Just Groovy!

Die Skriptsprache Groovy

Marcel Tilly

Die etablierten Skriptsprachen wie Perl, PHP, Python und Ruby bekommen in letzter Zeit vermehrt Konkurrenz. Gerade Skriptsprachen, die mit Java interagieren können, scheinen groß in Mode zu sein. Da gibt es bereits Jython und JRuby, als Java-basierte Varianten von Python und Ruby, aber auch neuere Skriptsprachen wie Nice und Beanshell. Eine Skriptsprache tritt plötzlich aus dem Schatten der anderen heraus: Groovy! Die Macher von Groovy hatten nämlich die großartige Idee im Java Community Process (JCP) einen Java Specification Request (JSR 241) zu beantragen. Groovy soll neben Java eine weitere standardisierte Programmiersprache für die Java Virtual Machine (VM) werden.

Die Idee, die dahinter steht, ist gar nicht mal abwegig, denn Groovy ist in Java implementiert und erzeugt Bytecode, der mit der VM ausgeführt werden kann. Zudem ist es möglich aus einem Groovy-Skript Java-Code aufzurufen und ebenso umgekehrt aus Java-Code Groovy-Code. Natürlich ist die Entwicklung von Groovy noch nicht abgeschlossen (derzeit gibt es Groovy 1.0 beta6) und die Wunschliste ist noch lang, dennoch lohnt sich bereits jetzt ein Blick darauf.

Groovy ermöglicht nicht nur die Interaktion mit Java, sondern vereinigt auch gute Entwurfsideen aus Python, Ruby und Smalltalk. Grundsätzlich gelten für das Design von Groovy folgende Punkte:

- Groovy soll einfach für Java-Entwickler erlernbar sein: Aus diesem Grund ist die Groovy-Syntax auch der Java-Syntax ähnlich und es gilt: Korrekter Java-Code ist auch korrekter Groovy-Code!
- Wenn irgendwie möglich, sollen Groovy-Klassen in „normale" Java-Klassen compiliert werden.
- Die Syntax von Groovy soll möglichst einfach, aber dennoch mächtig sein. Es soll dynamisches und statisches Typisieren möglich sein.

In machen Fällen erweitert Groovy die Standard-Java-Funktionalität um Methoden, die das Arbeiten mit Groovy vereinfachen. Die Vereinfachungen, die in Groovy realisiert sind, führen dazu, dass ein längeres Java-Programm (s. Listing 1) häufig in ein kleineres Groovy-Skript (s. Listing 2) umgesetzt werden kann. Dabei wurden die folgenden Features von Groovy verwendet:

```
public class Filter {
  public static void main( String[] args ) {
    List list = new java.util.ArrayList();
    list.add( "Peter" );
    list.add( "Paul" );
    list.add( "Mary" );
    Filter filter = new Filter();
    List shorts = filter.filterLongerThan( list, 4 )
    for ( String item : shorts ) { System.out.println( item ); }
  }
  public List filterLongerThan( List list, int length ) {
    List result = new ArrayList();
    for ( String item : list ) {
      if ( item.length() <= length ) { result.add( item ); }
    }
    return result;
  }
}
```

Listing 1: Ein Java-Programm

- am Zeilenende muss kein Semikolon stehen,
- spezielle Features zur Verarbeitung von Collections,
- Closures,
- keine explizite Klassendefinition.

Natürlich bietet Groovy noch viel mehr, weshalb sich ein detaillierterer Einstieg in Groovy auf jeden Fall lohnt.

```
list = ["Peter", "Paul", "Mary"]
shorts = list.findAll { it.size() <= 4 }
shorts.each { println it }
```

Listing 2: Das entsprechende Groovy-Skript

Installation

Die Installation von Groovy ist denkbar einfach. Das Archive, das von der Groovy-Website [GRO1] heruntergeladen werden kann, muss nur in ein beliebiges Verzeichnis entpackt werden. Abschließend muss noch die Umgebungsvariablen **GROOVY_HOME** auf das Installationsverzeichnis gesetzt und **GROOVY_HOME/bin** zur PATH-Variablen hinzugefügt werden. Schon stehen

- über den Aufruf **groovysh** eine interaktive Shell, in die direkt Groovy-Anweisungen eingegeben werden können, und
- über den Aufruf **groovyConsole** eine interaktive Console in einem eigenen Fenster

zur Verfügung. Zusätzlich können Groovy-Skripte über den Aufruf **groovy MyScript.groovy** direkt ausgeführt werden. Außerdem übersetzt der Aufruf **groovyc MyScript.groovy** das Skript in Java-Bytecode. In IDEs kann über die **GroovyShell**-Klasse ein Skript ausgeführt werden. Der Aufruf dazu sieht folgendermaßen aus:

```
java groovy.lang.GroovyShell foo/MyScript.groovy [arguments]
```

Die folgenden Jars aus der Groovy-Installation müssen dazu im Classpath sein:

- asm-1.4.1.jar
- asm-attr-1.4.1.jar
- asm-util-1.4.1.jar
- groovy.jar

Wem die liebgewonnenen Features in einer IDE, wie z. B. Code Completion und Syntax Highlighting, fehlen, dem seien die verfügbaren Plug-Ins (für Eclipse, IntelliJ, JEdit ...) empfohlen.

Just Scripting

Groovy-Skripte haben üblicherweise die Endung „.groovy“ und können ungeordnet Anweisungen, Methodendefinitionen und Klassendefinitionen enthalten – so kann man einfach los-“scripten“. Ein Groovy-Skript kann direkt per Konsole oder über das Bean Scripting Framework [BSF1] ausgeführt werden. Das folgende Skript

```
result = ['one', 2, 'three'].join(',')
print result
```

kann durch den Aufruf

```
GROOVY_HOME/bin/groovy.bat MyScript.groovy
```

ausgeführt werden und produziert die folgende Ausgabe:

```
one,2,three
```

Es wird direkt auf einer Liste **['one', 2 , 'three']** die **join()**-Methode ausgeführt, die dafür sorgt, dass die einzelnen Elemente der Liste mit dem in der **join()**-Methode übergebenen Zeichen verknüpft werden. Dabei ist egal, ob die Elemente der Liste vom selben Typ sind – Groovy beherrscht Autoboxing (das jetzt erst bei Java 1.5 eingeführt wird). Die Variable **result** ist untypisiert. Außerdem ist die **print()**-Methode direkt aufrufbar und an keine Klasse gebunden (ebenso **println**). Zugegeben, dass es sich dabei nur um Kleinigkeiten handelt, aber diese Vereinfachungen machen das Arbeiten mit Skriptsprachen so angenehm und beliebt.

Dasselbe Skript kann auch mit dem **groovyc**-Kommando in Bytecode übersetzt werden. In diesem Fall wird unter anderem auch eine **main()**-Methode erzeugt, sodass die Klasse mit einem „normalen“ java-Aufruf ausgeführt werden kann:

```
java -classpath GROOVY_HOME/lib/asm-1.4.1.jar;
  GROOVY_HOME/lib/asm-attrs-1.4.1.jar;
  GROOVY_HOME/lib/asm-util-1.4.1.jar;
  GROOVY_HOME/groovy-1.0-beta-5.jar;. MyScript
```

Natürlich kann diese Java-Klasse auch aus anderen Java-Implementierungen aufgerufen werden!

Methoden und Klassen müssen nicht vor dem ersten Aufruf definiert sein und Methoden können auch außerhalb von Klassendefinitionen deklariert werden. Diese müssen dann mit dem Schlüsselwort **def** gekennzeichnet sein. Diese „losen“ Methoden werden zu statischen Methoden in Java übersetzt.

Klassen und Beans

Ein kleines Beispiel soll helfen, die Features von Groovy ein bisschen näher zu erläutern. Ein eigenes kleines Weblog (Blog) mit den Klassen **Person**, **Entry** und **Blog** soll über einen eigenen Web-Server zur Verfügung gestellt werden. Die Einträge im Weblog sollen in einer Textdatei gepflegt werden können. Und natürlich alles in Groovy!

Ein erster Blick auf die **Blog**-Klasse (s. Listing 3) verrät schon, dass Klassen in Groovy auf das Wesentliche reduziert sind. Das **author**-Attribut ist vom Typ **Person** und wird gleich zugewiesen. Über den **new**-Operator wird ein Objekt vom Typ **Person** erzeugt und es werden gleich die Attribute **name** und **mail** im Konstruktoraufruf gesetzt. Ein Blick auf die **Person**-Klasse verrät, dass es diesen Konstruktor gar nicht gibt (wie es in Java wäre!):

```
class Person {
  String name
  String mail
}
```

In Groovy ist es möglich beliebige Attribute einer Klasse im Konstruktor durch die Übergabe einer Map mit Schlüssel/Wert-Paaren zu setzen. Auf die Attribute kann auch über den Ausdruck **author[name]** zugegriffen werden – das funktioniert natürlich auch mit existierenden JavaBeans. In Groovy heißen diese Beans selbstverständlich „Groovy Beans“! Die zur **Person**-Groovy-Bean äquivalente Java-Bean ist deutlich unfangreicher (s. Listing 4).

```
import java.net.*
import org.codehaus.groovy.sandbox.markup.StreamingMarkupBuilder
import java.io.File

pattern = ~"title:'(.*)',summary:'(.*)',scope:(private|public)"
// Datei mit den Einträgen
file = new File('entries.txt');
// erzeuge Blog Klasse
blog = new Blog()
// erzeuge MarkupBuilder
xml = new StreamingMarkupBuilder()
// definiere Closure
m = {
 html() {
  head() {title(blog.title)}
   body() {
    a(href:blog.link) { h1(blog.title) }
    a(href:'mailto://'+blog.author.mail) { h2(blog.author.name) }
    table() {
     for (e in blog.entries.findAll {it.scope =="public"}) {
      tr() { th(e.title) }
      tr() { td(e.summary) }
     }
    }
   }
 }
}
// starte Server
server = new ServerSocket(9991)
while(true) {
 println "Server [Port:9991] started..."
 server.accept() { socket |
  socket.withStreams { input, output |
   try {
    input.eachLine() { line |
     println line
     if (line.length() == 0) {
      throw new GroovyRuntimeException()
     }
    }
   } catch (GroovyRuntimeException b) { }
   output.withWriter { writer |
    reload()
    writer << "HTTP/1.1 200 OK\n"
    writer << "Content-Type: text/html\n\n"
    writer << xml.bind m
   }
  }
 }
}
// lädt Einträge aus der Datei
def reload() {
 line = 0
 // Liste löschen
 blog.entries.clear()
```

```
  file.readLines().each {
    line++
    m = it =~ pattern
    if (m.matches()) {
    blog.entries << new Entry(id:line,
                    title:m.group(1),
                    summary:m.group(2),
                    scope:m.group(3))
    } else {
      println "Line ${line} ,${it}' doesn't match!"
    }
  }
}
// Blog-Klasse
class Blog {
  title = "Groovy-Blog"
  link = "http://www.innoq.com/blog/mt"
  author = new Person(name:'Marcel Tilly',
                  mail:'marcel.tilly@innoq.com')
  entries = []
}
// Entry-Klasse
class Entry {
  title
  id
  summary
  scope
}
// Person-Klasse
class Person {
  name
  mail
}
```

Listing 3: Beispiel-Implementierung: Weblog „Blog.groovy"

```
public class Person {
  private String name;
  private String mail;
  public String getName() {
    return name;
  }
  public void setName(String value) {
    name = value;
  }
  public String getMail() {
    return mail;
  }
  public void setMail(String value) {
    mail = value;
  }
}
```

Listing 4: Die Klasse Person in Java

Input/Output

Im Folgenden sollen die Einträge des Weblogs über eine Textdatei gepflegt werden können. Eine Zeile entspricht dabei einem Eintrag und hat die Form:

```
title:'Installation',summary:'Wie wird Groovy Installiert',scope:public
```

Wie bei Perl oder anderen Skriptsprachen ist das Auslesen aus einer Datei auch in Groovy recht einfach:

```
import java.io.File;
new File("entries.txt").readLines().each{ println it }
```

Es wird die **File**-Klasse aus dem **java.io**-Package verwendet. In Groovy ist diese Klasse um die **readLines()**-Methode erweitert [GRO2], die eine **java.util.Collection** zurückgibt. Auf dieser Collection kann die **each()**-Methode aufgerufen werden, die wiederum die Collection-Klasse erweitert. Der **each()**-Methode wird eine Closure übergeben, die die weitere Verarbeitung übernimmt.

Closures

Closures sind Codeblöcke, die an eine Methode zur Ausführung übergeben werden können, und entsprechen in Java Anonymous-Inner-Classes. Der Unterschied zu Inner-Classes ist, dass der äußere Status in einer Closure verändert werden kann und ebenso der Inhalt einer Closure auch außerhalb sichtbar ist. Closures verhalten sich wie normale Objekte und können bei Bedarf aufgerufen werden. Zusätzlich ist es möglich Parameter an eine Closure zu übergeben. Eine Closure wird durch einen Namen und den Codeblock, der ausgeführt werden soll, definiert, z. B.

```
c = { firstname, name | println("Hello ${firstname} ${name}") }
c.call('Marcel', 'Tilly')
```

oder einfach:

```
c('Marcel', 'Tilly')
```

Wird kein Parameter explizit definiert, so kann implizit über das Schlüsselwort **it** auf das übergebene Element zugegriffen werden (siehe **each()**-Methode).

Der String, der in der **print()**-Methode ausgegeben wird, enthält zudem noch die eingebetteten Werte, ${firstname} und ${name}. Das ist vergleichbar mit Properties in Ant oder mit Werten in PHP. Die Syntax ist einfach: Auf ein vorgestelltes $ folgt der Variablennamen in geschweiften Klammern.

Zurück zum eigentlichen Problem. Wir wollten zeilenweise aus einer Datei lesen. Die Zeilen werden inhaltlich auf die Klasse **Entry** verteilt. Das Aufschlüsseln übernimmt dazu ein regulärer Ausdruck, der in Groovy mit dem Tilde-Zeichen (~) definiert wird:

```
pattern = ~"title:'(.*)',summary:'(.*)', scope:(private|public)"
```

Die **reload()**-Methode (Listing 3) übernimmt nun die Arbeit. Das Schlüsselwort **def** zeigt an, dass die Methode nicht innerhalb einer Klasse definiert ist. Die Closure innerhalb der **each()**-Methode ist in diesem Fall etwas komplexer. Sie prüft zunächst, ob die aktuelle Zeile (**it**) dem regulären Ausdruck (pattern) entspricht. Ist dieses der Fall, wird der **Entries**-Liste in der **Blog**-Klasse ein neuer Eintrag (Entry) hinzugefügt. Die einzelnen Attribute des Eintrags werden aus der Zeile ausgelesen, indem der Matcher **m** die Treffer des regulären Ausdrucks auswertet und über die **group()**-Methode auf die einzelnen Treffer zugreift.

Groovy Markup

In Groovy gibt es eine direkte Unterstützung zur Erzeugung von XML, HTML, SAX und W3C DOM. Um ein entsprechendes XML-Dokument zu erzeugen, muss nur ein Markup-Objekt erzeugt werden, dem dann die zu erzeugende Struktur übergeben wird. Syntaktisch handelt es sich dabei um eine Reihe von Closures, die auf Methoden aus-

geführt werden. Im Beispiel wird ein **StreamingMarkupBuilder** verwendet, dieser kann über die **bind()**-Methode auf einen beliebigen Stream schreiben. Zu diesem Zweck wird eine Closure **m** definiert, die eine HTML-Struktur unseres Weblogs ausgeben soll. Für jede Methode innerhalb der Closure **m** erzeugt der MarkupBuilder später ein Tag (z. B. <HTML>...</HTML>). Die Map, die einer Methode übergeben wird, wird als Attribute des XML-Elementes interpretiert.

In der **for**-Schleife gibt es noch eine Besonderheit. Die Entries des Blogs haben ein **scope**-Attribut, das angeben soll, ob die Einträge bereits im Blog sichtbar sein sollen oder nicht. Nur Einträge mit **public-Scope** sollen wirklich angezeigt werden. Hierzu wird ein GPATH-Ausdruck in der Closure der **findAll()**-Methode verwendet. GPATH ist die Groovy-Variante zu XPATH. Mit GPATH besteht die Möglichkeit auf dem Objektgraph zu navigieren oder Selektionen vorzunehmen. Die **findAll()**-Methode gibt nur Element zurück, deren **scope**-Attribute (**it** ist in diesem Fall vom Typ **Entry**) **public** sind. Natürlich lässt sich ein solcher GPATH-Ausdruck auch beliebig kombinieren, z. B. **it.scope== „public" && it.id > 3.**

Eine Anmerkung: In Groovy kann der ==-Operator zum Vergleich von Strings verwendet werden, da in Groovy Überladen von Operatoren möglich ist und der ==-Operator für Strings auf die **equals()**-Methode mappt.

Da beliebige XML-Strukturen mit dem MarkupBuilder erzeugt werden können, kann auch über die AntBuilder-Klasse ein Ant-Skript erzeugt werden. Ebenso gibt es eine SwingBuilder-Klasse, über die SWING-UIs beschrieben und erzeugt werden können. Das ist sehr elegant, da das Setzen von Properties und das Verknüpfen mit bestimmten Aktionen einfacher werden. In Java muss für Aktionen stets eine Klasse als Listener definiert werden. In Groovy übernehmen Closures diese Arbeit. Ein kleines Beispiel, das ein Label, ein Textfeld und einen Button erzeugt (s. Abb. 1), zeigt Listing 5.

```
import java.awt.BorderLayout
swing = new groovy.swing.SwingBuilder()

frame = swing.frame(
 title:'GroovyGUI',
 size:[200,60],
 location:[100,100],
 defaultCloseOperation:javax.swing.WindowConstants.EXIT_ON_CLOSE) {
 panel(layout:new BorderLayout()) {
  label(text:'Name',constraints:BorderLayout.WEST)
  field = textField(text:'Bitte umdrehen!',
                    constraints:BorderLayout.CENTER)
  button(text:'<->', constraints:BorderLayout.EAST,
   actionPerformed:{ field.setText(field.getText().reverse()) })
 }
}
frame.show()
```

Listing 5: Beispiel zum SwingBuilder

Abb. 1: Groovy SwingBuilder in Aktion

Groovy SQL

Natürlich würde ein richtiges Weblog-System die Inhalte nicht in einer Datei ablegen, sondern in einer Datenbank. Groovy bietet auch für den Zugriff auf Datenbanken Unterstützung (wie sollte es auch anders sein, wird schließlich von Java geerbt!). Über Groovy SQL können einfach SQL-Statements abgesetzt werden und das resultierende ResultSet kann dann mit Bordmitteln schnell und einfach verarbeitet werden.

Groovlets

Für das Beispiel ist nun alles vorhanden: Die Einträge können aus einer Datei gelesen werden und das HTML kann gerendert werden; fehlt also nur noch die Serverimplementierung. Der Server soll Requests entgegennehmen und die Blog-Einträge als HTML-Response zurückliefern. Hierzu kann die Java-Klasse **java.net.ServerSocket** verwendet werden. Für diese gibt es als Erweiterung [GR02] eine **accept()**-Methode, die eine Closure zur weiteren Verarbeitung erwartet. Die **withStreams()**-Methode des Sockets hat einen Input- und einen Output-Stream, die den Request und den Response abhandeln können. Im Input-Stream lassen wir nur jede Zeile des Requests ausgeben, auf den Output-Stream hingegen schreiben wir den HTTP-Response-Header und die gerenderte HTML-Seite. Der Aufruf der **reload()**-Methode sorgt dafür, dass für einen Request stets die aktuellsten Daten aus der Datei geholt werden.

Berechtigterweise stellt sich hier die Frage: Warum macht man das nicht über JSPs? Genau das haben sich die Entwickler von Groovy wohl auch gefragt. An den meisten Stellen wurde das Rad nicht neu erfunden. So verhält es sich auch hier. Die Antwort heißt: Groovlets! Es ist in Groovy möglich Java-Servlets zu schreiben. Es existiert eine **GroovyServlet**-Klasse, die dafür sorgt, dass Quelldateien, die auf „.groovy" enden, compiliert, geladen und gecacht werden. Die Variablen **session**, **output** und **request** sind implizit und können in einem Groovlet direkt verwendet werden. Auf der Groovy-Website findet sich ein nettes Beispiel zu Groovlets.

Die Weblog-Implementierung ist nun fertig und kann ausgeführt werden. Nach dem Aufruf

```
groovy BlogServer.groovy
```

sollte die Ausgabe

```
Server [Port:9991] started...
```

erscheinen. Mit einem Browser kann der Inhalt des Weblogs (**http://localhost:9991**) angezeigt werden.

Ein kleiner Tipp zum Schluss [STR1]: Falls Sie die Methoden zu einer Klasse vergessen haben, können Sie sich über die Groovy-Shell und ein entsprechendes Skript schnell eine Liste ausgeben lassen:

```
1> println java.io.File.methods.name.sort()
2> go
[canRead, canWrite, compareTo, compareTo, ...]
```

Fazit

Groovy bietet eine Menge an Features, die das Arbeiten mit Java vereinfachen. Viele der Features wurden im Artikel kurz vorgestellt, allerdings konnten die meisten nur oberflächlich angekratzt werden. Ihr Interesse sollte aber soweit geweckt sein, dass Sie auch einen weiteren Blick auf Groovy riskieren. Die Website bietet viele Beispie-

le zu allen Themen. Natürlich ist Groovy noch nicht für den Einsatz in kritischen Projekten geeignet, hierzu ist die Sprache noch zu „instabil“. Einige Punkte werden in der Groovy Community [WIK1] gerade noch diskutiert.

Kleinere, unkritische Aufgaben kann man aber bereits jetzt mit Groovy lösen. Das Arbeiten mit Dateien und XML-Dokumenten ist sehr angenehm, ebenso lassen sich mit Groovy schön und einfach Testskripte für den Unit-Test erstellen. Probieren Sie es mal aus!

Mich persönlich haben allerdings die unklaren Fehlermeldungen gestört, dennoch lässt sich sagen, dass es richtig Spaß macht mit Groovy zu arbeiten: Es „grooved“ sozusagen!

Literatur und Links

[BSF1] Bean Scripting Framework, **http://jakarta.apache.org/bsf/**
[GRO1] Groovy-Website, **http://groovy.codehaus.org**
[GRO2] Groovy JDK, **http://groovy.codehaus.org/groovy-jdk.html**
[JSR1] JSR 241, **http://www.jcp.org/en/jsr/detail?id=241**
[STR1] J. Strachan, Präsentation auf der Java One 2004, **http://www.codehaus.org/~jstrachan/GroovyJavaOne-2004.ppt**
[WIK1] Groovy Wiki, **http://wiki.codehaus.org/groovy/WishList**

Marcel Tilly ist Senior Consultant bei der innoQ Deutschland GmbH. Sein Arbeitsschwerpunkt ist derzeit die Konzeption und Umsetzung von Service-orientierten Architekturen sowie der Einsatz generativer Softwareentwicklung in Projekten. E-Mail: marcel.tilly@innoq.com.

Weiterführende Informationsquellen

http://groovy.codehaus.org