Ongeveer vijf jaar geleden startten James Strachan en Bob McWhirter het Groovy project[1]. Het doel was een dynamische taal te ontwikkelen voor het Java-platform, die naadloos aansluit op bestaande Java-ontwikkeling. Talen als Ruby, Smalltalk en Perl vormden de inspiratie voor Groovy. De dynamische eigenschappen en elegante syntax van Groovy hebben geleid tot standaardisatie van Groovy in JSR-241, waarmee het de tweede 'officiële' taal op het Java platform wordt (na Java zelf). In de daaropvolgende jaren is Groovy opgepakt door een voortdurend groeiende groep ontwikkelaars. Tegenwoordig[2] heeft Groovy standaardkoppelingen aan boord voor de meest uiteenlopende Java-frameworks: van Swing tot SWT en GraphicsBuilder, van XML lezen, schrijven en webservices tot aan GMaven en GAnt, en niet te vergeten Grails (Groovy on rails), waaraan we in het volgende nummer aandacht besteden.

Even wennen, maar dan wordt het: 'Groovy, baby!'

Levente Bokor
is senior consultant
bij Capgemini

De nauwe integratie tussen Groovy en Java is een belangrijke reden voor adoptie geweest. Hoewel Groovy een geheel eigen syntax heeft, levert een gecompileerd `.groovy`-bestand gewoon Java-classes en -packages op. Met Groovy kun je standaard Java interfaces implementeren, inclusief de Java 1.5 generics.

Wat is een dynamische taal nu precies? Nu, *dynamisch* betekent in deze dat referenties naar methodes en properties niet compile-time, maar pas run-time bepaald worden. En hoe deze bepaling precies verloopt, is ook aan te passen. Laten we één en ander verhelderen door middel van een voorbeeld.

```
class House {
  def size() {
    return 10
  }
}

def printSize (obj) {
  println "Number of objects: ${obj.size()}"
}

printSize (new House())
printSize (new java.util.ArrayList())

Number of objects: 10
Number of objects: 0
```

In bovenstaande listing is te zien dat datatypes veelal niet genoemd (hoeven) worden in Groovy. De methode `size()` in de class `House` geeft een Integer als resultaat, maar dit had elk ander object mogen zijn. Ook zijn puntkomma's aan het einde van de regel niet verplicht.

Het interessante is de methode `printSize (obj)`. Deze voert op obj de methode '`size()`' uit, zonder dat compile-time duidelijk is wat het precieze type van obj is. We kunnen er dus in stoppen wat we willen: zowel het '`new House()`' als de '`new java.util.ArrayList()`' worden succesvol door `printSize` behandeld. Pas run-time wordt de exacte methode uitgezocht.

De '`${...}`' syntax in Groovy is ook interessant (regel 8): het is mogelijk binnen strings expressies op te nemen, waarin verwezen mag worden naar variabelen die in de scope van de string bekend zijn.

Closures: dynamische code

Het flexibele type-systeem is maar een deel van de dynamiek van Groovy. In Groovy kan namelijk een stukje source code zelf ook als variabele gebruikt worden. Zo'n stukje 'uitvoerbaar geheel' wordt een *closure* genoemd. Een voorbeeld:

```
def helloClosure = { param -> println "Hello, ${param}!"}
```

```
println helloClosure instanceof groovy.lang.Closure

helloClosure.call("world")
helloClosure("java")

def execute (Closure c, String s) { c.call(s) }

execute (helloClosure, "groovy")

true
Hello, world!
Hello, java!
Hello, groovy!
```

Aan de variabele `helloClosure` wordt een closure toegekend, die bij uitvoeren de welbekende wereldgroet zal afdrukken. De closure heeft een argument (`param`) dat als onderdeel van de afgedrukte string zal dienen.
Een closure is een first-class type in Groovy, zoals regel 15 laat zien. Een closure is uit te voeren door de call() methode aan te roepen, maar Groovy staat het toe om closures syntactisch als methodes te beschouwen, waardoor direct '`helloClosure()`' geschreven mag worden.
Omdat closures gewone datatypes zijn, kunnen ze dus ook aan methodes meegegeven worden. Dit wordt in de `execute` method toegepast: binnenin de methode wordt het closure '`c`' uitgevoerd.
Groovy heeft een flink aantal extra methodes aan standaard Java types toegevoegd[3], waarmee in combinatie met closures krachtige constructies kunnen worden opgeschreven.

```
def list = ["a", "b", "c", "d"]
list.each( { item -> print item } )
def upperCaseList = list.collect { it.toUpperCase()
}
upperCaseList.each { print it }

abcdABCD
```

Bovenstaande listing bevat een aantal nieuwe syntaxmogelijkheden. '`list`' is een `ArrayList` met daarin vier strings. Groovy bevat standaard de methode '`each`', die door de `ArrayList` heen loopt, en voor elk element een meegegeven closure uitvoert.
De ronde haken van regel 24 zijn optioneel: indien een methode enkel een closure als argument heeft, mogen de ronde haken worden weggelaten (waardoor enkel de accolades van het closure overblijven). Dat is in regel 25 te zien. Daar zien we ook, dat als een closure geen parameters aangeeft, er een geïmpliceerde parameter '`it`' altijd aanwezig is. Hiermee kunnen closures nog korter worden opgeschreven.
De methode '`collect`' is een ander handigheidje voor verzamelingen. Ook `collect` voert de meegegeven closure uit voor elk element in de lijst. De resultaten van elke closure-aanroep worden echter verzameld in een nieuwe lijst, wat het resultaat van de `collect`-aanroep wordt. In het voorbeeld wordt hiermee een nieuwe lijst gemaakt, die de oorspronkelijke elementen in hoofdletters bevat.

Swingen met Groovy

Wie al eens een Swing applicatie in Java heeft geschreven, weet welke enorme hoeveelheid regels code soms nodig is om schijnbaar eenvoudige zaken te realiseren. Toch biedt Swing als framework unieke mogelijkheden, juist door de vele abstractielagen. Groovy biedt hier gelukkig uitkomst. De SwingBuilder groovy class stelt een ontwikkelaar in staat supersnel een Swing-applicatie op te zetten.
Zie hier een voorbeeld:

```
def count = 0;
SwingBuilder.build {
   frame (title:"My first groovy app", show: true,
size:[150,100],
        defaultCloseOperation: JFrame.EXIT_ON_CLOSE)
{
     borderLayout()
      label (id:"textlabel", text:"Clicked ${count}
time(s).",
            constraints: BorderLayout.CENTER)
   button (text:'Click Me', constraints:BorderLayout.
SOUTH,
             actionPerformed: {
       count++
       textlabel.text = "Clicked ${count} time(s)."
     } )
   }
}
```

De code spreekt vrijwel voor zich, als je thuis bent in Swing: een label wordt bijgewerkt met nieuwe tekst, zodra de knop wordt ingedrukt. De layout wordt geregeld door BorderLayout. De applicatie wordt be'indigd als het frame wordt gesloten (`EXIT_ON_CLOSE`). De magie ligt in het creatief gebruik van closures. Zie regel 29: het frame wordt gemaakt met bepaalde attributen; de closure die als laatste argument na `frame(...)` staat, is verantwoordelijk voor het aanmaken van de *inhoud* van het frame. Zo wordt met een verzameling geneste closures, de boomstructuur van de Swing user interface in Groovy weergegeven.
Er zijn nog een aantal andere handigheden ingebakken. Door '`show: true`' wordt bijvoorbeeld het frame automatisch direct getoond. Verder wordt het label uit regel 32 een id gegeven ('`textlabel`'), zodat er later vanuit regel 37 aan gerefereerd kan worden. Dit scheelt weer het aanmaken van een extra variabele.
Overigens bevat bovenstaand voorbeeld toch nog overbodige (dubbele) code: de tekst voor op het label wordt twee maal genoemd. Door gebruik te maken van de nieuwe data binding features[4] kan de code nog verder vereenvoudigd worden.

Slurpen mag weer

In een wereld waarin XML inlezen al snel met mooie XSD's en JAXB wordt verbonden, blijkt

toch vaak dat niet alles zo 'standaard' is als je het graag zou willen hebben. XML-syntax die niet helemaal aansluit, complexe XML-queries, tot aan het parsen van HTML, zijn zaken die in projecten langs kunnen komen. Gelukkig biedt Groovy ook voor het binnenhalen van XML uitkomst: de XMLSlurper.
Neem als voorbeeld deze xml-export van een taak-beheersysteem:

```
<?xml version="1.0" encoding="UTF-8"?>
<datalist>
  <element>
    <field>
      <field-name>Artifact ID</field-name>
      <field-value>artf321453</field-value>
    </field>
    <field>
      <field-name>Title</field-name>
       <field-value>Groovy on rails demo-applicatie
maken</field-value>
    </field>
  </element>
  <element>
    <field>
      <field-name>Artifact ID</field-name>
      <field-value>artf321456</field-value>
    </field>
    <field>
      <field-name>Title</field-name>
       <field-value>Handig XML voorbeeld bedenken</
field-value>
    </field>
  </element>
</datalist>
```

Het is aan de Groovy-applicatie om de xml in te lezen en tot een zinnige datastructuur om te zetten. Wat de XMLSlurper in feite doet, is de XML-boom als een boom van (fictieve) groovy-properties ter beschikking te stellen.

```
def datalist = new XmlSlurper().parse("test.xml");
def groovyItems = datalist.element.findAll { it.
field.any { it."field-value" =~ ".*oovy.*" } }
groovyItems.each { println it.field.find { it."field-
name".text () == "Title" }."field-value" }

def structure = datalist.element.list().collect { [
    id: it.field.find { it."field-name".text() ==
"Artifact ID" }."field-value",
   title: it.field.find { it."field-name".text() ==
"Title" }."field-value"
] }
structure.each { println "Item ${it.id} : ${it.
title}" }

Groovy on rails demo-applicatie maken
Item artf321453 : Groovy on rails demo-applicatie
maken
Item artf321456 : Handig XML voorbeeld bedenken
```

Allereerst wordt een XMLSlurper gemaakt waarmee de test.xml ingelezen wordt. De variabele `datalist` zal daarna 'wijzen' naar de root-tag van het XML-document (`<datalist>`). Als van een XMLSlurper een bepaalde property wordt opgevraagd, zal dit een nieuwe XMLSlurper opleveren die naar de sub-tag(s) met de naam van de property wijst. Bijvoorbeeld: `datalist.element` zal wijzen naar alle `<element>` tags binnen de `<datalist>` root tag.

We gaan op zoek naar alle `<element>` tags, waarvan enig `<field>` een `<field-value>` tag bevat, die aan de reguliere expressie *.*oovy.** voldoet. Lees de vorige zin aandachtig door, en vergelijk hem dan met regel 65. Beiden zijn equivalent. Dit is de daadwerkelijke kracht van Groovy: door het declaratief opschrijven, zijn Groovy expressies bijna direct taalkundig te interpreteren.
In regel 66 wordt van alle gevonden `<element>` tags, het `<field>` waarvan de `<field-name>` gelijk is aan '`Title`' opgezocht, en daarvan de `<field-value>` afgedrukt.
De code vanaf regel 68 lijkt meer op iets wat in een daadwerkelijke applicatie gedaan zal worden: het omzetten van de xml-structuur naar een handzamere vorm. In dit geval wordt voor elk `<element>` een map aangemaakt, waarbij elke map '`id`' en '`title`' entries heeft. Groovy's syntax om maps te maken is `[ key1: value1, key2: value2, ... ]`.
Vervolgens kan in regel 72 eenvoudig over `structure` geïtereerd worden, om de '`id`' en '`title`' van elk element af te drukken.

Bouwen aan nieuwe XML

Regelmatig zal ook XML geproduceerd moeten worden. Een goed voorbeeld is het uitvoeren van html. Op het produceren van XML is hetzelfde builder-patroon van toepassing waar ook de eerder genoemde SwingBuilder gebruik van maakt. In dit geval wordt gebruik gemaakt van de StreamingMarkupBuilder. Zie hier een voorbeeld, waarin we voortborduren op de hiervoor geïntroduceerde takenlijst, maar nu als daadwerkelijke class.

```
class Item { def id; def title }

def items = [
   new Item (id: "1", title: "Groovy on rails demo-
applicatie maken"),
   new Item (id: "2", title: "Handig XML voorbeeld
bedenken") ]

println new StreamingMarkupBuilder().bind{
  html() {
    body(bgcolor:'blue') {
      h1('Todo lijst')
      ul {
        items.each {
          li(it.title)
        }
      }
    }
  }
}

<html><body bgcolor='blue'><h1>Todo lijst</
h1><ul><li>Groovy on rails demo-applicatie maken</
li><li>Handig XML voorbeeld bedenken</li></ul></
body></html>
```

Zie allereerst de definitie van de class Item op regel 73. Met '`def title`' wordt zowel een private variabele title geïntroduceerd, alsmede public getters en setters. Dat is nog eens handig.

Een berg documentatie en een constante leercurve

Op regel 75 defini'ren we vervolgens een lijst van twee Item instanties. Op basis van deze lijst zal de HTML gegenereerd worden, met behulp van de StreamingMarkupBuilder. Binnen de `bind()` methode wijst het zich eigenlijk vanzelf: de StreamingMarkupBuilder accepteert elke methode-aanroep, waarbij de naam van de methode die aangeroepen wordt, resulteert in het aanmaken van een xml-tag met die naam. Als argument aan zo'n methode wordt een nested closure meegegeven, waarbinnen vervolgens op dezelfde manier sub-xmltags aangemaakt kunnen worden.
Door de mogelijkheden van XMLSlurper en StreamingMarkupBuilder te combineren, kunnen zeer krachtige XML processing scripts geschreven worden. Het resultaat is qua structuur vergelijkbaar met XSLT, maar veel krachtiger doordat alle mogelijkheden van Java en Groovy direct beschikbaar zijn.

Webservices op bestelling

De dynamische mogelijkheden van Groovy zijn een uitgelezen mogelijkheid om eenvoudig webservices aan te bieden, maar ook aan te roepen. Een Groovy-uitbreiding die hierop inspeelt, is GroovyWS[5]. Intern maakt GroovyWS gebruik van het CXF-framework van Apache, waardoor moderne standaarden als WS-Security onder handbereik zijn.
Het aanbieden van een webservice gaat als volgt:

```
class DiskService {
    long getFreeSpace (String path) {
      return new File(path).getUsableSpace();
    }
}

import groovyx.net.ws.WSServer
new WSServer().setNode("DiskService", "http://local-
host:6980/DiskService")
```

Een willekeurige class kan dus als service worden aangeboden, door aan `setNode` de classname mee te geven, samen met de URL waarop de webservice bereikbaar moet zijn. Dat is echt alles: bovenstaand programma is een volledig werkende webserver, die genoemde `DiskService` als WSDL aanbiedt. Alle noodzakelijke libraries zitten in GroovyWS.
Aanroepen van een webservice is al bijna even eenvoudig:

```
import groovyx.net.ws.WSClient
def proxy = new WSClient("http://localhost:6980/
DiskService?wsdl", this.class.classLoader)
println proxy.getFreeSpace ("/")
```

Indien eerder genoemde webservice opgestart is, zal dit de beschikbare disk-space op '/' weergeven (indien op een Unix-systeem gewerkt wordt, uiteraard).

In beide gevallen wordt geen code gegenereerd. De `WSClient` zal echter wel dynamisch classes aanmaken, aangezien CXF op die manier werkt. Daardoor heeft `WSClient` ook een link naar de huidige classloader nodig, om de aangemaakte classes in de juiste classloader terecht te laten komen.
Omdat GroovyWS een relatief nieuw project is, zal de integratie tussen CXF en Groovy nog niet altijd even vanzelfsprekend verlopen. Een voorbeeld is het feit dat Groovy bij elke gecompileerde class een 'metaclass' property mee genereert (onderdeel van het dynamisch gedrag). Het data binding framework van CXF (Aegis binding[6]) raakt hierdoor in de war, omdat het per default niet weet hoe deze 'metaclass' als XSD aan te bieden. Aegis kan die metaclass gewoon negeren, maar momenteel moet dat elke keer expliciet verteld worden.

Getting in the groove

De in dit artikel genoemde subprojecten van Groovy zijn nog maar het topje van de ijsberg. Op de homepage van Groovy zijn links de vinden naar tientallen andere groovy builders, waarvan zeker de GAnt en GMaven builders de moeite waard zijn. Een andere builder die momenteel flink in ontwikkeling is, is de GraphicsBuilder van Andres Alimary[7]. Dit laatstgenoemde project is een serieuze concurrent voor JavaFX.
Uit persoonlijke ervaring kan ik zeggen dat het voor elke Java-ontwikkelaar de moeite waard is eens serieus naar Groovy te kijken. Er is een berg documentatie en de leercurve is heerlijk constant. Loop gewoon eens door de voorbeelden heen en al snel zul je kleine Groovy scriptjes gaan maken die het developer-leven stukken eenvoudiger maken. De Groovy-community weet steeds weer de echt interessante projectjes aan te trekken, waarbij steeds Austin Powers' quote van toepassing blijft: '*Groovy, baby!*' «

Referenties

[1] Groovy in Action, Dierk König, 2007 Manning Publications
[2] http://groovy.codehaus.org/, An agile dynamic language for the Java platform
[3] http://groovy.codehaus.org/groovy-jdk/, The methods added to the JDK to make it more groovy
[4] http://groovy.codehaus.org/SwingBuilder.bind, Binding one property to another
[5] http://groovy.codehaus.org/GroovyWS, Webservices for Groovy
[6] http://cwiki.apache.org/CXF20DOC/aegis-databinding.html, Aegis databinding
[7] http://www.jroller.com/aalmiray/tags/graphicsbuilder, Andres Alimary's weblog