Seminar Simulation und Bildanalyse mit Java, SS2004

Tests in Informatik und Statistik

Modul-Tests mit JUnit

Robert Pintarelli

10.05.2004

Übersicht

- Motivation - Fehler in Softwarepaketen
- JUnit im Überblick
- Unit-Tests mit JUnit
- Das Innenleben von JUnit

Motivation

Grösse einiger Softwarepakete

- UNIX System V (Release 4.0 inkl. X11) : 3.7 Mio. Zeilen Quellcode
- R/3 von SAP: 7 Mio. Zeilen Quellcode
- Windows 2000: 30 Mio. Zeilen Quellcode

Motivation

Durchschnittliche Fehlerquoten

- 50-60 gefunden Fehler pro 1000 LOC während der Entwickung
- Bis zu 3 Fehler pro 1000 LOC nach der Auslieferung
- Bei sicherheitskritischen Anwendungen ca. 0.5 Fehler pro 1000 LOC

Motivation

Beispiele für Fehler in Software

- OB Wahl in Neu-Ulm 1995: 104% Wahlbeteiligung (tatsächlich 52%)
- Bluescreen
- Gepäckverteilungsanlage Flughafen von Denver: 16 Monate verzögerung, Schaden 550Mio USD
- Golfkrieg 1991: 28 Tote, fast 100 Verletzte wg. fehlerhafter Uhrsynchronisation

Motivation

zweites Weinberg’sches Gesetz

> If builders built buldings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

Motivation

Eigenschaften von Software

- Software ist immateriell
- Software ist komplex und zeigt keine „natürlichen“ Strukturen
- Software hat keine natürliche „Lokalität“ (Fehlereingrenzung dadurch oft sehr schwierig)
- Software ist nicht stetig

Unit Tests

- hohe Lokalität
- einfach Aufgebaut
- dienen als Beispiel
- Automatisierbar
- Unterstützen OO-Grundsätze

Unit Tests - Anforderungen an eine Testumgebung

- Die Sprache zur Testspezifikation ist die Programmiersprache selbst
- Anwendungscode und Testcode müssen getrennt werden können
- Unabhängikeit von Ausführung und Verifikation einzelner Testfälle
- Testfälle können beliebig in Testsuiten zusammengefasst werden
- Erfolg oder Misserfolg der Testausführung einfach erkennbar

JUnit im Überblick

- JUnit ist in Java verfasst
- Testfälle gekapselt über `TestCase`
- Testfälle können gruppiert werden
- *setUp()* und *tearDown()* sorgen für Unabhängigkeit der Testfälle
- Einfache Bedienung, schnelle Erfolgskontrolle

JUnit - die TestRunner

---

junit.textui.TestRunner textuelle kompakte Darstellung der Testergebnisse

junit.awtui.TestRunner grafische Darstellung mit Hilfe der AWT

junit.swingui.TestRunner grafische Darstellung mit Hilfe von Swing

JUnit - Der Test-First Ansatz

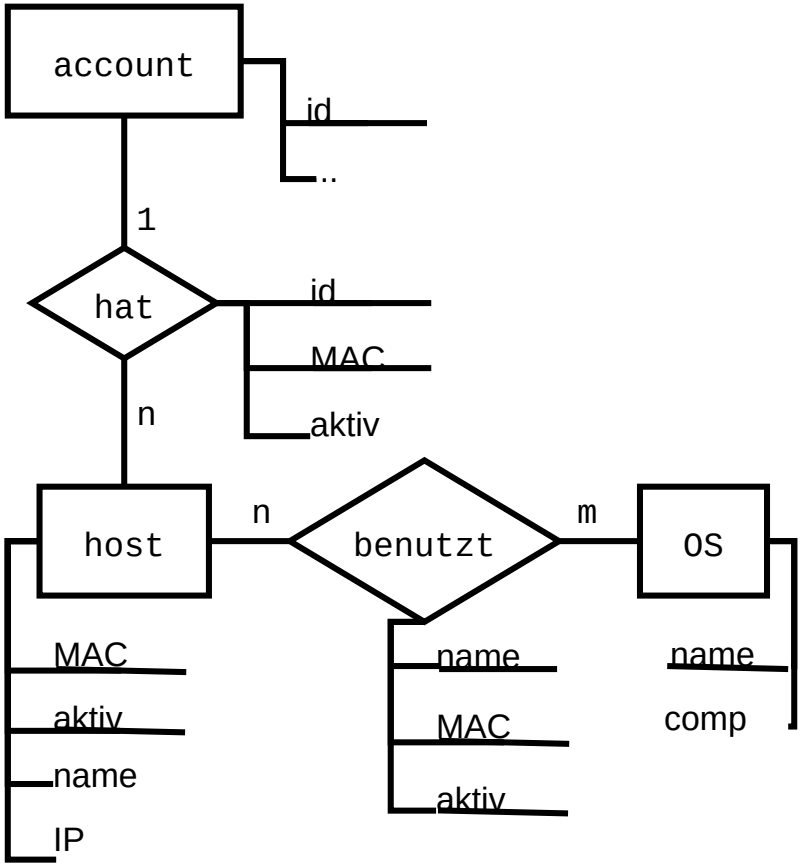
- Testfälle geben Implementierung vor
- Bottom-Up-Methode
- Häufige Reorganisation der Testfälle und der Implementierung (Refactoring)

JUnit - Grundgerüst eines Testfalls

```
import junit.framework.*; //JUnit bekannt machen

public class MeinTest extends TestCase {
  public MeinTest(String name) { //muss so sein
    super(name);
  }
  public void testTrivial() { //ein test
    assertTrue(true);
  }
}
```

JUnit - Beispiel



JUnit - Beispiel - Anforderungen an equals(Object other)

- muss transitiv sein.
- muss reflexiv sein.
- muss symmetrisch sein.
- Vergleich mit *null* muss/sollte immer *false* ergeben.
- Nur Vgl. mit Objekten gleichen Inhalts u. Typs darf *true* ergeben.

JUnit - Beispiel - Tests für equals(Object other)

- *public void testEqualsSymmetric();*
- *public void testEqualsReflexiv();*
- *public void testEqualsTransitiv();*
- *public void testEqualsOther();*
- *public void testEqualsNull();*

JUnit - Beispiel - Fixtures

- *setUp()* zum Bereitstellen der Testumgebung
- *tearDown()* zum Aufräumen
- *setUp()* und *tearDown()* werden vor bzw. nach jedem Aufruf einer *testXXX()*-Methode ausgeführt, unabhängig vom Testausgang

JUnit - Beispiel - hashCode()

- Objekte die bei *equals(Object other)* gleich sind, müssen auch den selben Hash-Wert liefern
- Sinnvollerweise sollte eine „gute“ Hash-Funktion gewählt werden
- *testHashCode()* zum Testen

JUnit - Beispiel - simpleHost()

- Tests und Methoden zum Hinzufügen von Betriebssystememem
- Helferklassen für Dotted-IP, MAC und DNS-Namen notwendig:
  - `seminar.RFC1034`
  - `seminar.RFC1117`
  - `seminar.ieee802`

JUnit - Beispiel - Testsuiten

- Testsuiten zum Gruppieren der Testfälle
- Bei JUnit über `TestSuite` realisiert
- *add()*-Methode zum Hinzufügen von Testfallklassen

JUnit - Beispiel - Abschluss

- Implementation der restlichen Testfälle und Methoden
- mit `AllTests` Code überprüfen

Das Innenleben von JUnit

- `TestCase` ist ein Command Pattern: die *run()*-Methode
- Template Methods in `TestCase`:
  - *setUp()*
  - *runTest()*
  - *tearDown()*

Das Innenleben von JUnit

- `TestResult` als Collecting Parameter
- Zählen von Misserfolgen durch *assertTrue()*, *assertFalse()* und *assertEquals()*
- Problem: *runTest()* als einzige bekannte Testmethode

Das Innenleben von JUnit

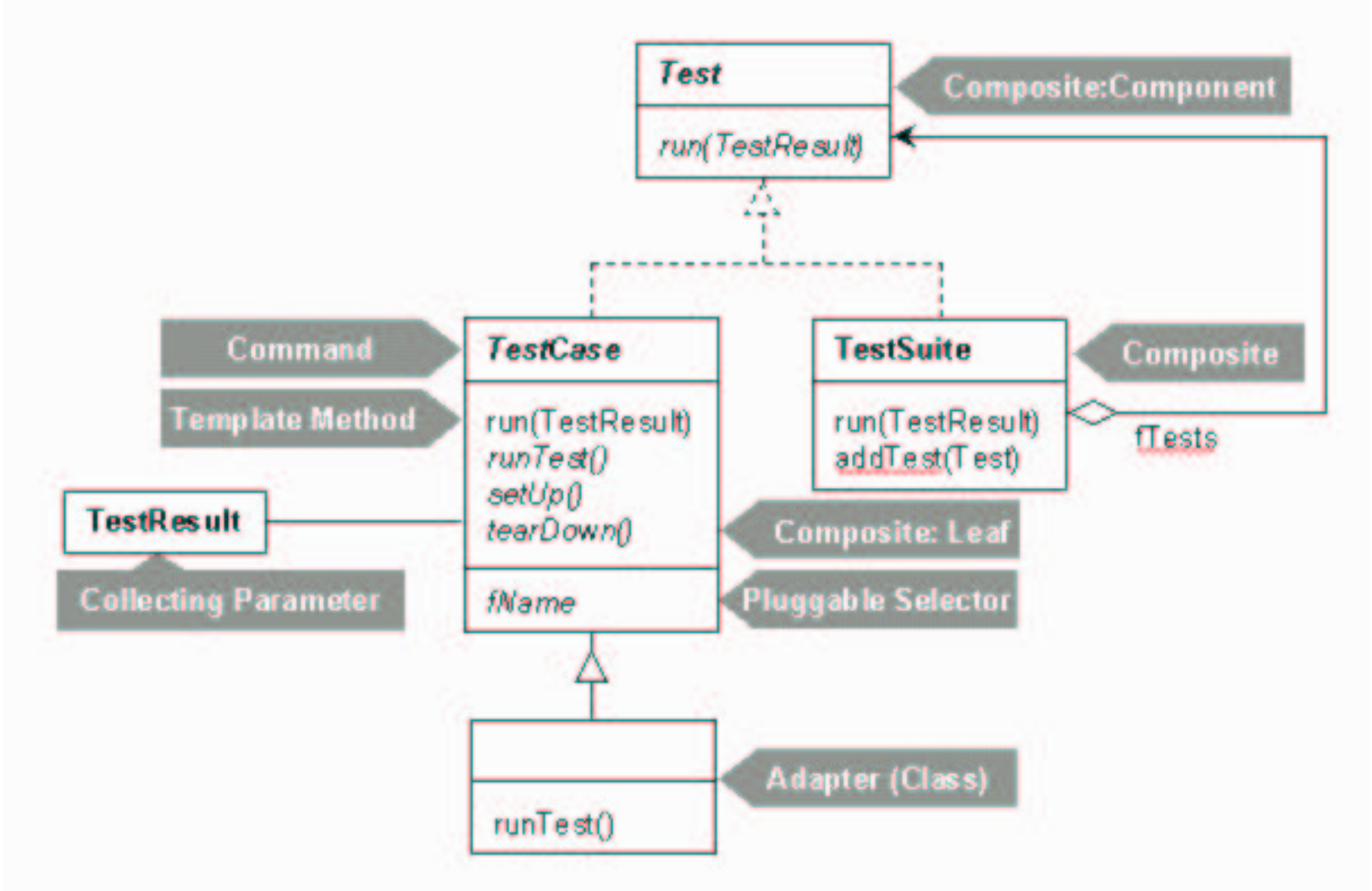
- Pluggable Selector
- individuelles Überschreiben von *runTest()* für jeden Testfall
- in Java leicht durch anonyme innere Klassen möglich

Das Inneleben von JUnit - Testsuiten

- bisher nur eine Testklasse mit einer *run()*-Methode
- Problem: Gruppieren von Tests ohne `TestCase` zu „Missbrauchen“?
- Lösung: Benutzen des Composite-Patterns

Das Innenleben von JUnit - Testsuiten

- Interface `Test` als Superklasse des Verbunds mit *run()*
- `TestCase` als Mitglied des Verbunds
- `TestSuit` als Mitglied des Verbunds, *run()* iteriert über verwaltete `TestCase`-Instanzen


Test
run(TestResult)
Composite:Component
Command
Template Method
TestCase
run(TestResult)
runTest()
setUp()
tearDown()
fName
TestSuite
run(TestResult)
addTest(Test)
Composite
fTests
TestResult
Collecting Parameter
Composite: Leaf
Pluggable Selector
runTest()
Adapter (Class)

Testen von Datenbankapplikation mit DBUnit

- `DatabaseTestCase` als Erweiterung von `TestCase`
- *getConnection()*
- *getDataSet()*

Testen von Datenbankapplikation mit DBUnit - Beispiel

- `seminar.DBWriter` zum Schreiben in eine Datenbank
- Testfall und Implementation zum Einfügen eines Betriebssystems

Ende

Noch Fragen?