



# Vorlesung (WS 2014/15) *Softwarekonstruktion*

Prof. Dr. Jan Jürjens

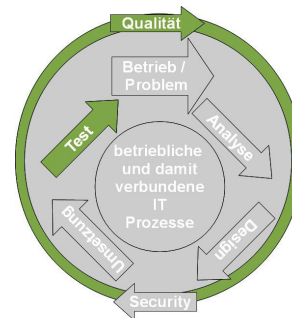
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

2.4: White-Box-Test

v. 20.02.2015



- Modellgetriebene SW-Entwicklung
- Qualitätsmanagement
- **Testen**
  - Grundlagen Softwareverifikation
  - Softwaremetriken
  - Black-Box-Test
  - **White-Box-Test**
  - Testen im Softwarelebenszyklus



[Basierend auf dem Foliensatz „Basiswissen Softwaretest - Certified Tester“ des „German Testing Board“, 2011]

**Literatur (s. Webseite):**

- Andreas Spillner, Tilo Linz: Basiswissen Softwaretest. **Kapitel 5.**
- Eike Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme. **Kapitel 7,8,9.**

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Kapitel 5

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>



- **Voheriger Abschnitt:** Dynamischer Test ohne Kenntnis der Programmlogik
- **Dieser Abschnitt:** Analyse interner Struktur des Testobjekts mit Hilfe:
  - Kontrollflussbasierter Test
  - Datenflussbasierter Test

## Was Sie wiederholen sollten!

Softwarekonzepte  
WS 2014/15



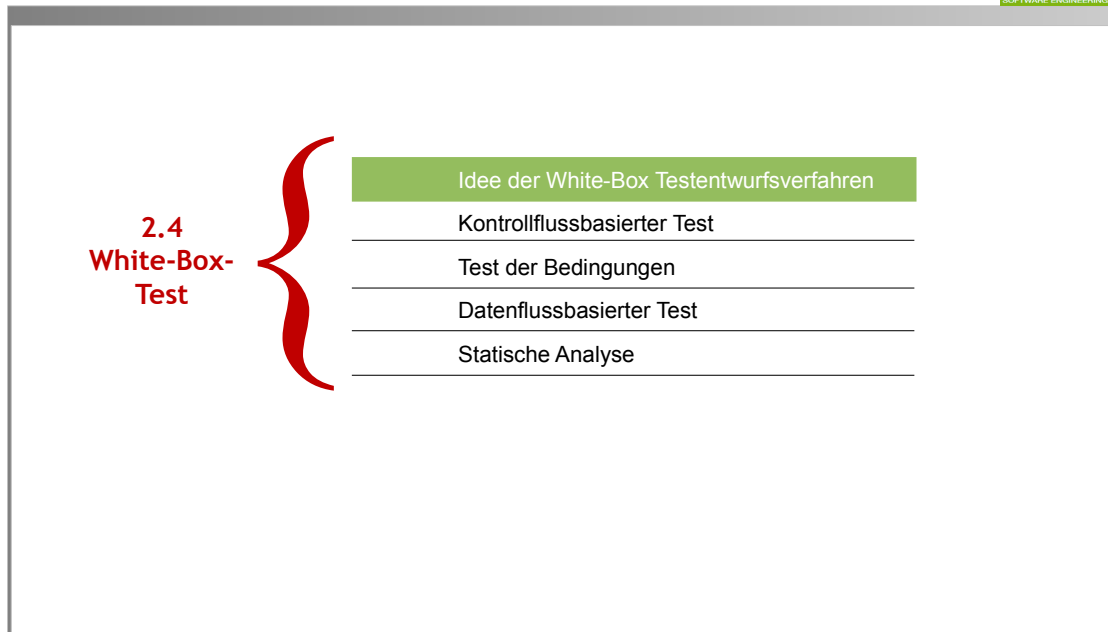
Wiederholung  
SWT

Was Sie für diesen Abschnitt von der Vorlesung SWT wiederholen sollten:

Testen mit Hilfe der

- **Anweisungsüberdeckung**
- **Zweigüberdeckung**
- **Pfadüberdeckung**
- **Mehrfachbedingungsüberdeckung**

Diese Themen werden wir noch einmal kurz behandeln, es ist aber sinnvoll sich die Methoden vorher noch einmal ins Gedächtnis zu rufen.



### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Kapitel 5

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>



**White-Box-Test:** Dynamisches Testverfahren, basiert auf Analyse interner Struktur der Testobjekts.

»**Fehleraufdeckende**« **Stichproben** möglicher Programmabläufe und Datenverwendungen suchen.

Zur

- Herleitung der Testfälle
- **Bestimmung der Vollständigkeit der Prüfung** (Überdeckungsgrad)

wird Information über innere Struktur des Testobjekts herangezogen.

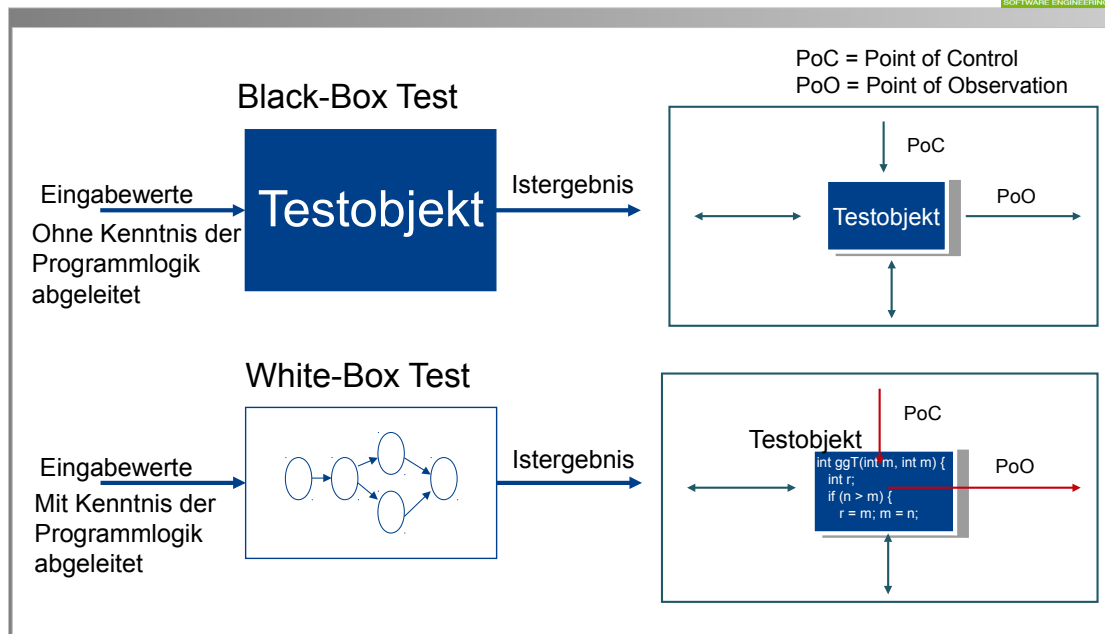
→ Strukturorientierte (strukturelle) Testentwurfsverfahren.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2 – Whitebox-Verfahren (S.149)



7

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5 (S.109-113)
- Abbildung 5.2 (S.113)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 335
- ab Folie 340



### 2.4 White-Box- Test



Idee der White-Box Testentwurfsverfahren

Kontrollflussbasierter Test

Test der Bedingungen

Datenflussbasierter Test

Statische Analyse





### Kontrollflussbezogenes Testen:

- **Dynamisches** Verfahren: Basiert auf Ausführung des Programms.
- **White-Box-Verfahren:** Nutzt Kenntnis der Programmstruktur aus.

### Analysemittel:

- **Kontrollflussgraph:** Verdeutlicht Kontrollfluss im Programm.

### Idee:

- **Auswahl der Testdaten:** Viele Durchläufe durch Kontrollfluss des Programms testen; dafür wenig Testfälle gebrauchen.

### Varianten, differenzieren nach:

- **Art** der verwendeten **Kontrollflusswege** oder **-wegstücke**.
- Art und Weise der Überdeckung.
- angestrebtem Überdeckungsgrad

9

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.1 (S.150-152)
- Abbildung 5.9 (S.150)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.1 – Kontrollflussgraphen (S.188-189)



### Kontrollflussgraph eines Programms P: **Gerichteter Graph**

$$G = (N, E)$$

mit 2 ausgezeichneten Knoten *nstart*, *nfinal*

- Knoten stellt **Anweisungen** / sequenzielle Anweisungsfolgen dar.
- Kante aus Menge der Kanten  $E \subseteq N \times N$  beschreibt möglichen **Kontrollfluss zwischen zwei Anweisungen**.
- Kante aus E auch **Zweig** genannt.
- Beide ausgezeichneten Knoten stellen **Anfangs- und Endeanweisung** eines Programms dar.

10

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.1 (S.150-152)
- Abbildung 5.9 (S.150)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.1 – Kontrollflussgraphen (S.188-189)



**Block:** Nichtleere Folge von Knoten.

- Nur durch ersten Knoten betretbar,
- Vom ersten Knoten ausgehend in vorgegebener Reihenfolge genau einmal deterministisch durchlaufbar.
- Maximal bezüglich ersten beiden Eigenschaften.

**Pfad / vollständiger Weg:**

- Folge von Knoten und Kanten, die mit Startknoten beginnt und Endknoten endet.

**Wege (T,P):**

- Menge vollständiger, endlicher Wege  $w$  des Kontrollflussgraphen, für die Testdatum  $t$  aus Menge der Testdaten  $T$  existiert, das Weg  $w$  ausführt.

11

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.1 – Kontrollflussgraphen (S.191)



- **Entscheidungsknoten:** Knoten mit mindestens 2 Nachfolgeknoten.
- **Entscheidungskanten:** Kanten, die ihren Ursprung in einem Entscheidungsknoten haben.
- **Zyklus:** Weg im Kontrollflussgraphen mit mindestens zwei Knoten, der an demselben Knoten beginnt und endet.
- **Einfacher Zyklus:** Zyklus, bei dem alle Knoten (außer Anfangs- und Endknoten) verschieden sind.

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.1 – Kontrollflussgraphen (S.188-191)

## Beispielprogramm: Funktion **ggT** ()



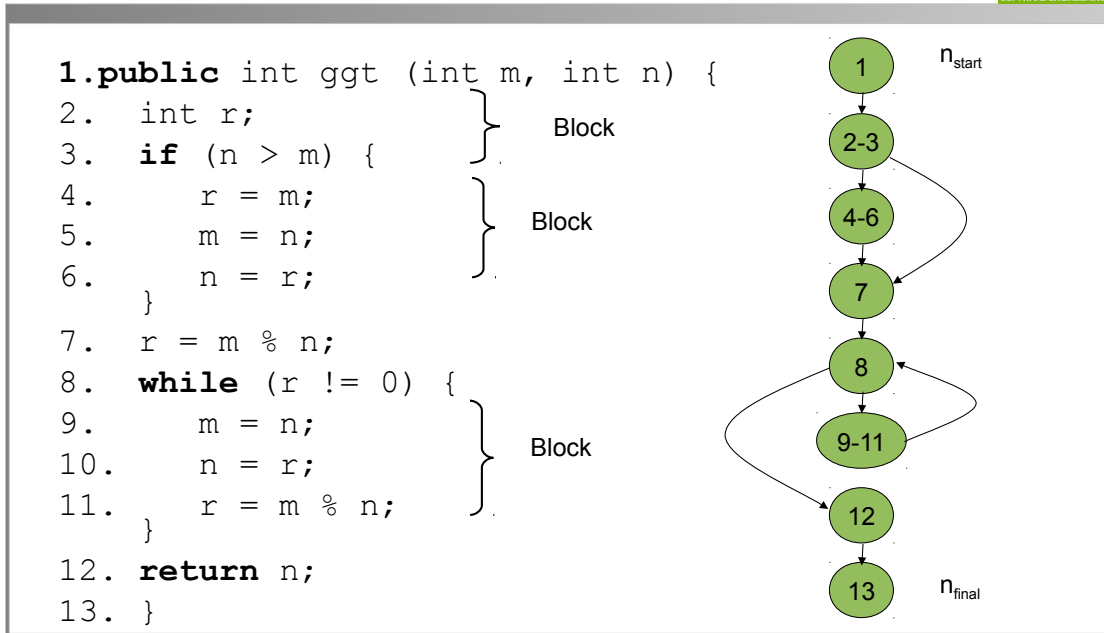
Bestimmung des **größten gemeinsamen Teilers (ggT)** zweier ganzer Zahlen  $m, n > 0$ :

$\text{ggT}(4,8)=4$ ;  $\text{ggT}(5,8)=1$ ;  $\text{ggT}(15,35)=5$

Spezifikation in Java (JML):

```
public int ggt(int m, int n) {  
  // pre: m > 0 and n > 0  
  // post: return > 0 and  
  // m@pre.mod(return) = 0 and  
  // n@pre.mod(return) = 0 and  
  // forall(i : int | i > return implies  
  //      (m@pre.mod(i) > 0 or n@pre.mod(i) > 0)  
  ... }  
}
```

# Beispielprogramm: Kontrollflussgraph von `ggT()`





**Zunächst:** Bestimmung der Pfade durch Kontrollflussgraphen, die durch Testfälle »zur Ausführung gebracht werden sollen«.

**Frage:** Mit welchen Eingaben werden diese Pfade erzwungen ?

**Idee:**

- Bedingungen **kontrollflussbestimmender Anweisungen** betrachten.
- Damit **Aussagen über Programmvariablen** »berechnen«.

**Achtung:** Nicht alle im Graphen vorhandenen Pfade (Zweige, Wegstücke, ...) können für das gegebene Programm tatsächlich ausgeführt werden (z.B. nicht erreichbare Programmstücke; nicht erreichbare Anzahl von Schleifendurchläufen), d.h. es kann Pfade ohne zugehörigen Testfall geben.

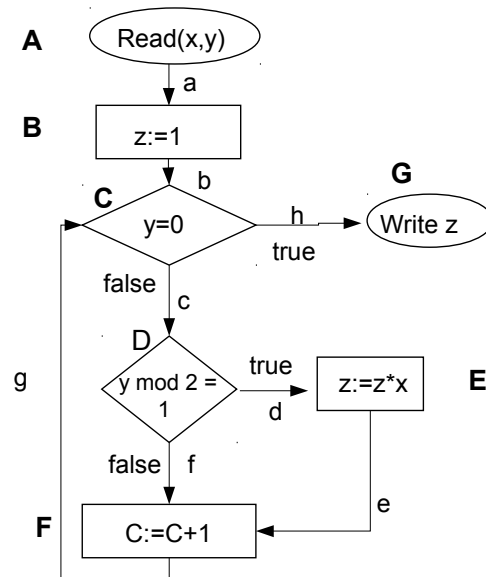
15



## Entscheidungs-Entscheidungsweg:

- Wegstück, welches bei **Entscheidungsknoten** oder **Anfangsknoten** beginnt
- Und alle folgenden Knoten und Kanten bis zum nächsten **Entscheidungsknoten** bzw. bis zum **Endknoten** des Kontrollflussgraphen (einschließlich) enthält.

Was sind hier die Entscheidungs-Entscheidungswege ?



16

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.1 – Kontrollflussgraphen (S.189-190)

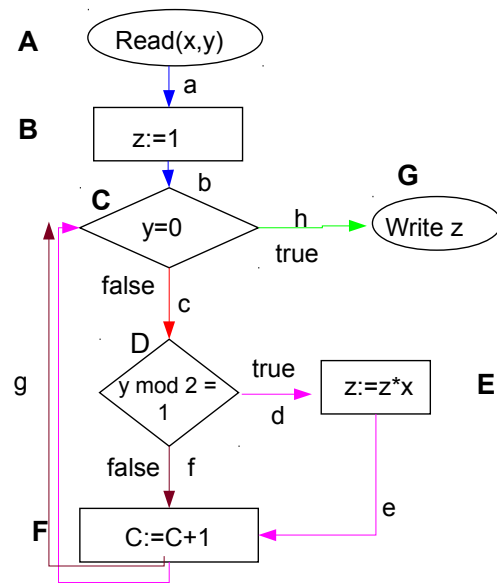




## Entscheidungs-Entscheidungsweg:

- Wegstück, welches bei **Entscheidungsknoten** oder **Anfangsknoten** beginnt
- Und alle folgenden Knoten und Kanten bis zum nächsten **Entscheidungsknoten** bzw. bis zum **Endknoten** des Kontrollflussgraphen (einschließlich) enthält.

Hier: {A,B,C}; {C,G}; {C,D}; {D,E,F,C}; {D,F,C}



17

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

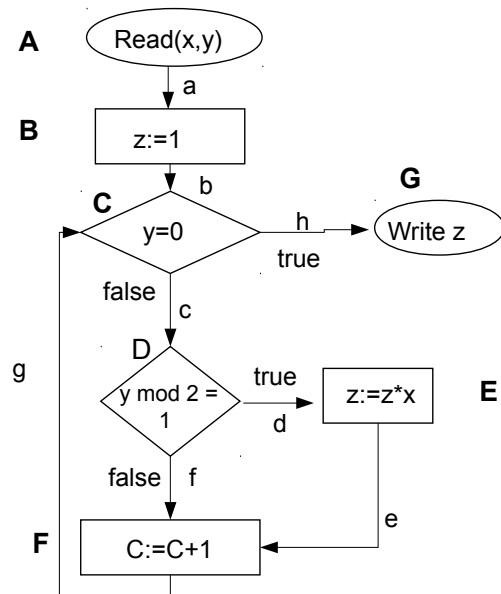
Abschnitt 7.1 – Kontrollflussgraphen (S.189-190)



**Segment:** Wegstück mit Eigenschaften:

- **Erster Knoten** des Wegstücks: Anfangsknoten des Kontrollflussgraphen oder Entscheidungsknoten oder Vereinigungsknoten (mehrere Inputs).
- **Letzter Knoten** des Wegstücks: Endknoten des Kontrollflussgraphen oder Entscheidungsknoten oder Vereinigungsknoten.
- Alle **andere Knoten** haben nur Eingangs- und Ausgangskante.

Was sind hier die Segmente ?



18

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

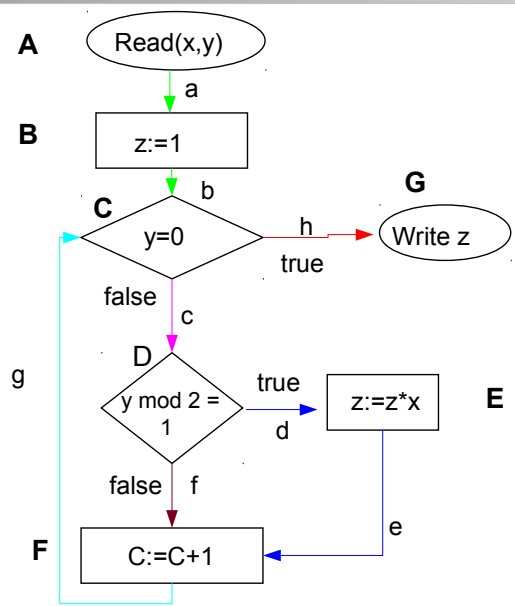
Abschnitt 7.1 – Kontrollflussgraphen (S.189-190)



**Segment:** Wegstück mit Eigenschaften:

- **Erster Knoten** des Wegstücks: Anfangsknoten des Kontrollflussgraphen oder Entscheidungsknoten oder Vereinigungsknoten (mehrere Inputs).
- **Letzter Knoten** des Wegstücks: Endknoten des Kontrollflussgraphen oder Entscheidungsknoten oder Vereinigungsknoten.
- Alle **andere Knoten** haben nur Eingangs- und Ausgangskante.

Hier: {A,B,C}; {C,G}; {C,D}; {D,E,F};  
{D,F}; {F,C}



19

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

Abschnitt 7.1 – Kontrollflussgraphen (S.189-190)



Durch kontrollflussbezogenes Verfahren aufdeckbare Fehler:

### **Berechnungsfehler:**

- Richtiger Kontrollflussweg im Programm ausgeführt, aber min. **ein berechneter Variablenwert falsch**.

### **Bereichsfehler:**

- **Falscher Kontrollflussweg** im Programm ausgeführt.  
(→ Eingabebereich des vorliegenden Kontrollflusses stimmt nicht mit Eingabebereich im korrekten Programm überein.)

### **Unterbereichsfehler:**

- Spezielle Bereichsfehler.
- „**Zuviel / zuwenig**“ **Kontrollfluss**:  
→ Abfrage fehlt oder es gibt zusätzliche, falsche Abfrage.

20

## **Literatur:**

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2 – kontrollflussbezogenes Testen (S.195)



- **Anweisungsüberdeckung**  
(*statement coverage;  $C_0$ -Überdeckung; alle Knoten*).
- **Entscheidungs-/Zweigüberdeckung**  
(*decision coverage;  $C_1$ -Überdeckung, alle Zweige*).
- **Grenze-Inneres-Test**  
(*boundary interior coverage*).
- **Pfadüberdeckung**  
(*path coverage;  $C_\infty$ -Überdeckung, alle Pfade*).

21

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Anweisungsüberdeckung (S.151)
- Zweigüberdeckung (S.153-154)
- Pfadtest (S.160)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)
- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)
- Abschnitt 7.2.4 – Schleifenüberdeckung (S.202-208)
  - Pfadüberdeckung (S.203-204)
  - Grenze-Inneres-Überdeckung (S.204-206)

### Anweisungsüberdeckung ( $C_0$ -Test):

- Testdatenmenge  $T$  erfüllt  $C_0$ -Überdeckung für Programm  $P$  g.d.w. es für **jede Anweisung  $A$**  des Programms  $P$  mind. ein **Testdatum  $t$**  aus  $T$  gibt, das Anweisung  $A$  **ausführt**.
- Anweisung  $A$  unter  $T$  ausgeführt g.d.w. der zur **Anweisung  $A$**  gehörende **Knoten  $k$**  in mind. einem **Weg** in  $Wege(T)$  vorkommt.
- **Testwirksamkeitsmaß  $TWM_0$** : Anweisungsüberdeckungsgrad (Verhältnis besuchter Knoten zur Gesamtzahl von Knoten):

$$TWM_0 =_{\text{def}} \frac{\text{Zahl der unter } T \text{ überdeckten Anweisungen}}{\text{Zahl aller Anweisungen}}$$

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Anweisungsüberdeckung (S.151)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)

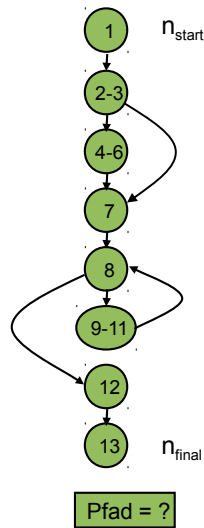
S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350

## Beispiel: Anweisungsüberdeckung für ggt ()

Softwarekonstruktion  
WS 2014/15



```
1. public int ggt(int m, int n) {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
7.     }
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
12.    }
13.    return n;
14. }
```

Beispiel für Pfad mit Anweisungsüberdeckung ? Zugehörige Testdaten ?

23

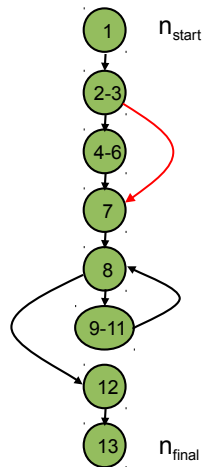
S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350

## Beispiel: Anweisungsüberdeckung für ggt () : Pfad

Softwarekonstruktion  
WS 2014/15



```
1. public int ggt(int m, int n) {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
7.     }
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
12.    }
13.    return n;
14. }
```

Pfad = (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

(Roter Zweig wird nicht ausgeführt.)

24

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350



## Beispiel: Anweisungsüberdeckung für `ggt()` : Testdaten

Softwarekonstruktion  
WS 2014/15



**Pfad:** (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

**Logischer Testfall:**  $\{ n > m \wedge n \bmod m \neq 0 \wedge n \bmod (n \bmod m) = 0 ; \text{ggt}(m,n) \}$

**Konkreter Beispiel-Testfall:**  
 $\{ m = 1, n = 2; \text{ggt}(m,n)=2 \}$

Oder:  
 $\{ m = 4, n = 6; \text{ggt}(m,n)=2 \}$

m	n	r
4	6	2

```
1. public int ggt(int m, int n) {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
7.     }
8.     r = m % n;
9.     while (r != 0) {
10.        m = n;
11.        n = r;
12.        r = m % n;
13.    }
14.    return n;
15. }
```

25

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350



**100%ige Anweisungsüberdeckung** nicht immer erreichbar.

- Z.B.: Ausnahmebedingungen kommen im Programm vor, die während Testphase mit erheblichem Aufwand oder gar nicht herzustellen sind.
- Kann auf **nicht erreichbare Anweisungen** („dead code“) hindeuten (ggf. statische Analyse durchführen).

26

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Anweisungsüberdeckung (S.151)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)

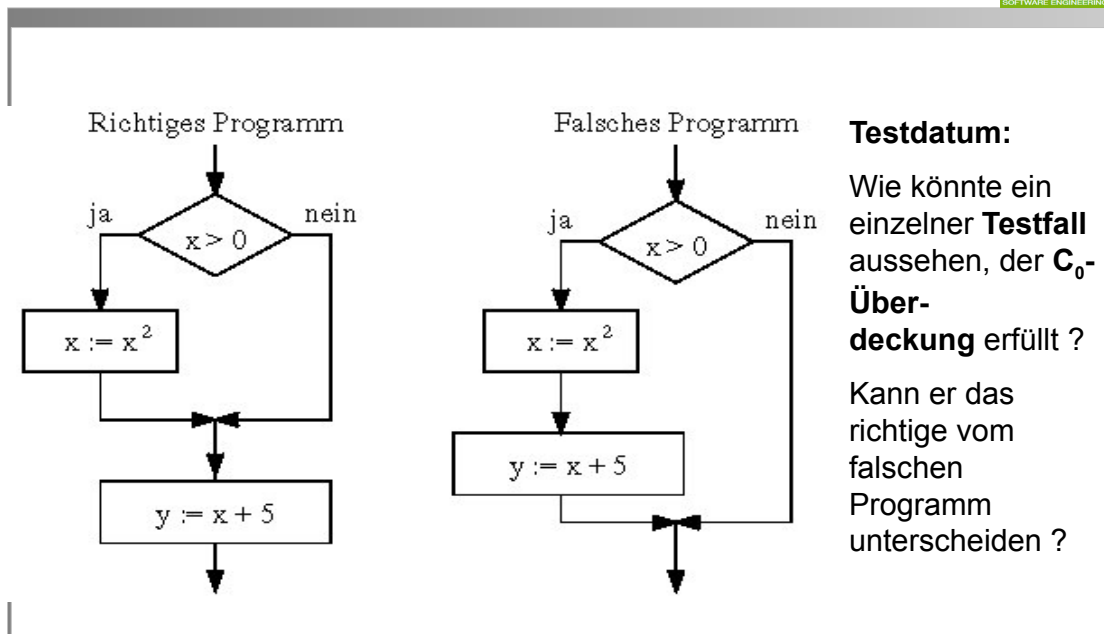
S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350

## Anweisungsüberdeckung: Aussagekraft (Beispiel)

Softwarekonstruktion  
WS 2014/15



27

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)
- Abbildung 7.4 (S.197)

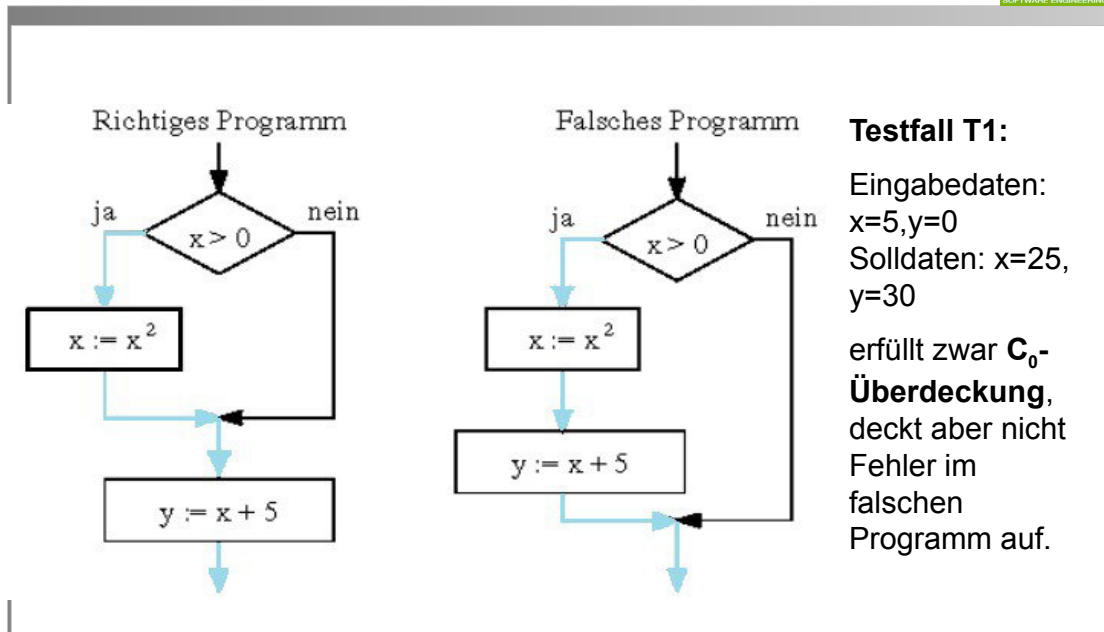
S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350

## Anweisungsüberdeckung: Aussagekraft (Beispiel)

Softwarekonstruktion  
WS 2014/15



28

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)
- Abbildung 7.4 (S.197)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350



### Schwaches Kriterium, z.B. weil:

• „Leere“ Kante (Kante überbrückt Knoten) nicht berücksichtigt.

#### • Beispiele:

- ELSE-Kante (zwischen IF und ENDIF) mit leerem ELSE-Teil (siehe Beispiel oben)
- Rücksprung zum Anfang einer Repeat-Schleife
- BREAK.
- **Fehlende** Anweisungen nicht erkannt !

### Bewertung:

- Geringe Zahl von Eingabedaten.
  - Notwendiges, aber **kein hinreichendes Testkriterium**.
  - **Mangelhafte Aussagekraft:** Nicht-ausführbare Programmteile werden entdeckt, alle anderen Fehler nur zufällig entdeckt.
- => Stärkeres Kriterium notwendig: Zweigüberdeckung !

29

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Anweisungsüberdeckung (S.151)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 345-350

## Zweigüberdeckung ( $C_1$ -Test, auch genannt Entscheidungsüberdeckung):

- Testdatenmenge  $T$  erfüllt  $C_1$ -Überdeckung für Programm  $P$ , g.d.w. es für jede Kante  $k$  im Kontrollflussgraphen von  $P$  mind. einen Weg in  $Wege(T,P)$  gibt, zu dem  $k$  gehört.
- Testwirksamkeitsmaß  $C_{\text{Zweig}}$  (Zweigüberdeckungsgrad, Verhältnis der besuchten Zweige zur Gesamtzahl von Zweigen):

$$C_{\text{Zweig}} =_{\text{def}} \frac{\text{Anzahl der besuchten Zweige}}{\text{Gesamtanzahl der Zweige}}$$

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Zweigüberdeckung (S.153-154)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354



- Ein anderes **Testwirksamkeitsmaß** für die **C<sub>1</sub>-Überdeckung**

$$TWM_1 =_{def} \frac{\text{Anzahl der überdeckten Entscheidungskanten}}{\text{Zahl aller Entscheidungskanten}}$$

- Im Gegensatz zum Testwirksamkeitsmaß C<sub>Zweig</sub> werden nicht die Zweige gezählt, sondern die **Entscheidungskanten** gezählt.  
→ Werte der Maße können unterschiedlich sein

31

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Zweigüberdeckung (S.153-154)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354

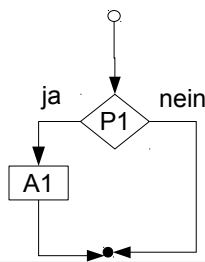


- Ein anderes **Testwirksamkeitsmaß** für die  $C_1$ -Überdeckung

$$TWM_1 =_{def} \frac{\text{Anzahl der überdeckten Entscheidungskanten}}{\text{Zahl aller Entscheidungskanten}}$$

- Im Gegensatz zum Testwirksamkeitsmaß  $C_{\text{Zweig}}$  werden nicht die Zweige gezählt, sondern die **Entscheidungskanten** gezählt.

→ Werte der Maße können unterschiedlich sein



**Frage:** Testwirksamkeitsmaß, wenn beim Test nur der „ja“-Zweig durchlaufen wird?

$$TWM_1 = ?$$

$$C_{\text{Zweig}} = ?$$

32

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Zweigüberdeckung (S.153-154)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354



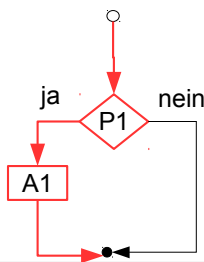


- Ein anderes **Testwirksamkeitsmaß** für die  $C_1$ -Überdeckung

$$TWM_1 =_{def} \frac{\text{Anzahl der überdeckten Entscheidungskanten}}{\text{Zahl aller Entscheidungskanten}}$$

- Im Gegensatz zum Testwirksamkeitsmaß  $C_{\text{Zweig}}$  werden nicht die Zweige gezählt, sondern die **Entscheidungskanten** gezählt.

→ Werte der Maße können unterschiedlich sein



**Frage:** Testwirksamkeitsmaß, wenn beim Test nur der „ja“-Zweig durchlaufen wird?

$$TWM_1 = 50\%$$

$$C_{\text{Zweig}} = 75\%$$

→ **Aussagekraft** von Testwirksamkeitsmaßen?

33

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Zweigüberdeckung (S.153-154)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

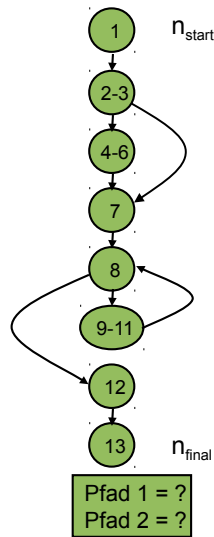
- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354

# Beispiel: Zweigüberdeckung für ggt ()



```
1. public int ggt (int m, int n)
   {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
   }
7.   r = m % n;
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
   }
12.  return n;
13. }
```

Entscheidung

Beispiel für Pfadmenge mit Zweigüberdeckung ? Testdaten ?

# Beispiel: Zweigüberdeckung für ggt () : Pfadmenge



```

1. public int ggt (int m, int n)
   {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
7.     }
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
12.    }
13.    return n;
   }
                
```

Pfad 1 = (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)  
 Pfad 2 = (1, 2-3, 7, 8, 12, 13)

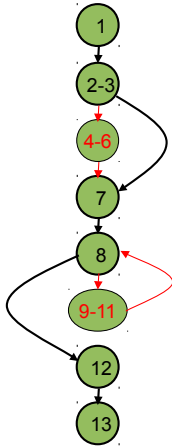
# Beispiel: Zweigüberdeckung für `ggt()`: Testdaten



**Pfad:** (1, 2-3, 7, 8, 12, 13)

**Logischer Testfall:**  $\{n \leq m \wedge m \bmod n = 0\}$   
`ggt(m, n)`

**Konkreter Testfall:**  $\{m = 4, n = 4; 4\}$



m	n	r
4	4	4

```
1. public int ggt(int m, int n) {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
7.     }
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
12.    }
13.    return n;
14. }
```

36



### Bewertung:

- Geringe Zahl von Eingabedaten.
- **Aussagekraft besser als  $C_0$** : Nicht-ausführbare Knoten und Zweige sicher entdeckt, alle anderen Fehler nur zufällig.
- **Durchlaufene Programmteile erkennbar** und ggf. optimierbar.
- Wie üblich: toter Code → keine 100%ige Überdeckung.
- **Unzureichend für Test von Schleifen.** (→ Grenze-Inneres-Überdeckung.)
- Abhängigkeiten zwischen Zweigen nicht berücksichtigt.  
(→ Pfadüberdeckungstest.)
- Kein Test komplexer, zusammengesetzter Bedingungen.  
(→ Bedingungsüberdeckungstest.)
- **Empirische Untersuchungen:** Bei 100%iger Abdeckung nur 25% der Fehler erkennbar.  
→ Aber immerhin noch besser als völlig unsystematisches Testen !

37

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Zweigüberdeckung (S.153-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

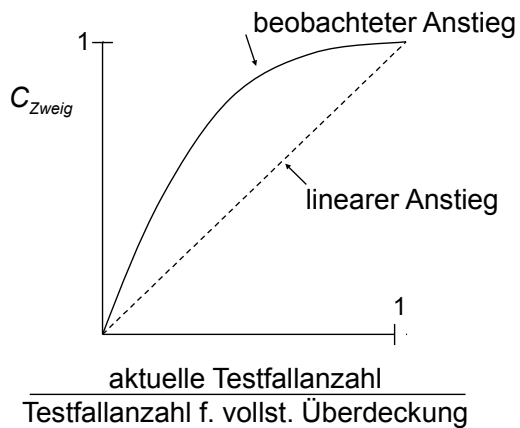
- Folie 352-354

## Zweigüberdeckungsrate vs. Anzahl Testfälle

Softwarekonstruktion  
WS 2014/15



**Zweigüberdeckungsrate** als Funktion der Testfallanzahl  
(gilt in ähnlicher Weise für andere Überdeckungsmaße) [Lig90]:



Empirisch beobachteter  
Zusammenhang (keine streng  
mathematische Notwendigkeit).

**Zur Diskussion:  
Was bedeutet das ?**

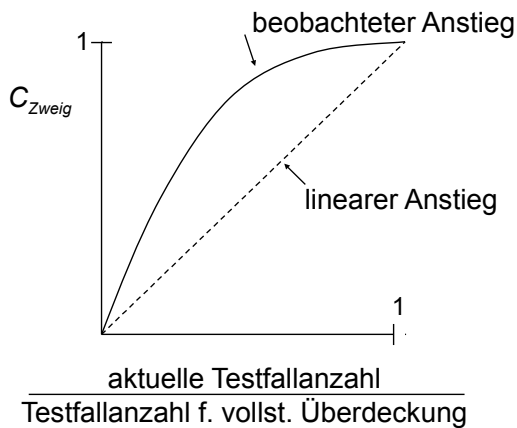
- S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf  
<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>  
• Folie 352-354

## Zweigüberdeckungsrate vs. Anzahl Testfälle

Softwarekonstruktion  
WS 2014/15



**Zweigüberdeckungsrate** als Funktion der Testfallanzahl  
(gilt in ähnlicher Weise für andere Überdeckungsmaße) [Lig90]:



Empirisch beobachteter  
Zusammenhang (keine streng  
mathematische Notwendigkeit).

### Zur Diskussion:

#### Was bedeutet das ?

→ Anfangs erhöht sich  $C_{\text{Zweig}}$   
bei Erhöhung der Testanzahl  
schnell, später langsam.  
(Vgl. „80-20-Regel“.)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354

## Diskussionsfrage: Anweisungsüberdeckung vs. Zweigüberdeckung

Softwarekonstruktion  
WS 2014/15



1) Kann man mit einer **100%iger Anweisungsüberdeckung** eine **100%ige Zweigüberdeckung garantieren?**

2) **Und umgekehrt ?**

Antwort:

1)

2)

40

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.1 – Anweisungstest (S.150-152)
- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)
- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354





1) Kann man mit einer **100%iger Anweisungsüberdeckung** eine **100%ige Zweigüberdeckung garantieren?**

2) **Und umgekehrt ?**

Antwort:

1) **Nein.** Vgl. Beispiel auf Folie 25/26.

2) **Ja:** Mit 100%iger Zweigüberdeckung kann man 100%ige Anweisungsüberdeckung garantieren.

- Beim Durchlauf aller Zweige werden alle Anweisungen überdeckt.

41

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.1 – Anweisungstest (S.150-152)
- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

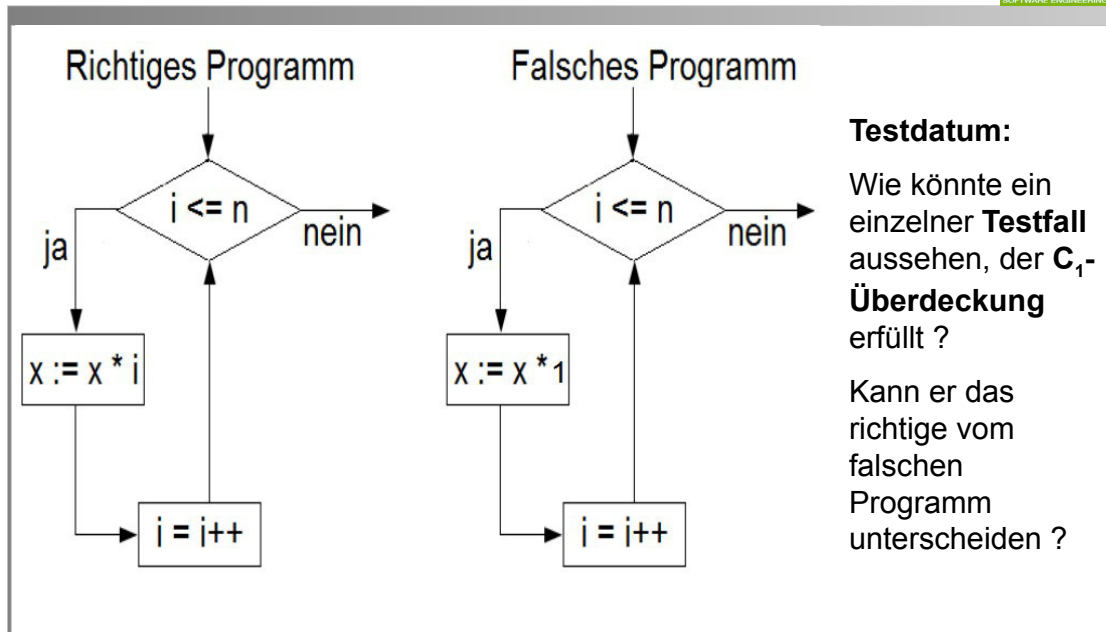
- Abschnitt 7.2.1 – Anweisungsüberdeckung (S.195-196)
- Abschnitt 7.2.2 – Zweigüberdeckung (S.196-198)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

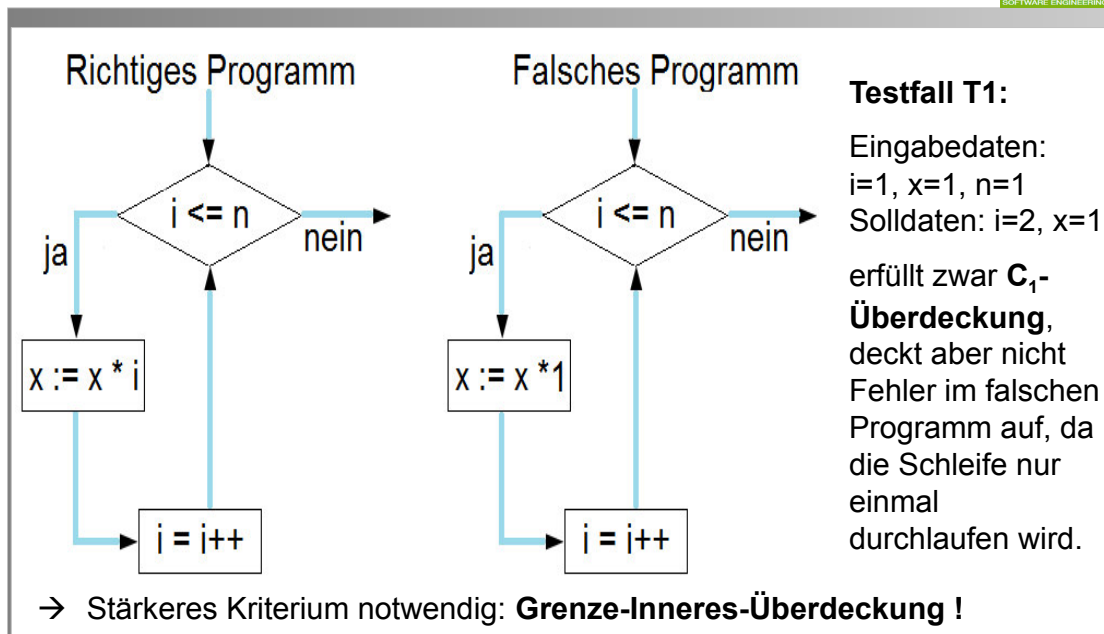
<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 352-354

# Zweigüberdeckung Aussagekraft (Beispiel)



# Zweigüberdeckung Aussagekraft (Beispiel)





**Grenze-Inneres-Überdeckung (gi):** Dynamisches, kontrollfluss-basiertes Testentwurfsverfahren. Fordert, dass **jede Schleife:**

- **genau einmal** ausgeführt wird und
- **mehr als einmal** ausgeführt wird.

Außerdem soll **jede abweisende Schleife** in min. einem Testfall **gar nicht** ausgeführt werden. Dies sind:

- **While, for**

(nur wenn Abbruchbedingung bei erstmaliger Auswertung wahr).

Zugehöriger Überdeckungsgrad:

$$\text{Grenze-Inneres-Überdeckungsgrad} = \frac{\text{Anzahl (gi) getestete Schleifen}}{\text{Gesamtzahl Schleifen}}$$

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

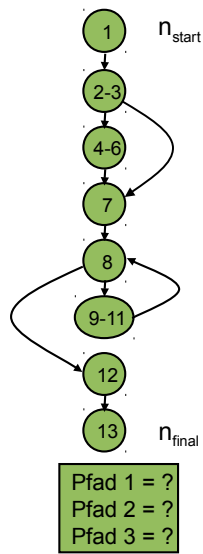
<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.4 – Schleifenüberdeckung (S.202-208)
  - Pfadüberdeckung (S.203-204)
  - Grenze-Inneres-Überdeckung (S.204-206)



```

1. public int ggT (int m, int n)
   {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
       }
7.     r = m % n;
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
       }
12.    return n;
13. }
    
```

Beispiel für Pfadmenge mit Grenze-Inneres-Überdeckung ?



```

1. public int ggt (int m, int n)
{
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
7.   }
8.   r = m % n;
9.   while (r != 0) {
10.    m = n;
11.    n = r;
12.    r = m % n;
13.  }
14.  return n;
15. }
    
```

Pfad 1 = (1, 2-3, 7, 8, 12, 13)  
 Pfad 2 = (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)  
 Pfad 3 = (1, 2-3, 4-6, 7, 8, 9-11, 8, 9-11, 8, 12, 13)



- In Aussage auf **Schleifen** beschränktes Kriterium.
  - Verlangt nur, Schleifen auf bestimmte Art zu testen.
  - Berücksichtigt keine Programmteile **außerhalb** von Schleifen.
  - Erkennt z.B. fehlende Anweisungen in „**leeren**“ **Zweigen** nicht !
  - „Mehr als einmal“ ausführen findet ggf. nicht Fehler, die erst ab bestimmter Durchlaufzahl manifest (s. nächste Folie).
- Daher in Praxis höchstens als **ergänzendes** Kriterium verwenden !
- Einzelne Schleifen **unabhängig voneinander** betrachtet (vgl. Zweige in Zweigüberdeckung).
- Für **verschachtelte Schleifen** gibt es spezialisierte, in ihrer Stärke abgestufte Überdeckungsmaße.

47

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

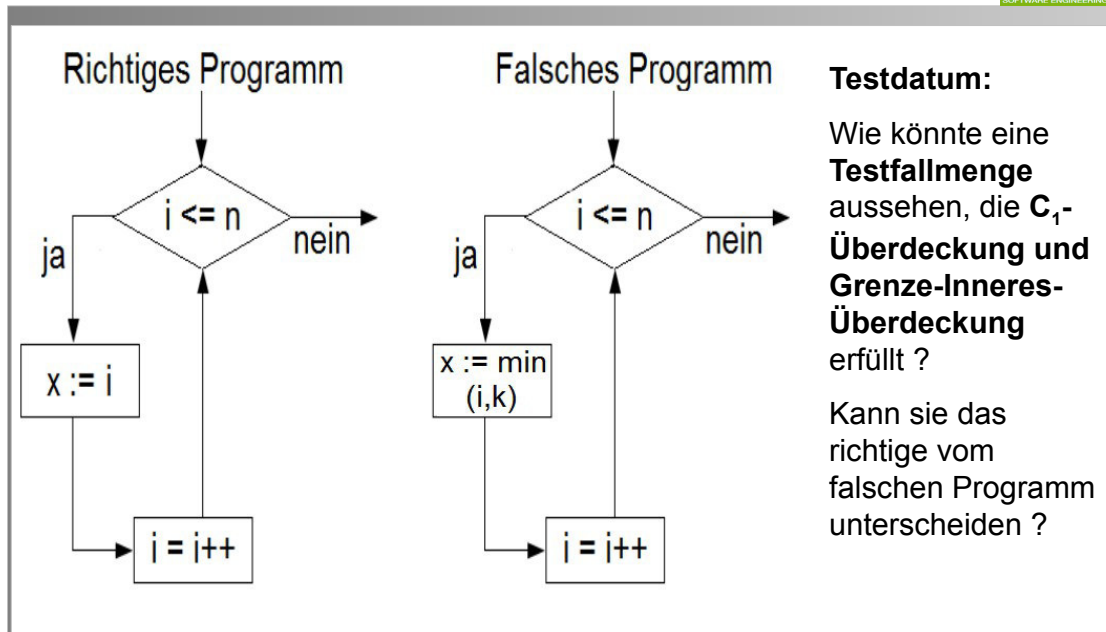
- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

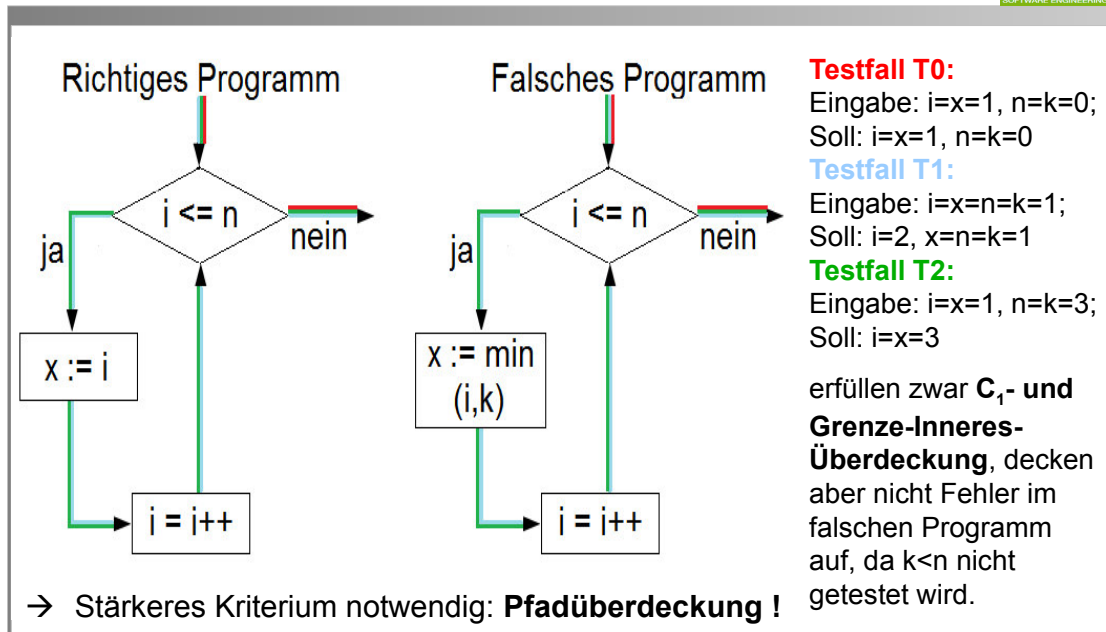
- Abschnitt 7.2.4 – Schleifenüberdeckung (S.202-208)
  - Pfadüberdeckung (S.203-204)
  - Grenze-Inneres-Überdeckung (S.204-206)

# Grenze-Inneres-Überdeckung Aussagekraft (Beispiel)





# Grenze-Inneres-Überdeckung Aussagekraft (Beispiel)



- Dynamisches, kontrollflussbasiertes Testentwurfsverfahren.
- Fordert **min. einmalige Ausführung jedes Pfads** im Kontrollflussgraphen.
- Auch als **C<sub>∞</sub>-Maß** bezeichnet.

$$\text{Pfadüberdeckungsgrad} = \frac{\text{Anzahl durchlaufene Pfade}}{\text{Gesamtzahl Pfade}}$$

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Pfadtest (S.160)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

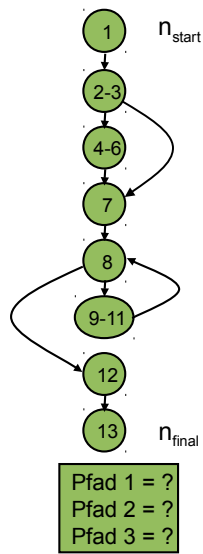
- Abschnitt 7.2.4 – Schleifenüberdeckung (S.202-208)  
→ Pfadüberdeckung (S.203-204)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 362-369

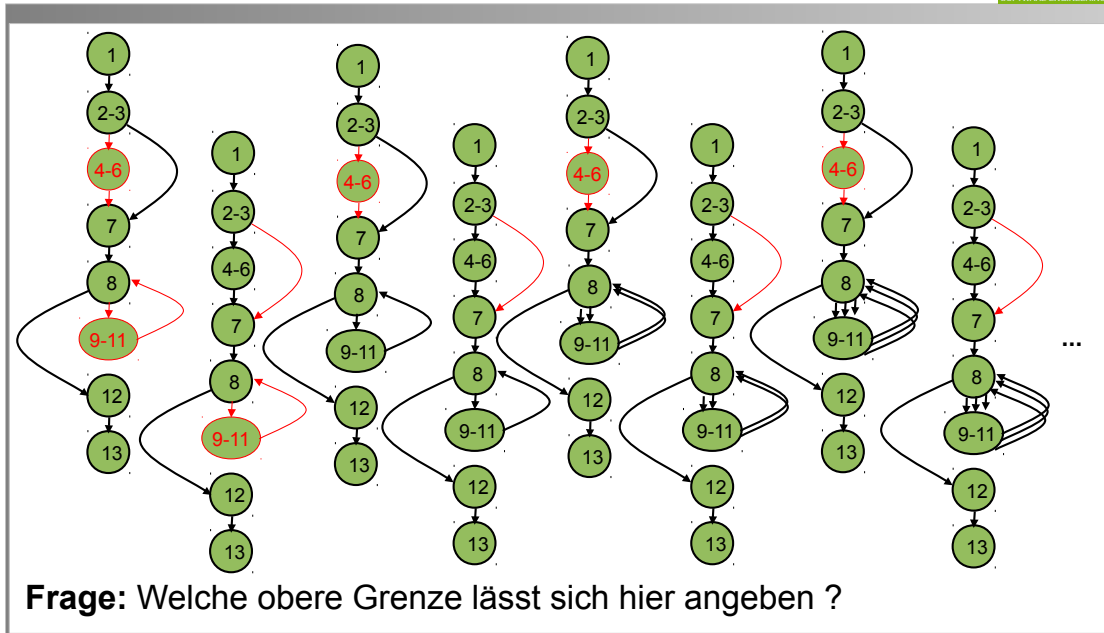
# Pfadüberdeckung für `ggt()`



```
1. public int ggt (int m, int n)
   {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
       }
7.     r = m % n;
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
       }
12.    return n;
13. }
```

Beispiel für Pfadmenge mit Pfadüberdeckung ?

# Beispiel: Pfadüberdeckung für ggt ()



52

## Pfadüberdeckung für `ggt()`: Obere Grenze



**Grobe obere Grenze** für maximale Anzahl Schleifendurchläufe in Abhängigkeit von Eingaben  $n, m$ :

• **Zeile 3-6:** ggf.  $m, n$  vertauschen  
→ betrachte  $\max(n, m)$  als obere Grenze für  $m, n$

• **Zeile 8-11:** Es gilt:  $\text{mod}(x, y) < \min(x, y)$ .  
→  $r$  in jedem Schleifendurchlauf strikt kleiner als im Durchlauf davor.  
→ höchstens  $\max(n, m)$  Durchläufe.  
→ höchstens  $\max(n, m)$  Durchläufe.

```
1. public int ggt (int m, int n)
   {
2.     int r;
3.     if (n > m) {
4.         r = m;
5.         m = n;
6.         n = r;
       }
7.     r = m % n;
8.     while (r != 0) {
9.         m = n;
10.        n = r;
11.        r = m % n;
       }
12.    return n;
13. }
```



**Frage:**

- Gibt es Programm, zu dem keine obere Grenze für Schleifenabbruch (und damit Pfadüberdeckung) angegeben werden kann ?



**Frage:** Programm ohne obere Grenze für Schleifenabbruch ?

**Antwort:**

**Beispiel:** Schleife, deren Abbruchwert von (echtem) Zufallszahlengenerator erzeugt wird (Benutzereingaben, radioaktiver Zerfall, ...):

Kann beliebig lange Wert false generieren, bevor true generiert wird.

→ **keine Grenze für Pfadüberdeckung**, wenn Schleifenabbruch abhängig von **externem** Nicht-Determinismus.



Warum werden durch **100%-ige Pfadüberdeckung**, die ja 100% der Pfade testet, **nicht 100% der Fehler gefunden** ?

**Antwort:**

- 
-





Warum werden durch **100%-ige Pfadüberdeckung**, die ja 100% der Pfade testet, **nicht 100% der Fehler gefunden** ?

**Antwort:**

- Es werden zwar alle Pfade getestet, aber **nicht mit allen möglichen Variablenbelegungen**.
- Es wird nur **vorhandene Funktionalität getestet** und nicht, welche Funktionalität evtl. fehlt.

## Pfadüberdeckung: Zusammenfassung der Bewertung

Softwarekonstruktion  
WS 2014/15



Bei zyklischen Kontrollflussgraphen **potenziell unendlich viele (beliebig lange) Pfade**.

- Aber: **Obere Grenzen** für mögliche Länge (und damit Anzahl) ggf. aus Spezifikation oder aus technischen Einschränkungen ableitbar.

In Praxis 100%ige Pfadüberdeckung oft nicht erreichbar (vgl. Beispiel T2.0 F12-13).

→ **Als theoretisches Vergleichsmaß wichtig.**

Nichtsdestotrotz: auch Pfadüberdeckung kann nicht alle Fehler finden (nicht alle Eingabewerte und Variablenbelegungen getestet).

→ Brauche alternatives Testparadigma als Ergänzung.

→ **Test der Bedingungen, Datenflussbasiertes Testen** (vgl. folgende Abschnitte)

58

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.2 – Entscheidungstest (S.152-155)
- Pfadtest (S.160)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 7.2.4 – Schleifenüberdeckung (S.202-208)
  - Pfadüberdeckung (S.203-204)

S. Dissmann – **Softwaretechnik – Teil 8: Testen von Software** – folien-swt-2014-07.pdf

<https://ls10-wiki.cs.uni-dortmund.de/swt/public:vorlesung>

- Folie 362-369



### 2.4 White-Box- Test



Idee der White-Box Testentwurfsverfahren

Kontrollflussbasierter Test

Test der Bedingungen

Datenflussbasierter Test

Statische Analyse

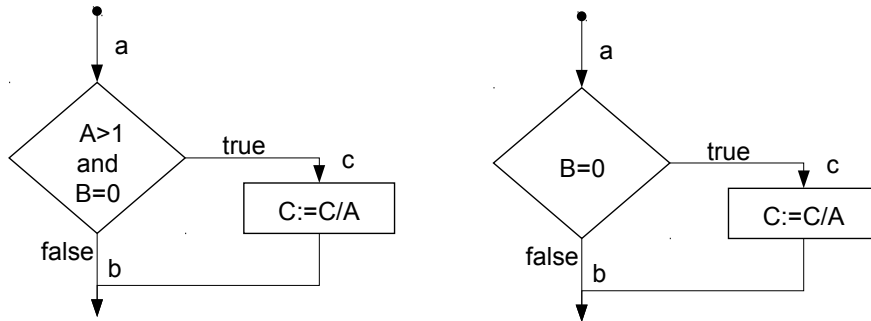
## Zweigüberdeckung: Ausdruckstärke

Softwarekonstruktion  
WS 2014/15



Kann es Testfälle mit Zweigüberdeckung geben, die richtiges und falsches Programm nicht unterscheiden ?

Antwort: ?



60

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Abbildung 9.3 (S.237)

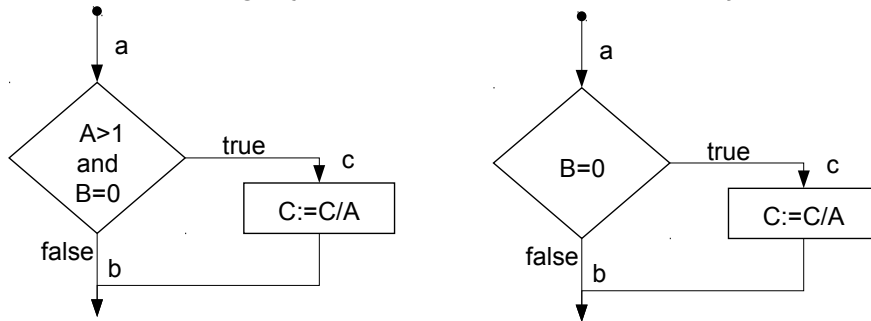
## Zweigüberdeckung: Ausdruckstärke

Softwarekonstruktion  
WS 2014/15



Kann es Testfälle mit Zweigüberdeckung geben, die richtiges und falsches Programm nicht unterscheiden ?

Antwort: Testfallmenge  $\{(A=2, B=C=0), (A=2, B=1, C=0)\}$ .



→ Brauchen stärkeres Kriterium, das Teilbedingungen berücksichtigt  
→ Bedingungsüberdeckung !

61

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Abbildung 9.3 (S.237)



- **Wahrheitswerte:** `false`, `true` (oft auch `0`, `1` oder »falsch«, »wahr«).
- **Atomare (Teil-)Bedingung (*condition*):**
  - Variablen vom Typ `boolean`.
  - Operationen mit Rückgabewert vom Typ `boolean`.
  - Vergleichsoperationen.
  - Z.B. `flag; isEmpty(); size > 0`
- **Zusammengesetzte Bedingung (*compound condition*):**
  - Verknüpfung von (Teil-)Bedingungen mit booleschen Operatoren.
  - Basis-Operatoren sind **und** ( $\wedge$ ,  $\cap$ ), **oder** ( $\vee$ ,  $\cup$ ), **nicht** ( $\neg$ ).
- **Entscheidung:** (zusammengesetzte) Bedingung, die Programmablauf steuert.
- **In Java:**
  - `&`, `|`, `^` Bitweise *und*, *oder* und *exklusiv-oder*-Verknüpfung.
  - `&&`, `||` Wie oben, aber lazy evaluation, z.B. `a && b = a ? b : false`.
  - `if ((size > 0) && (inObject != null)) {...} else {...}`.

62

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)



## Bedingungsüberdeckungen:

- Berücksichtigen **Bedingungen in Schleifen und Auswahlkonstrukten** zur Definition von Tests.

**Idee:** Zusammengesetzte Bedingungen nur einmal zu True und False auswerten ist nicht ausreichend.

→ **Teilbedingungen** (atomare Prädikatterme) **variieren**.

### Beispiel:

- if  $A > 1$  and  $(B = 2 \text{ or } C > 1)$  and  $D$  then ... else ...
- Entscheidungsprädikat:  $A > 1$  and  $(B = 2 \text{ or } C > 1)$  and  $D$
- Atomare Prädikatterme:  $A > 1$ ,  $B = 2$ ,  $C > 1$ ,  $D$

**Anderes Beispiel:** Vergleich der Zweigüberdeckung von if (a AND b AND c) {...} einer äquivalenten Konstruktion if a {if b {if c {...}}}

63

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)



## Einfache (oder auch atomare) Bedingungsüberdeckung ( $C_2$ -Test):

- Testdatenmenge  $T$  erfüllt  $C_2$ -Überdeckung g.d.w. es:
  - für jede Verzweigung im Programm mit zwei Ausgängen und
  - für jeden (atomaren) Prädikatterm  $p$  des zur Verzweigung gehörenden Booleschen Ausdrucks
  - zu jedem Wahrheitswert von  $p$  ein Testdatum  $t$  aus  $T$  gibt, bei dessen Ausführung  $p$  diesen Wahrheitswert annimmt.
- **Hinweis:** Verzweigungen mit mehr als zwei Ausgängen in äquivalentes Konstrukt mit zwei Ausgängen umformbar.
- Garantiert keine Zweigüberdeckung.  
→ Testen beider Wahrheitswerte kompletter (zusammengesetzter) Boolescher Ausdrücke nicht garantiert.

64

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)





## Einfache Bedingungsüberdeckung: Testwirksamkeitsmaß $C_2$

$$C_2 =_{\text{def}} \frac{\text{Anzahl Prädikate}_{\text{TRUE}} + \text{Anzahl Prädikate}_{\text{FALSE}}}{2 * \text{Anzahl der Prädikate}}$$

Wie üblich: 100%ige Überdeckung nicht erreichbar, wenn Belegung von Teilbedingungen mit true und false nicht erreicht werden kann (weil durch Programmkontext ausgeschlossen).

65

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)

## Beispiele zur einfachen Bedingungsüberdeckung

Softwarekonstruktion  
WS 2014/15



A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	C	$A \wedge B \wedge C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

2 Ausdrücke, 2 Testfälle

2 Ausdrücke, 2 Testfälle

3 Ausdrücke, 2 Testfälle

66

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Beispiel 9.2.4 (S.239)

## Einfache Bedingungsüberdeckung: Anzahl Testfälle

Softwarekonstruktion  
WS 2014/15



Wieviele Testfälle benötigt für einfache Bedingungsüberdeckung für  
Bedingung aus **2 Ausdrücken** ?

	A	B	$A \wedge B$
•	0	0	0
	0	1	0
	1	0	0
	1	1	1

Wieviele bei  $n > 2$  Ausdrücken ?

	A	B	C	$A \wedge B \wedge C$
•	0	0	0	0
	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
	1	1	0	0
	1	1	1	1

67

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Beispiel 9.2.4 (S.239)



Wieviele Testfälle benötigt für einfache Bedingungsüberdeckung für Bedingung aus **2 Ausdrücken** ?

- Falls Überdeckung erreicht werden kann, reichen 2: Entweder  $T1=((0,0), (1,1))$  oder  $T2=((0,1),(1,0))$ . Falls beide nicht erreicht werden können (weil je mind. eine Kombination nicht erreicht werden kann), bleiben ohnehin höchstens zwei Testfälle übrig.

Wieviele bei  **$n > 2$  Ausdrücken** ?

- Falls Überdeckung erreicht werden kann, reichen  $n$ .

Beweis per Induktion: Induktionsanfang s.o.

Induktionsschritt: Gilt für  $n+1$  auf Basis von  $n$ : Die  $n$  Testfälle decken mind. einen Wert (true oder false) für den  $n+1$ sten Ausdruck ab; für den anderen reicht ein weiterer Testfall.

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Bedingungstest (S.155-156)

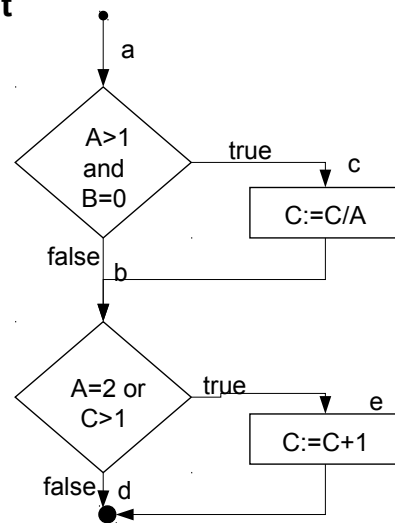
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Beispiel 9.2.4 (S.239)



Kann es einen  $C_2$ -Testfall geben, der **nicht alle Zweige testet** ?



69

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Abbildung 9.3 (S.237)



## Beispiel

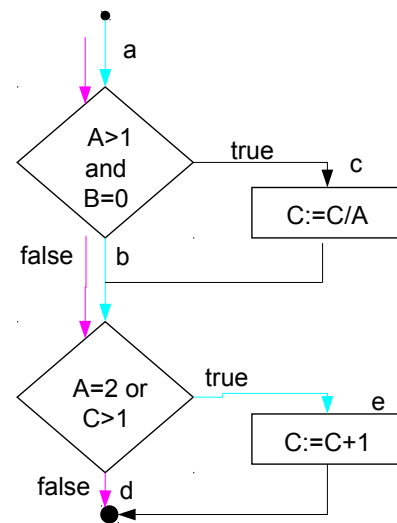
- Testdatum A=2,B=1,C=4

- A>1 true
- B=0 false
- A=2 true
- C>1 true
- Weg: abc

- Testdatum A=1,B=0,C=1

- A>1 false
- B=0 true
- A=2 false
- C>1 false
- Weg: abd

→ Zweig c wird nicht überdeckt!



70

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Abbildung 9.3 (S.237)



### **Zweig-/Bedingungsüberdeckung:**

Testdatenmenge erfüllt Zweig-/ Bedingungsüberdeckung  
gdw. sie **C<sub>2</sub>-Überdeckung** (atomare Bedingungsüberdeckung) und  
**C<sub>1</sub>-Überdeckung** (Zweigüberdeckung) **erfüllt**.

71

### **Literatur:**

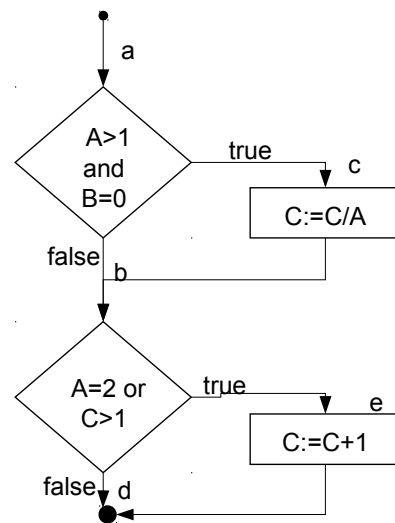
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Definition 9.2.2 (S.238)



Kann es einen **Zweig- / Bedingungs-  
überdeckungs-Testfall** geben, der  
**nicht alle Zweigkombinationen**  
testet ?



72

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Abbildung 9.3 (S.237)





Beispiel:

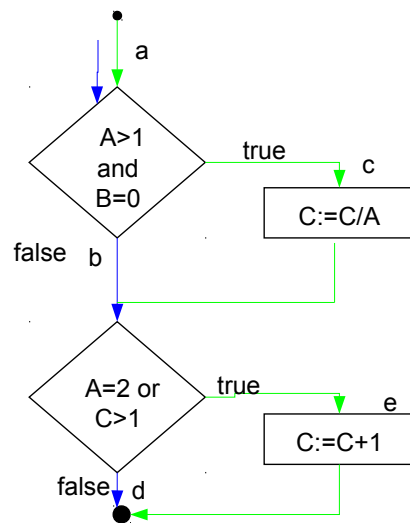
- Testdatum A=2,B=0,C=4:

- A>1 true
- B=0 true
- A=2 true
- C>1 true
- Weg: ace

- Testdatum A=1,B=1,C=1:

- A>1 false
- B=0 false
- A=2 false
- C>1 false
- Weg: abd

→ Alle Zweige nun überdeckt, aber  
nicht alle **Zweigungskombinationen!**



73

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Abbildung 9.3 (S.237)

## Zweig-/Bedingungsüberdeckung: Ausdrucksstärke

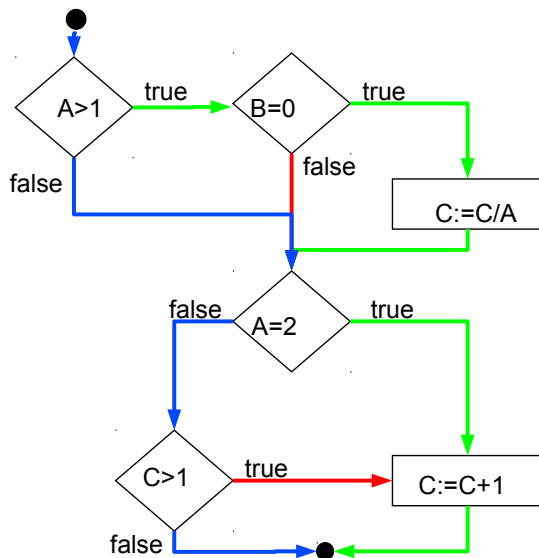
Softwarekonstruktion  
WS 2014/15



Zur Veranschaulichung: Umformung  
des Kontrollflussgraphen  
(normalerweise nicht vorgesehen,  
hier nur zu didaktischen Zwecken):

- Testdatum  $A=2, B=0, C=4$
- Testdatum  $A=1, B=1, C=1$
- 2 Zweige fehlen !

**Lösung:**  
Mehrfachbedingungsüberdeckung !



74

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Abbildung 9.3 (S.237)



### Mehrfachbedingungsüberdeckung ( $C_3$ - oder $C_2(M)$ -Überdeckung):

Testdatenmenge erfüllt Mehrfachbedingungsüberdeckung g.d.w.

- für jede Verzweigung im Programm mit zwei Ausgängen und
- für zur Verzweigung gehörenden **Booleschen Ausdruck**
- jede mögliche Kombination der Wahrheitswerte atomarer Prädikatterme ausgeführt wird.

Falls Programm keine Tautologien enthält: Ergibt true/false-Belegung der zusammengesetzten Booleschen Ausdrücke => Impliziert Zweigüberdeckung.

Wie üblich: 100%ige Überdeckung nicht erreichbar, wenn Kombination von Wahrheitswerten nicht erreicht werden kann.

- Z.B. bei  $(x > 0) \ \&\& \ (x < 5)$  ist (falsch, falsch) nicht realisierbar

75

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Definition 9.2.3 (S.238)

Folien-swt-2014-klein-07: F. 355

## Beispiel: Mehrfachbedingungsüberdeckung

Softwarekonstruktion  
WS 2014/15



Teste jede Kombination der Wahrheitswerte aller atomarer Ausdrücke !

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	C	$A \wedge B \wedge C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

2 Ausdrücke, 4 Testfälle

2 Ausdrücke, 4 Testfälle

3 Ausdrücke, 8 Testfälle

Frage: Wieviele Testfälle werden benötigt (für n atomare Prädikatterme) ?

76

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Beispiel 9.2.4 (S.239)

Folien-swt-2014-klein-07: F. 355



**Testfall-Anzahl bei  $n$  atomaren Ausdrücken:  $2^n$ .**

$$\text{Mehrfachbedingungsüberdeckungsgrad} = \frac{\text{Anzahl getesteter Kombinationen atomarer Ausdr.}}{2^{\text{Gesamtzahl atomarer Ausdrücke}}}$$

→ Aufwändig zu realisieren.

**Alle Kombinationen** durch Testdaten **nicht immer realisierbar.**

→ Weist nicht auf Fehler der Programmlogik hin!

→ Testbeurteilung wird erschwert!

**Besser:** Minimale Mehrfachbedingungsüberdeckung.

77

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

## Minimale Mehrfachbedingungsüberdeckung

Softwarekonstruktion  
WS 2014/15



Testdatenmenge erfüllt **Minimale Mehrfachbedingungsüberdeckung**  
( $C_2(mM)$ -Überdeckung, *minimal bestimmende Mehrfachbedingungsüberdeckung*) g.d.w.:

- für jede Verzweigung im Programm mit **zwei Ausgängen** und
- für zur Verzweigung gehörenden **Booleschen Ausdruck**
- folgende Kombinationen der Wahrheitswerte der enthaltenen atomaren Prädikatterme ausgeführt werden:
  - **Jede Kombination** von Wahrheitswerten, für die Änderung des Wahrheitswertes **eines Terms** den **gesamten** Wahrheitswert **ändert**.

[Bemerkung: Da diese Eigenschaft jede Kombination von Wahrheitswerten für sich betrachtet, gibt es immer eine **eindeutige** kleinste Kombinationsmenge, die mM erfüllt (im Gegensatz z.B. zur einfachen Bedingungsüberdeckung).]

78

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Definition 9.2.3 (S.238)

Folien-swt-2014-klein-07: F. 355

# Minimale Mehrfachbedingungsüberdeckung

Softwarekonstruktion  
WS 2014/15



$$\text{Minimale Mehrfachbedingungs-überdeckungsgrad} = \frac{\text{Anzahl getesteter MM Kombinationen}}{\text{Gesamtzahl MM Kombinationen}}$$

100% **minimale Mehrfachbedingungsüberdeckung** impliziert 100% **Entscheidungsüberdeckung**.

Anzahl Testfälle **im Normalfall** geringer als bei Mehrfachbedingungsüberdeckung.

Gesamtzahl **Kombinationen** für  $C_2(mM)$ -Überdeckung **bei reinen AND bzw. OR-Bedingungen** mit  $n$  atomaren Prädikaten:  $n+1$ .

**Algorithmus** für **Überprüfung** eines Testfalls auf MM-Bedingung: Direkte Überprüfung der Eigenschaft „Änderung des Wahrheitswertes **eines Terms** ändert den **gesamten** Wahrheitswert“.

**Algorithmus** für die **Generierung** einer MM-Testfallmenge: Sukzessive Überprüfung der MM-Bedingung für jeden möglichen Testfall.

79

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237-240)
- Definition 9.2.3 (S.238)

Folien-swt-2014-klein-07: F. 355

## Frage: Minimale Mehrfachbedingungsüberdeckung



Seien  $A_1$ ,  $A_2$ ,  $A_3$  atomare Ausdrücke und  $B = \text{NOT}(A_1) \text{ AND NOT}(A_2 \text{ AND NOT}(A_3))$  der durch unten abgebildete Wahrheitstabelle definierte Gesamtausdruck.

**Frage:** Welche Werte-Kombinationen von  $A_1$ ,  $A_2$  und  $A_3$  sind für **minimale Mehrfachbedingungsüberdeckung** mit Testfällen zu bewirken ?

A1	0	1	0	1	0	1	0	1
A2	0	0	1	1	0	0	1	1
A3	0	0	0	0	1	1	1	1
B	1	0	0	0	1	0	1	0



## Lösung: Minimale Mehrfachbedingungsüberdeckung

Softwarekonstruktion  
WS 2014/15



Antwort: **Mit Testfällen alle Kombinationen (Spalten) bewirken**, in denen mindestens eine rote Markierung ist.

Da hier MM-Kombinationen existieren, ist sowohl **B=0** als auch **B=1** **abgedeckt** und daher kein gesonderter Testfall notwendig, der (Gesamt-)Ausdruck **einmal zu wahr und einmal zu falsch** auswertet.

A1	0	1	0	1	0	1	0	1
A2	0	0	1	1	0	0	1	1
A3	0	0	0	0	1	1	1	1
B	1	0	0	0	1	0	1	0

**Bemerkung:** Für minimale Mehrfachbedingungsüberdeckung reicht **nicht**, alle Belegungen der Teilbedingungen und der Gesamtbedingung mit 0 und 1 zu testen (sonst würden hier die Testfälle 1,2,3,7 reichen).

81

# Beispiele: Minimale Mehrfachbedingungsüberdeckung



A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	C	$A \wedge B \wedge C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

2 Ausdrücke, 3 Testfälle

2 Ausdrücke, 3 Testfälle

3 Ausdrücke, 4 Testfälle

# Minimale Mehrfachbedingungs- überdeckung: Vgl. früheres Beispiel

Softwarekonstruktion  
WS 2014/15



Testdatum  $A=3, B=0, C=3$ :

- $A > 1$  true
- $B = 0$  true
- $A = 2$  false
- $C > 1$  false (da Auswertung  $C := C/A$ )

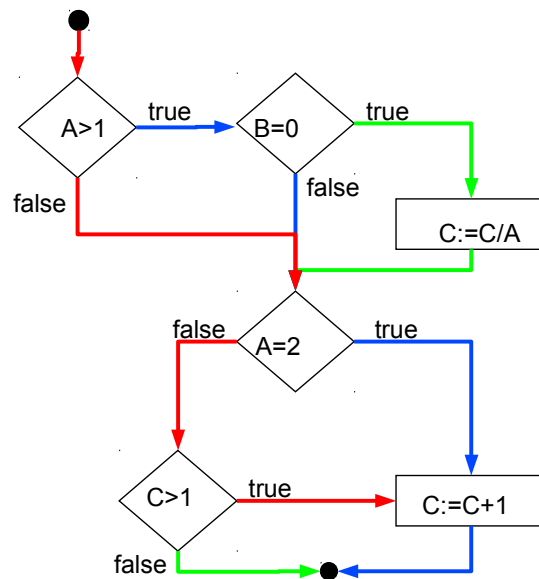
Testdatum  $A=2, B=1, C=0$ :

- $A > 1$  true
- $B = 0$  false
- $A = 2$  true
- $C > 1$  false

Testdatum  $A=1, B=0, C=2$ :

- $A > 1$  false
- $B = 0$  true
- $A = 2$  false
- $C > 1$  true

→ Alle Zweige geprüft.



83



In DO-178B (Software Considerations in Airborne Systems and Equipment Certification) geforderte »*Modified Condition/Decision Coverage*« (MC/DC) ist ähnlich der **Minimalen Mehrfachbedingungsüberdeckung**.

- Fordert für jeden atomaren Ausdruck in min. einem Fall zu zeigen, dass er **alleine Gesamtentscheidung beeinflussen** kann.
- **Zwei MM-Kombinationen pro atomaren Ausdruck** notwendig.
- Bei  $n$  atomaren und/oder/nicht-verknüpften Ausdrücken: **Testfall-Anzahl** bei MC/DC mindestens  $n+1$  und höchstens  $2n$  (also nur linear wachsend !).

A1	0	1	0	1	0	1	0	1
A2	0	0	1	1	0	0	1	1
A3	0	0	0	0	1	1	1	1
B	1	0	0	0	1	0	1	0



Problem: Indirekte Verwendung von zusammengesetzten Booleschen Ausdrücken, z.B.:

- Beispiel: `flag = a || (b && c); if (flag) {...}`

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Bewertung (S.159)



**Problem:** Messung der Überdeckung der Teilbedingungen.

- Programmiersprachen und Compiler **verkürzen Auswertung von booleschen Ausdrücken**, sobald Ergebnis feststeht.
- **Beispiel:** AND-Verknüpfung: eine Teilbedingung »false«  
→ Gesamtbedingung: »false« (egal welcher zweite Wert).
- Compiler **ändern Reihenfolge der Auswertung** in Abhängigkeit von booleschen Operatoren („lazy evaluation“, s. nächste Folie) aus Effizienzgründen.
- Wegen Verkürzung der Auswertung Überdeckung oft **nicht nachweisbar**.

86

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Bewertung (S.159)

# Bedingungsüberdeckung und »lazy evaluation«



»lazy evaluation«: Bedingungen so lange prüfen, bis Wahrheitswert feststeht.

- Im Programm: `if (A && B) then op1(); op2() ...`
  - Im Objectcode: `if (A) then if (B) then op1(); op2() ...`
- Einfache Bedingungsüberdeckung: **drei Fälle**, sonst B=1 nicht Programm berücksichtigt (Abbruch, sobald A=0 erkannt ist).
- **Gesamtausdruck zu 0 und 1** ausgewertet.
  - **Mehrfach-BÜ und MM-BÜ nicht erreichbar.**

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Abbruch der Auswertung



## **Einfache (oder atomare) Bedingungsüberdeckung ( $C_2$ -Test, *condition coverage*):**

- Atomare Prädikate einmal True und einmal False auswerten.
- Garantiert keine Zweigüberdeckung.

## **Mehrfach-Bedingungsüberdeckung ( $C_3$ -Test, *multiple condition coverage*):**

- Kombinationen atomarer Prädikate betrachten.
- Bei einigermaßen komplizierten Prädikaten nicht mehr handhabbar.

## **Minimale Mehrfach-Bedingungsüberdeckung ( $C_{\min\text{Mehrfach}}$ , *condition determination coverage*):**

- Kompromiss zwischen  $C_2$  und  $C_3$ ,
- jedes Prädikat (egal ob atomar oder zusammengesetzt) zu True und False ausgewertet,
- weniger als kombinatorische Abdeckung,
- mehr als beidseitige Auswertung der zusammengesetzten Prädikate.
- impliziert Zweigüberdeckung

88

## **Literatur:**

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 5.2.3 – Test der Bedingungen (S.155-159)
- Mehrfachbedingungstest (S.156-158)

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

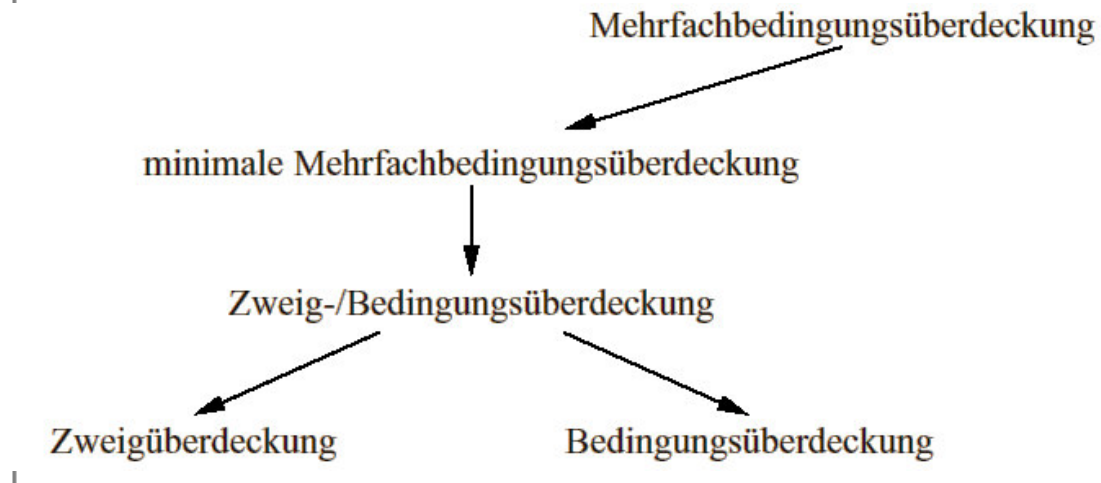
<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 9.2 (ab S.236)
- Atomare Bedingungsüberdeckung (S.237)
- Minimale Mehrfachbedingungsüberdeckung (S.238)
- Mehrfachbedingungsüberdeckung (S.240)





**K1** → **K2**: jede Testdatenmenge, die K1 erfüllt, erfüllt auch K2.



89

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.1 (ab S.251)
- Abbildung 10.3 (S.254)



**Anzahl Tests** pro Testkriterium (einzelne Entscheidung mit  $n$  Termen):

Kriterium	Zahl der erforderlichen Tests pro Entscheidung mit $n$ Termen
Bedingungsüberdeckung	$n$
Minimale Mehrfachbedingungsüberdeckung (reine AND- bzw. OR-Bedingungen)	$\leq n+1$
Mehrfachbedingungsüberdeckung	$2^n$

90

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.2 (S.257-260)
- Tabelle 10.2 (S. 259)



Falls **Anzahl ausgehender Kanten** pro Entscheidungsknoten und **Anzahl Variablen pro Segment** durch Konstante **begrenzt**, dann für Programm mit  $n$  Segmenten im schlechtesten Fall:

- **Pfadüberdeckung:** Keine a-priori-Grenze für Anzahl von Tests (nur vorhanden bei vorhandener Grenze an Anzahl Schleifendurchläufe).
- **Strukturierte Pfadüberdeckung:**  $O(2^n)$  Tests.
- **Zweig- und Anweisungsüberdeckung:**  $O(n)$  Tests.

91

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.2 (S.257-260)



## Aufgedeckte Fehler pro Testkriterium (vgl. [Rie97]):

Kriterium	Gefundene Fehler
Anweisungsüberdeckung	18% bis 41%
Zweigüberdeckung	16% bis 92%
Strukturierte Pfadüberdeckung	zweifache der Zweigüberdeckung (43% statt 21%*)
Pfadüberdeckung	dreifache der Zweigüberdeckung (62% bis 64% statt 21%*)
[minimale (Mehrfach-)] Bedingungsüberdeckung	keine Studien vorhanden

**Unterschiedliche Quoten** beruhen auf unterschiedlichen  
Untersuchungsansätzen.

\* Angabe 21% beruht auf Messung nach [How78].

92

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.3 (S.260-271)
- Abschnitt 10.3.1 – Prozentzahlen der gefundenen Fehler (S.261-262)



Anweisungs- und Entscheidungsüberdeckung für objektorientierte Software nicht ausreichend:

- Komplexität objektorientierter Systemen in **Beziehungen** zwischen **Klassen** bzw. Interaktionen zwischen Objekten verborgen.
- **Methoden** in Klassen oft von **geringer Komplexität**.
- Anweisungs- und/oder Zweigüberdeckung leicht erreichbar, aber wenig aussagekräftig (trotzdem notwendig, aber nicht hinreichend).
- Datenflussbasiertes Testen (nächster Abschnitt), insbesondere Interprocedural Data Flow Testing

93

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.3 (S.260-271)
- Abschnitt 10.3.1 – Prozentzahlen der gefundenen Fehler (S.261-262)

# Auswahl eines Kontrollfluss-Testverfahrens



		Liegt eine Prozedur, Funktion oder Operation einschließlich Quellcode vor?						
		ja				nein		
		Enthält die Prozedur, Funktion oder Operation ...						Black-Box-Testen
... nur Anweisungen	... nur Anweisungen und atomare, nichtzusammengesetzte Bedingungen	... nur Anweisungen, atomare, nichtzusammengesetzte Bedingungen und Schleifen	... nur Anweisungen und zusammengesetzte Bedingungen	... nur Anweisungen, zusammengesetzte Bedingungen und Schleifen				
Anwendungsüberdeckungstest	Zweigüberdeckungstest	Genügt es, Schleifen einmal zu wiederholen?		Reicht Überprüfung aller atomaren Bedingungen aus?		Genügt es, Schleifen einmal zu wiederholen?		
		ja	nein	ja	nein	ja	nein	
		<i>boundary interior</i> -Pfadtest	strukturierter Pfadtest	Zweigüberdeckungstest und einfacher Bedingungsüberdeckungstest	minimaler Mehrfach-Bedingungsüberdeckungstest	<i>boundary interior</i> -Pfadtest	strukturierter Pfadtest	
						Reicht die Überprüfung aller atomaren Bedingungen aus?		
						ja	nein	
						einfacher Bedingungsüberdeckungstest	minimaler Mehrfach-Bedingungsüberdeckungstest	

# Werkzeugunterstützung für Kontrollflusstesten (weitere s. Anhang)



## Cobertura (Freeware): <http://cobertura.sourceforge.net>

Features: Prozentuale Überdeckung, zyklomatische Komplexität (McCabe), Zweigüberdeckung, Zeilenüberdeckung.

## CodeCover (Freeware): <http://www.codecover.org>

Features: Entscheidungs-, Zweig-, Anweisungs-, Grenze-Inneres-Überdeckung.

## Jcoverage (Freeware): <http://jcoverage.sourceforge.net/>

Features: Zeigt wie oft jede Code-Zeile ausgeführt wurde.

## McCabe Test coverage:

<http://www.mccabe.com/ip-test.htm>

Features: Prozentuale Überdeckung der Pfade.

**Cobertura Coverage Report - de.cio.grafe**

Package	# Classes	Line Coverage	Branch Coverage	Complexity
de.cio.grafe	11	100%	100%	0
Classes in this Package				
Bus	1	100%	100%	0
BusControl	1	100%	100%	0
BusControlTest	1	100%	100%	0
BusControlTest2	1	100%	100%	0
BusControlTest3	1	100%	100%	0
BusControlTest4	1	100%	100%	0
BusControlTest5	1	100%	100%	0
BusControlTest6	1	100%	100%	0
BusControlTest7	1	100%	100%	0
BusControlTest8	1	100%	100%	0
BusControlTest9	1	100%	100%	0
BusControlTest10	1	100%	100%	0
BusControlTest11	1	100%	100%	0
BusControlTest12	1	100%	100%	0
BusControlTest13	1	100%	100%	0
BusControlTest14	1	100%	100%	0
BusControlTest15	1	100%	100%	0
BusControlTest16	1	100%	100%	0
BusControlTest17	1	100%	100%	0
BusControlTest18	1	100%	100%	0
BusControlTest19	1	100%	100%	0
BusControlTest20	1	100%	100%	0
BusControlTest21	1	100%	100%	0
BusControlTest22	1	100%	100%	0
BusControlTest23	1	100%	100%	0
BusControlTest24	1	100%	100%	0
BusControlTest25	1	100%	100%	0
BusControlTest26	1	100%	100%	0
BusControlTest27	1	100%	100%	0
BusControlTest28	1	100%	100%	0
BusControlTest29	1	100%	100%	0
BusControlTest30	1	100%	100%	0
BusControlTest31	1	100%	100%	0
BusControlTest32	1	100%	100%	0
BusControlTest33	1	100%	100%	0
BusControlTest34	1	100%	100%	0
BusControlTest35	1	100%	100%	0
BusControlTest36	1	100%	100%	0
BusControlTest37	1	100%	100%	0
BusControlTest38	1	100%	100%	0
BusControlTest39	1	100%	100%	0
BusControlTest40	1	100%	100%	0
BusControlTest41	1	100%	100%	0
BusControlTest42	1	100%	100%	0
BusControlTest43	1	100%	100%	0
BusControlTest44	1	100%	100%	0
BusControlTest45	1	100%	100%	0
BusControlTest46	1	100%	100%	0
BusControlTest47	1	100%	100%	0
BusControlTest48	1	100%	100%	0
BusControlTest49	1	100%	100%	0
BusControlTest50	1	100%	100%	0

**McCabe Test Coverage**

Package	Line Coverage	Branch Coverage	Complexity
de.cio.grafe	100%	100%	0
de.cio.grafe.Bus	100%	100%	0
de.cio.grafe.BusControl	100%	100%	0
de.cio.grafe.BusControlTest	100%	100%	0
de.cio.grafe.BusControlTest2	100%	100%	0
de.cio.grafe.BusControlTest3	100%	100%	0
de.cio.grafe.BusControlTest4	100%	100%	0
de.cio.grafe.BusControlTest5	100%	100%	0
de.cio.grafe.BusControlTest6	100%	100%	0
de.cio.grafe.BusControlTest7	100%	100%	0
de.cio.grafe.BusControlTest8	100%	100%	0
de.cio.grafe.BusControlTest9	100%	100%	0
de.cio.grafe.BusControlTest10	100%	100%	0
de.cio.grafe.BusControlTest11	100%	100%	0
de.cio.grafe.BusControlTest12	100%	100%	0
de.cio.grafe.BusControlTest13	100%	100%	0
de.cio.grafe.BusControlTest14	100%	100%	0
de.cio.grafe.BusControlTest15	100%	100%	0
de.cio.grafe.BusControlTest16	100%	100%	0
de.cio.grafe.BusControlTest17	100%	100%	0
de.cio.grafe.BusControlTest18	100%	100%	0
de.cio.grafe.BusControlTest19	100%	100%	0
de.cio.grafe.BusControlTest20	100%	100%	0
de.cio.grafe.BusControlTest21	100%	100%	0
de.cio.grafe.BusControlTest22	100%	100%	0
de.cio.grafe.BusControlTest23	100%	100%	0
de.cio.grafe.BusControlTest24	100%	100%	0
de.cio.grafe.BusControlTest25	100%	100%	0
de.cio.grafe.BusControlTest26	100%	100%	0
de.cio.grafe.BusControlTest27	100%	100%	0
de.cio.grafe.BusControlTest28	100%	100%	0
de.cio.grafe.BusControlTest29	100%	100%	0
de.cio.grafe.BusControlTest30	100%	100%	0
de.cio.grafe.BusControlTest31	100%	100%	0
de.cio.grafe.BusControlTest32	100%	100%	0
de.cio.grafe.BusControlTest33	100%	100%	0
de.cio.grafe.BusControlTest34	100%	100%	0
de.cio.grafe.BusControlTest35	100%	100%	0
de.cio.grafe.BusControlTest36	100%	100%	0
de.cio.grafe.BusControlTest37	100%	100%	0
de.cio.grafe.BusControlTest38	100%	100%	0
de.cio.grafe.BusControlTest39	100%	100%	0
de.cio.grafe.BusControlTest40	100%	100%	0
de.cio.grafe.BusControlTest41	100%	100%	0
de.cio.grafe.BusControlTest42	100%	100%	0
de.cio.grafe.BusControlTest43	100%	100%	0
de.cio.grafe.BusControlTest44	100%	100%	0
de.cio.grafe.BusControlTest45	100%	100%	0
de.cio.grafe.BusControlTest46	100%	100%	0
de.cio.grafe.BusControlTest47	100%	100%	0
de.cio.grafe.BusControlTest48	100%	100%	0
de.cio.grafe.BusControlTest49	100%	100%	0
de.cio.grafe.BusControlTest50	100%	100%	0

# Beispiel-Programm: Ziffernstringauswertung (1)

Softwarekonstruktion  
WS 2014/15



## Auswertung eines Ziffernstrings und Ermittlung des dargestellten Wertes:

```
PROCEDURE WerteZiffernfolgeAus (inZiffernString: Ttext) : REAL;
TYPE TWoBinIch = (VordemKomma, NachdemKomma);
  Tziffern = SET OF ['0', ..., '9']
CONST Cfehlercode = -1.0;
  Cziffern = Tziffern {'0', ..., '9'}
VAR Zchn : CHAR
  Wert, Genauigkeit : REAL;
  Position: CARDINAL ;
  WoBinIch: TWoBinIch ;
  Fehlerfrei : Boolean;
BEGIN
  Wert:=0.0;
  Genauigkeit:=1.0;
  WoBinIch:= VorDemKomma;
  Fehlerfrei:=TRUE;
  Position:=1;
  ...
  WHILE (Position <=TextVw.Length(inZiffernString))
    AND Fehlerfrei DO
    Zchn:=TextVW.LiesZeichen (inZiffernString,Position);
    IF Zchn IN Cziffern THEN
      IF WoBinIch=NachDemKomma THEN
        Genauigkeit:= Genauigkeit / 10.0;
      END; (* IF *)
      Wert:= 10.0 *Wert + Konvertiere (Zchn)
    ELSIF (Zchn='.') AND (WoBinIch=VorDemKomma) THEN
      WoBinIch:=NachDemKomma
    ELSE
      Fehlerfrei:=FALSE;
    END; (* IF *)
    INC (Position);
  END; (*WHILE*)

```

...

96



## Beispiel-Programm: Ziffernstringauswertung (2)

Softwarekonstruktion  
WS 2014/15



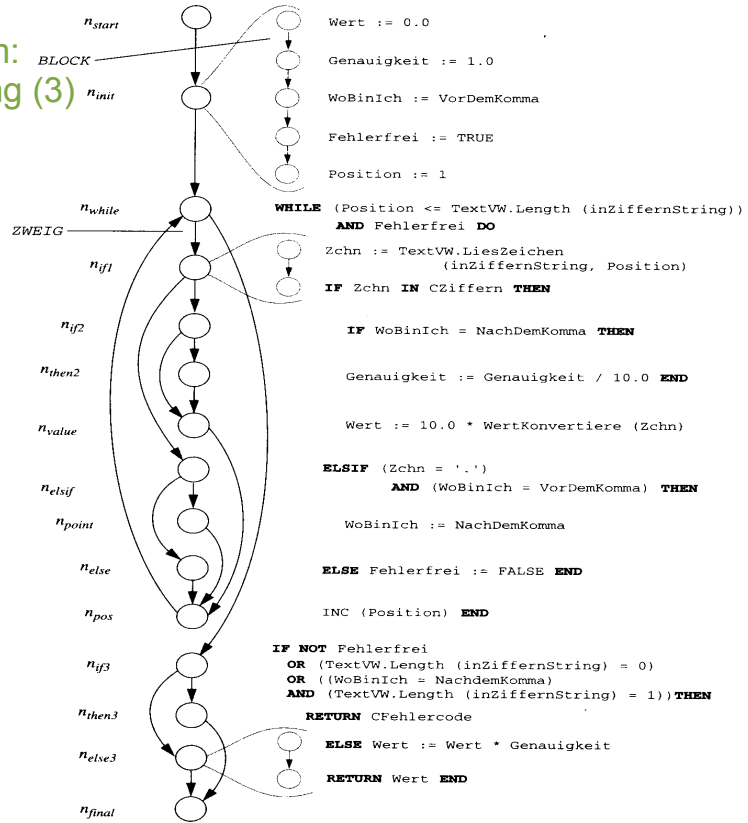
```
...  
  
IF NOT Fehlerfrei  
OR (TextVW.Length (inZiffernString) =0);  
OR (WoBinIch = NachDemKomma)  
AND (TextVwLenght (inZiffernString) = 1)) THEN  
    RETURN Cfehlercode  
    (* -- ILLEGALE ZEICHENFOLGE -- *)  
ELSE  
    Wert:=Wert * Genauigkeit;  
    RETURN Wert  
END (* IF *)  
END WerteZiffernFolgeAus;
```

97

## Beispiel-Programm: Ziffernstringauswertung (3)

Zugehöriger  
Kontrollfluss-  
graph

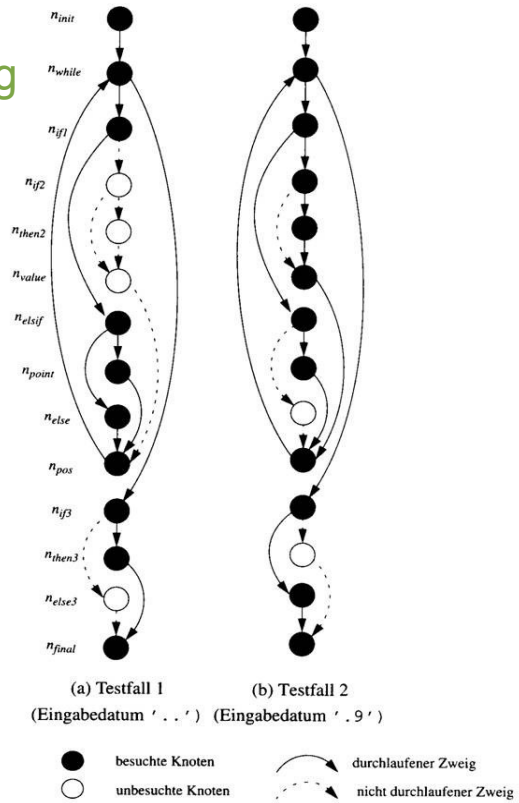
Testfälle für  
Anweisungs-  
überdeckung ?



# Ziffernauswertung: Anweisungsüberdeckung

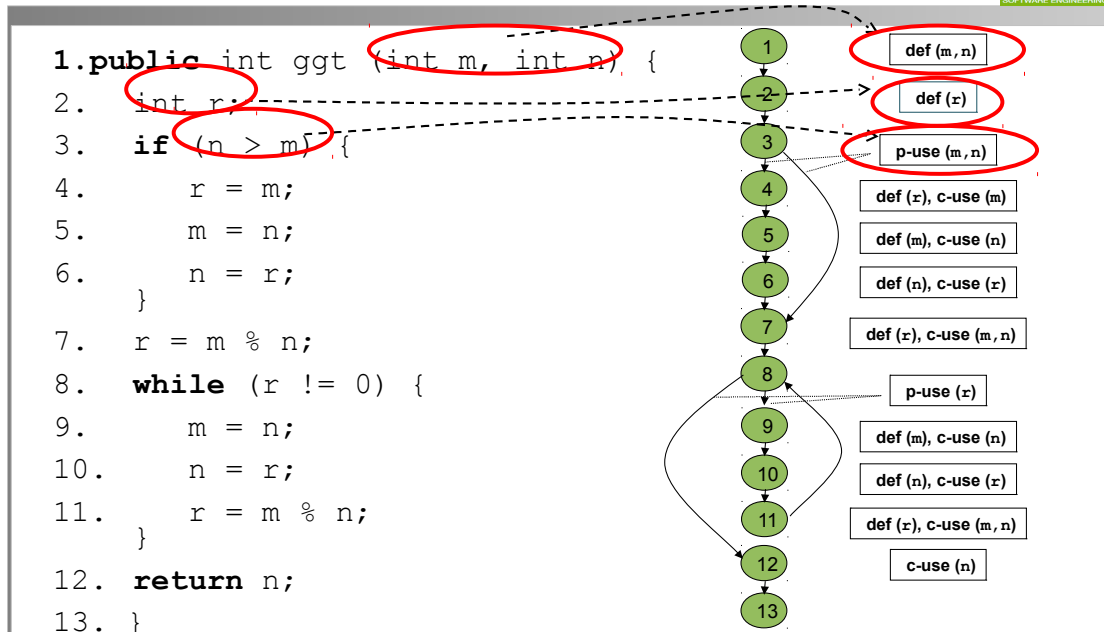
Testfälle für  
Anweisungs-  
überdeckung.

Erfüllt dies  
**Zweigüber-  
deckung** ?



## Beispiel ggt: Datenflussgraph

Softwarekonstruktion  
WS 2014/15



100

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

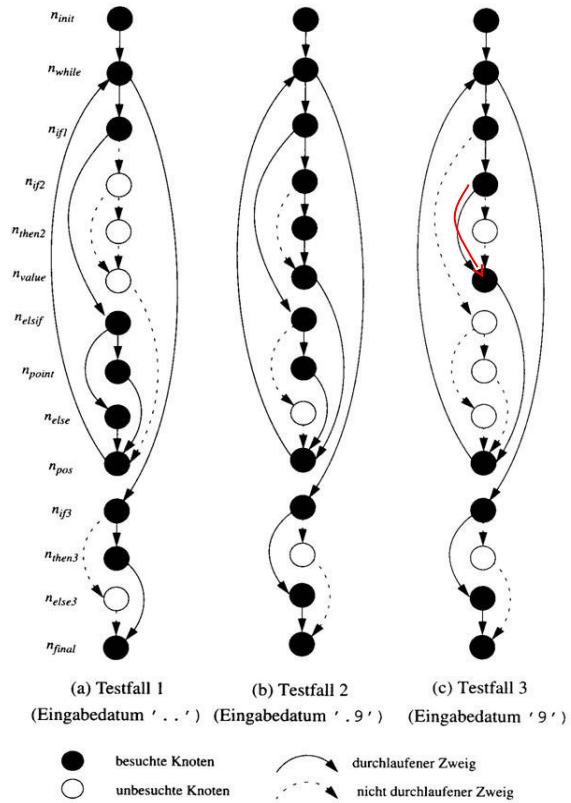
<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Definition 8.1.1 – Datenflussgraph (S.213-214)

## Ziffernauswertung: Zweigüberdeckung

Betrachtung beider Testfälle  
aus **Anweisungs-  
überdeckung**:

- nur Kante ( $n_{if2}, n_{value}$ ) wurde nicht durchlaufen.
- Pfad ( $n_{start}, n_{init}, n_{while}, n_{if1}, n_{if2}, n_{value}, n_{pos}, n_{while}, n_{if3}, n_{else3}, n_{final}$ ) enthält diese Kante.
- verwendbares Eingabedatum: '9'.

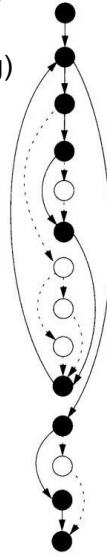


## Ziffernauswertung: Einfache Bedingungsüberdeckung (1)

### Testfall 3:

- Eingabe, die Prädikat (Teilbedingung) `Position <= TextVW.Length(inZiffernString)` zu **TRUE** auswertet → z.B. Eingabedatum '9'.
- Werte der Prädikate für Testfall protokollieren. → 1. und 3. Prädikat vollständig abgehandelt (jeweils **TRUE** und **FALSE**).
- Insgesamt für Testfall 3: 5 Prädikate mit T in erster Spalte der Tabelle, 8 mit F; 14 Prädikate insgesamt. Somit Überdeckung (für Testfall 3):

$$C_2 = \frac{5+8}{2 \cdot 14} = \frac{13}{28} = 46\%$$



Prädikate	'9'
	Position <= TextVW.Length (inZiffernString)
Fehlerfrei	T
WHILE-Bedingung	T,F
Zchn IN CZiffern	T
WoBinIch = NachDemKomma	F
Zchn = ','	-
WoBinIch = VorDemKomma	-
ELSIF-Bedingung	-
NOT Fehlerfrei	F
TextVW.Length (inZiffernString) = 0	F
WoBinIch = NachDemKomma	F
TextVW.Length (inZiffernString) = 1	T
IF3/AND-Prädikat	F
IF3-Bedingung	F

## Ziffernauswertung: Einfache Bedingungsüberdeckung (2)

Weiterer Testfall:

- Menge aller Eingaben, die **fehlerfrei** zu FALSE auswerten.  
→ Eingabedatum 'z'.
- Protokollierung der Werte der Prädikate für diesen Testfall.
- Überdeckung für **beide** Testfälle somit:

$$C_2 = \frac{8+12}{2*14} = \frac{20}{28} = 71\%$$

Prädikate	'9'	'z'
	Position <= TextVW.Length (inZiffernString)	T,F
Fehlerfrei	T	T,F
WHILE-Bedingung	T,F	
Zchn IN CZiffern	T	F
WoBinIch = NachDemKomma	F	-
Zchn = '.'	-	F
WoBinIch = VorDemKomma	-	T
ELSIF-Bedingung	-	F
NOT Fehlerfrei	F	T
TextVW.Length (inZiffernString) = 0	F	F
WoBinIch = NachDemKomma	F	F
TextVW.Length (inZiffernString) = 1	T	T
IF3/AND-Prädikat	F	F
IF3-Bedingung	F	T

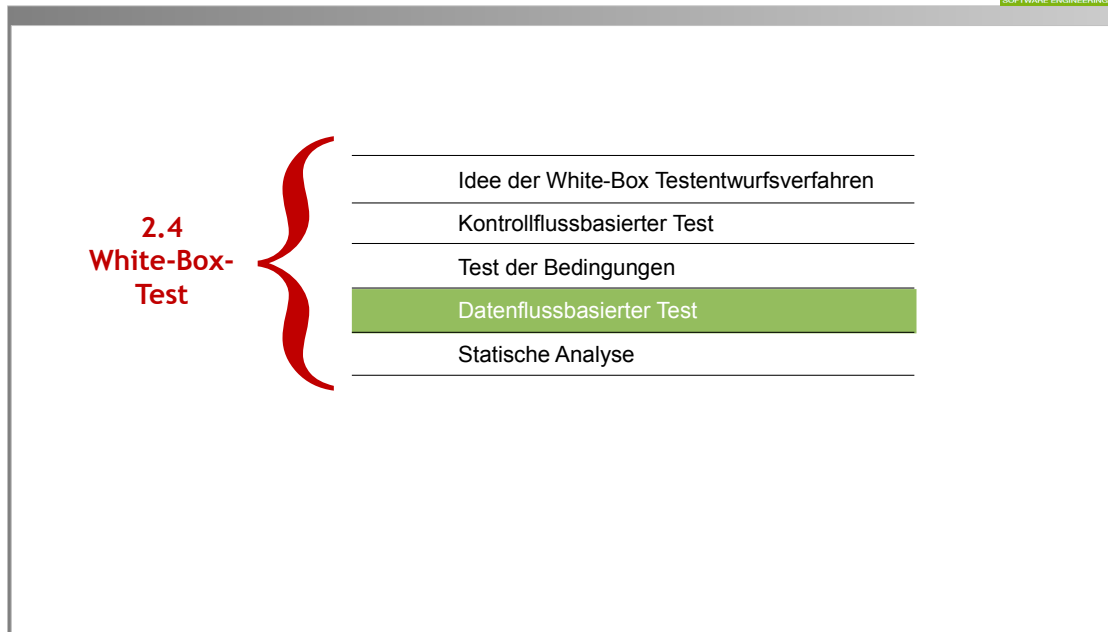
## Ziffernauswertung:

### Einfache Bedingungsüberdeckung (3)

Bemerkung: Die Zeilen „...-Bedingung“ bzw. „...-Prädikat“ für einfache Bedingungsüberdeckung nicht benötigt (hier nur zur Information); also einfache Bedingungsüberdeckung schon mit 5 Testfällen erreicht.

Prädikate	Eingabedaten					
	'9'	'z'	'9'	','	''	'.'
Position <= TextVW.Length (inZiffernString)	T,F					
Fehlerfrei	T	T,F				
WHILE-Bedingung	T,F					
Zchn IN CZiffern	T	F				
WoBinIch = NachDemKomma	F	-	T			
Zchn = '.'	-	F	T			
WoBinIch = VorDemKomma	-	T	T	T,F		
ELSIF-Bedingung	-	F	T			
NOT Fehlerfrei	F	T				
TextVW.Length (inZiffernString) = 0	F	F	F	F	T	
WoBinIch = NachDemKomma	F	F	T			
TextVW.Length (inZiffernString) = 1	T	T	F			
IP3/AND-Prädikat	F	F	F	F	F	T
IP3-Bedingung	F	T				





### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Kapitel 8 (ab S.212)



Welche **Ursachen** kann es haben, wenn **Anweisung falsches Ergebnis liefert**?

Antwort:

- 
-



Welche **Ursachen** kann es haben, wenn **Anweisung falsches Ergebnis liefert**?

Antwort:

- Anweisung ist **falsch**.
- Anweisung ist **korrekt**, aber referenzierte Werte werden vorher falsch berechnet.

**Beispiel:**

Anweisung  $A=B+C*D$  liefert falsches Ergebnis, wobei Anweisung korrekt ist, aber berechnete Werte vorher falsch berechnet wurden.



## Datenflussbasierter Test:

- **White-box-Verfahren.** → Nutzt **Programmstruktur** aus.
- **Dynamisches Verfahren:** Basiert auf Ausführung des Programms.
- **Hypothese: fehlerhafte Datenverwendung !**
- **Idee:** Testen der **Interaktion zwischen Anweisungen**, die Wert einer Variablen berechnen (**definieren**), und Anweisungen, die diesen Variablenwert benutzen (**referenzieren**).
- Testfälle unter Berücksichtigung der Datenverwendung **herleiten**.
- **Vollständigkeit** anhand Datenverwendung beurteilen.

```
1. public int ggt(int m, int n){  
2.     int r;  
3.     if (n > m) {  
4.         r = m;  
5.         m = n;  
6.         n = r;     ...  
}
```

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)



**Ziel** (wie beim Kontrollflusstesten): **Möglichst viele Fehler** finden, ohne **vollständige Pfadüberdeckung** erreichen zu müssen (zu aufwendig).

- **Unterscheidung** datenflussorientierter Verfahren:  
Alle Interaktionen oder nur Teil davon testen.  
(→ Datenflussbasierte Überdeckungsmaße).
- **Definition der Überdeckungsmaße** orientiert sich am Kontrollflussgraphen, erweitert um zusätzliche Informationen.  
→ **Datenflussgraph**.

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)



**Kontrollflussgraph**, bei dem zusätzlich zu jedem Knoten  $k$  Mengen  $DEF(k)$ ,  $UNDEF(k)$ , und  $REF(k)$  gehören.

- **$DEF(k)$** : Menge der Variablen  $x$ , für welche Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, der Variablen  $x$  Wert zuweist, der nicht anschließend in  $f$  undefiniert wird.
- **$UNDEF(k)$** : Menge der Variablen  $x$ , für welche Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, die Variablen  $x$  in undefinierten Zustand überführt, ohne  $x$  anschließend in  $f$  neu zu definieren.
  - Beispiele: lokale Iterationsvariablen in Schleifen, `free` (C), `undef` (Perl)
- **$REF(k)$** : Menge der Variablen  $x$ , für welche Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, die Variable  $x$  referenziert, ohne dass  $x$  vorher in  $f$  undefiniert wird.

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Definition 8.1.1 – Datenflussgraph (S.213-214)



- **X:=X+1** sei Anweisung des Knotens k
  - DEF(k) = {?}; REF(k) = {?}.
  - UNDEF(k) = {?}.
- **Buffer(bufpos):=c** sei Anweisung des Knotens k
  - DEF(k) = {?}.
  - REF(k) = {?}.
  - UNDEF(k) = {?}.
- **X:=A+B; Y:=C\*D; B:=Z; FREE(A); Y:=A+Z** sei Anweisungsfolge  
des Knotens k
  - DEF(k) = {?}.
  - REF(k) = {?}.
  - UNDEF(k) = {?}.

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Beispiele (S.214-215)



- **X:=X+1** sei Anweisung des Knotens k
  - DEF(k) = REF(k) = {X}.
  - UNDEF(k) = {}.
  
- **Buffer(bufpos):=c** sei Anweisung des Knotens k
  - DEF(k) = {Buffer}.
  - REF(k) = {bufpos,c}.
  - UNDEF(k) = {}.
  
- **X:=A+B; Y:=C\*D; B:=Z; FREE(A); Y:=A+Z** sei Anweisungsfolge  
des Knotens k
  - DEF(k) = {X,Y,B}.
  - REF(k) = {A,B,C,D,Z}.
  - UNDEF(k) = {A}.

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Beispiele (S.214-215)





Berücksichtigen:

- Rein **lokale Datenflüsse** vermeiden (wenn intern Referenz auf Definition folgt).  
→ Keine Berücksichtigung bei datenflussbezogenen Testkriterien.

Zwei Arten von lokalem Datenfluss:

- Innerhalb eines Blocks von sequentiell aufeinanderfolgenden Anweisungen.
- Bei Zuweisung innerhalb einer Bedingung (z.B. in C).

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Beispiel 8.1.4 – Lokaler Datenfluss (S.214)

## Lokaler Datenfluss in Block: Beispiel

Softwarekonstruktion  
WS 2014/15



**X:=A+B; Y:=C\*D; B:=Y, FREE(A); Y:=A+Z** sei Anweisungsfolge des Knotens

Enthält **rein lokalen Datenfluss** von Y:=C\*D nach B:=Y, also auf zwei Knoten aufteilen:

- **X:=A+B; Y:=C\*D;** sei Anweisungsfolge des Knotens k
  - DEF(k) = {X,Y}.
  - REF(k) = {A,B,C,D}.
  - UNDEF(k) = {}.
- **B:=Y, FREE(A); Y:=A+Z** sei Anweisungsfolge des Knotens l
  - DEF(l) = {Y,B}.
  - REF(l) = {Y,Z}.
  - UNDEF(l) = {A}.

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Beispiel 8.1.4 – Lokaler Datenfluss (S.214-215)



**Zuordnung von bedingten Anweisungen zu Knoten** so wählen, dass diese Knoten („Entscheidungsknoten“) nur Referenzen von Variablen enthalten (d.h.  $DEF(K)=UNDEF(K)=\{\}$ ).

- Beispiel: *If*  $((B=C+D))$  aufsplitten in  $B=C+D$  und *if*  $B$ .

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

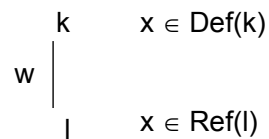
<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Beispiel 8.1.5 (S.215)

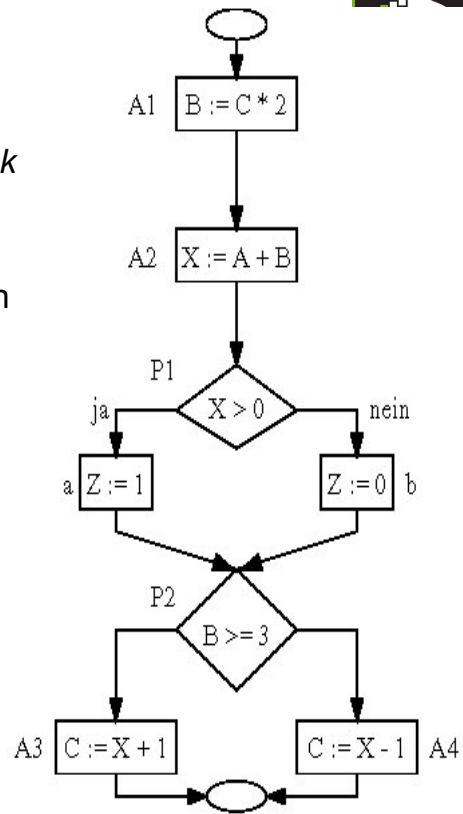
## DR-Weg

**DR-Weg:** „Die Definition von  $x$  in Knoten  $k$  erreicht eine Referenz von  $x$  im Knoten  $l$  über Weg  $w$ “:

- Gdw. Weg  $w$  im Kontrollflussgraph von  $k$  nach  $l$  führt und Variable  $x$  auf Weg **nicht neu definiert** oder **undefiniert** wird.



Beispiel für einen solchen Weg im nebenstehenden DFG ?



### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Abbildung 8.1 (S.216)
- Definition 8.2.6 – DR-Weg (S.219)

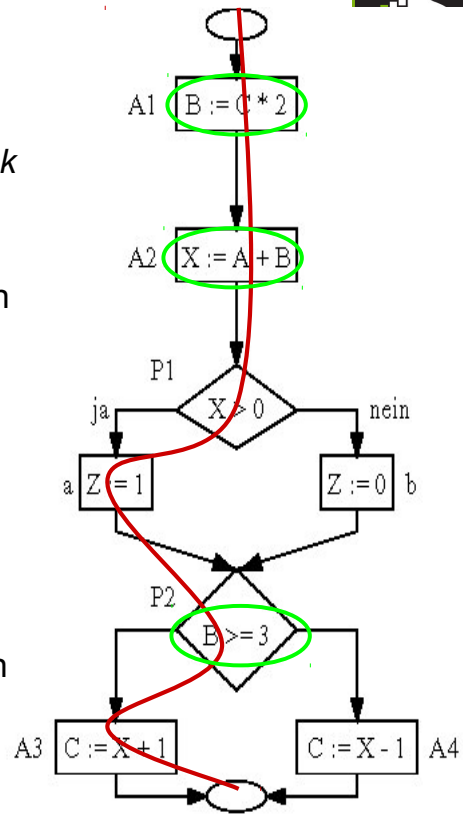
## DR-Weg

**DR-Weg:** „Die Definition von  $x$  in Knoten  $k$  erreicht eine Referenz von  $x$  im Knoten  $l$  über Weg  $w$ “:

- Gdw. Weg  $w$  im Kontrollflussgraph von  $k$  nach  $l$  führt und Variable  $x$  auf Weg **nicht neu definiert** oder **undefiniert** wird.

	$k$	$x \in \text{Def}(k)$
$w$		
	$l$	$x \in \text{Ref}(l)$

Beispiel für einen solchen Weg im nebenstehenden DFG: z.B. Definition von  $B$  in A1, Referenz in P2.



## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Abbildung 8.1 (S.216)
- Definition 8.2.6 – DR-Weg (S.219)



Analog zu den kontrollflussbasierten Kriterien: Kriterien an die Menge der Testpfade definieren.

### Kriterium „alle Definitionen“ (all-Defs):

- Resultat jeder Zuweisung (**Definition**) wenigstens einmal **benutzen** („referenzieren“).
- Testfallmenge  $T$  erfüllt Kriterium „alle Definitionen“ g.d.w.
  - Für jede Variable  $x$  und jede Definition von  $x$  existiert mindestens ein Weg in  $Wege(T)$ , auf dem die Definition eine Referenz von  $x$  erreicht.

### Literatur:

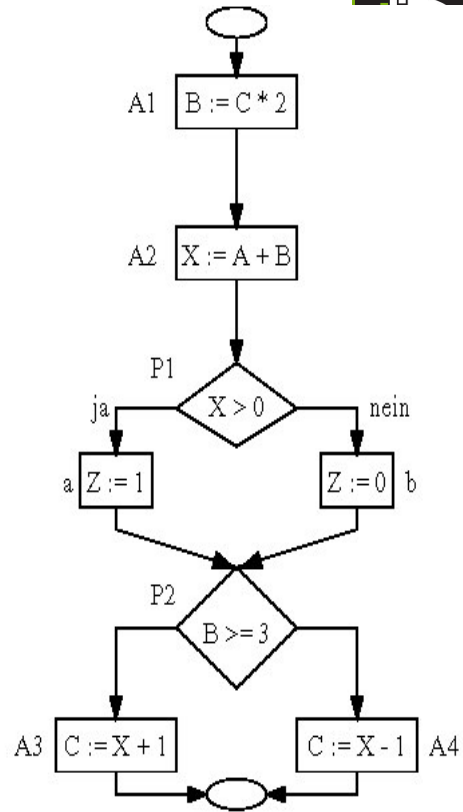
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)

## Alle Definitionen: Beispiel

- Welcher Testpfad erfüllt Kriterium „**alle Definitionen**“ (Resultat jeder **Definition** wenigstens einmal **benutzen**) z.B. für Variablen X und B ?
- Werden dabei **alle** (Definition, Referenz)-Paare dieser Variablen getestet ?



### Literatur:

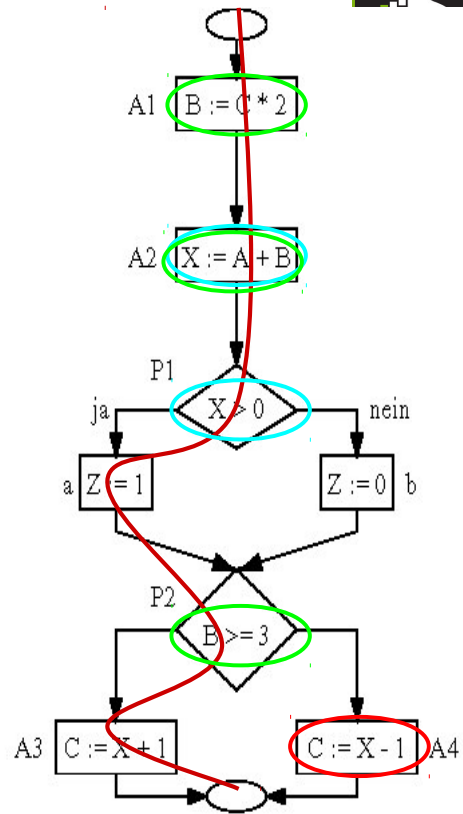
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)
- Abbildung 8.1 (S.216)

## Alle Definitionen: Beispiel

- Testpfad  $T=(A1, A2, P1, a, P2, A3)$  erfüllt Kriterium „**alle Definitionen**“ für Variablen X und B.
- Aber: nicht **alle** (Definition, Referenz)-Paare einer Variablen werden getestet (z.B. Definition von X in A2, Referenz von X in A4).



### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)
- Abbildung 8.1 (S.216)



## Alle DR-Interaktionen

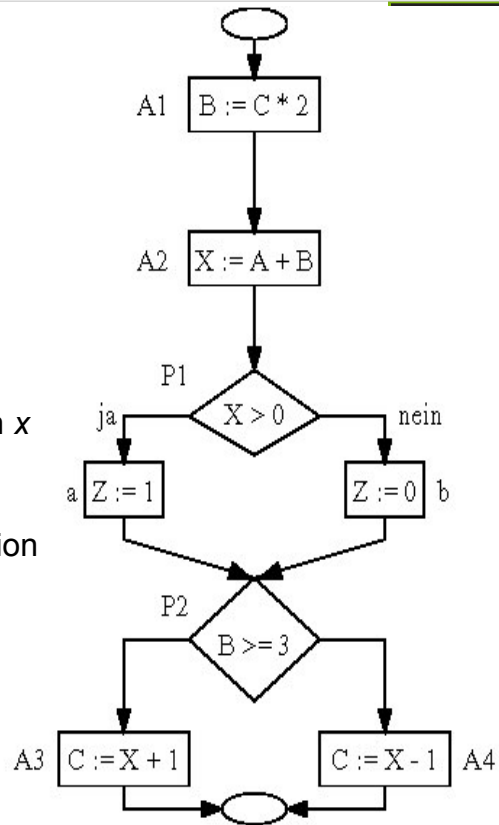
### Kriterium „alle DR-Interaktionen“:

**Ziel:** Alle Paare von Definitionen / Referenzen einer Variablen testen

Testdatenmenge  $T$  erfüllt Kriterium „alle DR-Interaktionen“ g.d.w.

- für jede Variable  $x$ , jede Definition von  $x$  und jede Referenz von  $x$ , die davon erreicht wird, **mindestens ein** Weg in  $Wege(T)$  existiert, auf dem die Definition eine Referenz von  $x$  erreicht.

**Beispiel:** Welche Testpfadmenge erfüllt dies für  $X$  und  $B$ ? Testet sie die Konsequenzen der Referenzen?



## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

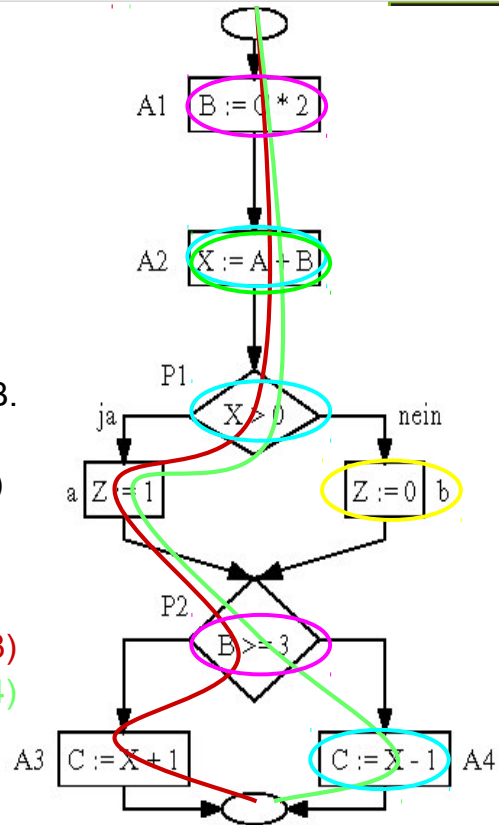
- Abschnitt 8.2 (S.215-221)
- Definition 8.2.2 – alle DR-Interaktionen (S.216)
- Abbildung 8.1 (S.216)

## Alle DR-Interaktionen

- T sei Testdatenmenge, die Wege (A1, A2, P1, a, P2, A3) und (A1, A2, P1, a, P2, A4) ausführt.
- T erfüllt Kriterium „**alle DR-Interaktionen**“ für Variablen X und B.
- Aber: Entscheidungskante *b* (= Konsequenz aus Referenz von X) nicht ausgeführt.

Weg (A1, A2, P1, a, P2, A3)

Weg (A1, A2, P1, a, P2, A4)



### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.2 – alle DR-Interaktionen (S.216)
- Abbildung 8.1 (S.216)



**Motivation:** Für Referenz einer Variablen im Entscheidungsknoten:  
Ausgang der Entscheidung wichtig.

**Kriterium „alle Referenzen“:**

- Ziel: **Alle ausgehenden Kanten** eines **Entscheidungsknotens** berücksichtigen.

Testdatenmenge  $T$  erfüllt Kriterium „**alle Referenzen**“ g.d.w.:

- Für jede Variable  $x$ , jede Definition von  $x$  im Knoten  $k$ , jede Referenz von  $x$  im Knoten  $l$ , die von Definition in  $k$  erreicht wird, und für jeden Nachfolgerknoten  $m$  von  $l$  die Wegemenge  $Wege(T)$  **mindestens ein** Wegstück  $u^*m=\{k,\dots,l,m\}$  enthält, wobei die Definition von  $x$  in  $k$  die Referenz von  $x$  in  $l$  über Weg  $u$  erreicht.

Hierbei:  $u^*m=\{k,\dots,l,m\}$  ist die Knotenfolge mit  $u=\{k,\dots,l\}$  gefolgt von Knoten  $m$ .

## Literatur:

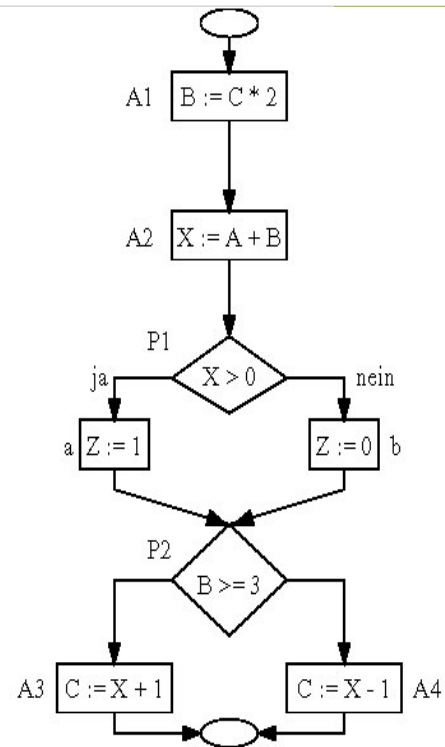
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.3 – alle Referenzen (S.217)

## Alle Referenzen: Beispiel

- **Beispiel:** Was sind Nachfolgerknoten von Referenzen von X und B ? Welche Testpfadmenge erfüllt „**alle Referenzen**“ (alle ausgehenden Kanten eines Entscheidungsknotens berücksichtigen) für X und B ?
- Testet sie alle Wege ?



### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.3 – alle Referenzen (S.217)
- Abbildung 8.1 (S.216)

## Alle Referenzen: Beispiel

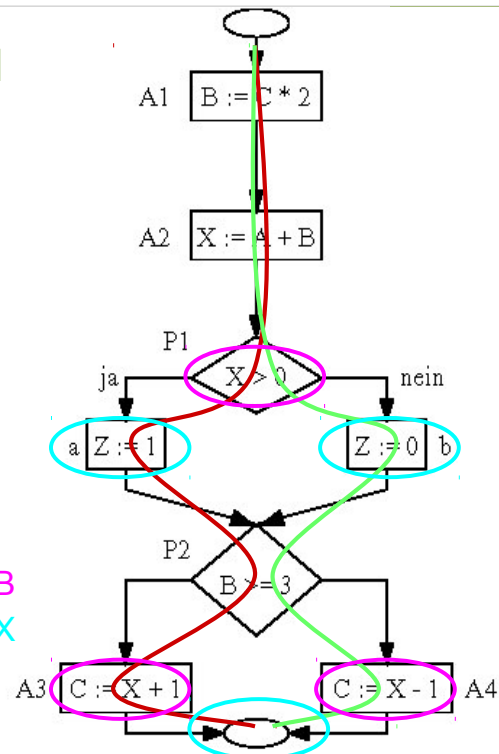
- T sei Testdatenmenge, die Wege (A1, A2, P1, a, P2, A3) und (A1, A2, P1, b, P2, A4) ausführt.
- T erfüllt Kriterium „**alle Referenzen**“ für Variablen X und B.
- **Aber:** Wege (A1, A2, P1, a, P2, A4) und (A1, A2, P1, b, P2, A3) nicht ausgeführt.

Nachfolgeknoten von Referenzen von B

Nachfolgeknoten von Referenzen von X

Weg (A1, A2, P1, a, P2, A3)

Weg (A1, A2, P1, b, P2, A4)



125

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.3 – alle Referenzen (S.217)
- Abbildung 8.1 (S.216)



- **Einfache Datenflusskriterien wie „Alle DR-Interaktionen“ und „alle Referenzen“: Kriterien zum Testen** aller Paare von Definitionen und Referenzen.
  - Jeweils **auf einem Weg** von Definition zur Referenz.
- Ausreichend unter Gesichtspunkt des Datenflusses.
  - Zwischen Definition und Referenz **keine Änderung der Variablen**.
- Etwas feinkörniger: zwischen **Entscheidungs-** und **Berechnungs-Referenzen** unterscheiden (nächste Folie).

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Diskussion (S.217-218)

## Variablen- / Objektverwendung: Computational vs. Predicative use

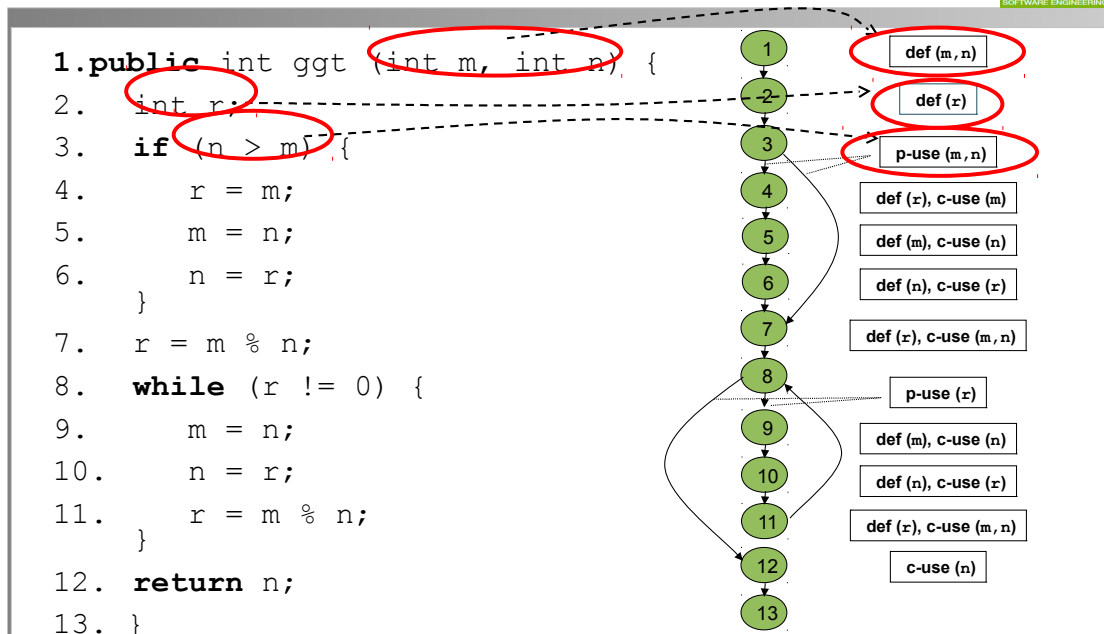


Mögliche Variablen- / Objektverwendung:

- Wertzuweisung, zustandsverändernd (**Definition / Definitional use**):
  - z.B. `r = m` oder `r = 5 : def(r)`
- Benutzung in Ausdrücken, zustandserhaltend: (**Berechnungs-Referenz / Computational use**):
  - z.B. `r = m mod n` oder `r = op1(m, n) : c-use (m, n)` und `def (r)`.
- Benutzung in Bedingungen, zustandserhaltend (**Entscheidungs-Referenz / Predicative use**):
  - z.B. `while (r != 0)` oder `if (r != 0) : p-use (r)`.

## Beispiel ggt: Datenflussgraph

Softwarekonstruktion  
WS 2014/15



### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.1 (S.212-215)
- Definition 8.1.1 – Datenflussgraph (S.213-214)

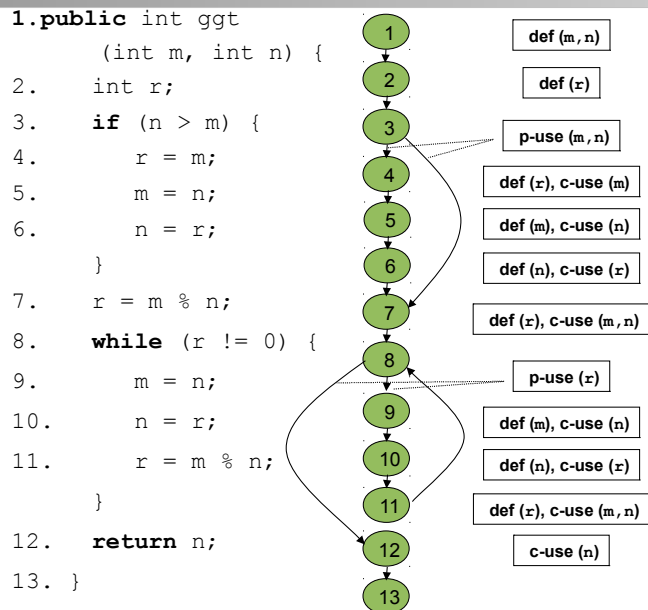




**All-defs:** Jede Definition min. einmal ohne dazwischen liegendes erneutes def in Referenz (c-use oder p-use) verwenden.

[Bem.: Egal ob c-use oder p-use, also konsistent mit Def. auf F. 114.]

Gibt es eine Testmenge, die All-defs erfüllt ?



## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)



Pfad (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13) erfüllt **All-defs** mit Ausnahme von „2: def(r)“ (nicht ohne dazwischen liegendes erneutes def verwendet):

- 1, def m: p-use 3-4
- 1, def n: p-use 3-4
- 4, def r: c-use 6
- 5, def m: c-use 7
- 6, def n: c-use 7
- 7, def r: p-use 8-9
- 9, def m: c-use 11
- 10, def n: c-use 11
- 11, def r: p-use 8-9

NB: „3-4“ zeigt an, welcher Zweig ausgehen von 3 in diesem Pfad genommen wurde (für All-defs aber nicht relevant.)

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
   }
7.   r = m % n;
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
   }
12.  return n;
13. }
                
```

## Literatur:

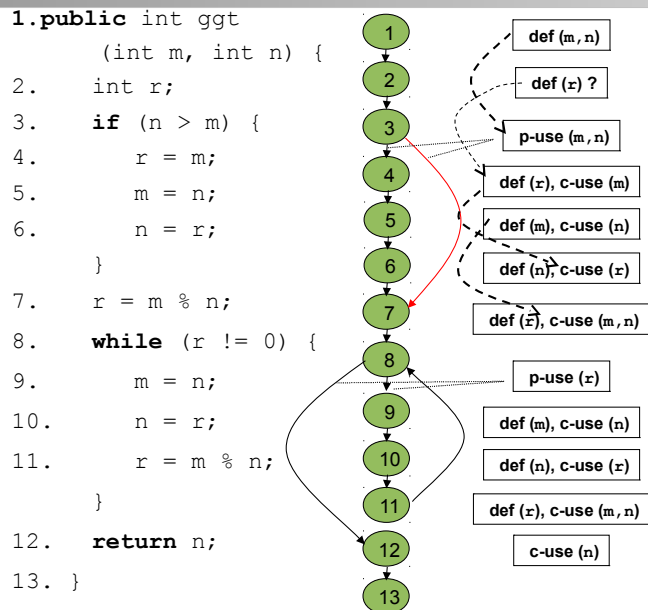
E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)



Erfüllt Pfad (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)  
auch **Alle DR-Interaktionen**  
[= jedes Paar def/ref (ohne dazwischen liegendes erneutes def ausführen) ?



131

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.1 – alle Definitionen (S.215)



**Alle DR-Interaktionen:** jedes Paar def/ref (ohne dazwischenliegendes erneutes def) ausführen:

1, def m: p-use 3-4

1, def m: c-use 4

1, def m: c-use 7

→ brauche else-Zweig 3-7 !

1, def n: p-use 3-4

1, def n: c-use 5

1, def n: c-use 7 (braucht 3-7 !)

4, def r: c-use 6

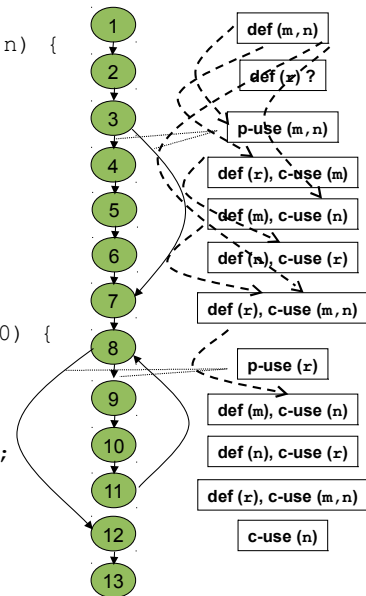
.....

→ Pfad reicht nicht, brauche weiteren Pfad für else-Zweig 3-7.

**NB:** Brauche keinen Pfad, der die Schleife 8-11 null mal ausführt, weil für „12: c-use(n)“ das vorhergehende „1/6: def(n)“ dann bereits durch „7: c-use(n)“ abgedeckt wird

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
7.   }
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
12.  }
13.  return n;
14. }
    
```



## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

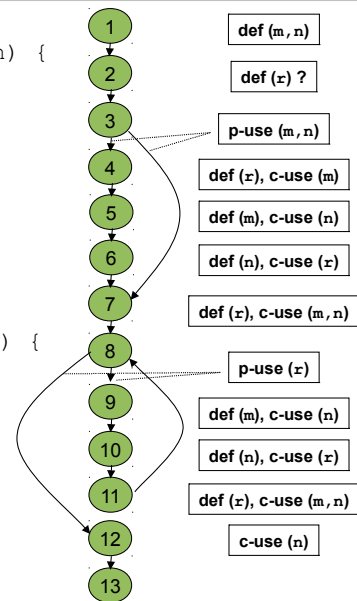
- Abschnitt 8.2 (S.215-221)
- Definition 8.2.2 – alle DR-Interaktionen (S.216)



Erfüllt die Pfadmenge von der vorherigen Folie **Alle-Referenzen** ?  
Reicht schon der einzelne Testfall von davor ?  
Was sind jeweils die Nachfolgerknoten ?

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
   }
7.   r = m % n;
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
   }
12.  return n;
13. }
    
```



133

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.3 – alle Referenzen (S.217)



**Alle B-Referenzen (c-uses):**

1, def m: c-use 4-5  
 1, def m: c-use 7-8  
 → brauche else-Zweig 3-7 !

1, def n: c-use 5-6  
 1, def n: c-use 7-8 (else!)  
 4, def r: c-use 6-7

.....

**Alle E-Referenzen (p-uses):**

1, def m p-use 3-4  
 1, def m p-use 3-7  
 → brauche else-Zweig 3-7 !

.....

→ Brauche beide Testfälle.

Nachfolgeknoten von p-uses von m.  
 Nachfolgeknoten von c-uses von m.

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
   }
7.   r = m % n;
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
   }
12.  return n;
13. }
        
```

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.3 – alle Referenzen (S.217)

# Beispiel ggt: Alle k-DR-Interaktionen



**Alle k-DR-Interaktionen:**  
Verkettung von DR-Paaren.  
**Beispiel für k=3:**

**(1, m, 4, r, 6):**

- m wird in 1 definiert, in 4 referenziert,
- r wird in 4 definiert, in 6 referenziert

→ damit hängt 6 von 1 ab!

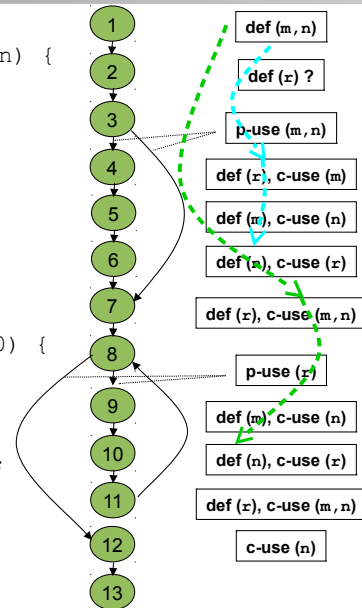
Analog für (1, m, 7, r, 10).

→ Teste Wegstücke

(1,2,3,4,5,6) und  
(1,2,3,7,8,9,10).

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
   }
7.   r = m % n;
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
   }
12.  return n;
13. }
    
```



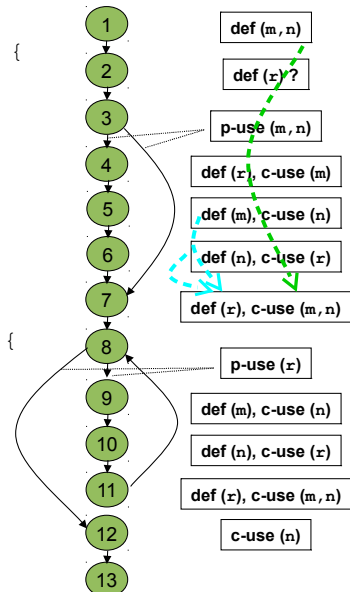
# Beispiel ggt: Kontextüberdeckung



Beispiel für  
**Kontextüberdeckung:**  
In 7 wird r definiert durch referenzierte Variablen m und n.  
„**Definitionskontext**“:  
Knoten, in denen m und n vorher definiert. Hier zwei Fälle:  
DK1= {(1,m),(1,n)}  
DK2= {(5,m),(6,n)}  
→ teste Wegstücke (1,2,7) und (5,6,7).

```

1. public int ggt
   (int m, int n) {
2.   int r;
3.   if (n > m) {
4.     r = m;
5.     m = n;
6.     n = r;
7.   }
8.   while (r != 0) {
9.     m = n;
10.    n = r;
11.    r = m % n;
12.  }
13.  return n;
14. }
    
```







- „**alle Definitionen**“ (**all-defs**):
  - Jede Definition min. einmal (ohne dazwischenliegendes erneutes def) im c-use oder p-use verwenden.
- „**alle DR-Interaktionen**“:
  - Jedes Paar def/ref (ohne dazwischenliegendes erneutes def) auf irgendeinem Weg ausführen.
- Variation: **k-DR-Interaktionen**
- **Kontextüberdeckung**

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)



- **Alle Referenzen:**
  - Alle ausgehenden Kanten eines Entscheidungsknotens berücksichtigen.
- **Alle Entscheidungs-/einige Berechnungs-Referenzen:**
  - Schwächere Überdeckung des Datenflusses, trotzdem Zweigüberdeckung und Testen „aller Definitionen“.
- **Alle Berechnungs-/einige Entscheidungs-Referenzen:**
  - Keine Zweigüberdeckung, aber Testen „aller Definitionen“ und „aller Referenzen“ in Berechnungsknoten.
- **Alle DR-Wege:**
  - Stärkere Überdeckung unter Kontrollflussaspekten durch Annäherung an Pfadtesten, allerdings ohne Schleifeniterationen.

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)



### Anzahl Testdaten pro Testkriterium:

Falls **Anzahl ausgehender Kanten pro Entscheidungsknoten** und **Anzahl Variablen pro Segment** durch Konstante **begrenzt**, dann für Programm mit  $n$  Segmenten **im schlechtesten Fall**:

- „**alle Definitionen**“:  $O(n)$  Testdaten.
- „**alle E- / einige B-Referenzen**“, „**alle B- / einige E-Referenzen**“, „**alle Referenzen**“, Kontextüberdeckung:  $O(n^2)$  Testdaten.
- „**alle DR-Wege**“:  $O(2^n)$  Testdaten.

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 8.2 (S.215-221)
- Definition 8.2.5 (S.215)
- Abschnitt 10.2 Anzahl der Testdaten pro Testkriterium (ab S.257)



Aufgedeckte Fehler pro Testkriterium (vgl. [Rie97]):

Kriterium	Fehler erkannt
Alle Definitionen	24%
Alle E- / einige B-Referenzen	34%
Alle B- / einige E-Referenzen	48%
Alle DR-Interaktionen	51%

„**Alle B-Referenzen**“: z.B. bis zu 88% aller Berechnungsfehler.

„**Alle E-Referenzen**“: z.B. 100% aller Bereichsfehler.

Für restliche Kriterien keine Studien vorhanden.

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 10.3 (ab S.260)
- Abschnitt 10.3.1 (S.261-262)
- Abschnitt 10.3.2.2 (S.267)



Folgende **Fehler schlecht aufdeckbar**:

- Fehlende Pfade.
- Bereichsfehler durch falsch platzierte Anweisungen und falsche arithmetische Operatoren.
- Berechnungsfehler bei speziellen Werten.

Aber: ca. **9% aller Fehler** nur **mit datenflussbezogenen Methoden** findbar.

[Quelle: Riedemann: Spezialvorlesung „Software-Testmethoden“]

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

- <http://www.ub.tu-dortmund.de/katalog/titel/687299>
- Abschnitt 10.3 (ab S.260)
- Abschnitt 10.3.2.2 (S.267)

# Überblick strukturelles Testen: Ansätze und Überdeckungskriterien

Softwarekonstruktion  
WS 2014/15



## **Kontrollflussbasiert:**

- **Anweisungsüberdeckung** ( $C_0$ -Überdeckung)
- **Zweigüberdeckung** ( $C_1$ -Überdeckung).
- **Grenze-Inneres-Überdeckung** ( $C_{gi}$ ).
- **Pfadüberdeckung** ( $C_\infty$ -Überdeckung).

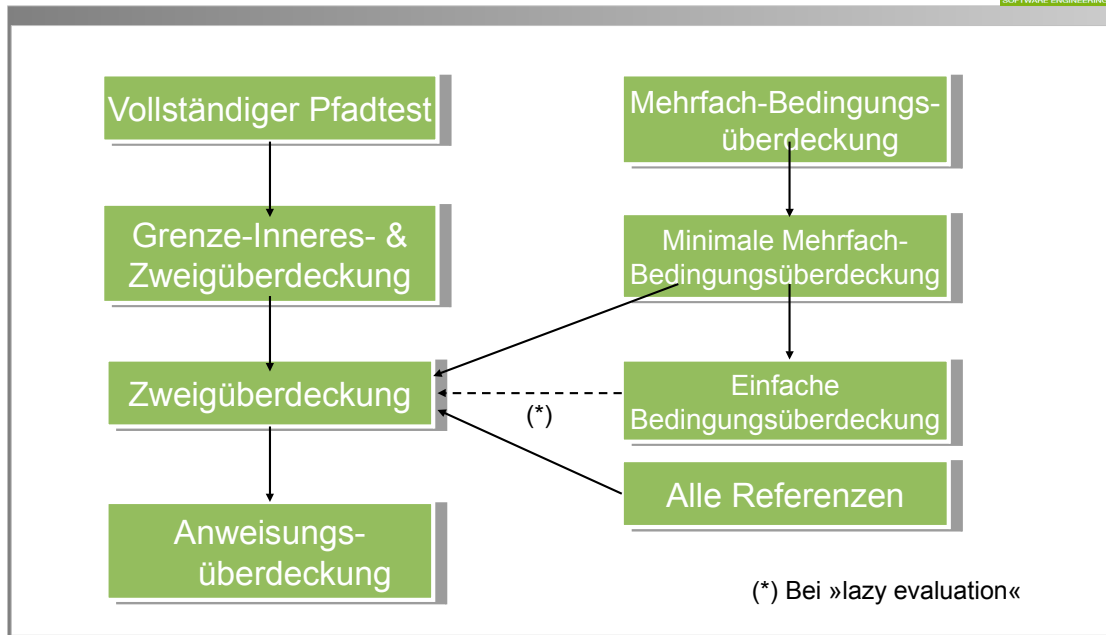
## **Bedingungsbasiert:**

- **Einfache Bedingungsüberdeckung.**
- **Mehrfachbedingungsüberdeckung.**
- **Minimal bestimmende Mehrfachbedingungsüberdeckung.**

## **Datenflussbasiert:**

- **Alle Definitionen** (all defs).
- **Alle Definition-Benutzung-Paare** (all def-uses).
- ...

# Mächtigkeit der White-Box-Techniken



# Bewertung der White-Box-Techniken



Überdeckungsmaß	Leistungsfähigkeit	Bewertung
Anweisungsüberdeckung ( $C_0$ )	Niedrig Entdeckt knapp ein Fünftel der Fehler	Notwendig, aber nicht hinreichend Entdeckt »dead-code«
Zweigüberdeckung (Entscheidungsüberdeckung, $C_1$ )	Mittel, schwankt aber stark Entdeckt ca. 30% aller Fehler und ca. 80% der Kontrollfluss-Fehler	Entdeckt nicht ausführbare Zweige Zielt auf Verzweigungen
Bedingungsüberdeckung ( $C_2$ )	Niedrig	Umfasst i.Allg. nicht die Anweisungs- und Entscheidungsüberdeckung
Grenze-Inneres Test ( $C_{GI}$ )	Mittel	Ergänzendes Kriterium nur für Schleifen
Mehrfach-Bedingungsüberdeckung	Hoch	Zielt auf komplexe Bedingungen Umfasst Entscheidungsüberdeckung Aufwand wächst stark
Datenflusstest	Mittel bis hoch, All c-uses ca. 50%, All p-uses ca. 34%, all defs ca. 25% (keine Berechnungsfehler!)	Zielt auf Variablen-Verwendung c-uses findet viele Berechnungsfehler
Pfadüberdeckung ( $C_\infty$ )	Sehr hoch Entdeckt über 70% der Fehler	In den meisten Fällen nicht praktikabel



## Einige Warnungen zum Thema Testüberdeckungen (1)

Softwarekonstruktion  
WS 2014/15



<http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/testinstallation/articles/247210>

„Acht Irrtümer über Code Coverage“:

- Irrtum 1: Man kann immer 100% Abdeckung erreichen.
  - z.B. nicht-erreichbarer Code
- Irrtum 2: Ein Coverage-Maß hat nur einen Namen.
  - z.B. minimale Mehrfachüberdeckung
- Irrtum 3: Ein Name bezeichnet immer dasselbe Coverage-Maß.
  - z.B. C0, C1, C2 unterschiedlich verwendet
- Irrtum 4: Es ist klar, wie Coverage gemessen wird.
  - Werkzeuge können Definitionen verschieden interpretieren

## Einige Warnungen zum Thema Testüberdeckungen (2)



- Irrtum 5: Für die Coverage ist es egal, wie der Code formuliert ist.
  - z.B. zusammengesetzte Bedingungen vs. kaskadierende IF-Ausdrücke
- Irrtum 6: Durch geschickte Programmierung kann man sich das Leben erleichtern.
  - Ziel ist nicht Kennzahlen-Optimierung sondern Fehler-Findung
- Irrtum 7: Reicht, Testfälle zur vollständigen Code-Abdeckung aus Code abzuleiten.
  - z.B. fehlende Code-Abschnitte
- Irrtum 8: Code-Coverage misst die Qualität des Codes.
  - 100% Überdeckung heisst nicht 100% fehlerfrei.

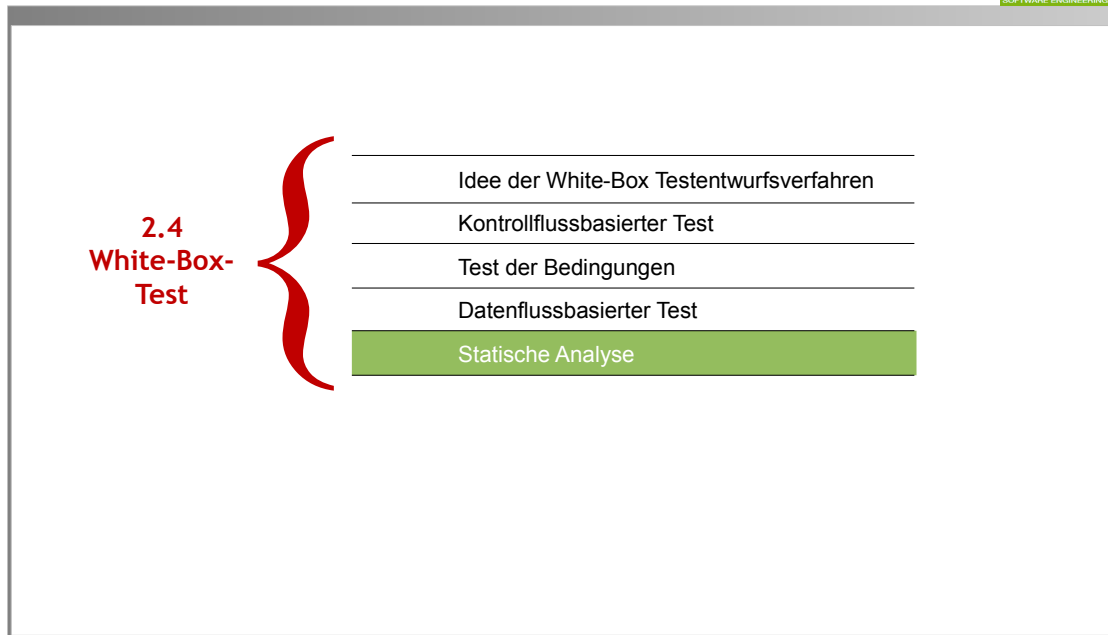
# Zusammenfassung: Dynamischer Test



- **Ziel:** Mit wenig Aufwand ausreichend unterschiedliche Testfälle erzeugen.  
→ Mit gewisser Wahrscheinlichkeit vorhandene Fehlerzustände zur Wirkung bringen.
- **Ausführung der Testfälle:** Codebasierte Überdeckung messen.
- **Obere Teststufen: Black-Box** Testentwurfsverfahren.
- **Untere Teststufen: White-Box** Testentwurfsverfahren.

Weil vollständiges Testen nicht möglich: Testpriorisierung notwendig.

Ein Ansatz: Statische Analyse → vgl. nächster Unterabschnitt !



### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)



Was haben wir bislang über das Testen gelernt ?

- Völliges Austesten nicht möglich (vgl. Kap. 2.0).
- Trade-off zwischen Testaufwand und Anteil gefundener Fehler (niemals 100%).
- Kontrollfluss- und Datenflussbasierte Überdeckungskriterien helfen bei Trade-off, aber garantieren keine 100%-ige Fehlerfreiheit.



**Lösung:** Verifikationsansätze, die nicht (nur) auf Ausführung des Programmes (= dynamisches Testen) beruhen:

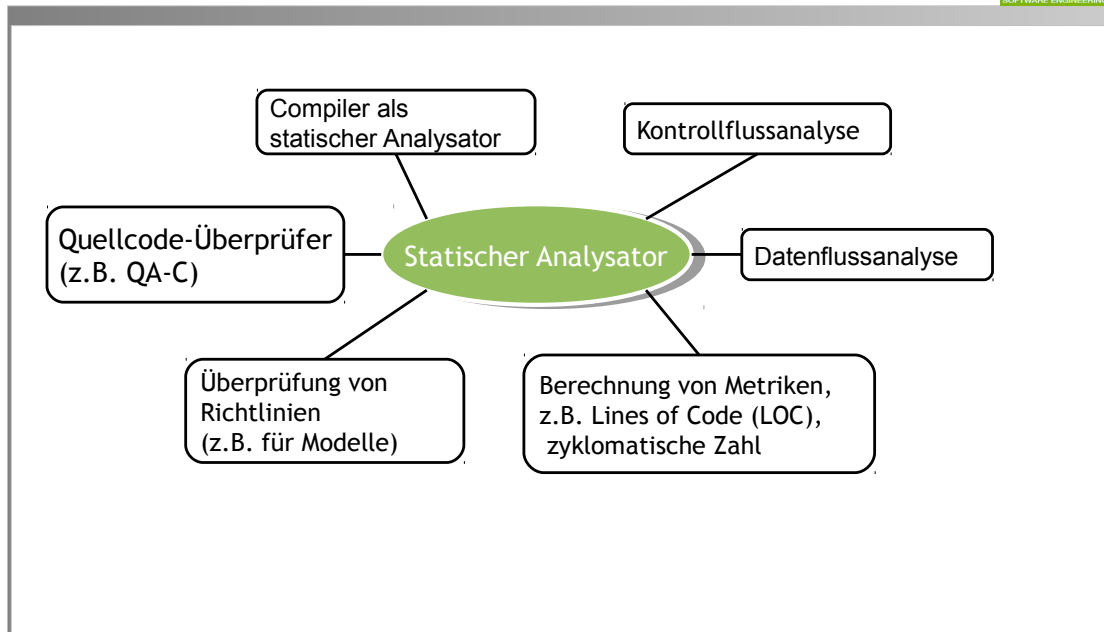
- **Statische Analyse:** vollautomatisch, aber nur bestimmte Fehlerklassen
  - z.B. Analyse des Kontrollflussgraphen auf Anomalien
- **Symbolische Ausführung**
- **formale Verifikation:**
  - vollautomatisch (z.B. **Modell-Checking**): leichtere Bedienung, eingeschränkte Mächtigkeit
  - oder teilautomatisiert (z.B. **interaktives Theorembeweisen**): anspruchsvolle Bedienung, prinzipiell (beinahe) uneingeschränkte Mächtigkeit

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

Abschnitt 4.2 – statische Analyse (S.99-108)



## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.1 – Compiler (S.101)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)
- Abschnitt 4.2.4 – Kontrollflussanalyse (S.104)
- Abschnitt 4.2.5 – Metriken (S.105-106)



- Suche nach Anomalien im Programmtext.
- **Anomalie:** Unstimmigkeit, die zur Fehlerwirkung führen kann.
  - Anomal: unregelmäßig, regelwidrig.
  - Kann Fehlerzustand sein, muss aber nicht.
- **Statische Analyse:** Nicht alle Fehlerzustände einfach nachweisbar (**Fehlerzustände** als Fehlerwirkung bei Ausführung).
  - Z.B.: bei Division Wert des Divisors in Variable halten
    - Variable kann **zur Laufzeit** Wert Null annehmen.
    - Fehlerwirkung, statisch nicht einfach erkennbar.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)





Einfach aber effektiv: **Manuelle** Analyse des **Kontrollflussgraphen** auf **Anschaulichkeit**.

- **Ziel:** Abläufe durch Programmstück leicht manuell erfassen.
- Teile des Graphen unübersichtlich
  - Zusammenhänge und Ablauf kaum nachvollziehbar.
  - Fehlerträchtig (und schlecht wartbar).
  - Überarbeitung des Programmtextes.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)
- Abschnitt 4.2.4 – Kontrollflussanalyse (S.104)



- **Kontrollflussanomalie:** Statisch feststellbare Unstimmigkeit beim Ablauf des Testobjekts:
  - Sprünge aus Schleifen heraus
  - Sprünge in Schleifen hinein
  - Programmstücke mit mehreren Ausgängen
- Müssen **keine Fehlerzustände** sein, **widersprechen aber Grundsätzen** strukturierter Programmierung und können fehlerträchtig sein.
- Kontrollflussgraph von Werkzeug generieren (gewährleistet **eins-zu-eins-Abbildung** zwischen Programmtext und Graph).

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.4 – Kontrollflussanalyse (S.104)



**Vorgänger-Nachfolger-Tabellen:** Beziehung der Anweisungen (Ausführungsabfolge).

Kein Vorgänger einer Anweisung. → Anweisung nicht erreichbar.

- **Fehlerzustand** ist erkannt.

**Keine Vorgänger- bzw. Nachfolge-Anweisung** für:

- Erste und letzte Anweisung eines Programm(teil)s.
- Programm(teil)e mit mehreren Eintritts- bzw. Austrittspunkten.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.4 – Kontrollflussanalyse (S.104)

# Vorgänger-Nachfolger-Tabelle: Beispiel ggt



	Anweisung	Nachfolger	Vorgänger
1. <b>public</b> int ggt(int m, int n) {	1	2	-
2.   int r;	2	3	1
3. <b>if</b> (n > m) {	3	4, 7	2
4.     r = m;	4	5	3
5.     m = n;	5	6	4
6.     n = r;	6	7	5
}	7	8	3, 6
7.   r = m % n;	8	9, 12	7, 11
8. <b>while</b> (r != 0) {	9	10	8
9.     m = n;	10	11	9
10.    n = r;	11	8	10
11.    r = m % n;	12	13	8
}	13	-	12
12. <b>return</b> n;			
13. }			



- Verwendung von Daten auf »Pfad« durch Programm (vgl. datenflussbasiertes Testen)

```
1. public int ggt(int m, int n){  
2.     int r;  
3.     if (n > m) {  
4.         r = m;  
5.         m = n;  
6.         n = r;     ...  
}
```

- Aufdeckung von Datenflussanomalien:
  - **Referenzierende Verwendung (Lesen) einer Variablen** ohne vorherige Initialisierung
  - **Nicht-Verwendung** eines zugewiesenen Wertes einer Variablen

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



## **Datenfluss-Zustände** von Variablen:

- **Undefiniert (u)**: Variable hat keinen definierten Wert.
- **Definiert (d)**: Variablen wird Wert zugewiesen.
- **Referenziert (r)**: Wert der Variablen wird verwendet.

## Damit Definition der **Datenflussanomalien**:

- **ur**-Anomalie: Undefinierter Wert (u) einer Variablen wird gelesen (r).
- **du**-Anomalie: Variable erhält Wert (d) und wird ohne Verwendung ungültig (u).
- **dd**-Anomalie: Variable erhält Wert (d) und ohne Verwendung einen zweiten Wert (d).

## **Literatur:**

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



```
1. void Tausch (int min, int max) { // d(min, max)
2.   int hilf; // u(hilf) (Java: hilf=0)
3.   if (min > max) { // r(min, max)
4.     max = hilf; // d(max), r(hilf)
5.     max = min; // d(max), r(min)
6.     hilf = min; // d(hilf), r(min)
7.   }
8. } // u(hilf)
```

Welche Anomalien sehen Sie ?

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



```
1. void Tausch (int min, int max) { // d(min, max)
2.   int hilf; // u(hilf) (Java: hilf=0)
3.   if (min > max) { // r(min, max)
4.     max = hilf; // d(max), r(hilf)
5.     max = min; // d(max), r(min)
6.     hilf = min; // d(hilf), r(min)
7.   }
8. } // u(hilf)
```

**ur-Anomalie** der Variablen `hilf`:

- **Gültigkeitsbereich** der Variablen auf Funktion beschränkt.
- Erste Verwendung der Variablen auf der rechten Seite einer Zuweisung.
- Variable hat hier **undefinierten Wert**, der referenziert wird.
- **Initialisierung** bei Deklaration der Variablen nicht vorgenommen.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)





```
1. void Tausch (int min, int max) { // d(min, max)
2.   int hilf; // u(hilf) (Java: hilf=0)
3.   if (min > max) { // r(min, max)
4.     max = hilf; // d(max), r(hilf)
5.     max = min; // d(max), r(min)
6.     hilf = min; // d(hilf), r(min)
7.   }
8. } // u(hilf)
```

dd (max)

**dd-Anomalie** der Variablen max:

- **Zwei Zuweisungen** ohne zwischenzeitige Verwendung.

Mögliche Ursachen bzw. Korrektur:

- Entweder: erste Zuweisung kann entfallen.
- Oder: Verwendung des ersten Wertes vergessen worden.
- Oder: Variablennamen / Reihenfolgen vertauscht.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



```
1. void Tausch (int min, int max) { // d(min, max)
2.   int hilf; // u(hilf) (Java: hilf=0)
3.   if (min > max) { // r(min, max)
4.     max = hilf; // d(max), r(hilf)
5.     max = min; // d(max), r(min)
6.     hilf = min; // d(hilf), r(min)
7.   }
8. } // u(hilf)
```

du (hilf)

#### du-Anomalie der Variablen hilf

- Variable hilf bekommt Wert zugewiesen, wird nirgends verwendet.
- Variable nur **innerhalb der Funktion** gültig.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



Nicht jede Anomalie führt direkt zu **fehlerhaftem Verhalten**:

- du-Anomalie: keine direkte Auswirkungen, Programm kann korrekt laufen.
- Genauere Untersuchung der anomalen Programmstellen, weitere Unstimmigkeiten ausfindig machen.

### Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



Im Beispiel: Anomalien **offensichtlich**.

**Aber:** Zwischen Anweisungen, die Anomalie führen, können beliebig viele andere Anweisungen stehen.

- Anomalien nicht mehr offensichtlich.
- Bei manueller Prüfung übersehbar.
- Werkzeug zur Datenflussanalyse deckt Anomalien auf.

## Literatur:

A. Spillner, T. Linz: **Basiswissen Softwaretest**

<http://www.ub.tu-dortmund.de/katalog/titel/1287855>

- Abschnitt 4.2 – statische Analyse (S.99-108)
- Abschnitt 4.2.3 – Datenflussanalyse (S.102-104)



„Ausführung“ eines Programms mit symbolischen Werten:

- Verwendung **symbolischer** statt konkreter Werte für Eingabevariablen.
- Mit diesen wird entsprechend ‚gerechnet‘ (z.B. unter Verwendung mit Logik-formalisierter Annahmen an die Werte, algebraische Eigenschaften der verwendeten Operatoren etc.).

## Beispiel

Anweisung:  $C := A + 2 B$ .

Schreibe symbolische Werte als  $w()$ .

Symbolische Ausführung der obigen Zuweisung ergibt:

$w(C) = w(A) + 2 w(B)$ .

## Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 12.3 (S.327-333)
- Beispiel 12.3.1 (S.327)



- **Symbolischer “Test“** deckt Vielzahl normaler Testdaten ab.
  - Vollständiges Testen wird prinzipiell möglich.
- Insbesondere werden **Fehler entdeckt**, bei denen für Teilmenge der Eingaben Ergebnisse falsch berechnet werden.
- Im Gegensatz zu formaler Programmverifikation keine **formale Programmspezifikation** erforderlich.

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 12.3 (S.327-333)
- Vorteile (S.330)

## Symbolische Ausführung: Nachteile

Softwarekonstruktion  
WS 2014/15



- Benötige **formale Definition** der Programmiersprache.
- Bislang kein vollständiger Ersatz, sondern Ergänzung für funktionsorientierte Tests (insbesondere für sicherheitskritische Systeme).
- Theorembeweiser nötig für logische Berechnungen während „Ausführung“.
- Kein **vollständiger** automatischer Theorembeweiser für mächtige Programmiersprachen vorhanden.  
→ Teilweise nur Approximation des Programmergebnisses möglich (aber immer noch umfassender als bei Testen).

### Literatur:

E. Riedemann: **Testmethoden für sequentielle und nebenläufige Software-Systeme**

<http://www.ub.tu-dortmund.de/katalog/titel/687299>

- Abschnitt 12.3 (S.327-333)
- Nachteile (S.330)



- **Referenzierung** einer Variablen mit nicht definiertem Wert.
- **Inkonsistente Schnittstellen** zwischen Modulen und Komponenten.
- Variablen, die nie verwendet werden.
- Unerreichbarer Code (»**dead code**«).
- Verletzung von Programmierkonventionen.
- **Sicherheitsschwachstellen.**
- **Syntax-Verletzungen** von Code und Softwaremodellen.
- ...



## Diskussion: Nutzen statischer Analysen



- **Frühe Erkennung** von Fehlerzuständen vor Testdurchführung.
- Frühe Warnung vor verdächtigen Aspekten in Code / Design.  
→ Berechnung von Metriken.
- **Identifizierung** von Fehlerzuständen.  
→ Durch dynamischen Test nicht effektiv aufzudecken.
- **Aufdecken von Abhängigkeiten** und Inkonsistenzen in Softwaremodellen.
- **Verbesserte Lesbarkeit**, Änderbarkeit und Wartbarkeit von Code und Design.
- **Vorbeugung** von Fehlerzuständen.



In **diesem** Abschnitt:

- White-Box-Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- Statische Analyse

Im **nächsten** Abschnitt:

- **Testen im Softwarelebenszyklus:** Teststufen und -arten.

# Zusammenfassung: White-Box Testentwurfsverfahren

Softwarekonstruktion  
WS 2014/15



**Grundlage** aller White-Box Testentwurfsverfahren: **Vorliegender Programmtext.**

Abhängig von Komplexität der Programmstruktur adäquate Testentwurfsverfahren auswählbar.

- Anhand Programmtextes und ausgewählten Testentwurfsverfahren Intensität der Tests festlegen (Ausgangskriterium/Testendekriterium).

**Problem:** »Nicht vorhandener Programmcode« bleibt unberücksichtigt.

- **Übersehene Anforderungen** durch White-Box Testentwurfsverfahren **nicht aufdeckbar.**
- Nur im Programm umgesetzte Anforderungen beim White-Box Testentwurfsverfahren überprüfbar.

Zur **Instrumentierung** **Werkzeug** verwenden.

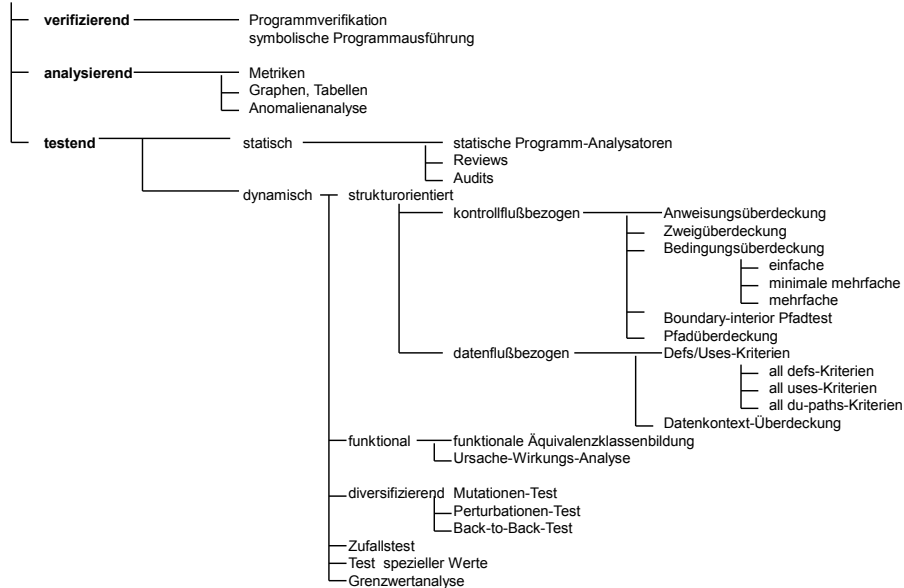
# Worin unterscheiden sich Testentwurfsverfahren?



<b>TESTART</b>	Welche Art von Tests werden durchgeführt?	Funktionale Anforderungen, Nicht-funktionale Anforderungen
<b>TESTSTUFE</b>	Für welche Testobjekte wird der Test spezifiziert?	Komponententest, Integrationstest, Systemtest, Abnahmetest
<b>TESTER</b>	WER testet?	Entwickler, (erfahrene) Tester, Benutzer, ...
<b>TESTABDECKUNG</b>	WAS wird abgedeckt?	Anforderungen, Anweisung, Entscheidung, ...
<b>POTENZIELLE FEHLER</b>	Welche potenziellen Fehler sollen identifiziert werden?	Behandlung von Grenzwerten Behandlung von Ausnahmen ...
<b>ARTEFAKT</b>	Was ist Grundlage für Auswahl und Herleitung der Testfälle?	Anforderungen -> Black-Box Code -> White-Box ...
<b>DOMÄNE / PARADIGMA</b>	Für welche spezielle Domäne bzw. Entwicklungsparadigma ist die Technik zugeschnitten?	OOP, Web-basiert, DB-basiert, Automotive, Sicherheitskritische SW, ...



### Übersicht Prüfverfahren:



173

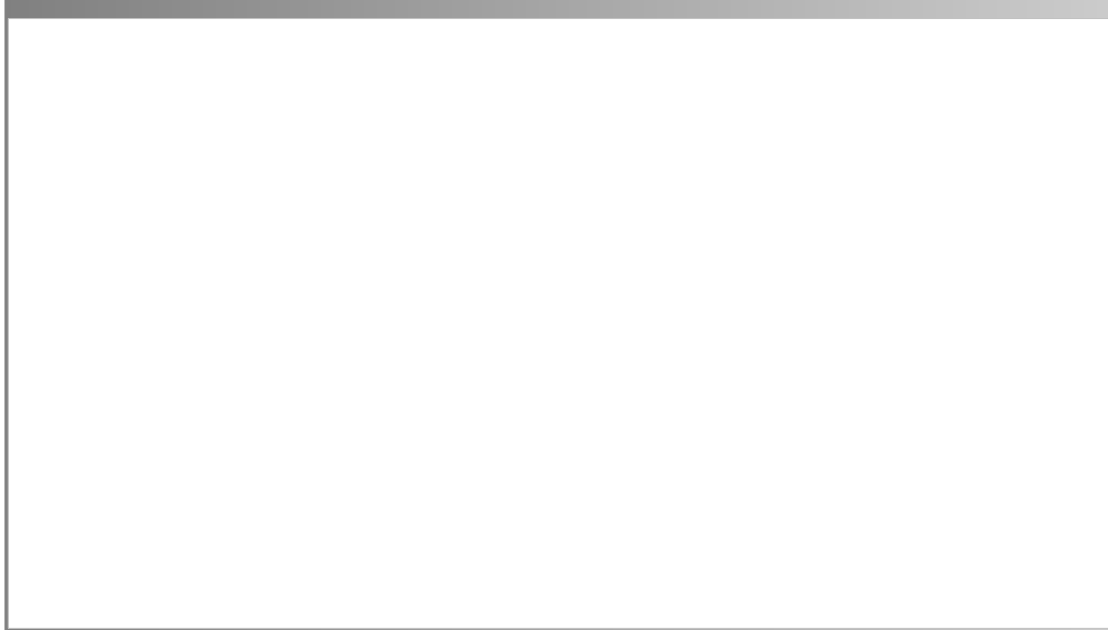
## Literatur:

H. Balzert: **Lehrbuch der Software-Technik/2 – Software-Management, Software-Qualitätssicherung**  
<http://www.ub.tu-dortmund.de/katalog/titel/645541>

- Teil III – Abschnitt 2.1 (S.278-284)
- Qualitätsprüfung (S.283)

# Anhang (weitere Einzelheiten und Beispiele)

Softwarekonstruktion  
WS 2014/15



Nur bis Anhang (Folie 111) generieren



**White-Box-Test:** Test, der auf Analyse interner Struktur einer Komponente oder eines Systems basiert.

»**Fehleraufdeckende**« **Stichproben** möglicher Programmabläufe und Datenverwendungen suchen.

**White-Box-Testentwurfverfahren:** Dokumentiertes Verfahren zur Herleitung und Auswahl von Testfällen, basierend auf interner Struktur einer Komponente oder eines Systems.

Alle **Testentwurfverfahren**, die zur

- Herleitung der Testfälle
- **Bestimmung der Vollständigkeit der Prüfung** (Überdeckungsgrad)

Information über innere Struktur des Testobjekts (z.B. Zweige) heranziehen.

→ Strukturorientierte, strukturbezogene oder strukturelle Testentwurfverfahren.



- **Kontrollfluss:** Abstrakte Repräsentation von Reihenfolgen von Ereignissen während Programmausführung.
  - Abfolge ausgeführter Anweisungen.
  - Muss nicht Reihenfolge der Anweisungen im Programmtext sein.
- **Determinierung** durch »steuernde« Anweisungen:
  - Unbedingte Sprünge
  - Bedingte Verzweigungen
  - Schleifen
- **Steuerung** durch Wahrheitswerte von Bedingungen:
  - Ermittelter Wahrheitswert:
    - »wahr«: → Ausführung des Programms mit THEN-Teil der IF-Anweisung
    - »unwahr«: → Durchlaufung des ELSE-Teil.
- **Schleifen** führen zu vorherigen Anweisungen zurück.
  - Bewirkt **mehrfache Ausführung** oder Schleifenkörper wird umgangen.

176





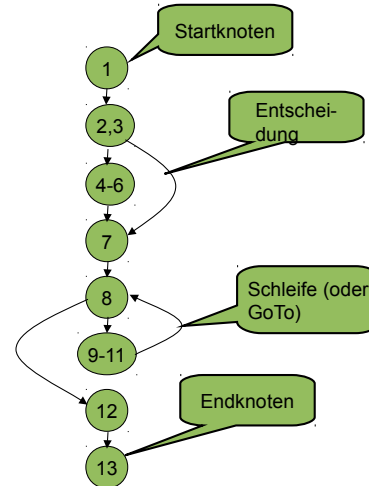
- Gerichteter Graph.
- **Knoten:** Anweisungen des Programms  
**Sequenzen von Anweisungen:** Darstellung als einziger Knoten (Block), falls
  - Sequenz nur durch ihren ersten Knoten betretbar.
  - Keine Änderung des Programmablaufes innerhalb Sequenz.→ Genau dann, wenn erste Anweisung der Sequenz ausgeführt wird, werden alle weiteren Anweisungen der Sequenz ausgeführt
- **Kanten:** Ausführungsreihenfolgen die durch Knoten repräsentierten Anweisungen:
  - Knoten mit mehreren ausgehenden Kanten repräsentieren Kontrollfluss steuernde Anweisungen.
  - IF-Anweisung und Schleifensteuerung: Zwei abgehende Kanten.
  - CASE-Anweisung: Mehrere abgehende Kanten.
- Genau ein **Startknoten**.
- Genau ein **Endknoten**.

177



## Bestimmung des größten gemeinsamen Teilers (ggT) zweier ganzer Zahlen m und n:

```
1. public int ggt(int m, int n) {  
2.     int r;  
3.     if (n > m) {  
4.         r = m;  
5.         m = n;  
6.         n = r;  
7.     }  
8.     while (r != 0) {  
9.         m = n;  
10.        n = r;  
11.        r = m % n;  
12.    }  
13. }  
14. }
```





**Minimale Mehrfachbedingungsüberdeckung ( $C_2$ (mM)-Überdeckung):**

X (A>1)	Y (B=0)	X and Y
false	false	false
false	true	false
true	false	false
true	true	true

X (A=2)	Y (C>1)	X or Y
false	false	false
false	true	true
true	false	true
true	true	true

Welche **Testkombination** muss jeweils **nicht getestet werden** ?



**Minimale Mehrfachbedingungsüberdeckung ( $C_2$ (mM)-Überdeckung):**

X (A>1)	Y (B=0)	X and Y
false	false	false
false	true	false
true	false	false
true	true	true

**Alle Kombinationen bis auf erste** testen, da bei dieser Änderung des Wahrheitswertes eines Terms **keine Änderung des Wahrheitswertes** der Kombination **bewirkt**.

X (A=2)	Y (C>1)	X or Y
false	false	false
false	true	true
true	false	true
true	true	true

**Alle Kombinationen bis auf letzte** testen, da bei dieser Änderung des Wahrheitswertes eines Terms **keine Änderung des Wahrheitswertes** der Kombination **bewirkt**.



### Minimale Mehrfachbedingungsüberdeckung ( $C_2$ (mM)-Überdeckung):

X (A>1)	Y (B=0)	X and Y
false	false	false
false	true	false
true	false	false
true	true	true

Testdatum  
A=3,B=0,C=3  
Testdatum  
A=2,B=1,C=0  
Testdatum  
A=1,B=0,C=2

X (A=2)	Y (C>1)	X or Y
false	false	false
false	true	true
true	false	true
true	true	true

# Werkzeugunterstützung für Kontrollflusstesten (1)

Softwarekonstruktion  
WS 2014/15



## Bullseye Coverage (Freeware):

<http://bullseyecoverage.software.informer.com>

Features: Funktionsüberdeckung, Zweigüberdeckung

## Quilt (Freeware): <http://quilt.sourceforge.net>

Features: Wieviel % des Codes überdeckt.

## Emma (Freeware):

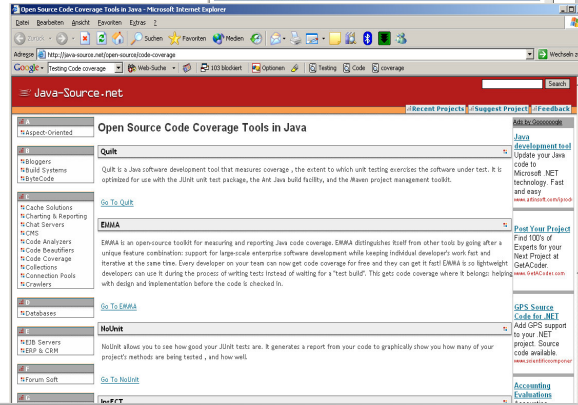
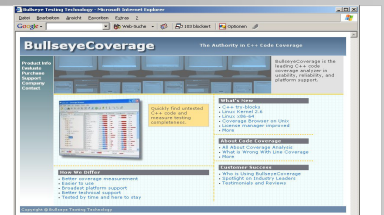
<http://emma.sourceforge.net>

Features: Testabdeckung; Erkennung von toten Programmteilen.

## NoUnit (Freeware):

<http://nunit.sourceforge.net>

Features: Zeigt wie gut Junit Tests sind.



# Werkzeugunterstützung für Kontrollflusstesten (2)



## Tessy (Freeware):

<http://www.hitex.com/index.php?id=code-coverage-in-tessy>

Features: Analysiert  $C_1$ - und  $C_2$ -Überdeckung.

## Hansel (Freeware): <http://hansel.sourceforge.net>

Features: Gibt an wie viel Prozent des angenommenen Tests wirklich getestet wurde.

## Testwell CTC++:

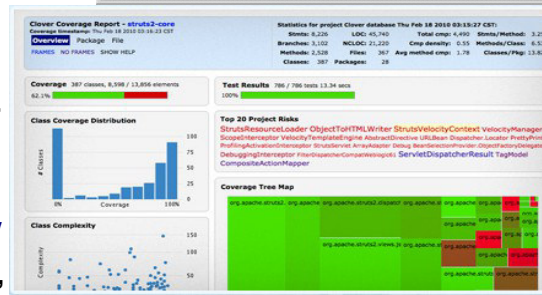
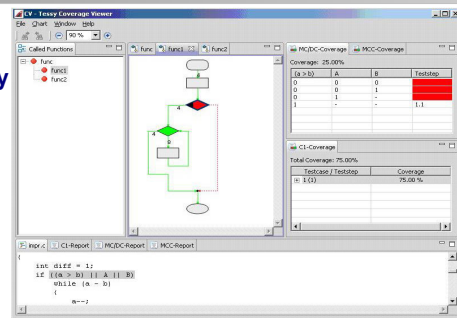
[http://www.verifysoft.com/de\\_ctcpp.html](http://www.verifysoft.com/de_ctcpp.html)

Features: Analysiert alle Testabdeckungsstufen.

## Clover:

<http://www.atlassian.com/software/clover/overview>

Features: Analysiert Anweisungs-, Zweig-, Entscheidungs- & Pfadüberdeckung.





**Ziel:** Aufdeckung vorhandener Fehlerzustände im Dokument.

- »**Statische Analyse**«: Beinhaltet keine Ausführung der Prüfobjekte.

Beispiele:

- Rechtschreibprüfprogramme als Art statische Analysatoren, die Fehler in Texten nachweisen. → Trägt zur **Qualitätsverbesserung** bei.
- Compiler führen statische Analyse des Programmtextes durch: Prüfen Einhaltung der Syntax der Programmiersprache.
- Spezielle Werkzeuge prüfen Einhaltung von **Programmierkonventionen**.

**Weiteres Ziel:** Ermittlung von Messgrößen oder Metriken, um Qualitätsbewertung durchzuführen. → Qualität messen.





- Statische Analyse: Mit **Werkzeugunterstützung** sinnvoll !
- Zu analysierendes Dokument: Muss formaler Struktur unterliegen, um durch Werkzeug überprüft werden zu können.
- Dokumente, die Formalismus unterliegen können, z. B.:
  - Technische Anforderungen
  - Softwarearchitektur
  - Softwareentwurf
- Informeller Text unterhalb Rechtschreibung und elementarer Grammatik nur mit KI analysierbar.
  - **Linguistische semantische Analyse:** aktueller Forschungsgegenstand.
  - **Aber:** Einführung von »**Normsprache**« ermöglicht einfachere Analysen.



- **Compiler:**
  - Führt statische Analyse des Programmtextes durch.
  - Prüft Einhaltung der Syntax der jeweiligen Programmiersprache.
  - Bietet zusätzliche Informationen.
- **Analysatoren:**
  - Einsatz zur gezielten Durchführung der Gruppen von Analysen
- **Fehler(zustände):**
  - Verletzung der Syntax.
  - Abweichungen von Konventionen und Standards.
  - Sicherheitslücken.
  - Kontrollflussanomalien.
  - Datenflussanomalien.



- **Fehler:** Verletzung der Syntax der Programmiersprache.
- Weitere Überprüfungen:
  - Verwendung einzelner Programmelemente.
  - Prüfung **typgerechter Verwendung** der Daten und Variablen bei streng typisierten Programmiersprachen.
  - Ermittlung von nicht deklarierten Variablen.
  - **Unerreichbarer Code.**
  - Fehler bei Über- oder Unterschreitung von Feldgrenzen.
  - Prüfung der **Konsistenz von Schnittstellen.**
  - Prüfung der Verwendung aller Marken als Sprunganweisung und Sprungziel.
- **Ergebnisse** in Form von Listen.
- Fehlerzustand nicht immer auffindbar.  
→ **Weitere Untersuchungen.**

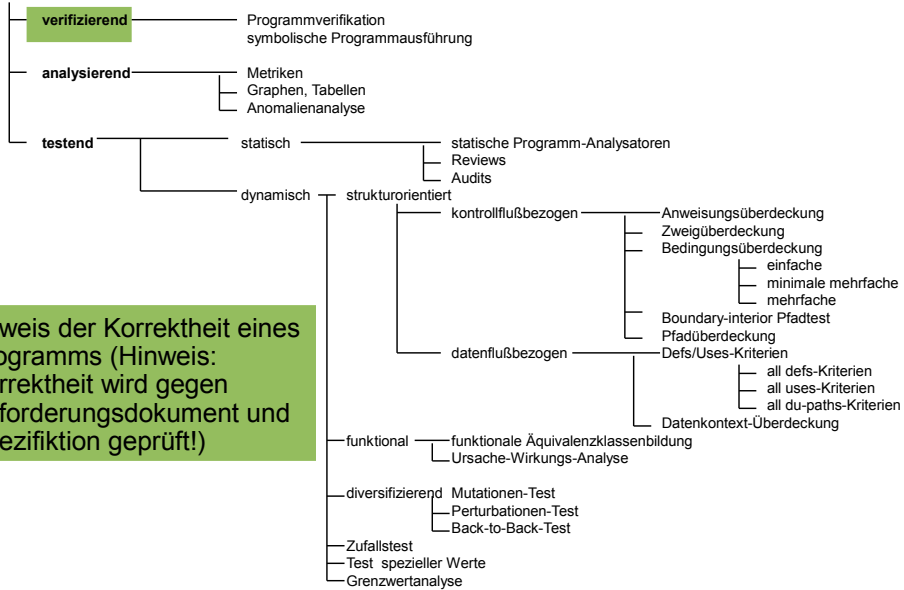
Fix	Severity	Abbreviation	Explanation
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members
F	High	UPCM	Unused Private Class Members



- **Sicherheitsprobleme** durch:
  - Verwendung bestimmter fehleranfälliger Programmkonstrukte.
  - Fehlende notwendige Überprüfungen.
- Beispiele:
  - **Fehlendes Abfangen** von Speicherüberläufen.
  - **Keine Überprüfung des Einhaltens** von Datenbeschränkungen bei Eingabe.
- **Analysewerkzeuge** können diese Mängel aufdecken.
  - Unterliegen einem bestimmten „Muster“.
  - Werden von Werkzeugen gesucht und analysiert.



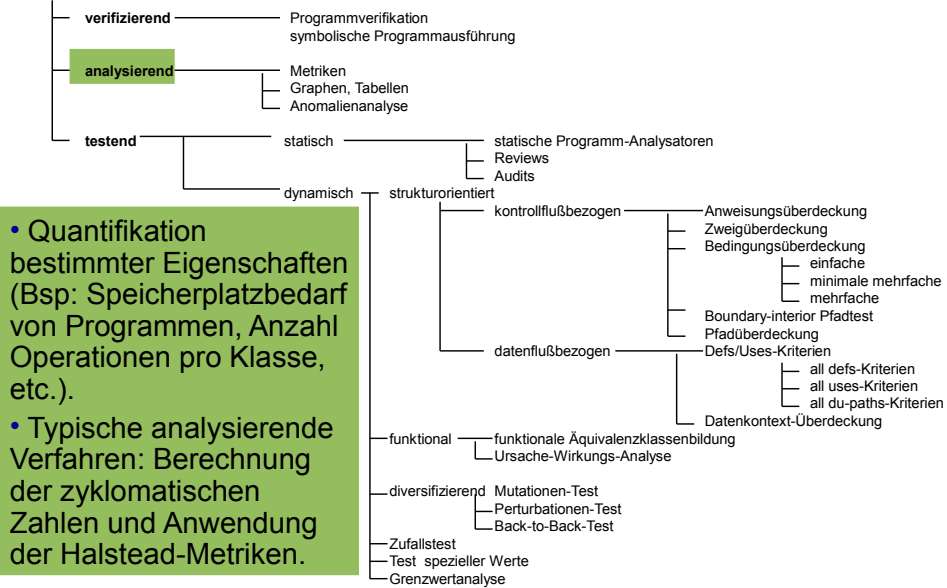
### Übersicht Prüfverfahren:



**Beweis der Korrektheit eines Programms (Hinweis: Korrektheit wird gegen Anforderungsdokument und Spezifikation geprüft!)**



### Übersicht Prüfverfahren

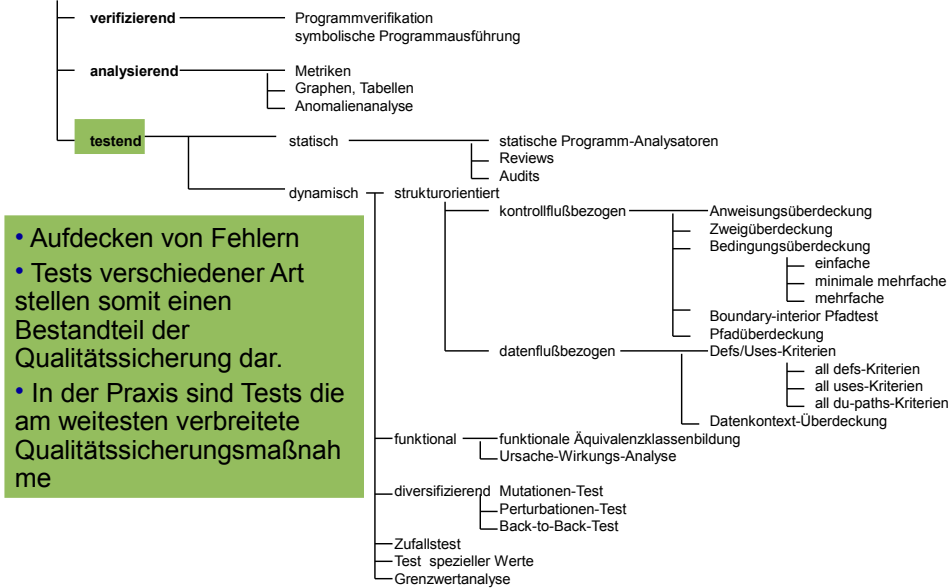


• Quantifikation bestimmter Eigenschaften (Bsp: Speicherplatzbedarf von Programmen, Anzahl Operationen pro Klasse, etc.).

• Typische analysierende Verfahren: Berechnung der zyklomatischen Zahlen und Anwendung der Halstead-Metriken.



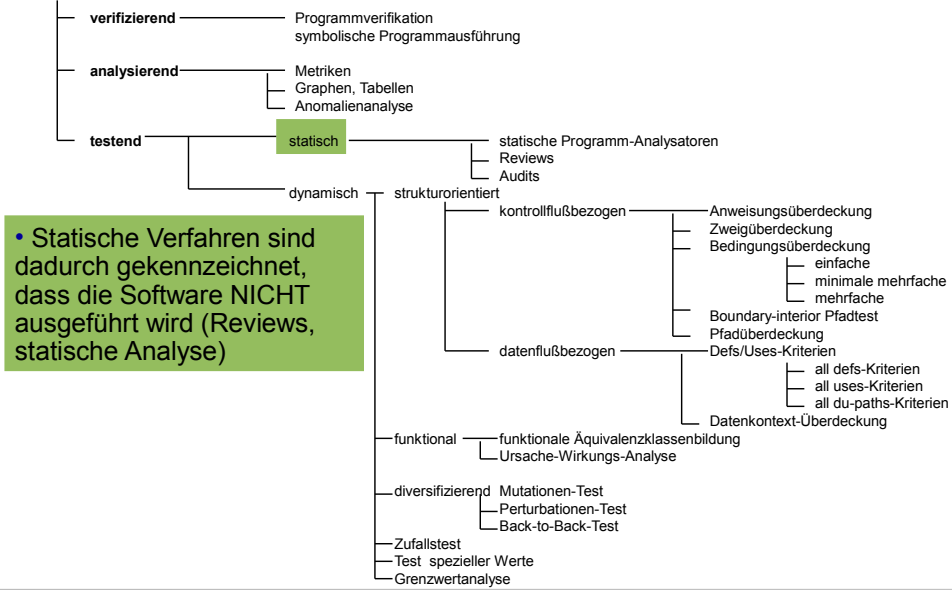
### Übersicht Prüfverfahren:



- Aufdecken von Fehlern
- Tests verschiedener Art stellen somit einen Bestandteil der Qualitätssicherung dar.
- In der Praxis sind Tests die am weitesten verbreitete Qualitätssicherungsmaßnahme



### Übersicht Prüfverfahren:

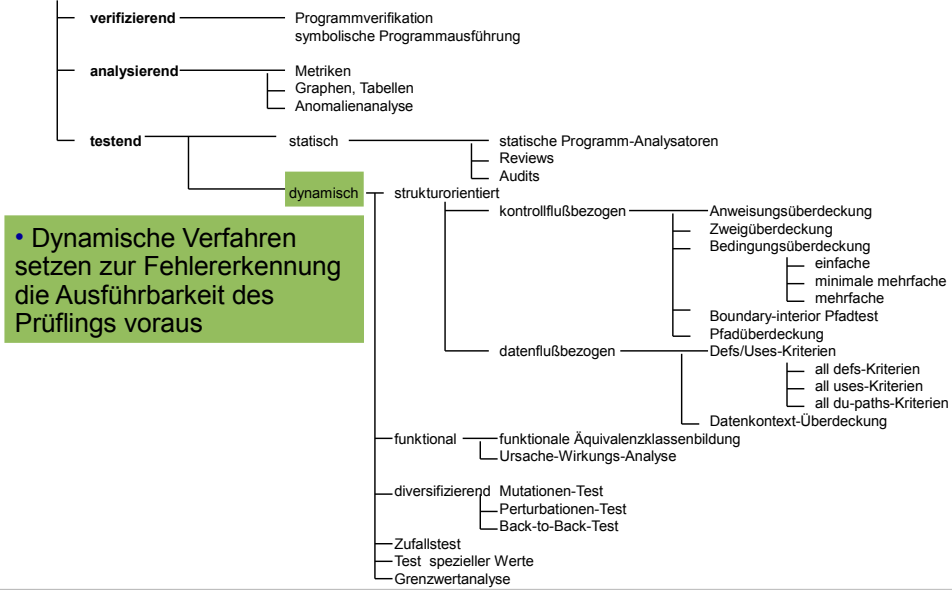


• Statische Verfahren sind dadurch gekennzeichnet, dass die Software NICHT ausgeführt wird (Reviews, statische Analyse)





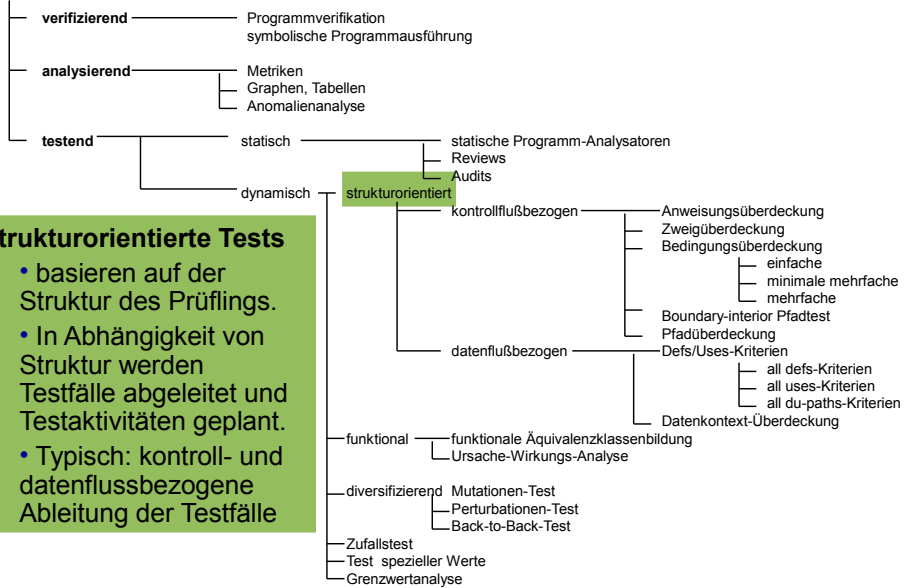
### Übersicht Prüfverfahren:



• Dynamische Verfahren setzen zur Fehlererkennung die Ausführbarkeit des Prüflings voraus



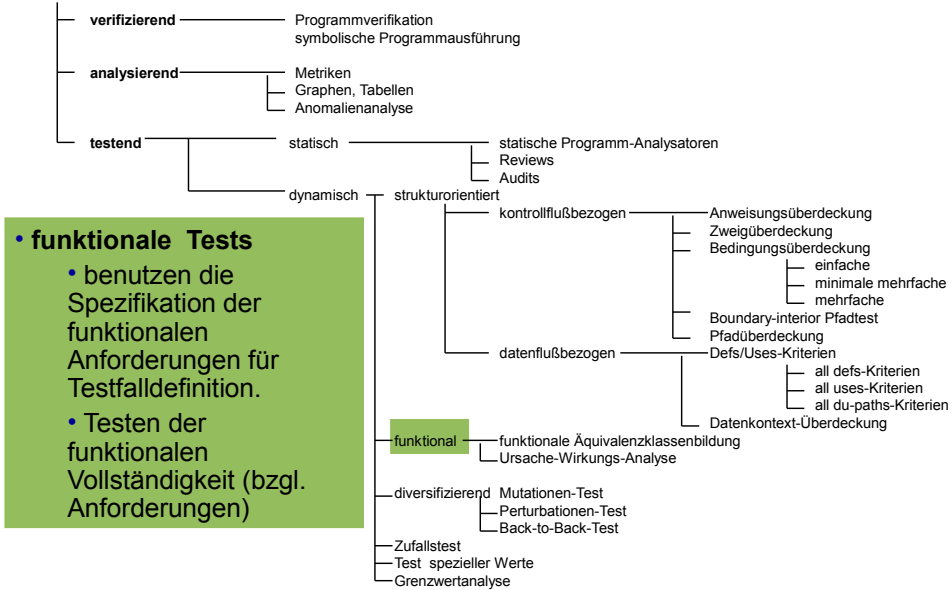
### Übersicht Prüfverfahren:



- **strukturorientierte Tests**
  - basieren auf der Struktur des Prüflings.
  - In Abhängigkeit von Struktur werden Testfälle abgeleitet und Testaktivitäten geplant.
  - Typisch: kontroll- und datenflussbezogene Ableitung der Testfälle



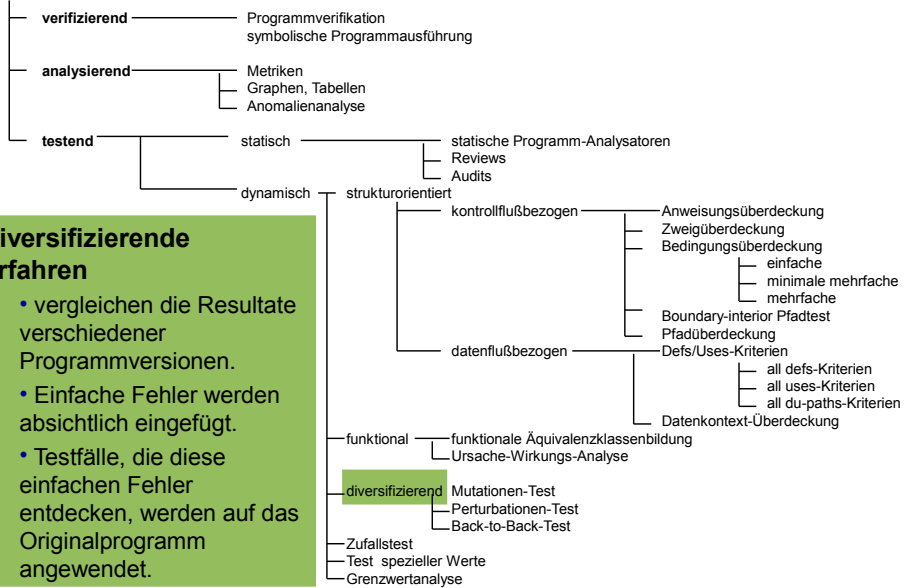
## Übersicht Prüfverfahren:



- **funktionale Tests**
  - benutzen die Spezifikation der funktionalen Anforderungen für Testfalldefinition.
  - Testen der funktionalen Vollständigkeit (bzgl. Anforderungen)



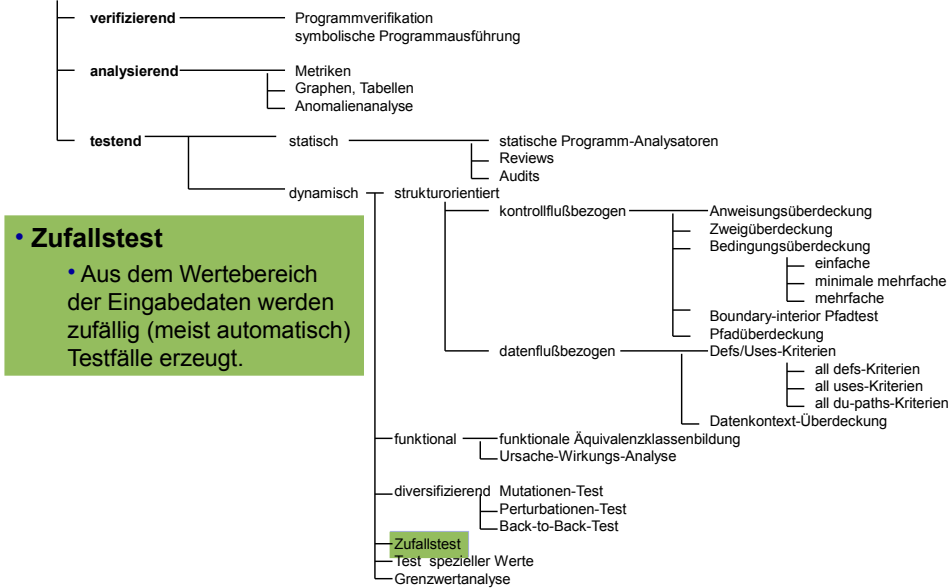
### Übersicht Prüfverfahren:



- **diversifizierende Verfahren**
  - vergleichen die Resultate verschiedener Programmversionen.
  - Einfache Fehler werden absichtlich eingefügt.
  - Testfälle, die diese einfachen Fehler entdecken, werden auf das Originalprogramm angewendet.



### Übersicht Prüfverfahren:

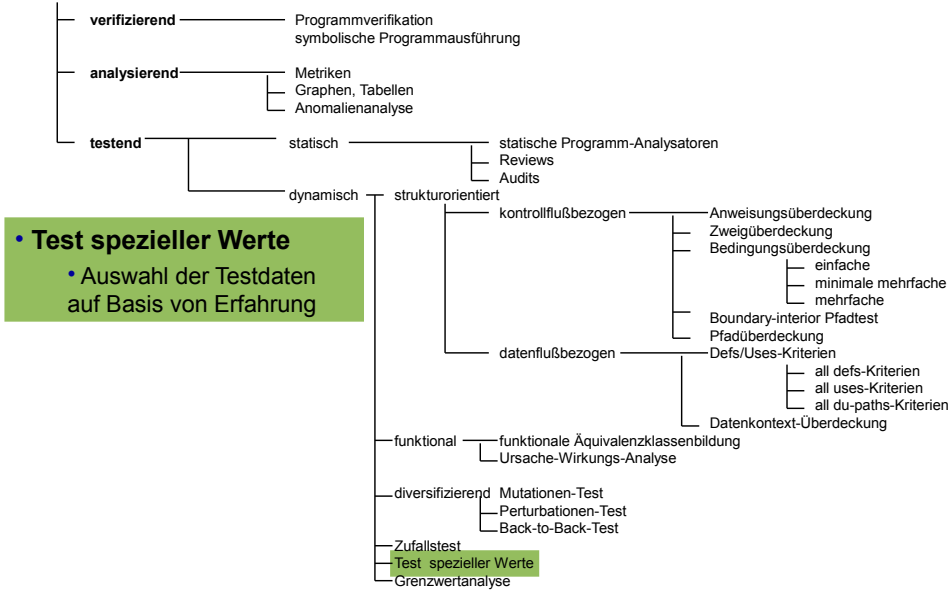


• **Zufallstest**

- Aus dem Wertebereich der Eingabedaten werden zufällig (meist automatisch) Testfälle erzeugt.



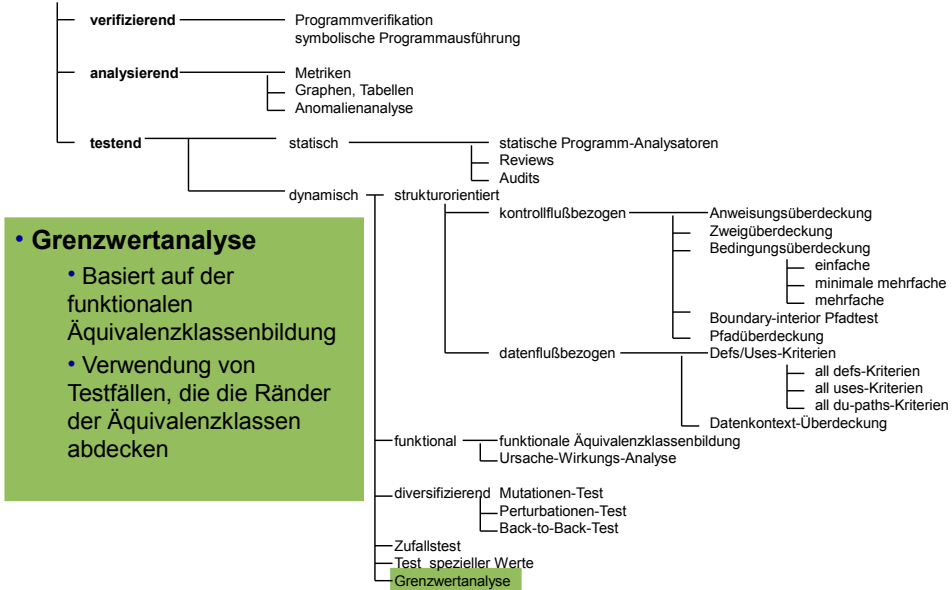
### Übersicht Prüfverfahren:



• **Test spezieller Werte**  
• Auswahl der Testdaten auf Basis von Erfahrung



### Übersicht Prüfverfahren:



• **Grenzwertanalyse**

- Basiert auf der funktionalen Äquivalenzklassenbildung
- Verwendung von Testfällen, die die Ränder der Äquivalenzklassen abdecken