Test-Driven Development

Softwerkskammer Rostock, 31.01.2018

Inhalt

1. Geschichte
2. Mantra
3. Die 3 Gesetze
4. Tests
    a. Benefits, Herausforderungen und Anatomie
5. Bedenken
6. Anti Pattern
7. TDD und XP

Eine kurze Geschichte

1960

Fest first punch cards

Mercury Space Program schrieb erst den zu erwartende Ausgabe und dann den “Code”

1989

Kent Beck

Liest über Mercury

1995

TDD Demo

Kent zeig Ward Cunningham eine Demo

1999

eXtreme Programming

Test-first programming

In XP, sofern möglich, werden Tests vor der Implementation geschrieben

Test sind so wichtig wie die Programmierung

2000

JUnit

Vereinfachen von Unit-Tests

2002

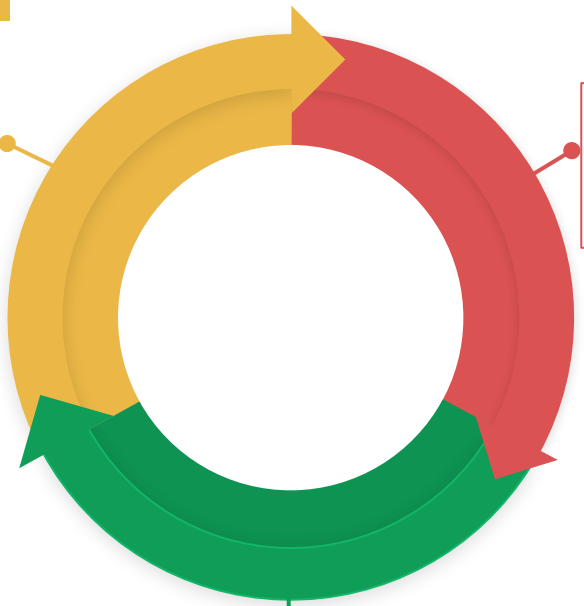
TDD Buch

Test-driven development by example

Das Mantra

1. Schreibe einen Test der nicht läuft
2. Bring ihn schnell zum Laufen
3. Refactor

Red Green Refactor



REFACTOR

Aufräumen
Keine neue Funktionalität

RED

Wunschinterface
Nicht mehr als erforderlich

GREEN

Gerade so viel das der Test grün wird

ROT

- Einfach starten (KISS)
- Regression tests
  - Bug gefunden, schreibe Test um diesen zu “beweisen”
- Dauert zu lange bis zum Grün, mach einen kleineren Tests
- Wenn du alleine arbeitest, höre auf mit einem fehlerhaften Test
- Im Team verlasse mit grün

Mock

```
class UnixFS:
    @staticmethod
    def rm(filename):
        os.remove(filename)

def test_unix_fs(mocker):
    mocker.patch('os.remove')
    UnixFS.rm('file')
    os.remove.assert_called_once_with('file')
```

GRÜN

- Fake it
  - Start mit einer Konstante für die Funktion
- Offensichtliche Implementation
  - Manche Dinge sind offensichtlich, bring es schnell hinter dich
- One to many
  - Bei collections fang an mit einem Element
- Fixture
  - Entferne gleiche Initialisierungen in Tests

Fixture

```
@pytest.fixture
def smtp():
    import smtplib
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

def test_ehlo(smtp):
    response, msg = smtp.ehlo()
    assert response == 250
```

GRÜN

Example-based test
(Triangulate)

Tests	Implementation
Specific/arbitrary	Specific
↓	↓
Triangulated	Generic

Property-based test
(Fuzzing / Generierte Daten)

Tests	Implementation
General	Specific/arbitrary
↓	↓
Specific	Generic

Bibliotheken

- Java: JUnit-Quickcheck
- .NET: FsCheck
- JavaScript: JSVerify
- Ruby: Rubycheck
- PHP: PhpQuickCheck
- Scala: ScalaCheck
- Haskell: QuickCheck
- Python: Hypothesis

Die 3 Gesetze

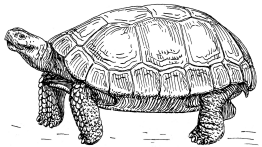
Die 3 Gesetze

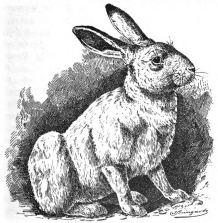
1. Du darfst erst dann Produktivcode schreiben, wenn vorher ein scheiternder Unit-Test geschrieben wurde
2. Du darfst nicht mehr von einem Unit-Test schreiben als man für das Scheitern brauch - Scheitern ist auch nicht kompilierender Code
3. Du darfst nicht mehr Produktivcode schreiben, als nötig ist, um den aktuell misslingenden Unit-Test zu bestehen

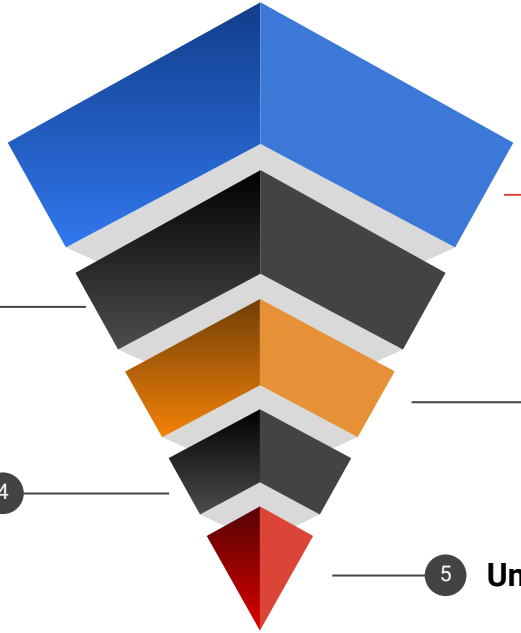
Weitere Aspekte

- Don’t repeat yourself (DRY)
- Keep it simple stupid (KISS)
- You aren’t gonna need it (YAGNI)
- Mehr als Test-First programming
- Unit-tests
  - Integration, functional, acceptance

Test Eistüte







1 Manual

System 2

3 Integration

Component 4

5 Unit

Warum Tests

- Bugs sind teuer, diese zu beheben auch
  - Defect Cost Increase (DCI)
- Früh, oft und automatisiert
- Bug gefunden > Test, Fix, Grün!
- Double-checking: Test sagt was ich erreichen möchte, die Software setzt es um

Benefits

- Ein funktionierendes System (Zu jeder Zeit!)
- Code Coverage
- Dokumentation
- Keine Angst vor Änderungen
- Vereinfacht Refactoring
  - Keine neuen Bugs oder vergessene Features
- Leicht zu testender Code
- Weniger Zeit im Debugger

Benefits

- Zwingt den Entwickler mehr über seinen Code nachzudenken
- Metrik für Fortschritt (Wieder 10 neue Tests heute geschafft, Beweis Feature Complete)
- Was du heute kannst besorgen verschiebe nicht auf morgen!
  - Man ist im Kontext, später nachher nicht
- Schneller Feedback Loop

Herausforderungen

- Disziplin
  - Die Tests schreiben sich nicht von alleine
- Legacy Code
- Erfahrung - Kenne deine Testmöglichkeiten
- UI, Netzwerk, Embedded, Unklare Anforderungen
- Alle oder (k)einer (Broken-Windows-Theorie)
- Tests sollten durch QA erweitert/validiert werden
- Ersetzt nicht: Performance, Stress, Usability

Woran erkenne ich gute Tests

- Setup vor dem Test ist minimal
- Häufig verwendete Setups an einem gemeinsamen Ort
- Tests laufen schnell
- Nicht zerbrechliche Tests, Wartbar
  - Reihenfolge sollte egal sein, Bonus Punkte wenn parallelisierbar
- Automatisch
- Wiederholbar
- Läuft überall

FIRST

FAST

ISOLATED

REPEATABLE

SELF-VALIDATING

TIMELY

Anatomie eines Tests

```
@Test
public void testDefaultValues() {
  // Arrange
  builder = new RequestBuilder("/someResource");
  // Act
  String text = builder.getText();
  // Assert
  assertHasRegexp("GET /someResource HTTP/1.1\r\n", text);
}
```

Bedenken

- Das dauert alles zu lange (mehr Testcode als alles andere)
  - Teufelskreis: Mehr Stress, weniger Tests, mehr Fehler, mehr Stress, weniger Tests, usw.
- Ändere ich etwas am Code muss ich etliche Tests und Mocks wieder anpassen
- Der Kunde sieht das ich was mache wenn ich es lokal bei Ihm fixe, warum sollte er sonst einen Wartungsvertrag abschließen
- Guter Code verursacht komplexere Bugs zu Fixen

Anti pattern

"The main failing of TDD is the assumption that you know all of the tests that you will need before you know your software."

"I write the test after, then I reorder the commits once the test passes, and make sure it fails predictably in the way you were expecting it to fail."

"TDD is this crazy religion of writing tests for everything, including the lowest-level helper functions in a module that don't correspond to anything that would be tested in a regression test suite."

TDD und XP

- Pairing
- Work fresh
- Continuous Integration
- Continuous Deployment
- Simple design
- Refactoring

Fragen?

Demo