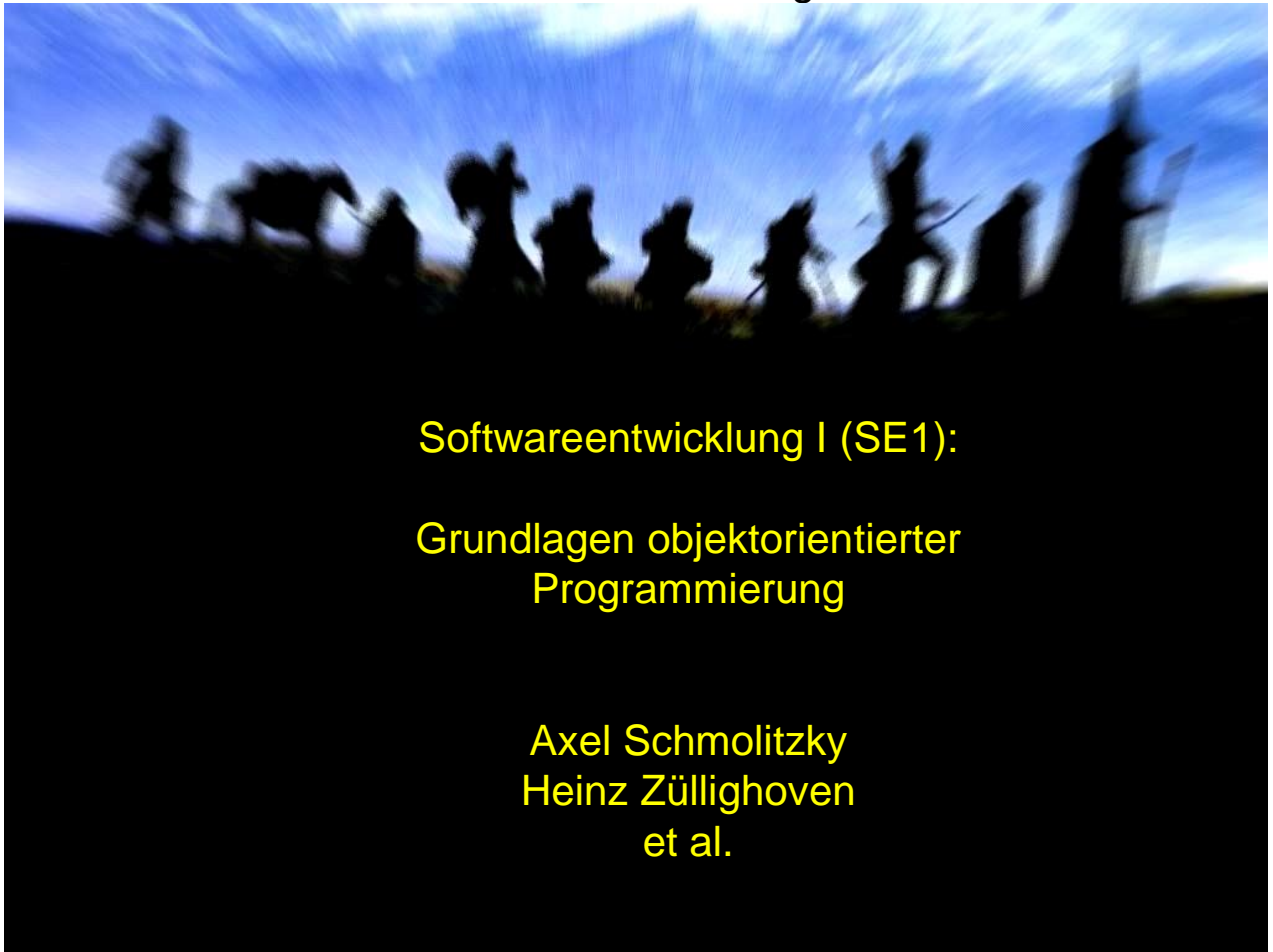
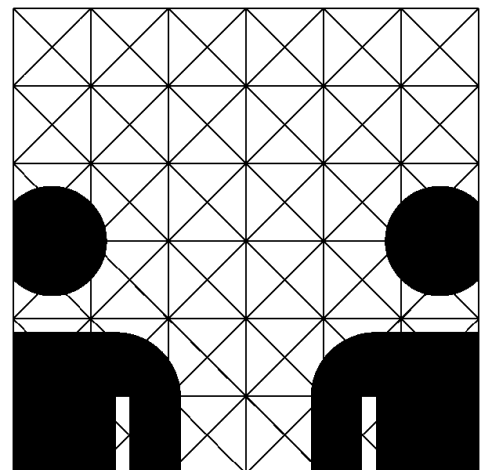


Prüfungsunterlagen
zur Lehrveranstaltung



Teil 3

Universität Hamburg
MIN-Fakultät
Department Informatik
WS 2011 / 2012



Softwareentwicklung I

SE1

Grundlagen objektorientierter Programmierung

Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 3

Verzeichnis der Folien

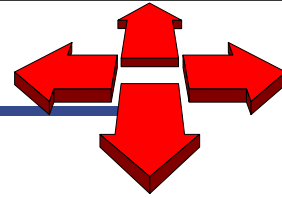
1. **Klassen, Typen und Interfaces**
2. Im Kern von allem: Dienstleister und Klienten
3. Wir erinnern uns: Dienstleistungen an der Schnittstelle
4. Kapselung
5. Die Doppelrolle einer Klasse
6. Trennung von Schnittstelle und Implementation
7. Ein einfaches Beispiel
8. Interfaces in Java
9. Zentrale Eigenschaften von Interfaces
10. Interfaces werden durch Klassen implementiert
11. Auswirkungen auf Klienten
12. Trennung von Schnittstelle und Implementation mit Interfaces
13. Interfaces als Spezifikationen
14. Spezifikation allgemein
15. Klassen und Interfaces definieren Typen
16. Der erweiterte objektorientierte Typbegriff (1. Fassung)
17. Statischer und dynamischer Typ (I)
18. Statischer und dynamischer Typ (II)
19. Methoden und Operationen
20. Java-Objekte vergessen niemals ihre Klasse: Typtests
21. Typzusicherungen
22. UML: Interfaces und Klassen im Diagramm
23. Zusammenfassung
24. **Einführung in das systematische Testen**
25. Motivation: Warum Testen?
26. Software Testing
27. Was ist Testen?
28. Zwei Zitate zum Thema Testen
29. Wann ist Software überhaupt „korrekt“?
30. Testen ist Handwerkszeug
31. Probleme beim Testen
32. Statische und dynamische Tests
33. Positives und negatives Testen
34. Vollständige Tests sind meist teuer...
35. ... aber durchaus möglich
36. Äquivalenzklassen und Grenzwerte
37. Modultest und Integrationstest
38. Black-Box-Test
39. White-Box-Test
40. Schreibtischtest
41. Grundregel: zu jeder Klasse eine Testklasse
42. Werkzeugunterstützung für Tests

- 43. JUnit
- 44. Umgang mit JUnit
- 45. Struktur einer JUnit-Testklasse (bis JUnit 3.8)
- 46. Struktur eines Testfalls
- 47. Vorgegebene Prüfmethode
- 48. Ausführen der Testfälle
- 49. Fehlschlagen von Testfällen: Nichtbestehen vs. Fehler
- 50. Zusammenfassung

51. Objektsammlungen

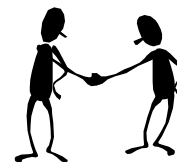
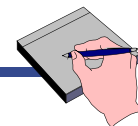
- 52. Mengen und Listen
- 53. Liste, theoretisch
- 54. Umgang mit einer Liste
- 55. Menge, theoretisch
- 56. Umgang mit einer Menge
- 57. Listen und Mengen, objektorientiert
- 58. Der Begriff „Sammlung“
- 59. Wichtige Begriffe zu Sammlungen
- 60. Eigenschaften von Sammlungen in Software
- 61. Der Umgang mit Sammlungen
- 62. Dimensionen möglicher Umgangsformen
- 63. Sammlungen in Java
- 64. Die Standard-Bibliothek von Java: Das Java API
- 65. Das Java-API im javadoc-Format
- 66. Das Interface Set (relevanter Ausschnitt)
- 67. Beispiel einer Set-Benutzung in Java
- 68. Das Interface Set
- 69. Die Operation equals
- 70. Redefinieren von equals
- 71. Aufgepasst: equals und hashCode hängen zusammen!
- 72. Das Interface List (relevanter Ausschnitt)
- 73. Beispiel einer List-Benutzung in Java
- 74. Iterieren über Sammlungen
- 75. Vergleich der Schleifenkonstrukte in Java
- 76. Wrapper-Klassen in Java
- 77. „Boxing“ und „Unboxing“ primitiver Typen
- 78. Auto-Boxing und Auto-Unboxing seit Java 1.5
- 79. Transitivität seit Java 1.5...
- 80. Welche Sammlung ist richtig für meine Zwecke?
- 81. Zusammenfassung

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.

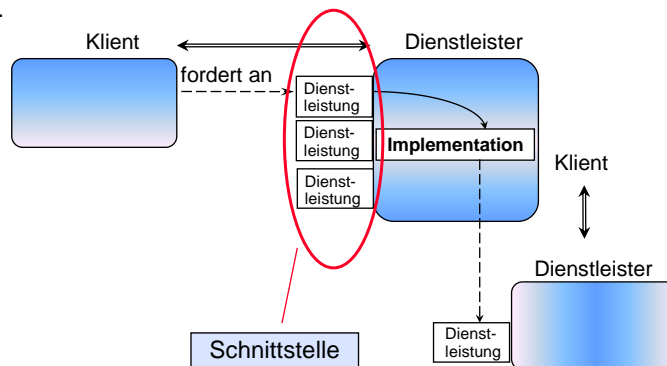
Im Kern von allem: Dienstleister und Klienten



- 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 Teile seiner Dienstleistung von anderen Anbietern einholen.



SE1 - Level 3

3

Kapselung



- Kapselung ist zunächst ein programmiertechnischer Mechanismus, der bestimmte Programmkonstrukte (z.B. Felder oder Methodenrümpfe) 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

4

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**

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.

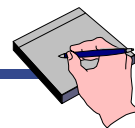


Interfaces in Java

- 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 Methodenrumpfe** (also keine Anweisungen);
 - definieren **keine Felder**;
 - sind **nicht instanzierbar** - es können keine Exemplare von Interfaces erzeugt werden;
 - werden von Klassen implementiert.



Randnotiz: In Java können in einem Interface auch **Konstanten** definiert werden (mit den Modifikatoren **static final**).



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

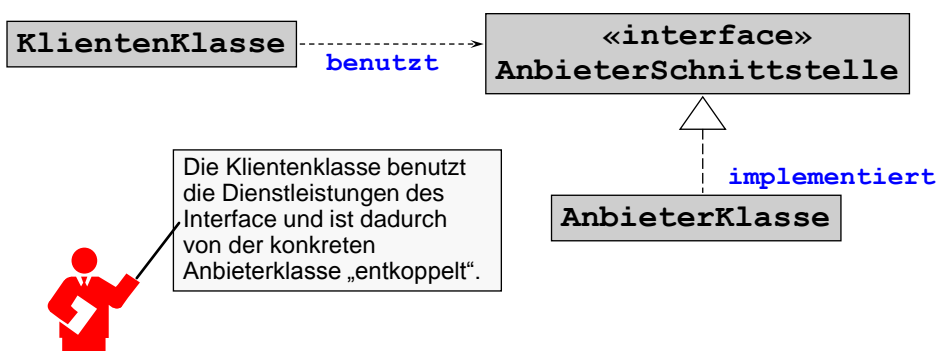
- Bei den Auswirkungen müssen zwei Zusammenhänge unterschieden werden:
 - Objektbenutzung
 - Objekterzeugung
- Bei der **Objektbenutzung** ändert sich durch Interfaces nichts: Klienten können die Operationen des Interface genauso aufrufen wie die einer Klasse.
- Lediglich bei der **Objekterzeugung** 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);
    }
}
```

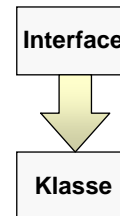
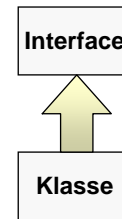
```
Konto konto1 = new KontoSimpel(100);
Konto konto2 = new KontoSimpel(100);
Ueberweiser ueberweiser = new Ueberweiser();
ueberweiser.ueberweise(konto1, konto2, 50);
```

Trennung von Schnittstelle und Implementation mit Interfaces



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.



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.



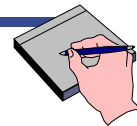
Die Trennung von Spezifikation und Implementation ist ein wesentliches Konstruktionsprinzip der imperativen und objektorientierten Programmierung!

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.

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.



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



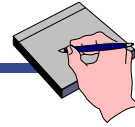
- Jedes dynamisch erzeugte Java-Objekt ist ein Exemplar von genau einer Klasse. Über die gesamte Lebenszeit eines Objektes ändert sich diese Zugehörigkeit zu der erzeugenden Klasse nicht.
- Ein Objekt kann deshalb jederzeit nach seiner erzeugenden Klasse gefragt werden. Java bietet dazu eine binäre Operation 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 **Typtest**, 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



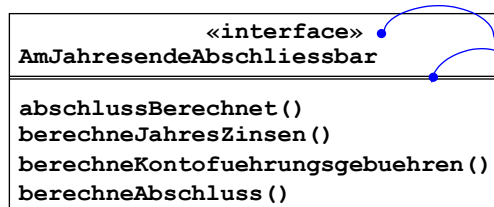
- Klienten sollten ausschließlich mit dem statischen Typ umgehen. Dies sichert ihre Unabhängigkeit gegenüber Änderungen.
- Es gibt jedoch Situationen, in denen Operationen des dynamischen Typs aufgerufen werden müssen. Dies wird durch **Typzusicherungen** ermöglicht.

`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.

UML: Interfaces und Klassen im Diagramm

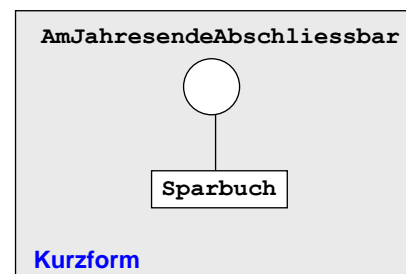
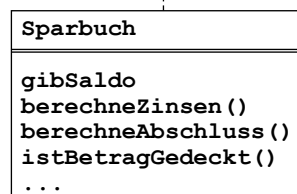


Interface «stereotype»

keine Attribute!



Realisierung



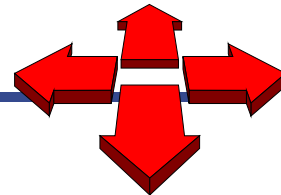
Kurzform



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

- Ergo: Testen kann sich lohnen.



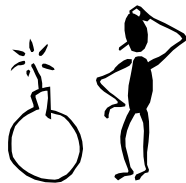
Software Testing

- Robert Binder:
„Software Testing is the execution of code using combinations of input and state selected to reveal bugs.“
- Definition IEEE610:
“The process of operating a system or component under specified conditions, observing the results, and making an evaluation of some aspect of the system or component.”

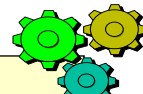


Was ist Testen?

- 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

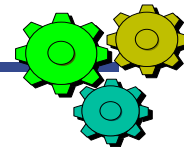


Wann ist Software überhaupt „korrekt“?

- 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

31

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

32

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

33

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

34

Level 3: Schnittstellen mit Interfaces

... 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

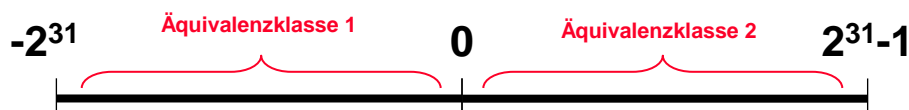
35

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

36

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



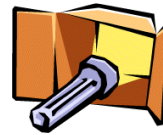
- Ein **Black-Box-Test** betrachtet nur das Ein-Ausgabeverhalten einer Software-Einheit und nicht deren interne Implementation (diese wird als ein „schwarzer Kasten“ angesehen). Die Testfälle können also nur auf Basis der Spezifikation bzw. der Schnittstellendefinition der Software-Einheit formuliert 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

39

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.



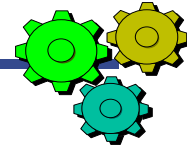
Eine Variante des Schreibtischtest ist ein **Code-Review**. Dieses wird vom Implementierer vorbereitet, indem die relevanten Quelltextteile für alle Teilnehmer (Größenordnung etwa 5 bis 10) des Reviews ausgedruckt werden. Nachdem alle Teilnehmer den Programmtext gelesen haben, wird das Design und mögliche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

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

SE1 – Level 3

40

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.



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.

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());
}
```

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!“

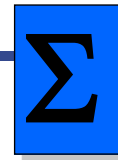
Fehlschlagen von Testfällen: Nichtbestehen vs. Fehler

- 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

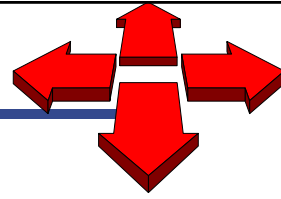
Error == Fehler bei der Testausführung

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.

Objektsammlungen



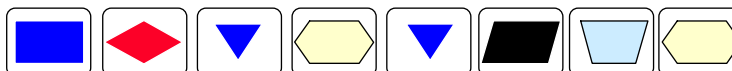
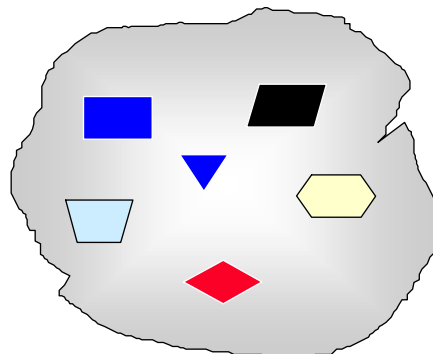
- Bei praktisch jeder größeren Programmieraufgabe werden **gleichartige Objekte zusammengefasst** oder gebündelt und solche **Zusammenfassungen als eigene Objekte** angesehen.
- Solche **Objektsammlungen** werden so häufig benötigt, dass praktisch für jede Programmiersprache vordefinierte Sammlungsbausteine zur Verfügung gestellt werden (entweder in der Sprache oder in ihren Bibliotheken).
- Wir nähern uns Objektsammlungen zunächst über ihre **Benutzung** aus Klientensicht. Erst wenn der **Umgang mit Sammlungen** diskutiert wurde, werden wir uns ihre interne Realisierung ansehen.

SE1 – Level 3

51

Mengen und Listen

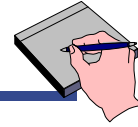
- **Listen** und **Mengen** sind Sammlungen, die in der theoretischen Informatik und in der Softwaretechnik oft verwendet werden.
- **Listen** sind **lineare** Sammlungen von gleichartigen Elementen (Werten), in denen **ein Element mehrfach** auftreten kann.
- **Mengen** sind **ungeordnete** Sammlungen von Elementen (Werten), in denen **jedes Element nur einmal** vorkommt.



SE1 – Level 3

52

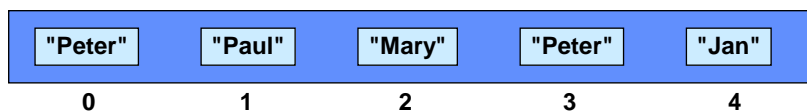
Liste, theoretisch



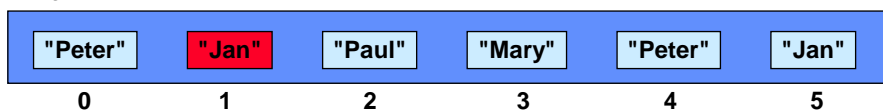
- Listen sind theoretisch betrachtet Aneinanderreihungen von **gleichartigen** Werten zu **Folgen**:
 - die **Reihenfolge** der Listenelemente ist von Bedeutung,
 - ein **Wert** kann in einer Liste **mehrfach** vorkommen.
- Für die theoretische Betrachtung ist es üblich, eine an der Mathematik orientierte **rekursive Definition** einer Liste zu verwenden:
 - Eine Liste ist entweder eine leere Liste (oft notiert als `[]`)
 - oder ein Listenelement gefolgt von einer Liste.
- Listen werden oft sequentiell (d.h. elementweise) durchlaufen, dabei wird eine Operation auf jedes Element der Liste angewendet, und die Reihenfolge wird beachtet.

Umgang mit einer Liste

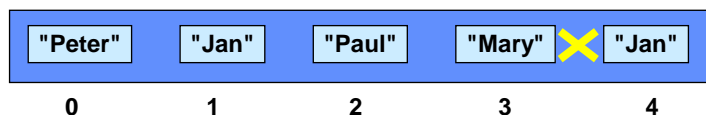
- Beispiel:
Eine Liste von Strings, etwa Namen für die Einteilung der Pausenaufsicht in einer Schule.



- Einfügen eines Namens an zweiter Position:



- Entfernen des zweiten Eintrags für "Peter":

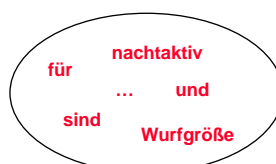


Menge, theoretisch

- **Menge** (engl. set): Eine Sammlung von gleichartigen Elementen, wobei jedes Element nur einmal vorkommt.
- Kann ein Element mehrfach vorkommen, spricht man von **Mehrfach- oder Multimengen** (engl. bag).
- Es gelten die bekannten mathematischen Mengenoperationen. Üblich sind:
 - **insert**: Füge ein Element zur Menge hinzu
 - **delete**: Entferne ein Element aus der Menge
 - **element**: Prüfe, ob ein Element in der Menge vorhanden ist
 - **union**: Vereinige zwei Mengen zu einer neuen.
 - **intersection**: Bestimme die Schnittmenge zweier Mengen.
 - **difference**: Bestimme die Differenzmenge zweier Mengen.
 - **empty**: Prüfe, ob eine Menge leer ist.

Umgang mit einer Menge

- Beispiel **Textanalyse**: Die Wörter eines längeren Textes können wir als eine **Menge von Wörtern** betrachten, wenn uns nicht interessiert, wie häufig ein bestimmtes Wort im Text vorkommt, sondern wir lediglich verarbeiten wollen, **welche Wörter** verwendet werden.
- Wenn wir für mehrere Texte solche Mengen bilden, dann können einige interessante Fragen beantwortet werden:
 - Ist ein bestimmtes Wort in **Text 1** enthalten, in **Text 2** aber nicht?
 - Welche Wörter sind lediglich in **Text 1** enthalten, nicht aber in **Text 2**?
 - Welche Wörter bleiben übrig, wenn wir die **Schnittmenge** zweier Texte von den Wörtern eines **dritten Textes** abziehen?



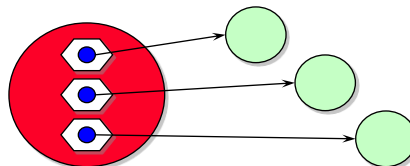
Listen und Mengen, objektorientiert

- Die theoretische Informatik beschreibt Sammlungen mit mathematischen (oft funktionalen) Konzepten.
- Objektorientierte Sammlungen werden eher **zustandsbasiert** betrachtet:
 - So werden z.B. Mengen und Listen als eigenständige Objekte unabhängig von ihren Elementen betrachtet.
 - Eine Menge ist dabei wie ein „ungeordneter Behälter“ für seine Elemente, die eingefügt und herausgenommen werden können.
 - Eine Liste ordnet ihre Elemente in Positionen an. Diese Ordnung kann vordefiniert oder vom Benutzer beeinflusst werden.

Der Begriff „Sammlung“



- Eine **Sammlung** von Objekten wird im Java-Umfeld auch unter dem englischen Begriff **Collection** gefasst.
 - Eine Sammlung ist ein Objekt, das eine Gruppe von anderen Objekten zusammenfasst.
 - Sammlungen werden verwendet, um andere Objekte zu speichern, gemeinsam zu manipulieren und Mengen von Objekten an eine andere weiter zu geben.
 - Sammlungen enthalten in der Regel Objekte **vom selben Typ**: eine Menge von Briefen, eine Menge von Konten.
 - Alternativ gebräuchliche Begriffe für Sammlung sind **Behälter** und **Container**.



Wichtige Begriffe zu Sammlungen



- Die Objekte, die in einer Sammlung gehalten werden, werden als die **Elemente** der Sammlung bezeichnet. Jede Sammlung bietet Operationen zum Einfügen und Entfernen von Elementen.
- Der Typ der enthaltenen Elemente (der **Elementtyp**) wird üblicherweise auch als eine Eigenschaft der Sammlung angesehen (Beispiel: eine Sammlung von Strings).
- Die Anzahl der enthaltenen Elemente bezeichnen wir als ihre **Kardinalität**. Bei Sammlungen ist diese üblicherweise nicht nach oben begrenzt: Sammlungen können **beliebig viele Elemente** enthalten.
- Wenn ein Element, das in eine Sammlung eingefügt wird, bereits in der Sammlung enthalten ist, nennt man das einzufügende Element ein **Duplikat**.

Eigenschaften von Sammlungen in Software

- Auch für Sammlungen gilt: Es muss eine deutliche Unterscheidung zwischen ihrer **Schnittstelle** (ihrem Umgang) und ihrer **Implementation** vorgenommen werden.
- Eigenschaften einer **Sammlungs-Schnittstelle**:
 - Umgang mit Duplikaten (erlaubt oder nicht?)
 - Handhabung einer Reihenfolge
- Eigenschaften einer **Sammlungs-Implementation**:
 - verwendete Datenstrukturen (Array, Verkettung, Kombination)
 - Effizienz (wie schnell sind einzelne Operationen in ihrer Ausführung?)

Ein Klient einer Sammlung ist **in erster Linie** an ihrer Schnittstelle interessiert; wie diese realisiert ist, ist hingegen meist nur zweitrangig.

Der Umgang mit Sammlungen

- Für den Umgang mit einer Sammlung ist wichtig, ob und wie eine Reihenfolge der Elemente gehandhabt wird. Dafür gibt es verschiedene Möglichkeiten:
 - es ist keine Reihenfolge definiert
 - die Reihenfolge ist benutzerdefiniert festgelegt
 - die Reihenfolge wird automatisch durch die Sammlung erstellt
- Außerdem ist wichtig, wie mit Duplikaten umgegangen wird. Wir unterscheiden zwei Möglichkeiten:
 - Duplikate sind zugelassen und erhöhen die Kardinalität.
 - Duplikate sind nicht zugelassen, das Duplikat wird nicht eingefügt.

Dimensionen möglicher Umgangsformen

	Reihenfolge irrelevant	Reihenfolge benutzerdefiniert	Reihenfolge automatisch
Duplikate zugelassen	Multimenge (Bag)	Liste (List)	sortierte Liste (Sorted List)
Duplikate nicht zugelassen	Menge (Set)	geordnete Menge (Ordered Set)	sortierte Menge (Sorted Set)

Sammlungen in Java

- In den Bibliotheken von Java gibt es eine umfangreiche Unterstützung für Sammlungen: das **Java Collections Framework (JCF)**.
- Dieses Framework besteht aus einer Reihe von Interfaces und einer Reihe von Klassen, die diese Interfaces implementieren.
- Dieses Framework hat sich über die Jahre so weit etabliert, dass es wie ein Teil der Sprache betrachtet werden kann.
- Im folgenden betrachten wir exemplarisch die beiden Interfaces **List** und **Set**, mit denen Listen und Mengen modelliert werden.



Die Standard-Bibliothek von Java: Das Java API

- Über die Sprache hinaus gehören zu jeder Java-Installation eine Reihe von Klassen und Interfaces. Diese liegen in einer Bibliothek, die als **Java Application Programmer Interface**, kurz **Java API**, bezeichnet wird.
- Diese Bibliothek ist zergliedert in kleinere Einheiten, in so genannte **Pakete** (engl.: packages).
- Das wichtigste Paket ist **java.lang**. Es enthält alle Klassen und Interfaces, die als Teil der Sprache angesehen werden. Dazu gehört beispielsweise die Klasse **String**, die wir deshalb ohne weiteres benutzen können.
- Klassen und Interfaces aus allen anderen Paketen müssen **importiert** werden, um direkt benutzbar zu sein. Das Java Collections Framework beispielsweise liegt im Paket **java.util**.
- Die entsprechenden **Import-Anweisungen** stehen immer zu Anfang einer Java-Übersetzungseinheit:

```
import java.util.Set;
/**
 * Klassenkommentar
 * ...
```

Das Java-API im javadoc-Format

Eine „Liste“ aller Pakete

Overview (Java 2 Platform SE 5.0) - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop

http://java.sun.com/j2se/1.5.0/docs/api/index.html

Search Print

Java™ 2 Platform Standard Ed. 5.0

Overview Package Class Use Tree Deprecated Index Help

PREV NEXT

FRAMES NO FRAMES

Java™ 2 Platform Standard Edition 5.0

API Specification

This document is the API specification for the Java 2 Platform Standard Edition 5.0.

See: Description

Java 2 Platform Packages

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.

All Classes

AbstractAction

AbstractBorder

AbstractButton

AbstractCellEditor

AbstractCollection

AbstractColorChooserPanel

AbstractDocument

AbstractDocument.AttributeC

AbstractDocument.Content

AbstractDocument.ElementEc

AbstractExecutorService

AbstractInterruptibleChannel

AbstractLayoutCache

AbstractLayoutCache.NodeDir

AbstractList

SE1 - Level 3

65

Das Interface Set (relevanter Ausschnitt)


Zahl der Elemente in der Collection

Suche von Elementen

Einfügen und Entfernen

```
public interface Set<E> extends Collection<E>
{
    // Sondierende Operationen
    int size();
    boolean isEmpty();
    boolean contains(Object o);

    // Verändernde Operationen
    boolean add(E o);
    boolean remove(Object o);
    void clear();
}
```



Dieses Interface (wie viele andere im JCF auch) ist generisch definiert. Im Kopf des Interfaces ist dabei in spitzen Klammern ein Platzhalter für den Elementtyp angegeben, hier <E>. E kann dann in den Signaturen der Operationen als Typ verwendet werden, hier etwa bei add.

Beispiel einer Set-Benutzung in Java

```
// Brainstorming: Wir sammeln Babynamen...
// Wir erzeugen ein Exemplar einer Klasse, die das Interface Set implementiert
Set<String> babynamen = new HashSet<String>();
int anzahl = babynamen.size(); // 0 (anfangs ist die Menge leer)
babynamen.add("Klara");
babynamen.add("Anna");
babynamen.add("Annika");
babynamen.add("Anna");
// Wie viele haben wir schon?
anzahl = babynamen.size(); // 3 ("Anna" war beim zweiten Mal ein Duplikat)
if (!babynamen.contains("Julia"))
{
    babynamen.add("Julia");
}
// "Annika" ist nicht gut...
babynamen.remove("Annika");
anzahl = babynamen.size(); // 3
// Eigentlich alles nicht gut, nochmal von vorn...
babynamen.clear();
```

Anmerkung: Dieses Beispiel illustriert den Umgang mit einem Set, wird aber hoffentlich niemals Eingang in seriösen Quelltext bekommen... ☺

SE1 – Level 3

67

Das Interface Set

- Ein Set definierte keine Ordnung der Elemente.
- Die Semantik von **add** ist so definiert, dass **keine Duplikate** eingefügt werden können.
- Gleichheit von Elementen wird mit der Operation **equals** geprüft.
- Formal gilt für alle **e1 != e2** im Set: **!e1.equals(e2)**
- So genannte Massenoperationen (engl.: bulk operations) haben bei Sets die Bedeutung von mathematischen **Mengenoperationen**.

s1.containsAll(s2): s2 Untermenge von s1 ?
s1.addAll(s2): s1 = s1 vereinigt mit s2
s1.removeAll(s2): s1 = s1 – s2
s1.retainAll(s2): s1 = s1 geschnitten mit s2



SE1 – Level 3

68

Die Operation equals

- Jede Klasse in Java bietet automatisch alle Operationen an, die in der Klasse `java.lang.Object` definiert sind.
- Unter anderem definiert jeder Referenztyp deshalb die Operation `equals`:

```
public boolean equals(Object other)
```

- **Jedes Objekt** kann über diese Operation gefragt werden, ob es gleich ist mit dem als Parameter angegebenen Objekt. Als Parameter kann eine **beliebige Referenz** übergeben werden.
- Die Standardimplementierung vergleicht die Referenz des gerufenen Objektes mit der übergebenen Referenz. Also:

<Test auf Gleichheit> standardmäßig **<Vergleich der Identität>**



Picasso: Mädchen vor dem Spiegel

SE1 – Level 3

69

Redefinieren von equals

- Für bestimmte Klassen ist es sinnvoll, dass sie eine eigene Definition von Gleichheit festlegen.
- Diese Klassen können die vorgegebene Implementation ändern, indem sie eine alternative Implementation angeben. In der Java-Terminologie **redefinieren** (engl.: to redefine) sie die Operation der Klasse `Object`.
- Entscheidend ist: Durch die Redefinition erhalten die Klienten der Klasse ein anderes Ergebnis beim Aufruf der Operation `equals`.
 - Im Fall von Sammlungen: Ein Set verwendet die (gegebenenfalls redefinierte) Operation des Elementtyps.

**Bekanntes Beispiel:
die Klasse String**



SE1 – Level 3

70

Aufgepasst: equals und hashCode hängen zusammen!

- In der Klasse `Object` ist in der Dokumentation der Operation `equals` folgender Hinweis zu finden:

„Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.“

(siehe: <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html>)

- Wenn wir also für die Objekte einer Klasse selbst festlegen wollen, wann zwei Exemplare als gleich anzusehen sind, und deshalb die Methode `equals` redefinieren, dann **müssen** wir auch die Methode `hashCode` (ebenfalls in der Klasse `Object` definiert) so redefinieren, dass sie für zwei gleiche Objekte den gleichen `int`-Wert als Ergebnis liefert.
- Wir nehmen dies erst einmal als **Maßgabe** hin; auf Level 4 von SE1 werden wir die Mechanik von Hash-Verfahren untersuchen und dann diesen Zusammenhang verstehen können.



Das Interface `List` (relevanter Ausschnitt)

Indexbasierter Zugriff auf die Elemente

Indexbasierte Modifikatoren

Bestimmung eines Element-Index

Bildung von Teillisten

```
public interface List<E>
extends Collection<E>
{
    // Alle Operationen wie in Set
    ...

    // zusätzlich: indexbasierte Operationen
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);

    int indexOf(Object o);
    int lastIndexOf(Object o);

    List<E> subList(int from, int to);
}
```



Beispiel einer List-Benutzung in Java

```
// Wir planen die Pausenaufsicht mit einer Liste von Namen...
// Exemplar einer Klasse erzeugen, die das Interface List implementiert
List<String> aufsichtsliste = new LinkedList<String>();
int laenge = aufsichtsliste.size(); // 0 (Liste anfangs leer)
aufsichtsliste.add("Peter");
aufsichtsliste.add("Paul");
aufsichtsliste.add("Mary");
aufsichtsliste.add("Peter");
aufsichtsliste.add("Jan");
// Duplikate sind erlaubt, also:
laenge = aufsichtsliste.size(); // 5

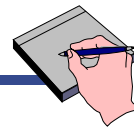
// Jan sollte doch die übernächste machen
aufsichtsliste.add(1,"Jan"); // Einfügen an Position, mit Verschieben des Restes

// Peter hat schon so oft beaufsichtigt...
aufsichtsliste.remove(aufsichtsliste.lastIndexOf("Peter"));
```

Auch dieses Beispiel soll ausschließlich die Kerneigenschaften einer List in Java veranschaulichen.

Die passende Visualisierung findet sich auf Folie 60.

Iterieren über Sammlungen



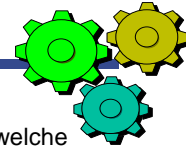
- Seit Java 1.5 gibt es eine **neue for-Schleife** (engl.: for-each loop), mit der sehr elegant über die Elemente einer Collection iteriert werden kann.



```
/**
 * Gib alle Personen in der Liste auf die Konsole aus.
 */
public void listeAusgeben(List<Person> personenliste)
{
    for (Person p: personenliste)
    {
        System.out.println(p.gibName());
    }
}
```

Lies als: „Für jede Person p in der personenliste...“

Vergleich der Schleifenkonstrukte in Java



- Wir kennen nun zwei sehr unterschiedliche for-Schleifen. Wann ist welche anzuwenden?
 - Die **neue for-Schleife** ist ausschließlich für Sammlungen vorgesehen. Wir verwenden sie beispielsweise, wenn wir einheitlich eine Operation auf **allen** Elementen einer Sammlung ausführen möchten.
 - Die **klassische for-Schleife** hingegen ist immer dann gut geeignet, wenn im Schleifenrumpf explizit Zugriff auf den Schleifenzähler benötigt wird oder wenn die Anzahl der Durchläufe vor der Schleifenausführung feststeht.
- Die **while-Schleifen** sollten eher in Fällen verwendet werden, in denen vorab unbekannt ist, wie viele Durchläufe es geben wird (beispielsweise beim Einlesen von Zeilen aus einer Datei oder wenn wir sequentiell in einer Sammlung nach einem Element mit bestimmten Eigenschaften suchen und abbrechen wollen, sobald das Element gefunden wurde).

SE1 – Level 3

75

Wrapper-Klassen in Java

- Als **Elementtyp** für die Sammlungen des Java Collection Framework sind **ausschließlich Referenztypen** zugelassen. Wir können also **nur Objekte** in einer Java Collection verwalten.
- Was aber ist, wenn wir eine **Menge von ganzen Zahlen** in unserer Anwendung brauchen? Oder eine **Liste von booleschen Werten**?
- Damit auch Elemente der primitiven Typen von Java in Sammlungen verwaltet werden können, ist für jeden primitiven Typ eine so genannte **Wrapper-Klasse** definiert:



Primitiver Typ		Wrapper-Klasse
int	→	Integer
boolean	→	Boolean
char	→	Character
long	→	Long
double	→	Double
float	→	Float
short	→	Short
byte	→	Byte

SE1 – Level 3

76

„Boxing“ und „Unboxing“ primitiver Typen

- Ein Wert eines primitiven Typs kann in einem Objekt des zugehörigen Wrapper-Typs „verpackt“ werden (engl.: **boxing**):
`Integer iWrapper = new Integer(42);`
- Die Referenz auf dieses Wert-Objekt kann dann in eine Menge eingefügt werden:
`Set<Integer> intSet = new HashSet<Integer>();
intSet.add(iWrapper);`
- Über die Operationen des Wrapper-Typs kann der verpackte Wert auch wieder „ausgepackt“ werden (engl.: **unboxing**):
`int i = iWrapper.intValue();`
- Für boolesche Werte analog:
`Boolean bWrapper = new Boolean(true);
boolean b = bWrapper.booleanValue();`



Auto-Boxing und Auto-Unboxing seit Java 1.5

- Weil das Ein- und Auspacken primitiver Werte mit Wrapper-Objekten zu aufgeblähten Quelltexten führt, wurden mit Java 1.5 einige Sprachregeln eingeführt, die die **automatische Umwandlung** regeln.
`int i = 42;
Integer iWrapper = i; // Auto-Boxing`
- Dies funktioniert auch als aktueller Parameter:
`List<Integer> intList = new LinkedList<Integer>();
intList.add(i);`
- Auch das Auspacken wurde vereinfacht:
`int i = iWrapper; // Auto-Unboxing`
- Ähnlich wie bei den Typumwandlungen zwischen den primitiven Typen passiert also sehr viel „hinter den Kulissen“!

Transitivität seit Java 1.5...

- Gleichheit ist üblicherweise **transitiv** definiert, mathematisch formuliert:

$$a = b \wedge b = c \Rightarrow a = c$$



- Gegeben folgende Java-Deklarationen:

```
Integer a = new Integer(5);
int b = 5;
Integer c = new Integer(5);
```

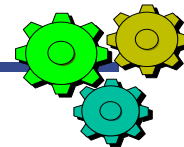
- Dann gilt wegen Auto-Unboxing:

```
a == b // automatisches Unboxing von a
b == c // " " " c
```

- aber:

```
a != c
```

Welche Sammlung ist richtig für meine Zwecke?



1. Auswahl eines Interfaces

- Zuerst sollte klar sein, welcher Umgang mit der Sammlung nötig ist (Duplikate erlaubt, Reihenfolge relevant, etc). Daraus folgt die Entscheidung für ein Sammlungs-Interface. Diese Entscheidung ist **problemabhängig!**
- Alle Variablen im Klienten-Code sollten ausschließlich vom Typ dieses Interfaces sein.

2. Auswahl einer Implementation

- Um mit diesem Interface wirklich arbeiten zu können, muss auch eine Implementation gewählt werden. Als erster Schritt ist jede Implementation ok, die das gewählte Interface implementiert.
- Für **List** gibt es zwei Implementationen im JCF: **ArrayList** und **LinkedList**. Für **Set** gibt es ebenfalls zwei: **HashSet** und **TreeSet**.
- Erst beim Tuning, wenn die Anwendung bereits ihren Zweck erfüllt und korrekt arbeitet, sollten Überlegungen darüber angestellt werden, ob eine andere Implementation eventuell besser geeignet wäre.

Zusammenfassung



- **Sammlungen** von Objekten werden bei praktisch jeder größeren Programmieraufgabe benötigt.
- **Listen** und **Mengen** sind zentrale Sammlungstypen, die viele Anwendungsfälle abdecken.
 - Listen haben eine manipulierbare Reihenfolge der Elemente und lassen Duplikate zu.
 - Mengen verwenden wir für Aufgaben, bei denen wir Duplikate vermeiden wollen und eine Reihenfolge nicht wichtig ist.
- Wir verwenden in Java die Interfaces **List** und **Set** aus dem **Java Collection Framework**, um den **Umgang** mit Listen und Mengen aus Klientensicht zu modellieren.
- Die Implementation dieser Interfaces interessiert uns als Klienten sehr häufig nicht.