FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Softwaretechnik – Übung

Build-Management mit Maven

Prof. Dr. Bodo Kraft

Lernziele
Build-Management mit Maven

- Grundlegendes Verständnis
  - Build-Prozess und -Probleme
  - Build-Management
- Einführung von Maven
  - Verwendung
  - Architektur

Build-Management Einführung

Definition

Build-Management Einführung

Build-Management hat das Ziel, eine Reihe von Aufgaben der Softwareentwicklung zu automatisieren. Dazu zählen:

- Konfigurieren des Systems
- Kompilieren des Source-Codes
- Packen der Binär-Artefakte
- Ausführen der Tests
- Deployment ins Produktivsystem
- Erzeugen der Dokumentation

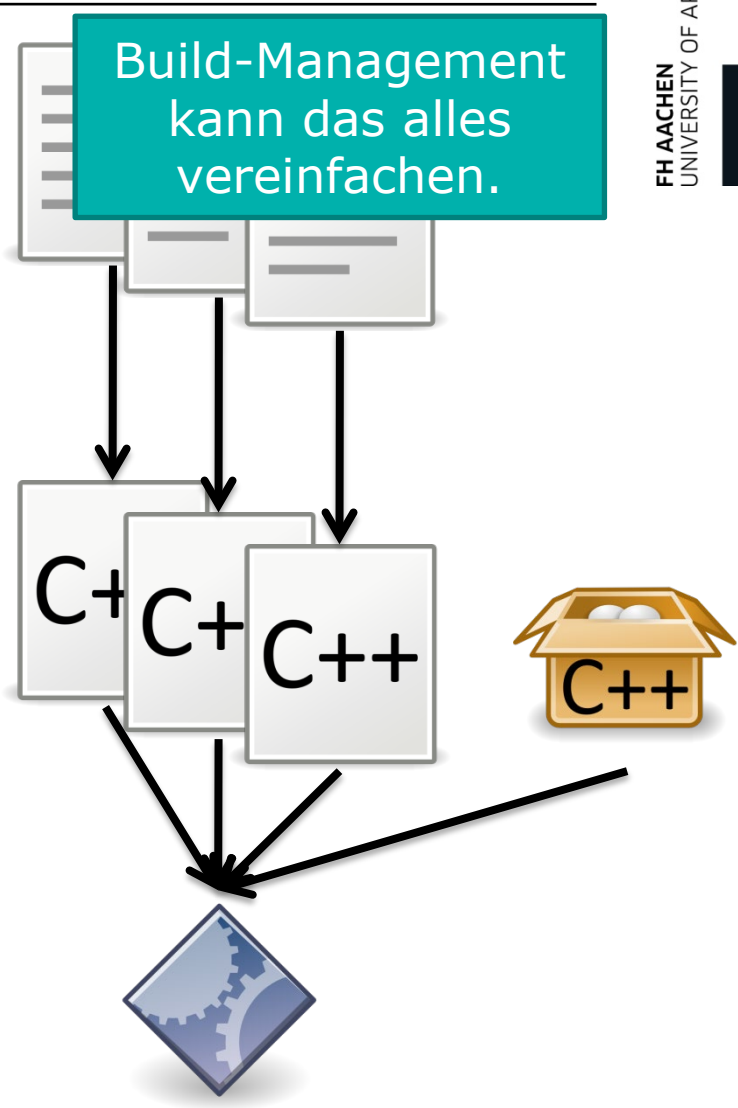
Der mühselige Weg von C++ zum Programm

Build-Management Einführung

1. Source-Code in Objektdateien überführen
   g++ -c file1.cpp
   g++ -c file2.cpp
   …

2. Objektdateien und externe Bibliotheken linken
   g++ -o prg *.o -lextern

Externe Bibliotheken müssen installiert oder lokal vorhanden sein.

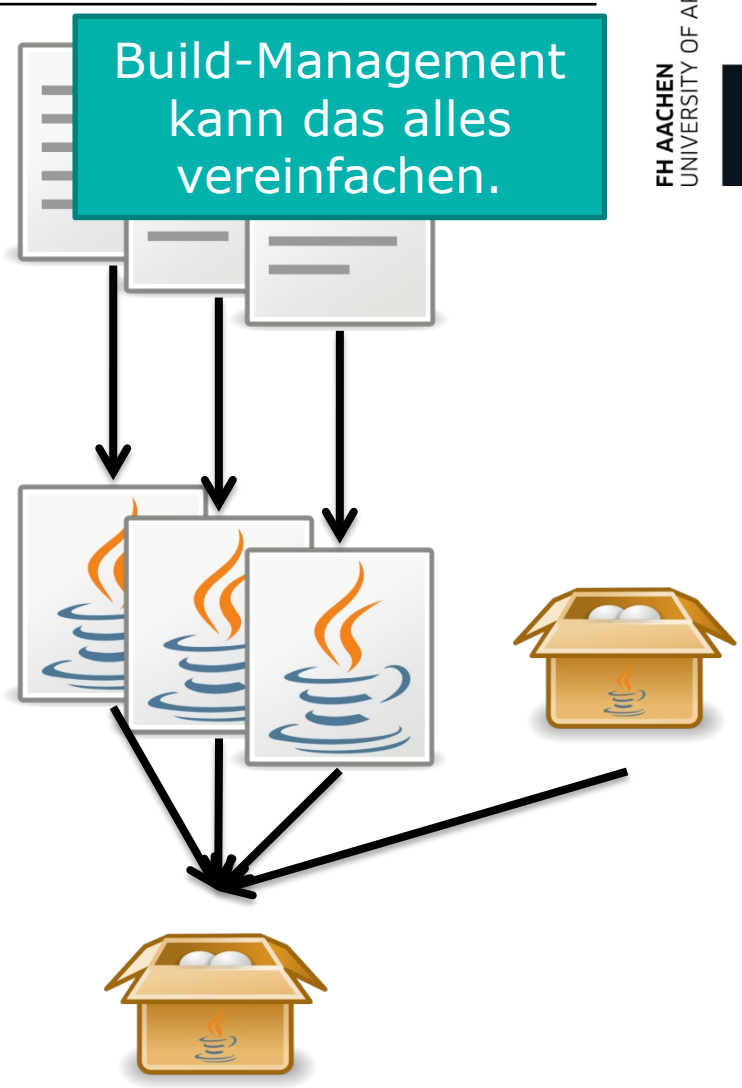


Der mühselige Weg von Java zum Programm

Build-Management Einführung

1. Source-Code in Class-Dateien überführen
   javac file1.java
   javac file2.java
   …
2. Class-Dateien und externe Bibliotheken zu einen Jar-Archiv zusammenführen
   jar cvf my.jar *.class
   # create manifest
   # include external jars
   # etc…

Externe Jar-Archive müssen von irgendwo bezogen werden.



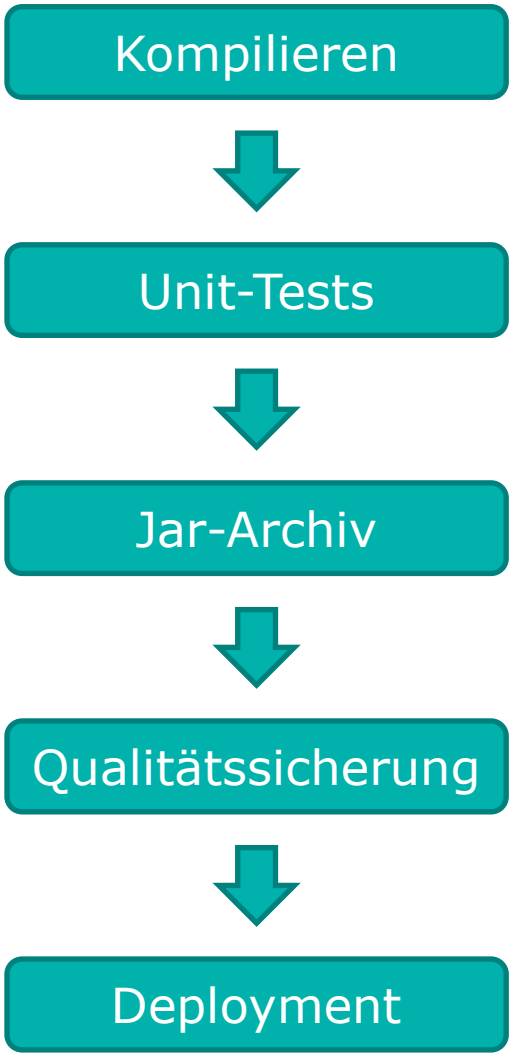
Build-Lifecycle

Build-Management Einführung

- Wie in vorherigen Beispielen gesehen, sind es einige Schritte vom Source-Code zum ausführbaren Programm.
- Die Schritte variieren von Projekt zu Projekt und von Technologie zu Technologie.
- Zusammenfassend werden diese Schritte **Build-Lifecycle** genannt.
- Neben der Erstellung des Programms können auch Schritte z. B. zur Qualitätssicherung eingebaut werden.

Build-Lifecyle - Ein Beispiel

Build-Management Einführung



1. Kompilieren der Java-Dateien
2. JUnit Tests ausführen
3. Jar-Archiv erstellen
4. z. B. Code-Coverage der Unit-Tests auswerten
5. Jar-Archiv zum Download bereitstellen

Build-Tools
Build-Management Einführung

- Es existieren eine Reihe an Tools, Build-Tools genannt, die Softwareentwickler unterstützen.
- Diese Tools bieten:
  - Kompilieren
  - Paketieren
  - Testen
  - Deploy
  - Dokumentieren (z. B. Javadoc)
- Beispiel: Make für C/C++, Ant für Java
- Aufwand reduziert sich, bleibt aber hoch.

Rolle des „Build-Engineers“

Build-Management Einführung

- Beschreibung:
  - Verantwortlich für Skripte zum Erzeugen, Testen und Verteilen der Software
  - Jedes Projekt besitzt einen Build-Engineer
- Aufgaben:
  - Aufsetzen und Anpassen des Build-Systems
  - Spezielle Bedürfnisse des Projekts berücksichtigen (Bibliotheken, Plattform)
- Herausforderung:
  - Initiales Auschecken und kompilieren dauert oft Stunden (Konfiguration)
- Wunsch: Auschecken und ein einzelnes Kommando stellt gesamte Entwicklungsumgebung bereit.

Maven aus der Vogelperspektive

Maven ist mehr als ein Build-Tool

Maven aus der Vogelperspektive

- Maven ist ein Projekt-Management-Tool:
  - ist eine Obermenge eines Build-Tools
  - generiert Reports in Form von Websites
  - etc.
- Maven ist ein Dependency-Management-Tool:
  - verwaltet Abhängigkeiten zu externen Bibliotheken
  - etabliert standardisierten Austauschprozess
- Maven ist individualisierbar:
  - Plugin-Architektur von Maven erlaubt dies
- Insbesondere Standardisierung für Java (Maven ist sprachneutral).

Zentrales Maven Prinzip

Maven aus der Vogelperspektive

Convention over Configuration (CoC)

- Paradigma, um die Komplexität von Konfigurationen zu reduzieren.
- Konventionen sind Standards, die keiner Konfiguration benötigen.

Convention over Configuration in Maven - Java

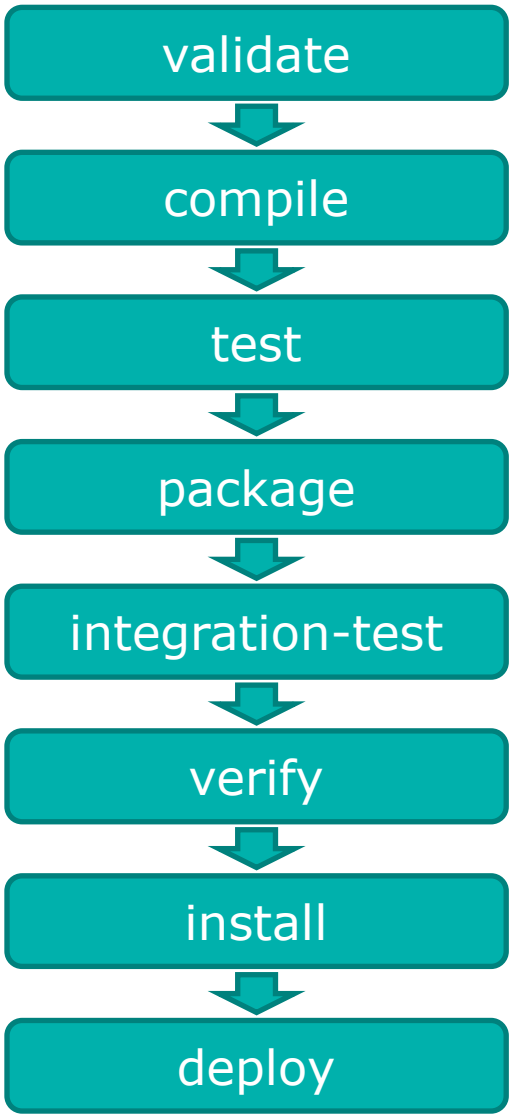
Maven aus der Vogelperspektive

- Konventionen:
  - **Source Code**: *${basedir}/src/main/java*
  - **Resources**: *${basedir}/src/main/resources*
  - **Test Code**: *${basedir}/src/test/java*
  - **Test Resources**: *${basedir}/src/test/resources*
  - **Byte Code**: *${basedir}/target/classes*
  - **JAR File**: *${basedir}/target/${artifact}.jar*

- Klingt trivial, aber mit Ant muss man das alles **konfigurieren**.

Maven Default Build-Lifecycle

Maven aus der Vogelperspektive



Validierung der Projektstruktur

Kompilierung des Quellcodes

Tests mit geeignetem Testframework

Kompilat wird zur Weitergabe gepackt

Ausführung von Integrationstests

Gültige Struktur des Paket prüfen ggf. Qualitätschecks

Software Paket wird im lokalen Repository installiert

Stabile Version der Software wird auf einem entfernten Repository installiert

Project Object Model – Konfigurationsdatei

Maven aus der Vogelperspektive

- Projekte mit Maven werden über eine XML-Datei konfiguriert
- pom.xml ≙ Beschreibung des Projektes
  - Projekttyp (Jar, War, etc.)
  - Entwickler
  - Projektlizenz
  - Abhängigkeiten
- Maven verwendet Deklarationen:
  - „Dies ist ein Jar-Projekt“
  - „der Source Code liegt unter src/main/java“
- In dieser Datei kann man die Konventionen, wenn das nötig ist, überschreiben.

Minimales Project Object Model

Maven aus der Vogelperspektive


```
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>de.fhaachen</groupId>
    <artifactId>swt/artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>jar</packaging>
</project>
```

Koordinaten des Projekts

POM-Version

Entwicklergruppe

Projekt der Gruppe

Aktuelle Version

Projekttyp

Build-Lifecycles und Maven-Kommandos

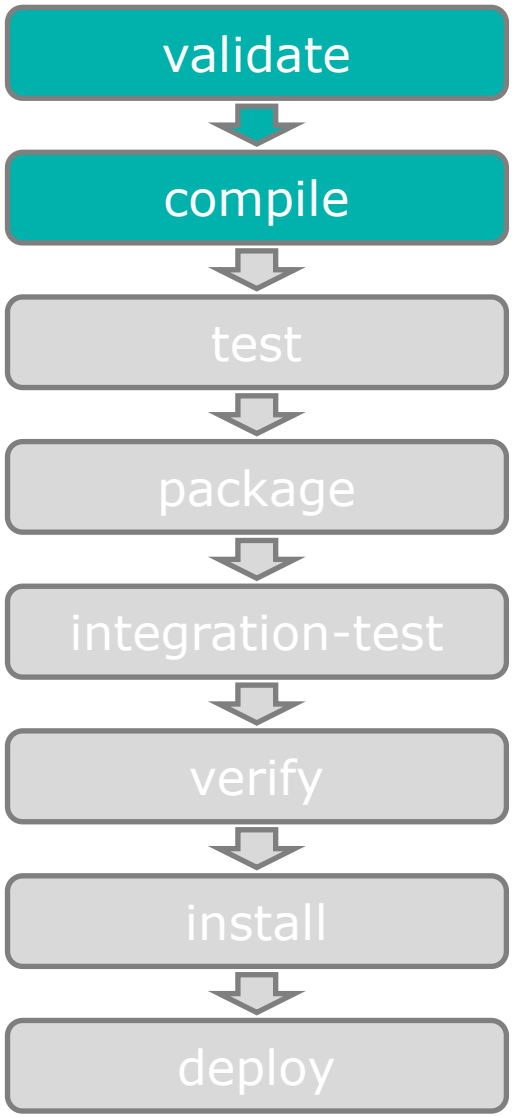
Maven aus der Vogelperspektive

pom.xml

```
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>de.fhaachen</groupId>
    <artifactId>asf</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>jar</packaging>
</project>
```

mvn compile

Der Source-Code soll kompiliert werden.



Es werden auch alle vorherigen Schritte ausgeführt

Build-Lifecycles und Maven-Kommandos

Maven aus der Vogelperspektive

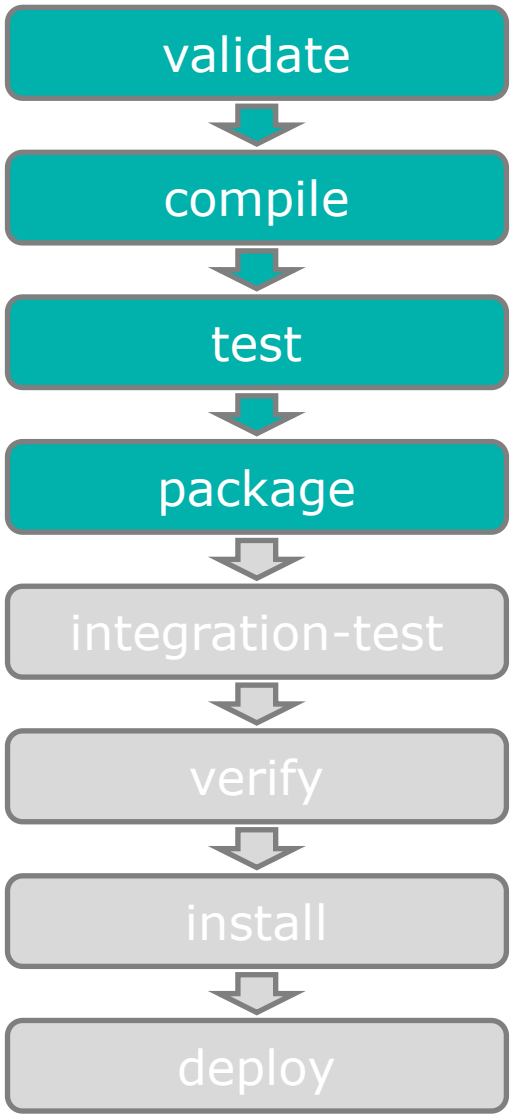
pom.xml

```
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>de.fhaachen</groupId>
    <artifactId>asf</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>jar</packaging>
</project>
```

mvn compile — Der Source-Code soll kompiliert werden.

mvn package — Das Jar-Archiv soll erstellt werden.

Es werden auch alle vorherigen Schritte ausgeführt



Build-Lifecycles und Maven-Kommandos

Maven aus der Vogelperspektive

pom.xml

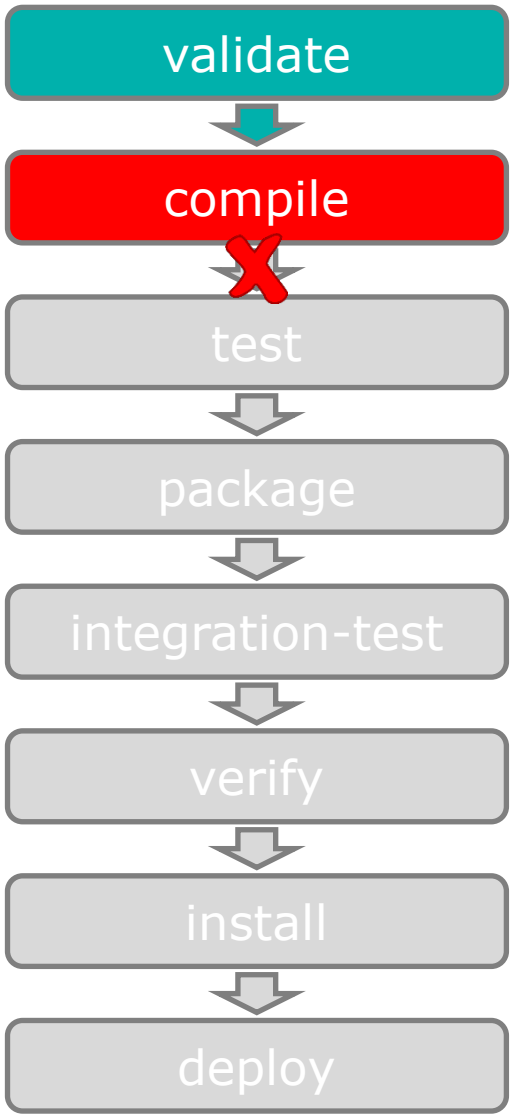
```
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>de.fhaachen</groupId>
    <artifactId>asf</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>jar</packaging>
</project>
```

mvn compile — Der Source-Code soll kompiliert werden.

mvn package — Das Jar-Archiv soll erstellt werden.

mvn install — Das Jar-Archiv soll ins lokale Repository installiert werden.

Bei Fehlern kommt es zum Abbruch!



Project Object Model

Project Object Model – Eckpunkte

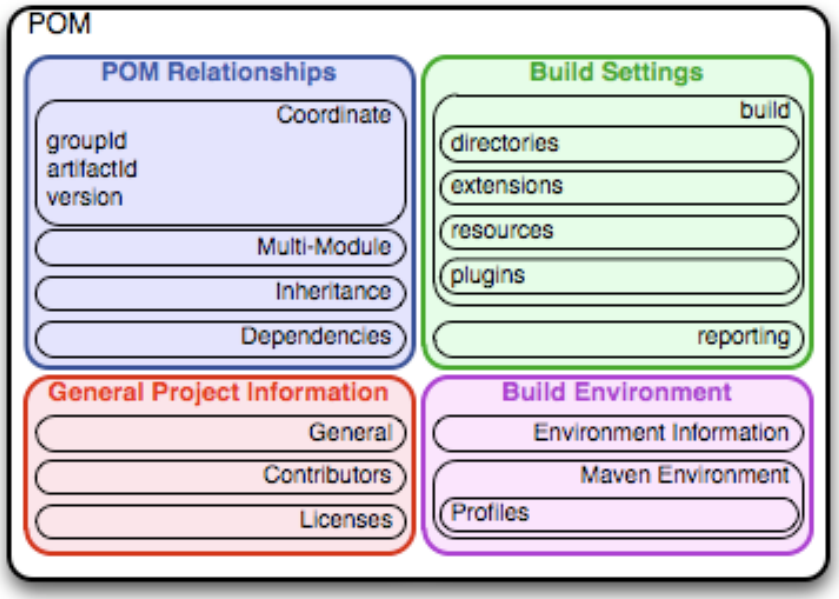
Project Object Model

- Project Object Model Datei (pom.xml) definiert das Softwareprojekt und seine Komponenten

- Deklarative Spezifikation:
  - Ort des Source-Codes
  - Liste der Abhängigkeiten
  - Projekttyp (Jar, War, etc.)

- POM ist nicht spezifisch für Java-Projekte
  - Es werden auch C/C++ oder C# Projekte mit Maven entwickelt.

Struktur der pom.xml
Project Object Model



General Project Information
- Projektname, URL, Liste der Entwickler, Lizenz

Build Settings
- Verzeichnisse von Source Code und Resourcen
- Plugins zur Steuerung des Build-Prozesses

Build Environment
- Spezifische Einstellungen für unterschiedliche Profile: **Test-Deploy vs. Deploy**

POM Relationships
- Definition der **Projekt-Koordinaten**
- Einbinden anderer Projekte über deren Projekt-Koordinaten

Definition der Projekt-Koordinaten

Project Object Model

Die Projekt-Koordinaten setzen sich aus vier Attributen zusammen:

groupId
Gruppiert eine Menge von Projekten, die z. B. von einer Organisation entwickelt werden. Bei Java-Projekten ist die groupId oft gleich mit dem Java Package.

artifactId
Identifier eines bestimmten Projekts. Kombination von groupId und artifactId muss eindeutig sein.

version
Versionsnummer des Projekts.

classifier
Optional. Kann genutzt werden, um Artefakte für verschiedene Plattformen (32/64bit) oder für verschiedene JDK Versionen bereitzustellen.

Kurzschreibweise: <groupId>[":" <artifactId>] [":" <version>] [":" <classifier>]
Beispiel: org.springframework:spring:2.5

Etablierung der Konventionen – Super POM

Project Object Model

- Alle Maven-Projekte leiten von einem Super POM ab (vgl. java.lang.Object)
- Super POM definiert Standards/Konventionen für alle Maven-Projekte
- Ausschnitt des Super POMs:

```
<project>
    <!-- ... -->
    <build>
        <!-- ... -->
        <finalName>${project.artifactId}-${project.version}</finalName>
        <!-- ... -->
    </build>
    <!-- ... -->
</project>
```

Name der Jar-Datei (ohne Extension)

- Den vollständigen Code findet man hier.

Integration von Maven in IDEs
Project Object Model

- IDEs wie Eclipse, Netbeans und IntelliJ haben eigene Schemata, wie z.B. Java Projekte definiert werden.

- Es ist z.B. nicht möglich ein Eclipse-Projekt mit Netbeans zu öffnen.

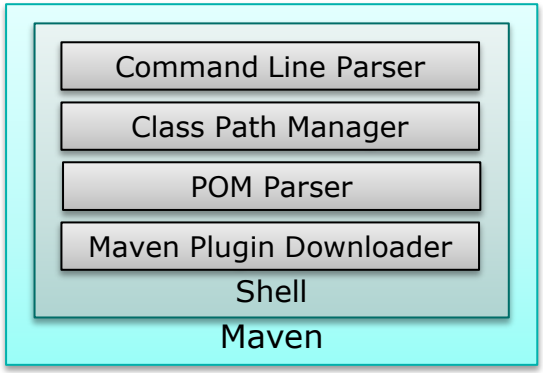
- Das POM bietet eine einheitliche Schnittstelle zum Verwenden verschiedener IDEs:
  - Netbeans und IntelliJ können nativ Maven-Projekte öffnen
  - Eclipse geht den Umweg über ein Maven-Plugin: `mvn eclipse:eclipse`

Maven Dependency Management

Grundlegende Maven Architektur

Project Object Model

- Wer Maven herunterlädt und installiert, der verwendet zunächst nur das Core-Framework von Maven



- Mit Plugins werden die eigentlichen Aufgaben durchgeführt, z. B.:
  - Jar-Archive erstellen
  - Source-Code kompilieren
  - Unit-Tests

Plugins werden über Dependency-Management heruntergeladen, sobald diese benötigt werden!

- Plugins sind das zentrale Feature von Maven
- Plugins sorgen dafür, das Maven sprachneutral ist.

Dependency Management
Project Object Model

- Bei den meisten Build-Tools müssen Abhängigkeiten zu anderer Software manuell aufgelöst werden.
    - Bibliotheken müssen manuell installiert werden (herunterladen und einbinden).

- Maven bietet einen automatischen Mechanismus.
- Dazu wurde eine Repository-Struktur etabliert.

- Es gibt zwei Arten von Repositories:
    - Remote-Repository
    - Local-Repository (siehe ~/.m2/repository)

Abhängigkeiten deklarieren

Project Object Model

```
<project>
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.1</version>
            <scope>test</scope>
        </dependency>
        <!-- Weitere dependency Tags -->
    </dependencies>
</project>
```

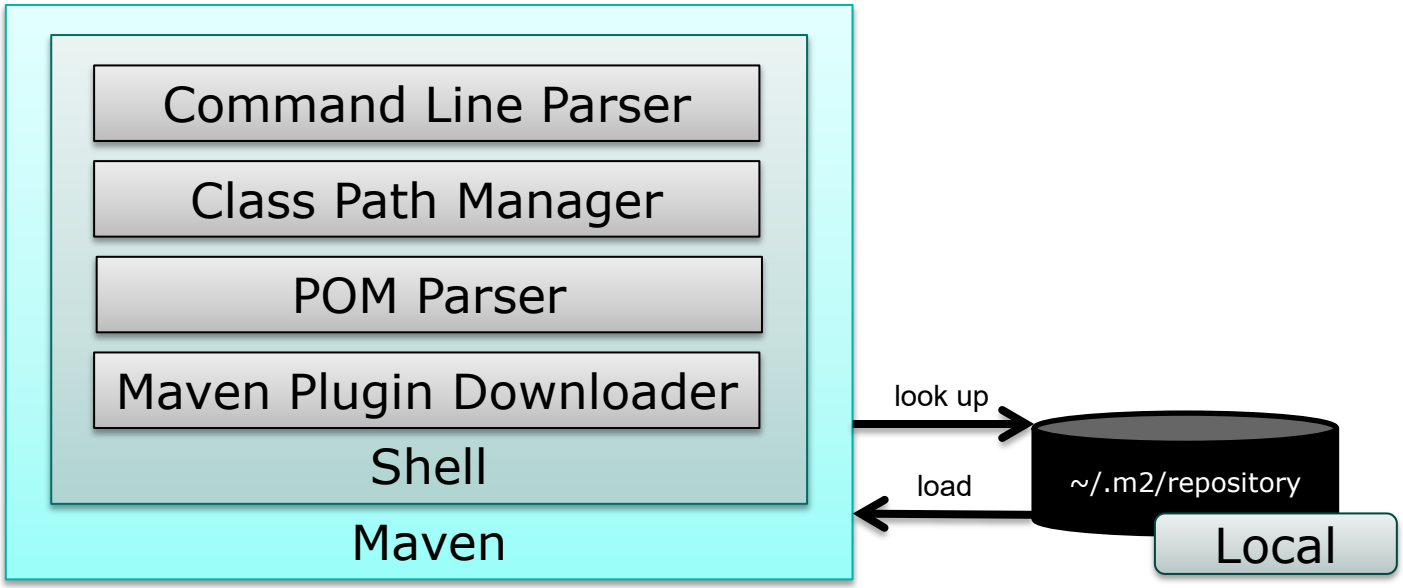
Transitive Abhängigkeiten werden automatisch mit aufgelöst.

Projekt-Koordinaten eines anderen Maven-Projekts

Gibt an, wie mit der Abhängigkeit umgegangen werden soll.

Plugin Execution Framework
Project Object Model

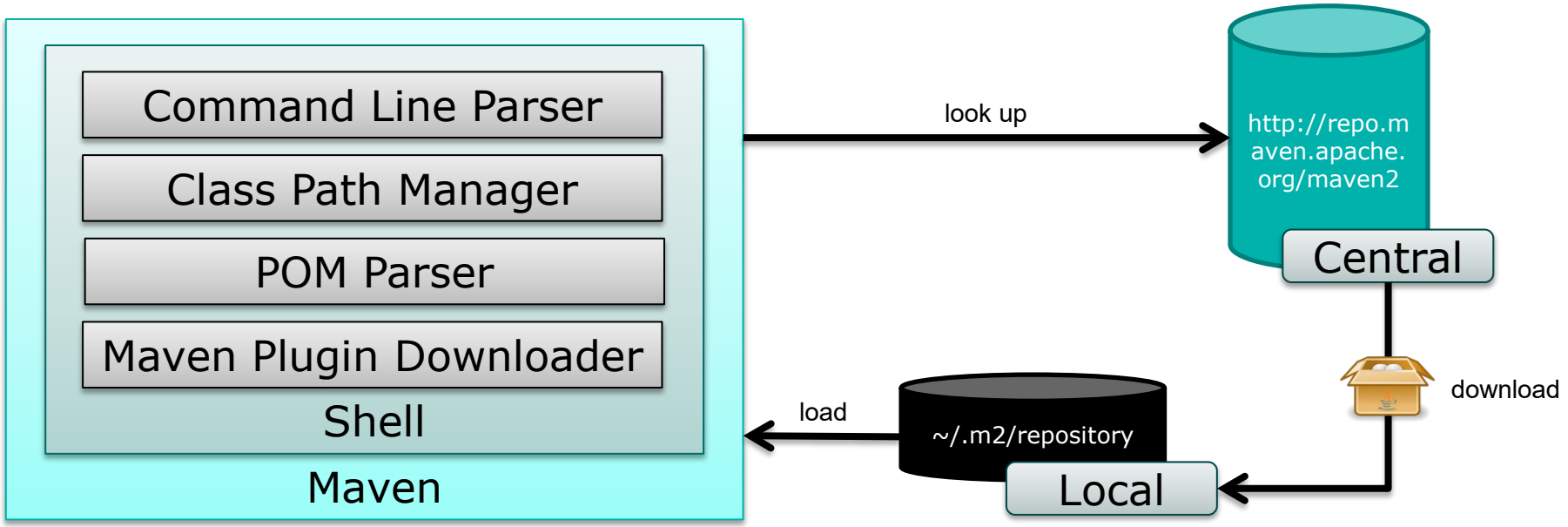
Abhängigkeiten und Plugins werden zuerst im lokalen Repository gesucht



Plugin Execution Framework

Project Object Model

Wird das Paket lokal nicht gefunden, wird es im zentralen Repository gesucht und heruntergeladen



Vielen Dank für die Aufmerksamkeit!

Fragen?