Informatik: Abstrakte Klassen

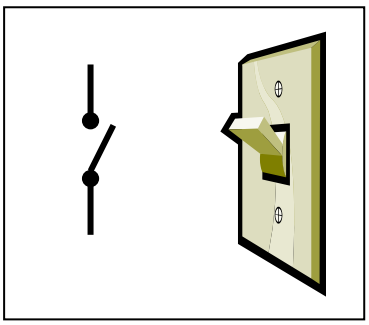
Ziele

- Die Grundideen objektorientierter Programmierung verstehen.
- Mit abstrakten Methoden und abstrakten Klassen arbeiten können.
- Zu einem Problem die Klassenhierarchie erstellen können.
- Die Methoden den Klassen zuordnen können.

Grundideen Objektorientierter Programmierung

Durch OOP soll Programm robuster, fehlerärmer und besser wart bar werden.

1. Grundidee: Abstraktion

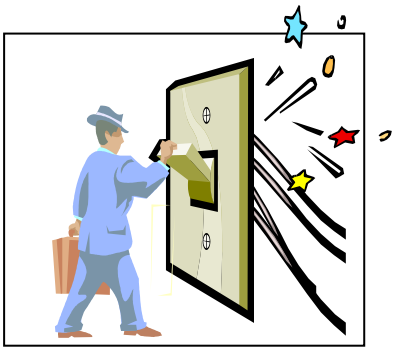


d. h. Trennung zwischen Konzept und Umsetzung, zw. technischem Handbuch und konkreter Apparatur zw. Funktionsweise eines Lichtschalters und konkretem Schalter

OOP zw. Klasse und Objekt
Objekte einer Klasse können sich untereinander unterscheiden

Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität eines Problems.

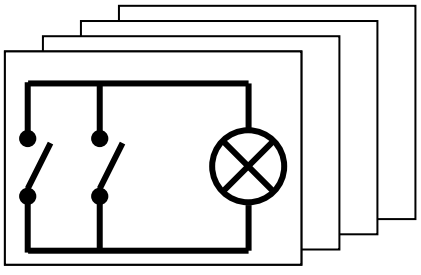
2. Grundidee: Kapselung



d. h. Zusammenfassung von Methoden und Variablen zu Klassen, Veröffentlichung der Kommunikationsmöglichkeiten mit Objekt

Kapselung hilft, die Komplexität der Bedienung eines Objektes zu verringern (klare Schnittstelle) und vermeidet undefinierte Interaktionen.

3. Grundidee: Wiederverwendung



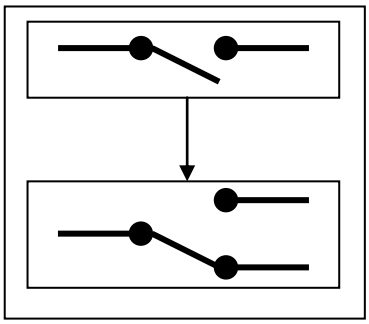
d. h. durch Abstraktion und Kapselung wird Wiederverwendung gefördert

Wiederverwendung hilft, Effizienz und Fehlerfreiheit beim Programmieren zu erhöhen.

4. Grundidee: Beziehungen

d. h. Objekte und Klassen existieren für gewöhnlich nicht alleine, sondern stehen in Beziehung zueinander.
Fahrrad ähnelt Motorrad, hat aber auch mit Auto Gemeinsames, Ein-/Ausschalter ähnelt Wechselschalter, ebenfalls der Kreuzschalter

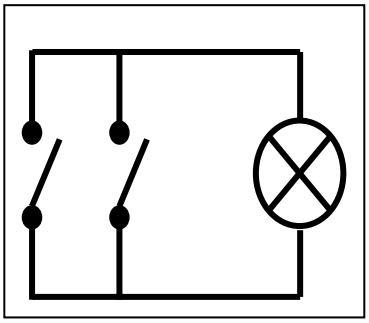
„is-a"-Beziehung



B is a A bedeutet dass B alle Eigenschaften von A besitzt und vermutlich noch einige mehr. B ist Spezialisierung von A, A ist Generalisierung von B.

Bei OOP durch Vererbung realisiert.

„part-of"-Beziehung



beschreibt aus welchen Objekten ein Objekt zusammengesetzt ist.

Bei OOP durch Instanz- bzw. Membervariablen realisiert, welche Objekte aufnehmen können.

Verwendungs- oder Aufrufbeziehung

Benutzt beispielsweise eine Methode während ihrer Ausführung ein temporäres Objekt, so besteht zw. beiden eine *Verwendungsbeziehung*:
Objekt x verwendet eine Instanz der Klasse Y um Operationen auszuführen.

```
class Mensch
{
  void machtLicht(Schalter s);
  ...
}
sepp.machtLicht(gangSchalter);
```

Taucht in Argumentliste einer Methode eine Objektvariable der Klasse T auf, so entsteht eine *Aufrufbeziehung*, denn das Objekt muss um eigene Operationen durchführen zu können, die Klasse T und wenigsten einige ihrer Methoden kennen.

5. Grundidee: Polymorphismus

bedeutet zweierlei...

Punkt 1:
Eine Objektvariable kann Objekte unterschiedlicher Klassen aufnehmen. Dabei gilt die Regel dass nur Objekte einer bestimmten Klasse bzw. einer daraus abgeleiteten Klasse angelegt werden können.

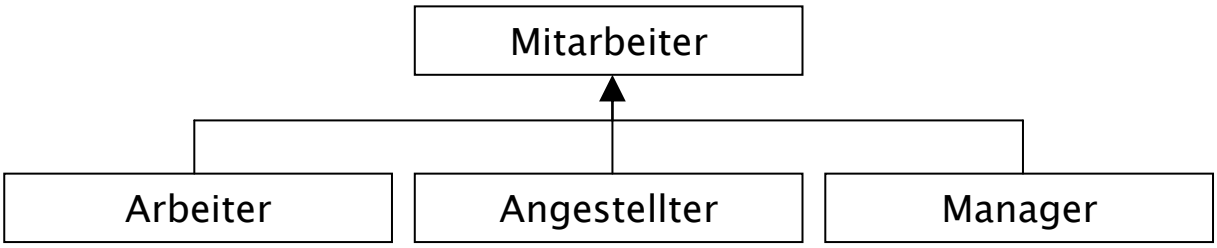
```
Schalter s1, s2;
s1 = new EinAusSchalter();
s2 = new WechselSchalter();

s1.betaetigen();
s2.betaetigen();
```

Voraussetzung: Methode `betaetigen` bereits in Klasse Schalter deklariert!!!

Punkt 2:
Überlagern bzw. Redefinieren von Methoden in Kombination mit Punkt 1 ermöglicht, dass zur Laufzeit in Abhängigkeit zum tatsächlich angelegten Objekt die passende Methode aufgerufen wird.

Abstrakte Methoden und abstrakte Klassen




```
public class GehaltsBerechnung
{
  private static final int ANZAHL_MA = 3;
  private static Mitarbeiter[] mitarbeiter;

  public static void main(String[] args) {
    mitarbeiter = new Mitarbeiter[ANZAHL_MA];
    mitarbeiter[0] = new Arbeiter("Sepp", 8, 160);
    mitarbeiter[1] = new Angestellter("Franz", 980, 300);
    mitarbeiter[2] = new Manager("Rudi", 2000, 1200);

    double gehaltsKosten = 0;
    for (int i = 0; i < mitarbeiter.length; ++i) {
      gehaltsKosten+=mitarbeiter[i].getMonatsbrutto();
      System.out.println(i + " " + mitarbeiter[i].toString());
    }
    System.out.println("Gehaltskosten: " + gehaltsKosten);
  }
}
```

Die Klasse `Mitarbeiter` muss die Methode `getMonatsbrutto()` enthalten, ansonsten könnte nicht `mitarbeiter[i].getMonatsbrutto()` aufgerufen werden.

In der Klasse `Mitarbeiter` kann diese Methode aber noch nicht implementiert werden!!!

```
public abstract class Mitarbeiter
  extends Object
{
  private String name;
  public Mitarbeiter(String name) {
    this.name = name;
  }
  public abstract double getMonatsbrutto();
  public String toString() {
    return name;
  }
}
```

Abstrakte Methoden enthalten nur die Deklaration, nicht aber die Implementierung.

Eine Klasse, welche abstrakte Methoden enthält

- muss selbst als **abstract** deklariert werden und
- kann nicht instanziert werden.

```
public class Arbeiter
  extends Mitarbeiter
{
  private double stundenLohn;
  private double anzahlStunden;
  public Arbeiter(String name,
    double stundenLohn, double anzahlStunden) {
    super(name);
    this.stundenLohn = stundenLohn;
    this.anzahlStunden = anzahlStunden;
  }
  public double getMonatsbrutto() {
    return stundenLohn * anzahlStunden;
  }
  public String toString () {
    return super.toString() + " " + stundenLohn
      + " " + anzahlStunden + " " +
      getMonatsbrutto();
  }
}
```

```
public class Manager extends Mitarbeiter
{
  private double fixGehalt;
  private double provision;
```

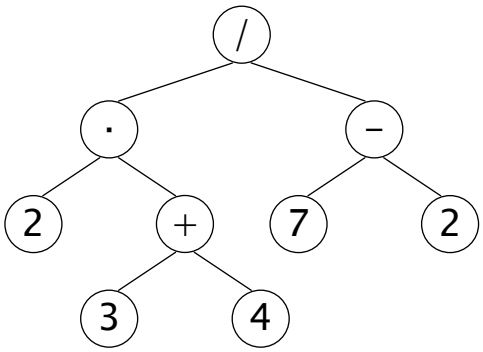
```
public class Angestellter
  extends Mitarbeiter
{
  private double grundGehalt;
  private double zulage;

  …

  public double getMonatsbrutto() {
    return grundGehalt + zulage;
  }
}
```

Beispiel: Mathematische Ausdrücke

$$\frac{2 \cdot (3+4)}{7-2}$$

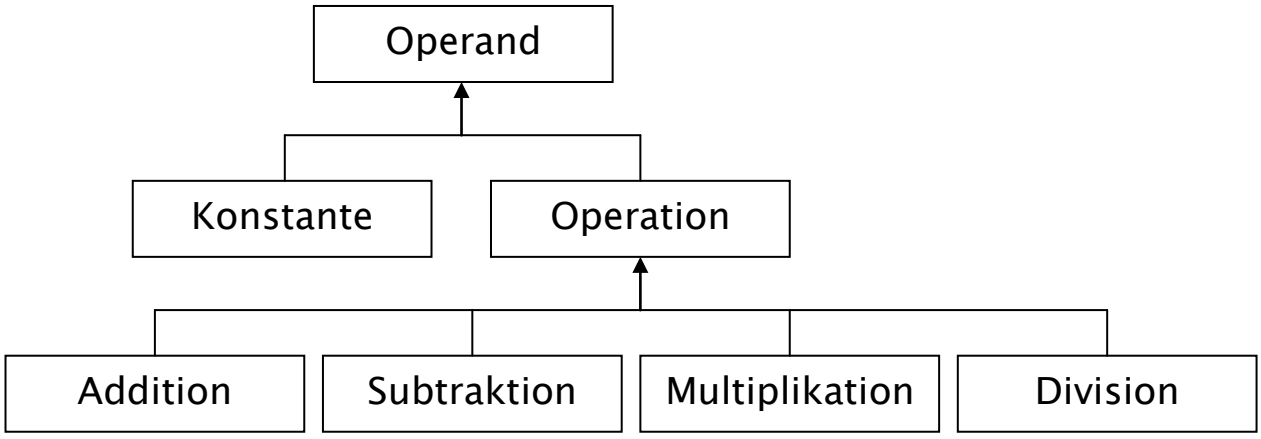


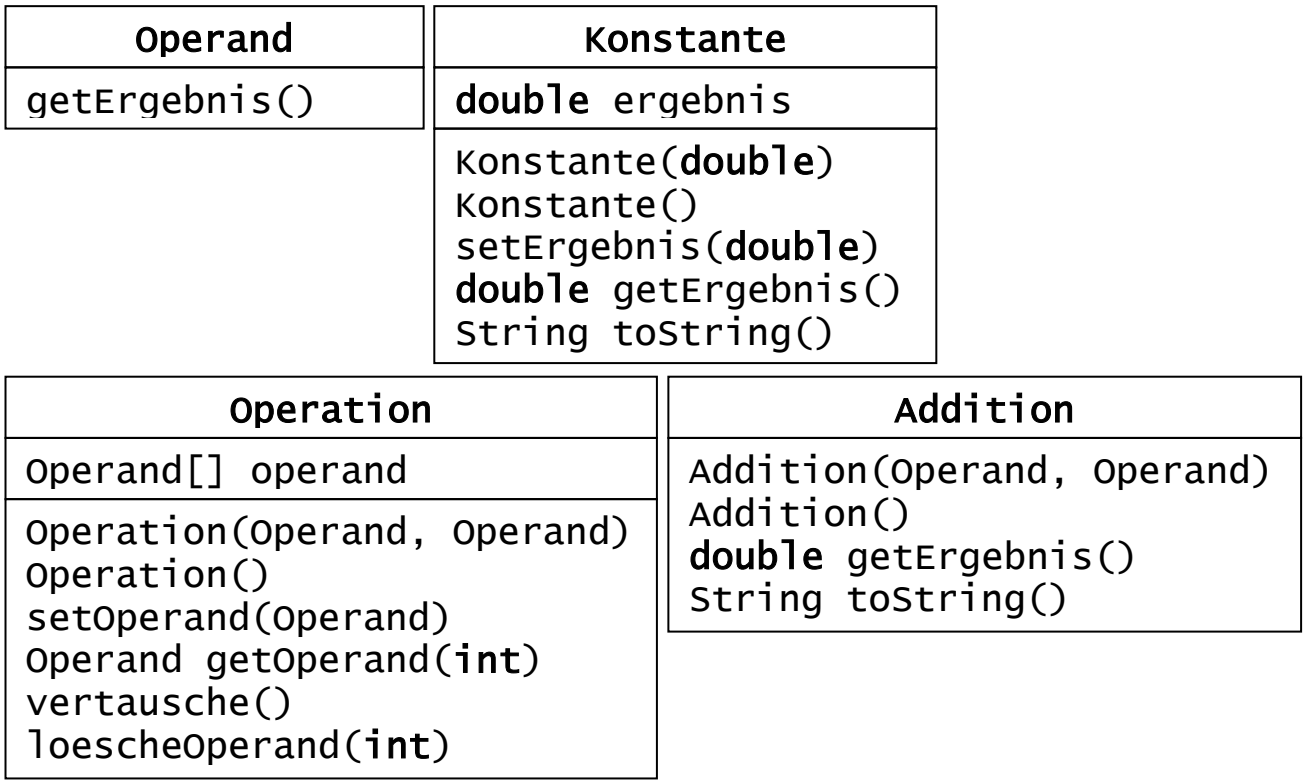
Konstante z. B. 2, 3, 4

Operand kann Konstante oder Operation sein

Operation besteht aus zwei Operanden, hat Ergebnis
kann Addition, Subtraktion, Multiplikation oder Division sein

Konstanten, Operanden und Operationen liefern ein Ergebnis.






```
public abstract class Operand
{
   public abstract double getErgebnis();
}


public class Konstante extends Operand
{
   private double ergebnis = 0.0;
   public Konstante(double ergebnis) {
      this.ergebnis = ergebnis;
   }
   public Konstante() {
      super();
   }
   public void setErgebnis(double ergebnis) {
      this.ergebnis = ergebnis;
   }
   public double getErgebnis() {
      return this.ergebnis;
   }
   public String toString() {
      return String.valueOf(this.ergebnis);
   }
}
```

```
public abstract class Operation extends Operand
{
   private Operand[] operand = new Operand[2];

   public Operation(Operand operand0, Operand operand1) {
      this.setOperand(operand0);
      this.setOperand(operand1);
   }

   public Operation() {
      super();
   }

   public void setOperand(Operand operand) {
      if (this.operand[0] == null)
         this.operand[0] = operand;
      else
         if (this.operand[1] == null)
            this.operand[1] = operand;
   }

   public Operand getOperand(int position) {
      if (position >= 0 && position <= 1)
         return this.operand[position];
      else
         return null;
   }

   public void vertausche() {
      Operand operand = this.operand[0];
      this.operand[0] = this.operand[1];
      this.operand[1] = operand;
   }

   public void loescheOperand(int position) {
      if (position == 0) {
         this.operand[0] = this.operand[1];
         this.operand[1] = null;
      } else
         if (position == 1)
            this.operand[1] = null;
   }
}
```

- Die Operanden werden mit setOperand der Reihe nach in die Operation gehängt.
- Beim Löschen werden die Operanden unter Umständen nach vorne verschoben.

```
public class Addition extends Operation
{
   public Addition(Operand operand0, Operand operand1) {
      super(operand0, operand1);
   }
   public Addition() {
      super();
   }
   public double getErgebnis() {
      double ret = 0.0;
      if (this.getOperand(0) != null)
         ret = this.getOperand(0).getErgebnis();
      if (this.getOperand(1) != null)
         ret = ret + this.getOperand(1).getErgebnis();
      return ret;
   }
   public String toString() {
      String ret = null;
      …
      return ret;
   }
}
```

Anlegen der Objekte und Auswertung:

```
Operation o =
   new Division(
      new Multiplikation(
         new Konstante(2)
      ,
         new Addition(
            new Konstante(3.0)
         ,
            new Konstante(4.0)
         )
      )
   ,
      new Subtraktion(
         new Konstante(7.0)
      ,
         new Konstante(2.0)
      )
   );
System.out.println(o.toString());
```