# Tutorial: Implementing a Groovy-based rule engine

In modern application design and especially service orientied architecture (SOA) there is often the question how to model and implement frequently changing business logic. Being able to do that is key to achieve agile solutions, which are closely aligned to the business.

For many the answer is to use rule engines, because they allow to separate business logic from application logic and provide domain specific languages (DSL) to the end user.

Using DSL is a great step towards agility and improved Business/IT-Aligment. Several open source and commercial rule engines are available, but they are often complex and difficult to use.

The basic principle of rule engines is quite simple. It is all about conditions and actions which are called in a certain order.

In this article we will implement a basic rule engine with DSL support in Groovy.

Why Groovy? Because of its dynamic features it is the ideal choice for those kinds of applications. It integrates perfectly with the Java runtime as the compiler emits hundred percent Java byte code.

Prerequisites:

1. Java Runtime environment (http://www.java.com)
2. Groovy installation (http://groovy.codehaus.org/)
3. Microsoft Excel

To begin let us define a business object that should be manipulated by rules.

```
class Developer{
  private String name
  private int experience
  private String level="unknown"
}
```

A typical rule would look like this:

```
if dev.experience > 1 && dev.experience < 3 then dev.level= "Junior"
```

It determines the title based on the years of working experience. But wait this is hard coded, so let us modularize it.

```
// Conditions
def c1 = {dev.experience>1}
def c2 = {dev.experience<3}

// Actions
def a1 = {dev.level="Junior"}

if c1 && c2 then a1
```

The above listing uses closures to express the conditions and actions. A closure can be seen as pointer to a function which can be called using this pointer.

The closure returns the result of the last executed action. Therefore

```
 def c1={dev.experience>1}
```

would be equivalent to

```
 def c1={return dev.experience>1}
```

For the sake of convenience we leave the return out.

For our rule engine the parameters have to be defined outside of the closures in order to separate them from the actual rule.

```
def c1={dev, experience -> dev.experience > experience}
def c2={dev, experience -> dev.experience < experience}
def a1={dev, level -> dev.level = level}

def developer = new Developer(name:"Peter",experience:2)
if c1(developer,1) && c2(developer,3) then a1(developer,"Junior")
```

The types of the closure parameters can be left out. They will be determined at runtime. The parameter list is specified on the left hand side of ->. The actual functionality is specified on the right hand side.

A rule engine allows to define multiple conditions and actions, so it is time to come up with the engine.

First of all the classes representing a rules are needed. The class Rule represents a single rule. Multiple rules are combined in a RuleSet.

```
class Rule{
  private boolean singlehit = true
  private conditions = new ArrayList()
  private actions = new ArrayList()
  private parameters = new ArrayList()
}

class RuleSet{
  private rules = new ArrayList()
}
```

As you can see in listing above we have three ArrayLists for the conditions actions and parameters. They will all contain closures to be executed by the rule engine.

The singlehit field affects the runtime behaviour of the rule engine. Single hit means that the engine stops processing rules once all conditions of a single rule return true with a given parameter set. If single hit is false then all parameter sets are processed.

The next listing shows the actual rule engine. It just has one method run which is used to process an arbitrary business object using a given rule set.

```
class RulesEngine{

 def run(RuleSet ruleset, Object bo){

    // Iterate over the rules
    ruleset.rules.each{ rule ->
       println "Executing rule in "  +
       (rule.singlehit?"singlehit":"multihit") + " mode."
       def exit=false // Exit flag for singlehit mode

       // Iterate over the parameter sets
       rule.parameters.each{ArrayList params ->
          def pcounter=0 // Points to the current parameter
          def success=true

          if(!exit){
             // Check all conditions
             rule.conditions.each{
               success = success && it(bo,params[pcounter])
               pcounter++
             }

             // If all conditions true, perform actions
             if(success && !exit){
                rule.actions.each{
                   it(bo,params[pcounter])
                   pcounter++
                }
                // If single hit, exit after first condition match
                if (rule.singlehit){
                  exit=true
                }
             }
          }
       }
    }
 }
}
```

First the run method  iterates through the rules. For each rule it iterates through all parameters and carries out the conditions with each parameter set. If all conditions are true it carries out all action closures. If singlehit is set to true, it stops after the first parameter set in causes all conditions to result true.

Now let us come up with a complete rule.

```
def addRule(RuleSet ruleset){

      // Rule definition
      def rule = new Rule()
      // ****************** CONFIGURATION ******************
      rule.singlehit=true
      // ***************************************************
      rule.conditions=[
      // ****************** CONDITIONS *********************
      {bo, p -> bo.experience>p},{bo, p -> bo.experience<=p}
      // ***************************************************
      ]
      rule.actions=[
      // ******************* ACTIONS ***********************
      {bo, p -> bo.level=p}
      // ***************************************************
      ]
      rule.parameters=[
      // **************** PARAMETERSETS ********************
      // Min experience, Max experience, Level
      [1,3,'Beginner'],
      [1,3,'Starter'],
      [4,6,'Junior'],
      [7,10,'Average'],
      [11,20,'Senior']
      // ***************************************************
      ]
      ruleset.rules<<rule
}
```

It is time to perform a first test.

```
// Business object creation
def dev = new Developer(name:"Peter",experience:2)
println "Before:" + dev.dump()

// Run the rules
def ruleset = new RuleSet()
addRule(ruleset)
def engine = new RulesEngine()
engine.run(ruleset,dev)

// Show result
println "After:" + dev.dump()
```

This application will dump out the following on the console:

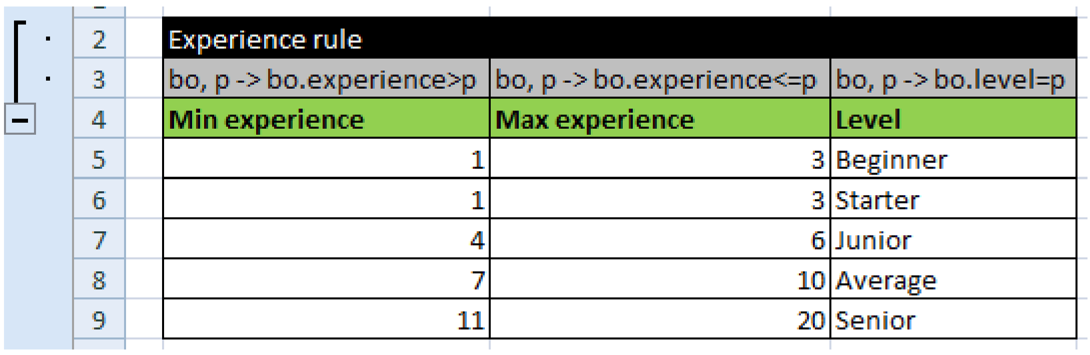
```
Before:<Developer@1876e5d name=Peter experience=2 level=unknown>
Executing rule in singlehit mode.
After:<Developer@1876e5d name=Peter experience=2 level=Beginner>
```

As you can see the level of the developer has been set based on rules. Because singlehit is set to true the level is *Beginner*. If it was set to false it would be *Starter*.

Although for a developer the rule definition inside of addRule is quite self-explanatory, a business user might not understand it.

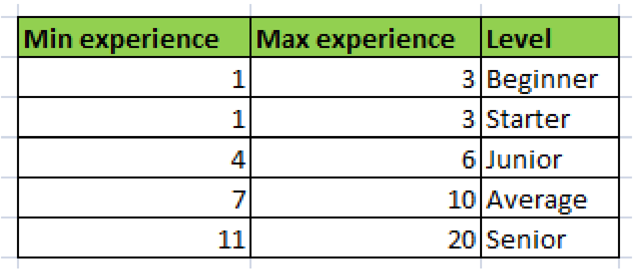
So it is time for some user friendliness in form of a domain specific language (DSL).

One of the tools that most business users know is Microsoft Excel. So this is what we give them.



| | | | |
|---|---|---|---|
| 2 | Experience rule | | |
| 3 | bo, p -> bo.experience>p | bo, p -> bo.experience<=p | bo, p -> bo.level=p |
| 4 | **Min experience** | **Max experience** | **Level** |
| 5 | 1 | 3 | Beginner |
| 6 | 1 | 3 | Starter |
| 7 | 4 | 6 | Junior |
| 8 | 7 | 10 | Average |
| 9 | 11 | 20 | Senior |

The first two rows are meaningful for the rules developer, as they contain the conditions and actions. In order to hide them from the business users they are grouped. By clicking on the minus they can be hidden, providing exactly what the business user needs.



| **Min experience** | **Max experience** | **Level** |
|---|---|---|
| 1 | 3 | Beginner |
| 1 | 3 | Starter |
| 4 | 6 | Junior |
| 7 | 10 | Average |
| 11 | 20 | Senior |

Now we have to read the rule from Excel. Thanks to the Scriptom library this is an easy task in Groovy.

```
def addRuleFromExcel(ruleset, filename){
    // Start excel
    def excel = new ActiveXObject('Excel.Application')
    def workbook = excel.Workbooks.Open(
                   new File(fme).canonicalPath)
    def sheet = workbook.ActiveSheet

    // Create rule
    def rule = new Rule()

    // Read conditions
    ['B3','C3'].each{
      rule.conditions << Eval.me("{ ${sheet.Range(it).Value}}")
    }

    // Read actions
    ['D3'].each{
      rule.actions << Eval.me("{ ${sheet.Range(it).Value}}")
    }

    // Read parameters
    for (l in 5..9){ // lines
        def set = []
        for (c in 'B'..'D'){// columns
           def cell = c+l
           set << sheet.Range(c+l).Value
        }
        rule.parameters << set
    }

    // Release excel
    workbook.Close()
    excel.Quit()

    // Assign rule to the ruleset
    ruleset.rules<<rule
}
```

Scriptom is a Java-COM-Bridge which allows to access COM-Objects via an automation interface. The code should be self-explanatory. In order to test it we have to replace the line

```
addRule(ruleset)
```

with

```
addRuleFromExcel(ruleset)
```

All Microsoft applications have automation interfaces, so we can programmatically access them. Please note that this this is a Windows only solution. If portability is required, Apache POI (http://poi.apache.org/) can be used instead.

This article explained the principles of rule engines and provided a prototypical implementation based on Groovy. The source code can be found here:

grules.groovy, grules.xls

**About the author:**

Wolfgang Pleus is an independent IT consultant in the area of service-oriented architecture (SOA). His primary interest is the implementation of SOA concepts at the application-, architecture- and strategy level and is driven by the question how to realize the SOA agility promise with state of the art technologies. He is supporting mission critical enterprise projects for more than 15 years. As author, speaker and trainer he regularly shares his experience nationally and internationally.

Contact: wolfgang.pleus@pleus.net