Getting Started with Rule Development

This guide will help you through the process of creating a custom rule for Kiuwan.

If you are looking for a quick tutorial on how to execute a custom rule analysis in Kiuwan, please check the Quick start guide page.

You can read more detailed information on this topic in the Rule Development Manual, included in the Kiuwan Local Analyzer bundle, under the /development folder.

Basic information

A rule for Kiuwan is a program that performs static analysis over the source code. How does this work?

As the diagram shows, the source code passes through a process named **parsing** to obtain an object called **AST** (Abstract Syntax Tree) which is a representation of the abstract syntactic structure of source code written in a programming language.

Each node of the tree denotes a construct occurring in the source code, so your rule has to travel among those nodes searching for the conditions which represent a violation of a certain good coding convention.

A rule is nothing more than a Java class which must accomplish the following rules:

- Extend **AbstractRule** (*com.als.core.AbstractRule*).
  For some technologies, there are more concrete abstract rule classes to extend from: *com.als.cobol.rules.AbstractCobolRule*, *com.optimyth.sql.rules.AbstractSqlRule*… Each one of them extends from *com.als.core.AbstractRule*.
- Implement an **initialize** method.
  This is where you can recover the values of the properties to customize your rule analysis. This method will be executed just once per launched analysis.
- Implement a **visit** method.
  This is where the main functionality of the rule is placed. This method will be executed one time per analyzed file.
- Implement a **post-process** method.
  This is where your rule can perform actions on information retrieved during the analysis of each file. This method will be executed just once per launched analysis.
- Report a **RuleViolation** (*com.als.core.RuleViolation)* in the **RuleContext** (*com.als.core.RuleContext*).

Example

```
import com.als.core.AbstractRule;
import com.als.core.RuleContext;
import com.als.core.RuleViolation;
import com.als.core.ast.BaseNode;

public class MyDummyRule extends AbstractRule {
  protected void initialize (RuleContext ctx) {
    super.initialize(ctx);
    // recover properties values. It could be empty if your rule hasn't
got any property.
  }

  protected void visit (BaseNode root, final RuleContext ctx) {
    // rule body...
    if (violation) {
      // Report a violation
      RuleViolation rv = createRuleViolation(ctx, nodeViolated.
getBeginLine());
      ctx.getReport().addRuleViolation(rv);
    }
  }

  protected void postProcess (RuleContext ctx) {
    super.postProcess(ctx);
    // postprocess body. It could be empty if your rule doesn't need to
postprocess any information
  }
}
```

By using the APIs provided you can traverse the AST in a variety of different ways. For further information, please check the Rule Development Manual, section "API Alternatives", under the developer folder in the Kiuwan Local Analyzer distribution.

- Abstract Syntax Tree (low-level and high-level) API.
- NavigableNode / TreeNode API.
- XPath rules.
- Query API.

There are additional static analysis facilities also available*. For further information, please check the Rule Development Manual, section **Additional Static Analysis facilities**, under the developer folder in the Kiuwan Local Analyzer distribution.

- Control-flow graph (CFG) and data-flow analysis.
- Library metadata.
- Tainting propagation.
- Local / Global Symbol Table.

(*) Check the specific coverage of these facilities by language.

Let's get started!

The best way to get to know the API is by developing your first rule. We are going to implement a rule which is described as follows:

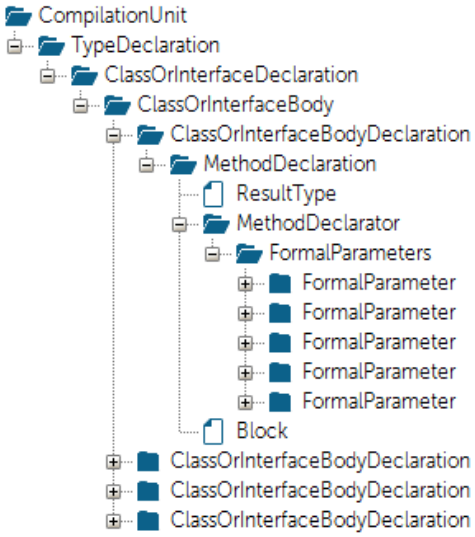
> **Rule description**
>
> When developing java source code, there should not be defined methods with more than four parameters.

First of all, you will need an example of a violation of the convention set for the chosen programming language. This is usually a short example of how to code against the rule. You will need the same example with fixed violations as well. Once you have both code fragments, you can parse them using Kiuwan Rule Developer and then explore the AST corresponding to the code. For this example, we will condense a violation and a fix in the same Java class:

Java source code

```
public class ExcessiveParametersSample {

  // VIOLATION
  public void methodWithMoreThanFour(String first, String second, String
third, String fourth, String fifth) { }

  // OK
  public void methodWithFour(String first, String second, String third,
String fourth) { }

  // OK
  public boolean methodWithLessThanFour (String first, String second) { }

  // OK
  public int methodWithNoParams () { }
}
```

You can copy and paste this code into the **Test Source Code** tab in Kiuwan Rule Developer and click **Generate AST**. An **Abstract Syntax Tree** for the java class will be created:



When you have the AST, you can develop the best approach to traverse the nodes in order to find the way to determine whether there is a violation of the convention in the code or not. For that, there is a basic, very useful way to travel to each one of the nodes: the **visitor** strategy.

Visitor strategy

```
import com.als.core.ast.TreeNode;
import com.als.core.ast.NodeVisitor;

public class MyDummyRule extends AbstractRule {
  protected void visit(BaseNode root, final RuleContext ctx) {
        // this 'visit' is executed on each one of the source code files
under analysis
    TreeNode.on(root).accept(new NodeVisitor() {
      public void visit(BaseNode node) {
                // this 'visit' is executed on each one of the nodes in
the AST of the current file under analysis
        // ...
      }
    });
  }
}
```

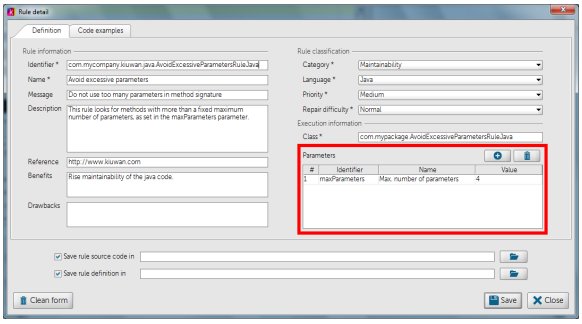
So, applying this strategy to our rule implementation is as simple as:

My first rule

```
import com.als.core.AbstractRule;
import com.als.core.RuleContext;
import com.als.core.RuleViolation;
import com.als.core.ast.BaseNode;
import com.als.core.ast.NodeVisitor;
import com.als.core.ast.TreeNode;

public class AvoidExcessiveParametersRuleJava extends AbstractRule {

  protected void visit(BaseNode root, final RuleContext ctx) {
    TreeNode.on(root).accept(new NodeVisitor() {
      public void visit(BaseNode node) {

            // search for "MethodDeclaration" nodes in the AST
        if (!node.isTypeName("MethodDeclaration")) return;

        // at this point, we are in a "MethodDeclaration" node
        // recover the "FormalParameters" node under the
"MethodDeclaration\MethodDeclarator" path
        TreeNode formalParameters = TreeNode.on(node).child
("MethodDeclarator").child("FormalParameters");

        // if there is more than 4 parameters defined, we should report a
violation
        if (formalParameters.isNotNull() &amp;amp;&amp;amp; formalParameters.
getNumChildren() > 4) {
          RuleViolation violation = createRuleViolation(ctx,
formalParameters.findLine());
          ctx.getReport().addRuleViolation(violation);
            }
          }
        });
  }
}
```

But what happens if we want to change the maximum number of parameters allowed in a method? Would we have to recompile the rule? No! We only need to add a parameter to the rule.

The parameter has to be defined in the rule definition (see Creating a rule in Kiuwan Rule Developer Quick start guide), so you can retrieve the value in your rule by using the initialize method.



My first rule with parameters

```
import com.als.core.AbstractRule;
import com.als.core.RuleContext;
import com.als.core.RuleViolation;
import com.als.core.ast.BaseNode;
import com.als.core.ast.NodeVisitor;
import com.als.core.ast.TreeNode;

public class AvoidExcessiveParametersRuleJava extends AbstractRule {

  private static final int DEFAULT_MAX_PARAMETERS = 3;
  private int maxParameters;

  public void initialize(RuleContext ctx) {
    super.initialize(ctx);
    // 'maxParameters' is the exact paremeter identifier created in the
rule definition
    maxParameters = getProperty("maxParameters", DEFAULT_MAX_PARAMETERS);
  }

  protected void visit(BaseNode root, final RuleContext ctx) {
    TreeNode.on(root).accept(new NodeVisitor() {
      public void visit(BaseNode node) {
        if (!node.isTypeName("MethodDeclaration")) return;
        TreeNode formalParameters = TreeNode.on(node).child
("MethodDeclarator").child("FormalParameters");

        // if there is more than 'maxParameters ' parameters defined, we
should report a violation
        if (formalParameters.isNotNull() &amp;amp;&amp;amp; formalParameters.
getNumChildren() > maxParameters) {
          RuleViolation violation = createRuleViolation(ctx,
formalParameters.findLine());
          ctx.getReport().addRuleViolation(violation);
            }
          }
        });
  }
}
```

You can define parameters of different types, as well as define parameters with a regular expression as value or even with a list of possible values.

Parameters definition

```
import com.als.core.util.StringUtil;
import java.util.regex.Pattern;

public class DummyRule extends AbstractRule {

  private static final String DEFAULT_STRING_PARAMETER_VALUE =
"defaultStringParameterValue";
  private String stringParameter;

  private static final int DEFAULT_INT_PARAMETER_VALUE = 0;
  private int intParameter;

  private static final boolean DEFAULT_BOOLEAN_PARAMETER_VALUE = true;
  private boolean booleanParameter;

  private final static String DEFAULT_PATTERN = "^.*Bean$";
  private Pattern patternProp;

  private static final String DEFAULT_LIST_PARAMETER_VALUE = "value1,
value2, value3, value4";
  private Set<String> stringParameterAsSet = new HashSet<String>(4);
  private List<String> stringParameterAsList = new ArrayList<String>(4);

  public void initialize(RuleContext ctx) {
    super.initialize(ctx);
    this.stringParameter = getProperty("stringParameter",
DEFAULT_STRING_PARAMETER_VALUE);
    this.intParameter = getProperty("intParameter",
DEFAULT_INT_PARAMETER_VALUE);
    this.booleanParameter = getProperty("booleanParameter",
DEFAULT_BOOLEAN_PARAMETER_VALUE);

    String patternStr = getProperty("patternProp", DEFAULT_PATTERN);
    this.patternProp = Pattern.compile(patternStr);

    String strVar = getProperty("stringListParameter",
DEFAULT_LIST_PARAMETER_VALUE);
    this.stringParameterAsList = StringUtil.asList(strVar, ',');
    this.stringParameterAsSet = StringUtil.asSet(strVar, ',');
  }

  public void visit(BaseNode root, final RuleContext ctx) {
    //...
  }
}
```

A word about performance

The rules of thumb to optimize rules:

- Minimize the number of queries to the AST.
- Get the nodes you need and only the ones you need whenever possible.
- Avoid caching AST nodes. When you cache a node you are caching the whole tree.
- DO NOT use global variables (fields)*. Use local variables instead. Rules have to be **thread-safe**!!
- Take advantage of *initialize*() and *postProcess*() methods for unique operations like obtaining the rule's properties. Both methods are executed just once per rule analysis execution, while *visit* () is executed once per analyzed file.
- Avoid using "*TreeNode.findXXX*" and other methods that produce a list of nodes. Instead of that use the visitor strategy.

(*) Except for representing the properties of the rule. In this case, the value is the same during the rule execution, so you won't have concurrency problems.