

(UNIT) TESTING

Lars Briem

(briem.lars@googlemail.com)

Duale Hochschule Baden Württemberg - Standort Karlsruhe

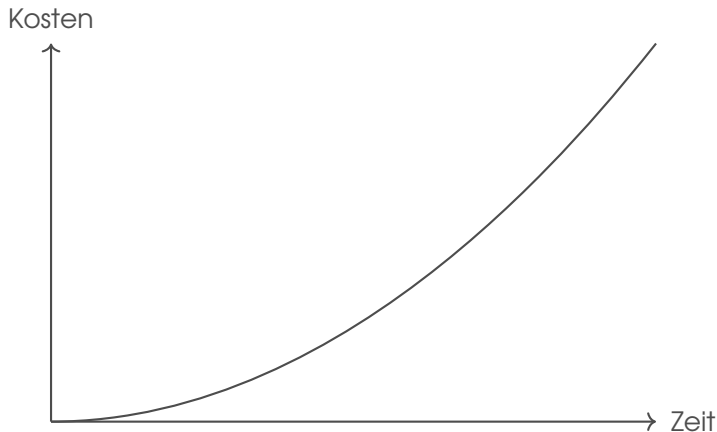
Bekannte Fehler in der Entwicklung

- ▶ Entwicklung ist fehlerbehaftet
- ▶ Viele bekannte Beispiele
 - ▶ Abstürze von Raketen
 - ▶ Offene Sicherheitslücken
- ▶ Fehlerursachen
 - ▶ Falsche Einschätzung von Risiken
 - ▶ Fehlendes Testen

Auswirkungen von Fehlern

- ▶ Fehler binden und vernichten Ressourcen
 - ▶ Fehler suchen
 - ▶ Fehler beheben
 - ▶ Kunden Fehler erklären
- ▶ Anschaulich steigen die Kosten eines Fehlers mit seiner Existenz
 - ▶ Selbst bei sofortigem finden und beheben

Geschätzte Kosten eines Fehlers



Verpflichtung zum Testen

- ▶ Entwickler sind gesetzlich dazu verpflichtet ihre Produkte zu Testen, dazu zählt auch Software
 - ▶ Nicht Testen ist grob fahrlässig
 - ▶ Garantie und Gewährleistung gilt auch bei Software
- ▶ Tests schützen bestehende Funktionen
 - ▶ Zufällige Veränderungen an bestehenden Funktionen werden erkannt
- ▶ Orientierungshilfe und Dokumentation

Test als Hilfsmittel

- ▶ Neuere Entwicklungsmethoden nutzen Tests
 - ▶ Test First
 - ▶ Test Driven Development
- ▶ Umkehrung von „zusätzlichem Aufwand“ für Tests zu großem Nutzen durch Tests
- ▶ Tests unterscheiden dabei zwischen zufälliger und gewollter Funktionalität

Arten von Tests

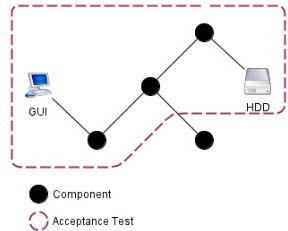
- ▶ Vielfalt an möglichen Testarten und Benennungen
 - ▶ Akzeptanztests
 - ▶ Integrationstests
 - ▶ Komponententests
 - ▶ Performancetests

⇒ wikipedia.org/wiki/Software_testing

Akzeptanztests

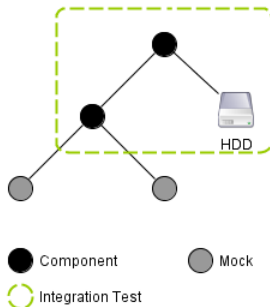
- ▶ Test des kompletten Systems
- ▶ Realistische Laufzeitumgebung
 - ▶ Hardware, Datenbank
- ▶ Steuerung mit Mitteln des Benutzers
 - ▶ Bedienoberfläche zur Interaktion
- ▶ Absegnung durch Auftraggeber

⇒ Ziel: Echte Bedienszenarien testen



Integrationstests

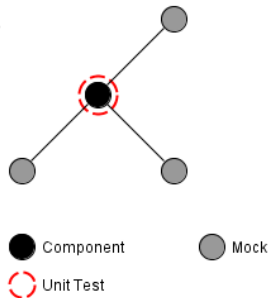
- ▶ Nur relevante Teile des Systems werden gestartet
- ▶ Nicht zu testende Teile durch Stellvertreter ersetzen
- ▶ Durchführung mittels Testframework
 - ▶ Methodenaufrufe
 - ▶ Interaktion der Systemteile untereinander



⇒ Ziel: Zusammenspiel der Komponenten sicherstellen

Komponenten- / Unit Tests

- ▶ Nur relevanter Teil des Systems wird gestartet
- ▶ Alle anderen Teile durch Stellvertreter ersetzen
- ▶ Durchführung mittels Testframework
 - ▶ Methodenaufrufe
 - ▶ Überprüfung der Rückgabewerte



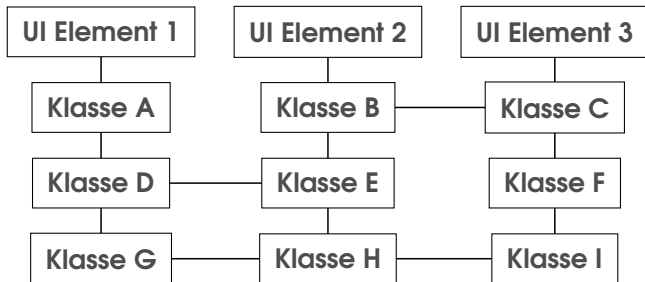
⇒ Ziel: Funktionalität einzelner Komponente (Unit) sicherstellen

Komponenten- / Unit Tests

- ▶ Tests für einzelne Komponenten (Units)
 - ▶ Unabhängig von anderen Komponenten
 - ▶ Pro Test ein Aspekt der Komponente
- ▶ Ausführbare und selbst-überprüfende Spezifikation der Komponente
 - ▶ „Komponente“ meist Klasse
- ▶ Aktive Dokumentation für Klasse
 - ▶ Dokumentiert Verwendung und erwartetes Verhalten in Regel- und Ausnahmefällen

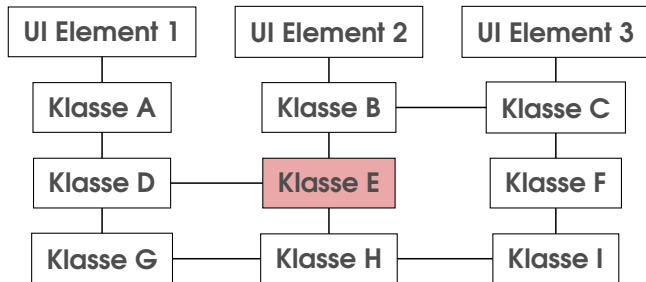
Was ist eine Komponente

- Ein Softwaresystem zur Auswertung von Daten



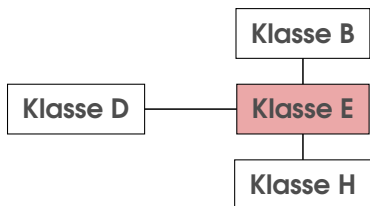
Was ist eine Komponente

- ▶ Ein Softwaresystem zur Auswertung von Daten
- ▶ Wie kann eine Komponente ohne Abhängigkeiten getestet werden?



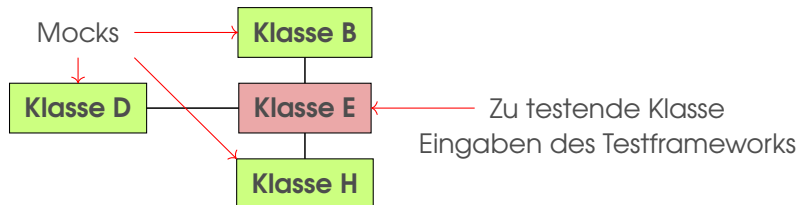
Was ist eine Komponente

- ▶ Ein Softwaresystem zur Auswertung von Daten
- ▶ Wie kann eine Komponente ohne Abhängigkeiten getestet werden?

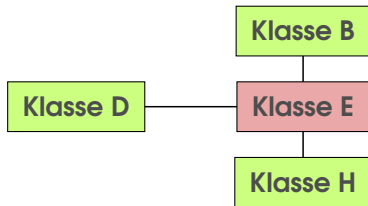


Was ist eine Komponente

- ▶ Ein Softwaresystem zur Auswertung von Daten
- ▶ Wie kann eine Komponente ohne Abhängigkeiten getestet werden?

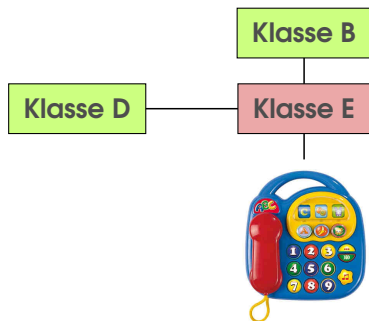


Isolation durch Mock-Objekte



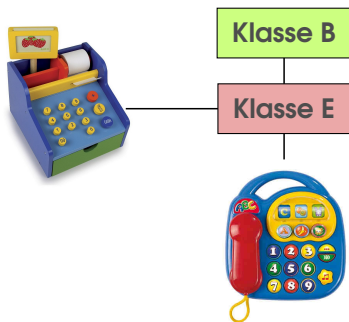
- ▶ Ersetzen der Abhängigkeiten
- ▶ Stellvertreter „Mocks“ verwenden
- ▶ Minimale notwendige Funktionalität (Fakes)
- ▶ „Gut genug“ für Test

Isolation durch Mock-Objekte



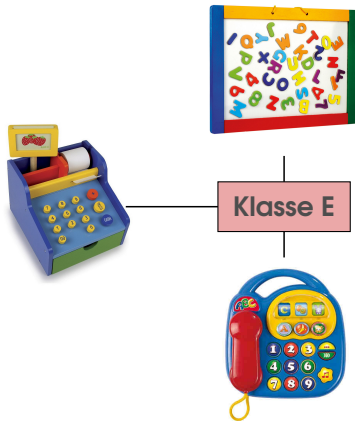
- ▶ Ersetzen der Abhängigkeiten
- ▶ Stellvertreter „Mocks“ verwenden
- ▶ Minimale notwendige Funktionalität (Fakes)
- ▶ „Gut genug“ für Test

Isolation durch Mock-Objekte



- ▶ Ersetzen der Abhängigkeiten
- ▶ Stellvertreter „Mocks“ verwenden
- ▶ Minimale notwendige Funktionalität (Fakes)
- ▶ „Gut genug“ für Test

Isolation durch Mock-Objekte



- ▶ Ersetzen der Abhängigkeiten
- ▶ Stellvertreter „Mocks“ verwenden
- ▶ Minimale notwendige Funktionalität (Fakes)
- ▶ „Gut genug“ für Test

xUnit Testframework

- ▶ Vorlage für Unit-Tests
- ▶ Aufbau von xUnit Tests vergleichbar zwischen den Implementierungen
 - ▶ Test endet mit einer „Assertion“ (Behauptung)
 - ▶ Testframework überprüft die Behauptung
- ▶ Trennung zwischen Produktiv- und Testcode
- ▶ Für (fast) alle Sprachen existieren Implementierungen
 - ▶ https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

xUnit Aufbau

```
@Test
public void removesBrackets {
    String text = "[blub]";

    String result = Util.removeBrackets(text);

    assertEquals("blub", result);
}
```

xUnit Aufbau

```
@Test
public void removesBrackets {
    String text = "[blub]";
    String result = Util.removeBrackets(text);
    assertEquals("blub", result);
}
```

← Arrange

Arrange Initialisieren der Testumgebung

xUnit Aufbau

```
@Test
public void removesBrackets {
    String text = "[blub]";
    String result = Util.removeBrackets(text);
    assertEquals("blub", result);
}
```

← Arrange

← Act

Arrange Initialisieren der Testumgebung

Act Ausführen des zu testenden Codes

xUnit Aufbau

```
@Test
public void removesBrackets {
    String text = "[blub]";           ← Arrange

    String result = Util.removeBrackets(text); ← Act

    assertEquals("blub", result);    ← Assert
}
```

Arrange Initialisieren der Testumgebung

Act Ausführen des zu testenden Codes

Assert Überprüfen des Ergebnisses


JUnit




- ▶ Ursprünglich entwickelt von Kent Beck
- ▶ Mittlerweile in Version 4.12
- ▶ Version 5 schon weit fortgeschritten, bringt einige neue Funktionen
- ▶ Quasi Standard in der Java Entwicklung



```
public class RemoveTest {  
  
    public RemoveTest() {  
        super();  
    }  
    @Before  
    public void beforeEachTest() {  
        // not needed here  
    }  
    @Test  
    public void removesBrackets() {  
        String text = "[blub]";  
        String result = Remove.brackets(text);  
        assertEquals("blub", result);  
    }  
    @After  
    public void afterEachTest() {  
        // not needed here  
    }  
}
```

JUnit

```
public class RemoveTest {  Klassenname

    public RemoveTest() {
        super();
    }
    @Before
    public void beforeEachTest() {
        // not needed here
    }
    @Test
    public void removesBrackets() {  Testname
        String text = "[blub]";
        String result = Remove.brackets(text);  Testinhalt
        assertEquals("blub", result);  Assertion
    }
    @After
    public void afterEachTest() {
        // not needed here
    }
}
```

```
public class RemoveTest {  
  
    public RemoveTest() {  
        super();  
    }  
    @Before  
    public void beforeEachTest() {  
        // not needed here  
    }  
    @Test  
    public void removesBrackets() {  
        String text = "[blub]";  
        String result = Remove.brackets(text);  
        assertEquals("blub", result);  
    }  
    @After  
    public void afterEachTest() {  
        // not needed here  
    }  
}
```

← Klassenname

← Optionaler Konstruktor

← Testname

← Testinhalt

← Assertion

JUnit

```
public class RemoveTest { ← ————— Klassenname

    public RemoveTest() { ← ————— Optionaler Konstruktor
        super();
    }
    @Before
    public void beforeEachTest() { ← ————— Optionale Initialisierung
        // not needed here
    }
    @Test
    public void removesBrackets() { ← ————— Testname
        String text = "[blub]";
        String result = Remove.brackets(text); ← ————— Testinhalt
        assertEquals("blub", result); ← ————— Assertion
    }
    @After
    public void afterEachTest() { ← ————— Optionales Aufräumen
        // not needed here
    }
}
```

Beispieltest in JUnit

```
public class RemoveTest {  
    @Test  
    public void removesBrackets() {  
        String text = "[blub]";  
        String result = Remove.brackets(text);  
        assertEquals("blub", result);  
    }  
    @Test  
    public void removesParentheses() {  
        String text = "(blub)";  
        String result = Remove.parentheses(text);  
        assertEquals("blub", result);  
    }  
    @Test  
    public void removesBraces() {  
        String text = "{blub}";  
        String result = Remove.braces(text);  
        assertEquals("blub", result);  
    }  
}
```

- ▶ Beliebige Reihenfolge
- ▶ Alle Tests einzeln ausführen
- ▶ Einzel- und Gesamtergebnis

Überprüfung durch `asserts`

- ▶ `assertEquals`: Überprüfung auf Gleichheit
- ▶ `assertSame`: Überprüfung auf gleiche Referenz
- ▶ `assertTrue`: Überprüfung, ob wahr
- ▶ `assertNotNull`: Überprüfung, ob ein Element existiert
- ▶ `assertThat`: Überprüfung mit übergebenem „Matcher“

Überprüfung durch Matcher

- ▶ Überprüfung von Zahlen auf größer und kleiner
 - ▶ `assertThat(number, lessThan(0));`
 - ▶ `assertThat(number, greaterThan(0));`
- ▶ Überprüfung von Objekten auf Gleichheit
 - ▶ `assertThat(object, equalTo(other));`
- ▶ Überprüfung von Listen und ähnlichem
 - ▶ `assertThat(list, contains("element"));`
 - ▶ `assertThat(list, empty());`

Überprüfung durch Matcher

- ▶ Überprüfung von Zahlen auf größer und kleiner
 - ▶ `assertThat(number, is(lessThan(0)));`
 - ▶ `assertThat(number, is(greaterThan(0)));`
- ▶ Überprüfung von Objekten auf Gleichheit
 - ▶ `assertThat(object, is(equalTo(other)));`
- ▶ Überprüfung von Listen und ähnlichem
 - ▶ `assertThat(list, contains("element"));`
 - ▶ `assertThat(list, is(empty()));`

⇒ Matcher können oft verschachtelt werden

Beispieltest aus der Wildnis

```
@Test
public void usesScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(2.0d);
    Distance length = new Distance(2, METER);

    Distance scaled = viewer.scale(length);

    assertThat(scaled.getLengthIn(METER), is(closeTo(4.0d, 1E-2)));
}
```

Umgang mit Exceptions

```
@Test
public void needsExistingScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(null);
    Distance length = new Distance(2, METER);

    try {
        Distance scaled = viewer.scale(length);
    } catch (NullPointerException exception) {
    }
}
```

Umgang mit Exceptions

```
@Test
public void needsExistingScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(null);
    Distance length = new Distance(2, METER);

    try {
        Distance scaled = viewer.scale(length);
    } catch (NullPointerException exception) {
    }
}
```

← Lläuft ohne
Fehler durch

Umgang mit Exceptions

```
@Test
public void needsExistingScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(null);
    Distance length = new Distance(2, METER);

    try {
        Distance scaled = viewer.scale(length);
        fail("Expected a NullPointerException to be thrown");
    } catch (NullPointerException exception) {
        assertThat(exception.getMessage(), "Scale factor missing");
    }
}
```

⇒ Wird in JUnit 3.x verwendet

Umgang mit Exceptions

```
@Test(expected=NullPointerException.class)
public void needsExistingScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(null);
    Distance length = new Distance(2, METER);

    Distance scaled = viewer.scale(length);
}
```

⇒ Ist seit JUnit 4.x möglich

Umgang mit Exceptions

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test(expected=NullPointerException.class)
public void needsExistingScaleFactor() {
    Viewer viewer = new Viewer();
    viewer.setScaleFactor(null);
    Distance length = new Distance(2, METER);

    thrown.expect(NullPointerException.class);
    thrown.expectMessage("Scale factor missing");
    Distance scaled = viewer.scale(length);
}
```

⇒ Seit 4.7 gibt es „Rule“s in JUnit

Genauigkeit bei Gleitkommazahlen

- ▶ Zeichenketten und Ganzzahlen können exakt überprüft werden
- ▶ Bei Gleitkommazahlen funktioniert das nicht
- ▶ Für Überprüfungen von Gleitkommazahlen muss eine Genauigkeit angegeben werden
 - ▶ `assertThat(value, closeTo(1.0d, 1E-2))`
- ▶ Genauigkeit kann in 1E-n Notation angegeben werden
 - ▶ n gibt die Anzahl an Nachkommastellen an

Arithmetische Spezialitäten

- ▶ Ganzzahl Operationen kennen keinen Overflow
 - ▶ `IntegerOverflowException` existiert nicht
 - ▶ `Integer.MAX_VALUE + 1 = Integer.MIN_VALUE`;
- ▶ Kleinster Wert von Datentypen haben unterschiedliche Semantik
 - ▶ `Double.MIN_VALUE` kleinster positiver Wert
 - ▶ `Integer.MIN_VALUE` „größter“ negativer Wert
- ▶ `Double` kennt kein `DivisionByZero`
 - ▶ `1.0d / 0.0d ⇒ Double.POSITIVE_INFINITY`
 - ▶ `1.0d / -0.0d ⇒ Double.NEGATIVE_INFINITY`

Ergebnis eines Tests

- ▶ Success – bestanden
 - ▶ Testmethode wurde erfolgreich durchlaufen
 - ▶ Keine Assertion hat einen Fehler gefunden
 - ▶ Leere Testmethoden bestehen immer
- ▶ Failure – fehlgeschlagen
 - ▶ Eine Assertion ist fehlgeschlagen
- ▶ Error – unerwartet fehlgeschlagen
 - ▶ Der Test wird durch einen unerwarteten Fehler (Exception oder Error) beendet

⇒ Ein bestandener Test ist, kein Beweis, dass die Software fehlerfrei ist

Eigenschaften guter Tests – A-TRIP

- ▶ Erstellung der Tests zeitlich nahe am Produktivcode oder vorher
- ▶ Gute Tests vereinfachen die Entwicklung, schlechte behindern sie unter Umständen
- ▶ Die A-TRIP Eigenschaften
 - ▶ Automatic
 - ▶ Thorough
 - ▶ Repeatable
 - ▶ Independent
 - ▶ Professional

Automatic – Automatisch

- ▶ Tests müssen einfach ausführbar sein
 - ▶ Maximal ein Knopfdruck oder Befehl zum Starten der Tests
- ▶ Tests müssen automatisch ablaufen
 - ▶ Keine manuelle Eingabe von Daten
- ▶ Tests müssen sich selbst überprüfen
 - ▶ Es gibt nur die Ergebnisse **bestanden** oder **fehlgeschlagen**

⇒ Minimale Anforderungen bei der Ausführung

Thorough – Vollständig

- ▶ Ein Test muss alles notwendige überprüfen
 - ▶ Notwendiges liegt im Ermessen des Entwicklers
- ▶ Iteratives Vorgehen zur Erstellung der Tests
 - ▶ Alle kritischen Stellen des Systems testen
 - ▶ Beim Auftreten eines Fehlers, einen Test schreiben, der nur diesen Fehler in Zukunft testet
- ▶ Fehler sind nicht gleichmäßig über den Code verteilt
 - ▶ Fehler „klumpen“ zusammen
 - ▶ Beim Auftreten eines Fehlers dessen Umgebung überprüfen

Repeatable – Wiederholbar

- ▶ Test muss beliebig wiederholbar sein und immer das gleiche Ergebnis liefern
 - ▶ Ergebnis ist unabhängig von der Umgebung
 - ▶ Zeit und Zufall sind häufige Fehlerquellen, genauso wie Multithreading
 - ▶ Operationen mit dem Dateisystem sind plattformabhängig
- ▶ Tests, die ohne Änderung fehlschlagen, sind selbst fehlerhaft
 - ▶ Fehlerhafte Tests sind besser als keine Tests

Independent – Unabhängig

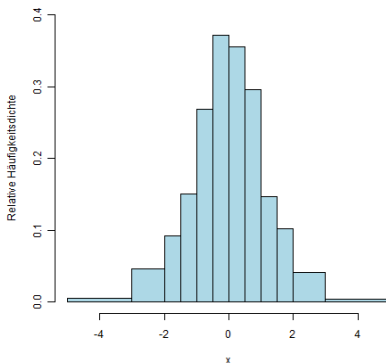
- ▶ Tests dürfen keine Abhängigkeit zu anderen Tests haben
 - ▶ Jeder Test ist alleine lauffähig, die Reihenfolge ist egal
 - ▶ z.B. keine Vorbereitung der Datenbank durch einen anderen Test
- ▶ Tests sollen stark auf ihre Aufgabe fokussiert sein
 - ▶ Liefert gute Rückmeldung, wo der Fehler auftritt
 - ▶ Setup(@Before) und Teardown(@After) vereinfachen das vorbereiten gleicher Testumgebungen für unterschiedliche Tests

Professional – Professionell

- ▶ Tests unterliegen den gleichen Qualitätsstandards wie „Produktivcode“
 - ▶ Wiederverwendung von bestehendem Code
 - ▶ Hilfsfunktionen und Klassen „nur“ zum Testen sind erlaubt und erwünscht
 - ▶ Fehler in Tests ebenfalls teuer
- ▶ Keine unnötigen Tests schreiben
 - ▶ Tests nur des „Tests“ wegen schreiben ist unnötig
 - ▶ z.B. „Getter“ testen
- ▶ Tests sind Teil der Dokumentation

Beispiel mit einem Histogramm

- Stellt die Häufigkeit von Werten dar



Eventuell aus der Wildnis

► Erfüllt dieser Code die A-TRIP Eigenschaften?

```
public class Histogram {
    int v();
    public Histogram(int l) {
        v = new int(l);
    }
    public void add(int a) {
        v(a) = v(a) + 1;
    }
    public void addRandom() {
        int a = (int) (Math.random() * 10);
        v(a) = v(a) + 1;
    }
    public void print(int i, int j) {
        for (; i <= j; i++) {
            System.out.println(i + ":" + v(i));
        }
    }
    public int size() {
        int elements = 0;
        for (int element : v) {
            elements += element;
        }
        return elements;
    }
}
```

```
public class HistogramTest {

    @Test
    public void test1() {
        Histogram histogram = new Histogram(9);
        histogram.add(0);
        histogram.add(1);
        histogram.addRandom();
        histogram.add(1);

        histogram.print(0,9);
    }
}
```


Eventuell aus der Wildnis

```
public class Histogram {  
  
    TreeMap<Integer, Integer> values = new TreeMap<>();  
    private Supplier<Integer> random;  
  
    public Histogram(Supplier<Integer> random) {  
        super();  
        this.random = random;  
    }  
  
    public void add(int number) {  
        values.merge(number, 1, (k, v) -> v + 1);  
    }  
  
    public void addRandom() {  
        int nextRandom = random.get() * 10;  
        add(nextRandom);  
    }  
  
    public void print(int from, int to,  
        BiConsumer<Integer, Integer> consumer) {  
        for (; from <= to; from++) {  
            consumer.accept(from, values.getOrDefault(from, 0));  
        }  
    }  
}
```

Eventuell aus der Wildnis

```
public class Histogram {
```

```
    TreeMap<Integer, Integer> values = new TreeMap<>();  
    private Supplier<Integer> random;
```

```
    public Histogram(Supplier<Integer> random) {  
        super();  
        this.random = random;  Übergeben der Zufallsabhängigkeit  
    }
```

```
    public void add(int number) {  
        values.merge(number, 1, (k, v) -> v + 1);  
    }
```


```
    public void addRandom() {  
        int nextRandom = random.get() * 10;  
        add(nextRandom);  
    }
```

```
    public void print(int from, int to,  
        BiConsumer<Integer, Integer> consumer) {  
        for (; from <= to; from++) {  
            consumer.accept(from, values.getDefault(from, 0));  
        }  
    }  
}
```

Eventuell aus der Wildnis

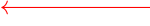
```
public class Histogram {
```

```
    TreeMap<Integer, Integer> values = new TreeMap<>();  
    private Supplier<Integer> random;
```

```
    public Histogram(Supplier<Integer> random) {  
        super();  
        this.random = random;  Übergeben der Zufallsabhängigkeit  
    }
```

```
    public void add(int number) {  
        values.merge(number, 1, (k, v) -> v + 1);  
    }
```

```
    public void addRandom() {  
        int nextRandom = random.get() * 10;  
        add(nextRandom);  
    }
```

```
    public void print(int from, int to,  
        BiConsumer<Integer, Integer> consumer) {  Übergeben der Ausgabe als Consumer  
        for (; from <= to; from++) {  
            consumer.accept(from, values.getDefault(from, 0));  
        }  
    }
```

Eventuell aus der Wildnis

```
public class HistogramTest {  
  
    @Test  
    public void usesRandomNumber() {  
        Histogram histogram = new Histogram(this::random);  
        histogram.add(0);  
        histogram.add(1);  
        histogram.addRandom();  
        histogram.add(1);  
  
        int from = 0;  
        int to = 10;  
        Map<Integer, Integer> values = new HashMap<>();  
        histogram.print(from, to, values::put);  
  
        assertThat(values.get(0), is(1));  
        assertThat(values.get(1), is(2));  
    }  
  
    private int random() {  
        return 42;  
    }  
}
```

Eventuell aus der Wildnis

```
public class HistogramTest {  
  
    @Test  
    public void usesRandomNumber() {  
        Histogram histogram = new Histogram(this::random);  
        histogram.add(0);  
        histogram.add(1);  
        histogram.addRandom();  
        histogram.add(1);  
  
        int from = 0;  
        int to = 10;  
        Map<Integer, Integer> values = new HashMap<>();  
        histogram.print(from, to, values::put);  
  
        assertThat(values.get(0), is(1));  
        assertThat(values.get(1), is(2));  
    }  
  
    private int random() {  
        return 42;  
    }  
}
```

← Zufall muss gesteuert werden

Eventuell aus der Wildnis

```
public class HistogramTest {
```

```
    @Test
```

```
    public void usesRandomNumber() {
```

```
        Histogram histogram = new Histogram(this::random);
```

```
        histogram.add(0);
```

```
        histogram.add(1);
```

```
        histogram.addRandom();
```

```
        histogram.add(1);
```

```
        int from = 0;
```

```
        int to = 10;
```

```
        Map<Integer, Integer> values = new HashMap<>();
```

```
        histogram.print(from, to, values::put);
```

```
        assertThat(values.get(0), is(1));
```

```
        assertThat(values.get(1), is(2));
```

```
    }
```

```
    private int random() {
```

```
        return 42;
```

```
    }
```

```
}
```

← Assertions sind Pflicht

← Zufall muss gesteuert werden

Wiederholung: Asserts in JUnit

- ▶ `assertEquals`: Überprüfung auf Gleichheit
- ▶ `assertSame`: Überprüfung auf gleiche Referenz
- ▶ `assertTrue`: Überprüfung, ob wahr
- ▶ `assertNull`: Überprüfung, ob ein Element existiert
- ▶ `assertThat`: Überprüfung mit übergebenem „Matcher“ (`contains()`, `equalTo()`, ...)

Überprüfungen mit JUnit Matchern

- ▶ Matcher bieten eine einfache Möglichkeit weitere Überprüfungen hinzuzufügen
- ▶ Matcher bestehen aus 3 Bestandteilen
 - ▶ Überprüfung eines Wertes mit einem erwarteten Wert
 - ▶ Beschreibung welcher Wert erwartet wird
 - ▶ Beschreibung welcher Wert überprüft wurde

Beispiel aus der Wildnis

- Eine Fahrplansuche berechnet eine Reise zwischen zwei Haltestellen

```
public class Search {  
    Journey journeyFrom(Stop start, Stop toEnd) {  
        ...  
    }  
}
```

```
public interface Journey {  
    Stop start();  
    Stop end();  
    Duration duration();  
}
```

Beispiel mit Standard-Matchern

```
@Test
public void startsJourneyAtCorrectStop() throws Exception {
    Stop mainStation = new Stop("Bahnhof");
    Stop toMarket = new Stop("Marktplatz");

    Search search = new Search();
    Journey journey = search.journeyFrom(mainStation, toMarket);

    assertThat(journey.start(), is(equalTo(mainStation)));
}
```

Beispiel mit Standard-Matchern

```
@Test
public void startsJourneyAtCorrectStop() throws Exception {
    Stop mainStation = new Stop("Bahnhof");
    Stop toMarket = new Stop("Marktplatz");

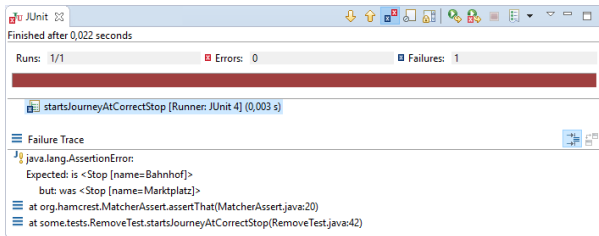
    Search search = new Search();
    Journey journey = search.journeyFrom(mainStation, toMarket);

    assertThat(journey.start(), is(equalTo(mainStation)));
}
```

Beispiel mit Standard-Matchern

`@Test`

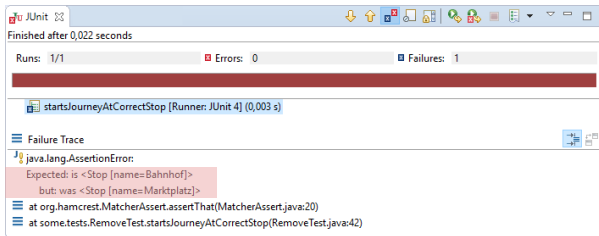
```
public void startsJourneyAtCorrectStop() throws Exception {  
    Stop mainStation = new Stop("Bahnhof");  
    Stop toMarket = new Stop("Marktplatz");  
  
    Search search = new Search();  
    Journey journey = search.journeyFrom(mainStation, toMarket);  
  
    assertThat(journey.start(), is(equalTo(mainStation)));  
}
```



Beispiel mit Standard-Matchern

`@Test`

```
public void startsJourneyAtCorrectStop() throws Exception {  
    Stop mainStation = new Stop("Bahnhof");  
    Stop toMarket = new Stop("Marktplatz");  
  
    Search search = new Search();  
    Journey journey = search.journeyFrom(mainStation, toMarket);  
  
    assertThat(journey.start(), is(equalTo(mainStation)));  
}
```



Verbesserung der Fehlermeldung

```
@Test
public void startsJourneyAtCorrectStop() throws Exception {
    Stop mainStation = new Stop("Bahnhof");
    Stop toMarket = new Stop("Marktplatz");

    Search search = new Search();
    Journey journey = search.journeyFrom(mainStation, toMarket);

    assertThat(journey, new StartsAt(mainStation));
}
```


Verbesserung der Fehlermeldung

```
@Test
public void startsJourneyAtCorrectStop() throws Exception {
    Stop mainStation = new Stop("Bahnhof");
    Stop toMarket = new Stop("Marktplatz");

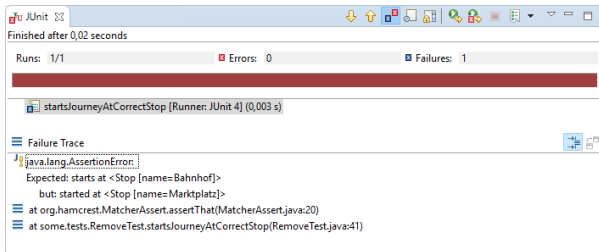
    Search search = new Search();
    Journey journey = search.journeyFrom(mainStation, toMarket);

    assertThat(journey, new StartsAt(mainStation));
}
```

Verbesserung der Fehlermeldung

@Test

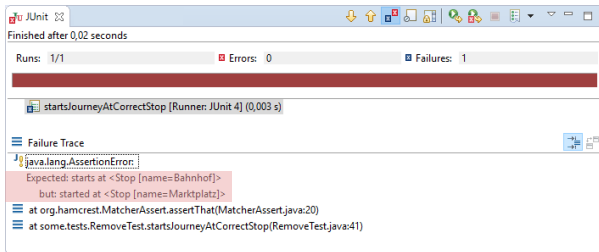
```
public void startsJourneyAtCorrectStop() throws Exception {  
    Stop mainStation = new Stop("Bahnhof");  
    Stop toMarket = new Stop("Marktplatz");  
  
    Search search = new Search();  
    Journey journey = search.journeyFrom(mainStation, toMarket);  
  
    assertThat(journey, new StartsAt(mainStation));  
}
```



Verbesserung der Fehlermeldung

@Test

```
public void startsJourneyAtCorrectStop() throws Exception {  
    Stop mainStation = new Stop("Bahnhof");  
    Stop toMarket = new Stop("Marktplatz");  
  
    Search search = new Search();  
    Journey journey = search.journeyFrom(mainStation, toMarket);  
  
    assertThat(journey, new StartsAt(mainStation));  
}
```



Selbst entwickelter Matcher

```
public class StartsAt extends TypeSafeMatcher<Journey> {  
    private final Stop start;  
    public StartsAt(Stop start) {  
        super();  
        this.start = start;  
    }  
}
```

```
    protected boolean matchesSafely(Journey journey) {  
        return start.equals(journey.start());  
    }  
}
```

```
}
```

Selbst entwickelter Matcher

```
public class StartsAt extends TypeSafeMatcher<Journey> {  
    private final Stop start;  
    public StartsAt(Stop start) {  
        super();  
        this.start = start;  
    }  
  
    protected boolean matchesSafely(Journey journey) {  
        return start.equals(journey.start()); ← Überprüfung der Werte  
    }  
  
}
```

Selbst entwickelter Matcher

```
public class StartsAt extends TypeSafeMatcher<Journey> {  
    private final Stop start;  
    public StartsAt(Stop start) {  
        super();  
        this.start = start;  
    }  
    public void describeTo(Description description) {  
        description.appendText("starts at ");  
        description.appendValue(start);  
    }  
    protected boolean matchesSafely(Journey journey) {  
        return start.equals(journey.start());  
    }  
    protected void describeMismatchSafely(Journey journey,  
        Description mismatchDescription)  
    {  
        mismatchDescription.appendText("started at ");  
        mismatchDescription.appendValue(journey.start());  
    }  
}
```

Selbst entwickelter Matcher

```
public class StartsAt extends TypeSafeMatcher<Journey> {  
    private final Stop start;  
    public StartsAt(Stop start) {  
        super();  
        this.start = start;  
    }  
    public void describeTo(Description description) {  
        description.appendText("starts at ");  
        description.appendValue(start);  
    }  
    protected boolean matchesSafely(Journey journey) {  
        return start.equals(journey.start());  
    }  
    protected void describeMismatchSafely(Journey journey,  
        Description mismatchDescription)  
    {  
        mismatchDescription.appendText("started at ");  
        mismatchDescription.appendValue(journey.start());  
    }  
}
```

Beschreibung, was erwartet wird

Beschreibung, was tatsächlich berechnet wurde

Selbst entwickelter Matcher

```
public class StartsAt extends TypeSafeMatcher<Journey> {  
    private final Stop start;  
    public StartsAt(Stop start) {  
        super();  
        this.start = start;  
    }  
    public void describeTo(Description description) {  
        description.appendText("starts at ");  
        description.appendValue(start);  
    }  
    protected boolean matchesSafely(Journey journey) {  
        return start.equals(journey.start());  
    }  
    protected void describeMismatchSafely(Journey journey,  
        Description mismatchDescription)  
    {  
        mismatchDescription.appendText("started at ");  
        mismatchDescription.appendValue(journey.start());  
    }  
}
```

Beschreibung, was erwartet wird

Beschreibung, was tatsächlich berechnet wurde

Verbesserung der Fehlermeldung

```
@Test
public void startsJourneyAtCorrectStop() throws Exception {
    Stop mainStation = new Stop("Bahnhof");
    Stop toMarket = new Stop("Marktplatz");

    Search search = new Search();
    Journey journey = search.journeyFrom(mainStation, toMarket);

    assertThat(journey, startsAt(mainStation));
}
```

↑
statische Factory-Methode

Fazit zu Matchern

- + Matcher verbessern die Lesbarkeit des Tests
 - + Statische Methoden zur Erstellung von Matchern verbessern die Lesbarkeit noch weiter
- + Matcher erzeugen sprechendere Fehlermeldungen
- + Matcher können wiederverwendet werden
 - + Die Fehlermeldungen sind an allen Stellen einheitlich
- Die Implementierung benötigt Zeit
 - Die Funktion von Matchern kann zusätzlich über eigene Unit Tests überprüft werden

Mock Objekte

- ▶ Mocks reduzieren die Abhängigkeit zu anderen Komponenten
 - ▶ Reduzieren den Aufwand zur Initialisierung von Abhängigkeiten
 - ▶ Ersatz für externe Abhängigkeiten (Datenbank, File, ...)
- ▶ Stellvertreter für „richtige“ Objekte
 - ▶ Ersatz für komplexe Objekte
 - ▶ Vergleichbar mit Licht- oder Stuntdouble in Filmen

Wiederholung: Isolation durch Mocks



- ▶ Ersetzen der Abhängigkeiten
- ▶ Stellvertreter „Mocks“ verwenden
- ▶ Minimale notwendige Funktionalität (Fakes)
- ▶ „Gut genug“ für Test

Mock Objekte erstellen

- ▶ Mocks können manuell erstellt werden
 - ▶ Abhängigkeiten durch Interfaces „kapseln“
 - ▶ Mock Objekt als eine weitere Klasse des Interfaces implementieren
 - ▶ Mocks enthalten nur das aktuell notwendige Verhalten des Interfaces
- ▶ Für jeden Test muss ein Mock entsprechend konfiguriert werden
 - ▶ Großer Aufwand für viele Tests

Verhalten eines Mock Objekts

- ▶ Mocks beeinflussen alle Phasen eines Tests
 - ▶ Arrange
 - ▶ Act
 - ▶ Assert
- ▶ Mocking Frameworks vereinfachen die Verwendung von Mocks
 - ▶ Easymock
 - ▶ mockito
 - ▶ PowerMock
 - ▶ JMockit

Verhalten eines Mock Objekts

- ▶ Mocks beeinflussen alle Phasen eines Tests
 - ▶ Arrange: Konfiguration der Mocks (Configure)
 - ▶ Act: Verwendung der Mocks
 - ▶ Assert: Überprüfen der Mocks (Verify)
- ▶ Mocking Frameworks vereinfachen die Verwendung von Mocks
 - ▶ Easymock
 - ▶ mockito
 - ▶ PowerMock
 - ▶ JMockit

Einfaches Beispiel

- ▶ Ein externes Gerät wird durch eine Software gesteuert und konfiguriert
 - ▶ Die externe Hardware verlangsamt den Test
 - ▶ Das Gerät steht nicht gleichzeitig jedem Entwickler zur Verfügung
- ▶ Die Hardware soll nicht getestet werden, nur die Komponente zur Ansteuerung
- ▶ Das Gerät wird durch ein Interface abgebildet

```
public interface Device {  
    List<Integer> readLastDistances();  
    void scanDistance();  
}
```


Einfacher Test mit Mocks und mockito

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);  
        List<Integer> distances = asList(20, 10);  
        when(device.readLastDistances()).thenReturn(distances);  
  
        DistanceMonitor monitor = new DistanceMonitor(device);  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();  
        verify(device).readLastDistances();  
    }  
}
```

Einfacher Test mit Mocks und mockito

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);           ← Mock erstellen  
        List<Integer> distances = asList(20, 10);  
        when(device.readLastDistances()).thenReturn(distances);  
  
        DistanceMonitor monitor = new DistanceMonitor(device);  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();  
        verify(device).readLastDistances();  
    }  
}
```

Einfacher Test mit Mocks und mockito

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);           ← Mock erstellen  
        List<Integer> distances = asList(20, 10);      ← Konfiguration  
        when(device.readLastDistances()).thenReturn(distances); ←  
  
        DistanceMonitor monitor = new DistanceMonitor(device);  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();  
        verify(device).readLastDistances();  
    }  
}
```

Einfacher Test mit Mocks und mockito

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);           ← Mock erstellen  
        List<Integer> distances = asList(20, 10);      ← Konfiguration  
        when(device.readLastDistances()).thenReturn(distances); ←  
  
        DistanceMonitor monitor = new DistanceMonitor(device); ←  
        Integer average = monitor.average();           ← Verwendung  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();  
        verify(device).readLastDistances();  
    }  
}
```

Einfacher Test mit Mocks und mockito

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);           ← Mock erstellen  
        List<Integer> distances = asList(20, 10);      ← Konfiguration  
        when(device.readLastDistances()).thenReturn(distances); ←  
  
        DistanceMonitor monitor = new DistanceMonitor(device); ←  
        Integer average = monitor.average();           ← Verwendung  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();                 ← Überprüfung  
        verify(device).readLastDistances();  
    }  
}
```

Analyse des Tests

- ▶ Ein Mock Objekt muss erstellt werden
- ▶ Es muss nicht notwendigerweise konfiguriert werden
 - ▶ Nicht konfigurierte Methodenaufrufe liefern Standardwerte zurück
 - ▶ z.B. `false` für boolean, `null` für Object
- ▶ Solange ein Mock nicht konfiguriert wird, kann es verwendet werden
 - ▶ Automatisches Umschalten von Konfiguration zu Verwendung
- ▶ Es muss nicht alles über `verify` geprüft werden

Normalform eines Tests mit Mocks

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);  
        List<Integer> distances = asList(20, 10);  
        when(device.readLastDistances()).thenReturn(distances);  
        DistanceMonitor monitor = new DistanceMonitor(device);  
  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        verify(device).scanDistance();  
        verify(device).readLastDistances();  
    }  
}
```

Normalform eines Tests mit Mocks

```
public class DistanceMonitorTest {
```

```
    @Test
```

```
    public void calculatesAverage() throws Exception {
```

```
        Device device = mock(Device.class);           Konfiguration
        List<Integer> distances = asList(20, 10);
        when(device.readLastDistances()).thenReturn(distances);
        DistanceMonitor monitor = new DistanceMonitor(device);
```

```
        Integer average = monitor.average();           Ausführung
```

```
        assertThat(average, is(15));                   Verifikation
        verify(device).scanDistance();
        verify(device).readLastDistances();
```

```
    }
```


```
}
```


Reihenfolge von Aufrufen

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);  
        InOrder inOrder = inOrder(device);  
        List<Integer> distances = Arrays.asList(20, 10);  
        when(device.readLastDistances()).thenReturn(distances);  
  
        DistanceMonitor monitor = new DistanceMonitor(device);  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        inOrder.verify(device).scanDistance();  
        inOrder.verify(device).readLastDistances();  
    }  
}
```

Reihenfolge von Aufrufen

```
public class DistanceMonitorTest {  
  
    @Test  
    public void calculatesAverage() throws Exception {  
        Device device = mock(Device.class);  
        InOrder inOrder = inOrder(device);  
        List<Integer> distances = Arrays.asList(20, 10);  
        when(device.readLastDistances()).thenReturn(distances);  
  
        DistanceMonitor monitor = new DistanceMonitor(device);  
        Integer average = monitor.average();  
  
        assertThat(average, is(15));  
  
        inOrder.verify(device).scanDistance();  
        inOrder.verify(device).readLastDistances();  
    }  
}
```



Schwierigkeiten von Mocks

- ▶ Verwendung von statischen Methoden macht „Mocking“ schwieriger
 - ▶ Der Einsatz von Dependency Injection erleichtert Mocking
- ▶ Tiefe Abhängigkeiten erschweren die Verwendung von Mocks
 - ▶ Lose Kopplung verringert den Aufwand zur Konfiguration von Mocks

Zusammenfassung Mocks

- ▶ Mocks sind eine einfache Möglichkeit teure Abhängigkeiten in Tests zu meistern
- ▶ Frameworks machen das Testen mit Mocks einfach und komfortabel
 - ▶ Verhalten der Mocks kann direkt bei jedem einzelnen Test konfiguriert werden
- ▶ Vorsicht vor dem Testen von reinem Mock-Verhalten

Wie können Tests getestet werden

- ▶ Testabdeckung für den Code messen
- ▶ Temporär Probleme in den Produktivcode einbauen
 - ▶ Richtige Unit Tests finden den Fehler
 - ▶ Vergleichbar mit der Vorgehensweise beim Auftreten eines echten Bugs
- ▶ Software einsetzen, die zufällig oder geplant den Produktivcode ändert
 - ▶ Jester – bereits seit 10 Jahren nicht mehr gepflegt
 - ▶ Mutation Testing – wird gerade „gehypt“

Testabdeckung oder Code Coverage

- ▶ Coverage misst wie viel Code während dem Test durchlaufen wurde
- ▶ Es gibt mehrere Arten von Coverage
 - ▶ Branch Coverage
 - ▶ Line Coverage oder Statement Coverage
- ▶ In manchen Bereichen ist die Messung von Code Coverage durch Normen vorgegeben

Branch Coverage

- ▶ Branch Coverage misst die Anzahl an durchlaufenen Pfaden

```
public class SomeTest {  
    @Test  
    public void hasAnswerOnEverything() {  
        assertThat(new Some().thing(true), is(42));  
    }  
}  
  
public class Some {  
    public int thing(boolean mode) {  
        if (mode) {  
            return 42;  
        }  
        return 17;  
    }  
}
```

Branch Coverage

- ▶ Branch Coverage misst die Anzahl an durchlaufenen Pfaden

```
public class SomeTest {  
    @Test  
    public void hasAnswerOnEverything() {  
        assertThat(new Some().thing(true), is(42));  
    }  
}
```

```
public class Some {  
    public int thing(boolean mode) {  
        if (mode) {  
            return 42;  
        }  
        return 17;  
    }  
}
```

← 1 von 2 Pfaden ⇒ 50% Branch Coverage

Line Coverage

- ▶ Line Coverage misst die Anzahl an durchlaufenen Quellcode-Zeilen

```
public class SomeTest {  
    @Test  
    public void hasAnswerOnEverything() {  
        assertThat(new Some().thing(true), is(42));  
    }  
}
```

```
public class Some {  
    public int thing(boolean mode) {  
        if (mode) {  
            return 42;  
        }  
        return 17;  
    }  
}
```

Line Coverage

- ▶ Line Coverage misst die Anzahl an durchlaufenen Quellcode-Zeilen

```
public class SomeTest {  
    @Test  
    public void hasAnswerOnEverything() {  
        assertThat(new Some().thing(true), is(42));  
    }  
}
```

```
public class Some {  
    public int thing(boolean mode) {  
        if (mode) {  
            return 42;  
        }  
        return 17;  
    }  
}
```

← 2 von 3 Zeilen \Rightarrow 66% Line Coverage

Aussagekraft von Code Coverage

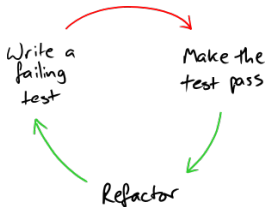
- ▶ Branch und Line Coverage machen unterschiedliche Aussagen
 - ▶ Es muss angegeben werden, wie Code Coverage gemessen wurde
- ▶ Coverage sagt nichts über die korrekte Funktionalität aus
 - ▶ Fehlende Assertions führen zur gleichen Coverage
 - ▶ Coverage gibt nur an, welcher Code durchlaufen wurde
- ▶ Bereich ohne Coverage deutet auf potentiell problematische Bereiche hin

Wann werden Tests geschrieben

- ▶ Klassisches Vorgehen bei der Entwicklung von Software
 - ▶ Neue Funktion oder Erweiterung planen
 - ▶ Funktion programmieren
 - ▶ Funktion lesbarer machen (Refactoring)
 - ▶ Funktion testen
 - ▶ Fehler beheben

Ein neuer Ansatz: Test First

- ▶ Umkehren der Reihenfolge von Entwicklung und Testen
 - ▶ Neue Funktion oder Erweiterung planen
 - ▶ Tests für neue Funktion schreiben
 - ▶ Funktion programmieren
 - ▶ Funktion lesbarer machen (Refactoring)
 - ▶ Fehler beheben



Test Driven Development (TDD)

- ▶ TDD ist eine Erweiterung von Test First
- ▶ Tests werden in kleineren Schritten entwickelt
 - ▶ Test nur soviel weiter entwickeln, dass er fehlschlägt
- ▶ Funktionen nur minimal ändern, dass Tests wieder erfüllt sind
 - ▶ „red bar green bar“ Prinzip

Kerngedanke von Test First und TDD

- ▶ TDD stellt den Test über den Produktivcode
 - ▶ Test ist wichtiger als Funktionalität
- ▶ Sobald der Produktivcode den Test erfüllt, ist die Entwicklung „abgeschlossen“
 - ▶ Führt zu minimal notwendigem Produktivcode
- ▶ Test dient als Wegweiser bei der Entwicklung des Produktivcodes
 - ▶ Entwickler nutzt seine eigene API
 - ▶ API wird angenehmer verwendbar

Nachteile von Test First und TDD

- Testen wird zur Pflicht
- Ungewohntes vorgehen für viele Entwickler
 - Einarbeitung und Umdenken im Kopf notwendig
- Der Aufwand zur Implementierung wird höher
- Nicht in allen Situationen kann TDD einfach angewandt werden
 - Die Kombination mit Legacy Systemen ist schwieriger, aber möglich

Vorteile von Test First und TDD

- + Testen wird zur Pflicht
 - + Volle Testabdeckung (Coverage 100%)
- + Fehlerrate in der Software sinkt
- + Angst vor dem Zerstören existierender Funktionen sinkt
 - + Sicherheitsnetz beim Refactoring
- + Automatische Spezifikation/Dokumentation
- + Tendentiell weniger Produktivcode
 - + Kleinere Komponenten
 - + Stabile Implementierung

Beispiel zu Test First

- ▶ Konvertierung von arabischen Zahlen zu römischen
- ▶ Römische Zahlensystem enthält keine „0“

Zeichen	I	V	X	L	C	D	M
Wert	1	5	10	50	100	500	1000

- ▶ Zahlen dazwischen werden kombiniert
 - ▶ 2 \Rightarrow II
 - ▶ 3 \Rightarrow III
 - ▶ 4 \Rightarrow IV

Beispiel zu Test First

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        return "I";  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        return "I";  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        if (2 == arabic) {  
            return "II";  
        }  
        return "I";  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        if (2 == arabic) {  
            return "II";  
        }  
        return "I";  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
  
    @Test  
    public void three() {  
        assertThat(RomanNumeral.of(3), is("III"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        if (3 == arabic) {  
            return "III";  
        }  
        if (2 == arabic) {  
            return "II";  
        }  
        return "I";  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
  
    @Test  
    public void three() {  
        assertThat(RomanNumeral.of(3), is("III"));  
    }  
}
```


Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        StringBuilder roman = new StringBuilder();  
        for (int i = 0; i < arabic; i++) {  
            roman.append("I");  
        }  
        return roman.toString();  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
  
    @Test  
    public void three() {  
        assertThat(RomanNumeral.of(3), is("III"));  
    }  
}
```

Beispiel zu Test First

```
public class RomanNumeral {  
    public static String of(int arabic) {  
        StringBuilder roman = new StringBuilder();  
        for (int i = 0; i < arabic; i++) {  
            roman.append("I");  
        }  
        return roman.toString();  
    }  
}
```

```
public class RomanNumeralTest {  
    @Test  
    public void one() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void two() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
  
    @Test  
    public void three() {  
        assertThat(RomanNumeral.of(3), is("III"));  
    }  
  
    @Test  
    public void four() {  
        assertThat(RomanNumeral.of(4), is("IV"));  
    }  
}
```

Freiwillige Hausaufgabe zu Test First

- ▶ Römische Zahlen weiterentwickeln
- ▶ Entwicklung einer Funktion zur Berechnung der Fibonacci Folge

Eingabe	0	1	2	3	4	5	6
Ausgabe	0	1	1	2	3	5	8

- ▶ Anfangen mit einem fehlschlagenden Test
- ▶ So lange den TDD Zyklus iterieren, bis die Funktion komplett ist

Weiteres zu Test First und TDD

- Zur Übung von Test First und TDD existieren weitere Übungen (Kata)

`http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsKata`

`http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata`

Umgang mit bestehendem Code

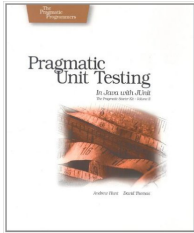
- ▶ Bestehender und ungetesteter Code (Legacy Code) ist oft nur schwer testbar
- ▶ Legacy Code wurde nicht für Tests mit Unit Tests entwickelt
 - ▶ Daher enthält er oft viele Abhängigkeiten
 - ▶ Abhängigkeiten sind oft nur schwer durch Mocks zu ersetzen
- ▶ Ungetesteter Code kann schrittweise weiterentwickelt werden
 - ▶ Die bestehende Ausgabe über Tests absichern
 - ▶ Anschließend Teile schrittweise auslagern und isoliert testen

Testen auf der grünen Wiese

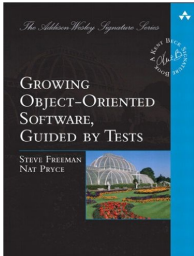
- ▶ Neue Projekte werden von Anfang an testbar entwickelt
- ▶ Die Entwicklung eines Features mit einem „Ende-zu-Ende“ Test beginnen
- ▶ Die einzelnen Schritte mit Hilfe von Unit Tests entwickeln
- ▶ Der Ende-zu-Ende Test zeigt an, sobald das Feature fertig entwickelt ist

Was fehlt noch?

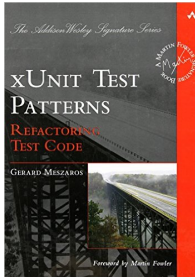
- ▶ Umgang mit paralleler Verarbeitung
 - ▶ Verhalten mehrerer Threads
- ▶ Testen von Komponenten der Benutzeroberfläche
 - ▶ Siehe UI Vorlesungen
- ▶ Test Triangulation für komplexere Berechnungen
- ▶ Zufalls- / Zeitabhängigkeit



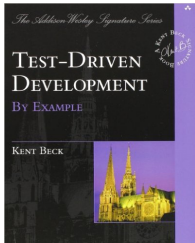
- ▶ Pragmatic Unit Testing
 - ▶ Andrew Hunt und David Thomas
 - ▶ The Pragmatic Programmers
 - ▶ ISBN: 978-0974514017



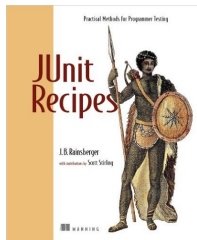
- ▶ Growing Object-Oriented Software, Guided by Tests
 - ▶ Steve Freeman und Nat Pryce
 - ▶ Addison-Wesley
 - ▶ ISBN: 978-0321503626



- ▶ xUnit Test Patterns
 - ▶ Gerard Meszaros
 - ▶ Addison-Wesley
 - ▶ ISBN: 978-0131495050



- ▶ Test Driven Development by Example
 - ▶ Kent Beck
 - ▶ Addison-Wesley
 - ▶ ISBN: 978-0321146533



- ▶ JUnit Recipes
 - ▶ J. B. Rainsberger
 - ▶ The Pragmatic Programmers
 - ▶ ISBN: 978-1932394238

Weitere Quellen

- ▶ Wiki von Ward Cunningham
 - ▶ <http://www.c2.com/cgi/wiki>
- ▶ Online Ressourcen zu xUnit Patterns
 - ▶ <http://xunitpatterns.com>
- ▶ Bildquellen
 - ▶ amazon.de
 - ▶ ebay.de
 - ▶ junit.org
 - ▶ simple-talk.com
 - ▶ wikipedia.de