

# Vorlesung "Software-Engineering"

---

Prof. Ralf Möller, TUHH, Arbeitsbereich STS

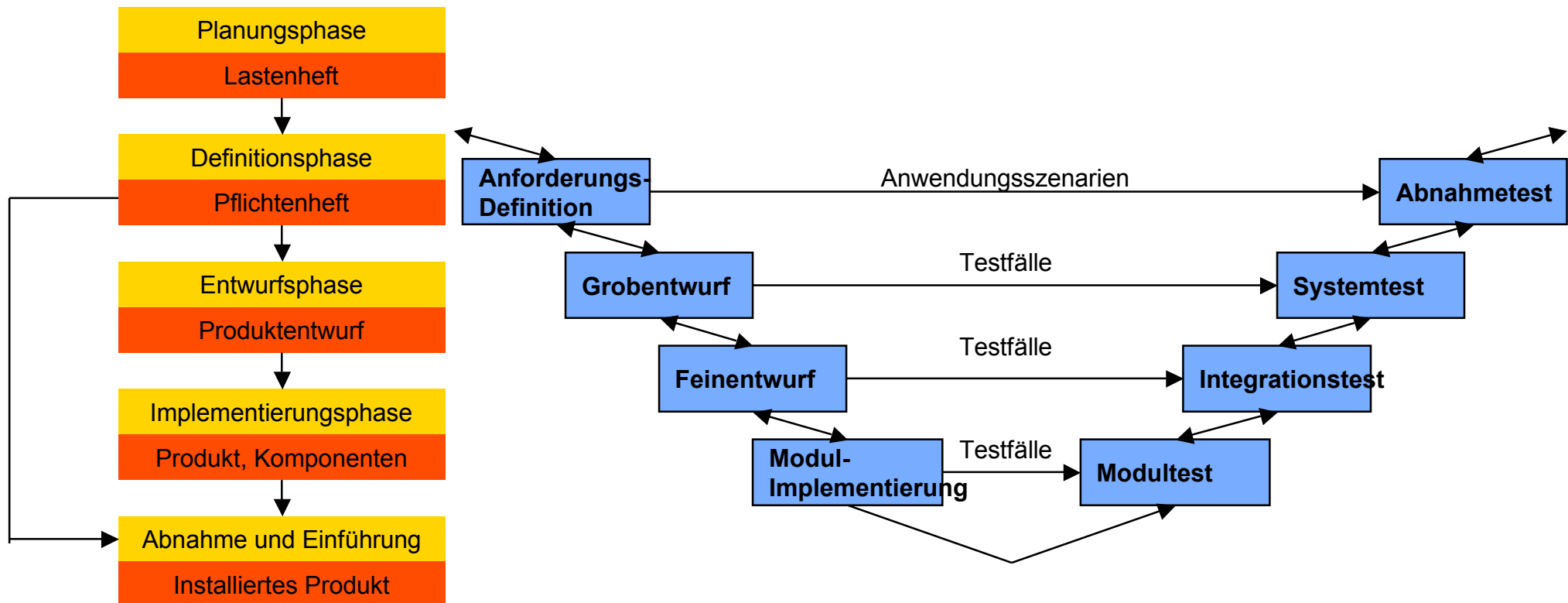
## ■ Vorige Vorlesung

- Analytische Qualitätssicherung
- Metriken
- Testverfahren

## ■ Heute:

- Fortsetzung: Testverfahren
- Konstruktive Qualitätssicherung

# Einordnung



# 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

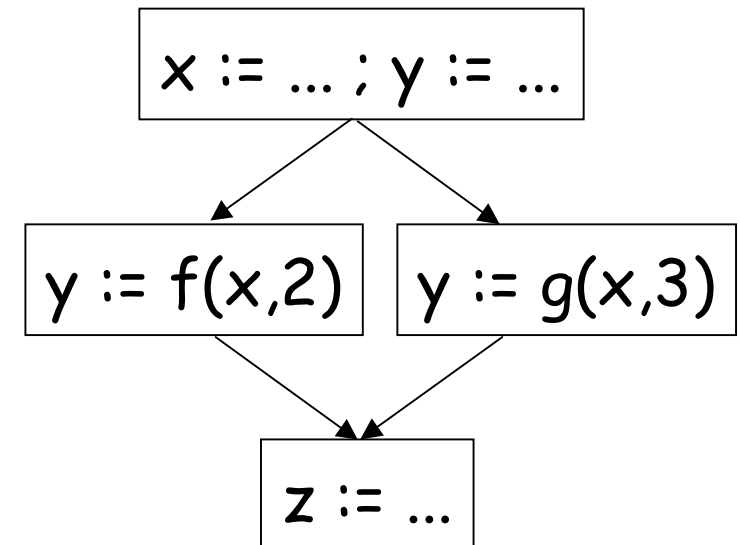
## ■ Testdaten:

- 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)

---

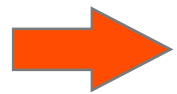
- Prinzip der "Überdeckung von Kontrollstrukturen" durch Testfälle (Coverage Analysis):
  - Gezieltes Durchlaufen (von Teilen) der Kontrollstrukturen durch geeignete Gestaltung der Testfälle
  - Angestrebter/erreichter Überdeckungsgrad in Prozent angegeben

# 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)

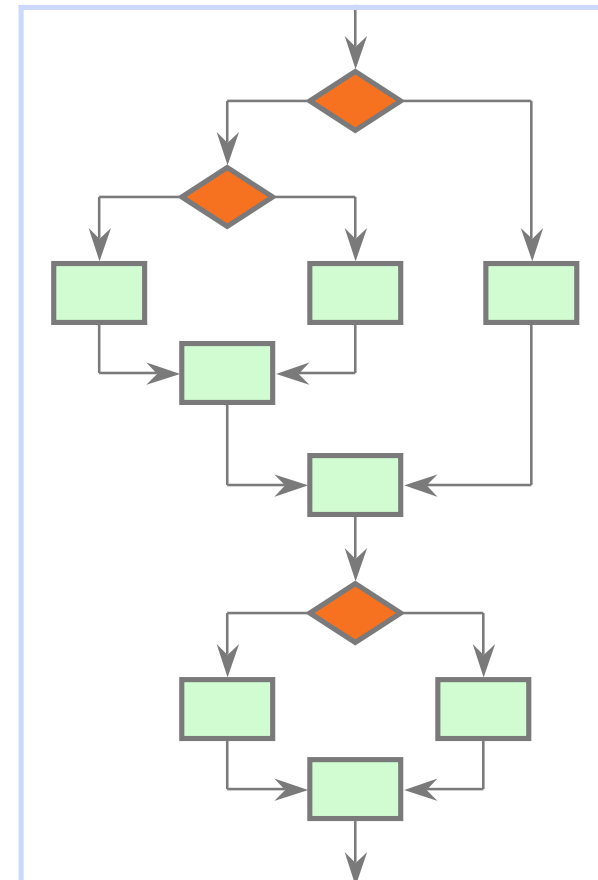
# Coverage Analysis



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

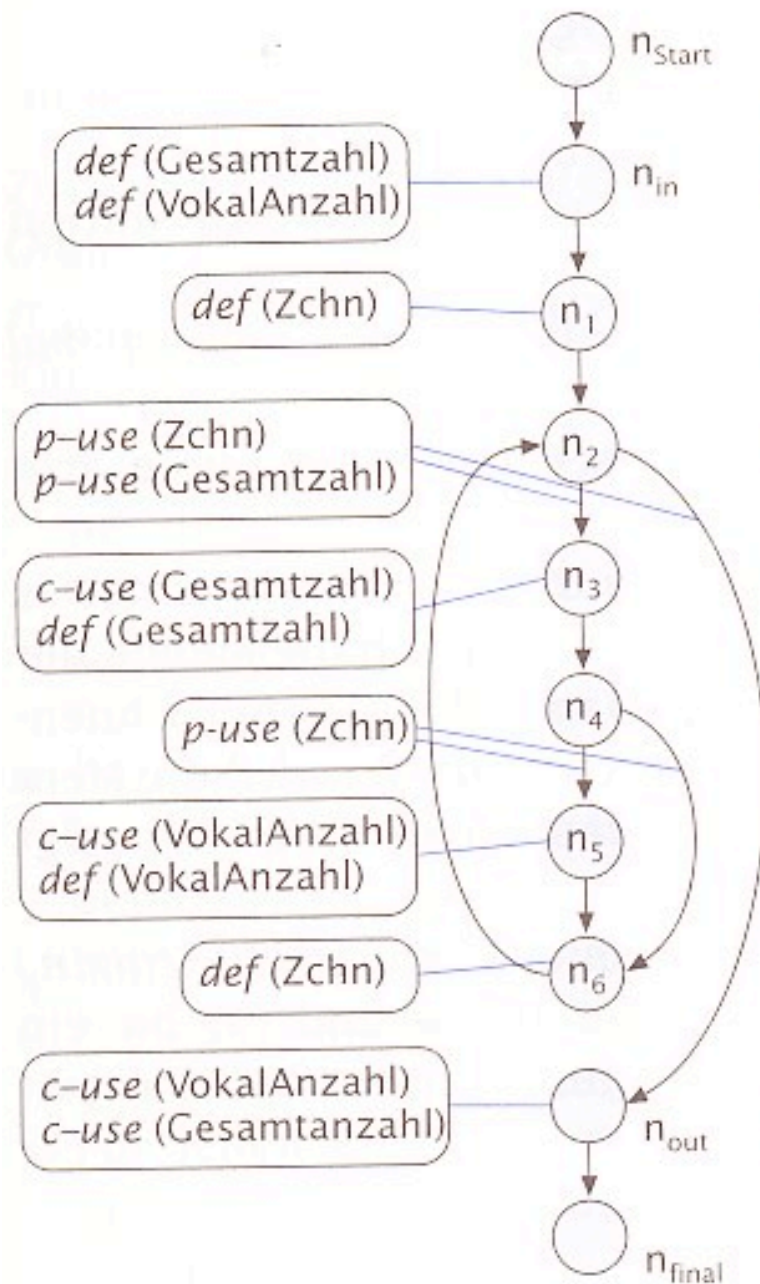




# Datenflußbezogene Verfahren (1)

---

- Erweiterung der Kontrollflußgraphen um Datenflüsse
- Unterscheidung verschiedener Situationen hinsichtlich des Zugriffs auf Variablen im Programm:
  - Definition von Variablenwerten (Wertzuweisung)
  - Berechnende Benutzung (zur Ermittlung eines Wertes in Ausdrücken)
  - Prädikative Benutzung (Auswertung der Bedingungen in bedingten Anweisungen oder Schleifen)



Import von 'VokalAnzahl'  
und 'Gesamtzahl'

```
char Zchn;
cin >> Zchn;
```

```
while ((Zchn >= 'A') && (Zchn <= 'Z')
      && (Gesamtzahl < INT_MAX))
```

```
Gesamtzahl = Gesamtzahl + 1;
```

```
if ((Zchn == 'A') || (Zchn == 'E') ||
    (Zchn == 'I') || (Zchn == 'O') ||
    (Zchn == 'U'))
    VokalAnzahl = VokalAnzahl + 1;
```

```
cin >> Zchn;
```

Export von 'VokalAnzahl'  
und 'Gesamtzahl'

Abb. 5.5-2:  
Kontrollflußgraph –  
Datenfluß-  
darstellung

# Datenflußbezogene Verfahren (2)

---

## ■ Prinzip des Verfahrens:

- Für jeden Definitionsknoten werden definitionsfreie Pfade zu allen Benutzungsknoten ermittelt
- Die Auswahl der Testdaten muß das Durchlaufen jedes identifizierten definitionsfreien Pfades sicherstellen

# Datenflußbezogene Verfahren (3)

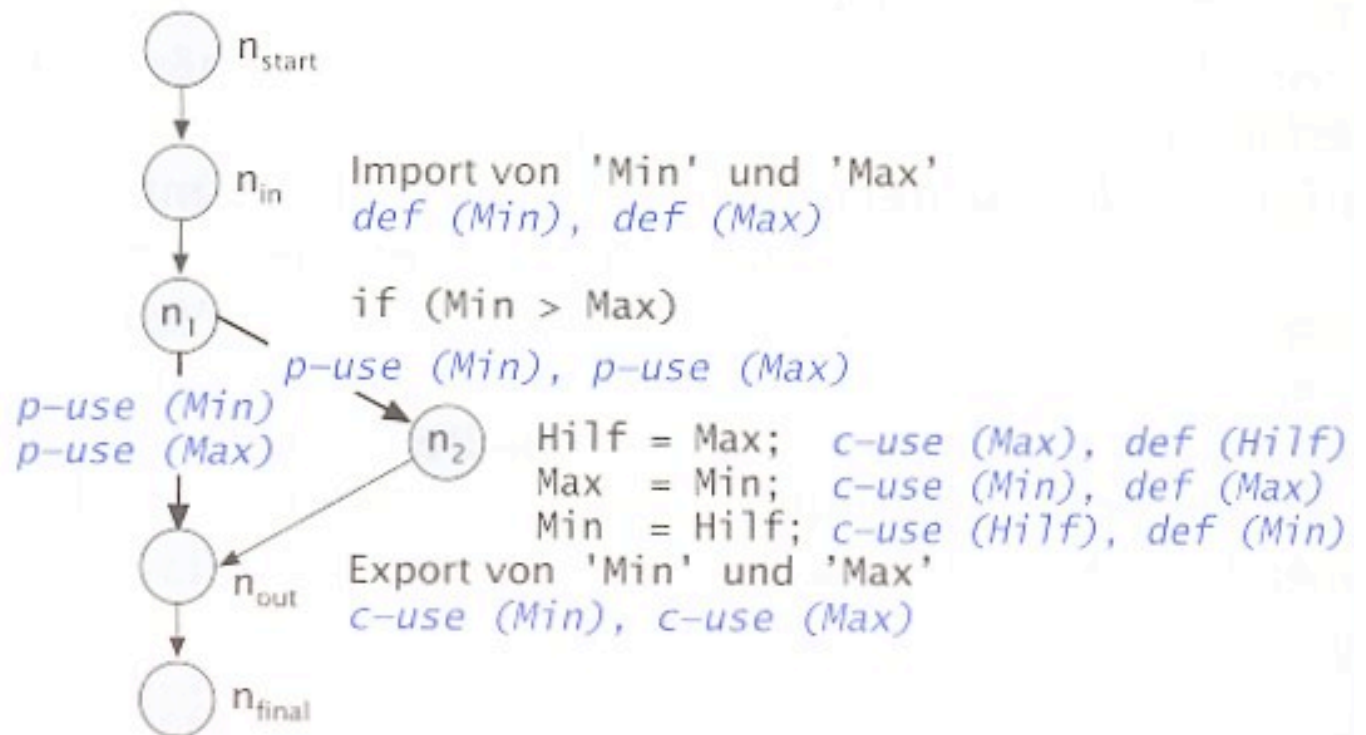
- Definition von drei Funktionen als Basis der Datenflußverfahren:
  - Funktion **def** bildet jeden Knoten auf die Menge der darin global definierten Variablen ab
  - Funktion **c-use** ordnet jedem Knoten die in ihm berechnend benutzten Variablen zu
  - Funktion **p-use** weist jeder Kante die Menge von Variablen zu, deren prädikative Auswertung zum Durchlaufen der Kante notwendig ist
- Definition der Menge  $\mathbf{du}(v, n_i) = \mathbf{dcu}(v, n_i) + \mathbf{dpu}(v, n_i)$ , wobei  $v \in \mathbf{def}(n_i)$  und folgendes gilt:
  - $\mathbf{dcu}(v, n_i)$  enthält alle Knoten  $n_j$  mit  $v \in \mathbf{c-use}(n_j)$  und ein Pfad von  $n_i$  nach  $n_j$  existiert, auf dem  $v$  nicht neu definiert wird
  - $\mathbf{dpu}(v, n_i)$  enthält alle Kanten  $(n_j, n_k)$  mit  $v \in \mathbf{p-use}((n_j, n_k))$ , wobei ein Pfad von  $n_i$  nach  $n_j$  existiert, auf dem  $v$  nicht neu definiert wird
- Ein **definitionsfreier Pfad**  $(n_1, \dots, n_k)$  bzgl. einer Variablen  $v$  ist eine Sequenz von Knoten, so daß  $v$  nicht Element von  $\mathbf{def}(n_i)$  mit  $i$  aus  $1..k$

# Beispiel (1)

## Quellprogramm

```
void MinMax  
(int Min, int Max)  
{  
    int Hilf;  
    if (Min > Max)  
    {  
        Hilf = Max;  
        Max = Min;  
        Min = Hilf;  
    }  
}
```

## Kontrollflußgraph in Datenflußdarstellung



Die *c-uses* bezogen auf Min und Max im Knoten  $n_2$  sind *global*, da die letzte, unmittelbar vorangehende Definition in Knoten  $n_{in}$  geschieht.

Der berechnende Zugriff auf Hilf ist – wegen der im gleichen Block vorangestellten Definition – *lokal*.

## Beispiel (2)

Pro Knoten Zuordnung der *global* definierten Variablen  $def(n_i)$  und der *globalen* berechnenden Zugriffe  $c-use(n_i)$

Jeder Kante Zuordnung der Variablen, für die ein prädikativer Zugriff erfolgt  $p-use(n_i, n_j)$

Knoten $n_i$	$def(n_i)$	$c-use(n_i)$	Kanten	$p-use$
$n_{in}$	{Min, Max}	{ }	$(n_1, n_2)$	{Min, Max}
$n_1$	{ }	{ }	$(n_1, n_{out})$	{Min, Max}
$n_2$	{Min, Max}	{Min, Max}		
$n_{out}$	{ }	{Min, Max}		

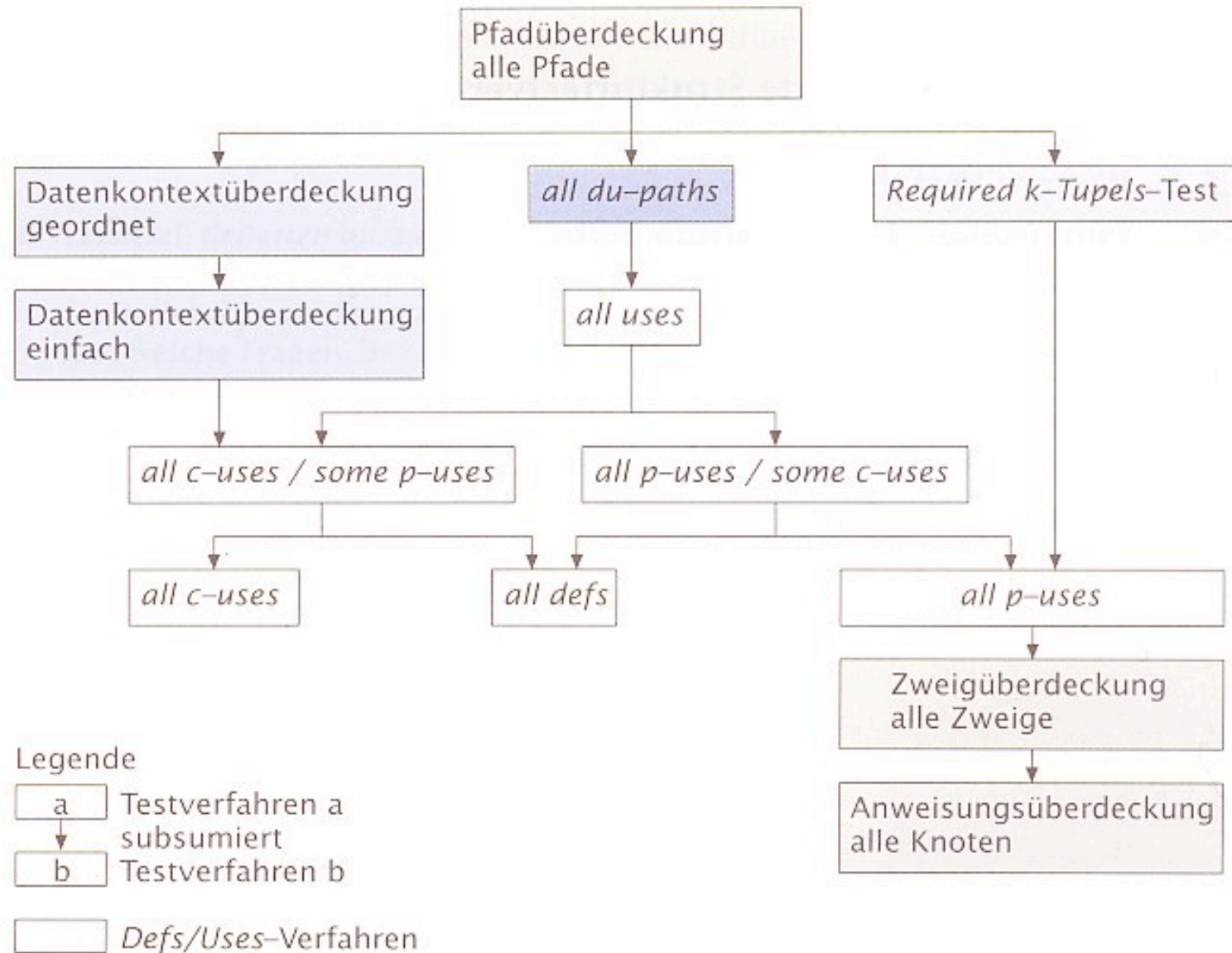
*dcu* und *dpu*

Variable $x$	Knoten $n_i$	$dcu(x, n_i)$	$dpu(x, n_i)$
Min	$n_{in}$	{ $n_2, n_{out}$ }	{( $n_1, n_2$ ), ( $n_1, n_{out}$ )}
Min	$n_2$	{ $n_{out}$ }	{ }
Max	$n_{in}$	{ $n_2, n_{out}$ }	{( $n_1, n_2$ ), ( $n_1, n_{out}$ )}
Max	$n_2$	{ $n_{out}$ }	{ }

# Datenflußbezogene Verfahren (4)

- All-defs-Kriterium
  - Für jeden Knoten und jede Variable  $v \in \text{def}(n_i)$  wird mindestens ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu **allen** Elementen von  $\text{du}(v, n_i)$  ausgeführt
- All-p-uses-Kriterium
  - Für jeden Knoten und jede Variable  $v \in \text{def}(n_i)$  wird mindestens ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu **allen** Elementen von  $\text{dpu}(v, n_i)$  ausgeführt
- All-c-uses-Kriterium
  - Für jeden Knoten und jede Variable  $v \in \text{def}(n_i)$  wird mindestens ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu **allen** Elementen von  $\text{dcu}(v, n_i)$  ausgeführt
- All-c-uses-some-p-uses-Kriterium
  - Für jeden Knoten und jede Variable  $v \in \text{def}(n_i)$  wird mindestens ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu **allen** Elementen von  $\text{dcu}(v, n_i)$  ausgeführt
  - Falls  $\text{dcu}(v, n_i)$  leer ist, wird ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu **einem** Element von  $\text{dpu}(v, n_i)$  ausgeführt
- Analog: All-p-uses-some-c-uses-Kriterium
- All-uses-Kriterium
  - Für jeden Knoten und jede Variable  $v \in \text{def}(n_i)$  wird mindestens ein definitionsfreier Pfad bzgl.  $v$  von  $n_i$  zu allen Elementen von  $\text{du}(v, n_i)$  ausgeführt.







# Funktionale Verfahren

---

- Überprüfung der in der Spezifikation festgelegten Funktionalität
  - Programmstruktur irrelevant
  - Beste Grundlage: formale Spezifikation
- Bestimmung der Testdaten:
  - Bildung "funktionaler Äquivalenzklassen":  
Eingabe- und Ausgabemengen, die jeweils zu einer (Teil-) Funktionalität gehören
  - Alle Werte einer Äquivalenzklasse verursachen ein identisches funktionales Verhalten eines Programms
- Auswahl von Testdaten aus den Äquivalenzklassen:
  - Zufällig
  - Test spezieller Werte (z.B. 0, nil)
  - Grenzwertanalyse (primär Grenzbereiche der Eingabemengen als Testdaten)

# Testen von Moduln

---

- Aufdecken von Fehlern in der isolierten Modulrealisierung
- Modultest wird meist als Teilphase der Implementierung betrachtet
- Notwendigkeit einer Testumgebung, die die anderen Module simuliert und das Modul selbst benutzbar macht (Fehler in Testumgebung?)

# Testumgebungen

---

- (Wiederholten) Aufruf des Testobjektes ermöglichen
- Eingabedaten für Testobjekt bereitstellen
- Externe Ressourcen und deren Ergebnisse/Aktivitäten simulieren (z.B. importierte Module)
- Ergebnisse der Testausführung ausgeben und speichern

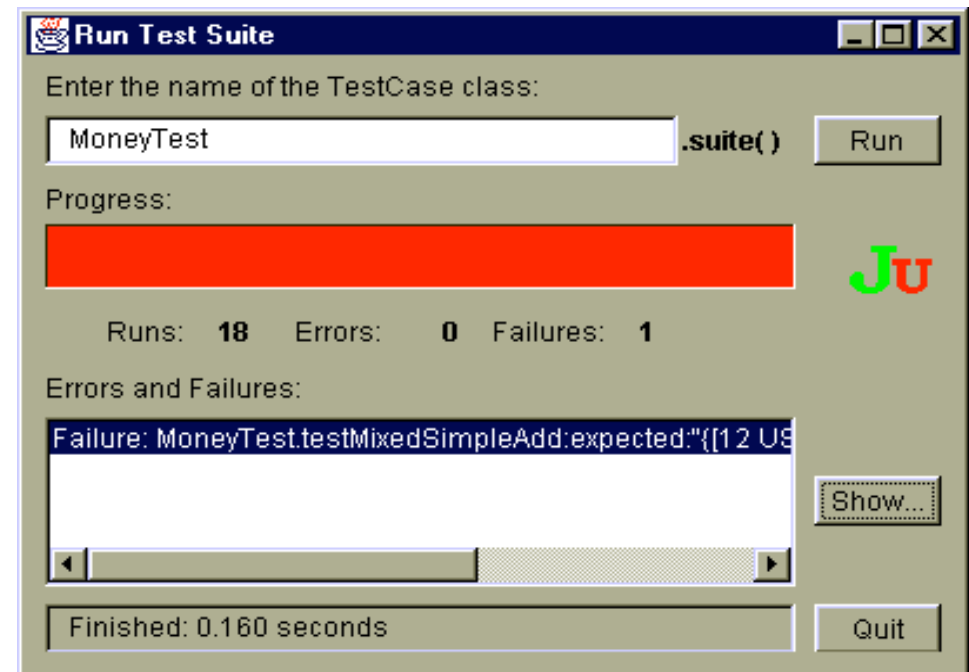
# Testen von Modulverbindungen

---

- Testen von Subsystemen  
(= Teilmengen von verknüpften Moduln)
- Prüfung der Kommunikation zwischen den Moduln
- Schrittweiser Aufbau größerer Subsysteme durch Hinzufügen weiterer Module/Subsysteme  
(inkrementelles Testen, Integrationstest)
- Ausführung meist Bottom-up, jedoch auch Top-Down-Vorgehensweise möglich
- Ebenfalls Testumgebung notwendig, "Stub"-Moduln bzw. Treiber-Moduln erforderlich

# 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]

---

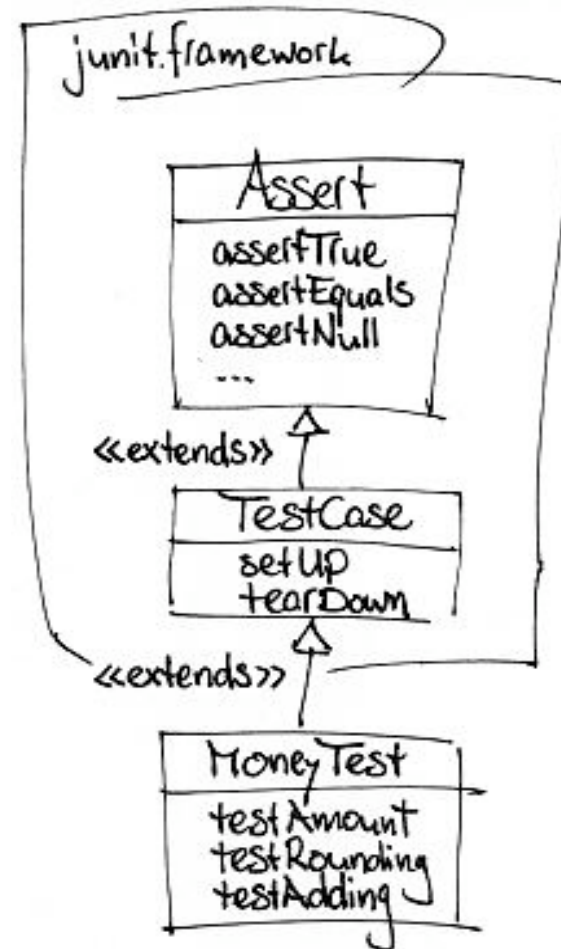
- JUnit erkennt die Testfälle anhand des Signaturmusters  
`public void testXXX()`
- Das Framework kann mit dem Java-Reflection-Package die Testfälle somit automatisch erkennen.

## 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.

## Assert [2/4]

---

- `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.

## Assert [3/4]

---

- `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.

Es existieren überladene Methoden mit einer Zeichenkette als erstem Argument für die Angabe der Fehlermeldung im Protokoll.

# Assert [4/4]

---

- `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.



# Testen von Exceptions [1/2]

---

- Exceptions werden in Java mit dem `catch` Block behandelt.
- Mit der Methode `fail` aus der `Assert` Klasse wird ein Testfehler ausgelöst und protokolliert.

## Testen von Exceptions [2/2]

---

- Im vorliegenden Beispiel wird beim Aufruf des Konstruktors der Klasse `MyClass` mit einer negativen Zahl die `IllegalArgumentException` ausgelöst.

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

# TestFixture [1/2]

---

- 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.

## TestFixture [2/2]

---

- In der Methode `setUp` werden Instanzvariablen initialisiert.
- Mit der Methode `tearDown` werden wertvolle Testressourcen wie zum Beispiel Datenbank- oder Netzwerkverbindungen wieder freigegeben.

# Beispiel TestFixture

---

```
import junit.framework.*;

public class MoneyTest extends TestCase {

    private Money money;

    ...

    protected void setUp() {
        money = new Money(2.00);
    }
    protected void tearDown() {
    }

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

    ...
}
```

# Lebenszyklus eines Testfalls

---

## ■ Testklasse MoneyTest mit Methoden testAdding und testAmount.

- new MoneyTest("testAdding")
- setUp()
- testAdding()
- tearDown()
- New MoneyTest("testAmount")
- setUp()
- testAmount()
- tearDown()

# TestSuite [1/5]

---

- Um eine Reihe von Tests zusammen ausführen zu können, werden die Tests zu TestSuites zusammengefasst.
- Eine Suite von Test wird dabei durch ein `TestSuite` Objekt definiert.
- Mit JUnit können beliebig viele Test in einer TestSuite zusammengefasst werden.

## TestSuite [2/5]

---

- 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]



# TestSuite [3/5]

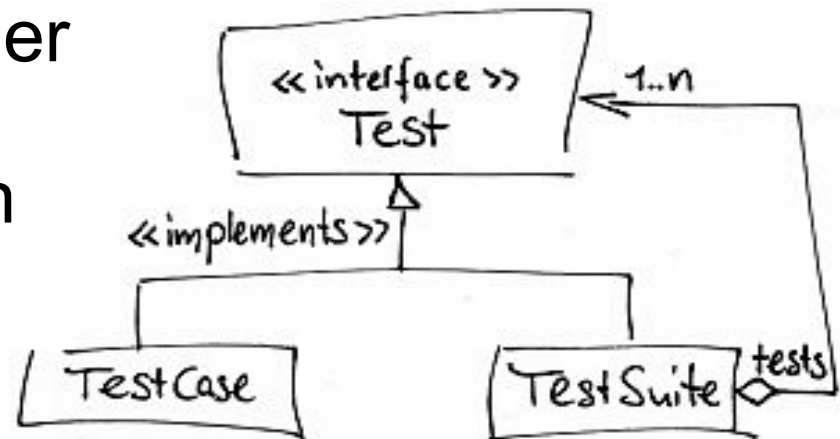
---

```
import junit.framework.*;

public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(MoneyTest.class);
        suite.addTestSuite(Formatter.class);
        suite.addTestSuite(TestSheet.class);
        return suite;
    }
}
```

## TestSuite [4/5]

- Damit beliebig viele TestCase und TestSuite Objekte zu einer umfassenden TestSuite-Hierarchie kombiniert werden kann, wird das Composite-Pattern verwendet.



## TestSuite [5/5]

---

- In vielen Fällen ist es praktisch, pro Package eine TestSuite-Klasse zu definieren.
- Es hat sich eingebürgert, diese Klasse `AllTests` zu nennen.

# Beispiel AllTests – Einbindung von Test-Klassen

---

```
package com.iq.htmlFormatter;

import junit.framework.*;

public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("htmlFormatter");
        suite.addTestSuite(SpacerTest.class);
        suite.addTestSuite(LevelTest.class);
        return suite;
    }

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

# Beispiel AllTests – Einbindung von AllTests-Klassen

---

```
package com.iq.html;

import junit.framework.*;

public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("Package html");
        suite.addTest(com.iq.html.tag.AllTests.suite());
        return suite;
    }

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

# Testausführung

---

- In vielen Fällen ist es praktisch, pro `AllTests` Klasse eine `main` Methode zu definieren, welche die graphische Benutzeroberfläche aufruft und die eigene Klasse als Parameter übergibt.

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

# Zusammenfassung

---

- Junit
  - Einfaches Framework für den Unit-Test von Java-Programmen.
  - Testfälle werden parallel zum eigentlichen Code entwickelt.
  - Die Tests können vollautomatisch ausgeführt werden.

# Qualitätssicherung durch Tests

---

- Ein Test ist prinzipiell nur geeignet, evtl. vorhandene Fehler eines Systems zu Tage treten zu lassen, nicht jedoch die Abwesenheit von Fehlern zu zeigen (vgl. Verifikation)
- Der **Überdeckungsgrad** ist ein Maß für den Grad der Vollständigkeit eines Tests
- Ein **Regressionstest** ist ein Testverfahren, bei dem eine Sammlung von Testfällen erstellt wird, die bei Änderungen am System (automatisch) erneut überprüft werden



# Qualitätssicherung durch Tests

---

- Beim **Alpha-Test** für Standardsoftware wird das System in der Zielumgebung des Herstellers durch ausgewählte Anwender erprobt
- Beim **Beta-Test** für Standardsoftware wird das System bei ausgewählten Zielkunden in einer eigenen Umgebung zur Probenutzung zur Verfügung gestellt. Auftretende Probleme und Fehler werden protokolliert. Beta-Tester erhalten typischerweise einen Preisnachlaß auf das endgültige Produkt. Typischerweise werden Beta-Tests iterativ mit aufeinanderfolgenden "*release candidates*" (RC) durchgeführt.
- Bei einer Individualsoftware findet ein **Abnahmetest** statt

# Testen: Zusammenfassung

---

- Rolle der Spezifikation beim Test
- Testsystematik
- Reproduzierbarkeit von Tests  
(insbesondere nach Änderungen)
- Tests decken Fehler auf
- Tests helfen nicht, Fehler zu vermeiden
- Herstellungsprozeß bedeutsam:  
-> Qualitätsmanagement, konstruktive Verfahren

# Konstruktive Verfahren

---

- Annahme: Die Qualität eines Produktes wird maßgeblich durch die Qualität des Herstellungsprozesses bestimmt
  - Qualität ins Bewußtsein der Projektmitglieder bringen
  - Rückverfolgbarkeit gewährleisten
  - Verantwortlichkeiten und Zuständigkeiten schaffen (Planstellen)
  - -> Organisatorischen Rahmen für Software-Herstellungsprozeß schaffen


# ISO 9000 (erstmalig standardisiert 1987)

---

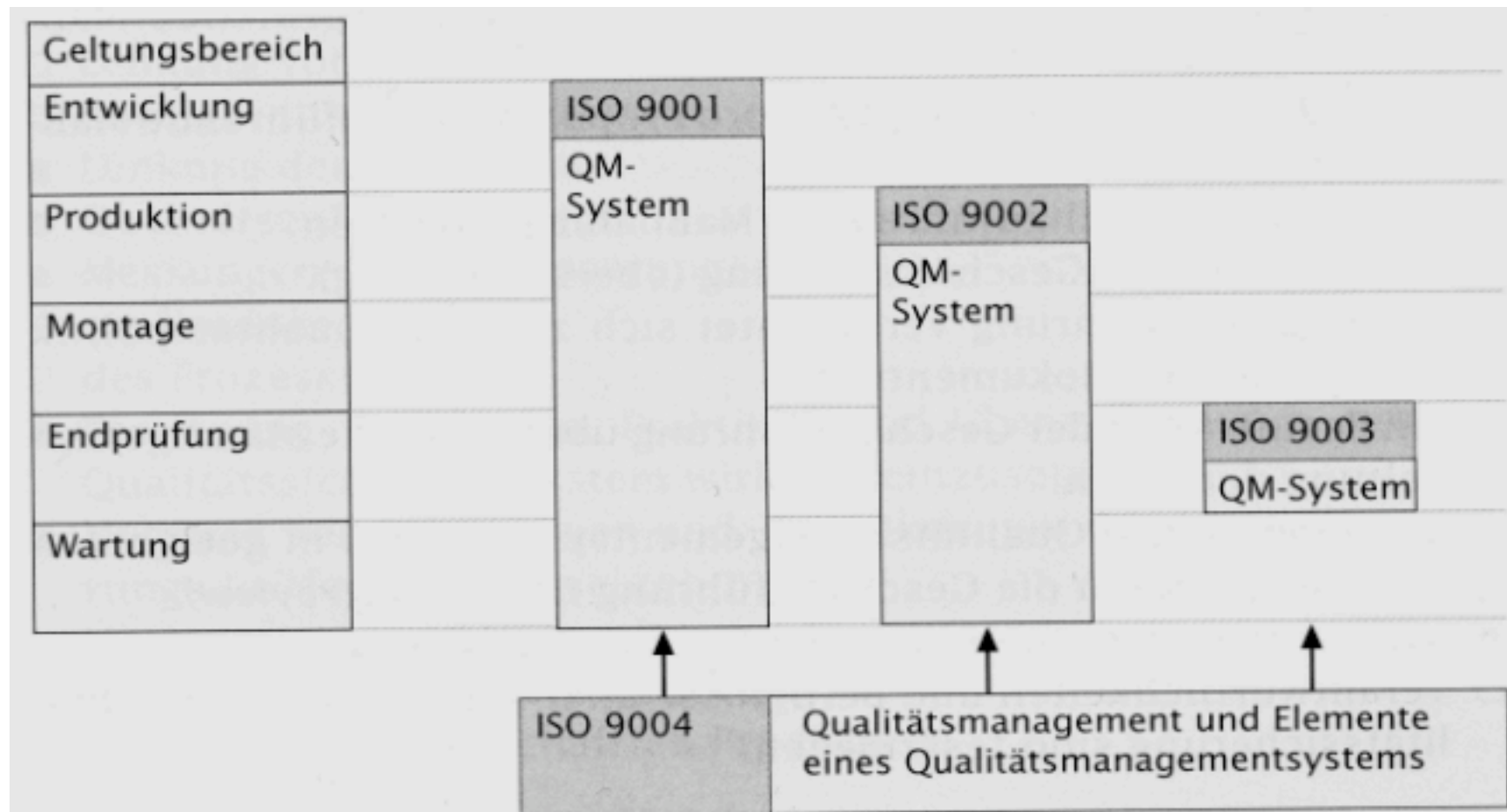
- Das ISO 9000 Normenwerk legt für ein Auftraggeber-Lieferanten-Verhältnis einen allgemeinen, übergeordneten, organisatorischen Rahmen zur Qualitätssicherung von Produkten (insbesondere ISO 9001).
- Geprägt durch industrielle Fertigung von Produkten, aber ausgeweitet auf Erstellung von Software (9000-3)
- Zertifizierung nach 9001 bzw. 9000-3 durch Zertifizierungsstelle, wenn ein Qualitäts-managementsystem vorhanden.
- Zertifiziert wird der Herstellungsprozeß,  
NICHT DAS PRODUKT

# ISO 9000 - Struktur (1)

---

ISO 8402	Begriffsbestimmungen
	
ISO 9000-1	Leitfaden zur Auswahl und Anwendung
ISO 9000-2	Allgemeiner Leitfaden zur Anwendung von 9001, 9002 und 9003
ISO 9000-3	Leitfaden zur Anwendung von 9001 auf Software
ISO 9000-4	Leitfaden zum Management von Zuverlässigkeitsprogrammen

# ISO 9000 - Struktur (2)



# ISO 9000-3

---

## ■ **Aufbau ISO 9000-3:**

- Rahmen
- Lebenszyklustätigkeiten
- Unterstützende Tätigkeiten

## ■ **Inhalt ISO 9000-3:**

- Entwicklung
- Lieferung
- Wartung von Software

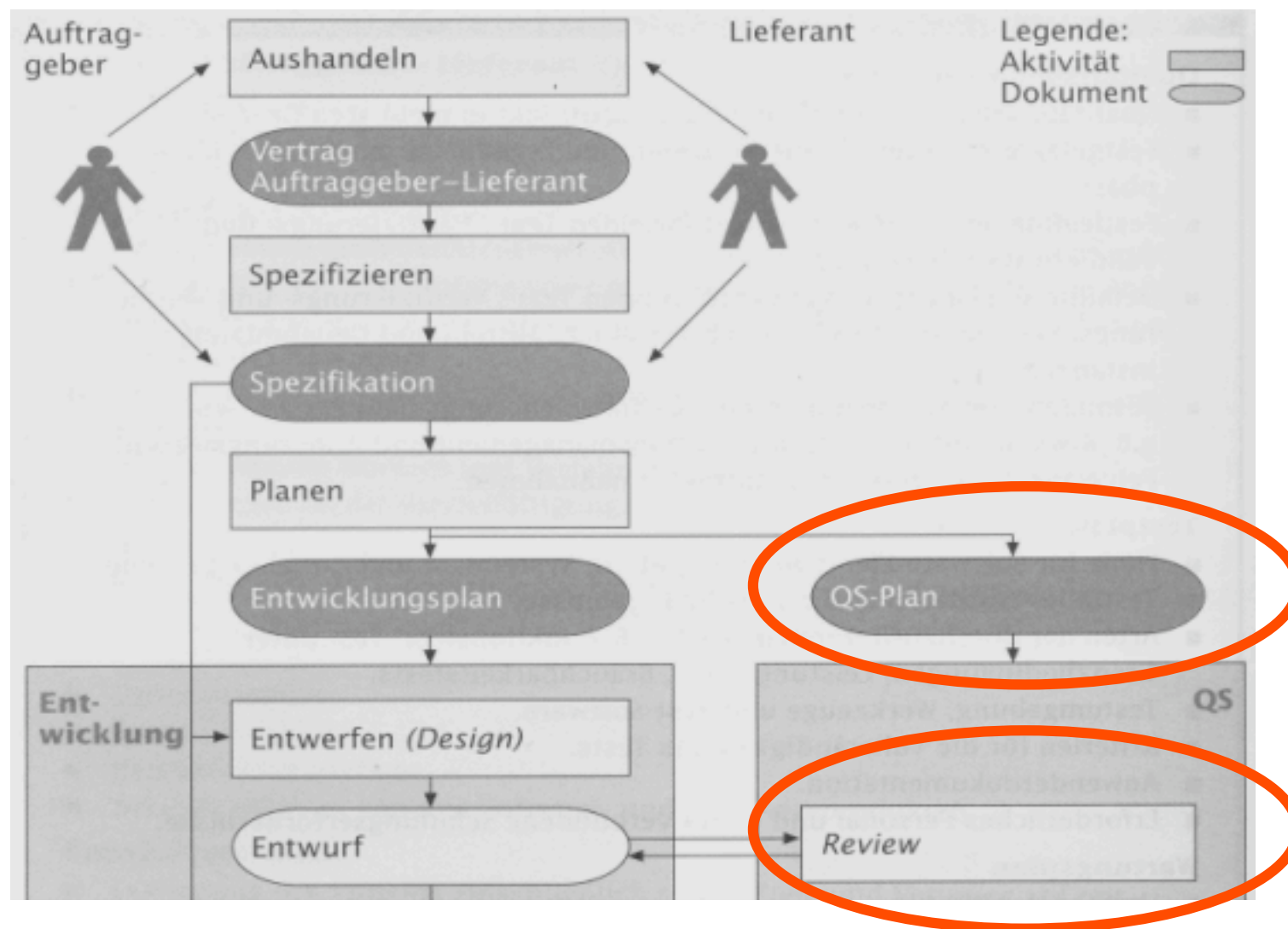
## ■ **Maßnahmen:**

- der Geschäftsführung
- der Mitarbeiter der Qualitätssicherung

## ■ **Anwendungsformen:**

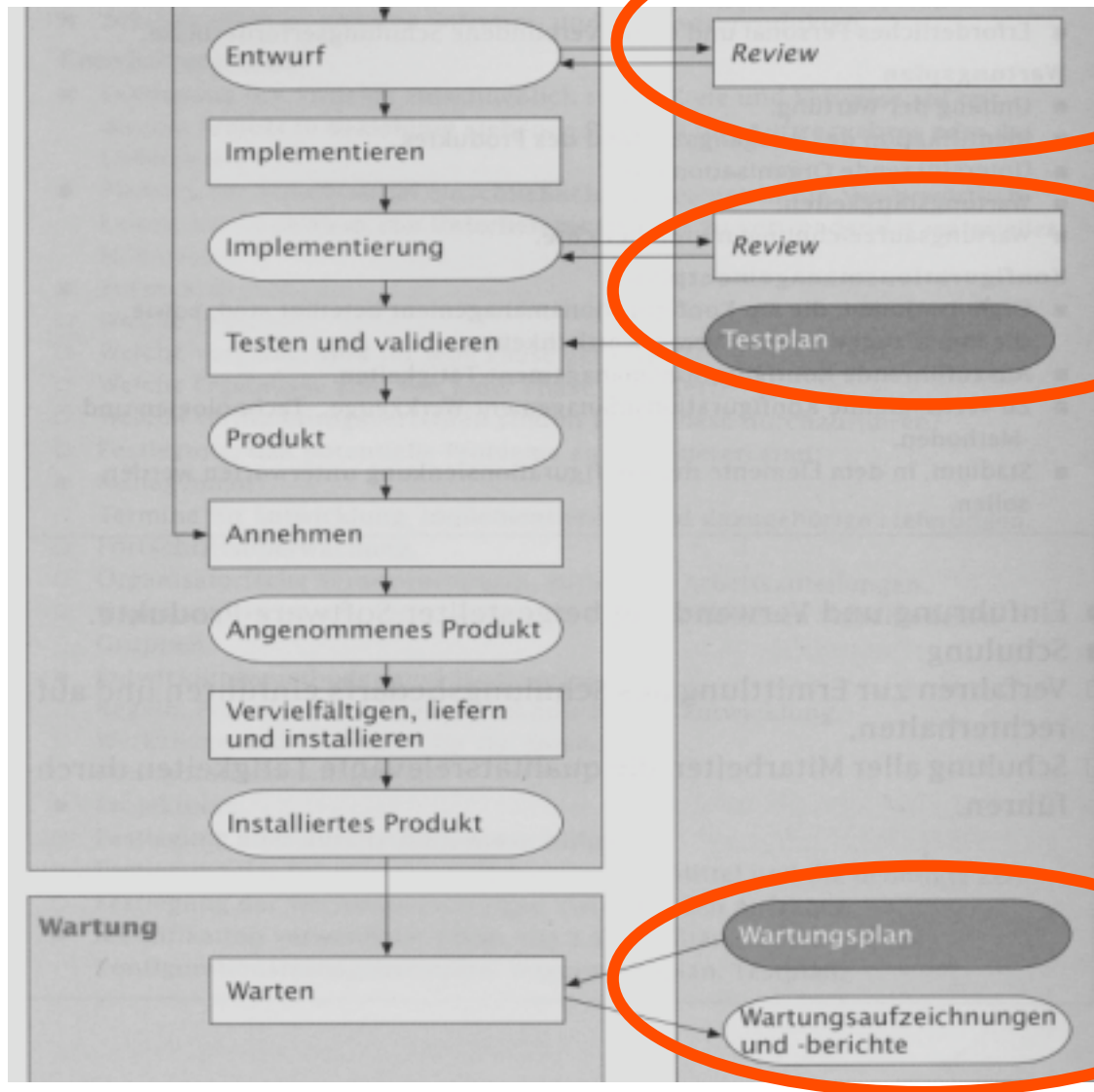
- Darlegung der Qualitätssicherung gegenüber Dritten
- Aufbau und Verbesserung eines QS-Systems

# ISO 9000-3 - Implizites Vorgehensmodell





# ISO 9000-3 - Implizites Vorgehensmodell



# ISO 9000-3 - Implizites Vorgehensmodell

## ■ Phasenunabhängige Tätigkeiten:

### ■ Konfigurationsmanagement:

- | Identifikation und Rückverfolgbarkeit d. Konfiguration
- | Lenkung von Änderungen
- | Konfigurations-Statusbericht

### ■ Lenkung der Dokumente

### ■ Qualitätsaufzeichnungen

### ■ Messungen und Verbesserungen:

- | am Produkt
- | des Prozesses

### ■ Festlegung von Regeln, Praktiken und Übereinkommen für den Einsatz eines Qualitätsmanagementsystems

### ■ Nutzung von Werkzeugen und Techniken, um QS-Leitfaden umzusetzen

### ■ Unterauftragsmanagement

# ISO 9000 - Bewertung

---

## ■ Vorteile:

- Aufmerksamkeit auf Qualitätssicherung
- Externe Zertifizierung und Wiederholung d. Audit
- Festlegen von Anforderungen
- Erleichtert die Akquisition von Aufträgen
- Weniger Produkthaftungsrisiko
- Stärkung Qualitätsbewußtsein

## ■ Nachteile:

- Unsystematischer Aufbau
- Keine sauber Trennung zwischen fachlichen, Management- und QS-Aufgaben
- Gefahr Software-Bürokratie
- Gefahr mangelnde Flexibilität
- Hoher Aufwand für Einführung und Pflege

# Andere Modelle

---

- TQM (Total Quality Management)
- CMM (Capability Maturity Model)
- SPICE (Software Process Improvement and Capability dEtermination)
  
- Siehe: Balzert, Lehrbuch der Softwaretechnik Band 2