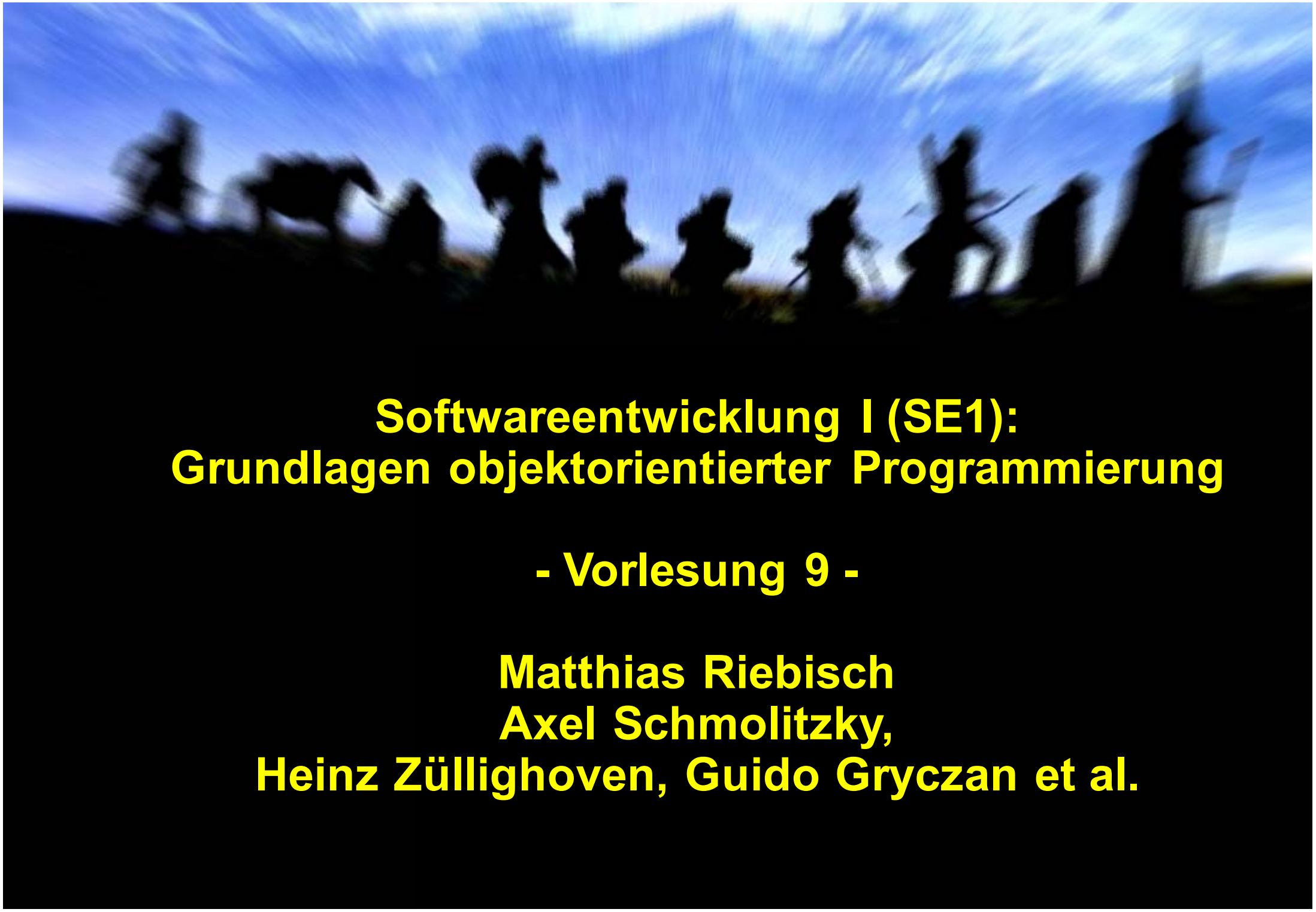
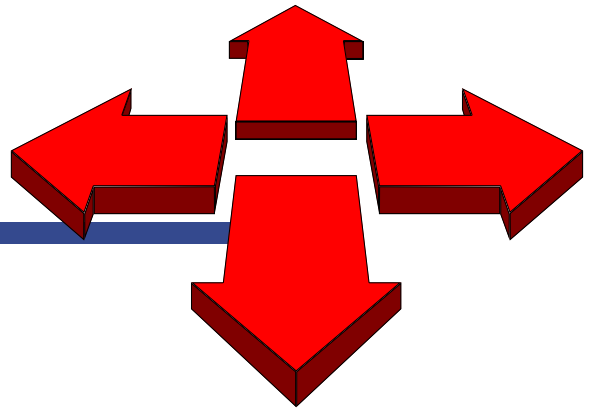


Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 9 -

Matthias Riebisch
Axel Schmolitzky,
Heinz Züllighoven, Guido Gryczan et al.

Klassen, Typen und Interfaces



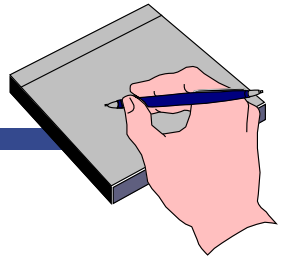
- Wir haben gesehen, dass eine **Klasse** einen **Typ** definiert:
  - Die öffentlichen Methoden einer Klasse bilden die **Operationen** dieses Typs.
  - Die Exemplare der Klasse bilden die **Wertemenge** des Typs.

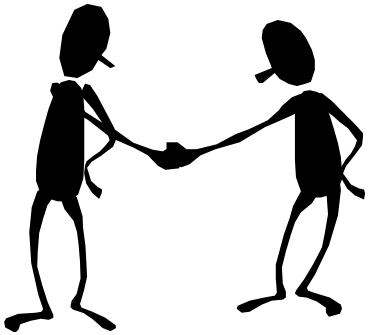
- Eine **Klasse** hat somit zwei zentrale Eigenschaften:
  - Sie definiert einen **Typ**.
  - Sie legt die **Implementation** dieses Typs über ihre Methoden und Felder fest.

- Ein **Interface** in Java definiert ausschließlich einen Typ und legt keine Implementation fest.

- Wir betrachten diesen Zusammenhang näher.

Arbeitsteilung als Ziel: 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.

Arbeitsteilung wozu?

- **Spezialisierung**: Aufteilung nach Wissensgebieten
  - Höhere Qualifikation der Spezialisten gegenüber Generalist

Abläufe in der Produktion	Workflow-Gestaltung
Leistungsfähigkeit bei Hochlast	Loadbalancing von Webservern
Rechtliche Regelungen	Geschäftsregeln, Constraints
Speicherung von Daten	Datenbank-Entwurf, Persistenz

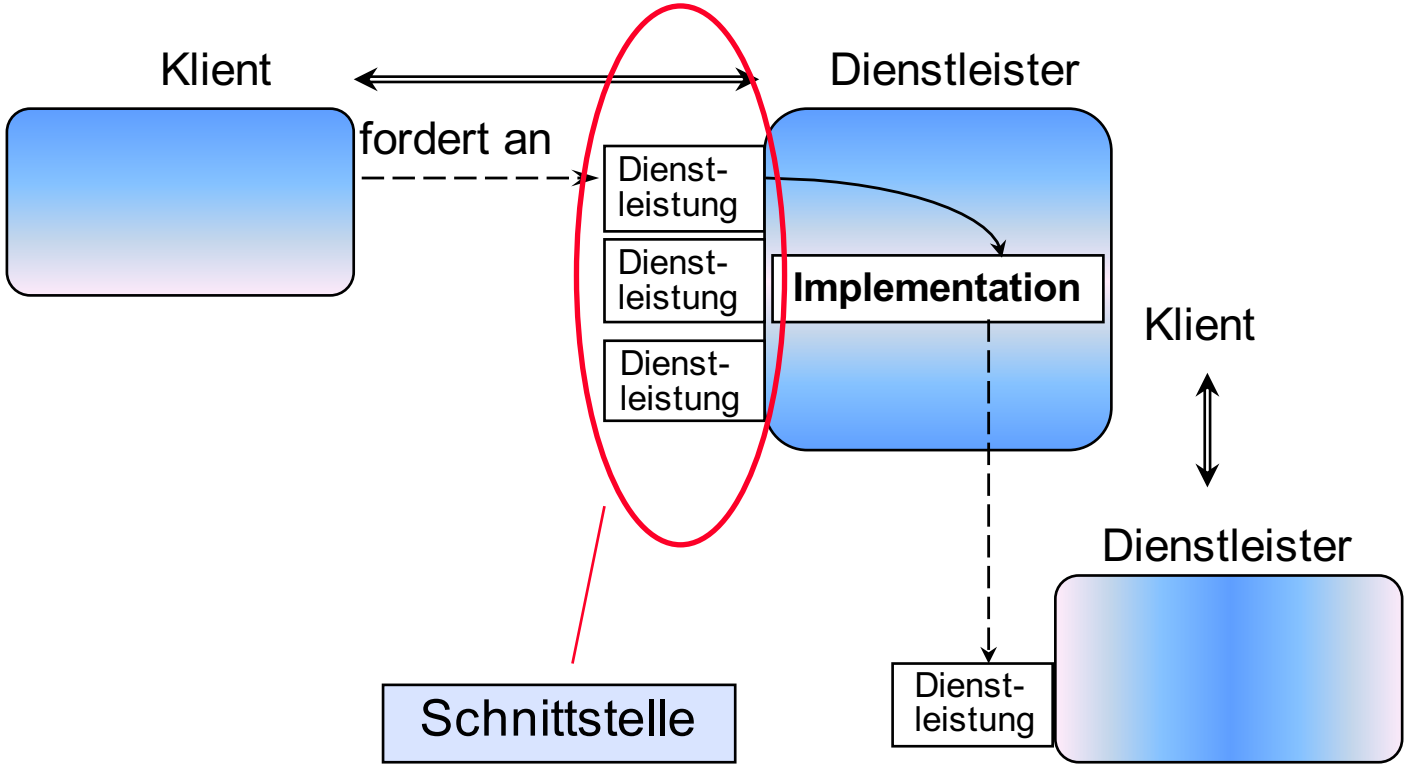
- **Aufteilung** in kleinere (Arbeits-)Einheiten:
  - Große Aufgaben zwischen mehreren Beteiligten

- Vereinfachung zwecks **Abgrenzung** zwischen Beteiligten
  - A kann sich darauf verlassen, dass B die Aufgabe erfüllt
  - A muss nicht verstehen, wie B die Aufgabe erfüllt
  - A ist nicht beeinflusst, wenn B die Aufgabe etwas anders erfüllt

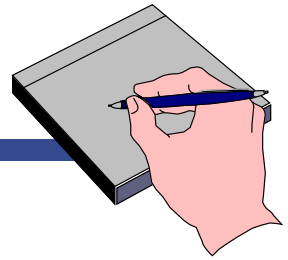
Entwurfsprinzipien
Modularisierung
Kapselung
Information Hiding
Separation of Concerns

Arbeitsteilung mittels Dienstleistungen an 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.



Kapselung – Mittel der Abgrenzung



- 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.

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**?

Außensicht, öffentliche Eigenschaften, Dienstleistungen, Schnittstelle

- Für die **Implementation** der Methoden einer Klasse ist relevant:
  - Wie sind die Operationen in den **Methodenrümpfen** umgesetzt?
  - Welche **Exemplarvariablen/Felder** definiert die Klasse?
  - Welche **privaten Hilfsmethoden** stehen in der Klasse zur Verfügung?

Innensicht, private Eigenschaften, Implementation

Trennung von Schnittstelle und Implementation

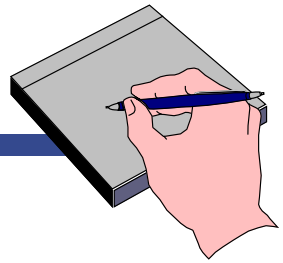
- Für einen **Klienten** ist die Art und Weise der Realisierung (die Innensicht) uninteressant. Er abstrahiert von der Umsetzung und ist ausschließlich an der gebotenen **Dienstleistung** (der Außensicht) interessiert.
- Diese beiden Aspekte einer Klasse lassen sich **begrifflich** und **technisch** 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**.

Entwurfsprinzipien
Modularisierung
Kapselung
Information Hiding
Separation of Concerns

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

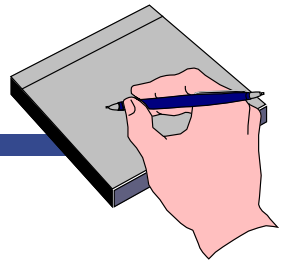


- Java stellt ein spezielles Sprachkonstrukt zur Verfügung, mit dem ausschließlich eine Schnittstelle definiert werden kann: **benannte Schnittstellen** (engl.: named interface).
  Sie werden mit dem Schlüsselwort `interface` definiert.
- Für das Konto-Beispiel sieht die benannte Schnittstelle so aus:

```
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 instanziierbar** - 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.

Das Interface verpflichtet den Entwickler der Klasse

```
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

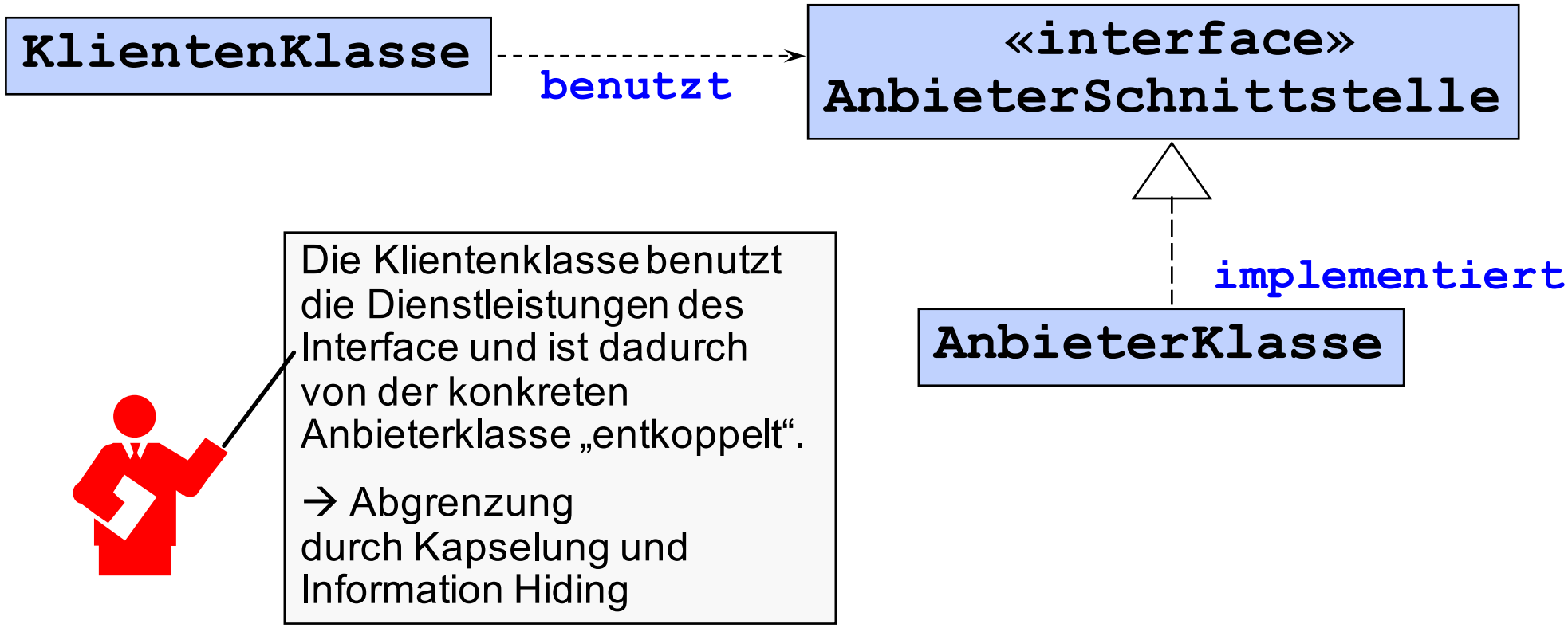
- Bei den Auswirkungen müssen zwei Zusammenhänge unterschieden werden:
  - Objektbenutzung
  - Objekterzeugung
- Bei der **Objektbenutzung** ändert sich durch Interfaces nichts: Klienten können die Operationen des Interface genauso aufrufen wie die einer Klasse.
- Lediglich bei der **Objekterzeugung**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

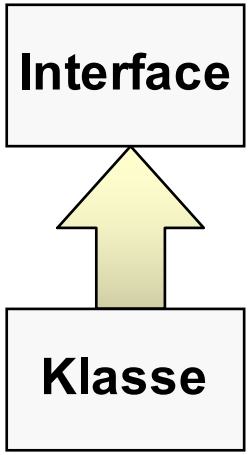
```
Konto konto1 = new KontoSimpel(100);
Konto konto2 = new KontoSimpel(100);
Ueberweiser ueberweiser = new Ueberweiser();
ueberweiser.ueberweise(konto1, konto2, 50);
```

Trennung von Schnittstelle und Implementation mit Interfaces

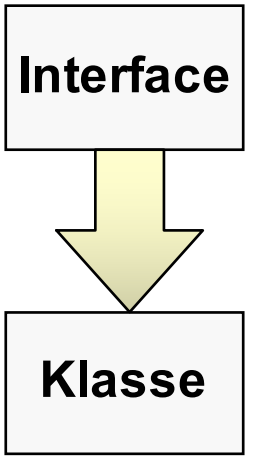


Interfaces als Spezifikationen

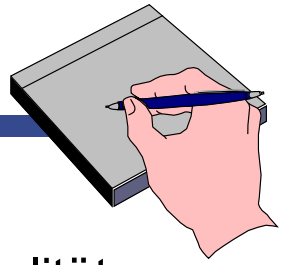
- Im Konto-Beispiel haben wir aus einer Klasse ihre Schnittstelle herausgezogen, indem wir sie als Interface formuliert haben.
  - Wir haben das Interface aus der Klasse **abgeleitet**.



- **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!

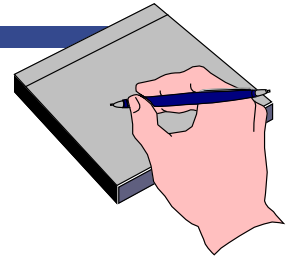
Klassen und Interfaces definieren Typen

- 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)

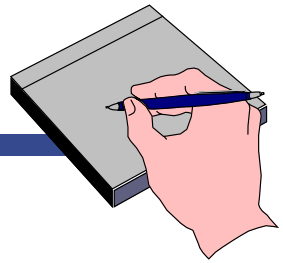


- Der **Begriff Datentyp** in nicht-objektorientierten Programmiersprachen:
  - Ein Typ definiert eine Menge an Werten, die eine Variable oder ein Ausdruck annehmen kann.
  - Jeder Wert gehört zu genau einem Typ.
  - Die Typinformation ist statisch aus dem Quelltext ermittelbar.
  - Ein Typ definiert die zulässigen Operationen.
- Der **Typbegriff objektorientierten Sprachen:** sogenannter **„Abstrakter Datentyp“**
  - Wert und Operationen werden in Objekt gekapselt.
  - Ein Typ definiert das Verhalten von Objekten durch eine Schnittstelle, ohne die Implementation der Operationen und des inneren Zustands festzulegen.

Abstrakter Datentyp → Klasse

- Folge:
  - Ein Objekt wird von **genau einer** Klasse erzeugt.

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
```

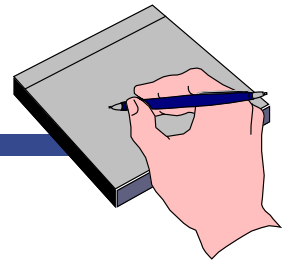
- Der statische Typ legt die **Operationen** fest, die über die Variable aufrufbar sind.

```
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.

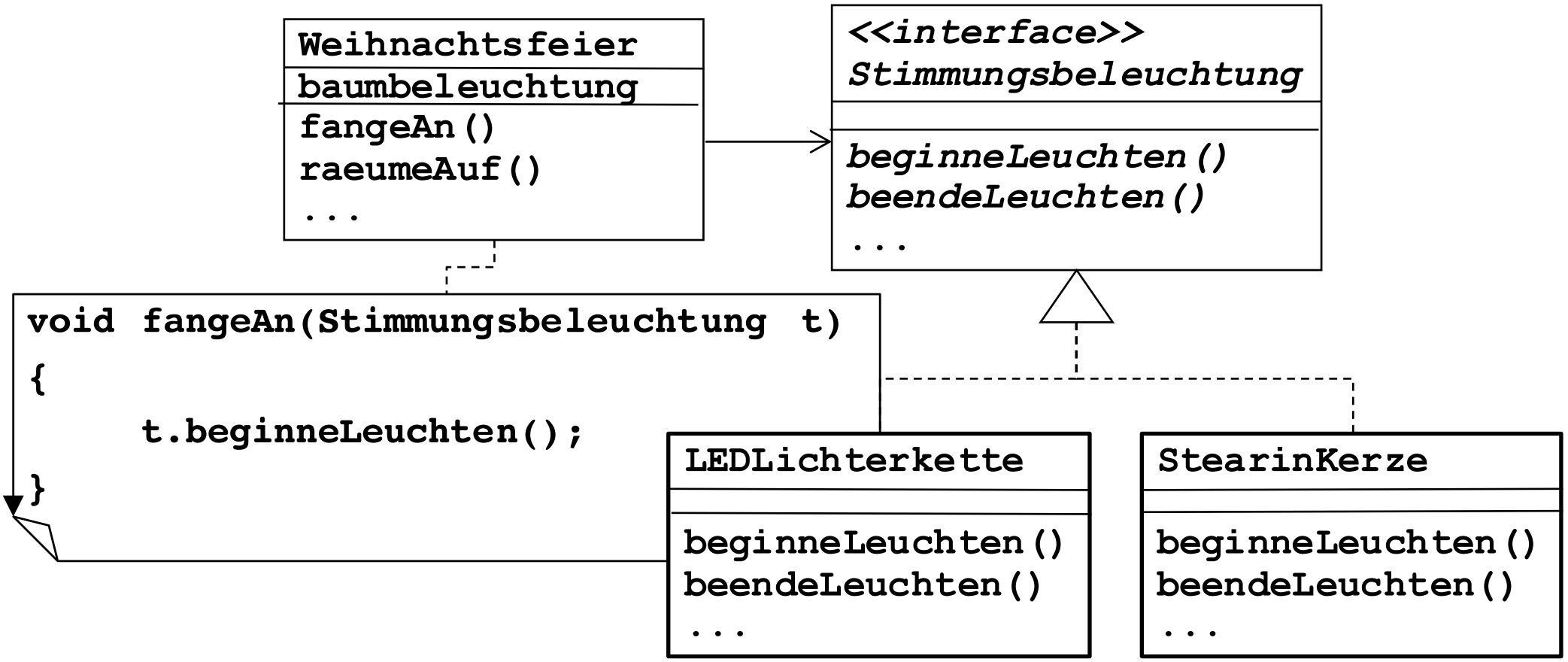
```
k = new KontoAnders();  // neuer dynamischer Typ von k: KontoAnders
```

- Ein **Objekt** hingegen **ändert seinen Typ nicht**; es bleibt sein Leben lang ein Exemplar seiner Klasse.
- Welche **konkrete Methode** eines **referenzierten Objekts** bei einem **Operationsaufruf** ausgeführt wird, hängt vom Typ des Objekts ab.

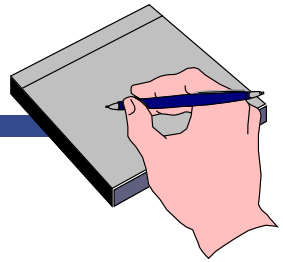
Dynamischer Typ und dynamisches Binden

Wenn ein Objekt über eine Referenz(-variable) erreichbar ist, dann gilt:

- Welche **konkrete Methode** eines **referenzierten Objekts** bei einem **Operationsaufruf** ausgeführt wird, hängt vom Typ des Objekts ab.



Methoden und Operationen



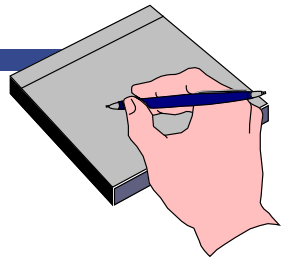
- Mit der konsequenten Unterscheidung von Schnittstelle und Implementation ist eine weitere begriffliche Differenzierung zwischen **Methode** und **Operation** sinnvoll:
  - Ein **Typ** definiert **Operationen**.
  - Eine **Klasse** hat **Methoden**.
- Da eine Klasse auch einen Typ definiert, gilt zusätzlich:
  - **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.

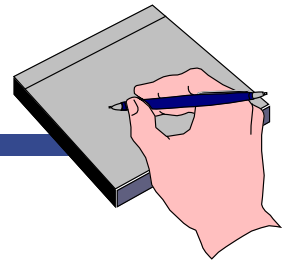
Java-Objekte vergessen niemals ihre Klasse: Typtests



- 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

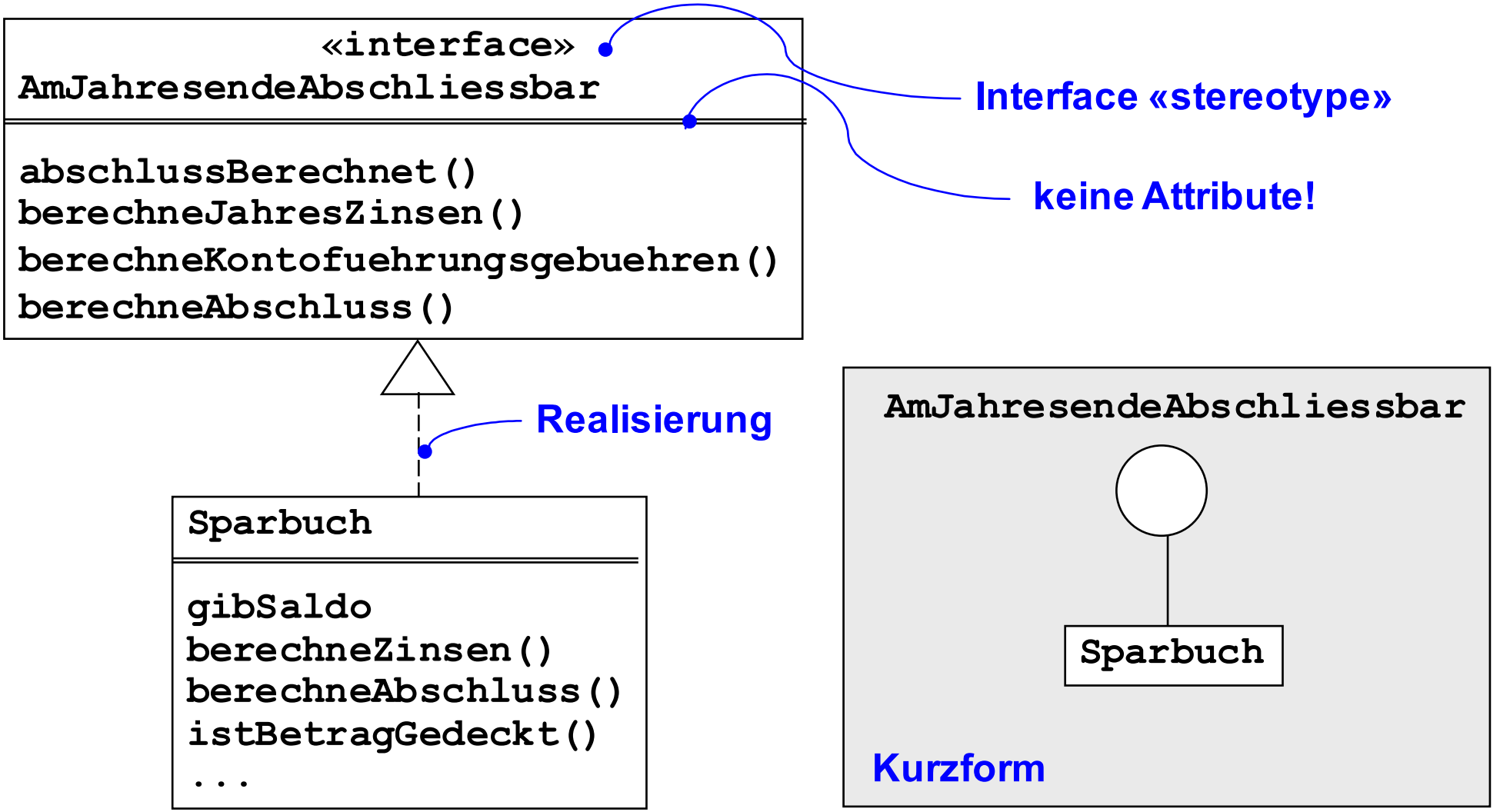


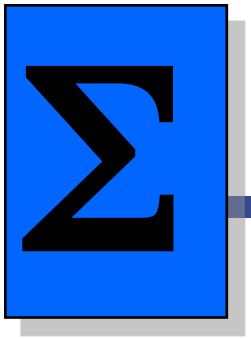
- Klienten sollten ausschließlich mit dem statischen Typ umgehen. Dies sichert ihre **Unabhängigkeit gegenüber Änderungen**.
- Es gibt jedoch Situationen, in denen Operationen des dynamischen Typs aufgerufen werden müssen. Dies wird durch **Typzusicherungen** ermöglicht.

```
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

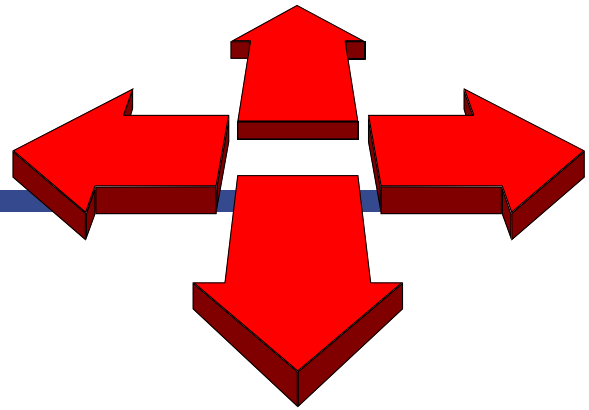




Zusammenfassung

- 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
- Testgetriebene Entwicklung
- Positives und Negatives Testen
- 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.weltraumfacts.info/wf-Dateien/wf1996-Dateien/ariane501.pdf

- 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.

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."

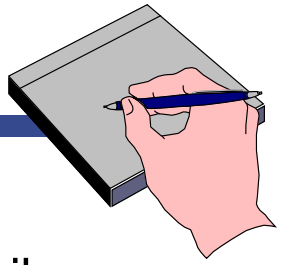
Donald Knuth

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: die Spezifikation ist selbst **formal definiert**.
- Problem: Wie kann nachgewiesen werden, dass **die Spezifikation selbst korrekt ist?**

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

Positives und negatives Testen



- Die gesamte Funktionalität einer Software-Einheit sollte durch eine Reihe von Testfällen überprüft werden.
- Ein **Testfall** besteht aus Eingabedaten sowie der Beschreibung der erwarteten Ausgabedaten.
- Wenn nur **erwartete/gültige** Eingabewerte getestet werden, spricht man von **positivem Testen**.
- Wenn **unerwartete/ungültige** Eingabewerte getestet werden, spricht man von **negativem Testen** – man provoziert Fehlermeldungen.
- 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!

Testgetriebene Entwicklung: „Test-First Principle“

1. Spezifikation der Schnittstelle entwickeln: Was soll eine Klasse tun
2. Testklasse erstellen mit Tests für
   - das erwünschte fehlerfreie Verhalten,
   - schon bekannte Fehlschläge oder
   - das nächste Teilstück an Funktionalität, das neu implementiert werden soll.
3. Code der Klasse schreiben / ändern, bis sie den Test erfüllt
4. Code überarbeiten: vereinfachen, aufräumen, Lesbarkeit verbessern

Wiederhole 1. bis 4. bis Implementierung abgeschlossen ist

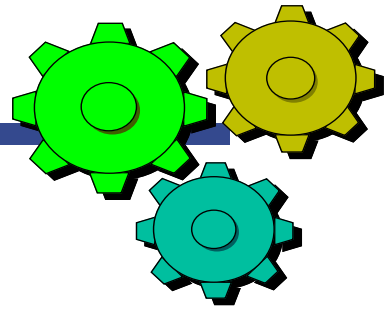
Aufwand hoch?

Bei häufigen Überarbeitungen und langwieriger Fehlersuche höher

Disziplin erforderlich?

Ja, aber Belohnung durch mehr Vertrauen und viel geringeren Überarbeitungsaufwand

Grundregel: zu jeder Klasse eine Testklasse



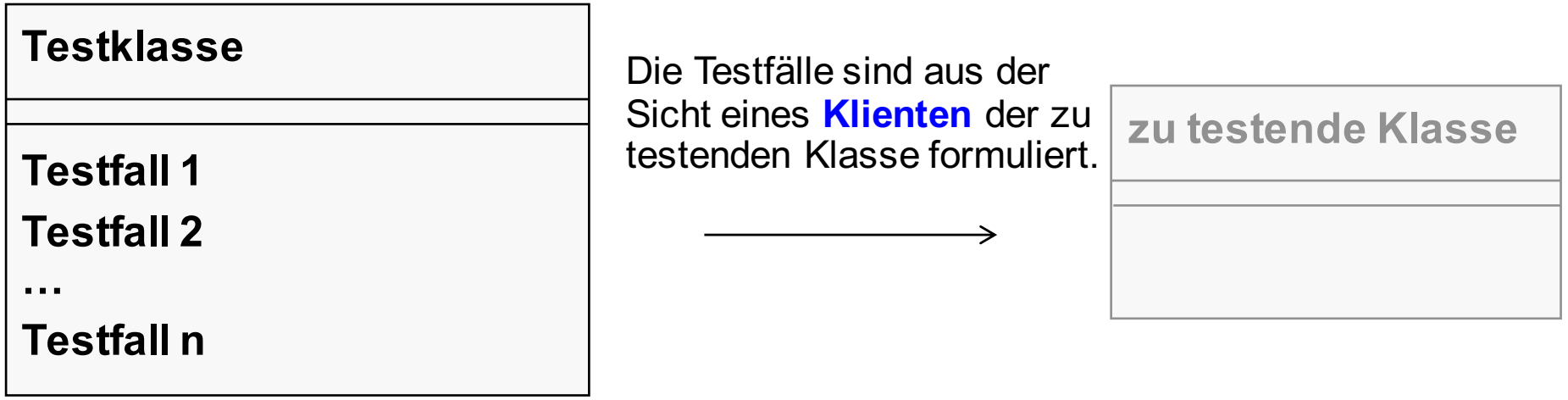
- Jede Klasse, die wir entwickeln, sollte gründlich getestet werden.
- Um unsere Tests zu dokumentieren und um sie wiederholen zu können, sollten wir sie **ausprogrammieren**.
- Die Grundregeln der objektorientierten Softwareentwicklung lauten deshalb:
  - Zu jeder testbaren Klasse existiert eine Testklasse, die mindestens die Tests entsprechend der Spezifikation realisiert.
  - Eine Testklasse enthält Testfälle für die gesamte Schnittstelle der zu testenden Klasse, jede Operation sollte mindestens einmal aufgerufen werden.
- Tests werden auf diese Weise wiederholbar, sogenannte **Regressionstests**.

Werkzeugunterstützung - 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**, jeder Testfall wird dabei in einer eigenen Methode implementiert.
  - JUnit muss so gestartet werden, dass es die Testfälle/Testmethoden dieser neu erstellten Testklasse ausführt.



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 **assert-Methoden** 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.

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

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

Ausführen der Testfälle

- Die Testfälle einer JUnit-Testklasse werden üblicherweise ausgeführt, indem JUnit **für jede Testmethode ein neues Exemplar** der Testklasse erzeugt und an diesem ausschließlich die jeweilige Testmethode aufruft.
  - ➢ Jede Testmethode kann von einem „frischen“ Objekt ausgehen. Somit gibt es auch keine „Reihenfolge“ der Testfälle in einer Testklasse.

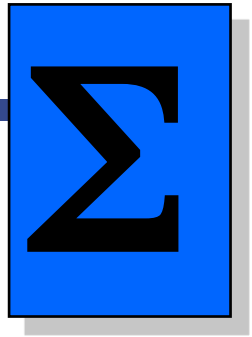
- Die Ausführung wird idealerweise innerhalb der Entwicklungsumgebung angestoßen; in BlueJ stehen bei Bedarf entsprechende Menüeinträge zur Verfügung.
- Laufen **alle Tests fehlerfrei** durch, erscheint ein **grüner** Balken; schlägt hingegen auch nur **ein Test fehl**, ist der Balken **rot**!

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

Vorgegebene Prüfmethoden

- JUnit liefert 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.
- In JUnit 4.0 werden diese Prüfmethoden üblicherweise über einen **statischen Import** innerhalb der Testklasse zugänglich gemacht.
- Eine Testklasse in JUnit 3.8 verfügt durch das Ableiten von `TestCase` automatisch über diese Prüfmethoden.

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.