

SOFTWARETESTS

Maurice Müller

2024-02

Software und Fehler

- Menschen schreiben Software
- Menschen machen Fehler

⇒ Software enthält Fehler

Bekannte Beispiele:

- Log4J (Java Library)
- Ariane 5 (Trägerrakete)
- Mars Climate Orbiter (Marssonde)

Auswirkungen von Fehlern

- jeder Fehler bindet Ressourcen
 - Zeit, Geld
 - Aufmerksamkeit, Nerven, Vertrauen
- die Kosten eines Fehlers steigen mit der Zeit seiner Existenz
 - zumindest ist das eine häufig geäußerte Theorie
- Multiplikator bei Veröffentlichung des Fehlers

Kosten von Fehlern

- durchschnittlicher Entwickler in Deutschland verdient 50-60T€ pro Jahr
- ~20% der Zeit wird mit Fehlersuche und Fehlerbehebung verbracht
- ⇒ 10 - 12T€ / Jahr / Entwickler alleine für Fehler
- zzgl. Folgekosten (s. Ariane 5 oder Mars Climate Orbiter)

Verpflichtung zum Testen

- Entwickler sind gesetzlich verpflichtet ihre Produkte zu testen
 - nicht zu testen ist **grob fahrlässig**
 - Gewährleistung gilt auch bei Software
- Tests schützen bestehende Funktionen
 - zufällige/ungewollte Änderungen werden direkt erkannt

Tests als Hilfsmittel

- Tests als Teil der Entwicklung
 - Test First
 - Test Driven Development
- Tests unterscheiden zwischen *zufälliger* und *gewollter* Funktionalität

Software ohne Tests

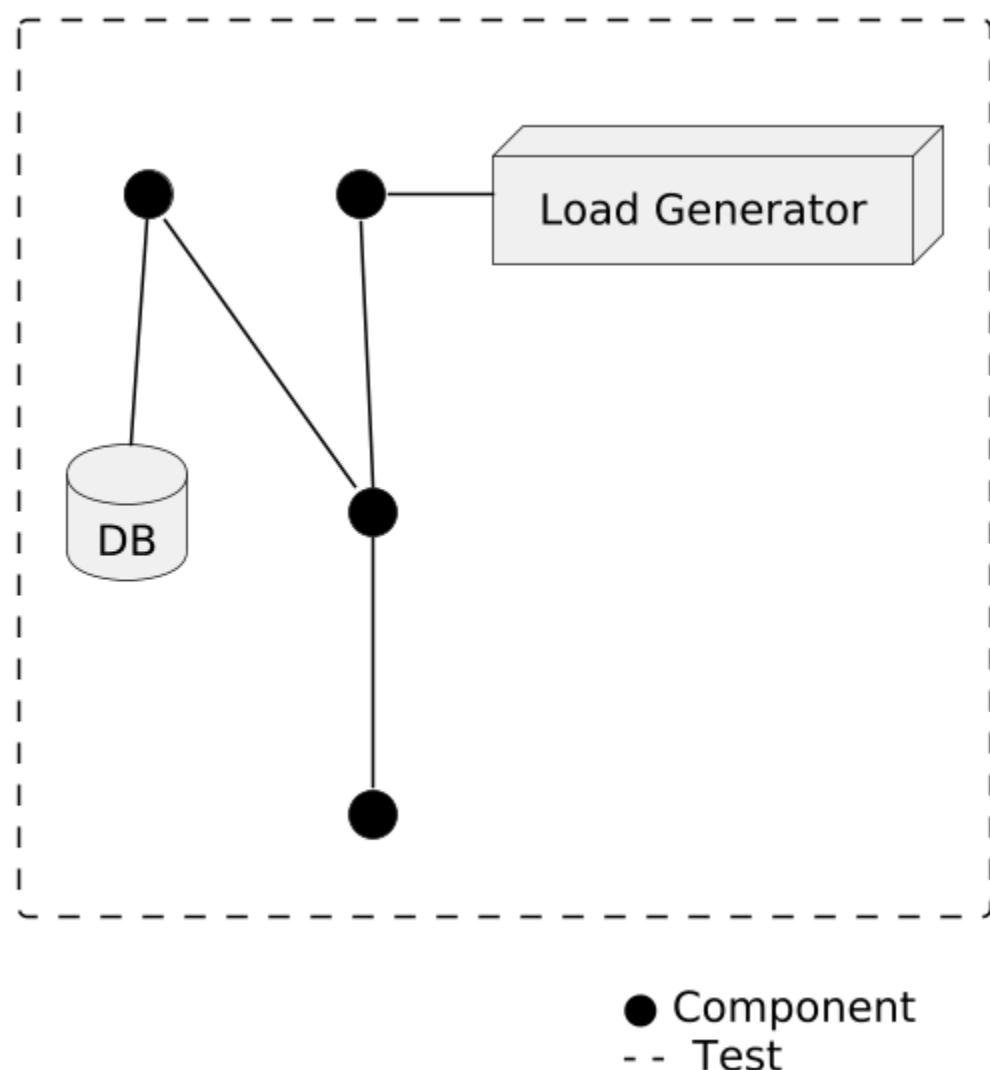


Explosion von Ariane 5, 1966 (Quelle: www.esa.int)

Testarten

- Lasttest
- Akzeptanztests
- Integrationstests
- Komponententests

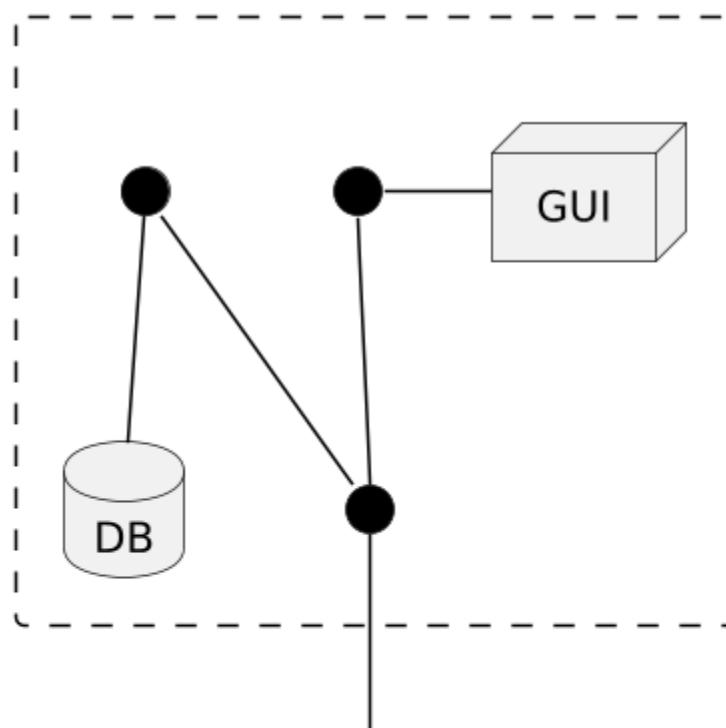
Lasttest



Lasttests

- vollständiges System wird gestartet
- echte bzw. Referenzumgebung
 - echte Datenbank, Hardware
 - (zahlreiche) simulierte Klienten/Benutzer
- parallele Bedienungsdurchführung mit Mitteln des Benutzers
- Ziel: echte Lastbedingungen erzeugen und Kenngrößen ermitteln
- *kein Ziel:* Überprüfen der Korrektheit

Akzeptanztests

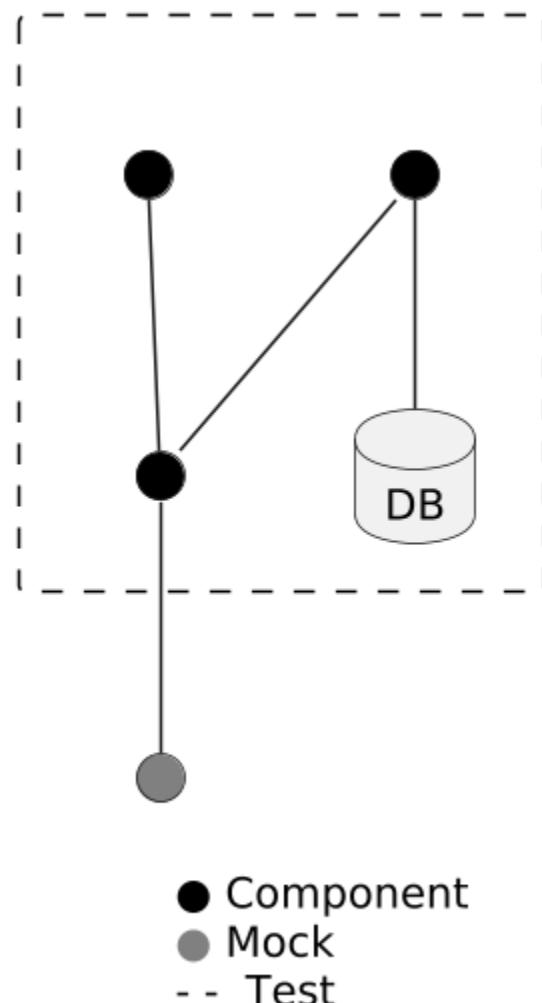


● Component
-- Test

Akzeptanztests

- vollständiges System wird gestartet
- möglichst passende Laufzeitumgebung
 - echte Datenbank, Hardware
- Durchführung mit Mitteln des Benutzers
 - z.B. Interaktion mit Bedienoberfläche
- Ziele
 - Durchspielen echter Bedienszenarien
 - Freigabe durch Auftraggeber
- *kein Ziel:* Prüfen aller möglichen Bedienwege

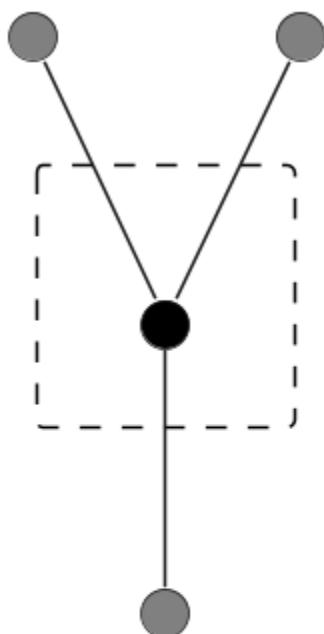
Integrationstests



Integrationstests

- nur die relevanten Teile des Systems werden gestartet
- nicht zu testende Systemteile werden mit Mocks ersetzt
- Durchführung mittels Testframework
 - Methodenaufrufe
 - Interaktion der Systemteile untereinander
- Ziel: Sicherstellung der korrekten Zusammenarbeit der Komponenten
- *kein Ziel:* Einzelheiten der Komponenten testen

Komponententests (Unit Tests)



- Component
- Mock
- Test

Komponententests (Unit Tests)

- nur der relevante Teil des Systems wird gestartet
- alle anderen Teile werden mit Mocks ersetzt
- Durchführung mittels Testframeworks
 - Methodenaufrufe
 - Prüfen der Rückgabewerte
- Ziel: korrekte Implementierung der Komponente (Unit) sicherstellen
- *kein Ziel:* Zusammenspiel oder Performanz

xUnit Frameworks

- xUnit-Frameworks haben einen vergleichbaren Aufbau
 - gleiches Konzept für Testdurchführung
- Assertion-basierte Überprüfung
 - `assertThat(actual, expected)`
 - *assertion* = Behauptung, Aussage
- Trennung zwischen Produktiv- und Testcode

xUnit-Aufbau

am Beispiel von JUnit (Java)

```
class MoneyTest { (1)

    @Test
    void testSumSameCurrency() { (2)
        String currency = "EUR";
        var result = new Money("100.00", currency)
            .plus(new Money("222.22", currency));
        assertThat(result.getAmount()).isEqualTo(new BigDecimal("322.22")); (3)
        assertThat(result.getCurrency()).isEqualTo(currency);
    }
}
```

- 1 Testklasse
- 2 Testmethode
- 3 Assertions (mind. eine sollte sein)

xUnit-Aufbau

am Beispiel von JUnit (Java)

```
class MoneyTest {  
    // Konstruktor normalerweise leer  
    public MoneyTest() {}  
  
    @Before // Vorbereitung vor jedem Testfall  
    void beforeEachTest() {}  
  
    @After // Aufräumarbeiten nach jedem Testfall  
    void afterEachTest() {}  
  
    @BeforeAll // Vorbereitung vor allen Testfällen  
    void beforeEachTest() {}  
  
    @AfterAll // Aufräumarbeiten nach allen Testfällen  
    void afterEachTest() {}  
}
```

xUnit-Aufbau

- mehrere Testmethoden pro Testklassen möglich
- alle Testmethoden sollten voneinander unabhängig sein
- Testreihenfolge ist nicht definiert / garantiert
- alle Ergebnisse werden zu einem Gesamtresultat zusammengefasst

```
class MoneyTest {  
    @Test  
    void testSumSameCurrency() {  
        String currency = "EUR";  
        var result = new Money("100.00", currency)  
            .plus(new Money("222.22", currency));  
        assertThat(result.getAmount()).isEqualTo(new BigDecimal("322.22"));  
        assertThat(result.getCurrency()).isEqualTo(currency);  
    }  
    @Test  
    void testMinusSameCurrency() {  
        String currency = "EUR";  
        var result = new Money("100.00", currency)  
            .minus(new Money("222.22", currency));  
        assertThat(result.getAmount()).isEqualTo(new BigDecimal("-122.22"));  
        assertThat(result.getCurrency()).isEqualTo(currency);  
    }  
}
```

AAA-Normalform

- AAA: Arrange, Act, Assert
- GWT: Given, When, Then

```
class MoneyTest {  
    @Test  
    void testSumSameCurrency() {  
        // ARRANGE / GIVEN  
        String currency = "EUR";  
        Money value1 = new Money("100.00", currency);  
        Money value2 = new Money("222.22", currency);  
        // ACT / WHEN  
        var result = value1.plus(value2);  
        // ASSERT / THEN  
        assertThat(result.getAmount()).isEqualTo(new BigDecimal("322.22"));  
        assertThat(result.getCurrency()).isEqualTo(currency);  
    }  
}
```

Exkurs: Datentypen für Geld

Welche Optionen gibt es, um einen Geldwert zu speichern?

- float
- double
- int (Haupteinheit), int (Centeinheit)
- BigDecimal

Float (Java)

```
float loan = 1.11f;  
float amortization = 0.4f;  
loan = loan - amortization;
```

erwartetes Ergebnis: 0.71

tatsächliches Ergebnis: 0.71000004

Double (Java)

```
double loan = 1.11;  
double amortization = 0.4;  
loan = loan - amortization;
```

erwartetes Ergebnis: 0.71

tatsächliches Ergebnis: 0.7100000000000001

Int, Int (Java)

```
int loanEuro = 1;  
int loanCent = 11;  
int amortizationEuro = 0;  
int amortizationCent = 40;  
  
loanEuro = loanEuro - amortizationEuro;  
if(amortizationCent > loanCent) {  
    loanEuro--;  
    loanCent = 100 - (amortizationCent - loanCent);  
}
```

erwartetes Ergebnis: 0 Euro 71 Cent

tatsächliches Ergebnis: 0 Euro 71 Cent

BigDecimal (Java)

```
BigDecimal loan = BigDecimal.valueOf(1.11);
BigDecimal amortization = BigDecimal.valueOf(0.40);
loan = loan.subtract(amortization);
```

erwartetes Ergebnis: 0.71

tatsächliches Ergebnis: 0.71

Float, Double

Codierung mit 4 Bytes (double 8 Bytes)

1. Vorzeichenbit (0: positiv, 1: negativ)
2. Exponent (float 8 Bit, double 11 Bit)
 - Exponent zur Basis 2
 - um negative Exponenten zu ermöglichen, wird ein Bias addiert (127 bzw. 1023)
 - Rest: Mantisse (Zahlenwert)
 - die Mantisse wird implizit mit einer '1 , ' ergänzt (die nicht codiert ist) und enthält daher immer nur den *Bruchteil*
 - $\Rightarrow (-1)^{\text{Vorzeichenbit}} \cdot 2^{\text{Exponent} - \text{Bias}} \cdot (1, \text{Mantisse})$

Binäre Brüche

6,25

- 6
 - $6 / 2 = 3 \Rightarrow 0b$
 - $3 / 2 = 1,5 \Rightarrow 1b$
 - $1 / 2 = 0,5 \Rightarrow 1b$
 - $\Rightarrow 6 = 110b$
- 0,25
 - $0,25 * 2 = 0,5 \Rightarrow 0b$
 - $0,5 * 2 = 1 \Rightarrow 1b$
 - $\Rightarrow 0,25 = 0,01b$

$\Rightarrow 6,25 = 110,01b$

Float Beispiel

- $\Rightarrow (-1)^{\text{Vorzeichenbit}} * 2^{\text{(Exponent} - 127)} * (\text{1,Mantisse})$
- Codierung von 0,25
 - $1b * 2^{-2}b * 1,0b = 0,01b = 0,25$
 - $(-1)^0 * 2^{(125-127)} * 1,0$
 - $\Rightarrow 0 \ 0111 \ 1101 \ 00000000000000000000000000000000$

Binäre Brüche

$0,1$

- $0,1 * 2 = 0,2 \Rightarrow 0b$
- $0,2 * 2 = 0,4 \Rightarrow 0b$
- $0,4 * 2 = 0,8 \Rightarrow 0b$
- $0,8 * 2 = 1,6 \Rightarrow 1b$
- $0,6 * 2 = 1,2 \Rightarrow 1b$
- $0,2 * 2 = 0,4 \Rightarrow 0b$
- ...

Float Beispiel

- $\Rightarrow (-1)^{\text{Vorzeichenbit}} \cdot 2^{(\text{Exponent} - 127)} \cdot (1, \text{Mantisse})$
- Codierung von 0,1
 - 0 0111 1011 1001 1001 1001 1001 1001 1001 101
 - $(-1)^0 \cdot 2^{(123-127)} \cdot$
1,10011001100110011001101b
 - 1b * 2^{-4} b * 1,1001100110011001101b
 - = 0,00011001100110011001101b ≈
0.10000001490116119384765625

Ergebnisse eines Unit-Tests

- **Success:** Bestanden
 - Testmethode läuft durch
 - alle Assertions sind erfüllt
 - eine leere Testmethode besteht immer
- **Failure:** Assertion nicht bestanden
 - mind. eine Assertion in der Testmethode schlägt fehl
- **Error:** Error/Exception
 - in der Testmethode tritt eine Exception auf
 - gewollte Exception können geprüft werden

A-TRIP

5 Regeln für gute Unit-Tests

- **Automatic** (automatisch)
- **Thorough** (gründlich)
- **Repeatable** (wiederholbar)
- **Independent** (unabhängig)
- **Professional** (professionell)

Automatic

- Tests müssen automatisch ablaufen
 - d.h., kein manuelles Eingreifen ist notwendig
- Tests müssen ihre Ergebnisse selbst überprüfen
 - für jeden Test nur das Ergebnis *bestanden* oder *nicht bestanden* zulässig
 - *nicht bestanden* wird als *gebrochen* / *failed* bezeichnet
- wenn möglich, in eine CI/CD-Pipeline einbinden

Automatic: Gegenbeispiel

```
@Test  
void testSum() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.println("Enter a number1:");  
    int number1 = scanner.nextInt();  
    System.out.println("Enter a number2:");  
    int number2 = scanner.nextInt();  
    int result = Calculator.sum(number1, number2);  
    println(number1 + "+" + number2 + "=" + result);  
}
```

- manuelle Eingaben sind notwendig
- der Test überprüft sich nicht selbst

Automatic: Beispiel

```
@Test  
void testSum() {  
    int number1 = 10;  
    int number2 = 30;  
    int result = Calculator.sum(number1, number2);  
    assertThat(result).isEqualTo(40);  
}
```

Thorough

- alles *Notwendige* wird geprüft
 - *notwendig* ergibt sich aus den Rahmenbedingungen
 - Randfälle berücksichtigen
- iteratives Vorgehen
 - **jede missionskritische Funktionalität muss getestet sein**
 - beim Auftreten eines Fehlers wird ein Test geschrieben, der ein erneutes Auftreten verhindert
- Fehler sind nicht gleichmäßig über den Code verteilt
 - Fehler häufen sich (**Lokalitätsprinzip**)
 - zusätzliche Tests im *Umfeld* eines Fehlers können sinnvoll sein

Thorough: Gegenbeispiel

```
class SumTest {  
    @Test  
    void testSum() {  
        assertThat(Calculator.sum(10, 30))  
            .isEqualTo(40);  
    }  
}
```

- keine Berücksichtigung von negativen Zahlen
- keine Berücksichtigung der Randfälle

Thorough: Beispiel

```
class SumTest {  
    void testPositive() {  
        assertThat(Calculator.sum(10, 30))  
            .isEqualTo(40);  
    }  
    void testZero() {  
        assertThat(Calculator.sum(0, 0))  
            .isEqualTo(0);  
    }  
    void testNegative() {  
        assertThat(Calculator.sum(-5, -4))  
            .isEqualTo(-9);  
    }  
    void testOverflow() {  
        assertThat(() -> Calculator.sum(Integer.MAX_VALUE, 1))  
            .throwsException(OverflowException.class);  
    }  
}
```

Repeatable

- Test muss beliebig wiederholbar sein
- Test muss immer das gleiche Ergebnis liefern
 - häufige Fehlerquellen: Datum, Zufall und Dateisystemzugriffe (OS-abhängig)
- ein spontan fehlschlagender Test ist selbst fehlerhaft
 - fehlerhafter Test kann schlechter sein als kein Test
 - Tests müssen zuverlässig funktionieren, sonst keine Absicherung

Repeatable: Gegenbeispiel

```
@Test  
public void testAgeCalculation() {  
    User user = new User()  
        .withBirthday(LocalDate.of(1990, 1, 1));  
    assertEquals(34, user.getAge());  
}
```

- 2024 stimmt das Ergebnis, schon ein Jahr später nicht mehr

Repeatable: Beispiel

```
@Test  
public void testAgeCalculation() {  
    User user = new User()  
        .withBirthday(LocalDate.now().minusYears(34));  
    assertEquals(34, user.getAge());  
}
```

Independent

- Tests dürfen keine Abhängigkeiten zu anderen Tests haben
 - weder explizit noch implizit
- Reihenfolge der einzelnen Tests darf keinen Einfluss haben
- ⇒ wenn ein Test fehlschlägt, schlagen nicht automatisch weitere fehl
 - die eigentliche Ursache ist schneller auffindbar

Independent: Gegenbeispiel

```
@Inject  
UserRespository userRespository;  
  
@Test  
public void testSaveUser() {  
    User user = new User().withId(1111);  
    assertThat(userRespository.save(user)).isTrue();  
}  
  
@Test  
public void loadUser() {  
    User user = userRespository.byId(1111);  
    assertThat(user).isNotNull();  
}
```

- *testSaveUser* muss vor *loadUser* ausgeführt werden

Independent: Beispiel

```
@Inject
UserRespository userRespository;

@Before
public void setUpUser() {
    userRespository.clear();
    User user = new User().withId(1111);
    assertThat(userRespository.save(user)).isTrue();
}

@Test
public void loadUser() {
    User user = userRespository.byId(1111);
    assertThat(user).isNotNull();
}
```

Professional

- Testcode unterliegt gleichen Standards wie Produktivcode
 - allerdings nicht zwangsläufig Tests von Testcode
- Testcode sollte so leicht verständlich wie möglich sein
- Testcode als Selbstzweck (Anzahl Tests erhöhen) ist nicht sinnvoll
 - Tests für unrelevante Teile sind ebenfalls nicht sinnvoll

Professional: Gegenbeispiel

```
@Test  
public void tst() {  
    String s1 = "Java", s2 = "JavaScript", s3 = "Java";  
    boolean b1 = s1 == s3;  
    boolean b2 = s1.equals(s3);  
    boolean b3 = s1.equals(s2.substring(0, 4));  
    boolean b4 = s2.equals("JavaScript");  
    assertTrue(b1 && b2 && b3 && b4);  
}
```

Professional: Beispiel

```
@Test
public void testStringEquality() {
    String java1 = "Java";
    String java2 = "Java";
    String javaScript = "JavaScript";

    boolean sameReference = java1 == java2;
    boolean subStringEquality = java1.equals(javaScript.substring(0, 4));
    boolean stringEquality = java1.equals("Java");

    assertTrue(sameReference);
    assertTrue(subStringEquality);
    assertTrue(stringEquality);
}
```

Testen der Tests

Who watches the watchmen?



Bild von Dall-E generiert

Quis custodiet ipsos custodes?

- Testabdeckung (Code Coverage) für den Code messen
- temporär Code ändern ⇒ Tests müssen reagieren
- Mutation Testing
 - automatisierte Änderungen (*Mutations*) im Code, um Tests auszulösen

Code Coverage

- wieviel Code wurde während den Tests durchlaufen
- Line Coverage
 - wieviele Zeilen wurden durchlaufen
- Branch Coverage
 - wieviele Abzweigungen wurden durchlaufen

Code Coverage: Line Coverage

- entweder wird eine Zeile Code durchlaufen oder nicht
- durchlaufen bedeutet **nicht zwangsläufig** getestet

```
class MyTest {  
    @Test  
    void test() {  
        assertThat(new MyClass().calc(true, 4))  
            .isEqualTo(16);  
    }  
}  
  
class MyClass {  
    void calc(boolean mode, int value) {  
        if(mode) {  
            return value * value;  
        }  
        return value + value;  
    }  
}
```

Line Coverage von 66,66%

Code Coverage: Branch Coverage

- jede Abzweigung wird einzeln betrachtet

```
class MyTest {  
    @Test  
    void test() {  
        assertThat(new MyClass().calc(true, 4))  
            .isEqualTo(16);  
    }  
}  
  
class MyClass {  
    void calc(boolean mode, int value) {  
        if(mode) {  
            return value * value;  
        }  
        return value + value;  
    }  
}
```

Branch Coverage von 50%

Testabdeckung

- hohe Testabdeckung (>80%) ist hilfreich, aber sehr aufwendig
- Kosten/Nutzen muss im Verhältnis stehen
- eine hohe Abdeckung sagt nichts über korrekte Funktionalität aus
 - es bedeutet nur, dass viel Code durchlaufen wurden
 - nur *ungetestete* Zeilen sind eindeutig (nicht getestet)
- Bereiche ohne Abdeckung deuten auf problematische Bereiche hin

Mutation Testing

- bei jedem Durchlauf wird eine Änderung im Code eingeführt
 - z.B. if-Bedingung negiert, Konstanten geändert, Methodenaufrufe entfernt, ...
- Tests müssen die Mutation entdecken
 - Tests schlagen fehl = *Mutation killed*
 - Tests bleiben grün = *Mutation survived*

Mutation Testing: Killed

```
class MyTest {  
    @Test  
    void test() {  
        assertThat(new MyClass().calc(true, 4))  
            .isEqualTo(16);  
    }  
}  
  
class MyClass {  
    void calc(boolean mode, int value) {  
        if (!mode) { // Mutation  
            return value * value;  
        }  
        return value + value;  
    }  
}
```

Mutation Testing: Survived

```
class MyTest {  
    @Test  
    void test() {  
        assertThat(new MyClass().calc(true, 4))  
            .isEqualTo(16);  
    }  
}  
  
class MyClass {  
    void calc(boolean mode, int value) {  
        if(mode) {  
            return value * value;  
        }  
        return value - value; // Mutation  
    }  
}
```

Mutation Testing

- sinnvolle und unaufwendige Zusatzprüfung
- kann viele *False-Positive*-Meldungen haben
 - nicht den dauerhaft überlebenden Mutationen nachgehen
 - nur neue Meldungen untersuchen
- Mutation Testing ist eine langfristige Investition
 - der einmalige Einsatz scheitert an Falschmeldungen

Testabzeichen

- es gibt viele Tests ⇒ welche sind wichtig? welche sind essentiell?

Testabzeichen

- Motivation
 - **#Requirement** - sichert eine Anforderung ab
 - **#BugFix** - verhindert einen aufgetretenen Fehler
- Nutzen
 - **#Regression** - hat einen neuen Fehler verhindert
 - **#Lifesaver** - hat vor Katastrophe gerettet
- Kosten
 - **Adjusted** - Anpassungen waren notwendig
 - **Erratic** - Fehlschlag ohne erkennbaren Zusammenhang
 - **Fragile** - Fehlschlag aufgrund Nebensächlichkeiten
- Abzeichen können mehrfach vergeben werden

Testabzeichen: Beispiel

```
class MyTest {  
    // #Requirement: https://issuetracker.example/ABC-123  
    // #Regression #Regression  
    @Test  
    void test1() {}  
  
    // #BugFix: https://issuetracker.example/BUG-222  
    // #Regression #Regression  
    // #Lifesaver  
    @Test  
    void test2() {}  
  
    // #BugFix: https://issuetracker.example/BUG-299  
    // #Fragile #Fragile  
    // #Erratic  
    @Test  
    void test3() {}  
}
```

Testabzeichen

- verdeutlichen Glaubwürdigkeit
- zeigen Nutzen
- ermöglichen fundierte Entscheidungen
 - Lohnt sich der Test?
 - Warum wird er ständig angepasst?
 - Liegt der Fehler am Test oder an meiner Änderung?
- Tests sind nicht alle gleichwertig / gleich gut

Lesbarkeit

- Tests sollten möglichst zugänglich sein
 - leicht lesbar
 - einfach strukturiert
 - direkt verständlich
- Abwägen zwischen Code-Duplikation und gemeinsamer Abstraktion
 - gemeinsame Abstraktion koppelt Tests (die unabhängig sein sollten)
- ggf. Einsatz von Fluent-Interfaces
 - `assertThat(resultList).hasSize(2).anyMatch(it
→ it.age() == 32)`

Mock-Objekte

- kurz Mocks
- Stellvertreter für echte Implementierung
- ersetzen Abhängigkeiten während eines Tests
- vergleichbar mit einem Crash Test Dummy
 - hat nur die unfallrelevanten Eigenschaften

Mocks: Begriff

- **Dummy:** *Lückenfüller* - muss da sein, aber wird nicht augerufen
- **Fake:** hat für den Test eine funktionierende Implementierung
 - benutzt *Abkürzungen* z.B. In-Memory-Datenstrukturen
 - kann vorgefertigte Antworten zurückgeben
- **Mock:** wie *Fake*, aber bringt Methoden zur Evaluierung mit
- **Spy:** *Fake* oder *Mock*, der Metriken aufzeichnet
 - z.B. wie oft wurde eine Methode aufgerufen oder
 - was für Parameter wurden übergeben
- im Alltag wird häufig nur *Mock* für alles benutzt

Mocks: Einsatz

- um eine Klasse isoliert testen zu können, müssen die Abhängigkeiten ersetzt werden
- selbst implementierte Mocks können aufwendig sein
 - es gibt viele gute Mock-Tools
- immer auf das Wesentliche beschränken ⇒ Mocks müssen nur "gut genug" sein

Mocks: Beispiel

```
class UserService {  
    UserRepository userRepo;  
  
    UserService(UserRepository userRepo) {  
        this.userRepo = userRepo;  
    }  
  
    int ageOfUser(long userId) {  
        User user = userRepo.byId(userId);  
        return Period.between(  
            LocalDate.now(),  
            user.getBirthday());  
    }  
}
```

Mocks: Beispiel

```
class UserRepoFake implements UserRepository {  
    Map<Long, User> userIdToUser = new HashMap<>();  
    void addUser(User user) {  
        userIdToUser.put(user.id(), user);  
    }  
    User getById(long id) {  
        return userIdToUser.get(id);  
    }  
}
```

Mocks: Beispiel

```
import java.time.LocalDate;@Test
void testAge() {
    UserRepository userRepo = new UserRepoFake();
    userRepo.add(new User().withId(3)
        .withBirthday(LocalDate.now().minusYears(7))
    );
    UserService service = new UserService(userRepo);
    int age = service.ageOfUser(3);
    assertThat(age).isEqualTo(7);
}
```

Mocks: Voraussetzungen

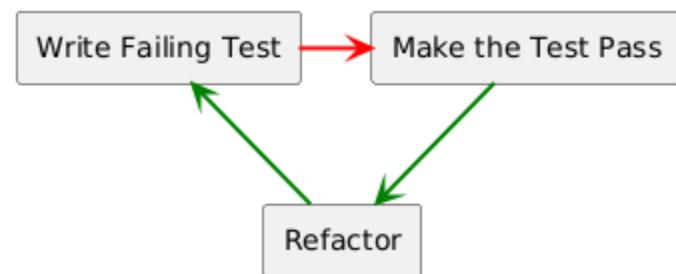
- statische Methoden (und vergleichbare Konstrukte) sind problematisch
- sinnvoller Einsatz nur, wenn Dependency Injection möglich
- tiefe Abhängigkeitsstrukturen sind aufwendig nachzubilden
 - Law Of Demeter
 - lose Kopplung erleichtert die Nutzung von Mocks

Test Driven Development (TDD)

- klassisches Vorgehen
 - Funktionalität planen
 - Funktionalität implementieren
 - Funktionalität umschreiben (Refactoring)
 - Tests schreiben
 - Fehler beheben

TDD Zyklus

- Vorgehen bei Test First
 - Funktionalität planen
 - Test schreiben
 - Funktionalität implementieren
 - Refactoring



TDD

- Test First und TDD stellen Tests über Produktivcode
- Produktivcode wird nur geschrieben, um einen Test zu erfüllen
 - minimale Implementierung ausreichend
- Tests weisen den Weg
- nach jedem Schritt kann die Implementierung verbessert werden

TDD: Vorteile

- Tests sind nicht optional
- vollständige Testabdeckung
- weniger Fehler
- automatische Spezifikation
- kleinere Komponente
- stabile Implementierung
- losere Kopplung

TDD: Nachteile

- erfordert Einarbeitung
- höherer Aufwand
- funktioniert nicht immer
- Tests sind nicht optional
- lohnt sich nicht immer

TDD: Beispiel

- Konvertierung von arabischen Zahlen zu römischen Zahlen
- römisches Zahlensystem kennt nicht die Ziffer "0"

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

- Zahlen dazwischen werden kombiniert
 - $2 \Rightarrow \text{II}$
 - $3 \Rightarrow \text{III}$
 - $4 \Rightarrow \text{IV}$

TDD: Beispiel Schritt 1.1

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1)  
            .isEqualTo("I"));  
    }  
}
```

TDD: Beispiel Schritt 1.2

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
}
```

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

TDD: Beispiel Schritt 2.1

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
}
```

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

TDD: Beispiel Schritt 2.2

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
}
```

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

TDD: Beispiel Schritt 3.1

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
    void convertThree() {  
        assertThat(RomanNumeral.of(3))  
            .isEqualTo("III");  
    }  
}
```

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

TDD: Beispiel Schritt 3.2

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
    void convertThree() {  
        assertThat(RomanNumeral.of(3))  
            .isEqualTo("III");  
    }  
}
```

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

TDD: Beispiel Schritt 3.3

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
    void convertThree() {  
        assertThat(RomanNumeral.of(3))  
            .isEqualTo("III");  
    }  
}
```

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

TDD: Beispiel Schritt 4.1

```
class Test {  
    void convertOne() {  
        assertThat(RomanNumeral.of(1))  
            .isEqualTo("I");  
    }  
    void convertTwo() {  
        assertThat(RomanNumeral.of(2))  
            .isEqualTo("II");  
    }  
    void convertThree() {  
        assertThat(RomanNumeral.of(3))  
            .isEqualTo("III");  
    }  
    void convertThree() {  
        assertThat(RomanNumeral.of(4))  
            .isEqualTo("IV");  
    }  
}
```

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

Freiwillige Hausaufgabe zu TDD

- römische Zahlen weiterentwickeln
- Entwicklung einer Funktion zur Berechnung der Fibonacci-Folge
- immer mit fehlschlagenden Tests beginnen
- so lange den TDD-Zyklus folgen, bis die Funktion komplett ist