

# Maven im Überblick

Dieses Kapitel beschreibt, wie Maven installiert und ein erstes Projekt erstellt wird. Anhand dieses Projekts werden die wichtigsten Maven-Aufrufe gezeigt. Auf die detaillierte Beschreibung in späteren Kapiteln wird entsprechend verwiesen.

## Hinweis

Maven verwendet für alle Aufgaben Plugins, die beim ersten Aufruf aus dem Netz geladen werden müssen. Wundern Sie sich also nicht, wenn die ersten Aufrufe etwas länger dauern. Da das Herunterladen der Bibliotheken auf der Konsole protokolliert wird, gehen beim ersten Aufruf eines Befehls die wesentlichen Informationen schnell unter. Rufen Sie zur Not den Befehl einfach noch mal auf. Gegebenenfalls rufen Sie vorher `mvn clean` auf, um generierte Dateien zu löschen.

## 2.1 Was ist Maven?

Maven ist ein deklaratives *Build Management System*. Das heißt, es wird lediglich der Inhalt des Projekts beschrieben, nicht die Struktur oder die Abläufe, die zur Kompilierung und Veröffentlichung notwendig sind. Die Philosophie hinter Maven heißt *Konvention vor Konfiguration* – Strukturen müssen nicht definiert werden, sondern sind vorgegeben. So wie die Projekt- und Verzeichnisstruktur ist auch die Reihenfolge der Arbeitsschritte vorgegeben, die Maven ausführt, um ein Projekt zu bauen. In der `pom.xml` von Maven 2.0.9 beschreiben die Entwickler Maven so:

*Maven is a project development management and comprehension tool. Based on the concept of a project object model: builds, dependency management, documentation creation, site publication, and distribution publication are all controlled from the declarative file. Maven can be extended by plugins to utilise a number of other development tools for reporting or the build process.*

### 2.1.1 POM

Maven verwendet ein Projektmodell (*POM – Project Object Model*), um Abhängigkeiten, Projektumgebung und Projektbeziehungen zu speichern. Das Projektmodell wird in der Datei `pom.xml` gespeichert und ist vererbbar.

### 2.1.2 Lebenszyklen

Die Entwickler von Maven gehen von immer wiederkehrenden Abläufen in Projekten aus, die in so genannten Lebenszyklen (*Lifecycles*) abgebildet werden. Der Standardzyklus ist der *Build-Lifecycle*, der aus einer festen Abfolge von Phasen besteht. Die einzelnen Phasen können mit Plugins verknüpft werden, die mit der Phase ausgeführt werden. Wird eine Phase aufgerufen, zum Beispiel `test`, werden alle Phasen des Zyklus, die vor `test` liegen, abgearbeitet. Das heißt, um Tests auszuführen, werden immer alle notwendigen Schritte ausgeführt:

1. Kompilieren des Produktivcodes
2. Kompilieren des Testcodes
3. Ausführen der Tests

### 2.1.3 Vereinfachtes Build-Management

Durch die Verwendung einheitlicher Verzeichnisstrukturen ist es einfach, in Projekte einzusteigen, die mit Maven verwaltet werden. Der *Build-Lifecycle* kapselt die einzelnen Phasen, die langwierige Analyse von *Makefiles* oder Build-Skripten, um herauszufinden, welcher Befehl als Erstes aufgerufen werden muss, entfällt.

### 2.1.4 Trennung von Code und Unit-Tests

Maven trennt den produktiven Projektcode physisch vom Code der Unit-Tests. Die Code-Basen liegen parallel in unterschiedlichen Quell-Verzeichnissen und die kompilierten Klassen werden in unterschiedlichen Verzeichnissen verwaltet. Genauso sind die Konfiguration der Unit-Tests und die dazugehörigen Ressourcen vom restlichen Code getrennt.

### 2.1.5 Verwaltung von Abhängigkeiten

Maven verwendet ein einziges lokales Verzeichnis, genannt *Repository*, in dem Bibliotheken zentral für alle Projekte abgelegt werden. Benötigte Bibliotheken werden selbstständig ins lokale Repository kopiert und aktualisiert. Damit erübrigt sich die Notwendigkeit, in Projekten Bibliotheken in das Versionskontrollsystem einchecken zu müssen. Transitiv Abhängigkeiten, also die Abhängigkeiten von Abhängigkeiten, werden selbstständig analysiert und aufgelöst. Unter Maven ist es daher sehr einfach, Bibliotheken auszutauschen. Auch das Entfernen von Abhängigkeiten aus Projekten wird deutlich einfacher: Verschwindet die Abhängigkeit, verschwinden auch die transitiven Abhängigkeiten. Es bleiben keine Dateileichen zurück.

### 2.1.6 Artefakte

Maven beschreibt genau genommen keine vollständigen Projekte, sondern Artefakte: eigenständige Teile eines Projekts, die einzeln ausgeliefert werden können. In kleinen Projekten lässt sich das gesamte Projekt mit einem einzigen Artefakt beschreiben. Ein Artefakt wird durch eindeutige Koordinaten beschrieben, die im POM abgelegt sind. Jedes Maven-Projekt erzeugt ein Artefakt.

### 2.1.7 Informationen zu Codequalität und Projektzustand

Mit Maven lassen sich Reports und Projektinformationen generieren, die in einer Projektwebseite zusammengefasst werden. Hierzu gehören unter anderem:

- API-Dokumentation
- Unit-Test-Ergebnisse und Testabdeckung
- Changelogs des Versionskontrollsystems
- Liste der Abhängigkeiten
- Reports zur Codequalität durch Tools wie PMD, Checkstyle und FindBugs
- Informationen zu Bug-Tracking, Mailing-Listen und *Continuous Integration*
- Verwendete Bibliotheken und deren Benutzung

## 2.2 Voraussetzungen

### 2.2.1 Betriebssystem

Maven läuft auf *Windows*-Systemen ab NT, *Mac OS X*- und *Linux*-Systemen. Persönlich verwendet habe ich *Maven* auf *Windows 2000 (SP4)*, *XP (SP2 und 3)*, sowie *SUSE Linux 9* und *Ubuntu 7.x* und *8.x*.

### 2.2.2 JDK

Maven benötigt zur Ausführung ein *JDK* ab Version 1.4. Es ist aber möglich, mit Maven auch Java-Projekte für ältere *JDKs* zu verwalten. Die Umgebungsvariable *JAVA\_HOME* muss auf das Java-Installationsverzeichnis verweisen.

### 2.2.3 Speicherplatz

#### Arbeitsspeicher

Es ist keine Mindestanforderung für Arbeitsspeicher angegeben. Mit den heute üblichen Speichergrößen sollte es auch keine Probleme geben. Maven 2.0.9 lief auf einem *Ubuntu 7.04*-System mit 512 MB RAM und *Java5* ohne Probleme. Das

mag sich bei großen Projekten, vor allem beim Generieren der Site, aber anders verhalten.

### Festplatte

Maven 2.0.10 selber benötigt etwa 2 MB auf der Festplatte. Für das lokale Repository kann dann je nach Projekten einiges an Speicher dazukommen. Auf der Maven-Homepage sind 100 MB als Richtwert angegeben, auf meinem Arbeitsrechner sind allerdings mehr als 330 MB durch das Repository belegt (ich habe in den letzten Monaten allerdings auch reichlich Plugins ausprobiert ...).


## 2.3 Installation

Um Maven zu installieren, sind folgende Schritte notwendig:

1. Laden Sie Maven vom Apache-Server herunter: <http://maven.apache.org/download.html>.

Verfügbar sind die Formate \*.zip, \*.tar.gz und \*.tar.bz2.

2. Entpacken Sie Maven in ein Verzeichnis Ihrer Wahl.

3. Die Umgebungsvariable M2\_HOME muss auf dieses Verzeichnis verweisen und der Befehlssuchpfad muss die ausführbaren Maven-Befehle kennen. Unter Windows müssen Sie dazu mit der Tastenkombination  + [Pause] die Systemeigenschaften aufrufen und dann auf dem Reiter ERWEITERT die Schaltfläche UMGEBUNGSVARIABLEN auswählen. Fügen Sie unter BENUTZERVARIABLEN mittels NEU die Variable M2\_HOME hinzu. Fügen Sie außerdem der Variablen PATH das Verzeichnis %M2\_HOME%\bin hinzu. Legen Sie dazu eine neue Variable PATH an und weisen Sie ihr den Wert %M2\_HOME%\bin;%PATH% zu.

Unter \*nix-Systemen werden die Variablenzuweisungen in einer Konsole eingegeben:

```
export M2_HOME=/usr/local/apache-maven-2.0.10
export path=$M2_HOME:$path
```

Mit Hilfe des Befehls `mvn --version` kann geprüft werden, ob die Installation erfolgreich war:

```
D:>mvn -version
Maven version: 2.0.10
Java version: 1.6.0_12
OS name: "windows xp" version: "5.1" arch: "x86" Family: "windows"
```

Falls der Rechner hinter einer Firewall liegt, muss noch die Datei `settings.xml` erstellt werden, die im Home-Verzeichnis des Users liegt: `~/m2/` unter *Unix/Mac*

bzw. C:\Dokumente und Einstellungen\%USERNAME%\m2\ unter *Windows*-Systemen.

```
<settings>
  <proxies>
    <proxy>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxyhost</host>
      <port>proxyport</port>
      <nonProxyHosts>localhost,zaphod,...</nonProxyHosts>
    </proxy>
  </proxies>
</settings>
```

Maven verwaltet alle Plugins und Abhängigkeiten wie *JARs* in einem Verzeichnis, das als lokales Repository bezeichnet wird. Standardmäßig wird dieses Repository ebenfalls im ~/m2-Verzeichnis erstellt. Sollte Ihr Benutzer-Profil auf einem Server gespeichert werden, empfiehlt es sich, ein anderes Verzeichnis für das lokale Repository anzugeben. Diese Einstellung wird ebenfalls in der Datei `settings.xml` vorgenommen:

```
<settings>
  ...
  <localRepository>D:/java/Maven2/repository</localRepository>
  ...
</settings>
```

## 2.4 Projekt erstellen

Um ein erstes Projekt zu generieren, wird das Archetype-Plugin aufgerufen:

```
mvn archetype:create \
  -DgroupId=de.mavenbuch.beispiel \
  -DartifactId=beispiel-projekt
```

Dieser Aufruf erstellt im aktuellen Verzeichnis ein Unterverzeichnis `beispiel-projekt`, das ein rudimentäres Java-Projekt enthält, bestehend aus einer Projektbeschreibungdatei, einer Java-Klasse und dem dazugehörigen *JUnit*-Test.

Die angegebene `groupId` wurde als Vorgabe für die Java-Paket-Struktur verwendet.

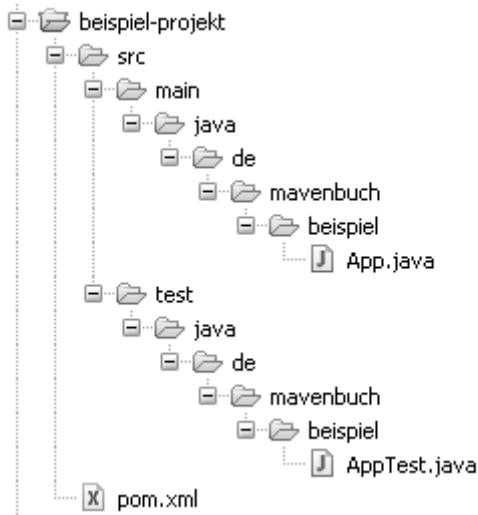


Abb. 2.1: Verzeichnisstruktur des Beispiel-Projekts

Die Projektbeschreibungsdokumentation `pom.xml` enthält, neben den eben angegebenen Werten, auch die, automatisch vergebene, Version `1.0-SNAPSHOT`, den `packaging`-Typ `jar`, der angibt, dass das Projekt als Standard-Java-Archiv gepackt werden soll, sowie `JUnit` als einzige Abhängigkeit:

```
<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>de.mavenbuch.beispiel</groupId>
  <artifactId>beispiel-projekt</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>beispiel-projekt</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

## Hinweis

Eventuell ist Ihnen beim Aufruf von `mvn archetype:create` folgende Warnung in der Maven-Ausgabe auf der Konsole aufgefallen:

```
...  
[WARNING] This goal is deprecated. Please use mvn archetype:generate  
instead  
...
```

Das sollte Sie allerdings nicht weiter beunruhigen. Da die Erzeugung eines Standard-Java-Projekts mit `archetype:create` so bequem ist, ignorieren wir die Warnung an dieser Stelle einfach. Das Goal `archetype:generate` wird in Abschnitt 2.8 erläutert.

## 2.5 Bauen – Testen – Packen

### 2.5.1 Basisfunktionen

#### Kompilieren

Wechseln Sie in das Verzeichnis und kompilieren Sie die Quellen:

```
mvn compile
```

Maven erstellt nun den Zielordner `target/classes` und kompiliert die Klasse aus dem Source-Pfad `src/main/java` in dieses Verzeichnis. Auf der Kommandozeile sieht das Ergebnis so aus:

```
D:\beispiel-projekt>mvn compile  
[INFO] Scanning for projects...  
[INFO] -----  
[INFO] Building beispiel-projekt  
[INFO]   task-segment: [compile]  
[INFO] -----  
[INFO] [resources:resources]  
[INFO] Using default encoding to copy filtered resources.  
[INFO] [compiler:compile]  
[INFO] Compiling 1 source file to D:\beispiel-projekt\target\classes  
[INFO] -----  
[INFO] BUILD SUCCESSFUL  
[INFO] -----  
[INFO] Total time: 1 second  
[INFO] Finished at: Thu Jan 15 17:32:11 CET 2009  
[INFO] Final Memory: 5M/10M  
[INFO] -----
```

## Wichtig

Das Übersetzen des Codes wird durch das Maven Compiler Plugin durchgeführt. Aus Kompatibilitätsgründen ist die Voreinstellung für Source-Code Version 1.3, für die kompilierten Klassen Version 1.1. Wollen Sie die Features einer aktuelleren Java Version nutzen, müssen Sie das Compiler Plugin in der `build` Sektion des *POM* entsprechend konfigurieren:

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
    ...
  </build>
  ...
</project>
```

### Unit-Tests ausführen

Um den Unit-Test auszuführen, rufen Sie

```
mvn test
```

auf. Nun werden folgende Schritte ausgeführt:

1. Der Zielordner `target/test-classes` wird erstellt, der die kompilierten Testklassen aufnimmt.
2. ... Die Testklassen werden kompiliert
3. ... und die Unit-Tests werden ausgeführt.

Anschließend erscheint die Ausgabe über den erfolgreichen Durchlauf der Tests:

```
...
-----
T E S T S
```

```
-----  
Running de.mavenbuch.beispiel.AppTest  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,  
    Time elapsed: 0.062 sec  
  
Results :  
  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
...
```

## Packen und Installieren

Der Aufruf der Phase `package` mit

```
mvn package
```

erzeugt ein *JAR-Archiv* im Ordner `target` des Projekts.

Der Aufruf von

```
mvn install
```

schließlich legt eine Kopie des JAR-Archivs und der Datei `pom.xml` im lokalen Repository ab.

## Aufräumen

Um das Projektverzeichnis aufzuräumen, sprich das `target`-Verzeichnis zu löschen, wird

```
mvn clean
```

aufgerufen.

Die Aufrufe lassen sich auch kombinieren. So ergibt der Aufruf

```
mvn clean package
```

ein erneutes Kompilieren und Packen des Projekts, nachdem der `target`-Ordner gelöscht wurde. Genau genommen ruft `mvn package` die Phase `package` des *Build-Zyklus* auf. Deren Ausführung beinhaltet alle notwendigen Schritte vom Kompilieren übers Testen bis zur `jar`-Erzeugung. Wie das alles genau zusammenhängt, beschreibt Kapitel 5.

## 2.5.2 Dokumentation

Mit Maven lässt sich aus den Informationen der POM eine Sammlung von verlinkten Webseiten generieren.

Rufen Sie

```
mvn site
```

auf und öffnen Sie nach Ablauf des Programms die Datei `target/site/index.html` in einem Browser:



Abb. 2.2: Durch Maven generierte Webseite

Probieren Sie einfach mal die Links im Menü auf der linken Seite aus. Unter PROJECT INFORMATION findet sich zum Beispiel eine Auflistung und Beschreibung der erzeugten Seiten (siehe Abbildung 2.3).

Sie können zusätzlich zu den Informationen aus der POM auch verschiedene Reports zu Change-Listen des Versionskontrollsystems, Codequalität und Testergebnissen sowie Links in die Javadoc und den Sourcecode generieren lassen. Zusätzlich kann die komplette Projektdokumentation in Gestalt von Dokumentationen und FAQs durch Maven generiert werden. Um zum Beispiel eine Analyse des Sourcecodes mit *PMD* durchführen zu lassen und das Ergebnis in die Webseite einzubinden, wird die Datei `pom.xml` um eine `reporting`-Sektion erweitert und das *PMD*-Plugin dort eingetragen:

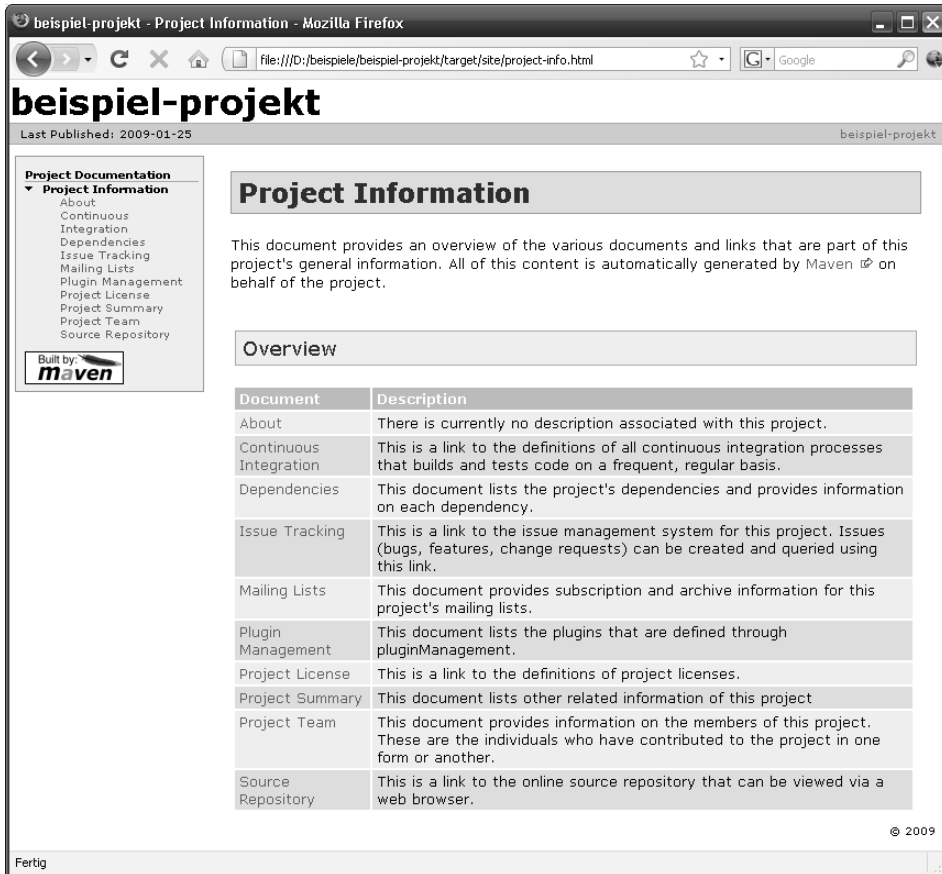


Abb. 2.3: Projektinformationen in der generierten Webseite

```
<project>
...
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
...
</project>
```

Kapitel 17 beschreibt die möglichen Elemente der von Maven generierten Dokumentation und ihre Konfiguration. Weitere Informationen zur Qualitätssicherung

mit Maven und zu Werkzeugen wie PMD, FindBugs und anderen finden sich in Kapitel 18.

## Javadoc

Der Aufruf

```
mvn javadoc:javadoc
```

erzeugt aus den Javadoc-Kommentaren im Code die entsprechende API-Dokumentation des Projekts im Verzeichnis `target/site/apidocs`.

Soll die Javadoc auch in der generierten Webseite eingebunden werden, muss die `reporting`-Sektion ergänzt und das Javadoc-Plugin dort eingetragen werden:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins
        </groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

Nach einem weiteren Aufruf von

```
mvn site
```

findet sich nun auf der generierten Webseite

```
target/site/index.html
```

unterhalb von `PROJECT INFORMATION` ein neuer Link `PROJECT REPORTS`, unter dem sich die Punkte `JAVADOCS` und `TEST JAVADOCS` befinden.

Wie das Javadoc-Tool kann auch das Javadoc-Plugin konfiguriert werden, um zum Beispiel eigene Stylesheets oder Doclets zu verwenden.

Um durch `mvn package` auch ein Archiv mit der Schnittstellenbeschreibung erzeugen zu lassen, wird die `pom.xml` um folgenden Eintrag ergänzt:

```
<project>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-javadocs</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...
```

Wird das Projekt nun mit `mvn package` gebaut, so wird ein weiteres Java-Archiv erstellt:

```
target/beispiel-programm-1.0-SNAPSHOT-javadoc.jar
```

Analog lässt sich mit dem Plugin `maven-source-plugin` ein Java-Archiv mit den Projektquellen erzeugen:

```
<project>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...
```

### Vorsicht

Die so erstellten Archive mit Dokumentation und Quellcode werden beim Aufruf von `mvn deploy` auch auf ein externes Repository kopiert. Das ist vielleicht nicht bei allen Projekten unbedingt erwünscht.

## 2.6 Projekte erweitern

Um das erstellte Projekt um Bibliotheken zu erweitern, muss ebenfalls die Datei `pom.xml` angepasst werden. Nehmen wir an, das Logging-Framework *Log4J* soll verwendet werden. Dazu erweitern wir die Sektion `dependencies`, in der Abhängigkeiten des Projekts verwaltet werden. In unserem Beispiel-Projekt befindet sich bereits ein Eintrag für *JUnit*, unter dem wir nun *Log4J* hinzufügen:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
</dependencies>
```

Dazu geben wir die Koordinaten der Bibliothek an: `groupId`, `artifactId` und `version`. Der Eintrag für *JUnit* enthält außerdem den Eintrag `<scope>test</scope>`, der den Gültigkeitsbereich angibt: *JUnit* wird nur für die Phasen `test-compile` und `test` benötigt. Lassen wir die `scope`-Angabe weg, wird automatisch der Gültigkeitsbereich `compile` angenommen. Führen wir nun wieder den Befehl `mvn compile` aus, wird die Datei `log4j-1.2.14.jar` in das Verzeichnis `log4j/log4j/1.2.14/` des lokalen Repositorys heruntergeladen und in unser Projekt eingebunden.

Mehr zu Abhängigkeiten, Gültigkeitsbereichen und wie Maven damit umgeht, erklärt Kapitel 7.

## 2.7 Ein Web-Projekt

### 2.7.1 Generieren einer Web-Applikation

Mit `mvn archetype:create` lassen sich auch andere Arten von Projekten, wie zum Beispiel ein Web-Projekt erzeugen:

```
mvn archetype:create \  
  -DgroupId=de.mavenbuch.beispiele \  
  -DartifactId=webapp-beispiel \  
  -DarchetypeArtifactId=maven-archetype-webapp
```

Anstelle der Verzeichnisse `src/main/java` und `src/test/java` stehen hier am Anfang `src/main/webapp` und `src/main/resources` zur Verfügung:

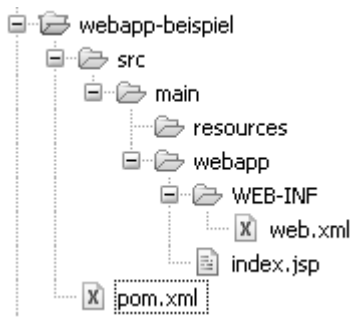


Abb. 2.4: Abbildung 2.4: Struktur des Web-Projekts

Die generierte `pom.xml` unterscheidet sich in ein paar Punkten von der aus dem ersten Beispiel:

```
<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>de.mavenbuch.beispiele</groupId>
  <artifactId>webapp-beispiel</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>webapp-beispiel Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
<build>
  <finalName>webapp-beispiel</finalName>
</build>
</project>
```

1. Der Eintrag `<packaging>war</packaging>` gibt an, dass ein *war*-Archiv erzeugt werden soll.
2. Mit dem Eintrag `<finalName>webapp-beispiel</finalName>` wird festgelegt, dass als Archivname immer *webapp-beispiel.war* verwendet wird, anstatt der Kombination aus *artifactId* und *Version*.

## 2.7.2 Servlet-API

Um die erstellte Web-Anwendung auch mit Leben zu füllen, benötigen wir die *Servlet-API* in unserem Klassenpfad. Allerdings nur zum Kompilieren, da normalerweise jeder *Servlet-Container* die *Servlet-API* in seiner Laufzeitumgebung mitbringt. Binden wir die Bibliothek auch zur Laufzeit ein, würde dies deshalb höchstwahrscheinlich zu einem Versionskonflikt führen. Um dies zu vermeiden, verwendet Maven den Gültigkeitsbereich `provided`, der angibt, dass die Bibliothek nicht durch Maven zur Laufzeit zur Verfügung gestellt werden muss:

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

## 2.7.3 Ausführen der Web-Anwendung

Normalerweise muss, um eine Web-Applikation auszuführen, ein *Servlet-Container* wie *Tomcat* oder *Jetty* konfiguriert und die Applikation *deployed* werden. Verwendet man jedoch das *Jetty-Plugin* `jetty-maven-plugin`, kann die Web-Applikation direkt mit Maven gestartet werden. Dazu muss lediglich die Datei `pom.xml` um einen *Plugin-Eintrag* erweitert werden:

```
<project>
  ...
  <build>
    <finalName>webapp-beispiel</finalName>

    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>7.0.0pre3</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Wird nun auf der Konsole

```
mvn jetty:run
```

aufgerufen, steht eine Jetty-Instanz auf Port 8080 zur Verfügung. Ein anderer Port kann mittels `-Djetty.port=<NNNN>` übergeben werden.

[jetb] enthält eine ausführliche Beschreibung über die Konfigurationselemente des Plugins.

Der Befehl `mvn package` erzeugt im Verzeichnis `target/${finalname}` ein komplettes Abbild der Web-Applikation. Dieser Pfad kann einem lokalen *Servlet-Container* als Basisverzeichnis mitgeteilt werden, um die Applikation während der Entwicklungsphase zu testen. Um unser Beispiel in *Tomcat 6.o.\** anzumelden, muss dazu im Tomcat-Installationsverzeichnis die Datei `conf/Catalina/local-host/webapp-beispiel.xml` mit folgendem Inhalt angelegt werden:

```
<Context
  path="/webapp-beispiel"
  docBase="<PFAD>/webapp-beispiel/target/webapp-beispiel"
  reloadable="true" />
```

## 2.8 Weitere Projekttypen

Für einfache Java-Projekte mag die Projekterzeugung mittels `mvn archetype:create` ausreichend sein, sobald aber andere (oder eigene) Templates verwendet werden sollen, sollte das bereits erwähnte Goal

```
mvn archetype:generate
```

verwendet werden.

Nach dem Aufruf bietet Maven eine Liste von Projekt-Archetypen an. Darunter befinden sich unter anderem Vorlagen für J2EE-, JPA- und Portlet-Projekte.

Der aufgeführte Typ `maven-archetype-quickstart` ist übrigens der Standardprojektyp, der beim Aufruf des Goals `archetype:create` ohne den Parameter `archetypeArtifactId` verwendet wird.

Wurde einer der angegebenen Archetypen aus der Liste ausgewählt, fragt Maven noch nach `groupId`, `artifactId`, `version` und `package` und generiert dann ein Projekt des ausgewählten Typs.

```
Choose a number:(1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/\
21/22/23/24/25/26/27/28/29/30/31/32/33/34/35/36/37/38/39/40/41)  \
15: : 10
Define value for groupId: : de.mavenbuch.beispiele
Define value for artifactId: : j2ee-beispiel
Define value for version: 1.0-SNAPSHOT: :
Define value for package: de.mavenbuch.beispiele: :
Confirm properties configuration:
groupId: de.mavenbuch.beispiele
artifactId: j2ee-beispiel
version: 1.0-SNAPSHOT
package: de.mavenbuch.beispiele
Y: :
```

In Abschnitt 18.4 wird gezeigt, wie eigene Vorlagen für die Verwendung mit dem Archetype-Plugin erzeugt werden können.

## 2.9 Entwicklungsumgebungen

Normalerweise benutzen, von einigen Puristen mal abgesehen, Entwickler moderne IDEs (*Integrated Development Environment*), die oft eigene, proprietäre Konfigurationsdateien verwenden.

Da Maven aber alle notwendigen Informationen des Projekts in der `pom.xml` zusammenhält, gibt es für die verbreitetsten IDEs auch Plugins, die die Projekte automatisch importierbar machen.

Da ich meine tägliche Arbeit mit Eclipse erledige, werden die anderen IDEs hier nur oberflächlich behandelt.

### 2.9.1 Eclipse

Für Eclipse gibt es mehrere Lösungen, um Maven-Projekte zu integrieren. Der direkteste ist die Generierung der Projektdateien `.classpath` und `.project` mit Hilfe des Maven-Eclipse-Plugins:

```
mvn eclipse:eclipse
```

Die Datei `.project` im unserem *beispiel-projekt* sieht danach wie folgt aus:

```
<projectDescription>
  <name>beispiel-projekt</name>
  <comment/>
  <projects/>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
    </buildCommand>
  </buildSpec>
  <natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
  </natures>
</projectDescription>
```

Die Datei `.classpath` hat folgenden Inhalt:

```
<classpath>
  <classpathentry kind="src" path="src/main/java"/>
  <classpathentry kind="src" path="src/test/java"
    output="target/test-classes"/>
  <classpathentry kind="output" path="target/classes"/>
  <classpathentry kind="con"
    path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="var"
    path="M2_REPO/junit/junit/3.8.1/junit-3.8.1.jar"/>
</classpath>
```

Dabei fällt auf, dass die Abhängigkeit des Projekts *beispiel-projekt* von *JUnit* relativ zu `M2_REPO` hergestellt wird. Damit Eclipse den Pfad `M2_REPO` auflösen kann, muss im *Build-Pfad* eine *Classpath-Variable* angelegt werden. Dies kann durch den Aufruf von

```
mvn -Declipse.workspace=<path-to-eclipse-workspace> \
  eclipse:add-maven-repo
```

geschehen oder manuell eingegeben werden (siehe Abbildung 2.5).

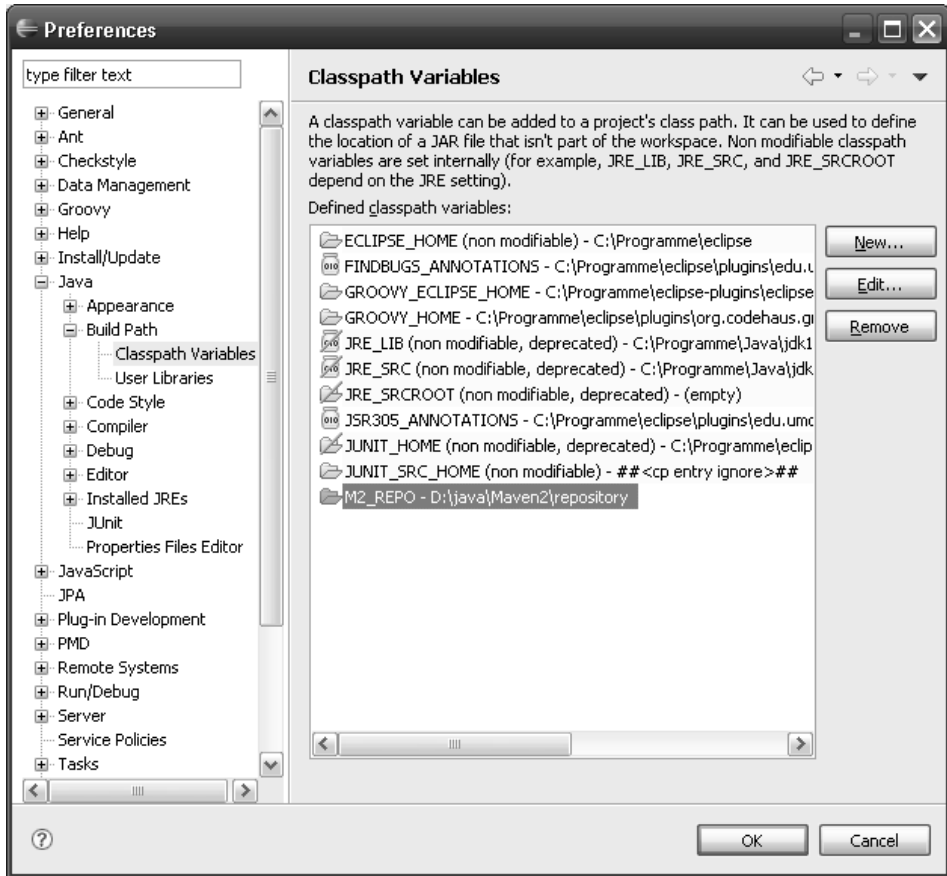


Abb. 2.5: Einstellung der Classpath-Variablen in Eclipse

Dazu muss in Eclipse unter `Windows/Preferences/Java/Build Path/Classpath Variables` eine neue Variable mit dem Namen `M2_REPO` eingetragen werden, die dann auf das Verzeichnis des lokalen Repositories verweist.

Nun kann das Projekt importiert werden. Hierzu wird der Projektordner angegeben, wenn im Kontextmenü des aktuellen Workspace der Befehl `Import -> Existing projects into workspace` ausgeführt wird (siehe Abbildung 2.6).

Nach diesem Schritt steht im Eclipse-Workspace das Projekt `beispiel-projekt` zur Verfügung.

Kapitel 16 beschäftigt sich ausführlich mit der Konfiguration des Eclipse-Plugins und möglicher Alternativen.

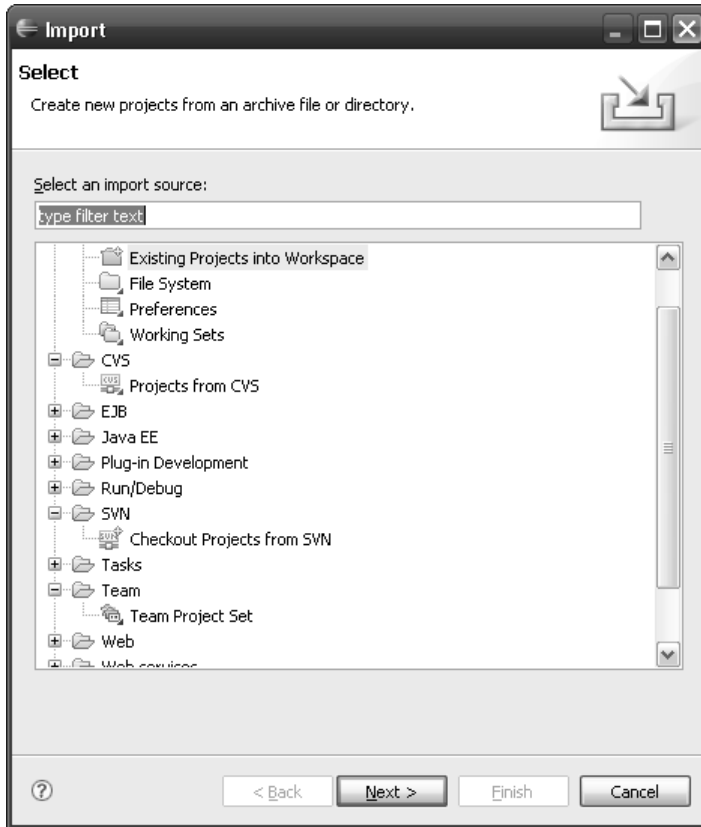


Abb. 2.6: Import von Projekten in Eclipse

## 2.9.2 Netbeans

Für Netbeans existiert das Projekt *Mevenide* (<http://mevenide.codehaus.org>), ein Plugin, das den Import von Maven-Projekten in Netbeans erlaubt.

Für ältere Netbeans-Versionen (4.x und 5.0) ist es auch möglich, mit

```
mvn netbeans-freeform:generate-netbeans-project
```

die notwendigen Konfigurationsdateien zu erzeugen. Das Maven-Plugin *netbeans-freeform* wird aber anscheinend nicht mehr weiterentwickelt.

Die Maven-Homepage bietet weitere Informationen zur Verwendung von *netbeans-freeform*:

```
http://maven.apache.org/guides/mini/guide-ide-netbeans/guide-ide-netbeans.html
```

### **2.9.3 IntelliJ IDEA**

*IntelliJ IDEA 8* unterstützt Maven von sich aus. Für ältere Versionen lassen sich mit

```
mvn idea:idea
```

die notwendigen Konfigurationsdateien erzeugen.