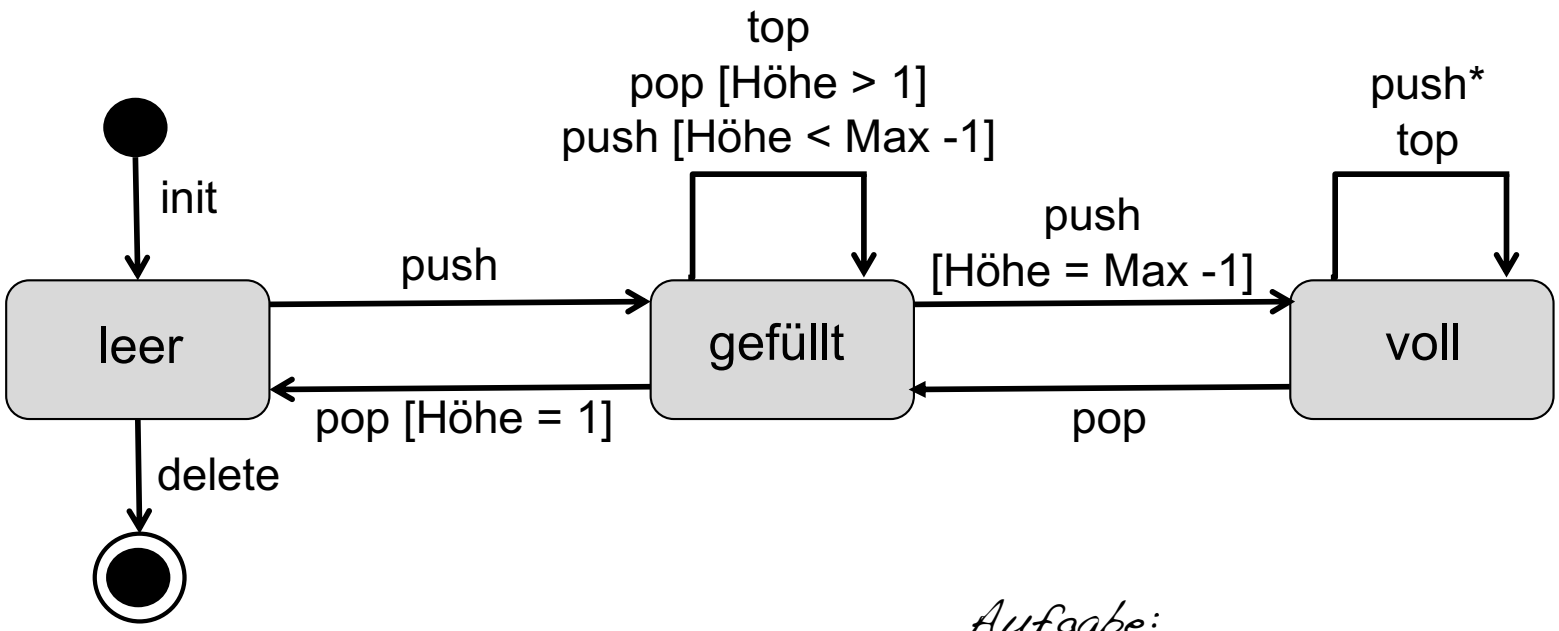
3. Zustandsbezogener Test

- Es gibt Systeme, bei denen die Ausgabe nicht nur von der aktuellen Eingabe abhängt, sondern auch vom Ablauf der vorherigen Eingaben
- Ein solches System kann sich, abhängig von der Verarbeitungshistorie, in verschiedenen Zuständen befinden
- Die Reaktion auf eine Eingabe ist abhängig vom aktuellen Zustand
- Solche Systeme können mit Zustandsautomaten modelliert werden
- Ein Zustandsdiagramm enthält
  - Zustände
  - Anfangs- und Endzustand
  - Zustandsübergänge
  - Guards (Wächter) als Bedingung für einen Zustandsübergang

- Als Beispiel modellieren wir einen Stapel (*Stack*) durch einen Zustandsautomaten

nach [SL12]



- Minimalanforderung an einen Test
  a. Testfall erreicht alle Zustände
  b. Testfall ruft alle Funktionen auf

Aufgabe:
Erzeugen Sie für jede der beiden Minimalanforderungen einen passenden Testfall (mit Max=4)!

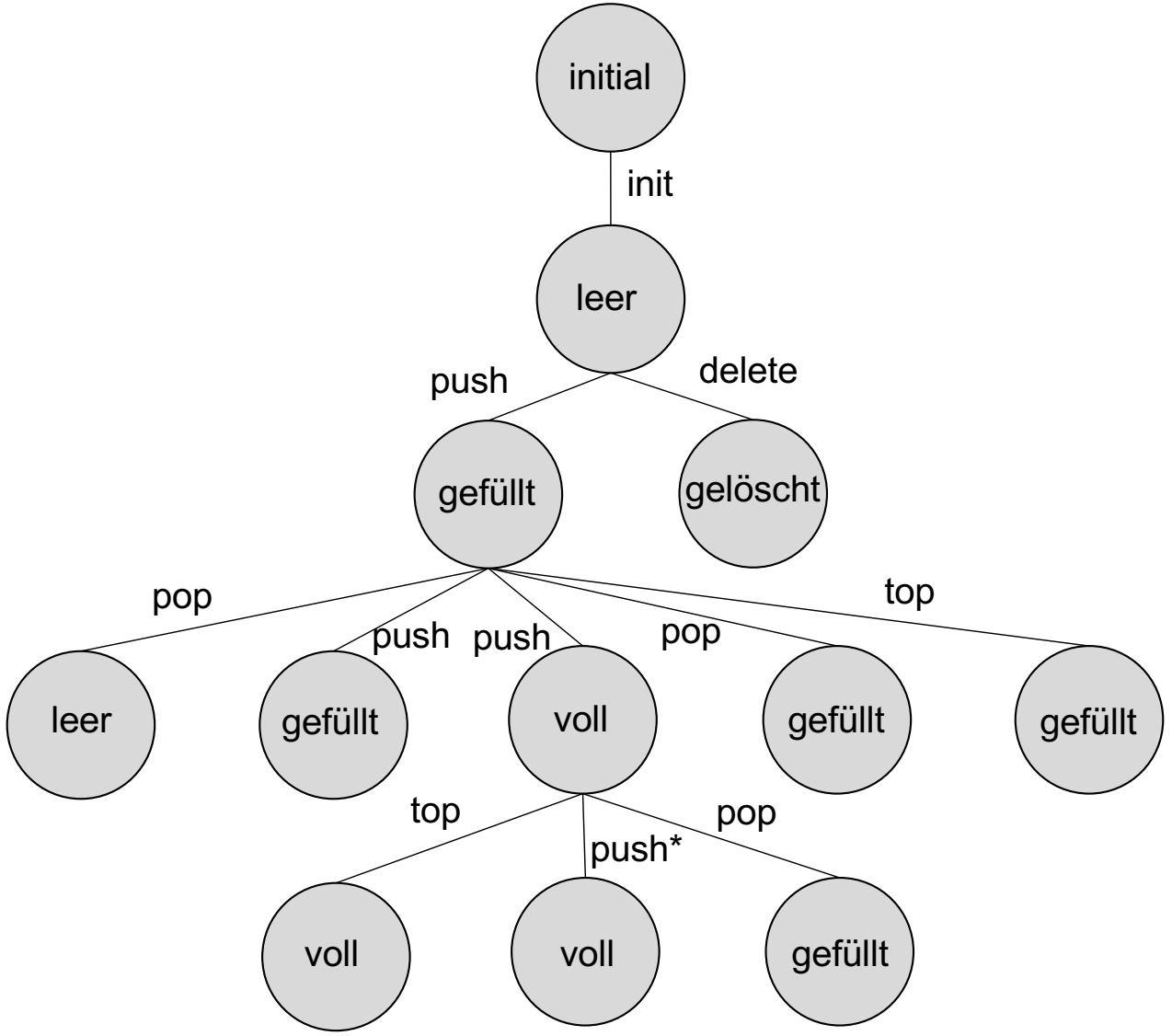
Wo liegen die Schwächen der beiden Testfälle?

- Testkriterium für einen zustandsbasierten Test

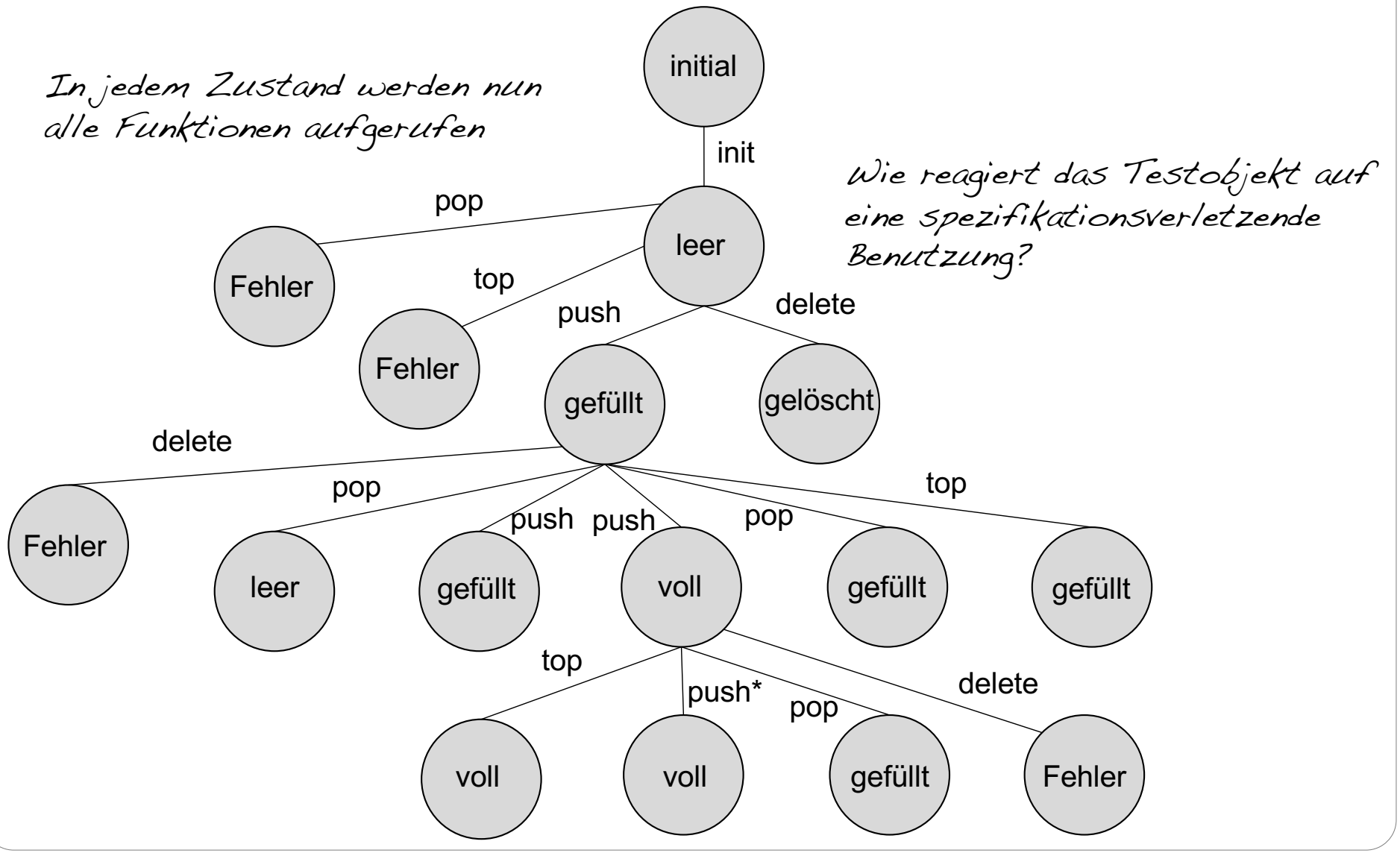
> *Für jeden Zustand sollen alle für diesen Zustand spezifizierten Funktionen mindestens einmal zur Ausführung kommen.*

- Zur Ermittlung der Testfälle wird aus einem zyklischen Zustandsdiagramm ein Entscheidungsbaum erstellt
- Der Entscheidungsbaum enthält alle Zustände und alle Zustandsübergänge
- Erzeugung des Entscheidungsbaums (Aufruf mit dem Startzustand)
  1. Für den aktuellen Zustand wir ein Knoten im Baum erzeugt
  2. Die Folgezustände werden als Söhne des aktuellen Knotens in den Baum eingehangen
  3. Die Folgezustände werden rekursiv traversiert, bis
     a. ein bereits besuchter Zustand erneut besucht wird
     b. ein Endzustand erreicht wird

- Übergangsbaum für den Stapel (Stack)



- Für einen Robustheitstest ist der Übergangsbaum zu erweitern



- Jeder Pfad im Übergangsbaum von der Wurzel bis zu einem Blatt repräsentiert einen Testfall
- Hinweis: Die Wächterbedingungen sind im Übergangsbaum nicht enthalten. Diese müssen aber bei der Bestimmung der Testfälle berücksichtigt werden. Z.B. dreimal push von „gefüllt“ nach „voll“ bei Max = 4.
- Definition Testfall
  - Anfangszustand
  - Eingabe
  - Soll: Ausgabe bzw. Verhalten
  - Soll: Endzustand
  - Pro Zustandsübergang: Ausgangszustand, Auslöser, Reaktion (Soll), Folgezustand (Soll)

Übungsaufgabe

Leiten Sie aus dem Übergangsbaum für den Stapel die enthaltenen Testfälle ab (ohne Robustheitstest).

- Der aktuelle Zustand eines Testobjekts ist häufig nur schwer zu ermitteln
- Überprüfung der Testfälle kann sehr aufwendig sein
- Überdeckungskriterien:
  - *x* Prozent der Zustände wurden mindestens einmal erreicht
  - *x* Prozent der Zustandsübergänge wurden mindestens einmal ausgeführt
  - *x* Prozent der spezifikationsverletzenden Zustandsübergänge wurden geprüft

4. Test spezieller Werte
   - Mit zunehmender Testerfahrung stellt man fest, dass bestimmte Eingaben, unabhängig von der konkreten Spezifikation, häufig Fehler aufdecken
   - Diese fehlersensitiven Eingaben sind zu dokumentieren und können dann in späteren Tests verwendet werden
   - Beispiele:
     - eine leere Eingabe
     - Eingabe mit Sonder- und Steuerzeichen

5. Zufallstest
   - Die Testfälle werden zufällig generiert
   - Die Datentypen und die Struktur der Eingaben müssen bei der zufälligen Generierung der Testdaten berücksichtigt werden
   - Eine entsprechende Werkzeugunterstützung ist erforderlich
   - Der Zufallstest ist ein ergänzendes Verfahren (z.B. können die Eingaben aus einer Äquivalenzklasse zufällig gewählt werden)
   - Der Vorteil von Zufallstests ist, dass auch nicht naheliegende Eingaben überprüft werden

Übungsaufgabe

Was spricht in der Praxis gegen die Durchführung von Zufallstests?

– Das *Property-based Testing* bietet ggf. eine praktikable Alternative zum Zufallstest

Property-based Testing

- Beim Unit-Test haben wir für Testfälle *x* geprüft, ob

$$f_{SUT}(x) = f_{Soll}(x)$$

  - Bei einer Abweichung liegt ein Fehler vor
  - Dabei wird $f_{Soll}(x)$ nicht berechnet, sondern ist in Form von Tabellen vorgegeben (in die Testfälle codiert). Dies erschwert Zufallstests

- Beim *property-based Testing* wird geprüft, ob ein Prädikat erfüllt ist

$$\forall x.P(x, f_{SUT}(x))$$

- Dabei ist f_{SUT} die zu testende Funktion und *P* ist eine ausführbare logische Relation zwischen der Eingabe *x* und der Ausgabe $f_{SUT}(x)$
  - Die Eingaben *x* können nun zufällig (in großer Anzahl) generiert werden
- Sobald das Prädikat nicht erfüllt ist, liegt ein Fehlschlag vor (dies muss nicht zwingend ein Fehler in f_{SUT} sein)

Aufgabe:

a) Es soll die statische Methode `String Util.reverse(String text)` getestet werden. Die Methode soll die Reihenfolge der Buchstaben im Text umkehren, z.B. `Util.reverse("Text").equals("txeT") == true` Welche Eigenschaft aus der Spezifikation von `reverse` hilft Ihnen dabei, das Prädikat zu formulieren? Hinweis: Es handelt sich um eine Invariante, die für alle Eingaben gilt.

b) Es soll die Methode `void Util.bubbleSort(Comparable[] feld)` getestet werden. Was wäre eine geeignete Möglichkeit, um das Prädikat zu formulieren?

c) Warum ist ein Fehlschlag (*failure*) nicht unbedingt ein Programmfehler (*error*)

- Ein bekanntes Werkzeug für das *property-based Testing* ist *QuickCheck* von Claessen und Hughes
  - Das Werkzeug generiert die Eingaben zufällig. Die Zeugen für einen Fehlschlag können daher unnötig komplex sein. Dies erschwert die Ursachensuche
  - Daher werden Zeugen von *QuickCheck* verkleinert (*shrinking*)

- QuickCheck ermöglicht die Betrachtung von extrem vielen Testfällen (im Vergleich zu klassischen Unit-Tests). Es gibt aber auch Einschränkungen
  - Es ist nicht ausgeschlossen, dass Eingaben mehrfach generiert werden
  - Es ist nicht ausgeschlossen, dass eine große Anzahl uninteressanter Eingaben generierte wird
  - Es existieren daher diverse Erweiterungen des Verfahrens (z.B. Berücksichtigung der bisherigen Code-Überdeckung bei der Generierung von Eingaben, kombinatorisches Vorgehen).

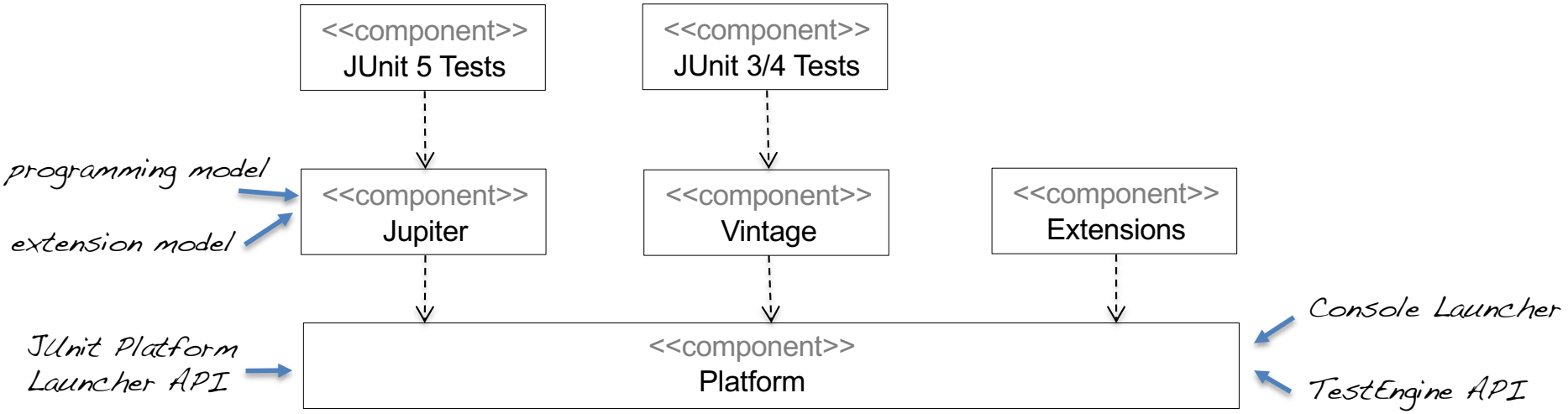
- Literaturauswahl:
  - Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ‘00), Montreal, Canada, September 18-21, 2000. pp. 268-279, ACM, 2000
  - Goldstein, H., Hughes, J., Lampropoulos, L., Pierce, B.: Do Judge a Test by its Cover. Combining Combinatorial and Property-Based Testing. In: Yosihda, N. (eds.) ESOP 2021, LNCS 12648, pp. 264-291, 2021

Testautomatisierung

- Der Vorteil von systematischen Tests ist die Wiederholbarkeit der Tests
- Wiederholung eines Tests auf Grund von
  - Programmänderungen (Fehlerkorrektur) -> Fehlernachtest
  - Programmerweiterungen -> Regressionstests
- Mit der Wiederholung von Tests können Modifikationen am Programm abgesichert werden
- Eine häufige Wiederholung erfordert eine Testautomation
- Die Testautomation kann auf Basis existierender Test-Frameworks und Werkzeugen erfolgen (Reduzierung des Aufwands)
- Als Beispiel betrachten wir das JUnit-Framework für Java-Programme

- JUnit bietet Unterstützung bei der Erstellung und der Durchführung von Programmtests
  - Entlastung von Routinetätigkeiten bei der Testfallprogrammierung
  - Automatisierung der Testdurchführung
  - Zählen und Berichten von Fehlern

- In der Praxis sind die Versionen JUnit 4 und JUnit 5 verbreitet. Wir betrachten JUnit 5 ← einen kleinen Ausschnitt
- JUnit 5 wurde zu einer Testplattform ausgebaut



- JUnit 5 - Testklasse
  - Die Testklasse muss weder eine spezielle Basisklasse noch einen speziellen Klassennamen besitzen (Empfehlung: der Klassenname erhält als Postfix den Zusatz `Test`)
  - Testmethoden werden durch die Annotation `@Test` ausgezeichnet

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

public class StringTest {

    @Test
    public void testStringtoUpperCase(){
        final String istWert = "Das ist ein Testfall.";
        final String sollWert = "DAS IST EIN TESTFALL.";

        assertEquals(sollWert, istWert.toUpperCase());
    }
}
```

Empfehlung: Nur ein Testfall pro Testmethode

- Falls eine Testklasse mehrere Testmethoden enthält, dann wird vor der Ausführung jeder einzelnen Testmethode eine neue Instanz der Testklasse erzeugt (Reduzierung von Seiteneffekten)

- Werden zur Erstellung einer konsistenten Testumgebung umfangreichere Initialisierungen benötigt, können diese in eine Methode ausgelagert werden, die mit `@BeforeEach` annotiert ist
  - die Methode wird dann vor jeder einzelnen Testmethode aufgerufen

- Die Freigabe von Ressourcen kann dann in einer mit `@AfterEach` annotierten Methode erfolgen.

- Wichtige JUnit 5-Annotationen
  - die annotierten Methoden sind `public`, `void` und ohne Parameter

`@Test`	Kennzeichnet eine Methode (`public`, `void` und ohne Parameter) als Testfall
`@BeforeEach`	Die gekennzeichnete Methode wird vor jeder Testmethode aufgerufen
`@AfterEach`	Die gekennzeichnete Methode wird nach jedem Testfall aufgerufen
`@BeforeAll`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, bevor die Testmethoden gestartet werden
`@AfterAll`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, nachdem die Testmethoden ausgeführt wurden
`@DisplayName("")`	Für Testklassen und Testmethoden können Namen für das Testprotokoll vergeben werden

  - Es können beliebig viele Methoden mit `@BeforeEach`, `@AfterEach`, `@BeforeAll` und `@AfterAll` gekennzeichnet werden
  - Die Reihenfolge der Ausführung ist nicht definiert

- Innerhalb einer Test-Methode findet ein Soll-Ist-Vergleich statt.
  - Für den Vergleich stehen eine Reihe von `assert`-Methoden zur Verfügung
  - Die `assert`-Methoden sorgen auch für eine Protokollierung des Testverlaufs

– Wichtige `assert`-Methoden

bei assertEquals und assertSame wird erst der Soll-Wert und dann der Ist-Wert angegeben

Methode	Beschreibung
`assertEquals(Object A, Object B, String message)`	Prüft `A.equals(B)`
`assertEquals(int a, int b, String message)`	Prüft `(a==b)`
`assertEquals(double a, double b, double v, String message)`	Prüft `abs(a-b)<=v`
`assertSame(Object A, Object B, String message)`	Prüft `(A==B)`
`assertTrue(boolean b, String message)`	Prüft `(b == true)`
`assertNotNull(Object A, String message)`	Prüft `(A != null)`

Fehlermeldung für JUnit-Protokoll, darf auch entfallen

– Eine beliebige Anzahl von Überprüfungen lässt sich mit `assertAll` und Lambda-Ausdrücken zusammenfassen

```
@DisplayName("Rationale Zahl“)
class RationalTest
    @DisplayName("multipliziert mit rationaler Zahlen")
    @Test
    public void testMultiply() {
       Rational l = new Rational(1,2);
       Rational r = new Rational(3,2);
       Rational e = l.multiply(r);
       assertAll("Fehler bei der Multiplikation",
                 ()-> assertEquals(3, e.getZaehler(),
                                   "Zähler falsch"),
                 ()-> assertEquals(4, e.getNenner(),
                                   "Nenner falsch"));
    }
```

- Erwartete Ausnahmen
  - Auf ungültige Eingaben muss das zu testende Programm geeignet reagieren
  - Eine Möglichkeit besteht im Auslösen (Werfen) einer Ausnahme (`Exception`)
  - Ein ungültiger Testfall würde in diesem Fall einen Fehler aufdecken, wenn keine Ausnahme ausgelöst wird
  - Diese Situation muss im Testfall geeignet abgebildet werden

- Wir testen die folgende Klasse

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

    public int divide(int n, int d){
        if(d == 0) throw
                    new ArithmeticException("Division durch 0");
        return n/d;
    }
}
```

die Methoden sind hier nur aus didaktischen Gründen nicht static (siehe @BeforeEach auf der nächsten Seite)

nur zur Verdeutlichung, nicht zwingend erforderlich

- Erwartete Ausnahme mit der Assertion `assertThrows`

```
public class CalculatorTest {
   private Calculator cal;

   @BeforeEach
   public void erzeugeCalculator(){
      cal = new Calculator();
   }

   @Test
   public void testDivision(){
      assertEquals(2, cal.divide(2,1));
   }

   @DisplayName("Test mit ungültigen Wert 0")
   @Test
   public void testDivisionDurchNull() {
      assertThrows(ArithmeticException.class,
                   ()->cal.divide(2,0),
                   "erwartete Exception nicht geworfen");

   }
}
```

Typ der erwarteten Ausnahme

Aufruf des SUT über Lambda-Ausdruck

Optionaler Text für das Testprotokoll

- Starten der Testfälle
  - über die Console

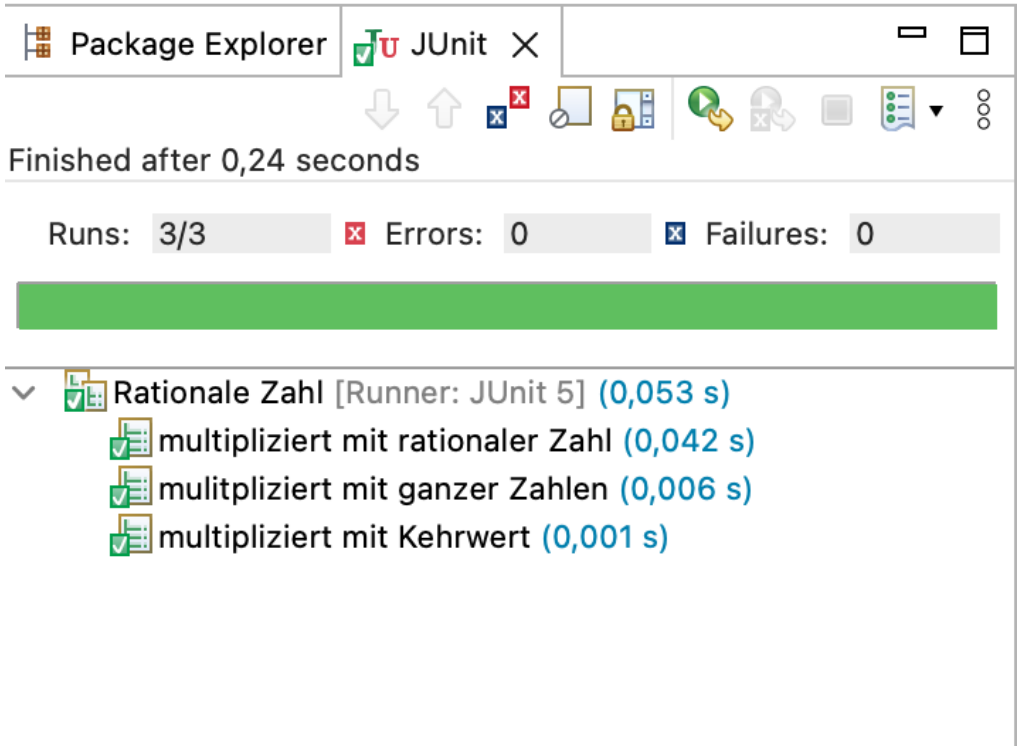
Die jar-Datei für den ConsoleLauncher kann über das Maven-Central-Repository bezogen werden

```
> java -jar junit-platform-console-standalone-1.9.1.jar
      --classpath /Users/dwiesmann/eclipse-workspace/SWTD/bin
      -c fh.swtd.rational.RationalTest
```

```
SWTD — -zsh — 80×31

dwiesmann@Dirks-MBP-16 SWTD % java -jar junit-platform-console-standalone-1.9.1.
jar --classpath /Users/dwiesmann/eclipse-workspace/SWTD/bin -c fh.swtd.rational.
RationalTest

Thanks for using JUnit! Support its development at https://junit.org/sponsoring

├─ JUnit Jupiter ✔
│  └─ Rationale Zahl ✔
│     ├─ multipliziert mit rationaler Zahl ✔
│     ├─ mulitpliziert mit ganzer Zahlen ✔
│     └─ multipliziert mit Kehrwert ✔
├─ JUnit Vintage ✔
└─ JUnit Platform Suite ✔

Test run finished after 180 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
[         3 tests found           ]
[         0 tests skipped         ]
[         3 tests started         ]
[         0 tests aborted         ]
[         3 tests successful      ]
[         0 tests failed          ]
```

über eine geeignete Benennung der Testmethoden, bzw. mit @DisplayName kann Einfluss auf die Lesbarkeit der Protokolle genommen werden

- Integration in IDE (z.B. Eclipse)



Errors: Unerwartete Ausnahmen
Failures: Abweichung beim Soll-/Ist-Vergleich (`assert`)

- Aufruf über Build-Tool mit Integration in das Berichtswesen (siehe später)

- Test-Suite
  - Die Testfälle (mit `@Test` annotierte Methoden) einer Testklasse werden von JUnit automatisch zu einer *Suite* zusammengefasst
  - Alle Testfälle einer *Suite* werden dann nacheinander abgearbeitet (auch wenn Tests fehlschlagen)
  - Es kann auch eine *Suite* von Testklassen gebildet werden, um alle Testfälle in den einzelnen Klassen ablaufen zu lassen

```
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({RationalTest.class, QueueTest.class})
public class AllTests {

}
```

- Parametrisierte Testmethoden
  - Eine Testmethode kann mehrfach mit unterschiedlichen Parametern aufgerufen werden
  - Eine Testmethode kann damit nacheinander verschiedene Testfälle realisieren
  - Die Testklasse stellt die Testfalldaten in einem (funktionalen) Stream zur Verfügung. Sie erhält dafür eine statische Methode als Datenquelle
  - Die Testfalldaten enthalten den Soll-Wert und die Eingabewerte
  - Ein spezieller Runner ruft dann für alle Testfalldaten aus dem Stream nacheinander die Test-Methode auf. Die Testmethode ist mit dem Namen der Datenquelle annotiert
  - Für jeden Testfall aus dem Stream wird eine neue Instanz der Testklasse erzeugt
  - Der Runner übergibt die Testfalldaten der Testmethode als Argumente
  - Ggf. ist ein spezieller `SimpleArgumentConverter` zu implementieren, um eine Typkonvertierung zwischen den in `Arguments` enthaltenen Typen und den Parametertypen der Testmethode vorzunehmen

```
public class CalculatorParameterizedTest {

   public static Stream<Arguments> daten(){
      Arguments[] testdaten = {Arguments.of(0, 2, 3),
                               Arguments.of(2, 4, 2),
                               Arguments.of(3, 7, 2),
                               Arguments.of(5, 10, 2)};
      return Arrays.asList(testdaten).stream();
   }

   @ParameterizedTest
   @MethodSource({"daten"})
   public void testDivisions(int soll, int zaehler, int nenner) {
      Calculator cal = new Calculator();
      assertEquals(soll, cal.divide(zaehler, nenner));
   }
}
```

Datenquelle

Daten für einen Testfall

Methodenname der Datenquelle

- Die Testfalldaten können auch aus einer csv-Datei eingelesen werden

```
public class CalculatorParameterizedTest {

    @ParameterizedTest
    @CsvFileSource(resources= {"/div.csv"}, numLinesToSkip = 1)
    public void testDivisions(int soll, int zaehler, int nenner) {
        Calculator cal = new Calculator();
        assertEquals(soll, cal.divide(zaehler, nenner));
    }
}
```

Kopfzeile überlesen

```
Soll, Zaehler, Nenner
0, 2, 3
2, 4, 2
3, 7, 2
5,10, 2
```

div.csv