

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

- 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

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

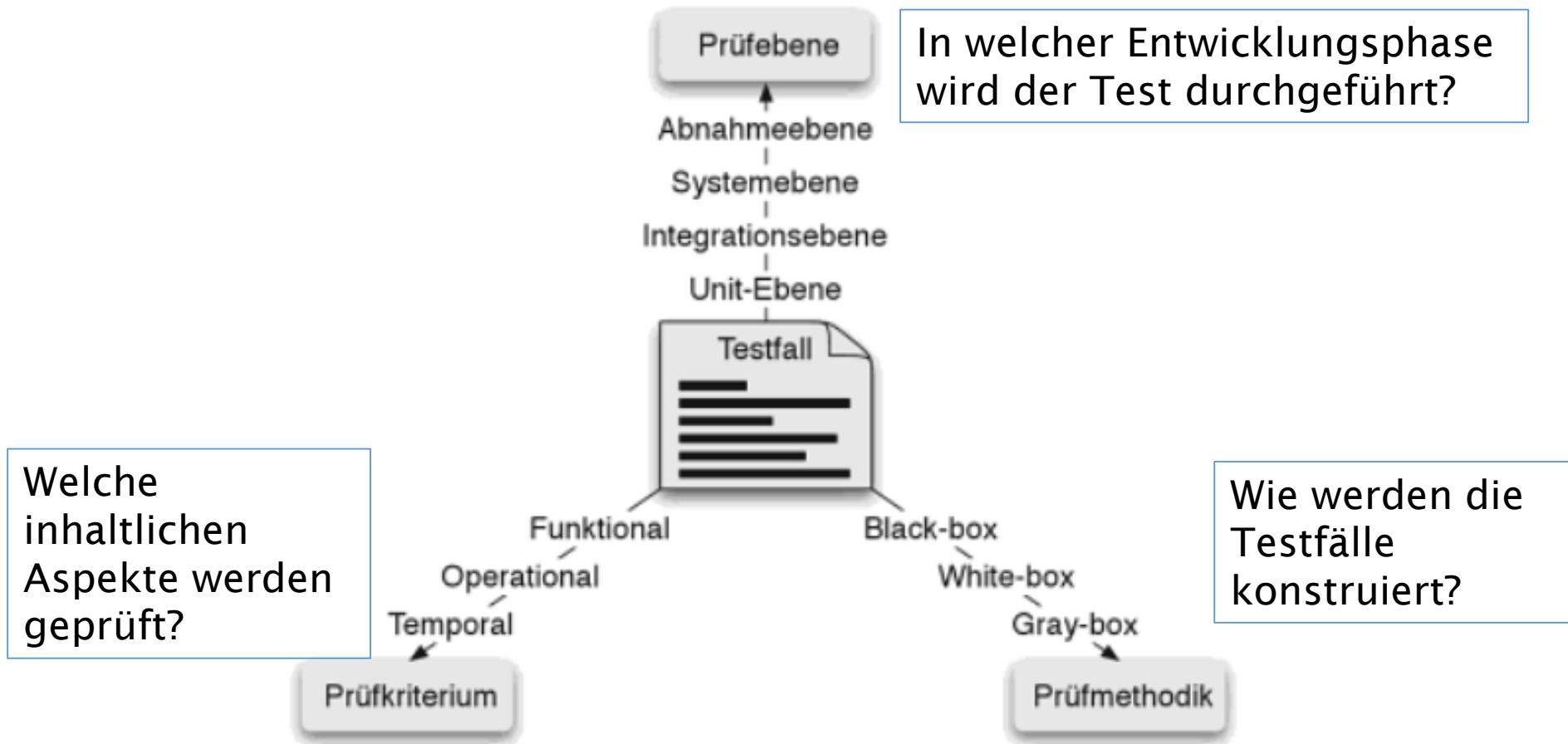
- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests

# Motivation

- Siehe die Folien zu Beginn der Vorlesung
- Siehe [http://de.wikipedia.org/wiki/Liste\\_von\\_Programmfehlerbeispielen](http://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen)
- [https://jaxenter.de/top-10-der-software-katastrophen-181?utm\\_source=nl&utm\\_medium=email](https://jaxenter.de/top-10-der-software-katastrophen-181?utm_source=nl&utm_medium=email)

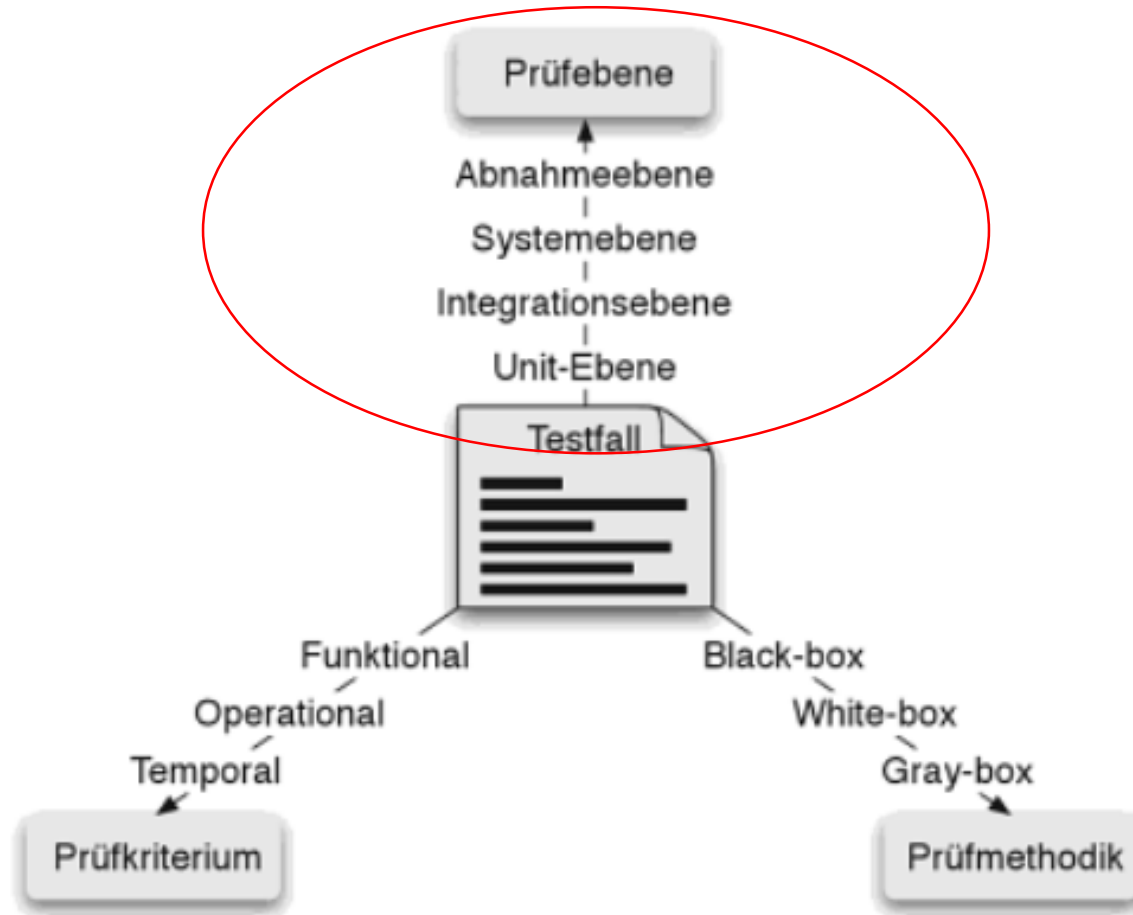
- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

# Testklassifikation



## Merkmalsräume der Testklassifikation

# Testklassifikation

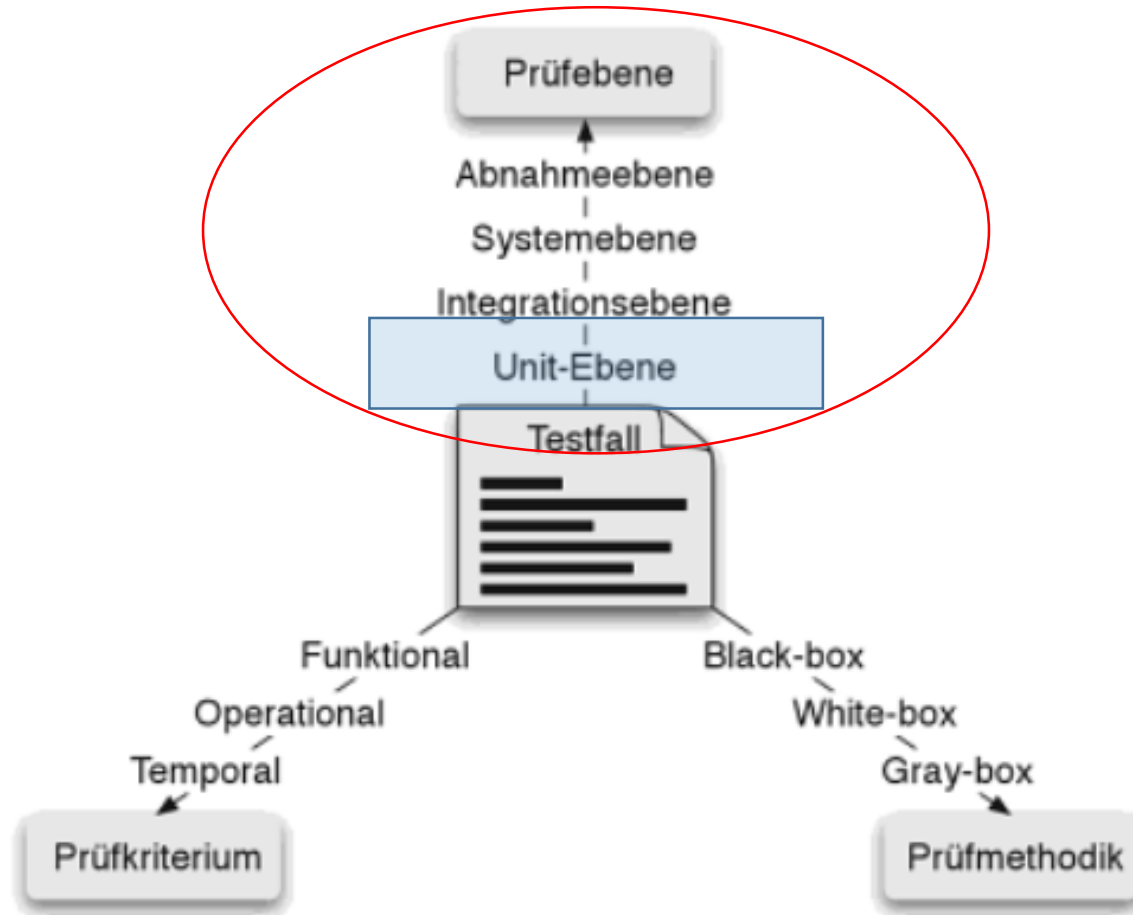


## Merkmalsräume der Testklassifikation



- **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

- Sie testen einzelne Einheiten isoliert.
- 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-Frameworks (z.B. JUnit) um Tests zu schreiben und durchzuführen.
- ➔ Ermöglicht, bei jeder Änderung **ALLE** Tests laufen zu lassen und das Ergebnis graphisch anzuzeigen.

## Komponenten:

- **Aufbau:**

Sie definieren die Testfälle mit Eingaben und erwarteten Ergebnissen.

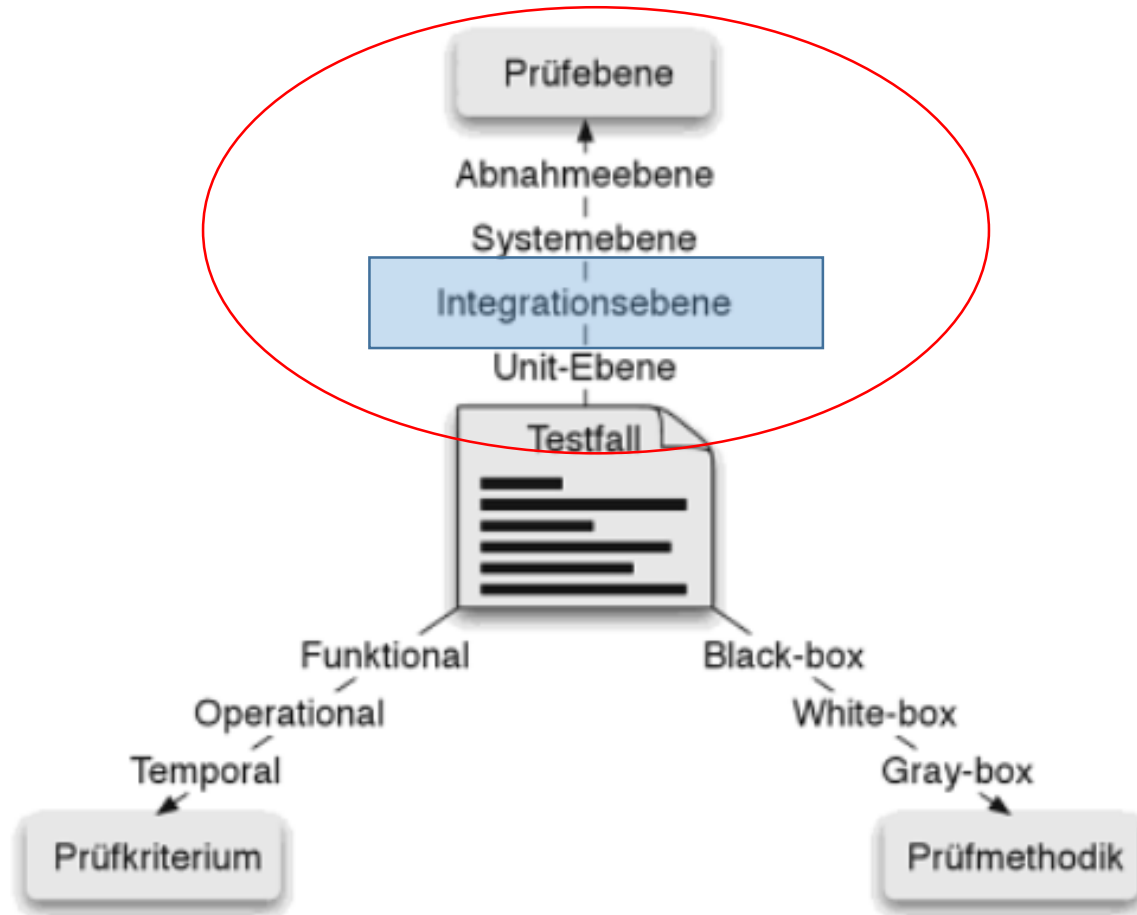
- **Aufruf**

Sie rufen die zu testenden Objekte oder Methoden auf.

- **Auswertung**

Vergleich des wirklichen mit dem erwarteten Ergebnis.

# Integrationstests



## Merkmalsräume der Testklassifikation

- 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.



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

## 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:

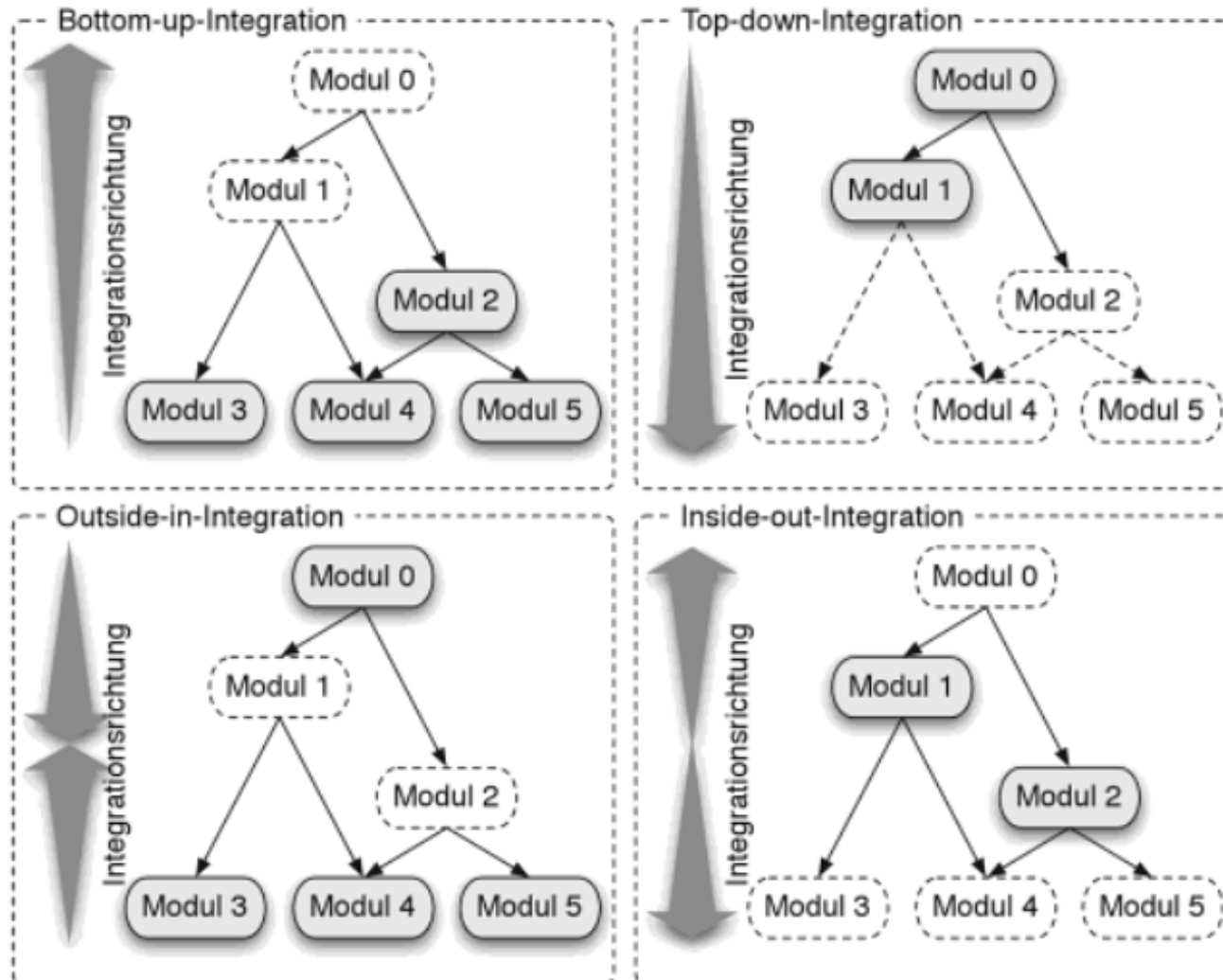
- Testtreiber und Mocks sind nicht nötig.

## Strukturorientierte Integration

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

- **Bottom Up** – Ausgangspunkt: 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

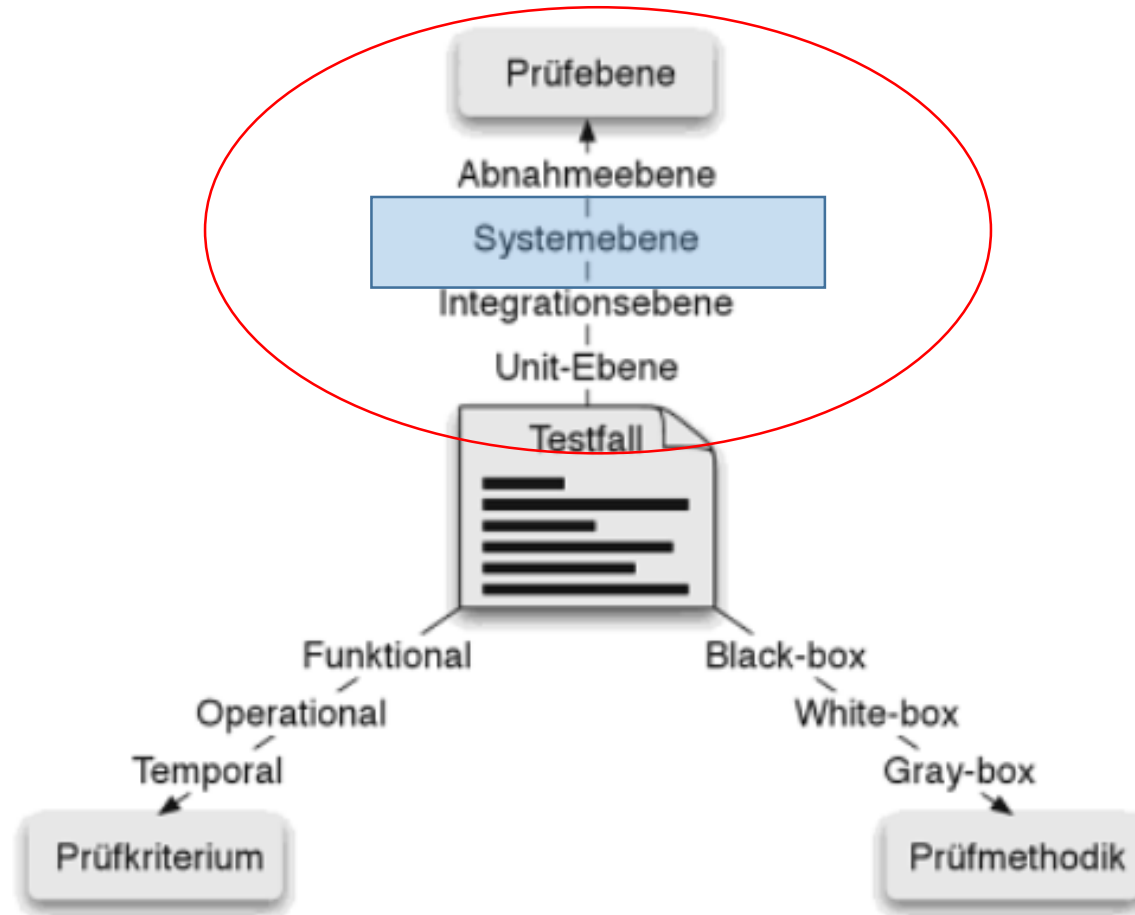


## **Funktionsorientierte Integration**

### 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

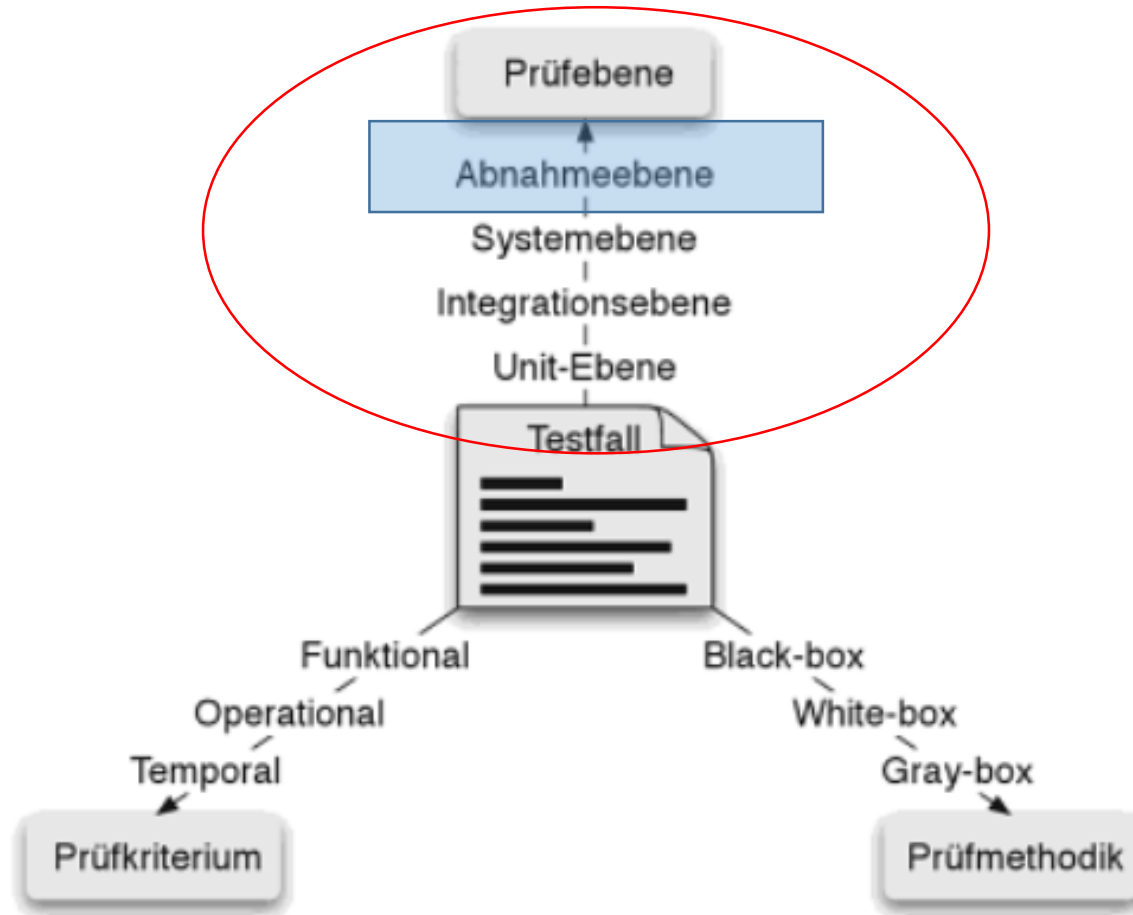
- Start, sobald alle Komponenten erfolgreich integriert sind.
- Nahezu ausschließlich aus funktionaler Sicht.
- Oftmals genauso aufwändig wie Unit und Integrationstest gemeinsam.

## Erschwerende Faktoren

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

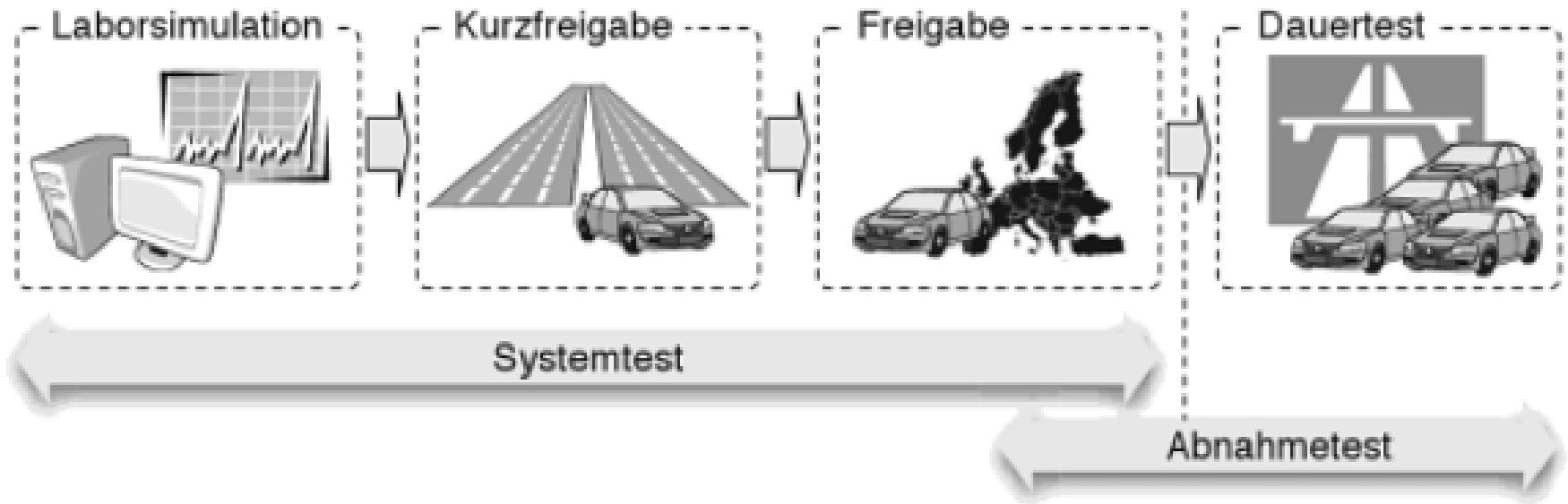


# Integrationstests



## Merkmalsräume der Testklassifikation

# Systemtest - Abnahmetest



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

Abnahmetests 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

## Anstelle von Abnahmetests: Feldtests

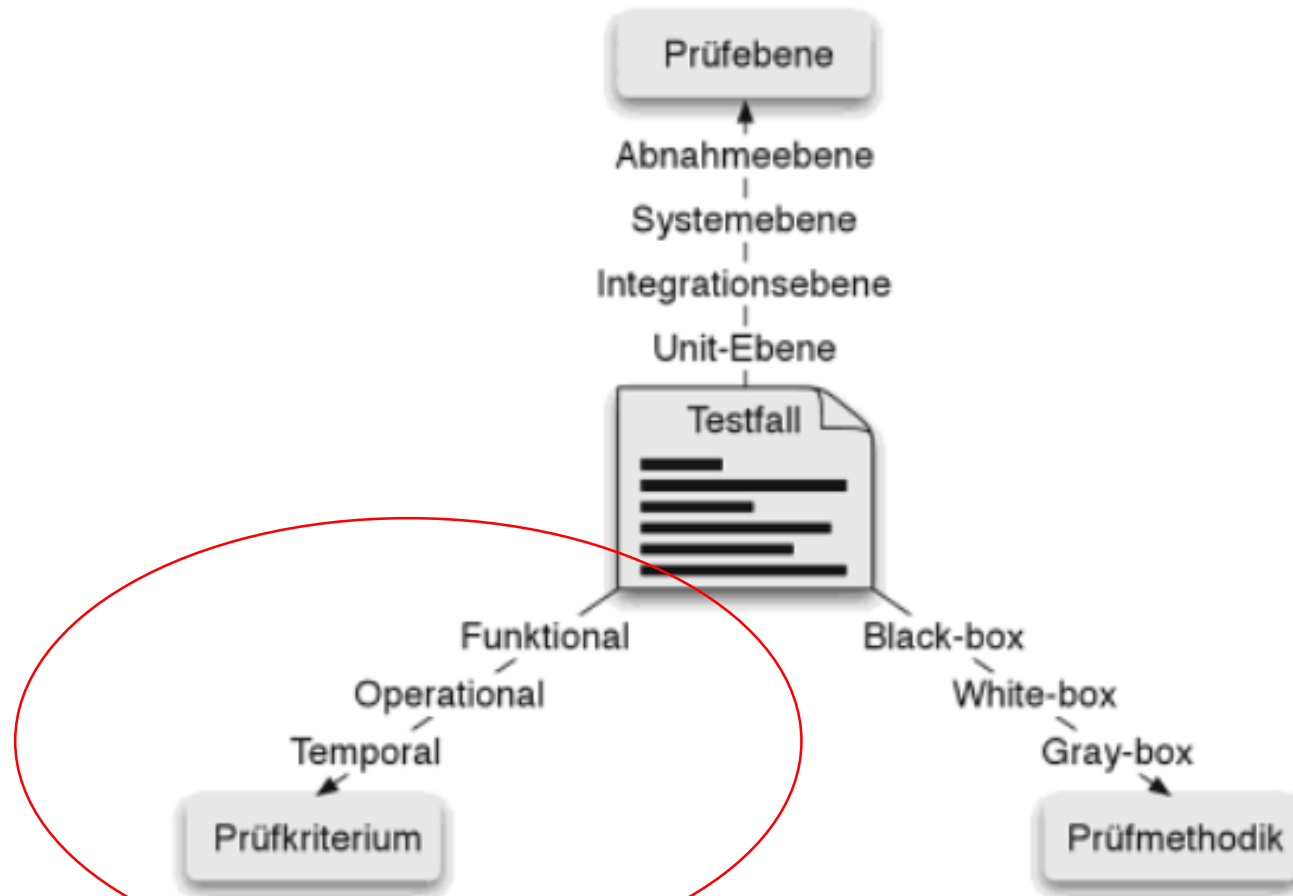
- **Alpha Tests**

In der Anwendungsumgebung des Herstellers. Durchführung durch ausgewählte Anwender.

- **Beta Tests**

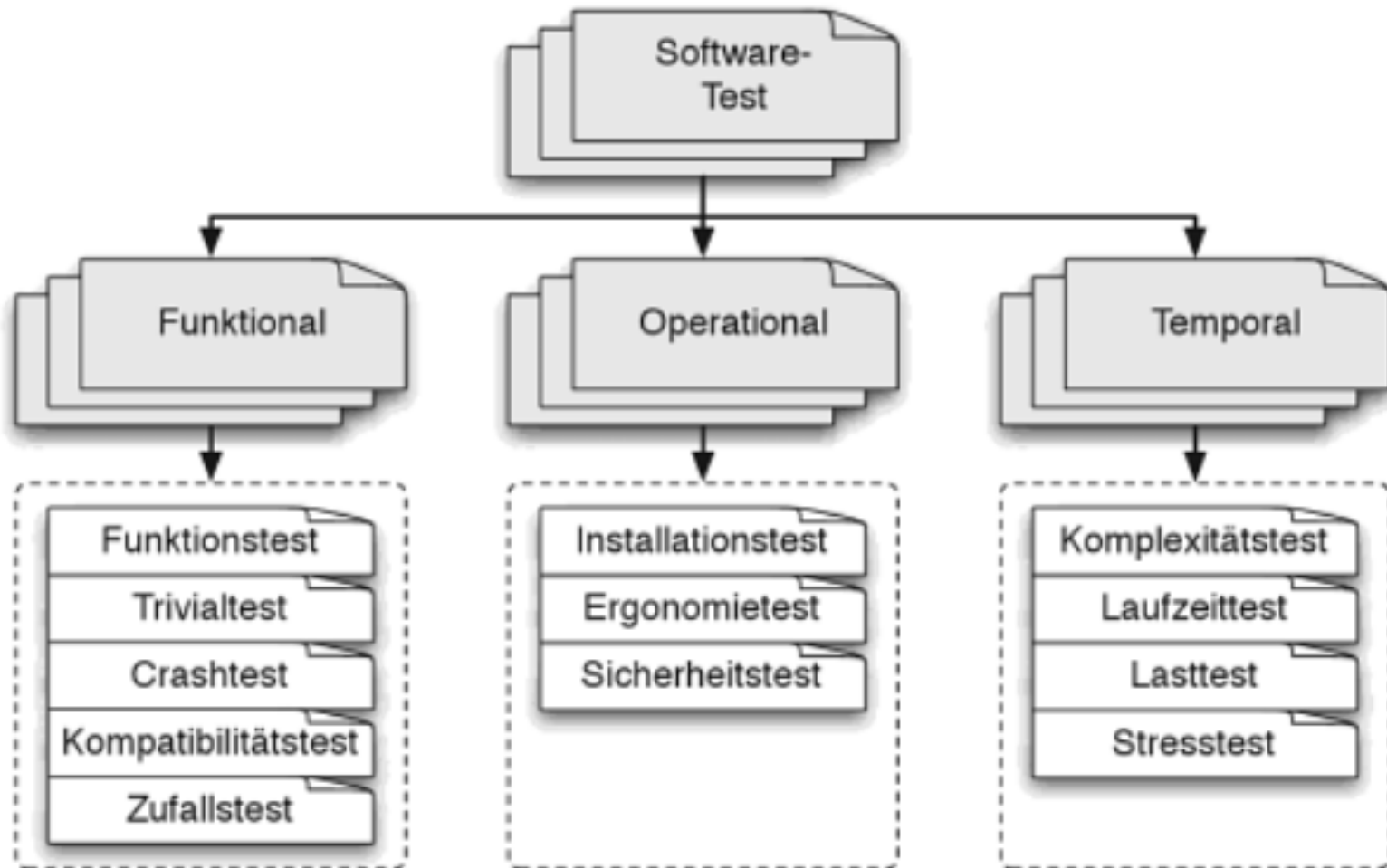
In der Umgebung des Kunden.  
Häufig inkrementell durchgeführt mit zunehmend größerem Nutzerkreis.

# Testklassifikation

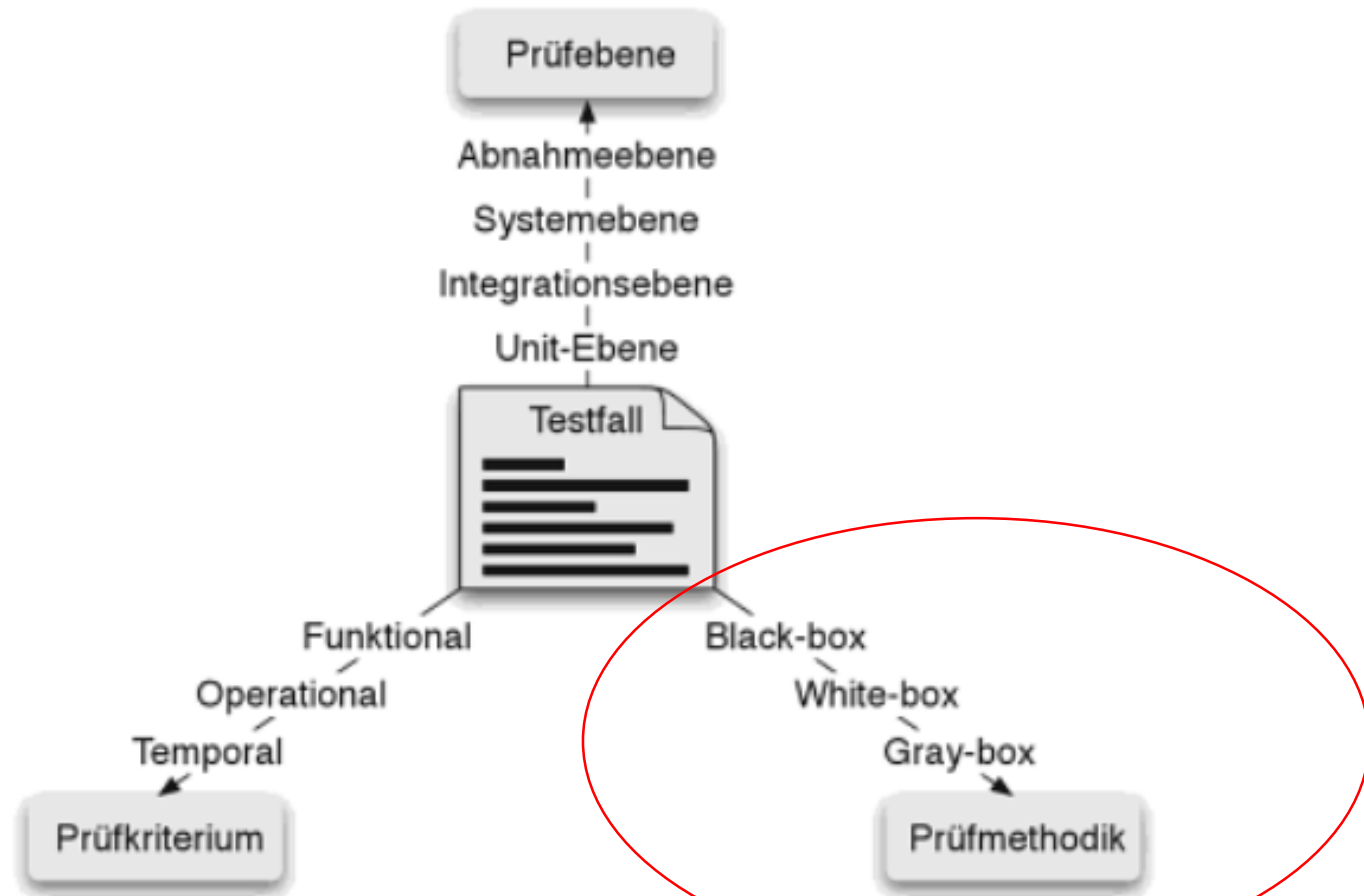


## Merkmalsräume der Testklassifikation

# Prüfkriterien



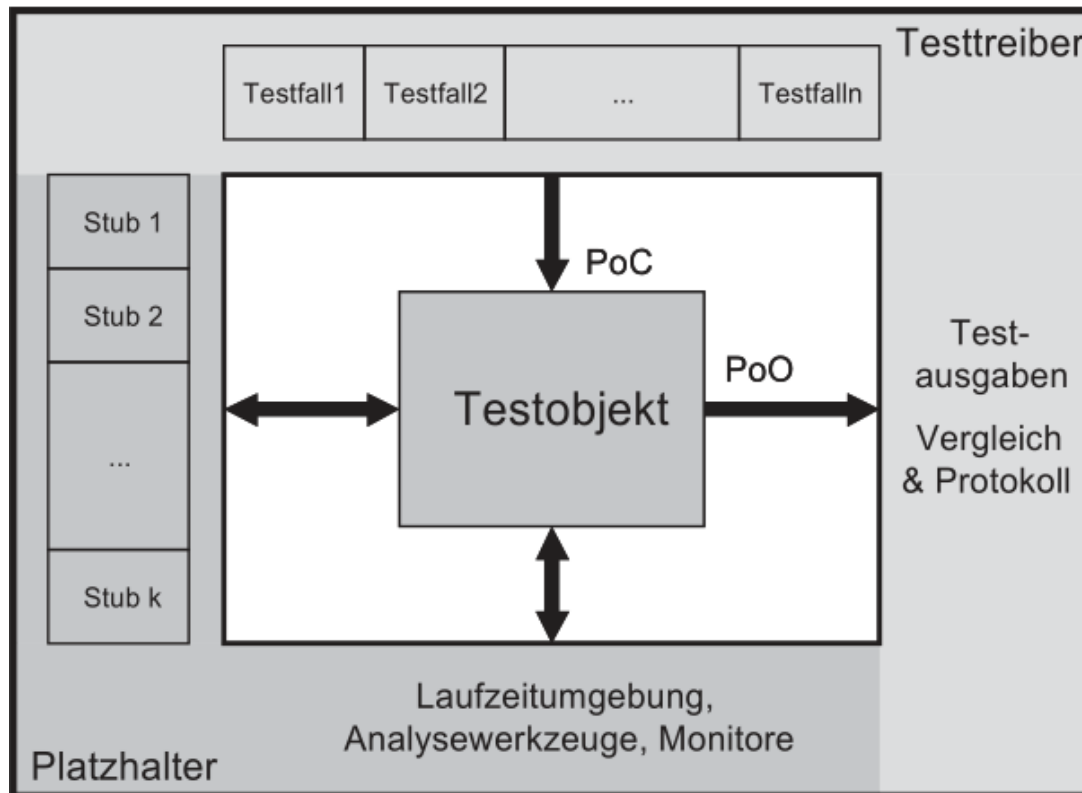
# Testklassifikation



## 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  
Observation

**PoC:**  
Point of Control



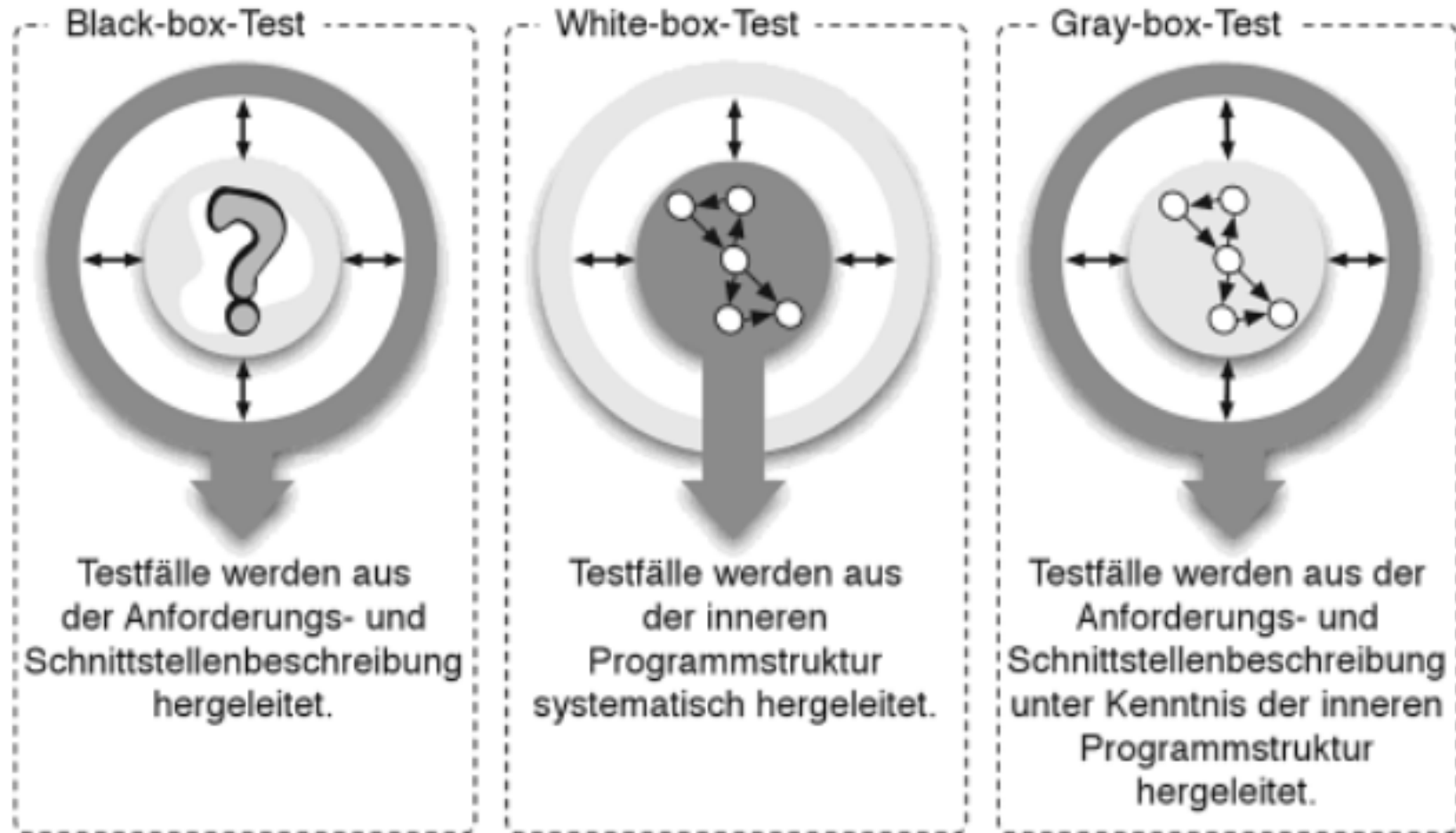
**Ziel:** Systematisches Vorgehen um mit möglichst wenig Aufwand möglichst viele Anforderungen zu überprüfen bzw. Fehler zu finden.

## Vorgehen:

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

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

# Prüftechniken im Vergleich



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

# Black Box Testverfahren

Die Verfahren werden auch als **spezifikationsbasierte Testentwurfungsverfahren** 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.

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

**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

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

Bsp `public static long abs(long a)`

**Äquivalenzklassen:**

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



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 team2 Referenz auf das zweite Team
```

```
    */
```

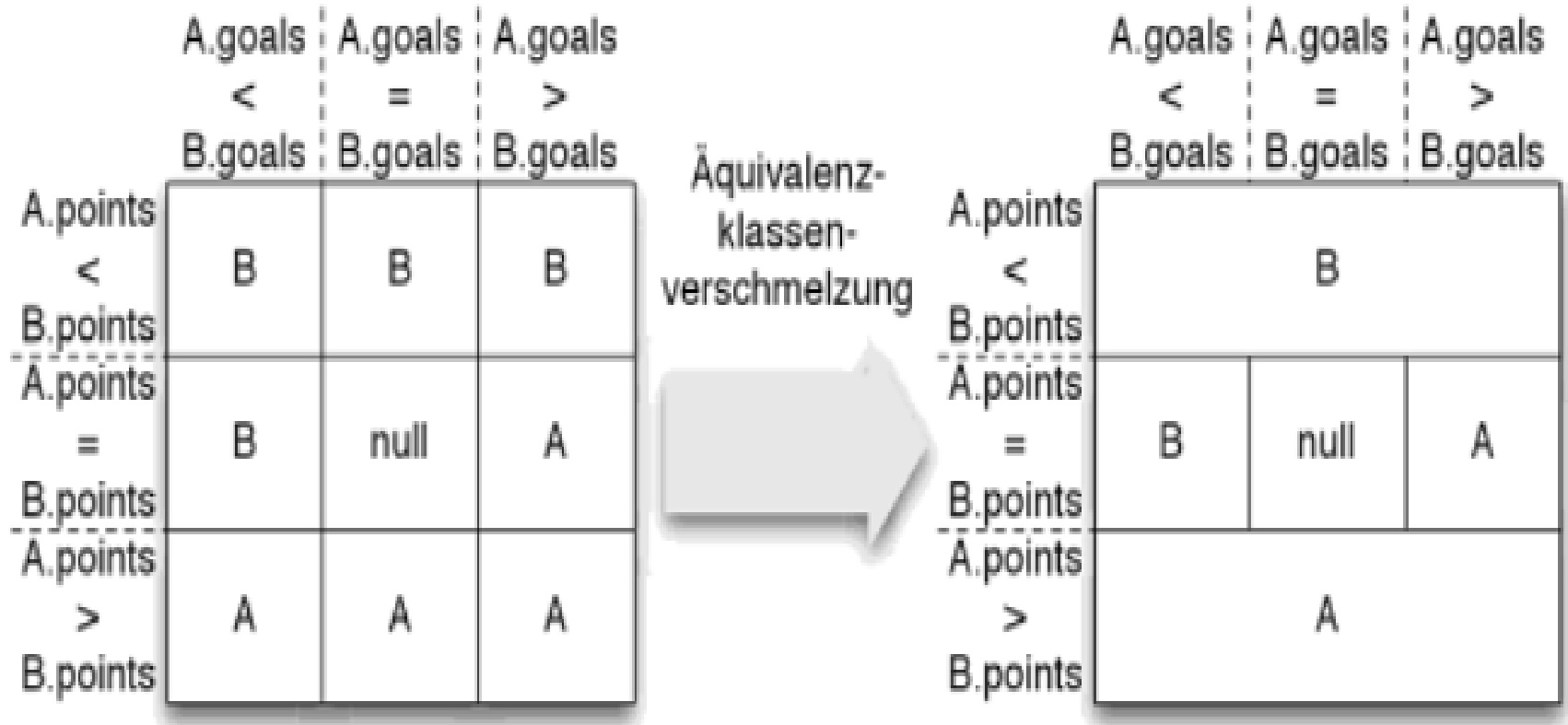
```
    public static Team calculateChampion(Team team1, Team team2){
```

```
        ~~~~~
```

```
    }
```

```
}
```

# Bsp Äquivalenzklassen



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.

## 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 <sup>a</sup> )	-4000 1500800

## → Äquivalenzklassen\_Aufgabe

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

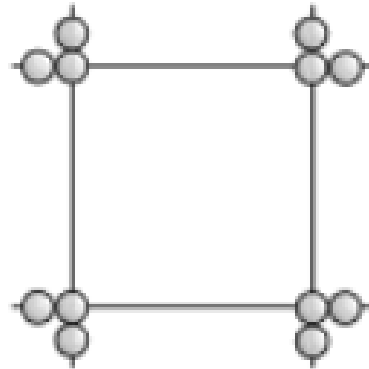


- Partitionierung identisch wie bei der Äquivalenzklassenbildung
- Testfallauswahl aber nicht beliebig innerhalb einer Äquivalenzklasse, sondern jeweils den Randwert und Tupel, bei denen ein einzelner Wert außerhalb der Äquivalenzklasse liegt.

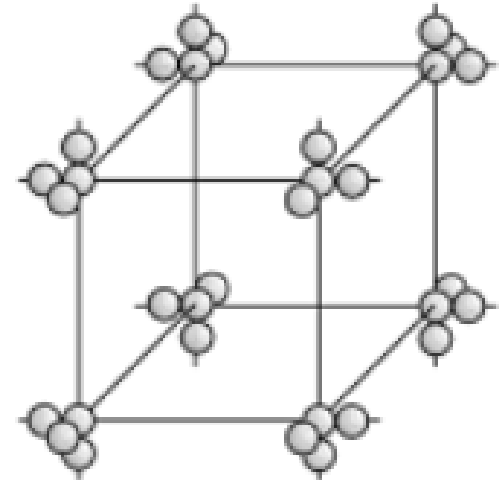
# Grenzwertbetrachtung



Eindimensionale  
Äquivalenzklasse  
4 Testfälle



Zweidimensionale  
Äquivalenzklasse  
12 Testfälle



Dreidimensionale  
Äquivalenzklasse  
32 Testfälle

Voraussetzung für diese Methode: Eingabewerte sind geordnet.

# Aufgabe

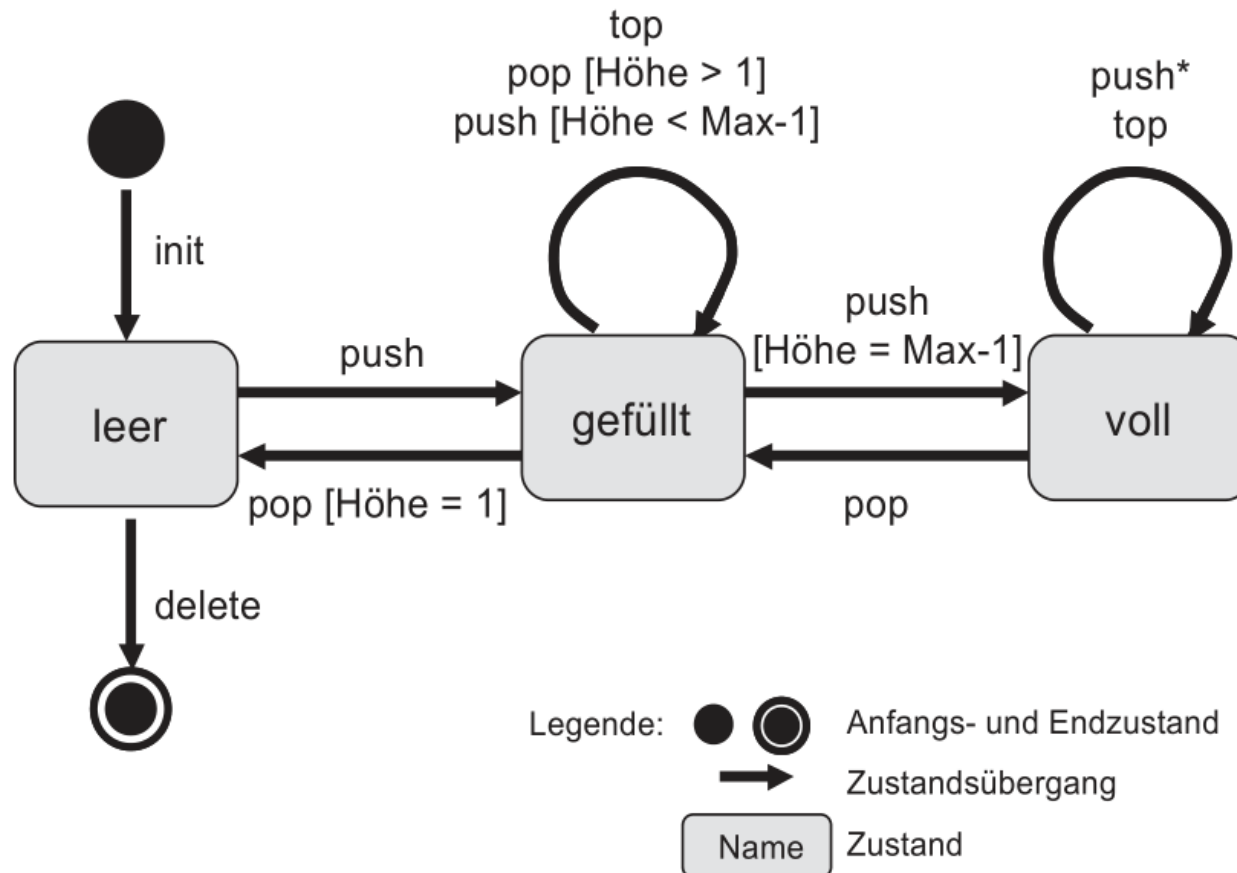
## → Grenzwert\_Aufgabe

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

- **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)



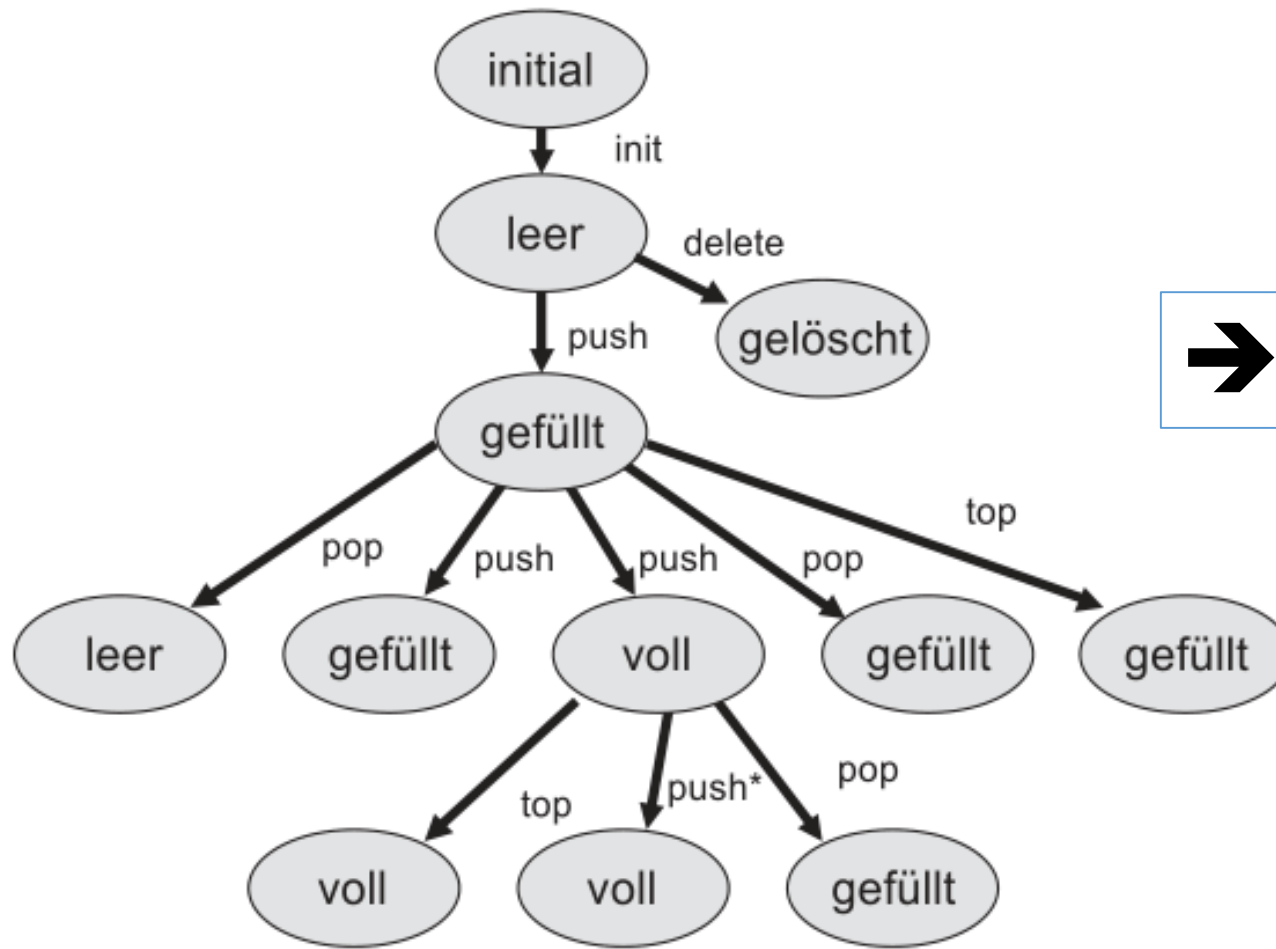
*Darstellung aus:  
A. Spillner, T.  
Linz: Basiswissen  
Softwaretest,  
dpunkt.verlag*

## 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.

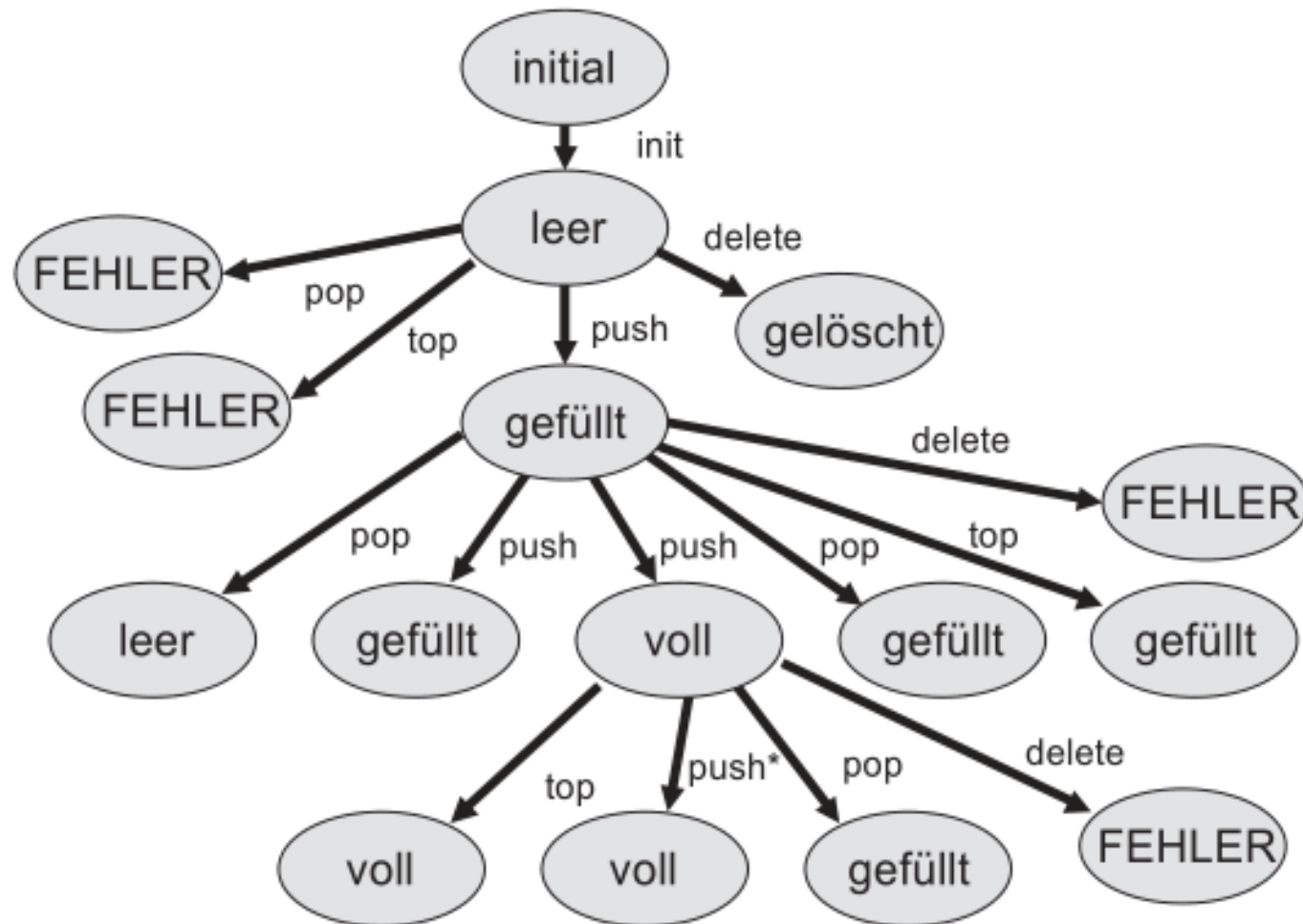
# Übergangsbaum für Stapelbeispiel



➔ 8 Testfälle

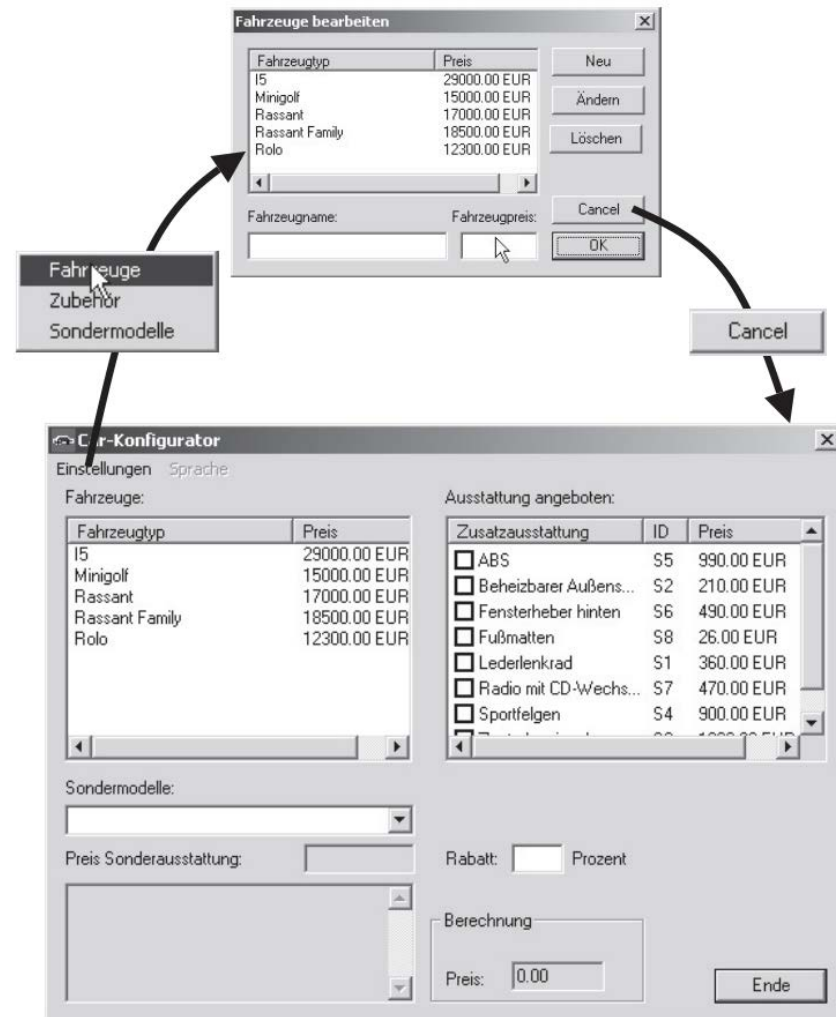


# Übergangsbaum für Robustheitstest



# Zustandsbasierter Test - Bsp

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



# Aufgabe zum Übergangsbaum

➔ Siehe Aufgabe FamilienstandsÜbergangsbaum

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

Speziell für Systemebene (Systemtest, Abnahmetest):

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

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

- **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 :**

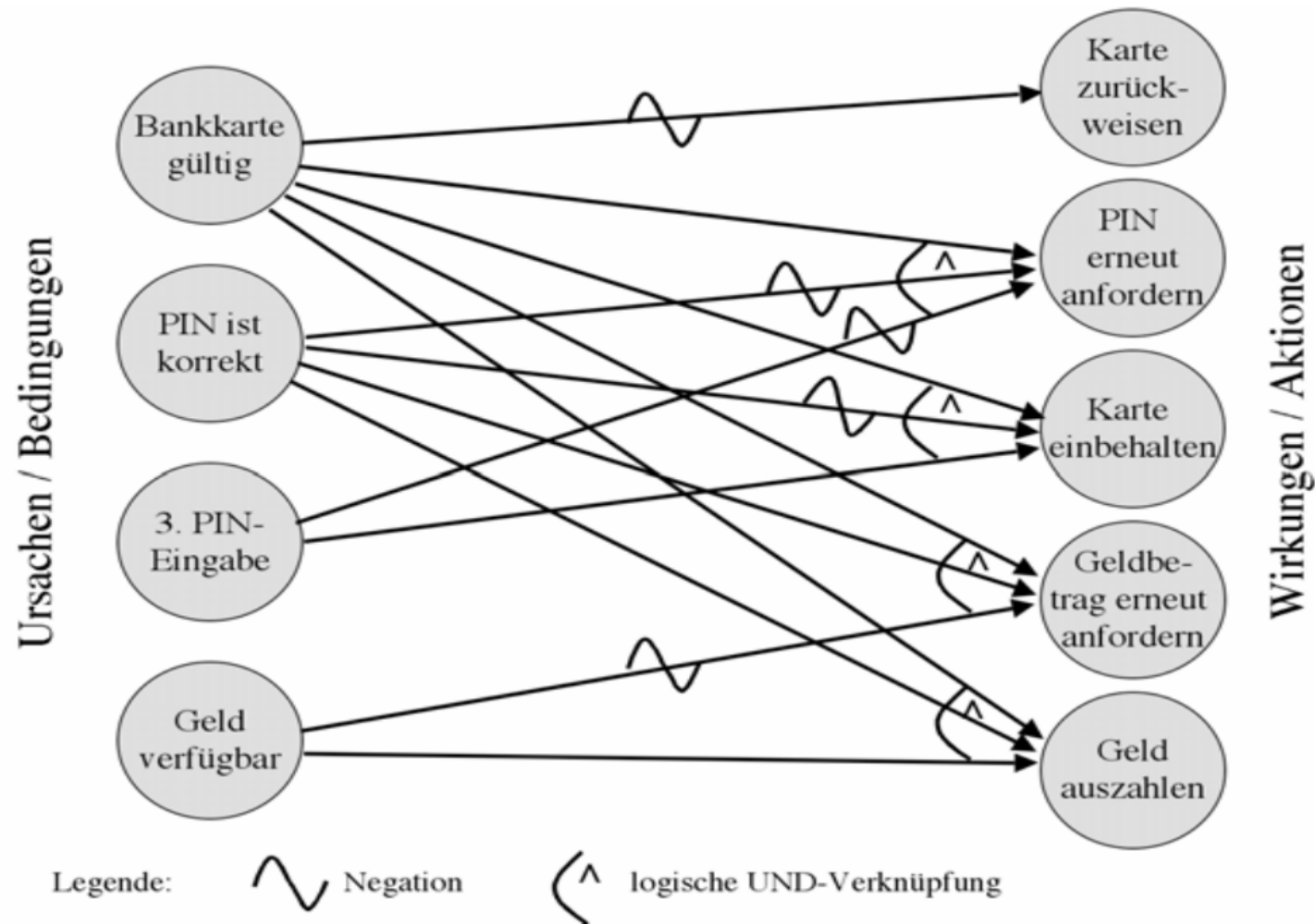
- Die Bankkarte ist gültig.
- Die PIN ist korrekt eingegeben.
- Es dürfen nur maximal drei PIN-Eingaben erfolgen.
- Geld steht zur Verfügung (im Automat und auf dem Konto).

**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
<b>Bedingungen</b>	Bankkarte gültig?	Nein	Ja	Ja	Ja	Ja
	PIN ist korrekt?	–	Nein	Nein	Ja	Ja
	Dritte PIN-Eingabe?	–	Nein	Ja	–	–
	Geld verfügbar?	–	–	–	Nein	Ja
<b>Aktionen</b>	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

- 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 kompatibel sein und im online sowie im offline Modus funktionieren.

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

→ 18 Konfigurationen,  
Bei verschiedenen  
Versionen  
entsprechend  
Mehr.

# Paarweises Testen

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



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

# 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 ziemlich schnell ziemlich komplex.

Eine Möglichkeit zur Konstruktion ist die Verwendung orthogonaler Felder.

Das Thema wird hier nicht weiter behandelt.

- 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.

## ■ 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

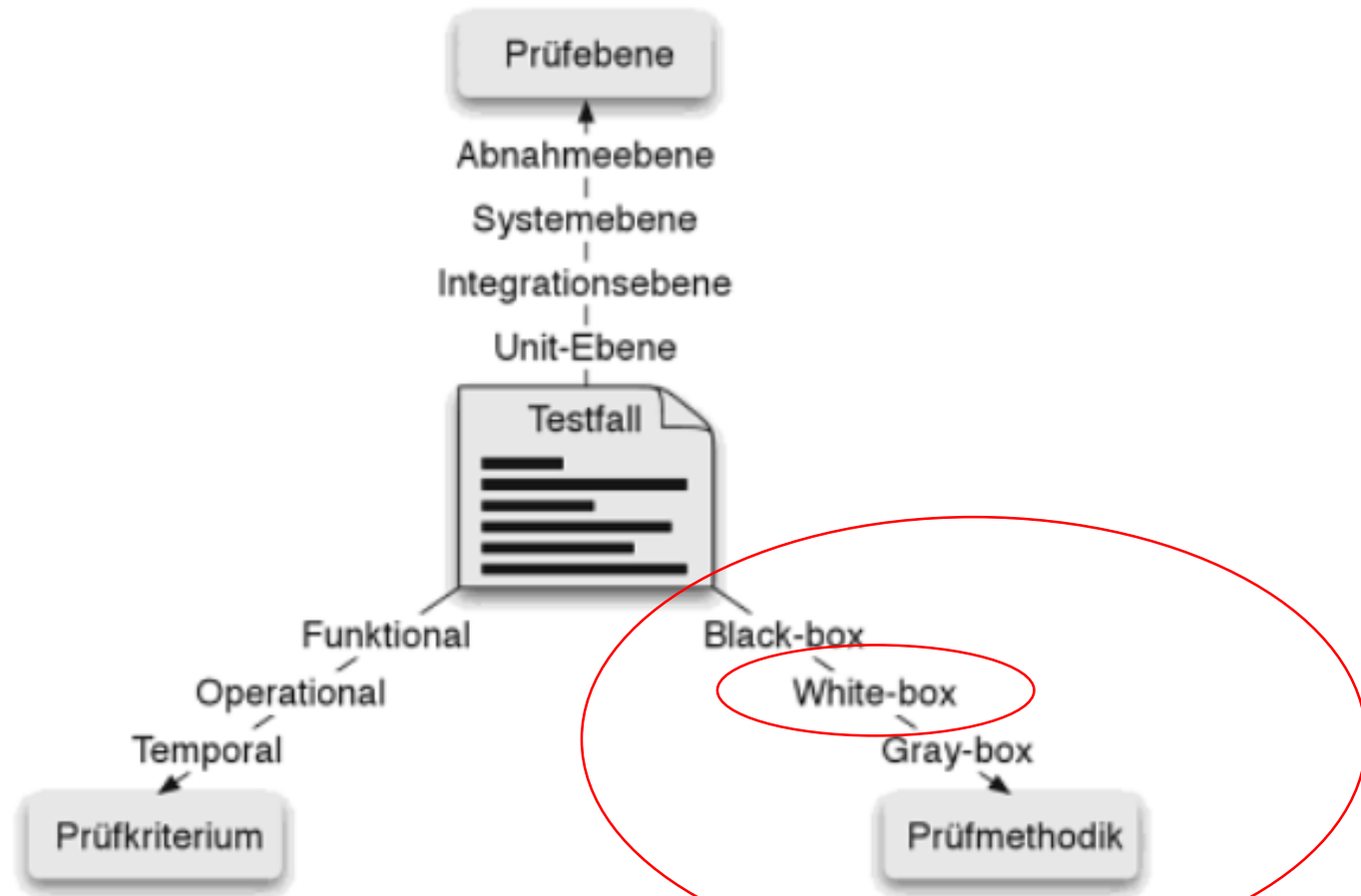
Erneuter Test eines bereits getesteten Programms bzw. einer Teilfunktionalität nach deren Modifikation mit dem Ziel, nachzuweisen, dass durch die vorgenommenen Änderungen keine Fehlerzustände eingebaut oder (bisher maskierte Fehlerzustände) freigelegt wurden.

## ■ Mutationstest

Technik, um Testverfahren zu beurteilen.

- 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

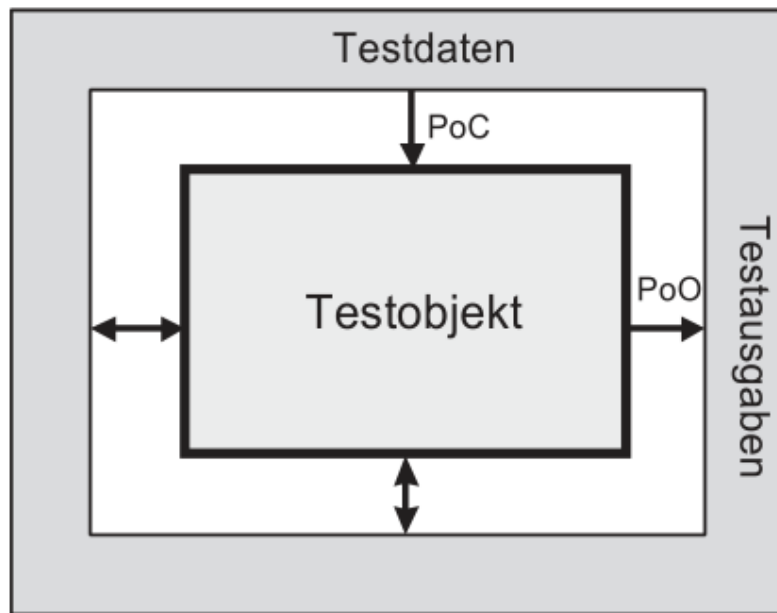
## **Nun: White Box Test:**

Basiert auf der Analyse der inneren Programmstrukturen

**White Box Test = Strukturtest**

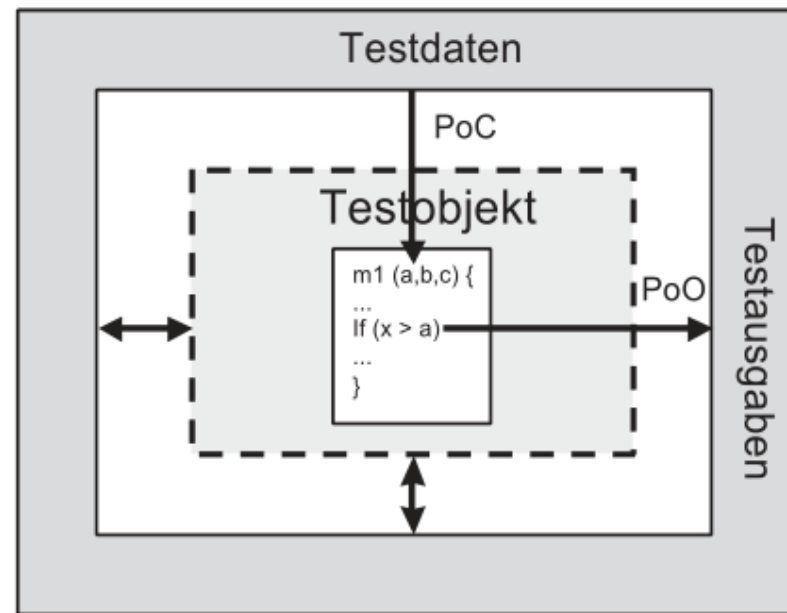
# Black Box vs White Box

## Blackbox-Verfahren



PoC und PoO »außerhalb«  
des Testobjekts

## Whitebox-Verfahren



PoC und/oder PoO »innerhalb«  
des Testobjekts



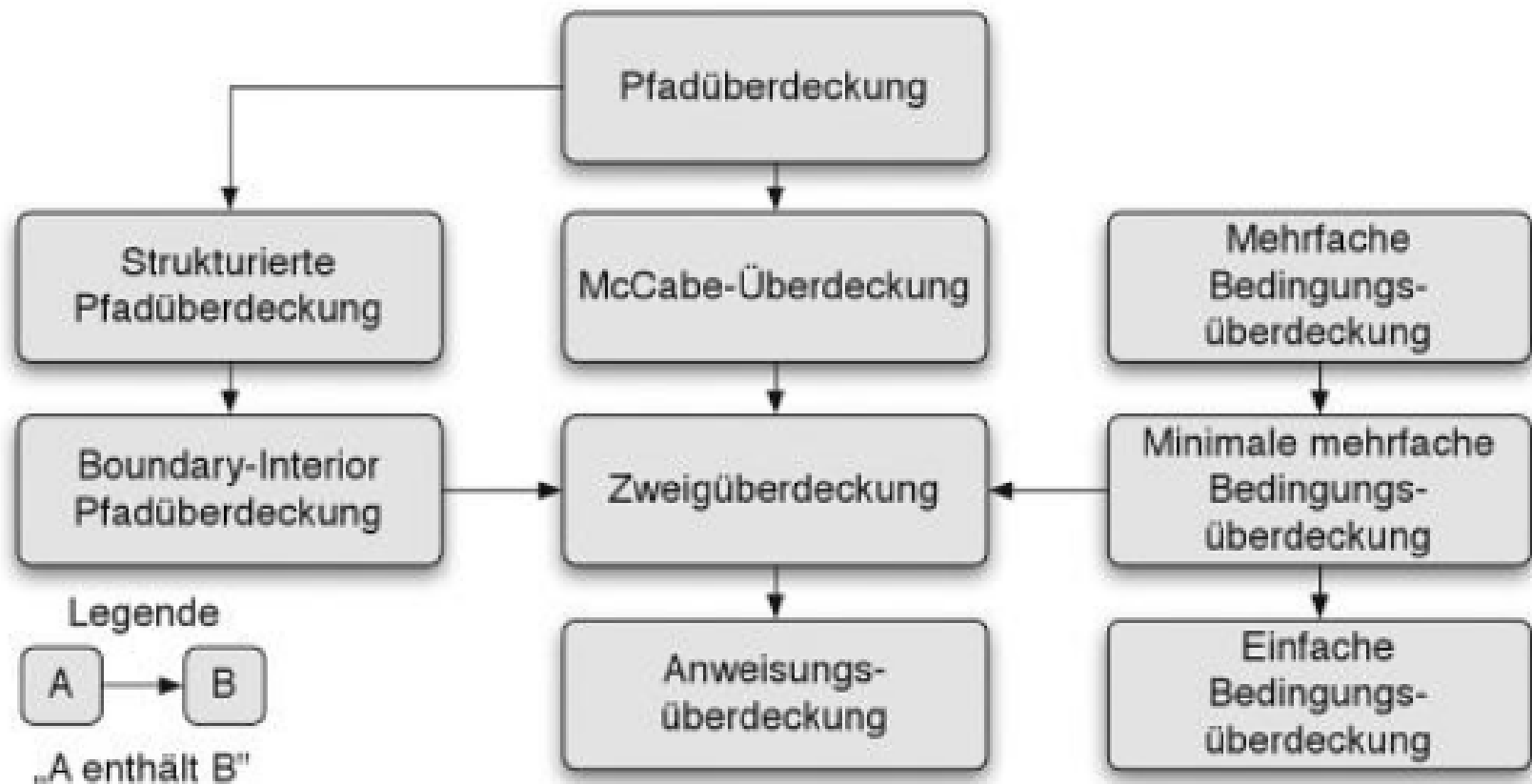
- **Kontrollflussorientierte Tests**

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

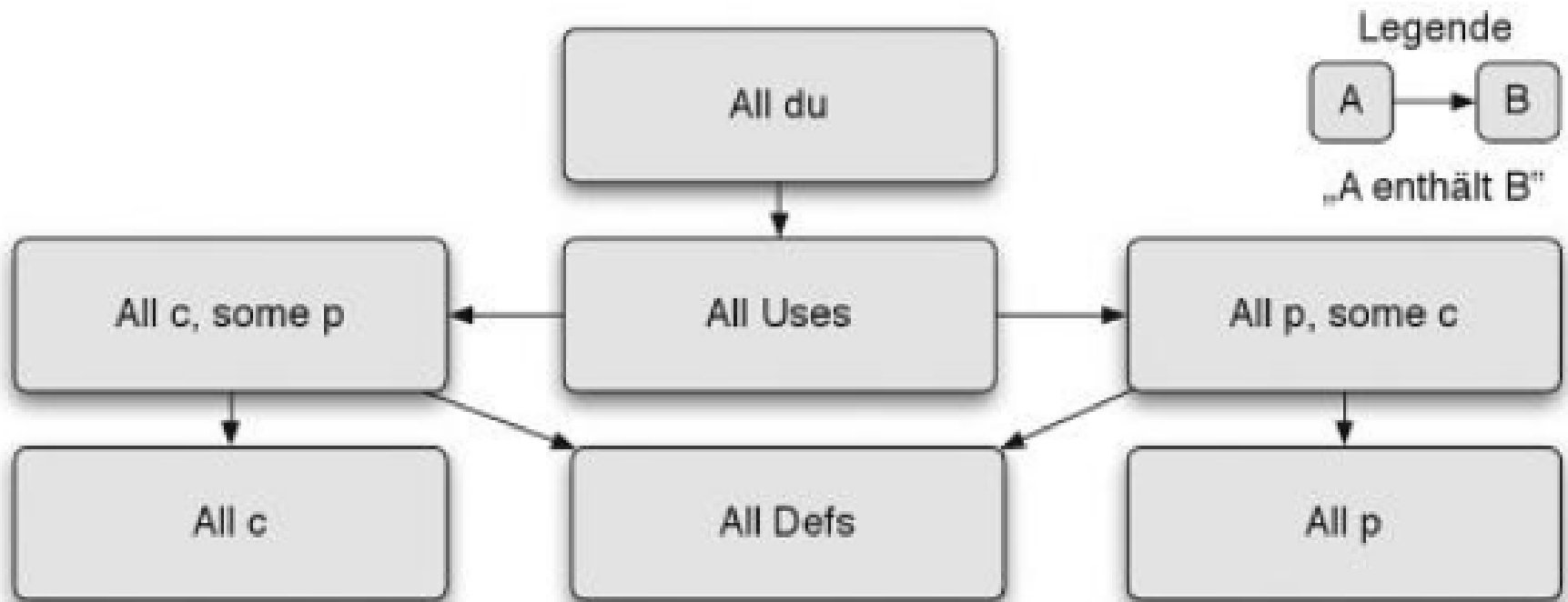
- **Datenflussorientierte Tests**

Heranziehen zusätzlicher Kriterien aus der Beschaffenheit der manipulierten Daten.

# Kontrollflussorientierte Strukturtests



# Datenflussorientierte Strukturtests



## 1. Strukturanalyse

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

## 2. Testkonstruktion

Ableitung der Testfälle entsprechend eines vorher definierten Überdeckungskriterium.

## 3. Testdurchführung

# Kontrollflussmodellierung - Bsp

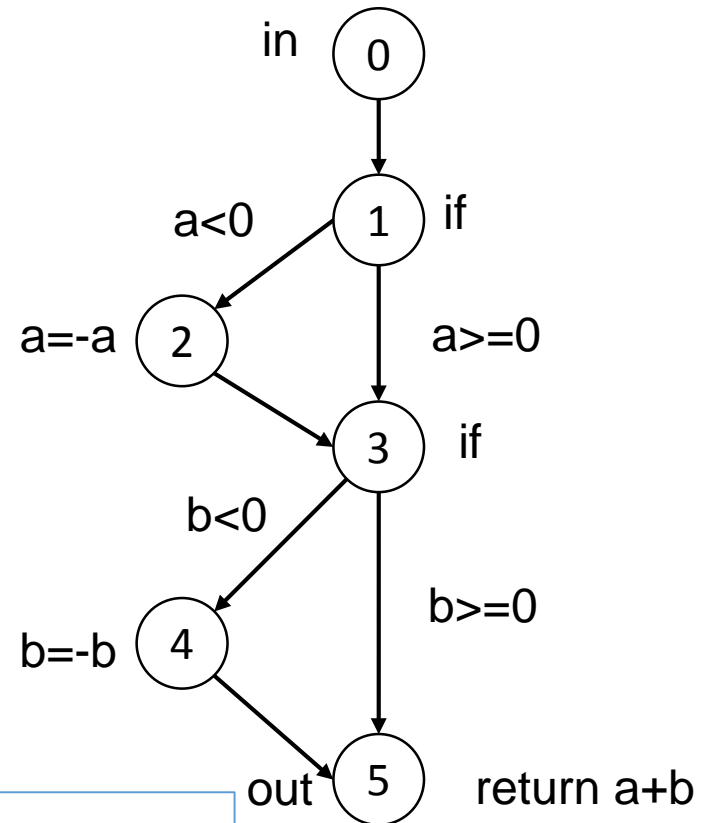
```
int manhattan(int a, int b) {
```

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

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

```
    return a+b;
```

```
}
```



**Immer:  
Ein einziger Einstiegspunkt,  
Ein einziger Ausstiegspunkt!**

- **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.

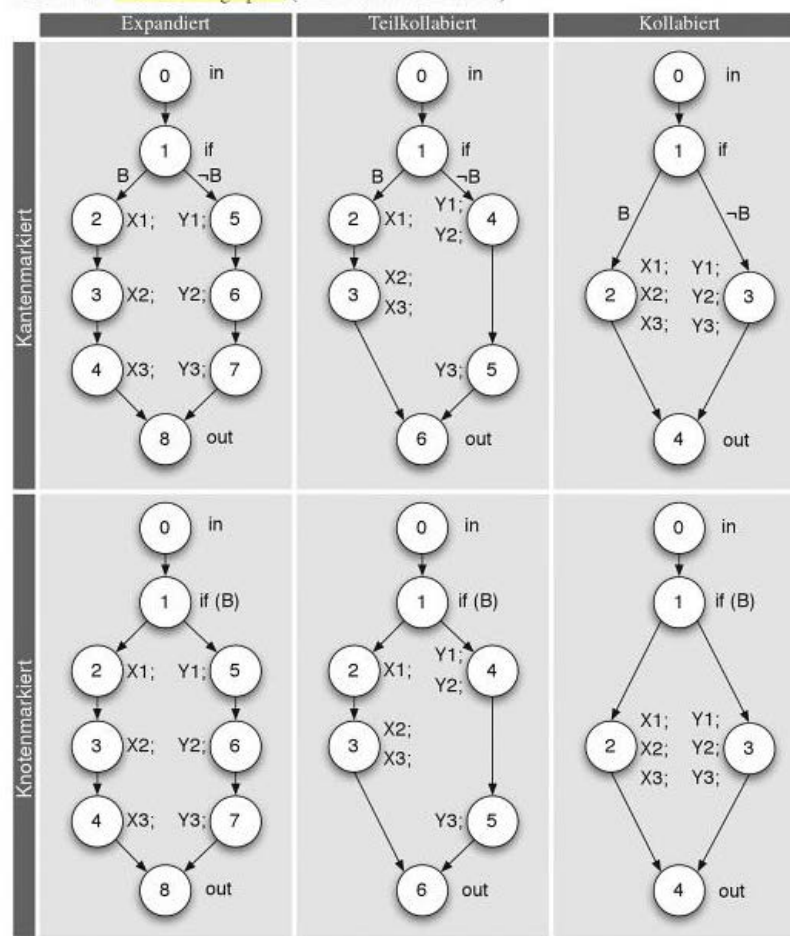
# Klassifikationsmerkmale - Bsp

expandiert

teilkollabiert

kollabiert

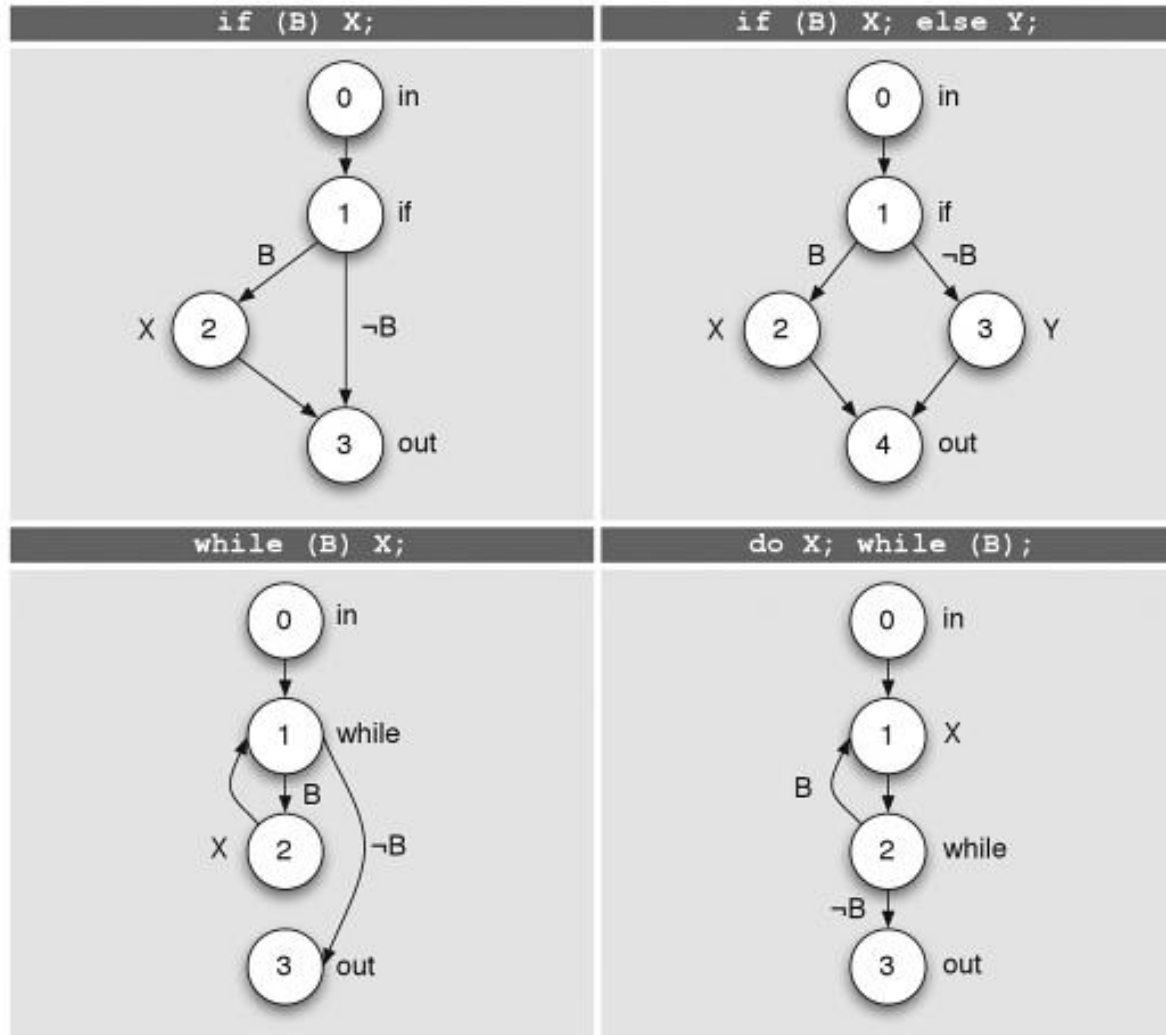
Kanten-  
markiert



Knoten-  
markiert



# Kontrollflussgraphen elementarer Konstrukte



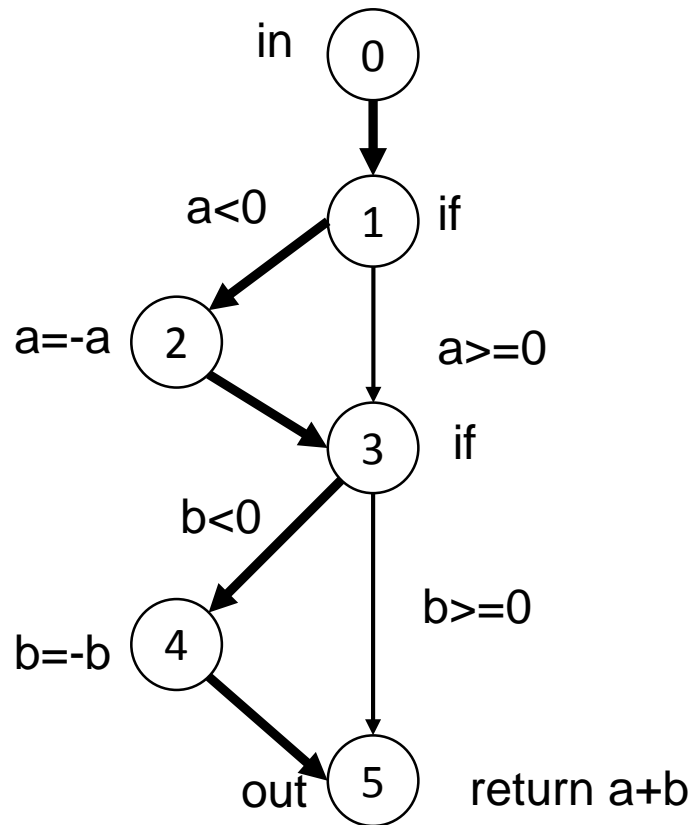
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

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

- 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 erzeugen 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.

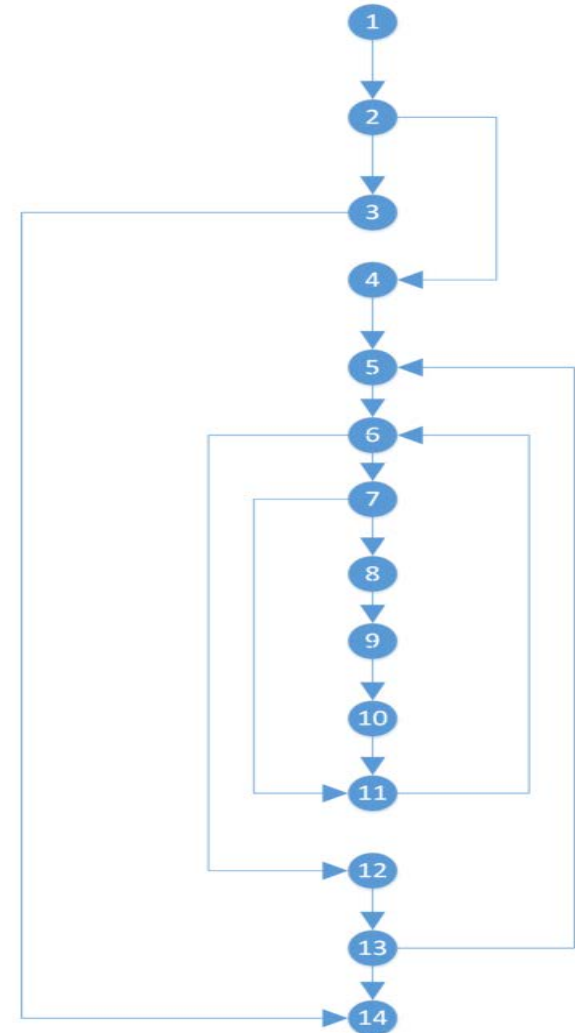


# Beispiel für Anweisungs Überdeckung

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 {  
3     if( array == null || array.length == 0 )  
4     {  
5         return;  
6     }  
7     int n = array.length;  
8     do  
9     {  
10        for( int i = 0; i < n - 1; i++ )  
11        {  
12            if( array[ i ] > array[ i + 1 ] )  
13            {  
14                int tmp = array[ i ];  
15                array[ i ] = array[ i + 1 ];  
16                array[ i + 1 ] = tmp;  
17            }  
18            n--;  
19        }  
20        while( n >= 0 );  
21    }  
22 }
```



Durch folgende Testfälle ist die Anweisungsüberdeckung zu 100% gegeben:

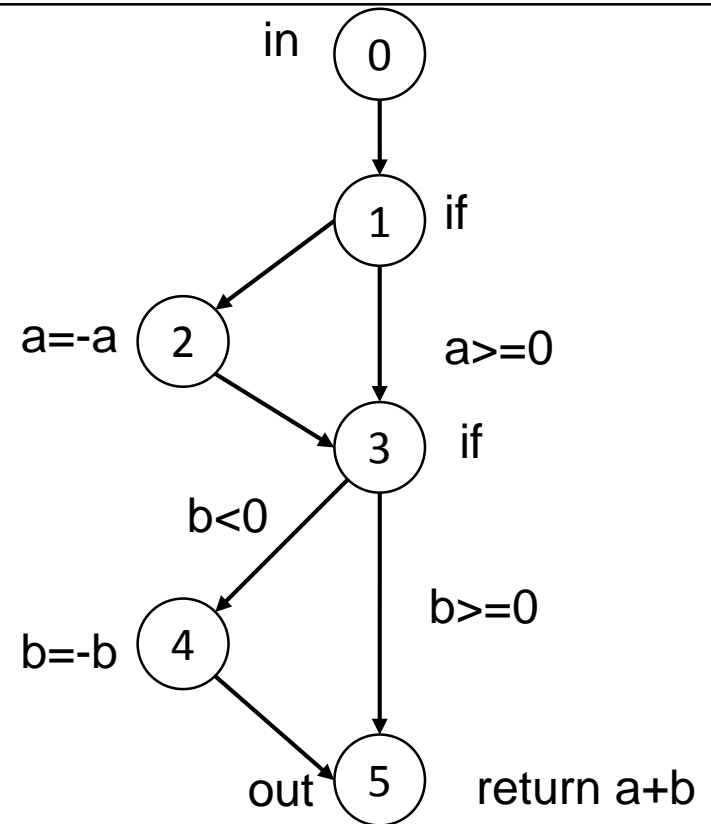
1. `bubbleSort(null)`
2. `bubbleSort(new int[]{2,1})`

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

- 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

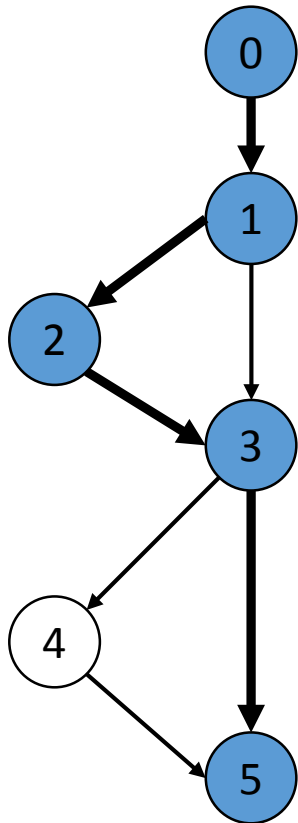
```
int manhattan(int a, int b) {  
  
    if(a<0) {  
        a=-a;  
    }  
    if(b<0) {  
        b=-b;  
    }  
  
    return a+b;  
}
```



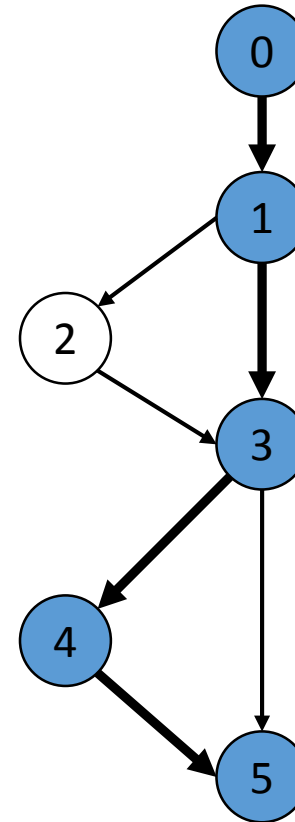
**Aufgabe: Definieren Sie Testfälle, so dass eine Zweigüberdeckung von 100 % erfüllt ist!**

# 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.

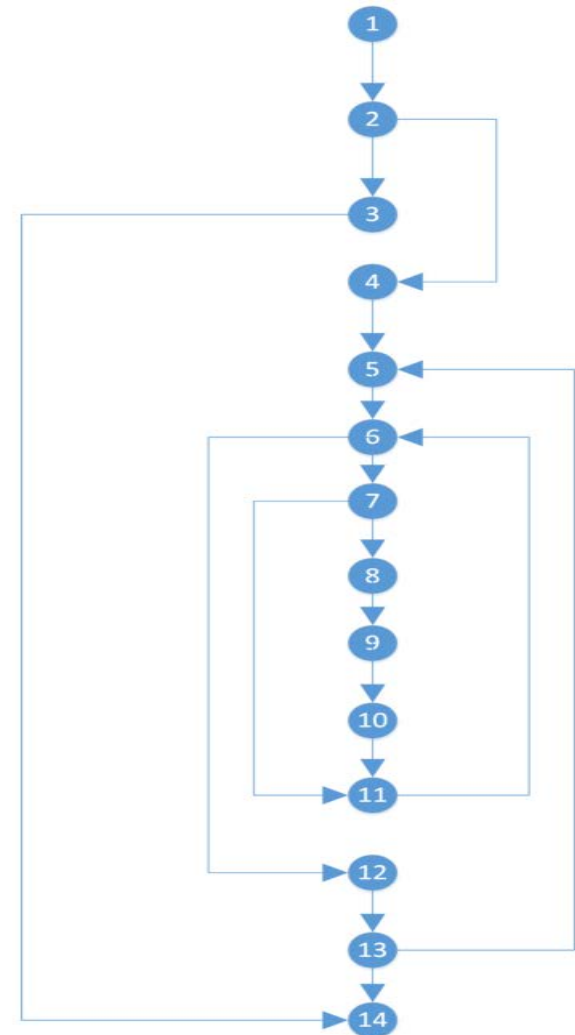


# Beispiel für Zweig Überdeckung

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 )  
4     {  
5         return;  
6     }  
7     int n = array.length;  
8     do  
9     {  
10        for( int i = 0; i < n - 1; i++ )  
11        {  
12            if( array[ i ] > array[ i + 1 ] )  
13            {  
14                int tmp = array[ i ];  
15                array[ i ] = array[ i + 1 ];  
16                array[ i + 1 ] = tmp;  
17            }  
18        }  
19        n--;  
20    }  
21    while( n >= 0 );  
22 }
```



# Übungsbeispiel - Testfälle

Durch folgende Testfälle ist die Zweigüberdeckung zu 100% gegeben:

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

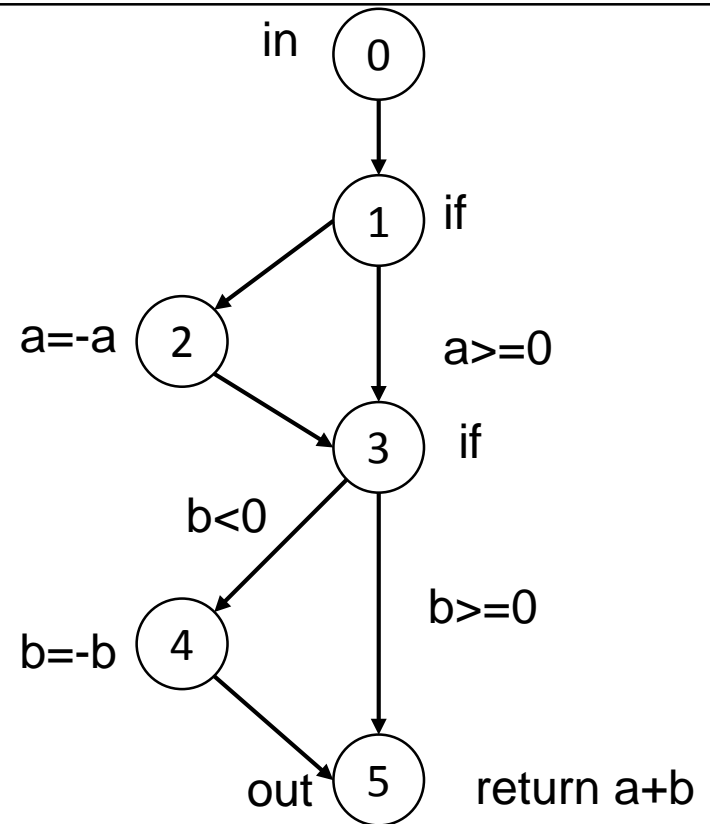
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- Bedingungsüberdeckung
- McCabe Überdeckung
- Defs Uses Überdeckung
- Required k-Tupel Überdeckung

# 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

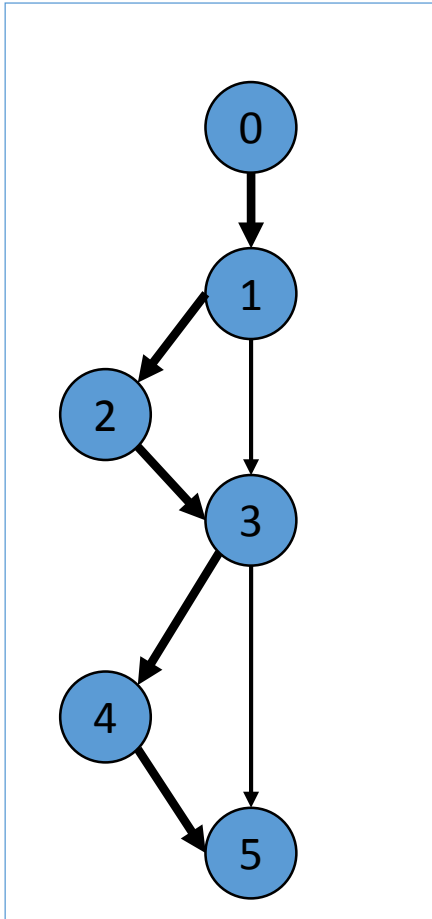
```
int manhattan(int a, int b) {  
  
    if (a < 0) {  
        a = -a;  
    }  
    if (b < 0) {  
        b = -b;  
    }  
  
    return a + b;  
}
```



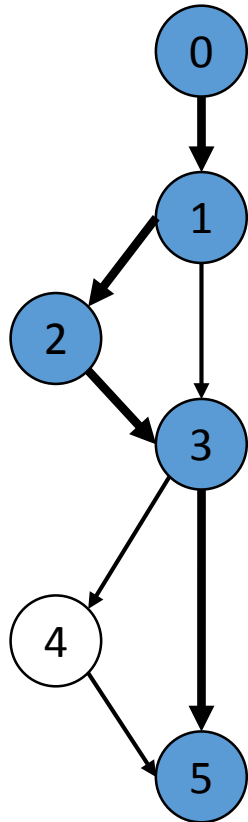
**Aufgabe: Definieren Sie Testfälle, so dass eine Pfadüberdeckung von 100 % erfüllt ist!**

# Pfadüberdeckung- Bsp manhattan

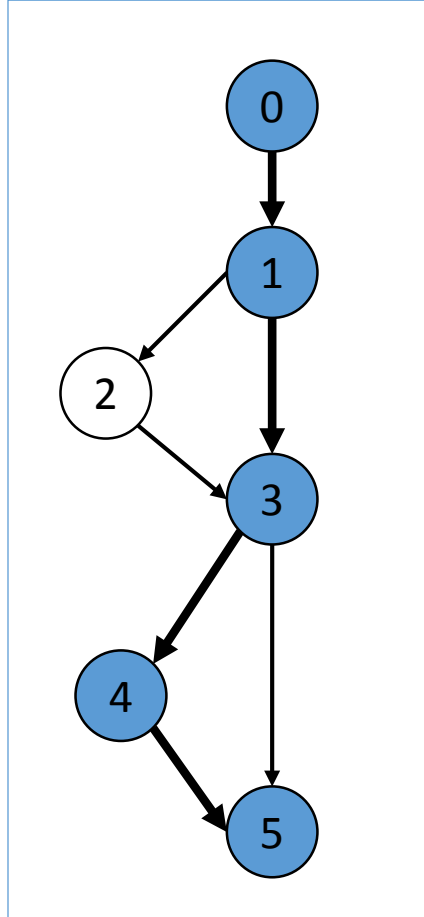
manhattan(-1,-1)



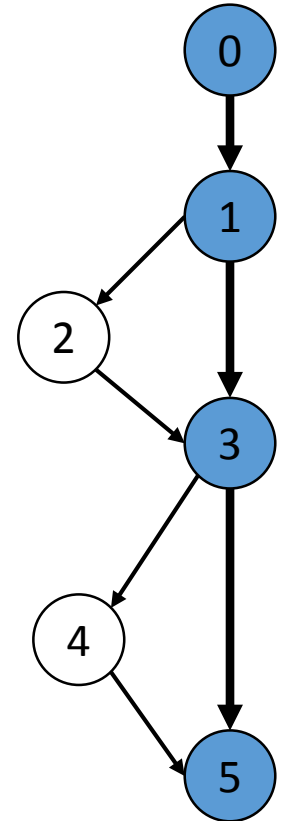
manhattan(-1,1)



manhattan(1,-1)

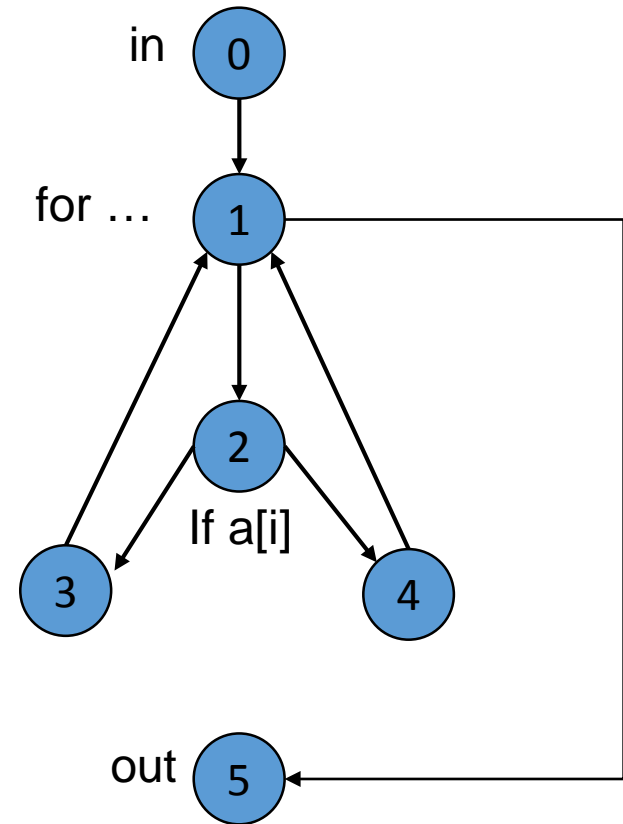


manhattan(1,1)



# Probleme der Pfadüberdeckung - Bsp

```
0 void foobar(int *a)
1 {
2   for(int i=0;i<512;i++) {
3     if(a[i]){
4       foo();
5     }
6     else{
7       bar();
8     }
9   }
10 }
```



→  $2^{512}$  verschiedene Pfade



# Varianten der Pfadüberdeckung

- Pfadüberdeckung in der Praxis normalerweise nicht realisierbar, nur theoretisch interessant.
- ➔ Varianten, um die Anzahl der Testfälle zu reduzieren.

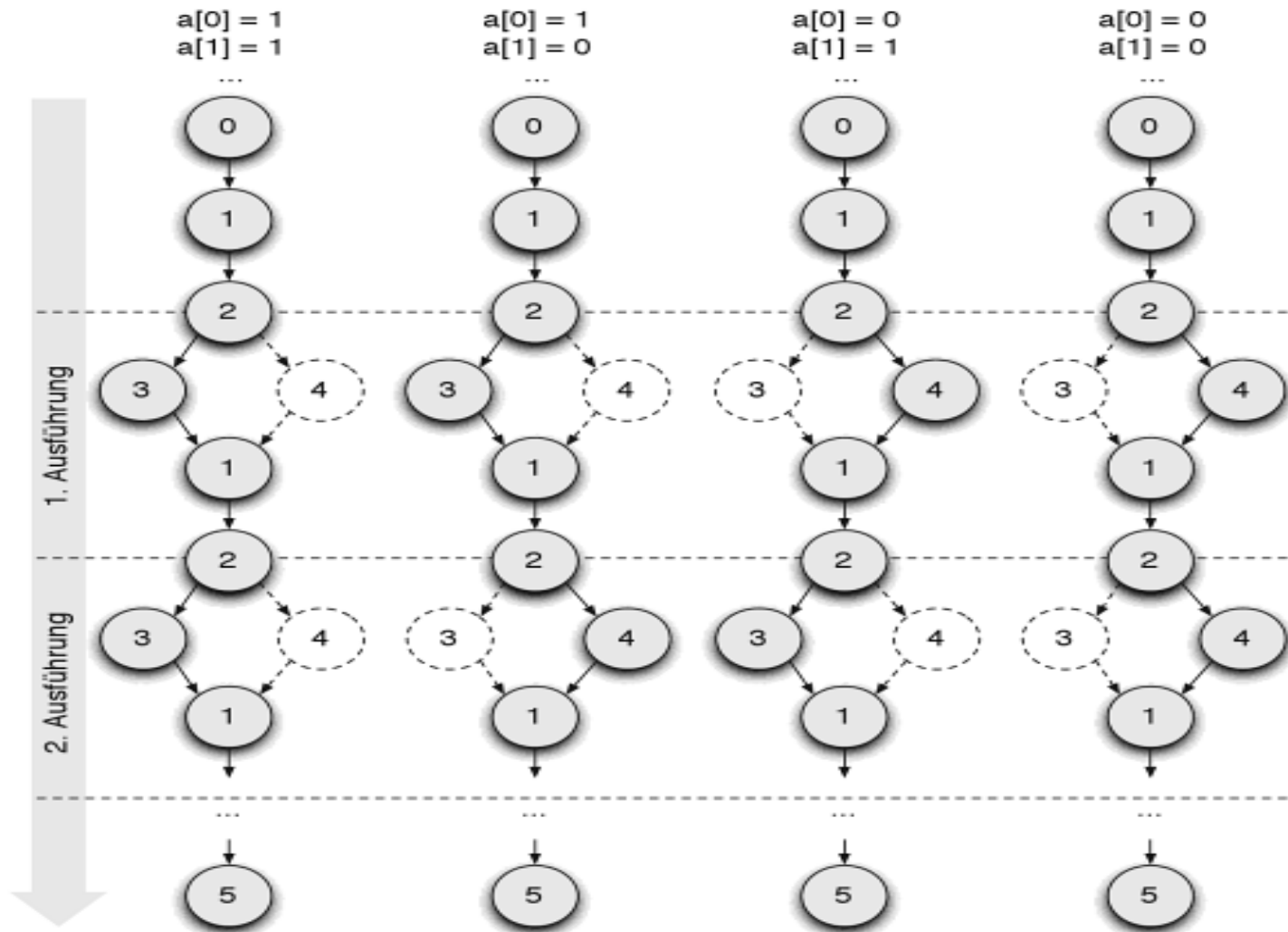
**Idee:** es ist nicht nötig, Schleifen öfter als ein paar Iterationen zu durchlaufen um fast alle Fehler zu finden.

## 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



## 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, insbds. bei verschachtelten Schleifen)

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

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

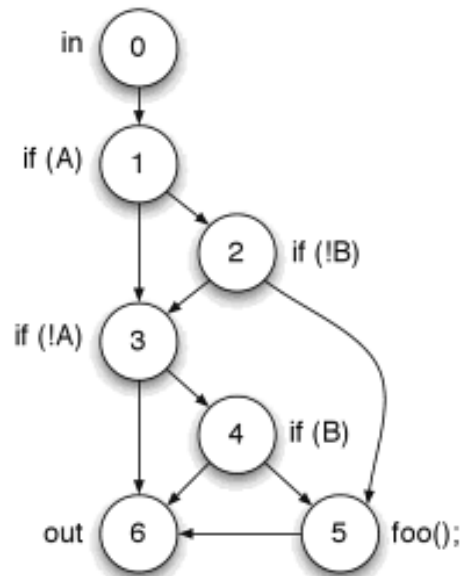
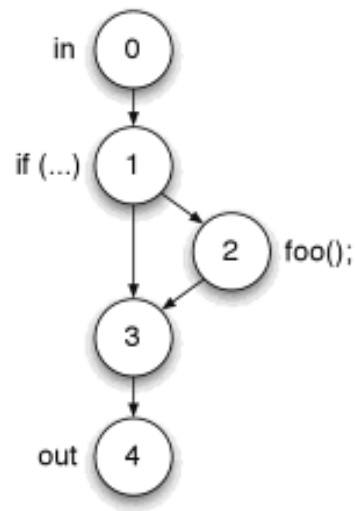
# Achtung bei der Zweigüberdeckung

Variante 1

```
if ((A && !B) || (!A && B))
{
    foo();
}
```

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

## **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



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

- **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*

Verfahren sind zu komplex um in der Praxis eine große Rolle zu spielen.

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

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**

Im industriellen Bereich im Wesentlichen relevant:

- **Anweisungsüberdeckung**

$$M_{c0} = (\text{Anzahl der überdeckten Knoten}) / (\text{Anzahl der Knoten}) * 100 [\%]$$

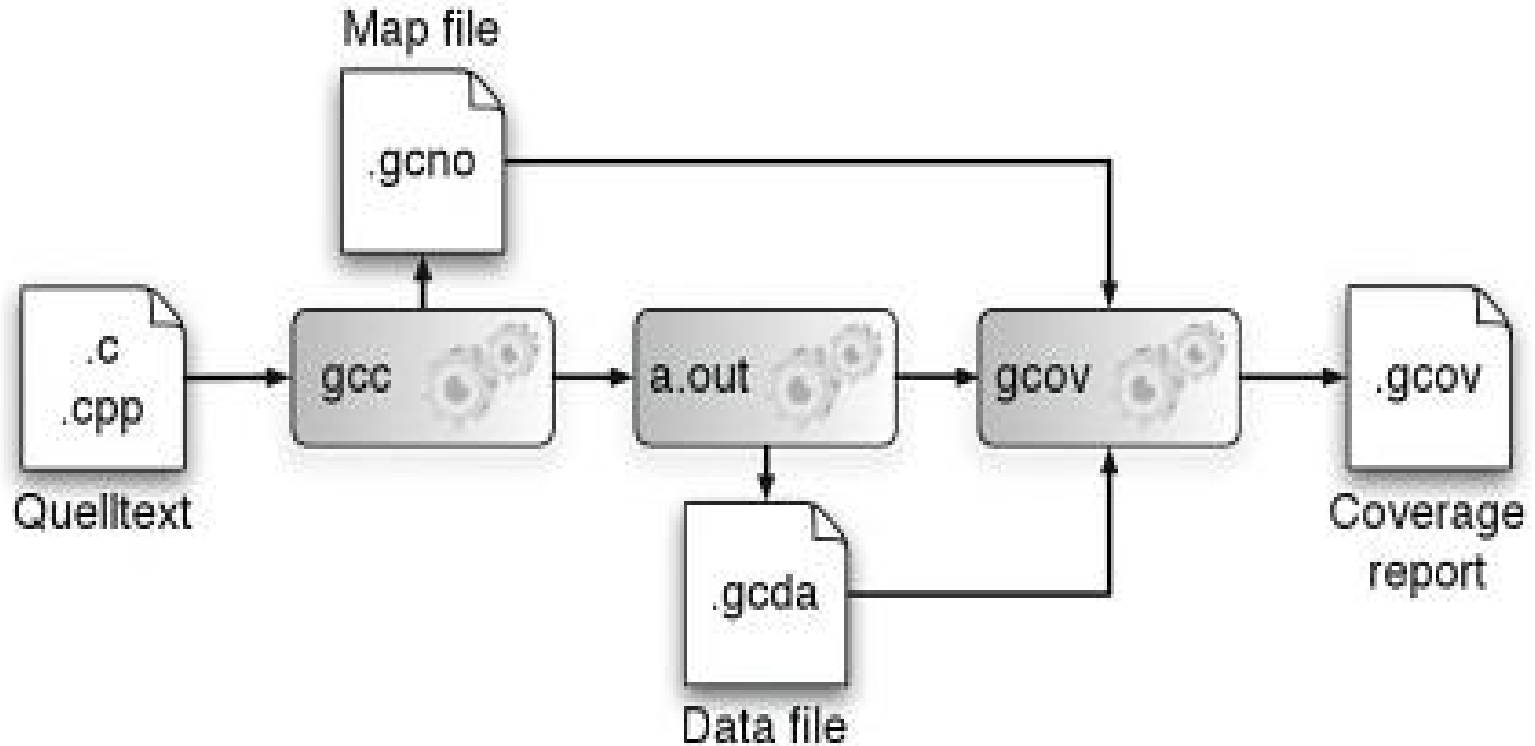
- **Zweigüberdeckung**

$$M_{c1} = (\text{Anzahl der überdeckten Kanten}) / (\text{Anzahl der Kanten}) * 100 [\%]$$

## 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



Quelle des Bilds: D. Hoffmann, Software Qualität, 2. Auflage

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

## 1. Kompilieren:

gcc -Wall -fprofile-arcs -ftest-coverage  
-omanhattan manhattan.c

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

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



## ■ 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
```

## ▪ Analysieren

gcov manhattan.c

➔ gcov manhattan.c  
File ,manhattan.c'  
Lines executed: 75.00% of 12  
Manhattan.c:creating ,manhattan.c.gcov'

- Analysieren:

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

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

➔ **Vorsicht bei Kompilieren mit Optimierung!**

# 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](http://en.wikipedia.org/wiki/Java_Code_Coverage_Tools)

- Gcovr (<http://gcovr.com>) : Python Package, das gcov erweitert um code coverage Informationen dazustellen:
  - Text output with coverage statistics indicated with summary statistics and lists of uncovered line, and
  - XML output that is compatible with the Cobertura code coverage utility.
- Lcov (<http://ltp.sourceforge.net/coverage/lcov.php>): Graphisches front-end für gcov.

Bsp eines einfachen Code Coverage reports (quelle:  
<http://ltp.sourceforge.net/coverage/lcov/output/index.html> )

## *LCOV - code coverage report*



Current view: **top level**

Test: **Basic example** ( [view descriptions](#) )

Date: **2015-10-08 10:24:18**

Legend: Rating: low: < 75 % medium: >= 75 % high: >= 90 %

	Hit	Total	Coverage
Lines:	20	22	90.9 %
Functions:	3	3	100.0 %
Branches:	8	10	80.0 %

Directory	Line Coverage ↕		Functions ↕		Branches ↕	
<a href="#">example</a>		90.0 %	9 / 10	100.0 %	1 / 1	75.0 %
<a href="#">example/methods</a>		91.7 %	11 / 12	100.0 %	2 / 2	83.3 %

Generated by: [LCOV version 1.12](#)

- Cobertura:  
<http://cobertura.github.io/cobertura/>  
(unterstützt noch nicht neueste Java Versionen)
  - JaCoCo: <http://eclemma.org/jacoco/>
- ➔ Kann z.B. in den maven build Prozess eingeklinkt werden.

# Prinzip von Cobertura in Maven

- Bytecode wird erweitert und unter target/generated-classes/cobertura abgelegt.
- Ausführung der Tests durch Surefire unter Verwendung dieser Klassen
- Beendigung der JVM zum Test führt zur Speicherung der Testergebnisse unter target/site/cobertura/cobertura/ser
- Maven Plugin für cobertura erzeugt daraus einen HTML Report.



Verfahren, um verschiedene Testverfahren quantitativ zu bewerten

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

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

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

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);
}
```

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

# Links und Literatur zu JUnit

- **Unfassende Darstellung:**  
M. Tamm: JUnit Profiwissen, 1. Auflage 3013, dpunkt.verlag
- **JUnit Website incl Tutorials:**  
<http://junit.org/junit4/>
- **JUnit Artikel**  
<http://www.vogella.com/tutorials/JUnit/article.html>
- **JUnit Tutorial**  
<http://www.javacodegeeks.com/2014/11/JUnit-tutorial-unit-testing.html>
- **Einführung in JUnit3:** Kent Beck: JUnit Pocket Guide, Kindle Edition, O'Reilly

***Achten Sie bei Tutorials und Büchern stets auf die richtige Version von JUnit!***

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

# Testautomatisierung

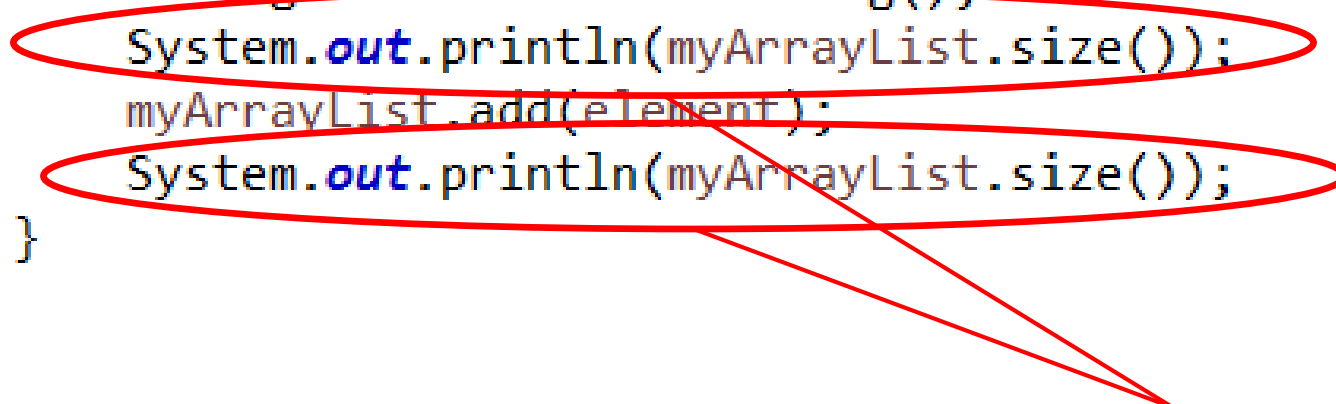
- Motivation / Idee
- Ziele von JUnit
- Features von JUnit
- Einfaches Beispiel
- TestSuites
- TestFixturees
- JUnit Annotations
- JUnit Assertions
- Namenskonventionen
- Parametrisierbare Tests
- JUnit Rules
- JUnit Categories
- JUnit Theories
- JUnit Custom Runners
- JUnit3 vs JUnit4



## Beispiel: Testen der java.util.ArrayList

### 1. Versuch:

```
public static void firstArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size());  
    myArrayList.add(element);  
    System.out.println(myArrayList.size());  
}
```



Prüfen und interpretieren!

## Beispiel: Testen der java.util.ArrayList

### 2. Versuch:

```
public static void secondArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size() == 0);  
    myArrayList.add(element);  
    System.out.println(myArrayList.size()==1);  
}
```

Prüfen!

## Beispiel: Testen der java.util.ArrayList

### 3. Versuch:



Man kriegt mit, wenn der Test fehlschlägt.  
➔ Keine Prüfung, sondern automatisierte Tests

```
public static void automatedArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    assertTrue(myArrayList.size()==0);  
    myArrayList.add(element);  
    assertTrue(myArrayList.size()==1);  
}
```

```
public static void assertTrue(boolean condition) {  
    if(!condition){  
        throw new RuntimeException("Assertion failed");  
    }  
}
```

# Warum überhaupt automatisieren

➔ Es gibt ganz viele Argumente.

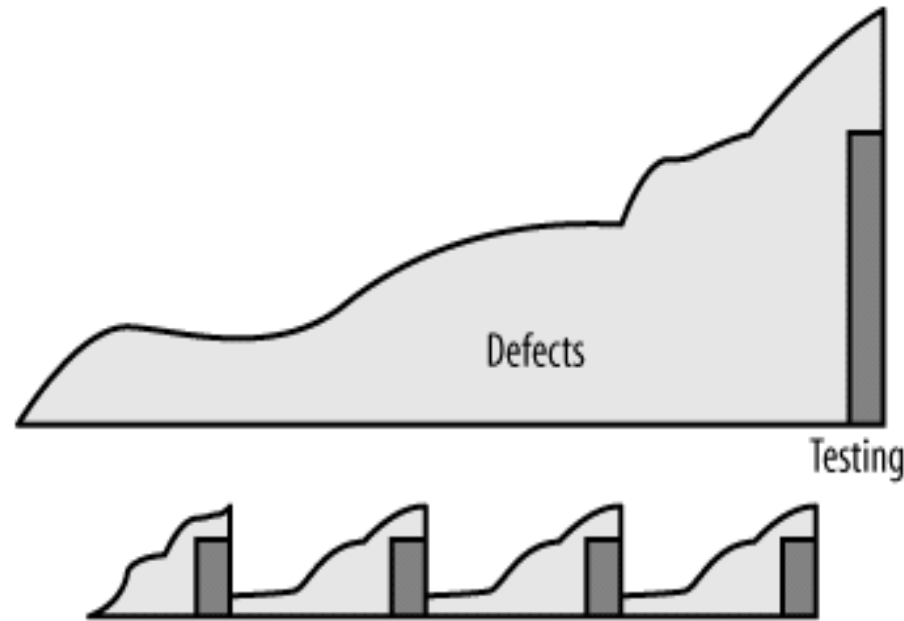
Hauptargument:

Vertrauen in die eigene Arbeit

# Zeitliche Vorteile der Testautomatisierung

- Kurzfristig für den Entwickler
  - Zeitersparnis bei Fehlerfinden und Korrigieren
- Langfristig für den Entwickler
  - Sicherheit, den Code langfristig warten zu können, ohne ihn zu brechen
- Für das Team und den Kunden
  - Einfache Integration von gutgetestetem Code

# Defect Entwicklung bei Testautomatisierung



*Figure 1-1. Frequent testing leaves fewer defects at the end*

Quelle: JUnit Pocket Guide, Kent Beck, Kindle Edition

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

JUnit bietet eine Infrastruktur, um viele Tests automatisiert laufen zu lassen und das Ergebnis wiederzugeben.

## JUnit

- Lässt Tests automatisiert laufen.
- Lässt viele Tests gemeinsam laufen und fasst das Ergebnis zusammen.
- Vergleicht Ergebnisse mit Erwartungen und teilt Unterschiede mit.



# Ziele von JUnit

- Tests sollen einfach zu schreiben sein.
- Es soll einfach sein, das Schreiben von Tests zu lernen.
- Schnelle Testausführung.
- Einfache Testausführung (per Knopfdruck, einfache Darstellung der Ergebnisse).
- Isolierte Ausführung, keine Beeinflussung von Tests untereinander.
- Tests sollen zusammensetzbar sein.

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

## Features von JUnit

- Infrastruktur für automatisierte Tests
  - Test schreiben
  - Test durchführen
  - Test auswerten
- Test vorbereiten
- Test nachbereiten
- Tests organisieren
- Parametrisierbare Tests
- ...

# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

# Einfaches Beispiel in Eclipse

Quelle: <https://www.javacodegeeks.com/2014/11/JUnit-tutorial-unit-testing.html>

Java Klasse:

```
package demopackage;

import java.util.Arrays;

public class FirstDayAtSchool {

    public String[] prepareMyBag() {
        String[] schoolbag = { "Books", "Notebooks", "Pens" };
        System.out.println("My school bag contains: "
            + Arrays.toString(schoolbag));
        return schoolbag;
    }

    public String[] addPencils() {
        String[] schoolbag = { "Books", "Notebooks", "Pens", "Pencils" };
        System.out.println("Now my school bag contains: "
            + Arrays.toString(schoolbag));
        return schoolbag;
    }

}
```

# Einfaches Beispiel in Eclipse

## JUnit Testklasse

```
package demopackage;

import static org.junit.Assert.*;

public class FirstDayAtSchoolTest {

    FirstDayAtSchool school = new FirstDayAtSchool();
    String[] bag1 = { "Books", "Notebooks", "Pens" };
    String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils" };

    @Test
    public void testPrepareMyBag() {
        System.out.println("Inside testPrepareMyBag()");
        assertEquals(bag1, school.prepareMyBag());
    }

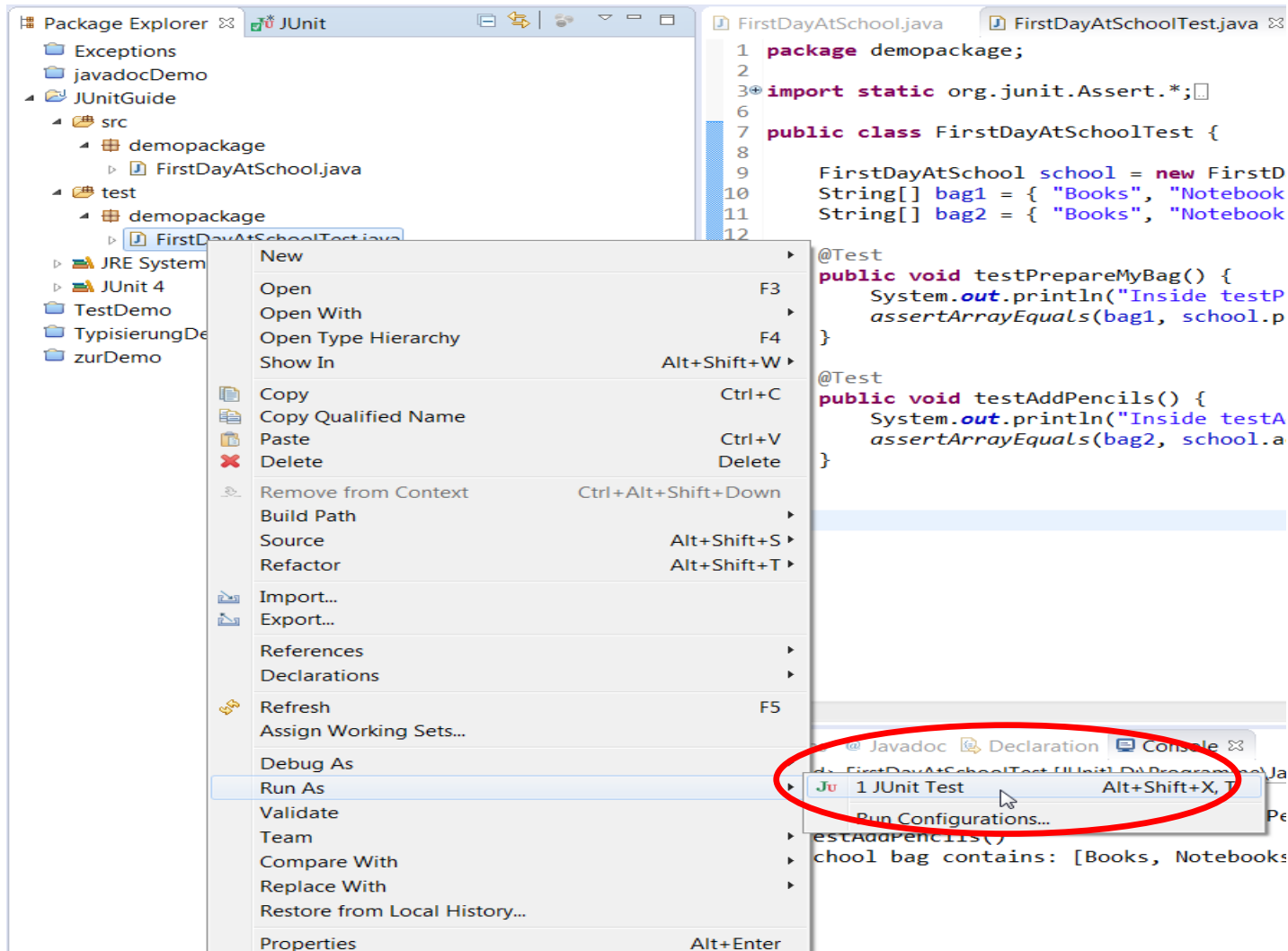
    @Test
    public void testAddPencils() {
        System.out.println("Inside testAddPencils()");
        assertEquals(bag2, school.addPencils());
    }
}
```

Beliebiger Name, keine  
spezielle Superklasse (JUnit 4)

Das sind Testmethoden

Assert Methode zur  
Überprüfung

# Test laufen lassen



The screenshot shows an IDE with the following components:

- Package Explorer (Left):** Shows the project structure. The 'test' package contains 'FirstDayAtSchoolTest.java', which is selected.
- Context Menu (Over 'FirstDayAtSchoolTest.java'):**
  - Options include: New, Open, Open With, Open Type Hierarchy, Show In, Copy, Copy Qualified Name, Paste, Delete, Remove from Context, Build Path, Source, Refactor, Import..., Export..., References, Declarations, Refresh, Assign Working Sets..., Debug As, **Run As** (highlighted), Validate, Team, Compare With, Replace With, Restore from Local History..., Properties.
  - A sub-menu for 'Run As' is open, showing:
    - JUnit (highlighted and circled in red)
    - 1 JUnit Test (selected)
    - Run Configurations...
- Main Editor (Right):** Shows the code for 'FirstDayAtSchoolTest.java':
 

```
1 package demopackage;
2
3 import static org.junit.Assert.*;
4
5 public class FirstDayAtSchoolTest {
6
7     FirstDayAtSchool school = new FirstD
8     String[] bag1 = { "Books", "Notebook
9     String[] bag2 = { "Books", "Notebook
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```
- Console (Bottom Right):** Shows the output of the test run:
 

```
@ Javadoc Declaration Console
FirstDayAtSchoolTest [JUnit4] (D:\Programme\Ja
testAddPencils()
school bag contains: [Books, Notebooks]
```

# Ergebnis

The screenshot shows the JUnit runner interface in an IDE. At the top, there are tabs for 'Package Explorer' and 'JUnit'. Below the tabs is a toolbar with various icons for test actions. The main area displays the test results: 'Finished after 0,016 seconds'. Below this, a summary bar shows 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'. A green progress bar is visible below the summary. At the bottom, a list of test cases is shown, with 'demopackage.FirstDayAtSchoolTest' selected, indicating it was run by JUnit 4 in 0,000 seconds.

Package Explorer JUnit

Finished after 0,016 seconds

Runs: 2/2 Errors: 0 Failures: 0

demopackage.FirstDayAtSchoolTest [Runner: JUnit 4] (0,000 s)



- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturetures](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

Es kann sinnvoll sein, nur einen Teil der Tests laufen zu lassen (z.B. alle Smoketests, alle Tests zu einer Komponente etc).

→ Unterstützung durch IDE Konfigurationen  
**Oder**

→ Erstellung von **JUnit TestSuites**

→ Standard Weg, unabhängig von einer IDE.

→ Test Suites können einfacher in einer Sourceverwaltung verwaltet werden.

# Test Suites

Ausführen von mehreren Tests aus verschiedenen Testklassen.

Bsp:

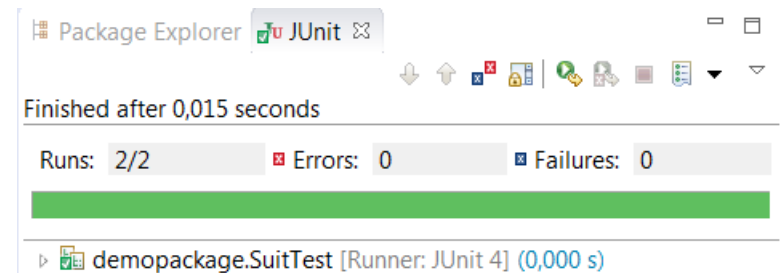
Anderer **Runner** → Gemeinsam laufen lassen!

```
package demopackage;
```

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)  
@Suite.SuiteClasses({ PrepareMyBagTest.class, AddPencilsTest.class })  
public class SuitTest {  
}
```

Was soll  
gemeinsam laufen?



# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixtures](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

## Bsp: Test einer Client-Server-Kommunikation

```
public void testPing(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
    client.send("ping");  
    assertEquals("ack", client.receive());  
    client.stop();  
    server.stop();  
}
```

# Fixtures - Motivation

Bsp: Test einer Client-Server-Kommunikation, sauber

```
public void testPingsauber() {  
    Server server = new Server();  
    server.start();  
    try {  
        Client client = new Client();  
        client.start();  
        try {  
            client.send("ping");  
            assertEquals("ack", client.receive());  
        } finally {  
            client.stop();  
        }  
    } finally {  
        server.stop();  
    }  
}
```

Test vorbereiten

Eigentlicher Test

aufräumen

# Fixtures

## Deklaration

```
Server server;  
Client client;
```

## Test setup

```
protected void bereiteTestVor(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
}
```

## Test Durchführung

```
public void testPing() {  
    client.send("ping");  
    assertEquals("ack", client.receive());  
}
```

## Aufräumen

```
protected void raeumeTestauf(){  
    try{  
        client.stop();  
    }finally{  
        server.stop();  
    }  
}
```

**Fixtures:** Vorbereitungscode und Nachbereitungscode für Testmethoden.

➔ Initialisierung

**Nutzen:**

- Isolation des eigentliche Testcodes
- Vermeidung redundanten Vor- und Nachbereitungscode



Realisierung von Testfixtures in JUnit 4 durch Annotationen:

***@Before***  
***public void method()***

This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).

***@After***  
***public void method()***

This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.

# Testautomatisierung

Realisierung von Testfixtures in JUnit 4 durch Annotationen:

## ***@BeforeClass***

***public static void method()***

This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.

## ***@AfterClass***

***public static void method()***

This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.

Wichtig: JUnit4 garantiert,

- alle @After Methoden werden immer aufgerufen werden, auch wenn eine davon eine Exception wirft.
- alle @After Methoden werden immer aufgerufen werden, auch wenn eine der @Before Methoden eine Exception wirft.

# Testautomatisierung

Beispiel:  
Klasse zu  
testen:  
Calculator.java

```
package demopackage;

public class Calculator {
    private static int result;

    public void add(int n) {
        result = result + n;
    }

    public void subtract(int n) {
        result = result - 1;           //Bug : result = result - n
    }

    public void multiply(int n) {}     //Not implemented yet

    public void divide(int n) {
        result = result / n;
    }

    public void square(int n) {
        result = n * n;
    }

    public void squareRoot(int n) {
        for (; ; ) ;                 //Bug : Endlosschleife
    }

    public void clear() {              // Ergebnis löschen
        result = 0;
    }

    public void switchOn() {           // Bildschirm einschalten, Piepsen, oder was
        result = 0;                   // Taschenrechner halt so tun
    }

    public void switchOff() { }        // Ausschalten

    public int getResult() {
        return result;
    }
}
```

## Testklasse CalculatorTest

```
package demopackage;

import static org.junit.Assert.*;

import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class CalculatorTest {

    private static Calculator calculator;

    @BeforeClass
    public static void switchOnCalculator() {
        System.out.println("\tSwitch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
    }

    @AfterClass
    public static void switchOffCalculator() {
        System.out.println("\tSwitch off calculator");
        calculator.switchOff();
        calculator = null;
    }

    @Before
    public void clearCalculator() {
        System.out.println("zu Beginn jedes Tests wird der Calculator zurückgesetzt");
        calculator.clear();
    }

    @Test
    public void test_add() {
        calculator.add(8);
        calculator.divide(2);
        assertEquals(calculator.getResult(), 4);
    }

    @Test(expected = ArithmeticException.class)
    public void test_divideByZero() {
        calculator.divide(0);
    }

    @Test
    public void test_multiply() {
        calculator.multiply(10);
        assertEquals(calculator.getResult(), 100);
    }
}
```

# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

# JUnit annotations

Annotation	Description
<b>@Test</b> <b>public void method()</b>	The @Test annotation identifies a method as a test method.
<b>@Test (expected =</b> <b>Exception.class)</b>	Fails if the method does not throw the named exception.
<b>@Test(timeout=100)</b>	Fails if the method takes longer than 100 milliseconds.
<b>@Before</b> <b>public void method()</b>	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
<b>@After</b> <b>public void method()</b>	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
<b>@BeforeClass</b> <b>public static void</b> <b>method()</b>	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
<b>@AfterClass</b> <b>public static void</b> <b>method()</b>	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
<b>@Ignore or</b> <b>@Ignore("Why</b> <b>disabled")</b>	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit3 vs JUnit4](#)



# Gängige Assert Methoden

Statement	Description
<b>fail(message)</b>	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
<b>assertTrue([message,] boolean condition)</b>	Checks that the boolean condition is true.
<b>assertFalse([message,] boolean condition)</b>	Checks that the boolean condition is false.
<b>assertEquals([message,] expected, actual)</b>	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
<b>assertEquals([message,] expected, actual, tolerance)</b>	Test that float or double values match. The tolerance is the number of decimals which must be the same.
<b>assertNull([message,] object)</b>	Checks that the object is null.
<b>assertNotNull([message,] object)</b>	Checks that the object is not null.
<b>assertSame([message,] expected, actual)</b>	Checks that both variables refer to the same object.
<b>assertNotSame([message,] expected, actual)</b>	Checks that both variables refer to different objects.

Komplett: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

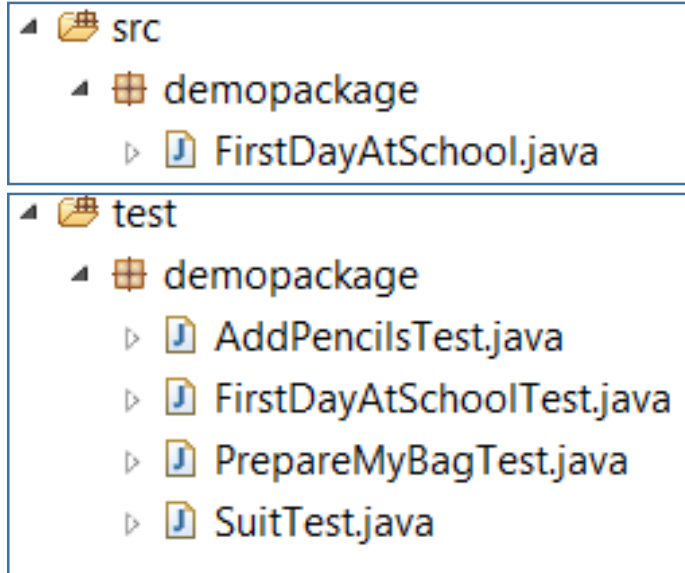
# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

## JUnit Konventionen

- Trennung Code under Test von Testcode

### JUnitGuide



Code under Test

Testcode

## JUnit Namenskonventionen

**Klasse**, die eine andere Klasse testet, hat den Namen der zu testenden Klasse + „Test“

### Bsp:

- zu testen: *Car.java*
- Testklasse: *CarTest.java*

## JUnit Namenskonventionen

### Test Methoden:

Konvention (nicht zwingend seit JUnit 4): Beginne den Namen mit „*test*“

Benennung (Konvention nach *M. Tamm: JUnit Profiwissen*):

- *test()*
- *test\_<Name der getesteten Methode>()*
- *test\_that\_<erwartetes Verhalten>()*
- *test\_that\_<erwartetes Verhalten>\_when\_<Vorbedingung>()*

# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

Parametrisierter Test → Instanzen für das Kreuzprodukt aus Test Daten Elementen und Testmethoden.

**Bsp:** Klasse zu testen: Fibonacci.java

```
package demo;  
  
public class Fibonacci {  
    public static int compute(int n) {  
        int result = 0;  
  
        if (n <= 1) {  
            result = n;  
        } else {  
            result = compute(n - 1) + compute(n - 2);  
        }  
  
        return result;  
    }  
}
```

# Testen verschiedener Kombinationen

Bisher:

```
package demo;

import static org.junit.Assert.*;

public class AlterFibonacciTest {

    @Test
    public void testCompute() {
        assertEquals(0, Fibonacci.compute(0));
        assertEquals(1, Fibonacci.compute(1));
        assertEquals(1, Fibonacci.compute(2));
        assertEquals(2, Fibonacci.compute(3));
        assertEquals(3, Fibonacci.compute(4));
        assertEquals(5, Fibonacci.compute(5));
        assertEquals(8, Fibonacci.compute(6));
    }
}
```



# Testen verschiedener Kombinationen - besser

## Testklasse: FibonacciTest

Jede Instanz des Tests  
wird mit dem  
Konstruktor und den  
Werten aus der  
`@Parameters` Methode  
konstruiert.

```
import static org.junit.Assert.assertEquals;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class FibonacciTest {

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
        });
    }

    private int fInput;
    private int fExpected;

    public FibonacciTest(int input, int expected) {
        fInput = input;
        fExpected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

## Parametrisierte Tests howto:

- Testklasse mit `@RunWith(Parameterized.class)` annotieren.
- Implementieren einer public static Methode, annotiert mit `@Parameters`, die eine Collection von Objekt Arrays als Test Datensätze zurückliefert.
- Public Konstruktor, der als Argument die Elemente eines Object Arrays aus der statischen `@Parameters` Methode annimmt.
- Instanzvariablen für Elemente der Testdaten.
- Testfälle, die diese Instanzvariablen als Quelle der Testdaten verwenden.

Möglichkeit, auf den Konstruktor zu verzichten:  
Annotation der Membervariablen mit  
`@Parameter(...)`

```
@Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, { 2, 1 },
        { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 } });
}

@Parameter(0)
public int fInput;

@Parameter(1)
public int fExpected;
```

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)


- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

## Motivation:

- @Before und @After wird verwendet, um Tests vorzubereiten und nachher aufzuräumen.
- Bsp: temporäres Verzeichnis anlegen, nachher löschen.
  - ➔ das braucht man öfter als nur in einer Testklasse

➔ Was tun?

## Lösungsmöglichkeiten:

1. Redundante `@Before` und `@After` Methoden in verschiedenen Klassen 
2. Basisklasse für alle Testklassen mit gemeinsam zu nutzenden `@Before` und `@After` Methoden 
3. Hierarchie von Testklassen mit Kombinationen von `@Before` und `@After` Methoden 

oder

JUnit Rules



# JUnit Rules

- JUnit Rules erlauben es, querschnittliche Vorbereitungs- und Aufräumarbeiten nur einmal zu implementieren und zu pflegen.
- Es gibt bereits vorgefertigte JUnit Rules, die mit JUnit ausgeliefert werden.

# JUnit Rules Beispiel

```
public class AssetManagerTest {  
  
    @Rule  
    public TemporaryFolder tempFolder = new TemporaryFolder();  
  
    @Test  
    public void countsAssets() throws IOException {  
        AssetManager am = new AssetManager();  
        File assets = tempFolder.newFolder("assets");  
        am.createAssets(assets, 3);  
        assertEquals(3, am.countAssets());  
    }  
}
```

*Zu TemporaryFolder siehe*

*<http://junit.org/junit4/javadoc/latest/org/junit/rules/TemporaryFolder.html>*



Um eine Testklasse mit einem als MethodRule Klasse programmierten Testaspekt auszustatten:

- Öffentliche Membervariable zur Testklasse.
- Diese mit `@Rule` annotieren
- Der Variablen eine neue Instanz der gewünschten MethodRule Klasse zuweisen.

## Diese Rules gibt es bereits:

- TemporaryFolder Rule
- ExternalResource Rules
- ErrorCollector Rule
- Verifier Rule
- TestWatchman/TestWatcher Rules
- TestName Rule
- Timeout Rule
- ExpectedException Rules

*Teilweise als  
Superklassen für selbst  
zu implementierende  
Klassen*

*Siehe <https://github.com/junit-team/junit4/wiki/Rules>*

# Entsprechungen

<b>@Before</b>	<b>@Rule</b>
<b>@BeforeClass</b>	<b>@ClassRule</b>

# RuleChain

RuleChain erlaubt es, Rules hintereinander auszuführen.

```
public static class UseRuleChain {  
    @Rule  
    public TestRule chain = RuleChain  
        .outerRule(new LoggingRule("outer rule"))  
        .around(new LoggingRule("middle rule"))  
        .around(new LoggingRule("inner rule"));  
  
    @Test  
    public void example() {  
        assertTrue(true);  
    }  
}
```



```
starting outer rule  
starting middle rule  
starting inner rule  
finished inner rule  
finished middle rule  
finished outer rule
```

## Eigene Rules schreiben:

- Klasse anlegen, die das Interface TestRule implementiert:

<http://junit.org/junit4/javadoc/latest/org/junit/rules/TestRule.html>

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```

# Custom Rules

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```

Repräsentiert den Aufruf  
der eigentlichen @Test  
Methode (incl aller before  
und after Methoden)

Weitere Beschreibung des Tests

```
public abstract class Statement {  
    /**  
     * Run the action, throwing a {@code Throwable} if anything goes wrong.  
     */  
    public abstract void evaluate() throws Throwable;  
}
```

Hier wird der eigentliche Test ausgeführt

# Custom Rules – Bsp Verifier

```
public abstract class Verifier implements TestRule {
    public Statement apply(final Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                base.evaluate();
                verify();
            }
        };
    }

    /**
     * Override this to add verification logic. Overrides should throw an
     * exception to indicate that verification failed.
     */
    protected void verify() throws Throwable {
    }
}
```

**Oft verwendetes Muster:** Erzeugen und Rückgabe eines anonymen inneren Statements, das in der evaluate Methode denjenigen Code enthält, der den Test ausmacht und an passender Stelle base.evaluate aufruft.

# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)



# JUnit Categories

JUnit Categories dienen dazu, Testmethoden zu gruppieren und nur Tests einer bestimmten Kategorie laufen zu lassen. Dies geschieht zusätzlich zu den TestSuites.

siehe

- ➔ M. Tamm: JUnit Profiwissen, 1. Auflage 2013, dpunkt.verlag
- ➔ <https://github.com/junit-team/junit4/wiki/Categories>
- ➔ <https://community.oracle.com/blogs/johnsmart/2010/04/25/grouping-tests-using-junit-categories-0>
- ➔ <https://examples.javacodegeeks.com/core-java/junit/junit-categories-example/>

# JUnit Categories - Bsp

## 1. Interface als Marker definieren

```
package demopackage;  
  
public interface MySpecialTests {  
  
}
```

## 2. Testmethoden annotieren

```
@Test  
@Category(MySpecialTests.class)  
public void test_add() {  
    calculator.add(1);  
    calculator.add(1);  
    assertEquals(calculator.getResult(), 2);  
}
```

## 3. Laufen lassen

```
@RunWith(Categories.class)  
@SuiteClasses({CalculatorTest.class})  
@IncludeCategory(MySpecialTests.class)  
public class KategorieSuite {  
  
}
```

# Testautomatisierung

- [Motivation / Idee](#)
  - [Ziele von JUnit](#)
  - [Features von JUnit](#)
  - [Einfaches Beispiel](#)
  - [TestSuites](#)
  - [TestFixturees](#)
  - [JUnit Annotations](#)
  - [JUnit Assertions](#)
  - [Namenskonventionen](#)
  - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
  - [JUnit Categories](#)
  - [JUnit Theories](#)
  - [JUnit Custom Runners](#)
  - [JUnit3 vs JUnit4](#)

## Idee:

Allgemeingültige Aussage als Test formulieren  
und durch eine Reihe von Testwerten  
überprüfen.

Paper dazu:

<http://web.archive.org/web/20110608210825/http://shareandenjoy.saff.net/tdd-specifications.pdf>

Titel:

The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests

## Beispiel: Test einer Multiplikation von BigInteger

### Variante 1:

```
@Test
public void testMultiplyBigInteger() {
    assertEquals(new BigInteger("20"), new BigInteger("10").multiply(new BigInteger("2")));
    assertEquals(new BigInteger("30"), new BigInteger("10").multiply(new BigInteger("3")));
    assertEquals(new BigInteger("50"), new BigInteger("25").multiply(new BigInteger("2")));
    assertEquals(new BigInteger("2"), new BigInteger("1").multiply(new BigInteger("2")));
}
```

Test von Berechnung mit bestimmten Eingabewerten gegen Erwartungswerte der Berechnung für diese Eingabewerte.

## Beispiel: Test einer Multiplikation von BigInteger

### Variante 2:

```
@Test
public void testGeneral(){
    testMultiplyInGeneral(10,2);
    testMultiplyInGeneral(10,3);
    testMultiplyInGeneral(25,2);
    testMultiplyInGeneral(1,2);
}

public void testMultiplyInGeneral(int firstInt, int secondInt){
    System.out.println(firstInt + " und " + secondInt);
    assertEquals(new BigInteger(String.valueOf(firstInt*secondInt)),
        new BigInteger(String.valueOf(firstInt)).multiply(new BigInteger(String.valueOf(secondInt))));
}
```

Formulierung der Testmethode als allgemeine „Theorie“, Test mit im Prinzip unendlich vielen Werten möglich. Keine explizite Vorgabe des erwarteten Ergebnisses als Wert!

➔ Diese Idee wird durch die Theories unterstützt.

## Beispiel: Test einer Multiplikation von BigInteger

### Variante 3:

```
@Theory
public void multiply_behaves_like_primitive(int firstInt, int secondInt){
    System.out.println("Test multiply_behaves_like_primitive() mit");
    System.out.println(firstInt + " und " + secondInt);
    assertEquals(new BigInteger(String.valueOf(firstInt*secondInt)),
        new BigInteger(String.valueOf(firstInt)).multiply(new BigInteger(String.valueOf(secondInt))));
}
```

```
@DataPoints
public static int[] INT_DATA = {1,2,3,4,5,6,7,8,9};
```

Umsetzung des Prinzips mithilfe von Theories.

# Bsp zu JUnit Theories

```
@RunWith(Theories.class)
public class BigIntegerTest {

    @Theory
    public void multiply_follows_commutative(BigInteger i1, BigInteger i2) {
        assertEquals(i1.multiply(i2), i2.multiply(i1));
    }

    @Theory
    public void divide_is_inverse_of_multiply(BigInteger i1, BigInteger i2) {
        assumeThat(i2.toString(), is(not("0")));
        assertEquals(i1.multiply(i2).divide(i2), i1);
    }

    @DataPoints
    public static BigInteger[] TEST_DATA = new BigInteger[]{
        new BigInteger("-1"),
        new BigInteger("0"),
        new BigInteger("42"),
        new BigInteger("12121234343434231213"),
    };
}
```



## Was ist an dem Beispiel besonders?

- Die Testmethoden haben mindestens einen Eingabeparameter.
- Es werden allgemeingültige Aussagen als Testmethoden formuliert.
- Der eigentliche Test läuft dann mit einem Set an Parametern durch, die man als `@DataPoints` annotiert.

- Die Testmethode und die Testdaten sind voneinander getrennt.
- Eine mit `@DataPoints` oder mit `@DataPoint` annotierte Variable oder statische Methode liefert ein Array oder einen Wert als Eingabewerte für einen bestimmten Datentyp.
- Der Theories Runner sammelt alle Beispielwerte ein unmittelbar bevor er die Theories Methode ausführt. Also nach allen `@BeforeClass` und `@Before` Methoden.

# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

## Bereits angesprochene JUnitRunner:

- *BlockJUnit4ClassRunner*: Default Runner
- *Suite*: Standard Runner um TEstSuites laufen zu lassen
- *Parameterized*: Runner für parametrisierte Tests
- *Categories*: Standard Runner für subsets von Tests, die entsprechend getagged sind.
- *Theories*: Runner für Testtheorien

# Eigene Test Runner

Test Runner sind für die Ausführung der Tests zuständig.

Um sie zu verwenden, ist die Annotation `@RunWith()` nötig.

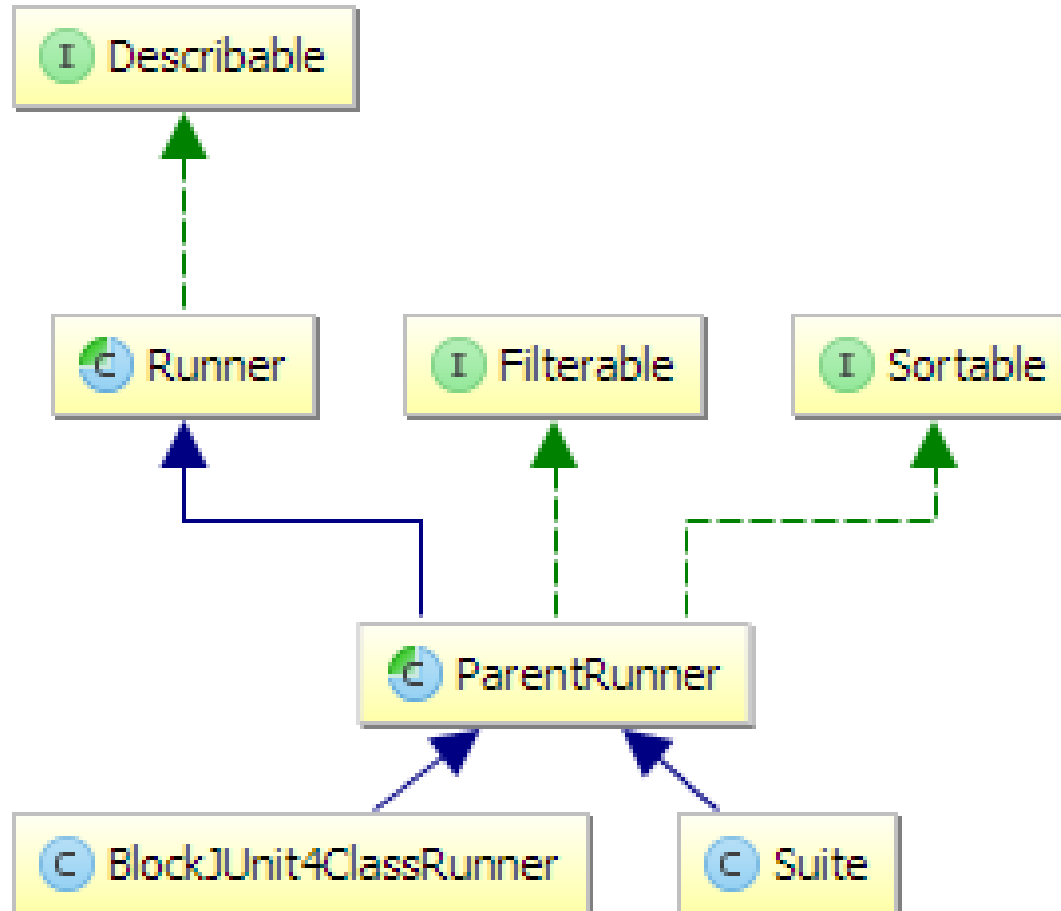
Testrunners Verantwortung:

- Testklassen Instanziierung
- Test Ausführung
- Reporting der Test Resultate

# Test Runner

- Ein Test Case kann den Runner selbst angeben:  
`@RunWith` Annotation
- Normalerweise genügen die bereits vorhandenen Runner. Wenn man selbst einen schreiben will, siehe
  - Michael Tamm: JUnit Profiwissen, dpunkt.verlag
  - <http://www.mscharhag.com/java/understanding-junits-runner-architecture>

# Runners Hierarchie



# Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)

- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)



# JUnit 3 vs JUnit 4

- Aktuell: JUnit4
- In Arbeit: JUnit5
- In vielen Projekten noch verwendet: JUnit3

```
package demopackage;

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {

    Calculator calculator;

    protected void setUp() throws Exception {
        System.out.println("\tSwitch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
        System.out
            .println("zu beginn jeden Tests wird der Calculator zuruecgesetzt");
        calculator.clear();
    }

    protected void tearDown() throws Exception {
        System.out.println("\tSwitch off calculator");
        calculator.switchOff();
        calculator = null;
    }

    public void testAdd() {
        calculator.add(1);
        calculator.add(1);
        assertEquals(calculator.getResult(), 2);
    }

}
```

# JUnit3 Testcase – Unterschiede zu JUnit4

```
package demopackage;
```

```
import junit.framework.TestCase;
```

```
public class CalculatorTest extends TestCase {
```

```
    Calculator calculator;
```

```
    protected void setUp() throws Exception {  
        System.out.println("\tSwitch on calculator");  
        calculator = new Calculator();  
        calculator.switchOn();  
        System.out.  
            .println("zu Beginn jeden Tests wird der Calculator zuruecgesetzt");  
        calculator.clear();  
    }
```

```
    protected void tearDown() throws Exception {  
        System.out.println("\tSwitch off calculator");  
        calculator.switchOff();  
        calculator = null;  
    }
```

```
    public void testAdd() {  
        calculator.add(1);  
        calculator.add(1);  
        assertEquals(calculator.getResult(), 2);  
    }
```

```
}
```

Kein `import static org.junit.Assert.*;`

Ableitung von `TestCase`

Methode `setUp()` und `tearDown()`  
Statt Annotierte Methoden

Name muss mit „test“ beginnen

# TestSuites in JUnit3

```
package meinpackage;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests extends TestSuite
{
    public static Test suite()
    {
        TestSuite mySuite = new TestSuite( "Meine Test-Suite" );
        mySuite.addTestSuite( meinpackage.MeineKlasseTest.class );
        // ... weitere Testklassen hinzufügen
        return mySuite;
    }
}
```

- Siehe Junit Profiwissen Ab Seite 91