

Aufbau eines Testwerkzeugs am Beispiel von JUnit

Üblicher Ansatz für Tests und Fehlersuche:

Print-Befehle, Debugger-Ausdrücke, Test-Skripte
möglichst über globale Variable *debug* steuerbar

Umsetzung in einem OO-Ansatz

Command Pattern

Idee: Objekte mit gemeinsamer *run*-Methode, in welcher die Test-Aktionen gekapselt sind.

```
public abstract class TestCase implements Test {  
    private final String fName; // identifiziert Test  
    public abstract void run(); // zu überladende Methode  
}
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

Wie hängt der Programmierer seinen Testcode ein?

Tests haben eine gemeinsame Struktur:

Aufsetzen der Testumgebung -> Code gegen diese Umgebung laufen lassen -> Ergebnisse mit den Erwartungen vergleichen -> Testumgebung auflösen

Template Method Pattern

Idee: Skelett eines Algorithmus vorgeben, die Methoden werden in Subklasse konkretisiert.

```
public void run() {  
    setUp(); /* jeweils protected und leere Methodenrumpfe */  
    runTest();  
    tearDown();  
}
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

Wie werden die Testergebnisse eingesammelt?

Ausgabe ist unsymmetrisch: Von erfolgreichen Tests ist nur Statistik interessant, sonst genauere Informationen über die Fehlerstelle.

Collecting Result Pattern

Idee: Der Methode ein Objekt übergeben, welches die Ergebnisse einsammelt.

```
public class TestResult extends Object {
    protected int fRunTests; /* Zähler der Testläufe */
    ... }
public void run(TestResult result) {
    result.startTest(this);
    setUp(); runTest(); tearDown();
}
public synchronized void starttest(Test test) { fRunTest++; }
    /* synchronized, da verschiedene Tests auf dasselbe Resultat
    schreiben könnten */
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

Der Test hat einen Fehler entdeckt. Was weiter?

Fehler können planmäßig und unerwartet auftreten. JUnit unterscheidet deshalb *failure* und *error*. Realisierung durch Ausnahmebehandlung mit spezieller Ausnahmeklasse *AssertionFailedError* für failures.

```
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try { runTest(); }
    catch (AssertionFailedError e) // planmäßige Ausnahmen
        { result.addFailure(this, e); }
    catch (Throwable e) // unplanmäßige Ausnahmen
        { result.addError(this, e); }
    /* An der Stelle sind alle Ausnahmen abgefangen! Keine Ausnahme
       wird aus TestCase.run herausgereicht! */
    finally { tearDown(); }
}
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

Wo kommen planmäßige *AssertionFailedErrors* her?

Werden von *assert*-Methoden aus der Klasse *TestCase* ausgelöst (es gibt noch mehr *assertXXX*-Methoden)

```
protected void assert (boolean condition) {  
    if (!condition) throw new AssertionFailedError();  
}
```

Auf sammeln in entsprechenden Aggregationen in *TestResult*

```
public synchronized void addError(Test test, Throwable t) {  
    fErrors.addElement(new TestFailure(test,t));  
}  
  
public synchronized void addFailure(Test test, Throwable t) {  
    fFailures.addElement(new TestFailure(test,t));  
}  
  
public class TestFailure extends Object { // Wrapper-Klasse  
    protected Test fFailedTest; protected Throwable fThrownException;  
}
```

6. Aufbau eines Testwerkzeugs

JUnit kommt mit verschiedenen fertigen Subklassen von *TestResult*, z.B. *TextTestResult* für textuelle Darstellung oder *UITestResult* für Einbindung in grafische Testumgebung. Erweiterungen sind möglich, z.B. *HTMLTestResult*.

TestCase: viele, aber nur wenig variierende Klassen

Lösung in JUnit: Verwendung innerer Klassen erspart das Erfinden von Klassennamen (Adapter Pattern)

```
TestCase test = // Wiederverwendung der generischen Klasse MoneyTest
    new MoneyTest(„testMoneyEquals“) {
        // neue innere Klasse als Subklasse
        protected void runTest() { testMoneyEquals(); }
        // Methode runTest überschrieben
    }
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

Anderer möglicher Lösungsansatz: Parametrisierte Klassen, wird erst von Java 5 unterstützt.

Kann auch über Reflection und Stringmanipulation simuliert werden.

Ausführung mehrerer Tests „im Stück“

Composite Pattern

Idee: Anordnung der Objekte in einer Baumstruktur, um Teil-Ganzes-Hierarchien auszudrücken. Einzelne Objekte und Objekt-Aggregationen werden auf dieselbe Weise behandelt.

Bestandteile:

Komponente: Schnittstellendefinition, mit welcher unsere Tests interagieren sollen. (*interface Test*)

Komposition: Implementierung dieser Schnittstelle samt Management von Test-Sammlungen. (*class TestSuite implements Test*)

Blatt: Repräsentation eines TestCase in einer solchen Komposition, welcher die Komponenten-Schnittstelle implementiert.

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

```
public interface Test {  
    public void run (TestResult test);  
}  
public class TestSuite implements Test {  
    private Vector fTests = new Vector();  
  
    public void run () { // Delegiert Testausführung auf die Kinder  
        for (Enumeration e=fTests.Elements(); e.hasMoreElements;) {  
            Test test = (Test) e.nextElement();  
            test.run(result);  
        }  
    }  
  
    public void addTest(Test test) { // Clients können neue Tests hinzufügen  
        fTests.addElement(test);  
    }  
}
```

6. Aufbau eines Testwerkzeugs

Nachteil dieser Methode: Alle Tests müssen per Hand in eine entsprechende TestSuite eingetragen werden (statischer JUnit-Zugang)

Alternative Lösung: Java sucht mit Reflection-Methoden nach Methoden mit entsprechendem Namen und fügt diese selbst zu einer TestSuite zusammen (dynamischer JUnit-Zugang).

Zusammenfassung

