Programmieren 3

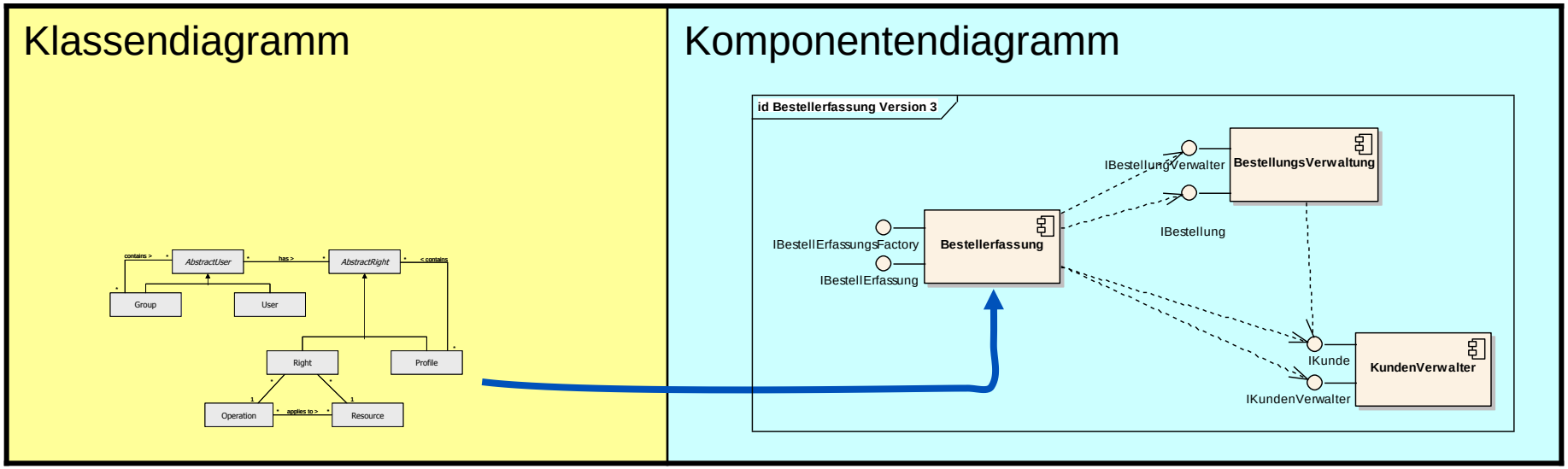
Kapitel 2: Klassen und Schnittstellen

- Schnittstellen
- Reflektion
- Innere Klassen

Martin Binder, FH Rosenheim, © 2015

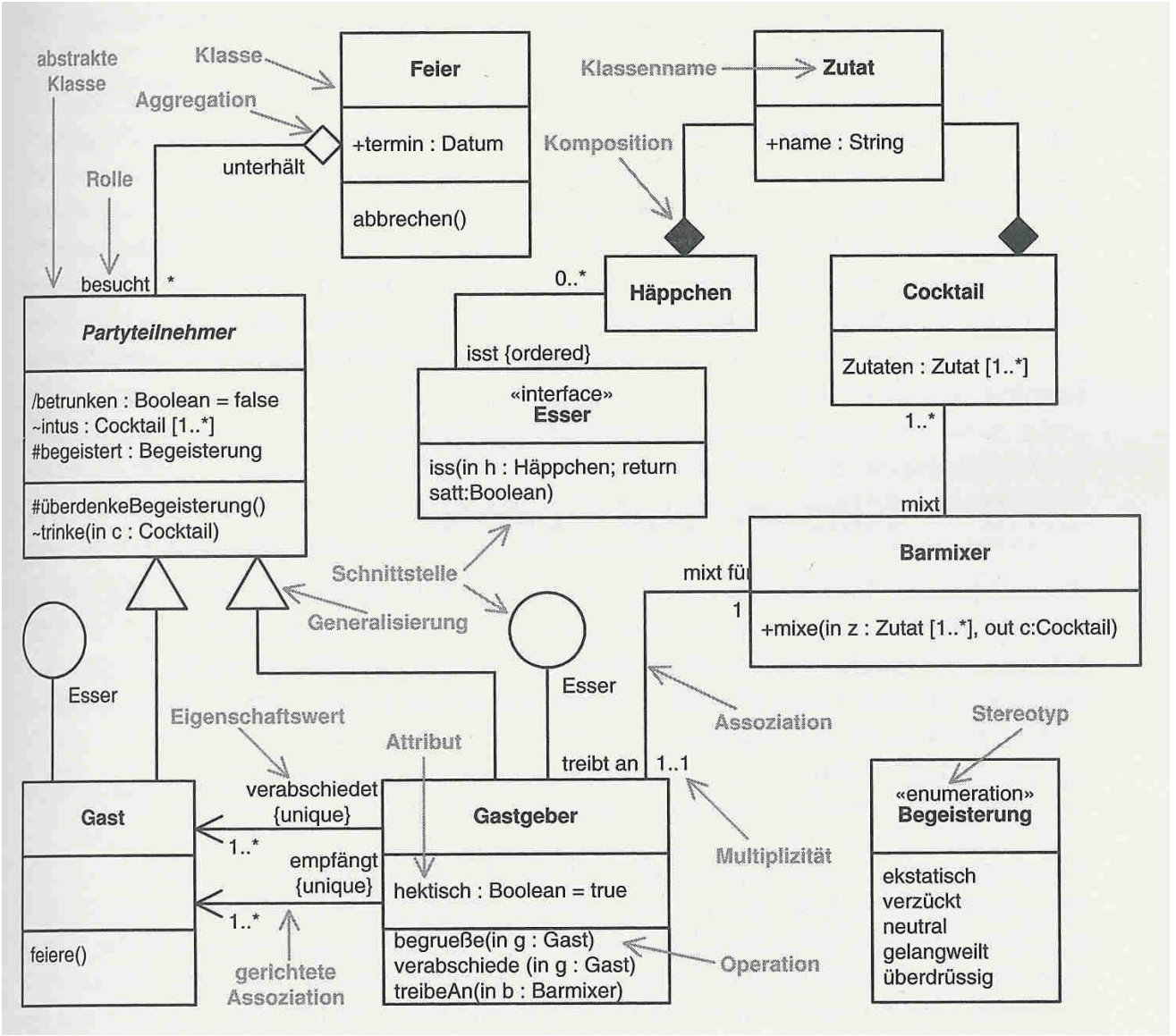
Motivation für Klassen und Schnittstellen

- Wiederholung von Konzepten von Programmieren 2
- Vertiefung der Konzepte von Programmieren 2 (Reflektion, verschachtelte Klassen)
- Bei guter Software in der Praxis werden zuerst die Schnittstellen entworfen, dann die Implementierung durchgeführt (Design by Contract)
- Ein fundiertes Verständnis von Klassen und Schnittstellen ist die Basis für die Entwicklung von Komponenten (s. Kapitel Komponenten und Schnittstellen)



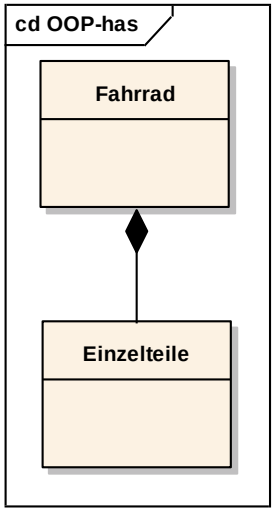
Kurzer Exkurs: Klassendiagramm in UML

- Elemente im Klassendiagramm
  - Klassen
  - Schnittstellen
  - Attribute
  - Operationen
  - Assoziationen
  - Generalisierung
  - Abhängigkeiten

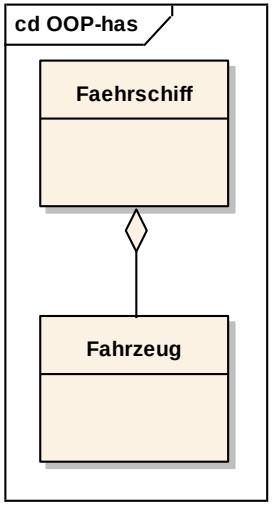


[UML glasklar, dpunkt.verlag]

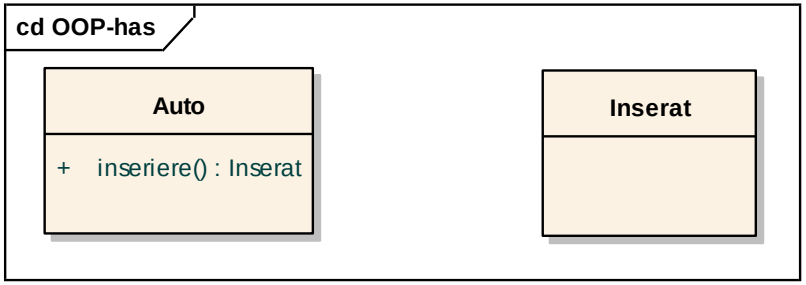
Klassendiagramm: Beziehungen zwischen Klassen und Objekten



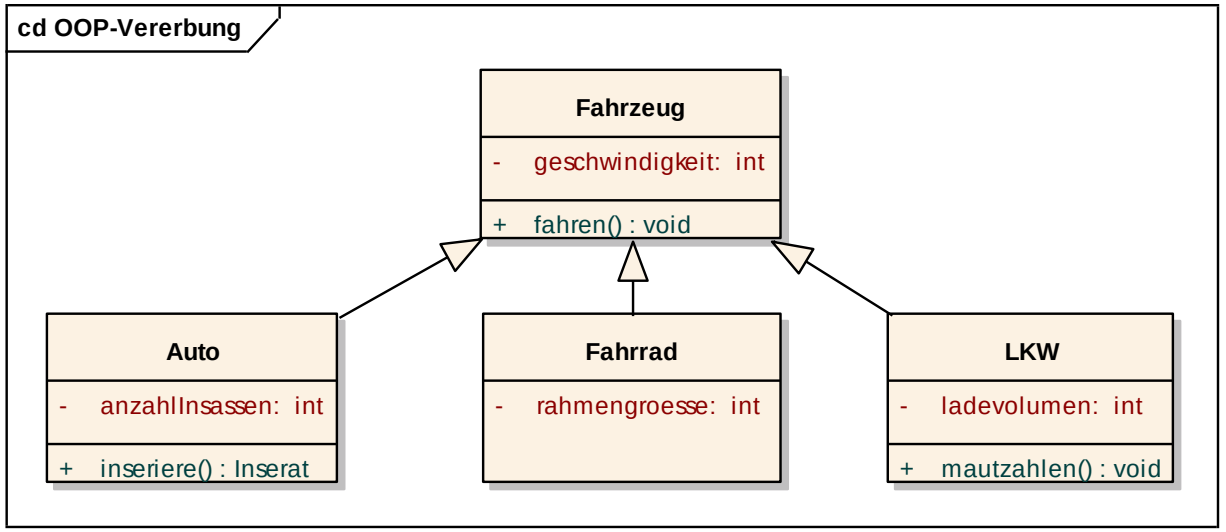
Komposition



Aggregation



Assoziation



Vererbung

Darstellung in **UML** (Unified Modelling Language)

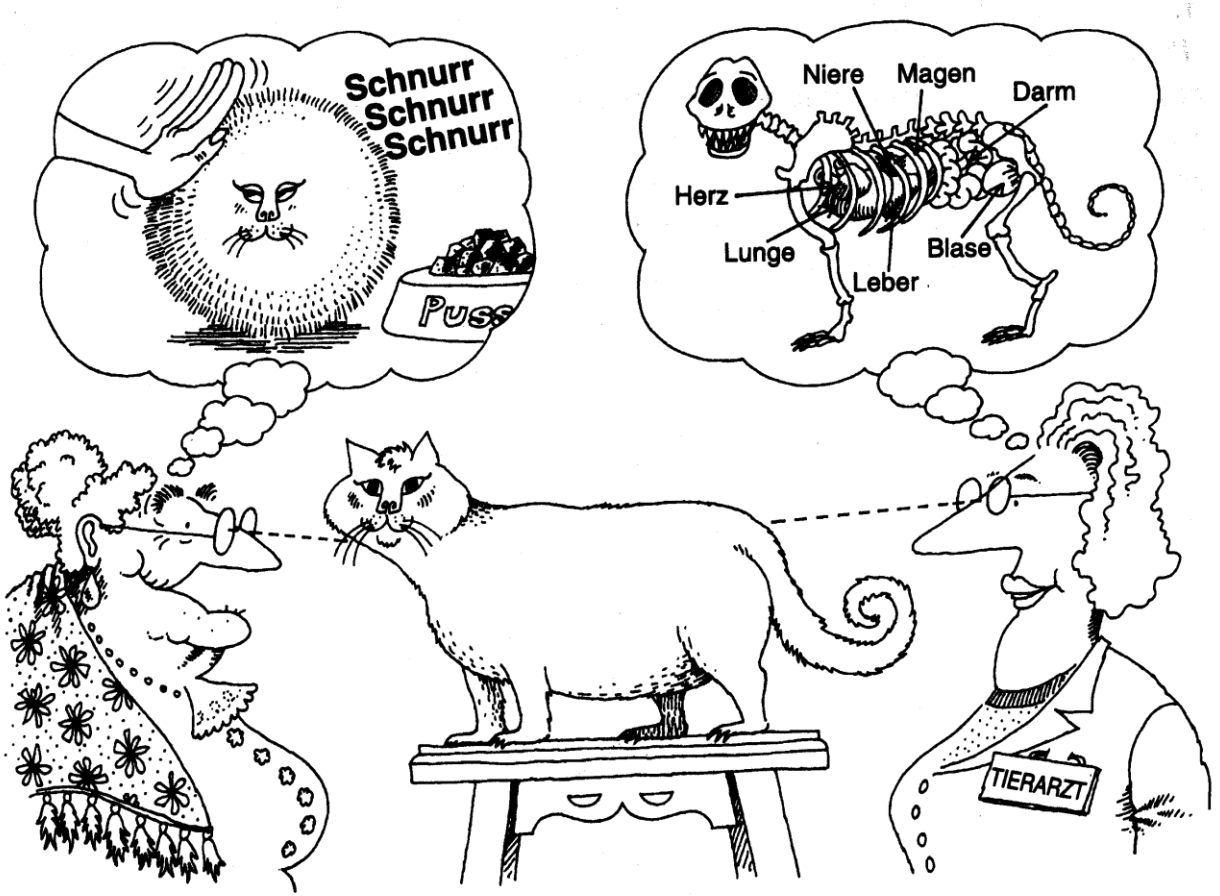
Programmieren 3

Kapitel 2: Klassen und Schnittstellen

- Schnittstellen
- Reflektion
- Innere Klassen

Martin Binder, FH Rosenheim, © 2015

Interfaces: Verschiedene Sichten und Konzepte



[Booch: "Objektorientierte Analyse und Design", Addison-Wesley, 1994]

Beispiel für Interface, Implementierung, Anwendung

```
// Stack-Interface
package de.fhro.inf.p3.util;

public interface IStack<T> {
    void push(T x);
    T pop ();
}
```

Schnittstelle

```
package de.fhro.inf.p3.util;

public class MyStack<T> implements IStack<T> {
   private int size;
   private Object[] elements;
   public MyStack() {
     elements = new Object[100];size = 0;
   }
   public void push(T x) {
      elements[size] = x; size += 1;
   }
   // ... Methode pop
}
```

Implementierung

```
// Anwendung des Stack-Interfaces
package any.package;
import de.fhro.inf.p3.util.IStack;

public class UseAStack {
  private IStack<Kunde> myStack;     // UseAStack kennt nur IStack

  public UseAStack(IStack<Kunde> stack){
      this.myStack = stack;   // keine Ahnung, was da kommt
      ...
  }
      ...
}
```

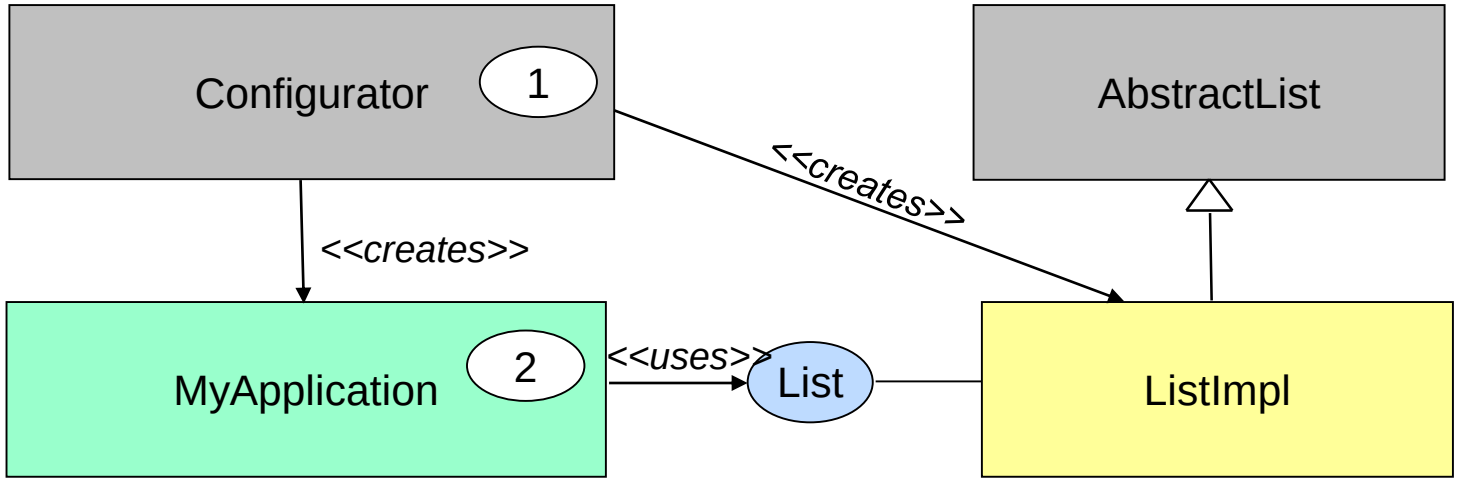
Anwendung

Vererbung von Interfaces

- Jedes Interface kann beliebig viele andere Interfaces erweitern.
- Semantik ist einfach: "Das kann ich alles!"

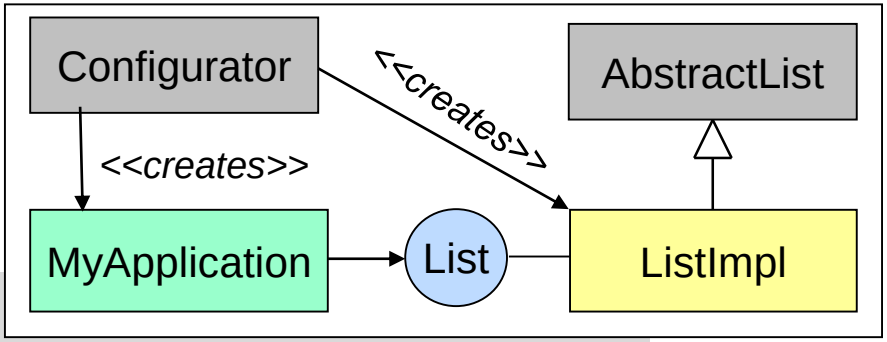
```
public interface ITransactionStack<T> extends IStack<T> {
   /**
    * macht alle push/pop-Operationen seit dem
    * letzten commit/rollback endgültig
    */
   void commit();

  /**
    * verwirft alle push/pop-Operationen seit dem
    * letzten commit/rollback
    */
   void rollback();
}
```

Gegen Schnittstellen programmieren



- Vorteile von Schnittstellen:
  - machen Software *leichter verständlich*, denn es genügt, die Schnittstelle zu betrachten.
  - ermöglichen einen erhöhten Abstraktionsgrad (Liste, Set, Map, Graph, …)
  - helfen *Abhängigkeiten* zu *reduzieren*, (von einer bestimmten Implementierung)
  - Schnittstellen erleichtern die Wiederverwendung von bewährten Implementierungen und sparen damit Arbeit
  - Viele Standardprobleme sind auf der Ebene von Schnittstellen bereits gelöst, z.B. in Form der Such- und und Sortieralgorithmen innerhalb des JDK

Wie kommt die Applikation zur Implementierung? Variante 1: Konfigurationsklasse



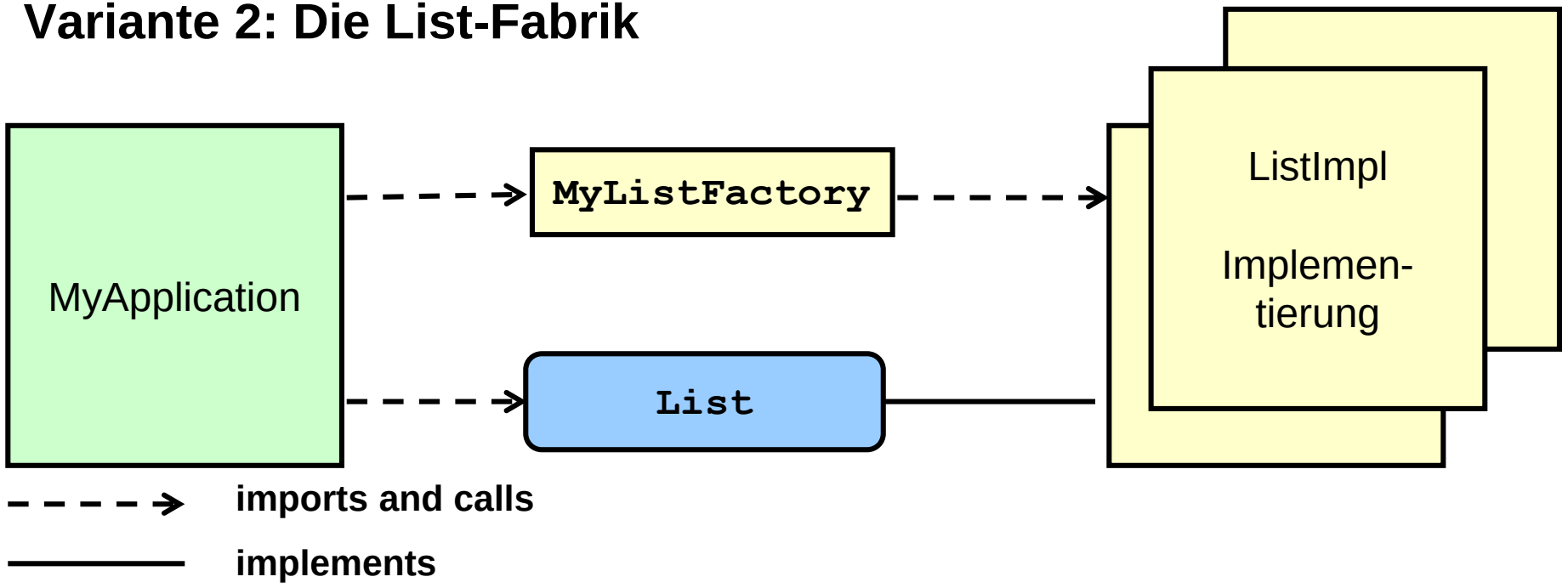

```
public class Configurator {
     ...
     public void tuwas() {
          List<Kunde> kundeList = new ListImpl<Kunde>();
          MyApplication appl = new MyApplication(kundeList);
          appl.start(..);
          …     }
}
```

1

```
public class MyApplication {
     private List<Kunde> myList;
     public MyApplication(List<Kunde> list) {
          myList = list;
     }
     public start() { ... }
}
```

2

Variante 2: Die List-Fabrik




```
import de.fhro.inf.p3.util.MyListFactory

public class MyApplication {
   private List<Kunde> myList;
      public MyApplication() {
         MyListFactory f =
                new MyListFactory();
         myList = f.makeList();
      }
      public start() { ... }
}
```

```
// Factory Pattern
package de.fhro.inf.p3.util;
import de.fhro.inf.p3.ListImpl;
import java.util.List;

public class MyListFactory {

   public <T> List<T> makeList () {
      return new ListImpl<T>();
   }
}
```

Abstrakte Klasse: Listen implementieren mit dem Vorlagen-Muster (1)

```
public abstract class AbstractList<E> implements List<E> {  // java.util
     ...
     public abstract int size();
     public abstract E get(int idx);
     ...
}
```

Gibt Teile der Implementierung vor, lässt aber Lücken

```
public class ArrayList<E> extends AbstractList<E> {   // java.util
     private Object[] elements;
     private int size = 0;
     public ArrayList() { ... }
     public int size() { return size; }
     public E get(int i) { return elements[i]; }
     ...
}
```

Füllt die Lücken (Abstrakte Methoden)

Listen implementieren mit dem Vorlagen-Muster (2)

```
public class Range extends AbstractList<Integer> {
     private int size;
     public Range(int size) { this.size = size; }
     public int size() { return size; }
     public Integer get(int idx) {
          if (0 <= idx && idx < size) {
               return new Integer(idx);
              //geht auch
              //return idx;
           }
          else {
               throw new IndexOutOfBoundsException();
          }
     }
}
```

Beispiel für ein Interface: Iterator

- Mach was mit allen Elementen eines Behälters: Iteratoren für Schleifen
- zeigen auf ein Element des Behälters, verwalten Schleifen, merken sich die aktuelle Position
- erzeugt über Methoden des Behälters: iterator( )
- Jede Collection hat ihre eigene Iterator-Implementierung
- Iterator: lesen vorwärts; ListIterator: beide Richtungen

```
public interface Iterator<T> {
   boolean hasNext();
   T next();              // Dereferenzierung: nächstes Element
   void remove();     // entferne aktuelles Element (optional)
}
```

```
...
Collection<Kunde> c = ...;              // keine Ahnung, was das ist
Iterator<Kunde> i = c.iterator();
while (i.hasNext()) {
   Kunde x = i.next();
   // tuwas mit x
}
...
```

Schnittstellen sind Abstraktionen: Beispiel Iterator

- Zugriff auf sequentielle Datenstrukturen
- Ergebnisse von Suchanfragen
- Folgen: natürliche Zahlen, Fibonacci-Zahlen, ...
- Reihen: exp, sin, cos, ...

```
public class Naturals implements Iterator<Integer> {
    private int next = 0;
    public boolean hasNext() { return true; }
    public Integer next() { return new Integer(next++); }
    public void remove() { throw new UnsupportedOperationException(); }
}
```

```
public class Exp implements Iterator<BigDecimal> {
    private int count = 1;
    private BigDecimal next = BigDecimal.ONE;
    public boolean hasNext() { return true; }
    public BigDecimal next() {
       next = next.divide(count++);
       return next;
    }
    public void remove() { throw new UnsupportedOperationException(); }
}
```

Transaktionen als Schnittstelle

```
public interface ITransaction {
    /**
     * bestaetigt eine Transaktion
     */
    void commit();


    /**
     * setzt die Welt auf den Zustand zu Beginn der laufenden
     * Transaktion zurueck.
     */
    void rollback();
}
```

Eine transaktionsfähige Liste

```
public class MyTransactionalList<E> extends ArrayList<E>
                                    implements ITransaction {

      private List<E> beforeImage = new ArrayList<E>();

       public MyTransactionalList(List<E> xs) {
           super(xs);
          beforeImage.addAll(this);
      }

      public void commit() {
           beforeImage.clear();
           beforeImage.addAll(this);
      }

      public void rollback() {
         this.clear();
         this.addAll(beforeImage);
      }
```

instanceof, Liskovsches Substitutionspinzip

x sei ein Java-Objekt ungleich null, C eine Klasse, I ein Interface

a)	**null instanceof C** **null instanceof I**	ist false für jede Klasse C und jedes Interface I
b)	**x instanceof C**	ist genau dann richtig, wenn x Objekt der Klasse C ist oder wenn x Objekt einer Unterklasse von C ist
c)	**x instanceof Object**	ist richtig für jedes Objekt x ungleich null
d)	**x instanceof I**	ist genau dann richtig, wenn x Objekt einer Klasse ist, die I implementiert

Liskovsches Substitutionsprinzip

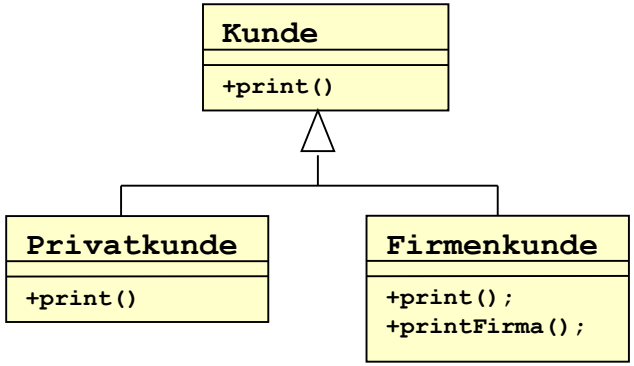
y sei eine Variable vom Typ C bzw. I. Dann ist die Zuweisung

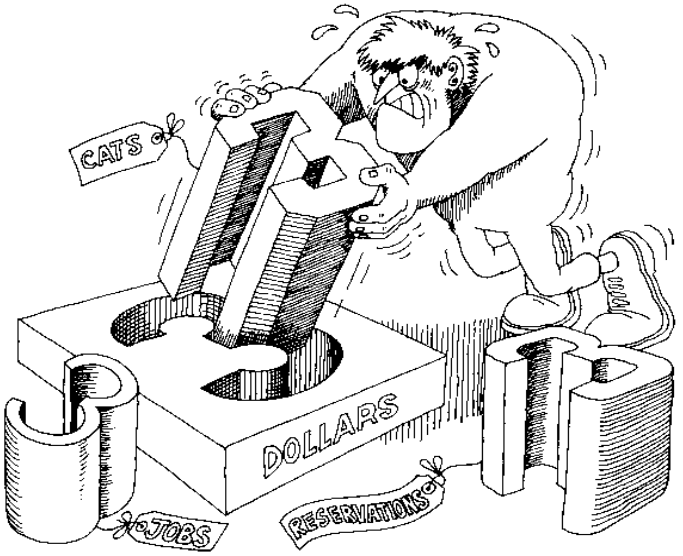
y = x // y wird ersetzt durch x; y darf nicht spezieller sein als x

genau dann korrekt, wenn gilt

x instanceof C bzw. **x instanceof I**

Liskovsches Substitutionsprinzip





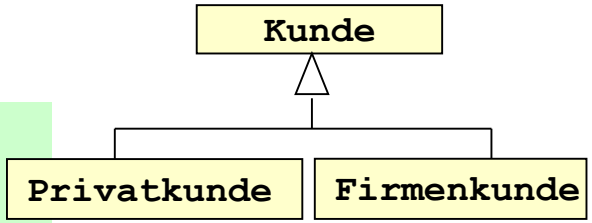

```
Kunde y = …;
FirmenKunde x = new FirmenKunde( );
y = x; // Liskov
```

Liskovsches ***Substitutions-Prinzip*** (LSP):

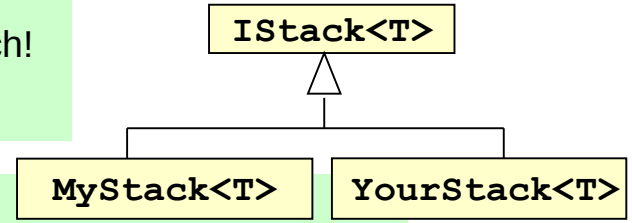
Let $\phi(x)$ be a property provable about objects *x* of type T. Then $\phi(y)$ should be true for objects *y* of type S where S is a ***subtype*** of T (Barbara Liskov, 1994)

Klassen sollen in jedem Fall durch alle ihre Unterklassen ersetzbar sein. (So wird meistens auch programmiert)

instanceof, Downcast, Upcast




```
Kunde k;
Firmenkunde fk = new Firmenkunde();
Privatkunde pk = new Privatkunde();
k = fk;                 // jeder Firmenkunde ist ein Kunde: Upcast automatisch!
fk = (Firmenkunde) k;   // Downcast, aber nicht umgekehrt
k = pk;                 // jeder Privatkunde ist ein Kunde: Upcast automatisch!
pk = (Privatkunde) k;   // Downcast, aber nicht umgekehrt
```




```
IStack<?> s;
IStack<?> ms  = new MyStack<…>();      // jeder MyStack ist ein IStack: Upcast automatisch!
IStack<?> ys  = new YourStack<…>();    // jeder YourStack ist ein IStack: Upcast automatisch!
ms instanceof IStack;                  // true
ms instanceof MyStack;                 // true
ms instanceof YourStack;               // false
```

Programmieren 3

Kapitel 2: Klassen und Schnittstellen

- Schnittstellen
- Reflektion
- Innere Klassen

Martin Binder, FH Rosenheim, © 2015

Reflektion (Introspection)

- Durch die Abstraktion mit Interfaces und Vererbung ist nicht immer klar um welche Klasse es sich konkret handelt.
- **Reflektion** ist ein Mechanismus mit dem man Klassen und Objekte zur Laufzeit untersuchen und im begrenztem Umfang modifizieren kann.
- Bietet Möglichkeit, Klassen zu laden und zu instanziieren, auf Methoden und Attribute zuzugreifen, ohne dass bereits zur Compile-Zeit ihr Name bekannt sein musste
- Anwendung:
  - Meta-Programme, die auf den Klassen und Objekten anderer Programme operieren
  - Beispiele: Hilfsprogramme zum Debuggen, Klassen-Browser, Serialisierung von Objekten
- Zu beachten ist, dass Zugriffe auf Methoden und Member-Variablen deutlich langsamer als bei direktem Zugriff ausgeführt werden

Die Klasse `java.lang.Class` (1)

- repräsentiert eine Java-Klasse oder ein Interface
- ein Class-Objekt pro Klasse in der JVM.
- Dynamisches Laden einer Java-Klasse zur Laufzeit:
  - man kennt den Klassennamen als String
  - lädt die Klasse in die JVM mit

diesen Namen kann man zur Laufzeit einlesen

```
String className = "de.fhro.inf.p3.util.MyStack";
try {
   Class c = Class.forName(className);
} catch (ClassNotFoundException e) { .. }
```

- und erzeugt dann ein Objekt dieser Klasse mit

IStack f = (IStack) c.newInstance();

Die Klasse `java.lang.Class` (2)

- Die Klasse ***`java.lang.Class`*** ist der Schlüssel zur Funktionalität des Reflection-APIs.

- Es gibt drei Möglichkeiten, um an das Class-Objekt heranzukommen

```
Class<Date>   c1 = java.util.Date.class;
Class<Date>   c2 = new java.util.Date().getClass();
Class<Object> c3 = Class.forName( "java.util.Date" );
```

- Die Methode **`newInstance()`** von Class kann zur Laufzeit neue Objekte der Klasse erzeugen.

```
IStack<String> source = new Stack1<String>();    // abstrakter Stack mit
source.push("xyz");                              // erster Stack-Implementierung
...
```

```
Class<IStack> stackClass = source.getClass(); // Class feststellen
IStack destination       = stackClass.newInstance(); //Achtung, Typ von
                                                     //IStack geht verloren!
                                                // neue Instanz von Stack1 anlegen
destination.push(source.pop());                 // Element kopieren
```

Generische Klasse `java.lang.Class<T>`

- Ab Java 5 ist `java.lang.Class` generisch, also `java.lang.Class<T>`
- Dabei ist `T` der Typ, den das Klassenobjekt repräsentiert
  - Beispiel: der Typ von `Date.class` ist `Class<Date>`, das vorherige Beispiel kann also typsicher als `Class<Date> c1 = java.util.Date.class;` geschrieben werden.
- Reflection-Code kann somit (teilweise) typsicher programmiert werden
- Dies ist weiterhin ein Beispiel für den Einsatz von Typ-Parametern außerhalb des Collection Frameworks.
- Komplexeres Einsatz Beispiel siehe http://download.oracle.com/javase/tutorial/extra/generics/literals.html

Methoden der Klasse `java.lang.Class`

- *boolean isInterface()* Liefert true, falls das Class-Objekt eine Schnittstelle beschreibt.
  boolean isArray() Liefert true, falls das Class-Objekt einen Array-Typ beschreibt.
  boolean isPrimitive() Testet, ob das Class-Objekt einen primitiven Datentyp beschreibt.

- *String getName()*
  Liefert für ein Class-Exemplar als String den voll qualifizierten Namen der repräsentierten Klasse oder Schnittstelle beziehungsweise des repräsentierten Array-Typs oder primitiven Datentyps.

- *int getModifiers()* Liefert die Modifizierer für eine Klasse oder eine Schnittstelle.

- *Field[ ] getFields()* Liefert ein Array mit Field-Objekten

- *Method[ ] getMethods()*

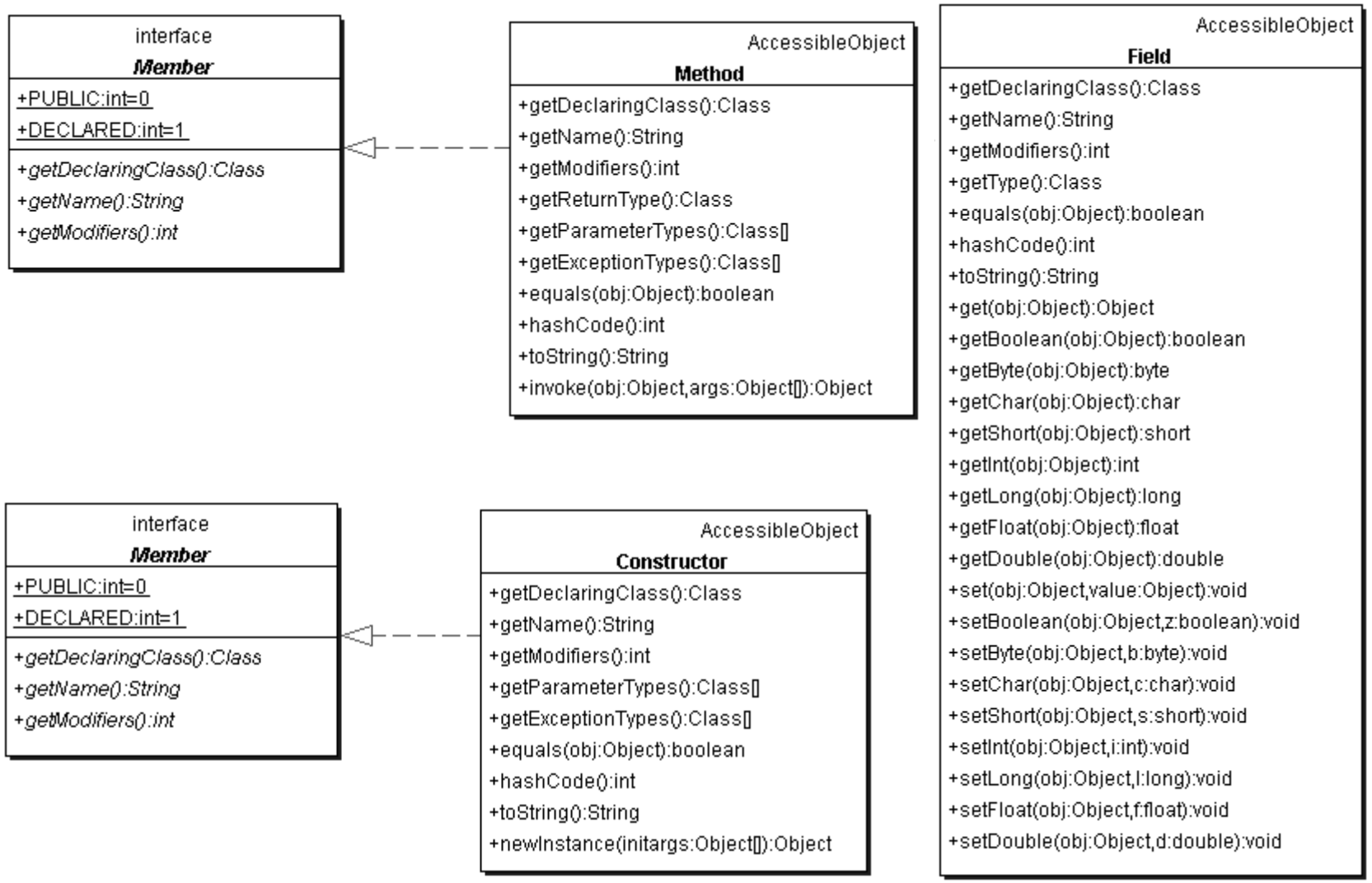
  Gibt ein Array von Method-Objekten zurück, die alle öffentlichen Methoden der Klasse/Schnittstelle beschreiben. Geerbte Methoden werden mit in die Liste übernommen

  Object invoke(Object obj, Object[ ] args)

  Die Methode ***invoke*** der Klasse ***Method*** dient dazu, die durch dieses Methodenobjekt repräsentierte Methode tatsächlich aufzurufen. (obj: Objekt auf dem die Methode ausgeführt werden soll, args: aktuellen Parameter an die Methode)

- *Constructor[ ] getConstructors()* Liefert ein Feld mit Constructor-Objekten.

Die Klassen `Method, Fields, Constructor`



Beispiel: generische toString()-Methode

```
public class ToStringHelper {
    public static String toString(Object o) {
          List<String> stringList = getFieldsFromObject(o, o.getClass());
          return o.getClass().getName().concat(stringList.toString());
    }

    private static List<String> getFieldsFromObject(Object o, Class clazz) {
          ArrayList<String> list = new ArrayList<String>();
          Field f[] = clazz.getDeclaredFields();
          AccessibleObject.setAccessible(f, true); // Zugriffsrechte lockern
          for (int i = 0; i < f.length; i++) {
              try {
                      list.add(f[i].getName() + "=" + f[i].get(o));
              } catch (IllegalAccessException e) { }
          }
              if (clazz.getSuperclass().getSuperclass() != null)
                  //Rekursiver Aufruf für Oberklassen
                   list.addAll(getFieldsFromObject, clazz.getSuperclass());
              }
        }
```

Programmieren 3

Kapitel 2: Klassen und Schnittstellen

- Schnittstellen
- Reflektion
- Innere Klassen

Martin Binder, FH Rosenheim, © 2015

Geschachtelte/Innere Klassen

- Ziele
  - Strukturierung von Typen
  - Verbindung von logisch aufeinander bezogene Objekte (Referenz)
  - Vertrauensbeziehung zwischen Typen (Zugriff auf Attribute)
  - Einzig dazu da um umgebender Klasse zu dienen
- Vier Arten
  - Statisch geschachtelte Klassen (hier nicht weiter betrachtet)
  - Innere Klassen, Attributklassen
  - Lokale innere Klassen
  - Anonyme innere Klassen
- Verwendung z.B. bei AWT, JavaBeans, Collections

Innere Klasse (Mitgliedsklasse, Elementklasse, member class)

- Objekt der inneren Klasse ist mit einer Instanz der umgebenden Klasse verbunden
- Objekte der Äußeren Klasse können mit keinem oder mehreren Objekten der inneren Klasse verbunden sein
- Zugriff auf alle Attribute und Methoden der äußeren Klasse
- Anwendung z.B. bei Implementierung von Interface Map (z.B. MyIterator, MySet, , MyCollection)

```
public class MySet<E> extends AbstractSet<E>  {
   ...   // Implementierung der Klasse

   public Iterator<E> iterator() {
        return new MyIterator<E>();
   }
   private class MyIterator<E> implements Iterator<E> {
        ...  //Implementierung des Iterators
   }
}
```

Lokale innere Klasse

- Definition in einem Code-Block (z.B. Methodenrumpf)
- Lokale innere Klassen
  - sind keine Attribute der Klasse
  - sind wie lokale Variablen, nur lokale Zugriffsrechte
  - Zugriff auf Variablen der umgebenden Methode nur falls diese mit Hilfe des Schlüsselworts final als konstant deklariert wurden

```
public static <E> Iterator<E> walkThrough(final E[] objs) {
    class Iter<T> implements Iterator<T> {
          ...  //Implementierung des Iterators
               // mit Zugriff auf final Parameter, z.B. objs
               // und final lokale Variablen der umgebenden Methode
    }

    return new Iter<E>( );
}
```

Anonyme innere Klasse

- Werden zu einem Zeitpunkt definiert und instanziiert (im New-Ausdruck)
- Besitzen keine expliziten Konstruktoren
- Deklarieren keine neuen Methoden
- Nur für kleine Klassen geeignet, sonst Code unlesbar

```
public static <E> Iterator<E> walkThrough(final E[] objs) {
    return new Iterator<E> () {
          ...  //Implementierung des Iterators
               // hasNext(), next(),remove()
               // mit Zugriff auf final Parameter, z.B. objs
               // und final lokale Variablen der umgebenden Methode
    };
}
```

```
Arrays.sort(args, new Comparator<String>() {
    public int compare(String o1, String o2) {
          return (o1.length() - o2.length());
    }
});
```

Beispiel: JButton, JFrame, ActionListener

```
public interface ActionListener {
        void actionPerformed(ActionEvent e);
}
```

```
public class MyFrame extends JFrame {
    Container c = getContentPane();
    JButton confirm = new JButton("confirm");
    confirm.addActionListener( ??? );
    c.add(confirm);

    JButton cancel = new JButton("cancel");
    cancel.addActionListener( ??? );
    c.add(cancel):
    ...
}
```

Wie bekommt der Button seinen Listener?

ActionListener: Eigene Klassen

```
class ConfirmListener implements ActionListener  {
    private MyFrame frame;
    ConfirmListener(MyFrame frame) { this.frame = frame; }
    void actionPerformed(ActionEvent e) {
         // tuwas unter Verwendung von frame
    }
}
```

```
class CancelListener implements ActionListener  {
    private MyFrame frame;
    CancelListener(MyFrame frame) { this.frame = frame; }
    void actionPerformed(ActionEvent e) {
         // tuwas unter Verwendung von frame }
}
```

ActionListener: Innere Klassen

```
public class MyFrame extends JFrame  {
    ...
    private class ConfirmListener implements ActionListener {
         public void actionPerformed(ActionEvent e) {
         // tuwas unter Verwendung von frame
    }
    private class CancelListener implements ActionListener {
         public void actionPerformed(ActionEvent e) {
         // tuwas unter Verwendung von frame
    }
    ...
    confirm.addActionListener(new ConfirmListener());
    cancel.addActionListener(new CancelListener());
}
```

ActionListener: Anonyme Klassen

```
public class MyFrame extends JFrame  {
    ...
    confirm.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                  // tuwas unter Verwendung von frame
          }
    } );
    cancel.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                  // tuwas unter Verwendung von frame
          }
    } );
    ...
}
```

Regeln für geschachtelte Klassen (aus Bloch)

- Geschachtelte Klassen zur Strukturierung des Quellcodes verwenden!
- Innere Klasse falls
  - Jede Instanz der inneren Klasse Referenz auf umgebende Instanz haben muss
- Lokale Klasse falls
  - Klasse in eine Methode gehört
- Anonyme Klasse falls
  - Instanzen nur von einem Ort erzeugt werden müssen
  - Und ein Typ der die Klasse charakterisiert (Interface oder abstrakte Basisklasse) bereits vorhanden ist

Zusammenfassung Klassen und Schnittstellen

- Fabriken und Schnittstellen trennen die Anwendung von der Implementierung und haben Vorteile bei der Entwicklung (Dummy-Implementierung) und Wartung (Austausch der Implementierung)
- Bei der Verwendung von Schnittstellen ist mindestens an einer Stelle die konkrete Implementierung zu konfigurieren
- Man sollte immer gegen die allgemeinste Schnittstelle programmieren, die den Zweck erfüllt
- Mittels Reflektion kann man stereotypen Code generisch erzeugen
- Reflektion ermöglich auch zur Laufzeit dynamische Analyse von Klassen und Objekte, sie ist aber nicht sehr performant
- Innere Klassen machen den Code kürzer und reduzieren die Menge an Paketen und Dateien, sie sind aber manchmal schwer zu lesen