Create new Kiuwan Rules

This guide shows you how to create new rules for your Kiuwan Analysis without any coding.

If you are a developer, please visit Rule development

Contents:

Two ways of extending the models with new rules without coding:

Split the behavior

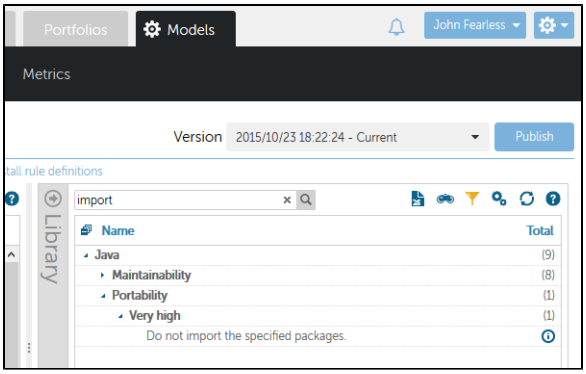
The first method is basically to create a new rule based on an existing one in your library. Let me go directly to an example.

You can find the following rule definition in your Kiuwan library (Models management – Rules – Library):

Do not import the specified packages.

Do not use the classes included in the specified packages, so you will prevent the use of dependent classes of the virtual machine or other private classes. The comma-separated list of patterns of packages is parameterized.

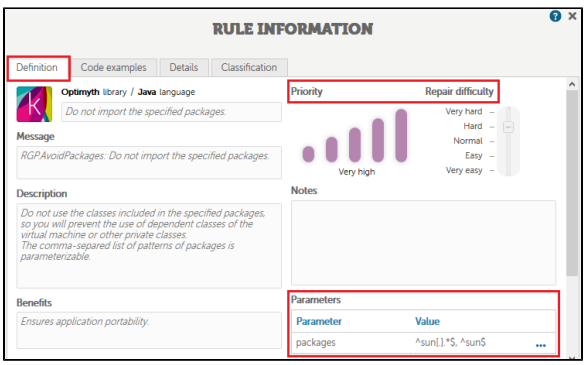
This rule is classified under 'Portability' and priority 'Very High'.

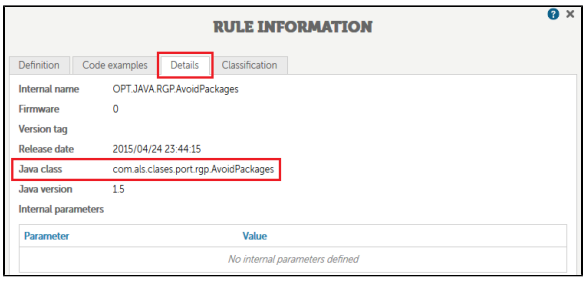


This is a very general rule. You can create new rules to not allow to import specific packages like:

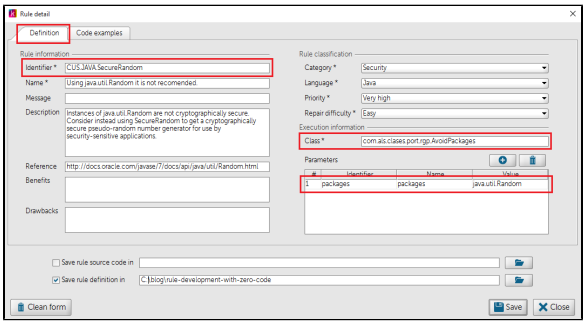
- 'com.sun.*' or 'com.ibm.*' packages may be defined as a portability issue (http://stackoverflow.com/questions/8565708/what-is-inside-com-sun-package).
- Use java concurrent collections instead java collections as a reliability issue (https://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html).
- Replace java.util.Random with the more secure java.security.SecureRandom. (http://docs.oracle.com/javase/7/docs/api/java/util/Random.html, http://stackoverflow.com/questions/11051205/difference-between-java-util-random-and-java-security-securerandom).

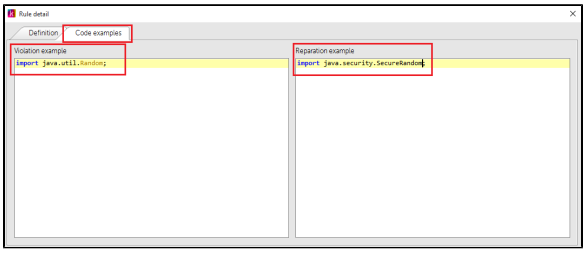
Based on this rule, you can write new rules simply by changing the **packages** parameter of the rule. Let see the details of this rule:





The first step is to open the Rule Developer distributed with Kiuwan Local Analyzer, and create a **New** rule:

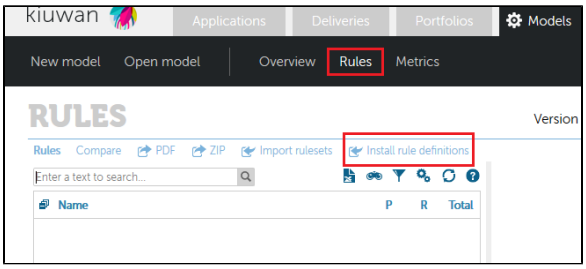


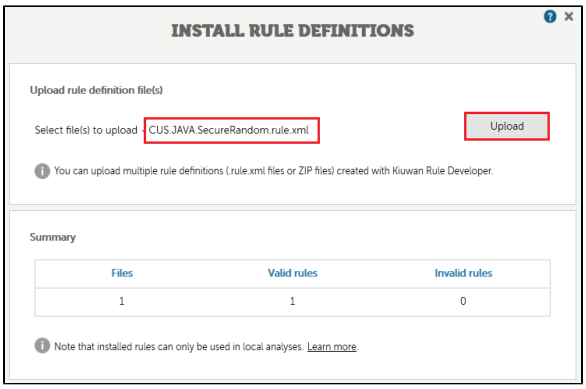


A couple of things to remember:

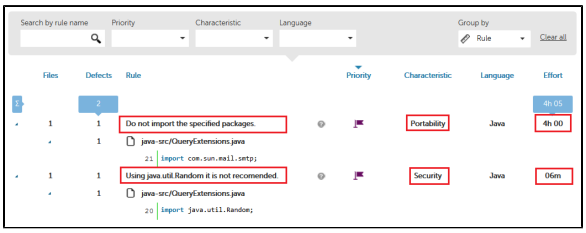
1. Rule identifiers must begin with CUS.
2. The class name has to be the same as the original Kiuwan rule.
3. Parameters are the same as the original rule. Update only the parameter value to the required value.
4. It is advisable to change the name and description of the rule to match the new rule behavior.

Once you save the rule, the Rule Developer will automatically create an XML file with the new rule definition. The second step is to import this XML file definition in your library and model (Models management > Rules > Install rule definitions):





And finally, assign this model to your application and run an analysis on it. You can see the new defects found by the rule in the defects screen:



Rule to have violations based on metric values

When you analyze your code, Kiuwan returns three types of information:

1. Indicators. These are complex metrics calculated from the defects and simple metrics found in your code. Some examples: Risk Index, Global Indicator, Effort, etc.
2. Defects. These are all breaches of the rules in your model.
3. Metrics. These are measurements of some characteristics of the source code. Examples are lines of code (LOC), Cyclomatic Completixy (https://en.wikipedia.org/wiki/Cyclomatic_complexity), Fan-out, etc.

Not all metrics provided by Kiuwan are taken into account for the indicators. For example, you can get the CBO for your java classes, but their values do not modify the Global Indicator.

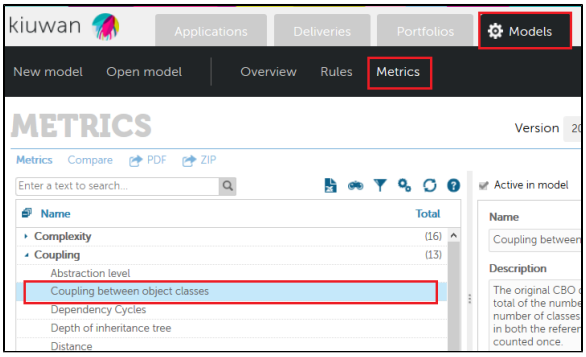
To achieve this, it would be interesting to create a new defect each time the metric threshold is breached.

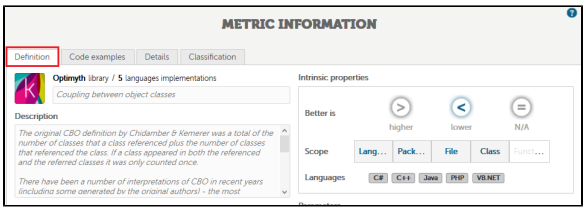
Kiuwan uses this idea in several of its rules, like the duplicated code, the cyclomatic complexity, etc.

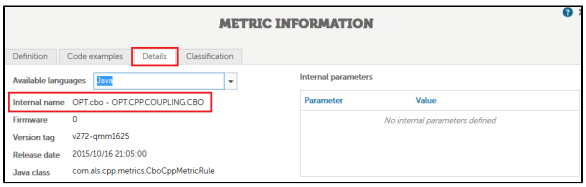
Now we are going to explain how to extend this mechanism to any metric calculated by Kiuwan.

Example 1. CBO in java.

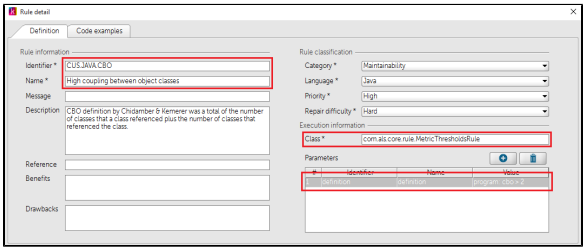
In Kiuwan's metrics library (Models management – Metrics – Library) you can find the following definition for the CBO metric:







You can now create a new rule definition with the Rule Developer that will check the value of the CBO metric against a threshold:



Remember:

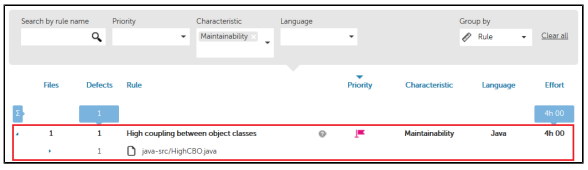
1. Rule identifiers must begin with CUS.
2. Class name has to always be: *com.als.core.rule.MetricThresholdsRule*.
3. You need to add a parameter, called definition, with the expression to check.
4. It is advisable to define a name and a description for the rule that explains the rule behavior for the specific metric, i.e. High

In this example, the expression is:

program: CBO > 2

> **IMPORTANT:** white space is mandatory between all operators and operands in the expression.

Once the rule is installed in your model, and a new analysis is run, you will get defects like the following:



Notice the first word in the expression, **program**. It indicates the scope for the values of the metric to use. Different metrics may have different scopes to calculate the metrics, as we will see in the next example.

Example 2. Ccn at function scope.

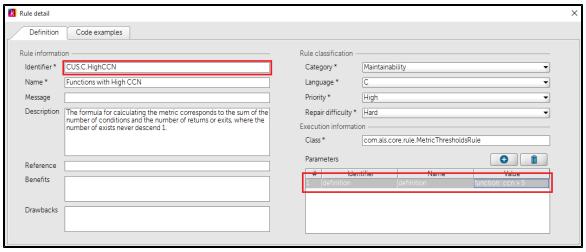
Let's see how to create a new rule-based in a metric with function scope.

> **Cyclomatic complexity**: The formula for calculating the metric corresponds to the sum of the number of conditions and the number of returns or exits, where the number of exits is never lower than 1.

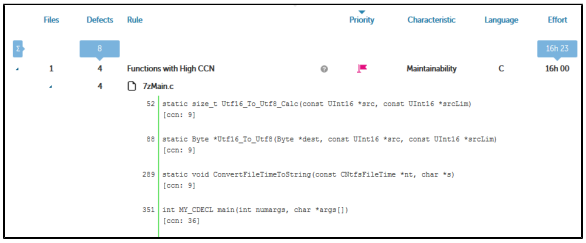
For the cyclomatic complexity you can define the following expression:

function: ccn > 5

So, the definition of the new rule will be:



Import this new rule in your model (as explained in the case above), run a new analysis, and you will get this type of defect when the Cyclomatic Complexity of a method (in C, for example) is greater than 5:



Grammar for expressions

In the above examples, we saw that the **definition** parameter value for 'com.als.core.rule. MetricThresholdsRule' class defines the expression to check. The expected format is:

*scope: expression (;scope: expression)**

'Scope' has to be one of these values:

- **program** (a.k.a. file)
- **type** (depends on the language. Valid values are: class, interface, struct, enum)
- **function** (functions or methods)

The expression also support the following operators:

- "+","-","*","/" (arithmetic operators)
- "and" and "or" (logical operators)
- ">", "<", ">=", "<=", "==", "!=" (comparison operators)

The metric name you use in the expression is the metric short name. For example, for the metric internal name: OPT.cbo, the metric short name is: cbo.

You can find the metric internal name in the details tab of the metric information window (see above captures).

This type of rules will report a violation when the defined expression evaluates to true.