JUnit

Was sind Unittests

Ein Unittest (zu Deutsch auch Modultest oder Komponententests) bezeichnet das Testen der einzelnen Bestandteile eines Softwaresystems auf deren korrekte Funktionsweise. Die Tests stellen die granularste Möglichkeit dar Softwarebestandteile zu testen. Damit bilden sie die Basis der Testpyramide (https://de.wikipedia.org/wiki/Modultest)
Unittests eignen sich hervorragend als Regressionstests. Die Validierungslogik ist fest programmiert und validiert Module auch bei mehrfacher Testausführung immer identisch (idempotent). Dazu gehört es aber auch, den Code so zu schreiben, dass sich dieser gut testen lässt. Das heißt, der getestete Code ist nicht von externen Faktoren wie beispielsweise Datenbanken oder einer Uhrzeit abhängig. Dabei kann es teilweise recht kompliziert sein diese Abhängigkeiten auszuklammern. Generell gilt aber: Gute Testbarkeit ist ein Indikator für hohe Code- / Softwarequalität.

Eignung

Der Einsatz von Unittests ist unbestritten eines der wichtigsten Instrumente zur Entwicklung nachhaltiger Softwaresysteme. Dennoch eignen sich Unittests nicht für jedes Szenario gleichermaßen.
Besonders gut eignen sich Tests…

- … wenn die Buildinfrastruktur ein automatisiertes Ausführen aller Tests unterstützt.
- … wenn eine Continous Integration Pipeline aufgesetzt werden soll. Automatisierte Tests sind die Grundvoraussetzung für einen automatischen Integrationsprozess (dasselbe gilt natürlich auch für Continous Deployment).

Eher ungeeignet sind Unittests bei…

- … Systemen mit einer sehr hohen Schnittstellenanzahl.
- … Systemen mit Schnittstellen zur Interaktion mit analogen Schnittstellen (grafische Benutzeroberfläche, Funkübertragung, …),
- … systemübergreifenden Testfällen.
- … Modulen deren Funktionsweise sich häufig grundlegend ändert.
- … Systemen an denen wenig Änderungen durchgeführt werden.

Vor- und Nachteile

Durch die Nutzung von Unittests ergeben sich im Rahmen eines Entwicklungsprojektes unter anderem die folgenden Vorteile:

- Der Entwickler erhält bereits sehr früh Feedback über neue Funktionen. Dabei gilt je früher Fehler gefunden werden desto günstiger ist ihre Korrektur.
- Mit zunehmender Testabdeckung beschleunigt sich die Entwicklung eines Softwaresystems. Dadurch das Fehler bereits in der Entwicklung auffallen haben entsprechende Systeme im Rahmen eines manuellen Tests bereits eine deutlich höhere Qualität und Stabilität. Es werden weniger Fehler gefunden was zu einer Reduktion der manuellen Test- und Abspracheaufwände führt.
- Das Schreiben von Tests erzwingt das Präzessieren von Anforderungen. Softwaretests interpretieren keine Antworten sie erlauben keine Abweichung zwischen Erwartetem- und Erwartungswert. Dadurch tauchen auch konzeptionelle Probleme im Projekt frühzeitig auf (Fail Fast). Zu Beginn eines Projektes ist dann meist noch genug Zeit um diese Probleme zu umgehen.
- Die Toolunterstützung für UnitTests ist mittlerweile technologieübergreifend sehr gut (JUnit, PHPUnit, JSUnit, NUnit, CppUnit, …).
- Unittests sind einfach auszuführen. Sie laufen überall und benötigen keine Vorbereitung.

Den vielen positiven Effekten stehen natürlich einige Aufwände bzw. Prämissen entgegen.

- Es ist nicht immer einfach UnitTests zu schreiben. Insbesondere bei älteren Systemen ist das nachträgliche ergänzen von Unittests mit nicht unerheblichen Aufwänden verbunden. Oft ist eine Restrukturierung des Systems erforderlich. Dies ist eine strategische Investition und rechnet sich nur dann, wenn sich das System auch zukünftig weiterentwickelt.
- Ändert sich die Funktionalität eines Systems müssen UnitTests mitgepflegt werden. Diese Aufwände steigen proportional zur Testabdeckung. Dabei ist es wichtig nicht nur die Compilefehler der Tests zu korrigieren, sondern diese auch inhaltlich zu überprüfen.
- Das Schreiben des ersten Tests erfordert meist initiale Aufwände. Je später Tests in einem Softwareentwicklungsprojekt eingeführt werden desto aufwendiger ist deren Integration. Der Aufwand sinkt mit steigender Anzahl an Tests deutlich.
- Das Schreiben richtiger UnitTests (so dass sie auch einen wirklichen Nutzen bringen) erfordert eine gewisse Erfahrung und ist nicht immer einfach.
- Nicht alle Codestellen sind für automatisierte Tests geeignet. Der Entwickler muss im Einzelfall Aufwand und Nutzen gegeneinander abwägen.

Was ist JUnit

JUnit ist das mit Abstand am weitesten verbreitete Framework zur Erstellung von Tests für Java-Programme. Es fokussiert die Erstellung von automatisierten Modultests für Java-Methoden. Das Framework wurde ursprünglich von Erich Gamme und Kent Beck entwickelt und orientierte sich an SUnit für Smalltalk.[1]
JUnit Logo

Mittlerweile gibt es auch Adaptionen in anderen Programmiersprachen, wie z.B. PHPUnit oder CppUnit. Auch ergänzen verschiedene Erweiterungen die sehr einfache Funktionalität des Frameworks. So löst z.B. die Erweiterung Mockito das wohl größte Problem von Junit, dem schreiben echter Modultests. Dabei gilt es nur die Funktionalität einzelner Module zu testen und nicht die von indirekt verwendeter Schnittstellen oder Datenquellen. Es tauscht diese durch Simulatoren aus die bei jeder Testausführung einheitlich reagieren, was das gezielte, saubere Testen von Modulen erlaubt.

Was ist ein Test?

Es gibt verschiedene Wege die Funktionsweise des eigenen Codes zu überprüfen. Der einfachste Weg ist die Nutzung des IDE Debuggers. Er erlaubt es zur Laufzeit Ausdrücke zu überprüfen, Variablen zu verändern und den Programmablauf zu beobachten.
Eine andere Alternative besteht in der Ausgabe von Variablen und Zuständen der Anwendung auf den Standard Output Stream (`System.out.println("Wert von a: " + a.toString())`).
Beide Varianten sind eher für zur Analyse eines Fehlers geeignet als zur regelmäßigen Überprüfung des Systemverhaltens. Sie erfordern die manuelle Analyse und Bewertung durch einen Entwickler und sind deswegen sehr aufwändig.
JUnit Tests sind programmierte Tests für Java die kein menschliches Eingreifen erfordern und einfach auszuführen sind.
JUnit Tests sind einfache Java Klassen. Sie gruppieren Tests zu einer bestimmten Komponente. Im Optimalfall haben Tests das Suffix Test und denselben Namen wie die zu testenden Klasse.

Die einzelnen Tests einer Klasse sind durch die Annotation `@org.junit.Test` markiert. Sie sollten voneinander unabhängig lauffähig sein, da keine Möglichkeit besteht die Ablaufreihenfolge zu beeinflussen. Um das zu gewährleisten ist es wichtig die Testdaten vor jedem Test neu zu initialisieren. Nur so ist eine Reproduzierbarkeit gegeben.
Tests bestehen im Wesentlichen aus drei Bestandteilen:

1. Testvorbereitung: In dieser Phase werden die zur Testdurchführung benötigten Daten erzeugt.

2. Funktionsaufruf: Der zu testende Funktionsbaustein wird mit den Testdaten aufgerufen und das Verarbeitungsergebnis wird ausgelesen.

3. Ergebnisvalidierung: Das Ergebnis des Funktionsbausteins wird validiert. Alternativ kann es je nach geprüfter Funktion auch interessant den Funktionsablauf zu nachträglich zu prüfen.

Ein einfaches Beispiel

Das folgende Codefragment zeigt einen Einfachen Test mit JUnit. Zunächst wird dort die zu testende Klasse initialisiert und anschließend eine Verarbeitung aufgerufen und das Ergebnis in einer Variable abgelegt. Im letzten Schritt lässt sich mittels Hilfsmethoden das Ergebnis validieren und im Falle einer Abweichung eine entsprechende Fehlermeldung darstellen.

```
@Test
public void add(){
    Calculator calc = new Calculator();

    int result = calc.add(2, 2);

    assertEquals("Addition ermittelt falsches Ergebnis.", 4, result);
}
```

Die Methoden zur Validierung stehen zum statischen Import von `import static org.junit.Assert.*` zur Verfügung. Neben `assertEquals` stehen weitere Methoden wie `assertTrue`, `assertFals`, `assertArrayEquals`, `assertNull`, etc. zur Verfügung.

Die Prüfungen sind dabei alle ähnlich aufgebaut:

- Meldung im Fall einer abweichung (optional);

- Erwartungswert (falls nicht bereits über Prüfmethode festgelegt).

- Der zu validierende Wert.

Testvor- und Nachbearbeitung

Häufig sind mehrere vorbereitende Schritte vor der Durchführung eines einzelnen Tests notwendig. Diese Maßnahmen sind oft deutlich aufwändiger als die eigentlichen Tests. Gleichzeitig ähnelt sich diese Testvorbereitung zwischen verschiedenen Tests stark.
Diese Problemstellung wirkt JUnit über die Bereitstellung von eigener Annotationen entgegen.
Über Methoden die mit `@org.junit.Before` annotiert sind, werden vor der Durchführung jedes Tests ausgeführt. So lässt sich das initialisieren von Variablen in einem Test zentralisieren.

KasseTest.java

```
package de.cherriz.training.test.kasse;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

public class KasseTest {

    private Kasse kasse;

    @Before
    public void setUp() {
        kasse = new Kasse();
        kasse.einzahlen(new Betrag(50, 0, false), new Betrag(24, 99,
false));
        kasse.einzahlen(new Betrag(50, 0, false), new Betrag(24, 99,
false));
        kasse.einzahlen(new Betrag(10, 0, false), new Betrag(9, 99,
false));
    }

    @Test
    public void kassensturz() {
        assertEquals("Kasseninhalt wird falsch berechnet.", 5997,
kasse.kassensturz());
    }

    @Test
    public void allesEntnehmen() {
        Betrag inhalt = kasse.allesEntnehmen();
        assertEquals("Eurowert falsch berechnet.", 59, inhalt.getEuro());
        assertEquals("Centwert falsch berechnet.", 97, inhalt.getCent());
        assertFalse("Abhebung darf nicht negativ sein.",
inhalt.isNegativ());
    }}
```

Ist die Testvorbereitung einmal implementiert lassen sich weitere Tests vergleichsweise leicht ergänzen.

Aufgaben:

1. Bewertet den folgenden Testfall. Erfüllt dieser die Anforderungen an einen Unit-Test?

```
@Test
public void kurzNamen(){
    List<String> namen = new LinkedList<>();
    namen.add("Anja");
    namen.add("Susanne");
    namen.add("Heidi");
    namen.add("Uwe");
    namen.add("Markus");

    List<String> kurzeNamen = namen.stream().filter(n -> n.length() <=
4).collect(Collectors.toList());

    assertEquals("Anja", kurzeNamen.get(0));
    assertEquals("Uwe", kurzeNamen.get(1));
}
```

2. Bewertet den folgenden Testfall. Erfüllt dieser die Anforderungen an einen Unit-Test?

```
@Test
public void networkAvailable() throws IOException {
    boolean reachable =
InetAddress.getByName("http://www.google.de").isReachable(100);

    assertTrue("Google ist nicht erreichbar.", reachable);
}
```

3. Bewertet den folgenden Testfall. Erfüllt dieser die Anforderungen an einen Unit-Test?

```
@Test
public void arbeitstag() throws IOException {
    LocalDate date = LocalDate.of(2018, 3, 12);

    DayOfWeek dayOfWeek = date.getDayOfWeek();
    boolean arbeitstag = dayOfWeek.getValue() < 6;

    assertTrue("Datum nicht als Arbeitstag erkannt.",
arbeitstag);
}
```

4. Überlegt euch für die folgende Klasse sinnvolle Testfälle.

Bibliothek.java

```
package de.cherriz.training.test.bibliothek;

import java.util.LinkedList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class Bibliothek {

    private List<Buch> buecher = new LinkedList<>();

    public void registrieren(Buch buch) {
        this.buecher.add(buch);
    }

    public Buch leihen(String iban, Kunde kunde) {
        Optional<Buch> buch = this.buecher.stream().filter(b ->
b.getIban().equals(iban)).filter(b -> b.getLeihe() == null).findFirst();
        if (buch.isPresent()) {
            Buch buchgeliehen = buch.get();
            buchgeliehen.setLeihe(new Leihe(kunde));
            return buchgeliehen;
        }
        return null;
    }
    public List<Buch> suchen(String titel) {
        return this.buecher.stream().filter(b ->
b.getTitel().contains(titel)).filter(b -> b.getLeihe() ==
null).collect(Collectors.toList());
    }

}
```