

Maven Einführung

Motivation: Das Bauen einer Anwendung

Erzeugung eines deploybaren Artefakts bzw. Anwendung aus Programmcode, Bibliotheken und sonstigen Ressourcen.

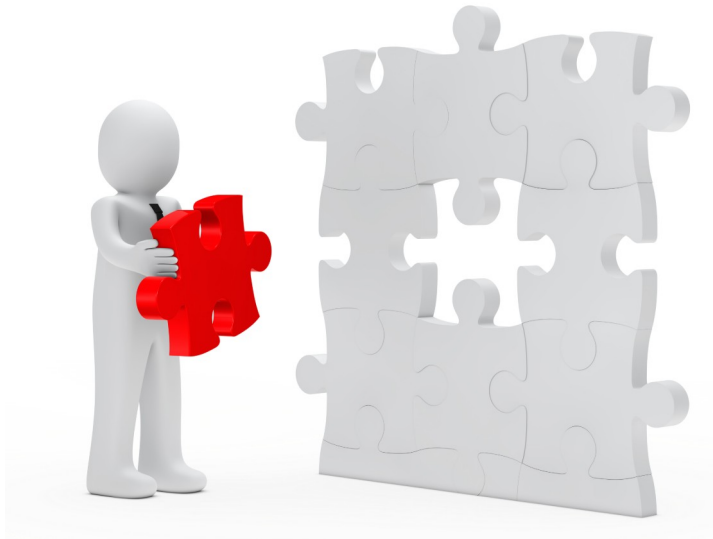


Bild von d3images auf Freepik

Problematik:

Komplexität aufgrund einer Vielzahl von Abhängigkeiten
(Bibliotheken, Versionen, IDE, ...)

Lösung:

Automatisierung des Build-Vorgangs

Build-Automatisierung

- Organisation von Bibliotheken
- Kompilierung von Source-Code in Binär-Code
- Ausführung von Tests
- Paketierung des Binär-Codes und andere Dateien in ein deploybares Artefakt bzw. Produkt
- In der Java-Welt stehen dafür im wesentlichen 3 Werkzeuge zur Verfügung
 - ANT (veraltet)
 - Maven
 - Gradle
- Aktuell (JetBrains Developer Survey 2023) populärstes Build-Werkzeug:
 - Maven 74%
 - Gradle 46%
 - ANT 6%

Was ist Maven

- Apache Maven (aktuelle Version 3) ist das beste Beispiel für ein Konfigurations-basiertes Build-Management-Tool. (Top-Level-Projekt der Apache Software Foundation)
- Von der offiziellen Webseite (<https://maven.apache.org/what-is-maven.html>):
 - Making the build process easy
 - Providing a uniform build system
 - Providing quality project information
 - Encouraging better development practices
- Es folgt der Philosophie, dass Projekte in der Regel sehr ähnlich aufgebaut sind und deshalb auch der Build-Vorgang immer ähnlich ablaufen wird.
- Maven definiert zu diesem Zweck einige Standards und einen Build-Lifecycle, der für diesen Standard ausgelegt ist.
- Man muss nur das konfigurieren, was vom Standard abweicht bzw. projektspezifisch ist (Convention over Configuration).
- Maven ist eigentlich über die Kommandozeile zu nutzen, ist aber heute zusätzlich in den üblichen Entwicklungsumgebungen integriert.

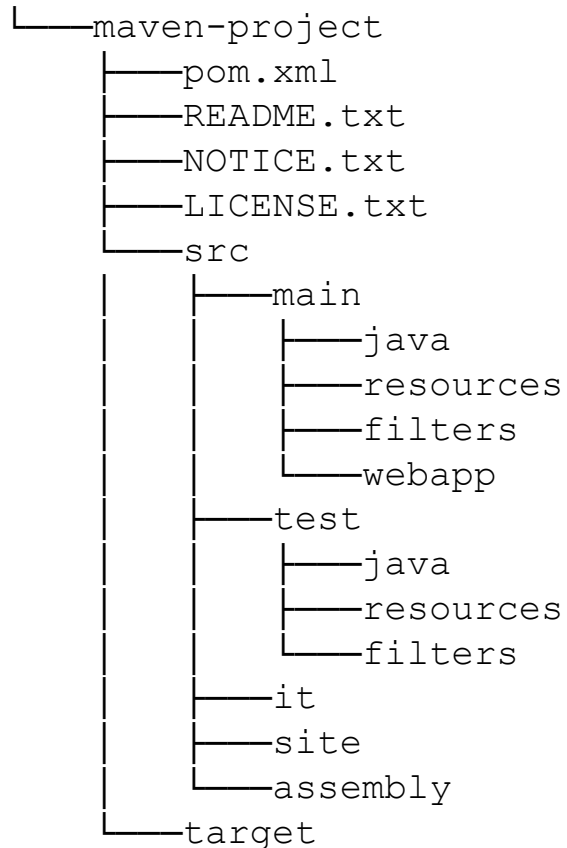
Begriffe – Teil 1

- **Project:**
Das zu bauende Software-Projekt
- **Project Object Model (POM):**
Die Metadaten, welche Maven benötigt, um das Projekt zu bauen. Üblicherweise eine "pom.xml" Datei an oberster Stelle im Projektverzeichnis.
- **Artifact:**
Etwas, was durch den Build-Prozess erzeugt oder konsumiert wird. Beispielsweise produziert der Build-Prozess ein JAR Artefakt als Output, gleichzeitig benötigt die Kompilierung auch Drittbibliotheken in Form anderer JAR Artefakte.
- **GAV:**
Jedes Artefakt ist primär durch sogenannte GAV Koordinaten (GroupId, ArtifactId, Version) eindeutig identifiziert
 - **GroupId:** ein eindeutiger Kennzeichner für eine Gruppe/Unternehmen, z.B. "org.springframework"
 - **ArtifactId:** ein eindeutiger Kennzeichner für ein Artefakt einer Gruppe, z.B. "spring-core"
 - **Version:** ein eindeutiger Kennzeichner für die Version eines Artefakt, z.B. "5.3.22".

Begriffe – Teil 2

- **Dependency:**
Eine Bibliothek, die zum Bauen und Ausführen eines Java Projekts benötigt wird, typischerweise Artefakte anderer Projekte
- **Plugin:**
Eine im Kontext von Maven ausführbare Funktion, meistens in Java programmiert. Plugins definieren Goals und nutzen die Metadaten der POM, um ihre Aufgabe auszuführen (z.B. Kompilieren, Tests ausführen, JAR Datei erzeugen)
- **Repository:** Ein Ablageort für Artefakte. Kann lokal auf dem eigenen PC, im Intranet oder im Internet existieren.

Standard-Verzeichnisstruktur



■ maven-project/pom.xml	Projektkonfiguration
■ maven-project/LICENSE.txt	Lizenz-Bedingungen
■ maven-project/NOTICE.txt	Infos zu Fremdresourcen
■ maven-project/src/main	Sourcecode
■ maven-project/src/test	Tests
■ maven-project/src/it	Integrationstests
■ maven-project/src/site	Generierte Dokumentation
■ maven-project/src/assembly	Deskriptoren für Binärcode
■ maven-project/target	Generierter Code

- Abänderung ist unüblich, kann aber konfiguriert werden.
- Aktuell nicht benötigte Verzeichnisse können auch fehlen.

POM (Project Object Model): Die projektspezifische Konfigurationsdatei

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>sk.train</groupId>
  <artifactId>java11_basis</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

- Für eine minimale Projektkonfiguration ist nur erforderlich:
 - project root Sauberes XML
 - modelVersion aktuell Version 4.0.0
 - groupId eindeutiger Identifier der Organisation/Projektgruppe (empfohlen: Java Package Naming)
 - artifactId eindeutiger Identifier für das Projekt innerhalb der „groupId“ (gibt jar-Namen vor)
 - version Versionsnummer
(Empfehlung Semantic Versioning 1.0.0: <https://semver.org/spec/v1.0.0.html>)
- group, artifact and version (GAV) sind die sogenannten Maven-Koordinaten und identifizieren das Maven-Artefakt eindeutig.
Folgende GAV-Schreibweise ist üblich: groupId:artifactId:version
Hier z.B: ***sk.train:java11_basis:1.0-SNAPSHOT***

POM Anpassungen

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>sk.train</groupId>
  <artifactId>javall_basis</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>javall_basis</name>
  <description>sample project</description>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

- Wir erben von unserer Eltern-POM , default „Maven Super-POM“, und via Voreinstellungen, können aber durchaus Anpassungen vornehmen (Convention over Configuration).
- ***mvn help:effective-pom*** zeigt uns die Effektive POM, die Maven tatsächlich benutzt.

Projekt-Informationen

- Es können übliche Informationen zum Projekt festgehalten werden.
- Beispiele:
https://maven.apache.org/pom.html#More_Project_Information

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <name>javall_basis</name>
  <description>sample project</description>
  <url>www.sampleproject.de</url>
  <inceptionYear>2023</inceptionYear>
  <licenses>
    <license> <!-- ... --> </license>
  </licenses>
  <developers>
    <developer>
      <id>mm</id>
      <name>Max Musterfrau</name>
      <email>muster@example.de</email>
      <!-- ... -->
    </developer>
  </developers>
  <contributors> <!-- ... --> </contributors>
  <organization> <!-- ... --> </organization>
  <issueManagement> <!-- ... --> </issueManagement>
  <ciManagement> <!-- ... --> </ciManagement>
  <mailingLists> <!-- ... --> </mailingLists>
  <scm> <!-- ... --> </scm>
</project>
```

Arbeitsweise

- Verschiedene Build-Vorgänge (Build Lifecycle) sind durch Maven definiert:
 - Clean Löscht alle Dateien, die bei einem vorherigen Build-Vorgang erzeugt wurden.
 - Site Generiert die Projektdokumentation und stellt diese auf dem Zielserver bereit (Deployment).
 - Default Der Standard-Vorgang, wenn keiner der anderen beiden gewählt ist.
- Jeder Build-Vorgang besteht aus einzelnen Phasen.
- Abhängig vom Paketierungsformat sind Arbeitsschritte (Goals) an die Phasen gebunden, sprich: werden innerhalb der Phase ausgeführt.
- Sogenannte Plugins realisieren ein oder mehrere Goals.
- Es lassen sich auch neue Plugins schreiben und damit neue Goals hinzufügen.
- Siehe auch: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>
<https://maven.apache.org/plugins/index.html>

Phasen des Default-Lifecycle im Überblick

Build phase	Description
validate	checks if the project is correct and all information is available
compile	compiles the source code into binary artifacts
test	executes the tests
package	takes the compiled code and package it, for example into a JAR file
integration-test	takes the packaged result and executes additional tests, which require the packaging
verify	performs checks if the package is valid
install	install the result of the package phase into the local Maven repository
deploy	deploys the package to a target, i.e. remote repository

- Wird eine Phase ausgeführt, so werden automatisch alle davor liegende Phasen gemäß obiger Reihenfolge ausgeführt und damit natürlich alle relevanten Goals der jeweiligen Phase:
 - ***mvn clean*** Aufruf des clean-Cycle
 - ***mvn verify*** Aufruf der verify-Phase und damit aller vorhergehenden Phasen
 - ***mvn clean install*** Aufruf des clean-Cycle und anschließender install-Phase
 - ***mvn compiler:compile*** Aufruf eines Goals innerhalb eines Plugins ohne vorhergehende Schritte

Ausgaben des Build-Cycles

- Einige Ausgaben werden direkt auf die Standardausgabe geschrieben, ansonsten wird SLF4J als Logging-Fassade benutzt.
- Logging erfolgt standardmäßig auf die Standardausgabe und nur für die Log-Level **info**, **warning**, und **error**. Dies kann durch Kommando-Optionen angepasst werden oder aber durch direkte Konfiguration der verwendeten Logging-Implementierung (default: {maven.home}/conf/logging/simplelogger.properties). siehe z.B: <https://www.baeldung.com/maven-logging>

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< sk.train:Maven_Basis_Java11 >-----
[INFO] Building Maven_Basis_Java11 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:3.2.0:clean (default-clean) @ Maven_Basis_Java11 ---
[INFO] Deleting E:\stephan\workspaces\intellij2\Maven_Basis_Java11\target
[INFO]
[INFO] --- maven-resources-plugin:3.3.0:resources (default-resources) @ Maven_Basis_Java11 ---
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.10.1:compile (default-compile) @ Maven_Basis_Java11 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to E:\stephan\workspaces\intellij2\Maven_Basis_Java11\target\classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.953 s
[INFO] Finished at: 2023-01-25T22:40:13+01:00
[INFO] -----
```

Dependency-Management

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

- Eine der wichtigsten Eigenschaften von Maven: Dependency-Management
- Anhand der GAV (groupId, artifactId and version) der benötigten Artefakte wird versucht, diese bereit zu stellen.
- Als Quellen werden in dieser Reihenfolge benutzt:
 - Andere Projekte im selben Maven Build (Maven reactor)
 - Lokales Repository (default: %HOME%\m2\repository)
 - Globales Repository (default: Maven Central)
- Statt globale Repositories im Web direkt zu verwenden, werden oft „Mirrors“ konfiguriert.

Scope der Dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

- Nicht jede Dependency wird für jeden Arbeitsschritt im Build benötigt.
- Folgende Scopes sind definiert:
 - compile (default) in allen Phasen verfügbar und mit deployed (propagated)
 - provided nur beim Compile und Test verfügbar, zur Laufzeit in der Zielumgebung verfügbar (nicht transitiv)
 - runtime nur zur Laufzeit bereit zu stellen, d.h. verfügbar bei Test und Runtime, aber nicht bei Compile
 - test nur bei Test verfügbar (nicht transitiv)
 - system wird durch das aktuelle System bereit gestellt und nicht in einem Repository gesucht
 - import Spezialfall bei Übernahme aus Parent-POM
- siehe auch: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

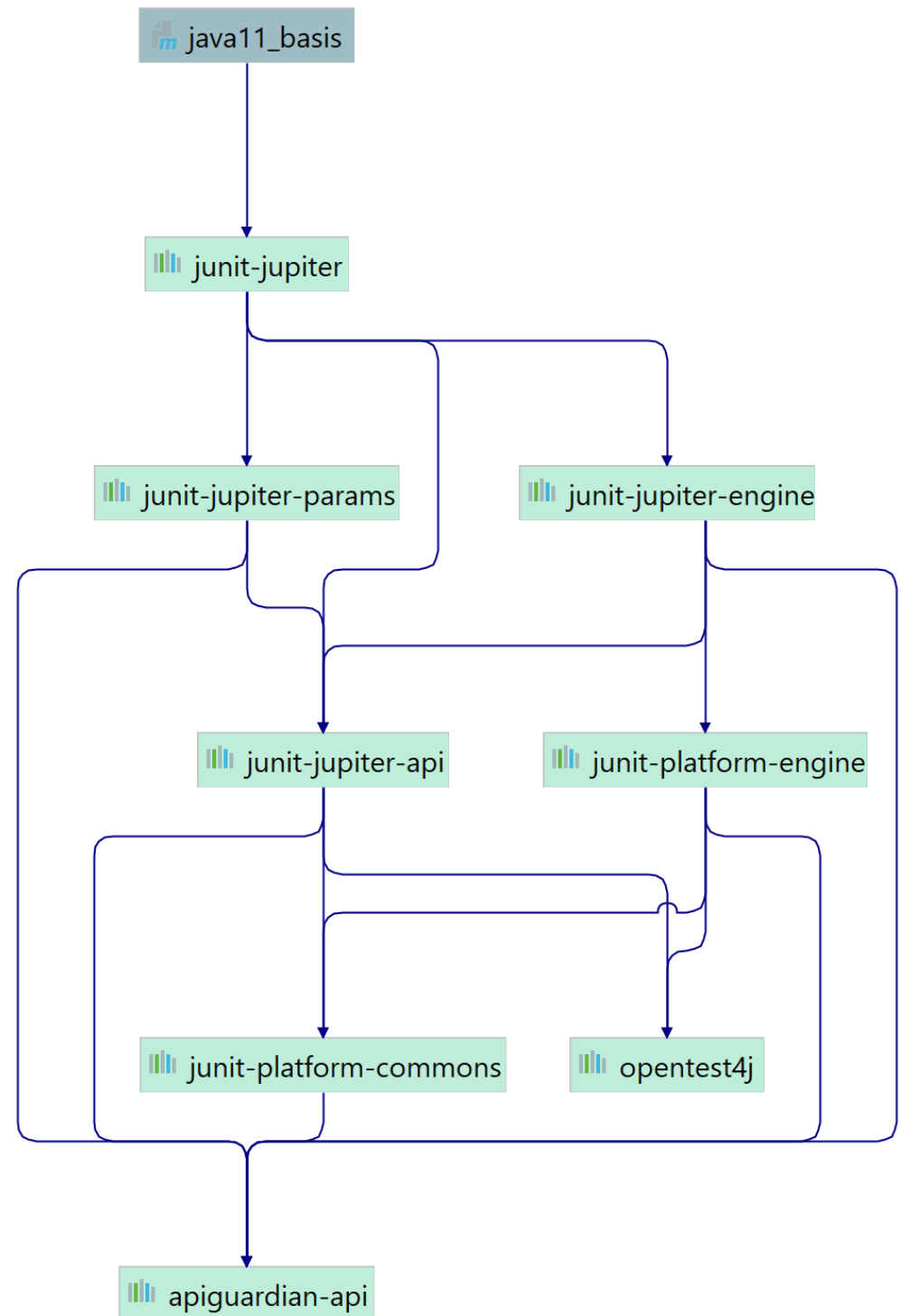
Beispiel Scope system

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependency>
    <groupId>mylib</groupId>
    <artifactId>mylib</artifactId>
    <scope>system</scope>
    <version>1.0</version>
    <systemPath>${basedir}\libs\mylibrary.jar</systemPath>
  </dependency>
  ...
</project>
```

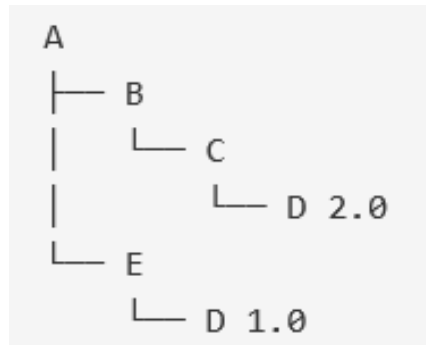
- Damit können lokal verfügbare Artefakte adressiert werden.
- Es muss dann via systemPath-Element der Pfad angegeben werden (*`${basedir}`* ist Standard-Property)

Transitive Dependencies

- Maven versucht auch die Artefakte bereit zu stellen, die wiederum von unseren direkten Dependencies benötigt werden.
- Zu Beginn wird Maven deshalb erstmal eine Menge von Artefakten in das lokale Repository laden!
- Anzeige des aktuellen Dependency-Tree:
mvn dependency:tree
(geht in der IDE meist hübscher)



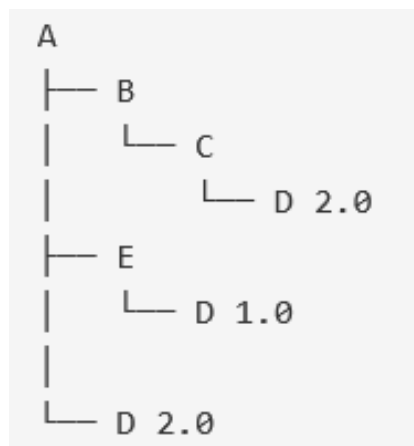
Transitivität und Konflikte



- Maven wählt die bzgl. der Ebenen und innerhalb einer Ebene nach der Reihenfolgen am nächsten (nearest) liegende Dependency aus.

Im Beispiel: D 1.0

- Das ist nicht immer die gewünschte Version!!



- Wir können aber stets die gewünschte Version als direkte Dependency formulieren (auch wenn wir diese nicht direkt benötigen).

Im Beispiel: D 2.0

- Dependencies können auch gezielt ausgeschlossen (excluded) werden bzw. als optional markiert werden.

Details siehe:

<https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html>

POM Inheritance

- Das Packaging muss *pom* sein im Parent-Projekt
- Elemente, die vererbt werden:
 - Dependencies
 - Developers und Contributors
 - Plugin-Listen
 - Plugin-Executions mit korrespondierenden IDs
 - Plugin Konfiguration
 - ...
- Sofern das Parent-Projekt nicht das Elternverzeichnis bzw. im lokalen Repository abgelegt ist, muss der Pfad angegeben werden.
- Details siehe:
<https://maven.apache.org/pom.html>

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <groupId>sk.train</groupId>
  <artifactId>maven-parent-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <!-- This needs to be set -->
  <packaging>pom</packaging>
  ...
</project>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <!-- Here we define the parent -->
  <parent>
    <groupId>sk.train</groupId>
    <artifactId>maven-parent-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!-- This is optional -->
    <relativePath>../maven-parent-project</relativePath>
  </parent>

  <artifactId>maven-inheritance-example</artifactId>
  <!-- version und groupId können vererbt werden -->
  ...
</project>
```

Bill Of Materials (BOM)

- Eine BOM ist eine Zusammenstellung von Dependencies, die bzgl. Versionen und Features zusammen passen.
- Eine BOM wird durch eine POM mit einer entsprechenden **<dependencyManagement>**-Sektion bereit gestellt.
- Paketierungsformat ist **pom**.
- Viele Frameworks nutzen diese Art Bereitstellungs-Mechanismus, z.B:
 - spring-data-bom:
BOM for Spring Data project
 - jackson-bom:
BOM for Jackson dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <groupId>sk.train</groupId>
  <artifactId>maven_sampleBOM</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <dependencyManagement>
    <dependencies>
      <!-- testing dependencies -->
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
      </dependency>
      <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>5.0.0</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```