Testwerkzeuge

Prof. Dr. Walter F. Tichy
Universität Karlsruhe

Testwerkzeuge der Software-Amateure

- Testen mit Ausgabe-Anweisungen im Programm
  - drucken Variablen an bestimmeten Stellen im Programm aus
  - werden nach Test gelöscht oder auskommentiert
  - besser: per Übersetzerdirektiven (#ifdef in C) zu- und ausschalten.
- Testen mit einem interaktiven Debugger
  - Haltepunkte festlegen
  - Variablen interatkiv an den Haltepunkten inspizieren
  - Information darüber ist nach Debuggerlauf verloren
- Testen mit Test-Skripten
  - laufen automatisch ab
  - verknüpft mit print-Anweisungen
- Allen gemeinsam: manuelle Überprüfung der Ausgaben

Nachteile dieser Methoden

- Bei Programmänderungen muss man Testfälle wiederholen.
  - Dies ist mühsam und praktisch oft unmöglich
    - Ausgabe-Anweisungen sind oft nicht mehr vorhanden
    - wenn doch vorhanden, sind die richtigen Auskommentierungen zu entfernen (und später wieder einzusetzen)
    - bei Testen mit Debugger ist keinerlei Information vorhanden, was gestestet werden soll.
  - Ergebnisse müssen manuell überprüft werden
    - nur der Programmierer weiß, was die Ausgaben bedeuten und wann sie richtig sind
    - selbst dieser verliert dieses Wissen mit der Zeit.
  - Es ist schwierig, Testfälle für viele Komponenten zusammenzufassen und gesammelt auszuführen
    - technisch schwierig (Testaufbauten unterschiedlich)
    - zu viele Ausgabedaten zu überprüfen.

Antwort der Profis:

1. Benutze Zusicherungen (assertions)

2. Schreibe automatisch ablaufende Testfälle, die sich selbst überprüfen.

1. Zusicherungen (Assertions)

- Zusicherungen sind boolsche Funktionen, die innerhalb von Methoden, meist am Anfang oder Ende, ausgeführt werden. Sie überprüfen dynamisch
  - Vor- und Nachbedingungen (z.B. dass ein übergebener Parameter positiv sein muss oder eine Referenz nicht unbestimmt sein darf)
  - Invarianten einer Datenstruktur.
- Im Fehlerfall melden sie sich mit einer Ausnahme oder Ausgabe.

Zusicherungen

- Bei Eiffel und Java (ab JDK 1.4) sind Zusicherungen in die Sprache integriert.
  - Java: assert
  - Eiffel: assume, ensure, invariant
- Bei C und C++ kann man sich mit Makros behelfen.
  - assert.h
- Wichtig: Zusicherungen können zu- und abgeschaltet werden.

Zusicherungen in Java (ab 1.4)

```
AssertStatement:
  assert Expression1;
  assert Expression1 : Expression2;
```

- Zusicherungen ausgeschaltet: keine Wirkung
- Zusicherungen eingeschaltet:
  - Erster Ausdruck wird ausgewertet.
  - Wenn ‚false': throw AssertionError
    - Programm endet i.d.R. mit Stapelabzug
    - Bei Form 2 wird zweiter Audruck ausgewertet und als new AssertionError(Expression2) übergeben
  - Wenn ‚true': keine weitere Wirkung
- Zusicherungen sind paketweise bei Start der VM einschaltbar.

Benutzung der Zusicherungen (1)

- Zusicherungen werden am Anfang und Ende von Methoden eingesetzt, um die Vor- und Nachbedingungen sowie Invarianten der Datenstruktur zu überprüfen.
  - am Anfang werden Parameterwerte überprüft, und am Ende Ergebnisse.
  - Am Ende einer Sortiermethode könnte z.B. eine Zusicherung überprüfen, ob die zu sortierenden Daten wirklich aufsteigend angeordnet sind.
  - Um nachzuprüfen, ob vor und nach einer Manipulation ein Baum noch balanziert ist, könnte man die Funktion istBalanciert() schreiben und in Zusicherungen aufrufen:
    - assert istBalanciert();

Benutzung der Zusicherungen (2)

- Sichere unerreichbare Codesegmente:

```
switch(farbe) {
    case GRÜN: ...      break;
    case ROT:  ...      break;
    default:   assert false;  // unerreichbar
```

- Prüfe Eingabeparameter nichtöffentlicher Methoden
- Prüfe Nachbedingungen und Invarianten aller Methoden
- Aber: Eingabeparameter öffentlicher Methoden
  - Besser mit Ausnahme IllegalArgumentException
    - Damit Klientenprogramme darauf reagieren können
    - Zumindest bei Bibliotheken
    - Falsche Parameter bei öffentlichen Methoden sind nicht unbedingt unerwartet (das ist eine Entwurfsentscheidung)

Benutzung von Zusicherungen (3)

- Mit Zusicherungen können Mißverständnisse und Fehlinterpretationen der Programmierer rasch aufgedeckt werden.
- Zusicherungen sind keine Testfälle--es werden lediglich bestimmte Bedingungen im Programmlauf überprüft.
  - Sofern sie zugeschaltet sind, werden sie beim Ablauf von Testfällen mit ausgeführt.
- In Produktionsläufen werden Zusicherungen aus Leistungsgründen abgeschaltet
  - Bei Java auch selektiv für einzelne Klassen
- Bei Auftreten eines Fehlers oder beim Testen von Programmänderungen werden die Zusicherungen wieder zugeschaltet.

2. Automatisch ablaufende Testfälle

- **Testfall:** führt ein Programm unter bekannten Bedingungen und bei gegebenen Eingabedaten aus; das erwartete Ergebnis ist ebenfalls bekannt.
  - Ziel ist das Aufdecken von Fehlern.
- Automatisch ablaufende Testfälle sind solche, die das Ergebnis ihrer Ausführung selbst überprüfen.
  - Sie geben nur einen boolschen Wert zurück (erfolgreich oder fehlgeschlagen)
  - oder lösen bei Fehlschlag eine Ausnahme aus.

Rahmenarchitektur für automatisch ablaufende Testfälle

- Benötigt ist eine Rahmenarchitektur, die
  - das Schreiben der Testfälle vereinfacht,
  - Testfälle zu Testsuiten zusammenfassen kann,
  - ganze Testsuiten automatisch ablaufen lässt und nur die Versager meldet,
  - standardisierend wirkt (jeder macht´s gleich, Zusammenfassung großer Testsuiten damit möglich).
- Regressionstest: wiederholtes Ausführen einer Testsuite nach Programmänderungen.
  - Hierfür ist die automatische Ausführung besonders wichtig, da man in der Praxis Tausende von Testfällen hat.

Beispiel: JUnit ein Testrahmen für Java

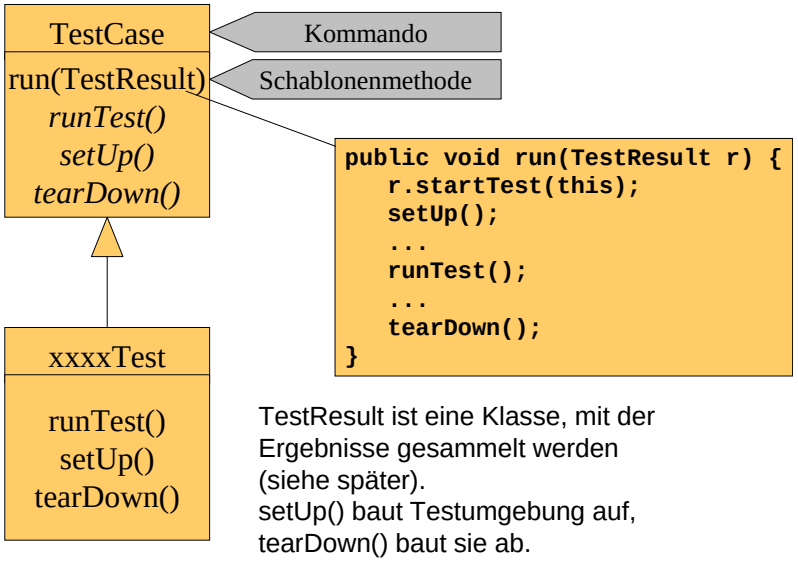
- Erlaubt schnellen und hierarchischen Aufbau von Testsuiten
- liefert die fehlgeschlagenen Testfälle
- textuelle oder graphische Anzeige

- Autoren: Erich Gamma, Kent Beck
- Wo:
  - http://sourceforge.net
  - http://junit.org

Aufbau von JUnit

- Grundidee: ein Testfall wird in eine Klasse gekleidet und erhält eine einheitliche Schnittstelle zur Ausführung:
  - run() (Kommando-Muster)
  - Dazu kommen noch Methoden für Auf- und Abbau der Testkonfiguration.
- Testfälle werden in ein Kompositum hineininstanziert, wodurch sich hierarchische Testsuiten aufbauen lassen.
- Zusätzlich gibt es eine Klasse TestResult, die die Test-Ergebnisse aufsammelt.
- Um eine Testsuite zu starten, ruft man die Methode run() auf.

Wichtigste Klasse: TestCase



TestResult ist eine Klasse, mit der Ergebnisse gesammelt werden (siehe später).
setUp() baut Testumgebung auf, tearDown() baut sie ab.

Wie macht man einen Testfall?

- Bilde Unterklasse von TestCase (genannt xxxTest) mit Instanzvariablen für Testumgebung.
  - Überschreibe die Methoden setUp(), runTest() und tearDown()
  - runTest() überprüft die Richtigkeit eines Ergebnisses typischerweise durch Vergleich mit dem erwarteten Resultat.
    - Dazu gibt es die Methoden assertEqual(), assertTrue(), assertNotNull() u.a.
- Bei Fehlschlag lösen die assert-Methoden eine Ausnahme aus
  - Wird von TestCase aufgefangen
  - Wird in TestResult gesammelt

Triviales Beispiel

```
public class MatheTest extends TestCase {
     protected double t1;
     protected double t2;

     public MatheTest(String name) {
         super(name);
     }

    protected void setUp() {
         t1= 2.0;
         t2= 3.0;
     }

    public void runTest() {
        double summe = t1 + t2;
        assertEqual(summe, 5.0);
    }
}
```

Ausführung mit:

```
Test test=
 new MatheTest("Addition")
test.run();
```

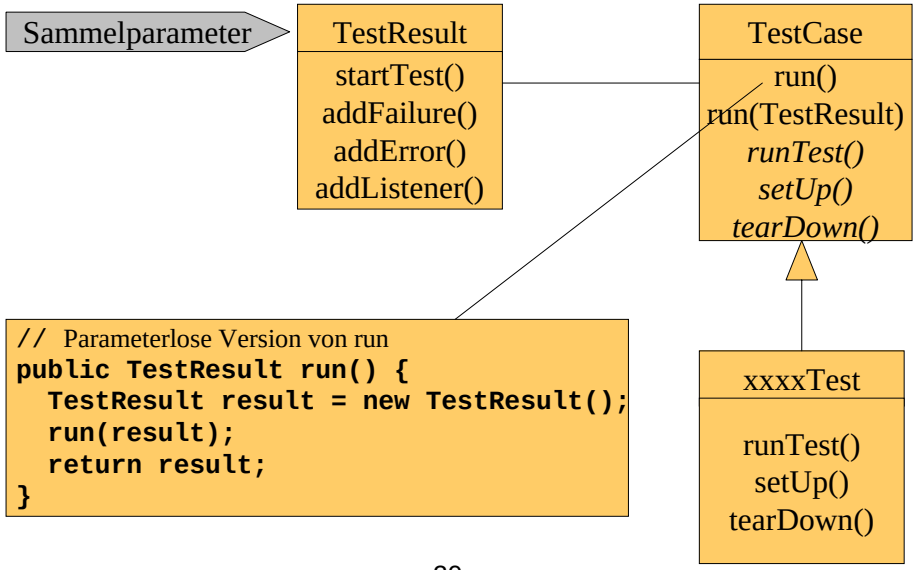

Ausnahmebehandlung

- Die Methode TestCase.run() muss zwei Arten von Ausnahmen behandeln:
  - die durch die assert-Methoden aufgrund entdeckter Fehler ausgelösten Ausnahmen
    - AssertionFailedError
  - unerwartete, vom Laufzeitsystem entdeckte Fehler wie z.B. Feldgrenzenfehler.
    - Unterklassen von RuntimeException
    - Sonstige schwere Probleme: xxxError
    - Diese Fehler werden automatisch entdeckt, ohne dass das Testprogramm danach prüft.

Mehr über TestCase.run()

```
public void run(TestResult result) {
  result.startTest(this);
  setUp();
  try {
     runTest();
  }
  catch (AssertionFailedError e) {
     result.addFailure(this, e);
  }
  catch (Throwable e) {
     result.addError(this, e);
  }
  finally {
     tearDown();
  }
```


Noch mehr über TestCase




```
// Parameterlose Version von run
public TestResult run() {
  TestResult result = new TestResult();
  run(result);
  return result;
}
```

Aufruf von Testmethoden (1)

- Meistens kann man mit der gleichen Testumgebung gleich mehrere Tests fahren.
  - Z.B. könnte MatheTest die Methoden testeAddition() und testeNullDivision() bieten.
  - Diese Methoden sitzen dann in der gleichen Klasse (aber nur eine runTest-Implementierung ist möglich).
- Benutze dafür eine innere Klasse als anonymen Klassenadapter. Dieser adaptiert den Namen der Testmethode auf die runTest-Schnittstelle.

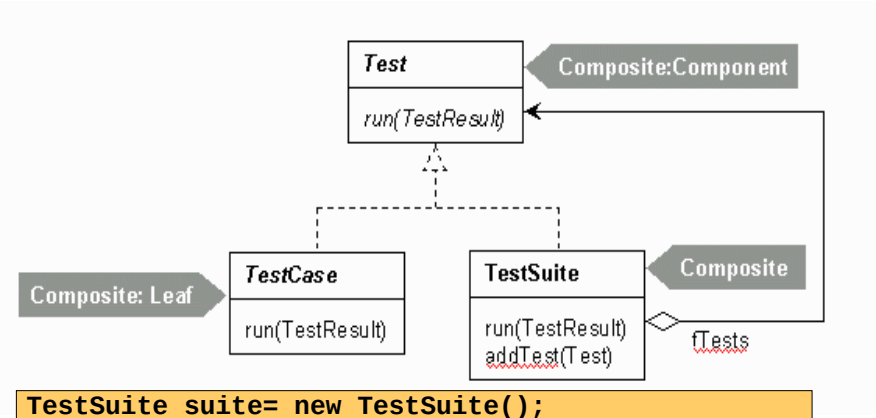
```
Test test = new MatheTest() {
  protected void runTest() { testeAddition(); }
});
test.run();
// analog für testeNullDivision
```


Aufruf von Testmethoden (2)

- Alternativ kann man die aufzurufende Methode dynamisch übergeben
  - Selektoreinschub, pluggable selector
- Wenn man im Konstruktor eines Testfalles den Namen einer enthaltenen Testmethode mitgibt, wird diese von runTest automatisch aufgerufen
  - und man überschreibt runTest dann nicht
- Bequemer, aber nicht typsicher wie die Adaptermethode.

```
Test testAdd = new MatheTest("testeAddition"));
Test testDiv = new MatheTest("testeNullDivision"));
testAdd.run();
testDiv.run()
```

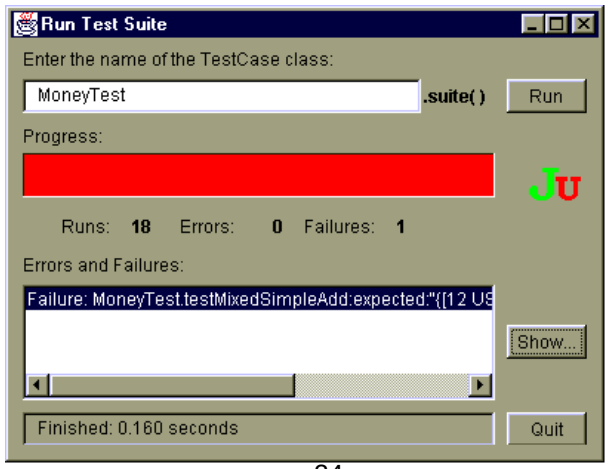

Testsuiten mit Kompositum



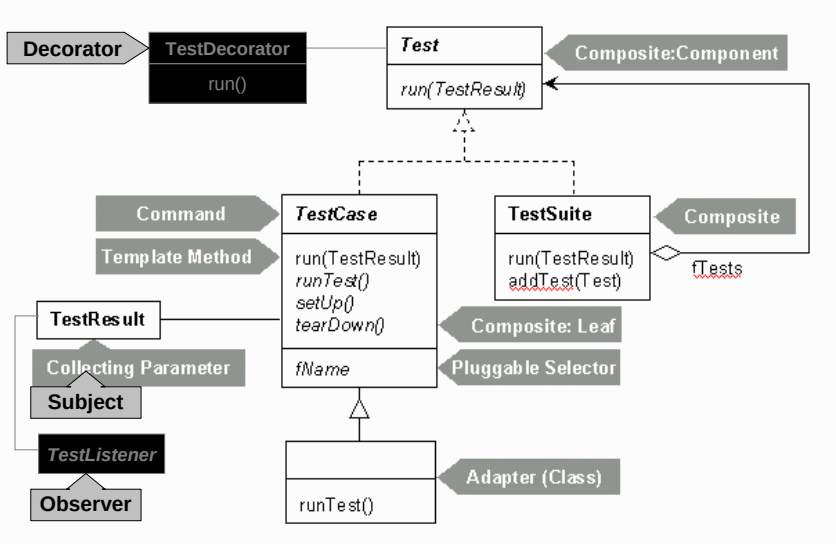

```
TestSuite suite= new TestSuite();
suite.addTest(new MatheTest("testeAddition"));
suite.addTest(new MatheTest("testeNullDivision"));
suite.run();
```


Anzeige „TestRunner“

- implementiert Schnittstelle TestListener
  - abstrakter Beobachter von TestResult



Das Gesamtbild von JUnit



Zusammenfassung

- Automatisch ausführbare Testfälle sind besonders für den Regressionstest wichtig.
- Regressionstests helfen verhindern, dass alte Fehler wieder auftauchen.
- JUnit und ähnliche Regressionstest-Werkzeuge sind in der Praxis extrem wichtig.
- Sie erfordern, dass Testfälle sich selbst überprüfen.
- Können mit Zusicherungen kombiniert werden.