JUG Vortrag Zürich, 6. Juni 2024

Jenseits von "Schreibt fleissig Unit Tests"

Effektive Teststrategien für stabile Software

Daniel Keller

Inhalt

Unit Testing ist ein hervorragendes Konzept

- Unterscheidung Microtests vs. Integrationstests
- Aufruf-Hierarchien, Schichten-Architektur und Faking & Mocking
- Microtesting hat mehrere Schwächen.
- Integrationstests – wenn richtig gemacht – sind stärker.
- Wie man bessere Testfälle findet (Boundaries, Condition Coverage, bewusst eingestreute Fehler)
- 100% Testabdeckung ist gut, ist aber erst der Anfang.
- Wann ist "genug getestet"? (Kurzantwort: Reviews)
- Unit Testing auch für die Datenbank (automatisierte Konsistenztests)

Unit Testing: Klar!

Zu testender Code:

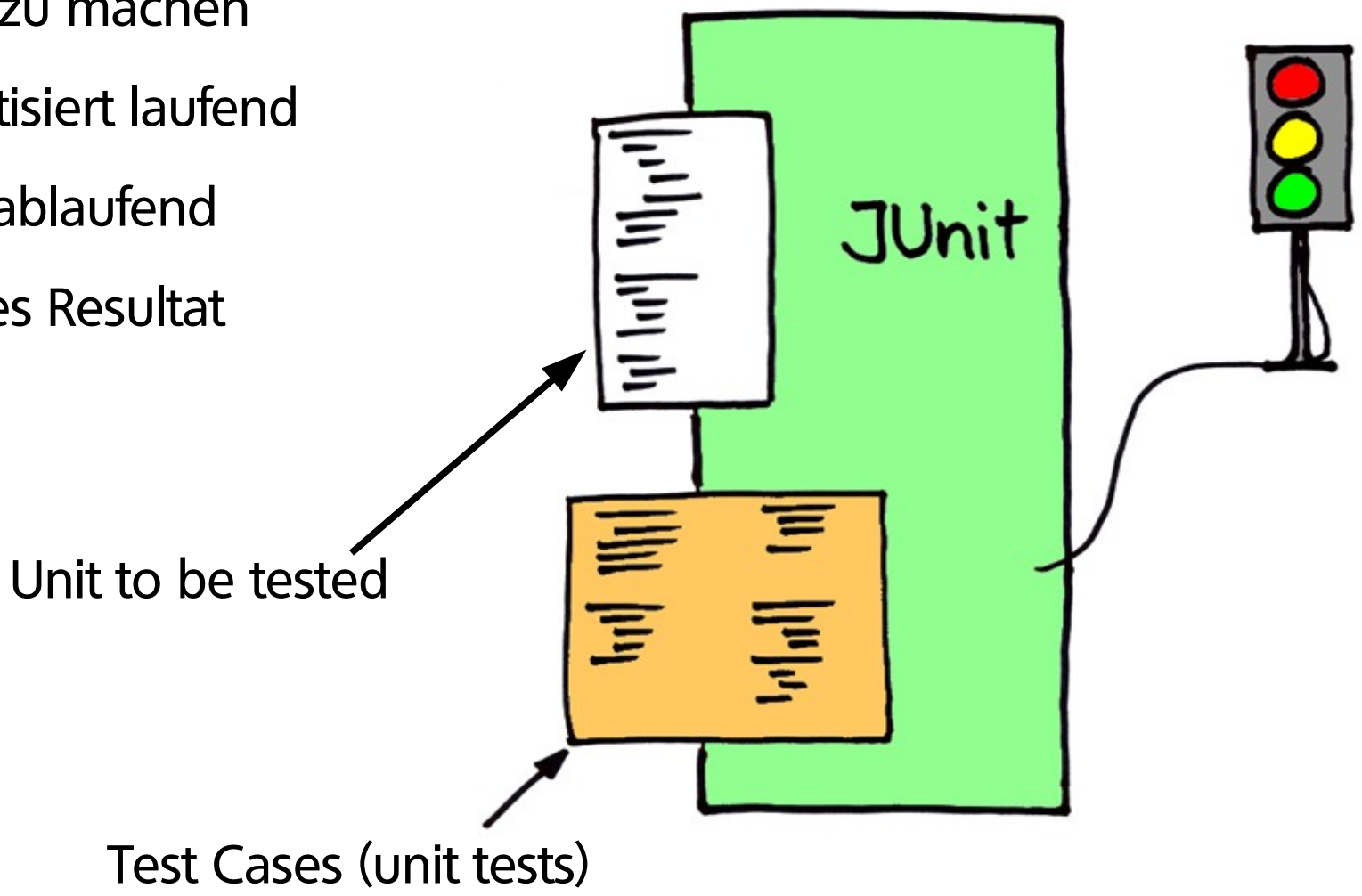
```
public static String floatWithTwoDigits( float f ) {
    return String.format( "%6.2f", f );
}
```

Unit Tests zu 'floatWithTwoDigits'

```
public void testFloatWithTwoDigits() {
    f = 34.5678F;
    expResult = " 34.57";
    result = Helper.floatWithTwoDigits(f);
    assertEquals(expResult, result);
}

public void testFloatWithTwoDigitsRounding() {
    float f = 2.009F;
    String expResult = "  2.01";
    String result = Helper.floatWithTwoDigits(f);
    assertEquals(expResult, result);
}

public void testFloatWithTwoDigitsLargeNegative() {
    f = -456.789F;
    ...
    ...
```

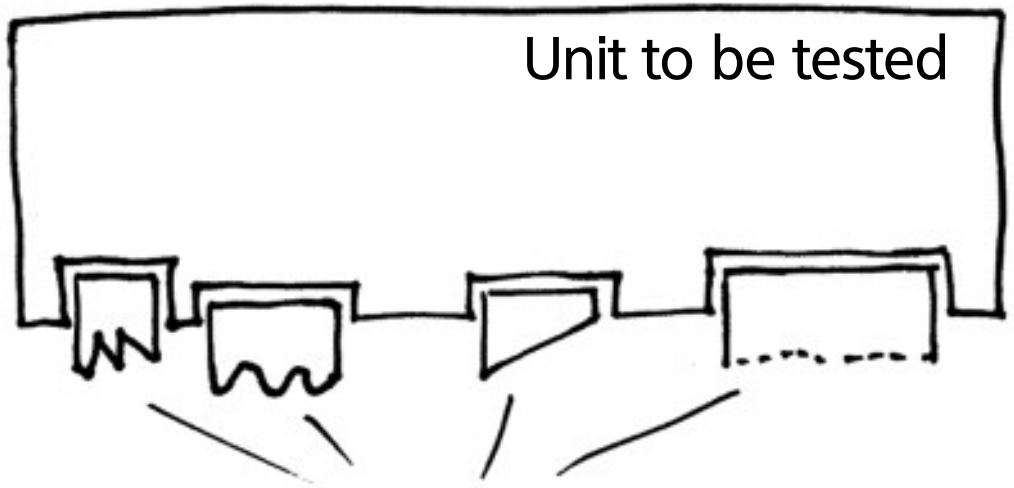
Prinzip: Unit Testing

- einfach zu machen
- automatisiert laufend
- schnell ablaufend
- einfaches Resultat



Faking & Mocking

- Fakes (manchmal ‘Stubs‘ genannt); simpler Ersatz, meist fixe Daten
- Mocking: a) intelligente Rückgabewerte, b) schauen/verifizieren, wie das gemockte Teil aufgerufen wird.



Beispiele:
DB-Aufrufe, File System,
3rd party Software
(SAP, SalesForce, ...)

External System: To Mock or not to Mock?

File-Synchronisation via FTP Server: Dateien hochladen, runterladen

Mocken? den FTP-Server? vielleicht, aber eher nicht

Laptop mit Linux hinstellen, FTP Server starten, lokal testen via Ethernet-Kabel. Test des kompletten Systems in kontrollierter Umgebung, Test-Szenarien einfach aufzusetzen (‚Copy Directory')

Seltsame Effekte mit z.B. Dateinamen welche Leerzeichen enthalten tauchen dann tatsächlich auf (ein Mock würde das vermutlich übersehen). Ebenso beispielsweise Timeout-Probleme.

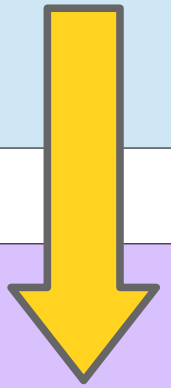
Externes System, beispielsweise Börsenwerte

Faken, Mocken? ja, klar, wir wollen ja bestimmte Szenarien testen, z.B. „steigende Pharma-Aktien"

Kleiner Ausflug: Aufruf-Hierarchien

Kunde

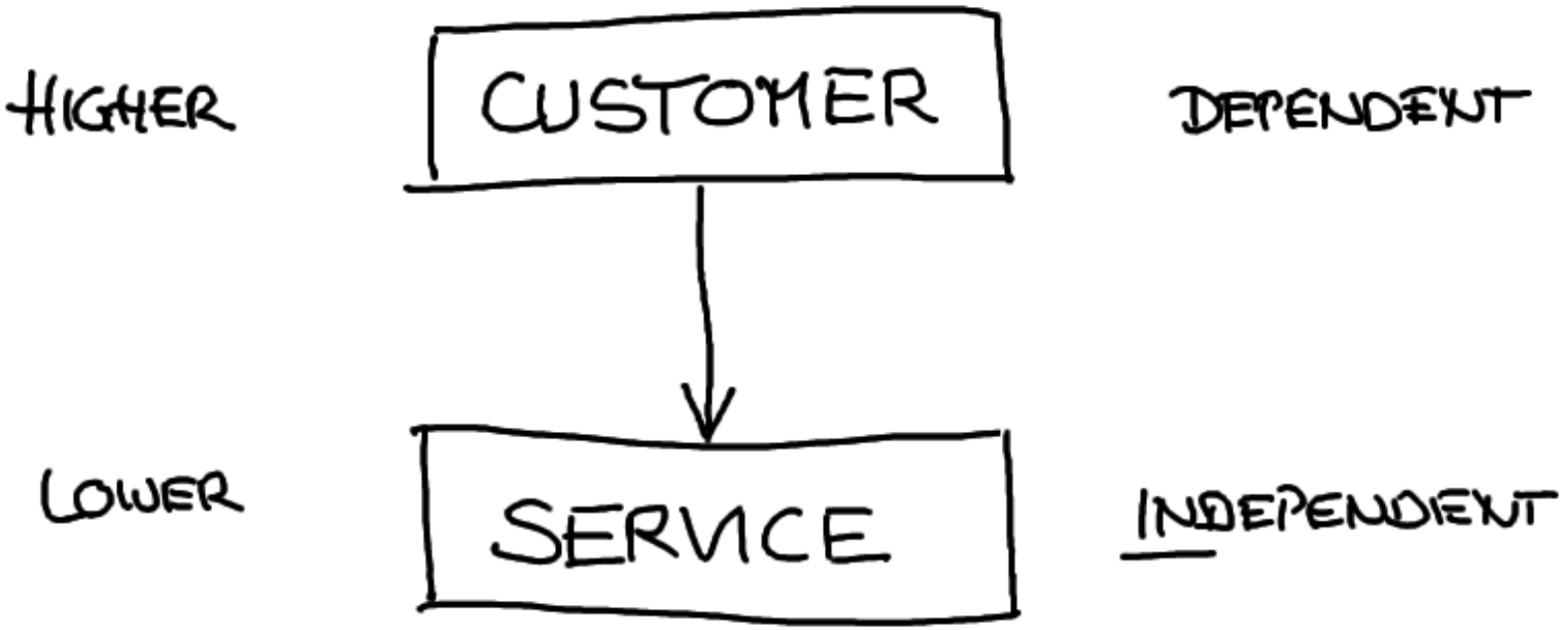
```
import services.DateHelper;

   ...

   workDays = DateHelper.calculateDiffInWorkingDays( beginDate, endDate );

   ...
```



Dienstleistung

```
public class DateHelper {

   ...

   public static int calculateDiffInWorkingDays( Date first, Date second ) {
      ...

      return workingDays;
   }
```

Abhängig und unabhängig



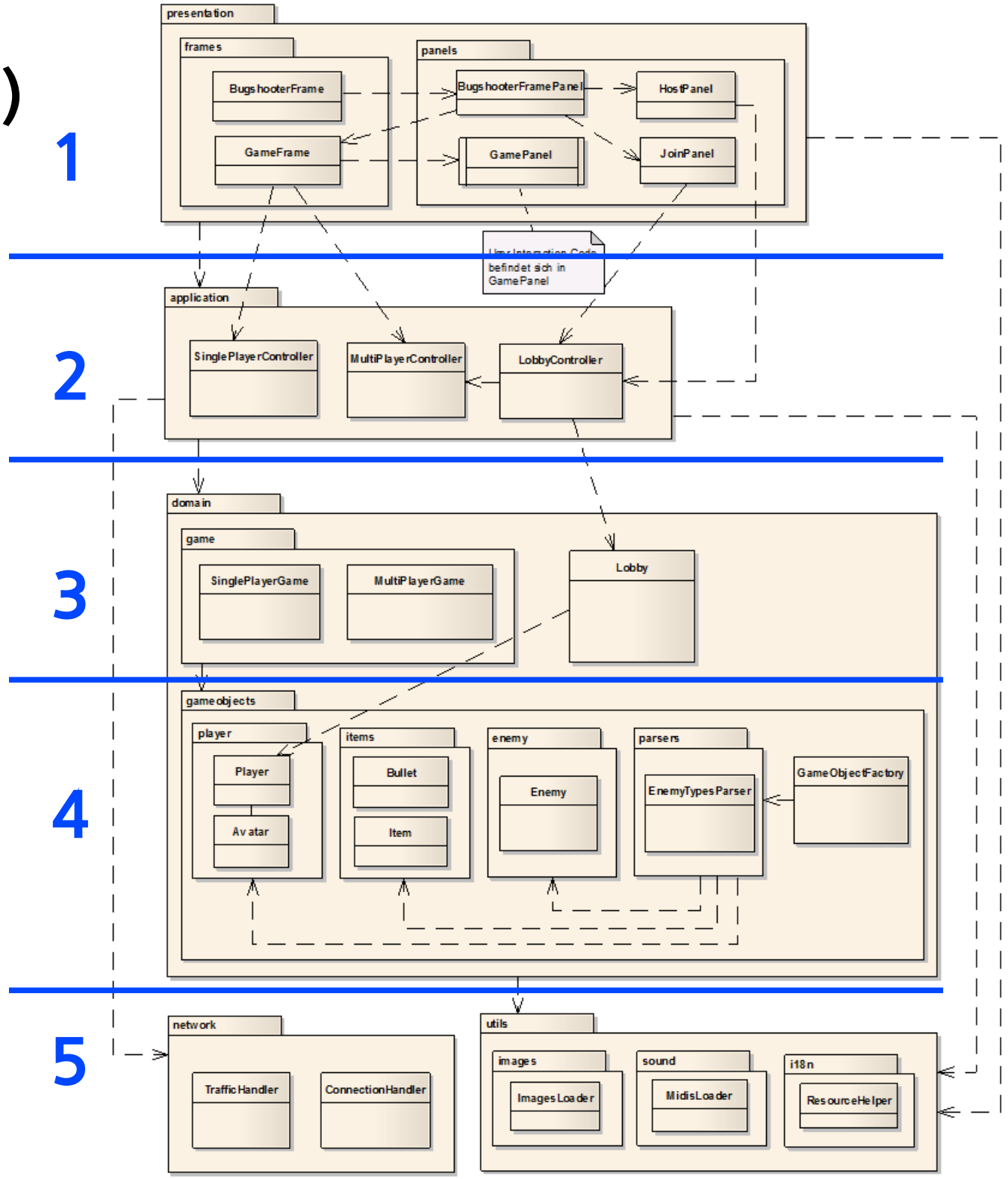
Schichten sind asymmetrisch!

Abhängigkeit beachten.

Layers (Schichten)

Alle Software mit mehr als 2000 Zeilen kann/soll in Schichten angeordnet werden.

Das ergibt eine Hierarchie von Abhängigkeiten und eine klarere Architektur.



Aufbau der Schichten, 4 Beispiele

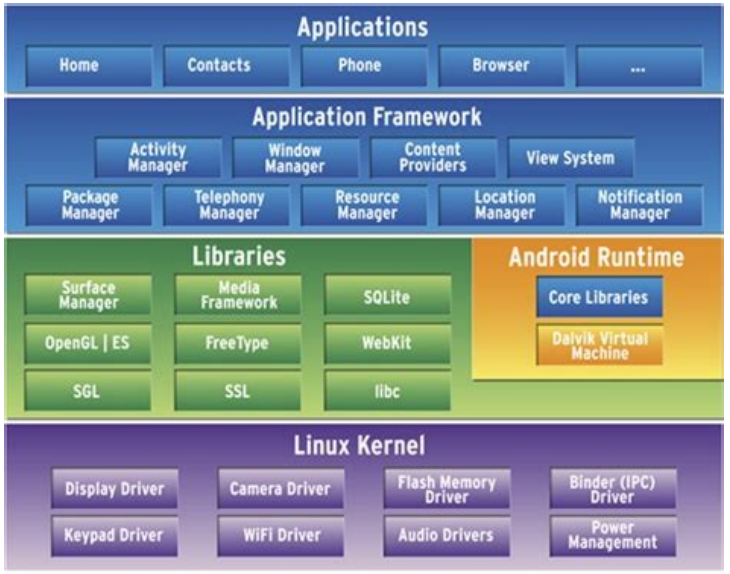
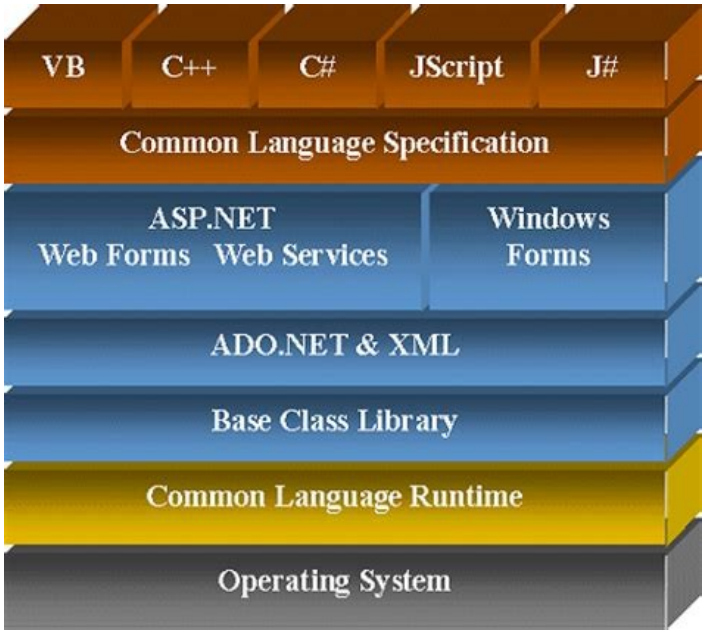
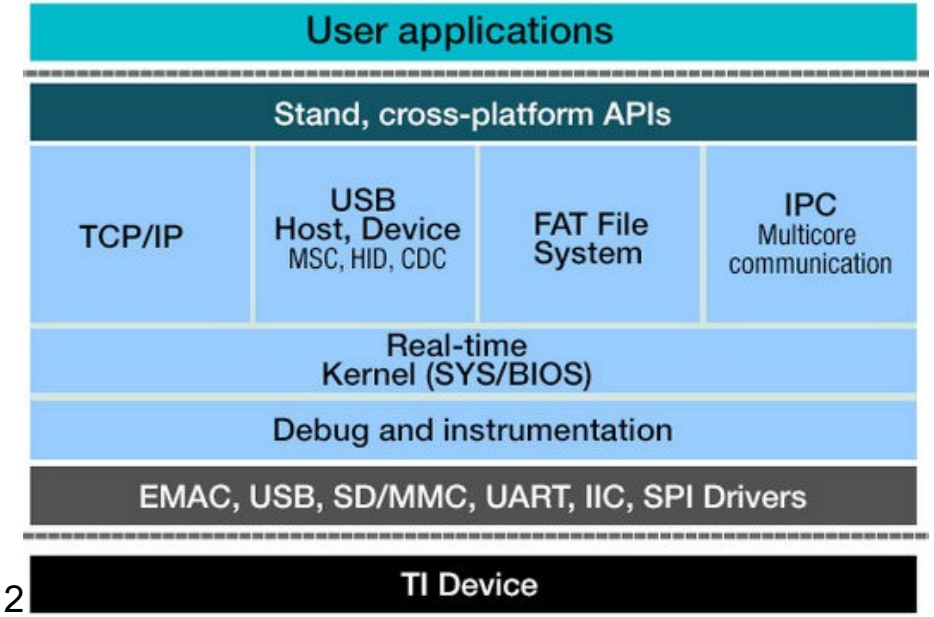


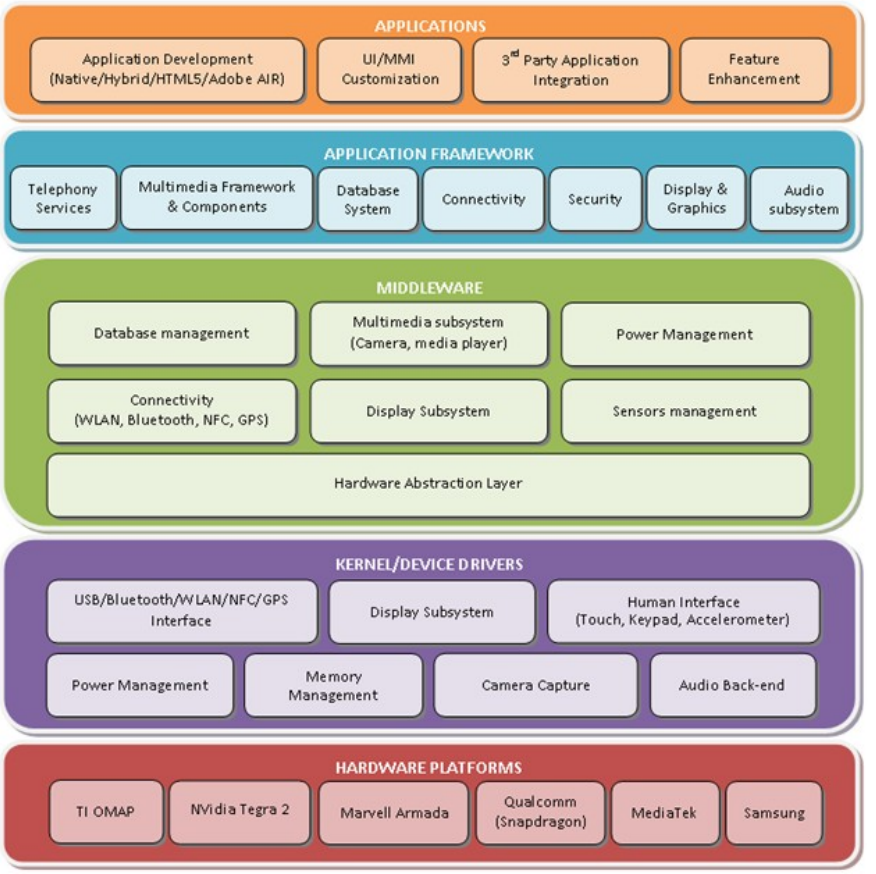
Fig 1. The Android system architecture uses a standard software-stack approach with Linux as its base.

oben

unten







Begriff 'Microtesting'

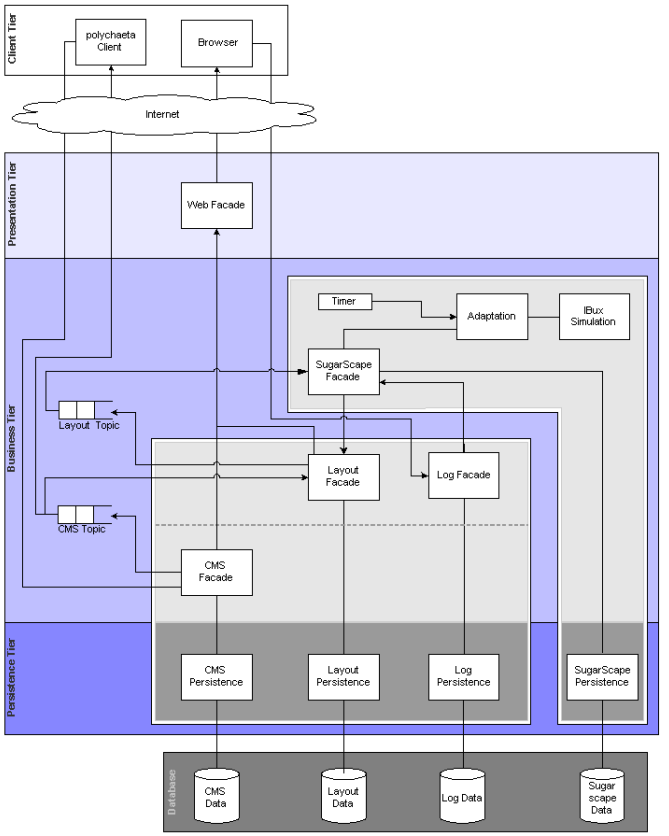
Microtests sind automatisierte Unit Tests

- Code Unit to be tested = Methode (seltener ganze Klasse)
- isoliert d.h. oft mit Faking & Mocking
- schnell ablaufend, wenige Sekunden
  damit man {test – change/improve code – test} oft machen kann,
  und damit Continuous Integration auch tatsächlich funktioniert

Microtesting eigentlich nur auf unteren Ebenen

Microtests (isolierte, schnell laufende Unit Tests) funktionieren nur gut auf den untersten Schichten einer Architektur.

Am einfachsten sind Libraries mit Microtests zu versehen, denn Libraries haben keine weiteren Abhängigkeiten (ausser vielleicht Computersprache und eventuell Betriebssystem).

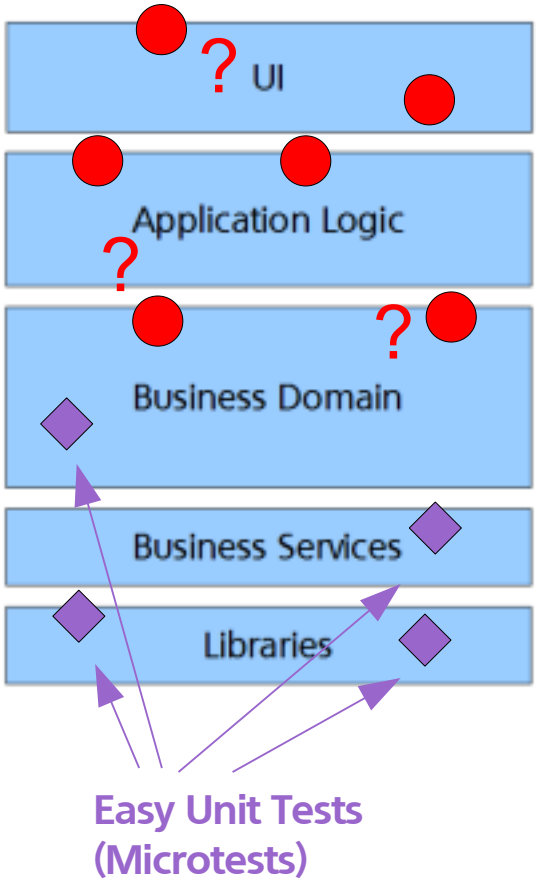


Unit Testing auf allen Ebenen

Unit Tests auf höheren Ebenen funktionieren nicht so gut, weil die getesteten Methoden fast immer viele Abhängigkeiten nach unten haben.

Diese Abhängigkeiten müssen durch Faking & Mocking eliminiert werden.

Faking & Mocking sind immer Vereinfachungen und enthalten Annahmen – Annahmen, die sich als falsch (oder zumindest einschränkend) herausstellen können, aber oft nicht bemerkt werden.
Zudem sind Fakes & Mocks zusätzlicher Code mit entsprechendem Wartungsaufwand.



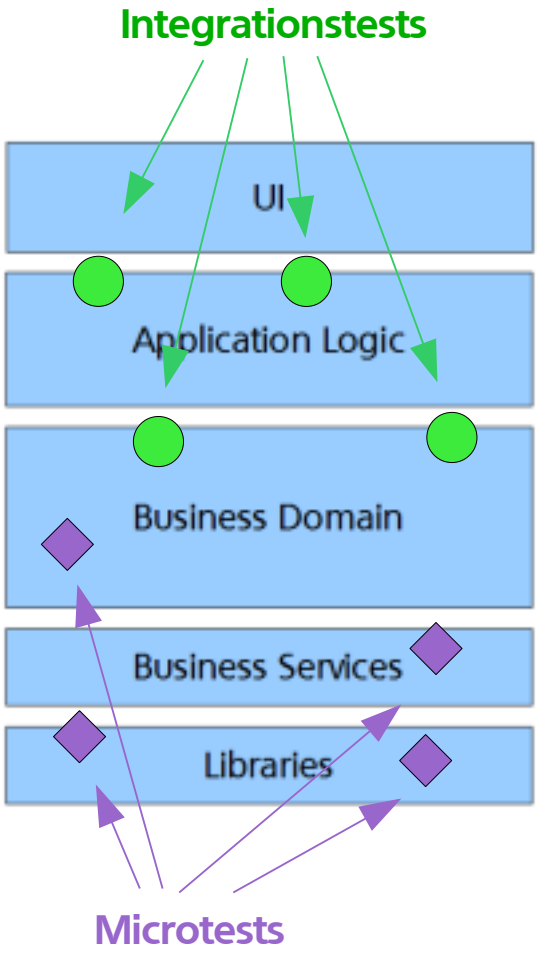
Integrationstests = Higher-level Unit Testing

Integrationstests testen das Zusammenspiel von Klassen - dort wo viele Missverständnisse entstehen können, trotz static type checking (Reihenfolge der Calls, Wertebereich von Parametern, Fehlerbehandlung, ...)

Bei Integrationstests versucht man, so wenig wie möglich zu Mocken.

Integrationstests auf oberster Ebene (direkt unter UI) testen das System als Black Box: früh definiert z.B. mit Sequenzdiagrammen, Contracts. Integrationstests direkt unter dem UI testen die Funktionalität des Systems, so wie es von aussen definiert wurde.

Integrationstests sind langsamer.



Beispiel Integrationstest

Rufen Sie in einem Unit Test z.B.

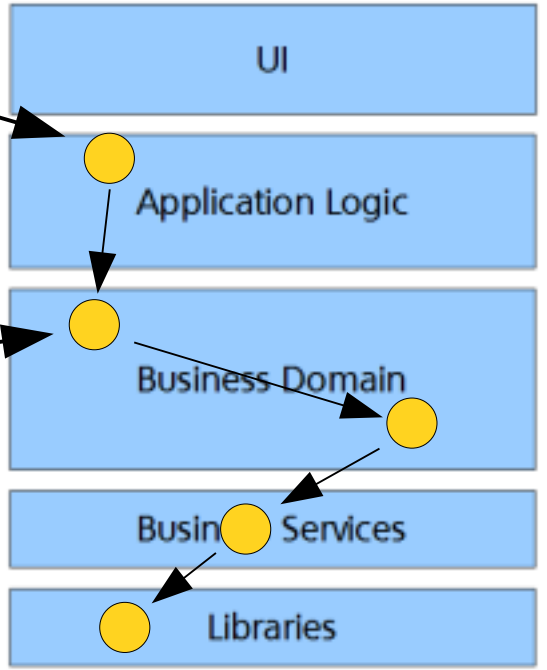
```
canCustomerPayByInvoice()
```

auf.

Dieser Aufruf wird dann eine ganze Reihe anderer Routinen aufrufen, z.B.

```
getPaymentHistoryOfCustomer()
```

bis hinunter zur Datenbank. *)
Und wenn alles korrekt zurückgeliefert wird, dann ist dieser Unit Test (eben ein Integrationstest) grün.

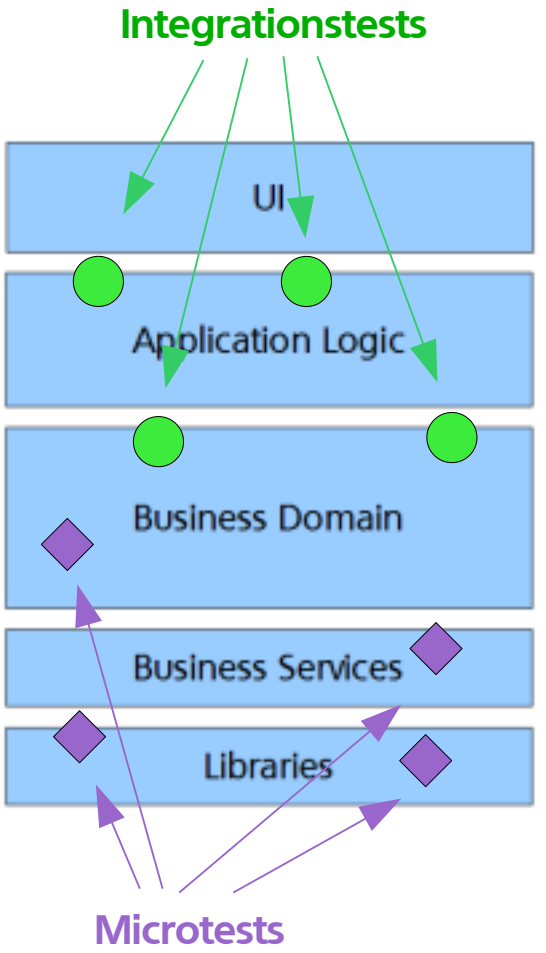


*) In die Datenbank müssen – damit das funktioniert – zuerst die richtigen Testdaten eingeschoben werden, sonst ist der Test nicht reproduzierbar

Fokus von Microtests vs. Integrationstest

Integrationstests testen eher die Anforderungen für das System.

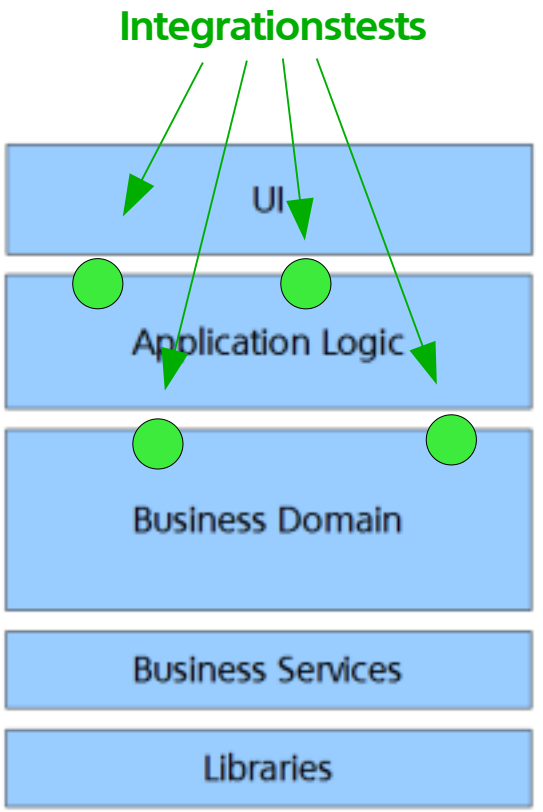
Microtests fokussieren auf die Implementation, auf Details der Lösung.



Vorteile von Integrationstests

Integrationstests sind in der Praxis oft deutlich wertvoller als Microtests. Gründe:

- Integrationstests entdecken mehr Fehler, weil sie realistischere Szenarios testen.
- Integrationstests sind langlebiger als Microtests, d.h. Integrationstests müssen nicht so oft umgeschrieben werden.
- Integrationstests können einen Teil der nicht-automatisierbaren End-To-End Tests (UI, Browser) ersetzen – und die verbleibenden konzentrieren sich dann rein auf die Darstellung.
- Integrationstests bieten die Möglichkeit des Einsatzes einer DSL (Domain Specific Language) und Einbindung der User beim Formulieren von automatischen Tests (z.B. cucumber.io). Das könnte sich bei Integrationstests lohnen.



- Integrationstests führen nicht zum ‚lokales Minimum' Effekt (s. unten)

Nachteil von Integrationtests

Integrationtests können hohe Laufzeiten zur Folge haben, d.h. ein Build mit ausführlichen Tests dauert länger als ein paar Minuten, u.U. länger als eine Stunde. Echte Continuous Integration (build/test nach jedem Commit) ist dann nicht mehr machbar.

Das ist aber m.E. der einzige wirkliche Nachteil von Integrationtests.

Beispiel für zeitintensive Integrationstests:
DB-Szenario unten hineinschieben (ist jedesmal im Bereich von 20 Sekunden bis Minuten).
Dann lässt man ein paar Unit Tests in Kombination laufen.
Danach: nächstes Szenario.

Mögliche Lösung: Integration Server macht alle zwei Stunden einen Lauf.

Oder „mixed" CI: alle schnellen Tests laufen bei jedem Commit, alle langsamen Tests laufen zusätzlich jeweils nachts.

Microtesting: Grenzen und Kritik

… und dann lässt man die Teile liegen und sagt „es läuft alles, es ist alles sauber getestet“

Microtesting: Grenzen und Kritik

Ist Ihnen aufgefallen, dass auf dem vorigen Bild kein Motor zu finden war?

Limitationen von Microtests

Limitationen von Microtests:

- Funktionieren nur auf den untersten Hierarchie-Ebenen gut.
- Testen nicht das Zusammenspiel von Komponenten.
- Gröberes Refactoring macht viele Unit Tests kaputt.
- Sind anfällig für lokale Minima.
- Unit Tests sind (leider oft) nichtssagend (”2+2 = 4“)
  ... und dann fügt jemand noch einen Test ”3+3 = 6” hinzu :-(
- Selbst eine sehr hohe Testabdeckung sagt noch nicht viel aus
  – aber man fühlt sich sicher.

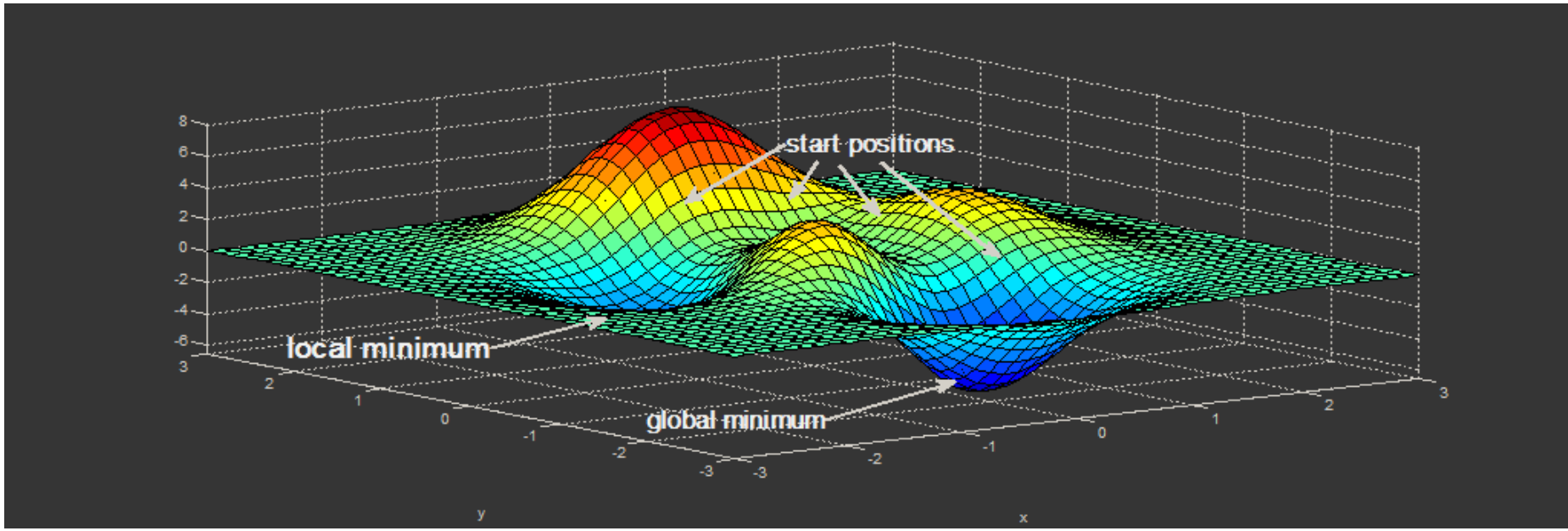
(Erklärungen folgen gleich)

Schwachstelle Microtesting: Lokales Minimum

Unit Tests verleiten einen beim Refactoring oft dahingehend, dass man wie auf einem lokalen Minimum stecken bleibt – man kommt nicht auf die bessere Lösung, welche gröberes Refactoring verlangen würde.

Und gröberes Refactoring macht man nicht "weil alle Unit Tests gerade so schön grün sind".

Bild: http://sf.anu.edu.au/~vvv900/gaussian/ts/opt-to-minima.html



Auf amerikanisch heisst das.: "You might paint yourself into a corner"

Aussageschwache Unit Tests "2+2 = 4"

Manchmal testen Unit Tests etwas Unnötiges:

- Tests von Funktionen weit verbreiteter Libraries. Da mache ich erst Tests, wenn ein Verdacht auf Fehlfunktion vorliegt.
- Tests, die aller Voraussicht nach *immer* 'richtig' zurückgeben werden (eben z.B. "2+2 = 4", oder "ich erstelle ein Objekt und schaue danach, ob es existiert"). Wenn schon, dann soll man schauen, dass das Test-Setup oder die Wahl der Testdaten einen Fehler provozieren könnten, z.B. Division by Zero oder Out of Memory.
- Denken Sie darüber nach, ob Sie random Tests einsetzen könnten:
  "Mutation Driven Testing – When TDD Just Isn't Good Enough"
  https://software.rajivprab.com/2021/02/04/mutation-driven-testing-when-tdd-just-isnt-good-enough/
- Interessanter (ketzerischer) Artikel von James O Coplien:
  "Why Most Unit Testing Is Waste"
  http://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf
  Dieser Link funktioniert leider nicht mehr, man muss googeln und dann findet man vielleicht:
  https://pythontest.com/strategy/why-most-unit-testing-is-waste/

Unit Tests für eine Library

Unit Tests für eine (neu eingesetzte) Bibliothek können aus zwei Gründen sehr sinnvoll sein:

A) Wir haben konkrete Beispiele, wie man die Bibliothek richtig aufruft. Alle können daraus lernen.

B) Wir schützen uns vor potentiellen Änderungen der Funktionalität in zukünftigen Versionen der Bibliothek. Das kann sehr wertvoll sein.

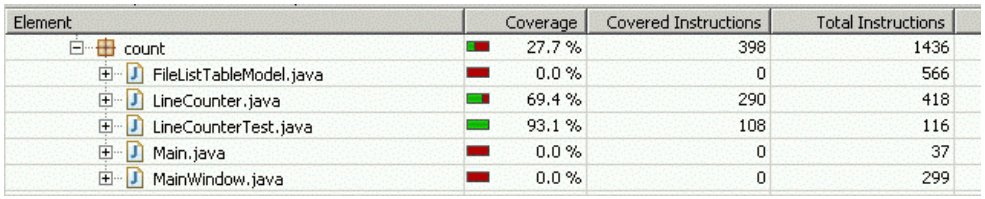
Aber diese Tests müssen nicht bei jedem Commit laufen, da genügt sicher ein Lauf pro Tag.

Testabdeckung

Arten der Testabdeckung, leicht vereinfacht:

- Anweisungsabdeckung (Statement Coverage)
- Zweigabdeckung (Branch Coverage)
- Bedingungsabdeckung (Decision Coverage)
- Pfadabdeckung (Path Coverage)

Anweisungsabdeckung



Element	Coverage	Covered Instructions	Total Instructions
count	27.7 %	398	1436
FileListTableModel.java	0.0 %	0	566
LineCounter.java	69.4 %	290	418
LineCounterTest.java	93.1 %	108	116
Main.java	0.0 %	0	37
MainWindow.java	0.0 %	0	299

Anweisungsabdeckung heisst:

- Jede Anweisung, welche durch die Unit Tests ausgeführt wird, zählt.
- Der Prozentsatz wird so berechnet:
  "Ausgeführte Anweisungen / Gesamtzahl der Anweisungen"

Die Anweisungsabdeckung ist der Standard bei den Testabdeckungs-Verfahren. Praktisch alle mir bekannten Tools messen nur die Anweisungsabdeckung.

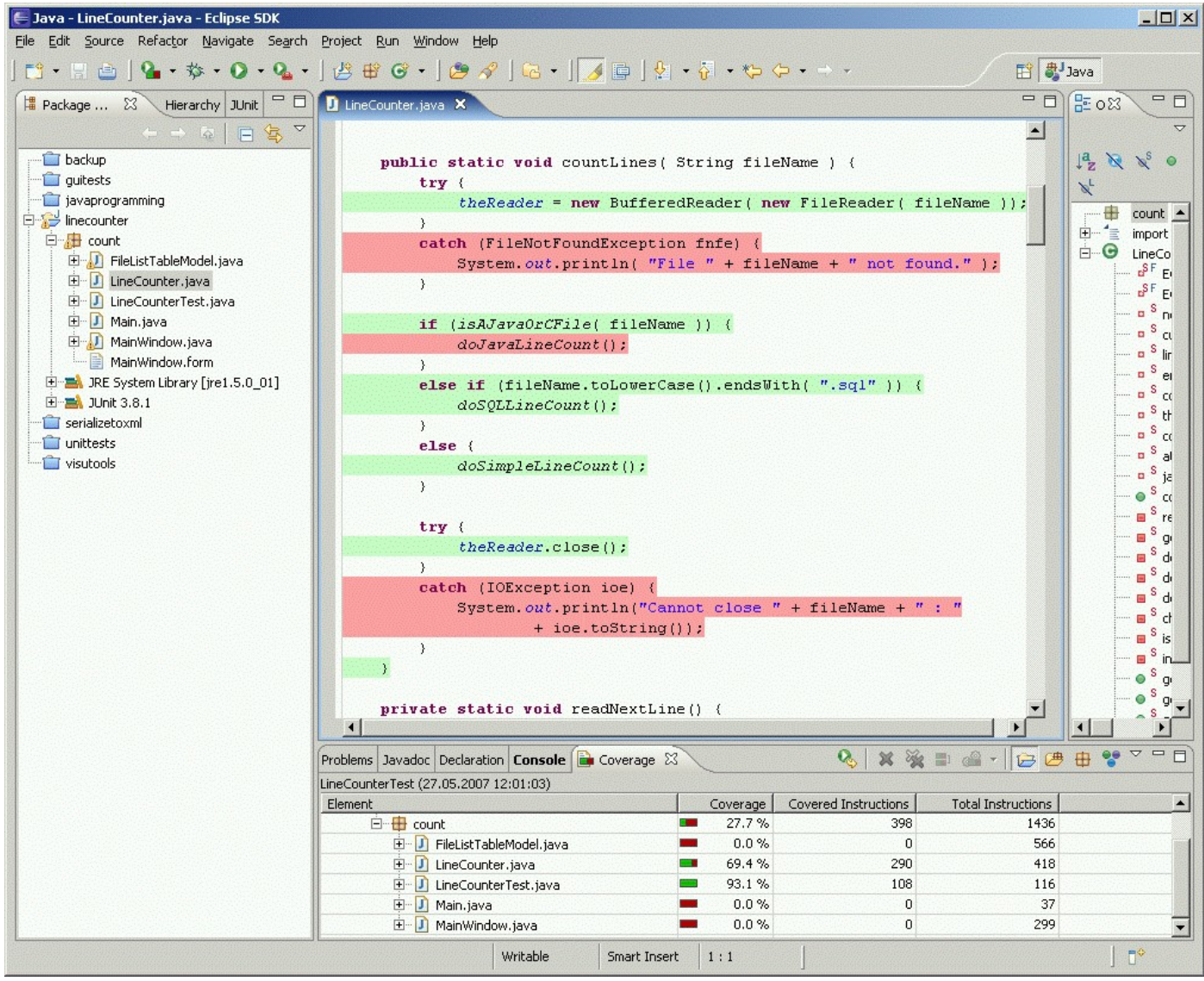
Achtung: die Anweisungsabdeckung ist die minimalste und am wenigsten aussagekräftige aller Testabdeckungs-Verfahren (siehe folgende Folien). Dafür ist sie einfach zu ermitteln.

So funktioniert die Messung der Anweisungsabdeckung

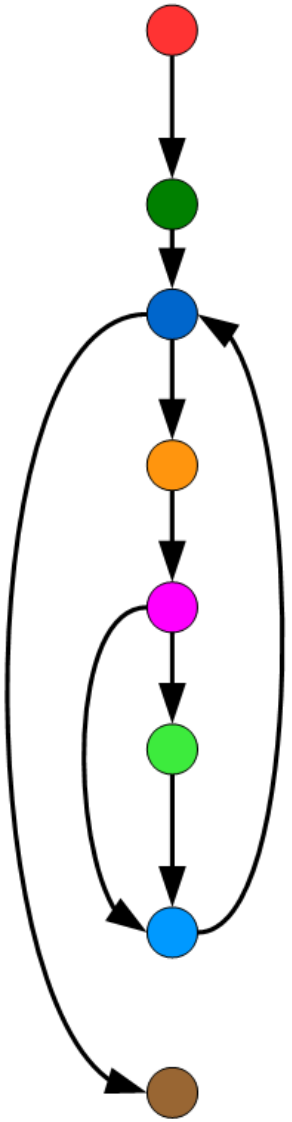
Der gesamte Code (Klassen + Unit Tests) wird mit Zählern auf jeder Anweisung (Zeile) instrumentiert.

Die Unit Tests werden auf dem zu testenden Code ausgeführt.

Die Zähler werden ausgewertet und der Code schön dargestellt: (Rot heisst: "Zähler ist Null"), plus Statistik.



Zweigabdeckung: Kontrollflussgraph



```
void zaehleZeichen(int &VokalAnzahl, int &Gesamtanzahl) {

   char Zchn;

   cin >> Zchn;

   while ((Zchn >= 'A') && (Zchn <= 'Z') &&
                                    (Gesamtzahl < INT_MAX)) {

      Gesamtzahl = Gesamtzahl + 1;

      if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||

          (Zchn == 'O') || (Zchn == 'U')) {

         VokalAnzahl = VokalAnzahl + 1;

      }

      cin >> Zchn;

   }

   return;

}
```

Beispiel zaehleZeichen() aus [Balzert98]

Zweigabdeckung

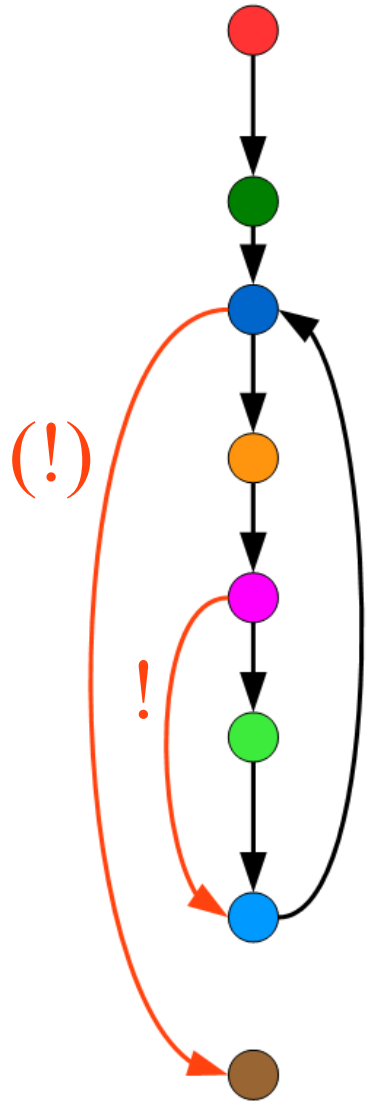
Zweigabdeckung heisst:

- Jeder Zweig, welcher durch die Unit Tests ausgeführt wird, zählt.
- Der Prozentsatz wird so berechnet:
  "Abgedeckte Zweige / Gesamtzahl der Zweige"

Hauptsächlichste Abweichung zu Anweisungs-Abdeckung:

- IF ohne ELSE mit Testfall, wo das IF grad *ganz ausgelassen* wird
- FOR und WHILE Testfälle bei denen man *gar nicht in* die Schleife geht

Zweigabdeckung im Graphen




```
void zaehleZeichen(int &VokalAnzahl, int &Gesamtanzahl) {
   char Zchn;
   cin >> Zchn;
   while ((Zchn >= 'A') && (Zchn <= 'Z') &&
                                    (Gesamtzahl < INT_MAX)) {
      Gesamtzahl = Gesamtzahl + 1;
      if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||
          (Zchn == 'O') || (Zchn == 'U')) {
         VokalAnzahl = VokalAnzahl + 1;
      }
      cin >> Zchn;
   }
   return;
}
```

Beispiel zaehleZeichen() aus [Balzert98]

Bedingungs- und Pfadabdeckung

Bedingungsabdeckung: *"Das Verhältnis von ausgewerteten atomaren Werten (Term, Bedingung, ...) innerhalb von Ausdrücken zu allen vorhandenen atomaren Werten in einem Modul."*

Pfadabdeckung: *"Das Verhältnis der getesteten Pfade (Wege im Kontrollflussgraph), zu allen möglichen Pfaden in einem Modul."*

(Zitate aus: http://robert.bullinger.over-blog.com/article-code-abdeckung-99961701.html)

Das Kriterium der Bedingungsabdeckung kann sehr hilfreich sein, noch weitere Testfälle zu entdecken (Thema: Testdaten-Selektion mit Aequivalenzklassen).

Pfadabdeckung führt sehr schnell zu einer potentiell unendlichen Anzahl von Fällen (wegen Schleifen).

Testdaten für minimale Abdeckung

Anweisungsabdeckung	"A"
Zweigabdeckung	"AB" (vermutlich auch "&")
Bedingungsabdeckung	"$", "G", "{" "A", "E","I", "O", "U", "T"

Ist das gut: „83% Test-Abdeckung“?

100% Anweisungsabdeckung der gesamten Code-Basis kann fast nie erreicht werden. Gründe:

- User Interface Code kann i.d.R. nicht Unit getestet werden.
- Es gibt Code-Teile, die nur extrem schwer mit Unit Tests abzudecken sind, z.B. Exceptions: wie fake ich eine „Disk full“ Exception oder eine „Connection lost“ Exception? (wohlgemerkt: zuverlässig wiederholbar während eines Unit Tests)

Was ist hier falsch?

```
// Input:  3.20 hours, Output "3h12"
// Input:  0.20 hours, Output "12min."
```

```
public static String hoursInFloatToString( float f ) {
    int hours = (int)f;
    int minutes = Math.round((f-hours) * 60);
    StringBuilder sb = new StringBuilder();
    if (hours > 0) {
        sb.append( hours );
        sb.append( "h" );
        if (minutes < 10) {
            sb.append( "0" );
        }
        sb.append( minutes );
    }
    else {
        sb.append( minutes );
        sb.append( "min." );
    }
    return sb.toString();
}
```

Zwei Testfälle liefern 100% Anweisungs-Abdeckung.

Aber es stecken zwei Fehler im Code.

Erster Fehler in `hoursInFloatToString()`

Nicht so subtiler Fehler: negative Zahlen.

Eingabe -1 führt zur Ausgabe '0min.' statt wie vermutlich gewünscht '-1h00'

```
@Test
public void test_negative() {
    assertEquals("-2h30", floatToHoursAndMinutes(-2.5f));
    // junit.framework.AssertionFailedError: expected:<-2h30> but was:<-30min.>
}
```

Zweiter Fehler in `hoursInFloatToString()`

Subtiler Fehler: Ein Rundungs- bzw. Überlauf-Fehler tritt auf:
Eingabe 6.999, Ausgabe '6h60' statt '7h00'

Wenn die Eingabe 0.992 ist, dann ist die Ausgabe: "60min."
ist vielleicht OK, weil die Antwort 60 Minuten (statt 1h00) einigermassen akzeptabel - oder eventuell sogar so gewünscht ist.

```
@Test
public void test_round_to_60() {
    assertEquals("7h00", floatToHoursAndMinutes(6.999f));
    // junit.framework.AssertionFailedError: expected:<7h00> but was:<6h60>
}
```

Wann ist genug getestet?

Haben wir genug Unit Tests?

Oder sollten wir mehr testen?

Es gibt mehrere Antworten auf diese Fragen.

- “Wir können nicht mehr Zeit für Tests aufwenden“
- “Mehr als 40% unseres Codes sind Unit Tests“
- “Wir haben schon seit fünf Tagen keine Fehler mehr gefunden“
- “Wir haben 83% Testabdeckung erreicht“

Welche der Antworten ist/sind gut?

Starke Aussagen – leider im Negativen:

- "Ca. 4% unseres Codes sind Unit Tests"
- "Wir haben 23% Testabdeckung erreicht"
- "3% unseres Aufwandes ging ins Testen, inkl. Usability Tests"

Wenig aussagekräftig (zwar positiv, aber ...):

- "45% unseres Codes sind Unit Tests"
- "Wir haben 93% Testabdeckung erreicht"
- "40% unseres Projektaufwandes war für's Testen"

Diese Art Aussagen zu "haben wir genug getestet?" eignen sich nur für den *negativen* Fall, als Indikatoren (immerhin), dass zuwenig getestet worden ist.

Als positive Aussage "jetzt haben wir genug getestet" eignen sie sich nicht.

Der beste Rat: machen Sie einen Review der Tests (Unit, Integration, System, Usability, Security...)

Testabdeckung? JA! ... aber

Eine hohe Testabdeckung ist eine notwendige aber nicht hinreichende Bedingung.

Also: streben Sie eine hohe Testabdeckung an, aber seien Sie mit der Zahl allein nicht zufrieden.

a) Machen Sie lieber gute Integrationstests, nicht nur Microtests.

b) Auf die Qualität der Testfälle kommt es an, nicht auf die Abdeckung, und wie so oft gilt: Qualität kann man nur mit Peer-Reviews erreichen.

Machen Sie es zur Regel, dass bei einem Code Review auch die zugehörigen Unit Teste gereviewt werden.

Unit Tests verbessern

Sorgfältige und kreative Auswahl der Testdaten:

- Mehrere Normalfälle, zählen Sie wie ein Fisch: "0, 1, 2, viel"
- Untere und obere Grenze bei Bereichen
- 'Leere Liste'
- Ausserhalb des Bereichs: ⇓ unterer und ⇑ oberer Grenze
- NULL statt Objekt
- String mit lauter Leerzeichen (statt eines Namens)
- Falls nicht-typisierte Sprache: String statt Integer

Testfälle für `hoursInFloatToString()`

```
normal:  0.5 -> 30min.
normal:  1.5 -> 1h30
normal:  1.7 -> 1h42
normal:  2.0 -> 2h00
vornull: 0.1 -> 6min.
vornull: 1.1 -> 1h06
bruch:   1.6666 -> 1h40
rundung: 0.5833 -> 35min.
rundung:  1.499 -> 1h30
rundung:  1.501 -> 1h30
rundung:  1.51  -> 1h31
grenze:   0.0 -> 0min.
grenze: 1.001 -> 1h00
grenze: 6.999 -> 7h00
negativ: -3.5 -> -3h30
```

Zur Testdaten-Wahl

Die Testfälle für gute Pfad- und Bedingungs-Abdeckung können zu aussagekräftigen Tests führen. Schauen Sie sich die Bedingungen im Code an und denken Sie sich Testfälle dafür aus. Achten Sie besonders auf zusammengesetzte Bedingungen (&&, ||).

```
if (remoteFile.isDirectory() && !remoteFile.getName().startsWith("."))
```

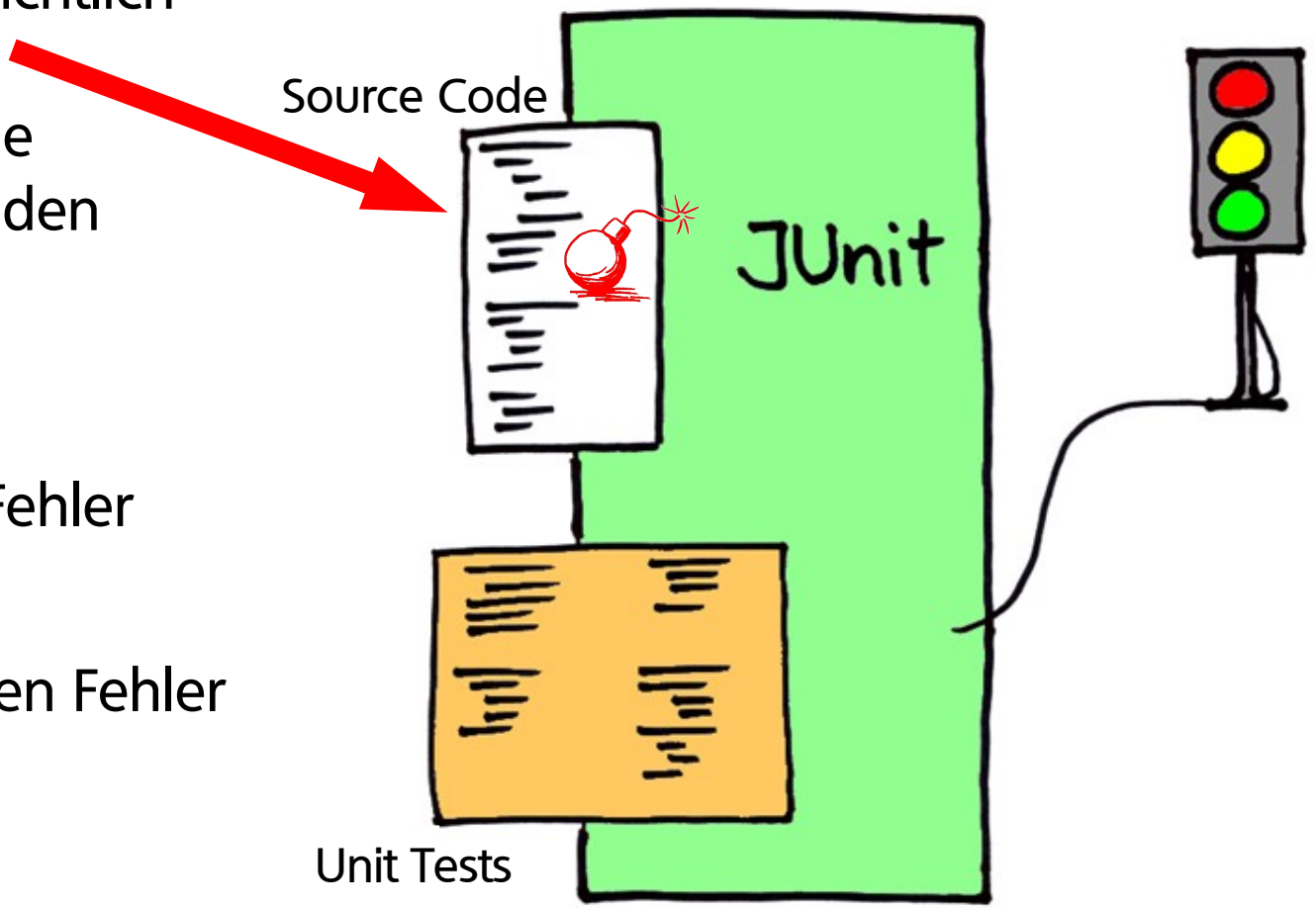
Ein **`if`** ohne **`else`** verdient besondere Beachtung: haben wir einen Testfall, der das leere **`else`** abdeckt? (das ist dann Zweig- statt Anweisungsabdeckung). Ebenso ein **`while`** das nie ausgeführt wird.

Konversionen String ⇔ float, z.B. bei Übertragung der Daten als Text (JSON) kann zu Fehlern führen, wenn der eine Computer in den USA das Komma einer Gleitkommazahl ganz anders interpretiert als der sendende Computer (DE): **`Prolactin 5,531 µg/l`**

Zeitumstellungen, Zeitangaben, Sommerzeit: alles UTC? … wie auch weitere Internationalisierungs-Themen (Sprachen, Feiertage…)

Strategie: "zufällige" Fehler in den Code streuen

Gehen Sie durch Ihren Source Code und fügen Sie absichtlich einen Fehler ein.
Dann schauen Sie, ob die vorhandenen Unit Tests den Fehler entdecken.



Falls die Unit Tests den Fehler nicht finden:

Unit Test kreieren, der den Fehler fängt.

Nochmals testen.

Eingebauten Fehler wieder entfernen.

Idea from "Mutation Driven Testing" by Rajiv Prabhakar

Testen Sie Ihre Unit Tests!

Beispiele für bewusst eingestreute Fehler:

- Off-by-one error einfügen (z.B. `<` durch ein `<=` ersetzen)
- Leeres Suchresultat liefern, obwohl eigentlich etwas gefunden werden müsste (bei sowas wie `getCustomerByID()`)
- Ein `&&` (AND) durch ein `||` (OR) ersetzen.
- Eine negative statt wie sonst üblich eine positive Zahl zurückgeben.
- Zeitverzögerung einbauen
- Kommentieren Sie eine beliebige Zeile aus.
- . . . Lassen Sie Ihrer Fantasie freien Lauf, es macht Spass! Tun Sie das, was ein naiver, nachlässiger Programmierer tun könnte.

Konsistenz-Tester für die Datenbank

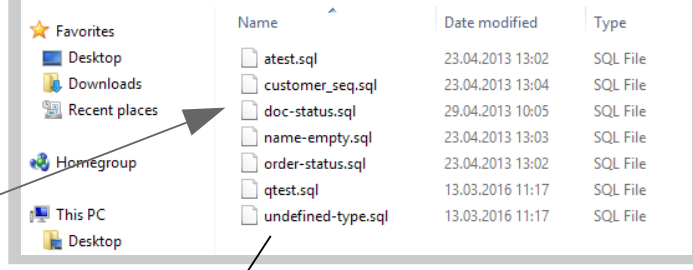
Konsistenzbedingungen einer DB prüfen, z.B.

- Gibt es Kunden, bei denen das Namensfeld leer ist?
- Gibt es Bestellpositionen ohne Bestellung?
- Gibt es unerledigte Bestellungen, die älter als 60 Tage sind?
- Gibt es Bestell-Dokumente, die einen Status 'cancelled' haben und trotzdem Bestellpositionen aufweisen, welche noch Artikel blockieren?
- Gibt es Merklisten mit ungültigen/veralteten Artikelnummern?
- Gibt es Rechnungen mit ungültigen Verweisen auf Bestellungen?

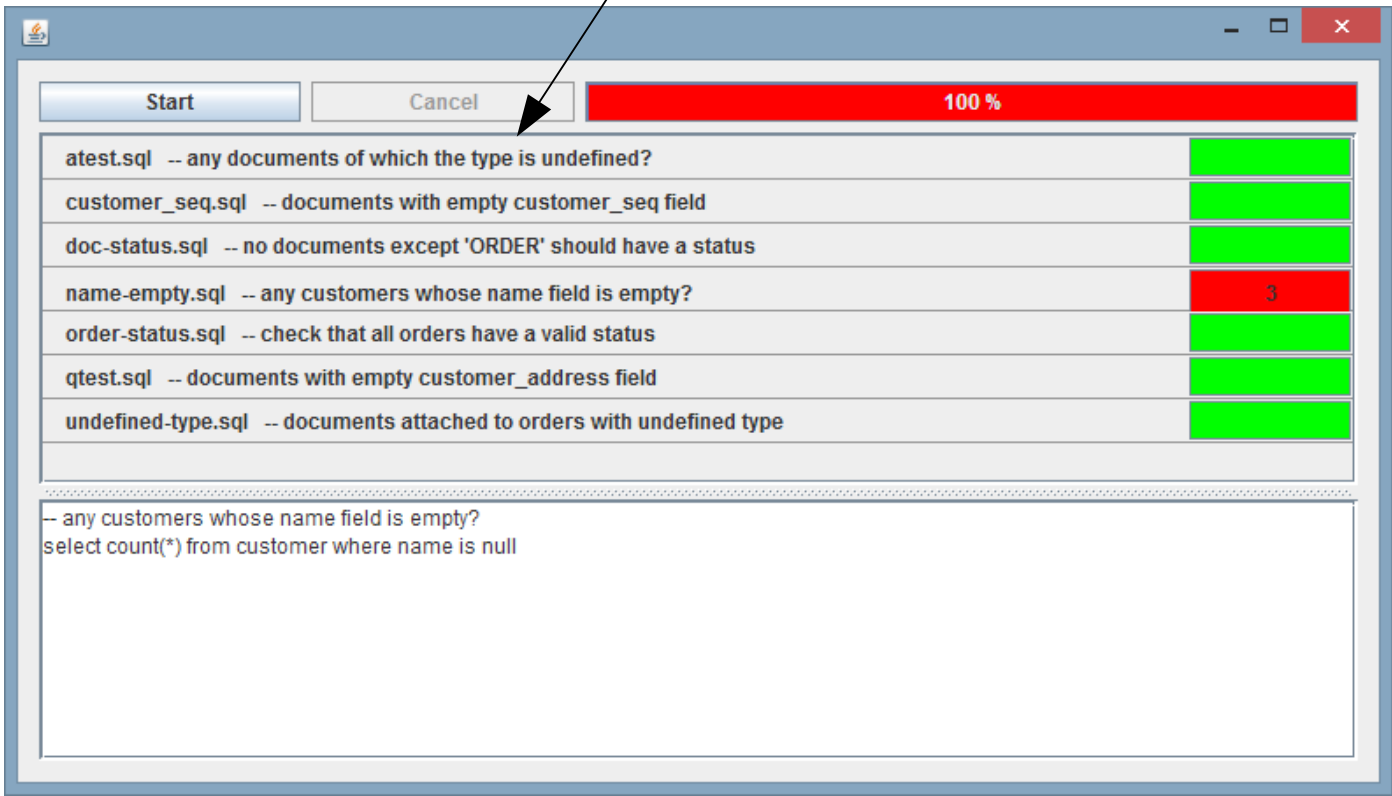
SQL Beispiele

```
-- Uebersetzungen ohne passenden Namensschluessel
SELECT * FROM TRANSLATION LEFT OUTER JOIN NAMES ON UPPER_STRING = NAME
WHERE NAME IS NULL

-- gibt es Auslagerinstructions, die eine ungueltige location.id haben?
select inst.id, inst.statecode from instruction inst, location loc where
inst.statecode in ('R', 'A', 'P') and inst.type = 'A' and
inst.fromlocationid (+) = loc.id and loc.id is null;

-- steht dieser Artikel in locations, die nicht artikelrein sind?
select c.id, c.code, a.code from carrier c, stockeditem st, article a,
location loc where st.articleid = a.id and st.carrierid = c.id and
c.locationid = loc.id and c.locationid in (select unique loc.id from
stockeditem st, article a, carrier c, location loc where st.articleid =
a.id and st.carrierid = c.id and c.locationid = loc.id and a.code =
'F00C3E2045') and a.code <> 'F00C3E2045';
```

SQL wie Unit Tests rot&grün



- SQL Files in einem Verzeichnis,
  eine Kommentarzeile zu Beginn (was testet das Skript)
  auf den nächsten Zeilen eine SQL Anweisung (auch mehrzeilig)
- Jede SQL Anweisung soll im Gut-Fall die Zahl Null zurückgeben
- Häufig left outer joins "wo fehlt etwas?"
- Alle SQL Files
  werden nacheinan-
  der ausgeführt
- Return = 0: Grün
  Return ‹› 0: Rot



Eigenbau:
14 Java Klassen
720 LOC
ca. 10 Arbeitstage

Zusammenfassung

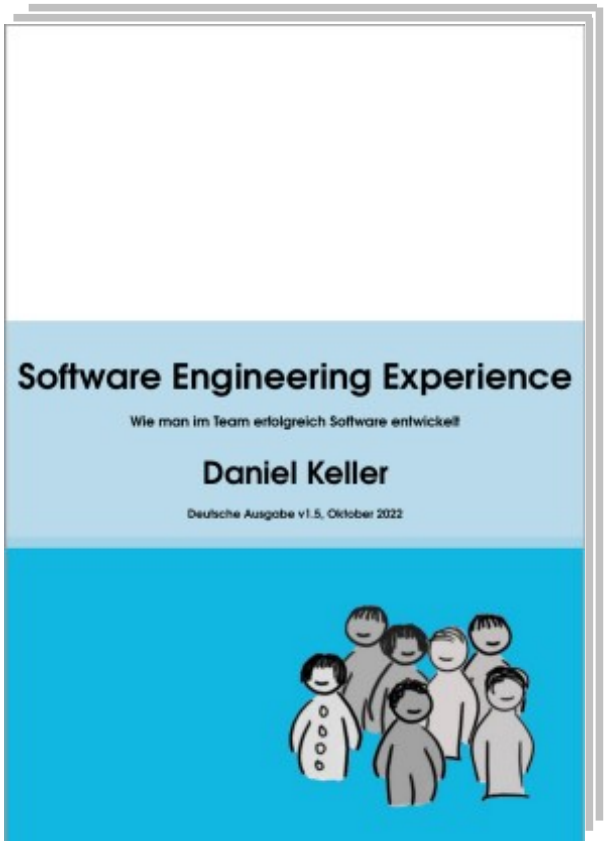
Unit Testing ist ein hervorragendes Konzept

- Unterscheidung Microtests vs. Integrationstests
- Aufruf-Hierarchien, Schichten-Architektur und Faking & Mocking
- Microtesting hat mehrere Schwächen.
- Integrationstests – wenn richtig gemacht – sind stärker.
- Wie man bessere Testfälle findet (Boundaries, Condition Coverage, bewusst eingestreute Fehler)
- 100% Testabdeckung ist gut, ist aber erst der Anfang.
- Wann ist "genug getestet"? (Kurzantwort: Reviews)
- Unit Testing auch für die Datenbank (automatisierte Konsistenztests)

Mein Software Engineering Buch

Sie können gratis mein Buch von 2021 als PDF in de/en herunterladen. Es sind 340 Seiten mit mehr als 300 Illustrationen.

Das Buch ist aus dem Unterricht in Software Engineering an der Hochschule Rapperswil www.hsr.ch (heute www.ost.ch) entstanden, wo ich von 2011 bis 2020 in Teilzeit das Thema Software Engineering unterrichtete – und natürlich aus meinen Software-Projekt-Erfahrungen in der Industrie seit 1977.



Download:
www.danielkeller.info (und dann etwas hinunterscrollen)