Vorlesung

TECHNISCHE HOCHSCHULE LÜBECK

Programmieren I und II

Unit 8

Testen (objektorientierter) Software

vor dem Hintergrund von Konzepten wie Polymorphie und Verträgen

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
1

1

Disclaimer

TECHNISCHE HOCHSCHULE LÜBECK

Zur rechtlichen Lage an Hochschulen:

Dieses Handout und seine Inhalte sind durch den Autor selbst erstellt. Aus Gründen der Praktikabilität für Studierende lehnen sich die Inhalte stellenweise im Rahmen des Zitatrechts an Lehrwerken an.

Diese Lehrwerke sind explizit angegeben.

Abbildungen sind selber erstellt, als Zitate kenntlich gemacht oder unterliegen einer Lizenz die nicht die explizite Nennung vorsieht. Sollten Abbildungen in Einzelfällen aus Gründen der Praktikabilität nicht explizit als Zitate kenntlichgemacht sein, so ergibt sich die Herkunft immer aus ihrem Kontext: „Zum Nachlesen ...“.

Creative Commons:

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
2

2

TECHNISCHE HOCHSCHULE LÜBECK

Prof. Dr. rer. nat. Nane Kratzke

Praktische Informatik und betriebliche Informationssysteme

- **Raum: 17-0.10**
- **Tel.:** 0451 300 5549
- **Email:** nane.kratzke@th-luebeck.de

@NaneKratzke

Updates der Handouts auch über Twitter #prog_inf und #prog_itd

3

Units

TECHNISCHE HOCHSCHULE LÜBECK

Unit 1 Einleitung und Grundbegriffe	**Unit 2** Grundelemente imperativer Programme	**Unit 3** Selbstdefinierbare Datentypen und Collections	**Unit 4** Einfache I/O Programmierung
Unit 5 Rekursive Programmierung und rekursive Datenstrukturen	**Unit 6** Einführung in die objektorientierte Programmierung und UML	**Unit 7** Konzepte objektorientierter Programmiersprachen	**Unit 8** Testen (objektorientierter) Programme
Unit 9 Generische Datentypen	**Unit 10** Objektorientierter Entwurf und objektorientierte Designprinzipien	**Unit 11** Graphical User Interfaces	**Unit 12** Multithread Programmierung

4

Abgedeckte Ziele dieser UNIT

TECHNISCHE HOCHSCHULE LÜBECK

Kennen existierender Programmierparadigmen und Laufzeitmodelle	Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)	Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenztypen (insbesondere Liste, Stack, Mapping)	Verstehen des Unterschieds zwischen Werte- und Referenzsemantik
Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen	Kennen des Algorithmus-begriffs, Implementieren einfacher Algorithmen	Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung	Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)
Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)	Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software	Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfsprinzipien	Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Mulithread Programmierung sowie funktionale Programmierung

Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
5

5

Themen dieser Unit

TECHNISCHE HOCHSCHULE LÜBECK

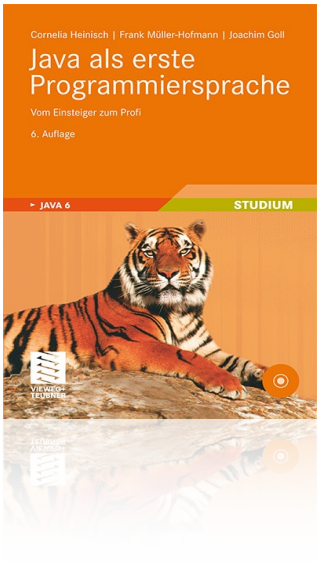
Polymorphie	Verträge	Unit Testing und Code Coverage
• Liskovsches Substitutionsprinzip (LSP) • Run Time Type Identification (RTTI)	• Vorbedingung • Nachbedingung • Klasseninvariante	• JUnit • C0, C1, C2

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
6

6

Zum Nachlesen ...

TECHNISCHE HOCHSCHULE LÜBECK



Kapitel 11

Vererbung und Polymorphie

11.4 Polymorphie und das Liskovsche Subsitutionsprinzip

11.5 Verträge

7

Polymorphie

TECHNISCHE HOCHSCHULE LÜBECK

Polymorphie ist neben Vererbung ein weiterer wichtiger Aspekt des objektorientierten Ansatzes. Polymorphie bedeutet Vielgestaltigkeit. Polymorphie gibt es bei Operationen (Methoden) und Objekten

Polymorphie von Operationen

- Eine Operation in verschiedenen Klassen wird durch jeweils eine eigene Methode mit demselben Namen implementiert.
- Hinter ein und demselben Namen einer Methode verbergen sich bei unterschiedlichen Klassen also unterschiedliche Methoden.
- Z.B. print() bei Personen und print() bei Studenten (zusätzliche Ausgabe der Matrikelnummer).

Polymorphie von Objekten

- Gibt es nur in Vererbungshierarchien
- An die Stelle eines Objektes einer Klasse in einem Programm kann auch ein anderes Objekt einer abgeleiteten Klasse treten, solange
  - die abgeleitete Klasse die Basisklasse nur erweitert,
  - die abgeleitete Klasse die Verträge überschriebener Methoden einhält und
  - die abgeleitete Klasse die Klasseninvarianten der Basisklasse erfüllt.

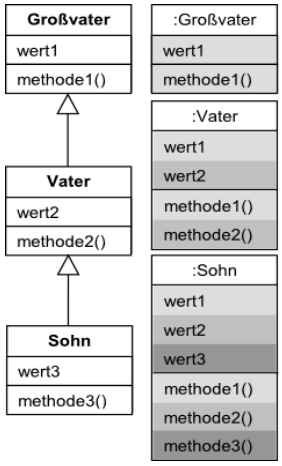
8

Polymorphes Verhalten bei der Erweiterung von Klassen

TECHNISCHE HOCHSCHULE LÜBECK

- Ein Objekt einer Unterklasse kann auch Methodenaufrufe einer Basisklasse beantworten
- Es verhält sich an dieser Stelle wie ein Objekt einer Basisklasse
- Es kann also in verschiedenen Gestalten auftreten
- Ein Sohn-Objekt, kann auch als Vater-Objekt, als Großvater-Objekt, als Urgroßvater-Objekt, etc. auftreten
- Es ist also vielgestaltig

- **Anders ausgedrückt: Objekte abgeleiteter Klassen können überall dort genutzt werden, wo auch Objekte der Basisklasse eingesetzt werden**
- **Dies ist der Grund, dass sich ganze Klassenbibliotheken problemlos wiederverwenden lassen**
- **Ein Sortieralgorithmus auf Personen funktioniert auch für von Personen abgeleitete Studenten.**



Quelle: Java als erste Programmiersprache

9

Liskovsches Subsititutionsprinzip

TECHNISCHE HOCHSCHULE LÜBECK

Prof. Barbara Liskov, Turing Award 2008, MIT

Quelle: Wikipedia

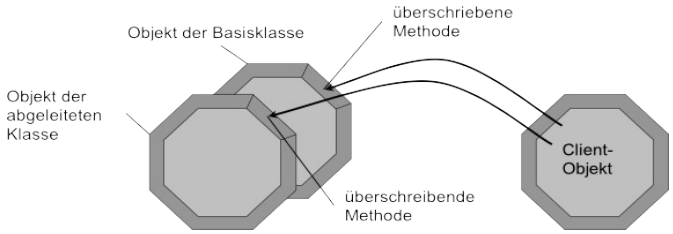
Dieser Sachverhalt der Objektorientierung ist auch als **Liskovsches Subsitution Principle (LSP)** bekannt.

LSP bei Erweiterung	LSP bei Überschreiben
• Objekte erweiterter Klassen können an die Stelle eines Objekts der Basisklasse treten (Student statt Person)	• Die überschriebenen Methoden müssen die Verträge der Basisklasse einhalten. • Darauf gehen wir im Anschluss noch ein.

10

Liskovsches Subsititutionsprinzip (II)

TECHNISCHE HOCHSCHULE LÜBECK



Quelle: Java als erste Programmiersprache

Polymorphie erlaubt es so – dank LSP – große Mengen generalisierten Code in Basisklassen zu implementieren, der dann später von Objekten beliebiger abgeleiteter Klassen benutzt werden kann. Zum Zeitpunkt der Entwicklung der Basisklassen ist noch nicht bekannt, welche Klassen zu späteren Zeitpunkten abgeleitet werden.

Der instanceof-Operator

TECHNISCHE HOCHSCHULE LÜBECK

- Mit dem instanceof-Operator kann zur Laufzeit getestet werden, ob eine Referenz auf ein Objekt eines bestimmten Typs zeigt.

```
a instanceof Klassenname
```

- Gibt true zurück, wenn a den Typ Klassenname hat

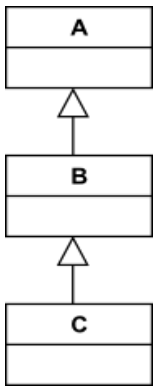
```
null instanceof Klassenname
```

- zeigt auf kein Objekt und ist daher immer false.

Der instanceof-Operator Beispiele:

TECHNISCHE HOCHSCHULE LÜBECK

```
Object a = new A();
Object b = new B();
Object c = new C();
```




```
a instanceof A
b instanceof A
c instanceof A
```

```
true
true
true
```

```
a instanceof B
b instanceof B
c instanceof B
```

```
false
true
true
```

```
a instanceof C
b instanceof C
c instanceof C
```

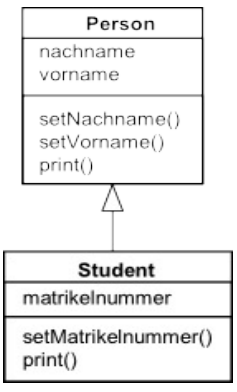
```
false
false
true
```


13

Run Time Type Identification (RTTI)

TECHNISCHE HOCHSCHULE LÜBECK

- RTTI ist die Erkennung eines Typs zur Laufzeit und wird für Polymorphie vom Laufzeitsystem benötigt.
- Dieses Prinzip soll am Bsp. Person und Student verdeutlicht werden.

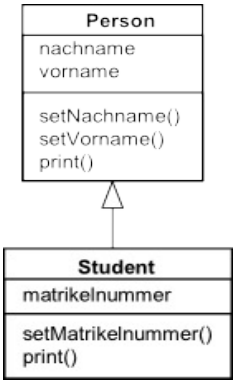


Quelle: Java als erste Programmiersprache

14

RTTI am Beispiel (I)

TECHNISCHE HOCHSCHULE LÜBECK



Quelle: Java als erste Programmiersprache

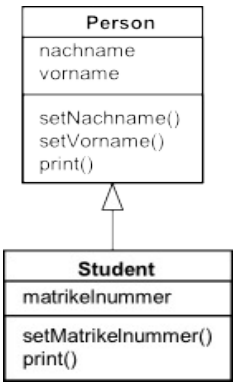
```
class Person {
  private String vorname;
  private String nachname;

  public void setNachname(String nn) { nachname = nn; }

  public void setVorname(String vn) { vorname = vn; }

  public void print() {
    System.out.print(„Name: “);
    System.out.println(vorname + „ “ + nachname);
  }
}
```

```
class Student extends Person {
  private int matrikelnummer;

  public void setMatrikelnummer(int mn) {
    matrikelnummer = mn;
  }

  public void print() {
    super.print();
    System.out.print(„Matrikelnummer: “);
    System.out.println(matrikelnummer);
  }
}
```


15

RTTI am Beispiel (II)

TECHNISCHE HOCHSCHULE LÜBECK




```
Person p = new Student();
p.setNachname(„Mustermann“);
p.setVorname(„Max“);
p.print();
```

Wie lautet also die Ausgabe?

```
Name: Max Mustermann
// Aufruf der Person.print() Methode
```

oder

```
Name: Max Mustermann
Matrikelnummer: 0
// Aufruf der Student.print() Methode
```

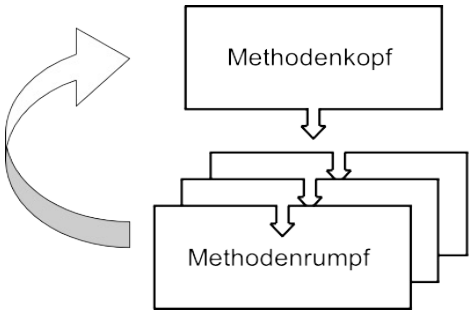
!!! Wichtig !!! RTTI stellt sicher, dass immer die Methode des **Objekttyps** nicht des **Referenztyps** aufgerufen wird. In diesem Fall also die Methode des Studenten und nicht der Person.

16

Statische und dynamische Bindung von Methoden

TECHNISCHE HOCHSCHULE LÜBECK

- Zuordnung eines Methodenrumpfs zu einem Methodenkopf (Signatur)
- Frühe Bindung erfolgt bereits zum Kompilierzeitpunkt
- Späte Bindung erfolgt zur Laufzeit



Quelle: Java als erste Programmiersprache

In JAVA hat man keinen direkten Einfluss darauf, ob spät oder früh gebunden wird. Aufgrund der Polymorphie kann erst zur Laufzeit entschieden werden, zu welcher Klasse ein Objekt gehört und welcher Methodenrumpf daher auszuführen ist. In Ausnahmefällen kann jedoch der Methodenrumpf bereits zur Kompilierzeit bestimmt werden – dies wird dann aus Performancegründen auch getan.

Dynamische Bindung von Methoden in JAVA

TECHNISCHE HOCHSCHULE LÜBECK

- An jeder Stelle an der ein Objekt einer Basisklasse verlangt wird, kann gem. LSP auch ein Objekt einer abgeleiteten Klasse eingesetzt werden.
- Daher wird grundsätzlich bei Aufruf von Instanzmethoden die SPÄTE BINDUNG durchgeführt.
- Zeigt so eine Referenz vom Typ einer Basisklasse auf ein Typ einer Subklasse, wird so immer die überschreibende Methode aufgerufen.
- Dies kann jedoch erst zur Laufzeit festgestellt werden, außer in den folgenden Ausnahmen:

Statische Bindung von Methoden in JAVA

TECHNISCHE HOCHSCHULE LÜBECK

`private`	`final`	`static`
• Methoden die als private gekennzeichnet sind, sind nur innerhalb der Klasse zugreifbar. • Polymorphie kann bei diesen daher nicht erfolgen, sie werden daher statisch, d.h. bereits zur Kompilierzeit gebunden.	• Methoden die als final gekennzeichnet sind, können in abgeleiteten Klassen nicht verändert werden. • Polymorphie kann bei diesen daher nicht erfolgen, sie werden daher statisch, d.h. bereits zur Kompilierzeit gebunden.	• Methoden die als static gekennzeichnet sind, sind Klassenmethoden und gehören zu einer Klasse. • Polymorphie kann bei diesen daher nicht erfolgen, sie werden daher statisch, d.h. bereits zur Laufzeit gebunden.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
19

19

Themen dieser Unit

TECHNISCHE HOCHSCHULE LÜBECK

Polymorphie	Verträge	Unit Testing und Code Coverage
• Liskovsches Substitutionsprinzip (LSP) • Run Time Type Identification (RTTI)	• Vorbedingung • Nachbedingung • Klasseninvariante	• JUnit • C0, C1, C2

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
20

20

Verträge

TECHNISCHE HOCHSCHULE LÜBECK

- Design by Contract wurde von Betrand Meyer als Entwurfstechnik eingeführt
- Eine Klasse besteht nicht nur aus Methoden und Datenfeldern, eine Klasse wird auch benutzt
- Design by Contract sieht diese Beziehungen als eine formale Übereinkunft zwischen beliebigen Partnern
- Beim Aufruf einer Methode muss sich der Aufrufer einer Methode und die aufgerufene Methoden sich gegenseitig aufeinander verlassen können
  - Aufrufer muss Vorbedingungen einhalten
  - Die aufgerufene Methode muss daraufhin Nachbedingungen sicherstellen und den inneren Zustand eines Objekts gültig lassen

Prof. Bertrand Meyer, ETH Zürich, Erfinder von Eiffel, System Software Award 2007 , Wikipedia

21

Zusicherungen

TECHNISCHE HOCHSCHULE LÜBECK

Verträge werden mittels sogenannter **Zusicherungen** formal spezifiziert.

Eine Zusicherung ist ein boolescher Ausdruck, der niemals falsch werden darf.

- **Design by Contract** verwendet drei verschiedene Arten von Zusicherungen
  - Vorbedingungen
  - Nachbedingungen
  - Invarianten

Quelle: Pixabay

22

Zusicherungen

Prof. Hoare, Oxford University, Turing Award 1980, Wikipedia

$\{P\}\ A\ \{Q\}$ (Hoare Triple)

Der Vertrag einer Methode A, umfasst die Vor- und Nachbedingungen.

Vorbedingung P (Precondition)

- Einschränkungen unter denen eine Methode funktioniert
- Beispiel:
  - push() darf auf einem Stack nicht aufgerufen werden, wenn dieser voll ist
  - pop() darf auf einem Stack nicht aufgerufen, wenn dieser leer ist

Nachbedingung Q (Postcondition)

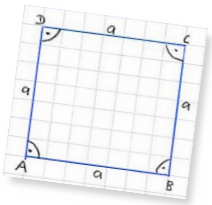
- Korrekter Zustand nachdem Aufruf einer Methode
- Bsp.:
  - Nach dem Aufruf von push() kann ein Stack nicht leer sein und muss ein Element mehr umfassen
  - Nach dem Aufruf von pop() kann ein Stack leer sein und muss ein Element weniger umfassen

23

Invarianten

TECHNISCHE HOCHSCHULE LÜBECK

- Eine **Invariante** bezieht sich nicht auf eine Methode, sondern **Objekte einer Klasse**.
- Invarianten „sichern“ den korrekten Zustand eines Objekts.
- Da **Invarianten von allen Methoden** einer Klasse **eingehalten** werden müssen, spricht man auch von Klasseninvarianten.
- Eine **Klasseninvariante** muss **vor** und **nach** dem **Aufruf** einer nach **außen sichtbaren Methode** gültig sein.



- **Beispiel einer Invariante für eine Klasse Quadrat:**
  - Alle Seiten des Quadrats sind gleich lang.
  - Es gibt vier Winkel mit genau 90 Grad.
  - Egal welche Methode aufgerufen wird, diese Bedingung muss immer gelten.

24

Vertrag einer Klasse

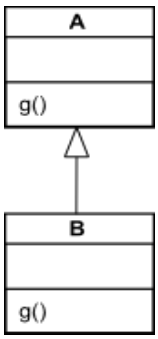
TECHNISCHE HOCHSCHULE LÜBECK

Solche Verträge dürfen durch abgeleitete Klassen nicht verletzt werden, daher müssen beim Überschreiben der Methoden bestimmte Regeln eingehalten werden!

25

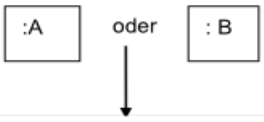
Regeln für das Einhalten von Methodenverträgen

TECHNISCHE HOCHSCHULE LÜBECK



Die Methode f() akzeptiert Parameter des Typs A und damit implizit auch aller Subklassen von A (also auch B).

Zur Laufzeit




```
void f (A a)
{
    . . .
    ergebnis = a.g();
    . . .
}
```

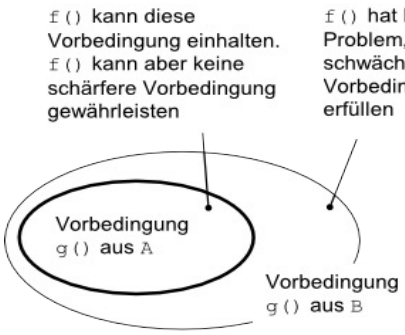
f() kann aber nur die für die Klasse A gemachten Verträge kennen und berücksichtigen. Diese müssen daher von allen von A abgeleiteten Klassen eingehalten werden!

26

Aufweichen von Vorbedingungen

TECHNISCHE HOCHSCHULE LÜBECK

Es gilt die Regel: Eine überschreibende Methode darf eine Vorbedingung aufweichen, aber niemals verschärfen!!!



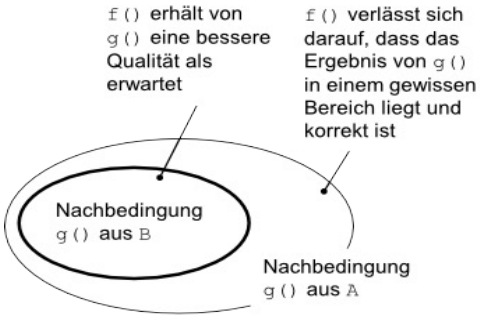
Quelle: Java als erste Programmiersprache

Beispiel: Wenn die ursprüngliche Methode einen Definitionsbereich von 1 bis 10 akzeptiert, dann darf die überschreibende Methode, den Definitionsbereich nicht reduzieren (z.B. 2 bis 9) wohl aber erweitern, z.B. (0 bis 100).

Verschärfen von Nachbedingungen

TECHNISCHE HOCHSCHULE LÜBECK

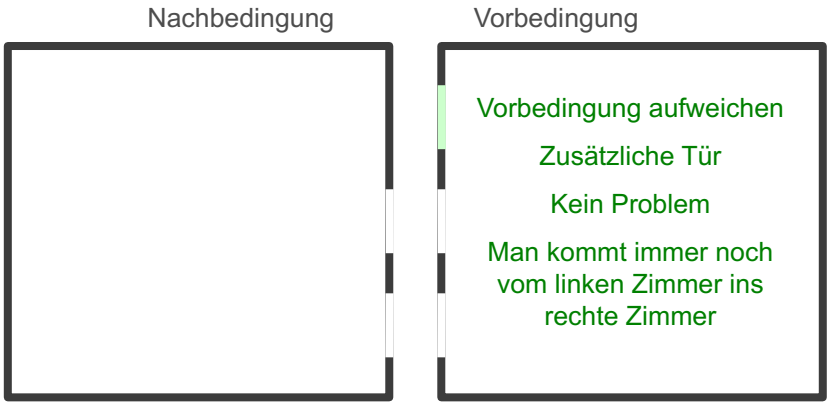
Es gilt die Regel: Eine überschreibende Methode darf eine Nachbedingung verschärfen, aber niemals aufweichen!!!



Beispiel: Wenn die ursprüngliche Methode einen Wertebereich von 1 bis 10 als Ergebnis liefert, dann darf die überschreibende Methode, den Wertebereich reduzieren (z.B. 2 bis 9) aber nicht erweitern, z.B. (0 bis 100).

Vor- und Nachbedingungen
Vorbedingungen aufweichen

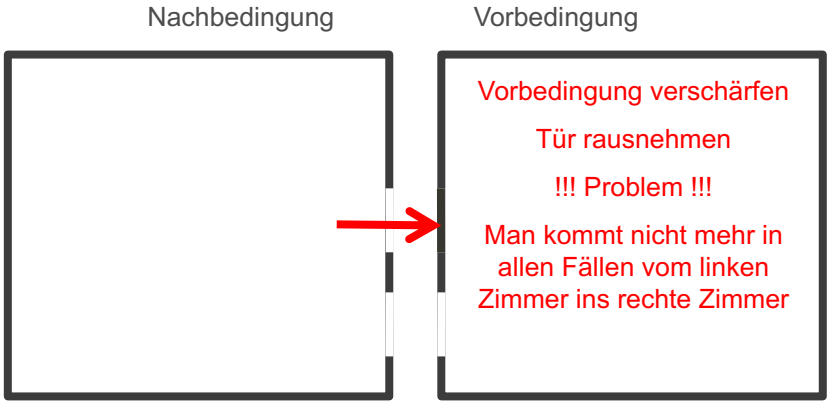
TECHNISCHE HOCHSCHULE LÜBECK



29

Vor- und Nachbedingungen
Vorbedingungen verschärfen

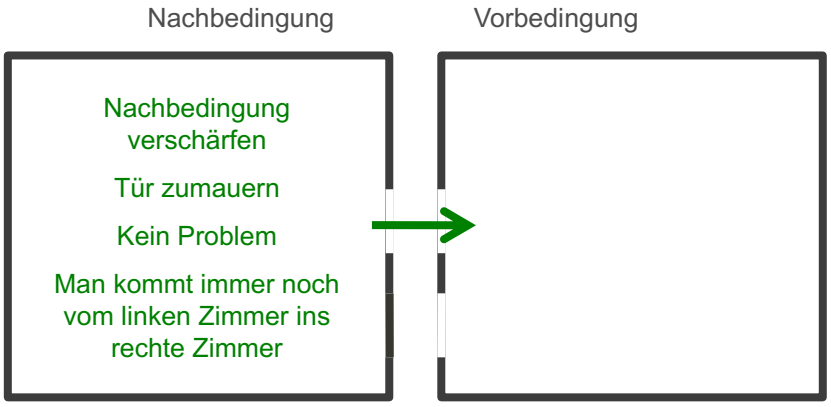
TECHNISCHE HOCHSCHULE LÜBECK



30

Vor- und Nachbedingungen
Nachbedingungen verschärfen

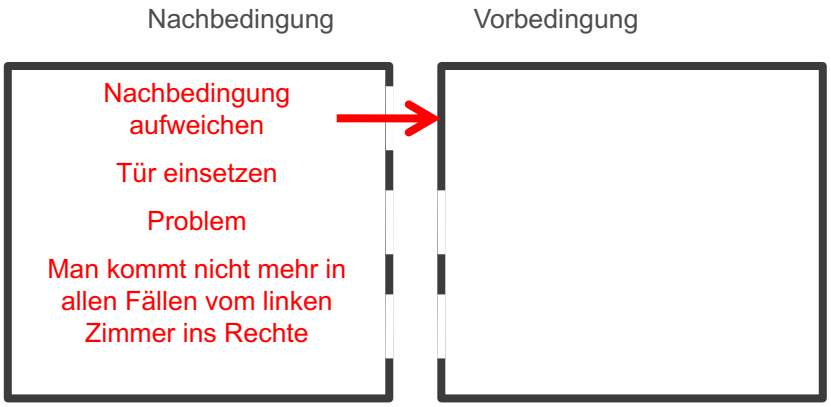
TECHNISCHE HOCHSCHULE LÜBECK



31

Vor- und Nachbedingungen
Nachbedingungen aufweichen

TECHNISCHE HOCHSCHULE LÜBECK



32

Klassen als Übergabe- und Rückgabetypen

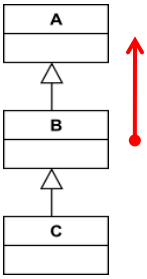
TECHNISCHE HOCHSCHULE LÜBECK

Aufweichung der Vorbedingung entspricht:

- In überschriebenen Methoden
- ist bei Aufruf anderer Methoden
- bei Übergabeparametern nur eine **Generalisierung** erlaubt
- `? super Type` (Lower Bound Wildcard)

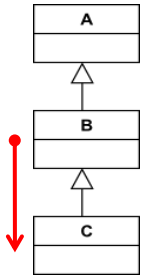
Verschärfung der Nachbedingung entspricht:

- In überschriebenen Methoden
- ist bei Rückgabetypen nur eine **Spezialisierung**
- bei return Statements erlaubt
- `? extends Type` (Upper Bound Wildcard)




```
void func(B par);

A a = new A();
B b = new B();
C c = new C();

Dann geht:
func(a);
func(b);

Aber (eigentlich) nicht:
func(c);
// Da C spezieller als B
// JAVA führt in diesen
Fällen automatisch einen
impliziten Upcast durch
```

```
B func();

A a = new A();
B b = new B();
C c = new C();

Dann geht:
return b;
return c;

Aber nicht:
return a;
// Da A genereller als B
```



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
33

33

Regeln für das Einhalten von Klasseninvarianten

TECHNISCHE HOCHSCHULE LÜBECK

- Beim Erweitern muss darauf geachtet werden, dass die von ihr abgeleiteten Klassen, die Gültigkeit der Klasseninvarianten der Basisklasse nicht verletzen.

Die Invarianten einer Klasse ergeben sich aus der logischen UND Verknüpfung der in ihr definierten Invarianten und der Invarianten, die in der/den Vaterklasse(n) definiert sind.

Beispiel: In der Klasse Polygon gilt die Invariante: Ein Polygon hat mindestens drei Punkte. Aus Polygon ist die Klasse Rechteck abgeleitet worden mit der zusätzlichen Invarianten: Ein Rechteck hat genau vier Punkte und vier Winkel mit jeweils 90 Grad.

Alle Invarianten die durch Objekte der Klasse Rechteck einzuhalten wären, sind also:

- Ein Objekt der Klasse Rechteck hat mindestens drei Punkte. UND
- Ein Objekt der Klasse Rechteck hat genau vier Punkte. UND
- Ein Objekt der Klasse Rechteck hat vier Winkel mit jeweils 90 Grad.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
34

34

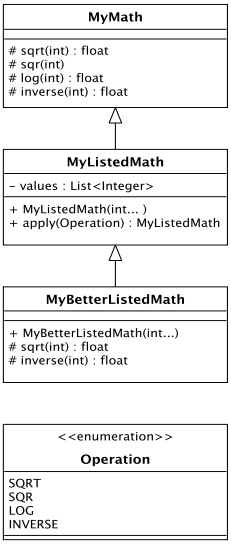
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Gegeben ist die Klasse `MyMath`:

```
public class MyMath {

  protected float sqrt(int v) {
    return (float) Math.sqrt(v);
  }

  protected int sqr(int v) throws MyMathException {
    if (v > Math.sqrt(Integer.MAX_VALUE))
      throw new MyMathException("...");
    if (v < -Math.sqrt(Integer.MAX_VALUE))
      throw new MyMathException("...");

    return v * v;
  }

  protected float log(int v) {
    return (float) Math.log10((double) v);
  }

  protected float inverse(int v) {
    return (float) 1.0 / v;
  }
}
```



35

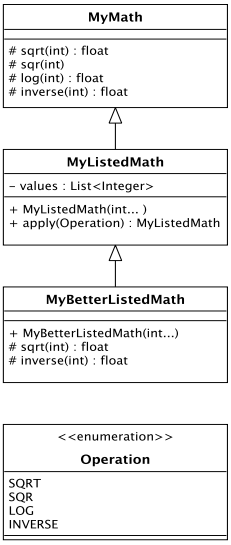
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Gegeben ist die Klasse `MyListedMath`:

```
public class MyListedMath extends MyMath {

  private List<Integer> values = new LinkedList<Integer>();

  public MyListedMath(int... vs) {
    for (int v : vs) { this.values.add(v); }
  }

  public MyListedMath apply(Operation op) {
    List<Integer> ret = new LinkedList<Integer>();
    for (int v : this.values) {
      try {
        switch (op) {
          case SQR: ret.add((int) sqr(v)); break;
          case SQRT: ret.add((int) sqrt(v)); break;
          case LOG: ret.add((int) log(v)); break;
          case INVERSE: ret.add((int) inverse(v)); break;
        }
      } catch (MyMathException ex) { System.out.println(ex.getMessage()); }
    }
    this.values = ret;
    return this;
  }

  public String toString() {
    return this.values.toString();
  }
}
```



36

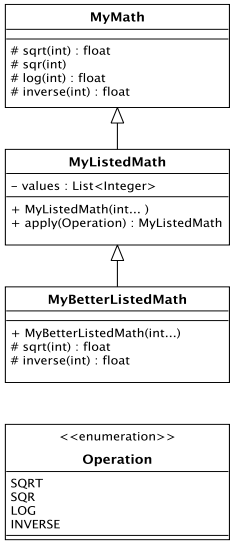
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Gegeben ist die Klasse `MyBetterListedMath`:

```
public class MyBetterListedMath extends MyListedMath {

  public MyBetterListedMath(int... vs) {
    super(vs);
  }

  protected float sqrt(int v) {
    if (v < 0) throw new ArithmeticException("...");
    return (float) Math.sqrt(v);
  }

  protected float inverse(int v) {
    if (v == 0) throw new ArithmeticException("...");
    return (float) 1.0 / v;
  }
}
```



37

Miniübung:



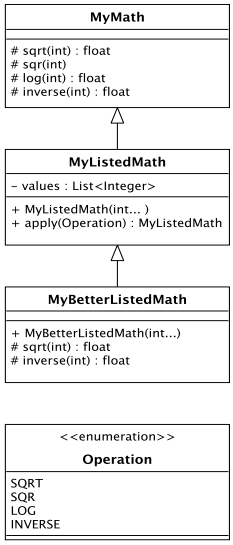
TECHNISCHE HOCHSCHULE LÜBECK

Der Aufruf von

```
MyListedMath values = new MyListedMath(
  Integer.MAX_VALUE / 1000000,  Integer.MAX_VALUE / 100000,
  Integer.MAX_VALUE / 10000,    Integer.MAX_VALUE / 1000,
  Integer.MAX_VALUE / 100,      Integer.MAX_VALUE / 10
);
System.out.println(values.apply(Operation.SQR).apply(Operation.INVERSE)
                         .apply(Operation.LOG).apply(Operation.SQRT)
);
```

führt zu folgender (beabsichtigten) Ausgabe:

```
sqr(214748) is undefined.
sqr(2147483) is undefined.
sqr(21474836) is undefined.
sqr(214748364) is undefined.
[0, 0]
```



38

Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

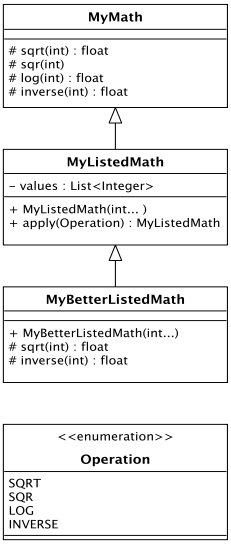
Der Aufruf von

```
MyListedMath better = new MyBetterListedMath(
  Integer.MAX_VALUE / 1000000,  Integer.MAX_VALUE / 100000,
  Integer.MAX_VALUE / 10000,    Integer.MAX_VALUE / 1000,
  Integer.MAX_VALUE / 100,      Integer.MAX_VALUE / 10
);
System.out.println(better.apply(Operation.SQR).apply(Operation.INVERSE)
                         .apply(Operation.LOG).apply(Operation.SQRT)
);
```

führt hingegen zu folgender (unbeabsichtigten) Ausgabe:

```
Exception in thread "main" java.lang.ArithmeticException: sqrt (-2147483648) not defined.
at MyBetterListedMath.sqrt(MyBetterListedMath.java:9)
at MyListedMath.apply(MyListedMath.java:23)
at MyListedMath.main(MyListedMath.java:71)
```

Warum?



39

Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Antwort: Weil Verträge nicht eingehalten worden sind!

```
public class MyMath {

  protected float sqrt(int v) {
    return (float) Math.sqrt(v);
  }

  protected float inverse(int v) {
    return (float) 1.0 / v;
  }
}
```

Vergrößern des Wertebereichs (Nachbedingung)!

(unerwartete Ex)

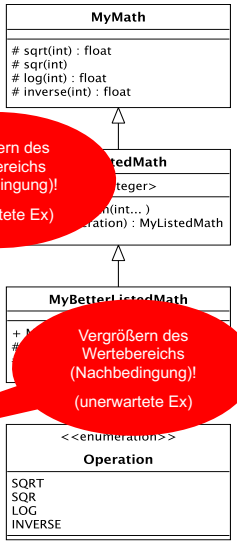
Verschärfung einer Vorbedingung!

```
class MyBetterListedMath extends MyListedMath {

  protected float sqrt(int v) {
    if (v < 0) throw new ArithmeticException("...");
    return (float) Math.sqrt(v);
  }

  protected float inverse(int v) {
    if (v == 0) throw new ArithmeticException("...");
    return (float) 1.0 / v;
```

Vergrößern des Wertebereichs (Nachbedingung)!

(unerwartete Ex)

Verschärfung einer Vorbedingung!



40

Fehler schleichen sich ein ...

TECHNISCHE HOCHSCHULE LÜBECK

Ich teste, weil ich keinem Programmierer traue.

Am wenigsten mir selber.

Quelle: Pixabay

41

Sie überschreiben eine Methode, dann dürfen Sie ...

- die Vorbedingung verschärfen
- die Vorbedingung aufweichen
- die Nachbedingung aufweichen
- die Nachbedingung verschärfen

Bildquelle: ARD

42

Verschärfen einer Bedingung bedeutet ...

- Mit generelleren Klassen zu arbeiten
- Mit spezifischeren Klasen zu arbeiten
- Abstrakte Methoden zu implementieren
- Schnittstellen zu nutzen

Bildquelle: ARD

43

43

Aufweichen einer Bedingung bedeutet u.a. ...

- Abstrakte Klassen zu nutzen
- Schnittstellen zu nutzen
- Upper Bound Wildcards zu nutzen
- Lower Bound Wildcards zu nutzen

Bildquelle: ARD

44

44

Themen dieser Unit

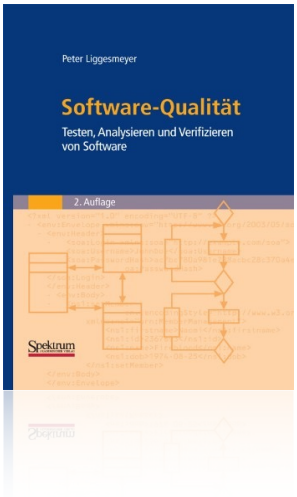
TECHNISCHE HOCHSCHULE LÜBECK

Polymorphie	Verträge	Unit Testing und Code Coverage
• Liskovsches Substitutionsprinzip (LSP) • Run Time Type Identification (RTTI)	• Vorbedingung • Nachbedingung • Klasseninvariante	• JUnit • C0, C1, C2

45

Zum Nachlesen ...

TECHNISCHE HOCHSCHULE LÜBECK



Kapitel 1 Einführung

1.1 Motivation

1.3.2 SW-Qualitätssicherung

1.4 Einordnung und Klassifikation der Prüftechniken

Kapitel 3 Kontrollflussorientierter Test

3.1 Kontrollflussorientierter Test

3.2 Anweisungsüberdeckungstest

3.3 Zweigüberdeckungstest

3.4 Bedingungsüberdeckungstest

46

Worum geht es nun?

TECHNISCHE HOCHSCHULE LÜBECK

Was ist Testen?

Code Coverage

Unit Testing

47

Warum Testen?

TECHNISCHE HOCHSCHULE LÜBECK

Nach Grounding

Jede 737 Max braucht über 100 Stunden Arbeit zum Neustart

Wenn die Behörden die Boeing 737 Max freigeben, können die Betreiber die Jets wieder startklar machen. Dafür sind erhebliche Arbeiten nötig.

30 Kommentare

28.05.19 - 6:16 | Timo Nowack

Boeing 737 Max von Southwest Airlines: 34 Jets sind bei der Fluglinie am Boden. Southwest Airlines

rem Erstflug.
die Rakete
durch die
emen
ann. Es stellte
ne 4
Anforderungen
ller als die
er Variable

Teil der
nötigt wurde
chs in letzter

370
nen der
stellt. Der
er 1997 (d.h.

48

Korrektheit
ist eine binäre Eigenschaft von Software

TECHNISCHE HOCHSCHULE LÜBECK

- Korrektheit besitzt keinen graduellen Charakter, d.h. eine Software ist entweder korrekt oder nicht.
- Eine fehlerfreie Software ist korrekt.
- Eine Software ist korrekt, wenn sie konsistent zu ihrer Spezifikation ist.

- **Existiert zu einer Software keine Spezifikation, so ist keine Überprüfung der Korrektheit möglich.**

Oder:

Ohne Spezifikation kann man keine Fehler programmieren.

Was Sie über das Testen wissen sollten!

TECHNISCHE HOCHSCHULE LÜBECK

"Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen."

Dijkstra, The Humble Programmer, ACM Turing Lecture 1972

(Original engl.: "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.")

"Wenn du effektivere Programmierer möchtest, wirst du bemerken, dass sie **keine Zeit mit debuggen** verschwenden sollten, statt dessen sollten sie keine Bugs einführen."

Dijkstra, The Humble Programmer, 1972

(Original engl.:" "If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.")

Software-Qualitätssicherung

TECHNISCHE HOCHSCHULE LÜBECK

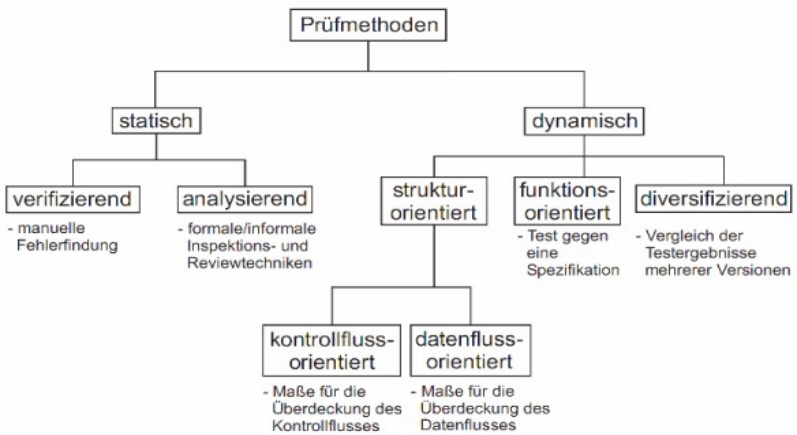
> **"Software-Qualitätssicherung stellt Techniken zur Erreichung der gewünschten Ausprägungsgrade der Qualitätsmerkmale von Software-Systemen zur Verfügung."**
>
> *Liggesmeyer, Software-Qualität, Spektrum Verlag, 2002, S. 27*

Beispiele von Qualitätsmerkmalen von SW. Wir konzentrieren uns in dieser Unit mal auf Korrektheit.

51

Klassifikation von SW-Prüftechniken

TECHNISCHE HOCHSCHULE LÜBECK



In Anlehnung an Liggesmeyer (Softwarequalitätssicherung)

52

Dynamisches Testen vs. Statische Analyse

TECHNISCHE HOCHSCHULE LÜBECK

Dynamisches Testen	Statische Analyse
• SW wird mit konkreten Eingabewerten versehen und ausgeführt • Es kann in der realen Umgebung getestet werden • Dynamische Testtechniken sind Stichprobenverfahren (Testfälle) • Durch dynamisches Testen wird nicht die Korrektheit bewiesen	• Es erfolgt keine Ausführung der zu prüfenden Software • Alle statischen Analysen können ohne Computerunterstützung durchgeführt werden • Es werden keine Testfälle gewählt • Vollständige Aussagen über Korrekt oder Zuverlässigkeit können nicht erzeugt werden

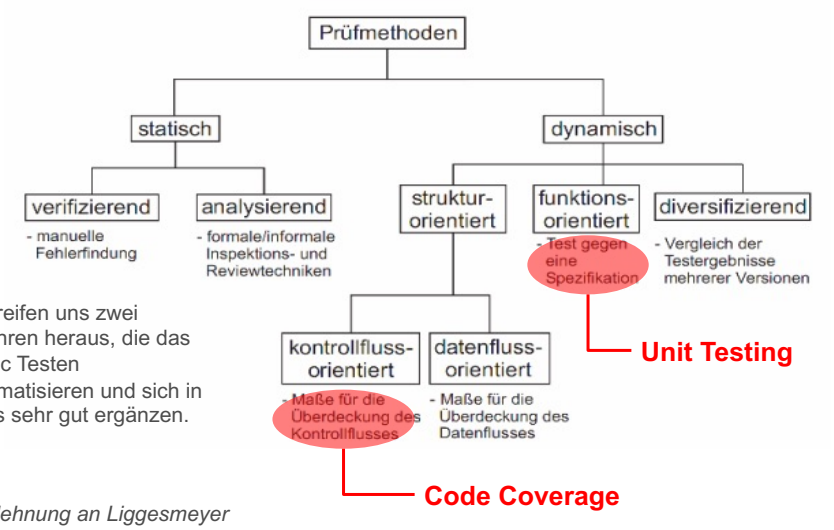
***Bsp: Einfachste Form des dynamischen Testens**, ist die Ausführung der zu testenden SW mit Eingaben einer Person. Sogenannte Ad-hoc Tests. Diese Teste machen Sie üblicherweise intuitiv und automatisch, wenn Sie programmieren. Ein ähnliches, (wenn auch weit systematischeres) Vorgehen ist das UNIT-Testing.*

***Bsp: Formale Beweisverfahren**, versuchen einen Beweis der Konsistenz einer Software und ihrer Spezifikation mit formalen Mitteln zu erbringen. Eine formale Spezifikation ist daher zwingend erforderlich. Die Verfahren ähneln sehr stark mathematischen Beweisen.*

53

Zwei Verfahren im Detail

TECHNISCHE HOCHSCHULE LÜBECK



Wir greifen uns zwei Verfahren heraus, die das ad-hoc Testen systematisieren und sich in Praxis sehr gut ergänzen.

In Anlehnung an Liggesmeyer

54

Wann haben Sie genug getestet?

TECHNISCHE HOCHSCHULE LÜBECK

Antwort: Wenn es extrem unwahrscheinlich wird, noch Fehler zu finden.

55

Wann haben Sie genug getestet?

Antwort: Wenn Sie mit Ihren Testfällen schon überall im Code waren!

Quelle: Pixabay

56

Worum geht es nun?

TECHNISCHE HOCHSCHULE LÜBECK

57

Wie lässt sich eine Code Coverage messen?

TECHNISCHE HOCHSCHULE LÜBECK

Anweisungs-überdeckung (C0)	Zweigüberdeckung (C1)	Bedingungs-überdeckung (C2)
• Einfachste Anweisungs-überdeckungs-methode • Mindestens alle Anweisungen sollen einmal ausgeführt werden • Das verlangt den Durchlauf durch alle Knoten des **Kontrollflussgraphen**	• Ausführung aller möglichen Zweige des zu testenden Programms. • Das verlangt den Durchlauf durch alle Kanten des **Kontrollflussgraphen**	• Betrachtet wird die logische Struktur von Entscheidungen • Gründliche Überprüfung zusammengesetzter Entscheidungen zu testender Software

58

Kontrollflussgraph

TECHNISCHE HOCHSCHULE LÜBECK

Der Kontrollflussgraph ist eine Methode zur Darstellung von Programmen. Ein Kontrollflussgraph ist ein gerichteter Graph

$$G = (N, E, n_{start}, n_{final})$$

N ist die endliche Menge der Knoten. $E \subseteq N \times N$ ist die Menge der gerichteten Kanten. n_{start} ist der Startknoten. n_{final} ist der Endknoten.

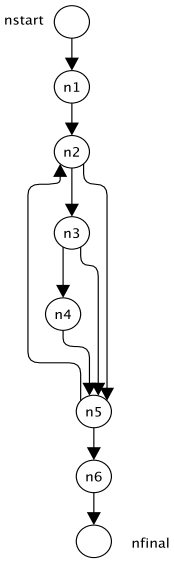
Knoten stellen Anweisungen dar. Eine gerichtete Kante von einem Knoten i zu einem Knoten j beschreibt einen möglichen Kontrollfluss von Knoten i zu Knoten j.

Die gerichteten Kanten werden als **Zweige** bezeichnet. Eine alternierende Sequenz aus Knoten und Anweisungen, die mit dem Startknoten n_{start} beginnt und mit dem Endknoten n_{final} endet heißt **Pfad**.

Beispiel eines Kontrollflussgraphen

TECHNISCHE HOCHSCHULE LÜBECK

```
/**
 * Liefert die Anzahl aller groß- und kleingeschriebenen
 * Vokale in einem String.
 * Vokale sind 'a', 'e', 'i', 'o' und 'u'.
 * @param s Zeichenkette
 * @return Anzahl an Vokalen für s != null, 0 für s == null
 */
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                    // n4
    }
  }                                                // n5
  return vokale;                                   // n6
}
```



Anweisungsabdeckungstest

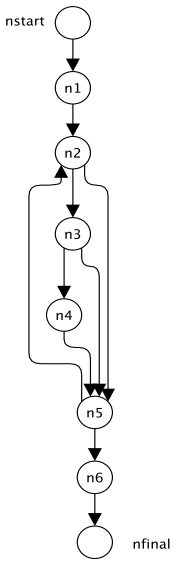
TECHNISCHE HOCHSCHULE LÜBECK

Der Anweisungsabdeckungstest ist die einfachste kontrollflussorientierte Testmethode. Das Ziel der Anweisungsüberdeckung ist die **mindestens einmalige Ausführung aller Anweisungen**,

also die **Abdeckung aller Knoten** in einem Kontrollflussgraphen.

Als Testmaß wird der erreichte Anweisungsüberdeckungsgrad definiert. Er ist das Verhältnis der ausgeführten Anweisungen zu der Gesamtzahl der im Prüfling vorhandenen Anweisungen.

$$C_0 = \frac{Anweisungen_{ausgeführt}}{Anweisungen}$$

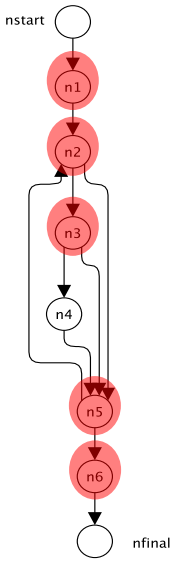


61

Anweisungsabdeckungstest Beispiel

TECHNISCHE HOCHSCHULE LÜBECK

```
public static int zaehleVokale(String s) {
  int vokale = 0;                              // n1
  for (char c : s.toCharArray()) {             // n2
    if (c == 'a' || c == 'A' ||                // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                // n4
    }
  }                                            // n5
  return vokale;                               // n6
}
int vs = zaehleVokale("Try");
```



62

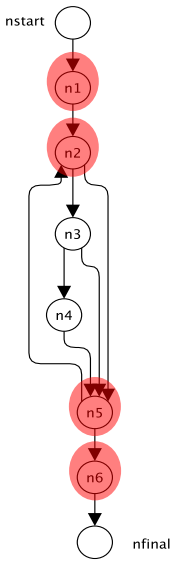
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Bestimmen Sie die Anweisungsüberdeckung der Routine `zaehleVokale` für den Eingabeparameter des leeren Strings "":

```
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                    // n4
    }
  }                                                // n5
  return vokale;                                   // n6
}
```



63

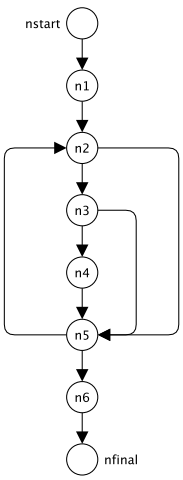
Zweigüberdeckungstest

TECHNISCHE HOCHSCHULE LÜBECK

Der Zweigüberdeckungstest ist eine strengere kontrollflussorientierte Testmethode. Der Anweisungsüberdeckungstest ist im Zweigüberdeckungstest vollständig enthalten. Das Ziel des Zweigüberdeckungstests ist die **Ausführung aller Zweige**,

also die **Abdeckung aller Kanten** in einem Kontrollflussgraphen.

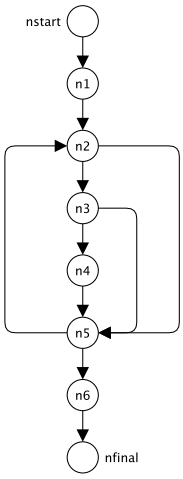
Als Testmaß wird das erreichte Zweigüberdeckungsmaß definiert. Er ist das Verhältnis der ausgeführten **primitiven Zweige** zu der Gesamtzahl aller vorhandenen primitiven Zweige.



64

Zweigabdeckung

TECHNISCHE HOCHSCHULE LÜBECK

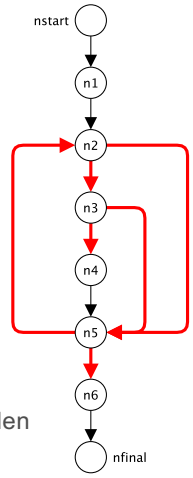


Bei der Zweigüberdeckung werden Zweige nicht berücksichtigt, die immer dann ausgeführt werden, wenn ein anderer Zweig ausgeführt wird.

Von Interesse sind also **Knoten,** die auf mehr als einem Weg verlassen werden können (rot markierte Kanten).

Außerdem muss nur ein Teil der Zweige (rot markiert) für eine Code Coverage instrumentiert werden.

$$C_1 = Zweige_{ausgeführt} / Zweige_{vorhanden}$$

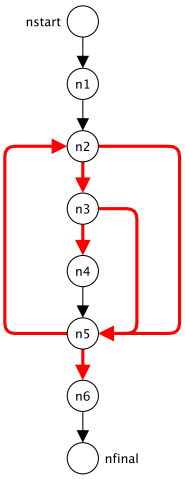


65

Zweigabdeckungstest Beispiel

TECHNISCHE HOCHSCHULE LÜBECK

```
public static int zaehleVokale(String s) {
  int vokale = 0;                              // n1
  for (char c : s.toCharArray()) {             // n2
    if (c == 'a' || c == 'A' ||                // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                // n4
    }
  }                                            // n5
  return vokale;                               // n6
}
int vs = zaehleVokale("Try");
```



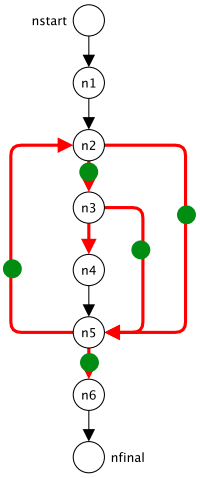
$$C_1 = Zweige_{ausgeführt} / Zweige_{vorhanden}$$

66

Zweigabdeckungstest Beispiel

TECHNISCHE HOCHSCHULE LÜBECK

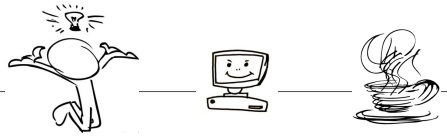
```
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                    // n4
    }
  }                                                // n5
  return vokale;                                   // n6
}
int vs = zaehleVokale("Try");
```



$C_1 = 5 / 6 = 83\%$

67

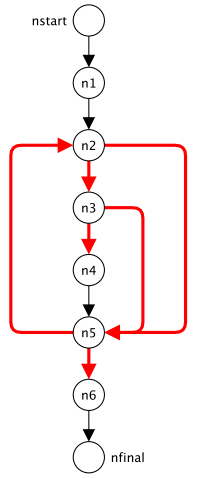
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Bestimmen Sie die Zweigüberdeckung der Routine `zaehleVokale` für den Eingabeparameter des leeren Strings "":

```
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                    // n4
    }
  }                                                // n5
  return vokale;                                   // n6
}
```



68

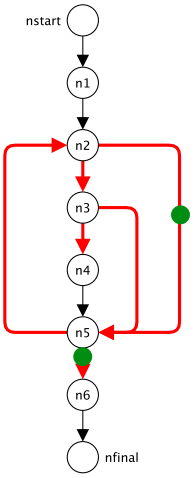
Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Bestimmen Sie die Zweigüberdeckung der Routine `zaehleVokale` für den Eingabeparameter des leeren Strings "":

```
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;                                    // n4
    }
  }                                                // n5
  return vokale;                                   // n6
}
```



$C_1 = 2 / 6 = 33\%$

69

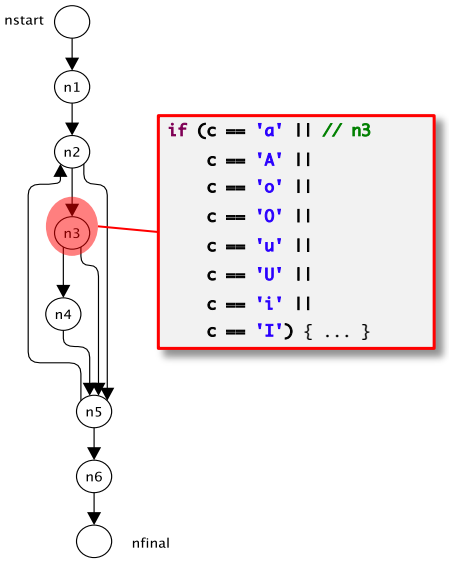
Bedingungsüberdeckungstest

TECHNISCHE HOCHSCHULE LÜBECK

Bei der Zweigabdeckung wird nicht betrachtet wie kompliziert eine logische Bedingung ist, die darüber entscheidet welcher Zweig durchlaufen wird.

Mittels Bedingungsüberdeckungstesten wird auch die logische Struktur von Entscheidungen zu testender Software berücksichtigt.

Es gibt eine Vielzahl an Tests in diesem Bereich. Wir befassen uns nur mit dem einfachsten Test (simple condition coverage test) um die Prinzipien der Bedingungsüberdeckung zu vermitteln.



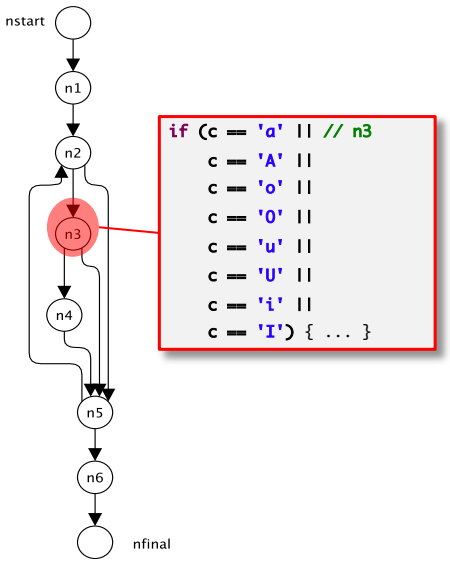
70

Bedingungsüberdeckungstest
(Simple Condition Coverage)

TECHNISCHE HOCHSCHULE LÜBECK

Das einfachste Verfahren des Bedingungsabdeckungstests fordert den Test aller atomaren Teilentscheidungen gegen `true` und `false`.

Im rechts stehenden Beispiel reicht also nicht das Testen mit einem beliebigen Vokal, sondern es müssen alle Vokale (in Groß- und Kleinschreibung) abgeprüft werden (inkl. Zeichenketten ohne Vokale), um eine 100% Simple Condition Coverage zu erzielen.



71

Eclipse Tools für Code Coverages
Frei verfügbar: eclEmma und eCobertura

TECHNISCHE HOCHSCHULE LÜBECK

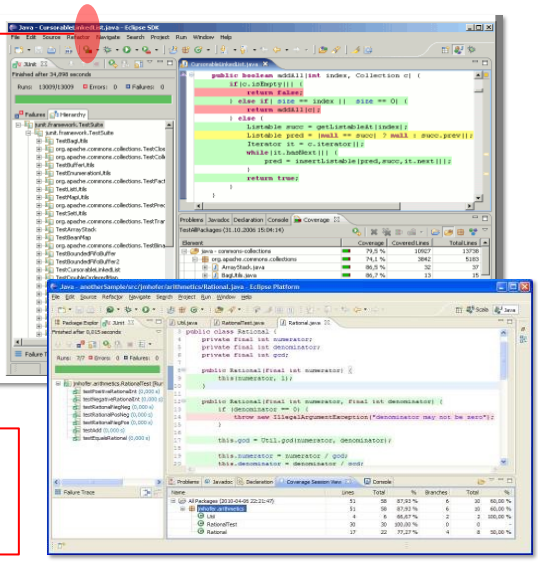
EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse workbench:

- **Fast develop/test cycle:** Launches from within the workbench like JUnit test runs can directly be analyzed for code coverage.
- **Rich coverage analysis:** Coverage results are immediately summarized and highlighted in the Java source code editors.
- **Non-invasive:** EclEmma does not require modifying your projects or performing any other setup.

Since version 2.0 EclEmma is based on the JaCoCo code coverage library. The Eclipse integration has its focus on supporting the individual developer in an highly interactive way. For automated builds please refer to JaCoCo documentation for integrations with other tools.

Originally EclEmma was inspired by and technically based on the great EMMA library developed by Vlad Roubtsov.

The update site for EclEmma is **http://update.eclemma.org/**. For the latest Eclipse version EclEmma is also available via the Marketplace Client, simply search for "EclEmma".



Mit Code Coverage Tools können Sie Programme wie gehabt starten, nur es wird die Code Coverage zur Laufzeit mit erhoben.

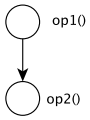
Bildquelle: Eclipse

72

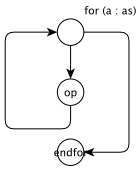
Kontrollflussgraphen Cheat Sheet

TECHNISCHE HOCHSCHULE LÜBECK

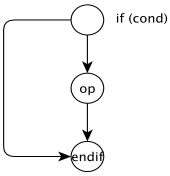
```
op1();
op2();
```



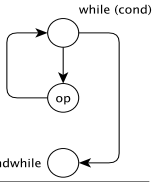

```
for (a : as) {
   op();
}
```



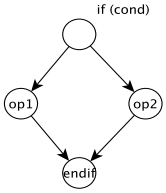

```
if (cond) {
   op();
}
```



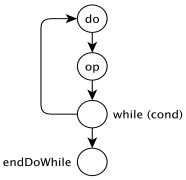

```
while (cond) {
  op();
}
```




```
if (cond) {
   op1();
} else {
   op2();
}
```



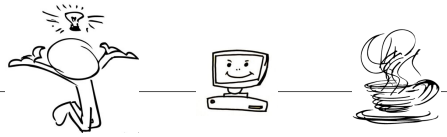

```
do {
  op();
} while (cond);
```



73

73

Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Geben Sie bitte den Kontrollflussgraphen für folgende Methode an:

```
public static int countChar(String s, char f) {
  int ret = 0;
  for (char c : s.toCharArray()) {
    ret += f == c ? 1 : 0;
  }
  return ret;
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

74

74

Miniübung:



TECHNISCHE HOCHSCHULE LÜBECK

Geben Sie bitte den Kontrollflussgraphen für folgende Methode an:

```
public static int countCharIf(String s, char f) {
  int ret = 0;
  for (char c : s.toCharArray()) {
    if (f == c) {
      ret++;
    }
  }
  return ret;
}
```


Worum geht es nun?

TECHNISCHE HOCHSCHULE LÜBECK

Modultests (Unit Testing)

TECHNISCHE HOCHSCHULE LÜBECK

Ziel des Modultests (unit tests) ist es, frühzeitig Programmfehler in den Modulen einer Software (z. B. von einzelnen Klassen) zu finden. Die Funktionalität der Module kann so meist einfacher getestet werden, als wenn die Module bereits integriert sind, da in diesem Fall die Abhängigkeit der Einzelmodule mit in Betracht gezogen werden muss.

Modultests werden heutzutage üblicherweise automatisch ausgeführt.

Die automatisierten Modultests haben den Vorteil, dass sie rasch und von jedermann ausgeführt werden können. Somit besteht die Möglichkeit, nach jeder Programmänderung durch Ablauf aller Modultests nach Programmfehlern zu suchen. Dies wird üblicherweise empfohlen, da damit etwaige neu entstandene Fehler schnell entdeckt und somit kostengünstig behoben werden können.

Quelle: Wikipedia, Modultest

Eigenschaften von Unit Tests

TECHNISCHE HOCHSCHULE LÜBECK

Isoliert	Test von Fehlverhalten	Laufende Ausführung	Test des Vertrag
• Modultests testen die Module isoliert, d. h. ohne die Interaktion der Module mit anderen. Ist das nicht der Fall spricht man nicht von Modultests, sondern von Integrationstests.	• Modultests testen ausdrücklich nicht nur das Verhalten des Moduls im Gutfall, beispielsweise bei korrekten Eingabewerten, sondern auch im Fehlerfall, beispielsweise bei unkorrekten Eingabewerten.	• Modultests sollten im Laufe der Entwicklung regelmäßig durchgeführt werden, um zu verifizieren, dass Änderungen keine unerwünschten Nebeneffekte haben. Modultests sollten daher von jedem Entwickler vor dem Einchecken durchgeführt werden.	• Modultests sollen gemäß dem Design-by-contract-Prinzip möglichst nicht die Interna einer Methode testen, sondern nur ihre externen Auswirkungen (Rückgabewerte, Ausgaben, Zustands-änderungen, Zusicherungen).

Unit Testing mit JUnit 4 und Eclipse (I)

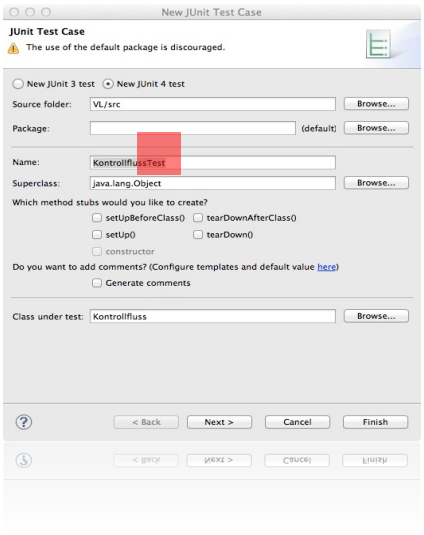
TECHNISCHE HOCHSCHULE LÜBECK

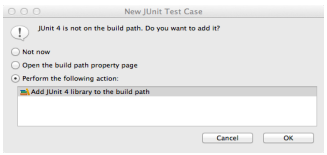
Selektieren Sie die zu testende Klasse und wählen Sie über die rechte Maustaste:

```
New -> JUnitTest Case
```

UnitTests, die sich auf eine Klasse beziehen, erhalten per Konvention den Namen der Klasse mit der Suffix Test.

Bestätigen Sie ggf. dass die Junit Bibliothek, dem Klassenpfad hinzugefügt werden soll.





79

Unit Testing mit JUnit 4 und Eclipse (II)

TECHNISCHE HOCHSCHULE LÜBECK

```
import static org.junit.Assert.*;

import org.junit.Test;

public class KontrollflussTest {

  @Test
  public void test() {
    fail("Not yet implemented");
  }

}
```

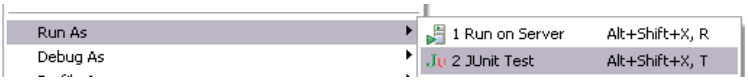
Dadurch erzeugt das JUnit Plugin eine Testklasse mit einer Testmethode.

Die Methode hat die Annotation @Test (und kann übrigens einen beliebigen Bezeichner haben).

Dadurch erkennt Eclipse, dass es sich um einen Testfall handelt.

Innerhalb dieser Methode wird ein Testfall definiert, in dem die zu testende Funktionalität mit Testwerten aufgerufen wird und mittels Assert korrekte Rückgabewerte abgeprüft werden.

In Eclipse können so alle in einem Projekt befindlichen Testfälle aufgerufen werden, in dem das Projekt selektiert wird und anschließend mittels rechter Maustaste:



80

Unit Testing mit JUnit 4 und Eclipse (III)

TECHNISCHE HOCHSCHULE LÜBECK

Ein Testfall für die Methode `zaehleVokale` könnte jetzt wie folgt aussehen:

```
import static org.junit.Assert.*;

import org.junit.Test;

public class KontrollflussTest {

  @Test
  public void testZaehleVokale() {
    assertEquals(1, Kontrollfluss.zaehleVokale("a"));
    assertEquals(1, Kontrollfluss.zaehleVokale("O"));
    assertEquals(3, Kontrollfluss.zaehleVokale("Hallo World"));
    assertEquals(0, Kontrollfluss.zaehleVokale(""));
    assertEquals(8, Kontrollfluss.zaehleVokale("aiouAIOU"));
    assertEquals(0, Kontrollfluss.zaehleVokale("123xyz"));
  }
}
```


81

Annotationen von JUnit 4 im Überblick

TECHNISCHE HOCHSCHULE LÜBECK

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

82

Asserts von JUnit 4 im Überblick

TECHNISCHE HOCHSCHULE LÜBECK

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertsEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertsEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

83

Unit Testing mit JUnit 4 und Eclipse (IV)

TECHNISCHE HOCHSCHULE LÜBECK

Dieser Testfall hat für die `zaehleVokale()` Methode übrigens eine Code Coverage von 100% C0 und 100% C2 und 100% C3 (simple coverage).

```
import static org.junit.Assert.*;

import org.junit.Test;

public class KontrollflussTest {

  @Test
  public void testZaehleVokale() {
    assertEquals(1, Kontrollfluss.zaehleVokale("a"));
    assertEquals(1, Kontrollfluss.zaehleVokale("O"));
    assertEquals(3, Kontrollfluss.zaehleVokale("Hallo World"));
    assertEquals(0, Kontrollfluss.zaehleVokale(""));
    assertEquals(8, Kontrollfluss.zaehleVokale("aiouAIOU"));
    assertEquals(0, Kontrollfluss.zaehleVokale("123xyz"));
  }
}
```

Demzufolge ist `zaehleVokale()` korrekt – oder?

84

Nein, denn wir haben bislang den Code getestet und nicht dessen Spezifikation abgeprüft!

TECHNISCHE HOCHSCHULE LÜBECK

```
/**
 * Liefert die Anzahl aller groß- und kleingeschriebenen
 * Vokale in einem String.
 * Vokale sind 'a', 'e', 'i', 'o' und 'u'.
 * @param s Zeichenkette
 * @return Anzahl an Vokalen für s != null, 0 für s == null
 */
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I') {
      vokale++;
    }
  }
  return vokale;
}
```

Was ist mit?

```
assertEquals(1, Kontrollfluss.zaehleVokale("e"));
```

Success or No Success?

!!! FEHLER !!!

Trotz 100% C0, C1 und C2 Abdeckung

85

Jetzt korrekt?

TECHNISCHE HOCHSCHULE LÜBECK

```
/**
 * Liefert die Anzahl aller groß- und kleingeschriebenen
 * Vokale in einem String.
 * Vokale sind 'a', 'e', 'i', 'o' und 'u'.
 * @param s Zeichenkette
 * @return Anzahl an Vokalen für s != null, 0 für s == null
 */
public static int zaehleVokale(String s) {
  int vokale = 0;                                  // n1
  for (char c : s.toCharArray()) {                 // n2
    if (c == 'a' || c == 'A' ||                    // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I' ||
        c == 'e' || c == 'E') {
      vokale++;
    }
  }                                                // n5
  return vokale;                                   // n6
}
```

Was ist mit?

```
assertEquals(0, zaehleVokale(null));
```

Success or No Success ?

86

Erst jetzt ist `zaehleVokale()` korrekt!

TECHNISCHE HOCHSCHULE LÜBECK

```java
/**
 * Liefert die Anzahl aller groß- und kleingeschriebenen
 * Vokale in einem String.
 * Vokale sind 'a', 'e', 'i', 'o' und 'u'.
 * @param s Zeichenkette
 * @return Anzahl an Vokalen für s != null, 0 für s == null
 */
public static int zaehleVokale(String s) {
  if (s == null) { return 0; }
  int vokale = 0;                                    // n1
  for (char c : s.toCharArray()) {                   // n2
    if (c == 'a' || c == 'A' ||                      // n3
        c == 'o' || c == 'O' ||
        c == 'u' || c == 'U' ||
        c == 'i' || c == 'I' ||
        c == 'e' || c == 'E') {
      vokale++;                                      // n4
    }
  }                                                  // n5
  return vokale;                                     // n6
}
```


87

Was ist schief gegangen?

TECHNISCHE HOCHSCHULE LÜBECK

Entwickele Testfälle aus der Spezifikation

NIEMALS aus dem Code!

```java
aehleVokale() {
  Kontrollfluss.zaehleVokale("a"));
  assertEquals(1, Kontrollfluss.zaehleVokale("O"));
  assertEquals(3, Kontrollfluss.zaehleVokale("Hallo World"));
  assertEquals(0, Kontrollfluss.zaehleVokale(""));
  assertEquals(8, Kontrollfluss.zaehleVokale("aiouAIOU"));
  assertEquals(0, Kontrollfluss.zaehleVokale("123xyz"));
  assertEquals(4, Kontrollfluss.zaehleVokale("eEEe"));
  assertEquals(0, Kontrollfluss.zaehleVokale(null));
}
```

Teste nicht nur die Normalfälle!
Teste Sonder- und Spezialfälle.
SEI FIEß!!!

```java
/**
 * Liefert die Anzahl aller groß- und kleingeschriebenen
 * Vokale in einem String.
 * Vokale sind 'a', 'e', 'i', 'o' und 'u'.
 * @param s Zeichenkette
 * @return Anzahl an Vokalen für s != null, 0 für s == null
 */
```


88

Code Coverage

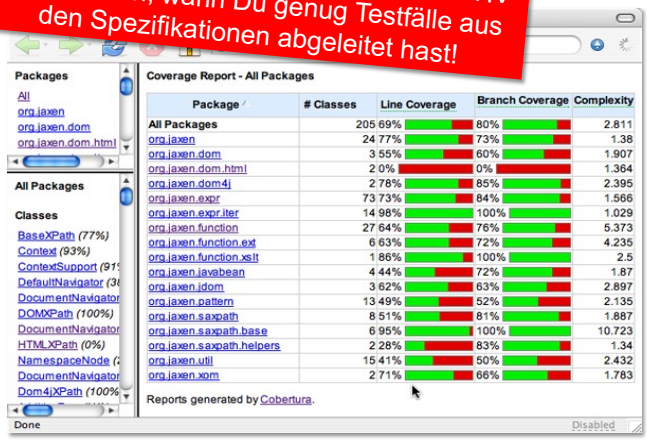
TECHNISCHE HOCHSCHULE LÜBECK

Nutze Code Coverage dazu, um OBJEKTIV festzustellen, wann Du genug Testfälle aus den Spezifikationen abgeleitet hast!

Tatsächlich existieren einige SW-Entwicklungsstandards, die vorschreiben wie hoch eine Code Coverage sein muss, damit eine SW abgenommen wird.

Bsp.: DO 178B Air (Luftfahrzeuge)

(So könnte z.B. eine Zweigüberdeckung > 85% C1, bei Medium Critical Components im QM-Plan vorgeschrieben sein).



89

Tools zum Unit Testen und Code Coverage

TECHNISCHE HOCHSCHULE LÜBECK

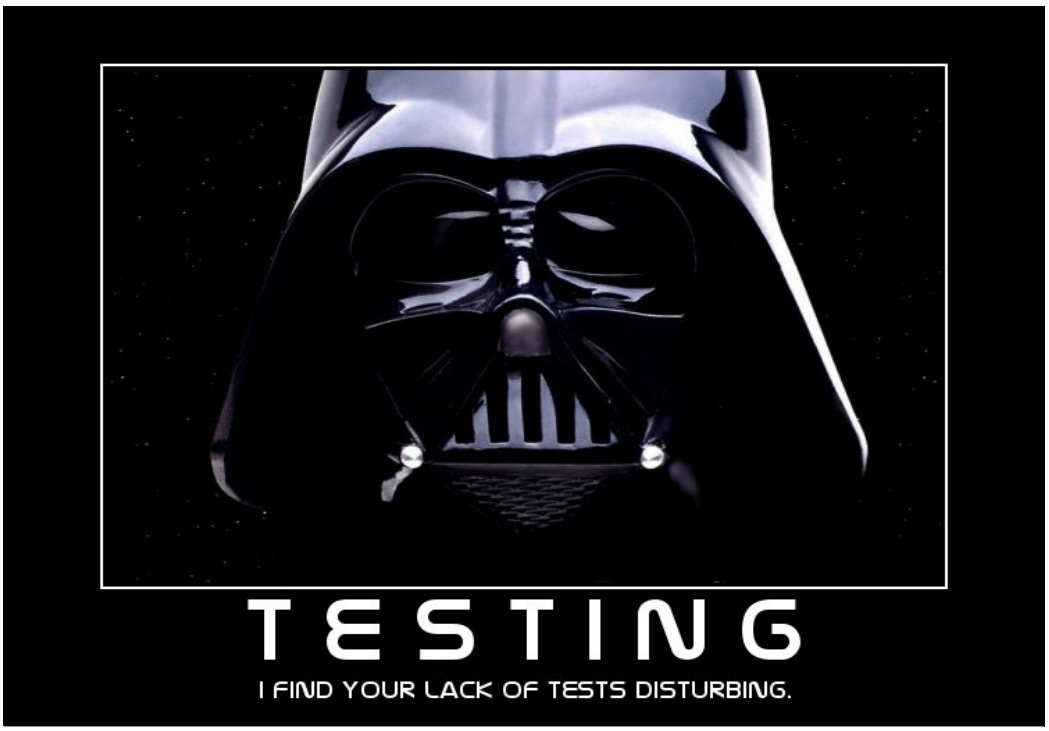
LV **JUnit Plugin** ist das vermutlich verbreiteste ECLIPSE Plugin zum Unit Testen für Java (gutes Tutorial finden Sie hier: http://www.vogella.com/articles/JUnit/article.html)

- **eCobertura** ist ein Code Coverage Plugin zum Messen der Zweigüberdeckung (und anteiliger Bedingungsabdeckung, http://ecobertura.johoop.de/)

LV **eclEmma** ist ein gut visualisierendes Code Coverage Plugin (leider nur) zum Messen der Anweisungs-/(Pfad-)überdeckung (http://www.eclemma.org/)

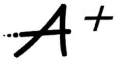
- **Clover** ist ein sehr mächtiges und komfortables Code Coverage Tool (leider kommerziell, http://www.atlassian.com/software/clover)

90



91

Zusammenfassung



TECHNISCHE HOCHSCHULE LÜBECK

- **Polymorphie**
  - Liskovsches Substitutionsprinzip
  - Run Time Type Identification
- **Verträge**
  - Einhalten von Verträgen bei abgeleiteten Klassen
    - Abschwächen von Vorbedingungen
    - Verschärfen von Nachbedingungen
    - Einhaltung der Klasseninvarianten der Basisklasse
- **Unit Testing**
  - JUnit
  - Teste Module isoliert, Teste Fehlverhalten, Teste laufend
  - Teste die Spezifikation nicht den Code
- **Code Coverage**
  - CO, C1, C2
  - Toolings

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
92

92