Gradle – Konkurrenz für Ant und Maven

Das Beste zweier Welten

Karl Banke

Apache-Werkzeuge wie Ant und Maven unterstützen Softwareentwickler seit Jahren. Das ebenfalls als Open Source realisierte Gradle will die beiden übertreffen. Ein Meilenstein der Version 1.0 dient als Vergleichsmaßstab.

Den Bauprozess (Build) in Java-Projekten dominieren derzeit vor allem die Werkzeuge Ant [a] und Maven [b]. Während ein Entwickler in Ant den Bau als eine Reihe voneinander abhängiger Schritte – sogenannter Targets – explizit definieren muss, verfolgt Maven einen konsequenten Convention-Over-Configuration-Ansatz. Die Struktur der Quelldateien und der Ablauf des Baus sind für ein bestimmtes Zielartefakt, etwa eine JAR-Bibliothek oder eine Webanwendung, relativ starr vorgegeben.

Zusätzlich bietet Maven ein ausgefeiltes Artefakt- und Abhängigkeitsmanagement und ermöglicht so, für jedes Projekt die benötigten Dependencies aus einem privaten oder öffentlichen Repository herunterzuladen und das erzeugte Artefakt selbst dort zu veröffentlichen. Beide Werkzeuge haben große Verbreitung gefunden, aber durchaus einige Nachteile. Ant-Skripte geraten recht schnell groß und unübersichtlich. Zudem sind neue Funktionen (Ant-Tasks) nur mit recht hohem Aufwand an Java-Programmierung zu erstellen. Maven hingegen punktet zunächst mit kompakten Build-Dateien und dem Dependency Management. Dafür sind die Prozesse hinsichtlich der verfügbaren Build-Phasen relativ starr, und es gilt als recht aufwendig, einen der zahlreichen Standardprozesse punktuell zu erweitern.

Hier kommt Gradle [c] als schlanke und flexible Alternative infrage. Es bietet nicht nur alle Vorteile von Maven und Ant – Dependency Management und Convention-Over-Configuration –, sondern außerdem die freie Gestaltung von Abhängigkeiten. Es ist durch eine eigene domänenspezifische Sprache (DSL) mächtig genug für eventuell nötige spezifische Anpassungen.

Gradle zu installieren ist denkbar einfach, es erfordert lediglich ein Java Development Kit ab Version 5. Nach dem Herunterladen und Auspacken einer der verschiedenen Gradle-Distributionen muss man lediglich das *bin*-Verzeichnis der Pfadvariablen hinzufügen und ist startbereit. Das Hauptkommando lautet wenig überraschend *gradle*. Mit der Option *--v* gibt es Versionsinformationen aus, beispielsweise Gradle 1.0-milestone-3, gefolgt von Versionsinformationen zu Groovy, Ant, Ivy, der JVM und dem Betriebssystem.

Beim Bauen eines Projekts erledigt Gradle nacheinander verschiedene Aufgaben, sogenannte Tasks. Eine build.gradle-Datei im Projektverzeichnis beschreibt ein Vorhaben, analog zu einer build.xml bei Ant oder der pom.xml bei Maven. Diese Datei enthält jedoch kein XML, sondern wird in der an Groovy angelehnten DSL erstellt. Im folgenden Beispiel sei ein existierendes, einfaches Maven-Java-Projekt in ein Gradle-Projekt umgewandelt. Es ist bewusst einfach gehalten und besteht lediglich aus zwei Java-Klassen und einigen JUnit-Tests.

Convention over Configuration

Ähnlich wie Maven bietet Gradle einen strikten Convention-over-Configuration-Ansatz. Für einen bestimmten Bauprozess wie die Erstellung eines Java-Programms muss man nicht mehr alle Schritte definieren, sondern es wird eine Reihe von Standardwerten angenommen. Die große Verbreitung von Maven hat beispielsweise dazu geführt, dass inzwischen viele Projekte die Maven-Konventionen zu den Ablageorten der

Quellcode-Dateien einhalten. Dies ermöglicht es, sich schnell in ein bestehendes neues Projekt einzuarbeiten.

Gradle geht hier einen Schritt weiter. Während Listing 1 den minimalen Umfang einer Maven-Projektdatei (*pom.xml*) definiert, ist eine einfache *build.gradle*-Datei nochmals deutlich kompakter. Sie besteht nur aus der Zeile

```
apply plugin: 'java'
```

Dies entspricht inhaltlich im Wesentlichen dem *packaging*-Tag in der *pom.xml*. Das Java-Plug-in [d] definiert eine Reihe von Tasks. Der Aufruf von *gradle tasks* bewirkt die Ausgabe aller Aufgaben des aktuellen Projekts. Hier erhält man schon eine recht umfangreiche Liste. Das Java-Plug-in unterstützt vertraute Tasks wie *clean*, *build* oder *test*. Dabei entspricht das Layout des Quellcodes der von Maven gewohnten Struktur.

Führt man diese einfache Build-Datei durch Aufruf von *gradle build* aus, kommt es erwartungsgemäß zu einem Fehler, da Gradle die Abhängigkeiten der *pom.xml* natürlich nicht kennt (siehe Listing 2). Das Werkzeug benötigt zwar weder Ant noch Maven, kann aber Maven-Repositories benutzen und bei Bedarf selbst eine *pom.xml* erzeugen. Und es enthält bei genauem Hinsehen Ant-Tasks und die Maven-Ant-Task-Bibliothek.

Maven-Repositories für Gradle nutzen

Gradle unterstützt das Konzept eines lokalen Repository, aus dem Abhängigkeiten zu laden sind. Über das Netzwerk heruntergeladene Abhängigkeiten speichert das Tool im Home-Verzeichnis unter *.gradle/cache* zwischen. Führt man Gradle im Unternehmen ein, kann es sein, dass es schon ein unternehmens- oder abteilungsweites Maven-Repository gibt. Erfreulicherweise ist es einfach, ein solches zu nutzen. Die folgenden Zeilen referenzieren das zentrale Maven-Repository und geben die Abhängigkeiten an.

```
repositories {
   mavenCentral()
}
dependencies {   compile group: ‚commons-log ⤷
   ging', name: ‚commons-logging', version: ‚1.1.1' ⤷
            testCompile group: ‚junit', name: ‚junit', ⤷
                                        version: ‚3.8.1' }
```

Gradle ermöglicht es, mehrere – auch firmeninterne – Repositories zu nutzen. Dabei können Maven-Repositories und die von Apaches populärem Dependency Manager Ivy [e] gleichzeitig verwendet werden. Ein eventuell vorhandenes lokales Maven Repository kann man ebenfalls nutzen. Mit der folgenden Zeile referenziert man beispielsweise das zentrale JBoss-Repository:

```
repositories {
   mavenCentral name: "jboss", urls: ⤷
      "http://repository.jboss.org/nexus/content/ ⤷
                                  groups/public-jboss/"
}
```

Da die Abhängigkeiten nun angegeben sind und aus dem Internet heruntergeladen werden können, ist der Bau erfolgreich.

```
c:\Users\bankkar\workspace\SimpleAccount ⤷
                                  Service>gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test
:check
:build
BUILD SUCCESSFUL
Total time: 5.607 secs
```

Gradle legt die erzeugten Artefakte standardmäßig im Unterverzeichnis *build* ab. Für ein Java-Projekt findet der Benutzer dort unter anderem die Verzeichnisse *classes*, *reports*, *test-results* und *libs*. Dort landen die erzeugten Java-Klassen, die Ergebnisse der Unit-Tests und ein generiertes Java-Archiv (JAR) für das Projekt. Mit der so erstellten Datei lässt sich zunächst der Bau des Projekts analog zu Mavens *pom.xml* durchführen.

Listing 1: *pom.xml* (Maven)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.iternum.demo</groupId>
  <artifactId>SimpleAccountInfo</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>SimpleAccountInfo</name>
  <url>http://maven.apache.org</url>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Mithilfe des Maven-Plug-in

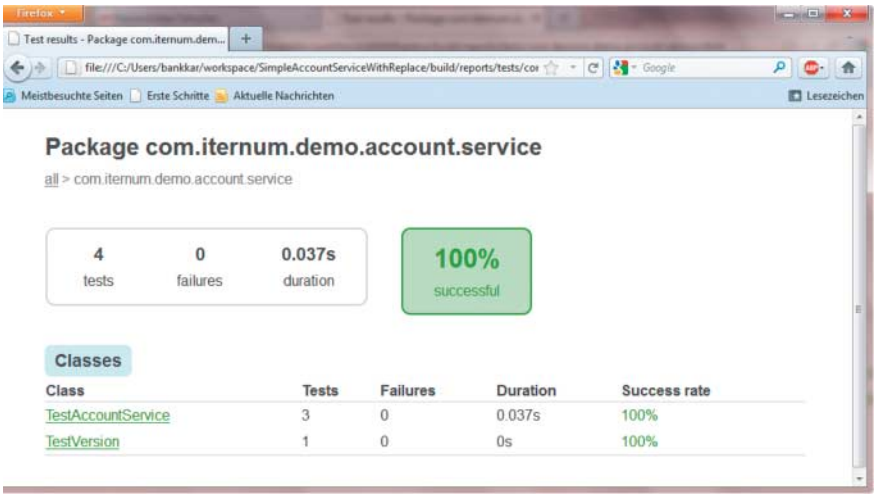
Auch das Deployment des erstellten Archivs in ein lokales oder zentrales Repository unterstützt Gradle. Hierzu dient – im Falle von Maven – das Maven-Plug-in. Es erstellt automatisch eine *pom.xml*, in der die benötigten Abhängigkeiten des erzeugten Artefakts definiert sind. Dafür muss man die folgenden Zeilen in *build.gradle* einfügen, mit denen man mit Gradle den vollen von Maven gewohnten Funktionsumfang erreicht.

```
apply plugin: 'maven'
project.group="com.iternum" project.artifactId= ⤷
      "SimpleAccountBean" project.version="1.0.0"
```

Die etwas kompakteren Build-Skripte rechtfertigen für sich sicherlich nicht den Einsatz eines neuen Build-Systems. Gradles eigentliche Stärke kommt zum Tragen, wenn spezifische Anpassungen des Bauprozesses erforderlich werden. Gradle-Build-Skripte sind nichts anderes als Programme in einer DSL, die sich stark an die Sprache Groovy [f] anlehnt. Ein Gradle Build gliedert sich in die Ausführung einzelner Tasks. Folgender Code definiert eine einfache Task und fügt eine Action hinzu, die beim Aufruf eine Ausgabe erzeugt.

iX-TRACT

- Das Open-Source-Werkzeug Gradle bündelt die Vorteile von Maven und Ant – Dependency Management und Convention over Configuration – und verfügt über eine eigene domänenspezifische Sprache.
- Über Java hinaus kann Gradle zum Bau von Projekten in anderen Sprachen dienen; es gibt Plug-ins für Scala und Groovy.
- Was dem Build-System noch fehlt, ist eine Einbindung in Eclipse und NetBeans.



Report mit dem Test einer neu erzeugten Methode zur Ermittlung der Versionsnummer, wie es die am Ende von Listing 3 stehenden Codebestandteile bewirken.

```
task meinTask << {
    println 'Hello World'
}
```

Tasks bestehen generell aus einer Liste von Actions, die durch Closures definiert werden können. Das Task-Objekt hat eine API, die es ermöglicht, diese Aktionsliste zu manipulieren. Hierzu dienen die Methoden *task.doFirst* und *task.doLast* sowie dessen Alias <<. Da alle existierenden Tasks im Build-Skript als Property direkt zugreifbar sind, ist dies denkbar einfach.

```
meinTask.doFirst {
    println 'Here we go'
}
meinTask.doLast {        println ‚That\'s all folks'
}
meinTask << {
    println 'Good Bye'
}
```

Die Ausführung von *meinTask* führt nun die Liste der Aktionen der Reihe nach aus:

```
C:\> gradle meinTask
:meinTask
Here we go
Hello World
That's all folks
Good Bye
```

Eine Task kann von der Ausführung einer anderen abhängig sein:

```
task deinTask (dependsOn: 'meinTask') << {
    println 'I depend'
}
```

Ant-Anwendern ist dieses Prinzip bekannt. Gradle-Tasks ähneln, was die Verwaltung von Abhängigkeiten angeht, stark den von Ant bekannten *<target>.. </target>*-Elementen. Im Gegensatz zu diesen sind sie jedoch deutlich flexibler. So können sie beliebigen Quellcode enthalten, während man in Ant lediglich auf die jeweils verfügbaren Ant-Tasks zurückgreifen kann. Benötigt man nun eine weitere Aufgabe im Projekt, kann man die zunächst projektspezifisch im Build-Skript definieren und zu einem späteren Zeitpunkt in eine wiederverwendbare Komponente extrahieren. Der Bauprozess wird dadurch agil. So lassen sich schnell neue Teststrategien oder Generierungsschritte einführen.

Existierende Tasks kann man erheblich leichter erweitern – etwa durch Hinzufügen von Closures mit den *doFirst*- und *doLast*-Methoden das Verhalten einer Task ergänzen. Die Abhängigkeiten einer Aufgabe lassen sich ebenfalls manipulieren. Dies gilt selbstverständlich nicht nur für die selbst geschriebenen, sondern für alle. Schließlich kann man ja neue Tasks durch Ableitung von existierenden erzeugen. Im Beispiel sei ein einfacher *Code Generator* definiert. Der soll aus Templates Quellcode-Dateien in einem temporären Verzeichnis erzeugen. Dieses wiederum soll Bestandteil des normalen Generierungsprozesses sein.

Im vorliegenden vereinfachten Code muss die *AccountService*-Klasse eine generierte Methode enthalten, die die aktuelle Version des Projekts zurückliefert. Dazu versehe man die Klassendatei im normalen Quellcode-Verzeichnis mit einer neuen Endung *.template*, sodass sie vom normalen Kompilierungsprozess ausgenommen bleibt. Das Template soll folgenden Codeblock enthalten:

```
public String getVersion() {
    return "${version}";
}
```

Nun muss man den Bauprozess so erweitern, dass das Erzeugen der Quellcode-Dateien vor dem Kompilieren der Quellen stattfindet. Dies bewirken die am Ende von Listing 3 stehenden Codebestandteile. Die Abbildung zeigt den Report mit dem Test der neu erzeugten Methode zur Ermittlung der Versionsnummer.

Fazit

Schon das hier Dargestellte verdeutlicht, dass Gradle geeignet ist, die Vorteile der verbreiteten Build-Werkzeuge Ant und Maven zu kombinieren und ihre für sich

Listing 2: Fehler wegen Maven-Abhängigkeiten

```
C:\Users\bankkar\workspace\SimpleAccountService\src\main\java\com\iternum\demo\account\service\AccountService.java:17:
                                                                                                    cannot find symbol
symbol  : variable LogFactory
location: class com.iternum.demo.account.service.AccountService
        private Log log = LogFactory.getLog(AccountService.class);
                          ^
4 errors

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':compileJava'.
Cause: Compile failed; see the compiler error output for details.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug
option to get more log output.

BUILD FAILED

Total time: 3.524 secs
```

iX-Wertung

- ⊕ Verschmelzung von Ant- und Maven-Eigenschaften
- ⊕ kompaktere und besser lesbare Skripte
- ⊕ flexible Erweiterbarkeit
- ⊕ gute Dokumentation
- ⊖ fehlende Integration in IDEs (Eclipse und NetBeans)

Listing 3: Vor dem Kompilieren Quellcode erzeugen

```
apply plugin: 'java'
apply plugin: 'maven'

project.group="com.iternum"
project.artifactId="SimpleAccountBean"
project.version="1.0.0"

repositories {
    mavenCentral name: "jboss", urls: "http://repository.jboss.org/nexus/content/groups/public-jboss/"
}

dependencies {
    compile group: 'commons-logging', name: 'commons-logging', version: '1.1.1'
    testCompile group: 'junit', name: 'junit', version: '3.8.1'
}

sourceSets {
    main {
        java {
            srcDir 'build/generated/java'
        }
        resources {
            srcDir 'build/generated/resources'
        }
    }
}

task generateSources(type: Copy) {

    from 'src/main'
    into project.buildDir.toString() + '/generated/java'
     include '**/*.template'
     expand(project.properties)
    // Use a closure to map the file name
    rename { String fileName ->
        fileName.replace('.template', '.java')
    }
}

compileJava.dependsOn generateSources
```

genommen schmerzhaften Nachteile zu beheben. Von Maven erbt Gradle das Dependency- und Artefakt-Management und den konsequenten Convention-over-Configuration-Ansatz bis hin zum standardisierten Projekt-Layout.

Von Ant kommen unter anderem viele bekannte Funktionen, die Organisation des Projekts durch voneinander abhängige Tasks und die hohe Flexibilität in der Anpassung des Bauprozesses. Die Abkehr von XML zur Definition der Build-Skripte führt zu besser lesbaren und kompakteren Skripten, die sich schnell und flexibel erweitern lassen.

Über das hier Diskutierte hinaus gibt es zahlreiche weitere interessante Aspekte von Gradle. Die Verwaltung voneinander abhängiger Projekte ist gut gelungen und komfortabler als der von Maven bekannte Ansatz. Außerdem lassen sich in Gradle die dynamischen Aspekte von Groovy voll ausnutzen. So können Tasks dynamisch zur Laufzeit des Baus erzeugt und geradezu beliebig manipuliert werden. Gradle kann darüber hinaus nicht nur zum Bau von Java-Projekten dienen, sondern für beliebige andere Sprachen; es gibt beispielsweise Plug-ins für Scala und Groovy. Der derzeit vielleicht größte Schwachpunkt von Gradle ist die fehlende Integration in populäre IDEs wie Eclipse oder NetBeans. Zwar bietet Gradle Plug-ins, die Projekte für verschiedene gängige IDEs erstellen können. Der komfortable Aufruf von Gradle aus einer IDE heraus ist jedoch derzeit nur über Umwege möglich und noch nicht mit der Integration vergleichbar, die für Ant oder Maven erhältlich ist.

Davon abgesehen bietet Gradle ein ausgereiftes und elegantes Build-System, das den Vergleich mit Ant und Maven nicht zu scheuen braucht und teilweise weit über diese hinausgeht. Sowohl die Migration existierender Ant- oder Maven-Builds als auch die Integration in existierende Infrastrukturen unterstützt Gradle gut. Zusammen mit der hervorragenden Dokumentation ist hier ein ernsthafter Konkurrent zu den etablierten Systemen entstanden. (hb)

Onlinequellen

[a] Ant
ant.apache.org

[b] Maven
maven.apache.org

[c] Gradle
www.gradle.org

[d] Gradles Java-Plug-in
www.gradle.org/java_plugin.html

[e] Ivy
http://ant.apache.org/ivy/

[f] Groovy-Homepage
groovy.codehaus.org/

KARL BANKE

ist Softwarearchitekt bei der iternum GmbH in Frankfurt/Main. Tätigkeitsschwerpunkte sind Java-EE-Anwendungen und SOA.

Alle Links: www.ix.de/ix1109078