

# Objektorientierte Programmierung

Hochschule Bochum

WS 19/20

Dr.-Ing. Darius Malysiak


# V. OOP Software Engineering

## Maven:

Struktur einer einfachen POM File:

```
<?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>playground</groupId>
  <artifactId>playground</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>RELEASE</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```



Dieses Artifact wird zur Compile-Phase benötigt

# V. OOP Software Engineering

## Maven:

Scopes in Maven:

- **Compile** (Default falls kein Scope spezifiziert wird): Artifact wird in den alle Classpaths gelegt.
- **Runtime**: Artifact wird in den Runtime & Test Runtime Classpath gelegt.
- **Test**: Artifact wird in den Test Compile & Test Runtime Classpath gelegt.
- **Provided**: Artifact ist nur beim Kompilieren und der Testausführung verfügbar, es wird nicht in die finale JAR Datei gelegt (falls Abhängigkeiten eingebettet werden sollen). Es wird erwartet, dass die Abhängigkeiten zur Laufzeit aufgelöst werden.
- **System**: Wie provided nur mit expliziter Pfadangabe zur JAR Datei.
- **Import**: Ähnlich zu System jedoch werden Abhängigkeiten in Abschnitt <dependencyManagement> von Maven verwaltet.


Der Scope definiert in welche Klassenpfade die zum Modul gehörenden JAR Dateien gelegt werden. Darüber hinaus wird definiert ob Abhängigkeiten bei explizitem Wunsch in die finale JAR Datei eingebettet werden.

# V. OOP Software Engineering

## Maven:

Einbetten der Abhängigkeiten:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>
```













Suffix für finale JAR Datei mit eingebetteten Abhängigkeiten

Kompiliert werden kann über einen expliziten Aufruf von ***mvn clean compile assembly:single***

# V. OOP Software Engineering

## Maven:

Einbetten der Abhängigkeiten:

Name	Änderungsdatum	Typ	Größe
 archive-tmp	28.12.2018 12:04	Dateiordner	
 classes	28.12.2018 12:02	Dateiordner	
 generated-sources	28.12.2018 12:02	Dateiordner	
 generated-test-sources	28.12.2018 12:06	Dateiordner	
 maven-archiver	28.12.2018 12:07	Dateiordner	
 maven-status	28.12.2018 12:02	Dateiordner	
 surefire-reports	28.12.2018 12:06	Dateiordner	
 test-classes	28.12.2018 12:06	Dateiordner	
 playground-1.0-SNAPSHOT.jar	28.12.2018 12:07	Executable Jar File	2.160 KB
 playground-1.0-SNAPSHOT-jar-with-dependencies.jar	28.12.2018 12:04	Executable Jar File	2.568 KB

# V. OOP Software Engineering

## Maven:

Welche Klasse wird beim Starten einer JAR Datei verwendet?

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>solutions.exercise10.Main</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Dies ist natürlich nur relevant für ausführbare Applikationen. Achtung auch diese werden als Artifacts bezeichnet!

# V. OOP Software Engineering

## Maven:

Phasen und Ziele:

Maven Phase

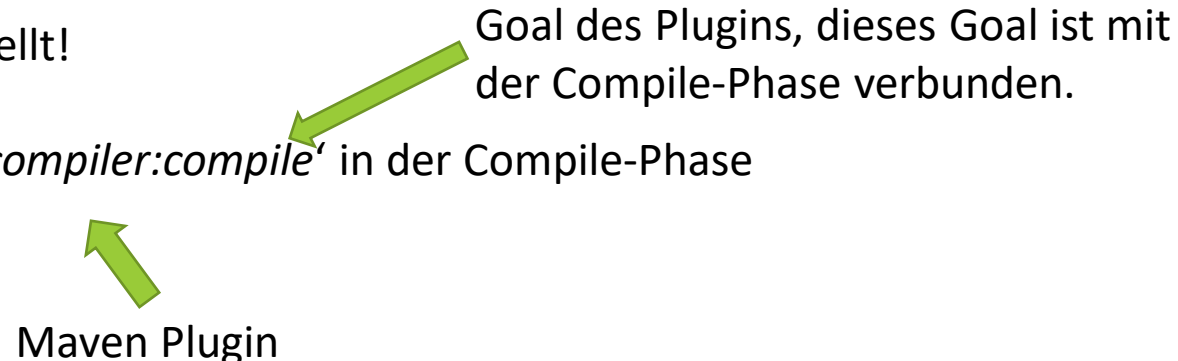


- **mvn compile**  
Alle Java Sources werden zu class Dateien kompiliert.
- **mvn package**  
,mvn compile' + die class Dateien werden in eine finale JAR Datei eingebettet.
- **mvn install**  
,mvn package' + die finale JAR Datei wird in den lokalen Maven Cache kopiert/installiert.
- **mvn deploy**  
,mvn install' + die finale JAR Datei wird in ein Maven Repository gepushed (z.B. lokales Artifactory, Maven Central)

# V. OOP Software Engineering

## Maven:

Phasen und Ziele:

- Jede Maven Phase besteht aus mindestens einem Ziel (Goal).
  - Goals werden durch Plugins bereitgestellt!
  - Jede Phase hat ein Default-Goal, z.B. `compiler:compile` in der Compile-Phase
- 
- Maven Plugin
- Goal des Plugins, dieses Goal ist mit der Compile-Phase verbunden.
- Maven Phasen: validate, package, compile, install, deploy, test, test-compile, integration-test,... (siehe [http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle\\_Reference](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference) )
  - Maven Phasen können erweitert bzw. konfiguriert werden.
  - *Theoretisch* können neuen Phasen hinzugefügt werden. -> Sehr Komplex!



# V. OOP Software Engineering

## Maven:

Phasen und Ziele:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>...</configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Hier wird in der Package-Phase zusätzlich eine konsolidierte JAR Datei erstellt.

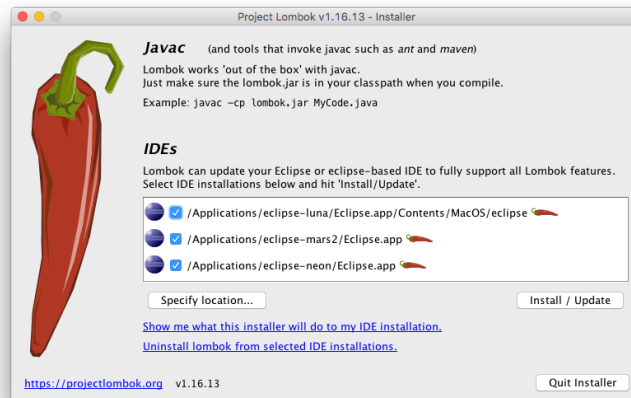
Die Standard-Abfolge der Phasen ist:

1. Validate
  2. Compile
  3. Test
  4. Package
  5. Verify
  6. Install
  7. Deploy
- } Default Lifecycle

# V. OOP Software Engineering

## Lombok:

Die Verwendung von Lombok wird aufgrund des AST-Hacks bei einigen IDEs erschwert. Hier muss Lombok zunächst installiert werden.



<https://projectlombok.org/setup/eclipse>

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.4</version>
  <scope>provided</scope>
</dependency>
```

# V. OOP Software Engineering

## Lombok:

```
@RequiredArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
public class Person {
    @NonNull
    String name;
    @EqualsAndHashCode.Exclude Integer age = 10;
}
```

- ‚RequiredArgsConstructor‘ fügt einen Konstruktor für alle ‚NonNull‘ annotierten Felder hinzu.
- ‚EqualsAndHashCode‘ für eine alternative Implementierung der ‚equals‘ Methode hinzu.

# V. OOP Software Engineering

## Lombok:

```
@RequiredArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
public class Person {
    @NonNull
    String name;
    @EqualsAndHashCode.Exclude Integer age = 10;
}
```

```
public class PersonVanillaJava {
    public String name;
    public transient Integer age=0;

    public PersonVanillaJava(String name)
    {
        this.name = name;
    }
}
```

```
public static void main(String[] args) {
    Person p1 = new Person("Hugo");
    Person p2 = new Person("Hugo");
    PersonVanillaJava p1_ = new PersonVanillaJava("Hugo");
    PersonVanillaJava p2_ = new PersonVanillaJava("Hugo");
    System.out.println("Lombok: "+p1.hashCode()+" <-> "+p2.hashCode());
    System.out.println("Vanilla Java: "+p1_.hashCode()+" <-> "+p2_.hashCode());
}
```

# V. OOP Software Engineering

## Lombok:

```
public static void main(String[] args) {  
    Person p1 = new Person("Hugo");  
    Person p2 = new Person("Hugo");  
    PersonVanillaJava p1_ = new PersonVanillaJava("Hugo");  
    PersonVanillaJava p2_ = new PersonVanillaJava("Hugo");  
    System.out.println("Lombok: "+p1.hashCode()+" <-> "+p2.hashCode());  
    System.out.println("Vanilla Java: "+p1_.hashCode()+" <-> "+p2_.hashCode());  
}
```

Lombok: 2260752 <-> 2260752

Vanilla Java: 21685669 <-> 2133927002

1. Ausführung ☹️ ? 😊

Lombok: 2260752 <-> 2260752

Vanilla Java: 21685669 <-> 2133927002

2. Ausführung ☹️ ? 😊

Ist dies gut oder schlecht?

Lombok: 2260752 <-> 2260752

Vanilla Java: 21685669 <-> 2133927002

3. Ausführung ☹️ ? 😊

# V. OOP Software Engineering

## Lombok:

Lombok: 2260752 <-> 2260752

Vanilla Java: 21685669 <-> 2133927002

1. Ausführung ☹️ ? 😊

Hier finden sich zwei verschiedene Ziele wieder:

**Lombok:** Hashcode soll die Identität anhand des Daten-Inhalts feststellen. ➡ Datenmanagement

**Java:** Hashcode soll die Identität anhand der Objektinstanz (Speicheradresse) feststellen. ➡ Objektmanagement

# V. OOP Software Engineering

## Hibernate:

- Framework für objektrelationales Mapping (ORM) und Persistierung in Datenbanken.
- Derzeit 17 Jahre alt (2001), aktuelle Version: 5.4.0
- Anstatt mit Tabellen wird hier mit Objekten gearbeitet, welche relationale Abhängigkeiten analog zu relationalen Datenbanken haben.
- Bringt eigene Abfragesprache ‚Hibernate Query Language‘ HQL mit.
- Definiert den Begriff ‚Entity‘ für POJOs welche persistiert werden können.

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-core</artifactId>  
  <version>5.4.0.Final</version>  
</dependency>
```

# V. OOP Software Engineering

## Hibernate:

```
@RequiredArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
@Entity
public class Person {
    @NotNull
    String name;
    @EqualsAndHashCode.Exclude Integer age = 10;
}
```



# V. OOP Software Engineering

Nächste Vorlesung:

Hibernate

+

***Anti-Patterns***

```
HashMap<String, Object> map = new HashMap<String, Object>() {{  
    put("test", new Object());  
    put("test2", new Object());  
}};
```