

Grundzüge der Informatik

V13. Qualitätssicherung: Testen

Dr. Thomas Kühne
Praktische Informatik
TU Darmstadt



Agenda

Qualitätssicherung

Warum Qualitätssicherung?
Überblick auf die Möglichkeiten

Verifikation

Partielle Korrektheit
Totale Korrektheit

Testen

Strukturelles Testen
Funktionales Testen



Das Testdilemma

Ein Imageproblem

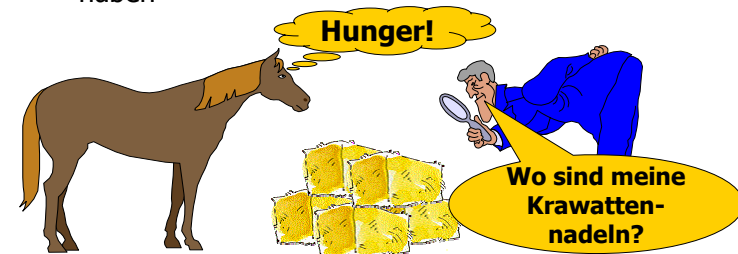
- In der Informatik haben Tester oft ein schlechtes Image
 - „Es hat nicht zum Programmierer gereicht“
- Zur Qualitätssicherung werden aber die Besten benötigt.
 - *Experten*-Gutachten (Tester, Prüfer, Sachverständiger)
- In anderen Disziplinen sind Personen, die Qualität auf nicht triviale Art und Weise sicherstellen, hoch angesehen
 - Softwarequalitätssicherung ist nicht trivial, daher verdient das systematische Testen von Software uneingeschränkte Anerkennung



Das Testdilemma

Warum wir testen müssen...

- Solange Software von menschlichen Gehirnen erstellt wird, wird sie i.A. fehlerhaft sein.
- Fehler müssen gefunden werden bevor sie Folgen haben



Das Testdilemma

Wie intensiv müssen wir testen?

- Wie kann man gründlich suchen?
- Ab wann kann man davon ausgehen, daß das mit hoher Wahrscheinlichkeit keine „Nadeln“ mehr enthalten sind?



Das Testdilemma

Was ist die Aufgabe von Tests?

- Nur das Aufspüren von Fehlern!
- Die Ursachenforschung und die letztendliche Beseitigung (Debugging) ist eine andere Disziplin



Das Testdilemma

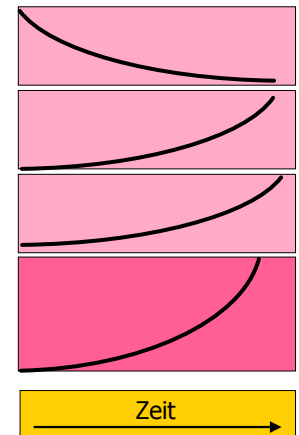
*If something can go wrong, it will— at the worst possible moment.
If nothing can go wrong, it still will.
If nothing has gone wrong, you have overlooked something.*

Murphy's Law

Bills These: Jedes funktionierende Computergrafikprogramm enthält eine gerade Anzahl von Vorzeichenfehlern

Wert & Kosten des Testens

- Entdeckungsrate
 - Aufspüren wird immer schwieriger
- Gefundene Fehler / Tests
 - Tests immer aufwendiger
- Fehlerqualität
 - Fehler immer hartnäckiger
- Kombination
 - Kosten pro gefundenen Fehler steigen über-proportional
 - Aufhören wenn es zu schwierig/teuer wird weitere Fehler zu finden?



Notwendigkeit einer Systematik

- In der **Praxis** relevante Gründe mit dem Testen aufzuhören sind:
 - Zeit aufgebraucht; der Kunde möchte das Produkt
 - Alle vorhanden Tests werden bestanden
- Daher ist es notwendig **objektive Kriterien** dafür zu haben ob noch zusätzliche Testfälle benötigt werden
 - Systematische Verfahren benötigt
 - => Testmethoden

Testmethode

Planmäßiges, auf ein Regelwerk aufbauendes Testverfahren.

Im Allgemeinen eine systematische Methode zur Auswahl und/oder Generierung von Tests.

- Strukturelle Testmethoden
- Funktionale Testmethoden

Strukturelle Methoden

Basieren auf der Anwendungsstruktur (White Box Tests)

- Testfälle und Testdaten werden nur aus der Struktur des Prüfgegenstands abgeleitet
- Die Vollständigkeit der Prüfung wird anhand von Strukturelementen (Anweisungen, Zweige, Daten) bewertet
- Die Spezifikation des Prüfgegenstands wird bei der Beurteilung der Vollständigkeit der Tests nicht beachtet

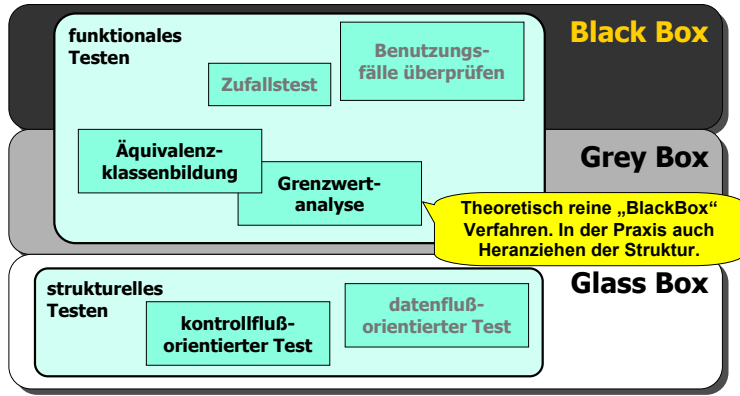
Funktionale Methoden

Basieren auf Anforderungen (Black Box Tests)

- Testfälle und Testdaten werden aus der funktionalen Spezifikation des Prüfgegenstands abgeleitet
- Vollständigkeit der Prüfung wird anhand der Spezifikation bewertet
- Im Idealfall sollte eine *formale* Spezifikation vorliegen
- Struktur des Prüfgegenstands wird nicht ausgewertet

Testklassifizierung

Klassifikation nach benötigter Information



Grundbegriffe

- Testfall
 - Besteht aus einem Satz von Testdaten, der die vollständige Ausführung eines zu testenden Programms bewirkt und dem Sollresultat.
 - Testsuite
 - Eine nicht leere Menge von Testfällen, die nach einem bestimmten Kriterium ausgewählt wurden
 - Testenkriterium
 - legt fest, welches Minimalziel durch die Testfälle erreicht werden soll => Testabdeckung
- Das Anfangs geforderte Kriterium für die Auswahl von Testfällen!

Testabdeckung

- Verhältnis zwischen tatsächlich ausgeführten Testfällen und theoretisch existierenden Testfällen

$$TAB = \frac{\text{ausgeführte Testfälle}}{\text{existierende Testfälle}}$$

- Bei einer Testabdeckung von 100% spricht man von einem erschöpfendem Test
- Erschöpfendes Testen ist meist weder praktikabel noch sinnvoll

Strukturelles Testen

Realisierung von Überdeckungstests

- Einfügen von Trace Statements in den Quellcode (*Instrumentieren*)
- Bei der Ausführung erzeugen die Trace Statements ein Logbuch
- Auswertung des Logbuchs zur Erstellung eines Abdeckungsberichts

Die im folgenden beschriebenen Verfahren sind nur mit entsprechender Werkzeugunterstützung sinnvoll anwendbar

Instrumentierung

- Mitprotokollieren, welche Teile des Prüflings bei der Ausführung durchlaufen wurden

```
int bestimmeMax (int A, int B)
{
  if (A > B)
  {
    ThenBranch = true;
    return A;
  }
  else
  {
    ElseBranch = true;
    return B;
  }
  ...
}
```

In den Quellcode eingefügte „Zähler“

- Nach dem Testlauf werden die Zählerstände ausgewertet

Kontrollfluß Testen

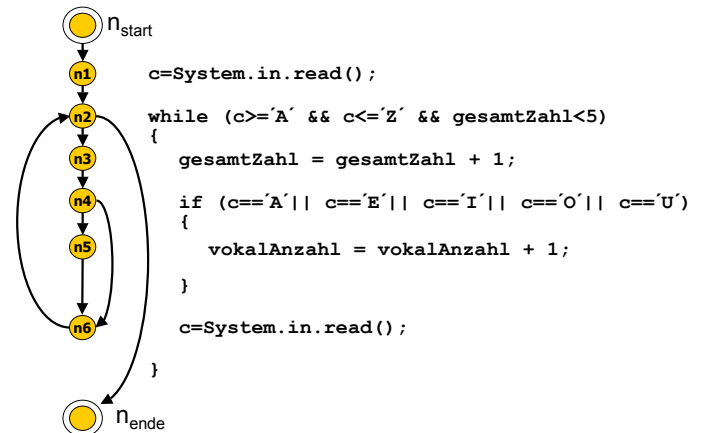
- Kontrollflußgraph
 - Der Kontrollfluß wird durch einen gerichteten Graphen (Kontrollflußgraph) dargestellt
 - Jeder Knoten stellt eine Anweisung dar
 - Die Knoten sind durch gerichtete Kanten verbunden
 - Kantenfolgen beschreiben einen möglichen Kontrollfluß von Knoten i zu Knoten j
- Zweige
 - Gerichtete Kanten
- Pfad
 - Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit einem Endknoten endet

Kontrollflußgraphen

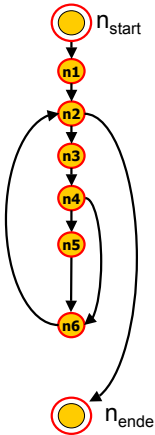
Definition und Terminologie

- Kontrollflußgraphen beinhalten grundsätzlich einen Start- und einen Endknoten
- Der Endknoten hat den Ausgangsgrad 0 und jeder normale Knoten liegt auf einem Pfad vom Startknoten zum Endknoten
- Knoten mit Ausgangsgrad 1 heißen Anweisungsknoten
- Alle anderen Knoten außer dem Endknoten heißen Prädikatknöten

Kontrollflußgraphen (Beispiel)



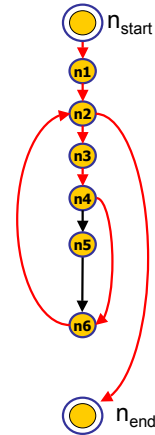
Überdeckungstests



Anweisungsüberdeckung (C0)

- Testfälle
 - <A, #>
- Hilft, toten Code zu finden
- Ist notwendig aber nicht ausreichend
- Besitzt wenig praktische Relevanz

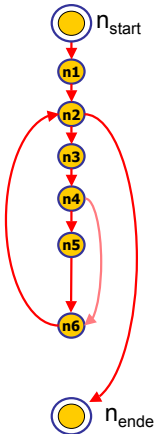
Überdeckungstests



Zweigüberdeckung (C1)

- Testfälle
 - <B, #>

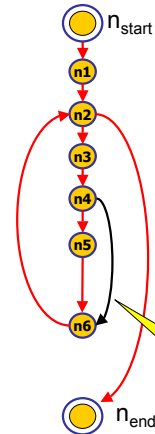
Überdeckungstests



Zweigüberdeckung (C1)

- Testfälle
 - <B, #> <A, #>
- Hilft, nicht ausführbare Zweige zu finden
- Anweisungsüberdeckung vollständig enthalten
- Berücksichtigt weder Kombination von Zweigen noch komplexe Bedingungen
- Gilt als das minimale Testkriterium

Überdeckungstests



Bedingungsüberdeckung (C2)

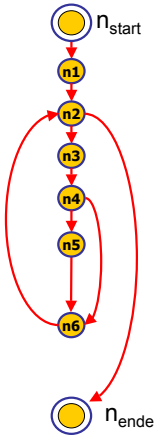
- Testfälle
 - <A, E, I, O, U, @> <€>
- Enthält weder Anweisungs- noch Zweigüberdeckung
- Da beides minimale Testkriterien sind, ist eine alleinige einfache Bedingungsüberdeckung nicht ausreichend

alle atomaren Bedingungen mind. einmal true & false

Der „else“ Zweig wird nie beschriftet



Überdeckungstests



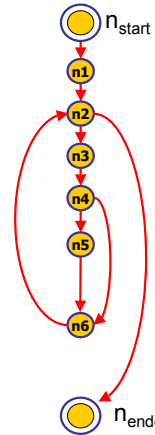
Mehrfach-Bedingungsüberdeckung

alle Kombinationen von atomaren Bedingungen gefordert

- Testfälle
 - <A, E, I, O, U, @>
 - <A, E, I, O, U, €>
 - <A, E, I, O, U, A>
 - <A, @>, <A, €>
- Zweigüberdeckung enthalten
- Kombinationsexplosion: N atomare Bedingungen => 2N Möglichkeiten
- Kombinationen teilweise unmöglich!



Überdeckungstests



Pfadüberdeckung (C7)

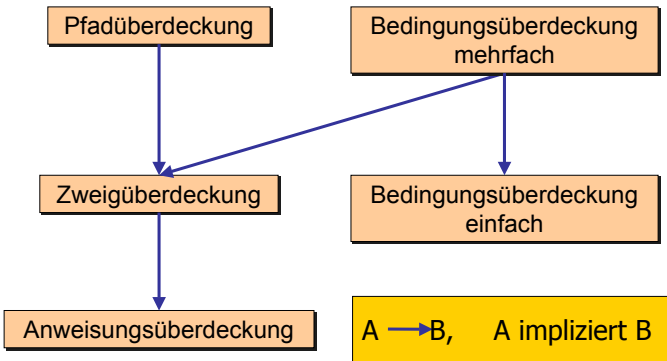
alle Kombinationen von Zweigen gefordert

- Testfälle
 - <#>, <B, #>, <A, #>
 - <A, B, B, B, B>
 - <B, A, B, B, B>, ...
- Zweigüberdeckung enthalten
- Schleifen führen praktisch zu unendlichen vielen Testfällen
- Höchste Erfolgsaussichten, aber völlig unpraktikabel



Überdeckungstests

Übersicht...



Überdeckungstests

Vorteile

- Defizite in der Teststrategie aufdecken
- Nicht erreichbaren Code entdecken
- Anhaltspunkt für den Testfortschritt

Nachteile

Produkt wird gegen sich selbst geprüft

- 100% Abdeckung nicht praktikabel
- Codeabdeckung ist kein Kriterium für vollständiges Testen
- Fehlende Funktionalität wird nicht erkannt

Funktionales Testen

- Es ist unzureichend, ein Programm lediglich gegen sich selbst zu testen
 - => Testfälle aus der **Programmspezifikation** ableiten
 - Programmstruktur wird nicht betrachtet

Ziel

- Möglichst umfassende, aber redundanzarme Prüfung der spezifizierten Funktionalität
- Überprüfung aller Programmfunktionen (**Funktionsüberdeckung**)

Funktionales Testen

- Hauptschwierigkeiten
 - Ableitung der geeigneten Testfälle
 - Vollständiger Funktionstest ist im allgemeinen nicht durchführbar
- Ziel der Testplanung
 - Testfälle so auswählen, daß die Wahrscheinlichkeit groß ist, Fehler zu finden
- Testfallbestimmung:
 - Funktionale Äquivalenzklassenbildung
 - Grenzwertanalyse

Äquivalenzklassenbildung

- **Motivation:** Die Anzahl von Testfällen wird sinnvoll eingeschränkt indem die Testdaten in Äquivalenzklassen zerlegt werden
- **Annahme:** Für jeden Repräsentanten aus einer Äquivalenzklasse reagiert das Programm auf die gleiche Weise
- **Beispiele** Annahme: Jeder Fall wird gleich behandelt
 - Klassen: **Vokal**, **Konsonant**, **kein Buchstabe**
 - Repräs.: E T %
 - Klassen: $x < 0$, $0 \leq x \leq 9$, $x > 9$
 - Repräs.: -5 5 20

Grenzwertanalyse

- **Motivation:** Fehler treten typischerweise bei Extremwerten bzw. an den „Rändern“ von Äquivalenzklassen auf
- **Annahme:** Die „extremen“ Repräsentanten sind nicht nur ebenso wirksam wie „normale“, sondern darüber hinaus besonders geeignet
- **Beispiele** Annahme: Jeder Fall wird gesondert behandelt
 - Klassen: **Vokal**, **Konsonant**, **kein Buchstabe**
 - Repräs.: A, I, ..., U B, Z '0', @, [, '0377'
 - Klassen: $x < 0$, $0 \leq x \leq 9$, $x > 9$
 - Repräs.: -1 0,9 10

Testklassifizierung

Stärken und Schwächen

- Strukturelles Testen
 - findet: Abbruchfehler, unerreichbare Zweige, Endlosschleifen, inkonsistente Bedingungen
=> Prüfung gegen sich selbst
- Funktionales Testen
 - findet: falsche Antworten
=> Prüfung gegen Spezifikation

Kombinierte Strategie

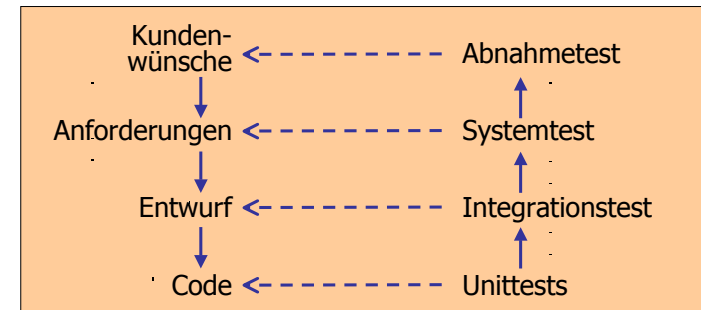
- Die Nachteile von strukturellen Verfahren...
 - nicht implementierte Funktionen werden nicht aufgespürt
 - das Ziel der Überdeckung produziert oft bzgl. der Funktionalität triviale Testfälle
- ... und die Nachteile von funktionalen Verfahren...
 - Achillesfersen der Implementierung werden nicht herangezogen
 - typischerweise höchstens 70% Zweigüberdeckung
- ...komplementieren sich gegenseitig
=> kombinierter Ansatz

Kombinierte Strategie

- Zunächst Funktionstest...
 - Anhand der Spezifikation Äquivalenzklassen bilden
 - Grenzwerte ermitteln
 - Testfälle erstellen
 - => Funktionsumfang systematisch geprüft
- ...anschließend Strukturtest
 - Oben erzielte Überdeckung analysieren
 - Nicht benutzte Bedingungen / Pfade identifizieren
 - Gewünschte Überdeckung erzielen
 - => Sicherstellung der Robustheit & Validieren der Spezifikation

Teststadien

Entwicklungsphasen mit zugehörigen Tests



A → B A wird vor B ausgeführt
X -> Y X findet Fehler in Y

V13 - 41 GDI1 - Qualitätssicherung

JUnit: Benutzungsschnittstelle

Fehlschlagende Testfälle

Detaillierte Meldungen

Erfolg (bzgl. der Testfälle!)

V13 - 42 GDI1 - Qualitätssicherung

JUnit: Benutzung

Testklasse für eine Währungsklasse (Money)

```
public class MoneyTest extends TestCase {
    private Money f12CHF, f14CHF;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }

    public void testSimpleAdd() {
        Money expected= new Money(26, "CHF");
        Money result= f12CHF.add(f14CHF);
        assert(expected.equals(result));
    }
    ...
}
```

Erzeugen von Testdaten

Ein Testfall

V13 - 43 GDI1 - Qualitätssicherung

JUnit: Benutzung

Testklasse für eine Währungsklasse (fortg.)

```
...
public void testEquals() {
    assert(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(m12CHF, new Money(12, "CHF"));
    assert(!f12CHF.equals(f14CHF));
}

public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
}
```

Noch ein Testfall

Eine Testsuite

V13 - 44 GDI1 - Qualitätssicherung

Nochmal: Das Imageproblem

Negative Stimmen

- Testen sei nicht kreativ
 - Produkt wird nicht mitgestaltet
 - Testprozedur starr festgelegt

Positive Stimmen

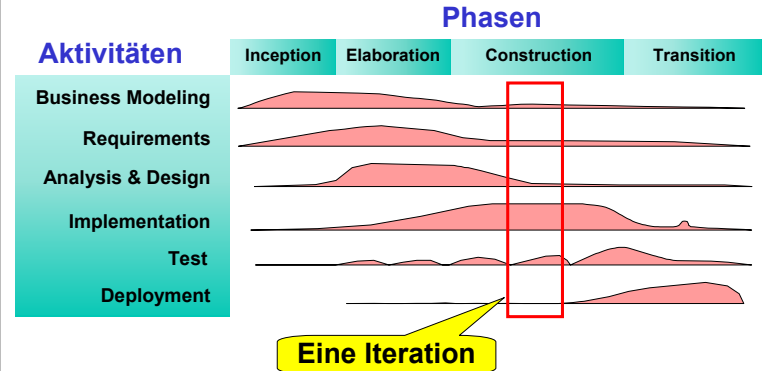
- Testen ist eine anerkannte Herausforderung
 - Geschicktes Vorgehen notwendig, um ein System auf hohem fachlichen und technischen Niveau mit wirksamen Testfällen zu bestücken
 - Dramatische Kosteneinsparung durch frühe Validierung

Testen: Zusammenfassung

- In der Praxis unverzichtbar
 - je früher Fehler gefunden werden desto besser (kostengünstiger)
 - Entwicklung ist deshalb typischerweise ein iterativer, rückgekoppelter, Prozeß
- Qualität nicht nur überprüfen sondern, wenn möglich, „hineinkonstruieren“
 - systematische Verfahren
 - => Software Engineering

Testen: Verschränkter Ansatz

„Validiere früh, validiere oft“



Qualitätssicherungsbegriffe

- Verifikation
 - Beweis, daß ein Produkt im Sinne der Spezifikation korrekt ist (funktionaler Test als grobe Annäherung)
- Validierung
 - Nachweis, daß ein Produkt in einer bestimmten Zielumgebung lauffähig ist und das tut, was der Benutzer wünscht (strukturelle Tests als „Belastungstests“)

Ein verifiziertes Programm muß nicht den Wünschen entsprechen und ein validiertes muß nicht korrekt im Sinne der Spezifikation sein

Qualitätssicherung

Die „Kapitulationserklärung“ der Softwaretechnik

- Es gibt vier Faktoren bei der Softwareentwicklung
- Der Kunde darf drei Faktoren priorisieren
- Der vierte Faktor ergibt sich aus dieser Wahl!
- Qualität darf nicht zum Stiefkind werden!

