TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Vererbung

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstraktionen auf Klassenebene

Abstraktion bezeichnet die Verallgemeinerung des konkreten Einzelfalls, wobei generelle Strukturmerkmale betont werden.
In der objekt-orientierten Software-Entwicklung:

- Extraktion der gemeinsamen Eigenschaften unterschiedlicher Objekte / Klasse
- Vermeiden von Duplikation (weniger Code → weniger Arbeit)

Beispiele:

- Personen in der Universität (Professoren, Studierende, Verwaltungsangestellte, ...)
- Komponenten von Fenstersystemen (Menues, Schaltflächen, Textfelder, ...)
- Ein-/Ausgabeschnittstellen (Dateien, Netze, ...)

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Vorgehen

1. Auffinden von Programmfragmenten (Attribute und Methoden) mit ähnlicher Bedeutung / Struktur
2. Erarbeiten einer abstrakteren Klasse, die die gemeinsamen Eigenschaften zusammenfasst und eine entsprechende (verkleinerte) Schnittstelle bereitstellt

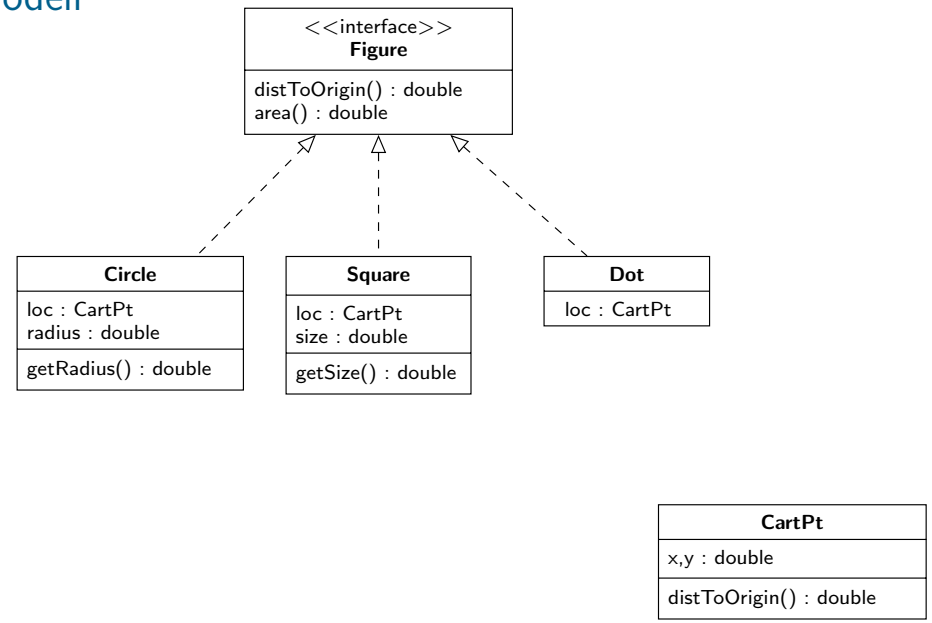
Fallstudie: Geometrische Figuren

In einem Zeichenprogramm sollen verschiedene geometrische Figuren in einem Koordinatensystem dargestellt werden (Einheit: ein Pixel).

Wir betrachten zunächst drei Arten von Figuren:

- Kreise mit dem Mittelpunkt als Referenzpunkt und gegebenem Radius
- Quadrate mit Referenzpunkt links oben und gegebener Seitenlänge
- Punkte, die nur durch den Referenzpunkt gegeben sind

Modell



Geometrische Figuren: Interfaces

```
// Interface fuer Geometrische Figuren
interface Figure {
  double distToOrigin();
  double area();
}
```


Geometrische Figuren: Referenzpunkt

```
// Punkt im zweidimensionalen kartesischen Koordinatensystem
class CartPt {
  private double x, y;

  CartPt(double x, double y) {
    this.x = x;
    this.y = y;
  }

  double distToOrigin() {
    return Math.sqrt(x * x + y * y);
  }

}
```

Geometrische Figur: Kreis

```
class Circle implements Figure {
  private CartPt loc;
  private double radius;

  Circle (CartPt loc, double radius) {
    this.loc = loc;
    this.radius = radius;
  }

  double area() {
    return radius * radius * Math.PI;
  }

  double distToOrigin() {
    return Math.max(loc.distToOrigin() - radius,0);
  }

  double getRadius() {
    return radius;
  }
}
```

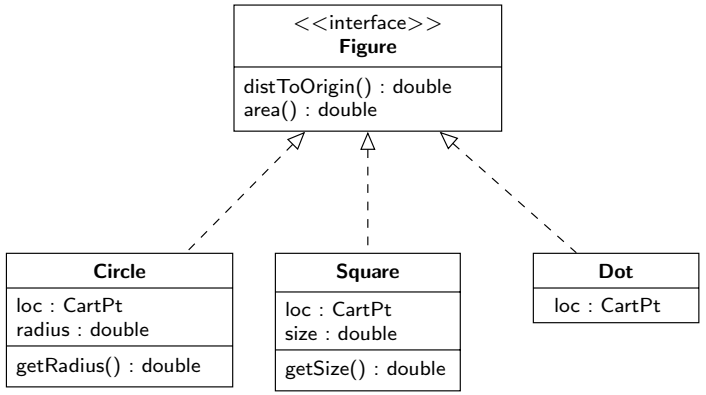

Geometrische Figur: Quadrat

```
class Square implements Figure {
  private CartPt loc;
  privat double size;

  Square (CartPt loc, double size) {
    this.loc = loc;
    this.size = size;
  }

  double area() {
    return size * size;
  }

  // Referenzpunkt ist linke untere Ecke
  double distToOrigin() {
    return loc.distToOrigin();
  }

  double getSize() {
    return size;
  }
}
```

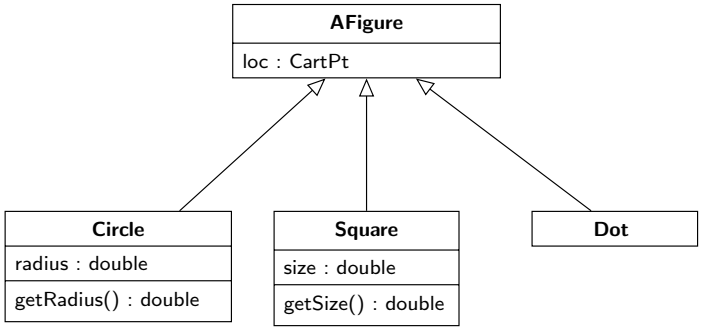

Geometrische Figur: Punkt

```
class Dot implements Figure {
  private CartPt loc;

  Dot(CartPt loc) {
    this.loc = loc;
  }

  double area() {
    return 0;
  }

  double distToOrigin() {
    return loc.distToOrigin();
  }
}
```


Ähnlichkeiten zwischen Klassen



Ähnlichkeiten zwischen Klassen



Begriffsklärung: Vererbung

Vererbung (engl. *inheritance*) im engeren Sinne bedeutet, dass eine Klasse Programmteile von einer anderen übernimmt.

Die erbende Klasse heißt **Subklasse**, die vererbende Klasse heißt **Superklasse**.

In Java sind die ererbten Programmteile Attribute, Methoden und geschachtelte Klassen, nicht vererbt werden Klassenattribute, Klassenmethoden und Konstruktoren.

Spezialisierung

Spezialisierung bedeutet das Hinzufügen speziellerer Eigenschaften zu einem Gegenstand oder das Verfeinern eines Begriffs durch Einführen weiterer Merkmale (z.B. berufliche Spezialisierung).

Vererbung unterstützt Spezialisierung durch:

- Hinzufügen von Attributen (**Zustandserweiterung**)
- Hinzufügen von Methoden (**Erweiterung der Funktionalität**)
- Anpassen, Erweitern bzw. Reimplementieren von Supertyp-Methoden (**Anpassen der Funktionalität**)

Vererbung von Attributen I

```
class AFigure {
  protected CartPt loc;
  AFigure(CartPt loc) {
    this.loc = loc;
  }
}
class Dot    extends AFigure { ... }
class Square extends AFigure { ... }
class Circle extends AFigure { ... }
```

- Konstruktor der `AFigure`-Klasse enthält und initialisiert nur das `loc`-Attribut
- `Square`-Objekte haben zusätzlich ein `size`-Attribut
- Wie werden die Felder initialisiert?

Vererbung von Attributen II

```
class Dot extends AFigure {
  Dot(CartPt loc) {
    super(loc);
  } ...
}

class Square extends AFigure {
  private double size;
  Square(CartPt loc, double size) {
    super(loc);
    this.size = size;
  } ...
}

class Circle extends AFigure {
  private double radius;
  Circle(CartPt loc, double radius) {
    super(loc);
    this.radius = radius;
  } ...
}
```


Vererbung von Attributen III

- Der Aufruf `super(...)` ruft den Konstruktor der Superklasse auf.
- Er muss **zu Beginn** des Konstruktors der Subklasse verwendet werden.
- Andernfalls wird zu Beginn der Defaultkonstruktor der Superklasse aufgerufen.
- Konstruktoren werden in Java nicht vererbt.

Vererbung von Methoden I

```
class Dot extends AFigure { ...
  double area()         { return 0; }
  double distToOrigin() { return loc.distToOrigin(); }
}

class Circle extends AFigure { ...
  private double radius;
  double area()         { return radius * radius * Math.PI; }
  double distToOrigin() { return loc.distToOrigin() - radius; }
}

class Square extends AFigure { ...
  private double size;
  double area()         { return size * size; }
  double distToOrigin() { return loc.distToOrigin(); }
}
```


Vererbung von Methoden II

- Methodendefinitionen können genauso vererbt werden wie Attribute
- Sinnvoll, wenn die Definition einer Methode für alle Subklassen identisch ist
- Problem: `distToOrigin()` ist in `Circle` anders definiert!

Überschreiben

In vielen Fällen ist es nötig, die Implementierung einer Methode der Superklasse in der Subklasse anzupassen, insbesondere um den erweiterten Zustand mit zu berücksichtigen.
Überschreiben (engl. overriding) einer ererbten Methode `m` der Superklasse bedeutet, dass man in der Subklasse eine neue, eigene Deklaration für `m` angibt. Die überschreibende Methode muss in Java die gleiche Signatur wie die überschriebene haben und mindestens so zugreifbar sein.
Die überschriebene Methode muss zugreifbar sein und kann durch `super` - Aufrufe benutzt werden:

```
super.<methodenName>( <AktParam1>,...)
```

Der aktuelle implizite Parameter eines `super`-Aufrufs ist der aktuelle implizite Parameter der aufrufenden Methode.

Beispiel: Überschreiben

```
class AFigure { ...
  double distToOrigin() {
    return loc.distToOrigin();
  }
}

// Klassen Dot und Square enthalten keine eigene Definition
    dieser Methode!!

class Circle { ...
  double distToOrigin() {
    return super.distToOrigin() - radius;
  }
}
```

- Der Aufruf von `loc.distToOrigin()` wäre die einzige Abhängigkeit von `CartPt` in der `Circle`-Klasse.
- Wir eliminieren diese Abhängigkeit durch Rückgriff auf die Implementierung der Superklasse.

Beispiel: Wetterdaten

In einem Programm zur Erfassung von Wetterdaten werden Messungen von Temperatur und Luftdruck verwaltet.

Für jeden Tag werden Minimal- und Maximalwerte der jeweiligen Messung gespeichert.

Temperature
high, low : double date : Date
getHigh() : double getLow() : double asString() : String

Pressure
high, low : double date : Date
getHigh() : double getLow() : double asString() : String

Aufgabe

Vermeiden Sie Code Duplikation durch Vererbung!

```
// Temperaturmessungen
// [in Celsius]
class Temperature {
  private Date date;
  private int high;
  private int low;

  Temperature (int high, int low,
    Date date) { ... }

  int getHigh() {
    return high;
  }

  int getLow() {
    return low;
  }

  String asString() {
    return date + " : " + low + "-"
     + high + "C");
  }
}
```

```
// Druckmessungen
// [in hPa]
class Pressure {
  private Date date;
  private int high;
  private int low;

  Pressure (int high, int low,
    Date date) { ... }

  int getHigh() {
    return high;
  }

  int getLow() {
    return low;
  }

  String asString() {
    return date + " : " + low + "-"
     + high + "hPA");
  }
}
```

Klasse `Recording`

```
// Messungen
class Recording {
    protected Date date;
    protected int high;
    protected int low;

    Recording (int high, int low, Date date) {
      this.high = high;
      this.low  = low;
      this.date = date;
    }
    int getHigh() {
      return high;
    }
    int getLow() {
      return low;
    }
    String asString() {
        return date + " : " + low + "-" + high);// ohne Einheit
    }
}
```


Klassen `Pressure` und `Recording`

```
class Pressure extends Recording {
  Pressure(int high, int low, Date date) {
    super(high, low, date);
  }

  String asString() {
    return super.asString() + "hPa";
  }
}

class Temperature extends Recording {
  Temperature(int high, int low, Date date) {
    super(high, low, date);
  }

  String asString() {
    return super.asString() + "C";
  }
}
```


Vererbung und Information Hiding

Durch die Vererbung gibt es nun zwei Arten, eine Klasse K zu nutzen:

- *Anwendungsnutzung*: Erzeugen und Verwenden der Objekt von K
- *Vererbungsnutzung*: Spezialisieren und Erben von K

Damit die erbende Klasse die geerbten Programmteile geeignet nutzen kann, benötigt sie meist einen intimeren Zugriff als ein Anwendungsnutzer.

Geschützter Zugriff

Viele Programmiersprachen bieten einen gesonderten Zugriffsbereich für Vererbung, der alle Subklassen einer Klasse umfasst.

Programmelemente, die als **geschützt** deklariert sind, d.h. mit dem Modifikator `protected`, sind in allen Subklassen zugreifbar.

Will man also Programmelemente, insbesondere Attribute, für Subklassen bereitstellen, müssen sie mindestens geschützten Zugriff gewähren.

Geschützter Zugriff ermöglicht allerdings erhebliches Verändern einer Klasse in Subklassen und birgt dementsprechend auch Gefahren, wie folgendes Beispiel zeigt.

Beispiel: Geschütztes Zugriffsrecht I

```
public class A_nicht_Null {
   protected int a = 1;
   public int getA() {
    return a;
   }
   protected void setA (int i) {
      if (i > 0) {
        a = i;
      }
   }
}

public class B {
    ...
   public void m(A_nicht_Null obj) {
      float f = 7 / obj.getA();
}  }
```


Beispiel: Geschütztes Zugriffsrecht II

Angesichts der Klasse `A_nicht_Null` kann die Anwendung davon ausgehen, dass die Instanzvariable `a` nie den Wert 0 annimmt.

Durch Vererbung können Subtyp-Objekte erzeugt werden, die sich ganz anders als die Objekte der Superklasse verhalten:

```
public class A_doch_Null extends A_nicht_Null {
   public int getA() {
    return -a;
   }
   public void setA (int i) {
    a = i;
  }
}
```


Beispiel: Geschütztes Zugriffsrecht III

```
public class Main {
   public static void main(String[] args) {
      A_doch_Null adn = new A_doch_Null();
      adn.setA(0) ;
      A_nicht_Null ann = adn;
      ... // hier koennte die Herkunft von ann
          // verschleiert sein
      new B().m(ann);
}  }
```

Um Szenarien wie im obigen Beispiel zu vermeiden, sollten Subklassen-Objekte das Verhalten der Superklassen-Objekte spezialisieren und sich ansonsten *konform* verhalten.

Zugreifbarkeit von vererbten Attributen

```
class C {
  public    int a = 0;
  protected int b = 1;
  private   int c = 2;
  int getC() {
    return c;
  }
}
```

```
class D extends C {
  int getB() {
    return b;
  }
}
```

```
public class Attributvererbung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut a: " + d.a);
    System.out.println("Attribut b: " + d.getB());
    System.out.println("Attribut c: " + d.getC());
  }
}
```

Verschattung von vererbten Attributen I

```
class C {
  public    int a = 0;
  public    int b = 2;
  private   int c = 3;
  int getC() {
    return c;
  }
}
```

```
class D extends C {
  public int e = 10;
  public int b = 12;
}
```

```
public class Zustandserweiterung {
  public static void main (String[] args) {
    D d = new D();
    System.out.println("Attribut e: " + d.e); // deklariertes e
    System.out.println("Attribut b: " + d.b); // deklariertes c

    System.out.println("Attribut a: " + d.a);       // ererbtes a
    System.out.println("Attribut b: " + ((C) d).b); // ererbtes b
    System.out.println("Attribut c: " + d.getC());  // ererbtes c
  }
}
```


Verschattung von vererbten Attributen II

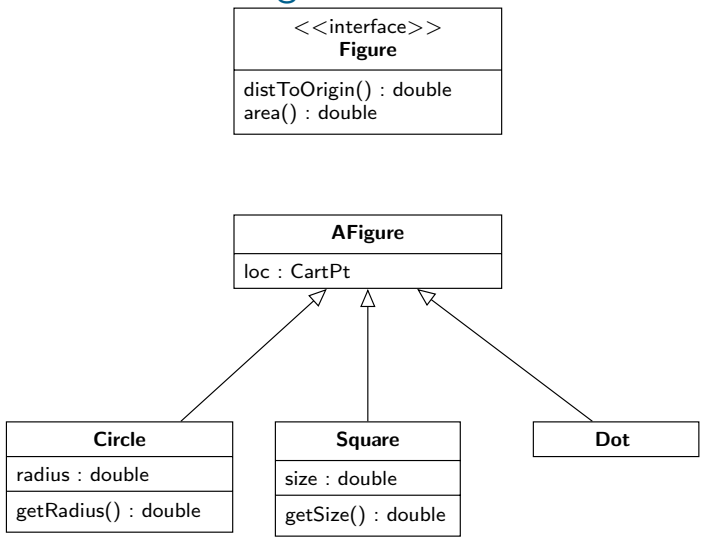
- Attribute können ererbte Attribute gleichen Namens verschatten. Dies sollte vermieden werden, kann aber nicht ausgeschlossen werden.
- Attribute werden statisch gebunden. Maßgebend ist also der (statische) Typ des selektierten Ausdrucks und nicht der Typ des Objekts, dass sich bei Auswertung des Ausdrucks ergibt.
- Verwenden Sie im Java-Visualizer die Option “Show overridden fields”, um alle (auch die verschatteten) Attribute zu visualisieren.

Beispiel: Geometrische Figuren I

Für geometrische Figuren haben wir bisher:

```
interface Figure {
  double area();
  double distToOrigin();
}
class AFigure {
  protected CartPt loc;
  AFigure (CartPt loc)   { ... }
  double distToOrigin() { ... }
}
class Dot extends AFigure {
  Dot (CartPt loc) { ... }
  double area()    { ... }
}
class Circle extends AFigure {
  Circle (CartPt loc, double radius) { ... }
  double area()                      { ... }
  double distToOrigin()              { ... }
  double getRadius()                 { ... }
} ...
```


Beispiel: Geometrische Figuren II



Beispiel: Geometrische Figuren III

- Wenn `AFigure` das Interface `Figure` implementieren soll, dann müssen alle Methoden, die `Figure` verlangt, auch in `AFigure` definiert werden.
- *Problem:* Es ist unklar, wie z.B. `area()` allgemein definiert werden soll, da jede geometrische Figur diese Methode anders definiert.

Begriffsklärung: Abstrakte Methoden & Klassen

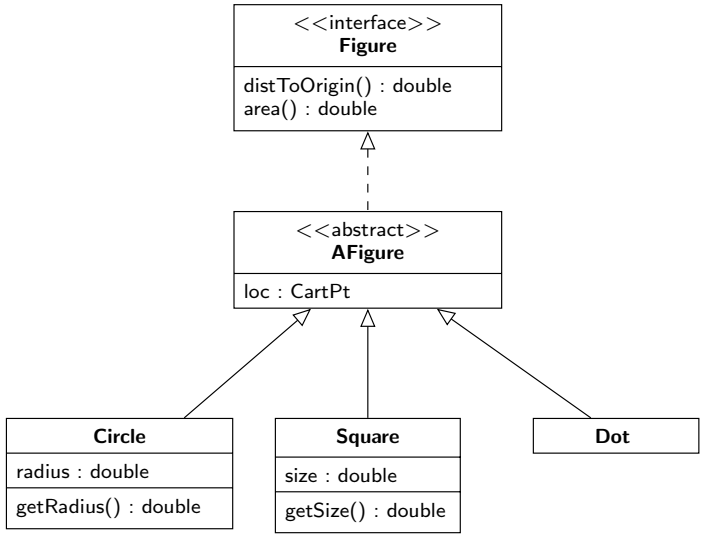
Eine Methode heißt **abstrakt**, wenn für sie kein Rumpf angegeben ist. Eine Klasse heißt **abstrakt**, wenn sie abstrakte Methoden besitzt oder als abstrakt deklariert ist (Modifikator `abstract`).

Es ist unzulässig, Instanzen abstrakter Klassen zu erzeugen.

Beispiel: Abstrakte `AFigure`-Klasse I

```
abstract class AFigure implements Figure {

  protected CartPt loc;

  AFigure (CartPt loc)   { ... }

  double distToOrigin()  { ... }
  abstract double area();
}
```


Modell



Spezialisierung durch die Subklasse

- In der zuvor gezeigten Version der Wetterdaten kann ein Programmierer direkt Objekte von `Recording` ableiten.
- Um dies zu verhindern, wandeln wir die Klasse `Recording` in eine abstrakte Klasse um.
- Ausserdem fügen wir eine abstrakte Methode hinzu, die eine String-Repräsentation der Messeinheit liefert.

Template-and-Hook Muster:
Das Template gibt die grundsätzliche Funktionalität vor und sieht Erweiterungspunkte (Hooks) vor, an denen Subklassen erforderliche Erweiterungen anbringen. Abstrakte Hook-Methoden zwingen die Subklassen die Erweiterungen durchzuführen.

Beispiel: Wetterdaten

```
// Messungen
abstract class Recording {
    protected Date date;
    protected int high;
    protected int low;
    Recording (int high, int low, Date date) {
      this.high = high;
      this.low  = low;
      this.date = date;
    }
    int getHigh() {
      return high;
    }
    int getLow() {
      return low;
    }
    String asString() {
        return date + " : " + low + "-" + high + unit();
    }
    abstract String unit();
}
```


Klassen `Pressure` und `Recording`

```
class Pressure extends Recording {
  Pressure(int high, int low, Date date) {
    super(high, low, date);
  }

  String unit() {
    return "hPa";
  }
}

class Temperature extends Recording {
  Temperature(int high, int low, Date date) {
    super(high, low, date);
  }

  String unit() {
    return "C";
  }
}
```


Dynamische Methodenauswahl bei Vererbung

Der dynamische Typ eines Objekts entspricht dem Klassentyp der Klasse, die bei der Erzeugung des Objektes angegeben wurde.

`<Ausdruck>.<methodenName>( <AktParam1>,...);`

Beim Methodenaufruf wird die Implementierung der Methode basierend auf dem dynamischen Typs des <Ausdruck>s ausgewählt.

Ist eine solche Implementierung in der Klasse des dynamischen Typs nicht vorhanden, wird die Implementierung jener Klasse gewählt, die in der Vererbungshierarchie am weitesten “unten” liegt (d. h. die von der Vererbung her am nächsten ist).

Beispiele: Dynamische Methodenauswahl

1. Auswahl zwischen Methode der Super- und Subklasse:

```
AFigure c = new Circle (new CartPt(4.0,4.0), 1.0);
...
c.distToOrigin();
```

2. Auswahl zwischen Methoden verschiedener Subklassen:

```
void getMaxArea (Figure[] df) {
  double maxArea = 0.0;
   for (int i = 0; i < df.length; i++) {
      Math.max(maxArea, df[i].area());
   }
   return maxArea;
}
```


Zusammenfassung I

Syntax der Klassendeklaration:

```
<Modifikatiorenlist>  class <Klassenname>
    [  extends <Klassenname> ]
    [  implements <Liste von Schnittstellennamen> ]
{
   <Liste von Attribut-,
    Konstruktor-,
    Methodendeklaration>
}
```

Eine Klassen **erweitert** eine direkte Superklasse und **implementiert** ggf. mehrere Schnittstellentypen.

- Der Entwurf geeigneter Klassenhierarchien ist ein zentraler Aspekt des objektorientierten Modellierung. Dabei sind Abstraktion und Spezialisierung sinnvoll zu kombinieren.

Zusammenfassung II

- Abstraktion/Generalisierung erlaubt es, Klassen zu deklarieren, die die relevante Gemeinsamkeiten einer Gruppe von Klassen ausdrücken.
- Spezialisierung erlaubt es, Klassen zu deklarieren, die die Funktionalität existierender Klassen erweitern.
- Vererbung bietet hierzu ein mächtiges Sprachkonzept: Statt Elemente explizit von der Super- in die Subklassen zu kopieren, steht die vererbten Elemente automatisch in der Subklasse bereit.
- Vererbung ist transitiv.