

Softwaretechnik und Programmierparadigmen

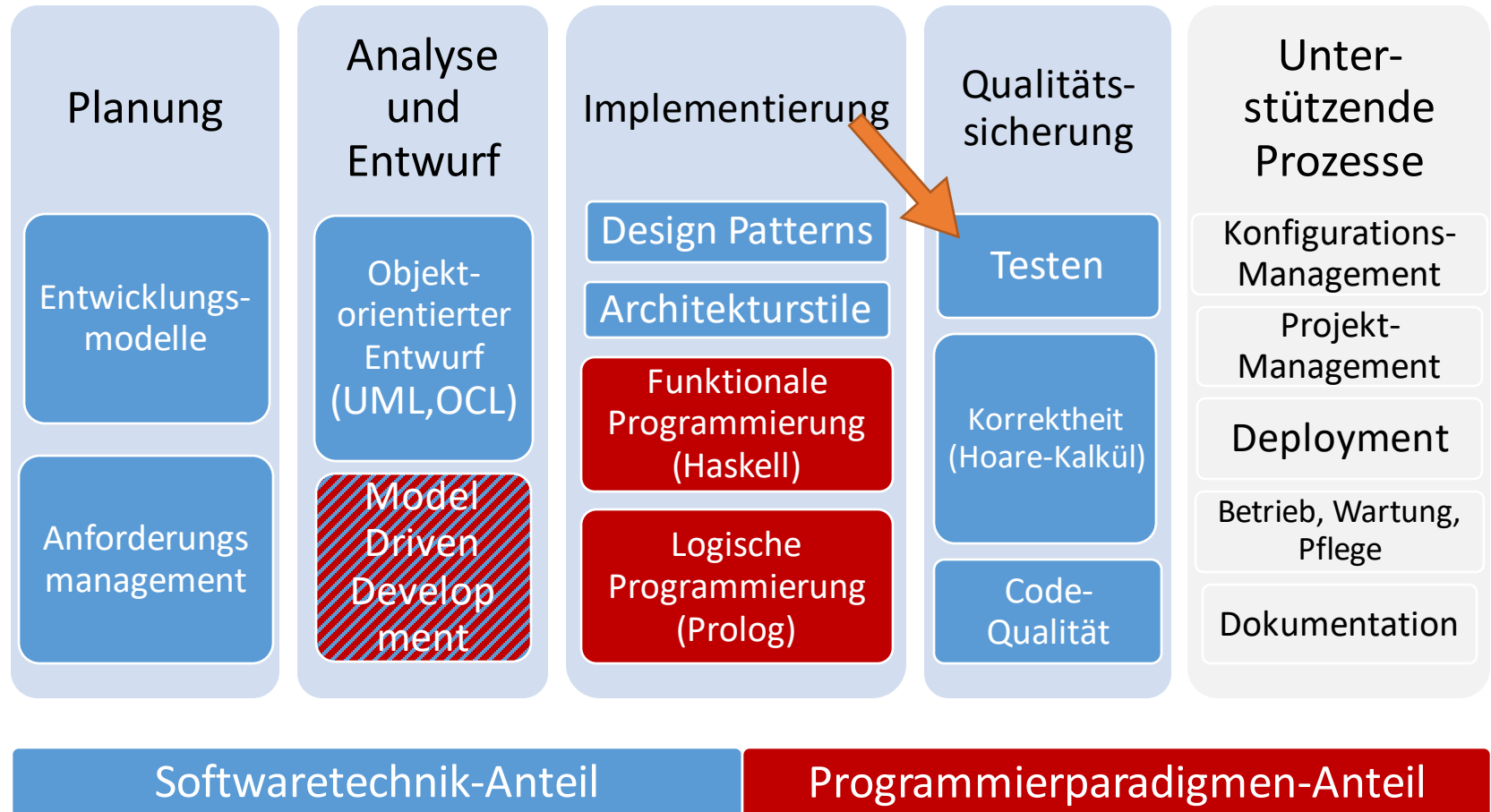
05 Testen

Prof. Dr. Sabine Glesner
Software and Embedded Systems Engineering
Technische Universität Berlin

Ankündigung

- Zweite Wiederholungsprüfung am 19.1.2023
- Bei Bedarf frühzeitig melden über **swtpp@sese.tu-berlin.de**

Diese VL



Motivation

Alles ist genau spezifiziert, dann kann ja nichts mehr schiefgehen, oder?



“Quality is never an accident; it is always the result of **intelligent effort.**”
– John Ruskin

Qualitätssicherung

Prozessqualität

Befasst sich mit der Verbesserung der Entstehung des Software-Produkts (Prozess).

- Managementprozesse und Entwicklungsmodelle
- Software-Infrastruktur (Build-Automatisierung, Testautomatisierung, ...)

Produktqualität

Befasst mit der Verbesserung der genannten Qualitätsmerkmale des Software-Produkts.

- Korrektheit
- Testen
- Konventionen
- Kommentare
- Statische Analyse
- Metriken
- ...

← Heute

Inhalt

Testen

- Einführung
- Strukturorientierte Tests (White-Box Tests)
- Funktionsorientierte Tests (Black-Box Tests)
- Ausblick: Wie wird getestet?

Inhalt

Testen

- Einführung
- Strukturorientierte Tests (White-Box Tests)
- Funktionsorientierte Tests (Black-Box Tests)
- Ausblick: Wie wird getestet?

Testen

„Aktivität, in der ein System oder eine Komponente **unter bestimmten Bedingungen** ausgeführt wird, während die **Ergebnisse beobachtet** werden, um ein Teil des Systems oder der Komponente **zu bewerten.**“
- ISO/IEC/IEEE:24765

Testen nimmt **großen Teil des Aufwands** für Software ein:

- **ca. 50% der Zeit** und mehr als **50% der Kosten** [„The Art of Software Testing“ Myers et al., 2011]
- **21% der Design-Time** für Debuggen und Testen [Embedded Market Study, 2014]

Trotzdem sind **Software-Fehler** allgegenwärtig:

- **59 Mrd Dollar Schaden** durch Bugs allein in den **USA** [NIST Report, 2002]
- **84 Mrd Euro Schaden** in **Deutschland** [IX-Studie, 2006]
- Seitdem viel mehr neue Software, auch im **sicherheitskritischen Bereich**

Analytische Maßnahmen im Vergleich

Formale Verifikation (*Hoare Logik, Weakest Precondition, Model Checking...*)

- Kann **Übereinstimmung mit einer formalen Spezifikation** nachweisen
- Spezielle Behandlung von **Schleifen/Rekursion** und **Terminierung** nötig
- **Komplexe Fälle** wegen unvollständiger **Automatisierung** nicht behandelbar

Testen

- Kann **Fehler im Produkt** aufdecken, bevor es ausgeliefert wird
- Kein Anspruch auf **Vollständigkeit**, nur **Stichproben**
- **Aussagekraft** ist durch **Auswahl** und Auswertung der **Testfälle** bestimmt

„Program **testing** can be used to show the **presence of bugs**,
but never to show their absence.“

- Edsger W. Dijkstra

Testklassifizierung

Wer testet?

- Entwickler, unabhängige Tester, Kunde bzw. Anwender

Wann wird getestet?

- Definiert durch den Softwarelebenszyklus (bzw. Entwicklungsprozess)

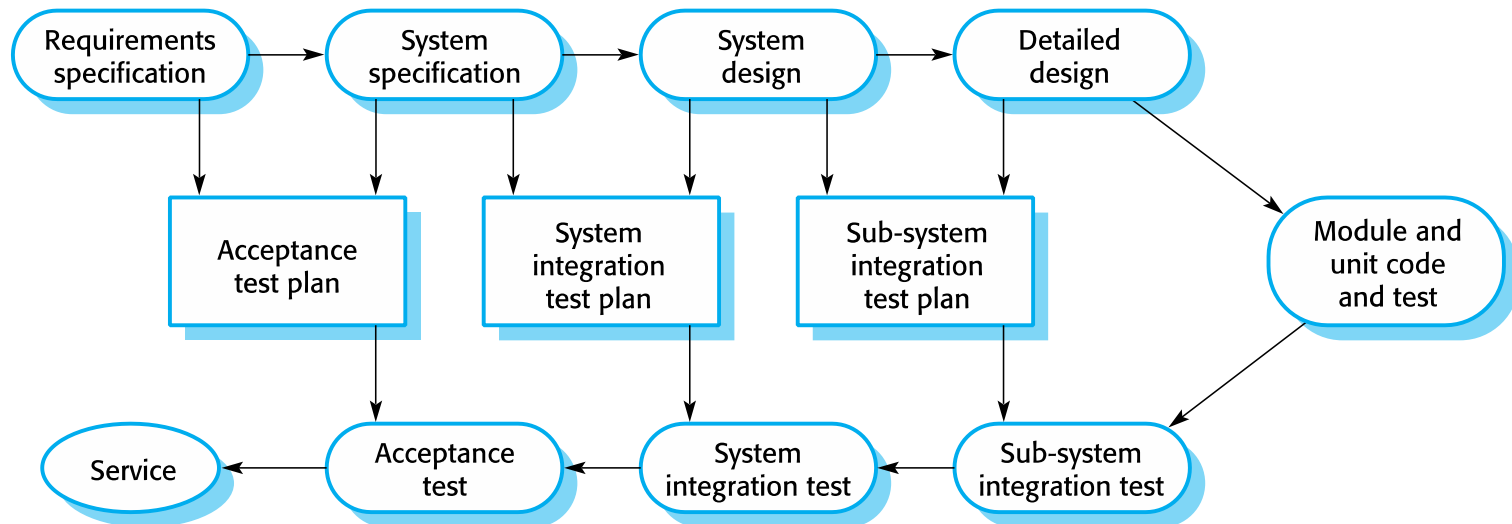
Was wird getestet?

- Testobjekte in Testumgebung
- Testbare Teile der Anforderungen oder der Spezifikation
- Funktionale vs. nicht-funktionale Tests
- Auswahl der Testfälle zur Erfüllung der Testziele

Wie wird getestet?

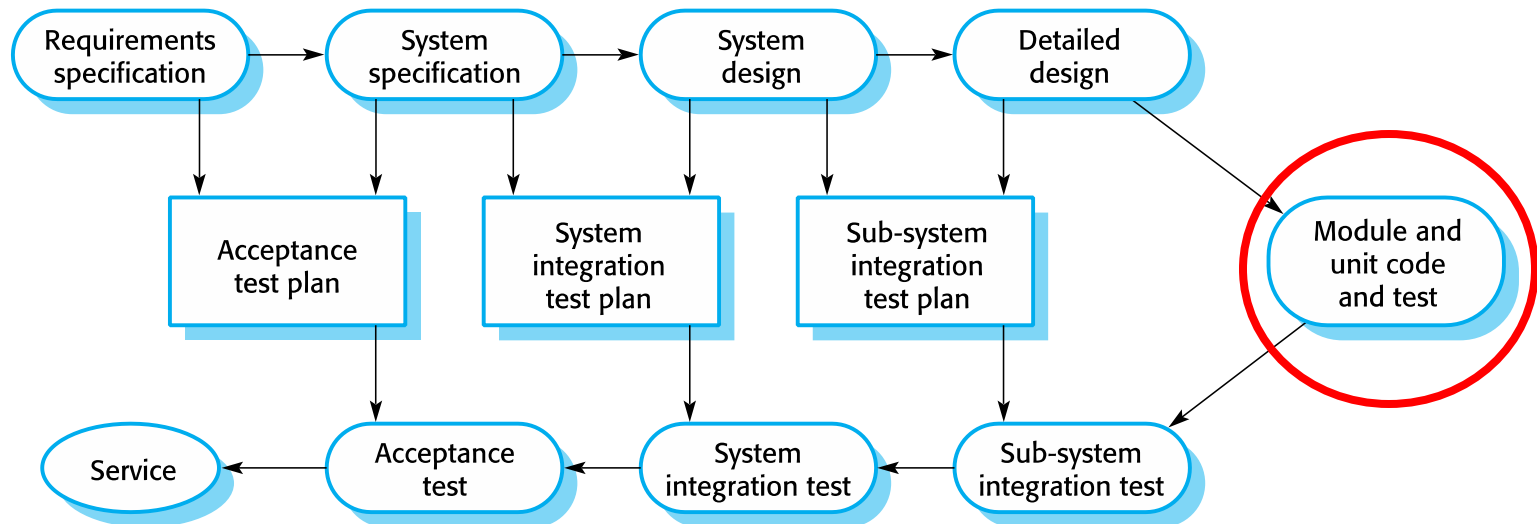
- Automatisierung
- Tools
- Bewertungskriterien

Testen im Softwarelebenszyklus



[Ian Sommerville, Software-Engineering, Chapter 2](#)

Modultest (*Unit-Test*)



Modultest (*Unit-Test*)

Testobjekt

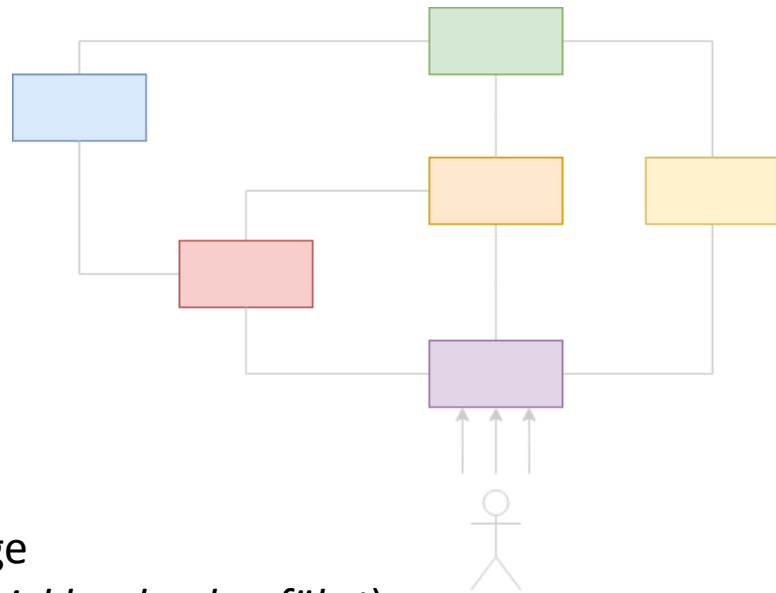
- Kleinste Testeinheit
- Klasse, Modul, Unit, ...
- Isoliert von anderen Einheiten

Testumgebung

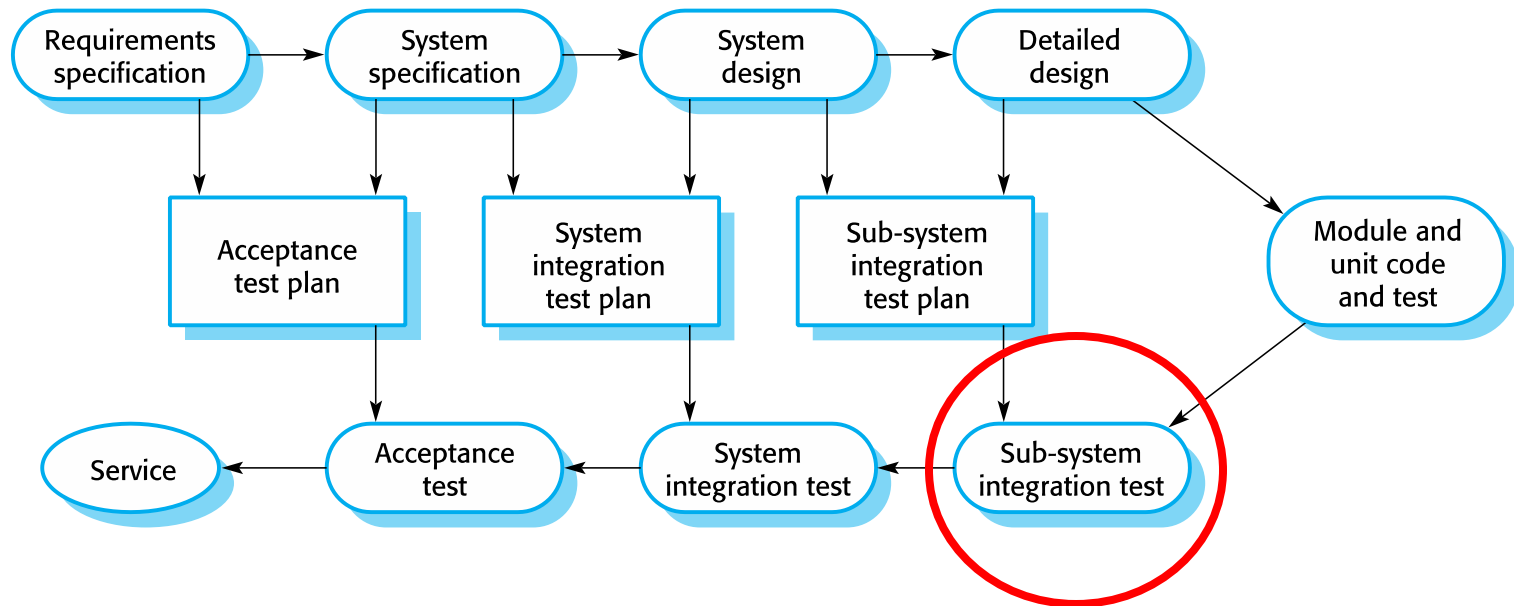
- Programmiersprache und -werkzeuge
- Entwicklungsnah (*häufig durch Entwickler durchgeführt*)
- Häufig mit Kenntnis des Programms (*White-Box*)

Testziele

- Funktionalität (*Ist die Berechnung korrekt?*)
- Robustheit (*Sonderfälle, unzulässige Eingaben, ...*)
- Effizienz (*Speicher, Zeit, ...*)
- Wartbarkeit (*Code-Struktur, Kommentare, ...*)



Integrationstest



[Ian Sommerville, Software-Engineering, Chapter 2](#)

Integrationstest

Testobjekt

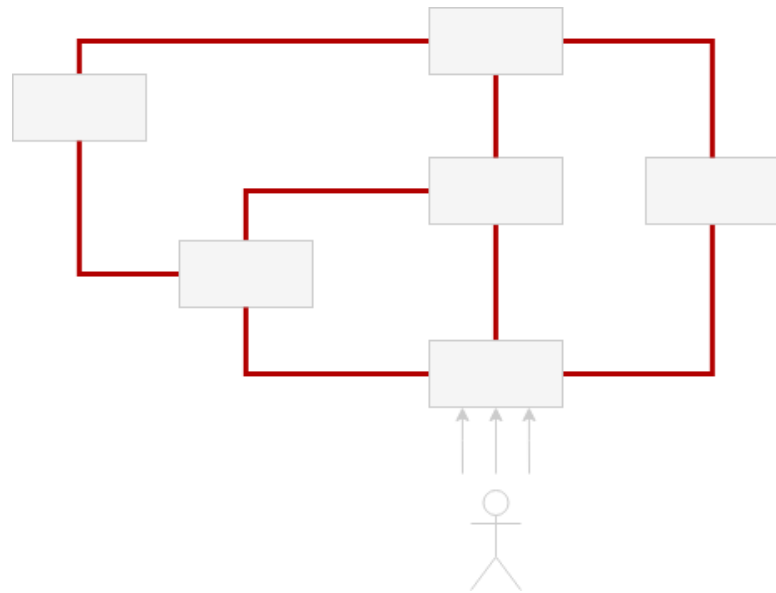
- Schnittstellen der zusammengesetzten Komponenten (auch Fremdsysteme)

Testumgebung

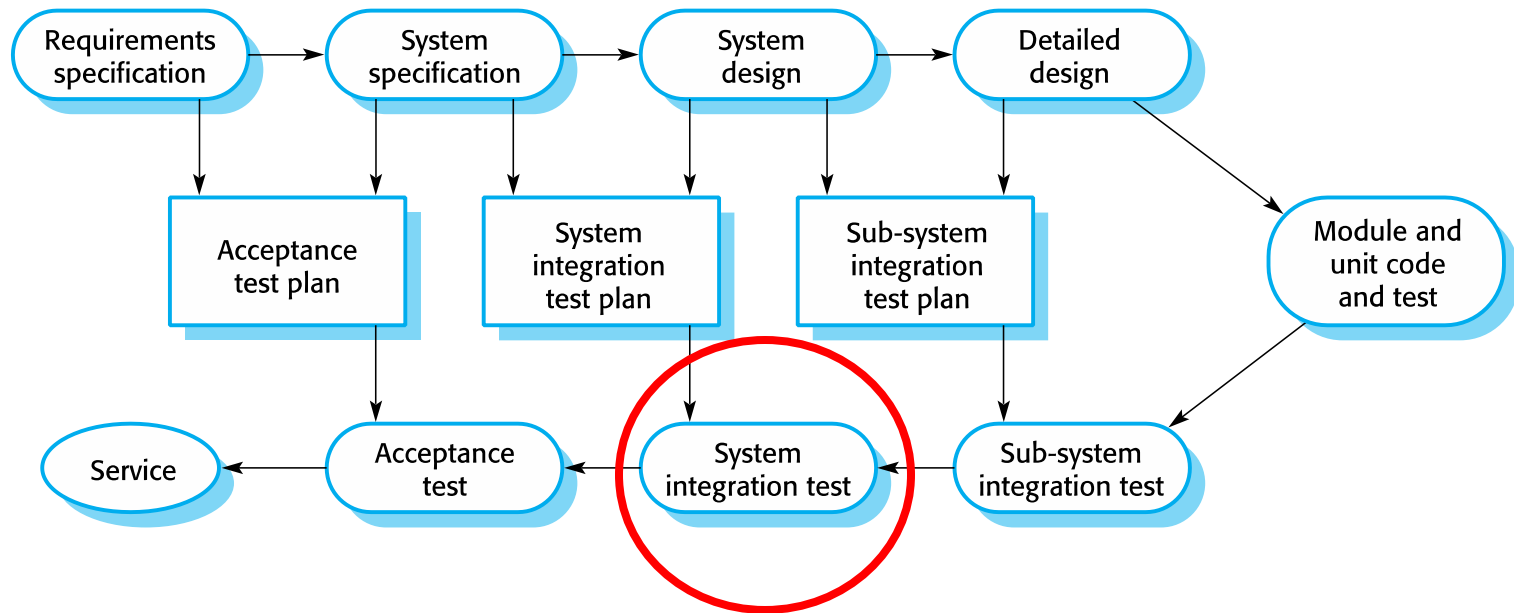
- Testdaten / -objekte notwendig
- Evtl. Wiederverwendbarkeit von Modultestdaten
- Monitoring / Protokollierung der Schnittstellen

Testziele

- Korrektes Schnittstellenformat (*bspw. fehlende Informationen*)
- Korrekter Datenaustausch (*keine bzw. falsche Daten werden übermittelt, inkonsistente Interpretation von Daten, Timing – und Überlastungsprobleme*)
- Vermeiden des „Big Bang“ (*Welche Integrationsstrategie?*)



Systemtest



[Ian Sommerville, Software-Engineering, Chapter 2](#)

Systemtest

Testobjekt

- Gesamte System aus Sicht des Anwenders

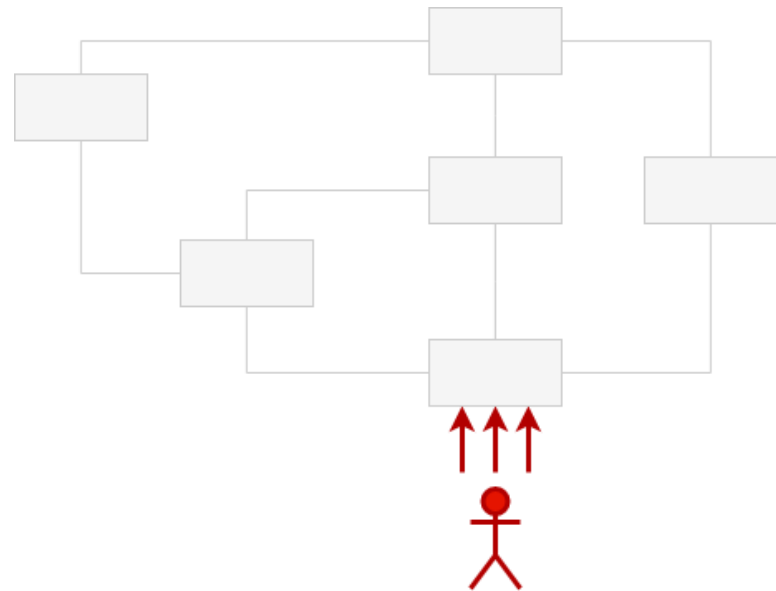
In der Verantwortung des Entwicklers

Testumgebung

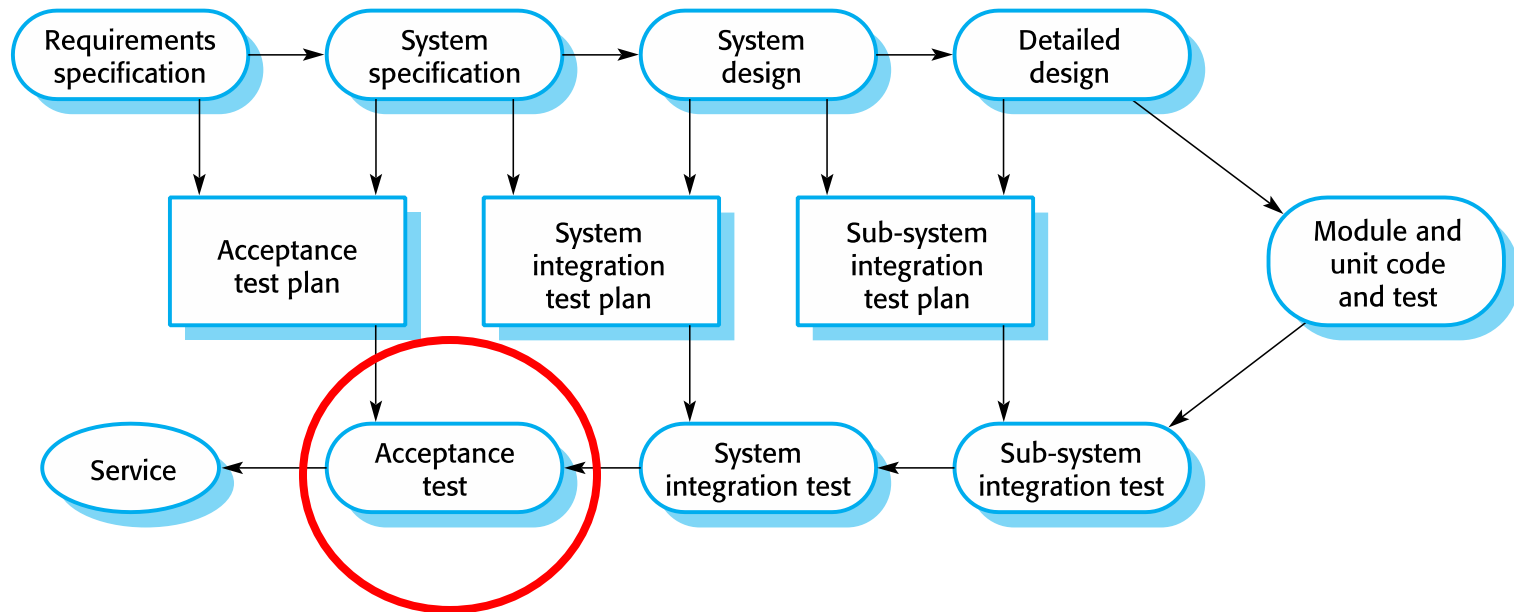
- Umgebung möglichst ähnlich zur Produktionsumgebung
- Achtung: Hohe Fehlerkosten bei Verwendung der Produktivumgebung (Downtime, Störung des Betriebs)

Testziele

- Konformität zu gestellten Anforderungen
- Prüfung der Dokumentation (Systemhandbuch, Benutzerhandbücher, ...)



Abnahmetest



Abnahmetest

Testobjekt

- Gesamte System aus Sicht des Anwenders

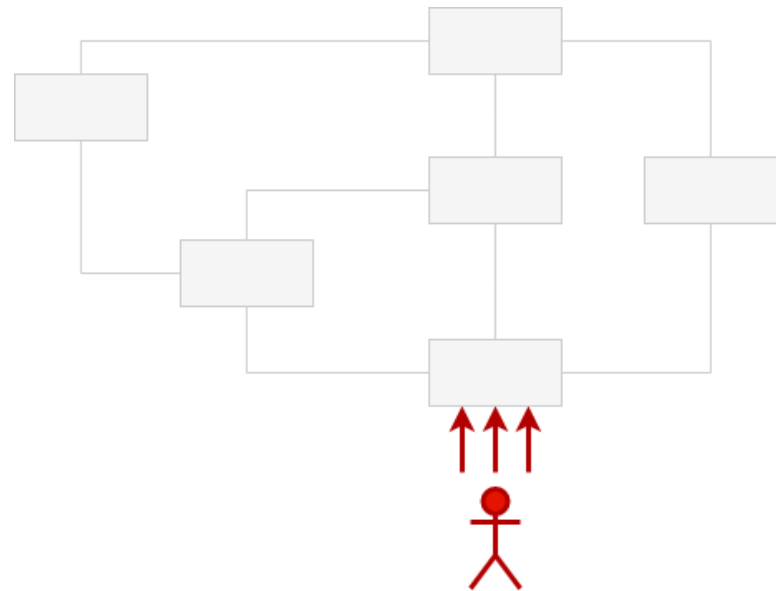
In der Verantwortung des Kunden

Testumgebung

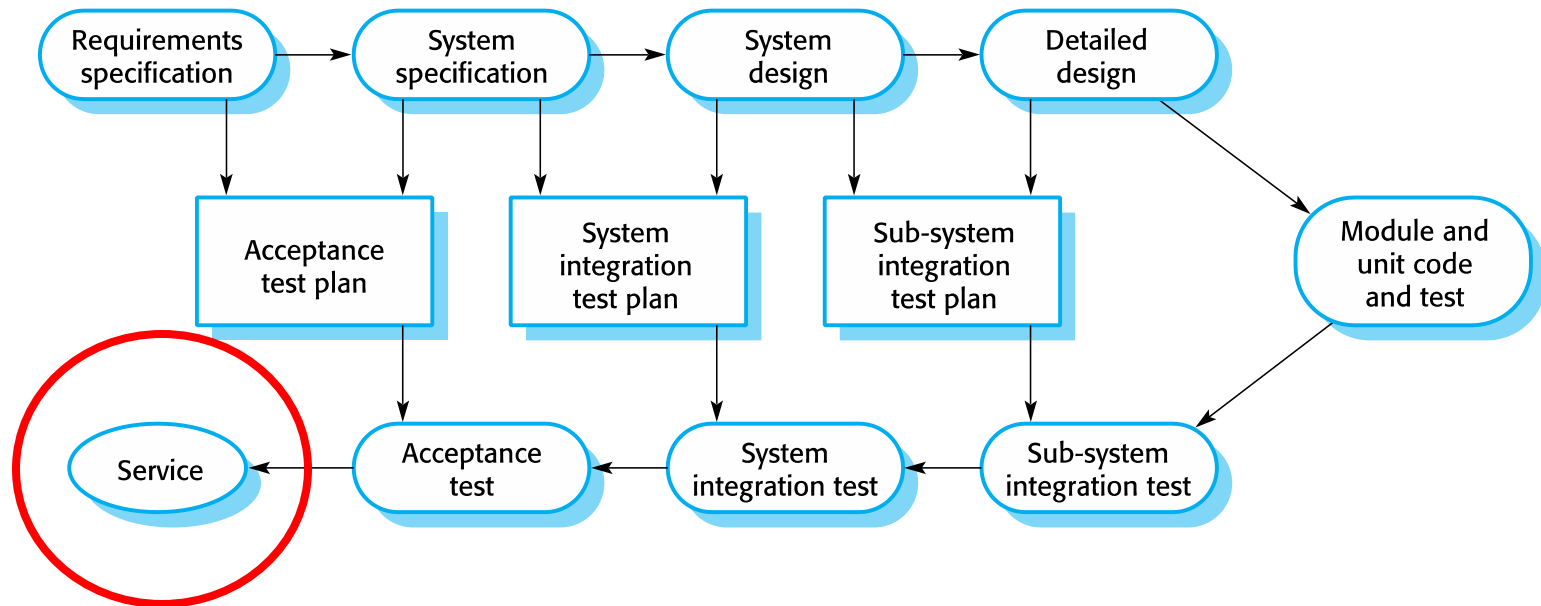
- Umgebung möglichst ähnlich zur Produktivumgebung
- Achtung: Hohe Fehlerkosten bei Verwendung der Produktivumgebung

Testziele

- Vertragliche Akzeptanz (vereinbarte Anforderungen)
- Akzeptanz der Benutzer
- Akzeptanz des Systembetreibers

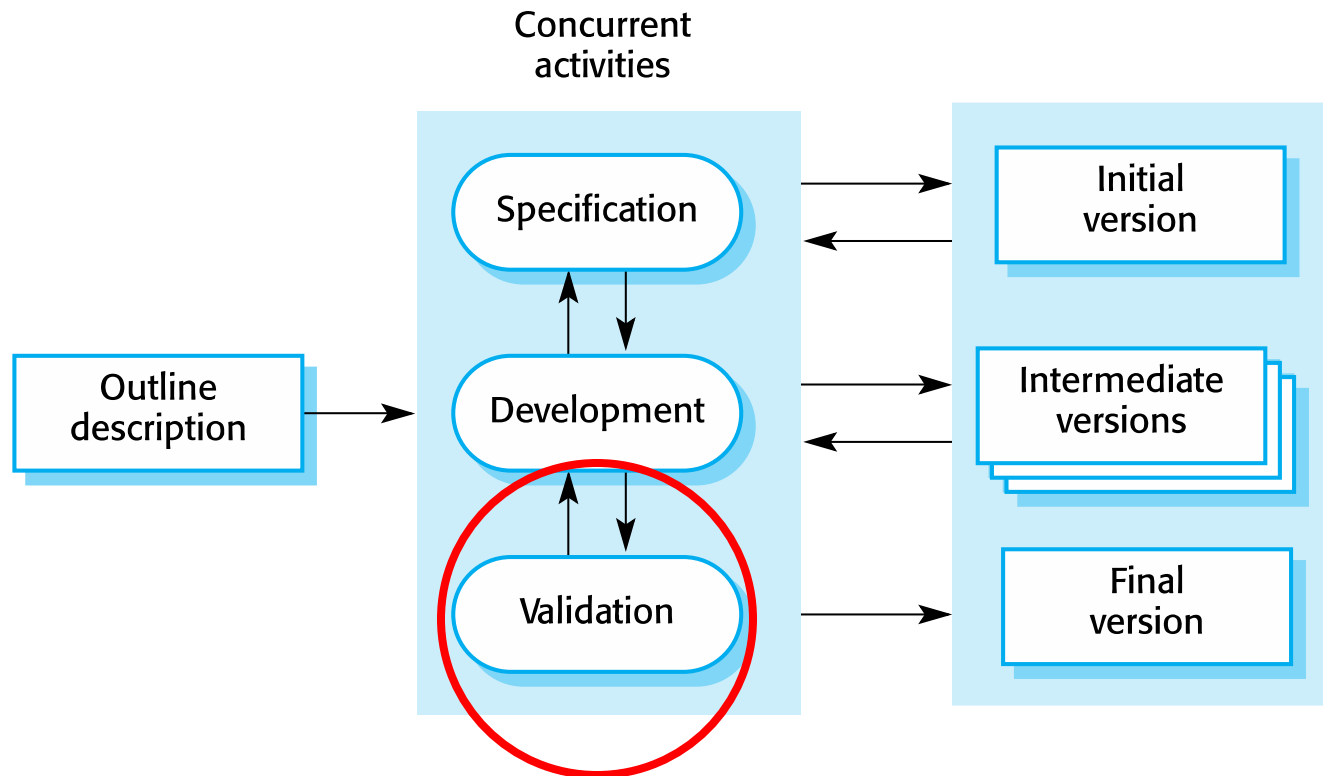


Und nun?



[Ian Sommerville, Software-Engineering, Chapter 2](#)

Regressionstest



[Ian Sommerville, Software-Engineering, Chapter 2](#)

Regressionstest

- Wiederkehrendes Testen bei Änderungen
 - Bestenfalls: Automatisierung
- Betrifft alle Phasen des Softwarelebenszyklus
- Vollständiger Regressionstest in allen Phasen ist teuer
 - Einschränkung bzw. Priorisierung der Tests möglich

Testziele

- Prüfung der Fehlerbehebung bzw. neuer Funktionalität
- Und: Vermeiden von unbeabsichtigten Seiteneffekten (Veränderung des ursprünglichen Verhaltens)

Gründe für Regressionstests

Softwarewartung

- Software altert nicht, aber ...
- Anpassung an eine veränderte Produktionsumgebung
- Hotfixes bei kritischen Fehlern

Weiterentwicklung bzw. inkrementelle Entwicklung

- Wiederkehrende Modul-, Integrations-, System- und Akzeptanztests
- Anpassung der Tests im Softwarelebenszyklus an (inkrementelles) Entwicklungsmodell

[Ian Sommerville, Software-Engineering, Chapter 2](#)

Was wird getestet? Anforderungen

Funktionaler Test

*Prüfung der von außen sichtbaren **Ein- und Ausgaben** des Testobjekts*

- Als Basis dienen funktionale Anforderungen

Nicht-funktionaler Test

Prüfung nicht-funktionaler Eigenschaften

- Als Basis dienen nicht-funktionale Anforderungen
- Häufig auf Systemtestebene
- Beispiele
 - Lasttest: Verhalten bei steigender Last
 - Stresstest: Verhalten bei Überlastung
 - Robustheitstest: Verhalten bei Ausfällen oder anormalen Bedingungen
- Problem: Quantifizierung von Anforderungen:
 - nicht: „akzeptable Antwortzeiten sind wichtig“
 - sondern: „Antwortzeit höchstens 5 Sekunden, in 80% der Fälle kleiner als 3 Sekunden“

Was wird getestet? Auswahl von Testfällen

Es gibt kein **allgemeingültiges** Vorgehen beim Testen

- Stattdessen müssen Prozesse auf den spezifischen Fall zugeschnitten werden

Optimal wäre das **Testen aller möglichen Ausführungen**, aber...

- Das hieße **alle möglichen Kombinationen** von Eingaben
- Dies führt zu einer **kombinatorischen Explosion** des Zustandsraumes
- *Erschöpfend (Exhaustive)* zu testen ist also i.A. **nicht durchführbar**

Wie bestimmt man, **welche Eingaben** getestet werden?

Auswahl von Testfällen

Testauswahl durch Entwickler

- ✓ Durch Erfahrung: Schwierige Stellen sind auch fehleranfällig
- × Aber Entwickler können „betriebsblind“ sein und Anteile übersehen
- Systematische Auswahl gesucht

Strukturorientierter Test (*White-Box Testing*)

Auswahl nach dem **Aufbau des Moduls**

- Ziel: hoher **Überdeckungsgrad** auf der Struktur
- **Kontrollflussüberdeckung** und **Datenflussüberdeckung**

Funktionsorientierter Test (*Black-Box Testing*)

Auswahl nach **Eigenschaften der Eingabe** oder der **Spezifikation**

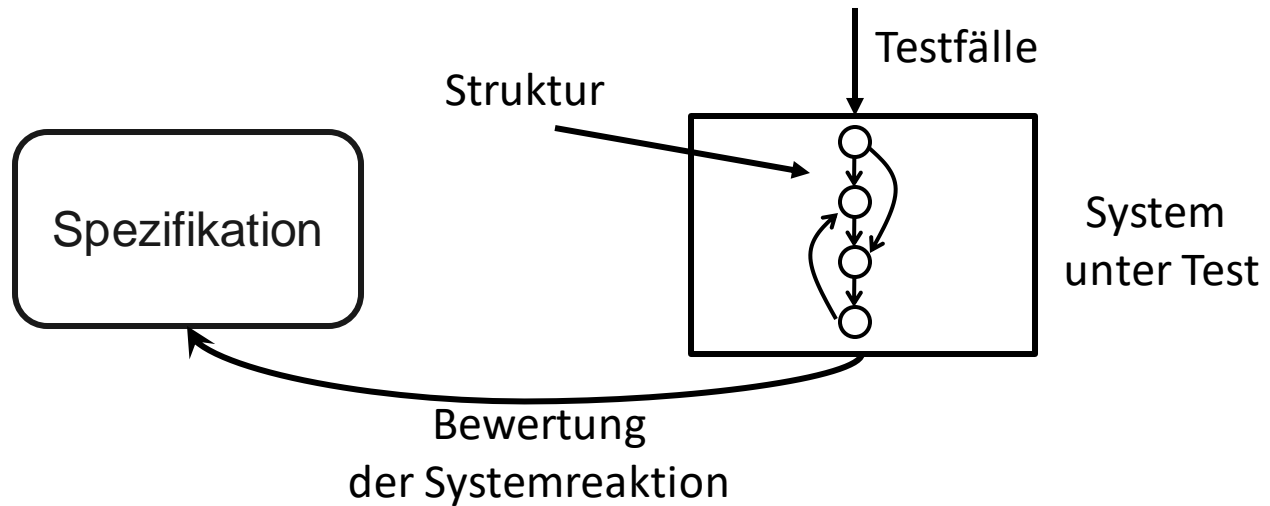
- Test **typischer Anwendungsfälle** (Nutzungsprofile)
- gezieltes Testen von **Rand- und Sonderfällen**

Inhalt

Testen

- Einführung
- **Strukturorientierte Tests (White-Box Tests)**
- Funktionsorientierte Tests (Black-Box Tests)
- Ausblick: Wie wird getestet?

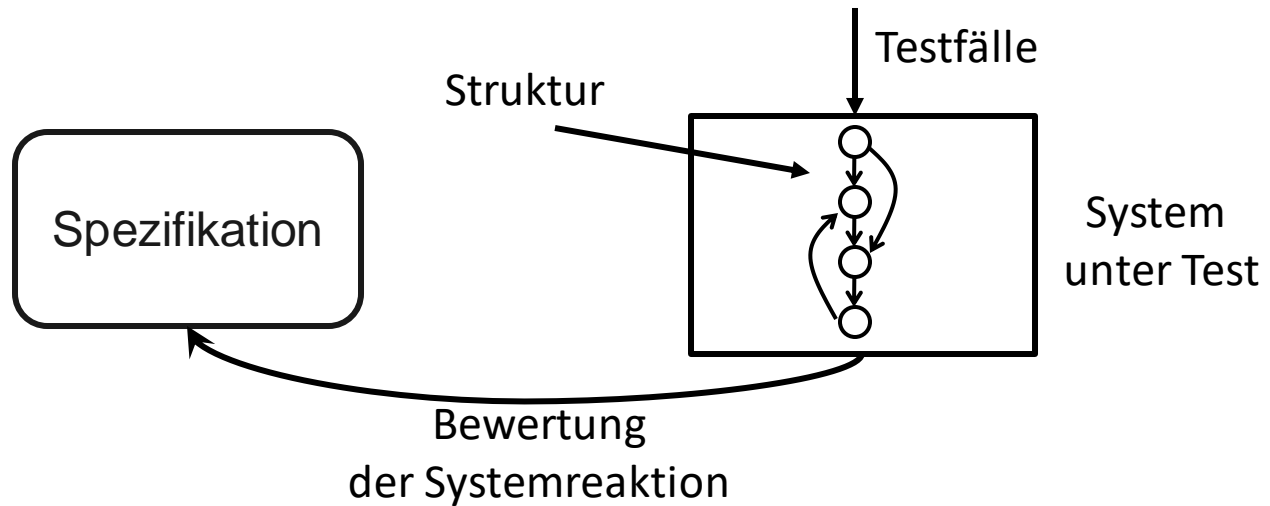
Strukturorientierter Test



Auch **White-Box**-Tests genannt

- Interner Aufbau (z.B. Quellcode) muss vorliegen
- Testfälle können erst **nach der Implementierung** aufgestellt werden
- Testfälle ergeben sich aus der Spezifikation und **Struktur der Software**

Strukturorientierter Test



Ziel ist es, einen **Überdeckungsgrad** zu erreichen

$$\text{Überdeckung (coverage)} = \frac{\text{Anzahl überdeckter Merkmale}}{\text{Anzahl vorhandener Merkmale}}$$

Kontrollflussüberdeckung

$$\text{Überdeckung (coverage)} = \frac{\text{Anzahl überdeckter Merkmale}}{\text{Anzahl vorhandener Merkmale}}$$

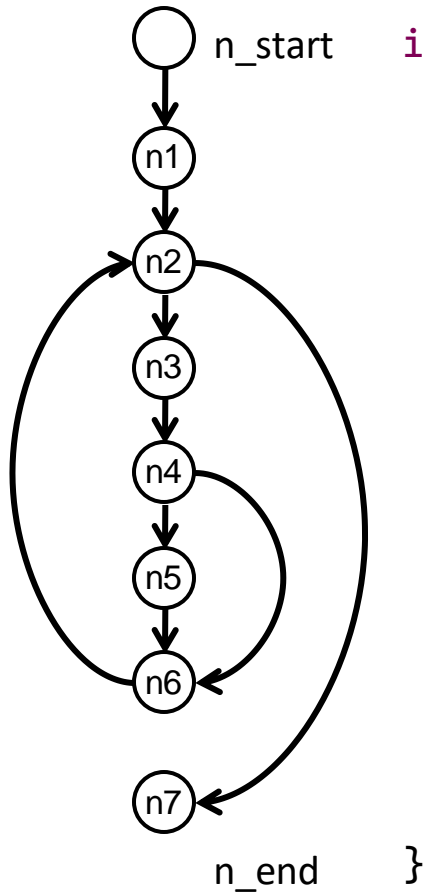
Misst die Überdeckung anhand des **Kontrollflussgraphen**

Ein Kontrollflussgraph ist ein gerichteter Graph, wobei die **Knoten Anweisungen** bzw. Basisblöcke darstellen, **Kanten den Kontrollfluss** (vgl. VL08)

Verschiedene Merkmale sind möglich

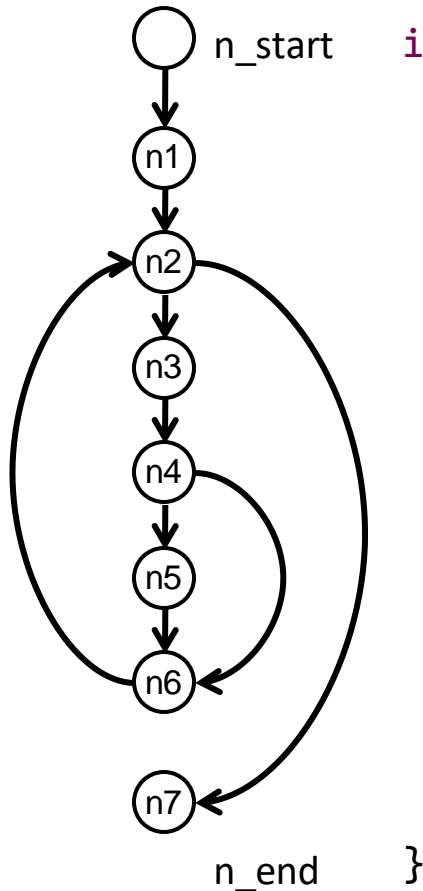
- **Anweisungsüberdeckung** (Knoten im Kontrollflussgraph)
- **Zweigüberdeckung** (Kanten im Kontrollflussgraph)
- **Pfadüberdeckung** (Pfade im Kontrollflussgraph)

Beispiel - countVowels



```
int countVowels(String s) {  
    1 int cnt = 0, vowels = 0;  
    2 while (cnt < s.length()) {  
        3 char v = s.charAt(cnt);  
        4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
        {  
            5 vowels++;  
        }  
        6 cnt++;  
    }  
    7 return vowels;  
}
```

Anweisungsüberdeckung



```
int countVowels(String s) {
```

```
1 int cnt = 0, vowels = 0;
```

```
2 while (cnt < s.length()) {
```

```
3 char v = s.charAt(cnt);
```

```
4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
{
```

```
5     vowels++;
```

```
6 }  
    cnt++;
```

```
7 }  
return vowels;
```

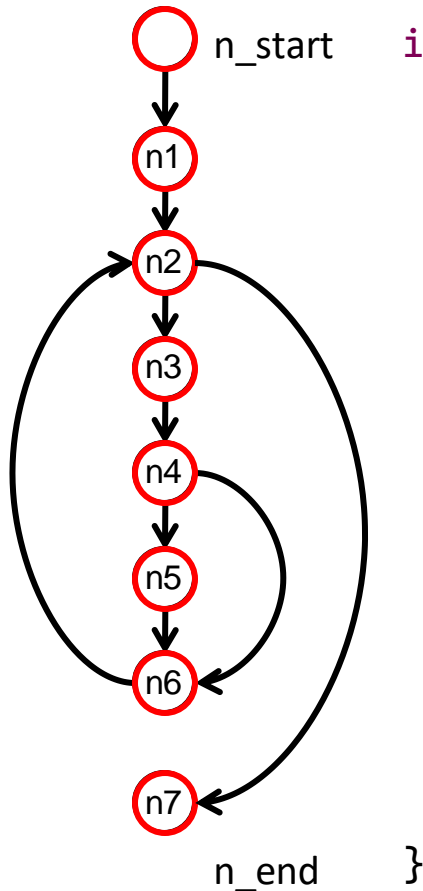
```
}
```

Wie hoch ist die **Anweisungsüberdeckung**
für den Testfall **countVowels("aa")**

Anzahl überdeckter **Knoten**

Anzahl vorhandener **Knoten**

Anweisungsüberdeckung

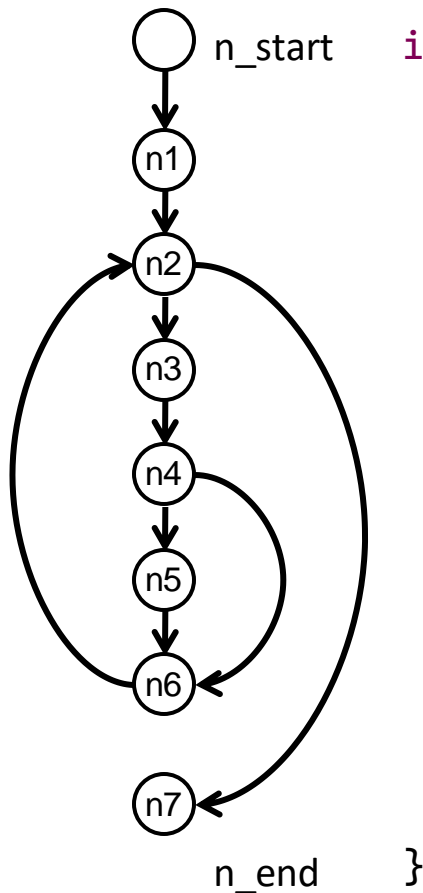


```
int countVowels(String s) {  
    1 int cnt = 0, vowels = 0;  
    2 while (cnt < s.length()) {  
        3 char v = s.charAt(cnt);  
        4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
        {  
            5 vowels++;  
        }  
        6 cnt++;  
    }  
    7 return vowels;  
}
```

Wie hoch ist die **Anweisungsüberdeckung**
für den Testfall **countVowels("aa")**

$$\frac{8}{8} = 100 \%$$

Zweigüberdeckung



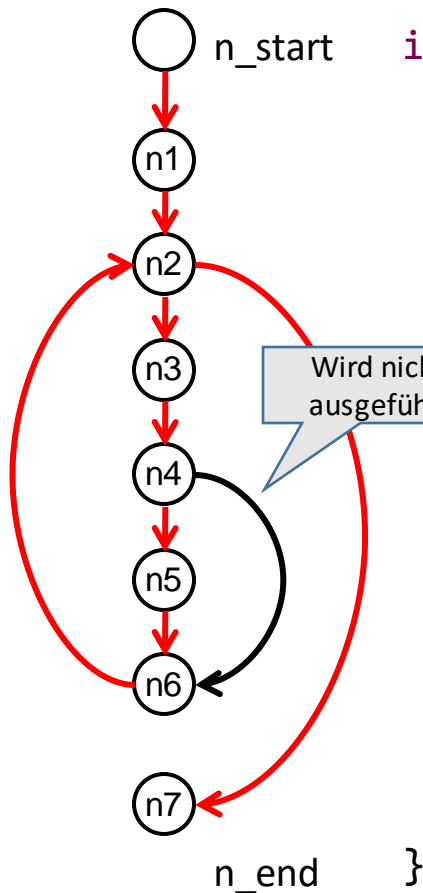
```
int countVowels(String s) {  
  1 int cnt = 0, vowels = 0;  
  2 while (cnt < s.length()) {  
    3 char v = s.charAt(cnt);  
    4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
      {  
        5 vowels++;  
      }  
    6 cnt++;  
  }  
  7 return vowels;  
}
```

Wie hoch ist die **Zweigüberdeckung**
für den Testfall **countVowels("aa")**

Anzahl überdeckter **Kanten**

Anzahl vorhandener **Kanten**

Einfache Überdeckungskriterien



```
int countVowels(String s) {
```

```
1 int cnt = 0, vowels = 0;
```

```
2 while (cnt < s.length()) {
```

```
3 char v = s.charAt(cnt);
```

```
4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
{
```

```
5     vowels++;
```

```
6 }  
    cnt++;
```

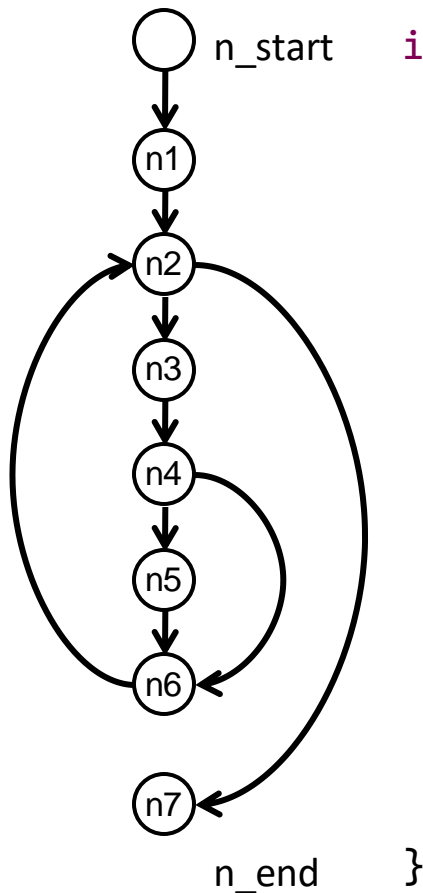
```
7 }  
return vowels;
```

```
}
```

Wie hoch ist die **Zweigüberdeckung**
für den Testfall **countVowels("aa")**

$$\frac{8}{9} = 89 \%$$

Zweigüberdeckung



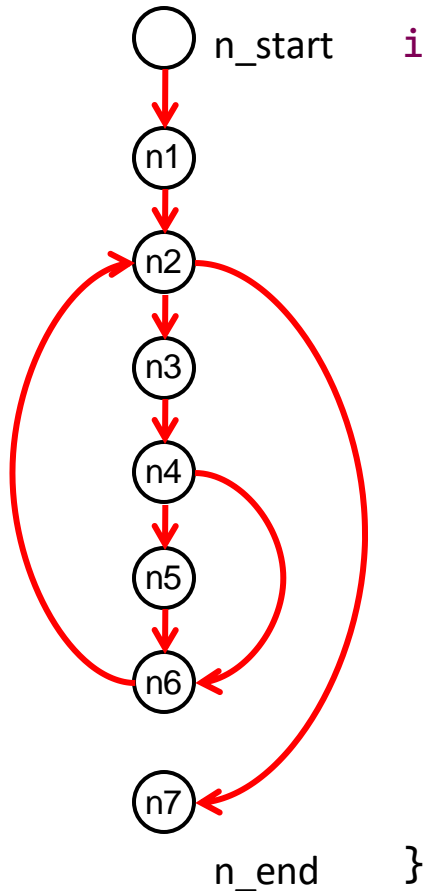
```
int countVowels(String s) {  
  1 int cnt = 0, vowels = 0;  
  2 while (cnt < s.length()) {  
    3 char v = s.charAt(cnt);  
    4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
      {  
        5 vowels++;  
      }  
    6 cnt++;  
  }  
  7 return vowels;  
}
```

Wie hoch ist die Zweigüberdeckung
für den Testfall `countVowels("ab")`

Anzahl überdeckter **Kanten**

Anzahl vorhandener **Kanten**

Einfache Überdeckungskriterien



```
int countVowels(String s) {  
  1 int cnt = 0, vowels = 0;  
  2 while (cnt < s.length()) {  
    3 char v = s.charAt(cnt);  
    4 if(v=='a' | v=='e' | v=='i' | v=='o' | v=='u')  
      {  
        5 vowels++;  
      }  
    6 cnt++;  
  }  
  7 return vowels;  
}
```

Wie hoch ist die Zweigüberdeckung
für den Testfall `countVowels("ab")`

$$\frac{9}{9} = 100 \%$$

Anweisungs- und Zweigüberdeckung

100% Anweisungsüberdeckung

- Notwendiges, aber nicht hinreichendes Testkriterium
- Kann Code finden, der nicht ausführbar ist
- Als eigenständiges Testverfahren nicht geeignet
- **Fehleridentifizierungsquote: 18%**

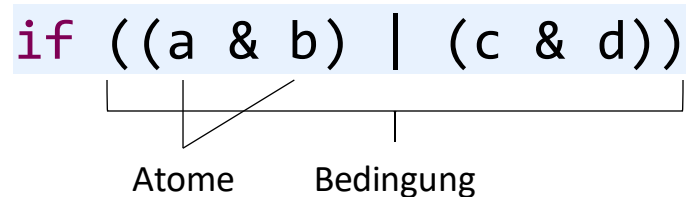
100% Zweigüberdeckung

- Gilt als *das minimale Testkriterium*
- Kann nicht ausführbare Programmzweige finden
- Kann häufig durchlaufene Programmzweige finden (Optimierung)
- **Fehleridentifikationsquote: 34%**

Was ist mit den anderen Bedingungen im Beispiel wie $v == 'i'$?

Bedingungsüberdeckung

Misst die Überdeckung anhand der vorhandenen **Bedingungen**



Verschiedene Merkmale sind möglich

- **Einfache** Bedingungsüberdeckung (Wahrheitswerte der **Atome**)
- **Mehrfache** Bedingungsüberdeckung (**Kombinationen** der Atome)
- **Minimale** Mehrfach-Bedingungsüberdeckung
- **Modifizierte Bedingungs-/Entscheidungsüberdeckung**

Einfache Bedingungsüberdeckung

Jede **atomare Teilentscheidung** muss einmal *true* und *false* sein

```
if ((a & b) | (c & d))
```

- a, b, c, d müssen jeweils einmal true, einmal false sein
- Beispielkombinationen (abcd) = (1100, 0011)

Welche Tests benötigt man für die
einfache Bedingungsüberdeckung von countVowels?

```
if (v=='a' | v=='e' | v=='i' | v=='o' | v=='u')
```

- Die Eingabe ist ein String
- Jeder Vokal muss vorkommen, damit jedes Atom mal true ist
- Möglicher Testfall: “aeiou”

Mehrfache Bedingungsüberdeckung

Überdeckt jede mögliche **Kombination atomarer Teilentscheidungen**

```
if ((a & b) | (c & d))
```

➤ Eingaben (abcd) = (0000, 0001, 0010, 0011, 0100, 0101, ..., 1111)

Bei **n** Teilentscheidungen bis zu **2ⁿ** Testfälle

- Kombinatorische Explosion
- Viele **redundante Testfälle** ohne Mehrwert

Besser wäre eine **sinnvolle Auswahl** der möglichen Kombinationen...

Minimale Mehrfach-Bedingungsüberdeckung

Jede **atomare Bedingung** muss einmal *true* und einmal *false* sein und die **Gesamt-Bedingung** muss min. einmal *true* und einmal *false* werden

```
if ((a & b) | (c & d))
```

- (abcd) = (1111), Bedingung ist true
- (abcd) = (0000), Bedingung ist false

Möglich wären auch (1000,0111) oder (1110,0001) oder (0100,1011)...

- Der Einfluss der Atome auf das Ergebnis ist unklar

Warum reicht die Beispiel-Lösung für die einfache
Bedingungsüberdeckung (abcd) = (1100,0011) hier **nicht** aus?

Die **Gesamt-Bedingung** wäre nie *false*

Modifizierte Bedingungs-/ Entscheidungsüberdeckung

Die Testfälle müssen demonstrieren, dass **jede atomare Teilentscheidung** den Wahrheitswert der Gesamtentscheidung unabhängig von den anderen Teilentscheidungen **beeinflussen** kann

```
if ((a & b) | (c & d))
```

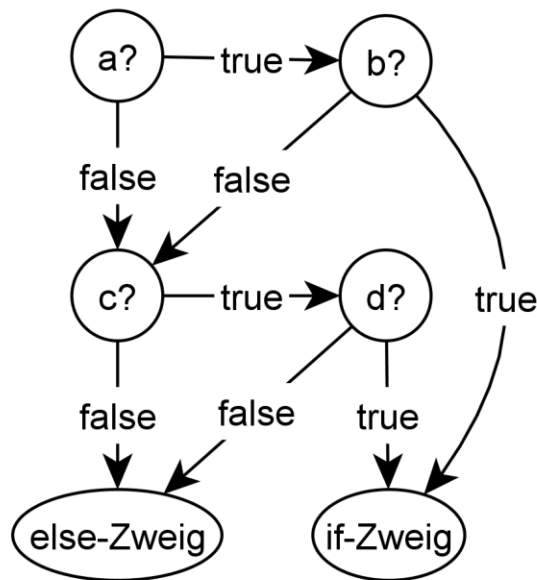
a	b	c	d	(a&b) (c&d)	entscheidend
false	true	true	false	false	a false, d false
true	true	true	false	true	a true, b true
true	false	false	true	false	b false, c false
false	true	true	true	true	c true, d true

Bedingte Auswertung vs Bedingungsüberdeckung

In Java gibt es die Operatoren **&&** und **||** zur bedingten Auswertung

```
if ((a && b) || (c && d))
```

Diese werden im CFG durch mehrere Verzweigungen abgebildet



- mehr Zweige, dafür atomare Bedingungen für Entscheidungen
- Hier erreicht man mit 100% Zweigüberdeckung die **minimale Mehrfach-Bedingungsüberdeckung**

Pfadüberdeckung

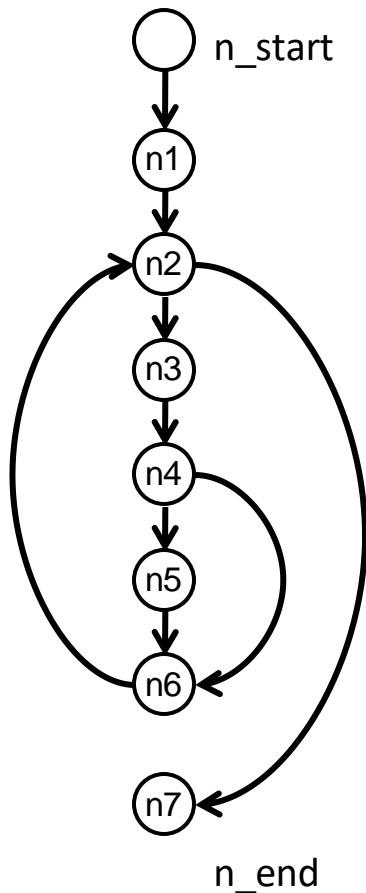
Misst die Überdeckung aller möglichen **Ausführungspfade**

$$\text{Pfadüberdeckung} = \frac{\text{Anzahl überdeckter Pfade}}{\text{Anzahl möglicher Pfade}}$$

Vollständige Pfadüberdeckung wäre **ideal**, ist aber meist nicht möglich

- Jeder Durchlauf einer **Schleife** bildet einen neuen Pfad
- **Anzahl möglicher Pfade** korreliert mit **Anzahl möglicher Schleifendurchläufe**
- Die **Anzahl möglicher Schleifendurchläufe** zu bestimmen hieße, die **Terminierung** zu beweisen
- Dies ist im Allgemeinen nur **semi-entscheidbar** [vgl. VL07 bzw. Halteproblem]

Vollständige Pfadüberdeckung Beispiel



Unsere countVowels-Funktion hat **eine Schleife**

```
while (cnt < s.length()) { ... }
```

Beispielhafte Pfade:

"" = n1,n2,n7

"b" = n1,n2,n3,n4,n6,n2,n7

"a" = n1,n2,n3,n4,n5,n6,n2,n7

"aa" = n1,n2,n3,n4,n5,n6,n2,n3,n4,n5,n6,n2,n7

"aba" = n1,n2,n3,n4,n5,n6,n2,n3,n4,n6,n2,n3,n4,n5,...

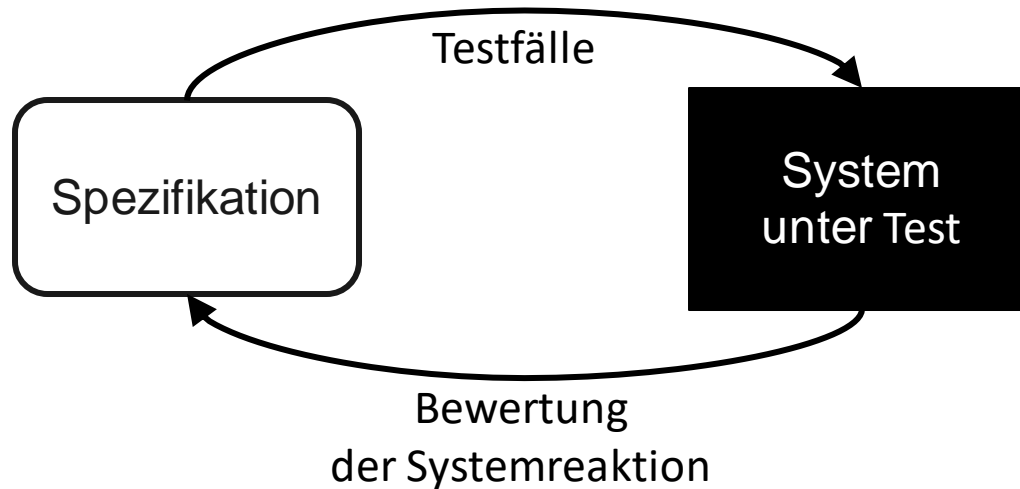
Die **Anzahl der Schleifendurchläufe** hängt mit der **Länge des Eingabestrings** zusammen

Inhalt

Testen

- Einführung
- Strukturorientierte Tests (White-Box Tests)
- Funktionsorientierte Tests (Black-Box Tests)
- Ausblick: Wie wird getestet?

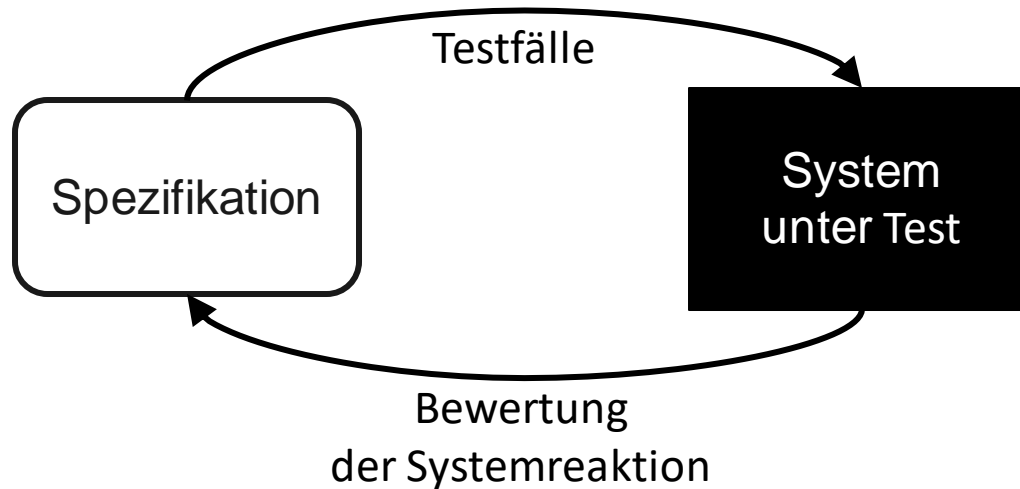
Funktionsorientierter Test



Auch **Black-Box**-Tests genannt

- Interner Aufbau muss nicht vorliegen
- Testfälle können schon **vor der Implementierung** aufgestellt werden
- Testfälle ergeben sich aus der **Spezifikation** der Software

Funktionsorientierter Test



Methoden zur Auswahl von Testfällen

- Äquivalenzklassenbildung
 - Auswahl **“repräsentativer Daten”**
 - z.B. durch Datenanalysen, Timing-Analysen, Pre/Post Analysen, Pfadbedingungen, ...
- Grenzwertanalyse
 - **Wertebereiche**, min/max Werte
- Entscheidungstabellen und Klassifikationsbäume

Äquivalenzklassentest

Ein- und Ausgabewerte werden in **Äquivalenzklassen** unterteilt

- Wertebereiche, für die **gleiches Verhalten** erwartet wird (laut Spezifikation)
- Definition der Wertebereiche hängt von Typ und Anwendungsfall ab (z.B. sinnvoll für Zahlentypen: Intervalle)

Für jeden Wert ergeben sich dabei **gültige** und **ungültige** Klassen

- Werte innerhalb und außerhalb des Wertebereichs für „Normalverhalten“
- Ergeben sich aus Fallunterscheidungen in der Spezifikation
- Eingaben, die unterschiedlich verarbeitet werden
- Ggf. mehrere ungültige Klassen, wenn verschiedene Prüfungen nötig sind

Testauswahl:

- aus jeder Äquivalenzklasse müssen beliebige **Repräsentanten** enthalten sein
- Bei mehreren Eingabewerten **Fehlerfälle unabhängig testen**: Pro Testfall höchstens ein Wert in einer ungültigen Äquivalenzklasse

Äquivalenzklassentest

Beispiel

Die Funktion `sale(int Preis, int Rabatt)` erhält als Eingabewerte einen **Preis in €** und einen abzuziehenden **Rabatt in Prozent**. Dieser wird nur auf Preise **ab 500 Euro** angewendet.

Äquivalenzklassen (rot=ungültig):

Preis: $] -\infty, 0[$, $[0, 500[$, $[500, \infty[$

Rabatt: $] -\infty, 0[$, $[0, 100]$, $] 100, \infty[$

Mögliche Testfälle:

Preis: `sale(-10, 5)`, `sale(50, 5)`, `sale(510, 5)`

Rabatt: `sale(50, -5)`, `sale(50, 5)`, `sale(50, 105)`

➤ Als Ergebnisse der Testfälle mit ungültigen Eingaben werden Fehler erwartet, die anderen prüfen den resultierenden Preis.

Grenzwertanalyse

In **Verzweigungen** und **Schleifen** gibt es oft Grenzwerte, für die die Bedingung gerade noch zutrifft (oder gerade nicht mehr)

➤ Gezielte Betrachtung der Grenzen von Wertebereichen sinnvoll

Auch hierbei werden Äquivalenzklassen gebildet, aber statt beliebigen Repräsentanten werden exakt die nächsten Werte von beiden Seiten der Grenze gewählt



Beispiel sale:

Preis: (-1, 5), (0, 5), (499, 5), (500, 5)

Rabatt: (50, -1), (50, 0), (50, 100), (50, 101)

Entscheidungstabellentest

Betrachtung aller möglichen Kombinationen von Eingaben (bzw. deren Äquivalenzklassen)

		T1	T2	T3	T4	T5	T6		
Bedingungen	Preis ≥ 0 und Preis < 500 ?	N	N	N	N	J	J	J	N
	Preis ≥ 500 ?	N	N	J	J	N	N	J	J
	Rabatt ≥ 0 und Rabatt ≤ 100 (gültig)	N	J	N	J	N	J	N	J
Aktion	Fehlermeldung: Preis oder Rabatt ungültig	J	J	J	N	J	N	-	-
	Ergebnis = Preis	N	N	N	N	N	J	-	-
	Ergebnis = Preis * (1-Rabatt/100)	N	N	N	J	N	N	-	-

Ergebnis: Ein Testfall pro Spalte

Entscheidungstabellentest

Problem: exponentielles Wachstum der Anzahl an Spalten

Lösung: optimierte Entscheidungstabelle

		T1	T3	T4	T6
Bedingungen	Preis ≥ 0 und Preis < 500 ?	N	-	N	J
	Preis ≥ 500 ?	N	-	J	N
	Rabatt ≥ 0 und Rabatt ≤ 100 (gültig)	-	N	J	J
Aktion	Fehlermeldung	J	J	N	N
	Ergebnis = Preis	N	N	N	J
	Ergebnis = Preis * (1-Rabatt/100)	N	N	J	N

- T1 und T2 bzw. T3 und T5 können zusammengefasst werden (es reicht ein ungültiger Eingang für den Fehler)

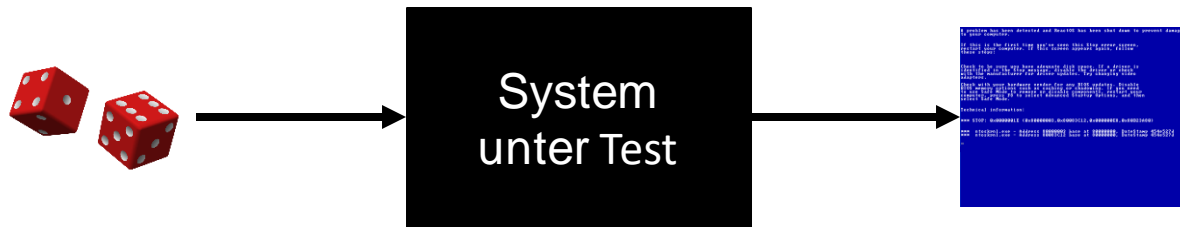
Randomisiertes Testen

In einigen Szenarios ist Testen mit der **Spezifikation nicht ausreichend**

- Es ist **keine Spezifikation** vorhanden
- Der vorhandenen Spezifikation wird **nicht vertraut**
- Die vorhandene Spezifikation ist unter Umständen **unvollständig**
- Die vorhandene Spezifikation ist **zu Abstrakt** [What You See Is Not What You eXecute]

In diesem Fall können **zufällige Eingaben** getestet werden

- Interessante Ausgabe ist meist Fehlverhalten (negative testing), wie **Exception** oder **Crash**



Fuzzing

Fuzzing ist **randomisiertes negativ Testen**, also zufällige Werte, die ein Fehlverhalten aufdecken sollen

- Negatives Testen ist besonders im **Security**-Bereich interessant
- Nicht aufgedecktes Fehlverhalten kann zu **enormen Schäden** führen

Verschiedene **Angriffsvektoren** können abgedeckt werden

- Graphische Oberflächen (GUIs)
- Kommandozeile (CLI)
- Programmierschnittstellen (APIs)
- Dateien
- Netzwerkschnittstellen
- Physischer Hardware Zugriff

Fuzzing

Je mehr über das System bekannt ist, desto **gezielter** kann man testen

- Information aus APIs, Protokollen, Formaten oder durch Reverse Engineering

Verwenden von Typinformation

- **Integers:** Neben 0 und MAX_VALUE, vor allem Werte um Vielfache von 2
- **Strings:** Überlange Strings können zu Buffer Overflows führen, Alphanumerische Werte können Trenn-/Endzeichen treffen

Verwenden von Protokollinformation

- **Name-Value Pairs:** Randomisieren auf die Values beschränken
- **Block Identifier:** Spezielle Werte steuern den Parsing-Prozess
- **Header Values:** Vorangestellte Metainformation über die Datei

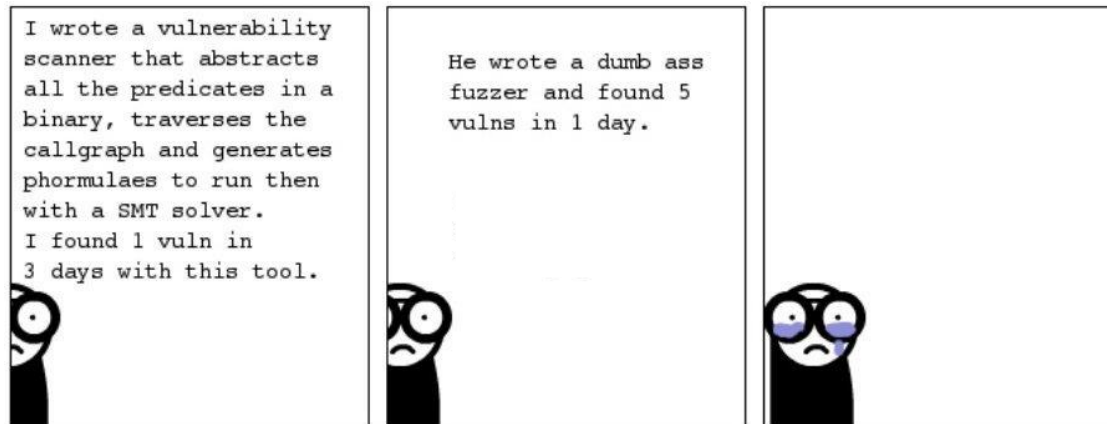
Grenzen des Fuzzing

Ein großes Problem beim Fuzzzen ist **mangelnde semantische Einsicht**

- Gefundene Fehler können nur schwer zum **Ursprung** zurückverfolgt werden

Komplizierte Fehler (Deep Bugs) werden nicht unbedingt erreicht

- Je spezieller die Umstände, desto unwahrscheinlicher werden sie getroffen
- Durch Code-Analyse (z.B. Taint-Analyse) können schwierige Fälle erreicht werden



Inhalt

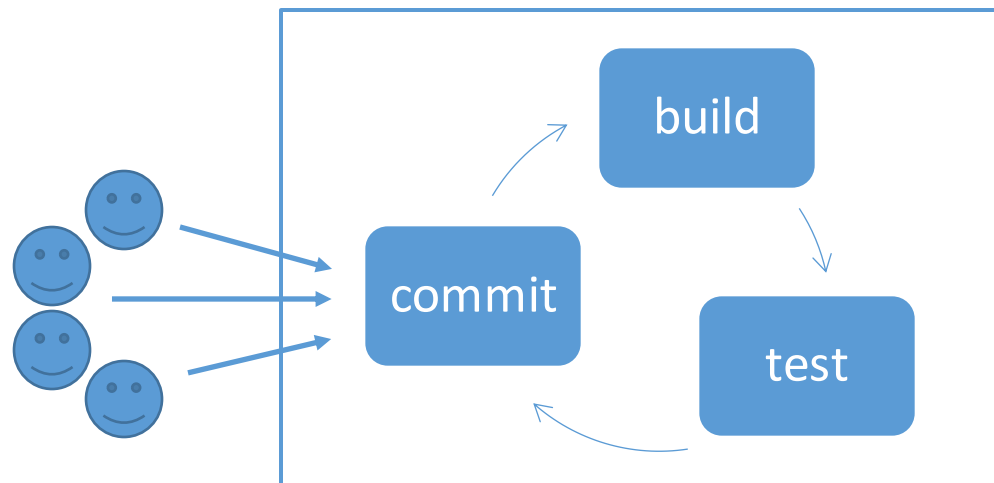
Testen

- Einführung
- Strukturorientierte Tests (White-Box Tests)
- Funktionsorientierte Tests (Black-Box Tests)
- **Ausblick: Wie wird getestet?**

Wie wird getestet? Problem: Integration!

Inkrementelle Entwicklung mit kleinen Änderungen ermöglicht eine kontinuierliche Integration (Continuous Integration)

Jede Änderung im geteilten Repository wird automatisiert konstruiert und getestet.



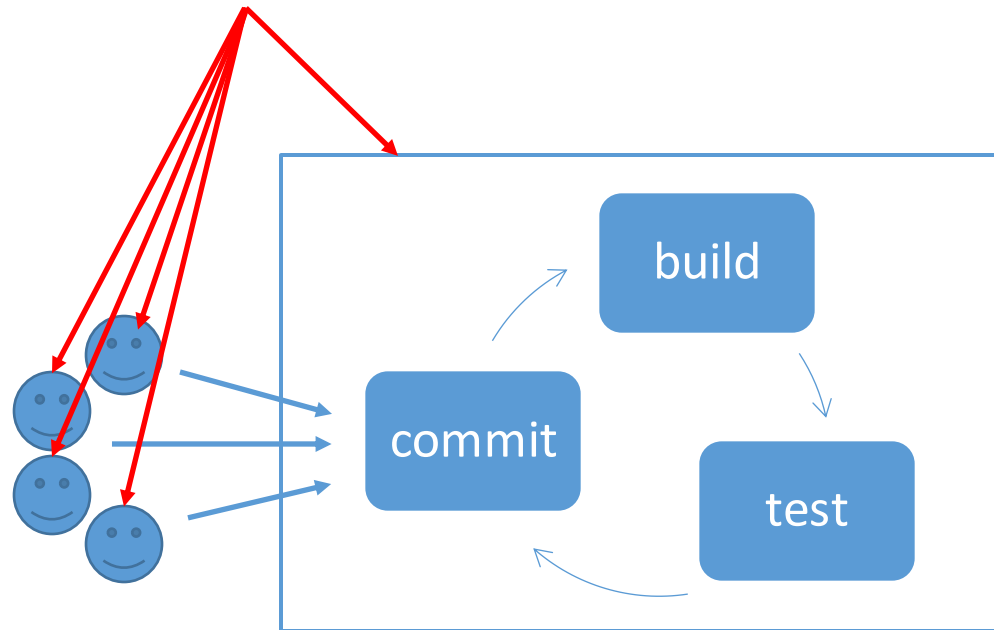
Exkurs: Tools

Entwickler, Entwicklungsserver und Produktionsserver benötigen die gleiche Softwareumgebung.

→ *Virtualisierung und Containerisierung*

Beispieltools

- *Docker*
- *Windocks*
- ...



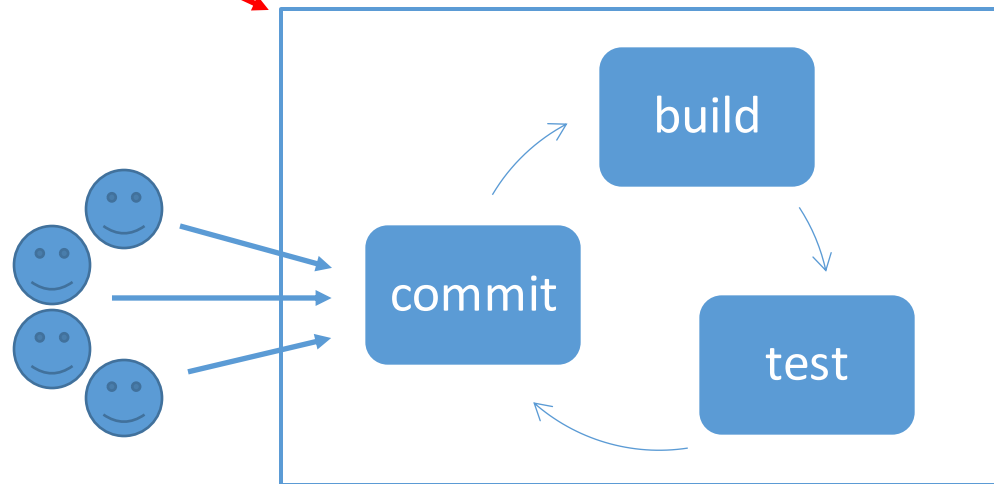
Exkurs: Tools

Verschiedene Prozesse der kontinuierlichen Integration werden in geteilter Infrastruktur ausgeführt werden.

→ *Continuous Integration Server*

Beispieltools

- *Jenkins*
- *Travis CI*
- ...



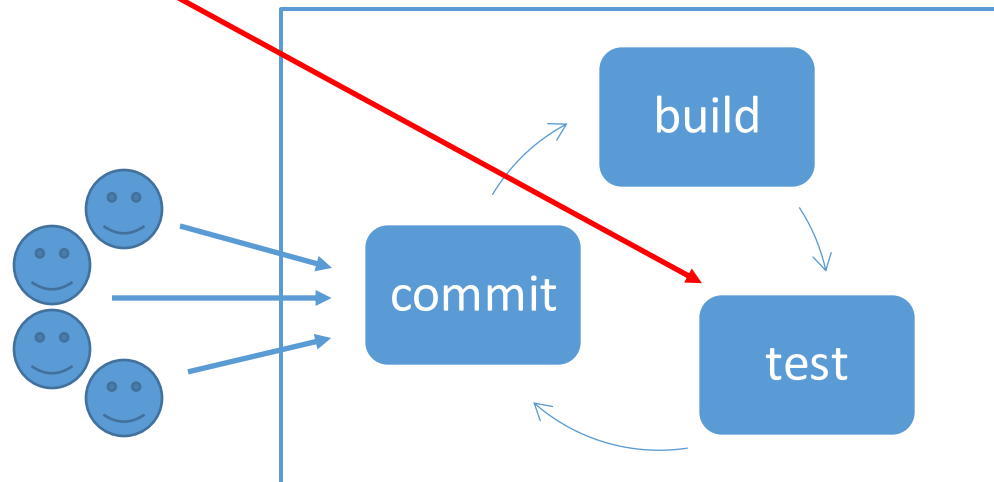
Exkurs: Tools

Der Code wird automatisch anhand von vorher definierten Testfällen überprüft.

→ *Test Framework*

Beispieltools

- *JUnit*
- *Cypress*
- ...



Wie wird getestet? - Testautomatisierung

Für wiederkehrende Tests (Regressionstest) ist eine Automatisierung wünschenswert.



Die meisten Testfälle können durch Skripte oder Test-Frameworks **automatisch durchgeführt** und **ausgewertet** werden

- Je öfter getestet wird, desto mehr lohnt sich Automatisierung
- Dokumentation der Tests und Reports werden vereinfacht
- Viele Frameworks für unterschiedliche Sprachen und Schwerpunkte

JUnit

Open source Framework in Java für Java (www.junit.org)

- JUnit verwendet Annotationen im Source Code (@....)

Funktion als TestCase deklarieren: mit @Test

```
@Test
public void myTestCase() {
    // ....
}
```

Klassen mit TestCases als Test-Suite zusammenfassen: mit @SuiteClasses

```
@RunWith(Suite.class)
@SuiteClasses({ MyTestClass1.class, MyTestClass2.class })
public class AllTests {}
```

JUnit

Best Practice: Für jede zu testende Klasse eine Test-Klasse mit Test-Cases erstellen. Den Gesamt-Test dann zu einer Testsuite zusammenfassen

Weitere wichtige Annotationen:




- *@Before* – vor jedem Testfall auszuführen, z.B. Testdaten vorbereiten
- *@After* – wird einmal nach jedem Testfall der Klasse ausgeführt, z.B. Testdaten aufräumen
- *@BeforeClass*, *@AfterClass* – einmal vor bzw. nach dem Ausführen der Tests einer Klasse ausführen, z.B. Testsystem aufbauen, Verbindungen beenden, etc.

JUnit wird in einer gesonderten Umgebung ausgeführt

➤ in Eclipse: Run As -> JUnit Test

JUnit - Überprüfung erwarteter Ergebnisse

JUnit Tests haben drei mögliche Ergebnisse:

-  Testfall war erfolgreich (ist durchgelaufen)
-  Testfall ist fehlgeschlagen (Fehlverhalten identifiziert)
-  Fehler bei der Test-Ausführung (Exception geworfen/ Test fehlerhaft)

Spezielle jUnit Assertions können verwendet werden, um Ergebnisse zu prüfen

```
@Test
public void test0() {
    assertTrue(countVowels("")==0);
}
@Test
public void test5() {
    assertEquals(2, countVowels("abcde"));
}
@Test
public void test10000() {
    fail("TODO");
}
```

Testfall erfolgreich
wenn Ergebnis true

Testfall erfolgreich
wenn Vergleich true

Testfall schlägt
immer fehl

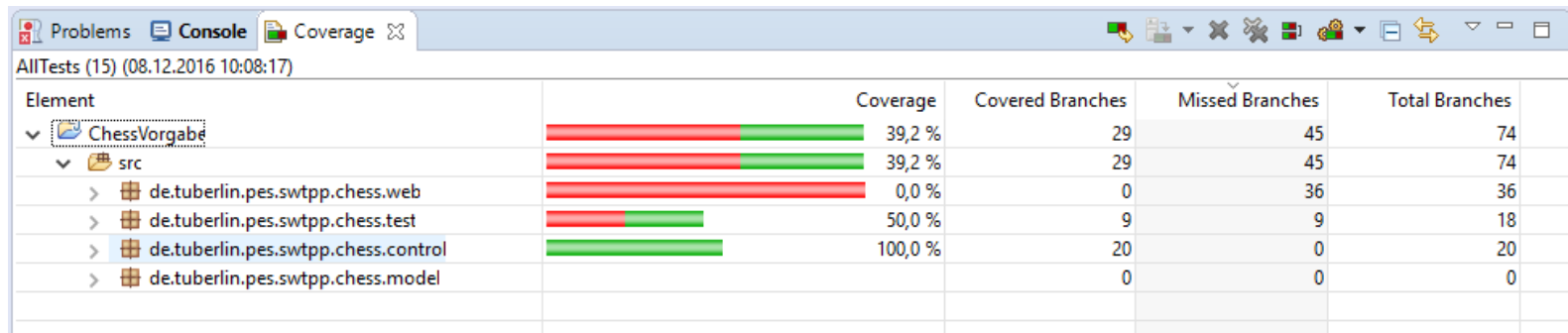
Testabdeckung messen

Eclipse-Plugin *Emma* kann die Testabdeckung für Java/JUnit messen







- Zeigt Anweisungs- und Zweigüberdeckung (und weitere) für eine Ausführung an

Achtung: Emma verwendet Java Byte-Code

- Kontrollflussgraph kann durch Optimierung abweichen
- Zweigüberdeckung beinhaltet Bedingungsüberdeckung



The screenshot shows the Eclipse IDE's Coverage view. The title bar indicates 'AllTests (15) (08.12.2016 10:08:17)'. The table displays coverage data for the 'ChessVorgabe' project and its sub-packages. Each row includes a visual progress bar (red for missed, green for covered), the coverage percentage, the number of covered and missed branches, and the total number of branches.

Element	Coverage	Covered Branches	Missed Branches	Total Branches
ChessVorgabe	 39,2 %	29	45	74
src	 39,2 %	29	45	74
de.tuberlin.pes.swtpp.chess.web	 0,0 %	0	36	36
de.tuberlin.pes.swtpp.chess.test	 50,0 %	9	9	18
de.tuberlin.pes.swtpp.chess.control	 100,0 %	20	0	20
de.tuberlin.pes.swtpp.chess.model	 0,0 %	0	0	0

Lernziele

- ☐ Welche Maßnahmen zur Qualitätssicherung gibt es?
- ☐ Warum müssen wir testen?
- ☐ Warum kann man nicht immer alles testen?
- ☐ In welchen Phasen der Entwicklung wird getestet?
- ☐ Welche Methoden gibt es grundsätzlich, Testfälle systematisch auszuwählen?
- ☐ Welche Überdeckungsmaße haben wir für strukturorientierte Tests kennengelernt?
- ☐ Erreicht man mit 100% Zweigüberdeckung auch 100% Anweisungsüberdeckung?
- ☐ Welche sprachlichen Elemente verhindern, mit 100% Pfadüberdeckung zu testen?
- ☐ Erreicht man mit 100% einfacher Bedingungsüberdeckung auch 100% Zweigüberdeckung?
- ☐ Warum ist Mehrfach-Bedingungsüberdeckung schwer zu erreichen?
- ☐ Kann man mit strukturorientierten Test erkennen, wenn eine Funktionalität vergessen wurde?
- ☐ Warum heißen die Funktionsorientierten Tests auch Black-Box-Tests?
- ☐ Welche Informationen werden zur Definition der Testfälle im funktionsorientierten Tests hergenommen?
- ☐ Wodurch erweitert die Grenzwertanalyse die Äquivalenzklassentests?
- ☐ Was stellen die Zeilen in der Entscheidungstabelle dar?
- ☐ Was stellen die Spalten in der Entscheidungstabelle dar?
- ☐ Wozu dient Testautomatisierung?
- ☐ Wie werden Testfälle in JUnit definiert?

Quellen

[1] <http://www.istqb.org/>, International Software Testing Qualifications Board

[2] Peter Liggesmeyer, ***Software-Qualität***
Testen, Analysieren und Verifizieren von Software
Spektrum, 2002

[3] Glenfield J. Myers ***The Art of Software Testing***
Wiley, 1979.

[4] IAN SOMMERVILLE, *Software-Engineering*, Pearson, 2012, 9. Aufl.

[5] A. Spillner und T. Linz, Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester ; Foundation Level nach ISTQB-Standard, dpunkt-Verlag, 2012, 5. Aufl.