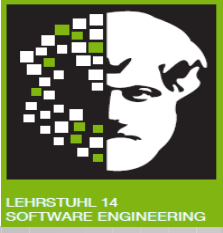


Willkommen zur Vorlesung

Methodische Grundlagen des Software-Engineering

im Sommersemester 2012

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

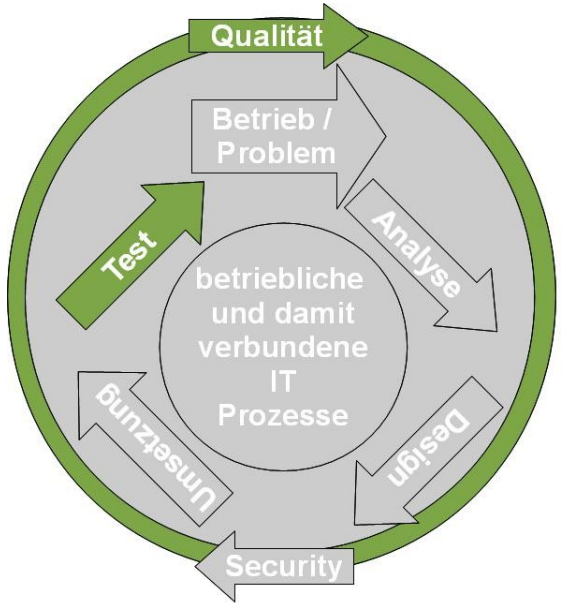
4.5 White-Box-Test

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)

V 2.0 / 2011

Einordnung
White-Box-Test

- Geschäfts-Prozesse
- Qualitätsmanagement
- Testen
  - Einführung
  - Grundlagen Softwaretesten
  - Testen im Softwarelebenszyklus
  - Statischer Test
  - Black-Box-Test
  - White-Box-Test
  - Test-Management
  - Testwerkzeuge
  - Fuzzing
- Sicheres Software Design

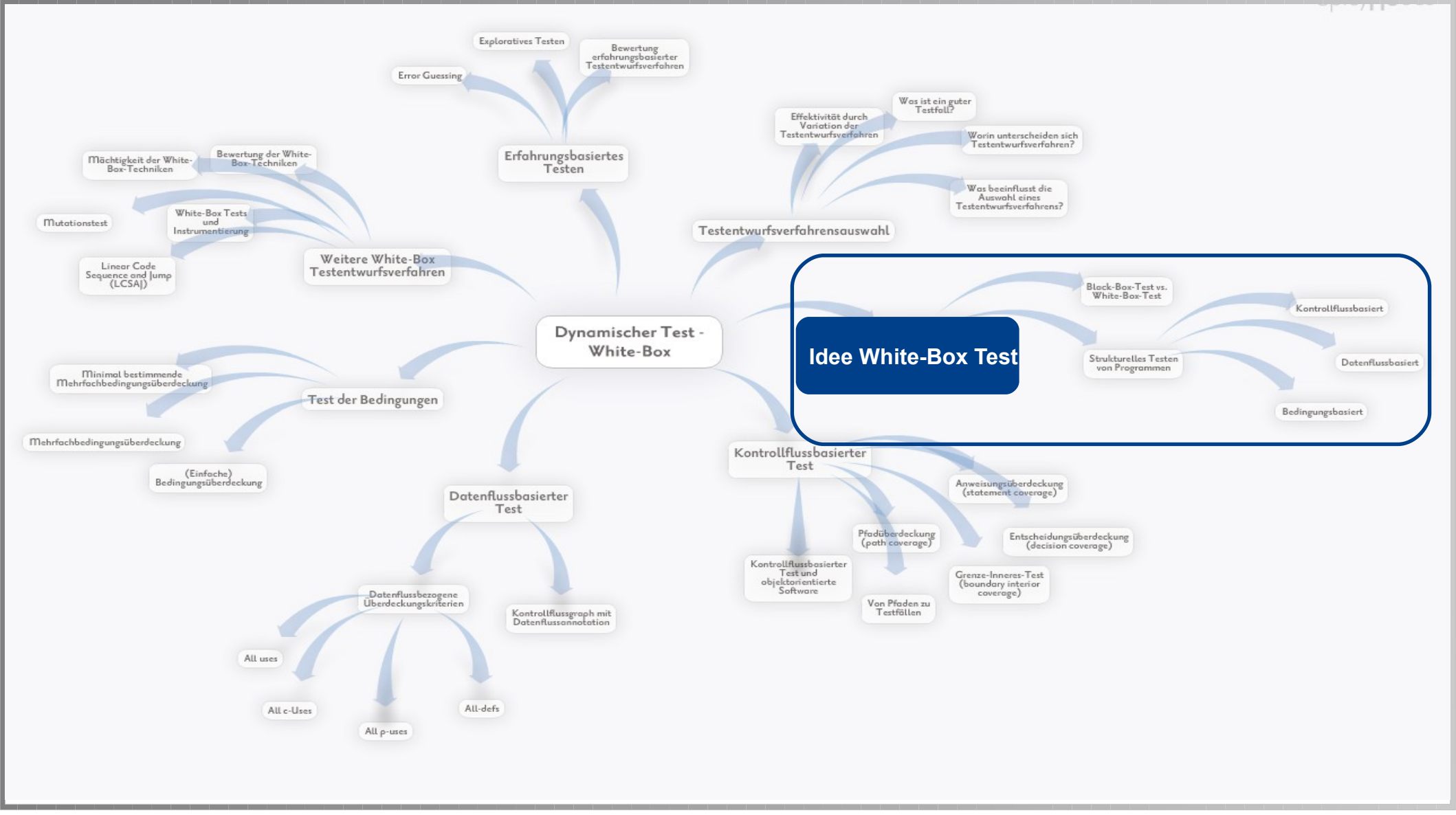


4.5 White-Box-Test

4.5 White-Box-Test

- Idee der White-Box Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- Test der Bedingungen
- Weitere White-Box Testentwurfsverfahren
- Erfahrungsbasiertes Testen
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Dynamischer Test – White Box



Zur Erinnerung: Black-Box Test vs. White-Box Test

Black-Box Test

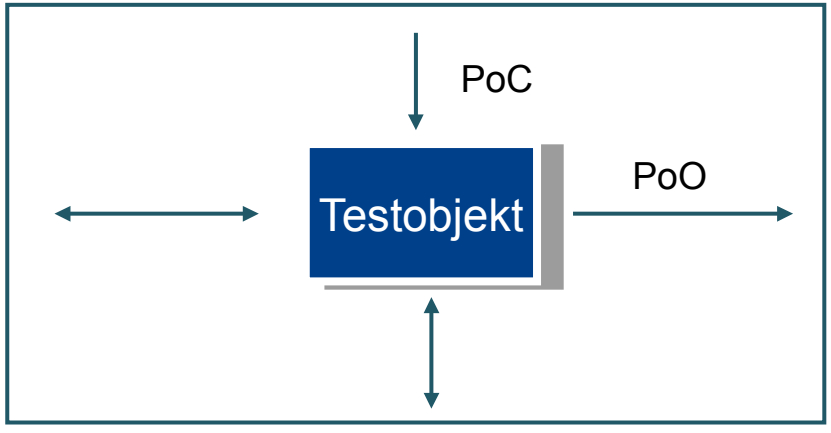
Eingabewerte

Ohne Kenntnis der Programmlogik abgeleitet

Testobjekt

Istergebnis

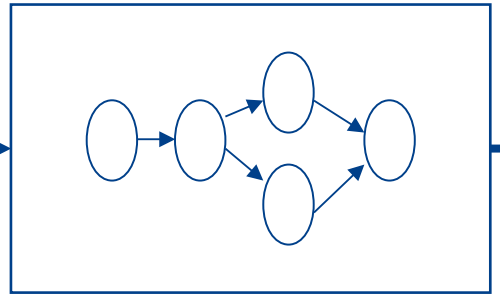
PoC = Point of Control
PoO = Point of Observation



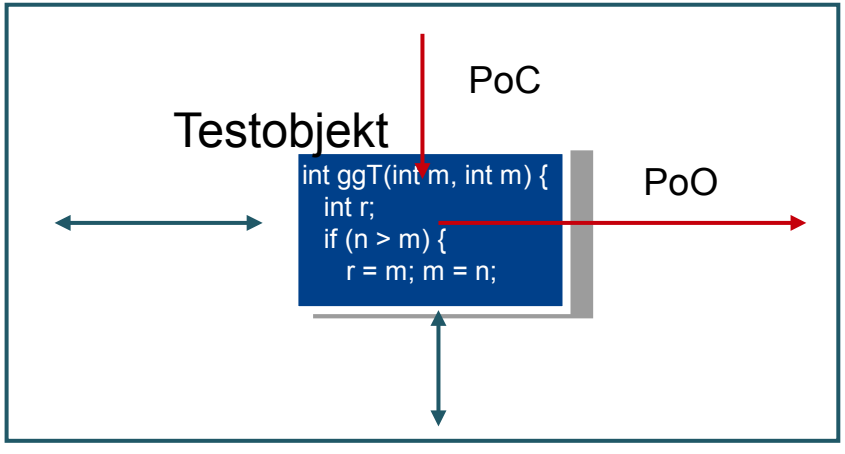
White-Box Test

Eingabewerte

Mit Kenntnis der Programmlogik abgeleitet



Istergebnis



White-Box-Testentwurfsverfahren: Begriffe

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

Alle Testentwurfsverfahren, 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 Testentwurfsverfahren genannt.

Überblick strukturelles Testen: Ansätze und Überdeckungskriterien

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∞-Überdeckung, alle Pfade)

Datenflussbasiert:

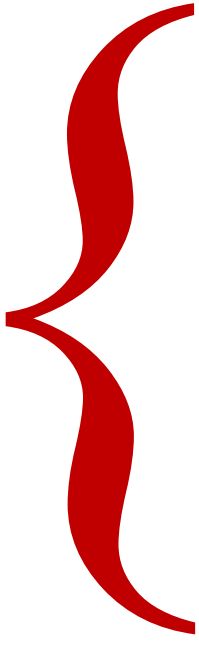
- Alle Definitionen (all defs)
- Alle Definition-Benutzung-Paare (all def-uses)

Bedingungsbasiert

- Einfache Bedingungsüberdeckung
- Mehrfachbedingungsüberdeckung
- Minimal bestimmende Mehrfachbedingungsüberdeckung

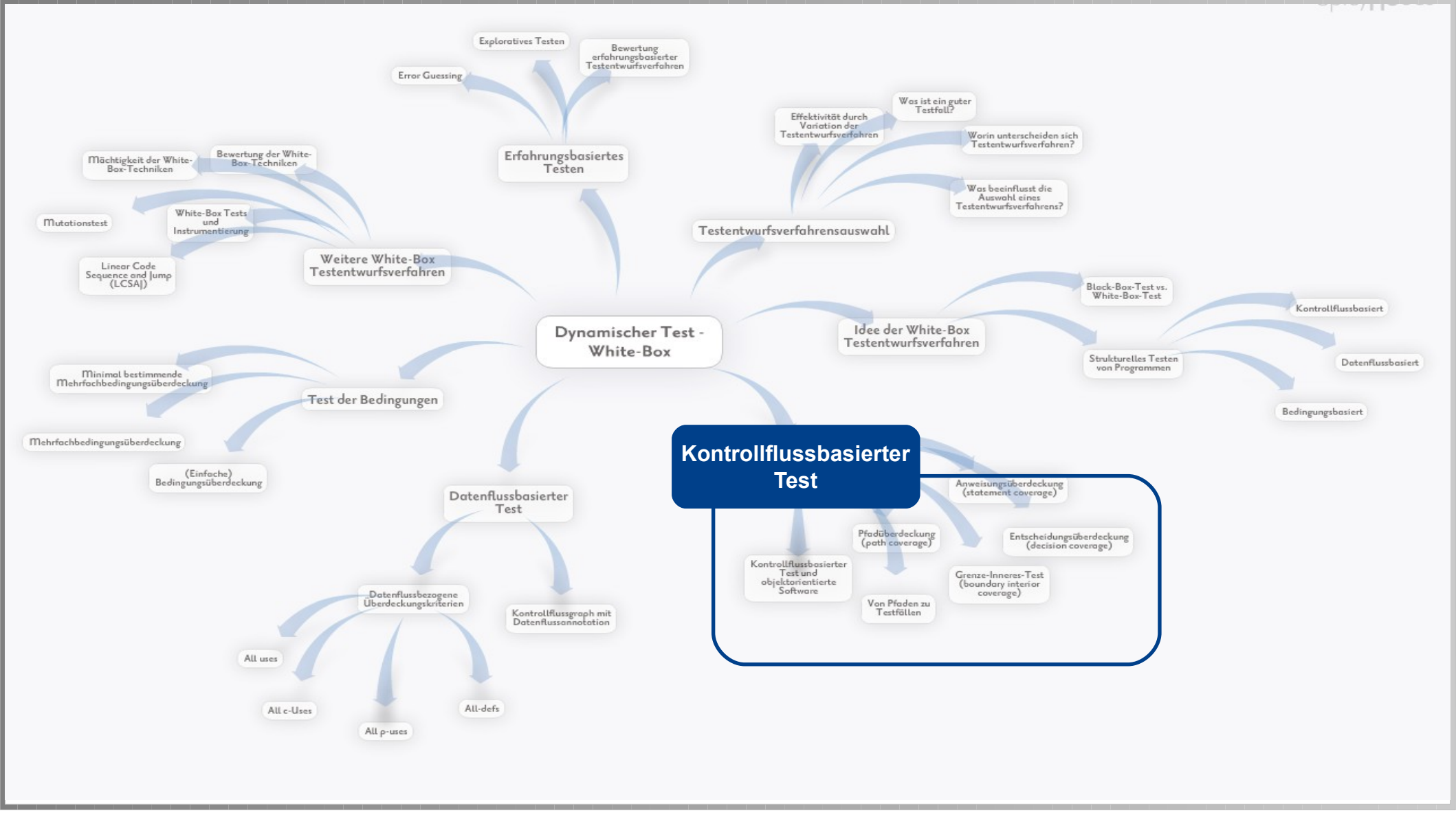
4.5 White-Box-Test

4.5 White-Box-Test



- Idee der White-Box Testentwurfsverfahren
- **Kontrollflussbasierter Test**
- Datenflussbasierter Test
- Test der Bedingungen
- Weitere White-Box Testentwurfsverfahren
- Erfahrungsbasiertes Testen
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Kontrollflussbasierter Test



Zur Erinnerung: Funktion `ggT()`

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

ggT(4,8)=4; ggT(5,8)=1; ggT(15,35)=5; ggT(0,0)=0 [per Konvention]

Zur Erinnerung: Spezifikation in UML / 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)
… )
```

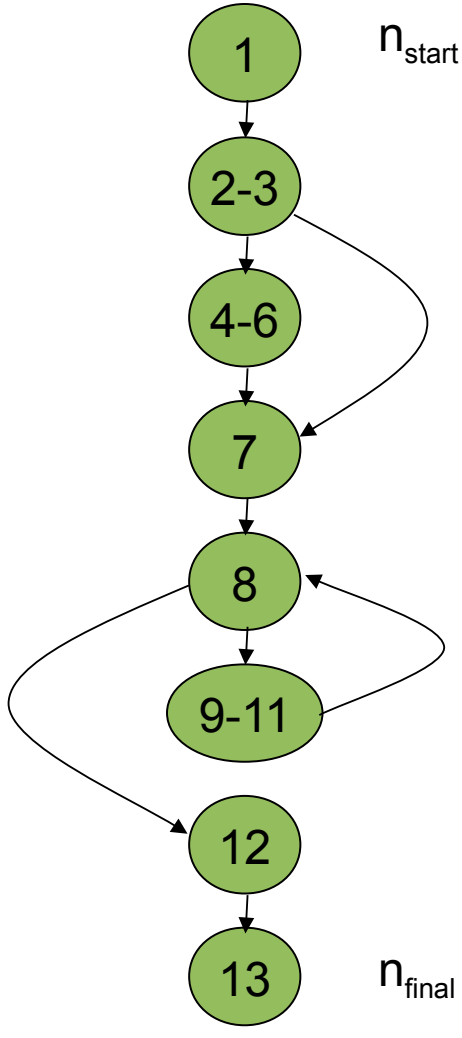
Zur Erinnerung: Kontrollflussgraph von `ggT()`


```
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) {
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. }
```

Block

Block

Block



Kontrollflussbezogene Überdeckungskriterien

- 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∞-Überdeckung, alle Pfade*)

Anweisungsüberdeckung

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

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

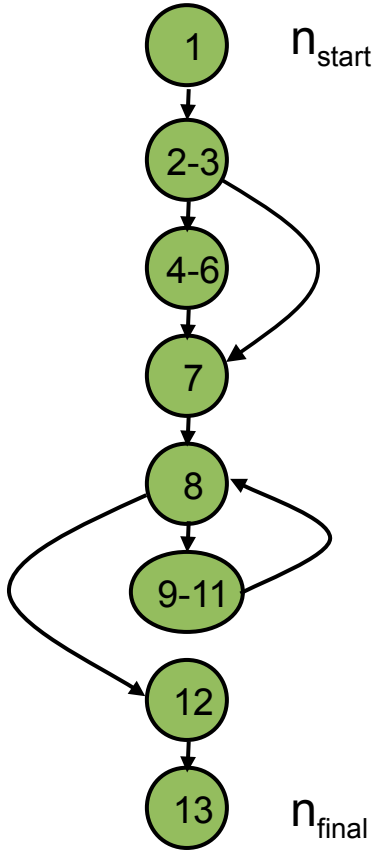
Jeder Testfall wird anhand eines Pfads durch den Kontrollflussgraphen bestimmt.

- 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 = Überdeckung).

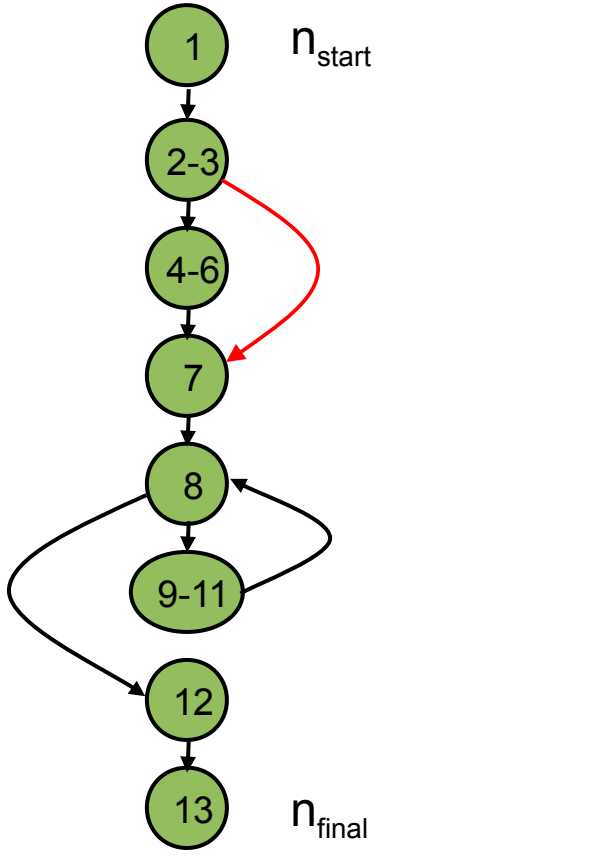
Beispiel: Anweisungsüberdeckung für `ggt()`



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

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

Diskussion der Anweisungsüberdeckung

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).

Entscheidungstest (1)

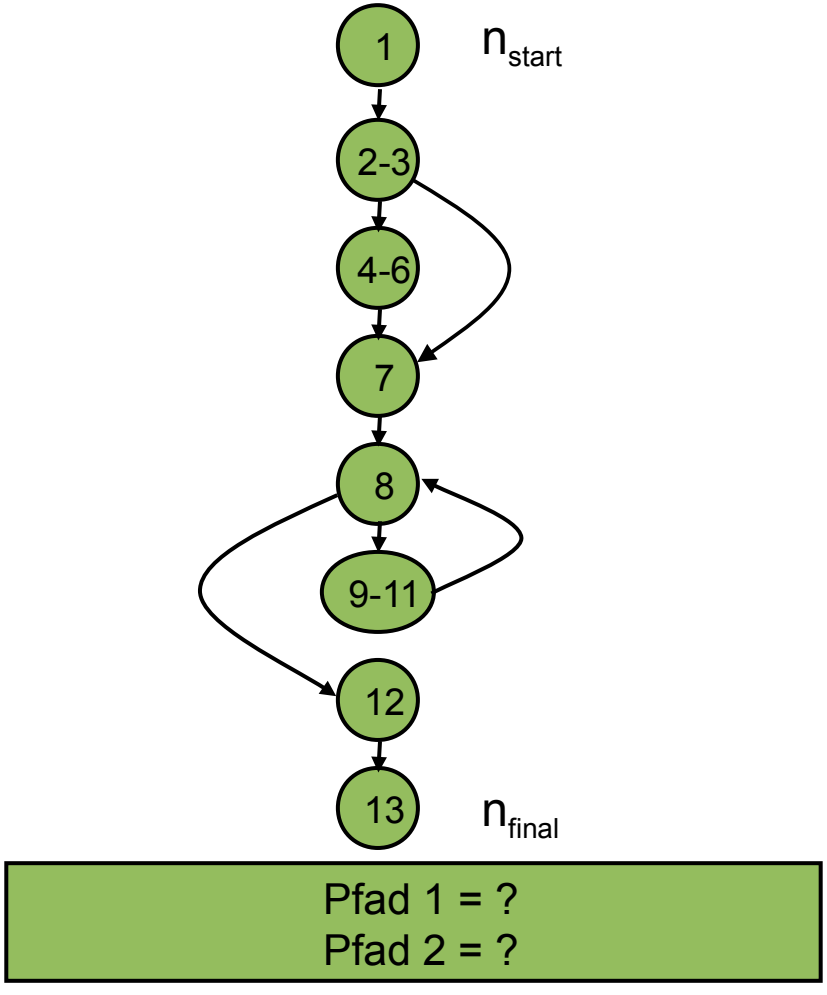
Entscheidungstest: Ein White-Box-Testentwurfsverfahren, bei dem Testfälle im Hinblick auf die Überdeckung der Entscheidungsausgänge entworfen werden.

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.

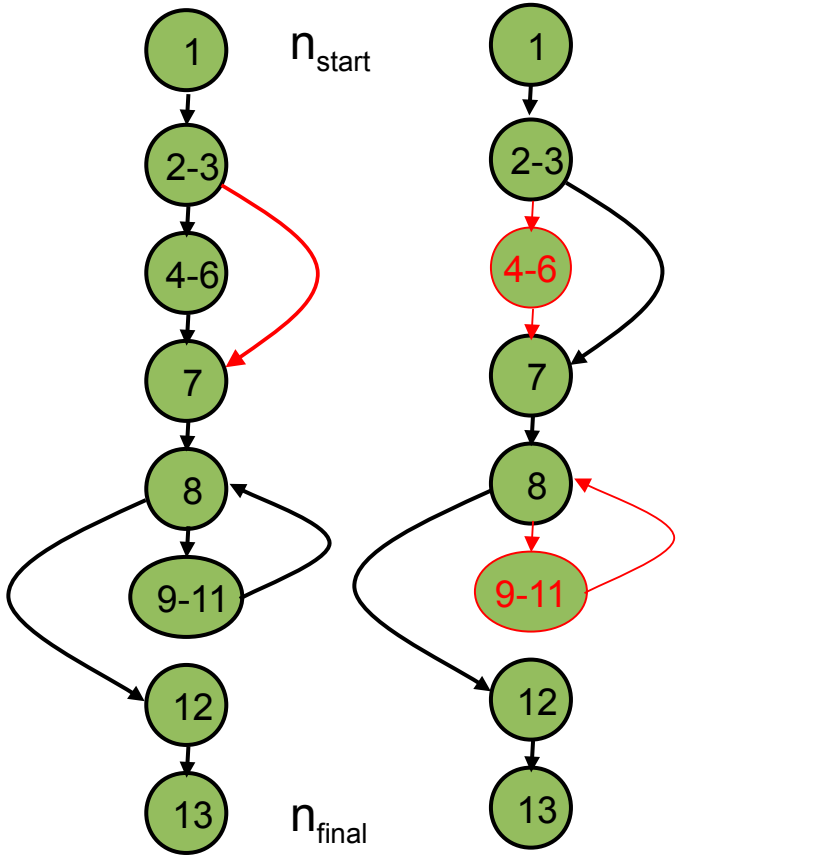
Beispiel: Entscheidungsüberdeckung für `ggt()`




```
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. }
```

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

Entscheidung

Entscheidungstest (2)

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.

Diskussion der Entscheidungsüberdeckung

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.

Grenze-Inneres-Überdeckung

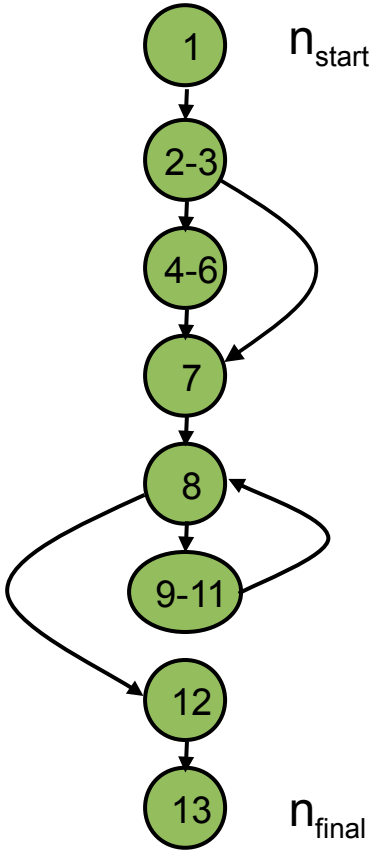
Grenze-Inneres-Überdeckung: Dynamisches, kontrollflussbasiertes Testentwurfsverfahren (gi) fordert, dass jede Schleife in mindestens einem Testfall:

- 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.

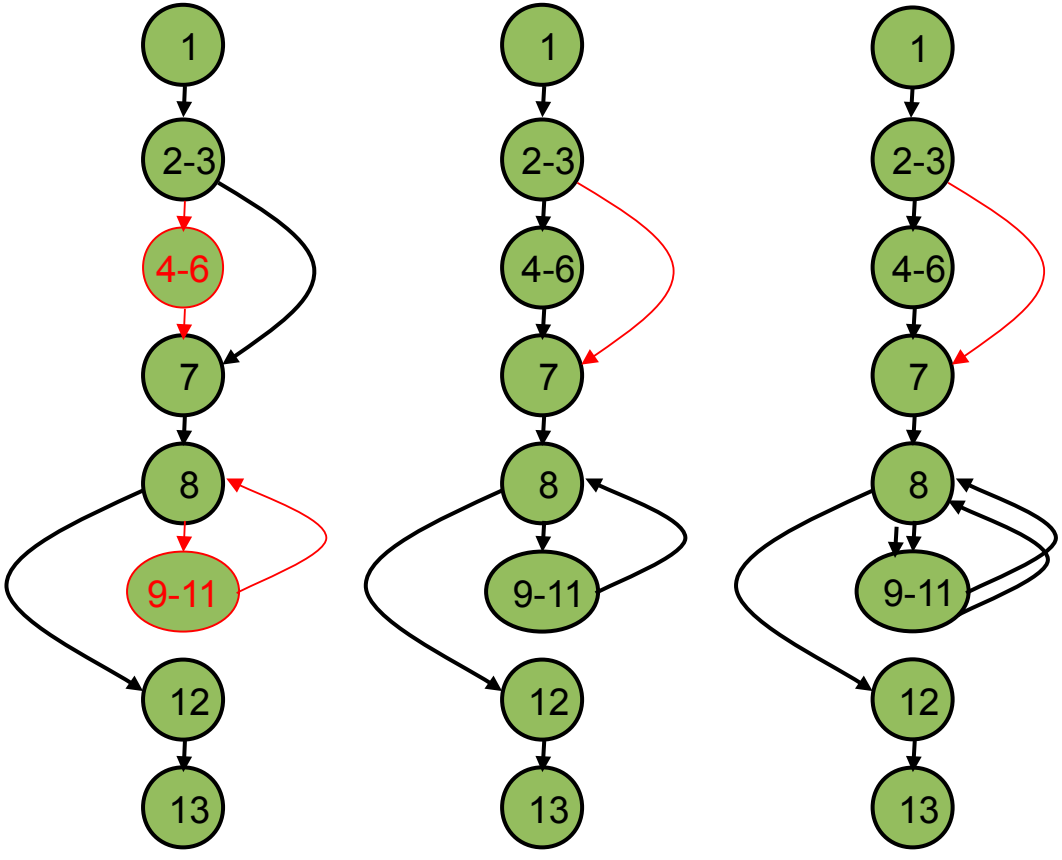
Grenze-Inneres-Überdeckung für `ggt()`



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. }
```

Grenze-Inneres-Überdeckung für `ggt()`



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. }
```

Diskussion der Grenze-Inneres-Überdeckung

- 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 Schliefen.
  - 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.

Pfadüberdeckung

Dynamisches, kontrollflussbasiertes Testentwurfsverfahren, fordert, dass jeder Pfad im Kontrollflussgraphen mindestens einmal ausgeführt wird. Auch als C∞-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.

Diskussion

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 ?

Diskussion

Frage:

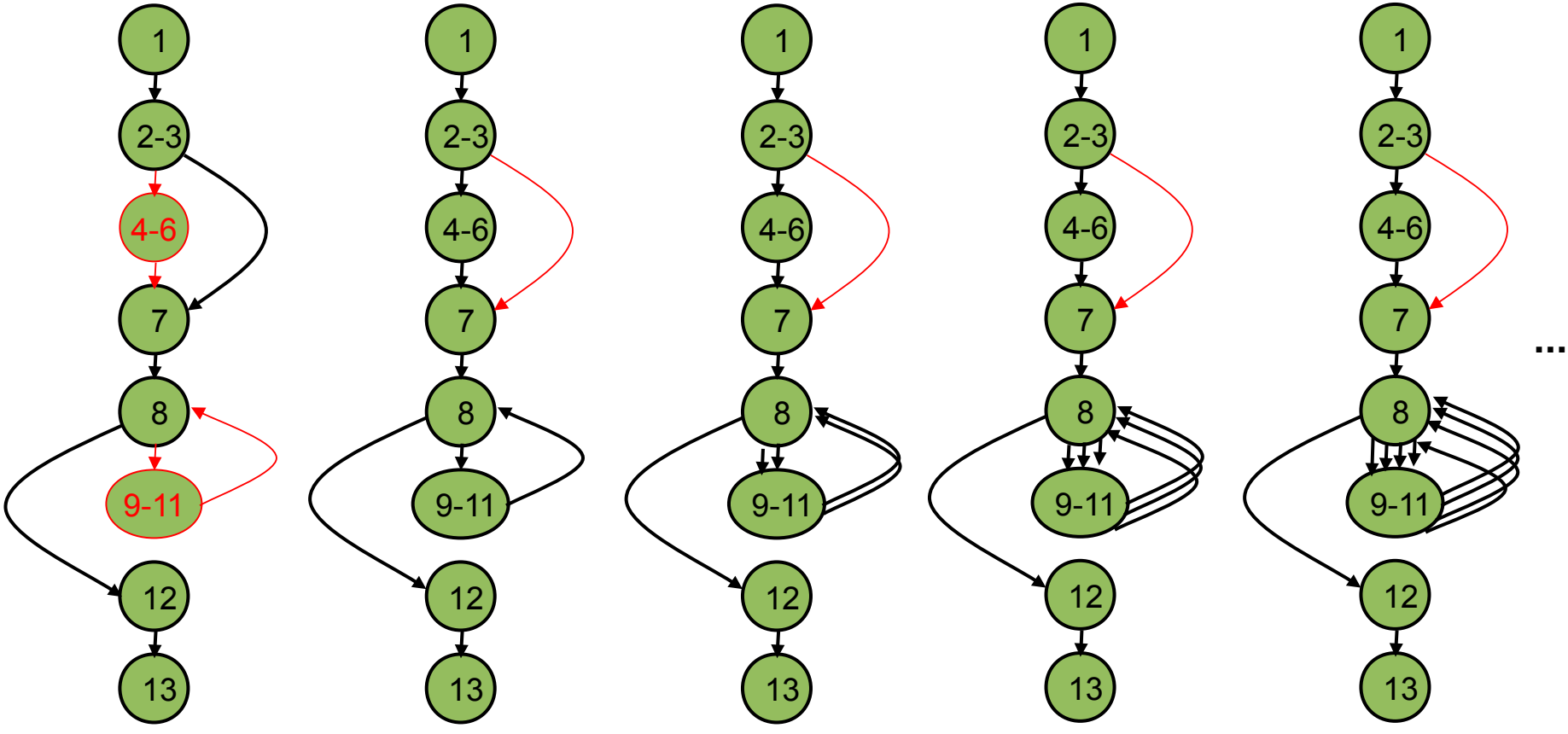
1) Können Sie sich ein Programm in Pseudo-Code vorstellen, zu dem keine obere Grenze angegeben werden kann ?
2) Lässt sich dieses Programm auf einem heutzutage üblichen realen Computer implementieren ?

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 `ggt()`

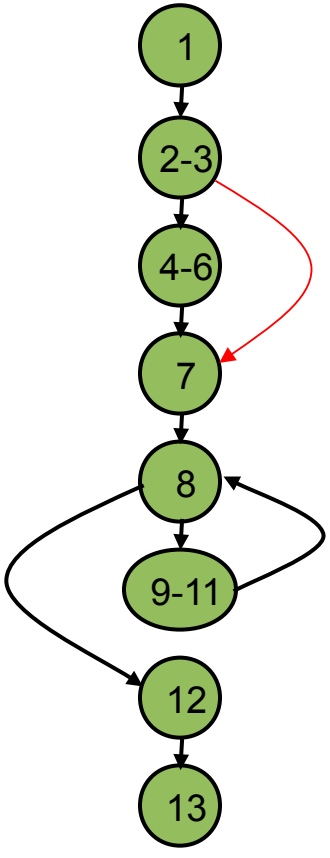


Frage: Welche obere Grenze lässt sich hier angeben ?

Von Pfaden zu Testfällen

- 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

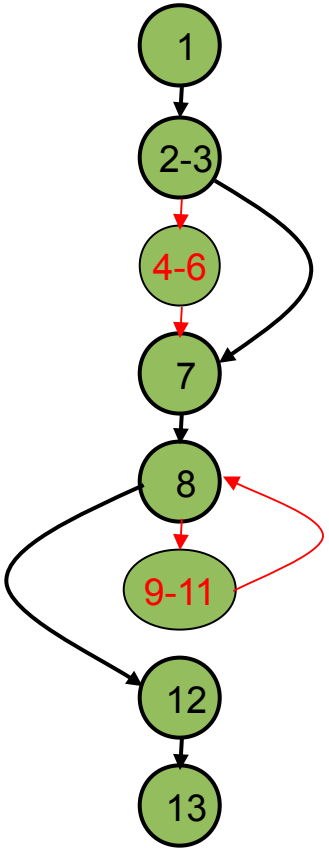


Pfad: (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

Logischer Testfall: { $n > m \wedge n \bmod m \neq 0 \wedge$
$n \bmod (n \bmod m) = 0$; 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: {$n \leq m \wedge m \bmod n = 0$; 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.      r = m % n;
8.      while (r != 0) {
9.          m = n;
10.         n = r;
11.         r = m % n;
        }
12.     return n;
13. }
```

m	n	r
4	4	4

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

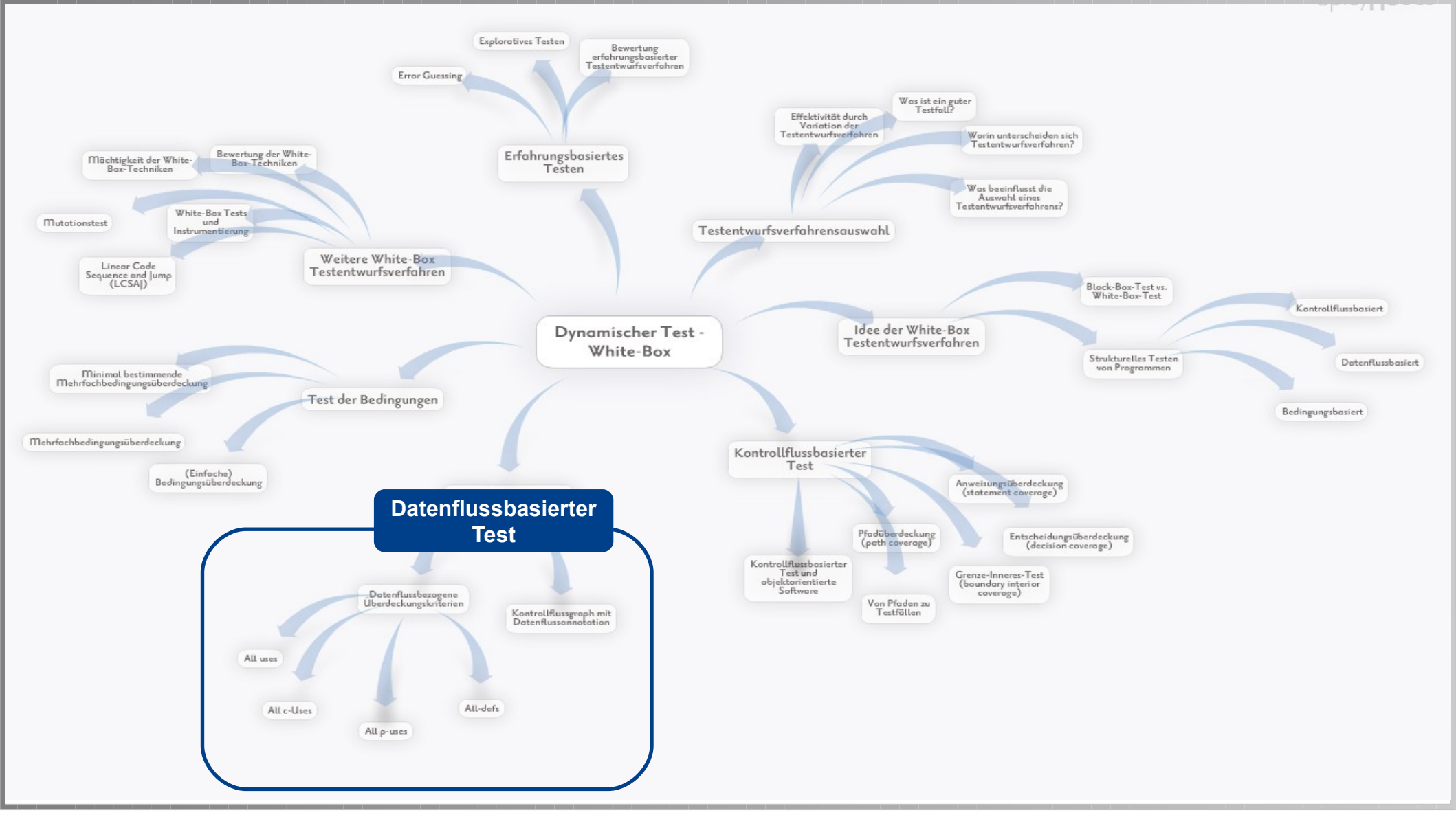
- Komplexität objektorientierter Systemen meist in den Beziehungen zwischen den Klassen bzw. den Interaktionen zwischen Objekten verborgen.
- Methoden in den Klassen sind normalerweise wenig umfangreich und von geringer Komplexität (McCabe: 2-4).
- Die geforderte Anweisungs- und/oder Entscheidungsüberdeckung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

4.5 White-Box-Test

4.5 White-Box-Test

Datenflussbasierter Test



Datenflussbasierter Test

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, def (`r`)
- Benutzung in Ausdrücken, zustandserhaltend
  - z.B. `r = m mod n` oder `r = op1(m,n)`
  - Computational use, c-use (`m`, `n`) und def (`r`)
- Benutzung in Bedingungen, zustandserhaltend
  - z.B. `while (r != 0)` oder `if (r != 0)`
  - Predicative use, p-use (`r`)

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

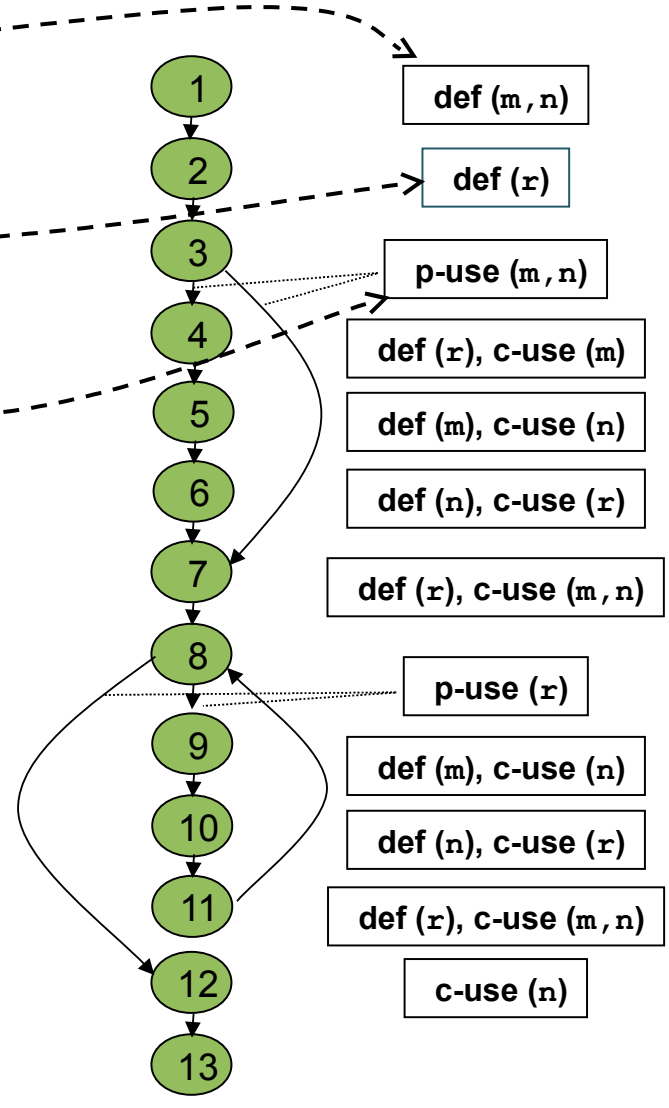
Kriterium „**alle DR-Interaktionen“**: jedes Paar def/ref (ohne dazwischenliegendes erneutes def) auf irgendeinem Weg ausführen.

Weitere Kriterien:

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

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



Datenflussbezogene Überdeckungskriterien: All-defs

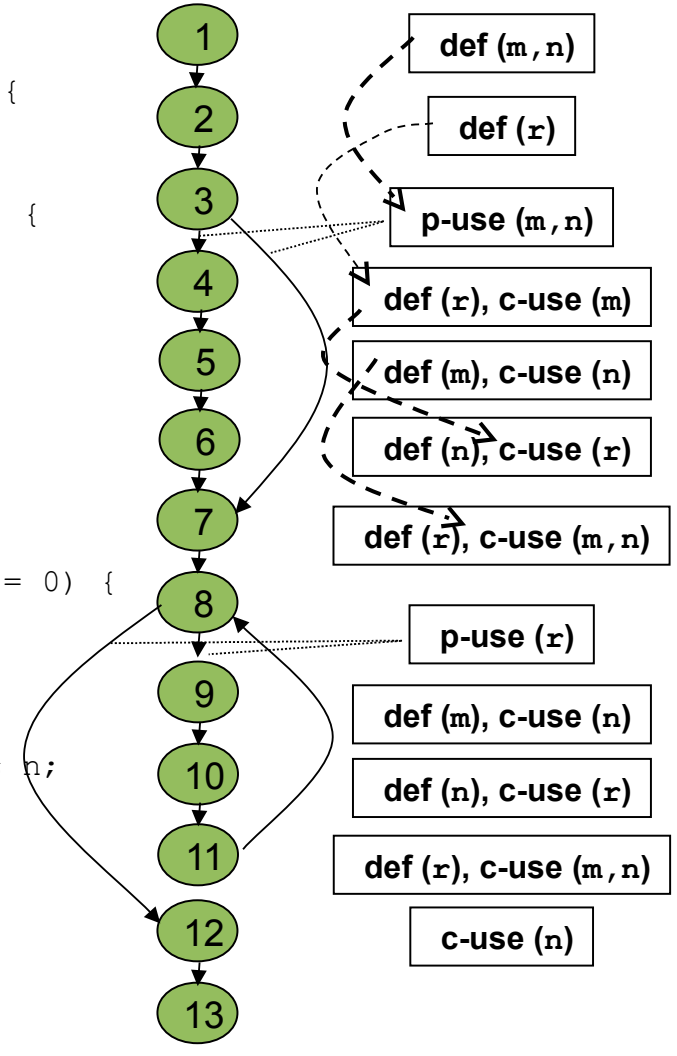
All-defs: Jede Definition min. einmal ohne dazwischen liegendes erneutes def in einem c-use oder p-use verwendet.

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



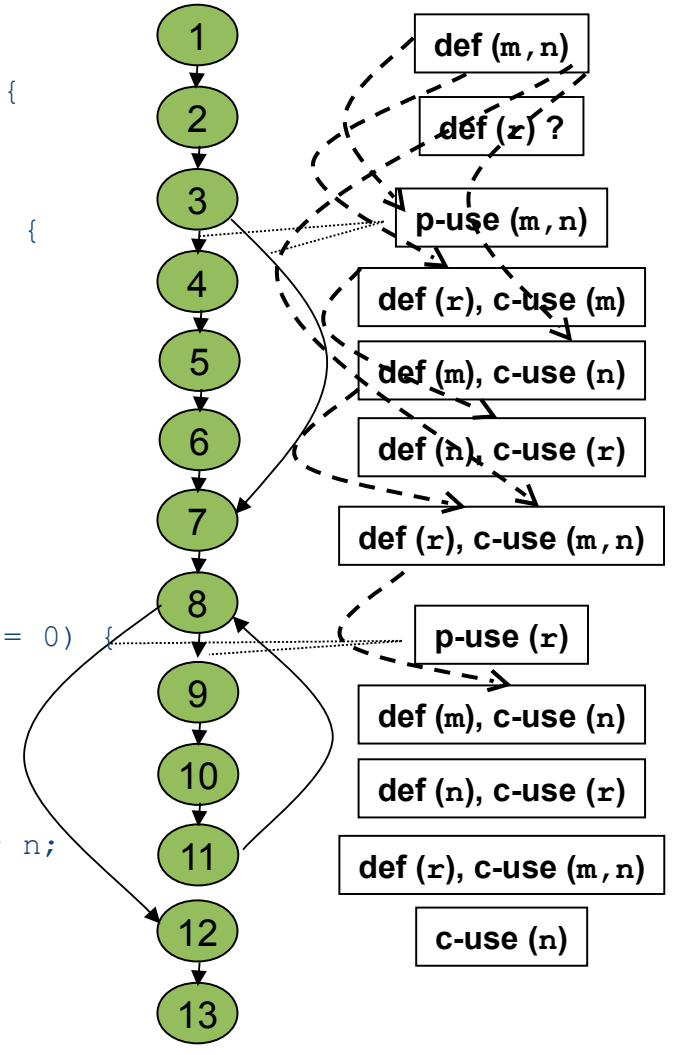
Datenflussbezogene Überdeckungskriterien: alle DR-Interaktionen

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



Datenflussbezogene Überdeckungskriterien: Alle B-/E-Referenzen

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

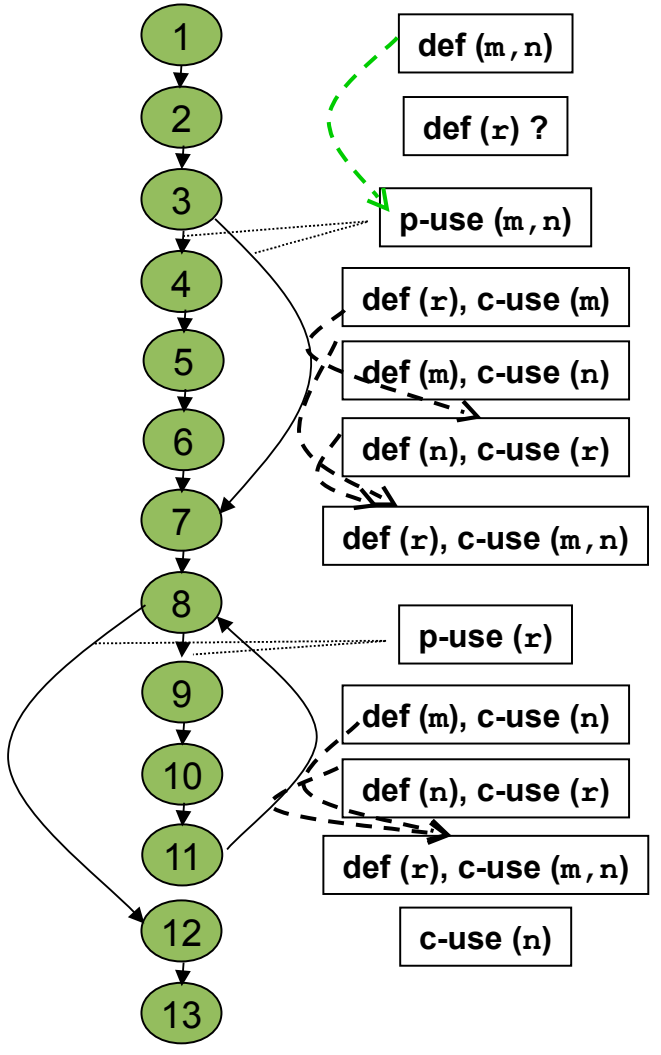
Alle E-Referenzen(p-uses)

1, def m p-use 3-4

.....

Weitere Kriterien:

- **alle k-DR-Interaktionen**
- **Kontextüberdeckung**



Datenflussbezogene Überdeckungskriterien

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

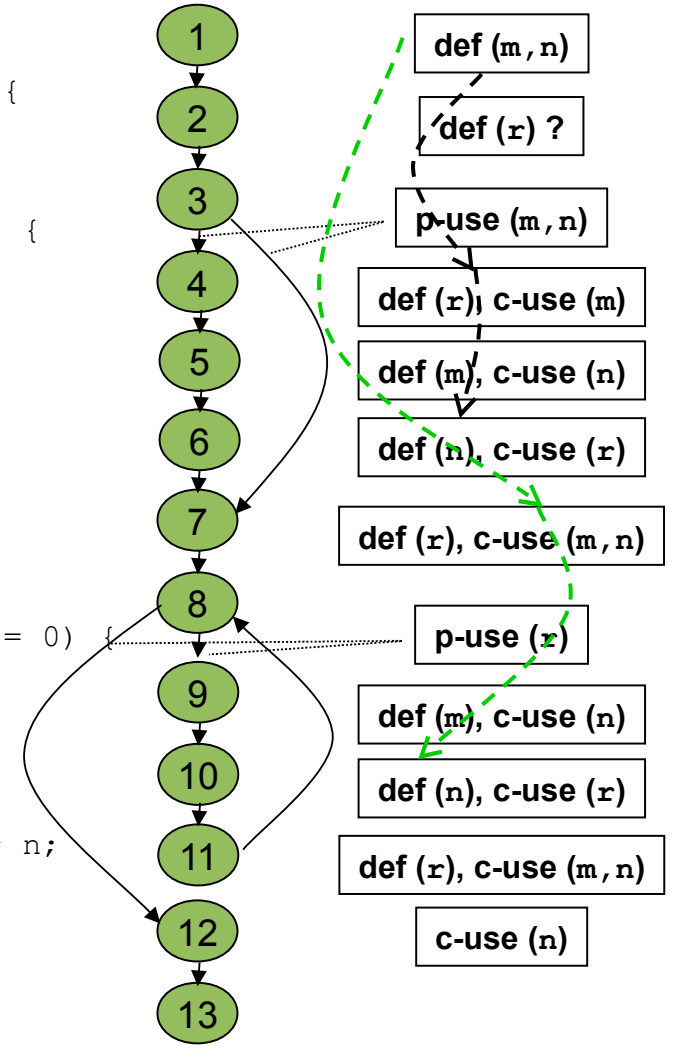
→ damit hängt 6 von 1 ab!

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



Datenflussbezogene Überdeckungskriterien

Beispiele für Kontextüberdeckung

In 7 wird r definiert durch die referenzierten Variablen m und n.

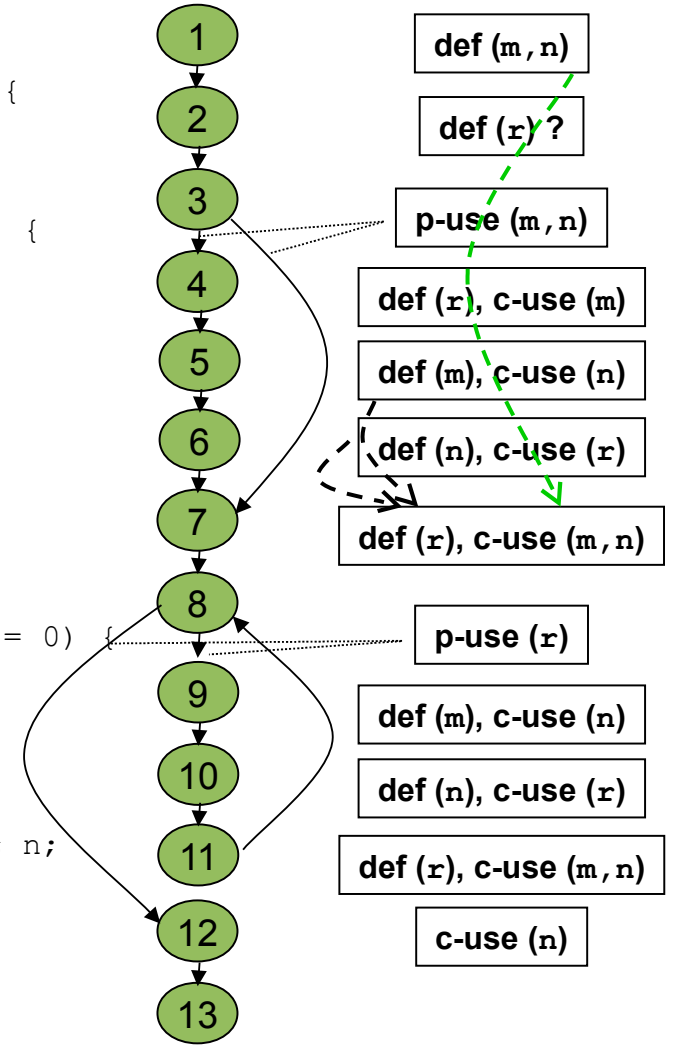
Der „**Definitionskontext**“ dafür sind die Knoten, in denen diese Werte von m und n vorher definiert worden sind, also gibt es hier zwei Fälle:

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



Bewertung des datenflussbasierten Tests

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

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

Testverfahren unterscheiden sich nach der Fehleraufdeckungsfähigkeit:

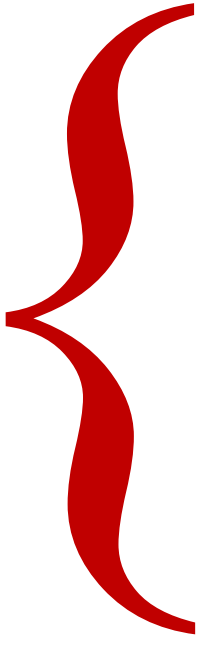
- „Alle B-Referenzen“ fand bspw. bis zu 88% aller Berechnungsfehler.
- „Alle E-Referenzen“ fand bspw. 100% aller Bereichsfehler.
- Folgende Fehler werden dagegen schlecht aufgedeckt:
  - Fehlende Pfade
  - Bereichsfehler durch falsch platzierte Anweisungen und falsche arithmetische Operatoren
  - Berechnungsfehler 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“]

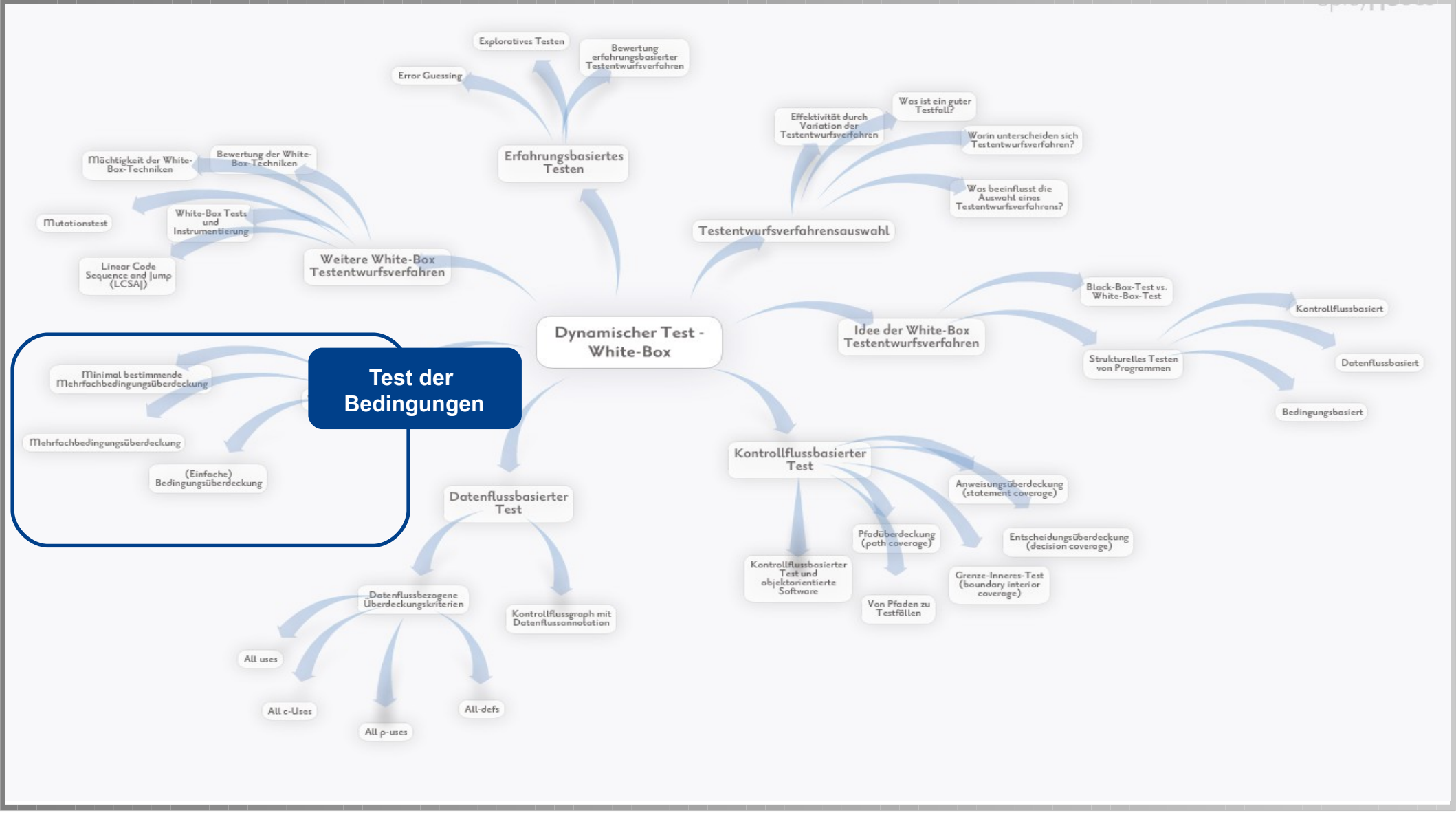
4.5 White-Box-Test

4.5 White-Box-Test



- Idee der White-Box Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- **Test der Bedingungen**
- Weitere White-Box Testentwurfsverfahren
- Erfahrungsbasiertes Testen
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Dynamischer Test – White-Box



Bedingungen in Programmen (und Spezifikationen)

- 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 {...}`

Test der Bedingungen

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.

Test der Bedingungen

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).*

Einfache Bedingungsüberdeckung

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-Testentwurfsverfahren, bei dem Testfälle so entworfen werden, dass Bedingungsausgänge 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 2*n* 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 ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A ∨ B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	C	A ∧ B ∧ 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

Mehrfachbedingungsüberdeckung

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

2 Ausdrücke, 4 Testfälle

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

2 Ausdrücke, 4 Testfälle

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

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

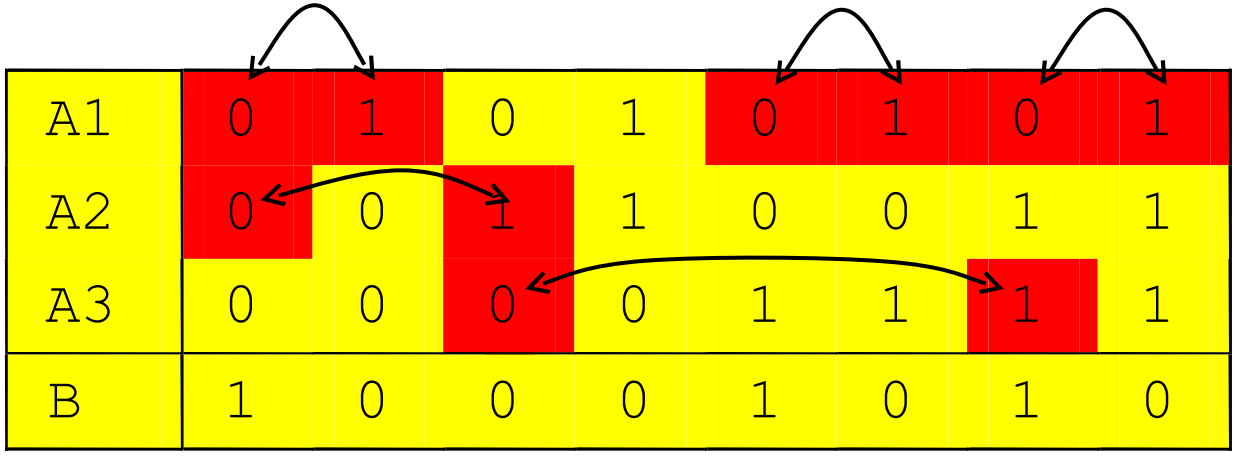
Seien A1, A2, A3 atomare Ausdrücke und B=NOT(A1) AND NOT(A2 AND NOT(A3)) der durch die unten abgebildete Wahrheitswerttabelle definierte Gesamtausdruck.

Frage: Welche Werte-Kombinationen von A1, A2 und A3 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.



Hinweis

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 2*n* (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

Bewertung der Bedingungstestentwurfsverfahren (1)

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 !

Bewertung der Bedingungstestentwurfsverfahren (2)

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.

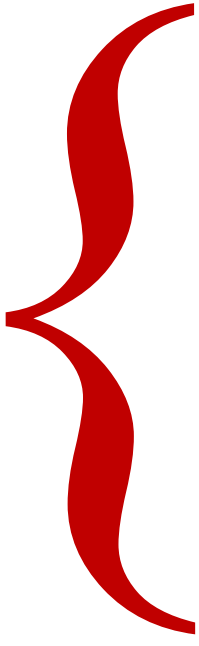
Abbruch der Auswertung

A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

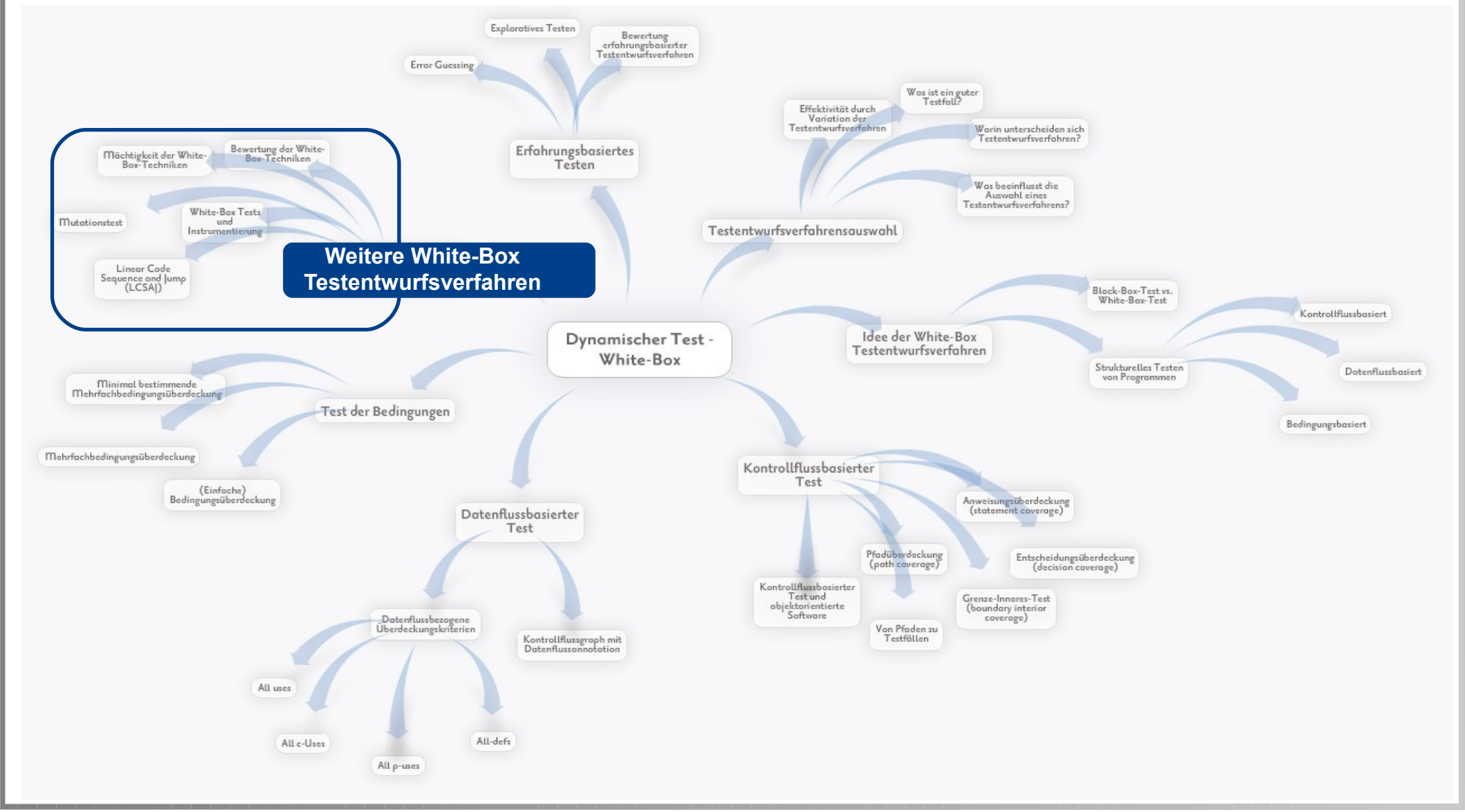
4.5 White-Box-Test

4.5 White-Box-Test



- Idee der White-Box Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- Test der Bedingungen
- **Weitere White-Box Testentwurfsverfahren**
- Erfahrungsbasiertes Testen
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Weitere White-Box Testentwurfsverfahren



Ausdrucks- und anweisungs-bezogenes Testen: Konzept

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 (→ bedingungsbasiertes 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 <, >, ≥, ≤, =, ≠

Fehlerarten:

- Relation „A r B“ enthält falsches Relationssymbol r.
- Relation „A r B“ ist falsch, es müsste „(A+k) r B“ mit k≠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 ε):

- -ε (größter negativer Wert – knapp unterhalb 0)
- 0
- ε (kleinster positiver Wert – knapp oberhalb von 0)

Datenbezogenes Testen

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)

Datenbezogenes Testen

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“.

Bewertung des anweisungs-, ausdrucks- und datenbezogenen Tests

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.]

Weitere White-Box Testentwurfsverfahren: LCSAJ

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.

Weitere White-Box Testentwurfs-verfahren: Mutationstest

Sind mit einem Black- oder White-Box Testentwurfsverfahren 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«.

Mutationstest: Bewertung

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 gekillt werden und das **Äquivalenzproblem** ist nicht entscheidbar !
  → **Mutationstest** kann nicht vollständig automatisiert werden !

White-Box Tests und Instrumentierung

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

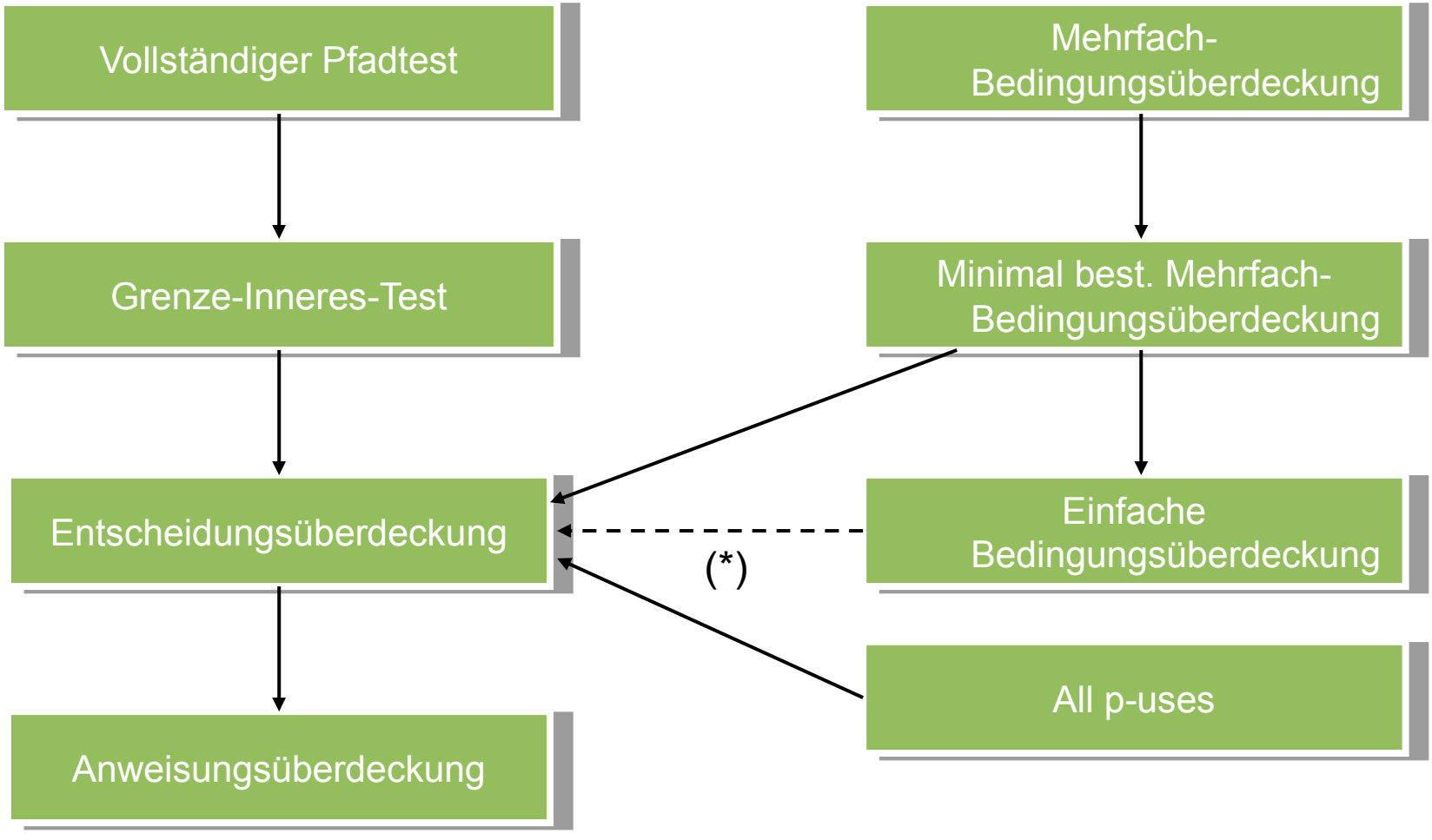
Um den Test auswerten zu können, muss ermittelt werden, welche Programmteile bereits ausgeführt wurden und welche noch nicht zur Ausführung gekommen sind.

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_∞)	Sehr hoch Entdeckt über 70% der Fehler	In den meisten Fällen nicht praktikabel

Diskussion

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

Diskussion

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 Testentwurfsverfahren

Grundlage aller White-Box Testentwurfsverfahren ist der vorliegende Programmtext.

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

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

White-Box Testentwurfsverfahren 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.

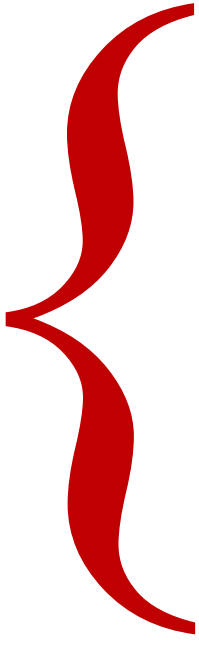
Problem: »Nicht vorhandener Programmcode« bleibt unberücksichtigt.

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

Zur Instrumentierung Werkzeug verwenden.

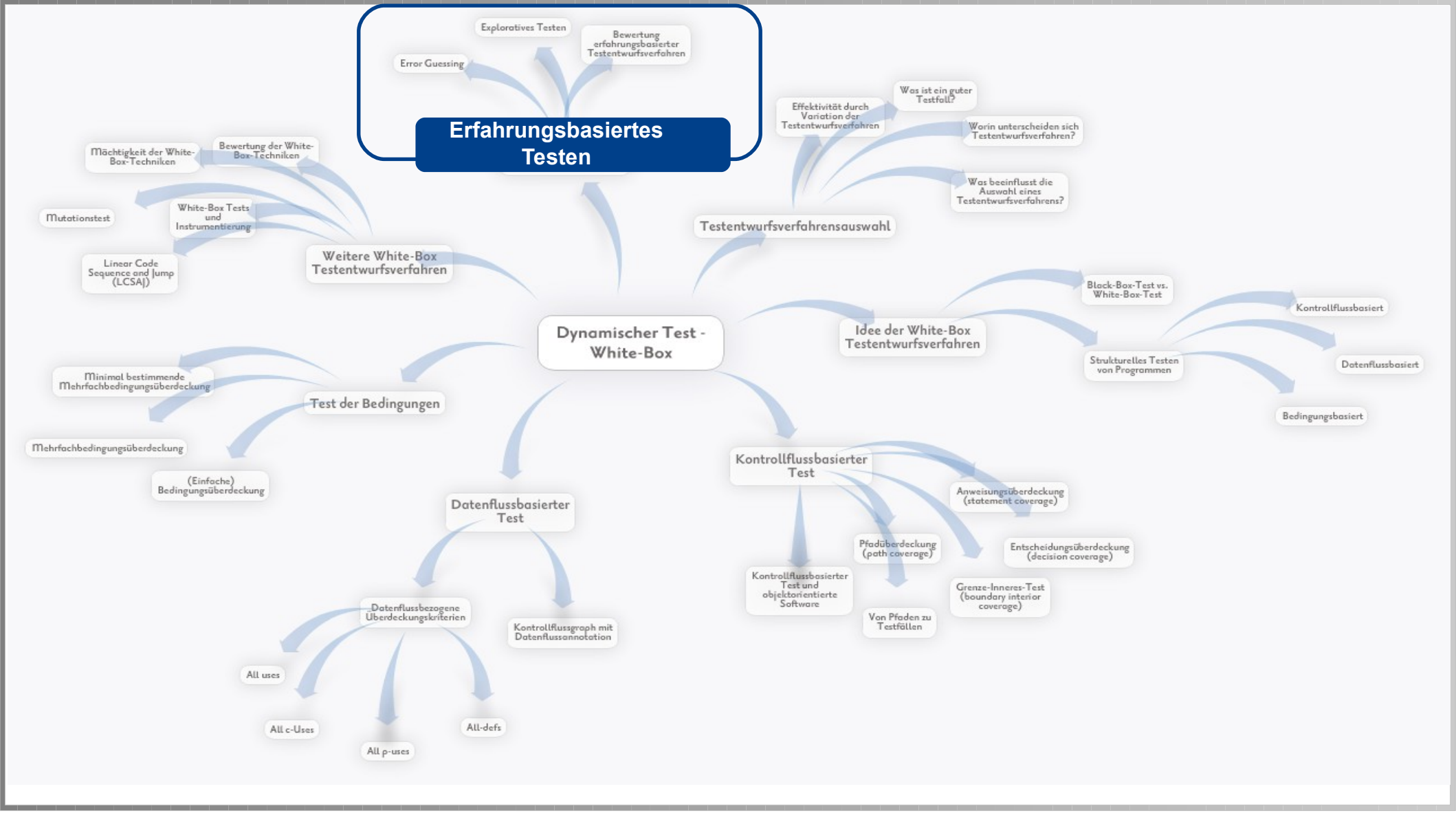
4.5 White-Box-Test

4.5 White-Box-Test



- Idee der White-Box Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- Test der Bedingungen
- Weitere White-Box Testentwurfsverfahren
- **Erfahrungsbasiertes Testen**
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Dynamischer Test – Erfahrungsbasiertes Testen



Erfahrungsbasierte Testentwurfsverfahren

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.

Error Guessing: Charakterisierung

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: Vorgehensweise

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

Vorsicht: Diese Testentwurfsverfahren 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.

Error Guessing: Fehlerkataloge

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 Hoglund 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.

Exploratives Testen: Charakterisierung

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.

Explorativ Testen: Eigenschaften und Aktivitäten

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

Exploratives Testen: Testvorbereitung

Einheiten:

Abgeschlossene Teile des Produktes

Testaufwand 1-2 Tage

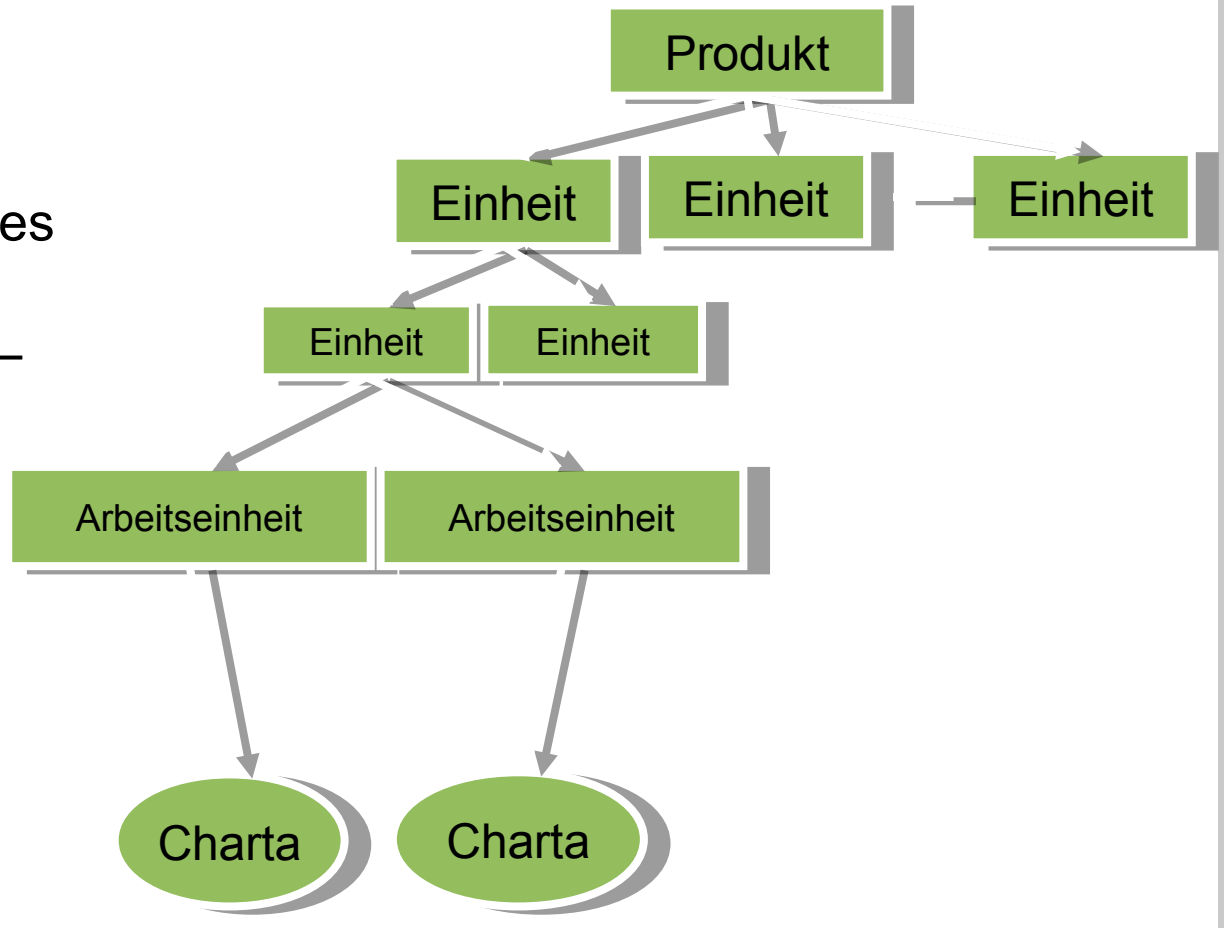
Arbeitseinheit:

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

Exploratives Testen: Test-Charta

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”, …

Exploratives Testen: Testdurchführung

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,

Exploratives Testen: Dokumentation und Auswertung

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,

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.

Bewertung erfahrungsbasierter Testentwurfsverfahren

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.

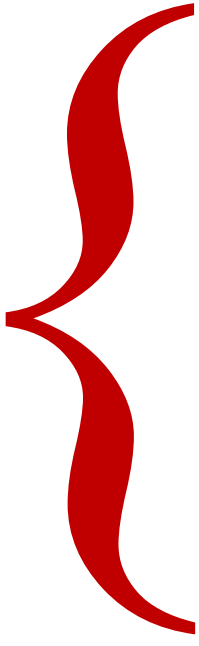
Probleme:

- 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.

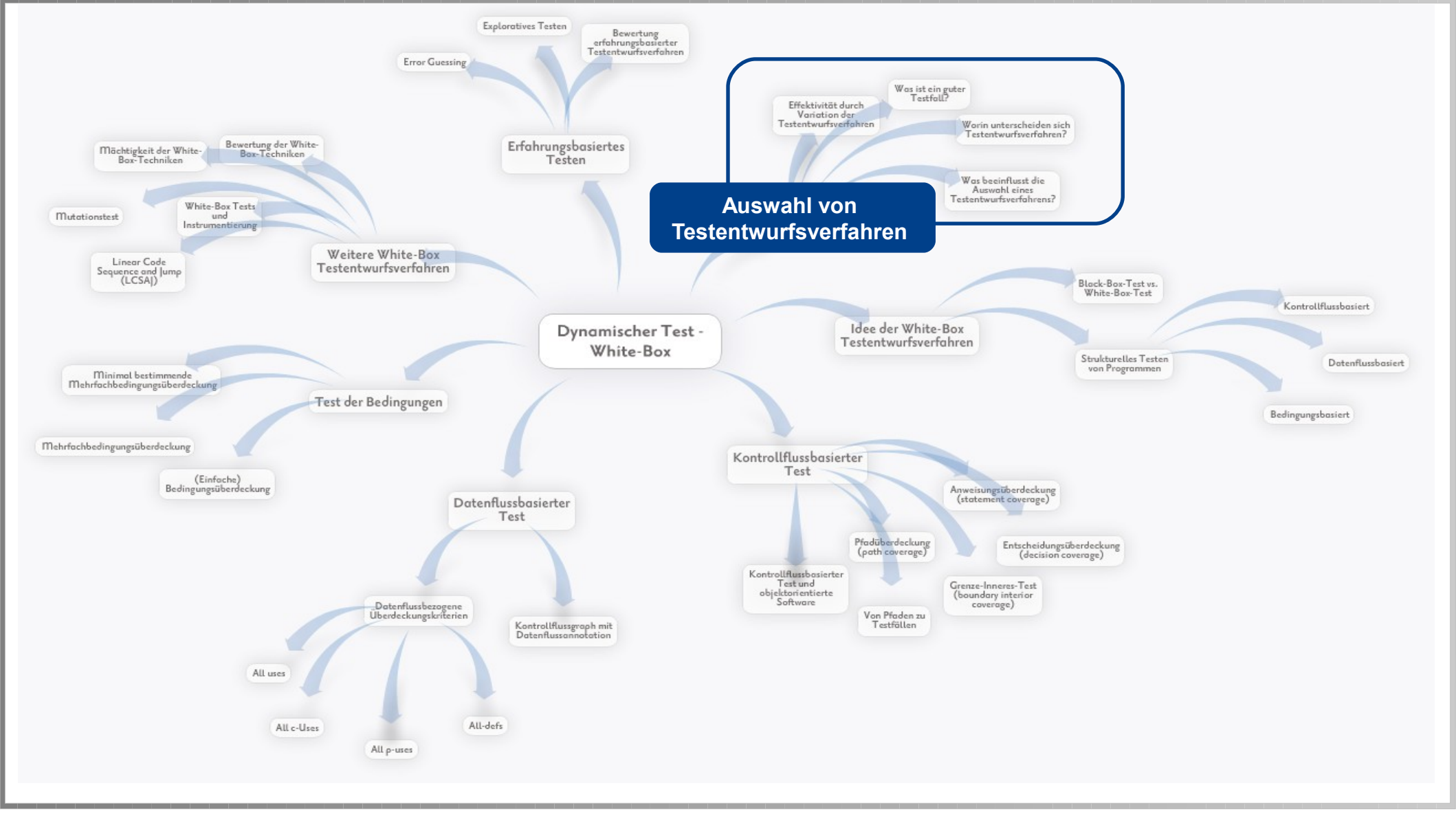
4.5 White-Box-Test

4.5 White-Box-Test



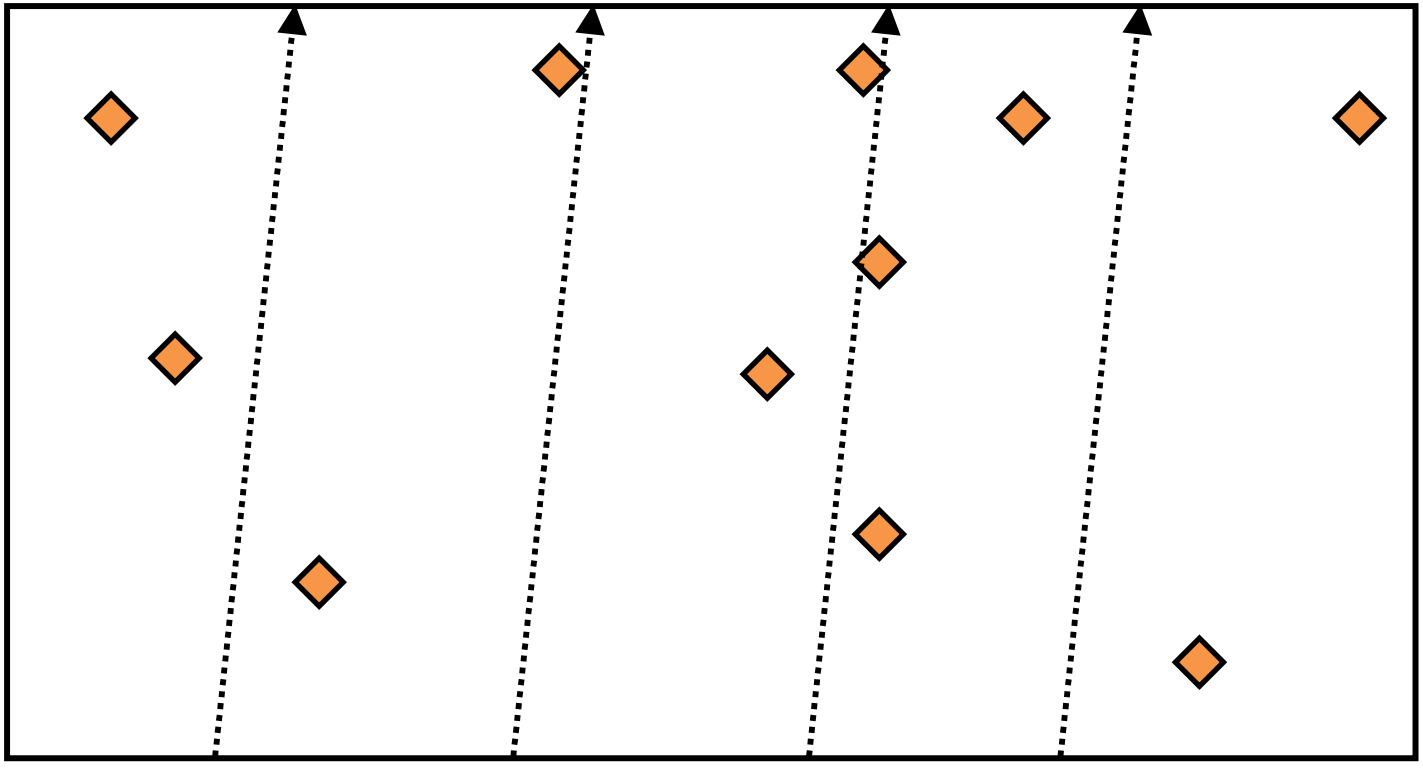
- Idee der White-Box Testentwurfsverfahren
- Kontrollflussbasierter Test
- Datenflussbasierter Test
- Test der Bedingungen
- Weitere White-Box Testentwurfsverfahren
- Erfahrungsbasiertes Testen
- Dynamischer Test: Testentwurfsverfahrensauswahl und Zusammenfassung

Dynamischer Test – Auswahl von Testentwurfsverfahren



Die Gefahr einer einseitigen Teststrategie (1)

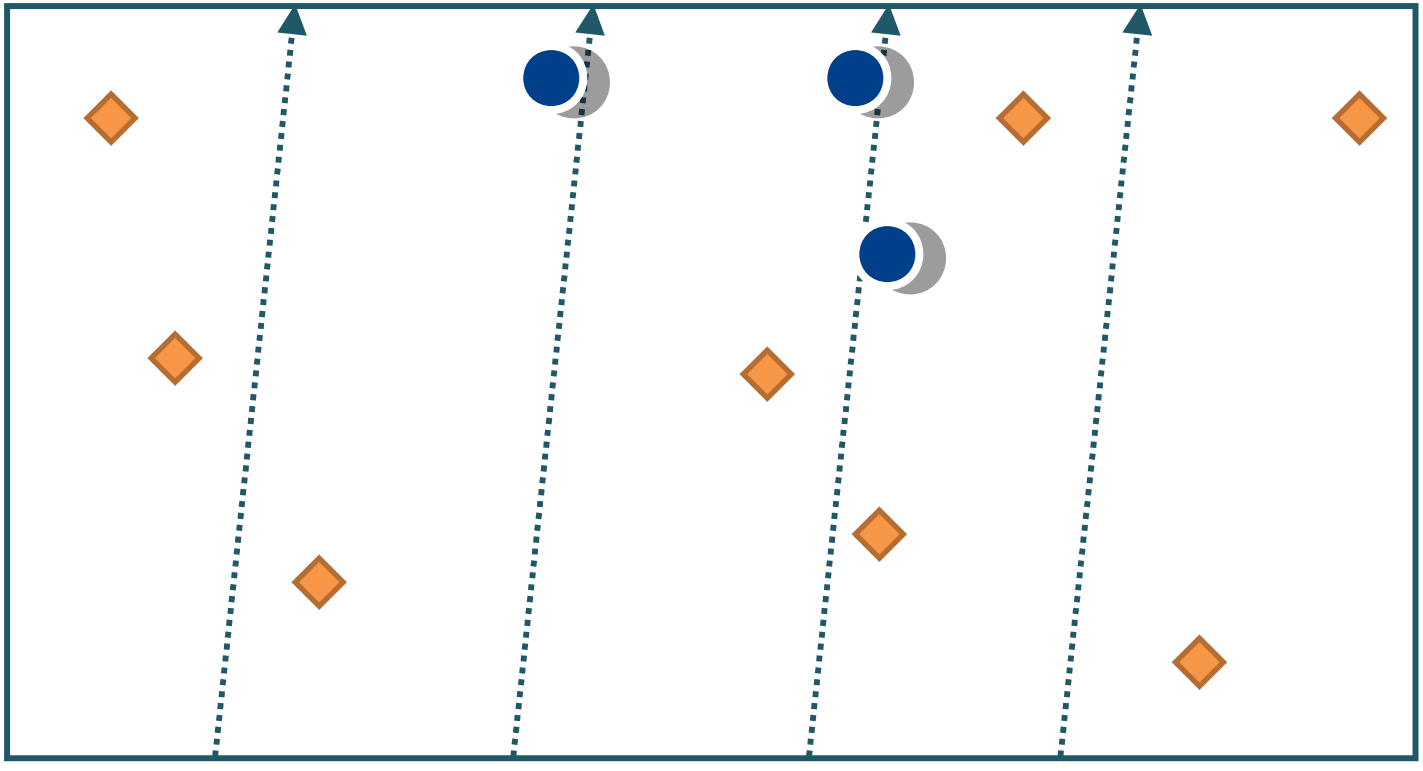
Nach: Rapid Software Testing,



◆ Fehlerzustand (»Mine«)

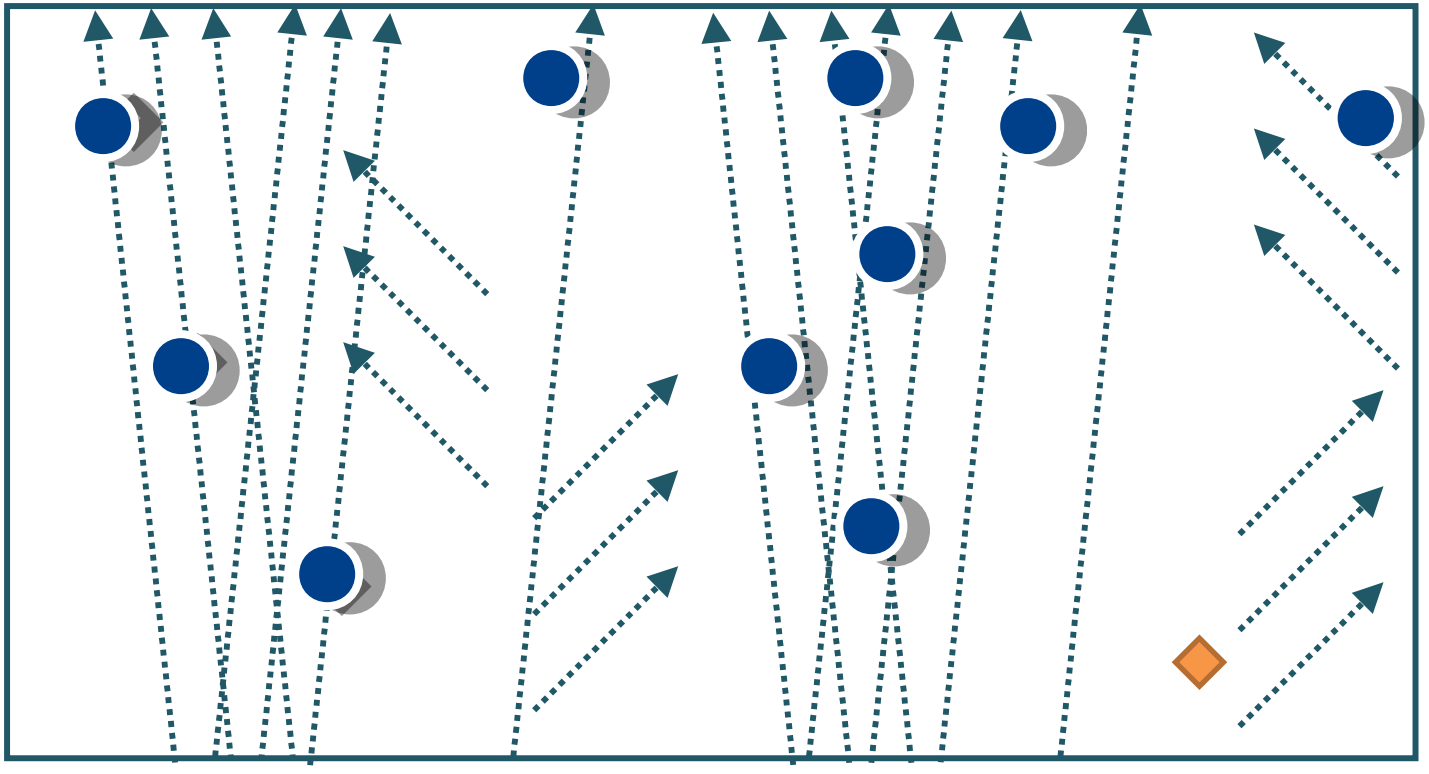
Die Gefahr einer einseitigen Teststrategie (2)

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



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

Effektivität durch Variation der Testentwurfsverfahren



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

Was ist ein guter Testfall ?

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?

TESTART	Welche Art von Tests werden durchgeführt?	Funktionale Anforderungen, Nicht-funktionale Anforderungen
TESTSTUFE	Für welche Testobjekte wird der Test spezifiziert?	Komponententest, Integrationstest, Systemtest, Abnahmetest
TESTER	WER testet?	Entwickler, (erfahrene) Tester, Benutzer, …
TESTABDECKUNG	WAS wird abgedeckt?	Anforderungen, Anweisung, Entscheidung, …
POTENZIELLE FEHLER	Welche potenziellen Fehler sollen identifiziert werden?	Behandlung von Grenzwerten Behandlung von Ausnahmen …
ARTEFAKT	Was ist Grundlage für Auswahl und Herleitung der Testfälle?	Anforderungen -> Black-Box Code -> White-Box …
DOMÄNE / PARADIGMA	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

Was beeinflusst die Auswahl eines Testentwurfsverfahrens ?

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.

Zusammenfassung: Dynamischer Test (2)

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.

Zusammenfassung: Dynamischer Test (3)

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, dass 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 ?

Sie sollten...

- Die Grundidee der White-Box Testentwurfsverfahren (strukturorientierte Testentwurfsverfahren) erläutern können
- White-Box Testentwurfsverfahren 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 Testentwurfsverfahren aufzählen können
- Error Guessing und exploratives Testen als erfahrungsbasierte Testentwurfsverfahren kennen und charakterisieren können
- Dynamische Tests bez. ihres Einsatzbereiches erläutern und mit anderen Testentwurfsverfahren zu einer Teststrategie kombinieren können

Diese Begriffe sollten Sie kennen...