis framework. This is known as a Rule Set. It can be implemented by adding `analysisRuleSet` extensions to your `plugin.xml` file, as in this example:

```

<extension
    point="com.ibm.rsaz.analysis.core.analysisRuleSet">
    <analysisRuleSet
        id="analysis.codereview.java.ruleset.quick"
        label="My Rule Set">
        <analysisRuleSetRule id="codereview.java.rules.awt.RuleAwtPeer"/>
        <analysisRuleSetRule id="codereview.java.threads.whilesleep"/>
        <analysisRuleSetCategory id="codereview.java.j2sebestpractices.loops"/>
    </analysisRuleSet>
</extension>

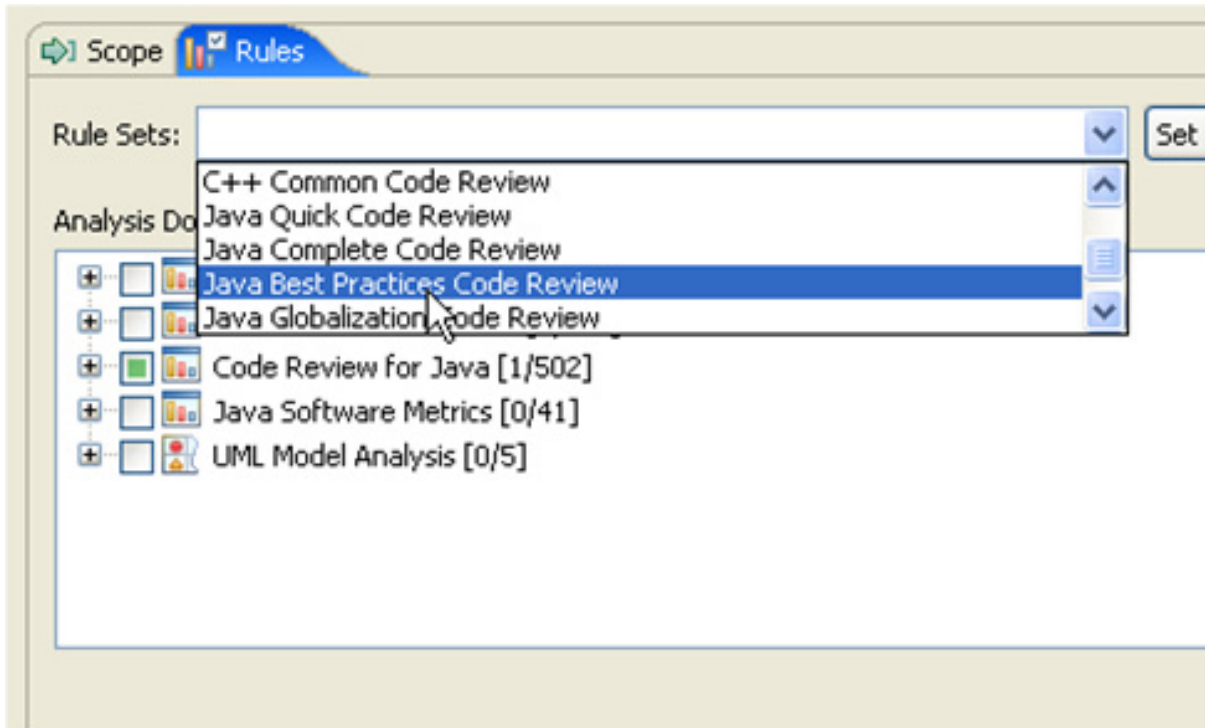
```

In this example, the defined Rule Set includes two separate rules and one entire category. Notice that including a category will automatically include any of the rules and categories that the category contains. The ID used in the `analysisRuleSetRule` and `analysisRuleSetCategory` entries is the unique identifier that was defined for the rules and categories, respectively.

Rule Sets are additive. If you define two Rule Set extensions with the same ID, then the Rule Set will include all defined elements of both extensions. In this case, however, only one of the extensions should define a value for the label.

Rule Sets will appear in the Rules tab of the analysis input dialog. The user can browse the known Rule Sets by using the drop-down list and selecting the desired item (see Figure 6). When the Set button is clicked, all rules and categories in the Rule Set will be selected in the rule tree.

Figure 6. Rule Sets under the Rules tab



Summary and what's next

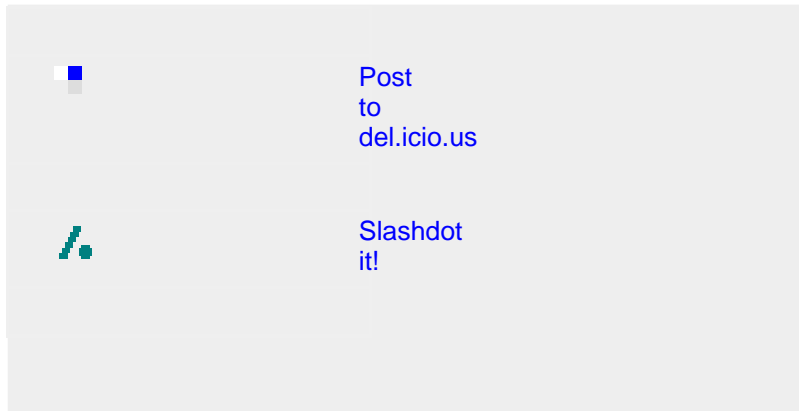
This article covered the concepts of rule severity, custom rule parameters, and creating rule templates. With this information and what Part 2 covered, you now have enough knowledge to write your own rules.

In Part 4 of this series, we will dive much deeper into the analysis API and use it to create new analysis providers, custom categories, and result viewers.

Share this article



Digg
this
story



Resources

Learn

- Visit the [Rational software area on developerWorks](#) for technical resources and best practices for Rational Software Delivery Platform products.
- Explore [Rational computer-based, Web-based, and instructor-led online courses](#). Hone your skills and learn more about Rational tools with these courses, which range from introductory to advanced. The courses on this catalog are available for purchase through computer-based training or Web-based training. Additionally, some "Getting Started" courses are available free of charge.
- Subscribe to the [Rational Edge newsletter](#) for articles on the concepts behind effective software development.
- Subscribe to the [IBM developerWorks newsletter](#), a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.
- Browse the [technology bookstore](#) for books on these and other technical topics.

Get products and technologies

- Download [trial versions of IBM Rational software](#).
- Download these [IBM product evaluation versions](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Tivoli®, and WebSphere®.

Discuss

- [Participate in the discussion forum for this content](#).
- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Steve Gutz

Steve Gutz is a development manager responsible for IBM Rational's code analysis tools. In addition architecting and managing IBM's commercial products Steve is also a past contributor on the Eclipse Test and Performance Tools project (TPTP), focusing on improvements in implementation and integration of basic code review tools. Previous to joining the IBM Ottawa Lab in 2002, he held senior management and executive positions in several public and private companies including two of his own successful start-ups. He is also the author of two books and many articles, and

is a regular conference speaker.