

Java Projekte bauen mit Maven

Eine kurze Einführung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Folien: <https://delors.github.io/prog-adv-java-projects/folien.de.rst.html>
<https://delors.github.io/prog-adv-java-projects/folien.de.rst.html.pdf>

Fehler melden: <https://github.com/Delors/delors.github.io/issues>

1. Java Projekte bauen

Ziele

0. **Management der Projektabhängigkeiten** (z. B. JUnit, Log4j, Hibernate, Spring,)
1. **Kompilieren** des Quellcodes
2. **Testen** des Quellcodes
3. **Paketieren** des Quellcodes (.jar oder .war Datei erzeugen inkl. *aller* Abhängigkeiten)
4. **Dokumentation** erstellen
5. **Reports** erstellen (z.B. Testabdeckung, Code-Qualität)
6. ... und vieles mehr, dass dann aber häufig Projektabhängig ist.

Achtung!

Ein wichtiges Meta-Ziel ist es, das Bauen der Software zu automatisieren und zu vereinfachen und *stabile Builds zu gewährleisten*.

D. h. zwei Entwickler, die das selbe Projekt auf unterschiedlichen Rechnern mit initial ggf. unterschiedlichen Versionen installierter Werkzeuge und Bibliotheken bauen, sollten dennoch das selbe Ergebnis erhalten.

Etablierte Build-Tools

- Ant^[1]
- **Maven**
- Gradle
- sbt
- make (nicht spezifisch für Java)

Warnung

IDEs wie IntelliJ IDEA, Eclipse, Visual Studio Code oder NetBeans bieten ebenfalls „Build-Unterstützung“. Diese ist aber bestenfalls für kleine Ein-Entwickler-Projekte geeignet.

^[1] Wurde in der Anfangsphase häufig verwendet. Heute nicht mehr.

Projektstruktur

Konvention, die praktisch über alle Build-Tools und IDEs hinweg gilt^[2]:

- Quellcode im Verzeichnis `src/main/java`
 - Testcode im Verzeichnis `src/test/java`
 - Ressourcen im Verzeichnis `src/main/resources`
 - Testressourcen im Verzeichnis `src/test/resources`
 - Konfigurationen und andere Ressourcen im Verzeichnis `src/main/resources`
 - gebaute Artefakte im Verzeichnis `target`
-

^[2] Andere Sprachen verwenden häufig ähnliche Strukturen. (Selbstverständlich, wird `java` dann durch den Namen der entsprechenden Sprache ersetzt.)

2. Maven

<https://maven.apache.org>

Aufsetzen eines Projekts mittels *Scaffolding*

Maven ermöglicht es, den Rumpf für ein Java-Projekt mit einer einfachen Befehlszeile zu erstellen:

```
mvn archetype:generate \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DarchetypeVersion=1.5 \
  -DinteractiveMode=false
```

Dies erzeugt eine initiale Build-Konfiguration für ein einfaches Java-Projekt und erzeugt die Projektstruktur.

[3]

Die `groupId` folgt dabei den selben Konventionen wie Java-Packages. Die `artifactId` ist der Name des Projekts.

[3] Es gibt eine Vielzahl von Archetypen, die unterschiedliche Projektstrukturen erzeugen und für unterschiedliche Anwendungsfälle optimiert sind.

Maven - Build Phasen

■ Eine Phase ist ein Schritt im Build-Lebenszyklus. Die *ersten* Phasen des Standardlebenszyklus sind:

1. `validate`
2. `generate-sources`
3. `process-sources`
4. `generate-resources`
5. `process-resources`
6. `compile`

■ Wenn eine Phase angegeben wird, dann werden alle vorherigen Phasen ausgeführt. Zum Beispiel führt `mvn compile` alle genannten Phasen in obiger Reihenfolge aus.

die wichtigsten Phasen des Standardlebenszyklus

validate:	überprüfen, ob das Projekt korrekt konfiguriert ist
compile:	kompilieren des Quellcodes des Projekts
test:	testet den kompilierten Quellcode mit einem geeigneten Unit-Testing-Framework.
package:	den kompilierten Code in ein verteilbares Format, z. B. ein JAR, verpacken.
integration-test:	Verarbeitet das Paket und stellt es, wenn nötig, in einer Umgebung bereit, in der Integrationstests ausgeführt werden können.
deploy:	bereitstellen in einer Integrations- oder Release-Umgebung

Spezialisierte Lebenszyklen (mit eigenen Phasen)

clean:	bereinigt Artefakte, die von früheren Builds erzeugt wurden. Phasen: <code>pre-clean</code> , <code>clean</code> , <code>post-clean</code>
site:	generiert eine Site-Dokumentation für dieses Projekt Phasen: <code>pre-site</code> , <code>site</code> , <code>post-site</code> , <code>site-deploy</code>

Beispiel Build-Konfiguration für ein Java Projekt

Code der Anwendung

```
1 package de.dhbw;
2
3 /**
4  * Implements the main method to greet a user by name.
5  */
6 public class HelloYou {
7
8     public static void main(String[] args) {
9         if (args.length == 0) {
10             System.out.println("Usage: java HelloYou <name>");
11             return;
12         }
13         System.out.println("Hello " + args[0] + "!");
14     }
15 }
```

TestCode (Testing System.out)

Header

```
10 public class HelloYouTest {
11
12     // Let's redirect System.out to capture the output!
13     private final PrintStream defaultOut = System.out;
14     private final ByteArrayOutputStream testOut = new ByteArrayOutputStream();
15 }
```

Setup

```
15 @BeforeEach
16 public void setOutputStream() {
17     final var out = new PrintStream(testOut);
18     System.setOut(out);
19 }
20
21 @AfterEach
22 public void resetSystemOut() {
23     System.setOut(defaultOut);
24 }
```

Eigentliche Tests

```
28 @Test
29 void testMainNoArgs() {
30     HelloYou.main(new String[0]);
31     assertEquals("Usage: java HelloYou <name>\n", testOut.toString());
32 }
33
34 @Test
35 void testMainArg() {
36     HelloYou.main(new String[] { "Bob" });
37     assertEquals("Hello Bob!\n", testOut.toString());
38 }
39 }
```

Benötigte Imports

```
1 package de.dhbw;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4 import org.junit.jupiter.api.Test;
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.AfterEach;
```

```
7 import java.io.ByteArrayOutputStream;
8 import java.io.PrintStream;
```

Maven - Build-Konfiguration

Header der Konfigurationsdatei

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
6         <modelVersion>4.0.0</modelVersion>
```

Allg. Metainformationen

```
8 <groupId>de.dhbw</groupId>
9 <artifactId>hello</artifactId>
10 <version>1.0</version>
11 <name>HelloYou</name>
12 <url>http://www.dhbw.de</url>
```

Buildumgebung

```
14 <properties>
15     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16     <maven.compiler.source>23</maven.compiler.source>
17     <maven.compiler.target>23</maven.compiler.target>
18 </properties>
```

Abhängigkeiten

```
20 <dependencies>
21     <dependency>
22         <groupId>org.junit.jupiter</groupId>
23         <artifactId>junit-jupiter</artifactId>
24         <version>5.12.0</version>
25         <scope>test</scope>
26     </dependency>
27 </dependencies>
```

Konfiguration des Builds

```
29 <build>
30     <plugins>
31
32         <plugin>
33             <groupId>org.jacoco</groupId>
34             <artifactId>jacoco-maven-plugin</artifactId>
35             <version>0.8.12</version>
36             <executions>
37                 <execution><goals><goal>prepare-agent</goal></goals></execution>
38                 <execution>
39                     <id>report</id>
40                     <phase>test</phase>
41                     <goals><goal>report</goal></goals>
42                 </execution>
43             </executions>
44         </plugin>
45
46         <plugin>
47             <artifactId>maven-surefire-plugin</artifactId>
48             <version>3.5.2</version>
49         </plugin>
50
51         <plugin>
52             <artifactId>maven-jar-plugin</artifactId>
53             <version>3.0.2</version>
54             <configuration>
```

```
67     <archive>
68       <manifest>
69         <addClasspath>true</addClasspath>
70         <mainClass>de.dhbw.HelloYou</mainClass>
71       </manifest>
72     </archive>
73   </configuration>
74 </plugin>
75 </plugins>
76 </build>
```

Projekt bauen und ausführen

Projekt bauen

`mvn package`

Projekt ausführen

Die gebauten Artefakte befinden sich im Verzeichnis target.

`java -jar target/hello-1.0.jar <Name>`

Übung

2.1. Build-Konfiguration eines Java Projekts

- entpacken Sie das Projekt prog-adv-java-projects/code/newton-code.zip.html.pdf.
- legen Sie eine `pom.xml` Datei an, um das Projekt zu bauen.
- Konfigurieren Sie eine Abhängigkeit zu JUnit 5.12 und konfigurieren Sie das `surefire` Plugin, um die Tests auszuführen.
- Nutzen Sie `mvn test`, um die Tests auszuführen.
- Konfigurieren Sie das `maven-jar-plugin`, um ein ausführbares JAR zu erzeugen. Vergessen sie nicht die `mainClass` zu konfigurieren.
- Nutzen Sie `mvn package`, um das Projekt zu bauen.
- Nutzen Sie `mvn site`, um eine Dokumentation des Projekts zu erstellen.
- Schauen Sie sich die erzeugten Artefakte an.
- Testen Sie ob Sie die Anwendung mit `java -jar target/newton-1.0-SNAPSHOT.jar` starten können.

Weiterführende Aufgaben

(In diesem Fall ist es Ihrer Aufgabe zu recherchieren wie die Einbindung/Konfiguration zu erfolgen hat.)

- Binden Sie Checkstyle in Ihre Projekt ein. D. h. wenn Sie die `mvn site` ausführen, dann soll automatisch ein Report in Hinblick auf die Einhaltung der Checkstyle-Regeln erstellt werden.
Schauen Sie sich den Report an und versuchen Sie für die Klasse `Liste` eine besser Einhaltung der Checkstyle Regeln zu erreichen.
- Binden Sie das Maven-Plugin JaCoCo ein, dass automatisch die Testabdeckung berechnet und in einem Report darstellt. Führen Sie danach `mvn test` aus (und ggf. `mvn site`) und schauen Sie sich den Report an.
Wie hoch ist bereits die Testabdeckung für die Klasse `List` obwohl diese gar nicht explizit getestet wurde?
- Schreiben Sie sinnvolle Tests für die Klasse `List` und erhöhen Sie die Anweisungsüberdeckung auf 100% - abgesehen von den Zeilen, die nur Exceptions werfen. D. h. Sie brauchen sich in den Tests nicht um den Code kümmern, der Exceptions wirft; ignorieren Sie diesen Aspekt für den Moment.
- Binden Sie ein Maven-Plugin ein, dass automatisch die JavaDoc erstellt und in einem Report darstellt.