Java Projekte bauen mit Maven

Eine kurze Einführung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0.5.1

Kontrollfragen: kontrollfragen.de.md.html
Folien: https://delors.github.io/prog-adv-java-projects/folien.de.md.html
https://delors.github.io/prog-adv-java-projects/folien.de.md.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues
KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

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. Grundlagen des Testens mit JUnit (Jupiter)

JUnit - Einführung

- JUnit ist das Standard-Test-Framework für Java.
- Für das Schreiben von einfachen Tests ist das Modul *JUnit Jupiter* zu verwenden.
- Tests liegen (per Konvention) in `src/test/java` und werden *nicht* mit der Anwendung ausgeliefert.

Aufbau einer Testklasse

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest { // Konvention: <Klasse>Test

    @Test void testAddition() {
        Calculator r = new Calculator();
        assertEquals(5, r.addiere(2, 3));
    }

    @Test void testDivisionDurchNull() {
        Calculator r = new Calculator();
        assertThrows(
            ArithmeticException.class,
            () -> r.dividiere(1, 0)
        );
}   }
```

> ▲ Achtung!
>
> Testklassen, die mit dem Namen `Test` enden, werden von Buildtools automatisch erkannt. Es handelt sich um eine fest etablierte und weit verbreitete Konvention.

- Jede mit `@Test` annotierte Methode ist ein Testfall.
- Testmethoden müssen `void` zurückgeben und dürfen keine Parameter haben.
- Es wird für **jeden** Testfall ein **neues** Objekt der Testklasse erzeugt; Tests sind daher voneinander isoliert.

Die wichtigsten Assertions

Alle Methoden befinden sich in `org.junit.jupiter.api.Assertions`:

Methode	Prüft
`assertEquals(expected, actual)`	`expected.equals(actual)`
`assertNotEquals(a, b)`	`!a.equals(b)`
`assertTrue(condition)`	`condition == true`
`assertFalse(condition)`	`condition == false`
`assertNull(object)`	`object == null`
`assertNotNull(object)`	`object ≠ null`
`assertThrows(Exc.class, () → …)`	Lambda wirft erwartete Exception
`assertArrayEquals(exp, act)`	Arrays elementweise gleich

Hinweis

Jede Assertion akzeptiert als *letzten* Parameter eine optionale Fehlermeldung (`String`), die bei einem Fehlschlag ausgegeben wird:

```
assertEquals(5, r.addiere(2, 3), "2+3 sollte 5 sein");
```

Setup und Teardown

```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class ListTest {

    private List list;

    @BeforeEach   // vor JEDEM einzelnen Test
    void setUp() {          list = new List();                          }

    @AfterEach    // nach JEDEM einzelnen Test
    void tearDown() {       /* z.B. Ressourcen freigeben */             }

    @Test
    void testIsEmpty() {    assertTrue(list.isEmpty());                 }

    @Test
    void testAddElement() { list.add(42); assertEquals(1, list.size()); }
}
```

`@BeforeEach` und `@AfterEach` werden vor bzw. nach *jedem* Testfall ausgeführt und eignen sich ideal, um einen definierten Ausgangszustand herzustellen.

Daneben gibt es `@BeforeAll` und `@AfterAll` (müssen `static` sein), die nur *einmal* pro Testklasse aufgerufen werden.

Typische Vorgehensweise beim Testen

Arrange – Act – Assert (AAA-Muster)

1. **Arrange** – Testdaten und Objekte vorbereiten.
2. **Act** – Die zu testende Methode aufrufen.
3. **Assert** – Das Ergebnis mit dem erwarteten Wert vergleichen.

```java
@Test
void testAbsoluterBetrag() {
    /* Arrange:*/ int eingabe = -7;
    /* Act:    */ int ergebnis = Math.abs(eingabe);
    /* Assert: */ assertEquals(7, ergebnis);
}
```

Hinweis

Ein Test sollte genau **einen** Aspekt prüfen. Besser mehrere kleine Tests als ein großer.

3. 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.

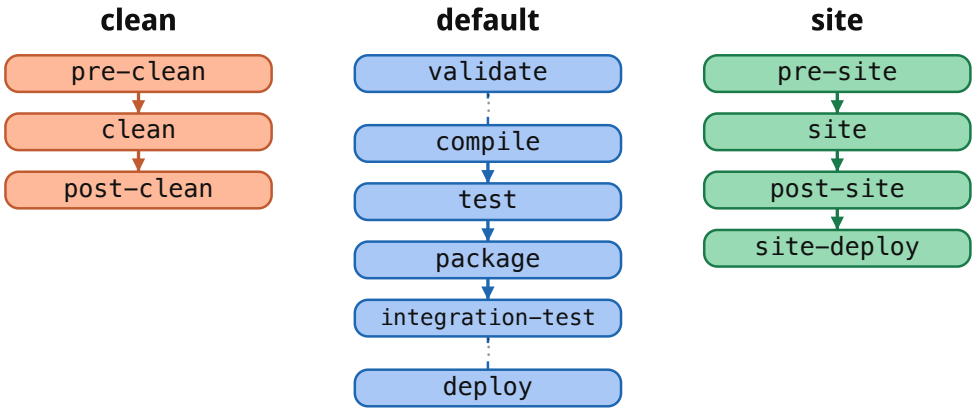
die wichtigsten Phasen des Standardlebenszyklus

`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: `pre-clean`, `clean`, `post-clean`
`site`:	generiert eine Site-Dokumentation für dieses Projekt Phasen: `pre-site`, `site`, `post-site`, `site-deploy`

Maven Lebenszyklen und Phasen - Übersicht



Beispiel Build-Konfiguration für ein Java Projekt

Code der Anwendung (in `src/main/java/<package>/<class>.java`)

```java
package de.dhbw;

/**
 * Implements the main method to greet a user by name.
 */
public class HelloYou {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java HelloYou <name>");
            return;
        }
        System.out.println("Hello " + args[0] + "!");
}   }
```

TestCode (in `src/test/java/<package>/<class>.java`)

(Herausforderung: Testing System.out)

Header

```java

public class HelloYouTest {

    // Let's redirect System.out to capture the output!
    private final PrintStream defaultOut = System.out;
    private final ByteArrayOutputStream testOut = new ByteArrayOutputStream();

```

Setup

```java

    @BeforeEach
    public void setOutputStream() {
        final var out = new PrintStream(testOut);
        System.setOut(out);
    }

    @AfterEach
    public void resetSystemOut() {
        System.setOut(defaultOut);
    }

```

Eigentliche Tests

```java

    @Test
    void testMainNoArgs() {
        HelloYou.main(new String[0]);
        assertEquals("Usage: java HelloYou <name>\n", testOut.toString());
    }

    @Test
    void testMainArg() {
        HelloYou.main(new String[] { "Bob" });
        assertEquals("Hello Bob!\n", testOut.toString());
```

```
    }
}
```

Benötigte Imports

```
package de.dhbw;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

```

Maven - Build-Konfiguration (`pom.xml` im Root Verzeichnis des Projekts)

Header der 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
>
  <modelVersion>4.0.0</modelVersion>

```

Allg. projektspezifische Metainformationen *(Achtung: Anpassung erforderlich!)*

```

  <groupId>de.dhbw</groupId>
  <artifactId>hello</artifactId>
  <version>1.0</version>
  <name>HelloYou</name>
  <url>http://www.dhbw.de</url>

```

Buildumgebung

```

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>25</maven.compiler.source>
    <maven.compiler.target>25</maven.compiler.target>
  </properties>

```

Abhängigkeiten

```

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.14.3</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

```

Konfiguration des Builds

```

  <build>
    <plugins>
```

```
      <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.8.14</version>
        <executions>
          <execution><goals><goal>prepare-agent</goal></goals></execution>
          <execution>
            <id>report</id>
            <phase>test</phase>
            <goals><goal>report</goal></goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.5.5</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.5.0</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <mainClass>de.dhbw.HelloYou</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

Projekt bauen und ausführen

Projekt bauen (inkl. `compile` und `test`)

```
mvn package
```

Projekt ausführen

Die gebauten Artefakte befinden sich im Verzeichnis target.

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

✎ Übung

3.1. Build-Konfiguration eines Java Projekts

Basisaufgaben

(Falls Maven (`mvn`) noch nicht installiert ist, installieren Sie es.)

- entpacken Sie das Projekt newton-code.zip.
- legen Sie eine `pom.xml` Datei an, um das Projekt zu bauen.
- Konfigurieren Sie eine Abhängigkeit zu JUnit 5.14.3 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. Stellen Sie sicher, dass Sie JUnit 5.14.3 nutzen; die Verwendung von KI Tools ist - außer für die Tests - gestattet.)

- 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.