Softwaretests

Softwaretests sind eine der wichtigsten Maßnahmen zur Qualitätssicherung in der Software-Entwicklung.

„Ein Test [...] ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“(Denert, 1991)

„Program testing can be used to show the presence of bugs, but never show their absence!“(Edsger W. Dijkstra)

Annahmen: Welche Annahmen werden für Eingaben und andere Komponenten getroffen?

Zusicherungen: Was wird garantiert?

Beispiel BeatTheNeighbours: Formale Spezifikation von Vorbedingungen (`requires`) und Nachbedingungen (ensures) in JML[4]:

```
//@ requires boxes != null && boxes.length > 0;
//@ ensures \result >= 0 && \result < boxes.length;
//@ ensures \result == 0 || boxes[\result] >= boxes[\result-1];
//@ ensures \result == boxes.length -1
//@                    || boxes[\result] >= boxes[\result+1];
public static int strategy(int[] boxes) {
  // ...
}
```

[4] https://www.openjml.org/

Informell mit JavaDoc[5]

```
/**
 * Associates the specified value with the specified key in this map.
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 *
 * @return the previous value associated with {@code key}, or
 *         {@code null} if there was no mapping for {@code key}.
 *         (A {@code null} return can also indicate that the map
 *         previously associated {@code null} with {@code key}.)
 * @throws ClassCastException if the specified key cannot be compared
 *         with the keys currently in the map
 * @throws NullPointerException if the specified key is null
 *         and this map uses natural ordering, or its comparator
 *         does not permit null keys
 */
public V put(K key, V value) {
```

[5] How to Write Doc Comments for the Javadoc Tool
https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Komponententests

- **Komponententests** (Unit-Tests) testen die funktionale Anforderungen an einzelne Software-Komponenten
- Ist eine Funktion korrekt implementiert?
- Verschiedene Eingaben verwenden
- Vorbedingung $\Rightarrow$ Testeingabe
  Nachbedingung $\Rightarrow$ Ergebnis
- Erstellen der Testfälle am besten **vor** dem Implementieren (*test-driven development*)

Beispiel: JUnit

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

public class BeatTheNeighboursTest {
  @Test
  public void testStrategy() {
    assertEquals(0, BeatTheNeighbours.strategy(new int[]{42}));
    assertEquals(4, BeatTheNeighbours.strategy(new int[]{1, 2, 3, 4, 5}));
    assertEquals(0, BeatTheNeighbours.strategy(new int[]{5, 4, 3, 2, 1}));
    assertEquals(2, BeatTheNeighbours.strategy(new int[]{1, 2, 3, 2, 1}));
  }
}
```

Verwendung von JUnit

- JUnit Bibliothek muss eingebunden werden ($\Rightarrow$ Gradle)
- Methoden mit Annotation `@Test` werden von JUnit ausgeführt
- Klasse und Methoden müssen in JUnit 4 `public` sein und dürfen nicht `static` sein
- Methoden aus der Klasse `Assert` dienen zum Vergleichen von erwartetem und tatsächlichem Ergebnis:
  - `assertEquals`, `assertSame`, `assertArrayEquals`
  - `assertTrue`, `assertFalse`
  - `assertNull`, `assertNotNull`
  - `fail`
  - `assertThat`
- Ausführen der Tests:
  - Über Build-System (`gradle test`)
  - Über IDE ($\Rightarrow$ Demo)

Testen von Seiteneffekten

Zustandsänderungen lassen sich leicht prüfen:

```
@Test
public void putGetTest1() {
    Map<String, String> m = newInstance();
    m.put("a", "x");
    assertEquals("x", m.get("a"));
}
```

Aber:

- Schwer zu testen, dass es keine anderen ungewollten Effekte gibt.
- Andere Effekte sind schwieriger zu testen:
  - Ein-/Ausgabe
  - Zugriff auf das Dateisystem oder Netzwerk
  - Grafische Benutzeroberfläche

Testen von Seiteneffekten

Teilweise existieren Lösungen:

- Ein-/Ausgabe
  ⇒ `System.setIn` und `System.setOut`
- Zugriff auf das Dateisystem oder Netzwerk
  ⇒ Temporäres Unterverzeichnis erstellen
  ⇒ Dummys/Mocks verwenden (siehe unten)
- Grafische Benutzeroberfläche
  ⇒ Selenium (https://www.selenium.dev/)

Meist die bessere Lösung:

- Seiteneffekte und Algorithmen/Logik voneinander trennen
- Algorithmen und Logik automatisiert mit Komponententests
- Rest mit automatisierten Systemtests (oder manuell)

Test-driven development führt zu testbarem Code.

Testbarer Code

```
// Variante 1:
void compress(Path input, Path output) throws IOException { ... }
void decompress(Path input, Path output) throws IOException { ... }

// Variante 2:
byte[] compress(byte[] input) { ... }
byte[] decompress(byte[] input) { ... }

void compressFile(Path input, Path output) throws IOException {
  byte[] inBytes = Files.readAllBytes(input);
  Files.write(output, compress(inBytes));
}
void decompressFile(Path input, Path output) throws IOException {
  byte[] inBytes = Files.readAllBytes(input);
  Files.write(output, decompress(inBytes));
}
```

$\Rightarrow$ `compress` und `decompress` leichter zu testen, da keine Seiteneffekte.

Testen von Code mit Abhängigkeiten

```
public class RoutePlanner {
  private StreetDb streetDb;

  public RoutePlanner() {
    this.streetDb = new StreetDb("https://example.com/maps");
  }

  public Route planRoute(Location from, Location to) {
    // ...
  }
}
```

Frage: Wie testen ohne den Server aufzusetzen?

Dependency Injection

```
public class RoutePlanner {
  private StreetDb streetDb;
  public RoutePlanner(StreetDb db) {
    this.streetDb = db;
  }
  public Route planRoute(Location from, Location to) {
    // ...
  }
}

// Im Test:
@Test
public void testRoutePlanner() {
  // Strassen-Datenbank durch Dummy/Mock ersetzen:
  StreetDb db = new DummyStreetDb();
  RoutePlanner rp = new RoutePlanner(db);
  Route r = rp.planRoute(...);
  // ...
}
```

Test Design

Wie schreibt man gute Testfälle?

- Blackbox Testing:
  Design der Tests basiert nur auf Spezifikation.
- Whitebox Testing:
  Design der Tests basiert auch auf der konkreten Implementierung.

Whitebox Testing - Randfälle

Fehler treten oft bei Randfällen auf.

```
//@ requires boxes != null && boxes.length > 0;
//@ ensures \result >= 0 && \result < boxes.length;
//@ ensures \result == 0 || boxes[\result] >= boxes[\result-1];
//@ ensures \result == boxes.length -1
//@                    || boxes[\result] >= boxes[\result+1];
public static int strategy(int[] boxes) {
  // ...
}
```

Randfälle der Spezifikation:

- kleinste Eingabe: Array mit einem Element
- Maximum an Position `0`
- Maximum an Position `boxes.length - 1`
- Gleiche benachbarte Zahlen

Whitebox Testing - Randfälle

```
@Test
public void testStrategy() {
    // kleinste Eingabe: Array mit einem Element
    assertEquals(0, BTN.strategy(new int[]{42}));
    // Maximum an Position boxes.length − 1
    assertEquals(4, BeatTheNeighbours.strategy(new int[]{1, 2, 3, 4, 5}));
    // Maximum an Position 0
    assertEquals(0, BeatTheNeighbours.strategy(new int[]{5, 4, 3, 2, 1}));
    // andere Testfaelle :
    assertEquals(2, BeatTheNeighbours.strategy(new int[]{1, 2, 3, 2, 1}));
    assertEquals(3, BeatTheNeighbours.strategy(
                        new int[]{ 7, 9, 13, 17, 16, 14, 13, 8, 5, 1}));
}
```

Whitebox Testing - Randfälle

Randfälle testen reicht nicht immer aus.
Beispiel: Falsche Implementierung, aber besteht alle Tests.

```
public static int strategy(int[] boxes) {
  int selected = boxes.length / 2;
  while (true) {
    if (selected == 0 || boxes[selected] >= boxes[selected - 1]) {
      if (selected == boxes.length - 1
          || boxes[selected] >= boxes[selected + 1]) {
        return selected;
      } else {
        selected = (boxes.length + selected) / 2;
      }
    } else {
      selected = selected / 2;
    }
  }
}
```

Property Based Testing

Idee:

- Eingaben generieren
- Zu testenden Code aufrufen
- Eigenschaften über Ergebnis prüfen

Property Based Testing

Methoden zum generieren von Eingaben:

- Exhaustive Testing
  Systematisch alle möglichen Eingaben generieren
- QuickCheck[6]
  Zufällige Eingaben erzeugen.
  Bei gefundenem Fehler Beispiel minimieren (Shrinking)
  - junit-quickcheck: https://github.com/pholser/junit-quickcheck
- SmallCheck[7]
  Systematisch alle möglichen Eingaben bis zu einer bestimmten Größe generieren.
  Beginnend bei kleinen Eingaben.
  - Java SmallCheck https://github.com/peterzeller/java-smallcheck

[6] Koen Claessen und John Hughes, 2000, QuickCheck: a lightweight tool for random testing of Haskell programs

[7] Colin Runciman, Matthew Naylor und Fredrik Lindblad, 2008, Smallcheck and lazy smallcheck: automatic exhaustive testing for small values

Beispiel: SmallCheck

```
@RunWith(SmallCheckRunner.class)
public class BeatTheNeighboursSmallcheck {

    @Property(maxDepth = 30, maxInvocations = 1000000, timeout = 30)
    public void testBeatTheNeighbours(int[] boxes) {
        assumeTrue(boxes != null && boxes.length > 0);
        int result = BeatTheNeighbours.strategy(boxes);
        assertTrue(result >= 0 && result < boxes.length);
        assertTrue(result == 0
                || boxes[result] >= boxes[result - 1]);
        assertTrue(result == boxes.length - 1
                || boxes[result] >= boxes[result + 1]);
    }
}
```

java.lang.AssertionError: Test failed when calling testBeatTheNeighbours with the following arguments:
boxes = [0, 0, 1, 0, 0, 0]
Test timed out after 30 seconds.

Blackbox Testing

Ziel: Alle Fälle der Implementierung abdecken
Coverage Kriterien:

- Path-Coverage
  Jeder Ausführungspfad durch die Methode muss mindestens einmal getestet werden.
  Mit Schleifen nicht möglich, da unendlich viele Pfade.
- Condition-Coverage
  Jede Bedingung muss einmal zu `true` und einmal zu `false` auswerten.
- Statement-Coverage
  Jede Anweisung muss einmal ausgeführt werden.

Tool Support: Jacoco[8]

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
LeftistTree(Object, Comparator)		0%		n/a	1	1	4	4	1	1
merge(PriorityQueue)		64%		25%	2	3	2	7	0	1
comparator()		0%		n/a	1	1	1	1	1	1
merge(LeftistTree.Node, LeftistTree.Node)		100%		100%	0	6	0	18	0	1
deleteMin()		100%		n/a	0	1	0	7	0	1
insert(Object)		100%		n/a	0	1	0	3	0	1
isEmpty()		100%		100%	0	2	0	3	0	1
LeftistTree(Comparator)		100%		n/a	0	1	0	3	0	1
getMin()		100%		n/a	0	1	0	1	0	1
Total	26 of 169	84%	3 of 16	81%	4	17	7	47	2	9

[8] Verwendung mit Gradle: https://docs.gradle.org/current/userguide/jacoco_plugin.html

Tool Support: Jacoco[9]

```
39.     @Override
40.     public void merge(PriorityQueue<E> x) {
41.         // efficient for same data-structure
42.         if (x instanceof LeftistTree<?>) {
43.             LeftistTree<E> other = (LeftistTree<E>) x;
44.             root = merge(root, other.root);
45.             //other.root = null;
46.         }
47.         // not that efficient but still possible
48.         else {
49.             while (!x.isEmpty()) {
50.                 insert(x.deleteMin());
51.             }
52.         }
53.     }
54.
```

[9] Verwendung mit Gradle: https://docs.gradle.org/current/userguide/jacoco_plugin.html