Sonderdruck aus JavaSPEKTRUM

Alle Macht dem Releaser

Maven-Projekte dem Dependent Graph Releaser überlassen

Robert Stoll

Hunderte Projekte auf dem aktuellsten Stand zu halten, bedingt oft großen manuellen Aufwand. Maven selbst bietet dafür zwar Instrumente an, diese stoßen aber schnell an ihre Grenzen. Abhilfe schafft der Dependent Graph Releaser.

Abhängigkeiten von Projekten lassen sich mittels Dependency-Management, Parent POMs oder BOM POMs verwalten. Bei der Aktualisierung von Versionen passiert es jedoch schnell, dass Projekte vergessen werden, die mit neuen Versionen aktualisiert werden müssten. Die Folge davon sind unentdeckte Fehler und nicht rückwärts kompatible Änderungen. Diese tauchen erst sehr viel später an den entsprechenden Stellen auf, als den meisten lieb wäre. Ob der entsprechende Verursacher dann noch verfügbar ist, ist ungewiss.

Verwaltung von Dependencies via Maven-Instrumente

Mittels `<dependencyManagement>` [Mvn1] besteht die Option, Abhängigkeiten zu konfigurieren, ohne dass diese verwendet werden. Die Verwendung als solches findet wie gewohnt unter `<dependencies>` statt. Dabei genügt es, `groupId` und `artifactId` zu definieren, da der Rest ja bereits im Dependency-Management spezifiziert ist (s. Listing 1). Werden die Abhängigkeiten an mehreren Orten im POM (beispielsweise in verschiedenen Profilen [Mvn2]) verwendet, wird die Version dennoch an einem einzigen Ort verwaltet.

Dependency-Management ermöglicht aber mehr als nur die Definition einer einheitlichen Version. Grundsätzlich lässt sich alles konfigurieren, was unter `<dependencies>` auch möglich ist. Zum Beispiel den `<scope>` festlegen oder transitive Dependencies mittels `<exclusions>` ausschließen. Die eigentliche Stärke des Dependency-Managements kommt meiner Meinung nach erst zum Tragen, wenn man die Version einer Abhängigkeit über mehrere Projekte hinweg zentralisiert verwalten möchte. Parent POMs stellen hier eine Option dar.

Parent POMs – Vererbung

Parent POMs [Mvn3] ermöglichen es, Definitionen an ihre Kinder zu vererben. Sollen eine oder mehrere Abhängigkeiten über verschiedene Projekte hinweg auf dem gleichen Stand gehalten werden,

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>ch.tutteli.atrium</groupId>
      <artifactId>atrium-cc-en_GB-robstoll</artifactId>
      <version>0.7.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>ch.tutteli.atrium</groupId>
    <artifactId>atrium-cc-en_GB-robstoll</artifactId>
  </dependency>
</dependencies>
```

Listing 1: Verwendung von Dependency-Management

Robert Stoll arbeitet als Software-Engineer bei der Löwenfels Partner AG. Nebst Konzepten entwerfen/implementieren widmet er sich der Produktivitätssteigerung im Entwicklungsprozess. Zudem hat er den Lead bei Open-Source-Projekten. Privat ist er Autor von Atrium, einer Assertion-Library für Kotlin.
E-Mail: info@loewenfels.ch

genügt es, für diese Projekte einen gemeinsamen Parent festzulegen (s. Listing 2). Im Parent werden die entsprechenden Abhängigkeiten im Dependency-Management definiert und in den Projekten wiederum unter `<dependencies>` verwendet. Falls nötig, können in den Projekten auch Werte überschrieben werden. Zum Beispiel lässt sich der `<scope>` von `compile` auf `runtime` stellen.

Bei der Verwaltung von ein paar Dutzend Projekten tritt dann oft folgende Situation auf: Eine Abhängigkeit wird bei Projekt A, B und C in der Version 1.0 verwendet, gleichzeitig auch in der Version 2.0 in den Projekten D, E und F. Vermutlich haben die Projekte A, B und C noch weitere Gemeinsamkeiten (ebenso die Projekte D, E und F). Der Einfachheit halber fasse ich Projekte mit Gemeinsamkeiten unter dem Begriff Projektpalette zusammen.

Es stellt sich die Frage, ob je Projektpalette ein eigener Parent eingeführt werden soll. Gleichzeitig soll vermieden werden, dass Projektpaletten übergreifende Gemeinsamkeiten nicht mehr zentral verwaltet werden. Ein Parent dieser Parents muss her. Ein Parent für Projektpalette ABC und ein Parent für Projektpalette DEF wird erstellt. Gemeinsamkeiten werden im Parent der Parents definiert.

Dieser Ansatz mag bei ein paar Dutzend Projekten noch gut gehen, doch je mehr Projekte verwaltet werden, desto mehr gerät man in die Bredouille. Der Aufbau einer ganzen Parent-Hierarchie ist nur noch schwer zu verwalten: Ändert sich etwas im Parent aller Parents, muss die Version dieses Parents in den Parents darunter nachgezogen werden. Ebenso muss die Version dieser Parents in den darunterliegenden Parents usw. aktualisiert werden, bis man zu den Projekten selbst gelangt. Es mag sich der eine oder andere Mehrfachvererbung wünschen, doch ein solches Konzept existiert in Maven nicht. Abhilfe schafft hier aber ein BOM POM.

BOM POMs – Komposition

BOM steht für *Bill Of Materials* und ein BOM POM zeigt in erster Linie kein Unterschied zu einem normalen Project Object Model (kurz POM), – davon abgesehen, dass üblicherweise `<packaging>pom</packaging>` verwendet wird. Zieht man den Vergleich zu Parent POMs und Vererbung, entsprechen BOM POMs [Mvn4] der Komposition. Auch bei der Verwaltung von Abhängigkeiten empfiehlt es sich, dem Grundsatz *Composition over Inheritance* [GoF94] (dt. Komposition vor Vererbung) aus der objektorientierten Programmierung zu folgen. Ein BOM POM unterscheidet sich von einem Parent POM nicht im Wesentlichen, allerdings in dessen Verwendung. Denn ein BOM POM wird ins `<dependencyManagement>` importiert (s. Listing 3), wodurch die deklarierten Abhängigkeiten im BOM POM übernommen werden. Im Gegensatz zum Parent POM können mehrere BOM POMs importiert werden.

Das zuvor beschriebene Problem einer Parent-Hierarchie kann so weitgehend eliminiert werden. Dazu wird pro Projektpalette ein BOM definiert sowie ein weiteres BOM für gemeinsam verwendete Abhängigkeiten. Hier gilt es jedoch auf die Reihenfolge der Importierungen zu achten, denn bei BOM POMs gilt: Wer zuerst kommt, mahlt zuerst (siehe Kasten für weitere Details).

```
<parent>
  <groupId>ch.loewenfels</groupId>
  <artifactId>loepa-parent</artifactId>
  <version>2.6.1</version>
</parent>
```

Listing 2: Definition eines Parent POMs

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>ch.loewenfels.eahv.services</groupId>
      <artifactId>nil-services-api</artifactId>
      <version>1.5.7</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Listing 3: Import eines BOM POMs

Am Rande sei bemerkt, dass POMs auch direkt in `<dependencies>` importiert werden können. Dabei werden die Abhängigkeiten nicht nur konfiguriert (wie es beim Import in `<dependencyManagement>` der Fall ist), sondern auch direkt verwendet. Definiere ich beispielsweise ein BOM POM für Tests mit Abhängigkeiten zu JUnit, Mockito und AssertJ, muss ich am Ende nur dieses eine POM importieren.

Zurück zum Problem der Parent-Hierarchie. Mittels BOM POMs wird zwar die Hierarchie aufgetrennt, eine Abhängigkeit zu den BOMs bleibt aber bestehen. Sprich: Wenn ich eine Änderung im BOM der Projektpalette A durchführe, muss ich die neue Version in allen Projekten aktualisieren, die dieser Palette angehören. Das *Single Responsibility Principle* [Mar03], zu Deutsch Prinzip der einzigen Verantwortung, kann auch hier die Wogen glätten.

Schauen wir uns das genannte BOM POM für Tests als Beispiel an. Neben JUnit, Mockito und AssertJ hätte ich auch alle weiteren Abhängigkeiten, die irgendwo in einem Projekt für `<scope>test</scope>` definiert wurden, aufführen können. Stattdessen habe ich mich auf die Abhängigkeiten für die Durchführung von Unittests konzentriert. Diese Variante ist natürlich anzustreben. Ein weiteres BOM POM könnte eines für GUI-Tests sein, das dann beispielsweise eine Abhängigkeit zu JUnit und Selenium definiert.

Der Vorteil der BOM POMs, die nur für eine Verantwortung beziehungsweise einen Aspekt definiert wurden, scheint einleuchtend. Bei einer Änderung in einem dieser POMs müssen nur jene

Wer zuerst kommt, mahlt zuerst

Welche Definition über mehrere POMs hinweg Vorrang vor anderen Definitionen hat, ist kompliziert. Folgend eine nicht abschließende Reihenfolge:

- Definitionen aus dem POM des Projekts, wobei bei mehreren die letzte Definition gewinnt.
- Definitionen im Parent POM, wobei bei mehreren die erste gewinnt. Zudem werden Definitionen weiter oben in der Hierarchie (z. B. im Parent des Parents) zuerst berücksichtigt.
- Definitionen in BOM POMs, wobei bei mehreren die erste gewinnt. Zudem wird die Reihenfolge der Imports bei mehreren BOM POMs berücksichtigt. Hier gilt weiter, dass Imports weiter oben in der Hierarchie (z. B. im Parent) Vorrang haben.

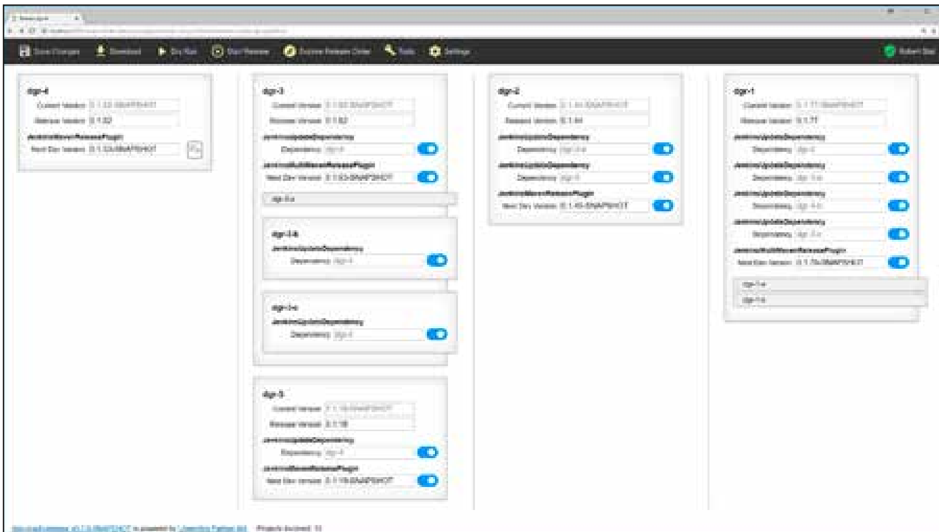

Abb. 1: Release-Pipeline

Projekte aktualisiert werden, die von diesem Aspekt Gebrauch machen. Das steht im Gegensatz zu einer Änderung an einem BOM POM, das alle Test-Abhängigkeiten enthält, da in diesem Fall wahrscheinlich alle Projekte davon betroffen sind.

Dennoch bleibt die Krux mit den Abhängigkeiten bestehen: Sie müssen verwaltet werden. Daran kommt man auch dann nicht vorbei, wenn man sich das Single Responsibility Principle zu Herzen nimmt. Heißt: Bei einer neuen Version einer Abhängigkeit muss entweder diese oder die Version des Parent POMs oder des BOM POMs nachgezogen werden. Und genau bei diesem schmerzlichen Punkt greift einem der Dependent Graph Releaser unter die Arme.

Dependent Graph Releaser

Das Projekt Dependent Graph Releaser, etwas kürzer dep-graph-releaser und im Weiteren einfach nur DGR genannt, wurde von dem Luzerner Softwareunternehmen Löwenfels Partner AG lanciert. Maßgeblicher Grund dafür war die Tatsache, dass die Verwaltung der unzähligen Projekte zu viel manuellen Aufwand nach sich gezogen hatte. Wir gingen davon aus, mit dem Problem nicht allein dazustehen. Deshalb veröffentlichten wir das Kotlin Multi-Plattform basierte Projekt unter der Lizenz *EUPL 1.2* (eine copyleft kompatible Lizenz) auf GitHub [GitH-a].

Der DGR sucht in einem gegebenen Ordner nach Maven-Projekten und analysiert die Abhängigkeiten der Projekte untereinander. Anhand dessen erstellt er einen Releaseplan für ein gewünschtes Projekt in Form eines Graphen und serialisiert diesen in eine JSON-Datei. Die Datei wiederum wird von einer Web-App deserialisiert und entsprechend im Browser als Release-Pipeline dargestellt (s. Abb. 1, auch in Groß unter [GitH-b] anzusehen und auszuprobieren).

Das Projekt, welches in Betrieb gehen soll, steht in der Spalte ganz links. Projekte, die davon abhängen, in den Spalten weiter rechts. Ein Projekt, das weiter rechts steht, hat jeweils mindestens eine Abhängigkeit auf ein Projekt in der Spalte links davon. Der DGR bedingt eine Integration mit einem oder mehreren Jenkins-Instanzen, um den Release durchführen zu können. Die visuelle Darstellung ist aber auch bereits so hilfreich. Der Betrachter sieht, welche Projekte von welchen abhängen und wo die Versionen verwaltet werden.

Im Beispiel in Abbildung 1 ist ersichtlich, dass das Projekt dgr-3 die Abhängigkeit zu dgr-4 an drei Orten pflegt: einmal im Multi-Module dgr-3 und dann noch in den Submodulen dgr-3-b und dgr-3-c (Submodule sind als Boxen innerhalb eines Multi-Modules dargestellt). Dies schreit förmlich nach einer Verschiebung ins Dependency-Management. Die visuelle Darstellung ist auch dann hilfreich, wenn der Release manuell vorgenommen werden muss. In diesem Fall zeigt der DGR, in welcher Reihenfolge die Projekte veröffentlicht werden müssen und wo Abhängigkeiten nachgezogen werden sollten. Sinn des DGR ist aber primär, dass der Release-Prozess automatisiert durchgeführt werden kann.

Integration mit Jenkins

Damit der DGR den Release automatisiert durchführen kann, muss er mit einem Jenkins-Build-Server verbunden werden. Schließlich stößt der DGR über REST-Aufrufe bestehende Jobs auf Jenkins an, die wiederum den Release durchführen. Der DGR überwacht mittels Polling die Jobs und stößt bei Erfolg die entsprechenden Folgejobs asynchron (und dadurch parallel) an. Als Beispiel: Wenn dgr-4 in Abbildung 1 veröffentlicht wurde, wird gleichzeitig bei allen Projekten, die dgr-4 verwenden – sprich dgr-3, dgr-5 und dgr-1 –, der entsprechende Aktualisierungsjob gestartet. Dieser muss auf dem Jenkins gesondert eingerichtet werden [GitH-c] und verwendet ein Artefakt vom DGR, um die Aktualisierung durchzuführen.

Möchte man die *Dry Run*-Funktionalität (dt. Trockenlauf) verwenden, so muss dafür ein weiterer Job eingerichtet werden. Bei einem Dry Run werden Projekte nur in ein lokales Maven-Repository installiert und geprüft. Sprich kein commit, kein git tag, kein Deploy – einfach nur ein Trockenlauf.

Unterschiede zur Jenkins-Pipeline

Jenkins selbst bietet auch die Möglichkeit, eine Pipeline einzurichten. Wieso sollte ich trotzdem den DGR verwenden? Dafür sprechen gleich mehrere Gründe: Einerseits ist die Visualisierung deutlich besser, andererseits gibt es wesentliche Unterschiede in der Funktionalität.

Dry Run als zusätzliches Feature wurde bereits genannt. Zudem lässt sich der DGR mit mehreren Jenkins-Instanzen integrieren. Die Pipeline präsentiert sich in einem solchen Fall für den Benutzer nicht anders. Welche Jobs auf welcher Jenkins-Instanz ausgeführt werden sollen, wird unter *Settings* mittels `remoteRegex` festgelegt. Dadurch können wir bei Löwenfels Partner AG die Migration von Jenkins 1.x auf Jenkins 2.x schrittweise durchführen. Zu Schulungszwecken kann man den Release-Prozess auch im Simulationsmodus starten und bewusst einzelne Schritte fehlschlagen lassen.

Weitere Features finden sich unter Tools und mittels Rechtsklick auf ein Projekt. So bietet der DGR die Möglichkeit an, eine Liste von git clone commands für die involvierten Projekte zu generieren. Oder eine Eclipse-PSF-Datei, damit die Projekte gleich ins Eclipse importiert werden können.

Flexibler Prozess

Einer der wichtigsten Unterschiede zur Jenkins-Pipeline besteht in der flexiblen Gestaltung des Prozesses. Dadurch bleibt der DGR ein

unterstützendes Werkzeug, ohne die Anwender dabei unnötig einzuschränken. Zum Beispiel: Aktuell unterstützt der DGR noch keine Versionsdefinitionen, die aus mehreren Maven Properties bestehen (z. B. `<version>${a.major}.${a.minor}</version>`). Ist in einem Projekt eine Abhängigkeit so definiert, schlägt zwar der Aktualisierungsjob mit einem entsprechenden Hinweis fehl, der Prozess ist aber dennoch nicht gescheitert. In einem solchen Fall kann man manuell die Version nachziehen und dem DGR entsprechend mitteilen, dass das Problem eigenhändig gelöst wurde. Danach kann der Prozess wieder gestartet werden – als wäre nichts gewesen.

Ein weiteres Beispiel zur Beleuchtung dieser Stärke: Angenommen, ein Jenkins-Build schlägt fehl, obwohl der Release als solches erfolgreich war. Der Grund für den Fehlschlag kann vielfältig sein: Out-of-Memory, IOException (z. B. Jenkins-Slave lief in ein Timeout) und weitere. In einem starren Prozess müsste der Release nochmals gebaut werden — obwohl der git tag womöglich bereits erstellt und die Artefakte deployt wurden. Beim DGR ist die Kontrolle primär beim Benutzer, auch hier kann dieser dem DGR mitteilen, dass der Release erfolgreich war und der Prozess fortgesetzt werden kann.

Fazit

Der Dependent Graph Releaser kann auf verschiedene Weise unterstützend im Release-Prozess eingesetzt werden: angefangen von der Visualisierung eines Releaseplans bis zum automatischen Durchführen von kaskadierenden Releases inklusive Aktualisierungen der Abhängigkeiten. Durch die flexible Prozessgestaltung bleibt einerseits die Kontrolle beim Nutzer und andererseits ergeben sich dadurch Interaktionsmöglichkeiten, die man sich bei anderen Werkzeugen wünschen würde. Einziger Wermutstropfen: Der Dependent Graph Releaser bedingt momentan noch eine Integration mit einem oder mehreren Jenkins-Instanzen. Die Integration anderer Build-Server und Werkzeuge ist aber vorstellbar und Feature-Requests sind willkommen.

Literatur und Links

[GitH-a] Projektwebsite des DGR,
https://github.com/loewenfels/dep-graph-releaser

[GitH-b] Online-Beispiel des DGR,
https://loewenfels.github.io/dep-graph-releaser/#release.json

[GitH-c] https://github.com/loewenfels/dep-graph-releaser/wiki/Jenkins-Server

[GoF94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[Mar03] R. C. Martin, Agile Software Development – Principles, Patterns, and Practices, Prentice Hall, 2003

[Mvn1] Dependency Management, https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Management

[Mvn2] Maven Profile, https://maven.apache.org/guides/introduction/introduction-to-profiles.html

[Mvn3] Parent POM, https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Project_Inheritance

[Mvn4] BOM POM, https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Importing_Dependencies