GROOVY AS A SWISS KNIFE - FROM ENTERPRISE TO SCIENCE

Josef Smolka
Faculty of Nuclear Sciences and Physical Engineering CTU in Prague
smolkjos@fjfi.cvut.cz

ABSTRACT:
The paper introduces the Groovy language as a multi-purpose tool applicable to numerous fields of software engineering, ranging from enterprise projects to specialized scientific tools. In addition, Groovy can also play a role of introductory language to object-oriented programming instead of nowadays popular Java.

KEYWORDS:
Java, Groovy, Grails, Domain-Specific Language

1 INTRODUCTION

August 29 of year 2003 is considered to be the birthday of a new dynamically typed language on the Java platform, called Groovy. The language received the name because it is built on top of all the groovy parts of a Java code [1]. Groovy was created with intentions to deliver a new scripting language to the Java platform, but nowadays, it is seriously competing with the main platform language, the Java itself [2]. In 2003, James Strachan, the author of Groovy, was playing with dynamic languages of that time, Python and Ruby, and with their respective JVM versions, Jython and JRuby, and felt a necessity to complement the overall complex (but lacking core functionalities at the same time) and sometimes problematic original language of Java platform with something new [1][2]. Dynamic languages like Python, Ruby and even Javascript enjoy great popularity by present developers, but transition to the Java platform was not so straightforward and JVM implementations of those languages suffer from it. This was the main motivation for designing new language from the ground, which would resemble Java in core syntax a compile directly to the Java byte code. Since 2003, Groovy has undergone a rough path. At the turn of the year 2003 and 2004, Groovy was on the brink of failure and was written off by many people [2]. Despite the expectations, Groovy stood the ground and at the beginning of 2007, Groovy version 1.0 was released.

1.1 The Clouds over the Java Language

The Java language itself failed to brink demanded features, like closures, type inference, mix-ins, list literals, tuples, multiple return values and others even in the recent version 7, released in 2011, and one begins to ask why Java community is so conservative (for example in comparison to C# and the whole .NET platform, which is also an enterprise platform). If the trend continues, the Java language (not the whole platform) could be soon displaced by another language as Groovy or Scala [2][3]. This paper is not meant as a criticism of Java, but as an example of how mentioned languages are gradually replacing Java in numerous fields.

1.2 A Brief Introduction to the Groovy Language

The main advantage of Groovy is practically identical syntax with the Java language. To understand examples presented in this paper, basic knowledge of Java is required. A new syntax in Groovy is introduced only in cases when there is no direct opposite in Java. For example, closures are the case. A simple “Hello World” closure in Groovy can be defined as:

```
def hello = { println "Hello World!" }
```
Of course, closures in Groovy can be parameterized. Parameters of the closure are listed before the -> operator:

```
def min = {a, b -> a < b ? a : b}
```

Closures can have so-called free variables, which are not listed in parameters list and are bound to variables of a context, where is the closure defined. There is also an implicit variable *it*, which corresponds to the first argument (if omitted). The main advantage is in use of the closure as the last method argument. For this case special method call syntax is introduced. Many standard collection methods take a closure as the last argument:

```
def numbers = [1, 5, 8, 2, 3, 10]
assert numbers.collect{a -> a + 1 } == [2, 6, 9, 3, 4, 11]
```

In this example, the *collect* method, which takes a closure as the last argument, is used to transform collection of numbers. Notice also a use of a collection literal. Other features of Groovy will be demonstrated further in the paper.

2 GROOVY AS A CORE OF J2EE PROJECTS

The vast majority of J2EE (Java Enterprise Edition) projects are still written in the pure Java language, as Java is the main language of the platform and enjoys considerable attention from framework, library and IDE developers and, first of all, support from such a colossus such as Oracle corporation is. The J2EE standard, in essence, is so tremendous and general that is practically impossible for common developers to utilize standard implementations in JDK (Java Development Kit). This fact made a space for many frameworks to emerge and to fill in the gap between the platform and a developer. Nevertheless, even if the developer utilizes such a framework, there is still a gap between needs of the developer and facilities that the core language provides. The gap has to be overcome by additional libraries, like Apache Commons, which have been substituting missing features in Java for many years.

2.1 Grails Rapid Development Framework

One of the most acclaimed application development frameworks for J2EE is, without debate, Spring. However, even Spring is suffering from design flaws in Java language. Let us imagine that the Spring framework is mixed with Groovy and a well-known concept called convention over configuration (or coding by convention). The result is called Grails – rapid, dynamic and robust application development framework for the Java platform, in which developer does not have to write single line of Java code (if omitting the fact, that the syntax of Groovy and Java is identical in many cases).

2.2 Grails Example

Let us consider following scenario. A tiny web application that should consume a specific web service has to be developed. The application should ask a user for some filter conditions, retrieve a list of objects from the web service, filter the list and present it back to the user. This is a quite common use-case in nowadays information systems. A core of a plain J2EE implementation would be a HTTP servlet, registered in the web.xml configuration file, which would take parameters from the HTTP request (filter inputs) and do the specified logic. As the service in the model situation is not backed up by any database system, the most tedious part to implement is a transformation of the received collection, although a collection of objects manipulation is an integral part of OOP. Let us take a look how Grails and Groovy deal with the described model scenario.

First of all, a domain object (a logical container for domain information) representing an item in the received collection has to be defined. For simplicity, the object has only two properties and is defined as follows:

```
class Item {
  String name
  String key
}
```

Notice that in Groovy, beans are defined with much more simpler syntax. A field with no access modifier is considered to be a private property with implicit public getter and setter. A default bean constructor, which takes map as an argument is also worth mentioning. It is possible to instantiate previously defined Item bean with the code: new Item(name: 'it1', key: 'abc'). Next, a bean representing a user input (a command object) is defined:

```
Class FilterCommand {
  String name
  String key

  static constraints = {
    key (matches: '[a-zA-Z]*')
  }
}
```

An instance of this bean is automatically created by a controller, which will be defined later (controller as in model-view-controller design pattern). It is true, that the domain object could also be used as a command object as well in the scenario, but it is not recommended mainly because of security concerns. The definition of command and domain beans can also contains validation rules as shown in the definition of the *FilterCommand* bean. So far, only the data were addressed, but no application logic. In Spring, as well as in Grails, it is a good manner, to implement an application logic in so-called service classes. The service class for the presented scenario could looks like this:

```
class ItemService {
  static transactional = false

  def getItems = {FilterCommand cmd ->
    def list = callService()
    return list.findAll{it.name == cmd.name && it.key == cmd.key}
  }
}
```

The service is marked as non-transactional, as it is not backed up by any transactional database. The only implemented logic is a filtration of the received collection (done by the closure). For a controller, handling a HTTP request, there is no substantial work to do, just to delegate the work.

```
class FilterController {
  def itemService
  def filter = { FilterCommand cmd ->
    render(view: 'list', model: [items: itemService.getItems(cmd)])
  }
}
```

Dependency injection of an *ItemService* instance is done automatically by the Spring's inversion of control container, which is integrated into Grails.

To sum it up, the main benefit of Grails is the expression power. The combination of Groovy with the Spring Web MVC framework and convention over configuration concept results in a development tool that allows a developer to concentrate more on the problem domain and not on the implementation itself.

3 GROOVY AS A DOMAIN-SPECIFIC LANGUAGE

Developers, who are not yet prepared to abandon Java as their primary language, often use Groovy as DSL (Domain-Specific Language). For example, in Oracle ADF, Groovy is used as an expression language for definition of transient attributes [4]. An application data model in ADF is described by XML metadata, which are used by an object-relational mapping framework to work with relational databases. Defined entities and views can have transient attributes that have no direct mapping to a database. Those attributes are defined with Groovy and are evaluated at runtime (after SQL query execution). For example, consider a classic „employee – department" schema with a one-to-many association. To sum salaries of all active employees in a department, one can define a transient attribute with following Groovy expression:

```
employees.sum('active ? salary : 0')
```

The *employees* variable is an accessor for the one-to-many association. The string argument of the sum method is in fact a Groovy expression as well, which is evaluated for every instance of the *Employee* entity participating in the association. The same result can be, of course, achieved with the SQL query itself, but there are situations in which the dynamic behavior (dynamic in a way that it is not necessary to execute the SQL query) is preferred.

Groovy in Oracle ADF plays only a minor part. In following paragraphs, I would like to mention more complex applications of Groovy as DSL.

3.1 Using Groovy to Promote a Declarative Style of Programming

During development of an enterprise information system for an electricity company, Groovy was employed as an expression language to support a hierarchical metadata system. Common use-cases, such as data input and presentation, in the system are described with a tree graph [5]. There are two kinds of nodes in the graph:

- control nodes, which resemble three basic structures of structural programming – sequence, selection and iteration,
- and specialized nodes describing computational rules, input wizards, validation rules, exports and reports.

Every node has set of properties, which are, in fact, Groovy expressions. Those expressions state conditions of processing, rendering, resulting content and so on. Since the Groovy is very similar to Java in syntax, developers had no problem with the adaptation to a new way of business logic implementation. The use of lambda expressions together with the Groovy collection API greatly influenced the efficiency of a development process [5].

Oracle ADF uses a similar approach. Whole web application is described by tree of components (task-flow, user interface, data model). Properties of these components are defined as Groovy expressions. On the other hand, the application logic is still written in pure Java.

3.2 Groovy among Bacteria

Simulation of a single bacterial colony is a quite common task in the field of software engineering. During the research of patterning of mutually interacting bacterial bodies, special simulator software was designed and implemented. A core of the simulator consists of a simple in-memory object database system with spatial indexing, object pools and a query language based on Groovy [6].

What experiments are simulated? Research shows, that monoclonal bacterial body can be greatly influenced by a presence of another body of the same strain in a way that indicates complex communication capabilities. This phenomenon is called quorum sensing. Goal of the experiments is to determine the mechanism how bacteria react to quorum signals and how multispecies bodies are built [6].

A data model covering needs of such simulation is not complicated and consists of only four main classes:

- Experiment – In relational database, this would represent something like a database schema. It is kind of an umbrella for all other objects in the database.
- Location – The simulator is counting with discrete grid models, so Location represents one cell in a grid.
- Entity – General object with distinguishable attributes and behavior suitable for individual-based simulations.
- Substance – An object proposed for reaction-diffusion simulations.

In result, a model of colony interactions is implemented by sequence of database queries. Proposed database system supports work in a pseudo-transaction context. All database updates are recorded to a special log. The log is applied to real data during commit of transaction. Queries on data are not influenced by the change logged during the transaction. This kind of behavior may seem odd on the first glance, but perfectly fits the nature of work during evaluation of a single step in a colony evolution [6].

Let take a look at signal substance diffusion model and a respective implementation in the simulator.

$$\frac{\partial C}{\partial t} = D\left(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}\right) \quad (1)$$

$$\Delta c_{S_{ij}} = \frac{D\Delta t}{(\Delta x)^2}\left(\sum_{S_{kl}\in O(S_{ij})} c_{S_{kl}} - 4c_{S_{ij}}\right) \quad (2)$$

An equation 1 is a classic diffusion equation, which is transformed into a discrete equivalent (equation 2) on a rectangular grid. The implementation in the simulator looks like this:

```
def S = { L, w ->
  return d.s.loc(L.x+w?.x?:0, L.y+w?.y?:0)?.subst(p.signal) ?: 0.0
}

def SD = { L ->
  def SSR = p.WAYS.sum {w -> S(L, w)}
  def SI = SP * (SSR - 4 * S(L, null))
  L.add(p.signal, SI)
}

d.s.locations.each{L -> SD(L)}
```

The first is a closure for getting signal concentration in the specified direction. Respective location in a grid is obtained with a help of spatial index, which is accessed by the *loc* method of the current space *s* in the database *d*. The location is addressed by x and y coordinates, which are evaluated as a sum of position of the current location *L* and a specified way *w*, which can be null. Note the use of *Safe Navigator Operator ?.* to avoid *NullPointerException* when accessing property on a null object and the *Elvis Operator ?:* to shorten the classic ternary operator if one of the results is null.
The value of the concentration is returned by the *subst* method for the specified substance instance *p.signal*.

The second closure *SD* deals with the signal concentration in the actual location *L*, it is an implementation of the inner part (in parentheses) of the equation 2. The dynamic variable *SSR* is signal concentration in surroundings of the location. The variable *SI* is the signal fluctuation in the location. The last statement is an application of the closure *SD* to all locations in the grid.

The proposed DSL for a description of the experiment and implementation of models in the simulator forms reliable foundation for further research.

4 GROOVY AS AN EDUCATIONAL TOOL

Java is often used as an introductory language to object-oriented programming (OOP), but it is a really good choice? Simple syntax and harmless memory model indicate that it is, moreover, three fundamental principles of OOP can be demonstrated in Java quite well. But then there are faint smells like primitive types, auto-boxing, no operators overloading, no closures, no list literals, and so on [7][8]. Thus, to demonstrate the basics, Java is more than sufficient, but to show the real hearth of OOP, more sophisticated languages like Groovy or Smalltalk are required. In real life applications of OOP, there is sparely only one instance of class present, but there are whole collections of instances. For example, to demonstrate operations like selection and transformation, which are most common operations, when it comes to handling collections, Groovy is a good choice.

```
class Char {
  String name
  String height
}

def clist = [ new Char(name: 'Ferda', height: '12'), new Char(name:
'Brouk', height: '10'), new Char(name: 'Beruška', height: '11')]

assert clist.findAll{c -> c.height > 10}.collect{c -> c.name} ==
['Ferda', 'Beruška']
```

The example code can show potential students of OOP that selection operation on collection is not only theory, but it should be integral part of the language core library to utilize the language efficiently. In groovy, selection is realized via the *findAll* method, which takes closure as the last argument.

5 CONCLUSION

The paper attempted to show that dominant position of Java as the platform main language is not unassailable and that Groovy could be the potential replacement. Example applications of Groovy in the variety of situations were given to support this statement.

6 ACKNOWLEDGEMENT

Creation of this paper was supported by a grant SGS 11/167.

LITERATURE

[1] Strachan, J. *Groovy – The Birth of a New Dynamic Language for the Java Platform.* [cit. 2012-04-01]. Available from WWW: <http://radio-weblogs.com/0112098/2003/08/29.html#a399>.

[2] Devijver, S. *Groovy Will Replace the Java Language as Dominant Language*. [cit. 2012-04-01]. Available from WWW: <http://groovy.dzone.com/news/groovy-will-replace-java-langu>.

[3] Strachan, J. *Scala as the Long Term Replacement For java/javac?* [cit. 2012-03-31]. Available from WWW: <http://macstrac.blogspot.com/2009/04/scala-as-long-term-replacement-for.html>.

[4] Grant, R. *Introduction to Groovy Support in JDeveloper and Oracle ADF 11g*. Oracle Corporation, 2009 [cit. 2012-04-22]. Available from WWW: <http://www.oracle.com/technetwork/developer-tools/jdev/introduction-to-groovy-128837.pdf>.

[5] Klika, D.; Smolka, J. *Application of declarative paradigm in enterprise IS*. In: Tvorba softwaru 2011. Ostrava: VŠB - Technická univerzita Ostrava, 2011.

[6] Smolka, J. *Using Java and Groovy in Simulation of Mutually Interacting Bacterial Bodies*. In: Objekty 2011. Žilina: Žilinská univerzita, 2011.

[7] Batsov, B. *Java.next() - the Groovy Programming Language*. [cit. 2012-04-01]. Available from WWW: <http://batsov.com/articles/2011/05/06/jvm-langs-groovy/>

[8] Iry, J. J*ava Has Type Inference and Refinement Types (But With Strange Restrictions).* [cit. 2012-04-01]. Available from WWW: <http://james-iry.blogspot.com/2009/04/java-has-type-inference-and-refinement.html>