Technische Universität
Kaiserslautern

o1o1seda o1o1oo

software engineering dependability

Software Entwicklung 2

Softwaretest

Inhalt

- Testphasen
- Dynamischer Test
- Strukturelle, kontrollflussorientierte Verfahren
- Strukturelle, datenflussorientierte Verfahren
- Funktionsorientierter Test
- Literatur

Lernziele

- Die Testphasen kennen und erläutern können
- Die Eigenschaften des dynamischen Tests erklären können
- Die strukturellen, kontrollflussorientierten Testtechniken kennen, erläutern und anwenden können
- Die funktionsorientierten Testtechniken kennen, erläutern und anwenden können

Testphasen

- Voraussetzung für eine Prüfung umfangreicher Softwaresysteme in Phasen ist deren modularer Aufbau. Der Vorteil des Prüfens in unterschiedlichen Phasen ist die Reduktion der jeweiligen Komplexität auf ein überschaubares Niveau
- Modultest (Überprüfung der Module)
  - Prüfung des korrekten Funktionierens eines Moduls bezogen auf seine Modulspezifikation
- Integrationstest (Überprüfung des Zusammenwirkens der Module)
  - Schrittweises Zusammenfügen der Module zum Gesamtsystem. Prüfung des korrekten Zusammenwirkens über Schnittstellen
- System-/Abnahmetest
  - Überprüfung der Funktionalität, Leistung und Qualität einer Software gegen die festgelegten Anforderungen
- Es können weitere Testphasen hinzutreten (z. B. Hardware-/Software-Integrationstest) und Testphasen aufgeteilt werden (z. B. Integrationstest in Subsystemintegrationstest und Systemintegrationstest)

Modultest, Klassentest

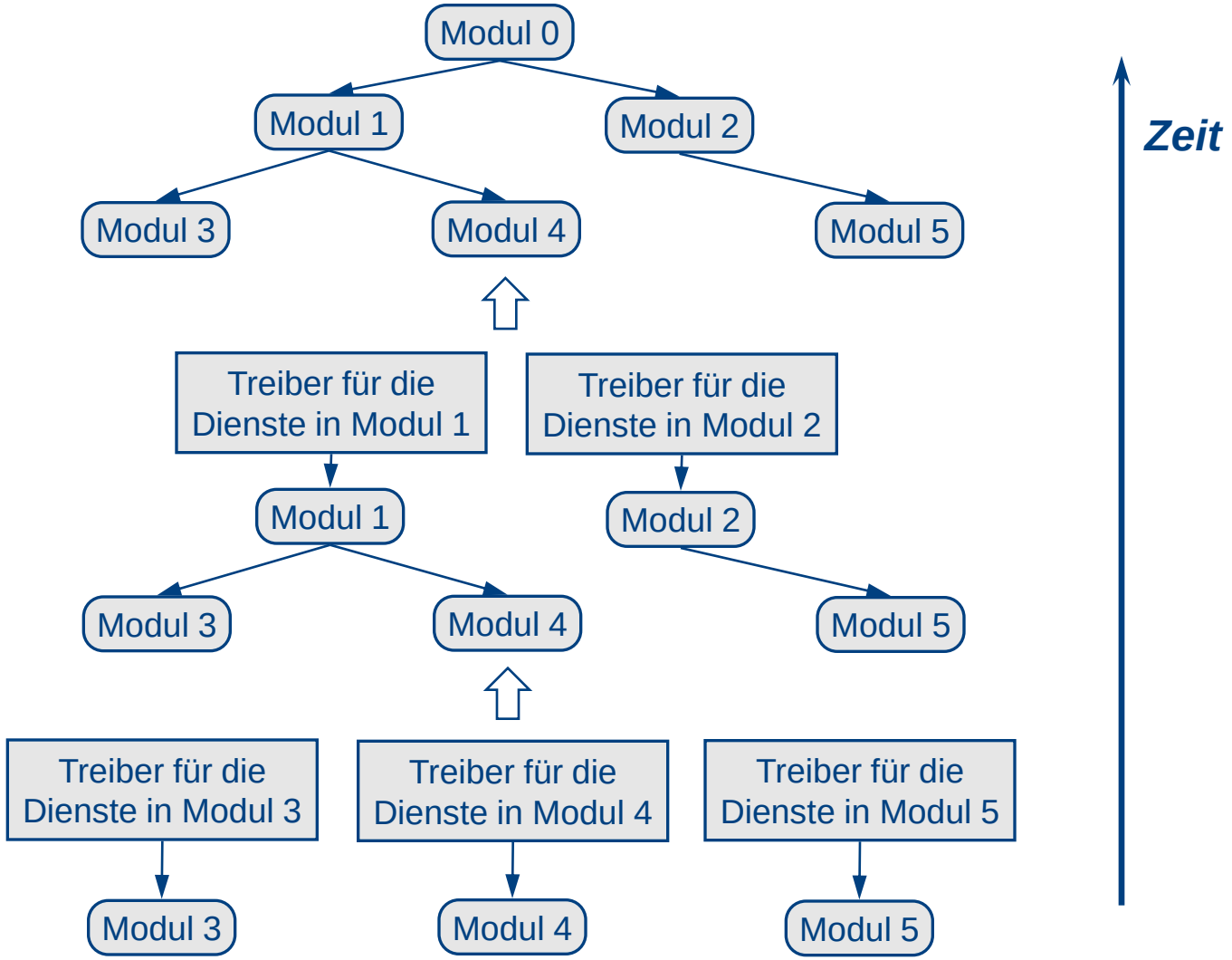
- Test eines einzelnen Moduls (oft eine Klasse)
- Treiber bzw. Dummies erforderlich ➔ Testumgebung, Testbett

- Akzeptierte Minimalkriterien
  - Funktionstest: Test gegen die in der Spezifikation festgelegte Funktionalität
  - Strukturtest: Test gegen die Codestruktur ➔ minimal: Zweigüberdeckungstest

Integrationstest

- Die Prüfverfahren des Integrationstests dienen zur Überprüfung der Schnittstellen und des Interagierens über Schnittstellen (Schnittstelle: Aufruf von Operationen, Funktionen mit und ohne Parameterübergabe, Verwendung von globalen Variablen oder Dateien)
- Integrationsstrategien
  - Bottom-Up-Integrationstest
  - Top-Down-Integrationstest
  - Outside-In-Integrationstest

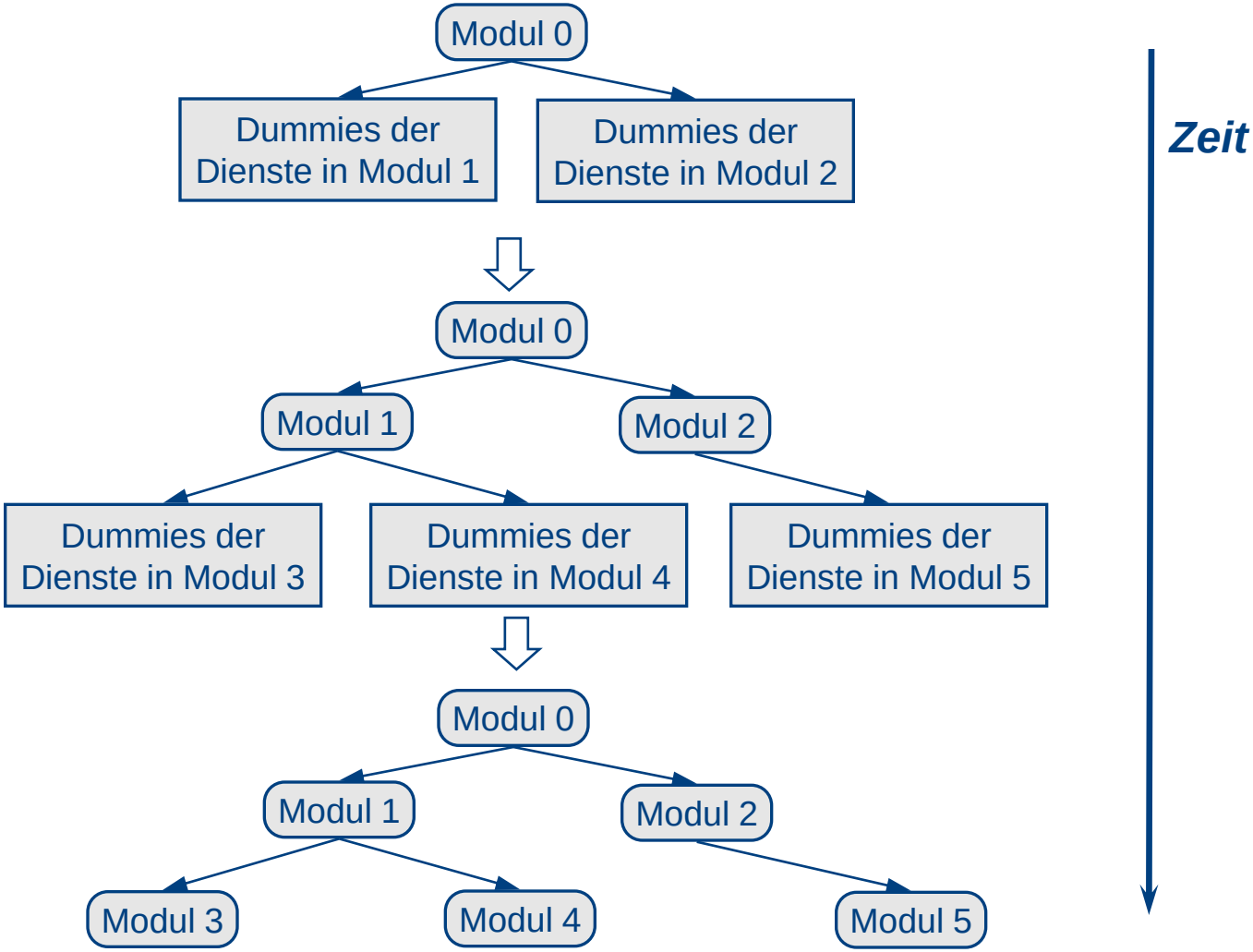
Integrationstest
Bottom-Up-Integrationstest



Integrationstest
Bottom-Up-Integrationstest

- Vorteile
  - Zusammenwirken zwischen zu prüfender Software, Systemsoftware und Hardware wird früh geprüft
  - Da Testdateneingaben über Treiber erfolgen, ist keine komplexe Zurückrechnung der Eingaben erforderlich
  - Bewusste Fehleingaben zur Prüfung von Fehlerbehandlungen sind leicht möglich
- Nachteile
  - Treiber erforderlich
  - Gezielte Prüfung der Fehlerbehandlung bei fehlerhaften Rückgabe-werten unterlagerter Routinen ist kaum möglich, da die realen Routinen benutzt werden
  - Ein vorzeigbares Produkt entsteht erst ganz zuletzt, da die koordi-nierenden Module erst dann hinzugefügt werden
  - Abnehmender Personalbedarf mit fortschreitendem Test

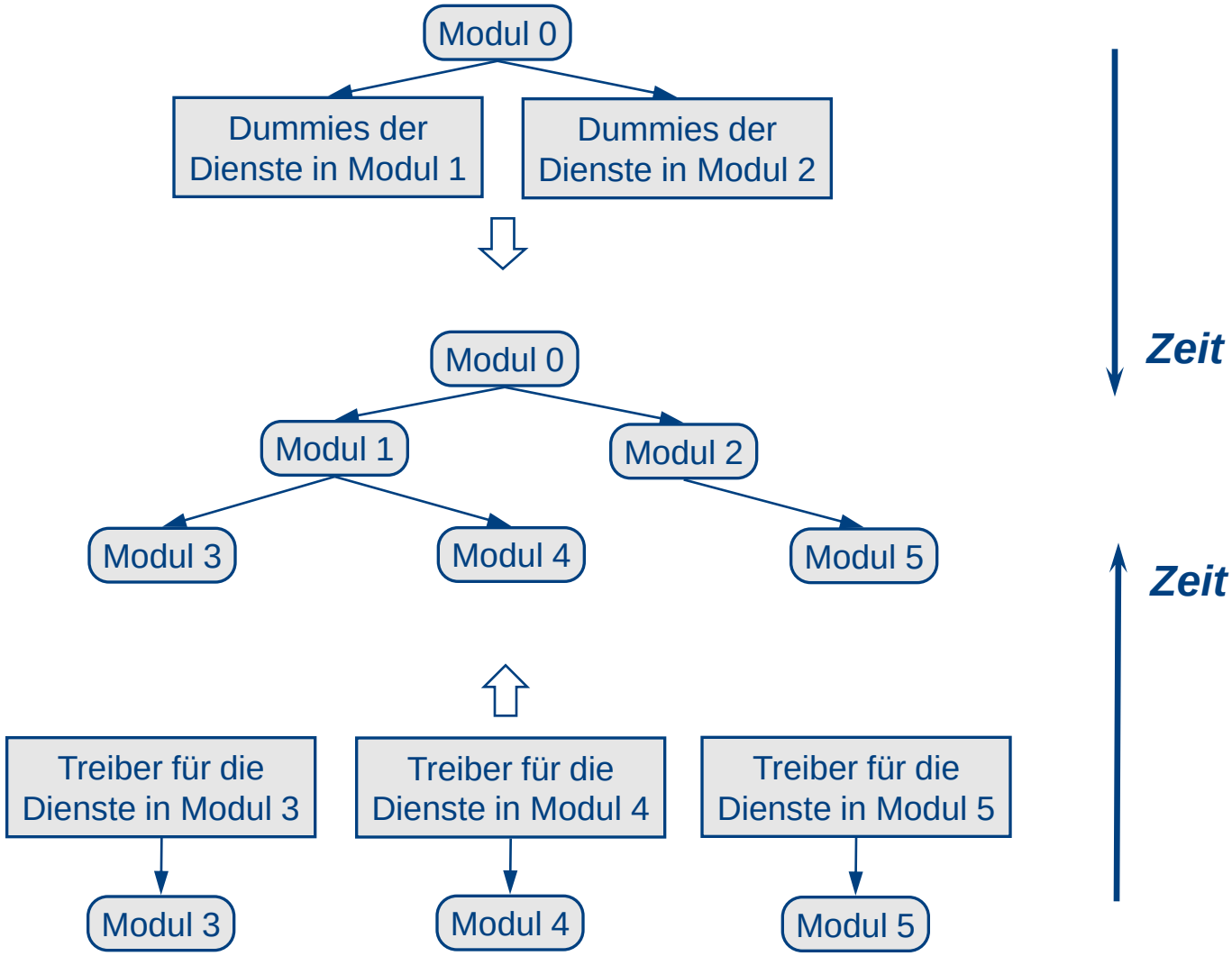
Top-Down-Integrationstest



Integrationstest
Top-Down-Integrationstest

- Vorteile
  - Wichtige Steuerungsfunktionalität wird zuerst geprüft
  - Bereits zu Beginn entsteht ein Produkt, das die groben Abläufe erkennen lässt
  - Gezielte Prüfung der Fehlerbehandlung bei fehlerhaften Rückgabe-werten unterlagerter Routinen ist möglich, da Rückgabewerte durch dummies eingegeben werden
- Nachteile
  - Dummies erforderlich
  - Mit zunehmender Integrationstiefe wird die Erzeugung bestimmter Testsituationen in tiefer angeordneten Modulen schwieriger
  - Zusammenwirken zwischen zu prüfender Software, Systemsoftware und Hardware wird spät geprüft
  - Zunehmender Personalbedarf während des Tests

Integrationstest
Outside-In-Integrationstest



Integrationstest
Outside-In-Integrationstest

- Vorteile
  - Wichtige Steuerungsfunktionalität wird zuerst geprüft
  - Bereits zu Beginn entsteht ein Produkt, das die groben Abläufe erkennen lässt
  - Gezielte Prüfung der Fehlerbehandlung bei fehlerhaften Rückgabewerten unterlagerter Routinen ist bei den Schichten, die von oben nach unten integriert werden, möglich, da Rückgabewerte durch dummies eingegeben werden. Diese Testvariante ist bei steuernden Modulen besonders wichtig, da diese umfang-reiche Fehlerbehandlungen durchführen
  - Zusammenwirken zwischen zu prüfender Software, Systemsoftware und Hardware wird früh geprüft
  - Da Testdateneingaben bei jenen Modulen, die von unten nach oben integriert werden, über Treiber erfolgen, ist keine komplexe Zurückrechnung der Eingaben erforderlich
  - Bewusste Fehleingaben zur Prüfung von Fehlerbehandlungen sind im unteren Bereich des Modulsystems leicht möglich
  - Der Personalbedarf ist während des Integrationstests konstanter
- Nachteil
  - Dummies und Treiber erforderlich

System- / Abnahmetest

- Test des fertigen Systems gegen die in der Anforderungsdefinition festgelegten Funktions-, Leistungs- und Qualitätsanforderungen
- Testfälle aus der Anforderungsdefinition, echte oder typische Daten mit denen das System im laufenden Betrieb versorgt wird, Erfahrungswerte, kritische Daten
- Test unter den realen Einsatzbedingungen mit der realen Systemkonfiguration
- Neben der QS-Abteilung sollen Benutzer des Systems oder andere Spezialisten für das System im Abnahmetest involviert sein

Dynamischer Test
Eigenschaften und Ziele

- Eigenschaften des dynamischen Tests
  - Übersetztes, ausführbares Programm wird mit konkreten Eingabewerten versehen und ausgeführt
  - Programm wird in der realen Umgebung getestet
  - Stichprobenverfahren
  - Korrektheit des getesteten Programms kann nicht bewiesen werden
- Merkmale des Einsatzes dynamischer Testverfahren in der Praxis
  - In einfacher Form weit verbreitet
  - Häufig unsystematisch angewendet
  - Tests oft nicht reproduzierbar
  - Diffuse Aktivität (Managementschwierigkeiten)

Dynamischer Test
Eigenschaften und Ziele

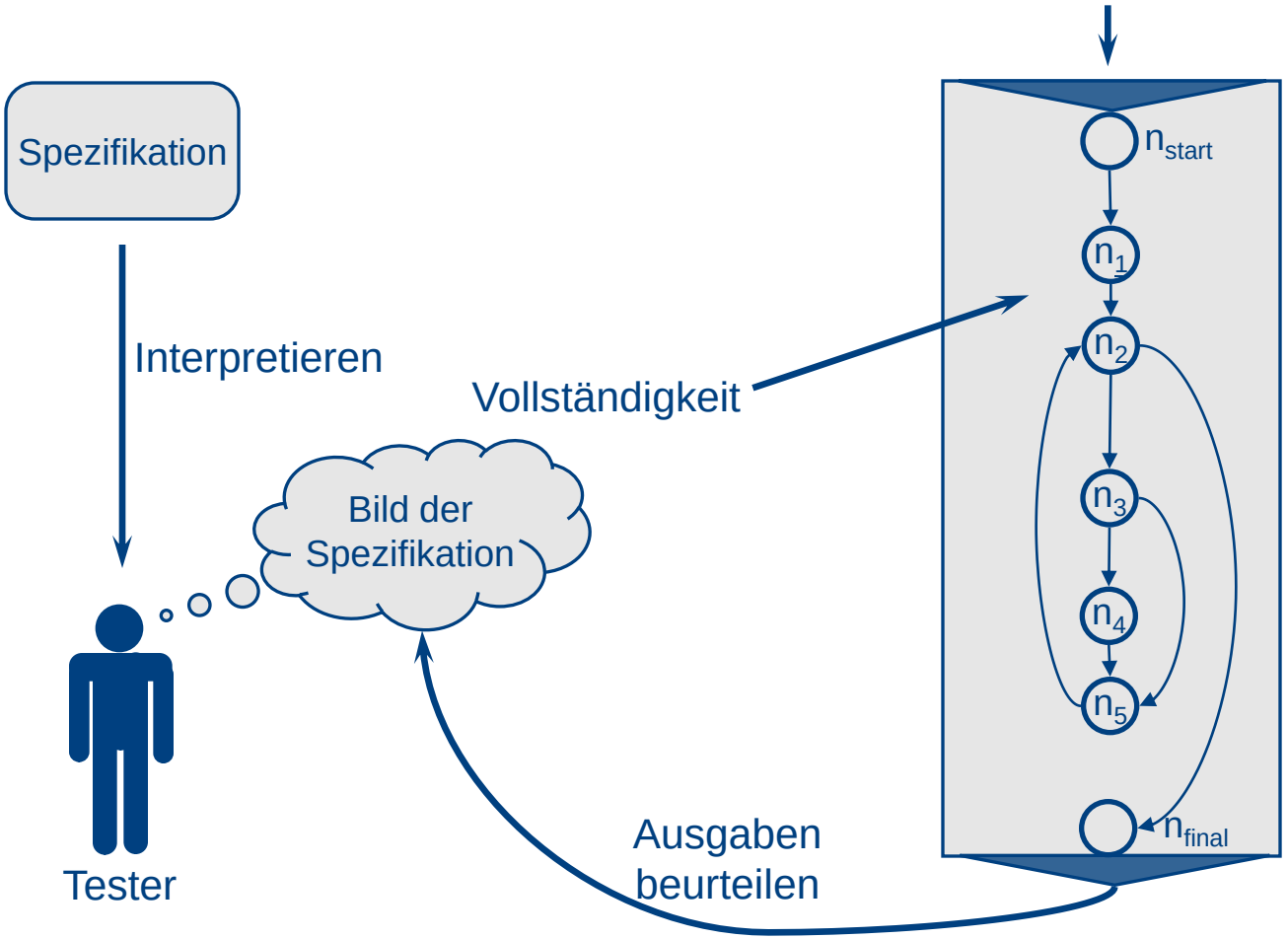
- Das Ziel der dynamischen Testverfahren ist die Erzeugung von Testfällen (Stichprobe der möglichen Eingaben), die
  - Repräsentativ
  - Fehlersensitiv
  - Redundanzarm
  - Ökonomisch

  sind

Dynamische Testtechniken: Überblick

- Strukturelle, kontrollflussorientierte Testtechniken
- Strukturelle, datenflussorientierte Testtechniken
- Funktionsorientierte Testtechniken

Strukturorientierte Verfahren
Arbeitsweise



Strukturorientierte Verfahren

- Arbeitsweise: Beurteilung der Adäquanz und der Vollständigkeit des Tests und evtl. Herleitung der Testdaten anhand der Modulstruktur. Beurteilung der Ausgaben anhand der Modulspezifikation
  - Vorteil: Implementationsstruktur wird beachtet (Anweisungen, Zweige, Datenzugriffe, etc.)
  - Nachteil: Nicht realisierte, aber spezifizierte Funktionen können nicht erkannt werden
- Kontrollflussorientierte Verfahren
- Datenflussorientierte Verfahren

Strukturorientierte, kontrollflussorientierte Testverfahren

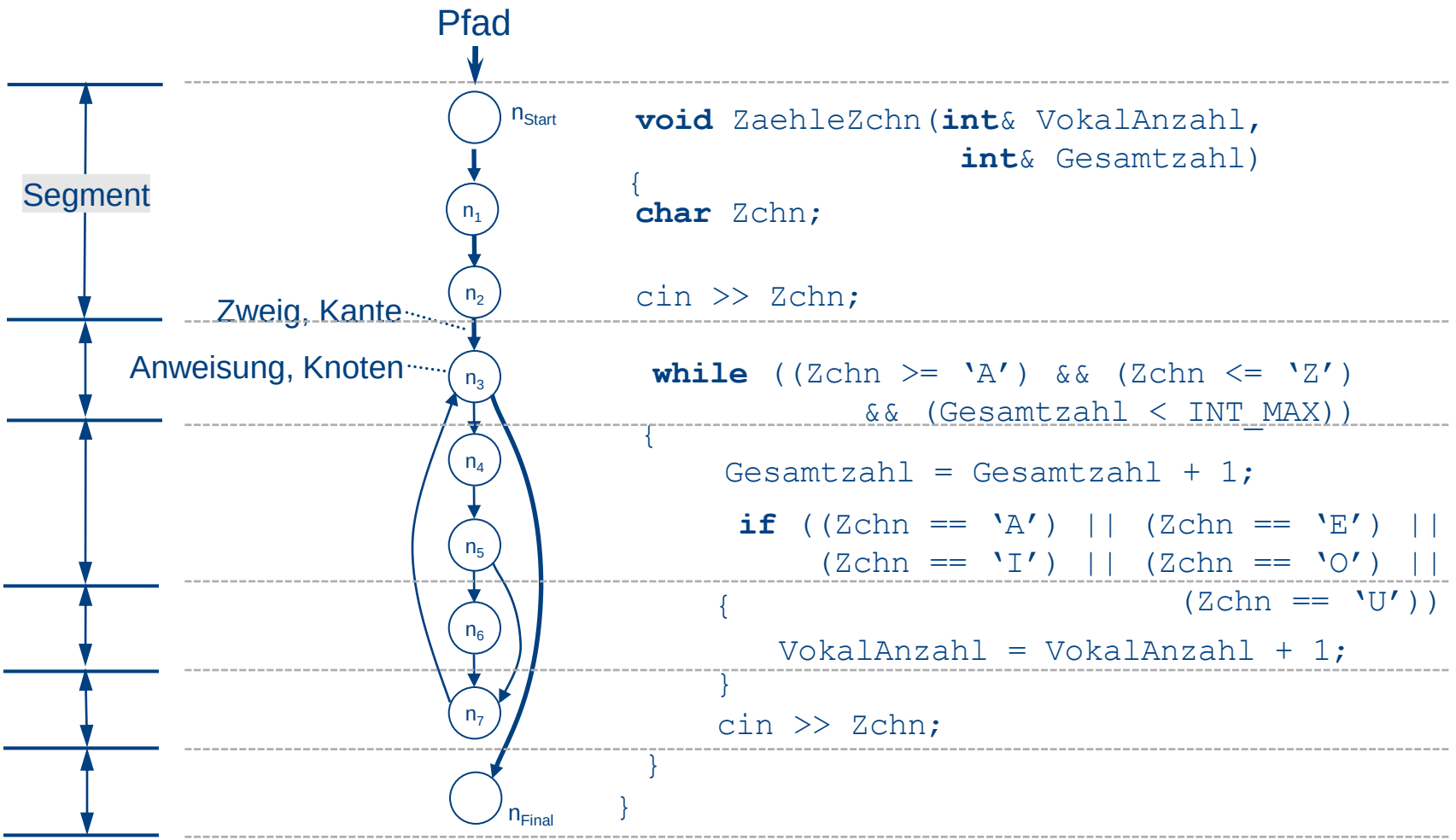
- Anweisungsüberdeckungstest
- Zweigüberdeckungstest
- Bedingungsüberdeckungstest
  - Einfacher
  - Minimaler mehrfacher
  - Mehrfacher
- LCSAJ-basierter Test
- *boundary interior*-Pfadtest
- Strukturierter Pfadtest
- Pfadtest

Die **kontrollflussorientierten Testverfahren** basieren auf der Kontrollstruktur bzw. dem Kontrollfluss. Die Basis bildet der Kontrollflussgraph.

Beispiel:

```
void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl)
// Vorbedingung: VokalAnzahl <= 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;
  } //end while
}
```

Kontrollflussdiagramm zur Operation ZaehleZchn

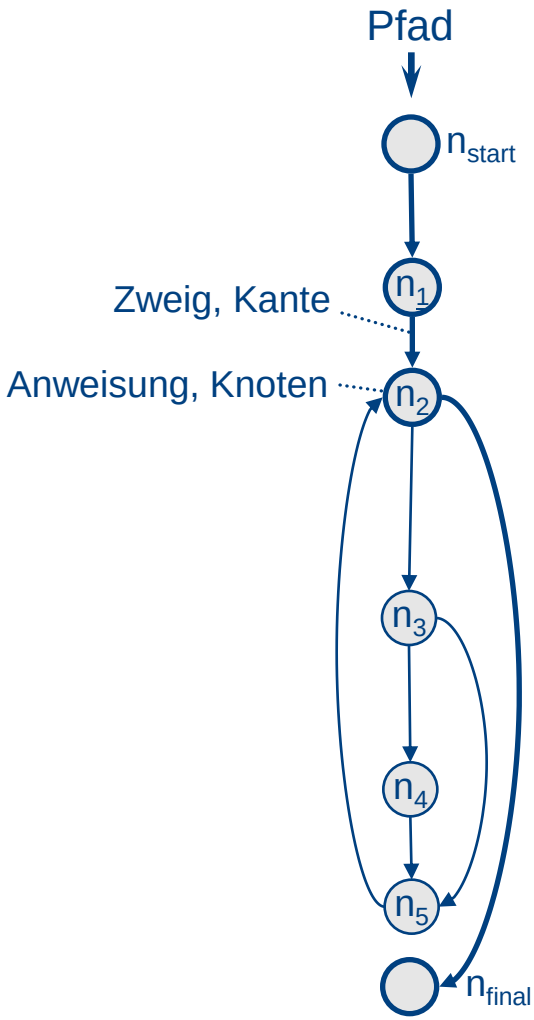



```
void ZaehleZchn(int& VokalAnzahl,
                int& Gesamtzahl)
{
char Zchn;

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;
}
}
```

Kontrollflussdiagramm zur Operation ZaehleZchn



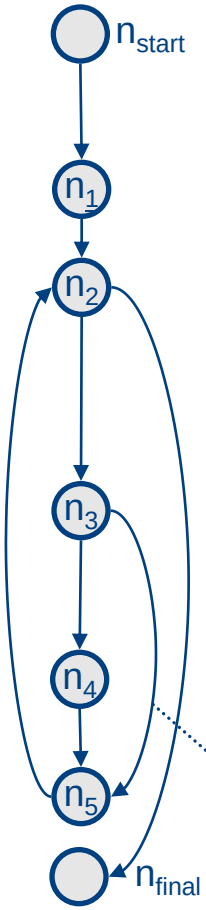

```
void ZaehleZchn (int& VokalAnzahl,
      int& Gesamtzahl)
{
   char Zchn;
   cin >> Zchn;
   while ((Zchn >= 'A') && (Zchn <= 'Z')
          && (Gesamtzahl < INT_MAX))
   {
      Gesamtzahl = Gesamtzahl+1;
      if ((Zchn == 'A') || (Zchn == 'E') ||
          (Zchn == 'I') || (Zchn == 'O') ||
                (Zchn == 'U'))
      {
         VokalAnzahl = VokalAnzahl + 1;
      }
      cin >> Zchn;
   }
}
```

- Der Anweisungsüberdeckungstest ist die einfachste kontrollflussorientierte Testtechnik. Man bezeichnet ihn abkürzend auch als C_0-Test. Im Englischen heißt er *statement coverage test*
- Das Ziel der Anweisungsüberdeckung ist die mindestens einmalige Ausführung aller Anweisungen des zu testenden Programms, also die Ausführung aller Knoten (Kreise) des Kontrollflussgraphen
- Als Testmaß wird der erreichte Anweisungsüberdeckungsgrad definiert. Er ist das Verhältnis der ausgeführten Anweisungen zu der Gesamtzahl der im Prüfling vorhandenen Anweisungen

$$C_{Anweisung} = \frac{Anzahl\ der\ ausgeführten\ Anweisungen}{Anzahl\ der\ Anweisungen}$$

- Sind alle Anweisungen des zu testenden Moduls durch die eingegebenen Testdaten mindestens einmal ausgeführt worden, so ist eine vollständige Anweisungsüberdeckung erreicht

- Der Anweisungsüberdeckungstest verlangt die Ausführung aller Knoten des Kontrollflussgraphen. Von den Testfällen wird verlangt, dass die entsprechenden Programmpfade alle Knoten des Kontrollflussgraphen enthalten
- Testfall
  Aufruf von ZaehleZchn mit: Gesamtzahl = 0
  Eingelesene Zeichen: 'A', '1'
  Durchlaufener Pfad: (n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_{final})
- Der Testpfad enthält alle Knoten. Er enthält aber nicht alle Kanten des Kontrollflussgraphen. Die Kante (n_3, n_5) ist nicht enthalten




```
void ZaehleZchn (int& VokalAnzahl,
      int& 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;
   }
}
```

Zweig (n_3,n_5) wird nicht notwendig ausgeführt

- Der Anweisungsüberdeckungstest gilt als zu schwaches Kriterium für eine sinnvolle Testdurchführung. Er besitzt eine untergeordnete praktische Bedeutung
- Der Standard RTCA DO-178B für Softwareanwendungen in der Luftfahrt fordert den Anweisungsüberdeckungstest für Software ab Stufe C (dritthöchste Stufe). Derartige Software kann im Falle eines Fehlverhaltens einen bedeutenden Ausfall *(major failure condition)* verursachen

- Das Ziel des Zweigüberdeckungstests ist die Ausführung aller Zweige des zu testenden Programms. Das verlangt den Durchlauf durch alle Kanten des Kontrollflussgraphen. Man bezeichnet ihn auch abkürzend als C_1-Test. Im Englischen sagt man *branch coverage test*
- Der Zweigüberdeckungstest ist eine strengere Testtechnik als der Anweisungsüberdeckungstest. Der Anweisungsüberdeckungstest ist im Zweigüberdeckungstest vollständig enthalten. Man sagt auch: Der Zweigüberdeckungstest **subsumiert** den Anweisungsüberdeckungstest
- Der Zweigüberdeckungstest gilt allgemein als das Minimalkriterium im Bereich des kontrollflussorientierten Testens
- Der Standard RTCA DO-178B für Softwareanwendungen im Bereich der Luftfahrt schreibt einen Zweigüberdeckungstest für Software ab Stufe B (zweithöchste Stufe) vor

- Beispiel

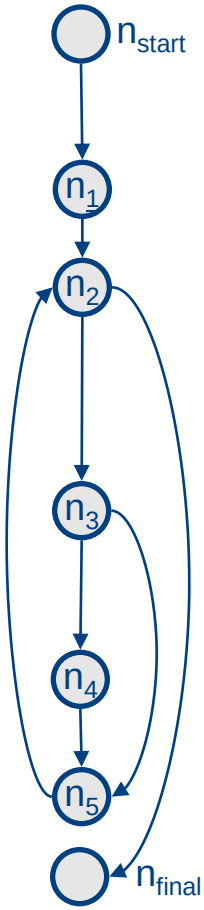
  Der Zweigüberdeckungstest fordert die Überdeckung aller Zweige eines Kontrollflussgraphen. Dies wird erreicht, falls jede Entscheidung des zu testenden Moduls mindestens einmal den Wahrheitswert falsch und wahr besessen hat

- Testfall

  Aufruf von ZaehleZchn mit: Gesamtzahl = 0
  Eingelesene Zeichen: „A“, „B“, „1“
  Durchlaufener Pfad: (n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{final})

- Der Testpfad enthält alle Kanten. Er enthält insbesondere die Kante (n_3,n_5), die durch den Anweisungsüberdeckungstest nicht notwendig ausgeführt wird. Der Zweigüberdeckungstest subsumiert den Anweisungsüberdeckungstest

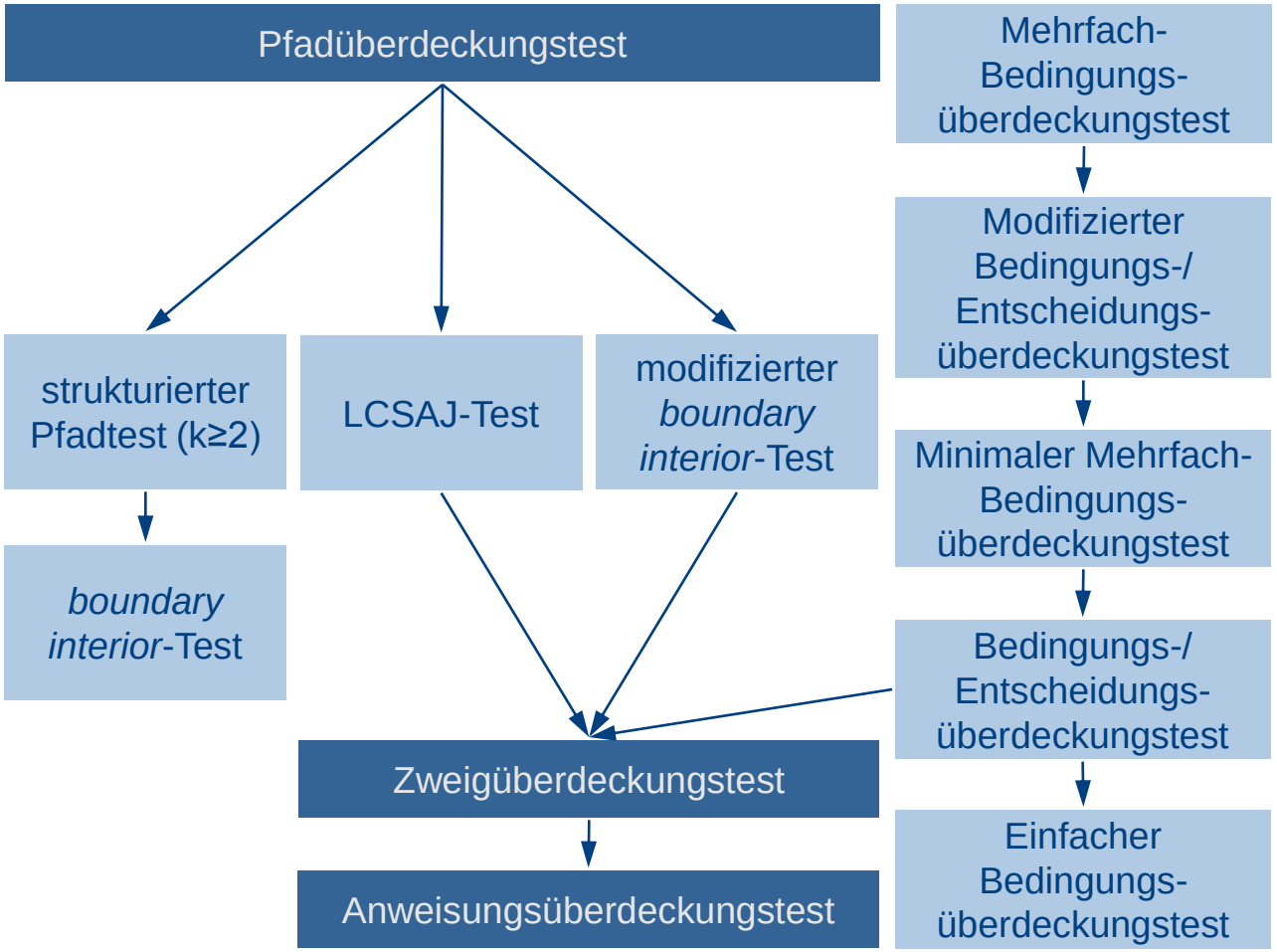
- Eine Programmausführung verursacht einen Durchlauf durch einen Programmpfad, der mehrere Zweige und Anweisungen enthält
- Frage: Wie kann dies in ein Testverfahren eingebracht werden?



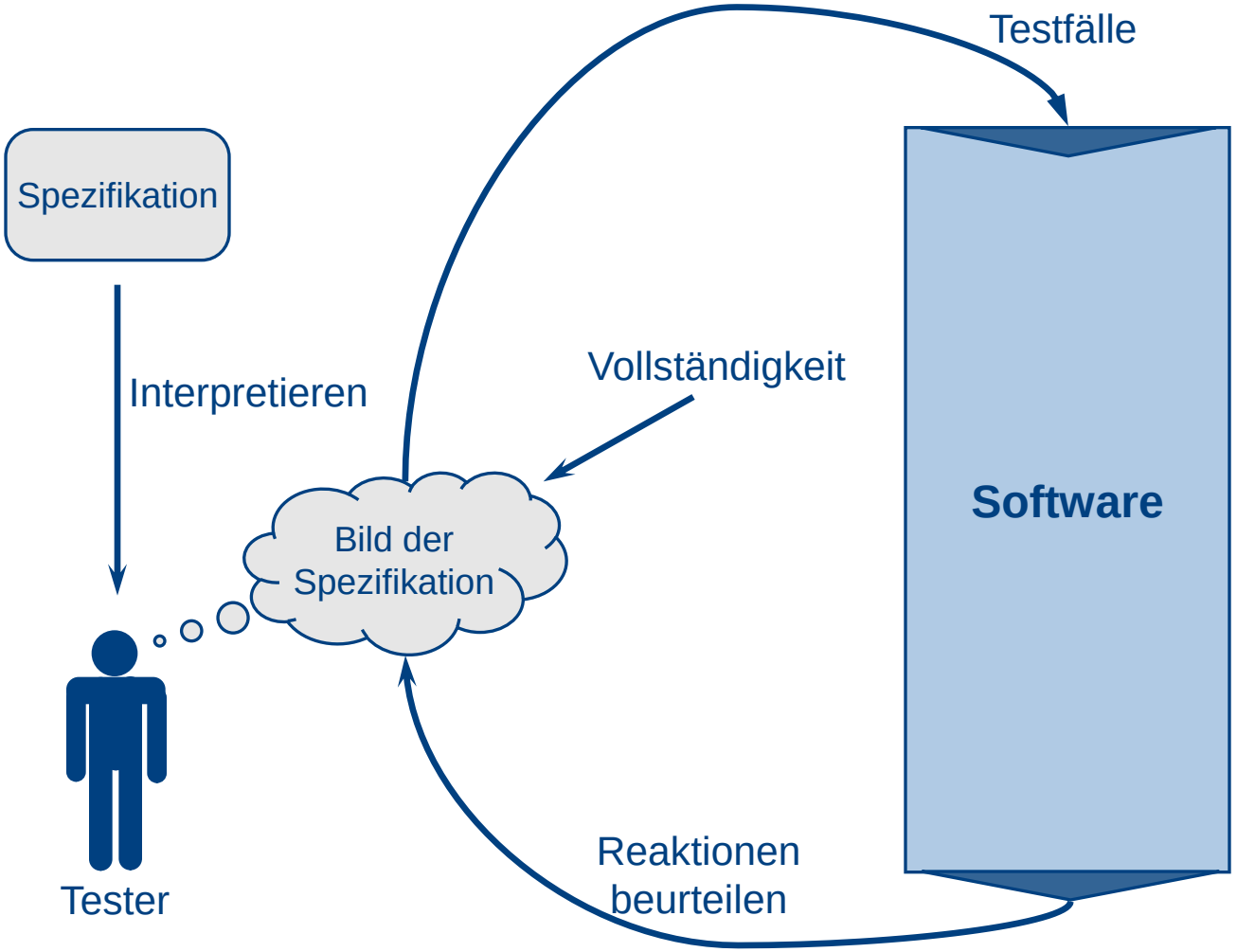

```
void ZaehleZchn (int& VokalAnzahl,
      int& Gesamtzahl)
{
   char Zchn;
   cin >> Zchn;
   while ((Zchn >= 'A') && (Zchn <= 'Z')
            && (Gesamtzahl < INT_MAX))
   {
      Gesamtzahl = Gesamtzahl+1;
      if ((Zchn == 'A') || (Zchn == 'E') ||
          (Zchn == 'I') || (Zchn == 'O') ||
                (Zchn == 'U'))
      {
          VokalAnzahl = VokalAnzahl + 1;
      }
      cin >> Zchn;
   }
}
```

- Eine vollständige Pfadüberdeckung fordert die Ausführung aller unterschiedlichen Pfade des zu testenden Programms
  - Ein Pfad p ist eine Sequenz von Knoten (i, n_1,.., n_m, j) des Kontrollflussgraphen mit dem Startknoten i und dem Endknoten j
- Nachteile
  - Der Pfadüberdeckungstest ist für reale Programme in der Regel nicht durchführbar, da sie eine unendlich hohe Anzahl von Pfaden besitzen können. Unter der Annahme, dass der größte Wert einer Variablen vom Typ INTEGER 32767 beträgt, erhält man für die Operation *ZaehleZchn* die unvorstellbar hohe Anzahl von 2^{32768}-1 Testpfaden (etwa $1{,}41 \cdot 10^{9864}$ Pfade). Die erforderliche Testdauer bei einem Tag und Nacht pausenlos durchlaufenden Test und einer zugrundegelegten Testintensität von 1000 Pfaden pro Sekunde würde $4{,}5 \cdot 10^{9853}$ Jahre dauern. Zum Vergleich: Das Alter der Erde wird mit etwas mehr als $4{,}5 \cdot 10^9$ Jahren angegeben. Ein vollständiger Pfadüberdeckungstest der Operation *ZaehleZchn* ist daher absolut ausgeschlossen
  - Oft ist ein Bruchteil der anhand des Kontrollflussgraphen konstruierbaren Pfade nicht ausführbar

Subsumptionsrelationen der kontrollflussorientierten Tests



Funktionsorientierter Test
Arbeitsweise



Funktionsorientierter Test

- Beurteilung der Adäquanz und der Vollständigkeit des Tests, sowie Herleitung der Testdaten und Beurteilung der Ausgaben anhand der Modulspezifikation
  - Vorteil: Vollständigkeit der Umsetzung der Spezifikation wird geprüft und Testfälle sind an der Soll-Funktionalität orientiert
  - Nachteil: Struktur der Implementation wird nicht berücksichtigt
- Hier vorgestellte Techniken
  - Funktionale Äquivalenzklassenbildung
  - Zustandsautomatenorientierter Test
  - Syntaxtest
  - Transaktionsflussbasierter Test
  - Test auf Basis von Entscheidungstabellen und Entscheidungsbäumen

Funktionsorientierter Test
Funktionale Äquivalenzklassenanalyse

- Bildung von Äquivalenzklassen der Eingabewerte auf der Basis der funktionalen Eigenschaften des Programms oder besser seiner funktionalen Spezifikation
- Werte aus einer Äquivalenzklasse
  - Verursachen ein identisches funktionales Verhalten und
  - Testen eine identische spezifizierte Programmfunktion
- Die Bildung der Äquivalenzklassen anhand der Spezifikation stellt sicher, dass alle spezifizierten Programmfunktionen mit Werten aus der ihnen zugeordneten Äquivalenzklasse getestet werden
- Aus den Ausgabewertebereichen können ebenfalls Äquivalenz-klassen erstellt werden

- Beispiel
  - Ist für eine Eingabe ein Wertebereich vorgesehen, so stellt dieser Bereich eine gültige Äquivalenzklasse dar, die an ihrer unteren und oberen Grenze durch ungültige Äquivalenzklassen eingerahmt wird
  - Eingabebereich: $1 \leq \text{Wert} \leq 99$
  - Eine gültige Äquivalenzklasse: $1 \leq \text{Wert} \leq 99$
  - Zwei ungültige Äquivalenzklassen: $\text{Wert} < 1$, $\text{Wert} > 99$

- Die Äquivalenzklassen sind eindeutig zu nummerieren. Für die Erzeugung von Testfällen aus den Äquivalenzklassen sind zwei Regeln zu beachten
  - Die Testfälle für gültige Äquivalenzklassen werden durch Auswahl von Testdaten aus möglichst vielen gültigen Äquivalenzklassen gebildet
  - Die Testfälle für ungültige Äquivalenzklassen werden durch Auswahl eines Testdatums aus einer ungültigen Äquivalenzklasse gebildet. Es wird mit Werten kombiniert, die ausschließlich aus gültigen Äquivalenz-klassen entnommen sind
- Auswahl der konkreten Testdaten aus einer Äquivalenzklasse nach unterschiedlichen Kriterien
- Oft verwendet: Test der Äquivalenzklassengrenzen **(Grenzwertanalyse)**

- Beispiel
  - Ein Programm zur Lagerverwaltung einer Baustoffhandlung besitzt eine Eingabemöglichkeit für die Registrierung von Anlieferungen
  - Werden Holzbretter angeliefert, so wird die Holzart eingegeben
  - Das Programm kennt die Holzarten Eiche, Buche und Kiefer
  - Ferner wird die Länge in Zentimetern angegeben, die stets zwischen 100 und 500 liegt
  - Als gelieferte Anzahl kann ein Wert zwischen 1 und 9999 angegeben werden
  - Außerdem erhält die Lieferung eine Auftragsnummer
  - Jede Auftragsnummer für Holzlieferungen beginnt mit dem Buchstaben H

- Äquivalenzklassenaufstellung

Eingabe	gültige Äquivalenzklassen	ungültige Äquivalenzklassen
Holzart	1) Eiche 2) Buche 3) Kiefer	4) Alles andere, z.B. Stahl
Länge	5) 100 ≤ Länge ≤ 500	6) Länge < 100 7) Länge > 500
Anzahl	8) 1 ≤ Anzahl ≤ 9999	9) Anzahl < 1 10) Anzahl > 9999
Auftrags-nummer	11) Erstes Zeichen ist H	12) Erstes Zeichen ist nicht H

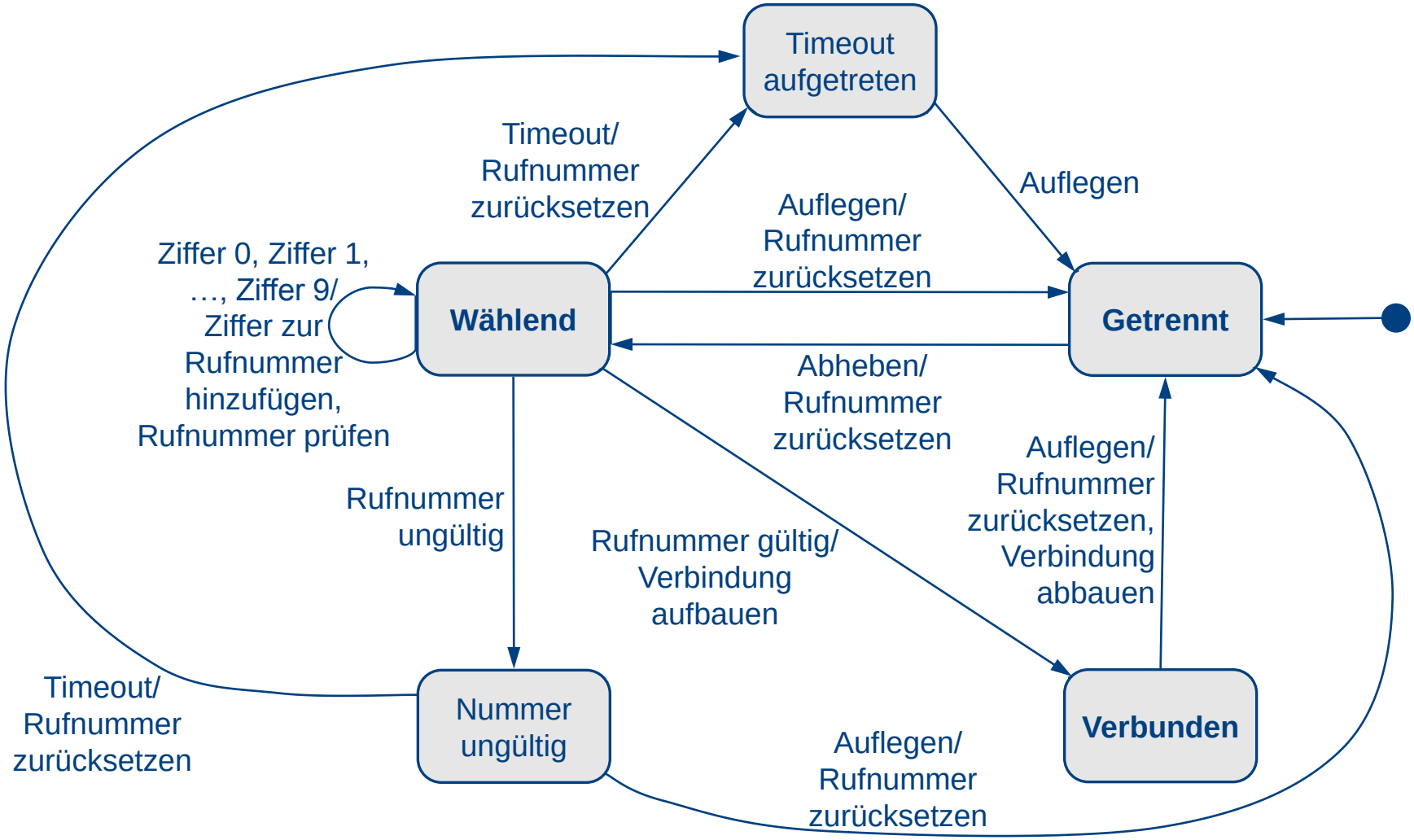
- Testfälle nach Äquivalenzklassenbildung mit Grenzwertanalyse

Testfall	1	2	3	4	5	6	7	8	9
(zusätzlich) getestete Äquivalenzklassen	1) 5) U 8) U 11)	2) 5) O 8) O	3)	4)	6) O	7) U	9) O	10) U	12)
Holzart	Eiche	Buche	Kiefer	Stahl	Buche	Buche	Buche	Buche	Buche
Länge	100	500	300	300	99	501	200	200	200
Anzahl	1	9999	100	100	100	100	0	10000	100
Auftrags-nummer	H1	H1	H1	H1	H1	H1	H1	H1	J2

- Beispiel zur Äquivalenzklassenbildung

- Die Klasse "Dreieck" enthält als Attribute die Längen der Dreiecksseiten Seite1, Seite2 und Seite3 als ganze Zahlen. Die Operation "Art ()" ermittelt die Dreiecksart auf Basis dieser Seitenlängen. Es werden folgende Fälle unterschieden
  - Kein Dreieck: Datenfehler der Seitenlängen
  - Gleichseitig
  - Rechtwinklig
  - Gleichschenklig
  - Ungleichseitig
- Die Art Rechtwinklig wird mit Vorrang ausgegeben, d.h. falls z.B. ein ungleichseitiges Dreieck rechtwinklig ist, so wird nicht Ungleichseitig, sondern Rechtwinklig ausgegeben

- Anwendungsgebiet: Software, die einen Zustandsautomaten realisiert
- Beispiel: Ausschnitt einer Modul-Spezifikation
  - Es ist ein Verbindungsaufbau und –abbau zwischen einem rufenden Teilnehmeranschluss und einem gerufenen Teilnehmeranschluss zu realisieren. Initial befindet sich die Verbindung im Zustand Getrennt. Bei aufgelegtem Hörer befindet sich die Software stets in diesem Zustand. Falls der Gesprächsaufbau begonnen wurde aber noch nicht beendet ist, so befindet sich die Software im Zustand Wählend. Falls der Gesprächsaufbau erfolgreich war, so befindet sich die Software im Zustand Verbunden
  - Ein erfolgreicher Gesprächsaufbau beginnt stets mit dem Abnehmen des Telefonhörers, gefolgt von der Wahl einer Ziffernfolge, die eine gültige Rufnummer darstellt. Ein Auflegen des Hörers führt stets zum vollständigen Abbruch des Gesprächs. Falls im Zustand Wählend ein Timeout auftritt, so kann nur durch Auflegen des Hörers in den Initialzustand zurückgekehrt werden


Timeout aufgetreten
Timeout/ Rufnummer zurücksetzen
Auflegen
Auflegen/ Rufnummer zurücksetzen
Ziffer 0, Ziffer 1, …, Ziffer 9/ Ziffer zur Rufnummer hinzufügen, Rufnummer prüfen
Wählend
Getrennt
Abheben/ Rufnummer zurücksetzen
Auflegen/ Rufnummer zurücksetzen, Verbindung abbauen
Rufnummer ungültig
Rufnummer gültig/ Verbindung aufbauen
Timeout/ Rufnummer zurücksetzen
Nummer ungültig
Verbunden
Auflegen/ Rufnummer zurücksetzen

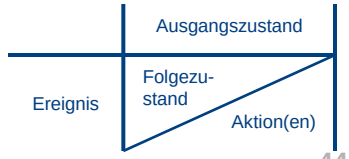
- Eine minimale Teststrategie ist die mindestens einmalige Abdeckung aller Zustände durch Testfälle. Besser ist das mindestens einmalige Durchlaufen aller Zustandsübergänge, das z. B. zu folgenden Testfällen führt
  - Getrennt, Abnehmen → Wählend, Auflegen → Getrennt
  - Getrennt, Abnehmen → Wählend, Timeout → Timeout aufgetreten, Auflegen → Getrennt
  - Getrennt, Abnehmen → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Rufnummer gültig → Verbunden, Auflegen → Getrennt
  - Getrennt, Abnehmen → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Rufnummer ungültig → Nummer ungültig, Auflegen → Getrennt
  - Getrennt, Abnehmen → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Ziffer 0 ... Ziffer 9 → Wählend, Rufnummer ungültig → Nummer ungültig, Timeout → Timeout aufgetreten, Auflegen → Getrennt
- Darüber hinaus ist es sinnvoll alle Ereignisse (Events) zu testen, falls Zustandsübergänge durch mehrere Events ausgelöst werden können. Dies ergibt eine Hierarchie von Testtechniken
  Alle Zustände ⊆ Alle Zustandsübergänge ⊆ Alle Events
- Wichtig: Test der Fehlerbehandlung nicht vergessen!

Funktionsorientierter Test
Zustandsbasierter Test

- Zustandsübergangstabelle

Ereignis \ Zustand	Getrennt	Wählend	Verbunden	Nummer ungültig	Timeout aufgetreten
Abheben	Wählend / Rufnr. zurücksetzen				
Auflegen		Getrennt / Rufnr. zurücksetzen	Getrennt / Rufnr. zurücksetzen, Verbind. abbauen	Getrennt / Rufnr. zurücksetzen	Getrennt / --
Ziffer 0		Wählend / Ziffer z. Rufnr. hinzuf., Rufnr. prüfen			
Ziffer 9		Wählend / Ziffer z. Rufnr. hinzuf., Rufnr. prüfen			
Timeout		Timeout aufgetreten / Rufnr. zurücksetzen		Timeout aufgetreten / Rufnr. zurücksetzen	
Rufnummer gültig		Verbunden / Verbind. aufbauen			
Rufnummer ungültig		Nummer Ungültig / --			

Legende:

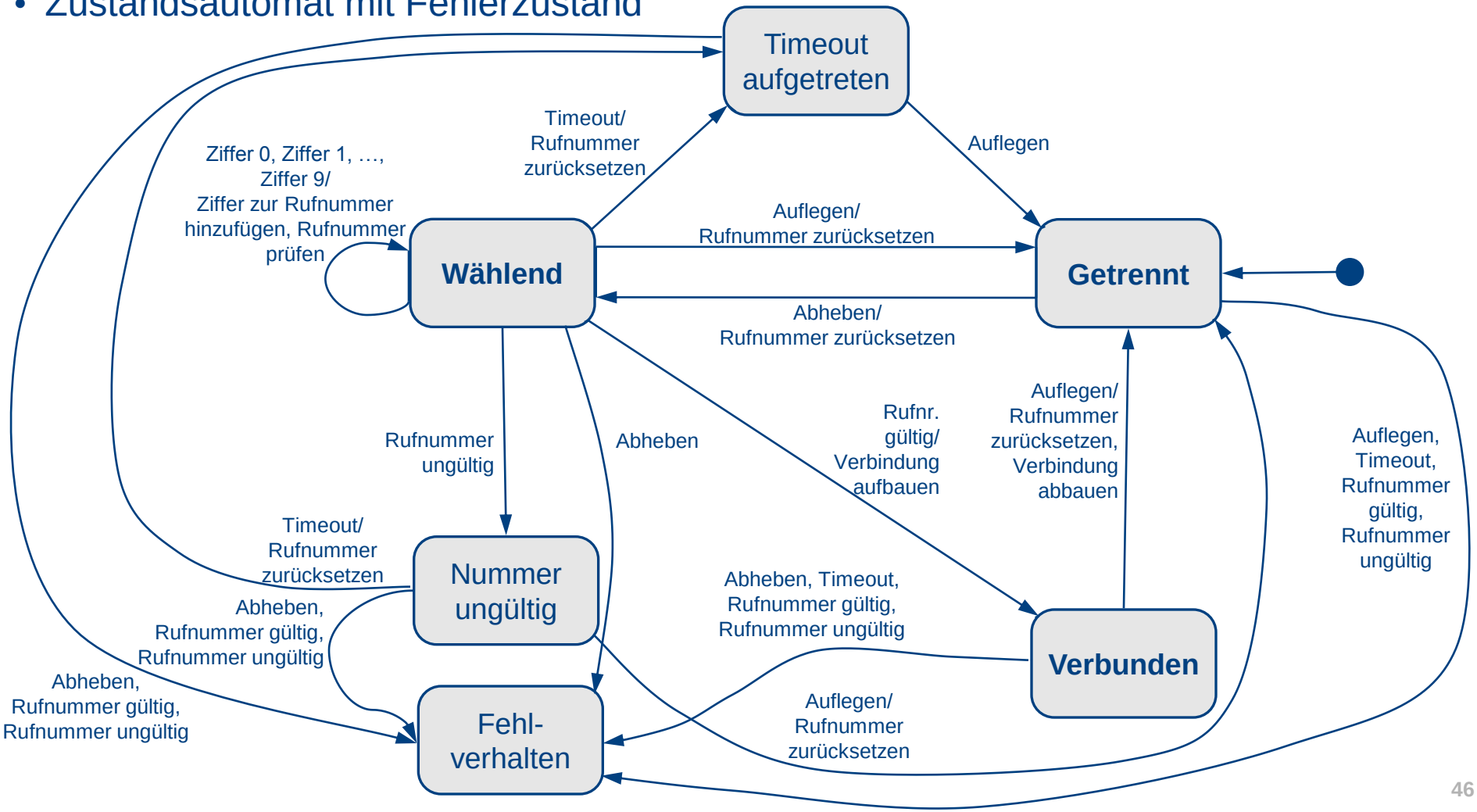


Funktionsorientierter Test
Zustandsbasierter Test

- Zustandsübergangstabelle mit Fehlerzustand

Ereignis \ Zustand	Getrennt	Wählend	Verbunden	Nummer ungültig	Timeout aufgetreten
Abheben	Wählend / Rufnr. zurücksetzen	Fehlverhalten / --	Fehlverhalten / --	Fehlver-halten / --	Fehlver-halten / --
Auflegen	Fehlver-halten / --	Getrennt / Rufnr. zurücksetzen	Getrennt / Rufnr. zurücksetzen Verbind. abbauen	Getrennt / Rufnr. zurücksetzen	Getrennt / --
Ziffer 0	Getrennt / --	Wählend / Ziffer z. Rufnr. hinzuf., Rufnr. prüfen	Verbunden / --	Nummer ungültig / --	Timeout aufgetre-ten / --
Ziffer 9	Getrennt / --	Wählend / Ziffer z. Rufnr. hinzuf., Rufnr. prüfen	Verbunden / --	Nummer ungültig / --	Timeout aufgetre-ten / --
Timeout	Fehlverhalten / --	Timeout aufgetreten / Rufnr. zurücksetzen	Fehlverhalten / --	Timeout aufgetre-ten / Rufnr. zurücksetzen	Timeout aufgetre-ten / --
Rufnummer gültig	Fehlverhalten / --	Verbunden / Verbind. aufbauen	Fehlverhalten / --	Fehlver-halten / --	Fehlver-halten / --
Rufnummer ungültig	Fehlverhalten / --	Nummer Ungültig / --	Fehlverhalten / --	Fehlver-halten / --	Fehlver-halten / --

- Zustandsautomat mit Fehlerzustand



Literatur

- Liggesmeyer P., Software-Qualität, Heidelberg: Spektrum-Verlag (2. Auflage) 2009