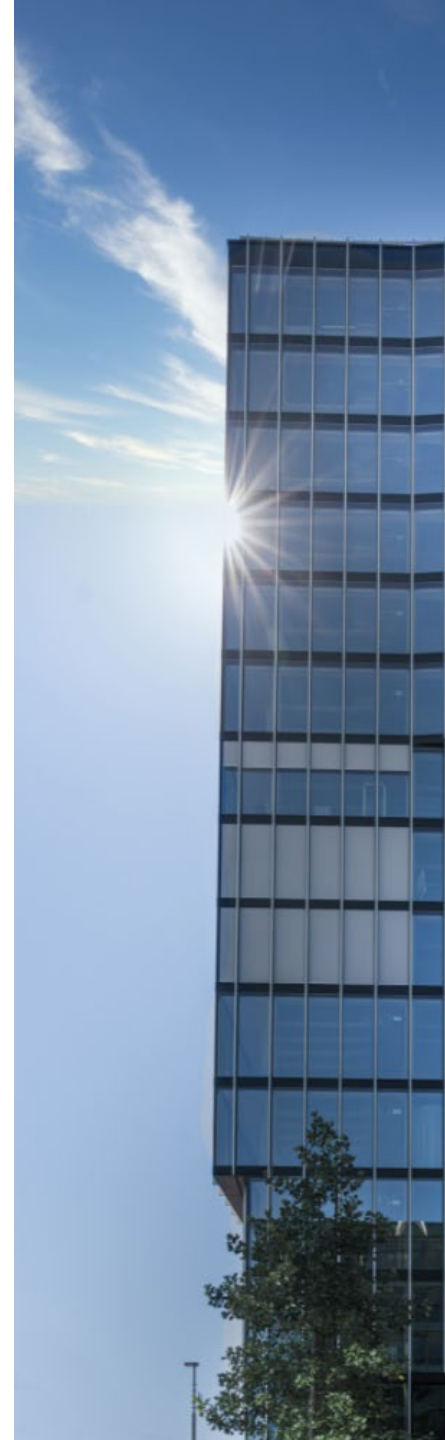


Objektorientierte Programmierung

Unit Testing

Automatisiertes Testen

Roland Gisler



Inhalt

- Testing – Einführung und Motivation
- Bedeutung des Testens
- Aussagekraft von Tests
- Verschiedene Testverfahren und -arten
- Unit-Testing
- Test First Ansatz
- Code Coverage (Grundlagen)
- Zusammenfassung und Quellen

Lernziele

- Sie kennen die Motivation, den Sinn und den Zweck des Testens.
- Sie wissen, was Sie mit Tests erreichen können und was nicht.
- Sie kennen verschiedene grundlegende Testarten und –verfahren.
- Sie können in Ihrer Entwicklungsumgebung einfache und gute Unit Tests, basierend auf dem JUnit-Framework, implementieren und anwenden.
- Sie kennen die Vorteile von Test First.

Warum testen wir?

Motivation für das Testen: Qualität

- Wie definieren wir in der Softwareentwicklung Qualität?
Qualität ist die Übereinstimmung mit den Anforderungen unter gleichzeitiger Einhaltung von Qualitätskriterien.
 - Die Anforderungen müssen überprüfbar formuliert sein,
 - typisch in Form von System- und Testspezifikation.
- Was sind Qualitätskriterien?
Funktionalität, Zweckdienlichkeit, Robustheit, Zuverlässigkeit, Sicherheit, Effizienz, Benutzbarkeit, Geschwindigkeit, ...
- Zur Prüfung, ob die Qualitätskriterien eingehalten werden, stehen uns verschiedene Massnahmen zur Verfügung:
 - Methodiken, Techniken, Vorgehensweisen etc.

Software, die (gut) getestet wird...

- steuert viele medizinische Geräte im Spital
 - Herz-/Kreislaufmaschine, Beatmungsgerät, Infusion etc.
 - Beispiele: Blutanalysegeräte von Roche
- kontrolliert und steuert verschiedenste Verkehrsmittel
 - Luftraumüberwachung, Autopiloten, Zugbeeinflussung etc.
 - Beispiele: Trägerrakete, Kampfflugzeug
- steuert aufwändige und komplizierte Prozesse
 - Beispiele: Chemische Industrie, Atomkraftwerke, Sicherheit etc.
- Möchte jemand in diesen Bereichen auf das Testen verzichten und auf die «geniale» Programmier*in vertrauen?



Notwendigkeit des Testens

Ariane 5: Flight 501

Am 4. Juni 1996 startete die ESA eine unbemannte Rakete mit vier Satelliten an Bord von Französisch-Guyana aus. 40 Sekunden nach dem Start explodierte die Rakete. Das verursachte einen Verlust von ca. 500 Millionen Dollar für Rakete und den Satelliten. Die Entwicklungskosten betrugen ca. 7 Milliarden Dollar.



Ursache für den Absturz: Der Bordcomputer stürzte 36.7 Sek. nach dem Start ab als er versuchte, den Wert der horizontalen Geschwindigkeit von 64 Bit Gleitkomma-darstellung in eine vorzeichenbehaftete 16 Bit Integer umzuwandeln. Die Zahl war aber grösser als $2^{15}=32'768$ und erzeugte einen Overflow. Das Lenksystem brach zusammen und gab die Kontrolle an eine redundante Einheit weiter, welche sofort denselben Fehler machte, weil es sich um die identische Software handelte.

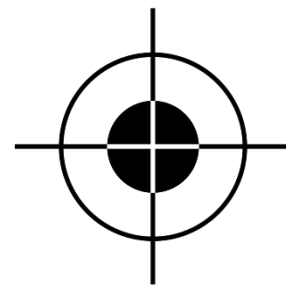
Qualitätssicherung durch Testen

- Eine der wesentlichsten Tätigkeiten ist das **Testen!**
- Dabei überprüfen wir das Verhalten eines Programms anhand der Spezifikation (Anforderungen).
- **Wichtig:** Wir sollten so viel wie möglich **automatisiert** testen, weil manuelles Testen sehr aufwändig und zeitintensiv ist!
- Daneben gibt es eine Reihe von Begleitmassnahmen, welche qualitätssichernd und –unterstützend wirken, beispielsweise:
 - Reviews, Entwicklungsprozess, Walkthrough, Metriken, Analysen, Regression, Automatisierung etc.

Was ist sinnvolles Testen?

Was bedeutet eigentlich «Testen»?

- Testen ist das **systematische, gezielte** und möglichst **effiziente** «Durchprobieren» nach verschiedenen Qualitätskriterien, wie z.B. der Funktionalität.
- Systematisch und gezielt heisst:
Man testet mit **wohlüberlegten** Eingabedaten / Testwerten.
- Effizient heisst:
Man versucht möglichst **aussagekräftige** Tests durchzuführen, was man wiederum hauptsächlich durch die Auswahl von **wohlüberlegten** Testdaten erreicht.
- ➔ Testen ist **kein** zielloses, chaotisches «Pröbeln»!
 - Testen ist wesentlich anspruchsvoller (und damit auch interessanter und spannender) als man gemeinhin denkt!



Aussagekraft von (erfolgreichen) Tests

- Man kann zwar immer nur das Vorhandensein von Fehlern zeigen, aber **nie die Abwesenheit von Fehlern beweisen**.
- Nach einem Test weiss man **nur**, dass ein Programm für die **getesteten Eingabedaten** korrekt läuft.
- Darum wählt man die Eingabedaten für Tests so, dass man aus dem Resultat auf die korrekte Verarbeitung von möglichst vielen Varianten von möglichen Eingabedaten rückschliessen kann.
 - Das schliesst somit typische als auch untypische Werte ein (Normal-, Sonder- und Ausnahmefälle).
- Beispiele für einen Sortieralgorithmus:
 - beliebige, zufällige Daten (Normalfälle)
 - vorsortierte Daten, keine Daten, etc. (Sonderfälle)
 - ungültige, nicht sortierbare Daten (Ausnahmefälle)

Beispiel – Umrechnung von Celsius nach Kelvin

- Mathematische Formel: $T_K = T_C + 273.15$
- Planung der Testdaten (Teil der Testspezifikation):

	Testfall	Eingabewert [°Celsius]	Erwartetes Resultat [Kelvin]
Normalfälle	1	20.0f	293.15f
	2	0.0f	273.15f
	3	-10.0f	263.15f
Sonderfälle (zulässig)	4	-273.15f	0.0f
	5	Float.MAX_VALUE – 273.15f	Float.MAX_VALUE
Ausnahme- fälle (unzulässig)	6	-273.16f	unzulässige Temperatur
	7	Float.MAX_VALUE	Bereichsüberlauf

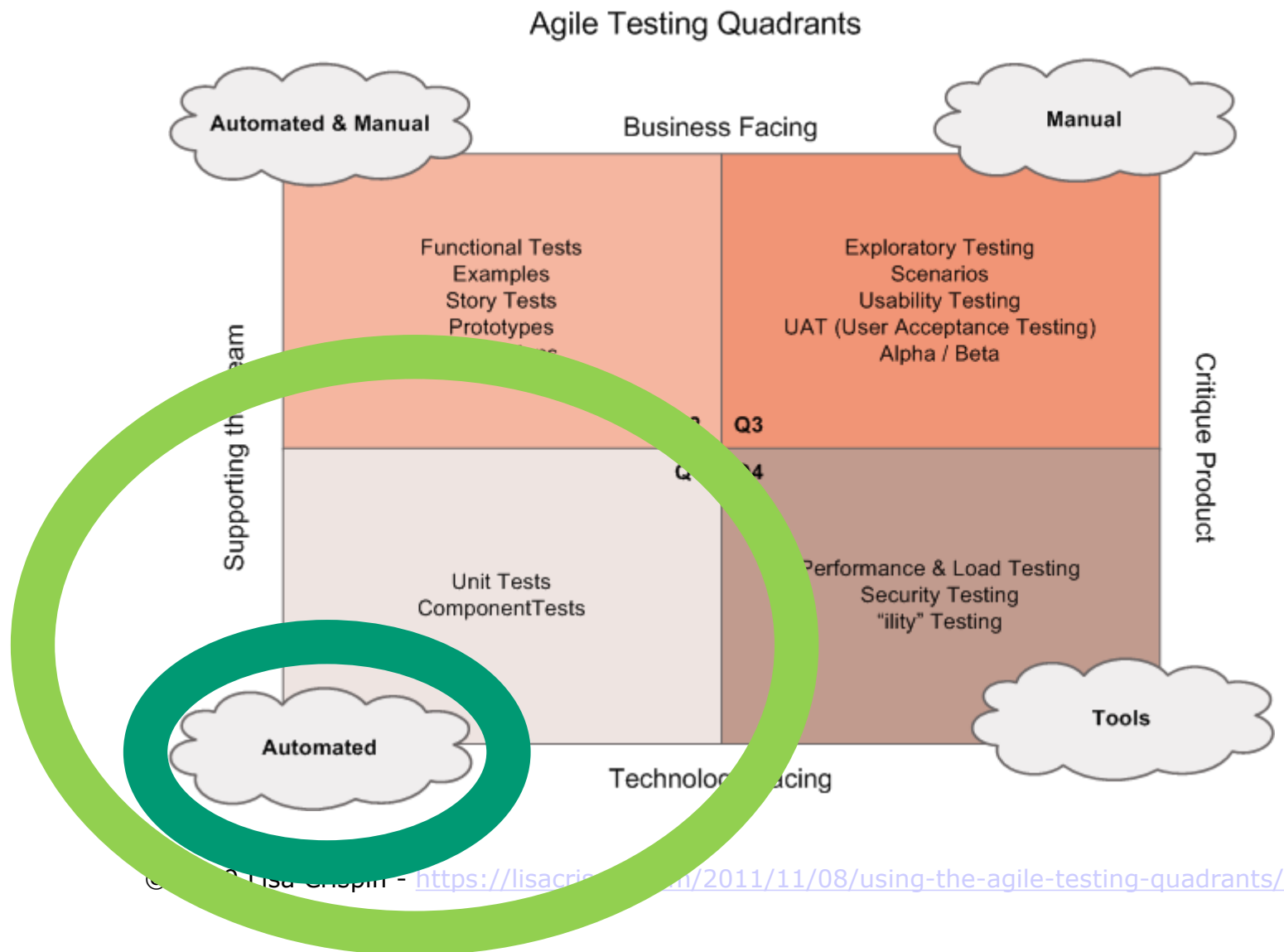
- Annahme: Die Funktion verwendet den Datentyp **float**.

Testverfahren

Verschiedene Testarten bzw. -verfahren

- **Unit Tests:** Man testet "nur" eine Methode, Klasse bzw. Einheit.
 - Sind sehr klein, übersichtlich, einfach, schnell ausführbar und einfach automatisierbar (Ausführung und Validation).
- **Integrationstests:** Man testet typisch mehrere Klassen (auch Module/Komponenten oder Teilsysteme) in ihrem Zusammenspiel.
 - Schon deutlich aufwändiger, aber auch automatisierbar.
 - Werden in späteren Modulen (VSK und SWDA) vertieft.
- **Systemtest:** Man testet das "ganze" System bestehend aus vielen Klassen bzw. Einheiten.
 - «Klassisches» Testen, erst spät möglich, aufwändig, aber auch automatisierbar! (Wird ebenfalls in VSK und SWDA vertieft.)
- Orthogonal dazu: **Black-** oder **White**box Testing:
Man testet **ohne** oder **mit** Kenntnis der Implementation.

Testquadranten nach Crispin




Unit Testing

Unit Tests

Unit Tests sind eine fundamentale Grundlage für zeitgemässe und iterative Entwicklung von Software!

- Die Testfälle werden **programmiert** und sind somit **automatisiert**, jederzeit schnell ausführbar und wiederholbar.
- Die Testfälle sind «self-validating», d.h. die **Verifikation** ob der Testfall erfolgreich durchgeführt wurde, erfolgt **automatisch**.
- Die getesteten Einheiten sind einzelne Methoden bzw. Klassen
➔ kleine und sehr überschaubare Testfälle!
- Für Unit Testing stehen zahlreiche **Frameworks** für verschiedenste Sprachen zur Verfügung. Diese sind meist sehr gut in die Entwicklungsumgebung integriert.
 - für Java: JUnit (am populärsten), UnitNG etc.

Unit Tests – Grosser Nutzen

- Automatisierte Unit Tests ersparen uns viel Zeit!
 - Einmal geschrieben, **n**-fach ausgeführt.
- Kein mühsames, «manuelles» Testen mehr.
- Keine «**System.out.println**»-Orgien mehr. 
- Jederzeit reproduzierbar, auch nach langer Zeit!
- Automatische Verifikation der Ergebnisse mit Reporting!
- Testfälle «dokumentieren» quasi die erwartete Funktion / das erwartete korrekte Resultat!
- Testfälle geben uns die Freiheit mit geringerem Risiko den Code zu verändern bzw. umzuschreiben (➔ Refactoring).

JUnit Test Framework

JUnit Test Framework

- Kent Beck und Erich Gamma haben mit JUnit das wohl bekannteste und erfolgreichste Test Framework entwickelt.
 - Ursprünglich für Unit-Tests konzipiert, wird es heute für alle automatisierbaren Testfälle eingesetzt.
 - ➔ Darum: **Nicht jeder JUnit-Testfall ist ein Unit-Test!**
- Mittlerweile liegt es in der Generation **5** (JUnit 5 Jupiter) vor.
- JUnit 5 enthält sehr viele Verbesserungen, ist aber nicht mehr Rückwärtskompatibel (Major-Release).
 - Bestehende, auf JUnit 4 basierende Testfälle müssen migriert werden. Einfach, aber dennoch mit Aufwand verbunden.
- Wir konzentrieren uns von Anfang an auf die neuste Version **5.x!**

Beispiel eines Unit 5 Tests in Java (mit Junit)

- Testfall entspricht dem «Build-Operate-Check»-Pattern:
 - ❶ Erstellen der Testobjekte bzw. –daten.
 - ❷ Manipulieren der Testobjekte bzw. –daten.
 - ❸ Verifikation der Ergebnisse (mit `assert*`-Methoden).

@Test

```
void testGetQuadrantInOne() {  
    final Point point = new Point(4,5);  
    final int quadrant = point.getQuadrant();  
    assertEquals(1, quadrant);  
}
```

- Auch bekannt als «**Triple A**»-Pattern: **A**rrange, **A**ct, **A**ssert.
- Haben Sie den Eindruck, das sei sehr einfach? **Ja!** So soll es sein!
 - Beispiel lässt sich auf eine Zeile kürzen: Noch einfacher!

JUnit 5: Self-validating mit assert*()-Methoden

- Einfacher Vergleich zwischen Soll- und Ist-Wert.
 - Methoden der Klasse `org.junit.jupiter.api.Assertions`
 - Für viele Datentypen überladen (elementar oder Klasse `Object`)
- Einige Beispiele (von sehr vielen Varianten):
 - `assertEquals(long expected, long actual)`
Prüft, ob die zwei long-Werte gleich sind.
 - `assertEquals(float expected, float actual, float delta)`
Prüft, ob zwei float-Werte gleich sind (mit Toleranz `delta`).
 - `assertEquals(Object expected, Object actual)`
Prüft, ob die Objekte gleich sind.
 - `assertTrue(boolean condition)`
Prüft, ob eine beliebige bool'sche Bedingung `true` ist.
 - ...

JUnit 5: Annotations für Test-Methoden

- Zur Konfiguration der einzelnen Tests stellt JUnit verschiedene **Annotations** zur Verfügung. Hier nur eine kleine Auswahl:

Annotation	Beschreibung
@Test void testMethod()	Markiert eine Methode als JUnit-Testfall (Pflicht). Darum ist die Namenskonvention test* eigentlich nicht mehr notwendig (aber trotzdem empfohlen!) Achtung: org.junit.jupiter.api.Test
@Test @Disabled(...) void testMethod()	Markiert einen Testfalls als (temporär) deaktiviert. Viel besser als auskommentieren, weil er so explizit als "skipped" ausgewiesen wird. Optional: String-Parameter mit Begründung.
@BeforeAll @BeforeEach @AfterEach @AfterAll ... void fooBar()	Markieren eine (bei *All statische) Hilfsmethode zur automatischen Ausführung vor/nach jedem/allen Testfällen in der Klasse. Optional, bei Unit Tests möglichst vermeiden .

- Mehr finden Sie unter: <http://junit.org/>

JUnit: Namenskonventionen für Klassen und Methoden



- Es gibt eine Namenskonvention für die Testklassen und die Testmethoden. Diese ist zwar technisch keine Bedingung, wird aber dennoch (mind. für Unit Tests) empfohlen, da sie sehr nützlich ist.
- **Testklasse:** Die Tests für eine Klasse **Demo** werden in einer eigenen Klasse **DemoTest** zusammengefasst.
 - **Test** als Appendix für die einfache Zuordnung.
 - Alle Testklassen werden in `/src/test/java` abgelegt!
- **Testmethoden:** Der Basisname für die Testfälle einer Methode `foo(...)` lautet **testFoo**[*XYZ*]().
 - **test**... als Prefix, *XYZ* als optionale, ergänzende Fallbeschreibung.
 - Testmethoden haben keine formalen Parameter.

Unit Tests in der IDE



- Aktuelle IDEs bieten reichhaltige Unterstützung bei Unit-Tests an!
 - Erzeugen von Codegerüsten.
 - Ausführung von Testfällen (alle, einzeln, selektiv).
 - Auswertung (Report / Statistik).
 - Debugging.
 - Code Coverage (optional)
 - ➔ Nutzen Sie alle diese Hilfen!
- Vorsicht bei den generierten Codegerüsten!
 - Sind nicht immer gut, aber immer **leer**!
 - Gute Unit Tests schreibt man mit etwas Übung oft am schnellsten direkt.
- JUnit **5.x** wird auch als **JUnit Jupiter** bezeichnet (z.B. Eclipse).

Demo

Empfehlungen - Für gute Unit Tests



- Besser viele kleine Testmethoden als wenige (eine) grosse!
 - weil im Fehlerfall dadurch viel bessere Selektivität!
- Möglichst wenige **assert*(...)**-Statements pro Testmethode.
 - ebenfalls höhere Selektivität im Fehlerfall und übersichtlicher.
- Methoden mit Returnwert lassen sich am einfachsten testen.
 - bei **void**-Methoden testet man ggf. indirekt über Statusabfragen auf dem getesteten Objekt.
- Getter&Setter-Methoden werden häufig gemeinsam oder indirekt und «beiläufig» mitgetestet.
- **Nie** nur für Testbarkeit Schnittstellen oder Sichtbarkeit ändern!
- Ist eine Klasse oder Methode schwierig zu testen, sollte man deren Design hinterfragen!

F.I.R.S.T. – Prinzipien



- **Fast** – Tests sollen schnell sein, damit man sie jederzeit und regelmässig ausführt.
- **Independent** – Tests sollen voneinander unabhängig sein, damit sie in beliebiger Reihenfolge und einzeln ausgeführt werden können.
- **Repeatable** – Tests sollten in/auf jeder Umgebung lauffähig sein, egal wo und wann.
- **Self-Validating** – Tests sollen mit einem einfachen boolschen Resultat zeigen ob sie ok sind oder nicht*.
- **Timely** – Tests sollten rechtzeitig, d.h. vor dem produktiven Code geschrieben werden → Test First, test-driven-development.

JUnit 4 - Hinweise zur letzten Generation

- **Assert*-Methoden** waren auf der Klasse **org.junit.Assert**.
- Bis JUnit 4 mussten sämtliche Testklassen und Methoden mit Sichtbarkeit **public** implementiert werden.
- Annotationen (z.B. **@Test**) alle aus Package **org.junit**
 - Es gibt unter JUnit 4 mehr Annotation (welche in JUnit 5 durch eine einheitliche **@extendWith**-API und durch Verwendung von **→Lambda-Expressions** ersetzt wurden).
- Weniger Dependencies (nur eine JAR-Datei), weil nicht sauber modularisiert und auch keine explizite, getrennte API.
- In manchen Projekten ist JUnit 4 noch immer der «Standard».
 - Mehr zur Version 4 finden Sie unter: <http://junit.org/junit4/>

Was war nochmal die Motivation für Tests?

Warum testen wir?

- Wir testen, um Fehler zu finden.

NEIN

- Besser:

Wir testen **kontinuierlich** während der Implementation, um von Anfang an die Gewissheit zu haben, dass es funktioniert!



- Fehler finden, bevor man sie gemacht hat!
 - Fehler korrigieren, bevor man sie implementiert hat!
 - Oder mindestens Fehler schon im Ansatz (wenn es noch niemand anders gemerkt hat) finden!
- Wie macht man das?

Test First

Test First Methodik - Grundlagen

- Entwickelt aus XP (extrem programming, u.a. von E. Gamma)
- Ganz einfacher Ansatz:
Immer VOR der Implementation die Testfälle schreiben!
- Vorteile:
 - Während dem Schreiben der Testfälle denkt man unmittelbar auch an die Implementation des zu testenden Codes. Dabei «reift» diese buchstäblich heran!
 - Typisch fallen einem dabei viele Ausnahmen und Sonderfälle ein, welche man bei der Implementation der eigentlichen Komponenten dann «automatisch» auch berücksichtigt.
 - Kaum ist die Komponente fertig, kann sie sofort getestet werden!

Test First Methode – Just do it!

- Sehr einfacher, aber genialer Ansatz!



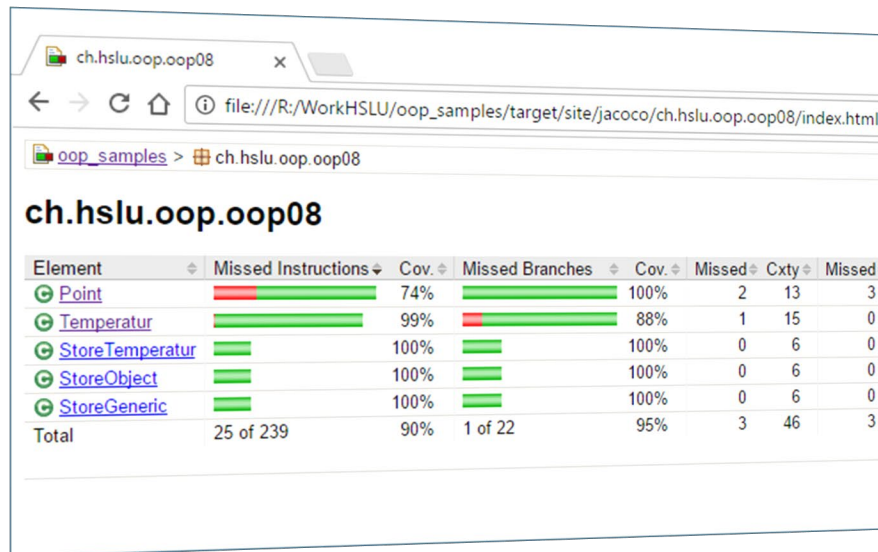
- Basiert technisch vollständig auf Unit Tests,
 - ist problemlos mit **allen existierenden** Werkzeugen für Unit Testing umsetzbar → somit alle Vorteile von Unit Tests!
- Einziger «Haken»:

Man muss es wollen und machen!

Qualitätssicherung von Testfällen

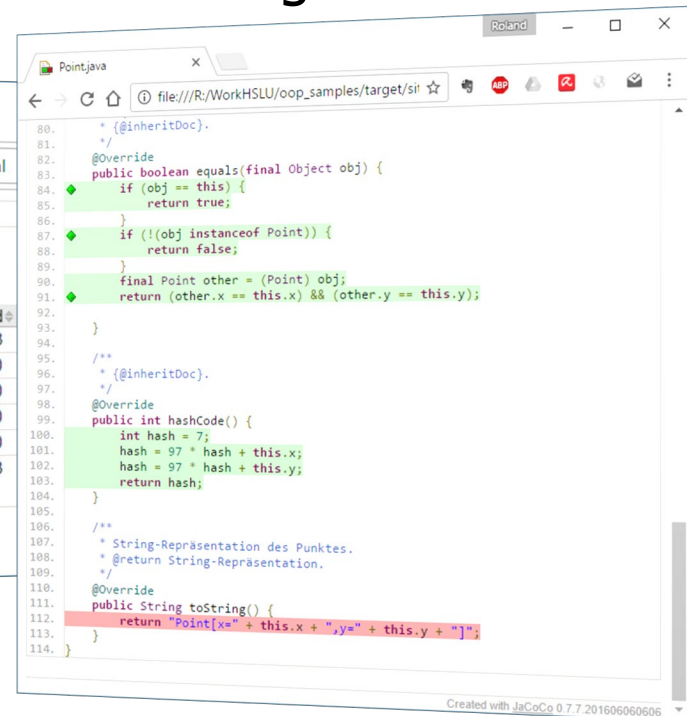
Messung der Codeabdeckung

- Herausforderungen:
 - Wie kann man die Qualität von Unit Tests beurteilen und verbessern?
 - Wie implementiert man mit möglichst geringem Aufwand trotzdem möglichst umfassende Testfälle (Effizienz!)?
- Ein **Hilfsmittel**: Messen der Codeabdeckung von Testfällen!



The screenshot shows the JaCoCo web interface for the project 'ch.hslu.oop.oop08'. It displays a table with coverage statistics for various elements. The table includes columns for Missed Instructions, Coverage (Cov.), Missed Branches, Coverage (Cov.), Missed, Cxty, and Missed. The elements listed are Point, Temperatur, StoreTemperatur, StoreObject, and StoreGeneric, along with a Total row.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed
Point	74%	100%	2	13	3		
Temperatur	99%	88%	1	15	0		
StoreTemperatur	100%	100%	0	6	0		
StoreObject	100%	100%	0	6	0		
StoreGeneric	100%	100%	0	6	0		
Total	25 of 239	90%	1 of 22	95%	3	46	3



The screenshot shows the source code of the 'Point.java' file. The code is displayed with line numbers from 80 to 114. Green highlights are visible on lines 84, 85, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, and 114, indicating that these lines were covered by the tests.

```
80.  * {@inheritDoc}.
81.  */
82.  @Override
83.  public boolean equals(final Object obj) {
84.      if (obj == this) {
85.          return true;
86.      }
87.      if (!(obj instanceof Point)) {
88.          return false;
89.      }
90.      final Point other = (Point) obj;
91.      return (other.x == this.x) && (other.y == this.y);
92.  }
93.
94.  /**
95.   * {@inheritDoc}.
96.   */
97.  @Override
98.  public int hashCode() {
99.      int hash = 7;
100.      hash = 97 * hash + this.x;
101.      hash = 97 * hash + this.y;
102.      return hash;
103.  }
104.
105.  /**
106.   * String-Repräsentation des Punktes.
107.   * @return String-Repräsentation.
108.   */
109.  @Override
110.  public String toString() {
111.      return "Point[" + this.x + "," + this.y + "]";
112.  }
113.
114. }
```

Code Coverage – Was ist das?

- Code Coverage ist eine Metrik, welche zur **Laufzeit** misst (zählt), welche Quellcodezeilen abgearbeitet wurden.
- Diese Messung wird typisch bei der Ausführung der Unit Testfälle (z.B. mit JUnit) durchgeführt.
 - Könnte aber auch zur «normalen» Laufzeit erfolgen, z.B. zur Messung welche Funktionen tatsächlich genutzt werden!
- Somit kann eine **Aussage** gemacht werden, **wie umfassend** der Code tatsächlich getestet wurde!
 - Mittel zur gezielten Effizienzsteigerung der Testfälle.
- Aber **Vorsicht**: Hohe Coverage ist **kein** Beweis für gute Testfälle oder gar Fehlerfreiheit!
- Wird im Modul **VSK** (Informatik) vertieft!

Unit Testing - Bilanz

Unit Tests: Bilanz

■ 👍 **Positiv:**

- Testen ist vollständig in die Implementationsphase integriert
➔ Aufgabe der Entwickler*in.
- Neue oder veränderte Methoden können unmittelbar, reproduzierbar und sehr schnell getestet werden.
- Test First Ansatz ist problemlos möglich und motivierend.
- Automatisiertes, übersichtliches Feedback / Reporting.
- Messung von Codeabdeckung kann integriert werden.

■ 👎 **Negativ:**

- Qualität der Testfälle muss im Auge behalten werden:
Es gilt **Qualität vor Quantität!**
- Für GUI(-Komponenten) aufwändiger.

Unit Tests: Bilanz – Nutzen ist unbestritten

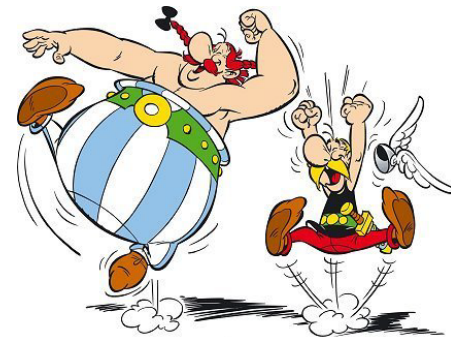
- Unterstützung durch Frameworks in nahezu jeder Sprache und gute Integration in viele Entwicklungsumgebungen gegeben.
- Etabliert, Ihr Nutzen ist unbestritten!
- Die Qualität ist aber stark abhängig von den Mitarbeitenden.

Trotzdem wird meistens bei den Unit-Tests als erstes «gespart», was sich im späteren Projektverlauf aber immer rächt!



Zusammenfassung

- Nicht nachträgliches Testen zur Fehlersuche, sondern kontinuierliches Unit-Testen zur Bestätigung, dass es funktioniert!
 - Test First mit Unit Tests: einfaches und attraktives Vorgehen.
- Grosse Unterstützung durch zahlreiche Werkzeuge, Frameworks und Plugins gegeben, JUnit ist bei Java Quasistandard:
 - Vollständige Integration der Unit Tests in die Implementationsphase des Projektes.
 - Automatisierung der Testausführung und Validation.
- Messung der Code Coverage als Motivationsfaktor.
- Qualität der Testfälle immer im Auge behalten!



Plötzlich macht Testen Spass!



Fragen?

Fragen bitte im
ILIAS-Forum



Quellen

- JUnit - <http://junit.org>
- TestNG - <http://testng.org>
- AssertJ - <https://assertj.github.io/doc/>
- EclEmma / JaCoCo - <http://www.eclemma.org/jacoco/>