HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

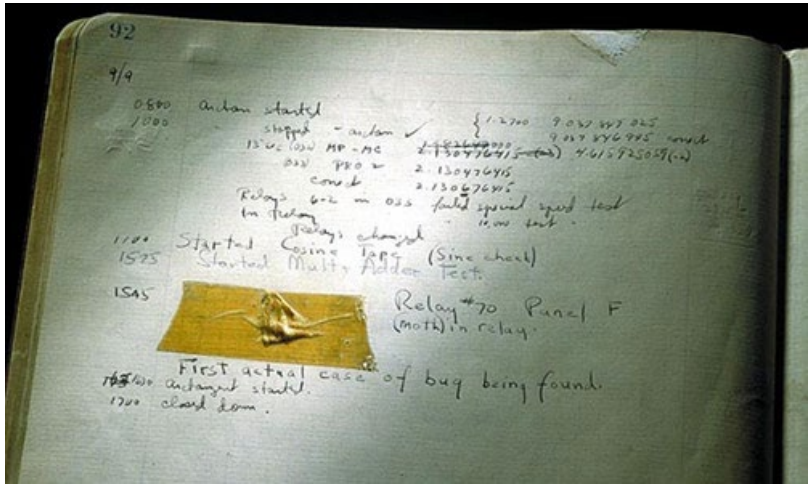
Clean Code: Unit Tests

Roland Gisler

FH Zentralschweiz

Unit-Tests nach Clean Code

- Clean Code: Kapitel 9 – Unit-Tests; Heuristiken T1 – T9
- Unit-Tests sind bei den meisten Entwickler*innen gut etabliert, es gibt aber noch viel Potential! (siehe Test-First)
- Qualität des Testcodes ist wichtig!
  - Testcode nicht als «Wegwerfcode» behandeln!
- Dank Unit-Tests finden wir Bugs schneller und früher!



4. April 1945 um 15:15 im Naval Weapons Center, Virginia:

Admiral Grace Murray Hopper protokolliert im Tagesjournal den ersten Bug in einem Programm das auf dem Harvard Mark II Aiken Relay Calculator life (so die Legende).

https://de.wikipedia.org/wiki/Grace_Hopper

Motivation

- Gute und umfassende Tests (egal welcher Art) sind die fundamentale Basis für alle weiteren Bemühungen zur Verbesserung der (Code-)Qualität.
  - Qualität in Form von :
    Reliability, Changeability, Efficiency, Security, Maintainability, Portability, Reusability etc.
- Gute Unit Tests sind die erste, schnellste, einfachste Teststufe.
  - Schnelles erstes Feedback «ob es funktioniert».
  - Regression, Basis für jedes Refactoring.
- Wichtig für Continuous Integration (siehe CI-Input).

Was ist ein Unit Test?

- Nicht alles was **J**Unit verwendet ist auch ein Unit Test!
  - JUnit kann zur Automatisation von beliebigen Testarten verwendet werden!

- Definition eines Unit Tests von Roy Osherove (Author von «The Art of Unit Testing»):

> «A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.»

Test Driven Development (TDD)

Die drei Gesetze des TDD:

§1: Produktiver Code darf erst implementiert werden, wenn es dafür einen Unit-Test gibt.

§2: Dieser Unit-Test darf nur gerade so viel Code enthalten, dass er fehlerfrei kompiliert, aber als Test scheitert.

§3: Man ergänzt jeweils nur gerade so viel produktiven Code, dass der scheiternde Test besteht.

- Der Zyklus dieses Ablaufes liegt dabei im Bereich von **Sekunden** bis **Minuten**!
  - Tests und Produktivcode werden praktisch zeitgleich geschrieben; Tests eilen nur wenig voraus.

Tests sauber halten!

- Für Testcode sollen die identischen Qualitätsstandards gelten wie für produktiven Code!
  - Gute Namensgebung, gute Struktur, verständlich ➔ CCD!
- Wer denkt: «Besser schmutzige Tests als gar keine Tests!», der ist auf dem Holzweg!
- Begründung: Testcode lebt länger als produktiver Code!
  - Produktiver Code wird z.B. refactored!
  - Wenn Testcode erodiert, dann haben wir doppelt verloren!
- Testcode «dokumentiert» den produktiven Code.
  - Muss umso mehr wart- und erweiterbar sein!
- Es ist ein weiter Weg von Unit Tests zu **guten** Unit Tests!

Was ist ein sauberer Unit Test?

- Drei Dinge machen einen sauberen Unit Test aus:
  1. Lesbarkeit durch **Klarheit.**
  2. Lesbarkeit durch **Einfachheit.**
  3. Lesbarkeit durch **Ausdrucksdichte.**
- Lesbarkeit (Klarheit, Einfachheit und Ausdrucksdichte) ist bei Testcode noch wichtiger als bei Produktivcode.
  - Mit möglichst wenig Code möglichst viel aussagen.
- Jeder Testfall nutzt das «**B**uild-**O**perate-**C**heck»-Pattern:
  1. Erstellen der Testdaten (Build).
  2. Manipulieren der Testdaten (Operate).
  3. Verifikation der Ergebnisse (Check).
- Vergleiche auch: «**Triple A**»-Pattern: **A**rrange, **A**ct, **A**ssert.

Domänenspezifische Testsprache (DS[T]L)

- Nicht so schlimm wie es klingt, wir bleiben bei der Sprache unserer Wahl!
- Aber: Wir schreiben uns ein domänenspezifisches Set von (typisch statischen) Utility-Methoden, welche…
  - den Testcode kompakter und aussagekräftiger machen.
  - ihrerseits natürlich die API des Testkandidaten verwenden.
- Beispiel für eigene **`assert`**-Methoden:

```
assertResponseIsXML()
assertResponseContainsElement(…)
```

➔ Identischer Ansatz wie bei AssertJ- oder Hamcrest-Library: Möglichst aussagekräftige und leichtverständliche Ausdrücke.

Nur ein `assert` pro Test

- Eine wirklich sehr drakonische Forderung!
  - Nicht immer sinnvoll umsetzbar, aber wenn: Viel besser!
- Weniger (bzw. keine) **`assert`**-Messages notwendig, weil die Testfälle selber schon sehr selektiv sind.
  - Einzelne Testmethoden werden überschaubarer und kleiner.

```
...
assertEquals(soll, ist, "Person verglei…");
...
```

```
@Test
testPersonEquals() {
    assertEquals(soll, ist);
}
```

Nur ein Konzept pro Test

- Auch bei Tests soll man SLA, SRP und SOC einhalten!
  - **S**ingle **L**evel of **A**bstraction (SLA).
  - **S**ingle **R**esponsibility **P**rinciple (SRP).
  - **S**eparation **O**f **C**oncerns (SoC).
- Ein untrügliches Zeichen für Verletzung:
  Eine Testmethode wird in mehrere Abschnitte gegliedert, vielleicht sogar noch mit Kommentarblöcken unterteilt.
- Zu grosse und lange Testmethoden sind mühsam!
  - Viel weniger selektive **`failure`**-Meldungen.
  - Weniger gezielte Wiederholung des Tests möglich.

  ➔Kurz: Alle Nachteile von **zu grossen** Methoden!

F.I.R.S.T. – Prinzip

- **F**ast – Tests sollen schnell sein, damit man sie jederzeit und regelmässig ausführt.
- **I**ndependent – Tests sollen voneinander unabhängig sein, damit sie in beliebiger Reihenfolge und einzeln ausgeführt werden können.
- **R**epeatable – Tests sollten in/auf jeder Umgebung lauffähig sein, egal wo und wann.
- **S**elf-Validating – Tests sollen mit einem einfachen boolschen Resultat zeigen ob sie ok sind oder nicht*.
- **T**imely – Tests sollten rechtzeitig, d.h. vor dem produktiven Code geschrieben werden ➔ Test-First, TDD.

* Beim Einsatz zeitgemässer Unit-Test Frameworks selbstverständlich

Uncle Bob's Unit-Tests Heuristiken

- **T1**: Unzureichende Tests.
- **T2**: Coverage-Werkzeug verwenden.
- **T3**: Triviale Tests nicht überspringen.
- **T4**: Ignorierte Tests zeigen Mehrdeutigkeit auf.
- **T5**: Grenzbedingungen testen.
- **T6**: Bei Fehlern die Nachbarschaft gründliche testen.
- **T7**: Muster des Scheiterns zur Diagnose nutzen.
- **T8**: Hinweise durch Coverage Patterns beachten.
- **T9**: Tests sollen schnell sein.

T1: Unzureichende Tests vermeiden

- Meistens wird nur bis «zum Gefühl dass es reicht» getestet.
- Clean Code fordert:
  Es wird alles getestet, was schief gehen kann!
  - Nach Murphy's Law geht ja auch alles schief…
- Man schreibt so lange Tests…
  - wie es Bedingungen gibt, die noch nicht geprüft werden.
  - Berechnungen stattfinden, die nicht validiert werden.
- Faktisch bedeutet das eine 100%-ige Testabdeckung!
  - Ist das eine realistische Forderung? Nein, nicht überall!
- Aber sicher ein gutes Ziel um Fortschritte zu machen… ☺

T2: Coverage-Werkzeug verwenden

- Coverage-Werkzeuge decken Lücken in den Tests auf.
  - Machen es (sehr) leicht, ungetesteten Code aufzudecken.
- Statement- und Decision-Coverage beachten!
- Beispiel:

```
57.       int quadrant = NO_QUADRANT;
58.       if ((x != 0) && (y != 0)) {
59.          if (x > 0) {
60.             if (y > 0) {
61.                quadrant = QUADRANT_1;
62.             } else {
63.                quadrant = QUADRANT_4;
64.             }
65.          } else {
66.             if (y > 0) {
67.                quadrant = QUADRANT_2;
68.             } else {
69.                quadrant = QUADRANT_3;
70.             }
71.          }
72.       }
73.       return quadrant;
```

Testfälle:

```
assertEquals(1, Geometrie.getQuadrant( 1,  1));
assertEquals(2, Geometrie.getQuadrant(-1,  1));
assertEquals(3, Geometrie.getQuadrant(-1, -1));
assertEquals(4, Geometrie.getQuadrant( 1, -1)); // B1

assertEquals(0, Geometrie.getQuadrant( 0,  1)); // B2

assertEquals(0, Geometrie.getQuadrant( 1,  0)); // B3

assertEquals(0, Geometrie.getQuadrant( 0,  0)); // B4
```

T3: Triviale Tests nicht überspringen

- Manchmal gibt es Tests, die man nicht macht, weil man der Meinung ist, sie seien zu trivial.
  - Setter/Getter, Konstruktoren, einfache Berechnungen…
- Clean Code sagt:
  Dokumentarischer Wert der Testfälle übersteigt die Produktionskosten ➔ es lohnt sich **doch**!
  - Sie vergrössern die Abdeckung.
  - Sind ein Motivationsfaktor, weil einfach zu schreiben.

«Keine Klasse, keine Funktion ist zu klein, um nicht automatisch getestet zu sein!»

T4: Deaktivierte Tests zeigen Mehrdeutigkeit auf

- Test auskommentiert oder (besser) deaktiviert:

```
@Disabled("Gibt es eigentlich einen Nullpunkt?")
@Test
void testGetQuadrantNullpunkt() {
    assertEquals(0, Geometrie.getQuadrant(0, 0));
}
```

- Wenn ein Testfall deaktiviert ist, deutet das häufig auf eine Unklarheit in den Anforderungen hin!
- Deaktivierte Testfälle sind somit ein **Warnsignal**!
  - Temporär ok, aber **nie** als «Providurium».
- Hinweis: Vor JUnit 5 hiess die Annotation **`@Ignore(…)`**

T5: Grenzbedingungen testen

- Testen mit Grenzwerten ist wichtig!
  - Meist wird «die Mitte» eines Algorithmus richtig implementiert, aber seine Grenzen falsch beurteilt.
- Beispiel:

```
public long addition(int sum1, int sum2) {
  long result = (long) sum1 + sum2;
  System.out.println("Addition ergibt: " + result);
  return result;
}
```

- Und der Testfall: PASSED

```
assertEquals( 2L * Integer.MAX_VALUE,
   t.addition(Integer.MAX_VALUE, Integer.MAX_VALUE));
```

- Konsolenausgabe: `Addition ergibt: 4294967294`

T6: Fehler-Nachbarschaft gründlich testen

- Fehler treten oftmals gehäuft auf!
- Findet man in einer Klasse / Funktion einen Fehler sollte man diese erschöpfend testen.
  - Es besteht eine hohe Wahrscheinlichkeit, dass darin noch weitere Fehler gefunden werden.

Murphy's Law sagt:

- Meist ist ein kleiner Fehler nur dazu da, dass sich dahinter ein viel grösserer Fehler verstecken kann.
- Findet man hingegen einen grossen Fehler, wird sich dahinter ein anderer grosser Fehler verstecken.

T7: Muster des Scheiterns zur Diagnose nutzen

- Wenn man genügend (und mit hoher Codeabdeckung) testet, kann man in scheiternden Tests manchmal Muster erkennen!
- Beispiele:
  - Alle Tests scheitern, bei welchen ein String eine bestimmte Länge überschreitet.
  - Alle Tests scheitern, bei welchen ein bestimmtes Argument negative Werte erhält.
- Es soll schon Fälle gegeben haben, wo das deutliche **rot**/**grün**-Muster im Testreport zu «**Aha**»-Erlebnissen geführt hat!

T8: Hinweise von Coverage Patterns nutzen

- Wenn ein Testfall scheitert: Codeabdeckung studieren!
  - Manchmal erkennt man aufgrund der Zeilen die (**nicht**) ausgeführt wurden sehr schnell den Fehler!
- Beispiel: **`Failure`** in JUnit-Test:

```
Punkt in Quadrant 2
expected:<2> but was:<3>
```

- «Pattern» der Codeabdeckung:
  - **`if`**-Zweig wurde nicht ausgeführt!
  - Bedingung ist falsch formuliert!

```
57.     int quadrant = NO_QUADRANT;
58.     if ((x != 0) && (y != 0)) {
59.        if (x > 0) {
60.           if (y > 0) {
61.              quadrant = QUADRANT_1;
62.           } else {
63.              quadrant = QUADRANT_4;
64.           }
65.        } else {
66.           if (y == 0) {
67.              quadrant = QUADRANT_2;
68.           } else {
69.              quadrant = QUADRANT_3;
70.           }
71.        }
72.     }
73.     return quadrant;
```

- Vergleiche mit Beispiel zu T2: Nicht nur die grünen und roten Bereiche ansehen, sondern auch die zusätzlichen Informationen studieren und nutzen!

T9: Tests sollen schnell sein

- Langsame Test werden selten oder gar nicht ausgeführt.
  - Ein Test, der nicht ausgeführt wird, ist nichts Wert.
- Unit-Tests sollten schnell sein, damit man sie immer und jederzeit ausführt.
  - Darum sollten die Tests auch nicht zu gross sein.
- Man soll alles Erforderliche tun, um die Tests zu beschleunigen!
  - ***Need for speed!***

Zusammenfassung – 1/2

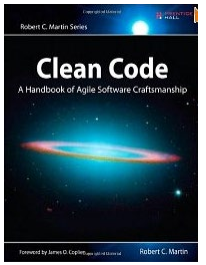
- **Qualität** des Testcodes ist **noch wichtiger** als diejenige vom produktivem Code!
  - Nur dank Testcode getrauen wir uns etwas im produktiven Code zu verändern.
  - Testcode unbedingt «sauber» halten.
- «Build-Operate-Check» - Pattern einhalten.
- Domänenspezifische Hilfsmethoden ergänzen, um die Lesbarkeit zu erhöhen.
- Wenn möglich nur ein **`assert`** pro Testfall.
- Nur ein Konzept (SoC) pro Testfall.
- F.I.R.S.T. – Prinzip!

Zusammenfassung – 2/2

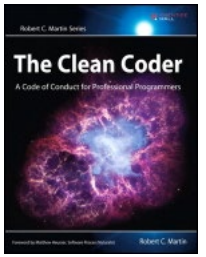
- Wir scheiben viele, kleine, selektive und auch triviale Tests, welche wir schnell, jederzeit und überall ausführen können!
- Wir verwenden Coverage-Werkzeuge um die Testabdeckung zu prüfen und laufend zu erhöhen.
  - Liefern auch bei Test-Failures nützliche Informationen!
- Wenn wir Fehler finden, kontrollieren wir die Nachbarschaft umso kritischer.
  - Fehler treten meist lokal gehäuft auf!
- Wenn wir einen Fehler «einfach so» (ohne Unit-Test) finden, dann schreiben wir noch **vor** dessen Behebung einen Unit-Test, welcher diesen Fehler zuverlässig aufdeckt!
  - Einen Fehler **nie** zweimal machen (oder ausliefern)!

Literaturhinweise / Quellen

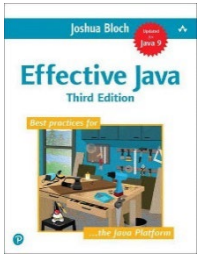
- Robert C. Martin: **Clean Code**
  A Handbook of Agile Software Craftsmanship
  Prentice Hall, März 2009
  ISBN: 0132350882 / EAN: 9780132350884



- Robert C. Martin: **The Clean Coder**
  A Code of Conduct for Professional Programmers
  Prentice Hall International, Mai 2011
  ISBN: 0137081073 / EAN: 9780137081073



- Joshua Bloch: **Effective Java**
  Best practices for the Java Plattform
  Pearson Addison-Wesley, Third Edition, Dezember 2017
  ISBN: 978-0-13-468599-1



HSLU Hochschule Luzern

Fragen?