

# QUALITÄTSSICHERUNG

## Wahlprojekt Vorlesung

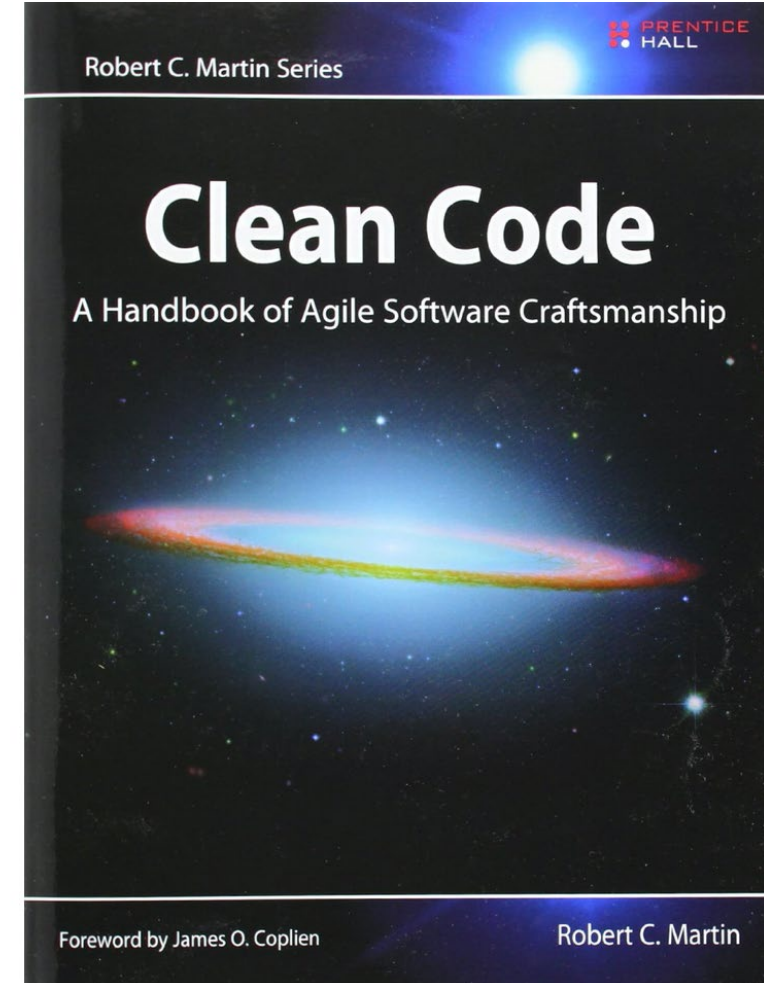
Dr. Eva-Maria Iwer  
Januar/ Februar 2021



# CLEAN CODE

First, you are a programmer. Second, you want to be a better programmer.

## Clean Code

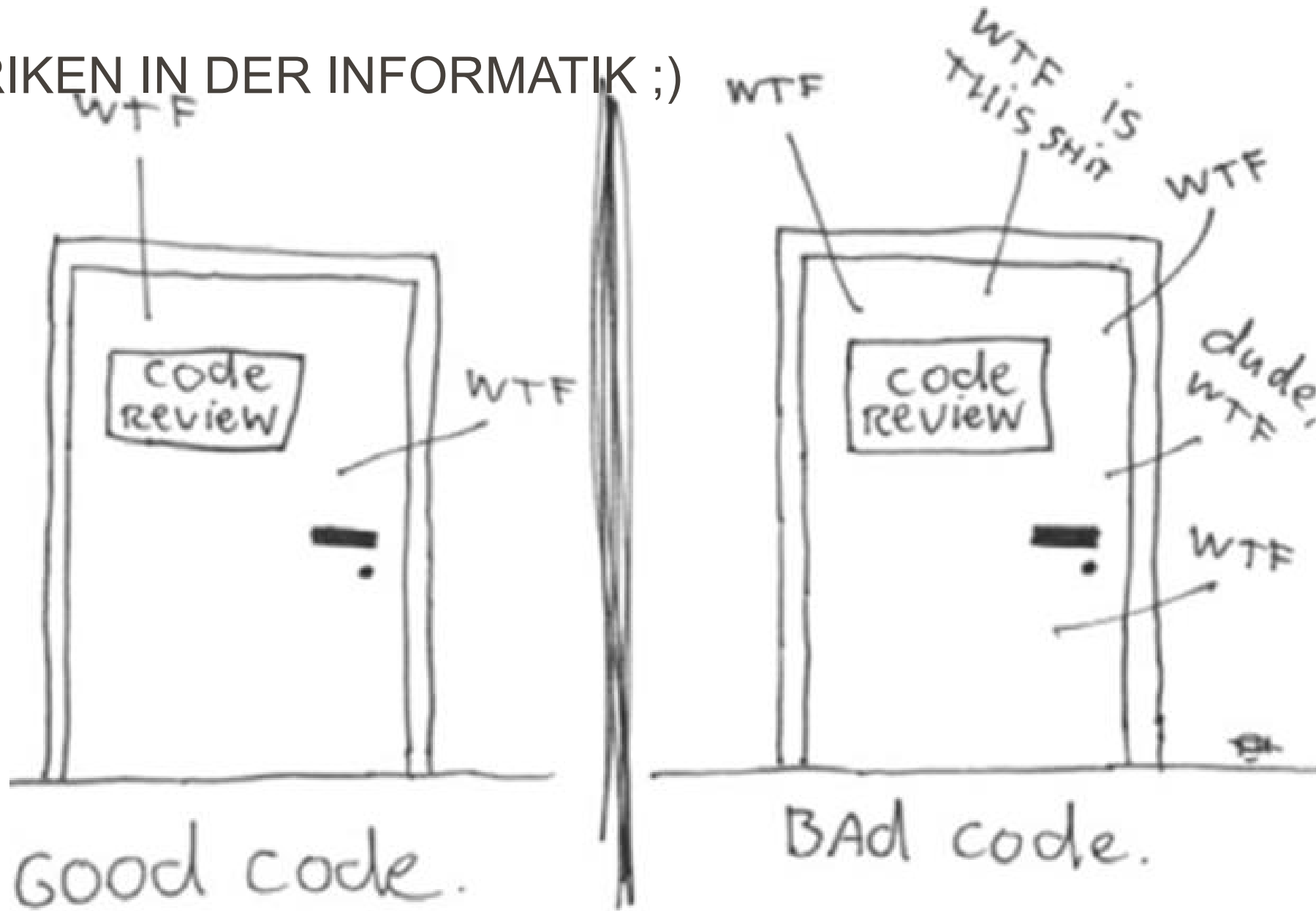


*„I LIKE MY CODE TO BE ELEGANT AND EFFICIENT. THE LOGIC SHOULD BE STRAIGHTFORWARD TO MAKE IT HARD FOR BUGS TO HIDE, THE DEPENDENCIES MINIMAL TO EASE MAINTENANCE, ERROR HANDLING COMPLETE ACCORDING TO AN ARTICULATED STRATEGY, AND PERFORMANCE CLOSE TO OPTIMAL SO AS NOT TO TEMPT PEOPLE TO MAKE THE CODE MESSY WITH UNPRINCIPLED OPTIMIZAION.“*

Bjarne Stroustrup – C++ Inventor

# METRIKEN IN DER INFORMATIK ;)

e RheinMain



(c) 2008 Focus Shift

Reproduced with the kind permission of Thom Holwerda.  
[http://www.osnews.com/story/19266/WTFs\\_m](http://www.osnews.com/story/19266/WTFs_m)

# CLEAN CODE

# WAS IST CLEAN CODE

„Clean“ ist in erster Linie Quellcode ab auch Dokumente, Konzepte, Regeln und Verfahren, die intuitiv verständlich sind.

Intuitiv verständlich ist alles, was mit wenig Aufwand in kurzer Zeit richtig verstanden werden kann.

# CLEAN CODE

## Prinzipien

- DRY – Don't repeat yourself
- KISS – Keep It simple, stupid
- Beware of Optimizations
- FCol – Favour Composition over Inheritance
- SRP – Single Responsibility Principle
- SoC – Separation of Concerns
- Coding Conventions
- ISP – Interface Segregation Principle
- DIP – Dependency Inversion Principle
- LSP – Liskov Substitutions Principle
- Information Hiding Principle
- Principle of Least Astonishment
- Tell, don't ask
- Law of Demeter
- YAGNI – You ain't gonna need it
- ...



# CLEAN CODE

## Praktiken

- Lesen
- Versionieren
- Scout Rule
- Root Cause Analysis
- Coding Conventions
- Documentation
- ...

# PRAKTIKEN



Code lesen



Bücher lesen



Artikel, How-Tos,  
Blogs, Foren, ... lesen

# PRAKTIKEN

## Versionierung

- Keine Angst vor Veränderungen

# PRAKTIKEN

## Scout Rule

- Verlasse den Code immer sauberer als du ihn betreten hast
- Verfall beginnt mit Kleinigkeiten, die lange genug unbeachtet liegen bleiben



# PRAKTIKEN

## Root Cause Analysis

- Nicht die Symptome bekämpfen, sondern die Ursachen



- Sind Regeln und Richtlinien für eine bestimmte Programmiersprache, die jeden Aspekt der Arbeit mit der Sprache behandelt
- Beispiele:
  - Benennung von Klassen, Methoden, Feldern, etc.
  - Datei- bzw. Projektstruktur
  - Struktur einzelner Dateien (Deklarationen, Funktionen, etc.)
  - Einrückungen, Klammerungen, White Spaces ...
  - Art und Form von Kommentaren
  - Generische Sprach-Verwendung („Best practices“)

- Keine Abkürzungen z.B. timeStamp statt tmp
- Suchfreundliche Namen z.B. days statt d
- Keine humoristischen Namen z.B. DeleteItems() statt DestroyEverything()
- Ein Bezeichnung pro Konzept z.B. Set[Name], Get[Name], Create[Name]...
- Keine Artikel z.B. BuddyList statt theBuddyList
- Klassennamen: Substantiv im Singular z.B. Customer
- Methodennamen: Verb (+ Substantiv) z.B. CreateCustomer, Create
- Listen, Array, Dictionaries: Plural z.B. List<Customer> customers
- Kein Implementierungsmerkmale im Namen z.B. nicht: CustomerSingleton
- Keine technische Bestandteile z.B. nicht: ClassNameImpl, ClassNameDelegate
- Keine Weasel Words verwenden – Weasel Words sind Wörter, die mehrere Bedeutungen haben bzw. deren Bedeutung stark vom Kontext abhängt.



# WEASEL WORDS

- AbstractDirectoryMonitor
- ErrorCorrectingYakizakanaMigrator
- SimpleGraphImporter
- RunnableCommandQueue
- ReadableStatementMediator
- ExecutableNodeUtil
- MultipleDeviceAdapter
- MultipleMutexInitializer
- StatefulConnectionInterpreter
- AutomaticSocketBuffer
- ThreadsafeRasterMonitor
- AbstractKeystrokeMarshaller
- ConfigurablePixelCollector
- CryptographicThreadGenerator
- ServiceManager
- CustomerRepository

# PRINZIPIEN

# PRINZIPIEN

## DRY

- Dupliziere keinen Code
- Duplizierter Code bedeutet:
  - Schlechte Wartbarkeit
  - Änderung in jedem Duplikat notwendig
  - Fehler werden meistens auch kopiert

# PRINZIPIEN

## KISS

- Löse Probleme so einfach wie möglich
- Nicht die neuesten Sprachfeatures oder Technologien verwenden, nur weil man das mal ausprobieren will.
- Lieber viel sehr einfache Methoden als eine sehr komplizierte verwenden.

# PRINZIPIEN

## Beware of Optimizations

- Optimierte nicht.
- Für Experten: Optimierte noch nicht.
- Oft erreicht man mit Optimierung sein Ziel nicht.
- Optimierter Code ist ...
  - komplizierter.
  - anfälliger für Fehler.
  - schlechter lesbar.

# PRINZIPIEN

## YAGNI

- Schreibe nur Code, der auch wirklich gebraucht wird.
- Schreibe Code genau zu dem Zeitpunkt, an dem er gebraucht wird.
- Abgeleitet davon:
  - Lösche nicht verwendeten Code
  - Nicht auskommentieren, löschen!
- Mehr Code bedeutet mehr Aufwand bei Änderung der Architektur.
- Mehr Code wird nicht bezahlt.

# PRINZIPIEN

Tell, don't ask

- Sage einem Objekt, was es zu tun hat, anstatt den Zustand des Objekts abzufragen und dann zu entscheiden.
- Kleinere Schnittstellen
- Losere Kopplung

# PRINZIPIEN

## SRP

- Klassen, Funktionen, Module usw. sollen nur für eine Aufgabe zuständig sein.
- Klare Verteilung der Aufgaben
- Kleine Klassen und Module



- Teile das Programm so auf, dass einzelne Teile möglichst wenige Abhängigkeiten zur anderen Teilen haben.
- Auf allen Ebenen anwendbar (z.B. Projekt, Klassen, Methoden).
- Projekt
  - Mehrere Projekte mit eigenen Aufgaben.
- Klassen
  - Abhängigkeiten über Interfaces auflösen.
  - Nur eine Aufgabe pro Klasse.

# PRINZIPIEN

## ISP

- Schnittstellen sollen nur benötigte Funktionen veröffentlichen
- Möglichst kleine Interfaces
- So wenig Abhängigkeiten wie möglich

# PRINZIPIEN

## DIP

- Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen

- Ein Subtyp darf die Funktionalität eines Basistyps lediglich erweitern, aber nicht einschränken.
- Subtypen müssen sich also wie ihr Basistypen verhalten

# PRINZIPIEN

## Information Hiding Principle

- Veröffentliche nur notwendige Methoden und Felder.
- Verbergen erleichtert die Nutzung der Schnittstelle.
- Reduziert Abhängigkeiten.
- Erleichtert die Wartbarkeit der Klasse.

# PRINZIPIEN

## Principle of Least Astonishment

- Programmteile sollten sich verhalten wie erwartet.
- Keine Nebeneffekte.
- Aus dem Namen einer Funktion sollte die Funktionalität klar erkennbar sein.

- Lange Abhängigkeitsketten zwischen den Klassen vermeiden. z.B. `x = a.B().C().D();`
- Erschwert Wartung, Erweiterung und Tests
- Ok sind:
  - ✓ Methoden und Felder der eigenen Klasse
  - ✓ Methoden und Felder von Parametern
  - ✓ Methoden und Felder von assoziierten Klassen
  - ✓ Methoden und Felder von lokal erzeugten Objekten
  - ✓ Methoden und Felder globaler Objekte

# METRIKEN



# CYCLOMATIC COMPLEXITY (CC)

- 
- CC bestimmt, wie komplex eine Methode oder Klasse ist.
- Methode:
  - Anzahl der Ausführungspfade (if, while, switch, ...)
- Klassen
  - Durchschnittliche Komplexität aller Methoden oder Summe der Komplexität aller Methoden (je nach Tool).
- Schwellenwert
  - Methoden mit  $CC < 15$  sind kompliziert, bis  $CC < 30$  aber ok

$$\text{LCOM3}(c) = \frac{m(c) - \left( \frac{\sum m_f(c)}{f(c)} \right)}{m(c) \cdot f(c)}$$

$m(c)$  := Anzahl an Methoden in  $c$

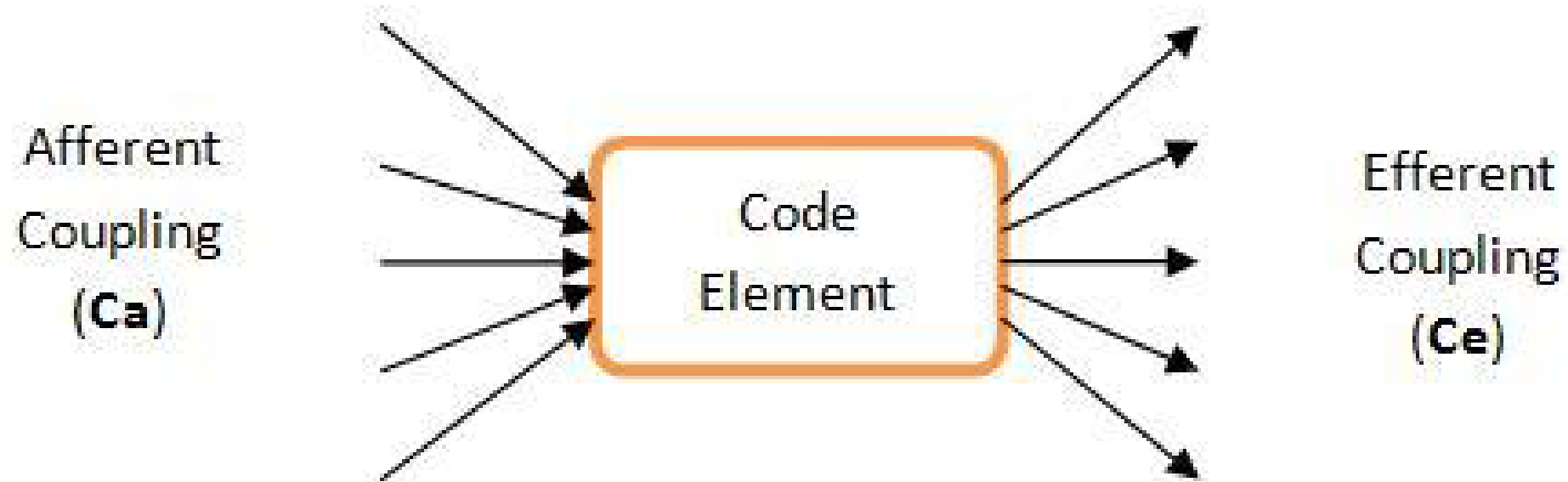
$f(c)$  := Anzahl an Feldern in  $c$

$m_f(c)$  := Anzahl an Methoden in  $c$  die auf das Feld  $f$  in  $c$  zugreifen

- Schwellenwert
  - LCOM3 = 0 bedeutet maximale Kohäsion
  - LCOM3 = 1 bedeutet keine Kohäsion
  - LCOM3 > 1 bedeutet Dead Code oder Variablen, die nur von außen benutzt werden.
  - LCOM3 < 1 ok

# COUPLING

- Beschreibt wie viele Typen von einer Klasse (oder Modul, Namespace, Assembly) abhängen oder umgekehrt.



# ANTI-PATTERN

- Big Ball of Mud: Software, die keine erkennbare Softwarearchitektur besitzt
- Gasfabrik: unnötig komplexe Systementwürfe für relativ simple Probleme
- Gottobjekt, Gottklasse und Blob: Objekt, das zu viel weiß bzw. macht
- Spaghetticode: sehr kompakte Systemstruktur, die von Sprungbefehlen geprägt ist, deren Kontrollfluss einem Topf Spaghetti ähnelt
- Sumo-Hochzeit: wenn ein Fat Client unnatürlich stark abhängig von der Datenbank ist
- Integrationsdatenbank: Datenbank, welche von mehreren Anwendungen direkt verwendet wird, um die Synchronisierung zwischen den Anwendungen sicherzustellen. “Integration databases – don’t do it! Seriously! Not even with views. Not even with stored procedures.”
- Zwiebel: Programmcode, bei dem neue Funktionalität um (oder über) die alte gelegt wird.
- Copy and Paste
- Lavafluss: immer mehr „toter Quelltext“ herumliegt
- Magische Werte: Daten (Literele) mit besonderer Bedeutung. Sie sind hartkodiert.

# QUALITÄTSSICHERUNG

- Begriffe der Qualitätssicherung
- Kosten und Nutzen der Qualitätssicherung
- Konstruktive Qualitätssicherung
- Manuelle Prüfmethoden
- Testen (Wiederholung)

## Weitere Literatur:

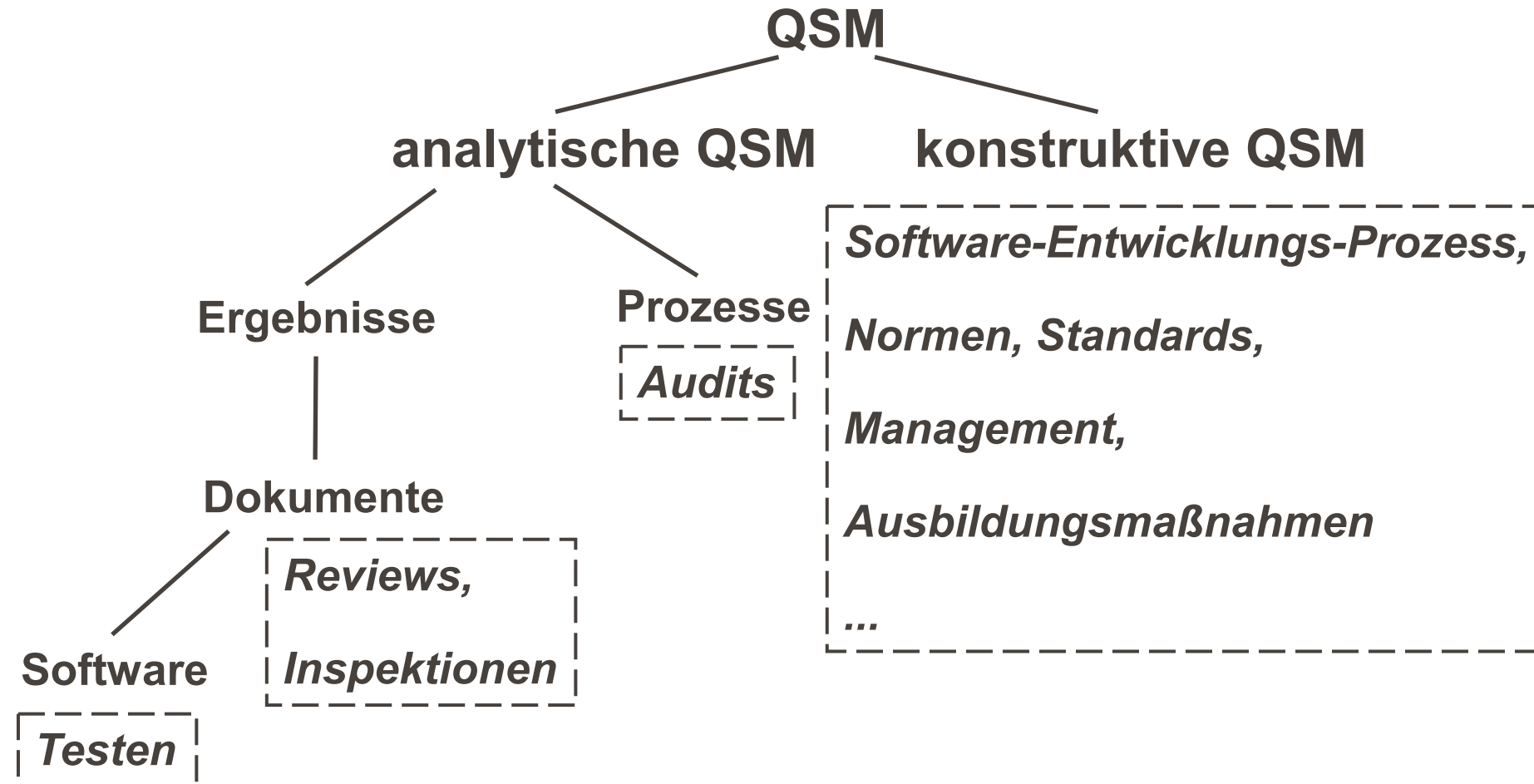
- [Lig02] P. Liggesmeyer, Software- Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002
- [Bal98] H. Balzert, Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998

- Qualitätssicherung (allgemein)  
Alle geplanten und systematischen Tätigkeiten, die notwendig sind, um ein angemessenes Vertrauen zu schaffen, dass ein Produkt oder eine Dienstleistung die gegebenen Qualitätsanforderungen erfüllt.
- Qualitätssicherung (softwarespezifisch)  
Die Gesamtheit aller Maßnahmen und Hilfsmittel, die mit dem Ziel eingesetzt werden, die gestellten Anforderungen an den Entwicklungs- und Wartungsprozess sowie an das Softwareprodukt zu erreichen

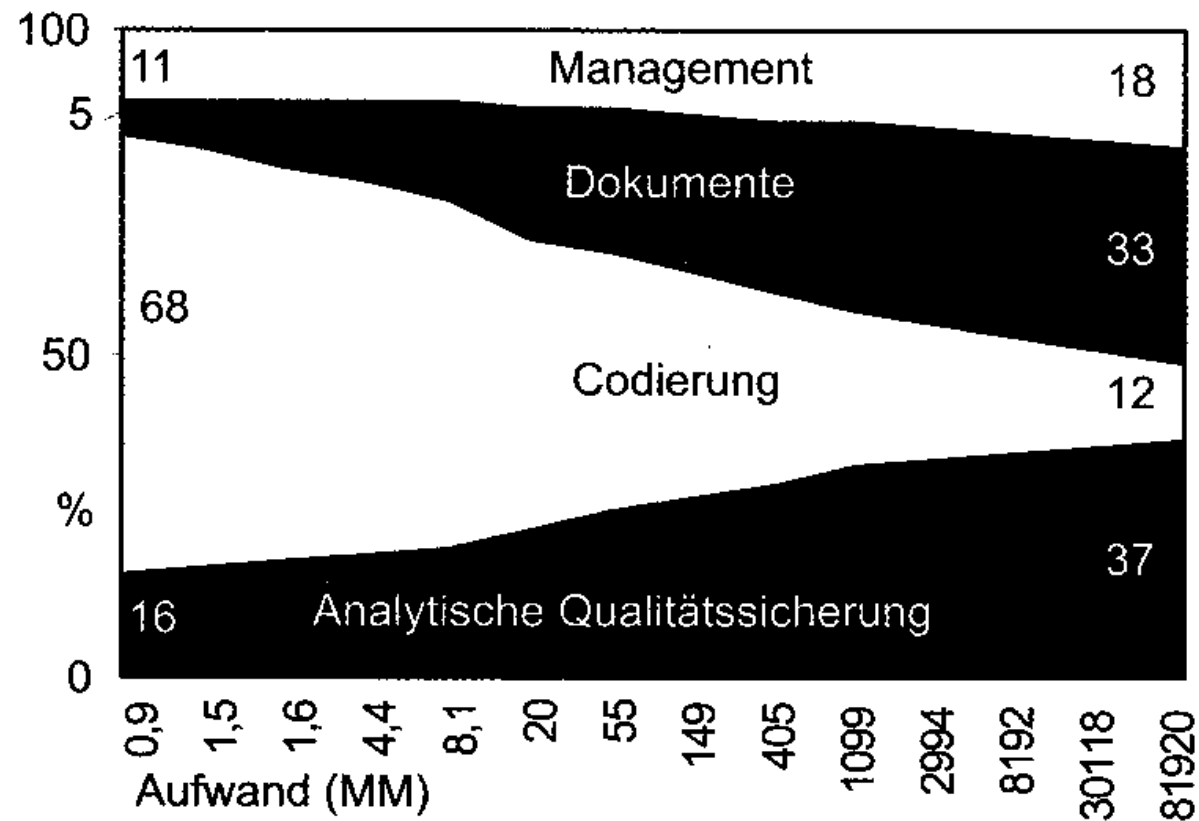
aus DIN 55350-11: Begriffe zu Qualitätsmanagement und Statistik, Teil 11, 8/95



# Qualitätssicherungsmaßnahmen (QSM)

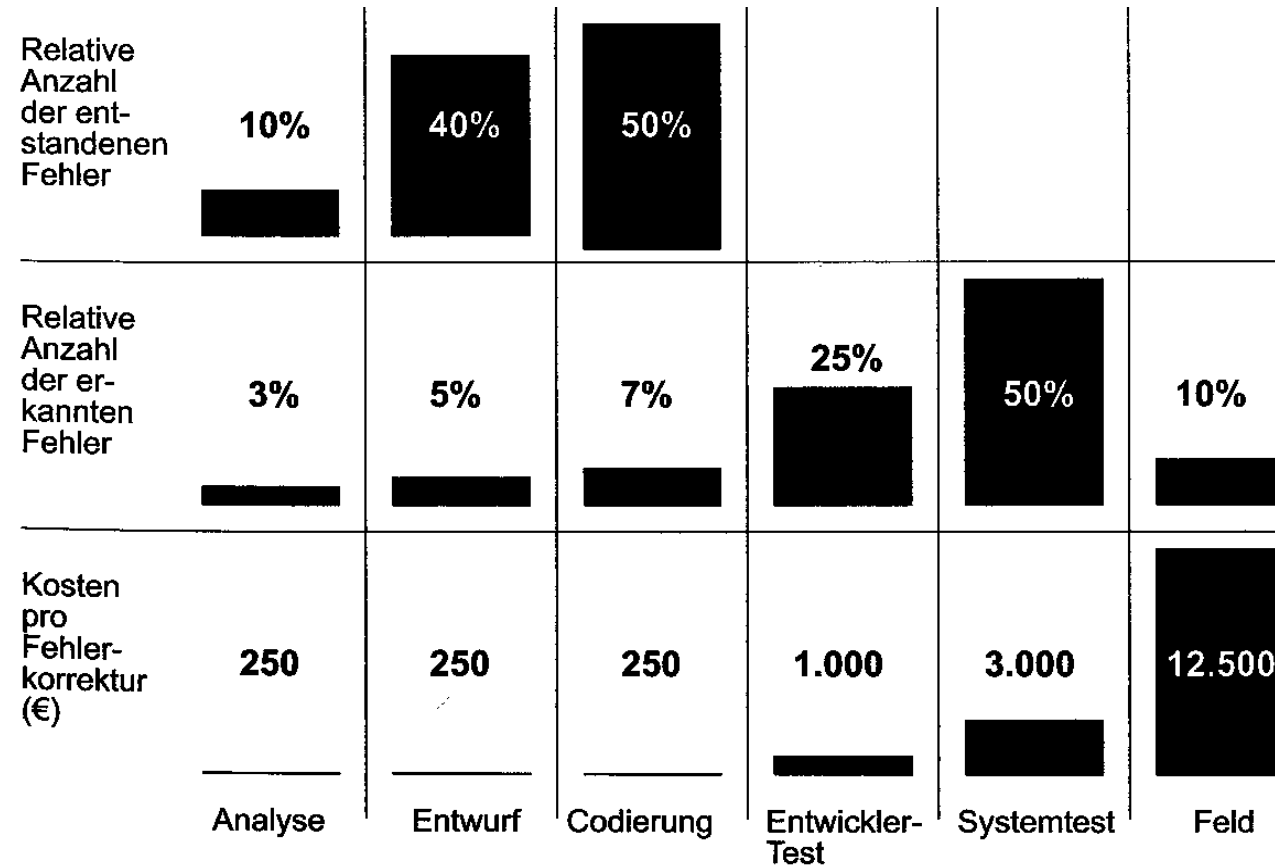


# Aufwand für analytische Qualitätssicherung

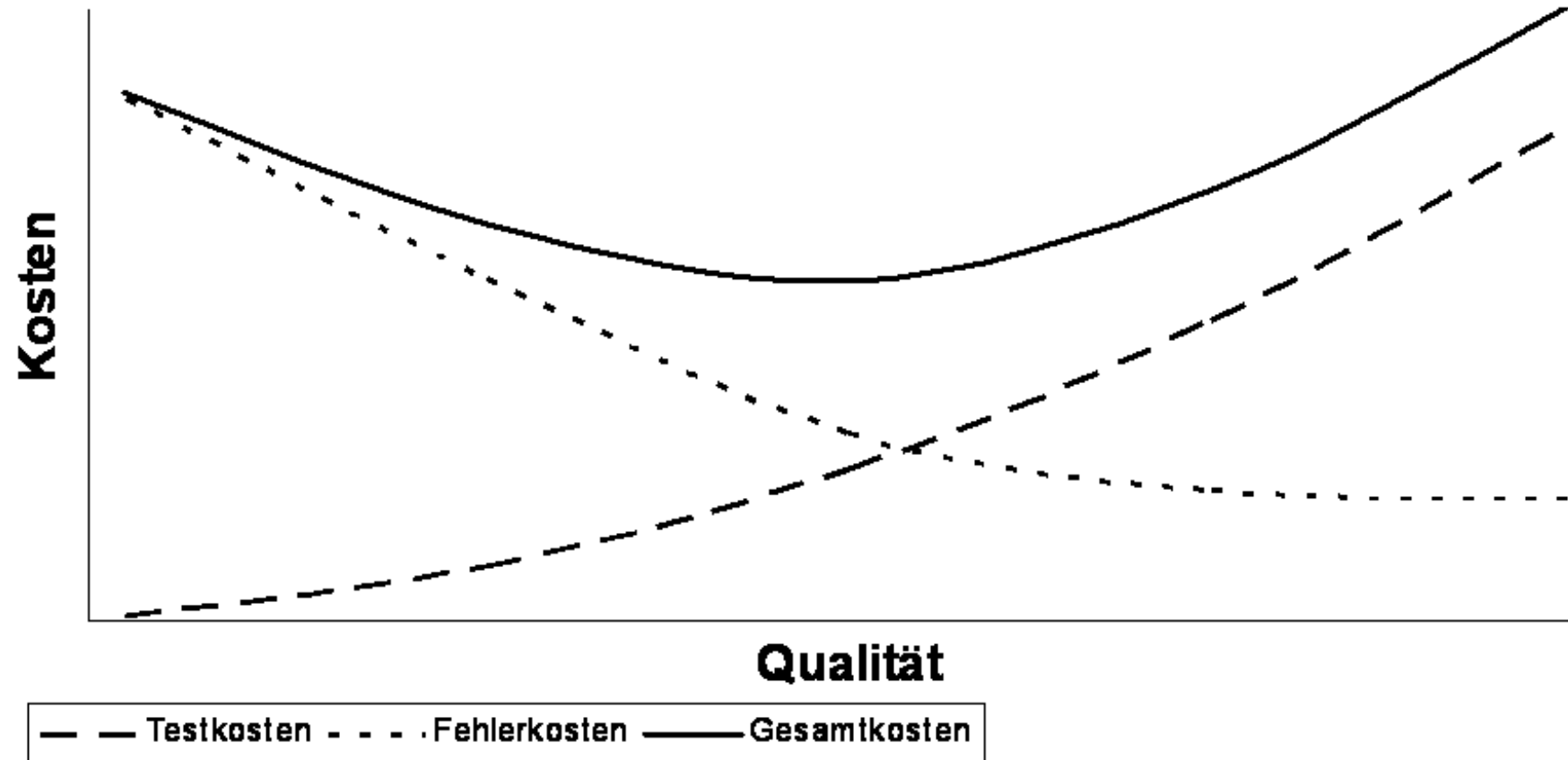


C. Jones, Applied Software Measurement, 1991

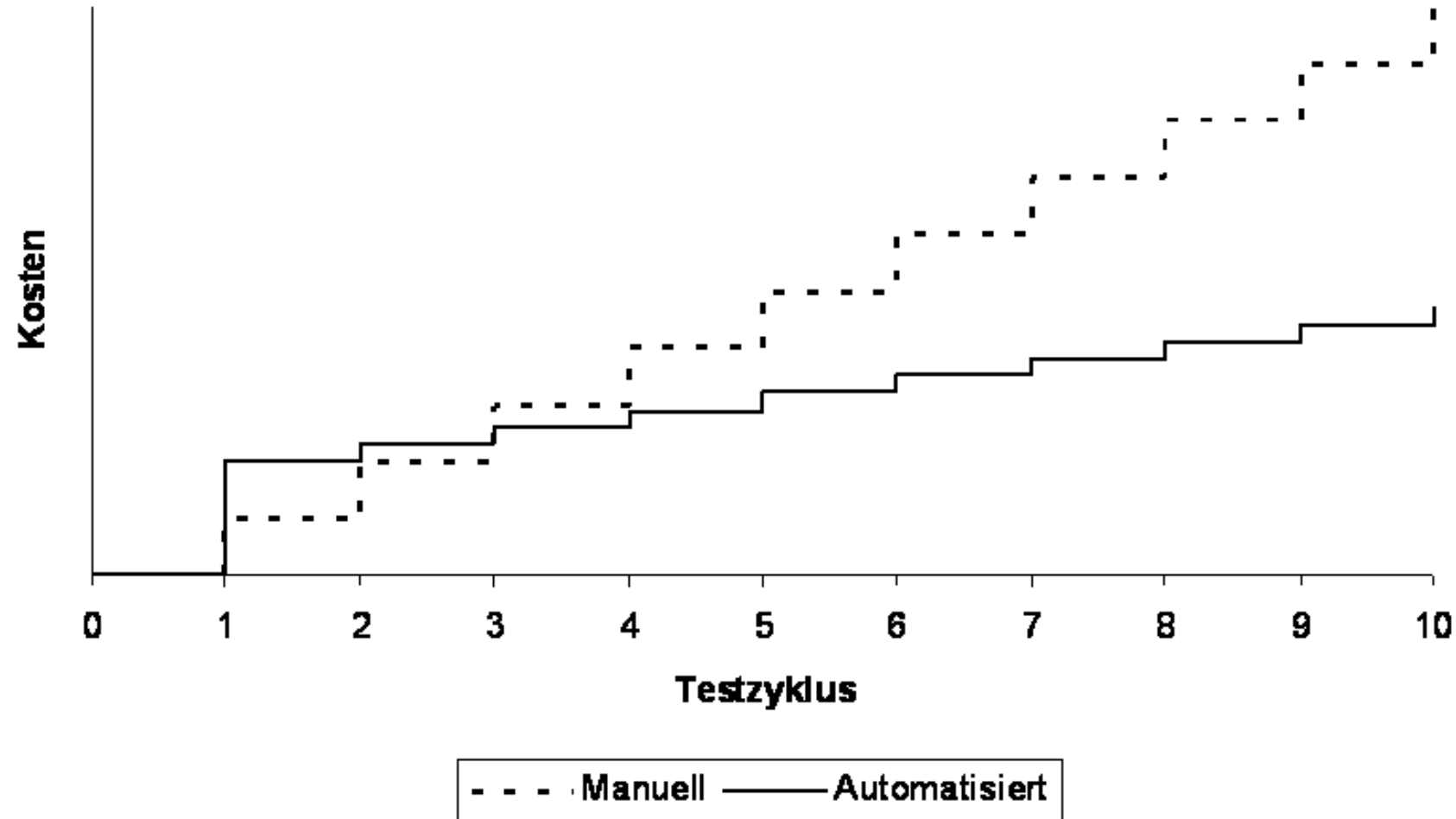
# Wann werden Fehler gefunden?



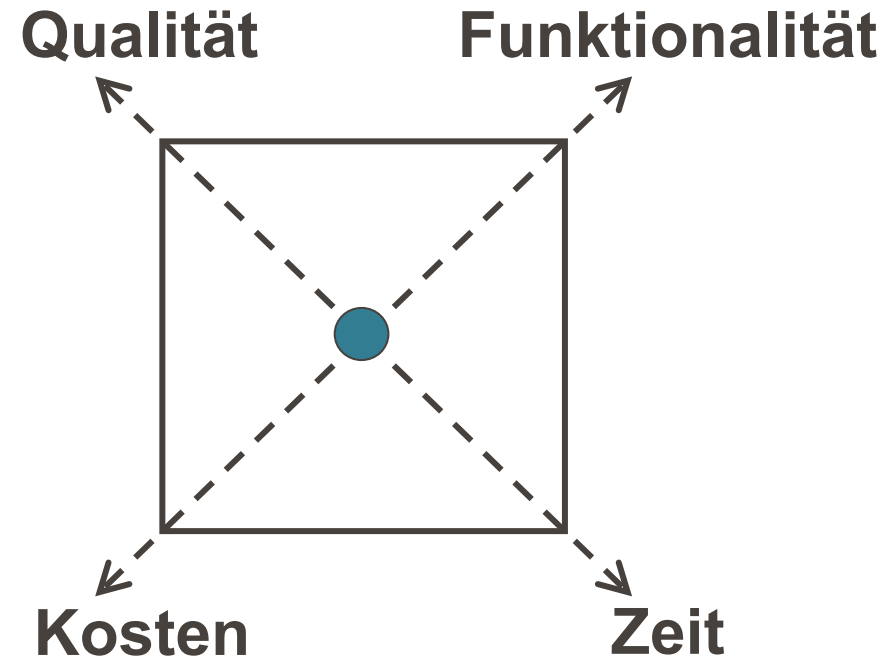
# Was kostet Qualität (1/2)



# Was kostet Qualität (2/2)



# Teufelsquadrat für IT-Projekte



- Konzentration auf ein Ziel erzwingt bei gleichen Rahmenbedingungen Vernachlässigung mindestens eines anderen Ziels (Verbesserung im Bild bedeutet ziehen des Schwerpunkts in diese Richtung)
- generelle Verbesserung nur durch Verbesserung der Rahmenbedingungen (z.B. der Prozesse)

Qualität als Spielball der Wirtschaft:

1. SW-Entwicklung scheint Prozess mit hohem Kostenreduzierungspotenzial
2. Qualitätsprüfungen leisten „offensichtlich“ keinen konstruktiven Beitrag zur Entwicklung, sollen deshalb wenig kosten

aber:

1. Großteil der Zeit der SW-Entwicklung wird für das Debuggen von SW mit zu wenig qualitätsgesicherten früheren Phasen benötigt
2. oft hohe Projektfolgekosten für Wartung nicht ausgereifter SW

Fazit: Qualitätssicherung senkt die Kosten bei der SW-Entwicklung. Nur wenn Sie für Patches bezahlt werden, kann man über andere Philosophien nachdenken.

1. Qualität: Der Begriff der Qualität ist in der DIN 55350-11 definiert als die Beschaffenheit einer Einheit bezüglich ihrer Eignung, festgelegte und abgeleitete Erfordernisse (Qualitätsanforderungen) zu erfüllen
2. Qualitätsanforderung: Der Begriff Qualitätsanforderung bezeichnet die Gesamtheit der Einzelanforderungen an eine Einheit, die die Beschaffenheit dieser Einheit betreffen
3. Qualitätsmerkmal: Die konkrete Beurteilung von Qualität geschieht durch so genannte Qualitätsmerkmale. Diese stellen Eigenschaften einer Funktionseinheit dar, anhand derer ihre Qualität beschrieben und beurteilt wird, die jedoch keine Aussage über den Grad der Ausprägung enthalten. Ein Qualitätsmerkmal kann über mehrere Stufen in Teilmerkmalen verfeinert werden.



1. Qualitätsmaß: Die konkrete Ausprägung eines Qualitätsmerkmals geschieht durch so genannte Qualitätsmaße. Dies sind Maße, die Rückschlüsse auf die Ausprägung bestimmter Qualitätsmerkmale zulassen (Beispiel: Durchschnittliche Antwortzeit für Laufzeiteffizienz)
2. Fehlverhalten: Ein Fehlverhalten oder Ausfall (failure) zeigt sich dynamisch bei der Benutzung eines Produkts. Beim dynamischen Test einer SW erkennt man keine Fehler, sondern Fehlverhalten bzw. Ausfälle.
3. Fehler: Ein Fehler oder Defekt (fault, defect) ist bei SW die statisch im Programmcode vorhandene Ursache eines Fehlverhaltens oder Ausfalls

# Qualitätsmerkmale

---

Korrektheit

---

Sicherheit

---

Zuverlässigkeit

---

Verfügbarkeit

---

Robustheit

---

Speicher- und Laufzeiteffizienz

---

Änderbarkeit

---

Portierbarkeit

---

Prüfbarkeit

---

Benutzbarkeit

---

Anmerkung: Qualitätsmerkmale können Wechselwirkungen haben

- die analytische Qualitätssicherung greift erst, wenn ein Produkt erstellt wurde
- interessant ist der Versuch, Qualität bereits bei der Erstellung zu beachten
- typische konstruktive Qualitätsmaßnahmen sind
  - Vorgabe der SW-Entwicklungsumgebung mit projekteigenem Werkzeughandbuch, was wann wie zu nutzen und zu lassen ist
  - Stilvorgaben für Dokumente und Programme (sogenannte Coding-Guidelines)
- Die Frage ist, wie diese Maßnahmen überprüft werden

1. Detailliertes Beispiel: Taligent-Regeln für C++  
(<http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/index.html>)
2. Sun hat auch Regeln für Java herausgegeben (nicht ganz so stark akzeptiert)
3. z. B. Eclipse-Erweiterung Checkstyle
4. Generell gibt es Regeln
  1. zur Kommentierung,
  2. zu Namen von Variablen und Objekten (z.B. Präfix-Regeln),
  3. zum Aufbau eines Programms (am schwierigsten zu formulieren, da die Programmarchitektur betroffen ist und es nicht für alle Aspekte „die OO-Regeln“ gibt)

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) {
    doSomethingAboutIt();
}

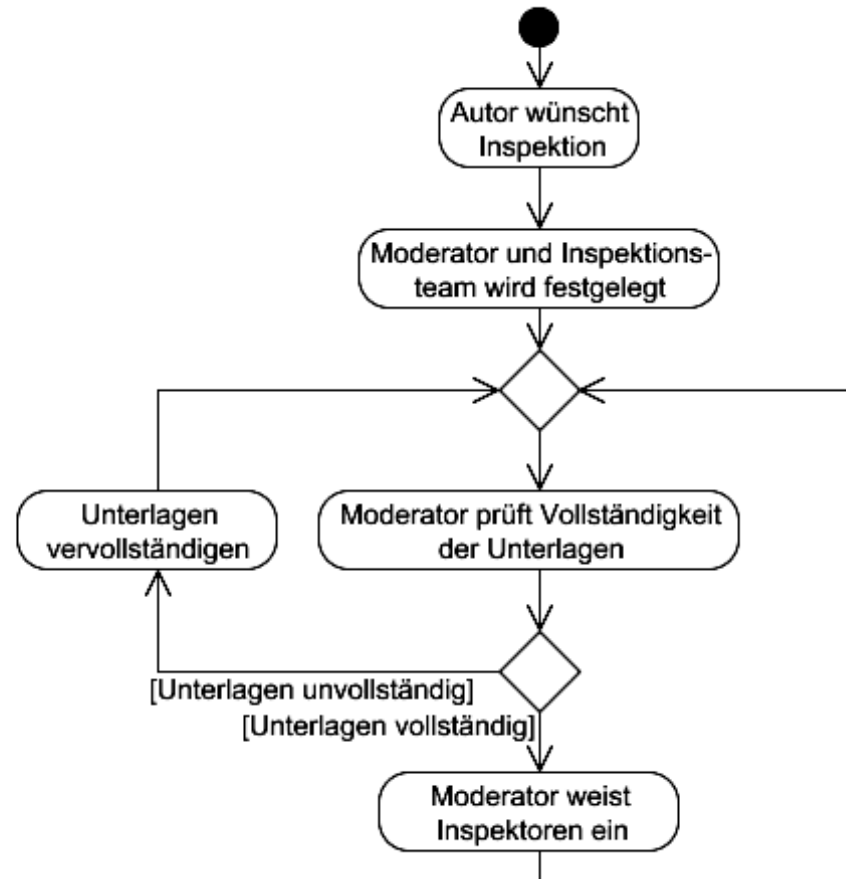
//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
    || !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

- Ausschnitt aus „Java Code Conventions“, Sun, 1997
- Inhalt soll sich nicht nur auf Formatierung beziehen

- Produkte und Teilprodukte werden manuell analysiert, geprüft und begutachtet.
- Ziel ist es, Fehler, Defekte, Inkonsistenzen und Unvollständigkeiten zu finden.
- Die Überprüfung erfolgt in einer Gruppensitzung durch ein kleines Team mit definierten Rollen.
- Anmerkung: Es werden hier Inspektionen, Reviews und Walkthroughs betrachtet (in abnehmender Formalität), in der Literatur ist „Reviews“ teilweise der Oberbegriff

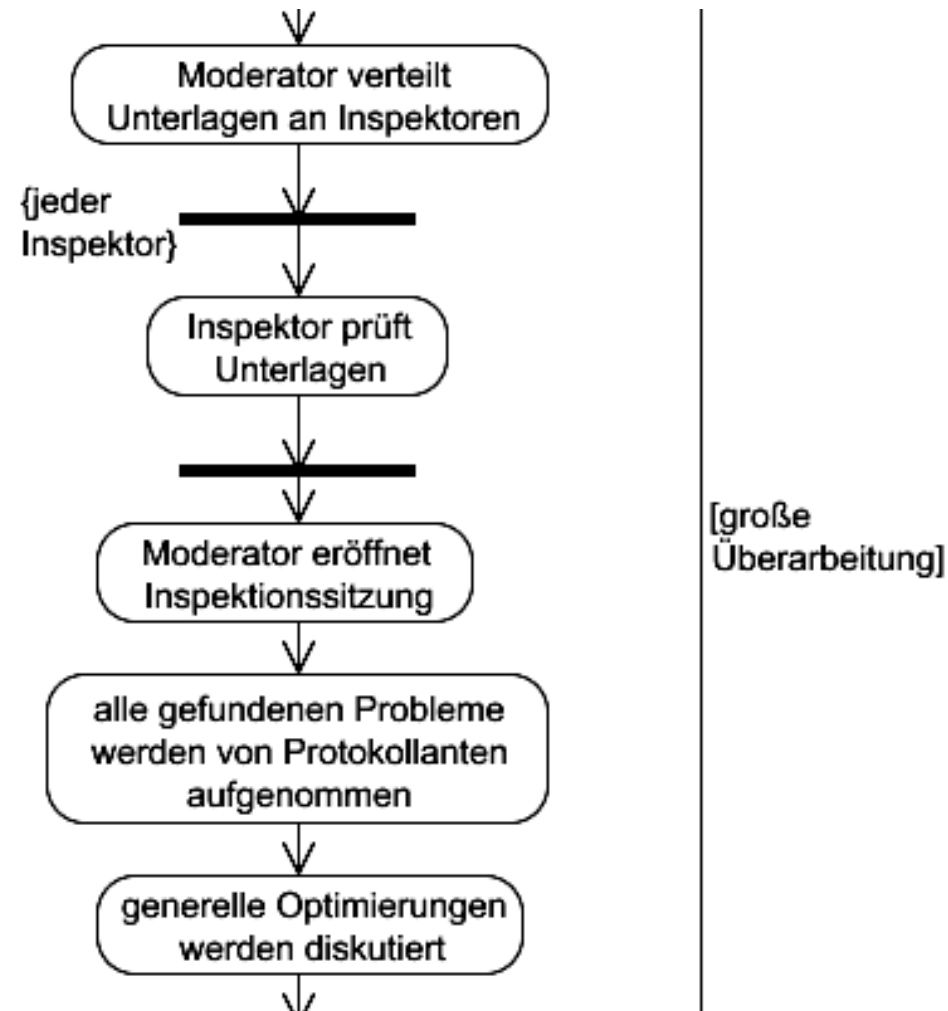
- notwendigen Aufwand und benötigte Zeit einplanen
- Jedes Mitglied des Prüfteams muss in der Prüfmethode geschult sein
- Prüfergebnisse nicht zur Beurteilung von Mitarbeitern nutzen
- Die Prüfmethode muss schriftlich festgelegt und deren Einhaltung überprüft werden
- Prüfungen haben hohe Priorität, d.h. sie sind nach der Prüfbeantragung kurzfristig durchzuführen
- Vorgesetzte und Zuhörer sollen an den Prüfungen *nicht* teilnehmen

# Inspektionsablauf (1/3)

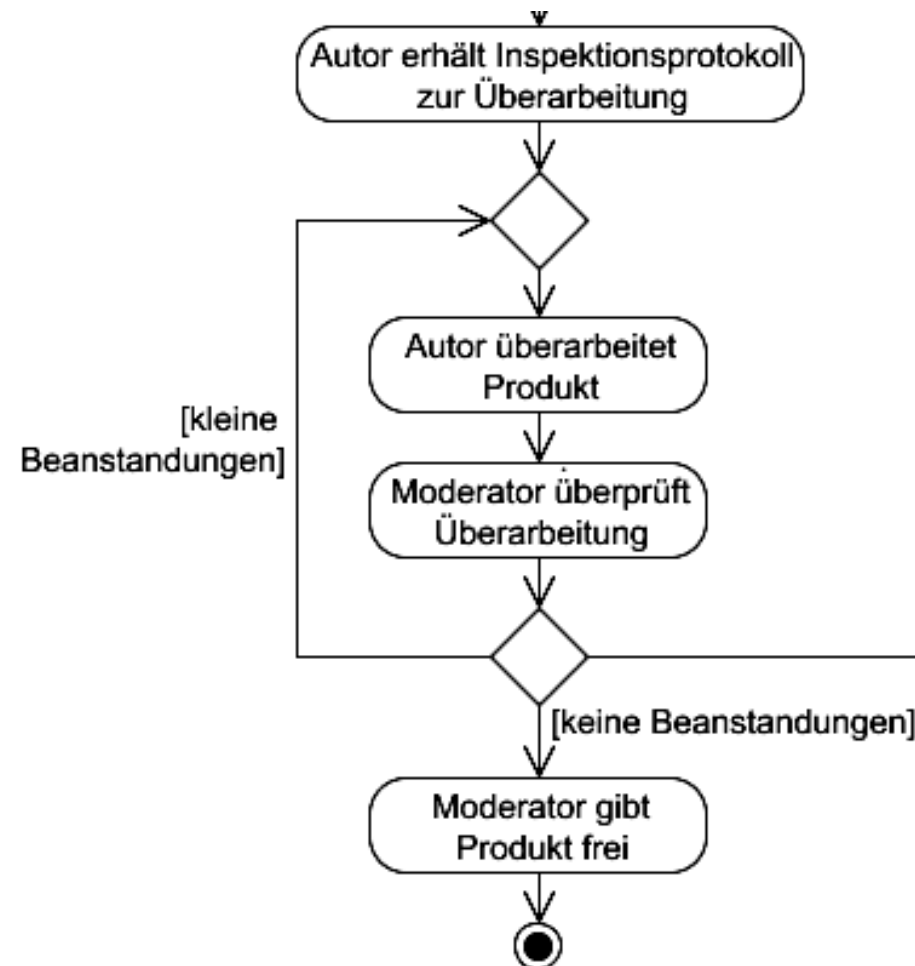




# Inspektionsablauf (2/3)



# Inspektionsablauf (3/3)



## Review

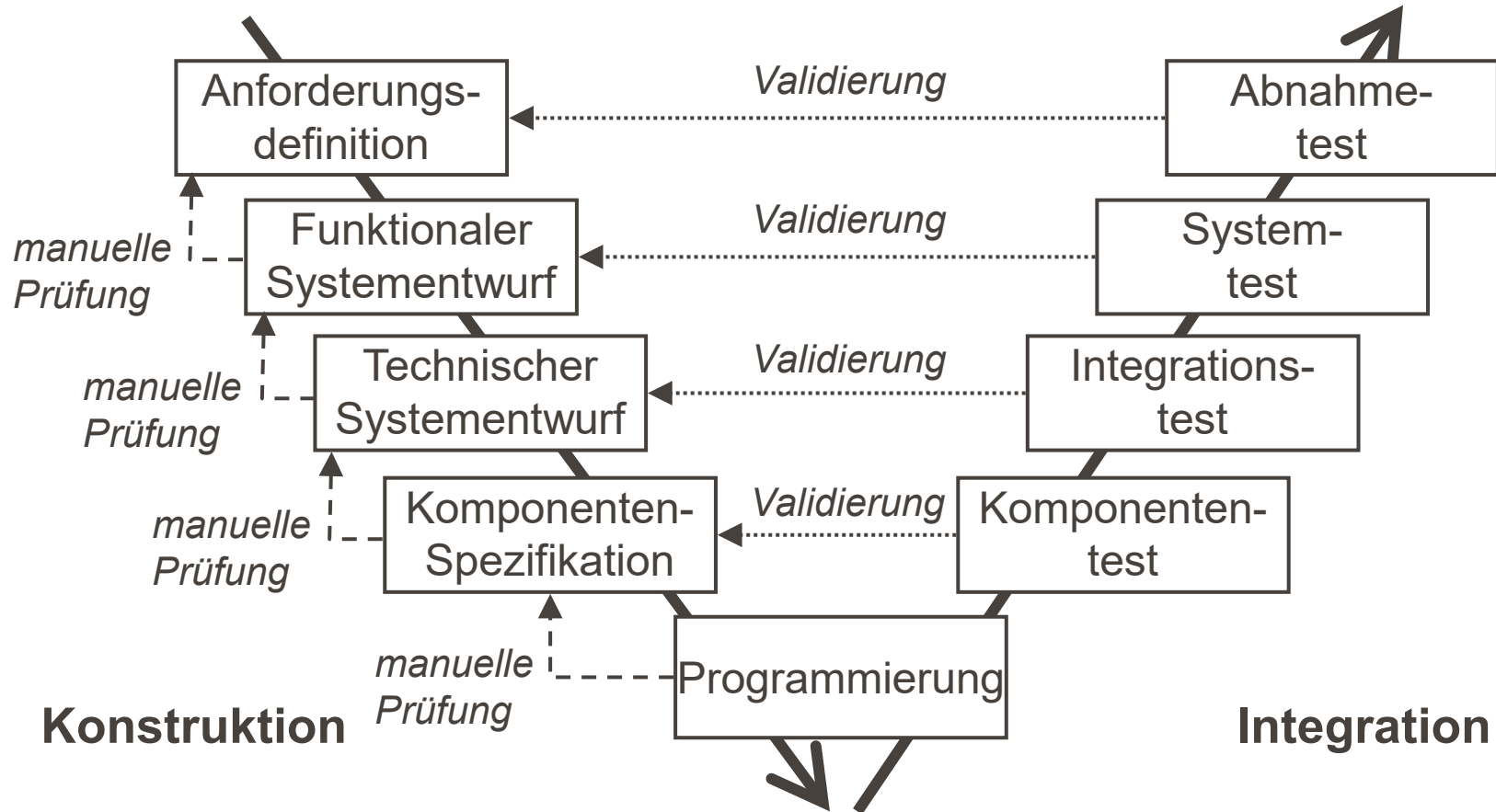
- weniger formal als Inspektion
- Gutachter überprüfen ein schriftliches Dokument, um dessen Stärken und Schwächen festzustellen.
- *Review*-Ablauf ist im wesentlichen wie der Ablauf einer Inspektion. Die Unterschiede bestehen darin, dass:
  - der Autor dabei ist (passiv und um Missverständnisse zu klären),
  - der Autor die verteilten und markierten Kopien des Prüfobjekts erhält und
  - das Review-Team im Falle einer Zurückweisung eine erneute Sitzung verlangen kann.
- Während des Reviews werden keine Lösungen diskutiert! Damit gute Ideen nicht verloren gehen, kann eine formlose „dritte Stunde“ angehängt werden.

## Walkthrough

- ist abgeschwächte Form des *Reviews*.
- Der Autor leitet als Moderator die *Walkthrough*-Sitzung. Zu Beginn [evtl. früher] der Sitzung erhält jeder Gutachter eine Kopie des Prüfobjektes.
- Das Prüfobjekt wird Schritt für Schritt vorgestellt und die Gutachter stellen spontane Fragen und versuchen so, mögliche Probleme zu identifizieren. Diese werden protokolliert.
- *Walkthrough*-Variante besteht darin, vor der Sitzung eine individuelle Vorbereitung durchzuführen.

- + Oft die einzige Möglichkeit, Semantik zu überprüfen.
- + Notwendige Ergänzung werkzeuggestützter Überprüfungen
- + Die Verantwortung für die Qualität der geprüften Produkte wird vom ganzen Team getragen.
- + Da die Überprüfungen in einer Gruppensitzung durchgeführt werden, wird die Wissensbasis der Teilnehmer verbreitert.
- + Jedes Mitglied des Prüfteams lernt die Arbeitsmethoden seiner Kollegen kennen.
- + Die Autoren bemühen sich um eine verständliche Ausdrucksweise, da mehrere Personen das Produkt begutachten.
- + Unterschiedliche Produkte desselben Autors werden von Prüfung zu Prüfung besser, d.h. enthalten weniger Fehler.
- In der Regel aufwändig (bis zu 20 Prozent der Erstellungskosten des zu prüfenden Produkts).
- Autoren geraten u.U. in eine psychologisch schwierige Situation (»sitzen auf der Anklagebank«, »müssen sich verteidigen«).

# allgemeines V-Modell



- Korrekt bedeutet: Unter folgenden Vorbedingungen sollen die Ergebnisse des Programms folgende Nachbedingungen erfüllen

- Wunsch alles zu testen meist nicht erfüllbar

```
public static boolean impliziert(boolean a,  
                                boolean b){  
    return !a || b;  
}  
public static void main(String[] args) { // Tests  
    boolean werte[]={true,false};  
    for(boolean a:werte)  
        for(boolean b:werte)  
            System.out.println("a="+a+" b="+b  
                               +" liefert "+impliziert(a,b));  
}
```

- Tests sollen möglichst viele möglichst kritische Fehler finden können

- Idee: Gegeben ist ein Modell (oder Spezifikation) und eine Anforderung, dann überprüft der Model-Checking-Algorithmus, ob das Modell die Anforderung erfüllt oder nicht. Falls das Modell die Anforderung nicht erfüllt, sollte der Algorithmus ein Gegenbeispiel liefern
- Theoretisches Ergebnis: Generell gibt es so ein Verfahren nicht (unentscheidbar), man muss Modellierungssprache und Anforderungslogik einschränken
- Ansatz: Berechne alle erreichbaren Zustände und analysiere sie
- Beispiel: SPIN entwickelt von Gerard Holzmann, zunächst als Simulator, dann Verifikationswerkzeug ([www.spinroot.com](http://www.spinroot.com))



# Assertions in JDK ab Version 1.4

- Sprachänderung
  - neues Keyword `assert`
- Erweiterung des `java.lang`-Pakets  
`java.lang.AssertionError`
- Neue Dokumentation
  - unter Guide to Features
  - unter Assertion Facility
  - unter javadoc  
`doc/guide/lang/assert.html`
- Compiler-Option `-source 1.4`  
`javac -source 1.4 Klasse.java`  
damit `javac` `assert` akzeptiert
- Interpreter-Option `-ea`  
`java -ea Klasse`

# Syntax und Semantik von assert

`assert` überprüft zur Laufzeit eine Zusicherung, ist diese nicht erfüllt,  
wird ein `AssertionError` ausgelöst

```
public void (int a, int b){  
    assert a>b && exemplar>5;  
    ...  
}
```

## Syntax:

```
assert <Boolescher_Ausdruck>  
assert <Boolescher_Ausdruck>:<bel_Objekt>
```

`<bel_Objekt>` ist meist ein `String`, der bei der Fehlermeldung mit  
angezeigt wird

- Zusicherungen sollen nichts kosten, wenn sie ausgeschaltet sind [nicht immer zu 100% möglich].

- Falsch:

```
private static double EPS=0.000001;

public static double sqrt(double par) {
    double res = Math.sqrt( par );
    boolean test = Math.abs(par -res *res) < EPS;
    assert test;
    return res;
}
```

- Richtig:

```
assert Math.abs( par - res * res ) < EPS;
```

# Vorbedingungsregel

- nur interne Vorbedingungen mit `assert` prüfen, die bei korrekter Software erfüllt sein müssten
- kein Ersatz für Ausnahmen

Beispiel:

```
public double sqrt(double param) {  
    if ( param < 0 ) throw  
        new IllegalArgumentException("Negativer Parameter");  
    return Math.sqrt( param );  
}
```

- Nicht als zweite Zeile:  
`assert param >= 0;`
- deswegen `AssertionError` und nicht `AssertionException`
- Assertions dürfen nur in der Entwicklung zu Fehlern führen

- Nachbedingungen lassen sich immer mit `assert` beschreiben
- Typisch stehen Nachbedingungen unmittelbar vor jedem `return`-Statement oder der abschließenden geschweiften Klammer
- Beispiel

```
private static double EPS=0.000001;
public static double sqrt(double par) {
    double res = Math.sqrt( par );
    assert Math.abs( par - res * res ) < EPS;
    return res;
}
```

- Invarianten von Objekten werden nach außen nie verletzt (bei internen Berechnungen erlaubt)
- Ansatz: Invariante wird als eigenständige Methode implementiert

```
private boolean invariant(...){  
    boolean erg = <Invariantenberechnung>;  
    return erg;  
}
```
- Überprüfung der Invariante `assert invariant(...);` am
  - Ende eines jeden Konstruktors
  - am Anfang jeder public-Methode
  - am Ende jeder public-Methode

# Fehlermöglichkeiten mit assert

```
while (iter.hasNext()) {  
    assert iter.next() != null;  
    do_something( iter.next() );  
}
```

```
public class Stack {  
    int size = 0;
```

---

```
    public int length() {  
        assert this.length() >= 0;  
        return size;  
    } // length  
} // Stack
```

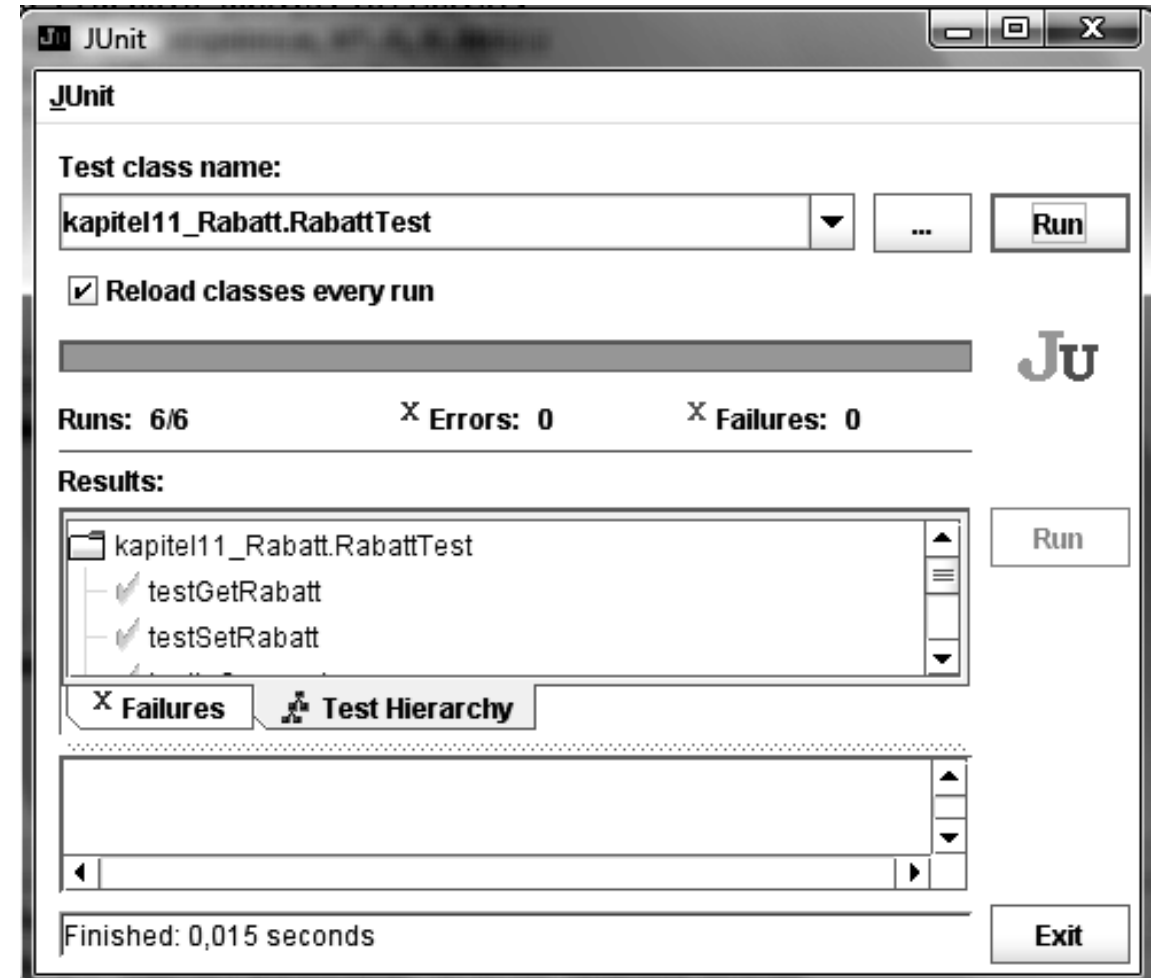
Vor dem Testen müssen Testfälle spezifiziert werden

- Vorbedingungen
  - Zu testende Software in klar definierte Ausgangslage bringen (z. B. Objekte mit zu testenden Methoden erzeugen)
  - Angeschlossene Systeme in definierten Zustand bringen
  - Weitere Rahmenbedingungen sichern (z. B. HW)
- Ausführung
  - Was muss wann gemacht werden (einfachster Fall: Methodenaufruf)
- Nachbedingungen
  - Welche Ergebnisse sollen vorliegen (einfachster Fall: Rückgabewerte)
  - Zustände anderer Objekte / angeschlossener Systeme



# JUNIT

- Framework, um den Unit-Test eines Java-Programms zu automatisieren
- einfacher Aufbau
- leicht erlernbar
- geht auf SUnit (Smalltalk) zurück
- mittlerweile für viele Sprachen verfügbar



- Testfälle werden in Java programmiert, keine spezielle Skriptsprache notwendig
- Idee ist inkrementeller Aufbau der Testfälle parallel zur Entwicklung
  - Pro Klasse wird mindestens eine Test-Klasse implementiert
- JUnit ist in Eclipse integriert, sonst muss das Paket junit.jar zu CLASSPATH hinzugefügt werden.
- wenn junit heruntergeladen wird, ist Klassen-Dokumentation unter \junit3.8.2\javadoc\index.html lesbar (<http://sourceforge.net/projects/junit/>)
- weitere Dokumentation unter \junit3.8.2\doc

- Jede Testklasse wird von der Framework-Basisklasse `junit.framework.TestCase` abgeleitet.
- JUnit erkennt die Testfälle anhand des Signaturmusters (Methodenname beginnt mit `test`)  
`public void testXXX()`
- Das Framework kann mit dem Java-Reflection Package die Testfälle somit automatisch erkennen.
- Die `assertEquals` Methode dient dazu, eine Bedingung zu testen.
- Ist eine Bedingung nicht erfüllt, d.h. `false`, protokolliert JUnit einen Testfehler.
- Der `junit.swingui.TestRunner` stellt eine grafische Oberfläche dar, um die Test kontrolliert ablaufen zu lassen.

- Die Klasse Assert definiert eine Menge von assertXY Methoden, welche die Testklassen erben.
- Mit den assertXY Methoden können unterschiedliche Behauptungen über den zu testenden Code aufgestellt werden.
- Trifft eine Behauptung nicht zu, wird ein Testfehler protokolliert.
- `assertTrue(boolean condition)`  
verifiziert, ob eine Bedingung wahr ist.
- `assertEquals(Object expected, Object actual)`  
verifiziert, ob zwei Objekte gleich sind. Der Vergleich erfolgt mit der equals Methode.
- `assertEquals(int expected, int actual)`  
verifiziert, ob zwei ganze Zahlen gleich sind. Der Vergleich erfolgt mit dem `==` -Operator.
- `assertNull(Object object)`  
verifiziert, ob eine Objektreferenz null ist.

- `assertEquals(double expected, double actual, double delta)`  
verifiziert, ob zwei Fließkommazahlen gleich sind. Da Fließkommazahlen nicht mit unendlicher Genauigkeit verglichen werden können, kann mit delta ein Toleranzwert angegeben werden.
- `assertNotNull(Object object)`  
verifiziert, ob eine Objektreferenz nicht null ist.
- `assertSame(Object expected, Object actual)`  
verifiziert, ob zwei Referenzen auf das gleiche Objekt verweisen.
- Für die primitiven Datentypen float, long, boolean, byte, char und short existieren ebenfalls `assertEquals` Methoden.
- Jede Assert-Methode gibt es mit einem zusätzlichen ersten Parameter vom Typ String, dieser String wird zum Fehler ausgegeben (also sinnvoll diese Methoden zu nutzen)

- Ein Testfall sieht in der Regel so aus, dass eine bestimmte Konfiguration von Objekten aufgebaut wird, gegen die der Test läuft.
- Diese Menge von Testobjekten wird auch als Test-Fixture bezeichnet.
- Damit fehlerhafte Testfälle nicht andere Testfälle beeinflussen können, wird die Test-Fixture für jeden Testfall neu initialisiert.
- In der Methode setUp werden Exemplarvariablen initialisiert.
- Mit der Methode tearDown werden wertvolle Testressourcen wie zum Beispiel Datenbank- oder Netzwerkverbindungen wieder freigegeben.

1. Exceptions werden in Java mit dem catch Block behandelt.
2. Mit der Methode fail aus der Assert Klasse wird ein Testfehler ausgelöst und protokolliert.
3. Im Beispiel wird beim Aufruf des Konstruktors der Klasse MyClass mit einer negativen Zahl die IllegalArgumentException ausgelöst.

```
public void testNegativeInitialisierung() {  
    try {  
        new MyClass(-10);  
        fail("Meine Meldung im Fehlerfall");  
    } catch (IllegalArgumentException e) {}  
}
```

- Um eine Reihe von Tests zusammen ausführen zu können, werden die Tests zu TestSuites zusammengefasst.
- Eine Zusammenfassung von Tests wird dabei durch ein `TestSuite` Objekt definiert.
- Mit JUnit können beliebig viele Tests in einer TestSuite zusammengefasst werden.
- Um eine Testsuite zu definieren, ist ein `TestSuite` Objekt zu bilden und mittels der `addTestSuite` Methode verschiedene Testfallklassen hinzuzufügen.
- Jede Testfallklasse definiert implizit eine eigene `suite` Methode, in der alle Testfallmethoden der betreffenden Klasse eingebunden werden. [Reflection Package]
- In vielen Fällen ist es praktisch, pro Package eine TestSuite-Klasse zu definieren.
- Es hat sich eingebürgert diese Klasse `AllTest` zu nennen.



- Bei Objekten mit internen Zuständen ist der Test von außen mit JUnit sehr schwierig
- es ist oftmals hilfreich, zusätzliche Methoden zu implementieren, die das Testen einfacher machen  

```
public String getInternenZustand(){ return "..."} 
```
- Häufiger reicht ein get für den internen Zustand nicht aus, es muss Methoden geben, mit denen man ein Objekt von außen in einen gewünschten Zustand versetzen kann (Ergänzung weiterer Testmethoden)
- Im Quellcode sollen Testmethoden eindeutig von anderen Methoden getrennt sein, damit sie ggfls. automatisch gelöscht werden können
- Bei komplexeren Klassen sollte man Teile der Entwicklung bereits testen, hierzu müssen die aufgerufenen Methoden zumindest in minimaler Form implementiert werden

---

Bis jetzt wurde nur eine Klasse betrachtet, die keine Assoziation zu anderen zu testenden Klassen hat, diese Klassen werden elementare Klassen genannt

---

Grundsätzlich sollte man versuchen, zunächst elementare Klassen und dann Klassen, die auf diesen aufbauen, zu Testen

---

Da es in der Entwicklung nicht garantiert werden kann, wann Klassen vorliegen, muss man sich häufiger mit einem Trick behelfen

---

Dieser Trick ist es, die benötigte Klasse soweit selbst zu implementieren, dass man die eigene Klasse testen kann. Diese Klasse wird Mock genannt.

---

Die Grundregel lautet, den Mock so primitiv wie möglich zu halten

---

Liegt die Klasse vor, die man temporär durch den Mock prüfen wollte, müssen diese Tests mit der realen Klasse wiederholt werden

- Es wird zunächst eine Klasse mit den notwendigen Methoden implementiert, die alle die leere Implementierung oder die Rückgabe eines Dummy-Werts enthalten

```
public void setParameter(int parameter){}
public int getParameter() { return 0;}
```
- Diese Implementierung wird soweit ergänzt, dass wir unsere Klasse testen können (möglichst einfache Fallunterscheidungen, man bedenke, dass man von der Korrektheit der anderen Klasse ausgeht)
- Es gibt Werkzeuge, die die einfachst möglichen Mocks automatisch generieren, die können dann ergänzt werden
- D. h., neben den Tests entsteht ein zusätzlicher Codieraufwand für Mocks, in größeren (erfolgreichen) Projekten kann der Anteil des Testcodes am Gesamtcode in Abhängigkeit von der Komplexität des Systems zwischen 30% und 70% liegen!

- Äquivalenzklassenbildung zerlegt Menge in disjunkte Teilmengen
- jeder Repräsentant einer Teilmenge hat das gleiche Verhalten bzgl. einer vorgegebenen Operation
- Beispiel: Restklassen (modulo  $x$ ), werden zwei beliebige Repräsentanten aus Restklassen addiert, liegt das Ergebnis immer in der selben Restklasse
- Übertragungsidee auf Tests: Eingaben werden in Klassen unterteilt, die durch die Ausführung des zu testenden Systems zu „gleichartigen“ Ergebnissen führen



# BEISPIELE FÜR ÄQUIVALENZKLASSEN VON EINGABEN

- erlaubte Eingabe:  $1 \leq \text{Wert} \leq 99$  (Wert sei ganzzahlig)
  - eine gültige Äquivalenzklasse:  $1 \leq \text{Wert} \leq 99$
  - zwei ungültige Äquivalenzklassen:  $\text{Wert} < 1$ ,  $\text{Wert} > 99$
- erlaubte Eingabe in einer Textliste: für ein Auto können zwischen einem und sechs Besitzer eingetragen werden
  - eine gültige Äquivalenzklasse: ein bis sechs Besitzer
  - zwei ungültige Äquivalenzklassen: kein Besitzer, mehr als sechs Besitzer
- erlaubte Eingabe: Instrumente Klavier, Geige, Orgel, Pauke
  - vier gültige Äquivalenzklassen: Klavier, Geige, Orgel, Pauke
  - eine ungültige Äquivalenzklasse: alles andere, z.B. Zimbeln

- man muss mögliche Eingaben kennen (aus Spezifikation)
- für einfache Zahlenparameter meist einfach:
  - Intervall mit gültigen Werten
  - eventuell Intervall mit zu kleinen und Intervall mit zu großen Werten (wenn z. B. alle int erlaubt, gibt es nur eine Äquivalenzklasse, etwas schwieriger bei double))
- explizit eine Menge von Werten vorgegeben:
  - jeder Wert eine Äquivalenzklasse dar
  - andere Eingaben möglich: dies zusätzliche Äquivalenzklasse
- falls nach Analyse der Spezifikation Grund zur Annahme besteht, dass Elemente einer Äquivalenzklasse unterschiedlich behandelt werden, ist die Klasse aufzuspalten

## Spezifikation:

- Als Beispiel dient eine Methode, genauer ein Konstruktor, zur Verwaltung von Studierendendaten, der ein Name, ein Geburtsjahr und ein Fachbereich übergeben werden. Dabei darf das Namensfeld nicht leer sein, das Geburtsjahr muss zwischen 1900 und 2000 liegen und es können nur die Fachbereiche FBING, FBBWL und FBPOL aus einer Aufzählung übergeben werden.

## BEISPIEL (2/5)

Äquivalenzklassen:

<b>Eingabe</b>	<b>gültige Äquivalenzklassen</b>	<b>ungültige Äquivalenzklassen</b>
<b>Name</b>	Ä1) nicht leer	Ä2) leer
<b>Geburtsjahr</b>	Ä4) $1900 < = \text{Geburtsjahr} \leq 2000$	Ä3) Geburtsjahr $< 1900$ Ä5) Geburtsjahr $> 2000$
<b>Fachbereich</b>	Ä6) FBING Ä7) FBBWL Ä8) FBPOL	



- Die Äquivalenzklassen sind eindeutig zu nummerieren. Für die Erzeugung von Testfällen aus den Äquivalenzklassen sind zwei Regeln zu beachten:
- gültige Äquivalenzklassen:
  - möglichst viele Klassen in einem Test kombinieren
- ungültige Äquivalenzklassen:
  - Auswahl eines Testdatums aus einer ungültigen Äquivalenzklasse
  - Kombination mit Werten, die ausschließlich aus gültigen Äquivalenzklassen entnommen sind.
  - Grund: für alle ungültigen Eingabewerte muss eine Fehlerbehandlung existieren

## Beispiel (3/5)

Testfälle nach einer Äquivalenzklassenanalyse: (jeder Klasse wird [mindestens] einmal getestet, die Testanzahl soll möglichst gering sein)

Test-nummer	1	2	3	4	5	6
geprüfte Äquivalenzklassen	Ä1 Ä4 Ä6	(Ä1) (Ä4) Ä7	(Ä1) (Ä4) Ä8	Ä2	Ä3	Ä5
Name	„Meier“	„Schmidt“	„Schulz“	„“	„Meier“	„Meier“
Geburtsjahr	1987	1989	1985	1988	1892	2006
Fachbereich	FBING	FBBWL	FBPOL	FBING	FBING	FBING
Ergebnis	ok	ok	ok	Abbruch	Abbruch	Abbruch

- Viele Software-Fehler sind auf Schwierigkeiten in Grenzbereichen der Äquivalenzklassen zurück zu führen (z.B. Extremwert nicht berücksichtigt, Array um ein Feld zu klein)
- Aus diesem Grund wird die Untersuchung von Äquivalenzklassen um die Untersuchung der Grenzen ergänzt
- Beispiel:  $1 \leq \text{Wert} \leq 99$  (wobei Wert ganzzahlig ist)
  - Äquivalenzklasse  $\text{Int-Wert} < 1$ : obere Grenze  $\text{Wert} = 0$  (untere Grenze spielt hier keine Rolle)
  - Äquivalenzklasse  $\text{Int-Wert} > 99$ : untere Grenze  $\text{Wert} = 100$  (obere Grenze spielt keine Rolle)
  - Äquivalenzklasse  $1 \leq \text{Int-Wert} \leq 99$  : untere Grenze  $\text{Wert} = 1$  und obere Grenze  $\text{Wert} = 99$
- Diese Grenzfallbetrachtung kann direkt in die Testfallerzeugung eingehen (es gibt Ansätze, bei denen zusätzlich ein Fall mit einem Wert aus der „Mitte“ der Äquivalenzklasse genommen wird)

# Beispiel (4/5)

- Testfälle nach einer Äquivalenzklassenanalyse und Grenzwertanalyse
- Anmerkung: Testfallanzahl erhöht sich meist

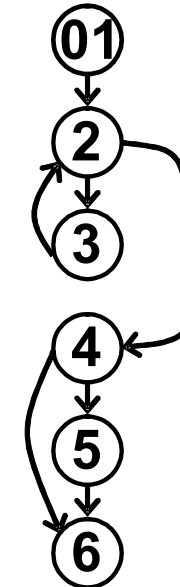
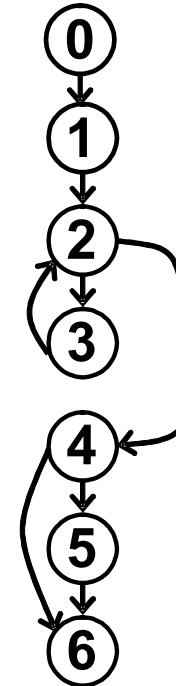
Test-nummer	1	2	3	4	5	6
geprüfte Äquivalenzklassen	Ä1 Ä4U Ä6	(Ä1) Ä4O Ä7	(Ä1) (Ä4) Ä8	Ä2	Ä3O	Ä5U
Name	„Meier“	„Schmidt“	„Schulz“	„“	„Meier“	„Meier“
Geburtsjahr	1900	2000	1985	1988	1899	2001
Fachbereich	FBING	FBBWL	FBPOL	FBING	FBING	FBING
Ergebnis	ok	ok	ok	Abbruch	Abbruch	Abbruch

1. Äquivalenzklassenbildung ist ein zentrales Verfahren, um systematisch Tests aufzubauen
2. für Methoden von Objekten spielt neben Ein- und Ausgaben der interne Zustand häufig eine wichtige Rolle (erst wenn Methode x ausführt wurde, dann kann Methode y sinnvoll ausgeführt werden)
3. Konsequenterweise muss man sich also mit dem Ein-/Ausgabeverhalten pro möglichem Objektzustand beschäftigen (was noch extrem aufwändiger sein kann)
4. das bisher vorgestellte Verfahren kann in der reinen Form nur für gedächtnislose Objekte genutzt werden

- eines Programms  $P$  ist ein gerichteter Graph  $KFG(P) =_{\text{def}} G = (V, E, V_{\text{Start}}, V_{\text{Ziel}})$
- $V$ : Menge der Knoten (Anweisungen des Programms)
- $E$  ist Teilmenge von  $V \times V$ : Menge der Kanten (Nachfolgerrelation bezüglich der Ausführung des Programms)
- $V_{\text{Start}}, V_{\text{Ziel}}$  aus  $V$  : Ausgewählte Knoten für Start, Ende des Programms ( $V_{\text{Ziel}}$  kann auch eine Menge von Knoten sein)

# Beispiel: KFG

```
public int aha(int ein) {  
    int erg=-ein/2;           // 0  
    int i=ein;               // 1  
    while(i>0) {             // 2  
        erg=erg+(i--);       // 3  
    }  
    if (ein <0 || ein%2==1) { // 4  
        erg=0;               // 5  
    }  
    return erg*2;            // 6  
}
```



- Wunsch: Graph sollte unabhängig von der Formatierung sein!
- Programm gibt Quadrat aller positiven geraden Zahlen, sonst 0 zurück
- Testergebnisse werden vereinfachend bei folgenden Testfallbeschreibungen weggelassen

# Normalisierung eines KFG durch Blockung

folgende Regeln, mit denen mehrere Knoten  $k_1, k_2, \dots, k_n$ , die nacheinander durchlaufen werden können, also  $k_1 \square k_2 \square \dots \square k_n$ , zu einem Knoten verschmolzen werden.

- Die Knotenfolge wird bei jedem Durchlauf immer nur über  $k_1$  betreten, es gibt außer den genannten Kanten keine weiteren Kanten, die in  $k_2, \dots, k_n$  enden.
- Die Knotenfolge wird bei jedem Durchlauf immer nur über  $k_n$  verlassen, es gibt außer den genannten Kanten keine weiteren Kanten, die in  $k_1, \dots, k_{n-1}$  beginnen.
- Die Knotenfolge ist maximal bezüglich a) und b).



- Ein *vollständiger Pfad* ist eine Folge von verbundenen Knoten (über Kanten) im KFG, die mit Vstart beginnt, und mit Vziel endet.
- Die möglichen Ausführungsreihenfolgen des Programms sind eine Teilmenge der vollständigen Pfade
- Wunsch: Durchlauf „repräsentativer“ vollständiger Pfade beim Test
- Überdeckung aller vollständigen Pfade ist im allgemeinen nicht ausführbar
- Ansatz: Verschiedene Approximationsstufen (Anweisungsüberdeckungstest, ..., Mehrfach-Bedingungsüberdeckungstest) für die Menge der vollständigen Pfade bei Auswahl der Testdurchläufe wählen

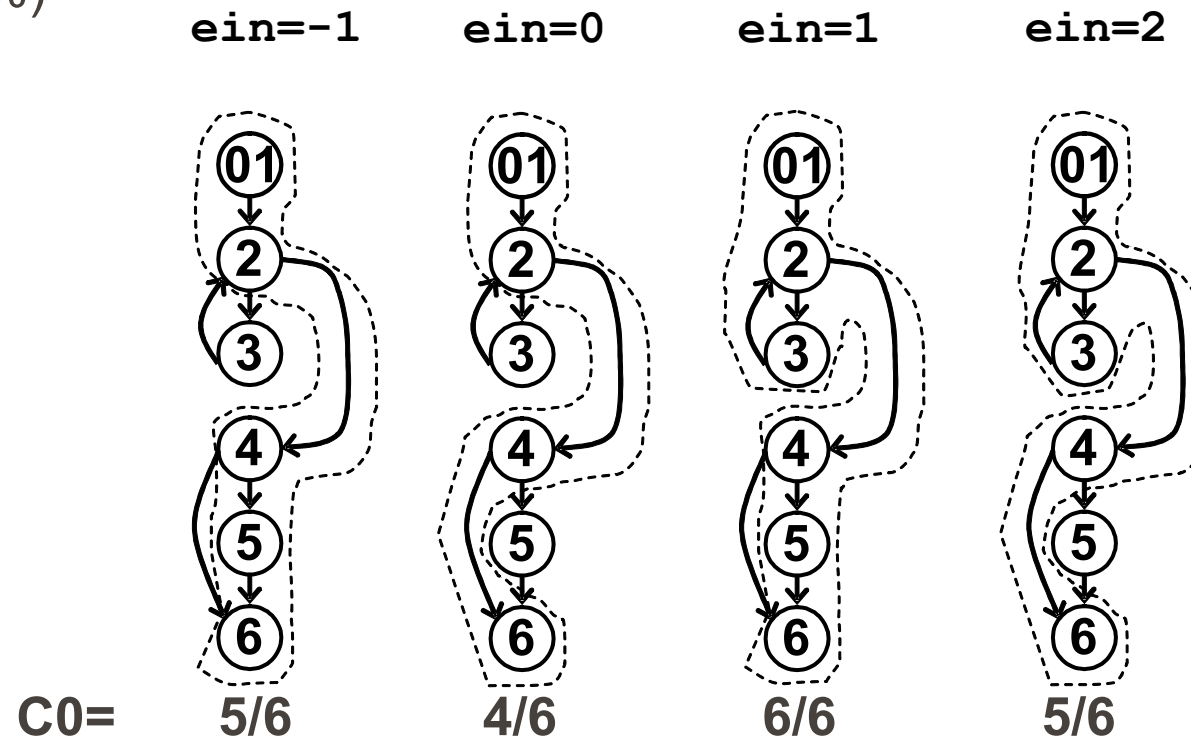
# Anweisungsüberdeckung (C0)

- Ziel: alle Anweisungen des Programms durch Wahl geeigneter Testdaten mindestens einmal ausführen, alle Knoten des KFG mindestens einmal besuchen.

- Testmaß 
$$C0 = \frac{\text{Anzahl der ausgeführten Knoten}}{|V|}$$

- Ziel:  $C0=1$  (=100%)

- für  $\{-1\}$   
 $C0 = 5/6$
- für  $\{-1, 0\}$   
 $C0 = 5/6$
- für  $\{-1, 2\}$   
 $C0 = 6/6$



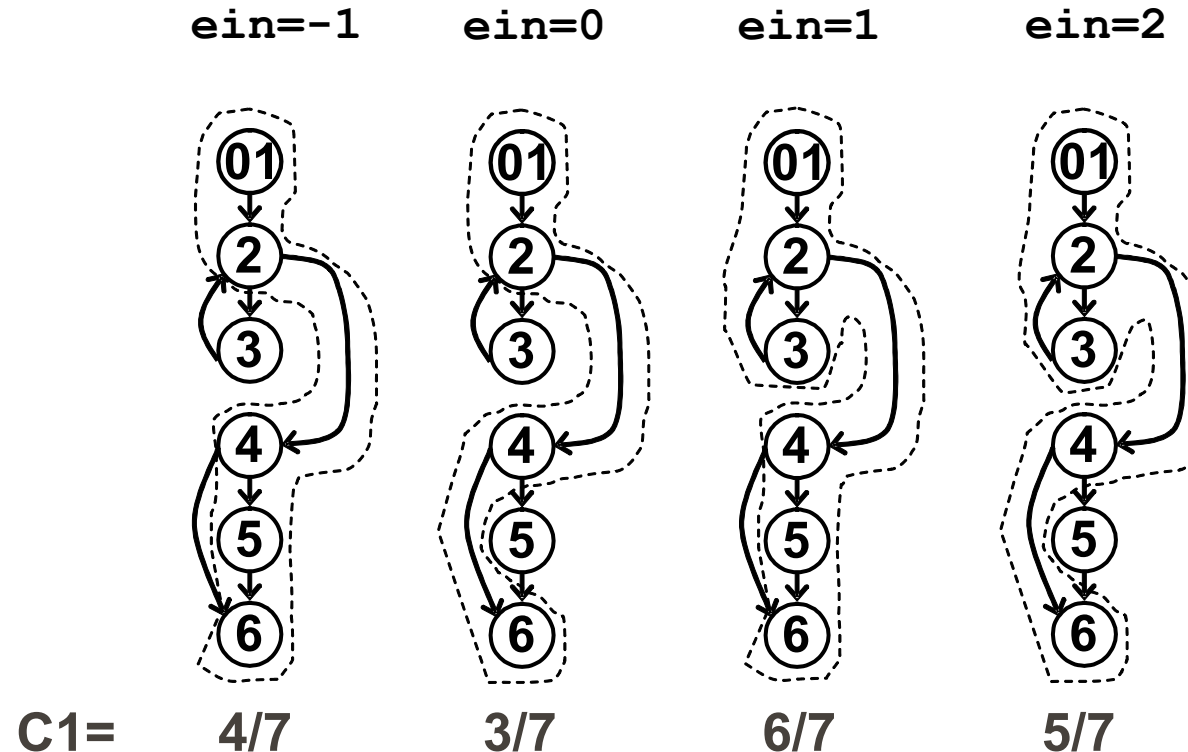
# Zweigüberdeckung (C1)

- Ziel :alle Kanten des KFG überdecken, d.h. alle Zweige des Programms einmal durchlaufen
- Testmaß

$$C1 = \frac{\text{Anzahl der durchlaufenen Kanten}}{|E|}$$

- Ziel: C1=1

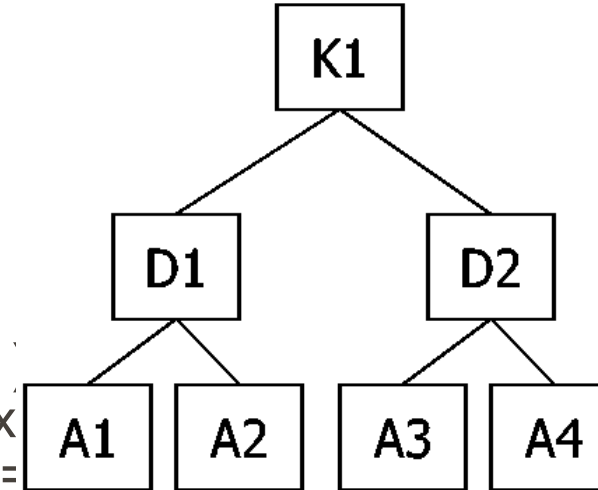
- für {-1}  
C1= 4/7
- für {-1,0}  
C1= 5/7
- für {-1,2}  
C1= 7/7



- Vorteile der Anweisungsüberdeckung:
  - einfach
  - geringe Anzahl von Eingabedaten
  - nicht ausführbare Programmteile werden erkannt
- großer Nachteil der Anweisungsüberdeckung:
  - Logische Aspekte werden nicht überprüft
- deshalb: Zweigüberdeckungstest gilt als das Minimalkriterium im Bereich des dynamischen Softwaretests,
  - schließt den Anweisungsüberdeckungstest ein,
  - fordert die Ausführung aller Zweige eines KFG,
  - jede Entscheidung mindestens einmal wahr und falsch
- Nachteile der Zweigüberdeckung:
  - Fehlende Zweige werden nicht automatisch entdeckt
  - Kombinationen von Zweigen sind unzureichend geprüft
  - Komplexe Bedingungen werden nicht analysiert
  - Schleifen werden nicht ausreichend analysiert

# Bedingungsüberdeckungstest

- Ziel: Teste gezielt Bedingungen in Schleifen und Auswahlkonstrukten
- Bedingungen sind Prädikate
  - A1,..., A4 atomar
- z.B.  $(x==1)$  [auch  $!(x==1)$ ]
- zusammengesetzt
- Konjunktion K
- Disjunktion D
- $((x==1) \parallel (x==2)) \&\& ((y==3) \parallel (y==4))$   
7 Teilprädikate:  $x==1, x==2, y==3, y==4, (x==1) \parallel (x==2), (y==3) \parallel (y==4), ((x==1) \parallel (x==2)) \&\& ((y==3) \parallel (y==4))$
- Hinweis: Bei Java und anderen Programmiersprachen ist der Unterschied zwischen  $\parallel$  und  $\&\&$  (mit Kurzschlussauswertung) und  $|$  und  $\&$  (ohne Kurzschlussauswertung) zu beachten  
`if(true || 5/0==0)` läuft, `if(true | 5/0==0)` läuft nicht



# Einfache Bedingungsüberdeckung (C2)

- Ziel: alle atomaren Prädikate einmal TRUE, einmal FALSE
- Testmaß:
$$C2 = \frac{|wahre\ Atome| + |falsche\ Atome|}{2 * |alle\ Atome|}$$
- $|alle\ Atome|$  = Anzahl aller Atome,  $|wahre\ Atome|$  = Anzahl aller Atome, die mit true ausgewertet werden (analog  $|falsche\ Atome|$ )
- Hinweis: Namensgebung in Literatur bei C2/C3 nicht eindeutig

Testfälle	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}	{-1,1,2}
$i > 0$	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
$ein < 0$	t	f	f	f	t,f	f	f	t,f	t,f
$ein \% 2 == 1$	-	f	t	f	f	f,t	t,f	t	t,f
C2-Überdeckung	2/6	3/6	4/6	4/6	4/6	5/6	5/6	5/6	6/6

# Kein Zusammenhang zwischen C1 und C2

- Die Nummerierung ist historisch gewachsen
- betrachte `if (a || b)`
  - `a=true, b=false`                      `a=false, b=true`
  - garantiert vollständige C2-Überdeckung
  - else-Zweig wird nicht durchlaufen, kein C1 oder C0
    - `a=true, b=false`                      `a=false, b=false`
  - if- und else-Zweig wird durchlaufen, damit C1 und C0
  - keine vollständige C2-Überdeckung, da `b=true` fehlt

# Minimale Mehrfachüberdeckung (C3)

- Ziel: alle Prädikate und Teil-Prädikate einmal TRUE, einmal FALSE
- Testmaß:

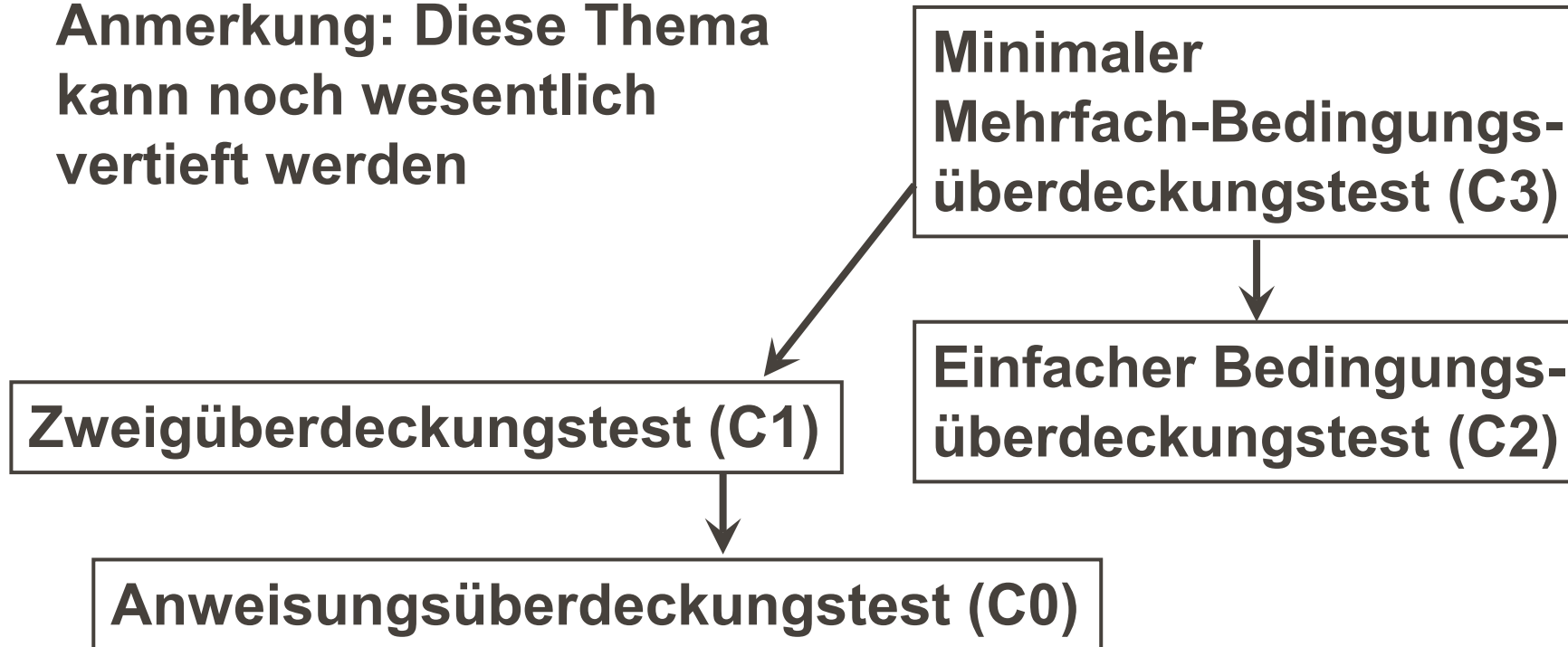
$$C3 = \frac{|wahre\ Teilprädikate| + |falsche\ Teilprädikate|}{2 * |alle\ Teilprädikate|}$$

Testfälle	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}	{-1,1,2}
i>0	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
ein<0	t	f	f	f	t,f	f	f	t,f	t,f
ein%2==1	-	f	t	f	f	f,t	t,f	t	t,f
ein<0    ein%2==1	t	f	t	f	t,f	f,t	t,f	t	t,f
C3-Über- deckung	3/8	4/8	5/8	5/8	6/8	7/8	7/8	6/8	8/8



# Zusammenhang zwischen den Testverfahren

**Anmerkung: Diese Thema  
kann noch wesentlich  
vertieft werden**



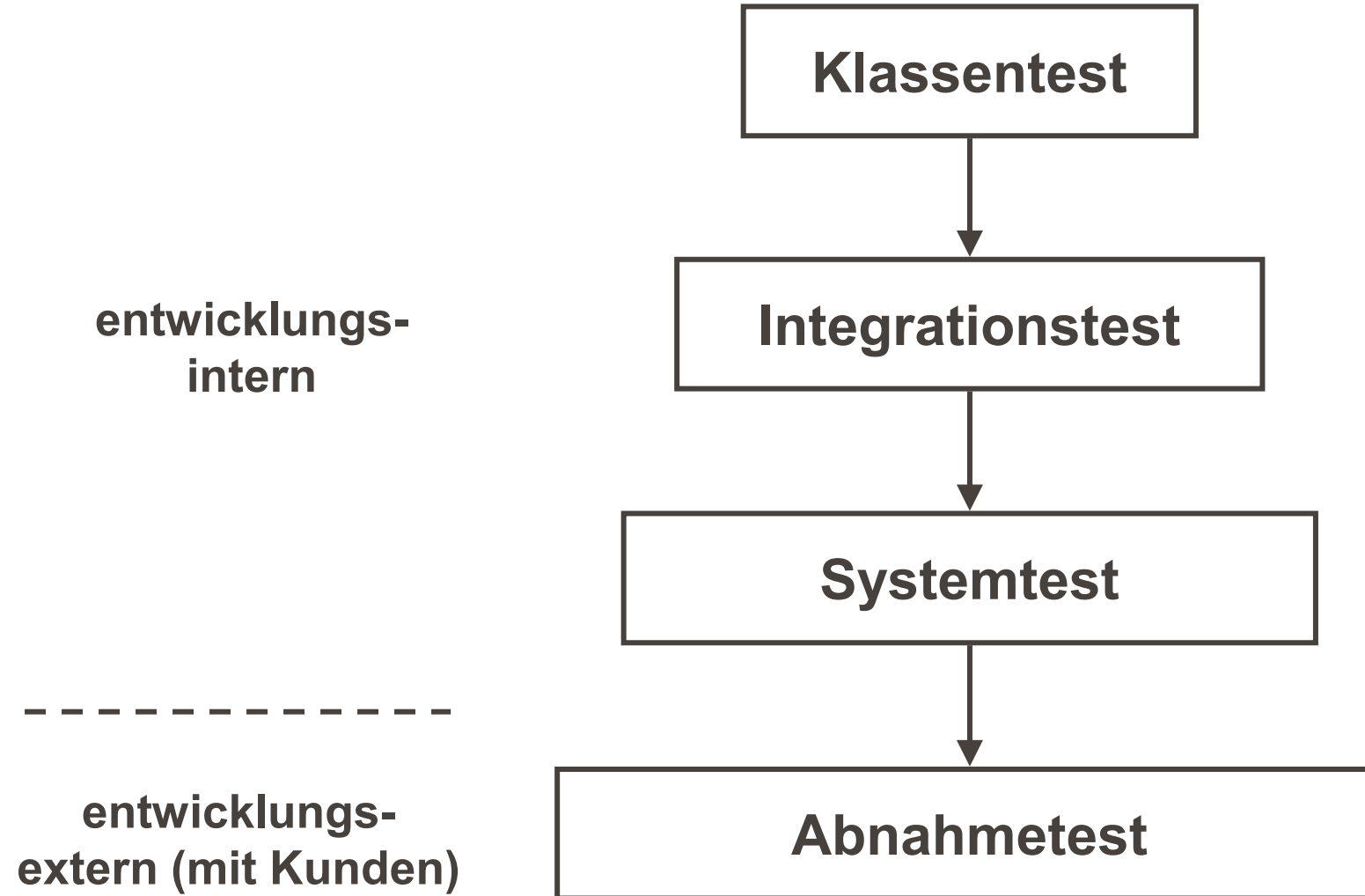
vollständige Überdeckung des einen bedeutet  
vollständige Überdeckung des anderen

- Ohne eine Tool-Unterstützung ist es sehr aufwändig und fehlerträchtig, solche Überprüfungen zu machen
- Für andere Sprachen als Java (und C#) sind Coverage-Werkzeuge vorhanden, die man bzgl. der zu untersuchenden Überdeckung einstellen kann
- Für Java gibt es nur verschiedene Programme, die den Anweisungsüberdeckungstest und teilweise den Zweigüberdeckungstest (CodeCover, Eclemma, Coverlipse, JCoverage, Hansel für JUnit, Clover) unterstützen

# Info: Standard DO-178B

- Um die Zertifizierung der Federal Aviation Administration (FAA) zu erhalten, muss Software für Luftverkehrssysteme den Richtlinien des Standards DO-178B für requirement-basiertes Testen und Code Coverage Analysen genügen.
- DO-178B-Levels orientieren sich an den Konsequenzen möglicher Softwarefehler: katastrophal (Level A), gefährlich/schwerwiegend (Level B), erheblich (Level C), geringfügig (Level D) bzw. keine Auswirkungen (Level E).
- Je nach DO-178B-Level wird der 100%-ige Nachweis folgender Testabdeckungen (Code Coverages) verlangt:
- DO-178B Level A:
  - Modified Condition Decision Coverage (MC/DC)
  - Branch/Decision Coverage
  - Statement Coverage
- DO-178B Level B:
  - Branch/Decision Coverage
  - Statement Coverage
- DO-178B Level C:
  - Statement Coverage

# Teststufen (grober Ablauf)



- Varianten:
  - *Unit-Test*: einzelne Methoden und/oder Klassen
  - *Modultest*: logisch-zusammengehörige Klassen, z.B. ein Package in Java
- Testziel: Prüfung gegen *Feinspezifikation*
  - Architektur, Design, Programmierkonstrukte
- Testmethode: *White-Box-Test*
- *Alle Module* müssen getestet werden
  - eventuell mit unterschiedlicher Intensität

- Module werden zu einem System integriert und gemeinsam getestet
- Testziele:
  - Werden Schnittstellen richtig benutzt?
  - Werden Klassen bzw. ihre Methoden richtig aufgerufen?
- Konzentration auf (*Export-*) *Schnittstellen*
  - Interne Schnittstellen können nicht mehr direkt beeinflusst werden
  - Geringere Testtiefe als beim Modultest
  - Grey-Box-Test (oder auch Black-Box)
- Techniken ähnlich wie bei Modultest
  - Pfadanalyse über die komplette Interaktion der Module oft nicht mehr sinnvoll
- Mit *minimaler Systemkonfiguration* beginnen, Integrationsstrategie spielt eine Rolle

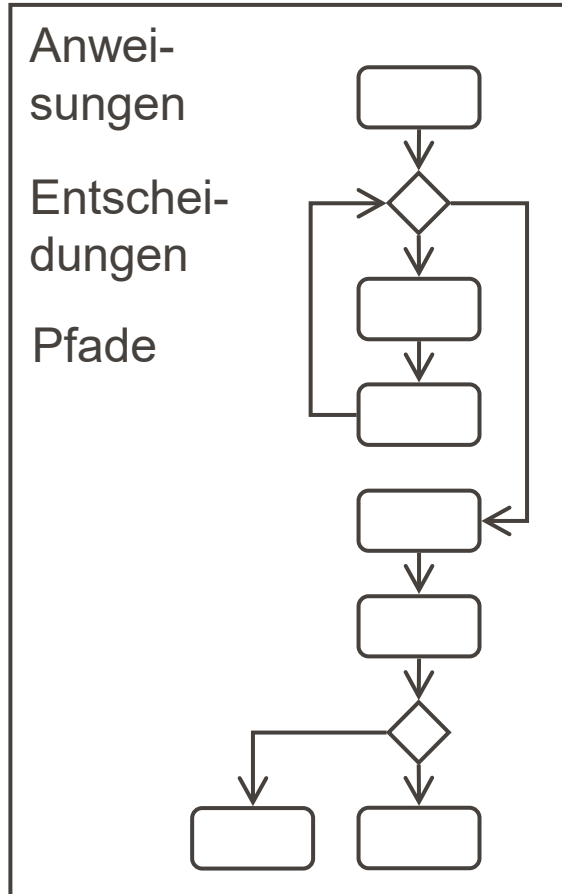
- Orientierung an den spezifizierten Systemaufgaben (z.B. Use Cases)
- Interaktion mit den (simulierten) Nachbarsystemen
- (endgültige) Validierung der *nicht-funktionalen* Anforderungen, z.B. Skalierbarkeit, Verfügbarkeit, Robustheit, ...

- wichtiger organisatorischer Prozess der QS
- Wiederherstellung des ursprünglichen Zustands der Testdaten
- Autarke Datenbestände:  
Kein gegenseitiges “Zerschießen“ der Testdaten
- für DB: Kollisionen in den Nummernkreisen der Schlüsselattribute vermeiden
- Datumsfelder können altern. Wie können Testdaten zum aktuellen Datum passen?
  - Tagesdatum aus (manipuliertem) Kalender-Modul holen



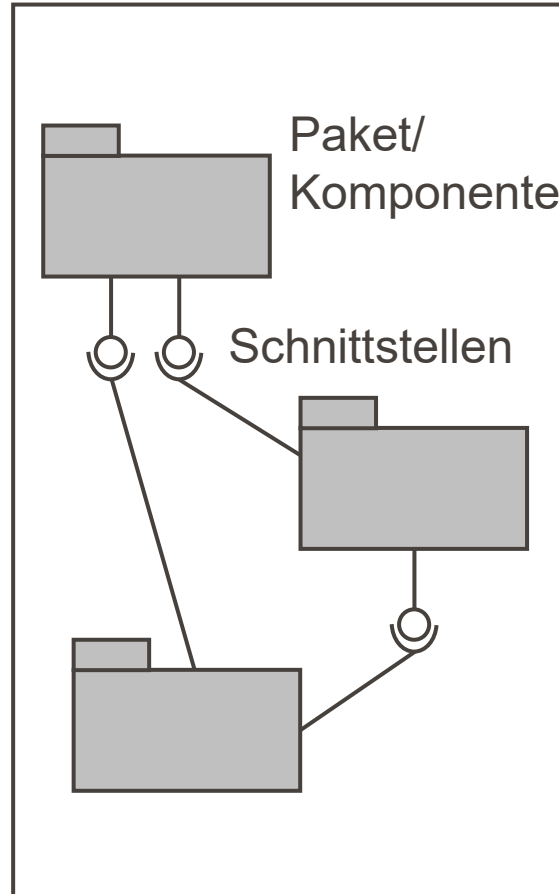
# Testansätze zusammengefasst

## White-Box-Test



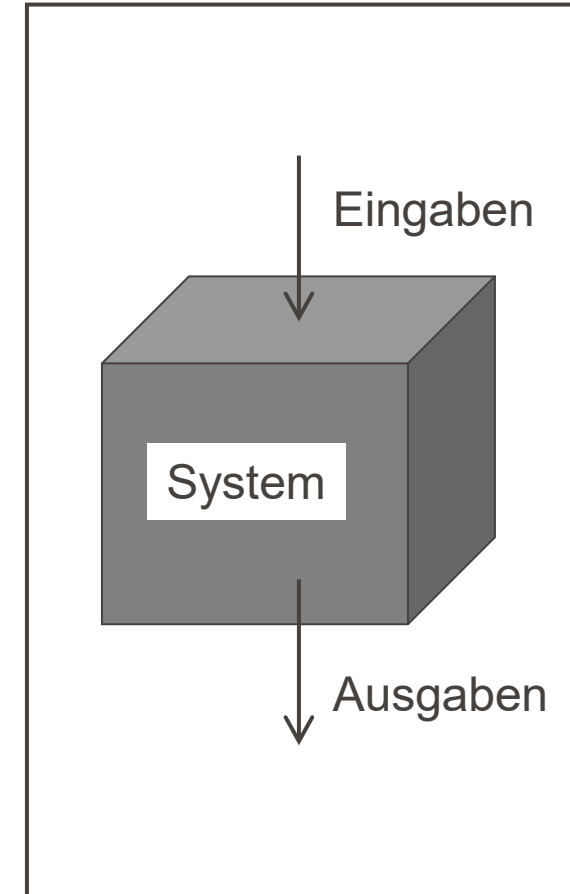
Methoden-/Klassentest

## Gray-Box-Test



Integrationstest

## Black-Box-Test

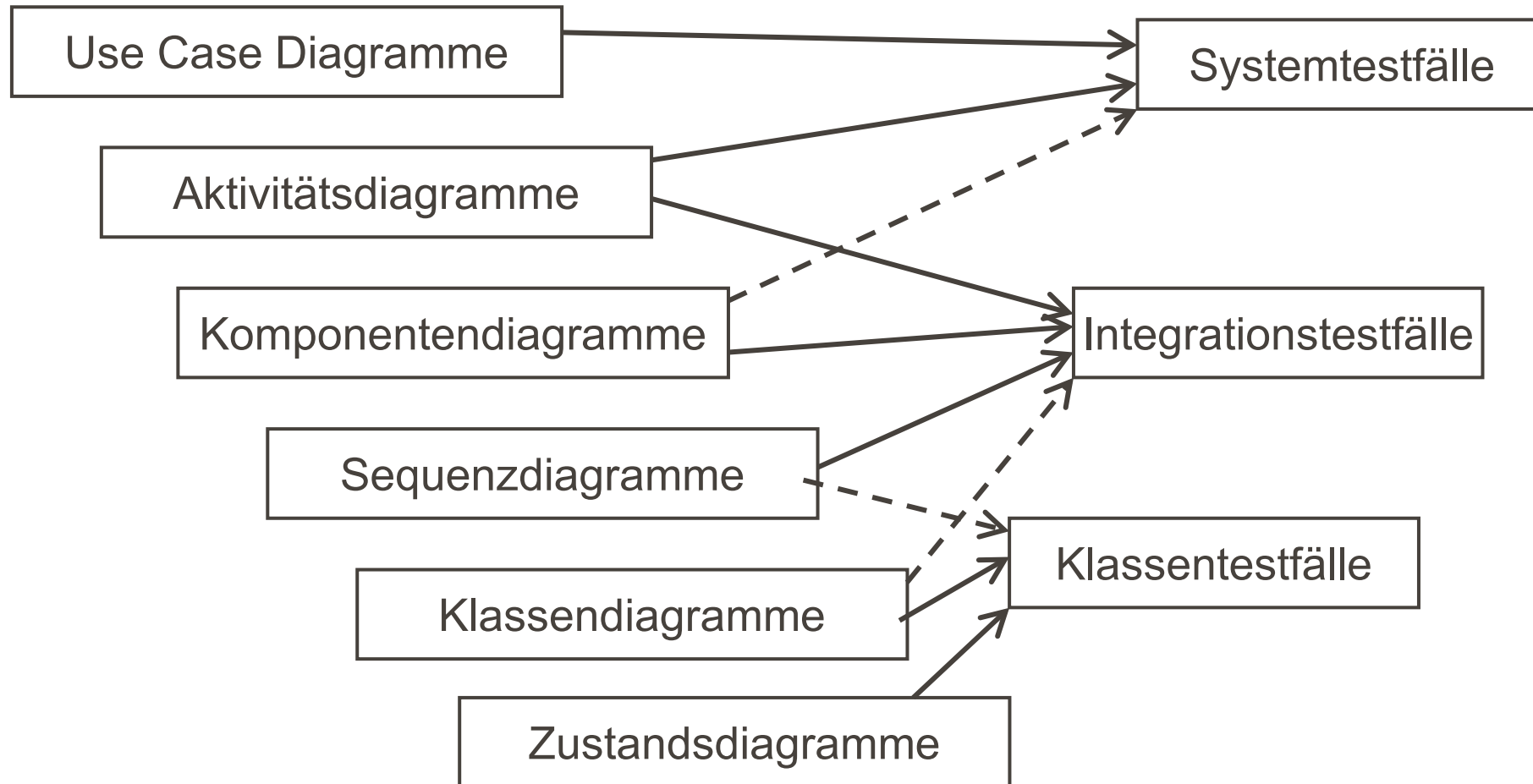


Systemtest

# Testfälle und die UML

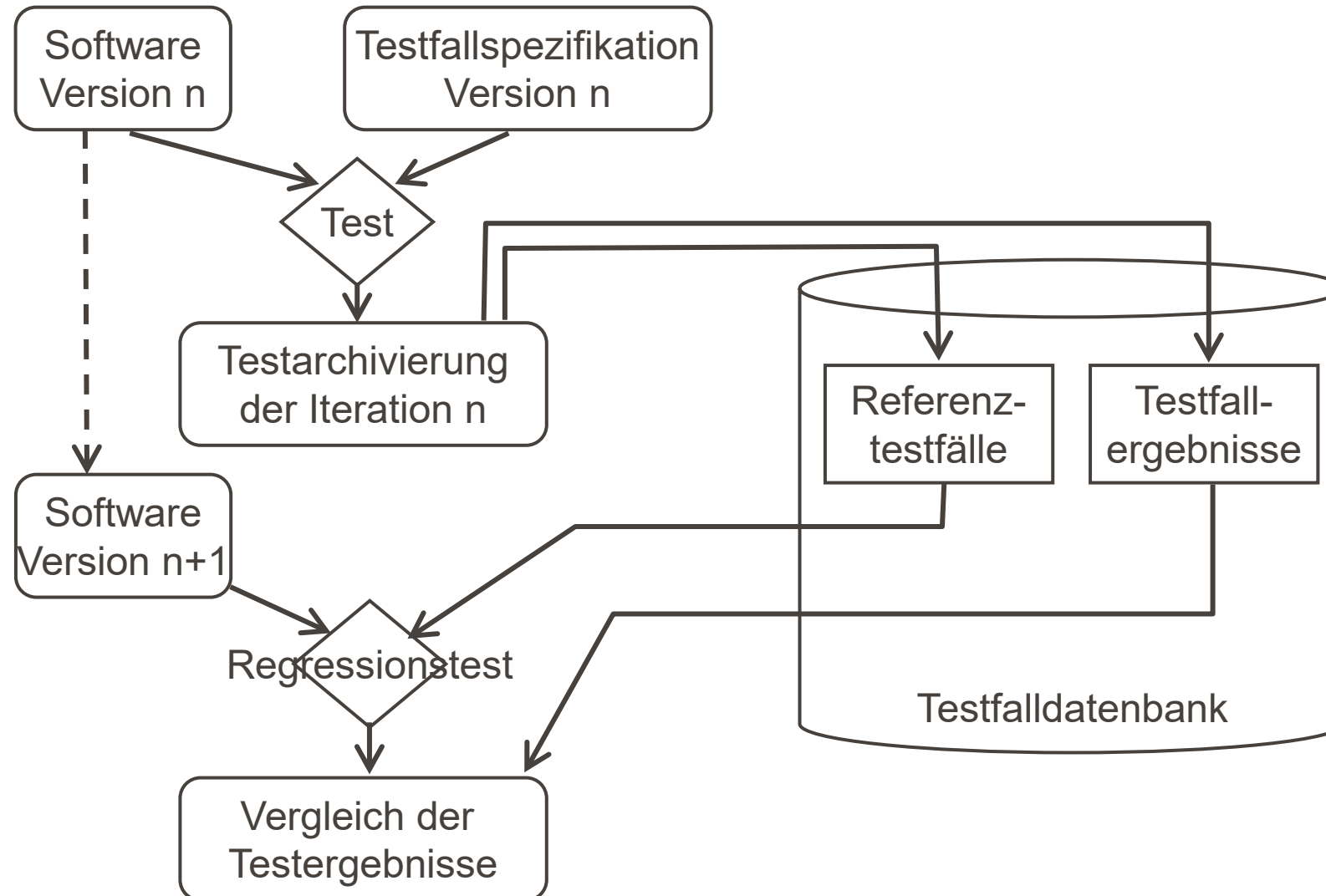
## Entwicklung in der UML

## Testen

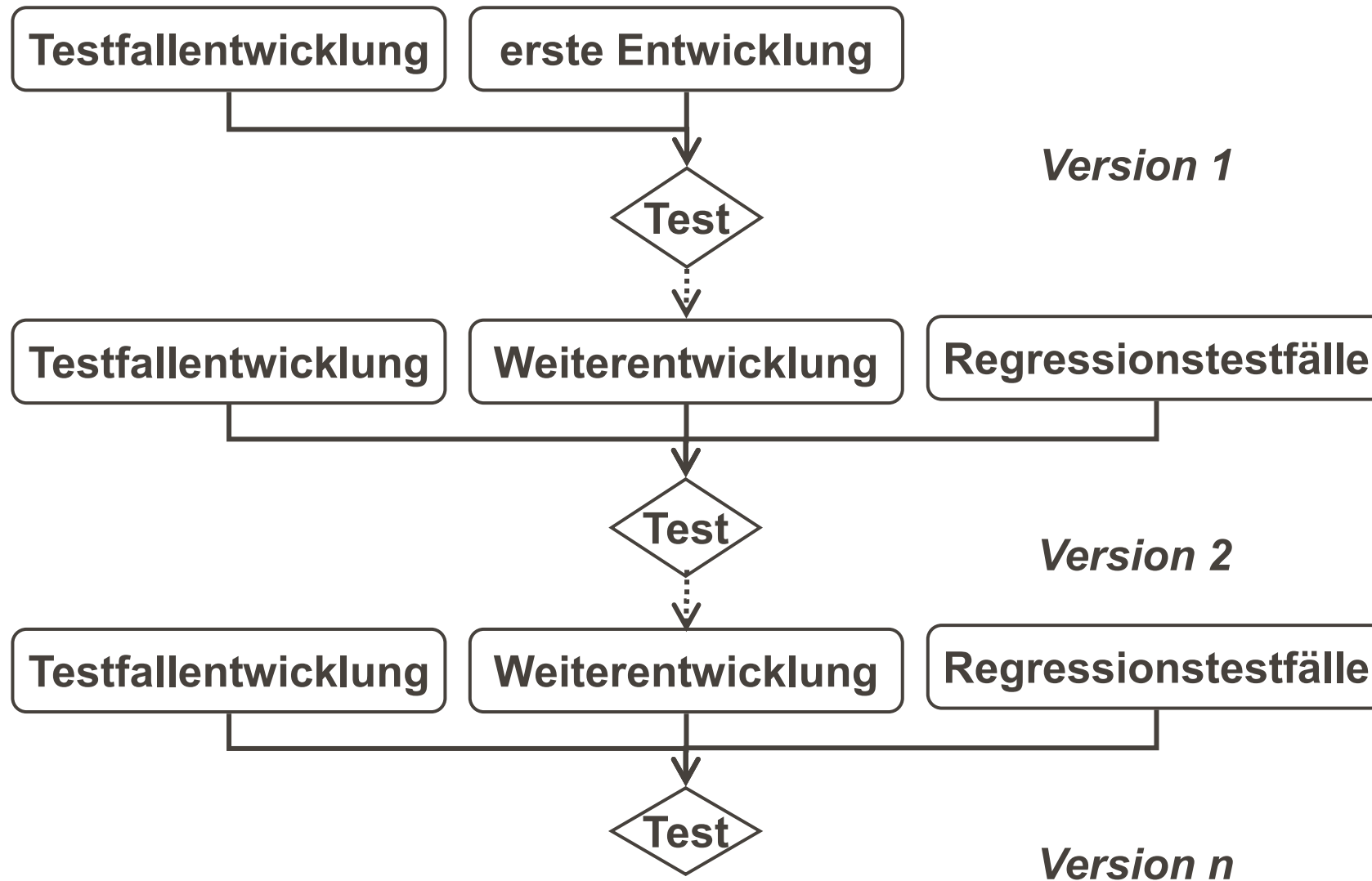


- Oftmals muss man recht ähnliche Daten zum Testen nutzen, dabei bietet es sich an, diese Testdaten in einem File zu speichern und die Daten schrittweise auszulesen
- Generell können professionelle Testwerkzeuge meist Daten aus Datenbanken, kommaseparierten Listen, Excel-Files, ... lesen
- TestNG als JUnit-Erweiterung erlaubt Testdatengenerierung für Unit-Tests

# Prinzip des Regressionstests



# Regressionstests im Entwicklungszyklus



- Änderungen an bereits freigegebenen Modulen sind notwendig
  - Gibt es Auswirkungen auf die alten Testergebnisse?
  - Wenn ja, welche?
  - Wiederholbarkeit der Tests
  - Wiederherstellung der Testdaten
- 
- Der Testprozess muss automatisierbar sein
  - Testfälle müssen gruppiert werden können, damit man sie wegen der untersuchten Funktionalität (oder auch Testdauer) gezielt einsetzen kann

- Der Test ist geteilt in Änderungstest (White-Box) und Regressionstest (Black-Box)
- Änderungstest vom Entwickler, er schreibt die Testfälle fort.
- Regressionstest von unabhängiger Testgruppe mit den alten plus neuen Testfällen durchgeführt
- Testgruppe ist für Pflege und Fortschreibung der Systemtestfälle verantwortlich

## Geforderte Performance

- Durchsatz bzw. Transaktionsrate
- Antwortzeiten

## Skalierbarkeit

- Anzahl Endbenutzer
- Datenvolumen
- Geografische Verteilung

## Zugriffskonflikte konkurrierender Benutzer

## Entspricht dem Zeitraum nach der Inbetriebnahme

## Simulation von

- Anzahl Endbenutzer,
- Transaktionsrate ,
- ...
- Über einen signifikanten Zeitraum (mehrere Stunden)



- wie oft wird jede Methode aufgerufen (oder Quellcodezeile durchlaufen)
- welche Methode ruft wie oft welche andere Methode (descendants) auf oder von welchen Methoden (callers) wird eine Methode wie oft gerufen
- wie viel Prozent der Gesamtlaufzeit wird mit Ausführung einer bestimmten Methode verbracht (ggf. aufgeteilt nach callers und descendants)

## Nutzen der ermittelten Daten

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

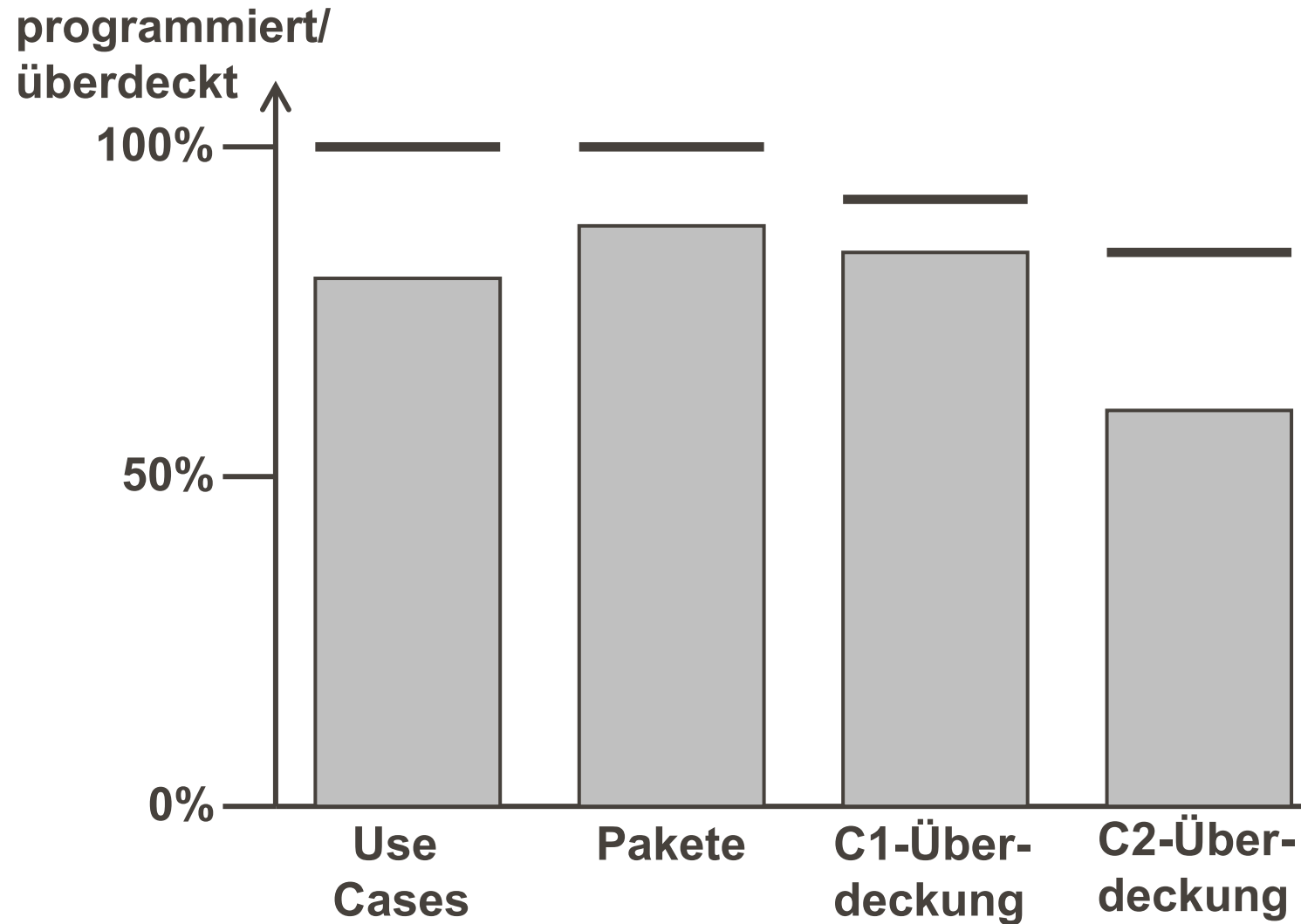
- welche Operationen fordern wie viel Speicherplatz an (geben ihn frei)
- wo wird Freigabe von Speicherplatz vermutlich bzw. bestimmt vergessen (memory leak = Speicherloch):
- bestimmt vergessen: Objekt lebt noch, kann aber nicht mehr erreicht werden (Garbage Collector von Java würde es entsorgen)
-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

- Hinweis: Java unterstützt das „Profiling“, hierzu stehen explizite Compiler- und Ausführungsoptionen zur Verfügung
- kommerzielle Produkte, z. B. „IBM/Rational Purify und Quantify
- freie Werkzeuge:
  - Test and Performance Tools Platform (TPTP) für Eclipse
  - Netbeans hat Performanceanalyse eingebaut
  - JMeter: <http://jakarta.apache.org/jmeter/> (besonders interessant für Lasttests von Web-Servern und Datenbankzugriffen)

- Grundsätzlich sind Oberflächen gewöhnliche SW-Module und können wie diese getestet werden.
- Für xUnit-Werkzeuge ist der Zugriff auf Oberflächen teilweise schwierig (xUnit muss auf Oberflächenkomponenten zugreifen und diese bedienen können, JButton in Java hat z.B. Methode `doClick()`)
- Für Oberflächen gibt es deshalb Capture & Replay-Werkzeuge
- Grundidee: Das Werkzeug zeichnet alle Mausbewegungen und Tastatureingaben auf, die können dann zur Testwiederholung erneut abgespielt werden
- Typisch ist, dass der Nutzer die aufgezeichneten Skripte modifizieren kann (z. B. Test von berechneten Daten)
- Tools können teilweise auch Oberfläche lesen (Frage ob Texte richtig ausgegeben), Snapshots vergleichen
- professionelle Beispiele: Winrunner von HP (früher Mercury), VisualTest von IBM-Rational
- zum Ausprobieren: Abbot (<http://abbot.sourceforge.net/>)

- Die bisherigen Überprüfungsverfahren sind recht aufwändig, es besteht der Wunsch, schneller zu Qualitätsaussagen zu kommen
- Der Ansatz ist die Einführung eines Maßsystems, aus dem System werden Zahlenwerte generiert, deren Werte ein Indiz für die Qualität eines Produktes geben
- Werden diese Maße automatisch berechnet, kann man Qualitätsforderungen stellen, dass bestimmte Maßzahlen in bestimmten Bereichen liegen
- Wichtig ist, dass man weiß, dass nur Indikatoren betrachtet werden, d.h. gewonnene Aussagen müssen nachgeprüft werden
- Ähnliche Ansätze in der Projektverfolgung und Analyse der Firmengesamtlage (-> Balanced Scorecard)  
-> siehe Qualitätsmanagement

# Metriken zur Ermittlung des Projektstands



- Lines of Code pro Methode (LOC)  
Nach der Grundregel des guten Designs sollte die maximale Zahl unter 20 liegen
- Methoden pro Klasse  
Die Zahl sollte zwischen 3 und 15 liegen
- Parameter pro Methode  
Die Zahl sollte unter 6 liegen
- Exemplarvariablen pro Klasse  
Die Zahl sollte unter 10 liegen [Entitäten ?]
- Abhängigkeiten zwischen Klassen  
Keine Klasse sollte von mehr als vier Klassen abhängen
- andere Maßzahlen, wie die Anzahl von Klassenvariablen und Klassenmethoden können auch als Indikatoren eingesetzt werden

1. Man konstruiere die Kontrollflussgraphen
2. Man messe die strukturellen Komplexität

Die zyklomatische Zahl  $z(G)$  eines Kontrollflussgraphen  $G$  ist:

$$z(G) = e - n + 2 \text{ mit}$$

- $e$  = Anzahl der Kanten des Kontrollflussgraphen
- $n$  = Anzahl der Knoten


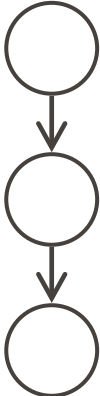
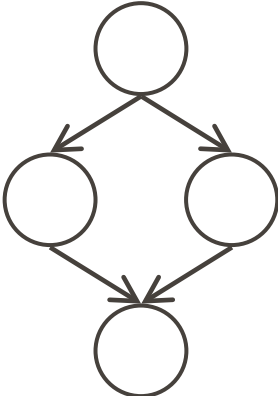
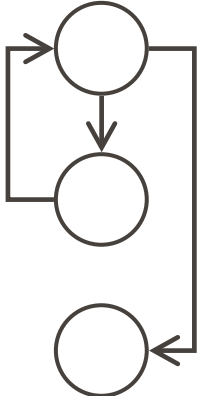
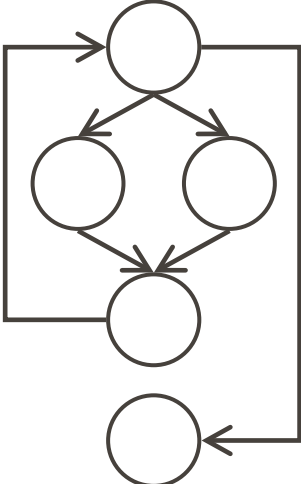
Zyklomatische Komplexität gibt Obergrenze für die Testfallanzahl für den Zweigüberdeckungstest an

In der Literatur wird 10 oft als maximal vertretbarer Wert genommen (für OO-Programme geringer, z. B. 5)

für Java:  $\#if + \#do + \#while + \#switch-cases + 1$



# Beispiele für die zyklomatische Zahl

					
Anzahl Kanten	0	2	4	3	6
Anzahl Knoten	1	3	4	3	5
McCabe Zahl	1	1	2	2	3

- Komplexität von Verzweigungen berücksichtigen
- gezählt werden alle Booleschen Bedingungen in `if(<Bedingung>)` und `while(<Bedingung>)`:  
anzahlBedingung
- gezählt werden alle Vorkommen von atomaren Prädikaten: anzahlAtome  
z. B.: `( a || x>3) && y<4` dann anzahlAtome=3
- erweitere McCabe-Zahl  
$$ez(G) = z(G) + anZahlAtome - anzahlBedingung$$
- wenn nur atomare Bedingungen, z. B. `if( x>4)`, dann gilt  $ez(G) = z(G)$

# LACK OF COHESION IN METHODS (LCOM\*)

- nutzt(a) die Zahl der Methoden, die eine Exemplarvariable a der untersuchten Klasse nutzen
- sei avgNutz die Durchschnitt aller Werte für alle Exemplarvariablen
- sei m die Anzahl aller Methoden der untersuchten Klasse
- $LCOM^* = (avgNutz - m) / (1 - m)$
- Ist der Wert nahe Null, handelt es sich um eine eng zusammenhängende Klasse
- Ist der Wert nahe 1, ist die Klasse schwach zusammenhängend, man sollte über eine Aufspaltung nachdenken
  
- Hinweis: Es muss vorher festgelegt werden, ob Klassenvariablen und Klassenmethoden berücksichtigt werden sollen
- Was passiert, wenn man konsequent exzessiv OO macht und auch in den Exemplarmethoden get() und set() Methoden nutzt? Wie ist LCOM\* dann rettbar?

# LCOM\*-Beispiel

```
public class LCOMSpielerei {  
    private int a;  
    private int b;  
    private int c;  
    public void mach1(int x) {  
        a=a+x;  
    }  
    public void mach2(int x) {  
        a=a+x;  
        b=b-x;  
    }  
    public void mach3(int x) {  
        a=a+x;  
        b=b-x;  
        c=c+x;  
    }  
}
```

**nutzt(a)=3  
nutzt(b)=2  
nutzt(c)=1  
avgNutzt=6/3=2  
LCOM\*= (2-3) /(1-3)  
= -1/-2= 0.5**

**für Eclipse gibt es Plugin Metrics**  
<http://sourceforge.net/projects/metrics/>  
**berechnet u. A. erweiterte  
McCabe-Zahl und LCOM\***