

# Vorlesung "Software-Engineering"

---

Prof. Ralf Möller, TUHH, Arbeitsbereich STS

## ■ Vorige Vorlesungen

### ■ Definition

- Pflichtenheft (requirements specification document)
- Detaillierte Anforderungsanalyse (detailed requirements engineering)

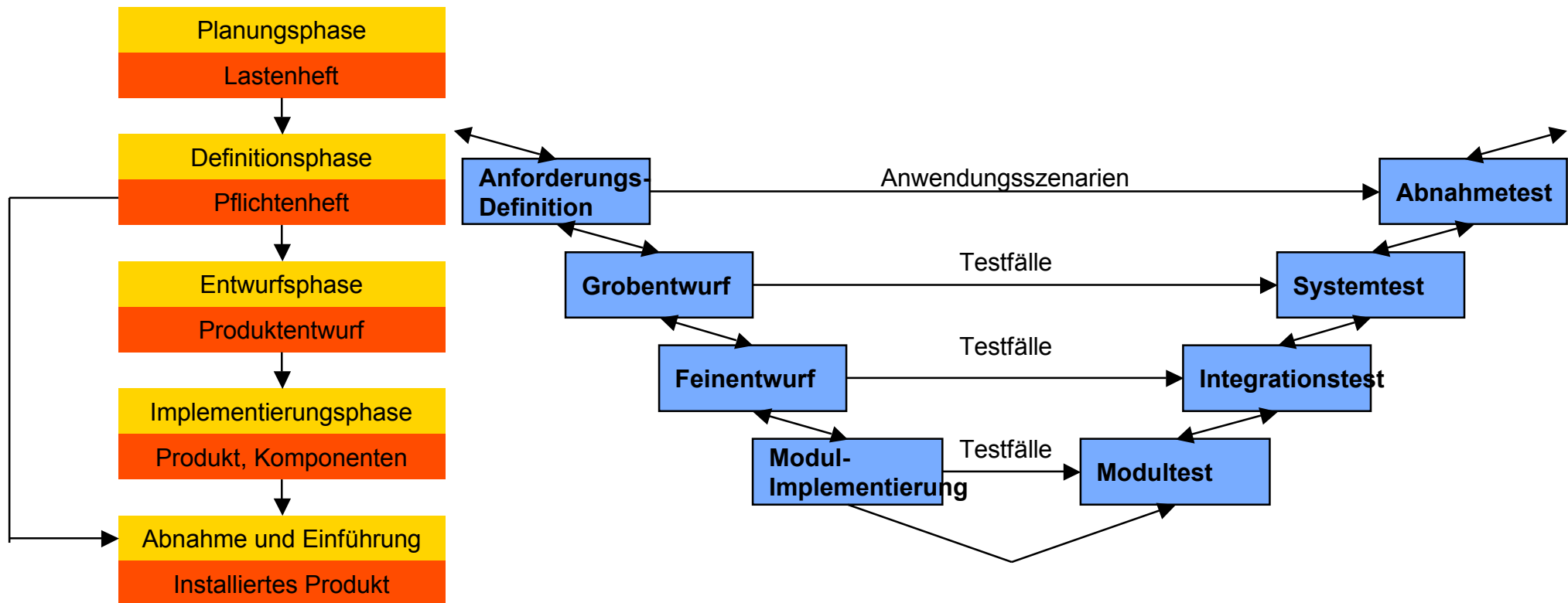
## ■ Heute: Maßnahmen zur Qualitätssteigerung

### ■ Metriken

### ■ Testen

### ■ Organisatorische Maßnahmen

# Einordnung



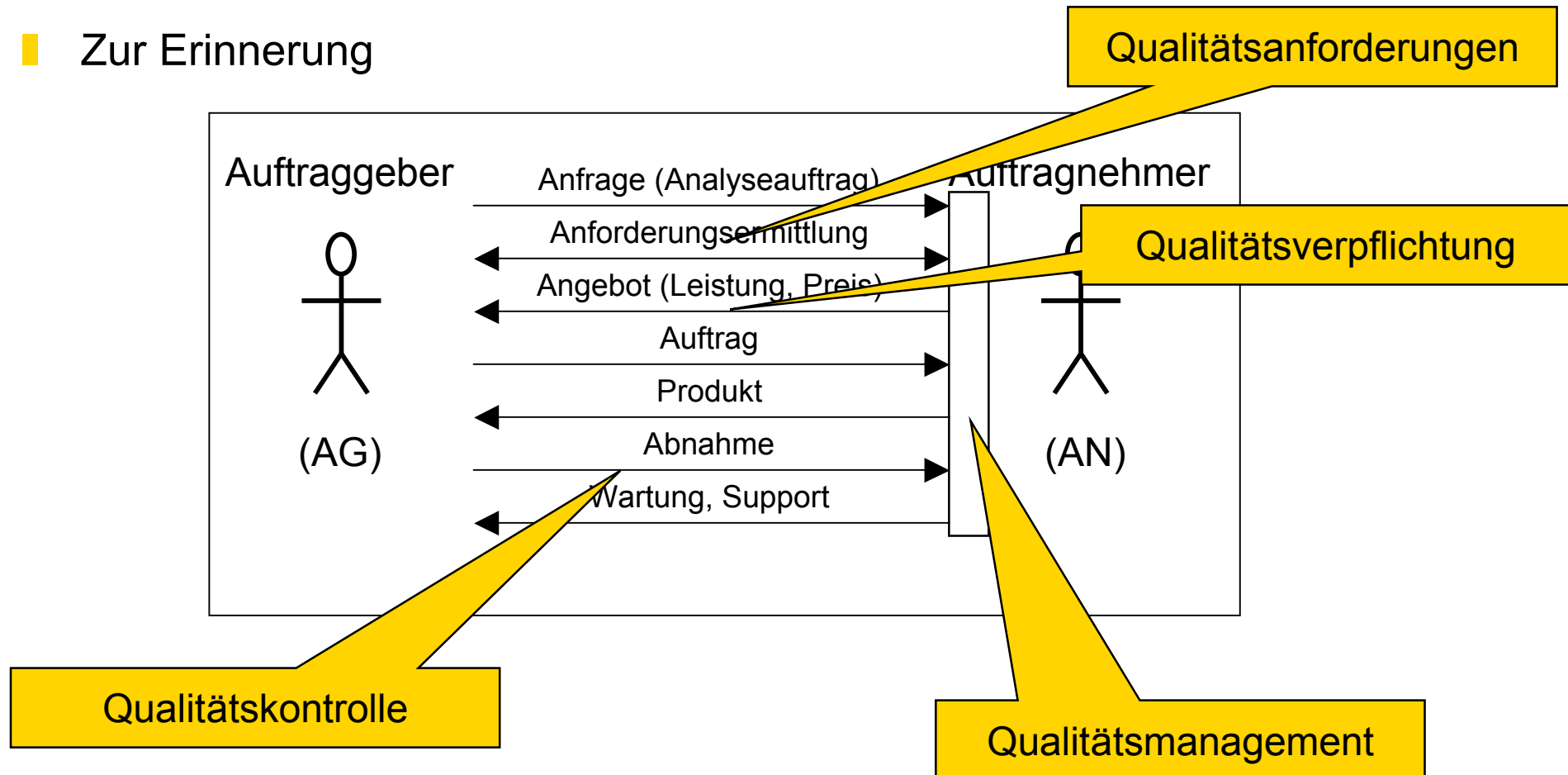
# Gegenstand der Vorlesung

---

- Unter dem Namen SW-Engineering werden Methoden und Prinzipien zur Lösung von „böartigen Problemen“ (Rittel 73) diskutiert
- Probleme sind „böartig“,
  - wenn sie in so viele Einzelteile verstrickt sind, daß es keine endgültige Spezifikation des Problems gibt.
  - wenn sich das wahre Gesicht erst bei der Entwicklung der Lösung zeit
  - wenn sich Anforderungen erst bei bzw. nach der Implementierung ergeben

# Qualität bei Individualsoftware

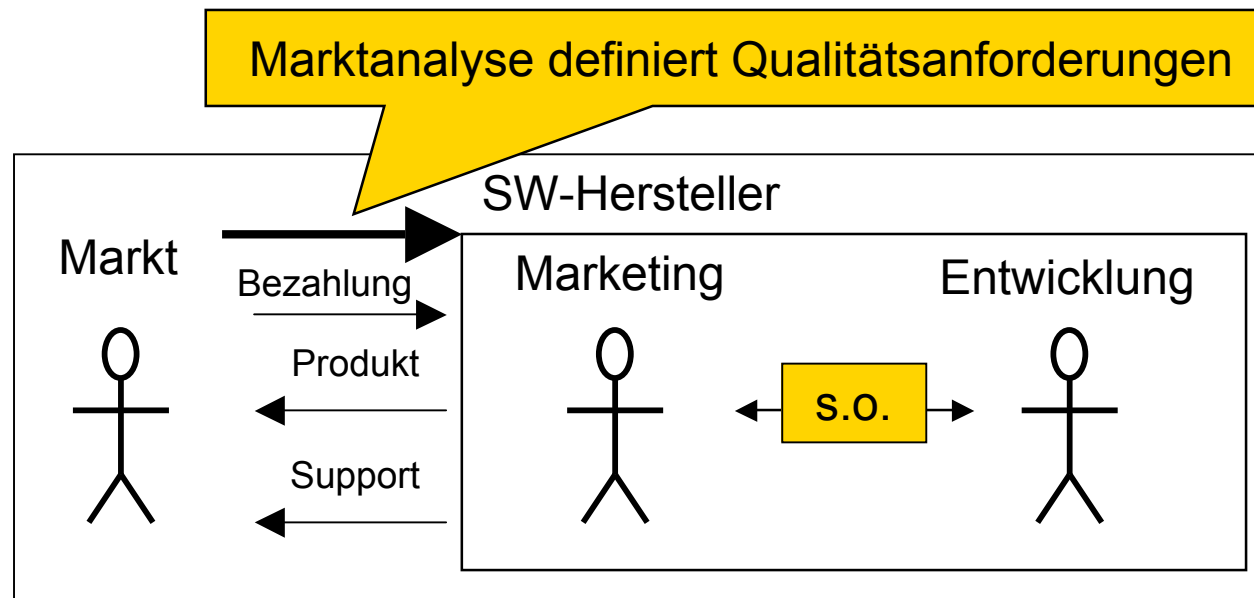
- Zur Erinnerung



- Ziel des **Qualitätsmanagements** beim Auftragnehmer ist die Erreichung von Qualitätsanforderungen in der gesamten Lebensdauer der Software

# Qualität bei Standardsoftware

## ■ Zur Erinnerung: Kapitel 2



# Qualitätsmanagement

---

- Das **Qualitätsmanagement** (QM) umfaßt alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortungen festlegen sowie diese durch Mittel wie Qualitätsplanung, Qualitätslenkung, Qualitätssicherung und Qualitätsverbesserung im Rahmen des Qualitätsmanagements verwirklichen (DIN ISO 8402).
- Man unterscheidet **produktorientiertes QM** und **prozeßorientiertes QM**, (ISO9000) wobei sich letzteres ausschließlich auf den Softwareentwicklungsprozeß selbst bezieht (Meilensteine, Qualitätssicherungsmaßnahmen, Rollendefinition, ...).
- **Analytische vs. konstruktive** Qualitätssicherung
- **Qualitätssicherung** (QS) sind alle geplanten und systematischen Tätigkeiten, die innerhalb des Qualitätsmanagement-Systems verwirklicht sind, um angemessenes Vertrauen zu schaffen, daß ein Produkt die Qualitätsforderung erfüllen wird (DIN ISO 8402).
- Die Qualitätssicherung umfaßt die **Validierung** und die **Verifikation**. Beide Tätigkeiten werden in der Praxis durch **Tests** unterstützt.

## Testing?

**Don't worry,** not *this* kind of testing...

---



# Qualitätssicherung der Korrektheit

---

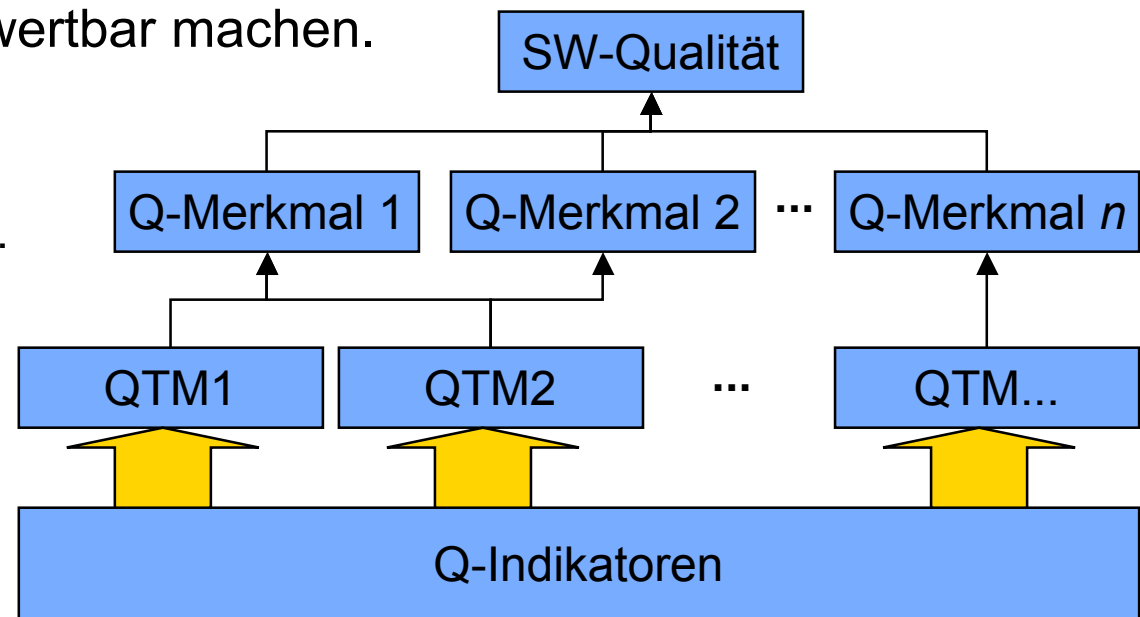
- Ein **Test** prüft die Qualität von Software, indem für einige, möglichst gut ausgewählte Testdaten die Übereinstimmung zwischen Anforderung und System untersucht wird
- Eine **Verifikation** prüft die Qualität von Software durch einen mathematischen Beweis, der die in der Spezifikation geforderten Anforderungen ausgehend von einem mathematisch exakten Modell des Systems beweist
- Ein **analysierendes Verfahren** stellt Maßzahlen oder Eigenschaften des Systems dar, die empirisch mit der Korrektheit von Software verknüpft sind.
  - Anzahl der Operatoren, Operanden, Länge der Implementierung, Anzahl der Knoten im Kontrollflußgraph, Anzahl uninitialisierter Variablen, Anzahl der Mehrfachzuweisungen ohne Lesezugriff, ...
  - Tiefe des Vererbungsbaums, Kopplung zwischen Klassen, Anzahl der Kinder pro Klasse, ...
  - Anzahl der Kommentare pro Programmeinheit, ...
- Eine **Validierung** überprüft, ob ein bestimmtes Prädikat, das bezüglich einer Software gelten soll, auch tatsächlich gilt

# Softwarequalität: Das FCM-Modell

## ■ Factors-Criteria-Metrics-Modell:

- **Quality Factors** beschreiben benutzerorientierte Qualitätsmerkmale (QMs) der Software.
- **Quality Criteria** sind Teilmerkmale (QTM), die Qualitätsmerkmale verfeinern.
- **Quality Metrics** sind Qualitätsindikatoren / Qualitätsmetriken, die Teilmerkmale meß- und bewertbar machen.

- Ein Teilmerkmal kann Einfluß auf mehrere Qualitätsmerkmale haben.
- Ein **Software-Qualitätsmaß** ist eine Metrik und eine Methode zur Berechnung der Softwarequalität aus den Indikatoren.

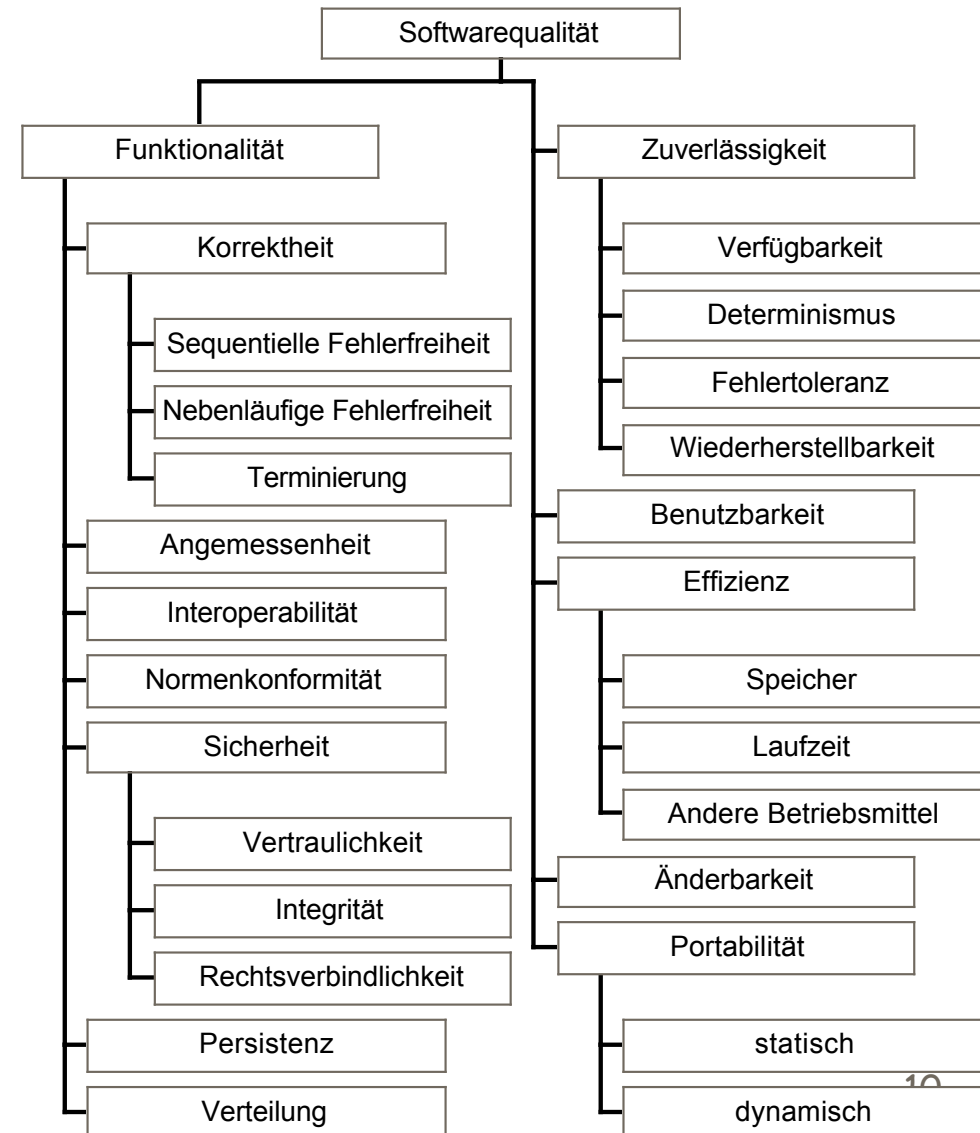


# Softwarequalitätsmerkmale nach DIN und ISO

- **DIN ISO 9126** definiert sechs Qualitätsmerkmale zur Beschreibung der Softwarequalität, die weitgehend disjunkte Teilmerkmale besitzen:

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Portabilität

- Funktionale Anforderungen und nicht-funktionale Anforderungen meist vermischt



# Metriken und ihre Eigenschaften (1)

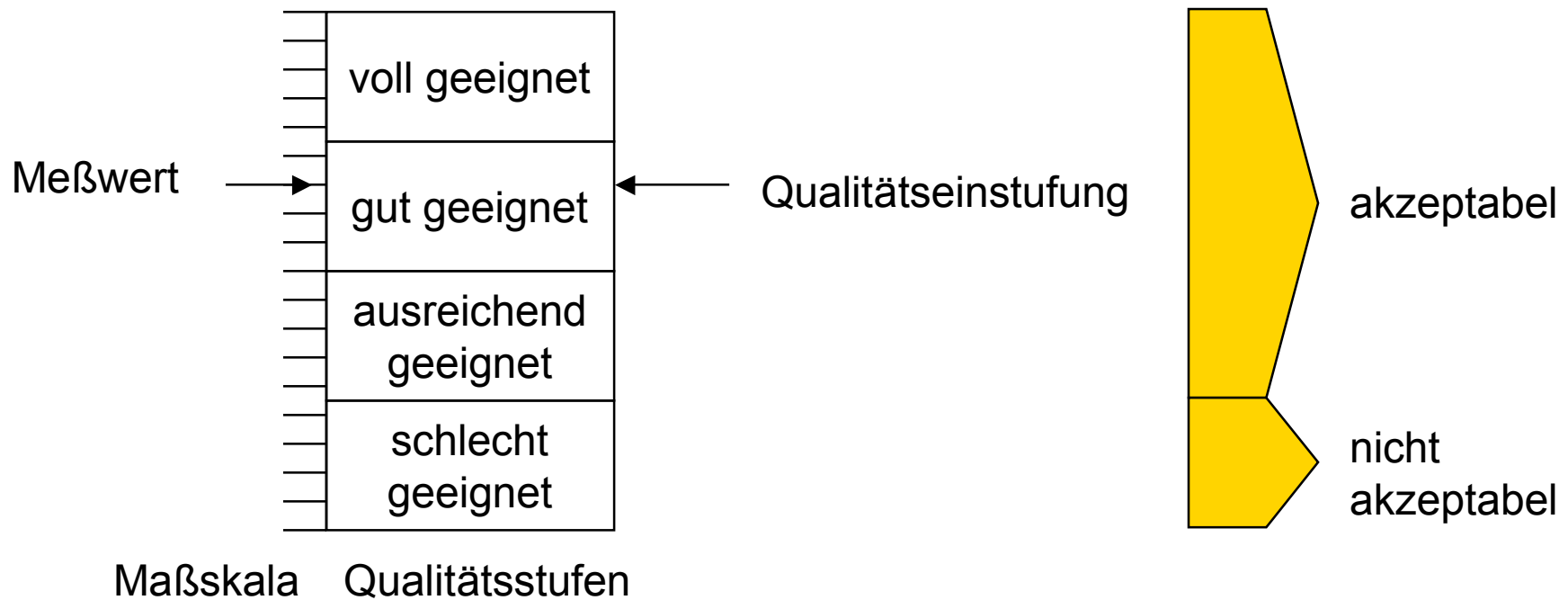
- Eine **Metrik** (ein Maßsystem) ist eine Algebra, die meßbare Eigenschaften durch **Zahlen** (Maßzahlen) ausdrückt.
- Metriken lassen sich anhand der für die Maßzahlen gültigen algebraischen Regeln (**Skalen**) klassifizieren:
  - Eine **Nominalskala** definiert eine endliche, ungeordnete Menge von diskreten Merkmalsausprägungen (bijektive Abbildung).  
*Beispiel:* Grundfarben.  
*Operationen:* Gleichheitstest
  - Eine **Ordinalskala** ist eine Nominalskala erweitert um eine vollständige Ordnung, und ist daher isomorph zu einer monoton steigenden Folge natürlicher Zahlen (je streng monotone Funktion ist geeignet).  
*Beispiele:* Windstärken, Hubraumklassen.  
*Zusätzliche Operationen:* Ordnungstest, Median, Rang, Rangkorrelationskoeffizient.
- Eine **Intervallskala** ist eine Ordinalskala erweitert um ein Abstandsmaß (Verhältnis von Intervallen:  $g(x) = a \cdot x + b$ ).  
*Beispiele:* Schulnoten.  
*Zusätzliche Operationen:* Arithmetisches Mittel und Standardabweichung.

Minimalanforderung an SW-Qualitätsmaß

# Metriken und ihre Eigenschaften (2)

- Eine **Rationalskala** (Verhältnisskala) verwendet als Maßzahlen reelle Zahlen und eine Maßeinheit. Dabei kann ein absoluter oder natürlicher Nullpunkt vorhanden sein. (Verhältnis von Meßwerten:  $g(x) = a \cdot x$ )  
*Beispiele:* Preise, Längen, Volumina, Zeiten.  
*Zusätzliche Operationen:* Quotientenbildung.
- Eine **Absolutskala** verwendet als Maßzahlen reelle Zahlen. Eine Skalenverschiebung ist nicht möglich ( $g(x) = x$ ).  
*Beispiele:* Häufigkeiten, Wahrscheinlichkeiten.
- Formal ist eine Metrik für eine Menge  $A$  eine **Distanzfunktion**  $d: A \times A \rightarrow \mathbb{R}$ , wobei für alle  $a, b \in A$  gilt:
  - $d(a, b) \geq 0$ ,  $d(a, a) = 0$
  - $d(a, b) = d(b, a)$
  - $d(a, b) \leq d(a, c) + d(c, b)$  für alle  $c \in A$  (Dreiecksungleichung)
  - $d(a, b) = 0 \Rightarrow a = b$

# Beispiel: Qualitätsstufen nach ISO 9126

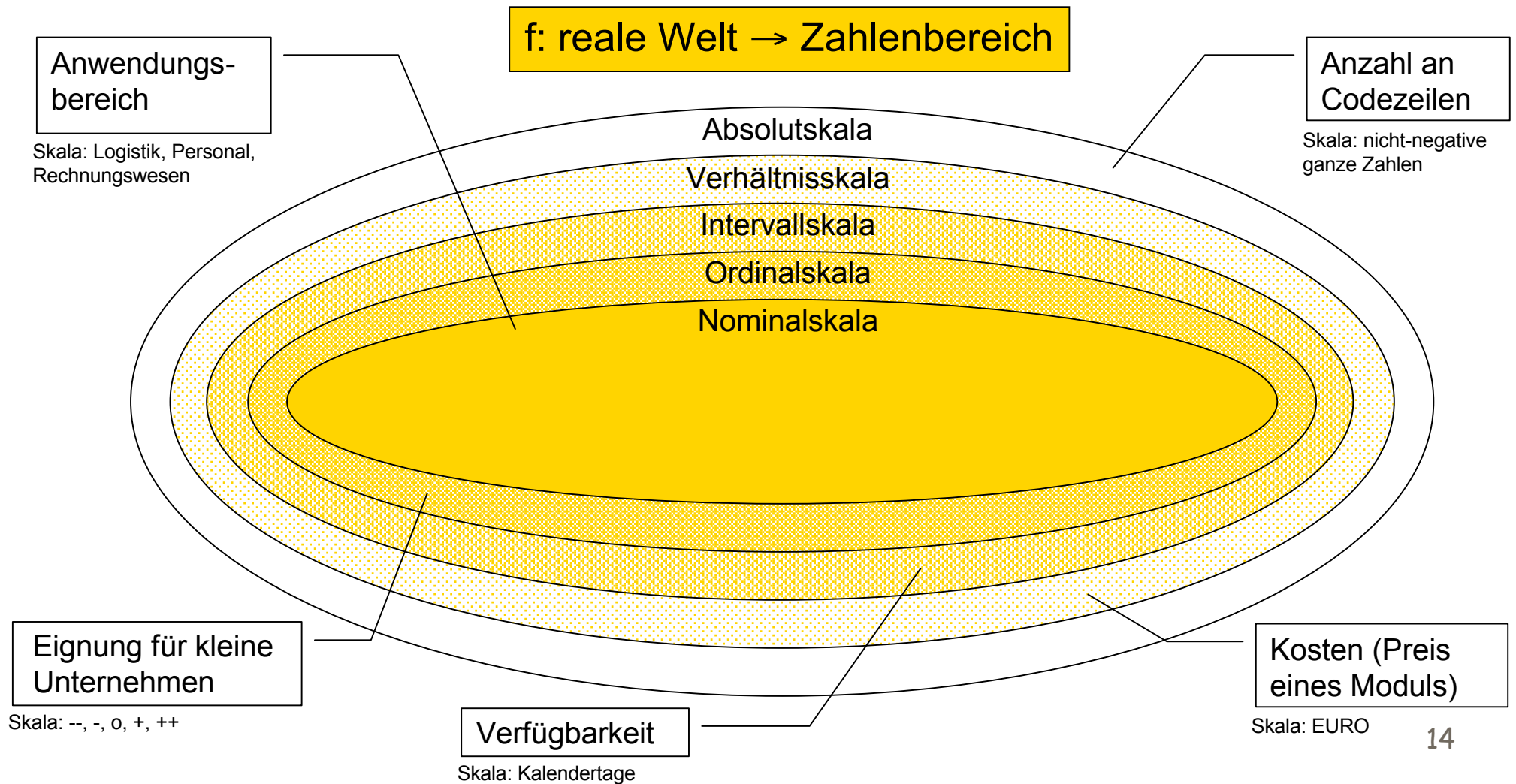


# Maßtheoretische Grundlagen

Nach: Clemens Holzmann  
Seminar Programmierstil, WS2002/03  
Institut für Systemsoftware  
Univ. Linz

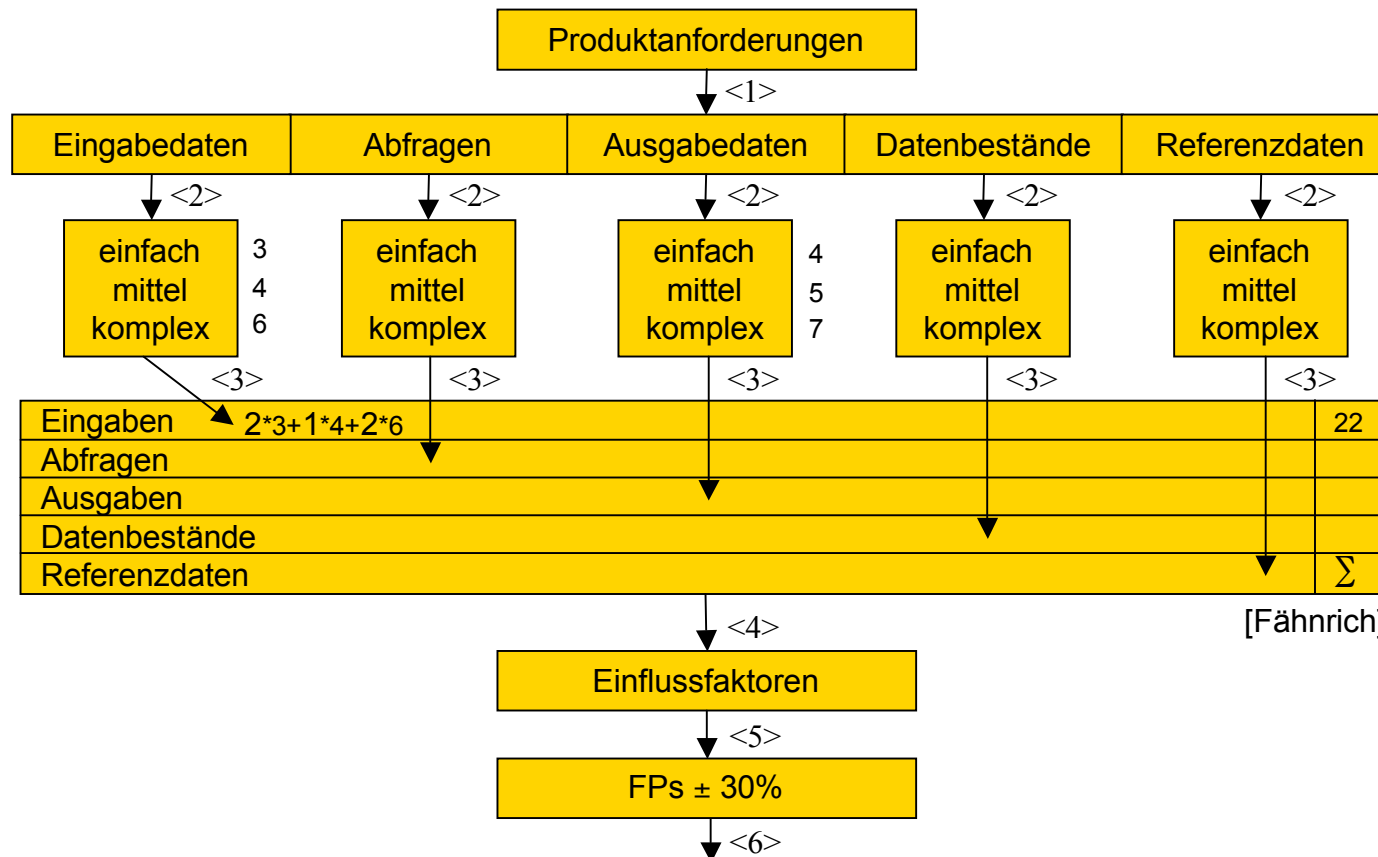
Skalenhierarchie am Beispiel eines Softwaremoduls

f: reale Welt → Zahlenbereich



# Beispiele für Metriken (1)

## Function-Points-Metrik



<1> Jede Anforderung kategorisieren

<2> Jede Anforderung klassifizieren

<3> Anforderungen ins Berechnungsformular eintragen

<4> Einflussfaktoren bewerten

<5> Bewertete FPs berechnen

<6> Aufwand ablesen

<7> Tabelle aktualisieren

# Beispiele für Metriken (2)

## ■ LOC (lines of code)

- ☒ ☺ Starke Korrelation mit anderen Maßen
- ☒ ☹ Komplexität von Anweisungen und Ablaufstrukturen unberücksichtigt, abhängig von Programmierstil/ -sprache

## ■ Halstead

$$\text{Umfang } V = (N_1 + N_2) * \text{Id}(n_1 + n_2)$$

$n_1, n_2$  Anzahl unterschiedl. Operatoren, Operanden

$N_1, N_2$  Gesamtzahl verwendeter Operatoren, Operanden

Operator kennzeichnet Aktionen (+, \*, While, For, ...)

Operand kennzeichnet Daten (Variablen, Konstanten, ...)

- ☒ ☺ Komplizierte Ausdrücke sowie viele verschiedene Variablen berücksichtigt
- ☒ ☹ Schwer messbar, Ablaufstrukturen unberücksichtigt

# Beispiele für Metriken (3)

## ■ McCabe

- Programm wird als gerichteter Graph dargestellt
- ☒ ☺ Einfach zu berechnen
- ☒ ☹ Komplexität von Anweisungen unberücksichtigt

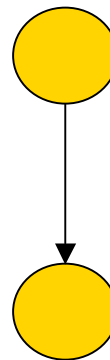
Allgemein

$V(g) = e - n + 2p$   
e ... Anzahl der Kanten  
n ... Anzahl der Knoten  
p ... Anz. verbundener Komponenten

Bei nur einem Ein- und Ausgang

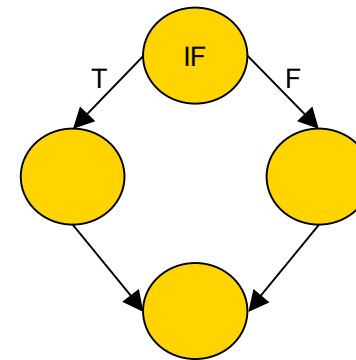
$V(g) = 1 + \text{Anzahl der Binärverzweigungen}$

Sequenz



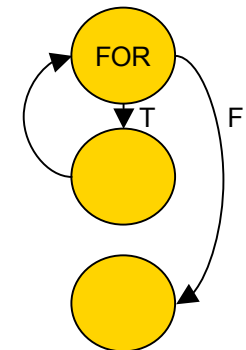
$$V(g) = 1 - 2 + 2 = 1$$
$$V(g) = 1 + 0 = 1$$

Auswahl



$$V(g) = 4 - 4 + 2 = 2$$
$$V(g) = 1 + 1 = 2$$

Abweisende Schleife



$$V(g) = 3 - 3 + 2 = 2$$
$$V(g) = 1 + 1 = 2$$

# Beispiele für Metriken (4)

## ■ Rechenberg

- ☒ 😊 Detailliert, betrachtet viele verschiedene Aspekte
- ☒ 😞 Schwer zu berechnen, nicht intuitiv verständlich

$CC = SC + EC + DC$  ... Gesamtkomplexität

SC ... Summe der Anweisungskomplexitäten aller Anweisungen  
Wertzuweisung=1, Goto=5, Prozeduraufruf=1+Parameterzahl, While/For=3, ...

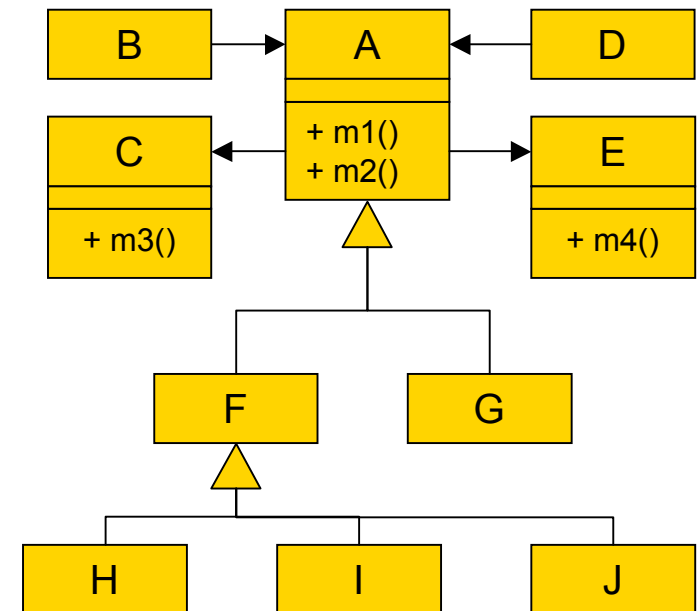
EC ... Summe der Ausdruckskomplexitäten aller Ausdrücke  
+/- =1, MOD=3, Indizierung=2, AND/OR=3, MUL/DIV=2, Dereferenzierung=2, ...

DC ... Summe der Datenkomplexitäten aller Bezeichner  
Lokale Namen=1, Formale Parameter=2, Globale Variablen=3

# Beispiele für Metriken (5)

## Objektorientierte Metriken

- CBO (coupling between objects)
  - | Anzahl der Klassen, mit denen eine Klasse gekoppelt ist
  - |  $CBO(A)=4$
- DIT (depth of inheritance tree)
  - | Maximaler Weg von der Wurzel bis zur betrachteten Klasse
  - |  $DIT(A)=0$ ,  $DIT(G)=1$ ,  $DIT(H)=2$
- NOC (number of children)
  - | Anzahl der direkten Unterklassen
  - |  $NOC(A)=2$ ,  $NOC(B)=0$ ,  $NOC(F)=3$
- RFC (response for a class)
  - | Anzahl der Methoden, die potentiell ausgeführt werden können, wenn Objekt auf eingehende Nachricht reagiert
  - |  $RFC(A)=4$ ,  $RFC(B)=0$ ,  $RFC(C)=1$



# Werkzeugunterstützung

---

## Werkzeugbeispiel JStyle 4.6

### ■ Code-Review

- Automatische Analyse des Sourcecodes
  - | Namenskonventionen, Designfehler, Redundanz, ...
- Skriptsprache zum Definieren eigener Regeln
- Beautifier mit umfangreichen Einstellmöglichkeiten

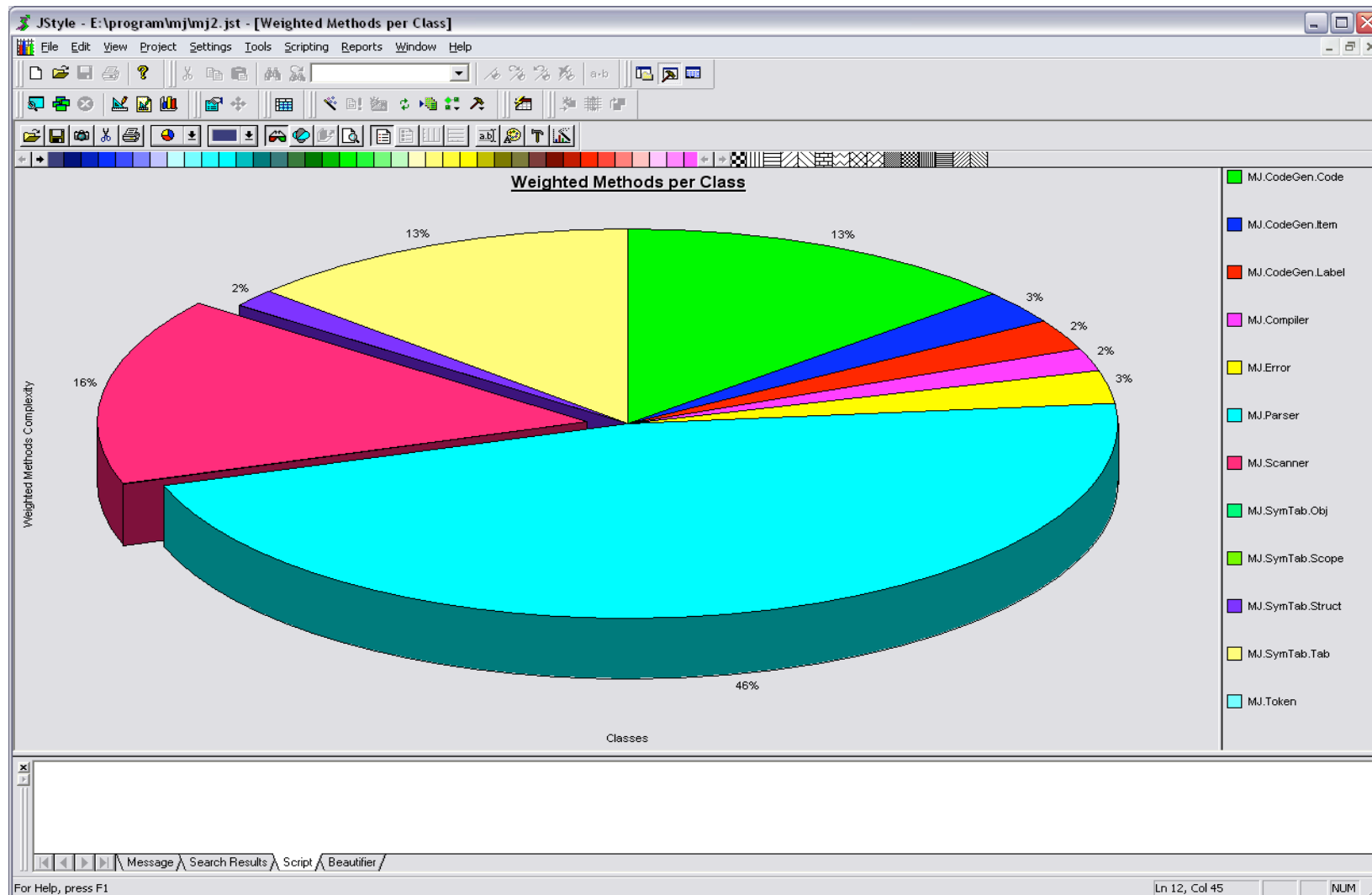
### ■ Metriken

- Berechnung einer Vielzahl von Sourcecode-Metriken
  - | Projekt-Level: Anzahl an Klassen, Kommentardichte, ...
  - | Klassen-Level: DIT, RFC, WMC, ...
  - | Methoden-Level: LOC, Halstead, McCabe, ...
- Diagramme zur Darstellung von Metriken
  - | Balken-, Torten- und Streudiagramm, Summenkurve, Box-Plot, ...

# Werkzeugunterstützung

Nach: Clemens Holzmann  
Seminar Programmierstil, WS2002/03  
Institut für Systemsoftware  
Univ. Linz

## Werkzeugbeispiel JStyle 4.6



# Testen

---

## Ziele:

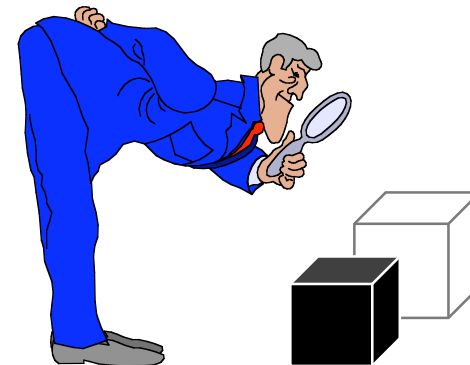
- Auffinden von Fehlern
- Nachweis der Funktionstüchtigkeit der Software
- Nachweis des korrekten Aufbaus der Software
- Nachweis der Leistungsfähigkeit der Software

## Methoden:

- Blackbox-Test
- Whitebox-Test

## Problem:

- Konstruktion von Testreihen und Testszenarios, die alle definierten Teile des getesteten Systems überdecken.



# Testprinzipien / Psychologie des Testens (1)

---

- Explizite Festlegung der erwarteten Werte / Resultate ist notwendig (vor dem eigentlichen Test!)
- Programme sollten nicht durch ihre Ersteller getestet werden
- Testergebnisse müssen gründlich überprüft werden
- Auch für ungültige, unerwartete, sinnlose Eingabedaten müssen Tests durchgeführt werden
- Tests müssen untersuchen
  - ob ein Programm etwas nicht tut, was es tun sollte
  - ob ein Programm etwas tut, was es nicht tun sollte

# Testprinzipien / Psychologie des Testens (2)

---

- "Wegwerftestfälle" sind unbedingt zu vermeiden
- Reproduzierbarkeit von Tests ist zu gewährleisten
- Tests müssen unter der Annahme geplant und durchgeführt werden, daß Fehler auftreten
- In Programmteilen, in denen bereits viele Fehler gefunden wurden, ist die Wahrscheinlichkeit weiterer Fehler besonders hoch
- Testen ist eine extrem kreative und intellektuell herausfordernde Aufgabe!

# Whitebox-Test

---

- Testen auf der Basis der Programmstruktur/-logik
- Möglichst viele Pfade durch das Programm, die sich durch den Kontrollfluß ergeben, durchlaufen
- Vollständige Whitebox-Tests nicht durchführbar
- Weitere Probleme der Whitebox-Tests:
  - Bezug zur Spezifikation häufig vernachlässigt
  - Programmlücken können nicht getestet werden
  - Fehler, die von speziellen Datenkonstellationen abhängig sind, werden u.U. nicht entdeckt

# Testverfahren: Statische Verfahren

---

- Software wird nicht auf dem Computer ausgeführt ("Human testing")
- Primärer Gegenstand der Tests ist der Programmtext (in Verbindung mit der Spezifikation)
- Maßnahmen
  - **Review/Code-Inspektion/Walkthrough**
  - Theoretische Verifikation und Komplexitätsanalysen

# Code-Inspektion / Walkthrough

---

- Durchführung in einem Team von ca. 4 Personen
- Neben dem Programmierer gehören ein Moderator und Programm- und Testspezialisten zum Team
- Durchführung
  - Programmierer erläutert die Programmlogik  
Anweisung für Anweisung
  - Teammitglieder stellen Fragen u. identifizieren Fehler

# Code-Inspektion / Walkthrough

---

- Programm wird mit Hilfe von Fehlerprüflisten analysiert
- Empfehlenswerte Dauer einer Sitzung ca. 2 Stunden (ca. 300 Anweisungen)
- Häufig eingesetztes Verfahren
- Erfolgreiches Verfahren: Studien ergaben Wirkungsgrade von 30% - 80% der Fehlerentdeckung
- Walkthrough: Durchspielen von vorbereiteten Testfällen durch die Mitglieder des Teams

# Testverfahren: Dynamische Verfahren

---

- Software wird mit Testdaten ausgeführt
- Ausführung in realer Umgebung
- Prinzip des Stichprobenverfahrens
- Notwendigkeit zur Auswahl von Testfällen und Testdaten

# Unterscheidung Testfälle - Testdaten

---

## ■ Testfall:

- eine aus der Spezifikation oder dem Programm abgeleitete Menge von Eingabedaten zusammen mit den zugehörigen erwarteten Ergebnissen

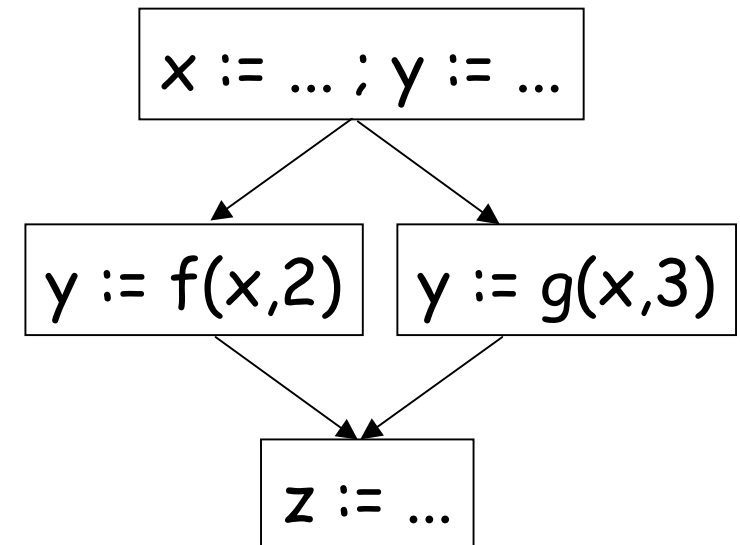
## ■ Testdaten:

- Teilmenge der Eingabedaten der Testfälle, mit denen das Programm tatsächlich ausgeführt wird

# Kontrollflußbezogene Verfahren (1)

- Darstellung der Programm(-teile) als Kontrollflußgraphen
  - Anweisungen (Anweisungsblöcke) als Knoten des Graphen
  - Kontrollfluss als gerichtete Kanten zwischen Knoten
  - Zweig: Einheit aus einer Kante und den dadurch verbundenen Knoten
  - Pfad: Sequenz von Knoten und Kanten, die am Startknoten beginnt und am Endknoten endet

```
x := ... ; y := ...  
IF x > 5 AND y < 2  
THEN y := f(x,2)  
ELSE y := g(x,3)  
z := ...
```



# Kontrollflussbezogene Verfahren (2)

---

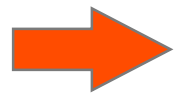
- Prinzip der "Überdeckung von Kontrollstrukturen" durch Testfälle (Coverage Analysis):
  - Gezieltes Durchlaufen (von Teilen) der Kontrollstrukturen durch geeignete Gestaltung der Testfälle
  - Angestrebter/erreichter Überdeckungsgrad in Prozent angegeben

# Arten der Überdeckung von Kontrollstrukturen

---

- Anweisungsüberdeckung:
  - Alle Anweisungen werden mindestens einmal ausgeführt
- Zweigüberdeckung:
  - Alle Verzweigungen im Kontrollfluss werden mindestens einmal verfolgt
- Bedingungsüberdeckung:
  - Alle booleschen Wertekonstellationen von (Teil-) Bedingungen werden einmal berücksichtigt
- Pfadüberdeckung:
  - Durchlaufen aller Pfade von Start- zum Endknoten (außer bei sehr kleinen Programmen nur theoretisch möglich)

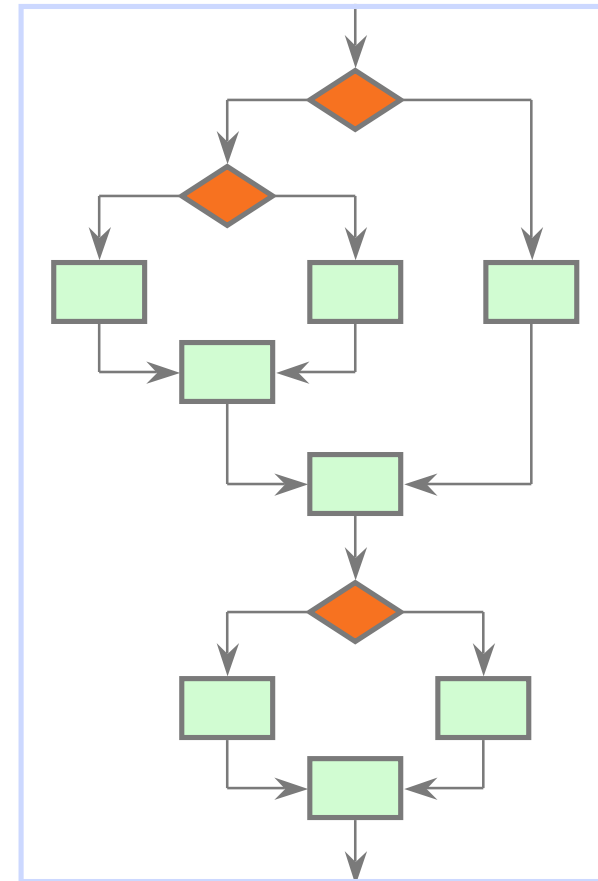
# Coverage Analysis



Validierung der Vollständigkeit von Testreihen anhand von Metriken

Arten:

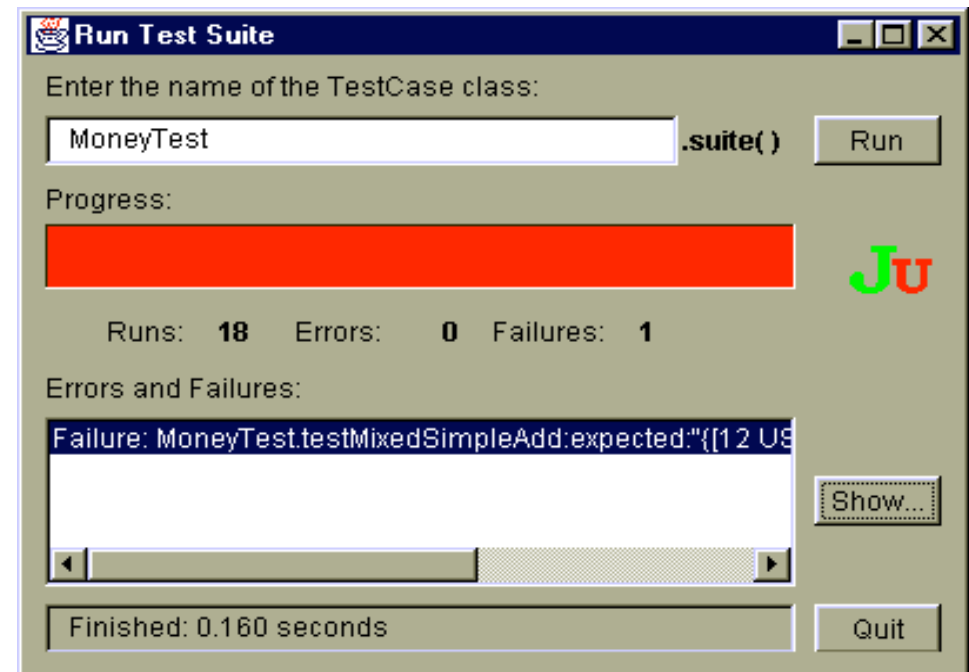
- Anweisungsüberdeckung (statement coverage)  
-> zu schwach
- Entscheidungsüberdeckung (branch coverage)  
-> minimum mandatory testing requirement
- Pfadüberdeckung (path coverage)  
-> in der Praxis nicht durchführbar



# JUnit

---

- Framework, um den Unit-Test eines Java-Programms zu automatisieren.
- einfacher Aufbau
- leicht erlernbar



Many thanks to Carrara Engineering for making their slides available: <http://www.carrara.ch/cspb/html/Tools/JUnit/>

# Konstruktiver Ansatz

---

- 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.

# Simple Test Case [1/2]

---

```
public class Money {  
  
    private double amount;  
  
    public Money(double amount) {  
        this.amount = amount;  
    }  
  
    public double getAmount () {  
        return amount;  
    }  
}
```

# Simple Test Case [2/2]

---

```
import junit.framework.*;

public class MoneyTest extends TestCase {

    public MoneyTest(String name) {
        super(name);
    }

    public void testAmount() {
        Money money = new Money(2.00);
        assertEquals("err-msg", 2.00, money.getAmount());
    }

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(MoneyTest.class);
    }
}
```

## Anatomie eines Testfalls [1/3]

---

- Jede Testklasse wird von der Framework-Basisklasse `junit.framework.TestCase` abgeleitet.
- Jede Testklasse erhält einen Konstruktor für den Namen des auszuführenden Testfalls.

## Anatomie eines Testfalls [2/3]

---

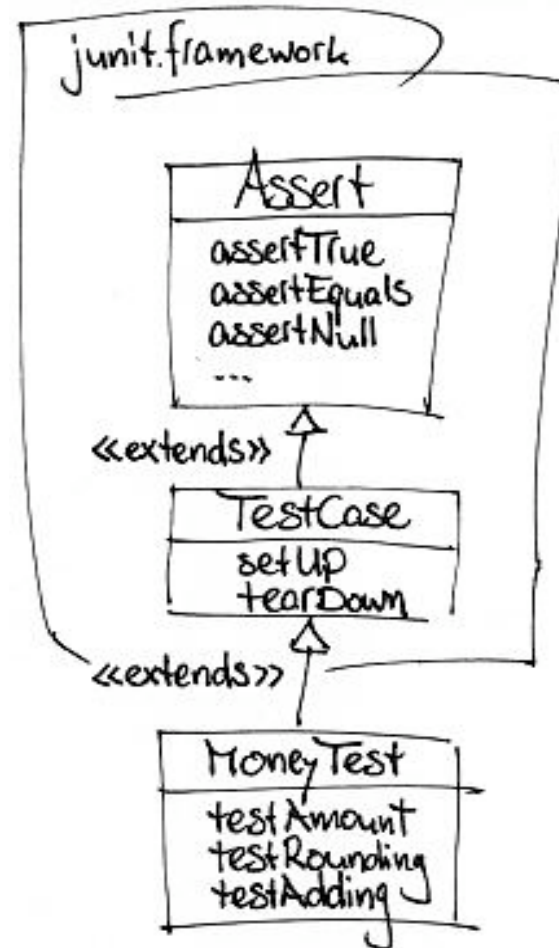
- JUnit erkennt die Testfälle anhand des Signaturmusters  
`public void testXXX()`
- Das Framework kann mit dem Java-Reflection-Package die Testfälle somit automatisch erkennen.

## Anatomie eines Testfalls [3/3]

---

- 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.

# Das JUnit-Framework



## Assert [1/4]

---

- Die Klasse `Assert` definiert eine Menge von `assert` Methoden, welche die Testklassen erben.
- Mit den `assert` Methoden können unterschiedliche Behauptungen über den zu testenden Code aufgestellt werden.
- Trifft eine Behauptung nicht zu, wird ein Testfehler protokolliert.

# Was kommt beim nächsten Mal?

