Universität Zürich[UZH]

Institut für Informatik

Martin Glinz

Software-Qualität – Ausgewählte Kapitel

Kapitel 3

Fortgeschrittene Testverfahren

3.1 Übersicht (Wiederholung aus Grundvorlesung)

- Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden. (Myers 1979)
- Wurde ein Programm sorgfältig getestet (und sind alle gefundenen Fehler korrigiert), so steigt die Wahrscheinlichkeit, dass das Programm sich auch in den nicht getesteten Fällen wunschgemäß verhält
- Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden.

  Grund: alle Kombinationen aller möglichen Werte der Eingabedaten müssten getestet werden

Test und Testvorgaben

- Testen setzt voraus, dass die erwarteten Ergebnisse bekannt sind
  - Entweder muss gegen eine Spezifikation
  - oder gegen vorhandene Testergebnisse (z.B. bei der Wiederholung von Tests nach Programm-Modifikationen) getestet werden (so genannter Regressionstest)

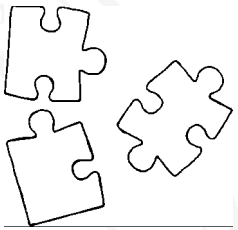
- Unvorbereitete und undokumentierte Tests sind sinnlos

Testsystematik: Test ist nicht gleich Test

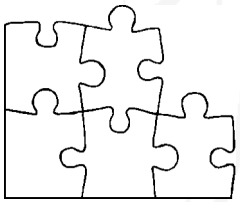
- Laufversuch: Der Entwickler „testet“
- Wegwerf-Test: Jemand testet, aber ohne System
- Systematischer Test: Spezialisten testen
  - Test ist geplant, Testvorschrift ist vorgängig erstellt
  - Programm wird gemäß Testvorschrift ausgeführt
  - Ist-Resultate werden mit Soll-Resultaten verglichen
  - Fehlersuche und -behebung erfolgen separat
  - Nicht bestandene Tests werden wiederholt
  - Testergebnisse werden dokumentiert
  - Test endet, wenn vorher definierte Testziele erreicht sind

Testgegenstand und Testarten

- Testgegenstand sind Komponenten, Teilsysteme oder Systeme
- Komponententest, Modultest (Unit Test)



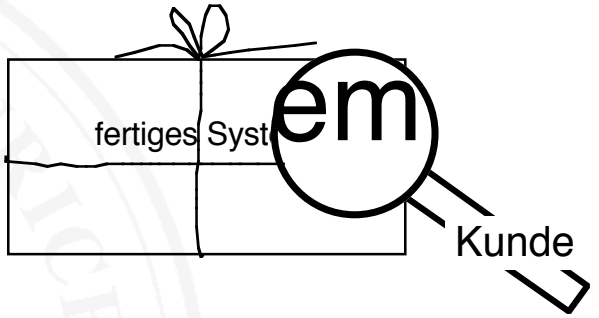
- Integrationstest (Integration Test)



- Systemtest (System Test)

Testgegenstand und Testarten – 2

- Abnahmetest (acceptance test)
  - eine besondere Form des Tests:
  - nicht: Fehler finden
  - sondern: zeigen, dass das System die gestellten Anforderungen erfüllt, d.h. in allen getesteten Fällen fehlerfrei arbeitet.



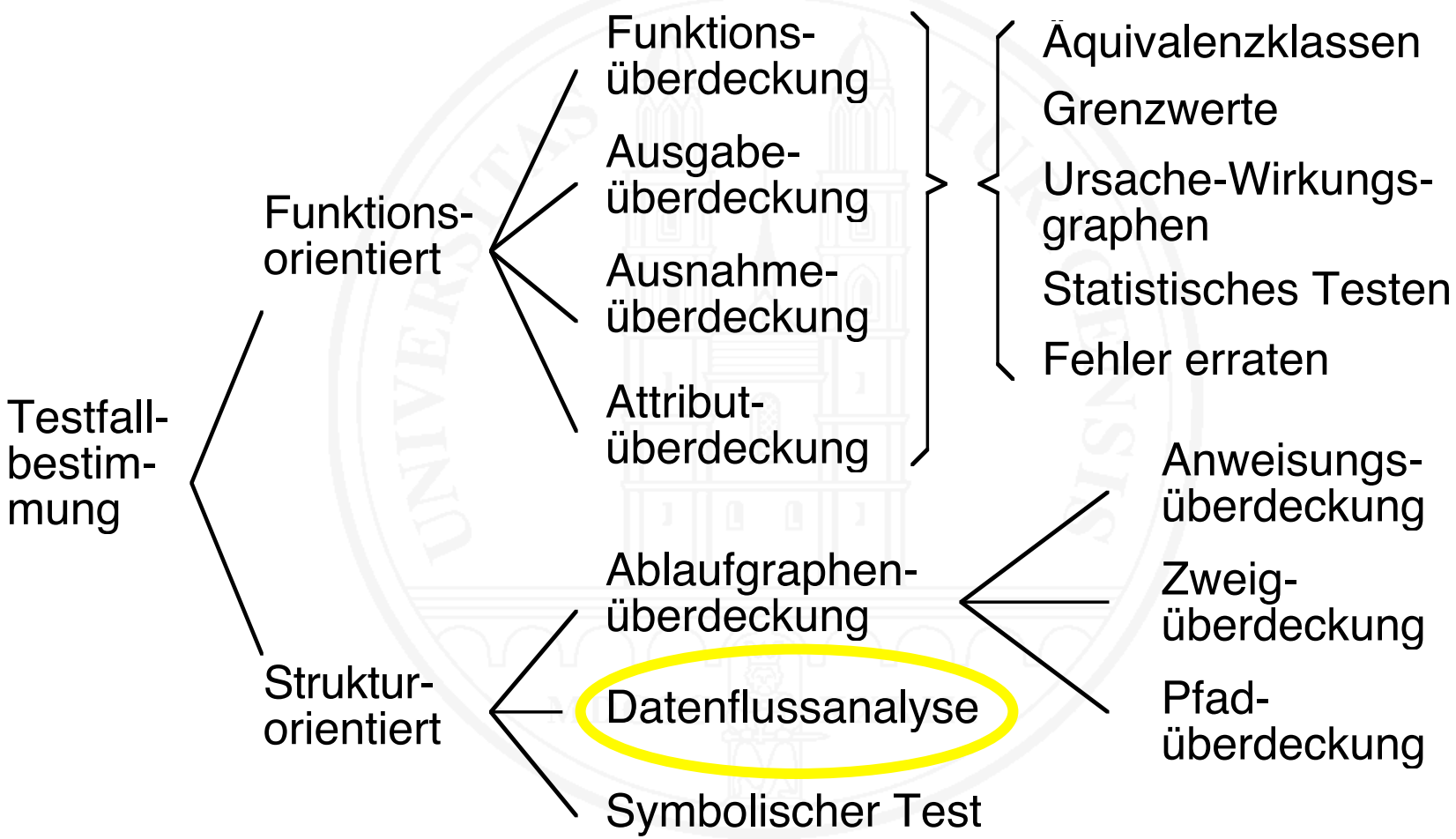
Testablauf

- Planung
  - Teststrategie: was - wann - wie - wie lange
  - Einbettung des Testens in die Entwicklungsplanung:
    - welche Dokumente sind zu erstellen
    - Termine und Kosten für Testvorbereitung, Testdurchführung und Testauswertung
  - Wer testet
- Vorbereitung
  - Auswahl der Testfälle
  - Bereitstellen der Testumgebung
  - Erstellung der Testvorschrift

Testablauf – 2

- Durchführung
  - Testumgebung einrichten
  - Testfälle nach Testvorschrift ausführen
  - Ergebnisse notieren
  - Prüfling während des Tests nicht verändern
- Auswertung
  - Testbefunde zusammenstellen
- Fehlerbehebung (ist nicht Bestandteil des Tests!)
  - gefundene Fehler(symptome) analysieren
  - Fehlerursachen bestimmen (Debugging)
  - Fehler beheben

Bestimmen von Testfällen



3.2 Datenflussorientiertes Testen

- Ein strukturorientierter Test
- Basiert auf einer Analyse der Datenflüsse in einem Programm
  - Bestimmung des Steuerflussgraphen
  - Annotierung des Steuerflussgraphen:
    - Wo wird eine Variable verändert?
    - Wo geht eine Variable in eine Berechnung ein?
    - Wo ist eine Variable Bestandteil einer Bedingung?
- Güte wird über verschiedene Überdeckungsgrade definiert
- Kann auch zur Beurteilung der Güte eines funktionsorientierten Tests herangezogen werden.

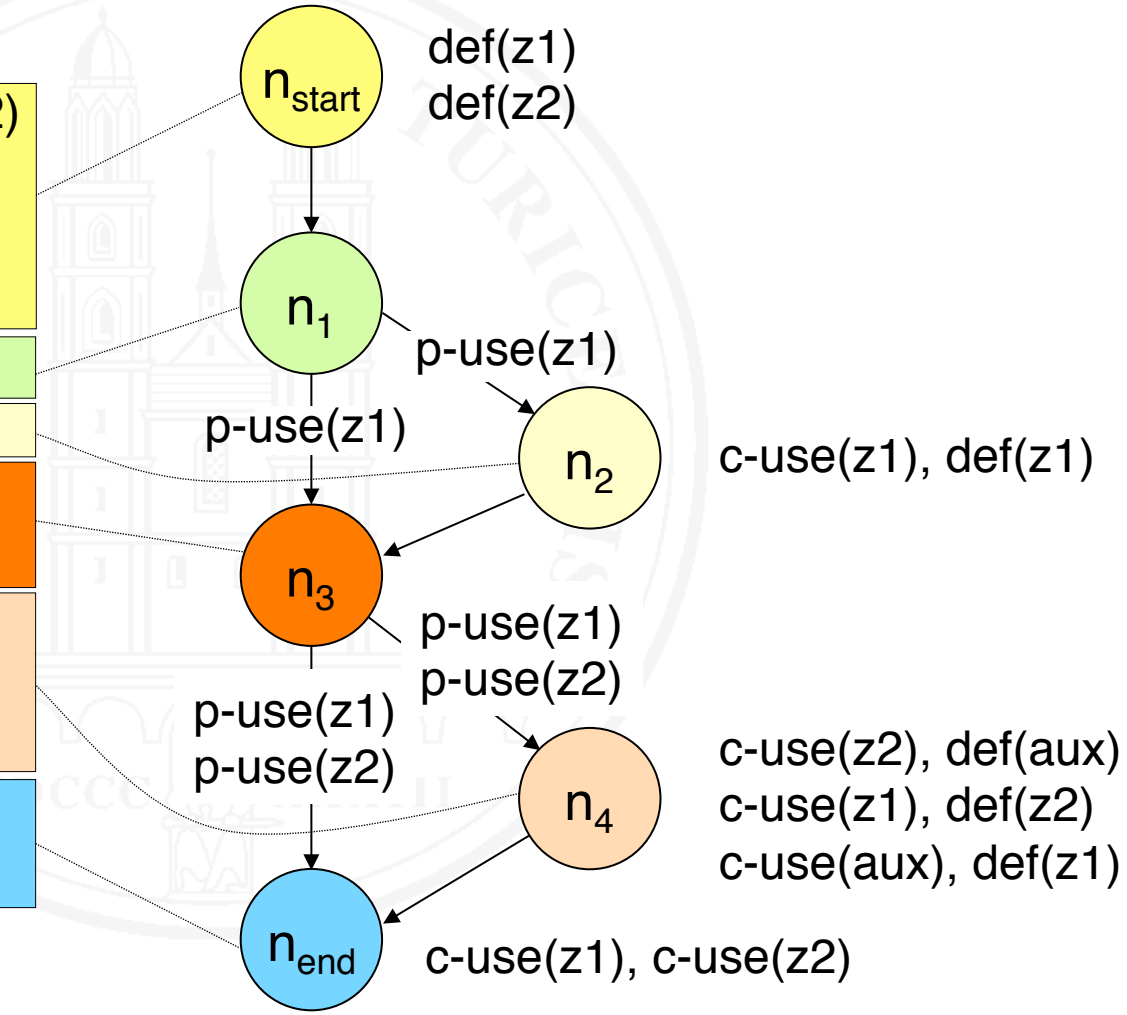
Beispiel

def()	Variable geschrieben
c-use()	berechnend benutzt
p-use()	prädikativ benutzt

Ein kleines Programm in C:

```
void SortAbs1 (int& z1, int& z2)
/* Sorts the set {|z1|, z2} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux = z2;
        z2 = z1;
        z1 = aux;
    }
}
```

Annotierter Steuerflussgraph:

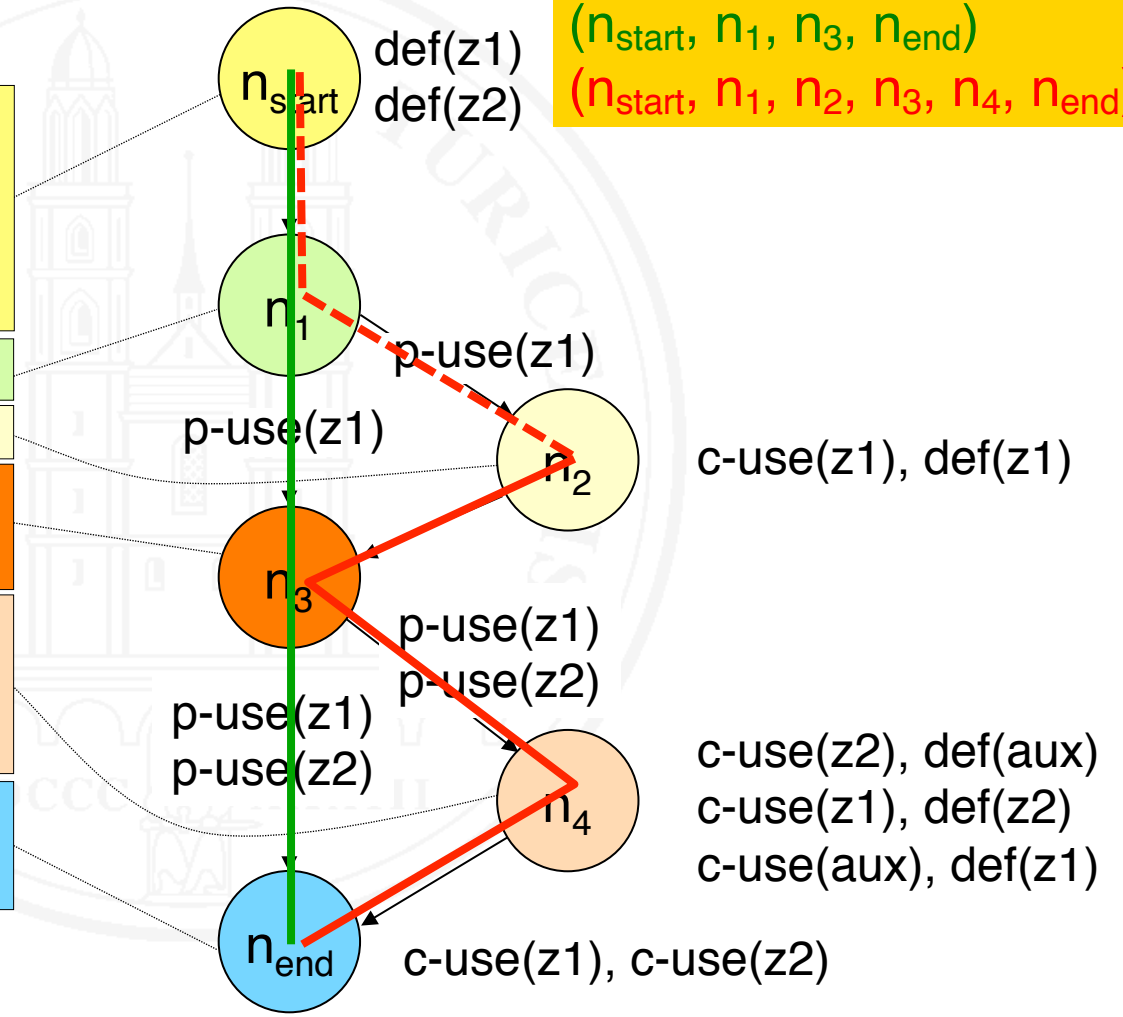


Ableitung von Testfällen

- Ein Pfad (n_n, ..., n_m) in einem Steuerflussgraphen heißt definitionsfrei bezüglich der Variablen x, wenn
  - def(x) im Knoten n_n
  - c-use(x) im Knoten n_m oder p-use(x) auf der Kante (n_{m-1},n_m)
  - Zwischen der Definition von x in n_n und der Benutzung in n_m oder auf der Kante (n_{m-1},n_m) erfolgt keine weitere Definition von x
- Testfälle werden so definiert, dass das Programm definitionsfreie Pfade einer bestimmten Überdeckungsklasse durchläuft, zum Beispiel:
  - *all defs*-Kriterium: Teste für jede Definition jeder Variablen x mindestens einen definitionsfreien Pfad zu einer Benutzung von x
  - *all p-uses*-Kriterium: Teste für jede Definition jeder Variablen x alle definitionsfreien Pfade zu jeder p-Benutzung von x
  - *all c-uses*-Kriterium: Teste für jede Definition jeder Variablen x alle definitionsfreien Pfade zu jeder c-Benutzung von x

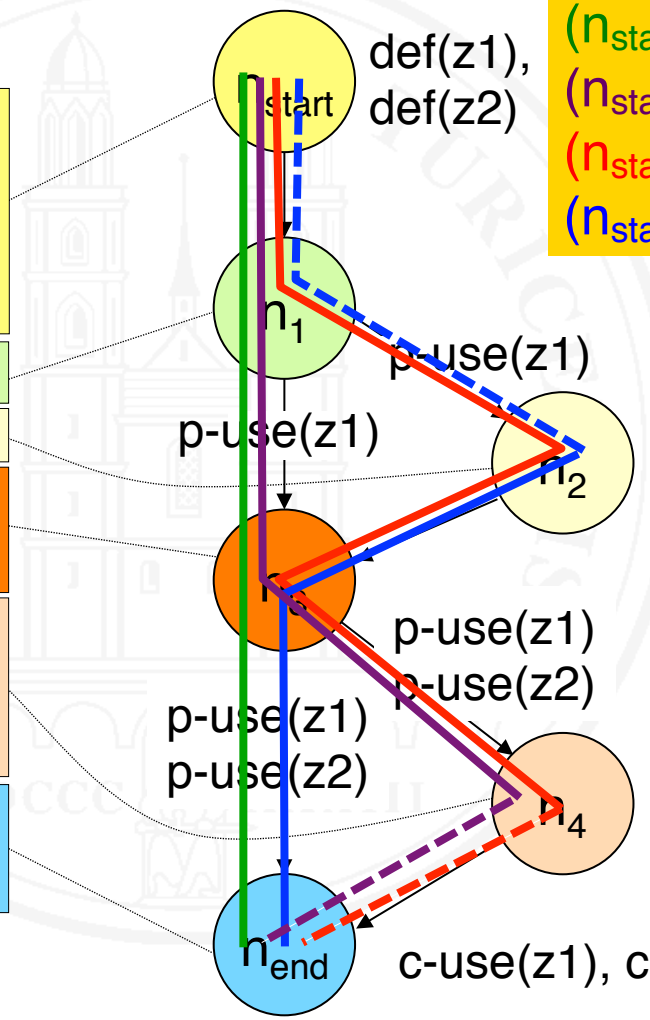
Ableitung von Testfällen – Beispiel 1: all-defs

```
void SortAbs1 (int& z1, int& z2)
/* Sorts the set {|z1|, z2} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux= z2;
        z2 = z1;
        z1 = aux;
    }
}
```



Ableitung von Testfällen – Beispiel 1: all-p-uses

```
void SortAbs1 (int& z1, int& z2)
/* Sorts the set {|z1|, z2} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux= z2;
        z2 = z1;
        z1 = aux;
    }
}
```



all-p-uses:
(n_{start}, n_1, n_3, n_{end})
(n_{start}, n_1, n_3, n_4, n_{end})
(n_{start}, n_1, n_2, n_3, n_4, n_{end})
(n_{start}, n_1, n_2, n_3, n_{end})

c-use(z1), def(z1)

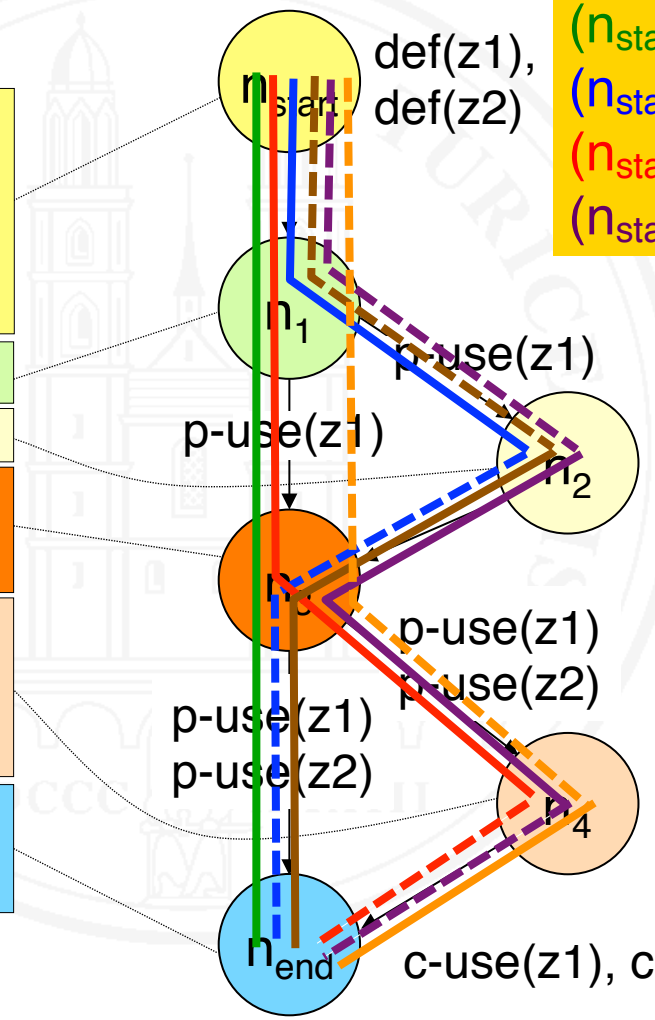
Hinweis: *all-p-uses* impliziert *Zweigüberdeckung*. Weshalb?

c-use(z2), def(aux)
c-use(z1), def(z2)
c-use(aux), def(z1)

c-use(z1), c-use(z2)

Ableitung von Testfällen – Beispiel 1: all-c-uses

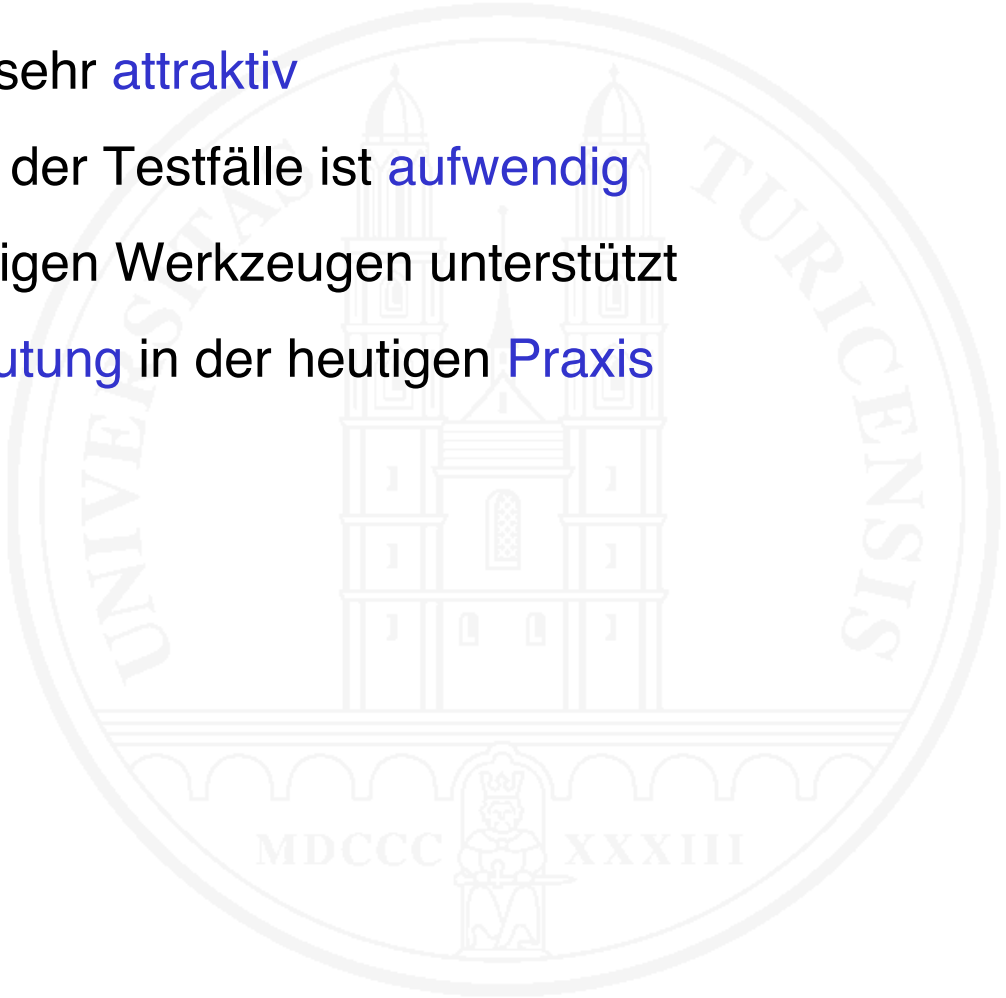
```
void SortAbs1 (int& z1, int& z2)
/* Sorts the set {|z1|, z2} */
{
    int aux;
    if (z1 < 0) {
        z1 = -z1;
    }
    if (z1 > z2) {
        aux= z2;
        z2 = z1;
        z1 = aux;
    }
}
```



all-c-uses:
(n_{start}, n_1, n_3, n_{end})
(n_{start}, n_1, n_2, n_3, n_{end})
(n_{start}, n_1, n_3, n_4, n_{end})
(n_{start}, n_1, n_2, n_3, n_4, n_{end})

Stellenwert des datenflussorientierten Testens

- Theoretisch sehr attraktiv
- Bestimmung der Testfälle ist aufwendig
- Nur von wenigen Werkzeugen unterstützt
- Wenig Bedeutung in der heutigen Praxis



3.3 Anwendungsfallorientiertes Testen

- Definition von Testfällen auf der Basis eines Anwendungsfallmodells
- Gehört zur Familie der funktionsorientierten Tests
- Ziel:Überdeckung aller Anwendungsfälle
- Pro Anwendungsfall
  - mindestens ein Testfall für den Normalablauf
  - mindestens ein Testfall für jeden möglichen alternativen Ablauf
- Eignet sich insbesondere für die Erstellung von Abnahmetests

Aufgabe: Testfälle ermitteln

Ermitteln Sie Testfälle für den nachstehenden Anwendungsfall:

Buch Ausleihen

Akteur(e): Benutzerin

Auslöser: Eine Benutzerin bringt ein Buch oder mehrere Bücher, das/die sie ausleihen möchte, zum Ausleiheschalter

Normalablauf:

1. Ausweiskarte der Benutzerin lesen und Angaben überprüfen
2. Signatur eines Buchs lesen und zugehörigen Katalogeintrag ermitteln
3. Ausleihe registrieren und Diebstahlsicherungsetikett deaktivieren
4. Wenn mehrere Bücher auszuleihen sind, mit den weiteren Büchern nach 2. und 3. verfahren
5. Leihschein drucken für alle ausgeliehenen Bücher
6. Der Benutzerin Bücher aushändigen, Vorgang abschließen

Aufgabe: Testfälle ermitteln – 2

Alternative Abläufe:

1.1 Ausweiskarte nicht vorhanden oder gelesene Ausweiskarte ist ungültig: Vorgang abbrechen

2.1 Buch ist vorgemerkt für andere Person: Buch zur Seite legen, mit Schritt 4 fortfahren

2.2 Benutzerin hat mehr als ein überfälliges Buch nicht zurückgebracht: Vorgang abbrechen

3.4 Paarweises Testen

- Problem: Programme mit vielfachen Kombinationsmöglichkeiten von Eingaben
- Im Prinzip müssten alle Kombinationen getestet werden
- ⇨ Zahl der benötigten Testfälle wächst exponentiell: nicht machbar
- Erfahrung: In den meisten Fällen genügt es, alle Paare von Eingabedaten zu testen, um Datenkombinationsfehler zu erkennen
- Die Zahl der benötigten Testfälle für paarweises Testen wächst nur logarithmisch: auch für größere Anzahl von Kombinationen testbar

$n = O(m^2 \log_2 k)$

k	Anzahl Eingabefelder
m	Anzahl Testdaten pro Eingabefeld
n	Benötigte Testfälle für paarweises Testen

Beispiel:

- 13 Eingabefelder (k= 13) mit je drei Werten (m=3)
- Ein Test aller Kombinationen erfordert 3^{13} = 1 594 323 Testfälle
- Für vollständiges paarweises Testen genügen 15 Testfälle:

	P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	*P13*
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	0	0	0	0
3	2	2	2	2	2	2	2	2	2	0	0	0	0
4	0	0	0	1	1	1	2	2	2	1	1	1	0
5	1	1	1	2	2	2	0	0	0	1	1	1	0
6	2	2	2	0	0	0	1	1	1	1	1	1	0
7	0	0	0	2	2	2	1	1	1	2	2	2	0
8	2	2	2	1	1	1	0	0	0	2	2	2	0
9	1	1	1	0	0	0	2	2	2	2	2	2	0
10	0	1	2	0	1	2	0	1	2	0	1	2	1
11	1	2	0	1	2	0	1	2	0	1	2	0	1
12	2	0	1	2	0	1	2	0	1	2	0	1	1
13	0	2	1	0	2	1	0	2	1	0	2	1	2
14	2	1	0	2	1	0	2	1	0	2	1	0	2
15	1	0	2	1	0	2	1	0	2	1	0	2	2

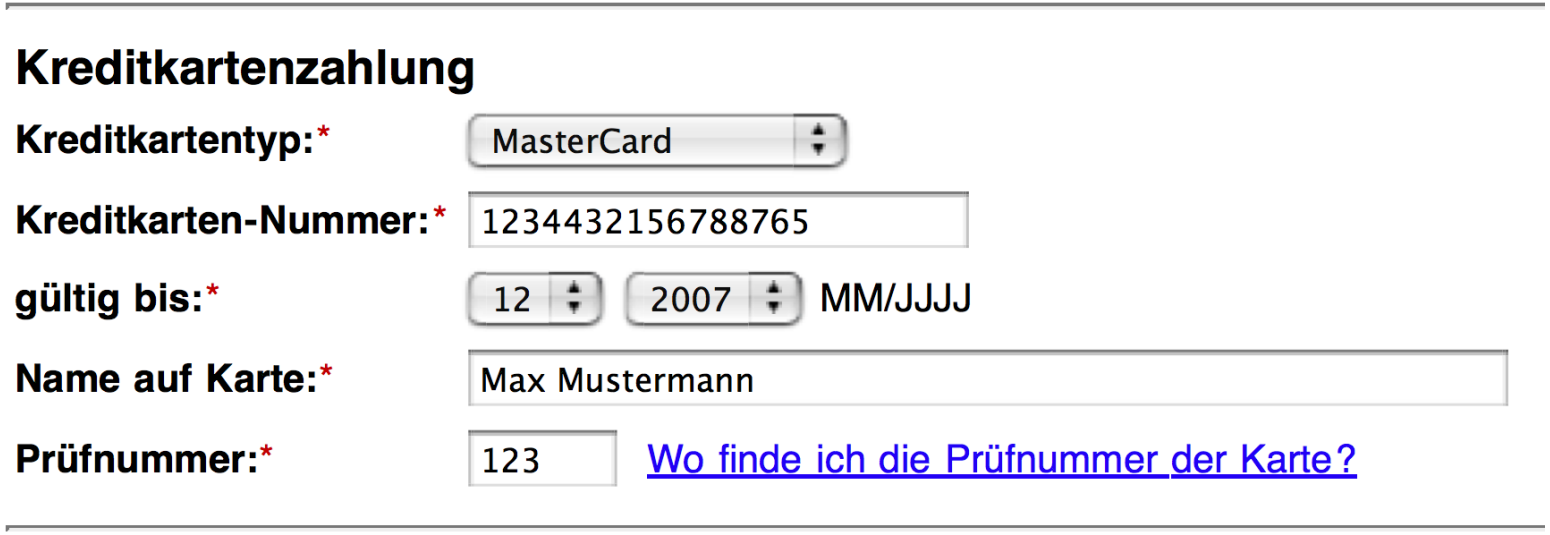
[Cohen et al. 1997]

Bestimmung der Testfälle

- Es gibt keinen einfachen Algorithmus, mit dem sich ein minimaler Satz von Testfällen manuell bestimmen lässt
- Cohen et al. (1997) geben einen Algorithmus zur maschinellen Berechnung an
- Paarweises Testen erfordert ein Werkzeug zur Bestimmung der Testfälle
- Kommerzielle Testwerkzeuge enthalten vielfach einen Generator für paarweises Testen
- Auch ein freies Perl-Skript zur Bestimmung aller Paare ist verfügbar [Bach 2006]

Beispiel: Test einer Kreditkartenzahlung

Folgende Zahlungsapplikation für eine Webseite ist zu testen:



Aufgabe: Überlegen Sie sich für jedes Eingabefeld die möglichen Äquivalenzklassen

Anzahl der Testfälle

- Angenommen, für jedes Eingabefeld werden drei Äquivalenzklassen bestimmt
- Es gibt sechs Eingabefelder mit je drei Testwerten
- Der Test aller Kombinationen erfordert $3^6 = 729$ Testfälle
- Paarweiser Test kommt mit 15 Testfällen aus
- Wie fehlersensitiv ist paarweises Testen in diesem Fall?
- Code für die Eingabe der Prüfnummer:

```
<TD><P>
 <INPUT TYPE="text" NAME="kreditkartePruefziffer" VALUE=""
 SIZE=6 MAXLENGTH=3>  <A HREF="/hilfe/de/pruefnummer.html"
 TARGET="_blank">Wo finde ich die Pr&uuml;fnummer der Karte?</A>
</P></TD>
```

Sensitivitätsanalyse

- Der Test aller Kombinationen findet einen Fehler:
  - Bei Mastercard und Visa ist die Prüfnummer dreistellig, bei American Express vierstellig
  - Die Eingabe einer vierstelligen Prüfnummer erweist sich jedoch als unmöglich
- Jeder Testfall {American Express, •, •, •,•, 1234}
  findet diesen Fehler, wobei „•“ für einen beliebigen Eingabewert steht
- ➪ Paarweises Testen genügt, um den Fehler zu finden

3.5 Testautomatisierung

- Kreative Tätigkeiten vs. Routinetätigkeiten beim Testen

Teststrategie

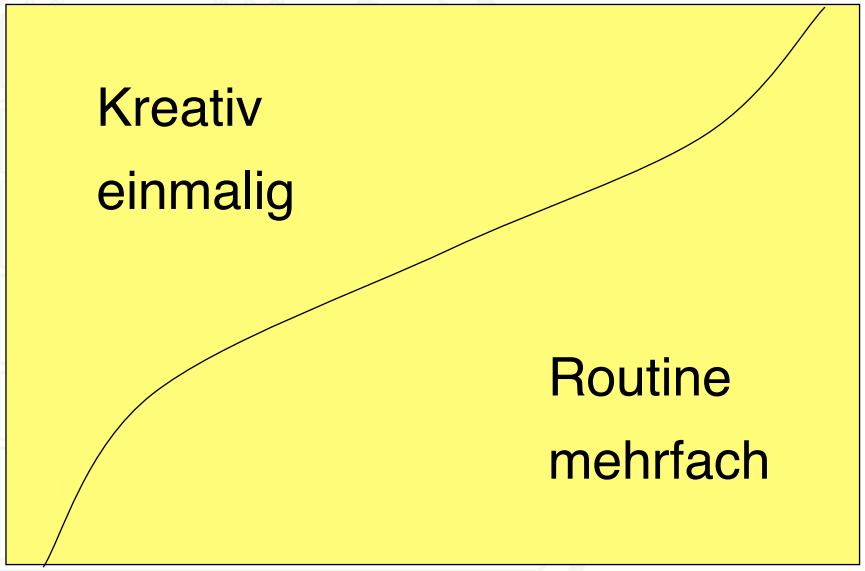
Ableiten von Testfällen

Aufbau der Testvorschrift

Einrichten des Tests

Durchführung des Tests

Auswertung des Tests



- Routinetätigkeiten sind leichter automatisierbar
- Automatisierung wiederholter Tätigkeiten ist wirtschaftlich

Vorteile und Grenzen der Testautomatisierung

- Vorteile
  - Große Zahl von Fällen testbar
  - Entlastung der Tester von Routinearbeit
  - Regelmäßiger Regressionstest möglich
  - Verbessert die Testproduktivität
- Grenzen
  - Kann manuelles Testen nicht vollständig ersetzen
  - Güte stark von der Güte des Testorakels abhängig
  - Automatisierung macht einen Test nur effizienter, nicht effektiver
  - Erstellungsaufwand muss in angemessenem Verhältnis zum Effizienzgewinn stehen
  - Kein Mittel gegen zu wenig Zeit oder unerfahrene Tester

Automatisierung der Testfall-Auswahl

- Generierung strukturorientierter Tests
  - Generierung von Testfällen, die ein gegebenes Überdeckungs-kriterium erfüllen, ist möglich
  - Problem: woher kommen die erwarteten Ergebnisse?
- Generierung von Schnittstellentests
  - Tests der formalen Eigenschaften einer Benutzerschnittstelle generierbar, zum Beispiel tote Links, nicht edierbare Eingabefelder
- Generierung funktionsorientierter Tests einschließlich des Orakels
  - erfordert eine formale Spezifikation
  - nur sehr begrenzt praktisch einsetzbar
- Unterstützung bei der Testfall-Auswahl, zum Beispiel Berechnung der Tupel für paarweises Testen

Automatisierung der Testvorschrift

Unterschiedliches Vorgehen auf den verschiedenen Teststufen:

- Modul- und Integrationstest
- Systemtest
- Abnahmetest

- Nicht nur die Testfälle, sondern auch der Vergleich mit den erwarteten Ergebnissen muss automatisiert werden

Automatisierung – Modul- und Integrationstest

- Testvorschrift wird programmiert
  - Typisch eine Methode pro Testfall
  - Vergleich mit erwarteten Ergebnissen wird ebenfalls programmiert
  - Testrahmen
    - vereinfacht die Programmierarbeit
    - Dient als Testtreiber
    - Visualisiert die Ergebnisse

- Bekanntester Testrahmen für automatisierten Modultest:
  - JUnit [Gamma und Beck 2000]
  - Mittlerweile auch für andere Sprachen verfügbar: CppUnit, PyUnit,...

Automatisierung – Systemtest

- Problem Benutzerinteraktion muss simuliert werden
- Typisch durch
  - Schreiben oder Aufzeichnen von Skripts,
  - die anschließend automatisch ablaufen,
  - in Skriptsprachen wie VBScript, Applescript, Perl, Python,...
- Ansatzmöglichkeiten
  - Auf der Präsentationsschicht
    - physisch
    - logisch
  - Auf der Funktionsschicht

Automatisierung auf der Präsentationsschicht

- Physisch: Tastendrücke, Mausbewegungen, Mausklicks,...
  - Realitätsnah
  - Skripte sehr „bodennah“: enthalten beispielsweise absolute Bildschirmkoordinaten
  - typisch weder lesbar noch änderbar
  - reagiert empfindlich auf kleinste, auch irrelevante Änderungen
  - Vergleich erwarteter und tatsächlicher Ergebnisse schwierig
- Logisch: Menubefehle auswählen, Dialogoptionen wählen,...
  - Dialogsimulation auf einer abstrakteren Ebene
  - stabilere, leichter lesbare und änderbare Skripte

Automatisierung auf der Funktionsschicht

- Zugriff auf die Systemfunktionalität
  - über Programmierschnittstellen
  - teilweise über Webschnittstellen
- Testet die Benutzerschnittstelle nicht
- Stabile, benutzerschnittstellenunabhängige Testprogramme bzw.. Testskripte
- Vergleich erwarteter und tatsächlicher Ergebnisse einfach
- Programmierschnittstellen müssen vorhanden sein
- Vorsicht: bieten auch mögliche Ansatzpunkte für Angriffe
- Automatisierung wird durch geeignete Software-Architektur vereinfacht
  - Entkopplung von Programmlogik und Präsentation
  - beispielsweise mit dem Model-View-Controller Muster

Automatisierung – Abnahmetest

- Erzeugung von Testfällen aus Anforderungen
  - Bei hinreichender Formalität sind aus Anforderungen Testfälle generierbar
  - Aus teilformalen Modellen sind zumindest Testgerüste generierbar
- Erzeugung von Testfällen aus Beispielen
  hier gezeigt an Hand des Testrahmens Fit [Cunnigham 2002]
  - Benutzer beschreiben erwartetes Verhalten in tabellarischer Form
  - Tester schreibt eine “Fixture”, welche die Tabelle in Beziehung zum zu testenden Programm setzt
  - Fit führt den Test automatisch aus und visualisiert die Ergebnisse

Automatisierung –Beispiel mit Fit

[Cunnigham 2002]

Benutzer spezifiziert Beispielfälle:

Payroll Fixtures, Weekly Compensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360

Programmierer schreibt “Fixture”:

```
public class WeeklyCompensation : ColumnFixture
{
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;
}
```

Fit führt Tests durch und visualisiert die Ergebnisse benutzergerecht:

Payroll Fixtures, Weekly Compensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $ 1040 *actual*

Automatisierung des Ergebnisvergleichs

- Für jeden Testfall müssen erwartetes und tatsächliches Resultat verglichen werden
- Möglichkeiten:
  - Vergleich während der Ausführung
  - Vergleich nach der Ausführung
- Mechanismen, welche erwartete und tatsächliche Ergebnisse automatisiert vergleichen, heißen Test-Orakel
- Probleme
  - Umfang der Überprüfung
  - Fehler im Orakel führen zu falsch-positiven Ergebnissen
  - Orakel können signifikante und bedeutungslose Abweichungen nicht unterscheiden: führt zu falsch-negativen Ergebnissen

Automatisierung der Testdurchführung/-auswertung

- Erfordert ausführbare Testvorschriften, einschließlich Testorakel
  - Programmierte Testvorschriften
  - Testskripts
- Einrichtung, Durchführung und Auswertung sind ganz oder teilweise automatisierbar
  - Beispiel: Cruisecontrol für automatisierten Modul- und Integrationstest

Literatur

A. Almagro, P. Julius (2001). *CruiseControl Continuous Integration Toolkit*. http://cruisecontrol.sourceforge.net

J. Bach (2006). *ALLPAIRS Test Case Generation Tool* (Version 1.2.1) http://www.satisfice.com/tools.shtml

K. Beck (2002). *Test Driven Development by Example*. Addison-Wesley.

D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton (1997). The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering* **23**, 7. 437-444.

M. Fewster, D. Graham (1999). *Software Test Automation*. New York: ACM Press.

E. Gamma, K. Beck (2000). *JUnit Test Framework*. http://www.junit.org

P. Liggesmeyer (2002). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Heidelberg; Berlin: Spektrum Akademischer Verlag.

S. Rapps, E.J. Weyuker (1985). Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering* **SE-11**, 4. 367-375.

W. Cunningham (2002). *Fit: Framework for Integrated Test*. http://fit.c2.com

A. Zeller (2005). *Why Programs Fail: A Guide to Systematic Debugging*. Amsterdam: Morgan Kaufmann und Heidelberg: dpunkt.