Testgetriebene Entwicklung

Arbeitskreis Objekttechnologie Norddeutschland
Hamburg, 18.03.2002

Frank Westphal westphal@acm.org
freier Berater, Hamburg

Tammo Freese tammo.freese@offis.de
OFFIS, Oldenburg

Effektives Testen

- möglichst zeitnah zur Programmierung
- automatisiert wiederholbar
- muss Spaß machen
- so oft wie Kompilieren durchgeführt
- so einfach wie Kompilieren
- soll Fehler finden,
  nicht Fehlerfreiheit beweisen
- muss mehr bringen als kosten

Motivation

- Tests sichern uns den Erhalt der vorhandenen Funktionalität bei Erweiterung und Überarbeitung.
- Refactoring verlängert die produktive Lebensdauer einer Software.
- Code lässt sich im Nachhinein oft nur schlecht testen.

Drei Direktiven

- Jede Änderung am Verhalten des Programms wird zuvor durch einen fehlschlagenden automatisierten Test motiviert.
- Der Code wird immer in die einfachste Form gebracht.
- Zur erfolgreichen Integration müssen alle gesammelten Tests laufen.

Test/Code Zyklus (1)

0. Wir überlegen uns erste Testfälle und notieren sie.

1.-3. Wir implementieren die Tests iterativ und inkrementell.

4. Wir sind fertig, wenn uns keine weiteren Tests einfallen, die unter Umständen fehlschlagen könnten.

Test/Code Zyklus (2)

1. Wir entwerfen einen Test, der zunächst fehlschlagen sollte.
2. Wir schreiben gerade soviel Code, dass der Test tatsächlich fehlschlägt.
3. Wir schreiben gerade soviel Code, dass alle Tests durchlaufen.

1. Wir entwerfen einen Test, der zunächst fehlschlagen sollte.

- Programming by Intention: Wir verwenden die zu testende Funktionalität einfach so, als ob sie schon realisiert wäre.
- Mit jedem Test wollen wir einen möglichst kleinen Schritt gehen.

2. Wir schreiben gerade soviel Code, dass der Test tatsächlich fehlschlägt.

- Den Test zunächst fehlschlagen zu sehen testet die Relevanz des Tests.

2a. Der Test läuft, obwohl er eigentlich fehlschlagen sollte.

- indirekte oder duplizierte Tests?
- falsche Tests?
- zuvor zu viel Code geschrieben?

3. Wir schreiben gerade soviel Code, dass alle Tests durchlaufen.

- Wir schreiben nicht mehr Code als die Tests fordern, weil dieser unspezifiziert und ungesichert wäre.
- Erst durch weitere Tests beweisen wir, dass die Lösung noch zu einfach ist.

3a. Tests schlagen fehl, obwohl sie eigentlich laufen sollten.

- Schlägt der neue Test fehl, erfüllt unser gerade geschriebener Code noch nicht die neue Anforderung.
- Schlagen andere Tests fehl, verletzt unser gerade geschriebener Code Teile der vorhandenen Spezifikation.

3b. Die Implementierung des Tests erfordert neue Methoden und damit weitere Tests.

- Das Inkrement ist zu groß gewählt.
- Uns fehlen Methoden und Objekte zur intuitiven Implementierung.
- Top-Down Abstieg und Bottom-Up Implementierung sind oft vermeidbar.

Refactoring

- Eine Änderung an der internen Struktur eines Programms, ohne sein beobachtbares Verhalten zu ändern,
  - um das Design fortlaufend zu verbessern.
  - um das Programm leichter verständlich zu machen.
  - um zukünftige Änderungen am Code zu erleichtern.
  - um der einsetzenden Entropie entgegen zu wirken.

Einfaches Design

- Ziel des Refactoring
- Ein Design ist einfach, wenn der Code
  1. alle seine Tests erfüllt.
  2. die Intention der Programmierer ausdrückt.
  3. keine duplizierte Logik enthält.
  4. möglichst wenig Klassen und Methoden umfasst.

JUnit

- Java-Framework zum Schreiben und Ausführen automatischer Unit Tests
- Tests werden in Java codiert.
- Entsprechende Frameworks sind für andere Programmiersprachen erhältlich.

Beispiel: TestCase

```
import junit.framework.*;

public class MoneyTest extends TestCase {
  public MoneyTest(String name) {
    super(name);
  }
  public void testAmount() {
    Money money = new Money(2.00);
    assertTrue(money.getAmount() == 2.00);
  }
}
```

Anatomie eines Testfalls

- Unterklasse von TestCase
- Testfallmethoden public void test...()
- Verwendung der geerbten Methoden assert...() und fail()
- Instanzvariablen für Testobjekte
- setUp() zum Aufbau von Testobjekten und Testressourcen
- tearDown() zur Ressourcenfreigabe

Programmierzüge

- **grün-rot:** Wir entwerfen einen Test, der zunächst fehlschlagen sollte. Wir schreiben gerade soviel Code, dass er tatsächlich fehlschlägt.
- **rot-grün:** Wir schreiben gerade soviel Code, dass alle Tests durchlaufen.
- **grün-grün:** Wir refaktorisieren den Code in die einfachste Form.

Tests als ausführbare Spezifikation

- Anforderungen lassen sich konkret und unmissverständlich in einem Test spezifizieren und später beliebig häufig automatisiert verifizieren.
- Anforderungen werden systematisch und zeitnah zur Programmierung analysiert und Analysefehler dadurch schnell entdeckt.
- Durch das feingranulare Testen treten meist interessante Grenzfälle hervor.

Tests als synchrone Dokumentation

- Unsere Testsuite liefert konkrete Beispiele für die Verwendung von Klassen und Methoden.
- Häufiges Testen stellt sicher, dass die Tests aktuell zur Codebasis bleiben.

Tests zur Schnittstellendefinition

- Wenn wir zuerst den Test schreiben, verwenden wir Klassen und Methoden, bevor wir sie implementieren.
- Die Klassenschnittstelle lehrt uns mit ihrer frühen Benutzung im Test, wie sie wirklich entworfen werden will.

Tests zur Modularisierung

- Unit Tests erfordern unabhängig voneinander testbare Einheiten.
- Nicht testgetrieben entwickelter Code enthält oft so starke Abhängigkeiten, dass seine Module praktisch nicht in Isolation getestet werden können.
- Testgetriebene Entwicklung führt zur Entkopplung der Programmteile.
- Der Code wird insgesamt einfacher.

Isoliertes Testen

- Probleme:
  - Programmeinheiten arbeiten nicht isoliert.
  - Aufbau der Testumgebung ist oft aufwändig.
  - Testen von Ausnahmesituationen ist schwierig.
  - Tests werden bei Überschreitung der Systemgrenzen langsam.
- Lösung: Mock-Objekte
  - Stubs mit eingebetteter Testfunktionalität
  - ersetzen von der zu testenden Unit verwendete Klassen während des Tests

Ökonomie des Testens

- Testen verlangsamt uns auf kurze Sicht.
- Testen macht uns mittelfristig agil und überlebensfähig, weil es verhindert, dass die Weiterentwicklung stagniert.

Bücher

- Martin Fowler: Refactoring
- Johannes Link: Unit Tests mit Java

Links

- www.junit.org
- www.frankwestphal.de
- www.mockobjects.com
- www.easymock.org