Software-Qualitätsmanagement

Kernfach Angewandte Informatik

Sommersemester 2004

Prof. Dr. Hans-Gert Gräbe

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.

6. Funktionale Testverfahren - Ergänzung

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

1. Herausfinden der **primitiven Operationen**, die in 1-1-Korrespondenz mit direkten Mausaktionen stehen.
2. 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.
3. Zu jeder Expansionssequenz wird ein **Zwischenoperator** konstruiert.
   - Bsp.: Edit, Cut, Paste (primitiv), Edit+Cut, Edit+Paste
   - Ansatz vermeidet Generieren von Testfällen nur für Edit.
4. Finde unter diesen die **abstrakten Operatoren**, die nach Aktivieren die GUI-Interaktion monopolisieren (Bsp: Edit+Preferences)

6. Funktionale Testverfahren - Ergänzung

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

- typische Struktur:

- Ansatz ist selbstähnlich, da Aktion dieselbe Struktur hat.
- 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

6. Funktionale Testverfahren - Ergänzung

5. **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.

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
}
```

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

7. 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 */
```

7. 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(); }
}
```

7. 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();
}
```

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;
}
```

7. 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
}
```

7. Aufbau eines Testwerkzeugs

Anderer möglicher Lösungsansatz: Parametrisierte Klassen, wird derzeit von Java aber nicht unterstützt.

Kann aber 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.

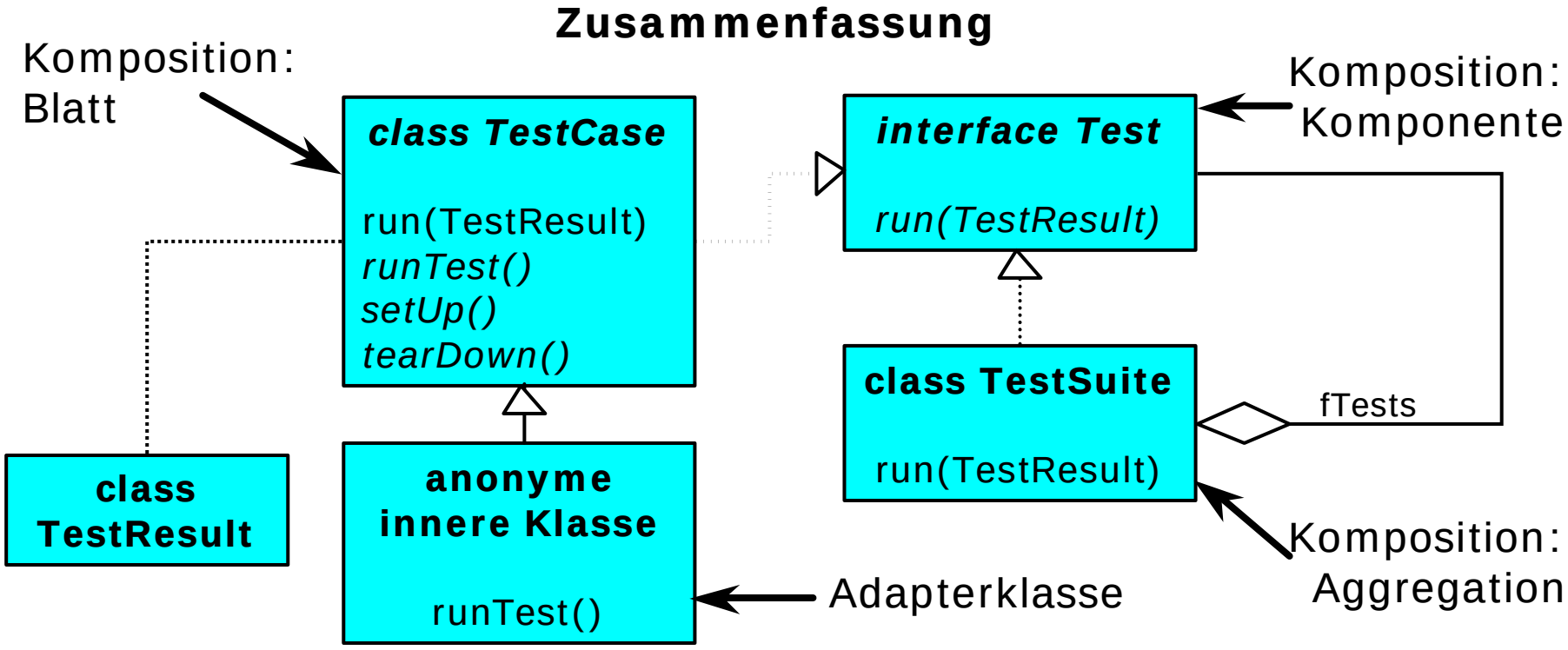
7. Aufbau eines Testwerkzeugs

```
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);
    }
}
```

7. 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).



Testmethodik

- Jede der Testmethoden hat ihre Vor- und Nachteile.
  - **Strukturtests** sind nicht in der Lage, fehlende oder fehlerhafte Funktionalität (Semantik) zu erkennen.
  - **Funktionstests** sind nicht in der Lage, Implementationsdetails zu prüfen (Syntax).
    - Spezifikation besitzt ein höheres Abstraktionsniveau als die Implementierung.
- Üblich sind deshalb Kombinationen der Verfahren zu einer stufenweise aufgebauten **Testmethodik.**

8. Testmethodik

- Anforderungen an eine stufenweise aufgebaute **Testmethodik**
  - Erfüllung anerkannter Minimalkriterien (Zweigüberdeckung, Funktionstest)
  - Fehlersensitive Testdaten
  - Testdaten für funktionelle Korrektheitsprüfung
  - Systematik, Wirtschaftlichkeit, Nachvollziehbarkeit
  - Verwendung geeigneter Werkzeuge
    - Protokoll, Testmetriken erstellen, Regressionstests

- Problem der Verteilung der Verantwortung zwischen Programmierer und Tester.
  - Test durch Programmierer: Eigenes Code-Review
    - Prinzip der frühen Fehlerentdeckung
    - Prinzip der entwicklungsbegleitenden QS
  - Test durch Testabteilung: Tests unter anderen als den Entwicklungsbedingungen möglich

8. Testmethodik

Verschiedene Sichten auf den Testprozess

- **Programmierer**: strukturelle Aspekte stehen im Vordergrund
  - **Frage**: Durchlaufen die Testdaten die richtigen Wege
  - **Aspekte**: Test von Programmteilen, Tests zu Optimierungszwecken
- **Endprüfung**: funktionale Aspekte stehen im Vordergrund
  - **Frage**: Ist der vereinbarte Leistungsumfang vollständig und korrekt realisiert?
  - **Aspekte**: alle Teilfunktionen, Abdecken des gesamten Eingabebereichs, Spezialfälle korrekt behandelt
- **Management**: organisatorische Aspekte stehen im Vordergrund
  - **Aspekte**: Abschätzung des Testaufwands, Beurteilung des Projektfortschritts, Beurteilung der Testgüte

Tests für Moduln mit hoher Kontrollflusskomplexität

Voraussetzungen: Testwerkzeug, Spezifikation, Quellcode

1. Funktionstest

- Testling wird instrumentiert
  - Ziel: Überdeckung im Hintergrund protokollieren
- Klassenbildung an Hand der Spezifikation, Auswahl von Grenzfällen, Spezialfällen und anderen Testfällen
  - Ohne Hinzuziehen des Quellcodes!
- Testfälle mit instrumentiertem Testling ausführen
  - Vergleich der Ist- mit den Soll-Werten

Ergebnis: Funktions- und Leistungsumfang wurde systematisch geprüft

8. Testmethodik

2. Strukturtest

- Die in Phase 1 erstellte Überdeckungsstatistik wird ausgewertet.
  - Ursachen für nicht überdeckte Anweisungen, Pfade, Bedingungen sind zu ermitteln
- Testfälle für die noch nicht durchlaufenen Zweige etc. erstellen bzw. diese entfernen, bis die geplante Überdeckungsrate erreicht ist.
- Problem gegen Ende:
  - Testsituation schwer herstellbar (etwa „Platte voll“)
  - Testfälle schwer herleitbar (relevanter Eingabebereich schwer zu bestimmen)
  - Für beide Fälle ist eine spezialisierte Testabteilung oft besser gerüstet

Ergebnis: Struktur wurde systematisch geprüft und optimiert

8. Testmethodik

3. Regressionstest

- Nochmaliger automatischer Durchlauf aller Testfälle mit Soll/Ist-Vergleich nach jeder Fehlerkorrektur.

Ergebnis: Konsistenz der Fehlerkorrekturen mit den bisherigen Testergebnissen wird sichergestellt.

- Hierfür ist der Einsatz eines entsprechenden Werkzeugs unerlässlich.

Erfahrungen zum Einsatz testender Verfahren

- Ohne Testwerkzeuge sowie konstruktive Voraussicht ist eine systematische, effektive und ökonomische Überprüfung der Implementierung nicht möglich.
- Funktionstest muss bereits im Entwurf berücksichtigt werden.
  - Tests sind bereits in der Entwurfsphase mit zu planen und in der Implementierungsphase parallel zum funktionalen Code zu implementieren.
    - Regel: a little test, a little code, ...
    - Im Moment des Codierens sind die Vorstellungen über das Funktionieren des Codes am detailliertesten. Die rechte Zeit, auch über Tests nachzudenken ...
    - Prinzip der frühzeitigen Fehlerentdeckung!
  - gute Regel ist: Tests zuerst schreiben.
    - M. Fowler: „Wenn du eine Print-Anweisung zum Debuggen schreiben willst, dann denke gleich über einen Testfall nach.“

8. Testmethodik

- Funktionstests bedürfen einer sorgfältigen Planung.
  - Tests sollten auch modelliert werden; Identifizieren gemeinsamer Testinitialisierungen (*setUp()*)
    - mehrfachen ähnlichen Testcode refaktorieren!
  - Alte Tests lauffähig zu halten ist fast so wichtig wie neue Tests zum Laufen zu bringen.
  - Tests sollten mehrmals an Tag laufen
    - z.B. in der Frühstückspause
- Auf zusätzlichen Strukturtest sollte nicht verzichtet werden.
  - Sinnvoll, wenn entscheidende Teile des Codes fertig sind, und auf der Basis der Ergebnisse der während der Funktionstests erstellten Metriken
- Integrationstests erst, wenn alle Systemkomponenten die Einzeltests bestanden haben.

8. Testmethodik

- Benutzerakzeptanz erfordert für die Tests einen hohen Automatisierungsgrad und komfortable Werkzeuge. (wie JUnit)
  - Verwendung integrativer konzeptioneller Testansätze
  - Nutzung vorhandener und bereits erstellter Testbibliotheken
    - Nach einiger Zeit ist Einfügen eines neuen Tests so einfach wie das Hinzufügen einer neuen Methode zu einer Klasse.
  - Mit Tests im Rücken ändert sich der Zugang zum Programmieren, da laufende Tests ein Gefühl von Sicherheit vermitteln.

- Die Testwerkzeuge müssen in die gesamte Entwicklungssystematik eingebettet sein und das Vorgehen durch entsprechende Testrichtlinien inhaltlich und methodisch untersetzt sein.

Literatur

[Balzert, Balzert, Liggesmeyer 93]

Balzert Helmut, Balzert Heide, Liggesmeyer P., *Systematisches Testen mit Tensor*, Mannheim: BI-Wissenschaftsverlag

[Girgis, Woodward 86]

Girgis M. R., Woodward M. R., *An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria*, in Proceedings Workshop on Software Testing, Banff, July 1986

[Memon, Pollack, Lou Soffa 99]

Memon, A.M., Pollack, M.E., Lou Soffa M., *Using a Goal-dirven Approach to generate test Cases for GUIs*, Proc. 21st Int. Conf. Software Engineering, ACM 1999

[JUnit]

Begleitdokumentation zu JUnit, siehe http://www.juint.org