

Software- Qualitätsmanagement

**Vorlesung im Modul 10-202-2319
Software-Management**

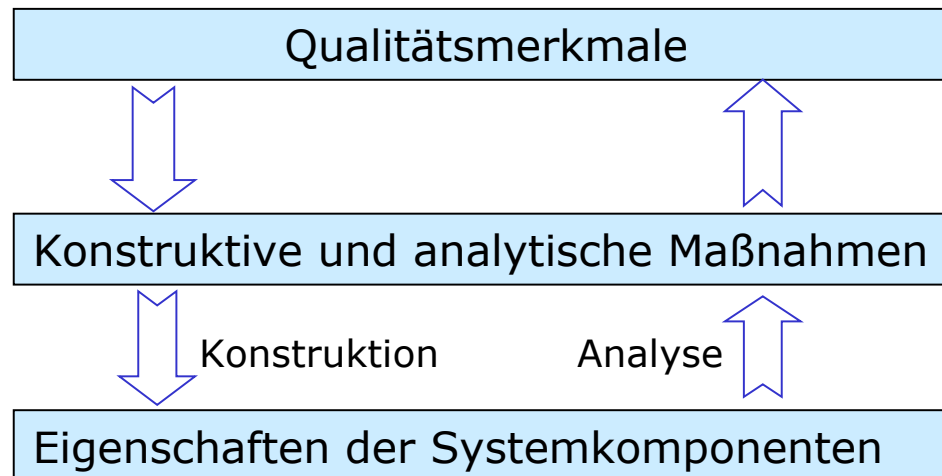
Sommersemester 2009

Prof. Dr. Hans-Gert Gräbe

<http://bis.informatik.uni-leipzig.de/HansGertGraebe>

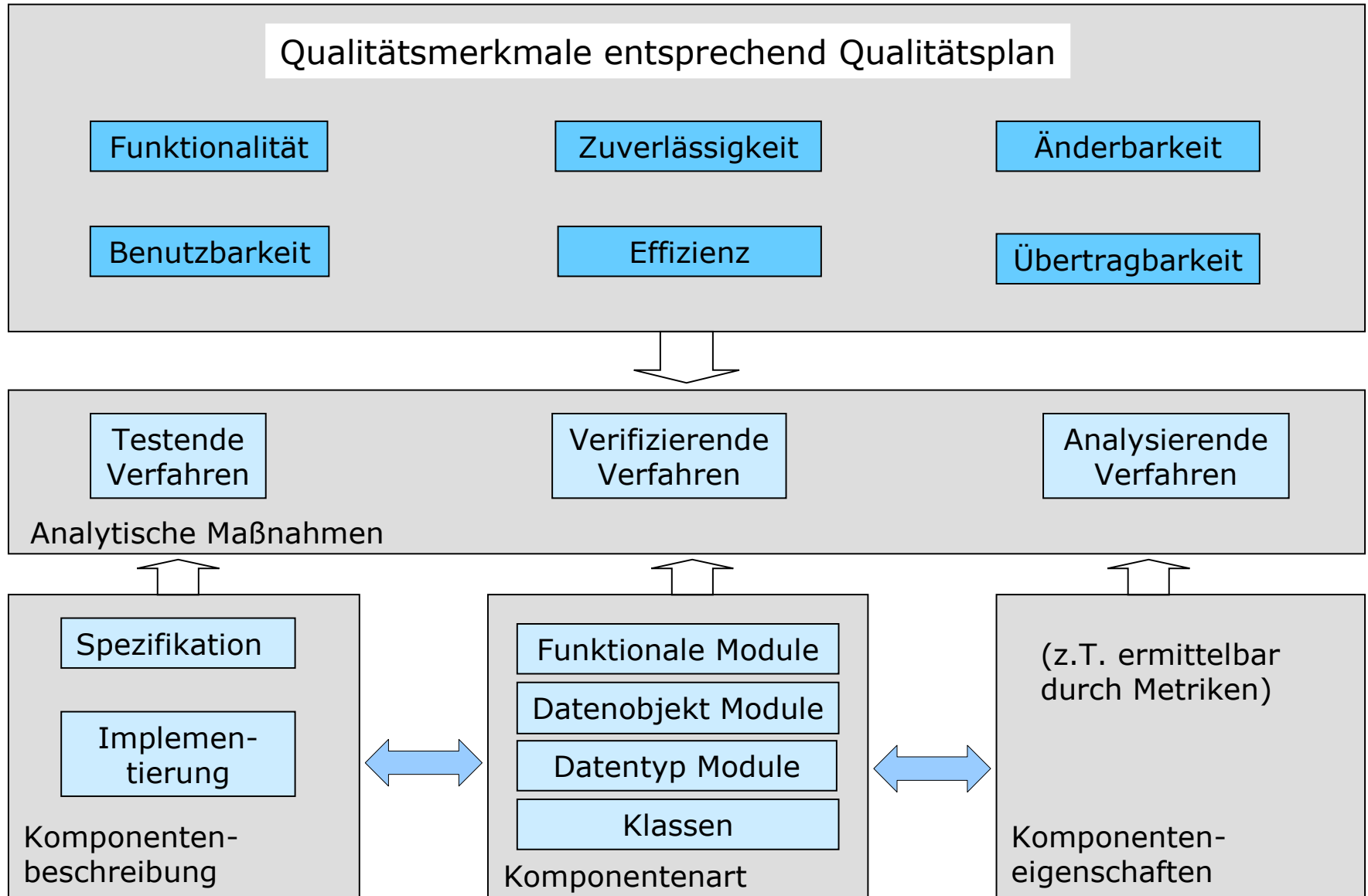
Software-Produktqualität ist abhängig von:

- Qualität der Systemkomponenten
- Qualität der Beziehungen zwischen den Komponenten



- Konstruktive Maßnahmen siehe VL „Software-Technik“
- analytische Maßnahmen beziehen sich im Wesentlichen auf Funktionalität, Zuverlässigkeit und evtl. Änderbarkeit

1. Einführung



2. Was sind Fehler?

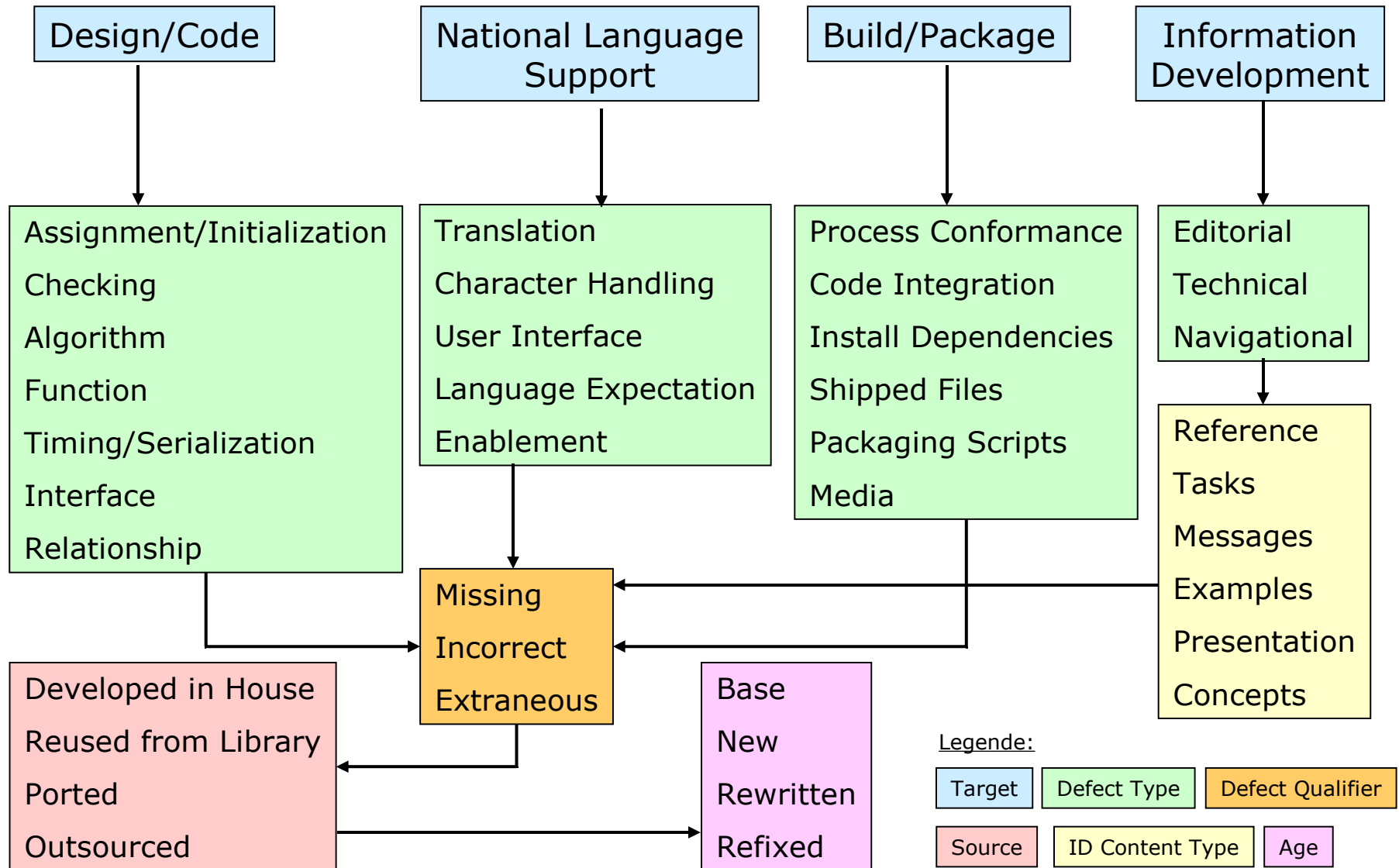
Fehler als zentraler Begriff

Konstruktives Ziel: Entwicklung fehlerfreier Software-Komponenten

Analytisches Ziel: Nachweis der Fehlerfreiheit von Software-Komponenten

Als **Fehler** wird

- jede Abweichung der tatsächlichen Ausprägung eines Qualitätsmerkmals von der vorgesehenen Soll-Ausprägung,
- jede Inkonsistenz zwischen der Spezifikation und der Implementierung und
- jedes strukturelle Merkmal des Programmtexts, das ein fehlerhaftes Verhalten des Programms verursacht, bezeichnet.



Analyseverfahren

Die Art und die Eigenschaften der Systemkomponenten bestimmen die Auswahl geeigneter analytischer Maßnahmen:

- Funktionale Module
- Datenobjekt-Module
- Datentyp-Module
- Klassen

Unterschied der eingesetzten analytischen Maßnahmen insbesondere zwischen Komponenten mit komplexen Kontrollstrukturen und Komponenten mit komplexen Datenstrukturen.

Testende Verfahren haben das Ziel, Fehler zu erkennen

- Dynamische Testverfahren
- Statische Verfahren

Verifizierende Verfahren sollen die Korrektheit einer Komponente beweisen

- Verifikation
- Symbolische Ausführung

Analysierende Verfahren sollen Eigenschaften von Komponenten darstellen oder vermessen

- Analyse der Bindungsart
- Metriken
- Grafiken und Tabellen
- Anomalienanalyse

1. Einführung

Testende Verfahren - Prinzipieller Zugang

- Das tatsächliche Verhalten wird **stichprobenartig** an Hand einer Menge von **Testfällen** untersucht und die Ergebnisse mit den erwarteten Ergebnissen (Spezifikation, Normen) verglichen und dokumentiert.
- Testfälle werden entsprechend den Testzielen speziell ausgewählt.
- Einsatz um
 - Programmfehler aufzufinden (Bugs)
 - Wiederauftreten von Fehlern zu vermeiden (Regressionstests)
- Fehlerfreiheit ist plausibel, aber nicht garantiert.
- Abzugrenzen von
 - **Verifikation** als strengem Korrektheitsbeweis
 - **Ausprobieren** als einer Entwicklungsmethode (trial and error)

Begriff des Programms

- Programm = schrittweise Transformation einer Menge von Eingabedaten in eine Menge von Ausgabedaten nach einem vorgegebenen Algorithmus
- Black-Box-Betrachtung: $f: X \rightarrow Y$
 - Spezifikation, funktionale Korrektheit
- Transformation = Abarbeiten einzelner Programmschritte, in denen die Daten entsprechend den angegebenen Instruktionen verändert werden.
 - zustandsorientierte Betrachtung: Datenfluss
 - übergangsorientierte Betrachtung: Kontrollfluss
- Programmstatus = Zustand der Gesamtheit der durch das Programm manipulierten Daten
 - Anweisungen und Deklarationen
 - Variablenbegriff als Wertcontainer
 - Sichtbarkeit und Lebensdauer
 - Compilezeit und Laufzeit

Statische und dynamische Testverfahren

Dynamische Testverfahren

- übersetztes und ausführbares Programm wird mit konkreten Eingabewerten ausgeführt
- evtl. Instrumentierung des Programms
- Test in realer Laufzeitumgebung
- Stichprobenverfahren (Testfälle)
- Ziele: Finden von Fehlern (debugging), Finden und Optimieren laufzeitkritischer Bereiche (profiling)
- Korrektheit kann so nicht bewiesen werden
- Klassifikation nach Herkunft der Testfälle

Statische Testverfahren

- Analyse des Quellcodes, evtl. testfallorientiertes Durchgehen
- typische Verfahren: manuelle Prüfmethode

Glossar

Prüfling, Testling oder **Testobjekt**

- das zu testende Programm oder die Komponente (Prüfling als allgemeiner Begriff)

Testverfahren

- grundlegendes Verfahren, mit dem einzelne Eigenschaften eines Testlings durch eine geeignete Anzahl von Testfällen untersucht werden

Testfall

- Satz von Testdaten, der die vollständige Ausführung eines zu testenden Programms bewirkt

Testdatum

- Eingabewert, der einen Eingabeparameter des Testobjekts instanziiert

1. Einführung

Testtreiber

- Testrahmen, mit dem eine nicht extern zugängliche Funktion interaktiv aufgerufen werden kann.
- Können durch Testwerkzeuge automatisch generiert werden.

Instrumentierung

- Der Quellcode des Testlings wird für die Analyse der Testfälle mit zusätzlichem Protokollcode versehen oder dieser aktiviert.
- Instrumentierter Testling wird übersetzt. Protokoll enthält Informationen über das Laufzeitverhalten des Testlings während des Abarbeitens der einzelnen Testfälle.

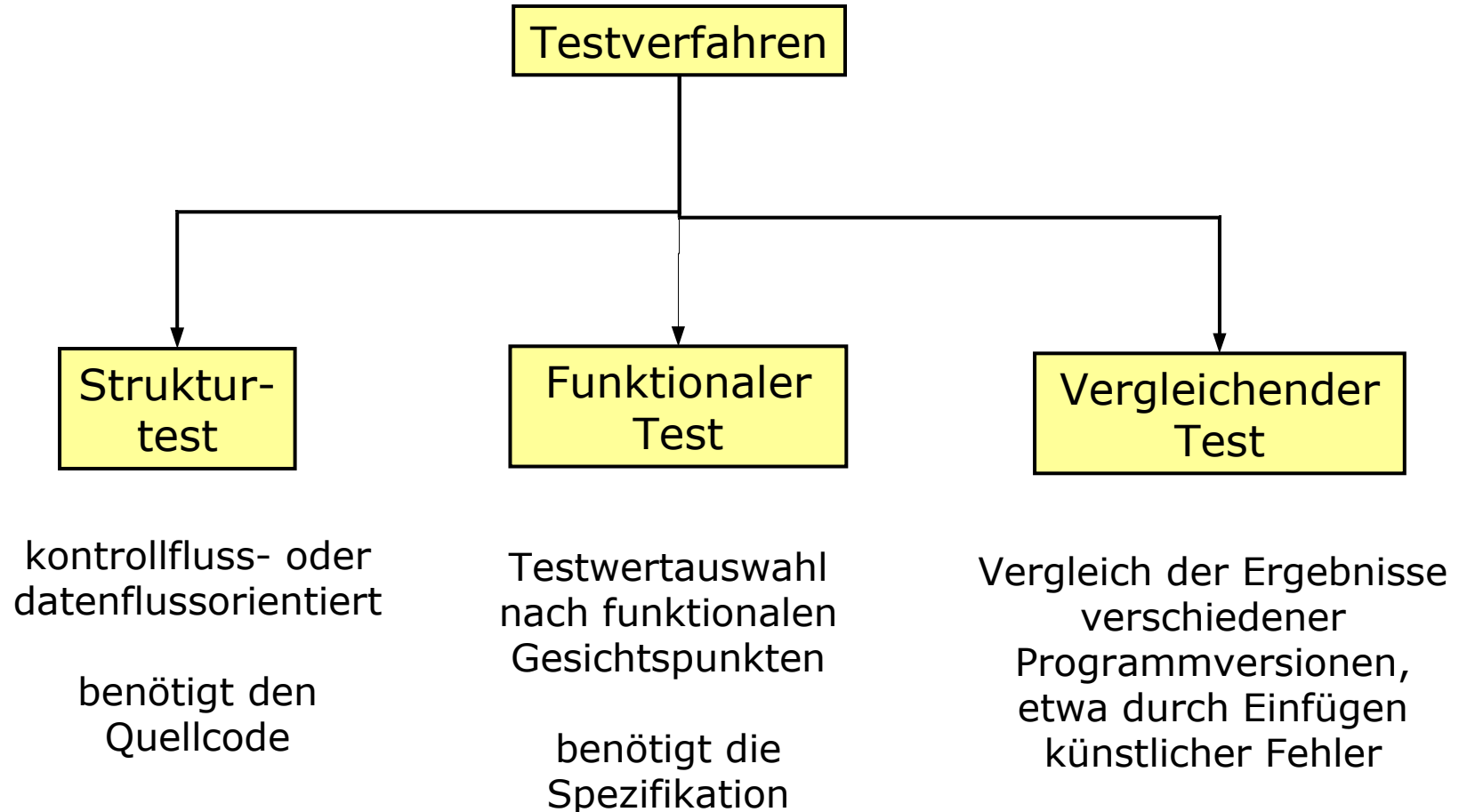
Überdeckungsgrad

- Maß für den Grad der Vollständigkeit eines Tests bezogen auf ein bestimmtes Testverfahren

Regressionstest

- automatische werkzeuggestützte Neuausführung bereits durchlaufener Tests nach Änderungen am Testling.

Klassifikation testender Verfahren



- **Strukturtest**

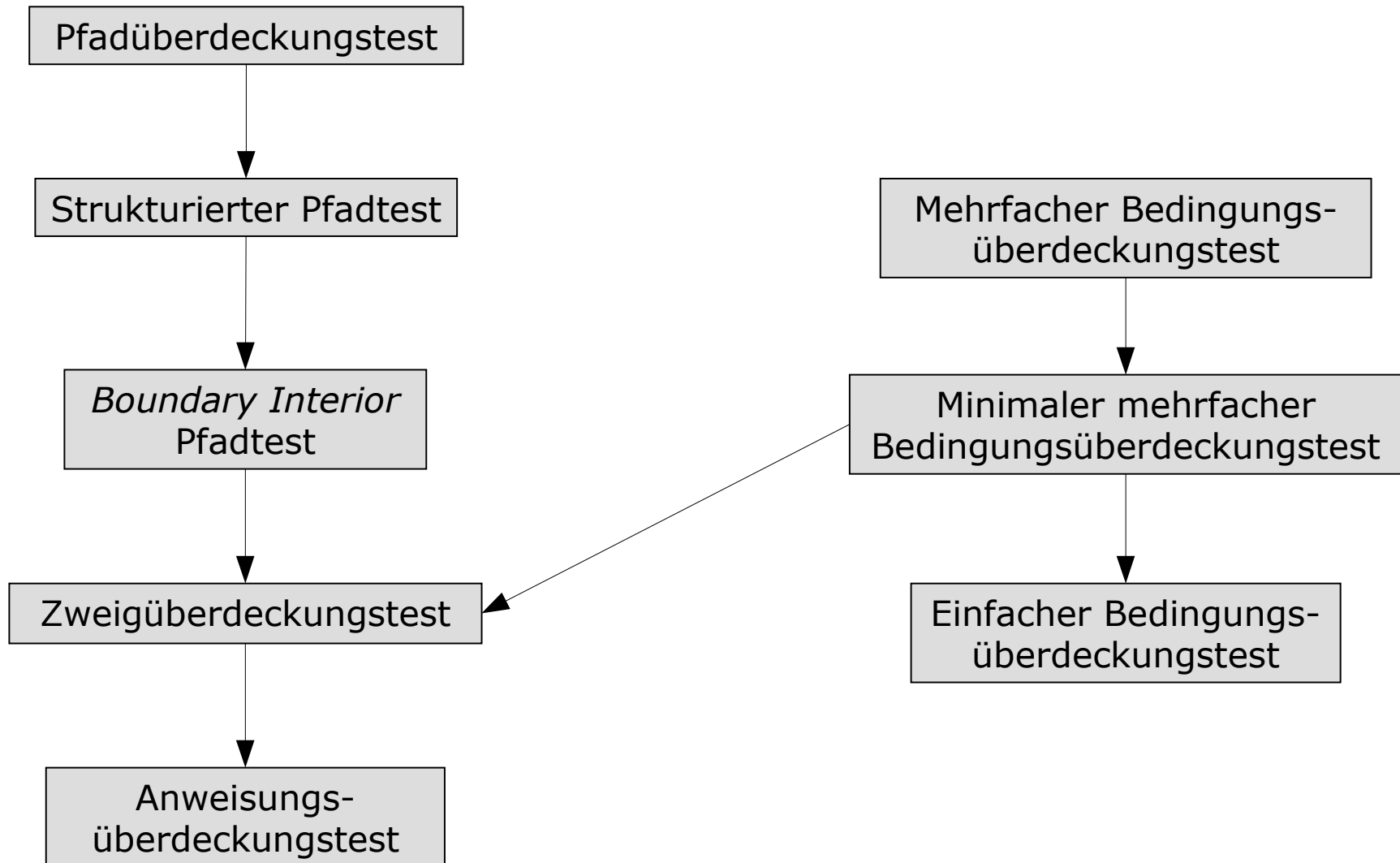
- kontrollflussorientiert (Monitoring des Programmflusses)
 - Anweisungsüberdeckung
 - Zweigüberdeckung
 - Pfadüberdeckung (volle Version kombinatorisch exponentiell!)
 - Bedingungsüberdeckung
- datenflussorientiert (Monitoring der Programmdaten)

- **Funktionaler Test**

- funktionale Äquivalenz
- Grenzwertanalyse
- Test spezieller Werte (Szenarios)
- Zufallstest
- zustandsübergangsgetriebene Tests

Überblick

- Basieren auf der Kontrollstruktur des zu prüfenden Programms
- Gehören zu den **Strukturtest-**, **White Box-** oder **Glass Box-Verfahren**
- **Ziel** ist, mit möglichst wenigen Testfällen alle Anweisungen, Zweige oder Pfade zu durchlaufen
- Sorgfältige Auswahl der Testfälle, um mögliche strukturelle Probleme genau zu überdecken.



Ein Beispiel

/* Funktion: ZaehleZchn

Aufgabe: Die Prozedur ZaehleZchn zählt die Zeichen sowie Vokale in einem String. Dazu werden die Klassenvariablen *Gesamtzahl* und *VokalAnzahl* der Klasse *count* inkrementiert.

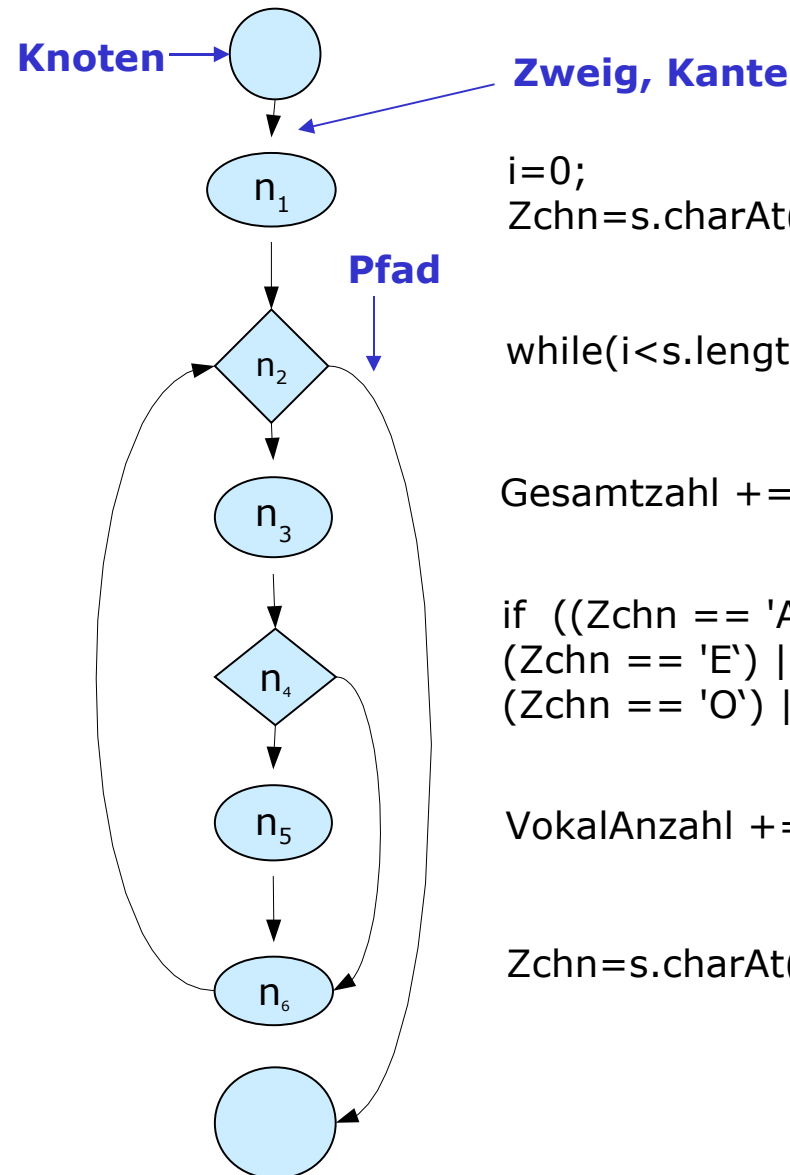
Randbedingung: Die Funktion stellt sicher, dass Gesamtzahl stets größer oder gleich VokalAnzahl ist. */

```
public static void main(String[] argv) {  
    Gesamtzahl=0;  
    VokalAnzahl=0;  
    for(int i=0; i<argv.length; i++)  
        zaehleZeichen(argv[i].toUpperCase());  
    System.out.println("Argumente enthalten "+Gesamtzahl+  
        " Zeichen, davon "+VokalAnzahl+" Vokale");  
}
```

```
public class count {  
  
    static int Gesamtzahl;  
    static int VokalAnzahl;  
  
    static void zaehleZeichen(String s) {  
        int i=0;  
        char Zchn=s.charAt(i++);  
        while(i<s.length()) {  
            Gesamtzahl+=1;  
            if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I')  
                || (Zchn == 'O') || (Zchn == 'U'))  
                VokalAnzahl+=1;  
            Zchn=s.charAt(i++);  
        }  
    }  
}
```

Kontrollflussgraph

- auch Programmablaufplan
- Gerichteter Graph, bestehend aus Knoten und Kanten
- Besitzt einen Start- und einen Endknoten
- Folge von Knoten und Kanten vom Start- zum Endknoten heißt Pfad



```
i=0;  
Zchn=s.charAt(i++)
```

```
while(i<s.length())
```

```
Gesamtzahl += 1;
```

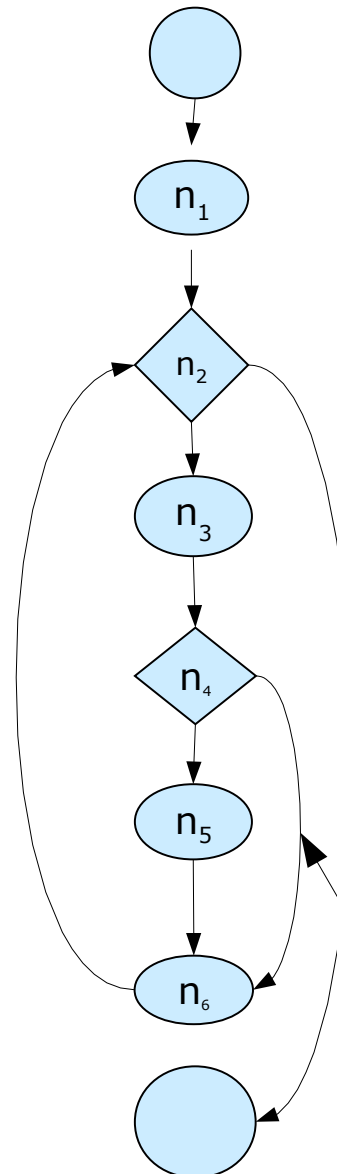
```
if ((Zchn == 'A') ||  
    (Zchn == 'E') || (Zchn == 'I') ||  
    (Zchn == 'O') || (Zchn == 'U'))
```

```
VokalAnzahl += 1;
```

```
Zchn=s.charAt(i++);
```

2.1 Anweisungsüberdeckungstest

- Auch C_0 -Test ($C = \text{Coverage}$) genannt
- Verlangt Ausführung aller Anweisungen (**Knoten**)
- Testmenge: $\{„A“\}$
- Ein Test reicht aus. Testpfad enthält alle Knoten, aber nicht alle Kanten



```
i=0;  
Zchn=s.charAt(i++)
```

```
while(i<s.length())
```

```
Gesamtzahl += 1;
```

```
if ((Zchn == 'A') ||  
(Zchn == 'E') || (Zchn == 'I') ||  
(Zchn == 'O') || (Zchn == 'U'))
```

```
VokalAnzahl += 1;
```

```
Zchn=s.charAt(i++);
```

Zweig (n_4, n_6) wird nicht
notwendig ausgeführt

2.1 Anweisungsüberdeckungstest

Eigenschaften:

- 100prozentige Überdeckung bedeutet: jede Anweisung wurde mindestens einmal ausgeführt
- Wesentliche Aspekte eines Programms werden nicht geprüft

Metrik: *Überdeckungsgrad* =

Zahl der ausgeführten Anweisungen / Gesamtzahl aller Anweisungen

Leistungsfähigkeit:

- Niedrigste Fehleridentifizierungsquote, 18 % der Fehler werden entdeckt

Bewertung:

- *Notwendiges*, aber nicht hinreichendes Testkriterium
- Nicht ausführbarer Code kann gefunden werden
- Eigenständig nicht geeignet, aber in Kombination mit anderen Verfahren

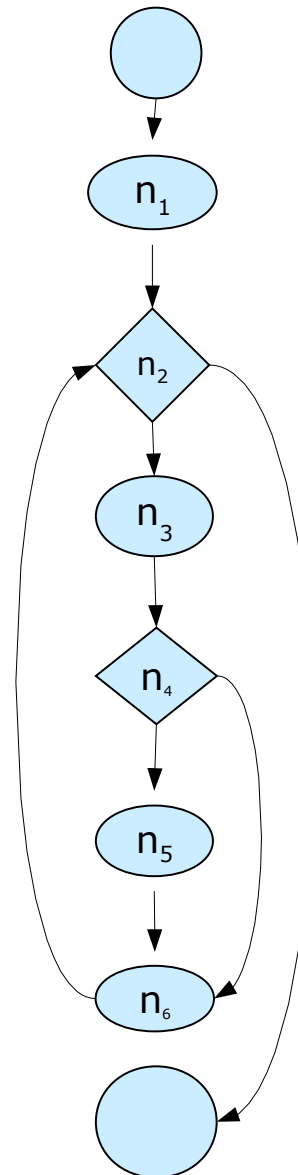
5. Testende Verfahren

2.2 Zweigüberdeckungstest

- Auch C_1 -Test genannt
- Verlangt Ausführung aller Zweige (**Kanten**)
- Testmenge: {„AB“}
- Ein Test reicht aus. Testpfad enthält alle Kanten.

Insbesondere sind die Kanten $n_4 - n_5 - n_6$ (Durchlauf mit „A“) sowie $n_4 - n_6$ (Durchlauf mit „B“) abgedeckt

- Zweigüberdeckung wird auch als Entscheidungsüberdeckung bezeichnet



```
i=0;  
Zchn=s.charAt(i++)
```

```
while(i<s.length())
```

```
Gesamtzahl += 1;
```

```
if ((Zchn == 'A') ||  
    (Zchn == 'E') || (Zchn == 'I') ||  
    (Zchn == 'O') || (Zchn == 'U'))
```

```
VokalAnzahl += 1;
```

```
Zchn=s.charAt(i++);
```

2.2 Zweigüberdeckungstest

Eigenschaften:

- 100prozentige Überdeckung bedeutet: jeder Zweig wurde mindestens einmal durchlaufen.
- Fehlende Zweige können nicht direkt entdeckt werden.

Metrik: *Überdeckungsgrad* =

Zahl der erfassten Kanten / Gesamtzahl aller Kanten

Leistungsfähigkeit:

- Höhere Fehleridentifizierungsquote als Anweisungsüberdeckung, ca. 34% der Fehler werden entdeckt, 79% der Kontrollflussfehler und 20% der Berechnungsfehler
- Leistungsfähigkeit schwankt in weitem Bereich zwischen 25% bis 75%

Bewertung:

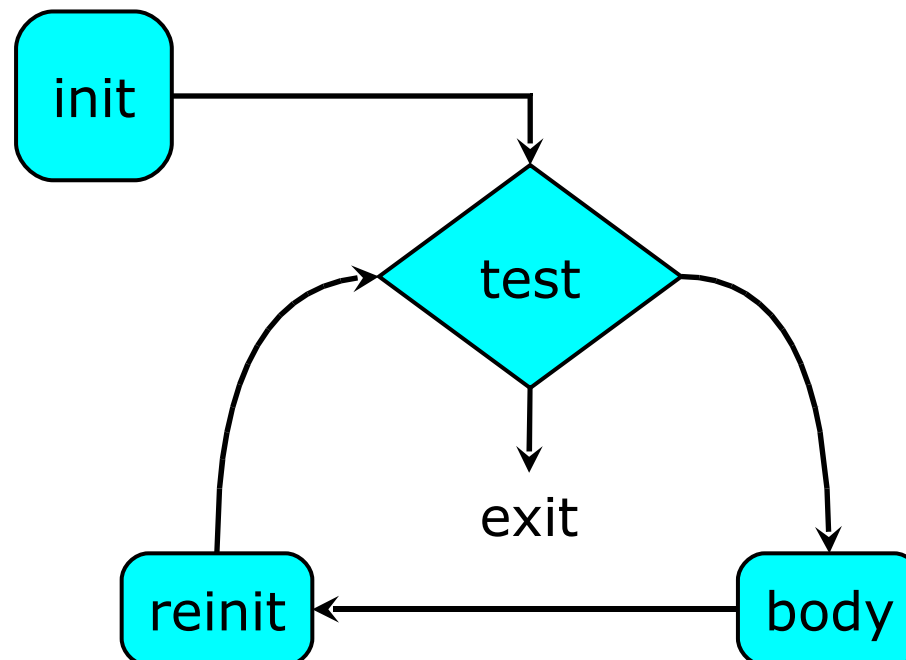
- Gilt als *das* minimale Testkriterium
- Nicht ausführbare Zweige können gefunden werden
- Korrektheit des Kontrollflusses an Verzweigungen wird kontrolliert
- Gezielte Optimierung häufig durchlaufener Programmteile möglich

Nachteile:

- Unzureichend für den Test von Schleifen
- Keine Berücksichtigung von Abhängigkeiten zwischen Zweigen
- Nicht geeignet für den Test komplexer Bedingungen
- Lösung der beiden ersten Nachteile: Pfadüberdeckungstest
- Lösung des letzten Nachteils: Bedingungsüberdeckungstests

Testen von Schleifen

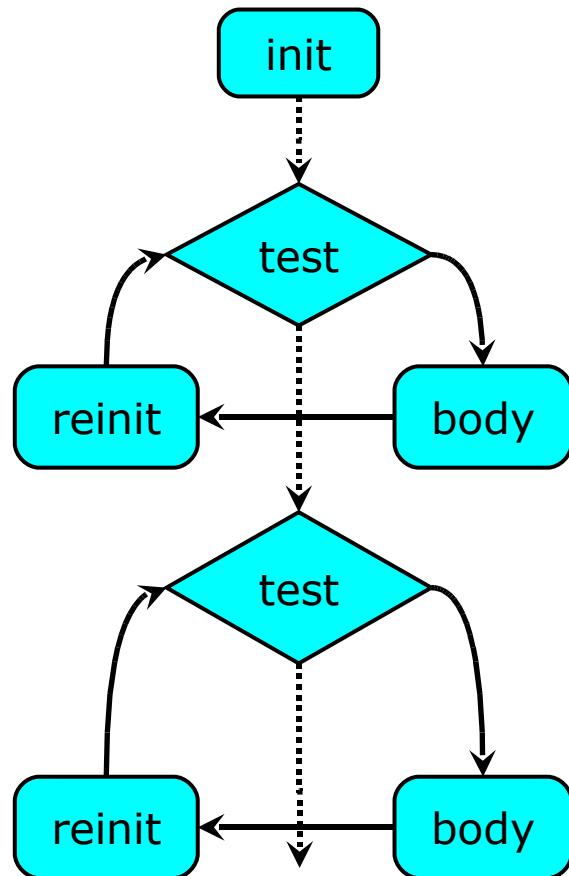
- Anweisungs- und Zweigüberdeckung haben Probleme mit dem Test von Schleifen
- Typische Schleifenstruktur:



5. Testende Verfahren

2.3. Pfadüberdeckungstests

Problem des Wachstums der Anzahl der Pfade



- Zahl der Testfälle von while-Schleifen ist nicht vorab bekannt oder nicht beschränkt
- Zahl der Testfälle konsekutiver Schleifen ist multiplikativ:
 $N = N_1 * N_2$
- Schleife mit Verzweigung im Körper: Ist N Schranke für Zahl der Schleifendurchläufe, so sind im worst case 2^N Testfälle erforderlich (exponentielles Wachstum) – siehe das Beispiel auf der nächsten Folie

5. Testende Verfahren

2.3 Pfadüberdeckungstest

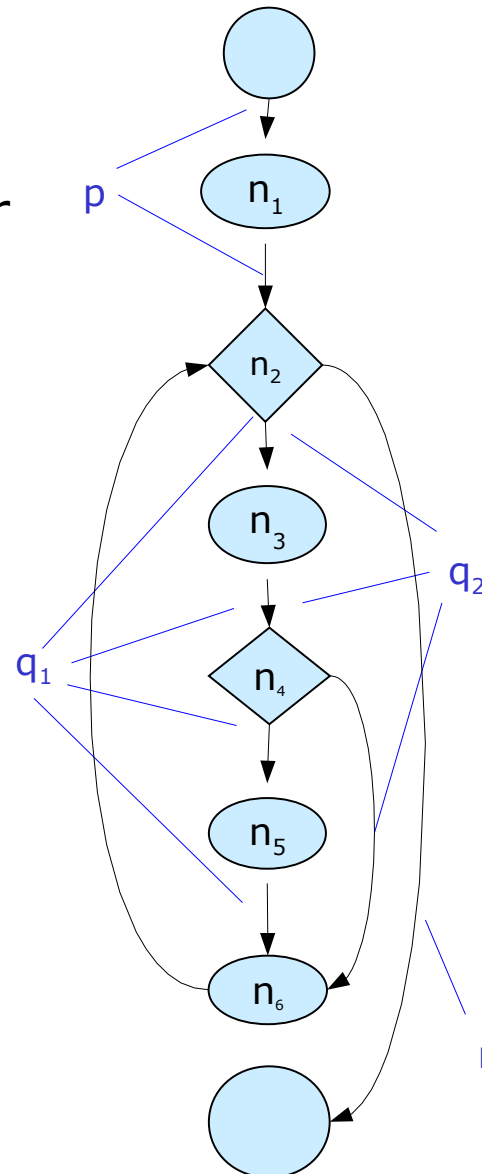
Brute force: Testbeispiele zur Ausführung **aller** unterschiedlichen Pfade im Programm

Beispiel:

Jeder Schleifendurchlauf trägt entweder zu Vokal oder zu Konsonant bei.

Pfade stehen in eindeutiger Korrespondenz zu den Worten über dem Alphabet $\{q_1, q_2\}$.

Anzahl der Testpfade bei Beschränkung auf N Schleifendurchläufe ist also $2^N - 1$.



```
i=0;  
Zchn=s.charAt(i++)
```

```
while(i<s.length())
```

```
Gesamtzahl += 1;
```

```
if ((Zchn == 'A') ||  
    (Zchn == 'E') || (Zchn == 'I') ||  
    (Zchn == 'O') || (Zchn == 'U'))
```

```
VokalAnzahl += 1;
```

```
Zchn=s.charAt(i++);
```

Allgemeiner Pfadüberdeckungstest

Eigenschaften:

- Pfadanzahl bei unbestimmten Wiederholungen (while, ...) nicht beschränkt.
- Ein Teil der konstruierbaren Pfade ist nicht ausführbar, da sich Bedingungen gegenseitig ausschließen können.

Leistungsfähigkeit:

- *Mächtigstes* kontrollflussorientiertes Testverfahren
- In einer vergleichenden Studie [Howden 78a, b, c] Erkennung von 18 von 28 Fehlern, um den Faktor 3 höhere Erkennung als Zweigüberdeckung
- Höhere Erfolgsquote nur durch Kombination mit anderen Verfahren

Bewertung:

- Praktische Bedeutung höchstens für Programmteile ohne Schleifen

Boundary Interior Test

- **Idee:** Unterscheide Durchlauf nur einmal, einmal, mehrmals
- Eingeschränkter Pfadüberdeckungstest, für Programme ohne Schleifen sogar identisch
- Schleifenabweisungstest sowie zwei Gruppen von Pfaden für jede Schleife im Programm:
 - Grenztest-Gruppe (*boundary tests*):
Pfade, welche die Schleife nur einmal durchlaufen
Testet Kombination `init -- body`
 - Gruppe zum Test der Reinitialisierung (*interior tests*):
Pfade, welche die Schleife mindestens einmal wiederholen
Testet Kombination `reinit -- body`
- Praktisch anwendbar (im Gegensatz zum allgemeinen Pfadüberdeckungstest)
- **Strukturierter Pfadtest** als Verallgemeinerung

Bedingungsüberdeckungstest

- **Ziel:** Analysiert und überprüft die Testbedingungen in Schleifen und Verzweigungen aus struktureller Perspektive
- **Ansatz:** Bedingung ist logische Verknüpfung atomarer boolescher Funktionen. Testfälle sollen verschiedene Kombinationen dieser atomaren Werte überdecken. Analyse des Termbaums.

- 3 verschiedene Varianten:

Einfache Bedingungsüberdeckung:

Überdeckt alle atomaren Werte einzeln (im Extremfall 2^b Testfälle, b = Anzahl der Blätter im Termbaum)

Mehrfach-Bedingungsüberdeckung:

Überdeckt alle möglichen Kombinationen atomarer Werte (im Extremfall 2^b Testfälle)

Minimale Mehrfach-Bedingungsüberdeckung:

Jede Bedingung (ob atomar oder nicht) muss für sich überdeckt sein (im Extremfall 2^k Testfälle, k = Anzahl der Knoten im Termbaum)

Bewertung:

- Einfache Bedingungsüberdeckung:
 - weder Zweig- noch Anweisungsüberdeckung enthalten
 - sehr schwaches Kriterium
- Mehrfach-Bedingungsüberdeckung:
 - Zweigüberdeckung ist enthalten, jedoch sehr aufwändig
- Minimale Mehrfach-Bedingungsüberdeckung:
 - Wie einfache BÜ, aber beachtet die hierarchische Struktur
 - Es müssen nicht nur die Blätter (Atome), sondern auch alle Teilbäume des Ausdrucksbaums überdeckt sein
 - Sinnvolle Weiterentwicklung des Zweigtests (entspricht Überdeckung des Wurzelknotens)

Auswahl geeigneter kontrollflussorientierter Testverfahren

- Liegt Programm im Quellcode vor?
 - Nein: Kein Strukturtestverfahren möglich
- Besteht Programm nur aus Anweisungen?
 - Anweisungsüberdeckung sinnvoll
- .. nur Anweisungen und Verzweigungen mit atomaren Testbedingungen
 - Zweigüberdeckung sinnvoll
- .. Anweisungen, Verzweigungen und Schleifen mit atomaren Testbedingungen
 - Pfadüberdeckung, je nach Komplexität der Schleifensemantik doppelte oder mehrfache Schleifenüberdeckung
- ... komplexe Testbedingungen
 - Kopplung geeigneter Verfahren mit Bedingungsüberdeckung

Datenflussorientierte Strukturtestverfahren

- Ebenfalls dynamisches Strukturtestverfahren
- Im Gegensatz zu kontrollflussorientierten Verfahren werden Datenbenutzungen und damit eher globale Aspekte getestet.
- Analyse der Programmdynamik an Hand der Dynamik der in Variablenwerten gespeicherten Programmzustände
 - Aufstellen des Datenflussdiagramms
 - Sichtbarkeit und Lebensdauer von Bezeichnern
 - Variablenidentitäten: Gültigkeitskontext und Kontrollfluss
 - lesende und schreibende Zugriffe auf Variablen (use, def)
 - Unterscheidung lesender Zugriffe in Anweisungen und Bedingungen (c-use, p-use)
- Eignen sich besonders für den Test von Datenobjekt- und Datentypmodulen sowie Klassen.
- Nur wenige Testwerkzeuge vorhanden.

Defs/Uses-Verfahren

Klassifikation der Variablenzugriffe nach

- Zuweisung (set-Methode, Definition, *def*)
 - Variablenwert wird an einer solchen Stelle geändert
- Zugriff zur Berechnung von anderen Werten (*computational-use*, *c-use*)
 - Auswirkung auf Wert anderer Variablen (Programmzustand)
- Zugriff zur Berechnung von Wahrheitswerten in Bedingungen (*predicate-use*, *p-use*)
 - Auswirkung auf Wert der Testbedingung (Kontrollfluss)

3. Datenflussorientierte Strukturtests

