Lektion 14

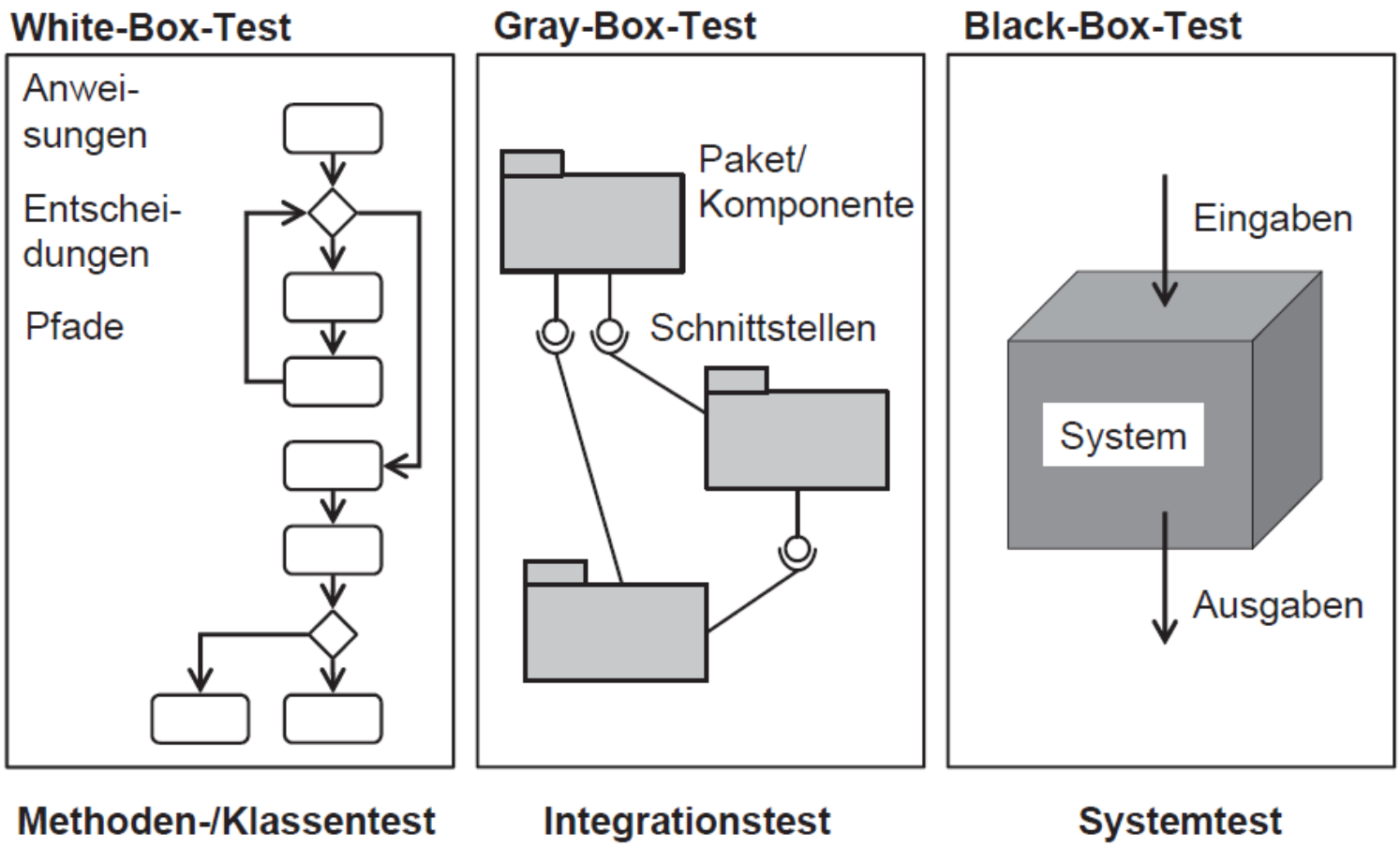
Unit Tests

Klassenbibliotheken

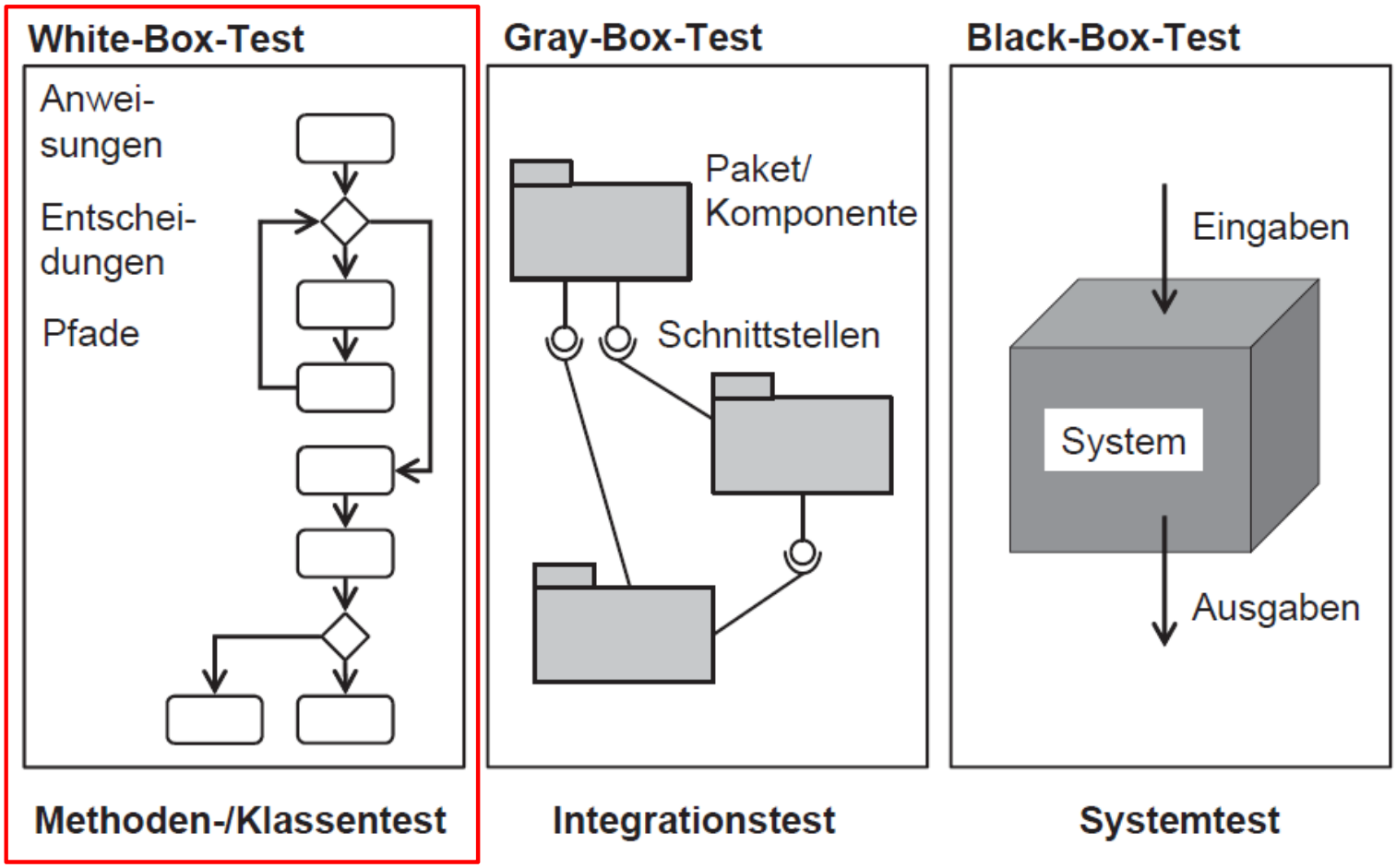
JUnit 5

Maven

Unit Tests



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



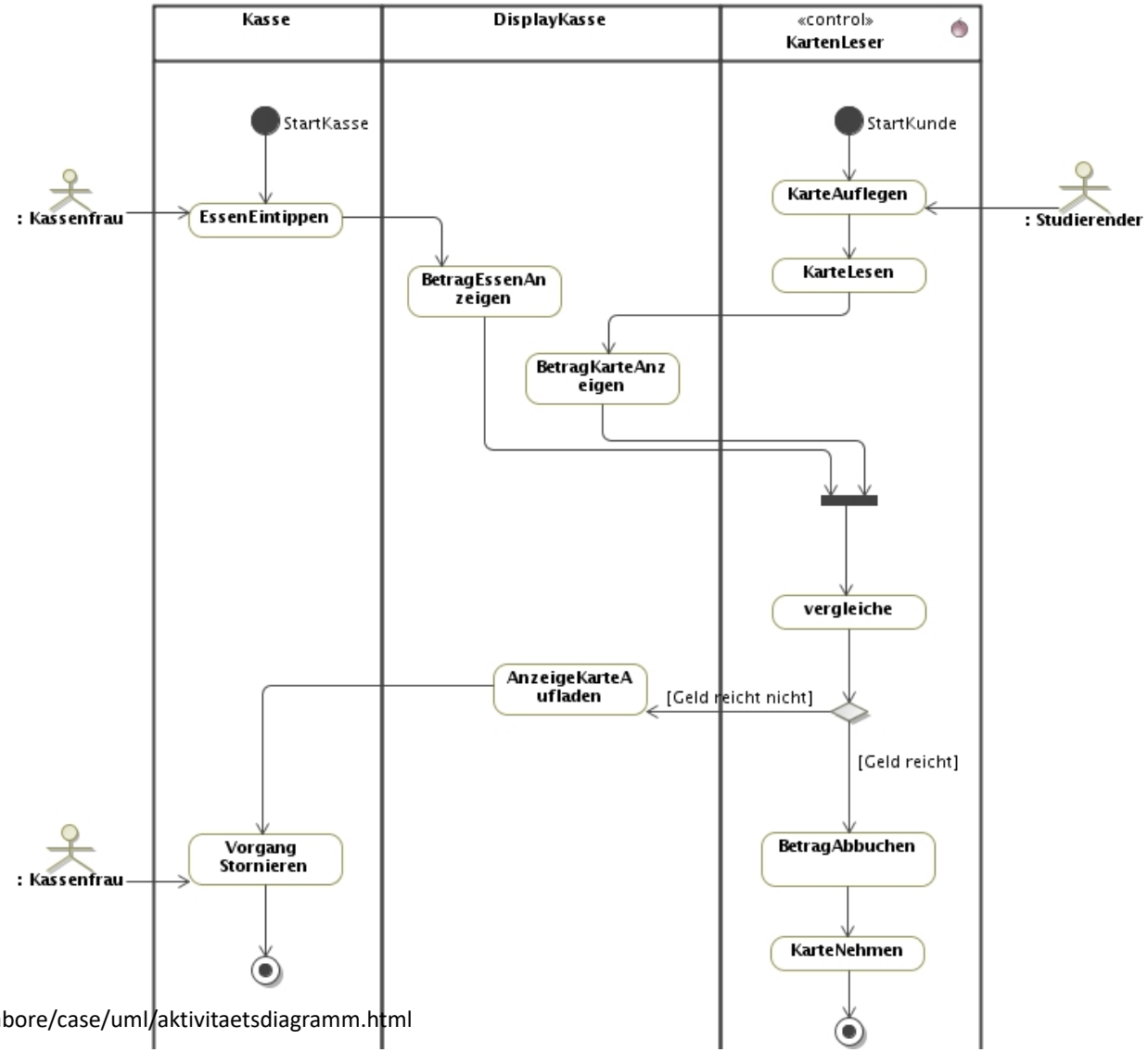
Stephan Kleuker: Qualitätssicherung durch Softwaretests, Springer Vieweg, S. 29, 2013, ISBN: 978-3834809292

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

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));
  }
}
```

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.

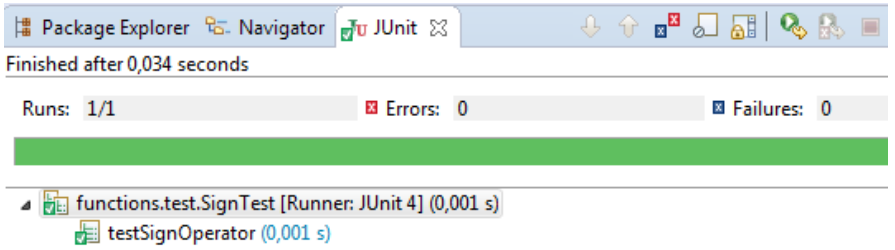
assertTrue lässt den Test weiterlaufen, wenn die in den Klammern angegebene Bedingung wahr ist.

Ansonsten wird der Test abgebrochen und schlägt fehl.

```java
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));
  }
}
```



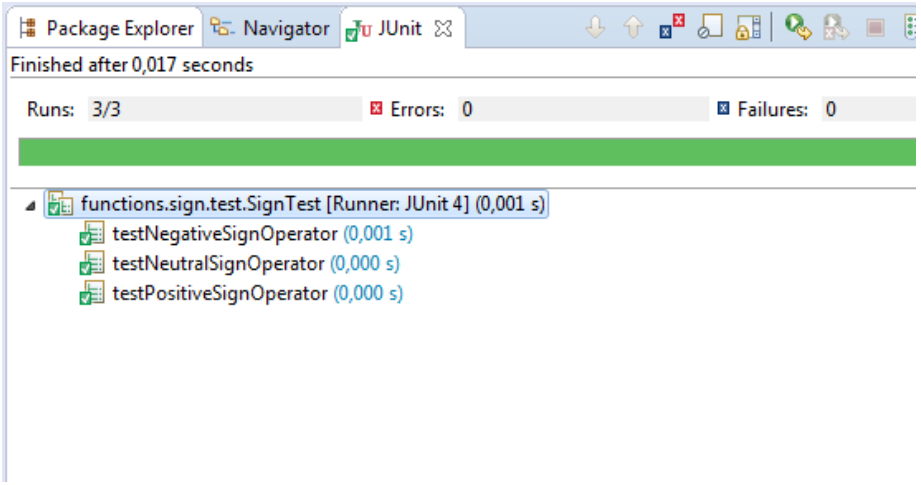
Wenn bei einem Fehlschlag klar sein soll, welche Bedingung fehlgeschlagen ist, schreiben wir lieber einzelne Tests.

```
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.

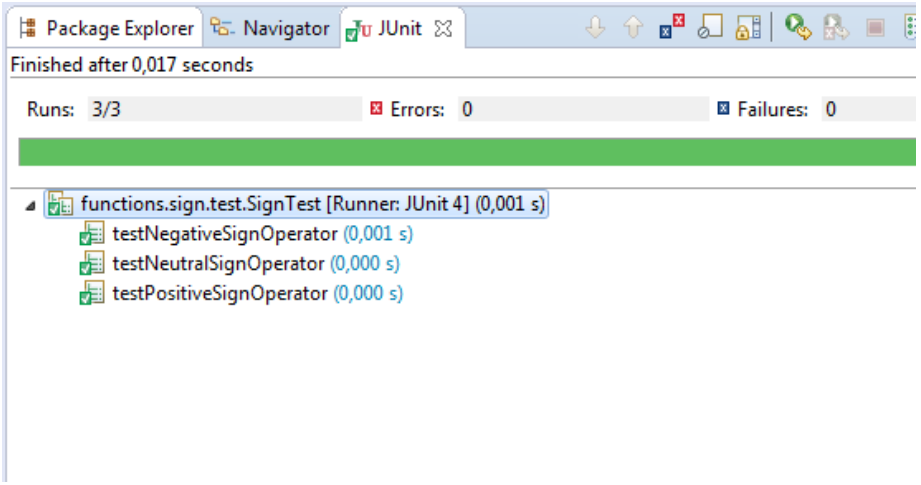


```
public class SignTest
{
  @Test
  public void testPositiveSignOperator()
  {
    assertTrue(1 == Sign.sign(5));
  }

  @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?

```java
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!

```java
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.

```java
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.

```java
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(errorMessage, "Ungültiges Argument");
    }
```

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)

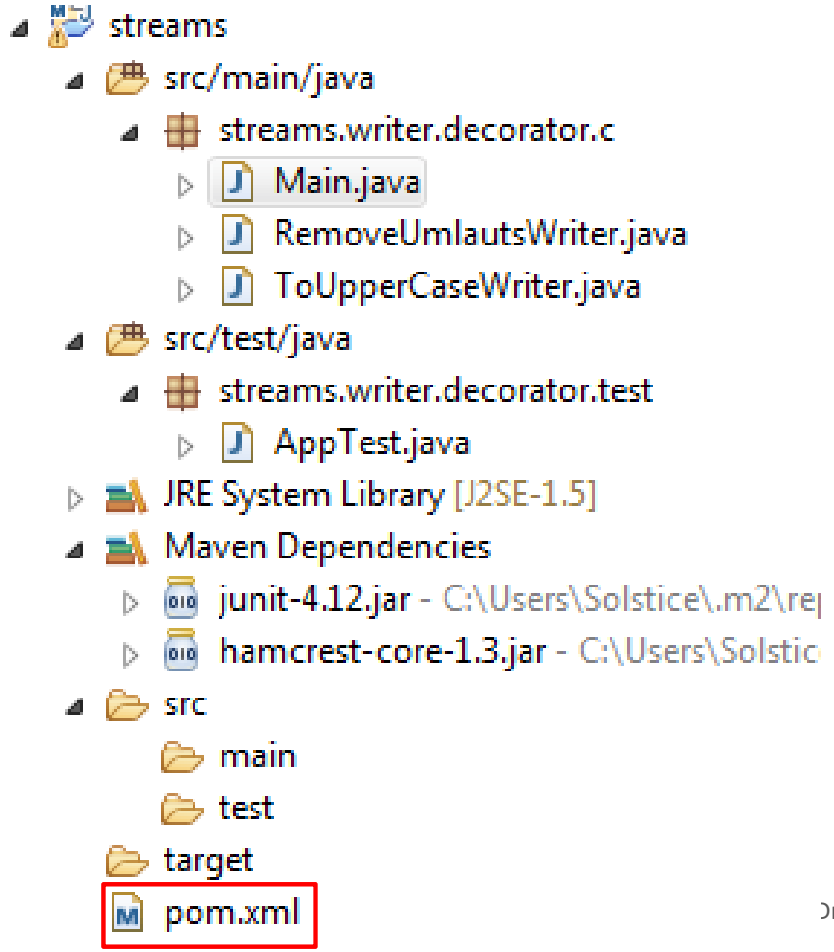
in Eclipse:

- New Maven project
- Click Next

- Select **pl.org.miki**
- Click Next

- groupId: **de.fhws**
- artifactId: **streams**

- Finish

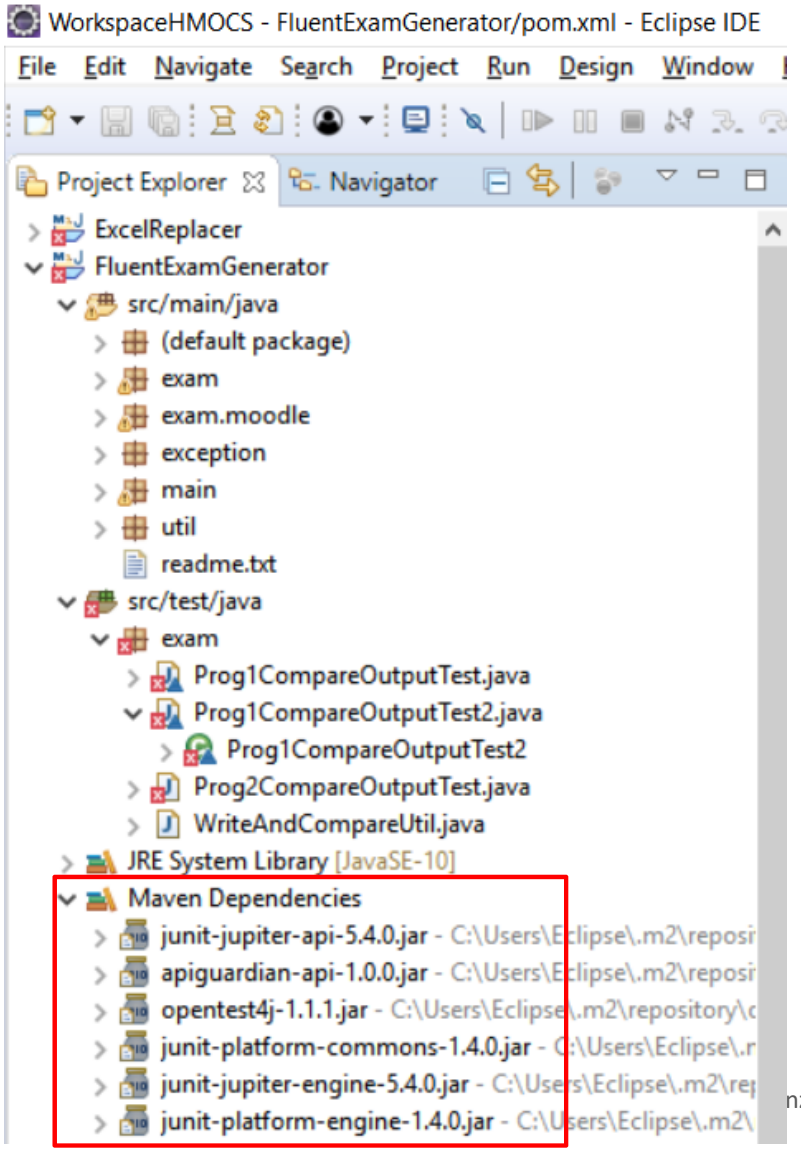


Maven (Build Management)

- 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.


WorkspaceHMOCS - FluentExamGenerator/pom.xml - Eclipse IDE
File Edit Navigate Search Project Run Design Window
Project Explorer
Navigator
ExcelReplacer
FluentExamGenerator
src/main/java
(default package)
exam
exam.moodle
exception
main
util
readme.txt
src/test/java
exam
Prog1CompareOutputTest.java
Prog1CompareOutputTest2.java
Prog1CompareOutputTest2
Prog2CompareOutputTest.java
WriteAndCompareUtil.java
JRE System Library [JavaSE-10]
Maven Dependencies
junit-jupiter-api-5.4.0.jar - C:\Users\Eclipse\.m2\reposi
apiguardian-api-1.0.0.jar - C:\Users\Eclipse\.m2\reposi
opentest4j-1.1.1.jar - C:\Users\Eclipse\.m2\repository\
junit-platform-commons-1.4.0.jar - C:\Users\Eclipse\.r
junit-jupiter-engine-5.4.0.jar - C:\Users\Eclipse\.m2\re
junit-platform-engine-1.4.0.jar - C:\Users\Eclipse\.m2\

```xml
<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>5.4.0</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>
  </dependencies>
</project>
```

pom.xml

Alt

JUnit 4

Unit Tests mit JUnit

- Download Sie die folgenden Jars:
- https://github.com/junit-team/junit4/wiki/Download-and-Install
  - junit.jar
  - hamcrest-core.jar
- und legen Sie sie auf den Klassenpfad.

- In Eclipse:
  - jar-files in Eclipseprojekt kopieren
  - Rechtsklick auf jar-File -> Build Path -> Add to Build Path

Alternative zum manuellen Download

Maven

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)

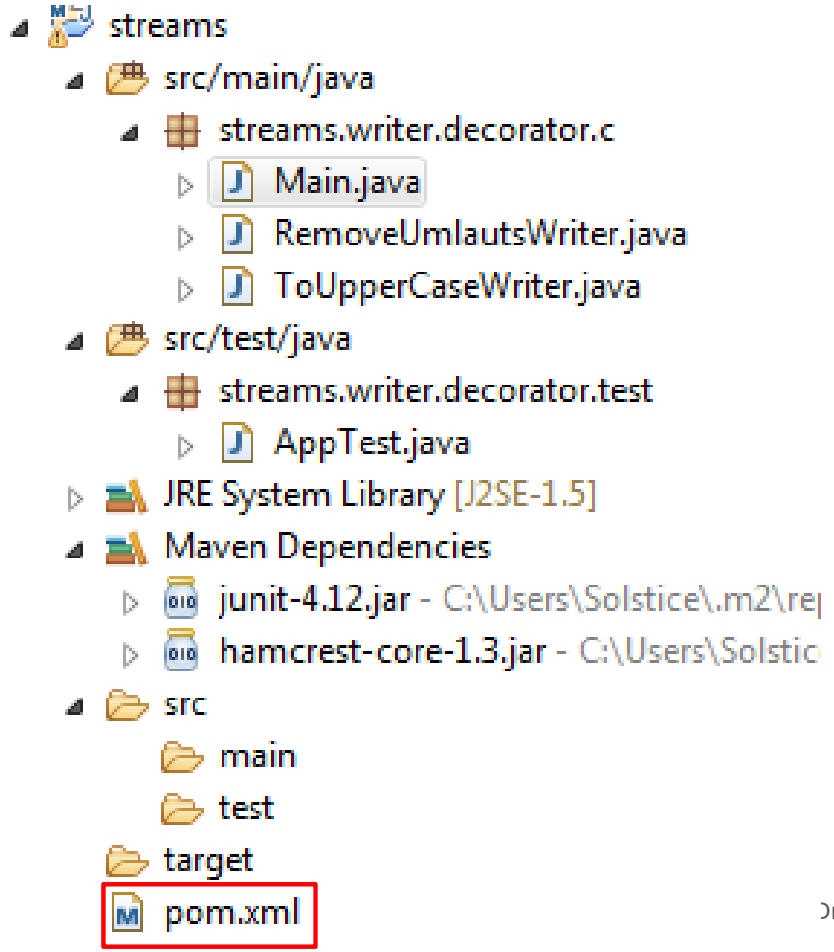
in Eclipse:

- 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>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</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>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

pom.xml

```
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.Assert.*;

import org.junit.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));
  }
}
```

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

Bei dieser Methode handelt es sich um eine Test-Methode. Beim Starten dieser Klasse mit dem JUnit.jar 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.

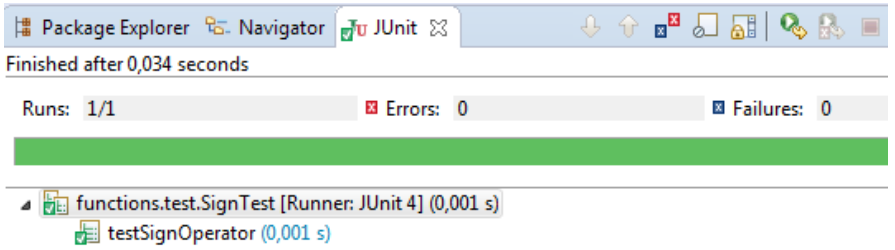
assertTrue lässt den Test weiterlaufen, wenn die in den Klammern angegebene Bedingung wahr ist.

Ansonsten wird der Test abgebrochen und schlägt fehl.

```java
import functions.Sign;
import static org.junit.Assert.*;

import org.junit.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));
  }
}
```



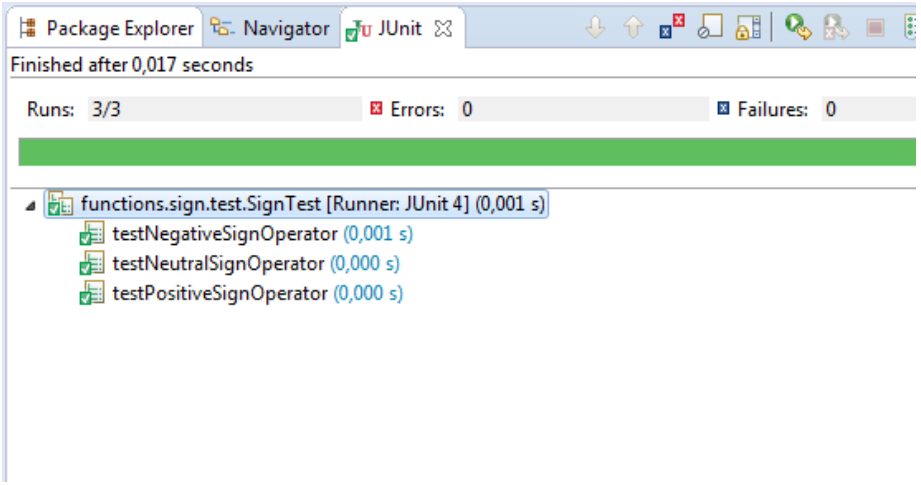
Wenn bei einem Fehlschlag klar sein soll, welche Bedingung fehlgeschlagen ist, schreiben wir lieber einzelne Tests.

```
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.



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

Wie geht man mit Exceptions um?

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class Oberflaeche
{
  public static String berechneOberflaeche(int a, int b, int c)
  {
    if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültige Parameter");
    return "Die Oberfläche des Quaders beträgt: " + (2*a*b+2*a*c+2*b*c);
  }

  @Test
  public void testFehlerfall()
  {
    try
    {
      berechneOberflaeche(1, 1, -1);
    }
    catch(RuntimeException e)
    {
      if (!e.getMessage().equals("Ungültige Parameter")) fail("Test fehlgeschlagen");
    }
  }
}
```

Wir übergeben bewusst fehlerhafte Argumente, um den Fehlerfall auszulösen!

fail lässt den Test fehlschlagen!

Der Test schlägt fehl, wenn **keine** RuntimeException mit der Fehlermeldung `Ungültiger Parameter` auftritt.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class Oberflaeche
{
  public static String berechneOberflaeche(int a, int b, int c)
  {
    if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültige Parameter");
    return "Die Oberfläche des Quaders beträgt: " + (2*a*b+2*a*c+2*b*c);
  }

  @Test
  public void testFehlerfall()
  {
    try
    {
      berechneOberflaeche(1, 1, -1);
      fail();
    }
    catch(RuntimeException e)
    {
      if (!e.getMessage().equals("Ungültige Parameter")) fail("Test fehlgeschlagen");
    }
  }
}
```

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

Der Test schlägt fehl, wenn **keine** RuntimeException mit der Fehlermeldung `Ungültiger Parameter` auftritt.