

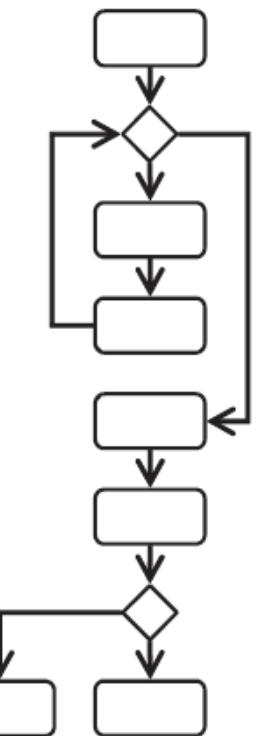
Lektion 14

Unit Tests
Klassenbibliotheken
JUnit 5
Maven

Unit Tests

White-Box-Test

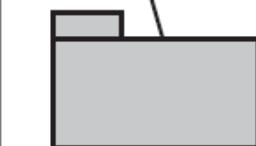
Anwei-sungen
Entschei-dungen
Pfade



Gray-Box-Test

Paket/
Komponente

Schnittstellen

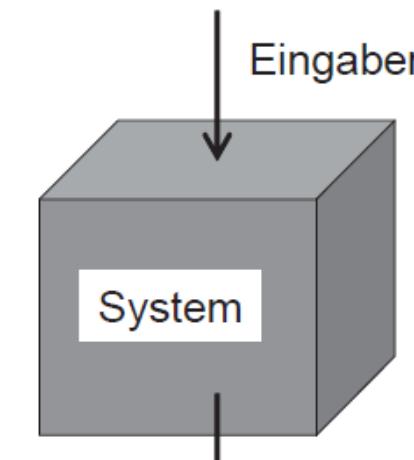


Black-Box-Test

Eingaben

System

Ausgaben



Methoden-/Klassentest

Integrationstest

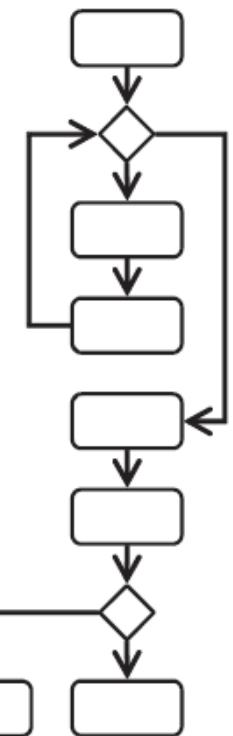
Systemtest

Stephan Kleuker: Qualitätssicherung durch Softwaretests, Springer Vieweg, S. 29,
2013, ISBN: 978-3834809292

© Prof. Dr. Steffen Heinzl

White-Box-Test

Anwei-sungen
Entschei-dungen
Pfade



Methoden-/Klassentest

Gray-Box-Test

Paket/
Komponente

Schnittstellen



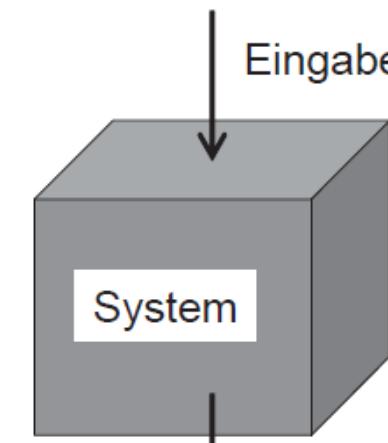
Integrationstest

Black-Box-Test

Eingaben

System

Ausgaben



Systemtest

Stephan Kleuker: Qualitätssicherung durch Softwaretests, Springer Vieweg, S. 29,
2013, ISBN: 978-3834809292

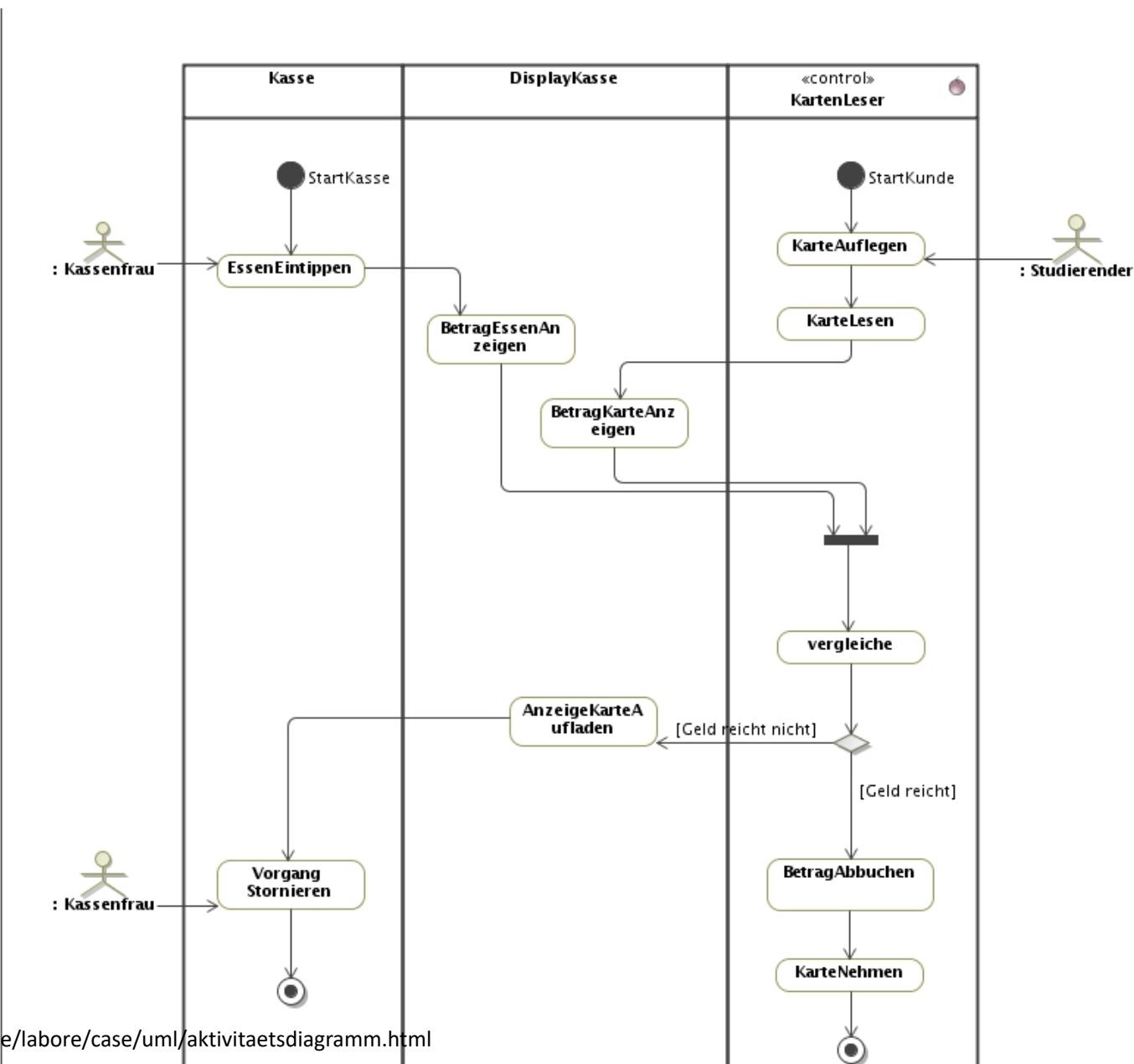
© Prof. Dr. Steffen Heinzl

Unit Tests gehören zu den White-Box-Tests.

Ein Test überprüft die Funktionalität

- einer Komponente,
- eines Code-Abschnitts (z.B. Pfad),
- einer Methode.

Ziel der Tests sollte es sein, eine möglichst große Abdeckung des Programmcodes zu erzielen,
d.h. alle Anweisungen und Pfade in einem Programm sollen mindestens einmal getestet werden.



source: <https://www.fbi.h-da.de/labore/case/uml/aktivitaetsdiagramm.html>

© Prof. Dr. Steffen Heinzl

Wie können wir Tests in Java schreiben?

Wir erweitern unser Programm um weitere Funktionalitäten durch das Einbinden “fremder” Klassen.

Eine Sammlung von Klassen (Klassenbibliothek) wird in einem sogenannten Java Archive (jar) zusammengefasst.

Programme können ihre Funktionalität erweitern, indem sie jar-Files auf den Klassenpfad legen.

JUnit 5

```
package functions;

public class Sign
{
    public static int sign(double x)
    {
        if (x < 0) return -1;
        else if (x > 0) return +1;
        else return 0;
    }
}
```

Wir wollen nebenstehende Klasse testen!

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

/** Klasse zum Testen der Vorzeichenfunktion */
public class SignTest
{
    @Test
    public void testSignOperator()
    {
        assertTrue(-1 == Sign.sign(-5));
        assertTrue(1 == Sign.sign(5));
        assertTrue(0 == Sign.sign(0));
    }
}
```

assertTrue lässt den Test weiterlaufen, wenn die in den Klammern angegebene Bedingung wahr ist.

Ansonsten wird der Test abgebrochen und schlägt fehl.

Die statischen Methoden der Klasse Assertions werden hier eingebunden. Bspw. kann anstelle von `Assertions.assertTrue` einfach `assertTrue` geschrieben werden.

Bei dieser Methode handelt es sich um eine Test-Methode. Beim Starten dieser Klasse mit den Junit-jars auf dem Klassenpfad werden alle mit `@Test` annotierten Methoden ausgeführt.

Hier findet der eigentliche Test statt:
Der Test gelingt,
wenn Sign von -5 die Zahl -1 zurückgibt,
wenn Sign von 5 die Zahl 1 zurückgibt,
wenn Sign von 0 die Zahl 0 zurückgibt.

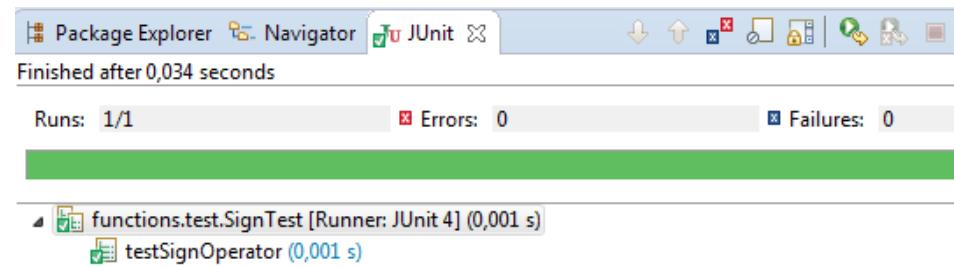
```

import functions.Sign;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

/** Klasse zum Testen der Vorzeichenfunktion */
public class SignTest
{
    @Test
    public void testSignOperator()
    {
        assertTrue(-1 == Sign.sign(-5));
        assertTrue(1 == Sign.sign(5));
        assertTrue(0 == Sign.sign(0));
    }
}

```



Wenn bei einem Fehlschlag klar sein soll, welche Bedingung fehlgeschlagen ist, schreiben wir lieber einzelne Tests.

```

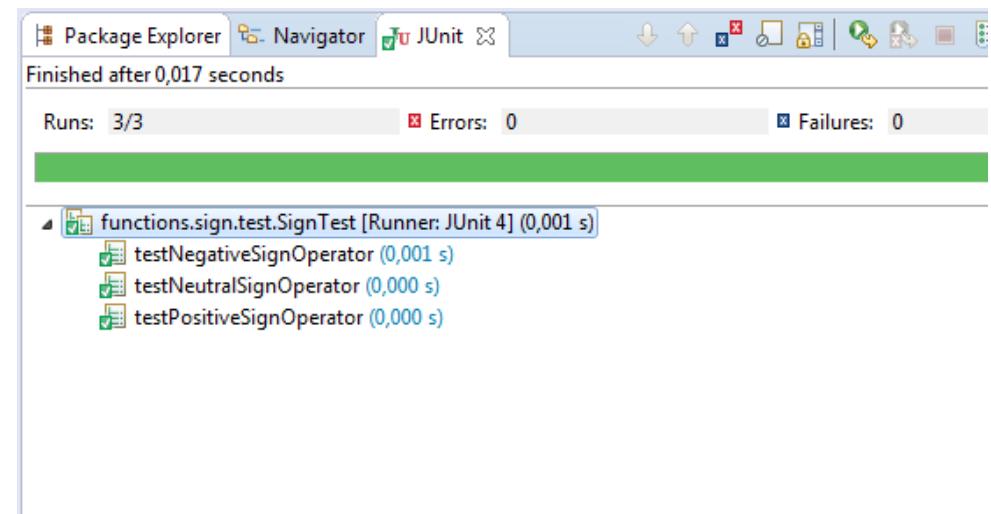
public class SignTest
{
    @Test
    public void testPositiveSignOperator()
    {
        assertTrue(1 == Sign.sign(5));
    }

    @Test
    public void testNegativeSignOperator()
    {
        assertTrue(-1 == Sign.sign(-5));
    }

    @Test
    public void testNeutralSignOperator()
    {
        assertTrue(0 == Sign.sign(0));
    }
}

```

Beim Ausführen der Klasse werden drei Tests ausgeführt.

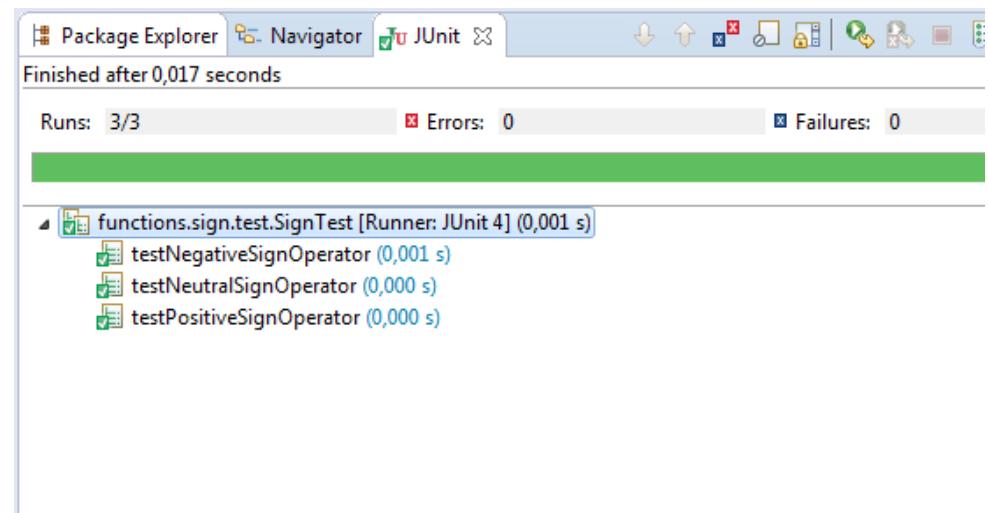


```
public class SignTest
{
    @Test
    public void testPositiveSignOperator()
    {
        assertTrue(1 == Sign.sign(5));
    }

    @Test
    public void testNegativeSignOperator()
    {
        assertTrue(-1 == Sign.sign(-5));
    }

    @Test
    public void testNeutralSignOperator()
    {
        assertTrue(0 == Sign.sign(0));
    }
}
```

Beim Ausführen der Klasse werden drei Tests ausgeführt.



Alle Test-Methoden, die eine Unit testen, werden Unit Test (Test Case) genannt.

Wie geht man mit Exceptions um?

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

public class Oberflaeche
{
    public static int berechneQuaderOberflaeche(int a, int b, int c)
    {
        if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültiges Argument");
        return 2*a*b+2*a*c+2*b*c;
    }

    @Test
    public void testFehlerfall()
    {
        try
        {
            berechneQuaderOberflaeche(1, 1, -1);
        }
        catch(RuntimeException e)
        {
        }
    }
}
```

Wir übergeben bewusst fehlerhafte Argumente,
um den Fehlerfall auszulösen!

```

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

public class Oberflaeche
{
    public static int berechneQuaderOberflaeche(int a, int b, int c)
    {
        if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültiges Argument");
        return 2*a*b+2*a*c+2*b*c;
    }

    @Test
    public void testFehlerfall()
    {
        try
        {
            berechneQuaderOberflaeche(1, 1, -1);
            fail("Runtime Exception erwartet");
        }
        catch(RuntimeException e)
        {

        }
    }
}

```

Wir übergeben bewusst fehlerhafte Argumente,
um den Fehlerfall auszulösen!

fail lässt generell einen Test fehlschlagen.

An dieser Stelle stellt fail sicher, dass der Test fehlschlägt,
wenn berechneOberflaeche fehlerfrei durchläuft.

```

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

public class Oberflaeche
{
    public static int berechneQuaderOberflaeche(int a, int b, int c)
    {
        if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültiges Argument");
        return 2*a*b+2*a*c+2*b*c;
    }

    @Test
    public void testFehlerfall()
    {
        try
        {
            berechneQuaderOberflaeche(1, 1, -1);
            fail("Runtime Exception erwartet");
        }
        catch(RuntimeException e)
        {
            String errorMessage = e.getMessage();
            assertEquals("Ungültiges Argument", errorMessage);
        }
    }
}

```

Wir stellen sicher,
dass die Fehlermeldung der Exception stimmt.

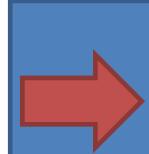
```

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

public class Oberflaeche
{
    public static int berechneQuaderOberflaeche(int a, int b, int c)
    {
        if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültiges Argument");
        return 2*a*b+2*a*c+2*b*c;
    }

    @Test
    public void testFehlerfall()
    {
        try
        {
            berechneQuaderOberflaeche(1, 1, -1);
            fail("Runtime Exception erwartet");
        }
        catch(RuntimeException e)
        {
            String errorMessage = e.getMessage();
            assertEquals("Ungültiges Argument", errorMessage);
        }
    }
}

```



Der Test schlägt fehl, wenn **keine** RuntimeException mit der Fehlermeldung **Ungültiges Argument** auftritt.

Zusammenfassend kann man sagen...

Es kann sehr viele Tests geben.

Tests laufen durch oder schlagen fehl.

Es sollte klar sein, welches assert den Fehler ausgelöst hat.

Test und Production Code sollten kurz hintereinander geschrieben werden, damit man keine Tests vergisst.

Diese Punkte kann man sich gut durch das Akronym F.I.R.S.T.
(siehe nächste Folie)
merken!

F.I.R.S.T.

- **Fast.** Tests should run quickly, so they can be run frequently to detect bugs early.
- **Independent.** Tests should not depend on each other and be runnable in any order.
- **Repeatable.** Tests should be repeatable in any environment (production, test, ...).
- **Self-Validating.** Tests should either pass or fail. There should be no manual evaluation like reading log files, comparing files, ...
- **Timely.** Unit tests should be written directly before production code. If you write production code first, you may decide that some code is too hard to test and never write the tests.

JUnit 5 Konfiguration mit Maven in Eclipse

Maven (Build Management)

- “tool [...] for building and managing any Java-based project”
- Wir verwenden Maven für das Dependency-Management.

Maven (Build Management)

- File -> New Maven Project
- Catalog -> Configure
- Add Remote Catalog

- Catalog File:
 - <https://repo.maven.apache.org/maven2/archetype-catalog.xml>
- Description:
 - Nach Belieben ausfüllen

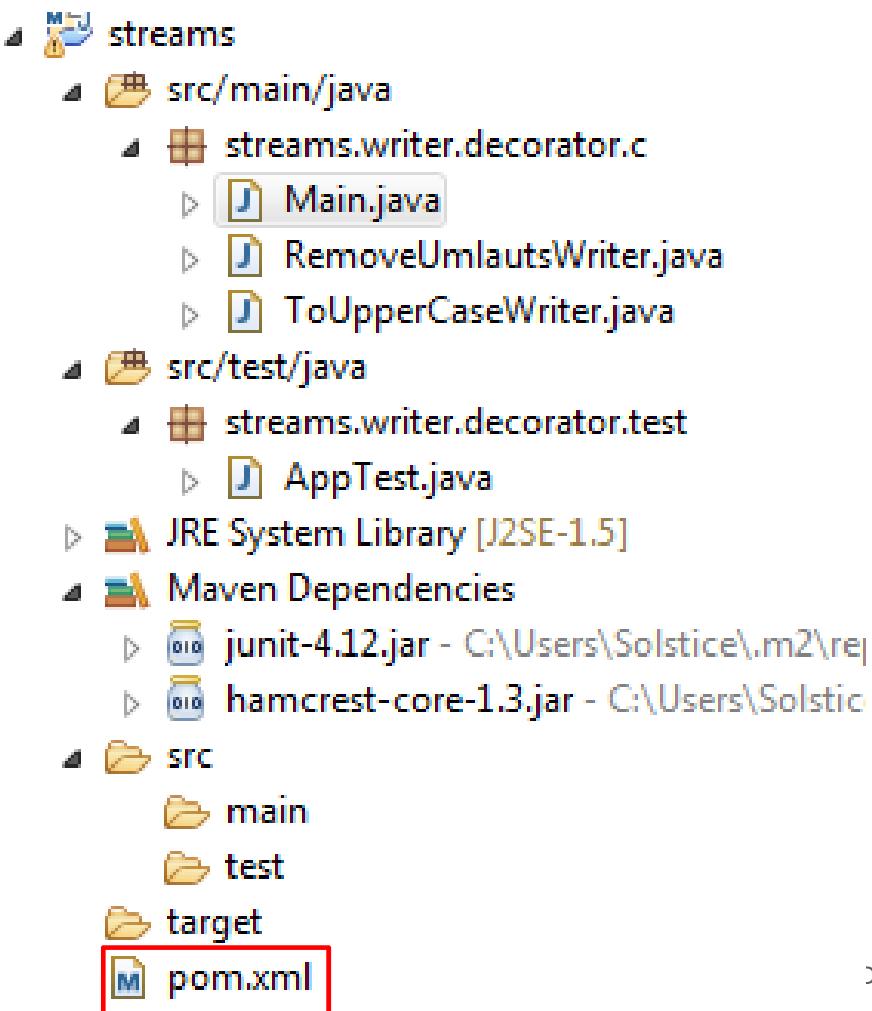
- Java 8 Archetype **pl.org.miki** suchen

- Apply & Close

Maven (Build Management)

in Eclipse:

- New Maven project
- Click Next
- Select `pl.org.miki`
- Click Next
- groupId: `de.fhws`
- artifactId: `streams`
- Finish

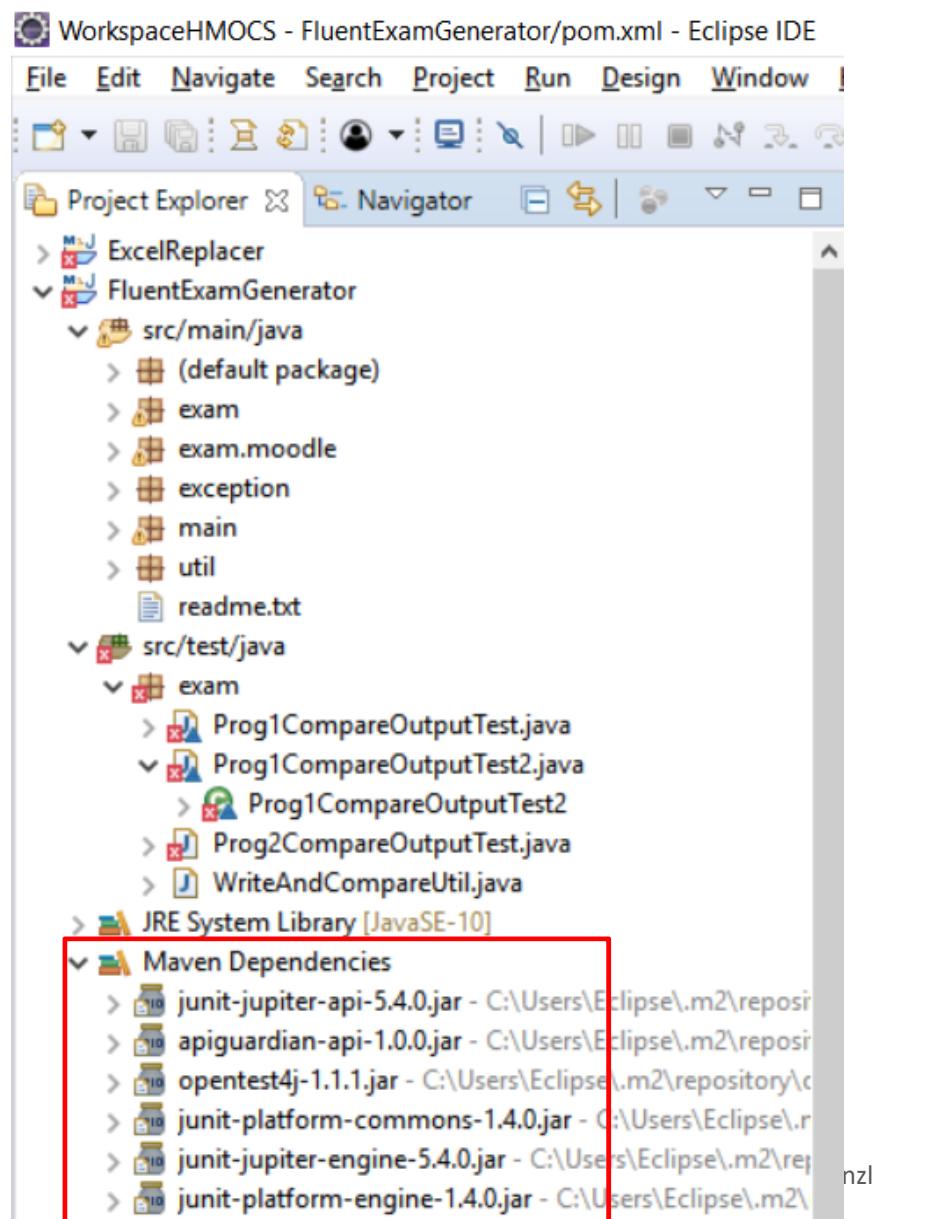


Maven (Build Management)

- Ergänzen/ersetzen Sie folgenden Eintrag in der pom.xml, falls nicht vorhanden:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.4.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.4.0</version>
    <scope>test</scope>
</dependency>
```

- Eclipse lädt automatisch die benötigten Jars herunter und bindet Sie in das Projekt ein.



```
<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>de.fhws</groupId>
  <artifactId>streams</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>streams</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

pom.xml