

**6.1 Fehlerquellen**

**6.2 Ausnahmebehandlung allgemein**

**6.3 Ausnahmen in Java**

**6.4 Ausnahmebehandlung in Java**

**6.5 Eigene Ausnahmeklassen**

**6.6 Testen von Programmen**



## Innensicht

### **Imperatives Programmieren**

#### **(Konstrukte aus Java)**

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

### **Objektorientiertes Programmieren**

#### **(Java)**

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

## Außensicht

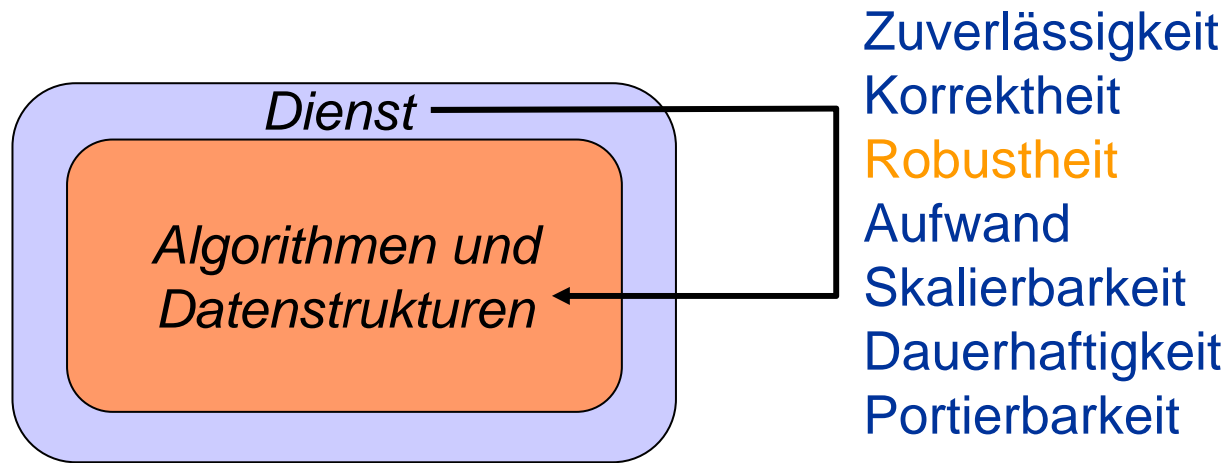
### **Dienste**

- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

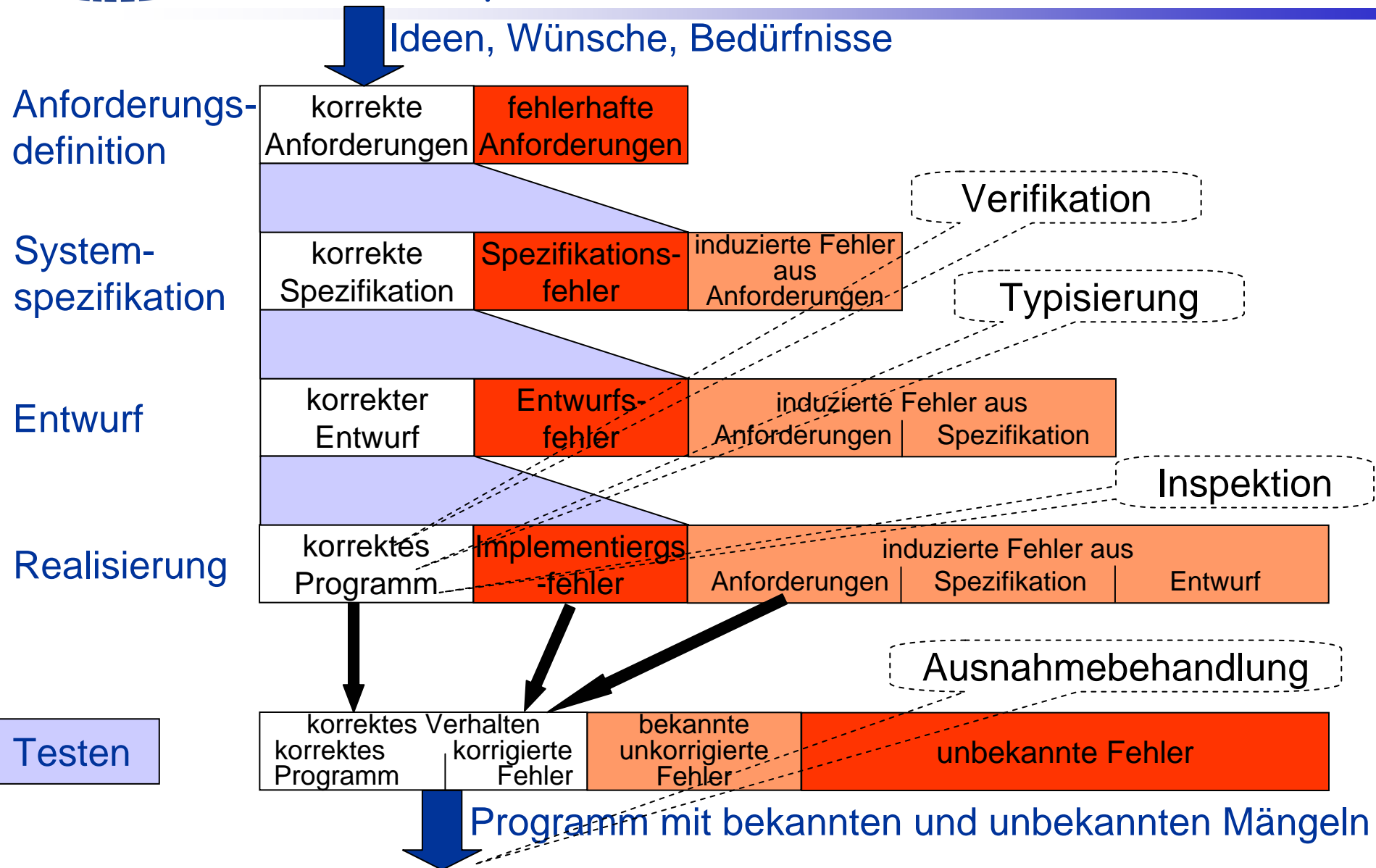
### **Objektorientierung**

- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung





# 6.1 Fehlerquellen



### Bisher: Sicherstellung von Korrektheit durch vorbeugende Maßnahmen

- zum Zeitpunkt der Implementierung mittels Zusicherungskalkül
- zur Übersetzungszeit mittels Typisierung

### Nunmehr: Wiederherstellen von Korrektheit durch korrigierende Maßnahmen während der Laufzeit:

- Erkennen einer fehlerhaften Situation. Dies muss das Programm (bzw. die Umgebung, in der es abläuft) selbst tun.
- Signalisieren der Situation an Programm oder Benutzer
- Spezielle Behandlung der Situation
- Dadurch geordnetes Beenden oder Fortführen des Programms



Eine Ausnahme (exception) ist ein Ereignis, durch das die normale Ausführungsreihenfolge unterbrochen wird und dessen Auftreten nicht durch das Programm beeinflusst werden kann.

- Erste Aufgabe: Erkennung von Ausnahmen
  - Standardisierte Ausnahmen bei häufigen Übergabefehlern  
z.B. Division durch 0, arithmetischer Überlauf, ...
  - Anwendungsspezifische Ausnahmen  
z.B. Kontenunterdeckung in einer Bank, ...
  - Ressourcenprobleme  
z.B. Gerätefehler, ...
- Zweite Aufgabe: Behandlung von Ausnahmen
  - Geordnetes Beenden oder Fortführen des Programms bei Auftritt einer Ausnahme
  - führt zur Ausnahmebehandlung (exception handling)



Man kann nicht jede einzelne Ausnahme vorhersagen.

- Man kann jedoch die Arten an Ausnahmen vorhersagen. Daher werden Ausnahmen klassifiziert.
- Jeder Ausnahme wird dazu ein Ausnahmetyp zugeordnet.

Die Behandlung orientiert sich dann am Ausnahmetyp.

- Dies begründet die Trennung von Erkennung und Behandlung.



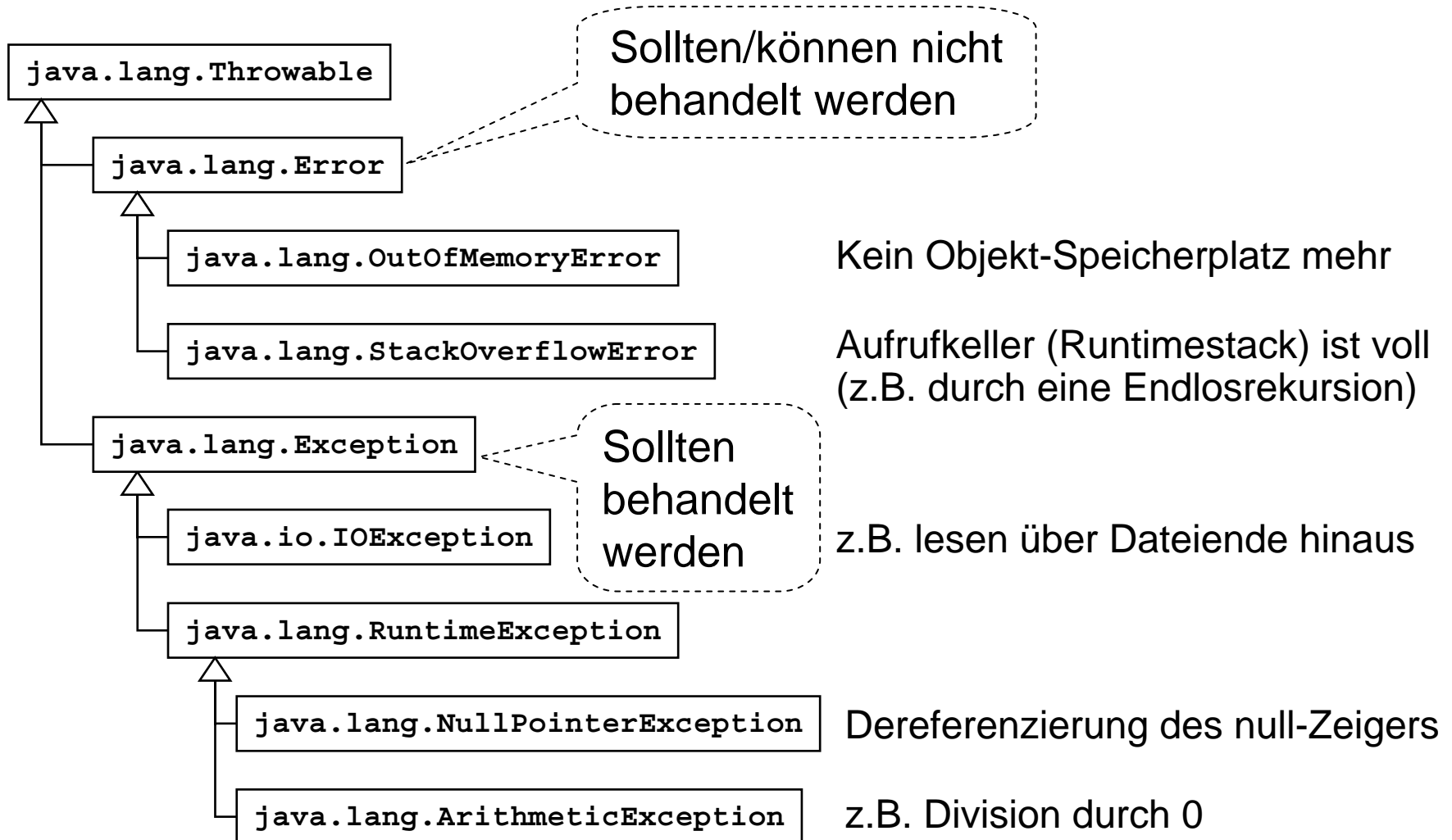
### Klassifizierung von Ausnahmen in Java

- Ausnahmetypen sind in Java gewöhnliche Klassen
- Sie erben aber alle von der speziellen Klasse `java.lang.Throwable`.

`java.lang.Throwable` hat (genau) 2 direkte Unterklassen:

- `java.lang.Error` zeigt Probleme bei der Programmausführung an, die das Programm nicht selbst beheben kann.
- `java.lang.Exception` ist die Oberklasse, unter der alle Ausnahmetypen zusammengefasst werden, die das Programm selbst behandeln möchte/sollte.





Erkennen einer Ausnahmesituation



Erzeugen einer Ausnahme von einem Typ



Anstoßen der Ausnahmebehandlung



Behandeln der Ausnahme gemäß Typ



- Bezeichnet man als das „Auslösen von Ausnahmen“ (to raise or to throw an exception).
- Das Erzeugen einer Ausnahme ist eine gewöhnliche Objekterzeugung mit `new` bzgl. der Ausnahmeklasse.
- Das Ausnahmeobjekt wird mit der Anweisung `throw <ExceptionExpr>` ausgelöst.

## Beispiel:

```
public void printMe(String p) {  
    if (p == null) {  
        throw new NullPointerException("Parameter p was null");  
    }  
    System.out.println(p);  
}
```

Erkennen

Erzeugen



- Die ausgeführte Methode wird sofort abgebrochen.
- Das Ausnahmeobjekt wird als eine Art Methodenresultat an die aufrufende Methode zurückgegeben.
- Entweder wird dort die Ausnahme behandelt oder die Ausnahme wird analog zur nächsten Aufrufermethode weitergegeben und so fort.



```
public class ExceptionTest {  
    public static void main(String[] args) {  
        System.out.println("before m();");  
        m();  
        System.out.println("after m();");  
    }  
  
    public static void m() {  
        System.out.println("before n();");  
        n();  
        System.out.println("after n();");  
    }  
  
    public static void n() {  
        System.out.println("before throw ...");  
        throw new RuntimeException("Let's raise this!");  
    }  
}
```



Wir führen das Programm aus und erhalten die folgende Ausgabe:

```
> java ExceptionTest
before m();
before n();
before throw ...
java.lang.RuntimeException: Let's throw this!
    at ExceptionTest.n(ExceptionTest.java:16)
    at ExceptionTest.m(ExceptionTest.java:10)
    at ExceptionTest.main(ExceptionTest.java:4)
```



- Wenn man eine ausgelöste Ausnahme nicht behandelt, führt sie schließlich zum Programmabbruch mit einer Fehlerausgabe:
  - In `m()` und `main()` wurde die ausgelöste Ausnahme nicht behandelt.
- Nach dem Verlassen von `main()` macht der Java-Interpreter eine Ausgabe, die berichtet, wo welche Ausnahme ausgelöst wurde:
  - Dies erleichtert die Fehlersuche, da man die Spur der Ausnahme über die aufrufenden Methoden mit Zeilennummern genau verfolgen kann.
  - Auch die Meldung, die in die Ausnahme eingebettet wurde, wird angezeigt.



### Behandeln der Ausnahme gemäß Typ:

- Mit einer **try-catch**-Anweisung:

```
try-Block { try {  
    // Mach` Sachen, die evtl. eine Ausnahme auslösen ...  
}  
catch-Klausel { catch (<ExceptionClass1> <ExceptionVar1>) {  
    // Handle die Ausnahme vom Typ <ExceptionClass1>.  
    // Sie ist an die Variable <ExceptionVar1> gebunden.  
}  
    catch (<ExceptionClass2> <ExceptionVar2>) {  
        // Handle die Ausnahme vom Typ <ExceptionClass2>.  
        // Sie ist an die Variable <ExceptionVar2> gebunden.  
    } ...  
}
```

Man kann beliebig viele **catch**-Klauseln angeben.



Wird in einem **try**-Block eine Ausnahme ausgelöst, dann...

- ...wird die Ausführung des **try**-Blocks abgebrochen und
- ...die **catch**-Klauseln werden von oben nach unten „durchprobiert“, ob das ausgelöste Ausnahmeobjekt eine Instanz eines dort angegebenen Typs ist.
- Passt der Typ oder ist er ein Obertyp, wird der Code in der entsprechenden **catch**-Klausel ausgeführt und danach geht die Ausführung mit dem Code nach der **try-catch**-Anweisung weiter.
- Passt keiner der **catch**-Blöcke, wird die Ausführung der **try-catch**-Anweisung abgebrochen und die Ausnahme an den Aufrufer übergeben.



```
public class ExceptionTest2 {  
    public static void main(String[] args) {  
        System.out.println("before m();");  
        try {  
            m();  
            System.out.println("after m();");  
        } catch (NullPointerException e1) {  
            System.out.println("This statement will not be reached.");  
        } catch (RuntimeException e2) {  
            System.out.println("This one catches our exception...");  
            System.out.println("And here it is: " + e2);  
        }  
        System.out.println("after try();");  
    }  
  
    public static void m() {  
        System.out.println("before n();");  
        n();  
        System.out.println("after n();");  
    }  
  
    public static void n() {  
        System.out.println("before throw ...");  
        throw new RuntimeException("Let's throw this!");  
    }  
}
```



- Wir führen das Programm `ExceptionTest2` aus und erhalten die folgende Ausgabe:

```
> java ExceptionTest2
```

```
before m();
```

```
before n();
```

```
before throw ...
```

```
This one catches our exception...
```

```
And here it is: java.lang.RuntimeException: Let's throw this!  
after try();
```

- Man beachte, dass die zweite `catch`-Klausel ausgeführt wurde!
- `after m();` wurde immer noch nicht ausgegeben, denn innerhalb des `try`-Blocks wird genauso abgebrochen wie zuvor.
- Nach der `try-catch`-Anweisung geht's normal weiter („`after try();`“ wurde ausgegeben).



- Manchmal möchte man, dass im Anschluss an den Code im `try`-Block ein Stück Code auf jeden Fall ausgeführt wird, egal ob im `try`-Block eine Ausnahme ausgelöst wurde oder nicht.
- Dieser Code kann am Ende der `try-catch`-Anweisung nach den `catch`-Klauseln in einem `finally`-Block angegeben werden.

## Beispiel:

```
public class ExceptionTest4 {  
    public static void main(String[] args) {  
        try {  
            m();  
        } finally {  
            // Dieser Code wird auf jeden Fall ausgeführt.  
            System.out.println("Print this - no matter what!");  
        }  
    }  
  
    public static void m() {  
        throw new RuntimeException("some exception");  
    }  
}
```



## Woher weiß man, welche Ausnahmen von einer Methode ausgelöst werden können?

- Damit ein Aufrufer die richtigen Ausnahmen behandeln kann, müssen die Ausnahmen, die eine Methode auslösen kann, in deren Signatur mit angegeben werden:

```
<ModifierList> <MethodName> (<ParameterList>) // wie zuvor  
throws <ExceptionClass1>, ..., <ExceptionClass n> // neu!
```

<ExceptionClass1>, ..., <ExceptionClass n> ist die Liste der Ausnahme-Klassen von potenziellen Ausnahmen, die in der Methode ausgelöst werden.

- Beachte: Auf Ausnahmen der Klasse `java.lang.RuntimeException` oder deren Unterklassen muss man immer gefasst sein, daher ist diese Art der Deklaration nicht zwingend notwendig.



Im Prinzip genügt es, einen Obertyp einer potenziell ausgelösten Ausnahme im Methodenkopf zu deklarieren:

```
public class ExceptionTest3 {  
    public void m() throws Throwable {  
        throw new java.io.IOException("Let`s face it: this  
                                         works!");  
    }  
}
```

Es ist jedoch guter Programmierstil, die ausgelösten Methodenklassen viel spezifischer zu deklarieren:

```
public class ExceptionTest3 {  
    public void m() throws java.io.IOException {  
        throw new java.io.IOException("Let`s face it: this  
                                         works!");  
    }  
}
```

Beim Überschreiben von Methoden gilt im Zusammenhang mit Ausnahmedeclarationen im Prinzip die Kovarianzregel, d.h.:

- Eine überschreibende Methode darf weniger Ausnahmen im Methodenkopf deklarieren.
- Eine überschreibende Methode kann Untertypen der deklarierten Ausnahmen aus der überschriebenen Methoden deklarieren.
- Eine überschreibende Methode darf keine Obertypen der deklarierten Ausnahmen aus den überschriebenen Methoden deklarieren (Verletzung des Ersetzungsprinzips).

Beispiel:

```
public abstract class A {  
    public abstract void m() throws Exception;  
}  
abstract class B extends A {  
    public abstract void m() throws Throwable; // Das geht nicht!  
}  
abstract class C extends A {  
    public abstract void m() throws NullPointerException;  
                                                // Das geht!  
}  
abstract class D extends A {  
    public abstract void m(); // Und das geht auch!  
}
```



## Wie programmiere ich eine Ausnahmeklasse?

Die eigene Ausnahmeklasse muss von `java.lang.Throwable` erben:

- Die benutzerdefinierte Klasse sollte eine Unterklasse von `java.lang.Exception` sein.
- `java.lang.Error` sollte man nicht nutzen, da man eigene (anwendungsorientierte) Ausnahmen wohl auch behandeln können sollte.

Die Ausnahme sollte den Fehlerzustand aufnehmen können, also muss man entsprechende Attribute vorsehen.

- Standardmäßig können `java.lang.Exception`-Objekte schon eine Fehlernachricht aufnehmen, zum Beispiel:

```
throw new Exception("my exception message");
```



```
package ord.ipd.banking;
```

```
public class Account extends  
{
```

```
    public static final double
```

```
    private double balance = 0
```

```
    private double maxDebt = 0
```

```
    ...
```

```
    public void changeAmount(d
```

```
        // Making too many debts
```

```
        if (getBalance() + amount
```

```
            throw new AccountExcept
```

```
        }
```

```
        // Increasing the balance by more than
```

```
        // MAX_AMOUNT is forbidden!
```

```
        if (amount > MAX_AMOUNT) {
```

```
            throw new AccountException(amount);
```

```
        balance += amount;
```

```
    }
```

```
    public double getBalance() { return balance };
```

```
    ...
```

```
}
```

```
package ord.ipd.banking;
```

```
public class AccountException extends  
    Exception
```

```
{
```

```
    // Positive in case of deposit,
```

```
    // negative in case of withdrawal.
```

```
    private double invalidAmount;
```

```
    public AccountException(double  
        invalidAmount) {  
        this.invalidAmount =  
            invalidAmount;  
    }
```

```
    public getInvalidAmount() {  
        return invalidAmount;  
    }
```



- Testen: Überprüfen von Abläufen eines Programms unter bekannten Bedingungen mit dem Ziel des Auffindens von Fehlern. Wegen des Stichprobencharakters:
  - Dijkstra: „Testing shows the presence of bugs, but never their absence.“
- Für ein zu überprüfendes Programm (Testobjekt) werden hierbei eine Reihe von Eingabedaten und die jeweils gewünschten Reaktionen/Ausgaben (Testfälle) festgelegt.
- Anhand von Ablaufprotokollen ist ersichtlich, ob das Programm den Test bestanden hat oder nicht.
- Der Überdeckungsgrad (coverage) gibt die Vollständigkeit eines Tests bezogen auf ein bestimmtes Testkriterium an.



## Statische Testverfahren

- Inspektion
- Walkthrough
- Review (häufig als Oberbegriff für Inspektion und Walkthrough verwendet)

## Dynamische Testverfahren

- Strukturtestverfahren
  - White-Box-Test
    - Kontrollflussorientiert
    - Datenflussorientiert
- Funktionale Verfahren
  - Black Box-Test



## Allgemeine Vorgehensweise

- Überprüfung des Programmcodes (Fehlersuche) anhand von Checklisten
- Üblich: Durchsicht im Team.
- Überprüfung erfolgt nicht durch den Autor

Checklisten enthalten häufige Fehlerarten. Diese hängen von den verwendeten Programmiersprachen und deren Übersetzer ab. Moderne Sprachen haben kürzere Listen.



## Definition

- „A review process in which a [...] programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems.“ ANSI/IEEE 729-1983

## Charakteristika

- Prüfung Produkte und Teilprodukte
- Keine Verwendung von Prüfkriterien
- Pflichtenheft, Pseudocode, Quellcode ...
- Ergebnis: Protokoll
- Stichprobenartige Prüfung (nicht vollständig)
- Autor ist Moderator
- Geringer Aufwand
- Geringer Nutzen (vgl. mit Inspektion)



Balzert: "Lehrbuch Grundlagen der Informatik",  
950 Seiten, Spektrum Akademischer Verlag, ISBN: 3827403588



## Definition

- „A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problem.“ ANSI/IEEE 729-1983

## Einsatz

- Freigabe von Teilprodukten für die nächste Entwicklungsaktivität

## Charakteristika

- Verwendung von Checklisten
- Pflichtenheft, Pseudocode, Quellcode ...
- Ergebnisse
  - Formalisiertes Protokoll, Fehlerklassifizierung, Inspektionsmetriken ...
- Stichproben
- Autor ist nicht der Moderator



Balzert: "Lehrbuch Grundlagen der Informatik",  
950 Seiten, Spektrum Akademischer Verlag, ISBN: 3827403588



## Datenzugriffe

- Ungesetzte Variablen benutzt?
- Indizierung innerhalb der Grenzen?
- Nicht-ganzzahlige Indizierungen?
- Zeiger ins Leere?
- Stimmen die Strukturdefinitionen über Komponentengrenzen hinaus überein?

## Datendeklaration

- Alle Variablen vereinbart?
- Attribute `public`, `static`, `private`, `protected` richtig verstanden?
- Felder und Zeichenreihen richtig initialisiert?
- Richtige Längen und Typen zugewiesen?
- Gibt es Variablen mit ähnlich klingenden Namen?
- Sind die Namenskonventionen eingehalten?



## Vergleiche

- Vergleiche zwischen unverträglichen Variablen?
- Vergleiche zwischen verschiedenen Typen korrekt?
- Vergleichsformulierung korrekt?
- Boolsche Ausdrücke korrekt?
- Vorrang der einzelnen Operationen richtig beachtet?
- Übersetzerbedingte Berechnung von Boolschen Ausdrücken korrekt?

## Berechnungen

- Rechenausdrücke mit nicht-arithmetischen Variablen?
- Berechnungen mit verschiedenen Typen?
- Ausdrücke mit Variablen unterschiedlicher Länge?
- Über-/Unterlauf bei Zwischenergebnissen?
- Konvertierungsfehler beim Übergang zum Dualsystem?
- Variablenwerte außerhalb des Wertebereichs?
- Vorrang der einzelnen Operationen richtig beachtet?
- Ganzzahldivision korrekt?



## Kontrollfluss

- Zu viele/wenige Alternativen in mehrarmigen Verzweigungen?
- Terminiert jede Schleife?
- Terminiert jede Rekursion?
- Sind mögliche Schleifenauslassungen korrekt?
- Iterationsfehler an den Grenzen?
- Irgendwelche unvollständigen Entscheidungen?
- Gibt es unerreichbare Programmteile?

## Schnittstellen

- Stimmen aktuelle und formale Parameter in Anzahl und Typen überein?
- Stimmen aktuelle und formale Parameter in den Maßeinheiten überein?
- Stimmen aktuelle und formale Parameter in den sonstigen Attributen überein?
- Sind die Aufrufe von Bibliotheksmethoden korrekt?
- Werden Eingabeparameter verändert?
- Vereinbarung globaler Variablen über alle Komponenten gleich?



## Vorteile

- Effizientes Mittel zur Qualitätssicherung
- Verantwortung für Qualität wird vom Team getragen
- Wissensbasis der Teilnehmer wird verbreitert, das Überprüfen in einer Gruppensitzung durchgeführt wird
- Autoren müssen sich um verständliche Ausdrucksweise bemühen

## Nachteile

- In der Regel sehr aufwändig (bis zu 20 Prozent der Erstellungskosten des zu prüfenden Produkts)
- Autoren geraten in eine psychisch schwierige Situation („sitzen auf der Anklagebank“)



Balzert: "Lehrbuch Grundlagen der Informatik",  
950 Seiten, Spektrum Akademischer Verlag, ISBN: 3827403588



## Funktionstest (Black-Box-Testing)

### Testen aufgrund externer Betrachtung:

- Das äußerlich beobachtbare Verhalten (Zustandsänderung, Ausgaben) des Testobjekts wird bei bestimmten Eingaben betrachtet.
- Das Verhalten wird mit dem nach Spezifikation erwarteten verglichen.
- Innere Abläufe brauchen und sollen nicht bekannt sein.

### Vorteile:

- Testfälle können unabhängig von der Implementierung erstellt werden.
- Vermeidet Scheuklappen bei der Bestimmung der Testfälle.

### Nachteile:

- Mögliche kritische Pfade in der Implementierung sind nicht bekannt und werden vielleicht nicht getestet.

Häufig: Überprüfung spezifizierter Funktionen mit typischen Werten und Grenzfällen



## Strukturtest (White-Box-Testing)

### Testen aufgrund interner Betrachtung:

- Der Programmablauf des Testobjekts wird bei bestimmten Eingaben betrachtet, die unter Kenntnis des inneren Ablaufs so gewählt werden, dass bestimmte Testkriterien erfüllt werden.

### Vorteile:

- Alle Programmteile können getestet werden.

### Nachteile:

- Fehlende Pfade können nicht getestet werden (Fehlen wird nur durch externe Betrachtung aufgedeckt).
- Die Anzahl aller möglichen Pfade durch ein Programm ist zu hoch und lässt vollständiges Testen nicht zu.



## Strukturtests erfordern meist Instrumentierung:

- Einbau von Anweisungen in das Testobjekt, z.B. Ausgabeanweisungen oder Zuweisungen an Testvariablen, mit denen der Ablauf verfolgt werden kann.

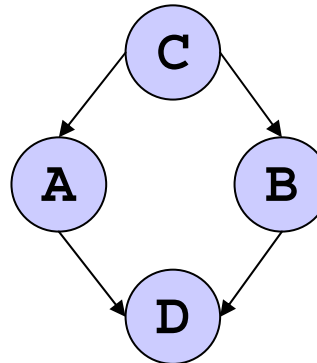
```
static int bestimmeMax (int i, int j) {  
    if (i > j){  
        System.out.println("ja durchlaufen");  
        return i;  
    }  
    else{  
        System.out.println("nein durchlaufen");  
        return j;  
    }  
}
```



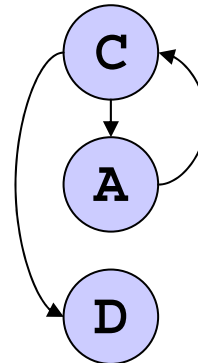
## Kontrollflussgraph (control flowgraph)

- Veranschaulichung eines Methodenrumpfs als Graph
- Anweisung werden zu Knoten im Graph
- Sprünge, Schleifen, Bedingungen induzieren die Kanten

```
if (C) A; else B; D;
```



```
while (C) A; D;
```

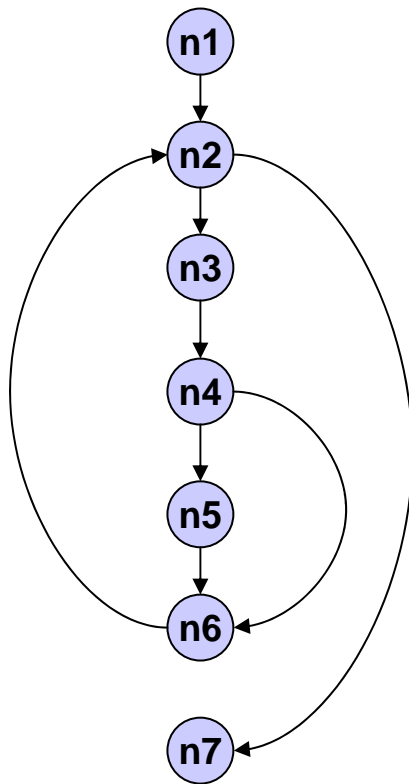


Kontrollflussgraphen können automatisch aus Programmen abgeleitet werden!

## Beispiel: Kontrollflussgraph für `countVowels()`

```
public int countVowels(char[] ca) {  
    int vowels = 0;  
    int i = 0;  
    while (i < ca.length) {  
        char c = ca[i];  
  
        if (c == 'A' || c == 'E' ||  
            c == 'I' || c == 'O' || c == 'U') {  
            vowels++;  
        }  
        i++;  
    }  
    return vowels;  
}
```





```

int vowels = 0;
int i = 0;

while(i < ca.length)

    char c = ca[i];

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

        vowels++;

    i++;

return vowels;
  
```

- Jeder Knoten soll beim Test mindestens einmal besucht werden.
- Grad der Überdeckung =  $\frac{\text{Anz. besuchte Knoten}}{\text{Anz. aller Knoten}}$

## Beispiel:

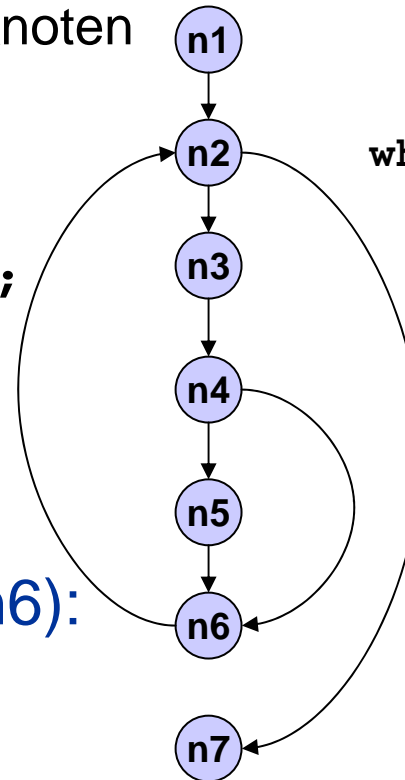
```
countVowels(new char[] { 'A' });
```

Durchlaufener Pfad:

n1, n2, n3, n4, n5, n6, n2, n7

**Testpfad enthält alle Knoten, aber nicht alle Kanten (hier n4-n6):**

- Wesentliche Aspekte des Programms werden nicht geprüft
- Niedrige Fehlerentdeckungsrate



```
int vowels = 0;
int i = 0;
```

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

```
char c = ca[i];
```

```
if (c == 'A' || ...
    c == 'U')
```

```
vowels++;
```

```
i++;
```

```
return vowels;
```



- Jede Kante soll beim Test mindestens einmal besucht werden.

## Beispiel:

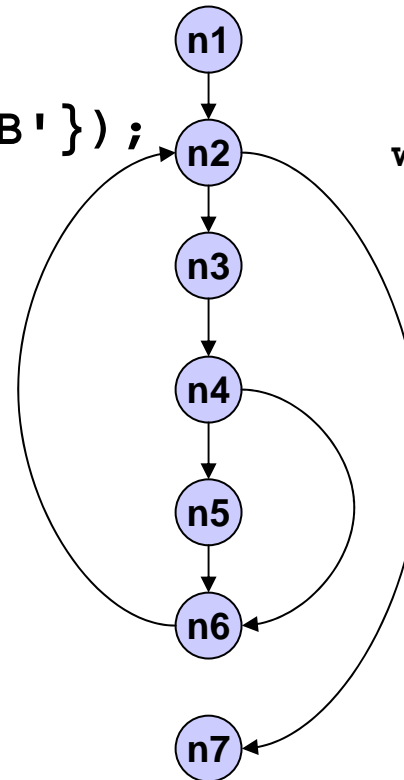
```
countVowels(new char[] { 'A', 'B' });
```

Durchlaufener Pfad:

n1, n2, n3, n4, n5, n6,  
n2, n3, n4, n6, n2, n7

**Testpfad enthält alle Kanten  
(insbes. n4-n6), aber nicht  
alle Pfade.**

- Höhere Fehlerentdeckungsrate als Anweisungsüberdeckung
- Moderater Aufwand



```

int vowels = 0;
int i = 0;

while(i < ca.length)

    char c = ca[i];

    if (c == 'A' || ...
        c == 'U')

        vowels++;

        i++;

return vowels;
  
```



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

## Nachteile:

- Keine Berücksichtigung von Abhängigkeiten zwischen Zweigen
- Nicht geeignet für den Test komplexer Bedingungen
- Lösung von 1.: Pfadüberdeckung
- Lösung von 2.: Bedingungsüberdeckung



- Alle unterschiedlichen Pfade sollen beim Test mindestens einmal durchlaufen.

## Im Beispiel:

Pfade haben die Form:

$$p (q | r)^* s$$

(p,q,r,s sind Knotenfolgen)

hier:  $p = n1$ ,

$q = n2 \ n3 \ n4 \ n5 \ n6$ ,

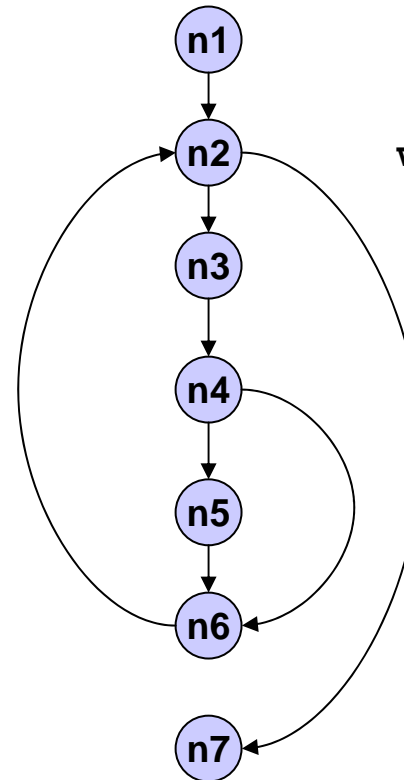
$r = n2 \ n3 \ n4 \ n6$ ,

$s = n2 \ n7$

## Daraus folgt:

Es gibt  $2 * 2^{\max\_len} - 1$  Pfade,

wobei  $\max\_len$  die maximale Länge eines `char`-Arrays ist



```

int vowels = 0;
int i = 0;

while(i < ca.length)

    char c = ca[i];

    if (c == 'A' || ...
        c == 'U')

        vowels++;

        i++;

    return vowels;
  
```



- Mächtigstes kontrollflussorientiertes Testverfahren
- Sehr hohe Fehlerentdeckungsquote in experimentellen Studien

## Nachteile:

- Pfadanzahl kann mit unbestimmter Anzahl von Wiederholungen (z.B. while-Schleife) exponentiell wachsen
- Deshalb geringe praktische Bedeutung

In der Praxis: Häufig feste Beschränkung der Anzahl von Schleifendurchläufen, dadurch handhabbare Anzahl von Pfaden!



## Einfache Bedingungsüberdeckung

- Alle atomaren Prädikate in Bedingungen sollen beim Test mindestens einmal WAHR und einmal FALSCH werden
- $\Rightarrow$  schwächer als Zweigüberdeckung

## Mehrfach-Bedingungsüberdeckung

- Bei zusammengesetzten Prädikaten kommen alle Kombinationen von WAHR und FALSCH der atomaren Prädikate vor
- $\Rightarrow$  exponentieller Aufwand

## Minimale Mehrfach-Bedingungsüberdeckung

- Jedes Prädikat - egal ob atomar oder zusammengesetzt - wird mindestens einmal WAHR und einmal FALSCH

```
a0 = a; b0 = b; c0 = c;
while (a != b || b != c) {
    if (a < b) a = a + a0;
    else if (b < c) b = b + b0;
    else c = c + c0;
}
```



Verfolgen des Flusses einer Variablen in einem Programm von der Eingabe oder Berechnung bis zur Ausgabe oder einer anderen Berechnung.



## Die Struktur entspricht Kontrollflussgraph

### Zusätzlich Annotationen:

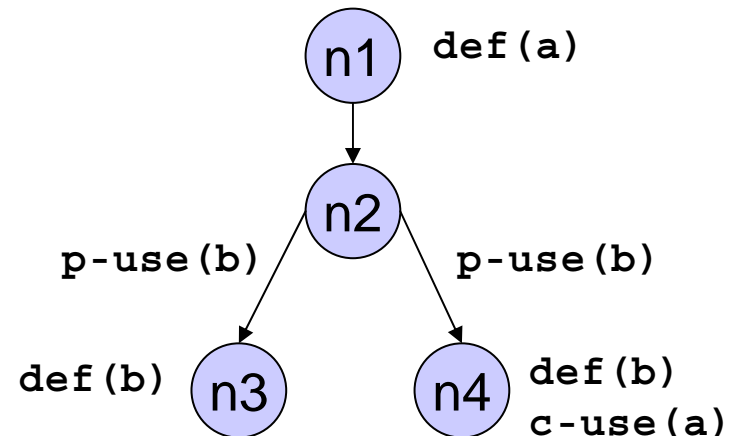
- Jede Wertzuweisung an eine Variable wird beim Knoten vermerkt (definition, def)
- Jeder Variablenzugriff in einem Ausdruck wird beim Knoten vermerkt (computational-use, c-use)
- Jeder Variablenzugriff in einem Prädikat wird an Ausgangskanten vermerkt (predicate-use, p-use)

### Beispiel:

`a = 6;`

`if (b != 9)`

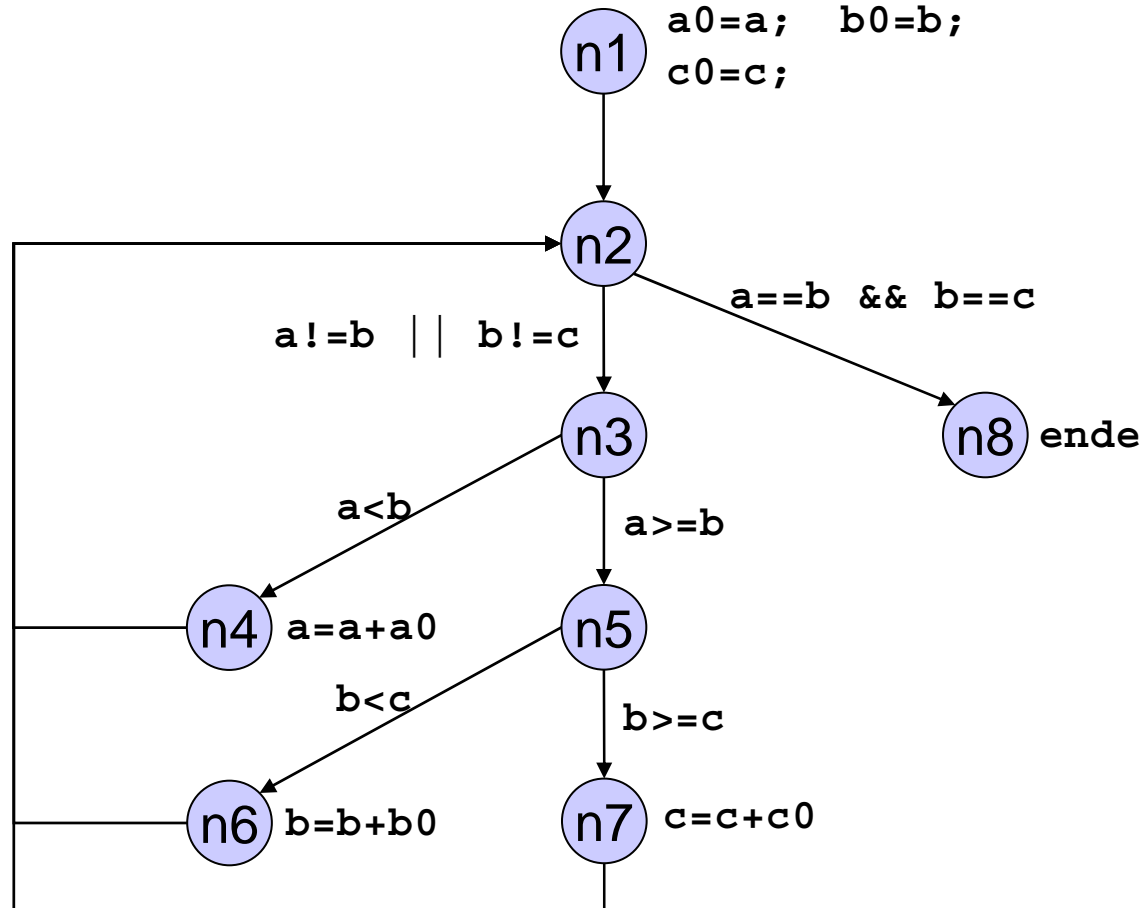
`b = 5; else b = 3 + a;`



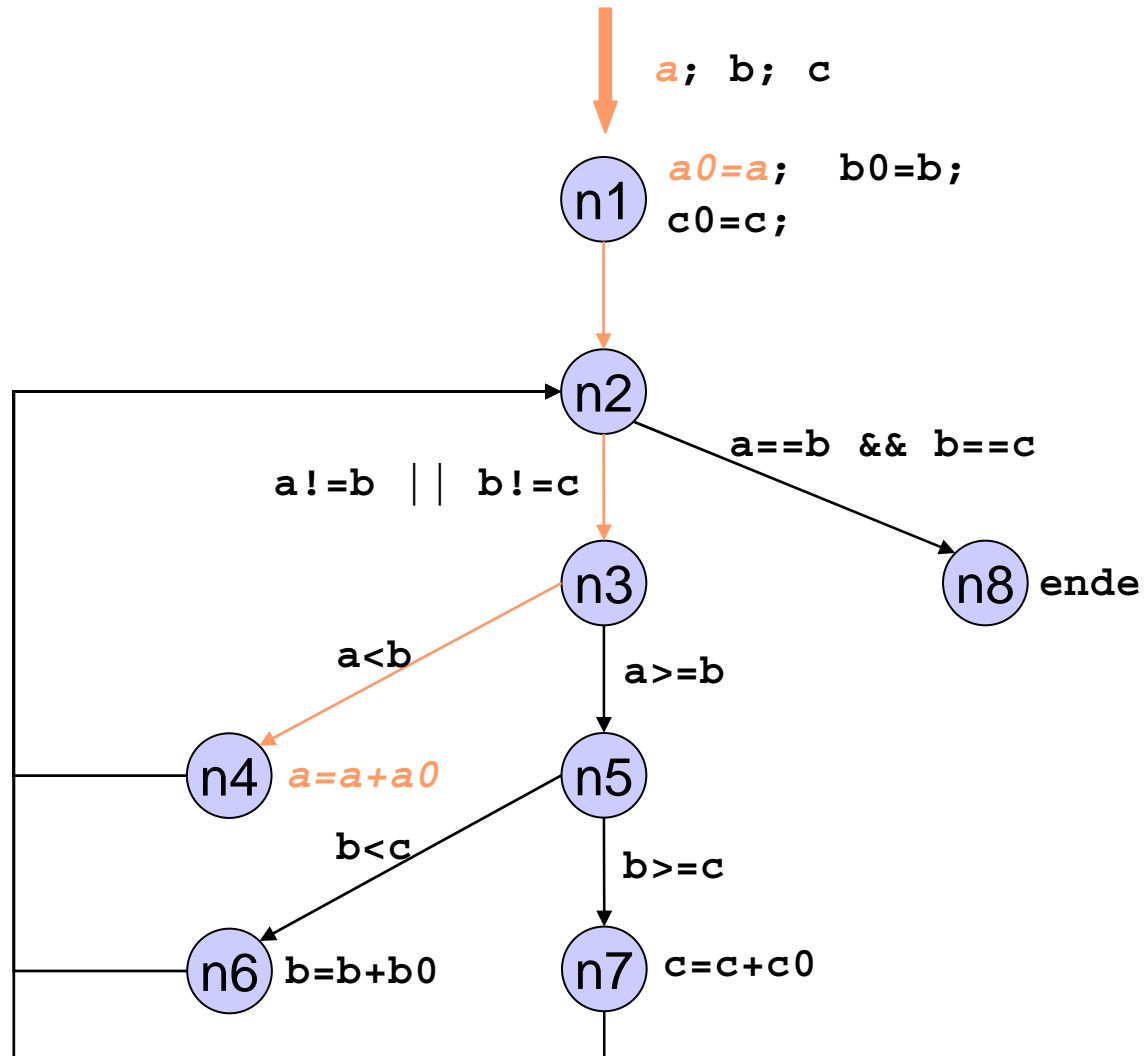
```

a0 = a; b0 = b; c0 = c;
while (a != b || b != c) {
    if (a < b)
        a = a + a0;
    else if (b < c)
        b = b + b0;
    else c = c + c0;
}

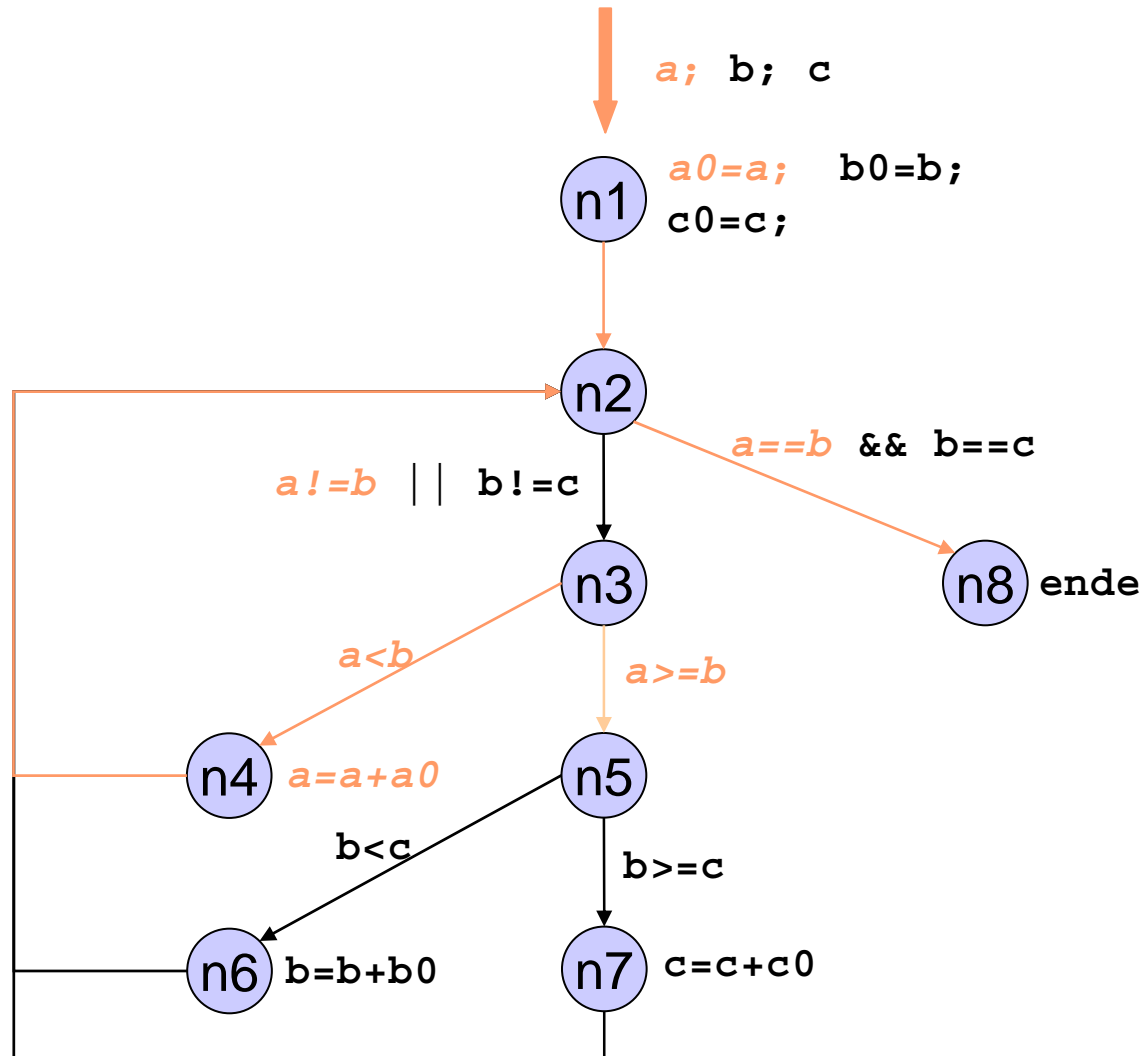
```



- Jede Kombination aus Variablendefinition und deren Nutzung in einem Ausdruck wird erfasst.
- Identifiziert Berechnungsfehler
- Beinhaltet weder Zweig- noch Anweisungsüberdeckung



- Jede Kombination aus Variablendefinition und deren Nutzung in einem Ausdruck sowie aus Variablendefinition und Nutzung in einem Prädikat wird erfasst
- Entdeckt ca. 70% der Programmierfehler
- Beinhaltet Zweigüberdeckung



## Vorteile:

- Verfahren ist experimentell. Teile des Verfahrens können durch Personen mit geringerer Qualifikation durchgeführt werden.
- Als Nebeneffekt können andere Qualitätssicherungseigenschaften mit überprüft werden.
- Werkzeuge zur Testunterstützung sind einfacher als diejenigen zur Programmverifikation
- Testaufwand ist durch Toleranzschwelle steuerbar
- Reale Produktionsumgebung kann berücksichtigt werden

## Nachteile:

- Korrektheit kann nicht bewiesen werden
- Vertrauen in getestetes Programm hängt von der Auswahl der Testfälle ab und vielen anderen Randbedingungen
- Nur beschränkte Zuverlässigkeit erreichbar
- Keine Unterscheidung möglich, ob der beobachtete Effekt dem Prüfling oder der realen Umgebung zuzuschreiben ist.



Balzert: "Lehrbuch Grundlagen der Informatik",  
950 Seiten, Spektrum Akademischer Verlag, ISBN: 3827403588

