Testgetriebene Entwicklung (TDD) – vom Dogma zum Pragmatismus

von Martin Schneider

Die Revolution: Testgetriebene Entwicklung

„Once upon a time tests were seen as someone else's job (speaking from a programmer's perspective). Along came XP and said no, tests are everybody's job, continuously.

Then a cult of dogmatism sprang up around testing–if you can conceivably write a test you must.“

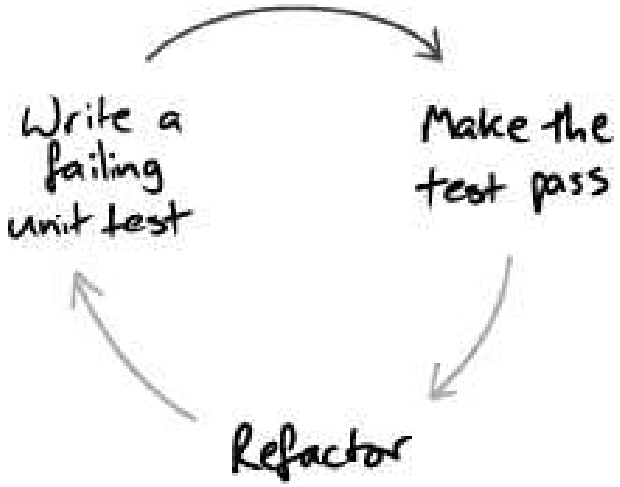
Kent Beck

Quelle: http://www.threeriversinstitute.org/blog/?p=187

Agenda

- Die Geschichte des TDD
  - ein kleiner (polemischer) Rückblick
- Umgang mit „Legacy“-Tests
  - Test-Smells
- Lessons Learned
  - Software-Design für TDD
  - Prozesselemente für TDD

Das Dogma: Test Driven Developement (TDD)



The Golden Rule of Test Driven Development

Never write new functionality without a failing test.

Kleine Geschichte des TDD

1997: Erstes JUnit (test.jar) von Kent Beck und Erich Gamma

1999: „eXtreme Programming Explained“ von Kent Beck

2000: Mock-Objekte (XP2000), EasyMock

2001: „Agile Software Development with Scrum” von Ken Schwaber

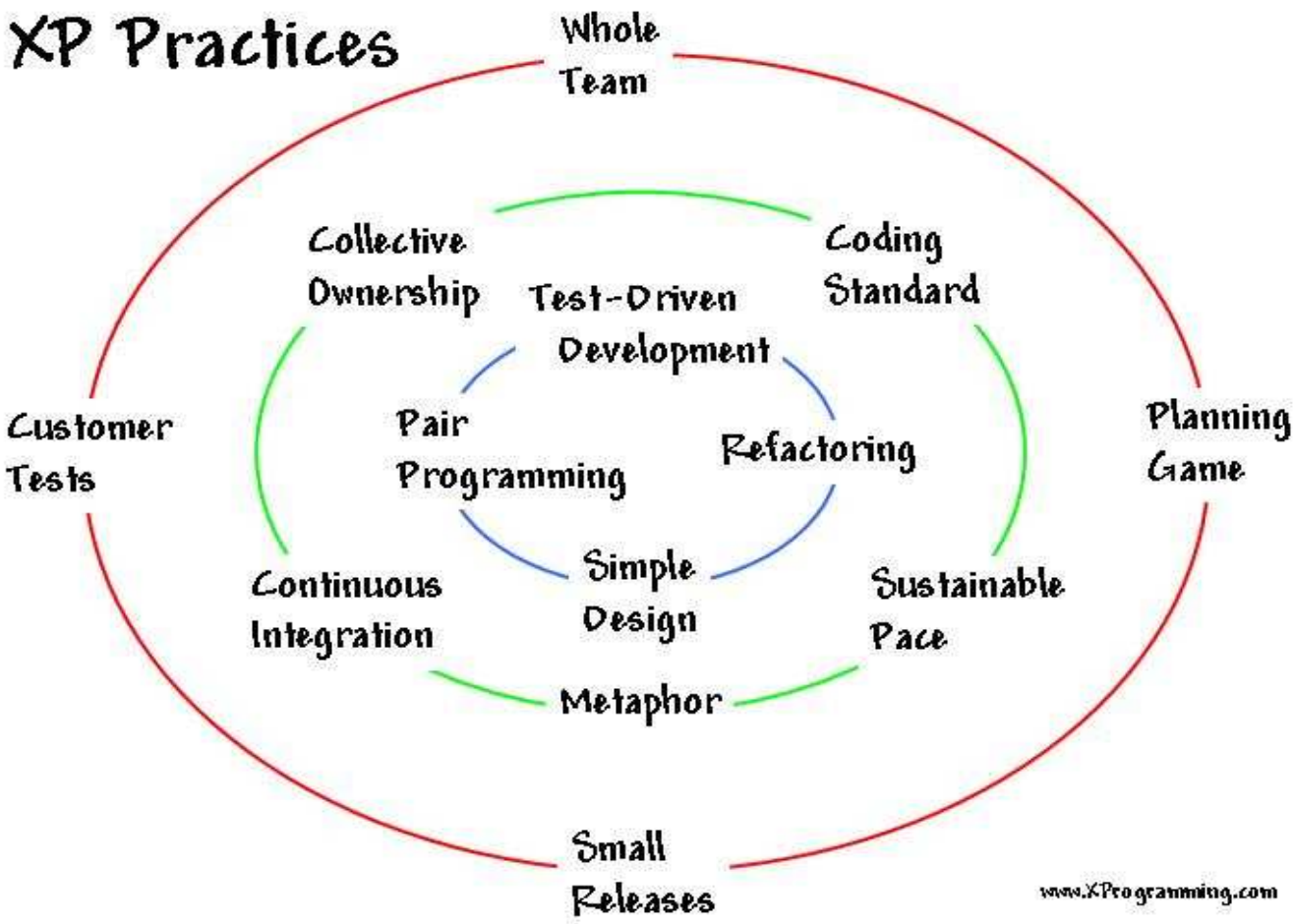
2004: jdk 5.0 „Tiger“, TestNG, Framework for Integrated Test (FIT)

2006: JUnit 4.0, „xUnit Test Patterns“

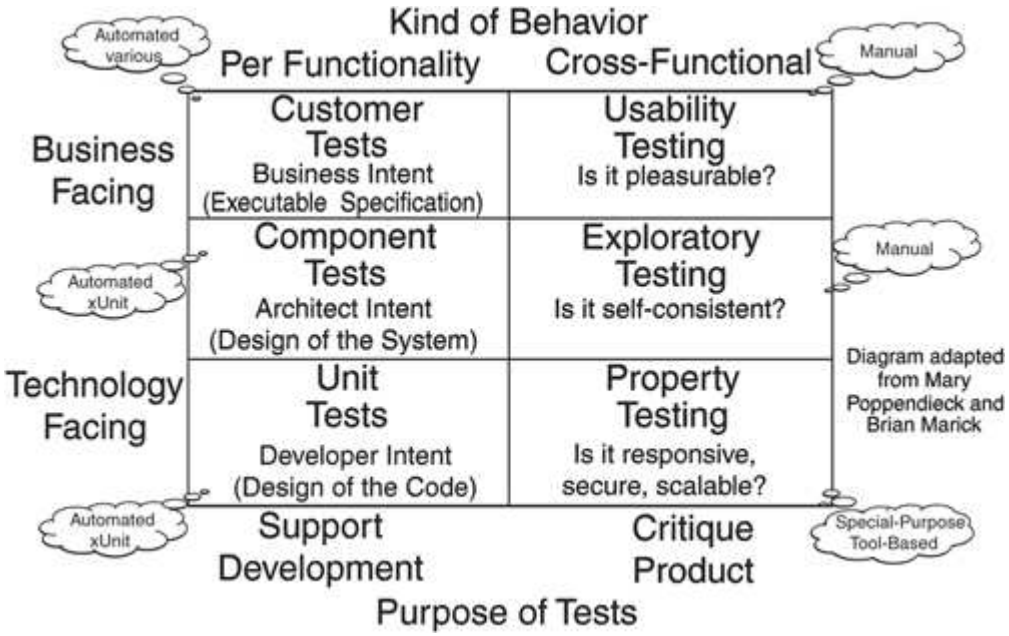
2009: Concordion

2010: Junit 4.8.2

Der Kontext: TDD und XP



Testarten


Kind of Behavior
Per Functionality
Cross-Functional
Automated various
Manual
Business Facing
Customer Tests
Business Intent
(Executable Specification)
Usability Testing
Is it pleasurable?
Automated xUnit
Component Tests
Architect Intent
(Design of the System)
Exploratory Testing
Is it self-consistent?
Manual
Technology Facing
Unit Tests
Developer Intent
(Design of the Code)
Property Testing
Is it responsive, secure, scalable?
Diagram adapted from Mary Poppendieck and Brian Marick
Automated xUnit
Special-Purpose Tool-Based
Support Development
Critique Product
Purpose of Tests

Was hat's gebracht?

- *Automatisiertes* Testen wurde plötzlich hipp
  - ca. 40% des Gesamtentwicklungsaufwandes für automatisches Testen verwenden ist o.k.!
- Sicherheitsnetz für den einzelnen Entwickler
  - Frühestmögliches Feedback des Systems
- Sicherheitsnetz für das gesamte Team
  - „Embrace Change“, d.h. Refaktorierbarkeit
  - Collective Code Ownership
- Zwang zu einfachem Design
  - Separation of Concerns
  - Lose gekoppelter Code

Was noch?

- Tests als Spezifikation und Dokumentation
  - Specification by Example
  - Code wird häufiger gelesen als geschrieben
- Ermöglicht Inkrementelles und Iteratives Vorgehen
  - Mikro-Iterarionen
  - Continuos Integration
- Patterns und Standards, die jeder versteht
  - XUnit-Frameworks

Verwerfungen

- Dogmatismus
  - Alles muss getestet werden, um jeden Preis
  - Test-First vs. Test-Last-Diskussion
- Vernachlässigung nichtfunktionaler Aspekte
- Monströse Aufwände für eigene Testframeworks
  - „Not invented here“-Syndrom
- Vielfalt von Testframeworks
  - Vitalität und Stabilität nicht überschaubar
- „Mock“-Wahnsinn
  - Mock-Db anstatt hsqldb
- Ravioli-Code

Ergebnis: Legacy-Tests

- Komplexe Tests
  - Schwer wartbar, schwer zu verstehen
  - Aufwändig und teuer zu pflegen
  - Testausführung wesentlich zu lang
  - Viele Seiteneffekte bei Änderungen der Fachklichkeit
- Kontextabhängige Tests
  - Datenabhängigkeiten
  - Abhängigkeiten von externen Systemen
- Degenerierte Tests
  - X-Pattern oder @Ignore

Die Idee: Analysieren und Operationalisieren

- Definition von Zielen
- Definition von Bewertungskriterien für existierende Tests
- Durchführung von Bestandsaufnahmen
  - Code-Reviews und Messungen
  - Bestandsaufnahme anhand von „Interviews“
- Definition eines team-einheitlichen Vorgehens und Vokabulars

Ziele des Testens

- Tests erhöhen die Qualität der Software
- Tests helfen das System zu verstehen
- Tests minimieren Risiken
- Tests müssen einfach ausführbar sein
- Tests müssen einfach zu erstellen und warten sein
- Tests dürfen die Weiterentwicklung des Systems nicht verteuern

Testspezifische Prozessaspekte

- Anforderungsmanagement/Änderungsmanagement:
  - Wird bei Anforderungsdefinition auf Testbarkeit Wert gelegt?
- Risikomanagement:
  - Werden risikobehaftete Anforderungen identifiziert und gesondert behandelt?
- Testprozess/Entwicklungsprozess:
  - Wer schreibt wann Tests?
  - Gibt es verbindliche Qualitätsziele oder zu erreichende Kennzahlen wie Testabdeckung pro Code, pro Funktionalität/Use-Case ?

Messbare Kriterien

- Testabdeckung:
  - Ist 50% Soll 80%
- Ausführungsdauer der Tests:
  - Ist 20h Soll 2 Stunden
- Verhältnis von Testcode zu Produktivcode:
  - Ist 1:3 Soll 1:1
- Verhältnis von Unit-Tests zu Integrationstests
  - Ist 1:10 Soll 5:1

Test-Smells

- Analog der Code-Smells a la Martin Fowler
- Ein „Smell“ ist ein Symptom eines Problems
- Code-Smells
  - Statisch im Test-Code versteckt
- Behaviour-Smell
  - Werden zur Laufzeit bei Testausführung sichtbar
- Project-Smells
  - Betreffen Entwicklungsprozess

Code-Smell: Obscure Test

- Eager Test
  - Zu viel auf einmal
  - Lösung: Independent Single-Condition Tests
- Mistery Guest
  - externe komplexe Fixture, z.B in Datei, DB ..
  - Schlechte Lesbarkeit
- General Fixture
  - Für Test irrelevante Information
  - Schlechte Lesbarkeit

Beispiel: Eager Test

```
public void testFlightMileage_asKm2() throws Exception {
    // set up fixture
    // exercise constructor
    Flight newFlight = new Flight(validFlightNumber);
    // verify constructed object
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    // set up mileage
    newFlight.setMileage(1122);
    // exercise mileage translator
    int actualKilometres = newFlight.getMileageAsKm();
    // verify results
    int expectedKilometres = 1810;
    assertEquals( expectedKilometres, actualKilometres);
    // now try it with a canceled flight
    newFlight.cancel();
    try {
       newFlight.getMileageAsKm();
       fail("Expected exception");
    } catch (InvalidRequestException e) {
       assertEquals( "Cannot get cancelled flight mileage",
                    e.getMessage());
    }
 }
```

Beispiel: Mistery Guest

```
public void testGetFlightsByFromAirport_OneOutboundFlight_mg()
                throws Exception {
    loadAirportsAndFlightsFromFile("test-flights.csv");
    // Exercise System
    List flightsAtOrigin =
          facade.getFlightsByOriginAirportCode( "YYC");
    // Verify Outcome
    assertEquals( 1, flightsAtOrigin.size());
    FlightDto firstFlight = (FlightDto) flightsAtOrigin.get(0);
    assertEquals( "Calgary", firstFlight.getOriginCity());
 }
```

Code-Smell: Conditional Test Logic

```
...
//verify Vancouver is in the list
actual = null;
i = flightsFromCalgary.iterator();
while (i.hasNext()) {
   FlightDto flightDto = (FlightDto) i.next();
   if (flightDto.getFlightNumber().equals(
      expectedCalgaryToVan.getFlightNumber())){
          actual = flightDto;
          assertEquals("Flight from Calgary to
Vancouver",expectedCalgaryToVan, flightDto);
          break;
      }
   }
}
...
```

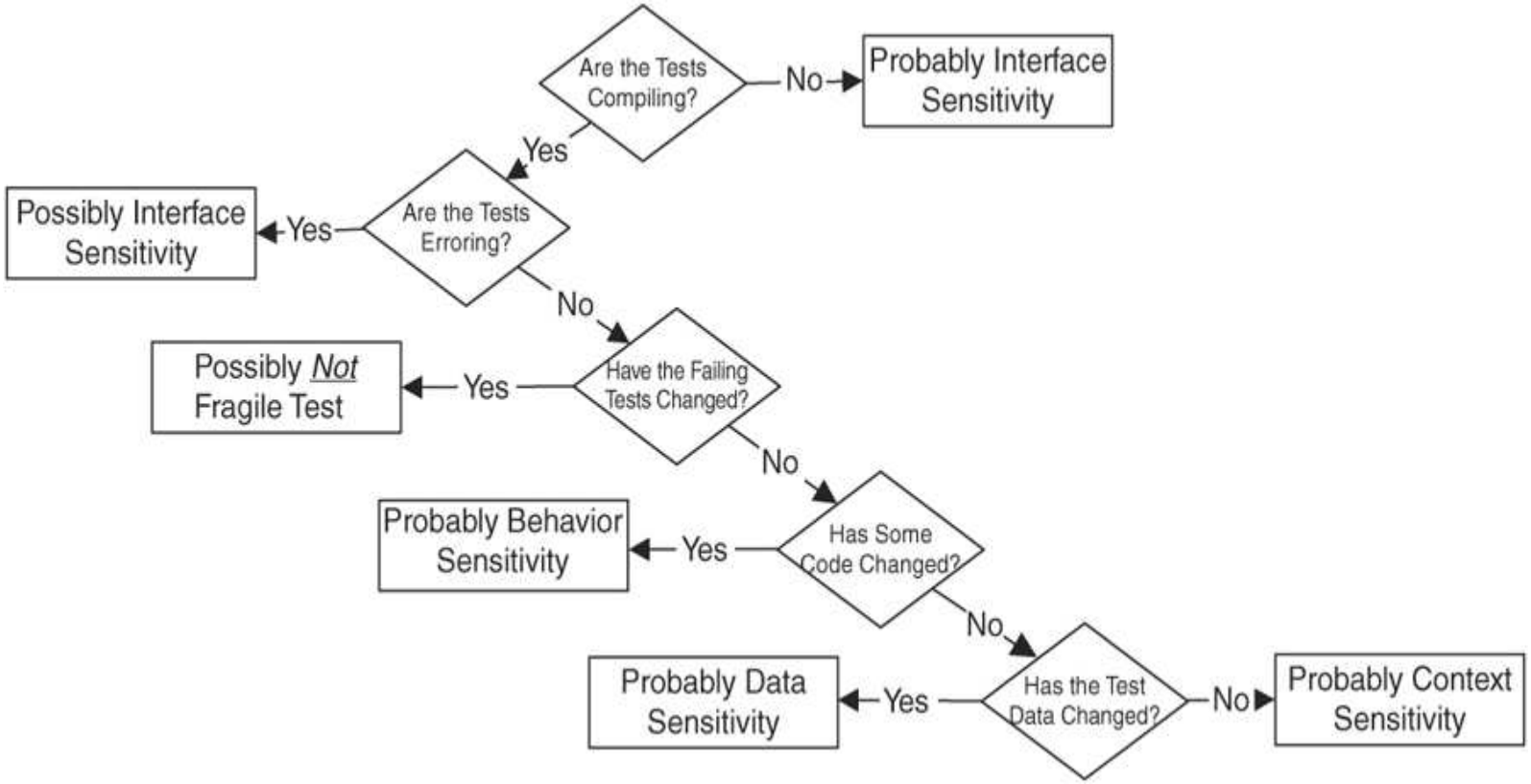
Weitere Code-Smells

- Test Code Duplication
  - Cut-and-Paste Code Reuse
- Hard to test code
  - Highly Coupled Code
  - Asynchronous Code
- Complex Test-Fixture Management
- Test Logic in Production
  - Ariane 5

Behaviour-Smells

- Slow Tests
- Assertion Roulette
  - Bei großen Testsuiten: Welcher Test schlägt fehl?
  - Nachstellbarkeit (Integration-Rechner vs. lokale IDE)
- Erratic Test
  - „Heisenbug“
- Fragile Test
- Frequent Debugging
- Manual Intervention

Fragile Test



Project-Smells

- Buggy Tests
  - Komplexe eigene Testframeworks
  - Zuviel weggemockt
- Developer not writing Tests
  - Zu wenig Zeit
  - Zu wenig Erfahrung
  - Hard to test code
- High Test Maintenance Cost
- Production Bugs
  - Bugs in Produktion trotz Tests

Ergebnisse

Smell	Anzahl Nennungen
Obscure Test	5
Hard to Test Code	4
Test Code Duplication	4
Complex Test-Fixture Management	7
Developer not writing Tests	7
High Test Maintenance Cost	7
Erratic Test	2
Fragile Test	3
Frequent Debugging	1
Manual Intervention	1

Beispielhafte konkrete Maßnahmen

- Slow-Test: Tests gegen hsqldb ausführen
- Obscure-Test: dbUnit, EasyMock, „richtige Unit-Tests“
- Hard-To-Test-Code: Refactorings, EasyMock, „richtige Unit-Tests“
- High-Maintenance-Cost: dbUnit
- Complex-Fixture-Mngmt: dbUnit
- Developer not writing Tests: 4-Augen-Prinzip

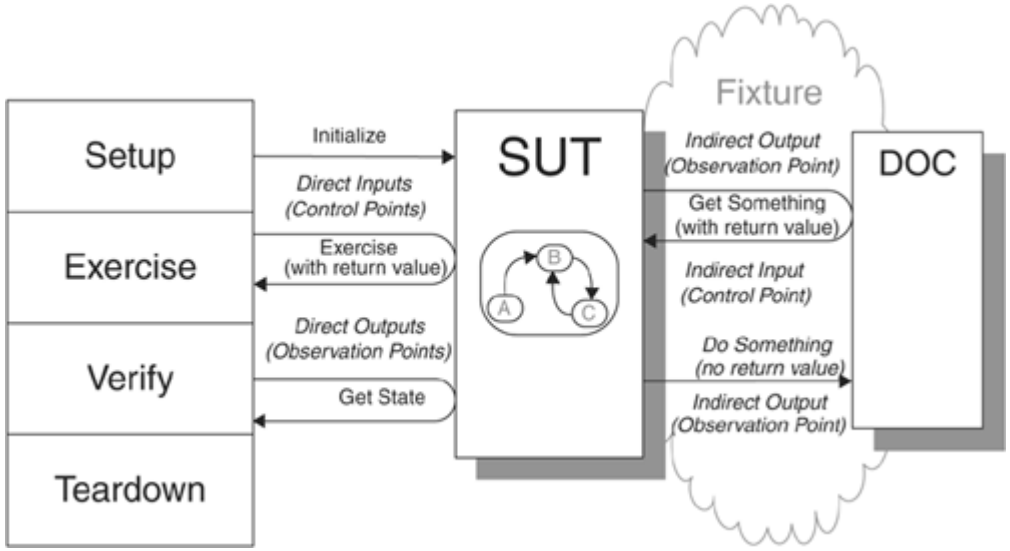
Definition von projektrelevanten Testarten

- Entwickler-Tests
  - „Eigentliche“ Unit-Tests
  - Isoliertes Testen atomarer Programmeinheiten
- Integrationstests
  - Testen das Zusammenspiel der Systemkomponenten
- Akzeptanztests
  - Testen auf die Erfüllung der Anwenderanforderungen
  - Idealerweise durch Anwender definiert und durch diese ausführbar

Ergebnisse

- Maßnahmen im Team akzeptiert
- Bewusstsein für Potential testgetriebener Entwicklung hergestellt
- Konkrete Maßnahmen und Ergebnisse
  - Tests gegen hsqldb anstatt Oracle Tests Faktor 7 – 10 schneller
  - dbunit eingeführt
  - EasyMock etabliert
  - Refactoring von Integrationstests hin zu Unit-Tests, erzielte Beschleunigung Faktor 1000!
  - Fit/Fitnesse prototypisch aufgesetzt

Vokabular: Der Testzyklus



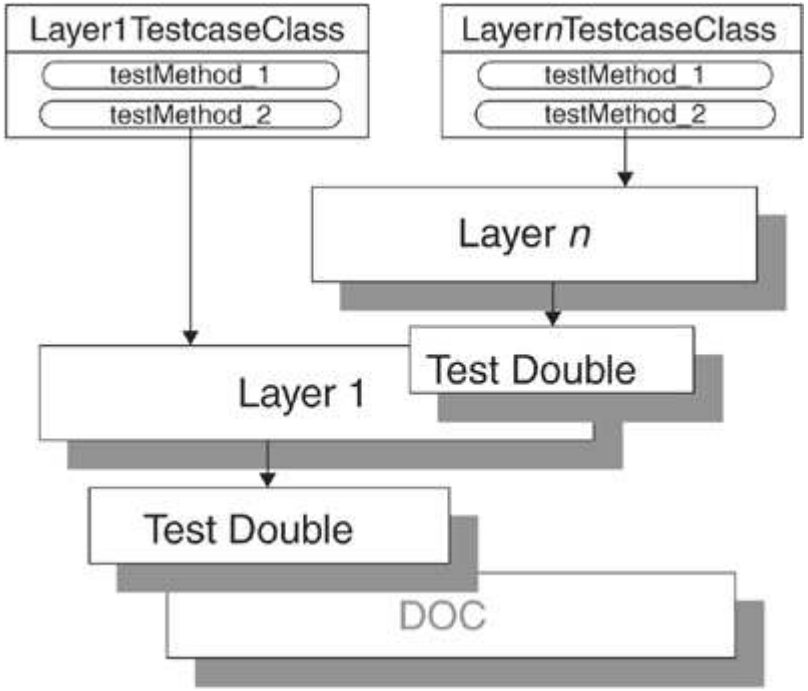
Kleine Test-Philosophie

- „Don’t modify the SUT“
- Keine Testlogik in Produktionscode
- Halte Tests unabhängig voneinander
- Minimiere Testüberlappung
  - Code-Coverage
- Minimiere Untestbaren Code
- Verifiziere ein Szenario pro Test-Methode

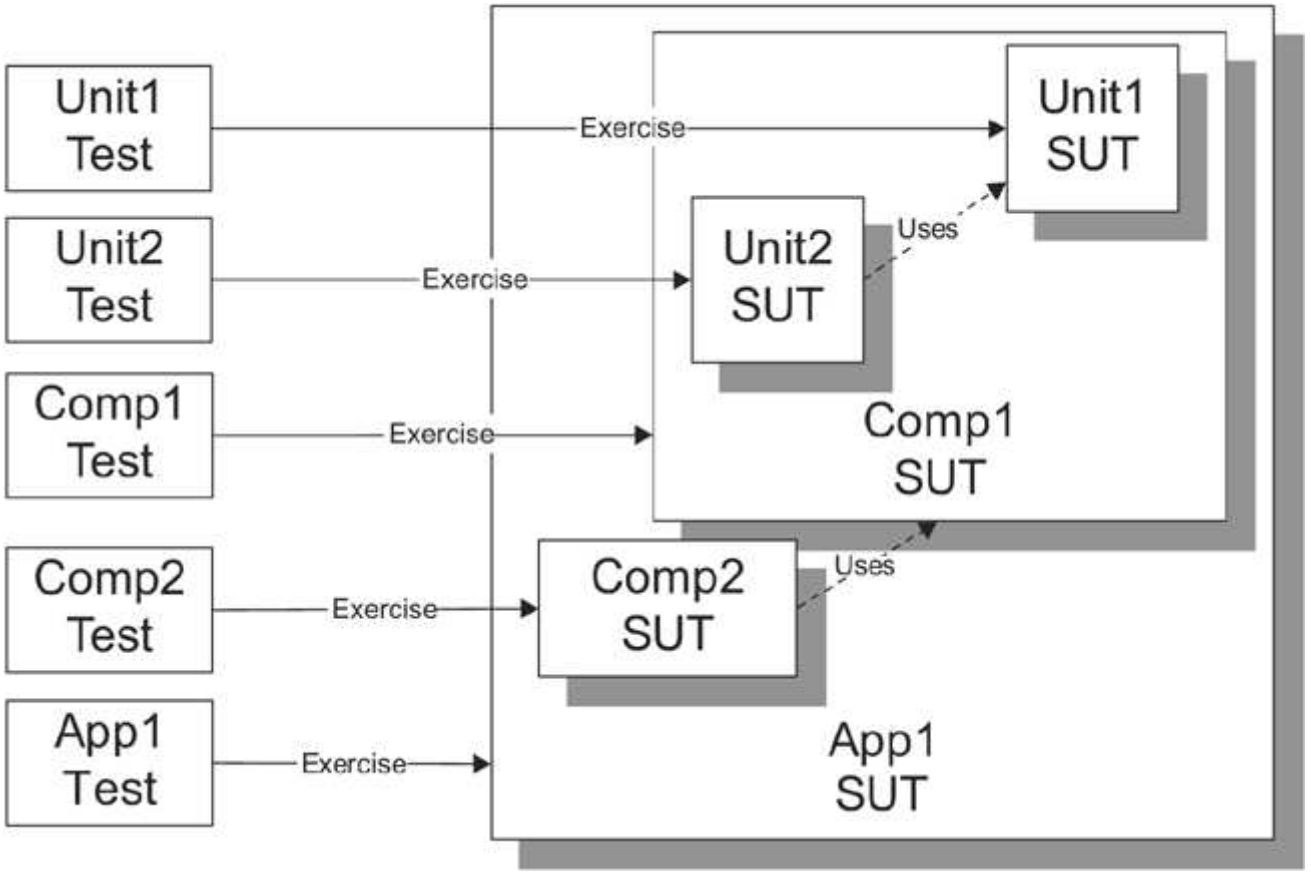
Kleine Test-Philosophie

- Testgetrieben, aber nicht um jeden Preis Test-First
- Tests als Beispiele verstehen, Fokus auf Lesbarkeit
- Einen Test komplettieren
  - Ggf. Skeletons für weitere anlegen
- Boundary-Conditions vs. Main-Success-Scenario
- Fokus auf State Verification Behavior Verification mit Vorsicht und Easyock
- Fixture Design auf Test-by-test Basis
  - Test-Daten über Faktories explizit im Code, nicht xml

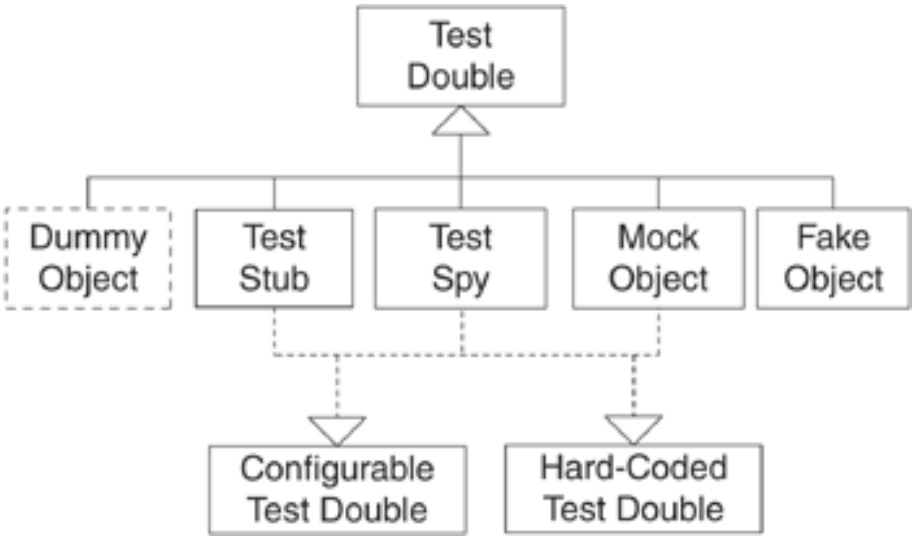
Testbarkeit durch Design



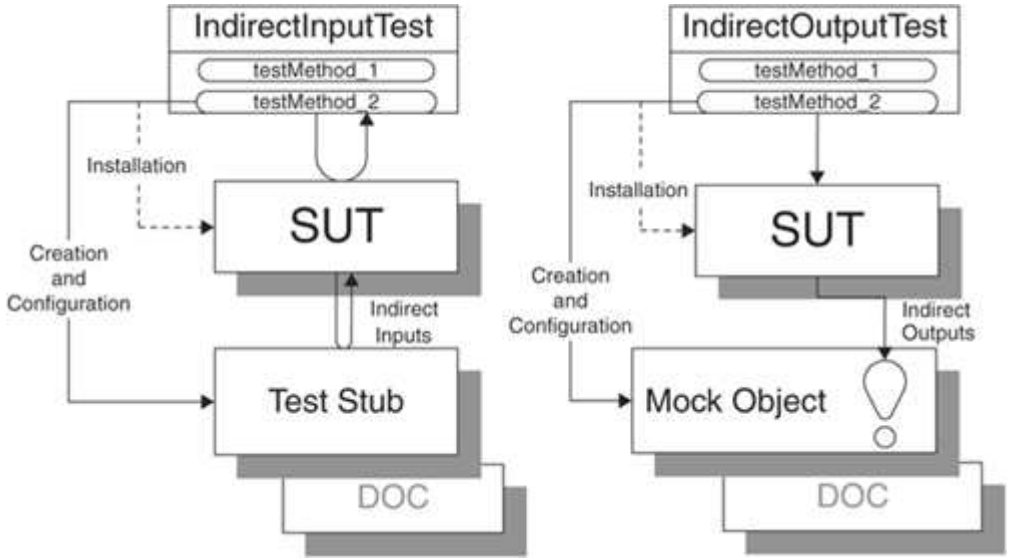
Testbarkeit durch Design



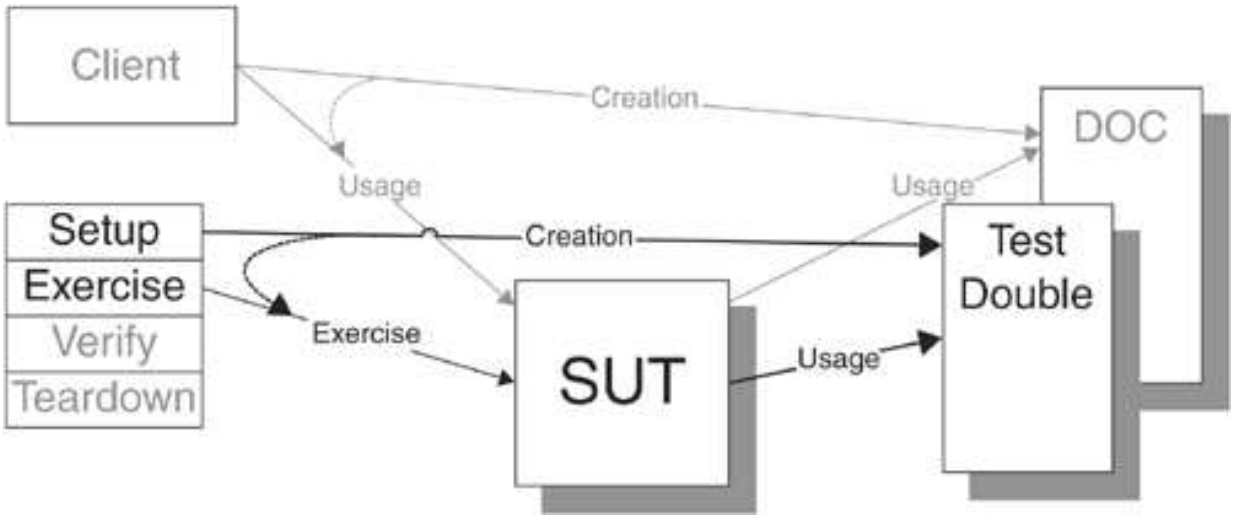
Kleine Testattrappenkunde



Teste Indirect Input und Output



Das Pattern: Dependency Injection



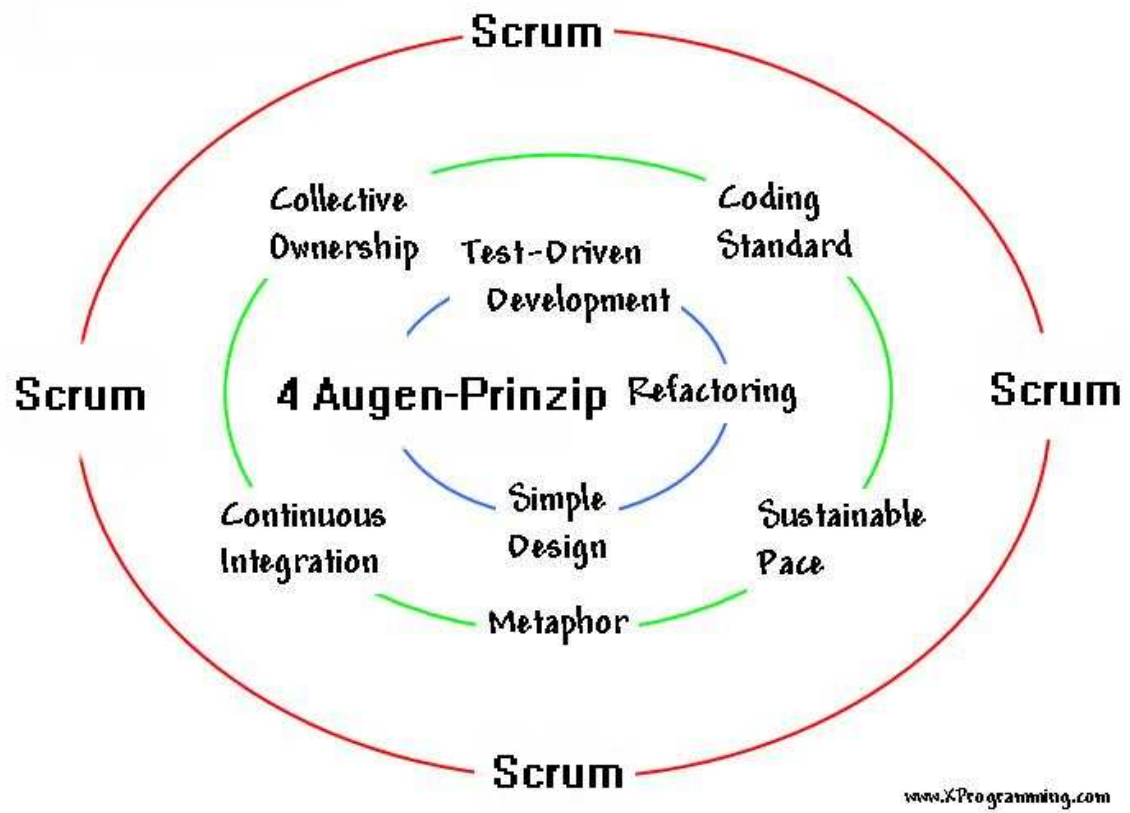
Pragmatischer TDD-Prozess

- Testgetrieben
  - Nicht zwangsläufig test-first!
  - Pragmatische Testabdeckung
- 4-Augen-Prinzip
  - Peer-Reviews
  - Bei schwierigen Stellen Pair-Programming
  - Collective Code Ownership
- Continuous Integration
- Build-Prozess mit Staging, definierten Releases
- Metriken, z.B. mit Sonar (http://www.sonarsource.org/)

Pragmatisch: Balanced Agility

- Scrum
- Continuos Integration (configuration management, check -in, check-outs, intra day integration, etc.)
- Testing (unit, regression, integration, system, acceptance)
- Release management (labeled releases, release notes, migrations, platforms, sandboxes, deployment, etc.)
- 4 Augen Prinzip

Balanced Agility




Scrum&XP

Quellen

http://xunitpatterns.com/Test%20Smells.html