Konfigurationsmanagement mit Maven 2

Michael Albrecht, Manfred Wolff

Michael Albrecht ist Chefarchitekt bei der NEUSTA GmbH und seit 2002 mit der Entwicklung und der Architektur von Java EE Projekten beschäftigt. E-Mail: m.albrecht@neusta.de

Manfred Wolff ist freiberuflicher Berater, technischer Projektleiter und Architekt im Bereich Java EE. In der letzten Zeit unterstützt er die Realisierung von Offshore Projekten z.B. mit Indien und in Tunesien. E-Mail: wolff@manfred-wolff.de

Wie oft haben wir nach Auslieferung eines Projekts gedacht: Wie schön wäre es jetzt noch einmal alles von vorn zu beginnen. Was bei kommerziellen Produkten keine gängige Praxis ist, kommt bei Open Source schon mal öfters vor: So geschehen bei Maven. Die erste Version war noch nicht in der Version 1.0 veröffentlicht, da wurde schon mit der Version 2.0 begonnen [MAVEN].

”Some code was moved to shared libraries and is now used by both, but much was written from scratch (even before Maven 1.0 was released). It was certainly inspired by the original, but is different.” (Brett Porter)

Wir wollen Maven 2 vorstellen und dabei die Frage beantworten: Lohnt sich der Umstieg von Maven 1 auf Maven 2?

Konfigurationsmanagement

Gerade in größeren Projekten ist es schwierig zu gewährleisten, dass alle Entwickler mit den gleichen, oder vergleichbaren Versionen, von eigenen Komponenten oder Fremdbibliotheken arbeiten. Das Problem der Fremdbibliotheken ist noch in den Griff zu bekommen, bei eigenen Komponenten, die auch von mehreren Entwicklern entwickelt und womöglich in unterschiedlichen Versionen benötigt werden und an unterschiedlichen Standorten entwickelt werden (z.B. Offshore), werden die Probleme schnell sichtbar:

- In der persönlichen Entwicklungsumgebung sind andere Bibliotheken eingestellt, als in der Produktionsumgebung (Zitat Enwickler: ”Bei mir geht es aber“).
- Das Deployment ist fehleranfällig weil händisch durch Kopieren Programmstände und Bibliotheken kopiert werden.
- Codestylefehler häufen sich und werden ab einer gewissen Häufigkeit nicht mehr berücksichtigt.
- Tests werden nur dann geschrieben wenn Zeit ist und zum Ende des Projekts mehren sich ungetestete Klassen, oder Tests sind nicht mehr gültig (veraltet).

Wenn wir von Konfigurations-Management sprechen, meinen wir die Anwendung von Standards und Vorgehensweisen zum Management einer sich entwickelnden Software. Dabei geht es um verschiedene Elemente in der Software-Entwicklung.

- *Build-Management*: Hier sind vor allem die Abhängigkeiten von Bibliotheken zu betrachten. Mit dem Open Source Framework Ant kann der Buildvorgang automatisiert werden. Durch den Einsatz von Maven können auch Abhängigkeiten von bestimmten Versionen einer

Software beschrieben und automatisiert werden, mit Maven 2 sogar transitive Abhängigkeiten. Dabei wird zwischen in Entwicklung befindlichen Versionen (`SNAPSHOT`) und ausgelieferten Versionen unterschieden.

- *Versions-Management*: Um konsistente Softwarestände reproduzierbar zu machen ist neben der Verfügbarkeit eines zentralen Repositories striktes Versionsmanagement wichtig. Dabei ist genau abzugrenzen, welche Systematik der Erhöhung von Major- oder Minornummerierungen zu Grunde liegt.
- *Release-Management*: Ein Release beschreibt einen Auslieferungsstand einer Software mit allen Bestandteilen. Hierzu gehört auch die Featureliste, die FAQ etc. All diese Dokumente, die zu einem Release gehören, so auch das Announcement des Releases, können mit Maven sehr gut beschrieben und gepflegt werden. Dabei bietet Maven eine einfache XML-Syntax an, um Dokumente zu erstellen und ab Maven 2 auch ein Wiki-ähnliches Format - APT (Almost Plain Text).
- *Deployment-Management*: Das Deployment-Management stellt sicher, dass immer nur konsistente Stände der Software auf die Server verteilt werden. Oft besteht ein Deployment nur aus dem Kopieren von Dateien auf den Server. Aber auch hier sind beliebig viele Fehlerquellen auszumachen.
- *Reporting*: Besonders bei Open Source Projekten, bei denen die Entwickler nie zu Meetings zusammenkommen, ist das Reporting ein wichtiges Hilfsmittel zur Qualitätskontrolle. Hierzu gehören Reports zu Umfang und Qualität von Junit-Tests, Metrikanalysen, styleguide Konformität etc.

Installation

Für die oben genannten Probleme bietet Maven 2.0 Lösungen an.

Im Kern bietet Maven zwei Dinge an:

- Ein Modell des Softwareprojekts in Form einer XML Datei (POM Project Object Model)
- Eine Menge von Tools, die auf diesem Object Model arbeiten.

Um Maven 2 zu installieren sind zunächst genau zwei Schritte notwendig:

1. Maven muss nach dem Download entpackt werden und zwar in einem Verzeichnis der Wahl.
2. Die Umgebungsvariable `M2_HOME` muss auf dieses Verzeichnis gelegt werden und das `${M2_HOME}/bin` Verzeichnis muss in den Pfad aufgenommen werden.

Mit Hilfe der Eingabe `mvn --version` kann nun direkt überprüft werden, ob die Installation erfolgreich war. Normalerweise kann jetzt sofort losgelegt werden. Falls Der Rechner hinter einer Firewall liegt, muss noch eine `settings.xml` erstellt werden, die im user home liegt (`~/.m2/` oder unter Windows `C:\Dokumente und Einstellungen\[user name]/.m2/`) Maven stellt zwei Konfigurationsdateien zur Verfügung:

- *pom.xml*: Diese Datei umfasst die gesamte Projektbeschreibung. Ein Schwerpunkt bildet dabei die Anhängigkeiten von externen Bibliotheken. Da auch transitive Abhängigkeiten automatisch aufgelöst werden (dazu später mehr) können im Abhängigkeitsgraf auch explizit Abhängigkeiten ausgeschlossen werden. In der `pom.xml` werden aber auch alle

anderen Projektinformationen hinterlegt wie Modulversionen, Konfigurationseinstellungen für Module und Plugins, Projektinformationen wie Comitter und Contributer, Repositoryeinstellungen und vieles mehr. POMs können vererbt werden.

- *settings.xml*: In dieser Datei können Einstellungen überladen werden, die Maven als Konvention annimmt. Liegt Ihr Rechner hinter einer Firewall, so sind die Proxy-Einstellungen auch hier vorzunehmen (siehe Kasten settings.xml).

Wie bereits unter Maven1 kann jetzt ein Anwendungsrumpf erstellt werden, der sofort lauffähig ist. [GENAPP]
Wenn man sich an die Maven-Konventionen hält, werden Konfigurationsdateien gespart. Beispiel: Maven legt standardmäßig das Repository unter `~/.mvn/repository` an. Wem es nicht stört, braucht nichts zu machen, ansonsten kann dieser Default überladen werden, es werden zusätzliche Einträge in der `pom.xml` nötig. Ein weiteres Beispiel: Maven geht davon aus, dass die Sourcestruktur `src/main/java` ist. Hält man sich an diese Konvention, ist nichts zu tun. Soll die Struktur geändert werden, so muss wiederum die `pom.xml` erweitert werden, die diese Konvention überlagert.

Tip: Wir haben uns angewöhnt lieber bestehende Projekte auf die Maven Konventionen zu migrieren (mit Eclipse und Refactoring ist dieses einfach möglich), als umständlich Konfigurationsdateien zu pflegen. Da alle unsere Projekte ”mavinifiziert“ sind, ist es für jeden Entwickler sehr einfach, sich in neue Projekte einzuarbeiten. Grundsätzliche Konventionen bei Maven sind:

- Ein Standard Directory Layout für alle Projekte.
- Ein Output für ein Projekt (nur ein `jar` oder ein `war` pro Projekt).
- Standardisierte Namenskonventionen.

Repository und Dependencies

Dependencies sind wohl das Kernproblem im Konfigurationsmanagement einer Software Anwendung.

Im Repository von Maven 2 sind die Abhängigkeiten wie folgt aufgelöst:.

```
<dependency>
   <groupId>gruppenname</groupId>
   <artifactId>artefaktname</artifactId>
   <version>versionsnummer</version>
</dependency>
```

Dann wird im lokalen Repository (im folgenden: `MAVEN LOCAL REPO`) die Abhängigkeit in folgendem Verzeichnis gesucht:

```
MAVEN LOCAL REPO/gruppenname/artefaktname/versionsnummer
```

In diesem Verzeichnis finden sich nach ordentlichem Download dann wenigstens vier Dateien:

```
artefaktname-versionsnummer.jar
artefaktname-versionsnummer.jar.sha1
artefaktname-versionsnummer.pom
artefaktname-versionsnummer.pom.sha1
```

Die sha1-Dateien werden als synchrone Schlüssel für die Gleichheit und Korrektheit der Dateien verwendet. Schließlich möchte man sich keine schadhaften oder beschädigten Dateien übers Netz

herunterladen.

Das Java Archiv `artefaktname-versionsnummer.jar` selbst ist ja das eigentliche Objekt der Begierde. Die Projektbeschreibungsdatei `artefaktname-versionsnummer.pom` wird verwendet, um die transitiven Abhängigkeiten aufzulösen.

Nicht nur die Software, die wir selbst schreiben, ist von Bibliotheken abhängig, sondern auch die Bibliotheken sind von anderen Bibliotheken abhängig. Man spricht in diesem Fall von transitiven Abhängigkeiten. Bisher wurden diese transitiven Abhängigkeiten weder von Ant noch von Maven 1 automatisch aufgelöst, sondern mussten manuell nachgetragen werden. Bei der Verwendung von anderen Frameworks wie Struts bedeutete dies, man musste die Liste der transitiven Abhängigkeiten über das Internet nachgooglen, um herauszufinden, welche Bibliothek in welcher Version gebraucht wird. Das kostete nicht nur Zeit, sondern „vermüllte“ nach und nach die Projektkonfiguration, so dass nach Einbindung aller Bibliotheken nicht mehr deutlich wird, welche Version einer Bibliothek direkt und welche indirekt benötigt wurde.

Der Wechsel eines Frameworks oder einer Bibliothek, die direkt benutzt wurde, führte dann unweigerlich dazu, dass die Dependency-Einträge in der Projektbeschreibung schlichtweg falsch wurden. Die gute Nachricht: Maven 2 löst diese transitiven Abhängigkeiten auf. Voraussetzung dafür ist allerdings eine saubere Projektbeschreibungsdatei in dem Repository, aus dem die Abhängigkeit geladen wird. Die Projektbeschreibungsdatei mitzuladen, falls sie nicht existiert, ist ein Automatismus bei Maven 2. Dabei gibt es natürlich auch immer wieder Probleme, wie sich in Abbildung 1 zeigt. Dabei gibt es nun unterschiedliche mögliche Lösungen bzw. Varianten

Maven 2 löst obiges Problem durch einen Vergleich der Versionsnummern. Dieser Vergleich geschieht über den Aufbau einer Versionsnummer nach dem Prinzip:

`1.2.3-20070228.4567-8`
Major.Minor.Bug fix - Qualifier - Build number

Es wird also zuerst über Major verglichen, dann Minor und so weiter. über diesen Vergleich werden als von Maven 2 automatisch höhere Dependencies bevorzugt.

Es können transitive Dependencies, die unerwünscht sind, speziell ausgeschlossen werden. Dazu müssen in der Projektbeschreibungsdatei folgende Einträge vorgenommen werden:

```
<dependency>
  <groupId>commons-cli</groupId>
  <artifactId>commons-cli</artifactId>
  <version>1.0</version>
     <exclusions>
        <exclusion>
           <groupId>commons-logging</groupId>
           <artifactId>commons-logging</artifactId>
        </exclusion>
     </exclusions>
</dependency>
```

In dem angegebenen Fall wird die transitive Abhängigkeit der Bibliothek von `commons-logging` über die Bibliothek `commons-cli` ausgeschlossen, egal welche Version dort benutzt wird. Leider wird schon an der Notation deutlich, dass dieses Ausschließen von transitiven Abhängigkeiten in jedem Einzelfall durchgeführt werden muss. Wenn die Abhängigkeit von einer weiteren Bibliothek ebenfalls geliefert wird, so muss sie ebenfalls ausgeschlossen werden.

Die Angabe einer Versionsnummer bei den Dependencies muss nicht durch einen festen Wert geschehen. Es können auch Versionsnummernbereiche angegeben werden.

Beispiel:

```
<dependency>
  <groupId>commons-cli</groupId>
  <artifactId>commons-cli</artifactId>
   <version>[1.0,)</version>
</dependency>
```

Hierbei bedeutet der Ausdruck `[1.0,)`, dass jede Auflösung mit einer Versionsnummer, die mindestens 1.0 ist, möglich ist. Die Notation folgt hier der aus der Mathematik bekannten Klammernnotation. Es kann also auch eine Version von 1.0 bis 1.1 durch `[1.0, 1.1]` angegeben werden.

Unterschiede Maven 1 und Maven 2

Hier in Kürze und stichwortartig die Unterschiede zwischen Maven1 und Maven2:

- Während bei Maven 1 die Informationen für ein Projekt in unterschiedlichen Konfigurationsdateien verstreut waren, (`build.xml`, `build.properties`, `maven.xml`) sind jetzt die Informationen gebündelt im POM (Project Object Model).
- Maven 2 unterscheidet zwischen Goals (analog den Zielen in Maven 1) und den Lifecycle-Phasen (kompilieren, validieren, war bauen, deployen etc.) des Build-Prozesses (phases). Mehr dazu im Kasten Maven 2 Lifecycle.
- Während bei Maven 1 die Konfigurationsdatei im User-Home quasi Pflicht ist, läuft Maven 2 auch ohne, nutzt dann aber auch eine Menge defaults.
- Während unter Maven 1 eine Reihe von Standardreports automatisch duch maven:site erzeugt wurde, müssen diese Reports jetzt explizit in der pom.xml angegeben werden. Neb en dem xdocs xml format, werden jetzt auch andere Formate unterstützt so z.B. ein wiki-ähnliches Format apt. Um die Dokumentation zu vereinheitlichen benutzen die Autoren ihr eigenes Maven-1.0.2 Plugin TexConverter, das seit September 2006 als Open Source Projekt vorliegt [TEXCONV]. Dieser Konverter ist in der Lage aus einem einheitlichen Format (TeX) verschieden Zielformate wie HTML oder das Maven XDOCS Format zu generieren.
- Einige Plugins, die unter Maven 1 nicht mit dem JDK 5 auskamen sind jetzt unter Maven 2 auch mit diesem JDK einsetzbar, weil Maven 2 von Anfang an auf JDK 5 basierte.
- Der Multiprojekt-Support ist bei Maven 2 deutlich verbessert und vereinfacht worden. Wendet man das Plugin `mvn eclipse:eclipse` auf solche in Submodule zerlegte Projekte an, so wird für jedes Submodule ein Eclipse-Projekt erzeugt.
- Plugins müssen nicht mehr mit Jelly geschrieben werden. Maven stellt Basisklassen zur Verfügung (MOJO) um komfortabel eigene Plugins zu schreiben. Argumente werden durch Dependency-Injection in das MOJO injeziert.
- Die interessanteste Neuerung erscheint mir aber, dass Maven 2 nicht nur das jar, sondern auch das POM in seinem Repository hält und so in der Lage ist abhängige Jars mit zu laden. Außerdem ist Maven 2 in der Lage auch gleich die Sourcecodes der abhängigen Bibliotheken zu laden, das macht das Debuggen einfacher.

Fazit

Maven 2 lohnt sich! Dadurch, dass das Projekt fast vollständig neu implementiert wurde, sind Fehler aus der Version 1 korrigiert worden. Das ganze Projekt erscheint logischer und durchgängiger. Unser Tip: Vorhandene Projekte sollten nicht migriert werden, auch wenn der Aufwand nicht sehr hoch ist. Hier erscheint uns die Weisheit „never touch a running system“ angebracht. Ein neues Projekt sollte jedoch immer mit Maven 2 begonnen werden. Der Lernaufwand für die Entwickler sich auf die neue Version einzulassen ist relativ gering.

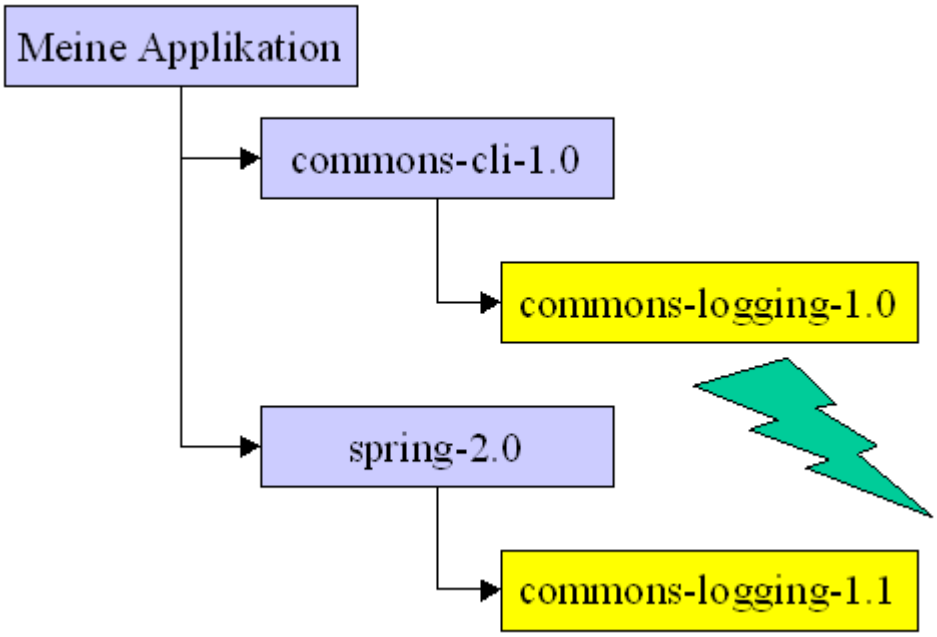
Referenzen:

[MAVEN] http://maven.apache.org

[GENAPP] Maven in fünf Minuten: http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

[TEXCONV] http://www.texconverter.org

Abbildung 1:


Meine Applikation
commons-cli-1.0
commons-logging-1.0
spring-2.0
commons-logging-1.1

Kasten settings.xml:

```
<settings 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/xsd/settings-1.0.0.xsd">
  <proxies>
    <proxy>
      <id>my-proxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>http-proxy.my.company.de</host>
      <port>3128</port>
      <username>myname</username>
      <password>mypassword</password>
    </proxy>
  </proxies>
</settings>
```

Kasten: Der Maven Livecycle

1. validate: überprüft die Gültigkeit der Konfigurationsdateien und überprüft das POM.
2. initialize: Führt notwendige Initialisierungen aus die nötig sind, bevor der Buildvorgang beginnen kann.
3. generate-sources: Generiert Code von anderen Quellen wie z.B. XDoclet.
4. process-sources: Führt, wenn Notwendig, Modifizierungen des Sourcecodes durch.
5. generate-resources: Generiert notwendige Ressourcen wie Konfigurationsdateien, Shellscripte etc. aus anderen Formaten.
6. process-resources: Führt notwendige Modifizierungen an Ressourcen aus z.B. das Kopieren von Ressourcen in den CLASSPATH.
7. compile: Kompiliert die Sourcen.
8. process-classes: Führt notwendigen Modifikationen an den Binaries aus wie z.B. Binärcode Manipulationen oder code-weaving im Bereich der aspektorientierten Programmierung..
9. generate-test-sources: Generiert Unit Test Code.
10. process-test-sources: Führt, wenn notwendig, Modifizierungen an den Test Sourcen durch.
11. generate-test-resources: Generiert Resourcen, die für Tests notwendig sind.
12. process-test-resources: Führt notwendige Modifizierungen von Test Ressourcen aus.
13. test-compile: Kompiliert die Test Sourcen.
14. test: Fhrt alle Unit Tests aus.
15. package: Führt die compilieren und getesteten Bestandteile der Anwendung zu einer Distribution zusammen, z.B. zu einer Jar-Datei.
16. preintegration-test: Führt notwendige Schritte für einen Integrationstest aus z.B. das Kopieren der Distribution auf einen Integrations-Server.
17. integration-test: Führt den Integrationstest aus..
18. post-integration-test: Führt den Integrationsserver wieder auf die letzte Baseline zurück.
19. verify: Verifiziert die Inhalte der Distribution, bevor sie ins zentrale Repository überführt wird.
20. install: Installiert die Distribution auf dem lokalen Maven Repository.
21. deploy: Installiert die Distribution auf dem entfernten Maven Repository.