universität innsbruck
Institut für Informatik

Software Engineering

3. Qualitätsmanagement

Ruth Breu

Übersicht

Literatur

- A. Spillner, Tilo Linz: Basiswissen Softwaretest, dPunkt
- P. Liggesmeyer: Software Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum

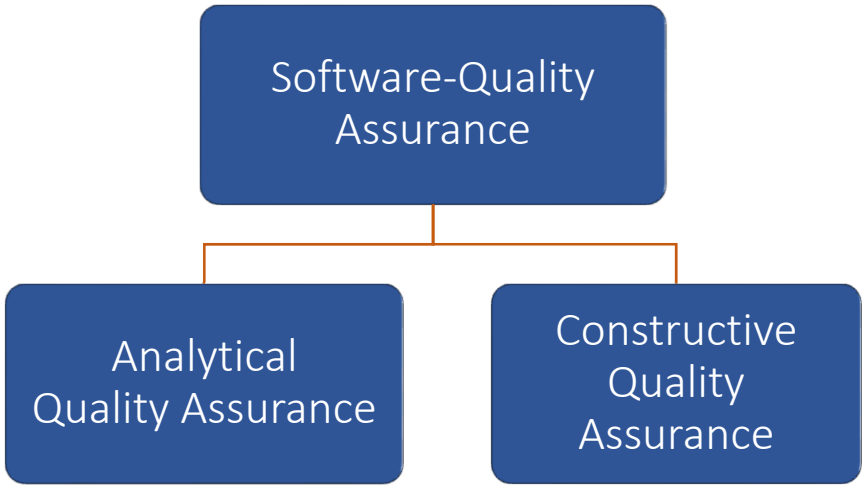
Begriffe

- Qualitätsmanagement (QM)
  - Beinhaltet alle Aktivitäten, um Qualität zu definieren, planen, überprüfen, sichern und zu verbessern
- Qualitätsmanagement-System (QMS)
  - Organisatorische Strukturen und Prozesse des Qualitätsmanagements
- Qualitätssicherung (Quality Assurance) (QA)
  - Techniken und Methoden der Qualitätssicherung

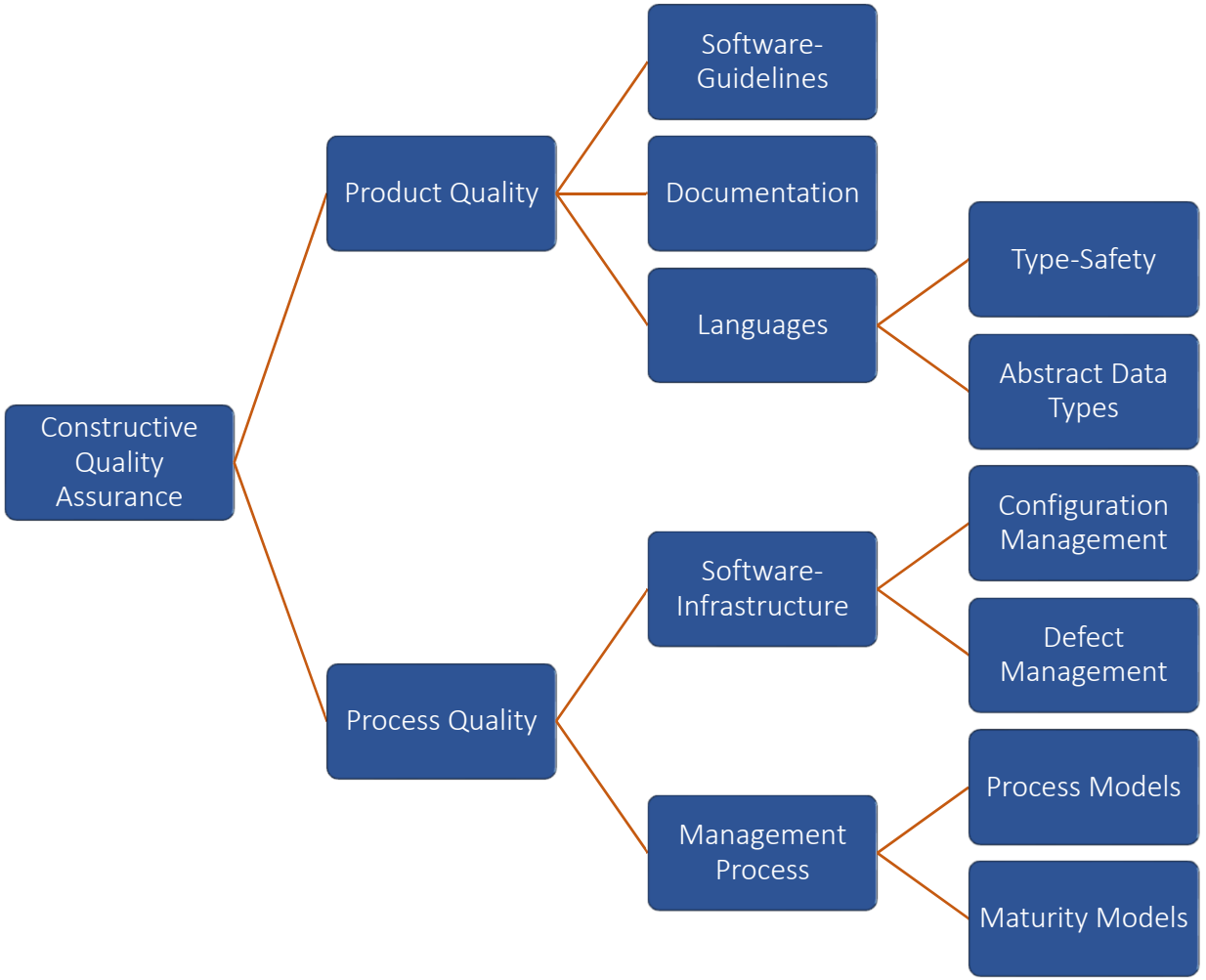
Produkt- und Prozessqualität

- Primäres Ziel ist Produktqualität (vgl. ISO 25010 Qualitätskriterien)
- Hilfsmittel zum Erreichen von Produktqualität: Prozessqualität
- Qualitätsattribute eines Softwareentwicklungsprozesses nach ISO 9002
  - Planbarkeit (Predictability)
  - Transparenz
  - Überprüfbarkeit (Controllability)
  - Teamfähigkeiten (Team skills)

Maßnahmen zur Qualitätssicherung



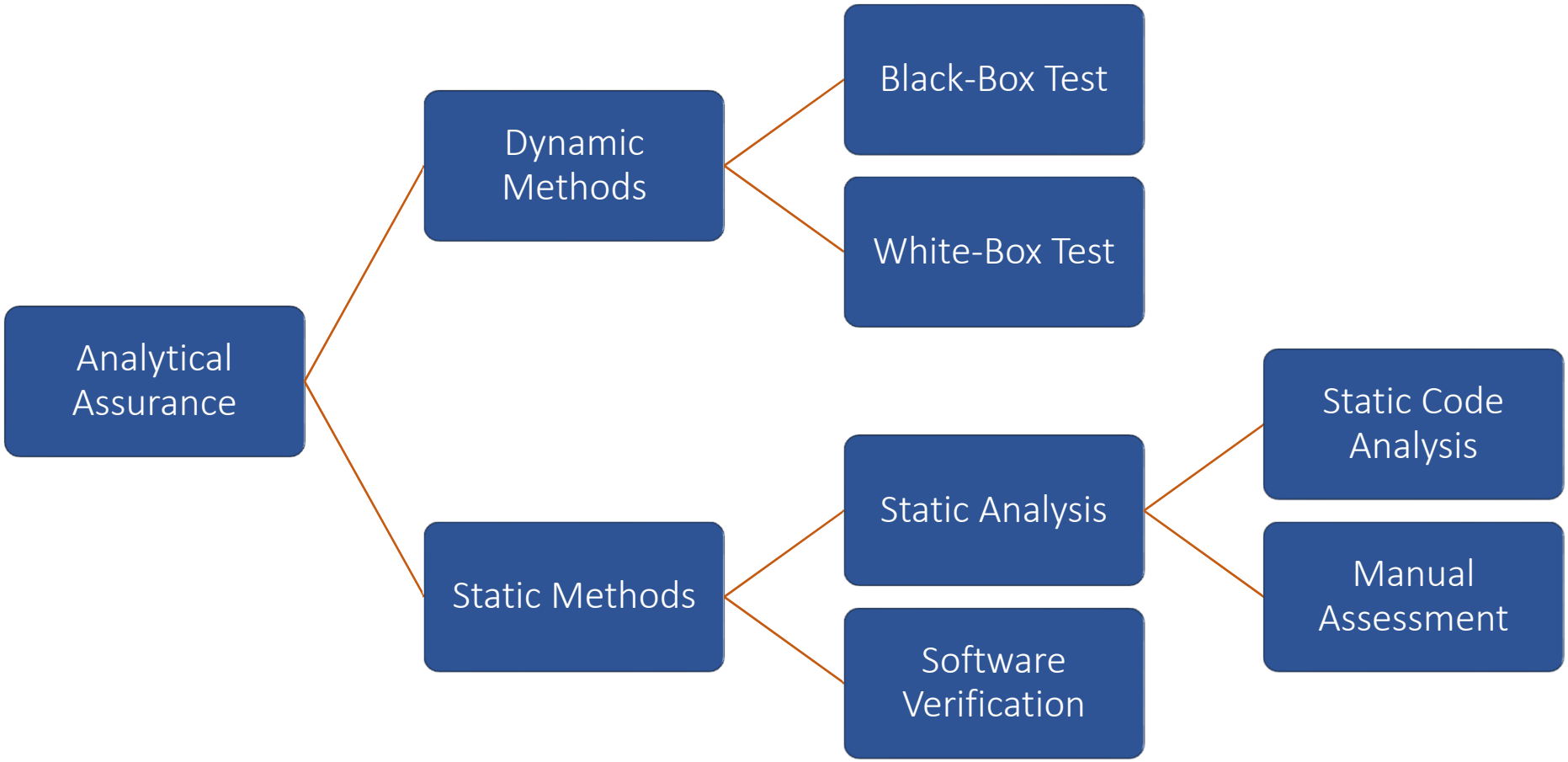
Konstruktive Qualitätssicherung



Konstruktive Qualitätssicherung – Management Prozesse

- Softwareentwicklungsprozesse
  - Definition von Aktivitäten, Artefakten, Rollen, Methoden in Softwareentwicklungsprojekten
  - Beispiel: V-Modell, Rational Unified Process, Agile Methoden
- Reifegradmodelle (Maturity Models)
  - Bewertung und Optimierung von Prozessen in der Softwareentwicklung auf Basis von Reifegraden
    - Beschreibung von Ebenen mit jeweiligen Anforderungen und Maßnahmen
    - Prüfmaßnahmen zur Bewertung eines Reifegrades
    - Beispiel: CMMI, ISO 15504 (SPICE)

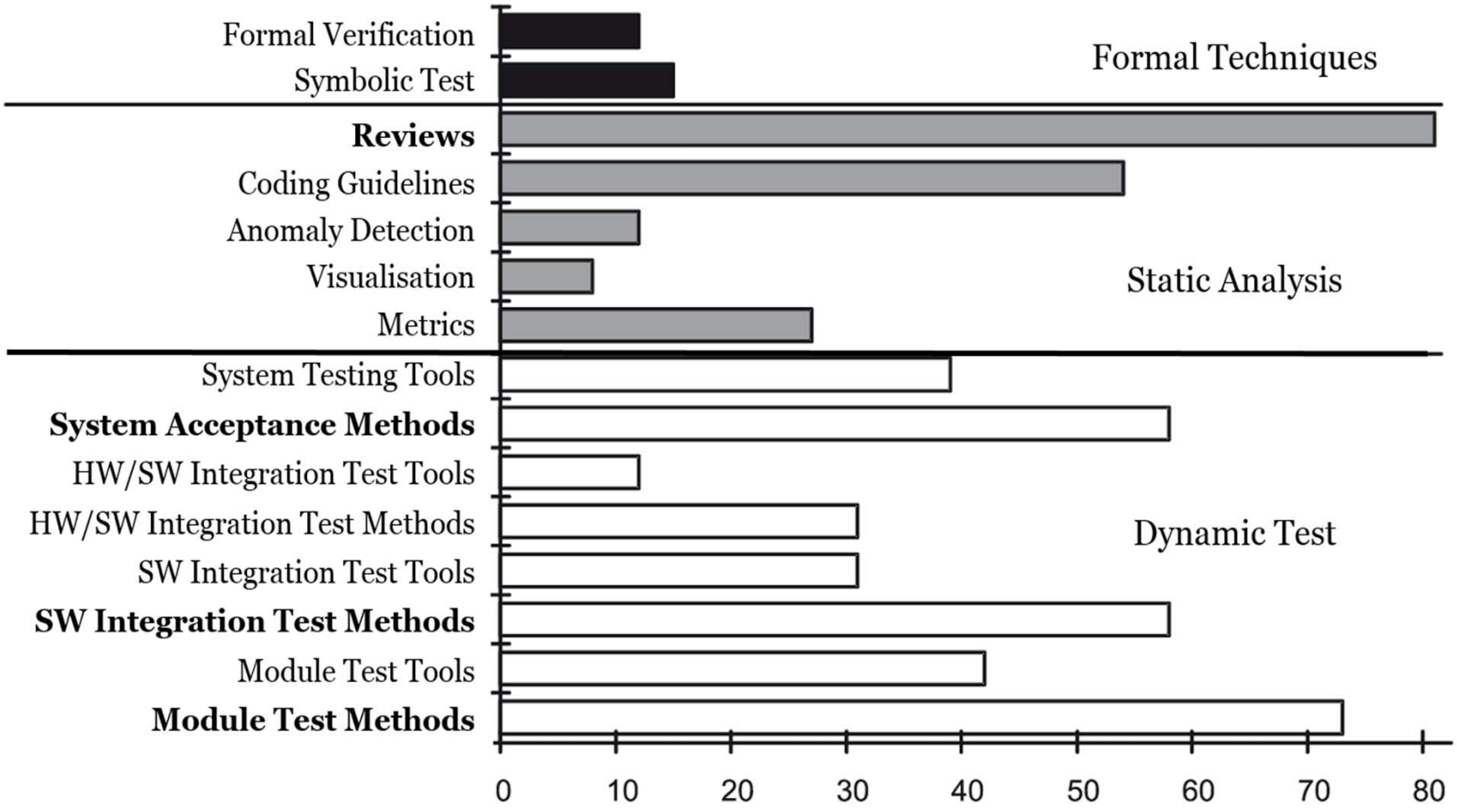
Analytische Qualitätssicherung



Analytische QA - Statische Methoden

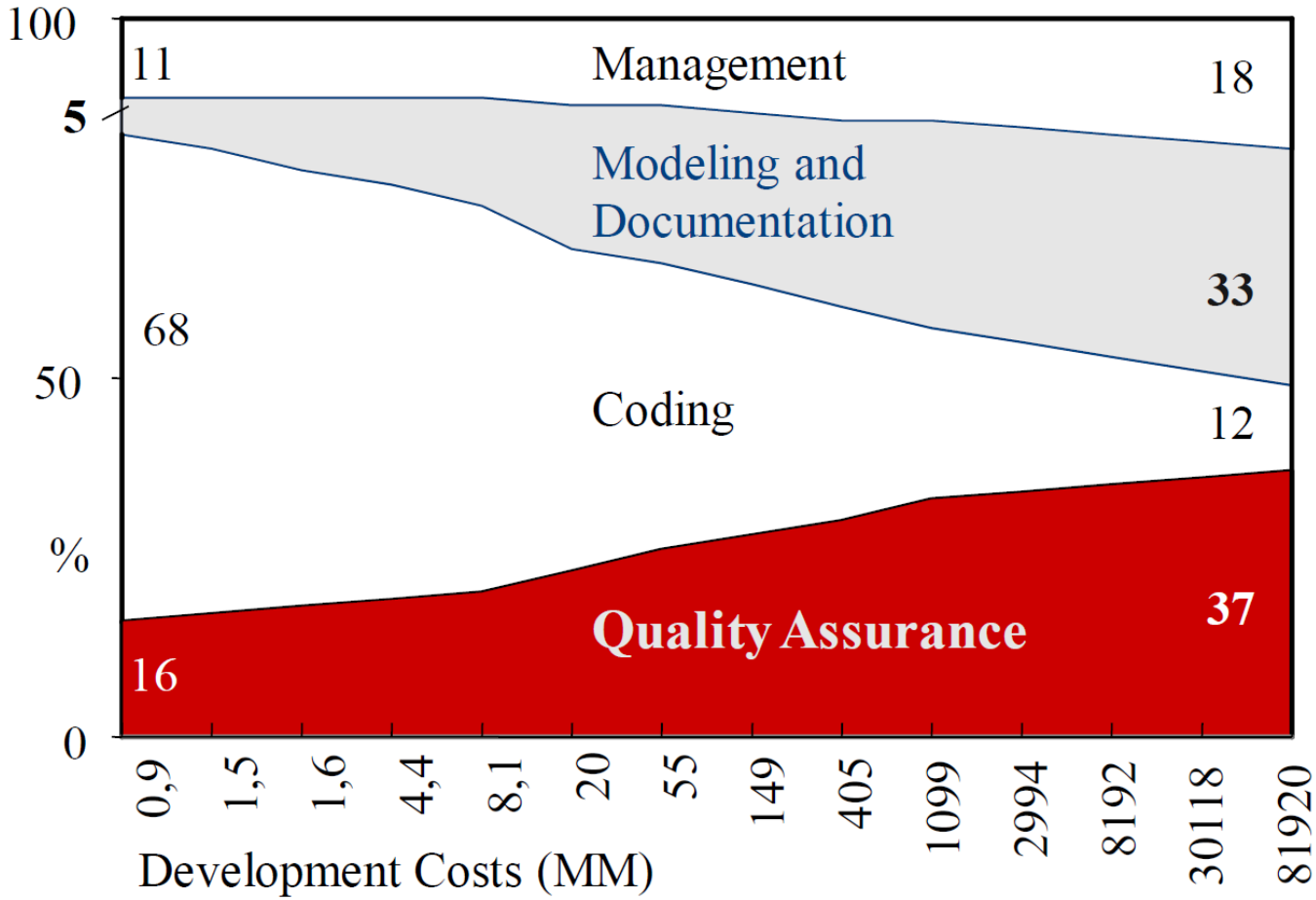
- keine Ausführung des Codes
  - Ableitung von Programmeigenschaften auf Basis von Artefakten (Source Code, Modelle)
- Statische Code Analyse
  - Automatisierte Code-Analyse
  - Erhebung und Bewertung von Metriken
- Manuelle Prüfung
  - Bewertung durch Experten (intern/extern)
  - Beispiel: Review
- Software Verifikation
  - Beweis von Programmeigenschaften auf Basis einer (formalen) Spezifikation und Methoden der Logik

Verfahren in der industriellen Praxis



Source: Liggesmeyer, P.: "Software-Qualitätssicherung"

Aufwände der Qualitätssicherung



Source: Liggesmeyer, P.: "Software-Qualitätssicherung"

Grundprinzipien der Qualitätssicherung /1

- Definition von Qualitätszielen
  - Definition von Qualitätszielen im Rahmen der Anforderungsspezifikation (vgl. nichtfunktionale Anforderungen im Pflichtenheft)
- Quantitative Qualitätssicherung
  - Qualitätsziele werden mit Kriterien der Zielerreichung verbunden („Fit Criterion“ einer Anforderung)
  - „Evidenz“ = Nachweis, dass Fit Criterion im untersuchten System erfüllt ist
- Maximale Konstruktive Qualitätssicherung
  - „Fehler, die nicht gemacht wurden, müssen nicht beseitigt werden.“

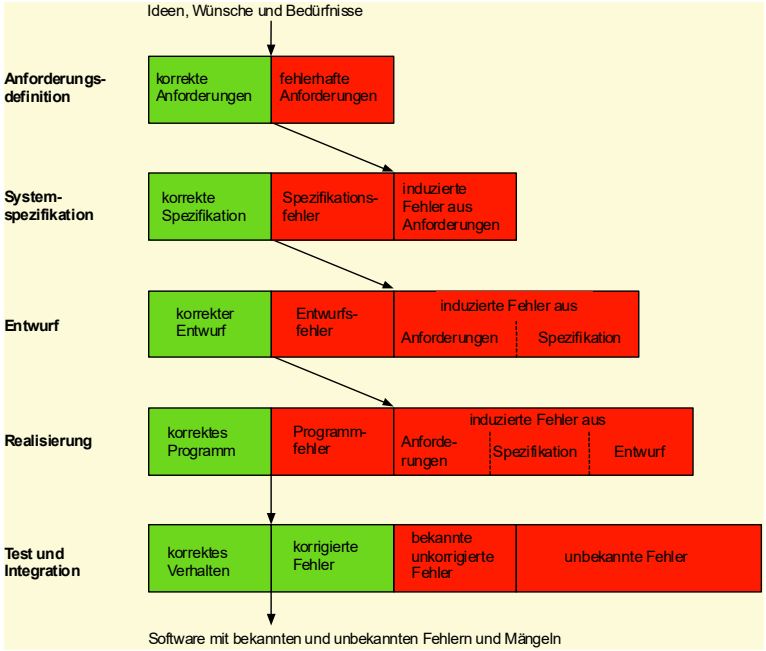
Grundprinzipien der Qualitätssicherung /2

- Frühes Finden von Fehlern
  - Je früher Fehler entdeckt werden, desto günstiger ist ihre Beseitigung.
- Integrierte Qualitätssicherung
  - Qualitätssicherungsmaßnahmen zum richtigen Zeitpunkt und begleitend zu allen Entwicklungsaktivitäten
- Unabhängige Qualitätssicherung
  - Nicht nur die Softwareentwickler sind für Qualitätssicherung zuständig

Ursprung von Fehlern

- Während Anforderungsspezifikation und Analyse werden mehr Fehler gemacht als während Design und Implementierung (60:40 Prozent).

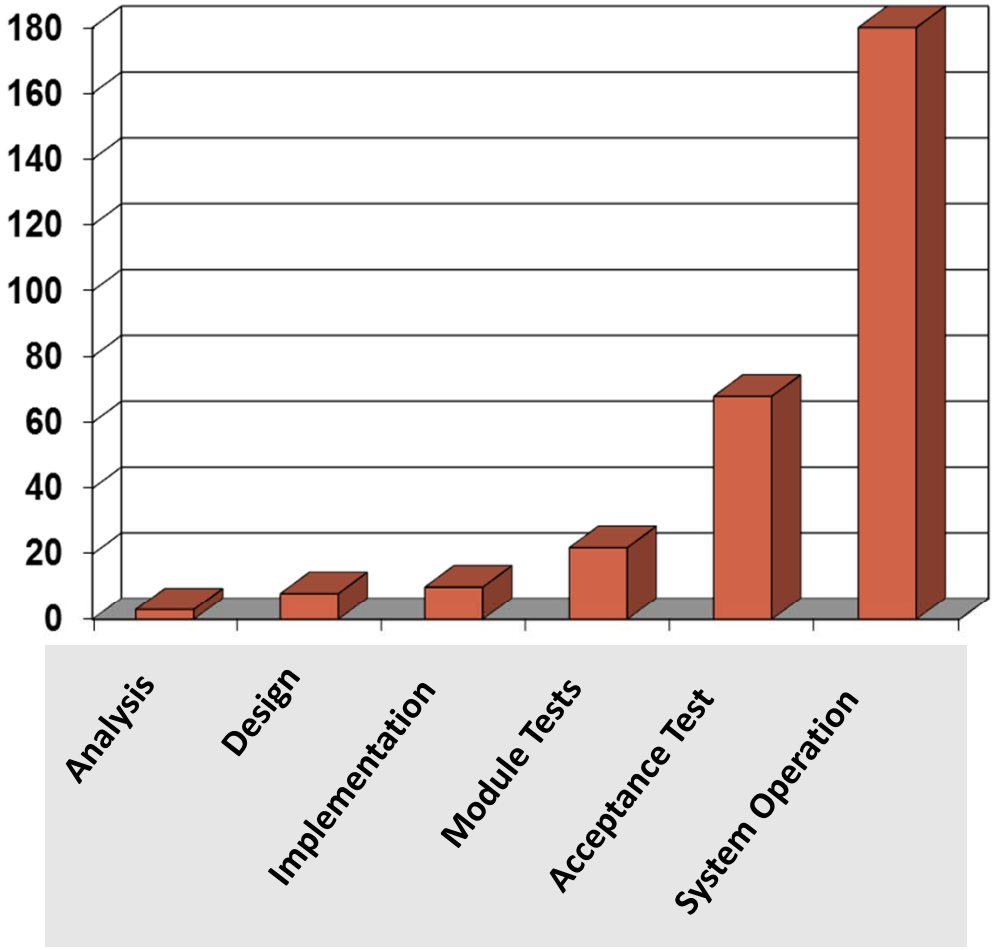
➡ Problem: frühe Fehler werden mitgeschleppt



Quelle: Barry Boehm, Balzert

Relative Kosten für die Beseitigung von Fehlern

relative Costs



Boehm, B. W.: "Software Engineering Economics"

3.2 Testende Verfahren

3.2.1 Testtypen

3.2.2 Glass-Box-Test

3.2.3 Black-Box-Test

3.2.4 Systemtests

3.2.5 Bug-Tracking-Tools

Fehlerbegriffe

- **Error** – Fehlhandlung des Entwicklers (fehlerhafte Programmierung)
- **Fault, Defect, Bug** – Fehlerzustand im Code, der zu einer Wirkung nach außen führen kann (aber nicht zwangsläufig muss)
- **Failure** – tatsächlich auftretender Fehler, Fehlerwirkung
- Errors führen zu Bugs, die Failures auslösen können. Manchmal führen Bugs erst bei Codeänderungen zu Failures.
- Die Aufgabe des Testers ist es, Testfälle zu schreiben, die Failures auslösen. Damit werden allerdings im allgemeinen nicht alle Bugs gefunden.

3.2.1 Testtypen (1)

- Nach Testobjekt
  - Modultests (Test eines einzelnen Moduls)
  - Integrationstests (Test mehrerer Module gleichzeitig, bis hin zum Gesamtsystem)
  - Systemtests (Test des gesamten Systems)
- Nach Testpersonen
  - **Entwicklertests** (interne Tests der Entwickler: => meist Modultest))
  - **Abnahmetest** (meist durch Auftraggeber oder unabhängige Testgruppe)

Testtypen (2)

- Nach Testgebiet
  - Funktionale Tests
  - Security Tests
  - Ergonomietests (Nutzerfreundlichkeit)
  - Lasttest/Stresstest (Belastungstests)
  - Installationstest/Recoverytest (Tests auf Installationsfähigkeit, Wiederanlauffähigkeit nach Systemabsturz)
  - Kompatibilitätstests (Verträglichkeit mit anderer Software)
  - ...

Testtypen (3)

- Nach Testfallgenerierung
  - Glass-Box-Tests
    - Die innere Struktur des zu testenden Objekts ist bekannt, die Testfälle werden daran ausgerichtet
  - Black-Box-Tests
    - Die innere Struktur des zu testenden Objekts ist unbekannt, die Testfälle werden aus der Spezifikation des Objekts abgeleitet

Definitionen (1)

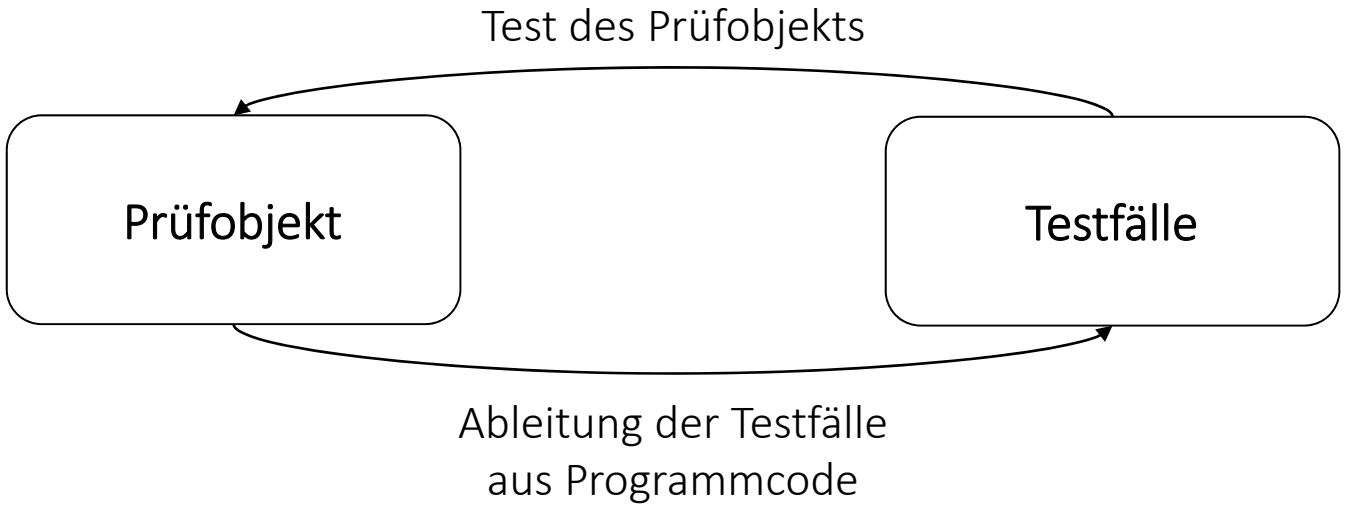
- Prüfobjekt
  - Das zu testende System/Programm/Modul/Funktion
- Testfall
  - Ein Testfall (für eine Funktion oder ein System) ist ein definierter Systemzustand, zusammen mit festgelegten Eingabeparametern und erwarteten Systemverhalten bzw. Endzustand.
- Testsuite
  - Eine Testsuite ist eine Zusammenfassung von Testfällen

Definitionen (2)

- Testtreiber
  - Testrahmen, der die Testfälle durchführt. Baut im allgemeinen den Ausgangszustand auf, übergibt die Parameter, und prüft die Ergebnisse, bzw. den Endzustand auf Korrektheit.
- Teststub
  - Trivialimplementierung einer Funktion, um darauf aufbauende Funktionen bereits testen zu können. (Z.B. für Oberflächenprototyp).
- Regressionstest
  - Eine Testsuite, die mittels eines Testtreibers immer wieder auch auf geänderten/weiterentwickelten Code (automatisch) angewendet werden kann.

3.2.2 Glass-Box-Test

- Beim Glass-Box-Test ist die innere Struktur des Programms bekannt.
- Die Testfälle werden aus der Struktur (z.B. dem Kontrollflussgraphen) abgeleitet.
- Zur Generierung der Testfälle muss die Spezifikation nicht notwendigerweise bekannt sein.



Programm ZaehleZeichen

- Aufgabe
  - Liest solange Zeichen von der Tastatur, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
  - Oder Gesamtzahl den größten durch den Datentyp int darstellbaren Wert INT_MAX erreicht
  - Ist ein gelesenes Zeichen ein Großbuchstabe, dann wird Gesamtzahl um 1 erhöht
  - Ist der Großbuchstabe ein Vokal, dann wird auch VokalAnzahl um 1 erhöht
  - Als Ausgabeparameter werden Gesamtzahl und VokalAnzahl übergeben

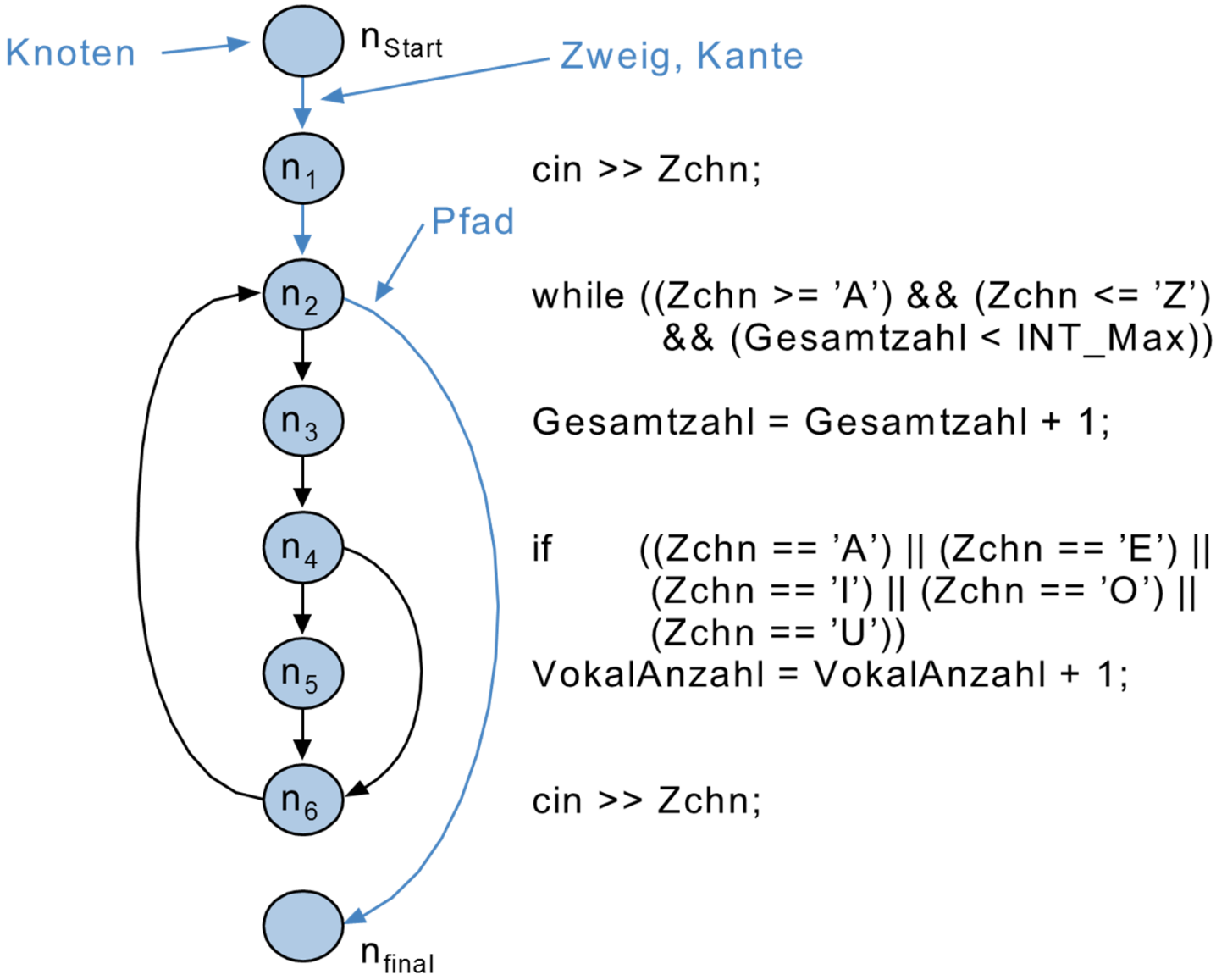
Quelle: Balzert

Programm ZaehleZeichen (Implementierung)

```
1  void ZaehleZchn(int &VokalAnzahl,
                                  int &Gesamtzahl)
2 {     char Zchn;
3       cin>>Zchn;
4       while ((Zchn>='A')&&(Zchn<='Z')&&
                  (Gesamtzahl<INT_MAX))
5       { Gesamtzahl = Gesamtzahl+1;
6       if((Zchn=='A')||(Zchn=='E')||(Zchn=='I')||
                (Zchn == 'O')||(Zchn == 'U'))
7          {   VokalAnzahl = VokalAnzahl + 1;
           }
8       cin>>Zchn;
9     } //end while
10 }.
```

Der Kontrollflussgraph

- Gerichteter Graph, der aus einer endlichen Menge von Knoten besteht
- Hat einen Startknoten und einen oder mehrere Endknoten
- Die Knoten sind durch gerichtete Kanten verbunden
- Jeder Knoten stellt eine ausführbare Anweisung dar
- Eine gerichtete Kante von einem Knoten i zu einem Knoten j beschreibt einen möglichen **Kontrollfluss** von Knoten i zu Knoten j
- Die gerichteten Kanten werden als **Zweige** bezeichnet
- Eine abwechselnde Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit einem Endknoten endet, heißt **Pfad**


Knoten
n_{Start}
Zweig, Kante
n_1
Pfad
n_2
n_3
n_4
n_5
n_6
n_{final}
cin >> Zchn;
while ((Zchn >= 'A') && (Zchn <= 'Z')
&& (Gesamtzahl < INT_Max))
Gesamtzahl = Gesamtzahl + 1;
if ((Zchn == 'A') || (Zchn == 'E') ||
(Zchn == 'I') || (Zchn == 'O') ||
(Zchn == 'U'))
VokalAnzahl = VokalAnzahl + 1;
cin >> Zchn;

Kontrollflussorientierte Strukturtestverfahren

- Bekannteste Testverfahren
  - **Anweisungs**überdeckungstest
  - **Zweig**überdeckungstest
  - **Pfad**überdeckungstest
- Ziel
  - Mit einer Anzahl von Testfällen alle vorhandenen **Anweisungen**, **Zweige** bzw. **Pfade** auszuführen

Anweisungs- & Zweigüberdeckungstest - Übersicht

- Anweisungsüberdeckungstest
  - Auch C0-Test (C = Coverage)
  - Ausführung aller Anweisungen, d.h. aller **Knoten** des Kontrollflussgraphen
- Zweigüberdeckungstest
  - Auch C1-Test
  - Ausführung aller Zweige, d.h. aller **Kanten** des Kontrollflussgraphen.
- Pfadüberdeckungstest
  - Umfassendstes kontrollflussorientiertes Testverfahren
  - Problem: Exponentiell viele Pfade
  - Verschiedene Testverfahren nähern sich dem Pfadüberdeckungstest an.

Anweisungsüberdeckungstest / C_0-Test

- Ziel
  - Mindestens einmalige Ausführung aller Anweisungen, d.h. aller **Knoten** des Kontrollflussgraphen
- **Überdeckungsmetrik** – Prozentsatz der ausgeführten Anweisungen
- Bewertung
  - Finden nicht ausführbaren Codes
  - nicht ausreichendes Testkriterium, da Kontrollstrukturen zu wenig Berücksichtigung finden

Zweigüberdeckungstest / C_1-Test

- Ziel
  - Ausführung aller Zweige des zu testenden Programms, d.h. Durchlaufen aller **Kanten** des Kontrollflussgraphen
- **Überdeckungsmetrik** – Prozentsatz aller ausgeführten Zweige
- Bewertung
  - 100% Überdeckung: Alle unterschiedlichen Fälle werden zumindestens einmal durchlaufen – es gibt keine nicht-ausführbare Zweige
  - Kombination von Zweigen oder komplexe Bedingungen werden nicht ausreichend berücksichtigt
  - Schleifen werden nicht ausreichend getestet, da ein einzelner Durchlauf durch den Schleifenkörper oft schon hinreichend ist
  - Fehlende Zweige können nicht entdeckt werden.

Anweisungs- & Zweigüberdeckungstest

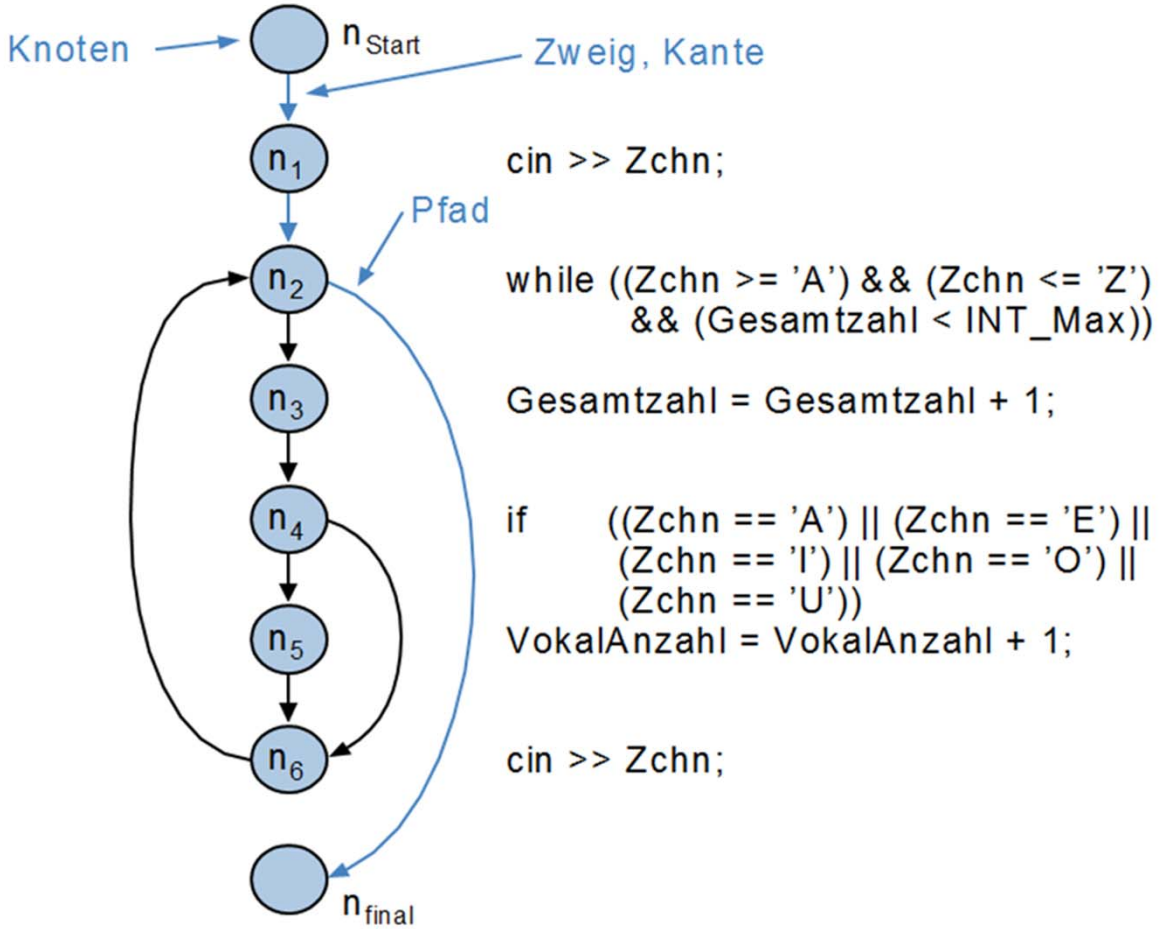
Beispiel

```
x = 1;
if (x >=1)
    x = x +1;
```

Möglicherweise ist in der Entscheidung die falsche Variable verwendet worden (dieser Fehler wird mit dem Anweisungsüberdeckungstest nicht notwendigerweise erkannt)

Pfadüberdeckungstestverfahren

- Pfadüberdeckungstestverfahren
  - Entwickelt, um Programme mit Wiederholungen bzw. Schleifen ausreichend testen zu können
- Pfadüberdeckungstest fordert die Ausführung aller unterschiedlichen Pfade des zu testenden Programms.



exponentiell viele Pfade

Der boundary interior Pfadtest (1/3)

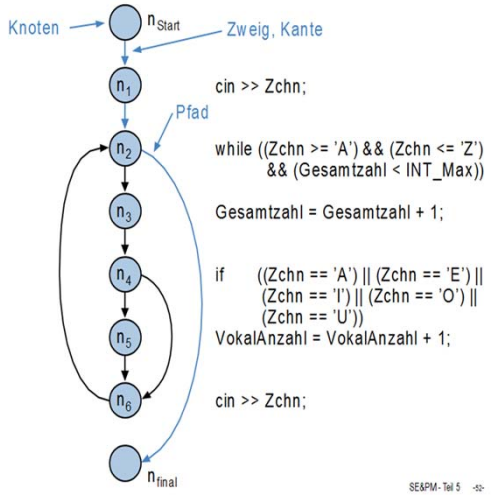
- Einordnung
  - Eingeschränkte, schwächere Version des Pfadüberdeckungstests
  - Für schleifenfreie Programme ist er mit dem Pfadüberdeckungstest identisch
- Ziel
  - Definition aller möglichen Testfälle, die den Schleifenkörper keinmal, einmal oder zweimal durchlaufen

Der boundary interior Pfadtest (2/3)

- Zwei Testfall-Gruppen
  - Grenztest-Gruppe *(boundary tests):*
    - Enthält alle Pfade, die die Schleife zwar betreten (=Test der Bedingung), sie jedoch nicht wiederholen
  - Gruppe zum Test des Schleifeninneren *(interior tests):*
    - Umfasst alle Pfade, bei denen der Schleifenkörper einmal oder zweimal durchlaufen wird

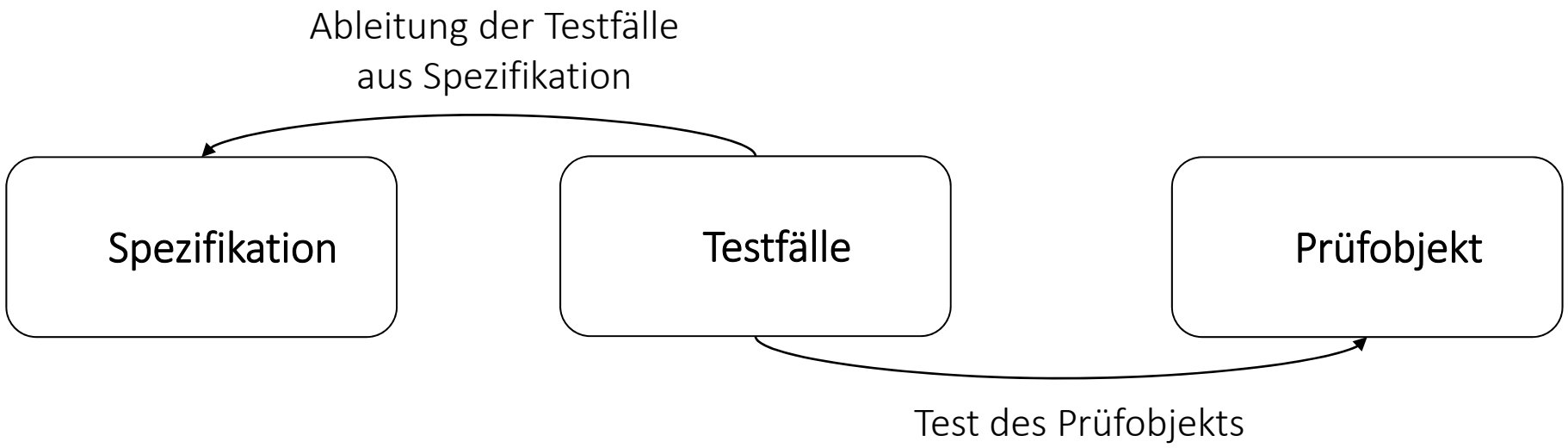
Der boundary interior Pfadtest (3/3)

- Beispiel: Testfälle für **`ZaehleZchn:`**
  1. Testfall für den Pfad außerhalb der Schleife
     a) Aufruf mit **`Gesamtzahl = 0, Zchn = ´1´`**
  2. Einmaliges Durchlaufen der Schleife
     a) Aufruf mit **`Gesamtzahl = 0, Zchn = 'A', '1'`**
     b) Aufruf mit **`Gesamtzahl = 0, Zchn = 'B', '1'`**
  3. Zweimaliges Durchlaufen der Schleife
     a) **`Zchn = 'E', 'I', '*'`**
     b) **`Zchn = 'A', 'H', '!'`**
     c) **`Zchn = 'H', 'A', '+'`**
     d) **`Zchn = 'X', 'X', ','`**



3.2.3 Black-Box-Tests

- Beim Black-Box-Test ist die innere Struktur des Programms nicht bekannt.
- Die Testfälle werden aus der Spezifikation abgeleitet.



Funktionale Testverfahren

- Aufgabe der Testplanung
  - Aus der Spezifikation Testfälle herzuleiten, mit denen das Programm getestet werden soll
  - Zu einem Testfall gehören
    - Eingabedaten in das Testobjekt
    - Erwartete Ausgabedaten oder Ausgabereaktionen (Soll-Ergebnisse)
- Herausforderungen
  - Ableitung der geeigneten Testfälle
  - Vollständiger Funktionstest ist i. allg. nicht durchführbar.

Funktionale Testverfahren

- Ziel einer Testplanung
  - Testfälle so auswählen, dass die Wahrscheinlichkeit groß ist, Fehler zu finden
- Testfallbestimmung
  - Funktionale Äquivalenzklassenbildung
  - Grenzwertanalyse
  - Zufallstest

Funktionale Äquivalenzklassenbildung

- Verfahren
  - Die Definitionsbereiche der **Eingabeparameter** und die Wertebereiche der **Ausgabeparameter** werden in **Äquivalenzklassen** zerlegt
  - Es wird davon ausgegangen, dass ein Programm bei der Verarbeitung eines **Repräsentanten aus einer Äquivalenzklasse** so reagiert, wie bei allen anderen Werten aus dieser Äquivalenzklasse.
- Voraussetzung
  - Sorgfältige Wahl der Äquivalenzklassen

Funktionale Äquivalenzklassenbildung

- Beispiel

```
void setzeMonat(short aktuellerMonat);
//Es muss gelten: 1 ≤ aktuellerMonat ≤ 12
```

  - Eine gültige Äquivalenzklasse:
    a) `1 ≤ aktuellerMonat ≤ 12`
  - Zwei ungültige Äquivalenzklassen:
    a) `aktuellerMonat < 1, aktuellerMonat > 12`
  - Aus den Äquivalenzklassen abgeleitete Testfälle:
    a) `aktuellerMonat = 5`
    b) `aktuellerMonat = -3`
    c) `aktuellerMonat = 25`

    (Auswahl beliebiger Repräsentanten)

Funktionale Äquivalenzklassenbildung

- Bewertung
  - Geeignetes Verfahren, um aus Spezifikationen – insbesondere aus Parameterbereichen – **repräsentative Testfälle** abzuleiten
  - Die Aufteilung in Äquivalenzklassen muss **nicht** mit der internen Programmstruktur übereinstimmen
  - Es werden einzelne Parameter betrachtet
  - Beziehungen, Wechselwirkungen und Abhängigkeiten zwischen Werten werden nicht behandelt
  - Spezifikation ist auf die Angabe von Wertebereichen beschränkt
    - Modellbasierte Testverfahren nutzen ausdrucksstärkere Spezifikationsmethoden, z.B. Zustandsautomaten

Regeln zur Äquivalenzklassenbildung

- Bildung von Eingabeäquivalenzklassen
  1) Eingabebedingung mit zusammenhängendem Wertebereich
     - 1 gültige Äquivalenzklasse
     - 2 ungültige Äquivalenzklassen
     - (vgl. Beispiel)

Regeln zur Äquivalenzklassenbildung

2) Eingabebedingung spezifiziert eine Menge von Werten
   - Werte werden evtl. unterschiedlich behandelt
   - Für jeden Wert **1 eigene gültige Äquivalenzklasse**
   - Für alle Werte mit Ausnahme der gültigen Werte **1 ungültige Äquivalenzklasse**.
   - Beispiel:
     - Jahreszeiten:
       - Frühling, Sommer, Herbst, Winter
     - 4 gültige Äquivalenzklassen:
       - Frühling, Sommer, Herbst, Winter
     - 1 ungültige Äquivalenzklasse:
       - XXX

Regeln zur Äquivalenzklassenbildung

3) Parameter wird durch Bedingung eingeschränkt
   - 1 gültige Äquivalenzklasse
   - 1 ungültige Äquivalenzklasse
   - Beispiel:
     - Das 1. Zeichen muss ein Buchstabe sein
     - 1 gültige Äquivalenzklasse:
       - Das 1. Zeichen ist ein Buchstabe
     - 1 ungültige Äquivalenzklasse:
       - Das 1. Zeichen ist kein Buchstabe (z.B. Ziffer oder Sonderzeichen)

Regeln zum Bilden von Testfällen

- Nach Identifikation der Äquivalenzklassen:
  - Repräsentanten auswählen
  - Anschließend Testfälle zusammenstellen:
- Regeln für Testfälle
  - Testfälle für **gültige Äquivalenzklassen**
    - Werden durch Auswahl von Testdaten aus gültigen Äquivalenzklassen gebildet – dabei sollten alle gültigen Äquivalenzklassen abgedeckt sein
    - Dies reduziert die Testfälle für gültige Äquivalenzklassen

Testfälle für ungültige Äquivalenzklassen

- Werden durch Auswahl eines Testdatums aus einer ungültigen Äquivalenzklasse gebildet
- Testfall wird mit Werten kombiniert, die ausschließlich aus gültigen Äquivalenzklassen entnommen sind
- Da für alle ungültigen Eingabewerte eine Fehlerbehandlung existieren muss, kann bei Eingabe eines fehlerhaften Wertes pro Testfall die Fehlerbehandlung nur durch dieses fehlerhafte Testdatum verursacht worden sein
- Regel erleichtert das Nachverfolgen unerwünschten Systemverhaltens.

Beispiel

- Drei Parameter x, y, z
  $1 \leq x \leq 10, \quad 11 \leq y \leq 20, \quad 21 \leq z \leq 30$
- Die Regel führt z.B. zur Definition folgender Testfälle

  5, 13, 22 (drei Repräsentanten aus gültigen Äqu.klassen)
  0, 13, 22 (erster Parameter aus ungült. Äqu.klasse)
  12, 15, 24 (erster Parameter aus ungült. Äqu.klasse)
  6, **10**, 24 (zweiter Parameter aus ungült. Äqu.klasse)
  5, **21**, 22 (zweiter Parameter aus ungült. Äqu.klasse)
  5, 13, **20** (dritter Parameter aus ungült. Äqu.klasse)
  5, 13, **31** (dritter Parameter aus ungült. Äqu.klasse)

Grenzwertanalyse

- Es wird nicht irgendein Element aus der Äquivalenzklasse als Repräsentant ausgewählt
- Es werden ein oder mehrere Elemente ausgesucht, so dass jeder Rand der Äquivalenzklasse getestet wird
  - Grund: Fehler entstehen häufig an den Rändern der Äquivalenzklassen
- Annäherung an die Grenzen der Äquivalenzklasse
  - Vom gültigen Bereich aus
  - Vom ungültigen Bereich aus

Grenzwertanalyse

- Beispiel

```
void setzeMonat (short aktuellerMonat);
```

  - Spezifikation: $1 \leq aktuellerMonat \leq 12$
  - 1 gültige Äquivalenzklasse: $1 \leq aktuellerMonat \leq 12$
  - 2 ungültige Äquivalenzklassen:
    $aktuellerMonat < 1$, $aktuellerMonat > 12$
  - Abgeleitete Testfälle
    - 1 $aktuellerMonat = 1$ (untere Grenze)
    - 2 $aktuellerMonat = 12$ (obere Grenze)
    - 3 $aktuellerMonat = 0$
      (obere Grenze der ungültigen Äquivalenzklasse)
    - 4 $aktuellerMonat = 13$
      (untere Grenze der ungültigen Äquivalenzklasse).

Beispiel Ableiten von Testfällen bei Grenzwertanalyse

- Drei Parameter x, y, z
  $1 \leq x \leq 10, \quad 11 \leq y \leq 20, \quad 21 \leq z \leq 30$
- Die Regel führt z.B. zur Definition folgender Testfälle

  1, 11, 30
  10, 20, 21
  (Werte aus gült. Äqu.klassen, jeweils beide Randwerte)
  0, 11, 21 (erster Parameter aus ungült. Äqu.klasse)
  11, 11, 21 (erster Parameter aus ungült. Äqu.klasse)
  1, **10**, 30 (zweiter Parameter aus ungült. Äqu.klasse)
  1, **21**, 30 (zweiter Parameter aus ungült. Äqu.klasse)
  10, 20 **20** (dritter Parameter aus ungült. Äqu.klasse)
  10, 11, **31** (dritter Parameter aus ungült. Äqu.klasse)

Kombinierter Funktions- und Strukturtest

- Nachteile Strukturtest
  - Fehlende Funktionalitäten werden **nicht** erkannt
  - Bei einzelnem Testziel, z.B. vollständiger Zweigüberdeckung, entstehen oft triviale Testfälle, ungeeignet zur Prüfung der Funktionalität
    - Der Funktionstest erzeugt aufgrund seiner Orientierung an der Spezifikation aussagefähige Testfälle.
- Nachteile Funktionstest
  - Nicht in der Lage, die konkrete Implementierung geeignet zu berücksichtigen
    - Ein vollständiger Funktionstest erfüllt daher in der Regel nicht die Minimalanforderungen einfacher Strukturtests
- Funktions- und Strukturtestverfahren daher miteinander kombinieren. Diese „Grey Box“ Testsuiten können in Tools wie Junit implementiert werden.

3.2.4 Systemtests

- Mit Systemtests soll das Zusammenwirken des Gesamtsystem getestet werden.
- Oft ist der (erfolgreiche) Systemtest der letzte Schritt vor der Auslieferung, bzw. der Akzeptanztest durch den Kunden selbst
- Der Systemtest wird in einem Testplan/Testdrehbuch vorbereitet, Tester arbeiten dann das Testdrehbuch ab
- Ergebnis ist ein Testprotokoll mit den durchgeführten Tests und den gefundenen Abweichungen

Inhalt des Testdrehbuchs

- Beschreibung der zu testenden Softwarekonfiguration
  - Softwareversion
  - Hardwareumgebung, Datenbank, ...
- Bereitstellen von Testdaten (z.B. für Datenbank)
- Testfallbeschreibungen
- Zeitplan, Dokumentation des Systemtests

Beispiel für Testbeschreibung

- Jeder Use Case sollte durch einen oder mehrere Testfälle abgedeckt werden.
- Da auch Use Cases Alternativen/Schleifen von Aktionen enthalten können, können auch auf der Ebene des Systemtests die Überdeckungskriterien von Glass-Box-Tests angewendet werden.

Testfall: unkorrekter Login	
Name:	Fehlerhafter Login
Bezug zur Spezifikation:	Use-Case Nr. 1.1.2.
Ausgangszustand:	Die Anwendung zeigt nach dem Start die Loginmaske.
Aktion	Der Nutzer gibt den Loginnamen „meier“ und das Passwort „123456“ in die Loginmaske ein
erwarteter Ergebniszustand:	Die Anwendung zeigt erneut einen Loginbildschirm mit der Fehlermeldung „Login und Passwort stimmen nicht überein“, das Feld für den Loginnamen ist mit „meier“ vorbelegt, das Passwortfeld ist leer.
beobachtete Abweichung	
☐ OK ☐ kosmetische Abweichungen ☐ mittlere Abweichungen ☐ große Abweichungen ☐ System unbenutzbar	

Testdaten

- Viele Anwendungsdomänen benötigen das langfristige Management von Testdaten-Beständen.
- Ansatz 1 – Dump von Daten aus der Produktionsumgebung
  - Pro
    - Realistische Daten
  - Con
    - Nicht immer verfügbar
    - Datenschutz-Probleme
    - Daten veralten (z.B. Datum einer Bestellung)

Ansatz 2 – SynthetischeTestdaten

- Definition per Hand
  - „Kunde mit Vertragstyp XXX und Kontosaldo von 5000 €“
- Generierung
  - Generierung von 1000 Kunden mit Dummy-Namen und -Adressen
  - Zusätzlich: Berücksichtigung von Constraints (z.B. Alter) und Verteilungen (z.B. Mittelwert)
- Verwendung verfügbarer externer Datenquellen
  - Z.B. Postleitzahlen und Adressen
- Ansatz 1 und 2 werden oft kombiniert
  - Datenbank-Dump + Pseudo-Anonymisierung von Personen
  - Dummy-Kunden mit echten Adressen
  - „Zeitreise“ von Testdaten (Datum von Bestellungen wird in der Testdatenbank laufend nachgezogen)

3.2.5 Fehlerverfolgungssysteme (Bug-Tracking Tools)

- Fehlerverfolgungssysteme
  - Ziel: Transparentes und nachvollziehbares Management von Fehlern
  - unterstützen die Verwaltung der Problemerfassung, der Fehleranalyse, und der Fehlerbehebung
  - geben jederzeit Auskunft über den Zustand des Fehlers
  - werten Fehlerdaten statistisch aus (z.B. Zahl der offenen Fehler, PUM, MTBF, MTTR, Zeitverlauf der Fehlerneuentdeckung)
  - geben evtl. der Person, die den Fehler entdeckt hat, Feedback, falls Problem gelöst
- Rollen im Fehlerverfolgungsprozess
  - Nutzer/Tester
  - Projektmanager
  - Testmanager
  - Entwickler

MTBF Mean Time between Failure
MTTR: Mean Time to Repair
PUM: Problems per User Month
(Problemberichte pro Nutzer pro Monat)

Fehleridentifikation

- Assoziierte Informationen:
  - Fehlernummer (laufende Nummer)
  - Datum des Erfassens
  - Art des Tests (Review/Walkthrough, Test)
  - Evtl. Testdaten
  - Beschreibung des Fehler/Verletzte Anforderung/Zugehöriger Testfall
  - Kritikalität des Fehlers
  - Lokalisierung/Modul/Dokument
  - Version des betroffenen Moduls/Dokuments
  - Aufwand des Findens
  - Maßnahme (Änderung Anforderungen/Änderung Dokument/Keine Änderung)
  - Fehlerstatus: Gefunden/Behoben/Getestet
  - Datum Behebung
  - Aufwand Fehlerbehebung
  - Von Behebung betroffene Module/Dokumente

Lebenszyklus eines Fehlers