



# Creating a DSL

## Using The Power of Groovy



Frank Pijpers

September 2009

# What is Groovy?

- A dynamic language for the JVM
- Inspired by Ruby, Python, Smalltalk
- Generates byte code for the JVM:
  - integrates natively with Java libraries
- Differentiates
  - GDK: additional libraries to simplify complex APIs
  - MOP: advanced meta-programming features

# Who is Groovy?

- Guillaume Laforge
  - Groovy Project Manager at Codehaus
  - JSR-241 Spec Lead
  - Initiator of the Grails Web framework
  - Co-author of « Groovy in Action » – Manning
  - Software Architect at OCTO Technology
- John Wilson
  - Former Groovy core committer
  - Groovy metaprogramming specialist
  - Owner of the Wilson Partnership

# Why is Groovy?

- 3 Use Cases
  - Prototyping / testing / scripting
  - Building standalone applications
  - Integrating scripting in JEE™ applications

# What is a DSL?

- DSL:
  - a Domain-Specific Language
  - also called a *little language*
- Wikipedia:
  - a Domain-Specific Language is a programming language
  - designed to be useful for a specific set of tasks
- A DSL is a language that closely models a certain domain of knowledge or expertise, where the concepts are tied to the constructs of the language.

- A domain specific language
  - Is a language
  - Covers a **particular domain of knowledge**
  - Has a form
    - Textual
    - Graphical
  - Produces a result
    - Configures objects
    - Represents data structures
    - Is a notation

- Internal or External
  - Embedded in a host language
  - Or standalone (usually a custom parser)
- Has certain quality attributes
  - Writability
  - Usability
  - Testability
  - Extensibility

- **Technical**
  - SQL, HTML, regex...
  - `SELECT * FROM USER WHERE NAME LIKE 'Guil%'`
  - `^[\w-\.\.]+@([\w-]+\.\.)+[\w-]{2,4}$`
- **Notation:** EBNF, Chess, Music, Rubik's cube
  - 1. e4 e5 2. Nf3 Nc6 3. Bb5 a6.
  - U R' U2 R U R' U R
- **Business:**
  - Life insurance policy
  - Bank accounting rules
  - Invoice description language
  - Hedge funds risk computation



# DSL – Reasons for Creating

- Use a **more expressive** language than a general programming language
- Share a **common metaphor** between developers and domain experts
- Have **domain experts help** design the business logic of an application
- **Avoid cluttering business** code with too much boilerplate technical code
- Cleanly **separate business logic** from application code

# Groovy & DSL?

- **Why Groovy?**
  - It allows you to create Embedded Domain-Specific Languages
  - Those internal DSLs can be embedded in your Java applications
  - Groovy lets you use more complex constructs when you face the limits of your DSL
- **Why not a custom lexer/parser?**
  - A lexer/parser is complex to implement, maintain and use
  - It is harder to evolve

# DSL – Closing Arguments

- DSLs are a great tool for **sharing a common metaphor** between experts and developers
- DSLs express the domain concepts **without the usual boilerplate code**
- Groovy's malleable syntax makes it the **perfect choice** for creating DSLs and integrating them in a Java application

- Prerequisites:
  - Java Runtime environment  
(<http://www.java.com>)
  - Groovy installation  
(<http://groovy.codehaus.org/>)
  - 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"  
}
```

# Typical Rule

- `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`

# Defining Parameters

- 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")`

# Rules & Rule Sets

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

# Rules Engine

- `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 } } } } }`

# 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
  ▪ }
```

# A 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>

# User Friendly User Interface

Experience Rule		
<code>bo,p -&gt; bo.experience &gt;p</code>	<code>bo,p -&gt;bo.experience&lt;=p</code>	<code>bo,p-&gt;bo.level=p</code>
Min experience	Max experience	Level
1		3 Beginner
1		3 Starter
4		6 Junior
7		10 Average
11		20 Senior

# Execute

- ```
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 (1 in 5..9){ // lines def set = [] for  
    (c in 'B'..'D'){// columns def cell = c+1 set <<  
    sheet.Range(c+1).Value } rule.parameters << set } // Release  
    excel workbook.Close() excel.Quit()  
  ▪ // Assign rule to the ruleset  
  ▪ ruleset.rules<<rule }
```



Source of your development.

01010100101010101011110101000111  
101001010113010001010010100101  
11101101010101010101010101010  
101010101010101010101010101010  
001010101010101010101010101010  
101010101010101010101010101010



embeddedsystems@sioux.nl

+31 (0)40 26 77 100