Testen

Matthias Kempka

Innoopract

March 28, 2008

Motivation

- Warum schreiben wir Software?

Motivation

- Warum schreiben wir Software?
  - Geld verdienen
  - Selbstbestätigung
  - Ruf erarbeiten
- Software erstellen heisst Werte (asset) schaffen
- Werte wollen erhalten werden

Konsequenz

Software wird weiterentwickelt

- Allerdings nicht notwendigerweise vom ursprünglichen Autor
  $\longrightarrow$ Code muss **kommunizieren**
- Und selten so wie es ursprünglich geplant war
  $\longrightarrow$ Code muss **flexibel** gehalten werden
- Am besten gelingt das, wenn der Code keine Extravaganzen entält
  $\longrightarrow$ Code muss **einfach** bleiben

Die drei Werte (values)

- **Kommunikation** (communication)
  - Der grösste Teil der Kosten von Software steckt in der Wartung
- **Einfachheit** (simplicity)
  - Unnötiger Code erschwert die Lesbarkeit
  - Aber die Komplexität des Problems bleibt bestehen
- **Flexibilität** (flexibility)
  - Software wird weiterentwickelt
  - Selten dort, wo man es ursprünglich erwartete

Prinzipien (principles)

- Werte sagen, was wir erreichen wollen, nicht wie wir es erreichen können.
- Prinzipien bleiben abstrakt, sind aber näher am Problem
  - z.B. Programmieren: Lokale Konsequenzen, Wiederholung vermeiden, Daten und Logik zusammen, ...
- Helfen beim Diskutieren über anzuwendende Muster
- Helfen beim Entwickeln neuer Muster in neuen Umgebungen, z.B. neue Sprache

Muster (patterns)

- Weitgehend konkrete Lösung für eine konkrete Problemklasse
- Erlauben schnelle Lösungsfindung für wiederkehrende Probleme
  - Entwurfsmuster (design patterns)
  - Implementierungsmuster (implementation patterns)
  - Analysemuster (analysis patterns)
  - Testmuster (test patterns)

Referenzen

- Extreme Programming explained (Kent Beck), *2005*
- Implementation patterns (Kent Beck), *Ende 2007*

JUnit 4

- Weiterentwicklung von JUnit 3
- Nutzt Annotations zum Definieren von Tests und Testeigenschaften $\longrightarrow$ Java 1.5 erforderlich
- Abwärtskompatibel: JUnit3-Tests und JUnit4-Tests können in einer Testsuite ausgeführt werden
- Features
  - Testmethoden und Testsuites
  - Fixtures in Klassen- und Methodengranularität
  - Erwartete Exceptions
  - Selber Test auf einer Vielzahl von Daten mit Parametern

Testdefinition

- Kein Ableiten von `TestCase` (Wird als JUnit 3-Test interpretiert)
- Test-Methoden sind mit `@Test` gekennzeichnet
- `assert*`-Methoden befinden sich in Klasse `Assert`
  - statischer Import oder
  - qualifizierter Zugriff oder
  - Ableiten von `Assert`

```
import static org.junit.Assert.*;
public class MeineTestKlasse {
  @Test
  public void name() throws Exception {
    ...
    assertEquals( ..., ... );
  }
}
```

Tests zusammenfassen in einer Testsuite

- JUnit3-TestSuites: Viel sich wiederholender Text
- JUnit4-TestSuites: Aufzählung von Klassen
  - Durch Verwendung von Annotations ist die eigentliche Klassendefinition leer

```
@RunWith(Suite.class)
@Suite.SuiteClasses( {
    MyTest.class,
    MyTest2.class
})
public class AllTests {
}
```

Fixtures

- Fixture ist die Umgebung für einen Testcase
- Aufsetzen und Abbauen des Fixtures innerhalb von `setup`- und `teardown`-Methoden
  - Vor und nach jeder Testmethode: `@Before` und `@After`
  - Vor und nach allen Testmethoden der Klasse: `@BeforeClass` und `@AfterClass`

```
@BeforeClass public static void setUpClass() { ...  }
@AfterClass public static void tearDownClass() { ... }

@Before public void setUp() { ... }
@After public void tearDown() { ... }
```

Erwartete Exceptions

- Parameter zur `@Test`-Annotation erklärt eine Exception zum erwarteten Ergebnis
- Test schlägt fehl, wenn die Exception nicht geworfen wird.

```
@Test(expected=java.lang.NullPointerException.class)
public void myTestMethod() {
    ...
}
```

- Analog: Parameter `timeout` lässt Test max. gegebene ms Zeit

Parameterized

- Test arbeitet mit einem Datum
- Selber Test ist interessiert an anderem Datum mit anderem Ergebnis
- → Verwendung des Testrunners `Parameterized`
  - Testklasse wird annotiert mit `@RunWith(value=Parameterized.class)`
  - statische Methode `public static Collection<Object[][]> data()` gibt eine Liste mit Werten zurück
  - Konstruktor nimmt jedes Wertepaar an und initialisiert Felder

Beispiel für Parameterized

```
@RunWith(value=Parameterized.class)
public class FactorialTest {
    private long expected;
    private int value;

    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] {
                              { 1, 0 },   // expected, value
                              { 24, 4 },
                              { 5040, 7 },
                              });
    }

    public FactorialTest(long expected, int value) {
        this.expected = expected;
        this.value = value;
    }

    @Test
    public void factorial() {
        Calculator calculator = new Calculator();
        assertEquals(expected, calculator.factorial(value));
    }
}
```

Demo

Demonstration zu `assertSame` vs. `assertEquals`, String Konstanten und Autoboxing.

Die Erkenntnisstufen eines Programmierers

1. Coden & Debuggen erzeugt lauffähige Programme
   - Diese Einstellung erzeugt auch Regressionen
2. Tests zeigen, dass ein Programm funktioniert
   - Wie viele Tests benötigt man dafür?
3. Tests zeigen, dass ein Programm nicht funktioniert
   - Man findet immer einen Test, der fehlschlägt
     $\longrightarrow$ never shipping again!
4. Tests schaffen Vertrauen in die Qualität von Software

Definitionen

Definition

Ein *Test Case* ist eine Zusammenstellung von Testmethoden, die ein gemeinsames Kriterium testen.

Definition

Eine *Test Suite* ist eine Auflistung von Test Cases, die zusammen ausgeführt werden.

Definitionen

Definition

Ein *Test Runner* verarbeitet eine Testbeschreibung, stellt eine Testumgebung her und führt eine Testmethode, einen Test Case oder eine Test Suite aus.

- Eine Testbeschreibung kann deklarativ oder programmatisch (z.B. nach Benutzerinteraktion) erfolgen.
- Eine Testumgebung simuliert die Laufzeitumgebung, die der auszuführende Code benötigt.

Beispiele:

- Ausführen einer Test Suite im automatischen Build System
- Suche aller Tests unterhalb eines Paketes, die anschliessend zusammen ausgeführt werden

Definitionen

Definition

Ein *Test Harness* ist der Testcode der und die Testumgebung, die zur Ausführung der Tests für den Produktivcode notwendig ist.

Er dient als Schutz des Codes vor

- ungewollten Seiteneffekten einer Änderung
- Regressionen

Definitionen

Definition

Ein *Whitebox Test* ist ein Test, der in der Art des Testens die Implementierung berücksichtigt

Beispiele:

- gezieltes Testen von Randbedingungen im Code
- gezielter Test für einen Fehler im Code

Definitionen

Definition

Ein *Blackbox Test* ist ein Test, der die Spezifikation ohne Rücksicht auf die Implementierung tested

Beispiele

- Akzeptanztest bei Abnahme des Produkts
- dokumentierender Unit-Test

Was ist ein Test?

- Der Begriff **Test** ist vielseitig verwendbar
  - Integrationstests
  - Performancetests
  - Akzeptanztests
  - Unit-Tests
  - ...
- Welche davon sind (mit realistischem Aufwand) automatisierbar?

Unit-Tests

Anforderungen an Unit-Tests

- Unabhängig
- Wiederholbar
- Schnell ($<<$0.1s)

Ein Test ist kein Unit-Test wenn:

- Datenbankzugriff erfolgt
- Dateisystemzugriff erfolgt
- Netzwerkzugriff erfolgt
- Spezielle Konfigurationen durch den Benutzers nötig sind (Editieren von Konfigurationsdateien etc.)

Das bedeutet nicht, dass Unit-Test-Frameworks nicht als Testtreiber für solche Tests verwendet werden dürfen!

Wohin gehört Testcode?

Problem: Das fertige Produkt soll keine Abhängigkeit zum Testframework enthalten

- Testcode enthält Abhängigkeit zum Testframework und zum Produktivcode
- $\longrightarrow$ Separate Orte für Test- und Produktivcode
  - ($\longrightarrow$ Separater Build für automatisierte Test)
  - In Eclipse-Projekten:
    - separate Sourcefolder
    - separate Projekte
  - Testklasse ist im selben Paket wie getestete Klasse (erlaubt Nutzung von default-Visibility z.B. bei Dependency Injection)

Wie organisiert man TestSuites

Problem: Tests sollen schnell laufen, damit sie oft ausgeführt werden, aber es gibt auch Testfälle, die ihre Zeit dauern.

- Namenskonvention für verschiedene Arten von Testcases
  - Aber nicht mehr als 2 oder 3 verschiedene Arten zulassen
- Suites, die alle Tests einer Testart zusammenfassen
- Eine Suite, die alle Suiten zusammenfasst zum Ausführen aller Tests

Entkoppeln

Problem: Getestete Klassen benötigen andere Klassen

- *Dependency Injection*: Übergeben der Abhängigkeiten im Konstruktor
- Bestes Vorgehen: Verwendung bereits existierender Klassen im Sourcecode
- Alternativ: Benutzung von *Fake* oder *Mock* Objekten

Fake Objekte

- Auch *stubs* genannt
- Einfache Implementierung eines Typs für Testzwecke
- So wenig Logik wie möglich

Mock Objekte

- Einfache Implementierung eines Typs für Testzwecke
- Enthält Testlogik
  - Oft: Setzen von erwarteten Werten vor Begin des Tests
  - Charakteristisch: Test ruft `myMock.validate()` auf statt selbst Assertions auszuführen
  - `Mock.validate()` enthält dann die Assertions.

Mock Objekt

```
class ApplicationTest extends TestCase {

  MockView mockView = new MockView();

  public void testApplication() {
    Application a = new Application() {
      protected View createView() {
        return mockView;
      }
    };
    a.run();
    mockView.validate();
  }

  private class MockView extends View
  {
    boolean isDisplayed = false;

    public void display() {
      isDisplayed = true;
    }

    public void validate() {
      assertTrue(isDisplayed);
    }
  }
}
```

Entkoppeln (2)

Problem: Getestete Klassen greifen auf Singletons zu

- Wieder: Dependency Injection
- Singleton ist Implementierung eines Interfaces
- Getestete Klasse erhält Instanz des Interfaces im Konstruktor
- Dann wieder: Benutzung von *Fake* oder *Mock* Objekten

UI Tests (I)

Problem: UI Code wehrt sich gegen Tests

- Layout macht Schwierigkeiten
  - Mit Unit-Tests nicht testbar
- Bis dicht unter die Oberfläche ist auch UI Code einfach nur Code
  - Gutes Design erlaubt dort auch Unit-Tests
- Spezielle Werkzeuge ermöglichen UI Tests.

UI Tests (II)

Funktionale GUI Tests testen Aktion und Reaktion von UI-Elementen

- *Record-and-Play*: Aufzeichnen von Benutzeraktionen zum automatisierten Abspielen
  - Sprache der Aufzeichnung oft deklarativ (XML), aber auch imperativer Code, oft Skript-Sprachen
  - Wie viele Tests zerbrechen, wenn ein Widget lediglich seinen Ort auf der Oberfläche verändert?
  - Werkzeuge mit grossen qualitativen Unterschieden am Markt erhältlich
- Geeignet falls Trennung von Entwicklungs- und Testabteilung besteht

UI Tests (III)

Funktionale GUI Tests testen Aktion und Reaktion von UI-Elementen

- *Programmatisch*: Ausprogrammieren der UI-Tests in der Sprache des Produktivcodes
  - Üblicherweise mithilfe einer Bibliothek zum Auffinden der Widgets
  - Bei ungeeigneter Bibliothek/Framework viel Overhead beim Synchronisieren von simulierter Benutzerinteraktion und Threads
- Geeignet falls Programmierer selbst die funktionalen Tests erstellen sollen.

Legacy Code

Problem: Alter Code hat keine Tests und benötigt Änderungen

- Prämissen:
  - Code ohne Tests ändern ist anfällig für Fehler
  - Es ist nicht wirtschaftlich, kompletten Code nachträglich mit Unit-Tests zu versehen
- Zu ändernden Code (und nur diesen) in den Test Harness bringen
- Dann mit Test-First Änderungen vornehmen

Test-First

- auch: *Test Driven Development (TDD)*
- Prozess für Programmierer
- Vorteile
  - Höhere Produktivität
  - Genauere Zeitschätzungen
  - Lesbarer Code
  - Unit-Tests mit 100% Coverage
  - Dokumentation Teil des Ergebnis
- Nachteile
  - Erfordert Disziplin
  - Erste Lernschritte fühlen sich an wie Rückschritt
  - Schwer zu vermitteln
  - Benötigt gesunden Menschenverstand

Test-First Vorgehen

- Testcode schreiben
- Test zum Kompilieren bringen
- Test zum Fehlschlagen bringen
- Code schreiben und Test zum Durchlaufen bringen
- Optional: Refaktorisieren
- Wiederholen