

**Swiss Engineering Institute**

Unsere Praxisuniversität.

*Reto Blunschi*

© 2018 Swiss Engineering Institute AG - an ERNI Group company

## *Camp 4 – Agiles Testing*

WLAN:

SEI AP

Passwort:

sei!2017



# Vorstellung & Erwartungen



- Swiss Engineering Institute  
an ERNI Company
- Trainer: Reto Blunschi

Ihre

- Erfahrungen?
- Erwartungen?

# *Logistik*

## *Ablauf*

- Module 2: Agiles Testing
  - 1. Tag 10:00 Uhr – ~18:00 Uhr
  - 2. Tag 09:00 Uhr – 17:00 Uhr
- 1/3 Theorie und Übungen, 2/3 Weiterentwicklung eures Prototypen

# Kursziele

## Agiles Testing



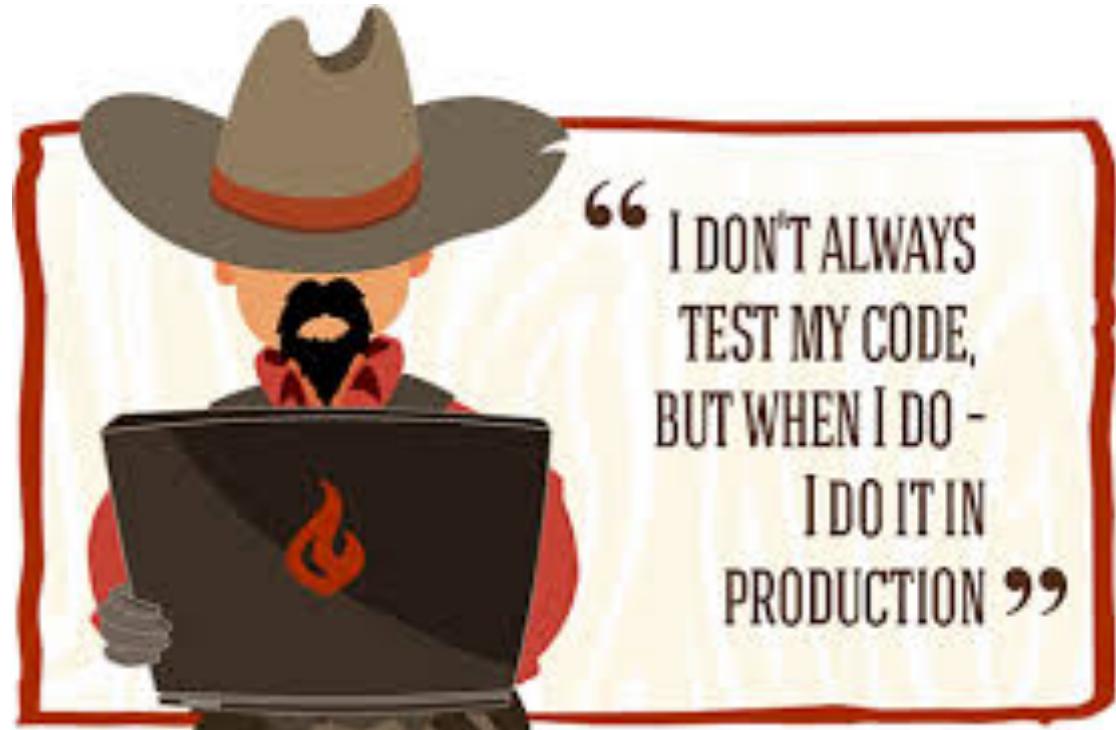
- Ich weiss warum Entwickler automatisiert testen
- Ich kenne die verschiedenen Teststufen und Testtypen
- Ich kann einen Unit Tests mit Javascript/Jasmine schreiben
- Ich habe TDD (Test Driven Development) ausprobiert
- Ich kann einen GUI Tests mit Protractor/Selenium schreiben
- Ich habe automatisches Testing in unserer App integriert.

# Inhaltsverzeichnis

## Agiles Testing

1. Motivation: Warum sollen Entwickler testen?
2. Teststufen und Testtypen
3. Unit-Tests
4. Test Driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests

Dazwischen:  
App Entwicklung



“ I DON'T ALWAYS  
TEST MY CODE,  
BUT WHEN I DO -  
I DO IT IN  
PRODUCTION ”

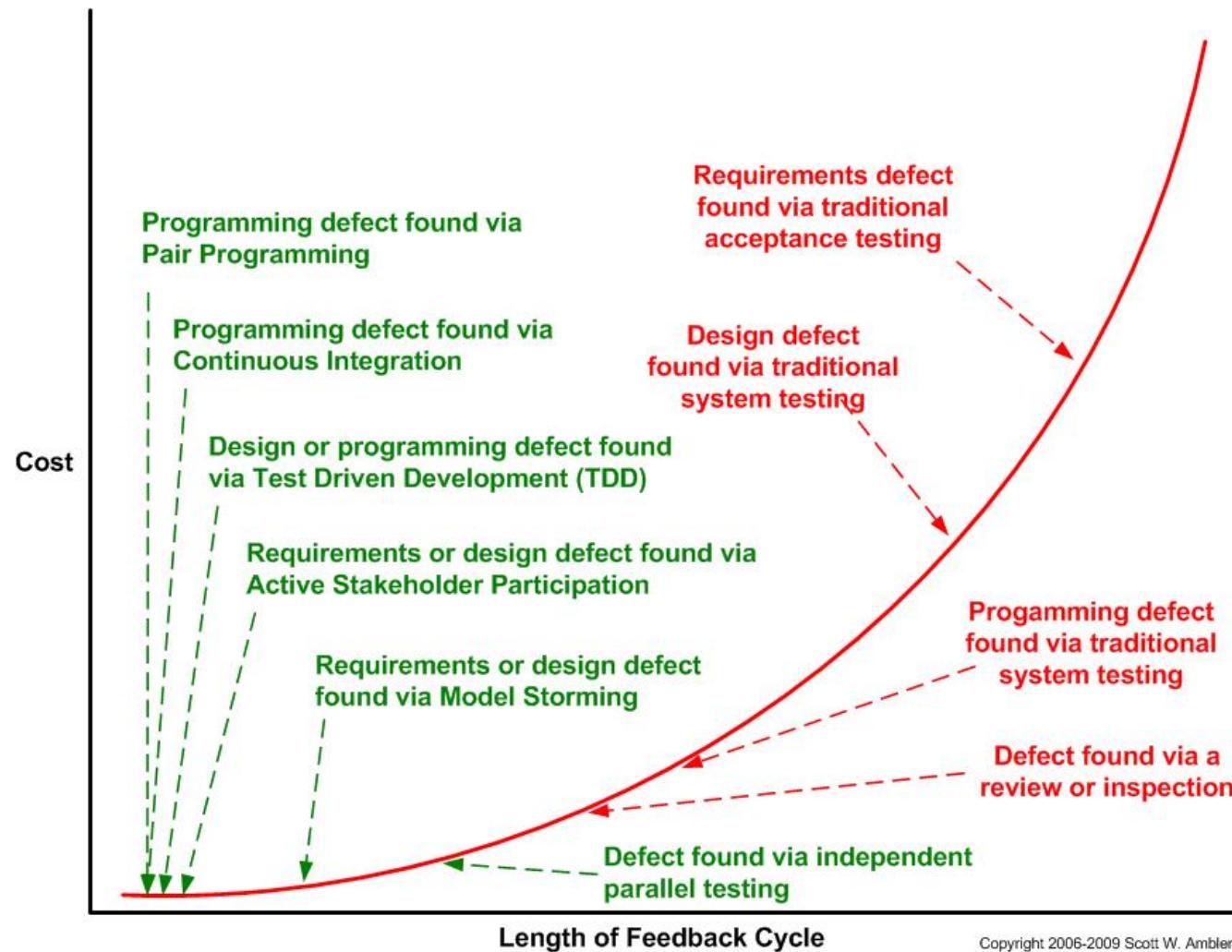
# *Warum Entwickler automatisiert testen*

## *Typische Aussagen in Software-Projekten*

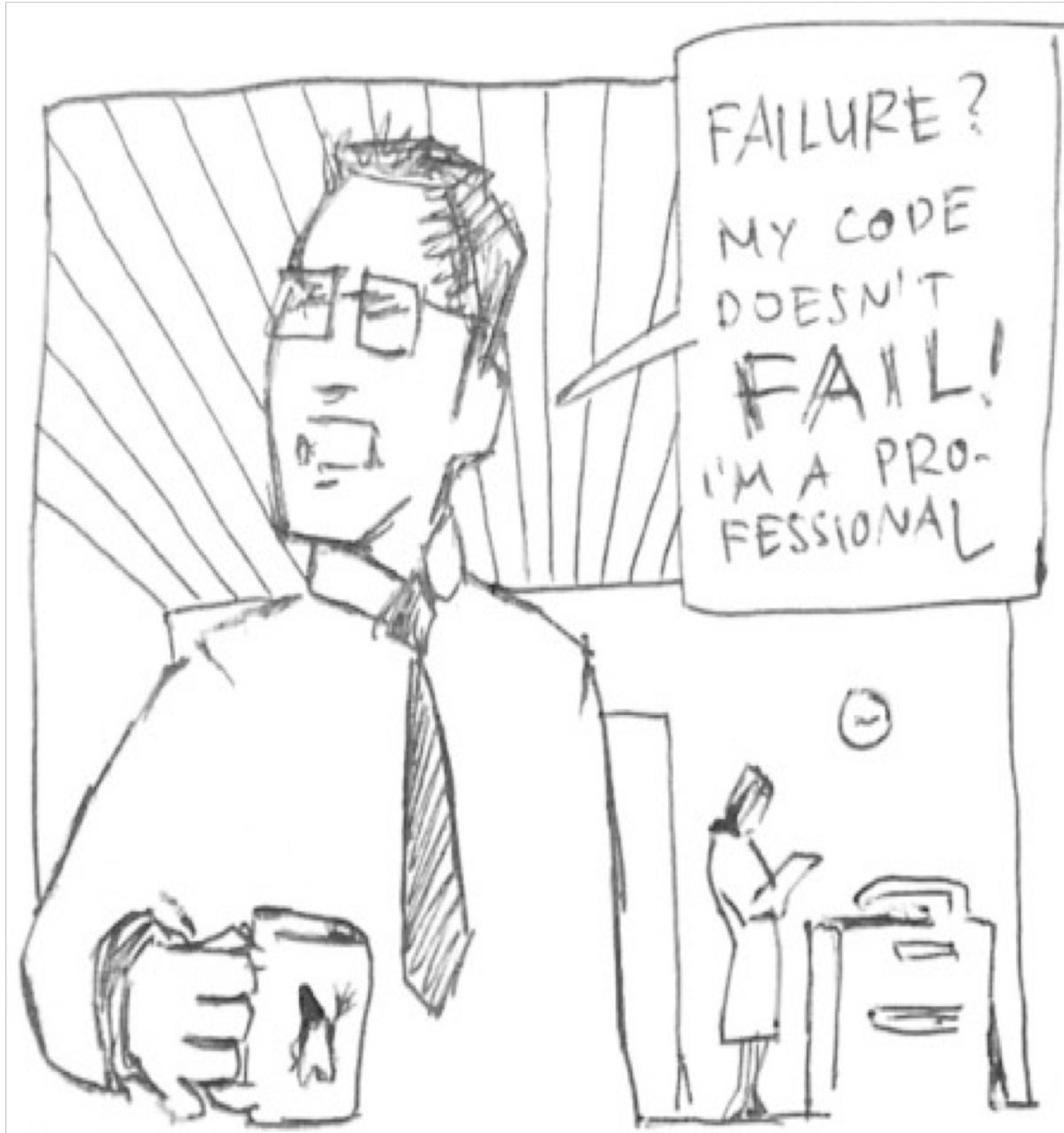
- „Je später ein Fehler gefunden wird, desto teurer ist es“  
-> oft Faktor 10 an Aufwand zwischen einzelnen Stufen
  - Agile Development -> Konstantes Refactoring  
„wie können wir wissen, ob unser System noch funktioniert nach einem Umbau“  
„Wir können das nicht ändern, das Risiko ist zu gross“
  - „Ein Bug gefixt -> drei neue eingebaut“
  - “Diesen Bug haben wir doch schon vor drei Monaten gefixt, jetzt ist er wieder da”
- in der Agilen Entwicklung ist automatisiertes, agiles Testen unerlässlich

# Kosten eines gefundenen Fehlers

## In Abhängigkeit zum Zeitpunkt



Copyright 2006-2009 Scott W. Ambler

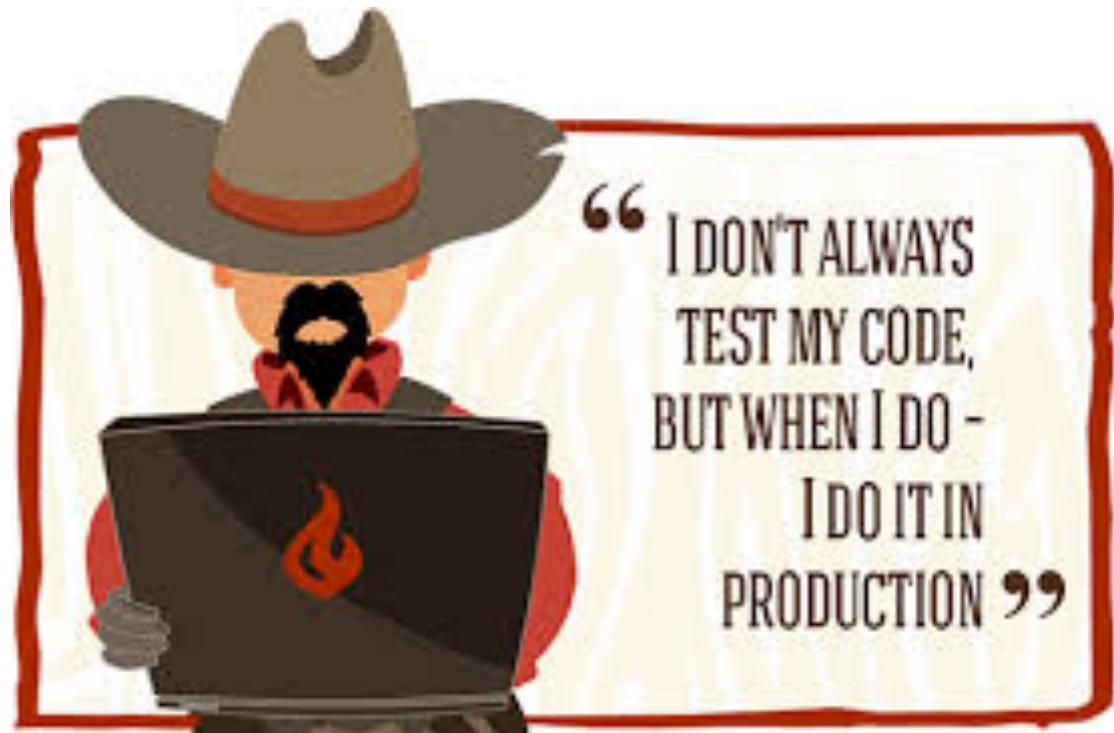


# Inhaltsverzeichnis

## Agiles Testing

1. Motivation: Warum Testing durch Entwickler
2. Teststufen und Testtypen
3. Unit-Tests
4. Test Driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests

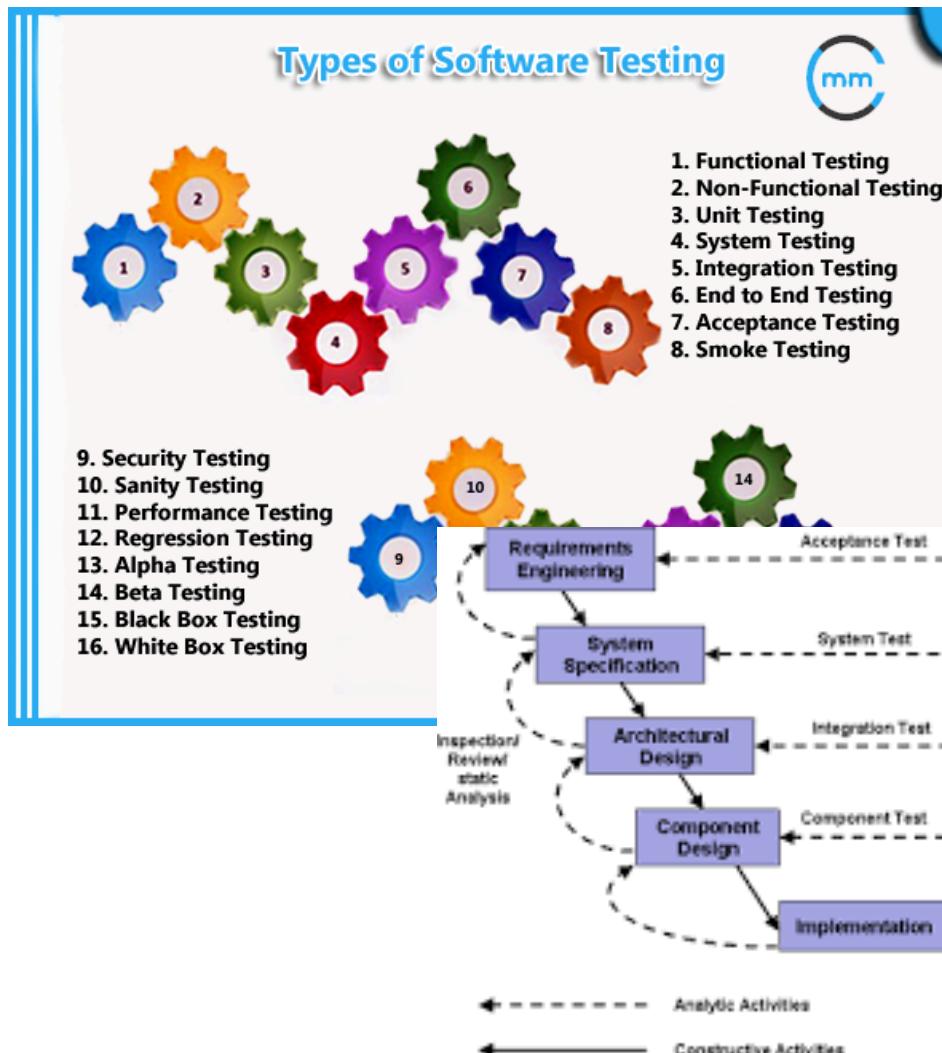
Dazwischen:  
Weiterarbeit am Prototypen



“ I DON'T ALWAYS  
TEST MY CODE,  
BUT WHEN I DO -  
I DO IT IN  
PRODUCTION ”

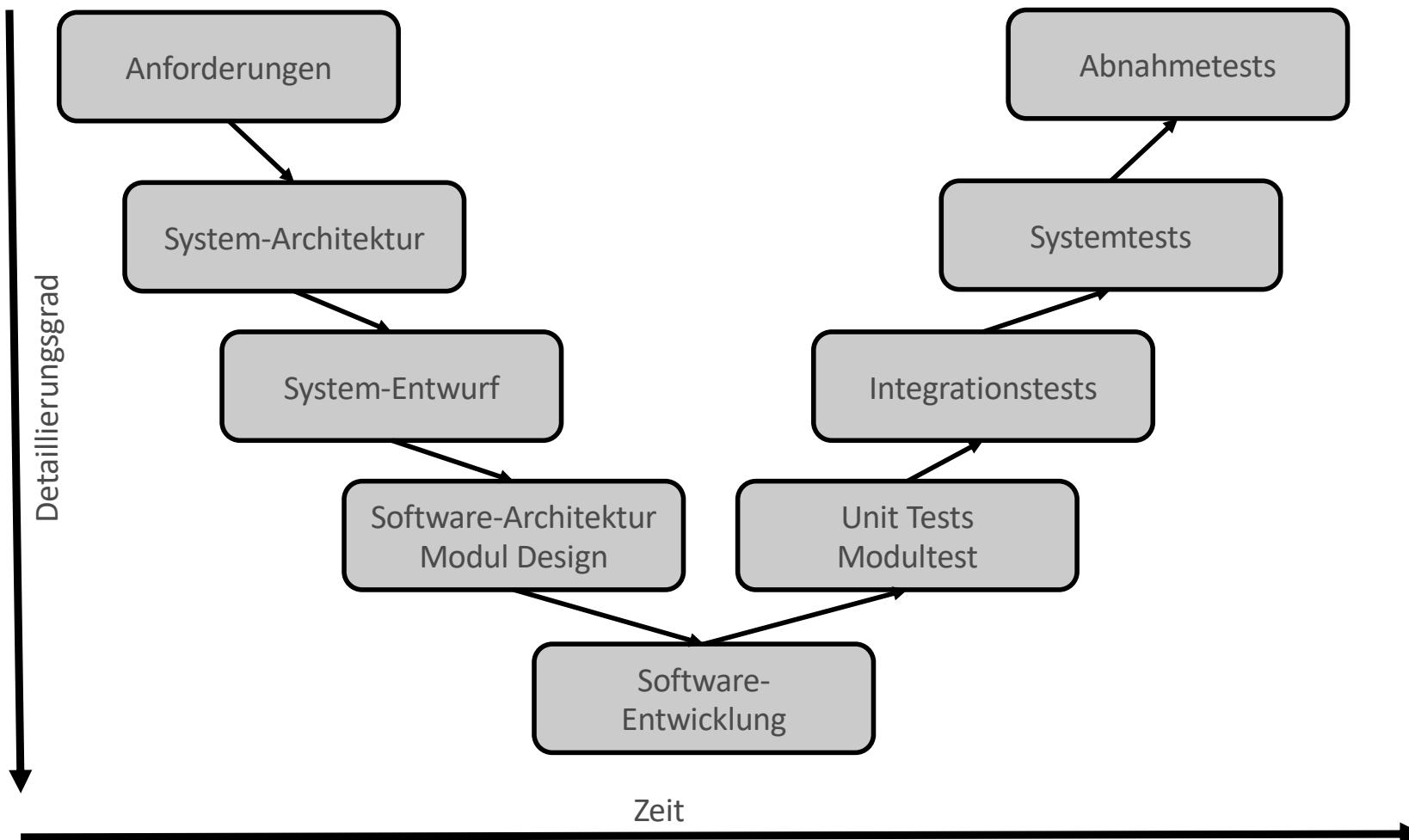
# Teststufen und Testtypen

Viele Definitionen – viele Namen für gleiche Dinge



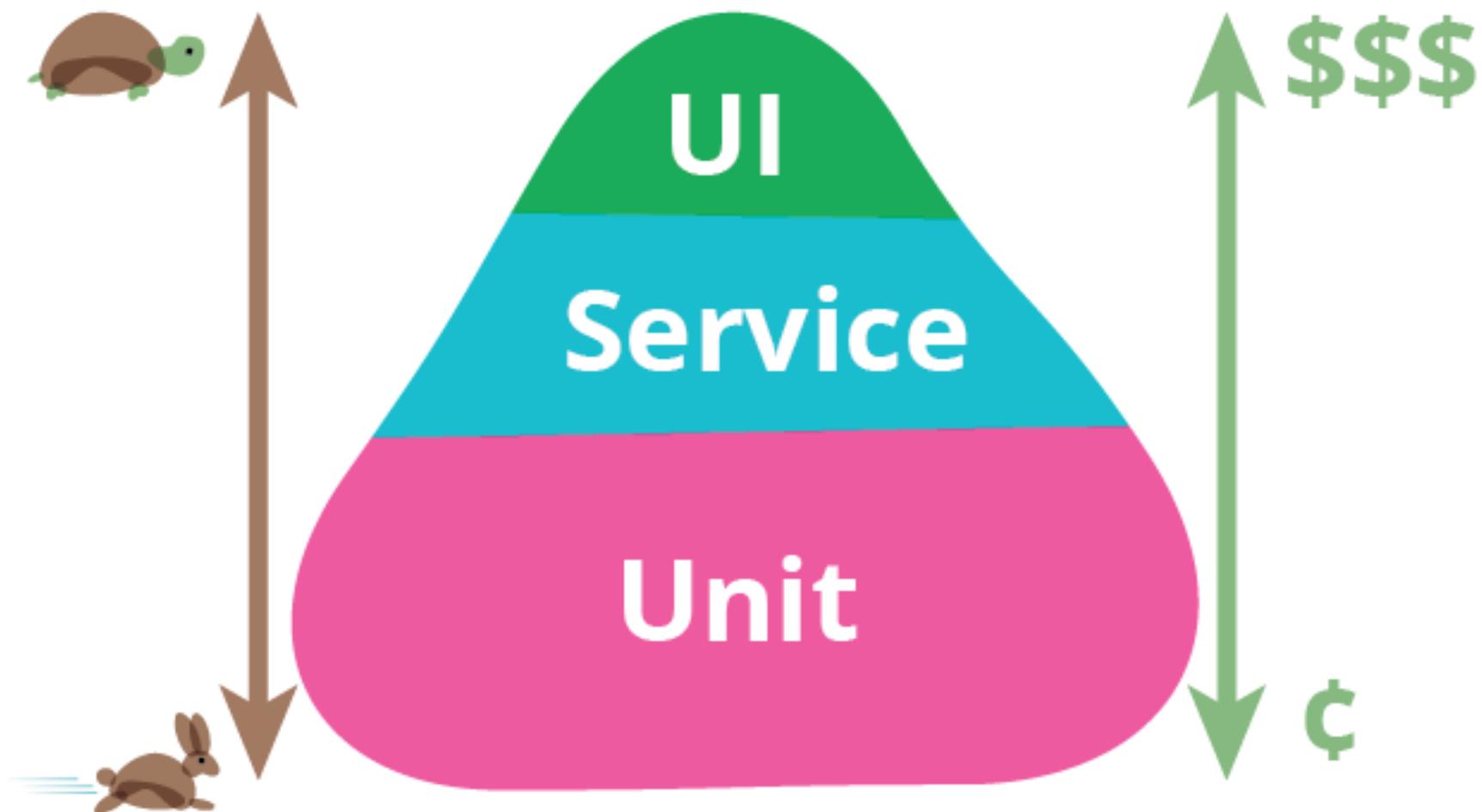
# *Teststufen – V-Modell*

## *Lifecycle eines Softwareprodukts – Traditionelle Sicht*



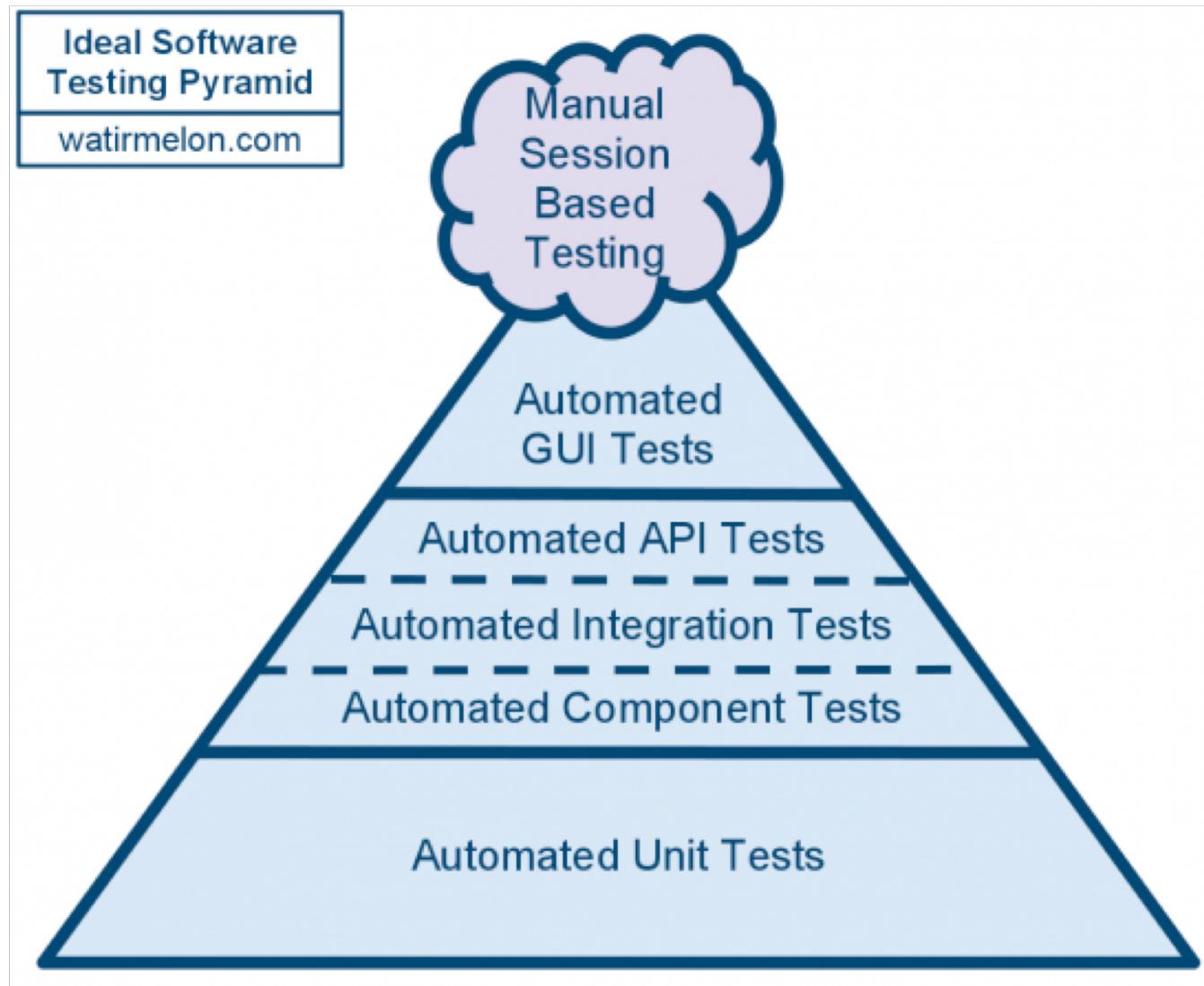
# *Agile Testpyramide*

## *Pragmatischer Ansatz*



- Martin Fowler, 2009: [Succeeding with Agile](#)

# *Testpyramide nach Mike Cohn*



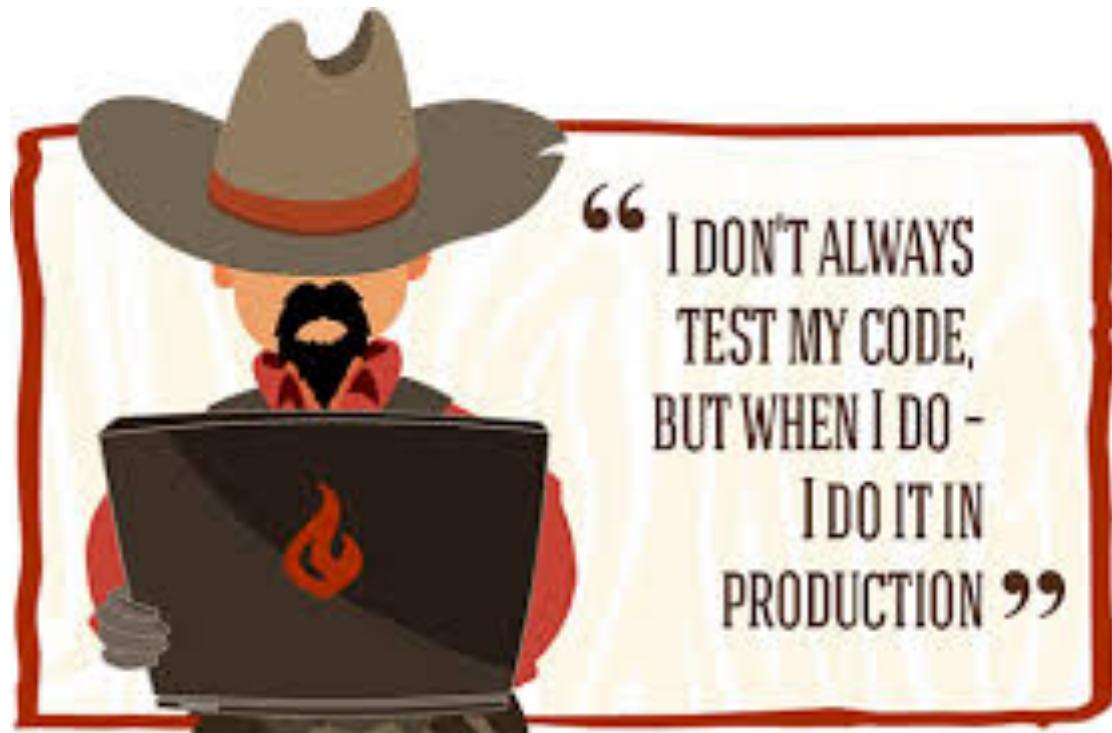
By Mike Cohn: <https://blog.namics.com/2012/06/tour-durch-die-testpyramide.html>

# Inhaltsverzeichnis

## Agiles Testing

1. Motivation: Warum Testing
2. Teststufen und Testtypen
3. Unit-Tests
4. Test Driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests

Dazwischen:  
Weiterarbeit am Prototypen



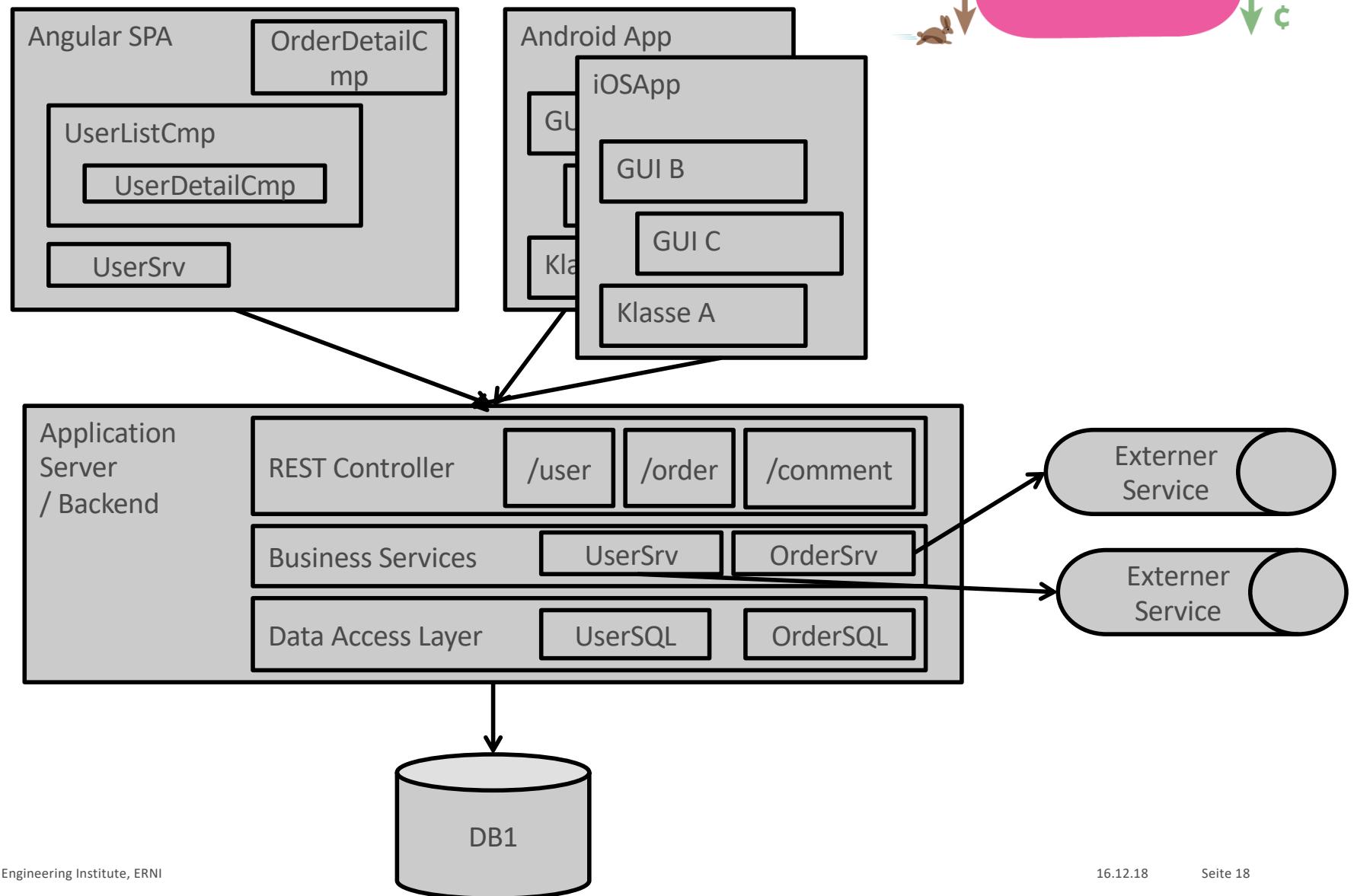
# *Unit testing*

*In Deutsch: Modultest/Komponententest*

- Funktionale Einzelteile werden einzeln getestet
  - Eine Funktion, eine Klasse, eine Komponente
- Unabhängiges Testen
  - Jede Komponente testen ohne die Abhängigkeiten zu andern Komponenten – Abhängigkeiten werden „simuliert“ (Mocks, Stubs, Spys, Drivers)
- Automatisiertes Testen:
  - Unitests werden automatisiert ausgeführt, bei jeder Veränderung des Quellcodes
- Unitests sind Whitebox-Tests
  - Der zu testende Quellcode ist bekannt

# Übung: Teststufen identifizieren

Wo sind Unit Tests sinnvoll?



# *Tools für Unit-Testing*

## *Die wichtigsten Hilfsmittel*

- Javascript / Typescript
  - Unittest-Frameworks:
    - Jasmine: <https://jasmine.github.io> - alt, erprobt, verbreitet (Angular)
    - Mocha: <https://mochajs.org/> - sehr flexibel, verbreitet, async
    - Jest: <https://facebook.github.io/jest/> - schnell (Facebook, React)
    - Ava: <https://github.com/avajs/ava> - schnell, parallel, modern
  - Testrunner:
    - Jedes der obigen Frameworks oder...
    - Karma: <https://karma-runner.github.io> - führt Unit-Tests im Browser aus
- Java / Android
  - Unittest-Frameworks:
    - JUnit 5: <http://junit.org/> alt, erprobt, verbreitet
    - TestNG: <http://testng.org/> - Alternative zu JUnit
    - Arquillian: <http://arquillian.org/> - Java EE, Integration Tests, innovativ
- iOS, Swift, ObjectiveC .net, C# usw.  
-> Viele Optionen -> Google is your friend

# *Übung: Einfacher Unittest in Javascript*

## *Unabhängiger Jasmine Unittest*

Aufgabe: <https://github.com/SEI-Testing/sei-unittest>

- 1. Unit-Tests für Methode fizzbuzz schreiben:
- 2. Allfällige Bugs in der Methode fixen (Hint: da sind mindestens 4 Bugs)

### **Spezifikation:**

- Fizzbuzz soll für ganze, positive Zahlen immer die übergebene Zahl zurückgeben, ausser wenn
  - durch 3 teilbar, dann „fizz“
  - durch 5 teilbar, dann „buzz“
  - durch 3 und 5 teilbar, dann „fizzbuzz“
  - wenn andere Primzahl als 3 oder 5: „prime“
- Für negative Zahlen:
  - Das Produkt mit der Zahl aus dem vorherigen Aufruf der Methode
- Für Zahlen zwischen 0 und 1
  - Das Produkt mit 0.2
- Bei andern nicht ganzen Zahlen
  - Undefined
- Bei übergebenen Strings soll eine Exception geworfen werden mit Meldung „Invalid Argument“

Doku: <https://jasmine.github.io/edge/introduction.html>

# *Probleme mit schlecht geschriebenen Unit Tests in echten Projekten*

- Fragile Unitests („brittle“/„zerbrechlich“)
  - Kleine Änderung am Code -> Dutzende Tests sind Rot
- Zu grosse Tests:
  - Ergebnis: „Etwas stimmt nicht“ hilft nicht beim Debuggen
- Zu kleine Tests:
  - Trivialitäten sollten nicht getestet werden
- Langsame Tests:
  - Wenn die Ausführung 30 Minuten dauert, macht das keiner
- Schlechte Code-Qualität
  - In Tests gelten keine tieferen Standards als für produktiven Code  
-> DRY gilt auch für Testcode -> Kein Copy / Paste usw.
- **Nicht deterministische Tests**
  - Manchmal Rot (Timing Issues, Concurrency Issues, ...)

# **FIRST**

## *Gute Unittests folgen dem FIRST Prinzip*

- Fast: soll schnell in der Ausführung sein: wir haben oft hunderte von Unittests
- Isolated: Unittests sind unabhängig voneinander: Test 2 läuft auch wenn Test 1 nicht gelaufen ist!
- Repeatable: Soll wiederholbar ausgeführt werden können: Ohne dass Datenbank zurückgesetzt werden muss
- Self-Validating: Der Test kann selber eindeutig bestimmen, ob er erfolgreich war oder nicht, kein Mensch muss das Resultat beurteilen.
- Timely: Der Test ist rechtzeitig vorhanden, nicht Wochen oder Monate nach dem Sourcecode

# **AAA: Arrange – Act - Assert**

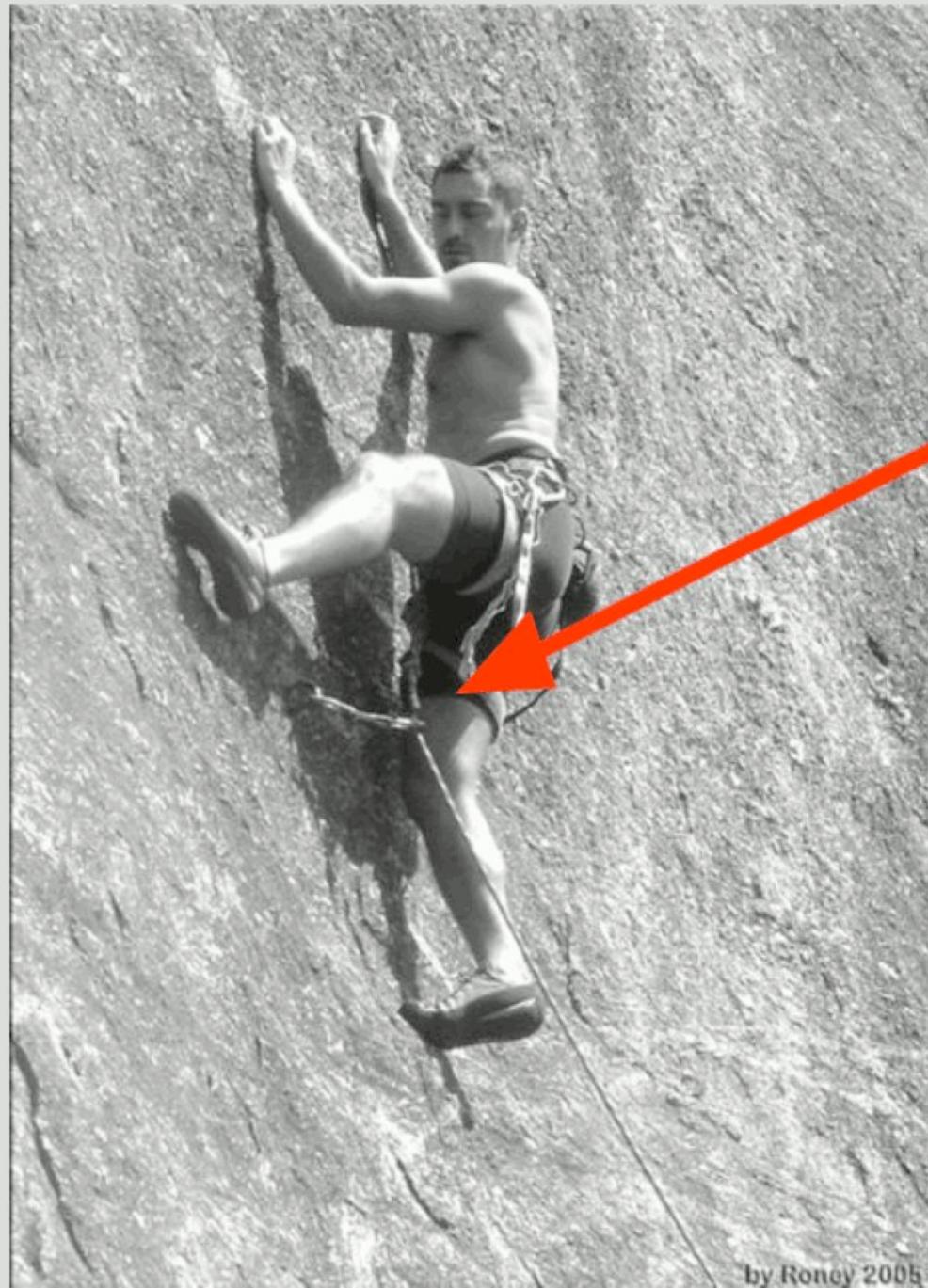
*„Ein guter Unit-Test besteht nur aus einem Aufruf“*

- In einem guten Unittest sind 3 klar erkennbare Schritte vorhanden
  - **Arrange:**      Startsituation wird definiert: Komponente erzeugen, Mocks, Stubs erzeugen usw.
  - **Act:**            Eigentliche zu testende Aktion wird ausgeführt
  - **Assert:**        Einzelnes fachliches Resultat wird überprüft
- 
- Jeder Schritt sollte nur einmal vorkommen  
(in ganz einfachen Tests kann Schritt „Arrange“ fehlen)

# *Parametrisierte Unit Tests*

*Stay DRY in Unit-Testing!*

- Oft muss derselbe Test mit vielen verschiedenen Input-Daten aufgerufen werden.
- Anfänger-Fehler: ganze Test-Methode kopieren.
  
- Lösung:
  - Parametrisierte Tests -> siehe Beispiel im Code
  
- Zu beachten:
  - Reporting soll immer noch pro Ausführung sein  
-> Keine Monster-Tests mit nur einem Resultat: OK/Failed
  
  - Durchlaufzeit/Performance kann wichtig werden



Unit Test

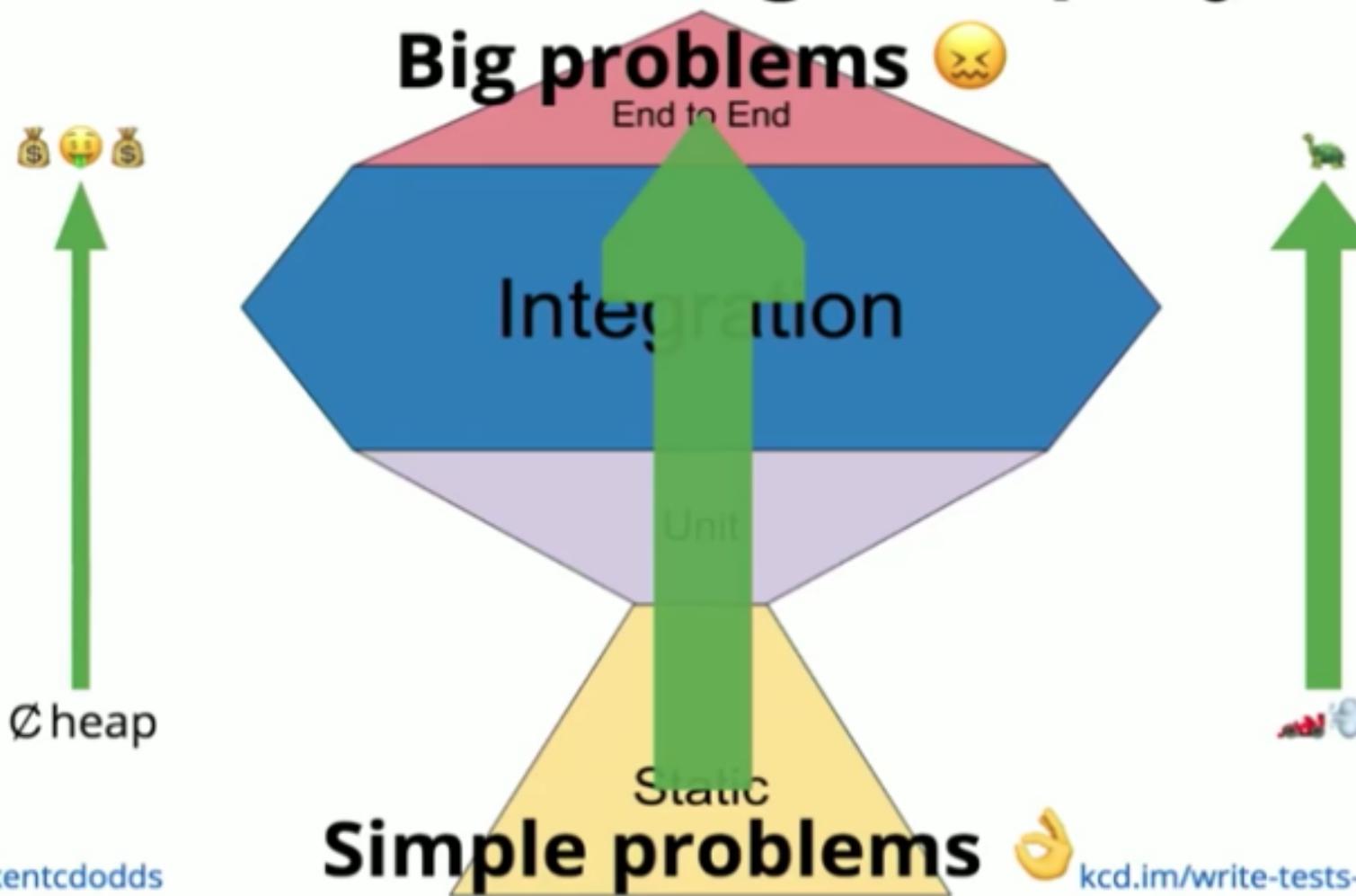
by Roney 2005



*Wert eines Tests*

*Wieviel Vertrauen bringt ein Test?*

# The Testing Trophy



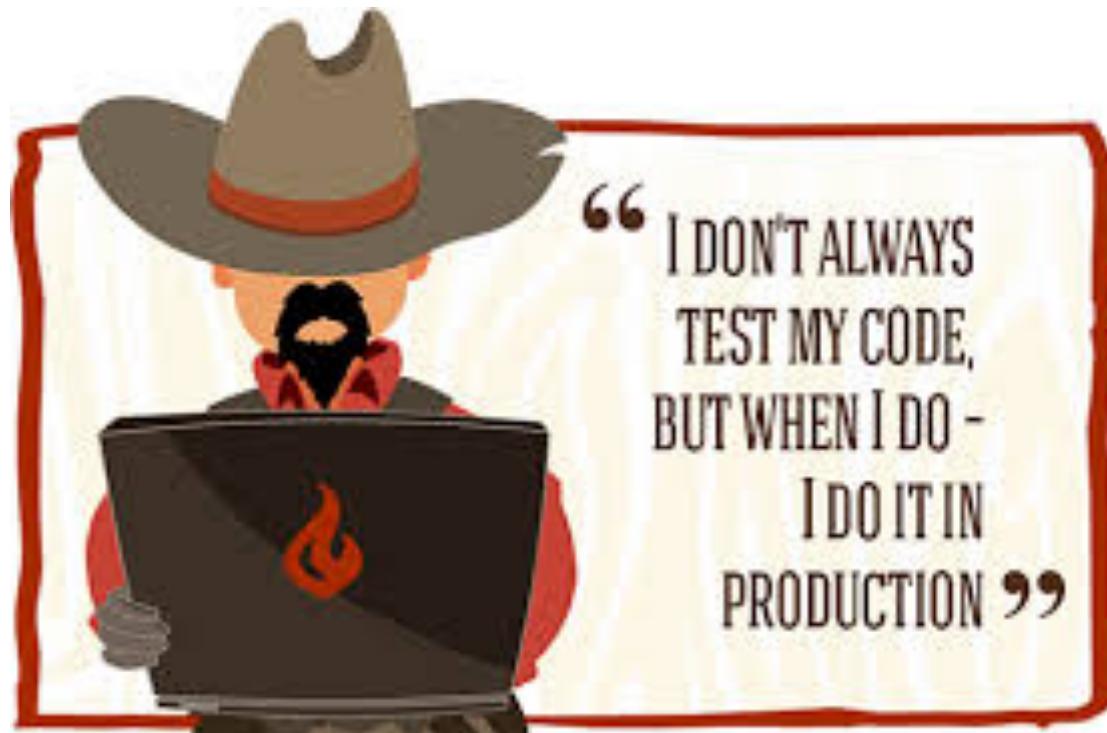
# *Pause – App Entwicklung*

# Inhaltsverzeichnis

## Agiles Testing

1. Motivation: Warum Testing
2. Teststufen und Testtypen
3. Unit-Tests
4. Test Driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests

Dazwischen:  
Weiterarbeit am Prototypen



# *Test Driven Development – Test first*

## *TDD – Entwicklungsmechanik*

# DIE REGELN

- Immer in 3 Schritten arbeiten und wiederholen:
    1. Neuen Test schreiben → Neuer Test ist Rot.
    2. Produktiver Code schreiben → Alle Tests sind Grün
    3. Produktiv- und Testcode aufräumen / refactoren  
→ Tests bleiben grün.

Zurück zu 1

# *Ziele von TDD*

## *Was soll TDD bringen?*

- Sicherheit:  
Sofortiges Feedback bei Änderungen, keine ungewollten Seiteneffekte bei zukünftigen Änderungen.
- Entkopplung:  
Komponenten werden automatisch „testbar“ geschrieben und sind dadurch entkoppelter
- Bessere APIs:  
Entwickler wird zum Denken als Konsument seiner Software gezwungen und schreibt dadurch besser wartbare Software.
- Effizienz:  
Es wird nur entwickelt was auch wirklich notwendig ist
- Beispiel-Code:  
Die Tests sind eine wertvolle Doku des Systems und werden immer up-to-date sein.

# *Übung: TDD Unit converter*

*Wir üben TDD – Ein Einheiten Converter (JS, TS, Java, ...)*

## 1. Anforderungen:

- Funktion, die Längen und Volumen zwischen metrischen und imperial Werten konvertieren kann:
  - m, km, mm <-> inch, foot, yard, mile
  - l, hl <-> quart, pint, gallon
- Beispiel Aufruf (JS): `convert(23, 'km').to('yard')`  
Beispiel Aufruf (Java): `converter.convert(23, "km", "yard")`  
ergibt Resultat: 25153.1
- Soll klare Exceptions liefern bei unbekannten Masseinheiten und unmöglichen Conversionen

## 2. JS: Im neuem Angular Projekt (ng new converter) zwei Dateien erstellen:

- `src/app/converter.ts` -> Die Implementation
- `src/app/converter.spec.ts` -> Die Tests (importiert converter.ts)
- oder: <https://github.com/SEI-Testing/convertTDD.git>
- Ng test : → TDD !

Java: <https://github.com/RBLU/converterjava>

# Mögliche Lösung Unitests

```
import {convert} from './converter';

describe( description: 'converter should convert lengths', specDefinitions: () => {
  it( expectation: 'should convert m to feet', assertion: () => {
    expect(convert( value: 1, fromUnit: 'm').to('ft')).toBe( expected: 3.28084)
  });

  it( expectation: 'should convert km to feet', assertion: () => {
    expect(convert( value: 1, fromUnit: 'km').to('ft')).toBe( expected: 3280.84)
  });

  it( expectation: 'should convert mm to feet', assertion: () => {
    expect(convert( value: 1, fromUnit: 'mm').to('ft')).toBe( expected: 0.00328084)
  });

  it( expectation: 'should convert miles to m', assertion: () => {
    expect(convert( value: 1, fromUnit: 'mile').to('m')).toBe( expected: 1609.34)
  });

  it( expectation: 'should convert pints to l', assertion: () => {
    expect(convert( value: 1, fromUnit: 'pint').to('l')).toBe( expected: 0.55)
  });

  it( expectation: 'should throw when units not match', assertion: () => {
    expect( spy: () =>{convert( value: 1, fromUnit: 'm').to('l')}).toThrow();
  });

  it( expectation: 'should throw exception when unknown fromUnit', assertion: () => {
    expect( spy: () => {convert( value: 1, fromUnit: 'ballba')}).toThrow();
  });

  it( expectation: 'should throw exception when unknown toUnit', assertion: () => {
    expect( spy: () => {convert( value: 1, fromUnit: 'm').to('aasdf')}).toThrow();
  });
});
```

# Mögliche Lösung: Implementation

```
export function convert(value, fromUnit) {
  const units = {
    'm': {factor: 1, cat: 'l'},
    'ft': {factor: 3.28084, cat:'l'},
    'km': {factor: 1/1000, cat:'l'},
    'mm': {factor: 1000, cat:'l'},
    'mile': {factor: 1 / 1609.34, cat:'l'},
    'l': {factor: 1, cat:'v'},
    'hl': {factor: 1/100, cat:'v'},
    'pint': {factor: 1/0.55, cat:'v'}
  };

  if (!units[fromUnit]) {
    throw new Error('unknown fromUnit: ' + fromUnit);
  }

  return {
    to: (toUnit) => {
      if (!units[toUnit]) {
        throw new Error('unknown toUnit: ' + toUnit);
      }

      if (units[fromUnit].cat !== units[toUnit].cat) {
        throw new Error('unit category not matching, cannot convert: from cat:' +
          units[fromUnit].cat + ', to cat:' + units[toUnit].cat);
      }
      return value / units[fromUnit].factor * units[toUnit].factor;
    }
  };
}
```

# *Continuous Integration Demo*

- Live Demo

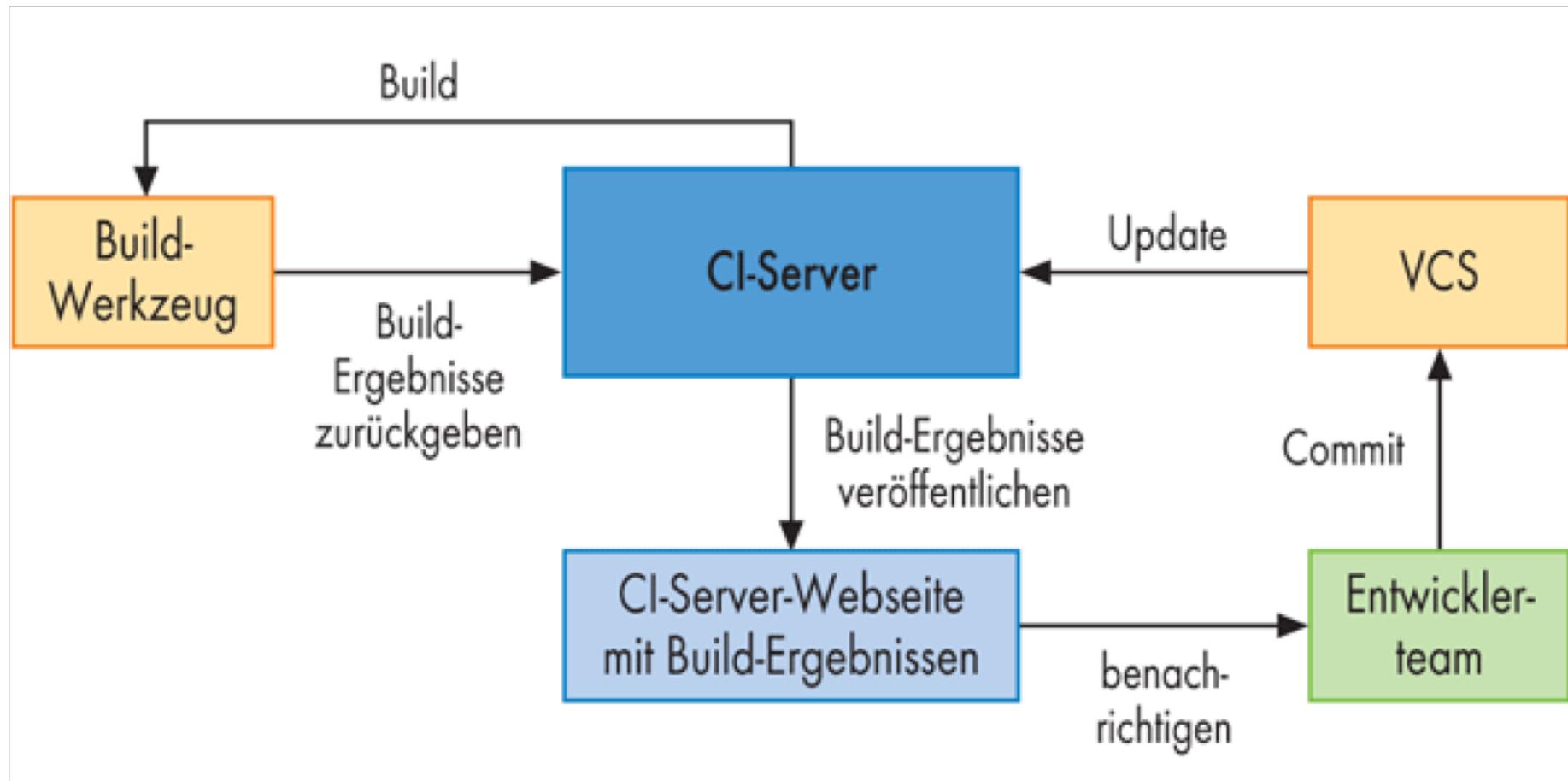
# *Continuous Integration – CI*

## *Wann werden agile automatisierte Tests ausgeführt?*

- In einem idealen Umfeld werden Automatisierte Tests ausgeführt:
  - Bei jedem „Speichern“: Unittests lokal auf der Entwicklermaschine
  - Vor jedem „Check-In“: Integrationstests lokal auf der Entwicklungsmaschine
  - Nach jedem „Check-In“: Unittests und Integrationstests auf sauberen, unabhängigen Maschinen:
    - Verschiedene OS
    - Verschiedene Browser
    - Verschiedene Devices usw.
  - Das Tool dazu: **Continuous Integration Server**

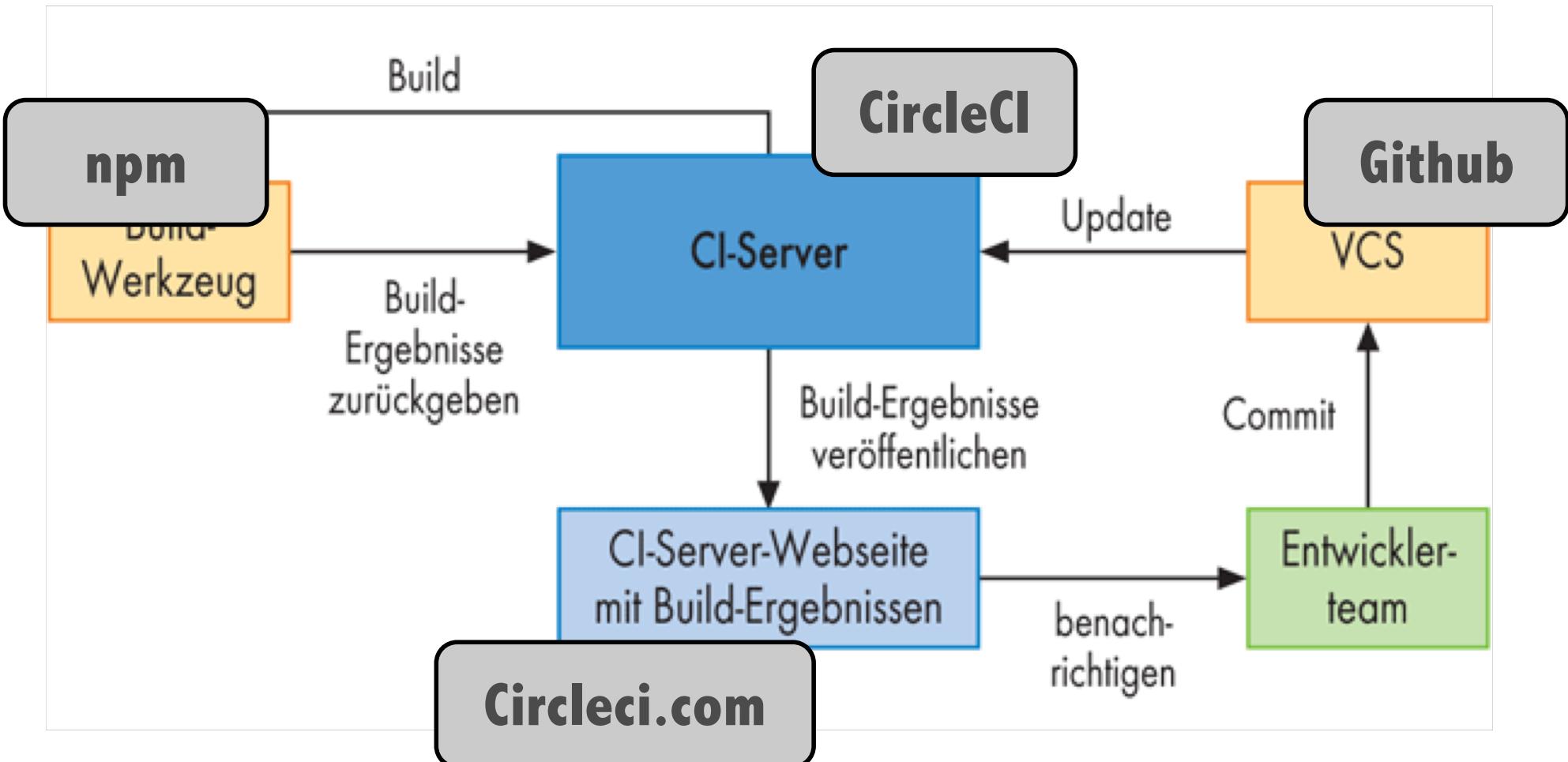
# *CI – Continuous Integration*

## Konzept



# *CI – Continuous Integration*

Beispiel-Anbieter in der Cloud: Gratis – schnell - problemlos



# *CI – Continuous Integration*

## *Andere Anbieter*

- Cloud Services
  - TravisCI
  - CircleCI
  - CodeShip
- Auf eigener Infrastruktur
  - Jenkins (Open Source Software)
  - Teamcity (JetBrains, Gratis für kleine Teams/Open Source)
  - MS Team Foundation Server
- Unterschiede sind klein – alle gut genug:
  - Für Prototypen und OSS ist die Einfachheit von CircleCI und TravisCI genial

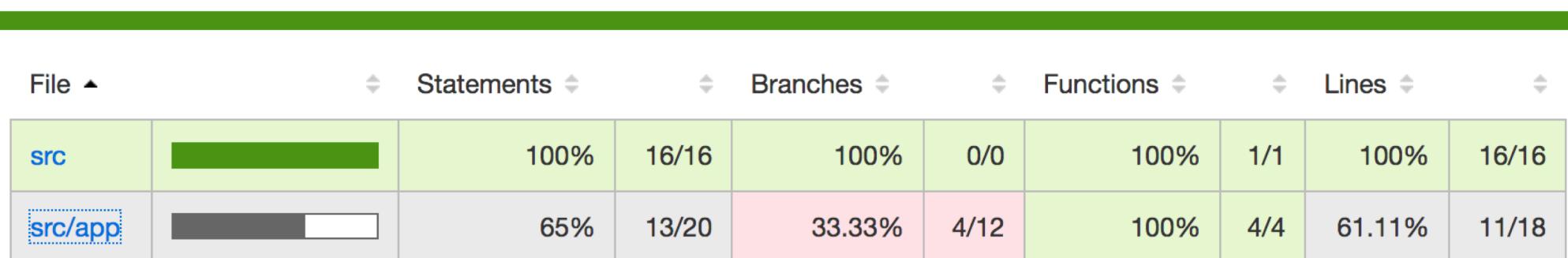
# *Coverage Demo*

## *Mit Angular cli*

- `ng test --code-coverage true`
- Doku: <https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/code-coverage.md>

### All files

80.56% Statements 29/36    33.33% Branches 4/12    100% Functions 5/5    79.41% Lines 27/34



# *Coverage*

## *Messen der Testabdeckung*

- Messen des Fortschritts beim Schreiben von Unittests
- Wie viel meines Codes wird von meinen Unit Tests ausgeführt?
  - In % der Anzahl Statements
  - In % der Anzahl Verzweigungen
  - In % der Anzahl Funktionen
  - In % der Anzahl Zeilen
- Coverage Messung sollte im Build integriert sein und automatisch ausgeführt werden.
- Oft beschließen Entwicklungs-Teams Zielgrößen für Coverage:

Was ist eine gute Zielgröße für Coverage?

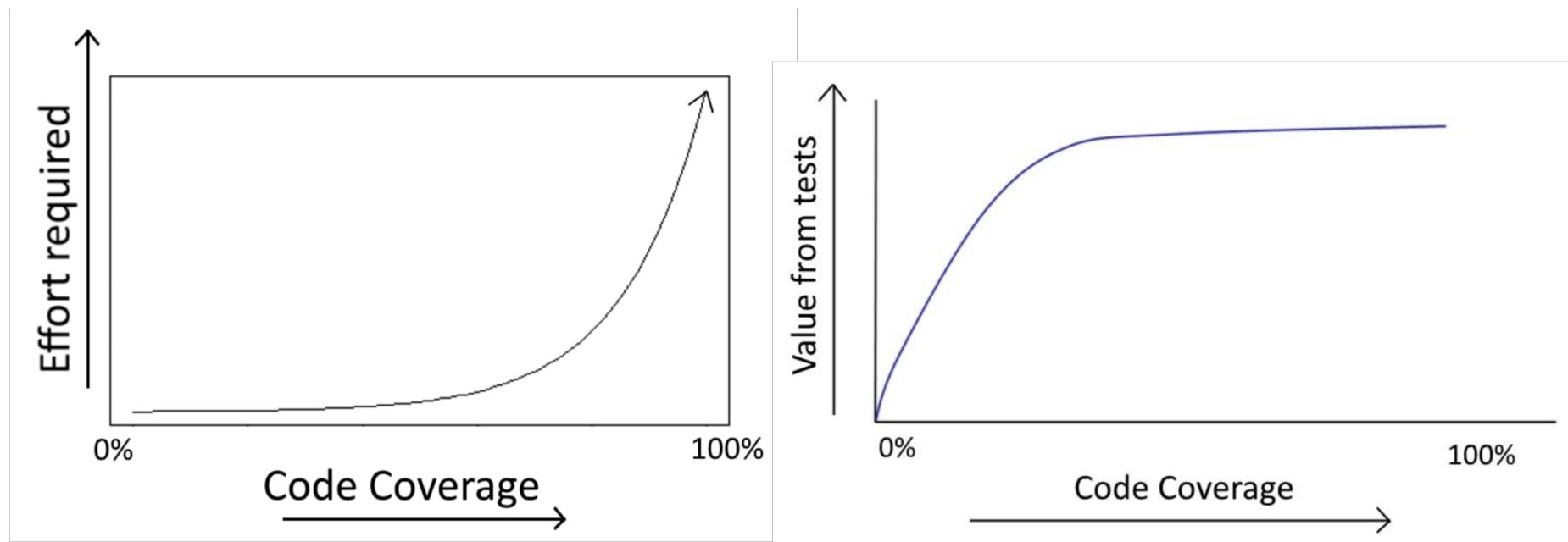
# *Antwort: Keine Coverage ist kein gutes Ziel!*

- Gute Alternative:  
**The Codepipes Testing Metrics (CTM)**

Metric Name	Description	Ideal value	Usual value	Problematic value
PDWT	% of Developers writing tests	100%	20%-70%	Anything less than 100%
PBCNT	% of bugs that create new tests	100%	0%-5%	Anything less than 100%
PTVB	% of tests that verify behavior	100%	10%	Anything less than 100%
PTD	% of tests that are deterministic	100%	50%-80%	Anything less than 100%

*Und wenn doch ein Zielwert*

*Pareto Prinzip: 20%*



# *Coverage Tools*

- Javascript:
  - Istanbul <https://github.com/istanbuljs> (OSS, kostenlos)
- Java:
  - JaCoCo: <http://www.jacoco.org>
  - Viele andere Tools (teilweise nicht mehr gepflegt)
- Objective-C, Swift, Python, usw...
  - Google it.

# *Pause - Arbeit am Prototypen*

# *Aussagen aus dem Testautomatisierungs -Alltag... Einverstanden?*

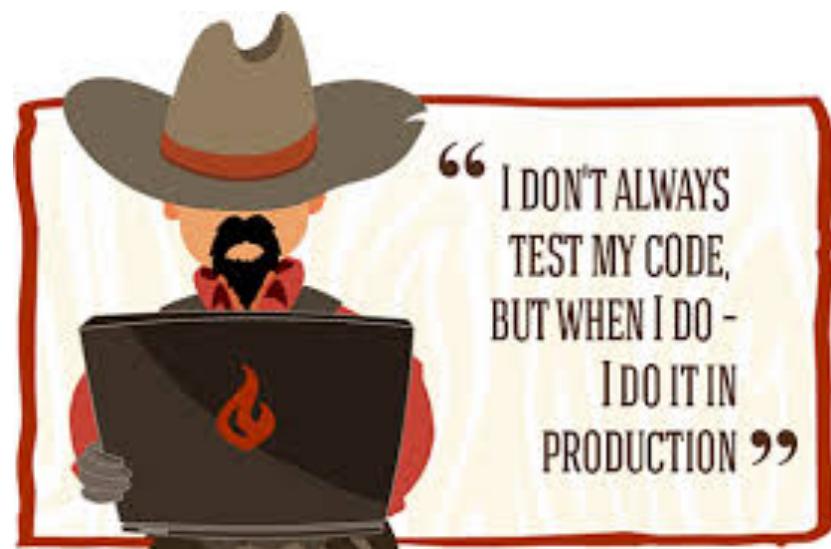
1. Every test should be automated.
2. Automation means we can downsize QA.
3. Automation will catch all bugs.
4. Automation must be written in the same language as the product code.
5. All tests should be such-and-such-level tests.
6. Unit tests aren't necessary because the QA team does the testing.
7. We can complete a user story this sprint and automate its tests next sprint.
8. Automation is just a bunch of “test scripts.”

<https://automationpanda.com/2017/10/01/test-automation-myth-busting/>

# *Inhaltsverzeichnis*

## *Agiles Testing*

1. Motivation: Warum agiles Testing
2. Teststufen und Testtypen
3. Unit-Tests
4. Test driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests



# *Was ist ein Integrationstest?*

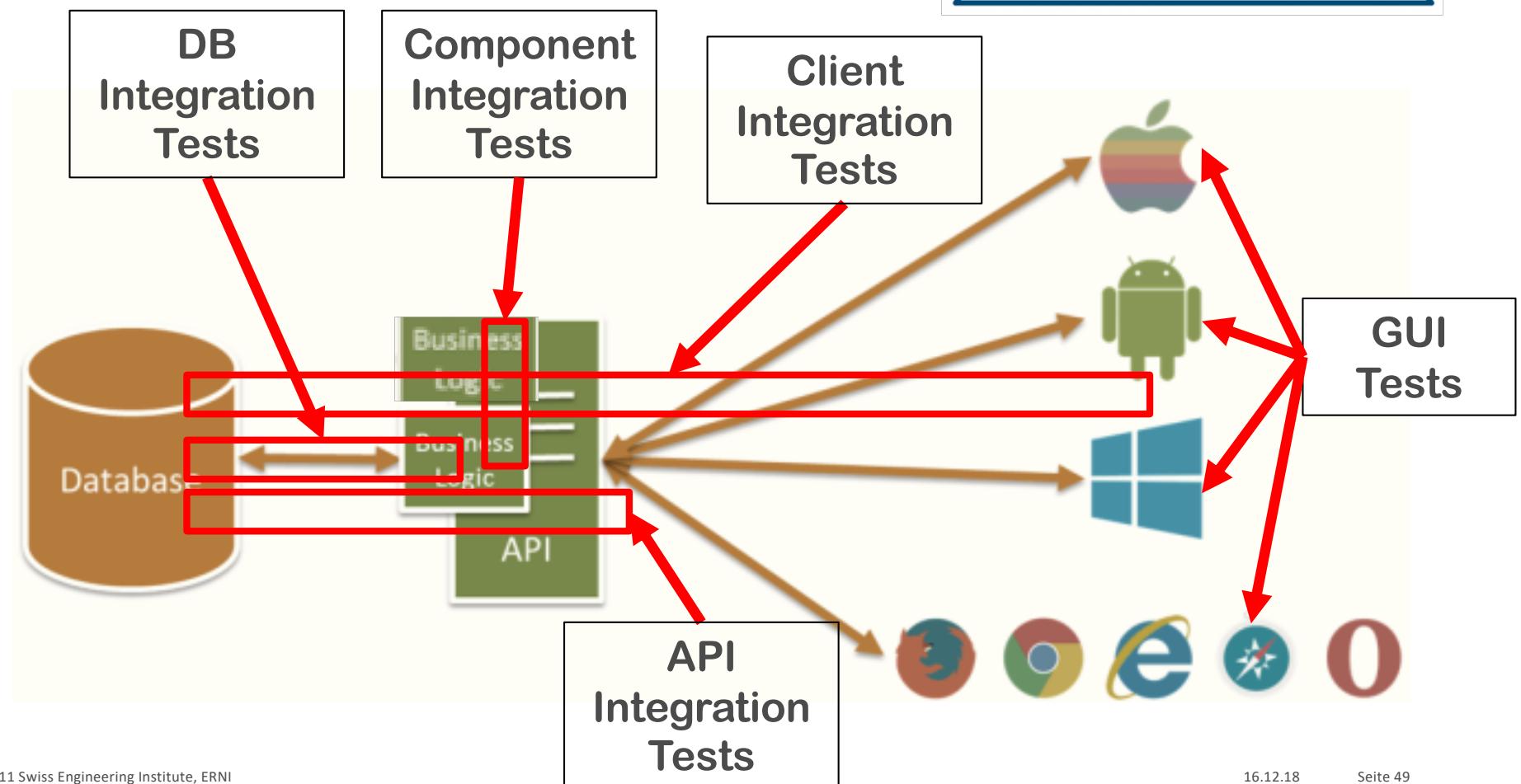
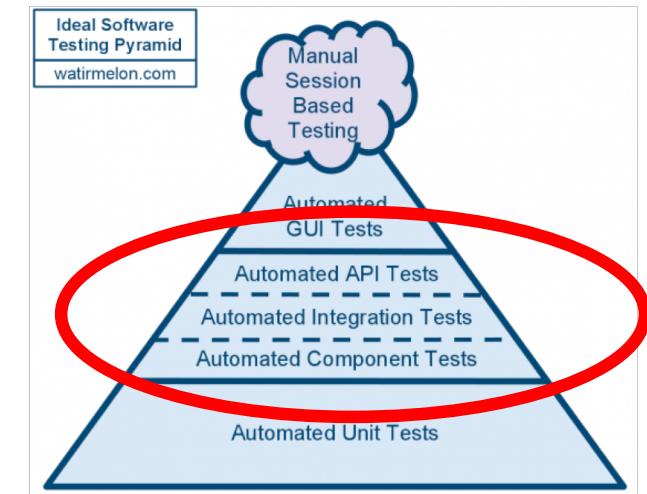
- Integrationstests werden mit denselben Tools geschrieben wie Unit-Tests (JUnit, Jasmine, ...)
- Integrationstests werden ebenfalls automatisiert und in die Continuous Integration eingebaut

ABER.... Ein Test ist immer dann ein **Integrationstest**, wenn

- Der Test eine Datenbank benutzt
- Der Test über das Netzwerk kommuniziert
- Der Test ein externes System nutzt (Mail-Server, Queue, usw)
- Der Test Files liest oder schreibt (oder anderes I/O macht)
- Der Test nicht auf Source-Ebene basiert sondern auf deployten Binaries
- Der Test mehrere Komponenten auf einmal testet

# Integrationstests

## Verschiedene Arten



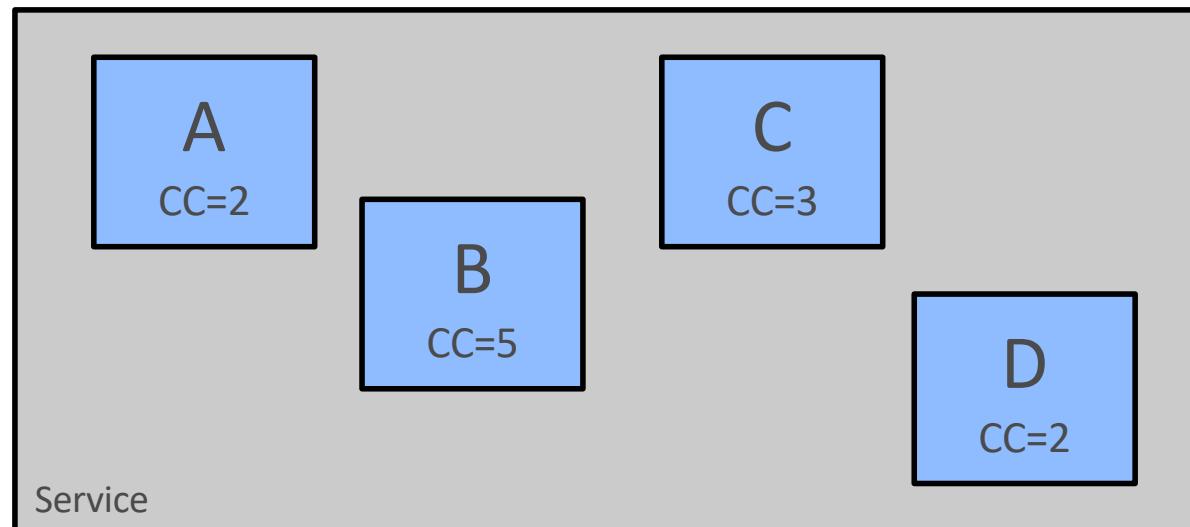
# *Service / API Level Integrations-Tests*

## *Automatisierte Service Tests*

- Typischer Einsatz von API-Tests:
  1. Integrationstest von einzelnen Komponenten:
    - z.B. Testfälle für Endpunkt “/user” (GET, PUT, POST, DELETE)
  2. End-to-End Test-Szenarien
    - z.B. Login -> Katalog -> Produkt auswählen -> Zahlen
- API-Tests sind Black-Box Tests – die Implementation ist nicht sichtbar für den Test
- In heutigen Architekturen mit RESTful APIs und Browser / Mobile Clients oft die wichtigste Art von Integrations-Tests, weil:
  - Gut automatisierbar
  - Gutes Kosten / Nutzen Verhältnis
  - APIs sind oft früh im Projektlauf relativ stabil
  - Oft mit denselben Tools automatisierbar wie Unit-Tests:
    - Junit, Jasmine, Mocha usw.

# *Unit-Tests und Integrationstests*

*Brauchen wir beides?*



## **Cyclomatic Complexity (CC)**

auf wieviele unterschiedlichen Pfaden kann ein Software-Modul (z.B. eine Klasse oder eine Methode in Java) durchlaufen werden

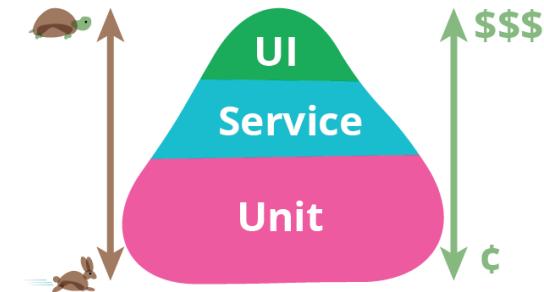
# *Integrations- und Unitest*

## *Camp Übung 3*

- <https://github.com/SEI-Testing/unitintegration.git>
- Teste und erweitere den UserService
- 2 Übungen:
  - Schreibe einen Unittest -> Benötigt Mock für die Db
  - Schreibe einen Integrationstest -> Benötigt laufende Postgresql DB
- Alternative: Unit und Integrationstest für Spring-Boot:
  - <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html>

# *Unitests vs. Integrationstests*

*Warum brauchen wir beides?*



- Nachteile von Unitests:
  - Testen keine Interaktionen zwischen Komponenten / keine Schnittstellen
  - Zwingen zu Entkopplung und „komplizierterem Design“ und „Dependency Injection“
  - Testen nur, was der Entwickler erwartet hat
- Nachteile von Integrationstests:
  - Sind teuer in 3 Aspekten:
    - Langsam (Browser und DB starten, DB zurücksetzen, Server starten....)
    - Fragil/zerbrechlich
    - Schwierig zu schreiben und sehr schwierig zu debuggen
  - Können keine Fehlerzustände simulieren
  - Zeigen nicht WO ein Fehler ist.

**Software-Entwicklung mit hoher Qualität benötigt zwingend beides!**

# *Snapshot Testing (Golden Master Testing)*

## *Nützliches Konzept*

- Anstelle von Assertions im Test-Code wird ein “Resultat” als Snapshot abgespeichert:
  - JSON-Antwort eines API Calls
  - GUI-Screenshot
  - Gerenderter HTML Code
  - ..... jegliches andere serialisierbare Resultat kann Snapshot sein
- In jedem automatisierten Testlauf wird dasselbe Resultat wieder produziert und mit dem gespeicherten Snapshot verglichen.
  - Wenn identisch -> Test grün
  - Wenn nicht identisch -> Test rot
    - Bei beabsichtigtem Change: -> Snapshot updaten
    - Bei unbeabsichtigtem Change: -> Code fixen
- Testet nicht „Korrektheit“ sondern schützt vor „unbeabsichtigtem Change“

# *Snapshot Testing*

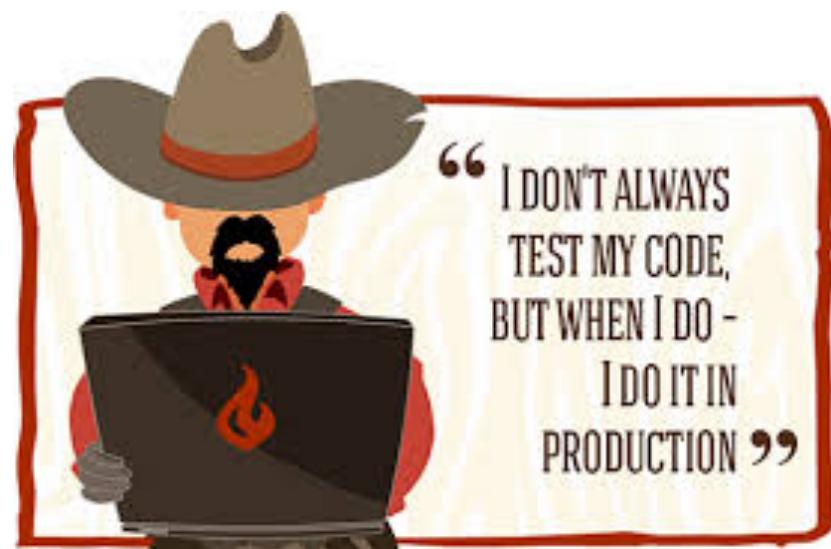
## *Wann sinnvoll? Wie*

- Snapshot-Testing ist sinnvoll wenn:
  - Die Anwendung/die Komponente sinnvolle als Snapshot verwendbare Resultate produziert (oder diese einfach im Test zu produzieren sind)
  - Die Snapshots **deterministisch** sind:
    - Screenshot mit Datum/Zeit vergleichen geht nicht
    - Zähler, timestamps, UUIDs usw sind “schwierig”
  - Snapshots relativ stabil sind über die Zeit
    - Kleine Änderung und alle Snapshots rot ist ineffizient.
- Tools:
  - Javascript/TypeScript: JEST Framework: Text-Vergleich (HTML, JSON, ..)
  - Depicted—dpxdt: perceptual Screenshot diff (by Google)
  - Kommerziell:
  - Applitools, ApprovalTest, TextTest, ReTest

# *Inhaltsverzeichnis*

## *Agiles Testing*

1. Motivation: Warum agiles Testing
2. Teststufen und Testtypen
3. Unit-Tests
4. Test driven Development
5. Continuous Integration
6. Coverage
7. Integrations-Tests
8. GUI Tests



“ I DON'T ALWAYS  
TEST MY CODE,  
BUT WHEN I DO -  
I DO IT IN  
PRODUCTION ”

# *GUI Level Integration-Tests*

## *GUI-Test, ...*

- „Automatisierter GUI Test“:
  - Aufgezeichnete und/oder programmierte Simulation eines Benutzers, der die Anwendung auf der GUI benutzt, mit automatischer Überprüfung des Resultats/des Outputs.
- Wird auch als End-2-End-Test bezeichnet.
- „Königs-Disziplin“ des automatisierten Testings:
  - Schwierig gut zu machen
  - Scheitert immer noch oft in Projekten
- Kann sehr viel manuelle Arbeit einsparen, wenn die Testfälle gut aufgebaut sind.
- „Aufzeichnen“ oder „Schreiben“?
  - Aufzeichnen klappt im allgemeinen längerfristig nicht!

# Tools

- Es gibt unzählige GUI Automation Testing Tools
- **Open Source**
  - Selenium
    - + viele AddOn/Wrappers darüber wie
      - Protractor
      - Selenide
      - ...
  - TestCafe
- **Kommerziell:**
  - HP Unified Functional Testing (UFT) (früher: HP Quicktest Pro, QTP9)
  - Tricentis TOSCA TestSuite
  - MS VisualStudio Coded UI Tests
  - Micro Focus SilkTest (sehr erprobtes altes Tool)
  - QFTest: gut und teuer
  - Und viele mehr.... : <https://www.testing-board.com/testautomatisierung-tools/>

# Selenium

## Der „Open Source Standard“



- Selenium Webdriver:
  - API um Browser fernzusteuern mit vielen Implementationen:
    - ChromeDriver, EventFiringWebDriver, FirefoxDriver, HtmlUnitDriver, InternetExplorerDriver, PhantomJS, WebDriver, SafariDriver
  - API kann von vielen verschiedenen Sprachen aus aufgerufen werden:
    - Java, Javascript, C#, Python, Ruby, Perl, usw.
  - Testfall-Struktur wählbar mittels Unit-Test Framework
    - Junit, Jasmine, Mocha....
  - Es gibt viele AddOns/ Wrapper zu Selenium, die die Benutzung vereinfachen für gewisse Szenarien:
    - Protractor (für Angular Apps)
    - Selenide
    - ...
  - Doku: [http://www.seleniumhq.org/docs/03\\_webdriver.jsp](http://www.seleniumhq.org/docs/03_webdriver.jsp)

# Beispiel Selenium Java

```
public class Selenium2Example {
    public static void main(String[] args) {
        // Create a new instance of the Firefox driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.
        WebDriver driver = new FirefoxDriver();

        // And now use this to visit Google
        driver.get("http://www.google.com");
        // Alternatively the same thing can be done like this
        // driver.navigate().to("http://www.google.com");

        // Find the text input element by its name
        WebElement element = driver.findElement(By.name("q"));

        // Enter something to search for
        element.sendKeys("Cheese!");

        // Now submit the form. WebDriver will find the form for us from the element
        element.submit();

        // Check the title of the page
        System.out.println("Page title is: " + driver.getTitle());

        // Google's search is rendered dynamically with JavaScript.
        // Wait for the page to load, timeout after 10 seconds
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().toLowerCase().startsWith("cheese!");
            }
        });

        // Should see: "cheese! - Google Search"
        System.out.println("Page title is: " + driver.getTitle());

        //Close the browser
        driver.quit();
    }
}
```

# *Beispiel: Selenium*

## *Javascript - NodeJS*

```
npm install selenium-webdriver
```

```
const {Builder, By, Key, until} = require('selenium-webdriver');

var driver = new Builder()
  .forBrowser('firefox')
  .build();

driver.get('http://www.google.com/ncr')
  .then(() =>
    driver.findElement(By.name('q')).sendKeys('webdriver', Key.RETURN))
  .then(() => driver.wait(until.titleIs('webdriver - Google Search'), 1000))
  .then(() => driver.quit());
```

# *PageObject Pattern*

- Wichtige Separierung
  - Selektoren/Layout einer Page kommen in ein PageObject
  - Testablauf bleibt im Test
- Ziel:
  - Aller Code, der abhängig ist vom Aufbau/Layout einer Page, wird an einem Ort gekapselt
  - Wenn die Anwendung ändert muss nur eine Stelle im Test angepasst werden.
- Beispiel:
  - <https://github.com/SeleniumHQ/selenium/wiki/PageObjects>

# *Übung: Selenium Test*

*Alternative: TestCafe Test (<https://DevExpress.github.io/testcafe/>)*

- Einfacher Test implementieren mit Selenium
  - Java oder Javascript
  - Browser nach Wahl
- Test einer „beliebigen Website“ oder eigenes Projekt
- Testfall nach folgender Struktur:
  1. Auf bestimmte Seite navigieren
  2. Input Feld abfüllen
  3. Form submitten
  4. Auf der Resultatseite eine Prüfung des Resultats vornehmen
- Test auf mobile Devices ausführen mittels  
<https://www.browserstack.com>

# *Was testet der GUI-Test (nicht)!*

- Assertions sind sehr wichtig:
  - Zeit investieren in die richtigen Assertions
    - Existenz eines Feldes?
    - Button disabled / enabled? Logik?
    - Text? Sprachen?
  - Was ist sinnvoll auf der GUI zu verifizieren?
- Layout: Wie kann das Layout automatisch getestet werden?
  - Soll man überhaupt?
  - Möglichkeit:
    - Snapshot-Testing mit Screenshot-Vergleich
    - Tool: Depicted—dpxdt: perceptual Screenshot diff (by Google)

# *Probleme mit automatisierten GUI Tests*

## *Warum scheitern viele solche Initiativen?*

- **Unzuverlässigkeit**

Oft führen kleine Timing Probleme zu Failed Tests, obwohl eigentlich alles ok ist (Andere App im Vordergrund, Virenschanner, ...)

- **Langsamkeit**

GUI-Tests sind ein vielfaches langsamer als API- oder Unit-Tests, man hat schnell stundenlange GUI-Test-Suiten

- **Zerbrechlichkeit**

Kleine Design-Änderung am GUI -> Dutzende von Testfällen sind rot und müssen angepasst werden

- **Schwierige Ausgangsbedingungen**

Jeder Test sollte wiederholbar von einem immer gleichen Zustand des Systems ausgehen können -> Datenbank Zustand?

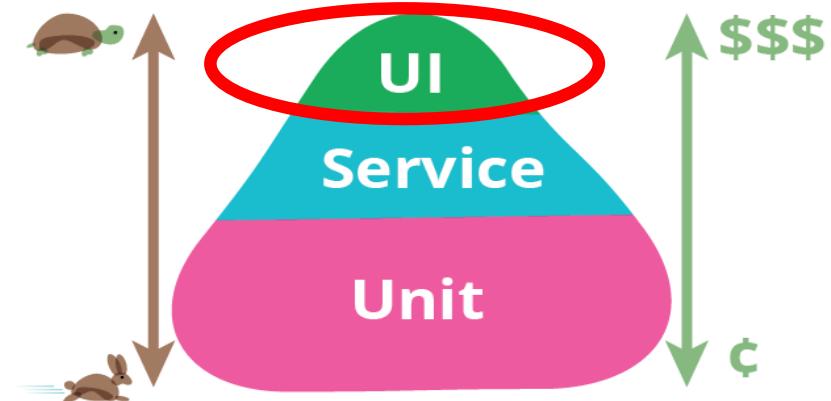
- **Viel Redundanz**

Jeder Test beginnt mit Login, Navigation, Daten laden, usw. bis er endlich das Testen kann, was er soll.

# Lösungsansätze

## Für gute automatisierte GUI-Tests

- Wenige Tests – aber die wichtigen
  - Spitze der Test-Pyramide!
  - Nur Happy-Paths im GUI-Test
    - Alles andere im API-Test / Backend Unit Test / Frontend Unit Test
- GUI-Tests „designen wie Software“
  - Wiederverwendbare Schritte nur einmal implementieren und in vielen Tests benutzen
  - Page-Objekte zur Abstraktion zum schnell ändernden Layout
    - [http://www.seleniumhq.org/docs/06\\_test\\_design\\_considerations.jsp#page-object-design-pattern](http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern)
- GUI-Tests integriert im CI
  - Laufen bei jedem Check-In, nicht nur vor dem Release



# *Software Testing Anti-patterns*

- Having unit tests without integration tests
- Having integration tests without unit tests
- Having the wrong kind of tests
- Testing the wrong functionality
- Testing internal implementation
- Paying excessive attention to test coverage
- Having flaky or slow tests
- Running tests manually
- Treating test code as a second class citizen
- Not converting production bugs to tests
- Treating TDD as a religion
- Writing tests without reading documentation first
- Giving testing a bad reputation out of ignorance

Quelle: <http://blog.codepipes.com/testing/software-testing-antipatterns.html>

# Feedback

## Feedback zum Kurs

- Haben wir die Lernziele erreicht?

### Kursziele

#### Agiles Testing



- Ich weiss warum Entwickler automatisiert testen
- Ich kenne die verschiedenen Teststufen und Testtypen
- Ich kann einen Unit Tests mit Javascript/Jasmine schreiben
- Ich habe TDD (Test Driven Development) ausprobiert
- Ich kann System Tests mit Protractor/Selenium schreiben
- Ich habe automatisches Unit Testing in unserem Prototypen integriert.

- Was hätte euch auch noch interessiert?
- Was können wir besser machen?
- Vielen Dank

Bei Fragen, Kommentaren, usw:

Reto Blunschi, [reto.blunschi@creagy.ch](mailto:reto.blunschi@creagy.ch)

Phone: +41 79 287 1848