Software-Test

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Qualitätssicherung

Wie stellt man sicher, dass die geforderte Qualität erreicht wird?

Es genügt nicht, allgemeine Qualitätskriterien aufzustellen – man muss auch sicherstellen, dass diese Qualität tatsächlich erreicht wird.







Maßnahmen der Qualitätssicherung

Konstruktive Maßnahmen sorgen *a priori* für bestimmte Eigenschaften des Produkts oder des Entwicklungsprozesses:

- Gliederungsschema für Pflichtenhefte
- Einsatz von Programmiersprachen mit statischer Typprüfung
- Richtlinien für den Entwicklungsprozess

Analytische Maßnahmen diagnostizieren *a posteriori* die Qualität des Produkts oder des Entwicklungsprozesses:

- Programmverifikation
- Programminspektion
- Software-Tests

Grundprinzipien

- Unabhängige Qualitätszielbestimmung
- Quantitative Qualitätssicherung
- Maximale konstruktive Qualitätssicherung
- Frühzeitige Fehlerentdeckung und -behebung
- Entwicklungsbegleitende Qualitätssicherung
- Unabhängige Qualitätssicherung







Unabhängige Qualitätszielbestimmung

Prinzip der unabhängigen Qualitätszielbestimmung:
Jedes Software-Produkt soll nach seiner Fertigstellung eine *zuvor bestimmte* Qualität besitzen – unabhängig von Prozess oder Produkt.

Kunden und Lieferanten sollen gemeinsam Qualitatsziel bestimmen

Ziel: Explizite und transparente Qualitätsbestimmung *vor* Entwicklungsbeginn!

Quantitative Qualitätssicherung

Prinzip der quantitativen Qualitätssicherung:
„Ingenieursmäßige Qualitätssicherung ist undenkbar ohne die Quantifizierung von Soll- und Ist-Werten.“ (Rombach)

Einsatz von *Metriken* zur Qualitätsbestimmung

Ziel: Qualitätssteigerung *messbar* machen!







Maximale konstruktive Qualitätssicherung

Prinzip der maximalen konstruktiven Qualitätssicherung
„Vorbeugen ist besser als heilen“ (Volksmund)

FORTRAN hat Mängel in der konstruktiven Qualitätssicherung: Der Tippfehler `DO 3 I = 1.3` statt `DO 3 I = 1,3` führte 1962 zur Zuweisung von `1.3` an `D03I` und zum Verlust der amerikanischen Venussonde Mariner-1.

Ziel: Frühzeitig Fehler vermeiden helfen (durch Einsatz klarer Spezifikationen, geeigneter Programmiersprachen usw.)!







Frühzeitige Fehlerentdeckung und -behebung

Prinzip der frühzeitigen Fehlerentdeckung und -behebung:
„Je früher ein Fehler entdeckt wird, desto kostengünstiger kann er behoben werden“

Vergleiche Prozessmodelle – Kosten der Fehlerbehebung *verzehnfachen* sich pro Phase

Ziel: Fehler müssen *so früh wie möglich* erkannt und behoben werden!







Entwicklungsbegleitende Qualitätssicherung

Prinzip der entwicklungsbegleitenden Qualitätssicherung:
Jeder Schritt, jedes Dokument im Entwicklungsprozess ist der Qualitätssicherung unterworfen.

Ziel: Qualitätssicherung *in jedem Schritt* der Software-Entwicklung!







Unabhängige Qualitätssicherung

Prinzip der unabhängigen Qualitätssicherung

„Testing is a *destructive* process, even a sadistic process“ (Myers)

Derjenige, der ein Produkt definiert, entwirft und implementiert, ist am schlechtesten geeignet, die Ergebnisse seiner Tätigkeit destruktiv zu betrachten.

Ziel: eine *unabhängige und eigenständige* organisatorische Einheit „Qualitätssicherung“!







Software-Tests

Die analytische Qualitätssicherung stützt sich im allgemeinen auf den Begriff *Fehler* ab:

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

Ziel des *Testens* ist, durch gezielte Programmausführung Fehler zu erkennen.

Program testing can be used to show the presence of bugs, but never to show their absence. (Dijkstra)







Was ist ein „Fehler“?

Das deutsche Wort *Fehler* wird für verschiedene Dinge gebraucht:

1. Der Programmierer macht einen *Fehler*...
2. ...und hinterlässt einen *Fehler* im Programmcode.
3. Wird dieser ausgeführt, haben wir einen *Fehler* im Programmzustand...
4. der sich als ein *Fehler* nach außen manifestiert.

Im Englischen spielt diese Rolle das Wort „Bug“
(= Käfer, Gespenst)







Was ist ein „Fehler“? (2)

Diese unterschiedlichen „Fehler“ werden besser so unterschieden:

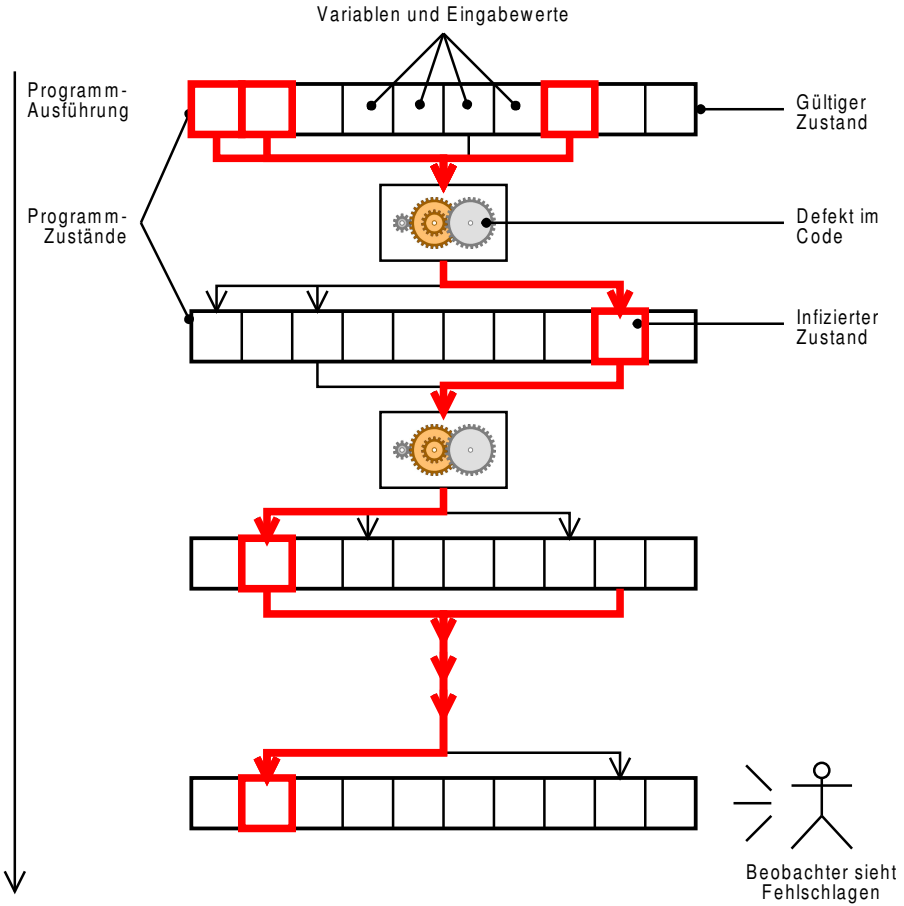
1. Der Programmierer begeht einen *Irrtum* (mistake). . .
2. . . . und hinterläßt einen *Defekt* (defect) im Programmcode.
3. Wird dieser ausgeführt, haben wir eine *Infektion* im Programmzustand. . .
4. der sich als ein *Fehlschlagen* (failure) nach außen manifestiert.







Vom Defekt zum Fehlschlagen



Vom Defekt zum Fehlschlagen (2)

So kommt es vom Defekt zum Fehlschlagen:

1. Der Programmierer begeht einen *Irrtum* (mistake)...
2. ...und hinterläßt einen *Defekt* (defect) im Programmcode.
3. Wird dieser ausgeführt, haben wir eine *Infektion* im Programmzustand...
4. der sich als ein *Fehlschlagen* (failure) nach außen manifestiert.

Es gilt: Fehlschlagen ⇒ Infektion ⇒ Defekt (⇒ Irrtum) ...

...aber nicht umgekehrt – nicht jeder Defekt führt zu einem Fehlschlagen!

Dies ist das *Kernproblem des Testens!*

Vom Fehlschlagen zum Defekt

Beispiel – Berechnung des Maximums dreier Zahlen:

```
int max3(int x, int y, int z) {
    int ret;
    if (x > y)
        ret = x;
    else
        ret = max(y, z);
    return ret;
}
```

Fehlschlagen `max3(5, 2, 9)` liefert 5

Infektion `ret` hat den Wert 5

Defekt statt `ret = x` muss es `ret = max(x, z)` heißen

Debugging = Schließen von Fehlschlagen auf Defekt







Welche Testfälle auswählen?

Ich kann nur eine beschränkte Zahl von Läufen testen – welche soll ich wählen?

Funktionale Verfahren Auswahl nach *Eigenschaften der Eingabe* oder der Spezifikation

Strukturtests Auswahl nach *Aufbau des Programms*

Ziel im Strukturtest – hohen *Überdeckungsgrad* erreichen:

⇒ Testfälle sollen möglichst viele Aspekte der Programmstruktur abdecken (Kontrollfluss, Datenfluss)







Anwendung: Zeichen zählen

Das Programm `zaehlezchn` soll

- Zeichen von der Tastatur einlesen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
- Die Zahl der eingelesenen Zeichen und Vokale ausgeben

```
$ zaehlezchn
Bitte Zeichen eingeben: HALLELUJA!
Anzahl Vokale:  4
Anzahl Zeichen: 9
$ _
```







Zeichen zählen – Benutzung


```
#include <iostream>
#include <limits>

using namespace std;

void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl);

int main()
{
    int AnzahlVokale = 0;
    int AnzahlZchn   = 0;
    cout << "Bitte Zeichen eingeben: ";
    ZaehleZchn(AnzahlVokale, AnzahlZchn);
    cout << "Anzahl Vokale:  " << AnzahlVokale << endl;
    cout << "Anzahl Zeichen: " << AnzahlZchn << endl;
}
```

Zeichen zählen – Realisierung

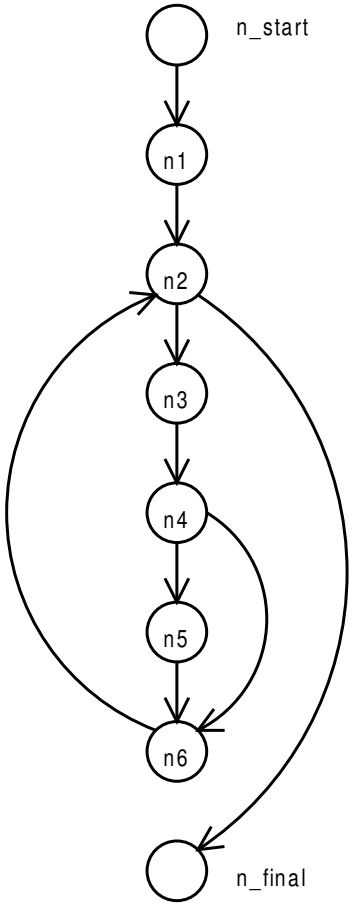

```
void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl)
{
    char Zchn;
    cin >> Zchn;
    while (Zchn >= 'A' && Zchn <= 'Z' &&
           Gesamtzahl < INT_MAX)
    {
        Gesamtzahl = Gesamtzahl + 1;
        if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
            Zchn == 'O' || Zchn == 'U')
        {
            VokalAnzahl = VokalAnzahl + 1;
        }
        cin >> Zchn;
    }
}
```







Kontrollflussgraph




```
cin >> Zchn;


while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)


Gesamtzahl = Gesamtzahl + 1;


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

VokalAnzahl = VokalAnzahl + 1;


cin >> Zchn;
```

Einfache Überdeckungsmaße

Eine Menge von Testfällen kann folgende Kriterien erfüllen:

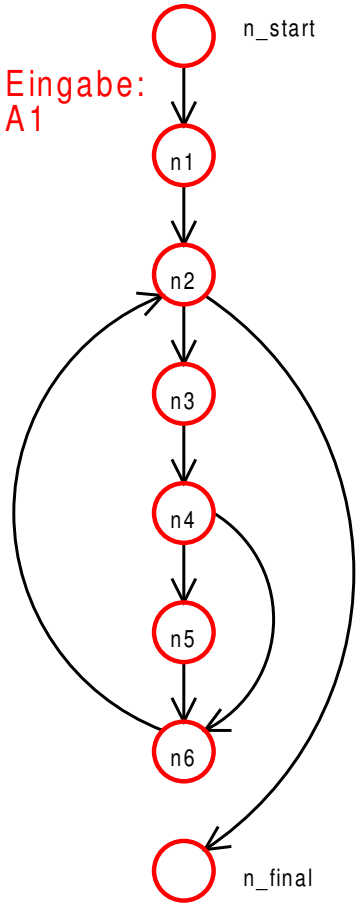
Anweisungsüberdeckung Jeder Knoten im Kontrollflussgraph muss einmal durchlaufen werden (= jede Anweisung wird wenigstens einmal ausgeführt)

Zweigüberdeckung Jede Kante im Kontrollflussgraph muss einmal durchlaufen werden; schließt Anweisungsüberdeckung ein

Pfadüberdeckung Jeder *Pfad* im Kontrollflussgraphen muss einmal durchlaufen werden

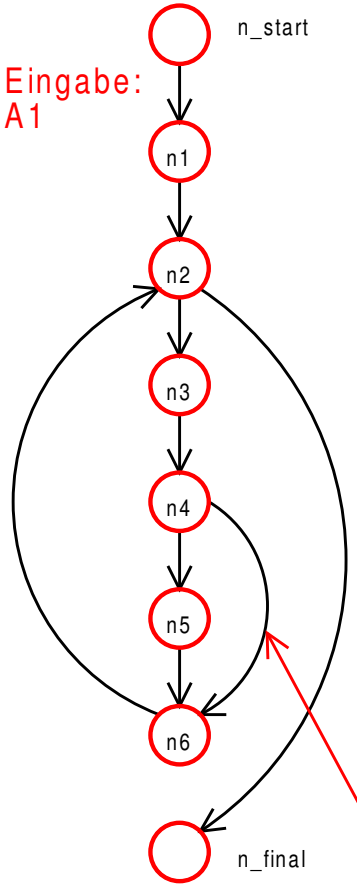
Der *Überdeckungsgrad* gibt an, wieviele Anweisungen / Zweige / Pfade durchlaufen wurden.

Anweisungsüberdeckung (C_0)



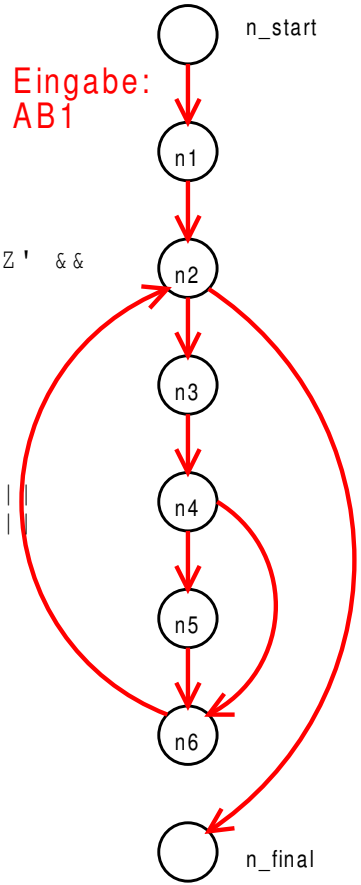

```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

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

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Zweigüberdeckung (C_1)




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

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

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Zweig (n4, n6)
wird nicht notwendig ausgeführt



Überdeckung messen mit GCOV

GCOV (GNU COVerage tool) ist ein Werkzeug, um die Anweisungs- und Zweigüberdeckung von C/C++-Programmen zu messen.

Dokumentation: `man gcov`

```
$ c++ -g -fprofile-arcs -ftest-coverage
        -o zaehlezchn zaehlezchn.C
$ ./zaehlezchn
Bitte Zeichen eingeben: KFZ.
Anzahl Vokale:  0
Anzahl Zeichen: 3
$ gcov zaehlezchn
 93.33% of 15 source lines executed in file zaehlezchn.C
Creating zaehlezchn.C.gcov.
```







Anweisungsüberdeckung messen


```
    void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl)
    {
1       char Zchn;
1       cin >> Zchn;
4       while (Zchn >= 'A' && Zchn <= 'Z' &&
                  Gesamtzahl < INT_MAX)
        {
3           Gesamtzahl = Gesamtzahl + 1;
3           if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
                Zchn == 'O' || Zchn == 'U')
            {
####            VokalAnzahl = VokalAnzahl + 1;
            }
3           cin >> Zchn;
        }
    }
```







Zweigüberdeckung messen


```
$ c++ -g -fprofile-arcs -ftest-coverage
        -o zaehlezchn zaehlezchn.C
$ ./zaehlezchn
Bitte Zeichen eingeben: KFZ.
Anzahl Vokale:  0
Anzahl Zeichen: 3
$ gcov zaehlezchn
 93.33% of 15 source lines executed in file zaehlezchn.C
Creating zaehlezchn.C.gcov.
$ gcov -b zaehlezchn
 93.33% of 15 source lines executed in file zaehlezchn.C
 90.91% of 11 branches executed in file zaehlezchn.C
 36.36% of 11 branches taken at least once in file zaehlezchn.C
100.00% of 10 calls executed in file zaehlezchn.C
Creating zaehlezchn.C.gcov.
```

Zweigüberdeckung messen (2)


```
    3          Gesamtzahl = Gesamtzahl + 1;
    3          if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
                   Zchn == 'O' || Zchn == 'U')
branch 0 taken = 0%
branch 1 taken = 0%
branch 2 taken = 0%
branch 3 taken = 0%
branch 4 taken = 0%
branch 5 taken = 100%
```

Branch 0–4 sind die Einzelbedingungen; Branch 5 ist der `else`-Fall.

Inkrementelles Messen


```
$ ./zaehlezchn
Bitte Zeichen eingeben: KFZ.
Anzahl Vokale:  0
Anzahl Zeichen: 3
$ gcov -b zaehlezchn
 93.33% of 15 source lines executed in file zaehlezchn.C
 36.36% of 11 branches taken at least once in file zaehlezchn.C
Creating zaehlezchn.C.gcov.
$ ./zaehlezchn
Bitte Zeichen eingeben: HUGO.
Anzahl Vokale:  2
Anzahl Zeichen: 4
$ gcov -b zaehlezchn
100.00% of 15 source lines executed in file zaehlezchn.C
 54.55% of 11 branches taken at least once in file zaehlezchn.C
Creating zaehlezchn.C.gcov.
```

Anweisungs- und Zweigüberdeckung

Anweisungsüberdeckung

- Notwendiges, aber nicht hinreichendes Testkriterium
- Kann Code finden, der nicht ausführbar ist
- Als eigenständiges Testverfahren nicht geeignet
- Fehleridentifizierungsquote: 18%

Zweigüberdeckung

- gilt als *das* minimale Testkriterium
- kann nicht ausführbare Programmzweige finden
- kann häufig durchlaufene Programmzweige finden (Optimierung)
- Fehleridentifikationsquote: 34%

Schwächen der Anweisungüberdeckung

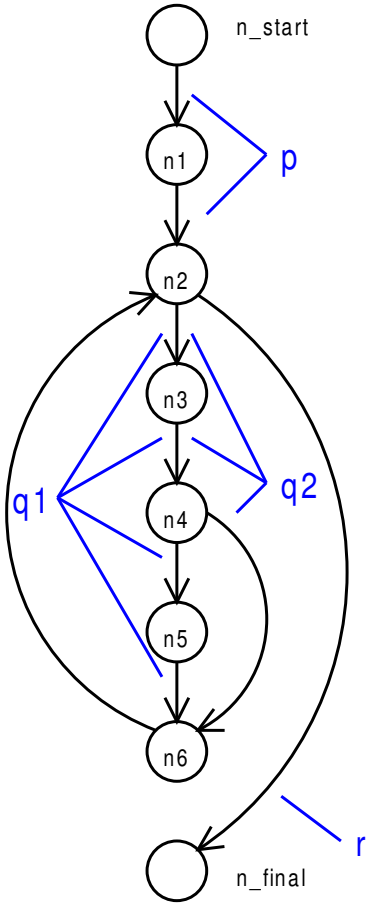
Warum reicht Anweisungsüberdeckung nicht aus?

Wir betrachten den folgenden Code:

```
x = 1;
if (x >= 1)          // statt y >= 1
    x = x + 1;
```

Hier wird zwar jede Anweisung einmal ausgeführt; die Zweigüberdeckung fordert aber auch die Suche nach einer Alternative (was hier schwerfällt $\Rightarrow$ Defekt).

Pfadüberdeckung




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

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

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Pfade $P = p(q_1|q_2)^*r$

$\Rightarrow |P| = 2^{INT\_MAX} - 1$

Pfadüberdeckung (2)

Pfadüberdeckung

- mächtigstes kontrollstrukturorientiertes Testverfahren
- Höchste Fehleridentifizierungsquote
- keine praktische Bedeutung,
  da Durchführbarkeit sehr eingeschränkt

Wichtige Variante: *Strukturierter Pfadtest* (jede Schleife wird wenigstens k-mal durchlaufen)

- Erlaubt die gezielte Überprüfung von Schleifen
- Überprüft zusätzlich Zweigkombinationen
- Im Gegensatz zum Pfadüberdeckungstest praktikabel
- Fehleridentifikationsquote: um 65% (Strukturierter Pfadtest)

Bedingungsüberdeckung

Wir betrachten die Bedingungen aus ZaehleZchn:

```
while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)                    (A)

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

Die *Struktur der Bedingungen* wird vom Zweigüberdeckungstest nicht geeignet beachtet

⇒ *Bedingungsüberdeckungstests*

Minimale Mehrfach-Bedingungsüberdeckung

Wichtigster Vertreter der Bedingungsüberdeckungstests:

- Jede *atomare Bedingung* muss wenigstens einmal `true` und `false` sein.
- Die *Gesamt-Bedingung* muss wenigstens einmal `true` und wenigstens einmal `false` werden.
- Schließt Zweigüberdeckung (und somit Anweisungsüberdeckung) ein
- $\Rightarrow$ realistischer Kompromiss





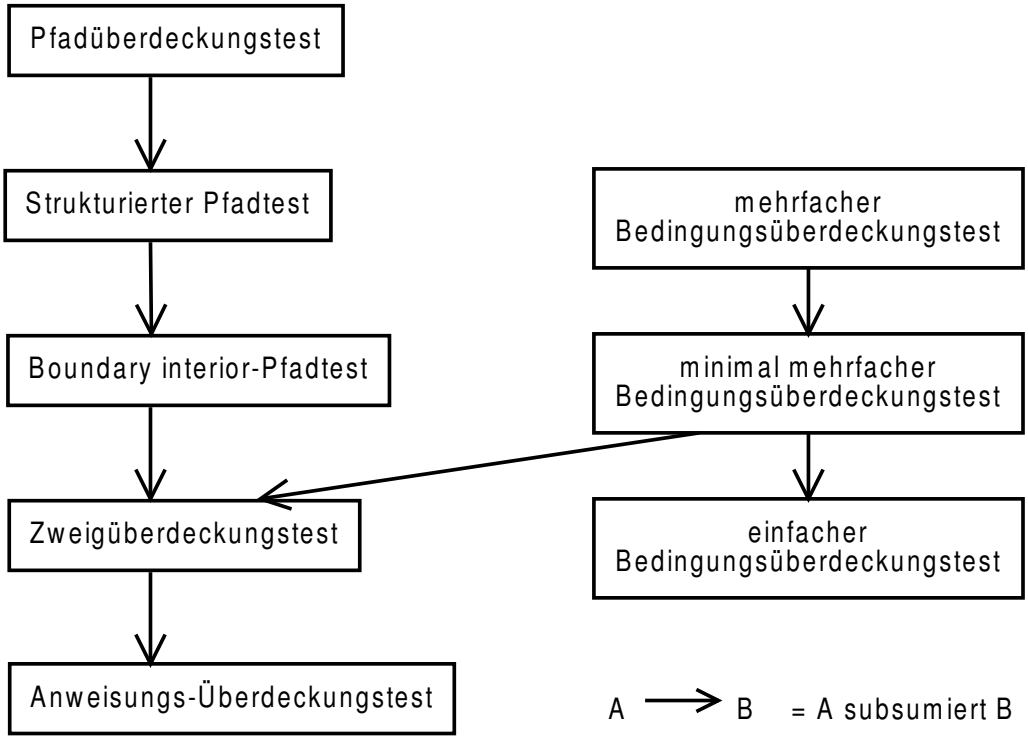


Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muss wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1							2	3
`Gesamtzahl`	0	1	2	3	4	5	6	0	INT_MAX
`Zchn`	'A'	'E'	'I'	'O'	'U'	'B'	'l'	'a'	'D'
`Zchn >= 'A'`	**T**	T	T	T	T	T	**F**	T	T
`Zchn <= 'Z'`	**T**	T	T	T	T	T	T	**F**	T
`Gesamtzahl < INT_MAX`	**T**	T	T	T	T	T	T	T	**F**
Bedingung (A)	**T**	T	T	T	T	T	**F**	F	F
`Zchn == 'A'`	**T**	**F**	F	F	F	F	–	–	–
`Zchn == 'E'`	**F**	**T**	F	F	F	F	–	–	–
`Zchn == 'I'`	**F**	F	**T**	F	F	F	–	–	–
`Zchn == 'O'`	**F**	F	F	**T**	F	F	–	–	–
`Zchn == 'U'`	**F**	F	F	F	**T**	F	–	–	–
Bedingung (B)	**T**	T	T	T	T	**F**	–	–	–

Verfahren im Überblick



Weitere Testverfahren im Hauptstudium!

Checkliste: Testbericht

Wir erwarten

- eine *Beschreibung der Tests* (analog zur exemplarischen Spezifikation)
- eine *Bewertung der Testgüte* als Abdeckung:
  - Abgedeckte Funktionalität im Pflichtenheft (normal 100%)
  - Abgedeckte Methoden des Programms (Ziel: 100%)
  - wenn technisch möglich (z.B. mit gcov): Zweig- und Anweisungsüberdeckung (Ziel: weniger als 100%)
- eine *Beschreibung der Testergebnisse* (d.h. wieviele Tests waren erfolgreich; wieviele sind fehlgeschlagen?)