Software Qualität

- Einführung
- Software Fehler
- Konstruktive Qualitätssicherung
- Software Test
- Statische Analyse

Literatur zu Software Test

- Andreas Spillner, Tilo Linz: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard (iSQI-Reihe), dpunkt.verlag

- Dirk W. Hoffmann: Software-Qualität, 2 Auflage, Springer Vieweg

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests

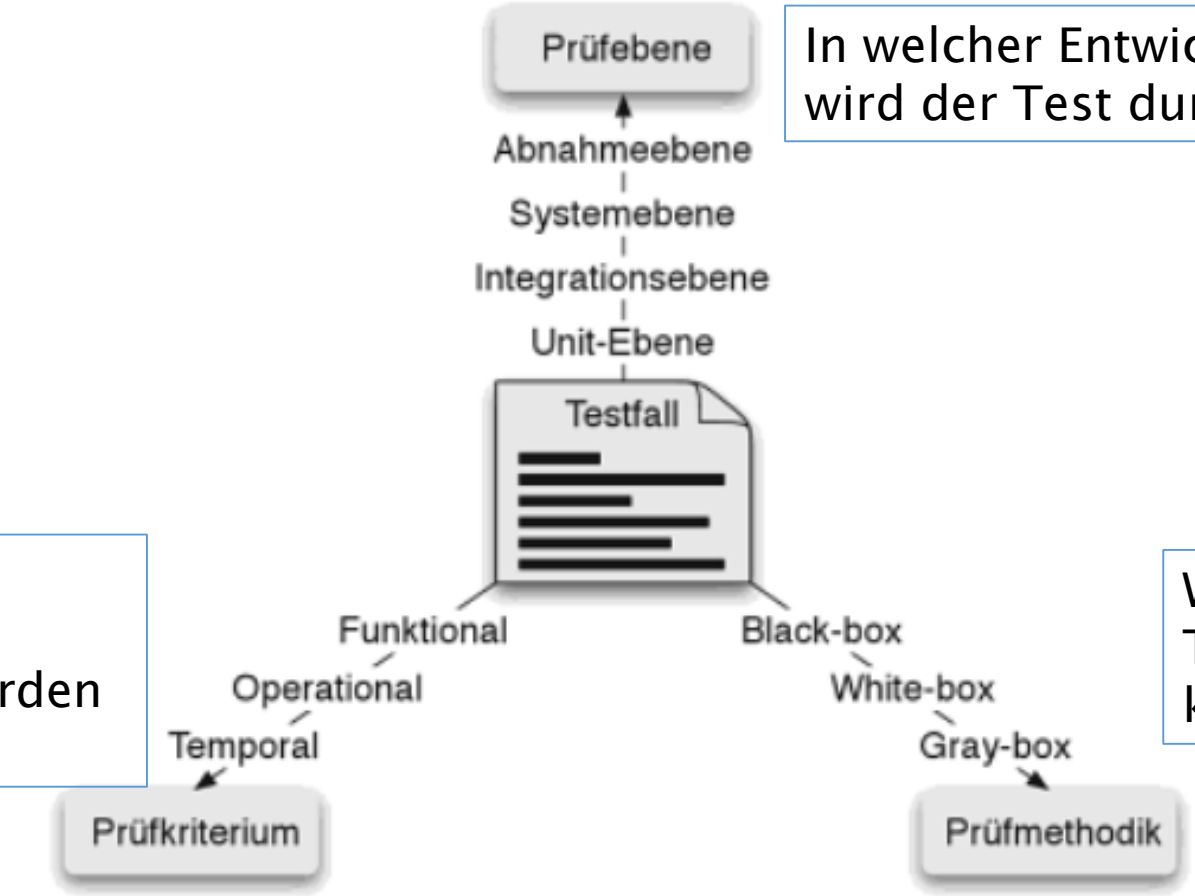
Motivation

- Siehe die Folien zu Beginn der Vorlesung
- Siehe http://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen
- https://jaxenter.de/top-10-der-software-katastrophen-181?utm_source=nl&utm_medium=email

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Testklassifikation



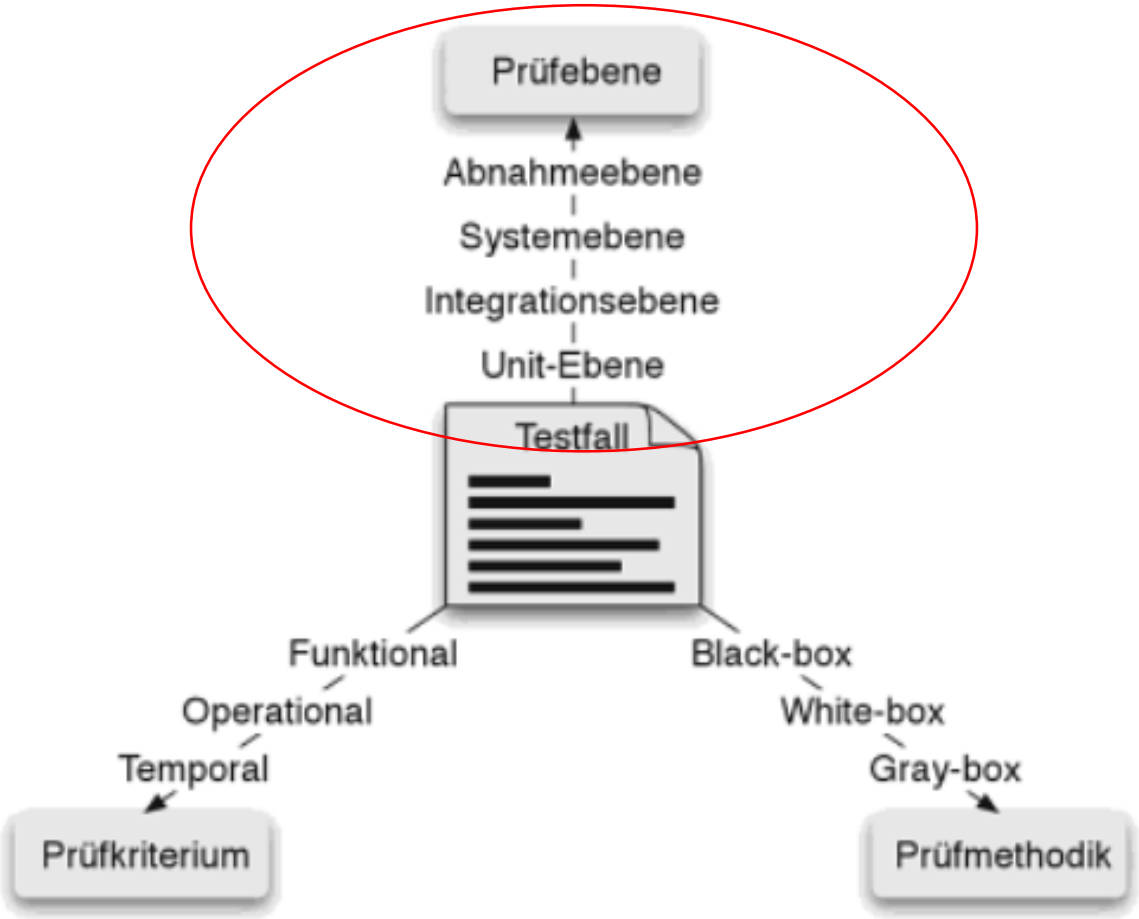
In welcher Entwicklungsphase wird der Test durchgeführt?

Welche inhaltlichen Aspekte werden geprüft?

Wie werden die Testfälle konstruiert?

Merkmalsräume der Testklassifikation

Testklassifikation

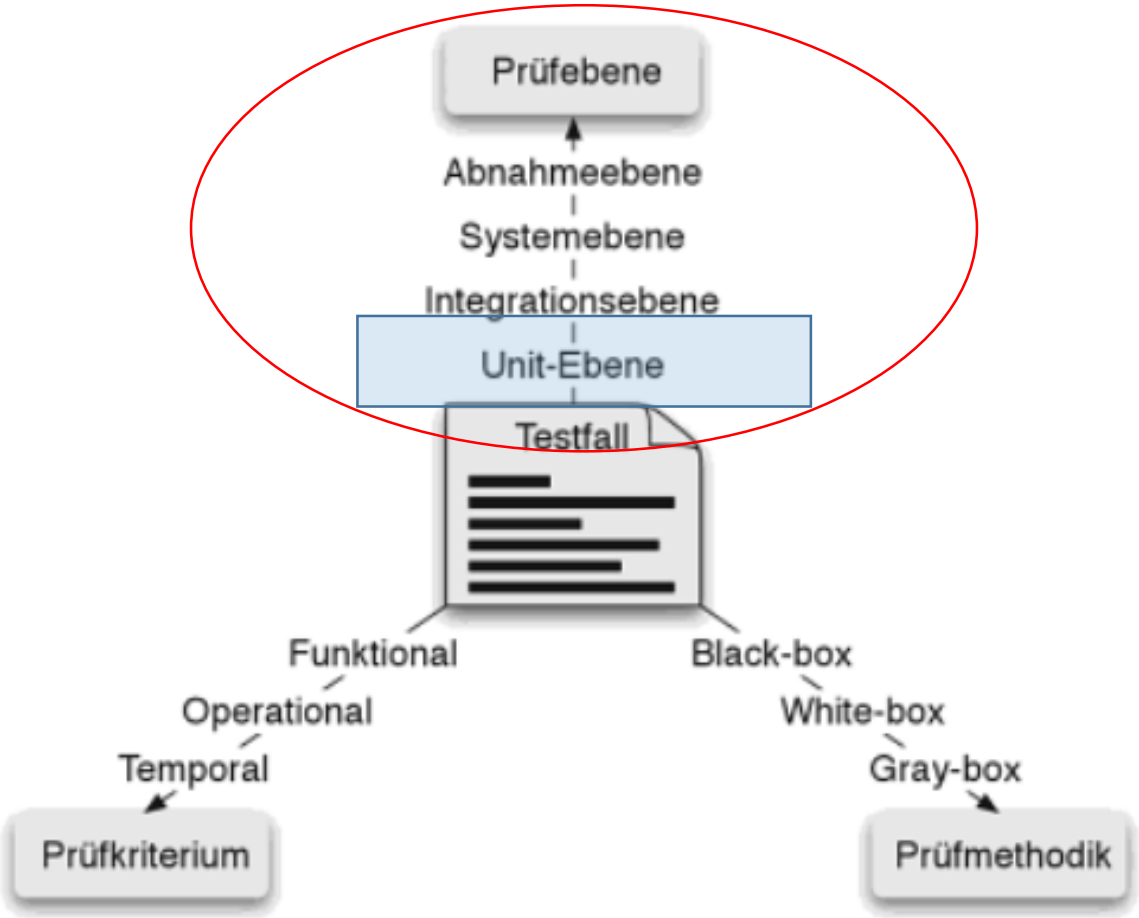


Merkmalsräume der Testklassifikation

Prüfebenen

- **Unit Tests:** Test von atomaren Einheiten, die groß genug sind, um eigenständig getestet zu werden.
- **Integrationstests:** einzelne Einheiten werden zu größeren Komponenten zusammengeführt ➔ Test, ob das Zusammenspiel funktioniert.
- **Systemtest:** Test des Systems als Ganzes auf Einhaltung der im Pflichtenheft festgelegten Eigenschaften.
- **Abnahmetest:** Ist ein Systemtest in der Umgebung und der Verantwortung des Auftraggebers.

Unit Tests



Merkmalsräume der Testklassifikation

Unit Tests

- Sie testen einzelne Einheiten isoliert.
- Der Test ist in erster Linie ein Defekt Test.
- Einheiten sind typischerweise:
  - Einzelne Methoden
  - Klassen mit mehreren Attributen und Methoden.
  - Zusammengesetzte Komponenten mit definierten Schnittstellen.

Unit Tests

- Die Testfälle sollen zeigen, dass die Komponente tut, was sie soll.
- Wenn Defekte enthalten sind, sollen sie aufgezeigt werden.

Zwei Arten von Unit Test Cases

- Normale Programmausführung: Die Komponente tut, was sie soll.
- Ungewöhnliche Eingaben, falsche Eingaben etc: Die Komponente kann damit umgehen, ohne zu crashen.

Automatisierte Unit Tests

- **Ziel:** Möglichst viele Modultests automatisieren.

- **Durchführung:** Verwendung eines Testautomatisierungs Framework (z.B. JUnit) um Tests zu schreiben und durchzuführen.

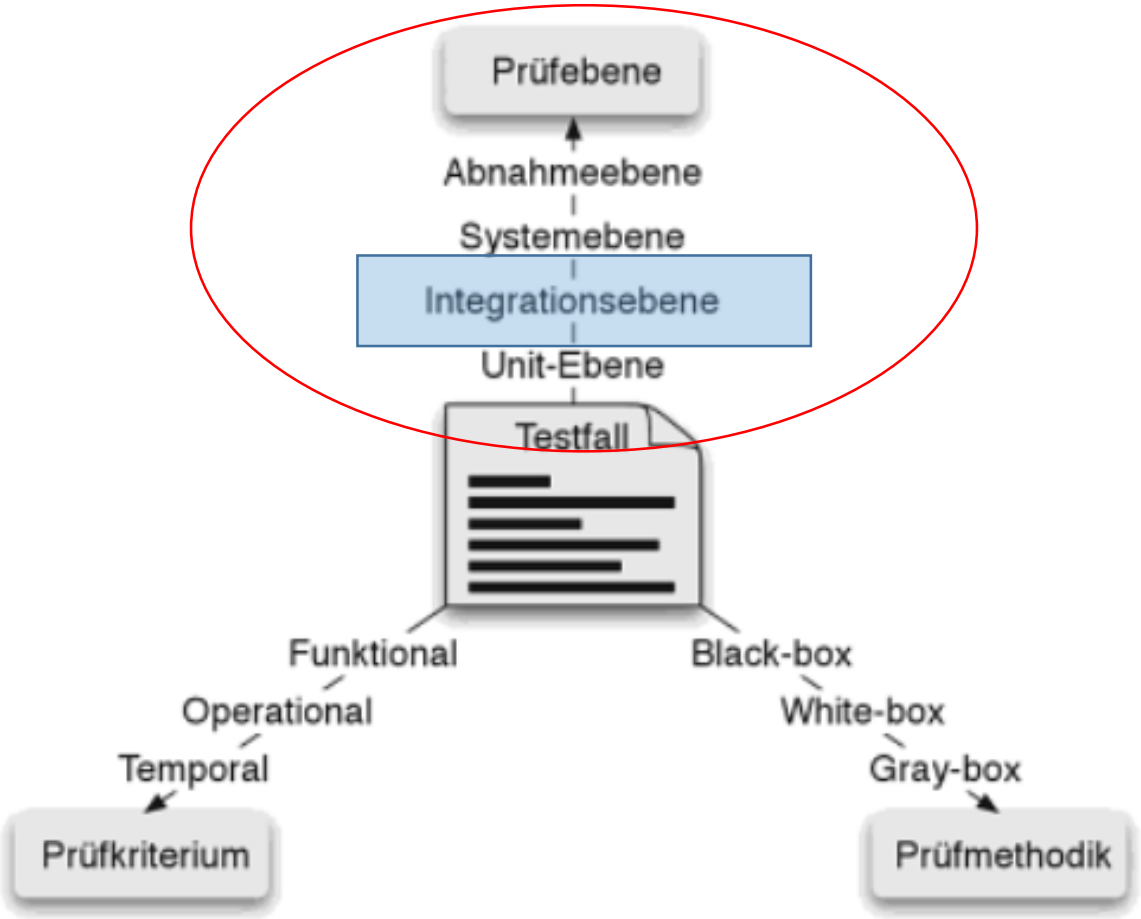
➔Ermöglicht, bei jeder Änderung **ALLE** Tests laufen zu lassen und das Ergebnis graphisch anzuzeigen.

Modultests

Komponenten:

- **Aufbau:**
  Sie definieren die Testfälle mit Eingaben und erwarteten Ergebnissen.
- **Aufruf**
  Sie rufen die zu testenden Objekte oder Methoden auf.
- **Auswertung**
  Vergleich des wirklichen mit dem erwarteten Ergebnis.

Integrationstests



Merkmalsräume der Testklassifikation

Integrationstests

- Nächsthöhere Abstraktionsebene gegenüber Unit Test.
- Wird eingesetzt, wenn einzelne Programmmodule zu größeren Software Komponenten zusammengesetzt werden
- Stellt sicher, dass die Komposition der separat getesteten Komponenten ein funktionsfähiges System ergibt.

Integrationsstrategien

- **Big Bang Integration**
- **Struktur Orientierte Integration**
  - Bottom-Up
  - Top-Down
  - Outside-in
  - Inside-Out
- **Funktionsorientierte Integration**
  - Termingetrieben
  - Risikogetrieben
  - Testgetrieben
  - Anwendungsgetrieben

Integrationsstrategien in der Praxis

Big Bang Integration
Entwicklung sämtlicher Module, anschließend Integration auf einen Schlag.

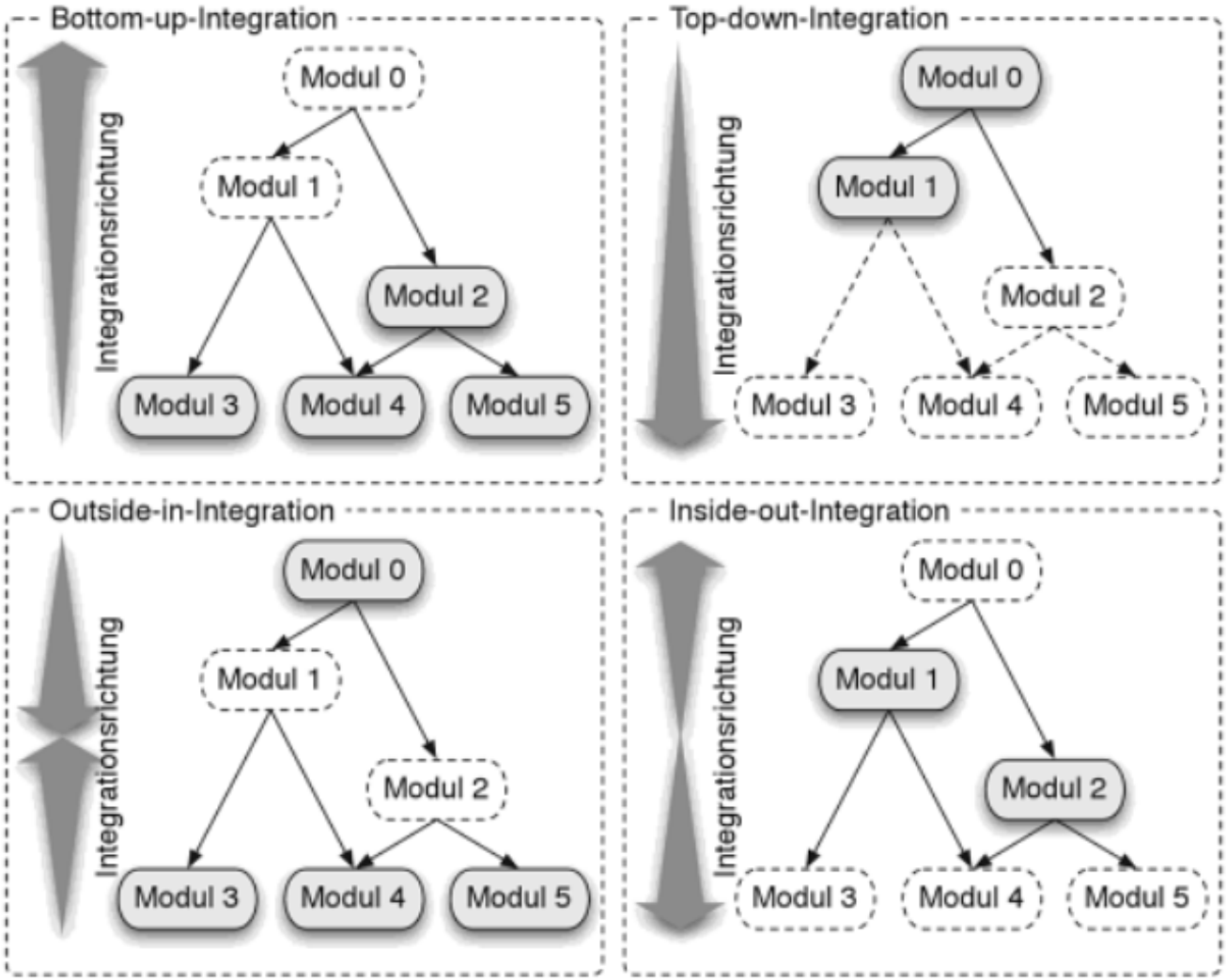
- **Nachteile**
  - Beginn erst wenn alle Module fertig sind
  - Gleichzeitige Integration aller Komponenten führt zu schwieriger Fehlersuche.
- **Vorteil:**
  - Testtreiber und Mocks sind nicht nötig.

Integrationsstrategien in der Praxis

Strukturorientierte Integration
Inkrementelle Integration der Module zum Gesamtsystem. Reihenfolge der Integration richtet sich nach den Abhängigkeiten der Module

- **Bottom Up** – Ausgangpunkt: Basiskomponenten, Verwendung von Testtreibern
- **Top Down** – Ausgangspunkt: Module der höchsten Schicht, Verwendung von Stubs
- **Outside-In** – Integration von beiden Seiten nach innen
- **Inside Out** – Integration von innen nach außen

Integrationsstrategien im Vergleich

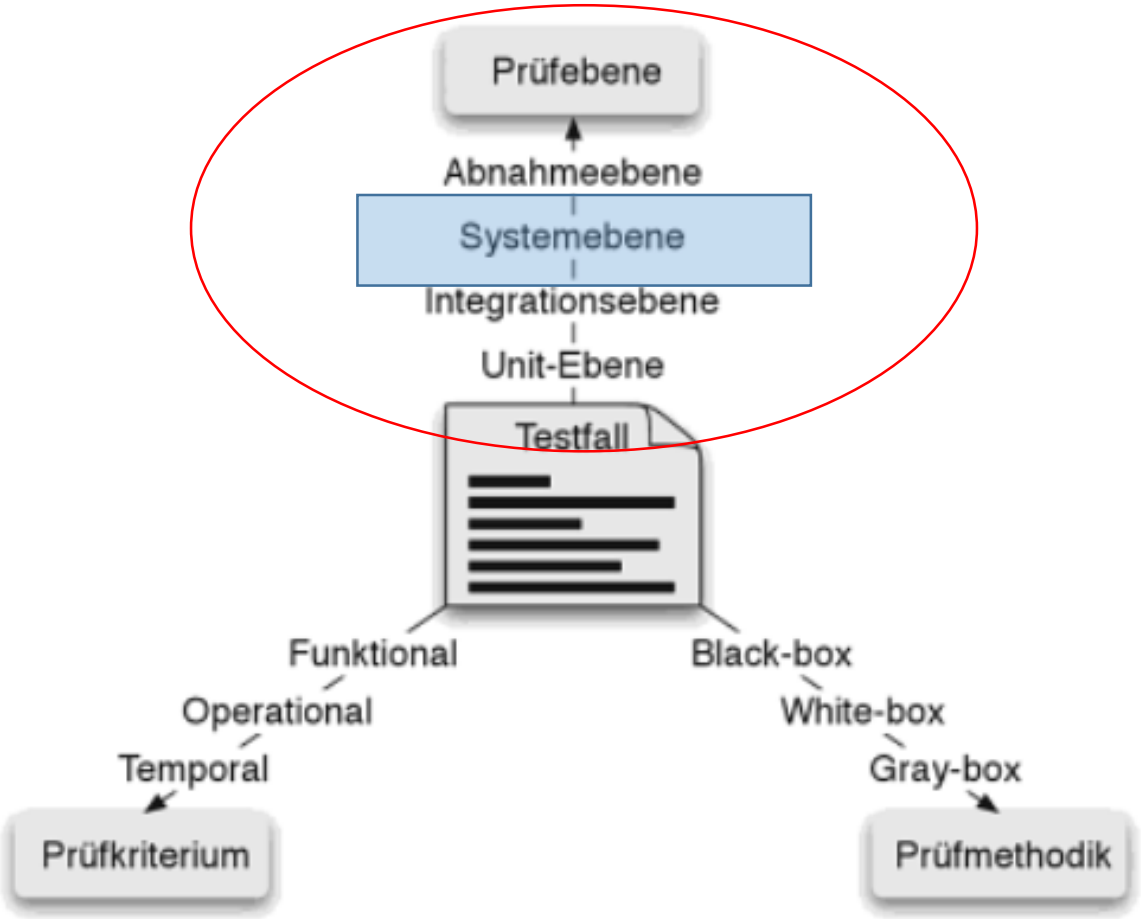


Integrationsstrategien in der Praxis

Funktionsorientierte Integration
Integration anhand funktionaler oder operationaler Kriterien

- Termingetriebene Integration – entsprechend der Verfügbarkeit
- Risikogetriebene Integration – riskanteste als erstes
- Testgetriebene Integration – Integration für bestimmte Testfälle
- Anwendungsgetriebene Integration – Integration für bestimmte Usecases

Systemtests



Merkmalsräume der Testklassifikation

Systemtest

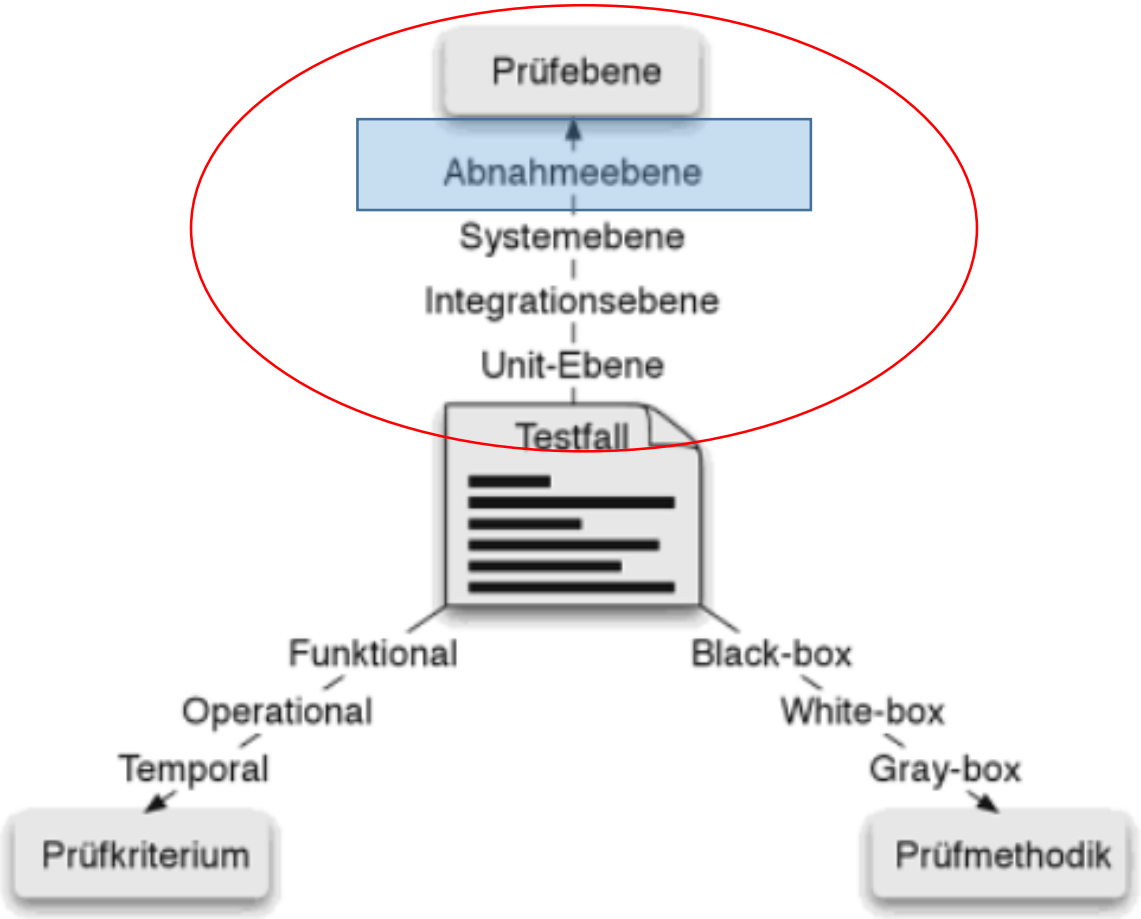
- Start, sobald alle Komponenten erfolgreich integriert sind.
- Nahezu ausschließlich aus funktionaler Sicht.

- Oftmals genauso aufwändig wie Unit und Integrationstest gemeinsam.

Erschwerende Faktoren

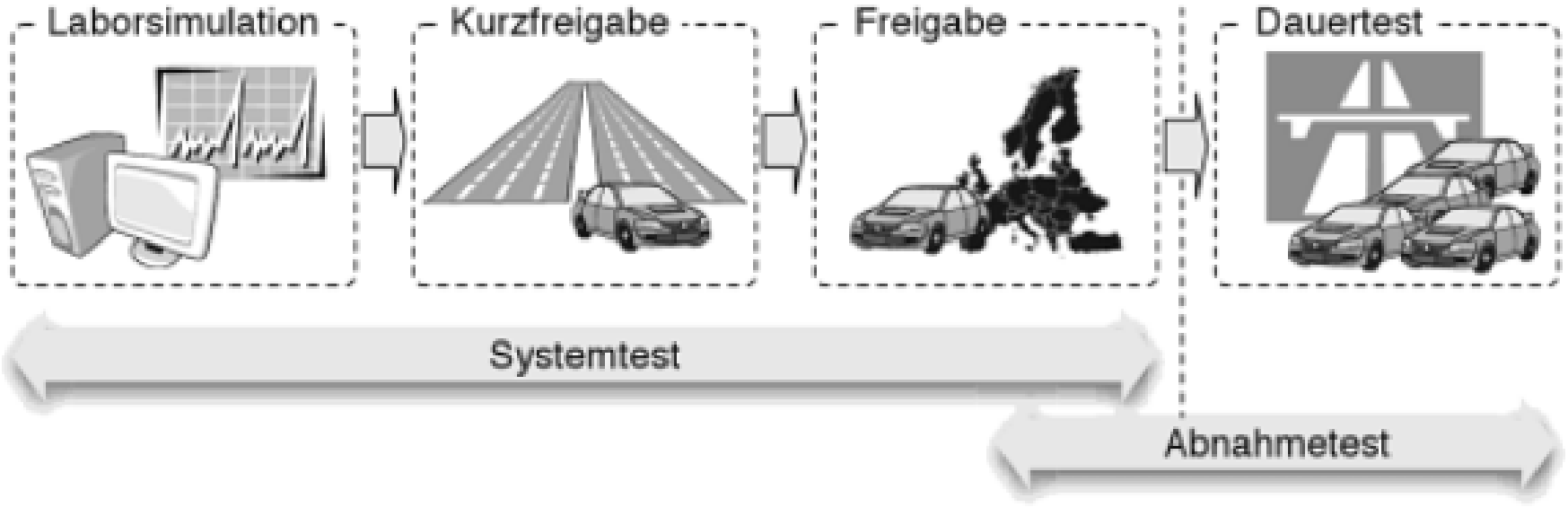
- Unvorhergesehene Fehler
- Unklare Anforderungen
- Testumgebung
- Eingeschränkte Debug Möglichkeiten
- Eingeschränkte Handlungsfähigkeit

Integrationstests



Merkmalsräume der Testklassifikation

Systemtest - Abnahmetest



Typische Phasen des System- und Abnahmetests eines KFZ-Steuergeräts.

Abnahmetests - Individualentwicklung

Abnahmetests sind ähnlich dem Systemtest

Unterschiede:

- Abnahmetest unter Federführung des Auftraggebers
- Abnahmetest findet in der realen Einsatzumgebung des Kunden statt. Durchführung mit authentischen Daten.

Abnahmetests sind juristisch relevant.

Empfehlenswert: Kunden bereits in die Systemtests einzubinden.

➔Kunde ist früh informiert

➔Teilabnahmen sind möglich

Abnahmetests – Produkt für Massenmarkt

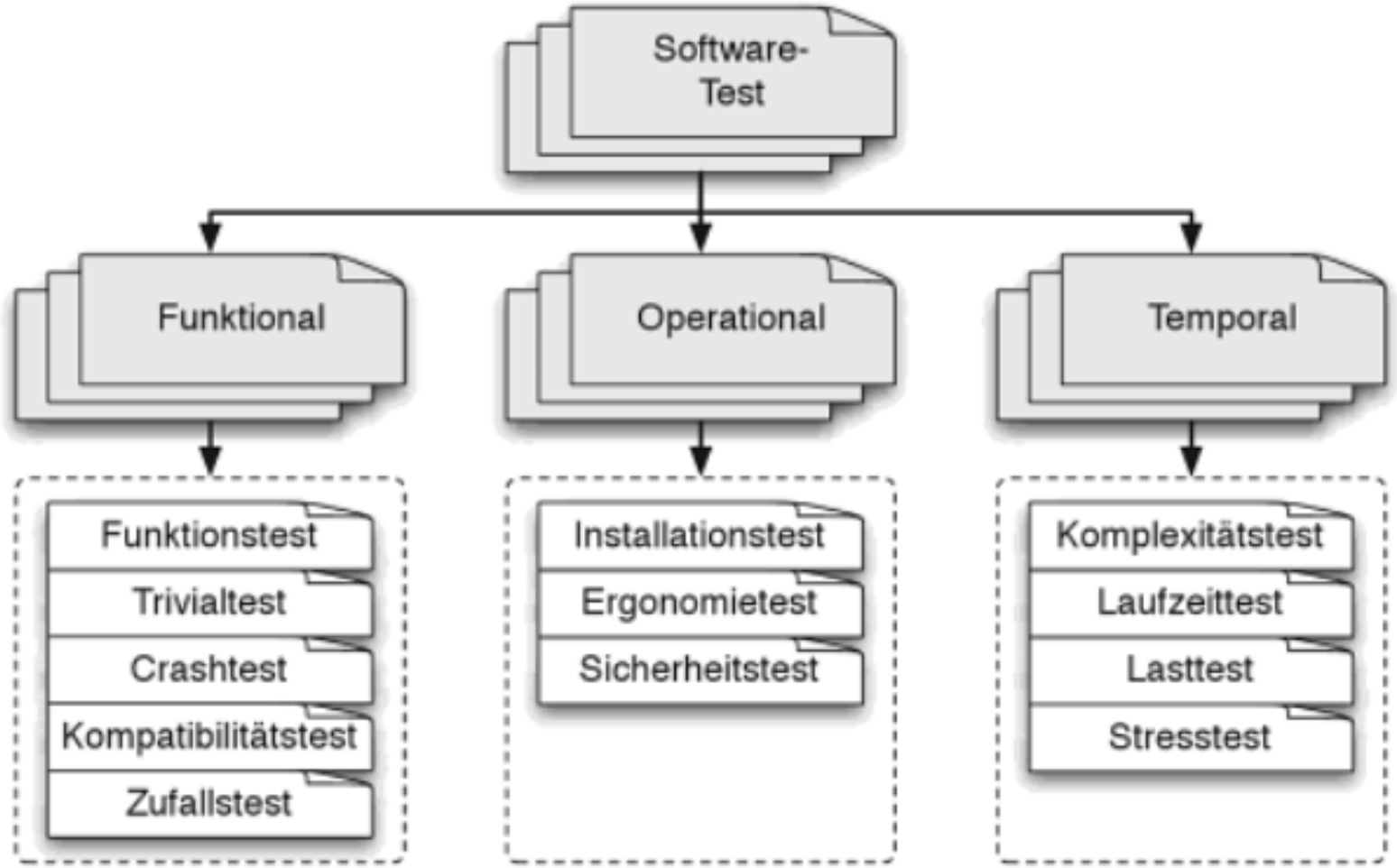
Anstelle von Abnahmetests: Feldtests

- **Alpha Tests**
  In der Anwendungsumgebung des Herstellers. Durchführung durch ausgewählte Anwender.
- **Beta Tests**
  In der Umgebung des Kunden.
  Häufig inkrementell durchgeführt mit zunehmend größerem Nutzerkreis.

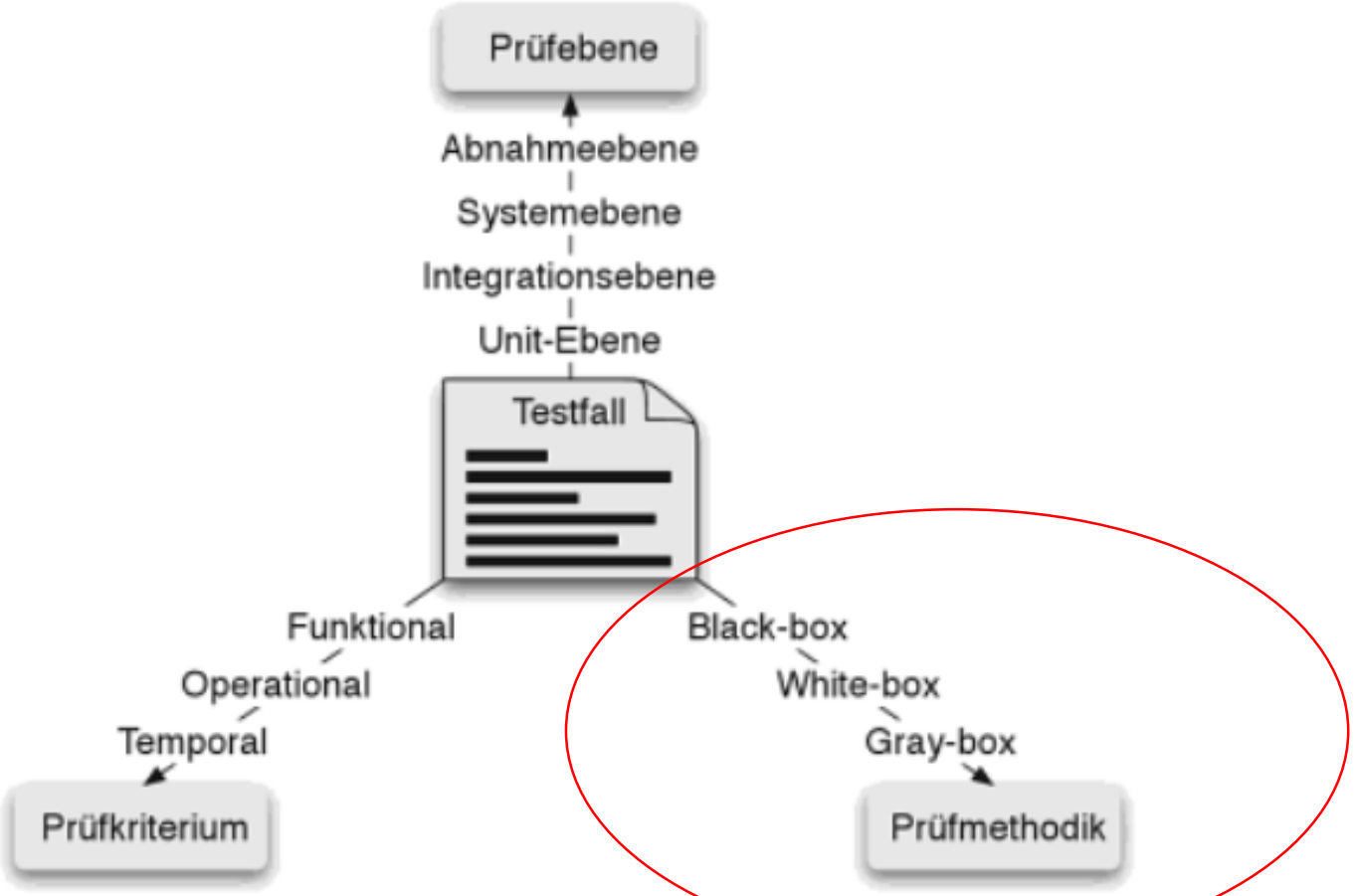
Testklassifikation

Merkmalsräume der Testklassifikation

Prüfkriterien



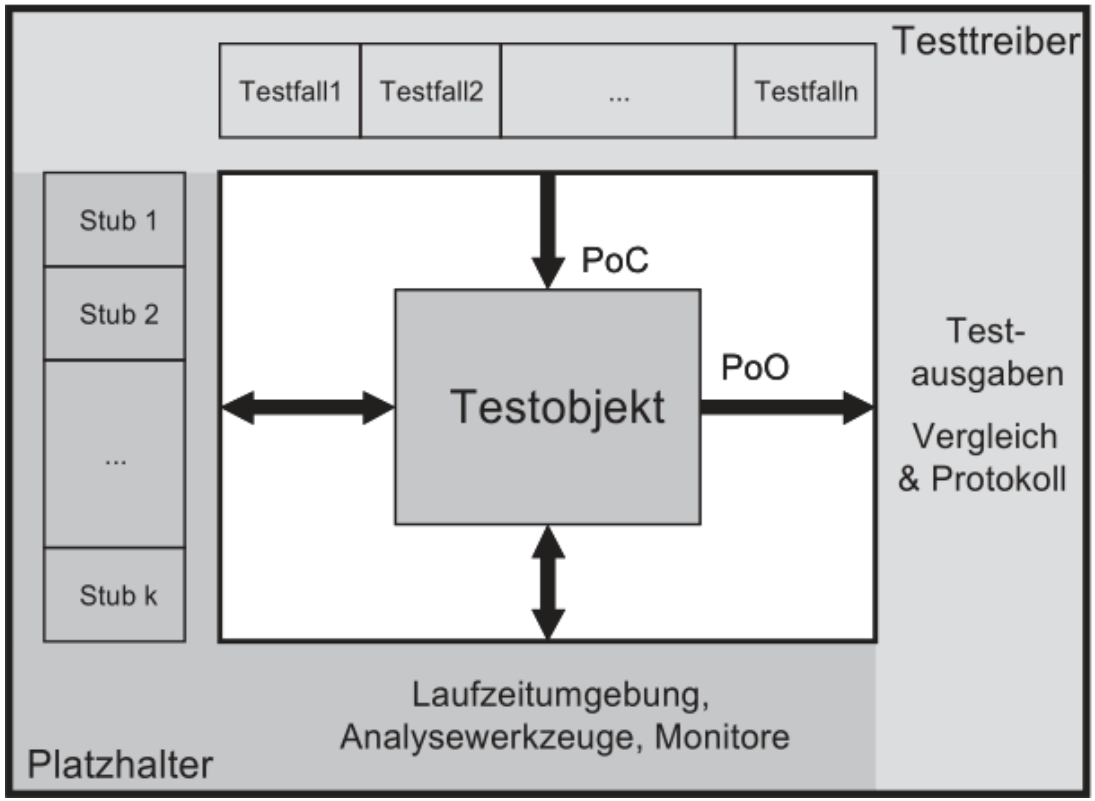
Testklassifikation



Merkmalsräume der Testklassifikation

Testrahmen

Für einen Test muss ein ablauffähiges Programm vorliegen. Für Unit und Integrationstests wird das Testobjekt in einen Testrahmen eingebettet.



PoO:
Point of Oberservation

PoC:
Point of Control

Erstellen von Testfällen

Ziel: Systematisches Vorgehen um mit möglichst wenig Aufwand möglichst viele Anforderungen zu überprüfen bzw. Fehler zu finden.

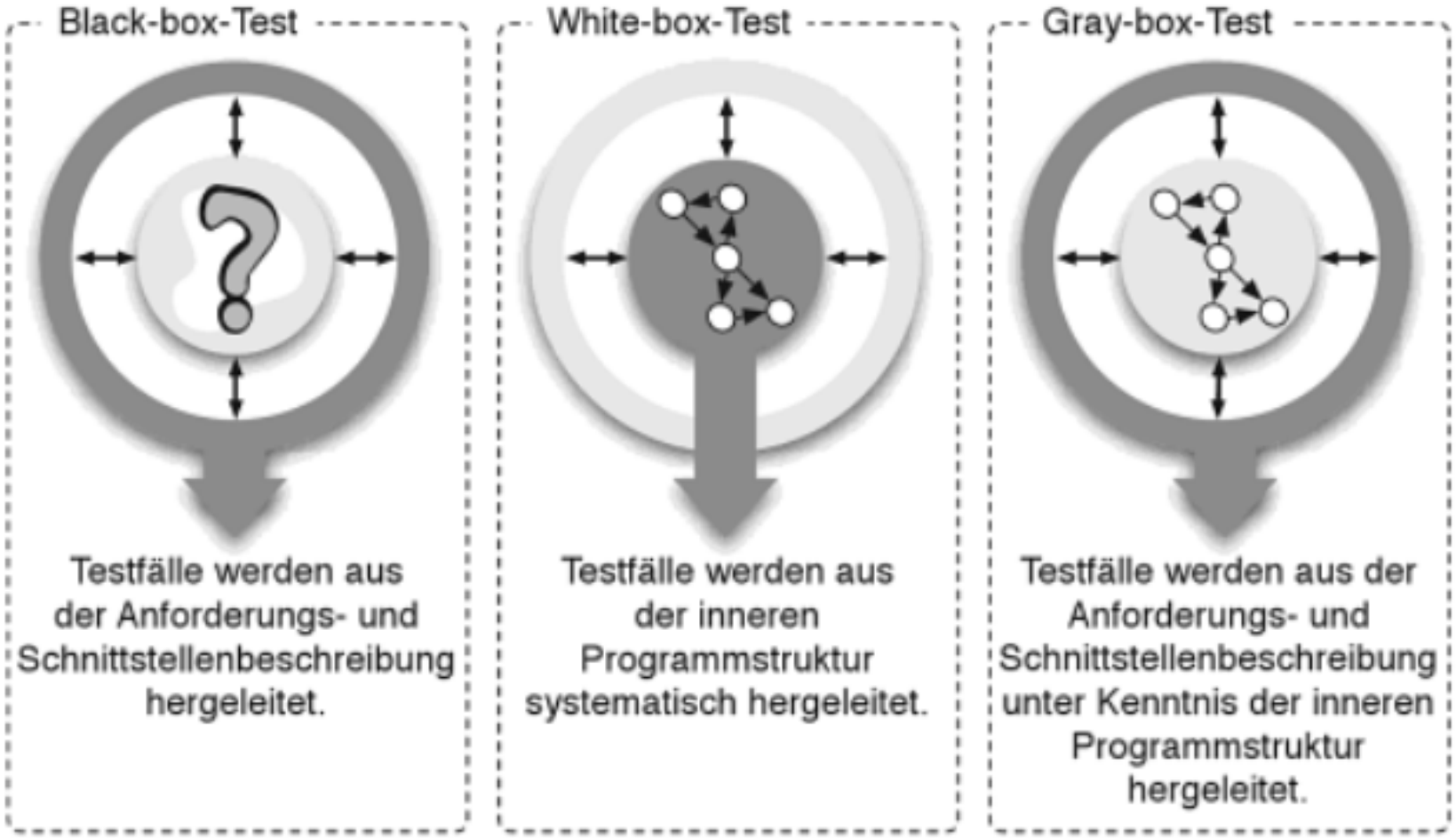
Vorgehen:

1. Die durch den Test verfolgten Ziele sowie Bedingungen und Voraussetzungen festlegen.
2. Testfälle spezifizieren.
3. Testausführung festlegen.

Prüfmethodik

- Black Box Tests
- White Box Tests
- Gray Box Tests

Prüftechniken im Vergleich



Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Black Box Testverfahren

Die Verfahren werden auch als **spezifikationsbasierte Testentwurfsverfahren** bezeichnet, da sie auf der Spezifikation (den Anforderungen) basieren.

Ein Test mit allen möglichen Eingabewerten und deren Kombination wäre ein vollständiger Test. Dies ist aber wegen der großen Zahl von möglichen Eingabewerten und Kombinationen unrealistisch. Eine sinnvolle Auswahl aus den möglichen Testfällen muss getroffen werden.

Black Box Testtechniken

- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter Test
- Paarweises Testen
- Diversifizierende Verfahren

Äquivalenzklassentest

Bsp: Methode der Berechnung eines Absolutbetrags aus java.lang.Math:

```
public static long abs(long a)
```

Testen aller mögliche Eingabewerte:

Long Variable hat 64 Bit ➔ 2^{64} Testfälle

➔ **Prinzip**

1. Bilden von Äquivalenzklassen
2. Testfallkonstruktion

Äquivalenzklassentest

Äquivalenzklassen: Teilmengen der möglichen Eingabewerte, die intern eine identische Verarbeitung erwarten lassen.

Bsp

```
public static long abs(long a)
```

Aquivalenzklassen:

[minLong,0] und [1,maxLong] oder

[minLong,-1] und [0,maxLong]

Mehrdimensionale Äquivalenzklassen

Im Fall einer Methode, die n Übergabeparameter entgegennimmt, sind n-dimensionale Äquivalenzklassen zu bilden.

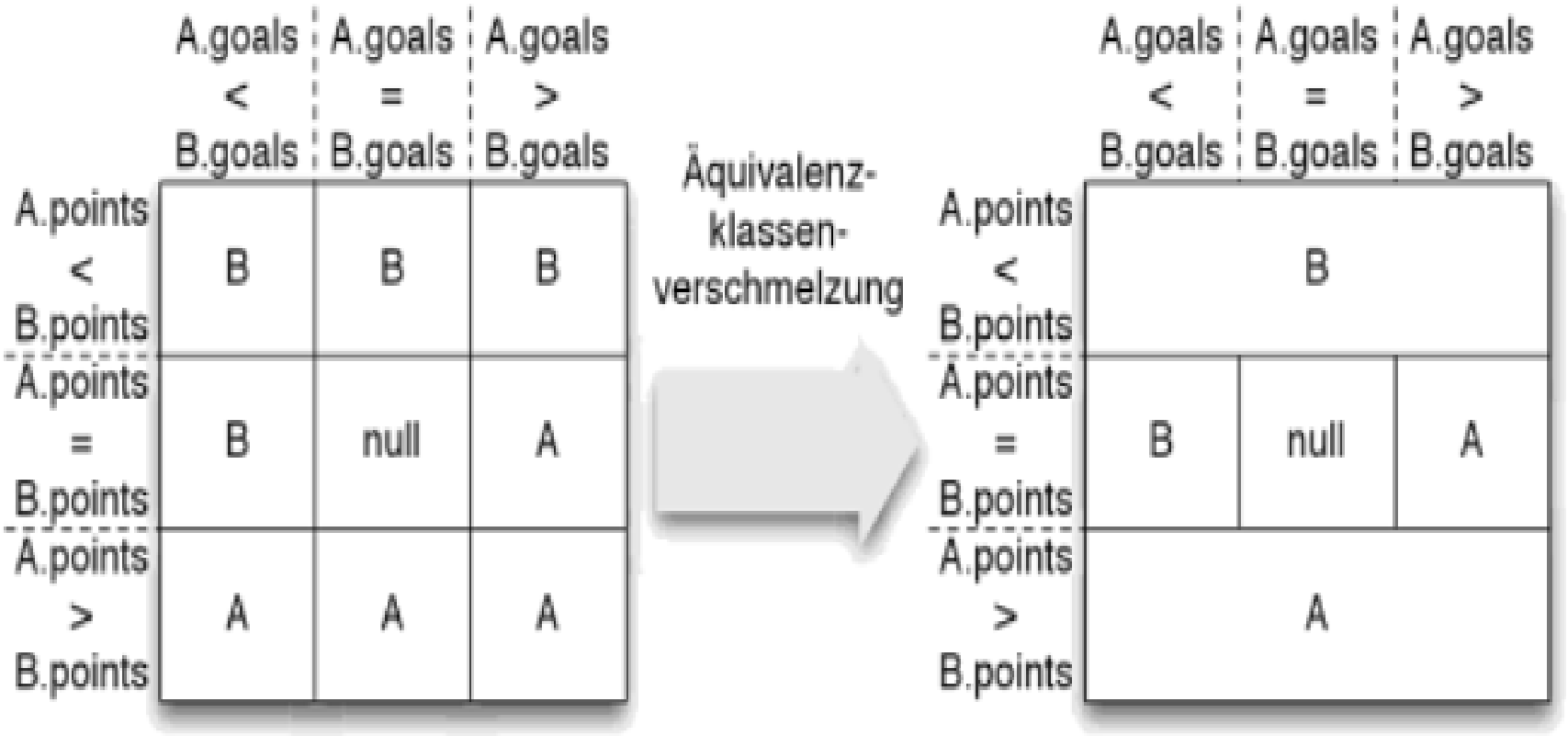
➔**Vorgehen:**

1. Äquivalenzklassenbildung für jeden Eingabeparameter suchen.
2. Zusammengehörige Äquivalenzklassen verschmelzen.

Bsp. Zur Äquivalenzklassenbildung

```
package aequivalenzklassen;

public class Team {

    int points;
    int goals;

    /** Bestimmt den Sieger unter zwei Mannschaften
     * @param team1 Referenz auf das erste Team
     * @param teaam2 Referenz auf das zweite Team
     */
    public static Team calculateChampion(Team team1, Team team2){

        ....
    }

}
```

Bsp Äquivalenzklassen



Äquivalenzklassen für partiell definierte Funktionen

Was, wenn die erlaubten Eingabewerte eine Teilmenge der durch den Datentyp definierten Eingabewerte sind?

➔ Zwei prinzipielle Vorgehensweisen:

- ~~**Partielle Partitionierung**~~
  ~~Äquivalenzklassenbildung unter Ausschluss der ungültigen Eingabewerte.~~
- **Vollständige Partitionierung**
  Äquivalenzklassenbildung bezieht die ungültigen Werte mit ein.

Äquivalenzklassen - Bsp

Beschreibung:

Über die Verkaufssoftware kann ein Autohaus seinen Verkäufern Rabattregeln vorgeben. In der Beschreibung der Anforderungen findet sich folgende Textpassage: „Bei einem Kaufpreis von weniger als

15.000 € soll kein Rabatt gewährt werden. Bei einem Preis bis zu 20.000 €sind 5 % Rabatt angemessen. Liegt der Kaufpreis unter 25.000 €, sind 7 % Rabatt möglich, darüber sind 8,5 % Rabatt einzuräumen.“

Äquivalenzklassen

Parameter	Äquivalenzklasse	Repräsentant
Verkaufspreis	gÄK1: $0 \leq x < 15000$ gÄK2: $15000 \leq x \leq 20000$ gÄK3: $20000 < x < 25000$ gÄK4: $x \geq 25000$	14500 16500 24750 31800

Äquivalenzklassen für ungültige Werte:

Parameter	Äquivalenzklasse	Repräsentant
Verkaufspreis	uÄK1: $x < 0$ (»negativer« – also falscher – Verkaufspreis) uÄK2: $x > 1000000$ (»unrealistisch hoher« Verkaufspreis[a])	-4000 1500800

Black Box Testtechniken

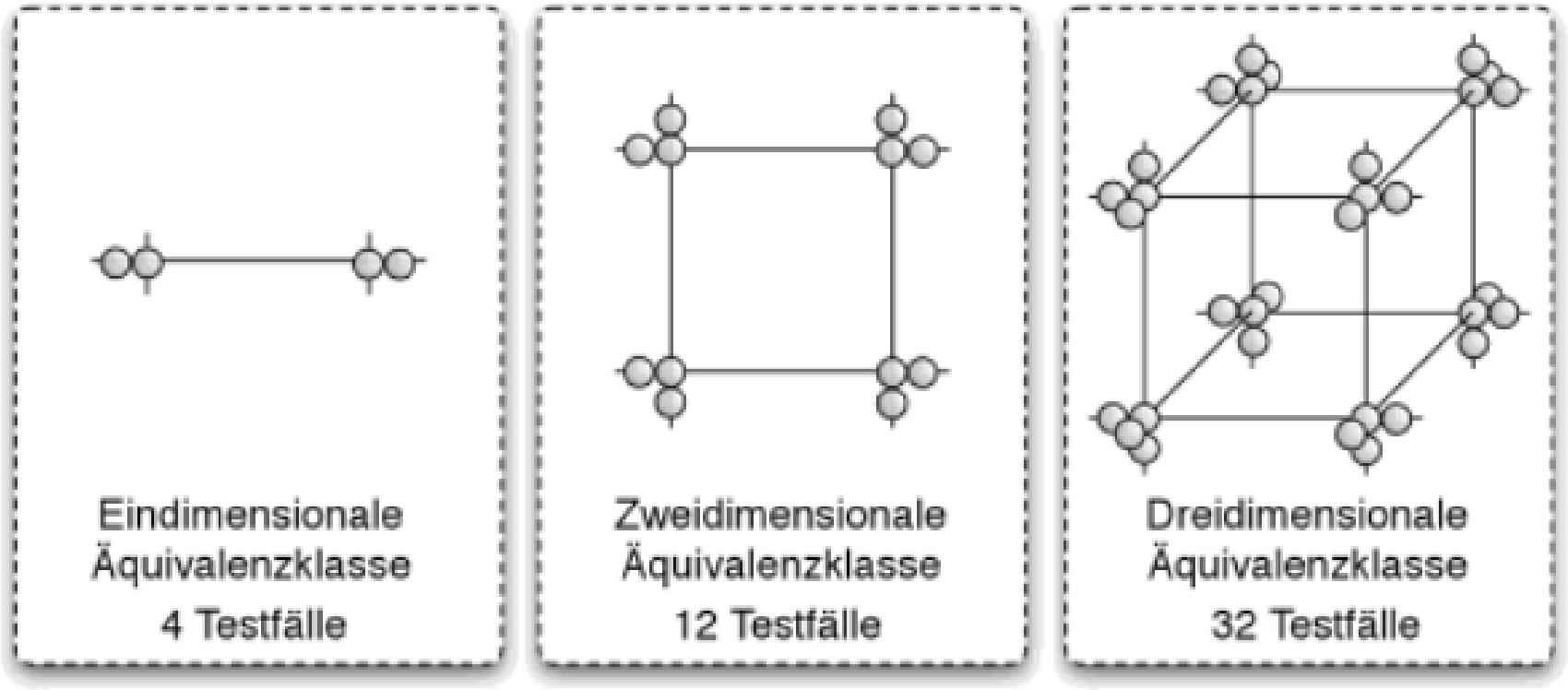
- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter test
- Paarweises Testen
- Diversifizierende Verfahren

Grenzwertbetrachtung

- Partitionierung identisch wie bei der Äquivalenzklassenbildung

- Testfallauswahl aber nicht beliebig innerhalb einer Äquivalenzklasse, sondern jeweils den Randwert und Tupel, bei denen ein einzelner Wert außerhalb der Äquivalenzklasse liegt.

Grenzwertbetrachtung



Voraussetzung für diese Methode: Eingabewerte sind geordnet.

Black Box Testtechniken

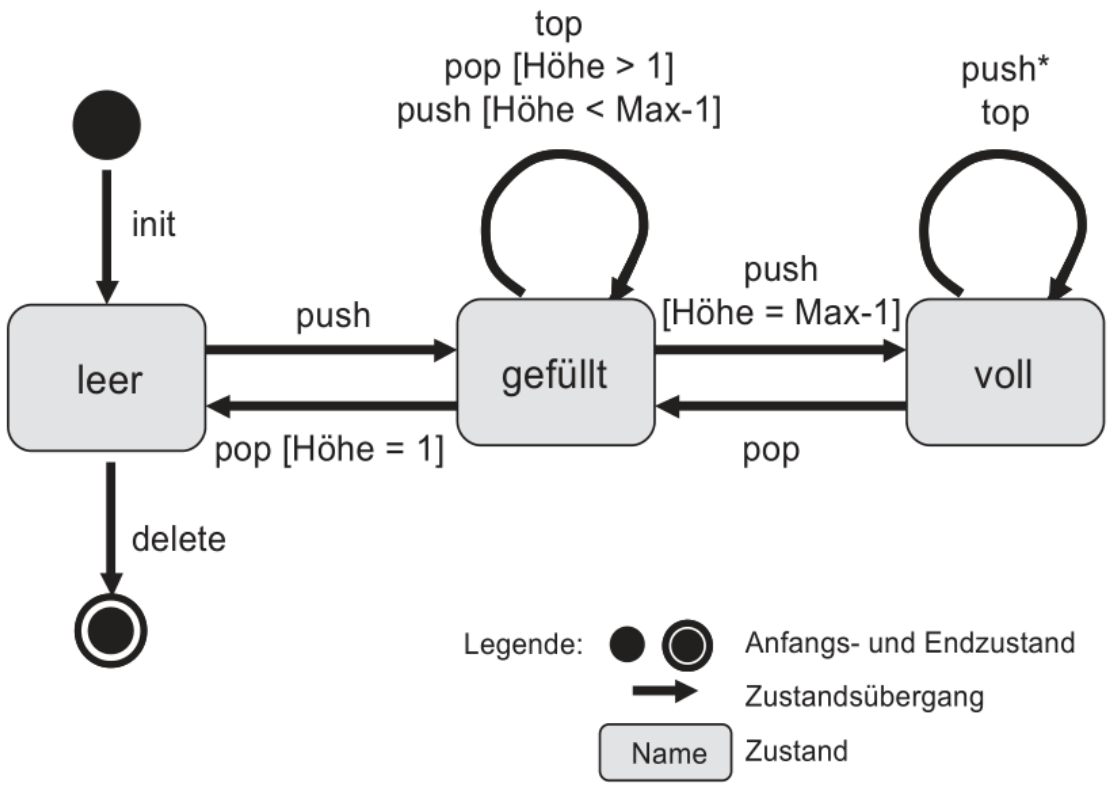
- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter Test
- Paarweises Testen
- Diversifizierende Verfahren

Zustandsbehafteter Software Test

- **Bisher**: Ergebniswerte werden ausschließlich durch Eingabewerte bestimmt.
- **Jetzt**: Programmfunktionen mit Gedächtnis (also Zustand)
- **Idee**: Alle möglichen Übergänge zwischen zwei Zuständen werden mit mindestens einem Testfall geprüft.

Zustandsbehafteter Software Test – Bsp.

Bsp: Stapel (z.B. von Tellern)



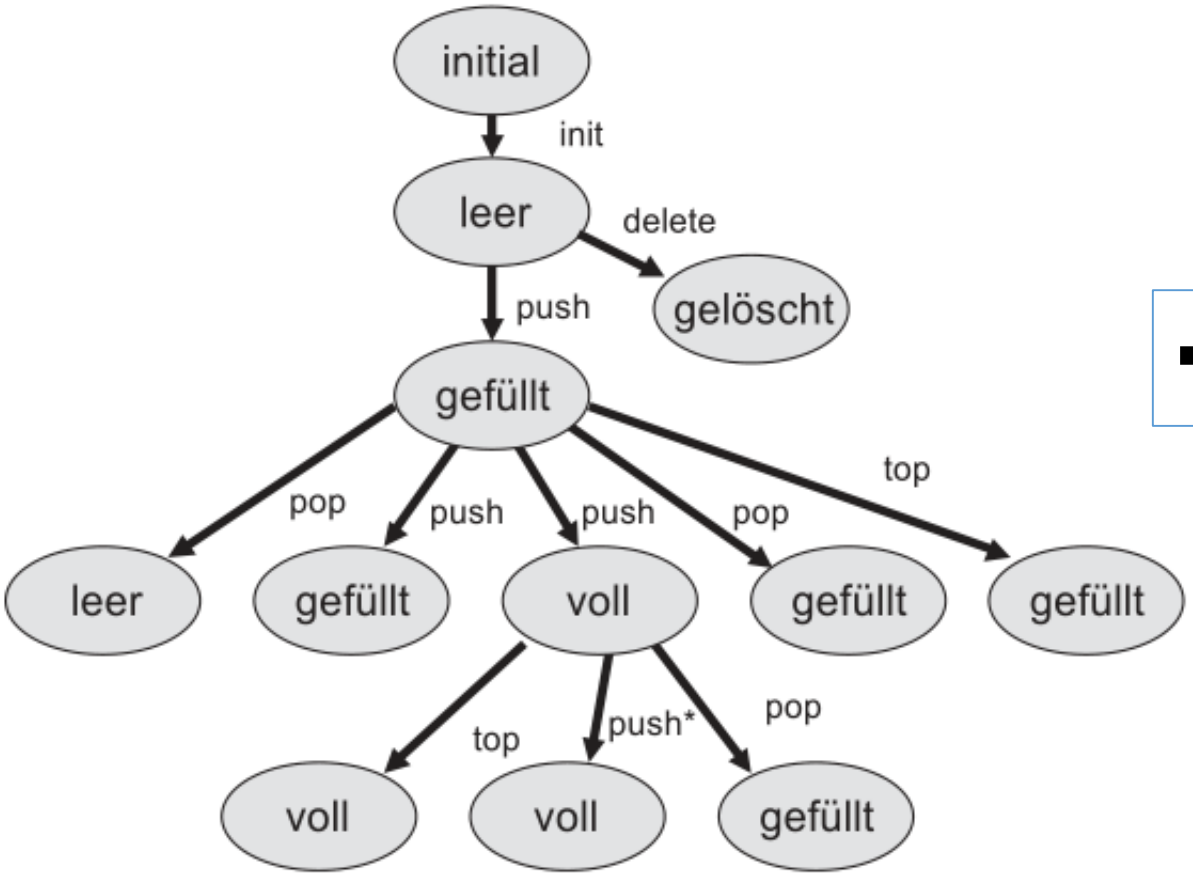
Zustandsbehafteter Software Test

Ermittlung der Testfälle mittels Übergangsbaum

Bildung eines Übergangsbaums:

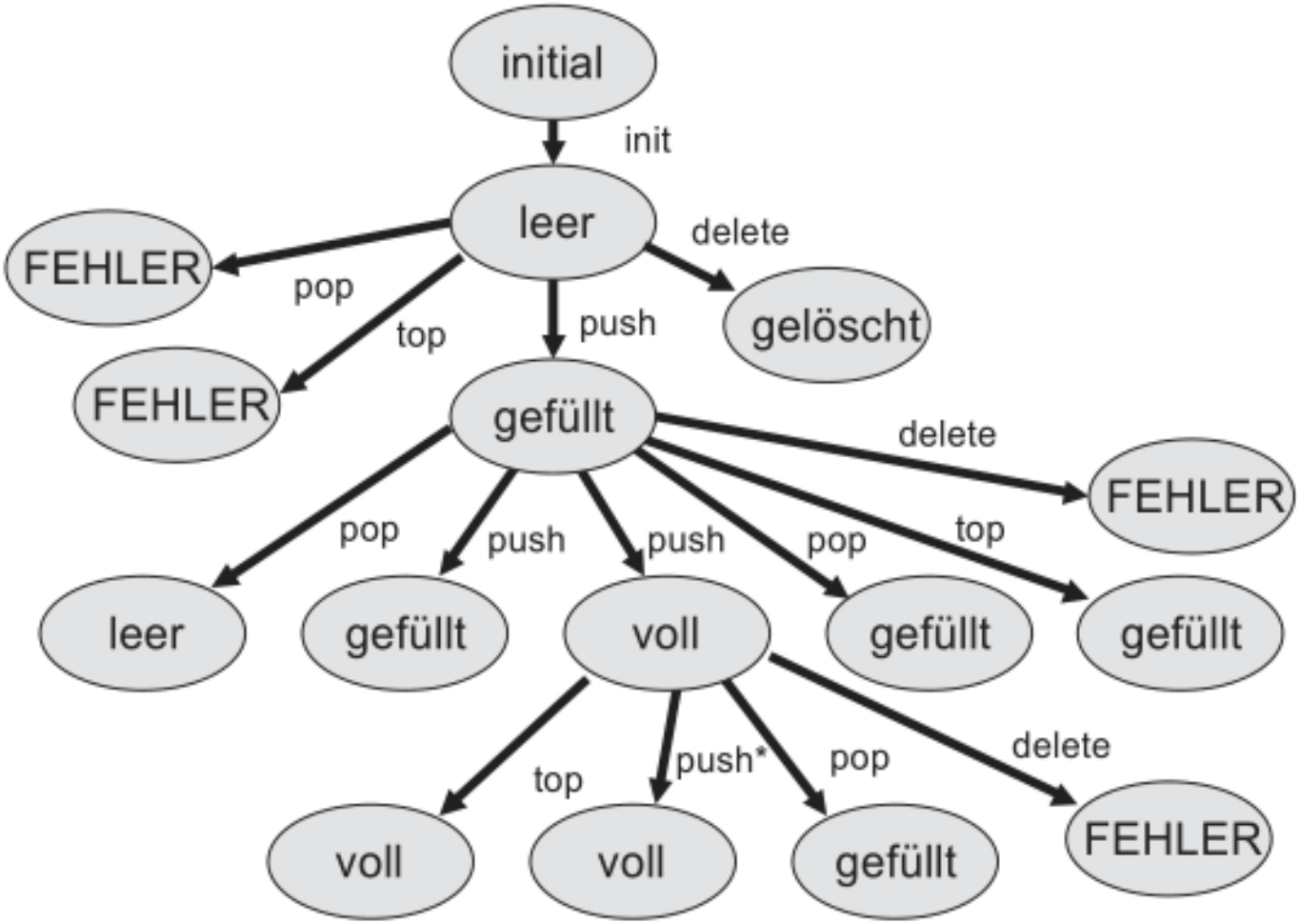
1. Der Anfangszustand ist die Wurzel des Baums.
2. Für jeden möglichen Übergang vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus eine Verzweigung zu einem Knoten, der den Nachfolgezustand repräsentiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangsbaums solange wiederholt, bis eine der beiden Endbedingungen eintritt:
   a. Der dem Blatt entsprechende Zustand ist auf dem Weg von der Wurzel zum Blatt bereits einmal im Baum enthalten. Diese Endbedingung entspricht einem Durchlauf von einem Zyklus im Zustandsdiagramm.
   b. Der dem Blatt entsprechende Zustand ist ein Endzustand und hat somit keine weiteren Übergänge, die zu berücksichtigen wären.

Übergangsbaum für Stapelbeispiel



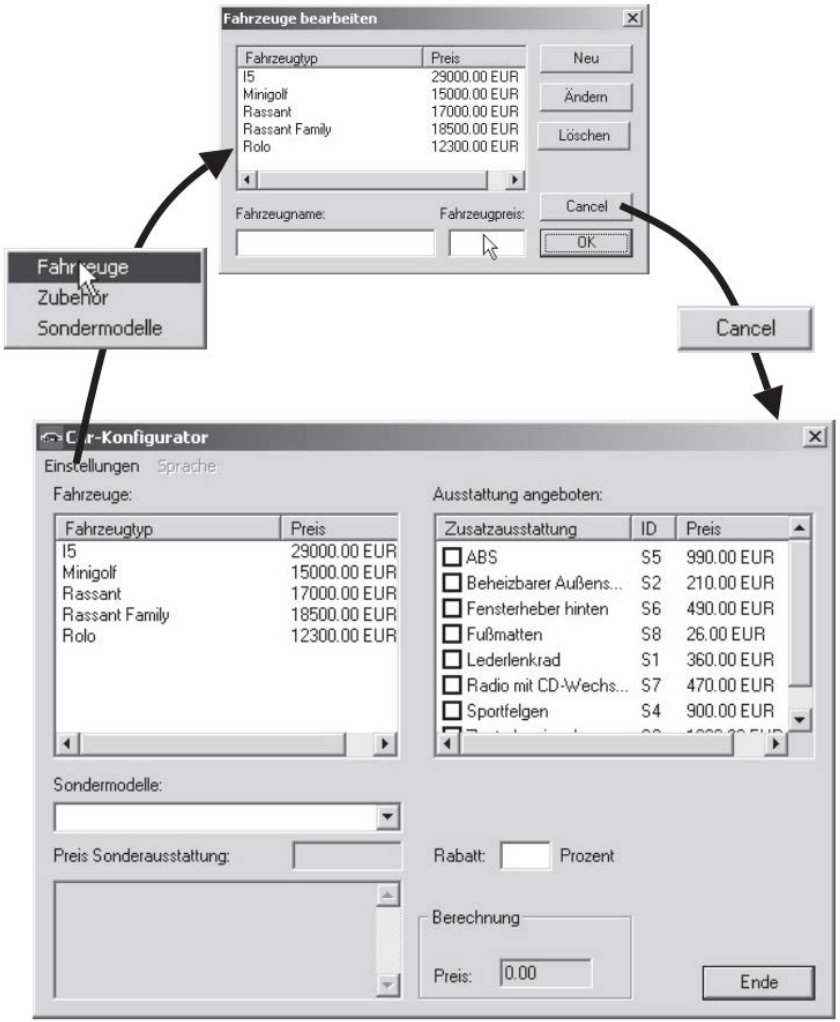
➔ 8 Testfälle

Übergangsbaum für Robustheitstest



Zustandsbasierter Test - Bsp

GUI Masken können als Zustände aufgefasst werden. Daher eignet sich der zustandsbasierte Test für die Testfallerstellung von GUI Abfolgen.



Black Box Testtechniken

- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter test
- Paarweises Testen
- Diversifizierende Verfahren

Use Case Test

Speziell für Systemebene (Systemtest, Abnahmetest):

Usecase basierter Black Box Test:
Alle möglichen Szenarien eines Usecases werden durch mindestens einen Testfall abgedeckt.

Black Box Testtechniken

- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter Test
- Paarweises Testen
- Diversifizierende Verfahren

Entscheidungstabellen basierter Test

- **Bisher:** Eingabeparameter unabhängig voneinander betrachtet.
- **Jetzt:** Berücksichtigung von derartigen Abhängigkeiten
- **Prinzipielles Vorgehen:**
  - Ursache – Wirkungs Graph erstellen
  - Entscheidungstabelle ableiten

Beispiel Geldautomat

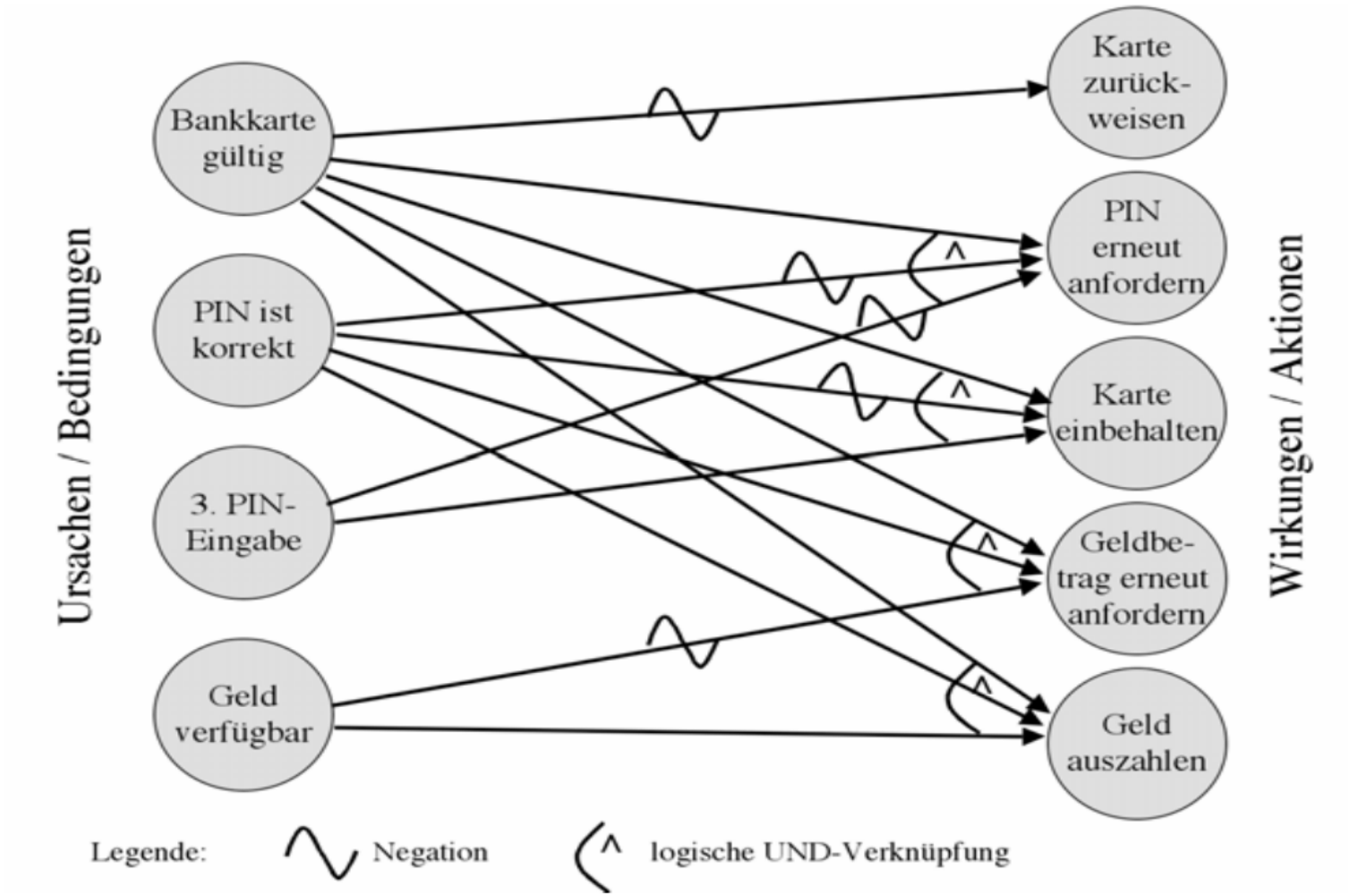
Um Geld aus einem Automaten zu bekommen, sind folgende Bedingungen zu erfüllen :

- Die Bankkarte ist gültig.
- Die PIN ist korrekt eingegeben.
- Es dürfen nur maximal drei PIN-Eingaben erfolgen.
- Geld steht zur Verfügung (im Automat und auf dem Konto).

Als Aktion bzw. Reaktion des Geldautomaten sind folgende Möglichkeiten gegeben:

- Karte zurückweisen
- Aufforderung, erneut die PIN einzugeben
- Karte einbehalten
- Aufforderung, neuen Geldbetrag einzugeben
- Geldbetrag auszahlen

Bsp für Ursache Wirkungsgraph



Graph ➔ Tabelle

1. Auswahl einer Wirkung.
2. Durchsuchen des Graphen nach Kombinationen von Ursachen, die den Eintritt der Wirkung hervorrufen bzw. nicht hervorrufen.
3. Erzeugung jeweils einer Spalte der Entscheidungstabelle für alle gefundenen Ursachenkombinationen und die verursachten Zustände der übrigen Wirkungen.
4. Überprüfung, ob Entscheidungstabelleneinträge mehrfach auftreten und ggf. Entfernen dieser Einträge.

Bsp für optimierte Entscheidungstabelle

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

Black Box Testtechniken

- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter test
- Paarweises Testen
- Diversifizierende Verfahren

Paarweises Testen

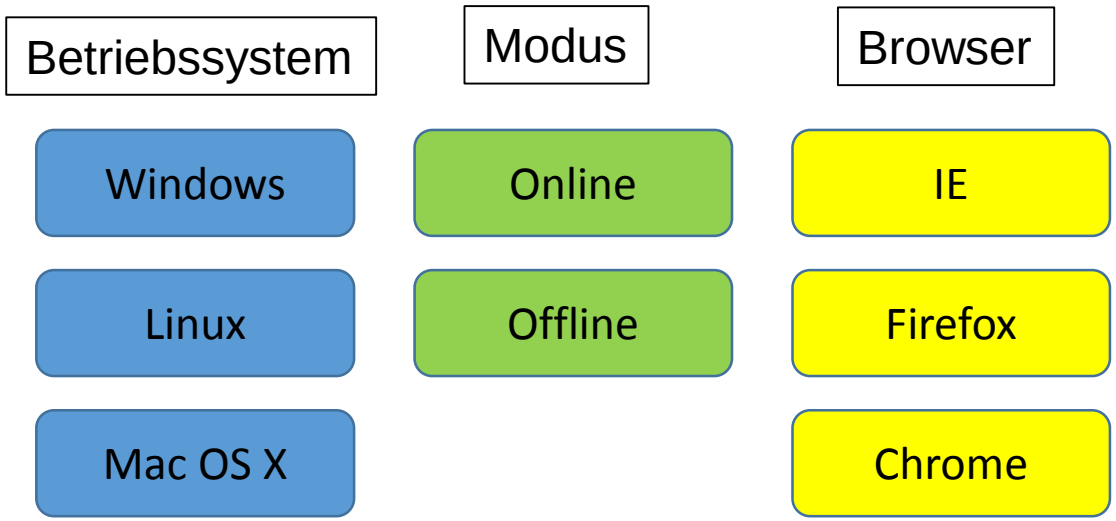
- Testfallkonstruktion in der Absicht, die Anzahl der Fälle aus rein kombinatorischen Überlegungen zu reduzieren.

- Annahme: Es müssen nicht alle möglichen, sondern nur alle paarweisen Kombinationen getestet werden, um alle Fehler zu finden.

➔ **Ansatz:** Sicherstellen, dass jeder Repräsentant einer Äquivalenzklasse mit jedem Repräsentanten der anderen Äquivalenzklassen in einem Testfall zur Ausführung kommt (d. h. paarweise Kombination statt vollständiger Kombination).

Paarweises Testen - Bsp

Webapplikation soll mit gängigen Browsern und gängigen Betriebssystemen kompatible sein und im online sowie im offline Modus funktionieren.



➔ 18 Konfigurationen, Bei verschiedenen Versionen entsprechend Mehr.

Paarweises Testen

- Pragmatischer Ansatz: Jedes Ausprägungspaar ist durch einen Testfalle abgedeckt
- **Nicht:** Alle kombinatorisch möglichen Konfigurationen

➔

- Jeder Web Browser mit jedem Betriebssystem
- Jeder Modus mit jedem Webbrowser
- Jeder Modus mit jedem Betriebsystem

Paarweiser vollständiger Test - Bsp

OS	Modus	Browser
Windows	Online	IE
Windows	Offline	Firefox
Windows	Online	Chrome
Linux	Online	Firefox
Linux	Offline	Chrome
Linux	Offline	IE
Mac OS X	Online	Chrome
Mac OS X	Offline	IE
Mac OS X	Online	Firefox

9 Testfälle anstelle von 18 !

Konstruktion der Testfälle

Für das paarweise Testen wird die Konstruktion der Testfälle schnell ziemlich komplex.

Eine Möglichkeit zur Konstruktion ist die Verwendung orthogonaler Felder.

Das Thema wird hier nicht weiter behandelt.

Black Box Testtechniken

- Äquivalenzklassen Test
- Grenzwertbetrachtung
- Zustandsbasierter Software Test
- Use Case Test
- Entscheidungstabellen basierter test
- Paarweises Testen
- Diversifizierende Verfahren

Diversifizierende Verfahren

- Diversifizierende Testverfahren vergleichen mehrere Versionen eines Programms gegeneinander.

- ➔ Eine zweite Implementierung übernimmt die Aufgabe der Spezifikation

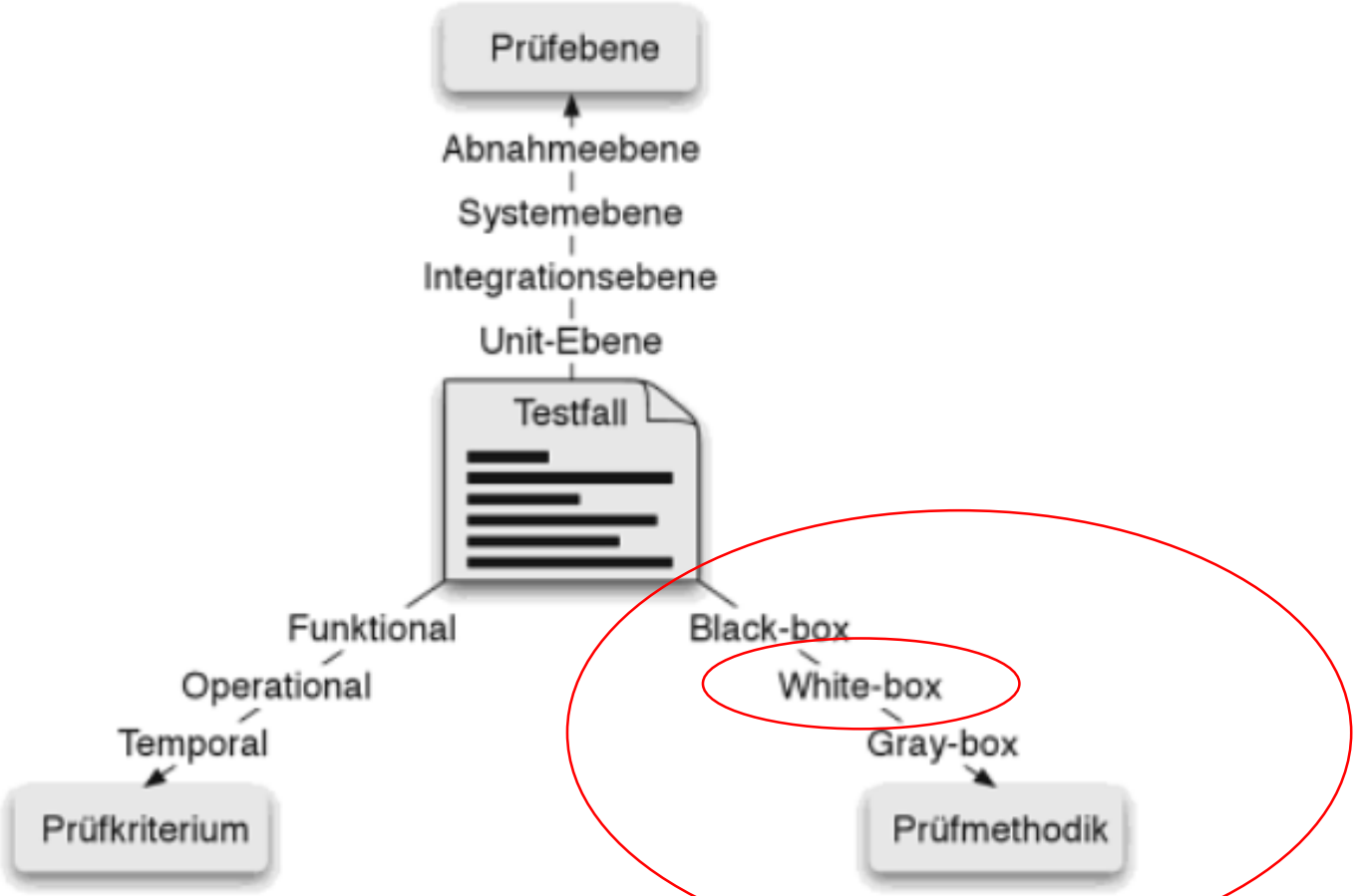
Diversifizierende Testverfahren

- **Back to back Tests**
  Test, bei dem zwei oder mehr Varianten einer Komponente oder eines Systems mit gleichen Eingaben ausgeführt werden und deren Ergebnisse dann verglichen werden. Im Fall von Abweichungen wird die Ursache analysiert.
- **Regressionstests**
  Erneuter Test eines bereits getesteten Programms bzw. einer Teilfunktionalität nach deren Modifikation mit dem Ziel, nachzuweisen, dass durch die vorgenommenen Änderungen keine Fehlerzustände eingebaut oder (bisher maskierte Fehlerzustände) freigelegt wurden.
- **Mutationstest**
  Technik, um Testverfahren zu beurteilen.

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Testklassifikation



Merkmalsräume der Testklassifikation

White Box Tests

Black Box Test:

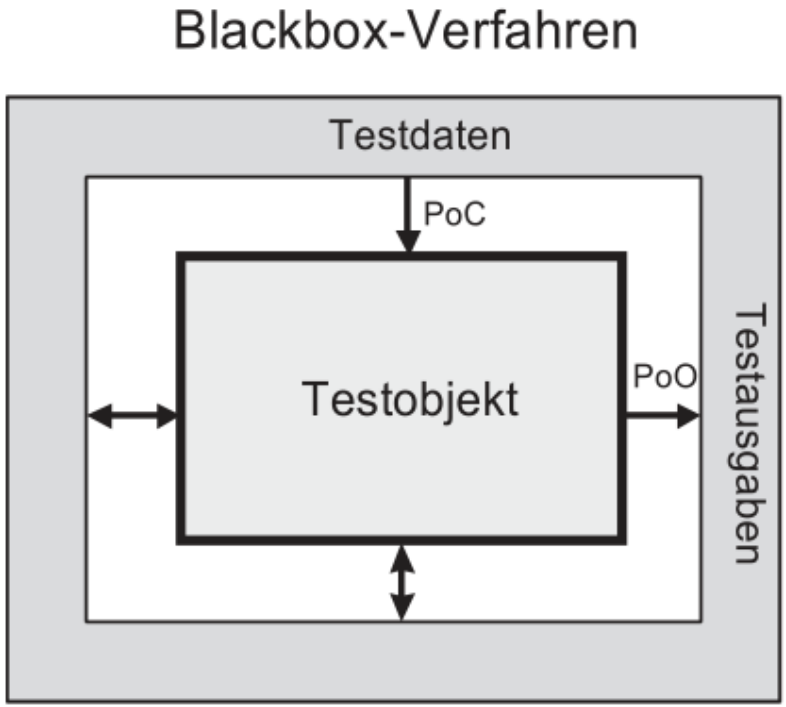
Input basiert ausschließlich auf der funktionalen Beschreibung

Nun: **White Box Test:**

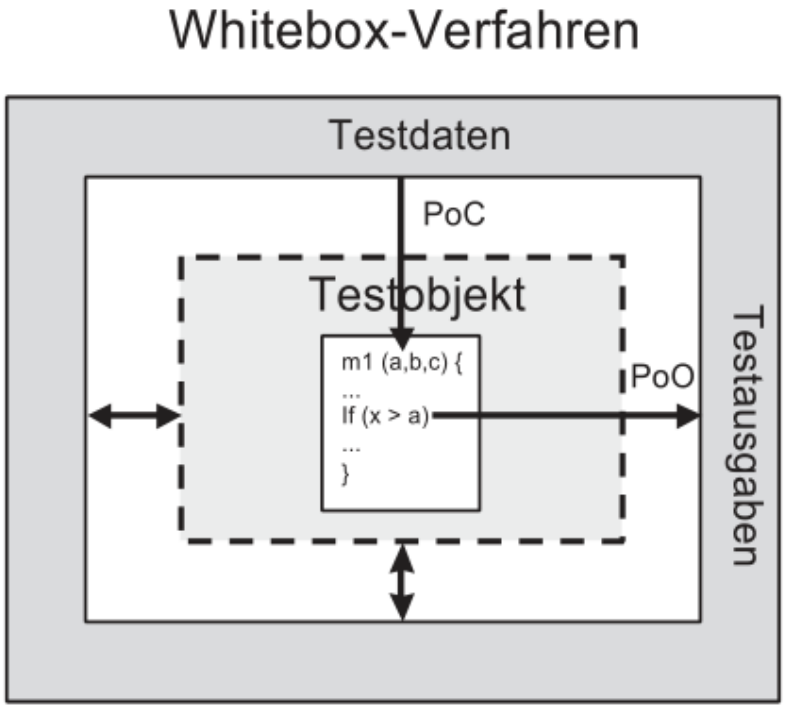
Basiert auf der Analyse der inneren Programmstrukturen

White Box Test = Strukturtest

Black Box vs White Box



PoC und PoO »außerhalb« des Testobjekts



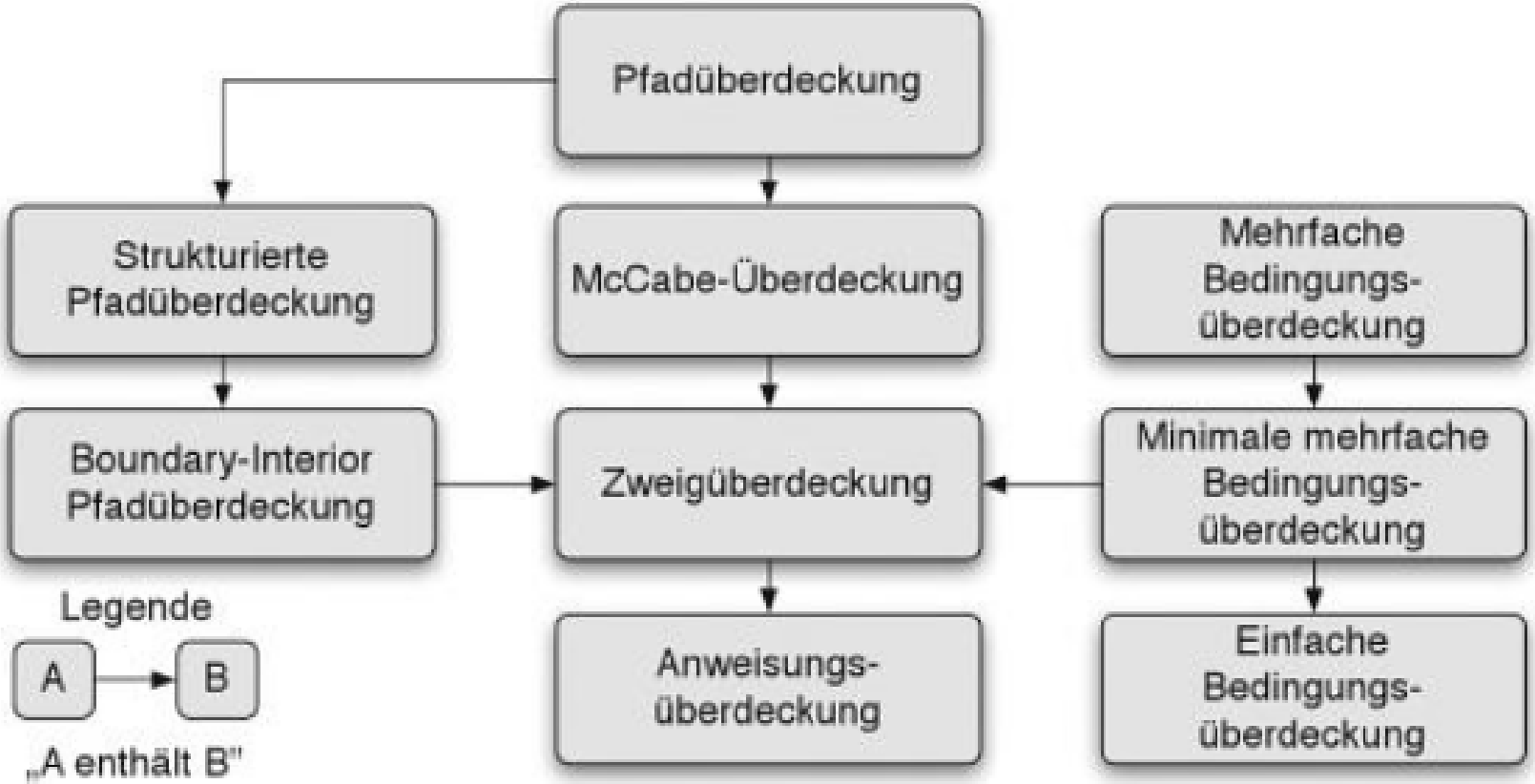
PoC und/oder PoO »innerhalb« des Testobjekts

Einteilung von Strukturtests

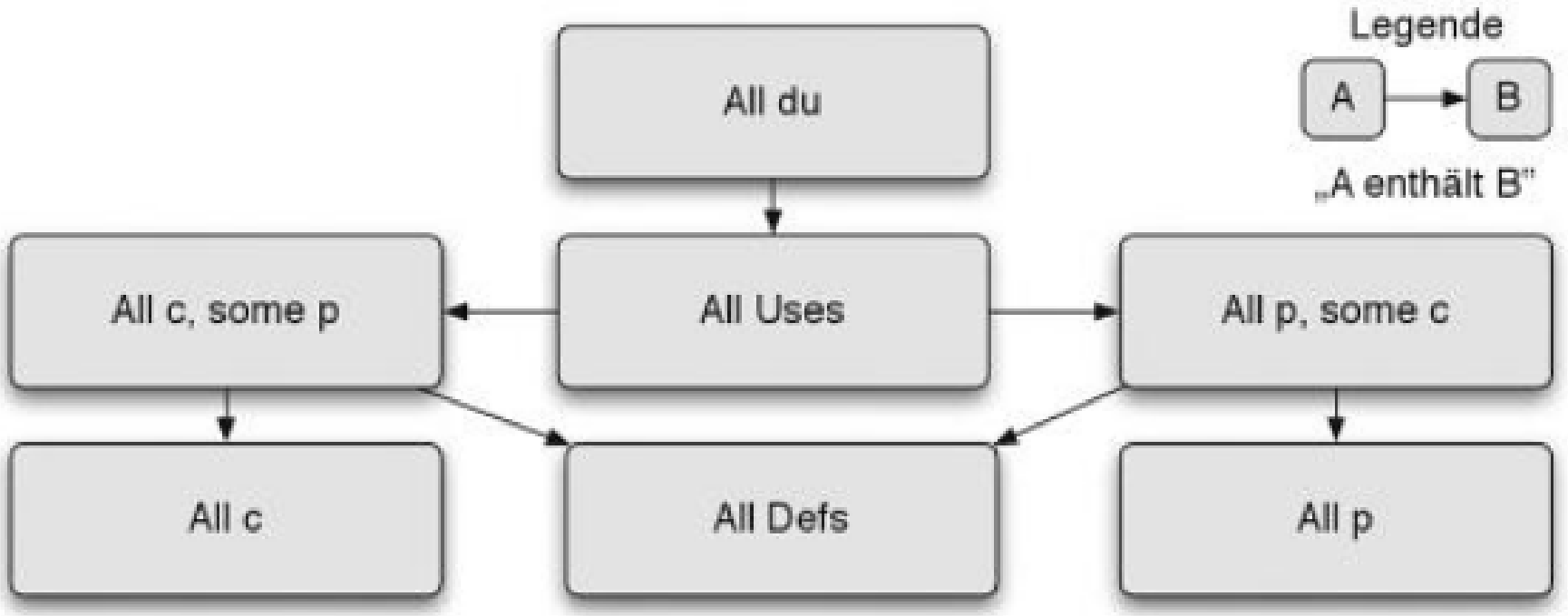
- **Kontrollflussorientierte Tests**
  Konstruktion der Testfälle ausschließlich auf Basis der internen Berechnungspfade eines Programms. Beschaffenheit der Testdaten spielt keine Rolle.

- **Datenflussorientierte Tests**
  Heranziehen zusätzlicher Kriterien aus der Beschaffenheit der manipulierten Daten.

Kontrollflussorientierte Strukturtests



Datenflussorientierte Strukturtests



Ablauf eines typischen Strukturtests

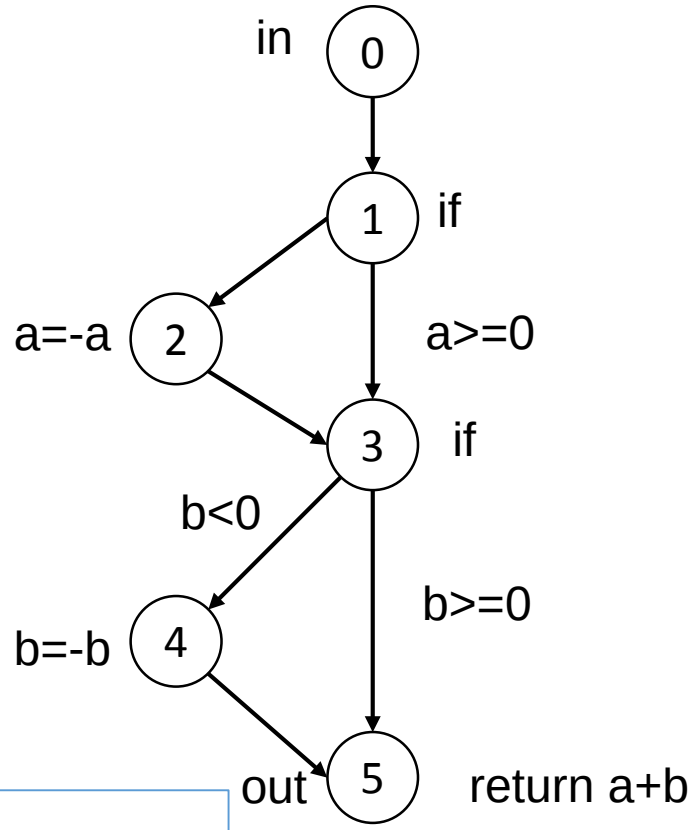
- **Strukturanalyse**
  Aus dem Programmcode wird der **Kontrollflussgraph** extrahiert.

- **Testkonstruktion**
  Ableitung der Testfälle entsprechend eines vorher definierten Überdeckungskriterium.

- **Testdurchführung**

Kontrollflussmodellierung - Bsp

```
int manhattan(int a, int b){

    if(a<0){
        a=-a;
    }
    if(b<0){
        b=-b;
    }

    return a+b;
}
```



Immer:
Ein einziger Einstiegspunkt,
Ein einziger Ausstiegspunkt!

Arten von Kontrollflussgraphen - 1

- **Kantenmarkierte Kontrollflussgraphen (üblich und hier behandelt)**
  Anweisungen werden den Knoten, Verzweigungsbedingungen den Kanten zugeordnet.

- **Knotenmarkierte Kontrollflussgraphen (relevant für Required k-Tupel Test)**
  Verzweigungsbedingungen werden ebenfalls den Knoten zugeordnet.

Weitere Unterscheidungen

- **Expandierte Kontrollflussgraphen**
  Jeder Befehl ein separater Knoten.

- **Teilkollabierte Kontrollflussgraphen**
  zwei oder mehrere sequenzielle Befehle in einem einzigen Knoten.

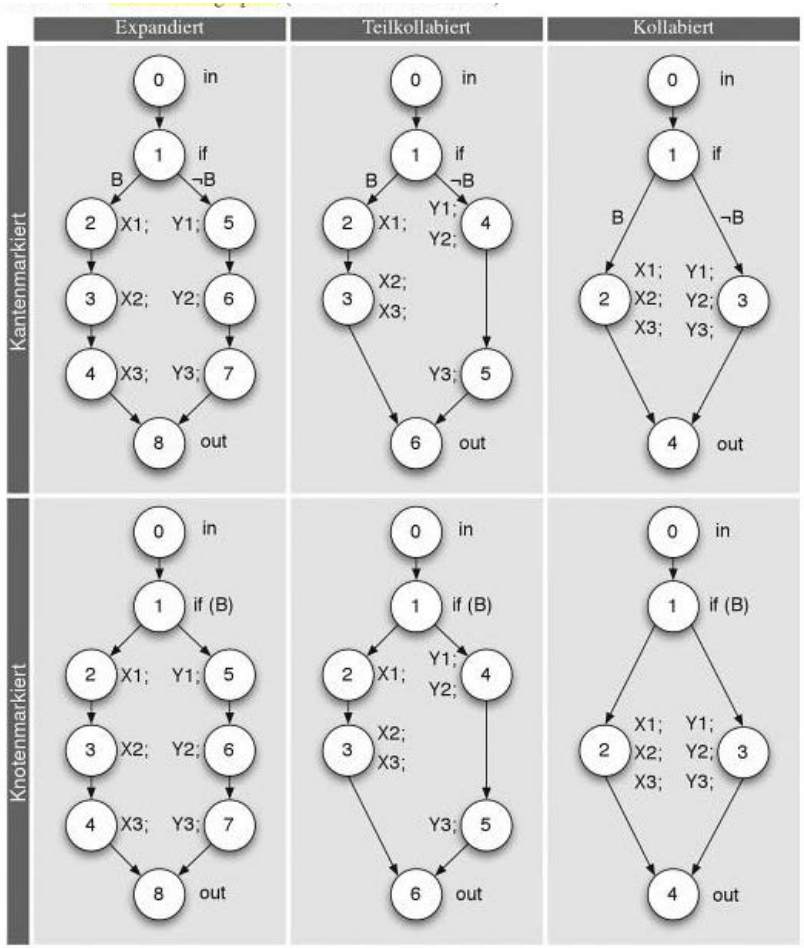
- **Kollabierte Kontrollflussgraphen (am häufigsten verwendet)**
  Verzweigungsfreie Befehlsblöcke in einem einzigen Knoten.

Klassifikationsmerkmale - Bsp

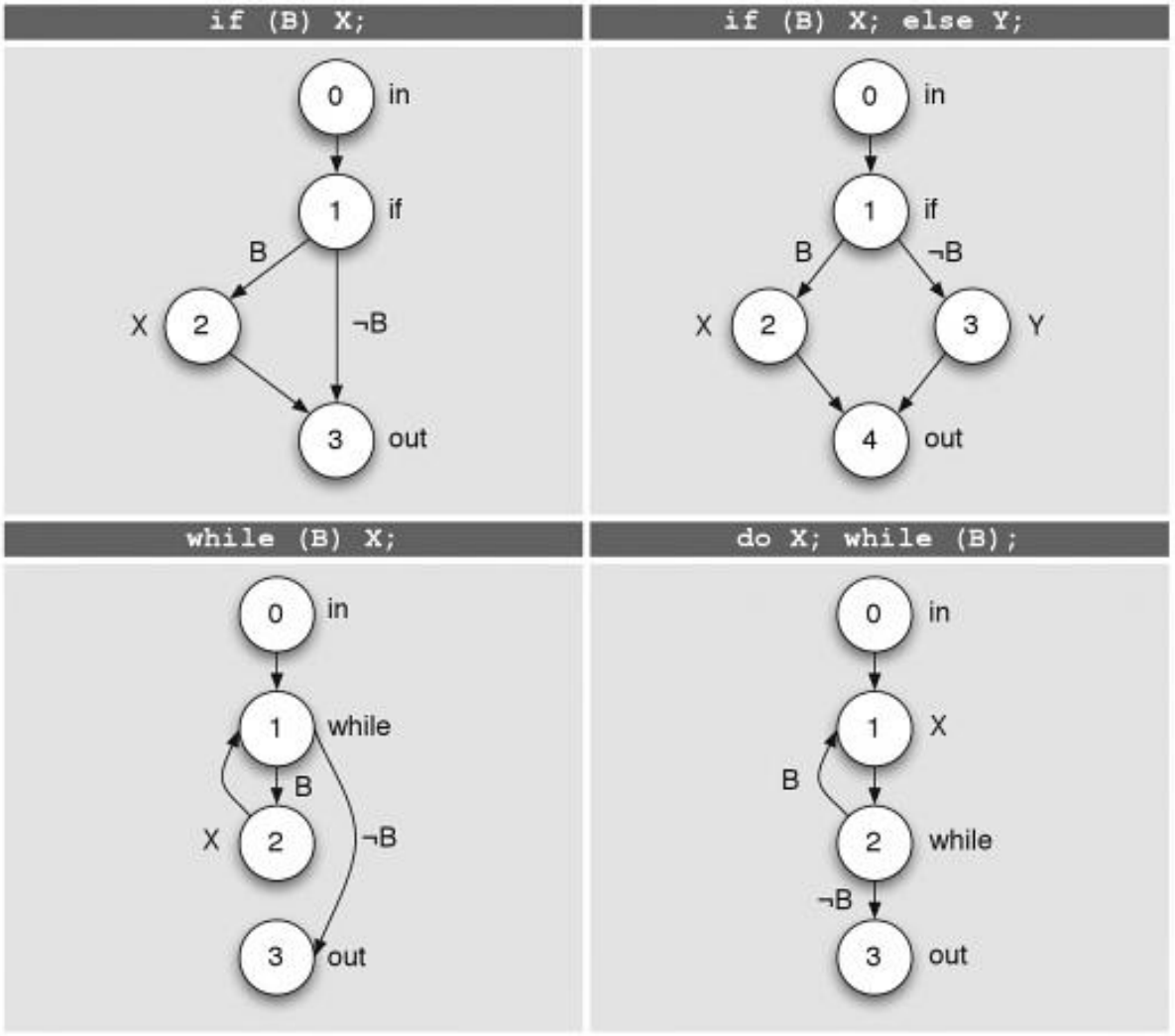
expandiert teilkollabiert kollabiert

Kanten-markiert

Knoten-markiert



Kontrollflussgraphen elementarer Konstrukte



Überdeckungskriterien

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

Überdeckungskriterien

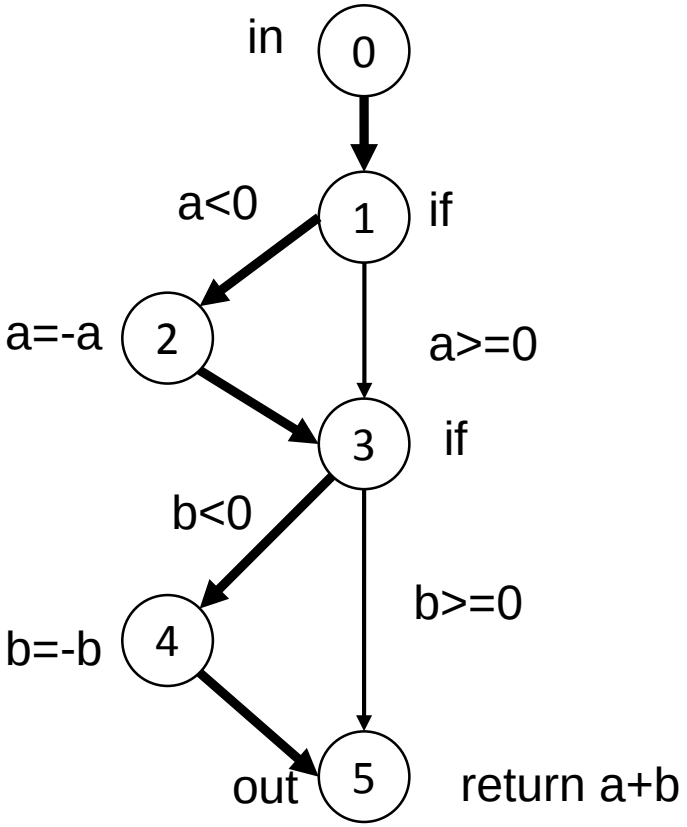
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

Anweisungsüberdeckung

- Testmenge wird so gewählt, dass alle Knoten des Kontrollflussgraphen mindestens einmal durchlaufen werden.
- Auch als C_0 Test bezeichnet
- Schwächstes der hier vorgestellten Kriterien.

Siehe Girgis, M.R., Woodward M.R.: An experimental comparison of the error exposing ability of program testing criteria. In: Proceedings of the Workshop of Software Testing, pp.64 -73. Banff(1986): Es konnten nur 18% der Fehler eines Software Systems aufgedeckt werden.

Anweisungsüberdeckungstest - Bsp



Bsp manhattan:

Testfälle:

- manhattan(-1,-1)

Überdeckung: {0,1,2,3,4,5}

Diskussion der Anweisungsüberdeckung

- Mit wenigen Testfällen zu erfüllen.
- Wichtige Fälle bleiben unberücksichtigt.
- Dennoch häufig schwer zu erreichen.
- Wird die Anweisungsüberdeckung nicht zu 100% erfüllt, ist das Risiko nicht kalkulierbar.

- Standard **RTCA DO-178B** für Software Anwendungen in der Luftfahrt verlangt Anweisungsüberdeckung für alle Komponenten, deren Ausfall zu einer bedeutenden aber nicht kritischen Fehlfunktion führen kann.

Bsp für schwer zu realisierende Anweisungsüberdeckung

```
uint8_t *hardToTest(...)
{
    /*Get the file properties*/
    if(stat(filename,&fileProperties)!= 0){
        return NULL;
    }

    /*Open file*/
    if(!(file=fopen(filename,"r"))){
        return NULL;
    }

    /*Allocate memory*/
    if(!(data=(uint8_t *)malloc(fileProperties.st_size))){
        return NULL;
    }

    return data;
}
```

Schwierigkeiten in dem Bsp

- Wie erhalten Sie die File Properties, können das File aber nicht öffnen?

- Wie erzeigen Sie einen Fehler für malloc?

➔malloc durch eine eigene Funktion ersetzen, damit die vorliegende Funktion getestet werden kann.

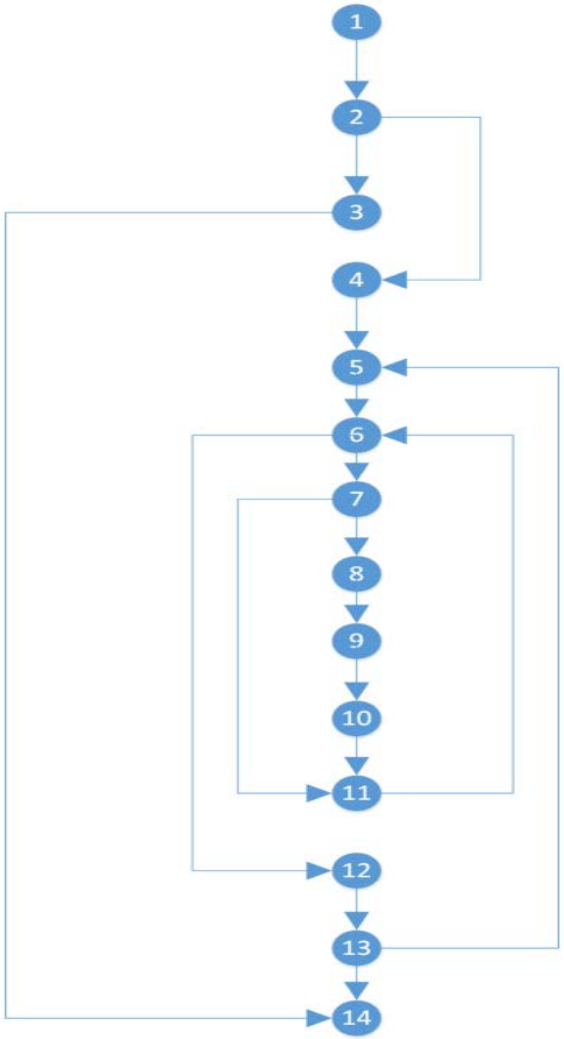
➔Derartige Situationen können dazu führen, dass die C_0-Überdeckung nur aufwändig zu erreichen ist.

Beispiel für Anweisungs Überdeckung

Erstellen Sie einen Kontrollfluss Graphen für das folgende Programm und definieren Sie eine möglichst niedrige Anzahl von Testfällen, so dass die Anweisungsüberdeckung zu 100% gegeben ist.

Übungsbeispiel - Kontrollflussgraph

```
1  static void bubbleSort( int [] array )
   {
2      if( array == null || array.length == 0 )
       {
3          return;
       }
4      int n = array.length;
5      do
       {
6          for( int i = 0; i < n - 1; i++ )
           {
7              if( array[ i ] > array[ i + 1 ] )
               {
8                  int tmp = array[ i ];
9                  array[ i ] = array[ i + 1 ];
10                 array[ i + 1 ] = tmp;
               }
11         }
12         n--;
       }
13     while( n >= 0 );
   }
14
```



Übungsbeispiel - Testfälle

Durch folgende Testfälle ist die Anweisungsüberdeckung zu 100% gegeben:

1. bubbleSort(null)
2. bubbleSort(new int[]{2,1})

Überdeckungskriterien

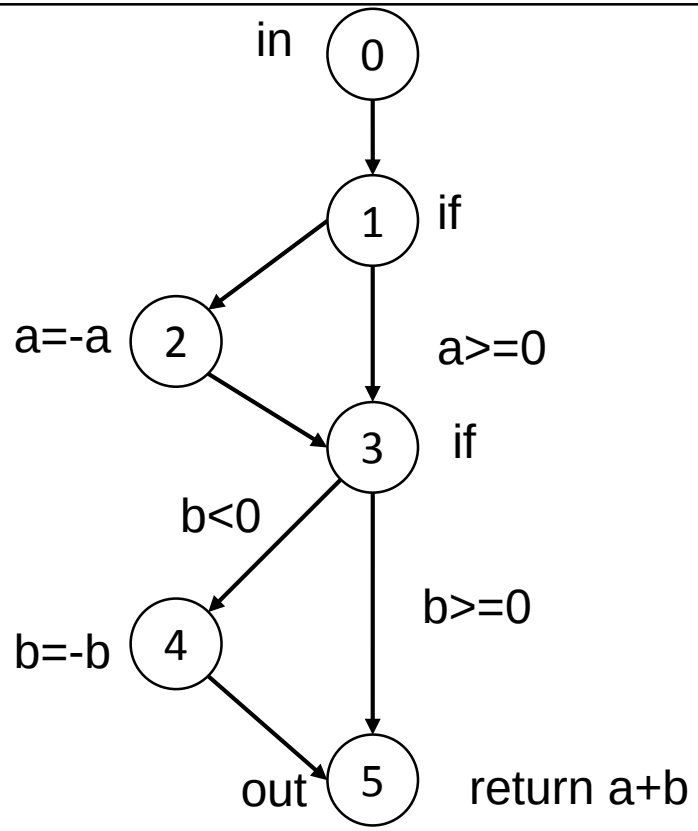
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

Zweigüberdeckung

- Auch als C_1-Überdeckung bezeichnet.
- Jede Kante des Kontrollflussgraphen muss durch mindestens einen Testfall durchlaufen werden.
- Wird oftmals als Minimalkriterium definiert.
- Sollte immer angestrebt werden.

Zweigüberdeckung

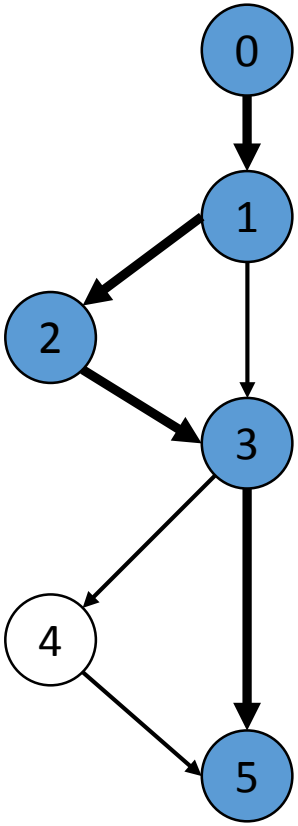
```
int manhattan(int a, int b){

    if(a<0){
        a=-a;
    }
    if(b<0){
        b=-b;
    }

    return a+b;
}
```



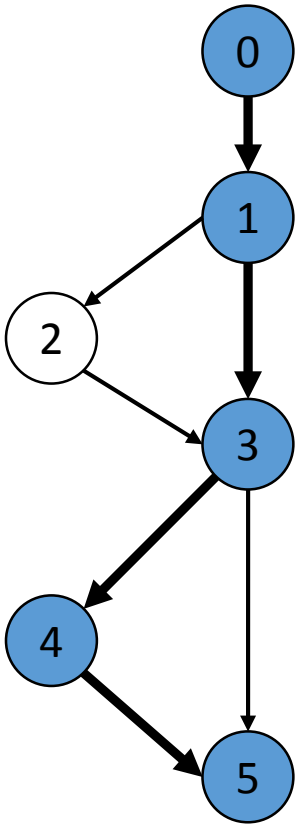
Aufgabe: Definieren Sie Testfälle, so dass eine Zweigüberdeckung von 100 % erfüllt ist!

Zweigüberdeckung – Bsp manhattan

manhattan(-1,1)



manhattan(1,-1)



Zweigüberdeckung

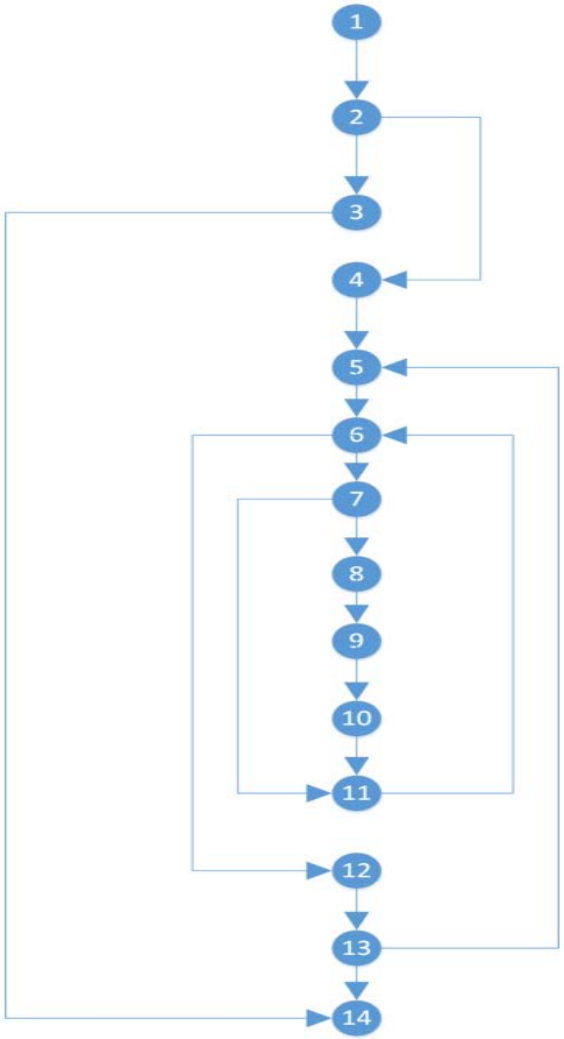
Standard RTCA DO-178B für Software Anwendungen in der Luftfahrt verlangt Zweigüberdeckung für alle Komponenten, deren Ausfall zu einer schweren, aber noch nicht katastrophalen Fehlfunktion führen kann.

Beispiel für Zweig Überdeckung

Erstellen Sie einen Kontrollfluss Graphen für das Programm aus dem letzten Kapitel und definieren Sie eine möglichst niedrige Anzahl von Testfällen, so dass die Zweig Überdeckung zu 100% gegeben ist.

Übungsbeispiel - Kontrollflussgraph

```
1  static void bubbleSort( int [] array )
2  {
3      if( array == null || array.length == 0 )
       {
           return;
       }
4      int n = array.length;
5      do
       {
6          for( int i = 0; i < n - 1; i++ )
           {
7              if( array[ i ] > array[ i + 1 ] )
               {
8                  int tmp = array[ i ];
9                  array[ i ] = array[ i + 1 ];
10                 array[ i + 1 ] = tmp;
               }
11         }
12         n--;
13     }
       while( n >= 0 );
14 }
```



Übungsbeispiel - Testfälle

Durch folgende Testfälle ist die Zweigüberdeckung zu 100% gegeben:

1. bubbleSort(null)
2. bubbleSort(new int[]{2,1,3})

Überdeckungskriterien

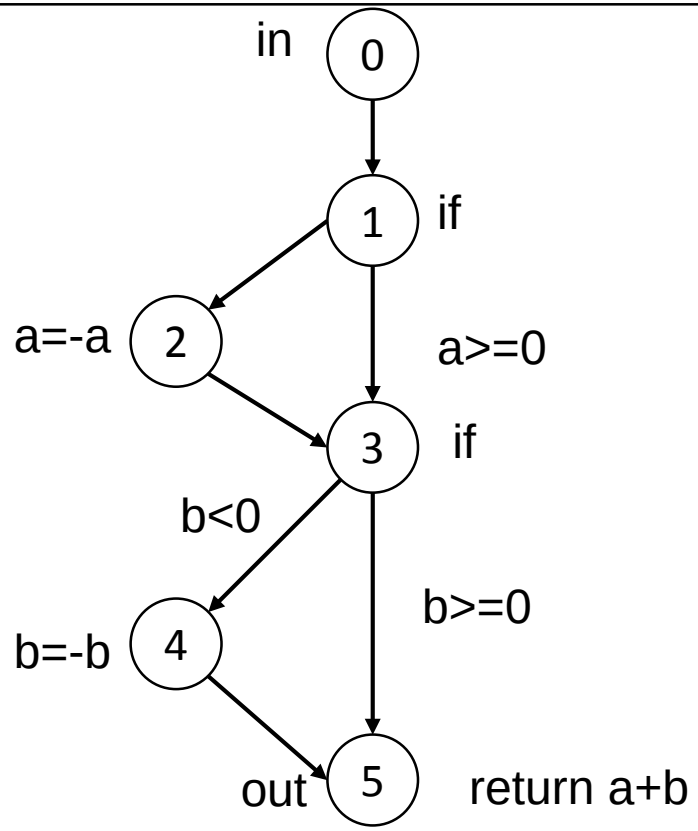
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

Pfadüberdeckung

- Dann erfüllt, wenn für jeden möglichen Pfad von Eingangsknoten zu Ausgangsknoten ein separater Testfall existiert.
- Mächtigste White Box Prüftechnik
- Anzahl der Pfade explodiert schnell, insbsd. wenn Schleifen vorhanden sind.

Pfadüberdeckung

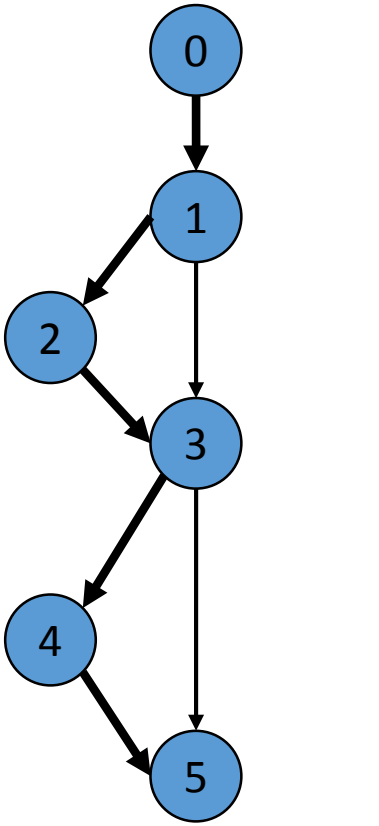
```
int manhattan(int a, int b){

    if(a<0){
        a=-a;
    }
    if(b<0){
        b=-b;
    }

    return a+b;
}
```



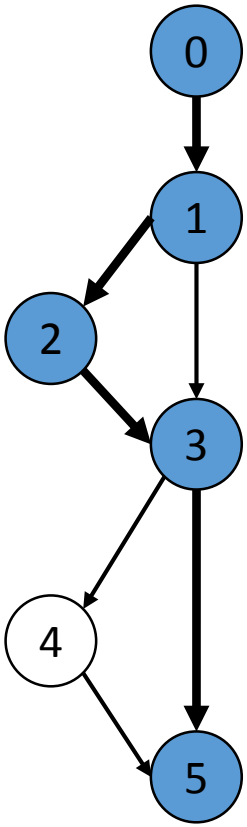
Aufgabe: Definieren Sie Testfälle, so dass eine Pfadüberdeckung von 100 % erfüllt ist!

Pfadüberdeckung- Bsp manhattan

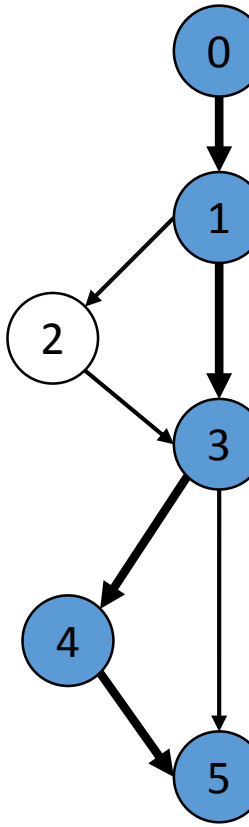
manhattan(-1,-1)



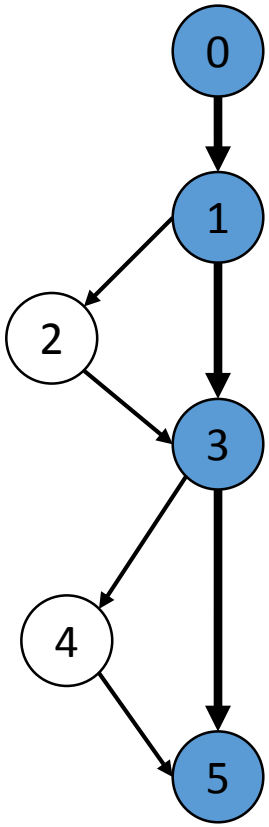
manhattan(-1,1)



manhattan(1,-1)

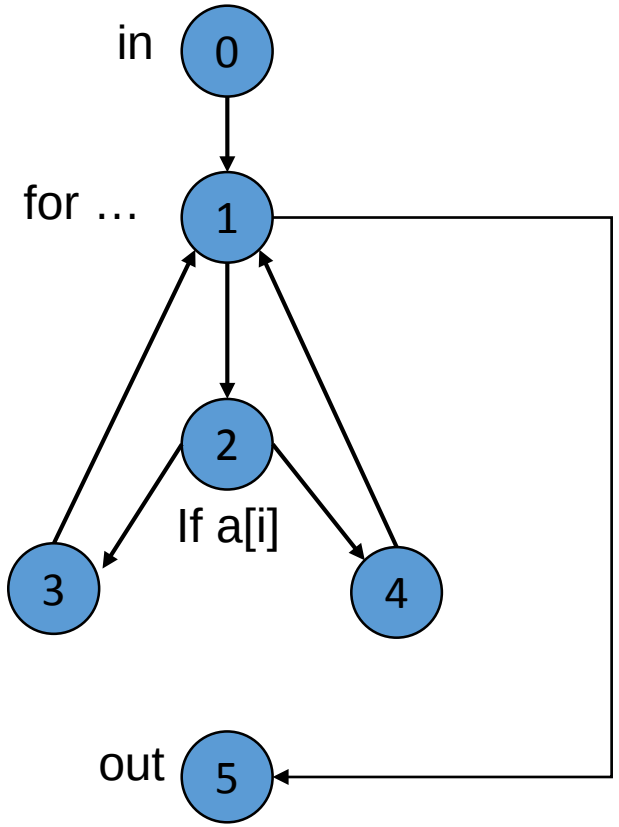


manhattan(1,1)



Probleme der Pfadüberdeckung - Bsp

```
0  void foobar(int *a)
   {
1      for(int i=0;i<512;i++){
2          if(a[i]){
3              foo();
           }
           else{
4              bar();
           }
       }
5  }
```



➔ 2^{512} verschiedene Pfade

Varianten der Pfadüberdeckung

- Pfadüberdeckung in der Praxis normalerweise nicht realisierbar, nur theoretisch interessant.

- ➔ Varianten, um die Anzahl der Testfälle zu reduzieren.

Idee: es ist nicht nötig, Schleifen öfter als ein paar Iterationen zu durchlaufen um fast alle Fehler zu finden.

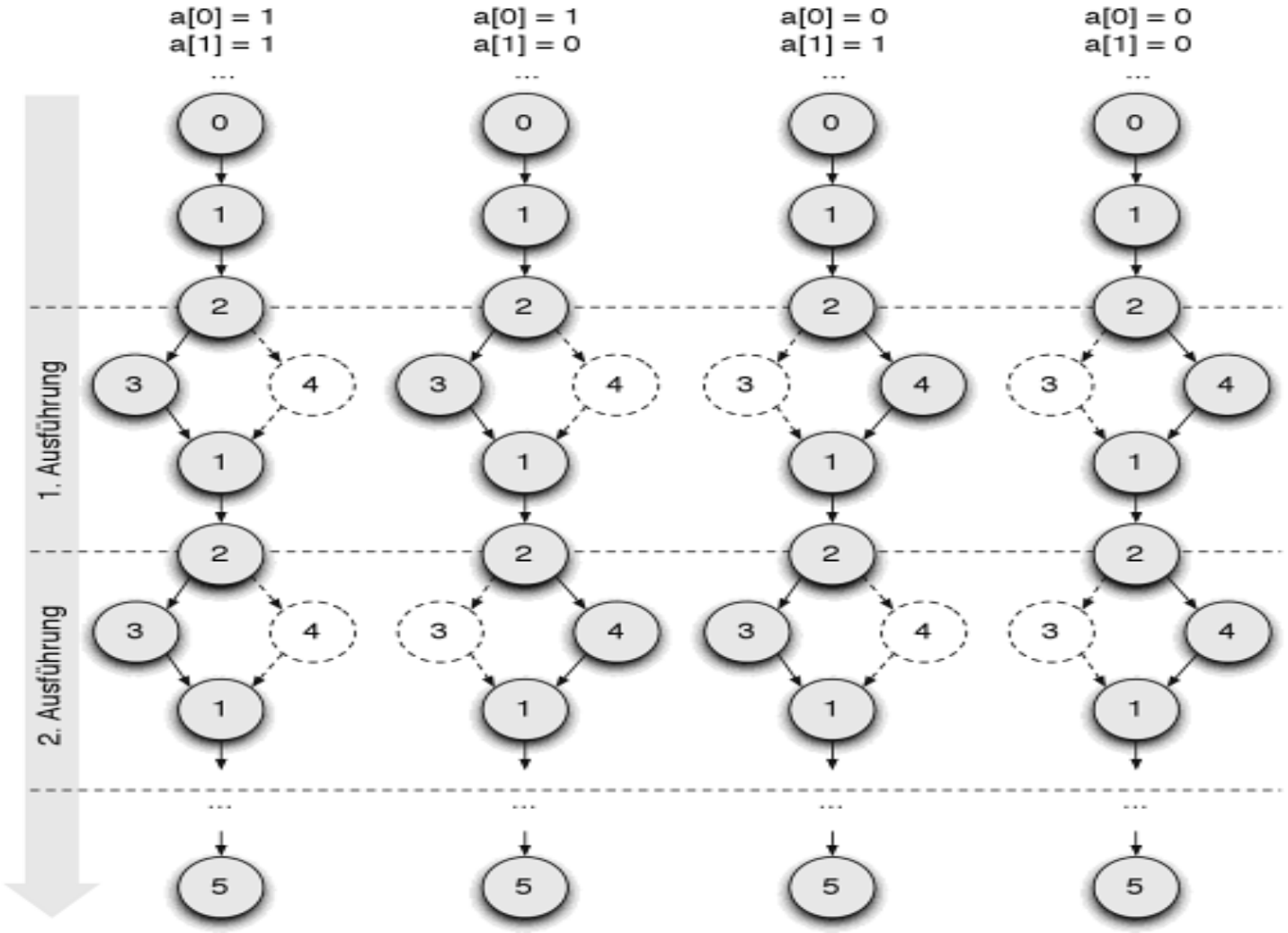
Varianten der Pfadüberdeckung

Boundary Interior Pfadüberdeckung:

Für jede Schleife drei Gruppen von Testfällen:

- **Äußere Pfade:**
  Schleifen werden nicht betreten.
- **Grenzpfade**
  Genau eine Iteration.
- **Innere Pfade**
  Mindestens eine weitere Iteration. Testfälle werden so gewählt, dass innerhalb der ersten beiden Iterationen alle möglichen Pfade abgearbeitet werden.

Bsp: Interior Pfade der Funktion foobar



Varianten der Pfadüberdeckung

Strukturierte Pfadüberdeckung:

Verallgemeinerung des Boundary-Interior Tests:

Es werden alle möglichen Ausführungspfade bis zur k-ten Schleifeniteration durchlaufen (statt nur die ersten beiden).

Immer noch sehr viele Pfade, insbsd. bei verschachtelten Schleifen)

➔ **Modifizierung:** k Iterationen nur für Schleifen, die keine inneren Schleifen mehr enthalten.

Überdeckungskriterien

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

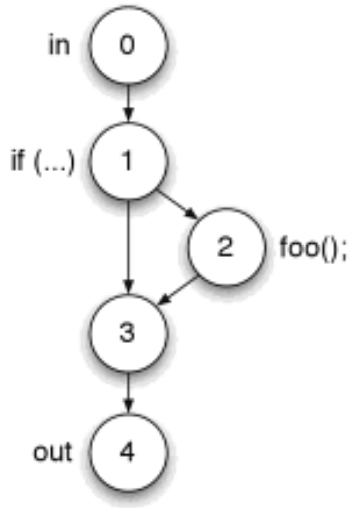
Achtung bei der Zweigüberdeckung

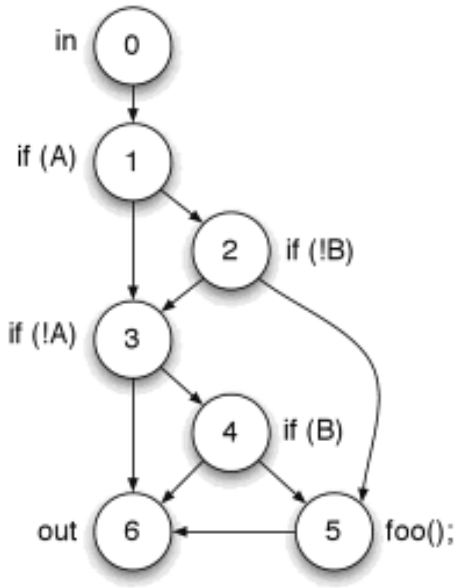
Variante 1

```
1  if ((A && !B) || (!A && B))
2  {
3      foo();
4  }
5
6
7
8
```

Variante 2

```
1  if (A)
2      if (!B)
3          goto foo:
4  if (!A)
5      if (B)
6          goto foo:
7  return;
8  foo: foo();
```





Identische Funktionalität, Unterschiedliche Implementierung
➔
Unterschiedliche Anzahl von Testfällen

➔ Bedingungsüberdeckung

Zusätzlich zu Kontrollflussgraph: Einbeziehung der logischen Struktur der if-Bedingungen

- Einfache Bedingungsüberdeckung
- Minimale Mehrfachbedingungsüberdeckung
- Mehrfachbedingungsüberdeckung

Varianten der Bedingungsüberdeckung

```
if ((A && !B) || (!A && B))
```

Einfache Bedingungsüberdeckung

„Alle atomaren Prädikate müssen mindestens einmal beide Wahrheitswerte annehmen"	A = 0, B = 1 A = 1, B = 0

Minimale Mehrfachbedingungsüberdeckung

„Alle atomaren und zusammengesetzten Prädikate müssen mindestens einmal beide Wahrheitswerte annehmen"	A = 0, B = 0 A = 0, B = 1 A = 1, B = 0

Mehrfachbedingungsüberdeckung

„Alle Wahrheitskombinationen müssen getestet werden"	A = 0, B = 0 A = 0, B = 1 A = 1, B = 0 A = 1, B = 1

Überdeckungskriterien

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

Weitere Überdeckungskriterien

- **McCabe Überdeckung**
  kontrollflussorientiertes Verfahren.
- **Defs Uses Überdeckung**
  Ableitung der Testfälle aus dem Datenfluss.
- **Required k-Tupel Überdeckung**
  Ebenfalls Ableitung der Testfälle aus dem Datenfluss.

Hier nicht weiter betrachtet

Nachzulesen unter Dirk W. Hoffmann: Software-Qualität, 2 Auflage, Springer Vieweg

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests

Testmetriken

Das Projekt hat Testverfahren und Testfälle definiert. Jetzt stellt sich die Frage:

Wie gut machen wir das?

- Wird ein Modul ausreichend getestet?
- Wie viele unentdeckte Fehler enthält das Programm?
- Wie leistungsfähig ist ein gegebenes Testverfahren?

➔**Quantifizierung durch Testmetriken**

Überdeckungsmetriken

Im industriellen Bereich im Wesentlichen relevant:

- **Anweisungsüberdeckung**

M_{C0}=(Anzahl der überdeckten Knoten)/(Anzahl der Knoten) * 100 [%]

- **Zweigüberdeckung**

M_{C1}=(Anzahl der überdeckten Kanten)/(Anzahl der Kanten) * 100 [%]

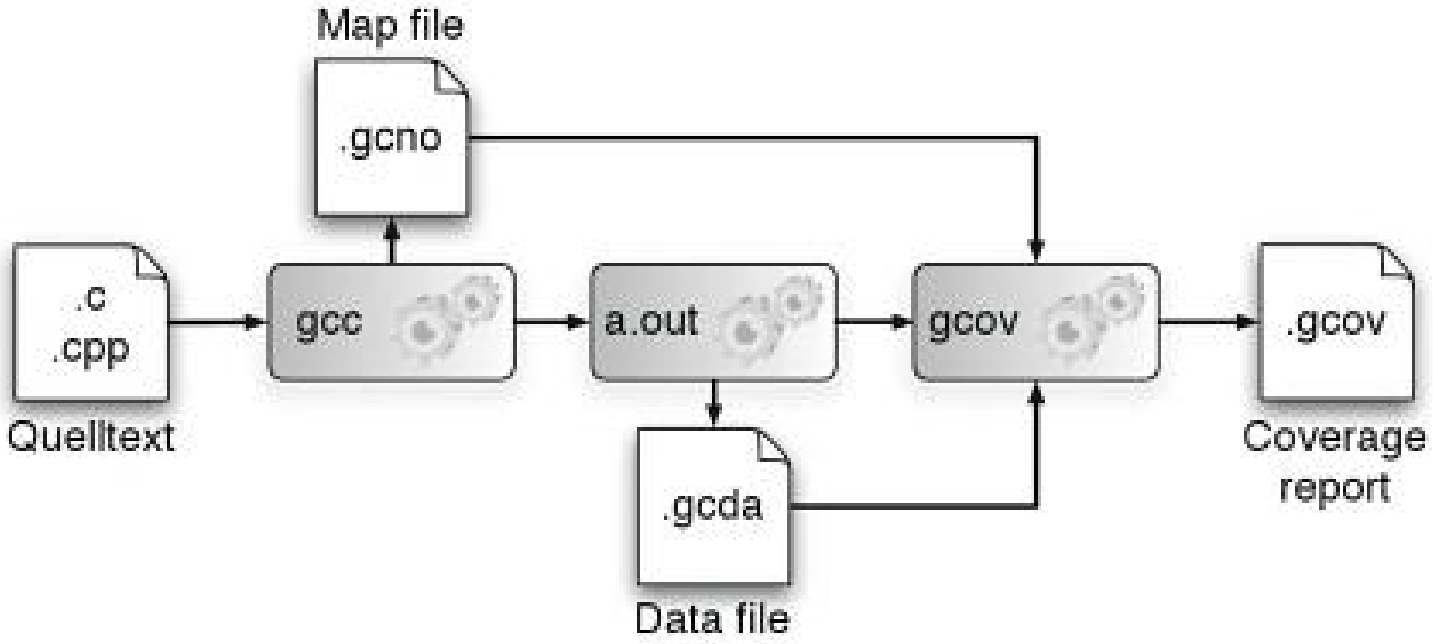
Überdeckungsmetriken

Verwendung:

- Abnahme für Module: Abnahme erst bei Testüberdeckung größer als vereinbarte Schwelle.

- Vergleich von Modulen, um Schwächen in der Testumgebung aufzudecken und Testressourcen zielgerichtet zu planen.

Überdeckungsmetrik – Bsp gcov



Toolunterstützung bei Überdeckungsmetriken

Z.B. der Profiler gcov (Teil der GNU Compiler Collection gcc)

1. Kompilieren:

gcc -Wall -fprofile-arcs -ftest-coverage –o manhattan manhattan.c

-fprofile-arcs: to save line execution count ➔ manhattan.gcno

-ftest-coverage: to record branch statistics ➔ manhattan.gcda (nach der Ausführung)

Gcov

- Ausführen:

```
D:\OTH\Vorlesungen\MST-SS2015\C-Beispiele>manhattan 1 1
Returnvalue is 2

D:\OTH\Vorlesungen\MST-SS2015\C-Beispiele>manhattan 2 2
Returnvalue is 4

D:\OTH\Vorlesungen\MST-SS2015\C-Beispiele>manhattan 1 -1
Returnvalue is 2
```

Gcov

- **Analysieren**

gcov manhattan.c

➔ gcov manhattan.c
File ‚manhattan.c‘
Lines executed: 75.00% of 12
Manhattan.c:creating ‚manhattan.c.gcov‘

Gcov

- Analysieren:

Datei *manhattan.c.gcov* mit detaillierter Überdeckungsinformation der Testläufe.

Gcov ermittelt die **Zeilenüberdeckung, nicht die Anweisungsüberdeckung !**

Code Coverage Tools

- Gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov
- gcov kurz: http://en.wikipedia.org/wiki/Gcov
- Code Coverage Tools im Java Umfeld:
  - http://c2.com/cgi/wiki?CodeCoverageTools
  - http://en.wikipedia.org/wiki/Java_Code_Coverage_Tools

Mutationstest

- Verfahren, um verschiedene Testverfahren quantitativ zu bewerten

Idee: Füge an zufälligen Stellen Fehler ins Programm und prüfe, wieviele davon durch die Tests gefunden werden.

Details in Hoffmann, Seite 218, hier nicht weiter behandelt.

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Grenzen des Software Tests

Software Tests sind oft schwierig und aufwändig.

Gründe

- Unklare oder fehlende Anforderungen
- Programmkomplexität
- Mangelnde Werkzeugunterstützung bei der Konstruktion
- Ausbildungs- und Fortbildungsdefizite
- Zeitprobleme

Schwierigkeit beim Test - Bsp

Kein Problem

```
float foo(unsigned char x)
{
    unsigned char i = 0;
     while (i<6){
        i++;
        x /=2;
     }
     return 1.0/(x-i);
}
```

Division durch 0
Bei bestimmten Eingabewerten

```
float foo(unsigned short x)
{
    unsigned char i = 0;
     while (i<6){
        i++;
        x /= 2;
     }
     return 1.0/(x-i);
}
```