Software-Qualitätsmanagement

Vorlesung im Modul 10-202-2319 Software-Management

Sommersemester 2009

Prof. Dr. Hans-Gert Gräbe

http://bis.informatik.uni-leipzig.de/HansGertGraebe

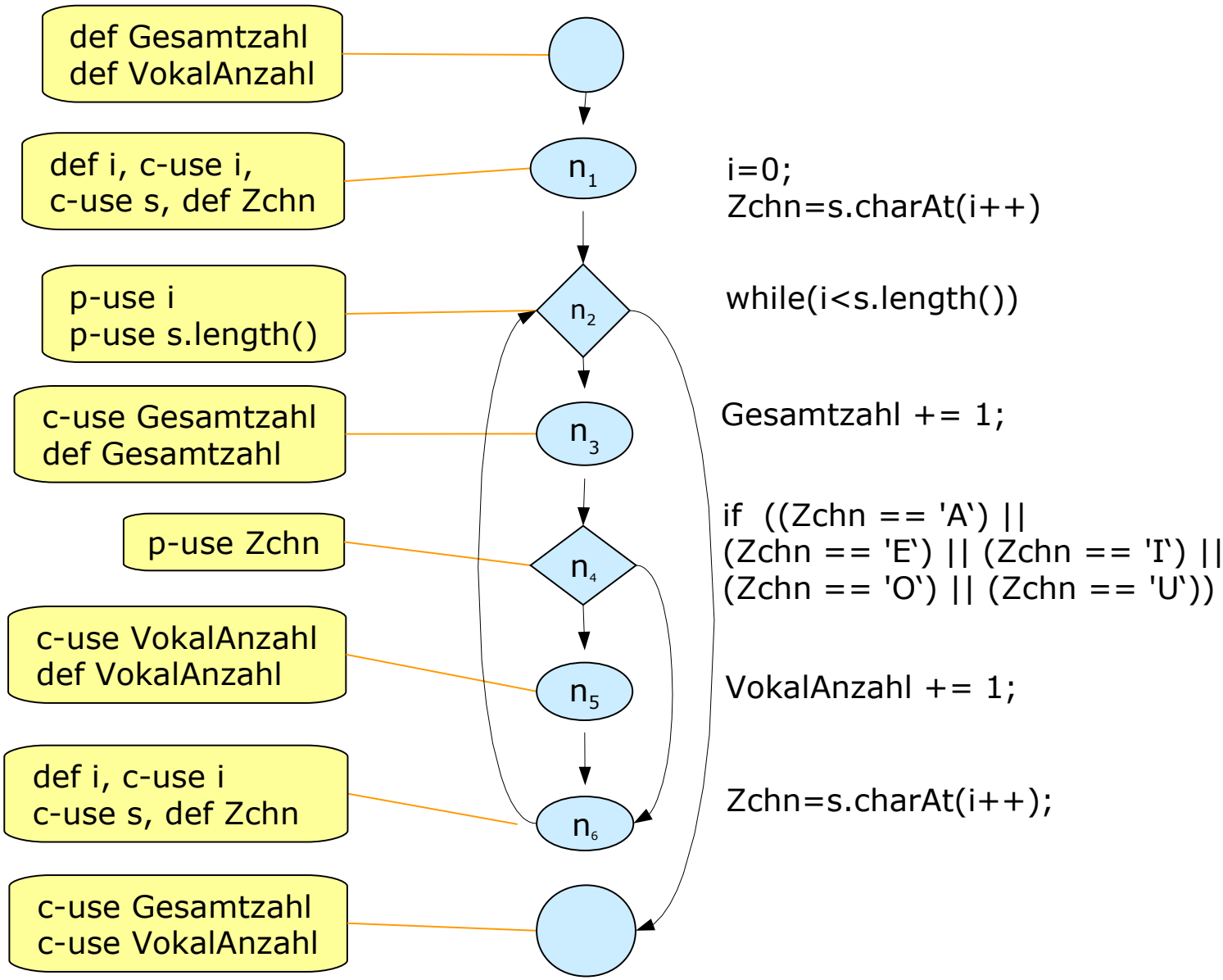
Datenflussorientierte Strukturtestverfahren

- Ebenfalls dynamisches Strukturtestverfahren
- Im Gegensatz zu kontrollflussorientierten Verfahren werden Datenbenutzungen und damit eher globale Aspekte getestet.
- Analyse der Programmdynamik an Hand der Dynamik der in Variablenwerten gespeicherten Programmzustände
  - Aufstellen des Datenflussdiagramms
    - Sichtbarkeit und Lebensdauer von Bezeichnern
    - Variablenidentitäten: Gültigkeitskontext und Kontrollfluss
    - lesende und schreibende Zugriffe auf Variablen (use, def)
    - Unterscheidung lesender Zugriffe in Anweisungen und Bedingungen (c-use, p-use)
- Eignen sich besonders für den Test von Datenobjekt- und Datentypmodulen sowie Klassen.
- Nur wenige Testwerkzeuge vorhanden.

Defs/Uses-Verfahren

Klassifikation der Variablenzugriffe nach

- Zuweisung (set-Methode, Definition, *def*)
  - Variablenwert wird an einer solchen Stelle geändert
- Zugriff zur Berechnung von anderen Werten (*computational-use, c-use*)
  - Auswirkung auf Wert anderer Variablen (Programmzustand)
- Zugriff zur Berechnung von Wahrheitswerten in Bedingungen (*predicate-use, p-use*)
  - Auswirkung auf Wert der Testbedingung (Kontrollfluss)


def Gesamtzahl
def VokalAnzahl
def i, c-use i,
c-use s, def Zchn
p-use i
p-use s.length()
c-use Gesamtzahl
def Gesamtzahl
p-use Zchn
c-use VokalAnzahl
def VokalAnzahl
def i, c-use i
c-use s, def Zchn
c-use Gesamtzahl
c-use VokalAnzahl
n1
n2
n3
n4
n5
n6
i=0;
Zchn=s.charAt(i++)
while(i<s.length())
Gesamtzahl += 1;
if ((Zchn == 'A') ||
(Zchn == 'E') || (Zchn == 'I') ||
(Zchn == 'O') || (Zchn == 'U'))
VokalAnzahl += 1;
Zchn=s.charAt(i++);

Datenflussgraph

- Knoten: jede Verwendung jeder Variablen B im Quelltext:
  (def B) (c-use B) (p-use B)
- Kanten: von (def B) zu allen nachfolgenden (use B), gelegentlich auch eine Kante pro möglichem Kontrollfluss
- Kantenmarken: Angabe des / der möglichen Kontrollflüsse (als Pfade im Kontrollflussgraphen)

Datenflussorientierte Test-Verfahren

- ***all defs* Kriterium**
  - Testfälle sind so zu wählen, dass jeder Wertzuweisung an eine Variable auch eine Wertbenutzung folgt.
  - Zu jeder Variablen gibt es einen Testfall, in welchem die Variable wenigstens einmal geschrieben und gelesen wird.
  - **Idee**: Jede Variable hat einen „Zweck", eine Semantik, die wenigstens einmal exemplarisch geprüft wird.
- Spezialfall: Kontrolle, ob alle definierten Variablen auch verwendet werden (statischer Test, den der Compiler durchführen kann)
  - Charakterisiert durch fehlende abgehende Pfeile im DFD.
  - Variablen, die nicht benutzt werden, weisen auf Programmfehler hin.
  - Problem fällt beim Aufstellen der Testfälle auf.

- ***all p-uses* Kriterium**
  - Testfälle sind so zu wählen, dass jede Kombination einer Wertzuweisung mit jeder prädikativen Nutzung dieses Variablenwerts überdeckt wird
  - Zu jedem Kontrollfluss, in dem eine Variable geschrieben und später in einer Bedingung gelesen wird, gibt es einen Testfall, welcher diesen Kontrollfluss abdeckt.
  - **Idee**: Prüfen der Auswirkung einer Wertzuweisung an eine Variable auf alle relevanten Bedingungsknoten
    - Fokus auf die bedingungsrelevanten Variablen
  - beinhaltet Zweigüberdeckung

- ***all c-uses* Kriterium**
  - Testfälle sind so zu wählen, dass jede Kombination einer Wertzuweisung mit jeder berechnenden Nutzung dieses Variablenwerts überdeckt wird
  - Zu jedem Kontrollfluss, in dem eine Variable geschrieben und später in einer Berechnung gelesen wird, gibt es einen Testfall, welcher diesen Kontrollfluss abdeckt.
  - **Idee**: Prüfen der Auswirkung einer Wertzuweisung an eine Variable auf alle kausal davon abhängenden Ausdrücke.
    - Fokus auf die berechnungsrelevanten Variablen

- ***all c-uses / some p-uses* Kriterium**
  - Testfälle sind so zu wählen, dass jede Kombination einer Wertzuweisung mit jeder berechnenden Nutzung dieses Variablenwerts überdeckt wird.
  - Falls kein berechnender Zugriff existiert, so muss der Wert in mindestens einem Prädikat benutzt werden.
  - **Idee**: Prüfen der Auswirkung einer Wertzuweisung an eine Variable auf alle kausal davon abhängenden Ausdrücke und exemplarischer Test von nur bedingungsrelevanten Variablen

- ***all p-uses / some c-uses* Kriterium**
  - Testfälle sind so zu wählen, dass jede Kombination einer Wertzuweisung mit jeder prädikativen Nutzung dieses Variablenwerts überdeckt wird.
  - Falls kein prädikativer Zugriff existiert, so muss der Wert in mindestens einer Berechnung benutzt werden.
  - **Idee**: Prüfen der Auswirkung einer Wertzuweisung an eine Variable auf alle davon abhängenden Bedingungen und exemplarischer Test von nur berechnungsrelevanten Variablen

- ***all uses* Kriterium**
  - Testfälle sind so zu wählen, dass jede Kombination einer Wertzuweisung mit jeder Nutzung dieses Variablenwerts überdeckt wird.
  - komplettester und damit aufwändigster datenflussorientierter Test

Leistungsfähigkeit nach Studie [Girgis, Woodward 86]
Vergleich *all defs, all p-/c-uses:*

- *all c-uses:* 48% der Fehler, insbesondere Berechnungsfehler,
- *all p-uses:* 34% und entdeckt Kontrollflussfehler,
- *all defs:* 24% der Fehler, aber keine Kontrollflussfehler

Weitere Verfahren

- **Idee**: Überdeckung längerer Sequenzen aus Zuweisung und Nutzung
  - *Required k-Tuples Test*
- **Idee**: Orientierung nicht an abgehenden, sondern an ankommenden Pfeilen im DFG
  - *Datenkontextüberdeckung*: jede mögliche Herkunft eines Werts wird überdeckt.
  - *geordnete Datenkontextüberdeckung*: zusätzliche Beachtung der Zuweisungsreihenfolge

Überblick

- **Idee**: Testfälle werden aus den Programmspezifikationen abgeleitet.
  - Quellcode wird nicht benötigt, deshalb auch „Black-Box-Verfahren" genannt
  - Strukturtest = Test der inneren Programmlogik
  - Funktionaltest = Test der äußeren Programmsemantik
- **Ziel**: möglichst umfassende, aber redundanzarme Prüfung der spezifizierten Funktionalität
  - Analog der strukturellen spricht man von funktioneller Überdeckung
  - Umfang oft im Pflichtenheft als Qualitätsprüfung vereinbart
- **Problem**: Bereich der möglichen Eingabewerte ist sehr groß oder sogar unendlich groß

Funktionale Äquivalenzklassenbildung

- **Idee:** Einteilung des Definitionsbereichs in endliche Anzahl von Klassen „ähnlicher" Werte und Prüfung an je einem exemplarischen Vertreter pro Klasse
  - Klasseneinteilung längs „typischen" Programmverhaltens
    - muss aber nicht mit innerer Programmstruktur zusammenhängen
  - Korrektheit auf einem typischen Vertreter lässt Korrektheit auf der ganzen Klasse erwarten
    - **Natürlich kein Beweis der Korrektheit**!
  - Nicht unbedingt Äquivalenzklassen im streng mathematischen Sinn, da sich Klassen überschneiden können.

- **Bewertung:**
  - Geeignet zur Herleitung repräsentativer Testfälle
  - Nachteil: Betrachtung von einzelnen Werten, dadurch werden keine Wechselwirkungen oder Abhängigkeiten getestet

Regeln zur Bildung von Klassen

Ist der Eingabebereich

1. Ein zusammenhängender Wertebereich $a \leq x \leq b$
   - drei Bereiche (einer gültig, zwei ungültig)
2. Eine Menge von n Werten, welche unterschiedlich behandelt werden
   - für jeden gültigen Wert eine Klasse sowie eine Klasse für alle ungültigen Werte
3. Eine Bedingung, die zwingend erfüllt sein muss
   - eine Klasse der Werte, für welche die Bedingung erfüllt ist und deren Komplement

Oft ergibt sich die Einteilung des Eingabebereichs als Vereinigung von Urbildmengen, d.h. Eingaben, welche dieselbe Ausgabe erzeugen, werden in eine Klasse zusammengefasst.

Grenzwertanalyse

Idee:

- Basiert auf der funktionalen Äquivalenzklassenbildung,
- Nutzt jedoch nicht irgendwelche Elemente aus den Klassen, sondern Werte, die am Rand der Klasse liegen.
  - Erfahrung besagt, dass durch Grenzwerte Fehler besonders effektiv entdeckt werden.
- Setzt sinnvollen Grenzbegriff (Topologie, Ordnung) auf der Menge der Eingabewerte voraus

Bewertung:

- Sinnvolle Erweiterung und Verbesserung der funktionalen Äquivalenzklassenbildung.

Ähnlich: Test spezieller Werte (Null-Tests, Nullpointer-Tests etc.),
Zufallstest (Auswahl zufälliger Repräsentanten der Klassen)

Test von Zustandsautomaten

- **Idee:** Technische Software ist oft als Menge von Zuständen und Übergängen modelliert und als Zustandsdiagramm spezifiziert. Testfälle sind an dieses Modell angepasst auszuwählen.
  - Minimale Teststrategie: Jeder Zustandsübergang ist mindestens einmal abzudecken
- Überdeckung aller Zustandsübergänge garantiert keinen vollständigen Test (analog Zweigüberdeckung)

- **Bewertung**
  - Gut geeignetes Testverfahren, falls die Spezifikation schon als Zustandsautomat vorliegt.
  - Gut geeignet für den Test von Klassen, wenn der Objektlebenszyklus vorliegt.

Test von GUI-Komponenten und anderen ereignisbasierten Teilen

Probleme

- Ereigniskonzept ist weitere Kommunikationsebene im Programm.
- Ereigniskonzept modelliert Nebenläufigkeit von Programmteilen.
- Ereignisse werden oft durch manuelle Nutzerinteraktionen ausgelöst und sind so nur bedingt automatisierbar.
- Ereignisse sind kaskadierend auf verschiedenen Abstraktionsebenen implementiert (Bsp.: MouseEvent vs. buttonPressed)

Test von GUI-Komponenten und anderen ereignisbasierten Teilen

Modellierung und Tests

- Ereignisse werden über Ereignis**kanäle** verteilt, die zur Designzeit als potenzielle Wege der Ereignispropagierung angelegt werden.
- Ereignisse fokussieren gewöhnlich auf einen speziellen **Anwendungsfall** mit eingeschränktem Zugriff auf die Datenlandschaft.
- Entspricht speziellem **Programmzustand** mit eingeschränkten funktionellen Möglichkeiten
  - Aufstellen eines entsprechenden **Zustandsdiagramms** mit entsprechenden **Knoten** sowie **Vor**- und **Nach**bedingungen
  - Das Generieren der Testfälle sollte dieser Strukturierung folgen
  - Oft hierarchischer Zugang sinnvoll.

Beispiel: Planung von GUI-Tests [Memon, Pollack, Lou Soffa, 1999]

- Herausfinden der **primitiven Operationen**, die in 1-1-Korrespondenz mit direkten Mausaktionen stehen.
- Finde unter ihnen die **Expansionsoperatoren** (etwa Pulldown-Menüs)
  - expandieren die Menge der verfügbaren Aktionen/Zustände und beschneiden zugleich andere Expansionsmöglichkeiten
  - andere Operatoren heißen **Interaktionsoperatoren**, da sie mit der unterliegenden Software interagieren.
- Zu jeder Expansionssequenz wird ein **Zwischenoperator** konstruiert.
  - Bsp.: Edit expandiert zu Cut, Paste (primitiv), Edit+Cut, Edit+Paste (Interaktionssequenzen)
  - Ansatz vermeidet Generieren von Testfällen nur für Edit.

- Finde unter diesen die **abstrakten Operatoren**, die nach Aktivieren die GUI-Interaktion monopolisieren (Bsp: Edit+Preferences)
  - Typische Struktur:

  - Ansatz ist selbstähnlich, da Aktion dieselbe Struktur hat.
  - Cluster- und Subclusterstruktur entspricht einer baumartigen Struktur der Aktionen und Teilaktionen
- Verschiedene Aktionen können gemeinsame Teilaktionen haben.
  - Bsp.:Open, SaveAs, Open.Select, Open.Up, Open.Home, Open.OK, SaveAs.Select, SaveAs.Up, SaveAs.Home, SaveAs.OK
  - Identifiziere **gemeinsame Teilaktionen**, da für diese nur einmal Testfälle zu generieren sind.

- Planung der **Testfälle**: Zunächst Plan für die „höherwertigen" abstrakten Operatoren, schrittweise Verfeinerung
  - Beispiel MS-WordPad: 325 primitive Operatoren, aber nur 32 der obersten Abstraktionsstufe
  - Vor- und Nachbedingungen sind auf dieser Ebene zudem meist einfach zu identifizieren
  - Teilpläne lassen sich daraus werkzeuggestützt generieren
  - abstrahiert von low-level-Details wie Fonts, Farben usw.
  - funktionale Änderungen am GUI können einfach in der Testsuite berücksichtigt werden.

Testen von Klassen

Im [Balzert] wenig systematisch aufbereitet, dort im Kap. 5.13.

- Alle bisherigen Testverfahren waren auf den funktionalen Test von Methoden unter dem imperativen Paradigma ausgerichtet.
- Kommunikation zwischen den Methoden desselben Objekts erfolgt sowohl über die Aufrufparameter als auch den Zustand der Objektattribute (Objekt ist immer implizit ein Parameter).

Kleinste sinnvolle Testeinheit im OO-Bereich ist also die Klasse.

Weitere Besonderheiten von Tests im OO-Bereich

- Wiederverwendbarkeitskonzept
  - Einsatzzweck von Klassen oft nicht genau umrissen
  - Allgemeinheit führt zu vielen möglichen Testfällen
- Vererbung von Attributen und Methoden
  - Redundanz wird eliminiert zu Lasten von zusätzlichen Abhängigkeiten
- Polymorphismus und dynamische Bindung
  - neue Testverfahren nötig
  - Test jeder möglichen Bindung für Polymorphismus nötig
- folgende Arten von Klassen sind zu unterscheiden:
  - normale Klassen
  - abstrakte Klassen
  - parametrisierte Klassen

Testen normaler Klassen:

1. Erzeugung einer instrumentierten Instanz der zu testenden Klasse.
2. Überprüfung jeder einzelnen Operation für sich.
   - zunächst Operationen, die den Objektzustand nicht ändern, anschließend die zustandsverändernden Operationen
   - Testfälle wie besprochen herleiten und Tests aufsetzen
   - Zustandsräume sind lokal an Objekte gebunden. Initialisierung und Auswertung des Tests erfolgt deshalb am Objekt.

3. Test jeder Folge abhängiger Operationen in der gleichen Klasse.
   - Alle potenziellen Verwendungen einer Operation sollten unter allen praktisch relevanten Bedingungen ausprobiert werden.
   - In den Tests muss jede Objektausprägung (bzw. wenigstens Äquivalenzklassen von Ausprägungen) simuliert werden.
   - Existiert Objektlebenszyklus, dann Zustands- und Zustandsübergangs-Überdeckungstests
4. Anhand der Instrumentierung prüfen, wie die Testüberdeckung aussieht. Fehlende Überdeckungen durch zusätzliche Testfälle abdecken.
   - bereits beschriebenes klassisches Vorgehen

Testen abstrakter Klassen:

- Aus einer abstrakten Klasse muss eine konkrete Klasse gemacht werden
- bei der Realisierung abstrakter Operationen ist die leere oder eine einfache, die Spezifikation erfüllende Implementierung zu wählen.

Testen parametrisierter Klassen (Template-Klassen, C++):

- Zunächst eine möglichst einfache konkrete Klasse erzeugen
- Parameter so wählen, dass der Test möglichst einfach wird

Testen von Unterklassen:

Besondere Gesichtspunkte beim Testen von Unterklassen:

- Alle Testfälle für geerbte und **nicht redefinierte** Operationen der Oberklasse müssen erneut ausgewertet werden.
  - Unterklasse definiert neuen Kontext
- Für **redefinierte** Operationen sind vollständig neue strukturelle Testfälle zu erstellen.
  - redefinierte Operation hat neue Implementierung
- Für **redefinierte** Operationen müssen die alten funktionalen Testfälle ausgewertet und durch neue ergänzt werden.
  - Instanzen der Unterklasse sind spezielle Instanzen der Oberklasse
  - Zusätzlich muss die neue Semantik der redefinierten Operation getestet werden.

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
    }
```

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 Methodenrümpfe */
    runTest();
    tearDown();
}
```

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 */
```

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(); }
}
```

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();
}
```

Aufsammeln 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;
}
```

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
}
```

Anderer möglicher Lösungsansatz: Parametrisierte Klassen, wird erst von Java 5 unterstützt. Siehe Junit 4

Kann auch über Reflection und Stringmanipulation simuliert werden.

Composite Pattern - Ausführung mehrerer Tests „im Stück"

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.

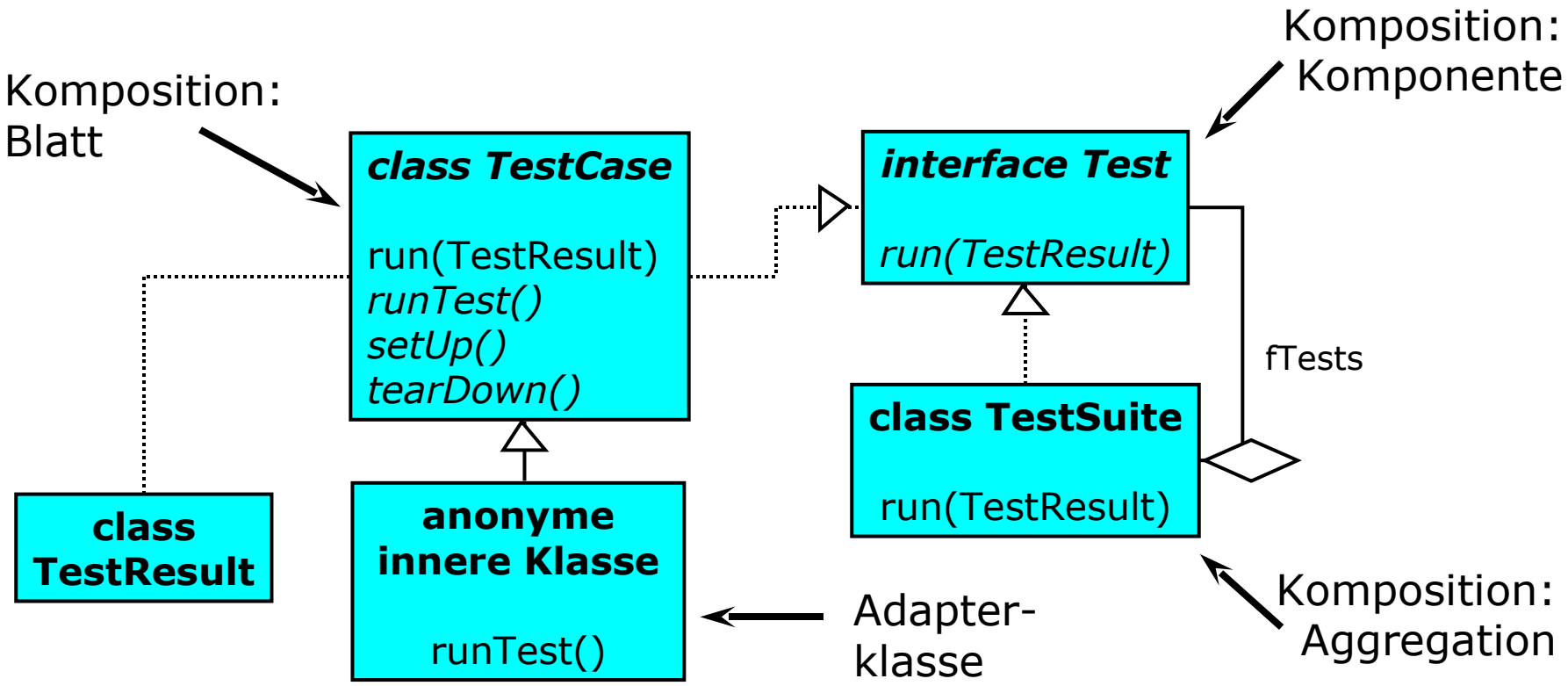
```
public interface Test {
    public void run (TestResult test);
}
public class TestSuite implements Test {
    private Vector fTests = new Vector();

    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);
    }
}
```

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

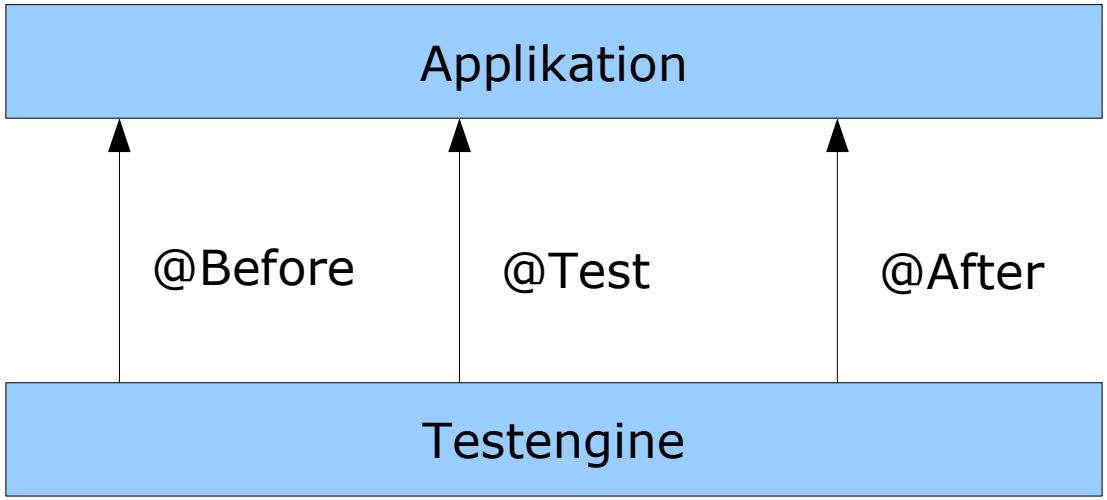
Alternative Lösung: Java sucht mit Reflection-Methoden nach Methoden mit entsprechendem Namen und fügt diese selbst zu einer TestSuite zusammen (dynamischer JUnit-Zugang).



UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Junit 4

- Neuer Zugang über Annotationen und Dependency Injection
- Annotationen als neues Ausdrucksmittel der Metaprogrammierung.
- Zugriff auf markierte Codeelemente über Reflection-Mechanismus
- JUnit 4 verwendet dieses Sprachkonstrukt, um beliebige Methoden beliebiger Klassen als ausführbaren Testfall kennzeichnen zu können.



Junit 4

- Kein Ableiten von TestCase mehr.
- Import static = Import der statischen Methoden einer Klasse in eine andere Klasse
- Testmethoden als @Test public void *beliebigerName*() ohne Parameter.

```
~~import junit.framework.TestCase;~~
import org.junit.Test;
import static org.junit.Assert.*;

public class EuroTest ~~extends TestCase~~ {
        @Test public void ~~test~~adding() {
                Euro two = new Euro(2.00);
                Euro sum = two.add(two);
                assertEquals("sum", new Euro(4.00), sum);
                assertEquals("two", new Euro(2.00), two);
        }
}
```

Junit 4

@Before und **@After**: Setup- und Teardown-Methoden werden wie Testfälle via Annotation gekennzeichnet

- @Before Methoden werden vor jedem Testfall ausgeführt
- @After Methoden nach jedem Testfall
- @Before Methoden der Oberklasse vor denen der Unterklasse
- @After Methoden der Unterklasse vor denen der Oberklasse

@BeforeClass und **@AfterClass**: Für kostspielige Test-Setups, die nicht für jeden einzelnen Testfall neu auf- und abgebaut werden können,

- @BeforeClass läuft für jede Testklasse nur ein einziges Mal und noch vor allen @Before Methoden,
- @AfterClass entsprechend für jede Testklasse nur einmal und zwar nach allen @After Methoden.
- Die so markierten Methoden müssen statisch sein

Junit 4

Erwartete Exceptions: Zum Testen von Exceptions kann der @Test Annotation mitgeteilt werden, dass die Ausführung Ihres Testfalls zu einer gezielten Exception führen soll:

```
public class EuroTest...
@Test(expected = IllegalArgumentException.class)
public void negativeAmount() {
        final double NEGATIVE_AMOUNT = -2.00;
        new Euro(NEGATIVE_AMOUNT); // should throw the exception
        }
}
```

Ähnlich mit anderen Parametern, etwa timeout