
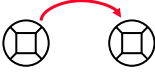




### Inhaltliche Gliederung von SE1

Stufe	Titel	Themen u.a.	Woche
1	 „Simple Klasse, simple Objekte“	Klasse, Objekt, Methode, Parameter, Feld, Variable, Sequenz, Zuweisung, Ausdruck, Syntax in EBNF, bedingte Anweisung, primitiver Typ	1 - 4
2	 „Objekte benutzen Objekte“	Typ, Referenz, UML: Klassen- und Objektdiagramme, Schleife, Rekursion, Sichtbarkeit und Lebensdauer, reguläre Ausdrücke	5 - 7
3	 „Schnittstellen mit Interfaces“	Black-Box-Test, Testklasse, Interface, Sammlungen benutzen	8 - 9
4	 „Hinter den Kulissen“	Arrays, Sammlungen implementieren: Array-Liste, verkettete Liste, Hashing; Sortieren; Stack; Graphen	10 - 14

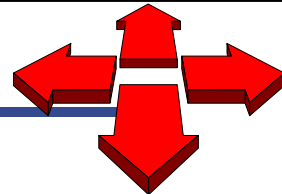
SE1 - Level 3

2

### Nicht vergessen: „Guess My Object“ ausprobieren!

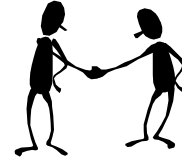
- **Janina Nemec**, eine SE1-Betreuerin, arbeitet derzeit im Rahmen ihrer **Bachelor-Arbeit** an dem Spiel „Guess My Object“.
- Es dient primär dazu, den Zusammenhang zwischen **Schnittstelle** (relevant für Klienten einer Klasse) und **Implementation** (relevant für Programmierer einer Klasse selbst) zu verdeutlichen.
- Die Idee ist einfach: Für jedes Spiel ist **ein BlueJ-Projekt** vorgegeben mit einer oder mehreren Klassen (für Level 1-Aufgaben immer eine Klasse).
- Bei diesen Projekten **fehlt** jeweils der **Quelltext**. Es können lediglich **Exemplare** der Klassen **erzeugt** und dann **interaktiv benutzt** werden.
- Aufgabe 1: Herausfinden, **was** die Objekte der Klasse tun.
- Aufgabe 2: Eine **eigene Implementation** angeben, die sich genauso verhält.
- **Schummeln** ist **verboten**: Exemplare dürfen **nicht** mit dem **Objekt-Inspektor** angesehen werden.
- Janina steht für alle weiteren Fragen zur Verfügung.

### Klassen, Typen und Interfaces



- Wir haben gesehen, dass eine **Klasse** einen **Typ** definiert:
  - Die öffentlichen Methoden einer Klasse bilden die **Operationen** dieses Typs.
  - Die Exemplare der Klasse bilden die **Wertemenge** des Typs.
- Eine **Klasse** hat somit zwei zentrale Eigenschaften:
  - Sie definiert einen **Typ**.
  - Sie legt die **Implementation** dieses Typs über ihre Methoden und Felder fest.
- Ein **Interface** in Java definiert ausschließlich einen Typ und legt keine Implementation fest.
- Wir betrachten diesen Zusammenhang näher.

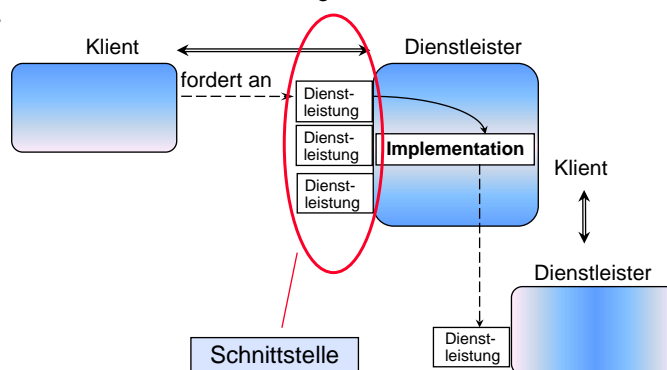
## Im Kern von allem: Dienstleister und Klienten



- Das Objekt, das bei einer bestimmten (Teil-)Aufgabe einen Dienst leistet, ist der **Dienstleister**.
- Das Objekt, das eine konkrete Dienstleistung eines anderen Objektes in Anspruch nimmt, wird als **Klient** bezeichnet.

## Wir erinnern uns: Dienstleistungen an der Schnittstelle

- Objekte bieten **Dienstleistungen** als **Methoden** an ihrer **Schnittstelle** an. Diese Dienstleistungen werden von anderen Objekten, den Klienten, benutzt. Dazu fordert der Klient eine Dienstleistung des Anbieters an.
- Der Dienstleister realisiert die Leistungen in den Rümpfen der öffentlichen Methoden bzw. in privaten Hilfsmethoden. Dabei kann er selbst wieder Dienstleistung von anderen Anbietern einholen.



## Kapselung



- Kapselung ist zunächst ein programmiertechnischer Mechanismus, der bestimmte Programmkonstrukte (z.B. Felder oder Methodenrumpfe) vor äußerem Zugriff schützt.
  - In Java können durch die Modifikatoren **public** und **private** Methoden und Felder einer Klassen für Klienten sichtbar gemacht oder gekapselt werden.
- Allgemein streben wir an, dass Klassen als **Black Box** betrachtet werden können, die nur relevante Informationen nach außen zeigen.
- **Vorteile** von Kapselung sind:
  - Das Ausblenden von Details vereinfacht die Benutzung.
  - Details der Implementation können geändert werden, ohne dass diese Änderungen Klienten betreffen müssen.



SE1 – Level 3

7

## Die Doppelrolle einer Klasse



- Aus Sicht der **Klienten** einer Klasse ist interessant:
  - Welche **Operationen** können an Exemplaren der Klasse aufgerufen werden?
  - Welchen Typ haben die **Parameter** einer Operation und welches **Ergebnis** liefert sie?
  - Was sagt die **Dokumentation** (Kommentare, javadoc) über die **Benutzung**?
- Für die **Implementation** der Methoden einer Klasse ist relevant:
  - Wie sind die Operationen in den **Methodenrumpfen** umgesetzt?
  - Welche **Exemplarvariablen/Felder** definiert die Klasse?
  - Welche **privaten Hilfsmethoden** stehen in der Klasse zur Verfügung?

**Außensicht,  
öffentliche  
Eigenschaften,  
Dienstleistungen,  
Schnittstelle**

**Innensicht,  
private  
Eigenschaften,  
Implementation**

SE1 – Level 3

8

## Trennung von Schnittstelle und Implementation

- Für einen **Klienten** ist die Art und Weise der Realisierung (die Innensicht) uninteressant. Er abstrahiert von der Umsetzung und ist ausschließlich an der gebotenen **Dienstleistung** (der Außensicht) interessiert.
- Diese beiden Aspekte einer Klasse lassen sich **begrifflich** und **technisch** von einander trennen.
- Dass die Unterscheidung nützlich ist, haben wir bereits gesehen:
  - In BlueJ lässt sich im Editor die Implementation einer Klasse oder ihre Schnittstelle anzeigen. Wenn wir eine Klasse lediglich benutzen wollen, reicht uns die Schnittstellensicht.
  - Das Java API (kurz für Application Programming Interface) bietet von allen Bibliotheksklassen als Dokumentation die Schnittstellensicht.
- Als Konsequenz einer Trennung ergibt sich: **Die gleiche Schnittstelle kann auf verschiedene Weise implementiert werden.**



## Ein einfaches Beispiel

- Eine Klasse **Konto** bietet an ihrer Schnittstelle die Operationen **einzahlen**, **auszahlen** und **gibSaldo**.
- Die **simple** Implementation benutzt eine Exemplarvariable, um den Saldo zu speichern. Jede Ein- und Auszahlung verändert den Wert dieser Variablen.
- Dieselbe Schnittstelle könnte auch **anders** realisiert werden: Die Informationen über jede Ein- und Auszahlung werden in einer Liste gespeichert. Der Saldo wird erst berechnet, wenn **gibSaldo** aufgerufen wird, indem die Ein- und Auszahlungen aufaddiert werden.
- Für einen Klienten würde sich nichts ändern: Er ruft in beiden Fällen die sichtbaren Operationen auf und erhält die gleichen Ergebnisse.



## Interfaces in Java



- Java stellt ein spezielles Sprachkonstrukt zur Verfügung, mit dem ausschließlich eine Schnittstelle definiert werden kann: **benannte Schnittstellen** (engl.: named interface). Sie werden mit dem Schlüsselwort **interface** definiert.
- Für das Konto-Beispiel sieht die benannte Schnittstelle so aus:

```
interface Konto
{
    void einzahlen(int betrag);
    void auszahlen(int betrag);
    int gibSaldo();
}
```

- Um die benannte Schnittstelle als Sprachkonstrukt in Java begrifflich von der Schnittstelle einer Klasse zu unterscheiden, nennen wir sie im Folgenden **Interface**.

## Zentrale Eigenschaften von Interfaces



- Interfaces ...
  - sind Sammlungen von **Methodenköpfen**. Alle Methoden in einem Interface sind implizit **public**;
  - enthalten **keine Methodenrümpfe** (also keine Anweisungen);
  - definieren **keine Felder**;
  - sind **nicht instanzierbar** - es können keine Exemplare von Interfaces erzeugt werden;
  - werden von Klassen implementiert.



## Interfaces werden durch Klassen implementiert

- Eine Klasse kann deklarieren, dass sie ein Interface implementiert.
- Die Klasse muss dann für jede Operation des Interfaces eine entsprechende Methode anbieten. Sie „erfüllt“ dann das Interface.

```
class KontoSimpel implements Konto
{
    private int _saldo;

    public void einzahlen(int betrag) {...}
    public void auszahlen(int betrag) {...}
    public int gibSaldo() {...}
}
```

- An allen Stellen, an denen ein Objekt mit einem bestimmten Interface erwartet wird, kann eine Referenz auf ein Exemplar einer implementierenden Klasse verwendet werden.

## Auswirkungen auf Klienten

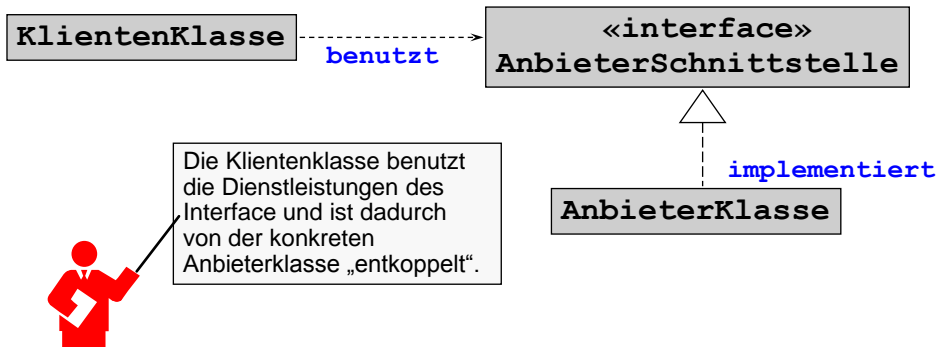
- Bei den Auswirkungen müssen zwei Zusammenhänge unterschieden werden:
  - Objektbenutzung
  - Objekterzeugung
- Bei der **Objektbenutzung** ändert sich durch Interfaces nichts: Klienten können die Operationen des Interface genauso aufrufen wie die einer Klasse.
- Lediglich bei der **Objekterzeugung** muss „Farbe bekannt“ werden: Da von einem Interface keine Exemplare erzeugt werden können, muss bei der Objekterzeugung immer eine implementierende Klasse angegeben werden.

**Nutzung:  
unverändert!**

```
class Ueberweiser
{
    public void ueberweise(Konto quelle,
                           Konto ziel,
                           int betrag)
    {
        quelle.auszahlen(betrag);
        ziel.einzahlen(betrag);
    }
}
```

```
Konto konto1 = new KontoSimpel(100);
Konto konto2 = new KontoSimpel(100);
Ueberweiser ueberweiser = new Ueberweiser();
ueberweiser.ueberweise(konto1, konto2, 50);
```

## Trennung von Schnittstelle und Implementation mit Interfaces

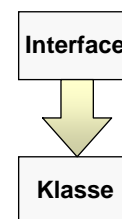
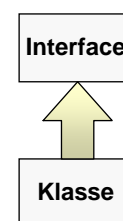


SE1 – Level 3

15

## Interfaces als Spezifikationen

- Im Konto-Beispiel haben wir aus einer Klasse ihre Schnittstelle herausgezogen, indem wir sie als Interface formuliert haben.
  - Wir haben das Interface aus der Klasse **abgeleitet**.
- **Es geht auch umgekehrt:**
  - Wir legen den **Umgang** für einen Typ fest, indem wir ein Interface definieren. Wir definieren die Operationen, indem wir ihre Köpfe festlegen und die gewünschten Auswirkungen als Kommentare beschreiben.
  - Später erstellen wir (oder eine andere Person) eine Klasse, die dieses Interface realisiert.
  - Das Interface bildet dann eine **Spezifikation**, die Klasse **eine mögliche Realisierung** (Umsetzung) dieser Spezifikation.

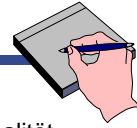


SE1 – Level 3

16



## Spezifikation allgemein



- Eine **Spezifikation** ist eine Beschreibung der **gewünschten** Funktionalität einer (Software-)Einheit.
- Eine gute Spezifikation beschreibt, **WAS** die Einheit leisten soll, aber nicht, **WIE** sie diese Leistung erbringen soll.
- Wir unterscheiden **informelle** (natürlichsprachliche) und **formale** (etwa mathematische) Spezifikationen.



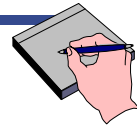
Die Trennung von Spezifikation und Implementation ist ein wesentliches Konstruktionsprinzip der imperativen und objektorientierten Programmierung!

Diese Trennung ist außerdem die Grundlage jeder professionellen Softwareentwicklung!

## Klassen und Interfaces definieren Typen

- Jede **Klasse** definiert in Java einen **Typ**:
  - durch ihre Schnittstelle (Operationen)
  - durch die Menge ihrer Exemplare (Wertemenge)
- Ein **Interface** definiert in Java ebenfalls einen **Typ**:
  - durch seine Schnittstelle
  - durch die Menge der Exemplare aller Klassen, die dieses Interface erfüllen, d.h. die die Schnittstelle des Interface implementieren.
- Für einen **Typ im objektorientierten Sinne** ist also wichtig:
  - welche Objekte gehören zur Wertemenge des Typs,
  - welche Operationen sind auf diesen Objekten zulässig
  - und **nicht**, wie die Operationen implementiert sind.

## Der erweiterte objektorientierte Typbegriff (1. Fassung)



- Der **klassische Typbegriff**:
  - Ein Typ definiert eine Menge an Werten, die eine Variable oder ein Ausdruck annehmen kann.
  - Jeder Wert gehört zu genau einem Typ.
  - Die Typinformation ist statisch aus dem Quelltext ermittelbar.
  - Ein Typ definiert die zulässigen Operationen.
- Der **erweiterte objektorientierte Typbegriff**:
  - Ein Typ definiert das Verhalten von Objekten durch eine Schnittstelle, ohne die Implementation der Operationen und des inneren Zustands festzulegen.
- Folge:
  - Ein Objekt wird von **genau einer** Klasse erzeugt.
  - Da eine Klasse auch mehrere Interfaces erfüllen kann, kann ein Objekt **zu mehr als einem** Typ gehören.

SE1 – Level 3

19

## Statischer und dynamischer Typ (I)



- Durch den erweiterten objektorientierten Typbegriff muss der statische vom dynamischen Typ einer **Referenzvariablen** unterschieden werden.
- Der **statische Typ einer Variablen** wird durch ihren deklarierten Typ definiert. Er heißt statisch, weil er zur Übersetzungszeit feststeht.  
**Konto k; // Konto ist hier der statische Typ von k**
- Der statische Typ legt die **Operationen** fest, die über die Variable aufrufbar sind.  
**k.einzahlen(200); // einzahlen ist hier eine Operation**
- Ein Compiler kann bei der Übersetzung prüfen, ob die genannte Operation tatsächlich im statischen Typ definiert ist.

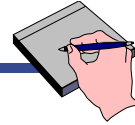
Hinweis: Dies alles gilt sinngemäß auch für Ausdrücke, deren Ergebnis ein Referenztyp ist.



SE1 – Level 3

20

## Statischer und dynamischer Typ (II)



- Der **dynamische Typ einer Referenzvariablen** hängt von der Klasse des Objektes ab, auf das die Referenzvariable zur Laufzeit verweist.

```
k = new KontoSimpel(); // dynamischer Typ von k: KontoSimpel
```

- Er bestimmt die Implementation und ist dynamisch in zweierlei Hinsicht:
  - Er kann erst zur Laufzeit ermittelt werden.
  - Er kann sich während der Laufzeit ändern.

```
k = new KontoAnders(); // neuer dynamischer Typ von k: KontoAnders
```

- Ein **Objekt** hingegen **ändert seinen Typ nicht**; es bleibt sein Leben lang ein Exemplar seiner Klasse.
- Der dynamische Typ einer Variablen (bzw. der Typ des referenzierten Objektes) entscheidet darüber, welche **konkrete Methode** bei einem **Operationsaufruf** ausgeführt wird. Da diese Entscheidung **erst zur Laufzeit** getroffen werden kann, wird dieser Prozess **dynamisches Binden** (einer Methode) genannt.



## Methoden und Operationen



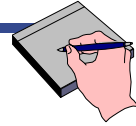
- Mit der konsequenten Unterscheidung von Schnittstelle und Implementation ist eine weitere begriffliche Differenzierung zwischen **Methode** und **Operation** sinnvoll:
  - Ein **Typ** definiert **Operationen**.
  - Eine **Klasse** hat **Methoden**.
- Da eine Klasse auch einen Typ definiert, gilt zusätzlich:
  - Die öffentlichen Methoden einer Klasse definieren die Operationen ihres Typs.**
- Eine Operation ist eine (Teil-)Spezifikation, die durch eine Methode realisiert werden kann.
- Interfaces definieren demnach Operationen und keine Methoden!

### Eine Operation, mehrere Methoden

Wenn es mehrere Möglichkeiten gibt, eine Schnittstelle zu implementieren, dann kann es auch mehrere Methoden geben, die dieselbe Operation implementieren.

Wir werden dazu noch Beispiele in SE1 sehen.

## Java-Objekte vergessen niemals ihre Klasse: Typtests



- Jedes dynamisch erzeugte Java-Objekt ist ein Exemplar von genau einer Klasse. Über die gesamte Lebenszeit eines Objektes ändert sich diese Zugehörigkeit zu der erzeugenden Klasse nicht.
- Ein Objekt kann deshalb jederzeit nach seiner erzeugenden Klasse gefragt werden. Java bietet dazu eine binäre Operation mit dem Schlüsselwort `instanceof` an. Ihre Operanden sind ein **Ausdruck mit einem Referenztyp** und der **Name eines Referenztyps** (Klasse oder Interface).
- Diese boolesche Operation `instanceof` nennen wir im Folgenden einen **Typstest**, weil sie prüft, ob der dynamische Typ des ersten Operanden dem genannten Typ entspricht.
- Beispiel:

```
Konto k; // Konto sei hier ein Interface
...
if (k instanceof KontoSimpel) // wenn k eine gültige Referenz
                               // auf ein Exemplar der Klasse
                               // KontoSimpel hält...
```

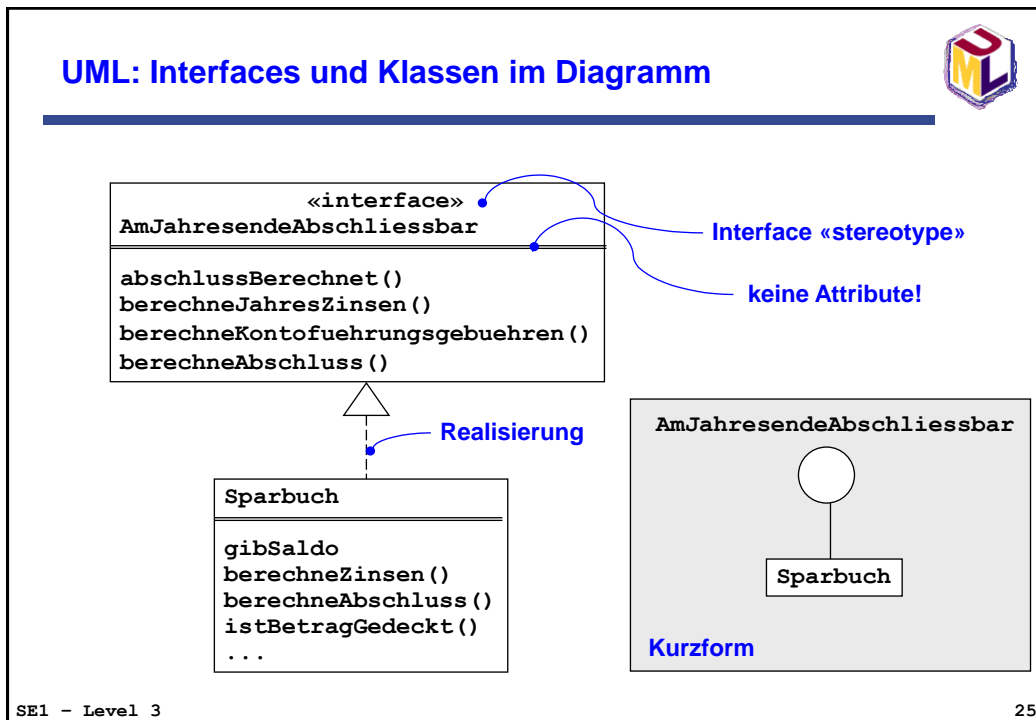
## Typzusicherungen



- Klienten sollten ausschließlich mit dem statischen Typ umgehen. Dies sichert ihre Unabhängigkeit gegenüber Änderungen.
- Es gibt jedoch Situationen, in denen Operationen des dynamischen Typs aufgerufen werden müssen. Dies wird durch **Typzusicherungen** ermöglicht.

```
Konto k; // Konto sei hier ein Interface
...
if (k instanceof KontoSimpel) // wenn k eine gültige Referenz auf ein
{                               // Exemplar der Klasse KontoSimpel hält...
    KontoSimpel ki = (KontoSimpel)k;
    ...
}
```

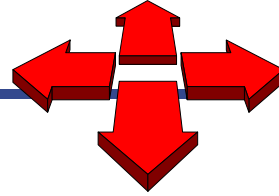
- Syntaktisch sieht dies wie eine Typumwandlung für die primitiven Typen aus – **es ist aber etwas völlig anderes!**
- Weder Objekt noch Objektreferenz werden verändert. Der Compiler erlaubt nun Aufrufe aller Operationen von `KontoSimpel`, die aber zur Laufzeit nur ausgeführt werden, wenn das Objekt den dynamischen Typ `KontoSimpel` hat.



## Zusammenfassung

- Die **Trennung von Schnittstelle und Implementation** ist ein zentrales Entwurfsprinzip der Softwaretechnik.
- Aufgrund der Trennung sind zu einer Schnittstelle **unterschiedliche Implementationen** möglich.
- Durch **Interfaces** lassen sich in Java benannte Schnittstellen von ihren implementierenden Klassen trennen.
- Interfaces können als **Spezifikationen** eingesetzt werden.
- Beim Umgang mit Interfaces müssen wir den **statischen** und den **dynamischen Typ** einer Variablen unterscheiden.

## Einführung in das systematische Testen



- Motivation
- Korrektheit von Software
- Testen ist Handwerkszeug
- Positives und Negatives Testen
- Äquivalenzklassen und Grenzwerte
- Black-Box-, White-Box- und Schreibtischtests
- Werkzeugunterstützung für Regressionstests

## Motivation: Warum Testen?

- Am 4. Juni 1996 explodierte die (unbemannte) europäische Raumrakete Ariane 501 ca. 40 Sekunden nach dem Start:

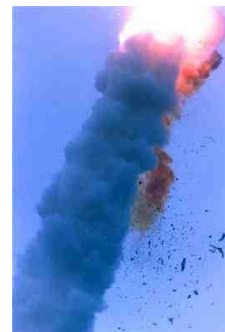
<http://www.oearv.at/wf-Dateien/wf1996-Dateien/arian501.html>

- Die Ursache für die Explosion war ein Programmfehler.
- Geschätzter Verlust durch die Explosion: ca. 1 Milliarde DM.

- *Nach Aussage des Untersuchungsberichts hätte der Fehler durch ausführlichere Tests der Steuerautomatik und des gesamten Flugkontrollsystems entdeckt werden können!*

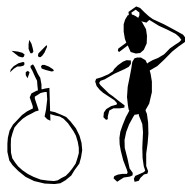
[http://www.esa.int/esaCP/Pr\\_33\\_1996\\_p\\_EN.html](http://www.esa.int/esaCP/Pr_33_1996_p_EN.html)

- Ergo: Testen kann sich lohnen.

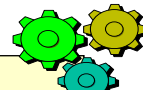


## Was ist Testen?

- Testen ist eine **Maßnahme zur Qualitätssicherung** mit dem Ziel, möglichst fehlerfreie Software zu erhalten.
- Testen dient zum **Aufzeigen von Fehlern** in Software; es kann i. A. **nicht** die Korrektheit der Software nachweisen.
- Testen ist damit das geplante und strukturierte Ausführen von Programmcode, um Probleme zu entdecken.



Testen sollte selbstverständlicher Bestandteil der Softwareentwicklung sein!!!  
Tests sollten automatisiert und wiederholbar sein.



## Zwei Zitate zum Thema Testen

*„Program testing can at best show the presence of errors, but never their absence.“*

Edsger W. Dijkstra



*"Beware of bugs in the above code; I have only proved it correct, not tried it."*

Donald Knuth

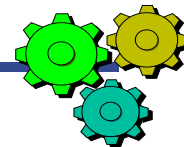


## Wann ist Software überhaupt „korrekt“?

- Die **Korrektheit** von Software kann immer nur in Relation zu ihrer **Spezifikation** gesehen werden - eine Software-Einheit ist korrekt, wenn sie ihre Spezifikation erfüllt.
- Der formale Nachweis, dass eine Software-Einheit ihre Spezifikation erfüllt, ist sehr aufwendig und schwierig.
- Voraussetzung für einen formalen Nachweis der Korrektheit ist, dass die Spezifikation selbst **formal definiert** ist. Dies ist nur sehr selten der Fall, meist sind Spezifikationen problembedingt nur informell formuliert.
- Auch wenn eine formale Spezifikation vorliegt: **Wie kann nachgewiesen werden, dass die Spezifikation selbst korrekt ist?**

Ergo: Für umfangreiche interaktive Programme sind formale Korrektheitsbeweise heute nicht machbar.

## Testen ist Handwerkszeug



- In der Praxis der Softwareentwicklung ist Testen nach wie vor ein wesentliches Mittel, um die Qualität von Software zu erhöhen.
- Für Software-Entwickler muss Testen zum Handwerkszeug gehören.
- Testen kann zwar keine Korrektheit nachweisen, aber den Eindruck belegen, dass eine Software-Einheit ihre Aufgabe in angemessener Weise erfüllt („**das Vertrauen erhöhen**“).
- Die Nützlichkeit einer Software kann sich häufig sowieso erst im Gebrauch zeigen.
- Aber: auch Testen hat seine Tücken...





## Probleme beim Testen

- **Technisch:**

- Testen ist schwierig (insbesondere bei grafischen Oberflächen).
- Testen braucht Zeit (und die ist in den meisten Projekten knapp).
- Tests müssen gut vorbereitet sein (Testplan).
- Tests müssen wiederholt werden (und damit wiederholbar sein).

- **Psychologisch:**

- Programmierer neigen dazu, nur die Fälle zu testen, die sie wirklich in ihrer Implementation abgedeckt haben.
- Testen ist stark beeinflusst durch die Programmiererfahrung.
- Häufig wird nur „positiv“ getestet.

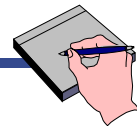


SE1 – Level 3

33

## Statische und dynamische Tests

- Linz und Spillner unterscheiden in **Basiswissen Softwaretest** grundlegend zwischen statischen und dynamischen Tests.
- Ein **statischer Test** (häufig auch **statische Analyse** genannt) bezieht sich auf die Übersetzungszeit und analysiert primär den Quelltext. Statische Tests können **von Menschen** durchgeführt werden (Reviews u.ä.) oder mit Hilfe von **Werkzeugen**, wenn die zu testenden Dokumente einer formalen Struktur unterliegen (was bei Quelltext zutrifft).
- **Dynamische Tests** sind alle Tests, bei denen die zu testende Software ausgeführt wird.

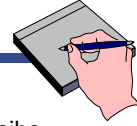


T. Linz, A. Spillner, *Basiswissen Softwaretest*, 4. überarbeitete und aktualisierte Auflage, dpunkt.verlag, 2010.  
[DAS deutschsprachige Buch zum Thema Testen; dient auch als Grundlage für den Certified Tester des ISTQB.]

SE1 – Level 3

34

## Positives und negatives Testen



- Die gesamte Funktionalität einer Software-Einheit sollte durch eine Reihe von Testfällen überprüft werden.
- Ein **Testfall** besteht aus der Beschreibung der erwarteten Ausgabedaten für bestimmte Eingabedaten.
- Wenn nur **erwartete/gültige** Eingabewerte getestet werden, spricht man von **positivem Testen**.
- Wenn **unerwartete/ungültige** Eingabewerte getestet werden, spricht man von **negativem Testen**.
- Positive Tests erhöhen das Vertrauen in die **Korrektheit**, negative Tests das Vertrauen in die **Robustheit**.



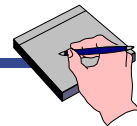
Vorsicht:  
Positiv/Negativ hat  
hier nichts mit  
true/false zu tun!



SE1 – Level 3

35

## Vollständige Tests sind meist teuer...



- In einem **vollständigen** Test werden **alle** gültigen Eingabewerte getestet.
- Vollständige Tests werden auch **erschöpfende** Tests genannt.
- Diese Bezeichnung ist durchaus passend, wie bereits an einem kleinen Beispiel belegt werden kann:

```
int multipliziere(int x, int y);
```

- Ein Test für alle gültigen Eingabewerte dieser Operation würde für Java **sehr** lange dauern...



SE1 – Level 3

36

### ... aber durchaus möglich

- Es gibt Klassen, die sehr einfache Objekte mit nur wenigen Zuständen definieren.
- Wenn alle Methoden einer solchen Klasse ihre Ergebnisse ausschließlich aufgrund dieser Zustände liefern, dann können wir mit wenigen Exemplaren alle möglichen Nutzungssituationen testen.

```
class Schalter
{
    private boolean _istAn;

    public Schalter(boolean anfangsAn)
    {
        _istAn = anfangsAn;
    }

    public void schalten()
    {
        _istAn = !_istAn;
    }

    public boolean istEingeschaltet()
    {
        return _istAn;
    }
}
```

- Exemplare dieser Klasse können sich nur in einem von zwei möglichen Zuständen befinden.
- Für einen vollständigen Test müssen wir zwei Exemplare erzeugen, da es genau zwei verschiedene Anfangszustände gibt.
- Ausgehend von diesen Anfangszuständen können mit wenigen Aufrufen alle Zustände getestet werden.

SE1 – Level 3

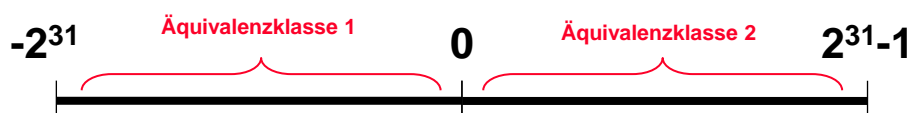
37

### Äquivalenzklassen und Grenzwerte



- Da vollständige Tests nicht praktikabel sind, werden verschiedene Eingabewertebereiche in **Kategorien** eingeteilt. Die Werte eines solchen Wertebereichs werden als für den Test **äquivalent** angesehen.
- Es brauchen dann nicht alle diese Werte getestet zu werden. Stattdessen reicht es, wenige Vertreter einer solchen **Äquivalenzklasse** zu testen.
- Dabei sind besonders die Werte für Tests von Interesse, die am Rande einer Äquivalenzklasse liegen: die so genannten **Grenzwerte**.

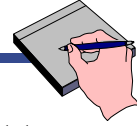
Grenzwertkandidaten für **multipliziere** sind etwa 0, **Integer.MAX\_VALUE** und **Integer.MIN\_VALUE**, zwei Äquivalenzklassen wären beispielsweise die positiven und die negativen **int**-Werte dazwischen.



SE1 – Level 3

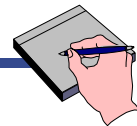
38

## Modultest und Integrationstest



- Wenn die Einheiten eines Systems (Methoden, Klassen) isoliert getestet werden, spricht man von einem **Modultest** (engl.: unit test). Modultests sind eher technisch motiviert und orientieren sich an den programmiersprachlichen Einheiten eines Systems.
- Wenn alle getesteten Einzelteile eines Systems in ihrem Zusammenspiel getestet werden, spricht man von einem **Integrationstest** (engl.: integration test).
- Da erfolgreiche Modultests die Voraussetzung für Integrationstests sind, betrachten wir vorläufig nur Modultests näher.
- Die Methoden zum Modultest lassen sich grob in **Black-Box-**, **White-Box-** und **Schreibtischtests** unterteilen.
- Black-Box- und White-Box-Tests sind **dynamische Tests** (das Testobjekt wird ausgeführt), Schreibtischtests sind **statische Tests**.

## Black-Box-Test



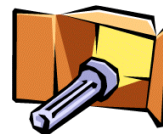
- Ein **Black-Box-Test** betrachtet nur das Ein-Ausgabeverhalten einer Software-Einheit und nicht deren interne Implementation (diese wird als ein „schwarzer Kasten“ angesehen). Die Testfälle können also nur auf Basis der Spezifikation bzw. der Schnittstellendefinition der Software-Einheit formuliert werden.
- Ein Black-Box-Test sollte die zu testende Schnittstelle vollständig abdecken. Unter Zuhilfenahme von Grenzwerten und Äquivalenzklassen sollte etwa bei einer Klasse **jede** Operation der Schnittstelle **mindestens einmal** aufgerufen werden.
- Black-Box-Tests sollten sowohl positiv als auch negativ durchgeführt werden.



## White-Box-Test



- Bei einem **White-Box-Test** wird eine Software-Einheit mit Blick auf ihre Implementation getestet (ein besserer Name wäre deshalb Glas-Box-Test).
- Es werden möglichst **alle Kontrollpfade** des Programmtextes getestet. D.h., bei jeder if-else-Anweisung sollte sowohl der if- als auch der else-Pfad getestet werden, bei jeder switch-Anweisung jedes case-Label angesprochen werden, jede private Methode aufgerufen werden etc.
- Ein White-Box-Test ist somit **noch stärker technisch orientiert** als ein Black-Box-Test. Er testet nicht das, was eine Software-Einheit machen soll, sondern das, was die Software-Einheit tatsächlich tut.
- White-Box-Tests werden häufig nur als Ergänzung von Black-Box-Tests durchgeführt.



SE1 – Level 3

41

## Schreibtischtest



- Beim **Schreibtischtest** wird der Programmtext auf dem Papier durchgegangen und der Programmablauf nachvollzogen. Da der Implementierer oft Fehler in seinem eigenen Programm übersieht, sollte zu solch einem **Walk-Through** eine zweite Person hinzugezogen werden, der der Programmablauf erklärt wird.



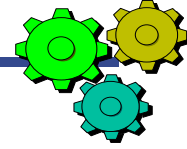
Eine Variante des Schreibtischtest ist ein **Code-Review**. Dieses wird vom Implementierer vorbereitet, indem die relevanten Quelltextteile für alle Teilnehmer (Größenordnung etwa 5 bis 10) des Reviews ausgedruckt werden. Nachdem alle Teilnehmer den Programmtext gelesen haben, wird das Design und mögliche Alternativen diskutiert.

Code-Reviews dienen damit eher der Verbesserung (Laufzeit, Speicherplatz) des Quelltextes als dem Finden von Fehlern.

SE1 – Level 3

42

## Grundregel: zu jeder Klasse eine Testklasse



- Jede Klasse, die wir entwickeln, sollte gründlich getestet werden.
- Um unsere Tests zu dokumentieren und um sie wiederholen zu können, sollten wir sie **ausprogrammieren**.
- Die Grundregeln der objektorientierten Softwareentwicklung lauten deshalb:
  - Zu jeder Klasse existiert eine Testklasse, die mindestens die notwendigen Black-Box-Tests realisiert.
  - Die Testklasse enthält Testfälle für alle Operationen der zu testenden Klasse (jede Operation sollte mindestens einmal aufgerufen werden).
- Da unsere Tests auf diese Weise wiederholbar werden, werden sie zu **Regressionstests**.

## Werkzeugunterstützung für Tests

- Häufig werden aufgrund **mangelnder Disziplin** Tests nur teilweise oder nur gelegentlich durchgeführt.
- Selbst wenn Tests automatisiert durchgeführt werden: **Wer reagiert** in welcher Weise auf die Ausgaben der Testläufe?
- Ein **Werkzeug** kann uns viele der administrativen Aufgaben beim Testen abnehmen.
- Idealerweise ist ein Testwerkzeug **eingebunden** in die **Entwicklungsumgebung**.



## JUnit

- Das bekannteste Werkzeug zur Unterstützung von Regressionstests für Java ist **JUnit**.
- JUnit...
  - ist selbst in Java geschrieben (von Kent Beck und Erich Gamma);
  - stellt einen Rahmen zur Verfügung, wie **Testklassen** geschrieben werden sollten;
  - erleichtert die häufige **Ausführung** dieser Testklassen und vereinfacht die **Darstellung** der Testergebnisse;
  - ist in verschiedene Entwicklungsumgebungen für Java eingebunden, unter anderem auch in BlueJ;
  - ist frei verfügbar: [www.junit.org](http://www.junit.org).



## Umgang mit JUnit

- Zwei Dinge sind zu tun, um einen Modultest mit JUnit durchzuführen:
  - Für eine zu testende Klasse muss eine **Testklasse** erstellt werden, die in ihrem Format den Anforderungen von JUnit entspricht. Diese Testklasse definiert eine Reihe von **Testfällen**.
  - JUnit muss so gestartet werden, dass es die Testfälle dieser neu erstellten Testklasse ausführt.

Testklasse
Testfall 1
Testfall 2
...
Testfall n

## Struktur einer JUnit-Testklasse (bis JUnit 3.8)

- Jede JUnit-Testklasse beerbt die Klasse **TestCase**, die von JUnit zur Verfügung gestellt wird. Durch dieses Beerben stehen in der erbbenden Testklasse etliche Methoden zur Verfügung, die das Testen unterstützen.
- Jeder **Testfall** wird in der Testklasse durch eine parameterlose Methode realisiert, deren Name mit „**test**“ beginnt.
- Ein Beispiel für eine solche **Testmethode**:  

```
public void testEinzahlen() ...
```
- Initialisierungen, die vor jedem Testfall ausgeführt werden sollen, können in der Methode **setUp** vorgenommen werden. Analog können in der Methode **tearDown** „Abbauarbeiten“ nach jedem Testfall definiert werden.



JUnit gibt es inzwischen in der Version 4.0. Diese Version macht u.a. das Beerben der Klasse **TestCase** unnötig. Leider sind die meisten IDEs noch nicht auf JUnit 4.0 umgestellt – auch BlueJ nicht. Wir verwenden deshalb JUnit 3.8.  
Die Unterschiede zwischen JUnit 3.8 und 4.0 sind aber rein technischer Natur – das Prinzip von Unit-Tests wird dadurch nicht berührt.

SE1 – Level 3

47

## Struktur eines Testfalls

- Innerhalb einer Testmethode werden üblicherweise einige Operationen an einem Exemplar der zu testenden Klasse aufgerufen.
- Die Ergebnisse dieser Aufrufe werden überprüft, indem geerbte **assert-Methoden** aus der Klasse **TestCase** aufgerufen werden. Dabei wird üblicherweise das **Ergebnis einer sondierenden Operation** am Testexemplar mit einem **erwarteten Ergebnis verglichen**.
- Stimmen die Werte nicht überein, wird ein **Nichtbestehen** (engl.: failure) signalisiert.

```
public void testEinzahlen()
{
    Konto k;

    k = new KontoSimpel();
    k.einzahlen(100);
    assertEquals("einzahlen fehlerhaft!", 100, k.gibSaldo());
}
```

SE1 – Level 3

48



## Vorgegebene Prüfmethoden

- Die Testklasse erbt von `TestCase` etliche Prüfmethoden, deren Name immer mit `assert` beginnt:
  - Jede Prüfmethode gibt es in zwei Varianten: Entweder mit einem eigenen **Meldungstext** oder ohne.
  - Mit `assertEquals` können zwei Werte (alle Basistypen werden unterstützt) oder Objekte auf Gleichheit geprüft werden.
  - `assertSame` prüft zwei Objekte auf Identität (**also eigentlich: zwei Referenzen auf Gleichheit**).
  - Mit `assertTrue` und `assertFalse` können boolesche Ausdrücke geprüft werden.
  - Mit `assertNull` und `assertNotNull` können Objektreferenzen auf `null` geprüft werden.



## Ausführen der Testfälle

- Die Testfälle einer JUnit-Testklasse werden üblicherweise ausgeführt, indem **für jede Testmethode ein Exemplar** der Testklasse erzeugt wird und darauf die jeweilige Testmethode aufgerufen wird.
- Dies erklärt die Benennung der JUnit-Klasse `TestCase`, von der die Testklasse erbt: Jedes Exemplar dieser Klasse soll für einen Testfall (engl. test case) stehen.
- Die Ausführung wird idealerweise innerhalb der Entwicklungsumgebung angestoßen; in BlueJ stehen bei Bedarf entsprechende Menüeinträge zur Verfügung.
- Laufen alle Tests fehlerfrei durch, erscheint ein **grüner** Balken; schlägt nur ein Test fehl, ist der Balken **rot**!



**Das JUnit-Motto: „Keep the bar green to keep the code clean!“**

## Fehlschlagen von Testfällen: Nichtbestehen vs. Fehler

- Für das Fehlschlagen eines Testfalls werden in JUnit zwei Ursachen unterschieden:
  - Bei einem der Vergleiche zwischen **erwartetem** Ergebnis und tatsächlich **geliefertem Ergebnis** stimmen diese **nicht überein**. In einem solchen Fall entspricht das getestete Objekt nicht den Erwartungen, die der Tester formuliert hat. Dieser Fall bedeutet aus Sicht des getesteten Objektes ein **Nichtbestehen** (in JUnit engl.: **failure**) des Tests, denn die Spezifikation wird nicht erfüllt (aus Sicht des Testers ist er übrigens ein erfolgreicher Testfall, denn der Tester hat ja einen Fehler gefunden).
  - Bei der Ausführung des Testfalles kommt es zu einem anderen Laufzeitfehler, etwa einer **NullPointerException** oder einer **ArithmeticException**. Alle diese sonstigen Fehler werden einfach als **Fehler** (in JUnit engl.: **error**) während des Tests bezeichnet.

Failure == Test nicht bestanden

Error == Fehler bei der Testausführung

## Zusammenfassung



- Testen** ist eine Maßnahme zur Qualitätssicherung.
- Gutes Testen ist anspruchsvoll.
- Testen gehört zum Handwerkszeug.
- In einem objektorientierten System sollte **zu jeder testbaren Klasse** eine **Testklasse** existieren.
- Bereits das Nachdenken über geeignete Testfälle führt häufig zu besserer Software.
- Testwerkzeuge** können uns Teile der Arbeit abnehmen.
- JUnit** ist das bekannteste Werkzeug zur Unterstützung von Regressionstests in Java.