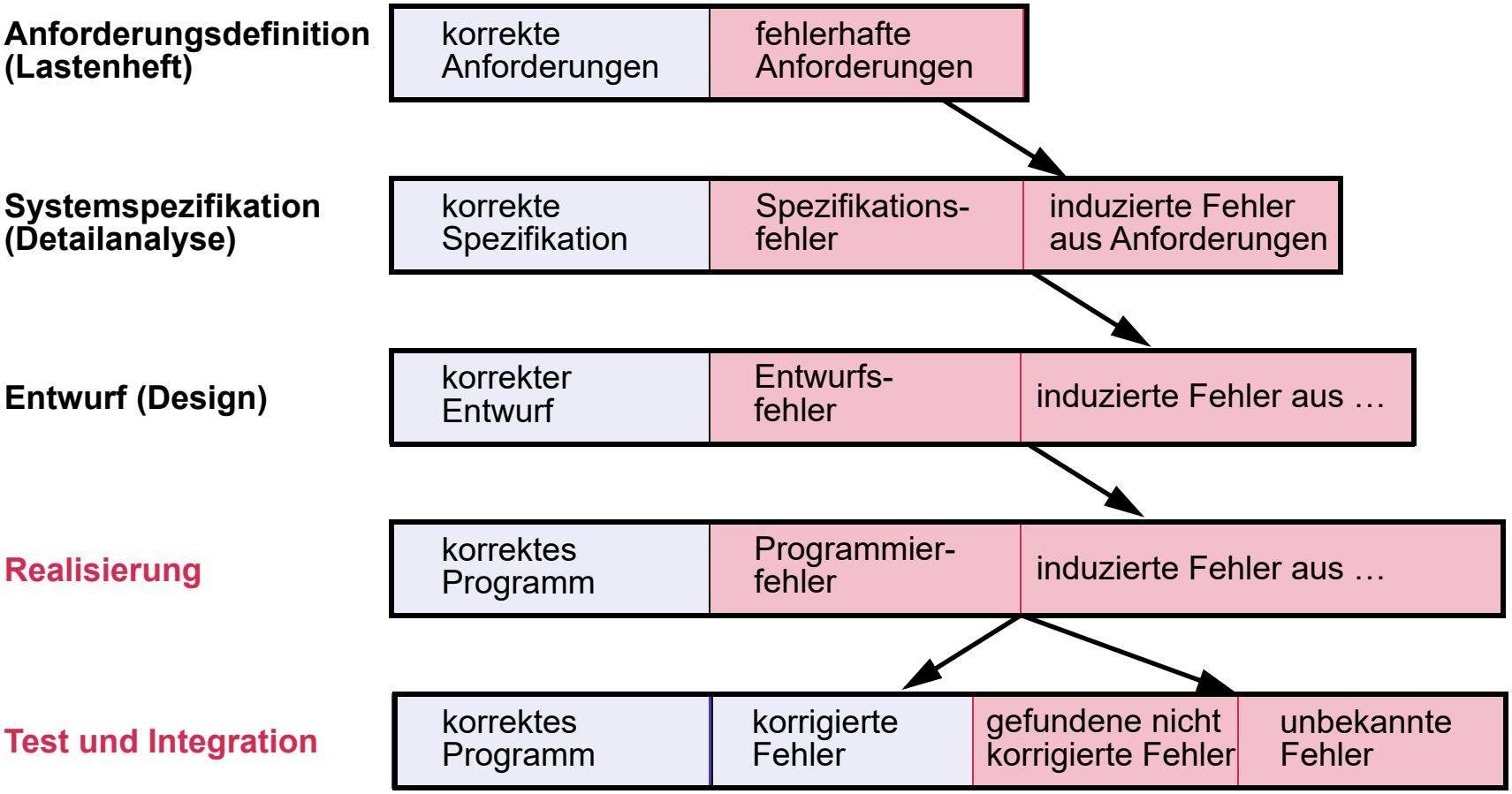
4. Dynamische Programmanalysen und Testen

The older I get, the more aggressive I get about testing. I like Kent Beck's rule of thumb that a developer should write at least as much test code as production code. Testing should be a continuous process. No code should be written until you know how to test it. Once you have written it, write the tests for it. Until the test works, you cannot claim to have finished writing the code. *[FS98]*

Lernziele:

- Schwierigkeiten des systematischen Tests von Software verstehen
- Minimalstandards (und Werkzeuge) für statische Analysen und dynamische Test von Software kennenlernen
- neuere Entwicklungen und Trends einschätzen können (modellbasiertes Testen, mutationsbasiertes Testen, ...)
- Werkzeuge für Performanzanalyse und Identifikation von Speicherlecks (durch Laufzeitinstrumentierungen) kennenlernen

4.1 Einleitung



Fehlerzustand, Fehlerwirkung und Fehlhandlung (DIN 66271):

- **Fehlerzustand (fault) - direkt erkennbar durch statische Tests:**
  - ⇨ inkorrektes Teilprogramm, inkorrekte Anweisung oder Datendefinition, die Ursache für Fehlerwirkung ist
  - ⇨ Zustand eines Softwareprodukts oder einer seiner Komponenten, der unter spezifischen Bedingungen eine geforderte Funktion beeinträchtigen kann
- **Fehlerwirkung (failure) - direkt erkennbar durch dynamische Tests:**
  - ⇨ Wirkung eines Fehlerzustandes, die bei der Ausführung des Testobjektes nach „aussen“ in Erscheinung tritt
  - ⇨ Abweichung zwischen spezifiziertem Soll-Wert (Anforderungsdefinition) und beobachtetem Ist-Wert (bzw. Soll- und Ist-Verhalten)
- **Fehlhandlung (error):**
  - ⇨ menschliche Handlung (des Entwicklers), die zu einem Fehlerzustand in der Software führt
  - ⇨ **NICHT einbezogen**: menschliche Handlung eines Anwenders, die ein unerwünschtes Ergebnis zur Folge hat

Ursachenkette für Fehler (in Anlehnung an DIN 66271):

- jeder Fehler (**fault**) oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden - Software nützt sich nicht ab
- er ist aufgrund des fehlerhaften Verhaltens (**error**) eines Entwicklers entstanden (und wegen mangelhafter Qualitätssicherungsmaßnahmen nicht entdeckt worden)
- ein Softwarefehler kommt nur bei der Ausführung der Software als Fehlerwirkung (**failure**) zum Tragen und führt dann zu einer ggf. sichtbaren Abweichung des tatsächlichen Programmverhaltens vom gewünschten Programmverhalten
- Fehler in einem Programm können durch andere Fehler **maskiert** werden und kommen somit ggf. nie zum Tragen (bis diese anderen Fehler behoben sind)

Validation und Verifikation (durch dynamische Tests):

- **Validation von Software:**

  Prüfung, ob die Software das vom Anwender „wirklich“ gewünschte Verhalten zeigt (in einem bestimmten Anwendungsszenario)

  ☞ Haben wir das richtige Softwaresystem realisiert?

- **Verifikation von Software:**

  Prüfung, ob die Implementierung der Software die Anforderungen erfüllt, die vorab (vertraglich) festgelegt wurden

  ☞ Haben wir das Softwaresystem richtig realisiert?

Achtung:

Eine „richtig realisierte“ = korrekte Software (erfüllt die spezifizierten Anforderungen) muss noch lange nicht das „wirklich“ gewünschte Verhalten zeigen!

Typische Programmierfehler nach [BP84]:

- **Berechnungsfehler**: Komponente berechnet falsche Funktion
  - ⇨ z.B. Konvertierungsfehler in Fortran bei Variablen, die mit I, J oder K anfangen und damit implizit als Integer deklariert sind
- **Schnittstellenfehler**: Inkonsistenz (bezüglich erwarteter Funktionsweise) zwischen Aufrufsstelle und Deklaration - siehe vor allem Abschnitt 4.3
  - ⇨ Übergabe falscher Parameter, Vertauschen von Parametern
  - ⇨ Verletzung der Randbedingungen, unter denen aufgerufene Komponente funktioniert
- **Kontrollflussfehler**: Ausführung eines falschen Programmpfades - siehe vor allem Abschnitt 4.4
  - ⇨ Vertauschung von Anweisungen
  - ⇨ falsche Kontrollbedingung (z.B. „kleiner“ statt „kleiner gleich“), “off by one”: Schleife wird einmal zuwenig oder zu oft durchlaufen

Typische Programmierfehler nach [BP84], Fortsetzung:

- **Datenflussfehler**: falscher Zugriff auf Variablen und Datenstrukturen - siehe vor allem Abschnitt 4.2 und Abschnitt 4.5
  - ⇨ Variable wird nicht initialisiert (Initialisierungsfehler)
  - ⇨ falsche Arrayindizierung
  - ⇨ Zuweisung an falsche Variable
  - ⇨ Zugriff auf Nil-Pointer oder bereits freigegebenes Objekt
  - ⇨ Objekt wird nicht freigegeben
- **Zeitfehler**: gefordertes Zeitverhalten wird nicht eingehalten - siehe Abschnitt 4.2
  - ⇨ Implementierung ist nicht effizient genug
  - ⇨ wichtige Interrupts werden zu lange blockiert
- **Redefinitionsfehler**: geerbte Operation wird nicht semantikerhaltend redefiniert - siehe Abschnitt 4.6
  - ⇨ ein „Nutzer“ der Oberklasse geht von Eigenschaften der aufgerufenen Operation aus, die Redefinition in Unterklasse nicht (mehr) erfüllt

Beispiel für fehlerhafte Prozedur:

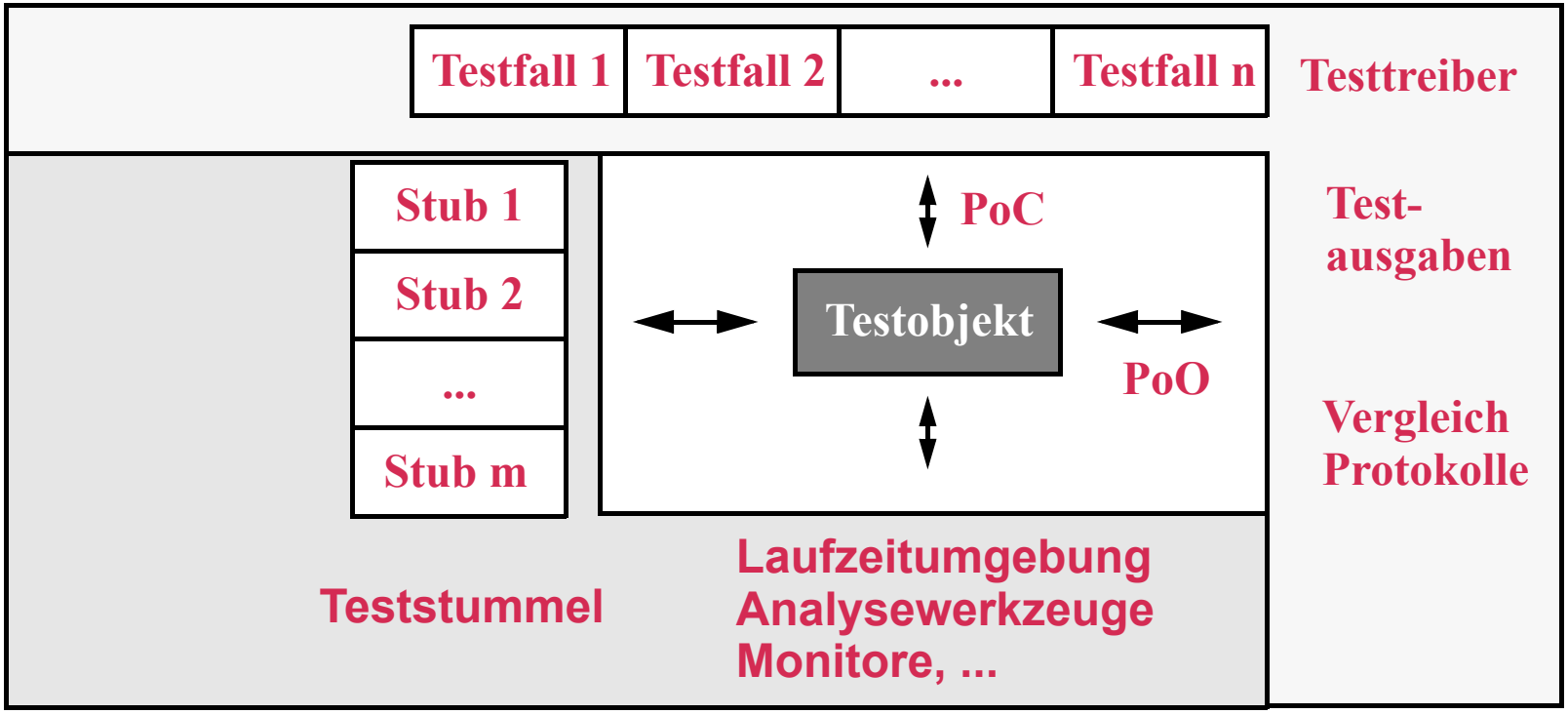
```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
VAR i: INTEGER;
BEGIN
   count := 0;                                          (* Initialisierungsfehler - i wird nicht initialisiert. *)
   WHILE s[i] # '.' DO
      IF s[i]= 'a' OR s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'   (* Kontrollflussfehler - keine Prüfung auf 'e'. *)
      THEN
         count := count + 2;                            (* Berechnungsfehler - count wird um 2 erhöht. *)
      END;
      count := i+1;                                     (* Datenflussfehler - Zuweisung nicht an i. *)
   END (* WHILE *)
END countVowels

   …
countVowels('to be . . . or not to be.', count);        (*Schnittstellenfehler - dot im Satz. *)
```

Was wird also getestet:

Testverfahren für Softwarekomponenten (Operation, Klasse, Modul/Paket, System) können danach klassifiziert werden, was getestet wird:

- **Funktionalitätstest**: das Ein-/Ausgabeverhalten der Software; das steht beim Testen (zunächst) stark im Vordergrund und wird in Abschnitt 4.3 bis Abschnitt 4.5 behandelt
- **[Benutzbarkeitstest**: es geht um die „gute“ Gestaltung der Benutzeroberfläche; schwieriges Thema, das hier nicht weiter vertieft wird **]**
- **Performanztest**: Laufzeitverhalten und Speicherplatzverbrauch einer Komponente werden gemessen und dabei oft durchlaufene ineffiziente Programmteile oder Speicherlecks identifiziert; siehe Abschnitt 4.2
- **Lasttest**: die Komponente wird mit schrittweise zunehmender Systemlast **innerhalb des zugelassenen/spezifizierten Bereiches** (für Eingabedaten) getestet
- **Stresstest**: die Systemlast wird solange erhöht, bis sie **außerhalb des zugelassenen/spezifizierten Bereiches** (für Eingabedaten) liegt; damit wird das Verhalten des Systems unter Überlast beobachtet

Wie wird getest - Aufbau eines Testrahmens gemäß [SL19]:



- **Point of Control (PoC)**:
  Schnittstelle, über die Testobjekt mit Testdaten versorgt wird‘
- **Point of Observation (PoO)**:
  Schnittstelle, über die Reaktionen/Ausgaben des Testobjekts beobachtet werden

Wie wird getestet - Arten von Testverfahren:

- **Funktionstest** (black box test): die interne Struktur der Komponente wird nicht betrachtet; getestet wird Ein-/Ausgabeverhalten gegen Spezifikation (informell oder formal); siehe Abschnitt 4.3 und Abschnitt 4.6
- **Strukturtest** (white box test): interne Struktur der Komponente wird zur Testplanung und -Überwachung herangezogen:
  - ⇨ Kontrollflussgraph: siehe Abschnitt 4.4
  - ⇨ Datenflussgraph: siehe Abschnitt 4.5
  - ⇨ [Automaten: siehe Abschnitt 4.6]
- **Diversifikationstest**: Verhalten einer Komponentenversion wird mit Verhalten anderer Komponentenversionen verglichen

Diversifikationstestverfahren:

Das Verhalten verschiedener Varianten eines Programms wird verglichen.

- **Mutationstestverfahren**: ein Programm wird absichtlich durch Transformationen verändert und damit in aller Regel mit Fehlern versehen
  - ⇨ Einbau von Fehlern lässt sich (mehr oder weniger) automatisieren
  - ⇨ eingebaute Fehler entsprechen oft nicht tatsächlich gemachten Fehlern
  - ⇨ eingebaute Fehler stören Suche nach echten Fehlern
- **N-Versionen-Programmierung**: verschiedene Versionen eines Programms werden völlig unabhängig voneinander entwickelt
  - ⇨ sehr aufwändig, da man mehrere Entwicklerteams braucht (und gegebenfalls sogar Hardware mehrfach beschaffen muß)
  - ⇨ Fehler der verschiedenen Teams nicht unabhängig voneinander (z.B. haben alle Versionen dieselben Anforderungsdefinitionsfehlern)

Mutationstestverfahren:

Erzeugung von Mutationen durch:

Vertauschen von Anweisungsfolgen, Umdrehen von Kontrollflussbedingungen, Löschen von Zuweisungen, Ändern von Konstanten, …

Zielsetzungen:

- ⇨ **Identifikation fehlender Testfälle**: für jeden Mutanten sollte mindestens ein Testfall existieren, der Original und Mutant unterscheiden kann
- ⇨ **Eliminination nutzloser Testfälle**: Gruppe von Testfällen verhält sich bezüglich der Erkennung von Mutanten völlig gleich
- ⇨ **Schätzung der Restfehlermenge** RF: $RF \sim GF * (M/GM - 1)$

  mit GM = Anzahl gefundener eingebauter Fehler (Mutationen)
  M = Gesamtzahl absichtlich eingebauter Fehler
  GF = Anzahl gefundener echter Fehler
  [F = Gesamtzahl echter Fehler = GF + RF]

  Annahme: $GF/F \sim GM/M$ (Korrelation gilt allenfalls für bestimmte Fehler)

N-Versionen-Programmierung (Back-to-Back-Testing):

Zielsetzungen:

- ⇨ eine Version kann als Orakel für die Korrektheit der Ausgaben einer anderen Version herangezogen werden (geht ab 2 Versionen)
- ⇨ Robustheit der ausgelieferten Software kann erhöht werden durch gleichzeitige Berechnung eines benötigten Ergebnisses durch mehrere Versionen einer Software
- ⇨ Liefern verschiedene Versionen unterschiedliche Ergebnisse, so wird Mehrheitsentscheidung verwendet (geht ab 3 Versionen)

Probleme:

- ⇨ N Versionen enthalten mehr Fehler als eine Version: falsche Versionen können richtige überstimmen oder gemeinsame Ressourcen blockieren
- ⇨ Fehler verschiedener Versionen sind nicht immer unabhängig voneinander: Fehler aus der Anforderungsdefinition oder typische Programmiersprachenfehler oder falsche Algorithmen können alle Versionen enthalten

Exkurs zur Berechnung von Ausfallwahrscheinlichkeiten:

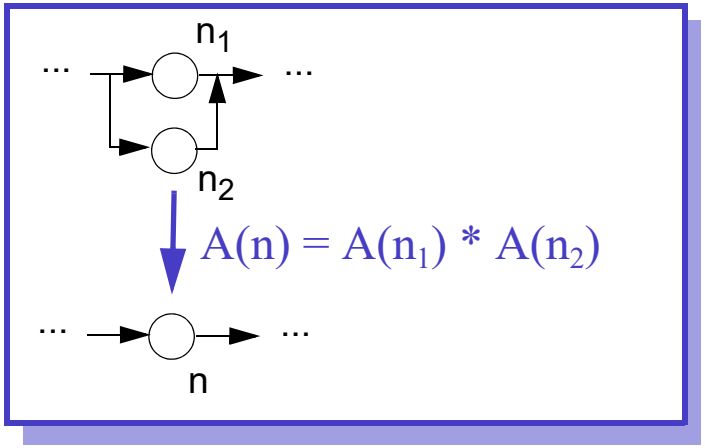
Für die Berechnung der Ausfallwahrscheinlichkeit eines (Software-)Systems wird der innere Aufbau des Systems wie folgt als ein gerichteter (Kontrollfluss-)Graph bzw. Abhängigkeitsgraph $S = (N, E, n_{start}, n_{final})$ dargestellt:

- die Knoten N sind die **Komponenten**, aus denen das System besteht
- die Kanten E beschreiben **Berechnungsabhängigkeiten** bzw. -pfade zwischen den Komponenten des Systems
- eine zusätzlich gegebene Funktion $A: N \rightarrow [0..1]$ legt für jede Komponente n deren **Ausfallwahrscheinlichkeit** A(n) fest, die unabhängig von den Ausfallwahrscheinlichkeiten anderer Komponenten sein soll

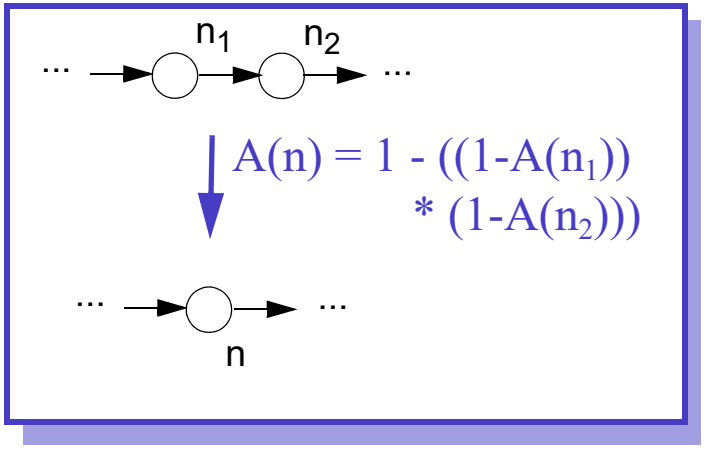
Ein solches System gilt *im einfachsten Fall* als genau dann **ausgefallen**, sobald es keinen Pfad von n_{start} nach n_{final} gibt, auf dem keine Komponente ausgefallen ist.

Einfache Rechenregeln für System-Ausfallwahrscheinlichkeit:

Eine „**Parallelschaltung**“ zweier unabhängiger Komponenten n_1 und n_2 fällt dann aus, wenn *beide* Komponenten ausfallen. Das Bild rechts zeigt die Ersetzung der Parallelschaltung der beiden Komponenten n_1 und n_2 durch eine Komponente n mit gleicher Ausfallwahrscheinlichkeit.



Eine „**Reihenschaltung**“ zweier unabhängiger Komponenten n_1 und n_2 ist dann *nicht* ausgefallen, wenn *beide* Komponenten nicht ausgefallen sind. Das Bild rechts zeigt die Ersetzung der Reihenschaltung von n_1 und n_2 durch eine Komponente n mit gleicher Ausfallwahrscheinlichkeit.



Rekursive Rechenregel für System-Ausfallwahrscheinlichkeit:

Die Berechnung der Ausfallwahrscheinlichkeit eines Systems S lässt sich bei Fokus auf den Status einer bestimmten Komponente n in zwei Fälle zerlegen:

1. Berechnung der Ausfallwahrscheinlichkeit von S unter der Annahme, dass die **Komponente n ausgefallen** ist = $A(S)_{\text{n ist ausgefallen}}$; dabei sei A(n) die Wahrscheinlichkeit, dass Komponente n ausgefallen ist.

2. Berechnung der Ausfallwahrscheinlichkeit von S unter der Annahme, dass die **Komponente n nicht ausgefallen ist** = $A(S)_{\text{n ist nicht ausgefallen}}$; dabei sei 1 - A(n) die Wahrscheinlichkeit, dass Komponente n nicht ausgefallen ist.

Damit ergibt sich:

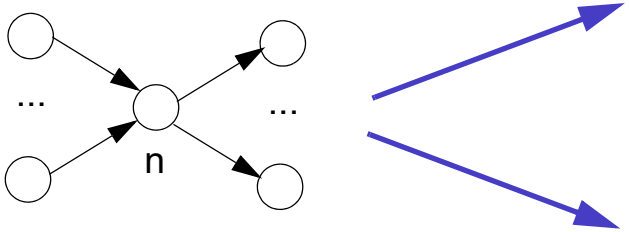
$$A(S) = A(n) * A(S)_{\text{n ist ausgefallen}} + (1-A(n)) * A(S)_{\text{n ist nicht ausgefallen}}$$

Als Komponente n für die Fallunterscheidung wählt man geschickterweise eine Komponente, die ansonsten die vollständige Dekomposition des betrachteten Graphen in einfach zu behandelnde Serien- und Parallelschaltungen verhindert.

Rekursive Rechenregel - Fortsetzung:

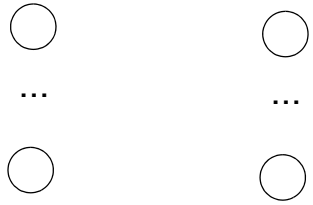
Die Berechnung der Wahrscheinlichkeiten $A(S)_{\text{n ist ausgefallen}}$ und $A(S)_{\text{n ist nicht ausgefallen}}$ in der obigen Formel erfolgt durch die Berechnung der Ausfallwahrscheinlichkeiten zweier „Ersatzschaltbilder“:

S:



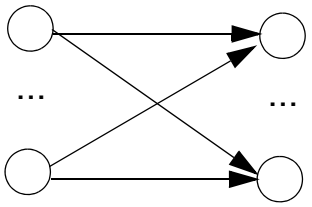
$$A(S) = A(n) * A(S_1) + (1-A(n)) * A(S_2)$$

S_1:



„Ersatzschaltung“ S_1 für S für immer ausgefallener und deshalb gelöschter Komponente n in S_1 .

S_2:



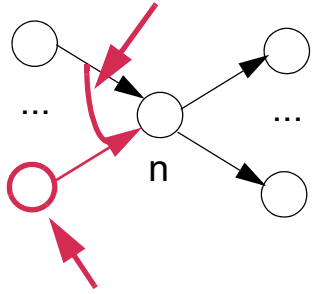
„Ersatzschaltung“ S_2 für S für immer funktionierende Komponente n in S_2 , die deshalb durch Verbindungen aller Nachfolger mit allen Vorgängern v. n ersetzt wird.

1. $A(S)_{\text{n ist ausgefallen}} = A(S_1)$
2. $A(S)_{\text{n ist nicht ausgefallen}} = A(S_2)$

Rekursive Rechenregel - Ergänzung:

In einigen Fällen muss man auch Komponenten behandeln, die nur dann „funktionieren“, wenn alle ihre Eingänge korrekte Eingaben erhalten (und damit alle ihre Vorgängerkomponenten nicht ausgefallen sind).
Wenn bei einer solchen Komponente eine Vorgängerkomponente ausfällt, fällt die Komponente selbst auch aus.

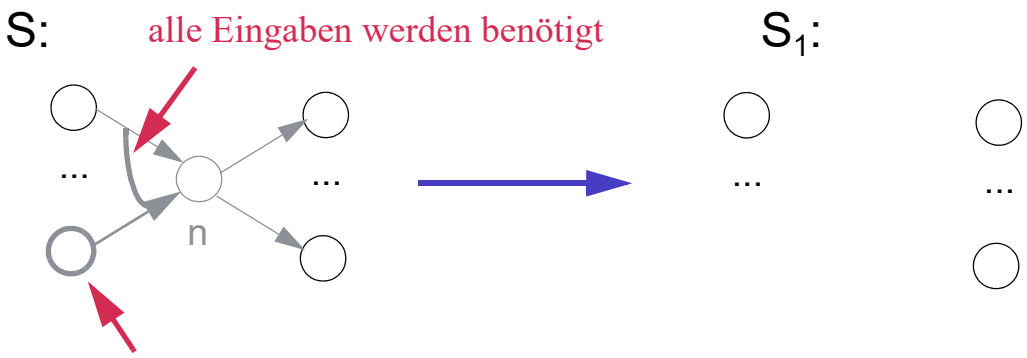
S: alle Eingaben werden benötigt



diese Komponente ist ausgefallen

Rekursive Rechenregel - Ergänzung:

In einigen Fällen muss man auch Komponenten behandeln, die nur dann „funktionieren“, wenn alle ihre Eingänge korrekte Eingaben erhalten (und damit alle ihre Vorgängerkomponenten nicht ausgefallen sind).
Wenn bei einer solchen Komponente eine Vorgängerkomponente ausfällt, fällt die Komponente selbst auch aus.

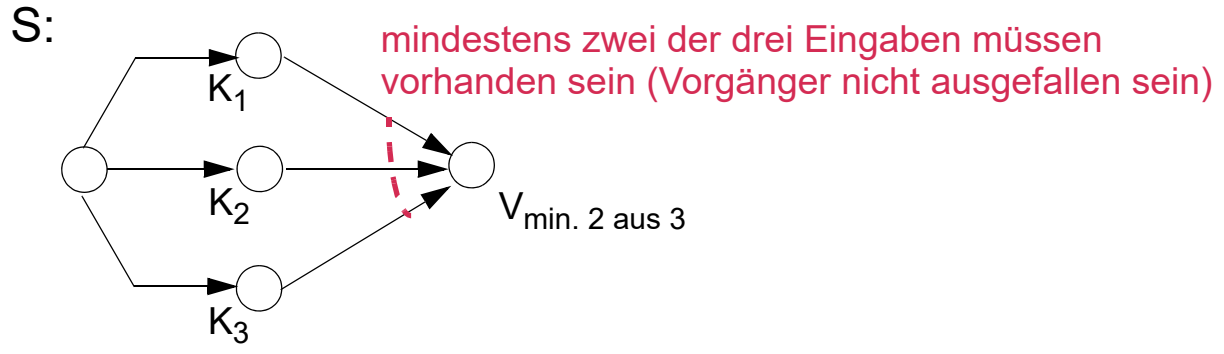


$$A(S) = A(S_1) = \ldots$$

„Ersatzschaltung“ S_1 für S in der Löschen propagiert, da die Komponente in der Mitte ebenfalls ausfällt, falls irgendeine der Komponenten ausfällt, die eine Eingabe für sie erzeugt. Ggf. muss Löschen über mehrere Stufen weiterpropagiert werden.

Beispiel „Dreifachredundanz mit 2-aus-3-Voter“:

Die Berechnungseinheit K eines Systems sei mit dreifacher Redundanz ausgelegt als Komponenten K_1, K_2 und K_3. Ein nachgeschalteter Voter V liefert ein Berechnungsergebnis, falls mindestens zwei Komponenten dasselbe Berechnungsergebnis liefern.



Achtung:

Die Wahrscheinlichkeit, dass der Voter ein falsches Ergebnis berechnet, hängt von seiner eigenen Ausfallwahrscheinlichkeit und der Funktionsweise seiner „Vorgänger“ ab. Diese stellen **keine** einfache „Parallelschaltung“ wie auf der vorigen Folie dar!

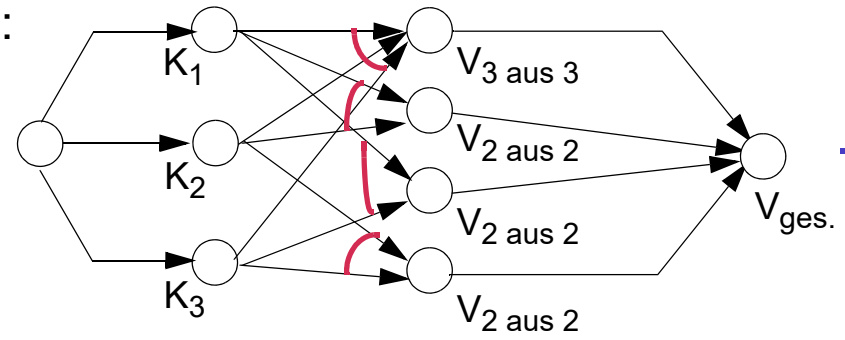
Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Fortsetzung:

Die Kernfunktionalität eines Systems sei mit dreifacher Redundanz ausgelegt als Komponenten K_1, K_2 und K_3. Ein nachgeschalteter Voter $V_{\text{min. 2 aus 3}}$ liefert ein Berechnungsergebnis, falls mindestens zwei Komponenten dasselbe Berechnungsergebnis (rechtzeitig) liefern. Zudem werden folgende vereinfachende Annahmen getroffen:

- die Ausfallwahrscheinlichkeiten $A(K_1) = A(K_2) = A(K_3)$ seien gleich, unabhängig und charakterisieren die Situation, dass das richtige Ergebnis (rechtzeitig) berechnet wird (die Komponenten wurden unabhängig voneinander entwickelt, etc.)
- die Wahrscheinlichkeit, dass zwei Komponenten dasselbe falsche Ergebnis berechnen geht gegen Null (da es sehr viele mögliche Berechnungsergebnisse gibt)
- der komplexe Voter $V_{\text{min. 2 aus 3}}$ lässt sich durch eine Parallelschaltung mehrerer einfacherer Voter mit gleicher Ausfallwahrscheinlichkeit $A(V_{xyz})$ ersetzen:
  - ⇨ $V_{3 \text{ aus } 3}$ liefert genau dann ein (richtiges) Ergebnis, wenn alle drei angeschlossenen Berechnungskomponenten dasselbe Ergebnis berechnen
  - ⇨ $V_{2 \text{ aus } 2}$ liefert genau dann ein (richtiges) Ergebnis, wenn beide angeschlossenen Berechnungskomponenten dasselbe Ergebnis berechnen

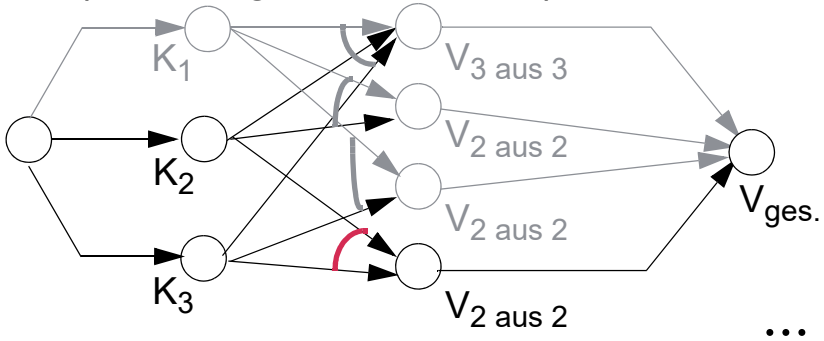
Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Fortsetzung:

S:



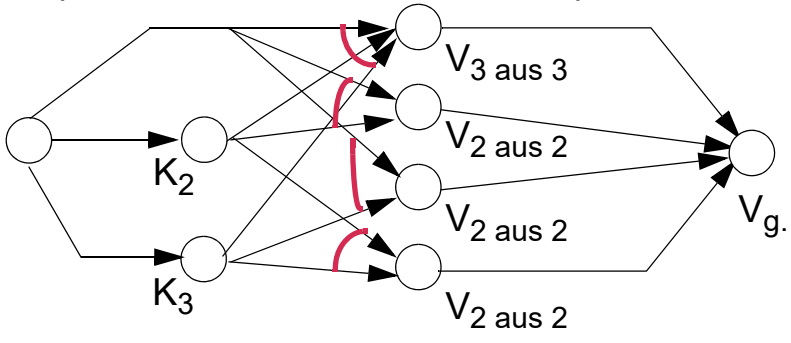
$$A(S) = A(K_1) * A(S_1) + (1 - A(K_1)) * A(S_2) = \ldots$$

S_1 (mit ausgefallenem K_1):

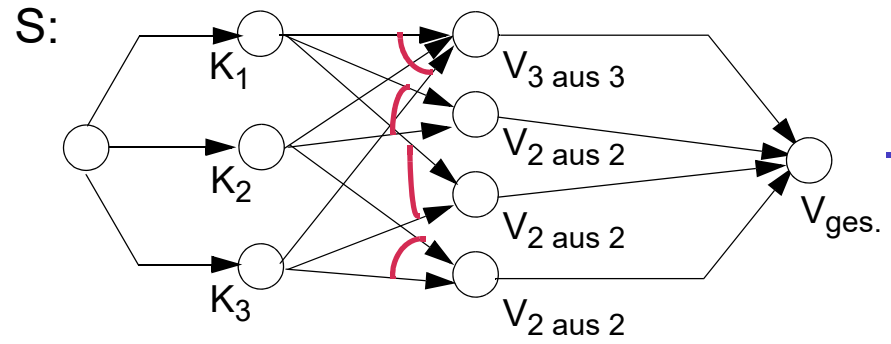


...

S_2 (mit funktionierendem K_1):

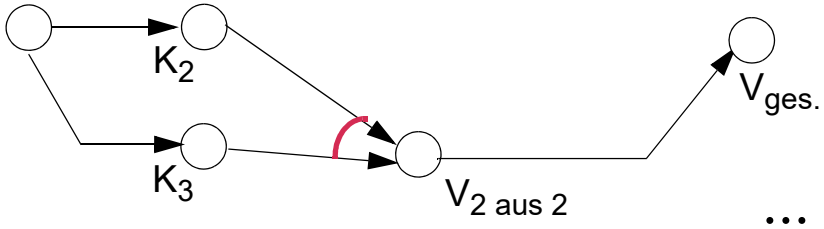


Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Vereinfachung:



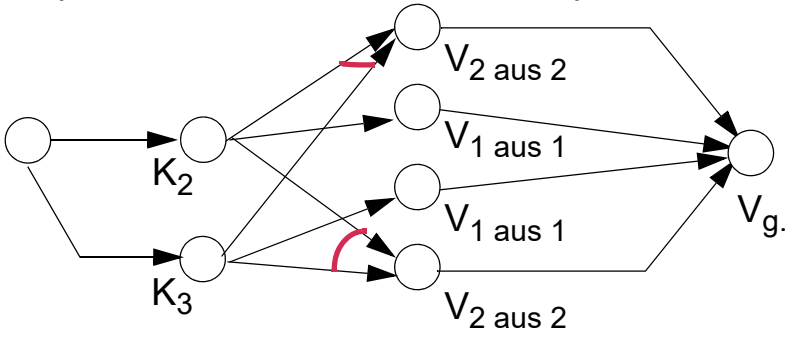
$$A(S) = A(K_1) * A(S_1) + (1 - A(K_1)) * A(S_2) = \ldots$$

S_1 (mit ausgefallenem K_1):



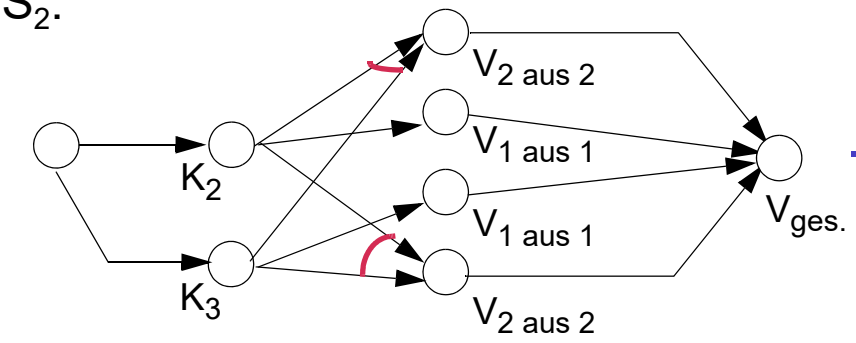
...

S_2 (mit funktionierendem K_1):



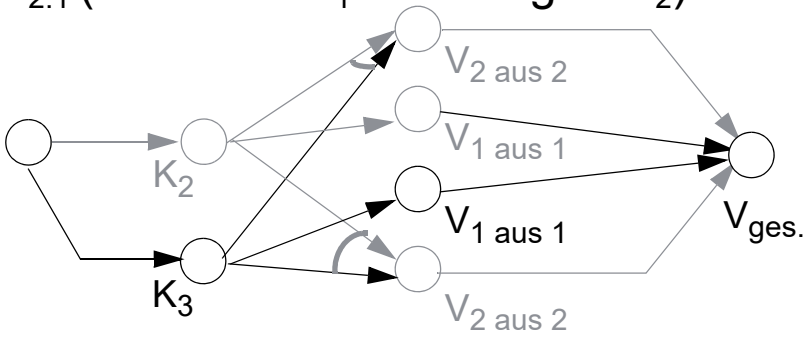
Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Fortsetzung:

S_2:

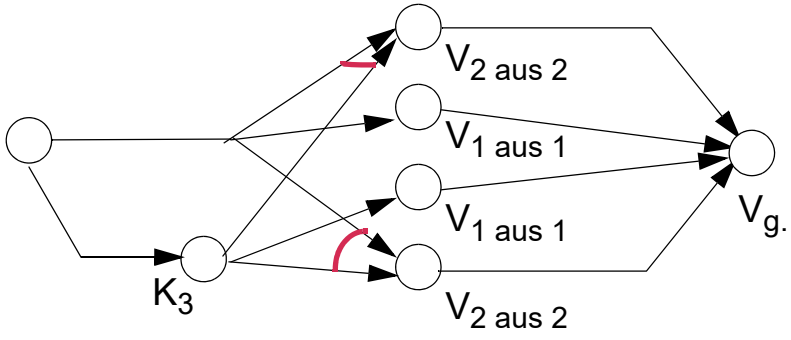


$$A(S_2) = A(K_2) * A(S_{2.1}) + (1 - A(K_2)) * A(S_{2.2}) = ...$$

$S_{2.1}$ (mit funkt. K_1 und ausgef. K_2):

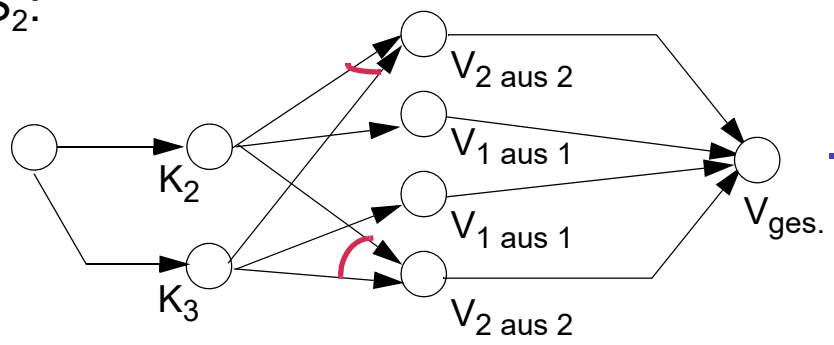


$S_{2.2}$ (mit funktionierendem K_1 und K_2):



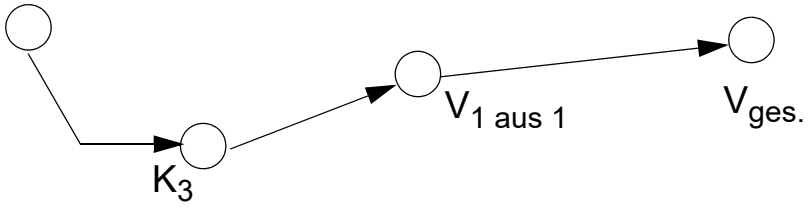
Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Vereinfachung:

S_2:

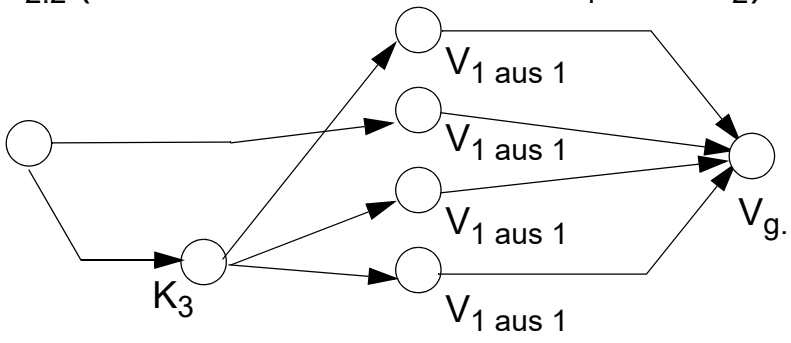


$$A(S_2) = A(K_2) * A(S_{2.1}) + (1 - A(K_2)) * A(S_{2.2}) = ...$$

$S_{2.1}$ (mit funkt. K_1 und ausgef. K_2):

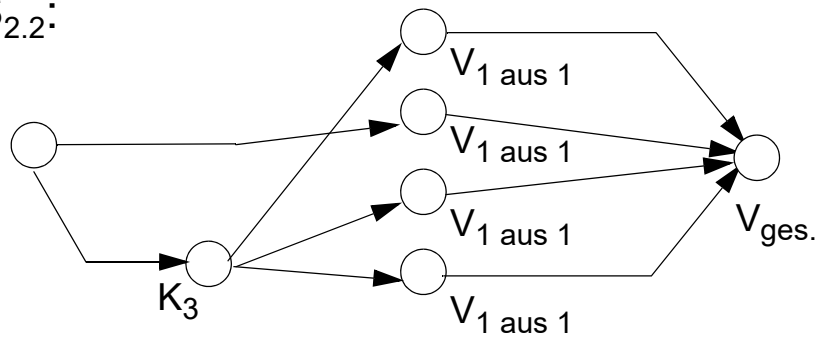


$S_{2.2}$ (mit funktionierendem K_1 und K_2):



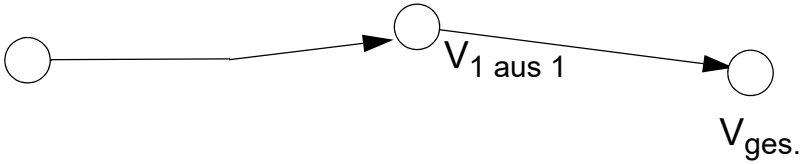
Beispiel „Dreifachredundanz mit 2-aus-3-Voter“ - Fortsetzung:

$S_{2.2}$:

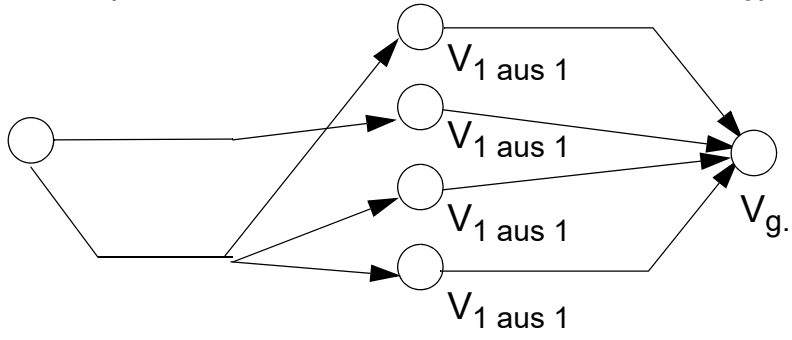


$$A(S_{2.2}) = A(K_3) * A(S_{2.2.1}) + (1 - A(K_3)) * A(S_{2.2.2}) = ...$$

$S_{2.2.1}$ (mit funkt. K_1 und K_2, ausgef. K_3):



$S_{2.2.2}$ (mit funktionierendem K_1 bis K_3):



Anmerkungen zu „Dreifachredundanz“-Beispiel:

Beim rekursiven Löschen von Komponenten und Konstruieren von Ersatzschaltbildern ist zu berücksichtigen, dass ein Voter $V_{k \text{ aus } k}$, der nur dann ein Ergebnis weiterliefert, wenn an allen k Eingängen derselbe Wert anliegt, bereits dann selbst ausfällt, wenn eine seiner Vorgängerkomponenten ausgefallen ist.

Damit führt der Ausfall von K_1 auf der vorigen Folie zum Ausfall der entsprechenden drei Voter $V_{3 \text{ aus } 3}$ und $V_{2 \text{ aus } 2}$, die Eingaben von der Komponente K_1 benötigen. Diese sind deshalb in S_1 grau dargestellt (und als ebenfalls gelöscht anzusehen).

Weitere Überlegung:

Der Voter $V_{3 \text{ aus } 3}$ wird nur dann benötigt, falls $A(V_{3 \text{ aus } 3}) > 0$ ist und diese Ausfallwahrscheinlichkeit unabhängig von den Ausfallwahrscheinlichkeiten der anderen drei Voter des Typs $V_{2 \text{ aus } 2}$ ist. Wenn wir davon ausgehen, dass der Voter $V_{3 \text{ aus } 3}$ praktisch gesehen auch immer dann ausfällt, wenn mindestens einer der Voter $V_{2 \text{ aus } 2}$ ausfällt, dann kann man den Knoten $V_{3 \text{ aus } 3}$ mit seinen ein- und auslaufenden Kanten aus dem Abhängigkeitsgraphen S entfernen.

Alternative Berechnung der Ausfallwahrscheinlichkeit von n Versionen:

n parallel geschaltete Versionen enthalten in etwa *n*-mal so viele Fehler wie eine Version, aber falls Wahrscheinlichkeit *A* für fehlerhafte Arbeitsweise einer Version *v* unabhängig vom Ausfall anderer Versionen ist, dann gilt:

- richtiges Ergebnis werde berechnet, solange höchstens $\lfloor (n - 1)/2 \rfloor$ Versionen fehlerhaft arbeiten (*n* ist sinnvoller Weise ungerade Zahl; ansonsten abrunden)
- Wahrscheinlichkeit für gleichzeitigen Ausfall von *k* unabhängigen Versionen:

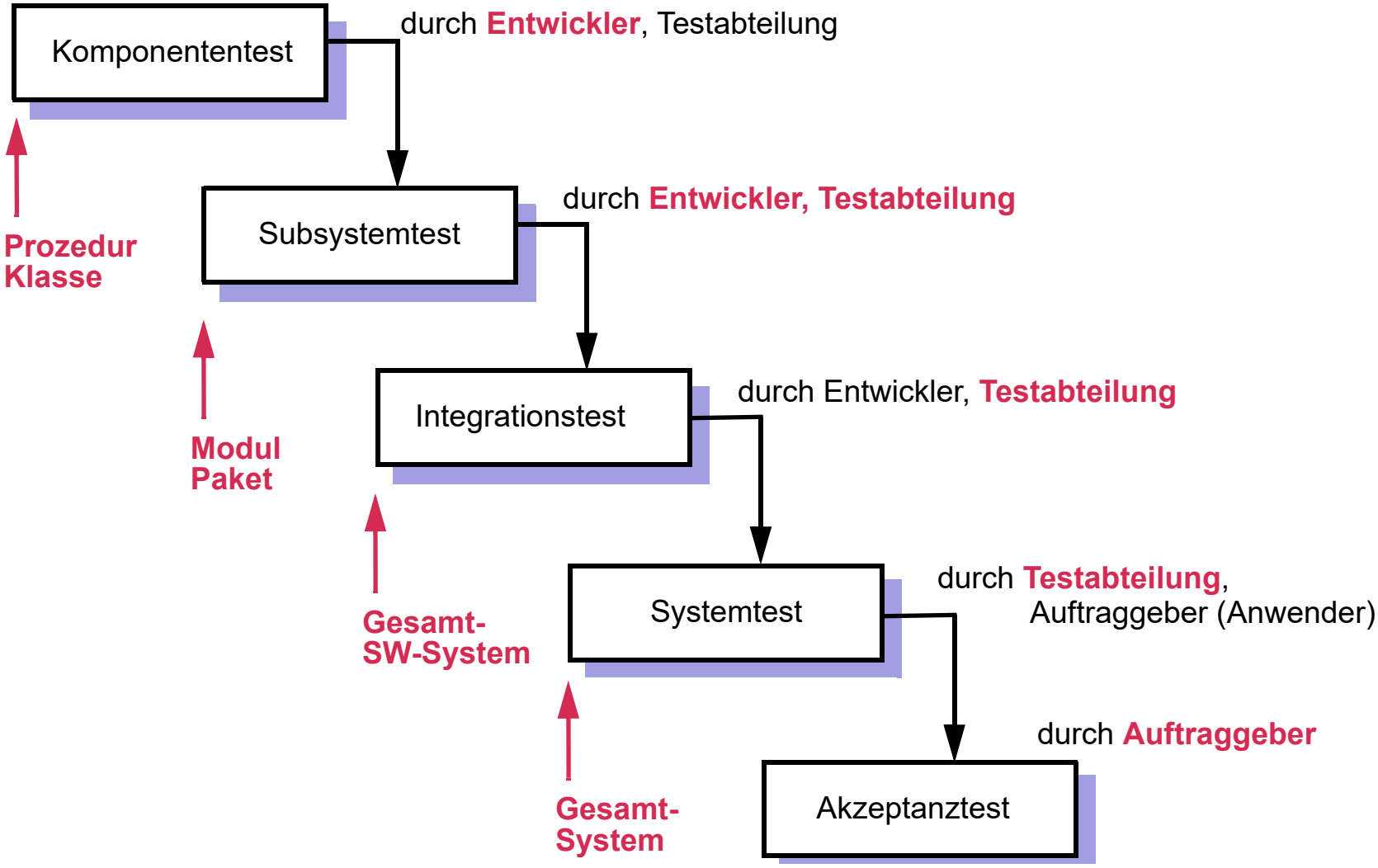
$$\binom{n}{k} \times A^k \times (1-A)^{(n-k)} = \frac{\prod_{i=n-k+1}^{n} i}{\prod_{i=1}^{k} i} \times A^k \times (1-A)^{(n-k)}$$

- Wahrscheinlichkeit für **Ausfall des Gesamtsystems** mit *n* Versionen:

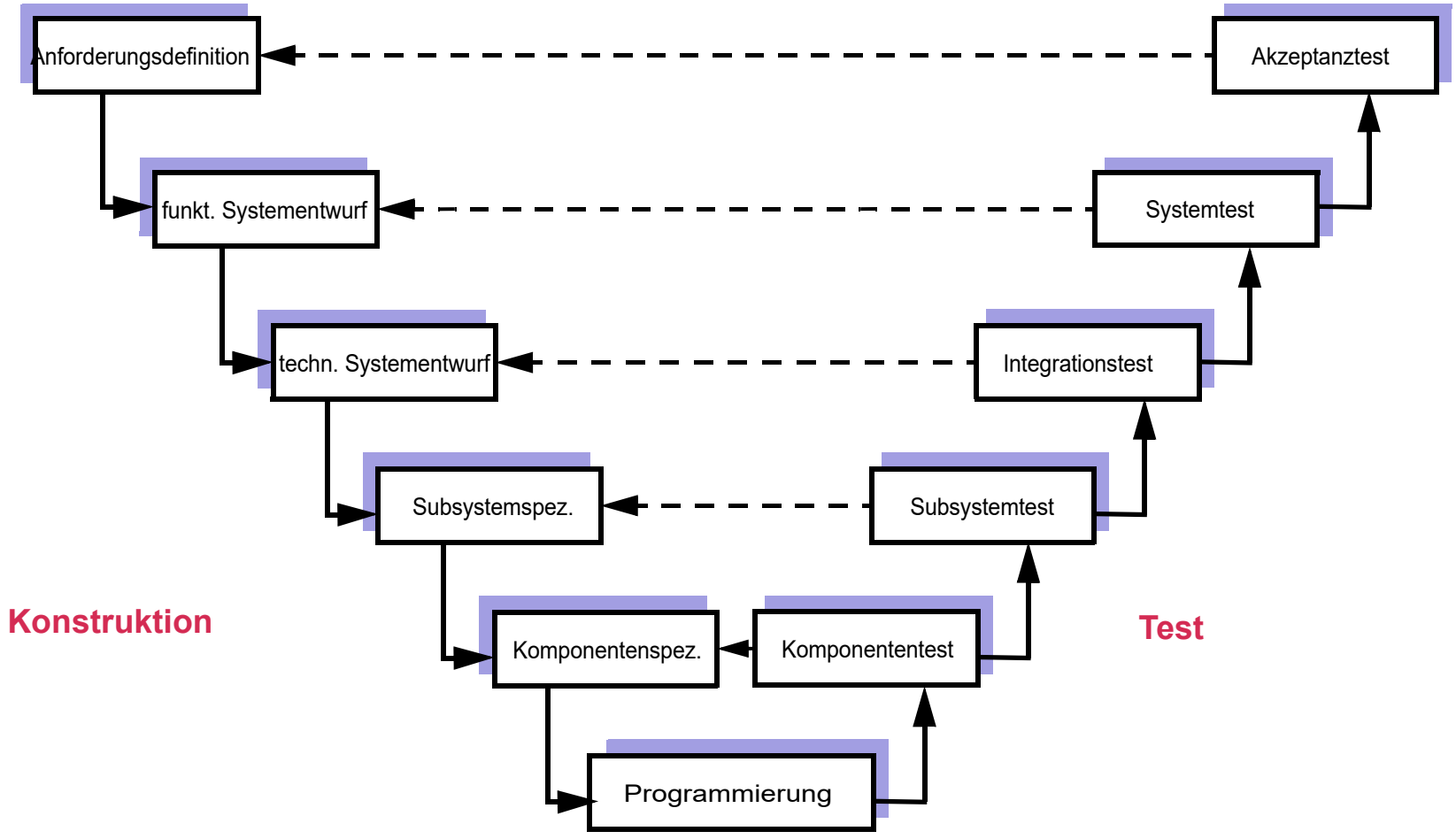
$$\sum_{k=\lfloor (n+1)/2 \rfloor}^{n} \binom{n}{k} \times A^k \times (1-A)^{(n-k)}$$

(bei n = 3: Wahrscheinlichkeit für Ausfall von 2 oder 3 Versionen)

Der Testprozess als Bestandteil des Softwareentwicklungsprozesses:



Einbettung in modifiziertes V-Modell:



Komponententest (Unit-Test) und Subsystemtest:

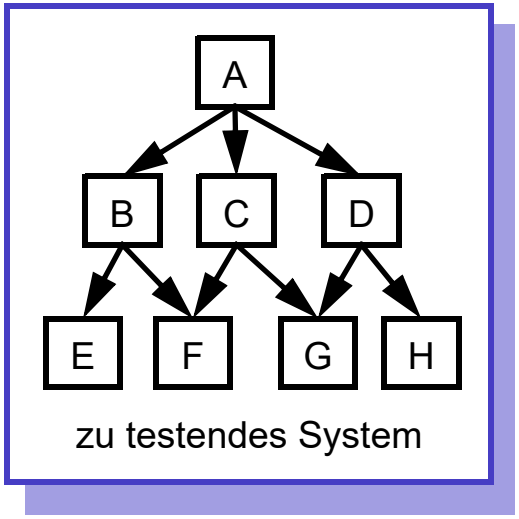
- jeweils ein **einzelner Softwarebaustein** wird überprüft, isoliert von anderen Softwarebausteinen des Systems
- die betrachtete Komponente (Unit) kann eine Klasse, Paket (Modul) sein
- **Subsystemtest** kann als Test einer besonders großen Komponente aufgefasst werden
- getestet wird gegen die Spezifikation der Schnittstelle der Komponente, dabei betrachtet werden funktionales Verhalten, Robustheit, Effizienz, ...
- Testziele sind das Aufdecken von Berechnungsfehlern, Kontrollflussfehlern, ...
- getestet wird in jedem Fall die Komponente für sich mit
  - ⇨ **Teststummel** (Platzhalter, Dummies, Stubs) für benötigte Dienste anderer Komponenten
  - ⇨ **Testtreibern** (Driver) für den (automatisierten) Test der Schnittstelle (für Eingabe von Parametern, Ausgabe von Parametern, ...)

Integrationstest:

- das gesamte Software-System (oder ein abgeschlossenes Teilsystem) wird getestet; Schwerpunkt liegt dabei auf Test des **Zusammenspiels** der Einzelkomponenten
- normalerweise wird vorausgesetzt, dass Einzelkomponenten vorab bereits getestet wurden
- auch hier müssen wieder **Testtreiber** (Testwerkzeuge) verwendet werden, die die zu testende Komponente aufrufen bzw. steuern
- auf **Teststummel** kann meist verzichtet werden, da alle benötigten Teilsysteme zusammen getestet werden
- **Testziel** ist vor allem das Aufdecken von **Schnittstellenfehlern** und insbesondere Fehler beim Austausch von Daten

Gängige Integrationsteststrategien:

- **"Big Bang"-Strategie**: alle Teile sofort integrieren und nur als Gesamtheit testen
  - ⇨ Lokalisierung von Fehlern schwierig
  - ⇨ Arbeitsteilung kaum möglich
  - ⇨ Testen beginnt zu spät



- **"Top-down"-Testverfahren**: zuerst A mit Dummies für B, C und D; dann B mit Dummies für E und F, …
  - ⇨ Erstellung „vernünftiger" Dummies schwierig
  - ⇨ Test der Basisschicht sehr spät
- **"Bottom-Up"-Testverfahren**: zuerst E, F, G und H mit Testtreibern, die Einbindung in B, C und D simulieren; dann B, C und D mit Testtreiber …
  - ⇨ Test des Gesamtverhaltens des Systems gegen Lastenheft erst am Ende
  - ⇨ Designfehler und Effizienzprobleme werden oft erst spät entdeckt

Gängige Integrationsteststrategien - Fortsetzung:

- [**Ad-Hoc-Integration**: die Komponenten werden in der (zufälligen) Reihenfolge ihrer Fertigstellung integriert und getestet]
- **Backbone-Integration (Inkrementelle Vorgehensweise)**: zunächst wird Grundgerüst erstellt, weitere Komponenten werden stückweise hinzugefügt
  - ⇨ wie erstellt und testet man Grundgerüst (z.B. Top-Down-Testen)
  - ⇨ Hinzufügen von Komponenten kann bisherige Testergebnisse entwerten
- **Regressionstest (für inkrementelle Vorgehensweise)**: da Änderungen neue Fehlerzustände in bereits getesteten Funktionen verursachen (oder bislang maskierte Fehlerzustände sichtbar machen) können werden
  - ⇨ möglichst viele Tests automatisiert
  - ⇨ bei jeder Änderung werden alle vorhandenen Tests durchgeführt
  - ⇨ neue Testergebnisse mit alten Testergebnissen verglichen

Nur inkrementelles Testen kombiniert mit Regressionstest passt zu den heute üblichen inkrementellen und iterativen Softwareentwicklungsprozessen (siehe Kapitel 5).

Systemtest - grundsätzliche Vorgehensweise:

- Nach abgeschlossenem Integrationstest und vor dem Abnahmetest erfolgt der Systemtest beim Softwareentwickler durch Kunden (**α-Test**)
- Variante: Systemtests bei ausgewählten Pilotkunden vor Ort (**β-Test**)
- Systemtest überprüft aus der **Sicht des Kunden**, ob das Gesamtprodukt die an es gestellten Anforderungen erfüllt (nicht mehr aus Sicht des Entwicklers)
- anstelle von Testtreibern und Teststummeln kommen nun soweit möglich immer die realen (Hardware-)Komponenten zum Einsatz
- Systemtest sollte nicht beim Kunden in der Produktionsumgebung stattfinden, sondern in möglichst realitätsnaher Testumgebung durchgeführt werden
- beim Test sollten die **tatsächlichen Geschäftsprozesse** beim Kunden berücksichtigt werden, in die das getestete System eingebettet wird
- dabei durchgeführt werden: Volumen- bzw. Lasttests (große Datenmengen), Stresstests (Überlastung), Test auf Sicherheit, Stabilität, Robustheit, ...

Systemtest - nichtfunktionale Anforderungen:

- **Lasttest**: Messung des Systemverhaltens bei steigender Systemlast
- **Performanztest**: Messung der Verarbeitungsgeschwindigkeit unter bestimmten Randbedingungen
- **Kompatibilität**: Verträglichkeit mit vorhandenen anderen Systemen, korrekter Im- und Export externer Datenbestände, ...
- **Benutzungsfreundlichkeit**: übersichtliche Oberfläche, verständliche Fehlermeldungen, Hilfetexte, ... - für die jeweilige Benutzergruppe
- **Benutzerdokumentation**: Vollständigkeit, Verständlichkeit, ...
- **Änderbarkeit, Wartbarkeit**: modulare Systemstruktur, verständliche Entwickler-Dokumentation, ...
- ...

Akzeptanztest (Abnahmetest):

- Es handelt sich um eine **spezielle Form des Systemtests**
  - ⇨ der Kunde ist mit einbezogen bzw. führt den Test durch
  - ⇨ der Test findet beim Kunden, aber in Testumgebung statt (Test in Produktionsumgebung zu gefährlich)
- auf Basis des Abnahmetests entscheidet Kunde, ob das bestellte Softwaresystem **mangelfrei** ist und die im Lastenheft festgelegten Anforderungen erfüllt
- die durchgeführten Testfälle sollten bereits im **Vertrag** mit dem Kunden spezifiziert sein
- im Rahmen des Abnahmetests wird geprüft, ob System von allen relevanten Anwendergruppen **akzeptiert** wird
- im Rahmen von sogenannten **Feldtests** wird darüber hinaus ggf. das System in verschiedenen Produktionsumgebungen getestet

Testen, testen, … - wann ist Schluss damit?

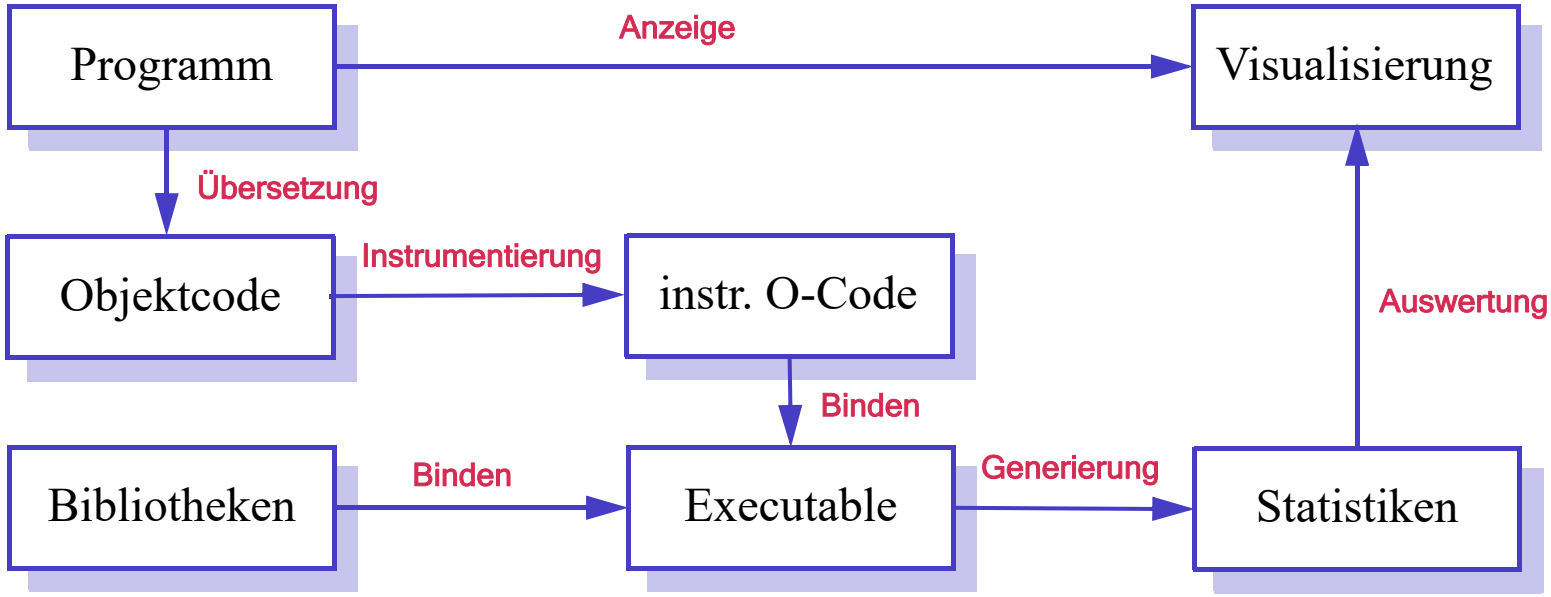
- **nie** - nach jeder Programmänderung wird eine große Anzahl von Testfällen automatisch ausgeführt (siehe Regressionstest)
-

Die sieben Grundsätze des Testens nach [SL19]:

1. Grundsatz:
   Testen zeigt die Anwesenheit von Fehlern (und nie die Abwesenheit)
2. Grundsatz:
   Vollständiges Testen ist nicht möglich
3. Grundsatz:
   Mit dem Testen frühzeitig beginnen
4. Grundsatz:
   Häufung von Fehlern (in bestimmten Programmteilen)
5. Grundsatz:
   Zunehmende Testresistenz (gegen existierende Tests)
6. Grundsatz:
   Testen ist abhängig vom Umfeld
7. Grundsatz:
   Trugschluss: Keine Fehler bedeutet ein brauchbares System

4.2 Laufzeit- und Speicherplatzverbrauchsmessungen

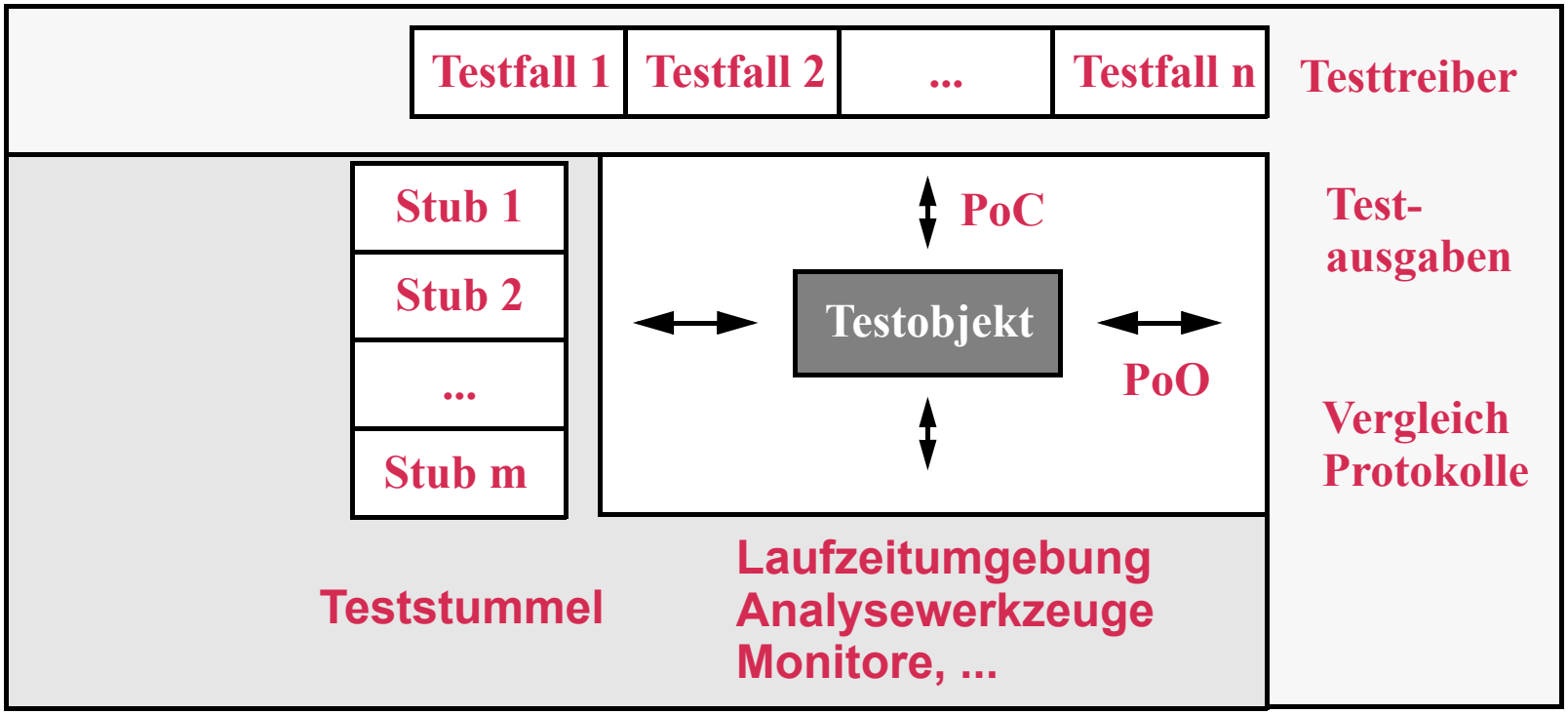
Gemeinsame Eigenschaft aller hier vorgestellten Werkzeuge/Verfahren ist:



⇨ Objektcode für untersuchte Software wird vor Ausführung „instrumentiert“ (um zusätzliche Anweisungen ergänzt)

⇨ zusätzliche Anweisungen erzeugen während der Ausführung statistische Daten über Laufzeitverhalten, Speicherplatzverbrauch, ...

Zur Erinnerung: Aufbau eines Testrahmens gemäß [SL19]:



- **Point of Control (PoC)**:
  Schnittstelle, über die Testobjekt mit Testdaten versorgt wird‘
- **Point of Observation (PoO)**:
  Schnittstelle, über die Reaktionen/Ausgaben des Testobjekts beobachtet werden

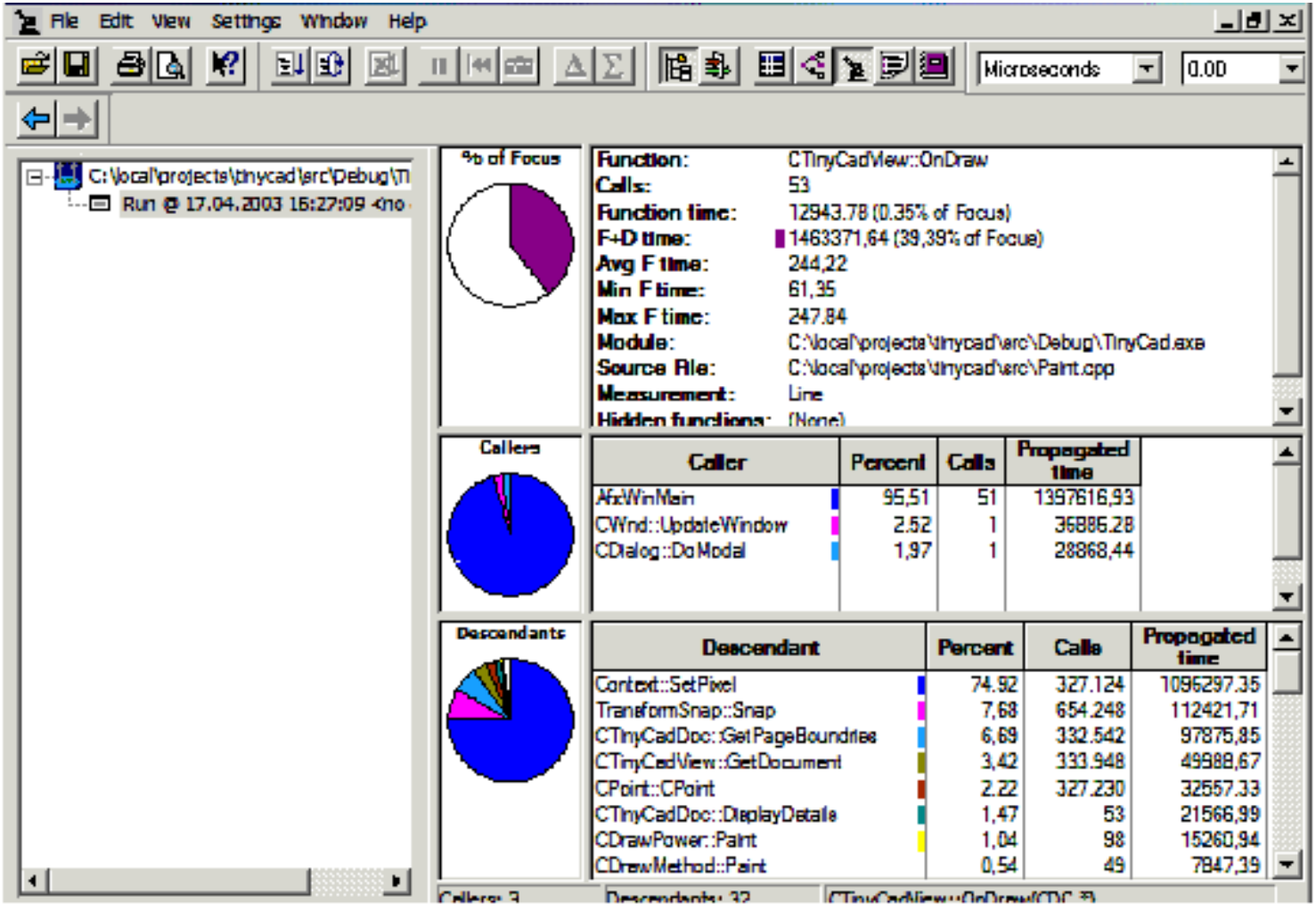
Untersuchung des Laufzeitverhaltens eines Programms:

- wie oft wird jede Operation aufgerufen (oder Quellcodezeile durchlaufen)
- welche Operation ruft wie oft welche andere Operation (**descendants**) auf oder von welchen Operationen (**callers**) wird ein Programm wie oft gerufen
- wieviel Prozent der Gesamtlaufzeit wird mit Ausführung einer bestimmten Operation verbracht (ggf. aufgeteilt nach callers und descendants)
- ...

Nutzen der ermittelten Daten:

- Operationen, die am meisten Laufzeit in Anspruch nehmen, können leicht identifiziert (und optimiert) werden
- tatsächliche Aufrufabhängigkeiten werden sofort sichtbar
- ...

Laufzeitanalyse mit Quantify (Rational/IBM):



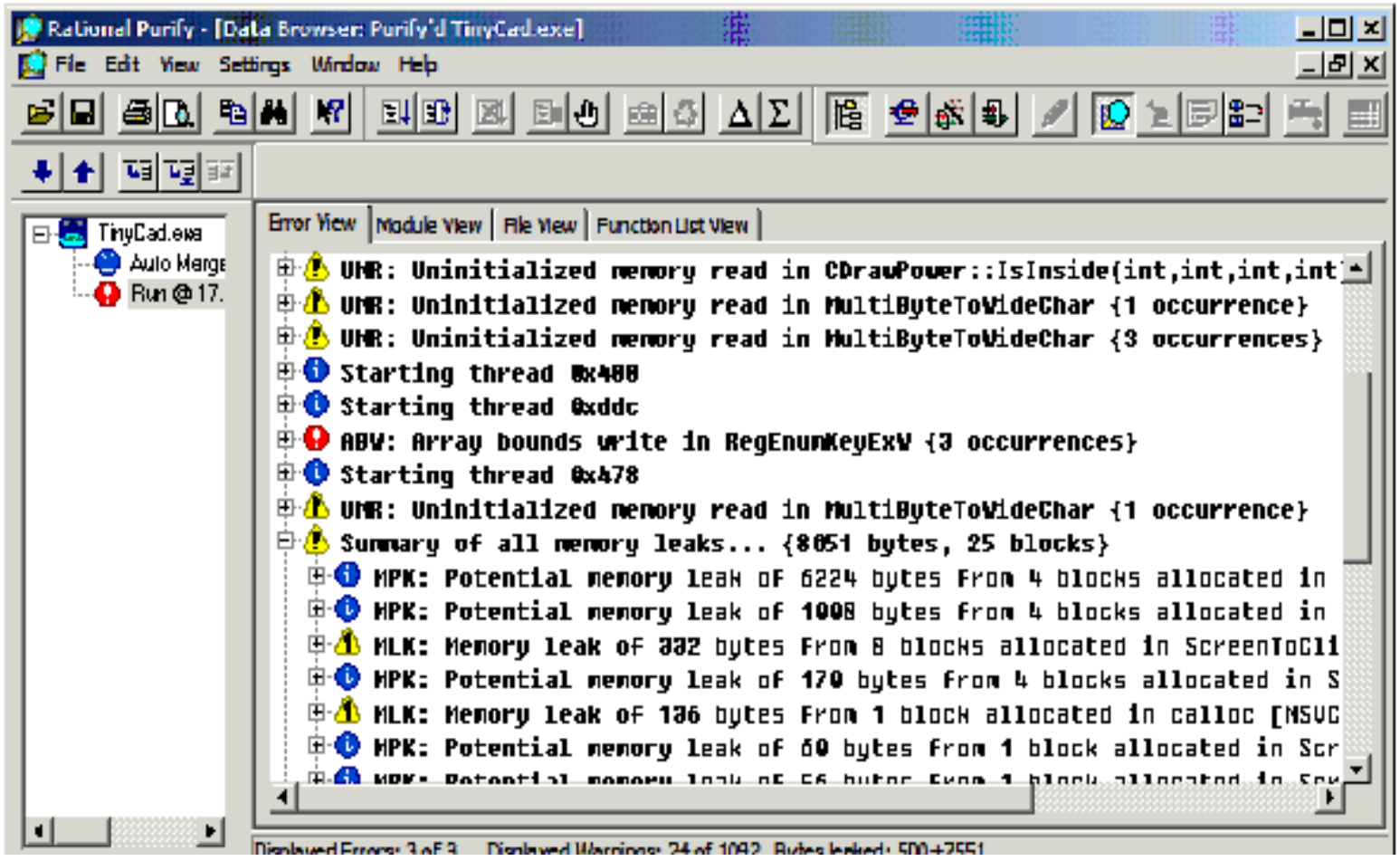
Erläuterungen zu Quantify-Beispiel:

- untersucht wird die Methode (Operation, Funktion) OnDraw
- im untersuchten Testlauf wurde OnDraw 53 mal aufgerufen
- die Ausführung der Methode OnDraw (inklusive gerufener Methoden) braucht 39% der Gesamtlaufzeit
- der Code der Methode OnDraw selbst benötigt aber nur 0,39% der Laufzeit
- die Ausführungszeit variiert sehr stark (61 bis 247 ms), Durchschnitt liegt aber nahe beim Maximum mit 244 ms
- die Methode OnDraw wird von Main 51 mal sowie von UpdateWindow und DoModal jeweils einmal im Beispiellauf gerufen
- die Methode OnDraw ruft ihrerseits die Methoden SetPixel, Snap, ...
- am meisten Zeit benötigen die 327.124 Aufrufe von SetPixel, nämlich fast 75% der Zeit

Untersuchung des Speicherplatzverhaltens eines Programms:

- welche Operationen fordern wieviel Speicherplatz an (geben ihn frei)
- wo wird Freigabe von Speicherplatz vermutlich bzw. bestimmt vergessen (memory leak = Speicherloch):
  - ⇨ bestimmt vergessen: Objekt lebt noch, kann aber nicht mehr erreicht werden (Garbage Collector von Java würde es entsorgen)
  - ⇨ vermutlich vergessen: Objekt lebt noch und ist erreichbar, wird aber nicht mehr benutzt (Garbage Collector von Java kann es nicht freigeben)
- wo wird auf bereits freigegebenen Speicherplatz zugegriffen (nur für C++) bzw. wo wird Speicherplatz mehrfach freigegeben (nur für C++)
- wo wird auf nicht initialisierten Speicherplatz zugegriffen; anders als bei der statischen Programmanalyse wird für jede Feldkomponente (Speicherzelle) getrennt Buch darüber geführt
- wo finden Zugriffe jenseits der Grenzen von Arrays statt (Laufzeitfehler in guter Programmiersprache)

Analyse des Speicherplatzverhalten mit Purify (Rational/IBM):

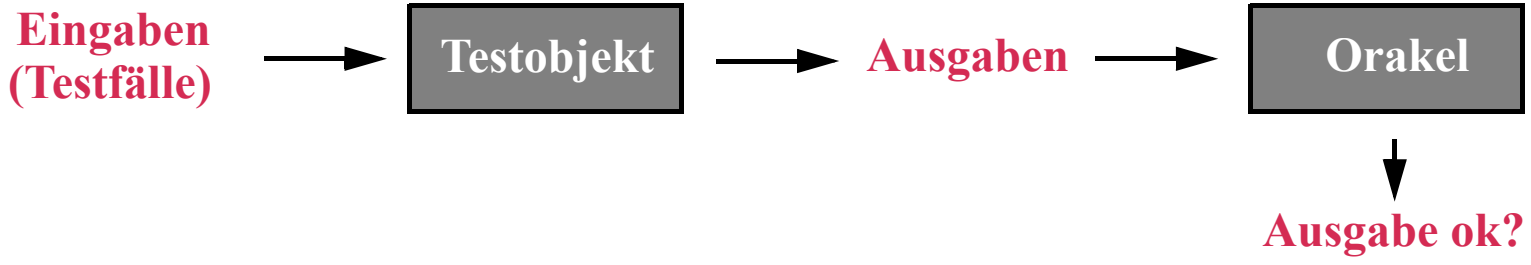


Achtung: so viele Speicherlöcher, ... ist typisch für Programmierung in C++

4.3 Funktionsorientierte Testverfahren (Blackbox)

Sie testen Implementierung gegen ihre Spezifikation und lassen die interne Programmstruktur unberücksichtigt (Programm wird als „Black-Box“ behandelt):

- für **Abnahmetest** ohne Kenntnis des Quellcodes geeignet
- setzt (eigentlich) vollständige und widerspruchsfreie **Spezifikation** voraus (zur Auswahl von Testdaten und Interpretation von Testergebnissen)
- **repräsentative Eingabewerte** müssen ausgewählt werden (man kann im allgemeinen nicht alle Eingabekombinationen testen)
- man braucht „**Orakel**“ für Überprüfung der Korrektheit der Ausgaben (braucht man allerdings bei allen Testverfahren)



Kriterien für die Auswahl von Testdaten:

An der Spezifikation orientierte **Äquivalenzklassenbildung**, so dass für alle Werte einer Äquivalenzklasse (Eingabewertklasse) sich das Softwareprodukt „gleich“ verhält:

- Unterteilung in Klassen von Eingabewerten, für die das Programm sich laut Spezifikation gleich verhalten muss
- Alle Klassen von Eingabewerten zusammen müssen den ganzen möglichen Eingabewertebereich des betrachteten Programms abdecken
- Aus jeder Äquivalenzklasse wird mindestens ein repräsentativer Wert getestet
- Unterteilung in gültige und ungültige Eingabewerte (fehlerhafte Eingaben, ...) wird durchgeführt und später bei der Auswahl von Testwerten berücksichtigt
- Oft gibt es auch eine gesonderte Betrachtung von Äquivalenzklassen für besonders „große“ oder besonders „kleine“ gültige Eingaben (Lasttest)
- Hier nicht mit betrachtet wird die Problematik der Suche nach Eingabewerteklassen, die zu bestimmten (Klassen von) Ausgabewerten führen

Regeln für die Festlegung von Eingabewerteklassen:

- für geordnete Wertebereiche:
  - ⇨]uv .. ov[ist ein offenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören nicht dazu)
  - ⇨ [uv .. ov] ist ein geschlossenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören dazu)
  - ⇨ Mischformen]uv .. ov] und [uv .. ov[sind natürlich erlaubt
- für ganze Zahlen (Integer):
  - ⇨ [MinInt .. ov] für Intervalle mit kleinster darstellbarer Integer-Zahl
  - ⇨ [uv .. MaxInt] für Intervalle mit größter darstellbarer Integer-Zahl
  - ⇨ offene Intervallgrenzen sind natürlich erlaubt
- für reelle Zahlen (Float) mit Voraussetzung kleinste/größte Zahl nicht darstellbar:
  - ⇨]- ∞ .. ov] oder]- ∞ .. ov[für nach unten offene Intervalle
  - ⇨ [uv .. ∞ [oder]uv .. ∞ [für nach oben offene Intervalle
  - ⇨ alle Mischformen von Intervallen mit festen unteren und oberen Grenzen

Regeln für die Festlegung von Eingabewerteklassen - Fortsetzung:

- für Zeichenketten (String):
  - ⇨ Definition über reguläre Ausdrücke oder Grammatiken
  - ⇨ Aufzählung konkreter Werte (siehe nächster Punkt)
- für beliebige Wertebereiche:
  - ⇨ {v1 v2 v3 ... vn} für Auswahl von genau n verschiedenen Werten
- für zusammengesetzte Wertebereiche:
  - ⇨ Anwendung der obigen Prinzipien auf die einzelnen Wertebereiche
  - ⇨ ggf. braucht man noch zusätzliche Einschränkungen (Constraints), die nur bestimmte Wertekombinationen für die Teilkomponenten zulassen
- Eingabewerteklassen, die aus genau einem Wert bestehen:
  - ⇨ [v] es ist aber auch die Repräsentation {v} oder v üblich

Auswahl von Testdaten aus Eingabewerteklassen:

- aus jeder Eingabewerteklasse wird mindestens ein Wert ausgewählt (Fehler- und Lasttestklassen werden später gesondert behandelt)
- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (siehe ISTQB-Prüfung; im Folgenden werden aus Platzgründen bei den Beispielen aber nur Grenzwerte selbst ausgewählt)
- als Intervalle dargestellte Eingabewerteklassen werden oft durch die so genannte **Grenzwertanalyse** nochmal in Unterklassen/Teilintervalle zerlegt:
  - ⇨]uv .. ov[wird zerlegt in [inc(uv)]]inc(uv) .. dec(ov)[[dec(ov)]
  - ⇨ [uv .. ov] wird zerlegt in [uv]]uv .. ov[[ov]
  - ⇨ ...
  - ⇨ inc(uv) liefert den nächstgrößeren Wert zu uv
  - ⇨ dec(ov) liefert den nächstkleineren Wert zu ov
  - ⇨ Achtung: bei Float muss für inc und dec die gewünschte Genauigkeit festgelegt werden, mit der aufeinanderfolgende Werte gewählt werden

Zusätzliche Wahl von Testdaten durch Grenzwertanalyse:

Bilden Eingabewerteklassen einen Bereich (Intervall), so selektiert die Grenzwertanalyse also immer Werte um die Bereichsgrenzen herum:

- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (im Folgenden aus Platzgründen weggelassen)
- Idee dabei: oft werden Schleifen über Intervallen gebildet, die genau für die Grenzfälle falsch programmiert sind
- Beispiel für die Grenzwertanalyse:
  zulässiger Eingabebereich besteht aus reellen Zahlen (Float) mit zwei zu betrachtenden Nachkommastellen zwischen -1 und +1 (Fehlerklassen in **Rot**):
  - ⇨ vor Grenzwertanalyse:
    ]- ∞ .. -1.0[[-1.0 .. +1.0]]+1.0 .. + ∞ [
  - ⇨ nach Grenzwertanalyse:
    ]- ∞ .. -1.01[[-1.01] [-1.0]]-1.0 .. +1.0[[+1.0] [+1.01]]+1.01 .. + ∞ [
  - ⇨ Beispiele für gewählte Werte: -2, -1.01, -1.0, 0, +1.0, +1.01, +2

Auswahl von Testeingaben für countVowels (mit Zeichenketten!):

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
  (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
```

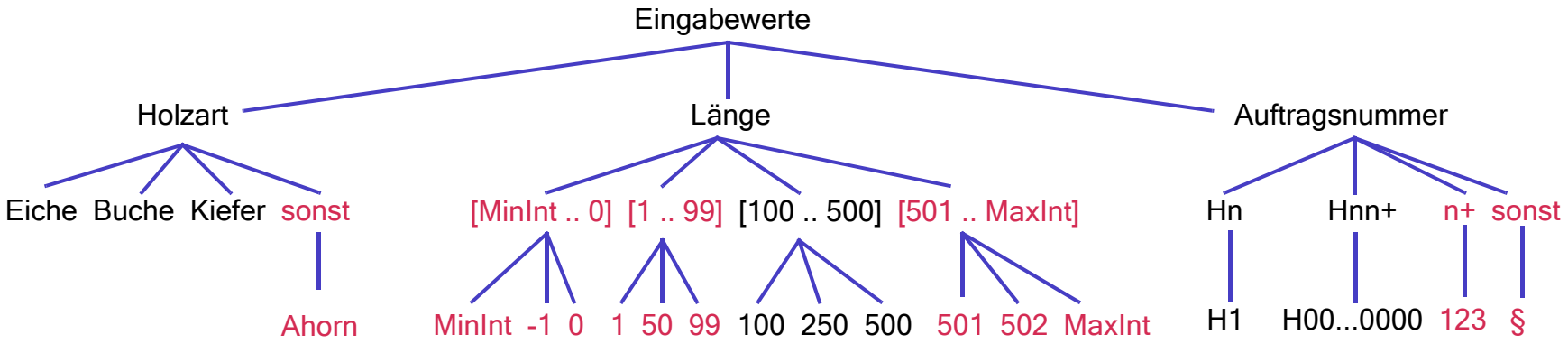
Beispiel Lagerverwaltung einer Baustoffhandlung - Holzbretter:

- bei Holzbrettern wird die Holzart eingegeben (als String)
- Eiche, Kiefer und Buche sind bekannt
- es wird die Brettlänge eingegeben (zwischen 100 und 500 cm)
- jede Lieferung erhält eine Auftragsnummer, die mit „H“ beginnt, gefolgt von mindestens einer (aber beliebig vielen) Ziffer(n)
- aus Brettlänge und Holzart errechnet sich der Preis des Auftrags

Weitere Regeln für die Bildung von Äquivalenzklassen bei Intervallen:

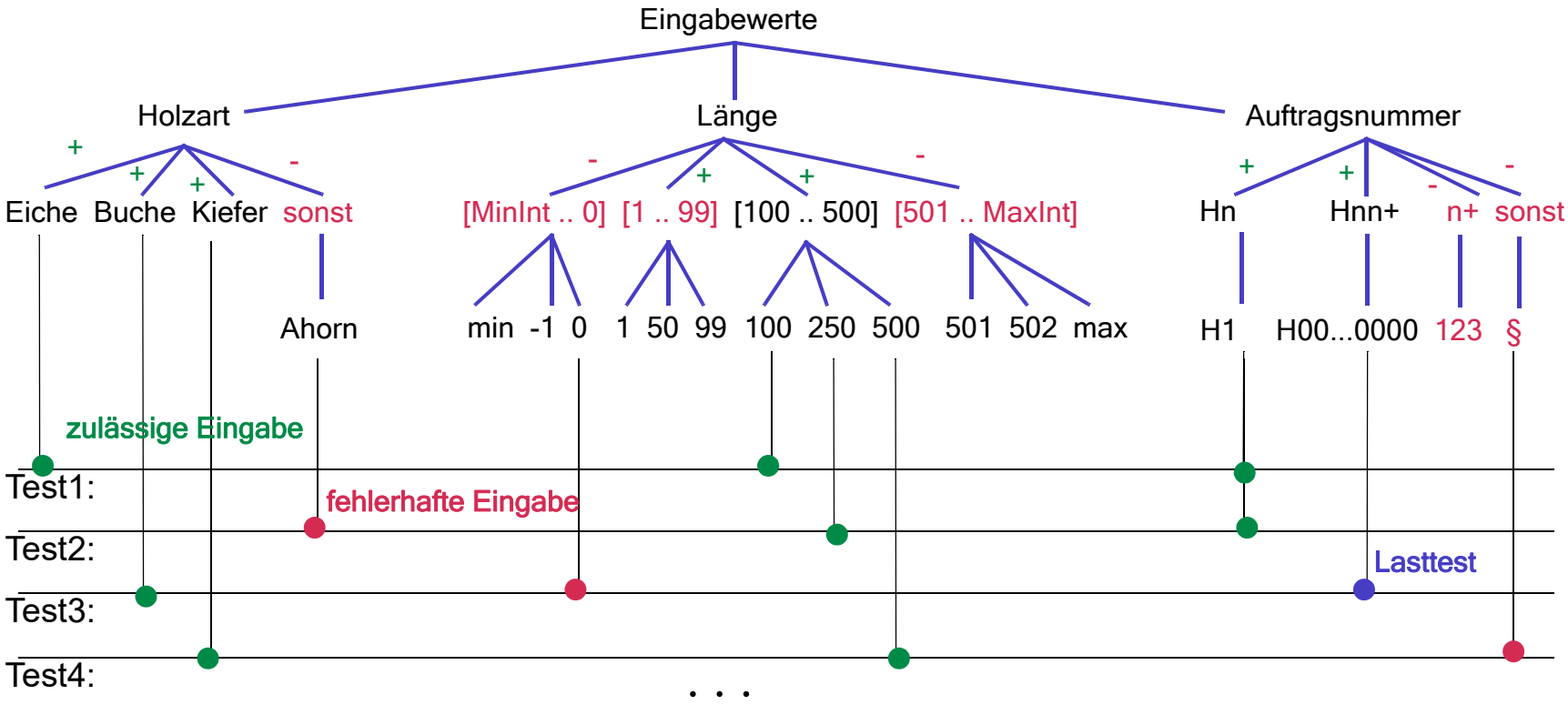
- aus Äquivalenzklassen, die (geschlossene) Intervalle sind, werden jeweils die beiden Grenzwerte und ein weiterer Wert (z.B. aus der Mitte des Intervalls) ausgewählt (ohne platzraubenden Zwischenschritt der Zerlegung in drei Teilintervalle)
- einelementige Äquivalenzklasse und Wert aus dieser Äquivalenzklasse werden nicht unterschieden, also statt [v] schreiben wir gleich v
- reguläre Ausdrücke werden zur Definition v. String-Äquivalenzklassen eingesetzt

Grafische Darstellung von Äquivalenzklassen als Klassifikationsbaum:



- zunächst wird „Eingabewerte“ in alle Eingabeparameter des Programms zerlegt
- zusammengesetzte Eingabeparameter werden weiter zerlegt (nicht im Beispiel)
- schließlich wird der Wertebereich eines atomaren Eingabeparameters betrachtet
- der Wertebereich wird in Äquivalenzklassen „ähnlicher“ Werte zerlegt
- dabei werden auch nicht erlaubte Eingabewerte (Fehlerklassen) betrachtet
- ebenso werden „extreme“ erlaubte Werte (Lasttestklassen) betrachtet
- aus jedem Wertebereich werden Repräsentanten für den Test ausgewählt

Unvollständige/fehlerhafte Auswahl von Eingabewertekombinationen:



Problem: trotz Bildung von Äquivalenzklassen bleiben (zu) viele mögliche Eingabewertekombinationen (im Beispiel sind 4 * 12 * 4 verschiedene Testläufe möglich).

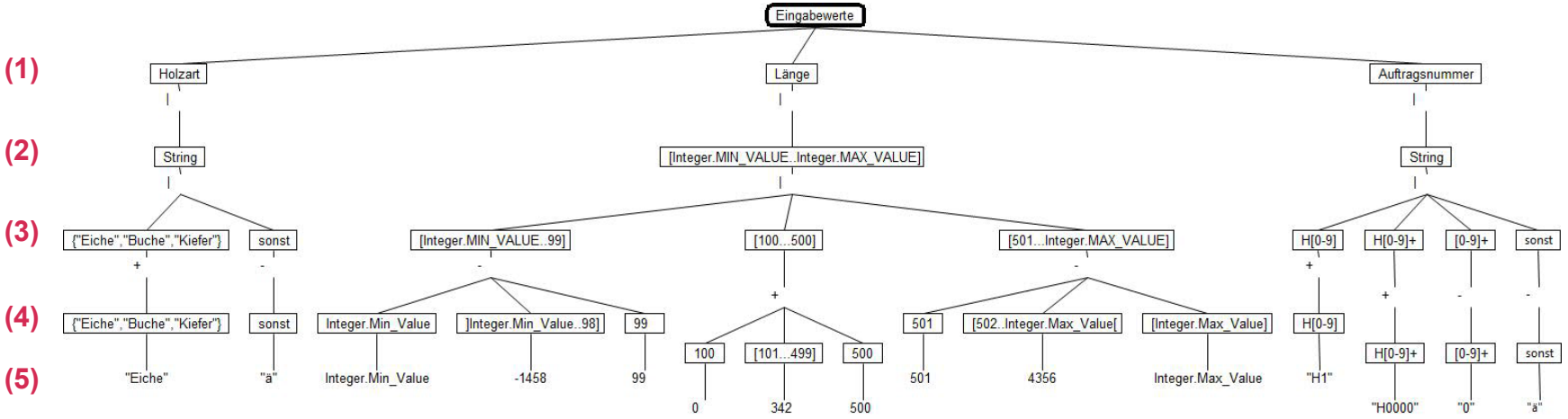
Heuristiken für die Reduktion möglicher Eingabewertekombinationen:

- aus jeder **Äquivalenzklasse** wird *mindestens* einmal ein Wert ausgewählt; es gibt also jeweils mindestens einen Testfall, in dem ein Eingabewert der Äquivalenzklasse verwendet wird (bei Grenzwertanalyse werden drei Werte ausgewählt)
- bei **abhängigen Eingabeparametern** müssen Testfälle für **alle Kombinationen** ihrer jeweiligen „normalen“ Äquivalenzklassen aufgestellt werden; Parameter sind abhängig, wenn sie gemeinsam das Verhalten des Programms steuern (und deshalb nicht unabhängig voneinander betrachtet werden können)
- der ausgewählte **Wert** einer **Fehleräquivalenzklasse** wird in genau einem Testfall verwendet (Fehleräquivalenzklassen sind solche Äquivalenzklassen, die unzulässige Eingabewerte zusammenfassen)
- der ausgewählte **Wert** einer **Lasttestklasse** wird ebenfalls genau einmal in einem Testfall verwendet (Lasttestklassen sind solche Äquivalenzklassen, die besonders „große/lange/...“ zulässige/gültige Eingabewerte zusammenfassen)
- hat man mehrere Eingabeparameter, wird **höchstens einer** mit einem Wert aus einer Fehler- oder Lasttestklasse belegt (verhindert Verdeckung von Fehlern)

Sinnvolle Auswahl von Testfällen am Beispiel demonstriert:

- die fehlerhaften Auftragsnummern H und 123 treten nur in jeweils einem Testfall auf, die fehlerhafte Eingabe Ahorn ebenfalls
- gleiches gilt für eine zu kleine und eine zu große Längenangabe
- die extrem lange Auftragsnummer H00...0000 tritt ebenfalls nur einmal in einem Testfall auf
- zu kleine und zu große Längenangaben, fehlerhafte Auftragsnummern, falsche Holzarten und die extrem lange Auftragsnummer treten nicht im selben Testfall auf
- alle zulässigen Kombinationen von Äquivalenzklassen für Holzarten und Längen-angaben müssen durchgetestet werden, da laut Spezifikation Holzart und Länge gemeinsam den Preis des Auftrags bestimmen
- da Auftragsnummern laut Spezifikation der Software unabhängig von Holzarten und Längen verarbeitet werden, wird jede Kombination von Holzart und Länge nur mit einer normalen Auftragsnummer getestet

„Normierter“ Aufbau von Klassifikationsbäumen für Übung / Klausur:



1. Ebene: alle Parameter(-namen) der betrachteten Funktion
2. Ebene: die Typen bzw. Wertebereiche der Parameter
3. Ebene: die Zerlegung der Wertebereiche in
   - Äquivalenzklassen für erlaubte Werte (mit „+“ markiert)
   - Fehleräquivalenzklassen (mit „-“ markiert)
4. Ebene: weitere Zerlegung der Wertebereiche mit Grenzwertanalyseverfahren
5. Ebene: konkrete Repräsentanten (Werte) der jeweiligen Äquivalenzklassen

Geändertes Beispiel für Testparameterauswahl:

Wieder soll eine Vorschrift zur Berechnung des Preises von Brettern getestet werden. Die Eingabeparameter sind dieses Mal:

- Holzart (Aufzählungstyp): Buche, Eiche, Kiefer
- Brettlänge (Zahl größer gleich Null) mit folgenden Intervallen, in denen unterschiedliche Berechnungsschemata verwendet werden:
  kurze Bretter mit [0 .. 99] cm
  normale Bretter mit [100 .. 300] cm
  (zu) lange Bretter mit]300 .. $+\infty$ [cm
- Auftragsart (Aufzählungstyp): DoItYourself, Normal, Express

Alle drei Parameter interagieren bei der Berechnung des Preises eines Bretts; also müssten eigentlich zumindest alle

3 x 3 x 3 = 27 Kombinationen v. Äquivalenzklassen

getestet werden.

Paarweiser Testansatz (Pairwise Testing):

Die Praxis zeigt, dass ca. 80% aller von bestimmten Parameterwertkombinationen ausgelösten Softwarefehler bereits durch Wahl bestimmter Paarkombinationen beobachtet werden können. Also werden beim „paarweisen“ Testen einer Funktion mit n Parametern nicht alle möglichen Kombinationen überprüft, sondern nur alle paarweisen Kombinationen.

Beispiel:

Holzart	Buche	Buche	Buche	Eiche	Eiche	Eiche	Kiefer	Kiefer	Kiefer
Länge	kurz	norm	lang.	kurz	norm	lang.	kurz	norm	lang
Auftragsart	DoIt	Norm	Expr.	Expr.	DoIt	Norm	Norm	Expr.	DoIt

Im Beispiel werden für die Erzeugung aller möglichen paarweisen Kombinationen von Äquivalenzklassen von Testwerten nur 9 Testfälle (Testvektoren) benötigt anstelle der 27 möglichen 3er-Kombinationen!

Bewertung der funktionalen Äquivalenzklassenbildung:

- Güte des Verfahrens hängt stark von **Güte der Spezifikation** ab (Aussagen über erwartete/zulässige Eingabewerte und Ergebnisse)
- Verwaltung von Äquivalenzklassen und Auswahl von Testfällen kann durch **Werkzeugunterstützung** vereinfacht werden
- **ausführbare Modelle** (z.B. in UML erstellt) können bei der Auswahl von Äquivalenzklassen, Testfällen helfen (und als Orakel verwendet werden)
- Test von Benutzeroberflächen, zeitlichen Beschränkungen, Speicherplatzbeschränkungen etc. (**nichtfunktionale Anforderungen**) wird kaum unterstützt
- mindestens **einmalige Ausführung** jeder Programmzeile wird nicht garantiert
- Test **zustandsbasierter Software** (Ausgabeverhalten hängt nicht nur von Eingabe, sondern auch von internem Zustand ab) geht so nicht
  - ⇨ später Spezifikation und Testplanung mit Hilfe von Automaten/Statecharts
  - ⇨ Grenzfall zwischen funktionalem und strukturorientiertem Softwaretest

Weitere Black-Box-Testverfahren - 1:

- **Erfahrungsbasierte Testverfahren:**
  - ⇨ **Intuitives Testen (Error Guessing)**: ein Testansatz, bei dem Testfälle auf Basis des Wissens der Tester über frühere Fehler oder allgemeines Wissen über Fehlerwirkungen (irgendwie) abgeleitet werden
  - ⇨ **Exploratives Testen**: ein Testansatz bei dem die Tester, basierend auf ihrem Wissen, der Erkundung des Testobjekts und dem Ergebnis früherer Tests, dynamisch neue Tests entwickeln
  - ⇨ **Checklistenbasiertes Testen**: ein Testansatz bei dem erfahrene Tester eine Liste von Kontrollpunkten (Checkliste) bzw. Regeln oder Kriterien (für das Testobjekt) zur Steuerung des Testprozesses nutzen
- **Zufallstest**: wählt aus Menge der möglichen Werte eines Eingabedatums zufällig Repräsentanten aus (ggf. gemäß bekannter statistischer Verteilung)
  - ⇨ zur Ergänzung gut geeignet, generiert oft „unerwartete“ Testdaten
- **Smoke-Test**: es wird nur Robustheit des Testobjekts getestet, berechnete Ausgabewerte spielen keine Rolle (auf Tastatur hämmern, ...)

Weitere Black-Box-Testverfahren - 2:

- **Syntax-Test**: ist für Eingabewerte der erlaubte syntaktische Aufbau bekannt (als Grammatik angegeben) kann man daraus systematisch Testfälle generieren; Beispiele:
  - ⇨ syntaktisch korrekte Email-Adressen
  - ⇨ zulässige Dateinamen, Verzeichnispfade
  - ⇨ Aufbau von Zahlen (Integers, Floats)
  - ⇨ ...
- **Zustandsbezogener Test**:
  siehe Abschnitt 4.6
- **Ursache-Wirkungs-Graph-Analyse / Entscheidungstabellenbasiertes Testen**:
  siehe folgende Folien
- **Anwendungsfallbasiertes Testen**:
  siehe folgende Folien

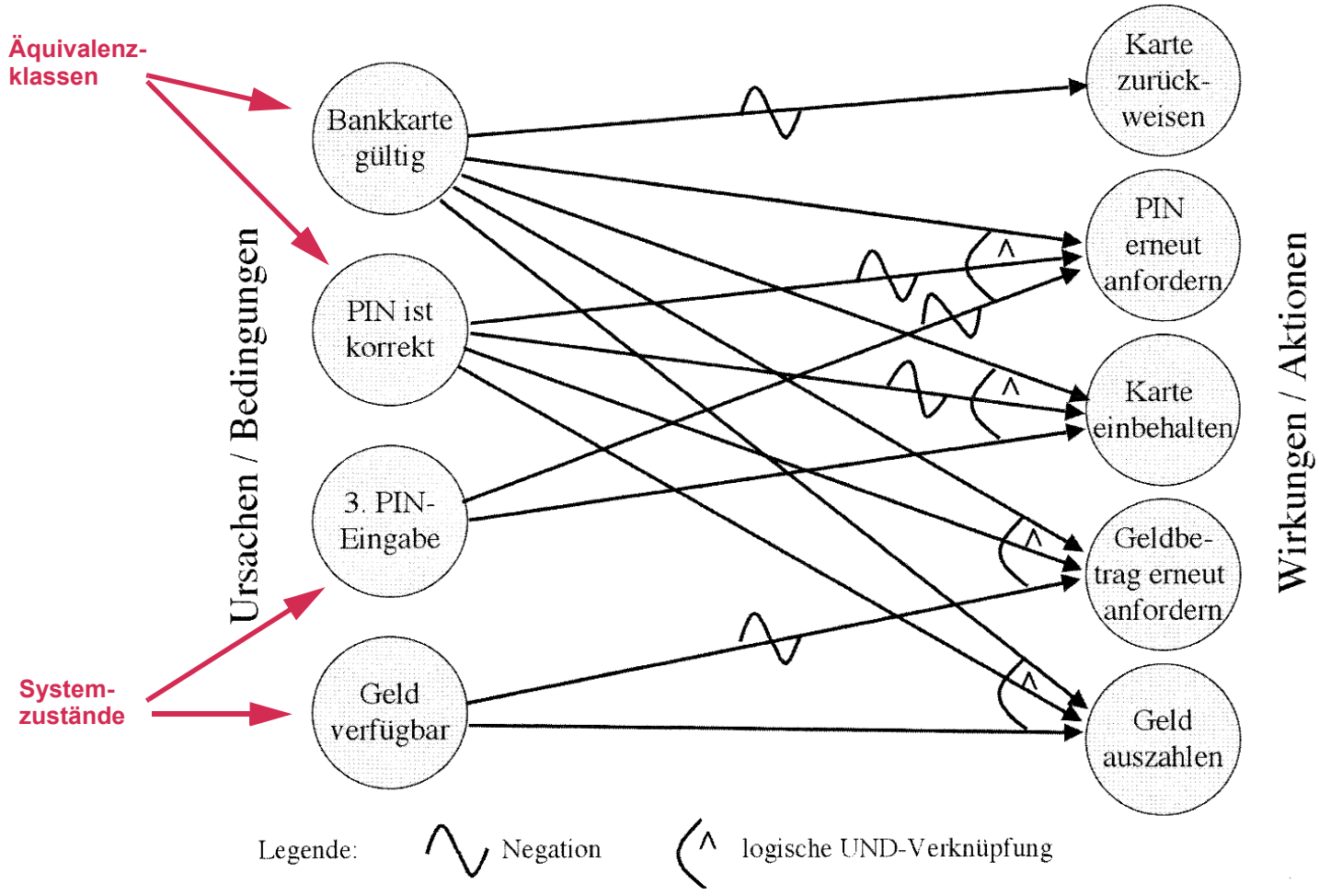
Ursache-Wirkungs-Graph-Analyse-Verfahren aus [SL19]:

Es handelt sich eigentlich um eine Kombination von zwei Verfahren. Die Basis bilden sogenannte „**Entscheidungstabellen**“, die Bedingungen an Eingaben und ausgelöste Aktionen eines Systems miteinander verknüpfen. Für die systematische Erstellung einer Entscheidungstabelle wird zunächst ein „**Ursache-Wirkungs-Graph**“ erstellt.

Ein Ursache-Wirkungs-Graph verknüpft Eingaben = **Ursachen** / Bedingungen mit daraus resultierenden Ausgaben = **Wirkungen** / Aktionen (durch grafische Darstellung aussagenlogischer Ausdrücke). Die weitere Vorgehensweise ist wie folgt:

1. Eine Wirkung wird ausgewählt.
2. Zu der Wirkung werden alle Kombinationen von Ursachen gesucht, die diese Wirkung hervorrufen.
3. Für jede gefundene Ursachenkombination wird eine Spalte der Entscheidungstabelle erzeugt.
4. Die Spalten der Entscheidungstabelle entsprechen Testfällen.
5. Die Zeilen der Entscheidungstabelle entsprechen allen Ursachen und Wirkungen.

Ursache-Wirkungs-Graph-Beispiel aus [SL19]:



Entscheidungstabellen-Beispiel aus [SL19]:

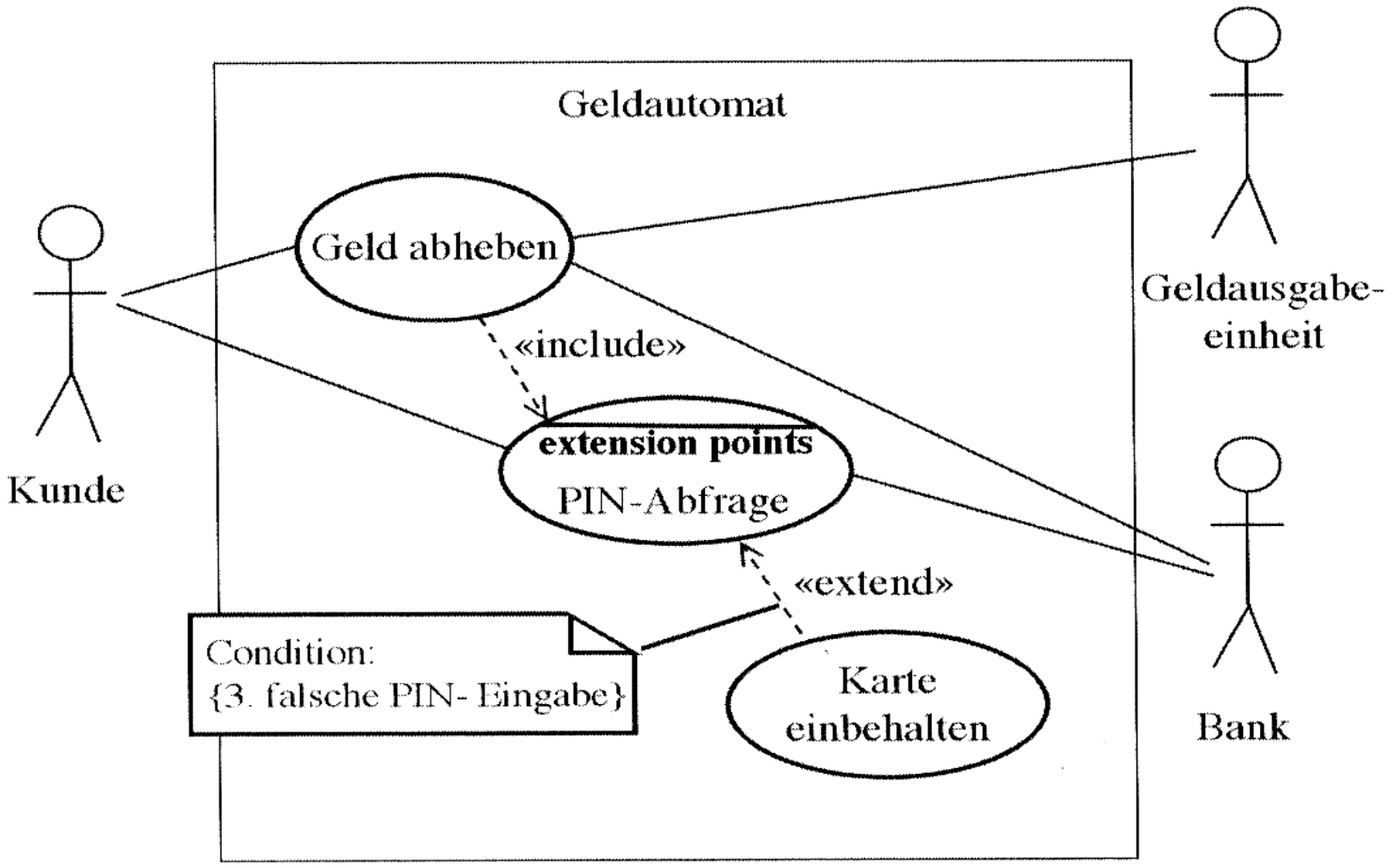
Entscheidungstabelle		TF1	TF2	TF3	TF4	TF5
Bedingungen	Bankkarte gültig?	Nein	Ja	Ja	Ja	Ja
	PIN ist korrekt?	–	Nein	Nein	Ja	Ja
	Dritte PIN-Eingabe?	–	Nein	Ja	–	–
	Geld verfügbar?	–	–	–	Nein	Ja
Aktionen	Karte zurückweisen	Ja	Nein	Nein	Nein	Nein
	PIN erneut anfordern	Nein	Ja	Nein	Nein	Nein
	Karte einbehalten	Nein	Nein	Ja	Nein	Nein
	Geldbetrag erneut anfordern	Nein	Nein	Nein	Ja	Nein
	Geld auszahlen	Nein	Nein	Nein	Nein	Ja

Anwendungsfallbasiertes Testen aus [SL19]:

Ausgangspunkt für diese Testmethodik ist die Beschreibung von sogenannten **Anwendungsfällen** (Nutzungsszenarien, Geschäftsvorfällen) eines Systems im Zuge der Anforderungsanalyse. Dabei kommt in der Regel die „Unified Modeling Language“ (UML) mit ihren Anwendungsfalldiagrammen zum Einsatz (siehe etwa Vorlesung „Software-Engineering - Einführung“):

- Anwendungsfalldiagramme erlauben die Beschreibung von Nutzungsszenarien (und damit von Akzeptanztestfällen) auf sehr hohem Abstraktionsniveau sowie die Unterscheidung zwischen normalen Abläufen und Ausnahmen.
- Werden einzelne Anwendungsfälle informell / in natürlicher Sprache beschrieben, so werden die dazugehörigen Testfälle „**manuell**“ erstellt.
- Werden für die Beschreibung einzelner Anwendungsfälle formalere Notationen wie Sequenz- oder Aktivitätsdiagramme der UML eingesetzt, so können Testwerkzeuge daraus **automatisch** Code für die Testfallausführung und -bewertung generieren.

Anwendungsfalldiagramm-Beispiel aus [SL19]:



Bewertung der Ursache-Wirkungs-Graph- und Anwendungsfall-Testens:

- Beide Methoden lassen sich sehr früh im Software-Lebenszyklus einsetzen und eignen sich insbesondere für die Erstellung von Akzeptanztestfällen.
- Die Ursache-Wirkungsgraph-Methode erlaubt die bei der Äquivalenzklassen-Bildung fehlende Verknüpfung von Eingaben und Ausgaben (Ursachen und Wirkungen).
- Das Anwendungsfallbasierte Testen erlaubt hingegen nicht nur die Beschreibung einzelner Testvektoren (Eingabewertkombinationen), sondern auch die Spezifikation ganzer Interaktionssequenzen zwischen Umgebung und zu testendem System.
- Insbesondere das Anwendungsfallbasierte Testen unterstützt aber nicht die systematische Identifikation fehlender Testfälle (für bestimmte Eingabetestvektoren).

Fazit:

Alle vorgestellten „Black-Box“-Testmethoden (Funktionsorientierte Testverfahren) besitzen ihre Stärken und Schwächen und ergänzen einander!!!

4.4 Kontrollflussbasierte Testverfahren (Whitebox)

Ausgangspunkt ist der Kontrollflussgraph einer Komponente; getestet werden sollen

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
VAR i: INTEGER;
BEGIN
   count := 0; i := 0;
   WHILE s[i] # '.' DO
      IF s[i]= 'b' OR s[i]= 'a' OR ... (* Kontrollflussfehler - falsche Prüfung auf 'b'. *)
      THEN
         count := 1;(* Berechnungsfehler - count wird nicht erhöht. *)
         i := i+1;(* Kontrollflussfehler - nur bedingte Inkr. von i  *)
      END;
   END (* WHILE *)
END countVowels
   …
countVowels('to be . . . or not to be.', count);(*Schnittstellenfehler - dot im Satz. *)
```

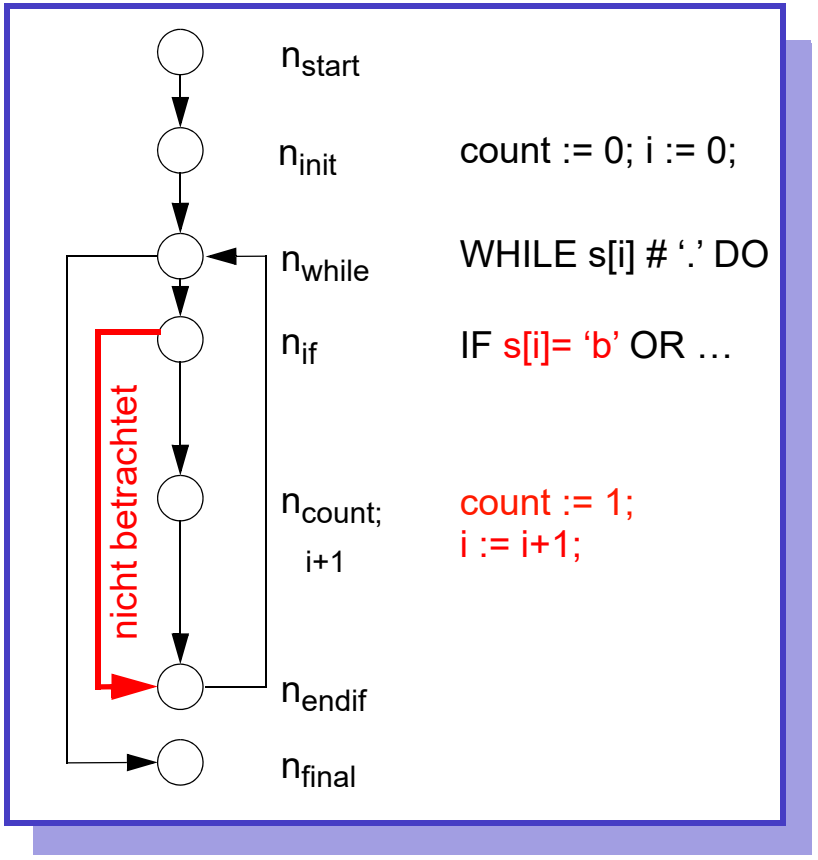
Grundideen des kontrollflussbasierten Testens:

- mit im Grunde zunächst beliebigen Verfahren werden **Testfälle festgelegt**
- diese Testfälle werden alle **ausgeführt** und dabei wird notiert, welche Teile des Programms durchlaufen wurden
- es gibt (meist) ein **Test-Orakel**, dass für jeden ausgeführten Testfall ermittelt, ob die berechnete Ausgabe (Verhalten des Programms) korrekt ist
- schließlich wird festgelegt, ob die vorhandenen Testfälle den Kontrollfluss des Programms hinreichend **überdecken**
- ggf. werden solange **neue Testfälle aufgestellt**, bis hinreichende Überdeckung des Quelltextes (Kontrollflusses) erreicht wurde
- ggf. werden **alte Testfälle gestrichen**, die dieselben Teile des Quelltextes (Kontrollflusses) überdecken

Kontrollflusstest - Anweisungsüberdeckung (C_0-Test):

Jeder Knoten des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- ❑ Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen traversiert werden
- ❑ viele Fehler bleiben unentdeckt



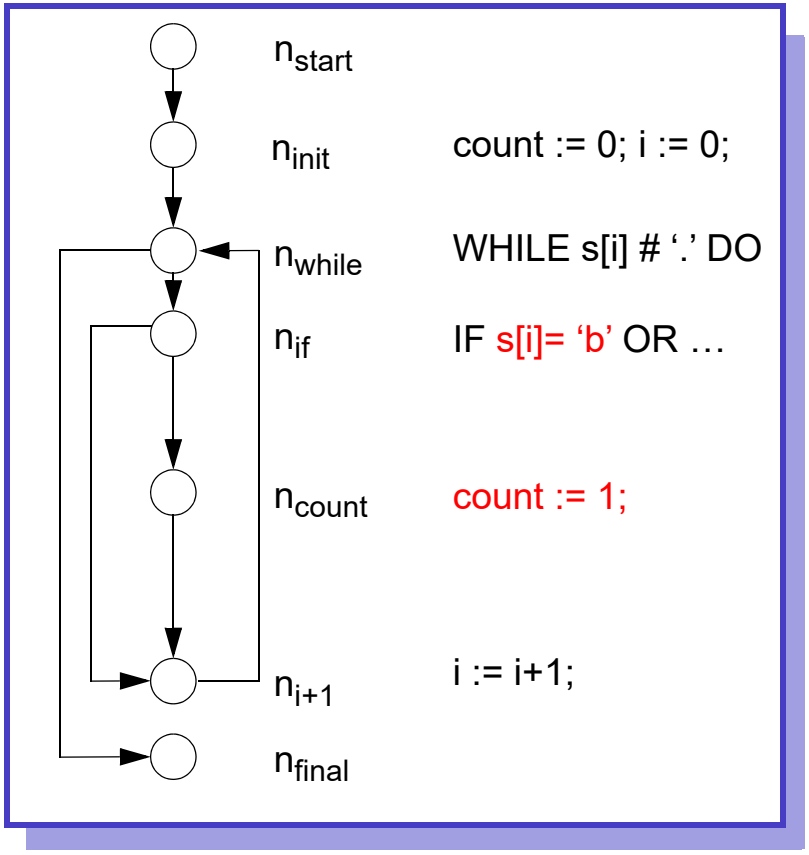
Beispiel:

- ☹ als Testfall genügt ein konsonantenfreier Satz wie etwa 'a.'
- ☹ Verschiebung von i := i+1; in if-Anweisung wird nicht entdeckt
- ☹ fehlerhafte Anweisung count := 1; wird nicht entdeckt
- ☹ fehlerhafte Bedingung wird nicht entdeckt

Kontrollflusstest - Zweigüberdeckung (C_1-Test):

Jede Kanten des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- realistisches Minimalkriterium
- umfasst Anweisungsüberdeckung
- Fehler bei Wiederholung oder anderer Kombination von Zweigen bleiben unentdeckt



Beispiel:

- ☹ Testfall 'ax.' für countVowels würde Kriterium bereits erfüllen
- ☹ Zuweisung count := 1; wird nicht als fehlerhaft erkannt
- ☹ ebenso nicht die Bedingung s[i] = 'b'

Kontrollflusstest - Entscheidungs-/Bedingungsüberdeckung:

Jede Teilbedingung einer Kontrollflussbedingung (z.B. von if- oder while-Anweisung) muss einmal den Wert true und einmal den Wert false annehmen.

atomare Bedingungsüberdeckung/-test:

- ⇨ keine Anforderung an Gesamtbedingung
- ⇨ bei falschem hasVowels (neues Beispiel!) reicht Testfall ‘baeiou.’
- ⇨ umfasst nicht mal Anweisungsüberdeckung

minimale Mehrfachbedingungsüberdeckung/-test:

- ⇨ jede Teil- und Gesamtbedingung ist einmal true und einmal false
- ⇨ bei falschem hasVowels_2 reicht Testfall ‘xbaeiou.’
- ⇨ Orientierung an syntaktischer Struktur von Kontrollflussbedingungen
- ⇨ Bedingung umfasst Zweigüberdeckung
- ⇨ trotzdem werden viele Bedingungsfehler nicht entdeckt

Atomare Bedingungsüberdeckung/-test mit ‘baeiou.´:

```
PROCEDURE hasVowels(s: Sentence; VAR exists: BOOLEAN);
   (* Returns true if sentence contains vowels. Sentence must be terminated by a dot. *)
VAR i: INTEGER;
BEGIN
   exists := FALSE; i := 0;
   WHILE s[i] # ‘.’ DO
      IF s[i]= ‘b’ OR s[i]= ‘a’ OR ... (* Kontrollflussfehler - Prüfung auf ‘b’ falsch. *)
      THEN
         exists := TRUE;
         i := i+1;(* Kontrollflussfehler - nur bedingte Inkr. von i  *)
      END;
   END (* WHILE *)
END hasVowels

   …
hasVowels(‘baeiou.’, count);
```

In diesem Fall ist die if-Bedingung bei jedem Schleifendurchlauf „wahr“, also hat man keine Zweigüberdeckung. Fehler werden nicht entdeckt!

Minimale Mehrfachbedingungsüberdeckung/-test mit ‘xbaeiou.´:

```
PROCEDURE hasVowels(s: Sentence; VAR exists: BOOLEAN);
   (* Returns true if sentence contains vowels. Sentence must be terminated by a dot. *)
VAR i: INTEGER;
BEGIN
   exists := FALSE; i := 0;
   WHILE s[i] # '.' OR NOT exists DO (* Kontrollflussfehler - OR statt AND falsch. *)
      IF s[i]= 'b' OR s[i]= 'a' OR ... (* Kontrollflussfehler - Prüfung auf 'b' falsch. *)
      THEN
         exists := TRUE;
      END;
      i := i+1;
   END (* WHILE *)
END hasVowels_2

   …
hasVowels_2('xbaeiou.', exists);
```

Man hat Zweigüberdeckung, Fehler werden trotzdem nicht gefunden! Besser wären hier viele kurze Eingaben mit **einem** Schleifendurchlauf anstelle einer langen Eingabe.

Modifizierter Bedingungsüberdeckungstest (MCDC):

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

Modifizierter Bedingungsüberdeckungstest (MCDC):

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)

Modifizierter Bedingungsüberdeckungstest (MCDC):

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)
- die Werte aller Teilbedingungen, die für das Gesamtergebnis der Bedingung irrelevant sind und deshalb ggf. wegen „short circuit“-Evaluation des Compilers nicht ausgewertet werden, werden als irrelevant gekennzeichnet (mit Zeichen „-“)

Modifizierter Bedingungsüberdeckungstest (MCDC):

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)
- die Werte aller Teilbedingungen, die für das Gesamtergebnis der Bedingung irrelevant sind und deshalb ggf. wegen „short circuit“-Evaluation des Compilers nicht ausgewertet werden, werden als irrelevant gekennzeichnet (mit Zeichen „-“)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen diese einmal auf true und einmal auf false und unterscheiden sich nur in der gerade betrachteten atomaren Teilbedingung (irrelevant kann mit true u. false gleichgesetzt werden)

Modifizierter Bedingungsüberdeckungstest (MCDC):

Der modifizierte Bedingungsüberdeckungstest (Definierter Bedingungstest, Minimaler Mehrfachbedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. „bessere“ Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)
- die Werte aller Teilbedingungen, die für das Gesamtergebnis der Bedingung irrelevant sind und deshalb ggf. wegen „short circuit“-Evaluation des Compilers nicht ausgewertet werden, werden als irrelevant gekennzeichnet (mit Zeichen „-“)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen diese einmal auf true und einmal auf false und unterscheiden sich nur in der gerade betrachteten atomaren Teilbedingung (irrelevant kann mit true u. false gleichgesetzt werden)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen die Gesamtbedingung einmal auf true und einmal auf false

Beispiele für „short circuit“-Evaluierung (von links nach rechts):

- IF <Ausdruck der true liefert> OR b THEN ... :
  - ⇨ die Belegung von b ist irrelevant für das Testen und wird in vielen Programmiersprachen deshalb nicht ausgewertet
- IF <Ausdruck der false liefert> AND b THEN ... :
  - ⇨ die Belegung von b ist irrelevant für das Testen und wird in vielen Programmiersprachen deshalb nicht ausgewertet

Einfaches Beispiel für modifzierten Bedingungsüberdeckungstest:

a	b	a OR b
0	0	0
1	-	1
0	1	1

Anmerkung: der erste Testfall wird für die Bedingung a und b genutzt.

Komplexeres Beispiel für Bedingungsüberdeckungsalternativen:

Art der Überdeckung	a	b	c	(a OR b) AND c
atomar	1	1	0	0
	0	0	1	0
Zweig	1	0	0	0
	1	0	1	1
min. mehrfach	0	0	0	0
	1	1	1	1
modifiziert mehrfach	0	0	-	0
	1	-	1	1
	0	1	1	1
	1	-	0	0

Kontrollflusstest - Pfadüberdeckung (C-„unendlich“-Test):

Jeder mögliche Pfad des Kontrollflussgraphen muss einmal durchlaufen werden.

- rein theoretisches Kriterium, sobald Programm Schleifen enthält (unendliche viele verschiedene Pfade = Programmdurchläufe möglich)
- dient als Vergleichsmaßstab für andere Testverfahren
- findet trotzdem nicht alle Fehler (z.B. Berechnungsfehler), da kein erschöpfender Test aller möglichen Eingabewerte (siehe Abschnitt 4.3)
- davon abgeleitetete in der Praxis durchführbare Verfahren:
  - ⇨ **boundary test**: *alle* Pfade auf denen Schleifen maximal einmal durchlaufen werden (ohne besondere praktische Bedeutung)
  - ⇨ **boundary interior test**: *alle* Pfade auf denen Schleifen maximal zweimal (in direkter Folge) durchlaufen werden (Achtung: Anzahl Pfade explodiert bei geschachtelten Schleifen und vielen bedingten Anweisungen)
  - ⇨ **modifizierter boundary interior test**: bei geschachtelten Schleifen wird beim Durchlauf einer äußeren Schleife die Anzahl der inneren Schleifendurchläufe nicht unterschieden

Abstraktes Beispiel für Boundary-Interior-Test:

```
IF b1 THEN n1 ELSE n2 END;
WHILE b2 DO
   n3;
   WHILE b3 DO
      IF b4 THEN n4 ELSE n5 END;
   END;
END;
```

Durchläufe für Boundary-Interior-Test:

n1	n2	(* kein Durchlauf *)
n1, n3	n2, n3	(* 1x äussere, 0x innere Schleife *)
n1, n3, n4	n2, n3, n4	(* 1x äussere, 1x innere Schleife *)
n1, n3, n5	n2, n3, n5	
n1, n3, n4, n4	n2, n3, n4, n4	(* 1x äussere, 2x innere Schleife *)
n1, n3, n4, n5	n2, n3, n4, n5	
n1, n3, n5, n4	n2, n3, n5, n4	
n1, n3, n5, n5	n2, n3, n5, n5	
n1, n3, n3	n2, n3, n3	(* 2x äussere, 0x u. 0x innere Schleife *)
n1, n3, n4, n3	n2, n3, n4, n3	(* 2x äussere, 1x u. 0x innere Schleife *)
...	...	(* es fehlen noch einige Zeilen *)

Abstraktes Beispiel für Boundary-Interior-Test:

```
IF b1 THEN n1 ELSE n2 END;
WHILE b2 DO
   n3;
   WHILE b3 DO
      IF b4 THEN n4 ELSE n5 END;
   END;
END;
```

Durchläufe für modifizierten Boundary-Interior-Test:

n1	n2	(* kein Durchlauf *)
~~n1, n3~~	~~n2, n3~~	~~(* 1x äussere, 0x innere Schleife; nicht benötigt *)~~
n1, n3, n4	n2, n3, n4	(* 1x äussere, 1x innere Schleife *)
n1, n3, n5	n2, n3, n5	
n1, n3, n4, n4	n2, n3, n4, n4	(* 1x äussere, 2x innere Schleife *)
n1, n3, n4, n5	n2, n3, n4, n5	
n1, n3, n5, n4	n2, n3, n5, n4	
n1, n3, n5, n5	n2, n3, n5, n5	
n1, n3, n3	n2, n3, n3	(* 2x äussere, 0x innere Schleife *)

Bewertung der Kontrollflusstests:

- Anweisungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe C** gefordert (Software, deren Ausfall zu einer bedeutenden, aber nicht kritischen Fehlfunktion führen kann)
- Zweigüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe B** gefordert (Software, deren Ausfall zu schwerer aber noch nicht katastrophaler Systemfehlfunktion führen kann)
- modifizierte Bedingungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe A** gefordert (Software, deren Ausfall zu katastrophaler Systemfehlfunktion führen kann)
- Zweigüberdeckung sollte für uns Mindestanforderung beim Testen darstellen (Kontrollflussfehler/Bedingungsfehler werden relativ gut gefunden, Datenflussfehler natürlich weniger gut)
- Einfache Variante von „modified boundary interior test“ zur Ergänzung: für jede Schleife gibt es Testfälle/Pfade, die sie gar nicht, genau einmal und (mindestens) zweimal ausführen (die Randbedingung „alle Pfade“ wird komplett aufgegeben)

4.5 Datenflussbasierte Testverfahren

Ausgangspunkt ist der Datenflussgraph einer Komponente bzw. der mit Datenflussattributen annotierte Kontrollflussgraph. Bei der Auswahl von Testfällen wird darauf geachtet, dass

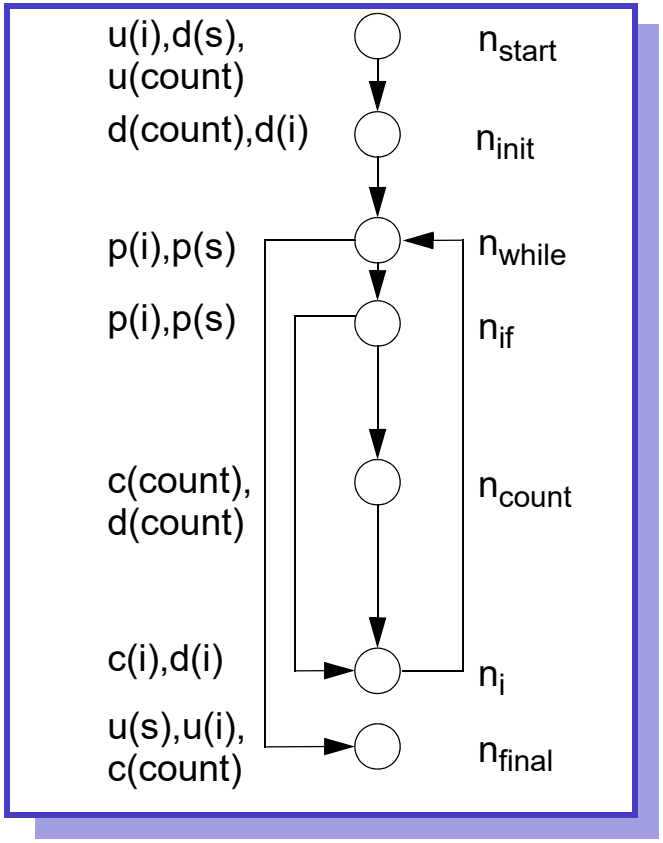
- ⇨ für jede Zuweisung eines Wertes an eine Variable **mindestens eine** (berechnende, prädikative) Benutzung dieses Wertes getestet wird
- ⇨ oder für jede Zuweisung eines Wertes an eine Variable **alle** (berechnenden, prädikativen) Benutzungen dieses Wertes getestet werden

Die datenflussbasierten Testverfahren haben folgende Vor- und Nachteile:

- ☺ einige Verfahren enthalten die Zweigüberdeckung und finden sowohl Datenflussfehler **als auch** Kontrollflussfehler
- ☺ besser geeignet für objektorientierte Programme mit oft einfachem Kontrollfluss aber komplexem Datenfluss
- ☹ es gibt kaum Werkzeuge, die datenflussbasierte Testverfahren unterstützen

Zur Erinnerung - Kontrollflussgraph mit Datenflussattributen:

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN    (* start *)

   count := 0; i := 0;(* init *)

   WHILE s[i] # '.' DO

      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

         count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;(* final *)
```



Kriterien für den Datenflusstest - 1:

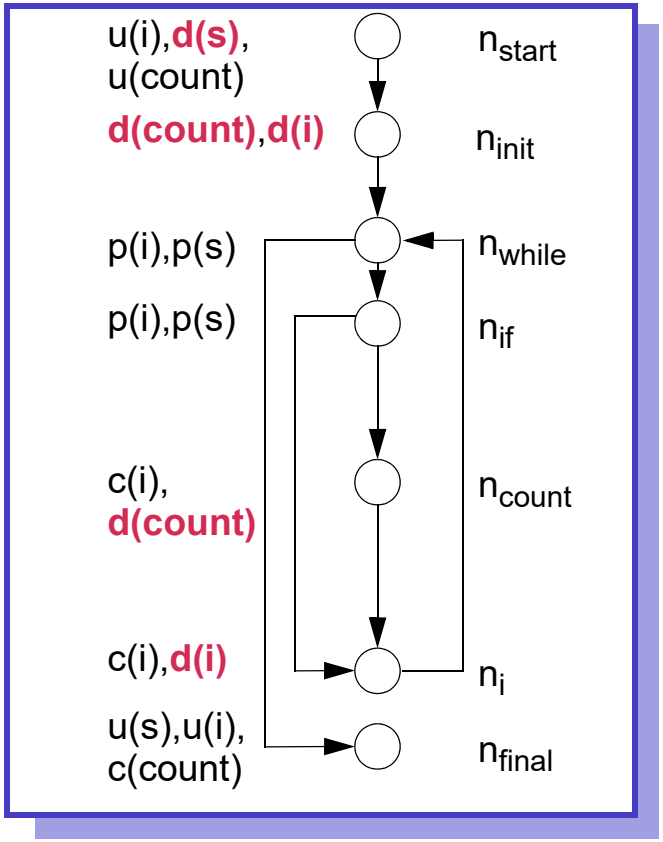
- **all-defs-Kriterium**: für jede Definitionsstelle d(x) einer Variablen muss **ein** definitionsfreier Pfad zu **einer** Benutzung r(x) existieren (und getestet werden)
  - ⇨ Kriterium kann statisch überprüft werden (entdeckt sinnlose Zuweisungen)
  - ⇨ umfasst weder Zweig- noch Anweisungsüberdeckung
  - ⇨ bei countVowels reichen Testbeispiele '.' und 'a.'
  - ⇨ Verfahren findet einige Berechnungsfehler und kaum Kontrollflussfehler
- **all-p-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet
  - ⇨ entdeckt vor allem Kontrollflussfehler
  - ⇨ Berechnungsfehler bleiben oft unentdeckt
  - ⇨ **Anmerkung**: manchmal wird auch Test **aller** definitionsfreien Pfade von d(x) zu allen p(x) gefordert, die Schleifen nicht mehrfach durchlaufen müssen (dann ist Zweigüberdeckung enthalten)

Kriterien für den Datenflusstest - 2:

- **all-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfad zu **allen** (erreichbaren) berechnenden Benutzungen c(x) getestet
  - ⇨ entdeckt vor allem Berechnungsfehler
  - ⇨ Kontrollflussfehler bleiben oft unentdeckt
  - ⇨ lässt sich wegen Bedingungen oft nicht erzwingen
- **all-p-uses-some-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet; gibt es keine prädikate Benutzung p(x), so wird wenigstens **ein** Pfad zu **einem** berechnenden Zugriff c(x) betrachtet
  - ⇨ entdeckt Kontrollfluss- und auch Berechnungsfehler
  - ⇨ umfasst all-def- und all-p-uses-Kriterium
- **all-c-uses-some-p-uses-Kriterium**: … (wird kaum benutzt)
- **all-uses-Kriterium**: all-p-uses- + all-c-uses-Kriterium (wird kaum benutzt)

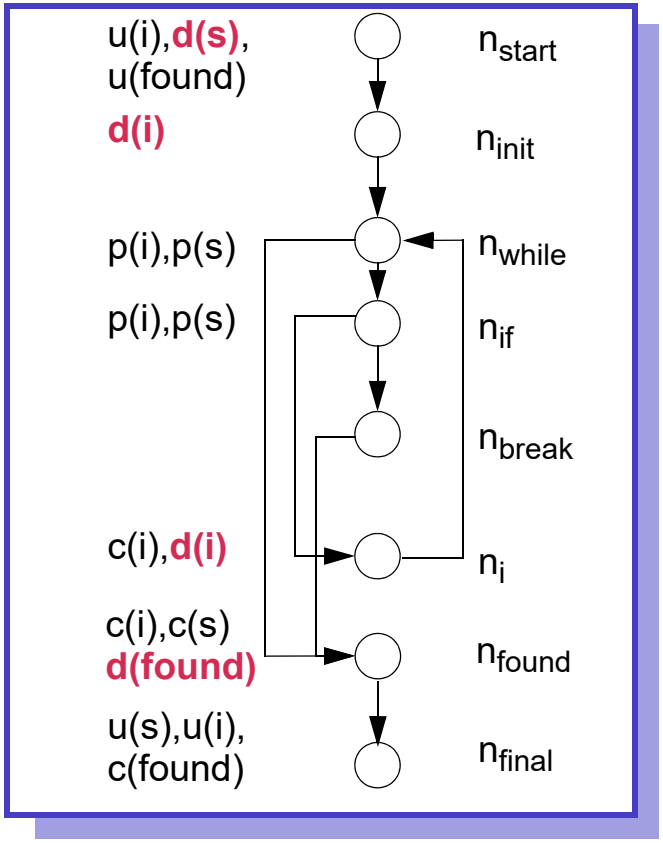
Beispiel für fehlende Zweigüberdeckung von all-defs mit '.' und 'a.':

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
  (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN   (* start *)

  count := 0; i := 0;(* init *)

  WHILE s[i] # '.' DO

    IF s[i]= 'a' OR s[i]= 'e' OR
       s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
    THEN

      count := i+1;(* Datenflussfehler *);

    END;

    i := i+1;

  END (* WHILE *)
END countVowels;(* final *)
```



Beispiel für fehlende Anweisungsüberdeckung von all-defs mit 'b.':

```
PROCEDURE findVowel(IN s: Sentence; OUT found: BOOLEAN);
   (* Searches for first vowel in sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;
BEGIN    (* start *)

   i := 0;  (* init *)

   WHILE s[i] # '.' DO

      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN
         break; (* Schleife wird  verlassen *)
      END;

      i := i+1;

   END (* WHILE *)

   found := (s[i] # '.');

END countVowels;(* final *)
```



All-p-uses-Test am Beispiel:

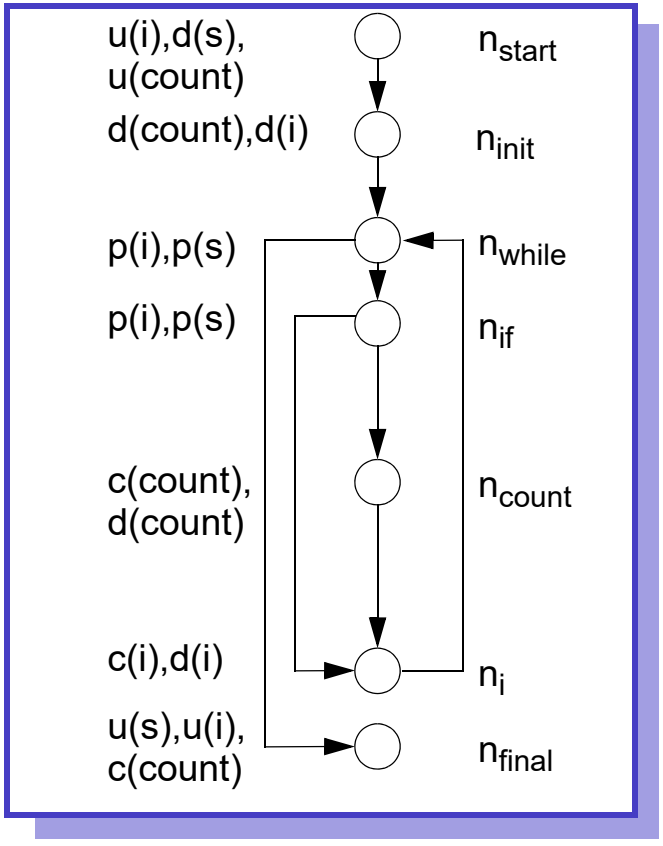
```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
  (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
```

1. für d(s) bei n_{start} : Pfad zu n_{while} und n_{if}
2. für d(i) bei n_{init} : Pfad zu n_{while} und n_{if}
3. für d(i) bei n_i : Pfad zu n_{while} und n_{if}

Es reicht also jede Eingabe mit folgendem Ausführungspfad: n_{start} , n_{init} , n_{while} , n_{if} , n_i , n_{while} , n_{if} , ... , n_{while} , n_{final}

Minimaler Testfall:

'bb.'



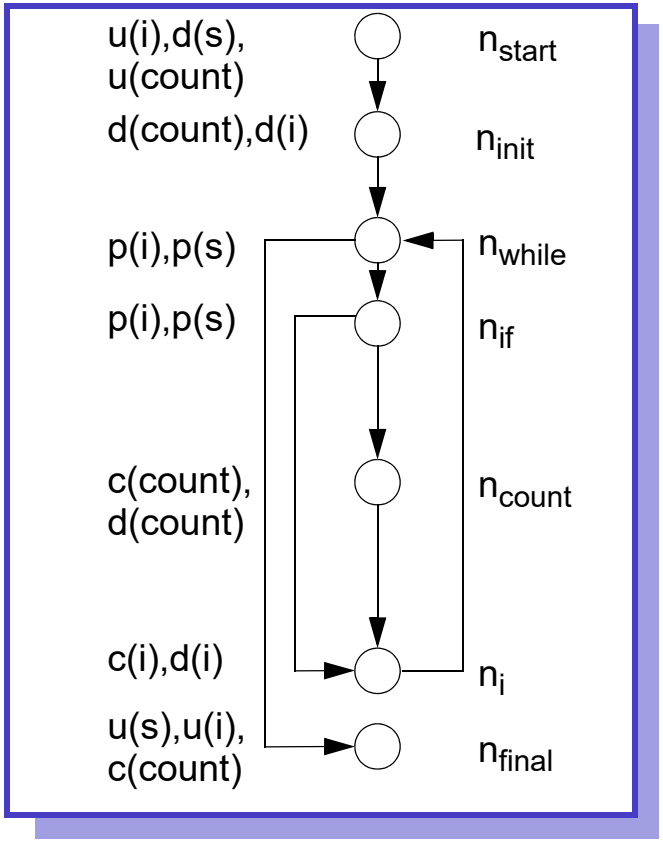
All-c-uses-Test am Beispiel:

PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

1. für d(i) bei n_{init} : Pfad zu n_i
2. für d(count) bei n_{init} : Pfad zu n_{count} und definitionsfreier, direkter Pfad zu n_{final}
3. für d(count) bei n_{count} : Pfad zu n_{count} und definitionsfreier, direkter Pfad zu n_{final}
4. für d(i) bei n_i : Pfad zu n_i

Minimale Anzahl von Testfällen:

1. 'a.'
2. '.'
3. 'aa.'
4. 'a.'



All-p-uses-some-c-uses-Test am Beispiel:

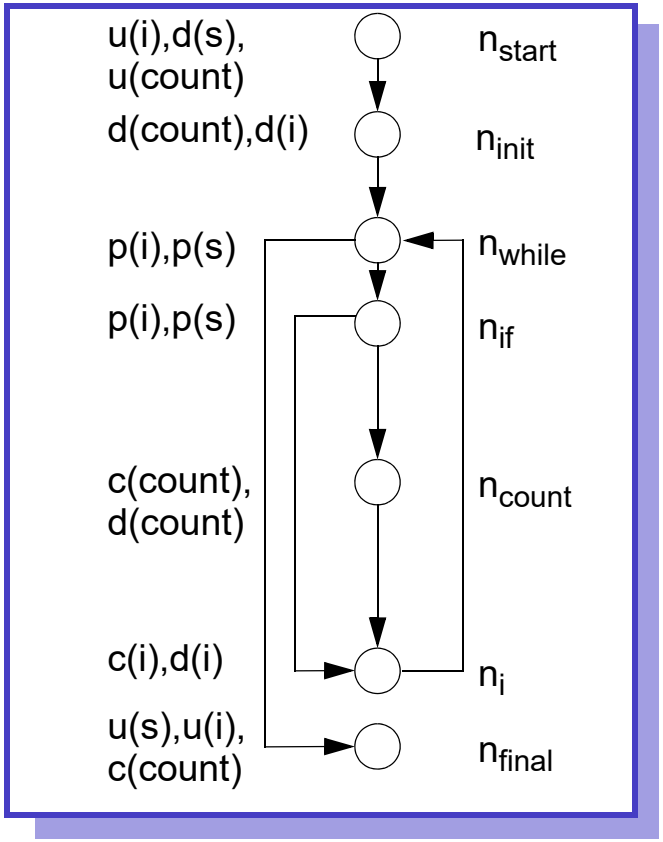
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

1. für d(s) bei n_{start} : Pfad zu n_{while} und n_{if}
2. für d(i) bei n_{init} : Pfad zu n_{while} und n_{if}
3. für d(count) bei n_{init} : Pfad zu n_{count} oder definitionsfreier, direkter Pfad zu n_{final}
4. für d(count) bei n_{count} : Pfad zu n_{count} oder definitionsfreier, direkter Pfad zu n_{final}
5. für d(i) bei n_i : Pfad zu n_{while} und n_{if}

Es reicht also jede Eingabe mit folgendem Ausführungspfad: n_{start} , n_{init} , n_{while} , n_{if} , ... , n_{count} , n_i , n_{while} , n_{if} , ... , n_{while} , n_{final}

Minimaler Testfall:

'ab.'



Zusammenfassung des überdeckungsbasierten Testens:

- man wählt ein oder mehrere Überdeckungskriterien aus, die der „Kritikalität“ der zu entwickelnden Software gerecht werden
- Kombination von einem kontrollflussbasierten und einem datenflussbasierten Überdeckungskriterium sinnvoll
- man wählt nach beliebiger Methodik initiale Menge von Testfällen aus (siehe etwa Abschnitt 4.3 über funktionsorientierte Testverfahren)
- dann wird (durch Code-Überdeckungsanalyse-Werkzeug) überprüft, zu wieviel Prozent die gewählten Kriterien erfüllt sind
  - ⇨ Prozentsatz ausgeführter Anweisungen beim C_0-Test
  - ⇨ ...
- es werden solange Testfälle hinzugefügt, bis eine vorab festgelegte Prozentzahl für alle gewählten Überdeckungskriterien erfüllt ist (90% oder ...)
- Achtung: 100% lässt sich in vielen Fällen nicht erreichen (wegen Anomalien)

4.6 Testen objektorientierter Programme (zustandsbez. Testen)

Prinzipiell lassen sich in objektorientierten Sprachen geschriebene Programme wie alle anderen Programme testen. Allerdings gibt es einige Besonderheiten, die das Testen sowohl erschweren als auch erleichtern (können):

- ☺ die Datenkapselung konzentriert Zugriffsoperationen auf Daten an einer Stelle und erleichtert damit das Testen (Einbau von Konsistenzprüfungen)
- ☹ die Datenkapselung erschwert das Schreiben von Testtreibern, die auf interne Zustände von Objekten zugreifen müssen
- ☺ die Vererbung mit Wiederverwendung bereits getesteten Codes reduziert die Menge an neu geschriebenem zu testenden Code
- ☹ die Vererbung ist eine der Hauptfehlerquellen (falsche Interaktion mit geerbten Code bzw. falsche Redefinition von geerbten Methoden)
- ☹ dynamisches Binden erschwert die Definition sinnvoller Überdeckungsmetriken ungemein (beim White-Box-Test)
- ☹ Verhalten von Objektmethoden ist (fast) immer zustandsabhängig

Prinzipien des Tests objektorientierter Programme:

- beim **„Black-Box“-Test** wie bisher vorgehen (Objekte als Eingabeparameter werden gemäß interner Zustände verschiedenen Äquivalenzklassen zugeordnet)
- beim **„White-Box“-Test** werden Kontrollflussgraphen erweitert, um so Effekte des dynamischen Bindens mit zu berücksichtigen (wird hier nicht weiter verfolgt)
- **Zustandsautomaten** werden zusätzlich zu Kontrollflussgraphen zur Testplanung herangezogen (dieses Verfahren wird meist dem „Black-Box“-Test zugeordnet)
- Einbau von **Konsistenzüberprüfungen** (Plausibilitätsüberprüfungen, assert in Java), in Methoden (zu Beginn und nach Abarbeitung von Methodencode)
- [**defensive Programmierung**: Code fängt alle erkennbaren Inkonsistenzen ab]
- **geerbter Code** wird wie neu geschriebener Code behandelt und immer vollständig im Kontext der erbenden Klasse neu getestet (Variation des Regressionstests)
- inkrementelles Testen mit **Regressionstests** unter Verwendung von Frameworks wie JUnit (http://www.junit.org); siehe Softwarepraktikum

Prinzipien beim Test einzelner Klassen - 1:

Es werden in [Bi00] vier verschiedene Arten von Klassen unterschieden:

1. **Nicht-modale Klassen**: Methoden der Klasse können immer (zu beliebigen Zeitpunkten) aufgerufen werden; interner Zustand der Objekte spielt dabei keine Rolle
   - ⇨ Methoden können isoliert für sich getestet werden; bei der Auswahl der Testfälle muss Objektzustand nicht mit berücksichtigt werden
   - ⇨ Beispiel: Gerätesteuerung mit setStatus-Methode u. getStatus-Methode, die Zustand liefert (Beispiel ist Grenzfall; besser Klasse ohne Attribute)

2. **Uni-modale Klasse**: Methoden können nur - unabhängig vom internen Zustand der Objekte - in einer bestimmten Reihenfolge aufgerufen werden (warum?)
   - ⇨ Testfälle müssen alle zulässigen und nicht zulässigen Reihenfolgen von Methodenaufrufen durchprobieren; interne Objektzustände nicht relevant (Automaten mit zulässigen Methodenaufrufen als Transitionen werden zur Testplanung herangezogen)
   - ⇨ Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methoden; init-Methode muss zuerst aufgerufen werden

Prinzipien beim Test einzelner Klassen - 2:

3. **Quasi-modale Klasse**: Zustand der Objekte bestimmt Zulässigkeit von Methodenaufrufen (und nur dieser)
   - ⇨ Methoden werden isoliert getestet, aber für alle zu unterscheidenden Äquivalenzklassen des internen Objektzustandes (Automaten mit Objektzuständen werden zur Testplanung herangezogen)
   - ⇨ Beispiel: Gerätesteuerung mit setStatus-Methode und getStatus-Methode, die nur 100.000 Status-Wechsel zulässt und dann den Dienst verweigert (nach Wartung verlangt)

4. **Modale Klasse**: Methoden können nur in fest vorgegebenen Reihenfolgen aufgerufen werden; zusätzlich hat Objektzustand Einfluss auf Zulässigkeit von Aufrufen
   - ⇨ Kombination der Testmethoden für uni-modale und quasi-modale Klassen notwendig; Testplanung mit Automaten
   - ⇨ Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methode mit zusätzlicher Beschränkung auf 100.000 Status-Wechsel

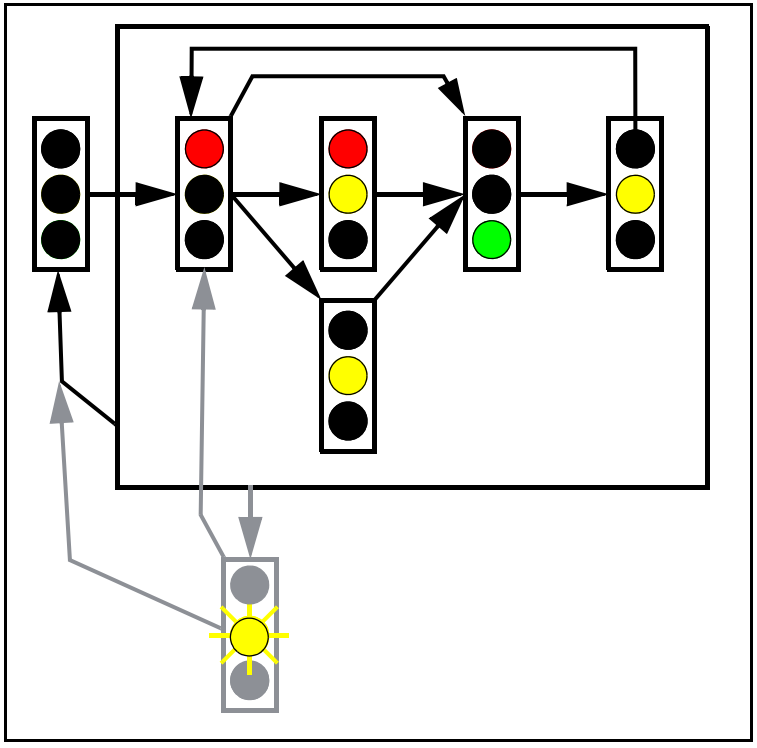
Tabellarische Übersicht über verschiedene Arten von Klassen beim Testen:

	Zustand bestimmt nicht Methodenaufrufbarkeit	**Zustand bestimmt Methodenaufrufbarkeit**
Aufrufreihenfolge der Methoden flexibel	**nicht modal**: Methoden isoliert testen	**quasi-modal**: Methoden isoliert testen für alle Zustandsäquivalenz-klassen
Aufrufreihenfolge der Methoden fest	**uni-modal**: alle zulässigen und verbote-nen Reihenfolgen testen	**modal**: alle zulässigen/verbotenen Reihenfolgen testen für alle Zustandsäquivalenzklassen

Beispiel Ampelsteuerung - modale Klasse:

Eine Klasse Ampel (TrafficLight) wird im folgenden als Beispiel verwendet. Sie bietet Grundfunktionen für die Realisierung von Ampelsteuerungen an und „verkapselt“ die Gemeinsamkeiten der Ampelsteuerungen verschiedener Länder:

- eine Ampel ist aus oder im Rot-Gelb-Grün-Zyklus oder blinkt (Zusatzfunktion)
- von Rotphase wird entweder über Gelb nach Grün gewechselt oder direkt
- in Gelbphase nach Rot leuchtet entweder nur gelbes Licht oder auch rotes Licht
- eine Ampel kann immer ausgeschaltet werden oder nach Blinken wechseln
- vom Zustand Blinken wechselt eine Ampel immer nach Rot
- eine defekte Ampel (ein Licht geht nicht) schaltet sich automatisch ab



Prinzipien bei der OO-Implementierung der Ampelsteuerung:

- folgende Arten von Methoden werden unterschieden (und später beim Testen unterschiedlich behandelt):
  - ⇨ **Konstruktoren** und **Destruktoren**, die Objekt erzeugen bzw. löschen
  - ⇨ beobachtende Methoden (**Observer**), die Objektzustand nicht ändern
  - ⇨ verändernde Methoden (**Modifier**)
- es werden sogenannte **Invarianten** (Invariants) aufgeschriebenen, die nach bzw. vor Ausführung aller Methoden immer gelten müssen:
  - ⇨ Invarianten können durch Observer nicht zerstört werden
  - ⇨ Invarianten müssen am Ende des Codes eines Konstruktors gelten
  - ⇨ Invarianten gelten vor Aufruf eines Modifiers und müssen am Ende des Codes eines Modifiers wieder gelten
- für Observer, Modifier und Destruktoren können **Vorbedingungen** (Preconditions) angegeben werden, die bei Aufruf (zusätzlich zu Invarianten) gelten
- für Konstruktoren, Modifier (und Observer) können **Nachbedingungen** (Postconditions) angegeben werden, die nach Ausführung zusätzlich zu Invarianten gelten

Implementierung der Ampelsteuerung in C++:

```
class TrafficLight {

public:
// invariant: (isOff() || isGreen() || isYellow() || isRed());
// invariant: (! ( isOff() && (isGreen() || isYellow() || isRed()) ) );

// Konstruktoren und Destruktoren
   TrafficLight();
   ~TrafficLight();

// öffentliche beobachtende Operationen (Observer)
   bool isGreen() const;
   bool isYellow() const;
   bool isRed() const;
   bool isOff() const;

// öffentliche verändernde Operationen (Modifier)
   virtual void lightOn();
      // precondition: isOff();
      // postcondition: isRed();
   virtual void lightOff();
      // precondition: (isGreen() || isYellow() || isRed());
      // postcondition: isOff();
```

Implementierung der Ampelsteuerung - Fortsetzung:

```
    virtual void offOnFault();
      // precondition: (! isOff());
      // postcondition: ( (greenOk() && yellowOk() && redOk()) || isOff() )
    virtual void greenOn();
      // precondition: isYellow() || isRed();
      // postcondition: isGreen();
    virtual void yellowOn();
      // precondition: isGreen() || isRed();
      // postcondition: isYellow();
    virtual void redOn();
      // precondition: isYellow();
      // postcondition: isRed();
private:
    bool off, green, yellow, red;
    // Initialisierung: off = true, green = yellow = red = false;
protected:
    // Überprüfung der Funktionsfähigkeit der drei Lampen
    bool greenOk(); bool yellowOk(); bool redOk();
};
```

Prinzipien bei der Spezialisierung der Ampelsteuerung:

- es dürfen beliebig neue Methoden, Attribute, ... als neue Funktionalität hinzugefügt werden
- es dürfen neue Invarianten hinzugefügt werden bzw. die bestehenden Invarianten der Klasse verschärft werden (geerbte und neue Methoden erhalten mindestens die geerbten Invarianten)
- es dürfen Methoden redefiniert werden, wenn dabei
  - ⇨ Vorbedingungen allenfalls gelockert werden (redefinierte Methode ist mindestens dann aufrufbar, wenn geerbte Vorbedingungen erfüllt sind)
  - ⇨ Nachbedingungen allenfalls verschärft werden (redefinierte Methode erfüllt mindestens die geerbten Nachbedingungen)

Achtung:

Werden Invarianten verschärft (hinzugefügt), so müssen auch die unverändert gebliebenen geerbten Methoden diese neuen Invarianten erfüllen!

Implementierung der Spezialisierung in C++:

```
class FlashingTrafficLight : public TrafficLight {
   // bietet neue Operation „blinkendes gelbes Licht“ an
   // verbietet direkten Übergang von rotem Licht zu grünem Licht
   // bei der Gelbphase nach der Rotphase ist auch das rote Licht an
public:
// erlaubte Verschärfung der geerbten Invarianten:
// invariant: (! ( (isOff() || isFlashing()) && (isGreen() || isYellow() || isRed()) ) );
// invariant: (! isGreen() && (isRed() || isYellow()) );
// invariant: (! (isOff() && isFlashing());
// verbotene Lockerung der geerbten Invarianten:
// invariant: (isOff() || isFlashing() || isGreen() || isYellow() || isRed())
// neue Observer-Operation
   bool isFlashing();
// neue Modifier-Operation
   virtual void flashingOn();
      // precondition: (isRed() || isYellow() || isGreen());
      // postcondition: isFlashing();
```

Erlaubte Verschärfung geänderter Invarianten:

```
class FlashingTrafficLight : public TrafficLight {
   // bietet neue Operation „blinkendes gelbes Licht“ an
   // verbietet direkten Übergang von rotem Licht zu grünem Licht
   // bei der Gelbphase nach der Rotphase ist auch das rote Licht an
public:
// invariant: (isOff() || isOn());  geänderte geerbte Bedingung v. TrafficLight
// invariant: !(isGreen() || isYellow() || isRed())  || isOn();  neue geerbte Bedingung
// ...
// erlaubte Verschärfung der geerbten Invarianten:
// invariant: !isFlashing() || isOn();   hinzugefügte Bedingung in FlashingTrafficLight
// neue Observer-Operation
   bool isFlashing();
// neue Modifier-Operation
   virtual void flashingOn();
      // precondition: (isRed() || isYellow() || isGreen());
      // postcondition: isFlashing();
```

Implementierung der Spezialisierung - Fortsetzung:

```
// Redefinitionen
   virtual void greenOn();
      // precondition: isYellow();  verbotene Verschärfung von (isYellow() || isRed())
      // postcondition: isGreen();
   virtual void yellowOn();
      // precondition: isGreen() || isRed();
      // postcondition: isYellow() && isRed(); erlaubte Verschärfung der Nachbedingung
   virtual void redOn();
      // precondition: isYellow() || isFlashing(); erlaubte Lockerung der Vorbedingung
      // postcondition: isRed();
private:
   bool flashing;
};
```

Anmerkung:

Die Einschränkung, dass die „blinkende Ampel“ nur noch in „grün“ schalten darf, wenn sie im Zustand „gelb“ ist, führt dazu, dass sich die blinkende Ampel nicht mehr wie eine „normale“ Ampel der Klasse TrafficLight verhält.

Vorüberlegungen zur Realisierung von Invarianten, ... :

- die teuere Überprüfung von Invarianten, ... muss durch „Compile“-Flag abschaltbar sein (entweder systemweit oder je Subsystem)
- die Überprüfungen dürfen das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht verändern
- bei abgeschalteten Überprüfungen sollte der Code für diese Überprüfungen nicht Bestandteil des ausführbaren Programms sein
- die Reaktion auf fehlgeschlagene Überprüfungen muss an einer Stelle (veränderbar) festgelegt werden (Programmabbruch, Ausnahmeerweckung, ...)
- die Überprüfungen sollten möglichst lesbar niedergeschrieben werden
- Invarianten werden auch vor der Ausführung einer Methode und nach der Ausführung von „Observer“-Methoden überprüft (um illegale Objektzugriffe entdecken zu können)

Zusicherungen (Assertions) in Java:

- Java besitzt ab Version 1.4 „assert“-Statement, das für den Einbau von Überprüfungen (Vor-/Nachbedingungen, Invarianten) genutzt wird
- die Überprüfung von „assert“-Statements kann durch Runtime-Flags generell oder klassenweise an- bzw. abgeschaltet werden („-enableassertions“ = „-ea“ und „.-disableassertions“ = „-da“)
- die Überprüfungen sollten das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht verändern (der Programmierer muss das sicherstellen)
- soll der Code für „assert“-Statements nicht Bestandteil des ausführbaren Programms sein, so muss der Compiler davon „überzeugt“ werden, dass der Code wegoptimiert werden kann:
  ```
  static final boolean assertChecksOn = ... ; // false to eliminate asserts
  if (assertChecksOn) assert ... ;
  ```
- „assert“-Verletzungen können als „AssertionError“-Ausnahmen abgefangen werden (und damit im Code festgelegte Reaktionen auslösen)

Makros für die Überprüfung von Invarianten, ... in C++:

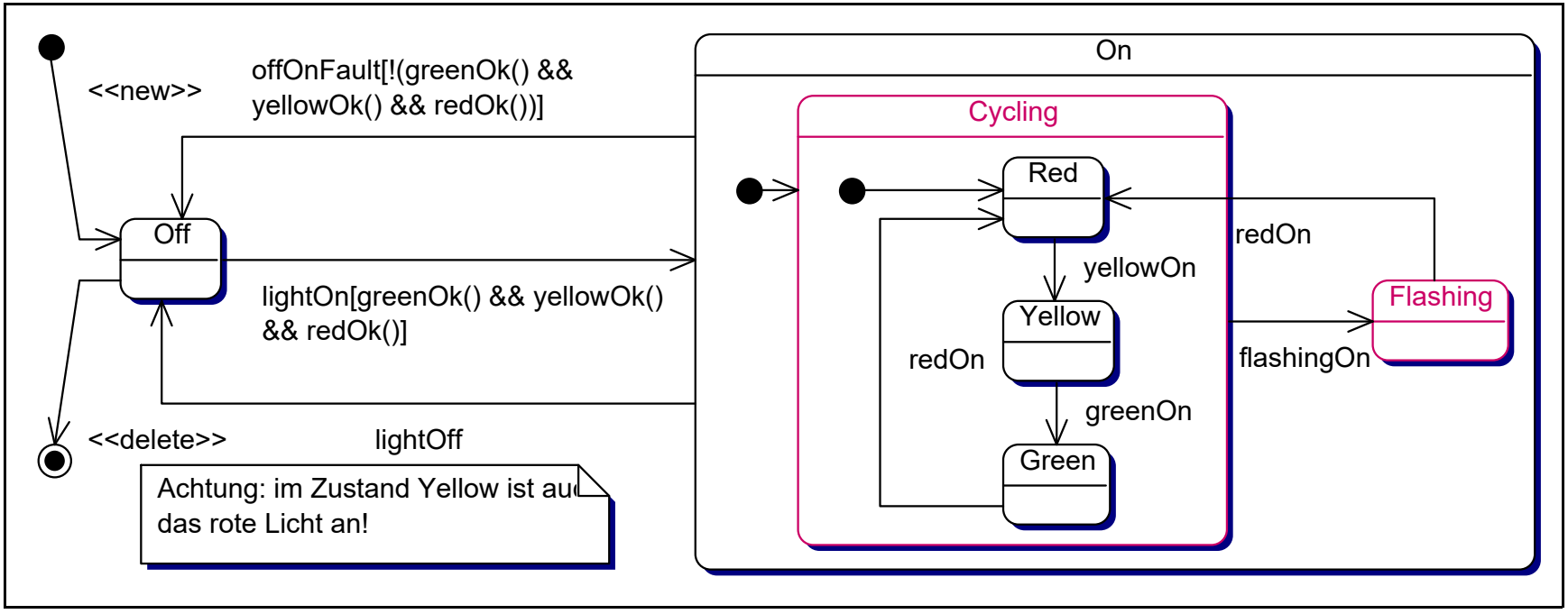
```
# ifdef DEBUG
      # define ASSERT(condition) {\
            if (!(condition)) {\
            < raise exception or terminate program or print error message or ... >\
            }  \
      }
      // nur im Debug-Modus werden Zusicherungen überprüft
# else
   # define ASSERT(condition)
# endif

# define INVARIANT  ASSERT( invariant() )
// im Debug-Modus wird als „protected“ Methode jeder Klasse codierte Invariante überprüft

# define POST(condition) {\
      ASSERT(condition); INVARIANT; \
   }
// zusätzlich zu Nachbedingung wird Einhaltung der Invariante sichergestellt

# define PRE(condition) {\
      INVARIANT; ASSERT(condition); \
   }
```

Überprüfung von Invarianten, ... in C++:

```
class TrafficLight {
public:
// Konstruktoren und Destruktoren
   TrafficLight() {
      ... ; INVARIANT;
   };
   ~TrafficLight() {
      INVARIANT; ...
   };
// öffentliche verändernde Operationen (Modifier)
   void lightOn() {
      PRE(isOff());
      ... ;
      POST(isRed());
   };
 ...
protected:
   inline bool invariant() const { return ( ... ); };
   // Übersetzer ruft ggf. inline-Methode nicht auf, sondern kopiert Rumpf an Aufrufstelle ein
};
```

Einsatz von Automaten (Statecharts) zur Testplanung:

Oft lassen sich die Vorbedingungen für den Aufruf von Methoden besser durch ein Statechart (hierarchischer Automat, siehe Software Eng. - Einführung) darstellen:



Ein solches Statechart kann dann - ähnlich wie ein Kontrollflussgraph - zur Planung von Testfällen zur Berechnung von Testüberdeckungsmetriken herangezogen werden.

Präzisierung der vier verschiedenen Arten von Klassen:

- Nicht modale Klasse:
  - ⇨ keine Methode der Klasse hat eine Vorbedingung über Attributen
  - ⇨ der Zustandsautomat der Klasse besteht aus einem Zustand
- Quasimodale Klasse:
  - ⇨ mindestens eine Methode hat eine Vorbedingung über Attributen
  - ⇨ der Zustandsautomat der Klasse besteht aus einem Zustand
- Unimodale Klasse:
  - ⇨ keine Methode der Klasse hat eine Vorbedingung über Attributen
  - ⇨ der Zustandsautomat der Klasse besteht aus mehreren Zuständen
- Modale Klasse:
  - ⇨ mindestens eine Methode hat eine Vorbedingung über Attributen
  - ⇨ der Zustandsautomat der Klasse besteht aus mehreren Zuständen

Definition von Testüberdeckungsmetriken für Statecharts:

- Tests müssen garantieren, dass alle Zustände mindestens einmal erreicht werden (entspricht Anweisungsüberdeckung aus Abschnitt 4.4)
- Tests müssen garantieren, dass jede Transition mindestens einmal ausgeführt wird (entspricht Zweigüberdeckung aus Abschnitt 4.4), nachdem hierarchisches Statechart in flachen Automaten übersetzt wurde (siehe SW Eng. - Einführung)
- Tests müssen garantieren, dass jede Transition mit allen sich wesentlich unterscheidenden Belegungen ihrer Bedingung ausgeführt wird (entspricht modifiziertem Bedingungsüberdeckungstest aus Abschnitt 4.4)
- Tests müssen alle möglichen Pfade durch Statechart (bis zu einer vorgegebenen Länge oder vorgegebenen Anzahl von Zyklen) ausführen (entspricht eingeschränkten Varianten der Pfadüberdeckung aus Abschnitt 4.4)
- zusätzlich zum Test aller explizit aufgeführten Transitionen werden für jeden Zustand alle sonst möglichen Ereignisse (Methodenaufrufe) ausgeführt (gewünschte Reaktion: Ignorieren oder Programmabbruch oder ...)

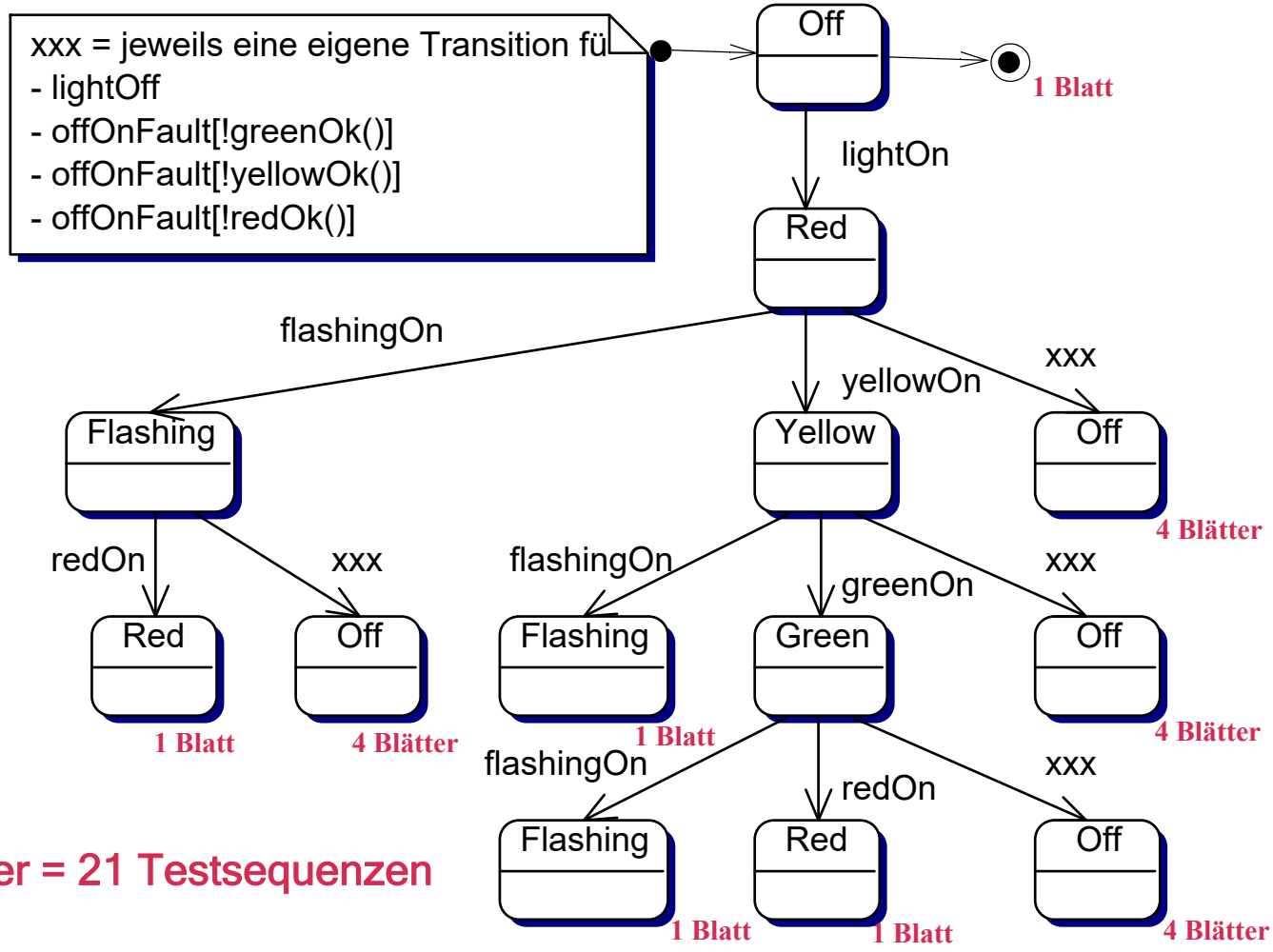
Testplanung mit Transitionsbaum (Übergangsbaum) aus [Bi00]:

1. das gegebene Statechart wird in einen flachen Automaten übersetzt
2. Transitionen mit komplexen Boole'schen Bedingungen werden in mehrere Transitionen mit Konjunktion atomarer Bedingungen übersetzt
   (Transition mit [(a1 && a2) || (b1 && b2)] wird ersetzt durch Transition mit [a1 && a2] und Transition mit [b1 && b2]
3. ein Baum wird erzeugt, der
   - ⇨ initialen Zustand als Wurzelknoten (ersten, obersten Knoten) besitzt
   - ⇨ Zustandsknoten im Baum werden expandiert, indem alle Transitionen zu anderen Zuständen (und sich selbst) als Kindknoten hinzugefügt werden
   - ⇨ jeder Zustand wird nur einmal als Knoten im Transitionsbaum expandiert
4. jeder Pfad in dem Baum (von Wurzel zu einem Blatt) entspricht einer Testsequenz
5. zusätzlich werden in jedem Zustand alle Ereignisse ausgelöst, die nicht im Transitionsbaum aufgeführt sind (spezifikationsverletzende Transitionen)

„Flachgeklopfter“ Automat mit einfacheren Transitionsbedingungen:

⇨

Ein möglicher Transitionsbaum für Ampelsteuerung:



21 Blätter = 21 Testsequenzen

Die normalen Testsequenzen:

1. new (Konstruktoraufruf), delete (Destruktoraufruf)
2. new, lightOn[greenOk() & yellowOk() & redOk()], flashingOn, redOn
3. new, lightOn[...], flashingOn, lightOff
4. new, lightOn[...], flashingOn, offOnFault[!greenOk()]
5. new, lightOn[...], flashingOn, offOnFault[!yellowOk()]
6. new, lightOn[...], flashingOn, offOnFault[!redOk()]
7. new, lightOn[...], yellowOn, flashingOn
8. new, lightOn[...], yellowOn, greenOn, flashingOn
9. new, lightOn[...], yellowOn, greenOn, redOn
10. new, lightOn[...], yellowOn, greenOn, lightOff
11. new, lightOn[...], yellowOn, greenOn, offOnFault[!greenOk()]
12. new, lightOn[...], yellowOn, greenOn, offOnFault[!yellowOk()]
13. new, lightOn[...], yellowOn, greenOn, offOnFault[!redOk()]
14. new, lightOn[...], yellowOn, lightOff
15. ...

Die Tests für irrelevante/irreguläre Transitionen:

1. new, lightOff — Test für den Zustand Off
2. new, lightOn[!greenOk()]
3. new, lightOn[!yellowOk()]
4. new, lightOn[!redOk()]
5. new, offOnFault[...]
6. new, redOn
7. new, yellowOn
8. new, greenOn
9. new, flashingOn
10. new, lightOn, FlashingOn, lightOn — Test für den Zustand Flashing
11. ...

Behandlung ereignisloser Transitionen:

Automatenmodelle für die Verhaltensbeschreibung wie UML-Statecharts erlauben oft auch die Definition ereignisloser Transitionen. Diese werden wie folgt beim Testen behandelt:

- Transition **ohne Ereignis mit Bedingung** [b]: sobald die Bedingung erfüllt ist, schaltet die Transition; beim Aufstellen von Testsequenzen sind zwei Fälle zu unterscheiden (zwei Unterbäume im Transitionsbaum):
  - ⇨ Bedingung b ist bereits erfüllt, wenn Startzustand der Transition betreten wird (Transition wird mit der Bedingung „[b == true]“ markiert und schaltet sofort bei der Testausführung)
  - ⇨ Bedingung ist nicht erfüllt, wenn Startzustand betreten wird; wird später aber erfüllt (Transition wird mit Ereignis „b -> true“ markiert und schaltet sobald die Bedingung b erfüllt ist)
- Transition **ohne Ereignis und ohne Bedingung**: die Transition schaltet, sobald der Startzustand betreten wird: der Startzustand erhält im Transitionsbaum eine ausgehende Kante/Transition ohne Markierung

Ausführung der Testsequenzen:

- **Testvorbereitung:** Objekt muss in initialen Zustand (zurück-)versetzt werden
- **Testausführung** einer Sequenz von Methodenaufrufen (Testvektor): es wird unterschieden
  - ⇨ **white-box-Sicht**: es gibt Zugriffsoperationen für Abfrage des internen Zustands; man kann also am Ende einer Testsequenz abfragen, ob richtiger Zustand erreicht wurde (interne Zustände der Implementierung müssen mit „extern“ definierten Zuständen korrespondieren)
  - ⇨ **black-box-Sicht**: Aussenverhalten muss überprüft werden - im Beispiel der Ampel „beobachtbares Verhalten“ der Ampel + ggf. Test, ob bei weiteren Eingaben sich die Steuerung so verhält, wie sie sich im entsprechenden Zustand verhalten müsste
- **Testbeendigung**: ggf. wird Sequenz von Methodenaufrufen so vervollständigt, dass am Ende der Ausführung Objekt sich in einem „terminalen“ Zustand befindet
- **Kombination** von Testsequenzen: um Aufwand für Initialisierung zu reduzieren, werden möglichst lange Testsequenzen generiert bzw. kombiniert

Abschließende Checkliste für Klassentest nach [Bi00]:

- jede Methode (auch die geerbten) wird mindestens einmal ausgeführt
- alle Methodenparameter und alle nach aussen sichtbaren Attribute werden mit geeigneter Äquivalenzklassenbildung durchgetestet
- alle auslösbaren (ausgehenden) Ausnahmen werden mindestens einmal ausgelöst
- alle von gerufenen Methoden auslösbaren (eingehenden) Ausnahmen werden mindestens einmal behandelt (oder durchgereicht)
- alle identifizierten Objektzustände (auch hier Äquivalenzklassenbildung) werden beim Testen erreicht
- jede zustandsabhängige Methode wird in jedem Zustand ausgeführt (auch in den Zuständen, in denen ihr Aufruf nicht zulässig ist)
- alle möglichen Zustandsübergänge (mit allen Kombinationen von Bedingungen an den Übergängen) werden aktiviert
- zusätzlich werden die üblichen Performanz-, Last-, ... -Tests durchgeführt

4.7 Mutationsbasierte Testverfahren

Hypothesen:

- Programmierer erstellen (oft) annährend korrekte Programme (Competent Programmer Hypothesis)
- Komplexe Fehler bedingen häufig die Existenz von simplen Fehlern (Coupling Effect)

Schlussfolgerungen:

- Typische Programmierfehler lassen sich oft auf kleine syntaktische Änderungen (Mutationen) eines korrekten Programmes zurückführen
- Solche Programmmutationen lassen sich automatisiert durchführen
- Testfälle sind „gut“, die bei gleichen Eingaben zu unterschiedlichen Ausgaben (unterschiedlichem Verhalten) eines Programms und eines seiner Mutanten führen

(Sinnloses) Beispielprogramm für mutationsbasiertes Testen:

```
int func_mutant(int x, int y, int z)

  int r;
  if (x < y) {
    r = x+1;
  } else {
    r = y+x;  /* mutierte bzw. fehlerhafte Stelle, korrekte Version:  r = y-x; */
  };

  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };

  return r;
}
```

Anforderungen an (stark-)mutationserkennende Testfälle

Der gesuchte Testfall soll bei seiner Ausführung

- ⇨ die fehlerhafte Stelle erreichen (**Reachability Condition**),
- ⇨ der Programmzustand soll infiziert werden (**Infection Condition**), also während der Ausführung zu veränderten Variablenbelegungen führen
- ⇨ und der infizierte Programmzustand soll in das Ergebnis propagiert werden (**Propagation Condition**), dieses also verändern.

Achtung:

- ⇨ Ist die „Propagation Condition“ erfüllt, dann ist auch die „Infection Condition“ zwangsläufig erfüllt.
- ⇨ Ist die „Infection Condition“ erfüllt, dann ist auch die „Reachability Condition“ zwangsläufig erfüllt.
- ⇨ Bislang für die Auswahl von Testfällen benutzte Programmüberdeckungskriterien konzentrieren sich auf die „Reachability Conditions“

Bewertung von Testfällen für Beispielprogramm:

- func-mutant(0, 1, 0): die mutierte Programmzeile wird nicht erreicht; der Testfall verletzt also die „**Reachability Condition**“!
- func-mutant(0, 0, 0): die mutierte Programmzeile wird erreicht, aber Variable r wird derselbe Wert in richtiger und mutierter Programmversion zugewiesen; der Testfall verletzt die „**Infection Condition**“!
- func-mutant(1, 0, 0): die mutierte Programmzeile wird erreicht, den Variablen r und a werden in der mutierten Programmversion andere Werte zugewiesen, aber der veränderte Wert von a wird nicht weiterverwendet und der Wert von r wird im folgenden wieder überschrieben; der Testfall verletzt die „**Propagation Condition**“!
- func-mutant(1, 1, 1): die mutierte Programmzeile wird erreicht, den Variablen r und a werden in der mutierten Programmversion andere Werte zugewiesen und der daraus berechnete Rückgabewerte unterscheidet sich bei korrektem und mutierten Programm!

Mutationsbasierte Testsuite-Bewertung:

- gegeben ist ein potentiell fehlerhaftes Programm p und eine **Test-Suite** TS, die aus einer Menge von Testfällen {tc1, tc2, ... } besteht
- zunächst wird eine Menge von **Mutanten** M = {m_1, m_2, ... } erzeugt, die sich in der Regel jeweils nur an einer Programmstelle vom Originalprogramm p unterscheiden (Programme m_i mit mehreren Unterschieden gegenüber p werden **Mutanten höherer Ordnung** genannt)
- dann werden alle Testfälle der Test-Suite TS auf dem Programm p und der Menge seiner Mutanten M ausgeführt
- die Test-Suite bzw. ein Testfall tötet einen Mutanten m_i , falls die Ausführung von p und m_i unterschiedliche Ausgaben erzeugen
- **Effektivität** einer Test-Suite: Anzahl der getöteten Mutanten
- **Effizienz** einer Test-Suite: Anzahl der benötigten Testfälle
- Erhöhung der Effizienz einer Test-Suite ohne Reduktion ihrer Effektivität: Testfälle, die keinen Mutanten töten, werden eliminiert; gleiches gilt für Teilmengen von Testfällen, die alle den selben Mutanten töten)

Mutationsbasierte Testsuite-Generierung:

Die werkzeuggestützte Erzeugung von Testfällen, die bestimmte Mutanten töten, ist ein „schwieriges“ Problem (i.A. nicht berechenbar).

- naiver Ansatz: zufallsgesteuerte Erzeugung von Testfällen
- verifikationsbasierter Ansatz: das Problem der Erzeugung eines (fehlenden) Testfalles, der einen bestimmten Mutanten m eines Programms p tötet, wird in ein Verifikationsproblem übersetzt:
  - ⇨ erzeugt wird ein neues Programm, das p und m hintereinander ausführt und die Ausgaben der Ausführung von p und m in verschiedenen Variablen speichert
  - ⇨ verifiziert wird dann die Eigenschaft des neuen Programms, für alle möglichen Eingaben bei der Ausführung von p und m immer die gleichen Ausgaben zu produzieren
  - ⇨ jedes von einem Verifikationswerkzeug produzierte Gegenbeispiel zu dieser Eigenschaft legt einen Testfall fest, der den Mutanten m tötet

Bewertung des Ansatzes [ABL05]:

Is Mutation an Appropriate Tool for Testing Experiments?

J.H. Andrews
Computer Science Department
University of Western Ontario
London, Canada
andrews@csd.uwo.ca

L.C. Briand Y. Labiche
Software Quality Engineering Laboratory
Systems and Computer Engineering Department
Carleton University
Ottawa, Canada
{briand, labiche}@sce.carleton.ca

ABSTRACT

The empirical assessment of test techniques plays an important role in software testing research. One common practice is to

One problem in the design of testing experiments is that real programs of appropriate size with real faults are hard to find, and hard to prepare appropriately (for instance, by preparing correct

- Ähnlichkeit zwischen mechanisch erzeugten Mutanten und realen Fehlern ist größer als zwischen manuell eingebauten Fehlern und realen Fehlern
- bei manuell eingebauten Fehlern wird die Erkennungsrate einer Test-Suite unterschätzt (manuell eingebaute Fehler sind oft schwerer zu detektieren als mechanisch erzeugte Mutationen)
- (ausgewählte) Mutanten sind nicht einfacher oder schwerer zu detektieren als reale Fehler

4.8 Testmanagement und Testwerkzeuge

Testen wird nach [SL19] immer wie folgt durchgeführt:

- **Testplanung**: es werden die zum Einsatz kommenden Methoden und Werkzeuge sowie die zu testenden Objekte in einem Testkonzept festgelegt
- **Testüberwachung und Teststeuerung**: Durchführung, Protokollierung und Bewertung der folgenden Aktivitäten inklusive Entscheidung über Testende
- **Testanalyse**: es wird festgelegt *was* genau zu testen ist durch Überprüfung vorhandener Anforderungspezifikationen, Testobjekte, ...
- **Testentwurf**: es wird festgelegt *wie* getestet wird; dabei werden abstrakte (mit Bedingungen für Eingabewerte) oder konkrete Testfälle (mit konkreten Eingabewerten) spezifiziert
- **Testrealisierung**: es werden die spezifizierten Testfälle implementiert
- **Testdurchführung**: die ausgewählten Testfälle werden ausgeführt
- **Testabschluss**: es werden die Ergebnisse der vorangegangenen Aktivitäten zusammengetragen und konsolidiert

Aufgaben, Qualifikationen und Rollen nach [SL19]:

- **Testmanager** (Leiter): ist für Management der Testaktivitäten, der Testressourcen und die Bewertung des Testobjekts (gegenüber Projektmanager) verantwortlich
- **Testdesigner** (Analyst): erstellt Testspezifikationen und ermittelt Testdaten
  [Ergänzung: könnte beim modellbasierten Testen auch für die Erstellung von Testmodellen und Festlegung von Überdeckungskriterien zuständig sein]
- [**Testfallgenerator**: werden Testfälle aus Modellen bzw. Spezifikationen automatisch erzeugt, so bedarf es einer weiteren Rolle, die die Testfallgenerierung mit Werkzeugunterstützung durchführt]
- **Testautomatisierer**: realisiert die automatisierte Durchführung der spezifizierten Testfälle durch ausgewählte Testwerkzeuge
- **Testadministrator**: stellt die Testumgebung mit ausgewählten Testwerkzeugen zur Verfügung (zusammen Systemadministratoren etc.)
- **Tester**: ist für Testdurchführung, -protokollkierung und -auswertung zuständig (entspricht „Certified Tester Foundation Level“-Kompetenzen)

Weitere Aspekte des Testmanagements nach [SL19]:

- Betrachtung von **Kosten- und Wirtschaftlichkeitsaspekten**
  - ⇨ Ermittlung und Abschätzung von Fehlerkosten
  - ⇨ Ermittlung und Abschätzung von Testkosten / Testaufwand
- Wahl einer **Teststrategie**
  - ⇨ proaktiv vs. reaktiv (Testmanagement startet mit Projektbeginn vs. Testaktivitäten werden erst nach der Erstellung der Software gestartet)
  - ⇨ Testerstellungsansatz / Überdeckungskriterien / ...
  - ⇨ Orientierung an Standards für Vorgehensmodelle (vgl. Kapitel 5)
- Testen und **Risiko**
  - ⇨ Risiken beim Testen (Ausfall von Personal, ...)
  - ⇨ Risikobasiertes Testen (Fokus auf Minimierung von Produktrisiken)

Meldung von Fehlern / Fehlermanagement:

Es müssen alle Informationen erfasst werden, die für das Reproduzieren eines Fehlers notwendig sind. Eine Fehlermeldung kann wie folgt aufgebaut sein:

- **Status**: Bearbeitungsfortschritt der Meldung (Neu, Offen, Analyse, Abgewiesen, Korrektur, Test, Erledigt)
- **Klasse**: Klassifzierung der Schwere des Problems (Menschenleben in Gefahr, Systemabsturz mit Datenverlust, ... , Schönheitsfehler)
- **Priorität**: Festlegung der Dringlichkeit, mit der Fehler behoben werden muss (sofort, da Arbeitsablauf blockiert beim Anwender, ...)
- **Anforderung**: die Stelle(n) in der Anforderungsspezifikation, auf die sich der Fehler bezieht
- **Fehlerquelle**: in welcher Softwareentwicklungsphase wurde der Fehler begangen
- **Fehlerart**: Berechnungsfehler, ...
- **Testfall**: genaue Beschreibung des Testfalls, der Fehler auslöst
- ...

Arten von Testwerkzeugen - 1:

Testwerkzeuge werden oft „**Computer-Aided Software Test**“-Werkzeuge genannt (**CAST-Tools**). Man unterscheidet folgene Arten von CAST-Tools:

- Werkzeuge zum **Testmanagement** und zur Testplanung: Erfassen von Testfällen, Abgleich von Testfällen mit Anforderungen, Verwaltung und (statistische) Auswertung von Fehlermeldungen, ...
- Werkzeuge zur **Testspezifikation**: Testdaten und Soll-Werte für zugehörige Ergebnisse werden (manuell) festgelegt und verwaltet
- **Testdatengeneratoren**: aus Softwaremodellen, Code, Grammatiken, ... werden automatisch Testdaten generiert
- **Testtreiber**: passend zu Schnittstellen von Testobjekten werden Testtreiber bzw. Testrahmen zur Verfügung gestellt, die die Aufruf von Tests mit Übergabe von Eingabewerten, Auswertung der Ergebnisse etc. abwickeln
- **Simulatoren**: bilden möglichst realitätsnah Produktionsumgebung nach (mit Simulation von anderen Systemen, Hardware, ...)

Arten von Testwerkzeugen - 2:

- **Testroboter** (Capture & Replay-Werkzeug): zeichnet interaktive Benutzung einer Bedienungsoberfläche auf und kann Dialog wieder abspielen (solange sich Oberfläche nicht zu stark verändert)
- Werkzeuge für Last- und **Performanztests** (dynamische Analysen): Laufzeitmessungen, Speicherplatzverbrauch, ...
- **Komparatoren**: vergleichen erwartete mit tatsächlichen Ergebnissen, filtern unwesentliche Details, können ggf. auch Zustand von Benutzeroberflächen prüfen
- **Code-Überdeckungsanalysatoren**: für gewählte Überdeckungsmetriken wird Buch darüber geführt, wieviel Prozent Überdeckung erreicht ist, welche Programmausschnitte noch nicht überdeckt sind
- [Werkzeuge für **statische Analysen**: Berechnung von Metriken, Kontroll- und Datenflussanomalien, ...]

4.9 Zusammenfassung

Dynamische Programmanalysen und das „traditionelle“ Testen sind die wichtigsten Mittel der analytischen Qualitätssicherung. Mindestens folgende Maßnahmen sollten immer durchgeführt werden:

- Suche nach Speicherlecks und Zugriff auf nichtinitialisierte Speicherbereiche (oder falsche Speicherbereiche) mit geeigneten Werkzeugen
- automatische Regressions-Testdurchführung mit entsprechenden Frameworks zur Testautomatisierung
- Überprüfung der Zweigüberdeckung durch Testfälle anhand von Kontrollflussgraph (durch Werkzeuge)
- Verwendung von „Black-Box“-Testverfahren mit Äquivalenzklassenbildung für Eingabeparameter (Objekte) zur Bestimmung von Testfällen
- Einbau möglichst vieler abschaltbarer Konsistenzprüfungen in Code (Vor- und Nachbedingungen) und ggf. defensive Programmierung

4.10 Ergänzende Literatur

[ABL05] J. H. Andrews, L.C. Briand, Y. Labiche: Is Mutation an Appropriate Tool for Testing Experiments?, in: Proc. 27th Int. Conf. on Software Engineering (ICSE´05), ACM Press (2005), S. 402 - 411

[Bi00] R. V. Binder: T*esting Object-Oriented Systems*, Addison-Wesley (2000), 1191 Seiten

[BP84] V. R. Basili, B.T. Perricone: *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol. 27, No. 1, 42-52, ACM Press (1984)

[FRSW03] F. Fraikin, E.H. Riedemann, A. Spillner, M. Winter: *Basiswissen Softwaretest - Certified Tester,* Skript zu Vorlesungen an der TU Darmstadt, Uni Dortmund, Bremen und FH Köln (2003)

[FS98] M. Fowler, K. Scott: *UML konzentriert: Die neue Standard-Objektmodellierungssprache anwenden*, Addison Wesley (1998), 188 Seiten

[MS01] J. D. McGregor, D. A. Sykes: *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley (2001)

[SL19] A. Spillner, T. Linz: ***Basiswissen Softwaretest***, dpunkt.verlag (2019; 6. Auflage), 351 Seiten

[Vi05] U. Vigenschow: *Objektorientiertes Testen und Testautomatisierung in der Praxis*, dpunkt.verlag (2005), 331 Seiten