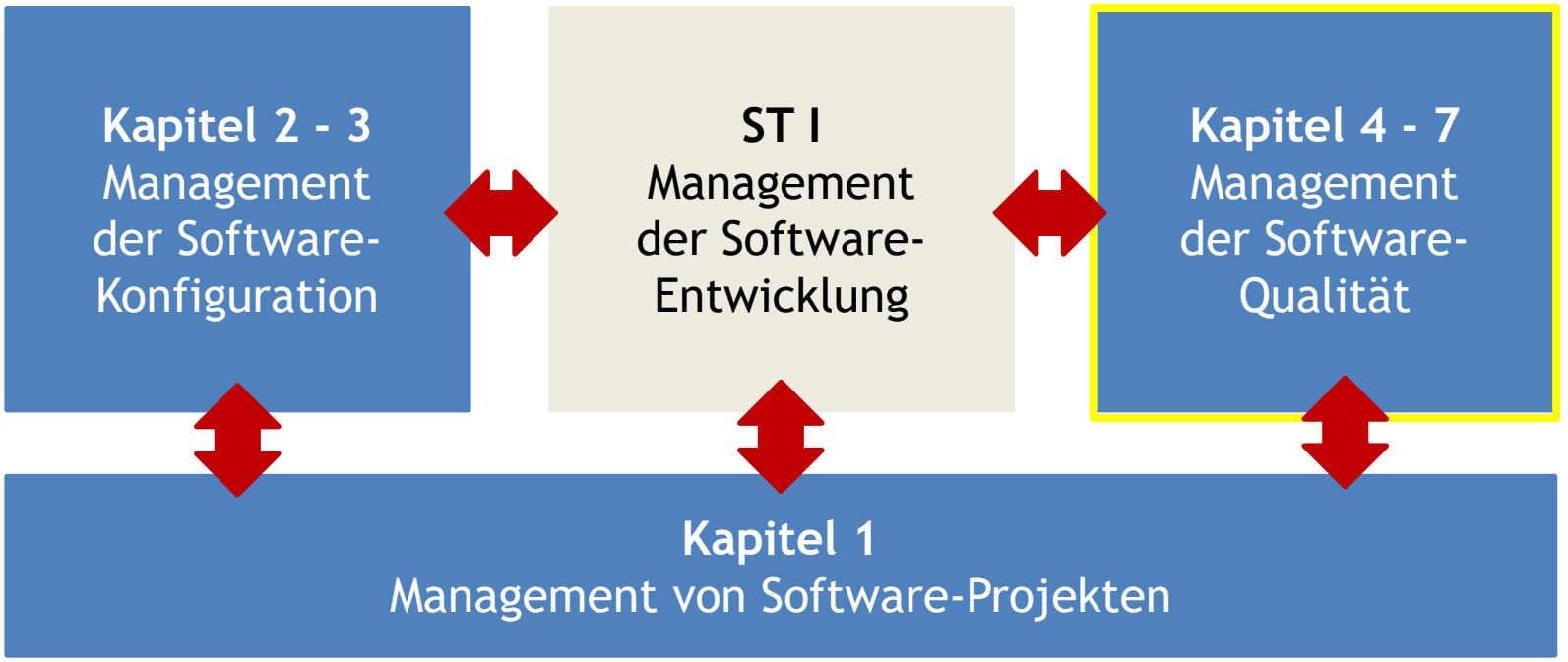
UNIVERSITÄT SIEGEN

# Vorlesung Softwaretechnik II

## Management der Software-Qualität: Dynamische Analysen

# Aufbau der Vorlesung



# Inhalt

- Einführung: Dynamische Programmanalyse
- Funktionsorientierte Testverfahren
- Kontrollflussorientierte Testverfahren
- Datenflussorientierte Testverfahren
- Objektorientierte Testverfahren
- Mutationsbasierte Testverfahren
- Weitere Testverfahren und Testmanagement

UNIVERSITÄT SIEGEN

# Einführung: Dynamische Programmanalyse

## Der Software-Test

# Kent Beck‘s Rule of Thumb

*“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.”*

[FS 1998]

# Zitat

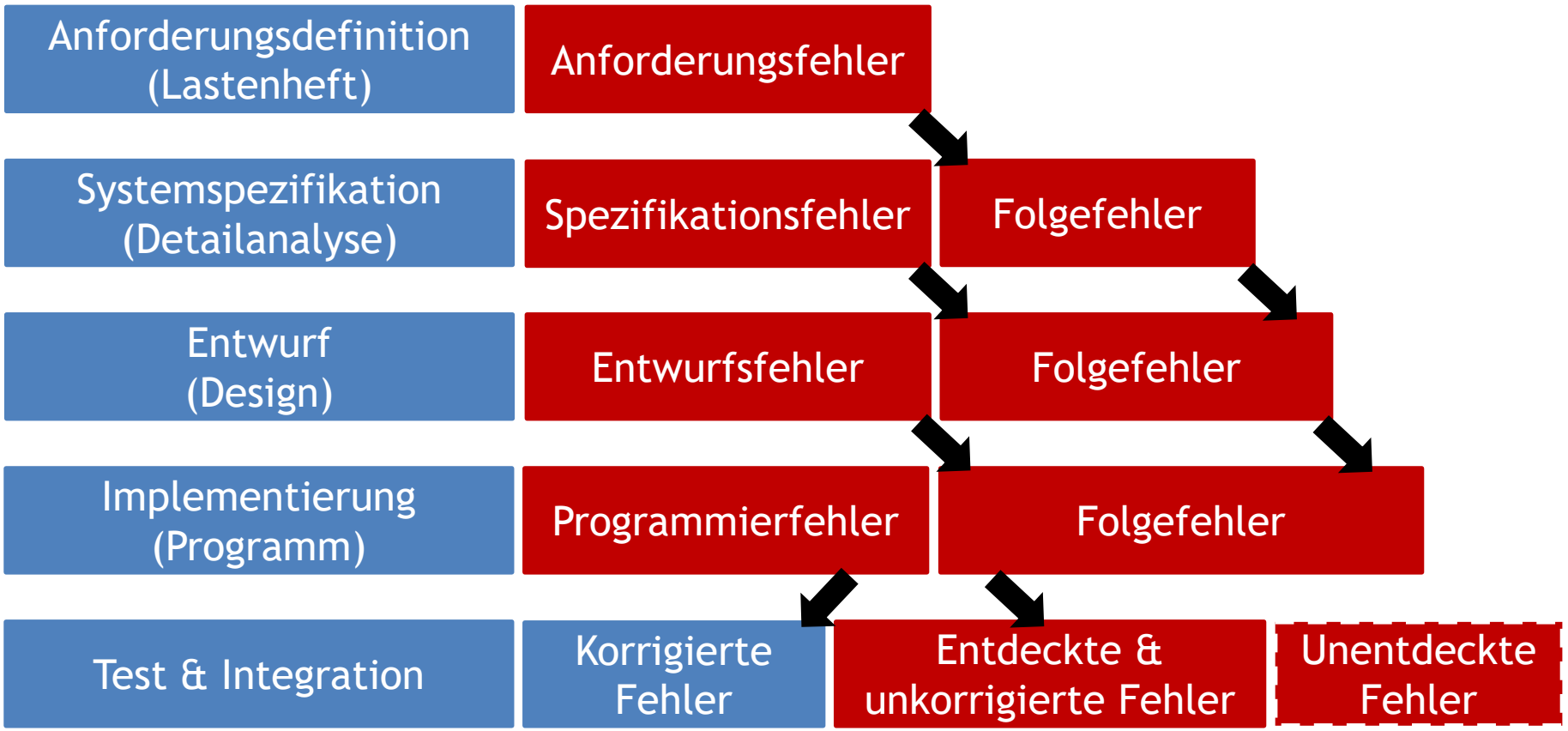
*“It is easier to change the specification to fit the program than vice versa.”*

[Perlisisms - "Epigrams in Programming" by Alan J. Perlis]

## Binsenweisheit

- Ein Programm ist ein rein virtuelles / mathematisches Objekt, das zwar fehlerhaft sein kann, aber für sich genommen keine Gefahr darstellen kann.
- Erst durch die Ausführung eines Programmes auf einer Hardware mitsamt der Möglichkeit, auf die physikalische Umgebung einzuwirken, können fehlerhafte Programme sicherheitsrelevant werden.

# Motivation: Auftreten von Fehlern



# Übungsaufgabe: Fehlerarten

```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0; i = 0;

  while (s[i] != '.') {

    if (s[i] = 'a' || s[i] = 'i'  || s[i] = 'o' || s[i] = 'u') {

        count = count + 2;

    }
    count = i+1;
  }
  return count;
}
```

```
// Aufruf der Methode
…
  String s = "to be … or not to be.";
  int nrVowels =   myObjetc.countVowels(s); // *
…
```

Welche Fehler enthält der Programmausschnitt?

(* Wir nehmen vereinfachend an, dass der String automatisch in ein char-Array konvertiert wurde).

# Auflösung

```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0;

  while (s[i] != '.') {

    if (s[i] = 'a' || s[i] = 'i'  || s[i] = 'o' || s[i] = 'u') {

        count = count + 2;

    }
    count = i+1;
  }
  return count;
}
```

**Initialisieurngsfehler:**
i nicht initialisiert

**Kontrollflussfehler:**
keine Prüfung auf 'e'

**Berechnungsfehler:**
count wird falsch erhöht.

**Datenflussfehler:**
Zuweisung nicht an i.

```
// Aufruf der Methode
…
  String s = "to be … or not to be.";
  int nrVowels = myObjetc.countVowels(s); // *
…
```

**Schnittstellenfehler:**
'.' nur am Satzende erlaubt.

(* Wir nehmen vereinfachend an, dass der String automatisch in ein char-Array konvertiert wurde).

# Diskussion: Fehlerarten

- Durch systematische Software-Tests sollen exemplarische Programmeingaben (im Beispiel: Eingabe-Sätze) selektiert werden, um Fehlerwirkungen möglichst vieler potentieller Fehlerzustände zu provozieren.
- Der Schritt zurück von der beobachteten Fehlerwirkung hin zum Verständnis der den Fehlerzustand auslösenden Fehlhandlung (und dessen Korrektur) ist hingegen nicht Bestandteil des Software-Tests (sondern „Debugging“, …)

# Anmerkung zum Fehler-Begriff

- Im deutschen Sprachgebrauch wird der Begriff „Fehler“ häufig sehr allgemein (bzw. unpräzise) für alles verwendet, was „falsch“ ist.
- Im englischen Sprachgebrauch werden hingegen verschiedene Begriffe verwendet, um unterschiedliche Aspekte hinsichtlich Fehlerursache, Fehlerwirkung und Fehlererkennung zu unterscheiden.

# Definition: Fault (Fehlerzustand)

- Zustand eines Softwareprodukts oder einer seiner Komponenten, der unter spezifischen Bedingungen (z.B. bestimmten Programmeingaben) eine geforderte Funktion beeinträchtigen kann.
- Der Programmtext beinhaltet Stellen mit inkorrekten Anweisungen, Bedingungen, Berechnungen, Datendefinitionen, Funktionsaufrufen, etc.

[DIN 66271]

# Definition: Failure (Fehlerwirkung)

- Abweichung zwischen einem spezifizierten Soll-Wert (z.B. in der Anforderungsdefinition) und einem beobachteten Ist-Wert (z.B. Ausgabewert einer Programmausführung).
- Eine bestimmte Programmausführung führt dazu, dass ein Fehlerzustand erreicht wird, sich danach auf die restliche Programmausführung auswirkt und schließlich auch nach „außen“ sichtbar wird.

# Definition: Error (Fehlhandlung)

- Menschliche Handlung (z.B. des Entwicklers oder Anforderungsmanagers), die zu einem Fehlerzustand in der Software führt.
- Das beinhaltet **nicht** menschliche Handlungen eines **Anwenders** (z.B. fehlerhafte Eingaben), die ein unerwünschtes Ergebnis zur Folge hat („Der Kunde hat immer Recht…“).

# Vom Error über den Fault zum Failure

1. **Error:** Ein Programmierer versteht eine Anforderung falsch / vertippt sich / nutzt ein Programmiersprachen-Konstrukt falsch / verwendet unpassenden Legacy-Code wieder / …
2. **Fault:** Der Error des Programmierers ergibt einen Fehlerzustand (z.B. eine inkorrekte Programmziele), die wegen mangelhafter Qualitätssicherung auch in der ausgelieferten Software landet.
3. **Failure:** Die ausgelieferte Software wird „irgendwann zufällig“ mit bestimmten Eingabedaten ausgeführt, die zu einer Fehlerwirkung führen, die durch den Fehlerzustand verursacht wird.

# Vom Error über den Fault zum Failure?

- Ein **Error** muss nicht zwangsläufig zu einem **Fault** führen: zwei Missverständnisse können sich gegenseitig neutralisieren (z.B. eine vom Programmierer falsch verstandene Anforderungen war von vornherein inkorrekt und wird dadurch unbeabsichtigt korrigiert).
- Ein **Fault** muss nicht zwangsläufig zu einem **Failure** führen: der Fehlerzustand befindet sich in unerreichbaren Programmabschnitten, wird von einem anderen Fehlerzustand maskiert oder der Anwender nutzt die Software niemals auf eine Weise, die die Fehlerwirkung auslösen wird.

# Validation und Verifikation von Software

**Validation:**

- Prüfen, ob die Software das vom Anwender erwartete Verhalten zeigt.
- „Haben wir die richtige Software entwickelt?“

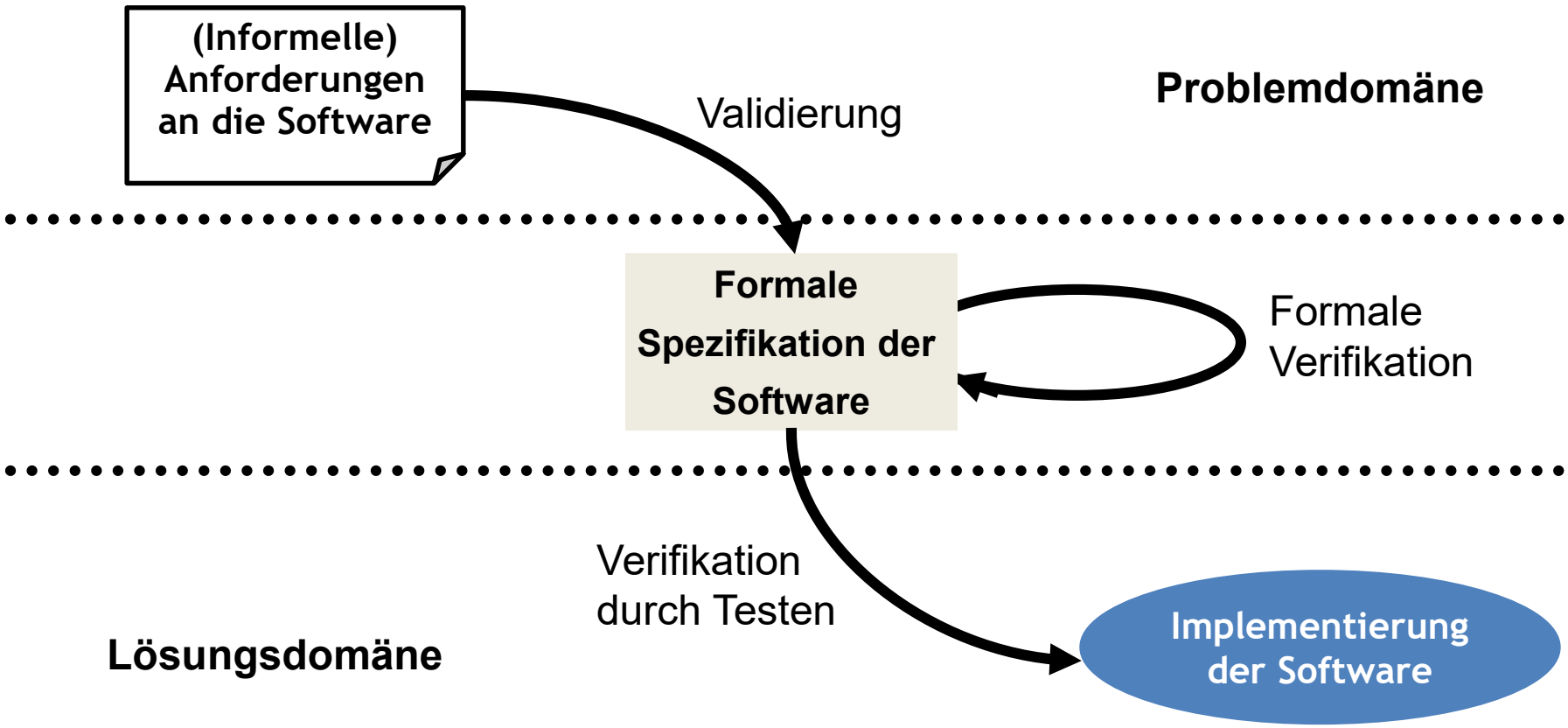
**Verifikation:**

- Prüfen, ob die entwickelte Software die Anforderungen erfüllt, die zuvor vertraglich festgelegt wurden.
- „Haben wir die Software richtig entwickelt?“

# Diskussion: Validation und Verifikation

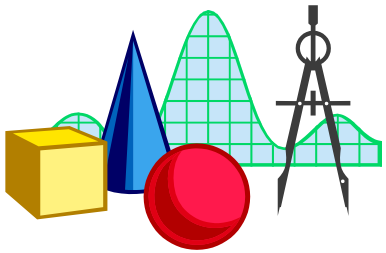
- Bei der Validation geht es um die Frage, ob die Anforderungen an die zu entwickelnde Software korrekt/vollständig erfasst wurden
  (Beispiel Scrum: Geben die Use-Stories die Nutzer-Erwartungen an das System richtig wieder?)
- Bei der Verifikation geht es um die Frage, ob die erfassten Anforderungen richtig in der Implementierung umgesetzt wurden
  (Beispiel Scrum: Wurden die User-Stories richtig implementiert?)

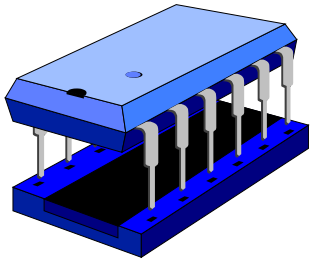
# Validierung, formale Verifikation und Testen



# Formale Verifikation vs. Testen

Abstrakte Spezifikation der Software





Konkrete Implementierung der Software

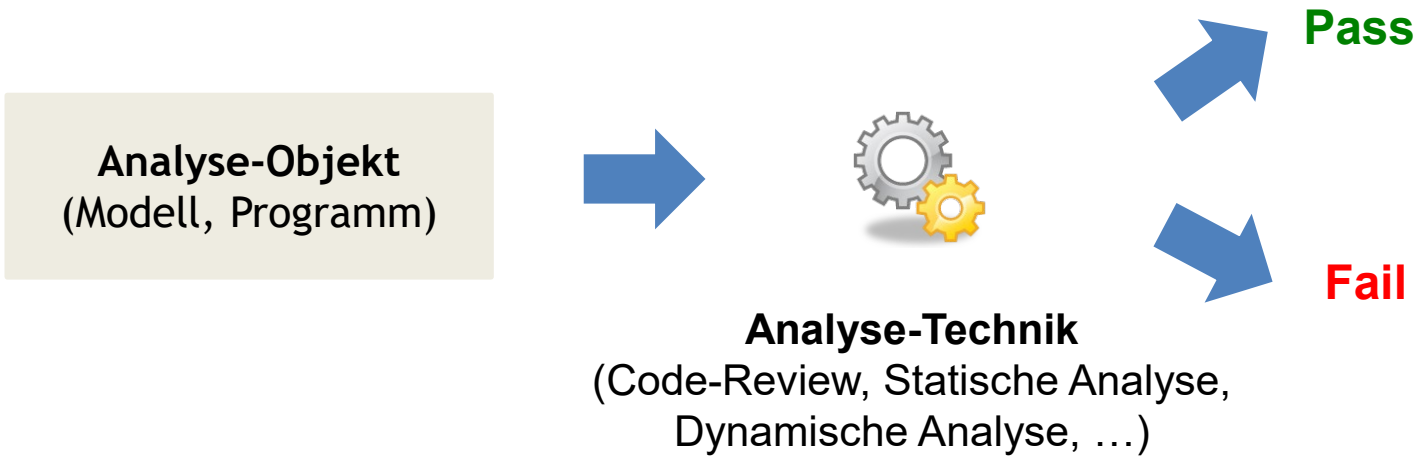
**Formale Verifikation**

- (Theoretisch) vollständiger Nachweis von Korrektheits-eigenschaften.
- Mathematische Manipulation eines abstrakten Modells.
- Aussagekraft abhängig von der Validität des Modells.

**Testen**

- Stichprobenartige (unvollständige) Suche nach Fehlverhalten.
- Experimentelle Manipulation der konkreten Implementierung.
- Ergebnisqualität abhängig von Validität des Modells

# Verlässlichkeit von Analyse-Ergebnissen




Eine Analyse-Technik, angewendet auf ein zu analysierendes Programm/Modell, liefert als „Diagnose“ entweder

- **Pass** (korrekt hinsichtlich der Anforderungen), oder
- **Fail** (inkorrekt hinsichtlich der Anforderungen) und dazu gegebenenfalls weitere Informationen zu „Symptomen“.

UNIVERSITÄT SIEGEN

# Anmerkungen zu Analyse-Ergebnissen

- Die Begriffe „Diagnose“ und „Symptom“ auf der vorherigen Folie wurden bewusst gewählt, da häufig medizinische Analogien verwendet werden.
- Sei A eine Methode, die zur Diagnose einer Krankheit B auf einen Patienten C angewendet wird:
  - Diagnose „gesund“ heißt **true negative**, falls C nicht an B erkrankt ist.
  - Diagnose „krank“ heißt **true positive**, falls C an B erkrankt ist.
  - Diagnose „gesund“ heißt **false negative**, falls C an B erkrankt ist.
  - Diagnose „krank“ heißt **false positive**, falls C nicht an erkrankt ist.
- Diese Begrifflichkeiten lassen sich entsprechend auf Analyse-Techniken für Software übertragen.

# Verlässlichkeit von Analyse-Ergebnissen

| | | | |
|---|---|---|---|
| Analyse-Objekt (fehlerfrei) |  | Pass | True Negative |
| Analyse-Objekt (fehlerhaft) |  | Fail | True Positive |
| Analyse-Objekt (fehlerhaft) |  | Pass | False Negative |
| Analyse-Objekt (fehlerfrei) |  | Fail | False Positive |

# Verlässlichkeit von Analyse-Techniken

Eine Analyse-Technik heißt:

- **konsistent** (engl.: sound), falls keine false positives auftreten können.
- **vollständig** (engl.: complete), falls keine false negatives auftreten können.

Eine Analyse-Theorie heißt:

- **Überapproximation** für eine zu analysierende Eigenschaften, falls keine false negatives möglich sind.
- **Unterapproximation** für eine zu analysierende Eigenschaft, falls keine false positives möglich sind.

# Diskussion: Verlässlichkeit von Analyse-Techniken

Diese Definitionen, wörtlich genommen, hieße:

- Eine Analyse-Technik, die alle Analyseobjekte „durchwinkt“, ist konsistent.
- Eine Analyse-Technik, die alle Analyseobjekte „verdächtig findet“, ist vollständig.

# Diskussion: Verlässlichkeit von Analyse-Techniken

- Eine direkte Folge des Satzes von Rice ist, dass eine korrekte (= konsistente **und** vollständige) Analyse-Theorie (und somit auch eine entsprechende Analyse-Technik) für nicht-triviale Programmeigenschaften unmöglich ist.
- In der Praxis ist es zumeist noch nicht einmal möglich, nachzuweisen, dass eine Analyse-Technik wenigstens eine der beiden Eigenschaften garantiert.

# Diskussion: Verlässlichkeit von Analyse-Techniken

- Für die Analyse **unkritischer Programm-Eigenschaften** sollte vor allem die Anzahl der false positives verringert werden, um die Nutzerakzeptanz für die Analyse-Technik nicht zu gefährden.
- Ansonsten tritt der Effekt zunehmender Resilienz gegen vermeintliche „Fehlalarme“ ein.
- Dagegen sind false negatives (gelegentliches „Übersehen“ von Problemstellen) nicht so tragisch.
- Beispiel: Detektion von Design-Problemen (Anti-Pattern, Code-Smells).

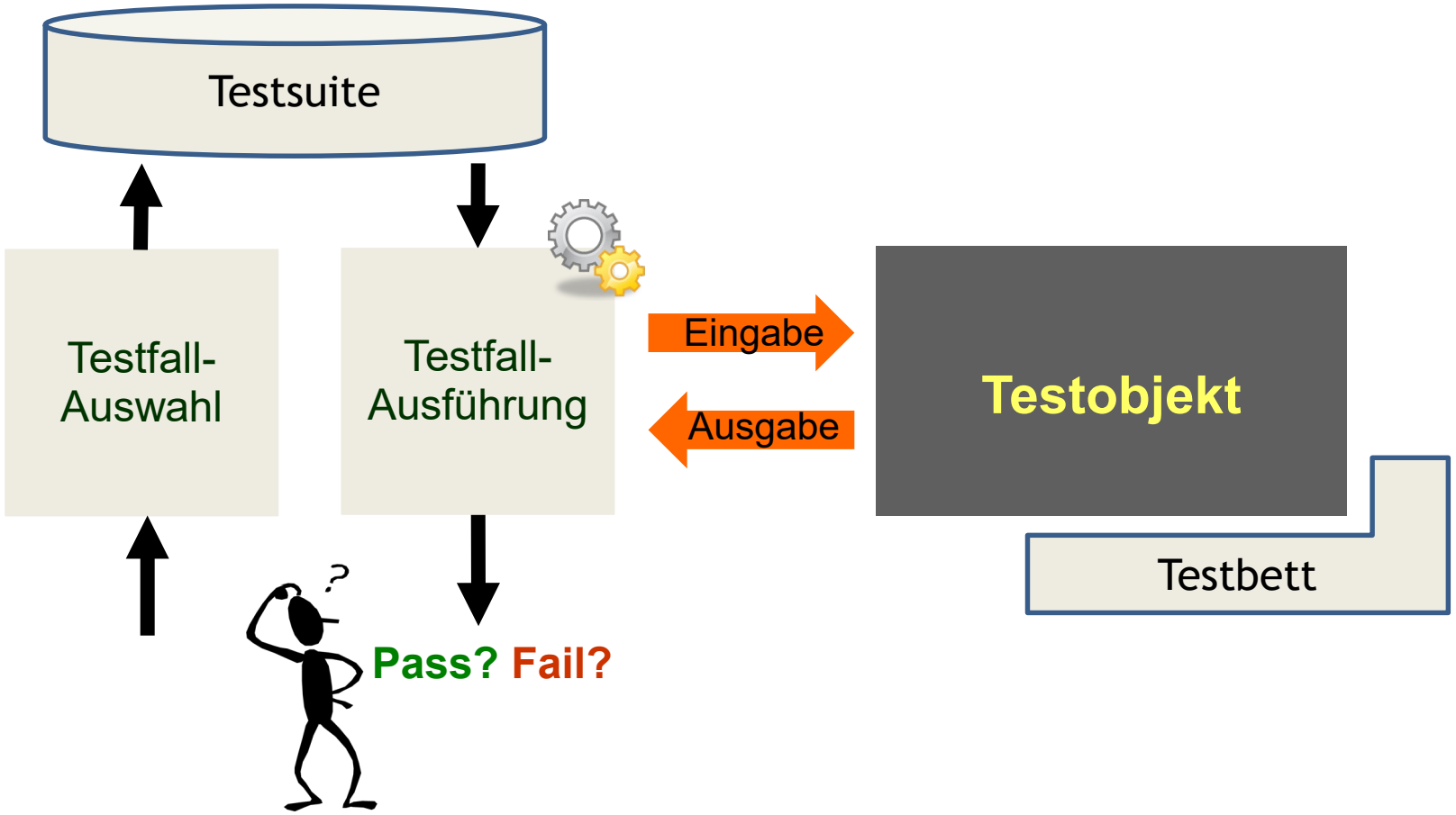
# Diskussion: Verlässlichkeit von Analyse-Techniken

- Für die Analyse **kritischer Programm-Eigenschaften** sollten hingegen auf jeden Fall false negatives vermieden werden, um kein potentiell fatales Problem zu übersehen.
- Ansonsten tritt der Effekt zunehmenden Misstrauens gegenüber den Analyse-Ergebnissen ein.
- Dagegen sind false positives („Fehleralarme“) hinzunehmen.
- Beispiel: Analyse der Einhaltung von Security-Policies.

# Präzision

Bei der Bewertung und dem Vergleich von Analyse-Techniken ist ein wichtiges Maß somit die **Präzision** der Ergebnisse, also das (mittlere) Verhältnis von false positives zu true positives und/oder false negatives zu true negatives.

# Testen als Analyse-Technik



# Auswahl und Ausführung von Testfällen

Die **Definition** eines Testfalls besteht aus:

- Auswahl von Eingabe-Daten für das Testobjekt.
- Ausgabe, die für diese Eingaben laut Spezifikation vom Testobjekt erwartet wird (Test-Orakel).

Die **Ausführung** eines Testfalls besteht aus:

- Experimentelle Anwendung der Eingabe-Daten des Testfalls auf das Testobjekt.
- Vergleich der am Testobjekt beobachteten Ausgabe mit der laut Testfall erwarteten Ausgabe.
- Abschließende Entscheidung, ob das Testobjekt den Testfall bestanden hat.

# Auswahl von Testsuiten

- Eine Testsuite ist eine (endliche) **Menge von Testfällen**, die aus einer (potentiell unendlichen) Menge möglicher Eingabe-Daten ausgewählt wird.
- Die **Effizienz** einer Testsuite bemisst sich durch den *Aufwand*, der für die Auswahl und Ausführung der Testfälle entsteht (gutes Schätzmaß: die *Anzahl* der gewählten Testfälle).
- Die **Effektivität** einer Testsuite ergibt sich durch die *Fehlerdetektionsrate* der ausgewählten Testfälle (die Höhe des Anteils der im Testobjekt vorhandenen Fehler, die durch mindestens einen Testfall entdeckt werden).

# Einschub: Effektivität

Häufiges Missverständnis im deutschen Sprachgebrauch:

- Effektivität hat nichts mit „ökonomisch schlauer Vorgehensweise“ zu tun.
- Effektivität beschreibt wortwörtlich ausschließlich das Maß des erzielten *Effekts* einer Aktivität (Beim Testen: Fehler finden!)
- Ein Effektivitätsmaß kann z.B. die zuvor besprochene Präzision sein.

# Effizienz vs. Effektivität

- Merke: Beide Ziele sind immer **gegenläufig**.
  - Die effizienteste Testsuite ist „leer“.
  - Die effektivste Testsuite ist „unendlich groß“.
- Wird von einem neuen Testverfahren behauptet, dass es Effizienz und Effektivität des Testens zugleich verbessert, ist das stets falsch.
- Vielmehr geht es bei Testverfahren darum, einen möglichst guten **Kompromiss** zwischen beiden Zielen zu erreichen.

# Effizienz schlägt Effektivität?

Ein Grund-Dilemma beim Testen besteht darin, dass während der Auswahl weiterer Testfälle zwar laufend die Effizienz, aber nicht die Effektivität sinnvoll bewertet werden kann:

- Die Größe der Testsuite und der bisherige Testaufwand (Zeit, Personen, …) sind jederzeit messbar.
- Der Anteil der mit der Testsuite bisher gefundenen Fehler an der Gesamtzahl ist hingegen unbekannt („Würde man bereits alle zu findenden Fehler kennen, bräuchte man ja nicht mehr zu testen…“).
- In diesem Kapitel der Vorlesung geht es im Wesentlichen darum, geeignete Ersatzkriterien für Effektivität und deren Auswirkung auf Effizienz kennenzulernen.

# Was wird getestet?

- **Funktionalitätstest**: Korrektes Ausgabeverhalten der Software für Eingabedaten innerhalb des spezifizierten Bereichs.
- **Robustheitstest: Sicheres** Ausgabeverhalten der Software für Eingabedaten außerhalb des spezifizierten Bereichs.
- **Lasttest**: Schrittweise quantitative Erhöhung der Systemlast (über Menge der Eingabedaten) *innerhalb* des zugelassenen/spezifizierten Bereiches.
- **Stresstest**: Schrittweise quantitative Erhöhung der Systemlast (über Menge der Eingabedaten) *außerhalb* des zugelassenen/spezifizierten Bereiches.
- **Benutzbarkeitstest**: Intuitive Gestaltung der Benutzeroberfläche.
- **Performanztest**: Laufzeitverhalten und Speicherplatzverbrauch, Identifikation ineffizienter Programmteile oder Speicherlecks.
- **Verlässlichkeitstest:** Korrektes Verhalten über längeren Betriebszeitraum.
- …

# Testarten

**Black-Box Tests:**

- Es wird das Ein-/Ausgabeverhalten des Testobjektes an der (Nutzer-)Schnittstelle gegen die Spezifikation getestet.
- Bei der Testfallauswahl werden interne Details (Struktur, Programm-Code etc.) des Testobjektes nicht betrachtet bzw. sind nicht verfügbar.

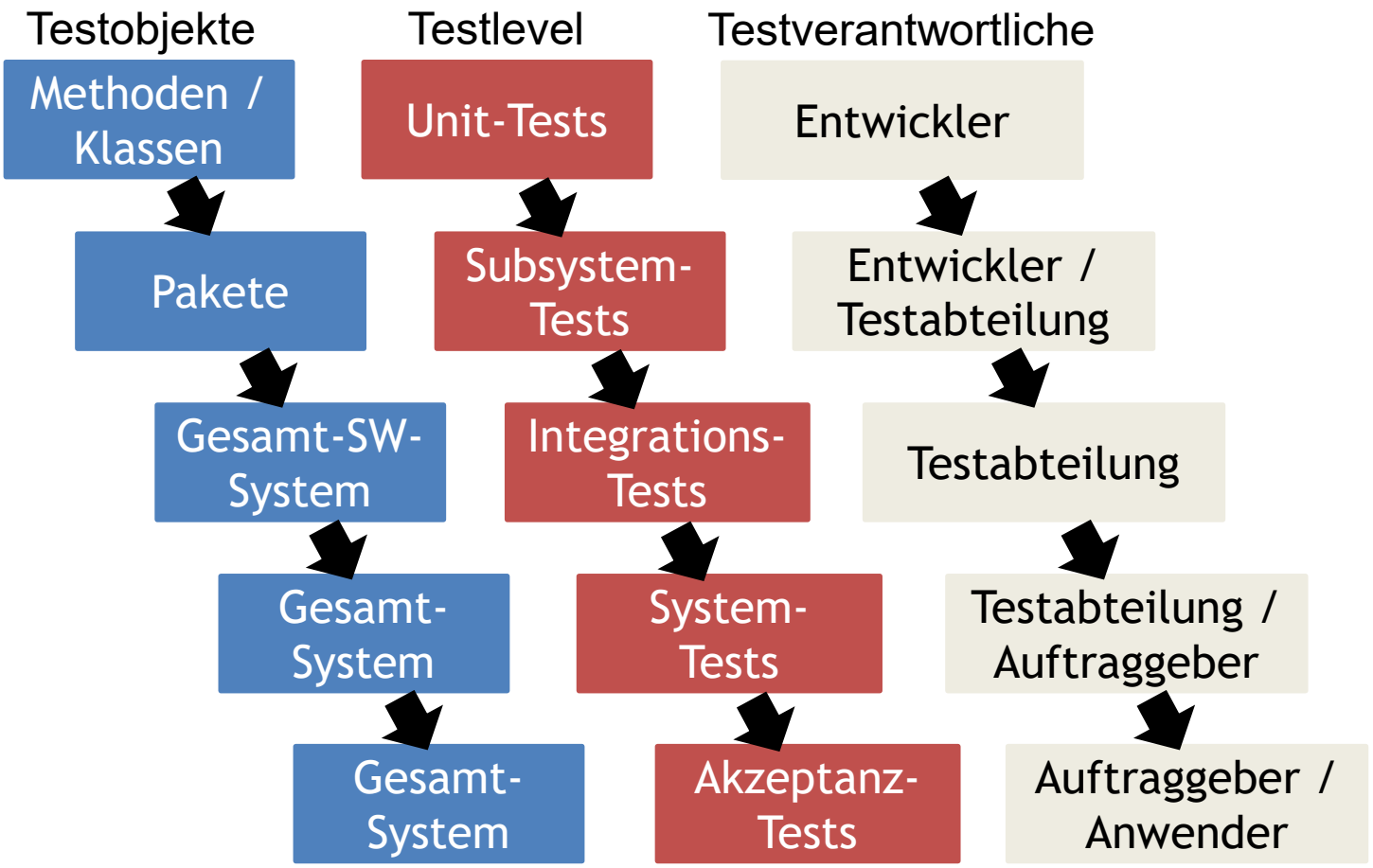
**White-Box Tests:**

- Interne Details des Testobjektes werden bei der Testfallauswahl und Testfallausführung herangezogen.

**Diversifikationstests:**

- Das Verhalten einer Version des Testobjektes wird nicht mit einer Spezifikation, sondern mit dem Verhalten einer anderen Version des Testobjektes verglichen.

# Testlevel


Testobjekte
Testlevel
Testverantwortliche
Methoden / Klassen
Unit-Tests
Entwickler
Pakete
Subsystem-Tests
Entwickler / Testabteilung
Gesamt-SW-System
Integrations-Tests
Testabteilung
Gesamt-System
System-Tests
Testabteilung / Auftraggeber
Gesamt-System
Akzeptanz-Tests
Auftraggeber / Anwender

# Systemtests

- Vor dem Abnahmetest erfolgt der Systemtest beim Softwareentwickler durch Kunden ($\alpha$**-Test**) und/ oder bei ausgewählten Pilotkunden vor Ort ($\beta$**-Test**)
- Beim Systemtest wird aus der **Sicht des Kunden** (nicht mehr aus Sicht der Entwickler) auf möglichst realistischer Hardware, Produktionsumgebungen und Geschäftsprozessen überprüft, ob das Gesamtprodukt die Anforderungen erfüllt.

# Akzeptanztests

- Auf Basis des Abnahmetests entscheidet der Kunde, ob das bestellte Softwaresystem **mangelfrei** ist, die im Lastenheft **festgelegten Anforderungen** erfüllt und von allen relevanten Anwendergruppen **akzeptiert** wird.
- Die durchgeführten Testfälle sollten bereits im **Vertrag** mit dem Kunden spezifiziert sein.

# Weitere Testbegriffe

- **Aktive Tests:** Stimulation des Testobjektes durch Eingaben und Beobachten des Ausgabeverhaltens.
- **Passive Tests:** Nur Beobachten des Ausgabeverhaltens bei der „natürlichen“ Interaktion des Testobjektes mit seiner Umgebung (auch „Monitoring“ genannt).
- **Positiv-Tests:** Untersuchung des Ausgabeverhaltens des Testobjektes für spezifizierte (erwartete) Eingaben (auch „Konformitätstesten“ genannt).
- **Negativ-Tests:** Untersuchung des Ausgabeverhaltens des Testobjektes für unspezifizierte (unerwartete) Eingaben (siehe z.B. Robustheitstests).

# Testen im agilen Zeitalter

Testen an sich und dabei insbesondere die folgenden Testtechniken spielen heute eine herausragende Rolle:

- **Unit-Tests:** Insbesondere für die testgetriebene Entwicklung objektorientierter Programme essentiell.
- **Regressionstests**: Bei jeder Änderung (z.B. Wartung oder Fehlerkorrektur) möglichst alle bisherigen Testfälle für auch bereits getestete Funktionen immer wieder erneut ausführen.
- **Inkrementelle Tests**: Integration neuer Bestandteile in ein bestehendes Programmgerüst wird durch ständige Verfeinerung von Testsuiten abgesichert.
- **Testautomatisierung:** Testauswahl und Testausführung soweit wie möglich automatisieren.

# Wann ist das Testende erreicht?

**In der Theorie: nie** – weitere Fehler sind nie ausgeschlossen.

**In der Praxis:** (sortiert von akzeptabel bis problematisch)

- Eine bestimmte Anzahl absichtlich implantierter Fehler (seeded bugs) wurden von einer Testgruppe gefunden (siehe Mutationstesten).
- Eine Testende-Kriterium wurde erreicht (siehe Abdeckungskriterien).
- Die Fehlerdetektionsrate sinkt unter eine magische Grenze (in der fragwürdigen Hoffnung, dass die Anzahl verbliebener Fehler mit der Anzahl/Häufigkeit gefundener Fehler korreliert).
- Das Testbudget ist verbraucht.
- Der Auslieferungszeitpunkt ist erreicht (Kunde testet unfreiwillig weiter ... ).

# Die 7 Grundsätze des Testens

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

UNIVERSITÄT SIEGEN

# Funktionsorientierte Testverfahren

## Black-Box Testen

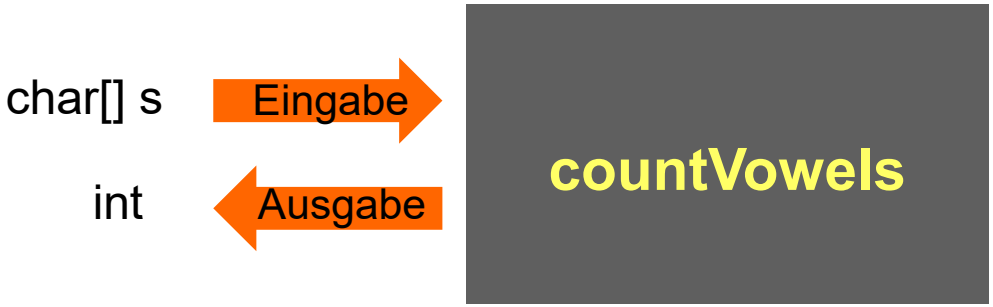
# Funktionsorientierte Testverfahren

- Testen des Ein-/Ausgabeverhaltens der Implementierung gegen die Spezifikation.
- Black-Box-Sicht: Programmstruktur und allen weiteren internen Details bleiben bei der Testfallauswahl und -Ausführung unberücksichtigt.
- Beispielsweise geeignet für Abnahmetest ohne Kenntnis / Zugriff des Quellcodes.

# Herausforderungen funktionsorientierter Testverfahren

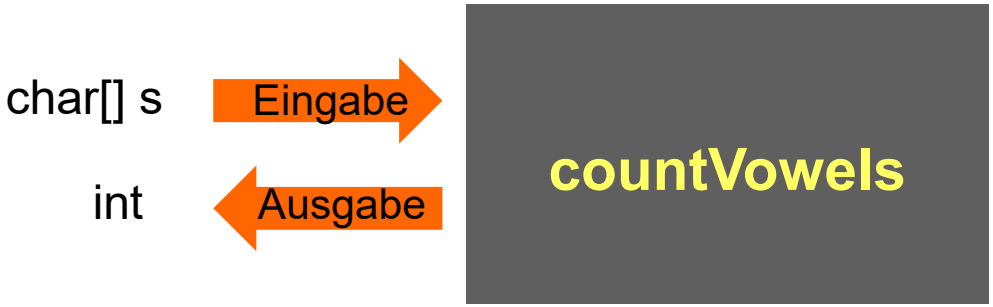
- Setzt (eigentlich) **vollständige und widerspruchsfreie Spezifikation** voraus (zur Auswahl von Testdaten und Interpretation von Testergebnissen)
- Zumeist sind nicht alle möglichen Eingabewert-Kombinationen testbar, deshalb müssen **repräsentative Eingabewerte** ausgewählt werden (sehr ähnlich zum Sampling).
- Für die (automatisierte) Überprüfung der Korrektheit der Ausgaben wird ein **Testorakel** gebraucht (das gilt allerdings für alle Testverfahren).

# Beispiel: Funktionsorientierter Test




- Die Eingabe-Domäne ist (theoretisch) die Menge aller möglichen Sätze beliebiger Länge.
- Eine Testerin muss repräsentative Sätze als Testfälle auswählen.

# Beispiel: Funktionsorientierter Test




Die Testerin könnte sich z.B. ein Test-Skript schreiben, das (pseudo-)zufällige „Sätze“ (Zeichenfolgen) mit einer konfigurierbaren Anzahl Vokale (= Testorakel) generiert und müsste sich dafür geeignete Kriterien überlegen:

- Wie lang sollten die Sätze mindestens/höchsten sein?
- Wie sollten die Vokale darin verteilt sein?
- Welche Extremfälle gibt es? (leerer Satz, Satz ohne Vokale, Satz nur mit Vokalen, Vokale genau an erster/letzter Position, …)
- …

# Äquivalenzklassen

- Partitionierung des (unendlichen) Eingabewertbereiches in eine endlichen Menge von **Äquivalenzklassen** (Eingabewertklassen).
- Für alle Repräsentanten einer Klasse sollte sich das Testobjekt gleich verhalten (laut Spezifikation).

# Auswahlkriterien für Äquivalenzklassen

- Trennung in **gültige** (spezifizierte) und **ungültige** (fehlerhafte) Eingabewertklassen.
- Gesonderte Betrachtung besonders **„großer“** und **„kleiner“** Eingabewertklassen (Lasttests).
- Bildung von Eingabewertklassen, die zu bestimmten (Klassen von) **Ausgabewerten** führen (allgemein sehr schwierig).

# Regeln für Eingabewertklassen

Für geordnete Wertebereiche (z.B. date):

| | |
|---|---|
| **]uv .. ov[** | offenes Intervall aller Werte zwischen **uv** und **ov** (ohne **uv** und **ov**). |
| **[uv .. ov]** | geschlossenes Intervall aller Werte zwischen **uv** und **ov** (mit **uv** und **ov**). |
| **]uv .. ov]** | halboffenes Intervall (ohne **uv**, mit **ov**) |
| **[uv .. ov[** | halboffenes Intervall (mit **uv**, ohne **ov**) |

# Regeln für Eingabewertklassen

Für ganze Zahlen (z.B. Integer):

| | |
|---|---|
| **[MinInt .. ov]** | Intervalle mit kleinster darstellbarer Integer-Zahl. |
| **[uv .. MaxInt]** | Intervalle mit größter darstellbarer Integer-Zahl. |

(offene Intervallgrenzen entsprechend)

# Regeln für Eingabewertklassen

Für reelle Zahlen (z. B. Float) ohne darstellbare kleinste/größte Zahl:

| | |
|---|---|
| **]-∞ .. ov]** oder **]- ∞.. ov[** | nach unten offene Intervalle. |
| **[uv .. ∞[** oder **]uv .. ∞[** | nach oben offene Intervalle. |

(alle Mischformen von Intervallen mit festen unteren und oberen Grenzen entsprechend)

# Regeln für Eingabewertklassen

Für beliebige Wertebereiche:

**{v1 v2 v3 ... vn}** Auswahl von genau n konkreten Werten

Für Zeichenketten (Strings):

- Reguläre Ausdrücke oder Grammatiken.
- Aufzählung konkreter Werte.

# Regeln für Eingabewertklassen

Für zusammengesetzte Wertebereiche:

- Anwendung entsprechender Regeln auf die einzelnen Wertebereiche
- Eventuell zusätzliche Einschränkungen für zulässige Wertekombinationen.

Für Wertebereiche, die aus genau einem Wert bestehen:

**[v]** der Wert selbst (auch **{v}** oder **v** üblich)

# Auswahl von Testfällen aus Eingabewertklassen

- Aus jeder Eingabewerteklasse wird **mindestens ein Wert** ausgewählt (Fehler- und Lastklassen werden später gesondert behandelt).
- Die Auswahl dieses Wertes kann **zufällig** erfolgen (oder auch bei Intervallen genau der Mittelwert des Intervalls sein oder...)

# Grenzwertanalyse

Unterteilung von Intervalle in Teilintervalle:

**]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 hierfür die gewünschte Genauigkeit festgelegt werden.

# Erläuterung: Grenzwertanalyse

- Mit der Grenzwertanalyse werden also Werte um die Bereichsgrenzen herum gewählt.
- Gewählt werden die Grenzen selbst sowie die um eins größeren und kleineren Werte.
- Idee dabei: oft werden Schleifen über Intervalle programmiert, die genau für die Grenzfälle fehlerhaft sind (z.B. falsches Abbruchkriterium).

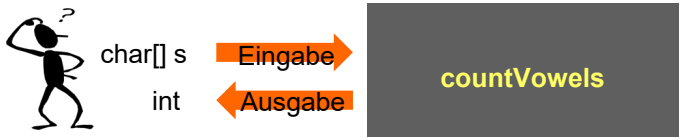
# Beispiel: Grenzwertanalyse

Eingabewertebereich sind die reellen Zahlen (Float) zwischen -1 und +1 mit zwei Nachkommastellen Genauigkeit (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 .. +∞[**

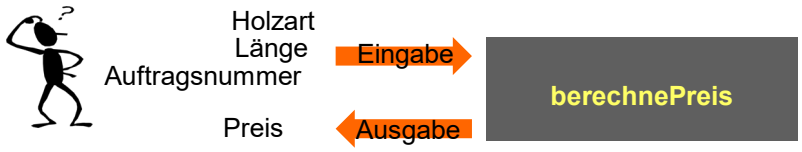
Beispiele für gewählte Werte: **-2, -1.01, -1.0, 0, +1.0, +1.01, +2**

# Beispiel: Funktionsorientierte Testauswahl




- Klasse 1: s endet mit einem Punkt und enthält keine Vokale:
  - 1a: s besteht nur aus einem Punkt.
  - 1b: s besteht aus einem Konsonanten gefolgt von einem Punkt.
  - 1c: s enthält sonstige Sonderzeichen (wie etwa !”§$%&/()=?).
- Klasse 1: s endet nicht mit einem Punkt.
- Klasse 3: s endet mit einem Punkt und enthält einen Vokal:
  - 3a: s enthält ein a, e, i, o, u
  - 3b: s enthält ein A, E, I, O, U
- Klasse 4: s enthält mehrere Vokale:
  - 4a: mehrere gleiche Vokale
  - 4b: mehrere verschiedene Vokale
- Klasse 5: Eingabe ist sehr lang und enthält ganz viele Vokale

# Beispiel: Lagerverwaltungs-Software für Holzbretter



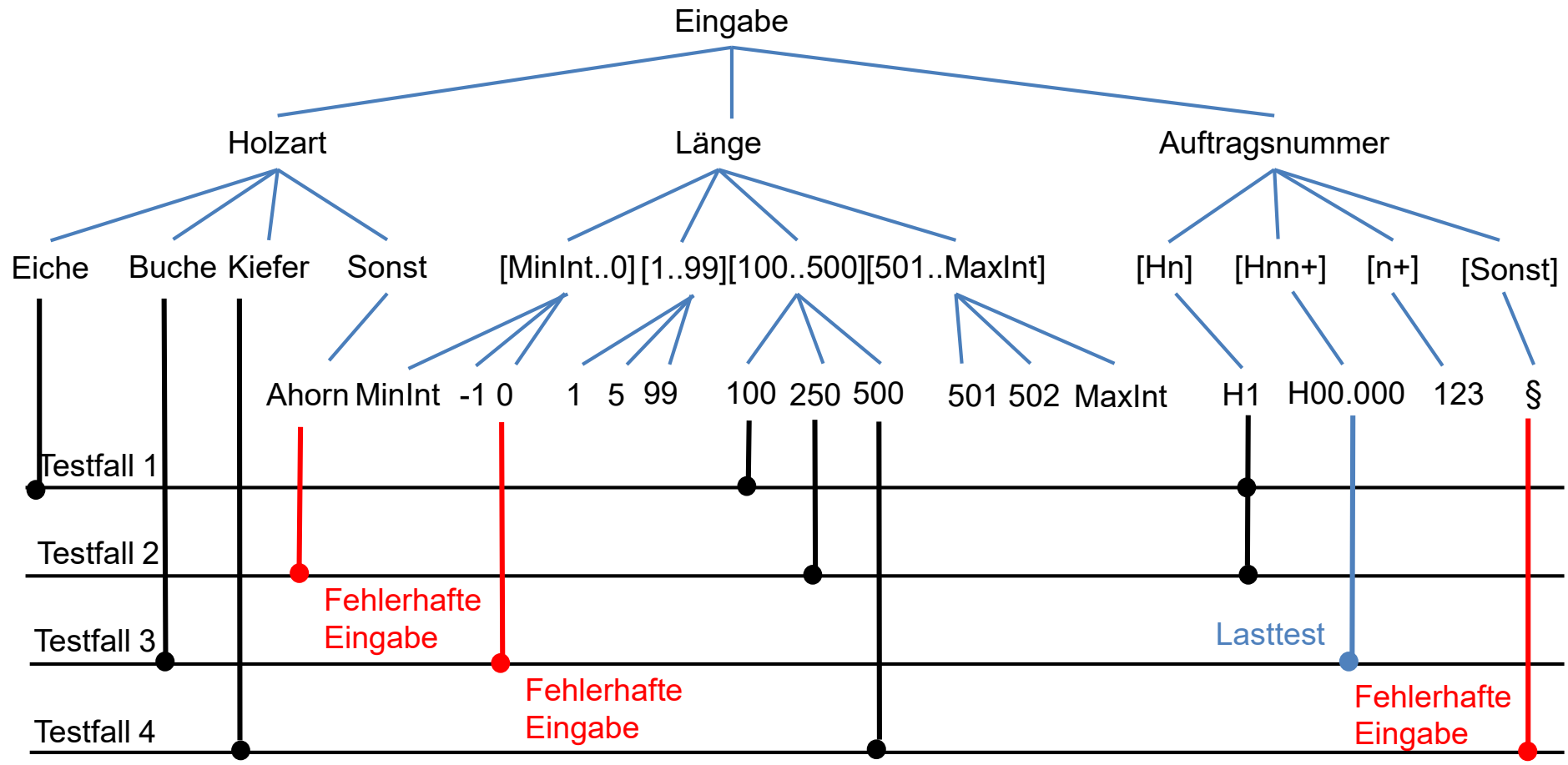

Eingaben:

- Holzart: Eiche, Kiefer, Buche (String).
- Brettlänge: zwischen 100 und 500 cm.
- Auftragsnummer: Beginn mit „H“ beginnt, gefolgt von mindestens einer (aber beliebig vielen) Ziffer(n)

Ausgabe:

- Preis des Auftrags (errechnet aus Brettlänge und Holzart).

# Beispiel: Äquivalenzklassen als Klassifikationsbaum



# Diskussion: Klassifikationsbaum

- Ein „vollständiger“ Funktionstest würde die Selektion sämtlicher möglicher Wertekombinationen verlangen.
- Trotz Bildung von Äquivalenzklassen bleiben immer noch (zu) viele mögliche Wertekombinationen übrig (im vorherigen Beispiel sind nur exemplarische Testfälle aufgeführt).
- Deshalb Verwendung von weiteren Heuristiken zur Reduktion möglicher Wertekombinationen.

# Heuristiken zur Testfallselektion

- Aus jeder **Äquivalenzklasse** wird *mindestens einmal* ein Wert in *einem* Testfall ausgewählt.
- Bei Grenzwertanalyse werden *drei* Werte ausgewählt.

# Heuristiken zur Testfallselektion

- Bei **abhängigen Eingabeparametern aus verschiedenen Äquivalenzklassen** muss *jede* Wertekombinationen aus den jeweiligen „normalen“ Äquivalenzklassen in *mindestens einem* Testfall ausgewählt werden.
- Parameter sind abhängig, wenn sie gemeinsam das Verhalten des Programms steuern (und deshalb nicht unabhängig voneinander betrachtet werden können).

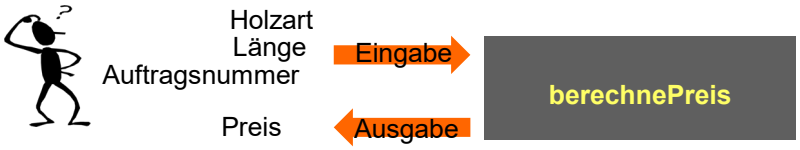
# Heuristiken zur Testfallselektion

- Aus jeder **Fehleräquivalenzklasse** wird *genau ein* Wert in *genau einem* Testfall ausgewählt.
- In jedem Testfall kommt *maximal eine* fehlerhafte Eingabe vor.
- Fehleräquivalenzklassen sind Äquivalenzklassen, die unzulässige Eingabewerte zusammenfassen.

# Heuristiken zur Testfallselektion

- Aus jeder **Lasttestklasse** wird *genau ein* Wert in *genau einem* Testfall ausgewählt.
- In jedem Testfall kommt *maximal eine* Lasttest-Eingabe vor, diese kann aber mit einer fehlerhaften Eingabe kombiniert werden.
- Lasttestklassen sind Äquivalenzklassen, die besonders „große/lange/...“ zulässige/gültige Eingabewerte zusammenfassen.

# Beispiel: Testfallselektion aus Äquivalenzklassen

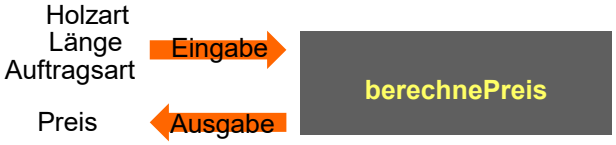



- Testfall 1: Holzart = „Eiche“, Länge = 100, Auftragsnummer = 1
- Testfall 2: Holzart = „Ahorn“, Länge = 250, Auftragsnummer = H1
- Testfall 3: Holzart = „Buche“, Länge = 0, Auftragsnummer = H00…0
- Testfall 4: Holzart = „Kiefer“, Länge = 500, Auftragsnummer = §

# Diskussion: Testfallselektion aus Äquivalenzklassen

- Alle zulässigen Kombinationen von Äquivalenzklassen für Holzarten und Längenangaben müssten durchgetestet werden, da laut Spezifikation Holzart und Länge gemeinsam den Preis des Auftrags bestimmen
- Die Auftragsnummer hat laut Spezifikation dagegen keinen Einfluss auf die Berechnung des Preises und kann in Testfällen jeweils mit einem Normalwert selektiert werden.
- In jedem Testfall wird maximal eine fehlerhafte Eingabe und maximal eine Lasttesteingaben selektiert, aber es wird nicht aus jeder Fehlerklasse mindestens eine fehlerhafte Eingabe in einem Testfall selektiert.

# Modifiziertes Beispiel




- Holzart (Aufzählungstyp): Buche, Eiche, Kiefer
- Länge (Zahl größer gleich Null) mit drei Intervallen, in denen unterschiedliche Preisschemata verwendet werden:
  - kurze Bretter mit [0 .. 99] cm
  - normale Bretter mit [100 .. 300] cm
  - (zu) lange Bretter mit ]300 .. +∞[ cm
- Auftragsart (Aufzählungstyp): DoItYourself, Normal, Express

# Modifiziertes Beispiel

- Alle **drei Parameter interagieren** bei der Berechnung des Preises eines Bretts.
- Somit müssten (mindestens) alle 3 x 3 x 3 = 27 Wertekombinationen der Äquivalenzklassen getestet werden (ohne Fehler und -Lasttests).
- Gleiche Idee wie beim Pairwise Sampling: betrachte „nur“ alle paarweisen Wertekombinationen der Äquivalenzklassen.

# Modifiziertes Beispiel: Paarweiser Testansatz

| **Holzart** | Buche | Buche | Buche | Eiche | Eiche | Eiche | Kiefer | Kiefer | Kiefer |
|---|---|---|---|---|---|---|---|---|---|
| **Länge** | kurz | normal | lang | kurz | normal | lang | kurz | normal | lang |
| **Art** | DoIt | Normal | Express | Express | DoIt | Normal | Normal | Express | DoIt |

Für paarweise Selektion: 9 statt 27 Testfälle.

# Diskussion: Funktionstest durch Äquivalenzklassen

- Gute Werkzeugunterstützung und einfach einsatzbar.
- Güte hängt wie immer stark von der Qualität der Spezifikation ab.
- Es wird nicht garantiert, dass jede Programmzeile einmal ausgeführt wurde.
- Ungeeignet für nichtfunktionale Anforderungen.
- Ungeeignet für zustandsbezogene Software (Ausgabeverhalten hängt von der nicht nur von der gerade getätigten Eingabe ab).

# Weitere Black-Box-Testverfahren

- **Intuitives Testen**: Aus dem Bauch heraus zusätzliche Testfälle festlegen (Expertenwissen und Erfahrung niemals unterschätzen!).
- **Zufallstest**: Zufällige Auswahl gemäß z.B. bekannter statistischer Verteilung (als Ergänzung gut geeignet, um „unerwartete“ Testdaten zu erhalten).
- **Smoke-Test / Monkey-Test**: Es wird nur Robustheit des Testobjekts getestet, berechnete Ausgabewerte spielen keine Rolle (wahllos auf Tastatur hämmern, ... ).
- **Syntax-Test**: Ist für Eingabewerte der erlaubte syntaktische Aufbau bekannt (als Grammatik angegeben) kann man daraus systematisch Testfälle generieren (z.B. Email-Adressen, URLs, siehe Fuzzing).
- **Ursache-Wirkungs-Graph-Analyse und Anwendungsfallbasiertes Testen** (hier nicht weiter betrachtet).

UNIVERSITÄT SIEGEN

# Kontrollflussorientierte Testverfahren

## White-Box Testen – Teil 1

# Grundidee kontrollflussbasierter Testverfahren

- **Auswahl** von Testfällen durch ein beliebiges Verfahren (z.B. zufällig oder durch Äquivalenzklassen).
- **Ausführung** der Testfälle und Beobachtung, welche Teile des *Kontrollflussgraphen* des Programms durchlaufen werden.
- **Bewertung**, ob die vorhandenen Testfälle den Kontrollflussgraphen ausreichend *überdecken*.
  - Gegebenenfalls werden solange *weitere Testfälle ausgewählt*, bis eine ausreichende Überdeckung des Kontrollflussgraphen erreicht wurde.
  - Gegebenenfalls werden *alte Testfälle gestrichen*, die dieselben (oder weniger) Teile des Kontrollflussgraphen überdecken.

# Kontrollflussbasierte Überdeckungskriterien

- Kontrollflussbasierte Überdeckungskriterien legen fest, auf welche Weise der Kontrollflussgraph eines zu testenden Programmes durch gewählte Testfälle zu überdecken ist.
- Zumeist definiert ein Überdeckungskriterium eine Menge von **Testzielen** auf dem Kontrollflussgraphen, die jeweils durch mindestens einen Testfall abzudecken (zu erfüllen) sind.
- Diese Kriterien werden auch **Testendekriterien** genannt, da sie steuern, wie lange weitere Testfälle zu selektieren sind.

# Anmerkung zu kontrollflussbasierten Testverfahren

- Kontrollflussbasierte (und später auch Datenflussbasierte) Testverfahren sind somit ein (Ersatz-)Maß zur Bewertung der **Effektivität** (= Grad der Überdeckung) einer bestehenden Menge von Testfällen.
- Techniken zur gezielten Selektion von Testfällen zur **effizienten** Abdeckung des Kontrollflusses (mit möglichst wenigen Testfällen) sind ein anderes (schwieriges) Thema.

# Beispiel: Fehlerhaftes Programm

```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0; i = 0;

  while (s[i] != '.') {

    if (s[i]='a'||s[i]='b'||s[i]='e'||s[i]='i'||s[i]='o'||s[i] = 'u') {

        count = 1;
        i = i+1;

    }
}
  return count;
}
```

```
// Aufruf der Methode
…
  String s = "to be … or not to be.";
  int nrVowels =   myObjetc.countVowels(s);
// *
…
```

Welche Fehler enthält der Programmausschnitt?

(* Wir nehmen vereinfachend an, dass der String automatisch in ein char-Array konvertiert wurde).

# Beispiel: Auflösung

```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0; i = 0;

  while (s[i] != '.') {
                    Falsche Prüfung auf 'b'
   if (s[i]='a'||s[i]='b'||s[i]='e'||s[i]='i'||s[i]='o'||s[i] = 'u') {

      count = 1;   count wird nicht erhöht
      i = i+1;     i wird an falscher Stelle erhöht

   }
}
  return count;
}
```

```
// Aufruf der Methode
…
  String s = "to be … or not to be.";
  int nrVowels =   myObjetc.countVowels(s);
// *
…
                    '.' nur am Satzende erlaubt.
```

(* Wir nehmen vereinfachend an, dass der String automatisch in ein char-Array konvertiert wurde).

# Beispiel: Kontrollgraph

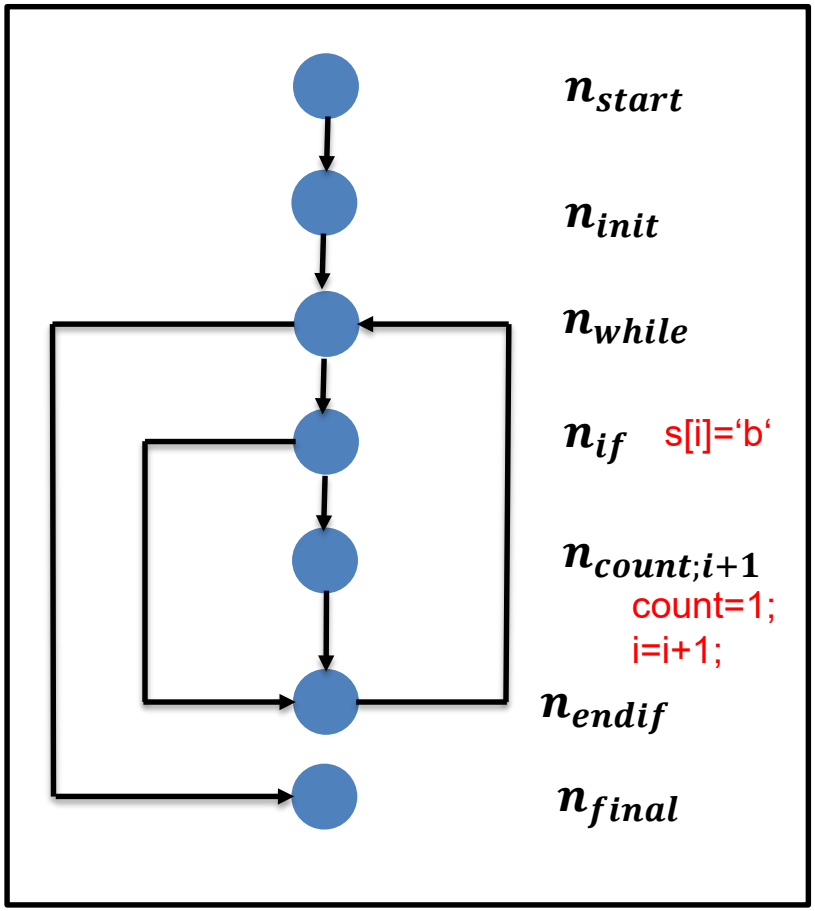
```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0; i = 0;

  while (s[i] != '.') {

    if (s[i]='a'||s[i]='b'||s[i]='e'||s[i]='i'||s[i]='o'||s[i] = 'u') {

      count = 1;
      i = i+1;

    }
}
  return count;
}
```



# Anweisungsüberdeckung ($C_0$-Test)

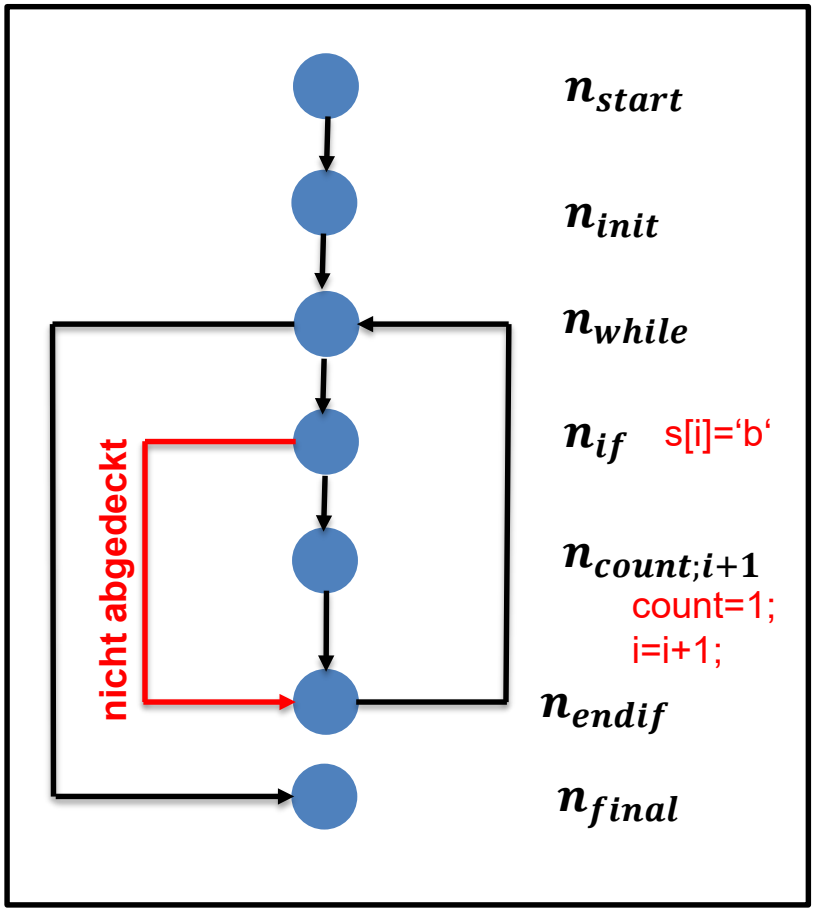
Die Anweisungsüberdeckung erfordert, dass jeder **Knoten** des Kontrollflussgraphen durch mindestens einen Testfall abgedeckt (erreicht/ausgeführt) wird.

# Beispiel: Anweisungsüberdeckung

- Für Anweisungsüber-deckung reicht ein Testfall bestehend aus einem konsonantenfreien Satz.

Beispiel: Testfall **“a.“**

- b-Fehler: *nicht* erkannt
- count-Fehler: *nicht* erkannt.
- i-Fehler: *nicht* erkannt.



# Diskussion: Anweisungsüberdeckung

- Minimalkriterium (wenn überhaupt…).
- Es wird dadurch noch nicht einmal garantier, das alle Kanten des Kontrollflussgraphen durchlaufen werden.
- Sehr viele Fehler bleiben unentdeckt.

# Zweigüberdeckung ($C_1$-Test)

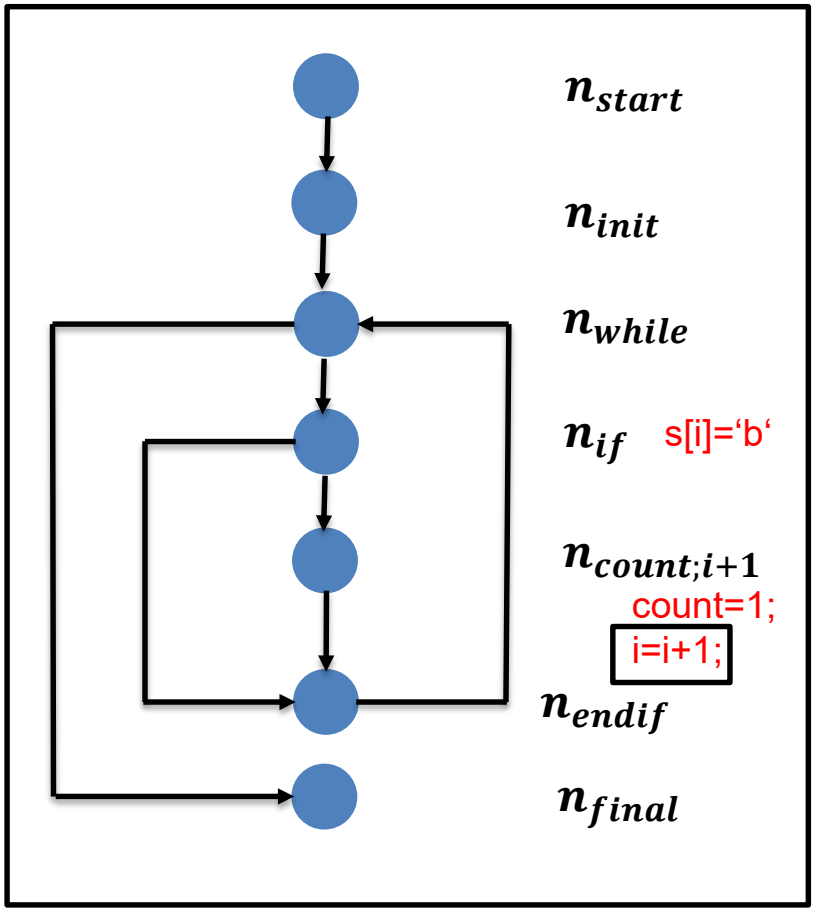
Die Zweigüberdeckung erfordert, dass jede **Kante** des Kontrollflussgraphen durch mindestens einen Testfall abgedeckt (erreicht/ausgeführt) wird.

# Beispiel: Anweisungsüberdeckung

- Für Anweisungsüber-deckung reicht ein Satz mit einem Konsonanten und einem Vokal.

Beispiel: Testfall **“ax.“**

- i-Fehler: erkannt.
- b-Fehler: *nicht* erkannt
- count-Fehler: *nicht* erkannt.



# Diskussion: Zweigüberdeckung

- Realistisches Minimalkriterium.
- Umfasst Anweisungsüberdeckung (d.h. jede Testsuite die Zweigüberdeckung erfüllt, erfüllt auch Anweisungsüberdeckung)
- Fehler bei Wiederholungen oder anderer Kombination von Zweigen bleiben unentdeckt.
  - b-Fehler: Es wird nicht erzwungen, dass jede Teilbedingung einer Verzweigung geprüft wird (hier: Sätze müssen nur irgendeinen Konsonanten und Vokal enthalten, nicht zwangsläufig das fehlerhafte ‚b‘).
  - count-Fehler: Es wird nicht erzwungen, dass unterschiedlich oft Schleifen-Verzweigungskombinationen durchlaufen werden (hier: Sätze müssen nicht unterschiedliche Anzahlen von Konsonanten und Vokalen beinhalten).

# Atomare Bedingungsüberdeckung

Die atomare Bedingungsüberdeckung erfordert, dass jede **Teilbedingung** einer Kontrollflussbedingung (z.B. von if- oder while-Anweisungen) mindestens einmal durch einen Testfall den Wert *true* und mindestens einmal durch einen Testfall den Wert Wert *false* annimmt.

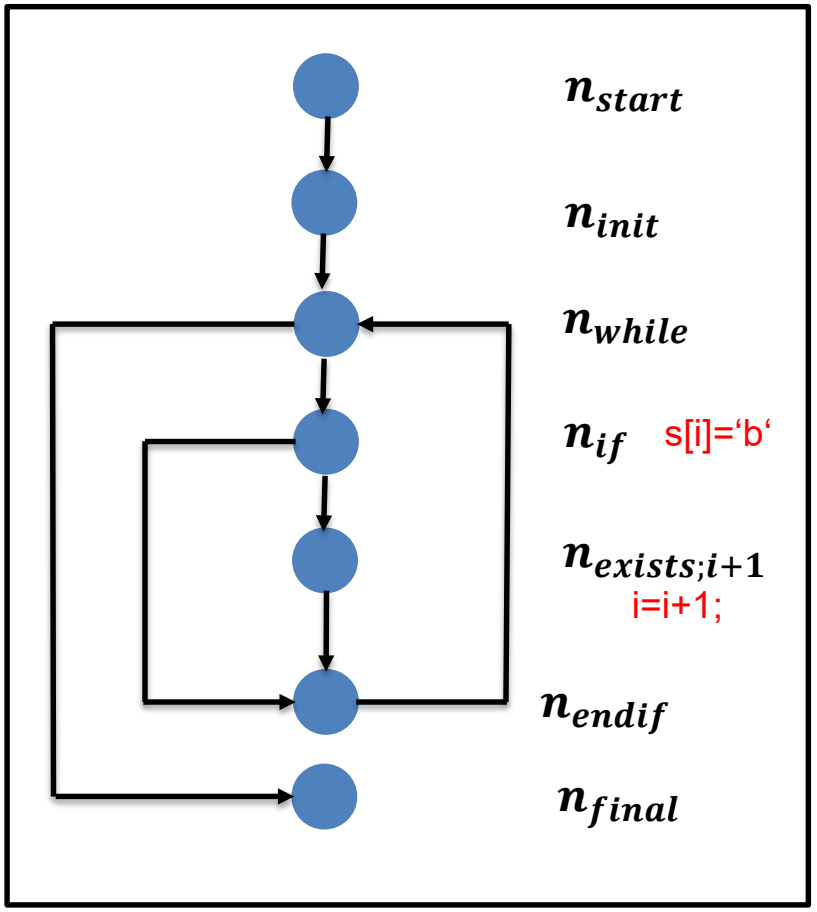
# Modifiziertes Beispiel

```
public boolean hasVowels(char[] s) {
// returns true if sentence s contains vowels.
// sentence must end with a dot.
  boolean exists; int i;
  exists = false; i = 0;

  while (s[i] != '.') {
                          Falsche Prüfung auf 'b'
    if (s[i]='a'||s[i]='b'||s[i]='e'||s[i]='i'||s[i]='o'||s[i] = 'u') {

       exists = true;
       i = i+1;        i wird an falscher Stelle erhöht

    }
}
  return exists;
}
```

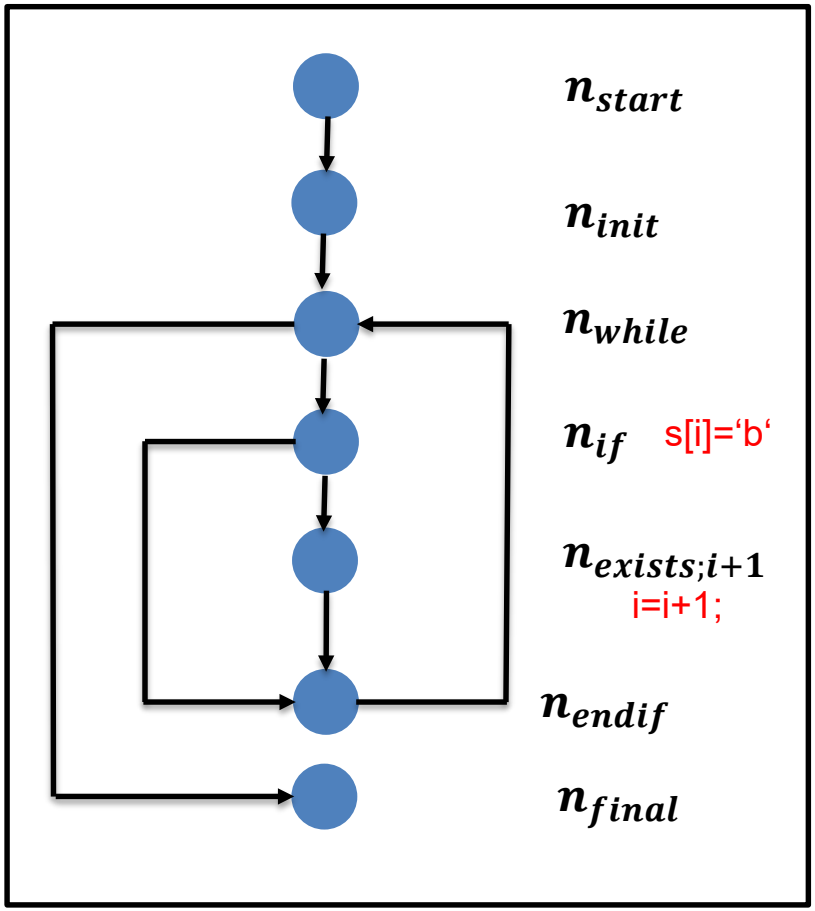


# Beispiel: Atomare Bedingungsüberdeckung

- Für atomare Bedingungsüberdeckung reicht ein Testfall bestehend aus einem Satz mit allen Vokalen und ‘b‘.

Beispiel: Testfall **“baeiou.“**

- b-Fehler: *nicht* erkannt
- i-Fehler: *nicht* erkannt.



# Diskussion: Atomare Bedingungsüberdeckung

- Es wird keine Anforderung an die Auswertung der Gesamtbedingungen gestellt.
- Für den gewählten Testfall ist die if-Bedingung bei jedem Schleifendurchlauf „wahr“, was zwar falsch ist, aber trotzdem am Ende eine „richtige“ Ausgabe liefert.
- Atomare Bedingungsüberdeckung umfasst somit nicht einmal Anweisungsüberdeckung!

# Minimale Mehrfachbedingungsüberdeckung

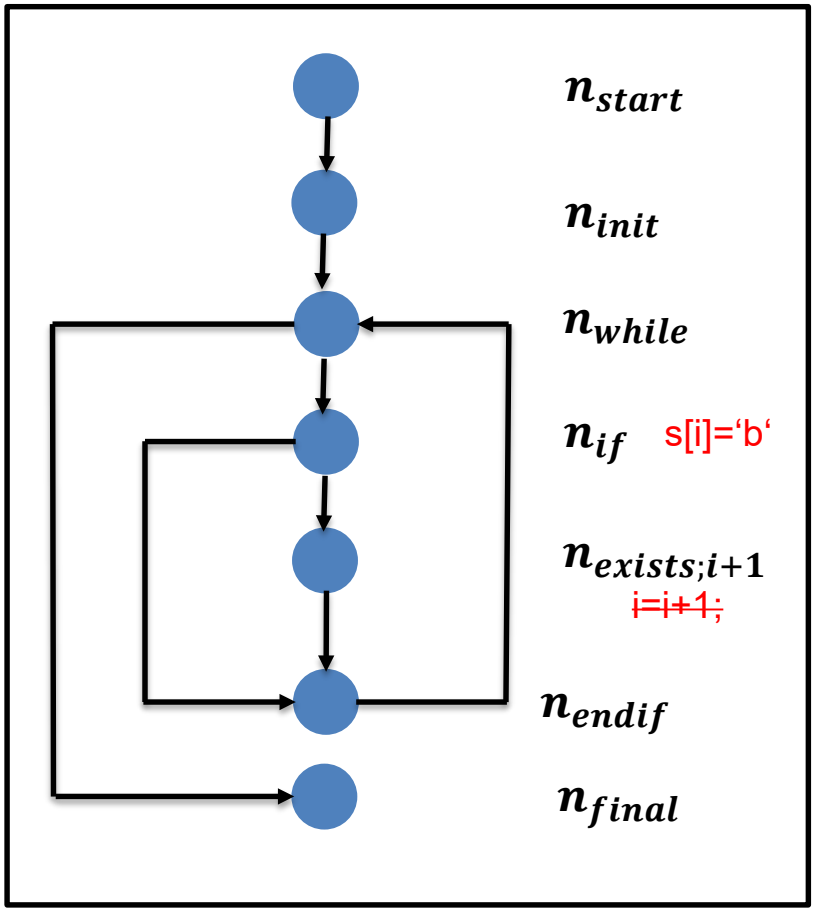
Die minimale Mehrfachüberdeckung erfordert, dass jede **Teilbedingung und die Gesamtbedingung** einer Kontrollflussbedingung (z.B. von if- oder while-Anweisungen) mindestens einmal durch einen Testfall den Wert *true* und mindestens einmal durch einen Testfall den Wert Wert *false* annimmt.

# Beispiel: Minimale Mehrfachbedingungsüberdeckung

- Für minimale Mehrfachbedingungs-überdeckung ein Satz mit allen Vokalen, ‘b‘ und einem weiteren Konsonanten.

Beispiel: Testfall **“xbaeiou.“**

- i-Fehler: erkannt.
- b-Fehler: *nicht* erkannt (nach Korrektur des i-Fehlers)



# Diskussion: Minimale Mehrfachbedingungsüberdeckung

- Minimale Mehrfachbedingungsüberdeckung umfasst Zweigüberdeckung, dennoch werden viele Fehler nicht gefunden.
- Im Falle von Bedingungs-Schleifenkombinationen ist es häufig ausreichend, einen „langen“ Testfall zu selektieren, um möglichst viele true/false-Bedingungsauswertungen zugleich abzudecken.
- Häufig wären stattdessen aber mehrere „kurze“ Testfälle hilfreich (z.B. “**b.**“, um den b-Fehler zu erkennen)

# Einschub: Short-Circuit-Evaluierung

**if** (<*Bedingung, die true liefert*> || b) {..}

**if** (<*Bedingung, die false liefert*> && b) {..}

In beiden Fällen ist (z.B. in Java) die Belegung von b irrelevant.

# Modifizierter Bedingungsüberdeckungstest

Der modifizierter Bedingungsüberdeckungstest beachtet die Short-Circuit-Evaluierung bei der Testfallauswahl.

Beispiel:

| a | b | a \|\| b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | - | 1 |

# Beispiel: Bedingungsüberdeckungsarten

| Kriterium | a | b | c | (a\|\|b)&&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 |
| Mod. mehrfach | 0 | 0 | - | 0 |
| | 1 | - | 1 | 1 |
| | 0 | 1 | 1 | 1 |
| | 1 | - | 0 | 0 |

# Pfadüberdeckung ($C_\infty$-Test)

Die Pfadüberdeckung erfordert, dass jeder **Pfad** des Kontrollflussgraphen durch mindestens einen Testfall abgedeckt (ausgeführt) wird.

# Diskussion: Pfadabdeckung

- Maximales Kriterium.
- Rein theoretischer Vergleichsmaßstab für realistische Testverfahren (sobald ein Programm Schleifen enthält, gibt es unendliche viele verschiedene Pfade).
- Findet paradoxerweise trotzdem nicht alle Fehler, da kein erschöpfender Test aller möglichen Eingabewerte erforderlich ist.

# Boundary-Test

Der Boundary-Test erfordert, dass alle **Pfade** des Kontrollflussgraphen, auf denen eine Schleife **maximal einmal** durchlaufen wird, durch mindestens einen Testfall abgedeckt (ausgeführt) werden.

(ohne praktische Bedeutung)

# Boundary-Interior-Test

Der Boundary-Test erfordert, dass alle **Pfade** des Kontrollflussgraphen, auf denen eine Schleife **maximal zweimal in direkter Abfolge** durchlaufen wird, durch mindestens einen Testfall abgedeckt (ausgeführt) werden.

# Modifizierter Boundary-Interior-Test

Der Boundary-Test erfordert, dass alle **Pfade** des Kontrollflussgraphen, auf denen eine Schleife **maximal zweimal** durchlaufen wird, durch mindestens einen Testfall abgedeckt (ausgeführt) werden.

# Abstraktes Beispiel: Boundary-Interior-Test

```
if (b1) n1; else n2;

while(b2) {
  n3;
  while(b3) {
    if(b4) n4; else n5;
  }
}
```

| | | |
|---|---|---|
| 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: Modifizierter Boundary-Interior-Test

```
if (b1) n1; else n2;

while(b2) {
  n3;
  while(b3) {
    if(b4) n4; else n5;
  }
}
```

| | | |
|---|---|---|
| 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 *) |

# Kontrollflussbasierter Test in der Praxis

- 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).

# Kontrollflussbasierter Test: Empfehlungen

- Zweigüberdeckung sollte immer Mindestanforderung beim Testen darstellen (Kontrollflussfehler/ Bedingungsfehler werden relativ gut gefunden, Datenflussfehler weniger gut).
- Ergänzend dazu eine einfache Variante des Modifizierten Boundary-Interior-Tests: für jede Schleife gibt es Testfälle/Pfade, die sie gar nicht, genau einmal und (mindestens) zweimal ausführen (d.h. die Randbedingung „alle Pfade“ wird komplett aufgegeben).

UNIVERSITÄT SIEGEN

# Datenflussorientierte Testverfahren

## White-Box Testen – Teil 2

# Grundidee datenflussbasierter Testverfahren

- **Auswahl** von Testfällen durch ein beliebiges Verfahren (z.B. zufällig oder durch Äquivalenzklassen).
- **Ausführung** der Testfälle und Beobachtung, welche Teile des *Datenflussgraphen bzw. des mit Datenflussattributen annotierten Kontrollflussgraphen* des Programms durchlaufen werden.
- **Bewertung**, ob die vorhandenen Testfälle den Datenfluss ausreichend *überdecken*.
  - Gegebenenfalls werden solange *weitere Testfälle ausgewählt*, bis eine ausreichende Überdeckung des Datenflusses erreicht wurde.
  - Gegebenenfalls werden *alte Testfälle gestrichen*, die dieselben (oder weniger) Teile des Datenflusses überdecken.

# Datenflussbasierte Überdeckungskriterien

Idee: Für jede Programmstelle mit einer Zuweisung eines Wertes an eine Variable sollen mindestens eine / … /alle (berechnenden, prädikativen) Benutzungen dieses Wertes im folgenden Programm getestet werden.

# Beispiel: Kontrollflussgraph mit Datenflussattributen

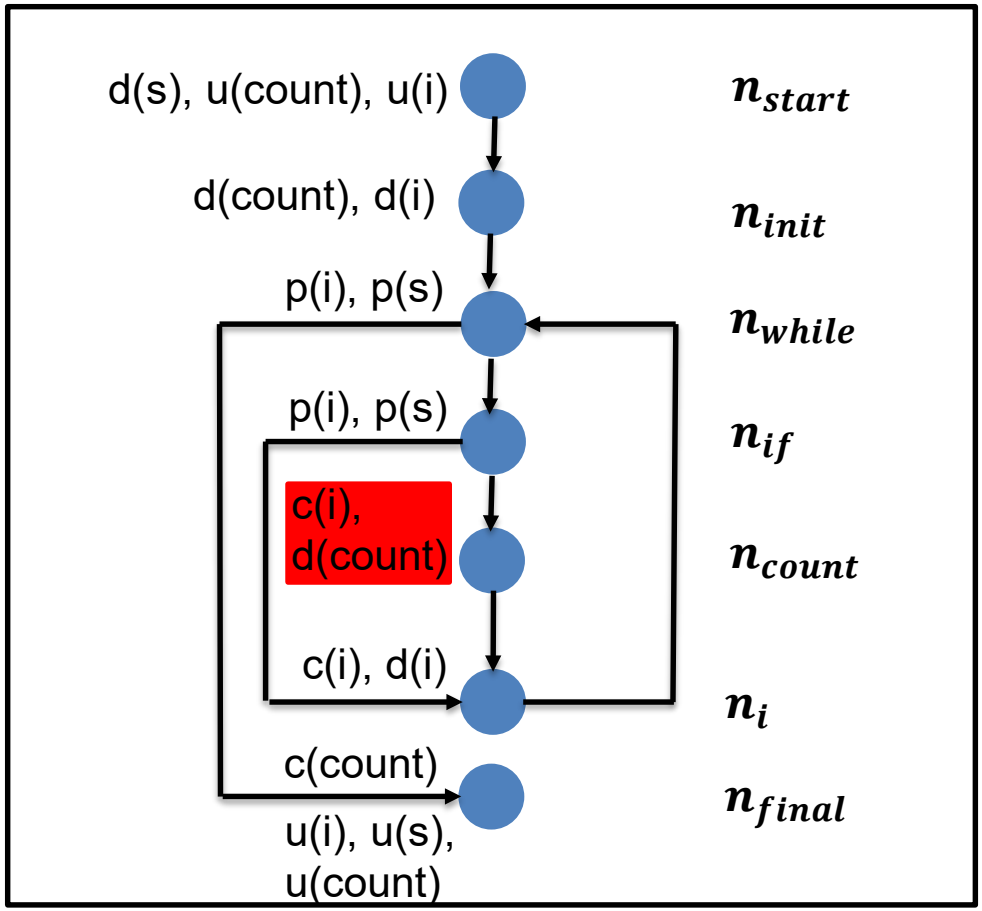
```
public int countVowels(char[] s) {
// count number of vowels in sentence s.
// sentence must end with a dot.
  int count, i;
  count = 0; i = 0;

  while (s[i] != '.') {

    if (s[i] = 'a' || s[i] = 'e' ||
        s[i] = 'i'  || s[i] = 'o' || s[i] = 'u') {

        count = i + 1;    Datenflussfehler

    }
    i = i+1;
  }
  return count;
}
```

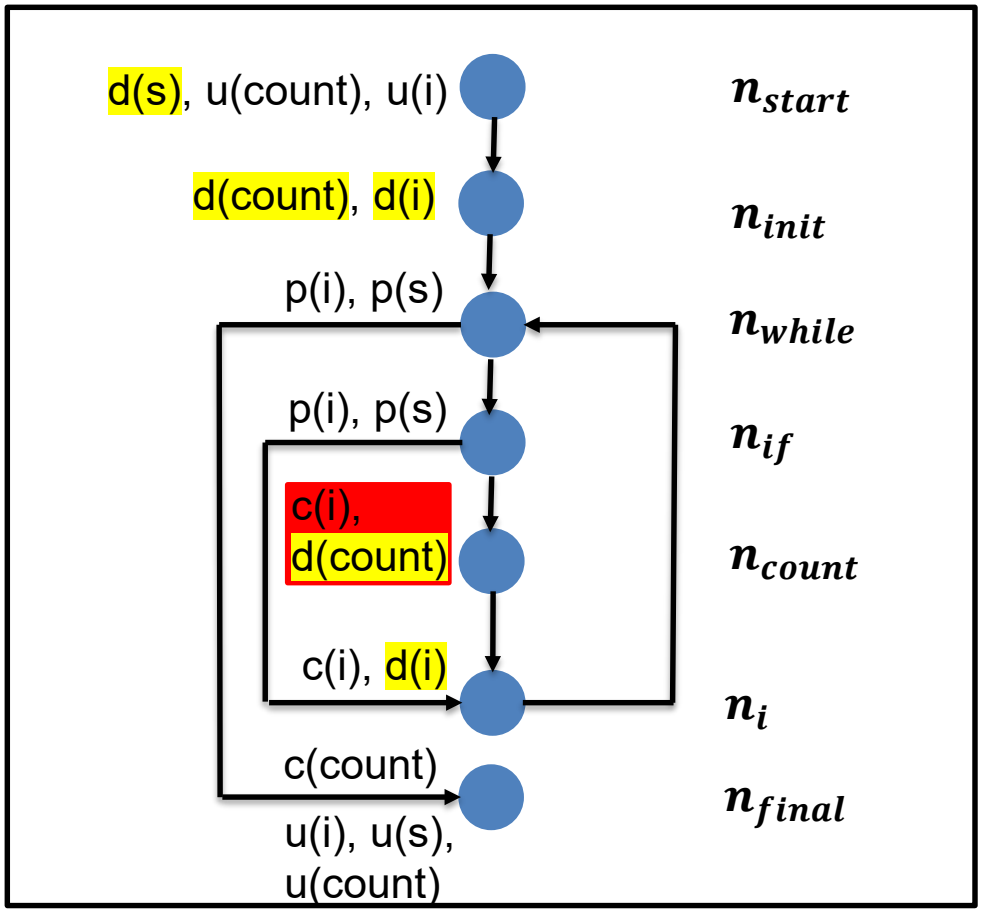


## all-defs-Kriterium

Das all-defs-Kriterium fordert, dass für **jede** Definitionsstelle *d(x)* einer Variablen mindestens **ein** definitionsfreier Pfad zu **einer** Benutzungsstelle *r(x)* durch einen Testfall abgedeckt wird.

# Beispiel: all-defs-Kriterium

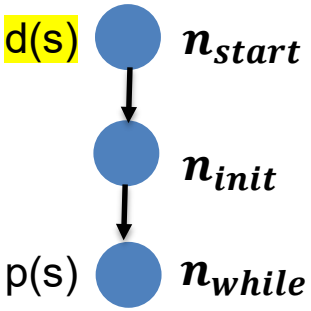
Fünf abzudeckende def-Stellen



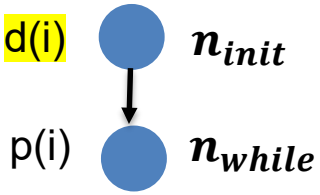
# Beispiel: all-defs-Kriterium

**Testfall 1: “.“**

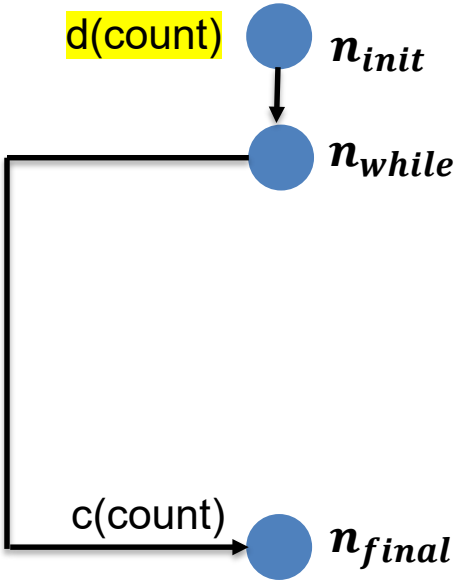
1. def-use-Paar




2. def-use-Paar



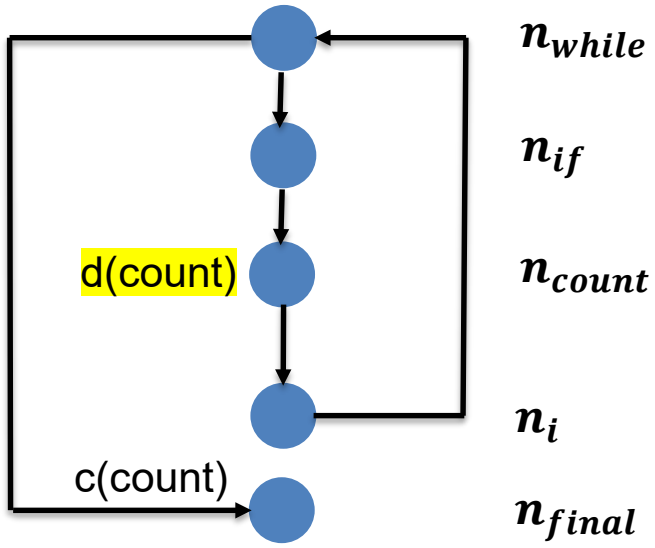

3. def-use-Paar



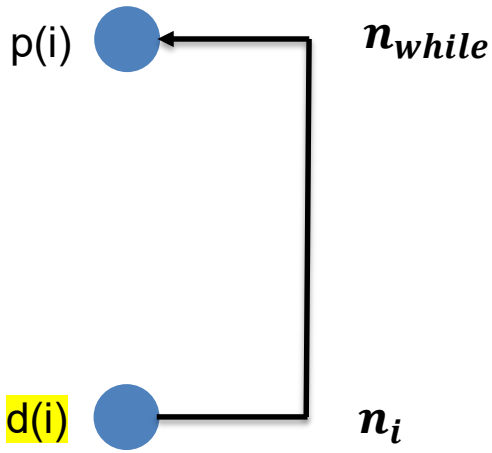
# Beispiel: all-defs-Kriterium

**Testfall 2: “a.“**

4. def-use-Paar




5. def-use-Paar



# Beispiel: all-defs-Kriterium

- Die beiden Testfälle erfüllen zusammen das all-defs-Kriterium.
- Das Kriterium umfasst nicht Zweigüberdeckung (im Beispiel wird der „leere“ else-Zweig wird nicht abgedeckt) und auch nicht Anweisungsüberdeckung!
- Der Fehler wird nicht gefunden.
- Fazit: Auch als Minimalkriterium zu schwach.

## all-p-uses-Kriterium

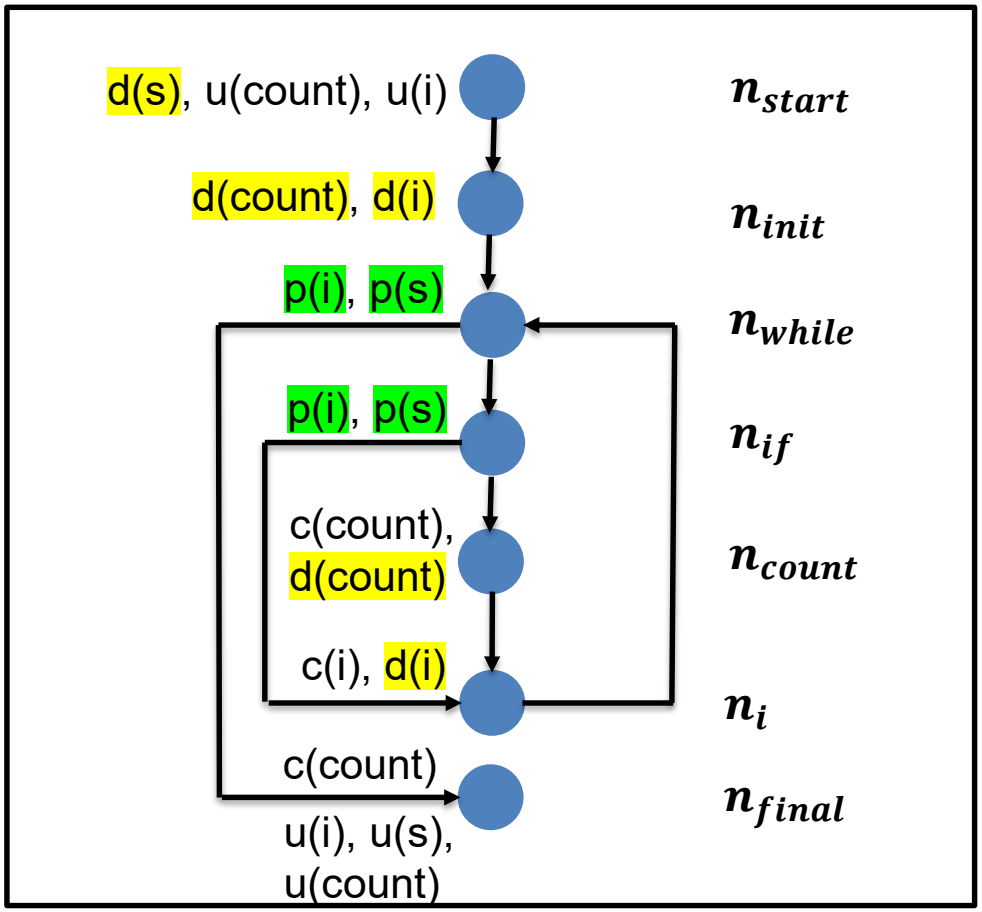
Das all-p-uses-Kriterium fordert, dass für **jede** Definitionsstelle *d(x)* einer Variablen mindestens **ein** definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Verwendungsstellen *p(x)* durch einen Testfall abgedeckt wird.

# Beispiel: all-p-uses-Kriterium

- Es reicht ein Testfall mit dem Ausführungs-pfad:

  $n_{start}, n_{init}, n_{while}, n_{if}, n_i, n_{while}, n_{if}, \dots, n_{while}, n_{final}$

Beispiel: Testfall: **“bb.“**



# Diskussion: all-p-uses-Kriterium

- Entdeckt vor allem Kontrollflussfehler, Berechnungsfehler bleiben hingegen oft unentdeckt.
- Manchmal wird auch die Abdeckung **aller** definitionsfreien Pfade von d(x) zu allen p(x) gefordert, die Schleifen nicht mehrfach durchlaufen müssen (dann ist Zweigüber-deckung enthalten)

## all-p-uses-Kriterium

Das all-c-uses-Kriterium fordert, dass für **jede** Definitionsstelle *d(x)* einer Variablen mindestens **ein** definitionsfreier Pfad zu **allen** (erreichbaren) berechnenden Verwendungsstellen *c(x)* durch einen Testfall abgedeckt wird.

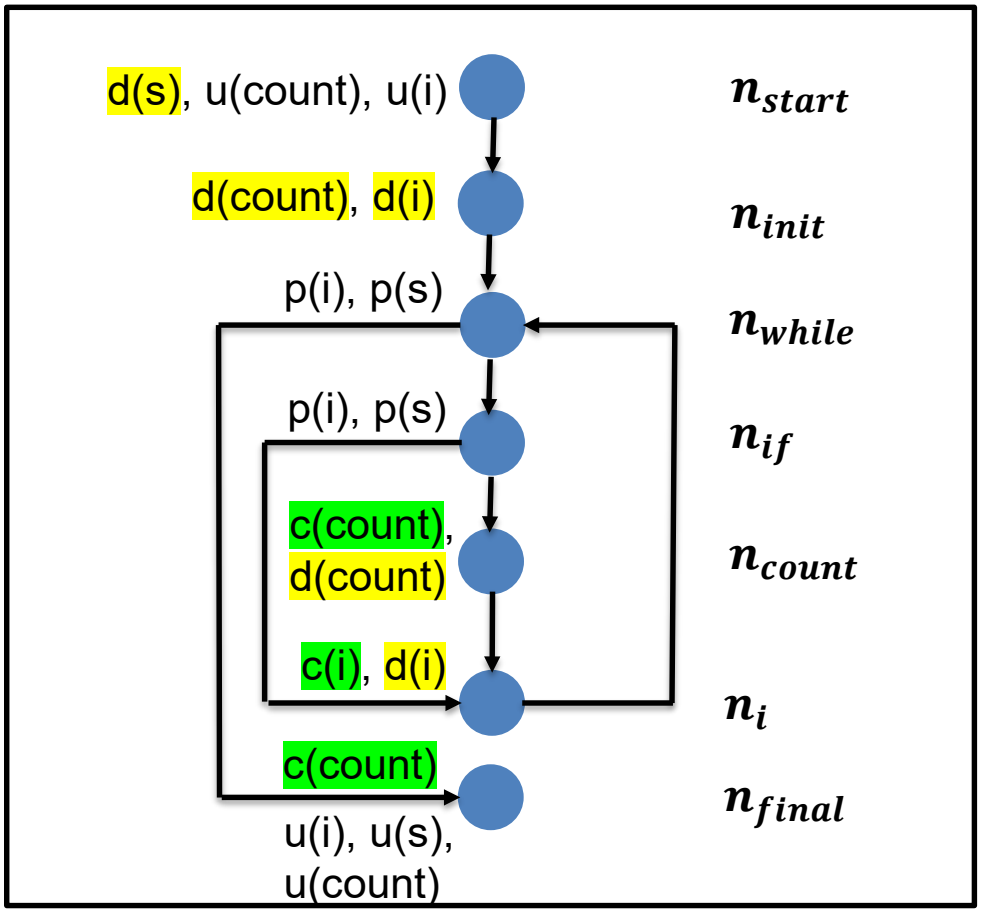
# Beispiel: all-c-uses-Kriterium

Abzudeckende Pfade:

- von $n_{init}$ nach $n_i$
- von $n_{init}$ nach $n_{count}$ und direkter Pfad zu $n_{final}$
- von $n_{count}$ nach $n_{count}$ und direkter Pfad zu $n_{final}$
- von $n_i$ nach $n_i$

Beispiel:

- Testfall 1: “**a.**“
- Testfall 2: “**.**“
- Testfall 3: “**aa.**“



# Diskussion: all-c-uses-Kriterium

- Entdeckt vor allem Berechnungsfehler, Kontrollflussfehler bleiben hingegen oft unentdeckt.
- Vollständige Abdeckung lässt sich aufgrund der von der Ausführung abhängigen Bedingungsauswertung oft nicht erzwingen.

## all-p-uses-some-c-uses-Kriterium

Das all-p-uses-Kriterium fordert, dass für **jede** Definitionsstelle *d(x)* einer Variablen mindestens **ein** definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Verwendungsstellen *p(x)* durch einen Testfall abgedeckt wird. Gibt es keine prädikate Benutzung *p(x)*, so muss wenigstens **ein** Pfad zu **einem** berechnenden Zugriff *c(x)* abgedeckt werden.

# Diskussion: all-p-uses-some-c-uses-Kriterium

- Entdeckt zahlreiche Kontrollfluss- und auch Berechnungsfehler.
- Umfasst all-def- und all-p-uses-Kriterium.
- Für das *countVowels*-Beispiel erfüllt zum Beispiel der einzelne Testfall “**ab.**“ dieses Kriterium.

# Weitere Datenflusskriterien

- all-c-uses-some-p-uses-Kriterium: analog zum all-p-uses-some-c-uses-Kriterium definiert.
- uses-Kriterium: all-p-uses- + all-c-uses-Kriterium (kaum benutzt).

# Diskussion: Datenflussbasierte Testverfahren

- Insgesamt besser geeignet für objektorientierte Programme mit oft einfachem Kontrollfluss aber komplexem Datenfluss.
- Es gibt kaum Werkzeuge, die datenflussbasierte Testverfahren unterstützen. Grund: Datenfluss in vielen Programmiersprachen sehr schwierig zu analysieren (Pointer etc.).

# Empfehlungen: White-Box Testen

- Auswahl von Überdeckungskriterien gemäß der „Kritikalität“ der zu entwickelnden Software.
- Kombination von einem kontrollflussbasierten und einem datenflussbasierten Überdeckungskriterium sinnvoll.
- So lange weitere Testfälle auswählen bis ~90% Abdeckung erreicht ist (100% lässt sich in vielen Fällen wegen Anomalien nicht erreichen).

UNIVERSITÄT SIEGEN

# Objektorientierte Testverfahren

## Zustandsbezogenes Testen

# Motivation

- Prinzipiell lassen sich die bisher betrachten Testverfahren, die sich auf Ebene einzelner Methoden bewegt haben, auch auf komplette objektorientierte Programme anwenden.
- Dabei gibt es allerdings einige Besonderheit, die das Testen einerseits vereinfachen können, andererseits aber auch erschweren.

# Motivation

- Prinzipiell lassen sich die bisher betrachteten Testverfahren, die sich auf Ebene einzelner Methoden bewegt haben, auch auf komplette objektorientierte Programme anwenden bzw. übertragen/erweitern.
- Dabei gibt es allerdings einige Besonderheiten, die das Testen einerseits vereinfachen können, andererseits aber auch erschweren.
- (Hier nur einige grundlegende Überlegungen)

# Testen objektorientierter Programme

Datenkapselung:

- Pro: Konzentration von Zugriffsoperationen auf Daten an einer Stelle erleichtert das Testen.
- Kontra: Erschwerter Zugriff auf interne Zustände von Objekten für Ein-/Ausgaben von Testfällen.

Vererbung:

- Pro: Wiederverwendung bereits getesteten Codes reduziert auch die Menge an neu geschriebenem zu testenden Code.
- Kontra: Ist eine Hauptfehlerquelle und muss umso intensiver getestet werden.

Dynamisches Binden:

- Kontra: Erschwert die Definition sinnvoller Überdeckungskriterien.
- Kontra: Verhalten von Objektmethoden ist (fast) immer zustandsabhängig.

# Ansätze zum Testen objektorientierter Programme

- **„Black-Box“-Test:** wie bisher (Objekte als Eingabeparameter werden gemäß interner Zustände verschiedenen Äquivalenzklassen zugeordnet).
- **„White-Box“-Test:** Kontrollflussgraphen erweitern um
  - Zustandsmodellierung von Objekten.
  - Effekte des dynamischen Bindens.
- **Geerbter Code** wird wie neu geschriebener Code behandelt und immer vollständig im Kontext der erbenden Klasse neu getestet (Variation des Regressionstests).
- **Konsistenzüberprüfungen** zur Beobachtung von Vor-/Nachbedingungen von Methodenaufrufen und Klasseninvarianten ergänzen (z.B. asserts in Java).

# Arten von Klassen

- **Nicht-modale Klassen**: Methoden der Klasse können immer (zu beliebigen Zeitpunkten) aufgerufen werden und der interne Zustand der Objekte spielt dabei keine Rolle.
- **Uni-modale Klassen**: Methoden können nur in einer bestimmten Reihenfolge aufgerufen werden, die aber unabhängig vom internen Zustand der Objekte ist.
- **Quasi-modale Klassen**: Methoden können nur in bestimmten Objektzuständen aufgerufen werden, die Aufrufreihenfolge ist nicht eingeschränkt.
- **Modale Klassen**: Methoden können nur in fest vorgegebenen Reihenfolgen und können in bestimmten Objektzuständen aufgerufen werden.

# Testen verschiedener Arten von Klassen

| | Aufrufbarkeit der Methode unabhängig vom Objektzustand | Aufrufbarkeit der Methode abhängig vom Objektzustand |
|---|---|---|
| Aufrufreihenfolge der Methode ist flexibel | **Nicht-modal:** Methoden isoliert testen | **Quasi-modal:** Methoden isoliert testen für alle Zustandsäqui-valenzklassen |
| Aufrufreihenfolge der Methode ist fest | **Uni-modal:** Alle zulässigen und verbotenen Reihenfolgen testen | **Modal:** Alle zulässigen und verbotenen Reihenfolgen testen für alle Zustands-äquivalenzklassen |

# Checkliste für Klassentests

- Alle Methoden (auch geerbte) mindestens einmal ausführen.
- Alle Methodenparameter und alle nach außen sichtbaren Attribute mit geeigneter Äquivalenzklassenbildung durchgetestet.
- Alle auslösbaren (ausgehenden) Ausnahmen mindestens einmal ausgelöst.
- Alle von aufgerufenen Methoden auslösbaren (eingehenden) Ausnahmen mindestens einmal behandelt (oder durchgereicht).
- Alle identifizierten Objektzustände (auch hier Äquivalenzklassenbildung) beim Testen erreicht.
- Alle zustandsabhängige Methode in jedem Zustand ausgeführt (auch in Zuständen, in denen Aufruf nicht zulässig ist).
- Alle möglichen Zustandsübergänge (mit allen Kombinationen von Bedingungen an den Übergängen) aktiviert.
- Dazu: die üblichen Performanz-, Last-, ... –Tests.

UNIVERSITÄT SIEGEN

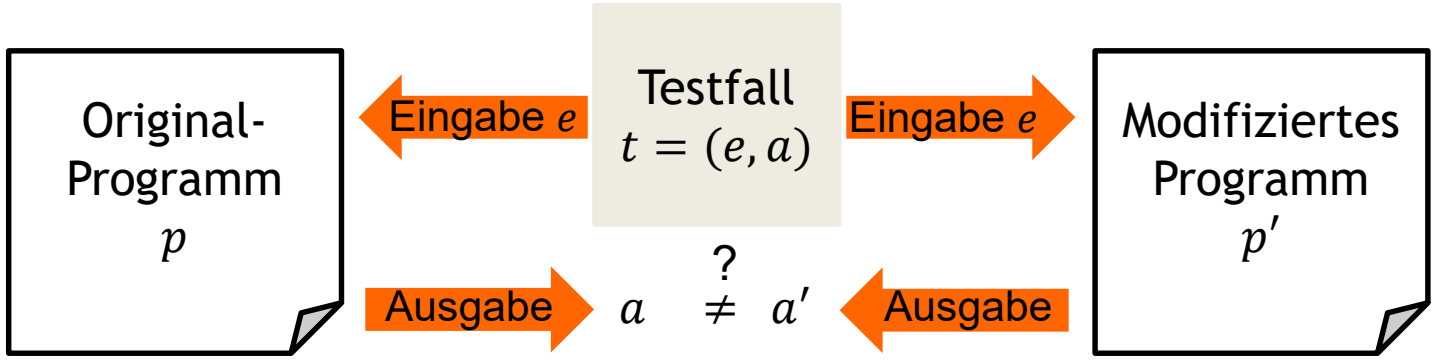
# Mutationsbasierte Testverfahren

## Testen von Testverfahren

# Motivation

- Wir haben bisher zahlreiche Abdeckungsbegriffe für die Steuerung der Auswahl von Testfällen kennengelernt.
- Offen ist dabei nach wie vor die Frage, ob es tatsächlich einen Zusammenhang zwischen dem Grad der Abdeckung gibt, den eine Testsuite hinsichtlich eines bestimmten Kriteriums erfüllt und der Fähigkeit des Testsuite, tatsächliche Fehler in realen Testobjekten zu finden.
- Hilfreich wäre hierfür eine Methodik für die Modifikation von Programmen zur **Simulation echter Fehler**, um Testverfahren testen zu können.

# Fehlerdetektion durch Testfälle




- Testfall $t$ **detektiert** den (simulierten) Programmfehler in $P'$, falls sich die Ausgabe von $P'$ von der Ausgabe des Originalprogramms $P$ für die Testeingabe von $t$ unterscheidet.
- Testsuite $T = \{t_1, t_2, \ldots t_n\}$ **detektiert** den simulierten Programmfehler in $P'$, falls $T$ mindestens einen Testfall enthält der den Programmfehler detektiert.

# Fehlerdetektionsrate

Sei $P$ eine Menge von modifizierten Programmen und $T$ eine Menge von Testfällen. Die Fehlerdetektionsrate von $T$ für $P$ ist:

$$E(T,P) = \frac{|\{\, p' \in P \mid \exists t \in T : t \text{ detektiert den Fehler in } p' \}|}{|P|}$$

# Bewertung von Testansätzen

Die Fehlerdetektionsrate liefert auch ein mögliches Maß für die den Vergleich verschiedener Testansätze.

Seien $T_A$ und $T_B$ die durch die Testansätze $A$ und $B$ für ein Programm $p$ generierten Testfälle:

- $A$ ist **effektiver** als $B$ hinsichtlich einer Menge $P$ für $p$ simulierte Programmfehler, falls $E(P, T_A) > E(P, T_B)$.
- $A$ ist **effizienter** als $B$ hinsichtlich der Testauswahl für Programm $p$, falls $|T_A| > |T_B|$.

# Diskussion: Bewertung von Testansätzen

- Die Aussagekraft des Effektivitätsmaßes steht und fällt mit der Menge $P$ betrachteter Programmmodifikationen zur Fehlersimulation.
- Eine mögliches Verfahren zur automatisierten Generierung einer Menge $P$ für ein (korrektes) Originalprogramm $p$ ist das Mutationstesten.

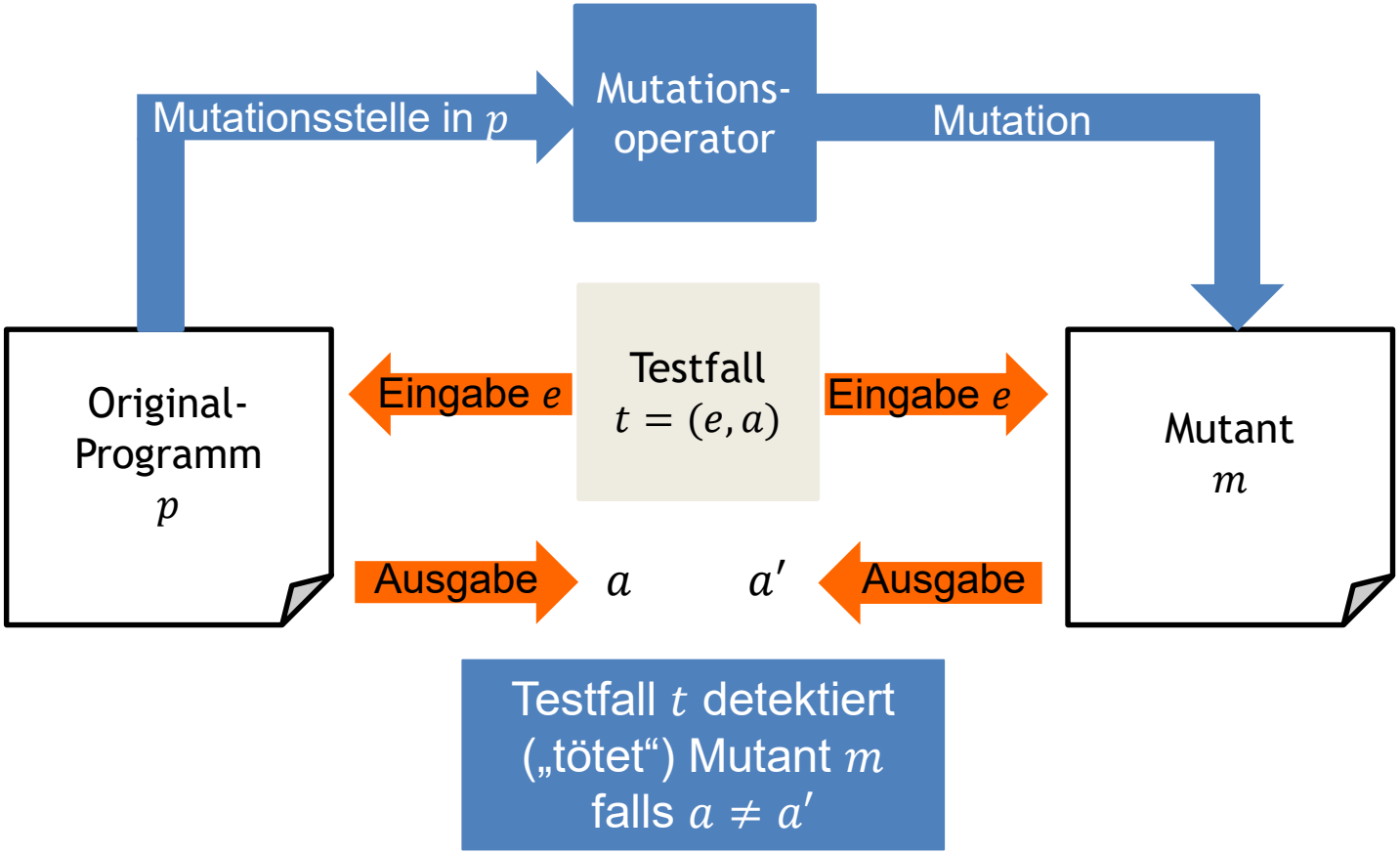
# Hypothesen des Mutationstestens

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

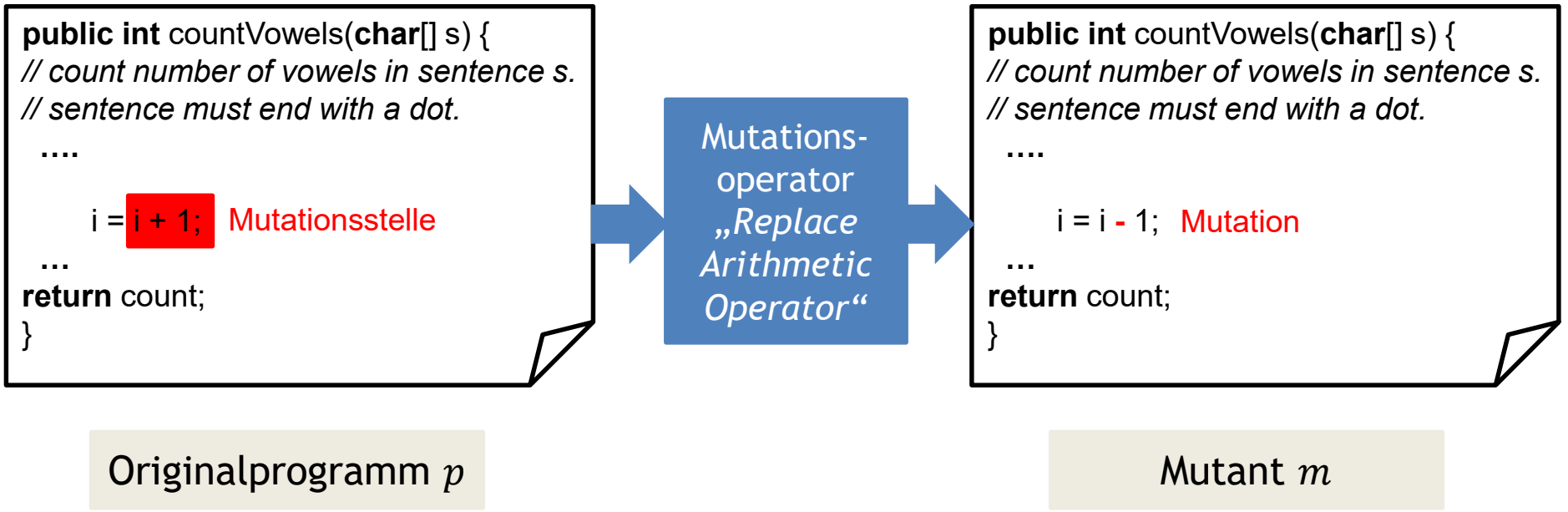
# Schlussfolgerungen für das Mutationstesten

- Typische „echte“ Programmierfehler lassen sich oft auf kleine, sehr einfache lexikalische/syntaktische Modifikationen (Mutationen) eines korrekten Programmes simulieren.
- Solche Programmmutationen lassen sich automatisiert durchführen.

# Mutationstesten



# Beispiel: Generierung von Mutanten




Originalprogramm $p$

Mutant $m$

# Mutationsoperatoren

- Austausch von arithmetischen, logischen, vergleichenden, … Operatoren in Anweisungen und Bedingungen.
- Austausch von Konstanten, Literalen, …
- Austausch von Namen (von Variablen, Methodenaufrufen, …)
- Löschen/Hinzufügen von (Teil-)Ausdrücken, …

# Diskussion: Mutationsoperatoren

- Eine Mutation soll eine einfache und effiziente Operation sein und möglichst wieder ein kompilierbares Programm ergeben.
- Deshalb sollten Mutationen keine größeren Umbauten im Original-Programm durchführen, die sich über mehr als eine Stelle/Zeile hinaus erstrecken.

UNIVERSITÄT SIEGEN

# Anforderungen an mutationsdetektierende Testfälle

Damit ein Testfall $t$ einen Mutanten $m$ eines Originalprogramms $p$ töten kann, muss $t$ die **RIPR**-Kriterien erfüllen:

- **Reachability Condition:** die Ausführung von t auf $m$ erreicht die Mutationsstelle.
- **Infection Condition:** die Ausführung von t auf $m$ führt zu einem fehlerhaften (infizierten) Programmzustand (d.h. eine Variablenwertbelegung, die von der Ausführung von $t$ auf $p$ abweicht).
- **Propagation Condition:** die Ausführung von t auf $m$ propagiert den fehlerhaften Programmzustand weiter bis ans Programmende.
- **Reveal Condition:** die Ausführung von t auf $m$ führt zu einer Programmausgabe, die durch den fehlerhaften Programmzustand von der Ausgabe für die Ausführung von $t$ auf $p$ abweicht.

# Diskussion: RIPR-Kriterien

- Erfüllt ein Testfalll die „Propagation Condition“, dann zwangsläufig auch die „Infection Condition“.
- Erfüllt ein Testfall die „Infection Condition“, dann zwangsläufig auch die „Reachability Condition“
- Erfüllt ein Testfall die „Reval Condition“, dann zwangsläufig auch die „Infection Condition“.

# Diskussion: RIPR-Kriterien

- Die Umkehrungen der Aussagen auf der vorherigen Folie sind hingegen nicht zwangsläufig richtig (siehe folgendes Beispiel).
- Die bisherigen Überlegungen zur Auswahl von Testfällen anhand von Überdeckungskriterien konzentrierten sich fast ausnahmslos auf die „Reachability Condition“ .
- (Datenflussbasierte Kriterien verfolgen mit Abstrichen noch abgeschwächte Formen von „Infection“ und „Propagation“).

# (Sinnfreies) Beispielprogramm

```
int p(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y-x; r = y+x; */
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

Mutations-operator *„Replace Arithmetic Operator“*

```
int m(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y+x;
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

# Beispiel: RIPR-Kriterien

Testfall $t = (x = 0, y = 1, z = 0)$

- Die Eingabewerte erfüllen die erste if-Bedingung, sodass der else-Zweig mit der Mutationsstelle nicht betreten wird.
- Der Testfall erfüllt somit die „Reachability Condition“ nicht.

```
int m(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y+x;
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

# Beispiel: RIPR-Kriterien

Testfall $t = (x = 0, y = 0, z = 0)$

- Die Mutationsstelle wird erreicht.
- An der Mutationsstelle wird *r* allerdings derselbe Wert (0+0=0) wie im Originalprogramm (0-0=0) zugewiesen.
- Der Testfall erfüllt somit die „Infection Condition“ nicht.

```
int m(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y+x;
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

# Beispiel: RIPR-Kriterien

Testfall $t = (x = 1, y = 0, z = 0)$

- Die Mutationsstelle wird erreicht.
- An der Mutationsstelle wird *r* ein anderer Wert (0+1=1) als im Originalprogramm (0-1=-1) zugewiesen (Infektion).
- Der abweichende Wert von *r* wird allerdings im zweiten else-Zweig wieder ungenutzt überschrieben.
- Der Testfall erfüllt somit die „Propagation Condition“ nicht.

```
int m(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y+x;
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

# Beispiel: RIPR-Kriterien

Testfall $t = (x = 1, y = 1, z = 1)$

- Die Mutationsstelle wird erreicht.
- Der Programmzustand wird infiziert.
- Die Infektion wird durch den zweiten if-Zweig über den ebenfalls abweichenden Wert von *a* an Variable *r* propagiert.
- Der abweichende Wert von Variable *r* ist schließlich Programmausgabe.
- Der Testfall erfüllt somit alle RIPR-Kriterien.

```
int m(int x, int y, int z) {
  int r;
  if (x < y) {
    r = x+1;
  }
  else {
    r = y+x;
  };
  int a = r+x+y;
  if (x <= y) {
    r = z*a;
  } else {
    r = x+y;
  };
return r;
}
```

# Diskussion: Mutationsdetektion

- Die **Effektivität** einer Testsuite $T$ lässt sich nun hinsichtlich einer Menge $M$ von Mutanten eines Originalprogramms $p$ als Mutationsdetektionsrate entsprechend bestimmen.
- Das vorherige Beispiel zeigt, dass die Anforderungen an die Effektivität von Testfällen durch Mutationstesten sehr viel höher sein können als bei den zuvor betrachteten Abdeckungskriterien.

# Bewertung von Testsuiten

Wie schon beim abdeckungsbasierten Testen kann Mutationstesten als „begleitende“ Methode zur Bewertung (und Verbesserung) zuvor selektierter Testsuiten $T$ genutzt werden:

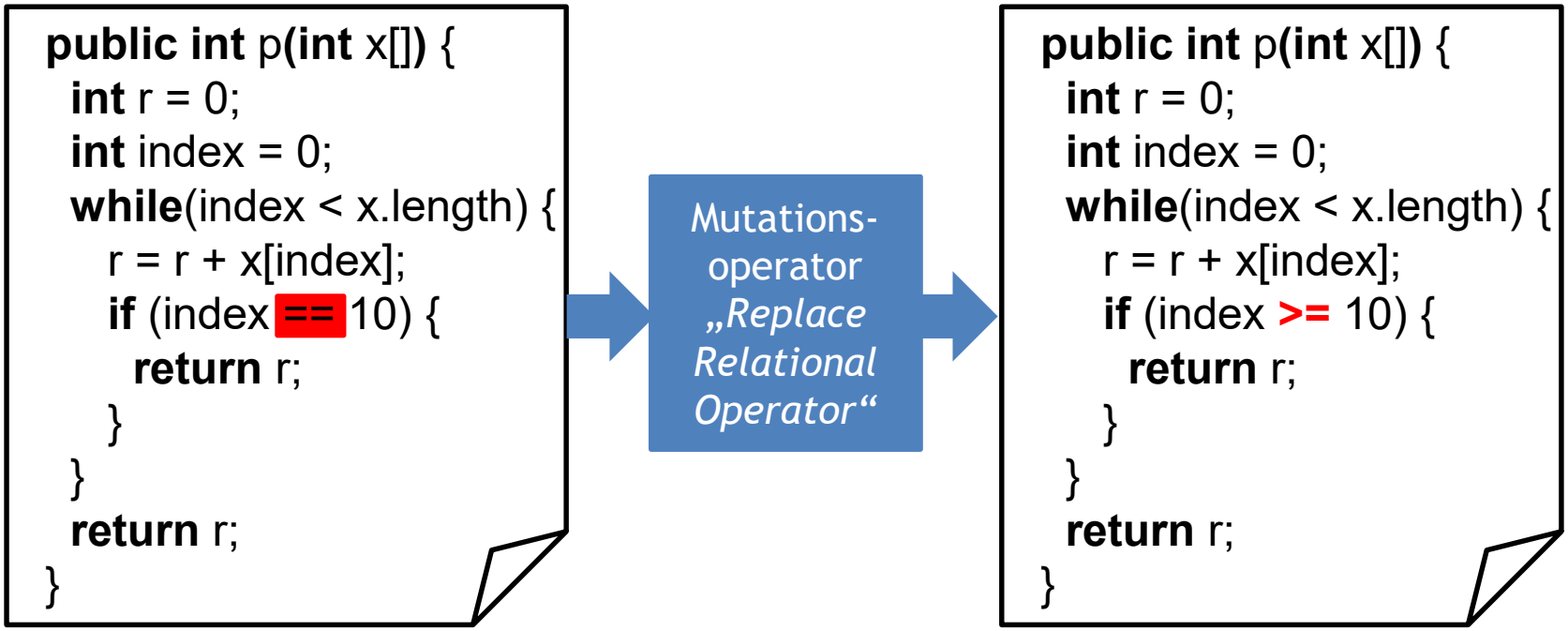
- Identifikation fehlender Testfälle: für jeden Mutant sollte mindestens ein detektierender Testfall in $T$ existieren.
- Eliminierung redundanter Testfälle: detektiert ein Testfall $t$ in $T$ die gleichen (oder weniger) Mutanten als ein Testfall $t'$, kann $t$ aus $T$ entfernt werden.

# Äquivalente Mutanten

Ein wesentliches Problem beim mutationsbasierten Testen ist die Gefahr **äquivalenter Mutanten**:

- Ein Mutant $m$ ist äquivalent zum Originalprogramm $p$, falls das Ausgabeverhalten von $m$ und $p$ für alle Programmeingaben gleich ist.
- Für äquivalente Mutanten existieren somit keine mutationsdetektierenden Testfälle.
- Einfaches Beispiel: die Mutationsstelle liegt in unerreichbaren Programmteilen.

# Beispiel: Äquivalente Mutanten




Originalprogramm $p$

Mutant $m$

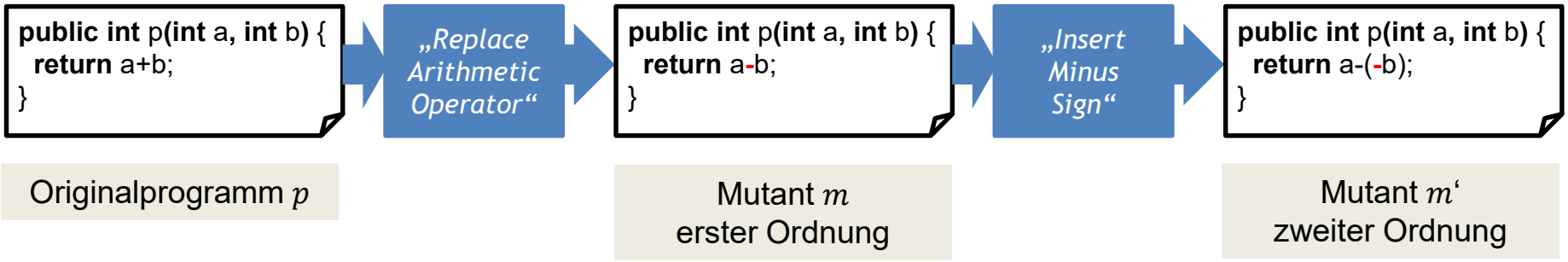
# Diskussion: Äquivalente Mutanten

- Die Generierung äquivalenter Mutanten sollte also möglichst vermieden werden, um die Bestimmung der Mutationsdetektionsrate einer Testsuite $T$ nicht zu verfälschen.
- Allerdings ist die Frage, ob zwei Programme (hier: $m$ und $p$) äquivalentes Ein-/Ausgabeverhalten haben, **unentscheidbar** (eine weitere Folge des Satzes von Rice).
- Somit ist eine Mutationsdetektionsrate von 1 meistens nicht erreichbar.
- (Zur Erinnerung: das Problem besteht auch schon bei abdeckungsbasierten Testverfahren).

# Higher Order Mutants

- Idee: Anwendung **mehrerer** Mutationen auf ein Originalprogramm, um potentiell noch „schwierigere“ (interagierende) Fehler zu simulieren.
- Der Begriff ist etwas irreführend: das Ziel ist nicht, Mutationsoperatoren zu mutieren…

# Beispiel: Higher Order Mutants




- Mutant $m'$ ist äquivalent zu $p$.
- Mögliches Problem von Higher Order Mutation: Äquivalente Mutanten durch gegenseitiges „Neutralisieren“ der Mutationen.

# Diskussion: Mutationstesten

- Potentiell wichtige zukünftige Methode zur experimentellen Bewertung und Vergleich der Effektivität von Testansätzen.
- In aktuellen Arbeiten wurde empirisch nachgewiesen, dass Mutationen tatsächlich echte Fehler adäquat simulieren können (oder zu mindestens ein besseres Effektivitätsmaß sind, als Abdeckungskriterien).
- Allerdings ist die gezielte Selektion mutationserkennender Testfälle noch schwieriger als das darin enthaltende Problem der abdeckungsverbessernden Testfälle.
- Bisherige Werkzeugunterstützung ist noch ausbaufähig.

UNIVERSITÄT SIEGEN

# Weitere Testverfahren und Testmanagement

# Werkzeuge zur Performanzanalyse

Werkzeuge zum Messen von Laufzeitverhalten und Speicherplatzverbrauch von (Teil-)Programmen haben zum Ziel:

- oft durchlaufene, ineffiziente Programmteile zu identifizieren und
- Speicherlecks zu finden.

# Instrumentierung

- Objektcode für die zu untersuchte Software wird vor Ausführung „instrumentiert“, d.h. um zusätzliche Anweisungen ergänzt.
- Die zusätzlichen Anweisungen erzeugen während der Ausführung statistische Daten über Laufzeitverhalten, Speicherplatzverbrauch, ...
- Die zusätzlichen Anweisungen sollen möglichst wenig Einfluss auf die „normale“ Programmausführung haben.

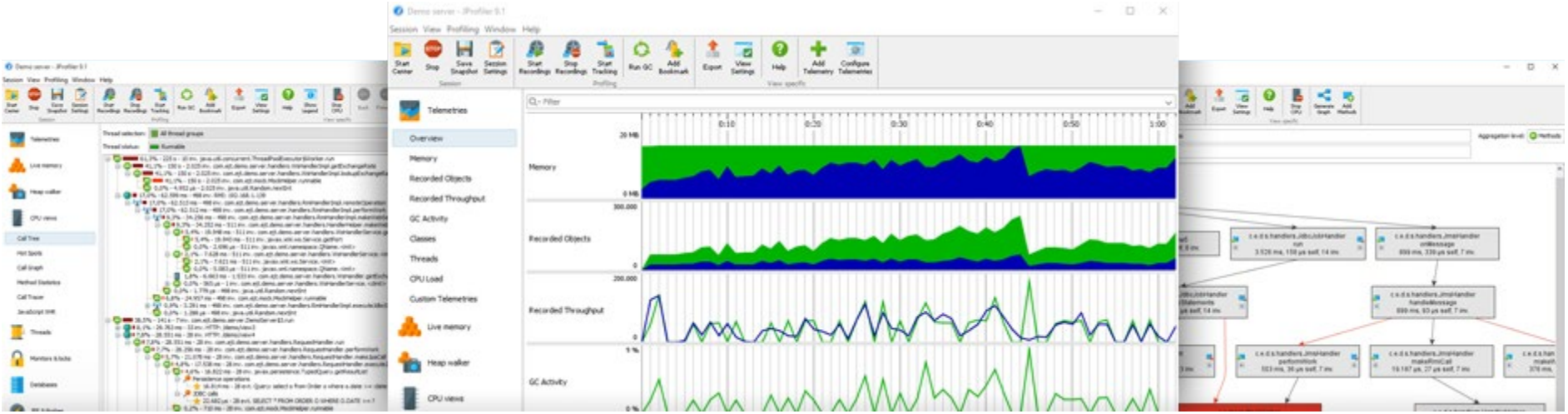
# Analyse des Laufzeitverhaltens

- Wie oft wird jede Operation aufgerufen (oder Quellcodezeile durchlaufen)?
- Welche Operation ruft wie oft welche andere Operation (**callees**) auf oder von welchen Operationen (**callers**) wird ein Programm wie oft gerufen?
- Wieviel Prozent der Gesamtlaufzeit wird mit der Ausführung einer bestimmten Operation verbracht?

# Analyse des Speicherplatzverhaltens

- Welche Operationen fordern wieviel Speicherplatz an (geben ihn frei)?
- Wo wird/bleibt unnötiger Speicherplatz allokiert (memory leaks)?
- Durch die Verwendung ineffizienter Datenstrukturen könne solche Probleme auch in Sprachen mit automatischer Speicherverwaltung (z.B. Garbage Collector in Java) entstehen.

# Beispiel: JProfiler

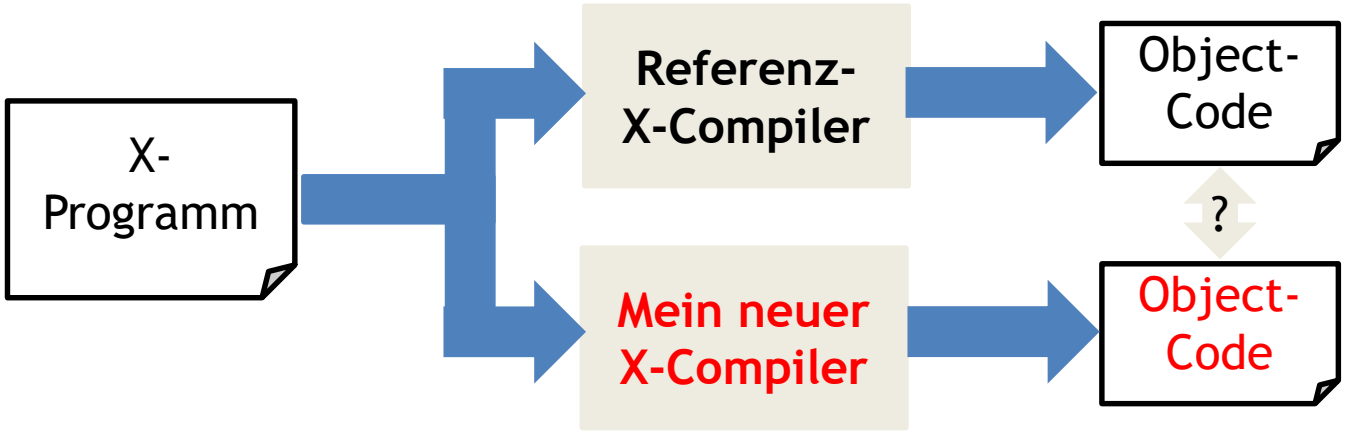



JProfiler

# Testen von Programmen mit komplexen Eingaben

- Bisher sind wir davon ausgegangen, dass zu selektierende Eingabedaten von Testobjekten atomare Datentypen (Integer, Float, …) oder einfach strukturierte, zusammengesetzte Datentypen (Strings, Arrays, …) aufweisen.
- In der Praxis besteht insbesondere beim Black-Box Test auf Komponenten- oder Systemebene das Problem, dass Eingabeschnittstellen sehr komplexe Eingabedaten erwarten.

# Beispiel: Testen von Compilern




Eine Neu-Implementierung eines Compiler für die Programmiersprache „X“ soll getestet werden.

# Beispiel: Testen von Compilern

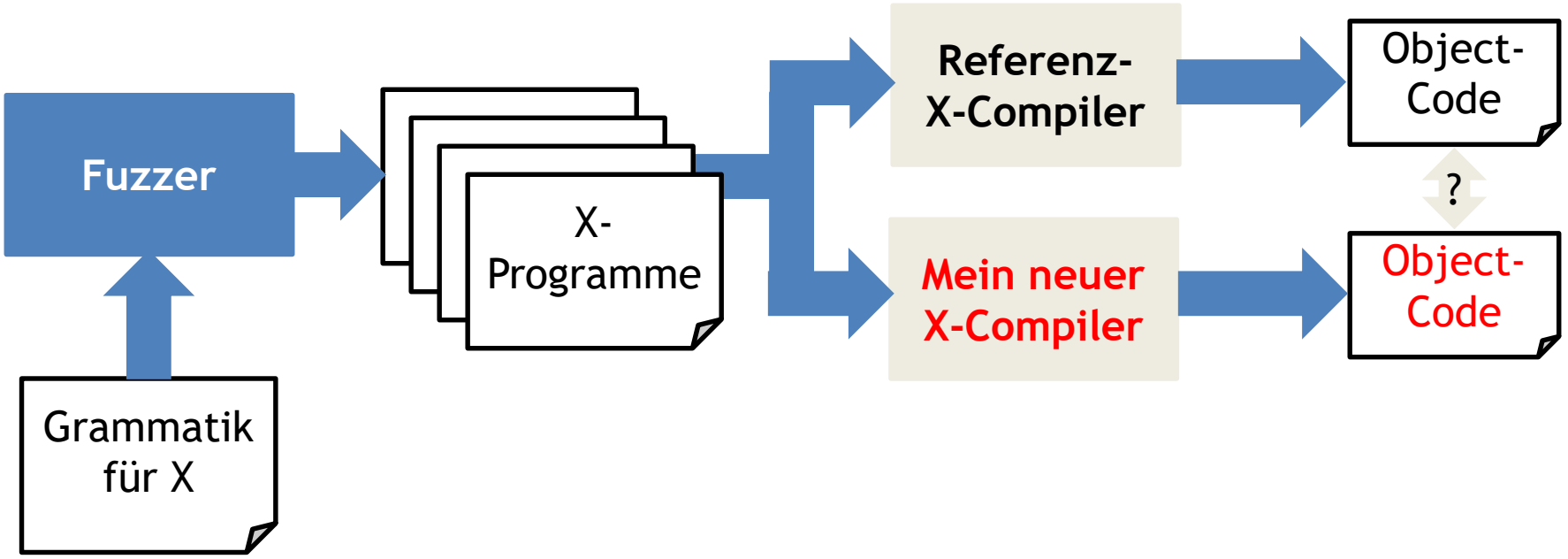
- Annahme: Es existiert bereits ein bewährter „Referenz-Compiler“ für „X“-Programme.
- Wende den Referenz-Compiler und den neuen Compiler auf das gleiche Eingabe-Programm an und vergleiche den generierten Object-Code.
- Der Vergleich kann beispielsweise durch die in diesem Kapitel beschriebenen Testverfahren erfolgen, indem Testfälle selektiert, auf beide Programme angewendet und die Ausgaben miteinander verglichen werden (**Back-to-Back-Testen**).
- Offene Frage: Woher kommen die Beispiel-X-Programme?

# Fuzzing

Ein **Fuzzer** ist ein Generator für Zufallseingaben für Programmschnittstellen mit komplexen Eingabeformaten.

- Grundlage: Grammatik der Eingabesprache (z.B. in BNF-Notation).
- Fuzzer „durchläuft“ zufällig die Grammatik-Regeln und generiert beliebige Sätze der Grammatik.

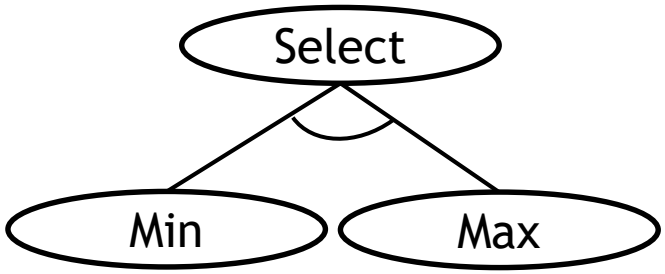
# Fuzzing


Fuzzer
Grammatik für X
X-Programme
Referenz-X-Compiler
Mein neuer X-Compiler
Object-Code
?
Object-Code

# Diskussion: Fuzzing

- Die generierten Eingabeprogramme müssen **nicht zwangsläufig kompilierbar** sein (das kann sogar erwünscht sein, um auch gleichzeitig Parser, Type-Checker etc. mitzutesten).
- Häufig eingesetzt für die Generierung **massenhafter** Eingabedateien sowie **ungewöhnlicher** Eingaben für Robustheitstests.
- Zuletzt sehr erfolgreich zur Generierung von Test-Webseiten eingesetzt, um **Security-Probleme** (z.B. SQL-Injection-Attacken) in Webbrowsern aufzudecken.

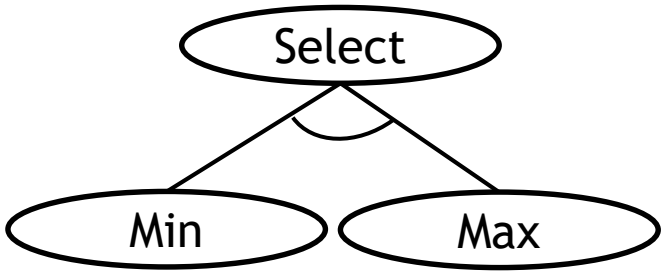
# Testen konfigurierbarer Programme




```
public int select(int x, int y) {
  if(x #ifdef Min < #elif Max > #endif y) {
    return x;
  } else {
    return y;
}
```

- Sehr einfache SPL mit zwei alternativen Features *Min* und *Max*.
- Auswahl des kleineren oder größeren Wertes zweier Eingabewerte *x* und *y*.

# Testen konfigurierbarer Programme




```
public int select(int x, int y) {
  if(x #ifdef Min < #elif Max > #endif y) {
    return x;
  } else {
    return y;
}
```

Testsuite-Selektion für Zweigabdeckung:

- Testfall 1: Eingabe (x=1, y=0), Ausgabe = ?
- Testfall 2: Eingabe (x=0, y=1), Ausgabe = ?

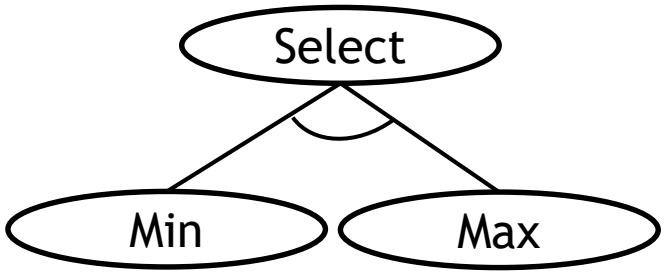
Offene Fragen:

- Welche Zweige decken die Testfälle jeweils ab?
- Welche Ausgaben werden für die Testfälle jeweils erwartet?

# Diskussion: Testen konfigurierbarer Programme

- Die Antwort auf beide Fragen lautet: das hängt von der betrachteten Programmkonfiguration ab.
- Idee: Erweiterung von Testfällen um **Presence Conditions** zur Beschreibung der Menge von Konfigurationen, für die der Testfall **gültig** ist.

# Abdeckung konfigurierbarer Programme




```
public int select(int x, int y) {
  if(x #ifdef Min < #elif Max > #endif y) {
    return x;
  } else {
    return y;
}
```

Testsuite-Selektion für Zweigabdeckung:

- Testfall 1: Eingabe (x=1, y=0), Ausgabe = 0, Presence Condition = Min
- Testfall 2: Eingabe (x=1, y=0), Ausgabe = 1, Presence Condition = Max
- Testfall 3: Eingabe (x=0, y=1), Ausgabe = 0, Presence Condition = Min
- Testfall 4: Eingabe (x=0, y=1), Ausgabe = 1, Presence Condition = Max

# Diskussion: Abdeckung konfigurierbarer Programme

- Es werden vier Testfälle für Zweigabdeckung benötigt, da das Programm (genauer besehen) *2 Varianten x 2 Zweige = 4 Zweige* hat.
- Techniken zur familienbasierten Testsuite-Selektion nutzen die **variabilitätskodierte Darstellung** zur automatisierten Ableitung von Presence Conditions für selektierte Testfälle (= Konjunktion der Presence Conditions entlang der Programmpfade, die der Testfall durchläuft).

# Bestandteile des Testmanagements

- **Testplanung:** es werden die zum Einsatz kommenden Methoden und Werkzeuge festgelegt.
- **Testspezifikation:** die Testfälle werden entweder manuell oder durch Werkzeuge generiert festgelegt.
- **Testautomatisierung:** die festgelegten Testfälle werden (automatisch ausführbar) implementiert.
- **Testausführung:** die ausgewählten Testfälle werden (durch Werkzeuge automatisiert) ausgeführt.
- **Testprotokollierung:** Testergebnisse werden protokolliert sowie erreichte Testüberdeckung, ggf. Auswahl weiterer Testfälle notwendig.
- **Testauswertung**: Bericht über Reifegrad des Testobjektes, Testaufwand, erreichter Überdeckungsgrad…

# Rollen im Testmanagement

- **Testmanager** (Leiter): Testplanung, Auswahl von Testwerkzeugen; vertritt „Testinteressen“ gegenüber Projektmanagern.
- **Testdesigner** (Analyst): erstellt Testspezifikationen und ermittelt Testdaten.
- **Testautomatisierer**: automatisierte Ausführung der spezifizierten Testfälle durch Testwerkzeuge
- **Testadministrator**: stellt Testumgebung mit ausgewählten Testwerkzeugen zur Verfügung.
- **Tester**: Testausführung, -protokollierung und -auswertung.

# Fehlermanagement

Es müssen alle Informationen erfasst werden, die für das Reproduzieren eines Fehlers notwendig sind:

- **Status**: Bearbeitungsfortschritt der Meldung (Neu, Offen, Analyse, Abgewiesen, Korrektur, Test, Erledigt).
- **Klasse**: Klassifizierung der Schwere des Problems (Menschenleben in Gefahr, Systemabsturz mit Datenverlust, ... , Schönheitsfehler).
- **Priorität**: Festlegung der Dringlichkeit, mit der Fehler behoben werden muss.
- **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.

# Weitere Testwerkzeuge (Auswahl)

- **Testtreiber**: Passend zu Schnittstellen von Testobjekten werden Testtreiber bzw. Testrahmen („Testbetten“) zur Verfügung gestellt, die die Aufrufe von Tests mit Übergabe von Eingabewerten, Auswertung der Ergebnisse etc. abwickeln.
- **Simulatoren**: Bilden möglichst realitätsnah Produktionsumgebung nach (mit Simulation von anderen Systemen, Hardware, ... ).
- **Testroboter** (Capture & Replay-Werkzeug): Zeichnet interaktive Benutzung einer Bedienungsoberfläche auf und kann Dialog wieder abspielen (solange sich Oberfläche nicht zu stark verändert).
- **Komparatoren**: Vergleichen erwartete mit tatsächlichen Ergebnissen, filtern unwesentliche Details, können ggf. auch Zustand von Benutzeroberflächen prüfen
- **Überdeckungsanalyse**: Für gewählte Überdeckungsmetrik wird Buch darüber geführt, wieviel Prozent Überdeckung erreicht ist, welche Programmausschnitte noch nicht überdeckt sind etc.

# Prüfungsstoff

- Grundproblematik von (dynamischen) Analyse-Techniken kennen.
- Schwierigkeiten des systematischen Testens von Software verstehen.
- Grundprinzipien des Funktions-, Kontrollfluss-, Datenfluss- und Mutationstestens kennen.
- Minimalstandards (und Werkzeuge) für den Test von Software kennen.

# Literatur


- P. Liggesmeyer: ***Software-Qualität: Testen, Analysieren und Verifizieren von Software***, Spektrum Akademischer Verlag (2009).
- A. Spillner, T. Linz: ***Basiswissen Softwaretest***, dpunkt.verlag (2012; 5. Auflage).