Informatik II

Übungsstunde 6

simon.mayer@inf.ethz.ch
Distributed Systems Group, ETH Zürich

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Statisches & Dynamisches Type Checking
- Zeit zum Programmieren...
  und fuer noch mehr Fragen

(Zum Teil) Neue Begriffe...

- Inheritance/Vererbung:
  - Klassen koennen Methoden und Eigenschaften von anderen Klassen uebernehmen, indem sie deklarieren, dass sie in einem is-a Verhaeltnis zu diesen stehen

```
public class Vehicle {
        double speed;
}

public class Car extends Vehicle {
        // speed ist hier verwendbar!
}
```

(Zum Teil) Neue Begriffe...

- Interfaces/Schnittstellen
  - Klassen koennen explizit bekanntgeben, dass sie eine bestimmte Funktionalitaet (= bestimmte Methoden) implementieren. So koennen vom Benutzer verschiedene Klassen, die dieselbe Funktionalitaet besitzen, austauschbar verwendet werden.

```
public interface Comparable {
	public boolean smallerThan(Comparable rhs);
}

public class Triangle implements Comparable {
	public boolean smallerThan(Triangle rhs) {
		return (this.area() < rhs.area())
	}
}
```

(Zum Teil) Neue Begriffe...

- Polymorphie/Dynamische Bindung
  - Der Typ einer Variable kann sich zur Laufzeit aendern (analog zur Vererbungs/Interfacehierarchie)

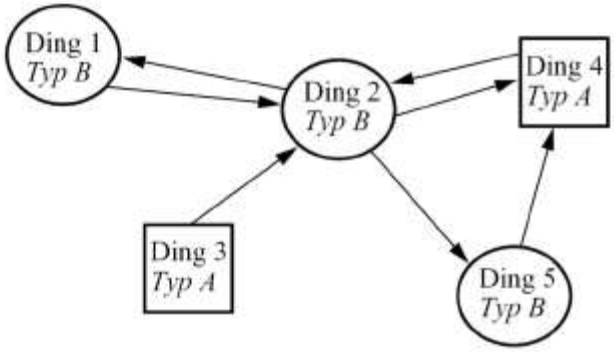
```
public class Vehicle {
        double speed;
}

public class Car extends Vehicle {
        // speed ist hier verwendbar!
}

public static int (String [] args) {
        Vehicle vehicle = new Car();
        Car vehicleAsCar = (Car) vehicle;
}
```

Objektorientiertes Programmieren

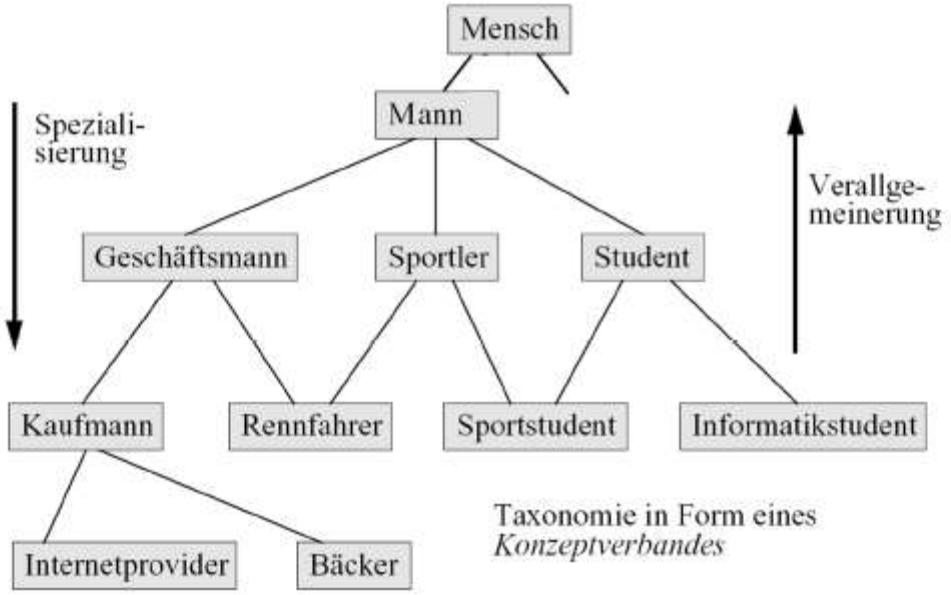
- **Weltsicht:** Die Welt besteht aus verschiedenen interagierenden „Dingen", welche sich klassifizieren lassen



Simulationssprache SIMULA war die erste objektorientierte Programmiersprache (1967)

- **Ziel:** Betrachteten Weltausschnitt strukturkonform mit kommunizierenden Objekten abbilden und modellieren

Konzepthierarchie

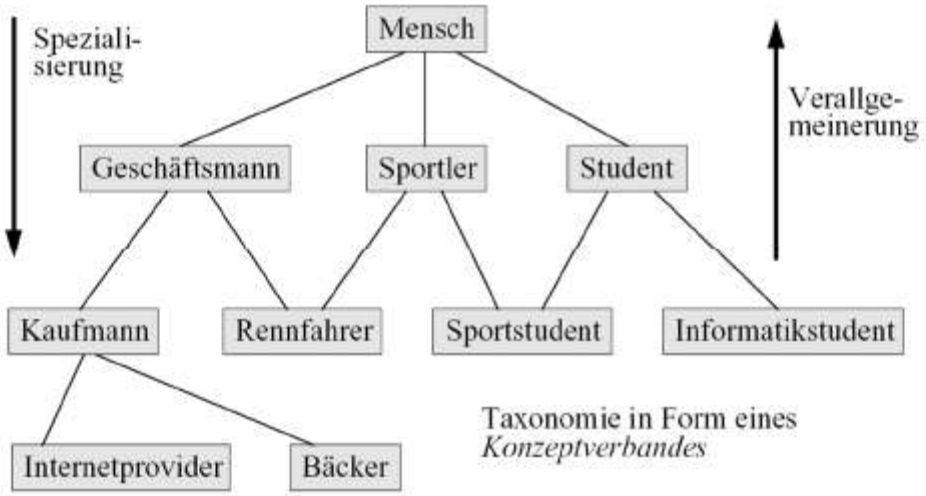


Taxonomie in Form eines *Konzeptverbandes*

- Spezialisierung, Verallgemeinerung
  - Ein Rennfahrer ist ein spezieller Sportler
  - Ein Informatikstudent ist ein Mensch (mit Programmierkenntnissen)

Vererbung („inheritance") (2)

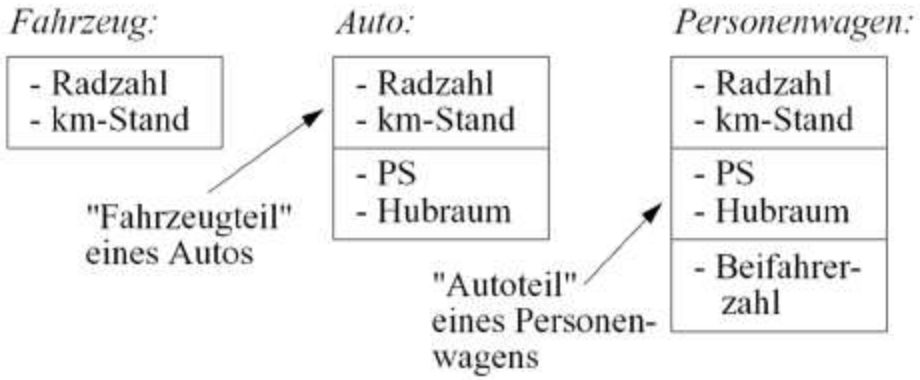
- Ein Objekt (als „Begriffsinstanz") ist oft polymorph
  - Ein Rennfahrer kann je nach Zweckmässigkeit auch entweder nur als Sportler oder als Geschäftsmann oder schlicht einfach als Mensch betrachtet werden



Taxonomie in Form eines *Konzeptverbandes*

Vererbungsrelation ist transitiv: ein Bäcker hat alle Eigenschaften eines („generischen") Menschen

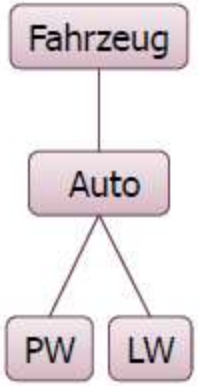
Abgeleitete Klassen, Redefinition (2)



- Eine Methode „Berechne_Steuer" lässt sich nicht für alle Fahrzeuge gleichermassen definieren
  - Man würde z.B. in „Auto" eine Standardmethode vorsehen (Benutzung von „Hubraum"), jedoch für spezielle Fahrzeuge (z.B. Elektroautos) diese Methode anders definieren

Zuweisungskompatibilität

- Objekte von abgeleiteten Klassen können an Variablen vom Typ der Basisklasse zugewiesen werden
  - `Fahrzeug f; Auto a; ... f = a;`
  - Variable f kann Fahrzeugobjekte speichern
  - Ein Auto ist ein Fahrzeug
  - Daher kann f auch Autoobjekte speichern



- Die Umkehrung gilt jedoch nicht!
  - `a = f;` ist verboten!
  - Variable a kann Autoobjekte speichern
  - Ein Fahrzeug ist aber kein Auto!

- „Gleichnis" zur Zuweisungskompatibilität: Auf einem Parkplatz für Fahrzeuge dürfen Autos, Personenwagen, Fahrräder... abgestellt werden. Auf einem Parkplatz für Fahrräder jedoch keine beliebigen Fahrzeuge!

Zuweisungskompatibilität (2)

- Merke also: *Eine Variable vom Typ „Basisklasse" darf auch ein Objekt der abgeleiteten Klasse enthalten*
- Man nennt diese Eigenschaft auch Polymorphie, da eine Referenz auf Objekte *verschiedenen Typs* zeigen kann (bzw. eine Variable Werte unterschiedlichen Typs haben kann)
- *Beispiel:* Eine Variable vom Typ „Referenz auf Fahrzeug" kann zur Laufzeit sowohl zeitweise auf PW-Objekte, als auch zeitweise auf LW-Objekte zeigen

Typkonversion („type cast")

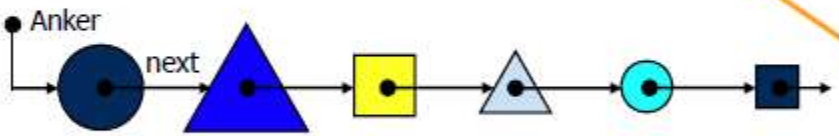
- Aus gutem Grund ist f.Hubraum verboten: Auf f könnte ja zufällig ein Fahrrad (ohne Hubraum!) „parken"!
- Durch Typkonversion kommt man aber notfalls auch über f an den Hubraum des Auto-Objektes:
  `System.out.println(((Auto)f).Hubraum);`
- Aber wenn dort derzeit kein Auto (sondern ein Fahrrad) parkt? Das gibt einen Laufzeitfehler „ClassCastException"!
- Dem kann man wie folgt vorbeugen:

```
if (f instanceof Auto)
   System.out.println(((Auto)f).Hubraum);
else System.out.println("kein Auto, kein Hubraum!");
```

Abstrakte Methoden und Klassen

- Abstrakte Methoden werden von den abgeleiteten Klassen jeweils spezifisch implementiert
- Eine Klasse mit abstrakten Methoden muss selbst als „abstract" deklariert werden
- Beispiel:

```
abstract class GeomObj {
  GeomObj next;
  public abstract double flaechenwert();}
}
```



Diese Methode muss für jede abgeleitete Klasse mit sinnvoller Semantik versehen werden!

Verarbeitung polymorpher Objekte

- Die Verarbeitung der einzelnen Objekte kann unabhängig von ihrem eigentlichen Typ geschehen
  - In der Praxis wird eine solche „Verarbeitung" i.Allg. wesentlich komplexer sein, als in obigem Beispiel angedeutet
- Es können, *ohne den Verarbeitungsalgorithmus anzupassen,* neue Objekttypen als Unterklassen von „GeomObj" eingeführt werden (z.B. „Quadrat")
  - Nur *hinzufügen*, aber (im Idealfall) *nichts verändern,* vermindert den Wartungsaufwand beträchtlich!
- Late binding: Es wird zur Laufzeit berechnet, welche konkrete Methode „angesprungen" wird; dies steht zur Übersetzungszeit (→ „early binding") noch nicht fest!

Generische Methoden und abstrakte Klassen

```
abstract class Sort {
  abstract boolean kleiner (Sort y);
  static void sort(Sort[] Tab) {
    for (int i=0; i<Tab.length; i++)
      for (int j=i+1; j<Tab.length; j++)
        if (Tab[i].kleiner(Tab[j])) {
          Sort swap = Tab[i];
          Tab[i] = Tab[j];
          Tab[j] = swap;
        }
  }
}
```

Achtung: Es wird absteigend sortiert!

Dieses einfache Sortierverfahren („deletion sort") ist ziemlich ineffizient!

- Wir fordern, dass die zu sortierenden Objekte vom Typ einer von Sort abgeleiteten Klasse sind
- In der abgeleiteten Klasse muss ausserdem die Methode „kleiner" (als totale Ordnungsrelation auf den Objekten) realisiert werden

Anwendung des Sortierverfahrens

- Es sollen zunächst einfache int-Werte sortiert werden, und zwar unter Benutzung der existierenden generischen Sortierroutine (ohne diese kennen zu müssen!):

```
class IntSort extends Sort {
  int w;
  IntSort(int i) {
    w = i;
  }
  boolean kleiner(Sort y) {
    return w < ((IntSort)y.w);
  }
}
```

Konstruktor

Hier wird die Relation "kleiner" definiert und implementiert

Dies ist die vom Anwender bereitgestellte Klasse (aufbauend auf der Basisklasse Sort)

Java Inheritance

- Nur Single Inheritance moeglich: Eine Klasse in Java kann nur einen Vorfahren haben!

- Beispiel:

  Eine Person, die studiert und ein Instrument spielt, kann **nicht** von `Student` und `Musician` erben!

  Dadurch Vermeidung des Diamantenproblems...

Java Interfaces

- Schnittstellen: Spezialtyp von abstrakten Klassen
- Enthaelt nur Methodensignaturen ohne Implementierung und Konstanten!
- Eine Klasse kann mehrere Interfaces implementieren!
  - Klasse kann z.B. Sort- und Serializable-Interface implementieren
    → Kann dann sortiert und serialisiert werden...

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Statisches & Dynamisches Type Checking
- Zeit zum Programmieren...
  und fuer noch mehr Fragen

Übung 6

1. Schnittstellen, Klassen, Typumwandlungen

- Static Casts: Bereits beim Kompilieren kann Gueligkeit des Casts festgestellt werden.

```
Vehicle vehicle = new Vehicle();
Bicycle bike = vehicle;
```

Laeuft nicht! Vehicle ist kein Bike!

Übung 6

1. Schnittstellen, Klassen, Typumwandlungen

- Dynamic Casts: Gueltigkeit kann nicht beim Kompilieren festgestellt werden.

```
Bicycle bike = (Bicycle) vehicle;
```

«Ich will den Cast explizit durchfuehren!»
Vielleicht wurde vehicle so gebaut:

```
Vehicle vehicle = new Bicycle();
```

Übung 6

2. Schnittstellen und Implementierungen
   - Vorgegebene Schnittstellen fuer Codegeruest...
   - Ihr sollt es passend zu diesen aufbauen!

   - Factory Methode: Gibt echtes Stack Objekt als iStack Interface zurueck

   - Ausserdem: Ein Test fuer `empty()` soll implementiert werden
     - Testen, dass leer leer ist
     - Testen, dass nichtleer nichtleer ist

Übung 6

3. Polymorphie: Sortieren per Comparable Interface
   - Implementieren von GenericList
   - Implementieren von `area()` fuer beide geometrische Objekte
   - Implementieren der Methode `smallerThan()` aus dem Interface `Comparable` in GeometricObject
     - Benutzen von `area()`...
   - Implementieren von `sort()`

Übung 6

4. Effizienter dynamisch wachsender Stack (freiwillig)
   - Dynamisch wachsender Stack, aber kein Array, keine Itemliste...
   - Besser: Liste aus Arrays «ChunkedStack»
     - Zu wenig Platz: Neues Array bauen, anhaengen an Liste
     - Letztes Array leer: Letztes Array loeschen und aus Liste entfernen

Ablauf

- Besprechung der Vorlesung
- Uebungsbezogene Themen:
  Statisches & Dynamisches Type Checking
- Zeit zum Programmieren...
  und fuer noch mehr Fragen

Projekt: Inheritance

- VehicleList
  - Inheritance
  - Interfaces
  - Dynamic Typing

Informatik II

Übungsstunde 6

simon.mayer@inf.ethz.ch
Distributed Systems Group, ETH Zürich