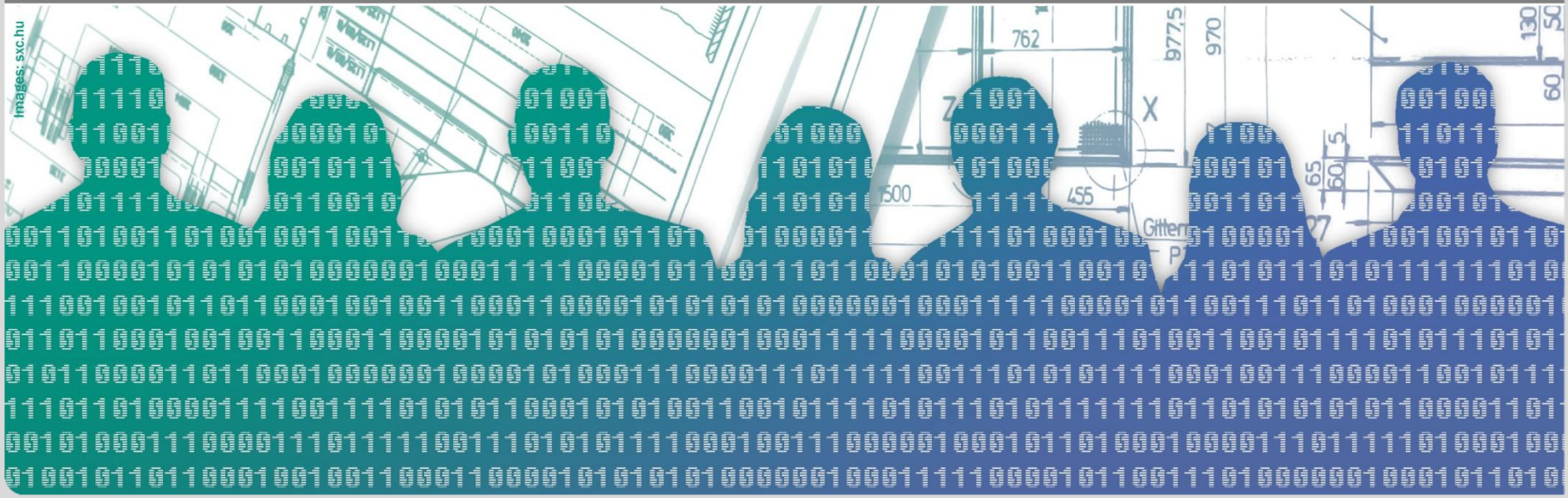
Kapitel 5.1 - Testen und Prüfen

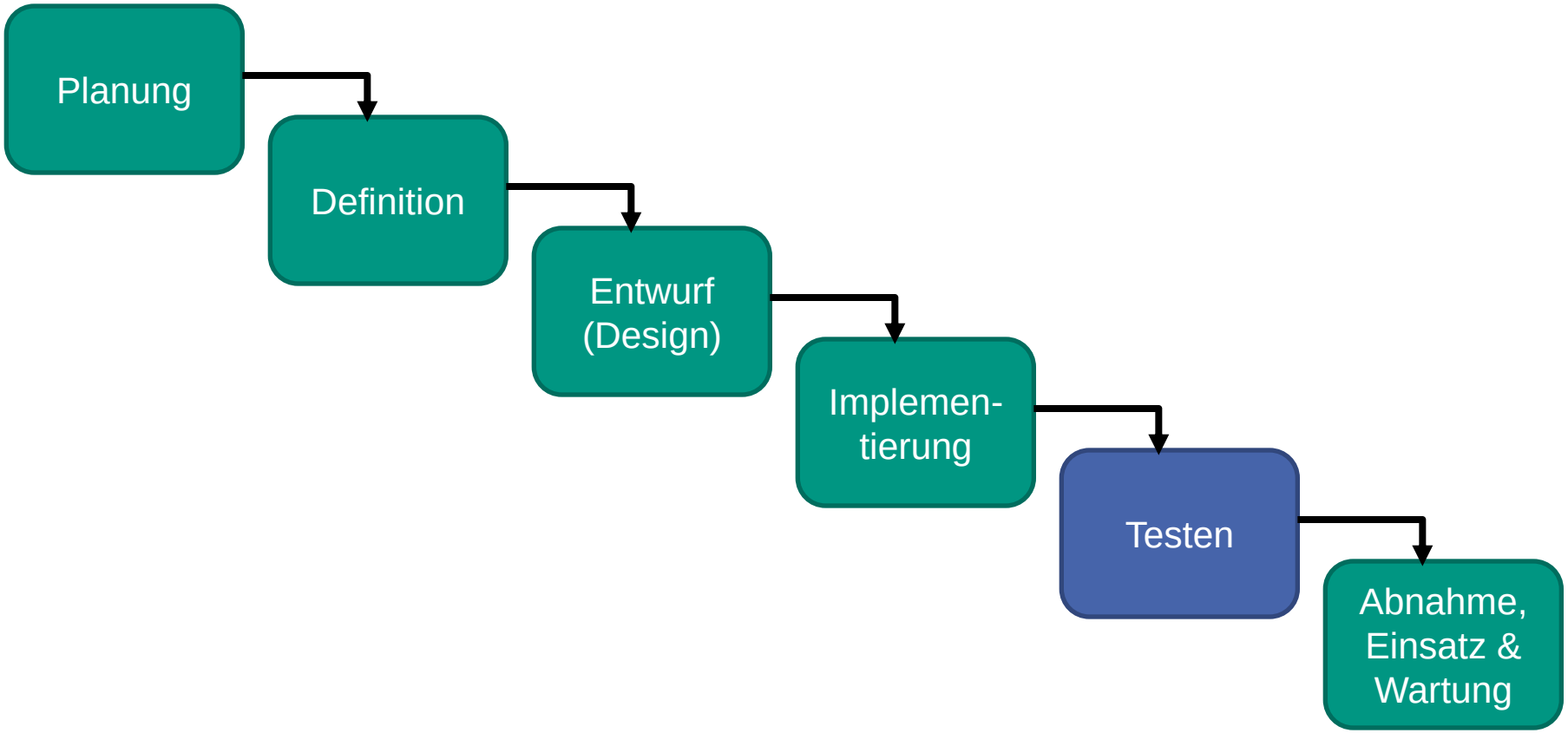
SWT I – Sommersemester 2010

Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik



Wo sind wir gerade?



Literatur

- Diese Vorlesung orientiert sich an Kapitel 11 aus

  B. Bruegge, A.H. Dutoit, **Object-Oriented Software Engineering: Using UML, Patterns and Java**, Pearson Prentice Hall, 2004 und 2010.

Motivation

- Software-Artefakte enthalten immer Fehler
- Je später ein Fehler gefunden wird, desto teurer ist es, ihn zu beheben
- Ziel ist es, Fehler möglichst früh zu finden

Fehleraufdeckung ist das Ziel der Testverfahren

„Testing shows the presence of bugs, not their absence.“

(Edsger W. Dijkstra)

Fehleraufdeckung ist das Ziel der Testverfahren

- Vollständiges Testen aller Kombinationen aller Eingabewerte ist außer bei trivialen Programmen nicht möglich (astronomische Anzahl von Testfällen).
- Korrektheit ist nur mit formalem Korrektheitsbeweis möglich (Korrespondenz von Spezifikation und Programm); dies ist heute nur für kleine Programme möglich.
- Zentrale Frage: Wann kann man mit dem Testen aufhören, nach Fehlern zu suchen? (Testvollständigkeitskriterien)

Fehleraufdeckung ist das Ziel der Testverfahren

- Vollständiges Testen aller Kombinationen aller Eingabewerte ist außer bei trivialen Programmen nicht möglich (astronomische Anzahl von Testfällen)
- Korrek
  möglic ramm);
  dies is
- Zentra fhören,
  nach F en)

Unterscheide

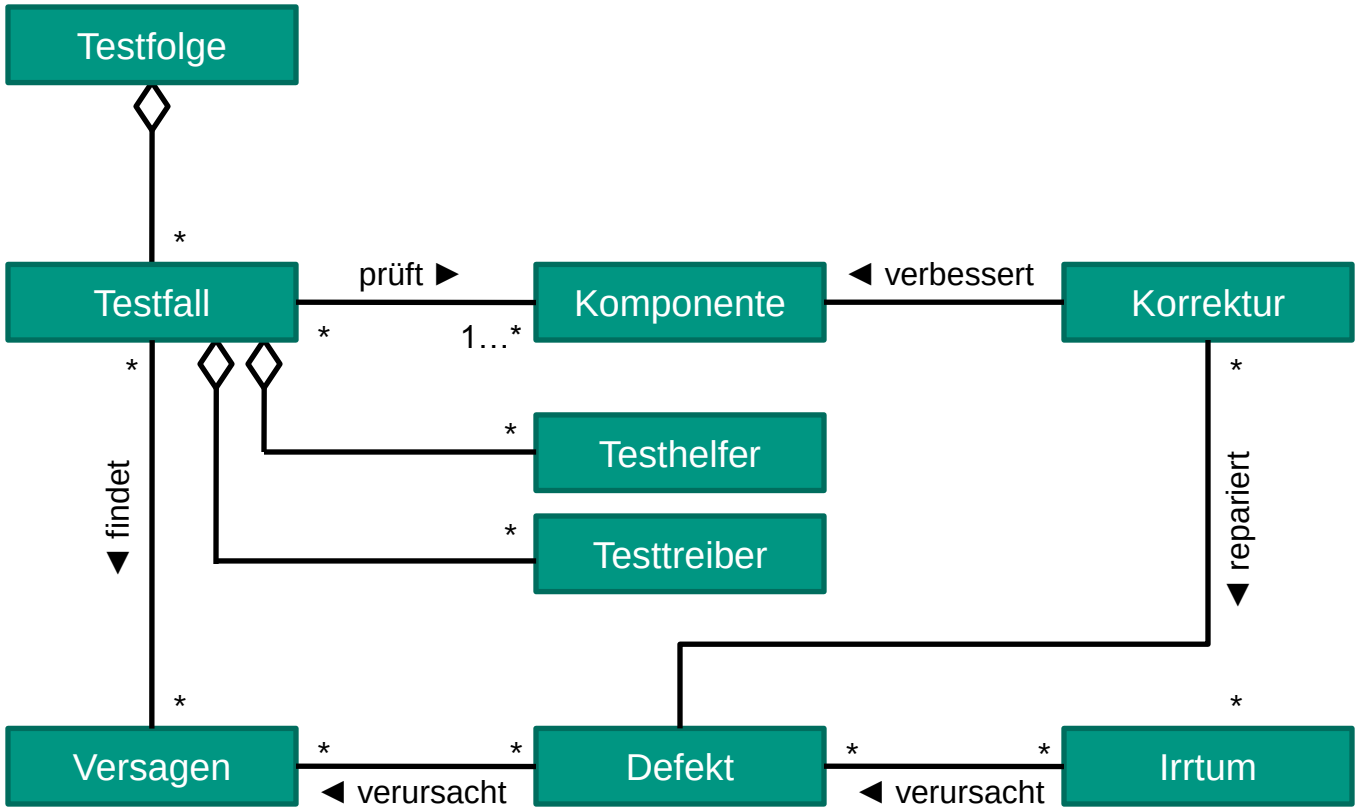
- Testende Verfahren → Fehler erkennen
- Verifizierende Verfahren → Korrektheit beweisen
- Analysierende Verfahren → Eigenschaften einer Systemkomponente bestimmen

Es gibt 3 Arten von Fehlern…

- Ein **Versagen** oder **Ausfall** (engl. *failure*, *fault*) ist die Abweichung des Verhaltens der Software von der Spezifikation (ein Ereignis).
- Ein **Defekt** (engl. *defect*) ist ein Mangel in einem Softwareprodukt, der zu einem Versagen führen kann (ein Zustand).
  Man sagt, dass sich ein Defekt in einem Versagen oder Ausfall manifestiert.
- Ein **Irrtum** oder **Herstellungsfehler** (engl. *mistake*) ist eine menschliche Aktion, die einen Defekt verursacht (ein Vorgang).

Fehler (engl. *error*)

Zusammenhang der Fehlerarten



Arten von Testhelfern

- Ein **Stummel** (engl. *stub*) ist ein nur rudimentär implementierter Teil der Software und dient als Platzhalter für noch nicht umgesetzte Funktionalität
- Eine **Attrappe** (engl. *dummy*) simuliert die Implementierung zu Testzwecken
- Eine **Nachahmung** (engl. *mock*) ist eine Attrappe mit zusätzlicher Funktionalität, wie bspw. das Einstellen der Reaktion der Nachahmung auf bestimmte Eingaben oder das Überprüfen des Verhaltens des „Klienten“

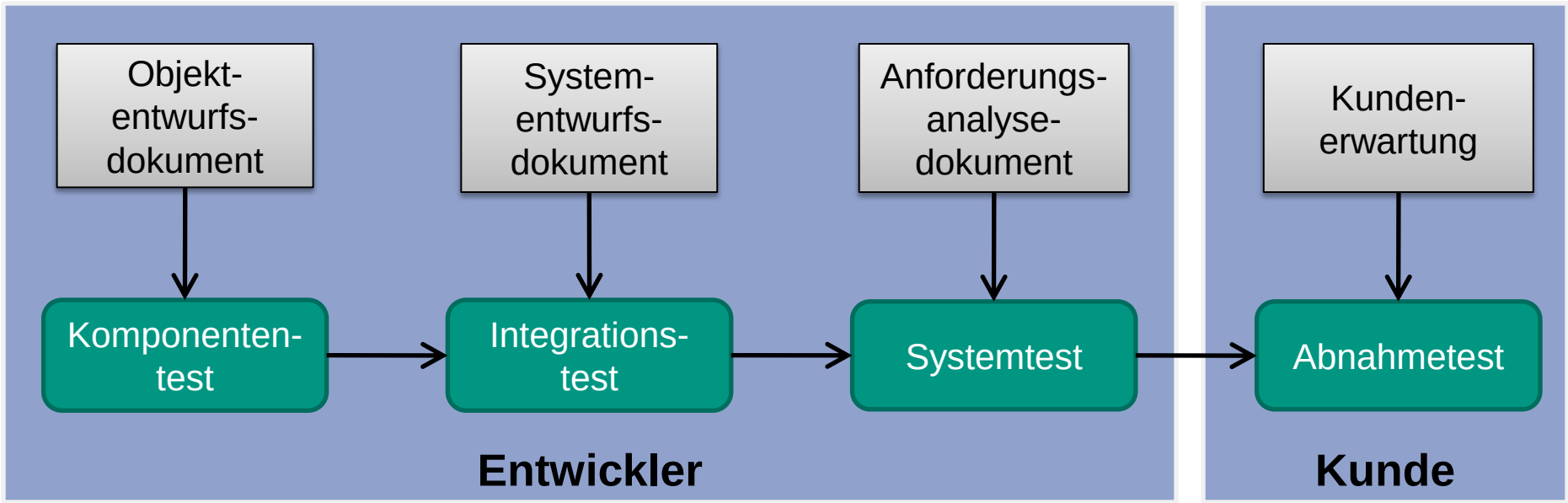
Fehlerklassen

- Anforderungsfehler (Defekt im Pflichtenheft)
  - Inkorrekte Angaben der Benutzerwünsche
  - Unvollständige Angaben über funktionale Anforderungen, Leistungsanforderungen, etc.
  - Inkonsistenz verschiedener Anforderungen
  - Undurchführbarkeit
- Entwurfsfehler (Defekt in der Spezifikation)
  - Unvollständige oder fehlerhafte Umsetzung der Anforderung
  - Inkonsistenz der Spezifikation oder des Entwurfs
  - Inkonsistenz zwischen Anforderung, Spezifikation und Entwurf
- Implementierungsfehler (Defekt im Programm)
  - Fehlerhafte Umsetzung der Spezifikation in ein Programm

Modul-/Softwaretestverfahren

- Ein **Softwaretest**, kurz Test, führt eine einzelne Software-Komponente oder eine Konfiguration von Softwarekomponenten unter bekannten Bedingungen (Eingaben und Ausführungsumgebung) aus und überprüft ihr Verhalten (Ausgaben und Reaktionen).
- Die zu überprüfende SW-Komponente oder Konfiguration wird **Testling**, **Prüfling** oder **Testobjekt** genannt.
- Ein **Testfall** besteht aus einem Satz von Daten für die Ausführung eines Teils oder des ganzen Testlings.
- Ein **Testtreiber** oder **Testrahmen** versorgt Testlinge mit Testfällen und stößt die Ausführung der Testlinge an (interaktiv oder selbsttätig).

Testphasen



Der **Komponenten-test** überprüft die Funktion eines Einzelmoduls durch Beobachtung der Verarbeitung von Testdaten.

Der **Integrationstest** überprüft schrittweise das fehlerfreie Zusammenwirken von bereits einzeln getesteten System-komponenten.

Der **Systemtest** ist der abschließende Test des Auftrag-nehmers in realer (bzw. realistischer) Umgebung ohne Kunden.

Der **Abnahmetest** ist der abschl. Test in realer Umgebung unter Beobachtung, Mitwirkung und/oder Federführung des Kunden beim Kunden.

Klassifikation testender Verfahren (1)

- Dynamische Verfahren

Testen

  - Strukturtests (white/glass box testing)
    - Kontrollflussorientierte Tests
    - Datenflussorientierte Tests
  - Funktionale Tests (*black box testing*)
  - Leistungstests (auch *black box*)

Prüfen

- Statische Verfahren
  - Manuelle Prüfmethoden (Inspektion, Review, Durchsichten, engl. *walkthrough*)
  - Prüfprogramme (statische Analyse von Programmen)

Klassifikation testender Verfahren (2)

- Dynamische Verfahren
  - Das übersetzte, ausführbare Programm wird mit bestimmten Testfällen versehen und ausgeführt
  - Das Programm wird in der realen (realistischen) Umgebung getestet
  - Das ist ein Stichprobenverfahren; Korrektheit des Programms wird **nicht bewiesen!**
- Statische Verfahren
  - Das Programm (die Komponente) wird nicht ausgeführt, sondern der Quellcode analysiert

Klassifikation testender Verfahren (2)

- Dynamische Verfahren
  - Das übersetzte, ausführbare Programm wird mit bestimmten Testfällen versehen und ausgeführt
  - Das Programm wird in der realen (realistischen) Umgebung getestet
  - Das ist ein Stichprobenverfahren; Korrektheit des Programms wird **nicht bewiesen!**
- Statische Verfahren
  - Das Programm (die Komponente) wird nicht ausgeführt, sondern Sourcecode analysiert

White Box Testen: Bestimmen der Werte **mit Kenntnis** von Kontroll- und/oder Datenfluss

Black Box Testen: Bestimmen der Werte **ohne Kenntnis** von Kontroll- und/oder Datenfluss aus der Spezifikation heraus

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest						
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Kontrollflussorientierte (KFO) Testverfahren

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
  - Vollständig
  - Strukturiert
- Bedingungsüberdeckung
  - Einfach
  - Mehrfach
  - Minimal mehrfach

Zuvor jedoch…

- Vollständigkeitskriterien für Testverfahren werden über „Kontrollflussgraphen“ definiert
  - Definition einer Zwischensprache
  - Definition der Transformation in die Zwischensprache
  - Definition des Kontrollflussgraphen
- Dann: Definition der Verfahren und ihrer entsprechenden Überdeckung

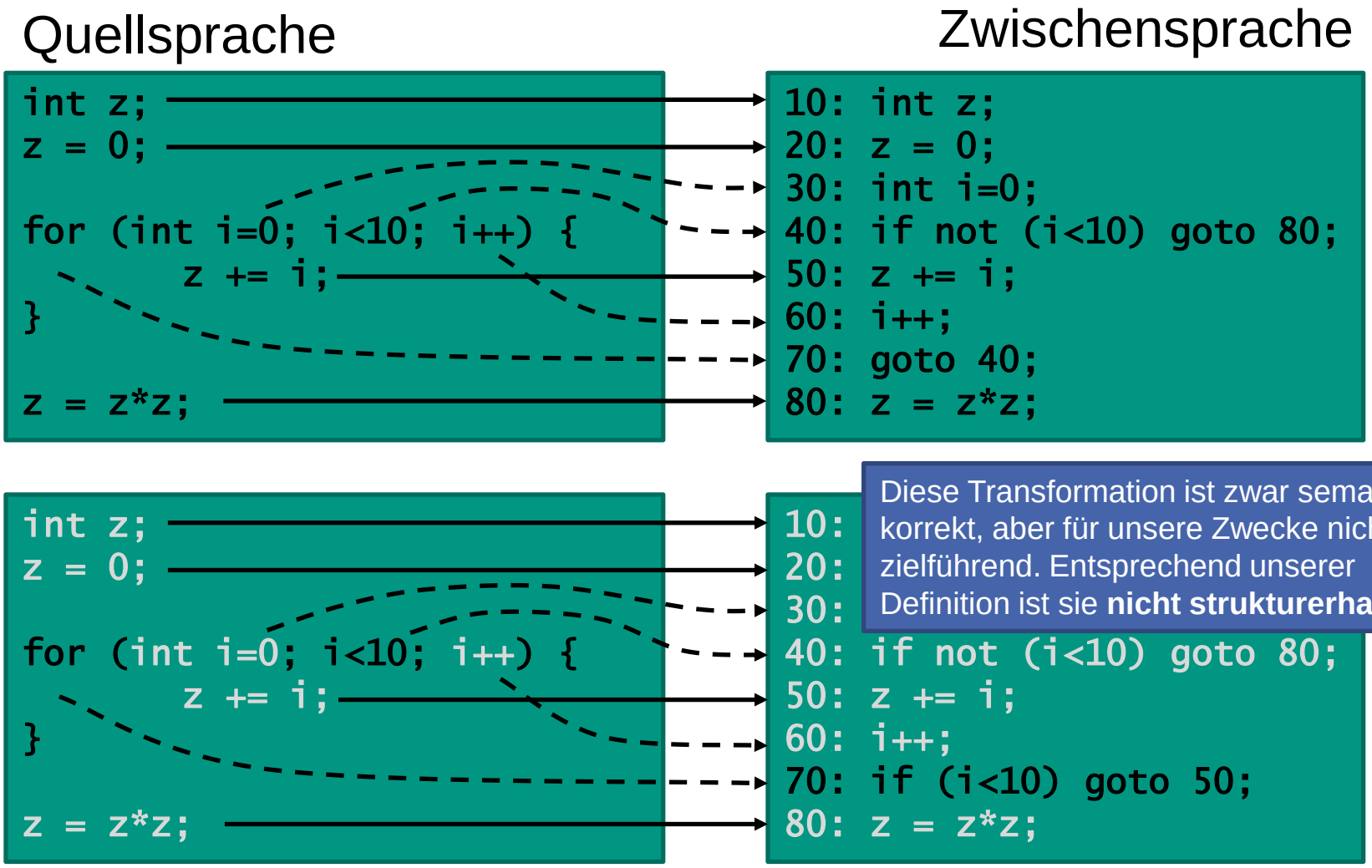
Definition: Zwischensprache

- Wir definieren eine **Zwischensprache**, bestehend aus:
  - beliebigen Befehlen außer solchen, die die Ausführungsreihenfolge beeinflussen (bedingte Anweisungen Sprünge, Schleifen, etc.),
  - bedingten Sprungbefehlen zu beliebigen aber festen Stellen der Befehlsfolge und
  - unbedingten Sprungbefehlen zu beliebigen aber festen Stellen der Befehlsfolge (zur Vereinfachung),
  - einer beliebigen Anzahl von Variablen.
- Die Zwischensprache orientiert sich an dem, was man landläufig unter „Assembler-Code“ versteht.
- Die Realisierung (insbes. auch die Nomenklatur der Befehle) dieser Zwischensprache ist hier unerheblich.

Definition: Strukturerhaltende Transformation

- Wir sprechen von einer **strukturerhaltenden Transformation** einer Quellsprache (z.B. Java) in die Zwischensprache, wenn
  - (ausschließlich) die Befehle, die die Ausführungsreihenfolge beeinflussen, durch Befehlsfolgen der Zwischensprache ersetzt werden, wobei
    - die Ausführungsreihenfolge der anderen Befehle bei gleicher Parametrisierung gleich bleibt mit der in der Quellsprache!
  - alle anderen Befehle unverändert übernommen werden
  - Transformationen, bei denen Anweisungsfolgen oder bedingte Sprünge repliziert werden, vermieden werden (kein Ausrollen von Schleifen, keine Optimierungen.)

Beispieltransformation



Definition: Grundblock (GB)

- Ein **Grundblock** bezeichnet eine maximal lange Folge fortlaufender Anweisungen der Zwischensprache,
  - in die der Kontrollfluss nur am Anfang eintritt und
  - die außer am Ende keine Sprungbefehle enthält.

Beispiel

```
a = 10;
b = c / a;
if b > d goto Grundblock x;
m = 3 * b;
...
```

1 Grundblock

nächster Grundblock

Definition: Kontrollflussgraph (KFG)

- Ein **Kontrollflussgraph** eines Programms P ist ein gerichteter Graph G mit

$$G = (N, E, n_{start}, n_{stopp})$$

wobei

- N die Menge der Grundblöcke in P,
- $E \subseteq N \times N$ die Menge der Kanten, wobei die Kanten die Ausführungsreihenfolge von je zwei Grundblöcken angeben (sequentielle Ausführung oder Sprünge)
- n_{start} der Startblock und
- n_{stopp} der Stoppblock ist.

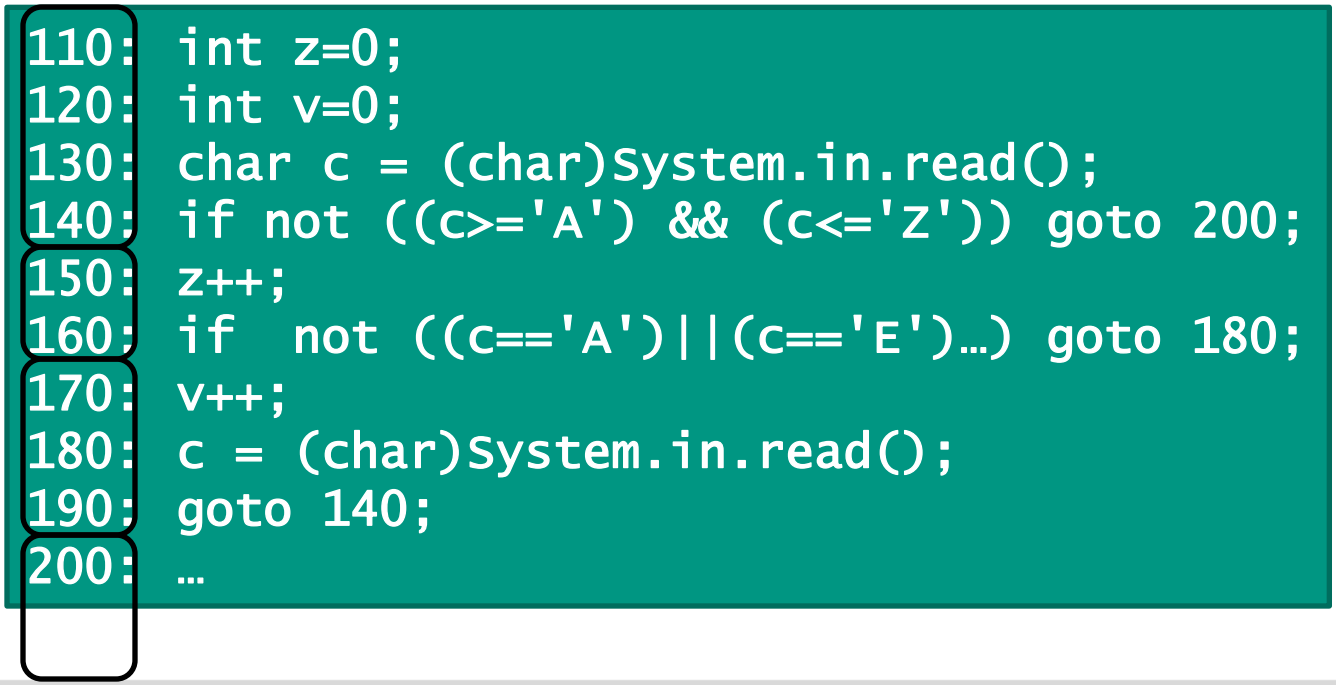
Kontrollflussgraph finden

Beispielprogramm

```
…
int z=0;
int v=0;
char c = (char)System.in.read();
while ((c>='A') && (c<='Z'))
{
  z++;
  if
  ((c=='A')||(c=='E')||(c=='I')||(c=='O')||(c=='U'))
  {
    v++;
  }
  c = (char)System.in.read();
}
…
```

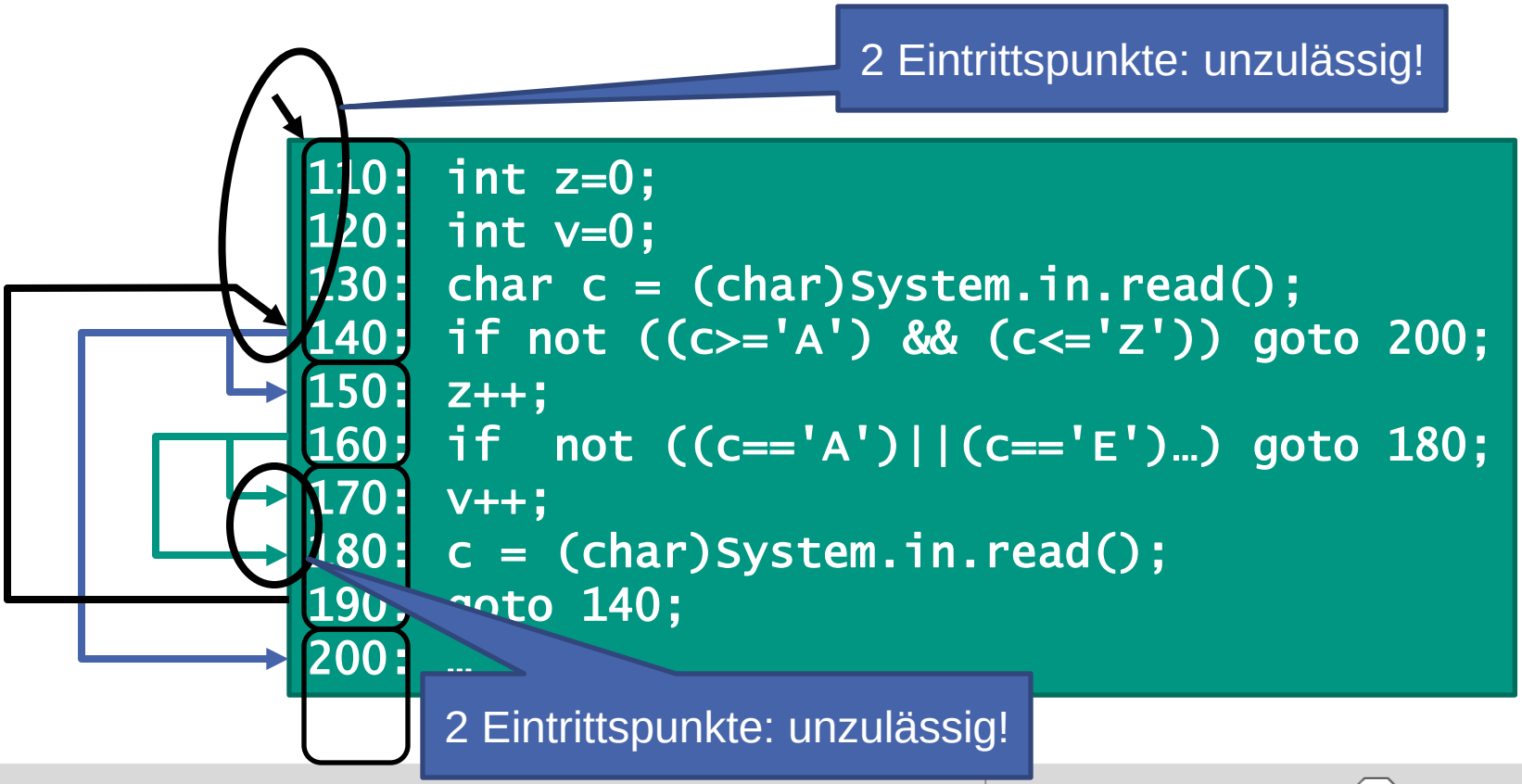
Kontrollflussgraph finden

1. Schritt: Transformiere in Zwischensprache
2. Schritt: Fasse alle Folgen, die mit einem Sprung enden, zu je einem Grundblock zusammen



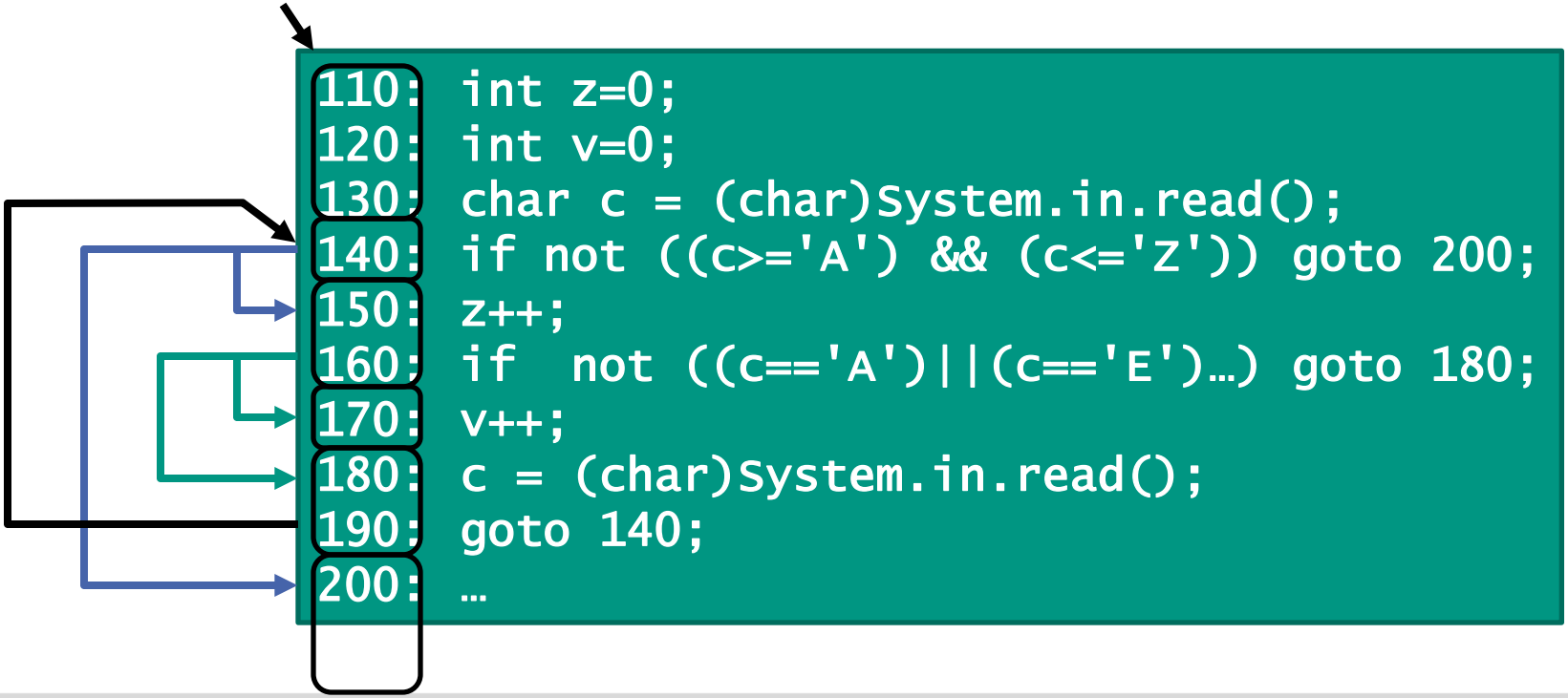
Kontrollflussgraph finden

3. Schritt: Prüfe, ob Eintritt nur am Anfang
4. Schritt: Teile ggf. auf

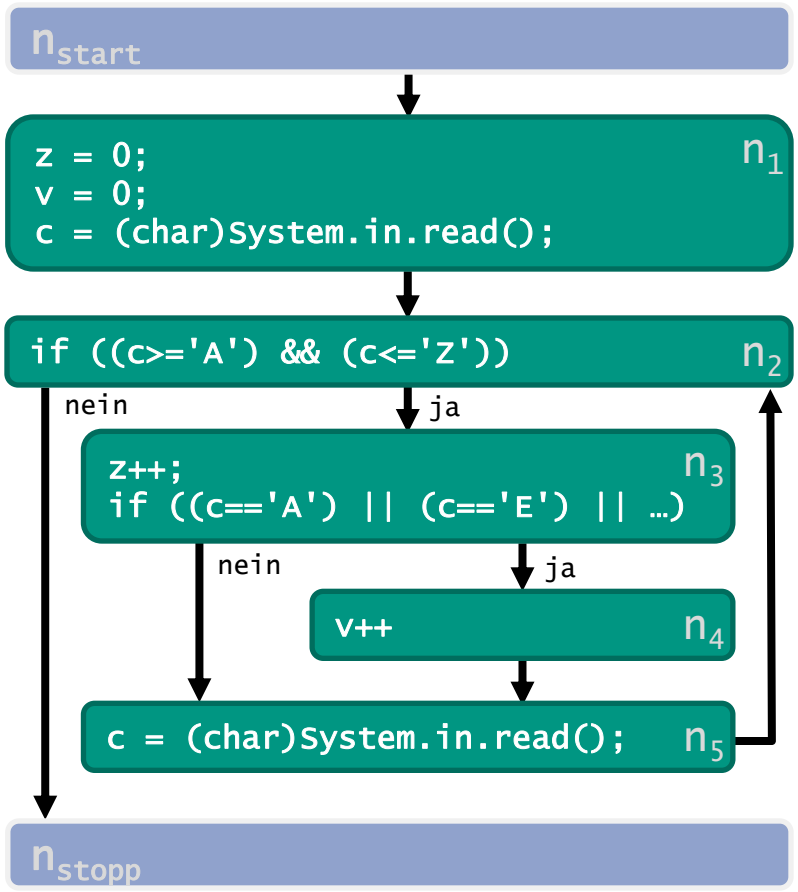


Kontrollflussgraph finden

3. Schritt: Prüfe, ob Eintritt nur am Anfang
4. Schritt: Teile ggf. auf

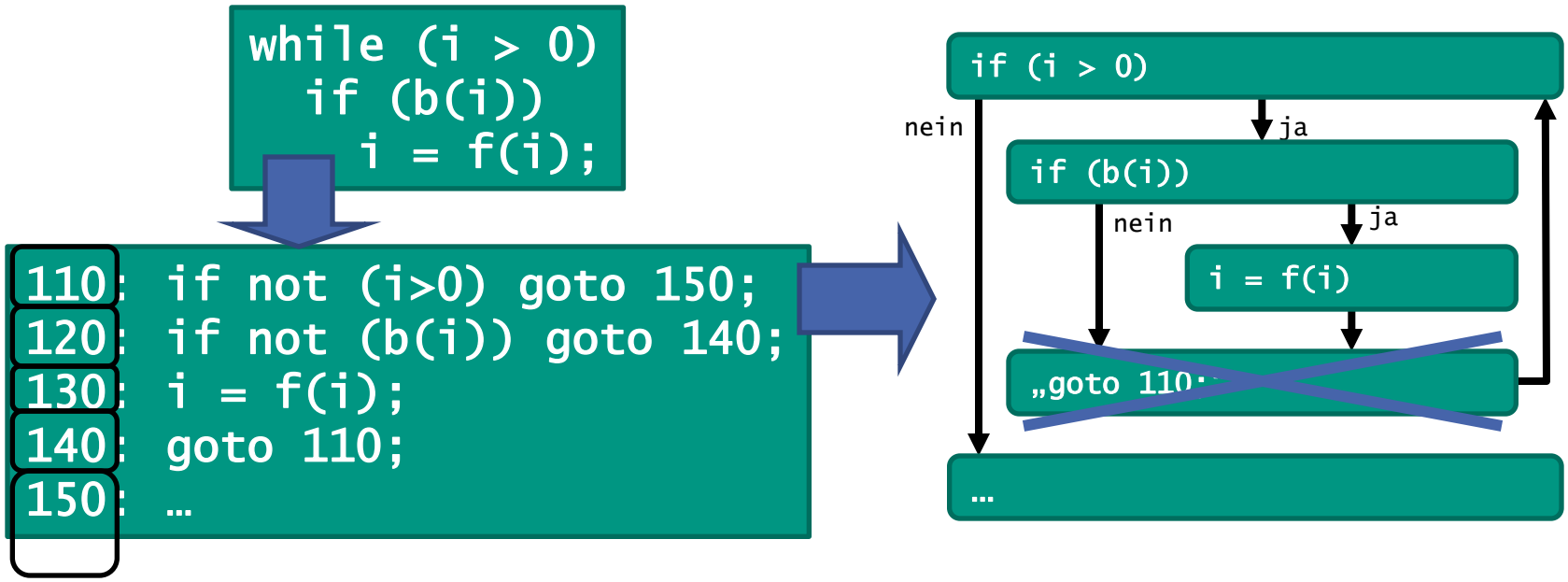


Beispielkontrollflussgraph



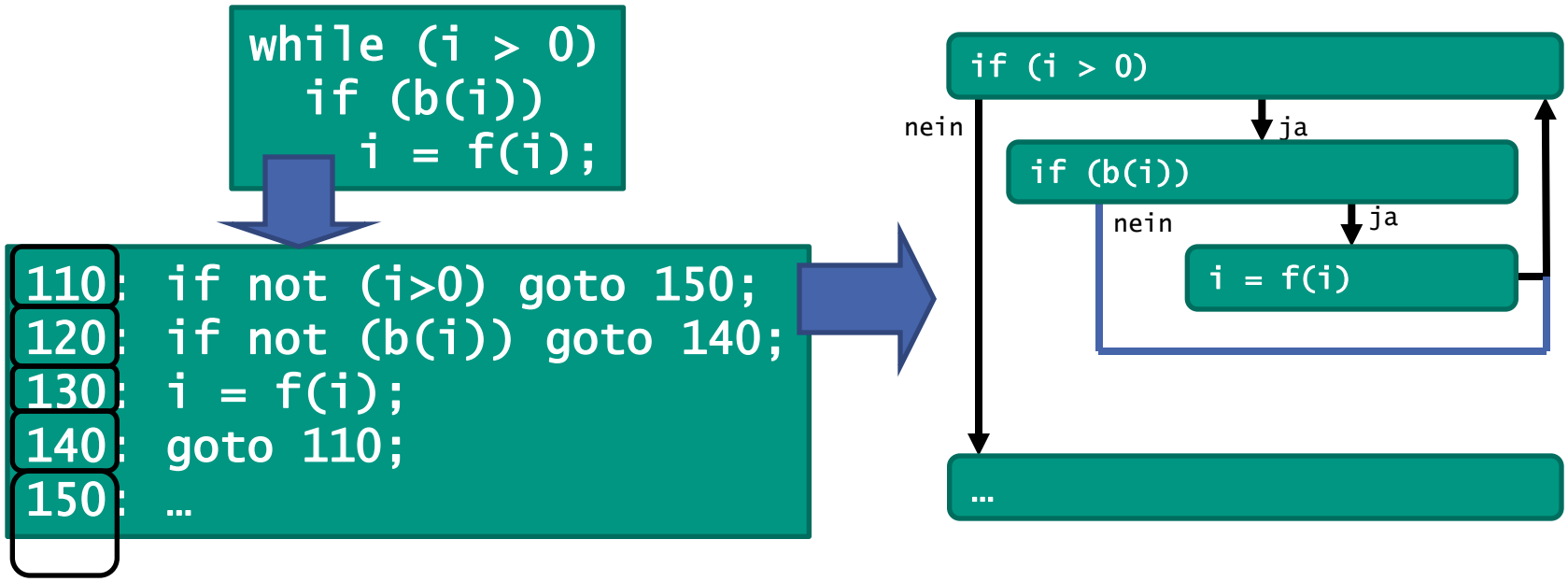
Kontrollflussgraph vereinfachen

- Eventuell:
  - Entsteht durch eine Aufteilung ein Grundblock, der nur noch einen unbedingten Sprungbefehl enthält, dann kann dieser entfernt werden. (Reduzierung der GB.)



Kontrollflussgraph vereinfachen

- Eventuell:
  - Entsteht durch eine Aufteilung ein Grundblock, der nur noch einen unbedingten Sprungbefehl enthält, dann kann dieser entfernt werden (Reduzierung der GB).



Definitionen: Zweig, vollständige Pfade

- Eine Kante e ∈ E in einem KFG G wird **Zweig** genannt. Zweige sind grundsätzlich gerichtet.

- Pfade im KFG die mit dem Startknoten n_{start} anfangen und beim Stoppknoten n_{stopp} aufhören heißen **vollständige Pfade**.

Definition: Anweisungsüberdeckung

- Die Teststrategie **Anweisungsüberdeckung** $C_{Anweisung}$ verlangt die Ausführung aller Grundblöcke des Programms P.
  - C steht für Coverage
  - Metrik, auch C_0 genannt

$$C_{Anweisung} = \frac{\text{Anzahl durchlaufener Anweisungen}}{\text{Anzahl aller Anweisungen}}$$

  - Nicht ausreichendes Testkriterium
  - Nicht ausführbare Programmteile können gefunden werden

Definition: Zweigüberdeckung

- Die **Zweigüberdeckung** C_{Zweig} verlangt das Traversieren aller Zweige im KFG.
  - Metrik, auch C_1 genannt

$$C_{Zweig} = \frac{\text{Anzahl traversierter Zweige}}{\text{Anzahl aller Zweige}}$$

  - Zweige, die nicht ausgeführt werden, können entdeckt werden
  - Weder Kombination von Zweigen (Pfade) noch komplexe Bedingungen berücksichtigt
  - Schleifen nicht ausreichend getestet
  - Fehlende Zweige nicht testbar

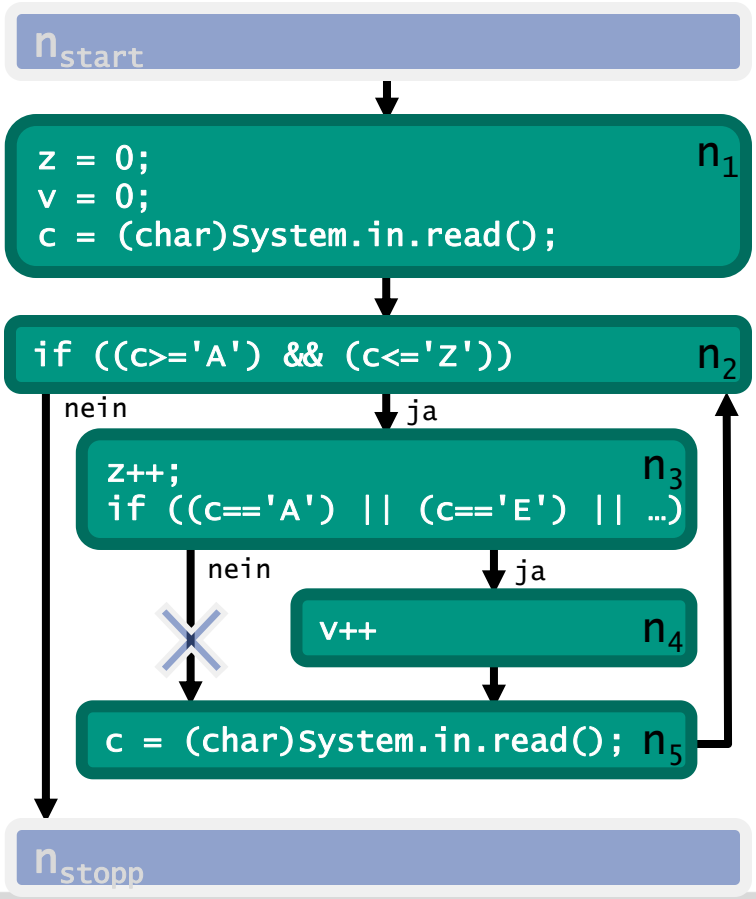
Anw.- vs. Zweigüberdeckung

Anweisungsüberdeckung, alle Knoten

Beispielfolge:

(n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_{stopp})

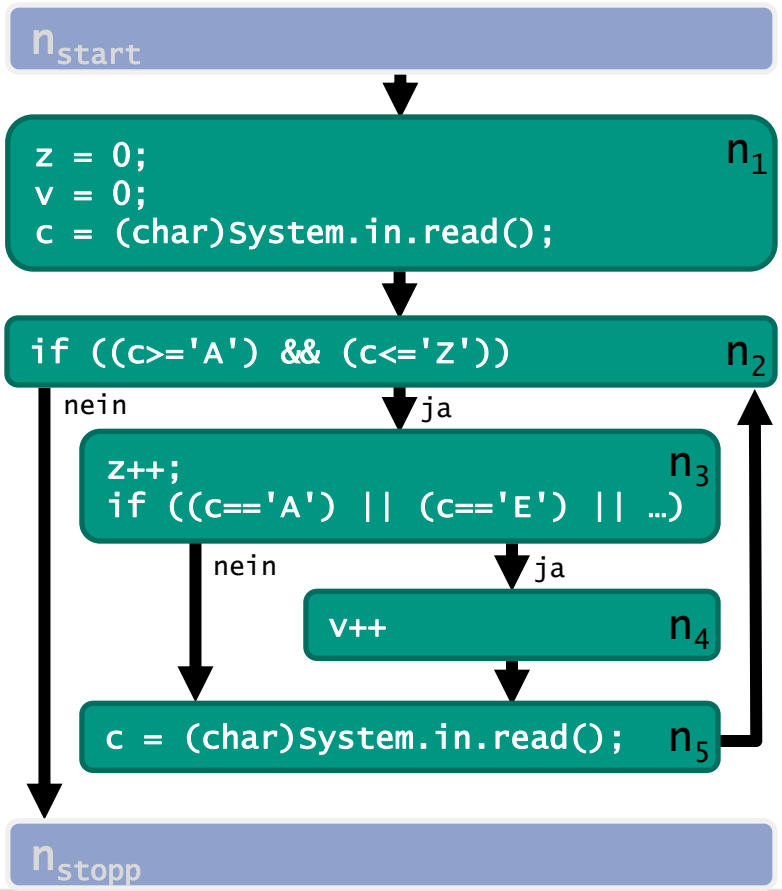
→ Zweig (n_3, n_5) wird nicht ausgeführt



Zweigüberdeckung, alle Kanten

Beispielfolge:

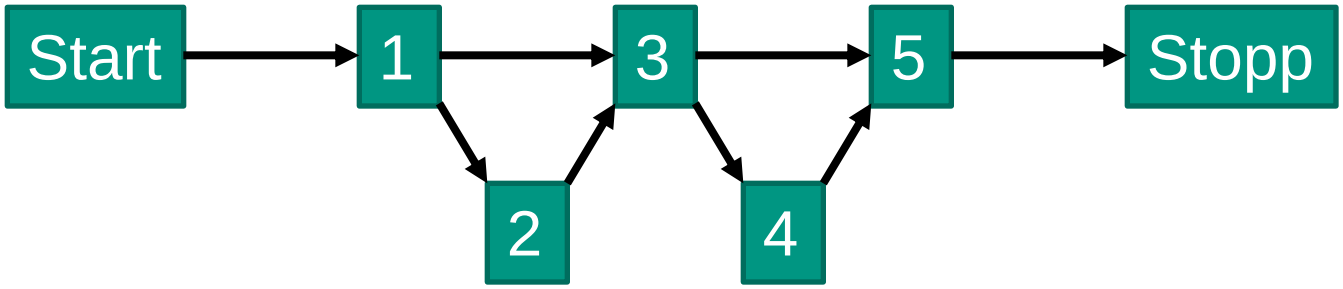
(n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{stopp})



Definition: Pfadüberdeckung

- Die **Pfadüberdeckung** fordert die Ausführung aller unterschiedlichen Pfade im Programm.
  - Pfadanzahl wächst bei Schleifen dramatisch an.
  - Manche Pfade können nicht ausführbar sein, durch sich gegenseitig ausschließende Bedingungen
  - Mächtigste KFO Teststrategie
  - Nicht praktikabel

Beispiel für Anweisungs-, Zweig- und Pfadüberdeckung



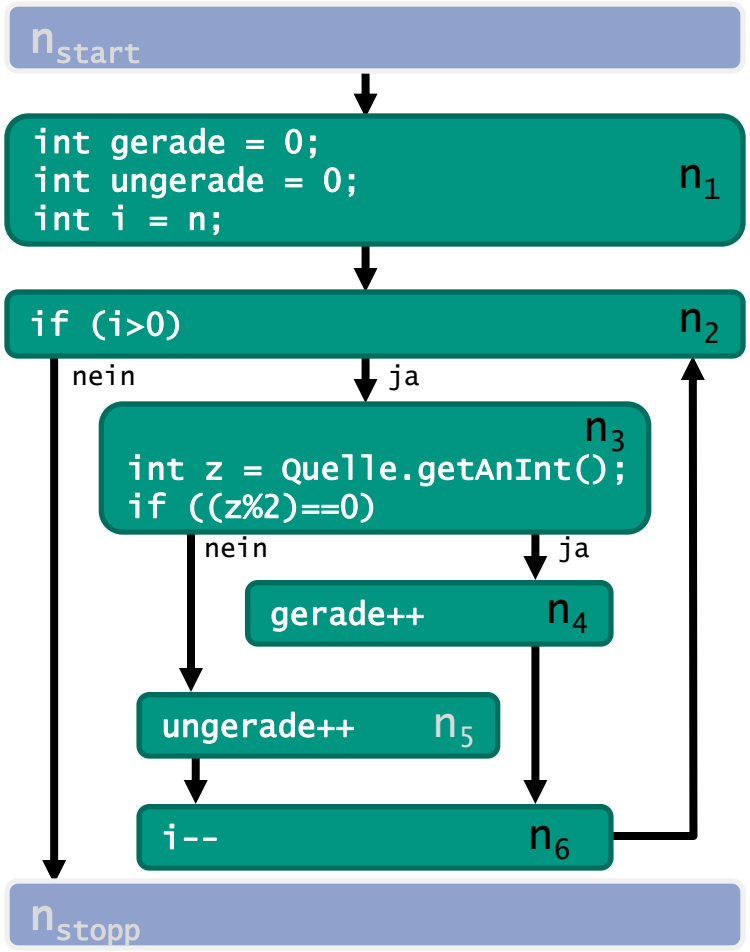
- Anweisungsüberdeckung
  A = { (Start, 1, 2, 3, 4, 5, Stopp) }
- Zweigüberdeckung
  Z = A ∪ { (Start, 1, 3, 5, Stopp) }
- Pfadüberdeckung
  P = Z ∪ { (Start, 1, 3, 4, 5, Stopp) }
  ∪ { (Start, 1, 2, 3, 5, Stopp) }

Aufwand der Pfadüberdeckung

Beispielcode

```
int gerade = 0;
int ungerade = 0;
int i = n;
while (i>0) {
  int z = Quelle.getAnInt();
  if ((z%2)==0) gerade++;
  else ungerade++;
  i--;
}
```

n	Anzahl Pfade
0	I
1	II
2	IIII
…	…
k	2^k



Bedingungsüberdeckungsverfahren

- Zweigüberdeckung nicht ausreichend bei zusammengesetzten oder hierarchischen Bedingungen
- Drei Ausprägungen der Bedingungsüberdeckungen (BÜ)
  - Einfache BÜ
  - Mehrfache BÜ
  - Minimal-mehrfache BÜ

Definition: Einfache BÜ

- Die **einfache Bedingungsüberdeckung** fordert, dass jede atomare Bedingung einmal mit *Wahr* (*W)* und einmal mit *Falsch* (*F*) belegt wird.
  - Enthält weder Zweig- noch Anweisungsüberdeckung
  - **Daher:** Nicht ausreichendes Testkriterium

Programmbeispiel

```
if ( A & B ) {
  S1;
}
else {
  S2;
}
```

- Eingaben für A und B für einfache BÜ?
- Eingabe $E_{Einfach}$ = {10, 01} erfüllt Einfache BÜ; aber S1 wird nicht ausgeführt
  - → kein effektiver Test!
  - Weder Zweig- noch Anweisungsüberdeckung
- Wir nennen einen Test effektiv, wenn er eine hinreichend große Chance hat, einen Defekt zu entdecken.

Definition: Mehrfache BÜ

- Die **mehrfache Bedingungsüberdeckung** fordert, dass die atomaren Bedingungen mit allen möglichen Kombination der Wahrheitswerte *W* und *F* belegt werden.
  - Ergibt bei n atomaren Bedingungen 2^n Kombinationen
  - Enthält Zweigabdeckung
  - Aufwendig zu realisieren, setzt Identifikation nicht möglicher Bedingungskombinationen voraus
  - Findet Defekt bei Bedingungen
  - Aber keinen größeren Einfluss bei Strukturfehlern in Bedingungen

Programmbeispiel

```
if ( A & B ) {
  S1;
}
else {
  S2;
}
```

- Eingaben für A und B für mehrfache BÜ?
- Eingaben $E_{mehrfach}$ = { 00, 01, 10, 11 } erfüllen mehrfache BÜ, Anweisungs- und Zweigüberdeckung.

Definition: Minimal-mehrfache BÜ

- Die **minimal-mehrfache Bedingungsüberdeckung** fordert, dass jede Bedingung, ob atomar oder zusammengesetzt, jeweils zu *W* und *F* evaluieren muss.
  - Enthält den Zweigtest
  - Invariante Bedingungen (Bedingungen die immer *W* oder *F* liefern) können entdeckt werden

Jeder Ausdruck muss (min.) 1 mal W und (min.) 1 mal F werden

$(a \wedge b) \vee (c \wedge d)$

$(a \wedge b \wedge c)$

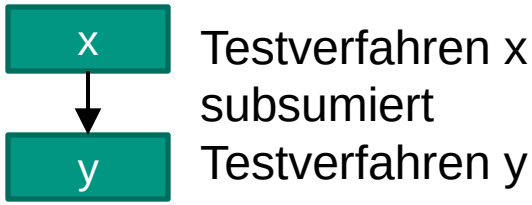
Programmbeispiel

```
if ( A & B ) {
  S1;
}
else {
  S2;
}
```

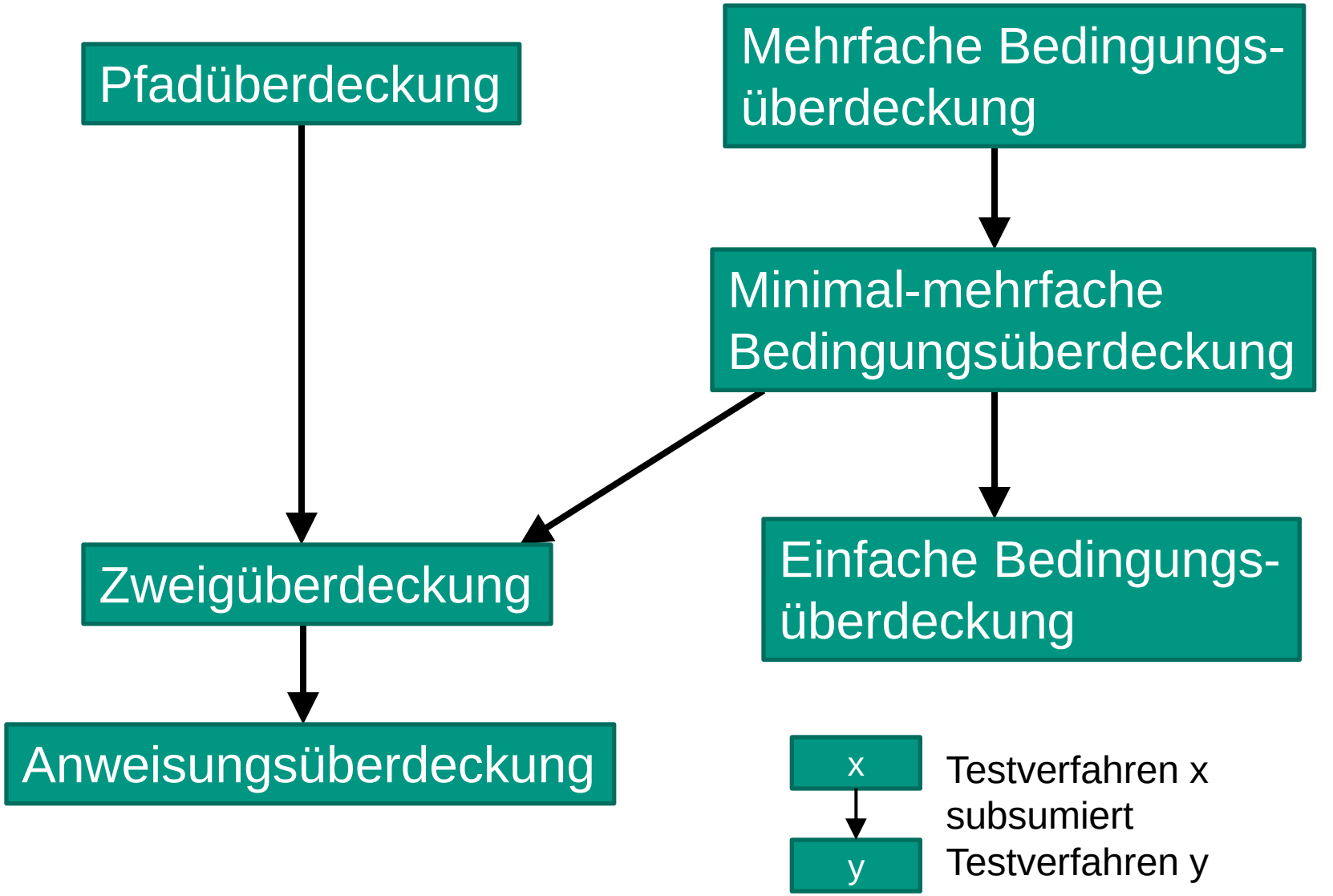
- Eingaben für A und B für minimal-mehrfache BÜ?
- Eingaben $E_{\text{minimal-mehrfach}} = E_{\text{einfach}} \cup \{ 11 \}$ erfüllen minimal-mehrfache BÜ
- { 00, 11 } erfüllt einfache und minimal-mehrfache BÜ

Definition: Subsumieren

- Ein Testverfahren für Kriterium x **subsumiert** ein Testverfahren für Kriterium y, wenn jede Menge von Pfaden, die Kriterium x erfüllt, auch Kriterium y erfüllt.



Hierarchie der KFO-Teststrategien



Definition: Kurzauswertung

- Bei der **Kurzauswertung** wird die Auswertung einer zusammengesetzten Bedingung abgebrochen, sobald das Ergebnis feststeht.
- Beispiel:
  - `if ( a!=0 && x/a > 1 ) …`
    Wenn a = 0 ist, so wird die Auswertung abgebrochen.
  - `if ( a!=0 || x/a > 1 ) …`
    Wenn a ≠ 0 ist, so wird die Auswertung abgebrochen.
  - Kurzauswertung in Java, z.B. bei „&&”- und „| |”-Operator.
  - In Java erfolgt bei den Operatoren „&“ und „|“ keine Kurzauswertung.

Seiteneffekt der Kurzauswertung

- Kurzauswertung beeinflusst Wahl der Eingaben:

```
if ( A && B ) {
  S1;
}
else {
  S2;
}
```

- **Hier**: $E_{einfach,\ kurz} = \{ 0x, 10, 11 \}$
- **Generell**: $|E_{einfach,\ kurz}| \geq |E_{einfach}|$
- Bei Kurzauswertung muss mehr getestet werden: die tatsächliche Überprüfung weiter rechts stehender Ausdrücke ist hier von der Auswertung des Terms links abhängig!

Seiteneffekt der Kurzauswertung

- Kurzauswertung beeinflusst Wahl der Eingaben:

```
if ( A && B ) {
  S1;
}
else {
  S2;
}
```

- **Hier**: $E_{mehrfach,\ kurz} = \{ 00, 01, 10, 11 \} = E_{mehrfach}$
- **Generell**: Gleichheit folgt aus den Definitionen (es gibt nur 1 Möglichkeit)

Seiteneffekt der Kurzauswertung

- Kurzauswertung beeinflusst Wahl der Eingaben:

```
if ( A && B ) {
  S1;
}
else {
  S2;
}
```

- **Hier**: $E_{\text{minimal-mehrfach, kurz}} = E_{\text{einfach, kurz}}$
- **Generell**: Mögliche Gleichheit folgt aus den Definitionen (Es gibt aber mehrere Möglichkeiten!)

Zusammenfassung: KFO Teststrategien (1)

- **Anweisungsüberdeckungstest** nur angebracht, wenn keine Schleifen oder Bedingungen ausgeführt werden, ansonsten nicht ausreichend
- **Zweigüberdeckung** angebracht, wenn keine Schleifen und nur atomare Bedingungen vorhanden
- **Pfadüberdeckung** aufwendigstes und schon für kleine Programme nicht zu realisierendes Kriterium
- **Behandlung von Schleifen**:
  - Konstruiere einen Satz Testfälle, die den Schleifenrumpf einmal queren,
  - Einen zweiten Satz, die den Schleifenrumpf mindestens zweimal queren
  - Verzweigungen innerhalb des Schleifenrumpfes mit Zweigüberdeckung berücksichtigen, d.h. beim einmaligen Queren alle Zweige abdecken, beim mehrmaligen Queren in der letzten Querung alle Zweige abdecken (andere egal).
  - Eine Schleife mit einer bedingten Anweisung braucht also Testfälle für 2*2 Pfade.

Zusammenfassung: KFO Teststrategien (2)

- **Einfache BÜ** nicht ausreichendes Kriterium, schwächer als Anweisungsüberdeckung
- **Mehrfache BÜ** sehr aufwendig, deckt keine Defekt in Bedingungsstruktur auf
- **Minimal-Mehrfache BÜ** gute Erweiterung des Konzepts der Zweigüberdeckung
- **Generell:** die verschiedenen Teststrategien sind auch Testvollständigkeitskriterien. (Beispiel: ein Test nach der Zweigüberdeckung ist vollständig, wenn C_1=1)

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest	✓					
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Funktionale Tests

- **Ziel**: Testen der spezifizierten Funktionalität
  - Testfälle aus der Spezifikation ableiten
  - Interne Struktur des Testlings nicht berücksich
    Tester unbekannt
  - Vorteile:
    - Testfälle unabhängig von der Implementierung erstellbar (vorher oder gleichzeitig)
    - Vermeidet „Kurzsichtigkeit“ bei der Auswahl
  - Nachteil:
    - mögl. kritische Pfade nicht bekannt & nicht getestet

formal – automatisierbar
informal – interpretierbar

Funktionale Tests

- **Ziel**: Testen der spezifizierten Funktionalität
  - Testfälle aus der Spezifikation ableiten
  - Interne Struktur des Testlings nicht berücksichtigt weil für den Tester u[nb]ekannt
  - Vorteile:
    - Testfälle u[nabhängig] gleichzeitig)
    - Vermeidet „Ku[rzschluss]“
  - Nachteil:
    - mögl. kritische [Fehler]

Testfälle enthalten

- Eingabedaten
- Erwartete Ausgabedaten/Reaktion (Soll)

Problem:

Sei die zu testende Funktion $f(x) := x^2$.

→ Offensichtlich ist es nicht möglich, einen vollständigen Funktionstest durchzuführen.

⇒ So wenig Testfälle wie möglich, aber so viele Testfälle wie nötig, dass die Wahrscheinlichkeit groß genug ist, einen Defekt zu finden.

Verfahren zur Testfallbestimmung

- Funktionale Äquivalenzklassenbildung
- Grenzwertanalyse
- Zufallstest
- Test von Zustandsautomaten

Funktionale Äquivalenzklassenbildung

- **Annahme**: Ein Programm/Modul reagiert bei der Verarbeitung eines Wertes aus einem bestimmten Bereich genau so wie bei der Verarbeitung jedes anderen Wertes aus diesem Bereich.
- **Beispiel**: Wir erwarten, dass eine Funktion, die 42^2 richtig berechnet, auch 40^2 richtig berechnet.
- **Ansatz**: Zerlege Wertebereich der Eingabeparameter und Definitionsbereich der Ausgabeparameter in Äquivalenzklassen (ÄK).

Funktionale Äquivalenzklassenbildung

- **Annahme**: Ein Programm/Modul reagiert bei der Verarbeitung eines Wertes aus einem bestimmten Bereich genau so wie bei der Verarbeitung jedes anderen Wertes aus diesem Bereich.
- **Beispiel**: Wir erwarten, dass eine Funktion, die 42^2 richtig berechnet, auch 40^2 richtig berechnet.
- **Ansatz**: Zerlege Werte[illegible]d Definitionsbereich der [illegible] Äquivalenzklassen (ÄK

Muss nicht im Definitionsbereich der Eingabeparameter liegen!
→ Fehlerbehandlung testen

Funktionale Äquivalenzklassenbildung

- Bildung der Äquivalenzklassen (Ansatzidee)
  - Partitionierung ausgehend von einer großen ÄK
  - Aufteilung entlang Definitionsbereichsgrenzen
  - Aufteilung entlang anzunehmenden(!) Verarbeitungsmethodengrenzen
- Auswahl der Repräsentanten
  - Sei m die Anzahl der ÄK der Eingabeparameter und n die Anzahl der ÄK der Ausgabeparameter.
  - Dann entstehen ≤ m·n verschiedene ÄK aus denen jeweils 1 beliebiger Repräsentant zum Testen verwendet wird

Äquivalenzklassenbildung: Beispiel (1)

- Funktion zur Bestimmung der Anzahl der Tage in einem Monat eines bestimmten Jahres (nach dem Gregorianischen Kalender).
- Eingabewerte: Jahr, Monat: Integer

ÄK	Beschreibung	Mögliche Werte
$AK_{M,31}$	Monate mit 31 Tagen	1,3,5,7,8,10,12
$AK_{M,30}$	Monate mit 30 Tagen	4,6,9,11
$AK_{M,Feb}$	Monat mit 28 oder 29 Tagen	2
$AK_{M,Fehler}$	Ungültige Eingaben	Monat < 1 oder Monat > 12
$AK_{J,SJ}$	Schaltjahre	1904, 2000
$AK_{M,Nicht-SJ}$	Nicht-Schaltjahre	1901, 1900
$AK_{J,Fehler}$	Ungültige Eingaben	Jahr < 0

Äquivalenzklassenbildung: Beispiel (2)

- Ableitung von Testfällen (gültige Eingaben):

Äquivalenzklasse	Eingabe		Soll-Ausgabe
	Monat	Jahr	
$AK_{M,31}$ und $AK_{J,Nicht\text{-}SJ}$	7	1900	31
$AK_{M,31}$ und $AK_{J,SJ}$	7	2000	31
$AK_{M,30}$ und $AK_{J,Nicht\text{-}SJ}$	6	1900	30
$AK_{M,30}$ und $AK_{J,SJ}$	6	2000	30
$AK_{M,Feb}$ und $AK_{J,Nicht\text{-}SJ}$	2	1900	28
$AK_{M,Feb}$ und $AK_{J,SJ}$	2	2000	29

Äquivalenzklassenbildung: Beispiel (3)

- Ableitung von Testfällen (ungültige Eingaben):

Äquivalenzklasse	Eingabe		Soll-Ausgabe
	Monat	Jahr	
$AK_{M,Fehler}$ und $AK_{J,Nicht\text{-}SJ}$	-1	1900	Fehler
$AK_{M,Fehler}$ und $AK_{J,SJ}$	13	2000	Fehler
$AK_{M,30}$ und $AK_{J,Fehler}$	6	-1	Fehler
$AK_{M,31}$ und $AK_{J,Fehler}$	7	-1	Fehler
$AK_{M,Feb}$ und $AK_{J,Fehler}$	2	-42	Fehler
$AK_{M,Fehler}$ und $AK_{J,Fehler}$	-1	-42	Fehler

Grenzwertanalyse

- Weiterentwicklung der funktionalen Äquivalenzklassenbildung
- Beobachtungen:
  - Off-by-one: Knapp daneben ist auch vorbei
  - Testfälle, die die Grenzen der Äquivalenzklassen und deren unmittelbare Umgebung abdecken sind besonders effektiv

⇒ Verwende nicht irgendein Element aus der ÄK, sondern solche auf und um den Rand (Annäherung von beiden Seiten)

- Spezielle Kandidaten: 0, 1, MAX_INT, NaN, …

Zufallstest

- Testen der Funktion mit zufälligen Testfällen
- **Beobachtung**: Tester neigen dazu, Testfälle zu erzeugen, die auch bei der Implementierung als nahe liegend erachtet wurden
- **Vorteil**: nichtdeterministisches Verfahren behandelt alle Testfälle gleich
- Sinnvoll als Ergänzung zu anderen Verfahren
- Sinnvoll wenn man viele Testfälle erzeugen möchte (z.B. für Sortierverfahren) und die Überprüfung der Korrektheit automatisch erfolgen kann.
- Sinnvoll als unterlagertes Kriterium (z.B. beim Verfahren der funktionalen ÄK-Bildung)

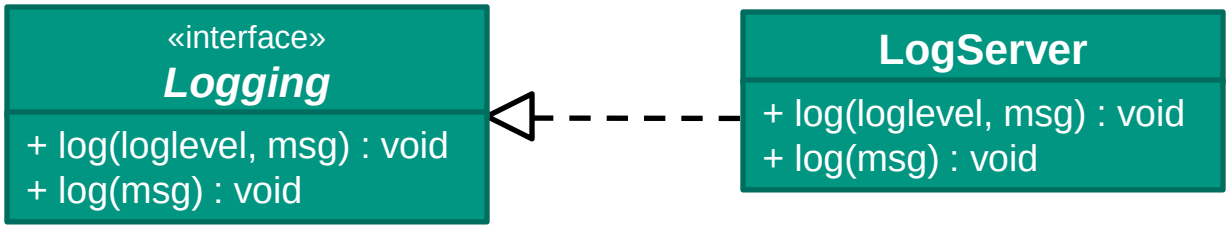
Verwendung von Testhelfern (1)

- Objekte leben nicht isoliert, Objekte arbeiten in einer Anwendung zusammen.
- Zusammenarbeit resultiert in Abhängigkeiten beim Testen.
- Wie können einzelne Eigenschaften getestet werden im Hinblick auf diese Abhängigkeiten?
  - Was genau soll getestet werden ?
  - Welche Probleme sollen getestet werden ?
- Wie können Tests geschrieben werden, die Abhängigkeiten auflösen ?

Verwendung von Testhelfern (2)

- Wenn Klassen voneinander abhängen, können noch nicht implementierte Klassen durch Testfehler (Stummel, Attrappe oder Nachahmung) ersetzt werden.
- Stummel oder Attrappen vertreten die noch fehlende Implementierungen.
- Stummel und Attrappen werden sukzessive durch echte Implementierungen ersetzt; Nachahmungen sind für zukünftiges Testen auch weiterhin nützlich.

Beispiel: Attrappe, Nachahmung (1)

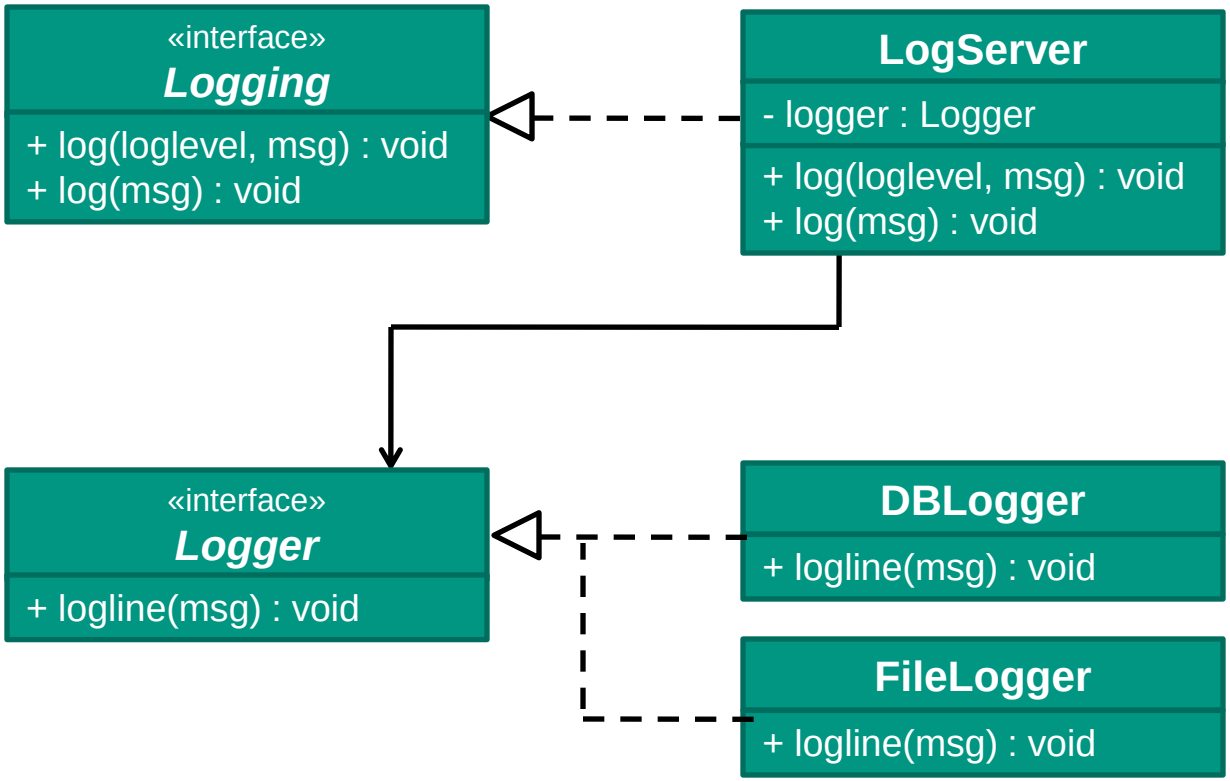


- Systemweite Logbuchführung bereitstellen
- `log(loglevel, msg)` protokolliert Nachrichten mit verschiedener Priorität
- `log(msg)` protokolliert Nachrichten mit Standardpriorität „2“

Beispiel: Attrappe, Nachahmung (2)

- **Problem**: Wohin protokolliert der LogServer ?
  - Für initiale Zwecke ist das Dateisystem vorgesehen
  - Relationale Datenbank könnte kommen
  - Beide Möglichkeiten nicht optimal für das Testen des LogServers.
- **Lösung**:
  - Verberge Protokollmedium hinter Schnittstelle **`Logger`**

Beispiel: Attrappe, Nachahmung (3)

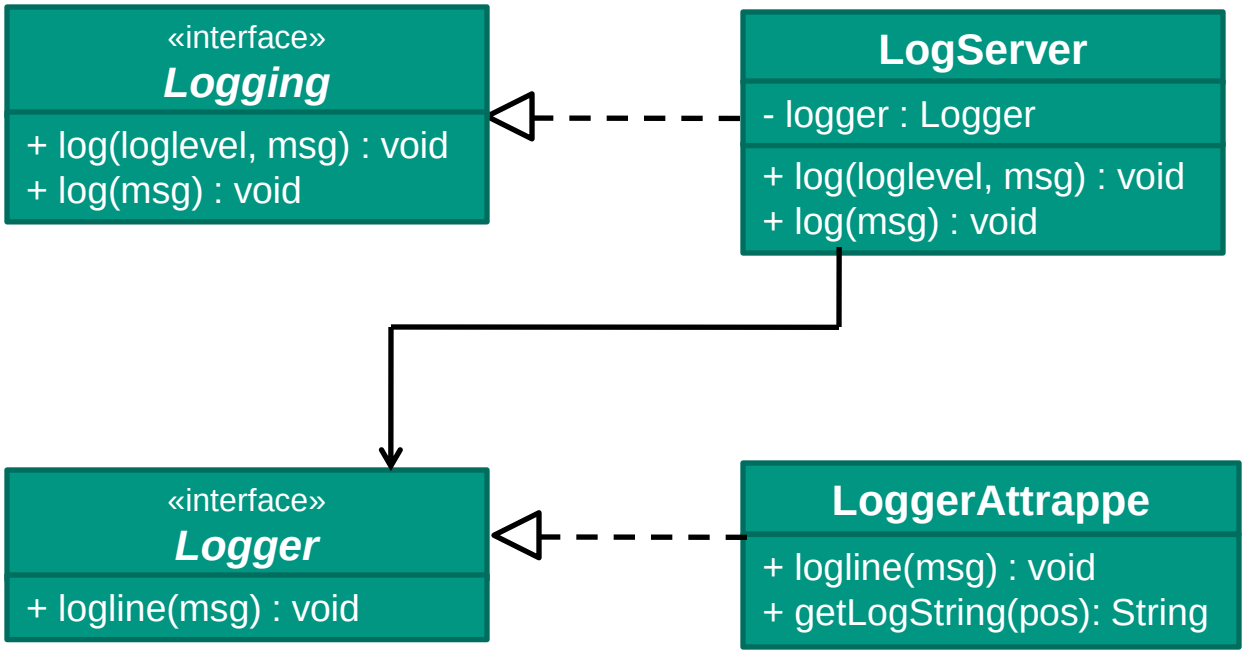


Beispiel: Attrappe, Nachahmung (4)

- Die wichtigste Eigenschaft des Loggers ist explizit:
  - zeilenweises Protokollieren
- Das Zielmedium wird hinter Schnittstelle verdeckt: `DBLogger`, `FileLogger`
- `LogServer` beinhaltet eine Instanz des Loggers.

- Was bringt uns das jetzt für das Testen ?

Beispiel: Attrappe (1)



Beispiel: Attrappe (2)

- Implementiert die Logger-Schnittstelle für **`LogServer`** und
- `getLogString` für den Test
  - `getLogString(pos)` stellt die `pos`-te Nachricht bereit
- Nun können wir einen Test des LogServers schreiben:

```
public void testSimpleLogging() {
   LoggerAttrappe logger = new LoggerAttrappe();
   Logging logServer = new LogServer(logger);

   logServer.log(0, "Erste Zeile");
   logServer.log(1, "Zweite Zeile");
   logServer.log("Dritte Zeile");

  assertEquals("(0): Erste Zeile",
       logger.getLogString(0));
   assertEquals("(1): Zweite Zeile",
       logger.getLogString(1));
   assertEquals("(2): Dritte Zeile",
       logger.getLogString(2));
}
```

Beispiel: Attrappe (3)

- LogServer, der von Protokollmedium unabhängig ist.
- Schnittstelle zum Testen unseres LogServers
- Überprüfen des `LogServers` ohne explizites Sichtbarmachen der Implementierung.
- **Frage**: Was kann man hier noch verbessern ?
- **Antwort**: Testcode kann zur `LoggerAttrappe` wandern. Sie wird damit zur `LoggerNachahmung`

Beispiel: Nachahmung (1)

```
public class LoggerNachahmung implements Logger {
    private List expectedLogs = new ArrayList();
    private List actualLogs = new ArrayList();

    public void addExpectedLine(String logString) {
      expectedLogs.add(logString);
    }
    public void logLine(String logLine) {
      actualLogs.add(logLine);
    }
    public void verify() {
      if (actualLogs.size() ! = expectedLogs.size()) {
        Assert.fail("Expected" + expectedLogs.size()
          +‘‘log entries but encountered‘‘+actualLogs.size());
      }
      for (int i = 0; i < expectedLogs.size(); i++){
        String expLine = (String) expectedLogs.get(i);
        String actLine = (String) actualLogs.get(i);
        Assert.assertEquals(expLine, actLine);
      }
    }
}
```

- Setzt die von LogServer erwartete Eingabe.
- Wird am Anfang eines Tests aufgerufen.

- Überprüft die Anzahl der Protokollmeldungen und deren Inhalt.
- Wird am Ende eines Tests aufgerufen.

Beispiel: Nachahmung (2)

- Testcode:

```
LoggerNachahmung logger;
@Test
public void testSimpleLogging() {
  logger.addExpectedLine("(0): Erste Zeile");
  logger.addExpectedLine("(1): Zweite Zeile");
  logger.addExpectedLine("(2): Dritte Zeile");
  logServer.log(0, "Erste Zeile");
  logServer.log(1, "Zweite Zeile");
  logServer.log("Dritte Zeile");
  logger.verify();
}
```

- Fehler werden erst beim Aufruf von `verify()` gemeldet
- Bei komplizierterem Verhalten ist damit Finden der Ursache schwierig

Beispiel: Nachahmung (3)

- Geänderte `LoggerNachahmung`:

```
public void logLine(String logLine) {
   Assert.assertNotNull(logLine);
   if (actualLogs.size() >= expectedLogs.size()) {
     Assert.fail("Too many log entries");
   }

   int currentIndex = actualLogs.size();
   String expectedLine =
     (String) expectedLogs.get(currentIndex);
   Assert.assertEquals(expectedLine, logLine);
   actualLogs.add(logLine);
}
public void verify() {
   if (actualLogs.size() < expectedLogs.size()) {
     Assert.fail("Expected" + expectedLogs.size()
      + "log entries but encountered" + actualLogs.size());
   }
}
```

Beispiel: Nachahmung (4)

- Nachahmung stellt für jede Eingabe eine `setExpected` Methode zur Verfügung, `addExpected` wenn mehrere Aufrufe erwartet werden
- Nachahmung enthält Testcode in `verify`
- Nachahmung überprüft Parameter, wie auch Reihenfolge der Methodenaufrufe (Protokollanalyse)
- **Achtung**: Es wird nicht die Nachahmung getestet, sondern die Benutzung der Nachahmung.

Test von Zustandsautomaten

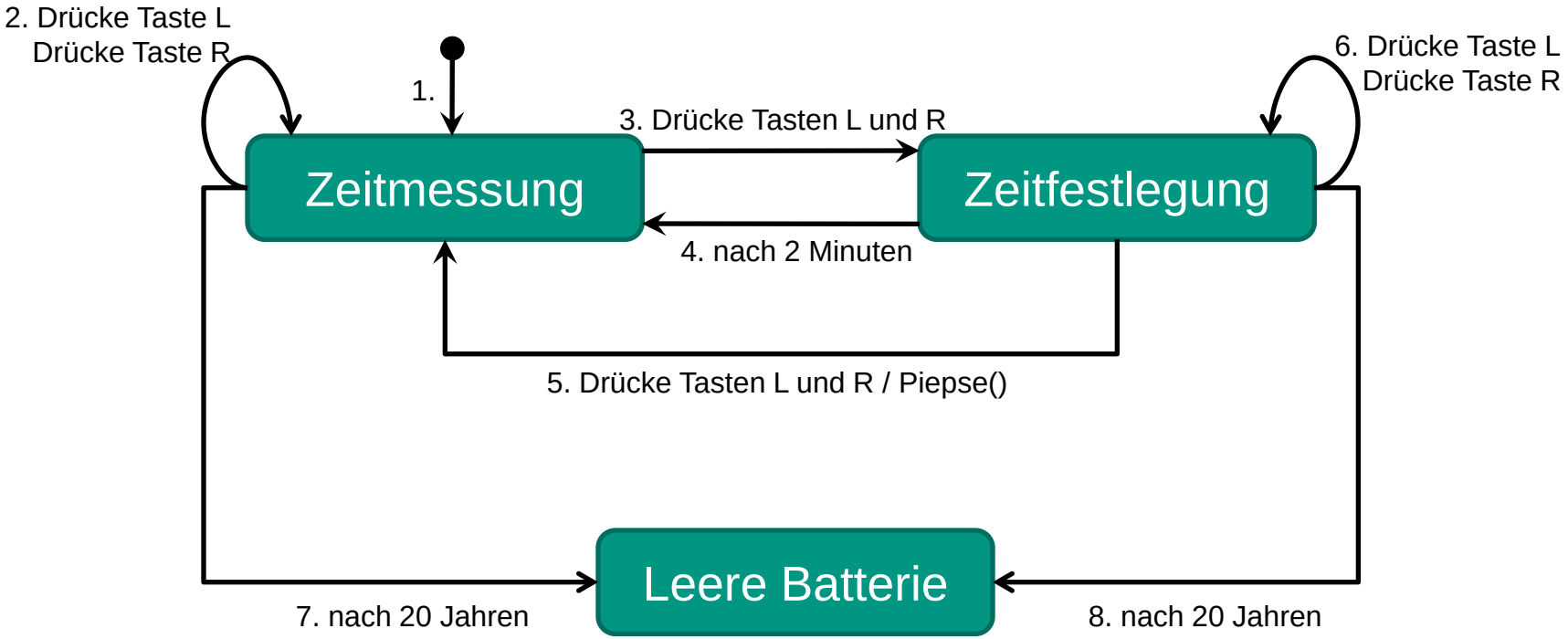
- Hat eine Komponente einen internen Zustand, können Testfälle aus den Zustandsübergängen abgeleitet werden
- **Ziel**: mindestens einmaliges Durchlaufen aller Übergänge
- **Achtung**:
  - Überdeckung aller Übergänge garantiert noch keinen vollständigen Test (vgl. Zweigüberdeckung)
  - Zum Nachvollziehen der Testsequenzen ist ggf. eine Instrumentierung der Komponente notwendig

Test von Zustandsautomaten: α-ω-Zyklus

- Der **α-ω-Zyklus** bezeichnet den kompletten Lebenszyklus eines Objektes, von der Speicherallokation (vor Ausführung jeglicher Konstruktoren) bis zur Speicherfreigabe (nach Ausführung aller Destruktoren).
- Test gerade bei automatisch speicherbereinigenden Sprachen problematisch
  - Java: `finalize()`-Zombies (in der `finalize()`-Methode wird evtl. wieder eine lebendige Referenz auf das zu entfernende Objekt erzeugt, dadurch wird es nicht freigegeben)
  - Schwache Referenzen – Objekte erleben ihr ω, wenn der Speicher knapp wird
  - „shut-down hooks“ – Code, der beim Beenden des (Teil-) Systems ausgeführt wird

Test von Zustandsautomaten: Beispiel „Uhr mit zwei Tasten“

- Gegeben: Uhr mit zwei Tasten „L“ und „R“



Test von Zustandsautomaten: Beispiel „Uhr mit zwei Tasten“

Testfall	Getestete Übergänge	Erwarteter Zustand
Leere Menge	1.	Zeitmessung
Drücke Taste L	2.	Zeitmessung
Drücke Tasten L und R gleichzeitig	3.	Zeitfestlegung
Warte 2 Minuten	4. „Timeout“	Zeitmessung
Drücke Tasten L und R gleichzeitig	3.	Zeitfestlegung
Drücke Tasten L und R gleichzeitig	5.	Zeitmessung
Drücke Tasten L und R gleichzeitig	3.	Zeitfestlegung
…	…	…

Es gibt noch weitere Testfälle.

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest	✓		✓			
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Leistungstests: Lasttests

- Testet das System/die Komponente auf Zuverlässigkeit und das Einhalten der Spezifikation **im erlaubten Grenzbereich**
  - Kann das System die geforderte Nutzerzahl bedienen?
  - Können garantierte Antwortzeiten eingehalten werden?
  - Kann diese Last beliebig lange bewältigt werden?
  - Wie ist die Auslastung der erwarteten Flaschenhälse?
  - Gibt es unerwartete Flaschenhälse?

Leistungstests: Stresstest

- Testet das Verhalten des Systems **beim Überschreiten** der definierten Grenzen
  - Wie ist das Leistungsverhalten bei Überlast?
    - Seitenflattern
    - Exponentielles Ansteigen der Antwortzeit
    - Stillstand
  - Kehrt das System nach dem Rückgang der Überlast zum definierten Verhalten zurück?
    - Wie lange dauert das?

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest	✓		✓	✓		
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Automatismen vs. manuelle Prüfung

- Machbarkeit
  - Syntax ↔ Semantik – Syntax kann geprüft werden, wie viel Semantik kann in die Syntax verpackt werden?
  - Berechenbarkeit – Ist das Problem, falls wir es formalisieren können, überhaupt (effizient) berechenbar?
- Geschwindigkeit
  - Zeit für die Formalisierung und/oder Codierung vs. Ausführungszeit der Prüfung
- Flexibilität
  - Kann das automatische Werkzeug mit dem konkret vorliegenden Anwendungsfall umgehen
- Aufwand (Kosten)
  - Suche, Beschaffung, Einführung (Schulung), Codierung vs. Durchführungsgeschwindigkeit, Personalaufwand

Manuelle Prüfung – Anmerkungen

- Einzige Möglichkeit, die Semantik zu prüfen
- Aufwendig (bis 20% der Erstellungskosten)
- Zeit für die Prüfungen muss eingeplant sein
- Psychologischer Druck bei der Begutachtung der Arbeit eines Einzelnen durch eine Gruppe (Anklagebank, Verteidigung)

→ Berücksichtigung solcherlei sozialen Konfliktpotentials im Prüfprozess

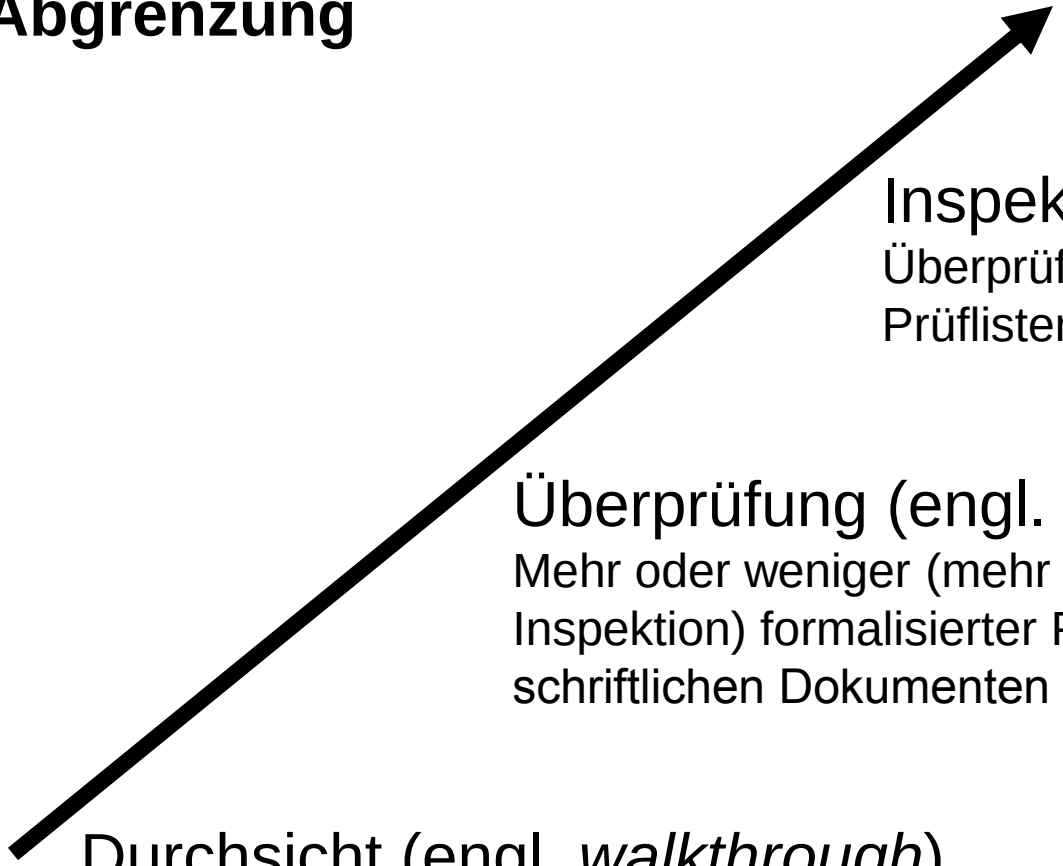
  - Die Arbeit, nicht die Person wird begutachtet
  - Die Arbeit aller Mitarbeiter regelmäßig begutachten
  - Möglichst keine Kunden/höheren Manager anwesend

Software-Inspektionen (Überblick)

- Mehrere Inspektoren (2 bis 8) untersuchen unabhängig voneinander dasselbe Artefakt (Software-Dokument)
- Gefundene Defekte werden aufgeschrieben und Gemeinsamkeiten durchgesprochen
- Probleme identifizieren, nicht lösen
- Strukturierter Prozess
  - Rollen und Formulare
  - Prüflisten und Szenarien

} Anpassung an Dokumenttyp

Abgrenzung



Inspektion

Überprüfung anhand von Prüflisten oder Lesetechniken

Überprüfung (engl. *review*)

Mehr oder weniger (mehr als Durchsicht, weniger als Inspektion) formalisierter Prozess zur Überprüfung von schriftlichen Dokumenten durch einen „externen“ Gutachter

Durchsicht (engl. *walkthrough*)

Der Entwickler führt einen oder mehrere beteiligte Kollegen durch einen Teil des Codes bzw. Entwurfs, den er geschrieben hat. Die Kollegen stellen Fragen und machen Anmerkungen zu Stil, Defekten, Einhalten von Entwicklungsstandards und anderen Problemen. (nach ANSI/IEEE 729-1983)

Definition: Inspektion

- Die Inspektion ist eine formale Qualitätssicherungstechnik, bei der Anforderungen, Entwurf oder Code eingehend durch eine vom Autor verschiedene Person oder eine Gruppe von Personen begutachtet werden. Zweck ist das Finden von Fehlern, Verstößen gegen Entwicklungsstandards und anderen Problemen.

(nach ANSI/IEEE Standard 729-1983)

Vor- und Nachteile von Inspektionen

- Vorteile
  - Anwendbar auf alle Softwaredokumente: Pflichtenhefte, Spezifikationen, Entwürfe, Code, Testfälle, …
  - Jederzeit und frühzeitig durchführbar
  - Sehr effektiv in der industriellen Praxis
- Nachteile
  - Gehen nur aufwendig von Hand
  - verbrauchen Zeit mehrerer Mitarbeiter
  - somit erst einmal teuer
  - „statisch“ (im Gegensatz zum Testen)

IBM
Hewlett Packard
Motorola
Siemens
NASA

Zahlen über Nutzen und Kosten

- Meist mehr als 50 Prozent aller entdeckten Defekte werden in Inspektionen gefunden (bis zu 90 Prozent). Faustregel: 50-75% bei Entwurfsfehlern (Empirie)
- Verhältnis der Fehlerbehebungskosten durch Inspektion versus durch Testen identifizierter Fehler oft zwischen 1:10 und 1:30
- Return on Investment (ROI) wird häufig mit weit über 500 Prozent angegeben
- Weitere Studie: Fagan, M., *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal 15, 3 (1976): 182-211

Warum Inspektionen effektiv sind

- Gedanke: „Viele sehen mehr als einer.“
- Inspektoren haben Abstand zum Software-Dokument
- Inspektoren bringen Erfahrung ein
- Gemeinsame Diskussion über die identifizierten Problempunkte (engl. *issues*)
  - Ist es ein Defekt?
  - Ist es eine Schwäche?
  - Ist es „nur unvorteilhaft“?
  - Gemeinsame Einsicht

Phasen einer Inspektion

1. Vorbereitung
2. Individuelle Fehlersuche (IF)
3. Gruppensitzung (GS)
4. Nachbereitung
5. Prozessverbesserung

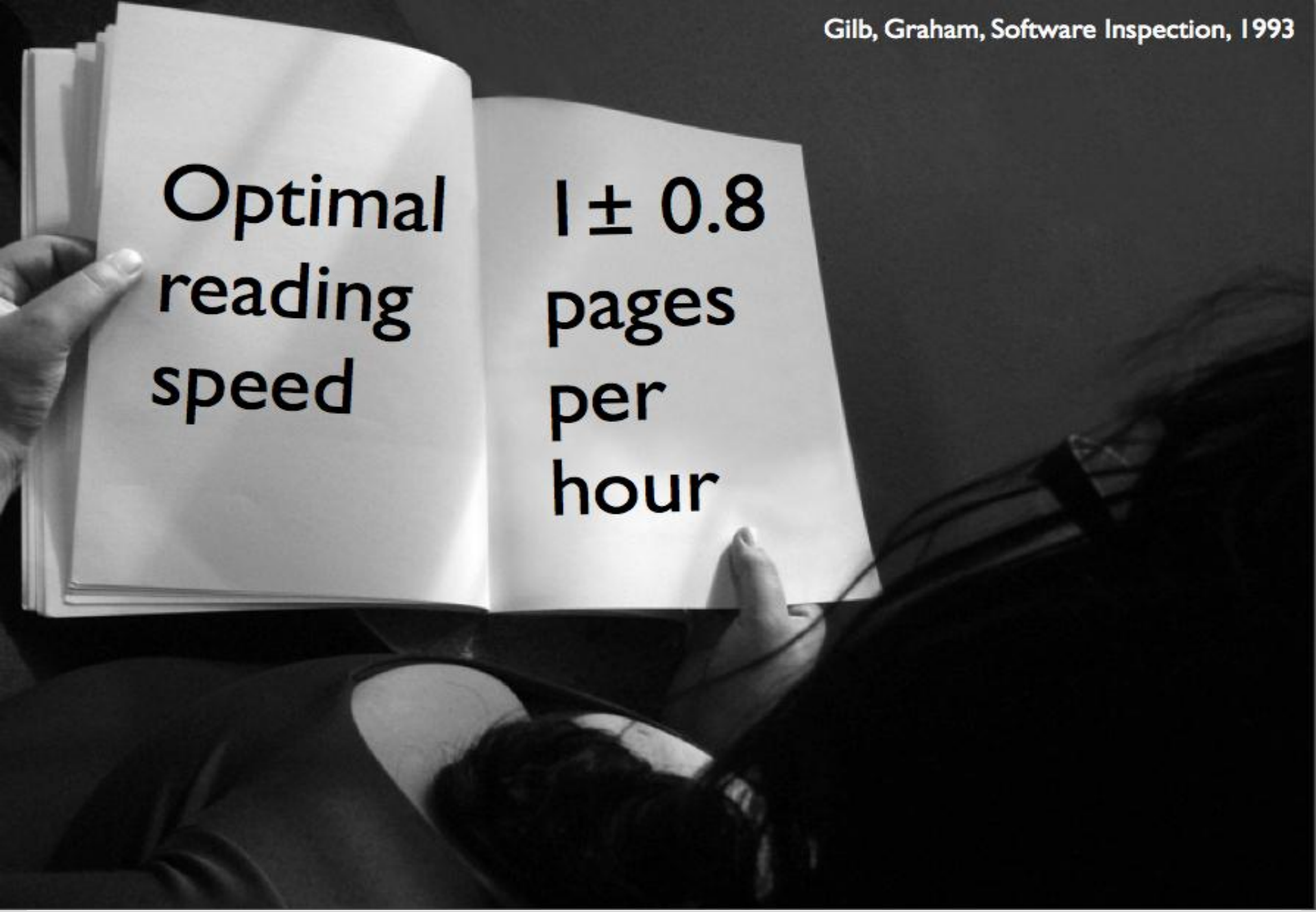
1. Vorbereitung

- Teilnehmer und ihre Rollen festlegen
- Dokumente und Formulare vorbereiten
- Lesetechniken festlegen
- Zeitlichen Ablauf planen
  - Optional: Initialtreffen
  - Ggf. Aufteilung des Prüflings: Eine GS soll nicht länger als 2 Stunden dauern (Konzentration auf das Wichtige, aktive Teilnahme)

2. Individuelle Fehlersuche

- Jeder Inspektor prüft für sich das Dokument
  - jeweils etwa 1 Seite pro Stunde pro Inspektor
  - Ergebnisse der IF müssen in 1 Gruppensitzung durchgegangen werden können

    ⇒ Teile von max. 1-4 Seiten netto
- Benutzt vereinbarte Lesetechnik
- Schreibt alle Problempunkte und die genaue Stelle im Dokument auf
- Problempunkte: (mögliche) Defekte, Verbesserungsvorschläge, Fragen


Gilb, Graham, Software Inspection, 1993
Optimal reading speed
1 ± 0.8 pages per hour

3. Gruppensitzung (Dauer: 2h)

- Individuell gefundene Problempunkte sammeln
- Jeden einzelnen Problempunkt besprechen
- Fragen zum Dokument klären
- Weitere Problempunkte sammeln, die während der Diskussion gefunden werden
- Verbesserungsvorschläge zur Durchführung von Inspektionen sammeln

Alternative Meinung: **Nicht diskutieren!**

4. Nachbereitung

- Liste mit allen Problempunkten wird an den Editor des Dokuments weitergeleitet
- Editor identifiziert tatsächliche Defekte und klassifiziert sie
- Editor leitet Änderung des Dokuments ein
- Alle Problempunkte werden bearbeitet
- Deren Bearbeitung wird überprüft
- Schätze Restdefekte
  - # nicht entdeckte Defekte ≈ # entdeckte Defekte
  - 1/6 der Korrekturen ist fehlerhaft/verursacht neuen Defekt
- Dokument wird freigeben wenn #geschätzte Def. < N

5. Prozessverbesserung

- Prüflisten und Szenarien anpassen
- Standards für Dokumente erarbeiten
- Defekt-Klassifikationsschema anpassen
- Formulare verbessern
- Planung und Durchführung verbessern

Rollen

- **Inspektionsleiter**: leitet alle Phasen der Inspektion
- **Moderator**: leitet die Gruppensitzung (meist der Inspektionsleiter)
- **Inspektoren**: prüfen das Dokument
- **Schriftführer**: protokolliert Defekte in der Gruppensitzung
- **Editor**: klassifiziert und behebt die Defekte (meist der Autor)
- **Autor**: hat das Dokument verfasst

Formular für Problempunkte (typisch)

- Stelle: Seite und Zeile
- Beschreibung
- Wichtigkeit : hoch, gering
- Problemtyp: Defekt, Anregung, Frage
- Defekttyp
- behoben: ja, nein
- nachgeprüft : ja, nein

Defektklassifikation

- schwerwiegend (engl. *major*) oder leicht (engl. *minor*)
- Defekt (engl. *defect*), Anregung (engl. *suggestion*) oder Frage (engl. *question*)
- Defekttyp, z.B. nach NASA SEL (engl. *orthogonal defect classification*)
  - A ambiguous information
  - E extraneous information
  - II inconsistent information
  - IF incorrect fact
  - MI missing information
  - MD miscellaneous defect

Lesetechniken – Überblick (1)

- Ad-Hoc
- Prüflisten
  - Liste mit Fragen zum Dokument
  - Fragen sollen das Aufspüren von Defekten erleichtern
  - jede Frage bezieht sich auf ein einzelnes Qualitätsmerkmal
  - Fragen müssen „abgehakt“ werden
  - Prüflisten abhängig von Art des Dokumentes, verwendeter Programmiersprache, verwendete Bibliotheken (z.B. für Zugriff auf Datenbanken), …

Lesetechniken – Überblick (2)

- Szenarien
  - Anleitung, wie das Dokument zu prüfen ist
  - Satz von Fragen
  - Szenario deckt bestimmte Sichtweise oder Art von Defekten ab
  - verschiedene Szenarien für die Inspektoren
  - im Allgemeinen effektiver als Prüflisten

Lesetechniken – Prüflisten (1)

- **Richtlinien für Gestaltung von Prüflisten**
  - Liste höchstens eine Seite lang und gegliedert
  - Möglichst präzise Fragen stellen
  - Fragen sollen Hinweis geben, auf welche Weise hohe Qualität erreicht werden kann
  - Fragen vermeiden, die automatisch geprüft werden können
  - Fragen auf neuestem technischen Stand halten

Lesetechniken – Prüflisten (2)

- **Typische Probleme**
  - Fragen nicht auf die Anwendung zugeschnitten
  - Fragen fördern Programmverständnis nicht
  - keine Hilfestellung, wie die Prüfliste abgearbeitet werden soll
  - ungeklärt, welche Zusatzinformationen benötigt werden
  - Fragen decken nicht alle Arten von Defekten ab

Lesetechniken – Prüflisten: Beispiel (1)

- Eine mögliche Prüfliste für Java-Code könnte nach folgenden Defektklassen gegliedert sein:
  1. Variablen, Attribute und Konstantendeklarationen
  2. Methodendefinitionen
  3. Klassendefinitionen
  4. Datenreferenzen
  5. Berechnungen
  6. Vergleiche
  7. Kontrollfluss
  8. Ein-/Ausgabe
  9. Modulschnittstelle
  10. Kommentare
  11. Codeformatierung
  12. Modularität
  13. Speicherverwendung
  14. Ausführungsgeschwindigkeit

Prüfliste von Christopher Fox, 1999.

Lesetechniken – Prüflisten: Beispiel (2)

1. Variablen, Attribute und Konstantendeklarationen
   1. Sind die Variablen-, Attribut- und Konstantennamen aussagekräftig?
   2. Stimmen die Bezeichner mit den Programmierrichtlinien überein?
   3. Gibt es Variablen oder Attribute mit ähnlichen Namen (Verwechslungsgefahr)?
   4. Wurde jede Variable und jedes Attribut korrekt typisiert?
   5. Wurde jede Variable und jedes Attribut korrekt initialisiert?
   6. […]
2. Methodendefinitionen
   1. Sind die Methodennamen aussagekräftig?
   2. Stimmen die Methodennamen mit den Programmierrichtlinien überein?
   3. Wird jeder Methodenparameter vor der Verwendung überprüft?
   4. Gibt die Methode mit jedem `return` den korrekten Wert zurück?
   5. Gibt es statische Methoden, welche nicht statisch sein sollten (und umgekehrt)?
   6. […]

Es können zu jeder Klasse weitere Prüffragen ergänzt werden.

Lesetechniken – Prüflisten: Beispiel (3)

3. Klassendefinitionen
   1. Besitzt jede Klasse geeignete Konstruktoren?
   2. Besitzt eine Klasse Instanzvariablen, welche besser in eine Oberklasse passen?
   3. Kann die Klassenhierarchie vereinfacht werden?
   4. […]
4. Datenreferenzen
   1. Für jede Referenz auf ein Feld: Ist jeder Index innerhalb der erlaubten Grenzen?
   2. Für jeder Objektreferenz oder Referenz auf ein Feld: Ist der Wert garantiert nie `NULL`?
   3. […]

Lesetechniken – Prüflisten: Beispiel (4)

5. Berechnungen
   1. Gibt es Berechnungen mit verschiedenen Datentypen (Ganzzahlen/Fließkommazahlenkonvertierung)?
   2. Bei Ausdrücken mit mehr als einem Operator: Ist die Reihenfolge der Auswertung korrekt?
   3. […]
6. Vergleiche
   1. Für jeden booleschen Vergleich: Wird die korrekte Bedingung überprüft?
   2. Werden die korrekten Vergleichsoperatoren verwendet?
   3. Wurde jeder boolesche Ausdruck vereinfacht, in dem die Negierungen in den Ausdruck hineingezogen wurden?
   4. Ist jeder boolesche Ausdruck korrekt?
   5. […]

Lesetechniken – Prüflisten: Beispiel (5)

7. Kontrollfluss
   1. Für jede Schleife: Wurde das Beste Schleifen-Konstrukt gewählt?
   2. Terminieren alle Schleifen?
   3. Wenn es mehrere Ausgänge bei einer Schleife gibt: Sind diese notwendig und werden sie korrekt behandelt?
   4. Besitzt jede `switch`-Anweisung einen `default`-Fall?
   5. […]
8. Ein-/Ausgabe
   1. Werden alle Dateien vor der Verwendung geöffnet?
   2. Werden alle Dateien nach der Verwendung wieder geschlossen?
   3. Werden alle Ausnahmen (`IOException`) korrekt behandelt?
   4. […]

Lesetechniken – Prüflisten: Beispiel (6)

9. Modulschnittstelle
   1. Stimmen die Anzahl, Reihenfolge, Typisierung und der Wert der Parameter bei jedem Methodenaufruf mit der Methodendefinition überein?
   2. Stimmen die Einheiten der Parameter (Kilometer vs. Meilen)?
   3. […]
10. Kommentare
   1. Besitzt jede Methode, Klasse, Datei einen geeigneten Kommentar?
   2. Besitzt jedes Attribut, jede Variable und Konstante einen Kommentar?
   3. Stimmen die Kommentare mit dem kommentierten Code überein?
   4. Helfen die Kommentare dabei, den Code zu verstehen?
   5. […]

Lesetechniken – Prüflisten: Beispiel (7)

11. Codeformatierung
    1. Wird eine einheitliche Einrückung und eine einheitliche Codeformatierung verwendet ?
    2. Für jede Methode: Ist sie nicht mehr als 60 Zeilen lang?
    3. Für jedes Klasse: Ist es nicht länger als 600 Zeilen?
    4. […]
12. Modularität
    1. Besteht eine geringe Kopplung (engl. coupling) zwischen verschiedenen Klassen?
    2. Besteht eine hohe Zusammenhalt (engl. cohesion) innerhalb jeder Klasse?
    3. Wurden Java-Klassenbibliotheken dort verwendet, wo es sinnvoll ist?
    4. […]

Lesetechniken – Prüflisten: Beispiel (8)

13. Speicherverwendung
    1. Sind alle Felder groß genug?
    2. Werden Objekt- und Feldreferenzen auf `NULL` gesetzt, wenn sie nicht mehr verwendet werden?
    3. […]
14. Ausführungsgeschwindigkeit
    1. Können bessere Datenstrukturen oder effizientere Algorithmen verwendet werden?
    2. Muss der Wert jedes mal neu berechnet werden oder lohnt es sich, den Wert zwischenzuspeichern?
    3. Kann eine Berechnung aus einer Schleife herausgezogen werden?
    4. Kann eine kurze Schleife ausgerollt werden?
    5. […]

Lesetechniken – Szenarien (1)

- **Arten**
  - Defekt-basiert
    - Datentypkonsistenzszenarien
    - Fehlfunktionsszenarien
    - Fehlende Funktionalität
    - Mehrfache/mehrdeutige Funktionalität
    - Falsche Implementierung
  - Perspektiven-basiert
    - Erklärung der Perspektive und ihrer Ziele
    - Anleitung zum Durcharbeiten des Dokuments
    - Liste mit Fragen zu dem Dokument

Typische Perspektiven

- Benutzer
- Entwickler
- Wartung
- Risiken
- Entwerfer
- Tester
- Qualität

Lesetechniken – Szenarien – Beispiel (1)

Beispiel für Wartungs-Perspektive

Assume you are inspecting an operation from the perspective of a maintainer. The main goal of a maintainer is to ensure that the collaboration diagram is written in a way that can be easily changed and maintained. High quality, therefore, means the conformance to specified design guidelines (low coupling, high cohesion) and the minimization of complexity.

Locate the collaboration diagram and the pseudocode description for the operation. Examine the diagram and the descriptions to identify points that diverge from good design practice.

While following the instructions answer the following questions:

1. Are there any ways in which the number of objects or the number of messages could be reduced?
2. Are there any cycles of messages in the collaboration diagram?
3. Is there any way in which the control structure of the operation could be simplified?
4. Do the messages entering an object indicate the possibility of low cohesion (unrelated messages)?
5. Is there a particularly high number of messages between a pair of objects?

Lesetechniken – Szenarien – Beispiel (2)

Beispiel für Code-Analyse-Perspektive

Assume you have the role of a code analyst. As a code analyst you have to ensure that the right functionality is implemented in the code.

In doing so, take the code document and determine the functions that are implemented in this code module.
Determine the dependencies among these functions and document them in the form of a call graph.
Starting with the functions at the leaves of the call graph, determine the implemented operation of each function in the following manner. Identify (sequences of) assignment operations and highlight them. Determine the meaning of these (sequences of) assignment operations. Combine the (sequences of) assignment operations by taking into account conditions and loops. Determine the meaning of the larger structures.
Repeat this until you have determined the operation that is implemented in a function. Document the operation of each function.
Check for each function, whether your description matches the description that is given in code comments and the description in the specification. If differences exist, check whether there is a defect. Document each defect you detect on the defect report form.

While following the instructions, ask yourself the following questions:

1. Does the operation described in the code match the one described in the specification?
2. Are there operations described in the specification that are not implemented?
3. Is data (i.e., constants and variables) used in a correct manner ?
4. Are all the calculations performed in a correct manner?
5. Are interfaces between functions used correctly?

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest	✓		✓	✓	✓	
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Prüfprogramme

- Die statische Analyse einer Software findet während der Übersetzung des Quelltextes statt.
- Für die statische Analyse des Quelltextes gibt es spezielle Anwendungen und Plugins für viele Entwicklungsumgebungen.

Prüfprogramme

- Warnungen und Fehler
  - Werden von der Entwicklungsumgebung angezeigt. Darunter fallen z.B.:
    - Evtl. nicht initialisierte Variablen
    - Nicht erreichbare Anweisungen
    - Unnötige Anweisungen
- Programmierstil überprüfen
  - Tabulatoren anstelle von Leerzeichen verwendet
  - JavaDoc-Kommentare vergessen
  - Parameter nicht als final deklariert
- Fehler anhand von Fehler-Mustern finden
  - Mit Hilfe einer statischen Analyse können Fehler anhand bestimmter Muster gefunden werden
- Plugins für Entwicklungsumgebungen und Werkzeuge, welche solche Funktionalitäten bieten, werden in Kapitel 5.2 besprochen.

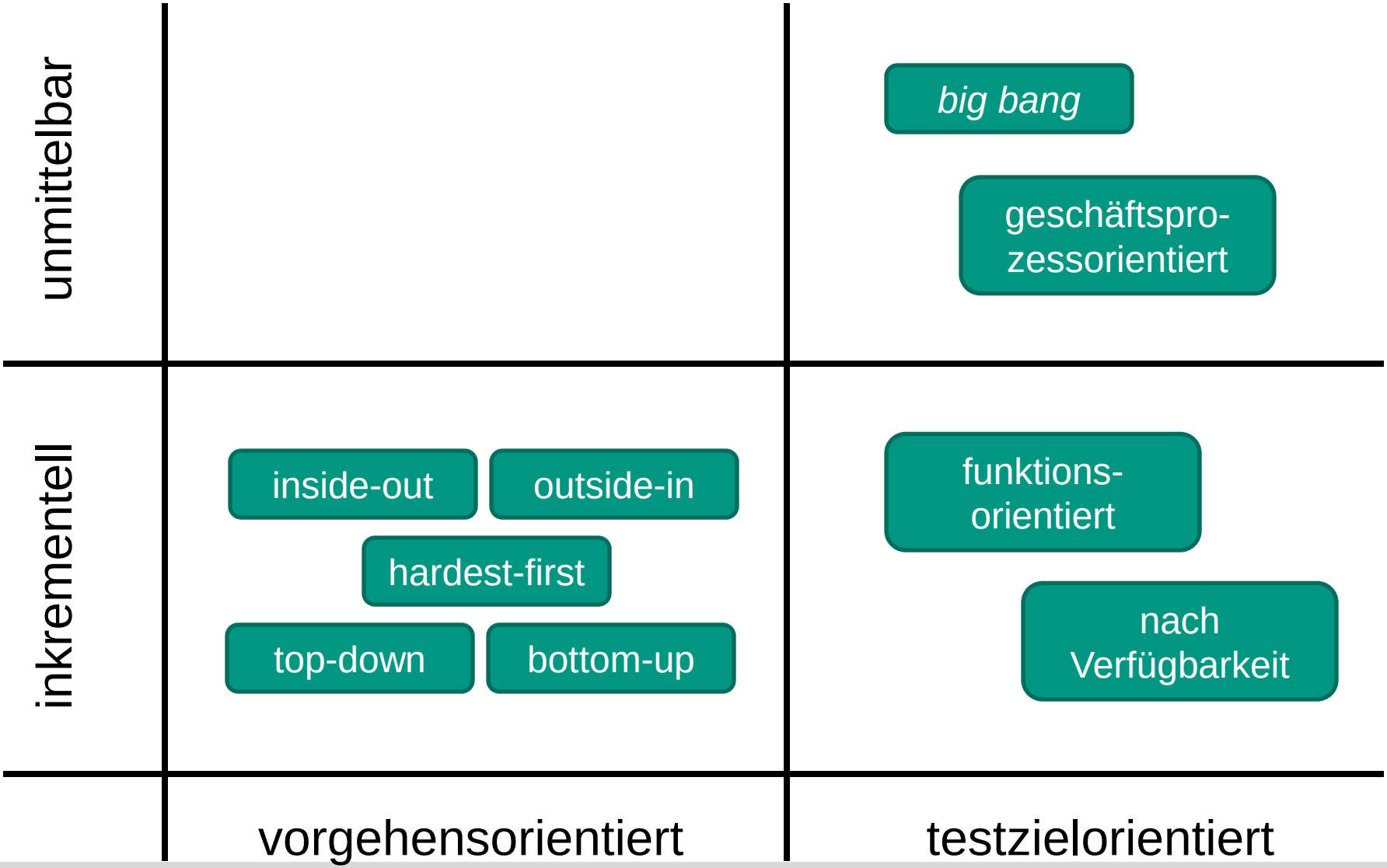
Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest						
Komponententest	✓		✓	✓	✓	✓
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

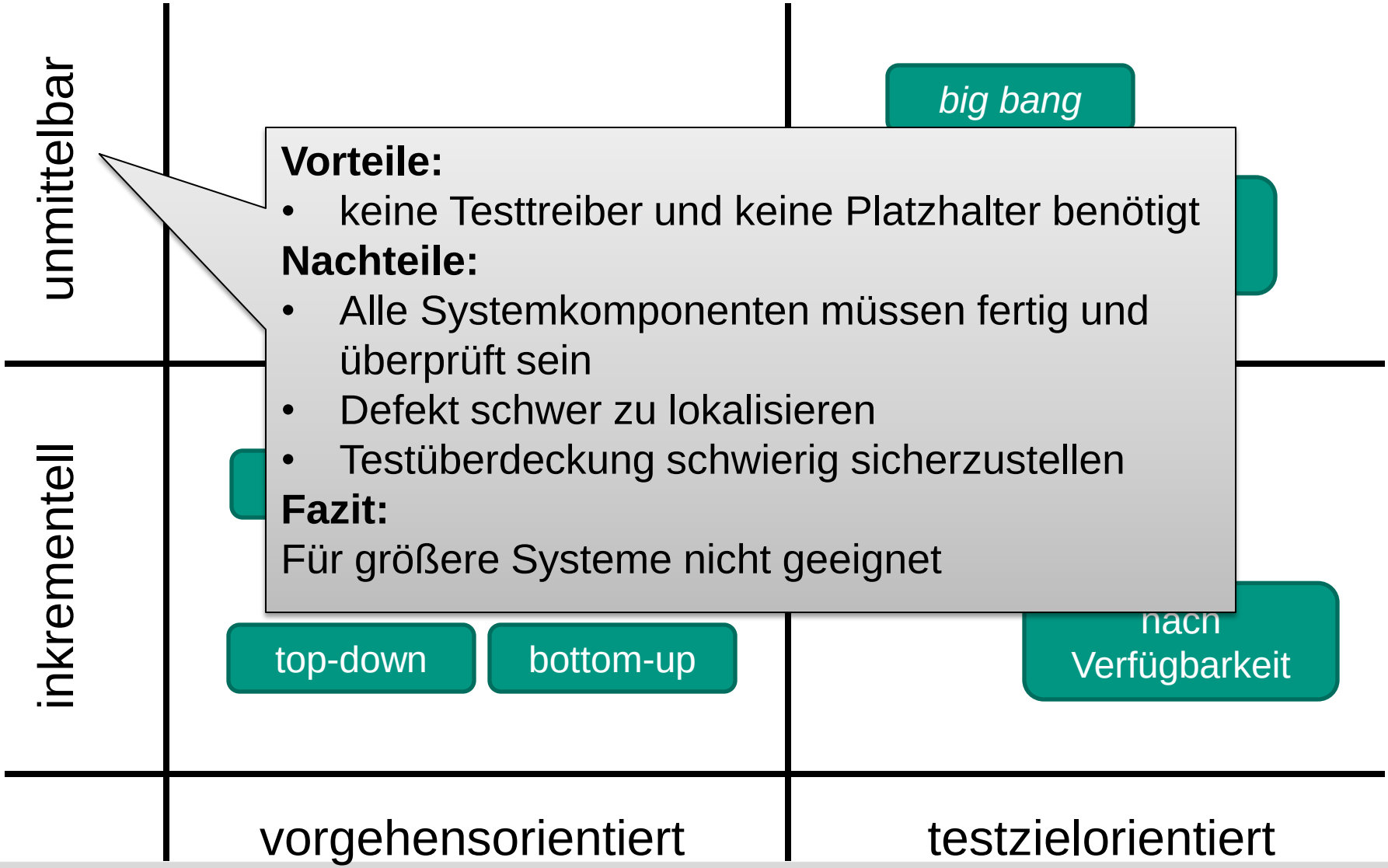
Integrationstest

- **Voraussetzung**: Jede involvierte Komponente wurde bereits für sich überprüft
- Schrittweise Integration
  - Integriere eine weitere Komponente in die bereits im Zusammenspiel geprüfte Komponentenmenge
  - … und prüfe erneut
  - … bis das System komplett integriert ist.
  - getestet wird das Zusammenspiel der Komponenten
  - Stummel und Attrappen werden mit den Implementierungen ersetzt

Integrationsstrategien



Integrationsstrategien

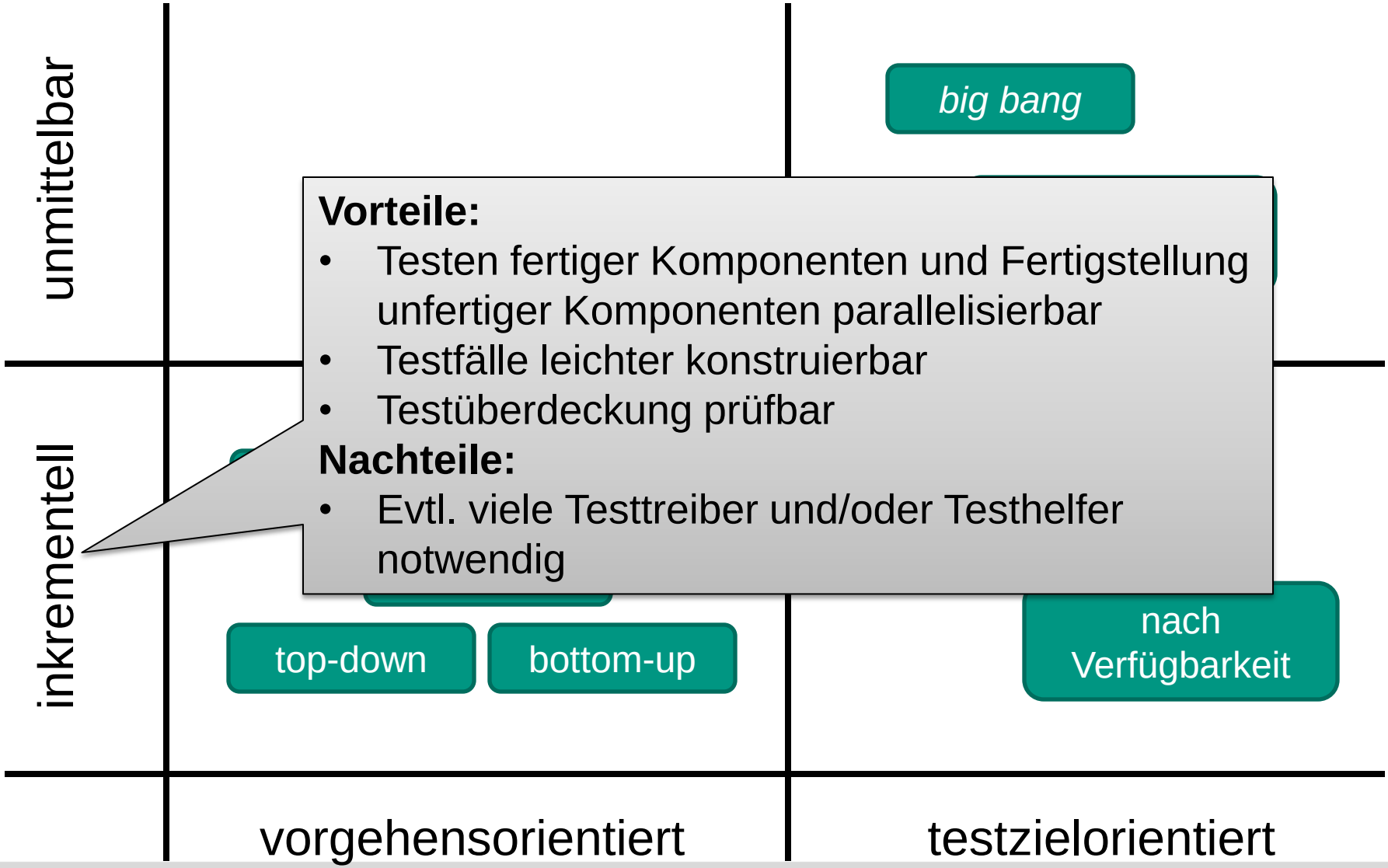


Vorteile:
- keine Testtreiber und keine Platzhalter benötigt

Nachteile:
- Alle Systemkomponenten müssen fertig und überprüft sein
- Defekt schwer zu lokalisieren
- Testüberdeckung schwierig sicherzustellen

Fazit:
Für größere Systeme nicht geeignet

Integrationsstrategien



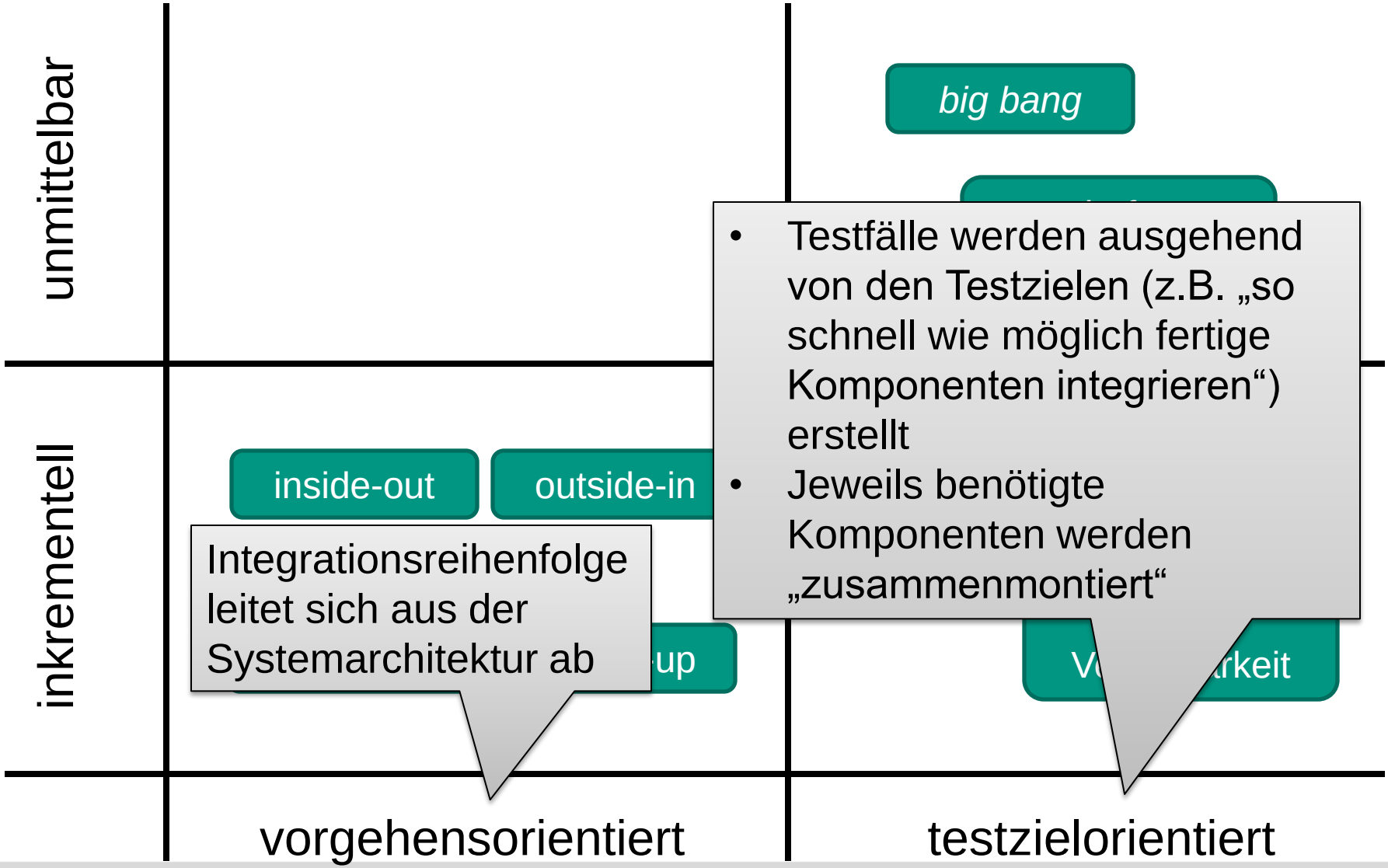
Vorteile:

- Testen fertiger Komponenten und Fertigstellung unfertiger Komponenten parallelisierbar
- Testfälle leichter konstruierbar
- Testüberdeckung prüfbar

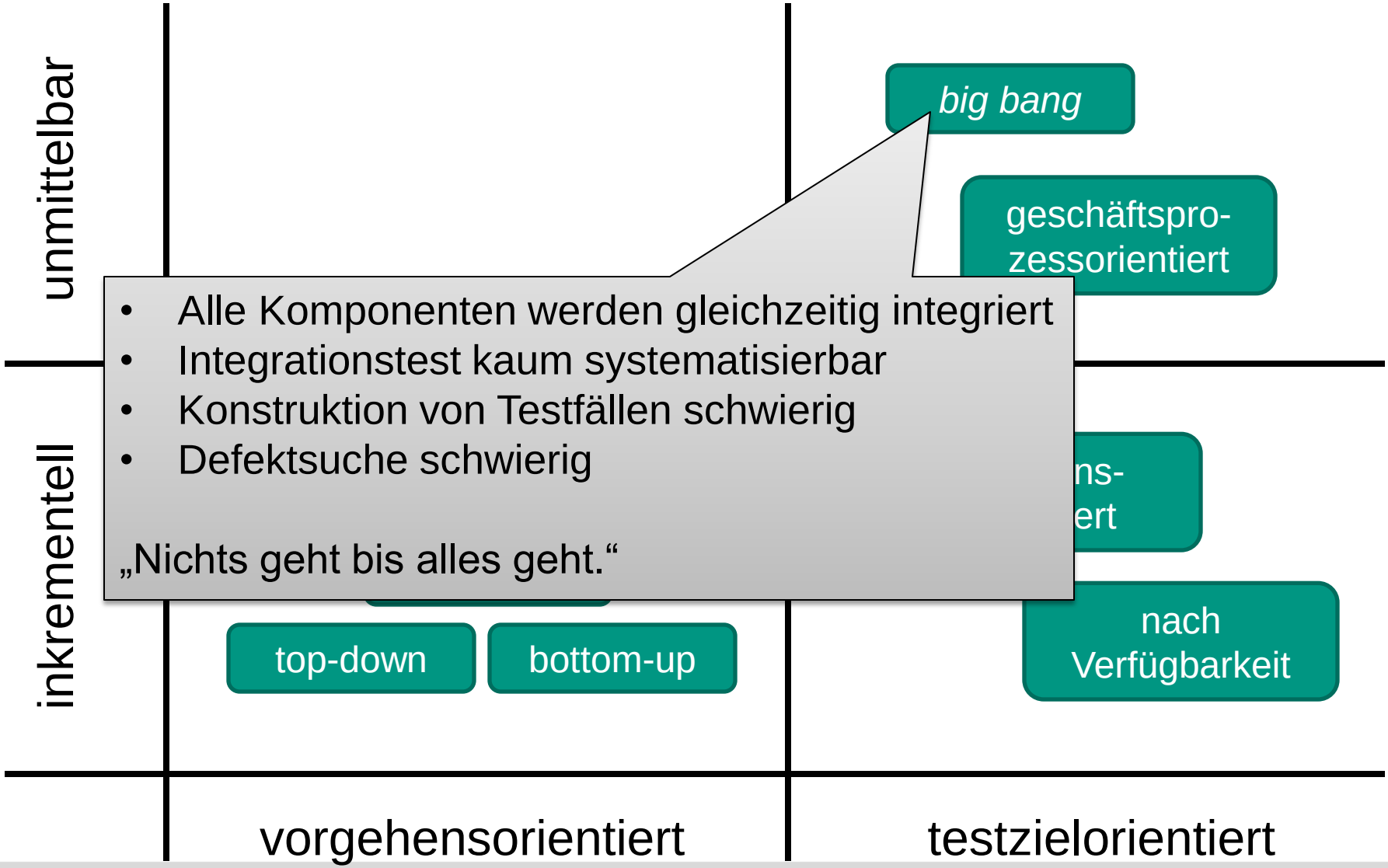
Nachteile:

- Evtl. viele Testtreiber und/oder Testhelfer notwendig

Integrationsstrategien



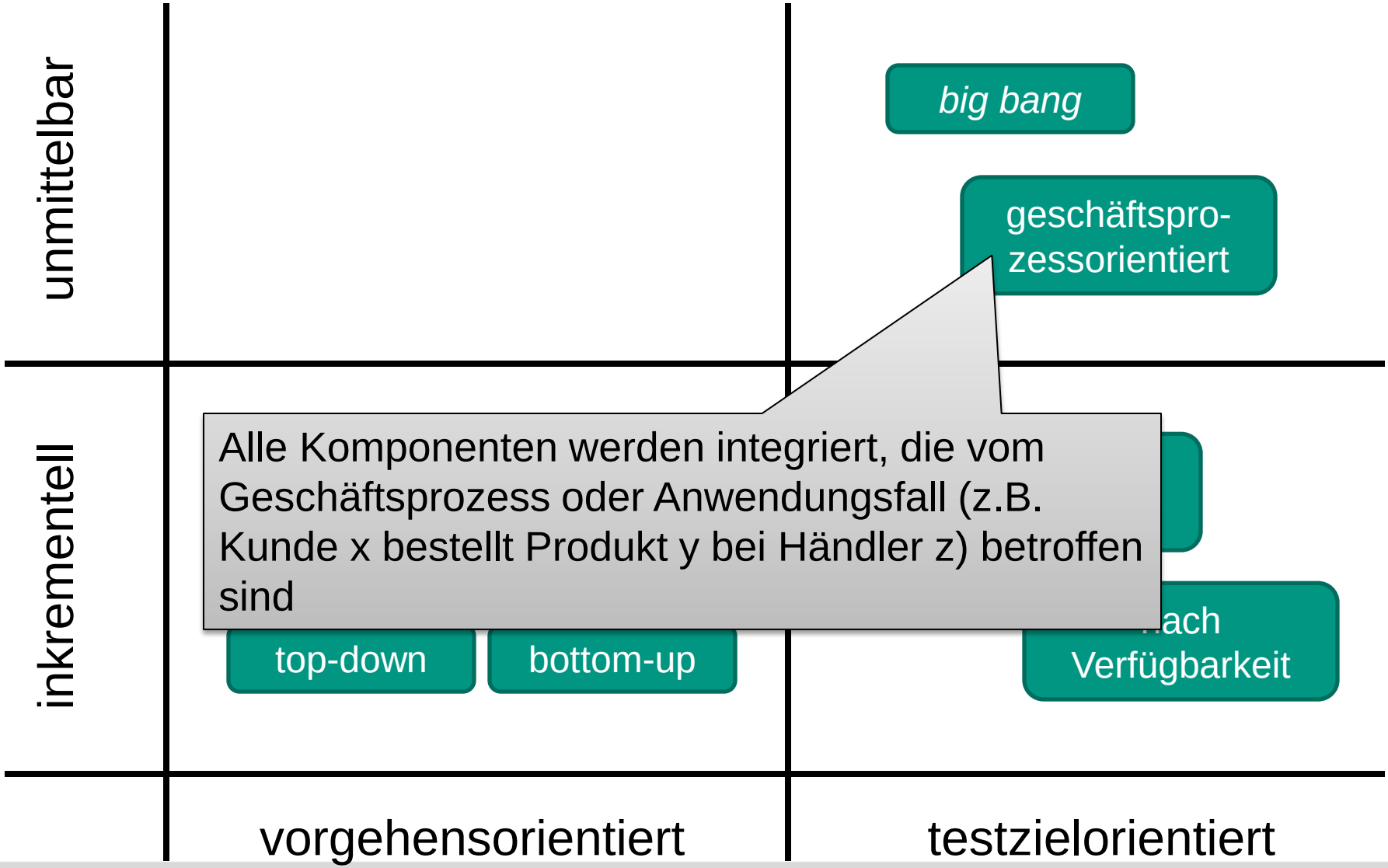
Integrationsstrategien



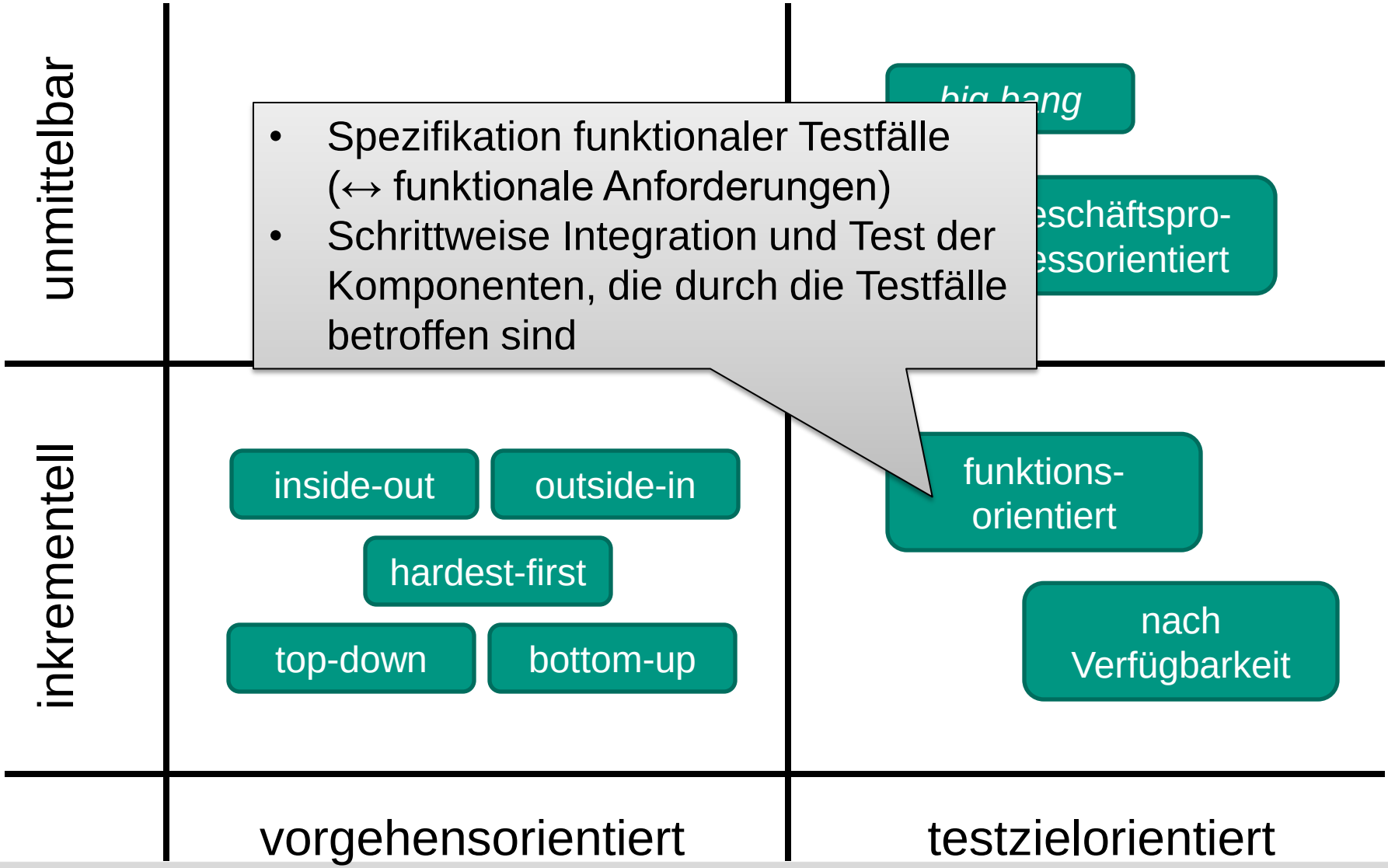
- Alle Komponenten werden gleichzeitig integriert
- Integrationstest kaum systematisierbar
- Konstruktion von Testfällen schwierig
- Defektsuche schwierig

„Nichts geht bis alles geht.“

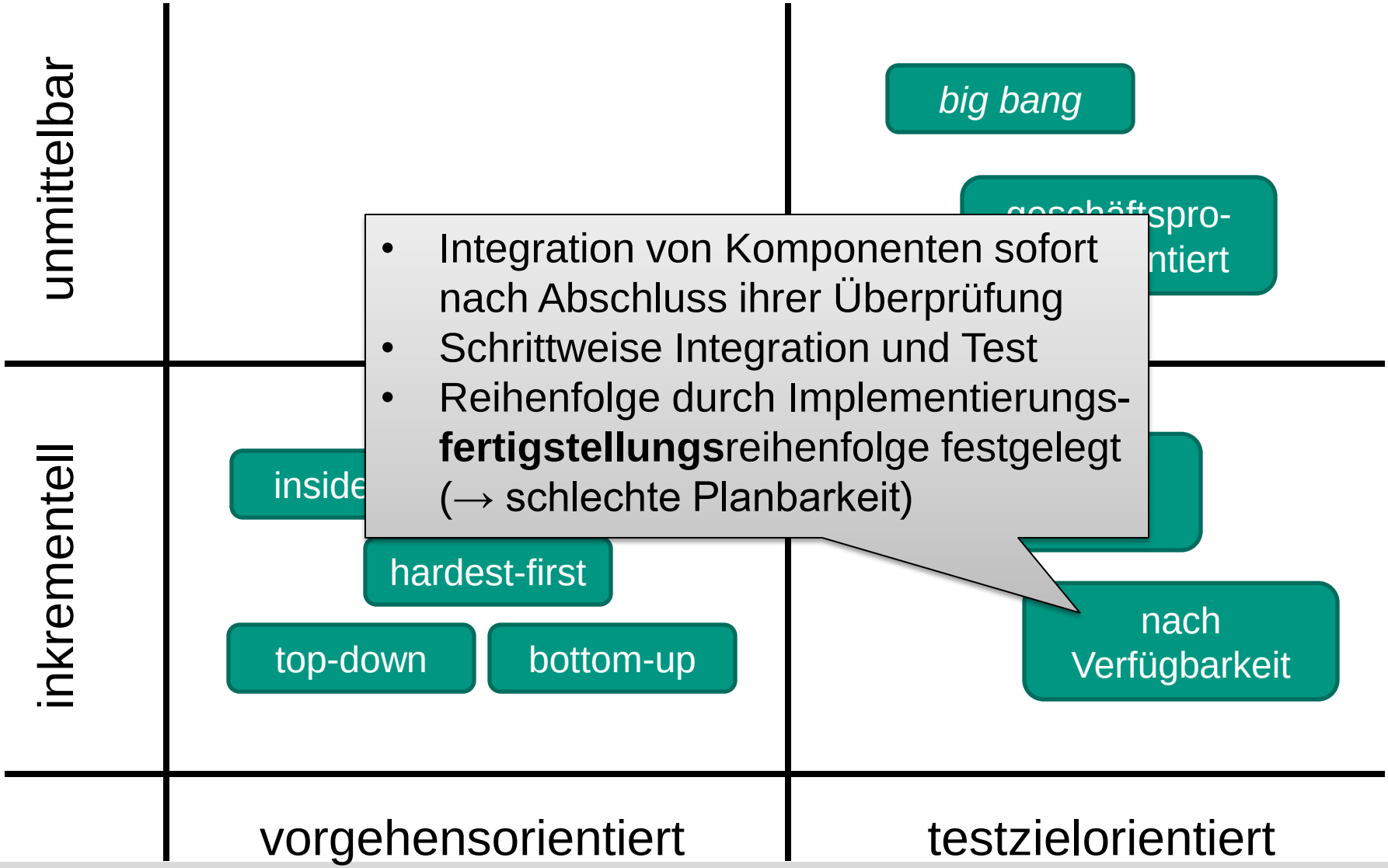
Integrationsstrategien



Integrationsstrategien

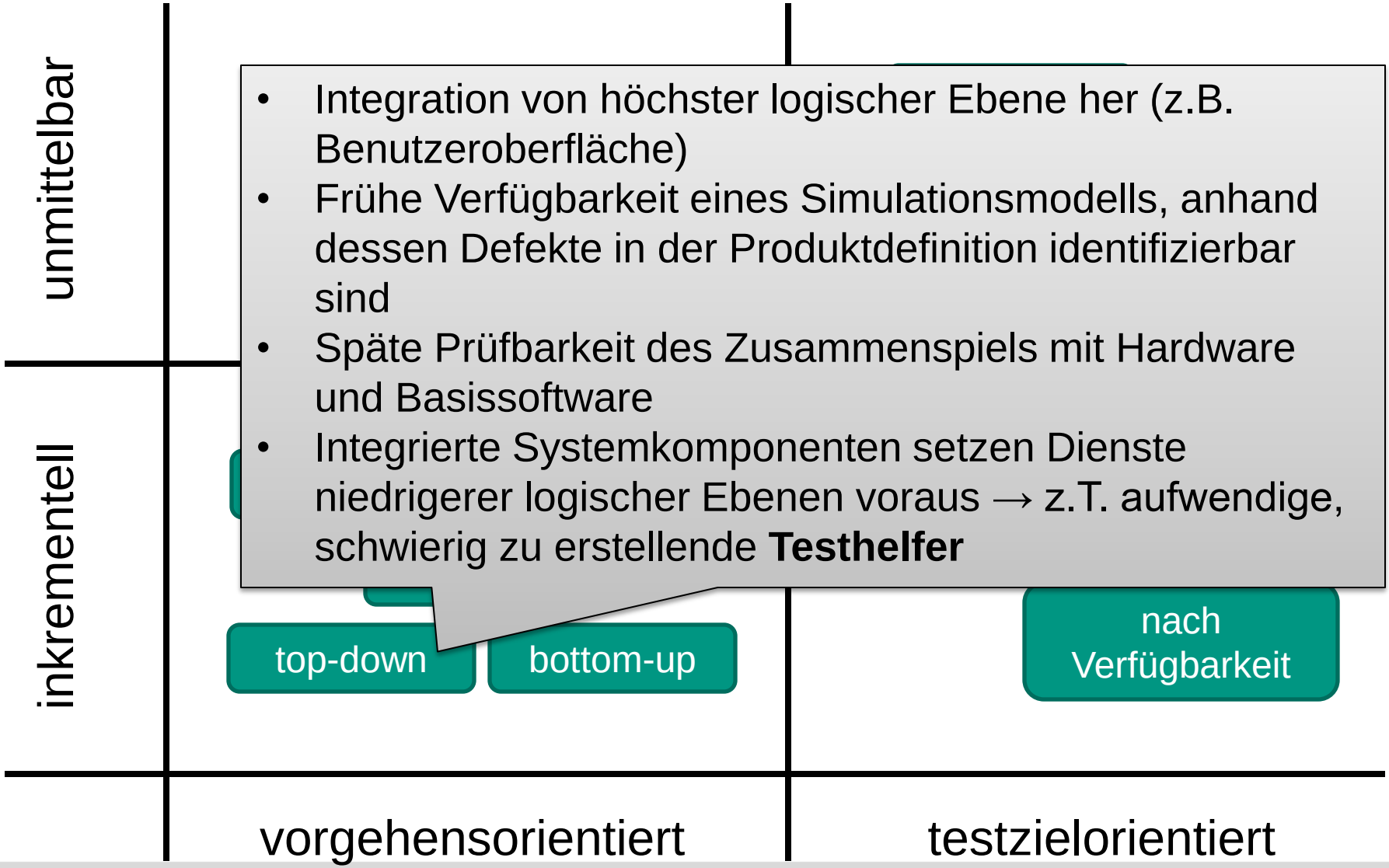


Integrationsstrategien



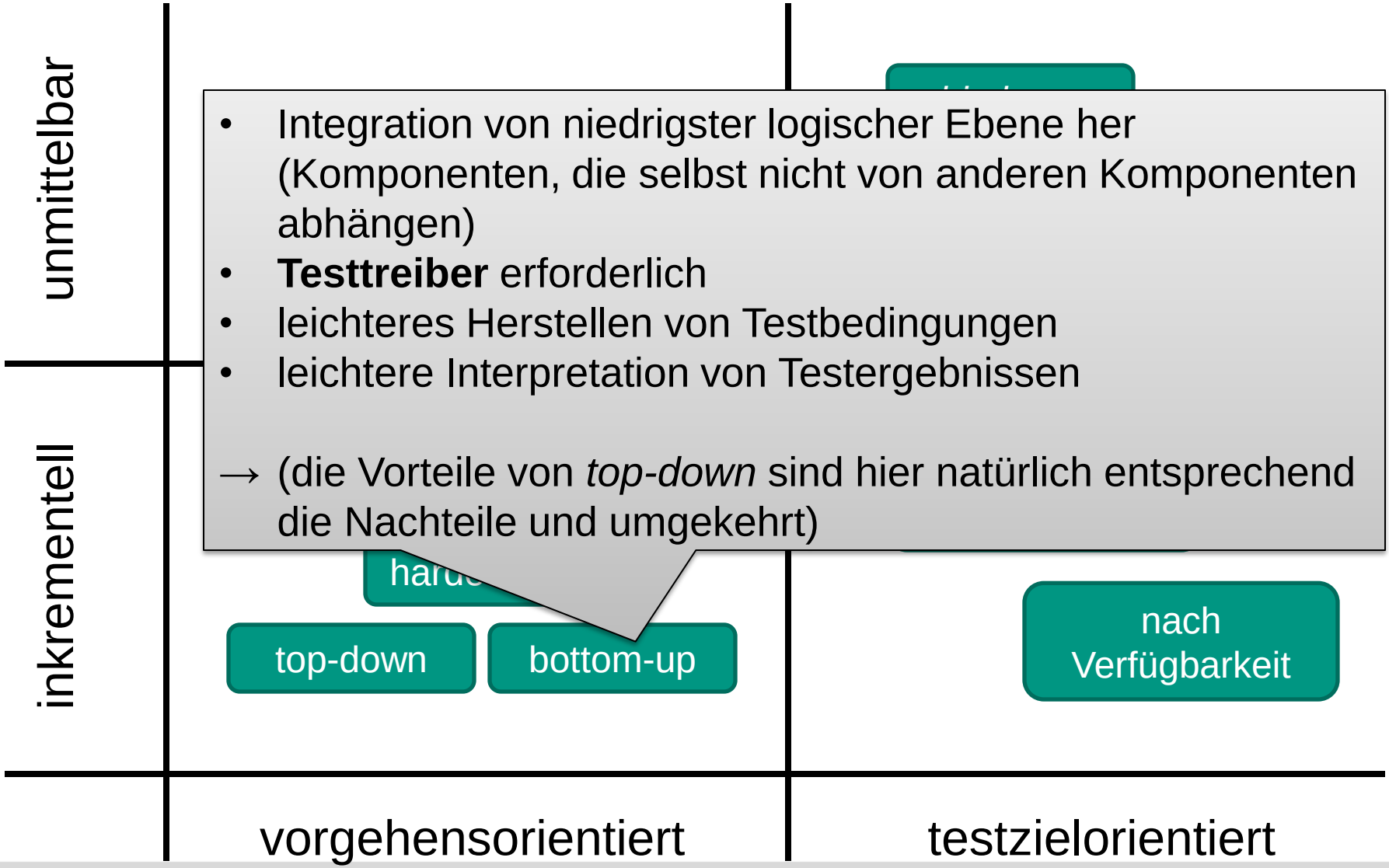
- Integration von Komponenten sofort nach Abschluss ihrer Überprüfung
- Schrittweise Integration und Test
- Reihenfolge durch Implementierungs-**fertigstellungs**reihenfolge festgelegt (→ schlechte Planbarkeit)

Integrationsstrategien



- Integration von höchster logischer Ebene her (z.B. Benutzeroberfläche)
- Frühe Verfügbarkeit eines Simulationsmodells, anhand dessen Defekte in der Produktdefinition identifizierbar sind
- Späte Prüfbarkeit des Zusammenspiels mit Hardware und Basissoftware
- Integrierte Systemkomponenten setzen Dienste niedrigerer logischer Ebenen voraus → z.T. aufwendige, schwierig zu erstellende **Testhelfer**

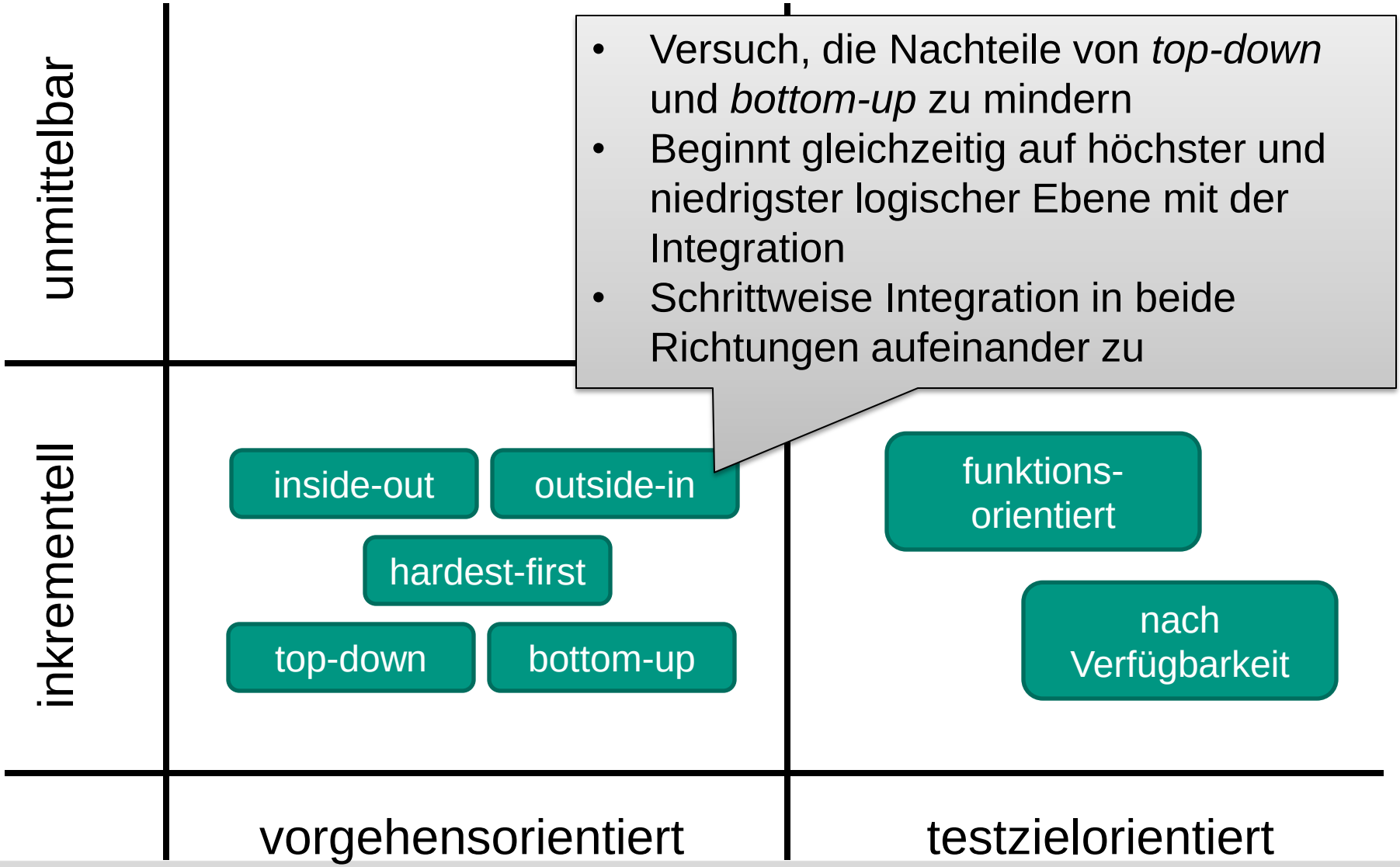
Integrationsstrategien



- Integration von niedrigster logischer Ebene her (Komponenten, die selbst nicht von anderen Komponenten abhängen)
- **Testtreiber** erforderlich
- leichteres Herstellen von Testbedingungen
- leichtere Interpretation von Testergebnissen

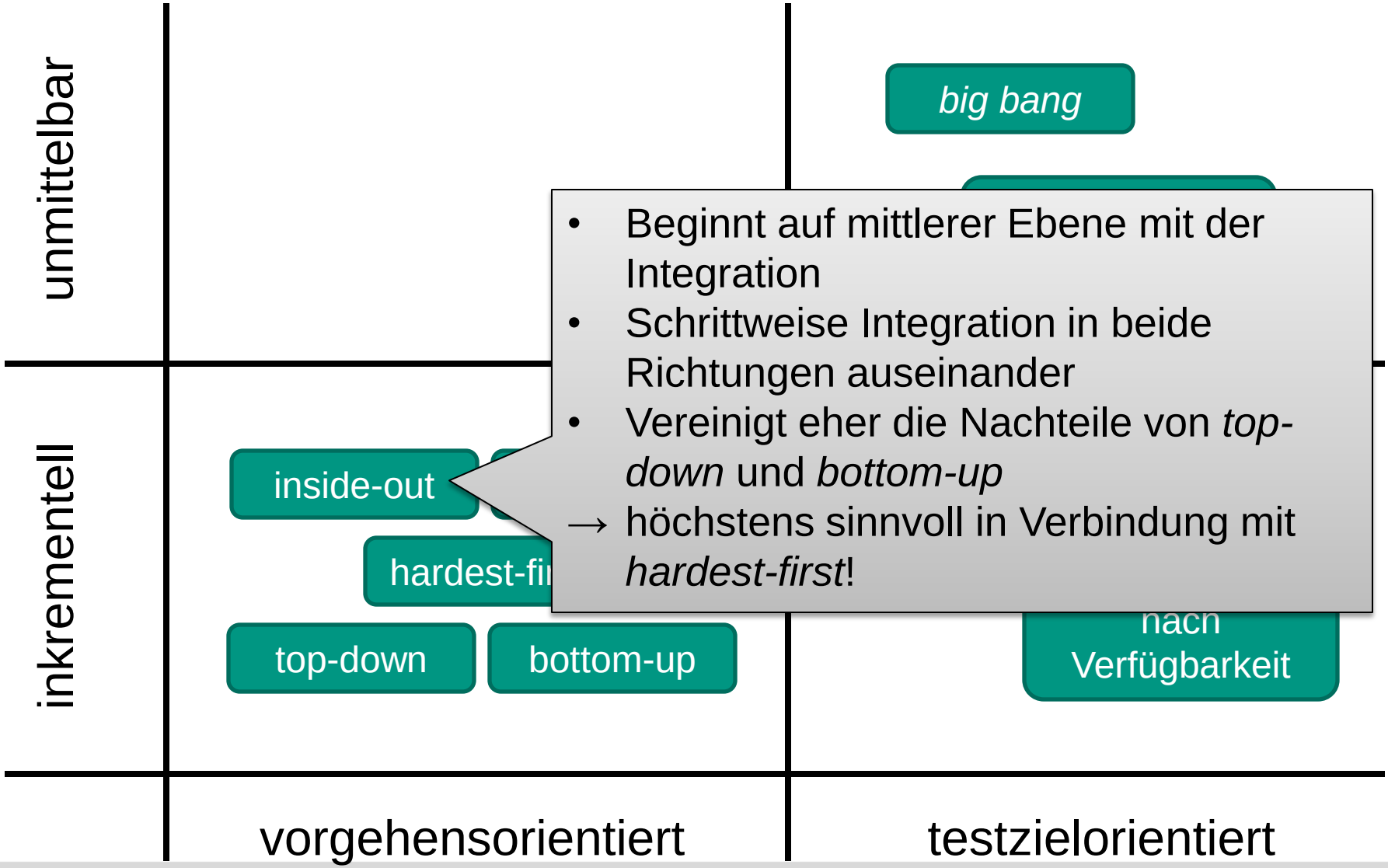
→ (die Vorteile von *top-down* sind hier natürlich entsprechend die Nachteile und umgekehrt)

Integrationsstrategien



- Versuch, die Nachteile von *top-down* und *bottom-up* zu mindern
- Beginnt gleichzeitig auf höchster und niedrigster logischer Ebene mit der Integration
- Schrittweise Integration in beide Richtungen aufeinander zu

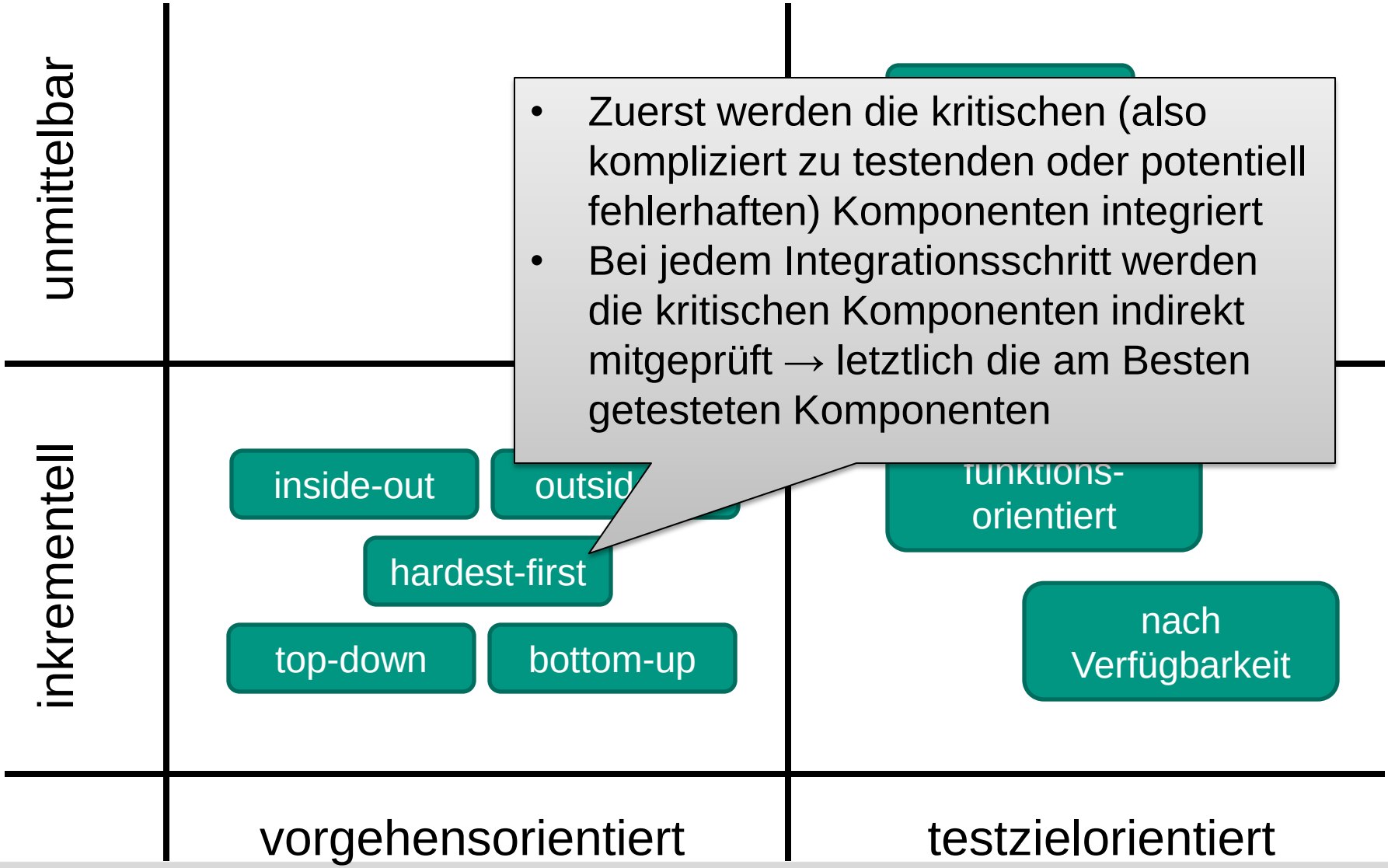
Integrationsstrategien



- Beginnt auf mittlerer Ebene mit der Integration
- Schrittweise Integration in beide Richtungen auseinander
- Vereinigt eher die Nachteile von *top-down* und *bottom-up*

→ höchstens sinnvoll in Verbindung mit *hardest-first*!

Integrationsstrategien



- Zuerst werden die kritischen (also kompliziert zu testenden oder potentiell fehlerhaften) Komponenten integriert
- Bei jedem Integrationsschritt werden die kritischen Komponenten indirekt mitgeprüft → letztlich die am Besten getesteten Komponenten

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest						
Integrationstest	✓	✓	✓	✓	✓	✓
Komponententest	✓		✓	✓	✓	✓
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Systemtest

- Prüfung des Komplettsystems gegen die Produktdefinition
- System als „*black box*“ – nur die äußere Systemansicht (Benutzerschnittstelle, externe Schnittstellen)
- Reale (realistische) Umgebung, z.B. eingebettetes System, technische Anlage, Bedienfeld, etc.

Klassifikation der Systemtests

- Der Systemtest wird in den funktionalen und den nicht funktionalen Systemtest eingeteilt.
  - **Funktionaler Systemtest:** Überprüfung der funktionalen Qualitätsmerkmale Korrektheit und Vollständigkeit.
  - **Nichtfunktionaler Systemtest:** Überprüfung nichtfunktionaler Qualitätsmerkmale, wie z.B.
    - Sicherheit
    - Benutzbarkeit
    - Interoperabilität
    - Dokumentation
    - Ausfallsicherheit

…und natürlich Leistungstests!

Definition: Regressionstest

- Ein **Regressionstest** ist die Wiederholung eines bereits vollständig durchgeführten Systemtests, z.B. aufgrund von Pflege, Änderung und Korrektur des betrachteten Systems.
- Zweck: sicherstellen, dass das System nicht in einen schlechteren Zustand als vorher zurückgefallen ist.
- Zur Vereinfachung der Testauswertung werden die Ergebnisse des Regressionstests mit den Ergebnissen des vorausgegangenen Tests verglichen.

Übersichtsmatrix: Was kommt im Folgenden?

Phase						
Abnahmetest						
Systemtest	✓	✓	✓	✓	✓	✓
Integrationstest	✓	✓	✓	✓	✓	✓
Komponententest	✓		✓	✓	✓	✓
Verfahren	Kontrollfluss-orientiert	Datenfluss-orientiert	Funktionale Tests	Leistungstests	Manuelle Prüfmethoden	Prüf-programme

Abnahmetests

- Spezieller Systemtest, bei dem
  - der Kunde beobachtet, mitwirkt und/oder Feder führt,
  - die reale Einsatzumgebung beim Kunden verwendet wird
  - und nach Möglichkeit echte Daten des Auftraggebers verwendet werden
- Der Auftraggeber kann Testfälle des Systemtests übernehmen und/oder modifizieren und/oder eigene Testszenarien durchführen, aber …
- … die formale Abnahme (für die der Abnahmetest die Grundlage ist) ist die bindende Erklärung der Annahme (im juristischen Sinne) eines Produkts durch den Auftraggeber

Literatur

- [Balzert98] H. Balzert. *Lehrbuch der Softwaretechnik*, 2. Band. Spektrum, 1998.
- [BrDu04] B. Bruegge, A.H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Pearson Prentice Hall, 2004, S. 435ff.
- [Lig01] P. Liggesmeyer. *Bedingungsüberdeckungstesttechniken: Vergleich, Bewertung und Anwendung in der Praxis* (Kurzfassung) unter http://pi.informatik.uni-siegen.de/stt/21_3/01_Fachgruppenberichte/217/8Liggesmeyer.pdf