

Software-Entwicklung 1

V09: Interfaces und Testen





Status der 8. Übungswoche



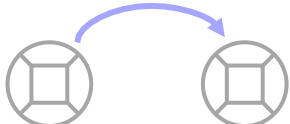
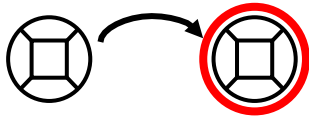
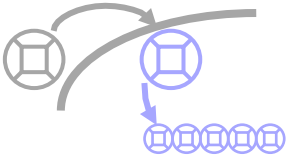
Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vor mittag	Gruppe 1 Erfüllt: 63%	Gruppe 3 Erfüllt: 53%	Gruppe 5 Erfüllt: 59%	Gruppe 6 Erfüllt: 69%	Gruppe 8 Erfüllt: 41%
Nach mittag	Gruppe 2 Erfüllt: 74%	Gruppe 4 Erfüllt: 58%	Vorlesung	Gruppe 7 Erfüllt: 68%	

Tutorium Level 3

- Mittwoch
21.12.16
- 18:30 Uhr
- D-018



Inhaltliche Gliederung von SE1

Stufe	Titel	Themen u.a.	Woche
1	 Algorithmisches Denken	Prozedur, Fallunterscheidung, Zählschleife, Bedingte Schleife	1 – 2
2	 Objektorientierte Programmierparadigma	Klasse, Objekt, Konstruktor, Methode, Parameter, Feld, Variable, Zuweisung, Basistypen	3 – 5
3	 Benutzung von Objekten	Klasse als Typ, Referenz, UML, Schleife, Rekursion, Zeichenketten	6 – 8
4	 Testen, Interfaces, Static, Arrays	Black-Box-Test, Testklasse, Interface, Sammlungen benutzen, Arrays	9 – 10
5	 Sammlungen	Sammlungen implementieren: Array-Liste, verkettete Liste, Hashing; Sortieren; Stack; Graphen	11 – 14

Überblick

1

Klassen und Typen

2

Interfaces

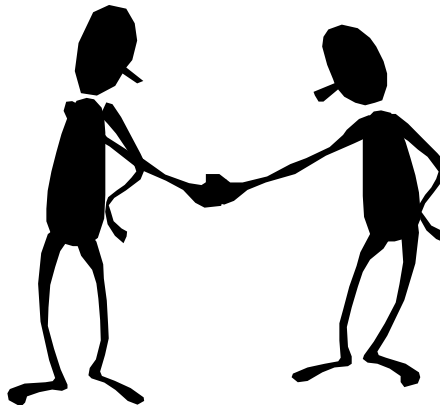
3

Testen

Dienstleister und Klienten

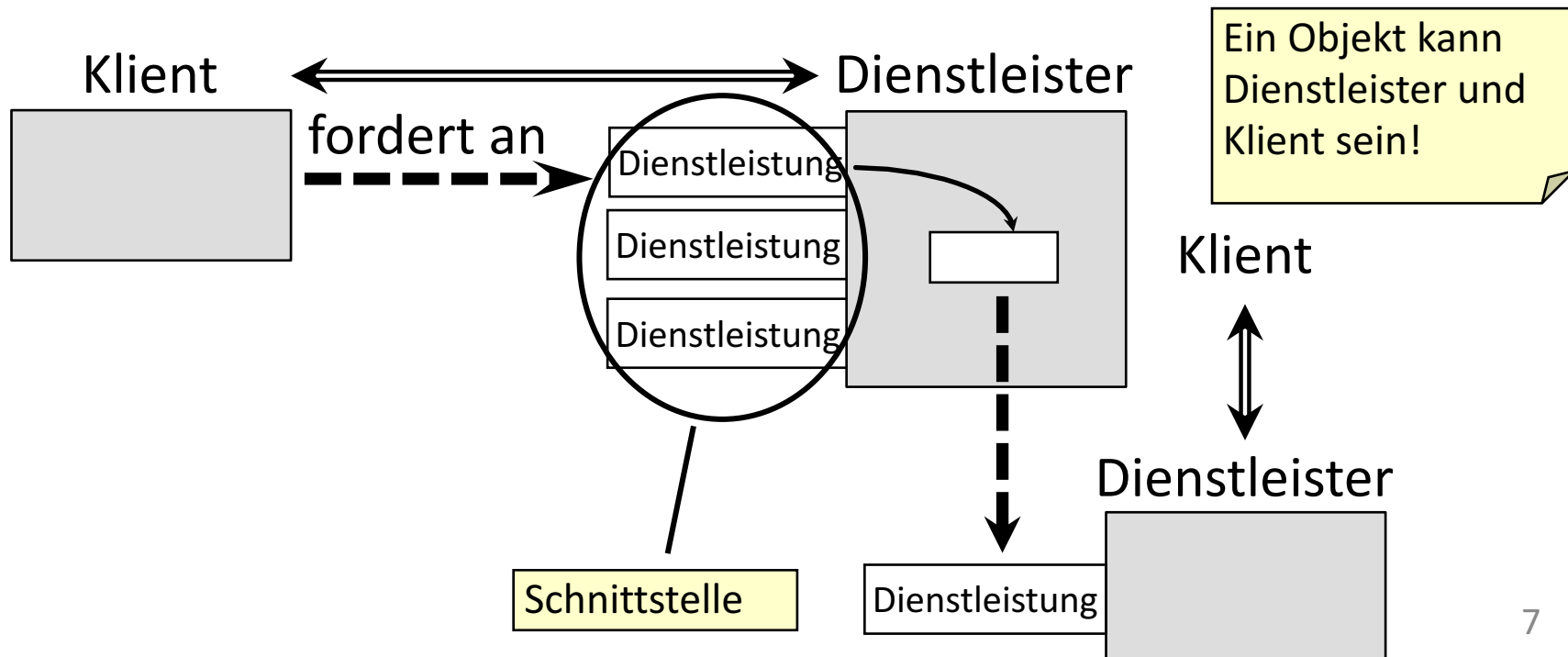


- Nimmt konkrete Dienstleistung eines anderen Objektes in Anspruch
- Leistet bei einer Teilaufgabe einen Dienst

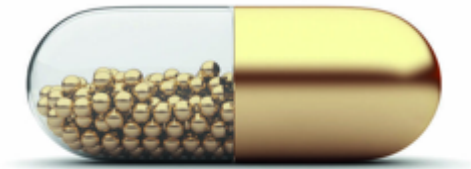


Dienstleistungen an der Schnittstelle

- Objekte bieten Dienstleistungen als **Methoden** an ihrer **Schnittstelle** an
- Dienstleistungen werden von anderen Klienten benutzt
- Klient fordert eine Dienstleistung des Anbieters an
- Der Dienstleister kann selbst Teile seiner Dienstleistung von anderen Dienstleistern einholen



Kapselung



- Schützt den Zugriff auf Programmkonstrukte (z.B. Felder oder Methodenrumpfe) vor äußerem Zugriff
 - In Java mit den Schlüsselworten **public** und **private**
- Klassen sollten eine **Black Box** sein
- Klassen zeigen nur **nur relevante Informationen** nach außen
- Vorteile von Kapselung sind:
 - Das Ausblenden von Details **vereinfacht die Benutzung**
 - Details der Implementation können **geändert** werden, **ohne den Klienten zu ändern**

Doppelrolle einer Klasse



- Für die **Klientensicht**:

- Welche **Operationen** können an den Exemplaren 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 **Implementierung** der Methoden:

- Wie sind die Operationen in den Methodenrümpfen umgesetzt?
- Welche Exemplarvariablen/Felder definiert die Klasse?
- Welche privaten Hilfsmethoden hat die Klasse?

Innensicht,
private
Eigenschaften,
Implementation

Trennung von Schnittstelle und Implementierung



- In **BlueJ** lässt sich entweder die Implementierung einer Klasse oder ihre Schnittstelle anzeigen
- Für die **Benutzung** reicht die **Schnittstellensicht** aus
- Die Java **API** (Application Programming Interface) bietet von allen Bibliotheksklassen als **Dokumentation** die Schnittstellensicht
- Als Konsequenz der Trennung ergibt sich:

Die gleiche Schnittstelle
kann auf **verschiedene Weise implementiert** werden

Konto-Schnittstelle mit 2 Implementierungen

- Klasse **Konto** bietet an ihrer Schnittstelle die Operationen
 - **einzahlen**
 - **auszahlen**
 - **gibSaldo**
- Eine Implementierung benutzt eine **Feldvariable**, um den Saldo zu speichern
 - Jede Ein- und Auszahlung verändert den Wert dieser Variablen
- Eine Implementierung könnte eine **Liste** benutzen:
 - Speichert jede Ein- und Auszahlung in einer Liste
 - Saldo wird erst berechnet, wenn **gibSaldo** aufgerufen wird, indem die Ein- und Auszahlungen aufaddiert werden
- **Für Klienten** würde sich nichts ändern: Er ruft in beiden Fällen die sichtbaren Operationen auf und erhält die gleichen Ergebnisse

Interfaces in Java

- Java bietet Sprachkonstrukt für Schnittstellendefinition

interface

- Konto-Schnittstelle:

```
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**

Überblick

1

Klassen und Typen

2

Interfaces

3

Testen

Eigenschaften von Interfaces

- Sammlungen von **Methodenköpfen**
- Alle Methoden in einem Interface sind implizit **public**
- Enthalten **keine Methodenrumpfe**
- Definieren **keine Felder**
- Sind **nicht instanzierbar** (keine Exemplare)
- Werden von Klassen **implementiert**



Interfaces werden durch Klassen implementiert

- Eine Klasse kann deklarieren, dass sie ein Interface implementiert
- Die Klasse muss für jede Operation des Interfaces eine 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() {...}
}
```

- Immer wenn ein Objekt mit einem bestimmten Interface erwartet wird, kann eine Referenz auf ein Exemplar einer **implementierenden Klasse** verwendet werden

Auswirkungen auf Klienten

- Die Objektbenutzung bleibt durch Interfaces **unverändert**
- Klienten können die Operationen des Interfaces genauso aufrufen wie die einer Klasse

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

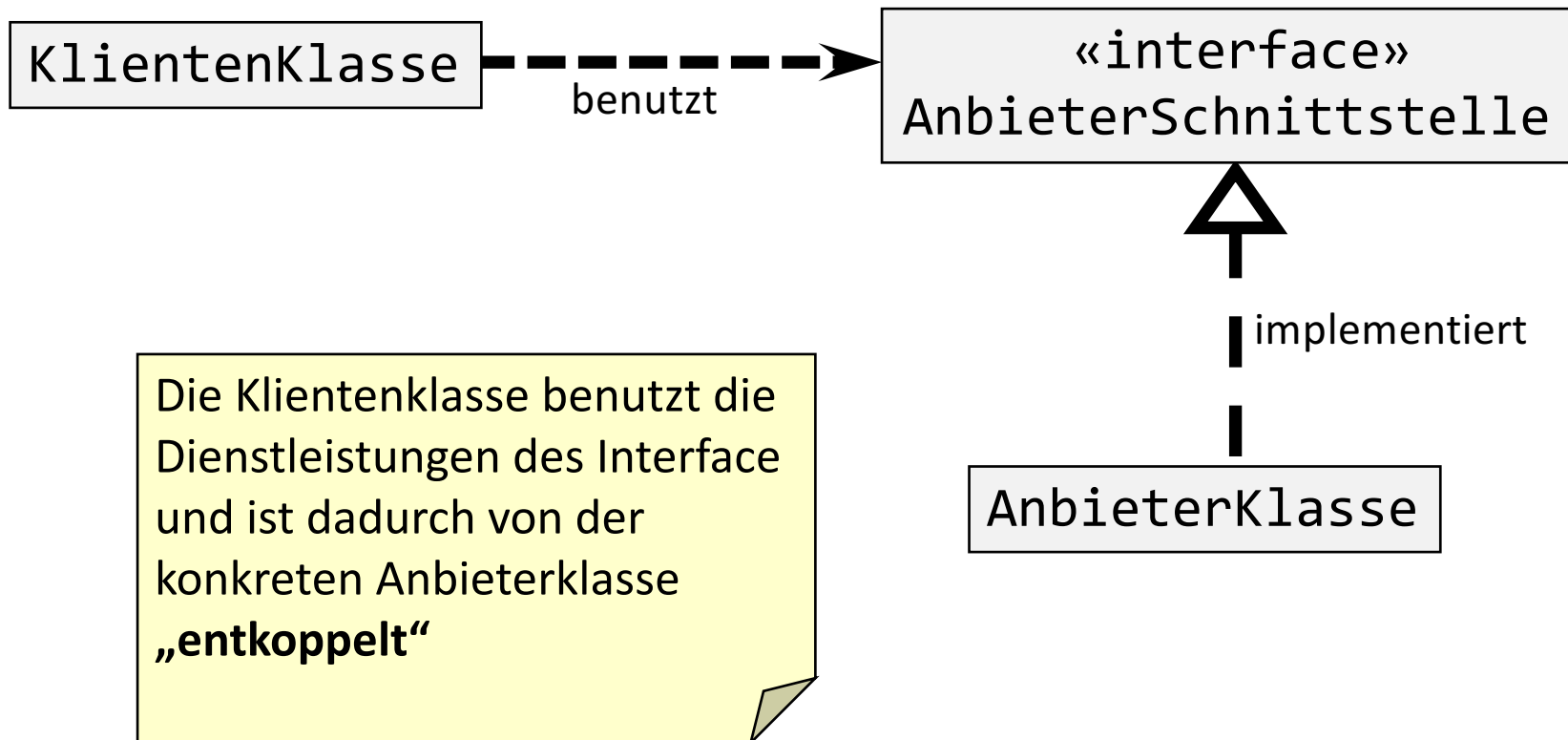
Nutzung:
unverändert!

Auswirkung bei Erzeugung

- Bei der Objekterzeugung muss eine implementierende Klasse angegeben werden

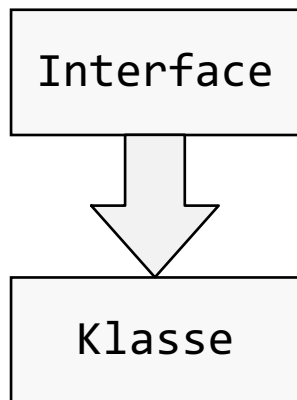
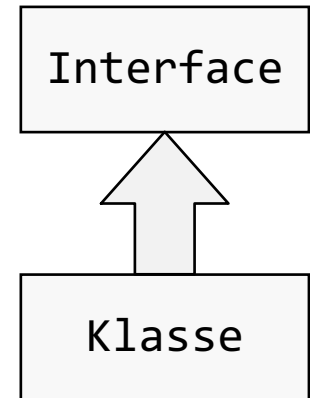
```
Konto konto1 = new KontoSimpel(100);  
Konto konto2 = new KontoSimpel(100);  
Ueberweiser ueberweiser = new Ueberweiser();  
ueberweiser.ueberweise(konto1, konto2, 50);
```

Trennung von Schnittstelle und Implementation mit Interfaces



Interfaces als Spezifikationen

- Das Interface Konto ist aus einer Klasse **abgeleitet**
- Die Methoden wurden in einem Interface beschrieben



- Wir definieren ein Interface und legen den Umgang für einen Typ fest
- Die Köpfe der Operationen werden festgelegt (mit Kommentar)
- Das Interface bildet eine **Spezifikation**, die Klasse eine mögliche Realisierung

Spezifikation allgemein

- Beschreibung der gewünschten Funktionalität einer (Software-)Einheit
- **Was** soll die Einheit leisten?
- Aber **nicht, wie** sie diese Leistung erbringt wird
- Wir unterscheiden
 - informell (natürlichsprachliche)
 - formal (z.B. mathematische)



Klassen und Interfaces definieren Typen

- Jede Klasse definiert 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 OO Sinne ist wichtig:
 - Welche Objekte gehören zur Wertemenge des Typs
 - Welche Operationen sind auf diesen Objekten zulässig
 - NICHT wie die Operationen implementiert sind



Erweiterter objektorientierter Typbegriff

- Der **klassische** Typbegriff:
 - Ein Typ definiert eine **Menge an Werten**
 - 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 Typ

- Unterschied zwischen **statischem** und **dynamischem** Typ einer Referenzvariable
- Der **statische Typ** einer Variablen wird durch ihren **deklarierten Typ** definiert
- Statisch, weil er zur Übersetzungszeit feststeht

```
Konto k; // Konto ist hier der statische Typ von k
```

- Der statische Typ legt die aufrufbaren Operationen der Variable fest

```
k.einzahlen(200); // einzahlen ist hier eine Operation
```

- Ein Compiler überprüft zur Übersetzungszeit, ob die genannte Operation im statischen Typ definiert ist

Dynamischer Typ

- 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
- Dynamischer Typ einer Variablen entscheidet darüber, welche konkrete Methode bei einem Operationsaufruf ausgeführt wird
- Diese Entscheidung wird erst zur Laufzeit getroffen und wird **dynamisches Binden** (einer Methode) genannt

Typstest

- Jedes erzeugte Java-Objekt ist ein Exemplar von genau einer Klasse
- Diese Zugehörigkeit zu der erzeugenden Klasse ändert sich nicht
- Ein Objekt kann mit **instanceof** überprüft werden welcher Klasse es angehört
- Diese boolesche Operation nennen wir im Folgenden einen **Typstest**



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

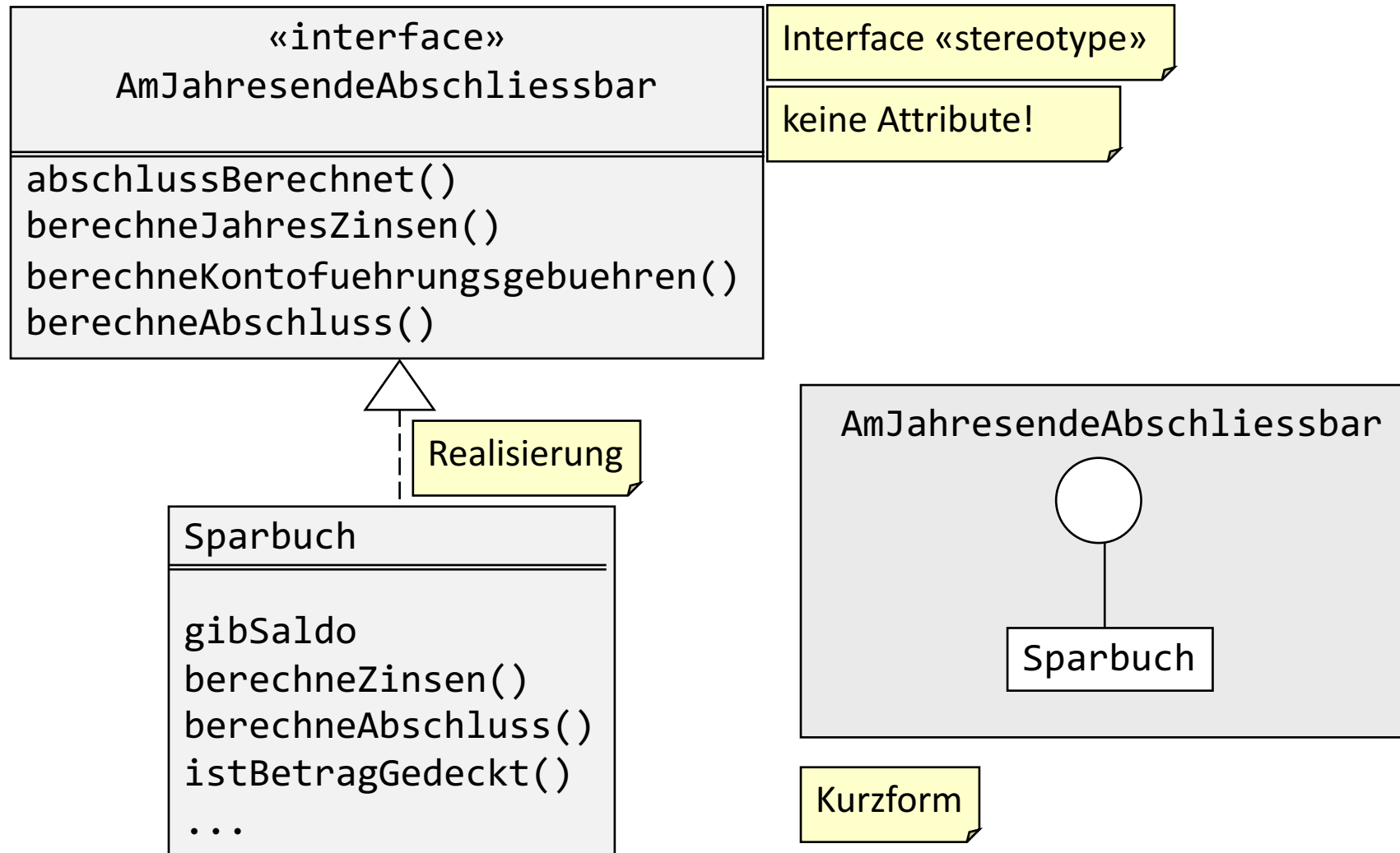
Typzusicherungen

- Klienten sollten ausschließlich mit dem statischen Typ umgehen
- Bei Situationen in denen Operationen des dynamischen Typs aufgerufen werden müssen, muss der Typ zugesichert werden

```
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
 - Ist aber etwas völlig anderes!
- Weder Objekt noch Objektreferenz werden verändert
- Der Compiler erlaubt nun Aufrufe aller Operationen von KontoSimpel 26

UML: Interfaces im Klassendiagramm



Zusammenfassung

1 Die Trennung von **Schnittstelle** und **Implementation** ist ein zentrales Entwurfsprinzip der Softwaretechnik.

2 Aufgrund der Trennung sind zu einer Schnittstelle **unterschiedliche Implementationen** möglich.

4 Interfaces können als **Spezifikationen** eingesetzt werden.

5 Beim Umgang mit Interfaces müssen wir den **statischen** und den **dynamischen Typ** einer Variablen unterscheiden.

Überblick

1

Klassen und Typen

2

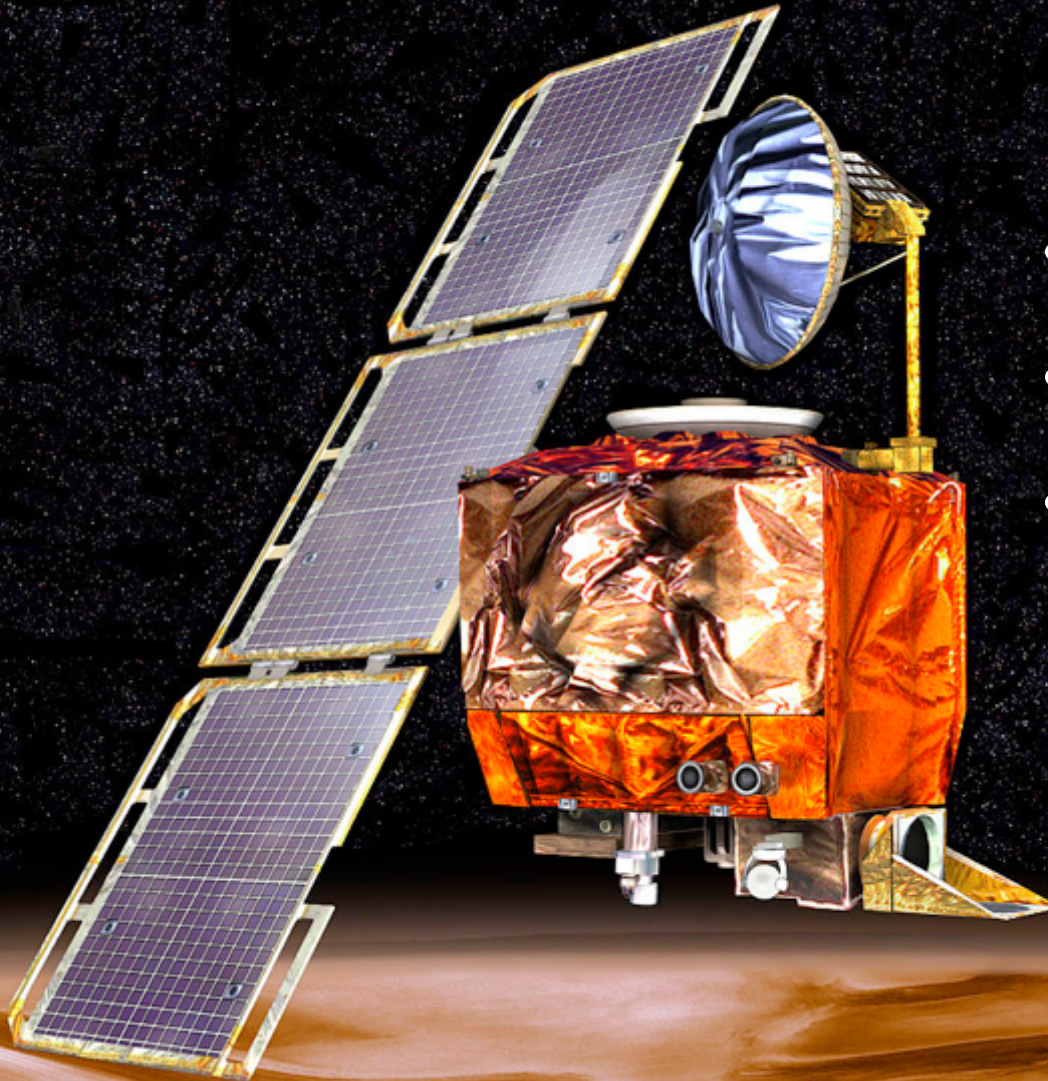
Interfaces

3

Testen

Verschwinden der NASA Sonde 1999

- 6 Jahre Entwicklung
- ~ 300 Millionen Euro
- Softwarefehler in der Einheitsumrechnung



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 im Allgemeinen **nicht** die Korrektheit der Software nachweisen
- Testen ist damit das geplante und strukturierte Ausführen von Programmcode, um **Probleme zu entdecken**



- Selbstverständlicher Bestandteil der Softwareentwicklung
- Tests sollten 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
- Formale Beweise, dass eine Software-Einheit ihre Spezifikation erfüllt, ist aufwendig und schwierig
 - Voraussetzung: Die Spezifikation ist selbst **formal definiert**
 - 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?**
- 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
- Tests müssen gut vorbereitet sein (Testplan)
- Tests müssen wiederholt werden (und damit wiederholbar sein)

Psychologisch:

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

Positiv- und Negativ-Tests



- 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**

Statische und dynamische Tests

- **Statische Tests**

- Beziehen sich auf die Übersetzungszeit und analysieren primär den Quelltext
- Können von Menschen durchgeführt werden (z.B. Reviews) oder mit Hilfe von Werkzeugen

- **Dynamische Tests**

- Sind alle Tests, bei denen die zu testende Software ausgeführt wird

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:

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

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

...aber durchaus möglich

Zum Beispiel bei Klassen mit nur wenigen Zuständen

```
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 sind genau **zwei verschiedene Exemplare** notwendig

Modultest und Integrationstest

- Isoliertes Testen sind **Modultests** (engl.: unit test)
- Wenn alle getesteten Einzelteile eines Systems in ihrem Zusammenspiel getestet werden, spricht man von einem **Integrationstest** (engl.: integration test)
- Wir betrachten Modultests näher, da sie die Voraussetzung für Integrationstests sind:
 - 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 vs White-Box Test



- **Ignoriert** interne **Implementation**
- Beschränkt sich auf Eingabe/Ausgabe Verhalten
- *Funktionales Testen*
- Benutzt für **Validierung**



- Berücksichtigt die interne Implementation
- *Strukturelles Testen*
- Benutzt für **Verifikation**

Beispiel Black-Box-Test

```
public class MyCalendar {  
    public int getNumDaysInMonth(int month, int year)  
    { ... }  
}
```

- Monat: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
- Jahr: (1900, ..., 1999, 2000, ..., 2015)
- Wie viele Fälle brauchen wir, um das komplette Black-Box-Testen von `getNumDaysInMonth()` durchzuführen?

Äquivalenzklassen

Äquivalenzklassen für den Parameter Monat:

- Monate mit 30 Tagen
- Monate mit 31 Tagen
- Februar
- Unzulässige negative Monate (z.B. 0, -1)
- Unzulässige Monate >12 (z.B. 13)

Äquivalenzklassen für den Parameter Jahr:

- Normale Jahr
- Schaltjahre:
 - Alle 4 Jahre
 - Nicht alle 100 Jahre
 - Alle 400 Jahre
- Unzulässige Jahre
- Vor 1900, nach 2015

Schreibtischtest

- Beim **Schreibtischtest** wird der Programmtext auf dem Papier durchgegangen und der Programmablauf nachvollzogen
- Bei einem **Walk-Through** sollte eine zweite Person hinzugezogen werden

```
Calendar work = (Calendar) this.clone();
work.setLenient(true);

// now try each value from getLeastMaximum() to getMaximum() one by one
// we get a value that normalizes to another value. The last value that
// normalizes to itself is the actual minimum for the current date
int result = fieldValue;

do {
    work.set(field, fieldValue);
    if (work.get(field) != fieldValue) {
        break;
    } else {
        result = fieldValue;
        fieldValue--;
    }
} while (fieldValue >= endValue);

return result;
}
```

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.

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 notwendigen Black-Box-Tests realisiert
 - Eine Testklasse enthält Testfälle für die gesamte Schnittstelle 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

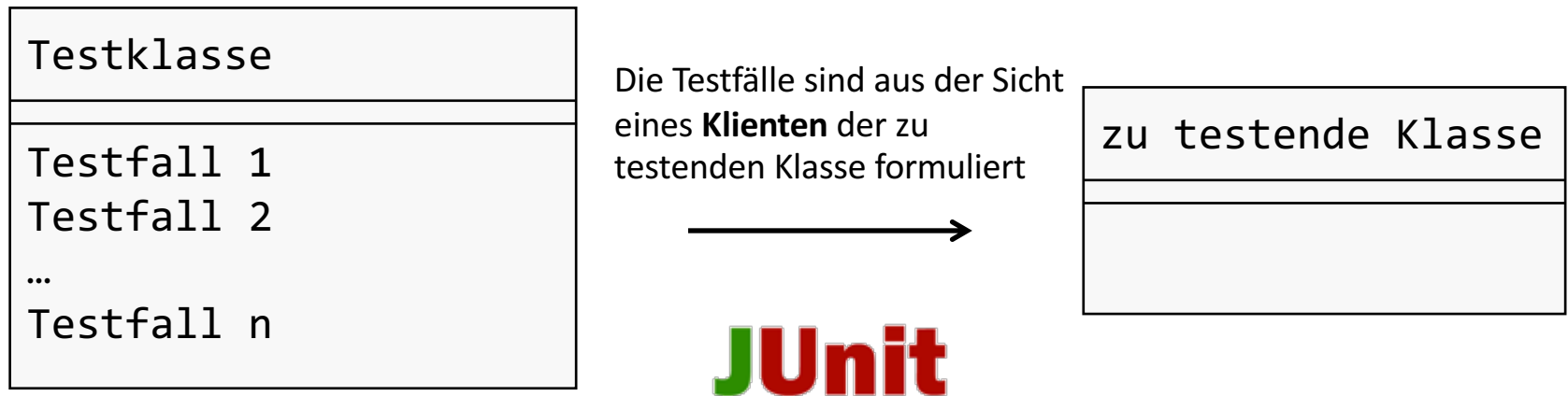
- Häufig werden aufgrund **mangelnder Disziplin** Tests nur teilweise oder nur gelegentlich durchgeführt
- Selbst wenn Tests automatisiert durchgeführt werden: **Wer reagiert** in welcher Weise auf die Ausgaben der Testläufe?
- Ein **Werkzeug** kann uns viele der administrativen Aufgaben beim Testen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

JUnit

- **JUnit** ist das bekannteste Werkzeug zur Unterstützung von Regressionstests für Java
- 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
- In verschiedene Entwicklungsumgebungen für Java eingebunden, unter anderem auch in BlueJ
- Frei verfügbar: **www.junit.org**

Nutzung von JUnit

- Zwei Dinge sind zu tun, um einen Modultest mit JUnit durchzuführen:
 - Erstellung einer **Testklasse** zu einer Klasse, entsprechend dem JUnit-Format
 - 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 eine Testfalls

- Innerhalb einer Testmethode werden Operationen an einem Exemplar der zu testenden Klasse aufgerufen
- Mit **assert-Methoden** werden die Ergebnisse von **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());
}
```

Auführen 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!“



Fehlschlagen von Testfällen: Nichtbestehen vs. Fehler

- Für das Fehlschlagen eines Testfalls werden in JUnit zwei Ursachen unterschieden:
 1. 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).
 2. Bei der Ausführung des Testfalles kommt es zu einem anderen Laufzeitfehler, etwa einer **NullPointerException** oder einer **ArithmeticException**. Alle diese sonstigen Fehler werden als **Fehler** (in JUnit engl.: **error**) während des Tests bezeichnet.

Failure == Test nicht bestanden

Error == Fehler bei der Testausführung

Struktur einer JUnit-Testklasse (JUnit bis 3.8 vs. 4.x)

- **Ab Version 4.0** von JUnit besteht eine Testklasse aus einer Reihe von Testmethoden, die jeweils mit der Annotation **@Test** versehen sein müssen.
- **Bis JUnit 3.8** musste eine Testklasse von der Klasse **TestCase** abgeleitet werden, die von JUnit zur Verfügung gestellt wird.
- Jeder **Testfall** wird in einer 3.8-Testklasse durch eine parameterlose Methode realisiert, deren Name mit „**test**“ beginnen muss.
- Ein Beispiel für eine solche **Testmethode**:
public void testEinzahlen() ...

Vorgegebene Prüfmethoden



- **Prüfmethoden von Junit** beginnend mit `assert` :
 - Prüfmethoden gibt es in zwei Varianten: Mit 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

Zusammenfassung

1

Testen ist eine **Maßnahme zur Qualitätssicherung**. Bereits das Nachdenken über geeignete Testfälle führt zu besserer Software

2

Gutes Testen ist anspruchsvoll und zeitintensiv. **Testwerkzeuge** können uns Teile der Arbeit abnehmen

3

JUnit ist das bekannteste Werkzeug zur Unterstützung von Regressionstests in Java.

4

In einem objektorientierten System sollte **zu jeder testbaren Klasse** eine **Testklasse** existieren.