HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

Dependency Management

Roland Gisler

FH Zentralschweiz

Inhalt

- Einleitung
- Maven Repositories.
- Dependency Management bei Java.
- Dependency Management mit Apache Maven.
- Dependency Scopes.
- Transitive Dependencies und Konflikte.
- Versionierung von Dependencies / Snapshots.
- «Managed Dependencies» in Multimodul-Projekten.
- Deployment von Dependencies.

Lernziele

- Sie haben ein grundsätzliches Verständnis von Dependency Management.
- Sie wissen wie am Beispiel von Java und Apache Maven das Dependency Management funktioniert.
- Sie sind mit den Begriffen «dependency scopes» und «transitiv dependencies» vertraut und können diese erklären.
- Sie kennen das Versionskonzept und die Funktionsweise von Snapshots.
- Sie wissen auf welche Art Dependencies deployed werden.

Einleitung

Einleitung - Dependency Management (DM)

- Das Dependency Management beschreibt die Organisation und die Techniken für den Umgang mit Abhängigkeiten zu anderen Modulen.
  - Modul wird in diesem Kontext vereinfacht als Überbegriff für Package, Library, Bundle oder Komponente verwendet.
  - Häufig wird auch der Begriff «Package Management» verwendet.
- Abhängigkeiten können sowohl auf interne und externe Modulen bestehen.
  - Intern: Modul im selben Projekt.
  - Extern: Dritt-Modul, aus anderem Projekt oder Organisation.
- Abhängigkeiten werden typisch in binärer (kompilierter) Form aufgelöst. Deshalb kommen dafür so genannte Binär-Repositories und Packagemanager(-Tools) zum Einsatz.

Beispiele für Dependency Management (DM)

- Einige populäre Systeme / Repositories für DM und PM:
  - **NuGet** – Package Manager für .NET-Plattform.
  - **apt** – Advanced Packaging Tool – Packetverwaltung für Linux.
  - **Yum** – Yellowdog Updater, Modified – Packetverwaltung Linux.
  - **P2** – OSGi-basiertes Komponentensystem, Eclipse Equinox.
  - **npm** – Node Package Manager für JavaScript / node.js.
  - **Gems** – Packetmanager für Ruby.
- Allen gemeinsam ist: Zentrale Ablage auf Server,
  ein standardisiertes Format, zusätzliche Metainformationen,
  typisch mit Abhängigkeiten (Dependencies) versehen,
  Sicherung der Konsistenz (z.B. über hash-Mechanismen),
  geregelte Zugriffsprotokolle, Suchmöglichkeiten etc.

Maven Repository

Maven Repository

- Es gibt verschiedene öffentliche Repos (OSS):
  - Maven Central: https://search.maven.org/ (das Original!)
  - Maven Central: https://central.sonatype.dev/ - neues UI
  - Google Maven Repo: https://maven.google.com/
- Natürlich hat man auf öffentliche Repo's keine Schreibrechte!
  - Bzw. nur ausgewählte Personen über definierte Prozesse.
- Organisationen betreiben typisch **interne** Repositories.
  - Ursprünglich realisiert als einfach Dateisysteme / Webserver.
- Professionelle Produkte mit diversen Zusatzfunktionen:
  - Apache Archiva - https://github.com/apache/archiva
  - JFrog/Artifactory - https://jfrog.com/community/open-source/
  - Sonatype/Nexus - https://de.sonatype.com/products/nexus-repository/

Maven Repository an der HSLU

- Firmen und Organisationen nutzen ein eigenes Repository oft auch als Mirror von öffentlichen Repositories.
- HSLU RepoHub Nexus: https://repohub.enterpri
  - Dient auch als Mirror zu öffentl
- Alle herunt zudem in einem lokalen
  rechner gespeichert (caching).
  Repository: **`$HOME/.m2/repository`**

Jetzt aber wirklich ALLE! ;-)

- Damit dieses HSLU-Repository verwendet wird, müssen Sie in das Verzeichnis **`$HOME/.m2`** die Konfigurationsdatei **`settings.xml`** (auf ILIAS zur Verfügung gestellt) kopieren!
  - Haben Sie das gemacht? Kontrolle: Achten Sie während des Builds auf die URL's beim Download der Dependencies.

Dependency Management für Java

Dependency Management für Java

- Binäre Module (kompilierte Projekte) werden bei Java typisch als JAR-Dateien (oder EAR, WAR, RAR, JMOD etc.) ausgetauscht.
- Java selber kennt zwar seit Version 9 neue Mechanismen zur Modularisierung und Definition von Abhängigkeiten, das umfasst aber **nicht** die Versionierung und die Ablage.
- Ursprünglich wurden deshalb die JAR-Dateien einfach von Hand z.B. in **`/lib`**-Verzeichnisse direkt in die Projekte kopiert.
  - Fehleranfällig, hohe Redundanz, grosser Platzbedarf, …
  - Ergebnis: «JAR Hell», wenn sich Entwickler*innen nicht aktiv um die Abhängigkeiten (im Classpath) gekümmert haben.
- Im Jahr 2001 führte Apache Maven mit seinem Buildsystem auch ein einfaches Dependency Management ein, das zum Quasi-Standard wurde und sehr populär ist.

Dependency Management (DM) mit Maven Repositories

- Grundsätzlich sollte man Unterscheiden zwischen:
  - Das **Format** für die zentralen Ablage der meist binären Artefakten mit zusätzlichen Metainformation im ➔Repository.
  - Das **Werkzeug** welches es ermöglicht, Artefakte von Repositories zu suchen, zu beziehen, zu deployen und ggf. auch zu verwalten.
- Während beim Format der Repositories Maven zum Quasi-Standard geworden ist, gibt es bei den Werkzeugen eine grosse Vielfalt:
  - Apache Ivy – einziges «reines» Dependency-Management Tool.
  - Apache Maven – in Buildtool integriert, das «Original».
  - Das DM zahlreicher weiterer Buildtools basiert auch auf Maven-Repos: Buildr, Groovy Grape, Gradle/Grails, SBT etc.

Dependency Management mit Apache Maven

Maven - Identifikation

Eine Maven Projekt identifiziert sich über drei Attribute, die als «maven coordinates» bezeichnet werden:

- **GroupId**: Meistens zusammengesetzt aus dem «reverse domain name» der Organisation und einem Zusatz für eine OE, eine Projektgruppe etc. (➔ grosse Ähnlichkeit zu Java Package Name)
  Beispiel: `ch.hslu.vsk`
- **ArtifactId**: Entspricht häufig dem Namen des Projektes bzw. den darin enthaltenen Modulen.
  Beispiel: `stringpersistor-api`
- **Version**: Empfohlen wir eine dreistellige Versionsnummer (Semantic Versioning, http://semver.org).
  Beispiel: `6.0.2`

Maven Coordinates

- Die Identifikation eines Projektes ist somit eines der wichtigsten Pflichtelement in einem POM (**pom.xml**):

```
<groupId>ch.hslu.vsk</groupId>
<artifactId>stringpersistor-api</artifactId>
<version>6.0.2</version>
```

- Mittels diesen Koordinaten wird/soll eine Dependency weltweit absolut eindeutig identifiziert werden können!

- Weitere Beispiele in Kurzform:
  - **org.apache.logging.log4j:log4j-api:2.19.1** – ok.
  - **nl.jqno.equalsverifier:equalsverifier:3.10.1** – ok.
  - **junit:junit:4.12** – eine alte «Sünde».
  - **org.junit.jupiter:junit-jupiter-api:5.9.1** – besser!
  - **ch.hslu.vsk:stringpersistor-api:6.0.2** – perfekt! ☺

Deklaration von Dependencies im POM

- Benötigte Dependencies können im POM (**pom.xml**) eines Projektes im Element **<dependencies/>** eingetragen werden:

```
<dependency>
    <groupId>ch.hslu.vsk</groupId>
    <artifactId>stringpersistor-api</artifactId>
    <version>6.0.2</version>
    <scope>compile</scope>
</dependency>
```

- Diese werden beim Build automatisch vom Repository (default: Maven Central) herunter geladen, und im lokalen Repository (**$HOME/.m2/repository**) gespeichert.
- Der Buildprozess referenziert die Artefakte (typisch: JAR-Dateien) dort mit einem entsprechenden Classpath.

Dependency Scopes

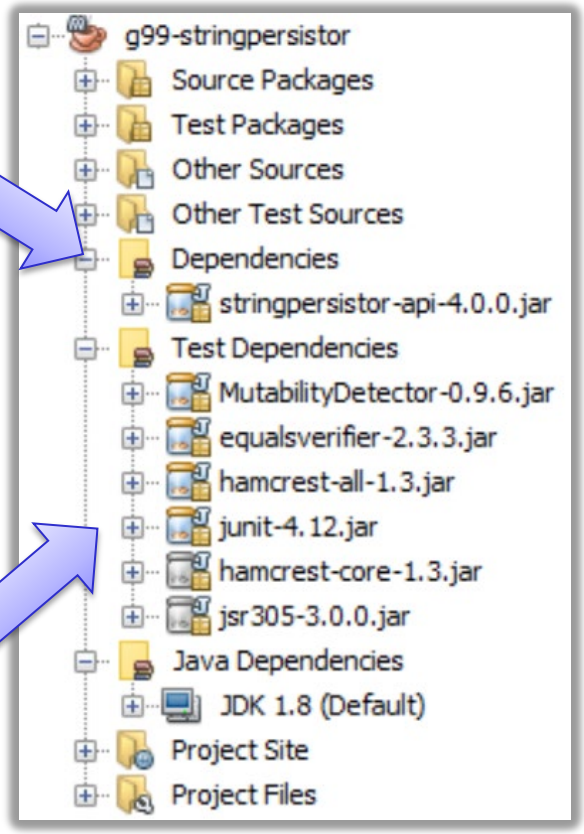
Dependency Scopes

- Im Beispiel oben fällt das Element `<scope>compile</scope>` auf, das optional pro Abhängigkeit definiert werden kann.
  - Damit wird der Zweck und Geltungsbereich (➔ Scope) der Dependency qualifiziert, **was unbedingt empfohlen wird!**
- Maven kennt verschiedene Scopes (hier nur die wichtigsten drei):
  - `compile` – Dependency wird für die Kompilation und zur Laufzeit des Programmes benötigt (Default).
  - `test` – Dependency ausschliesslich für die Kompilation und Ausführung der Testfälle (Beispiele: JUnit, AssertJ etc.).
  - `runtime` – Dependency nur für Laufzeit, aber nicht für Kompilation, z.B. für dynamisch geladene Implementationen.
- Aus den Scopes werden in Maven **spezifische** Classpaths!
  - Daraus ergibt sich eine **implizite Verifikation** des Designs.

Dependency Scopes in der IDE

Hier zeigen sich die IDE's unterschiedlich Intelligent:

- NetBeans ist derzeit die einzige IDE welche die Scopes nicht nur visualisiert, sondern auch die daraus resultierenden, getrennten Klassenpfäde während der Entwicklung aktiv unterhält und nutzt.
- Dadurch wird absolut zuverlässig vermieden, dass z.B. in einer produktiven Klasse (im Pfad **`src/main/java`**) eine Referenz auf eine Klasse aus einer Dependency vom **test**-Scope erstellt werden kann!



➔ Qualitätssicherung des Design's einer Software.

Transitive Dependencies

Transitive Dependencies und Konflikt-Resolving

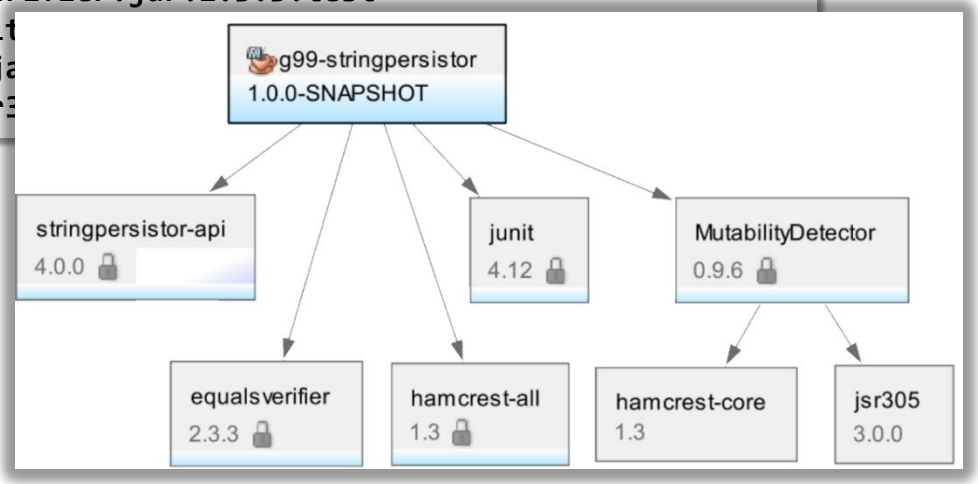
- Ein sehr nützliches Feature des Maven-DM ist die automatische Auflösung von so genannten **transitiven** Dependencies:

- Auflösung der Dependencies:
  - Modul **A** ist von Modul **B**, und dieses von Modul **C** abhängig.
  - Modul **A** ist somit **transitiv** auch von Modul **C** abhängig.
  - Bei Kompilation von A wird Modul **C** somit auch miteinbezogen.
- Durch direkte oder transitive Abhängigkeiten können auch (Versions-)Konflikte oder Zyklen auftreten!
  - Maven erkennt solche Konflikte und meldet diese.
  - Einfachere Versionskonflikte werden (durch wählbare Strategien) automatisch aufgelöst.

Dependency-Graph

- Da Maven die Dependencies detailliert auswerten muss, werden diese als Graph ausgewertet ➔Graphentheorie im Modul AD.
  - Suche von Zyklen (die nicht erlaubt sind).
  - Auflösung von (Versions-)Konflikten z.B. über kürzesten Pfad.
- Der Graph steht sowohl über CLI als auch grafisch zur Verfügung:

```
[INFO] --- maven-dependency-plugin:3.0.1:tree (default-cli) @ g99-stringpersistor ---
[INFO] ch.hslu.vsk17hs.g99:g99-stringpersistor:jar:1.0.0-SNAPSHOT
[INFO] +- ch.hslu.vsk:stringpersistor-api:jar:4.0.0:compile
[INFO] +- junit:junit:jar:4.12:test
[INFO] +- org.hamcrest:hamcrest-all:jar:1.3:test
[INFO] +- nl.jqno.equalsverifier:equalsverifier:jar:2.3.3:test
[INFO] \- org.mutabilitydetector:Mutabilit
[INFO]    +- org.hamcrest:hamcrest-core:ja
[INFO]    \- com.google.code.findbugs:jsr3
```



Versionierung und Snapshots

Versionierung

- Grundsätzlich sind alle Dependencies versioniert.
- Der Einsatz von «Semantic Versioning» (http://semver.org) wird empfohlen und von vielen Plugins und Dependencies auch eingehalten.
- Auf Basis dieser Semantik kann der Dependency Resolver diverse Automatisierungen und Vereinfachungen anbieten:
  - Erkennen von neueren Versionen.
  - Automatische Verwendung des neusten Bugfixes.
  - Angabe von Versionsbereichen welche Kompatibel sind etc.
- Gute Repositories sind so konfiguriert, dass eine einmal deployte Version **nicht** mehr überschrieben werden kann!
  - Das garantiert wirklich nachvollziehbare Buildprozesse!

Semantic Versioning (semver.org)

- **Major**-Release (**X**.x.x): Veränderungen in der API, in der fachlichen Funktion und/oder in der Konfiguration, welche zu früheren Versionen nicht kompatibel sind.
  - In den meisten Fällen sind Anpassungen notwendig.
- **Minor**-Release (x.**X**.x): Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration, welche aber vollständig Rückwärtskompatibel sind.
  - Ohne Nutzung der Neuerungen keine Anpassungen notwendig.
- **Bugfix/Maintenance**-Release (x.x.**X**): Reine Korrekturen oder Änderungen in der Implementation, voll rückwärtskompatibel, keinerlei neue Funktionen, keine veränderte Funktionen.
  - Direkter, sofortiger Einsatz möglich bzw. notwendig (Bugfix).

Versionierung mit Snapshots

- In einer dynamischen Entwicklungsphase sind fixe Versionen aber eher hinderlich, die Versionierung würde sonst (für jede kleinste Änderung) förmlich «gallopieren» müssen.
  - Es würde eine Unmenge von (unnützen) Versionen produziert, welche später auch nie mehr benötigt werden.
- Darum wurde das **Snapshot**-Konzept integriert:
  Sobald man einer Version den Appendix **`-SNAPSHOT`** trägt, gilt diese als «erneuerbar» und (noch) **nicht** stabil, sondern in Entwicklung.
  - Sie wird bei **jedem** Build **immer wieder** vom Repository aufgelöst und aktualisiert.
  - Im Repository sind Snapshots mit einem Timestamp versehen.
- Beispiel: **`6.0.3-SNAPSHOT`**
  - Die noch nicht stabil veröffentlichte, zukünftige Version **`6.0.3`**

«Managed Dependencies» in Multimodul-Projekten

Multimodul-Projekt

- Bei Projekten die aus mehreren Submodulen bestehen, können mehrere Submodule von der gleichen Dependency abhängig sein.
  - Beispiel: Jedes Submodul verwendet z.B. Log4J oder JUnit.
- Es macht sehr viel Sinn (bzw. es ist technisch sogar notwendig) dass in jedem (Sub-)Modul die **selbe** Version verwendet wird.
  - Dependencies (GroupID, ArtifactID, Version und Scope) werden aber in jedem Modul **redundant** definiert ➔ **Schlecht**!
- Lösung: Im übergeordneten (Master-)POM kann über das Element **`dependencyManagement`** eine Liste von Dependencies inkl. Version und Scopes als «Baseline» oder «Valid Version Set» vordefiniert werden.
  - Submodule müssen dann nur noch Group- und ArtifactId angeben. Rest wird vom Parent-POM einheitlich vererbt.

Managed Dependencies - Beispiel

- Definition im (Parent-)POM:

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.apache.logging.log4j</groupId>
         <artifactId>log4j-api</artifactId>
         <version>2.19.1</version>
         <scope>compile</scope>
      </dependency>
   </dependencies>
   ...
</dependencyManagement>
```

- Nutzung in einem Dependencies-Element, im (Sub-)POM:

```
<dependencies>
   <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      // Version und Scope entfällt, weil von Parent geerbt!
   </dependency>
</dependencies>
```

Managed Dependencies – Diskussion

- Diese Technik kann auch in einem Single-Modul Projekt angewandt werden, wobei der Nutzen dort etwas weniger offensichtlich ist.
- Tatsächlich wurde das schon im «**`oop_maven_template`**» aus den Modulen OOP und AD so praktiziert.
  - ➔Vereinfacht eine spätere Migration auf ein Mehrmodul-Projekt.
  - ➔Versionen können quasi «vordefiniert» werden.
- Eine alternative Technik ist die Verwendung einer so genannten «**b**ill **o**f **m**aterial»-Abhängigkeit (BOM).
  - Damit können verschiedene Versionen in einer «virtuellen» Release-Unit quasi als «Baseline» referenziert werden.
  - Der Lieferant bestimmt welche zueinander passenden Versionen zum Einsatz kommen sollen, wenn wir sie denn benötigen.
  - Diese BOM wird selber als (versionierte) Abhängigkeit definiert.

Deployment

Deployment von Java Modulen

- Die häufigste Art von Deployment sind JAR-Dateien.
- Beispiel für Artefakt **ch.hslu.vsk:stringpersistor-api:6.0.2**:
  - POM (Metainformationen): **stringpersistor-api-6.0.2.pom**
  - JAR (Binary): **stringpersistor-api-6.0.2.jar**
  - JavaDoc: **stringpersistor-api-6.0.2-javadoc.jar**
  - Source (bei OSS): **stringpersistor-api-6.0.2-sources.jar**
  - Zu allen Artefakten werden noch Hashes produziert.
- Dass die Quellen und die JavaDoc auch als JAR (eigentlich ZIP) geliefert werden ist Konvention und kann Unwissende verwirren.
  - Letztlich aber egal, die Einheitlichkeit ist wichtiger!
- Vorteil, z.B. für Entwicklungsumgebungen:
  Es ist implizit klar, wo die Dokumentation und ggf. der Source für ein bestimmtes JAR gefunden wird ➔ Selbstkonfiguration.

Deployment in Maven Repositories

- Das Deployment in öffentliche Repositories (z.B. Maven Central) wird sehr restriktiv gehandhabt, weil danach nichts mehr verändert werden darf (auch kein Löschen!)
  - Stabilität von Builds muss gewahrt bleiben!
- In Firmen kann man durchaus (z.B. veraltete) Artefakte auch mal löschen, aber auch das muss sehr gewissenhaft erfolgen.
  - «Repository-Pflege» wird häufig unterschätzt oder vergessen!
- Sehr oft dürfen Entwickler*innen (zurecht) nicht direkt deployen!
- Man bedient sich stattdessen eines nachvollziehbaren, automatisierten und verifizierbaren Release-Prozesses, welcher von einem ➔Buildserver ausgeführt wird.
- Auch bei uns in VSK wird das so gehandhabt!

Demo?

Weiterführende Informationen

- Eine gute Übersicht zum Thema Dependency Management erhalten Sie in der Dokumentation von Apache Maven: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

Fragen