universitätbonn ROOTS

Kapitel 10a
Testen

Stand: 11.1.2011

Teilweise nach:

Bernd Bruegge, Allen Dutoit:

„Object-Oriented Software Engineering – Conquering Complex and Changing Systems“,
Prentice Hall, 2-te Ausgabe, 2004

Übersicht

- Terminologie
- Fehlertypen
- Fehlerbehandlung
- Qualitätssicherung vs. Testen
- Teststrategien

- Komponententests
  - Unit-Tests
  - Integrationstests
- Systemtests
  - Funktionstests
  - Performanztests
  - Akzeptanztests
  - Installationstests

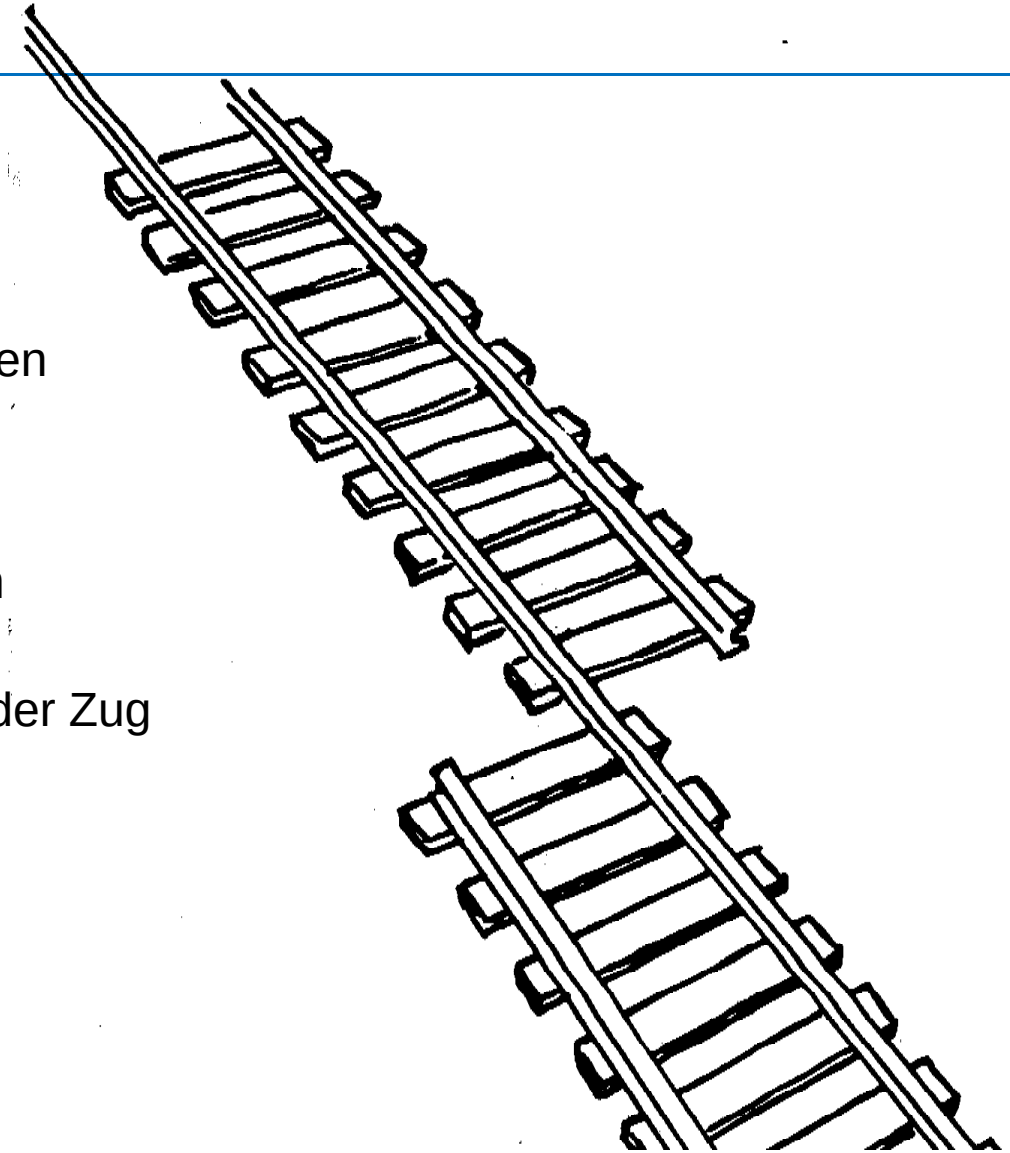
Terminologie

- **Versagen (*Failure*)**
  - Jedes Abweichen des beobachteten Verhaltens vom spezifizierten Verhalten eines Systems.
- **Fehlerhafter Zustand (*Error, erroneous state*)**
  - Das System befindet sich in einem Zustand, in dem ein Fortsetzen des Betriebs zu einem Versagen führen würde.
- **Fehlerursache (*Fault/Bug*)**
  - Die Mechanische oder algorithmische Ursache eines fehlerhaften Zustands
- **Zuverlässigkeit (*Reliability*)**
  - Ein Maß für den Grad an Übereinstimmung zwischen dem beobachtetem und dem spezifizierten Verhalten

Es gibt viele verschiedene Typen von Versagen und Arten damit umzugehen. $\Longrightarrow$

Was ist das?

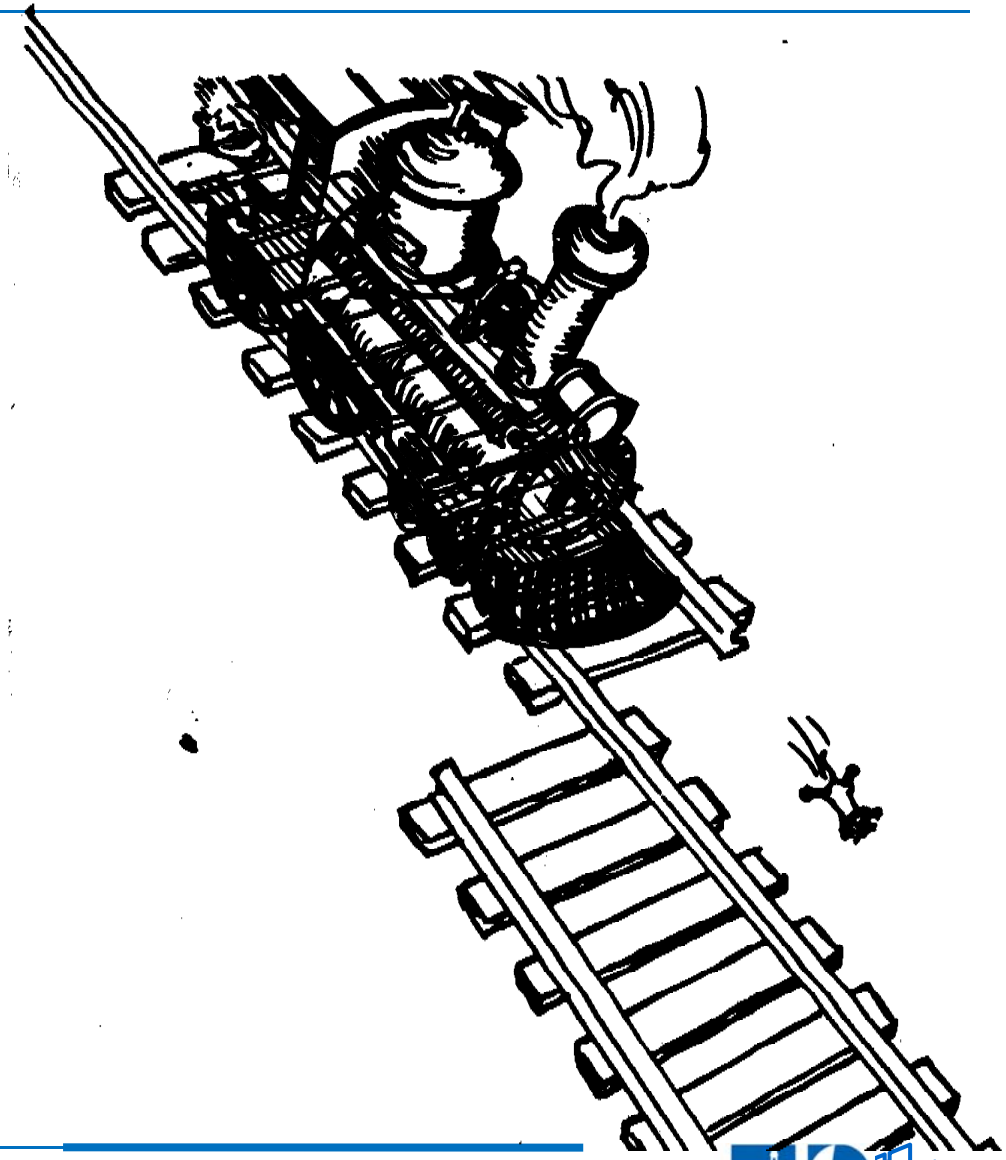
- Kein *Versagen*
  - Keine Spezifikation des erwarteten Verhaltens
  - Kein beobachtetes Verhalten
- Kein *Fehler(zustand)*
  - Kein Zustand der zu einem *Versagen* führen würde
  - Nur Schienen, kein fahrender Zug
- Keine *Fehlerursache*
  - Kein *Fehler*
    → keine *Fehlerursache*



Fehlerhafter Zustand (“Fehler”)

Angenommen es gäbe eine Spezifikation die besagt, dass der Zug von Gleis 1 auf Gleis 2 fahren soll

- Das beobachtete Verhalten während des Tests zeigt, dass diese Vorgabe nicht erfüllt wird
- Das Bild zeigt einen *Fehlerzustand*, der binnen kurzem zum *Versagen* des Systems führen wird
- Die Fehlausrichtung der Gleise kann als die *Fehlerursache* identifiziert werden



Algorithmische Mängel

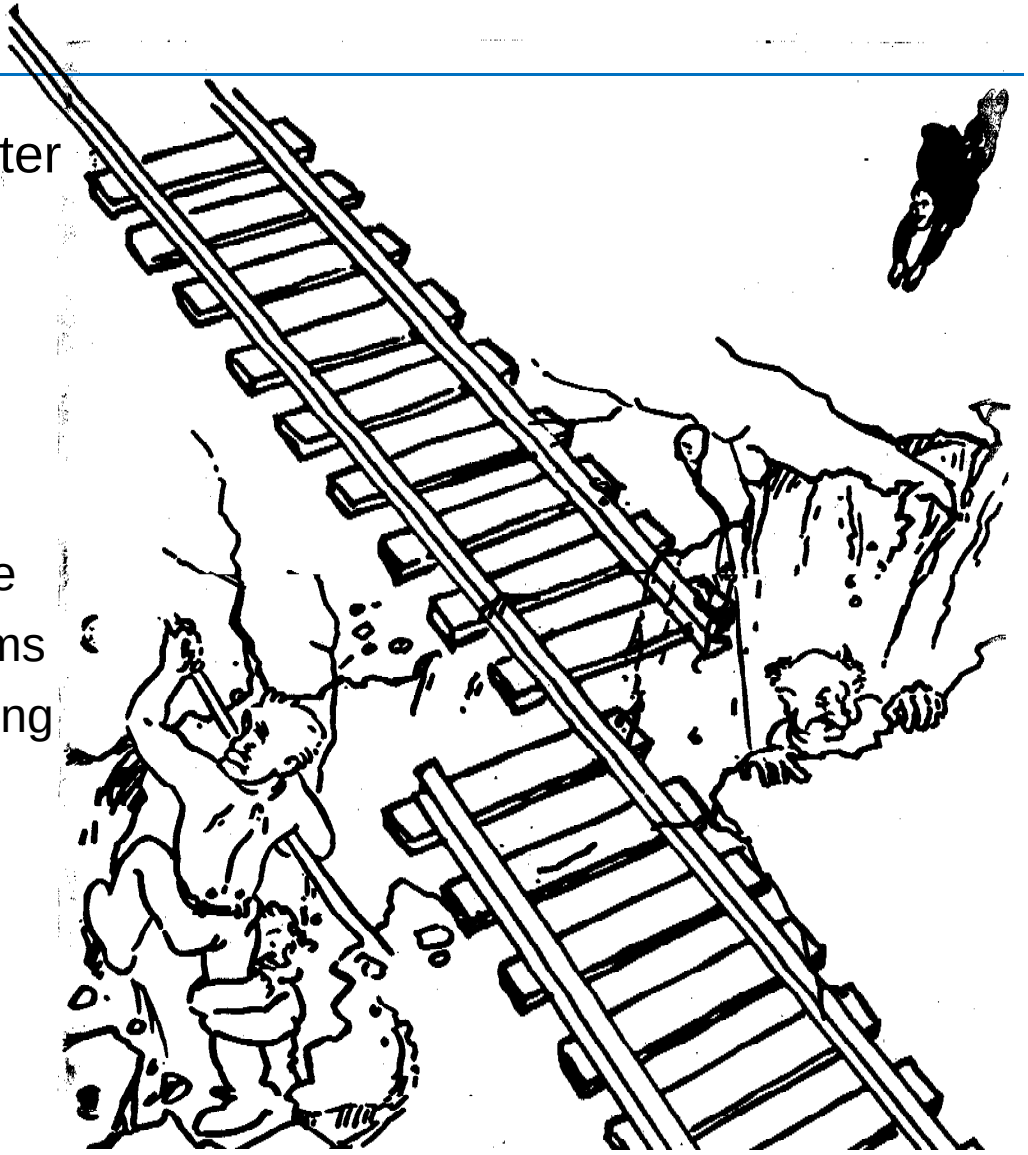
Algorithmische Mängel (2)

- Definition
  - Es wurde ein System entwickelt, das nicht seiner Spezifikation entspricht.

- Gründe
  - Missverständnisse unter den Entwicklern bzgl. der Spezifikation
  - Falsche Implementierung der Spezifikation
    - ⇨ Eine Schleife wird zu früh verlassen
    - ⇨ Eine Schleife wird zu spät verlassen
    - ⇨ Die falsche Bedingung wurde überprüft
    - ⇨ Eine Variable wurde nicht initialisiert
    - ⇨ ...
    - ⇨ Falsche Größe einer Datenstruktur
    - ⇨ Das System arbeitet langsamer als nicht-funktionalen Anforderungen gestatten.

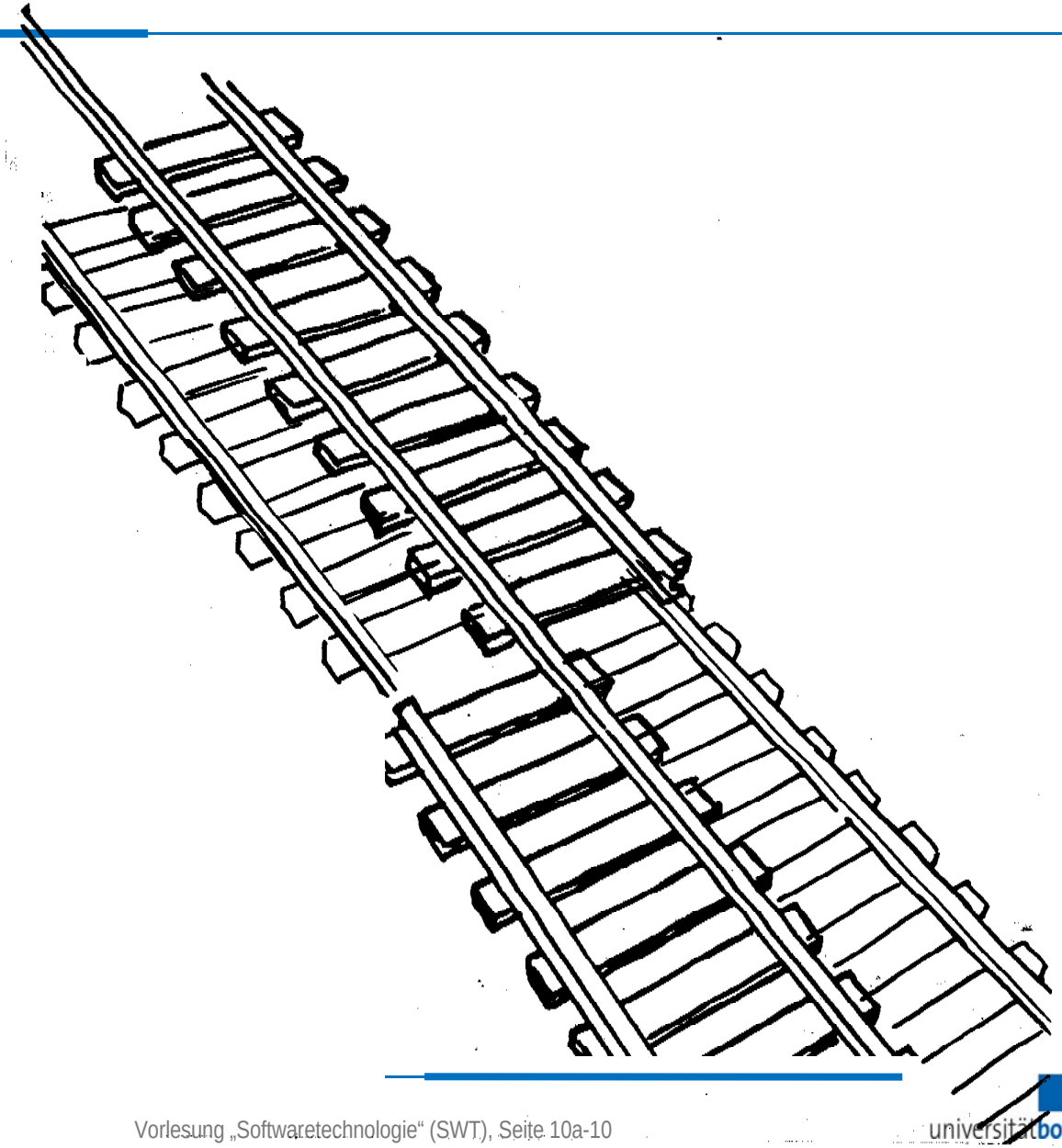
Mechanische Fehlerursachen

- Das System kann trotz korrekter Implementierung der Anwendung versagen ...
- ... aufgrund mechanischer Fehlerursachen
  - Erdbeben
  - Versagen der Virtual Machine
  - Versagen des Betriebssystems
  - Versagen der Stromversorgung
  - Brand
- Beachte
  - Versagen einer Komponente kann die Fehlerursache sein, die zum Versagen einer anderen Komponente führt

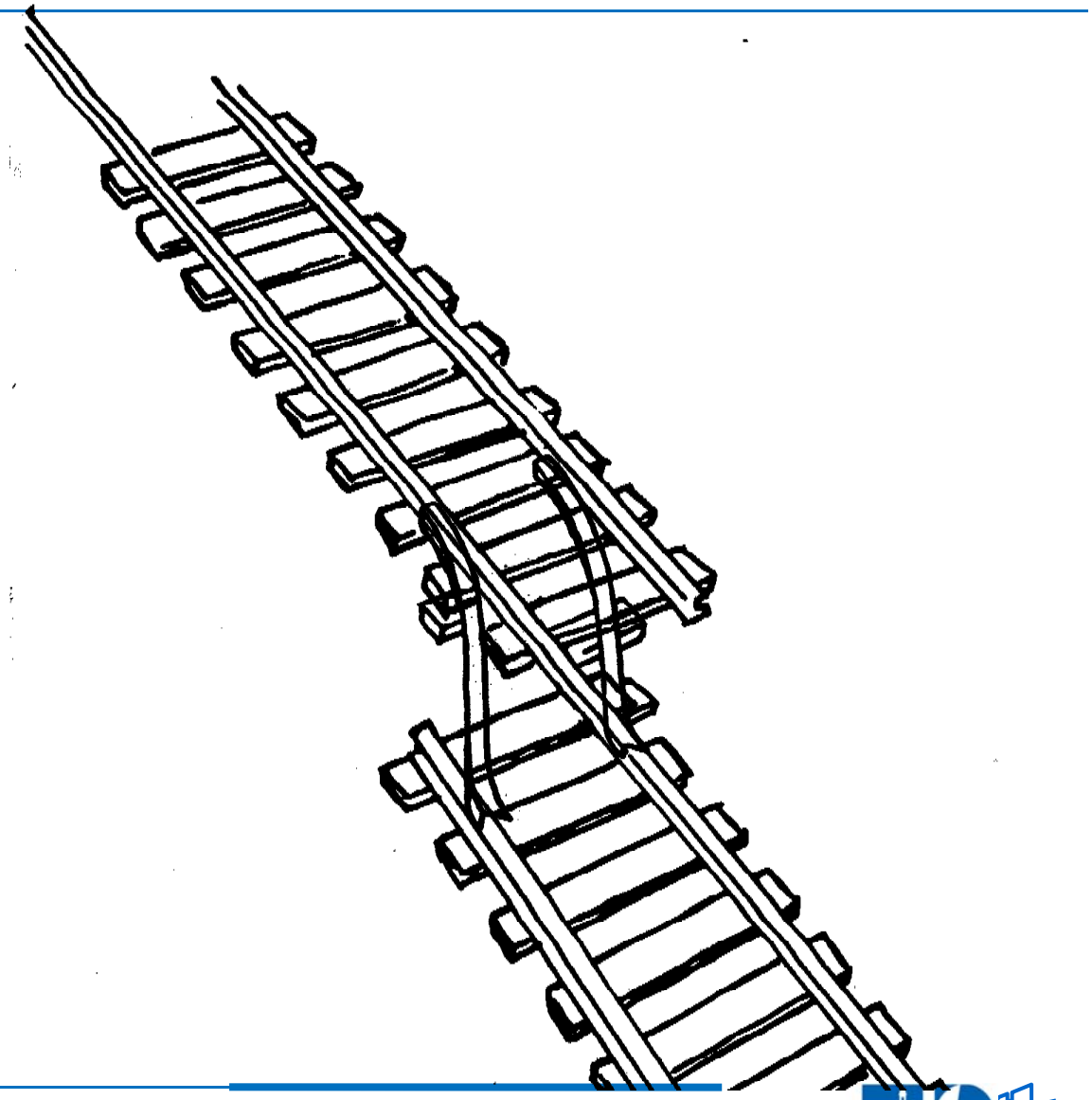


Wie soll mit Fehlern und Versagen umgegangen werden?

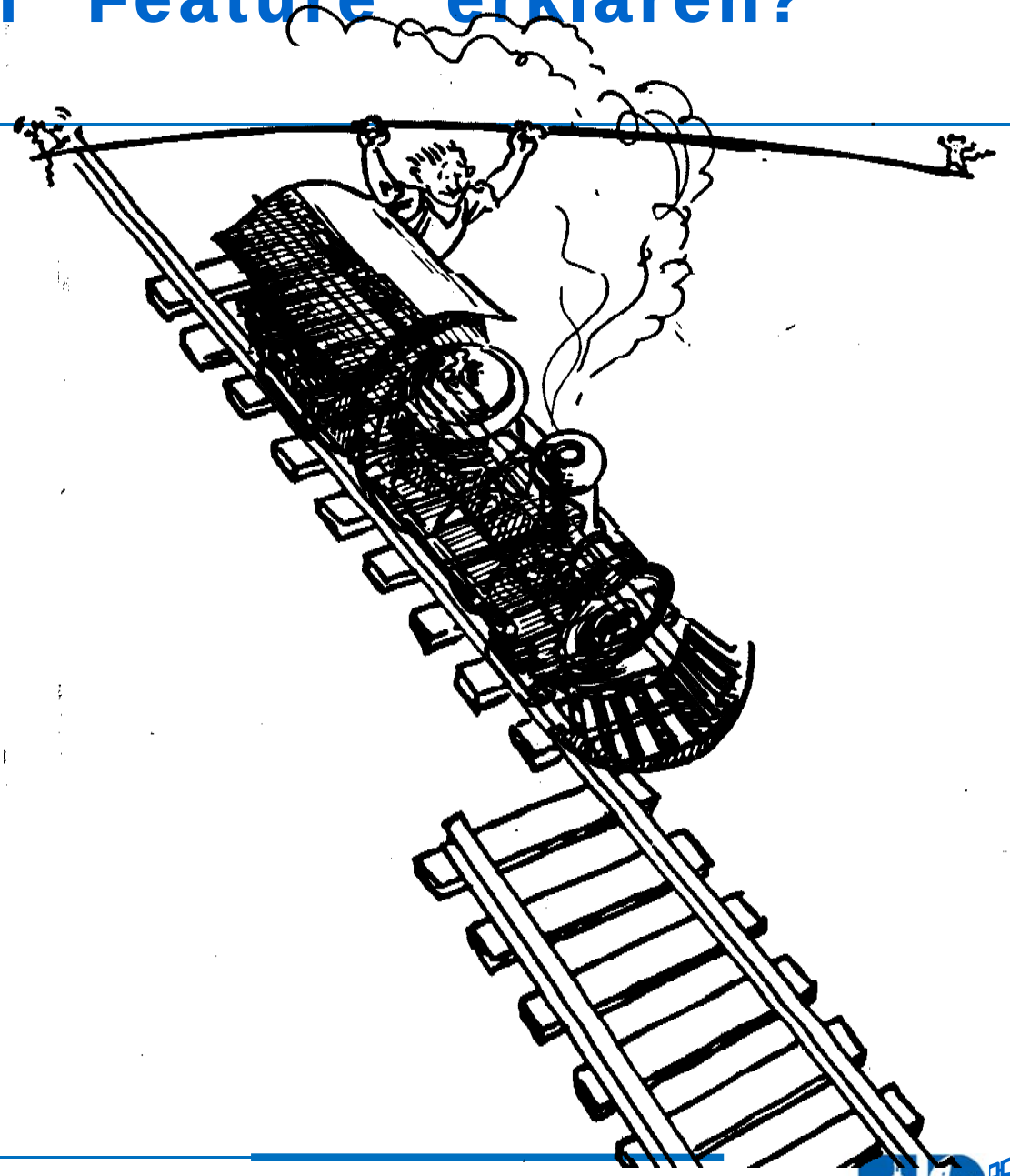
Redundanz?



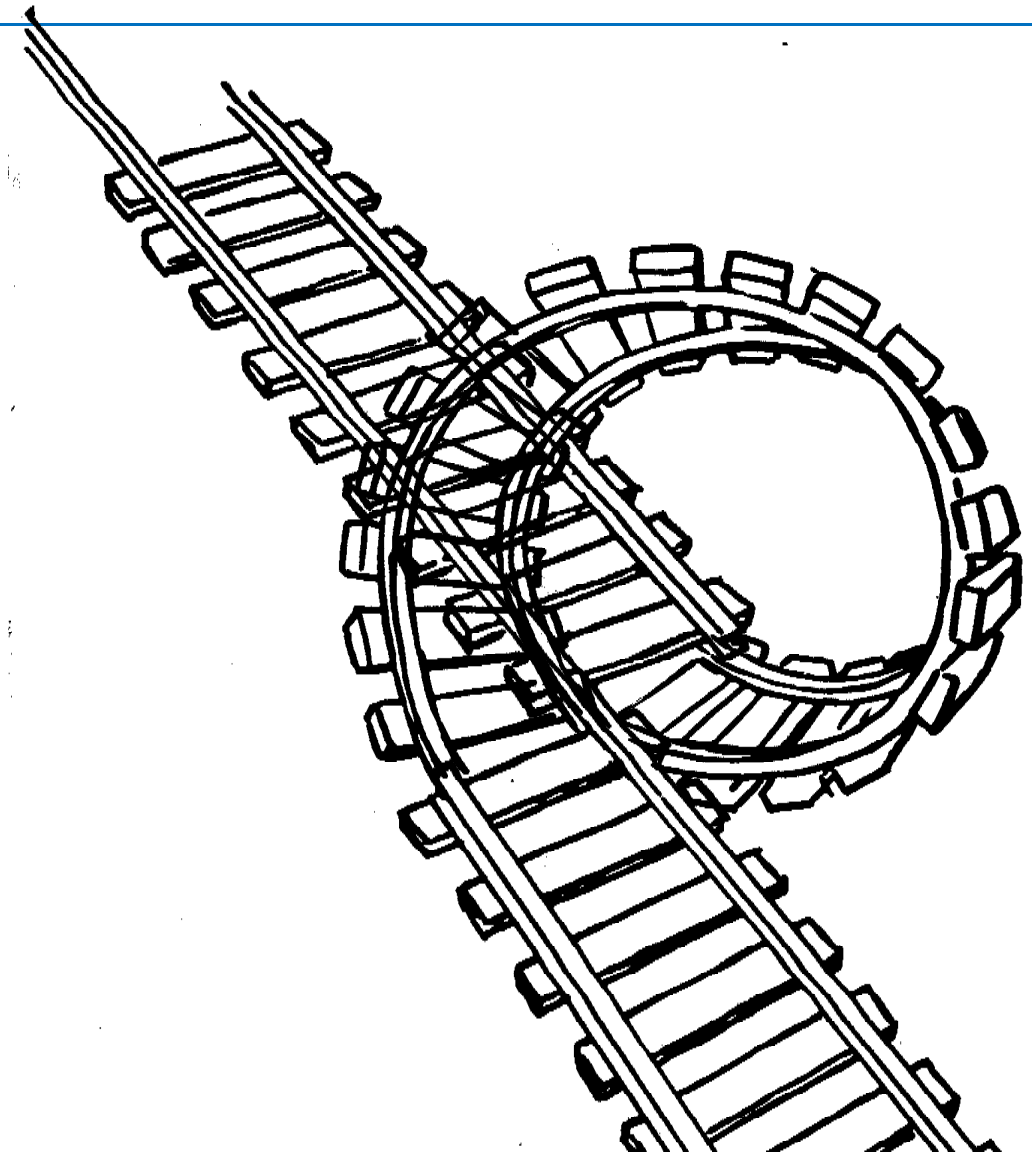
Zusammenflicken („patching“)?



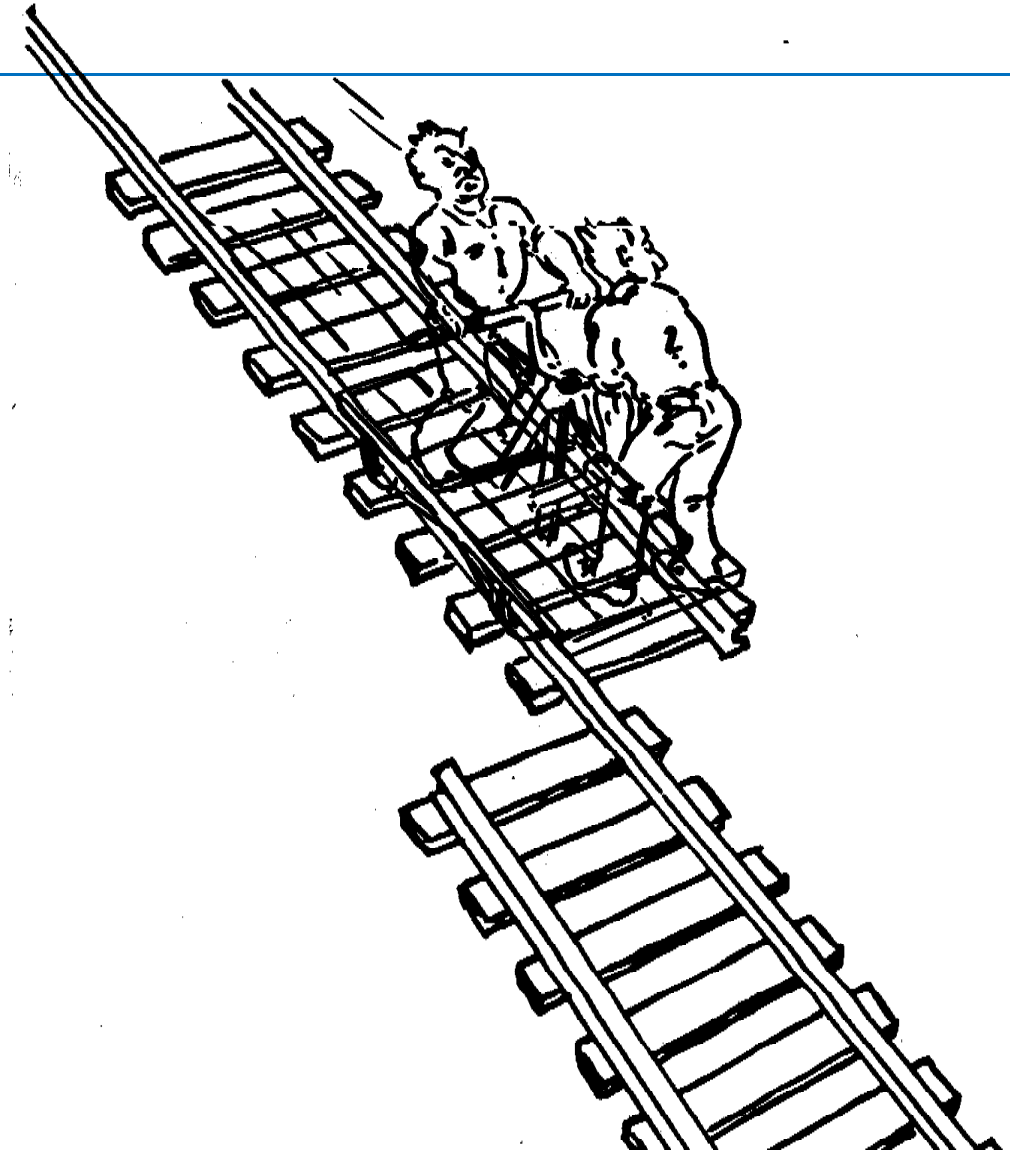
Den “Bug” zum “Feature” erklären?



Verifikation?



Testen?



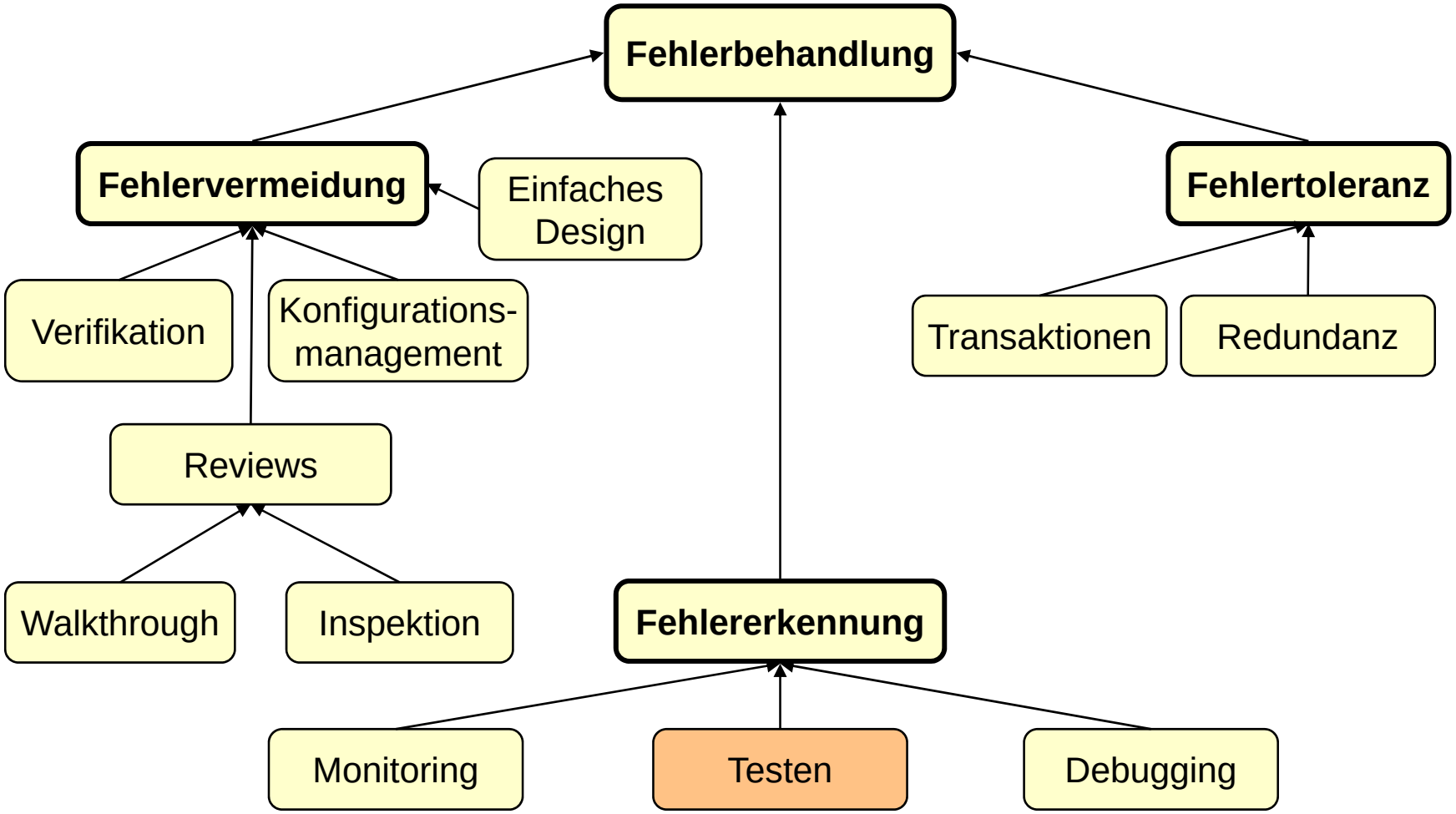
Fehlerbehandlung

- Verifikation
  - Geht von einer hypothetischen Umgebung aus, die sich in der Regel nicht mit der tatsächlichen Umgebung deckt.
  - Der Beweis kann selbst fehlerhaft sein (vernachlässigt z.B. wichtige Nebenbedingungen oder ist schlicht und einfach falsch).
- Redundanz
  - Meist viel zu teuer
- Zusammenflicken („Patching“)
  - Schadet der Performance und Wartbarkeit
- „Bugs“ zu „Features“ erklären
  - Schlechte Gepflogenheit
- Testen (diese Vorlesung)
  - In der Praxis oft die einzig mögliche systematische Vorgehensweise.

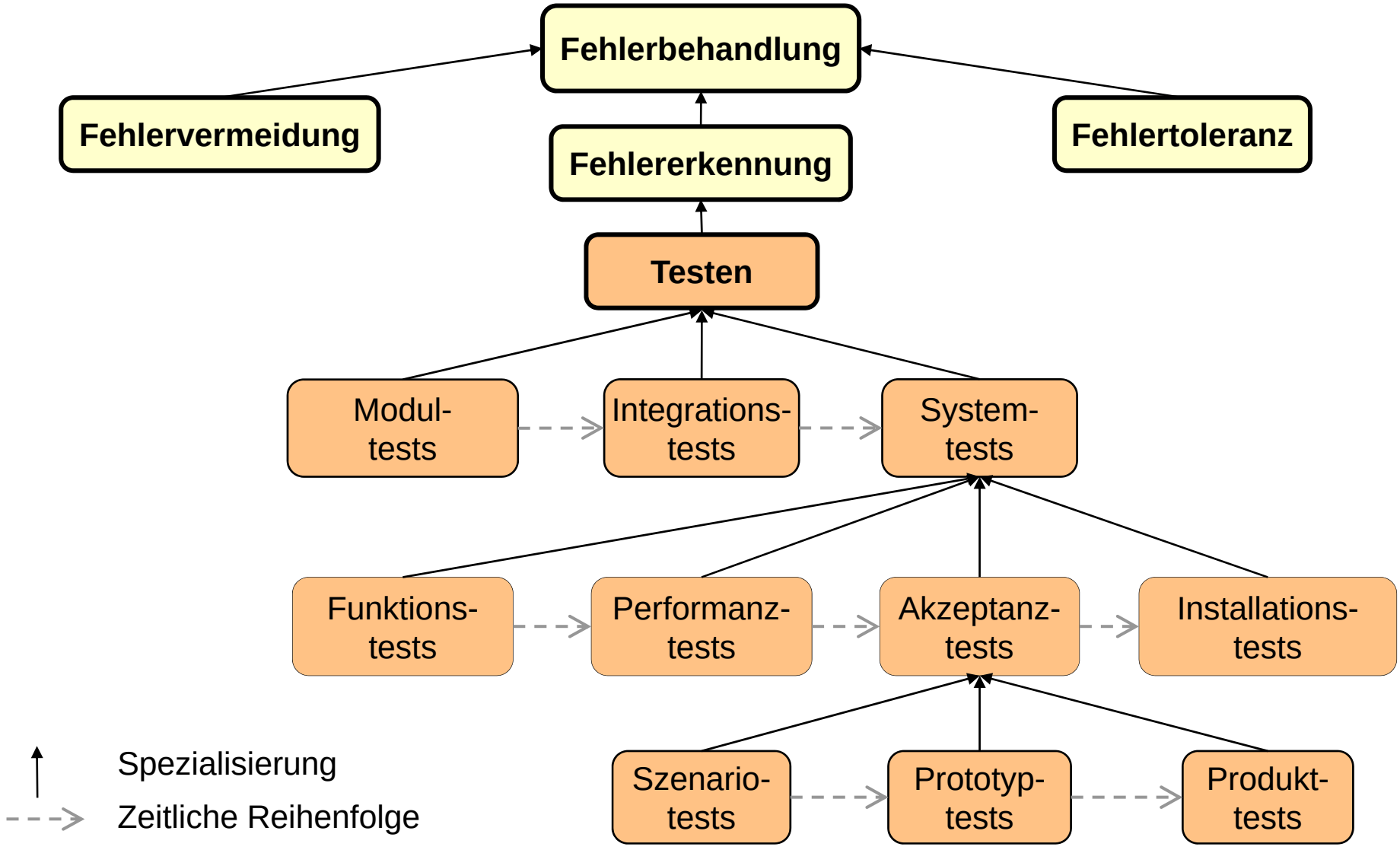
Systematischer Umgang mit Fehlern

- Fehler**vermeidung** (vor der Auslieferung des Systems):
  - Verifikation: algorithmische Fehler vermeiden
  - Versionskontrolle: Inkonsistenzen der Systemkonfiguration vermeiden
  - Programmiermethodik: Durch gutes Design Komplexität reduzieren
  - Code-Reviews: Duch gemeinsame Besprechung Fehlerursachen erkennen und vorbeugen
- Fehler**erkennung** (während Systemlauf):
  - Monitoring: Laufzeitdaten protokollieren, z.B. zum Auffinden von Leistungsproblemen oder Sicherheitslücken.
  - Debugging: Reaktion auf ungeplante Ausfälle.
  - Testen: Führt Ausfälle geplant herbei.
- Fehler**toleranz** (Wiederherstellung nach einem Systemausfall):
  - Transaktionen: Rücksetzen auf letzten konsistenten Zustand
  - Redundanz: Mehrere Systeme laufen parallel, verschiedene Hard- und Software!

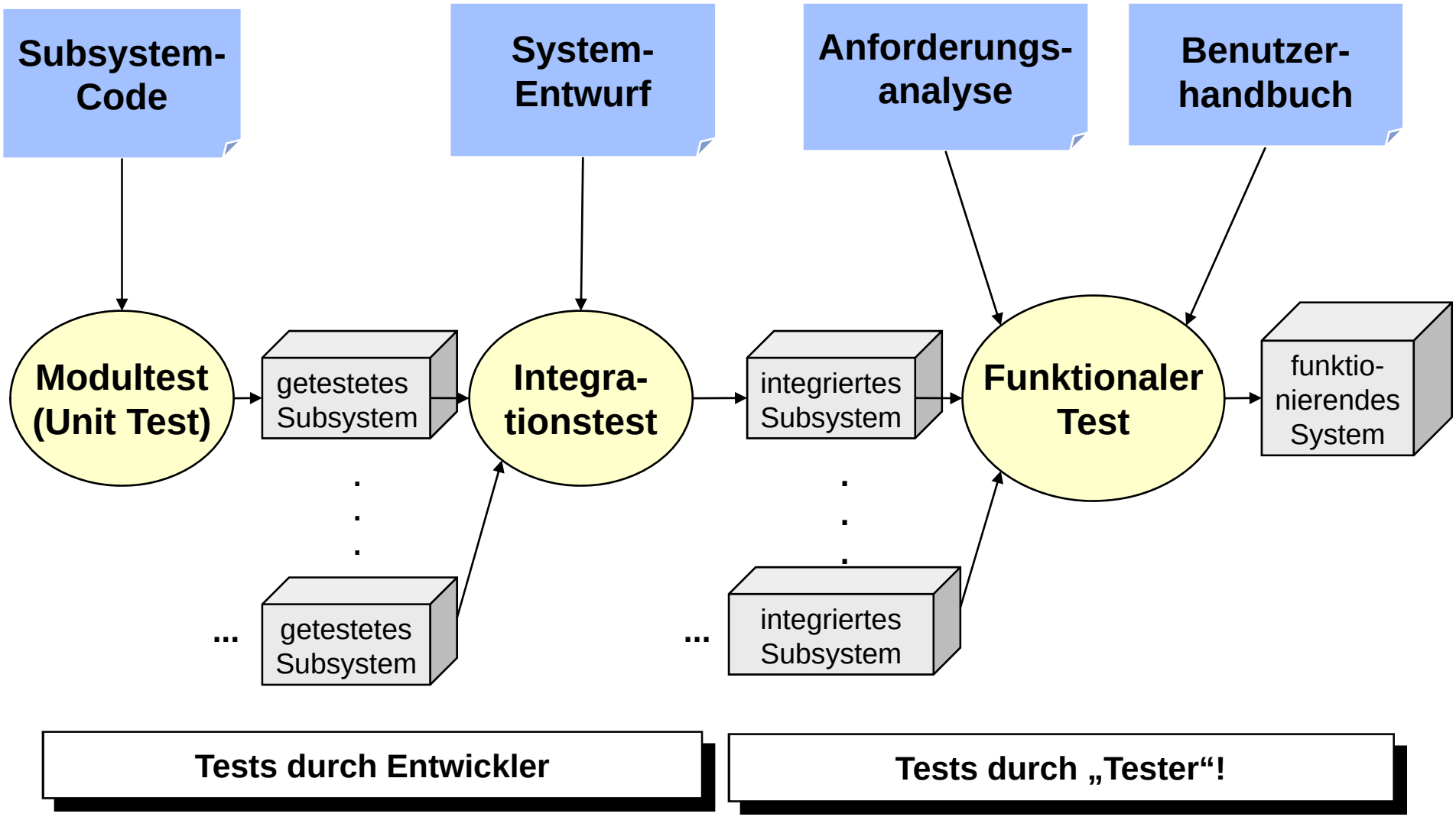
Fehlerbehandlungstechniken



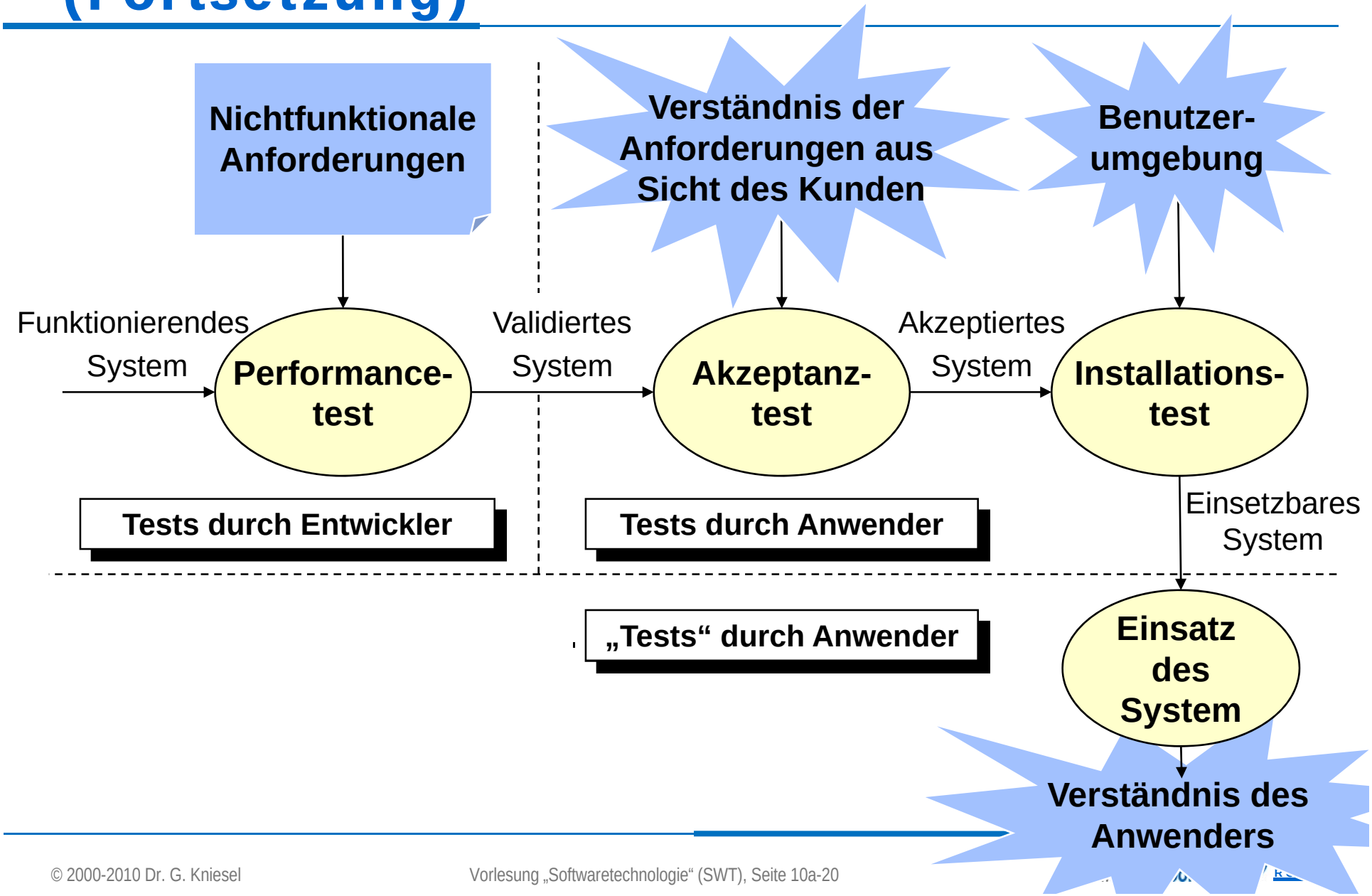
Test-Arten


Fehlerbehandlung
Fehlervermeidung
Fehlererkennung
Fehlertoleranz
Testen
Modul-tests
Integrations-tests
System-tests
Funktions-tests
Performanz-tests
Akzeptanz-tests
Installations-tests
Szenario-tests
Prototyp-tests
Produkt-tests
Spezialisierung
Zeitliche Reihenfolge

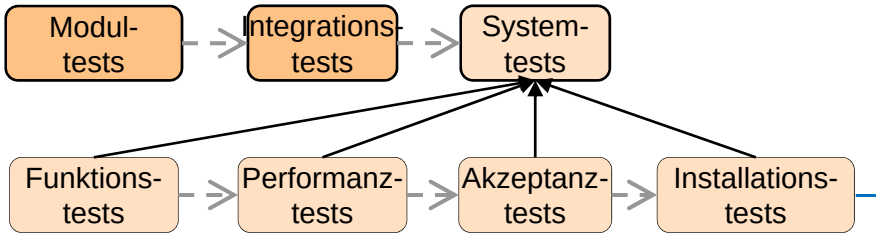
Testaktivitäten im Zusammenhang



Aktivitäten während des Testens (Fortsetzung)

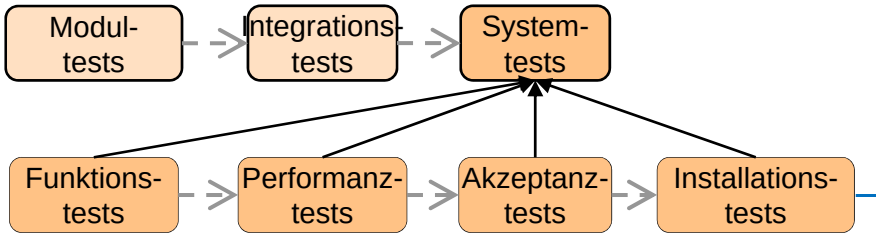


Komponententests



- Modultests
  - **Was**: Einzelnes Subsystem
  - **Ziel:** Sicherstellen dass das Subsystem korrekt programmiert ist und die gewünschte Funktionalität ausführt
  - **Wer**: Wird von Entwicklern durchgeführt
- Integrationstests
  - **Was:** Gruppen von Subsystemen (Sammlung von Klassen); evtl. das ganze System
  - **Ziel:** Überprüfen der Schnittstellen zwischen den Subsystemen
  - **Wer:** Wird von Entwicklern durchgeführt
- Automatisierung wichtig
  - Modul- und Integrationstests sollten möglichst automatisiert ablaufen! (→ Nächstes Kapitel)

Systemtests



- Systemtests
  - **Was:** Das gesamte System
  - **Ziel:** Feststellen, ob das System die (funktionalen und globalen) Anforderungen erfüllt
  - **Wer:** Wird von Testern (=/= Entwicklern) durchgeführt

- Akzeptanz- / Anwendbarkeitstests
  - **Was:** Bewertung des von den Entwicklern ausgelieferten Systems
  - **Ziel:** Demonstrieren, dass das System den Anforderungen des Kunden entspricht und zum Gebrauch bereit ist
  - **Wer:** Wird vom Kunden durchgeführt; Kann die testweise Ausführung typischer Transaktionen vor Ort beinhalten

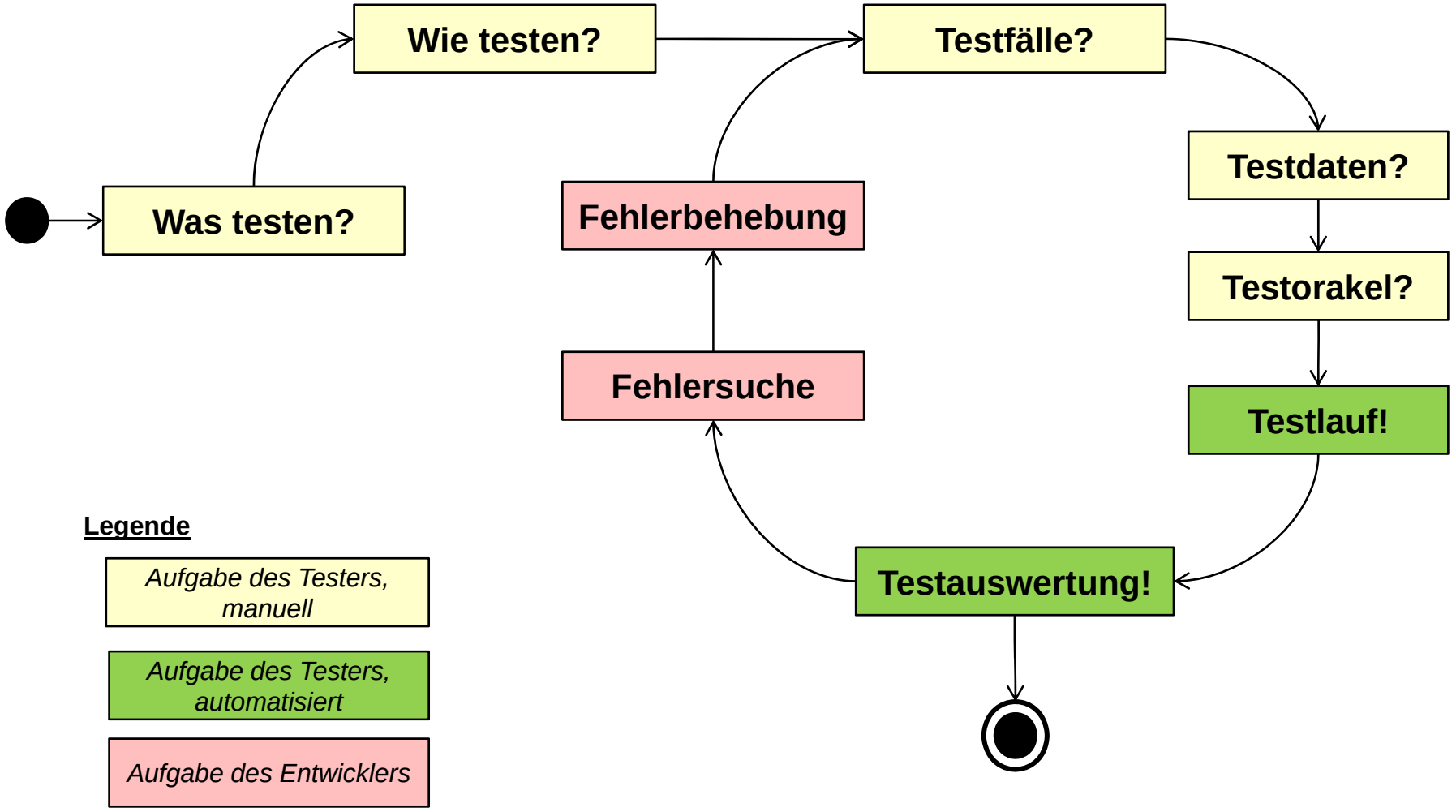
Testen in 7 Schritten

- **1. Testgegenstand**: Bestimme was getestet werden soll
  - Vollständigkeit der Anforderungen, ...
  - Testen des Codes auf Zuverlässigkeit, ...
  - Testen des Entwurfs auf Kohärenz, ...
- **2. Testverfahren**: Entscheide wie getestet wird
  - Inspektion des Codes, ..., Beweise
  - Black-box, White-box
  - Strategie für Integrationstests (big bang, bottom up, top down, sandwich)
- **3. Testfälle**: Entscheide was genau getestet wird
  - Ein Testfall ist eine Menge von Testdaten oder Situationen, die benutzt werden um die zu testende Einheit (Code, Modul, System) oder das gemessene Attribut zu überprüfen.

Testen in 7 Schritten

- **4. Testdaten**: Entscheide wie der Testfall charakterisiert werden kann
  - Daten, die einem bestimmten Testfall entsprechen
- **5. Testorakel**: Entscheide anhand wovon der Testerfolg geprüft wird
  - Ein Orakel besteht aus den vorhergesagten Ergebnissen für eine Menge von Testfällen
  - Das Testorakel muss geschrieben werden, bevor das eigentliche Testen stattfindet.
- **6. Testlauf**
  - Die eigentliche Ausführung des Tests und das Sammeln der Testergebnisse
- **7. Testauswertung**
  - Vergleich der Testergebnisse mit dem Orakel

Testen hat einen eigenen Lebenszyklus



Modultests

Modultests (‚*Unit Tests*‘)

- Informell
  - Inkrementelles Programmieren
- Statische Analyse
  - Ausführung „per Hand“: Lesen des Quellcodes
  - Walk-Through: Informelle Präsentation vor anderen Entwicklern
  - Code Inspection: Formelle Präsentation vor anderen Entwicklern
  - Automatisierte Werkzeuge überprüfen auf
    - ⇨ Abweichung von Coding-Standards
    - ⇨ syntaktische und semantische Fehler
- Dynamische Analyse
  - Black-box-Tests (Überprüfen das Ein-/Ausgabeverhalten)
  - White-box-Tests (Überprüfen die interne Logik des Systems oder Objekts)
    - ⇨ Datenstruktur-basierte Tests (Datentypen bestimmen Testfälle)

Ablauf von Unit-Tests

- 1. Erstelle Unit-Tests, sobald der Objektentwurf abgeschlossen ist.
  - Black-box-Test: Testet Use Cases und Funktionales Modell
  - White-box-Test: Testet dynamisches Modell
  - Datenstrukturtest: Testet das Objektmodell
- 2. Entwickle die Testfälle
  - Ziel: Finden einer minimalen Anzahl von Testfällen, um so viele Pfade wie möglich abzudecken
- 3. Eliminiere redundante Testfälle
  - Verschwende keine Zeit
- 4. Führe Schreibtischtests am eigenen Code durch
  - Reduziert die für das Testen nötige Zeit
- 5. Erstelle eine Testumgebung
  - Testtreiber und Stubs werden für Integrationstests benötigt
- 6. Beschreibe das Testorakel
  - Oft das Ergebnis des ersten erfolgreich ausgeführten Tests

Ablauf von Unit-Tests (Fortsetzung)

- 7. Führe den Testfall aus
  - Regressionstests nicht vergessen: Führe die Testfälle jedes Mal durch, wenn Änderungen vorgenommen wurden
- 8. Vergleiche die Ergebnisse mit denen des Testorakels
  - Automatisiere dies so weit wie möglich.

Black-box-Tests

- Testfallauswahl anhand von Analysewissen über funkt. Anforderungen
  - Use cases
  - Erwartete Eingabedaten
  - Ungültige Eingabedaten

- Fokus: Ein-/Ausgabeverhalten
  - Das Modul besteht den Test, wenn für jede gegebenen Eingabe die Ausgabe vorhersagbar ist.
  - Es ist meist unmöglich, jede mögliche Eingabe zu erzeugen („test cases“)

Black-box-Tests: Partitionierung

- Reduzierung der Anzahl der Testfälle durch Partitionierung
  - Zerlege Eingabe in Äquivalenzklassen
  - Wähle Testfälle für jede der Äquivalenzklassen.
    (Beispiel: Wenn ein Objekt negativen Zahlen akzeptieren soll, reicht es aus, mit einer negativen Zahl zu testen).

- Wahl der Partitionierung (Richtlinien):
  - Gültige Eingabewerte liegen in einem Intervall. Wähle Testfälle aus drei Äquivalenzklassen:
    - ⇨ Unterhalb des Intervalls
    - ⇨ Im Intervall
    - ⇨ Oberhalb des Intervalls
  - Gültige Eingabewerte bilden eine diskrete Menge. Wähle Testfälle aus zwei Äquivalenzklassen:
    - ⇨ Gültiger diskreter Wert
    - ⇨ Ungültiger diskreter Wert

White-box-Tests

- Wissen über Entwurf und Implementierung nutzen um
  - Anzahl von Testfällen zu begrenzen oder
  - gründliche Testabdeckung zu gewährleisten.

- Testfallauswahl anhand
  - Entwurfswissen über Systemaufbau, Algorithmen, Datenstrukturen
    - ⇨ Kontrollstrukturen: Verzweigungen, Schleifen, ...
    - ⇨ Datenstrukturen: Datensätze, Felder, Arrays, ...
  - Implementierungswissen über Algorithmen
    - ⇨ Erzwinge Division durch Null
    - ⇨ Verwende eine Abfolge von Testfällen für Interrupt Handler.
    - ⇨ ...

White-box-Tests: Abdeckungen

- Fokus: Gründlichkeit (hohe Abdeckung)
  - Abdeckung ist prozentuales Maß der vom Test durchlaufenen Teile der getesteten Komponente im Verhältnis zur Gesamtanzahl solcher Teile.

$$Abdeckung = \frac{Anzahl\ getesteter\ Teile}{Anzahl\ aller\ Teile}$$

- Zu testende „Teile“
  - jede Anweisung → Anweisungsabdeckung
  - jede Verzweigung → Verzweigungsabdeckung
  - jede Schleife → Schleifenabdeckung
  - jeder Pfad → Pfadabdeckung

- Ein „Teil“ gilt als getestet wenn es während des Testlaufs mindestens einmal durchlaufen wurde

Anweisungsabdeckung

- Eine Anweisung ist abgedeckt, wenn sie ausgeführt wird.
- Anweisung != Codezeile

- Nicht einfach, 100% Anweisungsabdeckung zu erreichen
  - Fehlerbedingungen / seltene Ereignisse:
    - ⇨
```
if (param > 20) {
   print(„Dies sollte nie geschehen!");
}
```
  - Exceptions inmitten einer Anweisungsfolge:
    - ⇨
```
doThis();
doThat();
```
← Falls `doThis()` eine Exception wirft wird `doThat()` nicht ausgeführt.

- Synonyme
  - Anweisungsausführungs-, Zeilen-, Block- oder Segmentabdeckung
  - Engl.: statement execution, line, block, segment coverage

Verzweigungsabdeckung

- Ein Programm sollte alle möglichen Alternativen durchlaufen:

```
if (x) { print ("a"); }
else   { print ("b"); }
```

- Der Wert von x muß mal „wahr“ und mal „falsch“ sein.

- Schützt vor Fehlern, die dadurch entstehen, dass bestimmte Anforderungen in einem Zweig nicht erfüllt sind.

```
if (flag) { s = "a string"; }
else      { s = null; }
print(s.length);
```

- Obiger Code wirft eine Exception, falls `flag` gleich `false` ist.

Verzweigungsabdeckung

- Fehlendes else ist auch ein Zweig der getestet werden muss!
- Beispiel:

```
s = null;
if (flag) { s = "a string" }
print(s.length);
```

- Obwohl der Fall „*flag = false*“ nicht explizit getestet wird, kann er zur Laufzeit auftreten und der length-Zugriff erfolgt dann auf „null“.

- 100% Verzweigungsabdeckung impliziert 100% Anweisungsabdeckung, sofern das Programm nicht „Ausnahmen“ (‚exceptions‘) erzeugt.

- Synonyme
  - arc coverage
  - decision coverage
  - all edges coverage

Pfadabdeckung: Warum?

- Es gibt Fehlerklassen, die durch Verzweigungsabdeckung nicht erkannt werden:

```
s = null;
if (a) { s = "a string" }
if (b) { print(s.length)}
```

- 100% Verzweigungsabdeckung ist gegeben, wenn `(a, b)` mit `(true, true)` und `(false, false)` belegt wird.

- Aber: Für `(a, b)` = `(false, true)` gibt‘s einen Zugriff auf `null` ☹

Pfadabdeckung: Idee / Definition

- Ziel ist es sicherzustellen, dass alle Pfade durch das Programm durchlaufen werden.

```
s = null;
if (a) { s = "a string" }
if (b) { print(s.length)}
```

- Es gibt vier Pfade durch das obige Stück Code:
  - (a, b) = (true, true), (true, false), (false, true), (false, false)
  - Für 100% Pfadabdeckung müssen alle vier durchlaufen werden.

- Ein Pfad ist jeder Weg von Startpunkt zum Endpunkt im Kontrollflussgraphen des analysierten Codes.
- Return-Anweisungen haben Kanten zum Endpunkt
  - Exceptions nicht → sie stellen kein „normales“ Ende dar!
- Fehlende else-Anweisungen zählen als Kanten im Graphen
  - In obigem Beispiel ist gerade der „fehlende“ Zweig a=false kritisch!

Pfadabdeckung: Probleme

- Problem 1: 100% Pfadabdeckung ist nicht immer möglich:

```
if (a) x;
y;
if (!a) z;
```

  - Ein Test des obigen Programms, der x erreicht kann nicht z erreichen und umgekehrt, da die Bedingungen komplementär sind.
  - 50% Pfadabdeckung sind das beste, was hier erreicht werden kann.

- Problem 2: Halteproblem (unendliche Anzahl von Pfaden bei Schleifen)
  - Praxis: nur einen Schleifendurchlauf / keine zyklischen Pfade testen
- Problem 3: Es gibt zu viele Pfade durch ein Programm.
  - Praxis: Beschränkung auf kleine Codebereiche (z.B. eine Prozedur). Pfadabdeckung größerer Komponenten oder gesamter Programme nicht praktikabel.

Pfadabdeckung

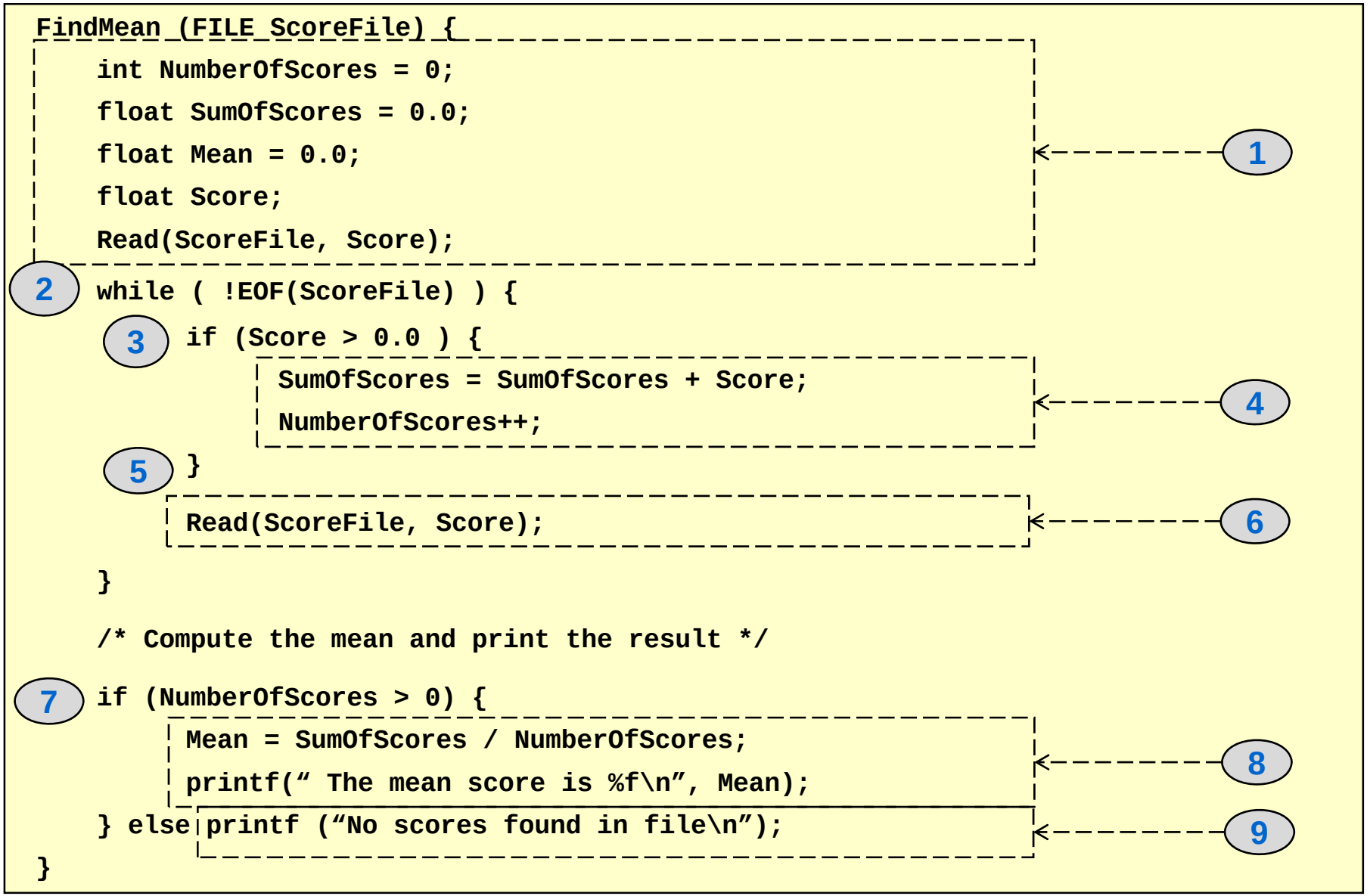
- 100% Pfadabdeckung impliziert 100% Verzweigungsabdeckung und 100% Anweisungsabdeckung

- Synonyme
  - predicate coverage
  - basic path coverage
  - LCSAJ (Linear Code Sequence And Jump) coverage

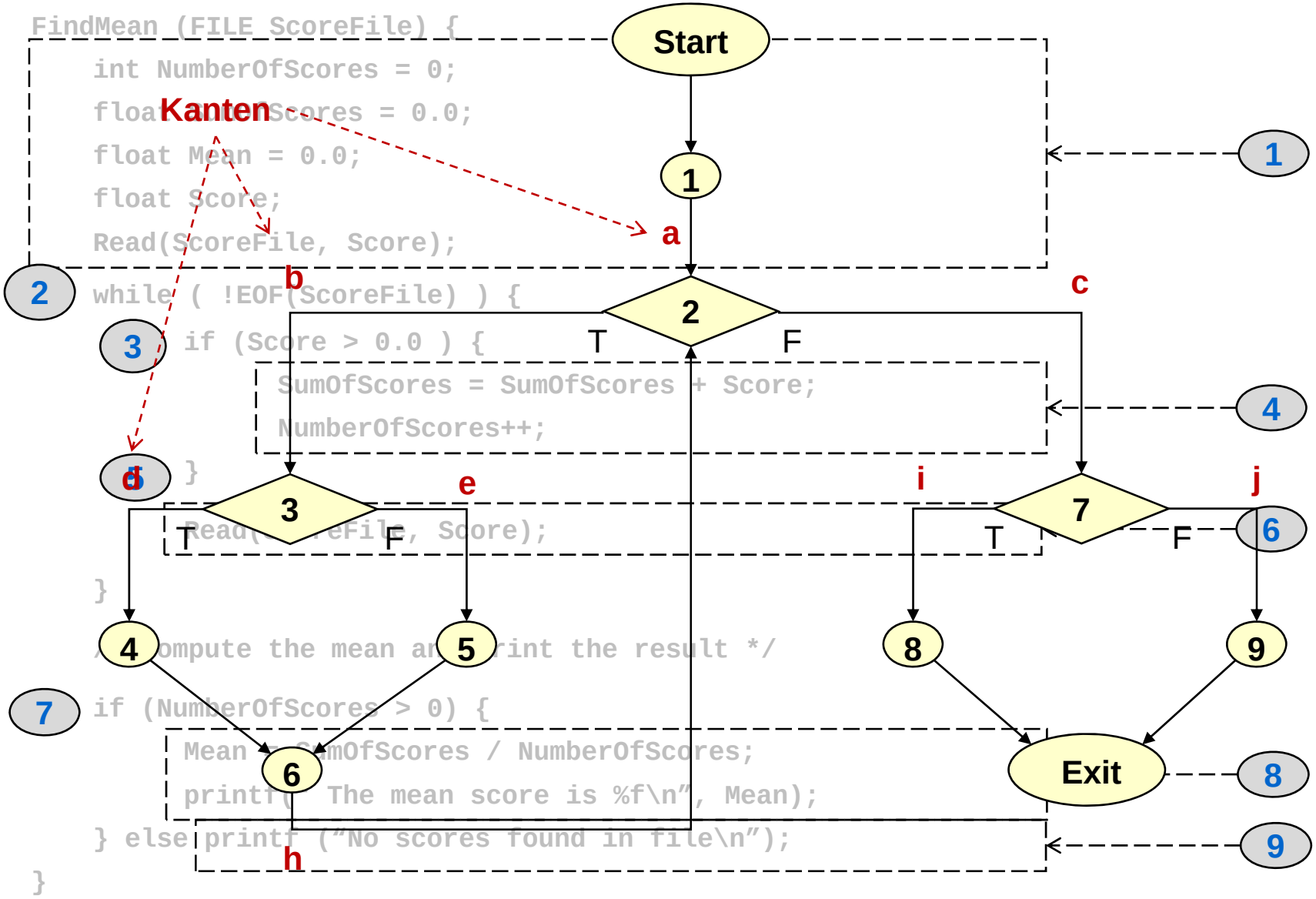
Beispiel: White-box-Test anhand Pfadabdeckung

```
FindMean (FILE ScoreFile) {
    int NumberOfScores = 0;
    float SumOfScores = 0.0;
    float Mean = 0.0;
    float Score;
    Read(ScoreFile, Score);
    while ( !EOF(ScoreFile) ) {
          if (Score > 0.0 ) {
                SumOfScores = SumOfScores + Score;
                NumberOfScores++;
          }
          Read(ScoreFile, Score);
    }
    /* Compute the mean and print the result */
    if (NumberOfScores > 0) {
          Mean = SumOfScores / NumberOfScores;
          printf(" The mean score is %f\n", Mean);
    } else printf ("No scores found in file\n");
}
```

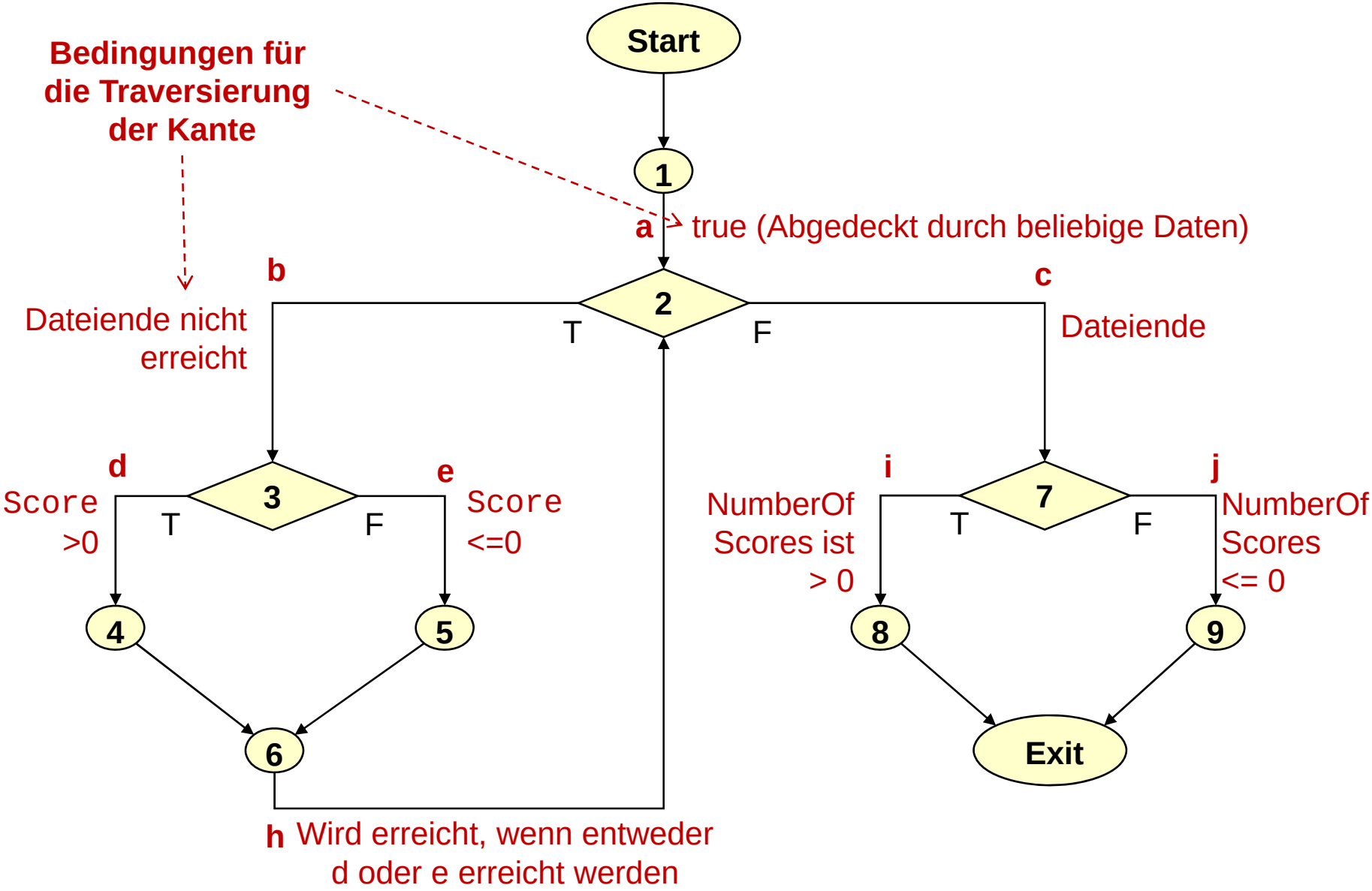
1. Bestimmung von Kontrollflussknoten



2. Konstruktion des Kontrollflussdiagramms



3. Bestimmung von Bedingungen für Kantendurchlauf



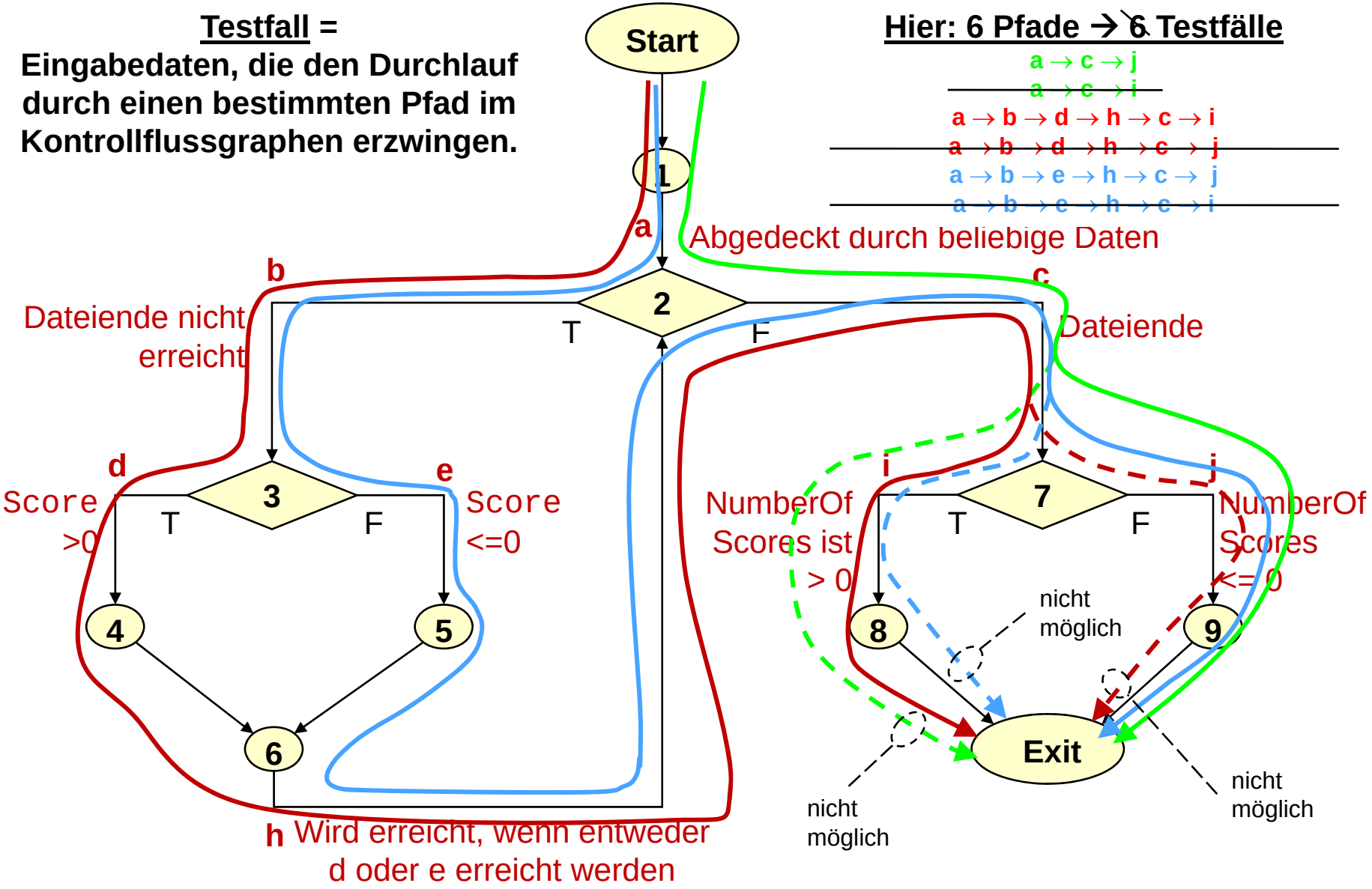
4. Finden der Testfälle: Pfade

Testfall =
Eingabedaten, die den Durchlauf durch einen bestimmten Pfad im Kontrollflussgraphen erzwingen.

Hier: 6 Pfade → 6 Testfälle

~~3~~
~~4~~
~~5~~

a → c → j
~~a → c → i~~
a → b → d → h → c → i
~~a → b → d → h → c → j~~
a → b → e → h → c → j
~~a → b → e → h → c → i~~

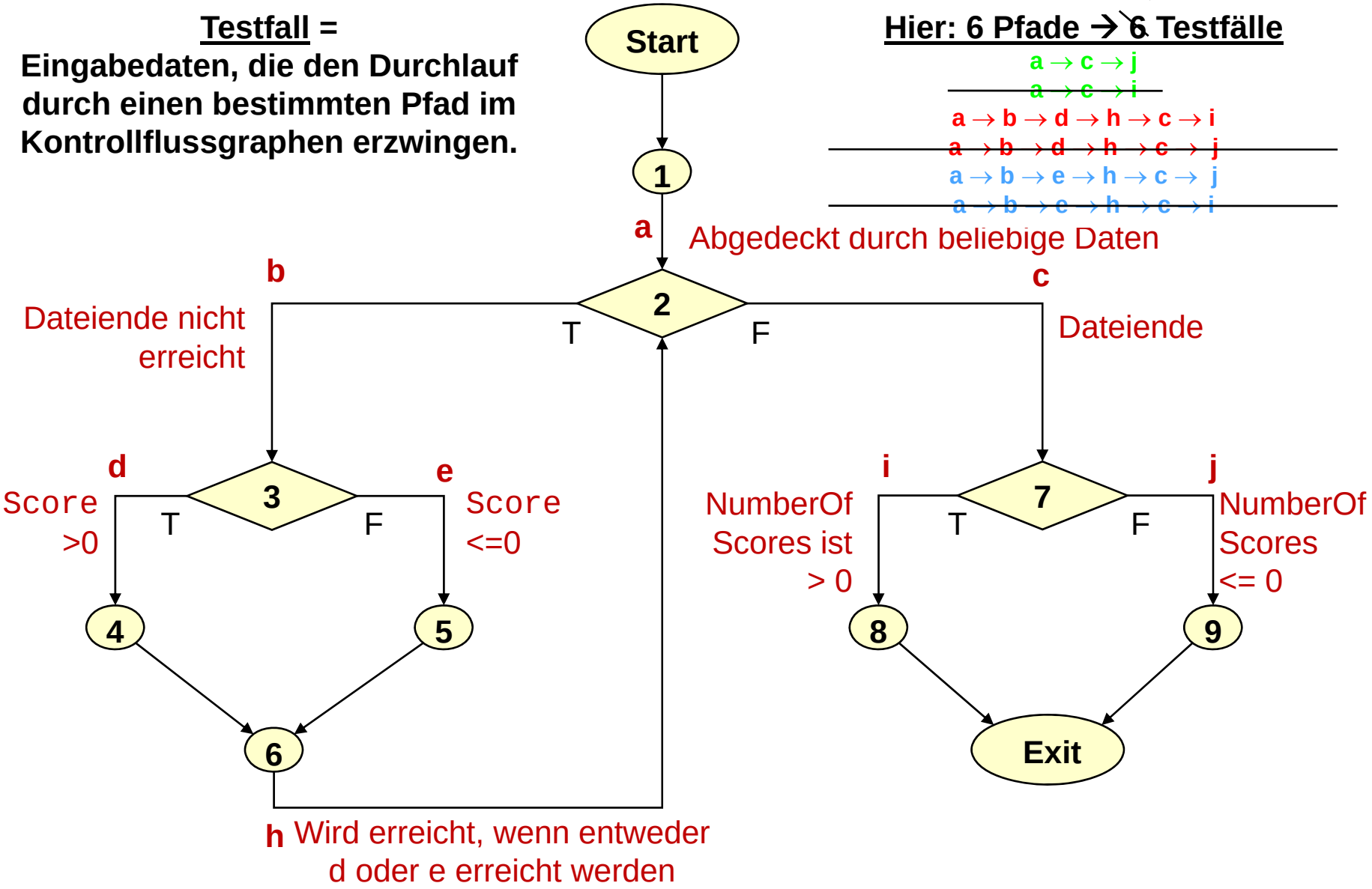


4. Finden der Testfälle: Bedingungen

Testfall =
Eingabedaten, die den Durchlauf durch einen bestimmten Pfad im Kontrollflussgraphen erzwingen.

Hier: 6 Pfade → 6 Testfälle

- a → c → j
- ~~a → c → i~~
- a → b → d → h → c → i
- ~~a → b → d → h → c → j~~
- a → b → e → h → c → j
- ~~a → b → c → h → c → i~~



4. Finden der Testfälle: Eingabedaten, die die Bedingungen erzwingen

Testfall =
Eingabedaten, die den Durchlauf durch einen bestimmten Pfad im Kontrollflussgraphen erzwingen.

Hier: 6 Pfade → 6 Testfälle

a → c → j
a → c → i
a → b → d → h → c → i
a → b → d → h → c → j
a → b → e → h → c → j
a → b → e → h → c → i

Relevante Kantenbedingungen

- **c:** Dateiende
- **i:** NumberOfScores > 0 ≅ positive Zahlen gelesen
- **j:** NumberOfScores <= 0 ≅ keine positive Zahlen gelesen
- **b:** Dateiende nicht erreicht
- **d:** `Score` >0 ≅ positive Zahl gelesen
- **e:** `Score` <=0 ≅ keine positive Zahlen gelesen

Minimale Testfälle / Testdaten

1. Leere Datei
2. Datei mit positiver Zahl
3. Datei mit negativer Zahl

Zusammenfassung Pfadabdeckung

1. Kontrollflussgraphen erstellen
2. Kanten beschriften
3. Kantenbedingungen bestimmen
4. Pfade bestimmen
5. Unmögliche Pfade eliminieren

Für jeden Pfad:

6. Kanten die keine Bedingung mit sich tragen eliminieren
7. Kanten, deren Bedingung durch die vorangegangenen Schritte erfüllt wurden eliminieren
8. Eingabedaten bestimmen, die die Bedingungen der restlichen Kanten erfüllen
9. Testdatensatz erzeugen
10. Testorakel dafür erzeugen

White-Box-Tests versus Black-Box-Tests

- White-box-Tests
  - Potentiell unendliche Anzahl von Pfaden muss getestet werden.
  - Getestet wird anhand der tatsächlich anstatt des erwarteten Verhaltens
  - Keine Erkennung fehlender Use Cases
- Black-box-Tests
  - Potentielle kombinatorische Explosion der Testfälle (gültige & ungültige Daten)
  - Oft ist es unklar, ob die gewählten Testfälle einen bestimmten Fehler entdecken
  - Keine Entdeckung belangloser Use Cases („Features“)

- White-box-Tests und Black-box-Tests sind die beiden Extreme des Modultest-Kontinuums. Beide Testtypen sind erforderlich.
- Jede Auswahl von Testfällen liegt dazwischen und hängt ab von:
  - der Anzahl möglicher logischer Pfade
  - der Art der Eingabedaten
  - dem Rechenaufwand
  - der Komplexität von Algorithmen und Datenstrukturen

Integrationstests

Komponentenbasierte Teststrategie

- **System** = Sammlung von Subsystemen (Mengen von Klassen), die während der System- und Objektentwurfsphasen bestimmt wurden.

- **Teststrategie** = Die Reihenfolge, in der Subsysteme für Tests und für die Integration ausgewählt werden
  - *Big-bang*-Integration (nicht-inkrementell)
  - *Bottom-up*-Integration
  - *Top-down*-Integration
  - *Sandwich Testing*
  - Variationen der dieser Strategien

Einzelne Schritte des Integrations-Testens

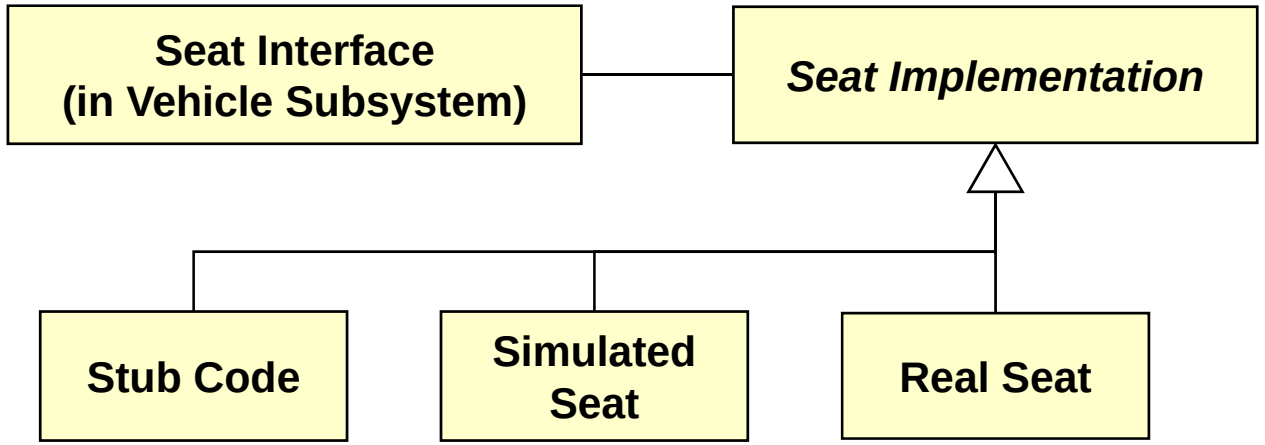
Wiederhole die Schritte 1 bis 6 bis das gesamte System getestet ist:

1) Wähle eine Komponente zum Testen aus
2) Führe für alle Klassen der Komponente Modul-Tests durch
3) Erledige Anpassungen, um den Integrationstest lauffähig zu machen (Treiber, Stubs)
4) Führe Tests durch (black- und/oder white-box)
5) Führe Buch über Testfälle und Testaktivitäten

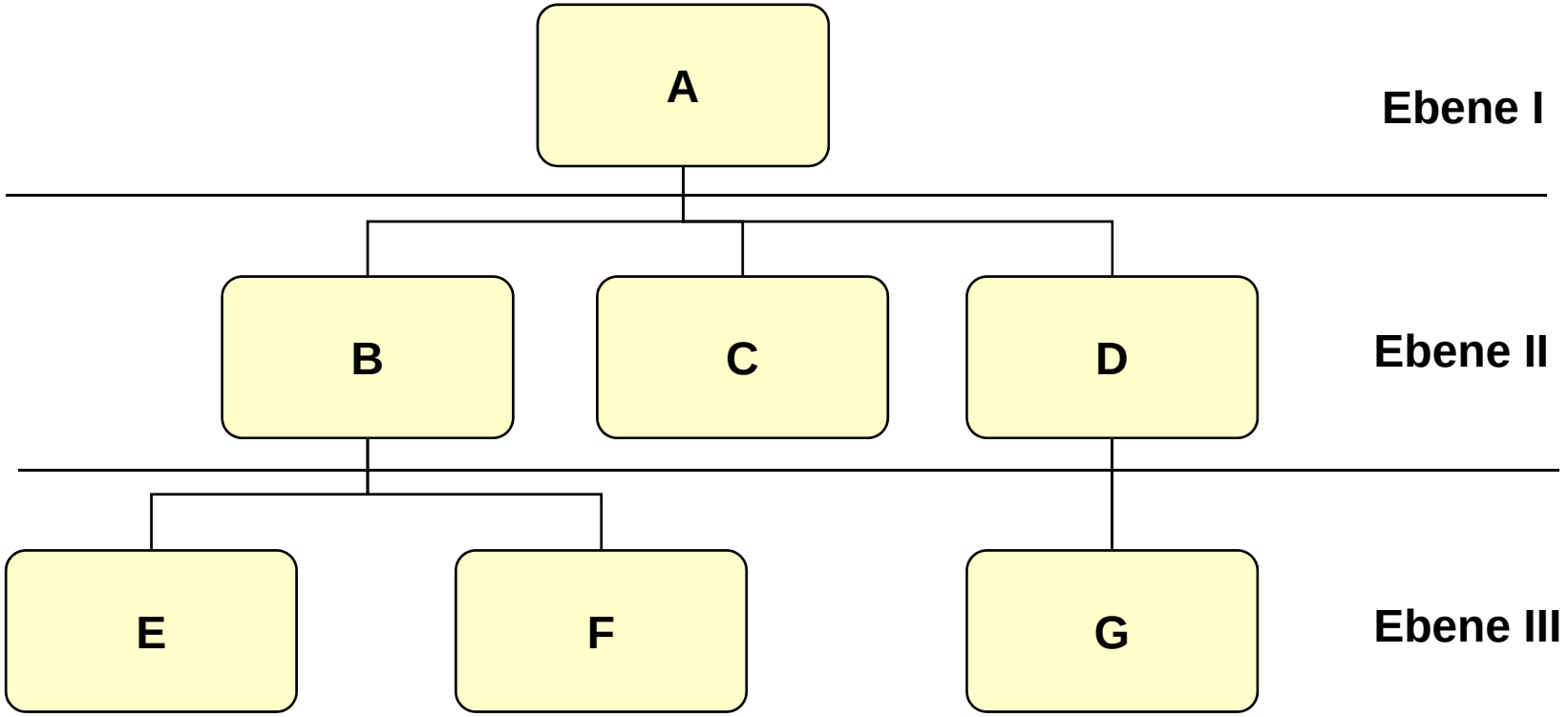
→ Das primäre Ziel von Integrationstests ist es, Fehler in der (momentanen) Konfiguration der Komponenten zu identifizieren.

Einsatz des Bridge-Patterns, um frühe Integrationstests zu ermöglichen

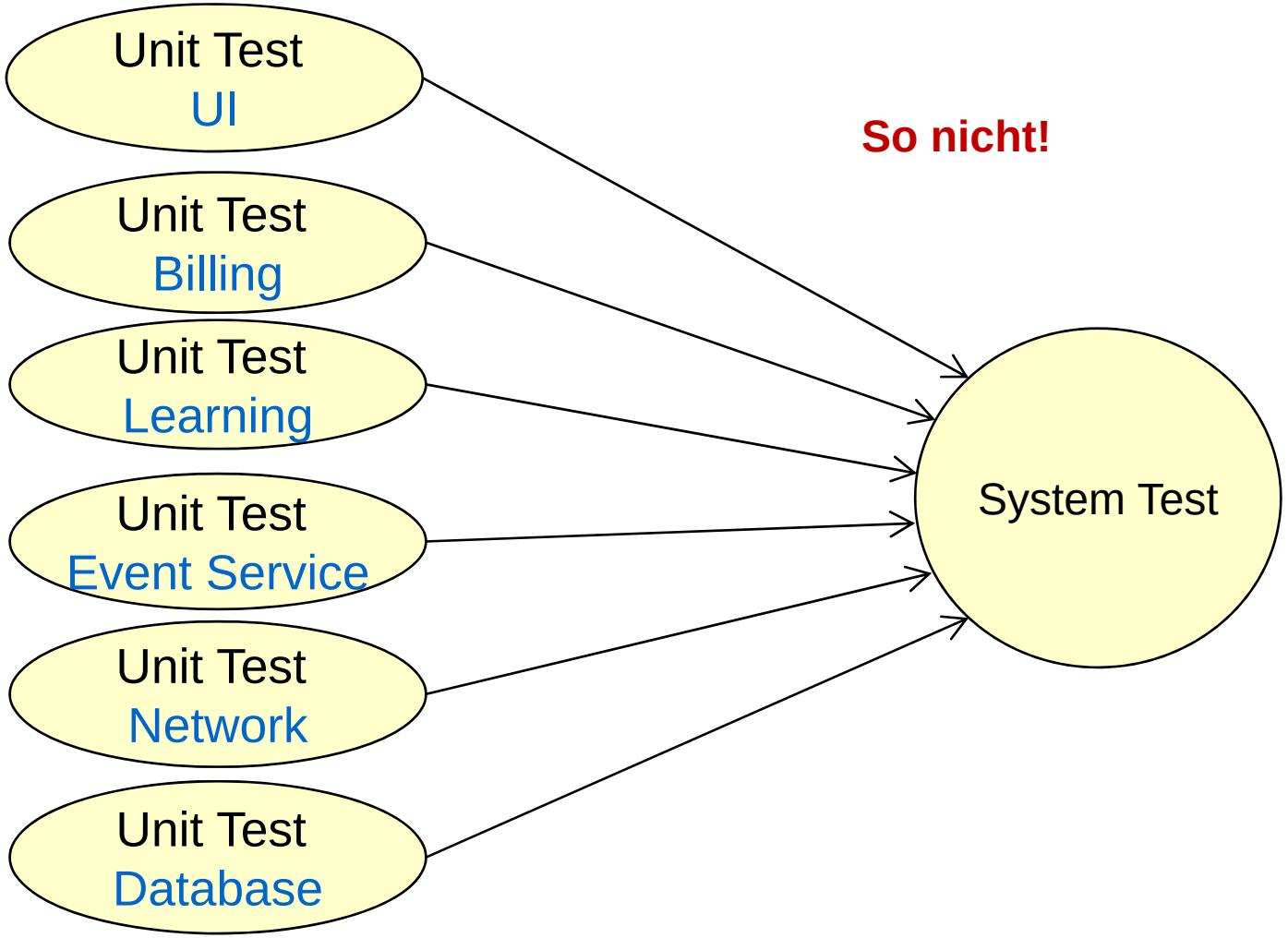
- Verwendung des Bridge-Patterns, um mehrere Implementierungen für die selbe Schnittstelle zur Verfügung zu stellen
- Ermöglicht Schnittstelle zu einer Komponente, die unvollständig, noch nicht bekannt, oder während des Testens nicht verfügbar ist.



Beispiel für Integrationstests: Aufrufhierarchie mit drei Ebenen

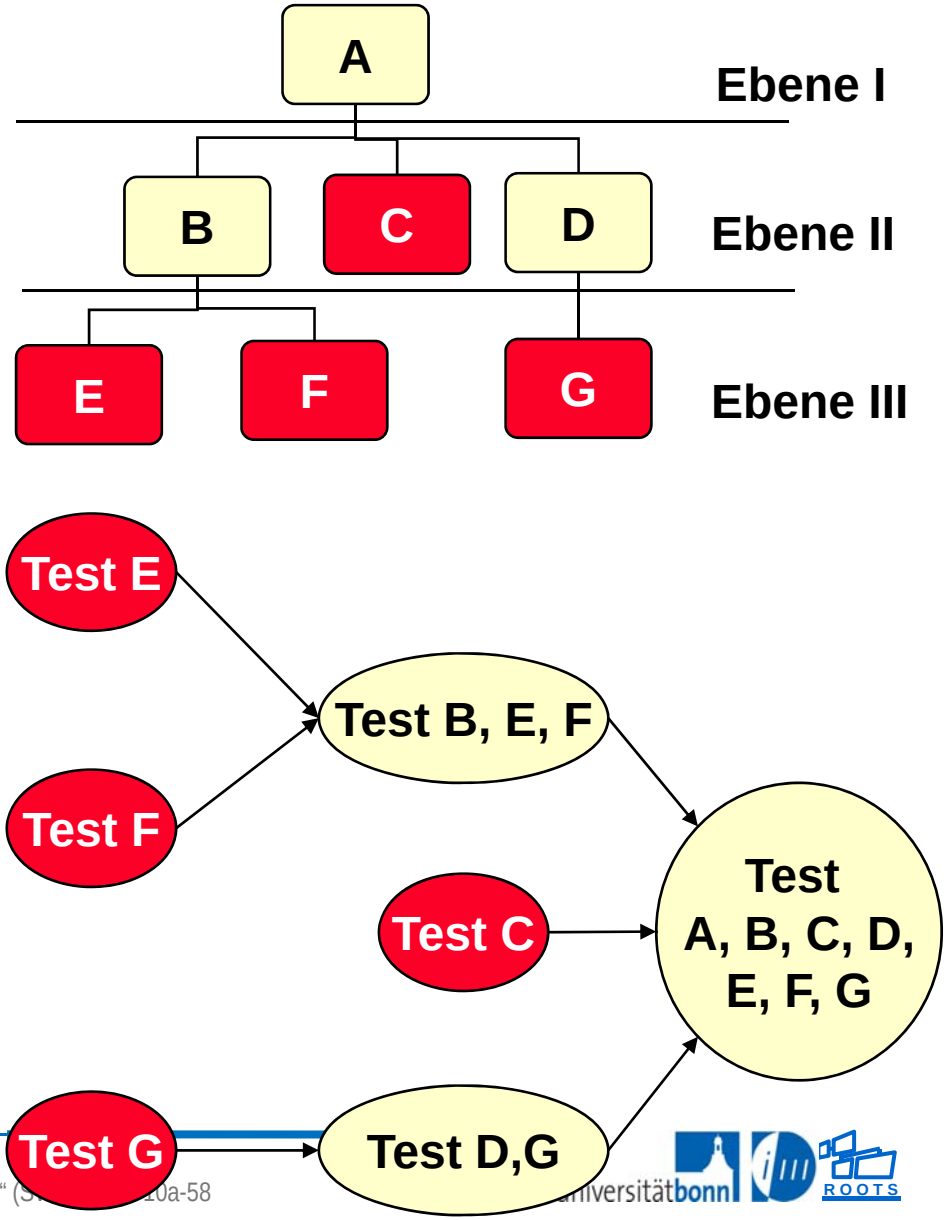


Integrationstests: Big-Bang-Ansatz



Bottom-up-Integrationstests

- Subsysteme in der untersten Ebene der Aufrufhierarchie werden einzeln getestet.
- Dann werden die Subsysteme der nächsten Ebene getestet, die die vorher getesteten Subsysteme aufrufen.
- Dieser Vorgang wird solange wiederholt, bis alle Subsysteme getestet sind.
- Testtreiber
  - Programm, das einen Testfall auf einem Subsystem ausführt
  - ... und optimalerweise die Testergebnisse selbst auswertet.



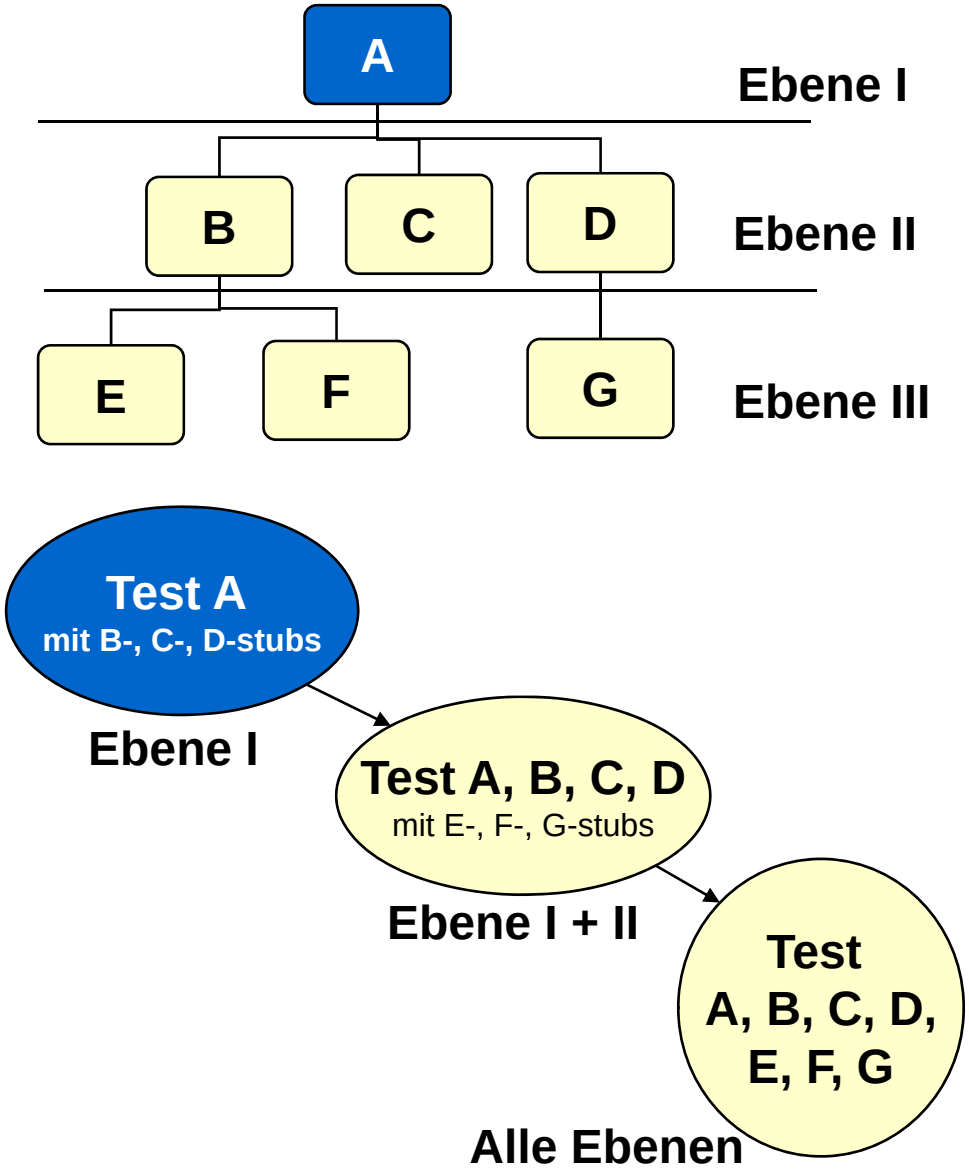
Vor- und Nachteile der Bottom-Up-Integrationstests

- Schlecht für funktional aufgeteilte Systeme
  - Das wichtigste Subsystem wird zuletzt getestet!

- Nützlich für die Integration der folgenden Systeme
  - Objektorientierte Systeme
  - Systeme mit strikten Anforderungen an die Performance
  - Echtzeitsysteme

Top-down Integration Testing

- Teste zunächst die oberste Schicht
- Teste dann alle Subsysteme, die von dem getesteten Subsystemen aufgerufen werden.
- Wiederhole diesen Vorgang, bis alle Subsysteme im Test enthalten sind.
- Test stub
  - Methode/Programm, die das Verhalten eines fehlenden Subsystems simuliert
  - Antwortet auf Testaufrufe mit korrekten Daten, ohne die entsprechenden Funktionalitäten wirklich zu implementieren.

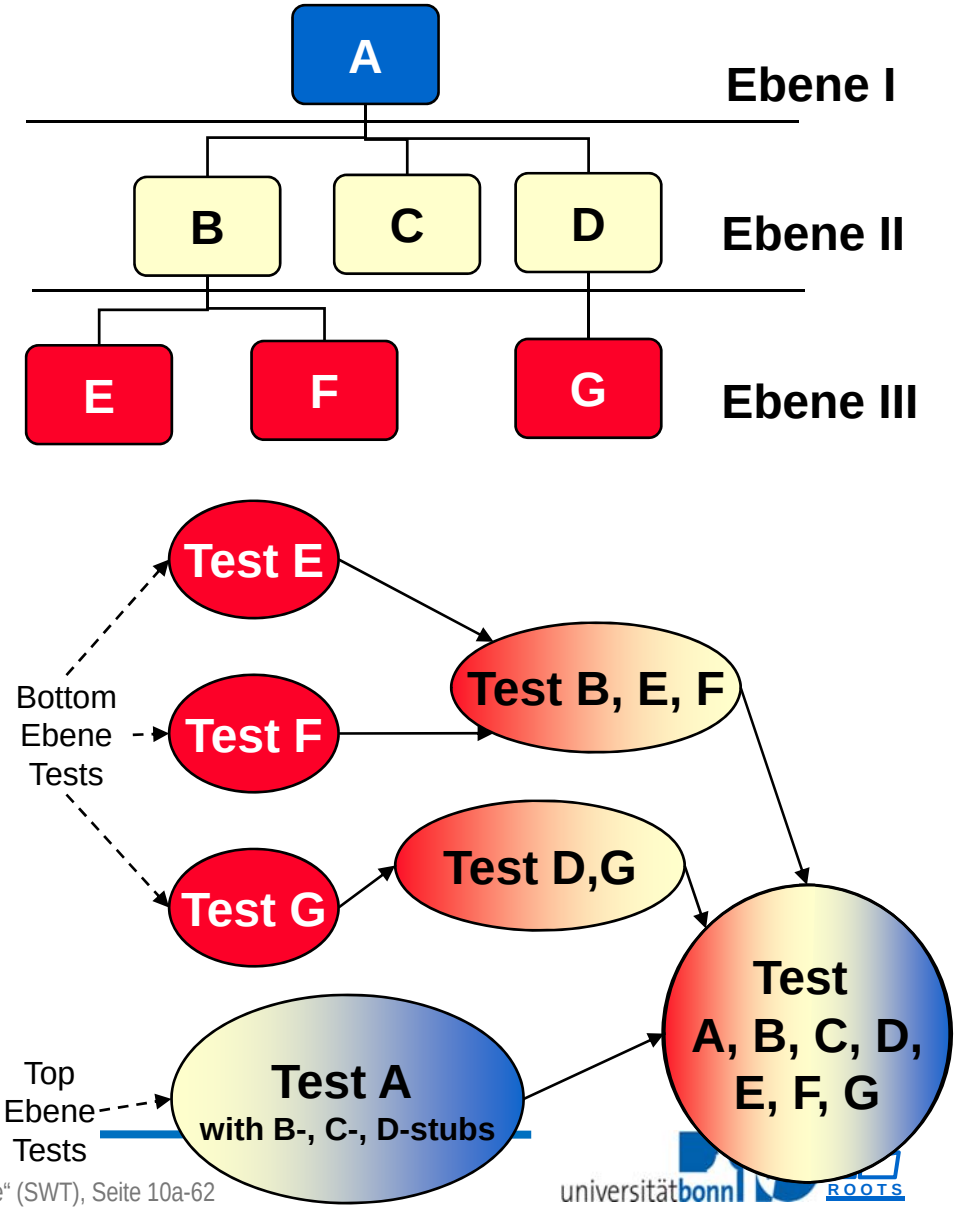


Vor- und Nachteile der Top-down-Integrationstests

- Testfälle können hinsichtlich der Funktionale Anforderungen eines Systems formuliert werden.
- Das Schreiben von Stubs kann schwierig sein:
  - Stubs müssen das Testen aller möglichen Bedingungen erlauben.
  - Evtl. wird eine große Anzahl von Stubs gebraucht, vor allem wenn die tiefste Schicht des Systems viele Methoden beinhaltet.
- Vermeidung einer großen Anzahl von Stubs: Modifizierte Top-down-Teststrategie
  - Teste vor dem Zusammenfügen der Schichten jede Schicht der Systemdekomposition einzeln.
  - Nachteile der modifizierten Top-down-Teststrategie: Sowohl Stubs als auch Treiber werden benötigt.

Sandwich-Teststrategie

- Kombiniert Top-down- und Bottom-up-Teststrategien
  - parallel ausgeführt
- Das System wird in drei Schichten betrachtet
  - Eine „*Zielschicht*“ in der Mitte
  - Eine Schicht über dem „Ziel“
  - Eine Schicht unter dem „Ziel“
  - Die Tests „konvergieren“ auf die „*Zielschicht*“
- „Wie finde ich die Zielschicht, wenn es mehr als drei Schichten gibt?“
  - Heuristik: Versuche die Anzahl der Treiber und Stubs zu minimieren.



Vor- und Nachteile von Sandwich-Tests

- Pro
  - Frühes Testen der Benutzeroberfläche möglich
  - Tests der oberen und unteren Schicht können parallel erfolgen
  - Keine Stubs und Treiber für die obere und untere Schicht erforderlich
    - ⇨ Zielschicht ersetzt Treiber für untere Schicht
    - ⇨ Zielschicht ersetzt Stubs für obere Schicht
- Contra
  - Keine Modultests der Klassen der Zielschicht (B,C,D) vor der Integration
    - ⇨ Zielschicht wird nicht so gründlich getestet
- Lösung
  - Modifizierte Sandwich-Teststrategie: Jede Schicht *vor* der Integration komplett testen
    - ⇨ Mehr Stubs und Treiber erforderlich, aber ...
    - ⇨ ... insgesamt kürzeste Testdauer, da viele Tests parallel ausgeführt werden

Systemtests

Systemtests

- Varianten
  - Funktionale Tests
  - Strukturelle Tests
  - Sicherheit und Zuverlässigkeit
  - Performancetests
  - Installationstests
  - Akzeptanztests

- Je expliziter die Anforderungen, desto einfacher das Testen
  - Qualität der Use Cases
    - ⇨ bestimmt wie leicht / schwer es ist, funktionale Tests zu schreiben
  - Qualität der Subsystemdekomposition
    - ⇨ bestimmt wie schwer es ist, strukturelle Tests zu schreiben
  - Qualität der nicht-funktionalen Anforderungen und Nebenbedingungen
    - ⇨ bestimmt wie schwer es ist, Tests für Sicherheit, Zuverlässigkeit und Performance zu entwerfen.

Sicherheit und Zuverlässigkeit

- Sicherheitstests
  - Versuch, die Sicherheits-Anforderungen des Systems zu verletzen
- Qualitätstests
  - Testet Zuverlässigkeit, Wart- und Verfügbarkeit des Systems
- Konfigurationstests
  - Testet verschiedene Hard- und Softwarekonfigurationen
- Kompatibilitätstest
  - Testet die Kompatibilität zu existierenden Systemen
- Umgebungstest
  - Testet Toleranz gegenüber Hitze, Feuchtigkeit, Erschütterungen, etc, sowie Portabilität
- Testen menschlicher Faktoren
  - Testet Benutzerschnittstelle mit Endbenutzern
- Recovery-Tests
  - Testet den Umgang des Systems mit Fehlern und/oder Datenverlust

Performancetests

- Timingtests
  - Auswertung von Reaktionszeiten und der benötigten Zeiten zur Ausführung einer Funktion
- Volumentests
  - Was passiert, wenn große Datenmengen verarbeitet werden?
- Stresstests
  - Längerer Betrieb an den spezifizierten Grenzen des Systems (max. Anzahl an Benutzern, Lastspitzen)

Akzeptanztests

- Ziel: Zeigen, dass das System fertig für den regulären Betrieb ist.
  - Auswahl der Tests erfolgt durch Kunden/Sponsor
  - Akzeptanztests werden vom Kunden, nicht vom Entwickler durchgeführt.
  - Viele Tests können von den Integrationstests übernommen werden.
- Problem
  - Die Mehrheit aller Fehler wird typischerweise vom Kunden gefunden, nachdem das System in Betrieb genommen wurde.
  - Schlechtes Image, potentielle Kunden könnten abgeschreckt werden.

Daher gibt es zwei zusätzliche Formen von Tests:

- *Alpha-Test*
  - Kunde verwendet die Software am Arbeitsplatz der Entwickler.
  - Software wird in einer kontrollierten Umgebung getestet, wobei der Entwickler immer in der Nähe bleibt, um Bugs zu beheben.
- *Beta-Test*
  - Werden beim Kunden durchgeführt (Entwickler ist nicht anwesend)
  - Software kommt zu realistischem Einsatz in der Zielumgebung.

Zusammenfassung

- Problemfeld
  - Versagen, Fehlerzustand, Fehlerursache
- Fehlerbehandlung
  - Vermeidung, Toleranz, Suche und Behebung
- Testarten
  - Black-box, White-Box, Modultests, Integrationstests, System-Tests
- Testabdeckung
  - Anweisungs-, Verzweigungs- Pfadabdeckung
- Integrationstests
  - Bottom-Up, Top-down, Sandwich, …