

# Software Engineering

## SS 2005

**Prof. Dr. Barbara Paech, Jürgen Rückert**



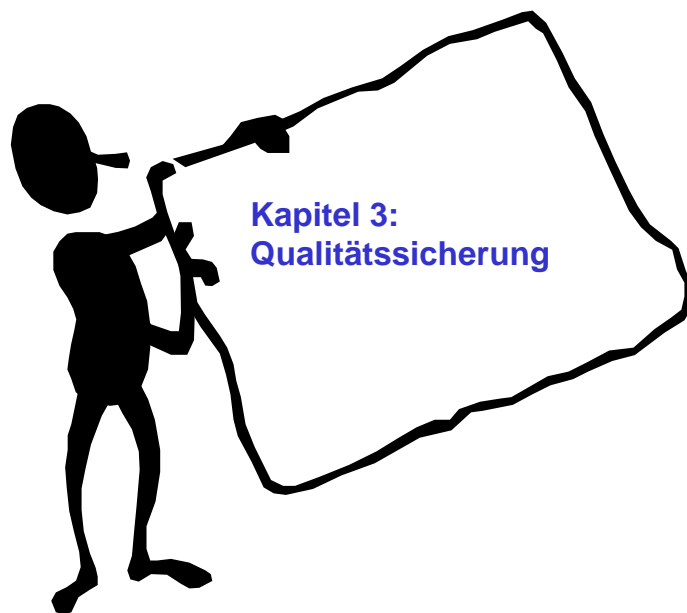
Institut für Informatik  
Im Neuenheimer Feld 326  
69120 Heidelberg  
<http://www-swe.informatik.uni-heidelberg.de>  
[paech@informatik.uni-heidelberg.de](mailto:paech@informatik.uni-heidelberg.de)



RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

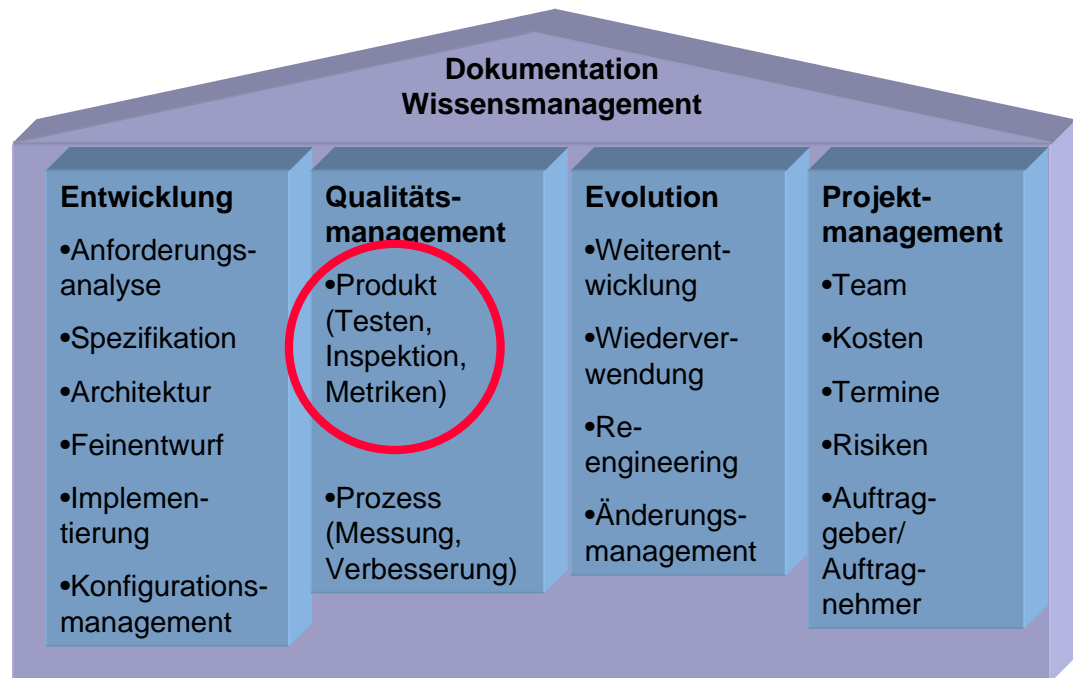


- 3. Qualitäts-sicherung
  - 3.1. Einführung
  - 3.2. Testen
    - 3.2.1. Kompen-  
tentest



## 1.3.1. Aufgabenbereiche des Engineering

3. Qualitäts-  
sicherung
- ▶ 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Kompen-  
tentest



## 3.1. Was ist Qualität?

3. Qualitäts-  
sicherung
- ▶ 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Kompen-  
tentest

- ◆ **Qualität:** Gesamtheit von Merkmalen und Merkmalswerten einer Einheit bzgl. ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen [ISO8402]
- ◆ Achtung: bezieht sich auf **Produkt und Prozess!**
- ◆ Achtung: Erfordernisse (Anforderungen) **ändern** sich über die Zeit!
- ◆ Achtung: **Qualität** muss immer mit **Kosten und Zeit** in Bezug gesetzt werden

## 3.1. Berichtigtes Beispiel: Ariane 501

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompen-  
tentest

- ◆ Folien dazu auszugsweise übernommen von Holger Schlingloff, Humboldt Universität Berlin
- ◆ Ariane5 als Nachfolgerin der Ariane4-Familie mit über 100 erfolgreichen Starts
- ◆ 6-12t Nutzlast (gegenüber 2-5t A4)
- ◆ Jungfernflug am 4.6.1996



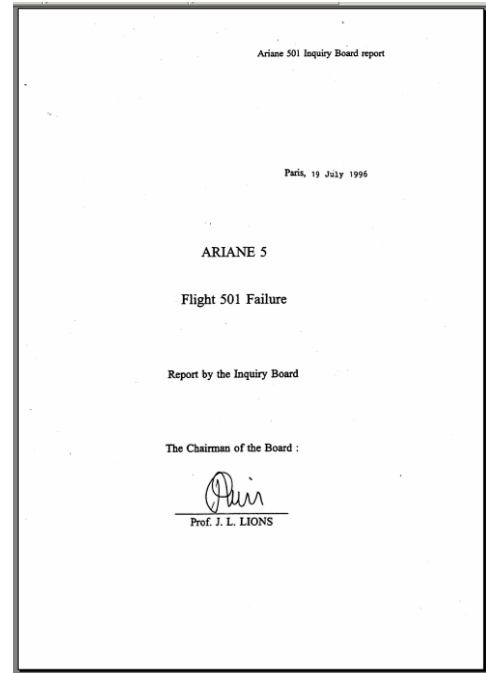
3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompen-  
tentest

- ◆ Missionsverlust
  - Nutzlast zerstört, Kosten > 500 M€
  - Programm 3 Jahre aufgehalten
- ◆ Untersuchungskommission
  - Bericht vom 19.6.1996 (nach nur 14 Tagen !!!)
  - erhältlich unter <http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf>



Folie 7

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompen-  
tentest

- ◆ **H<sub>0</sub>-3 Sek.** Das Haupttriebwerk wird gezündet.
- ◆ **H<sub>0</sub> Sek.** Beide Booster werden gezündet, Start endgültig.
- ◆ **H<sub>0</sub>+7,5 Sek.** Die Haltebolzen werden gelöst, A501 hebt ab.
- ◆ **H<sub>0</sub>+37 Sek.** Die Rakete befindet sich in einer Höhe von 3,5km und hat eine Geschwindigkeit von 857 km/h, als plötzlich die Steuermotoren beide Boosterdüsen und das Haupttriebwerk bis zum Anschlag lenken.
- ◆ **H<sub>0</sub>+39 Sek.** Die Lage der Rakete ist schräg zu ihrer Flugbahn. Durch die einwirkenden aerodynamischen Kräfte beginnt die Rakete auseinander zu brechen.
- ◆ **H<sub>0</sub>+41 Sek.** Der automatische Selbstzerstörungsmechanismus wird ausgelöst und die Rakete planmäßig gesprengt.



Folie 8

## 3.1. Fehlerursachen (1)

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

- ◆ **Primäre Fehlerursache:** Operandenfehler bei Konvertierung der Variablen horizontal\_bias, Fehlen von Ausnahmebehandlungsroutinen (**Programmierfehler**).
- ◆ Nur für 4 von 7 Variablen waren die int- Umwandlungen geschützt, um die maximale Prozessorauslastung von 80% nicht zu überschreiten (**Kostengründe**).
- ◆ Für die 3 ungeschützten Variablen waren Wertebereiche angenommen worden, aber nicht dokumentiert (**verteilte Verantwortlichkeit**).
- ◆ Die Annahmen waren nicht an Hand der geplanten Flugbahn verifizierbar, da diese nicht zur Anforderungsspezifikation gehörte (**Management**).

## 3.1. Fehlerursachen (2)

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

- ◆ Die Default-Einstellung bei einer Ausnahme war es, den Prozessor abzuschalten statt „so gut wie möglich“ weiterlaufen zu lassen. Die Fehlertoleranzmechanismen gingen ausschließlich von zufälligen (Hardware-), nicht von systematischen (Software-) Fehlern aus (**Branchenkultur**).
- ◆ Die Sensorkalibrierungsroutine wird nach dem Start nicht mehr benötigt und hätte ausgeschaltet werden können (nur bei Startabbruch der Ariane 4 sinnvoll, um Neukalibrierung zu vermeiden) (**Wiederverwendung**).
- ◆ Die Grundannahme dass es nicht sinnvoll ist, funktionierende Software der Ariane 4 zu verändern, war falsch. **Die Grundannahme, dass Software zuverlässig ist falls keine Fehler offenbar sind, war falsch.**
- ◆ Es gab keinen **Review**, bei dem die Entscheidung, den Systemtest auszulassen, kritisch hinterfragt worden wäre.

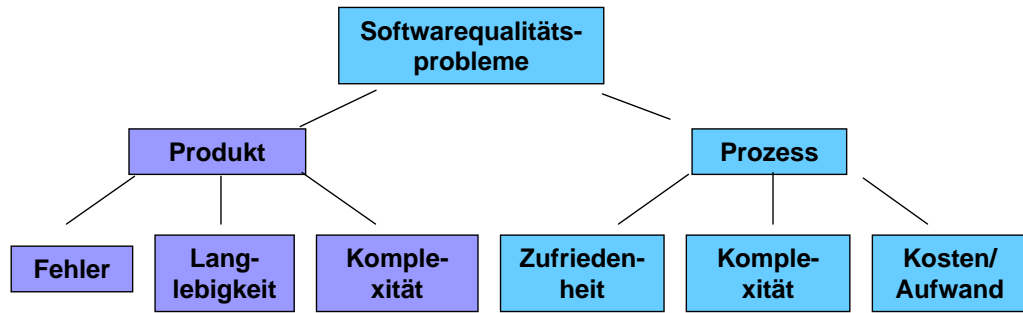
## 3.1. Typische Qualitätsprobleme

3. Qualitätssicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponententest



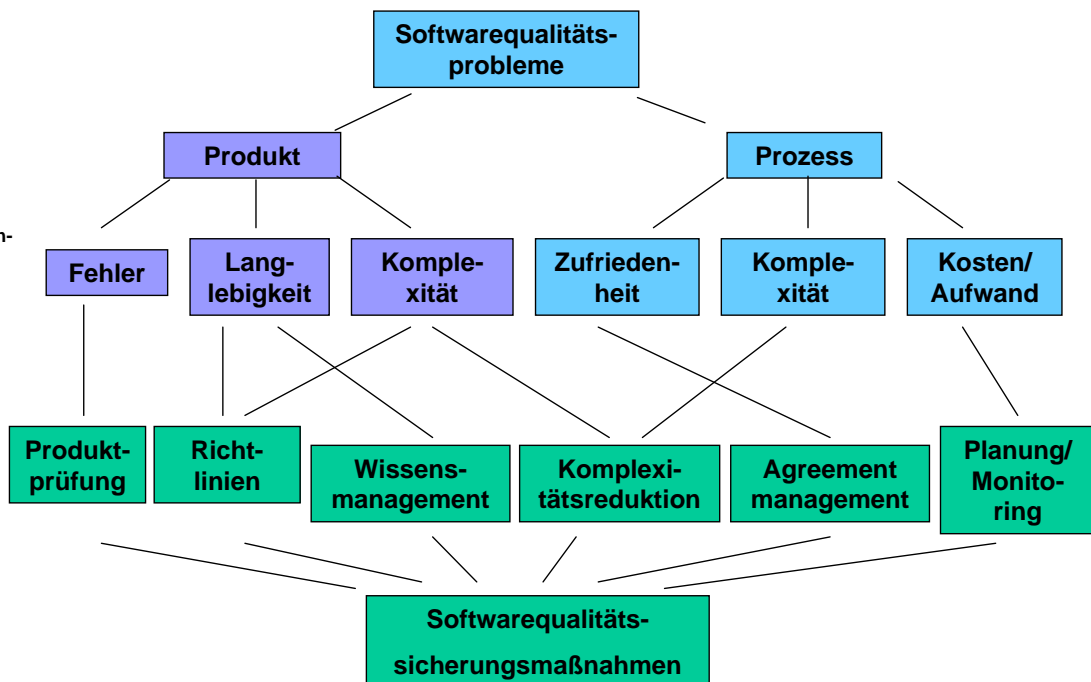
## 3.1. Typische QS-Maßnahmen

3. Qualitätssicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponententest



## 3.1. QS-Maßnahmen ARIANE 501 (1)

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

### ◆ Planung/Monitoring

- **Risikomanagement:** Die *Risiken* erkennen, angemessene technische Maßnahmen planen, durchsetzen und überprüfen.
- **Kostenmanagement:** Die Kosten einer vorbeugenden Maßnahme *in Relation* zu den Kosten eines Fehlers sehen

### ◆ Wissensmanagement

- **Wiederverwendung:** Bestehende Software darf nicht unbesehen für eine neue Aufgabe *wiederverwendet* werden. Vorher muss geprüft werden, ob ihre Fähigkeiten den Anforderungen der neuen Aufgabe entsprechen.
- **Spezifikation:** Die *Fähigkeiten* einer Software sowie alle *Annahmen*, die sie über ihre Umgebung macht, müssen sauber *spezifiziert* sein.
- **Dokumentation:** Kooperieren zwei Software-Komponenten miteinander, so müssen eindeutige *Zusammenarbeitsregeln* definiert, dokumentiert und eingehalten werden: Wer liefert wem was unter welchen Bedingungen.

## 3.1. QS-Maßnahmen ARIANE 501 (2)

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

### ◆ Produktprüfung /Richtlinien

- **Fehlerbehandlung:** Jede potentielle *Fehlersituation* in einer Software muss entweder *behandelt* werden oder die Gründe für die Nichtbehandlung müssen so *dokumentiert* werden, dass die Gültigkeit der dabei getroffenen Annahmen überprüfbar ist.
- **Fehlertoleranz:** Mehrfache identische Auslegung von Systemen hilft nicht gegen Entwurfsfehler.
- **Sicherer Zustand:** Bei Störungen in sicherheitskritischen Systemen ist *Abschalten* nur dann eine zulässige Maßnahme, wenn dadurch wieder ein sicherer Zustand erreicht wird.
- **Systemtest:** Beim *Test* von Software, die aus mehreren Komponenten besteht, genügt es *nicht*, jede Komponente *nur isoliert* für sich zu testen. Umfangreiche Systemtests unter möglichst realistischen Bedingungen sind notwendig.
- **Review:** Jedes Programm muss - neben einem sorgfältigen Test - durch kompetente Fachleute *inspiziert* werden, weil insbesondere die Erfüllbarkeit und Adäquatheit von Annahmen und Ergebnissen häufig nicht testbar ist.

## 3.1. Wie stellt man Qualität sicher?

- ◆ **Qualitätsmanagement:** etablierter Prozess zum Umgang mit Qualität
  - Wie kann man Qualität beschreiben
  - Wie kann man Qualität einer Software bewerten
  - Wie kann man Qualität erreichen
  - Wie kann man Mängel finden
  - Wie kann man Mängel beheben
  - Wie kann man Mängel vermeiden
- ◆ **Qualitätssicherung (QS):** konkretes Vorgehen zur Sicherstellung der Qualität
  - Konstruktive QS
  - Analytische QS

## 3.1. Konstruktive Produkt-QS

- ◆ Versucht die Qualität **durch systematische Entwicklung** sicherzustellen
  - Richtlinien
  - Templates
  - Werkzeuge
  - Notationen
  - Methoden
  - Ausbildung
- ◆ Entstehen meist aus Erfahrung mit analytischer QS

## 3.1. Analytische Produkt-QS

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompen-  
tentest

- ◆ **Ziel:** Überprüfen, ob Programm die Aufgabenspezifikation erfüllt
- ◆ **Prüfmethoden:**
  - **Beweis** (mathematischer Nachweis, nur bei einfachen Programmen händisch möglich, ansonsten komplexe Beweis-Tools)
  - **Test** (Ausprobieren für eine sorgfältig ausgewählte Menge von Eingaben)
  - **Inspektion** (systematisches Durchlesen)
  - **Metriken** (automatische Bestimmung von Charakteristika)

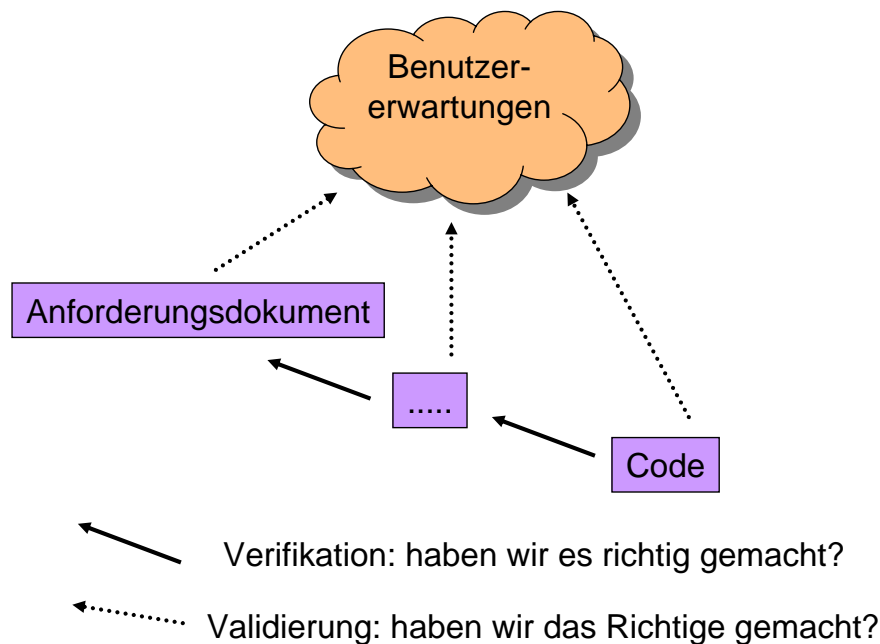
## 3.1. Validierung vs. Verifikation

3. Qualitäts-  
sicherung

3.1. Einführung

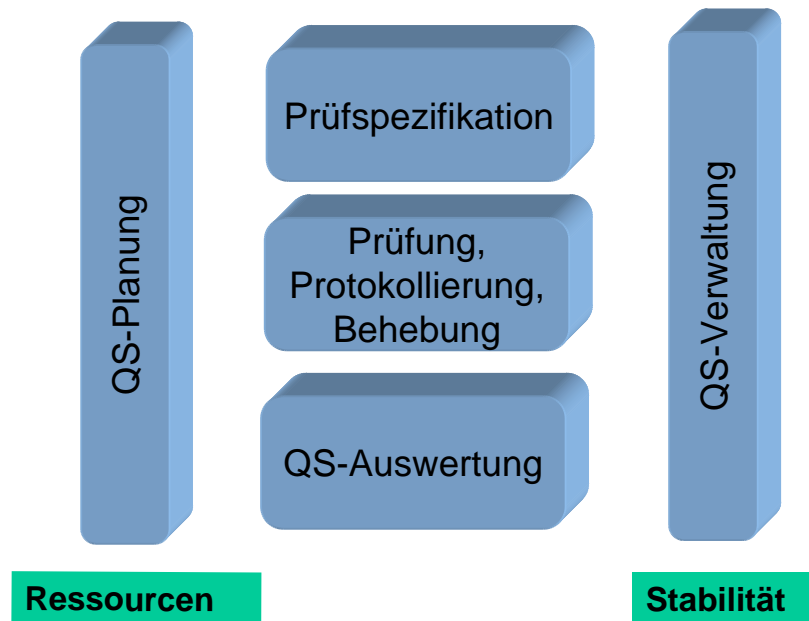
3.2. Testen

3.2.1. Kompen-  
tentest



## 3.1. analytische QS-Aufgaben

3. Qualitäts-  
sicherung
- ▶ 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Kompo-  
nententest



## 3.1. QS-Rollen und ihre Verantwortlichkeiten

3. Qualitäts-  
sicherung
- ▶ 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Kompo-  
nententest

- ◆ **QS-Manager:** QS-Planung und -auswertung (Was wird wann wie intensiv geprüft?)
- ◆ **QS-Designer:** Prüfmethode und Prüfspezifikation (Wie wird geprüft?)
- ◆ **Prüfer:** Prüfung, Protokollierung (Was sind die Prüfergebnisse?)
- ◆ **Prüfautomatisierer:** Prüfwerkzeuge (Wie werden Werkzeuge eingesetzt?)
- ◆ **QS-Administrator:** QS-Verwaltung (Wie werden Prüfdaten und - ergebnisse verwaltet?)

## 3.1. analytische QS-Planung

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test

- ◆ Ziel: Festlegung einer **Prüfstrategie**
- ◆ Prüfstrategie definiert **Prüfmethoden** und **Prüfaufwand** aufbauend auf einer **Kosten/Risiko-**Abschätzung (relativ zu **Qualitätszielen**)
  - **Kosten: Direkte Fehlerkosten** (beim Kunden durch Fehlerwirkung entstehende Kosten) + **Indirekte Fehlerkosten** (beim Hersteller durch Kundenunzufriedenheit entstehende Kosten) + **Fehlerkorrekturkosten** (beim Hersteller durch Fehlerkorrektur entstehende Kosten)
  - **Risiken:** Reifegrad des Entwicklungsprozesses, Prüfbarkeit der Software (Dokumentation, Komplexität, Systemumgebung), Mitarbeiterqualifikation

## 3.1. Details der Prüfstrategie

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test

- ◆ **Pro Dokument und Entwicklungsergebnis**
  - Festlegung der Prüfmethode
  - Festlegung des Abdeckungsgrades
- ◆ **Priorisierung der einzelnen Prüfungen**
  - **Eintrittswahrscheinlichkeit eines Fehlers** (z.B. häufig vom Kunden genutztes Prüfobjekt, intern kritisches Prüfobjekt, komplexes Prüfobjekt)
  - **Fehlerschwere** (Für Kunden: Schaden beim Auftreten bzw. Zufriedenheit; Für Hersteller: Größe des Fehlerbehebungsaufwandes)
  - **Priorität der Anforderungen**

### ◆ QS-Konzept [IEEE 829-1998]

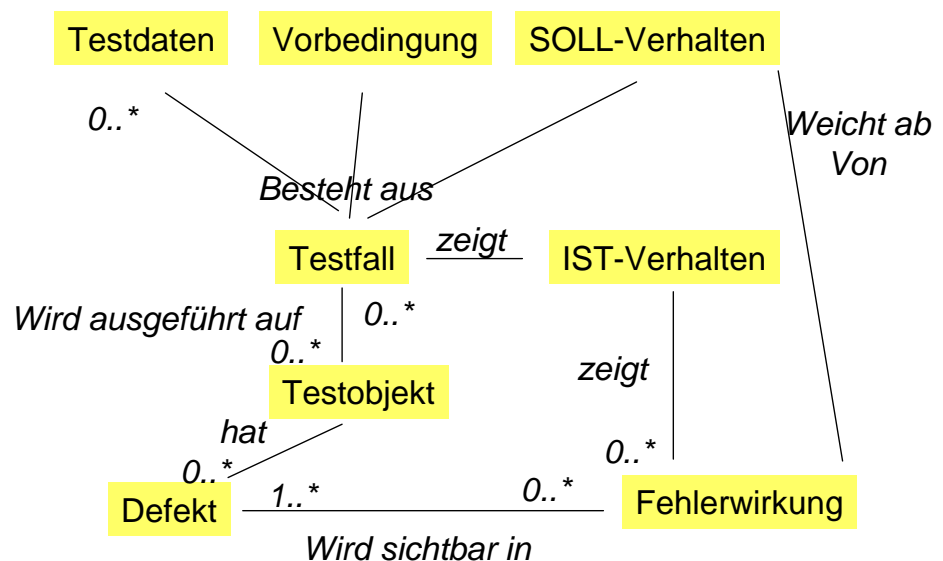
- Einführung
- Qualitätsziele (z.B. zu erreichende Zuverlässigkeit)
- Prüfobjekte und Festlegung, welche Funktionen und Eigenschaften geprüft werden
- Prüfstrategien
- Kriterien für Prüfungsabnahme, Prüfungsabbruch, Prüfungsfortsetzung
- Prüfungsdokumentation
- Prüfumgebung
- Prüfaufgaben und Verantwortliche
- Zeitplanung, Risiken
- Freigabe des QS-Konzepts

- ◆ 3.2.1. Komponententest
- ◆ 3.2.2. Systemtest
- ◆ 3.2.3. Usabilitytest
- ◆ 3.2.4. Integrationstest

## 3.2. Was ist ein Test?

- ◆ Ein **Fehler** ist die **Nichterfüllung einer Anforderung**. Er wird deutlich durch die Abweichung zwischen dem IST-Verhalten und dem SOLL-Verhalten. Genauer ist zu unterscheiden zwischen **Defekt (Fehlerursache)** und **Fehlerwirkung**.
- ◆ **Debugging**: Lokalisieren und Beheben eines Defekts
- ◆ **Testen**: systematische Aufdeckung von Fehlerwirkungen durch Ausführung der Software

## 3.2. Begriffe im Umfeld Testen

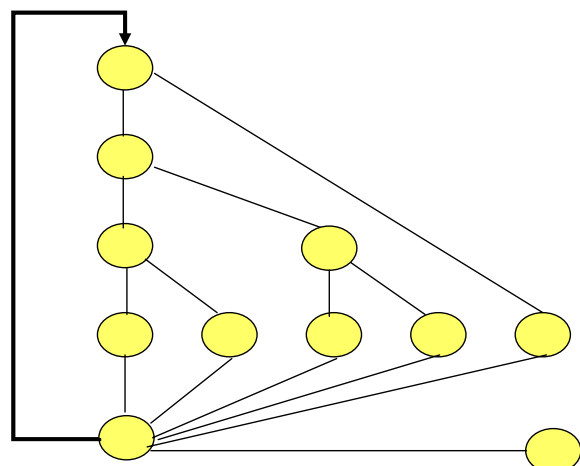


## 3.2. Testen ist schwierig

- ◆ Hoher Aufwand, schlecht zu fokussieren
  - Anzahl Testläufe
  - Testvorbereitung (Stubs, Testorakel)

## 3.2. Übung: Testaufwand

- ◆ Programm:



- ◆ Wieviele Abläufe sind zu testen für einen vollständigen Test bei maximal  $x$  Schleifendurchläufen?
- ◆ Wieviele sind es für  $x = 20$ ?

## 3.2. Testdurchführung

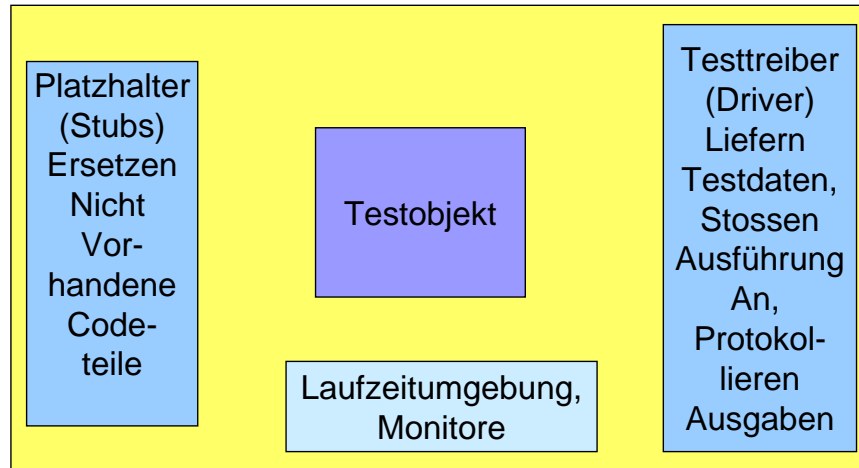
- ◆ Falls Code nicht allein ablauffähig (z.B. Funktion, Klasse), muss **Testrahmen** bereitgestellt werden

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test



## 3.2. Was ist eine Testspezifikation?

- ◆ Ziel: Definition der **Testobjekte und ihrer Testfälle** (unter Verwendung der festgelegten Testmethoden) sowie der **Testendekriterien**
  - **Logische Testfälle** (Wertebereich für Ein/Ausgabe)
  - **Konkrete Testfälle** (spezifische Ein/Ausgaben)
- ◆ Testfalldefinition: Festlegung des SOLL-Verhaltens anhand eines **Testorakels**
  - Anforderungsspezifikation
  - Benutzungshandbuch
  - Ausführbarer Prototyp (formale Spezifikation)
  - Alte Versionen

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test

## 3.2. Testfallspezifikation in REQuest

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

- ◆ Einleitung
- ◆ Endekriterien für Gesamttest bzw. Testobjekt
- ◆ Ressourcen
- ◆ Testfallbeschreibung
  - Typ (Komponenten, Integration, System...)
  - Ziel (Testobjekt, Problem, Testmethode)
  - Vorbedingung (Testumgebung einrichten)
  - Schritte (Eingabe, erwartete Ausgabe, erwartete Ausnahmen, Testabbruch)
  - Nachbedingung (Testumgebung abbauen)
  - Ressourcen (Werkzeuge)
  - Testendekriterium für Testfall

## 3.2. Welche typischen Testmethoden gibt es ?

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Kompo-  
nententest

- ◆ **Black-Box-Verfahren:**
  - Testen die **Außenwirkung** des Testobjekts
  - Keine Steuerung des Ablaufs des Testobjekts
  - Nutzen Kenntnisse über die Schnittstelle
  - Beispiele: Äquivalenzklassen, Grenzwertbezogen, Zustandsbezogen
- ◆ **White-Box-Verfahren:**
  - Testen gezielt die **verschiedenen Abläufe** des Testobjekts
  - Nutzen Kenntnisse über den inneren Aufbau (Code)
  - Beispiele: Überdeckung der Anweisungen, der Zweige, der Bedingungen, der Pfade

### ◆ Positiv-Test:

- korrekte Eingaben (erwartet korrekte Ergebnisse)

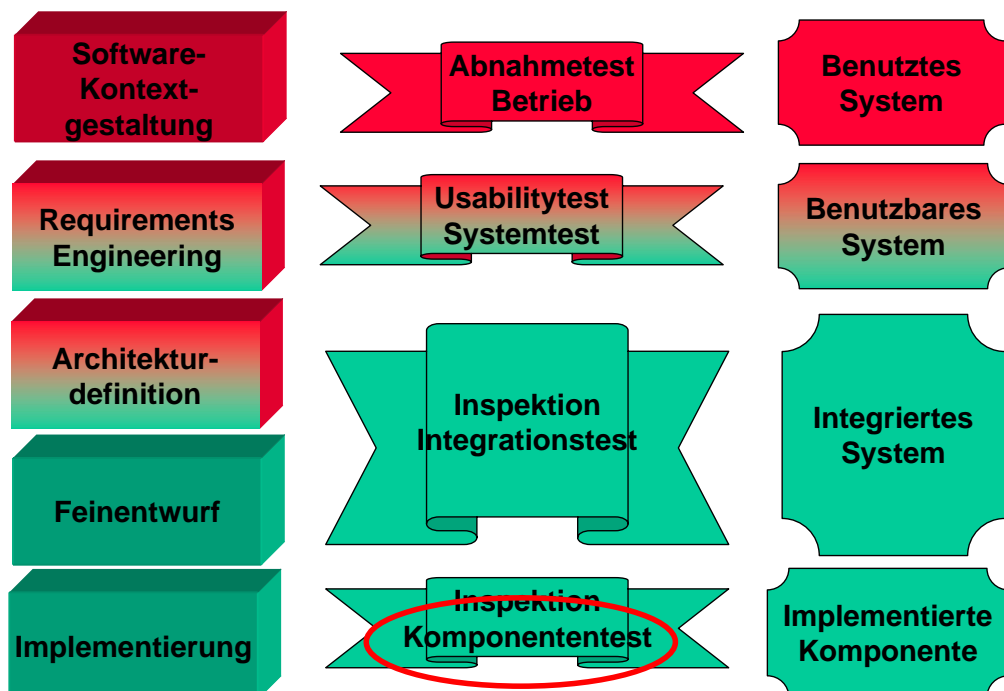
### ◆ Negativ-Test:

- unzulässige Eingaben (erwartet sinnvolle Ausnahmebehandlung)

### ◆ Intuitiver Test:

- Beruht auf Kenntnis typischer Fehlersituationen
- Sollte immer zusätzlich zu systematischeren Verfahren durchgeführt werden

## 1.3.1. Aktivitäten und Ergebnisse der Entwicklung und Qualitätssicherung





- ◆ **Komponente** = abgeschlossene Code-Einheit, z.B. Klasse, Funktion, Modul
- ◆ White-box und Black-box möglich
- ◆ **Typische Fehlverhalten:**
  - Nichtterminierung
  - Falsches oder fehlendes Ergebnis
  - Unerwartete oder falsche Fehlermeldung
  - Inkonsistenter Speicherzustand
  - Unnötige Ressourcenbelastung
  - Unerwartetes Ausnahmeverhalten (z.B. Absturz)



- ◆ 3.2.1.1. Black-box Verfahren
- ◆ 3.2.1.2. White-box Verfahren



- ◆ Testen ablauffähige Einheit, z.B. Operation
- ◆ Typische Testmethoden
  - Intuitiv (Erfahrung) bzw. Zufall
  - Äquivalenzklassenbildung, insbes. Grenzwertanalyse
  - Zustandsbezogener Test
- ◆ Kann NICHT unnötige Programmteile erkennen



- ◆ **Äquivalenzklasse:** Teilmenge der möglichen Eingabewerte
- ◆ **Annahme:** Programm reagiert für alle Werte aus der Äquivalenzklasse prinzipiell gleich
- ◆ Testfälle decken alle Äquivalenzklassen ab: **mindestens ein Vertreter pro Klasse**
- ◆ **Grenzwerte:** falls Werte in der Äquivalenzklasse geordnet, teste an jedem Rand den exakten Grenzwert, sowie die beiden benachbarten Werte (innerhalb bzw. außerhalb)
- ◆ **Typische Äquivalenzklassen**
  - Zulässige/unzulässige Datenbereiche (insbesondere bei komplexen Formaten)
  - Grenzwerte
  - Unterteilung nach verschiedenen Ausgabewerten (ggf. auch Äquivalenzklassen der Ausgabewerte)

## 3.2.1.1. Äquivalenzklassentest (2)

- ◆ Testfallableitung bei **mehreren** Eingaben:
  - Kombiniere alle gültigen Äquivalenzklassen der verschiedenen Eingaben
  - Kombiniere jede ungültige Äquivalenzklasse mit anderen gültigen Äquivalenzklassen
- ◆ Vereinfachung:
  - Nur häufige Kombinationen
  - Nur Testfälle mit Grenzwerten
  - Nur Abdeckung paarweiser Kombinationen
- ◆ Minimal:
  - Jeder Repräsentant kommt in einem Testfall vor

## 3.2.1.1. Übung: Äquivalenzklassen (1)

- ◆ Funktion:
 

```
int search (int[] a, int k)
pre: a.length > 0
post: (result >= 0 and a[result] ==k) or (result == -1 and (not
exists i. i>=0 and i< a.length and a[i] == k))
```
- ◆ Gültige Äquivalenzklassen:
  - Param1:  $a.length > 0$ ,
  - Param2:  $k$  in  $a$ ,  $k$  nicht in  $a$
  - Verfeinerung von Param1: a)  $a.length = 1$ , c)  $a.length > 1$
  - Verfeinerung von Param2: a)  $k$  ist erstes, b) mit  $k$  gleiches Element oder c) letztes Element von  $a$  oder d)  $k$  nicht in  $a$
- ◆ Ungültige Äquivalenzklasse:
  - Param1:  $a.length = 0$ , (auch:  $R$  ... als Werte...)

Wieviele Testfälle nötig?

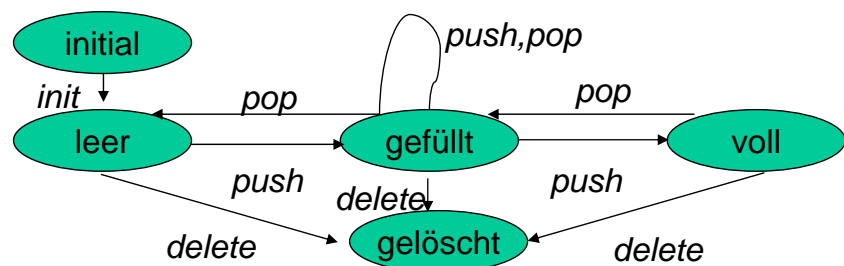
## 3.2.1.1. Übung: Äquivalenzklassen (2)

◆ Testfälle:

Feld a	Element k	Ergebnis (result)
[]	x	ungültig
[x]	x (x in a)	1
[y]	x (x not in a)	-1
[...,x,...]	x (x Mitte)	n
[...]	x (x not in a)	-1
[x,...]	x (x Anfang))	1
[...,x]	x (x Ende)	n

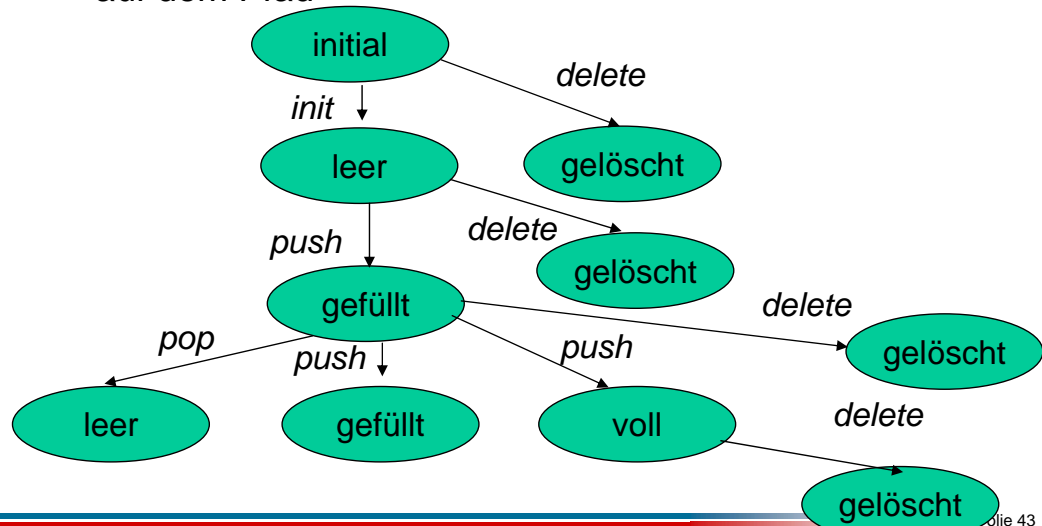
## 3.2.1.1. Zustandsbezogener Test (1)

- ◆ Berücksichtigt neben Ein/Ausgaben auch die **Historie** (der erreichte Zustand)
- ◆ Typisches Beispiel: Stapel  
Zustände: initial, leer, gefüllt, voll, gelöscht
- ◆ Testenkriterium: alle Zustände und alle darin zulässigen Funktionen (alle Zweige) einmal abgedeckt



## 3.2.1.1. Zustandsbezogener Test (2)

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



## 3.2.1.2. White-Box Test

- ◆ Anweisungsüberdeckung
- ◆ Zweigüberdeckung (auch mit Bedingungsüberdeckung)
- ◆ Pfadüberdeckung
- ◆ Benötigt Kontrollflussgraph
- ◆ **Testendekriterium** ist Grad der Abdeckung
- ◆ Vor allem geeignet für Komponententest
- ◆ Kann **übersehene Anforderungen** NICHT aufdecken

## 3.2.1.2. Beispiel: Binäre Suche

//Annahme : a ist sortiert

- 3. Qualitäts-sicherung
  - 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Komponenten-test

```
int binsearch ( int [] a, int k)
{
    int bottom = 0 ;
    int top = a.length - 1 ;
    int mid ;
    result.found = false ; result.index = -1 ;

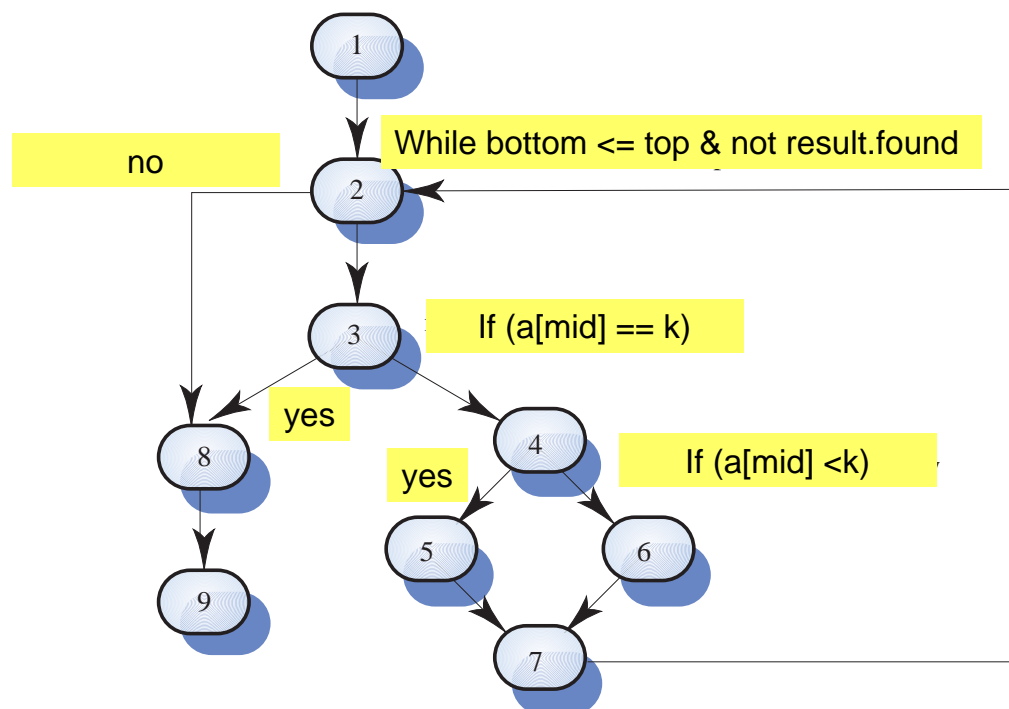
    while ( bottom <= top ) and not result.found
        mid = (top + bottom) / 2 ;
        if (a[mid] == k)
        {
            result.index = mid ;
            result.found = true ;
        } // if part
        else
        {if (a [mid] < k)
            bottom = mid + 1 ;

            else
                top = mid - 1 ;
        }//else part;
    return result.index
} // binsearch      .....
```

Folie 45

## 3.2.1.2. Binäre Suche Kontrollflußgraph

- 3. Qualitäts-sicherung
  - 3.1. Einführung
  - 3.2. Testen
  - 3.2.1. Komponenten-test



Folie 46

## 3.2.1.2. Anweisungsüberdeckung

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test

- ◆ **Knoten** im Kontrollflussgraph repräsentieren (Sequenzen von) Anweisungen
- ◆ 100% -Überdeckung schwer, wenn Ausnahmefälle schwer herzustellen
- ◆ Beispiel Binäre Suche:
  - 100%-Überdeckung mit Reihenfolge 1,2,3,4,5,7,2,3,4,6,7,2,3,8,9 (auch als 2 getrennte Sequenzen möglich)
  - Welche Testeingaben nötig, um diese Sequenz zu erreichen?

Folie 47

## 3.2.1.2. Zweigüberdeckung

3. Qualitäts-  
sicherung

3.1. Einführung

3.2. Testen

3.2.1. Komponenten-  
test

- ◆ **Kanten** im Kontrollflussgraph repräsentieren Zweige
- ◆ 100%-Zweigüberdeckung sollte erreicht werden
- ◆ Beispiel Binäre Suche:
  - Anweisungsüberdeckung 1,2,3,4,5,7,2,3,4,6,7,2,3,8,9 führt Kante 2-8 nicht aus
  - Hinzunahme von 1,2,8,9 als weiteren Testfall nötig
- ◆ Für **komplexe Bedingungen** sollte mindestens jede Teilbedingung einmal überdeckt werden, besser alle Kombinationen

Folie 48



- ◆ **Pfade** im Kontrollflussgraph entstehen durch Verbindung von Zweigen
- ◆ 100%-Pfadüberdeckung nicht realistisch,
- ◆ Beispiel Binäre Suche:
  - Anweisungsüberdeckung 1,2,3,4,5,7,2,3,4,6,7,2,3,8,9 und Zweigüberdeckung 1,2,8,9, führt Pfad mit mehr als 3 Schleifendurchgängen nicht aus
  - Durch Vorgabe der Anzahl von Schleifenwiederholungen machbar

- ◆ Wird in Prozent formuliert, z.B.
  - 100% Zweigabdeckung
  - 90% Anweisungsabdeckung, d.h. 90% der Anweisungen sind durch Testfälle zu überprüfen.



- ◆ A. Spillner, T.Linz: Basiswissen Softwaretest, dpunkt Verlag, 2002
- ◆ IEEE 829-1998, Standard for Software Test Documentation
- ◆ I. Sommerville, Software Engineering, Pearson Studium, 2001

### 3.2.1. Zusammenfassung Komponententesten



- ◆ Testen muss von Anfang eines Projektes an geplant und vorbereitet werden
- ◆ Es gibt viele Testmethoden: Herausforderung ist eine minimale Testfallmenge für eine angestrebte Qualität zu definieren