

Willkommen zur Vorlesung  
*Softwarekonstruktion*  
im Wintersemester 2012 / 2013

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

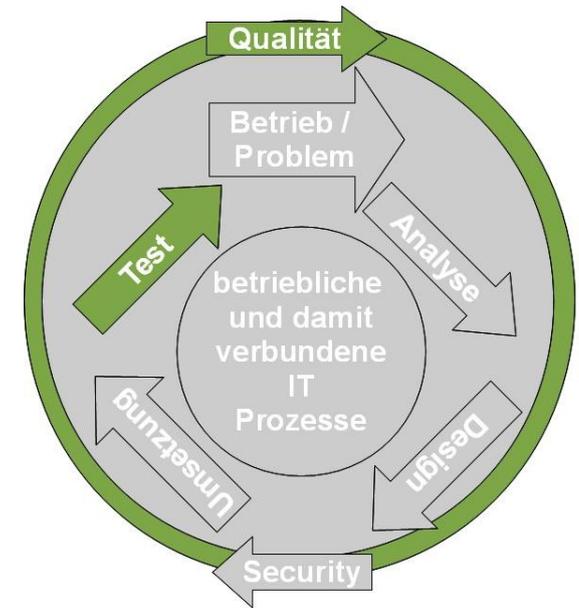
## 2.3 White-Box-Test

Basierend auf dem Foliensatz  
„Basiswissen Softwaretest - Certified Tester“  
des „German Testing Board“  
(nach Certified Tester Foundation Level Syllabus,  
deutschsprachige Ausgabe, Version 2011)  
(mit freundlicher Genehmigung)

© Copyright 2007 – 2013 by GTB  
V 2.0 / 2011

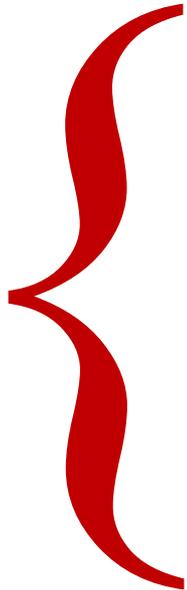
Der zum Kapitel 2 (Testen) der Vorlesung gehörende Foliensatz ist als Werk urheberrechtlich geschützt durch das German Testing Board; d.h. die Verwertung ist – soweit sie nicht ausdrücklich durch das Urheberrechtsgesetz (UrhG) gestattet ist – nur mit Zustimmung der Berechtigten zulässig. Der Foliensatz darf nicht öffentlich zugänglich gemacht oder im Internet frei zur Verfügung gestellt werden.

- Qualitätsmanagement
- **Testen**
  - Einführung
  - Grundlagen Softwaretesten
  - Testen im Softwarelebenszyklus
  - **White-Box-Test**
  - Black-Box-Test
  - Statischer Test
  - Test-Management
  - Testwerkzeuge
- Modellgetriebene SW-Entwicklung





## 2.3 White-Box- Test



Idee der White-Box Testentwurfverfahren

Kontrollflussbasierter Test

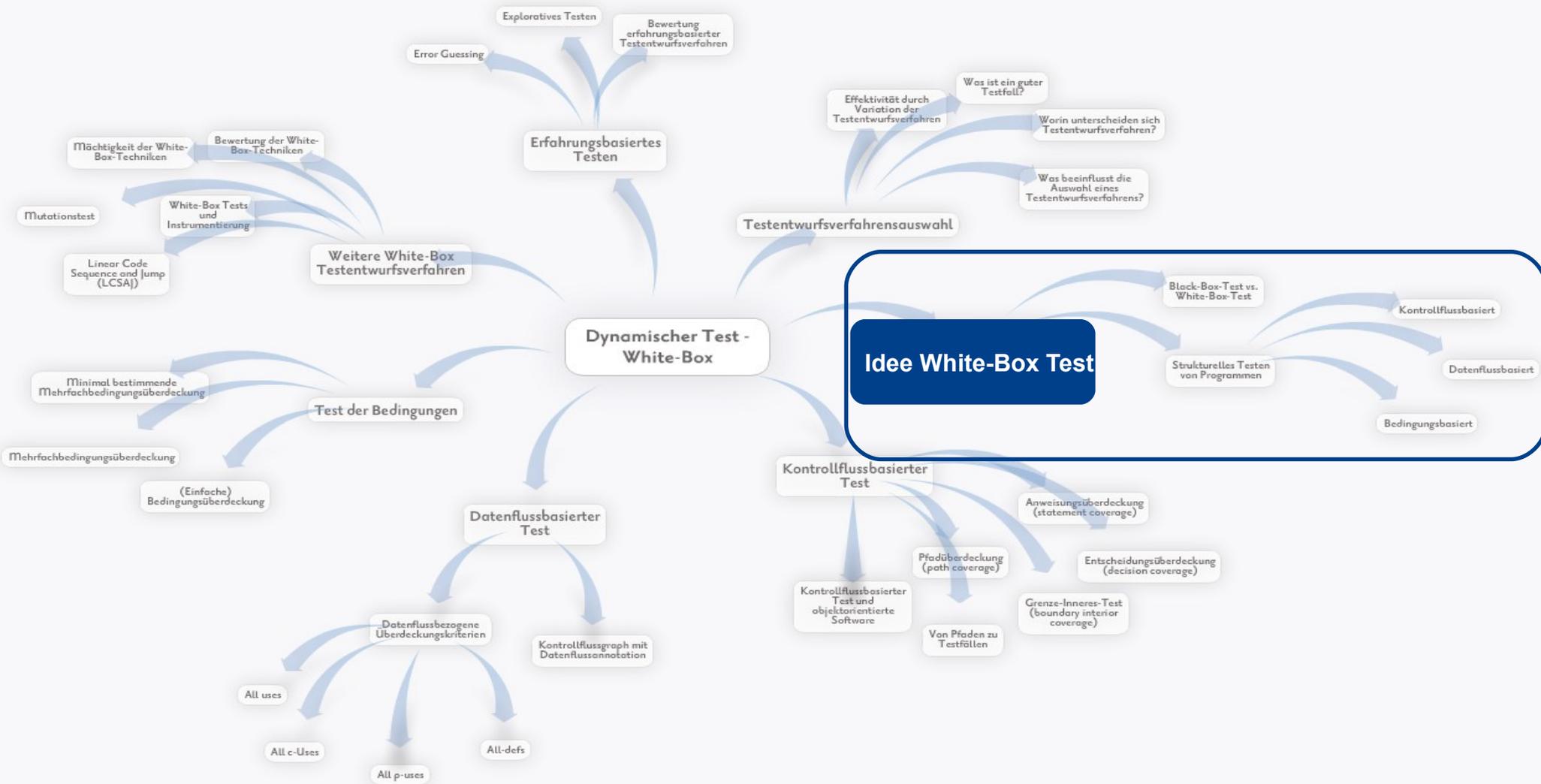
Datenflussbasierter Test

Test der Bedingungen

Weitere White-Box Testentwurfverfahren

Erfahrungsbasiertes Testen

Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung



# Black-Box Test vs. White-Box Test

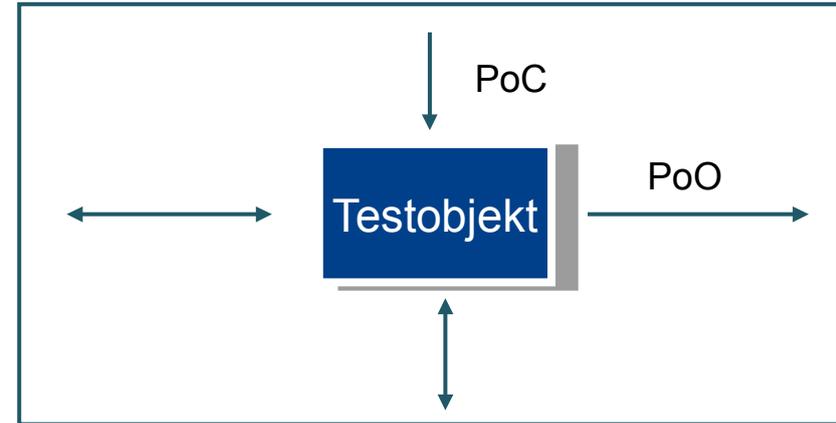
## Black-Box Test

Eingabewerte  
Ohne Kenntnis der  
Programmlogik  
abgeleitet



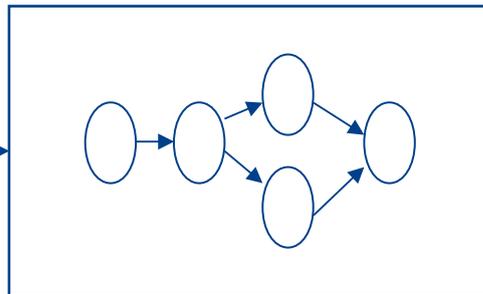
Istergebnis

PoC = Point of Control  
PoO = Point of Observation

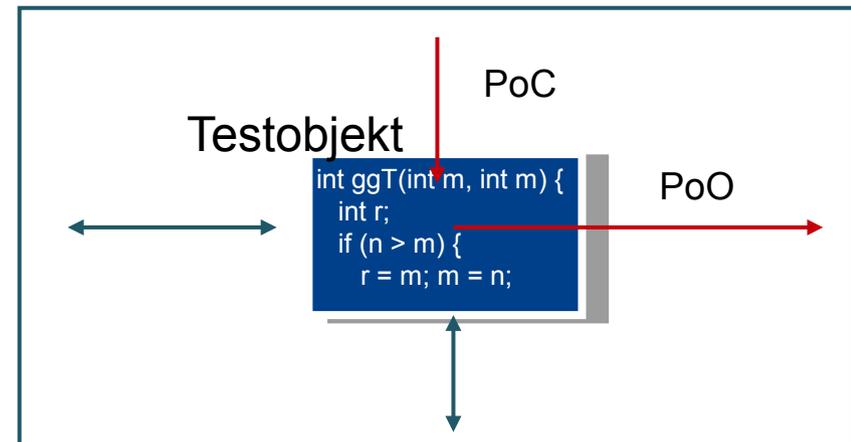


## White-Box Test

Eingabewerte  
Mit Kenntnis der  
Programmlogik  
abgeleitet



Istergebnis



**White-Box-Test:** Ein Test, der auf der Analyse der internen Struktur einer Komponente oder eines Systems basiert.

Gesucht werden »fehleraufdeckende« Stichproben der möglichen Programmabläufe und Datenverwendungen.

**White-Box-Testentwurfungsverfahren:** Ein dokumentiertes Verfahren zur Herleitung und Auswahl von Testfällen, basierend auf der internen Struktur einer Komponente oder eines Systems.

Alle Testentwurfungsverfahren, die zur Herleitung oder Auswahl der Testfälle sowie zur Bestimmung der Vollständigkeit der Prüfung (Überdeckungsgrad) Information über die innere Struktur des Testobjekts (z.B. Zweige, Pfade, Daten) heranziehen.

Daher auch strukturorientierte, strukturbezogene oder strukturelle Testentwurfungsverfahren genannt.

## Kontrollflussbasiert:

- Anweisungsüberdeckung (C0-Überdeckung, alle Knoten des Kontrollflussgraphen)
- Entscheidungsüberdeckung (C1-Überdeckung, bei Knoten mit mehr als einer ausgehenden Kante alle diese Kanten; auch: alle Zweige des Kontrollflussgraphen, Zweigüberdeckung)
- Grenze-Inneres-Überdeckung (Cgi, alle Schleifen einmal abgewiesen, einmalig ausgeführt und wiederholt ausgeführt)
- Pfadüberdeckung ( $C_{\infty}$ -Überdeckung, alle Pfade)

## Datenflussbasiert:

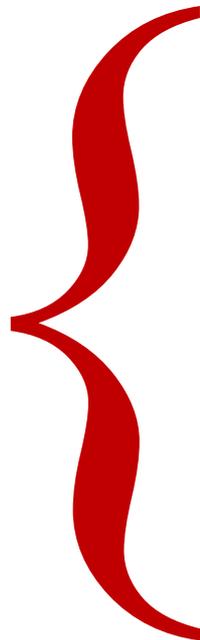
- Alle Definitionen (all defs)
- Alle Definition-Benutzung-Paare (all def-uses)

## Bedingungsbasiert

- Einfache Bedingungsüberdeckung
- Mehrfachbedingungsüberdeckung
- Minimal bestimmende Mehrfachbedingungsüberdeckung



## 2.3 White-Box- Test



---

Idee der White-Box Testentwurfsverfahren

Kontrollflussbasierter Test

---

Datenflussbasierter Test

---

Test der Bedingungen

---

Weitere White-Box Testentwurfsverfahren

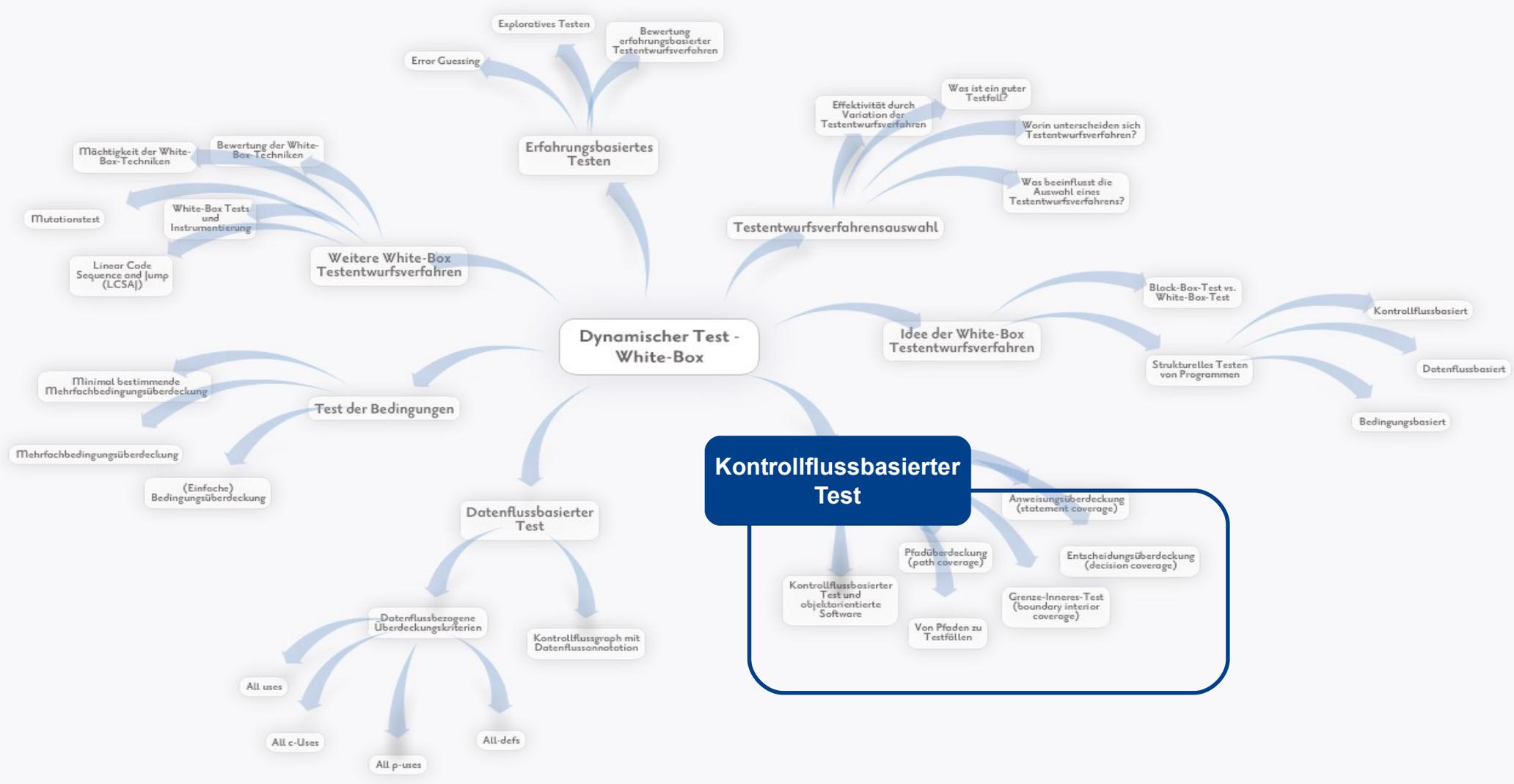
---

Erfahrungsbasiertes Testen

---

Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung

---



- Auswahl eines geeigneten Testverfahrens

		Liegt eine Prozedur, Funktion oder Operation einschließlich Quellcode vor?					
		ja			nein		
		Enthält die Prozedur, Funktion oder Operation ...					Auswahl anhand der Kapitel 5.5 bis 5.13 vornehmen
... 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-Pfadtest</i>	strukturierter Pfadtest	Zweigüberdeckungstest und einfacher Bedingungsüberdeckungstest	minimaler Mehrfach-Bedingungsüberdeckungstest	<i>boundary interior-Pfadtest</i>	strukturierter Pfadtest
				Reicht die Überprüfung aller atomaren Bedingungen aus?			
				ja	nein		
				einfacher Bedingungsüberdeckungstest	minimaler Mehrfach-Bedingungsüberdeckungstest		

- Kontrollflussbezogene Testverfahren
  - sind white-box-Verfahren, denn sie nutzen die Kenntnis der Programmstruktur aus.
  - Sie sind dynamische Verfahren, denn sie basieren auf der Ausführung des Programms mit verschiedenen Testdaten.
- Ziel
  - Testdaten so wählen, dass möglichst viele Durchläufe durch den Kontrollfluss des Programms getestet werden und dass dafür möglichst wenig Testfälle gebraucht werden.
  - Die Varianten der kontrollflussbezogenen Verfahren differenzieren nach angestrebtem Überdeckungsgrad und nach Art und Weise, in der der Begriff der Überdeckung verstanden wird.
- Analysemittel
  - Kontrollflussgraph: Verdeutlicht den Kontrollfluss in einem Programm

- Der Kontrollflussgraph eines Programms  $P$  ist ein gerichteter Graph  $G = (N, E)$  mit 2 ausgezeichneten Knoten  $n_{start}, n_{final}$ 
  - Ein Knoten stellt Anweisungen (oder strikt sequenzielle Anweisungsfolgen dar).
  - Eine Kante aus der Menge der Kanten  $E \subseteq N \times N$  beschreibt den möglichen Kontrollfluss zwischen zwei Anweisungen. Eine Kante aus  $E$  wird auch Zweig genannt.
  - Die beiden ausgezeichneten Knoten stellen die Anfangs- und Endanweisung eines Programms dar.
  - Beispiel

- 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*)
```

...

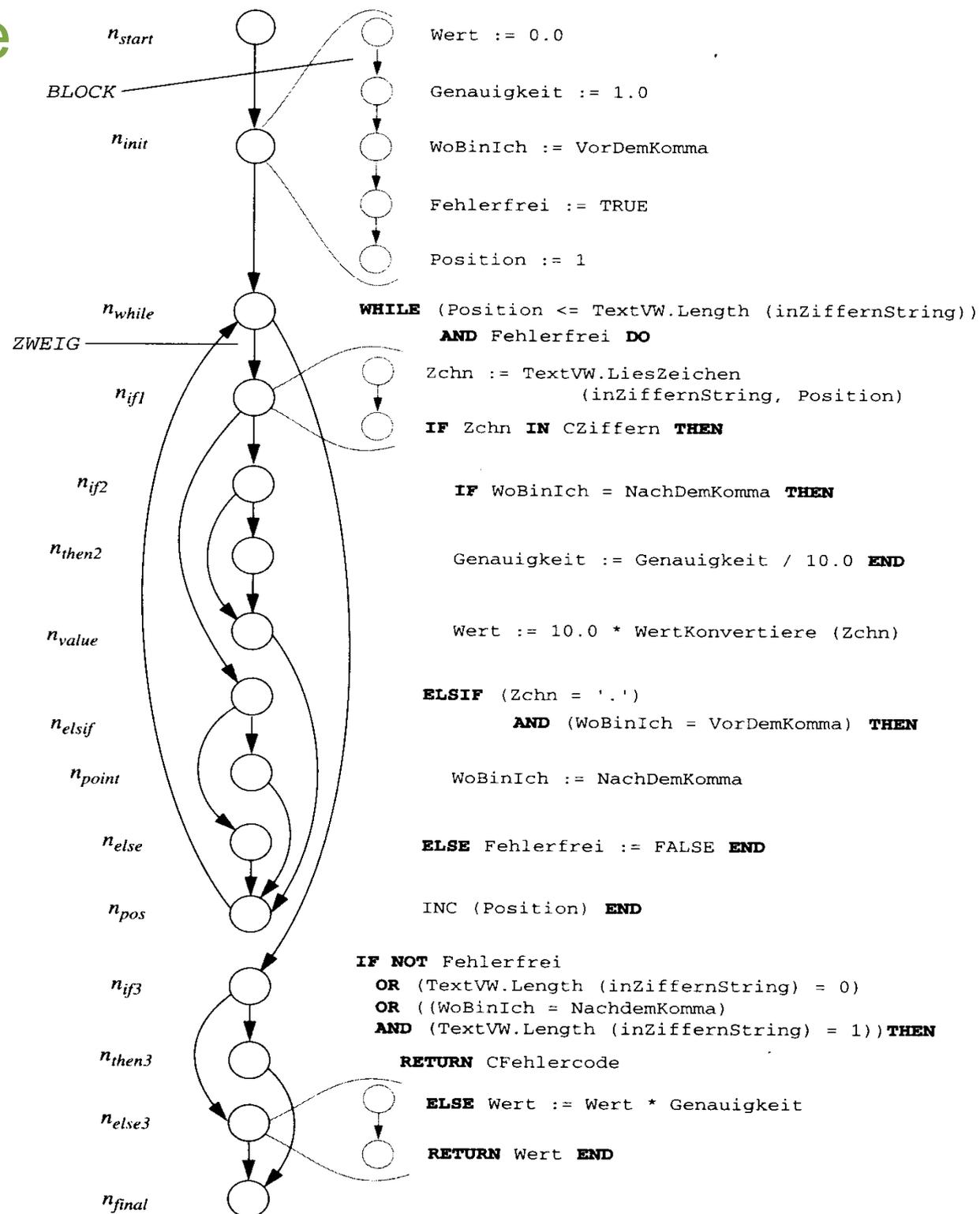
...

```
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;
```

# Kontrollflussbezogene Testverfahren

## Durchspielen an der Ziffernfolge 28.9

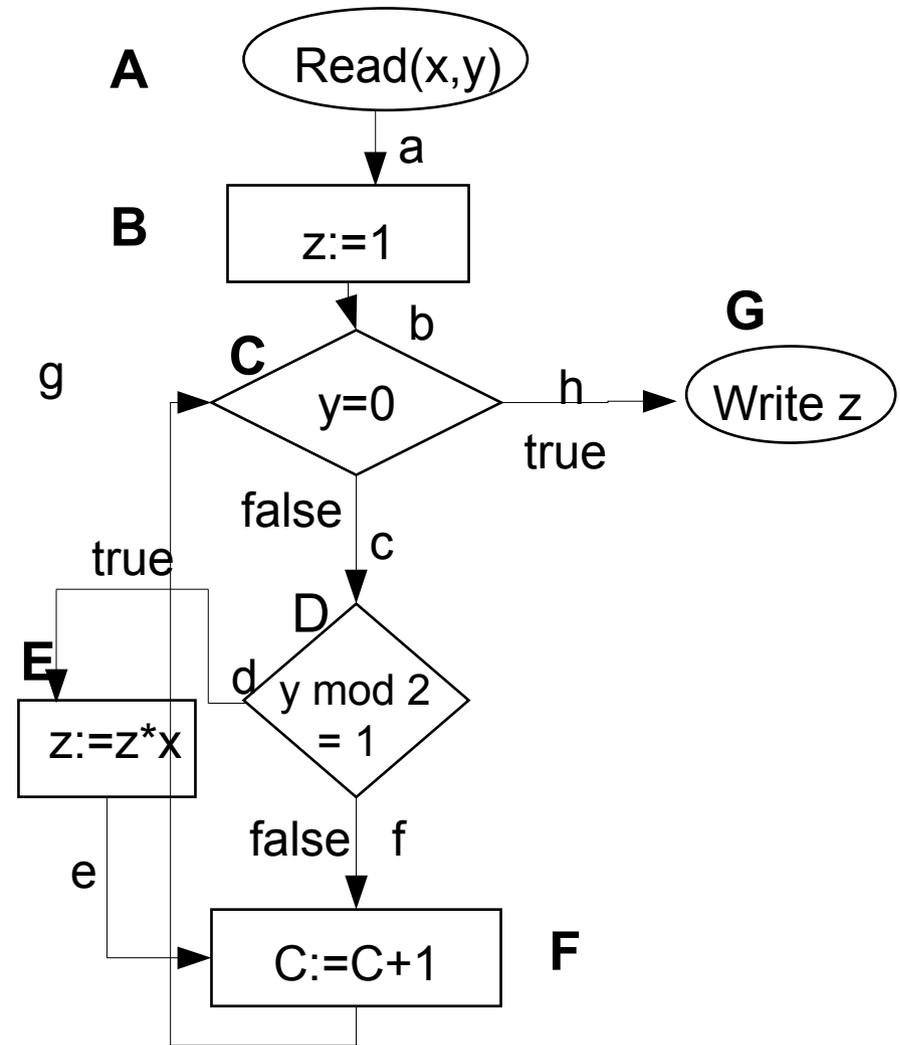
- inZiffernString: 28.9
- Wert:
- Genauigkeit:
- WoBinIch:
- Fehlerfrei:
- Position:
- Zchn:



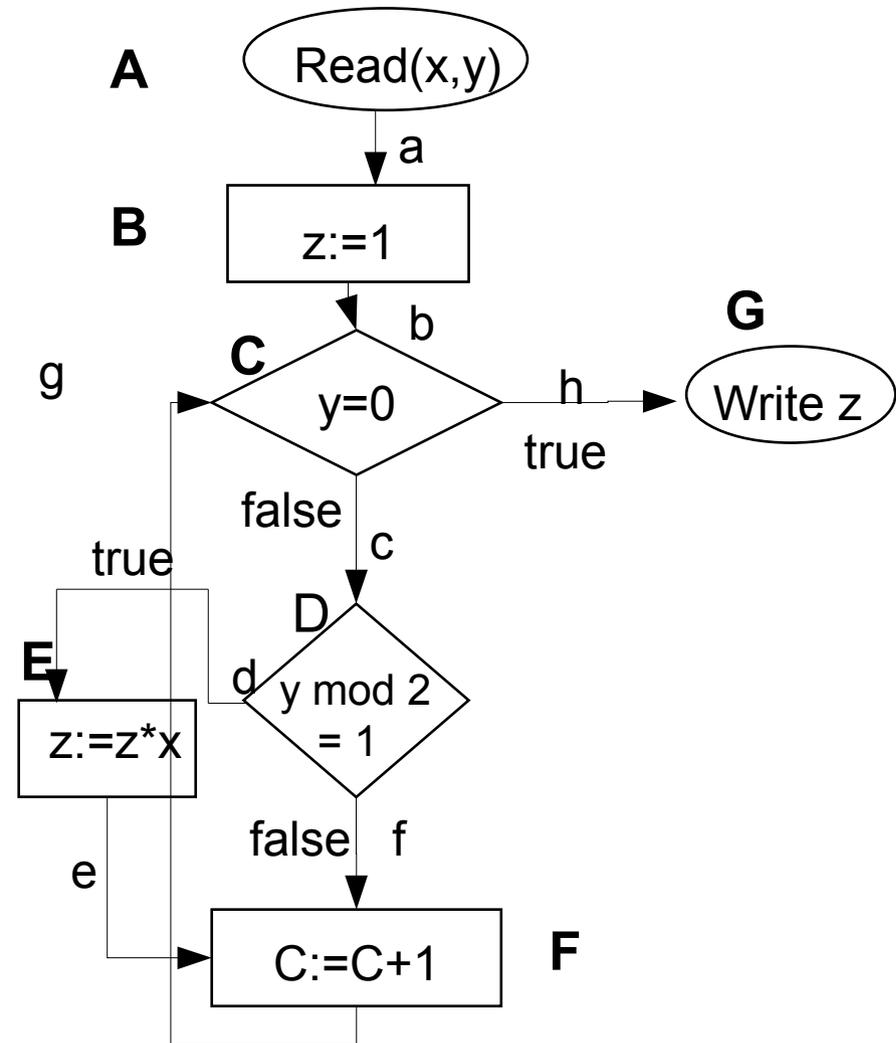
- Der Kontrollflussgraph eines Programms  $P$ 
  - Unter einem Block verstehen wir eine nichtleere Folge von Knoten,
    - die ausschließlich durch den ersten Knoten betreten werden kann,
    - die - sobald der erste Knoten durchlaufen wird - in der vorgegebenen Reihenfolge deterministisch genau einmal durchlaufen wird,
    - die bezüglich der ersten beiden Eigenschaften maximal ist.
  - Pfad/vollständiger Weg
    - Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit dem Endknoten endet
  - Wege  $(T,P)$ 
    - Ist die Menge der vollständigen, endlichen Wege  $w$  des Kontrollflussgraphen (des Programms  $P$ ), für die es ein Testdatum  $t$  aus der Menge der Testdaten  $T$  gibt, das den Weg  $w$  ausführt.
  - Ein Entscheidungsknoten ist ein Knoten mit mindestens 2 Nachfolgerknoten.

- Der Kontrollflussgraph eines Programms  $P$ 
  - Ein Zyklus ist ein Weg im Kontrollflussgraphen mit mindestens zwei Knoten, der an demselben Knoten beginnt und endet.
  - Ein einfacher Zyklus ist ein Zyklus, bei dem alle Knoten (außer Anfangs- und Endknoten) verschieden sind.

- Ein **Entscheidungs-Entscheidungsweg** ist ein Wegstück, welches bei einem **Entscheidungsknoten** oder dem **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}



- Ein **Segment** ist ein Wegstück mit den Eigenschaften
  - Der **erste Knoten** des Wegstücks ist der Anfangsknoten des Kontrollflussgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten (mehrere Inputs).
  - Der **letzte Knoten** des Wegstücks ist der Endknoten des Kontrollflussgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten
  - Alle **andere Knoten** haben nur eine Eingangs- und eine Ausgangskante
- Hier: {A,B,C}; {C,G}; {C,D}; {D,E,F}; {D,F}; {F,C}



- Fehler, die durch kontrollflussbezogene Verfahren aufgedeckt werden können
  - Berechnungsfehler
    - Sind Fehler, bei denen zwar der richtige Kontrollflussweg im Programm ausgeführt wird, aber mindestens **ein berechneter Variablenwert falsch** ist.
  - Bereichsfehler
    - Sind Fehler, bei denen **nicht der richtige Kontrollflussweg** im Programm ausgeführt wird. Dies liegt daran, dass der Eingabebereich (Menge aller Eingabewerte, die einen Weg ausführen) des vorliegenden Kontrollflussweges nicht mit dem Eingabebereich des korrekten Kontrollflussweges im korrekten Programm übereinstimmt.
  - Unterbereichsfehler
    - Sind (spezielle Bereichs-)Fehler, bei denen **ein Kontrollfluss zu wenig oder zu viel** im Programm vorkommt, d.h. es fehlt eine Abfrage oder es gibt eine zusätzliche, falsche Abfrage.

Bestimmung des **größten gemeinsamen Teilers (ggT)** zweier ganzer Zahlen  $m$  und  $n$  (*greatest common divisor, gcd*):

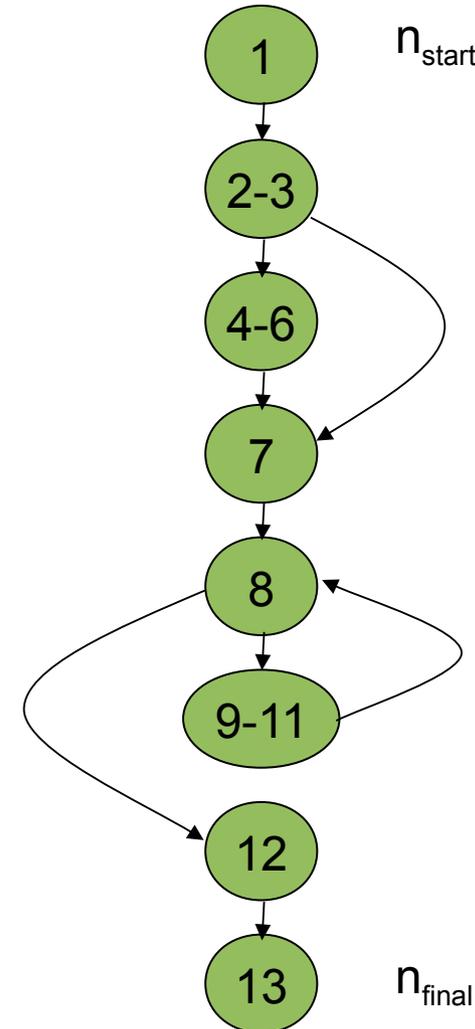
$\text{ggT}(4,8)=4$ ;  $\text{ggT}(5,8)=1$ ;  $\text{ggT}(15,35)=5$ ;  $\text{ggT}(0,0)=0$  [per Konvention]

Zur Erinnerung: Spezifikation in JML / Java:

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

# Zur Erinnerung: Kontrollflussgraph von ggT ()

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

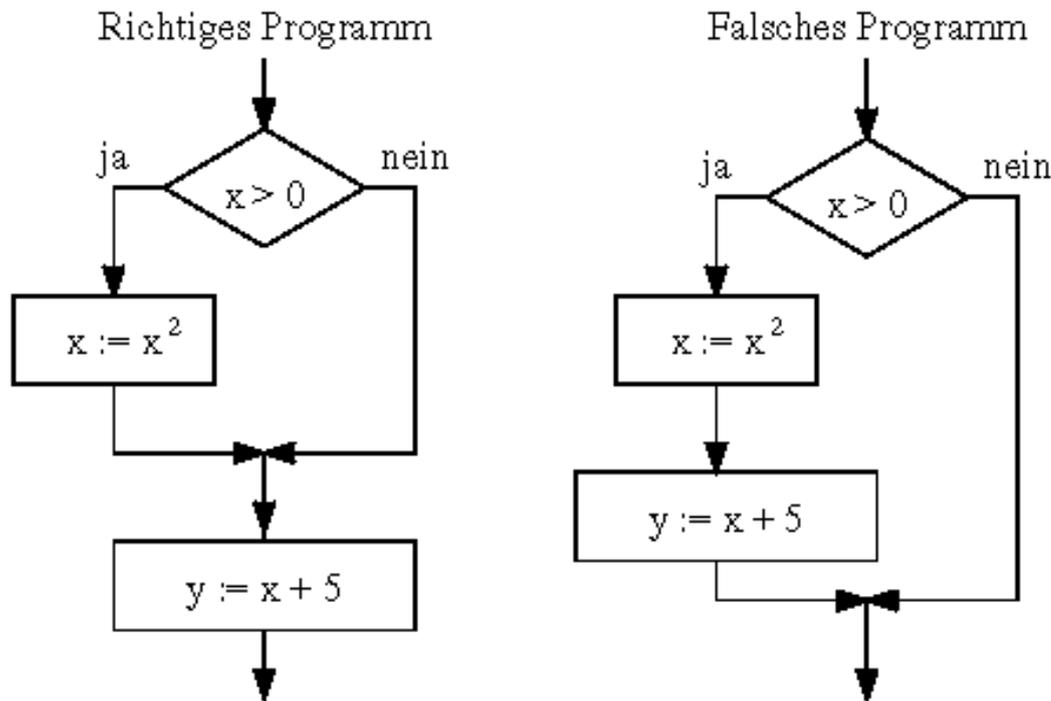


- Anweisungsüberdeckung (*statement coverage; C0-Überdeckung; alle Knoten*)
- Entscheidungs-/Zweigüberdeckung (*decision coverage; C1-Überdeckung, alle Zweige*)
- Grenze-Inneres-Test (*boundary interior coverage*)
- Pfadüberdeckung (*path coverage; C $\infty$ -Überdeckung, alle Pfade*)

- Anweisungsüberdeckung ( $C_0$ -Test) (C für Coverage)
  - Eine Testdatenmenge T erfüllt die  $C_0$ -Überdeckung für Programm P genau dann, wenn es für jede Anweisung A des Programms P ein Testdatum t aus T gibt, das die Anweisung A ausführt.
  - Eine Anweisung A wird unter T ausgeführt/überdeckt, genau dann wenn der zur Anweisung A gehörende Knoten k in einem Weg von Wege(T) vorkommt.
  - Als Testwirksamkeitsmaß  $TWM_0$  wird der Anweisungsüberdeckungsgrad definiert (Verhältnis der besuchten Knoten zur Gesamtzahl von Knoten)

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

- Anweisungsüberdeckung ( $C_0$ -Test): Beispiel



Testdatum

T1: Eingabedatum:  $x=5, y=0$ ,  
Solldatum:  $x=25, y=30$

erfüllt zwar die  $C_0$ -  
Überdeckung, deckt aber nicht  
den Fehler im falschen  
Programm auf

Dynamisches, kontrollflussbasiertes Testentwurfsverfahren, das die mindestens einmalige Ausführung aller Anweisungen des Testobjekts fordert.

$$\text{Anweisungsüberdeckungsgrad} = \frac{\text{Anzahl durchlaufener Anweisungen}}{\text{Gesamtzahl Anweisungen}}$$

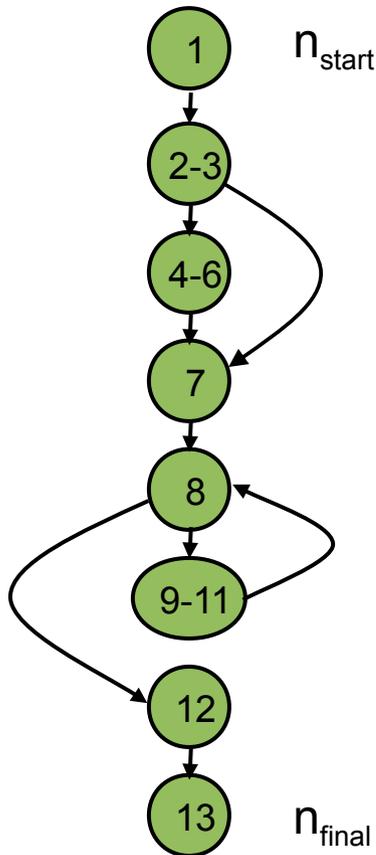
Jeder Testfall wird anhand eines Pfads durch den Kontrollflussgraphen bestimmt.

- Bei dem Testfall müssen die auf dem Pfad liegenden Kanten des Graphen durchlaufen werden, d.h. die Anweisungen (Knoten) in der entsprechenden Reihenfolge zur Ausführung kommen.
- Bei der Berechnung wird nur gezählt, ob eine Anweisung bei der Ausführung überhaupt durchlaufen wurde, die Häufigkeit der Ausführung spielt keine Rolle.
- Für die einzelnen Testfälle sind, neben den Vor- und Nachbedingungen, auch die erwarteten Ergebnisse und das erwartete Verhalten des Testobjektes vorab zu bestimmen und danach mit dem tatsächlichen Ergebnis bzw. Verhalten zu vergleichen.

Wenn der zuvor festgelegte Deckungsgrad erreicht ist, wird der Test als ausreichend angesehen und beendet.

Auch als C0-Maß bezeichnet (**C**overage = **Ü**berdeckung).

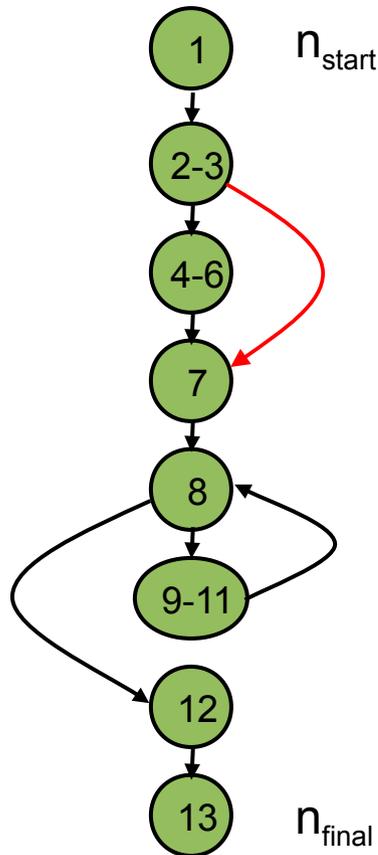
# Beispiel: Anweisungsüberdeckung für `ggt()`



Pfad = ?

```
1. public int ggt(int m, int n) {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0 and
   //...
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;
}
```

# Beispiel: Anweisungsüberdeckung für `ggt()`



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

```
1. public int ggt(int m, int n) {  
    // pre: m > 0 and n > 0  
    // post: return > 0 and  
    // m@pre.mod(return) = 0 and  
    //...  
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;  
}
```

Die Anweisungsüberdeckung ist ein in der Aussage **schwaches** Kriterium.

- Für die Anweisungsüberdeckung ist eine „leere“ Kante, d.h. eine Kante, die lediglich ein oder mehrere Knoten überbrückt, ohne Bedeutung.
- Beispiele: ELSE-Kante (zwischen IF und ENDIF) mit leerem ELSE-Teil; Rücksprung zum Anfang einer Repeat-Schleife; BREAK.
- Möglicherweise fehlende Anweisungen in dem enthaltenden Programmteil werden nicht erkannt !

Eine 100%ige Überdeckung der Anweisungen ist in der Praxis nicht immer erreichbar.

- Z.B. wenn Ausnahmebedingungen im Programm vorkommen, die während der Testphase nur mit erheblichem Aufwand oder gar nicht herzustellen sind.
- Kann aber auch auf nicht erreichbare Anweisungen (»dead code«) hindeuten (ggf. statische Analyse durchführen).

- Anweisungsüberdeckung ( $C_0$  -Test)
  - Bewertung
    - Geringe Zahl von Eingabedaten
    - Notwendiges, aber kein hinreichendes Testkriterium
    - Mangelhafte Aussagekraft: nicht-ausführbare Programmteile werden sicher entdeckt, alle anderen Fehler werden nur zufällig entdeckt

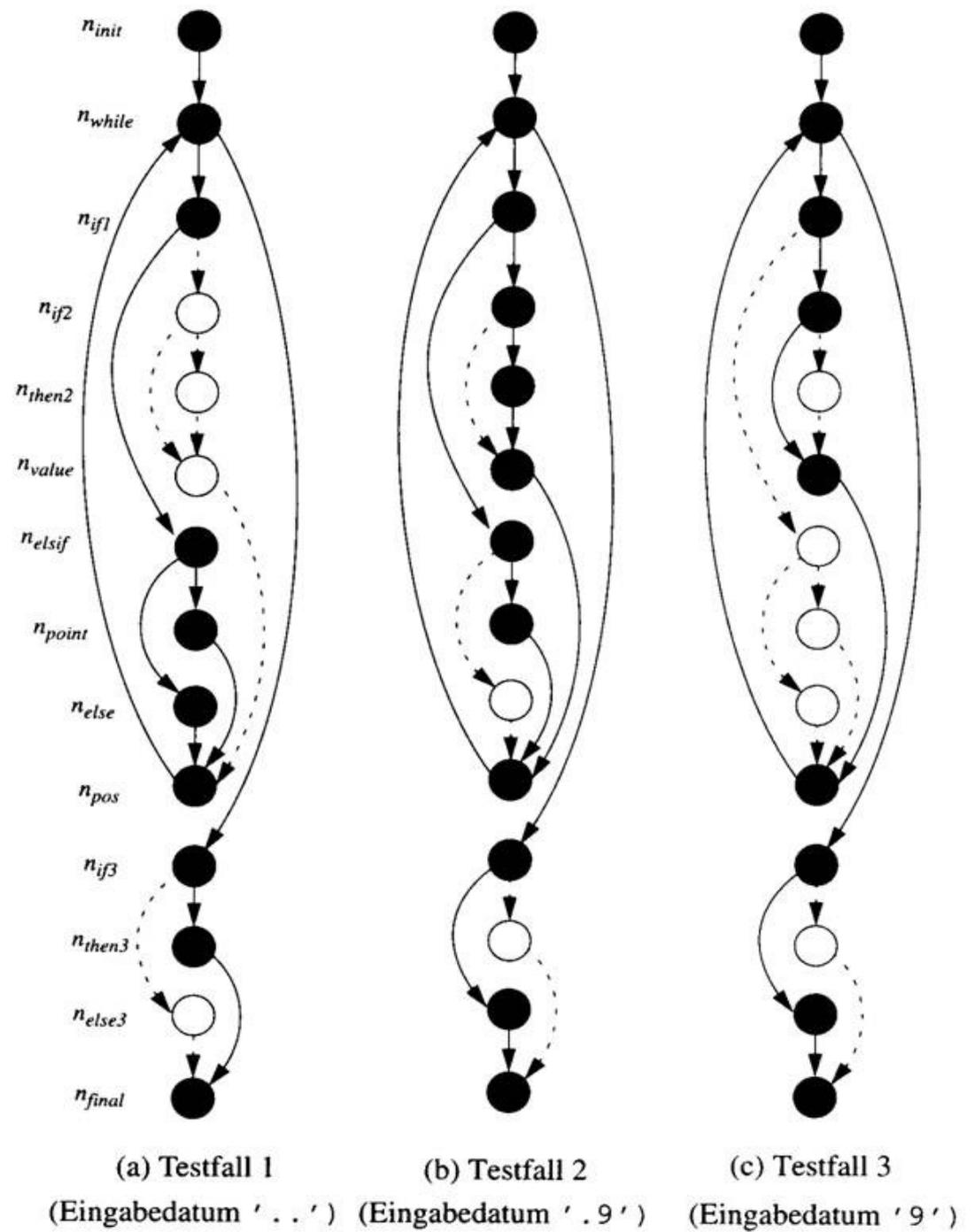
## • Zweigüberdeckung ( $C_1$ -Test)

- strenger als Anweisungsüberdeckung, alle durch Selektion oder Iteration bedingten Verzweigungen im Kontrollfluss sind mindestens einmal zu verfolgen.
- Eine Testdatenmenge  $T$  erfüllt die  $C_1$ -Überdeckung für Programm  $P$ , genau dann wenn es für jede Kante  $k$  im Kontrollflussgraphen von  $P$  einen Weg in  $Wege(T,P)$  gibt, zu dem  $k$  gehört
- Als Testwirksamkeitsmaß  $C_{\text{Zweig}}$  wird der Zweigüberdeckungsgrad definiert (Verhältnis der besuchten Zweige zur Gesamtzahl von Zweigen)

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

- Betrachtung der beiden Testfälle aus der Anweisungsüberdeckung:
  - nur die Kante (nif2, nvalue) wurde nicht durchlaufen
  - der Pfad (nstart, ninit, nwhile, nif1, nif2, nvalue, npos, nwhile, nif3, nelse3, nfinal) enthält diese Kante
  - verwendbares Eingabedatum: ‚9‘

# Kontrollflussbezogene Testverfahren

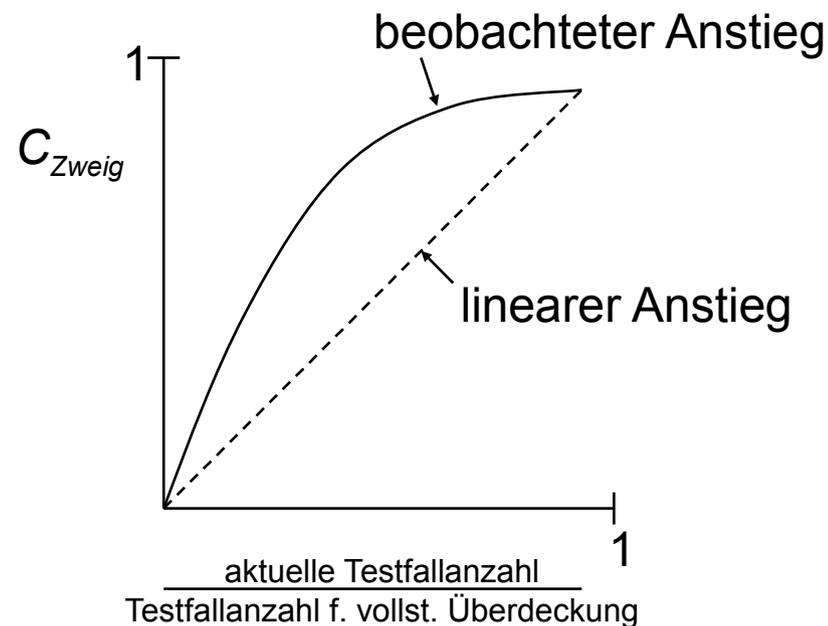


- Zweigüberdeckung ( $C_1$ -Test)
  - Bewertung
    - geringe Zahl von Eingabedaten
    - Aussagekraft besser als  $C_0$ : nicht-ausführbare Knoten und Zweige werden sicher entdeckt, alle anderen Fehler werden nur zufällig entdeckt
    - Oft durchlaufene Programmteile können erkannt und ggf. optimiert werden
    - Unzureichend für den Test von Schleifen
      - Pfadüberdeckungstest
    - Abhängigkeiten zwischen Zweigen nicht berücksichtigt
      - Pfadüberdeckungstest
    - Kein Test komplexer, zusammengesetzter Bedingungen
      - Bedingungsüberdeckungstest
    - empirische Untersuchungen ergeben, dass auch bei 100%iger Abdeckung nur 25% der Fehler erkannt werden, dennoch: besser als völlig unsystematisches Testen!

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

Beobachteter Zusammenhang!

Keine streng mathematische Notwendigkeit



Was bedeutet das?

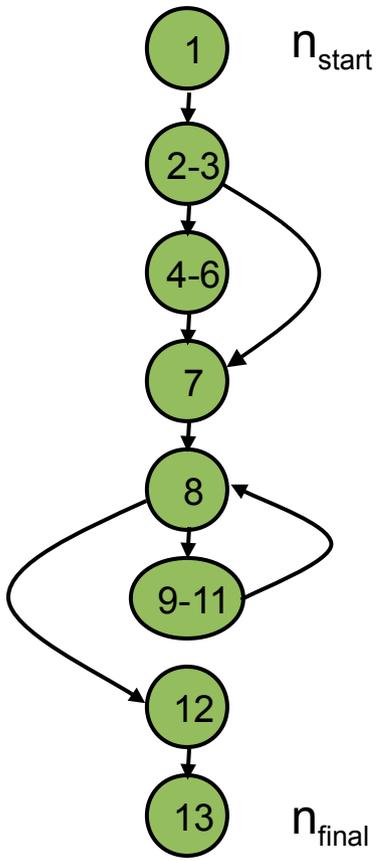
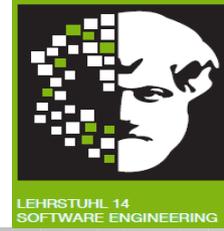
**Entscheidungstest:** Ein White-Box-Testentwurfsverfahren, bei dem Testfälle im Hinblick auf die Überdeckung der Entscheidungsausgänge entworfen werden.

**Entscheidung:** Eine Stelle in einem Programm, an der der Kontrollfluss in zwei oder mehrere alternative Wege verzweigen kann. Ein Knoten mit zwei oder mehreren ausgehenden Kanten.

**Entscheidungsausgang:** Das Ergebnis einer Entscheidung, das den einzuschlagenden Weg im Kontrollfluss bestimmt.

**Entscheidungsüberdeckung:** Der Anteil an Entscheidungsausgängen, die durch eine Testsuite geprüft wurden. 100% Entscheidungsüberdeckung bedeutet sowohl 100% Zweigüberdeckung als auch 100% Anweisungsüberdeckung.

# Beispiel: Entscheidungsüberdeckung für ggt ()

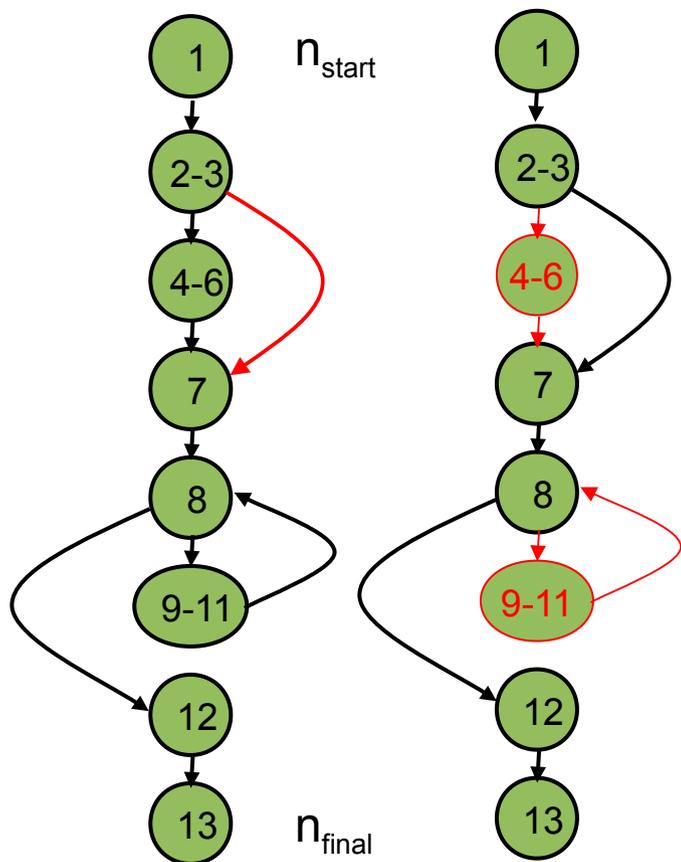


Pfad 1 = ?  
Pfad 2 = ?

```
1. public int ggt (int m, int n)
   {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0 and
   //...
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;
}
```

Entscheidung

# Beispiel: Entscheidungsüberdeckung für ggt ()



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

```
1. public int ggt (int m, int n)
   {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0 and
   //...
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;
}
```

Entscheidung

Jeder Testfall wird anhand eines Pfads durch den Kontrollflussgraphen zur betrachteten Entscheidung bestimmt.

- Jede Entscheidung ist zu allen Fällen auszuwerten (IF / WHILE / FOR-Bedingungen zu wahr und falsch, CASE-Ausdruck zu allen Alternativen).
- Die Häufigkeit der Entscheidungsergebnisse spielt keine Rolle, es zählt nur, ob jedes mögliche Ergebnis mindestens einmal vorlag.
- Für die einzelnen Testfälle sind neben den Vor- und Nachbedingungen wieder die erwarteten Ergebnisse und das erwartete Verhalten des Testobjektes vorab zu bestimmen und mit dem tatsächlichen Ergebnis bzw. Verhalten zu vergleichen.

$$\text{Entscheidungsüberdeckungsgrad} = \frac{\text{Anzahl getestete Entscheidungsergebnisse}}{\text{Gesamtzahl Entscheidungsergebnisse}}$$

# Entscheidungsüberdeckung vs. Zweigüberdeckung



**Entscheidungsüberdeckung** betrachtet in erster Linie die Entscheidungen; diese sind mindestens einmal zu beiden (allen) Möglichkeiten auszuwerten. Jeder Testfall wird also anhand einer Entscheidung ausgewählt.

**Zweigüberdeckung (branch coverage)** zielt darauf ab, alle Kanten (Zweige) im Kontrollflussgraphen abzudecken, jeder Testfall wird anhand eines Pfads durch den Kontrollflussgraphen bestimmt. Durch die Testfälle müssen alle Kanten des Graphen durchlaufen werden (auch die „leeren“ Kanten, die Knoten bzw. Anweisungen „überbrücken“) – jede Entscheidung wird also zu allen Möglichkeiten ausgewertet. Auch als C1-Maß bezeichnet.

$$\text{Zweigüberdeckungsgrad} = \frac{\text{Anzahl durchlaufener Zweige}}{\text{Gesamtzahl Zweige}}$$

Bei der Betrachtung vollständiger Testpfade kommen die beiden Definitionen zum selben Ergebnis.

Die Entscheidungsüberdeckung ist ein in der Aussage stärkeres Kriterium als die Anweisungsüberdeckung.

- Entscheidungsüberdeckung berücksichtigt bei einer Verzweigung des Kontrollflusses beide (bei einer IF-Bedingung) bzw. alle (bei einer CASE-Anweisung) Möglichkeiten und bei Schleifen die Umgehung des Schleifenkörpers und ein Rücksprung zum Schleifenanfang.
- Entscheidungsüberdeckung kann fehlende Anweisungen (z.B. in leeren ELSE-Zweigen) im Gegensatz zur Anweisungsüberdeckung erkennen !

Die Entscheidungsergebnisse werden aber unabhängig voneinander betrachtet und es werden keine bestimmten Kombinationen der einzelnen Entscheidungen gefordert.

Resümee:

- Eine 100%ige Entscheidungsüberdeckung ist anzustreben (ist aber – wie bei der Anweisungsüberdeckung – in der Praxis nicht immer erreichbar).
- Nur wenn neben allen Anweisungen auch jede mögliche Verzweigung des Kontrollflusses in der Testphase berücksichtigt wird, kann der Test als ausreichend eingestuft werden.

- Bedingungsüberdeckungen
  - berücksichtigen Bedingungen in den Schleifen und Auswahlkonstrukten des Bausteins zur Definition von Tests
  - Annahme: es reicht nicht, dass zusammengesetzte Bedingungen (sog. Entscheidungsprädikate) nur einmal zu True und einmal zu False ausgewertet werden, sondern es müssen die Teilbedingungen (atomare Prädikatterme) variiert werden
  - 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 {...}}}

- Bedingungsüberdeckungen
  - Einfache (oder atomare) Bedingungsüberdeckung ( $C_2$ -Test):
    - alle atomaren Prädikate einmal True und einmal False auswerten (Rückverweis)
    - Garantiert keine Zweigüberdeckung
  - Mehrfach-Bedingungsüberdeckung ( $C_3$ -Test):
    - alle Kombinationen der atomaren Prädikate werden betrachtet.
    - Bei einigermaßen komplizierten Prädikaten ist dies nicht mehr handhabbar
  - minimale Mehrfach-Bedingungsüberdeckung (CMinMehrfach):
    - Kompromiss zwischen  $C_2$  und  $C_3$ ,
    - jedes Prädikat (egal ob atomar oder zusammengesetzt) wird zu True und False ausgewertet,
    - weniger als die kombinatorische Abdeckung,
    - mehr als die beidseitige Auswertung der zusammengesetzten Prädikate.

- Einfache Bedingungsüberdeckung ( $C_2$ -Test)
  - Eine Testdatenmenge  $T$  erfüllt die  $C_2$ -Überdeckung genau dann, wenn es für jede Verzweigung im Programm mit zwei Ausgängen und für jeden 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 (z.B. case) können stets in ein äquivalentes Konstrukt mit zwei Ausgängen umgeformt werden
  - Garantiert keine Zweigüberdeckung, da Testen beider Wahrheitswerte der kompletten, d.h. der zusammengesetzten Booleschen Ausdrücke nicht garantiert ist

- Einfache Bedingungsüberdeckung ( $C_2$ -Test)

- Beispiel

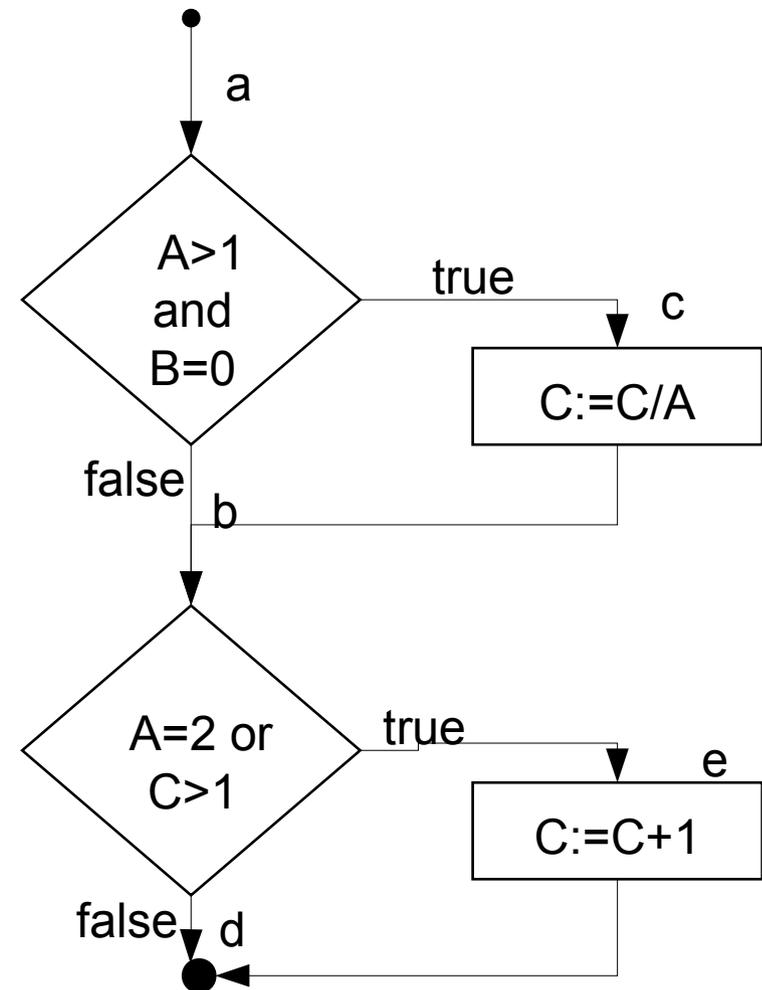
- Testdatum  $A=2, B=1, C=4$

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

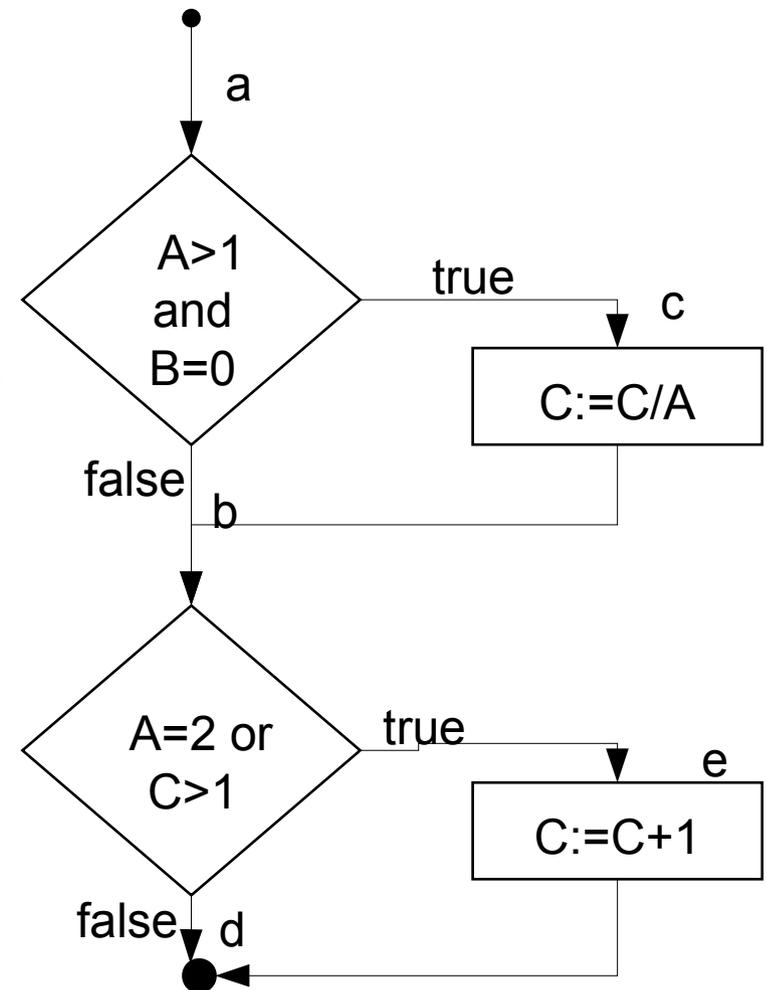
- Testdatum  $A=1, B=0, C=1$

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

➤ Zweig c wird nicht überdeckt!



- Zweig-/Bedingungsüberdeckung
  - Eine Testdatenmenge erfüllt die Zweig-/Bedingungsüberdeckung genau dann, wenn sie die  $C_2$ -Überdeckung (atomare Bedingungsüberdeckung) und die  $C_1$ -Überdeckung (Zweigüberdeckung) erfüllt



- Zweig-/Bedingungsüberdeckung

- Beispiel

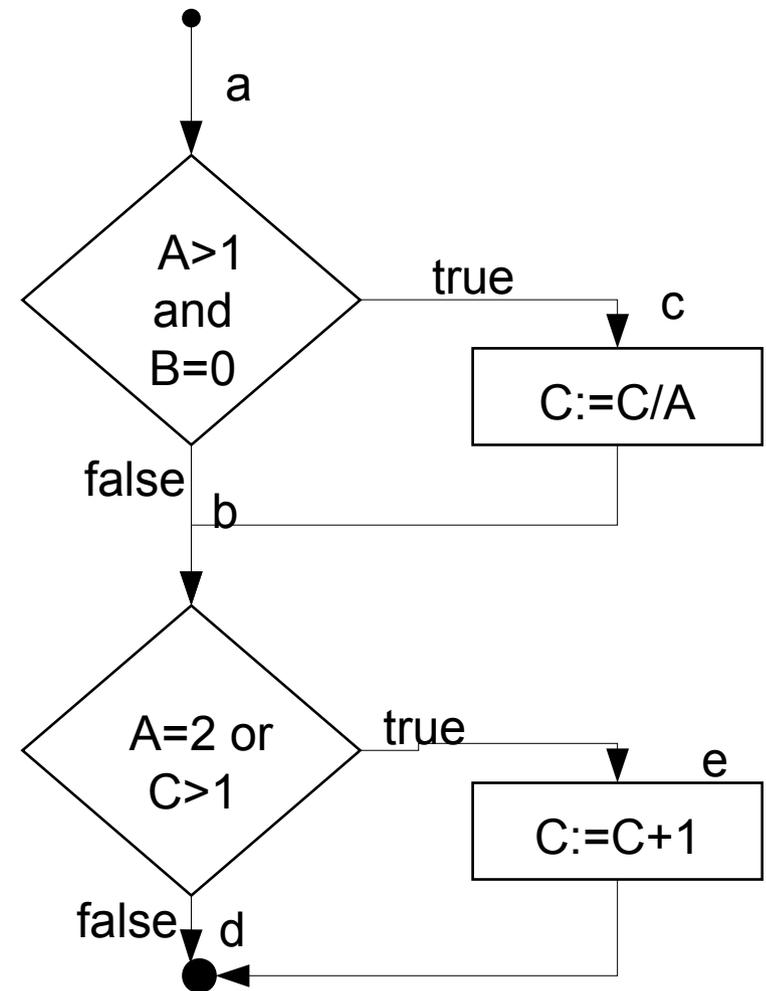
- Testdatum A=2,B=0,C=4

- 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

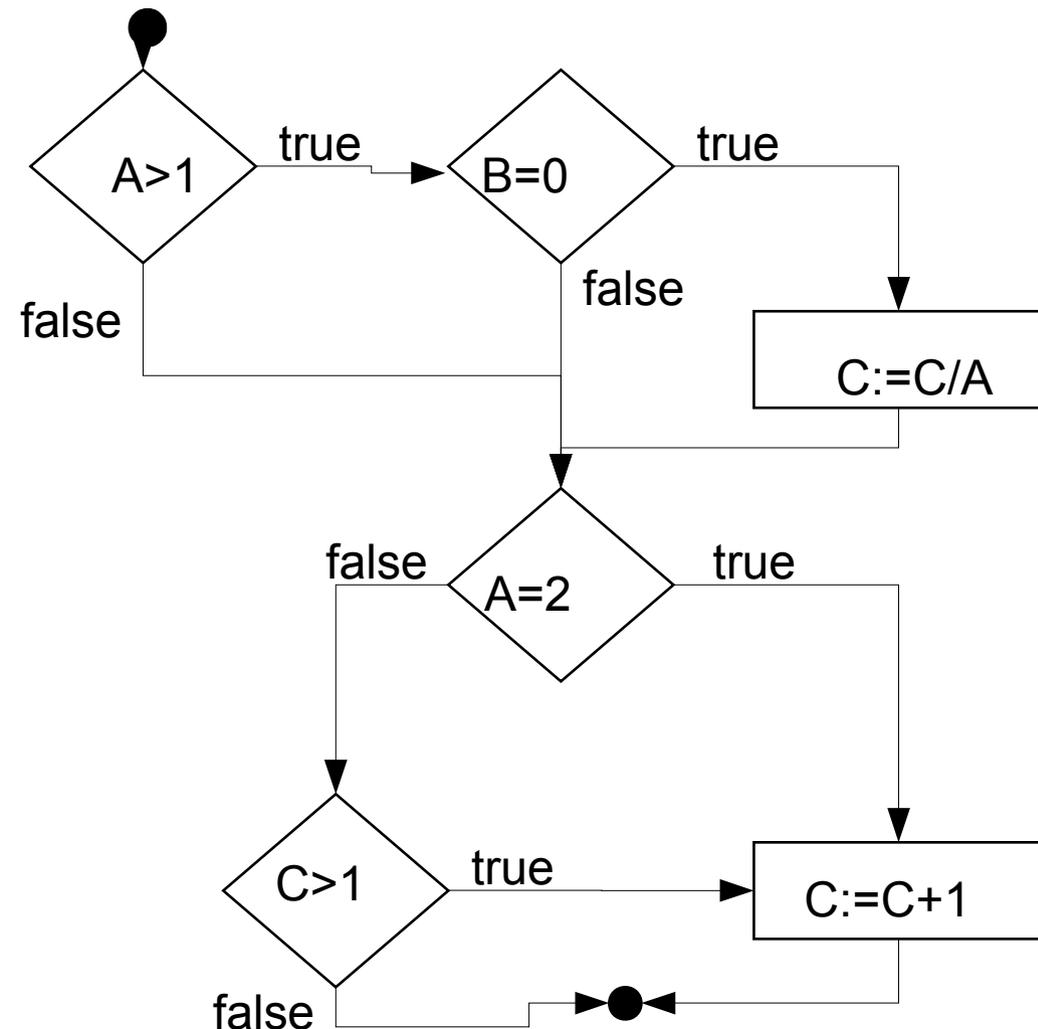
➤ Alle Zweige nun überdeckt, aber nicht alle Zweigkombinationen



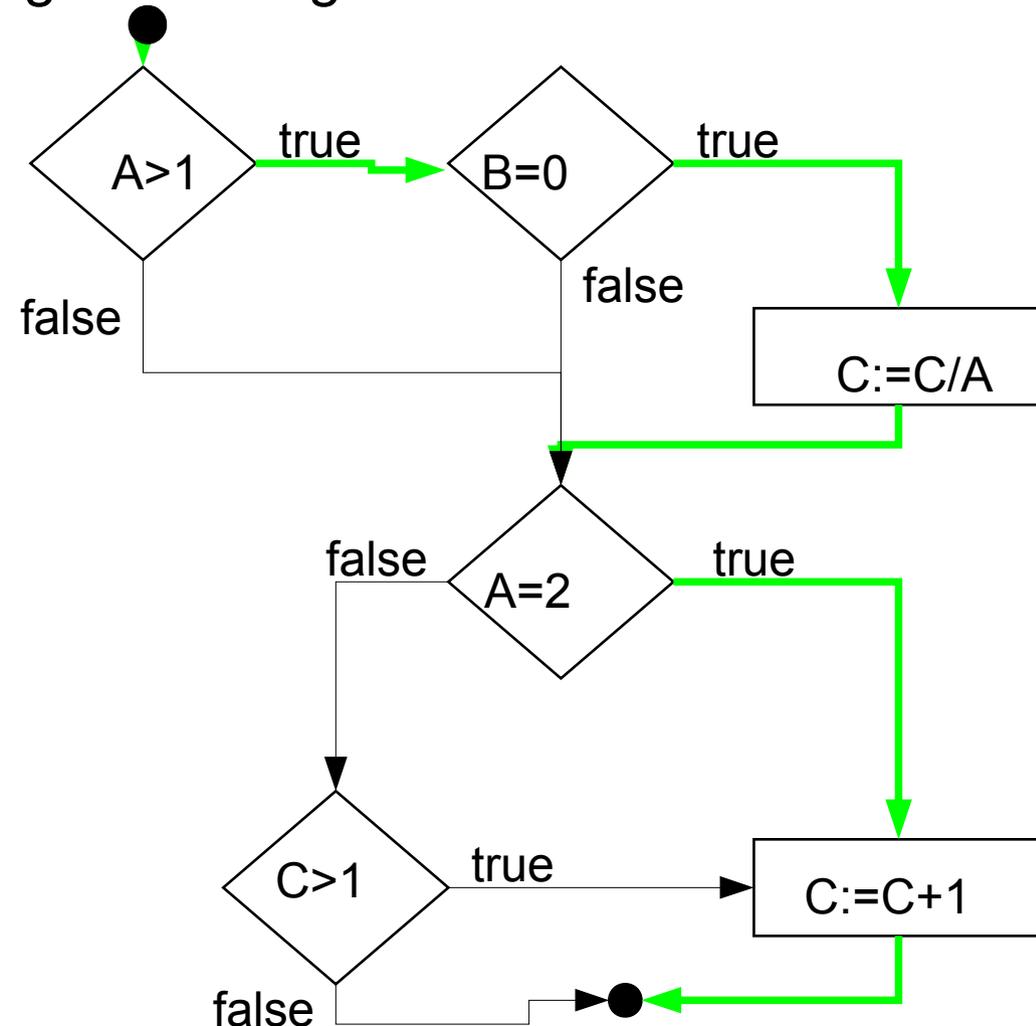
Zweig-/Bedingungsüberdeckung:  
Beispiel für fehlende Überdeckung  
von Zweigkombinationen.

Um dies zu zeigen: Äquivalente  
Umformung des  
Kontrollflussgraphen.

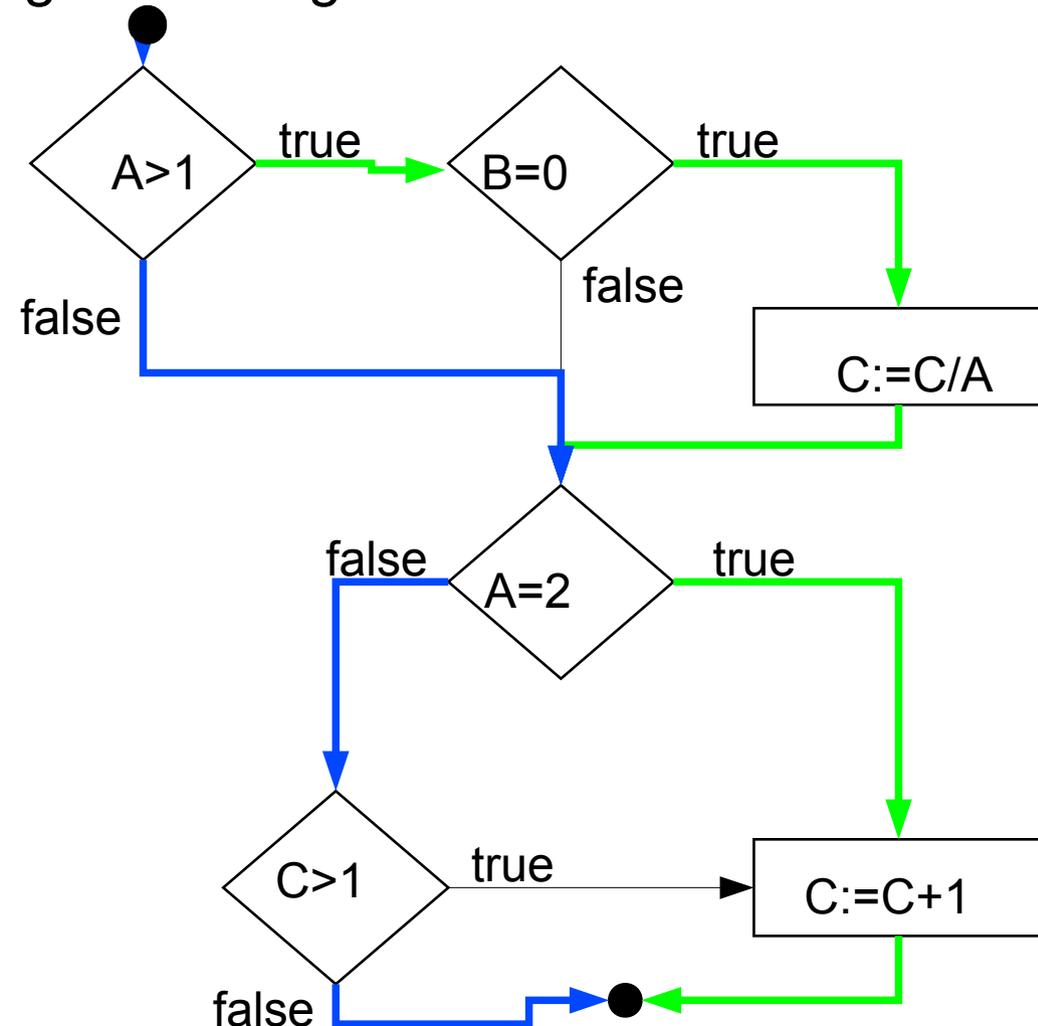
NB: Hier nur aus didaktischen  
Gründen, im Allgemeinen wird  
der Kontrollflussgraph nicht  
umgeformt (da dies auch das  
Überdeckungsergebnis  
verfälschen würde) !



- Zweig-/Bedingungsüberdeckung
  - Beispiel für fehlende Überdeckung von Zweigkombinationen
    - Testdatum  $A=2, B=0, C=4$



- Zweig-/Bedingungsüberdeckung
  - Beispiel für fehlende Überdeckung von Zweigkombinationen
  - Testdatum  $A=2, B=0, C=4$
  - Testdatum  $A=1, B=1, C=1$

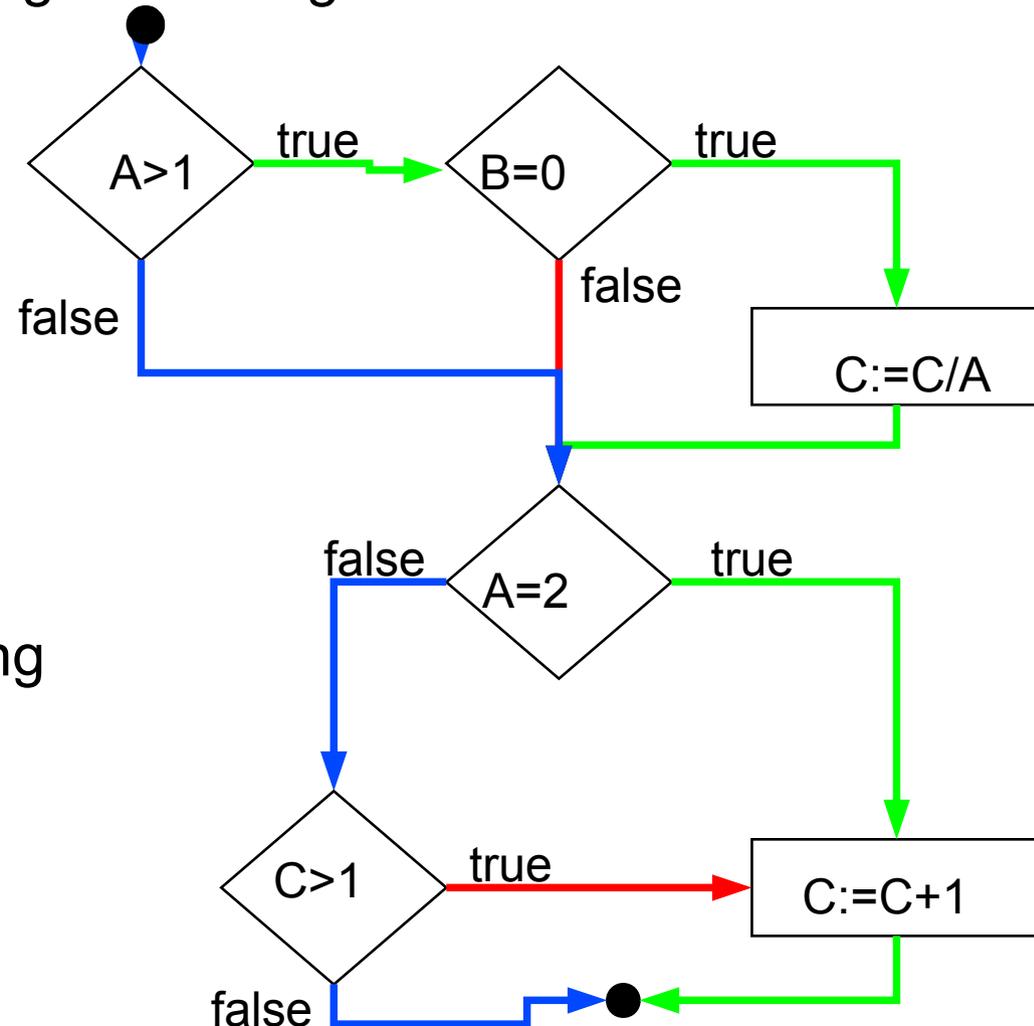


- Zweig-/Bedingungsüberdeckung
  - Beispiel für fehlende Überdeckung von Zweigkombinationen

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

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

- 2 Zweige fehlen!



- Lösung:  
Mehrfachbedingungsüberdeckung

- Mehrfachbedingungsüberdeckung ( $C_3$ -Überdeckung)
  - Eine Testdatenmenge erfüllt die Mehrfachbedingungsüberdeckung genau dann wenn
  - für jede Verzweigung im Programm mit zwei Ausgängen und für den zur Verzweigung gehörenden Booleschen Ausdruck **jede mögliche Kombination der Wahrheitswerte** der atomaren Prädikatterme ausgeführt wird

- Mehrfachbedingungsüberdeckung ( $C_3$ -Überdeckung)

X ( $A > 1$ )	Y ( $B = 0$ )	X and Y
false	false	false
false	true	false
true	false	false
true	true	true

Jeweils  $2^n$  Variationen

X ( $A = 2$ )	Y ( $C > 1$ )	X or Y
false	false	false
false	true	true
true	false	true
true	true	true

- Mehrfachbedingungsüberdeckung ( $C_3$ -Überdeckung)
  - Enthält Zweigüberdeckung
  - Aufwändig zu realisieren ( $2^n$  Variationsmöglichkeiten pro n atomaren Bedingungen)
  - Nicht immer sind alle Kombinationen durch Testdaten realisierbar
  - Dies muss nicht auf einen Fehler der Programmlogik hinweisen!
  - Testbeurteilung wird erschwert!
  - Besser: minimale Mehrfachbedingungsüberdeckung

## Minimale Mehrfachbedingungsüberdeckung ( $C_2(mM)$ -Überdeckung)

- Eine Testdatenmenge erfüllt die minimale Mehrfach-bedingungsüberdeckung genau dann wenn
- für jede Verzweigung im Programm mit zwei Ausgängen und für den zur Verzweigung gehörenden Booleschen Ausdruck folgende Kombinationen der Wahrheitswerte der atomaren Prädikatterme, die mit ein- und mehrstelligen logischen Operatoren verknüpft sind, ausgeführt werden:
- Jede mögliche Kombination von Wahrheitswerten, bei denen die **Änderung** des Wahrheitswertes eines Terms den Wahrheitswert der logischen Verknüpfung **ändern** kann.
- Die Anzahl der Testfälle ist **im Normalfall** geringer als bei der Mehrfachbedingungsüberdeckung.
- Die Gesamtzahl der zu betrachtenden **Kombinationen** für eine  $C_2(mM)$ -Überdeckung beträgt **bei reinen AND bzw. OR-Bedingungen** mit n atomaren Prädikaten  $n+1$ .

- 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 die **erste** zu testen, da bei dieser eine 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 die **letzte** zu testen, da bei dieser eine 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

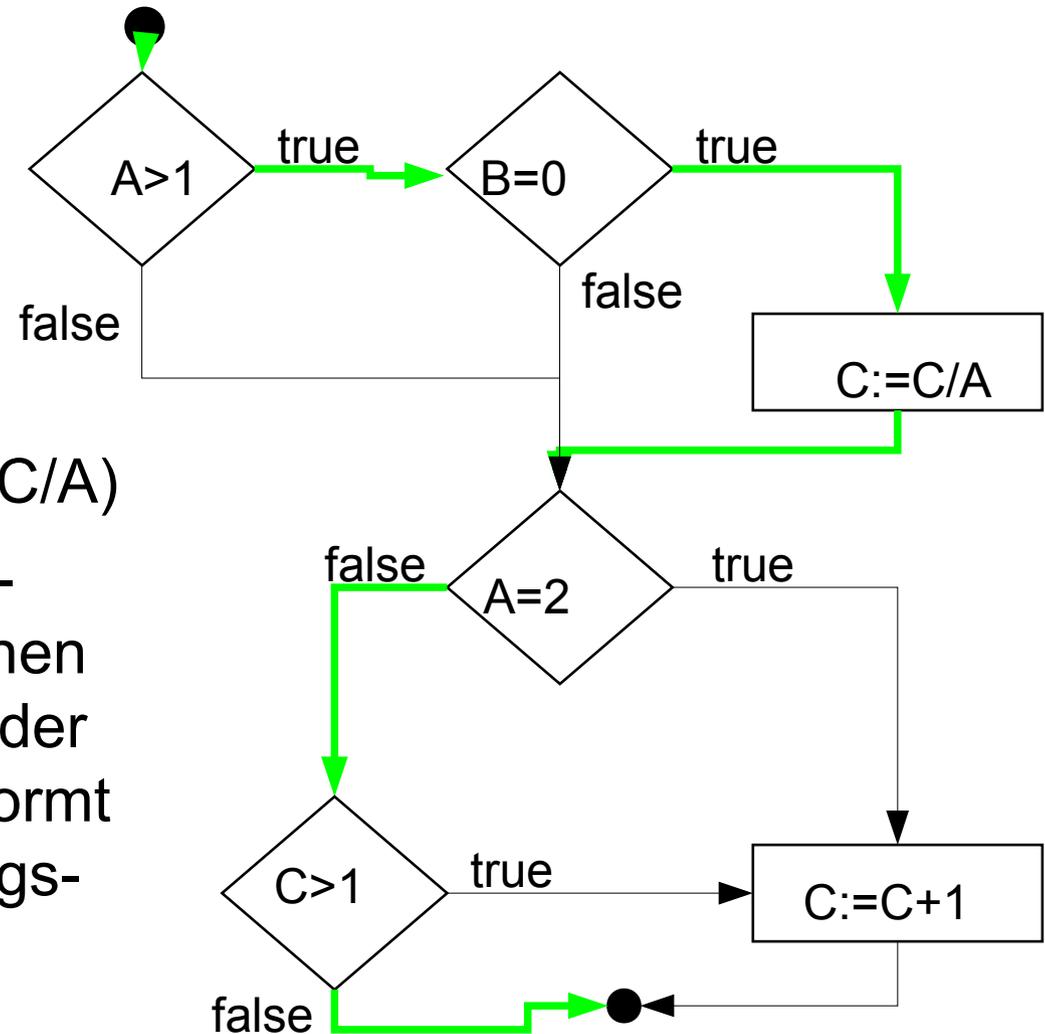
X (A=2)	Y (C>1)	X or Y
false	false	false
false	true	true
true	false	true
true	true	true

Minimale Mehrfachbedingungsüberdeckung Beispiel:

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

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

NB: Umformung des Kontrollflussgraphen hier nur aus didaktischen Gründen, im Allgemeinen wird der Kontrollflussgraph nicht umgeformt (da dies auch das Überdeckungsergebnis verfälschen würde) !



- Minimale Mehrfachbedingungsüberdeckung

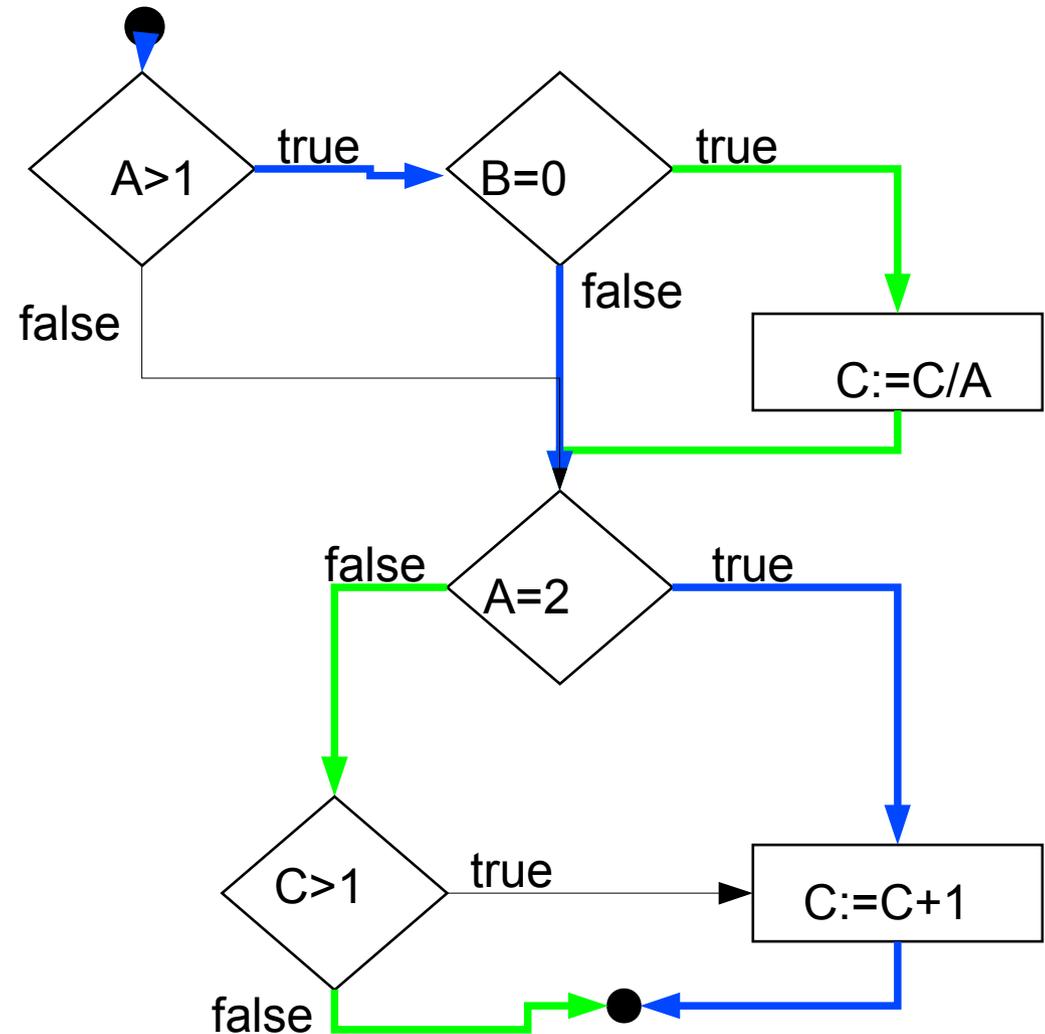
- Beispiel

- 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



- Minimale Mehrfachbedingungsüberdeckung

- Beispiel

- Testdatum A=3,B=0,C=3

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

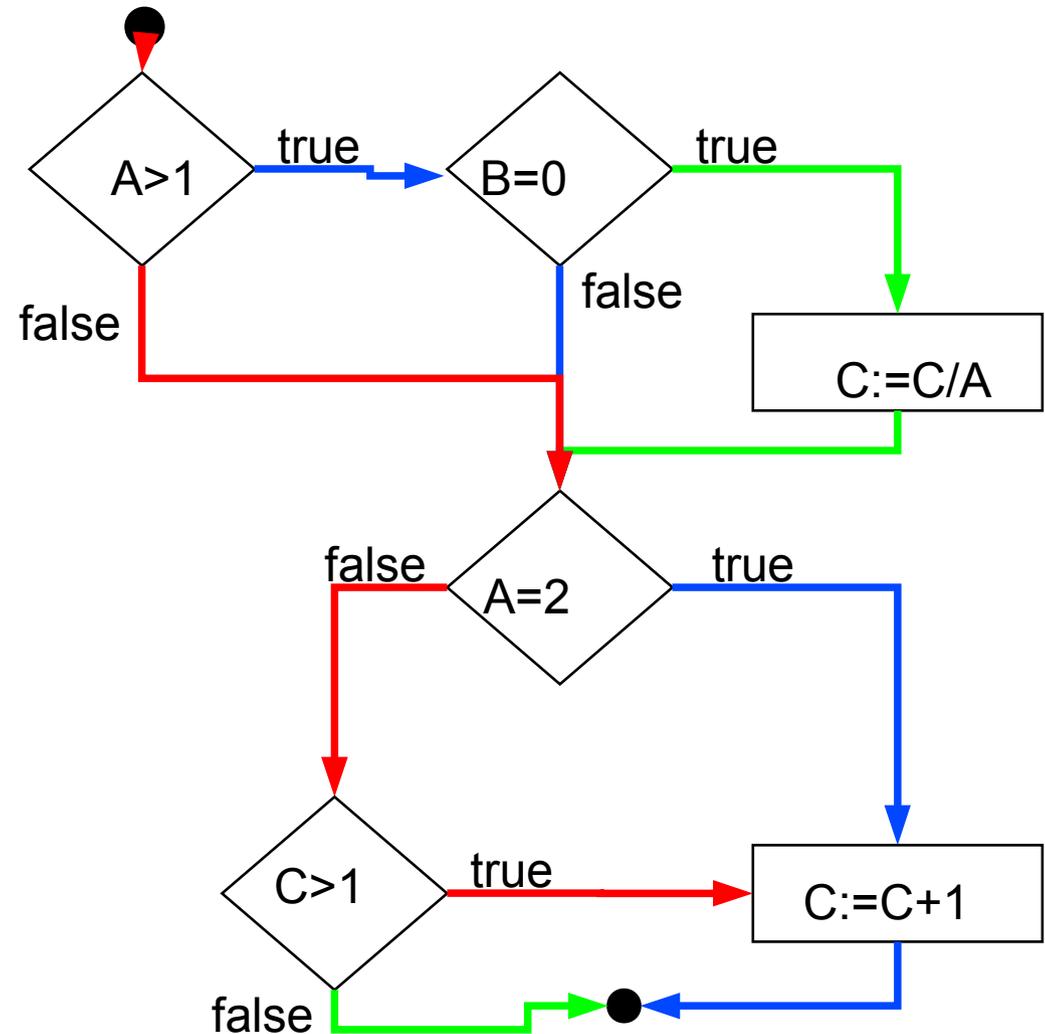
- Testdatum A=2,B=1,C=0

- 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



## Einfache Bedingungsüberdeckung: Testwirksamkeitsmaß $C_2$

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

# Kontrollflussbezogene Testverfahren

**Einfache**  
Bedingungs-  
überdeckung  
2. Beispiel:  
Prozedur  
WerteZiffernfolgeAus

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			
IF3/AND-Prädikat	F	F	F	F	F	T
IF3-Bedingung	F	T				

- Vorgehen
  - Erster Testfall:
  - Menge aller Eingaben, die das Prädikat `Position <= TextVWLenght(inZiffernstring)` zu `TRUE` auswerten
    - > Eingabedatum '9'
  - Protokollierung der Werte der Prädikate für diesen Testfall  
=> erstes und drittes Prädikat wird vollständig abgehandelt

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

(5 Prädikate mit T in erster Spalte der Tabelle, 8 mit F; 14 Prädikate insgesamt)

- Vorgehen
  - Zweiter Testfall:
  - Menge aller Eingaben, die Fehlerfrei zu FALSE auswertet
    - > Eingabedatum 'z'
  - Protokollierung der Werte der Prädikate für diesen Testfall

Überdeckung für **beide** Testfälle somit:

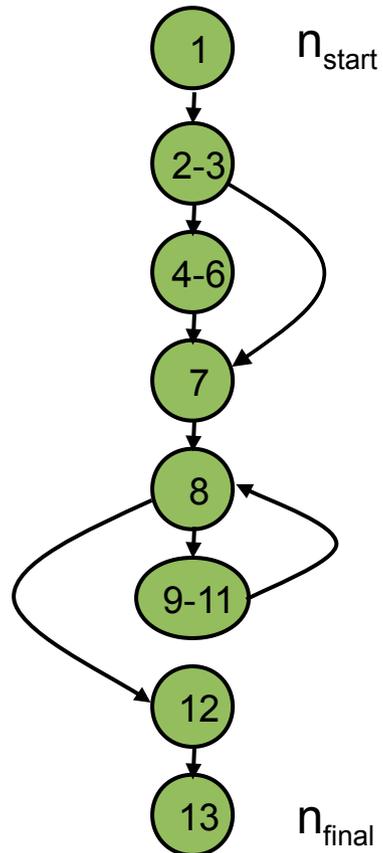
$$C_2 = \frac{8+12}{2 * 14} = \frac{20}{28} = 71\%$$

Grenze-Inneres-Überdeckung: Dynamisches, kontrollflussbasiertes Testentwurfsverfahren (gi) fordert, dass jede Schleife in mindestens einem Testfall:

- gar nicht (nur bei abweisenden Schleifen, **while**, **for** nur wenn Abbruchbedingung bei erstmaliger Auswertung wahr),
- genau einmal und
- mehr als einmal ausgeführt wird.

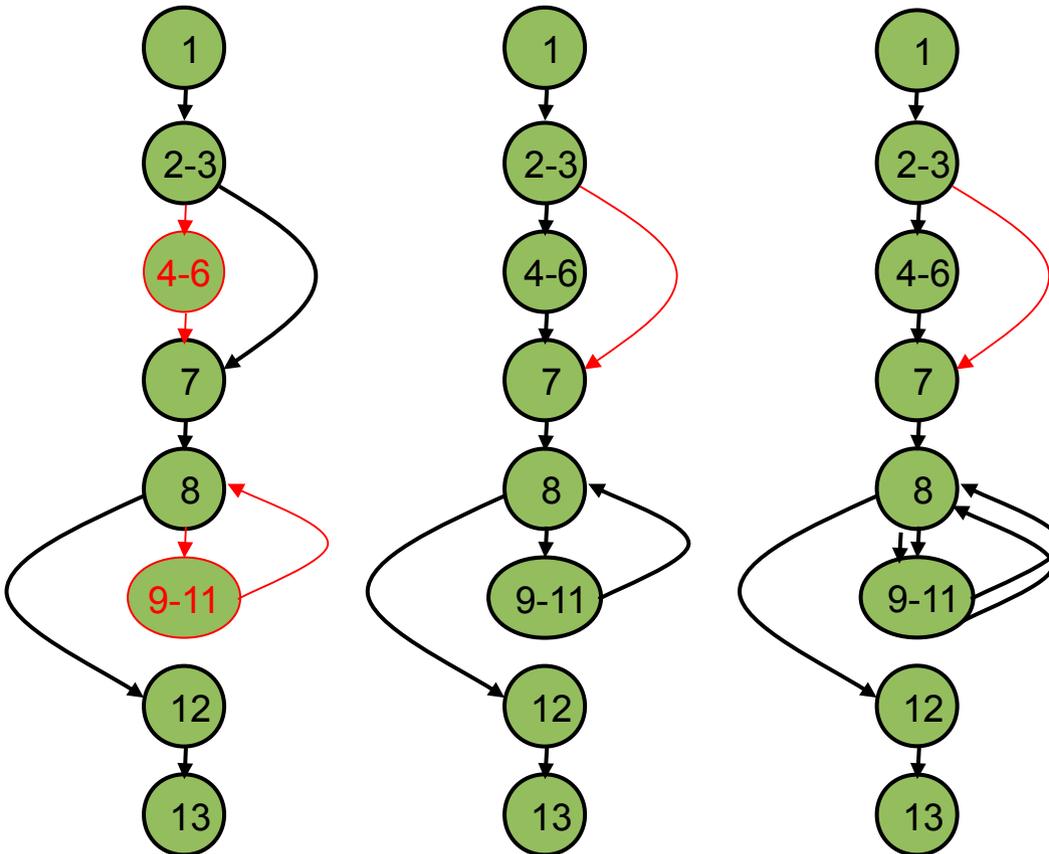
$$\text{Grenze-Inneres-Überdeckungsgrad} = \frac{\text{Anzahl (gi)getestete Schleifen}}{\text{Gesamtzahl Schleifen}}$$

Testfälle werden wieder anhand der entsprechenden Pfade durch den Kontrollflussgraphen bestimmt.



Pfad 1 = ?  
Pfad 2 = ?  
Pfad 3 = ?

```
1. public int ggt (int m,
   int n) {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0
   and
   //...
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. }
```



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)

```
1. public int ggt (int m,
   int n) {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0
   and
   //...
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. }
```

- Die Grenze-Inneres-Überdeckung ist ein in der Aussage auf die 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 !
- Daher in der Praxis höchstens als ergänzendes Kriterium verwenden !
- Die einzelnen Schleifen werden unabhängig voneinander betrachtet.
- Für verschachtelte Schleifen gibt es spezialisierte, in ihrer Stärke abgestufte Überdeckungsmaße.

Dynamisches, kontrollflussbasiertes Testentwurfsverfahren, fordert, dass jeder Pfad im Kontrollflussgraphen mindestens einmal ausgeführt wird. Auch als  $C^\infty$ -Maß bezeichnet.

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

Bei zyklischen Kontrollflussgraphen potenziell unendlich viele Pfade.

- Aber: Obere Grenzen für die Anzahl ggf. aus Spezifikation oder aus technischen Einschränkungen.

In der Praxis nicht erreichbar, eher als theoretisches Vergleichsmaß wichtig.

Frage:

- 1) Können Sie sich ein Programm in Pseudo-Code vorstellen, zu dem keine obere Grenze angegeben werden kann ?
- 2) Lässt sich dieses Programm auf einem heutzutage üblichen realen Computer implementieren ?

Frage:

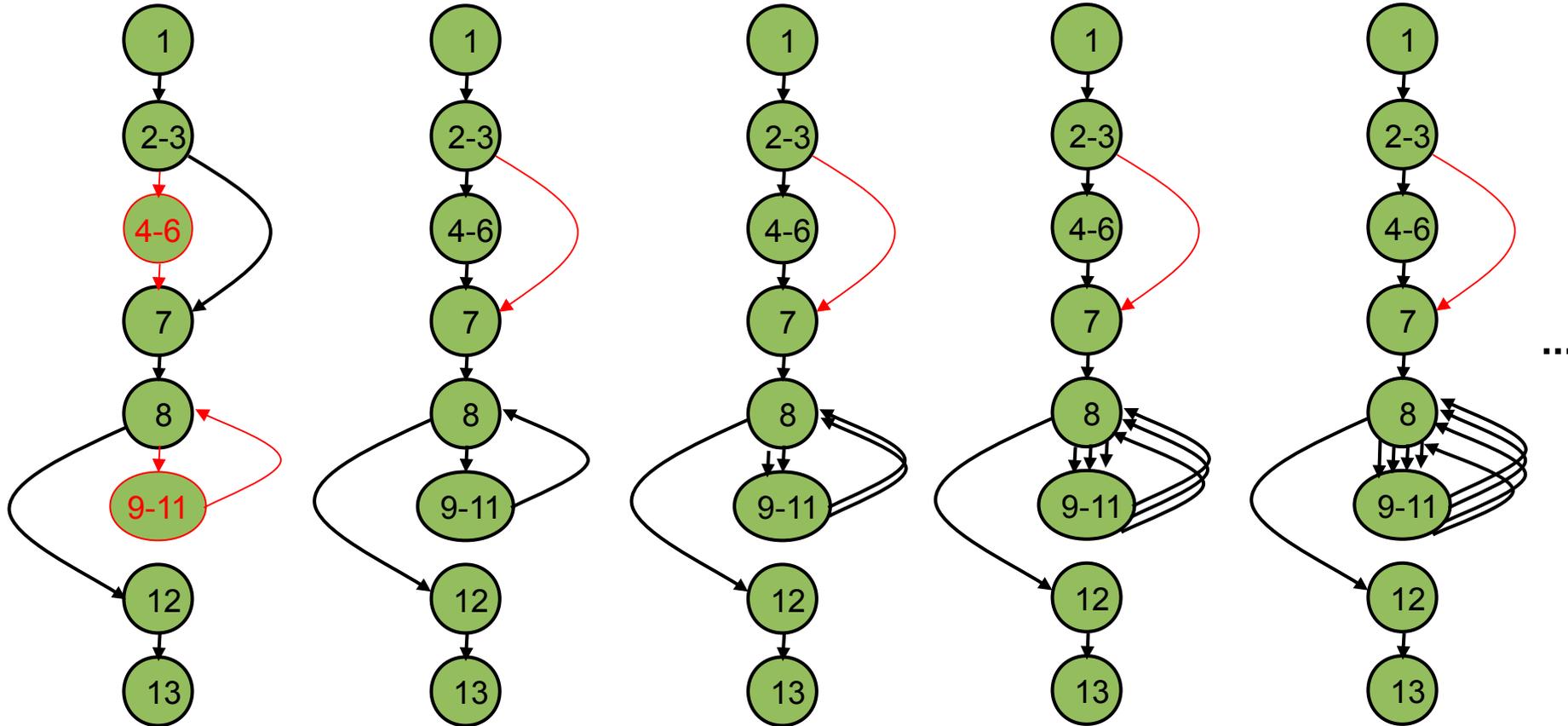
- 1) Können Sie sich ein Programm in Pseudo-Code vorstellen, zu dem keine obere Grenze angegeben werden kann ?
- 2) Lässt sich dieses Programm auf einem heutzutage üblichen realen Computer implementieren ?

Antwort:

- 1) Eine Schleife, in der ein boolescher Zufallswert generiert wird, die den Wert der Abbruchbedingung ergibt. Prinzipiell kann beliebig lange der Wert false generiert werden, bevor ein true generiert wird.
- 2) Heute übliche Computer können nur endlich viele verschiedene Speicherzustände einnehmen. Daher kann der Pseudo-Zufallswert-Generator auch nur eine endliche Menge von Zufallswertfolgen erzeugen. Daraus lässt sich eine endliche obere Grenze ableiten (die allerdings unpraktikabel groß sein kann). Bei Verwendung eines Nicht-Standard-Zufallswert-Generators (z.B. ein radioaktives Zerfallselement, oder Benutzereingaben) gilt diese Einschränkung allerdings nicht.

=> Eine Grenze lässt sich dann nicht angeben, wenn die Schleifenabbruchbedingung von einer externen Quelle von (nicht eingrenzbarem) Nicht-Determinismus abhängt. (z.B. der Benutzer oder spezielle Hardwareelemente).

# Beispiel: Pfadüberdeckung für $\text{ggT}()$



Frage: Welche obere Grenze lässt sich hier angeben ?

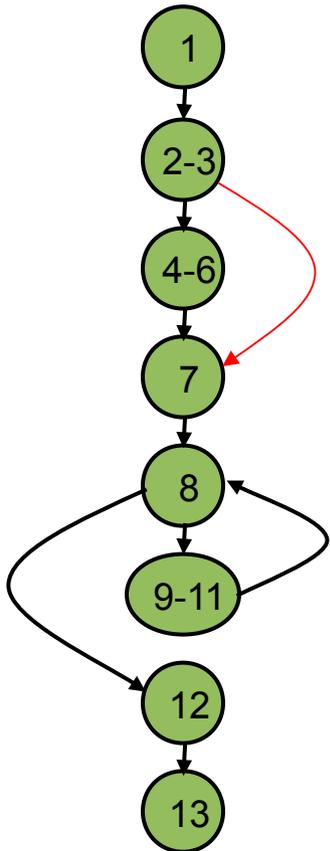
- Bisher wurden lediglich Pfade durch den Kontrollflussgraphen bestimmt, die durch die Testfälle »zur Ausführung gebracht werden sollen«.
- Frage: Mit welchen Eingaben werden diese Pfade erzwungen ?
- Idee: Die Bedingungen der kontrollflussbestimmenden Anweisungen betrachten.
- Damit Aussagen über Programmvariablen »berechnen«.

# Testfall zur Anweisungsüberdeckung

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 Testfall:  $\{ m = 4, n = 6; 2 \}$



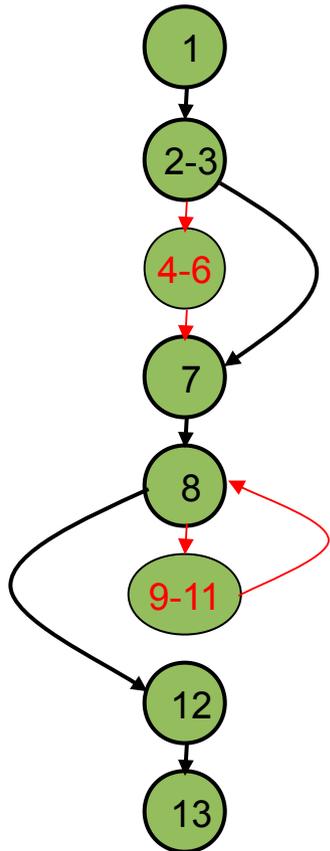
m	n	r
4	6	2

# Testfall zur Entscheidungsüberdeckung

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

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

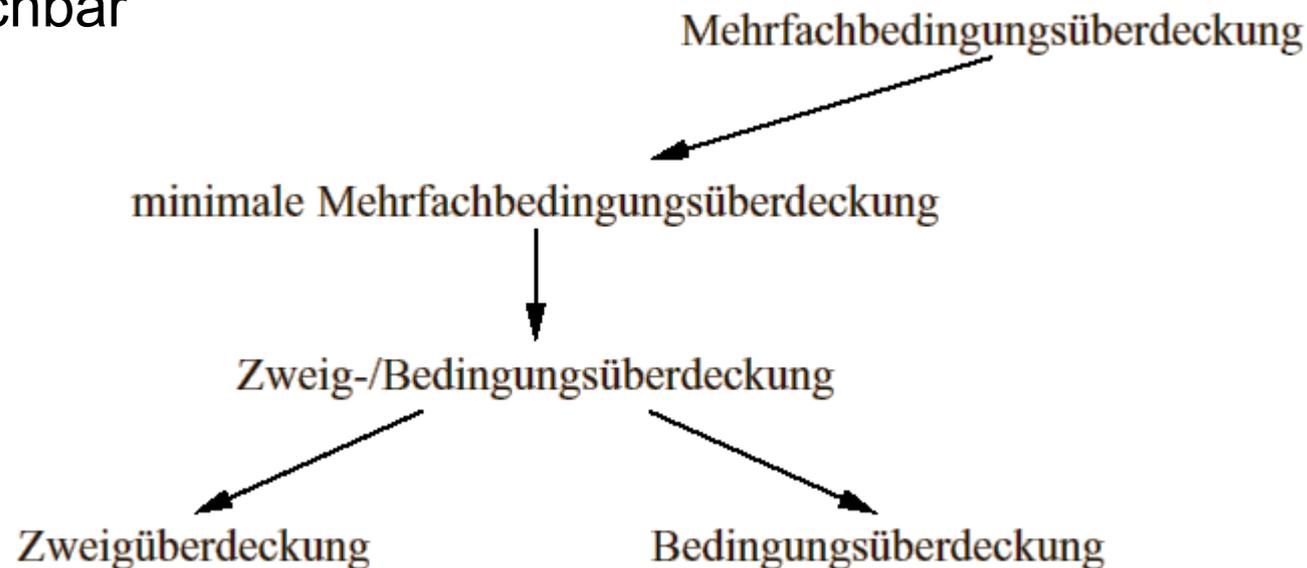
Konkreter Testfall:  $\{ m = 4, n = 4; 4 \}$



```
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. }
```

m	n	r
4	4	4

- Zusammenhang der kontrollflussbezogenen Verfahren
  - Eine Kante von Kriterium K1 nach K2 bedeutet, dass K1 das Kriterium K2 strikt enthält, d.h. dass K1 Kriterium K2 enthält, aber K2 nicht K1 enthält.
  - K1 enthält K2, genau dann, wenn jede Testdatenmenge, die K1 erfüllt, auch K2 erfüllt
  - Falls kein Weg von K1 nach K2 existiert, sind die Kriterien unvergleichbar



- Anzahl der Tests pro Testkriterium
  - Aufwand nur vergleichbar für Testen einer einzelnen Entscheidung mit  $n$  Termen

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

- Anzahl der Testdaten pro Testkriterium
  - Falls die Anzahl der ausgehenden Kanten pro Entscheidungsknoten und die Anzahl der Variablen pro Segment durch eine Konstante begrenzt sind, dann gilt für ein Programm mit  $n$  Segmenten im schlechtesten Fall:
    - Pfadüberdeckung ( $C_4$ -Überdeckung; jeder Pfad wird mindestens einmal durchlaufen): keine a-priori-Grenze für die Anzahl von Testdaten (nur vorhanden bei vorhandener Grenze an Anzahl der Schleifendurchläufe)
    - Strukturierte Pfadüberdeckung ( $C_p(k)$ ; jede Schleife wird höchstens  $k$  mal ausgeführt):  $O(2^n)$  Testdaten
    - Zweig- und Anweisungsüberdeckung:  $O(n)$  Testdaten

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

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

- Unterschiedliche Quoten beruhen auf unterschiedlichen Untersuchungsansätzen

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

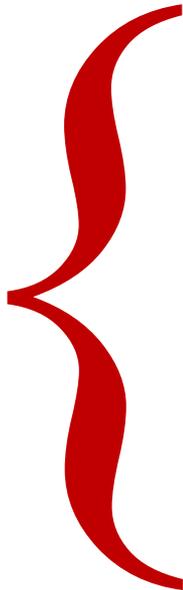
Sowohl die Anweisungs- als auch die Entscheidungsüberdeckung ist für objektorientierte Systeme nur **unzureichend** geeignet.

- Komplexität objektorientierter Systemen meist in den Beziehungen zwischen den Klassen bzw. den Interaktionen zwischen Objekten verborgen.
- Methoden in den Klassen sind normalerweise wenig umfangreich und von geringer Komplexität (McCabe: 2-4).
- Die geforderte Anweisungs- und/oder Entscheidungsüberdeckung ist dann mit wenig Aufwand erreichbar.

Wenn eine Werkzeugunterstützung zur Ermittlung der Überdeckungswerte vorhanden ist, kann diese allerdings genutzt werden, um nicht aufgerufene Methoden oder Programmteile zu erkennen.



## 2.3 White-Box- Test



---

Idee der White-Box Testentwurfsverfahren

---

Kontrollflussbasierter Test

---

Datenflussbasierter Test

---

Test der Bedingungen

---

Weitere White-Box Testentwurfsverfahren

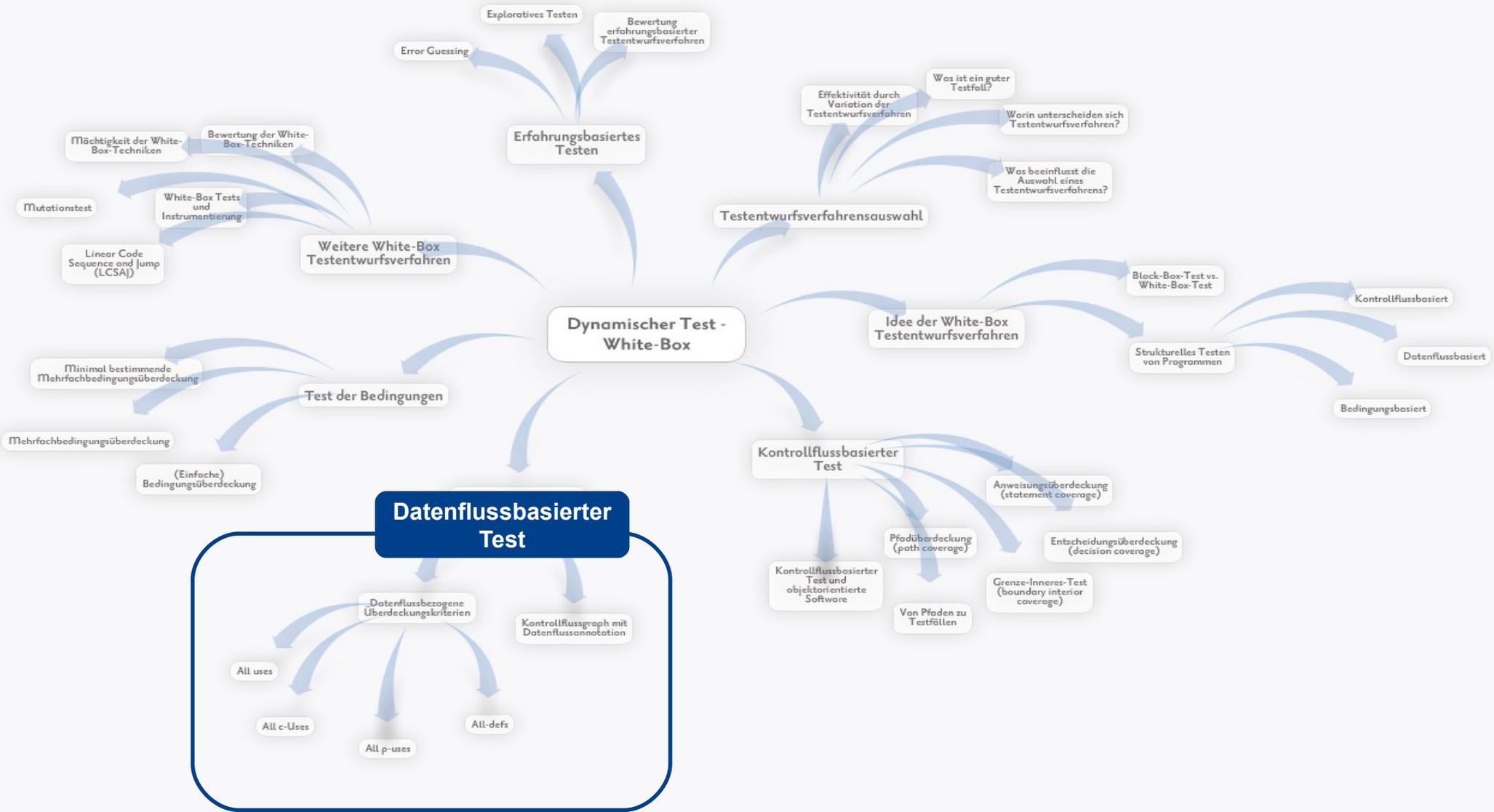
---

Erfahrungsbasiertes Testen

---

Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung

---



Dynamischer Test, bei dem die Testfälle unter Berücksichtigung der Datenverwendungen im Testobjekt hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der Datenverwendung bewertet wird.

Hypothese: fehlerhafte Datenverwendung !

Test bezüglich der Variablen/Objektverwendung:

- Wertzuweisung, zustandsverändernd
  - z.B.  $r = m$  oder  $r = 5$
  - Definitional use,  $\text{def}(r)$
- Benutzung in Ausdrücken, zustandserhaltend
  - z.B.  $r = m \bmod n$  oder  $r = \text{op1}(m, n)$
  - Computational use, c-use  $(m, n)$  und  $\text{def}(r)$
- Benutzung in Bedingungen, zustandserhaltend
  - z.B.  $\text{while}(r \neq 0)$  oder  $\text{if}(r \neq 0)$
  - Predicative use, p-use  $(r)$

- Datenflussbezogene Testverfahren sind (wie kontrollflussorientierte Verfahren) white-box-Verfahren, denn sie nutzen die Kenntnis der Programmstruktur aus.
- Sie sind dynamische Verfahren, denn sie basieren auf der Ausführung des Programms mit verschiedenen Testdaten.
- Idee: Getestet werden Interaktion zwischen Anweisungen, die den Wert einer Variablen berechnen (definieren), und Anweisungen, die diesen Variablenwert benutzen (referenzieren).

- Beispiel: Anweisung  $A=B+C*D$  liefert ein falsches Ergebnis, wobei die Anweisung korrekt ist, aber die berechneten Werte vorher falsch berechnet wurden
- Datenflussorientierte Verfahren unterscheiden sich darin, ob sie alle Interaktionen oder nur einen Teil davon testen (->Überdeckungsmaße)
- Definition der Überdeckungsmaße orientiert sich am Kontrollflussgraphen, erweitert um zusätzliche Informationen
  - Datenflussgraph

- Datenflussgraph
  - Ein Datenflussgraph ist ein Kontrollflussgraph, bei dem zusätzlich zu jedem Knoten  $k$  die Mengen  $DEF(k)$ ,  $UNDEF(k)$ , und  $REF(k)$  gehören.
  - $DEF(k)$ : Menge der Variablen  $x$ , für welche die Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, der Variablen  $x$  einen Wert zuweist, der nicht anschließend in  $f$  undefiniert wird.
  - $UNDEF(k)$ : Menge der Variablen  $x$ , für welche die Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, die Variablen  $x$  in einen undefinierten Zustand überführt, ohne  $x$  anschließend in  $f$  neu zu definieren.

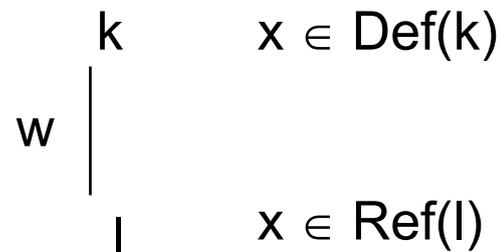
- Datenflussgraph ff.
  - REF(k): Menge der Variablen  $x$ , für welche die Anweisungsfolge  $f$ , die zum Knoten  $k$  gehört, die Variable  $x$  referenziert, ohne dass  $x$  vorher in  $f$  undefiniert wird.
    - Dabei zu berücksichtigen: Rein lokale Datenflüsse müssen vermieden werden, denn die werden bei den datenflussbezogenen Testkriterien nicht berücksichtigt. Ein Datenfluss heißt rein lokal, wenn intern eine Referenz auf eine Definition folgt.
  - Datenflussschema
    - Entsteht durch Ersetzen der Anweisungen und Entscheidungsprädikate durch formale Bezeichner.



- Datenflussgraph: Beispiele

- $X:=A+B; Y:=C*D; B:=Y, \text{FREE}(A); Y:=A+Z$  sei Anweisungsfolge des Knotens  $k$ 
  - Enthält rein lokalen Datenfluss von  $Y:=C*D$  nach  $B:=Y$ , also auf 2 Knoten aufteilen:
    - $X:=A+B; Y:=C*D$ ; sei Anweisungsfolge des Knotens  $k$ 
      - $\text{DEF}(k) = \{X, Y\}$
      - $\text{REF}(k) = \{A, B, C, D\}$
      - $\text{UNDEF}(k) = \{\}$
    - $B:=Y, \text{FREE}(A); Y:=A+Z$  sei Anweisungsfolge des Knotens  $l$ 
      - $\text{DEF}(l) = \{Y, B\}$
      - $\text{REF}(l) = \{Y, Z\}$
      - $\text{UNDEF}(l) = \{A\}$

- Datenflussgraph: weitere Definitionen
  - Die Zuordnung von bedingten Anweisungen zu Knoten sei so gewählt, dass diese Knoten (auch Entscheidungsknoten) nur Referenzen von Variablen enthalten, d.h.  $DEF(K)=UNDEF(K)=\{\}$ 
    - Beispiel:  $If ((B=C+D))$  aufsplitten in  $B=C+D$  und  $if B$
  - Man sagt „Eine Definition von  $x$  im Knoten  $k$  erreicht eine Referenz von  $x$  im Knoten  $l$  über den Weg  $w$ “
    - genau dann, wenn der Weg  $w$  im Kontrollflussgraphen von  $k$  nach  $l$  führt und die Variable  $x$  auf dem Weg nicht neu definiert oder undefiniert wird.



Kriterium „**alle Definitionen**“ (**all-defs**): Jede Definition mindestens einmal (ohne dazwischenliegendes erneutes def) in einem c-use oder p-use verwenden.

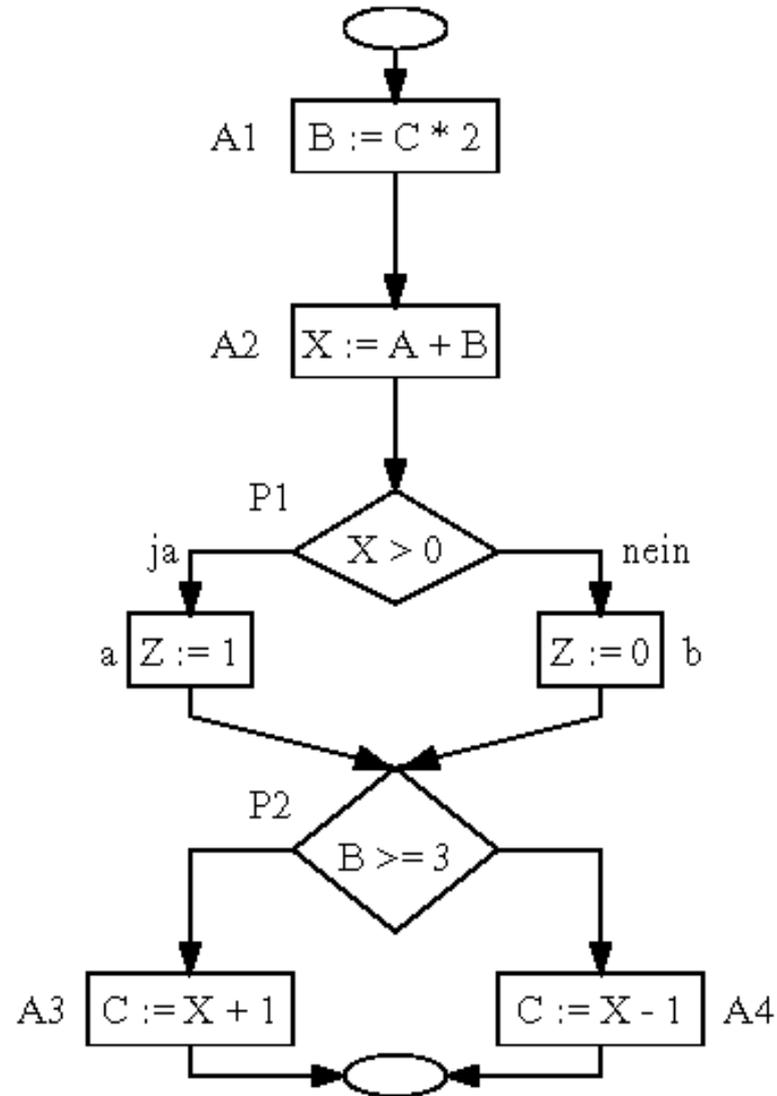
Kriterium „**alle DR-Interaktionen**“: jedes Paar def/ref (ohne dazwischenliegendes erneutes def) auf irgendeinem Weg ausführen.

Weitere Kriterien:

- „**Alle B-Referenzen**“ (**all-c-uses**)
- „**Alle E-Referenzen**“ (**all-p-uses**)
- **3-DR-Interaktionen**
- **Kontextüberdeckung**

- 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

- Kriterium „alle Definitionen“
  - Kriterium: Das Resultat einer Zuweisung (Definition) wird wenigstens einmal benutzt
  - Eine Testdatenmenge  $T$  erfüllt das Kriterium „alle Definitionen“ g.d.w. es
    - für jede Variable  $x$  und jede Definition von  $x$  mindestens einen Weg in  $Wege(T)$  gibt, auf dem die Definition eine Referenz von  $x$  erreicht.

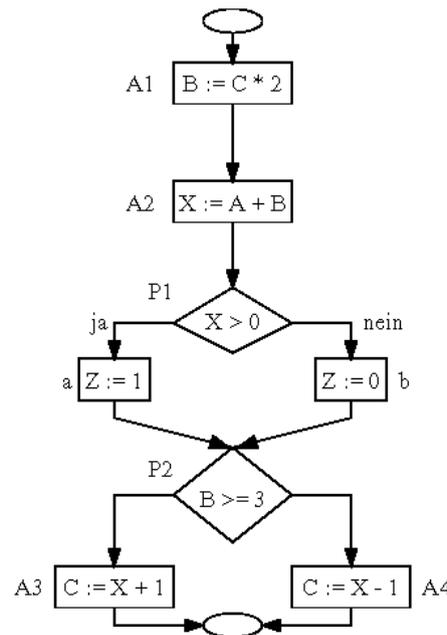


## • Kriterium „alle Definitionen“: Beispiel

- T sei die Testdatenmenge, die den Weg (A1, A2, P1, a, P2, A3) ausführt
- T erfüllt das Kriterium „alle Definitionen“ für die Variablen X und B
- Aber: nicht ALLE Paare von Definitionen/Referenzen einer Variablen werden getestet (Definition von X, Referenz von X in A4)

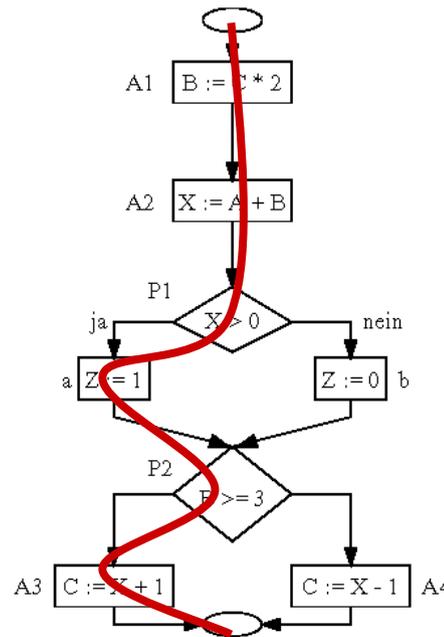
$X \in \text{Ref}(P1)$

$B \in \text{Ref}(P2)$



## • Kriterium „alle Definitionen“: Beispiel

- T sei die Testdatenmenge, die den Weg (A1, A2, P1, a, P2, A3) ausführt
- T erfüllt das Kriterium „alle Definitionen“ für die Variablen X und B
- Aber: nicht ALLE Paare von Definitionen/Referenzen einer Variablen werden getestet (Definition von X, Referenz von X in A4)



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

- Kriterium „alle DR-Interaktionen“
  - Ziel: ALLE Paare von Definitionen/Referenzen einer Variablen testen
  - Eine Testdatenmenge T erfüllt das Kriterium „alle DR-Interaktionen“ g.d.w. es
    - für jede Variable x, jede Definition von x und jede Referenz von x, die davon erreicht wird, mindestens einen Weg in  $Wege(T)$  gibt, auf dem die Definition die Referenz von x erreicht.

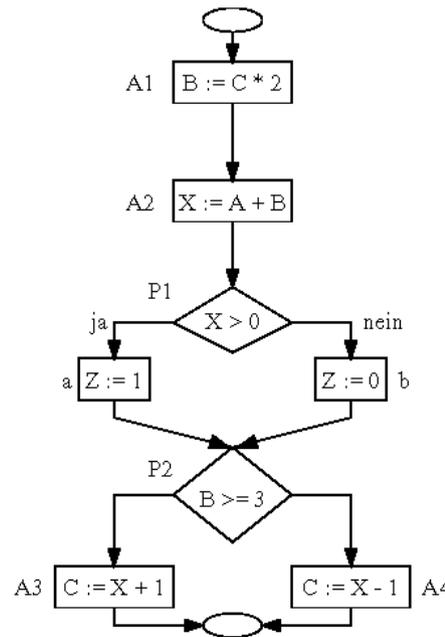
## • Kriterium „alle DR-Interaktionen“: Beispiel

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

$X \in \text{Ref}(P1)$

$B \in \text{Ref}(P2)$

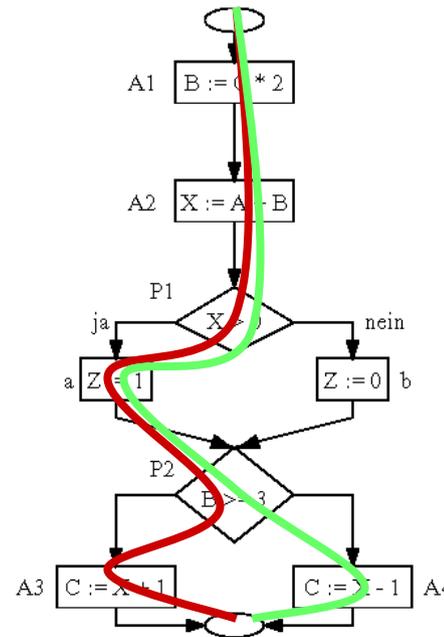
$X \in \text{Ref}(A3)$



$X \in \text{Ref}(A4)$

## • Kriterium „alle DR-Interaktionen“: Beispiel

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



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

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

- Motivation: Für die Referenz einer Variablen in einem Entscheidungsknoten ist es von Bedeutung wie der Ausgang der Entscheidung ist.
- Kriterium „alle Referenzen“
  - Ziel: alle ausgehenden Kanten eines Entscheidungsknotens berücksichtigen
  - Eine Testdatenmenge  $T$  erfüllt das Kriterium „alle Referenzen“ g.d.w. es
    - für jede Variable  $x$ , jede Definition von  $x$  in einem Knoten  $k$ , jede Referenz von  $x$  in einem Knoten  $l$ , die von der Definition in  $k$  erreicht wird, und für jeden Nachfolgerknoten  $m$  von  $l$  die Wegemenge  $Wegemenge(T)$  mindestens ein Wegstück  $u^*m$  enthält, wobei die Definition von  $x$  in  $k$  die Referenz von  $x$  in  $l$  über den Weg  $u$  erreicht.
    - Hierbei:  $u^*m$  Knotenfolge mit  $u=\{k,\dots,l\}$  gefolgt von Knoten  $m$ , d.h.  $u^*m=\{k,\dots,l,m\}$

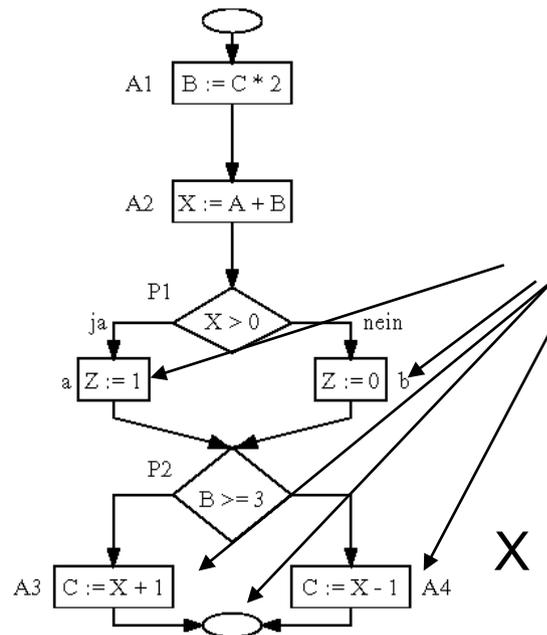
Kriterium „alle Referenzen“ (= all-uses, impliziert all-c-uses und all-p-uses)

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

$X \in \text{Ref}(P1)$

$B \in \text{Ref}(P2)$

$X \in \text{Ref}(A3)$

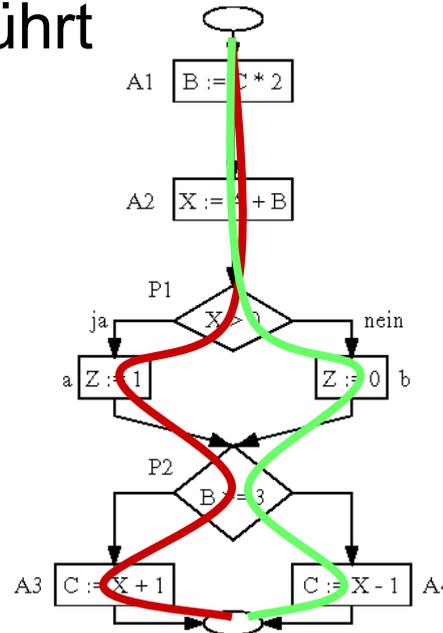


Nachfolgerknoten m

$X \in \text{Ref}(A4)$

- Kriterium „alle Referenzen“

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



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

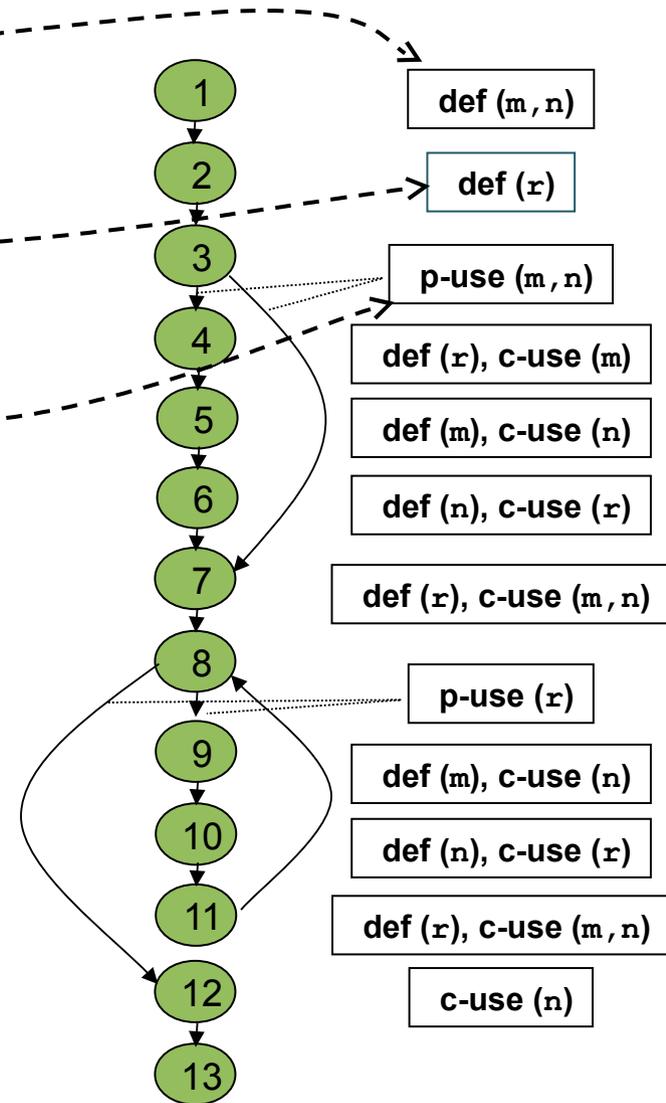
Weg (A1, A2, P1, b, P2, A4)

- Einfache Datenflusskriterien
  - „Alle DR-Interaktionen“ und „alle Referenzen“ dienen als Kriterien zum Testen aller Paare von Definitionen und Referenzen, aber jeweils nur auf einem Weg von Definition zur Referenz.
  - Dies ist allerdings unter dem Gesichtspunkt des Datenflusses ausreichend, da zwischen Definition und Referenz keine Änderung der Variablen erfolgen kann.

# Kontrollflussgraph mit Datenflussannotation



```
1. public int ggt (int m, int n) {  
  // pre: (m > 0) and (n > 0)  
  // post: ...  
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. }
```



# Datenflussbezogene Überdeckungskriterien: All-defs

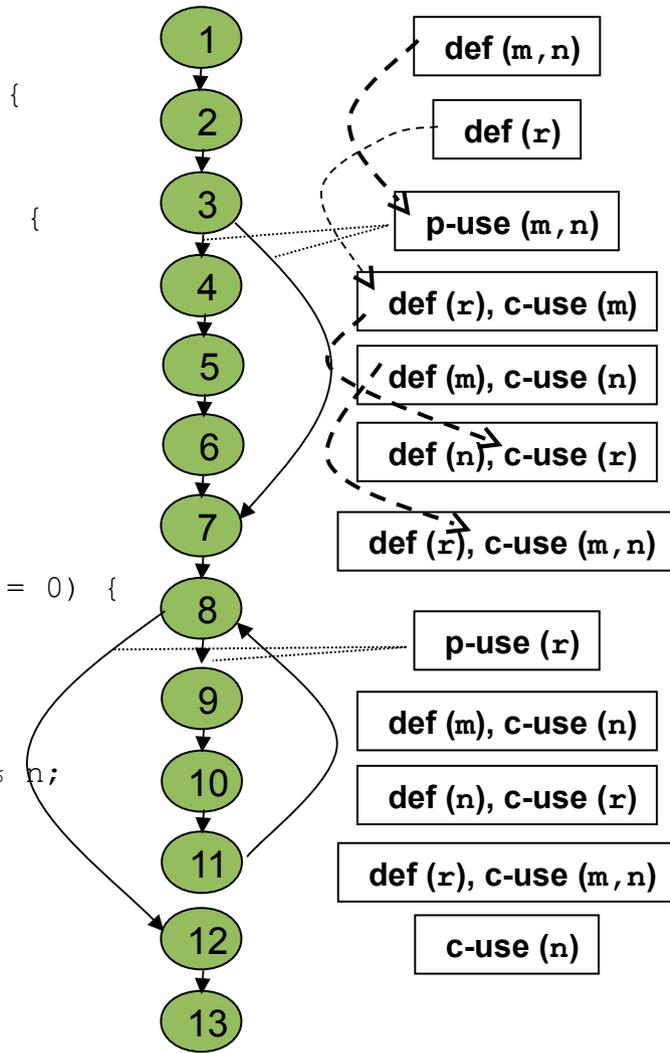
**All-defs:** Jede Definition min. einmal ohne dazwischen liegendes erneutes def in einem c-use oder p-use verwendet.

Beispiel:

All-defs:

- 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

```
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. }
```



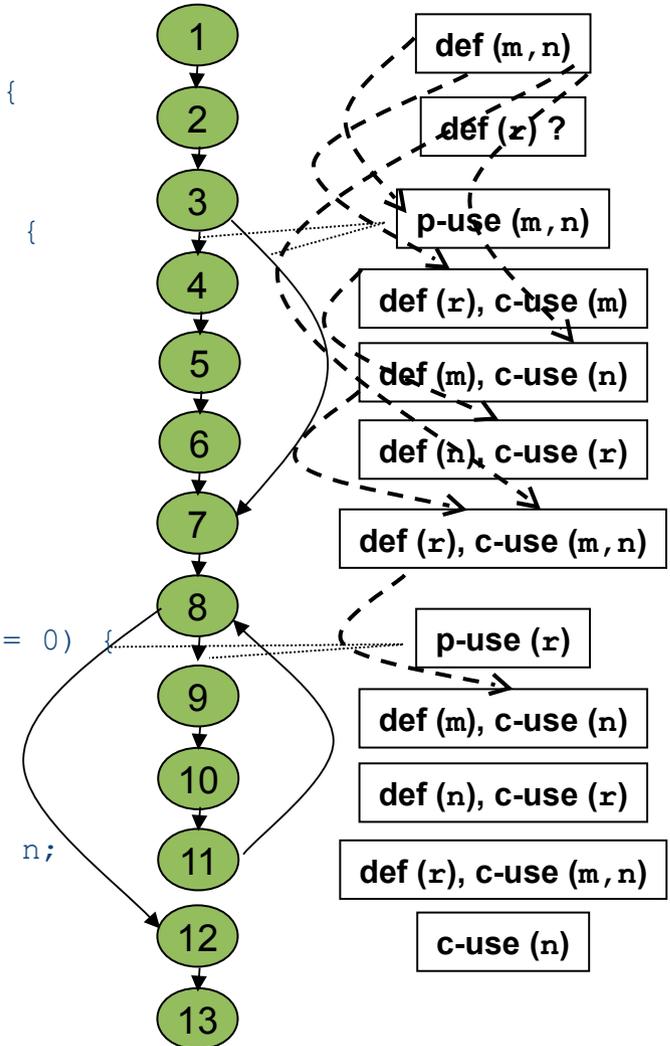
**Kriterium alle DR-Interaktionen :**  
jedes Paar def/ref (ohne dazwischenliegendes erneutes def) auf irgendeinem Weg ausführen

**Beispiel:**

- 1, def m: p-use 3-4 (Kante!)
- 1, def m: c-use 4
- 1, def m: c-use 7 (else!)
- 1, def n: p-use 3-4
- 1, def n: c-use 5
- 1, def n: c-use 7 (else!)
- 4, def r: c-use 6
- 5, def m: c-use 7
- 7, def r: p-use 8-9
- .....

```

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. }
    
```



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

4, def r: c-use 6

5, def m: c-use 7

6, def n: c-use 7

9, def m: c-use 11

10, def n: c-use 11

## Alle E-Referenzen(p-uses)

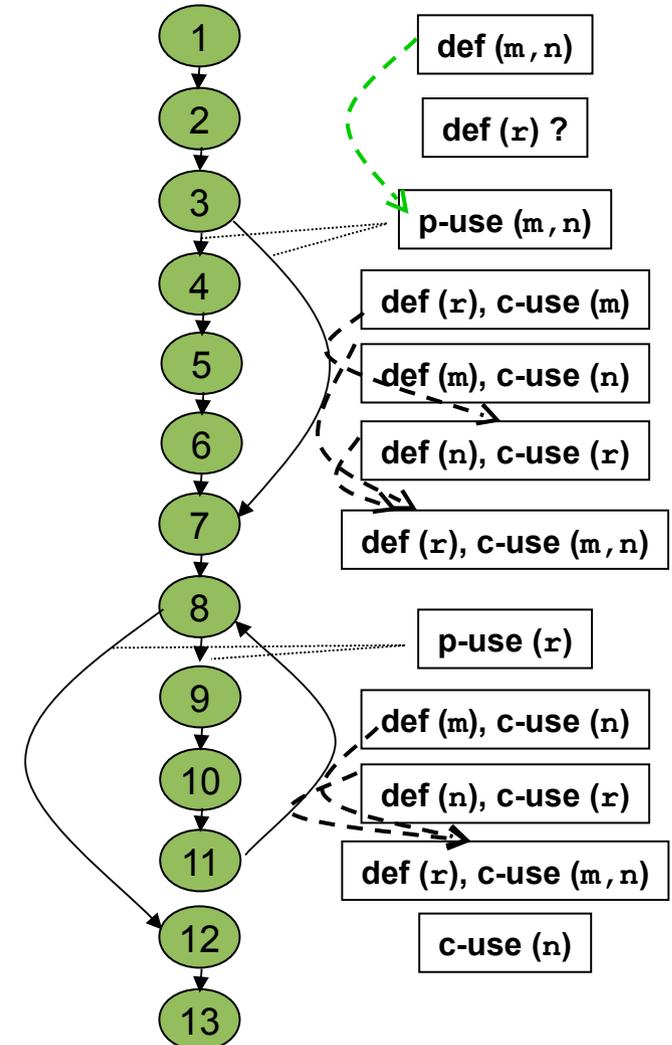
1, def m

p-use 3-4

.....

## Weitere Kriterien:

- alle k-DR-Interaktionen
- Kontextüberdeckung



## Beispiele für alle 3-DR-Interaktionen:

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

m wird in 1 definiert, in 4 referenziert,  
r wird in 4 definiert und in 6 referenziert  
→ damit hängt 6 von 1 ab!

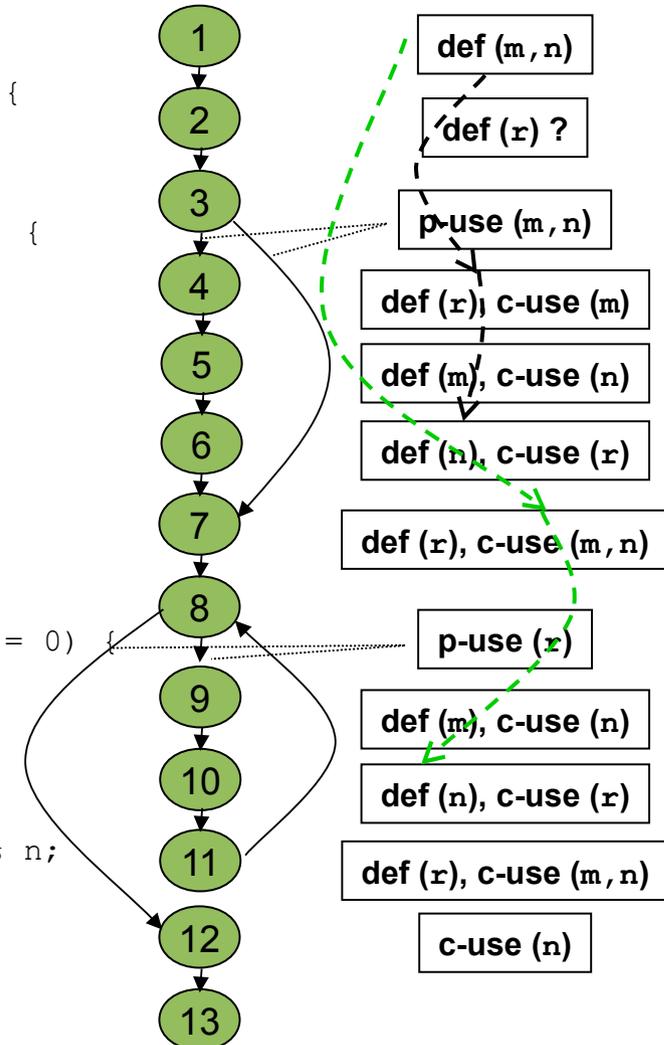
(1, m, 7, r, 10)

.....

→ Teste die 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.   }
8.   r = m % n;
9.   while (r != 0) {
10.    m = n;
11.    n = r;
12.    r = m % n;
13.  }
14.  return n;
15. }
    
```



## Beispiele für Kontextüberdeckung

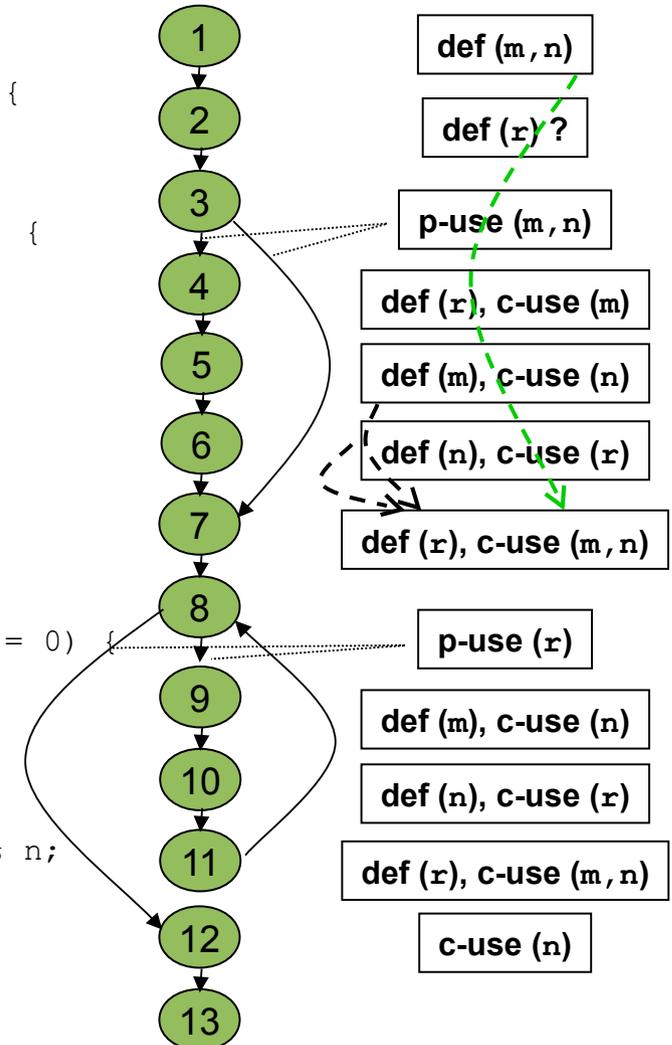
In 7 wird  $r$  definiert durch die referenzierten Variablen  $m$  und  $n$ . Der „Definitions-kontext“ dafür sind die Knoten, in denen diese Werte von  $m$  und  $n$  vorher definiert worden sind, also gibt es hier zwei Fälle:

DK1 =  $\{(1,m), (1,n)\}$

DK2 =  $\{(5,m), (6,n)\}$

→ teste die 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.   r = m % n;
9.   while (r != 0) {
10.    m = n;
11.    n = r;
12.    r = m % n;
13.  }
14. return n;
15. }
```



- Anzahl der Testdaten pro Testkriterium
  - Falls die Anzahl der ausgehenden Kanten pro Entscheidungsknoten und die Anzahl der Variablen pro Segment durch eine Konstante begrenzt sind, dann gilt für ein Programm mit  $n$  Segmenten im schlechtesten Fall:
    - Kriterium „alle DR-Wege“ erfordert  $O(2^n)$  Testdaten
    - Kriterien „alle E-/einige B-Referenzen“, „alle B-/einige E-Referenzen“, „alle Referenzen“ und Kontextüberdeckung erfordern  $O(n^2)$  Testdaten
    - Kriterium „alle Definitionen“ erfordert  $O(n)$  Testdaten

- 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%

- Für restliche Kriterien keine Studien vorhanden

Testverfahren unterscheiden nach dem Aufwand (bei  $n$  Segmenten im Programmteil):

- „Alle Definitionen“ ist am einfachsten ( $O(n)$  Testdaten).
- „Alle E-Referenzen“, „alle B-Referenzen“, Kontextüberdeckung bzw. „alle DR-Wege“ sind am aufwendigsten ( $O(n^2)$  bzw  $O(2^n)$  Testdaten).

Testverfahren unterscheiden sich nach der Fehleraufdeckungsfähigkeit:

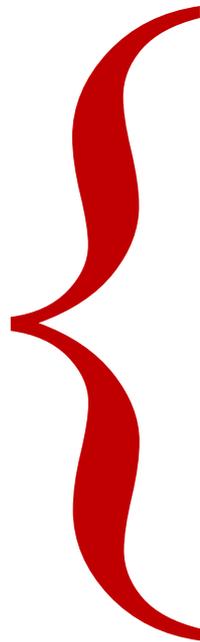
- „Alle B-Referenzen“ fand bspw. bis zu 88% aller Berechnungsfehler.
- „Alle E-Referenzen“ fand bspw. 100% aller Bereichsfehler.
- Folgende Fehler werden dagegen schlecht aufgedeckt:
  - Fehlende Pfade
  - Bereichsfehler durch falsch platzierte Anweisungen und falsche arithmetische Operatoren
  - Berechnungsfehler bei speziellen Werten (Grenzwerte und spezielle Werte werden nicht verlangt)

Ca. 9% aller Fehler wurden nur mit datenflussbezogenen Methoden gefunden.

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



## 2.5 White-Box- Test



Idee der White-Box Testentwurfsverfahren

Kontrollflussbasierter Test

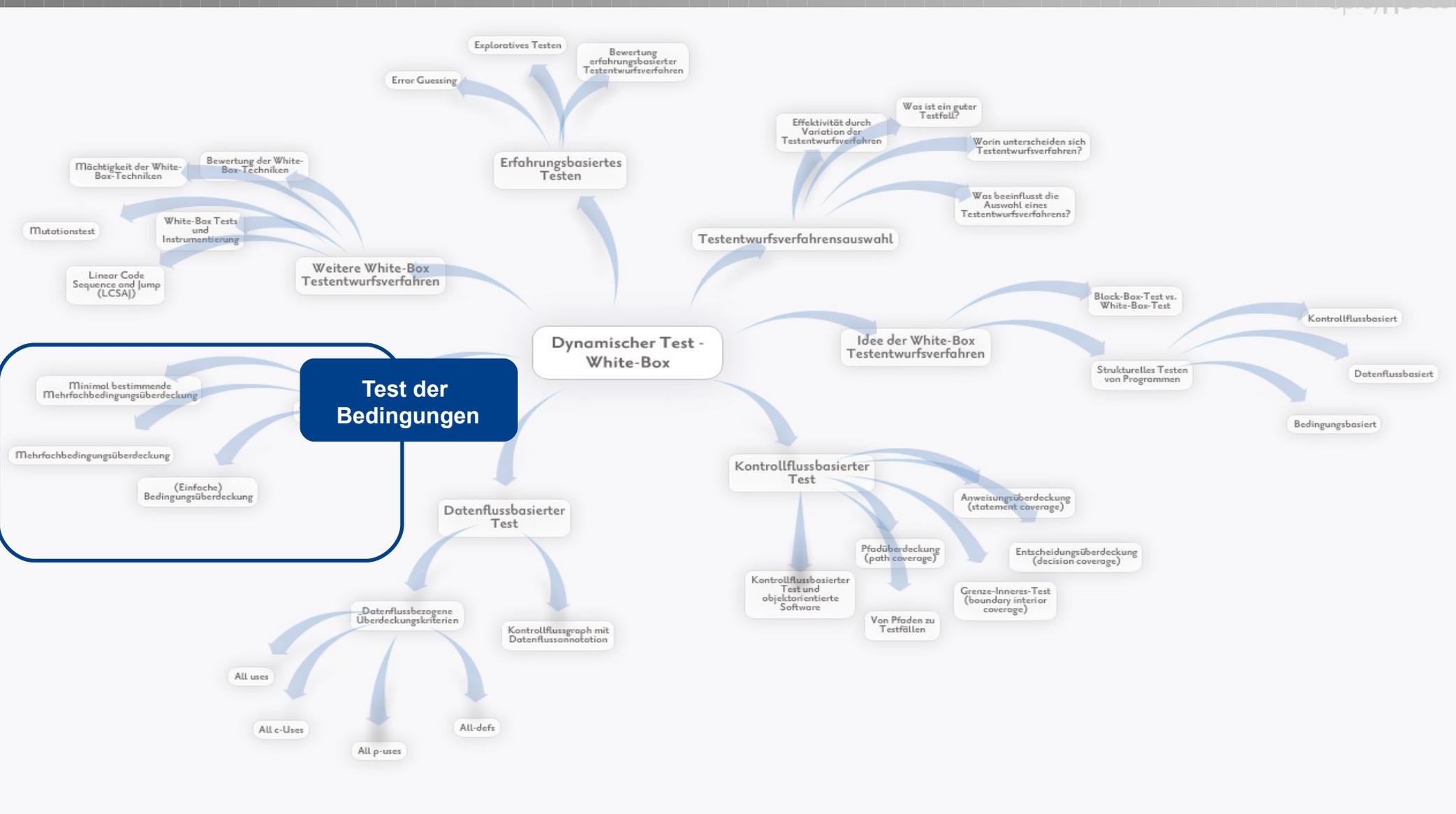
Datenflussbasierter Test

Test der Bedingungen

Weitere White-Box Testentwurfsverfahren

Erfahrungsbasiertes Testen

Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung



- 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 ist (zusammengesetzte) Bedingung, die den 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 {...}`

Bei der **Entscheidungsüberdeckung** wird ausschließlich der ermittelte **Ergebnis-Wahrheitswert** einer Bedingung berücksichtigt.

- Anhand dieses Wertes wird entschieden, welche Verzweigung im Kontrollflussgraphen verfolgt wird bzw. welche Anweisung als nächste im Programm zur Ausführung kommt.

**Problem:** Setzt sich eine Bedingung aus **mehreren Teilbedingungen** zusammen, die über logische Operatoren miteinander verknüpft sind, so muss im Test die strukturelle Komplexität der Bedingung berücksichtigt werden.

Hierbei werden unterschiedliche Anforderungen und damit auch Abstufungen der Testintensität mit Berücksichtigung der zusammengesetzten Bedingungen unterschieden.

Als **Überdeckungskriterien** werden Verhältnisse zwischen den bereits erreichten und allen geforderten Wahrheitswerten der (Teil-)Bedingungen gebildet.

Bei den Testentwurfsverfahren, welche die Komplexität der Bedingungen im Programmtext des Testobjekts in den Mittelpunkt der Prüfung stellen, ist es sinnvoll, eine vollständige Prüfung (100% Überdeckung) anzustreben.

Im Folgenden:

- **(Einfache) Bedingungsüberdeckung** (*condition coverage*)
- **Mehrfachbedingungsüberdeckung** (*multiple condition coverage*)
- **Minimal bestimmende Mehrfachbedingungsüberdeckung** (*condition determination coverage*).

**Bedingungsüberdeckung:** Der Anteil der (atomaren) Teilbedingungen, die durch eine Gruppe von Testfällen ausgeführt worden sind. 100% (einfache) Bedingungsüberdeckung bedeutet, dass jede (atomare) Teilbedingung in jeder Entscheidung mindestens einmal mit den Werten True und False ausgeführt wurde.

**Einfacher Bedingungstest:** Ein White-Box-Testentwurfverfahren, bei dem Testfälle so entworfen werden, dass Bedingungswege zur Ausführung kommen.

Überdeckung der atomaren Teilbedingungen einer Entscheidung mit »wahr« und »falsch« gefordert: Teste jeden atomaren Ausdruck einmal zu wahr und einmal zu falsch. Bei  $n$  atomaren Ausdrücken mindestens 2, höchstens  $2n$  Testfälle.

$$\text{Bedingungsüberdeckungsgrad} = \frac{\text{Anzahl zu wahr und falsch getesteten atom A}}{\text{Gesamtzahl atomarer Ausdrücke}}$$

Achtung: Die einfache Bedingungsüberdeckung ist ein schwächeres Kriterium als die Anweisungs- oder auch Entscheidungsüberdeckung, da nicht verlangt ist, dass unterschiedliche Wahrheitswerte bei der Auswertung der gesamten Bedingung im Test zu berücksichtigen sind.

# Beispiele zur einfachen Bedingungsüberdeckung

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

**Mehrfachbedingungstest:** Ein White-Box-Testentwurfsverfahren, das die Überdeckung der atomaren Teilbedingungen einer Entscheidung mit WAHR und FALSCH in allen Kombinationen fordert (engl. branch condition combination testing).

Bei  $n$  atomaren Ausdrücken (a.A.) ist die Testfall-Anzahl  $= 2^n$

- Wächst exponentiell mit der Anzahl unterschiedlicher atomarer Ausdrücke !

$$\text{Mehrfachbedingungsüber.grad} = \frac{\text{Anzahl getesteter Kombinationen atom A}}{2 \text{ Gesamtzahl atomarer Ausdrücke}}$$

Bei der Auswertung der Gesamtbedingung ergeben sich i.d.R. auch beide Wahrheitswerte (sonst: Tautologie !).

- Die Mehrfachbedingungsüberdeckung erfüllt somit auch die Kriterien der Anweisungs- und Entscheidungsüberdeckung.
- Sie ist ein umfassenderes Kriterium, da sie auch die Komplexität bei zusammengesetzten Bedingungen berücksichtigt.

Problem: Manche Kombinationen sind nicht durch konkrete Testfälle realisierbar:

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

# Beispiel: Mehrfachbedingungsüberdeckung

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

# Minimal bestimmende Mehrfachbedingungsüberdeckung

**Minimal bestimmende Mehrfachbedingungsüberdeckung:** Der Anteil aller einfachen Bedingungsergebnisse, die von einer Testsuite ausgeführt wurden und unabhängig voneinander einen Entscheidungsausgang beeinflussen (engl. Modified branch condition combination testing).

Teste den (Gesamt-)Ausdruck einmal zu wahr und einmal zu falsch sowie jede Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswertes eines atomaren Ausdrucks den Wahrheitswert des zusammengesetzten Ausdrucks ändern kann (MM-Kombinationen) !

$$\text{Min. best. Mehrfachbed.üb.grad} = \frac{\text{Anzahl getesteter MM Kombinationen}}{\text{Gesamtzahl MM Kombinationen}}$$

100% minimal bestimmende Mehrfachbedingungsüberdeckung impliziert 100% Entscheidungsüberdeckung.

Gesamtzahl der MM-Kombinationen ist bei reinen and- bzw. or-Bedingungen mit n atomaren Ausdrücken nur n+1.

# Beispiel: Minimal bestimmende Mehrfachbedingungsüberdeckung

Teste den (Gesamt-)Ausdruck einmal zu wahr und einmal zu falsch sowie jede Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswertes eines atomaren Ausdrucks den Wahrheitswert des zusammengesetzten Ausdrucks ändern kann!

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

# Übung: Minimal bestimmende Mehrfachbedingungsüberdeckung



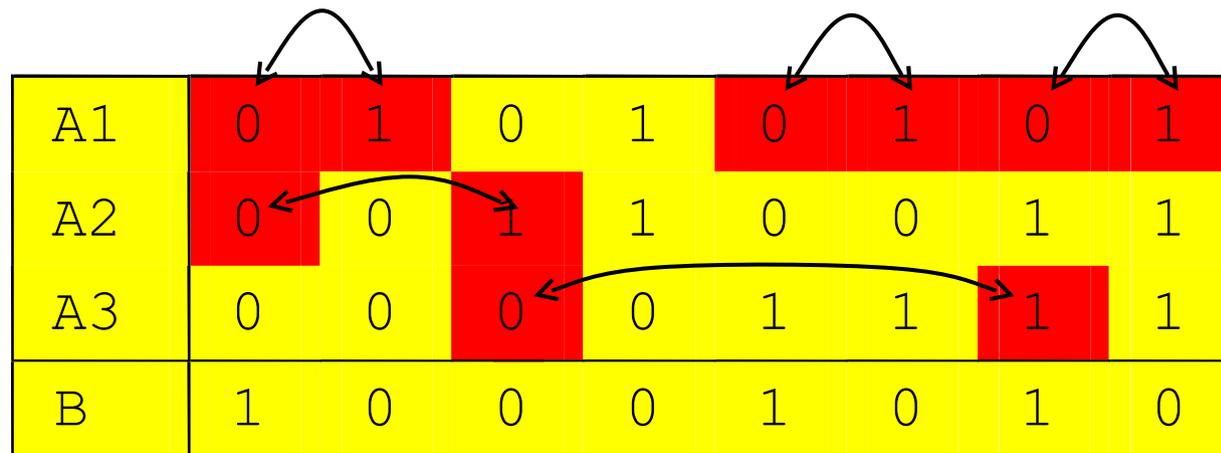
Seien  $A_1$ ,  $A_2$ ,  $A_3$  atomare Ausdrücke und  $B = \text{NOT}(A_1) \text{ AND NOT}(A_2 \text{ AND NOT}(A_3))$  der durch die unten abgebildete Wahrheitstabelle definierte Gesamtausdruck.

Frage: Welche Werte-Kombinationen von  $A_1$ ,  $A_2$  und  $A_3$  sind nach der minimal bestimmenden 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

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 den (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

Die in der DO-178B (Software Considerations in Airborne Systems and Equipment Certification) geforderte »*Modified Condition/Decision Coverage*« (MC/DC) ist ähnlich der Minimal bestimmenden Mehrfachbedingungsüberdeckung (*condition determination coverage*).

- Fordert aber nur, für jeden atomaren Ausdruck in mindestens einem Fall zu zeigen, dass er alleine die Gesamtentscheidung beeinflussen kann.
- Es werden zwei MM-Kombinationen pro atomaren Ausdruck benötigt.
- Bei  $n$  atomaren und/oder/nicht-verknüpften Ausdrücken ist die Testfall-Anzahl bei MC/DC mindesten  $n+1$  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

Diagram illustrating the Modified Condition/Decision Coverage (MC/DC) matrix. The matrix shows the values of atomic expressions (A1, A2, A3) and the overall decision (B) for various combinations of conditions. Red numbers indicate the values of the atomic expressions. Arrows indicate the dependencies between the atomic expressions and the overall decision.

Auf die intensive Prüfung oder Aufteilung von komplexen Bedingungen kann möglicherweise ganz verzichtet werden, wenn diese vor dem dynamischen Test einem Code-Review unterzogen werden und deren Korrektheit dort nachgewiesen wird.

Es kann sinnvoll sein, komplexe zusammengesetzte Bedingungen in verschachtelte, einfache Abfragen aufzuteilen und für diese Abfolge von Abfragen dann einen Entscheidungstest durchzuführen.

Ein Nachteil der Bedingungsüberdeckungen ist, dass sie boolesche Ausdrücke beispielsweise nur innerhalb *einer* IF-Anweisung prüfen.

- Manchmal wird z.B. nicht erkannt, dass die IF-Bedingung aus mehreren Teilbedingungen zusammengesetzt ist und die minimal bestimmende Mehrfachbedingungsüberdeckung angewendet werden sollte.
- **Beispiel:** `flag = a || (b && c); if (flag) {...}`
- Alle booleschen Ausdrücke für die Erstellung der Testfälle heranziehen !

Problem: Messung der Überdeckung der Teilbedingungen.

- Einige Programmiersprachen und Compiler verkürzen die Auswertung von booleschen Ausdrücken, sobald das Ergebnis feststeht.
- Beispiel: Ist bei einer AND-Verknüpfung von zwei Teilbedingungen für eine Teilbedingung der Wert »false« ermittelt, dann ist die Gesamtbedingung ebenfalls »false«, egal welchen Wert die zweite Teilbedingung liefert.
- Einige Compiler ändern auch die Reihenfolge der Auswertung in Abhängigkeit von den booleschen Operatoren, um möglichst schnell ein Endergebnis zu erhalten und die weiteren Teilbedingungen nicht auswerten zu müssen.
- Testfälle, die eine Überdeckung von 100% erreichen sollen, können zwar ausgeführt werden, wegen der Verkürzung der Auswertung lässt sich die Überdeckung allerdings nicht nachweisen.

# Bedingungsüberdeckung und »lazy evaluation«

»lazy evaluation« bedeutet, dass Bedingungen nur so lange geprüft werden, bis der Wahrheitswert feststeht.

- Im Programm: `if (A && B) then op1 (); op2 () ...`
- Im Objectcode: `if (A) then if (B) then op1 (); op2 () ...`

Hier müssen bei der einfachen BÜ drei Fälle verwendet werden, da sonst  $B=1$  nicht wirklich im Programm ausgewertet wird (Abbruch, sobald  $A=0$  erkannt ist).

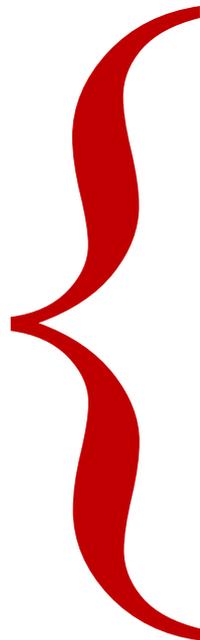
- Damit wird auch der Gesamtausdruck zu 0 und 1 ausgewertet.
- Mehrfach-BÜ und MM-BÜ sind nicht erreichbar.

Abbruch der Auswertung

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

## 2.5 White-Box- Test



---

Idee der White-Box Testentwurfsverfahren

---

Kontrollflussbasierter Test

---

Datenflussbasierter Test

---

Test der Bedingungen

---

Weitere White-Box Testentwurfsverfahren

---

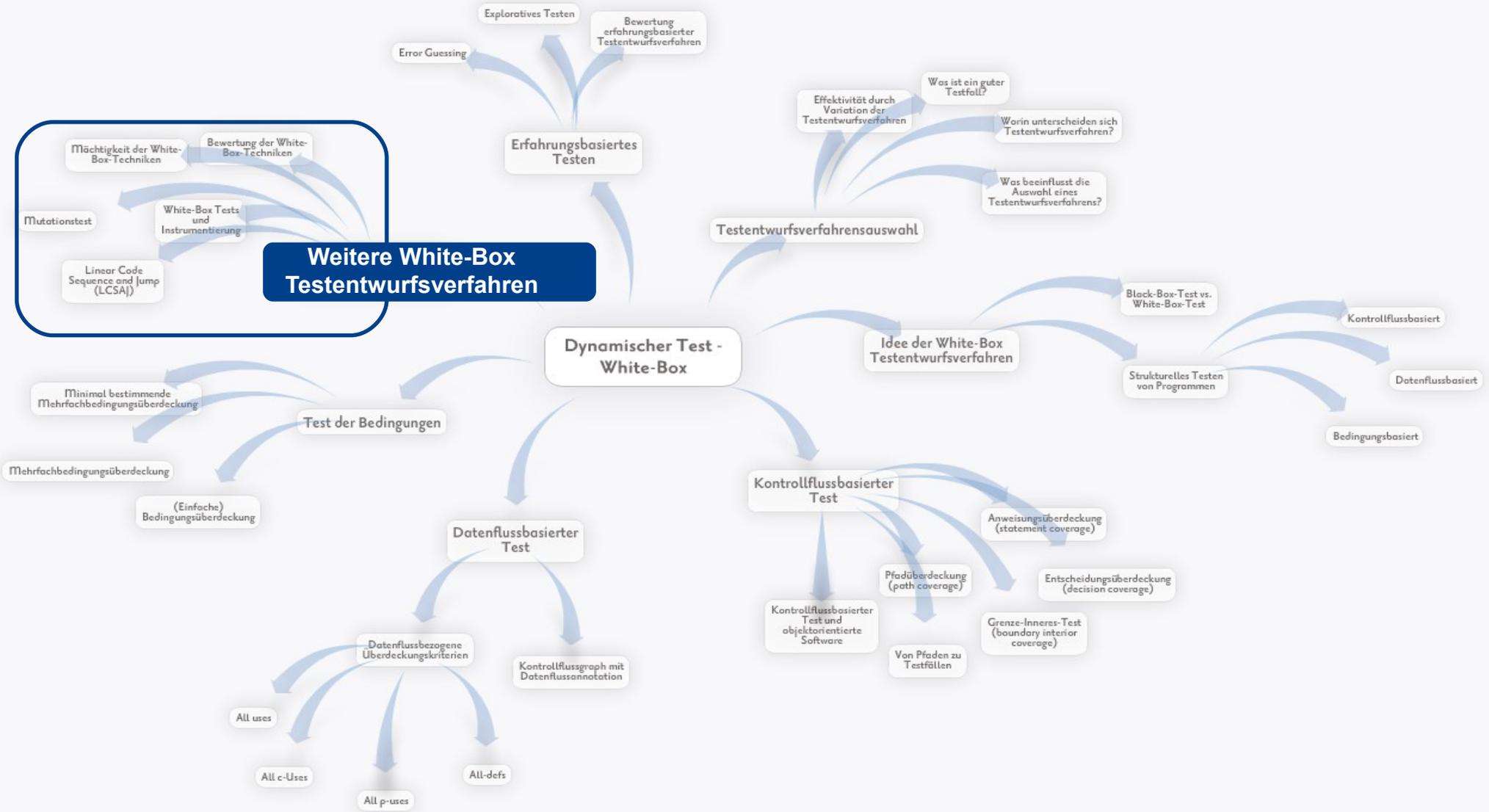
Erfahrungsbasiertes Testen

---

Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung

---

# Weitere White-Box Testentwurfungsverfahren



**Konzept:** Mögliche Fehler in Ausdrücken und Anweisungen aufspüren.

Grobe Modellierung: **Anweisungen:**

- Zugriff auf Variablen (Datenzugriff)
- Speicherung in Variablen (Datenspeicherung)

Feinere Modellierung: **Ausdrücke** und **Relationen**

- Arithmetische Ausdrücke
- Arithmetische Relationen
- Boolesche Ausdrücke (→ **bedingungs-basiertes Testen**)

## Datenzugriff:

- Fehlerart: Zugriff auf falsche Variable
- Testdaten: **Datenzugriffskriterium:**  
Alle Variablen im Programmteil müssen *vor* dem Zugriff *verschiedene* Werte haben.

## Datenspeicherung:

- Fehlerart: Speicherung in falscher Variable
- Testdaten: **Datenspeicherungskriterium:**  
Wird einer Variablen ein Wert zugewiesen, muss er anders als der bisherige Wert sein.

## Arithmetische Ausdrücke

- mit Variablen, Konstanten und  $+$ ,  $-$ ,  $*$ ,  $/$  und  $**$
- ohne „/“ ist es ein Polynom (1 Variable) oder Multinom (mehrere Variablen)

## Fehlerarten:

- Einfache additive oder multiplikative Fehler
- Fehler in Polynom oder Multinom, der die Variablenmenge nicht ändert und den höchsten Exponenten nicht erniedrigt

## Testdaten:

- **Additives/multiplikatives Fehler-Kriterium:**  
(Teil-)Ausdruck muss Wert **ungleich 0** erhalten (damit multiplikative Fehler entdeckt werden)
- **Polynom- bzw. Multinom-Kriterium:**  
Für ein **Polynom** vom Grad  $n$  sind  **$n+1$  unabhängige Testdaten** ausreichend.  
Für ein **Multinom** (Polynom mit mehreren Variablen) mit höchstem Exponenten  $n$  ist eine **Kaskadenmenge von  $k$ -Tupeln vom Grad  $n+1$**  ausreichend, wenn  $k$  die Anzahl der Variablen ist (s. Howden 1978d).

## Arithmetische Relation

- Seien A und B arithmetische Ausdrücke
- sei „r“ eines der sechs Relationssymbole  $<$ ,  $>$ ,  $\geq$ ,  $\leq$ ,  $=$ ,  $\neq$

### Fehlerarten:

- Relation „A r B“ enthält falsches Relationssymbol r.
- Relation „A r B“ ist falsch, es müsste „(A+k) r B“ mit  $k \neq 0$  realisiert werden.

### Testdaten: **Arithmetisches Relations-Kriterium:**

Es sind drei Testdaten notwendig und hinreichend, bei denen die Differenz A-B folgende Werte annimmt (mit minimalem positivem Wert  $\varepsilon$ ):

- $-\varepsilon$  (größter negativer Wert – knapp unterhalb 0)
- 0
- $\varepsilon$  (kleinster positiver Wert – knapp oberhalb von 0)

**Konzept:** Betrachtung der Zusammensetzung von (Eingabe- und Ausgabe-) **Daten** auf verschiedenen **Abstraktionsstufen**.

- **Datenkapsel** (zusammenhängende Daten, z.B. Tabelle, Klasse, struct)
- **Datenfelder** (aus denen eine Datenkapsel besteht)
- **Repräsentative Werte pro Datenfeld** (ergeben sich eher aus der Spezifikation, z.B. Äquivalenzklassen, Grenzwerte)
- **Datenzustände** (alle möglichen Kombinationen von allen repräsentativen Werten für alle Felder einer Datenkapsel)

## Testdaten für Datenüberdeckungen:

- **Datenkapselüberdeckung:** Durch Test wird mindestens ein Eingabefeld oder ein Ausgabefeld jeder Datenkapsel „angesprochen“ (d.h. wird eingelesen bzw. ausgegeben).
- **Feldüberdeckung:** Durch Test wird jedes Eingabefeld und jedes Ausgabefeld jeder Datenkapsel „angesprochen“.
- **Überdeckung repräsentativer Werte:** Durch Test wird jeder repräsentative Wert jedes Eingabe- und Ausgabefeldes jeder Datenkapsel „angesprochen“.

Testverfahren unterscheiden sich nach dem Aufwand, aber wie genau ?

Testverfahren unterscheiden sich nach der Fehleraufdeckungsfähigkeit:

- **Datenzugriffskriterium** fand **79%** aller Bereichsfehler (**100%** bei falsch platzierter Anweisung).
- **Arithmetisches Relationskriterium** fand **100%** aller Bereichsfehler durch fehlerhafte arithmetische Operatoren, falsche konstante Werte und falsch platzierte Anweisungen, aber nur **70%** aller fehlerhaften relationalen Operatoren.
- Kriterium **additive / multiplikative Fehler** fand nur
  - **56%** von fehlenden Berechnungen,
  - **13%** falsche konstante Werte,
  - **11%** fehlerhafte Variablenreferenzen;
  - **10%** falsche relationale Operatoren und
  - **alle anderen Fehler gar nicht.**
- Es gab **keine** Fehler, die **nur** mit obigen Methoden gefunden wurden !

[Quelle: Riedemann: Spezialvorlesung Test-Methoden.]

Sind Programme zu testen, die in älteren Programmiersprachen vorliegen, so können diese Testobjekte Sprünge enthalten.

- Die bisher vorgestellten Testentwurfsverfahren berücksichtigen den Test der Sprünge nur unzureichend.

Im »**Linear Code Sequence and Jump**« (**LCSAJ**) Testentwurfsverfahren werden die Folgen von Anweisungen in den Mittelpunkt der Untersuchung gestellt, die sequentiell ausgeführt werden und durch einen Sprung auf eine andere Anweisung als die nächst folgende Anweisung beendet werden.

- Kombinationen solcher linearer Codesequenzen mit einem Sprung am Ende sind beim Test zu berücksichtigen.
- Das Verfahren prüft durch die Kombination der Sequenzen mehr als die Entscheidungsüberdeckung, ist allerdings nicht so umfangreich wie die Pfadüberdeckung.

Sind mit einem Black- oder White-Box Testentwurfsv erfahren Testfälle erstellt worden und soll etwas über deren Güte, d.h. die Wahrscheinlichkeit, damit Fehler aufzudecken, ausgesagt werden, so können Fehler in ein Programm eingebaut und dann geprüft werden, ob die Testfälle diese Fehler finden.

Der »Mutationstest« systematisiert diese Idee durch die Definition von Mutationsoperatoren, mit denen Programme automatisch verändert werden:

- Mathematische Operatoren durch inverse ersetzen ( $+ \rightarrow -$ ,  $* \rightarrow /$ )
- Logische Operatoren in Bedingungen durch inverse ersetzen
- Variablenbezeichner vertauschen
- ...

Gemessen wird dann, welche Testfälle welche Mutanten »killen« (d.h. mindestens einen Fehler im Mutanten entdecken).

Mengen von Testfällen sind »gut«, wenn sie alle Mutanten »killen«.

- Erkenntnis: Programme weichen nur geringfügig von dem vorgesehenen Verhalten ab.
- Diese Abweichungen besteht meistens aus einfachen Fehlern wie Definition oder Referenz falscher Variablen.
- Ziel der Mutationsanalyse
  - Erzeugung von mehreren Versionen eines Programms  $P$ , die jeweils eine Abweichung repräsentieren.
  - Spezifikation einer Testdatenmenge  $T$ , die in der Lage ist, diese Abweichungen in den Programmversionen zu entdecken.  
→ Korrespondierende Fehler im ursprünglichen Programm  $P$  können somit erkannt werden.

## Vorgehen

- Erzeugung einer Menge von Programmen  $P_1, \dots, P_n$ , die sich vom gegebenen Programm  $P$  nur „in einer Kleinigkeit“ unterscheiden. Diese  $P_i$  heißen Mutanten von  $P$ .
- Die Mutanten werden durch Mutationstransformationen erzeugt, z.B.
  - Referenz einer anderen Variablen
  - Definition einer anderen Variablen
  - Verfälschung arithmetischer Ausdrücke um multiplikative oder additive Konstanten oder Veränderung von Koeffizienten
  - Änderung arithmetischer Relationen um additive Konstanten oder Verfälschung der Relation
  - Verfälschung boolescher Ausdrücke

Vorgehen ff.

- Auf jede Mutante  $P_i$   $i=1, \dots, n$  werden solange Tests  $t$  aus  $T$  angewendet und die Ergebnisse  $P_i(t)$  mit dem Ergebnis  $P(t)$ , bis die Mutante „tot“ ist
- Eine Mutante  $P_i$  ist lebendig (bzgl.  $T$  und  $P$ ) g.d.w.  $P_i(t) = P(t)$  für alle Tests  $t$  aus  $T$
- Eine Mutante  $P_i$  ist tot (bzgl.  $T$  und  $P$ ) g.d.w.  $P_i(t) \neq P(t)$  für mindestens einen Test  $t$  aus  $T$



## Programm P

```
int grow(int* input){
    int j, result;
    Result = 0;
    for (j =1; j < 3; j++) {
        if (input[j] > input[result])
            result = j;
    }
    return result;
}
```

Wie könnte ein Mutant  
beispielsweise aussehen ?

## Programm P

```
int grow(int* input){
    int j, result;
    Result = 0;
    for (j =1; j < 3; j++) {
        if (input[j] > input[result])
            result = j;
    }
    return result;
}
```

## Mutant P'

```
int grow(int* input){
    int j, result;
    Result = 0;
    for (j =1; j < 3; j++) {
        if (input[j] >= input[result])
            result = j;
    }
    return result;
}
```

## Resultat

- Ein Mutationsanalyse-Ergebnis, bei dem alle Mutanten tot sind, erhöht die Zuversicht, dass die Testdatenmenge T (den Mutanten) „entsprechende“ Fehler im Programm P finden würde.

## Grenzen des Verfahrens

- Der Mutant und das Originalprogramm können äquivalent sein, so dass keine Testfälle existieren, die den Mutanten töten
- Eine lebendige Mutante  $P_i$  ist zum Programm P äquivalent, d.h. für alle Eingabedaten e gilt  $P_i(e) = P(e)$
- Beispiel:

```
int max(int* input){
    int j, result;
    Result = 0;
    for (j =0; j < 3; j++) {
        if (input[j] > input[result])
            result = j;
    }
    return result;
}
```

## Grenzen des Verfahrens ff.

- Es müssen geeignet viele Mutanten erzeugt worden sein, die alle „typischen“ Fehler enthalten.
- Zweifelhaft, ob Kopplungseffekt immer gilt
  - Kopplungseffekt: Testdaten, die einfache Fehler aufdecken, entdecken auch große Fehler.

## Schwache Mutationsanalyse:

- Betrachtung einzelner Komponenten eines Mutanten und nicht des gesamten Mutanten
- D.h. es wird nur gefordert, dass direkt nach der Ausführung einer fehlerhaften Anweisung ein fehlerhafter Programmzustand vorliegt
- D.h. es wird nicht wie bei der starken Mutationsanalyse gefordert, dass sich der Fehler bis zur Programmausgabe fortpflanzt

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern:

- Beim Datenzugriff (Datenzugriffskriterium)
  - Fehlerart: Zugriff auf eine falsche Variable
  - Testdaten: Müssen so gewählt werden, dass alle Variablen im Programm vor dem Zugriff verschiedene Werte haben.
- Bei der Datenspeicherung (Datenspeicherungskriterium)
  - Fehlerart: Speicherung eines Wertes in einer falschen Variablen
  - Testdaten: Müssen so gewählt werden, dass wenn einer Variablen ein Wert zugewiesen wird, er anders als der bisherige Wert sein muss.

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern:

- Bei arithmetischen Ausdrücken
  - Ausdruck besteht aus Operanden (Variablen, Konstanten) und Operatoren (+, -, \*, /, \*\*)
  - Fehlerart: einfache additive oder multiplikative Fehler
  - Testdaten: Der betroffene (Teil-)Ausdruck muss einen Wert ungleich 0 erhalten, damit multiplikative Fehler gefunden werden. Additive Fehler wirken sich immer aus
  - Z.B. statt Ausdruck  $k*x + 1$  wird fälschlicherweise Ausdruck  $x+1$  verwendet ( $k \neq 1$ )

x	$k*x + 1$	$x+1$
0	1	1
1	$k + 1$	2

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern

- Bei arithmetischen Ausdrücken ff.
  - Fehlerart: Ein Fehler in einem Polynom oder Multinom (Polynom mit mehreren Variablen), der die Variablenmenge nicht ändert und den höchsten Exponenten im Ausdruck nicht erniedrigt
  - Testdaten:
    - Polynom vom Grad  $n$ :  $n+1$  unabhängige Testdaten
    - Z.B. statt Ausdruck  $x^2+1$  wird fälschlicherweise Ausdruck  $x+1$  verwendet
    - $(2+1)$  Testdaten finden diesen Fehler:

$x$	$x^2 + 1$	$x + 1$
0	1	1
1	2	2
2	5	3

- Multinom: hier nicht betrachtet

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern

- bei arithmetischen Relationen
  - A, B arithmetische Ausdrücke und r eines der Relationssymbole  $<$ ,  $=$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$
  - Fehlerarten:
    - Die Relation der Form „A r B“ enthält ein falsches Relationssymbol
    - „A r B“ wird statt „(A+k) r B“ verwendet
  - Testdaten: Drei Testdaten, bei denen die Differenz A-B den größten negativen Wert (knapp unter 0), den Wert 0 und den kleinsten positiven Wert annimmt
    - Z.B. statt Ausdruck  $x=y$  wird fälschlicherweise Ausdruck  $x>y$  verwendet

x	y	$x>y$	$x=y$
0	$\varepsilon$	F	F
1	1	F	T
$\varepsilon$	0	T	F

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern

- bei booleschen Ausdrücken
  - S. Bedingungsüberdeckung
- bei allgemeinen Ausdrücken
  - Fehlerart: Überflüssige Teilausdrücke (Kriterium „kürzere Ausdrücke“)
  - Testdaten: Für jeden Ausdruck und jeden seiner Teilausdrücke muss es ein Testdatum geben, das diesen Ausdruck ausführt, so dass bei der Ausführung Ausdruck und Teilausdruck verschiedene Werte annehmen (damit der überflüssige Rest erkannt wird)
    - Beispiel: für Ausdruck  $x*(x-2) + (x-1)$  sind die Testdaten  $x=0,1$  oder  $2$  nicht geeignet

$x$	$x*(x-2)$	$(x-1)$	$x*(x-2) + (x-1)$
0	0	-1	-1
1	-1	0	-1
2	0	1	1

## Schwache Mutationsanalyse zur Aufdeckung von Fehlern

- bei allgemeinen Ausdrücken ff.
  - Fehlerart: Vertauschte Teilausdrücke (Kriterium „alle kürzeren Ausdrücke“)
  - Testdaten: Für jeden Ausdruck und jeden seiner Teilausdrücke muss es ein Testdatum geben, das diesen Ausdruck ausführt und bei dessen Ausführung alle (Teil-)Ausdrücke im Programm verschiedene Werte haben (damit eine Vertauschung auffällt)

**Vorteil:** Fehler-Orientierung der Methode

**Probleme / Nachteile:**

- Man muss **viele Mutanten** erzeugen, die alle **typischen Fehler** abdecken.
- **Kopplungseffekt** müsste gelten:  
Testdaten killen **einfache** Mutanten (mit einer Abweichung)  
→ Testdaten killen auch **komplexe** Mutanten  
(mit Menge von Abweichungen)
- **Hoher Testaufwand** (ca. 50%) zum Killen der **letzten 10%** nicht äquivalenter Mutanten.
- **Regressionstests** sind aufwändig.
- **Äquivalente Mutanten** können **nicht** getötet werden und das **Äquivalenzproblem** ist **nicht entscheidbar** !  
→ **Mutationstest** kann **nicht vollständig automatisiert** werden !

Bei den White-Box Testentwurfverfahren wird gefordert, dass bestimmte Programmteile zur Ausführung kommen bzw. Bedingungen unterschiedliche Wahrheitswerte annehmen.

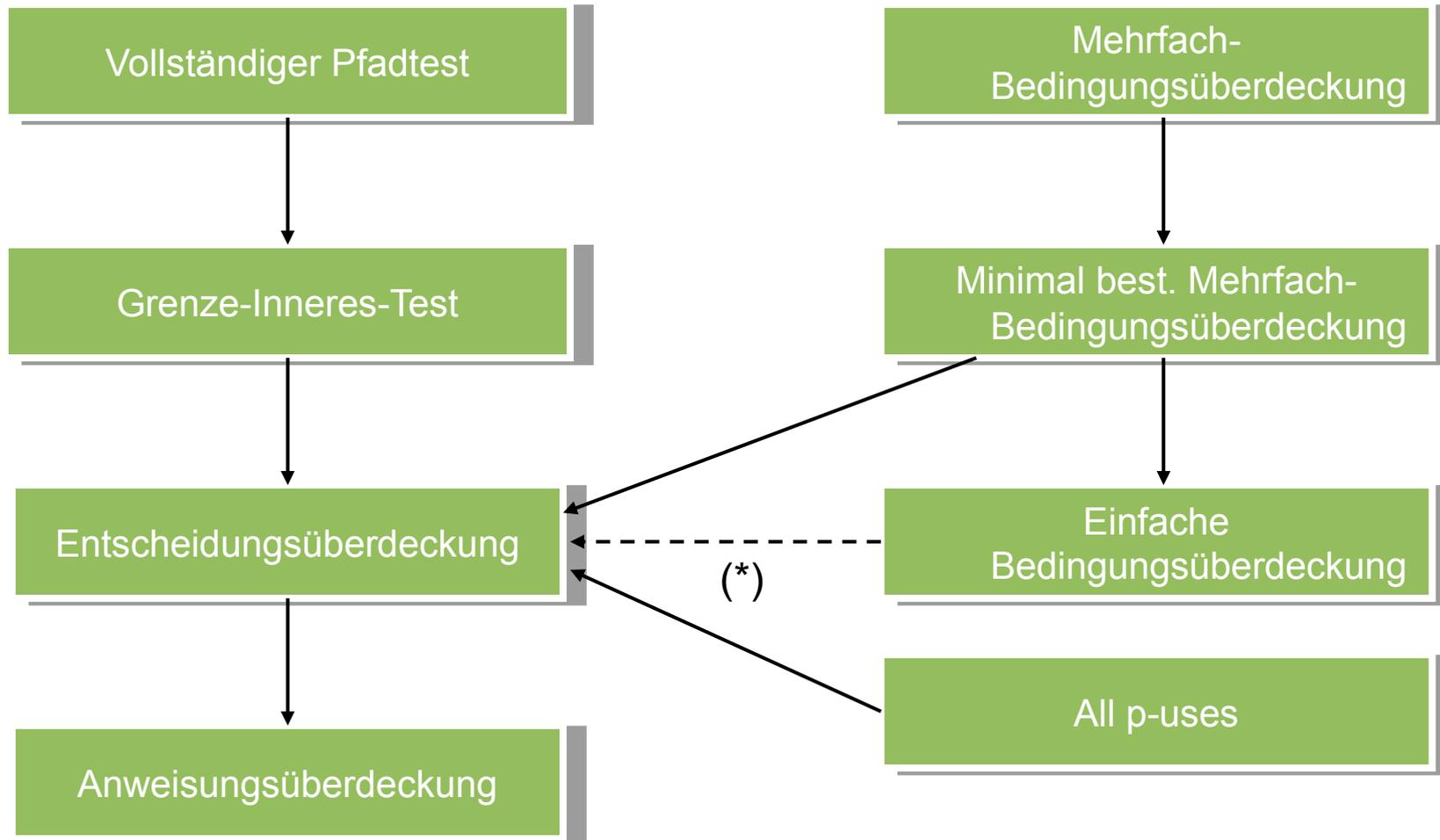
Um den Test auswerten zu können, muss ermittelt werden, welche Programmteile bereits ausgeführt wurden und welche noch nicht zur Ausführung gekommen sind.

Dazu muss das Testobjekt an strategisch wichtigen Stellen vor der Testausführung instrumentiert werden:

- Es werden zusätzlich Anweisungen wie z.B. Zähler eingebaut und mit Null initialisiert, die dann beim Durchlauf an den entsprechenden Stellen inkrementiert werden.
- Am Ende der Testläufe enthalten die Zähler die Anzahl von Durchläufen durch die jeweiligen Programmteile.
- Ist ein Zähler auf Null geblieben, so sind die entsprechenden Programmteile nicht ausgeführt worden.

Instrumentierung größerer Programme nur mit Werkzeug sinnvoll !

# Mächtigkeit der White-Box-Techniken



(\*) Bei »lazy evaluation«

# 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« Mit anderen Testentwurfsverfahren kombinieren!
Entscheidungsüberdeckung (Zweigüberdeckung, $C_1$ )	Mittel, schwankt aber stark Entdeckt ca. 30% aller Fehler und ca. 80% der Kontrollfluss-Fehler	Minimales Ausgangskriterium (Testendekriterium) 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	Zielt auf komplexe Schleifen Nur als Ergänzung
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 Kaum Tools verfügbar
Pfadüberdeckung ( $C_\infty$ )	Sehr hoch Entdeckt über 70% der Fehler	In den meisten Fällen nicht praktikabel

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

Warum werden durch eine 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 evt. fehlt.

# Zusammenfassung: White-Box Testentwurfungsverfahren

Grundlage aller White-Box Testentwurfungsverfahren ist der vorliegende Programmtext.

Abhängig von der Komplexität der Programmstruktur können das/die adäquaten Testentwurfungsverfahren ausgewählt werden.

- Z.B. Entscheidungsüberdeckung, bei if-Anweisungen mit leeren else-Zweigen.
- Anhand des Programmtextes und des/der ausgewählten Testentwurfungsverfahren wird dann die Intensität der Tests festgelegt (Ausgangskriterium/Testendekriterium).

White-Box Testentwurfungsverfahren zur Testfallermittlung eher für die unteren Teststufen.

- Beispielsweise wenig sinnvoll, erst beim Systemtest eine Überdeckung einzelner Anweisungen / Entscheidungen erreichen zu wollen, da dort nur noch komplette Systemfunktionen, nicht einzelne Anweisungen oder Bedingungen geprüft werden.

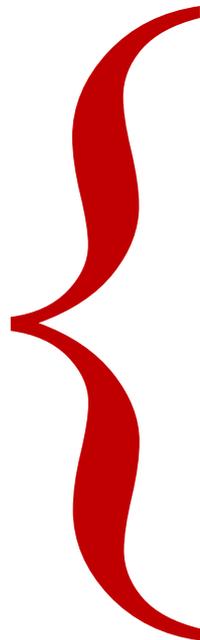
Problem: »Nicht vorhandener Programmcode« bleibt unberücksichtigt.

- Anforderungen, die übersehen und daher nicht realisiert wurden, werden durch die White-Box Testentwurfungsverfahren nicht aufgedeckt.
- Nur die Anforderungen, die auch im Programm umgesetzt worden sind, können bei den White-Box Testentwurfungsverfahren überprüft werden.

Zur Instrumentierung Werkzeug verwenden.



## 2.5 White-Box- Test



---

Idee der White-Box Testentwurfsverfahren

---

Kontrollflussbasierter Test

---

Datenflussbasierter Test

---

Test der Bedingungen

---

Weitere White-Box Testentwurfsverfahren

---

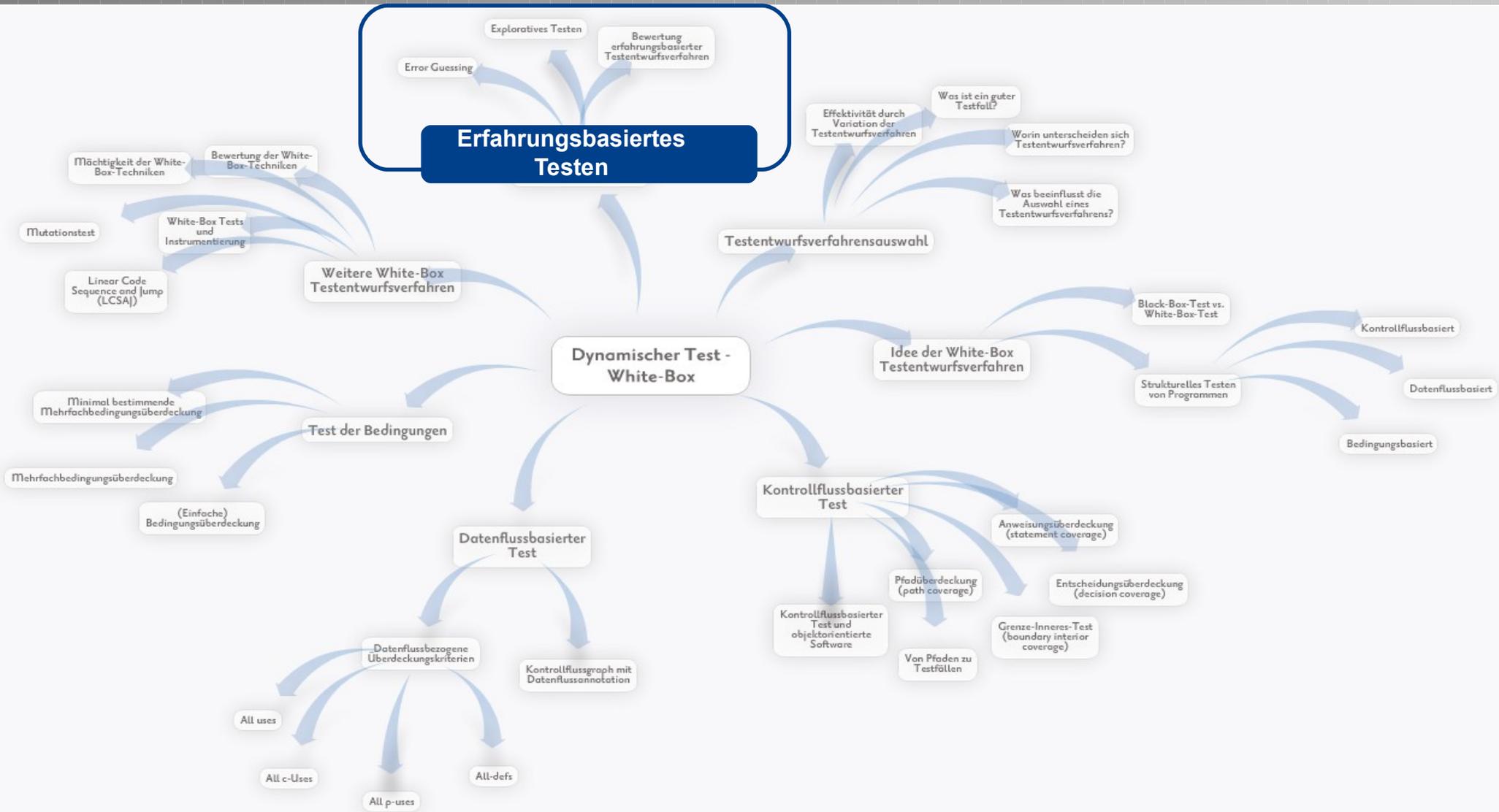
Erfahrungsbasiertes Testen

---

Dynamischer Test: Testentwurfsvorgangsauswahl  
und Zusammenfassung

---

# Dynamischer Test – Erfahrungsbasiertes Testen



Der Erfolg systematischer Testentwurfsverfahren hängt stark von der Qualität der Dokumente ab, auf deren Grundlage Testfälle definiert werden.

- Qualität bedeutet:  
Lesbarkeit, Testbarkeit, Vollständigkeit, Eindeutigkeit, Widerspruchsfreiheit, ...
- Oft sind auch Formalisierungsgrad und Granularität nicht adäquat, um formale bzw. systematische Testentwurfsverfahren anzuwenden.

Grundidee erfahrungsbasierter Testentwurfsverfahren:

- Erfahrung/Wissen der Tester wird bei der Definition von Testfällen einbezogen.
- Kompensation der mangelnden Qualität der Eingangsdokumentation durch die Erfahrung und durch das Wissen der Tester.
- Definition von Testfällen, die sehr schwer durch systematische Testentwurfsverfahren abgeleitet werden können.

Erfahrungsbasierte Testentwurfsverfahren eignen sich sehr gut als Ergänzung der bisher vorgestellten systematischen Testentwurfsverfahren.

Beispiele für erfahrungsbasierte Testentwurfsverfahren sind Error Guessing (intuitives Testen) und exploratives Testen.

1. Testentwurfsverfahren, bei dem Erfahrung und Wissen der Tester genutzt werden, um vorherzusagen, welche Fehlerzustände in einer Komponente oder einem System aufgrund der Fehlhandlungen vorkommen, und um Testfälle so abzuleiten, dass diese Fehler aufgedeckt werden.
2. Methode und Herleitung oder Auswahl der Testfälle, gestützt auf Erfahrung und Wissen des Testers.

Möglicherweise am weitesten verbreitete Testentwurfsverfahren (oft auch intuitive Testfallermittlung genannt).

Tests werden durch Können und Intuition des Testers, und aus seiner Erfahrung mit ähnlichen Applikationen und Technologien, abgeleitet.

Systematische Vorgehensweise:

- Erstelle einen Fehlerkatalog mit möglichen Fehlerzuständen und Fehlerwirkungen.
- Entwerfe Testfälle, die auf diese Fehler abzielen.

Error Guessing kann zur Unterstützung systematischer Testentwurfungsverfahren eingesetzt werden. Ermittelt Tests, die von systematischen Testentwurfungsverfahren nicht / schwer erfasst werden.

Vorsicht: Diese Testentwurfungsverfahren können äußerst unterschiedliche Grade von Effizienz erreichen, da sie abhängig von der Erfahrung des Testers sind.

Strukturierte Herangehensweise:

- Liste möglicher Fehler erstellen und dann Testfälle entwerfen, die auf diese Fehler abzielen.
- Diese Liste der Fehlerzustände und Fehlerwirkungen kann aufgrund von Erfahrung, verfügbaren Daten über Fehlerzustände und Fehlerwirkungen sowie von Allgemeinwissen darüber, warum Software sich falsch verhalten kann, erstellt werden.
- Liste kann auch für die Entwickler von großem Nutzen sein, da vor der Implementierung schon auf eventuelle Probleme und Schwierigkeiten hingewiesen wird, die dann während der Implementierung berücksichtigt werden können und somit der Fehlervermeidung dienen.

Liste mit möglichen Fehlern und fehlerverdächtigen Situationen.

- Umfangreiches Erfahrungswissen oft nur in den Köpfen erfahrener Tester.
- Erfahrungen über immer wieder auftretende Fehler werden vermerkt und stehen somit allen Testern zur Verfügung.

Zahlreiche publizierte Fehlerlisten:

- Cem Kaner, Jack Falk, & Hung Quoc Nguyen, Testing Computer Software: enthält eine Liste von über 400 Fehlern
- James A. Whittaker: How to Break Software: A Practical Guide to Testing: enthält mögliche „Angriffe“, die zu einem Fehler führen können  
James A. Whittaker: How to Break Software Security: enthält mögliche „Sicherheits-Angriffe“
- Mike Andrews and James A. Whittaker: How to Break Web Software: Functional and Security Testing of Web Applications and Web Services
- Greg Hogg and Gary McGraw: Exploiting Software: How to Break Code
- ...

Problem: Aktualität, Fehlerlisten nicht domänen- und produktspezifisch.

Anpassung allgemeiner Fehlerlisten und die Definition und Pflege eigener Fehlerkataloge immer notwendig.

Informelles Testen, bei dem keine Testvorbereitung stattfindet und keine erkennbare Testentwurfsverfahren verwendet werden.

Es werden keine erwarteten Ergebnisse vorab spezifiziert und die Testdurchführung erfolgt mehr oder minder willkürlich.

Gleichzeitige Testfallanalyse und Testfallentwurf, Testrealisierung und -durchführung, Testprotokollierung und insbesondere auch Lernen.

- Grundlage eine Test-Charta, der die Testziele und mögliche Testideen zu entnehmen sind.
- Durchführung innerhalb festgelegter Zeitfenster.
- Besonders gut geeignet, wenn es nur wenig oder ungeeignete Spezifikationen gibt, unter hohem Zeitdruck getestet wird, oder wenn andere, formalere Testentwurfsverfahren unterstützt oder ergänzt werden sollen.
- Exploratives Testen kann auch zur Überprüfung des Testprozesses dienen, um etwaige Lücken aufzuspüren, falls dabei doch noch Fehler gefunden werden.

## Eigenschaften:

- **Paralleler Testentwurf und Testdurchführung:** Entwicklung von Strategien zur Untersuchung des Produktes, Definition und Ausführung von Testfällen.
- **Überprüfbare Ergebnisse:** Ergebnisse/Erfahrungen aus vorherigen Tests werden dokumentiert und ausgewertet und beeinflussen die nächsten Tests.
- **Exploration:** »Kennenlernen« des Produktes (Funktionen, verarbeitete Daten, unreife Produktbereiche, ...).
- **Heuristiken:** Leitfäden und Faustregeln, erleichtern die Auswahl von Testfällen.

## Aktivitäten:

- Testvorbereitung
- Testdurchführung
- Dokumentation
- Testauswertung

## Einheiten:

Abgeschlossene Teile des Produktes

Testaufwand 1-2 Tage

---

## Arbeitseinheit:

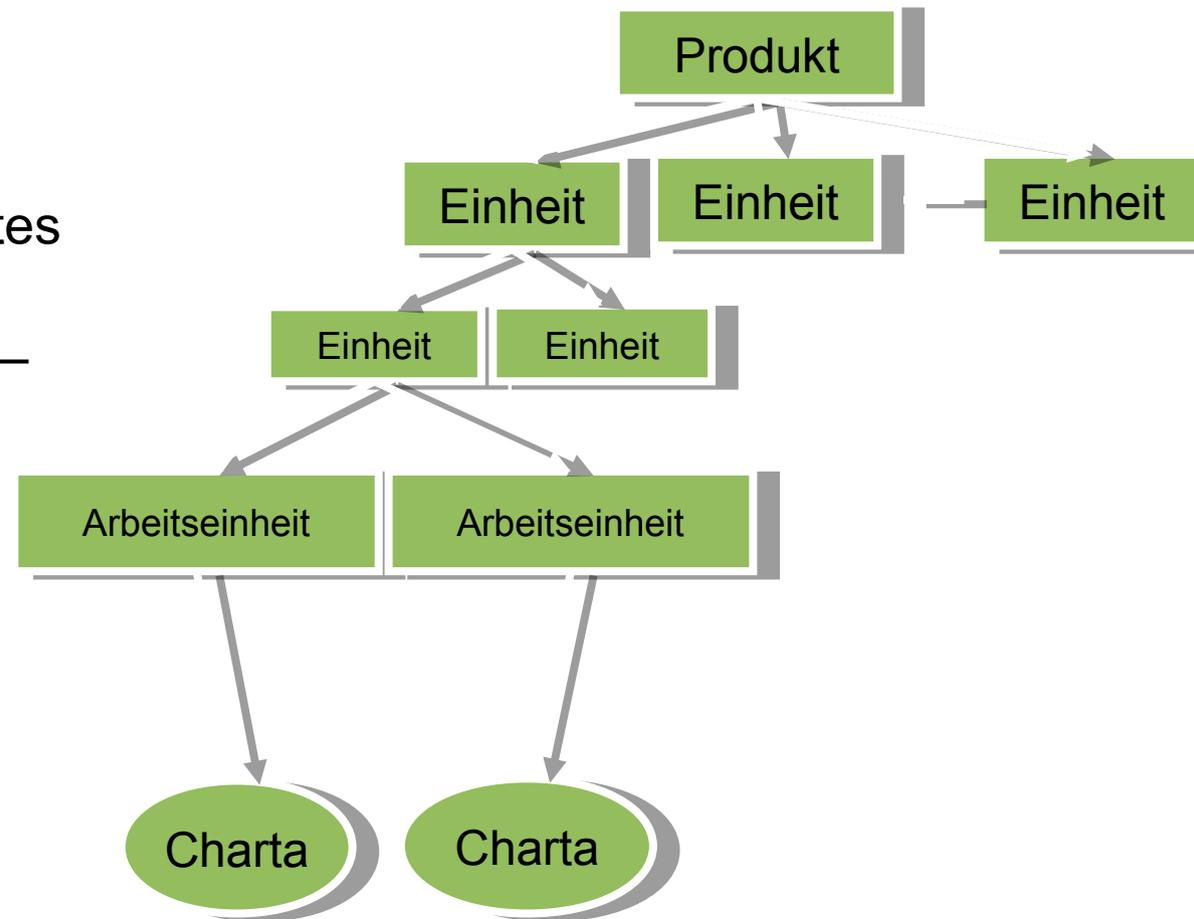
Abarbeiten innerhalb einer Sitzung  
„an einem Stück“ möglich

Testaufwand 1-2 Stunden

---

## Charta:

Gibt Ziel und Aufgabe für  
die Sitzung vor



Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach

Gibt klares Ziel und Aufgabe einer Sitzung vor:

- Warum soll die entsprechende Einheit getestet werden ?
- Was soll getestet werden ?
- Wie getestet werden soll (Testentwurfsverfahren) ?
- Welche Probleme sollen adressiert werden ?

Zusatzinformationen, die enthalten sein können:

- Werkzeuge, die benutzt werden können
- Explorationsstrategien
- Dokumente, die einbezogen werden müssen

Kein detaillierter Testplan !

Am Anfang: Allgemeine Test-Chartas zur Überprüfung und zum Kennenlernen der Grundfunktionalität.

- Beispiel: “Analysiere Grafik-Einfüge-Funktion eines Textverarbeitungsprogramms”

Später: Detaillierte Test-Chartas zum detaillierten Test der entsprechenden Funktionalität, umfassen auch Test von Qualitätsanforderungen wie beispielsweise Performanz oder Usability.

- Beispiel: “Teste das Einfügen eines Cliparts”, “Teste das Einfügen eines Bildes aus einer Datei”, ...

Zeitlich begrenzte Sitzung.

- Abgeschlossene Arbeitseinheit.
- Zielgerichtet: Durchführung der Tests, die in einer Charta „geplant“ wurden.
- Dauer: Kurz genug um flexibel auf Testergebnisse reagieren zu können, lang genug um einen abgeschlossenen Teilbereich des Produktes zu testen und um aussagekräftige Auswertungen zu erzeugen.
- Kurz: 60 Minuten, Normal: 90 Minuten, Lang: 120 Minuten
- Ohne Unterbrechung (z.B. Email, Telefon, usw..).
- Überprüfbar (die Ergebnisse jeder Sitzung werden protokolliert).

Während der Sitzung:

- Ziel nicht aus den Augen verlieren.
- Notizen während der Durchführung: Um später nachvollziehen zu können, welche Tests warum durchgeführt wurden und um Testauswertung zu ermöglichen.

Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach

## Dokumentation:

Chartas geben Überblick über das, was getestet werden soll.

Sitzungsprotokolle erlauben Auswertung und Fortschrittsüberwachung:

- Beginn der Sitzung
- Charta, getestete Einheit(en)
- Tester
- Dauer gesamt, für Vorbereitung, Testdurchführung, Fehleranalyse, Dokumentation
- Testdaten
- Notizen (Was habe ich gemacht? Warum? Hinweise, Fragen, Anomalien, aufgetretene Schwierigkeiten, ...)
- Fehlerbericht (adäquate Granularität, um Tests wiederholen zu können)
- ...

## Auswertung:

- Nach jeder Sitzung werden die Ergebnisse der Sitzung ausgewertet und mit dem Testmanager besprochen.
- Auswertungsergebnisse und Erfahrungen der aktuellen Sitzung fließen in die nächste Sitzung ein.

Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach

**Explorationsstrategien:** Helfen, das Produkt zu analysieren und Testfälle auszuwählen.

Zahlreiche »Muster«:

- Intuition (zufälliges Ausprobieren, Suche nach ähnlichen Fehlern, Funktionen, ...)
- Modelle (mental oder explizit modelliert, UML, ...)
- Beispiele (realistische Benutzungsszenarien, User Stories, ...)
- Interferenz (Störung des „normalen“ Ablaufs, Hardwarefehler, ...)
- Fehleranalyse (Schritte weglassen, alternative Schritte ausführen, Konfiguration ändern, ...)
- Gruppenwissen (pair testing, brainstorming, ...)
- Aktives Lesen (Mehrdeutigkeiten, Unvollständigkeiten, ...)

**Heuristiken:** Systematische Gewinnung neuer Erkenntnisse auf Basis von Erfahrung. Sehr nützlich beim explorativen Testen. Achtung: Keine Garantie, dass sie zur richtigen Lösung führen. Daher: Kontext und Gründe hinter der Heuristik verstehen bevor sie angewendet wird.

Beispiele: Heuristiken zur Identifikation von potenziellen Fehlern.

- Neue Funktionalität ist fehleranfälliger als reife Funktionalität
- Geänderte Funktionalität ist fehleranfälliger als reife Funktionalität
- Späte Änderungen verursachen Fehler
- Teste Grenzwerte
- ...

## Vorteile:

- Ergänzt systematische Testentwurfsverfahren.
- Deckt Fehler auf, die schwer durch systematische Testentwurfsverfahren aufgedeckt werden können.
- Kurzfristig einsetzbar, erfordert keine lange Vorplanung.
- Fördert Kreativität und Spontaneität.
- In Gruppen ergeben sich Synergien zwischen erfahrenen und unerfahrenen Testern.
- Anwendbar, wenn wenig Dokumentation oder Domänenwissen vorhanden (Kennenlernen des Produktes, »Exploration«).

## Schwächen:

- Verfall in bestimmte Denkmuster verhindert das Aufdecken wichtiger Probleme.
- Vom Wissen und Erfahrung der Tester abhängig.
- Keine Automatisierbarkeit.

Erfahrungsbasierte Testentwurfsverfahren lassen sich eigentlich nicht eindeutig den Black- oder White-Box Testentwurfsverfahren zuordnen, da weder die Anforderungen noch der Programmtext ausschließliche Grundlage für die Überlegungen und Prüfungen sind.

Anwendungsbereich eher in den höheren Teststufen, da auf den niedrigeren meist ausreichende Informationen für die anderen Testentwurfsverfahren zur Verfügung stehen, wie beispielsweise der Programmtext oder eine detaillierte Spezifikation.

Erfahrungsbasierte Testentwurfsverfahren unabhängig vom Testobjekt oft mit gutem Erfolg einsetzbar.

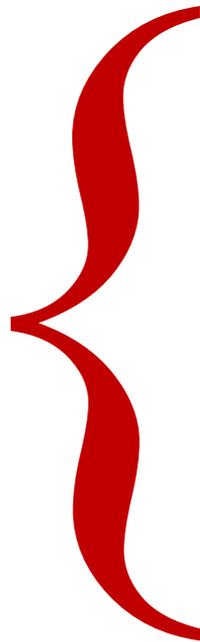
## **Probleme:**

- Vor- und Nachbedingungen, erwartete Ausgaben und erwartetes Verhalten des Testobjektes bei intuitiven Testfällen oft schwierig festzulegen.
- Keine Messungen der Intensität oder Vollständigkeit der Testfallermittlung möglich.
- Kein Ausgangskriterium (Endekriterium) wie bei den systematischen Testentwurfsverfahren.
- Existiert ein Fehlerkatalog, kann eine gewisse Vollständigkeit überprüft werden.

**Fazit:** Erfahrungsbasierte Testentwurfsverfahren nicht als primäre Testentwurfsverfahren einsetzen, sondern zur Abrundung und Unterstützung der methodischen Testentwurfsverfahren.



## 2.5 White-Box- Test



---

Idee der White-Box Testentwurfsverfahren

---

Kontrollflussbasierter Test

---

Datenflussbasierter Test

---

Test der Bedingungen

---

Weitere White-Box Testentwurfsverfahren

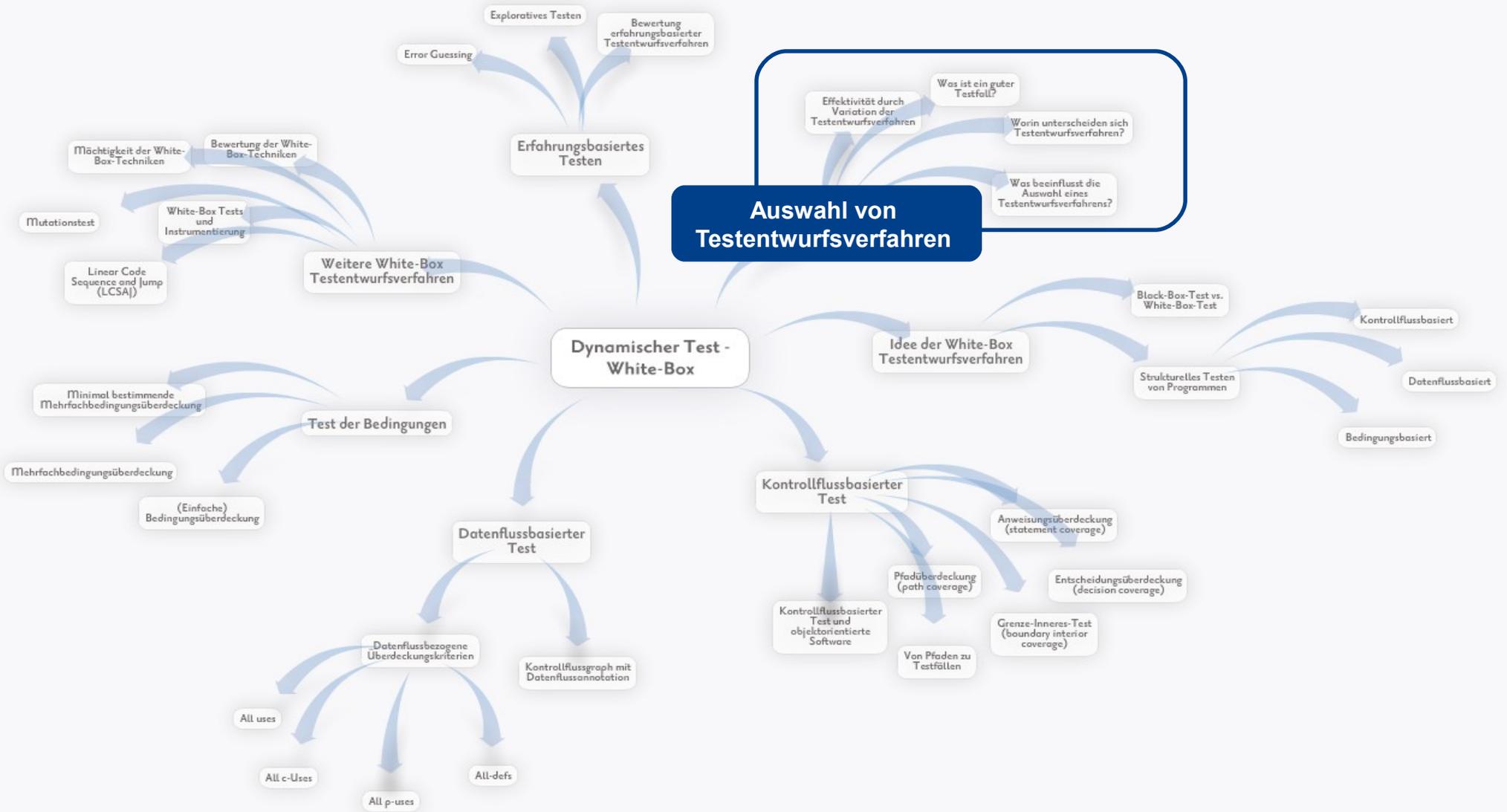
---

Erfahrungsbasiertes Testen

---

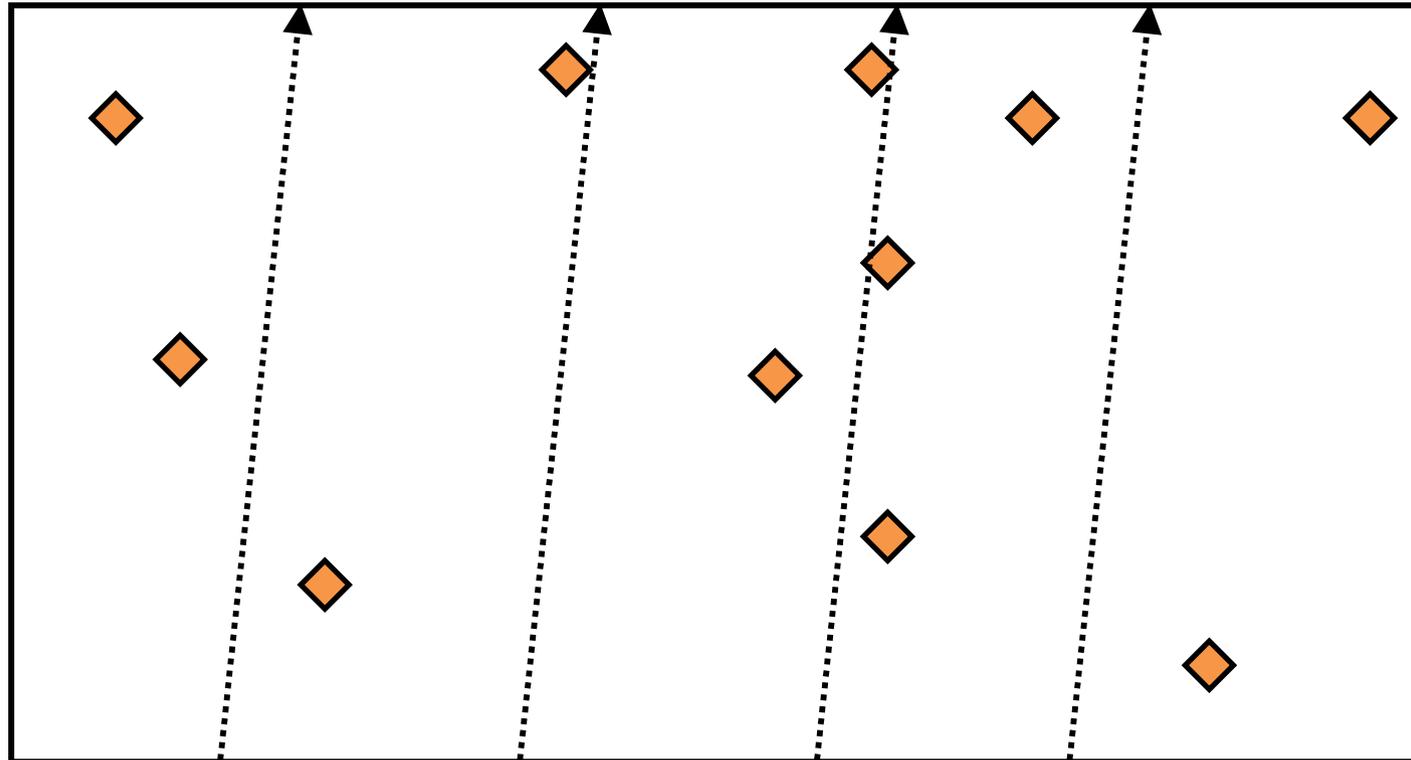
Dynamischer Test: Testentwurfverfahrensauswahl  
und Zusammenfassung

# Dynamischer Test – Auswahl von Testentwurfungsverfahren



# Die Gefahr einer einseitigen Teststrategie (1)

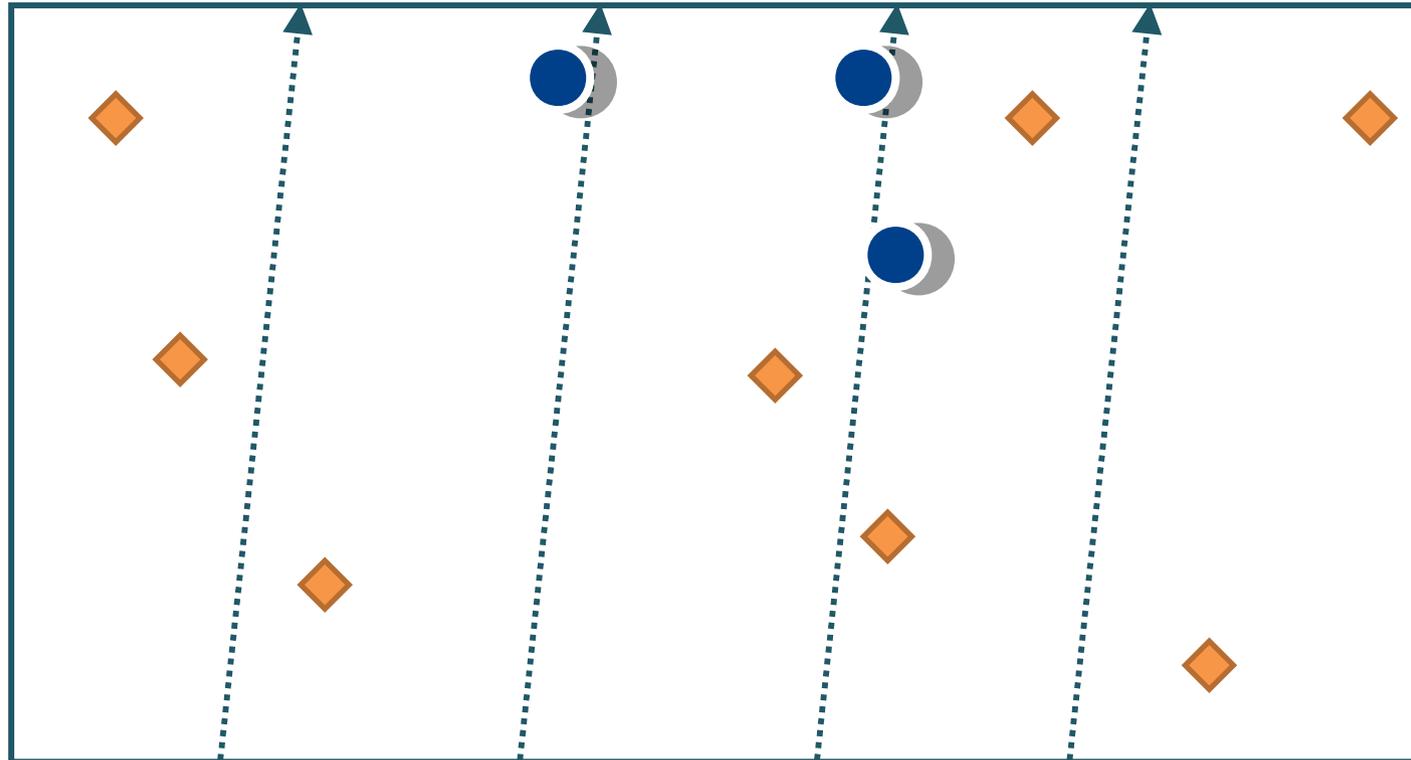
Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach



◆ Fehlerzustand («Mine«)

# Die Gefahr einer einseitigen Teststrategie (2)

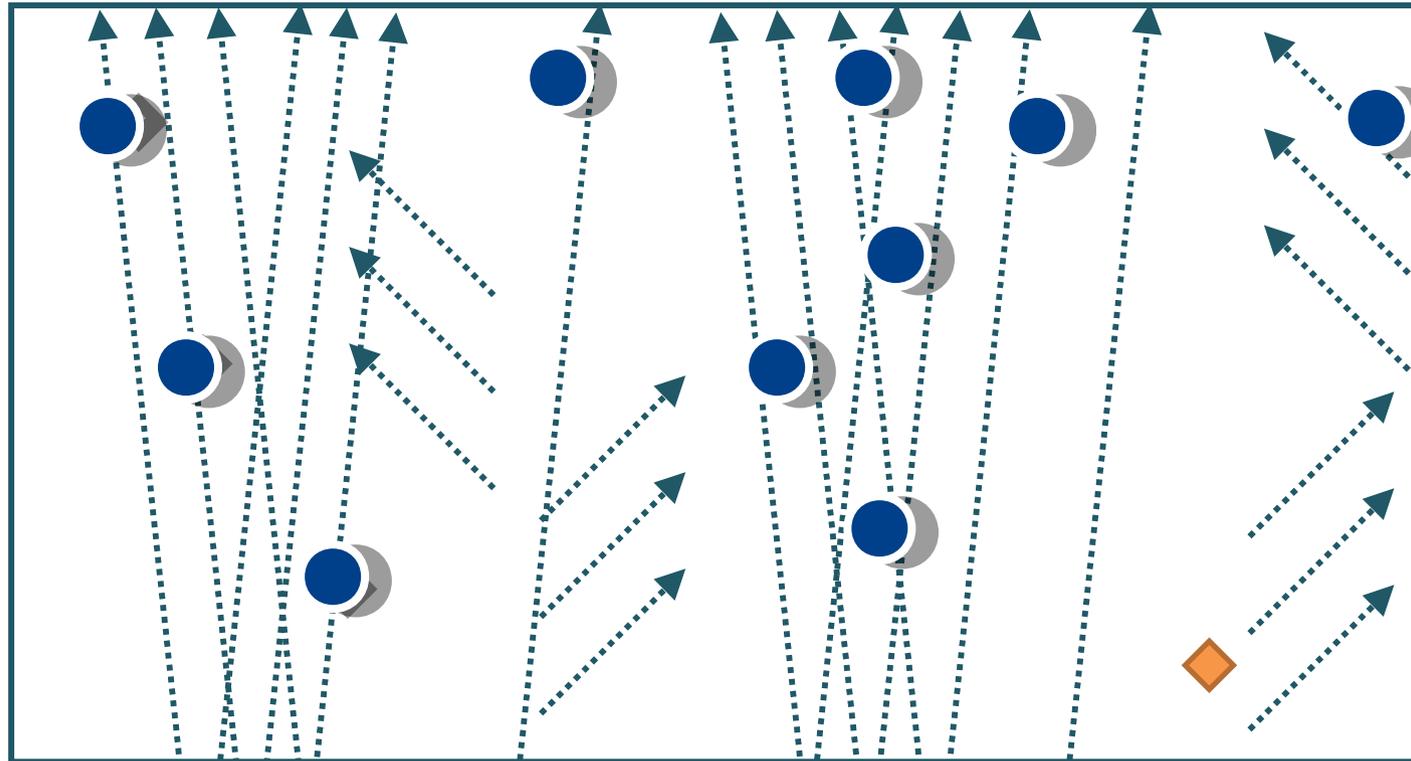
Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach



◆ Fehlerzustand (»Mine«) ● behoben (zur Wirkung gebracht)

# Effektivität durch Variation der Testentwurfsverfahren

Nach: Rapid Software Testing,  
copyright © 1996-2002 James Bach



◆ Fehlerzustand (»Mine«) ● behoben (zur Wirkung gebracht)

Attribute eines guten Testfalls:

- Mächtig – Testfall deckt Fehlerwirkung auf, falls diese existiert.
- Valide – Testfall deckt tatsächliche Fehlerwirkung auf.
- Relevant – Testfall deckt Fehlerwirkung auf, die für die Stakeholder (z.B. Kunden) relevant sind.
- Glaubwürdig.
- Repräsentativ.
- Nicht-redundant – Fehlerwirkungen werden nicht durch einen anderen Testfall abgedeckt.
- Ausführbar – Testfall kann, so wie er spezifiziert ist, ausgeführt werden.
- Wartbar, wiederholbar, einfach evaluierbar, adäquate Komplexität, geringe Kosten, ...

Ausprägung der einzelnen Attribute hängt von verwendeter Technik ab. Beispiele:

- Anwendungsfallbasierter Test – fokussiert auf Validität, Relevanz, Glaubwürdigkeit, Komplexität, weniger auf Wartbarkeit.
- Äquivalenzklassenbildung – nicht-redundant, wiederholbar, einfach wartbar. Keine Fokussierung auf Repräsentativität oder Glaubwürdigkeit.

# Worin unterscheiden sich Testentwurfsverfahren?



<b>TESTSTART</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, ...

# »Parameter« eines Testentwurfsverfahrens

Jedes Testentwurfsverfahren adressiert einen oder mehrere dieser »Parameter«, wobei die restlichen »Parameter« offen gelassen werden.

»Parameter«:

- Testart, Teststufe, Tester, Abdeckung, potenzielle Fehler, Artefakte, Charakteristika des Produktes (Domäne, Technologie)
- Verfügbare Dokumentation und ihre Qualität
- Regulatorische Anforderungen / Standards, Kunden- / Vertragsanforderungen
- Projekt- und Produktrisiken
- Testziele, angestrebter Automatisierungsgrad
- Projektfaktoren:  
Qualifikation der Tester, Erfahrung, Zeit und Geld, Softwareentwicklungsmodell

Kombination von Testentwurfsverfahren mit jeweils unterschiedlichem Fokus möglich.

Einige Testentwurfsverfahren sind in einem bestimmten Kontext für bestimmte Situationen und Teststufen besser geeignet. Andere Testentwurfsverfahren sind in allen Teststufen gleichermaßen einsetzbar.

Jedem Testentwurfsverfahren liegt eine Fehlerannahme (auch als Fehlermodell oder Fehlerhypothese bezeichnet) zu Grunde, d.h. die Annahme über mögliche Fehlerwirkungen, die durch die Anwendung des entsprechenden Testentwurfsverfahrens aufgedeckt werden

- Folge: Unterschiedliche Testentwurfsverfahren finden unterschiedliche Arten von Fehlerwirkungen.
- Empfehlung: Verwendung unterschiedlicher Testentwurfsverfahren !

# Zusammenfassung: Dynamischer Test (1)

Ziel: mit wenig Aufwand ausreichend unterschiedliche Testfälle zu erzeugen, die mit einer gewissen Wahrscheinlichkeit vorhandene Fehlerzustände zur Wirkungen bringen.

Testentwurfsverfahren zur Erstellung der Testfälle passend zum Testobjekt auswählen.

Auf jeden Fall ausreichende Prüfung der Funktionalität des Testobjektes gewährleisten.

Bei der Aufstellung aller Testfälle erwartete Ergebnisse und Reaktionen des Testobjektes mit vermerken, so dass eine Prüfung der Funktionalität bei jeder Auswertung der durchgeführten Testfälle erfolgt.

Bei jedem Testobjekt Äquivalenzklassenbildung in Kombination mit der Grenzwertanalyse zur Erstellung der Testfälle einsetzen.

Bei der Ausführung dieser Testfälle die codebasierte Überdeckung messen und die nach der Durchführung aller Testfälle der Äquivalenzklassenbildung und Grenzwertanalyse bereits erreichte Anweisungs- oder Entscheidungsüberdeckung ermitteln.

Haben unterschiedliche Zustände einen Einfluss auf den Ablauf innerhalb des Testobjektes, muss ein zustandsbasierter Test durchgeführt werden.



Bisher nicht ausgeführte Teile des Testobjektes werden dann gezielt einem der White-Box Testentwurfsverfahren unterzogen.

- Je nach Kritikalität und Beschaffenheit des Testobjektes entsprechend aufwändiges White-Box Testentwurfsverfahren wählen.
- Als minimales Kriterium die Entscheidungsüberdeckung verwenden.
- Bei den White-Box Testentwurfsverfahren soll die Struktur des Testobjektes die Grundlage bei der Auswahl der Testentwurfsverfahren sein - sind z.B. komplexe Bedingungen im Testobjekt vorhanden, so ist die minimal bestimmende Mehrfachbedingungsüberdeckung das adäquate Testentwurfsverfahren, um fehlerhafte Bedingungen zu erkennen.
- Bei den Überdeckungsmessungen darauf achten, das Schleifen auch mehr als einmal wiederholt werden.
- Bei kritischen Systemteilen muss die Prüfung der Schleifen anhand von entsprechenden Methoden erfolgen.
- Die Pfadüberdeckung ist als eher theoretisches Maß anzusehen und hat auf Grund des riesigen Aufwands für die Praxis keine Bedeutung.

White-Box Testentwurfsverfahren sinnvollerweise auf den unteren Teststufen einsetzen; auf den oberen Teststufen sind die Black-Box Testentwurfsverfahren die adäquaten Methoden.

Auf die erfahrungsbasierte Ermittlung der Testfälle als Ergänzung sollte nicht verzichtet werden – es ist immer sinnvoll, die Erfahrungen der Tester zu nutzen, um weitere Fehler aufzudecken.

Testen umfasst immer die Kombination von unterschiedlichen Testentwurfsverfahren, da es kein Testentwurfsverfahren gibt, das alle Aspekte, die beim Testen zu berücksichtigen sind, gleich gut abdeckt.

- Die Auswahl der Testentwurfsverfahren und die Intensität der Durchführung sind anhand der Kritikalität und dem zu erwartenden Risiko im Fehlerfall festzulegen.

# Folgende Fragen sollten Sie jetzt beantworten können

- Was bedeutet der Begriff Anweisungsüberdeckung ?
- Worin unterscheiden sich Anweisungs- und Entscheidungsüberdeckung ?
- Nach welcher Formel wird die erreichte Anweisungsüberdeckung berechnet ?
- Wozu dient die Instrumentierung ?
- Exkurs: Worauf zielt die Bedingungsüberdeckung ab ?
- Exkurs: Worin unterscheiden sich die einfache Bedingungsüberdeckung und die Mehrfachbedingungsüberdeckung ?
- Was ist unter erfahrungsbasierten (exploratives) Testen zu verstehen ?



- Die Grundidee der White-Box Testentwurfungsverfahren (strukturorientierte Testentwurfungsverfahren) erläutern können
- White-Box Testentwurfungsverfahren zur Ermittlung von Testfällen charakterisieren und voneinander abgrenzen können
- Die Anweisungsüberdeckung und die Entscheidungsüberdeckung für das kontrollflussbasierte Testen kennen und auf einfache Beispiele anwenden können
- Die Ideen der Grenze-Inneres-Überdeckung und des datenflussbasierten Testens kennen und erläutern können
- Unterschiedliche Arten der Bedingungsüberdeckung kennen und auf einfache Beispiele anwenden können
- Weitere Arten von White-Box Testentwurfungsverfahren aufzählen können
- Error Guessing und exploratives Testen als erfahrungsbasierte Testentwurfungsverfahren kennen und charakterisieren können
- Dynamische Tests bez. ihres Einsatzbereiches erläutern und mit anderen Testentwurfungsverfahren zu einer Teststrategie kombinieren können

# Diese Begriffe sollten Sie kennen...

