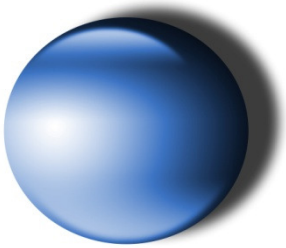
Software Engineering

RWTH AACHEN



Einführung in die Softwaretechnik

Vorlesung, Teil 2

Kapitel 8. Qualitätsmanagement: Testen

Prof. Dr. Bernhard Rumpe
Lehrstuhl Informatik 3 (Software Engineering)
Rheinisch-Westfälische Technische Hochschule Aachen

http://www.se-rwth.de/

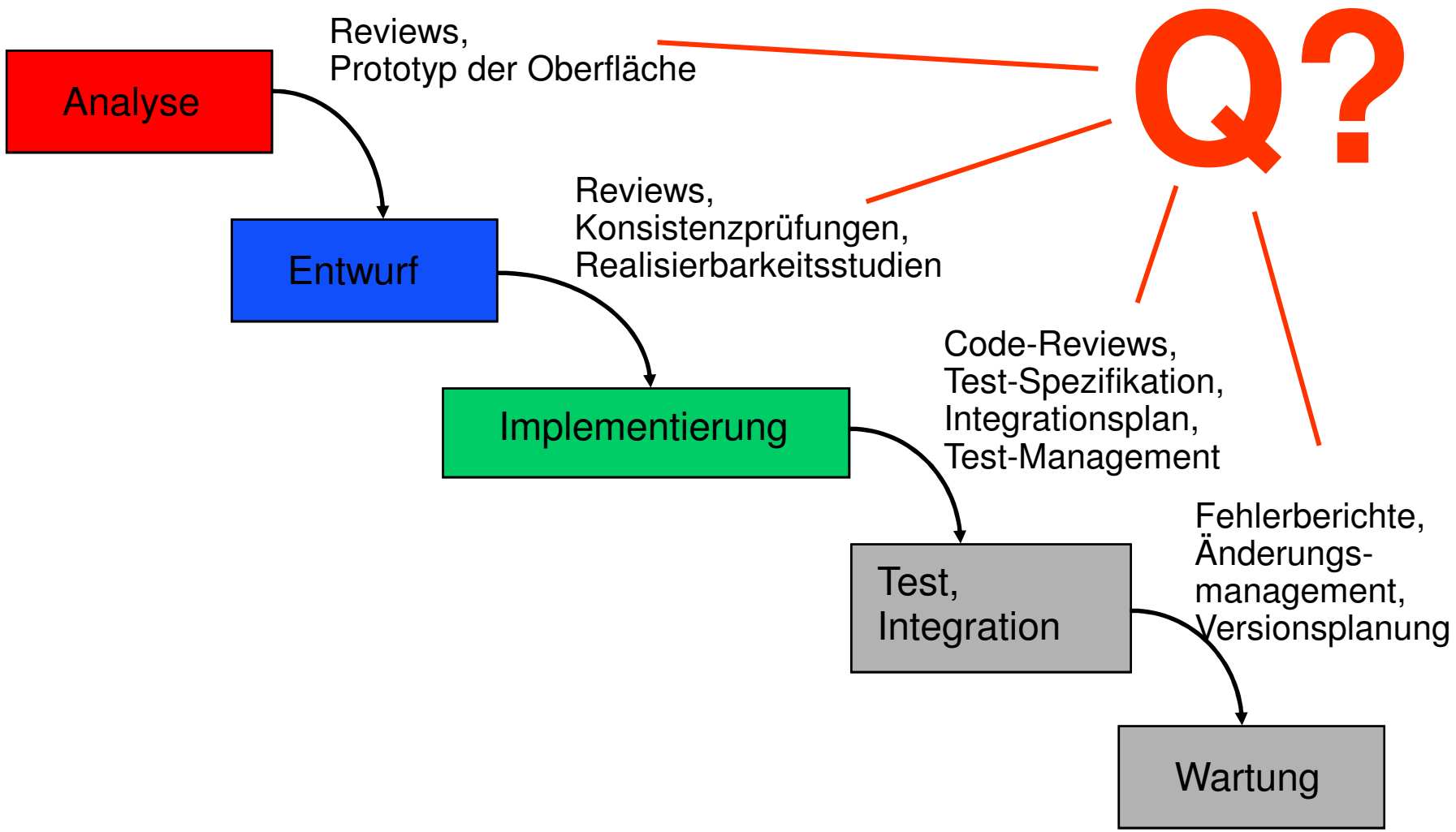
8. Qualitätsmanagement
8.1. Prozessqualität

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Literatur:
- Sommerville 24-25
- Balzert Band II, LE 12-13

Prozessintegrierte Qualitätssicherung



Qualitätssicherung für Analyse und Entwurf

- Hohe Bedeutung früher Phasen für Produktqualität !
  - Deshalb: Alle Dokumente frühzeitig überprüfen ("Validation").
- Techniken:
  - Anforderungskatalog-Begutachtung
    - Echte Benutzer einbeziehen
    - Anforderungskatalog auf Vollständigkeit und Korrektheit prüfen
  - Use-Case-Szenarien
    - Echte Benutzer einbeziehen
    - "Funktionsfähigkeit" der abstrakten Modelle demonstrieren
  - Prototyping
    - Prototyp auf der Basis der Analyse/Entwurfs-Dokumente
    - Echte Benutzer einbeziehen
    - Vorgezogener Akzeptanztest
  - Abgleich des Entwurfs mit Use-Cases/Anforderungskatalog
    - Erste verifizierende Tätigkeiten

Begutachtung (Review)

- Produkt wird von Expertengremium begutachtet
  - Anwendbar auf fast alle Phasen der Entwicklung
- Was wird begutachtet?
  - Genau definiertes Dokument (Art, Status, Prozesseinbindung)
  - Teil der Gesamtplanung des Projekts (Termin)
- Wer begutachtet?
  - Teammitglieder (Peer-Review)
  - Externe Spezialisten
  - Echte Benutzer
  - Moderator, Manager
- Wie läuft die Begutachtung ab?
  - Einladung
  - Vorbereitung, Sammlung von Kommentaren
  - Begutachtungssitzung(en), Protokolle
  - Auswertung und Konsequenzen (Wiederholung, Statusänderung)

Regeln für wirkungsvolle Begutachtung

- "Checklisten" für die Gutachter vorbereiten
  - z.B. "Enthält das Dokument alle laut Firmenstandard vorgesehenen Inormationen?" - "Gibt es für jede Klasse eine informelle Beschreibung?" - "Sind Kardinalitäten im Klassendiagramm vollständig und korrekt?" - "Sind alle Klassen kohärent?" - ...
- Das Dokument wird begutachtet, nicht die Autoren!
- Richtige Vorkenntnisse der Gutachter sind wesentlich.
- Die Rolle des Moderators ist anspruchsvoll:
  - Vermittlung in persönlichen Konflikten und Gruppenkonflikten
  - Fachlicher Gesamtüberblick
- Das Ziel ist Einigkeit, nicht ein Abstimmungsergebnis!
- Ergebnisse von Begutachtungen müssen Auswirkungen haben.
- Wesentliche Beiträge von Gutachtern müssen Anerkennung finden.
- Begutachtung ist auch anwendbar auf Programm-Code (Code-Inspektion).

Produktqualität und Prozessqualität

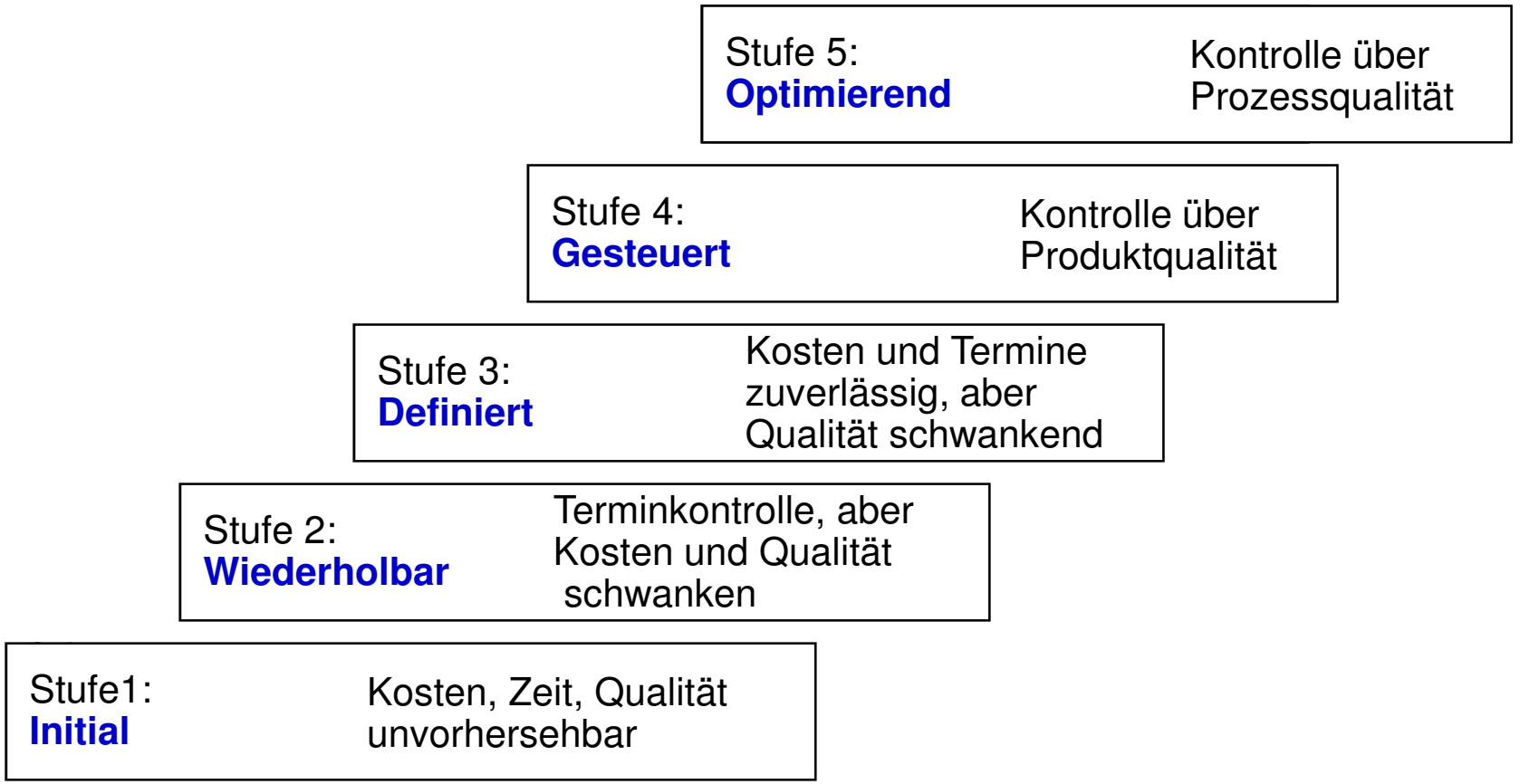
- Software:
  - Kaum Qualitätsmängel durch Massenproduktion
  - Qualitätsmängel im Herstellungsprozess begründet
- Qualitätsmanagement:
  - Organisatorische Maßnahmen zur Prüfung und Verbesserung der Prozessqualität
  - Beachtung von internen Kunden-/Lieferantenbeziehungen
- **Qualität des Entwicklungsprozesses!**
- Ansätze:
  - ISO 9000
  - Total Quality Management (TQM)
    - Kaizen: Ständige Verbesserung
  - Capability Maturity Model (CMM)
  - Software Process Improvement and Capability Determination (SPICE)
  - *Business (Process) Engineering*

ISO 9000

- Internationales Normenwerk
  - ISO 9000: Allgemeiner Leitfaden
    - ISO 9000-3: Leitfaden zur Anwendung von ISO 9001 auf Software
  - ISO 9001: Modelle zur Darlegung der Qualitätssicherung insbesondere in Entwurf und Entwicklung
- Allgemeingültige Forderungen an die Organisation eines Unternehmens:
  - Regelung von Zuständigkeiten
  - Erstellung und korrekte Verwaltung wesentlicher Dokumente
  - Existenz eines unabhängigen Qualitätssicherungs-Systems
- Zertifizierung:
  - durch unabhängige (zertifizierte) Prüforganisationen
  - Audit, Prüfsiegel
- ISO 9000 prüft nicht die Qualität des Produkts, sondern die Qualität der Organisation !

Capability Maturity Model (CMM)

- Einstufungsverfahren für Reifegrad der Organisation
  - entwickelt von der Carnegie-Mellon University



8. Qualitätsmanagement
8.2. Test und Integration

Analyse	Entwurf	Implementierung	**Test, Integration**	Wartung

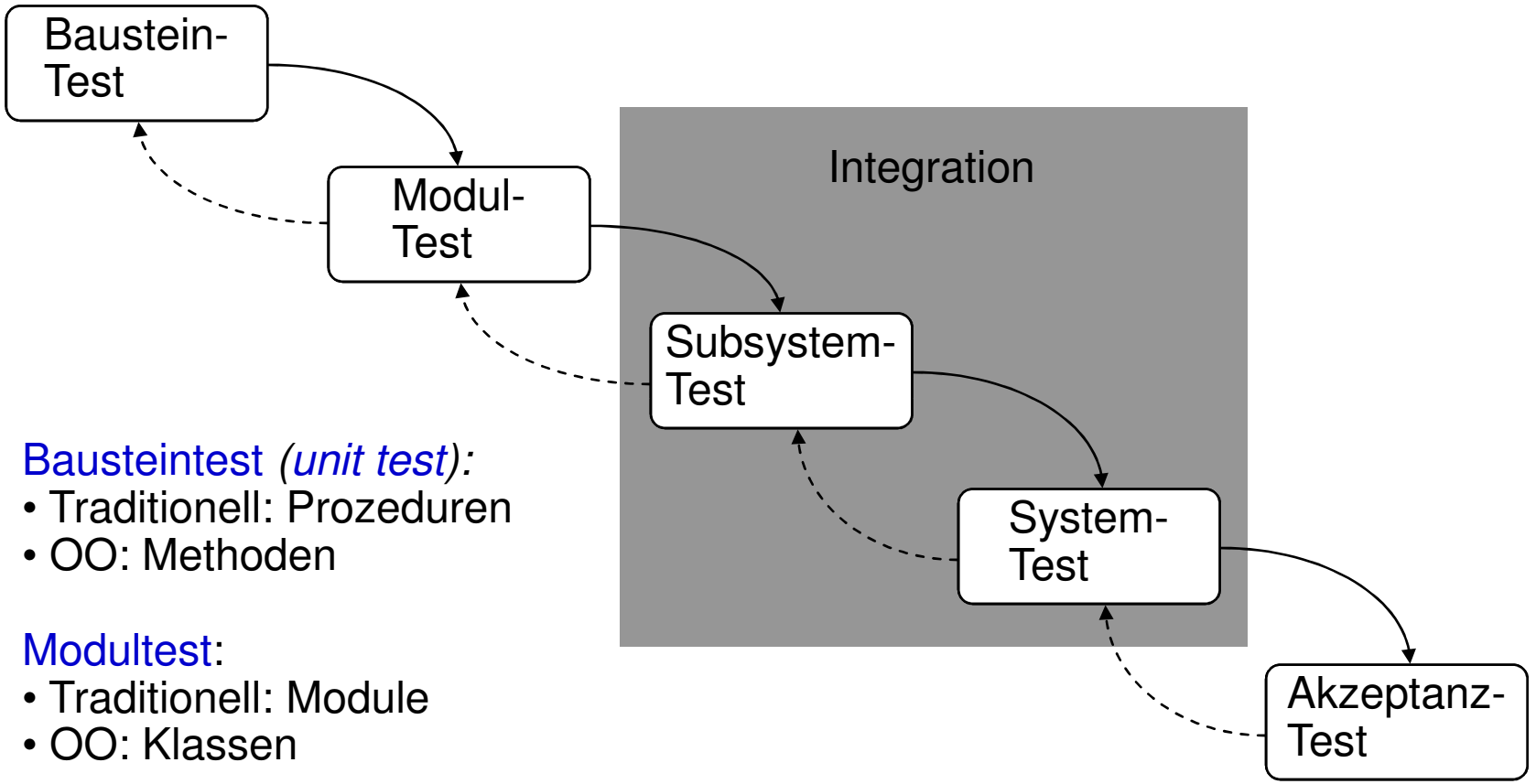
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Allg. Literatur:
- Sommerville 19-20
- Balzert Band 2, LE 14-15

Speziell:
- Robert V. Binder: Testing object-oriented systems, Addison-Wesley 2000

Integration und Test



Bausteintest *(unit test)*:
- Traditionell: Prozeduren
- OO: Methoden

Modultest:
- Traditionell: Module
- OO: Klassen

Subsystemtest:
- OO: Pakete

Manche Autoren:
Subsystemtest = Integration = "Integrationstest"

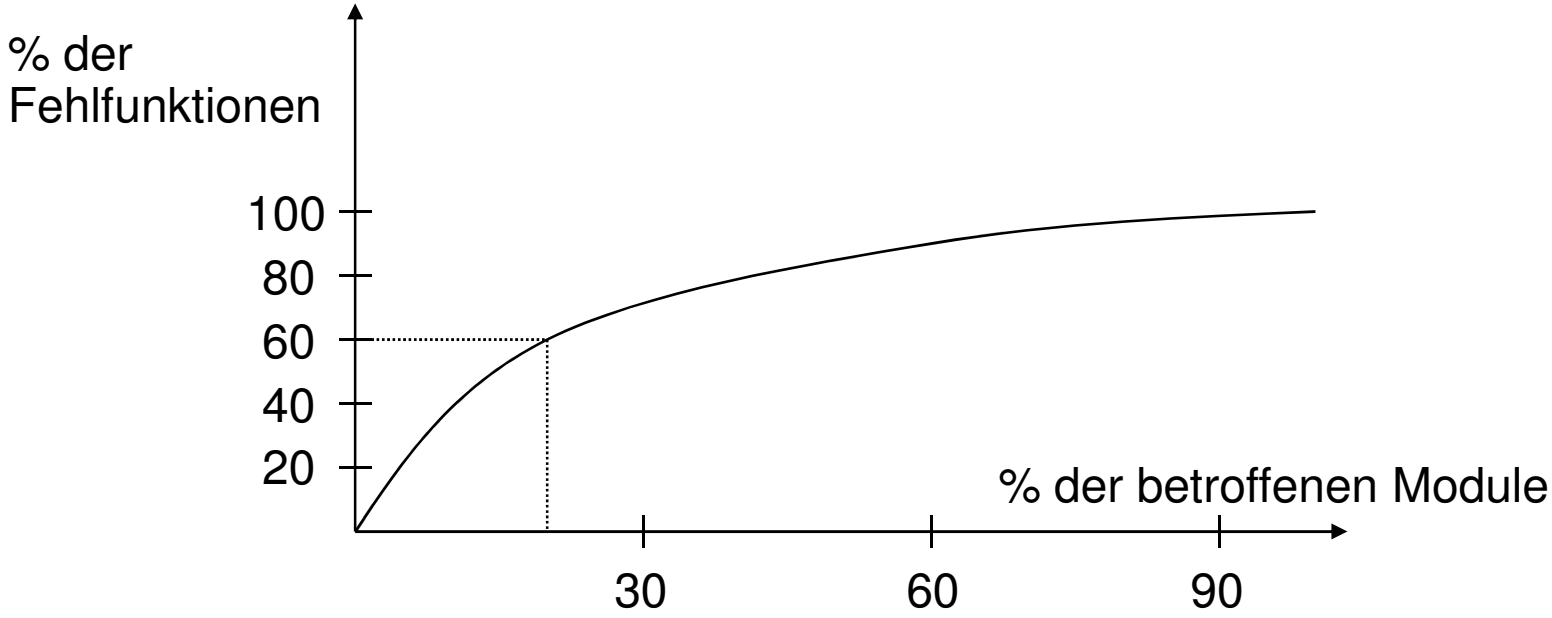
Systematisches Testen: Begriffe

- *Testgegenstand (Testling, Prüfling):*
  Programm bzw. Komponente, die zu testen ist
- *Testfall:*
  Satz von Eingabedaten, der die (vollständige) Ausführung des Testgegenstands bewirkt
- *Testdaten:*
  Daten, die für den Testfall benötigt werden
- *Testtreiber:*
  Rahmenprogramm für Ausführung von Tests
- *Attrappe (Stub, Dummy):*
  Ersatz für ein (noch) nicht vorhandenes Unterprogramm
- *Regressionstest:*
  Nachweis, dass eine geänderte Version des Testgegenstands früher durchgeführte Tests erneut besteht
- *Alphatest:* Test experimenteller Vorversion durch Benutzer
- *Betatest:* Test funktional vollständiger Software durch Benutzer

Fehlfunktionen

- Fehlfunktion *(fault):*
  Unerwartete Reaktion des Testlings
- Unterscheidung zwischen dem fehlerhaften Code (*error, bug*) und dem Auftreten des Fehler (*fault*):
  - Ein „fault“ kann aufgrund mehrerer „bugs“ auftreten.
  - Ein „bug“ kann mehrere „faults“ hervorrufen.
- Arten von Fehlfunktionen:
  - Falsches oder fehlendes Ergebnis
  - Nichtterminierung
  - Unerwartete oder falsche Fehlermeldung
  - Inkonsistenter Speicherzustand
  - Unnötige Ressourcenbelastung (z.B. von Speicher)
  - Unerwartetes Auslösen einer Ausnahme, "Abstürze"
  - Falsches Abfangen einer Ausnahme
- Testen kann nur die Anwesenheit von Fehlern nachweisen, nicht deren Abwesenheit!

Pareto-Prinzip



- Fenton/Ohlsson 2000 und andere empirische Untersuchungen:
  - 20 % der Module sind verantwortlich für 60 % der Fehler
  - Diese 20 % der Module entsprechen 30 % des Codes
- Testmuster: "Scratch'n sniff"
  - Fehlerhafte Stellen deuten oft auf weitere Fehler in der Nähe hin.

Testplanung

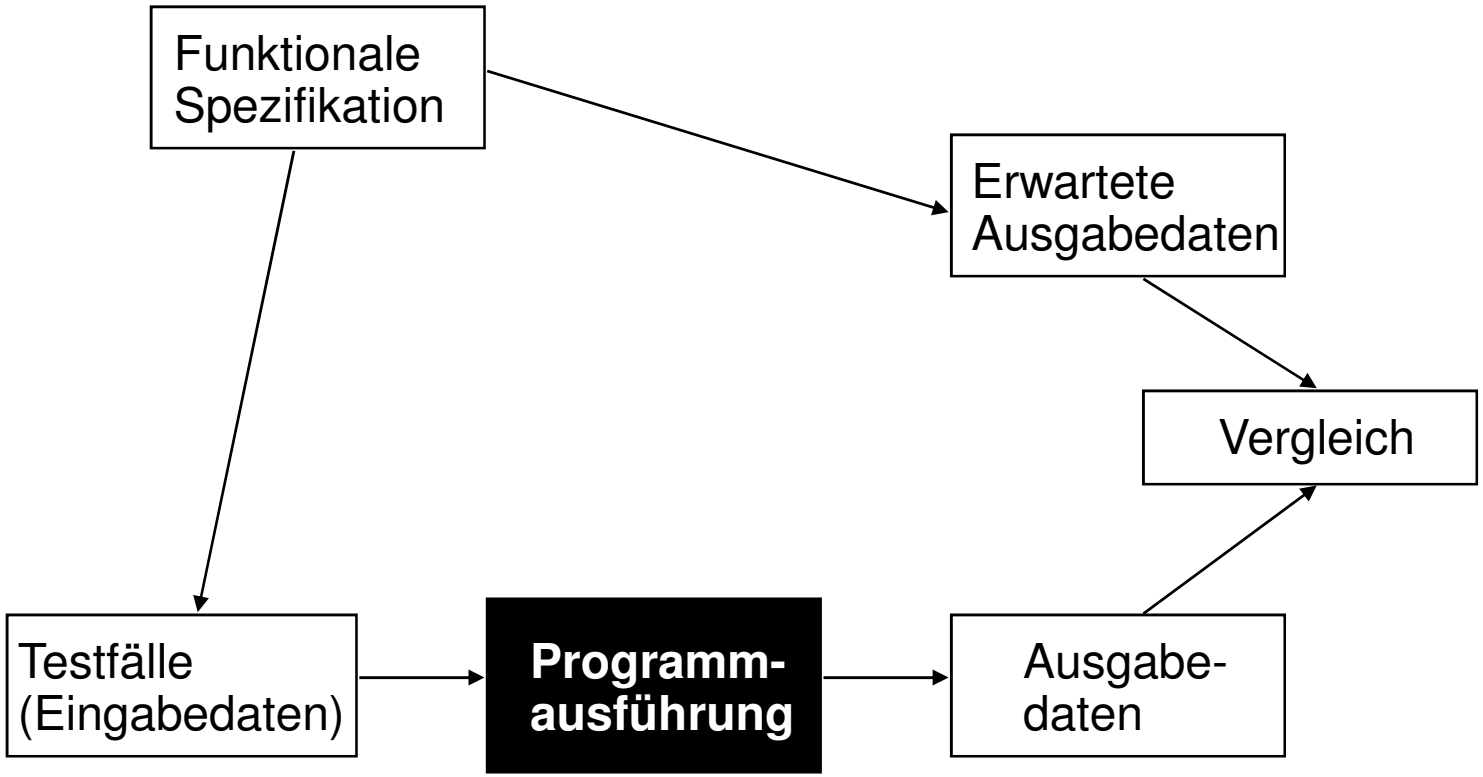
- Testen ist aufwändig, deshalb ist gute Planung nötig!
- Testplanung sollte bereits mit der Anforderungsanalyse beginnen und im Entwurf verfeinert werden (V-Modell, Test-First-Ansatz)!

- Typische Bestandteile einer *Test-Spezifikation:*
  - Phasenmodell des Testprozesses
  - Zusammenhang zur Anforderungsspezifikation
    - z.B. dort festgelegte Qualitätsziele
  - Zu testende Produkte
  - Zeitplan für die Tests
  - Abhängigkeiten der Testphasen
  - Aufzeichnung der Testergebnisse
  - Hardware- und Softwareanforderungen
  - Einschränkungen und Probleme

Klassifikation von Testverfahren

- Funktionaler Test *(black box test)*
  (Testfallauswahl beruht auf Spezifikation)
  - Äquivalenzklassentest, Grenzwerte, Test spezieller Werte
  - Zustandstest auf Spezifikationsbasis
- Strukturtest *(white box test, glass box test)*
  (Testfallauswahl beruht auf Programmstruktur)
  - Kontrollflussorientierter Test
    - Anweisungsüberdeckung, Zweigüberdeckung, Pfadüberdeckung
  - Datenflussorientierter Test
    - Definition/Verwendung von Variablen
- Weitere:
  - Zufallstest ("*monkey test*")
  - Stress-, Lasttest
  - etc.

Funktionaler Test *(black box)*



Äquivalenzklassen

- Äquivalenzklasse:
  - Teilmenge der möglichen Datenwerte der Eingabeparameter
  - Annahme: Programm reagiert für alle Werte aus der Äquivalenzklasse prinzipiell gleich
  - Test je eines Repräsentanten jeder Äquivalenzklasse

- Finden von Äquivalenzklassen:
  - Kriterien für Werte entwickeln und diese wo sinnvoll kombinieren
  - Zulässige und unzulässige Teilbereiche der Datenwerte
  - Unterteilung der zulässigen Bereiche nach verschiedenen Ausgabewerten

Äquivalenzklassen: Beispiel

```
int search (int[] a, int k)
```

Vorbedingung: a.length >= 0

Nachbedingung: $(result \geq 0 \wedge a[result] == k) \vee$
$(result == -1 \wedge (\neg\exists\ i\ .\ 0 \leq i < a.length \wedge a[i] == k))$

⇨ discuss

- Kriterien für Äquivalenzklassen:
  -
  -
  -
  -
- Äquivalenzklassen: / Testfälle:
  -
  -
  -
  -

Grenzwertanalyse und Test spezieller Werte

- Grenzwerte: Randfälle der Spezifikation
  - Werte, bei denen "gerade noch" ein gleichartiges Ergebnis zum Nachbarwert erzielt wird, bzw. "gerade schon" ein andersartiges
  - Beispiele:
    - Ränder von Zahl-Intervallen
    - Schwellenwerte

- Spezielle Werte:
  - Zahlenwert 0
  - Leere Felder, Sequenzen und Zeichenreihen
  - Einelementige Felder, Sequenzen und Zeichenreihen
  - Null-Referenzen
  - Sonderzeichen (Steuerzeichen, Tabulator)

Grenzwertanalyse: Beispiel

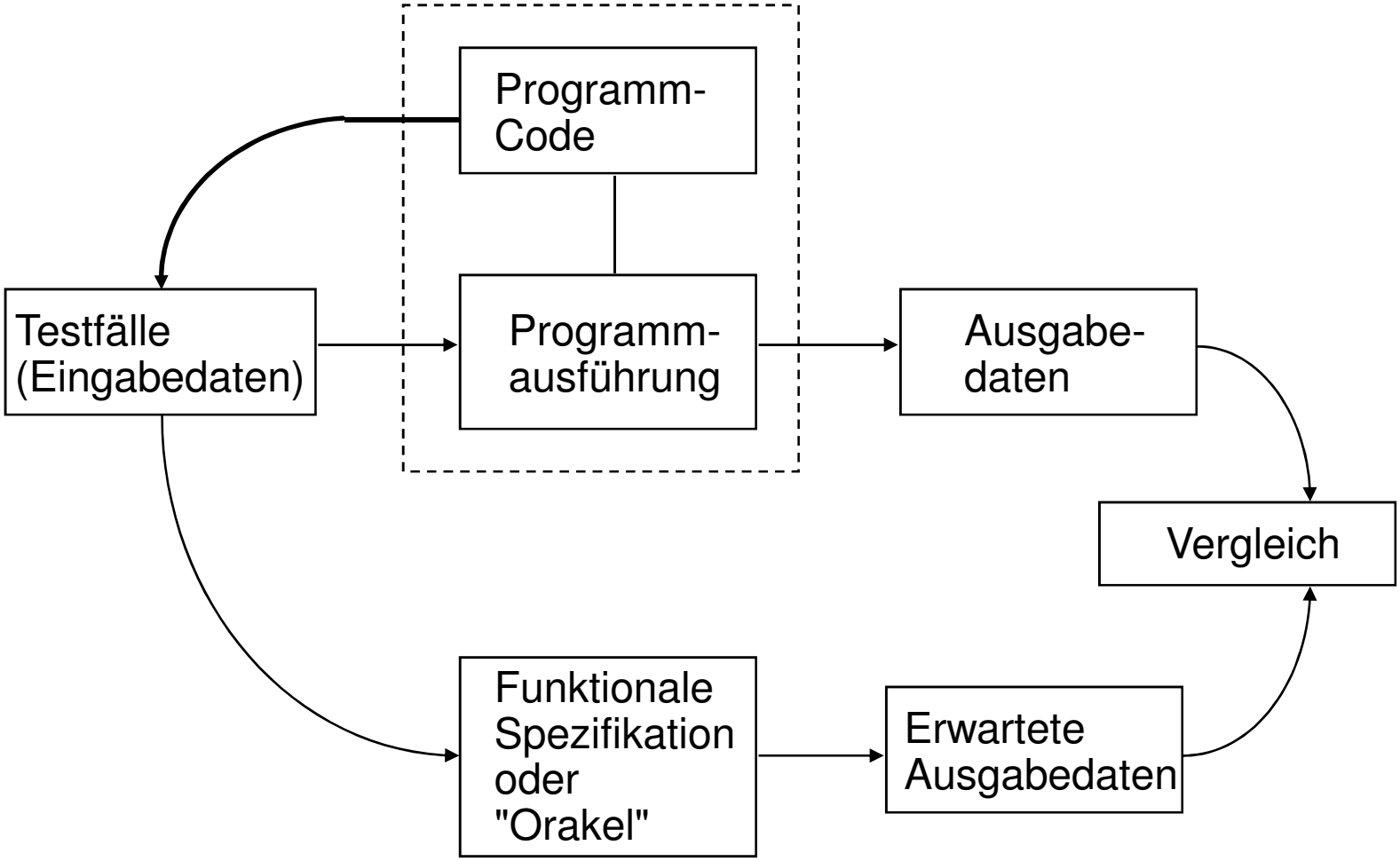
```
int search (int[] a, int k)
```

Vorbedingung: a.length > 0

Nachbedingung: (result ≥ 0 ∧ a[result] == k) ∨
(result == –1 ∧ (¬∃ i . 0 ≤ i < a.length ∧ a[i] == k))

- Weitere Kriterien für Äquivalenzklassen:
  -
  -
  -
  -
  -
  -
  -
-
  -
  -

Strukturtest *(glass-box)*



Überdeckungsgrad *(coverage)*

- Maß für die Vollständigkeit eines Tests
- Welcher Anteil des Programmtexts wurde ausgetestet?
- **Messung** der Überdeckung:
  - *Instrumentierung* des Programmcodes
  - Ausgabe statistischer Informationen
- **Planung** der Überdeckung:
  - gezielte Anlage der Tests auf volle Überdeckung
- Überdeckungsarten:
  - *Anweisungsüberdeckung:*
    Anteil ausgeführter Anweisungen
  - *Zweigüberdeckung:*
    Anteil ausgeführter Anweisungen und Verzweigungen
  - *Pfadüberdeckung:*
    Anteil ausgeführter Programmablaufpfade
- Hilfsmittel:
  - *Kontrollflussgraph*

Beispielprogramm: Binäre Suche

```
public static final int NOTFOUND = -1;

// Binaere Suche auf Array a.
// Annahme: a ist sortiert
// Ergebnis ist NOTFOUND, wenn k nicht in A enthalten ist.
// Ergebnis ist i falls a[i] gleich k ist.

public static int binSearch (int[] a, int k) {
    int ug = 0, og = a.length-1,
         m, pos = NOTFOUND;
    while (ug <= og && pos == NOTFOUND) {
        m = (ug + og) / 2;
        if (a[m] == k)
            pos = m;
        else
            if (a[m] < k)
                ug = m + 1;
            else
                og = m - 1;
    };
    return pos;
}
```

Beispielprogramm: Binäre Suche Kontrollflussgraph

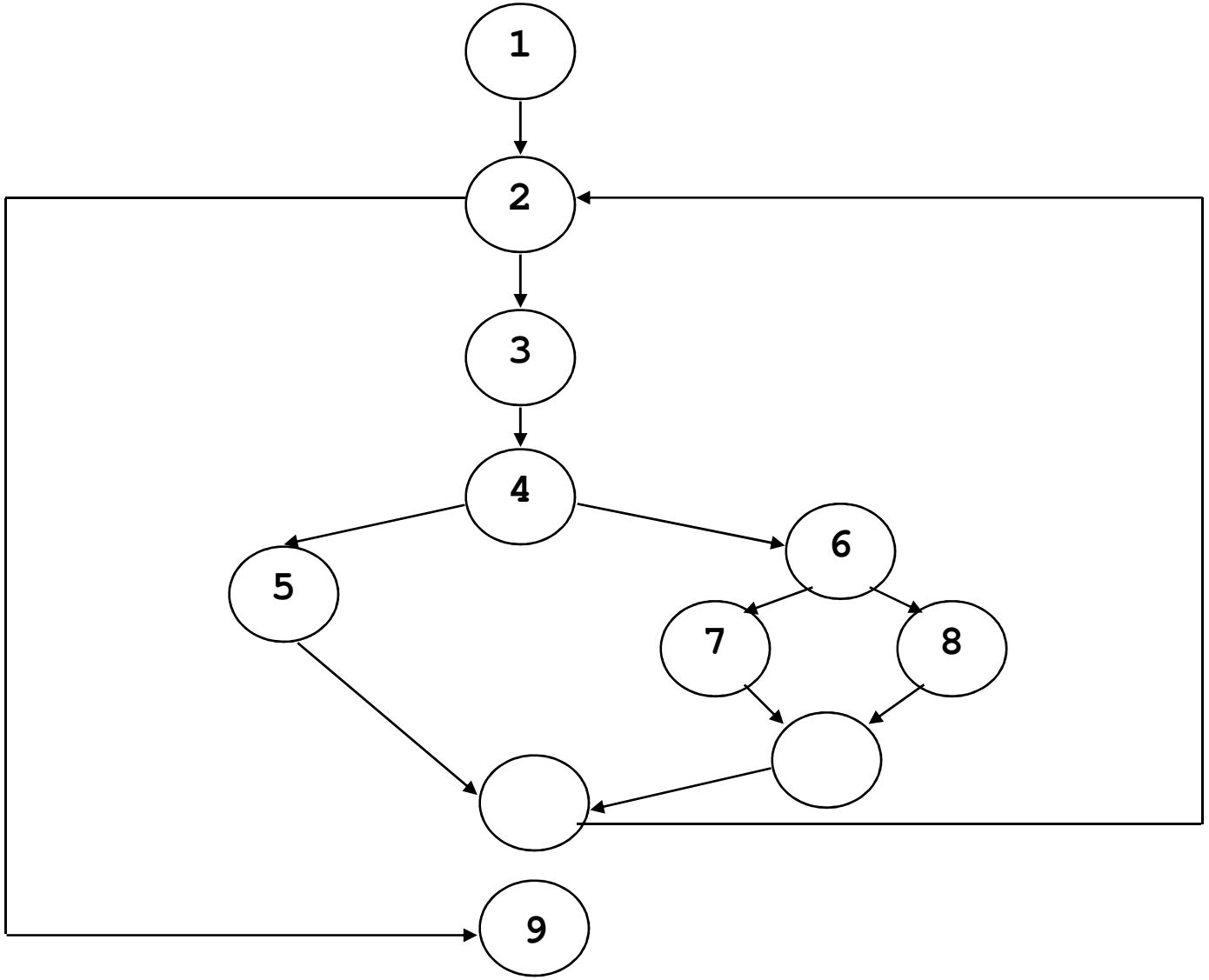

```
public static final int NOTFOUND = -1;

public static int binSearch (int[] a, int k) {
    int ug = 0, og = a.length-1,
        m, pos = NOTFOUND;
    while (ug <= og && pos == NOTFOUND) {
        m = (ug + og) / 2;
        if (a[m] == k)
            pos = m;
        else
            if (a[m] < k)
                ug = m + 1;
            else
                og = m - 1;

    };
    return pos;
}
```

Beispiel: Ergebnis: Kontrollflussgraph

discuss



Anweisungsüberdeckender Test

- Überdeckung aller Anweisungen: *C_0-Test*
  - Jede Anweisung (numerierte Zeile) des Programms wird mindestens einmal ausgeführt.

- Beispiel:

  a = {11, 22, 33, 44, 55}, k = 33

  überdeckt Anweisungen: 1, 2, 3, 4, 5, 9

  a = {11, 22, 33, 44, 55}, k = 15

  überdeckt Anweisungen: 1, 2, 3, 4, 6, 7, 8, 9

  Beide Testfälle zusammen erreichen volle Anweisungsüberdeckung.

- Messen der Anweisungsüberdeckung:
  - "Instrumentieren" des Codes (durch Werkzeuge)
  - Einfügen von Zählanweisungen bei jeder Anweisung

Zweigüberdeckender Test

- Überdeckung aller Zweige: *C_1-Test*
  - Bei jeder Fallunterscheidung (einschließlich Schleifen) werden beide Zweige ausgeführt (Bedingung=true und Bedingung=false).
  - Zweigtest zwingt auch zur Untersuchung "leerer" Alternativen:

```
if (x >= 1)
        y = true; // kein else-Fall
```

- Beispiel:
  - Die beiden Testfälle der letzten Folie überdecken alle Zweige.

- Messung/Instrumentierung von Anweisung i:
  - Fallunterscheidung:

```
if (...) { bT[i]++; ... } else { bF[i]++; ... }
```

  - While-Schleife:

```
while (...) { bT[i]++; ... } bF[i]++;
```

Pfadüberdeckung

- *Pfad* =
  Sequenz von Knoten im Kontrollflussgraphen,
  so dass in der Sequenz aufeinanderfolgende Knoten
  im Graphen direkt miteinander verbunden sind.

  Anfang = Startknoten, Ende = Endknoten.

- Theoretisch optimales Testverfahren
- Explosion der Zahl von möglichen Pfaden, deshalb **nicht praxisrelevant**. (Schleifen haben unendliche Pfad-Anzahlen)
- Praktikablere Varianten, z.B. *bounded-interior*-Pfadtest:
  Alle Schleifenrümpfe höchstens n-mal (z.B. einmal) wiederholen

Testen objektorientierter Programme

- Klassische Verfahren anwendbar für einzelne Methoden
  - aber: Methoden vergleichsweise kurz und einfach
- Komplexität liegt zum großen Teil in der *Kooperation*.
- Probleme mit Vererbung: Tests der Oberklassen müssen u.U. für alle Unterklassen wiederholt werden:
  - Beeinflussung von Variablen der Oberklasse
  - Redefinition von Methoden der Oberklasse

- Spezielle Techniken für objektorientiertes Testen:
  - Zustandstests
  - Sequenztests

Zustandstests

- Spezifikationsbezogen ("black box"):
  - Verwendung von Zustandsdiagrammen (z.B. UML) aus Analyse und Entwurf
  - Abdeckungskriterien:
    - alle Zustände
    - alle Übergänge
    - für jeden Übergang alle Folgeübergänge der Länge n
  - Praxistauglich
- Programmbezogen ("glass box"):
  - Automatische Berechnung eines Zustandsdiagramms
  - Zustand = Abstraktion einer Klasse zulässiger Attributwerte
  - Bestimmung der Übergänge erfordert symbolische Auswertung von Methoden
    - problematisch bei Schleifen und Rekursion
  - Im Forschungsstadium

Wohin mit dem Test-Code?

- Zur Durchführung von Tests entsteht zusätzlicher Code:
  - Testtreiber
  - Testattrappen (Stubs)
  - Testsuiten
- Testcode muss aufbewahrt, Tests nach allen größeren Änderungen wiederholt werden
- Alternativen:
  - Testcode als Bestandteil der Klassen
    - einfach zu verwalten, vergrößert Code
  - "Spiegelbildliche" Hierarchie von Testklassen
    - Redundanzproblem
    - Erleichtert Verwendung von Test-Frameworks (z.B. *JUnit*)
  - Testskripten in eigenen Sprachen
    - klassischer Ansatz, keine Vererbung von Testfällen möglich
  - Test-Unterklassen
    - problematisch wegen Mehrfachvererbung

Inkrementelles Testen

- Programmierer testen meist nicht gerne.
  - „Testen ist zerstörerische Tätigkeit.“
  - Zu spätes Testen führt zu komplizierten Fehlersuchen!
- Inkrementelles Testen / Test-First-Ansatz:
  Tests entstehen parallel zum Code (oder sogar vor dem Code)
- Programmierer schreiben Tests für praktischen Nutzen:
  - Klare Spezifikation von Schnittstellen
  - Beschreibung kritischer Ausführungsbedingungen
  - Dokumentation von Fehlerbeseitigung
  - Festhalten von notwendigem Verhalten vor größerem Umbau ("Refaktorisierung")
- Tests archiviert und automatisch ausführbar
- Weitere Information:
  - K. Beck, E. Gamma: Programmers love writing tests (*JUnit*)
  - K. Beck: Extreme programming explained, Addison-Wesley 2000

Zusammenfassung: 9. Qualitätsmanagement

- Qualitätsmanagement beginnt bereits mit den frühen Aktivitäten und begleitet das gesamte Projekt

- Qualitätsmanagement beinhaltet primär
  - Reviews, Inspektionen
  - Tests
- aber auch:
  - Bildung von Prototypen,
  - Einbindung von Kunden, intensive Kommunikation, gutes Arbeitsumfeld, etc.

- Testen ist eine sehr komplexe Tätigkeit, die sehr viel Erfahrung fordert
  - Tests sollten gut geplant und wenn möglich bereits frühzeitig begonnen werden
- Reviews erfordern gute Soft-Skills von allen Beteiligten und sind nur konstruktiv