

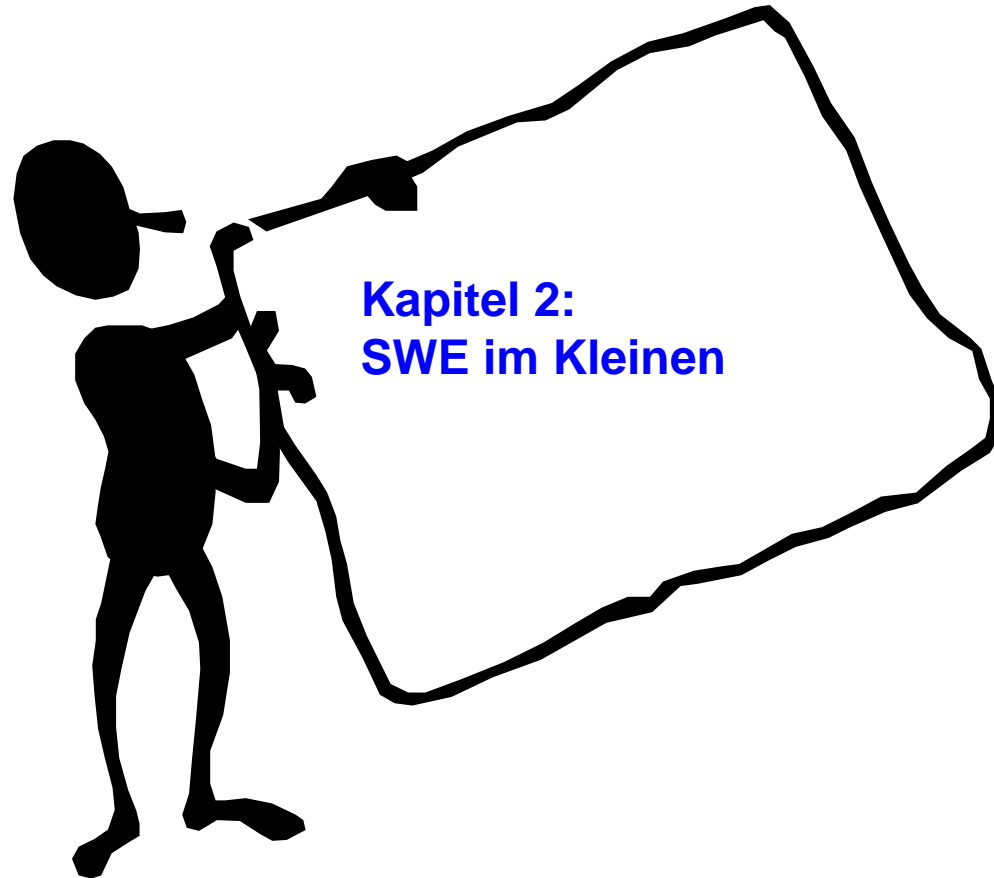
Einführung in Software Engineering

Barbara Paech, Marcus Seiler

Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
<http://se.ifi.uni-heidelberg.de>
paech@informatik.uni-heidelberg.de



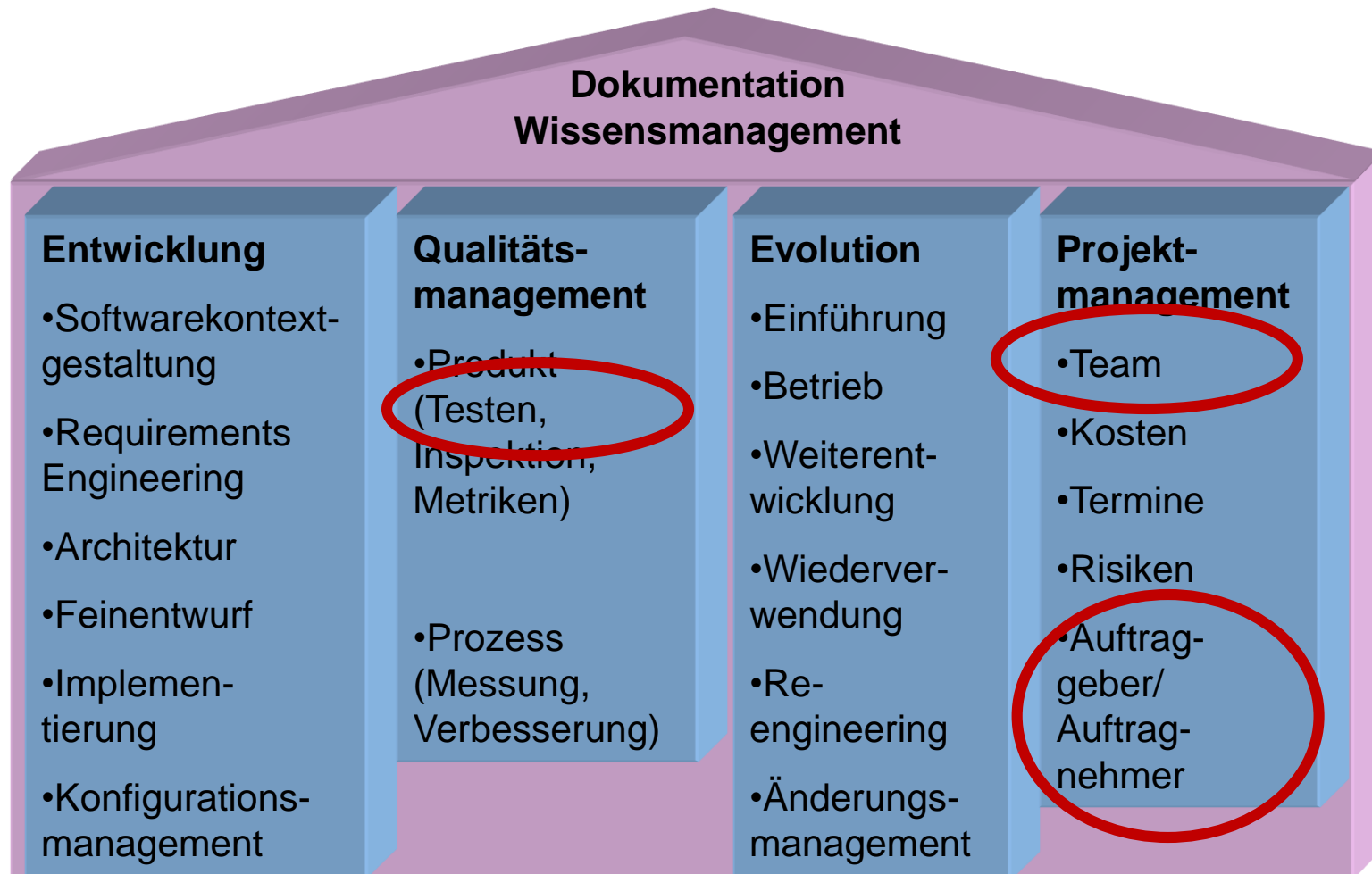
RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

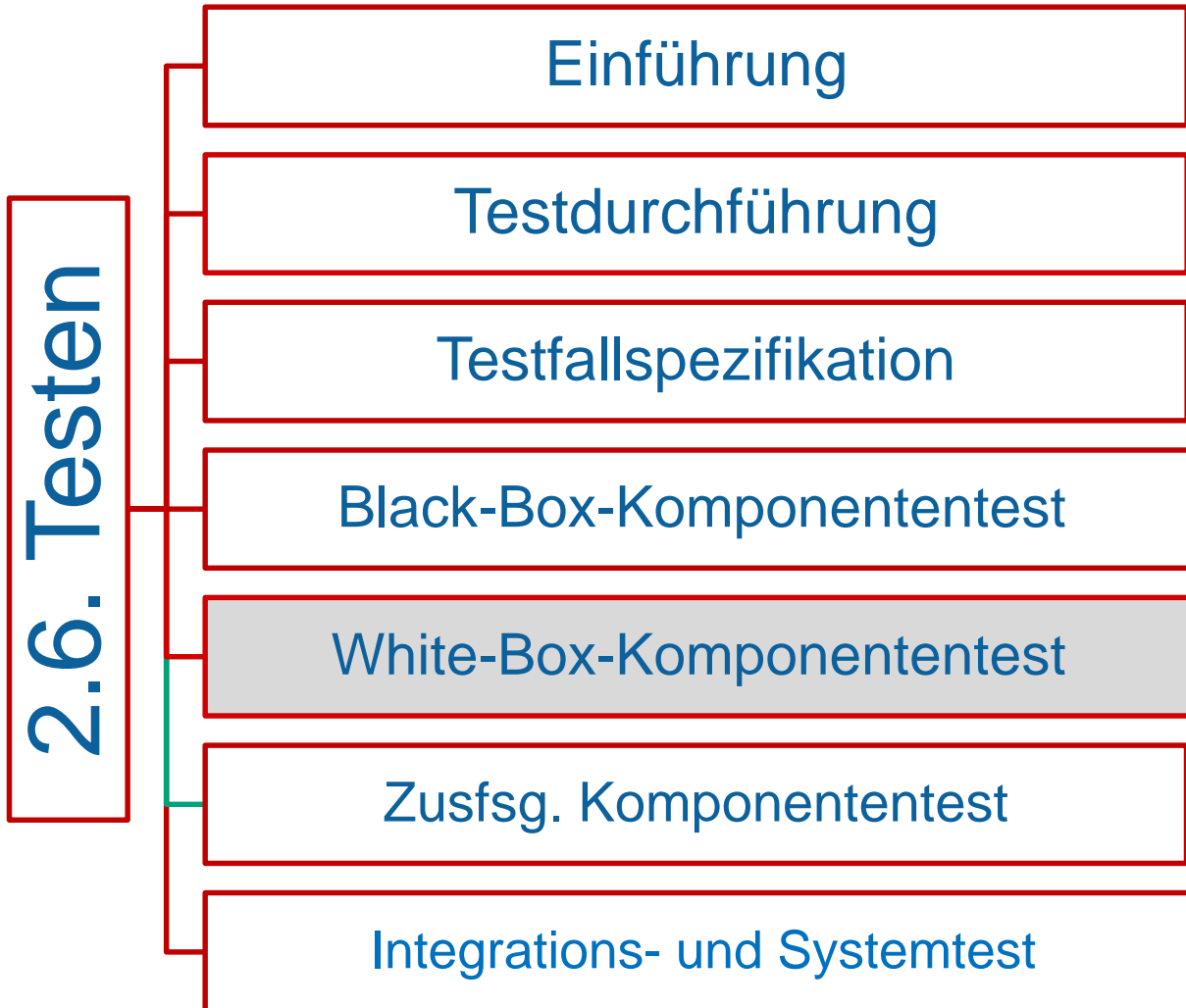


2. SWE im Kleinen (3.Teil)

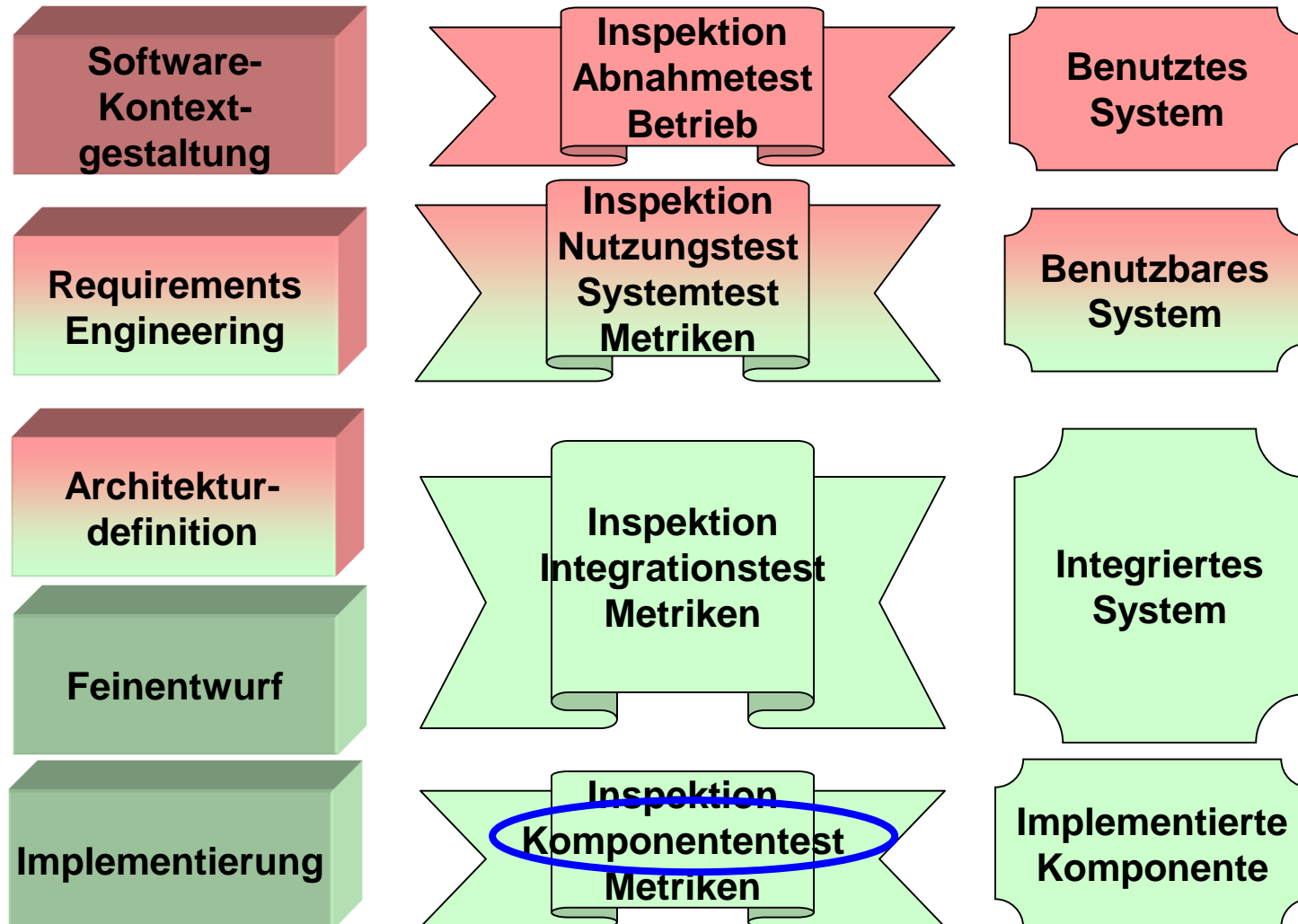
- 2.1. Vorgehen
- 2.2. Einführung Qualitätssicherung
- 2.3. Organisatorische Qualitätssicherung
- 2.4. Statische Codeprüfung
 - Einführung
 - Metriken
 - Inspektion
- 2.5. Umgang mit Fehlern
- 2.6. Testen (Dynamische Codeprüfung) Fortsetzung
- 2.7. Kommunikation im Projekt

Aufgabenbereiche des Software Engineering





Aktivitäten und Ergebnisse der Entwicklung und Qualitätssicherung



- **White-Box-Komponententest**
 - Kontrollflussgraph
 - Anweisungs-, Zweig-, Pfad-, Termüberdeckung

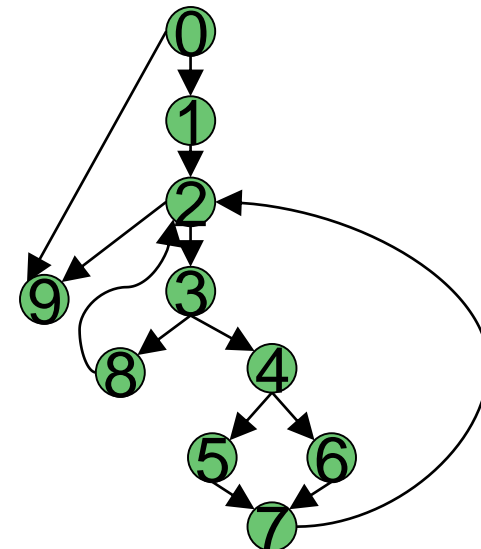
- Siehe auch Kapitel 19.5.1, 19.5.2. und 19.6. bei Ludwig/Lichter (Hausaufgabe 3.5.)
- Nachfolgend ein paar Folien dazu
 - Hier teilweise auch Synonyme zu Begriffen genannt, z.B. **White-Box** statt **Glass-Box**, **Kontrollflussgraph** statt **Ablaufgraph**
 - Zusätzlich: wichtige Unterscheidung **logische** und **konkrete** Testfälle

- Kontrollflussgraph
- Überdeckungen/Ziele bei Black-Box
 - Funktionsüberdeckung, Eingabeüberdeckung, Ausgabeüberdeckung, Leistungsgrenzen ausloten, Mengengrenzen ausschöpfen, Fehlersituationen herbeiführen
- Überdeckungen bei White-box/Glass-box
 - Anweisungsüberdeckung, Zweigüberdeckung, Termüberdeckung, Pfadüberdeckung
- Techniken zur Testfallauswahl beim Black-Box-Test
 - Äquivalenzklassen Test, Zustandsbasierter Test, Anwendungsfallbasierter Test

- Vor allem geeignet für Komponententest
- Leitet Testfälle aus der **Kenntnis des Codes** ab
- Prüft, ob die Codeteile (Anweisungen etc) korrekt funktionieren
- Kann **übersehene Anforderungen** NICHT aufdecken
- Für die Erstellung der Testfälle ist **zusätzlich die Anforderungsspezifikation** nötig, um die Eingabedaten und das SOLL-Ergebnis angeben zu können

- Systematik zur Erstellung der Testfälle ist Grad der Abdeckung des Codes
- Anweisungsüberdeckung
- Zweigüberdeckung (auch mit Termüberdeckung)
- Pfadüberdeckung
- Kann am **Kontrollflussgraph** abgelesen werden
- Defensiver Code (Fehler aus anderen Programmteilen abfangen) und Code für ungewöhnliche Fälle ist oft „schwer erreichbar“

- Ein Kontrollflussgraph ist eine Abstraktion des Codes, so dass
 - Alle Verzweigungen sichtbar werden
 - Aufeinanderfolgende Anweisungen zusammengefasst werden
 - Für Rücksprünge eigene Knoten eingeführt werden



Kontrollflussgraph Binäre Suche

//Annahme : a ist sortiert

```

1.  int binsearch (int[] a, int k) {
2.      result.found = false;
3.      result.index = -1 ;
4.      if (a.length <> 0) {
5.          int bottom = 0;
6.          int top = a.length - 1;
7.          int mid;
8.          while ( bottom <= top ) and not result.found {
9.              mid = (top + bottom) / 2 ;
10.             if (a[mid] == k) {
11.                 result.index = mid;
12.                 result.found = true;
13.             } // if part
14.             else {
15.                 if (a [mid] < k)
16.                     bottom = mid + 1;
17.                 else
18.                     top = mid - 1;
19.             } //else part
20.         } //end while
21.     } //end if
22.     return result.index
23. } //binsearch

```

0

1

2

3

8

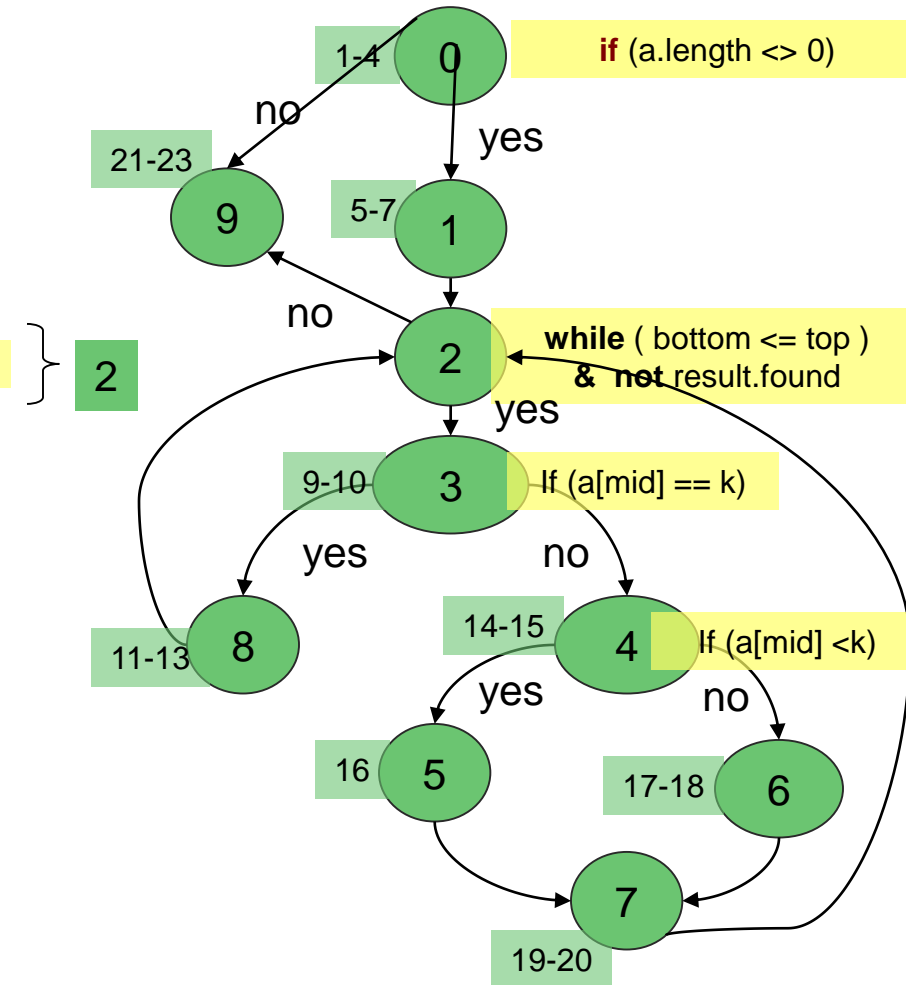
4

5

6

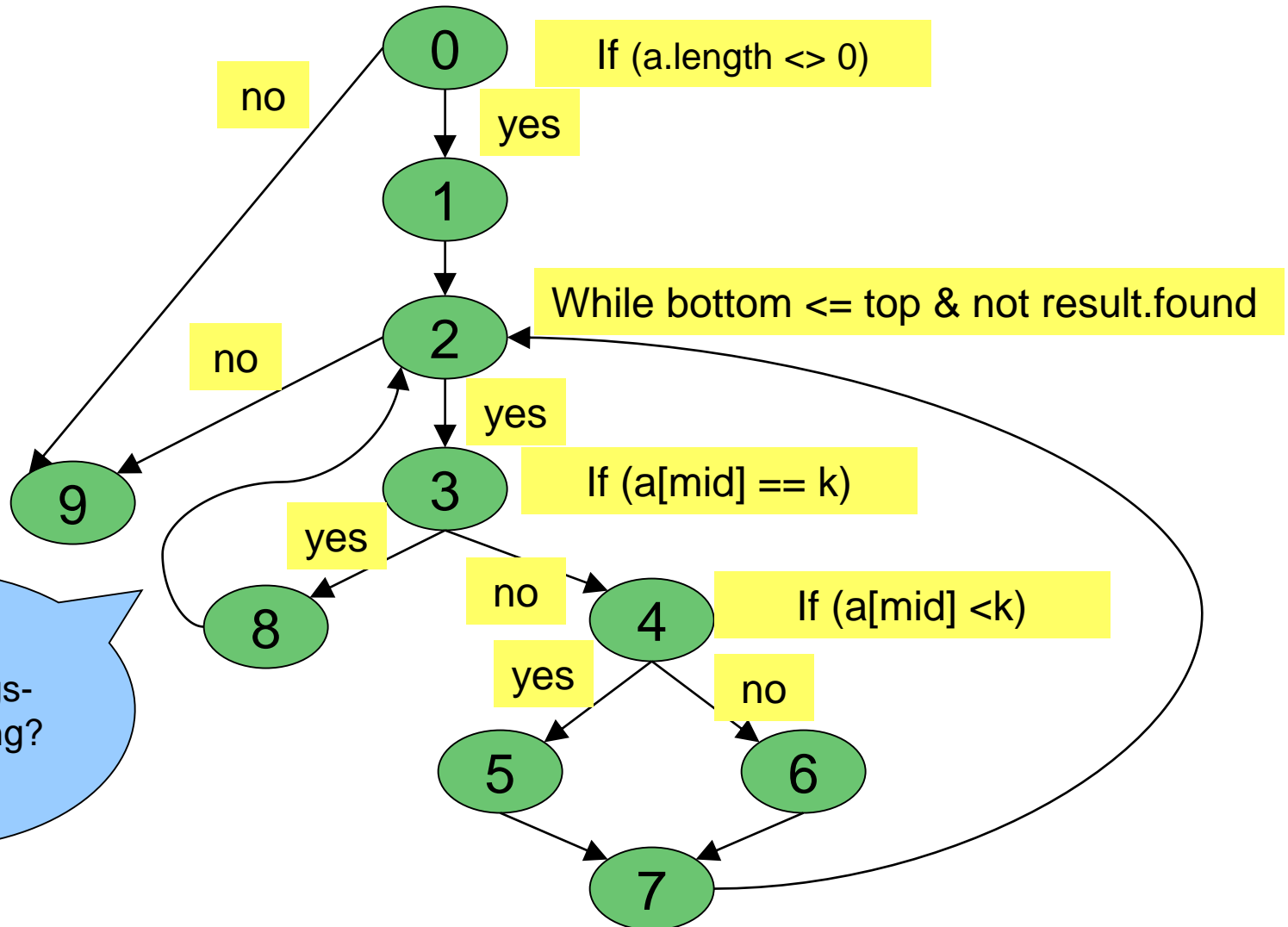
7

9



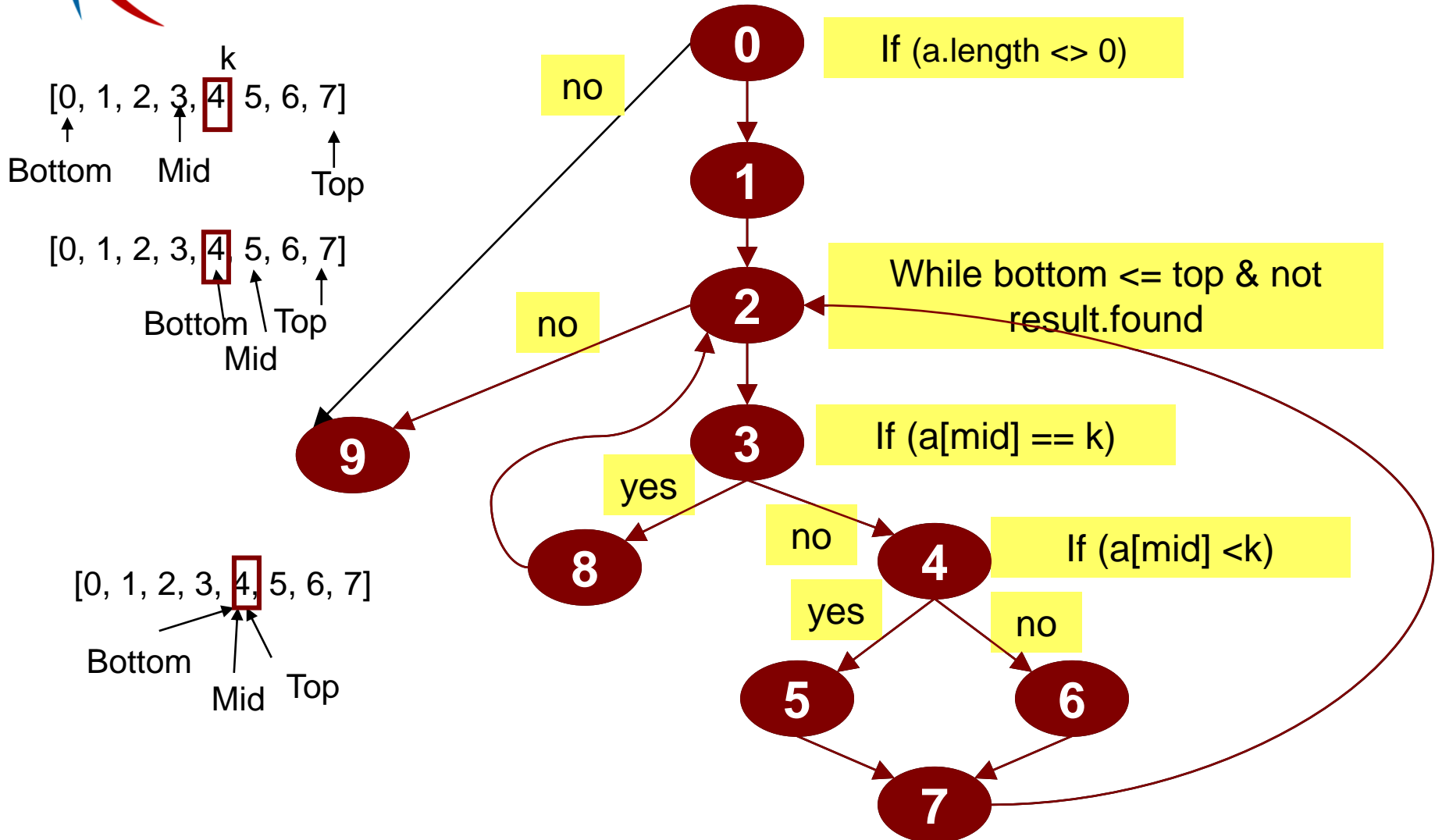
- In dem Beispiel Binäre Suche wurden die folgende Regeln verwendet:
 - Aufeinanderfolgende Zuweisungen werden zusammengefasst, bis zum Ende eines Blockes, insbesondere nächste Verzweigung oder Schleife. Die Abfrage der Verzweigung kann auch noch am Ende dazu genommen werden (Beispiel: 0,1,3,5,8,9)
 - Der Knoten für den Nein-Fall einer Verzweigung enthält das else und wie oben beschrieben aufeinanderfolgende Zuweisungen (Beispiel: 4,6)
 - Für die Bedingung einer Schleife ist ein eigener Knoten nötig (wg. Rücksprung) (Beispiel: 2)
 - Der Schluss einer echten Verzweigung (if-then-else) muss ein eigener Knoten sein (zur Vereinigung der Zweige) (Beispiel: 7)
 - Das Programmende muss ein eigener Knoten sein (Beispiel: 9)

Abdeckung Kontrollflussgraph



- **Knoten** im Kontrollflussgraph repräsentieren (Sequenzen von) Anweisungen
- 100% -Überdeckung heißt: alle Knoten müssen überdeckt werden
- Beispiel Binäre Suche:
 - 100%-Überdeckung mit Reihenfolge 0,1,2,3,4,5,7,2,3,4,6,7,2,3,8,2,9 (auch als 2 getrennte Sequenzen möglich)
 - Welche Testeingaben nötig, um diese Sequenz zu erreichen?
 - Entspricht dem Black-Box-Testfall k in a , k „in der Mitte“
 - Erster Schleifendurchlauf entspricht k in a als erstes Element
 - Letzter Schleifendurchlauf entspricht $a.length = 1$, k in a
- Anweisungsüberdeckung heißt NICHT, dass alle Äquivalenzklassen abgedeckt sind
 - Für $a.length > 0$, k not in a wären weitere Testfälle nötig

Binäre Suche Anweisungsüberdeckung



- **Kanten** im Kontrollflussgraph repräsentieren Zweige
- 100%-Zweigüberdeckung sollte erreicht werden
- Stellt sicher, dass jeder **Ausgang einer Verzweigung** durchlaufen wird
- Beispiel Binäre Suche:
 - Anweisungsüberdeckung 0,1,2,3,4,5,7,2,3,4,6,7,2,3,8,2,9
führt Kante 0-9 nicht aus
 - Hinzunahme von 0,9 als weiteren Testfall nötig
 - Entspricht Black-Box-Äquivalenzklasse $a = []$

- Weitere Möglichkeit: Für **komplexe Bedingungen** sollte mindestens jede Teilbedingung einmal überdeckt werden, besser alle Kombinationen
- `if (A or B)` wird `wahr` bzw `falsch`, wenn man die Belegung von `A` ändert (also `A wahr` oder `A falsch`) und `B falsch` ist. Der Fall `B wahr` muss also für Zweigabdeckung nicht ausgeführt werden.
- **Termüberdeckung** fordert, dass für alle Teilterme sowohl `wahr` als auch `falsch` getestet wird, wenn das für die gesamte Bedingung einen Unterschied machen kann (ansonsten ist der Teilterm irrelevant und kann weggelassen werden).
 - Oben ist `B wahr` für den Fall `A falsch` entscheidend.

- **Pfade** im Kontrollflussgraph entstehen durch Verbindung von Zweigen
- 100%-Pfadüberdeckung nicht realistisch
- Beispiel Binäre Suche:
 - Anweisungsüberdeckung 0,1,2,3,4,5,7,2,3,4,6,7,2,3,8,2,9 und Zweigüberdeckung 0,9, führt z.B. nicht aus
 - **Pfad 0,1,2,9**
 - **Pfad mit mehr als 3 Schleifendurchgängen**
- Pfadabdeckung nur mit vorgegebener Anzahl von Schleifenwiederholungen machbar, z.B.
 - 0-3 Durchläufe und typische sowie Maximalanzahl Durchläufe

■ **Nachteil**

- Es ist schwer, geeignete Eingabewerte zu finden, um einen spezifischen Ablauf zu erreichen
- Es ist schwer, dazu die geeignete Ausgabe zu finden

■ **Vorteil**

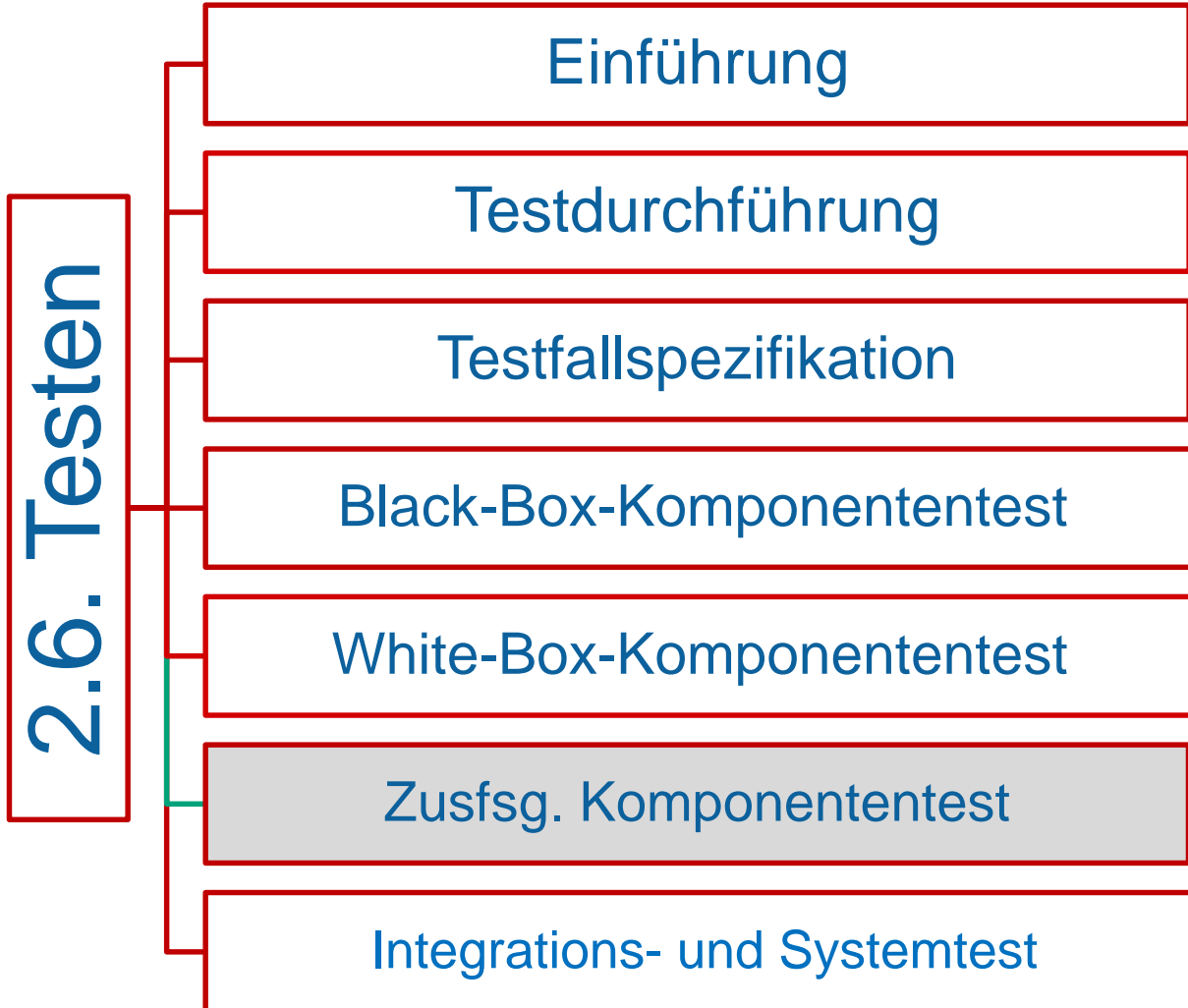
- Jeder Codeteil aktiviert => entdeckt unerreichbaren Code, grobe Fehler im Code

■ **Meistens GreyBox: Kombination Black+White**

- Definiere Testfälle für Blackbox
- Prüfe, welcher Code damit nicht erreicht bzw. welche Abläufe im Kontrollflussgraph nicht erreicht werden
- Versuche dafür zusätzliche Testfälle zu finden (ggf. ohne vorgegebene Ausgabe)

- Siehe Vorlesungsarbeitsblatt
 - Kontrollflussgraph
 - Anweisungsüberdeckung
 - Zweigüberdeckung

- Siehe auch Kapitel 19.8 bei Ludewig/Lichter



- Komponententest kann auch auf **größere Programmeinheiten**, z.B. Prozeduren oder Klassen angewendet werden
 - Abdeckung jeder Einheit
 - Abdeckung der Aufrufe für Teile der Einheiten
 - Abdeckung der Ausführungssequenzen der Einheiten (siehe auch zustandsbasierter Test)

- Klasse ist eine besondere Art von Komponente
- Einzelne Methoden durch GreyBox-Methode testen
- Wichtig ist die Festlegung einer Testreihenfolge für Methoden
 - Konstruktoren
 - Get-Methoden
 - Boolesche Methoden (is...)
 - Set-Methoden
 - Iteratoren
 - Komplexe Berechnung oder Ablaufsteuerung in einer Klasse
 - Sonstige Methoden
 - Destruktoren

- Weiterhin wichtig: Festlegung einer Testreihenfolge für das **Zusammenspiel** der Operationen
- Operationen sind abhängig voneinander, da sie auf gemeinsamen Daten (Attributen der Klasse) arbeiten
- Kann durch **zustandsbezogenen Test** abgeprüft werden
 - Verwendet **Kontrollzustandsdiagramm**, um Datenzustände deutlich zu machen, die das Zusammenspiel der Methoden beeinflussen.
 - Testfälle ergeben sich dann durch sequentiellen Aufruf der Methoden in der durch das Zustandsdiagramm vorgegebenen Reihenfolge

■ Black-Box-Verfahren:

- Testen die **Außenwirkung** des Testobjekts
- Keine Steuerung des Ablaufs des Testobjekts
- Nutzen Kenntnisse über die Schnittstelle, Spezifikation
- Beispiele: Äquivalenzklassen, Grenzwertbezogener Test, Zustandsbezogener Test

■ White-Box-Verfahren:

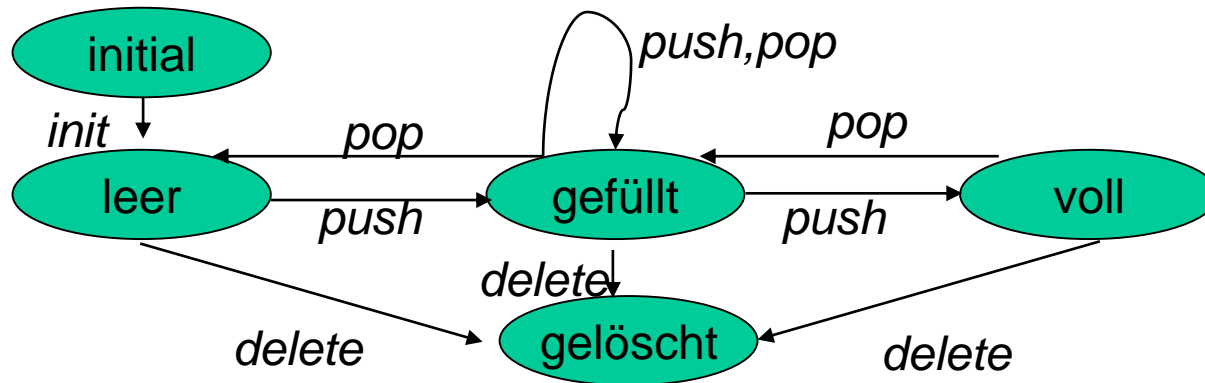
- Testen gezielt die **verschiedenen Bestandteile und Abläufe** des Testobjekts
- Nutzen Kenntnisse über den inneren Aufbau (Code)
- Beispiele: Überdeckung der Anweisungen, der Zweige, der Bedingungen, der Pfade

■ Intuitiver / erfahrungsbasierter Test:

- Beruht auf Kenntnis typischer Fehlerzustände
- Sollte immer **zusätzlich** zu systematischen Verfahren durchgeführt werden

Zustandsbezogener Test (1)

- Berücksichtigt neben Ein/Ausgaben auch die **Historie** (den erreichten Zustand)
- Zustände beschreiben die Vor/Nachbedingungen der Operationen / Ereignisse
- Typisches Beispiel: Stapel
- Zustände: initial, leer, gefüllt, voll, gelöscht
Ereignisse: init, pop, push, delete

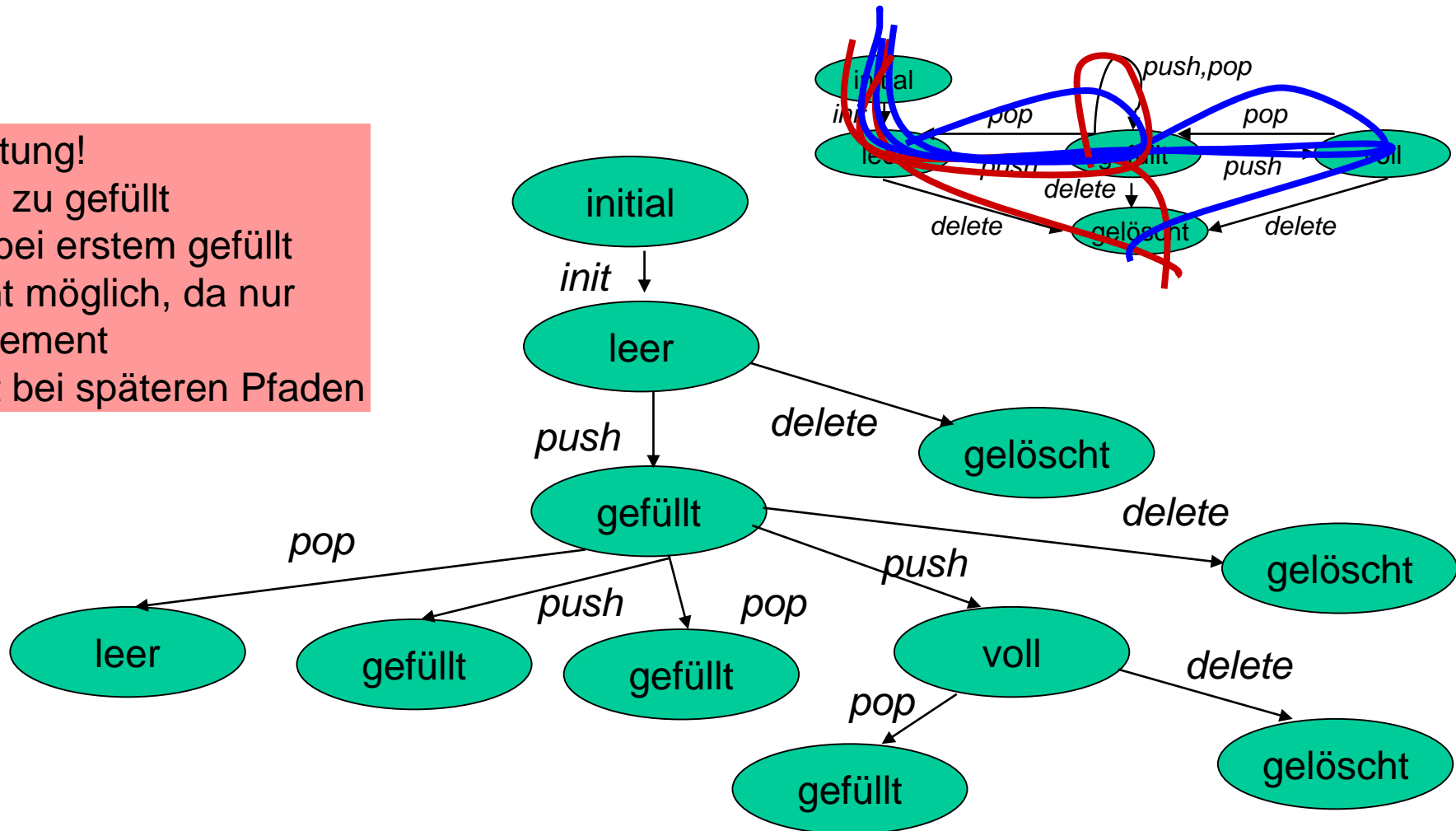


- **Systematik zur Testfallableitung:** alle Zustände und alle darin zulässigen Operationen (alle Zweige) einmal abgedeckt

- Testfallableitung mit **Übergangsbaum:**
 - Abrollen des Diagramms bis Blatt in jedem Pfad entweder
 - **terminierend** oder
 - **Wiederholung** eines Knotens auf dem Pfad

Zustandsbezogener Test (3)

Achtung!
Pop zu gefüllt
ist bei erstem gefüllt
nicht möglich, da nur
1 Element
Erst bei späteren Pfaden

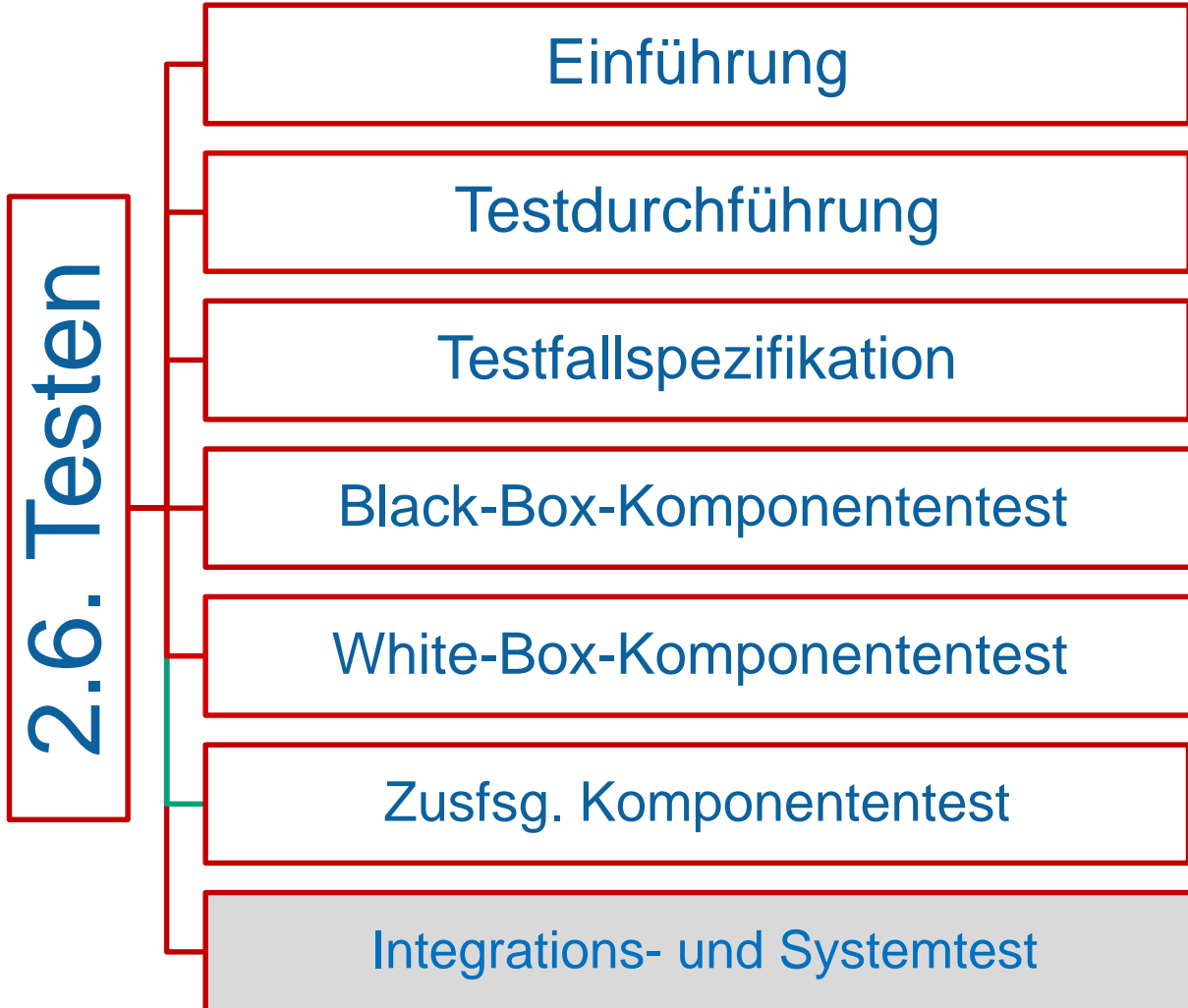


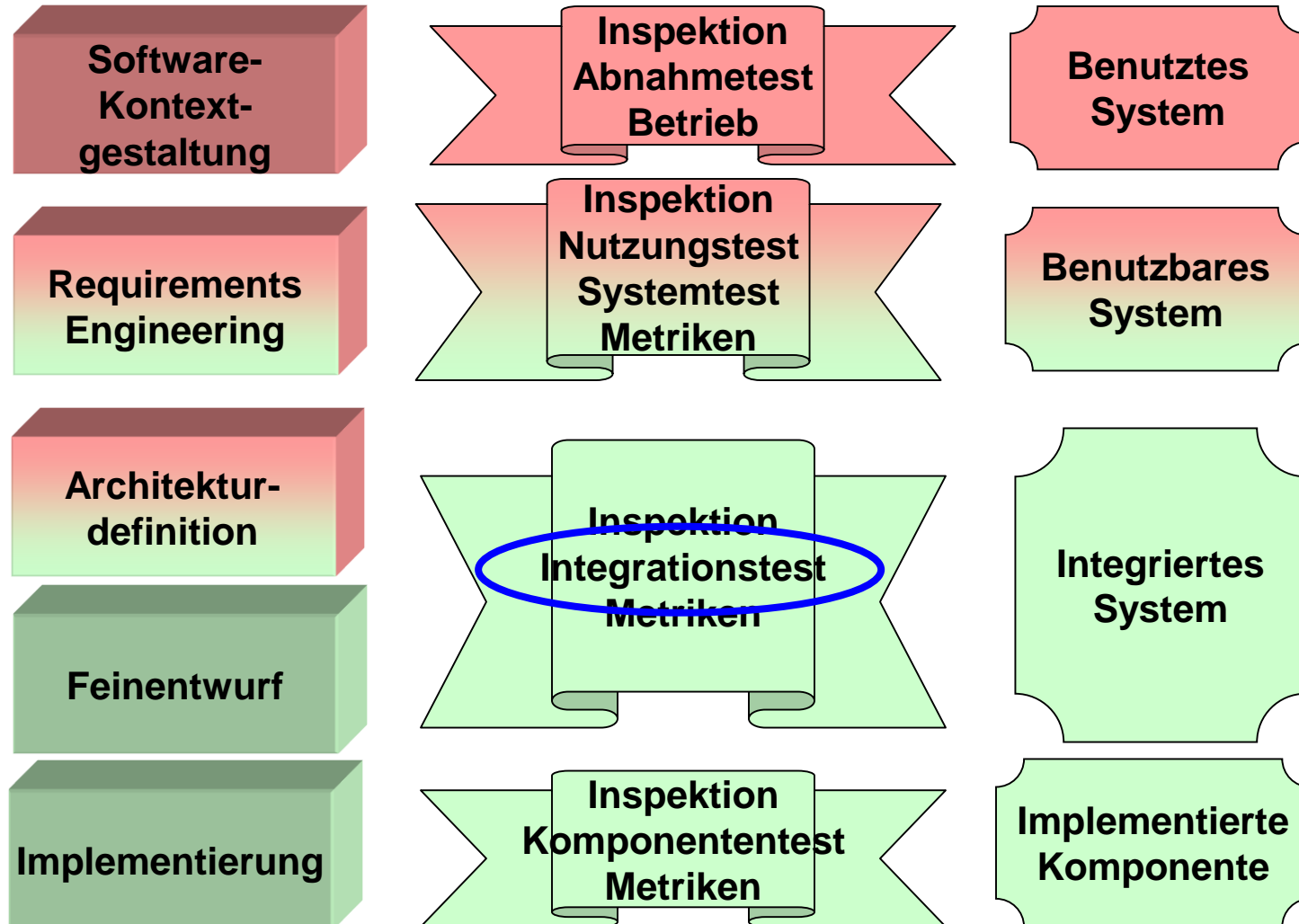
- Testfälle für Stapel sind alle Pfade im Ableitungsbaum bis zu einem Blatt
 - Init - delete
 - Init – push – pop (einmal so dass leer und einmal so dass gefüllt)
 - Init – push – push
 -

- Mögliche **Abhängigkeiten** zwischen den Methodenaufrufen
 - **Nicht-modal**: keine
 - Testfälle müssen internen Zustand nicht berücksichtigen
 - Teste verschiedene Reihenfolgen, um unerwünschte Seiteneffekt zu entdecken
 - Teste wiederholte Aufrufe der selben Methode
 - **Uni-modal**: feste Reihenfolge (z.B. Ampel)
 - Testfälle testen alle Reihenfolgen
 - **Quasi-modal**: inhaltsabhängige Reihenfolge (z.B. Stapel)
 - Testfälle für alle Zustände und Zustandsübergänge
 - **Modal**: fachliche Reihenfolge (z.B. Konto)
 - Wie quasi-modal
 - Zusätzliche Berücksichtigung fachlicher Zusammenhänge (z.B. Überziehungszinsen)

- Zusätzlich Negativ-Test!
 - Teste unzulässige Reihenfolgen

- Komponententest beinhaltet
 - Test der einzelnen Prozeduren (Operationen) der Komponente
 - Black-Box und White-Box
 - Test des Zusammenspiels der Operationen von Klassen
 - Zustandsbezogener Test
 - (siehe auch Integrationstest später)
- Komponenten sind oft die Klassen. Falls größere Komponenten: Test der einzelnen Klassen und dann Integrationstest der Klassen



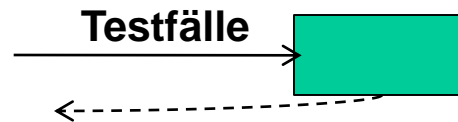


- **Integrationstest** testet **Zusammenspiel der Komponenten**
- setzt voraus, dass Komponenten schon getestet sind (inkl. Fehlerkorrektur)
- Testet gegenüber einer technischen Spezifikation (Entwurf)

- Komponenten werden **schrittweise** zusammengesetzt (abhängig von Integrationsteststrategie)
- Benötigt Einheiten, die Schnittstellen zwischen Komponenten beobachten (**Monitore**)

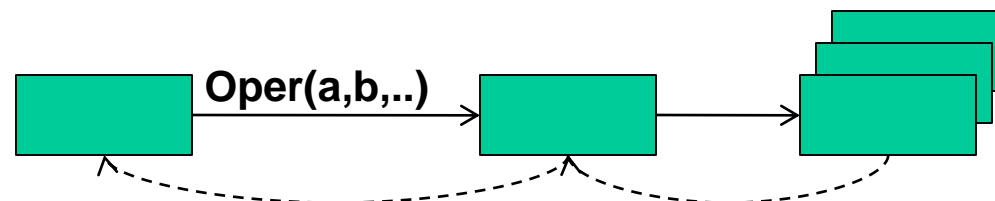
■ Komponententest

- Nur ein Testobjekt, Aufruf von Operationen durch Testtreiber mit durch Testtechnik bestimmten Eingaben (Testen der inneren Logik der Operation, **mglt. vollständige Abdeckung der Operationsabläufe**)



■ Integrationstest

- Mehrere Testobjekte, Aufruf von Operation aus einer anderen Operation heraus (Testen des Zusammenspiels, Schnittstelle, **mglt. vollständige Abdeckung des spezifischen Zusammenspiels**)



Wie hängen Komponenten voneinander ab?

■ Kommunikation

- Eine Komponente verwendet einen Dienst (Operation/Prozedur) einer anderen Komponente.

■ Gemeinsame Daten

- Zwei Komponenten greifen auf die gleichen Daten (z.B. Variablen, Datei, Datenbank) zu oder nutzen die gleiche Daten als Kommunikationsmedium, um Daten auszutauschen. Dieser Zugriff ist immer asynchron.

■ Gemeinsame Systemressourcen

- Zwei Komponenten verwenden die gleichen Systemressourcen (z.B. CPU, Speicher, Festplatte).

- Es können beim Zusammenspiel viele Probleme entstehen. Das sollte durch Tests weitgehend ausgeschlossen werden. Typische Probleme sind:
- Falscher **Empfänger** einer Nachricht
 - Falsche Komponente erhält die Nachricht
- Falscher **Dienst**
 - Fehlende, überlappende oder widersprüchliche Dienste
- Falscher **Aufrufzeitpunkt**
 - Verletzung der Vorbedingungen des Empfängers

- **Datenübergabeprobleme**
 - Übergebene Daten syntaktisch falsch
 - Übergebene Daten verschieden interpretiert

- Verletzung der **Nachrichtenreihenfolgen/-sequenzen**
 - Verletzung des Zusammenspiels der Kontrollzustandsdiagramme

- Fehler durch **inkorrektes Speichermanagement**
 - (allocation/deallocation)

- **Betriebssystem- und Middlewareaufrufe** (z.B. Datenbank, object request broker)

Gründe für Integrationsprobleme

- Oft Konfigurations- und Versionsinkonsistenzen (Probleme oft schon beim Binden)
- Unterschiedliche Interpretation der Anforderungen, insbes. bei verteilter Entwicklung (mangelnde Abstimmung im Team)
- Falsche Verwendung einer Schnittstelle
- Synchronisation bei Echtzeit

- Legen die **Reihenfolge** der Integrationsschritte fest (abhängig von Architektur, Projektplan und Testkonzept)
- Betrachte **Aufrufgraph**: X hat gerichtete Kante zu Y, wenn Y durch X aufgerufen wird
 - Blätter sind Komponenten, die nichts mehr aufrufen
 - Wurzeln sind Komponenten, die nicht aufgerufen werden
- **Top-Down**:
 - Beginne mit Wurzel A. Ersetze von A benötigte Komponenten mit **Stubs** und ersetze Stubs inkrementell mit richtigen Komponenten.
- **Bottom-Up**:
 - Beginne mit Blatt. Ersetze aufrufende Komponenten durch **Testtreiber** und ersetze Testtreiber inkrementell mit richtigen Komponenten.
- Typischerweise beides gemischt nötig!

Wdh. Testbegriffe (3)

■ Testrahmen (test bed)

- Sammlung aller Programme (u. a. Testtreiber und Platzhalter), die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen

■ Testumgebung

- Gesamtheit aller Hardware- und Softwarekomponenten (auch der Testrahmen), die notwendig sind, um Testfälle durchzuführen

■ Platzhalter (stub)

- Platzhalter werden beim Komponenten- und Integrationstest benötigt, um noch nicht implementierte Komponenten für die Testdurchführung zu ersetzen bzw. zu simulieren

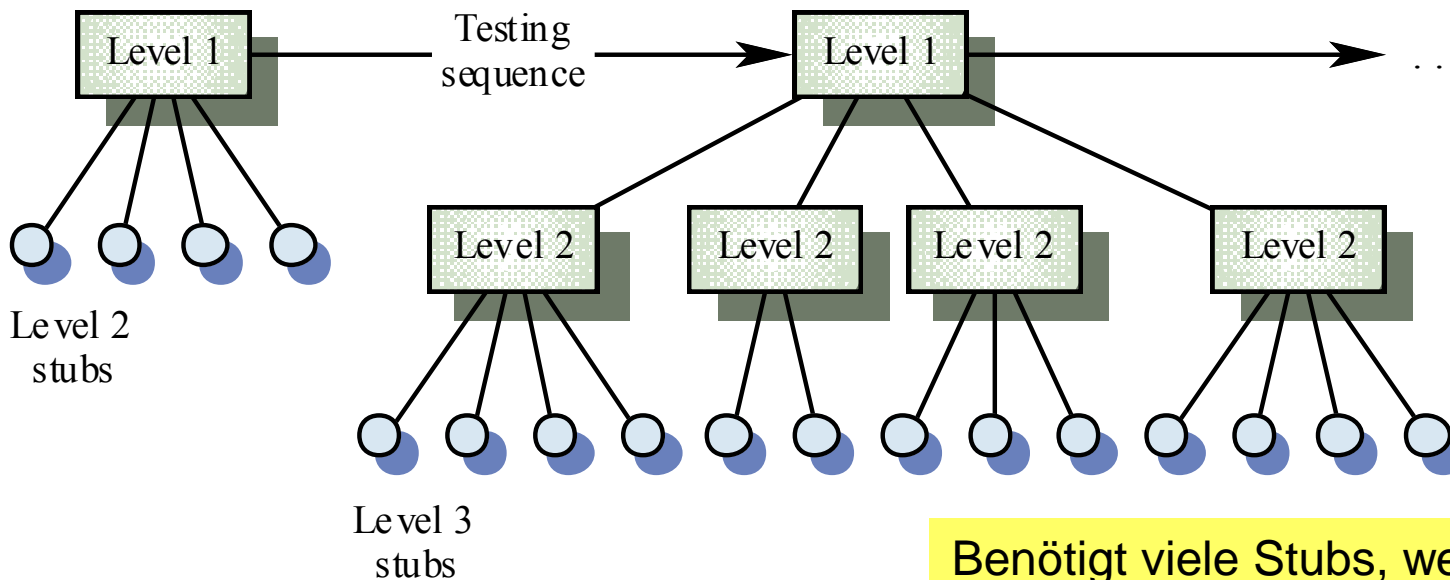
■ Testtreiber

- Programm, das ermöglicht, ein Testobjekt ablaufen zu lassen, mit Testdaten, zu versorgen und Ausgaben/Reaktionen des Testobjektes entgegenzunehmen.

Top-Down Integrationsteststrategie

Level 1 = Wurzel = Komponenten,
die von keinen anderen Komponenten aufgerufen werden

Level 2 = Komponenten,
die nur von Level 1 Komponenten aufgerufen werden

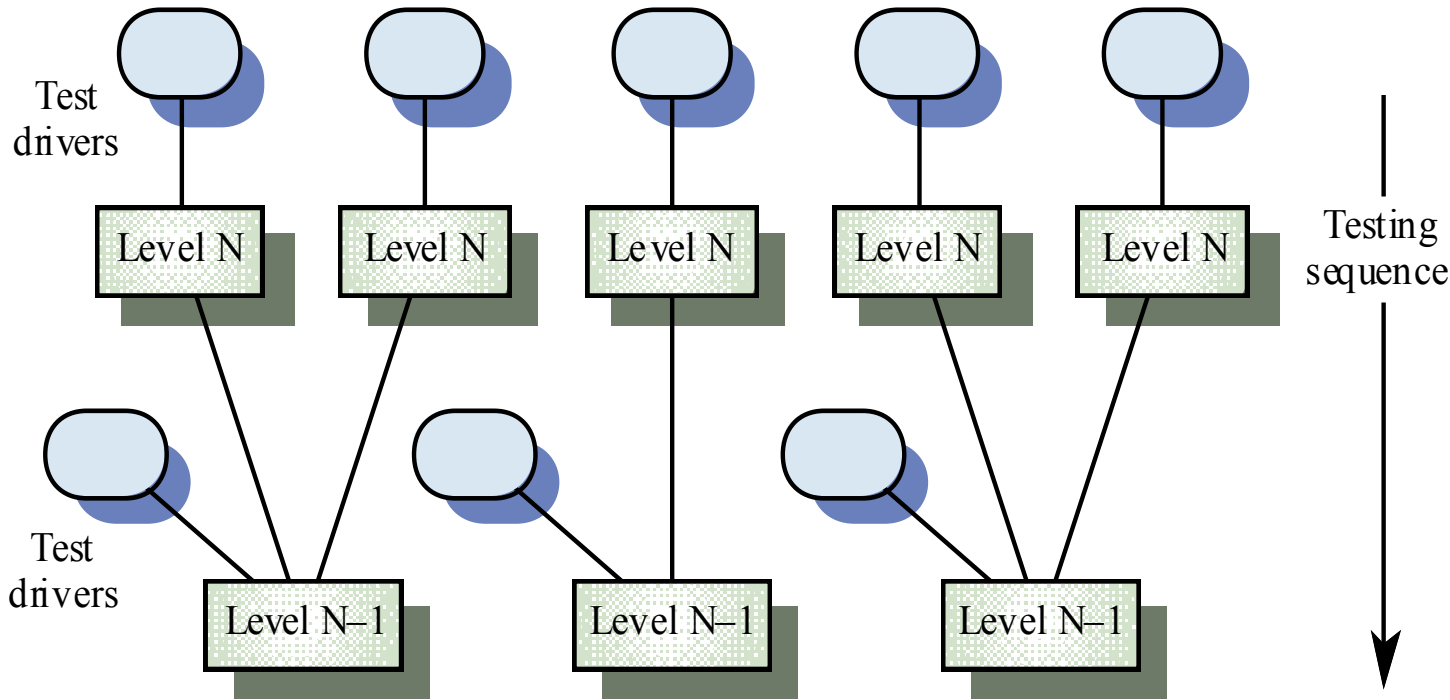


Benötigt viele Stubs, wenige Testtreiber

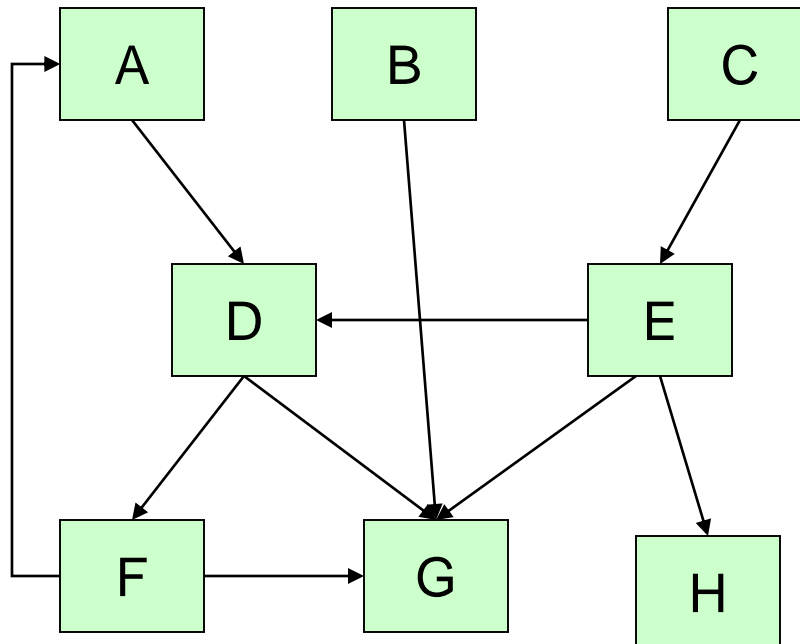
Bottom-Up Integrationsteststrategie

Level n = Blatt = Komponenten, die keine anderen Komponenten aufrufen

Level n-1 = Komponenten,
die nur Level n Komponenten aufrufen

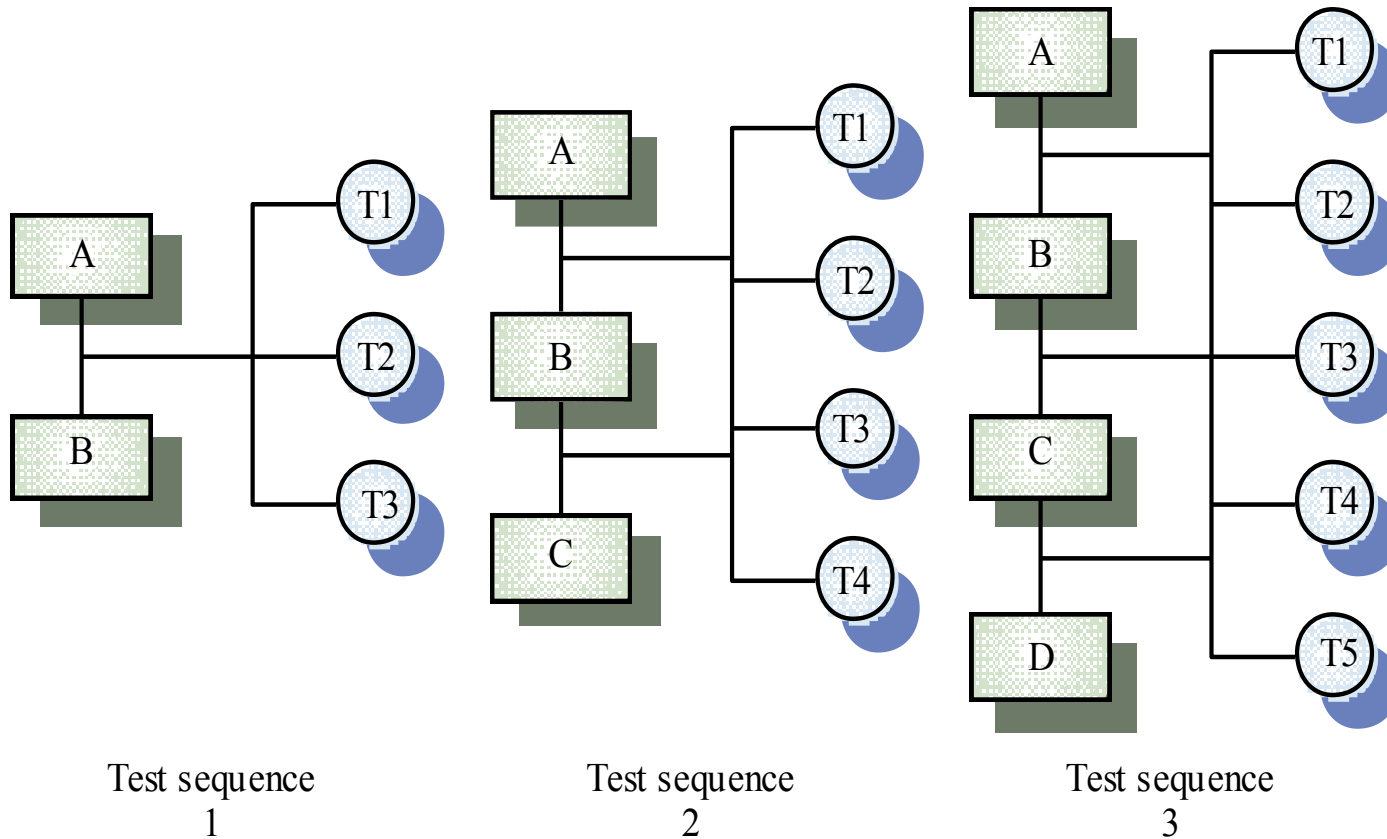


Benötigt viele Testtreiber, keine Stubs



Achtung! Zyklen müssen aufgebrochen werden

IMMER Inkrementelle Integration



Wiederholung der Testfälle stellt sicher, dass neue Komponente nicht bisheriges Zusammenspiel stört. Neue Testfälle testen neues Zusammenspiel.

- Testen objektorientierter Software benutzt auch Integrationstestideen
- Zuerst Testen einzelner Klassen (Einbezug von Vererbung)
- Danach Testen von gegenseitig abhängigen Objekten
 - Aufgrund mangelnder Hierarchie ist hier Top-Down oder Bottom-Up nicht strikt durchführbar
 - Besser z.B. szenario-basiert (zielorientiert) oder thread-basiert (ereignisorientiert)

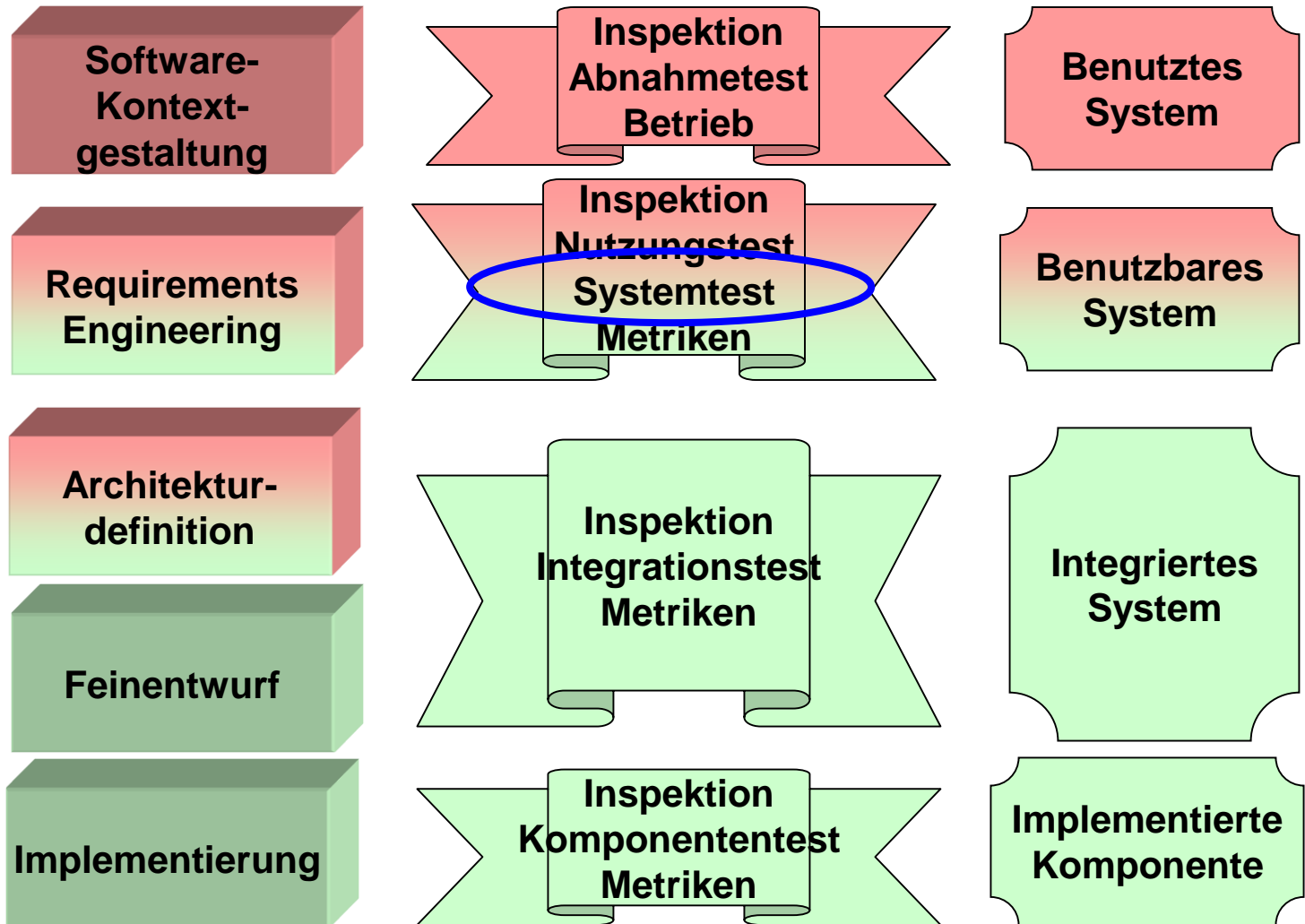
- Abhängigkeiten zwischen Komponenten
 - Kommunikation
 - Gemeinsame Daten
 - Gemeinsame Systemressourcen
- Abhängigkeiten zwischen Klassen
 - **Kommunikation**
 - Operationsaufrufe / Attributverwendung
 - Objekte oder Referenzen auf Objekte als Operationsparameter
 - **Gemeinsame Daten**
 - Globale Variablen
 - **Gemeinsame (System)Ressourcen**
 - Komposition / Aggregation (Erzeugungsabhängigkeiten)
 - Gemeinsamer Namensraum
 - **Vererbung**

- Operation nicht vorhanden (z.B. wg. Konfigurations- und Versionsproblemen)
- Falsche Operation/Objekt angesprochen (z.B. bei Bindung zur Laufzeit)
- Absturz/ Deadlock wegen Konflikten bei der Nutzung gemeinsamer Ressourcen (Prozesse, Benutzungsschnittstelle)

Zusammenfassung Integrationstest

- Integrationstest sehr wichtig, aber sehr aufwändig
- Bestimmung geeigneter Integrationsstrategie vor allem heuristisch
 - Testfokus: Welche Abhängigkeiten sind besonders fehleranfällig? (kann man aus Daten alter Projekte erheben)
 - Integrationsreihenfolge: Minimierung von Kosten, d.h. möglichst wenig Stubs und Treiber
- Integrationstest wird in der Praxis oft ausgelassen => BIG BANG beim Systemtest
- **Besser:** Use Cases (siehe später) als Testfall nehmen, aber dann wie Integrationstestfall behandeln => eigene Stubs, Treiber, Monitoring

Aktivitäten und Ergebnisse der Entwicklung und Qualitätssicherung



- Testet ob **Kunden-Anforderungen richtig umgesetzt** wurden (**Verifikation**)
- Testumgebung sollte **Produktiv-Umgebung mglst. nahe** kommen (also keine Stubs und Testtreiber)
- Produktiv-Umgebung oft selbst nicht geeignet, wegen Schadensrisiko und mangelnder Kontrolle
- Kann **einzelne Funktionen**, aber auch **Funktionssequenzen** (für Geschäftsprozesse) testen (siehe Use case basierter Test später)
- Sollte Test **von Qualitätsmerkmalen** beinhalten wie Performanz, Sicherheit (siehe später)

- Wenn man das gesamte System als eine Komponente ansieht, kann man **Systemtest als Black-Box-Test** durchführen, der die einzelnen Systemfunktionen testet.
- Probleme dabei sind
 - **Anforderungsspezifikation des Systems ist unübersichtlicher** als Spezifikation einer Komponente => Bestimmung von SOLL-Verhalten schwieriger
 - Expliziter Bezug zu Anforderungen wichtig
 - **Vor/Nachbedingungen und Eingaben sind komplexer:** müssen sehr viele Daten im System und Daten bzw. Aktionen auf der Benutzungsoberfläche berücksichtigen
 - **Funktionen machen oft erst im Zusammenspiel Sinn** (z.B. Konto Einzahlen, erst nach Anlegen) => es ist wichtig und oft einfacher typische Abläufe zu testen (siehe **Use Case basierter Test** später)

- **Identifikation der Funktionen** mit ihren Ein/Ausgaben aus der Spezifikation
- Gültige und ungültige **Äquivalenzklassen für Eingaben** (funktionsübergreifend bestimmen),
 - Grenzwerte beachten;
 - **unterschiedliche GUI-Aktionen** für die gleiche Eingabe sind unterschiedliche Äquivalenzklassen;
 - ungültige Äquivalenzklassen sollten die Fehlermeldungen abdecken
 - Ist eine **falsche Eingabe auf der GUI ausgeschlossen** (z.B. Auswahlliste), ist auch keine ungültige Äquivalenzklasse nötig
- **Äquivalenzklassen** für wichtige Unterschiede bei den **Ausgaben im Nichtfehler-Fall** bestimmen, Grenzwerte beachten
- Logische Testfälle entsprechend der Äquivalenzklassen bestimmen und konkrete Testfälle dazu auswählen

Testfallbeschreibung für Systemfunktionstest

- **Name** beschreibt den untersuchten Fall
- **Typ** Systemtest
- **Anforderung** benennt die für den Testfall wesentlichen Anforderungen
- **Vorbedingung**
 - Relevanter Zustand der Benutzungsschnittstelle und relevanter Zustand der Systemdaten
- **Nachbedingung**
 - Relevanter Zustand der Benutzungsschnittstelle und relevanter Zustand der Systemdaten
- **Testschritte**
 - **Eingabe:** Eingabewerte für Systemfunktion (egal ob durch NutzerIn eingegeben oder aus Systemzustand entnommen)
 - **erwartete Ausgabe:** Ausgabewerte der Systemfunktion (auch egal ob in der Funktion berechnet oder aus Systemzustand entnommen)
 - **erwartete Ausnahmen**

Zusammenfassung Teststufen

Teststufe	Komponenten	Integration	System
Zu entdeckende Fehler			
Test-rahmen			
Teststrategie			
Typische Fehler			

Zusammenfassung Teststufen

Teststufe	Komponenten	Integration	System
Zu entdeckende Fehler	Fehler in einzelnen Komponenten	Fehler beim Zusammenspiel bestimmter Komponenten	Fehler in Systemfunktionen
Testrahmen	Komponente, Stubs, Treiber	Komponenten, Stubs, Treiber, Monitore	Integriertes System
Teststrategie	Abdeckung der Operationen	Abdeckung der Abhängigkeiten (Top-Down, Bottom-Up)	Abdeckung der Systemfunktionen und UC
Typische Fehler	Falscher Kontrollfluss in den Operationen. Falsche Datenstrukturen	Probleme bei Empfänger, Daten, Vorbedingung, Reihenfolge, Synchronisation, Deadlock	Falsches Zusammenspiel Nutzungsschnittstelle und Systemkern

Es gibt noch weitere Teststufen: Abnahme-Test, Usability-Test => siehe später

- IEEE 829-1998, Standard for Software Test Documentation
- R. Binder: Testing object-oriented Systems, Addison-Wesley, 2000
- L. Copeland: A practitioner's Guide to Software Test Design, Artech Haouse Publishers, 2003
- J. Ludewig, H. Lichter, Software Engineering, dpunkt 2012
- A. Spillner, T.Linz: Basiswissen Softwaretest, dpunkt Verlag, 2010
- U. Vigerschow: Objektorientiertes Testen und Testautomatisierung in der Praxis, dpunkt Verlag, 2005

- Qualitätssicherung muss die ganze Softwareentwicklung begleiten
- Je nach Prüfobjekt andere Prüfstrategie und andere Prüftechniken (Test von Komponenten, von Integration, von System, von Nutzung sowie Inspektion oder Metrik)
- Inspektion und Metriken auf alle Artefakte anwendbar
- Näheres zu QS im Großen in Kapitel 3.

Kernfragen der Softwareentwicklung

- ✓ **Qualität:** Wie stellen wir sicher, dass das Softwaresystem **tut, was es soll?**
- ✓ **Beteiligte:** Wie erreichen wir, dass das Softwareprodukt **für die NutzerInnen nützlich** ist?
- ✓ **Beteiligte:** Wie erreichen wir, dass das Softwaresystem **effizient** zu entwickeln und insbesondere **für neue EntwicklerInnen verständlich und langfristig weiterzuentwickeln** ist?
- ✓ **Kosten/Zeit:** Wie stellen wir sicher, dass das Softwaresystem mit den **vorgegebenen Ressourcen** (Geld, Technologie, Leute) **im Zeit- und Kostenrahmen fertiggestellt** wird?



Bisher vor allem
Kosten/Zeit
(agiler Prozess)
und Qualität



2.7. Kommunikation im Projekt

Kommunikation im Projekt

Kommunikation mit KundIn

Eigenschaften von Software-Projekten

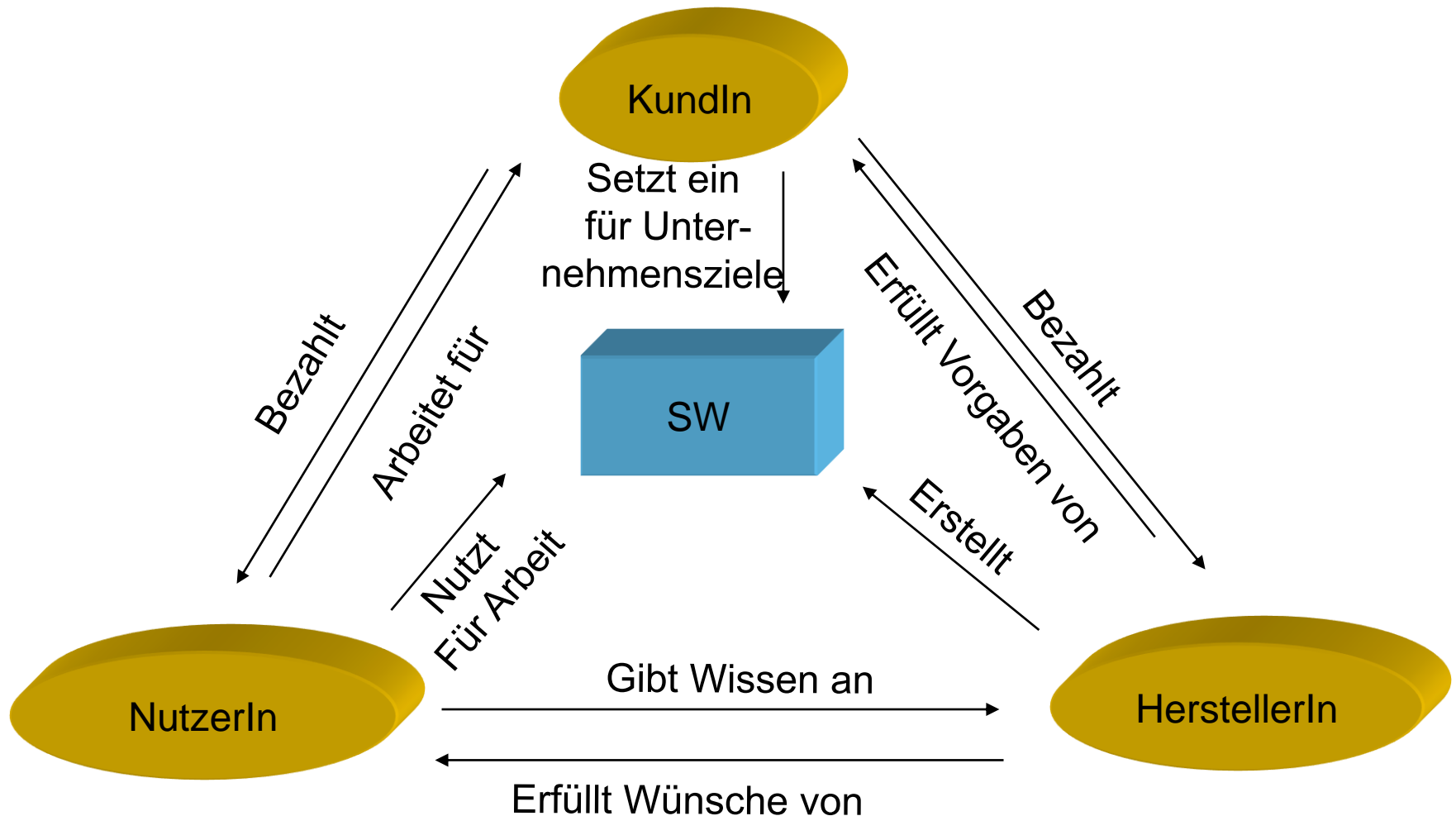
- Was lässt sich über (Software-)Projekte wirklich Allgemeingültiges aussagen?
 - Die **Laufzeit** jedes Projekts ist **begrenzt**.
 - Jedes Projekt hat einen „Erzeuger“ (= eine Person oder Institution, die es initiiert hat, typisch das höhere Management).
ProjekteigentümerIn ist der Erzeuger oder vertritt dessen Interessen.
ProjektleiterIn ist gegenüber ProjekteigentümerIn verantwortlich.
 - Jedes Projekt hat einen **Zweck**, ein Bündel von Zielen.
Das wichtigste Ziel ist meist, eine **Software herzustellen oder zu verändern**; sie ist also das Resultat des Projekts, das **Produkt**.

Andere wichtige Ziele sind: Erweiterung des Know-hows,
Bereitstellung von Bausteinen für spätere Projekte,
Auslastung der MitarbeiterInnen.

Eigenschaften von Software-Projekten (2)

- Werden die Ziele in hohem Maße erreicht, so ist das **Projekt erfolgreich**.
 - Das Produkt hat einen „Abnehmer“ oder wird (hoffentlich) einen haben. Dieser Abnehmer ist der/die **KundIn**. Die NutzerInnen gehören zur KundIn.
 - Das Projekt verbindet **Menschen**, **Resultate** (Zwischen- und Endprodukte) und **Hilfsmittel (Ressourcen)**.
 - Die **Organisation** bestimmt deren **Rollen** und **Beziehungen** und die **Schnittstellen** des Projekts nach außen.
- Wir verwenden hier die Begriffe **KundIn** und **HerstellerIn**.
- HerstellerIn = Oberbegriff für alle Organisationen, die irgendetwas, insbesondere Software, produzieren. Auch **Dienstleistungen** sind eingeschlossen.
Damit können auch Behörden, Universitäten und Einzelpersonen unter den Begriff HerstellerIn fallen.

Beteiligte beim SWE



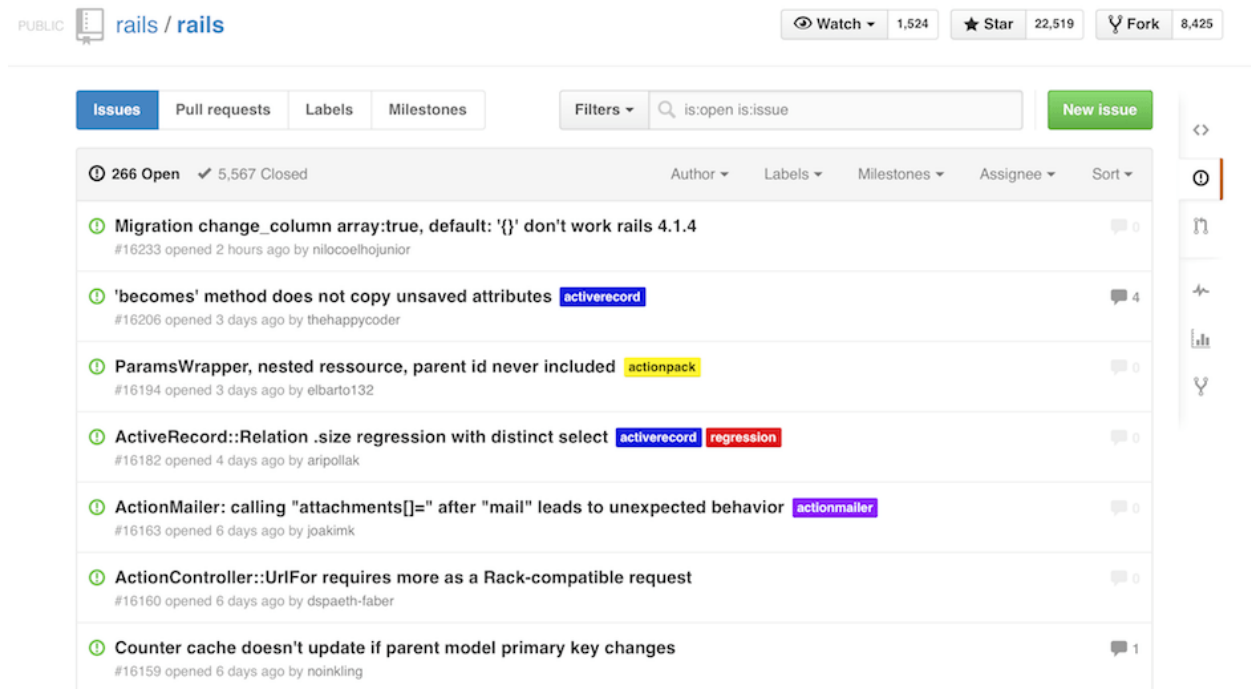
Interessengruppen	Beweggrund	Sachkenntnis
Software EntwicklerIn(1)	Ein technisch hervorragendes System zu produzieren und die neuesten Techniken zu verwenden	Neueste Techniken und kreatives Entwurfsgeschick
Software EntwicklerIn(2)	Bereits bestehende Software oder Entwürfe zu verwenden	Kenntnis bestehender Systeme
System –AnalytikerIn	Genaue Anforderungen zu produzieren	Problemanalyse
Technische/r AutorIn	Lehrmaterial zu entwickeln, das den Bedürfnissen der Anwender entspricht	Schriftstellerisches Geschick; Design der Dokumentation
NutzerInnenvertretung	Änderung mit möglichst wenig Störung und möglichst großem Nutzen einzuführen	Kenntnis der Organisation, der NutzerInnen und Aufgaben
Training und NutzerInnen-unterstützung	NutzerInnen zu unterstützen und neue Unterstützung zu erzeugen	Kenntnis der Nutzungsprobleme
Unternehmens-/Markt-analytikerIn	Die Konkurrenz zu übertrumpfen	Kenntnis der Unternehmens- und Marktbedürfnisse
Projektmanagement	Das Projekt erfolgreich mit den zur Verfügung gestellten Mitteln durchzuführen	Kenntnis der Produktplanung und vorhergegangener Projekte

=> Gleiche Rolle heißt nicht immer gleiche Interessen

[Macaulay 1996]

- Beteiligte verteilen Arbeit und stimmen sich darüber ab (Issue Tracker)
- Konkrete Verteilung ist von der Projektorganisation abhängig

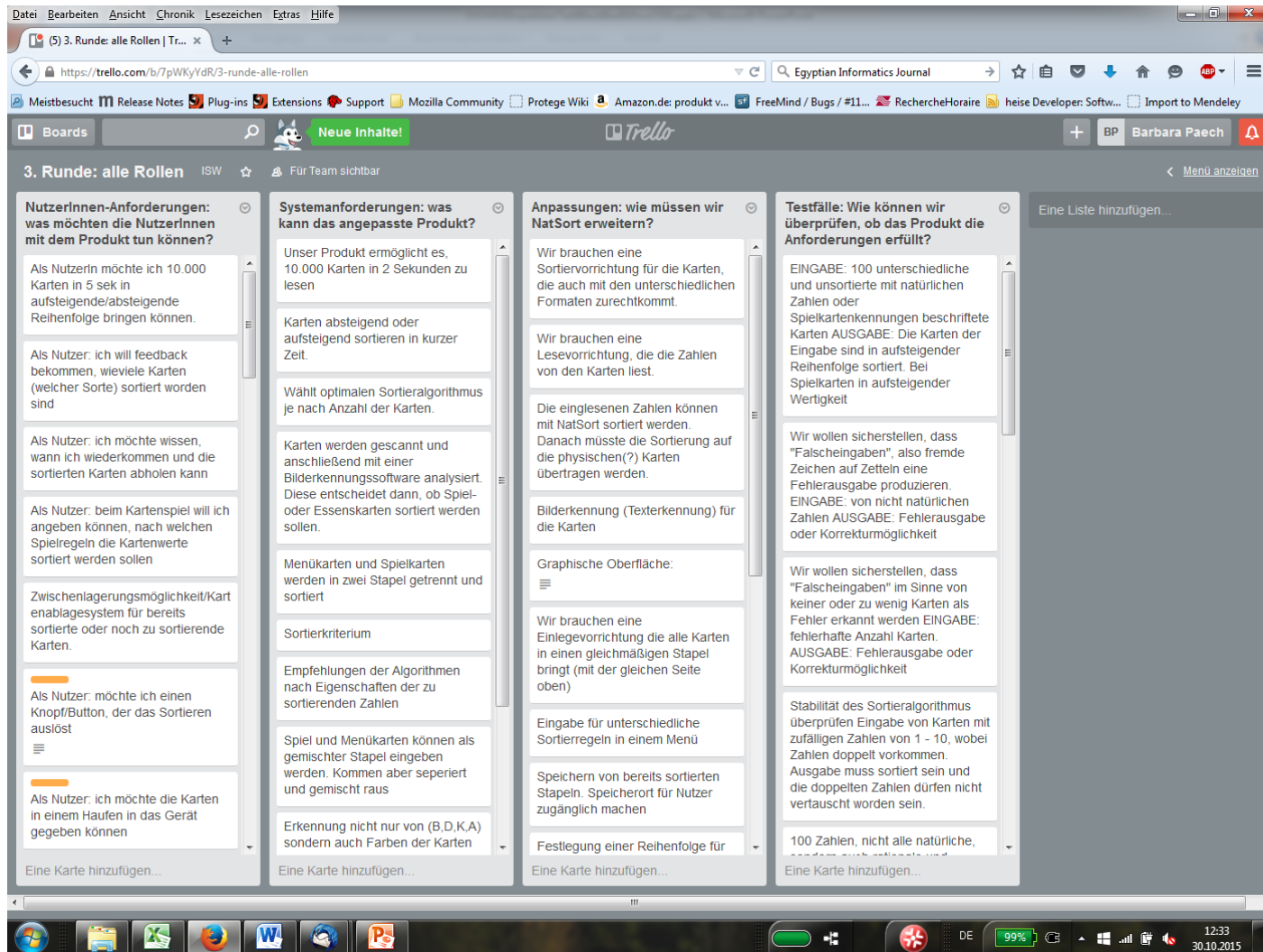
<https://github.com/features>



The screenshot shows the GitHub interface for the 'rails / rails' repository. At the top, it indicates the repository is 'PUBLIC' and shows statistics: 1,524 Watchers, 22,519 Stars, and 8,425 Forks. Below this, there are tabs for 'Issues', 'Pull requests', 'Labels', and 'Milestones'. A search bar contains the text 'is:open is:issue'. To the right of the search bar is a 'New issue' button. The main content area displays a list of issues, each with a status icon (open or closed), a title, a number, and the author. The issues listed are:

- #16233 Migration change_column array:true, default: '{}' don't work rails 4.1.4 (opened 2 hours ago by nilocoelhojunior)
- #16206 'becomes' method does not copy unsaved attributes (opened 3 days ago by thehappycode)
- #16194 ParamsWrapper, nested resource, parent id never included (opened 3 days ago by elbarto132)
- #16182 ActiveRecord::Relation .size regression with distinct select (opened 4 days ago by aripollak)
- #16163 ActionMailer: calling "attachments[]" after "mail" leads to unexpected behavior (opened 6 days ago by joakimk)
- #16160 ActionController::UrlFor requires more as a Rack-compatible request (opened 6 days ago by dspaeth-faber)
- #16159 Counter cache doesn't update if parent model primary key changes (opened 6 days ago by noinkling)

Beispiel: Trello als Issue Tracker



The screenshot shows a Trello board titled "3. Runde: alle Rollen" (Round 3: all roles) with four columns representing different user requirements and system specifications. The board is organized into four columns, each with a title and a list of requirements or specifications. The columns are:

- NutzerInnen-Anforderungen: was möchten die NutzerInnen mit dem Produkt tun können?** (User requirements: what do users want to be able to do with the product?)
 - Als NutzerIn möchte ich 10.000 Karten in 5 sek in aufsteigende/absteigende Reihenfolge bringen können.
 - Als Nutzer: Ich will feedback bekommen, wieviele Karten (welcher Sorte) sortiert worden sind
 - Als Nutzer: ich möchte wissen, wann ich wiederkommen und die sortierten Karten abholen kann
 - Als Nutzer: beim Kartenspiel will ich angeben können, nach welchen Spielregeln die Kartenwerte sortiert werden sollen
 - Zwischenlagerungsmöglichkeit/Kart enablagensystem für bereits sortierte oder noch zu sortierende Karten.
 - Als Nutzer: möchte ich einen Knopf/Button, der das Sortieren auslöst
 - Als Nutzer: ich möchte die Karten in einem Haufen in das Gerät gegeben können
- Systemanforderungen: was kann das angepasste Produkt?** (System requirements: what can the adapted product do?)
 - Unser Produkt ermöglicht es, 10.000 Karten in 2 Sekunden zu lesen
 - Karten absteigend oder aufsteigend sortieren in kurzer Zeit.
 - Wählt optimalen Sortieralgorithmus je nach Anzahl der Karten.
 - Karten werden gescannt und anschließend mit einer Bilderkennungssoftware analysiert. Diese entscheidet dann, ob Spiel- oder Essenskarten sortiert werden sollen.
 - Menükarten und Spielkarten werden in zwei Stapel getrennt und sortiert
 - Sortierkriterium
 - Empfehlungen der Algorithmen nach Eigenschaften der zu sortierenden Zahlen
 - Spiel und Menükarten können als gemischter Stapel eingegeben werden. Kommen aber seperiert und gemischt raus
 - Erkennung nicht nur von (B,D,K,A) sondern auch Farben der Karten
- Anpassungen: wie müssen wir NatSort erweitern?** (Adaptations: how do we need to extend NatSort?)
 - Wir brauchen eine Sortiervorrichtung für die Karten, die auch mit den unterschiedlichen Formaten zurechtkommt.
 - Wir brauchen eine Lesevorrichtung, die die Zahlen von den Karten liest.
 - Die einglesenen Zahlen können mit NatSort sortiert werden. Danach müsste die Sortierung auf die physischen(?) Karten übertragen werden.
 - Bilderkennung (Texterkennung) für die Karten
 - Graphische Oberfläche:
 - Wir brauchen eine Einlegevorrichtung die alle Karten in einen gleichmäßigen Stapel bringt (mit der gleichen Seite oben)
 - Eingabe für unterschiedliche Sortierregeln in einem Menü
 - Speichern von bereits sortierten Stapeln. Speicherort für Nutzer zugänglich machen
 - Festlegung einer Reihenfolge für
- Testfälle: Wie können wir überprüfen, ob das Produkt die Anforderungen erfüllt?** (Test cases: how can we check if the product meets the requirements?)
 - EINGABE: 100 unterschiedliche und unsortierte mit natürlichen Zahlen oder Spielkartenkennungen beschriftete Karten AUSGABE: Die Karten der Eingabe sind in aufsteigender Reihenfolge sortiert. Bei Spielkarten in aufsteigender Wertigkeit
 - Wir wollen sicherstellen, dass "Falscheingaben", also fremde Zeichen auf Zetteln eine Fehlerausgabe produzieren. EINGABE: von nicht natürlichen Zahlen AUSGABE: Fehlerausgabe oder Korrekturmöglichkeit
 - Wir wollen sicherstellen, dass "Falscheingaben" im Sinne von keiner oder zu wenig Karten als Fehler erkannt werden EINGABE: fehlerhafte Anzahl Karten. AUSGABE: Fehlerausgabe oder Korrekturmöglichkeit
 - Stabilität des Sortieralgorithmus überprüfen Eingabe von Karten mit zufälligen Zahlen von 1 - 10, wobei Zahlen doppelt vorkommen. Ausgabe muss sortiert sein und die doppelten Zahlen dürfen nicht vertauscht worden sein.
 - 100 Zahlen, nicht alle natürliche, sondern auch zufällige und

Teamorganisation

- Software-Entwicklung ist ein **arbeitsteiliger Prozess**.
- Die am Projekt beteiligten Personen – das **Projektteam** – sind zu **organisieren**:
 - Festlegung von **Weisungsbefugnis**
 - Festlegung von **Kommunikationswegen**
 - Je nach Größe des Projekts kann das Projektteam in mehrere Teams zerfallen. Ein Team sollte einen **definierten Bereich** bearbeiten und aus **höchstens fünf bis sieben Personen** bestehen.
- Im Folgenden werden einige typische Organisationsformen für Teams vorgestellt. Diese Übersicht ist natürlich **holzschnittartig** vereinfacht, in der Praxis gibt es alle möglichen **Mischformen**.



„Ein-Personen-Team“ („Einzelkämpfer“)

- Überall, wo sehr begrenzte Aufgaben zu bearbeiten oder einfach nicht mehr Leute verfügbar sind, werden „Einzelkämpfer“ eingesetzt.
 - **Merkmale:** Eine Person, arbeitet sehr selbständig an einer Aufgabe.
 - **Vorteil:** Fast kein Kommunikationsaufwand.
 - **Nachteile:** Keine GesprächspartnerIn, kein Dokumentationsdruck, Risiko des Ausfalls.
 - **Typisch für:** kleine Projekte, Aufgabenpakete, die sich nicht weiter sinnvoll über mehrere Personen verteilen lassen, unregelmäßige Wartungsaufgaben, Wartung und Weiterentwicklung von Produkten.
 - **Empfehlung:** „Einzelkämpfer“ einbinden, vernetzen.

Gruppen aus zwei Personen

- **Merkmale:** Entstehung der Gruppe
 - durch den freien Beschluss der Beteiligten zur Zusammenarbeit. Keine Führungsrolle! („Doppel“) oder
 - durch die Weisung an eineN HelferIn („Sherpa“), eineN SpezialistIn zu unterstützen. („Tandem“)
- **Vorteile:** Gespräch möglich, implizite QS, keine Katastrophe durch Ausfall einer Person.

Sinn des Tandems kann es auch sein, das Wissen des Spezialisten auf zwei Köpfe zu verteilen.
Folge: Sherpa = SchülerIn
- **Nachteile:** Schwierig, wenn sich die beiden nicht gut verstehen
- **Typisch für:** Pair Programming; Einarbeitung

Anarchisches Team

- **Merkmale:** Die EntwicklerInnen arbeiten im Wesentlichen autonom, nach eigenen Vorgaben und Maßstäben. Hierarchische Beziehungen fehlen oder werden faktisch ignoriert, weil der Wille, die Zeit und/oder die Fähigkeit zur Führung fehlen.
- **Vorteile:** EntwicklerInnen sind selbstbestimmt, keine Hierarchie-Probleme, kaum bürokratische Hemmnisse.
- **Nachteile:** Standards, Normen lassen sich nicht durchsetzen. Die Entstehung der erforderlichen Resultate ist Glückssache (d. h. gewisse Dokumente entstehen in aller Regel **nicht**). Die Organisation insgesamt ist nicht lernfähig, Planung, Einführung neuer Methoden und Werkzeuge sind von der Laune der MitarbeiterInnen abhängig.
- **Typisch für:** Organisationen mit schwacher Führungsstruktur, d. h. Kleingruppen; „Einzelkämpfer“ können in der Regel als anarchisch eingestuft werden; das gilt auch für Studierende ☺ ; Behörden, Forschungseinrichtungen, auch Firmen.

Demokratisches Team

- **Merkmale:** Die Beteiligten sind grundsätzlich gleichberechtigt. Sie erzielen durch ausreichende Kommunikation einen Konsens über die Ziele und Wege, und sie verhalten sich diszipliniert.
- **Vorteile:** Die Fähigkeiten der Beteiligten werden optimal genutzt, Probleme werden frühzeitig erkannt und gemeinsam bekämpft.
- **Nachteile:** Hoher Kommunikationsaufwand. Unter Umständen Paralyse (Dissens, Fraktionsbildung)
- **Typisch für:** Forschungsgruppen (unter günstigen Umständen), auch kleine Software-Unternehmen.

Hierarchisches Team

- **Merkmale:** Die Gruppe steht unter der Leitung einer Person, die für die Personalführung, je nach Projektform auch für das Projekt verantwortlich ist.
Varianten: GruppenleiterIn übernimmt Stabs- oder Linienfunktion.
- **Vorteile:** Einfache Kommunikationsstruktur, klare Zuständigkeiten, auf Mitarbeiterebene gute Ersetzbarkeit.
- **Nachteile:** Lange Kommunikationswege (d. h. oft schlechte Information), GruppenleiterIn stellt ein hohes Risiko dar; Gruppenmitglieder sind kaum zur Kooperation motiviert.
- **Typisch für:** Unternehmen und Behörden — die traditionelle Organisationsform.

Chief-Programmer-Team

- **Merkmale:** Spezielle Variante des hierarchischen Teams. Die wesentlichen Unterschiede sind die Differenzierung der Rollen in der Gruppe und die Entwickler-Funktion der/s **Chief-Programmers**.
 - Die Gruppe besteht aus der/m Chief-Programmer und **StellvertreterIn, BibliothekarIn**, die/der alle Verwaltungsfunktionen übernimmt, sowie aus einigen **ProgrammierernInnen**.
 - Zusätzlich kann es weitere **SpezialistInnen** geben, z.B. Projekt-VerwalterIn, "Werkzeugmacher", DokumentiererIn, Sprach- oder System-ExpertInnen, TesterIn.
- **Vorteile:** Die Gruppe kann — wie ein Operationsteam, das Vorbild dieser Struktur war — außerordentlich effizient arbeiten.
- **Nachteile:** Hohe Ansprüche an die Disziplin, vermutlich auch die Gefahr, dass die/der Chief-Programmer „abhebt“, sich überschätzt.
- **Typisch für:** ??? (mit Einschränkungen: Open-Source-Projekte) .Diese Struktur wurde bei IBM in USA erfunden, es ist unklar, ob sie anderswo auch eingesetzt wurde und mit welchem Erfolg.

2.7. Kommunikation im Projekt

Kommunikation im Projekt

Kommunikation mit KundIn

- Die Auftragsbeziehung mit der/m KundIn bestimmt wesentlich, welche Kommunikation möglich ist
 - KundIn in gleicher Firma wie HerstellerIn?
 - Kommunikation einfacher, aber oft implizit
 - Bezahlung meist implizit
 - KundIn in anderer Firma wie HerstellerIn?
 - Kommunikation schwieriger wg. anderer Ort, Kultur
 - Bezahlung durch Vertrag geregelt
 - Kommunikation oft durch formale Dokumente

Auftragsbeziehungen (1)

- Projekte lassen sich klassifizieren nach Art der **Auftragsbeziehung** und der **Abrechnung**.
- **KundIn und HerstellerIn in derselben Organisation:**
 - **Entwicklungsprojekt** (z.B. bei großem Software-Hersteller wie SAP)
 - Ein Software-Produkt wird entwickelt, damit es später **auf dem Markt** angeboten werden kann.
 - KundIn = AuftraggeberIn = Marketingabteilung; Finanzierung aus dem **Entwicklungsbudget**.
 - **EDV-Projekt** (z.B. bei großen Banken, Versicherungen, Produzenten wie BASF)
 - Im Hause wird Software für den **eigenen Bedarf** entwickelt.
 - HerstellerIn und KundIn sind in derselben Organisation, Bezahlung erfolgt mit „Papiergeld“. Konflikte werden durch gemeinsame Vorgesetzte gelöst (oder nicht gelöst).

Auftragsbeziehungen (2)

■ Auftragsprojekt

- HerstellerIn entwickelt die Software nach den Wünschen von **externem/r KundIn**.
- Klare Rollenverteilung, expliziter Vertrag, Lieferungen sind mit Zahlungen verknüpft. Oft gibt KundIn Merkmale des Projekts vor, z. B. die einzusetzenden Programmiersprachen.
- Bei kleiner HerstellerIn oft als Basis für Entwicklungsprojekt
- **Spezialfall: Systemprojekt**
 - KundIn bestellt ein **komplexes System**, z.B. eine Industrieanlage. Das System enthält mehr oder minder viel Software (d. h. die Software ist **Teil des Systems**). Die Software-EntwicklerInnen erbringen ihre Leistung also zusammen mit den übrigen IngenieurInnen der HerstellerIn.
 - Unterschiedlich starke Einflussnahme der KundIn auf die Software!

Zusammenfassung der Projekttypen

Projektart	Ergebnis	AuftraggeberIn		
		internes Marketing	internes Management	externeR KundIn
Entwicklungs- - projekt	Produkte, Systeme für den Markt	x	(x)	(x)
EDV- Projekt	Datenverwaltung, Informationssysteme		x	(x)
Auftrags- projekt	Kundenspezifisches Softwaresystem			x
System- projekt	Industrieanlagen, technische Systeme		(x)	x

- In kleinen Firmen wird meist vorhandene Software bearbeitet. Das ist dem EDV-Projekt ähnlich, aber KundIn und HerstellerIn nicht in gleicher Organisation.

KundIn und Anforderungen

- KundIn erwartet, dass die Software über einen gewissen Zeitraum hinweg als **williger und billiger Diener** zur Verfügung steht, also
 - bestimmte **Leistungen erbringt**,
 - **ohne** umgekehrt **erhebliche Leistungen** (in Form von Kosten, Aufwand, Mühe, Ärger) **zu fordern**.
Die Software soll dienen, nicht umgekehrt.
- Werden die Erwartungen, die Anforderungen der/s KundIn, nicht vollständig und präzise erfasst, ist damit zu rechnen, dass das entwickelte Produkt die Anforderungen nicht (vollständig) erfüllt.
- Die vollständige und präzise Erfassung der Anforderungen ist die **allerwichtigste technische Voraussetzung** für eine erfolgreiche Software-Entwicklung.

- Anforderungen werden meist nur in kurzen Sätzen erfasst
 - Bei agilen Projekten z.B. User Story : Als NutzerIn X will ich Y tun
 - Diese werden als **Feature Request** meist auch in Issue Tracker verwaltet
 - Das ist gut zur Arbeitsverteilung, aber es fehlt der Überblick, was die Software insgesamt tut (keine zusammenhängenden Anforderungsspezifikation)
 - Das ist insbesondere problematisch, wenn Software und Team kontinuierlich wachsen.
-
- siehe Hausaufgabe Blatt 4
 - siehe SWE im Großen ab nächster Woche

Zusammenfassung SWE im Kleinen

- SWE im Kleinen sollte agil organisiert werden
 - Schlanke Dokumente
 - Viel Kommunikation (auch mit KundIn)
 - Viel Qualitätssicherung

- Siehe auch Gute Programmierung (nächste Folie, grüne Themen noch nicht behandelt)

Gute Programmierung

- **Programmierung (größtenteils nicht in ISW)**
 - Code Guidelines and Code Layout => siehe Richtlinien
 - Errors, Error Handling, and Exceptions => Programmierung
 - Programming Languages and Paradigms => Programmierung
 - Performance, Optimization and Representation
 - Refactoring and Code Share
 - Simplicity
- **Kapitel Kommunikation Entwickler**
 - Domain Thinking => Modellierung
- **Kapitel Softwareentwicklungsprozess**
 - Design Principles and Coding Techniques
=> siehe Entwurf, Evolution
- **Kapitel Qualitätssicherung**
 - Tests, Testing and Testers => Testen
 - Bugs and Fixes => Fehlermanagement
 - Build and Deploy => Build Management
 - Learning, Skills and Expertise (u.a. => Inspektionen)
 - Nocturnal or Magical (u.a. => Build Management)
 - Teamwork and Collaboration => Pairprogramming
 - Tools, Automation and Development Environments => Übung Eclipse, EMF; UNICASE
- **Kapitel Kommunikation Kunde**
 - Users and Customers
- **Kapitel Evolution**
 - Reuse vs. Repetition
- **Kapitel Projektmanagement**
 - Schedules, Deadlines and Estimates

- K. Henney (ed.) , 97 Things Every Programmer should know, O'Reilly, 2010