

Martin Glinz Harald Gall

# Software Engineering

## Kapitel 8

# Testen von Software



Universität Zürich  
Institut für Informatik

# 8.1 Grundlagen

---

8.2 Vorgehen

8.3 Testfälle

8.4 Testverfahren

8.5 Testplanung und -dokumentation

8.6 Testen von Teilsystemen

8.7 Besondere Testformen

8.8 Kriterien für den Testabschluss

# Was ist Testen?

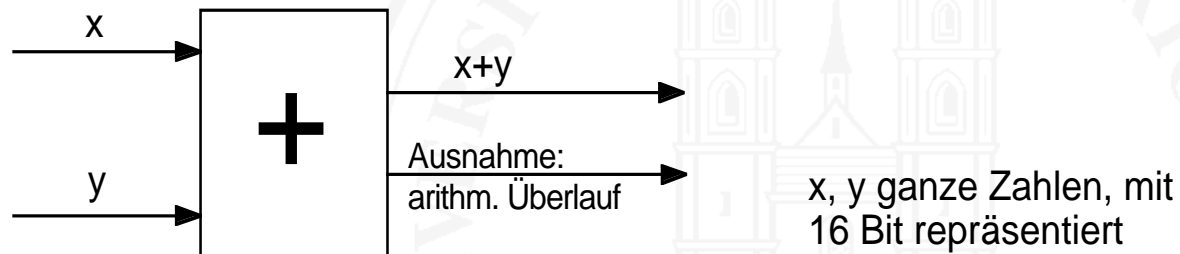
---

- **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

# Mini-Übung 8.1

Wieviele Testfälle wären erforderlich, um die Addition zweier 16-Bit Festkommazahlen vollständig zu testen?



# Aufwand für vollständigen Test

---

- Ein vollständiger Test ist nur in wenigen Ausnahmefällen möglich
  - Beispiel: Entscheidungsbaum mit 10 binären Eingängen:  $2^{10}$  Testfälle
- Im Allgemeinen kann **Korrektheit** durch Testen **nicht bewiesen** werden
  - Beispiel: Vergleich von zwei 4-Byte Zeichenketten:  $2^{64}$  Testfälle

Kleine Aufwandrechnung:  $2^{64} \approx 1,8 \cdot 10^{19}$

**Szenario 1: Manueller Test mit 1 Testfall/s**

Vollständiger Test dauert ca.  $1,8 \cdot 10^{19}$  s  $\approx$  **58,5 Milliarden Jahre**

**Szenario 2: Automatisierter Test** auf 1000 Rechnern parallel,  
1 Testfall /  $\mu$ s  $\rightarrow$   **$10^9$  Testfälle/s**

Vollständiger Test dauert ca.  $1,8 \cdot 10^{10}$  s  $\approx$  **58,5 Jahre**

# 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**)

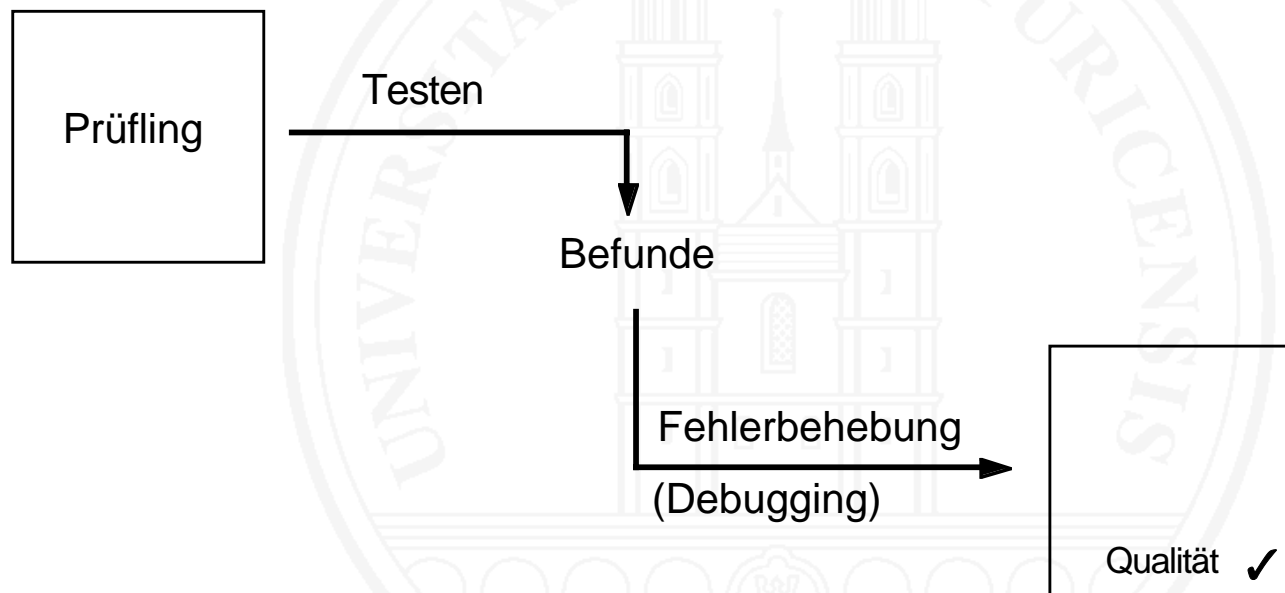


- Unvorbereitete und undokumentierte Tests sind **sinnlos**

# Testen ist nicht alles

---

- Mit Testen werden nur Fehlersymptome, nicht aber die Fehlerursachen gefunden



- Mit Testen können nicht alle Eigenschaften eines Programms geprüft werden (z.B. Wartbarkeit)

8.1 Grundlagen

8.2 Vorgehen

---

8.3 Testfälle

8.4 Testverfahren

8.5 Testplanung und -dokumentation

8.6 Testen von Teilsystemen

8.7 Besondere Testformen

8.8 Kriterien für den Testabschluss



# Testsystematik: Test ist nicht gleich Test

---

- **Laufversuch:** Der Entwickler „testet“
  - Entwickler übersetzt, bindet und startet sein Programm
  - Läuft das Programm nicht oder sind Ergebnisse offensichtlich falsch, werden die Defekte gesucht und behoben (“Debugging”)
  - Der „Test“ ist beendet, wenn das Programm läuft und die Ergebnisse vernünftig aussehen
- **Wegwerf-Test:** Jemand testet, aber ohne System
  - Jemand führt das Programm aus und gibt dabei Daten vor
  - Werden Fehler erkannt, so werden die Defekte gesucht und behoben
  - Der Test endet, wenn der Tester findet, es sei genug getestet

# Testsystematik – 2

---

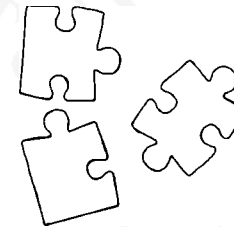
- **Systematischer Test:** Spezialisten testen
  - Test ist geplant, Testvorschrift ist vorgängig erstellt
  - Programm wird gemäß Testvorschrift ausgeführt
  - Ist-Resultate werden mit Soll-Resultaten verglichen
  - Fehlersuche und -behebung erfolgen separat
  - Nicht bestandene Tests werden wiederholt
  - Testergebnisse werden dokumentiert
  - Test endet, wenn vorher definierte Testziele erreicht sind

# Testgegenstand und Testarten

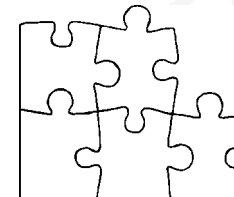
---

- Testgegenstand sind **Komponenten**, **Teilsysteme** oder **Systeme**

- **Komponententest**, Modultest (Unit Test)



- **Integrationstest** (Integration Test)



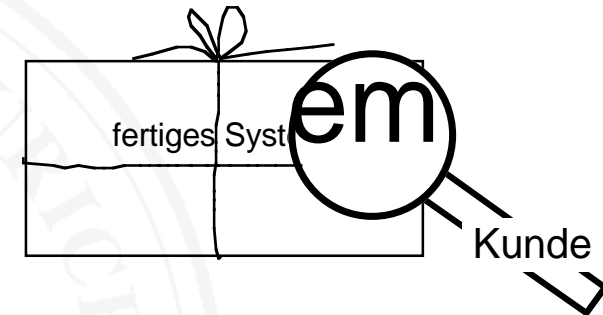
- **Systemtest** (System Test)



# 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

8.1 Grundlagen

8.2 Vorgehen

**8.3 Testfälle**

---

8.4 Testverfahren

8.5 Testplanung und -dokumentation

8.6 Testen von Teilsystemen

8.7 Besondere Testformen

8.8 Kriterien für den Testabschluss

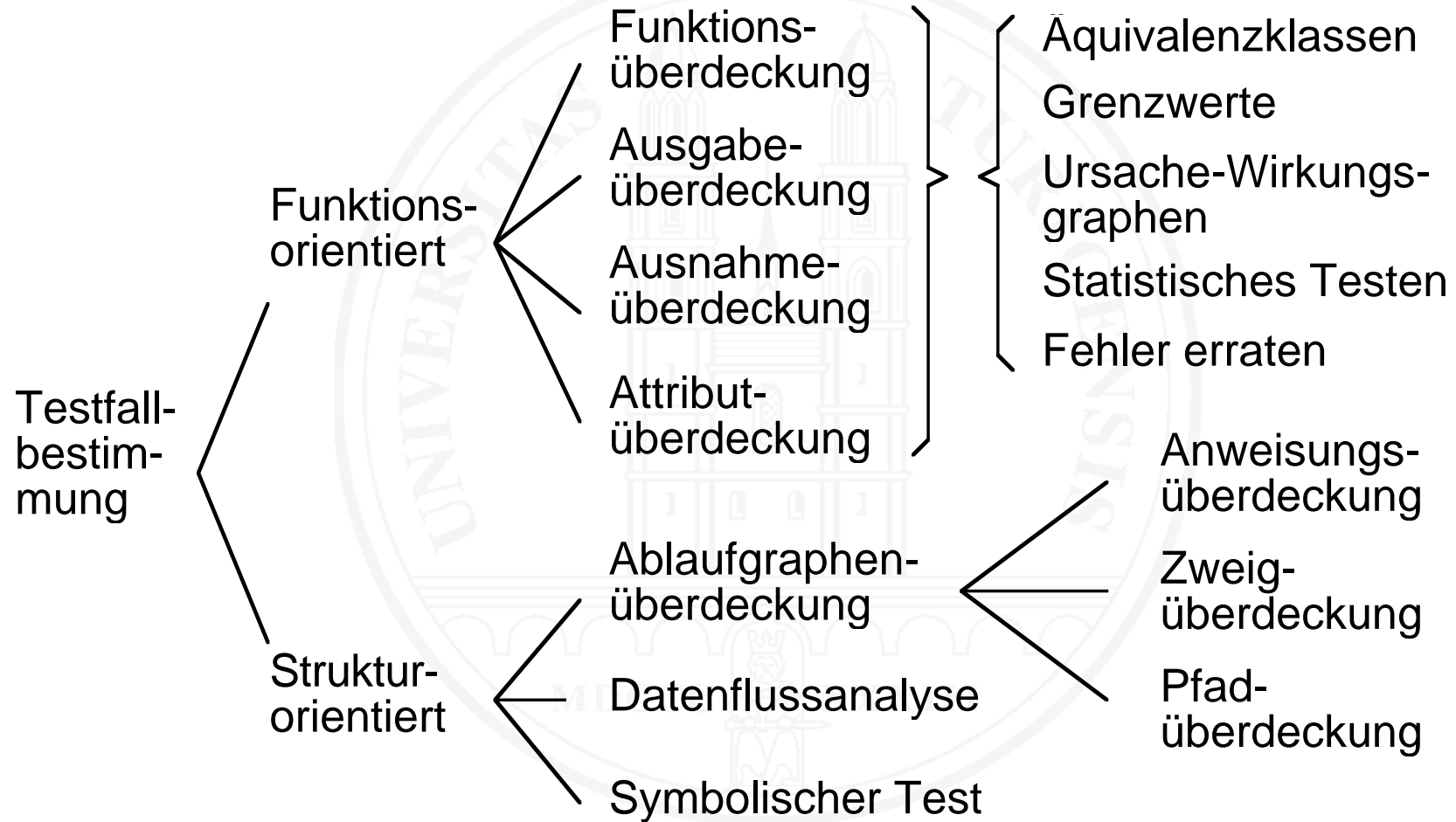
# Auswahl von Testfällen

---

- Auswahl der Testfälle ist eine zentrale Aufgabe des Testens
- Anforderungen an Testfälle
  - repräsentativ
  - fehlersensitiv
  - redundanzarm
  - ökonomisch
- Ziel: Mit **möglichst wenig Testfällen** **möglichst viele Fehler** finden kommen
- Verschiedenen Verfahren → Abschnitt 8.4



# Bestimmen von Testfällen



8.1 Grundlagen

8.2 Vorgehen

8.3 Testfälle

**8.4 Testverfahren**

---

8.5 Testplanung und -dokumentation

8.6 Testen von Teilsystemen

8.7 Besondere Testformen

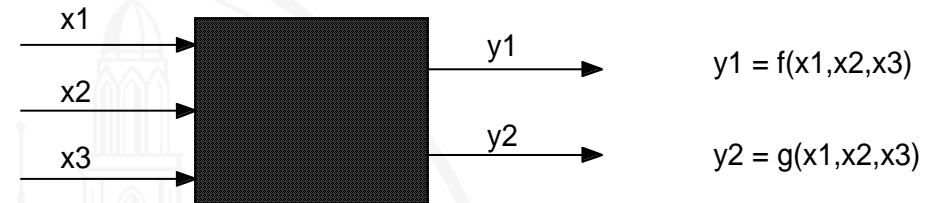
8.8 Kriterien für den Testabschluss

# Zwei Klassen von Verfahren

---

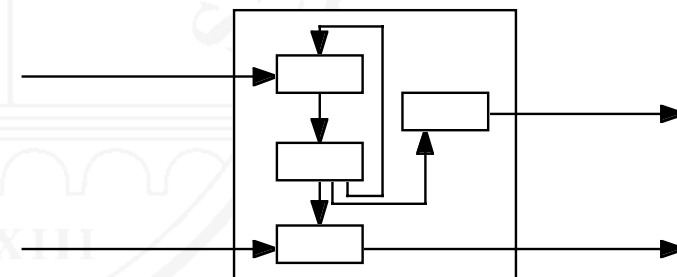
## ○ Funktionsorientierter Test (Black-Box-Test)

- Testfall-Auswahl aufgrund der Spezifikation
- Programmstruktur kann unbekannt sein



## ○ Strukturorientierter Test (White-Box-Test, Glass-Box-Test)

- Testfall-Auswahl aufgrund der Programmstruktur
- Spezifikation muss ebenfalls bekannt sein (wegen der erwarteten Resultate)



## 8.4.1 Funktionsorientierter Test

---

Mögliche **Testziele** (einzeln oder in Kombination):

- **Funktionsüberdeckung**: jede spezifizierte Funktion mindestens einmal aktiviert
- **Ausgabeüberdeckung**: jede spezifizierte Ausgabe mindestens einmal erzeugt
- **Ausnahmeüberdeckung**: jede spezifizierte Ausnahme- bzw. Fehlersituation mindestens einmal erzeugt
- **Attributüberdeckung**: alle geforderten Attribute (soweit technisch möglich) getestet
  - insbesondere Erreichung der spezifizierten **Leistungsanforderungen**
    - unter normalen Bedingungen
    - unter möglichst ungünstigen Bedingungen (**Belastungstest**)

# Auswahl der Testfälle

---

- Problem der Testfall-Auswahl: die gewählten Testziele mit
  - möglichst wenig
  - möglichst gutenTestfällen umsetzen
- Klassische Techniken:
  - Äquivalenzklassenbildung
  - Grenzwertanalyse
  - Ursache-Wirkungsgraphen
  - Statistisches Testen (random testing)
  - Fehler raten (error guessing)

# Äquivalenzklassenbildung

---

- Gleichartige Eingabedaten
  - werden zu **Klassen** zusammengefasst
  - aus jeder Klasse wird ein **Repräsentant** getestet
- Die Klasseneinteilung ist eine **Äquivalenzrelation**
- Beispiel: Multiplikation von ganzen Zahlen  
Mögliche Äquivalenzklassen
  - x und y positiv
  - x positiv, y negativ
  - x negativ, y positiv
  - x und y negativ

# Grenzwertanalyse

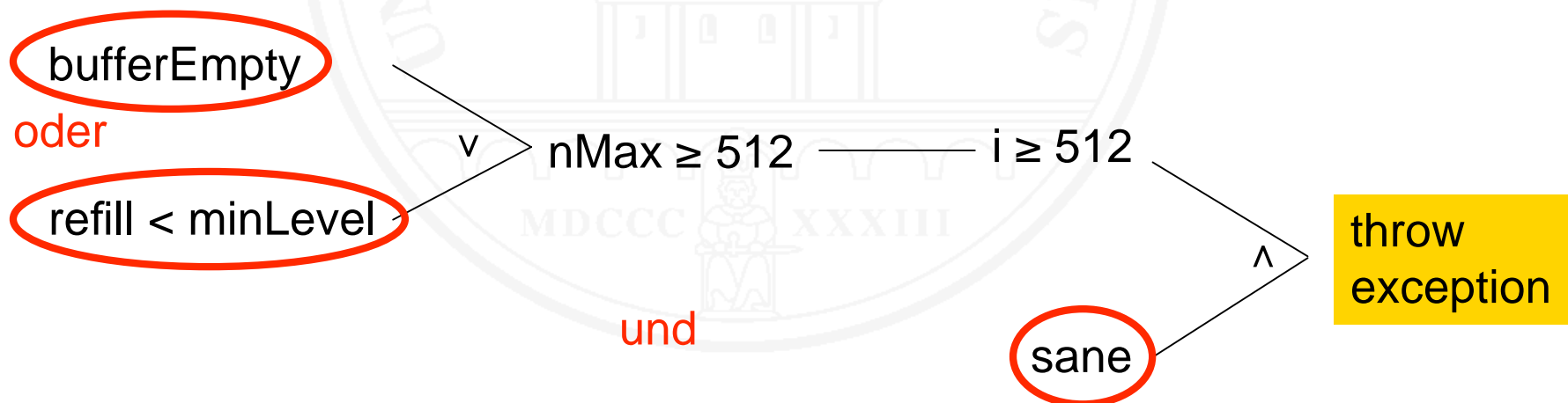
---

- An den **Grenzen** zulässiger Datenbereiche treten erfahrungsgemäß häufig Fehler auf
- Testfälle für solche **Grenzfälle** auswählen
- Beispiel: Multiplikation von ganzen Zahlen  
Mögliche Grenzfälle
  - $x$  ist null
  - $y$  ist null
  - $x$  und  $y$  sind beide null
  - Produkt läuft positiv über
  - Produkt läuft negativ über

# Ursache-Wirkungsgraphen

- Ursache-Wirkungsgraphen dienen zur **systematischen Bestimmung** von **Eingabedaten**, die ein gewünschtes **Ergebnis** bewirken
- Beispiel: In folgendem Fragment soll die Ausnahme erzeugt werden

```
...  
if (bufferEmpty | (refill < minLevel)) {  
  for (i:=0; i<= nMax; i++) {  
    if (i >= 512 && sane) throw new OverflowException ("charBuffer")  
  }  
}
```





# Statistisches Testen (random testing)

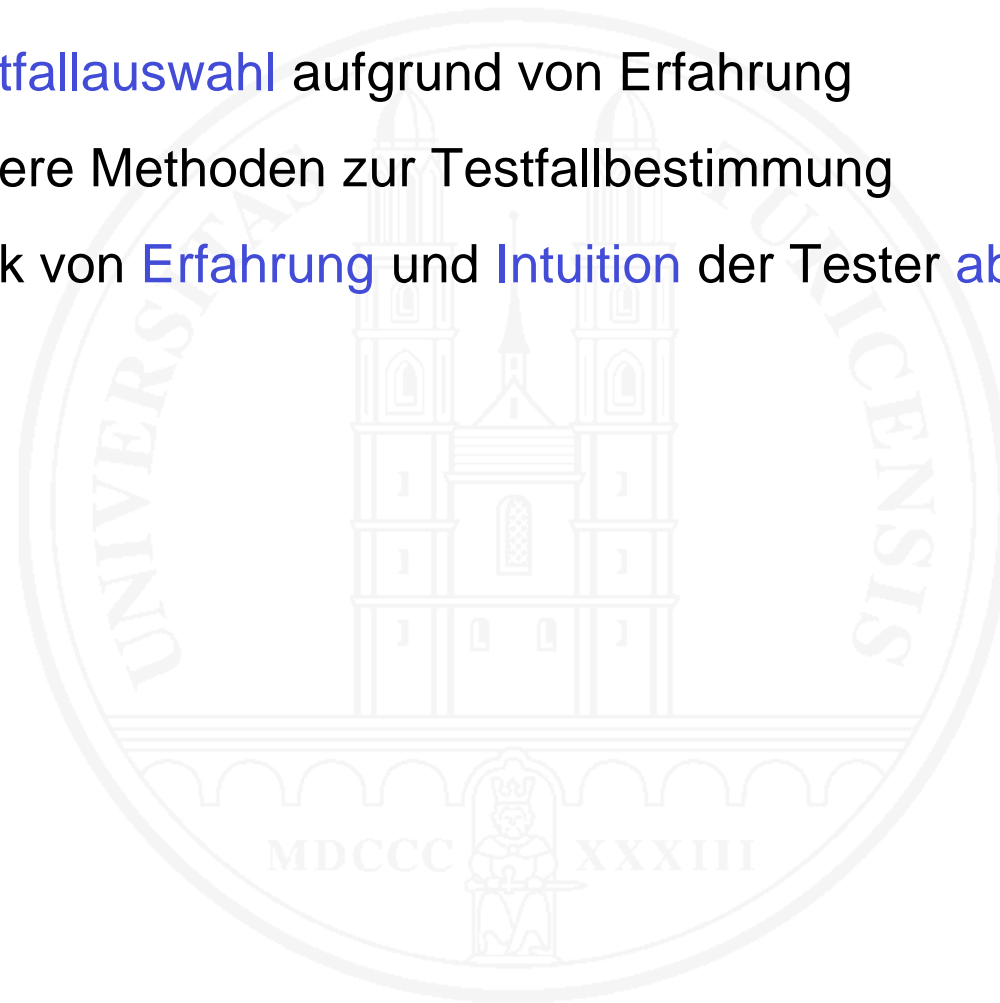
---

- Eingabedaten werden **zufällig ausgewählt**
- Die gezielte Testfall-Auswahl wird durch eine **große Zahl von Testfällen ersetzt**
- Bei hinreichend großer Zahl von Testfällen ohne Befund sind **statistische Aussagen** über die **Zuverlässigkeit** der Software möglich
- **Automatisierter Testablauf** mit Orakel notwendig (**Orakel**: stellt fest, ob tatsächliches und erwartetes Ergebnis übereinstimmen)
- Problem: Auswahl der Eingabedaten muss der **tatsächlichen Verteilung** der Eingabedaten im produktiven Betrieb der Software entsprechen
- Näherungsweise über **Benutzungsprofile** möglich

# Fehler raten (error guessing)

---

- **Intuitive Testfallauswahl** aufgrund von Erfahrung
- **Ergänzt** andere Methoden zur Testfallbestimmung
- **Qualität** stark von **Erfahrung** und **Intuition** der Tester **abhängig**



# Beispiel zum funktionsorientierten Test: Spezifikation

---

Gegeben sei ein Programm, das folgende Spezifikation erfüllen soll:

Das Programm fordert zur Eingabe von drei nicht negativen reellen Zahlen auf und liest die eingegebenen Werte.

Das Programm interpretiert die eingegebenen Zahlen als Strecken  $a$ ,  $b$  und  $c$ .

Es untersucht, ob die drei Strecken ein Dreieck bilden und klassifiziert gültige Dreiecke.

Das Programm liefert folgende Ausgaben:

- kein Dreieck wenn  $a+b \leq c$  oder  $a+c \leq b$  oder  $b+c \leq a$
- gleichseitiges Dreieck, wenn  $a=b=c$
- gleichschenkliges Dreieck, wenn  $a=b$  oder  $b=c$  oder  $a=c$
- unregelmäßiges Dreieck sonst

## Beispiel: Spezifikation – 2

---

Das Programm zeichnet ferner alle gültigen Dreiecke winkeltreu und auf maximal darstellbare Größe skaliert in einem Fenster der Größe 10x14 cm. Die Seite c liegt unten parallel zur Horizontalen. Alle Eckpunkte haben einen Minimalabstand von 0,5 cm vom Fensterrand.

Das Programm liefert eine Fehlermeldung, wenn andere Daten als drei nicht negative reelle Zahlen eingegeben werden. Anschließend wird mit einer neuen Eingabeaufforderung versucht, gültige Werte einzulesen.

# Beispiel: Testüberdeckungskriterien

---

## a) Aktivierung aller Funktionen

- Prüfen und Klassifizieren
- Skalieren und Zeichnen

## b) Erzeugen aller Ausgaben

- kein Dreieck
- gleichseitiges Dreieck
- gleichschenkliges Dreieck
- unregelmäßiges Dreieck

## c) Erzeugung aller Ausnahmesituationen

- ungültige Eingabe

# Beispiel: Äquivalenzklassenbildung

---

## Klasse, Subklasse

## Repräsentant

kein Dreieck

a größte Seite

4.25, 2, 1.3

b größte Seite

1.3, 4.25, 2

c größte Seite

2, 1.3, 4.25

gleichseitiges Dreieck

4.2, 4.2, 4.2

gleichschenkliges Dreieck

a=b

4.71, 4.71, 2

b=c

3, 5.6, 5.6

a=c

11, 6, 11

# Beispiel: Äquivalenzklassenbildung – 2

---

unregelmäßiges Dreieck

$\alpha$  spitz,  $\beta$  spitz 3, 5, 6

$\alpha$  spitz  $\beta$  rechtwinklig 3, 5, 4

$\alpha$  spitz  $\beta$  stumpf 3, 6, 4

$\beta$  spitz,  $\gamma$  spitz 6, 3, 5

$\beta$  spitz  $\gamma$  rechtwinklig 4, 3, 5

$\beta$  spitz  $\gamma$  stumpf 4, 3, 6

$\gamma$  spitz,  $\alpha$  spitz 5, 6, 3

$\gamma$  spitz  $\alpha$  rechtwinklig 5, 4, 3

$\gamma$  spitz  $\alpha$  stumpf 6, 4, 3

# Beispiel: Äquivalenzklassenbildung – 3

---

ungültiger Eingabe

negative Zahlen

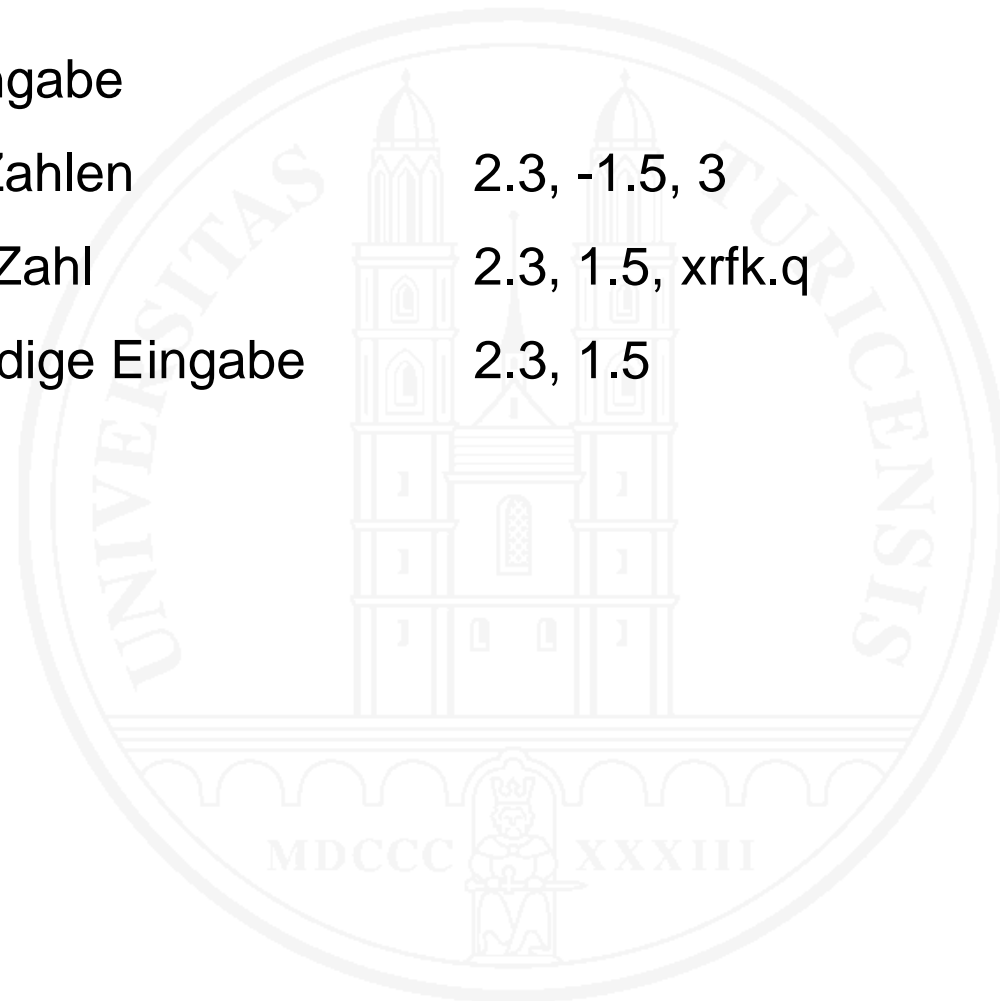
2.3, -1.5, 3

Text statt Zahl

2.3, 1.5, xrfk.q

unvollständige Eingabe

2.3, 1.5





# Beispiel: Grenzwertanalyse

---

## Grenzfall

## Testwerte

kein Dreieck

$$a=b=c=0$$

0, 0, 0

$$a=b+c$$

6, 2, 4

$$b=a+c$$

2, 6, 4

$$c=a+b$$

2, 4, 6

sehr flaches Dreieck

$$c=a+b - \varepsilon, \varepsilon \text{ sehr klein}$$

3, 4, 6.999999999999999

$$b=a+c - \varepsilon, \varepsilon \text{ sehr klein}$$

3, 6.999999999999999, 4

sehr steiles Dreieck

$$c \text{ klein, } a=b \text{ sehr groß}$$

$10^7$ ,  $10^7$ , 5

## 8.4.2 Strukturorientierter Test

---

- Auch als **Glass-Box-Test** oder **White-Box-Test** bezeichnet
- Auswahl der Testfälle so, dass
  - der **Programmablauf** oder
  - der **Datenfluss** im Programm
- überprüft wird
  
- Meistens wird der **Programm-Ablauf** getestet: Testfälle werden so gewählt, dass das Programm systematisch durchlaufen wird
- In der Regel nur für Modultest und teilweise für Integrationstest möglich

# Überdeckungen

---

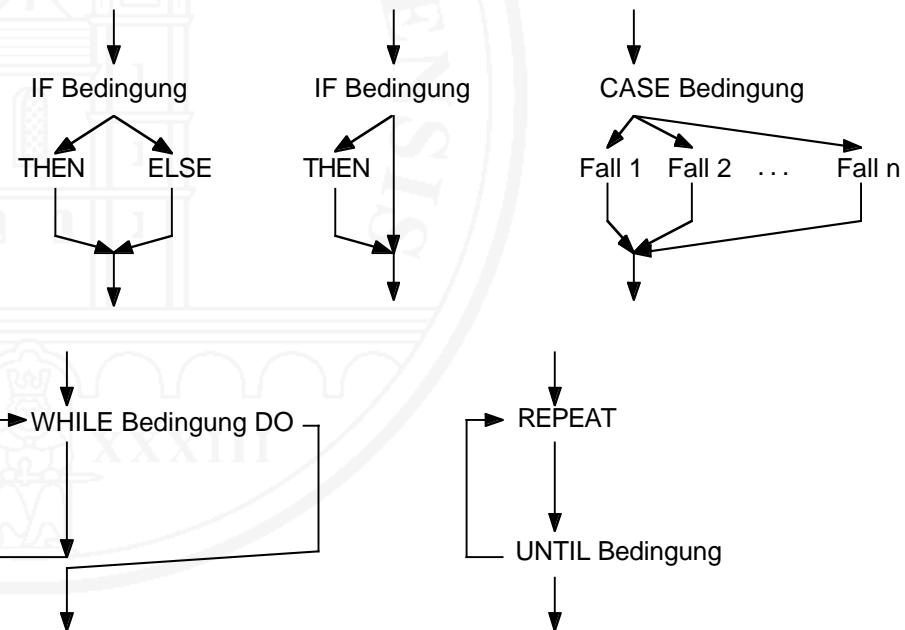
Gebräuchlich sind drei **Testziele** für den strukturorientierten Test des Programmablaufs:

- **Anweisungsüberdeckung:** Jede **Anweisung** des Programms wird mindestens **einmal** ausgeführt
- **Zweigüberdeckung:** jeder **Programmzweig** wird mindestens **einmal** durchlaufen
- **Pfadüberdeckung:** jeder **Programmpfad** wird mindestens **einmal** durchlaufen

# Bestimmung von Zweigen und Pfaden

- Bestimmung der **Programmzweige**:
  - Betrachtung von Verzweigungen und Schleifen
  - Bei Programmiersprachen mit geschlossenen Ablaufkonstrukten:  
if-Anweisungen und Schleifen haben je zwei Zweige  
Eine CASE /switch-Anweisung: so viele Zweige wie Fälle

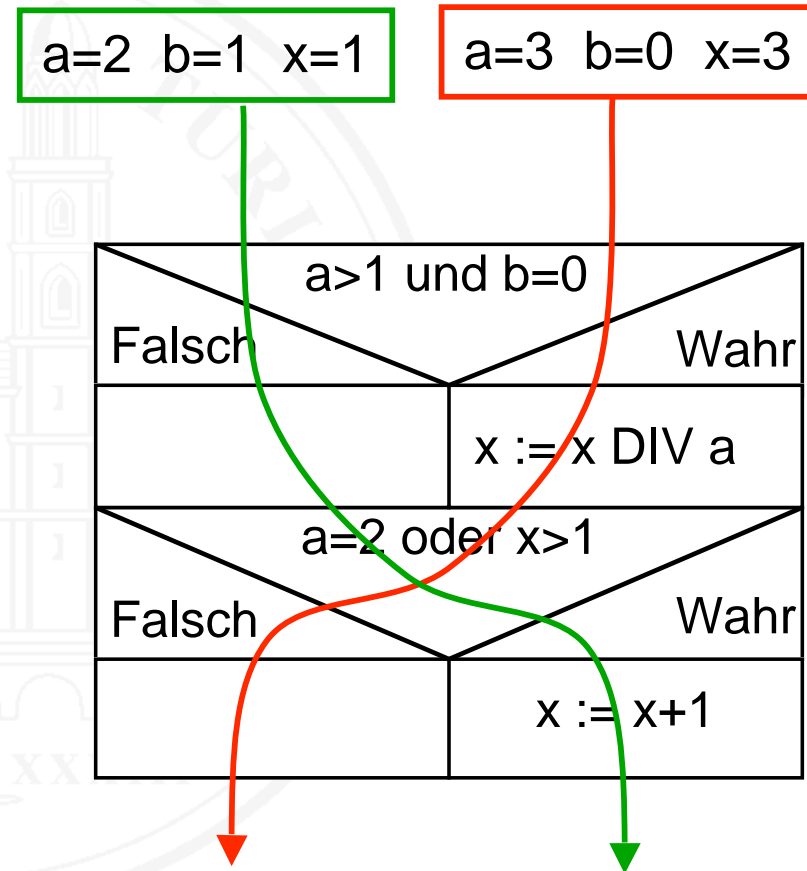
- Bestimmung der **Pfade**:
  - Alle Kombinationen aller
  - Programmzweige bei
  - maximalem Durchlauf
  - aller Schleifen



# Beispiel

(nach Myers, 1979)

```
...  
VAR  
  a, b, x: INTEGER;  
...  
BEGIN  
...  
IF (a > 1) AND (b = 0)  
  THEN x := x DIV a;  
IF (a = 2) OR (x > 1)  
  THEN x := x + 1;  
...  
...
```



# Beispiel – 2: Notwendige Testfälle

---

- **Anweisungsüberdeckung** mit dem Testfall:

a=2 b=0 x=1

- **Zweigüberdeckung** mit den Testfällen:

a=3 b=0 x=3

a=2 b=1 x=1

- **Pfadüberdeckung** mit den Testfällen:

a=1 b=1 x=2      a=3 b=0 x=3

a=2 b=0 x=4      a=1 b=1 x=1

# Güte eines strukturorientierten Tests

---

- Die **Testgüte** hängt von gewählter **Überdeckung** und erreichtem **Überdeckungsgrad** ab
- **Überdeckungsgrad** – Prozentuales Verhältnis der Anzahl überdeckter Elemente zur Anzahl vorhandener Elemente
- Beispiel: Der Testfall  $a=3$   $b=0$   $x=3$  erreicht 50% Zweigüberdeckung
- **Anweisungsüberdeckung** ist ein **schwaches Kriterium**. Fehlende Anweisungen werden beispielsweise nicht entdeckt
- **Zweigüberdeckung** wird **in der Praxis angestrebt**. Dennoch: falsch formulierte Bedingungsterme (z.B.  $x>1$  statt  $x<1$ ) werden nicht entdeckt
- **Pfadüberdeckung** ist in fast allen Programmen, die Schleifen mit Verzweigungen enthalten, **nicht testbar**

## Mini-Übung 8.2

Wieviele Pfade hat das folgende Programmfragment in Modula-2?

```
...  
CONST MaxZeilenLaenge = 80;  
...  
BEGIN  
...  
FOR I := 1 TO MaxZeilenLaenge DO  
  IF (Zeichen[I] = " ")  
    THEN AnzahlLeer := AnzahlLeer + 1;  
...  
...
```



8.1 Grundlagen

8.2 Vorgehen

8.3 Testfälle

8.4 Testverfahren

**8.5 Testplanung und -dokumentation**

---

8.6 Testen von Teilsystemen

8.7 Besondere Testformen

8.8 Kriterien für den Testabschluss

# Testplanung

---

- Qualitätsprüfung muss geplant werden:
  - Was - wann - nach welcher Strategie prüfen
- Für das Testen:
  - welche Testverfahren einsetzen
  - welche Testdokumente erstellen
  - wann – welche Tests – mit welchen Leuten durchführen

# Testdokumentation

---

- Das wichtigste Dokument für Testvorbereitung und -durchführung ist die **Testvorschrift**
- Die Testvorschrift kann gleichzeitig als **Testprotokoll** dienen, wenn zu jedem Testfall das Testergebnis notiert wird
- Eine **Testzusammenfassung** bildet den Nachweis über die Durchführung und das Gesamtergebnis eines Tests
- Es gibt **Normen** mit sehr umfangreichen Vorschriften für Testplanung und -dokumentation (IEEE 1987, 1988, 1998a, 1998b)
  - für den Test kritischer Software sollten diese verwendet werden
  - für gewöhnliche Software genügen die hier genannten Dokumente

# Aufbau einer Testvorschrift – 1

---

## **1. Einleitung**

### **1.1 Zweck**

Art und Zweck des im Dokument beschriebenen Tests

### **1.2 Testumfang**

Welche Konfigurations-Einheiten der entwickelten Lösung getestet werden

### **1.3 Referenzierte Unterlagen**

Verzeichnis aller Unterlagen, auf die im Dokument Bezug genommen wird

## **2. Testumgebung**

### **2.1 Überblick**

Testgliederung, Testgüte, Annahmen und Hinweise

### **2.2 Testmittel**

Test-Software und -Hardware, Betriebssystem, Testgeschirr, Werkzeuge

# Aufbau einer Testvorschrift – 2

---

## **2.3 Testdaten, Testdatenbank**

Wo die für den Test benötigten Daten bereit liegen oder bereitzustellen sind

## **2.4 Personalbedarf**

wieviele Personen zur Testdurchführung benötigt werden

## **3. Annahmekriterien**

Kriterien für

- erfolgreichen Test-Abschluss
- Test-Abbruch
- Unterbrechung und Wiederaufnahme des Tests

## **4. Testfälle**

(siehe unten)

# Darstellung von Testfällen

---

- **Aufbau** eines Testfalls
  - Testfall-Nummer
  - Eingabe
  - Erwartetes Resultat
  - Feld zum Eintragen des Befunds
- **Gliederung** der Testfälle
  - Testfälle mit gemeinsamen Vorbereitungsarbeiten werden zu **Testabschnitten** zusammengefasst
  - Zu jedem Testabschnitt werden Zweck (was wird getestet), Vorbereitungs- und Aufräumarbeiten dokumentiert
  - Zur Verbesserung der Übersicht werden Testabschnitte untergliedert in **Testsequenzen**

# Beispiel: Testfälle – 1

---

## Testabschnitt 1: Korrekte Eingaben

- Zweck:
- testet die Klassifikationsfunktion
  - testet bei echten Dreiecken die Zeichne-Funktion

Vorbereitungsarbeiten: keine

Aufräumarbeiten: keine

Hinweis: alle Zahlen sind als Dezimalzahlen einzugeben


Testsequenz 1-1: Kein Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-1-1	4.25, 2, 1.3	kein Dreieck	
1-1-2	1.3, 4.25, 2	kein Dreieck	
1-1-3	2, 1.3, 4.25	kein Dreieck	




# Beispiel: Testfälle – 1

---

## Testsequenz 1-2: regelmäßiges Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-2-1	4.2, 4.2, 4.2	gleichseitig 	
1-2-2	4.71, 4.71, 2	gleichschenkelig 	
1-2-3	3, 5.6, 5.6	gleichschenkelig 	
1-2-4	11, 6, 11	gleichschenkelig 	

## Testsequenz 1-3: unregelmäßiges Dreieck

TestfallNr.	Eingabe	erwartetes Resultat	Befund
1-3-1	3, 5, 6	unregelmäßig 	
1-3-2	3, 5, 4	unregelmäßig 	
1-3-3	3, 6, 4	unregelmäßig 	
...	...	...	

...



# Programmierte Testvorschriften

---

- An Stelle textuell beschriebener Testfälle können auch **programmierte Testfälle** verwendet werden
  - Jeder **Testfall ist ein Objekt**
  - Enthält **Testdaten** und **erwartetes Resultat**
  - Ruft den Testling auf
  - **Vergleicht** erwartetes und geliefertes Resultat
- Eingebettet in ein passendes Laufzeitsystem sind **teilautomatisierte Tests** möglich
- Geeignet vor allem für Komponenten- und Integrationstest
- Nützlich als **kontinuierlicher Regressionstest** bei inkrementeller Entwicklung
- In Java-Umgebungen: **JUnit**

# Testzusammenfassung

- Dokumentiert
  - Testgegenstand
  - Verwendete Testvorschrift
  - Gesamtbefund
  - Wer hat getestet
- Wichtig für die Archivierung von Testergebnissen

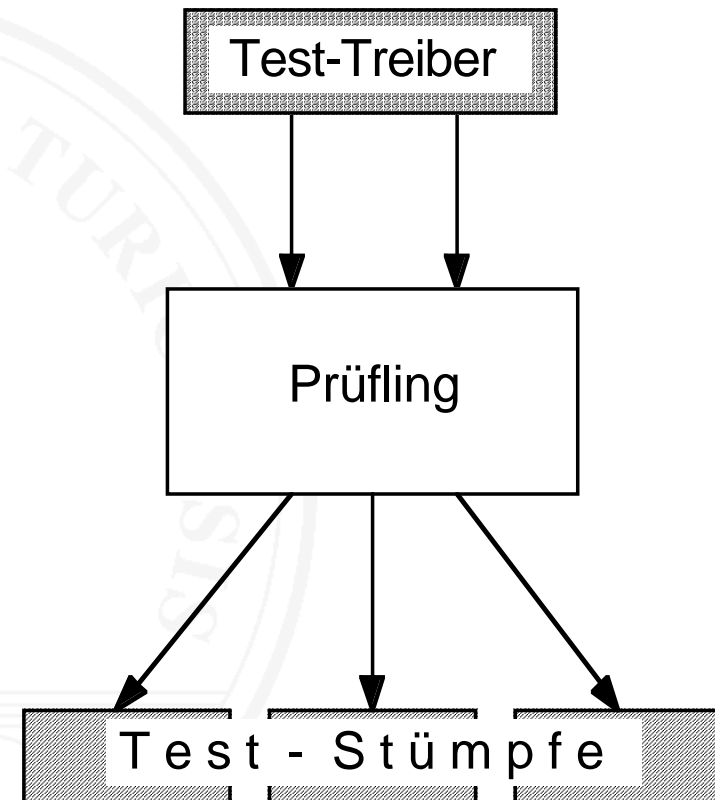
TESTZUSAMMENFASSUNG			
Test Nr.:		Arbeitspaket Nr.:	
Testbeginn (Datum und Zeit):		Test Dauer:	
Testende (Datum und Zeit):			
GEGENSTAND UND ZWECK DES TESTS			
Projekt/Produkt:		Release Nr.:	
Geliefert von:			
<input type="radio"/> Einzeltest <input type="radio"/> Integrationstest <input type="radio"/> Systemtest <input type="radio"/> Abnahmetest			
TESTVORSCHRIFT			
Nummer/Version		Titel	
EMPFEHLUNG			
<input type="radio"/> akzeptieren (keine Wiederholung des Tests)		<input type="radio"/> wie es ist	
<input type="radio"/> nicht akzeptieren (Wiederholung des Tests)		<input type="radio"/> kleine Fehler	
<input type="radio"/> Test nicht beendet		<input type="radio"/> einige grössere Fehler <input type="radio"/> einige fatale Fehler	
ZUSAMMENFASSUNG			
BEILAGEN			
<input type="radio"/> Liste der getesteten Software-Einheiten <input type="radio"/> Liste der Problemmeldungen <input type="radio"/> andere:			
TESTTEAM			
Name	(Leiter)	Datum	Visum

- 8.1 Grundlagen
  - 8.2 Vorgehen
  - 8.3 Testfälle
  - 8.4 Testverfahren
  - 8.5 Testplanung und -dokumentation
  - 8.6 Testen von Teilsystemen**
  - 8.7 Besondere Testformen
  - 8.8 Kriterien für den Testabschluss
-

# Testgeschirr

---

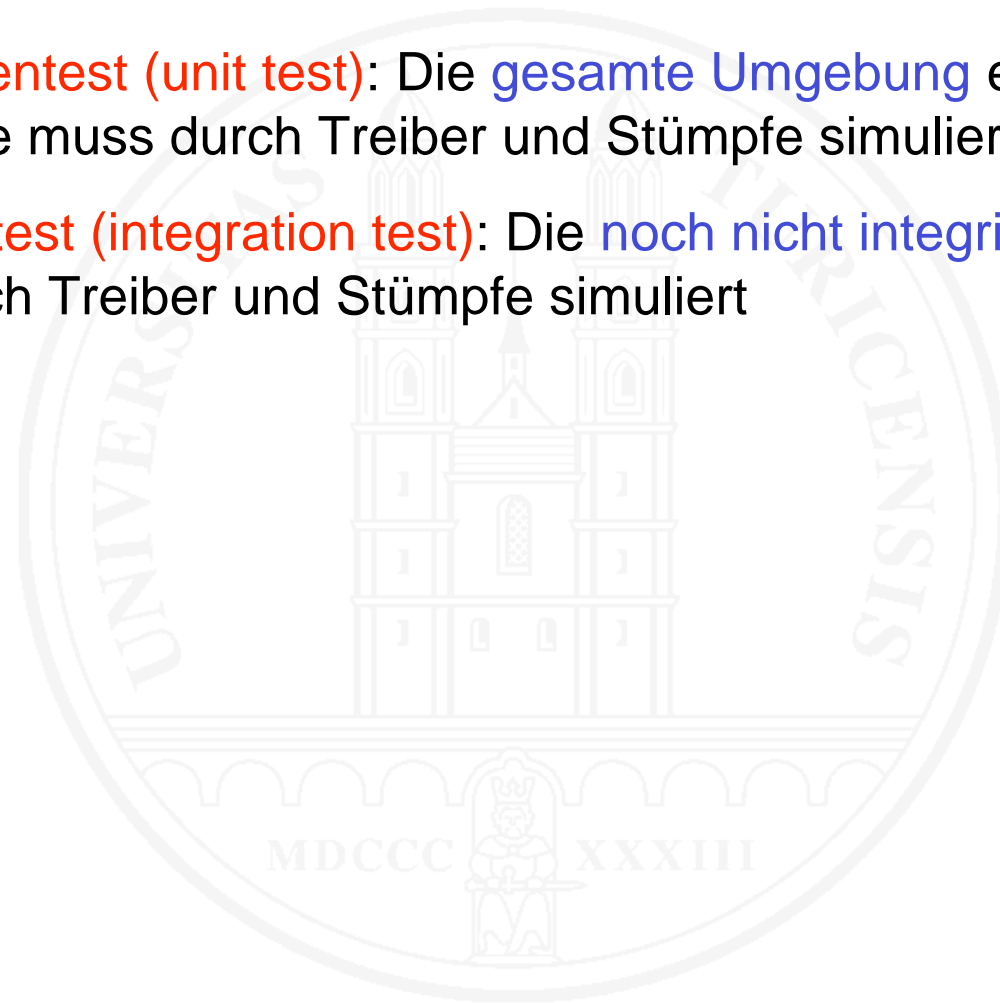
- Zum Testen unvollständiger Software wird ein **Testgeschirr (test harness)** benötigt
- Besteht aus **Testtreiber (test driver)** und **Teststümpfen (test stubs)**
- Testtreiber
  - Ruft den Prüfling auf
  - Versorgt den Prüfling mit Daten
  - Nimmt Resultate entgegen und protokolliert sie
- Teststumpf
  - Berechnet oder simuliert die Ergebnisse einer vom Prüfling aufgerufenen Operation



# Verwendung

---

- **Komponententest (unit test)**: Die **gesamte Umgebung** einer Komponente muss durch Treiber und Stümpfe simuliert werden
- **Integrationstest (integration test)**: Die **noch nicht integrierten Teile** werden durch Treiber und Stümpfe simuliert



# Integrationstest

---

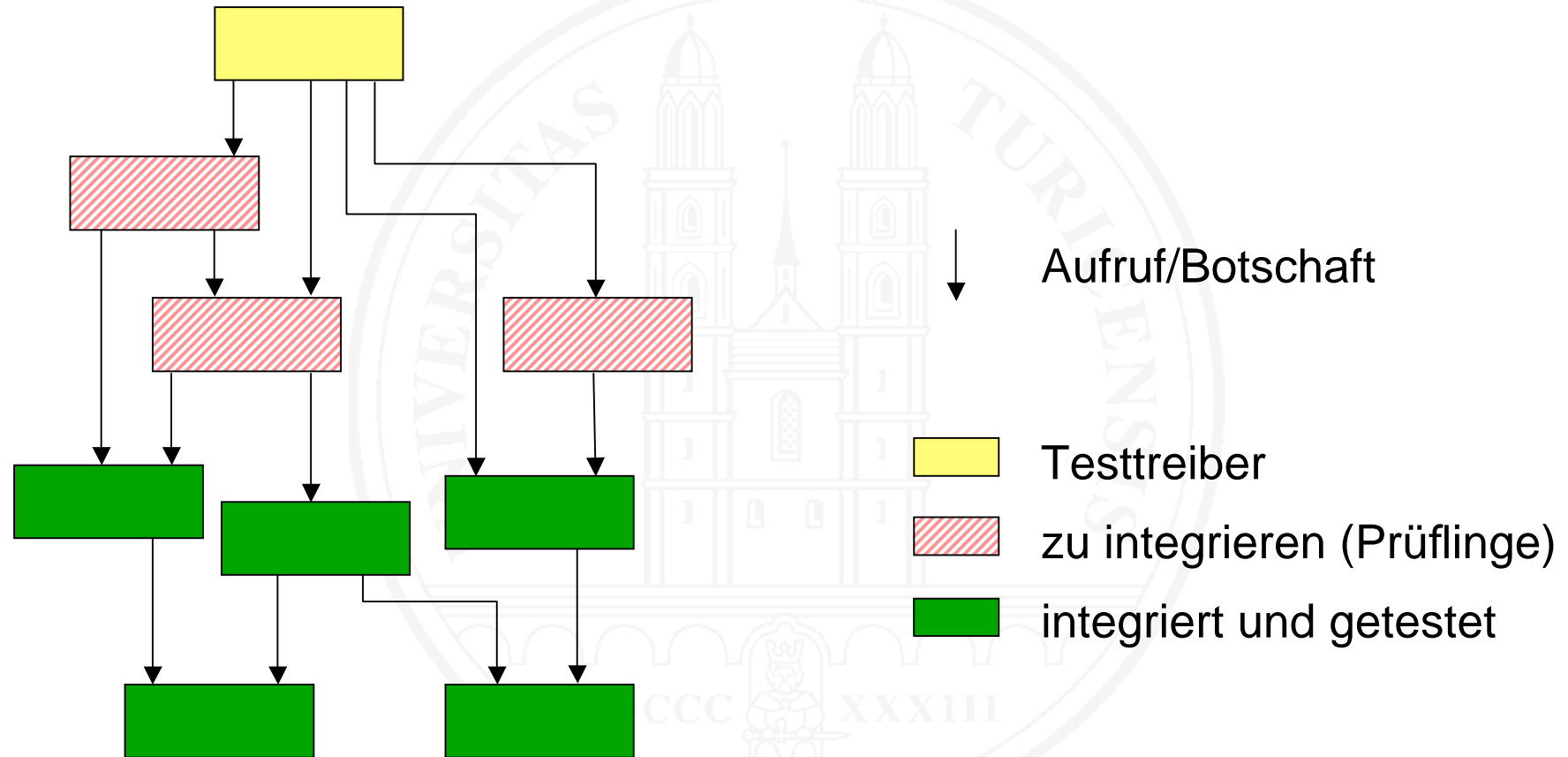
- Ein System **schrittweise** zusammenbauen
- Das **Funktionieren der Baugruppen** durch Tests überprüfen

## Ansätze:

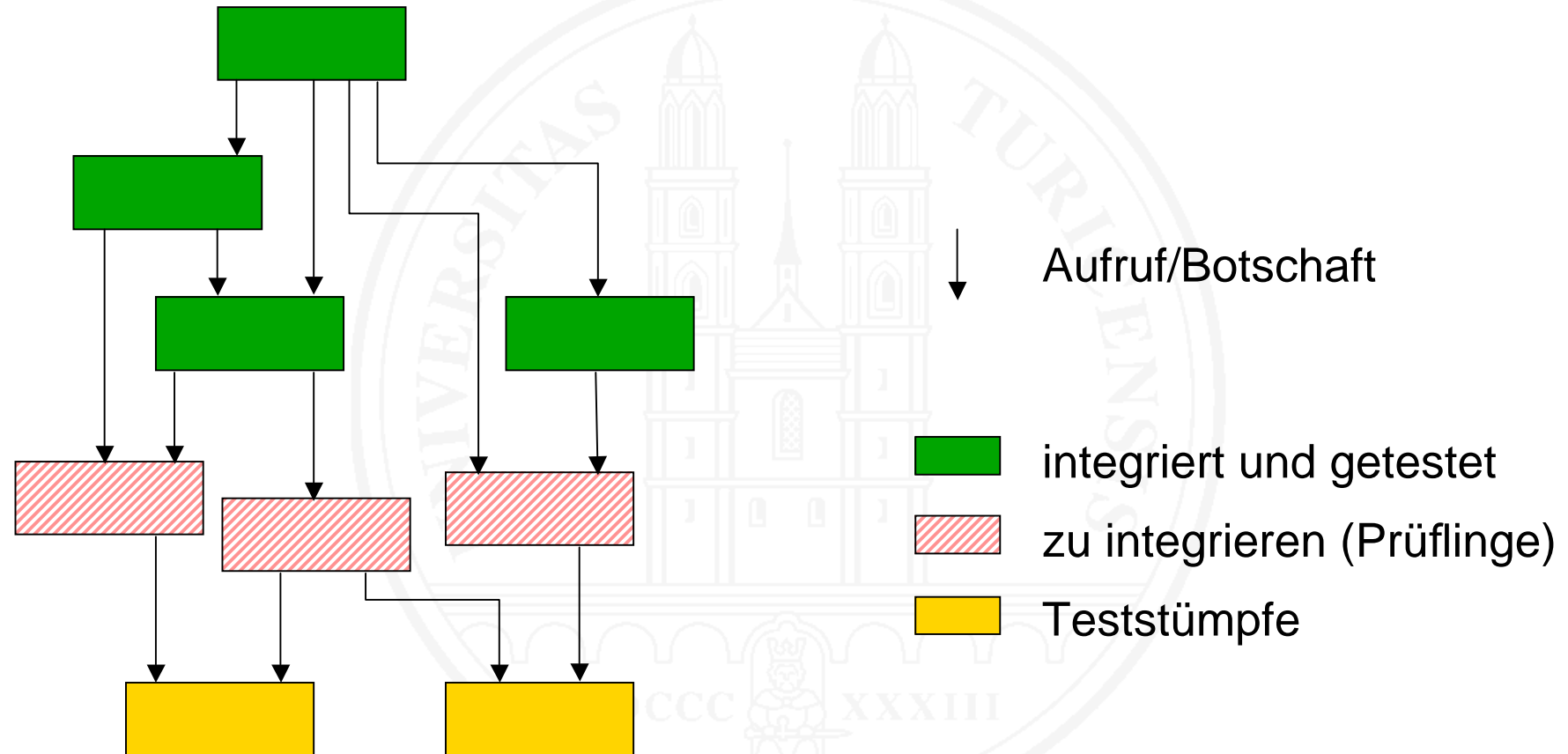
- **Aufwärtsintegration (bottom-up integration)**
  - Beginnt mit elementaren Komponenten
  - braucht keine Stümpfe, nur Treiber
- **Abwärtsintegration (top-down integration)**
  - Beginnt mit einem "hohlen" Gesamtsystem
  - Braucht keine Treiber, nur Stümpfe
- **Mischformen** sind möglich

# Aufwärtsintegration

---



# Abwärtsintegration





- 8.1 Grundlagen
- 8.2 Vorgehen
- 8.3 Testfälle
- 8.4 Testverfahren
- 8.5 Testplanung und -dokumentation
- 8.6 Testen von Teilsystemen
- 8.7 Besondere Testformen**

---

- 8.8 Kriterien für den Testabschluss

# Testen nicht-funktionaler Anforderungen

---

- Testen von **Leistungsanforderungen**
  - **Leistungstest** – Zeiten, Mengen, Raten, Intervalle
  - **Lasttest** – Verhalten bei (noch regulärer) Starklast
  - **Stresstest** – Verhalten bei Überlast
  - **Ressourcenverbrauch**
- Testen **besonderer Qualitäten**
- ⇒ Nur wenig ist testbar, zum Beispiel
  - **Zuverlässigkeit**
  - **Benutzbarkeit**
  - **Sicherheit (teilweise)**

# Testen von Benutzerschnittstellen

---

- **Funktionalität:** alle Funktionen über Dialog zugänglich?
- **Benutzbarkeit:** Bedienbarkeit, Erlernbarkeit, Anpassung an Kundenbedürfnisse
- **Dialogstruktur:** Vollständigkeit, Konsistenz, Redundanz, Metapherkonformität
- **Antwortzeitverhalten**

# Testen Web-basierter Benutzerschnittstellen

---

- Zusätzlich zu den Standardtests sind erforderlich:
  - **Linktest/URL-Test:** Alles am richtigen Ort? Richtig verknüpft? Zugänglich?
  - **Sicherheitstest**
  - **Zugangstest:** Sichtbarkeit, Erreichbarkeit, Verfügbarkeit
  - **Kompatibilitätstest:** unabhängig vom Browser?
- außerdem wichtig:
  - Lasttest
  - Stresstest

- 
- 8.1 Grundlagen
  - 8.2 Vorgehen
  - 8.3 Testfälle
  - 8.4 Testverfahren
  - 8.5 Testplanung und -dokumentation
  - 8.6 Testen von Teilsystemen
  - 8.7 Besondere Testformen
  - 8.8 Kriterien für den Testabschluss**
-

# Wann ist genug getestet? – Testabschlusskriterien

---

- Wenn mit den in der Testvorschrift festgelegten Testdatensätzen **keine Fehler mehr gefunden** werden
  - Sinnvolles Kriterium, wenn der Umfang des Prüflings eine systematische Auswahl von Testfällen mit ausreichender Überdeckung ermöglicht
  - Übliches Kriterium bei der Abnahme
- Wenn die **Prüfkosten pro entdecktem Fehler** eine im voraus festgelegte Grenze überschreiten
  - Sinnvolles Kriterium für das Beenden des Systemtests
  - Setzt die Erfassung der Prüfkosten und der Anzahl gefundener Fehler voraus

# Abschlusskriterien – 2

---

- Wenn während der **Ausführung einer im voraus festgelegten Menge von Testfällen** keine Fehler auftreten
  - Beispielsweise im Systemtest mit zufällig bestimmten Testdaten. Die Anzahl der hintereinander fehlerfrei auszuführenden Testfälle bestimmt sich aus der geforderten Zuverlässigkeit (Poore et al. 1993)
- Wenn die vorher festgelegte **Obergrenze für die Fehlerdichte unterschritten** wird
  - Muss mit statistischen Methoden bestimmt werden (Musa und Ackerman, 1989)

# Literatur

---

- Beck, K., E. Gamma (2002). *JUnit Cookbook*. <http://www.junit.org>, Stichwort Documentation
- Beizer, B. (1995). *Black-Box Testing*. New York, etc.: John Wiley & Sons.
- Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Prüfung. Eine Fibel*. Zürich: vdf und Stuttgart: Teubner.
- IEEE (1987). *Standard for Software Unit Testing*. ANSI/IEEE Std 1008-1987. IEEE Computer Society Press.
- IEEE (1988). *Standard Dictionary of Measures to Produce Reliable Software*. IEEE Std 982.1-1988. IEEE Computer Society Press.
- IEEE (1998a). *IEEE Standard for Software Test Documentation*. ANSI/IEEE Std 829-1998. IEEE Computer Society Press.
- IEEE (1998b). *Standard for Software Verification and Validation Plans*. ANSI/IEEE Std 1012-1998. IEEE Computer Society Press.
- Liggesmeyer, P. (1990). *Modultest und Modulverifikation*. BI-Wissenschaftsverlag, Reihe Angewandte Informatik Bd. 4, Mannheim etc.
- Liggesmeyer, P. (2002). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Berlin: Spektrum Akademischer Verlag.
- Musa, J.D., A.F. Ackerman (1989). Quantifying Software Validation: When to Stop Testing? *IEEE Software* **6**, 3 (May 1989). 19-27.



# Literatur – 2

---

Myers, G.J. (1979). *The Art of Software Testing*. New York, etc.: John Wiley & Sons. [in dt. Übersetzung: *Methodisches Testen von Programmen*. 4. Auflage. Oldenbourg, München, 1991.]

Pol, M., T. Koomen, A. Spillner (2000). *Management und Optimierung des Testprozesses*. Heidelberg: dpunkt.verlag.

Poore, J.H., H.D. Mills, D. Mutchler (1993). Planning and Certifying Software System Reliability. *IEEE Software* **10**, 1 (Jan 1993). 88-99.

Spillner, A., T. Linz (2002). *Basiswissen Softwaretest*. Heidelberg: dpunkt.Verlag

Zeller, A. (2006). *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco: Morgan Kaufmann und Heidelberg: dpunkt.verlag.

Im Skript [M. Glinz (2005). *Software Engineering*. Vorlesungsskript, Universität Zürich] lesen Sie Kapitel 9.5.

Im Begleittext zur Vorlesung [S.L. Pfleeger, J. Atlee (2006). *Software Engineering: Theory and Practice*, 3rd edition. Upper Saddle River, N.J.: Pearson Education International] lesen Sie Kapitel 8 und 9.

Hinweis: Im Gegensatz zu Pfleeger und Atlee betrachten wir Reviews (pp.377-379), Programmbeweise (pp. 380-384) und statische Analyse (pp. 402-403 ) *nicht* als Testverfahren, sondern als *eigenständige*, vom Testen abzugrenzende Prüfverfahren.