# Edit and P(r)ay – Oder doch lieber testen?

*Anja Papenfuß-Straub, ING Deutschland*

*Tests und Testautomatisierung sind nichts Neues, allerdings ist ihre Bedeutung in den letzten Jahren gestiegen. Mit der zunehmenden Digitalisierung ändert sich das Kundenverhalten massiv. Von uns als Entwickler wird die zeitnahe Umsetzung und Live-Stellung neuer Features erwartet. Dabei sehen wir uns mit zunehmend schnellerer technologischer Entwicklung konfrontiert. Wir müssen sicherstellen, dass unser Code funktioniert, und trotzdem fällt uns die Testautomatisierung unserer Software noch schwer. Dieser Artikel soll ermutigen und zeigen, dass Testen sinnvoll und gar nicht so schwer ist.*

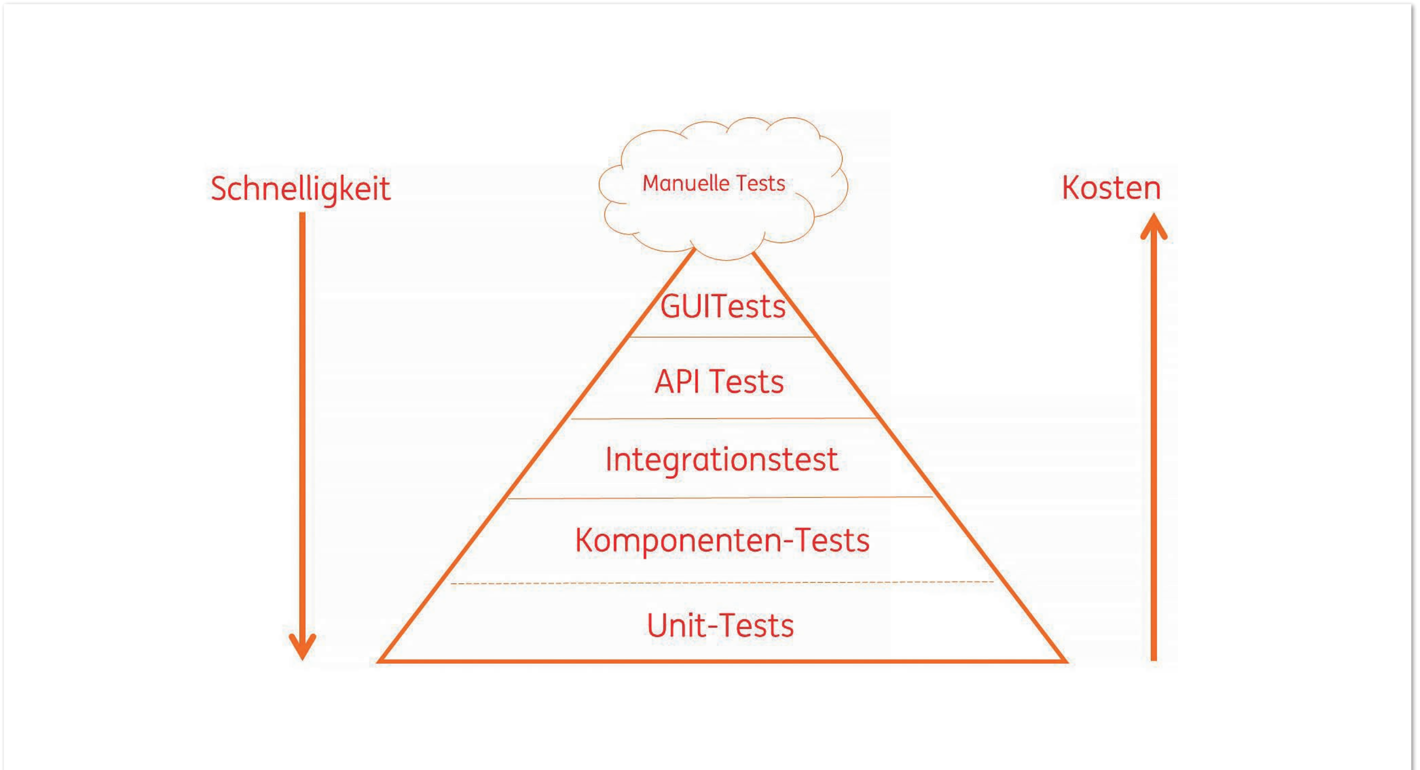


*Abbildung 1: Testpyramide (© Anja Papenfuß-Straub)*

Vor über zehn Jahren sah die Softwareentwicklung noch anders aus. Ich kann mich noch gut an umfangreiche Debugging-Sessions erinnern, um Fehler auf dem Produktivsystem zu finden. Bei einem meiner ehemaligen Arbeitgeber hing am IT-Chef-Büro ein riesiges Poster: „Testen ist was für Mädchen, Jungs gehen live!". Unit-Tests waren explizit verboten, sie galten als zu teuer. Die Tester durften aufwendige End-2-End-Tests schreiben (damals HttpUnit-Tests). Wir waren froh über diese Tests, brachten sie doch zumindest ein kleines Gefühl von Sicherheit. Jeder, der Fehler auf Produktionssystemen suchen und beheben muss, weiß, wie nervenaufreibend so etwas ist. Erstaunlicherweise gab es relativ wenig größere Zwischenfälle. Immerhin plante unser Kunde eine lange Testphase vor jedem Release ein. Das half ebenfalls, noch Fehler zu entdecken, und wir hatten nur wenige Releases pro Jahr.

Doch die Zeiten ändern sich. Was dieser digitalen Welt fehlt, ist Geduld. Darum sind die Unternehmen zunehmend unter Druck, auf Kundenwünsche zeitnah zu reagieren. Damit sind auch wir als Entwickler gefordert, immer schneller neue Features zu liefern und diese in kurzen Release-Zyklen produktiv zu stellen. Dabei dürfen sich Code-Qualität und Security keinesfalls verschlechtern. Mittlerweile arbeiten die meisten von uns in agilen Teams und sind als „DevOps" verantwortlich für Entwicklung und Betrieb unserer Software. Wir können nicht mehr auf lange Testphasen durch Fachtester und Fachteams vor Live-Stellungen zählen. Darum ist eine Testautomatisierung zwingend erforderlich. In der agilen Arbeitsweise sind darum Tests bereits Teil der „Definition of Done" und ein Kriterium für die Abnahme von Features. Sie sind die Grundvoraussetzung zur Absicherung unserer Software. Die Unternehmen haben erkannt, dass Produktionsfehler um ein Vielfaches teurer sind als Fehler, die während der Entwicklungsphase erkannt werden. Im schlimmsten Fall drohen Reputationsverlust bei den Nutzern und finanzielle Einbußen.

## Software Lifecycle

Um die Bedeutung zu verstehen, die dem Testen tatsächlich gebührt, lohnt sich ein Blick auf den Software Lifecycle. Software wird ständig verändert und weiterentwickelt, zumindest erfolgreiche Software. Wir ändern Code ganz einfach über unsere Entwicklungsumgebung. Ändern und erweitern wir unseren Code, ohne dass wir die Code-Struktur anpassen, wird unser Code starr und dadurch schlechter änderbar. Der Grad der Unordnung innerhalb unseres Codes nimmt zu, auch bekannt als Software-Entropie.

Bei jeder neuen zu implementierenden Funktion müssen wir einen Kompromiss finden zwischen den begehrten neuen Features und der vorhandenen Code-Basis. Gilt unser Augenmerk nur der Entwicklung von neuem Code zulasten der Bestandsbasis, führt das zuerst zum Verlust der strukturellen Qualität (Architektur, Entwurfsmuster etc.) und im schlimmsten Fall auch zum Verlust der funktionalen Qualität (Fachlichkeit). Im Laufe der Jahre hat wohl fast jeder Entwickler seine Erfahrungen mit Legacy-Anwendungen gemacht. Oft war die ursprüngliche Absicht des Code-Erstellers nach mehreren Anpassungen nicht mehr einfach erkennbar. Wir sprechen in diesem Fall von Software-Erosion beziehungsweise von technischen Schulden.

Schnelle Änderungen an diesem Code sind kaum oder nur langsam möglich, sie sind fehleranfällig und teuer. Software lebt aber von Veränderungen und muss anpassbar bleiben. Das heißt, dass eine Anpassung der Software einfach und schnell durchführbar ist. Dabei ist die Struktur des Codes klar erkennbar und der Code ist ohne großen Aufwand veränderbar. Die einzige Möglichkeit, Software-Erosion unter Beachtung der Software-Entropie zu verhindern, ist regelmäßiges Refactoring. Um unsere Neuentwicklungen und Refactorings abzusichern, braucht es einen Test-Harnisch.

```
@Autowired
private SearchManager classUnderTest;

@Test
public void findName(){
  //Arrange
  SearchQuery query = new SearchQuery("name");
  //Act
  List<SearchResult> results = classUnderTest.find("name");
  assertThat(results).extracting(SearchResult::name)
     .containsOnly("name");
}
```

*Listing 1: Tripple A (Arrange Act Assert) mit assertj als Assertion Framework*

Der Test-Harnisch fungiert als Sicherheitsnetz und besteht aus vielen einfachen und gut wartbaren Entwicklertests (Unit- bzw. Komponententests) sowie wenigen Integrationstest (Testpyramide, siehe *Abbildung 1*). Die Testpyramide definiert die Testarten und deren Umfang unter Beachtung von Kosten und Nutzen. Günstige und schnelle Testarten werden bevorzugt (Unit-Tests). Mit jeder Anpassung und Erweiterung unseres Codes vergrößern wir die Test-Basis und verfeinern die Maschen unseres Sicherheitsnetzes. Das hilft uns, die strukturelle und funktionale Qualität zu erhalten beziehungsweise zu verbessern.

## Code Coverage und Testqualität

Code Coverage, ein prozentualer Wert, der die Testabdeckung im Code misst, wird gern als Druckmittel und Heilsbringer verwendet. Zahlen wie diese werden auch vom Management gut verstanden. Aber welche Aussagekraft hat dieser Wert? Wenn wir ehrlich sind, ist eine hohe Code Coverage noch lange kein Garant für gute und sinnvolle Tests. Code Coverage prüft nur die Codezeilen und Abzweigungen, die bei einem Test durchlaufen wurden. Die Validierung des Testergebnisses und die Richtigkeit unseres Codes spielen dabei keine Rolle. Verlassen wir uns also rein auf die Testabdeckung, wiegen wir uns in falscher Sicherheit.

Trotzdem bin ich nicht grundsätzlich gegen Code-Coverage-Vorgaben. Eine geringe Code Coverage ist immerhin ein Indiz für notwendige Nachbesserungen in Sachen Tests. In Kombination mit statischer Code-Analyse sind durchaus gute Ergebnisse zu erzielen, um die Test- beziehungsweise Code-Qualität zu erhalten und zu verbessern.

Was aber macht einen guten Test aus? Kurz gesagt sollte ein Test dem **F.I.R.S.T**-Prinzip entsprechen, dem Grundsatz für Test-Driven-Development (TDD):

- **Fast:** Die Testlaufzeiten sollten gering sein, bei reinen Unit-Tests sogar im Millisekunden-Bereich. Integrationstests können auch länger laufen.
- **Isolated/Independent:** Testmethoden einer solchen Testklasse dürfen sich nicht gegenseitig bedingen. Sollten Tests aufeinander aufbauen, wird die Wartung dieser Tests extrem erschwert. Die Testmethoden können nicht unabhängig voneinander einzeln ausgeführt werden.
- **Repeatable:** Die Tests sollen wiederholbar sein und bei jedem Lauf das gleiche Ergebnis liefern.
- **Self-Verifying:** Es sollte nur ein Aspekt pro Test getestet werden. Idealerweise besteht ein Test aus Arrange (Initialisieren des Test-Setups), Act (Ausführen des zu testenden Codes) und Assert (der Validierung des Ergebnisses), siehe *Listing 1*. Durch die automatische Assertion ist eine manuelle Verifikation unnötig. Darum: Spart euch Logausgaben in Tests! Wer soll die lesen?
- **Thorough/Timely:** Das Ziel ist nicht, eine hundertprozentige Code Coverage zu haben, sondern einen Test-Harnisch aufzubauen, dem wir vertrauen. Und zu guter Letzt sollten Tests immer vor dem Implementieren der Funktionalität geschrieben werden (TDD).

## Code-Qualität

Bei umfangreichen Refactorings in großen Bestandssystemen habe ich leidvoll erfahren: Einer der größten Hinderungsgründe, um gute Tests zu schreiben, ist und bleibt schlechte Code-Qualität. Schlechte Code-Qualität ist meist die Hauptursache, warum wir das Schreiben von Tests als aufwendig und langsam empfinden. Im Folgenden ein paar Beispiele, die jeden Entwickler, der Tests schreiben soll, in die Knie zwingen.

**Hohe Komplexität:** Wer kennt sie nicht, die Gott-Klassen. Riesig und sehr komplex vermitteln sie den Eindruck, alles zu können und jeden zu kennen. Neben Angst erzeugen diese Klassen vor allem Ärger beim Testen. Das Test-Setup für diese Klasse wird extrem aufwendig und die Wartung teuer. Außerdem leidet die Lesbarkeit. So wenig, wie der Code der Klasse verstanden wird, so wenig wird auch der Test dafür verstanden. Nebenbei ist so eine hohe Komplexität immer ein Indiz für Code Smell, zum Beispiel wenn das Single Responsibility Principle (SRP) nicht beachtet wurde. Das heißt, die Klasse enthält viele Funktionen, für die sie eigentlich nicht zuständig ist. Hier sollten wir die Abhängigkeiten analysieren, die Gemeinsamkeiten extrahieren und in separate Klassen überführen, die dann einzeln auch viel besser testbar sind.

**Falsch verwendete Vererbung:** Im Informatik-Studium wird uns Vererbung immer noch als Allheilmittel verkauft. Aber Hand aufs Herz, verwenden wir Vererbung wirklich immer richtig? Ist uns das Liskovsche Substitutionsprinzip immer gegenwärtig, wenn wir programmieren? Wir sind dazu erzogen worden, Code-Duplizierung zu vermeiden, und dabei dem Irrglauben verfallen, dass Vererbung hier der richtige Weg sei. Vererbung ist allerdings die engste Form der Kopplung. Angesicht der Bedeutung der Änderbarkeit von Software ist jedoch eine lose Kopplung (Komposition anstelle von Vererbung) zu bevorzugen („composition over inheritance"). Kompositionsklassen sind die geschicktere Möglichkeit, um Code-Duplizierung und eine enge Kopplung zu vermeiden. Sie sind in der Regel besser und isoliert testbar.

```java
public final class SpecialDateUtil {

    public static String getTimeOfDay() {
        DateTime time = new DateTime();
        if (time.getHour() >= 0 && time.getHour() < 6) {
          return "Night";
        } if (time.getHour() >= 6 && time.getHour() < 12) {
          return "Morning";
        } if (time.getHour() >= 12 && time.getHour() < 18) {
          return "Afternoon";
        } return "Evening";
    }
}
```

*Listing 2: Statische Klasse mit statischer Methode*

```java
@Test
public void getTimeOfDay_6AM_Morning() {
  try
  {
     // Setup: change system time to 6 AM
     ...
     String timeOfDay = SpecialDateUtil.getTimeOfDay();

     assertThat(timeOfDay).isEqualTo("Morning");
  }
  finally
  {
     // Teardown: roll system time back
     ...
  }
}
```

*Listing 3: Testmethode für Listing 2*

**Statische Klassen:** Die Namen dieser Klassen enden meist auf „Util“ oder „Commons“ *(siehe Listing 2)*. Es ist sehr einfach, diese zu schreiben und zu verwenden. Schwierig wird hier aber das Testen vor allem für die Klassen, die diesen statischen Code verwenden *(siehe Listing 3)*. Das Setup solcher Tests wird komplizierter, da die Elemente zum Durchlaufen des statischen Codes vorher initialisiert werden müssen. Statische Klassen sind ein Indiz für eine Design-Schwäche, die darauf hinweist, dass hier eine Methode ist, die eigentlich zu einer anderen Klasse gehört. Die Wahrheit ist: Statische Klassen mutieren oft zu riesigen Monstern, zu einer Halde für Methoden, die Entwickler sinnvoll für andere halten. Oft haben wir uns wenig Gedanken gemacht, wo diese statischen Methoden eigentlich hingehören – im Idealfall zur Klasse, welche die statische Methode aufruft. Statische Klassen enthalten prozeduralen Code, alles andere ergibt zumindest keinen Sinn. Wir bewegen uns mit Java als Programmiersprache allerdings in der objektorientierten Welt. Passt prozeduraler Code zum Unit-Testing? Beim Unit-Test sollen wir über Instanziierung ein Stück der Applikation isoliert testen. Dabei versuchen wir, alle Abhängigkeiten durch Mocks etc. zu ersetzen. Prozeduraler Code kennt jedoch so ein „wiring“ gar nicht, da gibt es keine Objekte, keinen Code und keine Daten, die separierbar sind. Darum ist das Testen hier so schwierig *(siehe Listing 3)*.

Der Test für die Klasse aus *Listing 2* wird, je nach Ausführzeitpunkt des Tests, ein anderes Ergebnis liefern. Das heißt, die Methode enthält einen veränderbaren globalen State und verhält sich nicht deterministisch.

Um das nicht-deterministische Verhalten im Test zu berücksichtigen, muss jeweils das System-Datum kompliziert gesetzt und nach dem Test zurückgesetzt werden. Aus einem reinen Unit-Test wird dann schnell ein Integrationstest, in dem umständlich ein Environment gesetzt werden muss. Ein Black-Box-Test ist nicht möglich, denn wir müssen den Code kennen, um ihn zu testen. Die if-else-Kaskade macht das Testen auch umfangreicher, da wir für eine zu testende Methode vier Testmethoden schreiben müssen.

Wir wollen selbst bestimmen, welchen Wert „timeOfDay“ *(siehe Listing 2)* bekommt, damit wir nicht noch einmal die Testaufwände aus *Listing 3* duplizieren. Die SpecialDateUtil-Klasse enthält nicht

```java
public class FlyerHeadlineGenerator {

  public String createFlyerHeadline(){
     String timeOfDay = SpecialDateUtil.getTimeOfDay();
     return "Welcome to our event next thursday " + timeOfDay;
  }
}
```

*Listing 4: Verwendung statischer Klasse aus Listing 2*

```
public class DateWordGenerator {

  public String getTimeOfDay(DateTime time) {
     if (time.getHour() >= 0 && time.getHour() < 6) {
       return "Night";
     } if (time.getHour() >= 6 && time.getHour() < 12) {
       return "Morning";
     } if (time.getHour() >= 12 && time.getHour() < 18) {
       return "Afternoon";
     } return "Evening";
  }
}
```

*Listing 5: Code-Anpassung für bessere Testbarkeit*

nur einen globalen State, sondern sie ist auch noch statisch und dadurch nur mit Zusatz-Frameworks mockbar.

Um den Code testbarer zu machen, lösen wir den globalen State auf und übergeben das DateTime-Objekt als Parameter. Klasse und Methoden sind nicht mehr statisch und können zum Beispiel als Singleton Instance weiterleben. Weitere Beispiele für Code, der Testbarkeit behindert:

- Logik im „Constructor"
- Erzeugen eigener Abhängigkeiten durch „new Objects" innerhalb von Methoden
- Verletzung des Prinzips „law of demeter"
- if-else-Kaskaden im Code
- Abhängigkeiten zu riesigen Kontext-Objekten
- Verwenden statischer Initializer

## Fazit

Testen ist sinnvoll und testgetriebene Entwicklung bringt jede Menge Vorteile. Immerhin hilft sie, saubere und testbare Architekturen zu bauen sowie regelmäßige und abgesicherte Refactorings durchzuführen. Die Akzeptanz eines Vorgehens, unabhängig davon, ob es sich um Testautomatisierung oder andere Themen handelt, hat viel mit unserer inneren Einstellung zu tun. Wir hängen an unseren Gewohnheiten, sie geben uns ein Gefühl von Sicherheit. Wir sind eher bereit, diese zu verändern, wenn wir positive Erfahrungen machen. Mit der Erkenntnis, dass wir durch testgetriebene Entwicklung sicherstellen können, dass unser Code funktioniert, und dass wir befähigt werden, bessere Qualität zu liefern, wächst auch die Bereitschaft, Zeit in das Schreiben von Tests zu investieren.

Statische Unternehmensvorgaben sind sinnvoll, aber diese allein helfen nicht. Der Wandel vom auftragsbezogenen Abarbeiten fachlicher Anforderungen hin zu verantwortungsbezogenem Entwickeln von Software funktioniert nur mit der Veränderung unserer Grundeinstellung. Unser Ziel als Entwickler sollte letztendlich sein, gute, wartbare und stabile Software zu schaffen, die den Ansprüchen an funktionale und qualitativ gute Software gerecht wird.

Wir sollten als selbstorganisierte Teams fähig sein, der Forderung nach neuen Features selbstbewusst zu begegnen. Die Verantwortung eines Entwickler-Teams geht über die Entwicklung neuer Fachlichkeit hinaus und braucht Zeit. Letztendlich müssen alle Beteiligten diesen Weg gehen wollen. Am Ende gewinnen wir alle, weil wir nur dadurch nachhaltig Schnelligkeit und Sicherheit garantieren können. Also lasst uns immer daran denken:

***„Softwaretesting is not about finding bugs. It's about delivering great software"***

(© Harry Robinson, Software Test Lead, Microsoft)

## Quellen


[1] Frank Westphal - Testgetriebene Entwicklung mit JUnit & FIT, dpunkt.verlag, 2006
[2] *http://misko.hevery.com/2008/07/24/how-to-write-3v1l-untestable-code/*
[3] *http://misko.hevery.com/2008/12/15/static-methods-are-death-to-testability/*
[4] *https://www.atlassian.com/blog/add-ons/deliver-faster-and-better-software-using-test-automation*
[5] *http://www.nils-haldenwang.de/german/das-liskovsche-substitutionsprinzip-in-java-invarianz-kovarianz-kontravarianz*
[6] *https://www.codepedia.org/toptal/unit-tests-how-to-write-testable-code-and-why-it-matters/*

**Anja Papenfuß-Straub**
ING Deutschland
*Anja.Papenfuss-Straub@ing.de*


Anja hat fast 19 Jahre Erfahrung als Java-Softwareentwicklerin und ist seit über sechs Jahren Java-Backend-Entwicklerin und Application Architect bei der ING Deutschland am Standort Nürnberg. Dabei liegt ihr Fokus auf Code-Qualität und Testbarkeit. Sie hat die ING Deutschland aktiv bei der Einführung neuer Testautomatisierungs-Vorgaben für die Softwareentwicklung begleitet und die Entwickler bei der Umsetzung unterstützt.