Unit Tests 1

Fahrplan

- **Einstieg ins Testing** (Was ist das? Was bringt das?)
- **Unit Tests mit Jest**

TECH STARTER

Einstieg ins Testing

Warum testen wir Software?

Überprüfen einer Software:

- Ob sie sich verhält wie erwartet
- Ob Fehler auftreten
- Um Wahrscheinlichkeiten für Fehler zu minimieren
- Um Qualität zu gewährleisten
- Um langfristige Kosten zu minimieren

TECH STARTER

Was Fehler in Software anrichten kann

Unsaubere Urlaubsplanung

- American Airlines hat zu viele Piloten gleichzeitig in den Urlaub gelassen => 15.000 Flüge ohne Piloten (link)

Lebensgefährlich

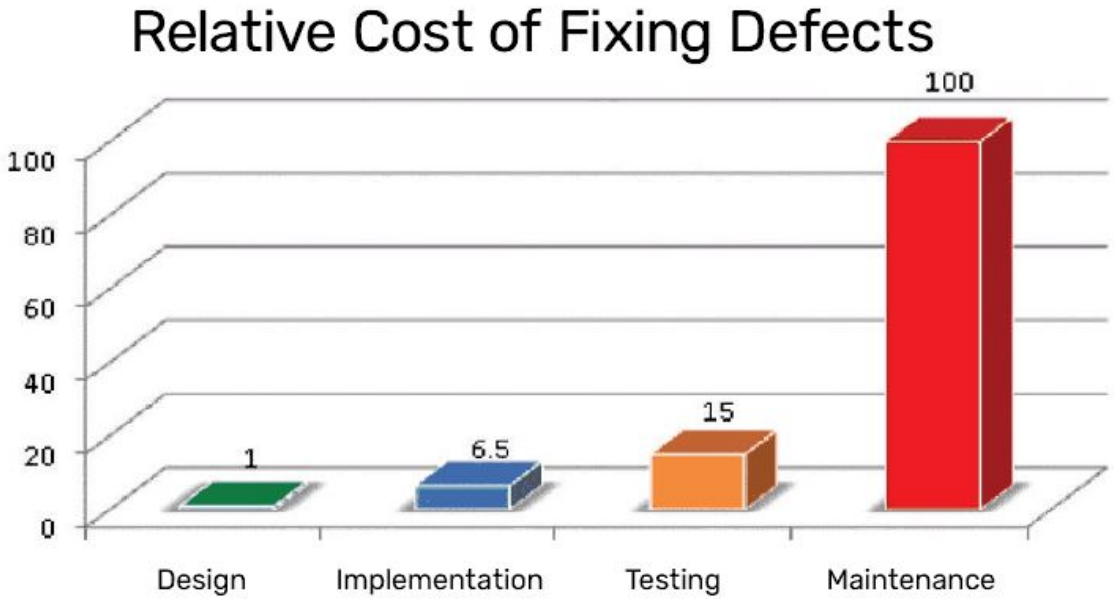
- Tausende Patienten in Großbritannien haben falsche Medikamente erhalten (link)

Raketen-Projekte in die Luft gesprengt

- In den 90er Jahren sind mehrfach Raketen aufgrund von Softwarefehlern fehlgeschlagen (link)

TECH STARTER

Kosten von Fehlern in verschiedenen Stadien



IBM System Science Institute

TECH STARTER

Vorteile von Tests

- Fehler erkennen
- Qualität sichern
- Benutzerzufriedenheit
- Risiko minimieren

Nachteile von Tests

- Zeitaufwand
- Kosten
- Zeigen Anwesenheit von Fehlern, **nicht** die Abwesenheit

TECH STARTER

Testarten

- Unit-Tests
- Integrationstest
- Systemtests
- Akzeptanztests
- Security Tests
- Performance Tests
- …

TECH STARTER

Was sind Unit Tests?

Unit Tests zielen auf die *kleinsten Einheiten (Units)* ab:

- Funktionen oder Methoden
- Klassen

Ziel:

- Jede Unit des Codes funktioniert fehlerfrei und wie erwartet

Stabile Bausteine => Stabiles Fundament => Stabiles Haus

TECH STARTER

Eigenschaften von Unit Tests

Unit Tests:

- Testen Bestandteile möglichst **isoliert** von anderen Units (*später dazu: Mocking*)
- Werden **automatisiert** ausgeführt
- Sind **schnell**
- Sind **reproduzierbar** (unabhängig von Umgebung)

TECH STARTER

Unit Tests in NodeJS mit Jest

Jest ist ein JavaScript Testing Framework, welches Tests möglichst lesbar und übersichtlich macht.

Testing Frameworks:

- Bieten lesbaren Syntax für Tests

```
test("adds 1 + 2 to equal 3", () => {
  const sum = calc.sum(1, 2);
  expect(sum).toBe(3);
});
```

“Erwarte, dass sum 3 ist”

- Finden Test-Dateien (häufig mit “test” im Namen, konfigurierbar)
- Führen Tests isoliert aus <= einzeln und unabhängig von anderen Test

TECH STARTER

Unit Tests in NodeJS mit Jest

Wie nutzen wir dieses?

- `npm i jest`;
- In scripts von package.json:
  `"test": "jest"` <= eventuell mit Optionen wie `--coverage`
- *.js Dateien mit zu testendem Code füllen
- Dateien die mit *.test.js enden mit Tests füllen
- `npm run test`

```
PASS unit-testing/calculator/calculator.test.js
  √ adds 1 + 2 to equal 3 (7 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.979 s, estimated 1 s
Ran all test suites.
```

TECH STARTER

Best Practices

- Für jedes Modul/jede Datei im Quellcode erstellen wir eine zugehörige Test-Datei
  Beispiel: `apfel.js + apfel.test.js`

- Aussagekräftige TestNamen!

❌

```
test("Test 433"), () => {};
test("Error"), () => {};
test("Leerer Array"), () => {};
```

✅

```
test("Summe von 100 und 333 ist 433"), () =>{ }
test("Erstellen von User ohne Passwort wirft Error"),
test("Leerer Array ist sortiert"), () =>{ }
```

TECH STARTER

Best Practices

Lesbare Test

- Aussagekräftige Namen
  (auch für Variablen)

- Lesbarkeit > Codedichte
  => Code lieber aufteilen als
  alles in eine Zeile zu quetschen

- Gute Praxis: *Arrange Act Assert*

```
test("adds 1 + 2 to equal 3", () => {
  //Arrange
  const [a, b] = [1, 2];

  //Act
  const result = calc.sum(a, b);

  //Assert (bzw. Expect)
  expect(result).toBe(3);
});
```

TECH STARTER