Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Grundlagen der Programmierung & Algorithmen und Datenstrukturen

Einführung in die strukturierte Programmierung – Teil 4

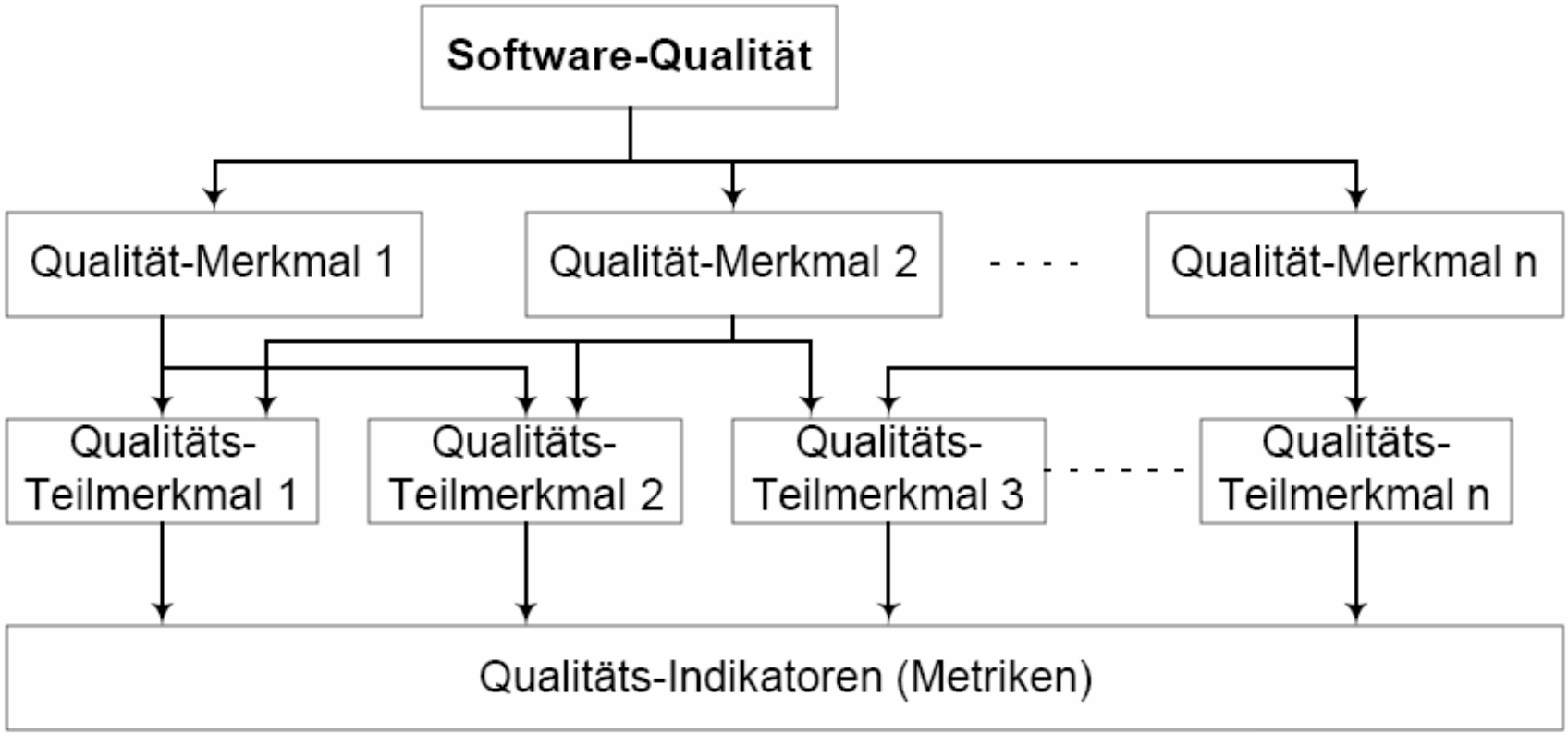
Die Inhalte der Vorlesung wurden primär auf Basis der angegebenen Literatur erstellt. Darüber hinaus orientiert sich diese an der Vorlesung von Prof. Dr.-Ing. Faustmann (ebenfalls HWR Berlin).

Inhalt des Parts

- Software-Qualität
  - Aspekte der Softwarequalität (inkl. ISO9126 bzw. ISO25000)
  - Verbesserung der Verständlichkeit
- Software-Test (Aspekt der Zuverlässigkeit)
  - Praxisorientierte Testprinzipien
  - Verfahren und Methoden
  - Testabdeckung
  - Black-Box-Stapeltest

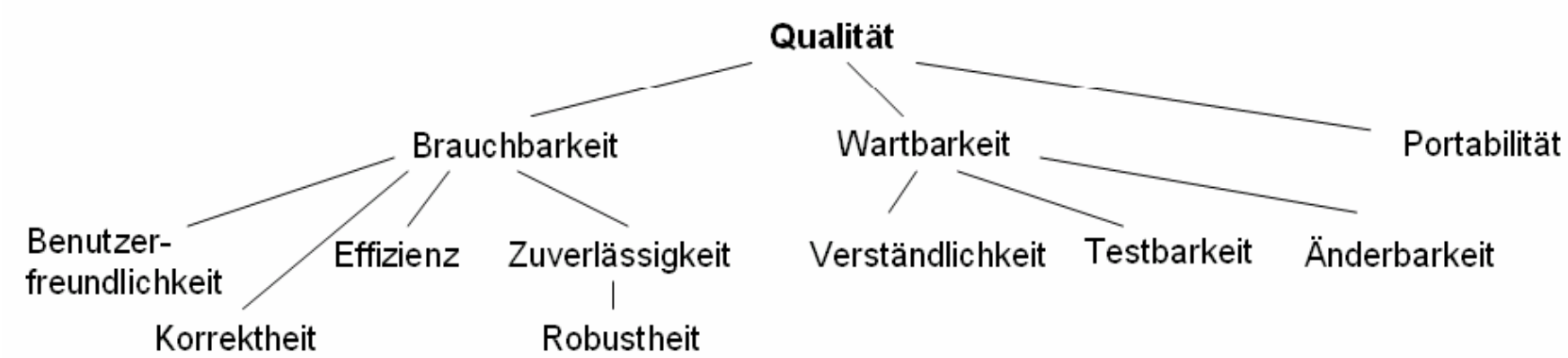
Software-Qualität

Software-Qualität



Software-Qualität

- Qualitätsanforderungen an Software können aus verschiedenen Perspektiven (Interessenslagen) betrachtet werden.
- Während z.B. Wartbarkeit und Portabilität für den Hersteller und späteren Betreiber der Software von Interesse sind, sind für den Kunden alle Aspekte der Brauchbarkeit wichtig.



Überblick zur ISO 25000

(ehm. ISO 9126 und ISO 14598)

- Die ISO 25000 definiert „Product Quality Evaluation System“ für SW-Produkte

  Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

- Ablösung der lange genutzten ISO 9126
  - Definition der Qualitätsmerkmale
  - Software Produkt Qualität - Externe Metriken
  - Software Produkt Qualität - Interne Metriken
  - Produktbezogene Verwendung von Softwaremetriken
  - …
- ISO/IEC 14598 Prozess für Produktanalyse

Quelle der Abb.: https://iso25000.com/index.php/en/iso-25000-standards, letzter Zugriff – Nov. 2020

ISO 25010 als Kern des Standards

- Die aufgezeigten Qualitätskriterien sind über den Lebenszyklus der Softwareentwicklung zu berücksichtigen bzw. entsprechend den Anforderungen auszuprägen.
- Mit Hilfe der Möglichkeiten einer konstruktiven und analytischen Qualitätssicherung lassen sich viele der im Bild dargestellten Aspekte erfassen.

Verbesserung der SW-Qualität

- Verständlichkeit - Maintainability
  - Dokumentation
  - Durchführen von Code-Reviews
  - Verwendung sprechender Bezeichner
  - Aufbau des Quellcodes verbessern/ übersichtlicher gestalten
- Zuverlässigkeit – Reliability
  - Automatisches Erkennen von Ausnahmesituationen
  - Durchführung von Tests für besondere Situationen
  - Vermeiden von Gefahrensituationen (z.B. Lokalität von Variablen)
- Änderbarkeit- Portability
  - Vermeidung von Redundanzen (z.B. Feldlängen)
  - Vermeiden von Abhängigkeiten (z.B. Verwendung def. Schnittstellen)

Übung - Qualitätsmodelle

- Welche Ziele sind mit der Bereitstellung eines Standards für die Bewertung der Qualität von Softwareprodukten verbunden?
- Machen Sie sich mit den 5 Bereichen (divisions) der ISO/IEC 25000 grob vertraut und dokumentieren Sie diese stichpunktartig.
- Wählen Sie aus der ISO25010 drei Qualitätsmerkmale aus.
- Mit Hilfe welcher Methoden lassen sich die ausgewählten Qualitätsmerkmale während der Softwareentwicklung absichern.

→ Gruppenübung – 20 min.

→ Auswertung von 3 Gruppenergebnisse – maximal 5 min.

Verbesserung der Verständlichkeit

- Namenskonventionen
  - Standardisierte Art der Benennung und sprechende Namen für
    - Variablen
    - Typen
    - Funktionen
- Quelltextstruktur
  - Einrückungen (Tabulatoren) fördern die Übersicht
  - Explizite Klammerungen von Anweisungsblöcken vermeiden durch Anpassungen bedingte Fehler
- Unit-Spezifikationen
  - Kurzbeschreibung der Leistung von Programmeinheiten wie
    - Methoden und Funktionen
    - Klassen
    - Pakete und Module

Verbesserung der Verständlichkeit

Allgemeine Namenskonventionen für Java:

- Laufvariablen werden als `i, j, k` eingesetzt.
- Typennamen werden groß geschrieben, Variablennamen immer klein:

```
Person person; // Eine Variable vom Typ Person
```

- Verwendung sprechender Namen für Bezeichner, z.B.:

```
naechsteZeileliefern()
```

- Zwei Bezeichner dürfen sich nicht ausschließlich in der Groß- und Kleinschreibung unterscheiden.
- Konstanten werden nur mit Großbuchstaben bezeichnet:

```
public static final int MWST = 16;
```

- Funktionen und Methoden werden mit einem führenden Verb bezeichnet:

```
getIndex() statt nur index()
```

- Ansonsten Programmierkonventionen des jeweiligen Entwicklungshauses!

Verbesserung der Verständlichkeit

Übung - Namenskonventionen:

- Ändern Sie die folgende Funktion so um, dass sämtliche Bezeichner den vorangegangenen Regeln entsprechen. Erstellen Sie ein Java-Programm!

```
int funktion(int a, int b){
   int x=a;
   if(b==0){
         return(1);
   }
   for(;b>1;b--){
         x*=a;
   }
   return(x);
}
```

Verbesserung der Verständlichkeit

Quelltextstruktur:

- Anweisungsblöcke sind mit einer Tab-Position einzurücken.
- Die öffnende Klammer des Anweisungsblockes ist auf der vorherigen Zeile zu platzieren.
- Die schließende Klammer kann mit einem Kommentar versehen werden.
- Anweisungsblöcke sind auch bei nur einer Operation immer in Klammern zu setzen.
- Beispiel:

```
public static int getAbsolut(int zahl){
    if(zahl < 0){
        return(-zahl);
    } else {
        return(zahl);
    } //Abschluss der if-Anweisung
} //Abschluss der getAbsolut-Methode
```

Übung – Bitschieben (links/rechts)

Welche Ausgabe wird durch die folgende Java-Methode auf der exakt Konsole erzeugt?

```
public static void getTestBitschieben(){
    //Bitmuster bei 16 Bit - |0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|
    int bitPattern1 = 1;
    //Bitmuster bei 16 Bit - |0|0|0|0|0|0|1|0|0|0|0|0|0|0|0|0|
    int bitPattern2 = 512;
    System.out.println("Ausgangssituation vor linksschieben "+bitPattern1);
    System.out.println("Ausgangssituation vor rechtsschieben "+bitPattern2);
    System.out.println("-------------------------------------");
    for (int i = 1; i<10;i++){
        bitPattern1 = bitPattern1 << 1; //auch bitPattern <<= 1
        System.out.println("linksschieben << "+bitPattern1);
        bitPattern2 = bitPattern2 >> 1; //auch bitPattern <<= 1
        System.out.println("rechtsschieben >> "+bitPattern2);
    }
}
```

Verbesserung der Verständlichkeit

Übung - Quelltextstruktur:

- Verändern Sie die im Transferbereich enthaltene Datei *DezToBin.java* so, dass die Programmstruktur durch Einrückungen und Klammerungen verständlicher wird.
  - Erläuterung des Programms durch Kommentare
  - Grafische Dokumentation des Programmablaufs (inkl. der jeweils gültigen Belegung genutzter Variablen – Bitmuster berücksichtigen)
  - Präsentation ausgewählter Ergebnisse
- Bemerkung:
  - Innerhalb des Programms werden Bit-Operatoren zur bitweisen Verknüpfung von Operanden (Bitmuster beachten) verwendet.
  - Informieren Sie sich über benötigte Wahrheitstabellen (AND, OR, XOR und NOT) mit Hilfe einschlägiger Literatur bzw. des Internets.

Verbesserung der Verständlichkeit

Unit-Spezifikation – allgemeine Angaben:

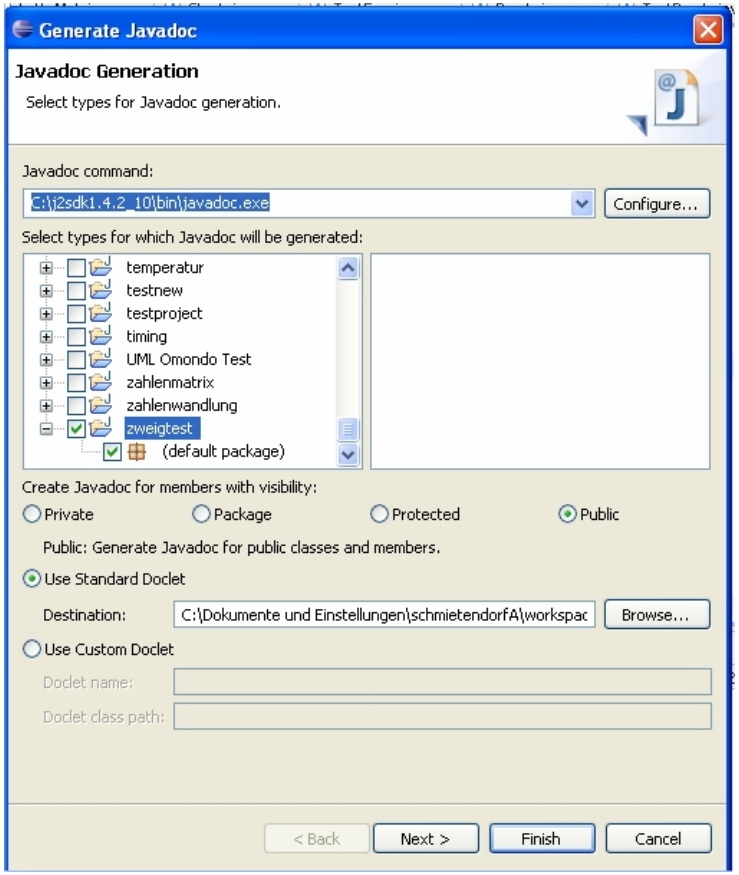
- Die Spezifikation einer Unit beschreibt, was sie leisten soll, nicht wie!
  Bsp: `/* Effekt: Sortiert die Elemente von Feld aufsteigend */`
- Bei einer abstrakteren Einheit (Klasse, Paket):
  - Zweck: Wozu dient die Einheit?
  - @author: Wer ist der Autor?
  - @version: Aktuelle Versionsnummer
  - Änderungshistorie: Liste von Änderungen mit Datum, Autor und Inhalt
- Bei einer Methode wird angegeben:
  - Vorbedingungen: Was darf vorausgesetzt werden?
  - Effekt: Welche Änderungen werden durchgeführt?
  - @param Welche Paramter werden benutzt?
  - @return Was liefert die Methodenausführung?
  - @throws Welche Ausnahmesituationen werden wie behandelt?

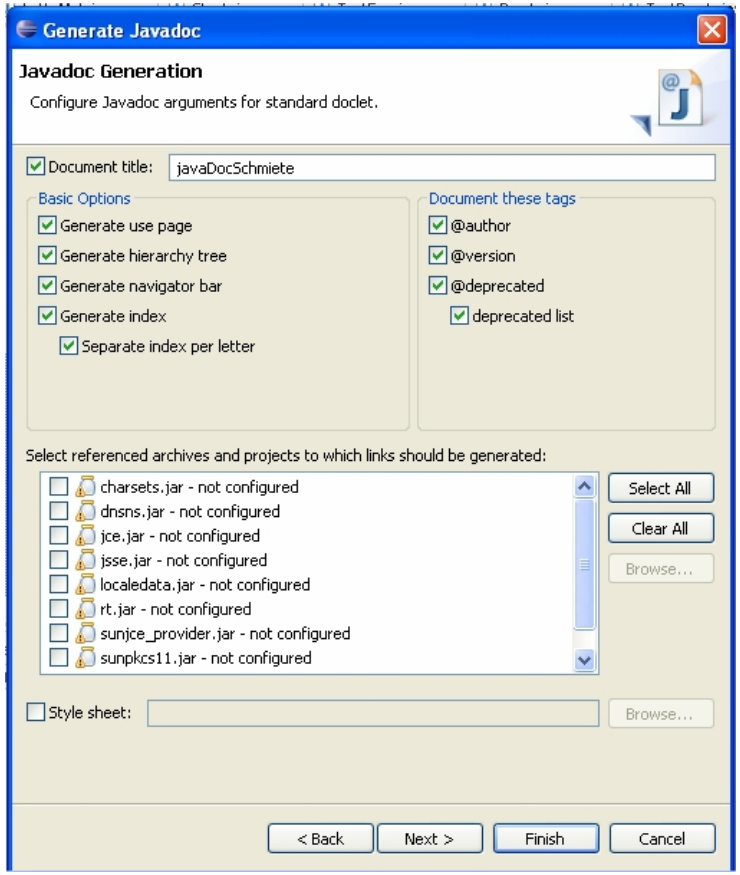
Verbesserung der Verständlichkeit

Unit-Spezifikation – unter Eclipse:

- Zunächst sind Javadoc-Kommentare in den Quellcode einzufügen.
  - Das kann durch Eclipse unterstützt werden
  - Source -> Add Javadoc Comment
- Anschließend kann eine HTML-Dokumentation erzeugt werden
  - Project -> Generate Javadoc
  - Auswahl zu dokumentierender Klassen
  - Angabe, welche javadoc-Tags berücksichtigt werden sollen.
- Berücksichtigung der Option „Open generated index file in browser“
  - Sofortige Anzeige nach dem Generieren

Verwendung von javadoc.exe





Verbesserung der Verständlichkeit

Unit-Spezifikation
Einer Klasse

```
/**
 * Modul Methodentest
 *
 * Zweck Experimente Call by Value / Call by Reference
 * in Java imm Call by Value (ggf. Kopie der Referenz)
 * @author schmietendorfA
 * @version 1.1 14.11.2017
 *
 * Historie 12.11.2017, AS, Grundgeruest implementiert
 *          14.11.2017, AS, Methoden zum Call by Value
 */
public class TestCall {
```

All Classes
TestCall

Package **Class** Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class TestCall

```
java.lang.Object
  └ TestCall
```

public class **TestCall**
extends java.lang.Object

Modul Methodentest Zweck Experimente Call by Value / Call by Reference in Java immer Call by Value (ggf. Kopie der Referenz)

Version:
1.1 14.11.2017 Historie 12.11.2017, AS, Grundgeruest implementiert 14.11.2017, AS, Methoden zum Call by Value

Author:
schmietendorfA

Verbesserung der Verständlichkeit

Unit-Spezifikation einer Methode:

```
/**
        sort

        Effekt:                 Sortiert die Elemente von
                                zahlenfeld aufsteigend
        Vorbedingungen:         zahlenliste zeigt auf ein Feld von
                                int-Werten
        @throws:                EmptyException, falls Anzahl == 0
*/

void sort(final int[] zahlenliste) {
   ...
}
```

Verbesserung der Verständlichkeit

Unit-Spezifikation - Übung:

Versehen Sie das von Ihnen erstellte Programm zur Berechnung von Pi mit einer geeigneten Unit-Spezifikation. Generieren Sie eine Javadoc-Dokumentation aus Eclipse heraus und betrachten Sie das Ergebnis mit Hilfe eines Browser. Welche alternative Dokumenttypen können mit Hilfe des Javadoc-Systems ebenfalls erzeugt werden?

→ Übung im Selbststudium realisieren – ca. 20 min.

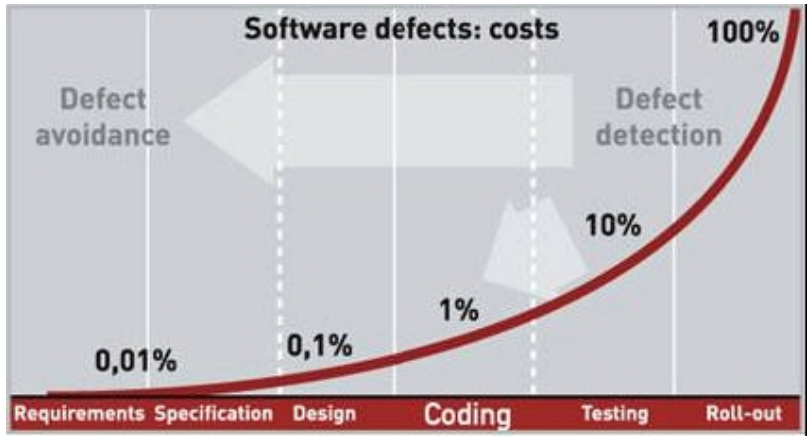
Software-Test

Motivation zum Software-Testen

- Fehlgeschlagene Projekte
  - Gepäckabfertigungssystem des Flughafen Denver
  - Verlust der Ariane 5 Rakete
  - 1h Systemausfall im Börsenhandel Verlust von 7,8 Mio. US-$ (Quelle: FAZ vom 17.01.2002)
- Unzufriedene Benutzer
  - Bananen-Software – reift beim Kunden
  - Verspätete Einführungen der Software
  - Hohe Kosten für Support-Organisationen
- Hohe Aufwände für spätere Anpassungen
  - Jahr 2000 und Euroumstellung

Motivation zum Software-Testen

- Benötigte Aufwände zur Fehlerbeseitigung steigen mit zunehmender Projektdauer
- Aufwand für Erstellung von Code ist gleich der Zeit für die Fehlererkennung
- Testaufwand sollten zwischen 25 bis 50% liegen
- 2/3 aller Fehler stecken im Quellcode



Fehler und Fehlerfreiheit

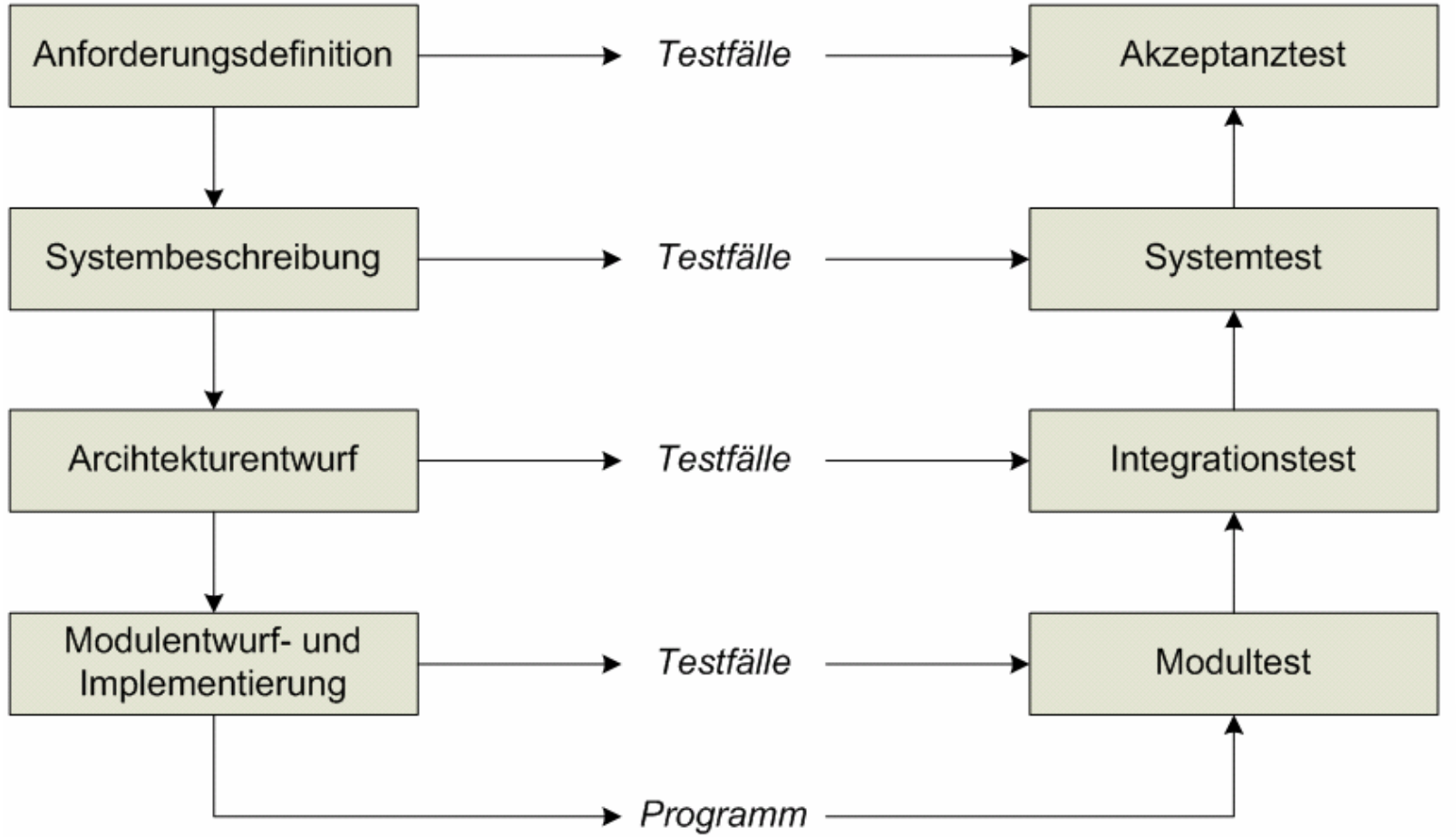
Ein Fehler ist jede Abweichung der tatsächlichen Ausprägung einer Qualitätseigenschaft (z.B. Funktionalität, Übertragbarkeit, Effizienz) von einer explizit oder implizit geforderten Eigenschaft.

„Program testing can be used to show the presence of bugs, but never to show their absence.“ (E. Dijkstra)

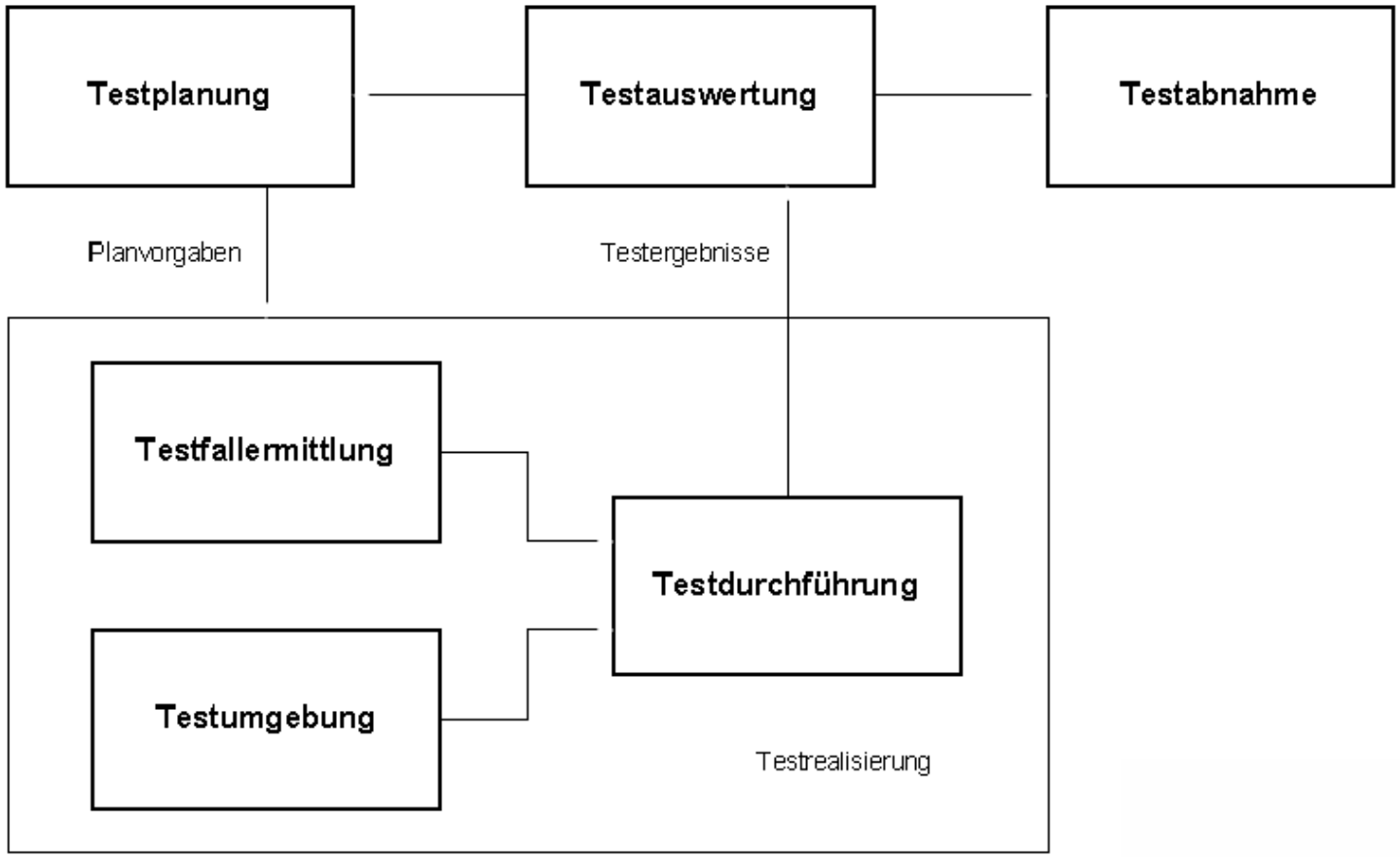
Grundprinzipien des Testens

- Fehler möglichst frühzeitig aufdecken
- Aus Fehlern lernen und zukünftig vermeiden
- Unabhängige Tests durchführen
- Testziele formulieren und bewerten
- Testfälle professionell handhaben
- Testaktivitäten planen
- Testaktivitäten dokumentieren

Phasen der SW-Entwicklung und korrespondierende Testtypen



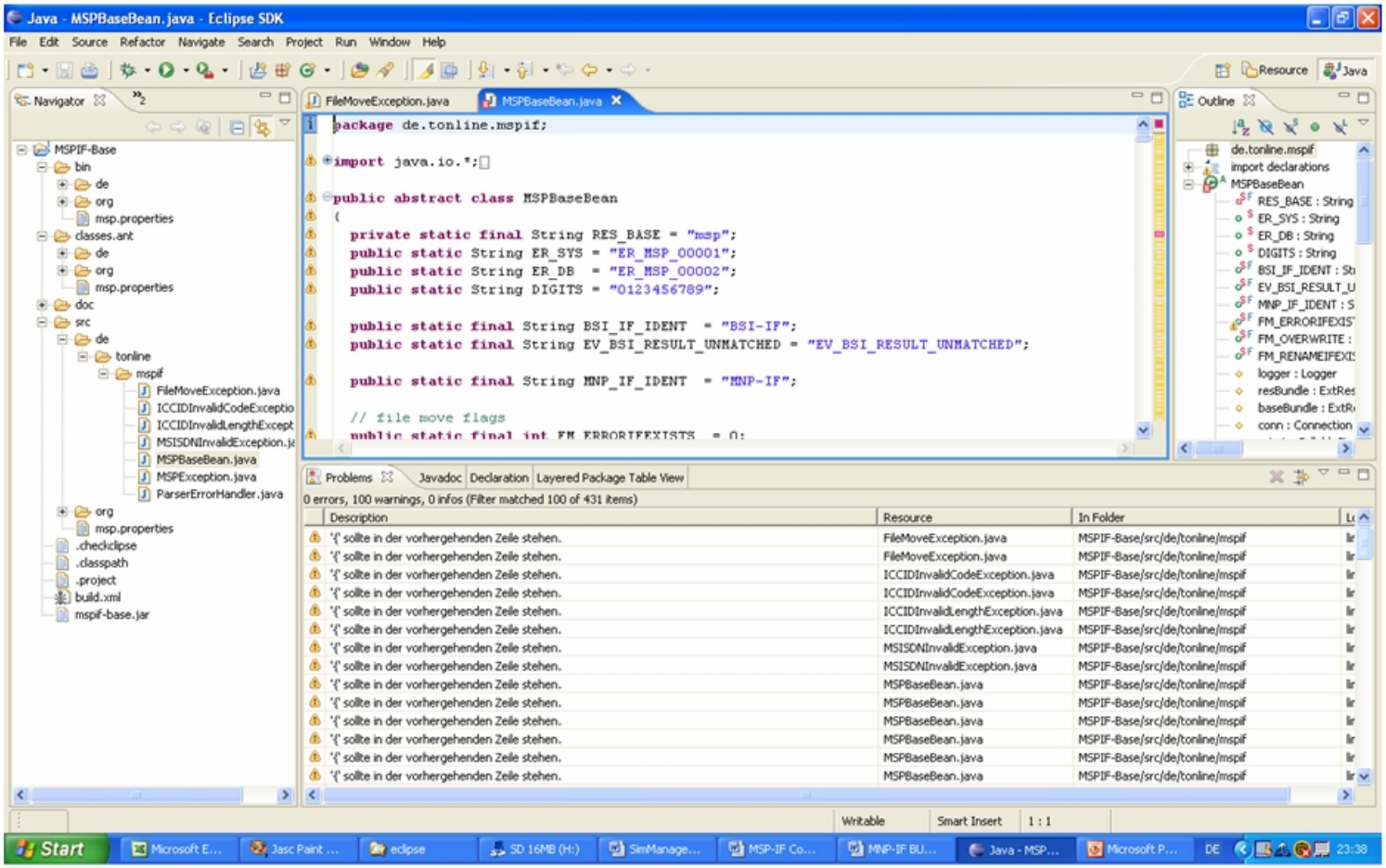
Testaktivitäten



Verfahren und Methoden

- Statischer Programmtest (Sichtung des Quellcodes)
  - Checklistenverfahren
  - Code-Reviews
- Dynamischer Programmtest (Ausführung des Quellcodes)
  - Black-Box-Test (typ. aus Sicht des Anwenders)
  - White-Box-Test (typ. aus Sicht des Entwicklertest)

Unterstützung des statischen Tests



Quelle: RSM – Resource Standard Metrics, M Squared Technologies LLC 2006
URL: http://msquaredtechnologies.com

White-Box-Test (Entwicklersicht)

- Kontrollflußorientierte Strukturtestverfahren
  - Anweisungsüberdeckungstest (statement coverage – C_0)
  - Zweigüberdeckungstest (branch coverage – C_1)
  - Pfadüberdeckungstest (path coverage – C_2)
  - Bedingungsüberdeckungstest (condition coverage – C_3)
- Datenflußorientierte Strukturtestverfahren
  - Wertzuweisung – Definitionen (def)
  - Berechnung von Werten (c-use)
  - Bildung von Wahrheitswerten (p-use)

Testabdeckung – coverage level

- Anweisungsüberdeckungstest, jede Anweisung (d.h. alle Knoten des Kontrollflussgraphen) wird einmal ausgeführt.
- Zweigüberdeckungstest, jeder Zweig (d.h. alle Kanten des Kontrollflussgraphen) wird einmal durchlaufen.
- Pfadüberdeckungstest, jeder Pfad im Kontrollflussgrafen muss einmal durchlaufen werden.
- Der Überdeckungsgrad (theoretisch: coverge level c0 bis c7) gibt an, wie viele Anweisungen / Zweige / Pfade durchlaufen wurden.

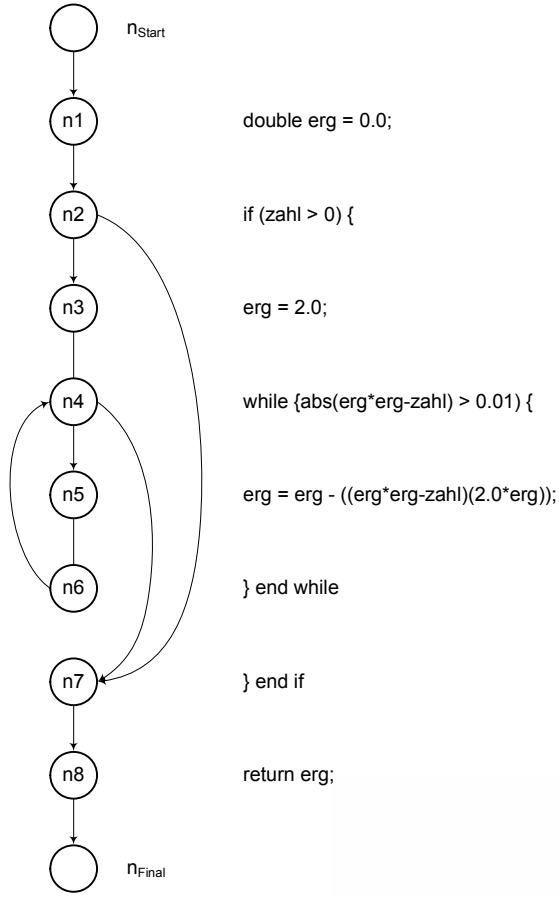
Aufgabenstellung

Gegeben ist das folgende Programm. Es berechnet für nicht negative reelle Radikanden die reelle Quadratwurzel, die als Funktionswert zurückgeliefert wird. Werden negative Werte übergeben, so wird als Ergebnis der Wert 0 zurückgegeben.

```
public static double getWurzel(double zahl){
    double erg = 0.0;
    if (zahl > 0){
        erg = 2.0;
        while (Math.abs(erg * erg - zahl) > 0.000001){
            erg = erg - ((erg * erg - zahl)/(2.0 * erg));
        } // end while
    } //end if
    return erg;
}
```

Überführung in einen Kontrollflussgrafen

```
public static double getWurzel(double zahl){
    double erg = 0.0;
    if (zahl > 0){
        erg = 2.0;
        while (Math.abs(erg * erg - zahl) > 0.000001){
            erg = erg - ((erg * erg - zahl)/(2.0 * erg));
        } // end while
    } //end if
    return erg;
}
```

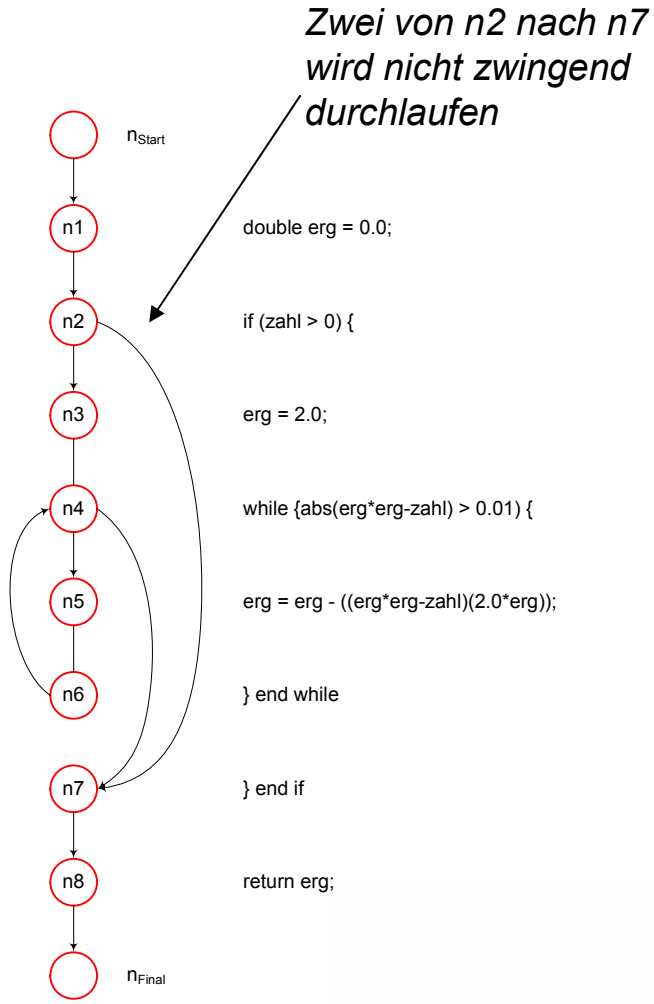


Anweisungsüberdeckungstest

C_0-Test

```
public static double getWurzel(double zahl){
    double erg = 0.0;
    if (zahl > 0){
        erg = 2.0;
        while (Math.abs(erg * erg - zahl) > 0.000001){
            erg = erg - ((erg * erg - zahl)/(2.0 * erg));
        } // end while
    } //end if
    return erg;
}
```

Kurzübung: Welcher Testfall stellt für die gegebene Beispielprozedur einen Anweisungsüberdeckungstest dar? Wie sieht der dabei durchlaufene Pfad aus?

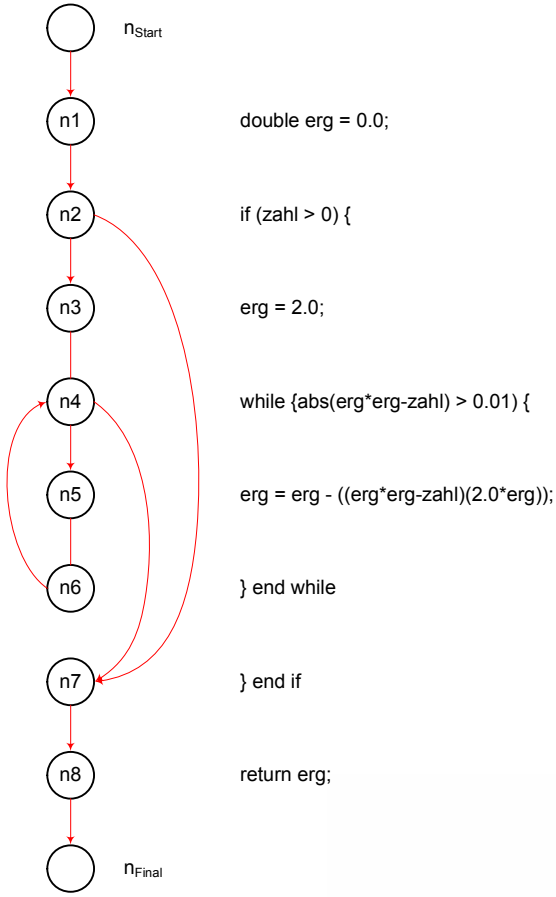


Zweigüberdeckungstest

C_1-Test

```
public static double getWurzel(double zahl){
    double erg = 0.0;
    if (zahl > 0){
        erg = 2.0;
        while (Math.abs(erg * erg - zahl) > 0.000001){
            erg = erg - ((erg * erg - zahl)/(2.0 * erg));
        } // end while
    } //end if
    return erg;
}
```

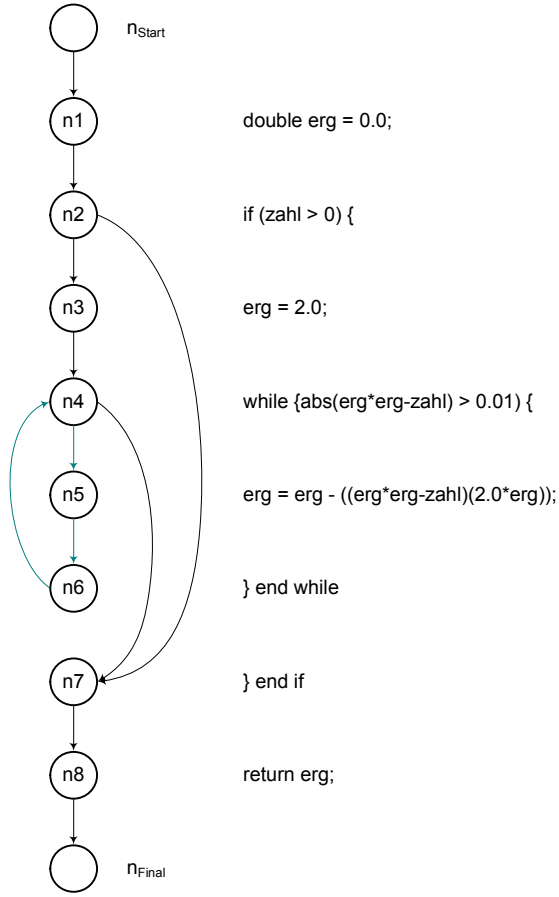
Kurzübung: Welcher Testfall stellt hier einen Zweigüberdeckungstest dar? Wie sieht der dabei durchlaufene Pfad aus?



Pfadüberdeckungstest

C_2-Test

```
public static double getWurzel(double zahl){
    double erg = 0.0;
    if (zahl > 0){
        erg = 2.0;
        while (Math.abs(erg * erg - zahl) > 0.000001){
            erg = erg - ((erg * erg - zahl)/(2.0 * erg));
        } // end while
    } //end if
    return erg;
}
```



Bem.: Programme der Wirtschaftinformatik verfügen über unendlich viele Pfade, daher schränkt man die möglichen Schleifendurchläufe S ein. Unterschieden werden z.B. folgende Pfadüberdeckungstests:
- C_2b (boundary mit S <=2)
- C_2c (constraint mit S = n_{def})

Bewertung

- Anweisungsüberdeckung (alleine nicht ausreichend)
  - Kann Code finden, der nicht ausführbar ist
  - Als eigenständiges Testverfahren nicht geeignet
  - Fehleridentifizierungsquote: 18%
- Zweigüberdeckung (minimales Testkriterium)
  - kann nicht ausführbare Programmzweige finden
  - kann häufig durchlaufene Programmzweige finden
  - Fehleridentifikationsquote: 34%
- Pfadüberdeckung (geringe Praxisrelevanz)
  - Mächtigstes kontrollstrukturiertorientiertes Testverfahren
  - kann häufig durchlaufene Programmzweige finden
  - Fehleridentifikationsquote: 64%

Anweisungsüberdeckungstest

Geben Sie für die Java-Methode `getPI(…)` - Berechnung von Pi nach der Leibnitz-Reihe - entsprechende Testfälle an, um der Kundenanforderung zum Nachweis eines Anweisungsüberdeckungs-testes zu genügen. Führen Sie die Testfälle im Rahmen eines Java-Programms aus und vergleichen Sie programmtechnisch die Ergebnisse mit denen der Spezifikation.

```
public static double getPI_A(int durchlauf) {
    double pi    = 1;
    double eins = 1;
    int nenner   = 1;

    for (int i = 1; i <= durchlauf; i++) {
        nenner = nenner + 2;
        if ((i % 2) == 1) { // ist die Zahl gerade?
            pi = pi - (eins/nenner);
        } else {
            pi = pi + (eins/nenner);
        }
    }
    //Rückgabe des Ergebnisse
    return 4 * pi;
}
```

Blackbox-Stapeltest

- Im Stapeltest wird erwartetes Verhalten mit dem tatsächlichen Verhalten einer Methode verglichen. Bei der Aufrufsituation kann es sich um zwei Alternativen handeln:
  - Ein normales Verhalten der Methode ist möglich
  - Es handelt sich um eine Ausnahme bzw. fehlerhaftes Verhalten
- Im ersten Fall wird der von der Methode gelieferte Wert (z.B. return-Wert) mit dem erwarteten Wert verglichen. Stimmen beide überein, kann der Stapeltest fortgeführt werden.
- Im zweiten Fall kommt die Behandlung auf die mögliche Reaktion der Methode an:
  - Fall 1: Die Methode liefert einen Return-Fehlerwert zurück. Dieser kann mit dem erwarteten Fehlerwert verglichen werden. (z.B. Verwendung einer if-Anweisung)
  - Fall 2: Die Methode wirft eine Ausnahme. Hier muß ein try...catch-Block eingesetzt werden. Da es sich um eine provozierte Ausnahme handelt, wird im catch-Anweisungsblock nichts gemacht. Wird jedoch keine Ausnahme geworfen, muss nach dem Methodenaufruf eine Fehlermeldung im try-Block generiert werden.

Blackbox-Stapeltest

- Fehlerprüfungen innerhalb einer Methode sind vor jeglicher Zustandsänderung durchzuführen
- Wenn ein Fehler erkannt wurde, ist die Methodenausführung abzubrechen, z.B. durch Auslösen einer Ausnahme (keine Fehlerunterdrückung!)
- Beispiel:

```
public static double getSQRT(double radikant)throws ArithmeticException{
    if (radikant<0) {
        throw new ArithmeticException();
    }
    return Math.sqrt(radikant);
}
```

- Eine aufrufende Routine kann den Fehler mit einem try...catch-Block fangen:

```
try{
    getSQRT(-1); // Ausnahme wird provoziert
}catch(ArithmeticException e){
    System.out.println("keine negativen Wurzeln" + e.getMessage());
}
```

Übung - Blackbox-Stapeltest

Realisieren Sie für die im Transferbereich gegebene Klasse Stack einen Blackbox Stapeltest. Gehen Sie dabei wie folgt vor:

- Legen Sie einen Stack der Größe 5 entsprechend folgender Notation an:
  `Stack myStack = new Stack(5);`
- Prüfen Sie, ob `myStack.isEmpty()` einen korrekten Wert liefert.
- Legen Sie fünf Elemente (d.h. Integerzahlen) auf den Stack `myStack.push(x)`, die Sie sich programmgestützt merken.
- Versuchen Sie eine Ausnahme zu provozieren, indem Sie ein weiteres Element auf den bereits vollständig gefüllten Stack legen.
- Prüfen Sie, ob `myStack.isEmpty()` einen korrekten Wert liefert.
- Räumen Sie den Stack nacheinander ab und überprüfen Sie anhand der gespeicherten Werte, ob `myStack.pop()` immer den korrekten Wert liefert.
- Versuchen Sie erneut eine Ausnahme zu provozieren, indem von dem leeren Stack noch ein weiteres Element abräumen.
- Prüfen Sie, ob `myStack.isEmpty()` einen korrekten Wert liefert.

Dokumentation der Tests

- Bestandteile des Testobjekts und Ihre Verwendung
- Erwartete Testausgabedaten (Soll- Ergebnisse)
- Komponenten und Schnittstellen der Testumgebung
- Beschreibung der Testausführung und der Abläufe
- Alle verwendeten Testfälle
- Menge der Testeingabedaten gleicher Wirkung
- Ausgewählte Testeingabedaten
- Beschreibung eines Fehlers mit Fehlerquelle, -ursache und -folgen

Weiterführende Literatur

- William E. P.; Randall W. R.: Die 10 goldenen Regeln des Software-Testens, verlag moderne industrie Buch AG & Co. KG, Bonn
- Spillner, A.; Linz, T.: Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester, , dpunkt-Verlag
- Ebert, C.; Dumke, R.; Bundschuh, M.; Schmietendorf, A.: Best practices in Software-Measurement, Springer-Verlag, 06/2004
- Pol, M.; Koomen, T.; Spillner, A.: Management und Optimierung des Testprozesses, dpunkt-Verlag