UNIKASSEL
VERSITÄT

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Software testen

Jens Kosiol

Wintersemester 23/24

(teilweise angelehnt an Folien von Prof. Dr. Gabriele Taentzer)

Überblick

- Einführung
  - Warum testen wir?
  - Was ist ein Test?
  - Grundlegende Syntax (JUnit)
- Wie testen wir?
  - JUnit als Testframework
- Wieviel testen wir?
  - Coverage-Kriterien
- Wann testen wir?
  - Test-driven development (TDD) und das test-first Prinzip

Passengers stranded as Covid passport crash hits NHS app

Technical problem stopped people from loading passes that show proof of vaccination and are required for international travel

By **Mike Wright**
13 October 2021 • 4:43pm

Related Topics
Vaccines, NHS, Apps, Vaccination, Coronavirus

Latest UK deaths: 108 -32%
See figures for your area

Start your free trial today
Subscribe now

Passengers have been left stranded at airports after the NHS app went down, leaving them without access to vaccine passports.

Travellers reported being barred from flights and stuck abroad after the app crashed for around four hours on Wednesday.

The outage meant people were unable to load their Covid pass, proving they have had their vaccine shots, which is needed for travel abroad.

Softwarefehler in Ladesäulen

Andere Marken tanken gratis an Teslas Superchargern

Von t-online, jnm

13.09.2020
Lesedauer: 2 Min.

Ein Tesla an einem Supercharger in der Schweiz: Die neuesten Ladesäulen betankten versehentlich auch andere Autos – und zwar gratis (Quelle: Geisser/imago-images-bilder)

Ein Softwarefehler in den Schnellladesäulen von Tesla führt offenbar dazu, dass E-Autos anderer Marken dort aktuell kostenlos laden können. Die Ladebuchsen sind erst seit Kurzem mit anderen Herstellern kompatibel.

Definition Test

Ein Test ist die Ausführung eines Programms mit künstlichen Daten und der Abgleich des beobachteten mit dem erwarteten Verhalten, um Fehler oder Anomalien aufzudecken, erwünschtes Verhalten zu bestätigen oder Informationen über nicht-funktionale Eigenschaften des Programms zu gewinnen.

[Nach: Ian Sommerville, Software Engineering, 9. Aufl., Pearson 2012, S. 246]

Testziele

Erwünschtes Verhalten dokumentieren (Validierungstests):

- Zeigen, dass das Programm sich (in den getesteten Situationen) verhält wie gewünscht
- Überprüfen, dass Programmänderungen (Erweiterungen, Refactoring, …) existierende Funktionalität nicht beschädigt haben (Regressionstests)
- Umfassende Tests auch von Basisverhalten
- Benutzen der User Stories für Testszenarien

Fehlerhaftes Verhalten aufdecken (Fehlertests):

- Tests fehleranfälliger, komplizierter Methoden
- Tests mit außergewöhnlichen Eingabewerten

„Program testing can be used to show the presence of bugs, but never to show their absence!"

[Edsger Dijkstra in: Notes On Structured Programming (1970)]

Grundlegende Teststrategien

- Black-Box-Test: Das interne Verhalten und die interne Struktur des getesteten Systems sind nicht bekannt.
  - Es ist bekannt, was ein Softwaresystem leisten soll, aber nicht, wie es das tut.
  - Testfälle ergeben sich oft aus Anforderungsbeschreibung
- White-Box-Test: Das zu testende System ist vollständig bekannt.
  - Kenntnis des Codes fließt ins Testdesign mit ein
- Gray-Box-Test: Interna des zu testenden Systems sind teilweise bekannt

Testarten

- Unit test (dt.: Modul- oder Komponententest): Testet eine eigenständige (kleine) Einheit (Modul, Klasse, Interface, Methode)
  - Whitebox-Test
  - Häufig vom Entwickler selbst durchgeführt
  - Es sollten keine Abhängigkeiten zu anderen Komponenten bestehen (Verwendung von Stubs und Mock-Objekten)
- Integration test (dt.: Integrationstest): Testet, dass das Zusammenspiel verschiedener Komponenten funktioniert
  - Kann Gray-Box-Test sein: Schnittstellen von Modulen bekannt, aber nicht deren Code
- Manchmal auch noch feinere Unterteilungen (Funktionstest, Systemtest, Abnahmetest, …)

In dieser Vorlesung: Einführung in die Entwicklung von unit tests

Testdurchführung

Es gibt verschiedene Möglichkeiten, Tests durchzuführen.

- Manuelle Tests: direkte Benutzung der Software
- Testausgaben (print line): direkt in Code und Ausgabe
- Debugging: schrittweises Ausführen eines Programms unter Beobachtung des internen Zustands
- Automatisiertes Testen: Implementierung von Testfällen in separaten Testklassen, die automatisiert ablaufen können

Anforderungen an (Unit-) Tests

- Anforderungsbezogen
- Reproduzierbar
- Überprüfbar
- Mit der Fehlersuche koppelbar
- Schnell
- Unabhängig
- Gründlich
- …

Ein primitiver Test

```
public class Adder {

   static public int add(int a, int b) {
      return a+b;
   }



   public static void main(String[] args) {
      AdderTest.addingZeroIsLeftNeutral();
   }
}

public class AdderTest {
   public static void addingZeroIsLeftNeutral() {
      boolean testResult = (5 == Adder.add(0, 5));
      System.out.println(
         "Adding with 0 is left neutral:" +
         testResult);
   }
}
```

- Wie rufe ich diesen Test sinnvoll auf und führe ihn automatisiert wiederholt durch?
- Wie stelle ich sicher, dass der Test nicht mit dem Produktivcode kompiliert wird?

Testframeworks bieten:

- Hilfreiche Syntax für Tests
- Separation von Test- und Produktivumgebung
- Möglichkeit der automatisierten Ausführung
- Fehlermeldungen

JUnit

- Testframework für unit tests in Java
- Entwickelt ab 1997; zuerst von Kent Beck und Erich Gamma
- Lässt sich in gängige Buildprozesse integrieren (Ant, Maven, Gradle)
- Wichtige Konzepte:
  - Annotationen: Erklären Code zu Test und bestimmen, welche Rolle welcher Code im Test spielt
  - Assertions: Drücken aus, was getestet wird

Ein erster Test mit JUnit

```
public class Adder {
  public int add(int a, int b) {
    return a+b;
  }
}
```



```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;

class AdderTest {
  private Adder adder = new Adder();

  @Test
  void addingZeroIsLeftNeutral() {
    assertEquals(5, adder.add(0, 5));
  }

  @Test
  void addingZeroIsRightNeutral() {
    assertTrue(adder.add(5, 0) == 5);
  }
}
```

← benötigte JUnit Importe
(benötigte Assertions static importieren)

← Einrichten einer Testsituation

← @Test-Annotation über jedem Test
← Sprechender Name für Testmethode
← Test per passender Assertion

← @Test-Annotation über jedem Test
← Sprechender Name für Testmethode
← Test per passender Assertion

Grundsätzlicher Aufbau eines Unit-Tests

Aufbau nach „arrange, act, assert“ (auch: „given, when, then“):

- Arrange (given): Einrichten der Testsituation
  - Initialisieren der benötigten Objekte
  - Setzen der benötigten Felder und Links
- Act (when): Durchführen des Tests
  - Aufruf der zu testenden Methode mit passenden Parameterwerten
- Assert (then): Überprüfen des Testergebnisses
  - Vergleich zwischen tatsächlichem und erwartetem Verhalten
  - Nachvollziehbare Rückmeldung über das Testergebnis
- (optional Teardown: Abbau der Testsituation)

Beispiel Arrange

```
Room mathRoom = new Room()
  .withTopic("math")
  .withCredits(17);

Student karli = new Student()
  .withName("Karli")
  .withCredits(0)
  .withMotivation(214);
```

Titel: Karli versucht zu graduieren und scheitert

Beschreibung:

1. Ausgangssituation: Karli möchte einen Abschluss an der Study-Right University erwerben. Zur Zeit braucht er dafür 214 Credit Points (CP). Karli startet mit 0 CP und einer Motivation von 214 Punkten. Er befindet sich außerhalb der Universität.
2. Karli betritt den Raum *math*, nimmt an der Vorlesung teil, erhält automatisch 17 CP und verliert 17 Motivationspunkte. Er hat nun 17 CP und 197 Motivationspunkte.

Fluent Interfaces

```
public class Student {
  private String name;
  private int motivation;
  private int credits;
  private Room at;

  public Student withName(String name) {
    this.name = name;
    return this;
  }

  public Student withMotivation(int motivation) {
    this.motivation = motivation;
    return this;
  }
}
```

Setter-Methoden mit `this` als Rückgabewert erlauben verkettete Aufrufe auf Objekt.

Beispiel Act

```
karli.enterRoom(mathRoom);
karli.takeCourse();
```

Titel: Karli versucht zu graduieren und scheitert

Beschreibung:

1. Ausgangssituation: Karli möchte einen Abschluss an der Study-Right University erwerben. Zur Zeit braucht er dafür 214 Credit Points (CP). Karli startet mit 0 CP und einer Motivation von 214 Punkten. Er befindet sich außerhalb der Universität.
2. Karli betritt den Raum *math*, nimmt an der Vorlesung teil, erhält automatisch 17 CP und verliert 17 Motivationspunkte. Er hat nun 17 CP und 197 Motivationspunkte.

Beispiel Assert

```
assertEquals(mathRoom, karli.getAt());
assertTrue(197 == karli.getMotivation());
assertEquals(17, karli.getCredits());
```

Titel: Karli versucht zu graduieren und scheitert

Beschreibung:

1. Ausgangssituation: Karli möchte einen Abschluss an der Study-Right University erwerben. Zur Zeit braucht er dafür 214 Credit Points (CP). Karli startet mit 0 CP und einer Motivation von 214 Punkten. Er befindet sich außerhalb der Universität.
2. Karli betritt den Raum *math*, nimmt an der Vorlesung teil, erhält automatisch 17 CP und verliert 17 Motivationspunkte. Er hat nun 17 CP und 197 Motivationspunkte.

Beispiel vollständiger Test

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

public class StudentTest {
   @Test
   void testTakeCourse() {
      Room mathRoom = new Room().withTopic("math").withCredits(17);
      Student karli = new Student().withName("Karli").withCredits(0).withMotivation(214);

      karli.enterRoom(mathRoom);
      karli.takeCourse();

      assertEquals(mathRoom, karli.getAt());
      assertTrue(197 == karli.getMotivation());
      assertEquals(17, karli.getCredits());
   }
}
```

JUnit Assertions

Alle JUnit Assertions sind statische Methoden in der Klasse `org.junit.jupiter.api.Assertions` (https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html).

`assertAll`	Prüft, dass alle Programme nicht zu Fehlern führen
`assertEquals`	Überprüft auf Gleichheit
`assertInstanceOf`	Überprüft, ob übergebenes Objekt vom erwarteten Typ ist
`assertNull`	Stellt sicher, dass übergebenes Objekt `null` ist
`assertSame`	Überprüft Objekte auf Gleichheit
`assertThrows`	Überprüft, ob Ausführung eine `exception` vom erwarteten Typ wirft
`assertTimeout`	Überprüft, ob Ausführung innerhalb eines vorgegebenen Zeitrahmens geschieht
`assertTrue`	Überprüft, dass ein Boolescher Ausdruck zu `true` auswertet

…

Für viele der Assertions steht auch eine Negation zur Vefügung (`assertNotEquals`, `assertNotNull`, `assertFalse`, …).

Syntax assertAll

```
class AssertionsDemo {
    private final Person person = new Person("Jane", "Doe");

    @Test
    void dependentAssertions() {
        // Within a code block, if an assertion fails the
        // subsequent code in the same block will be skipped.
        assertAll("properties",
            () -> {
                String firstName = person.getFirstName();
                assertNotNull(firstName);

                // Executed only if the previous assertion is valid.
                assertAll("first name",
                    () -> assertTrue(firstName.startsWith("J")),
                    () -> assertTrue(firstName.endsWith("e"))
                );
            },
            () -> {
                // Grouped assertion, so processed independently
                // of results of first name assertions.
                String lastName = person.getLastName();
                assertNotNull(lastName);
            }
        );
    }
}
```

- Die Importe sind der Übersichtlichkeit halber ausgelassen
- `assertAll` kann mit einer Überschrift starten (properties, first name)
- Zentraler Parameter ist eine beliebige Anzahl von `executables`
- Jede der ausführbaren Funktionen wird ausgeführt (statt beim ersten Scheitern den Test als gescheitert abzubrechen)
- Innerhalb einer solchen Funktion kann es zum Abbruch dieser (!) Funktion kommen.

https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions

Fehlermeldung scheiternder Test


```java
public class StudentTest {
   @Test
   void testTakeCourse() {
      …

      assertEquals(mathRoom, karli.getAt());
      assertTrue(198 == karli.getMotivation());
      assertEquals(17, karli.getCredits());
   }
}
```

```
Expected :true
Actual   :false
<Click to see difference>

org.opentest4j.AssertionFailedError Create breakpoint : expected: <true> but was: <false> <6 internal lines>
    at de.pm2324.StudentTest.testTakeCourse(StudentTest.java:24) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <32 internal lines>
    at jdk.proxy1/jdk.proxy1.$Proxy2.stop(Unknown Source) <7 internal lines>
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.run(GradleWorkerMain.java:69)
    at worker.org.gradle.process.internal.worker.GradleWorkerMain.main(GradleWorkerMain.java:74)
```

Fehlermeldung mit assertAll

```java
public class StudentTest {
   @Test
   void testTakeCourseWithAssertAll() {
      …

      assertAll("Run all of the following tests",
         () -> assertEquals(mathRoom, karli.getAt()),
         () -> assertTrue(198 == karli.getMotivation()),
         () -> assertEquals(18, karli.getCredits())
      );
   }
}
```

```
org.opentest4j.MultipleFailuresError: Run all of the following tests (2 failures)
    org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
    org.opentest4j.AssertionFailedError: expected: <18> but was: <17> <3 internal lines>
```

```
Expected :true
Actual   :false
<Click to see difference>

org.opentest4j.AssertionFailedError Create breakpoint : expected: <true> but was: <false> <6 internal lines>
                                                    ll$1(StudentTest.java:45) <11 internal lines>
                                                    entTest.java:43) <31 internal lines>
```

```
Expected :18
Actual   :17
<Click to see difference>

org.opentest4j.AssertionFailedError Create breakpoint : expected: <18> but was: <17> <6 internal lines>
    at de.pm2324.StudentTest.lambda$testTakeCourseWithAssertAll$2(StudentTest.java:46) <11 internal lines>
    at de.pm2324.StudentTest.testTakeCourseWithAssertAll(StudentTest.java:43) <31 internal lines>
```

Optionale Fehlerbenachrichtigung

```
public class StudentTest {
   @Test
   void testTakeCourse() {
      …

      assertEquals(198, karli.getMotivation(),
         "Karli should have left a motivation" +
         "of 198 after taking their first" +
         "math course"
      );
   }
}
```

- Als letzten, optionalen Parameter haben viele Assertions (`assertEquals`, `assertInstanceOf`, `assertTrue`, …) eine Fehlerbenachrichtigung vom Typ `String`.
- Bei scheiterndem Test kann die Ausgabe die Lokalisierung des Fehlers erleichtern.

```
Karli should have left a motivation of 198 after taking their first math course
Expected :198
Actual   :197
<Click to see difference>

org.opentest4j.AssertionFailedError Create breakpoint : Karli should have left a motivation of 198 after taking their first math course ==>
    at de.pm2324.StudentTest.testTakeCourseWithMessage(StudentTest.java:65) <31 internal lines>
```

Methodenarten in JUnit

- JUnit unterscheidet zwischen Testmethoden und Lifecycle-Methoden.
  - Testmethoden sind die Methoden, die die eigentlichen Tests durchführen (annotiert mit @Test).
  - Lifecycle-Methoden sind Methoden zum Einrichten der Testsituation und Managen der benötigten Objekte (Annotationen später).
- Beide Arten von Methoden sind (fast) immer `void` und nie `abstract`.
- Die Sichtbarkeit darf für keine der beiden Methodenarten `private` sein; JUnit empfiehlt, einfach keinen Modifikator für die Sichtbarkeit zu verwenden.

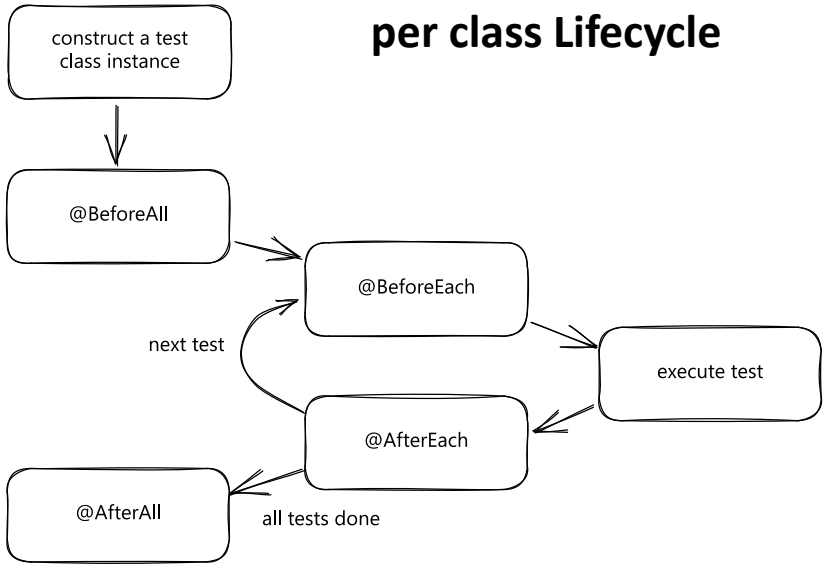
Testausführung in JUnit

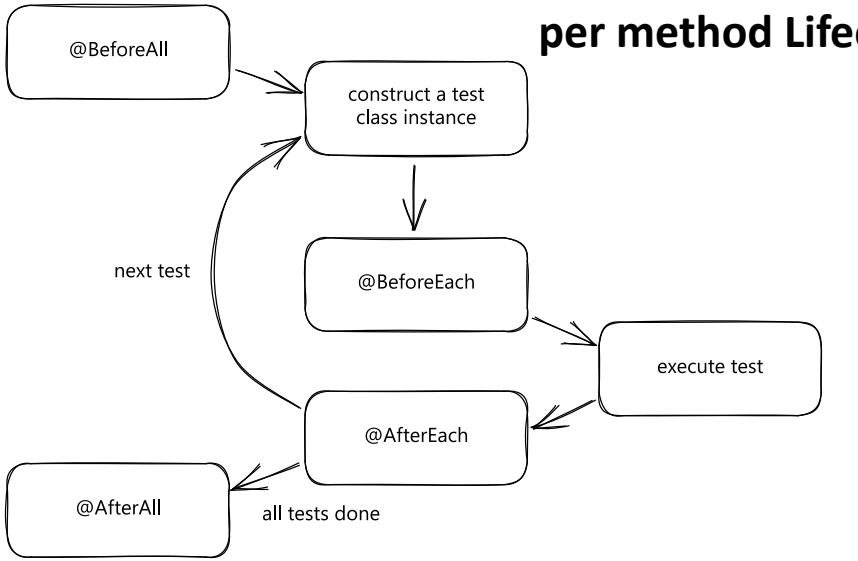
Der Standard in JUnit ist ein per-method Lebenszyklus für Testinstanzen:

- Beim Ausführen einer Testklasse wird jede (nicht deaktivierte) Testmethode der Testklasse ausgeführt.
- Ausführungsverhalten: Vor jeder Ausführung einer Testmethode wird eine neue Instanz der Testklasse geschaffen und die Methode auf dieser Instanz ausgeführt (fördert isolierte Ausführbarkeit).
- Ausführungsreihenfolge der Testmethoden: „deterministic but intentionally non-obvious“ (kann über Annotationen gesteuert werden)
- Der Lebenszyklus kann auf per-class umgestellt werden; dann stehen Annotationen zur Verfügung, um das Einrichten der Testsituationen zu managen (lifecycle-Methoden).

Lifecycle-Methoden

- Eine Testklasse kann mit `@TestInstance(Lifecycle.PER_CLASS)` annotiert werden. → Die Testklasse wird einmal instanziiert und diese Instanz für alle Testmethoden verwendet.
- Methoden in der Testklasse können annotiert werden mit:
  - `@BeforeAll`: Wird *einmal* vor Durchführung aller Testmethoden durchgeführt (mit Lifecycle `per method` nur auf statische Methoden anwendbar)
  - `@BeforeEach`: Wird vor jeder Durchführung einer Testmethode durchgeführt
  - `@AfterEach`: Wird nach jeder Durchführung einer Testmethode durchgeführt
  - `@AfterAll`: Wird einmal nach der Durchführung aller Testmethoden durchgeführt (mit Lifecycle `per method` nur auf statische Methoden anwendbar)





[Bildquelle: https://www.arhohuttunen.com/junit-5-test-lifecycle/]

Falsche Testkonzeption

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

public class StudentTest {
   Student karli = new Student().withName("Karli").withCredits(0).withMotivation(214);
   Room mathRoom = new Room().withTopic("math").withCredits(17);

   @Test
   void testEnterRoom() {
      karli.enterRoom(mathRoom);
      assertEquals(mathRoom, karli.getAt());}

   @Test
   void testTakeCourse() {
      karli.takeCourse();

      assertTrue(197 == karli.getMotivation());
      assertEquals(17, karli.getCredits());}
}
```

JUnit führt Tests nicht zwangsläufig in der gegebenen Reihenfolge durch!

Korrigierte Tests

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

public class StudentTest {
   Student karli = new Student().withName("Karli").withCredits(0).withMotivation(214);
   Room mathRoom = new Room().withTopic("math").withCredits(17);

   @Test
   void testEnterRoom() {
      karli.enterRoom(mathRoom);
      assertEquals(mathRoom, karli.getAt());
   }
   @Test
   void testTakeCourse() {
      karli.withAt(mathRoom);
      karli.takeCourse();
      assertTrue(197 == karli.getMotivation());
   }
}
```

- Die Testmethoden in der Testklasse teilen eine gemeinsame Ausgangssituation.
- Die Testmethoden sind unabhängig voneinander.
- `testTakeCourse` ist nicht mehr von der Korrektheit der Methode `enterRoom` abhängig.

Codeabdeckung

Beim Ausführen von Tests kann man protokollieren, welche Teile des Codes der zu testenden Anwendung ausgeführt wurden.

- Ist Code, der erfolgreich getestet wurde, zwangsläufig korrekt?
- Es gibt verschiedene Abdeckungsmetriken, um zu messen, wie Tests den Code abdecken; sie bieten unterschiedlich viel Sicherheit.
  - Daten- oder Codeabdeckung
  - Verschiedene Formen von Codeabdeckung

Code-Abdeckungsmetriken

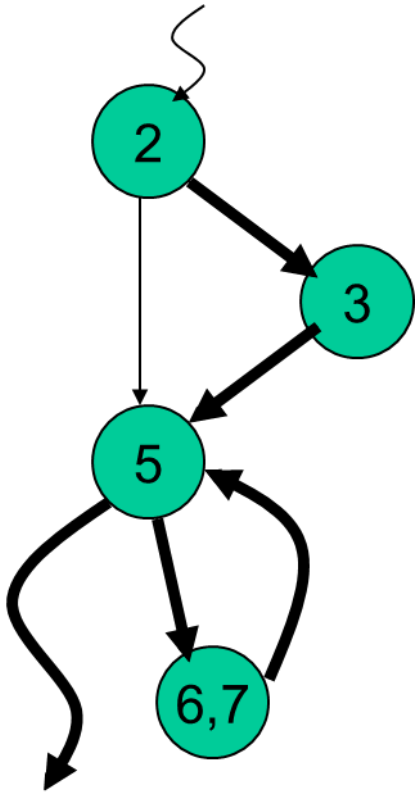
- Function coverage: Wurde jede Methode ausgeführt?
- Statement coverage: Wurde jedes Statement ausgeführt?
- Branch coverage: Wurde, bei verzweigenden Kontrollstrukturen, jede Möglichkeit ausgeführt?
- Condition coverage: War jeder Boolesche Ausdruck bei den Ausführungen einmal „wahr“ und einmal „falsch“?
- Path coverage: Wurde jeder mögliche Pfad durch das Programm getestet?
- …

Als Metriken wird jeweils gemessen, welcher Anteil der Möglichkeiten bei Ausführung der Testsuite abgedeckt wird.

Achtung: Hohe Abdeckung allein sagt nichts über die Qualität einer Testsuite!

Wie viele Testfälle für welches Coverage-Kriterium?

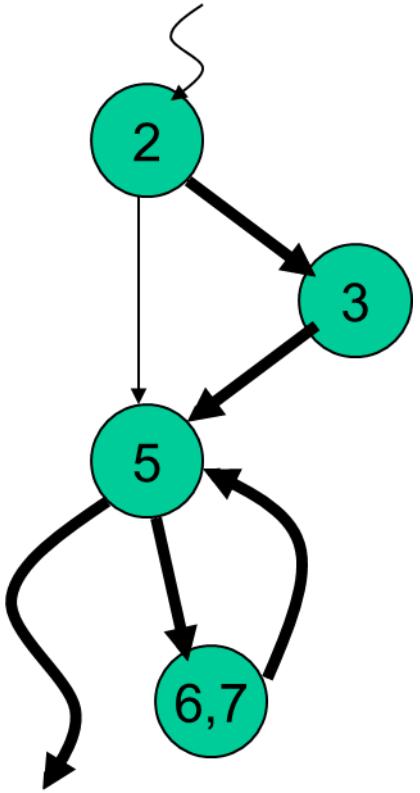
```
1  input(y)
2  if (y <= 0) then
3     y := -y
4  end_if
5  while (y > 0) do
6     input(x)
7     y := y-1
8  end_while
```



Ergebnisse

Vollständige Abdeckung kann für die verschiedenen Kriterien auf folgende Art erreicht werden:

- Function coverage: ein Testfall mit beliebigem Input für y (Funktion wird ausgeführt)
- Statement coverage: ein Testfall mit negativem Input für y
- Branch coverage: zwei Testfälle, einmal mit positivem und einmal mit negativem Input für y
- Condition coverage: Hier wie bei Branch coverage
- Path coverage: kann nicht erreicht werden, da jede ganze Zahl zu ihrem eigenen Pfad führt



Fazit Codeabdeckung

- Function und Statement coverage
  - Leicht zu messen
  - Leicht zu erreichen
  - Schwach
- Path coverage
  - Schwer zu messen
  - Im Allgemeinen nicht erreichbar (Schleifen)
  - Stark
- Branch und Condition coverage
  - Guter Kompromiss
- Alle diese Metriken ignorieren Datenabdeckung!

Datenabdeckung

Datenabdeckung überprüft, ob alle Methoden mit verschiedenen möglichen Werten für ihre Parameter aufgerufen wurden.

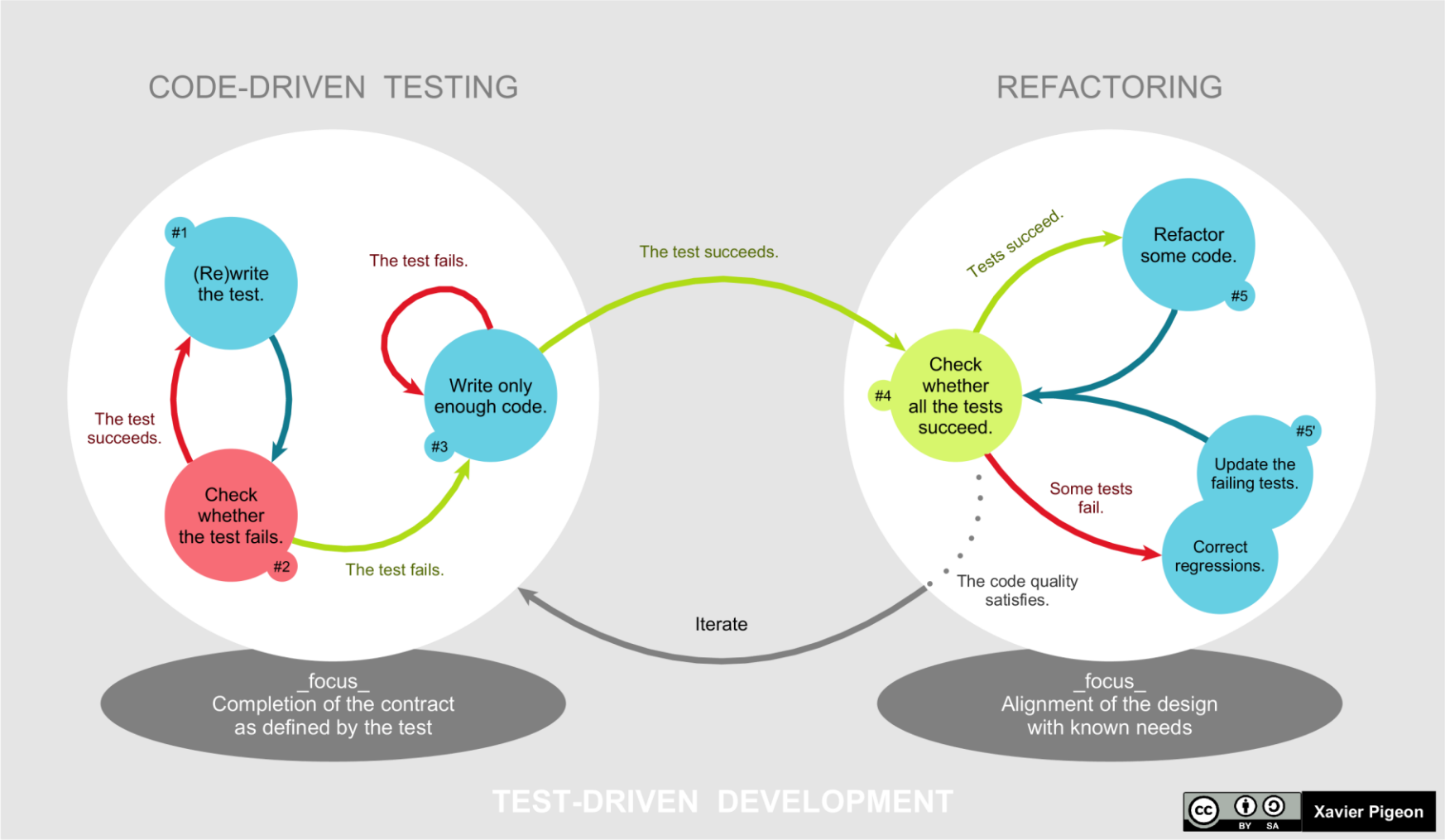
Für das Testen von `meineMethode(type1: par1, …, typen: parn)`

- werden die Werte der Typen `type1, …, typen` jeweils in eine endliche Anzahl von Klassen kategorisiert;
- jeweils ein konkreter Wert pro Klasse gewählt;
- `meineMethode` auf dem kartesischen Produkt der Werte getestet.

Beispiele:

- String: `null`, leerer String, whitespace, ein kurzer String, ein sehr langer String
- Integer: `-Integer.MAX_VALUE`, -1, 0, 1, `Integer.MAX_VALUE`
- Typische Eingabewerte und „verbotene“ Datenwerte beachten!

Test-driven development: Ablauf



Test-driven development: Grundsätzliches

- In klassischen Prozessmodellen für die Softwareentwicklung findet Testen oft im Anschluss an die Entwicklung des Systems statt.
  - Fehlerreparatur zu diesem Zeitpunkt ist teuer.
  - Testen wird regelmäßig aus Zeitgründen weggelassen.
- Test-driven development (TDD) und test-first werden ab den späten 90er-Jahren im Zuge neuer Prozessmodelle (agile Methoden, eXtreme Programming) als Alternative vorgeschlagen. Versprochene Vorteile:
  - Führt zu einfachem und testbaren Code
  - Führt zu fehlerfreierem Code, da umfangreich getestet
  - Führt zu geringerem Zeitaufwand beim Debuggen
  - Führt zu früherem Fokus auf Anforderungsdetails (die die Testquellen bilden)
  - Führt zum ständigen Vorhandensein einer lauffähigen Version (die möglicherweise nicht alle Feature umsetzt)