

4. Hierarchische Datenstrukturen in OO-Systemen

Hierarchien in OO-Systemen

Objektorientiertes Programmieren bietet zahlreiche *Strukturierungs-Möglichkeiten* für Objekte und deren zugehörige Klassen.

Die Strukturierung der Objekt- und Klassen-Abhängigkeiten ist wesentliches Ziel der Designphase.

Abhängigkeiten treten hierbei auf zwei wesentlichen Ebenen des Programmes auf.

- Klassenhierarchie (durch Vererbung),
- Abhängigkeitshierarchie (Wer kennt wen?).

Einfachste Abhängigkeit

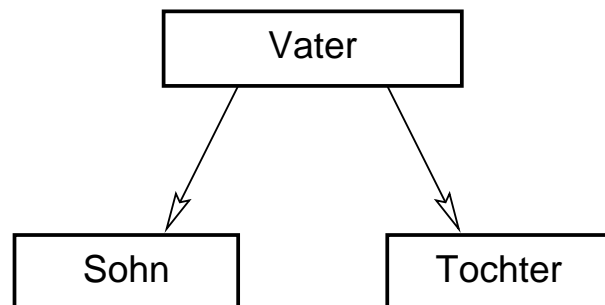
Ein Objekt kann selbst wieder Objekte als Instanz-Variablen haben.

Dieser einfache Zusammenhang ermöglicht das Anlegen sehr komplexer Datenstrukturen.

(Einfachstes) Beispiel:

```
class Vater extends Object{  
    public String name;  
    public Kind Sohn;  
    public Kind Tochter;  
}
```

```
class Kind extends Object{  
    public String name;  
}
```

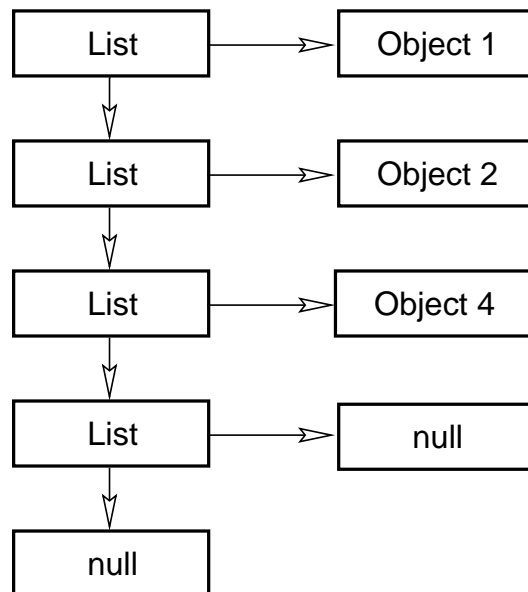


Listen (**Vector**)

Eine *Liste* von Objekten ist ein Objekt, welches (beliebig viele) andere Objekte in geordneter Reihenfolge enthalten kann. (In Java werden Listen durch die Klasse **Vector** realisiert.)

Möglicher Programmcode:

```
class List extends Object{  
  
    public head Object;  
    public tail List;  
  
    public List(){  
        head = null;  
        tail = null;  
    }  
  
    public addObject(Object o){  
        List p = this;  
        while (p.tail != null)  
            p = p.tail;  
        p.tail = new List();  
        p.head = o;  
    }  
    ....  
}
```



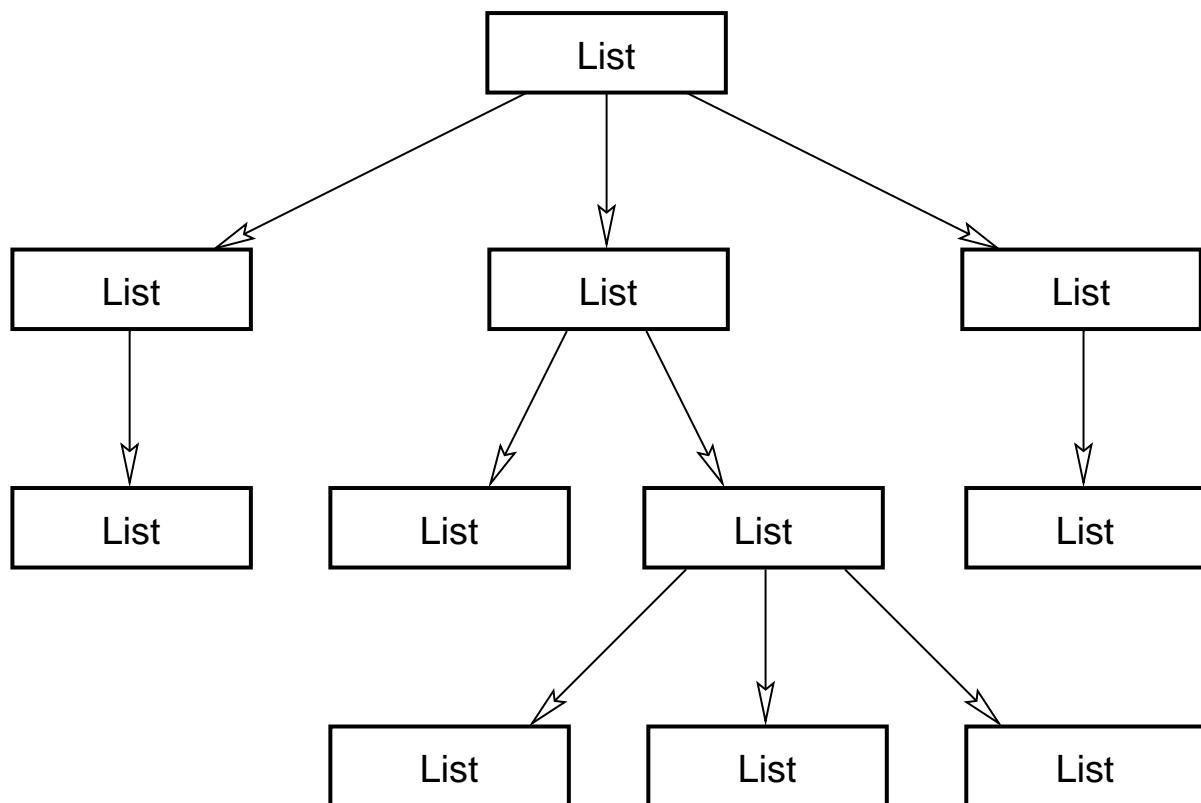
Die Java Klasse **Vector**

```
public class Vector extends Object implements Cloneable, Serializable {
    // Public Constructors
    public Vector(int initialCapacity, int capacityIncrement);
    public Vector(int initialCapacity);
    public Vector();
    // Protected Instance Variables
    protected int capacityIncrement;
    protected int elementCount;
    protected Object[] elementData;
    // Public Instance Methods
    public final synchronized void addElement(Object obj);
    public final int capacity();
    public synchronized Object clone();
    public final boolean contains(Object elem);
    public final synchronized void copyInto(Object[] anArray);
    public final synchronized Object elementAt(int index);
    public final synchronized Enumeration elements();
    public final synchronized void ensureCapacity(int minCapacity);
    public final synchronized Object firstElement();
    public final int indexOf(Object elem);
    public final synchronized int indexOf(Object elem, int index);
    public final synchronized void insertElementAt(Object obj, int index);
    public final boolean isEmpty();
    public final synchronized Object lastElement();
    public final int lastIndexOf(Object elem);
    public final synchronized int lastIndexOf(Object elem, int index);
    public final synchronized void removeAllElements();
    public final synchronized boolean removeElement(Object obj);
    public final synchronized void removeElementAt(int index);
    public final synchronized void setElementAt(Object obj, int index);
    public final synchronized void setSize(int newSize);
    public final int size();
    public final synchronized String toString();
    public final synchronized void trimToSize();
}
```

Bäume

Baumartige Strukturen sind durch unsere Klasse `List` bereits implementiert.

Dies kommt daher, dass die Objekte, die in der Liste gespeichert sind, selbst wieder Listen sein können.



Graphische Abhängigkeiten

Bei graphischen Programmen treten häufig natürliche *Abhängigkeiten* der einzelnen Objekte untereinander auf (z.B. Diagrammeditor “Pfeil zeigt auf Box”).

Designziele:

- Bestehende Abhängigkeiten sollen unabhängig von der aktuellen Darstellung verwaltet werden.
- Abhängigkeiten sollen durch editieren nicht (ungewollt) verändert werden (z.B. beim Verschieben von Objekten).

Zwei Hierarchieebenen

Es ergeben sich (mindestens) zwei sich überlagernde Hierarchieebenen.

Klassenhierarchie:

- Was ist Spezialfall von was?
- Beispiel: *Schnittpunkt* ist Spezialfall von *Punkt*.
- Beispiel: *Pfeil* ist Spezialfall von *Linie*.

Abhängigkeitshierarchie:

- Wer *kennt* wen? Wer *besitzt* wen?
- Wer legt die Position von wem fest?
- Wer muss vor wem berechnet werden?
- Beispiel: *Schnittpunkt* ist festgelegt durch *zwei Geraden*.

Konkrete Beispiele

Beispiel 1: Diagramm Editor

- Objekte: `Box` und `Arrow`
 - `Box`: keine Relationen,
 - `Arrow`: geht von `box1` zu `box2`.

Beispiel 2: Geometrisches Zeichenprogramm

- Objekte: `Point`, `Line` und `IntersectionPoint`
 - `Point`: keine Relationen,
 - `Line`: von `point1` nach `point2`,
 - `IntersectionPoint`: von `line1` und `line2`,
 - `IntersectionPoint`: ist selbst ein `Point`.

Beispiel 3: Graphische Benutzeroberfläche

- Objekte: `Component` und `Container`
 - `Component`: ist Teil eines `Containers`,
 - `Container`: ist selbst ein `Component`.

Beispiel 1: Diagramm Editor

Klassen und Instanz-Variablen:

```
class Diagram extends Object{
    public Vector objects;    // Liste aller Diagrammelemente
}

class DiagramObject extends Object{
    public Color col;        // Farbe eines Elements
}

class BoxObject extends DiagramObject{
    public int x1,y1,x2,y2;   // Position der Box (im Fenster)
}

class Box extends BoxObject{ // Strukturierende Unterklasse
}                               // z.B. verantwortlich fuer draw

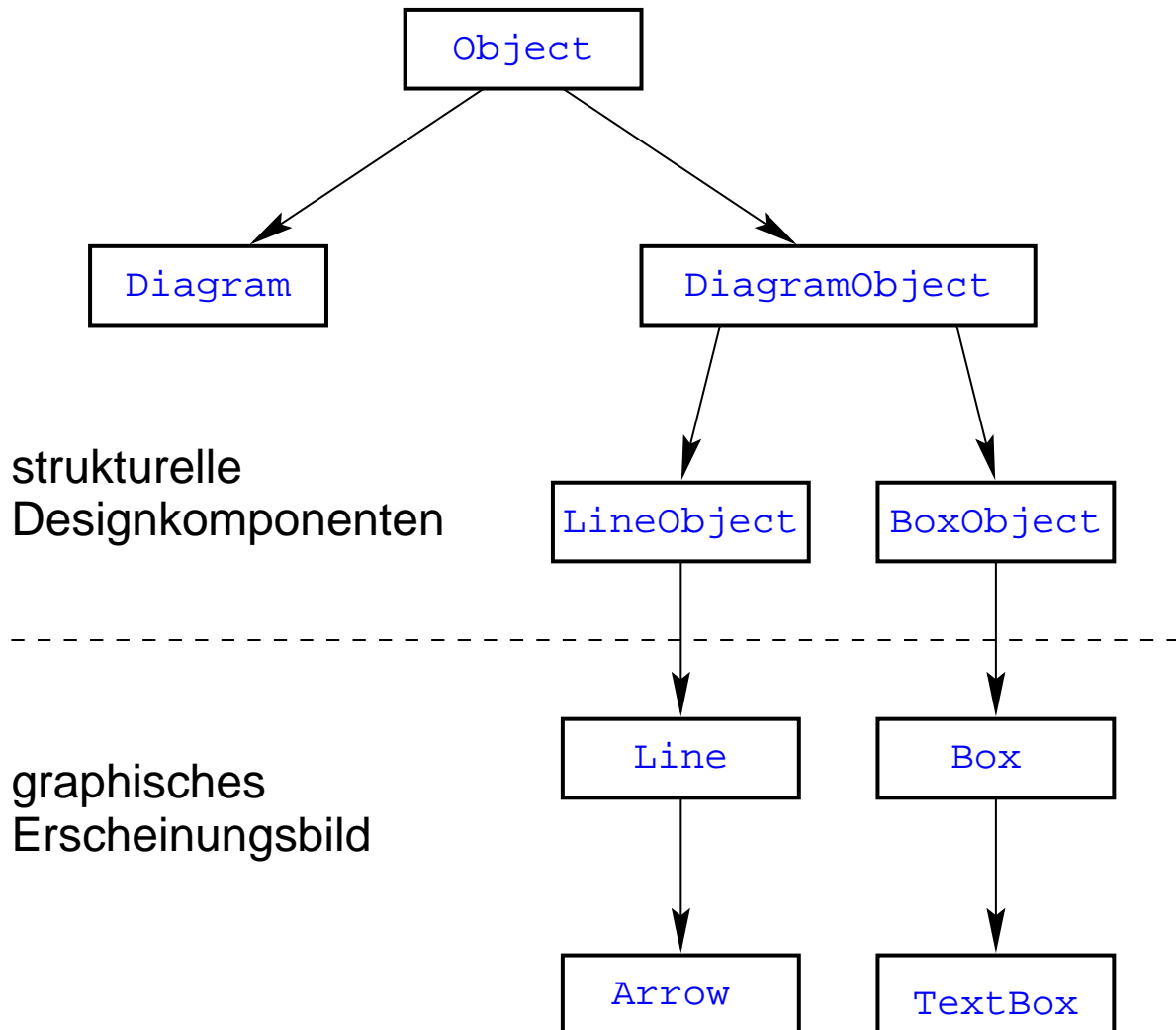
class TextBox extends Box{
    String text;             // Beschriftung der Box
}

class LineObject extends DiagramObject{
    BoxObject box1,box2;     // Gerade von box1 nach box2
}

class Line extends LineObject{ // Strukturierende Unterklasse
}                               // z.B. verantwortlich fuer draw

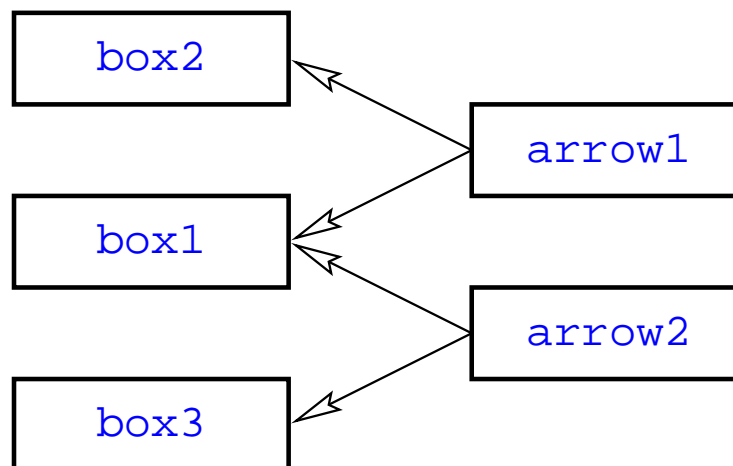
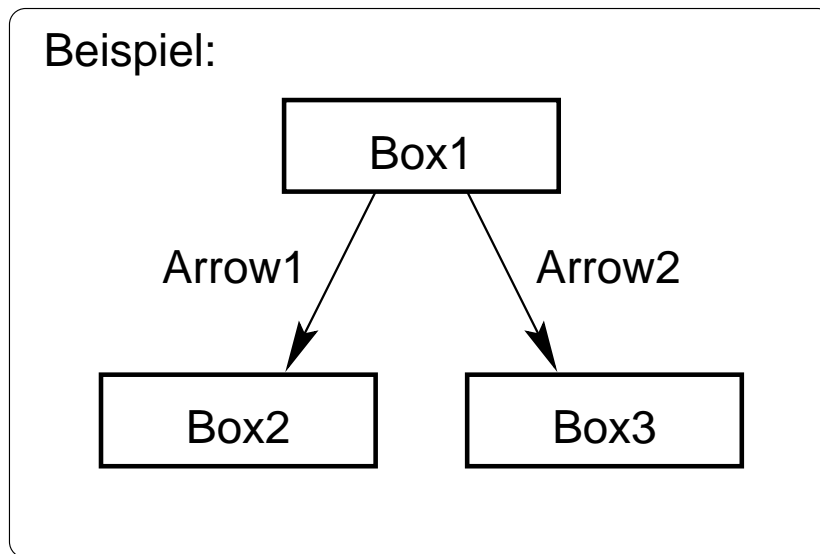
class Arrow extends Line{
    int size;                // Groesse des Pfeils
}
```

Klassenhierarchie



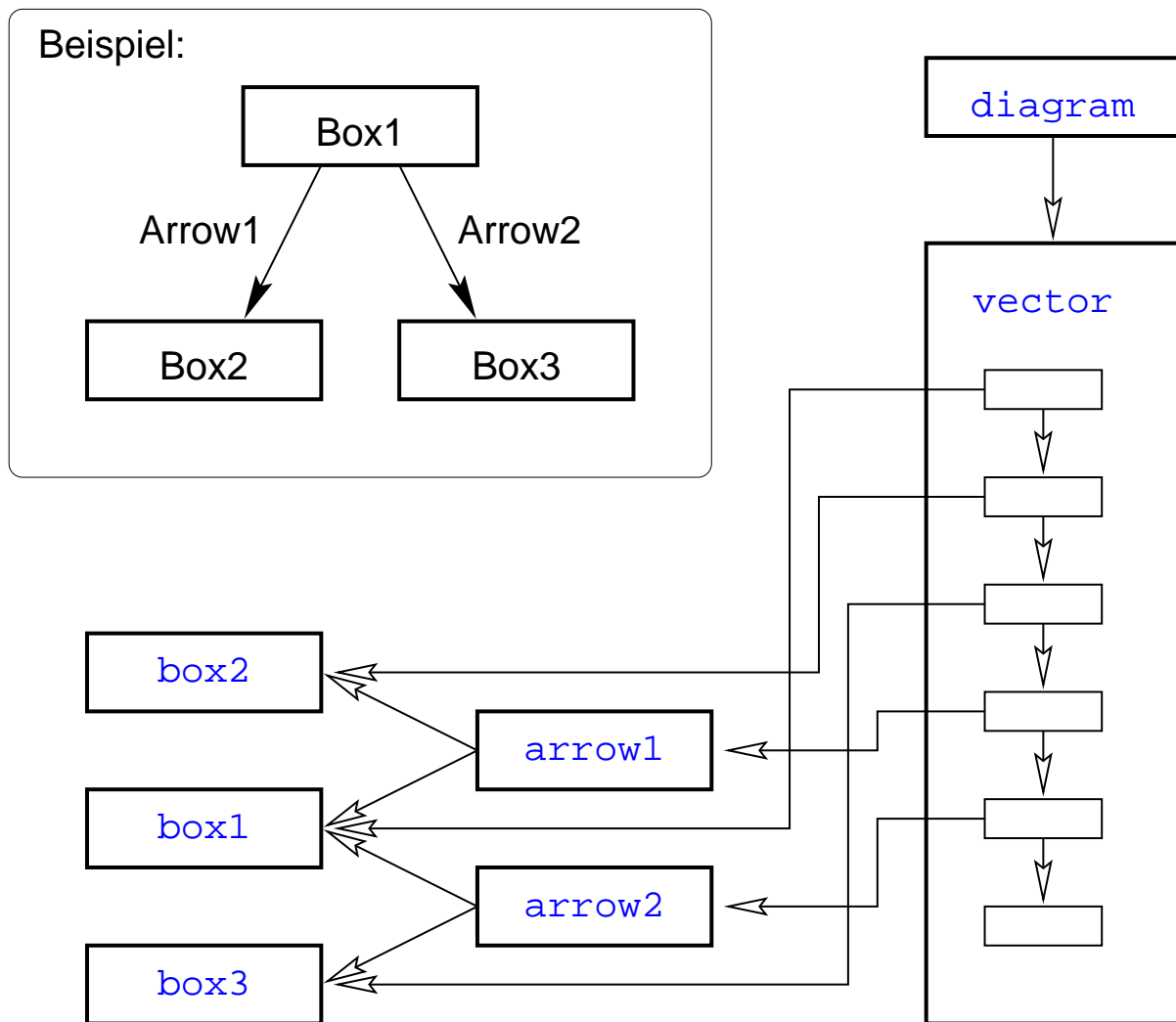
Wer kennt wen?

Bei diesem Objekt-Design speichert ein Objekt der Klasse `LineObject` selbst (Referenzen auf) die Boxen, die die beiden Anfangspunkte der Linie definieren.



Wer kennt wen? (II)

Das Objekt `diagram` speichert eine Liste (`vector`) aller "Komponenten" der Zeichnung.



Die Zeichenmethoden

Die Klasse `Diagram` ist verantwortlich dafür, dass die einzelnen Diagrammkomponenten gezeichnet werden

```
class Diagram extends Object{
    public void paint(Graphics g){
        for (int i = 0; i < objects.size(); i++) {
            ((DiagramObject)objects.elementAt(i)).draw(g);
        }
    }
}
```

Jede Komponente “weiss”, wie sie sich selbst zeichnen kann.

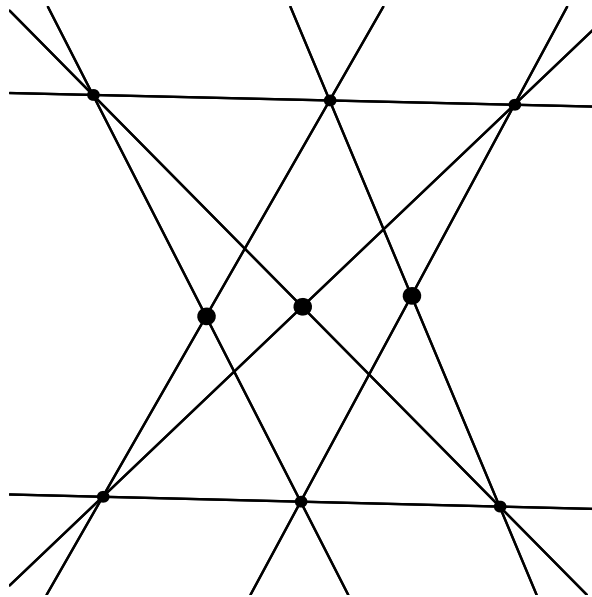
```
class DiagramObject extends Object{
    abstract public void draw(Graphics g);
}

public class Box extends BoxObject {
    public void draw(Graphics g) {
        g.setColor(color);
        g.drawRect(x,y,w,h);
    }
}

public class Line extends LineObject {
    public void draw(Graphics g) {
        g.setColor(color);
        g.drawLine(box1.getCenter().x, box1.getCenter().y,
            box2.getCenter().x, box2.getCenter().y);
    }
}
```

Beispiel 2: Geometrisches Zeichenprogramm

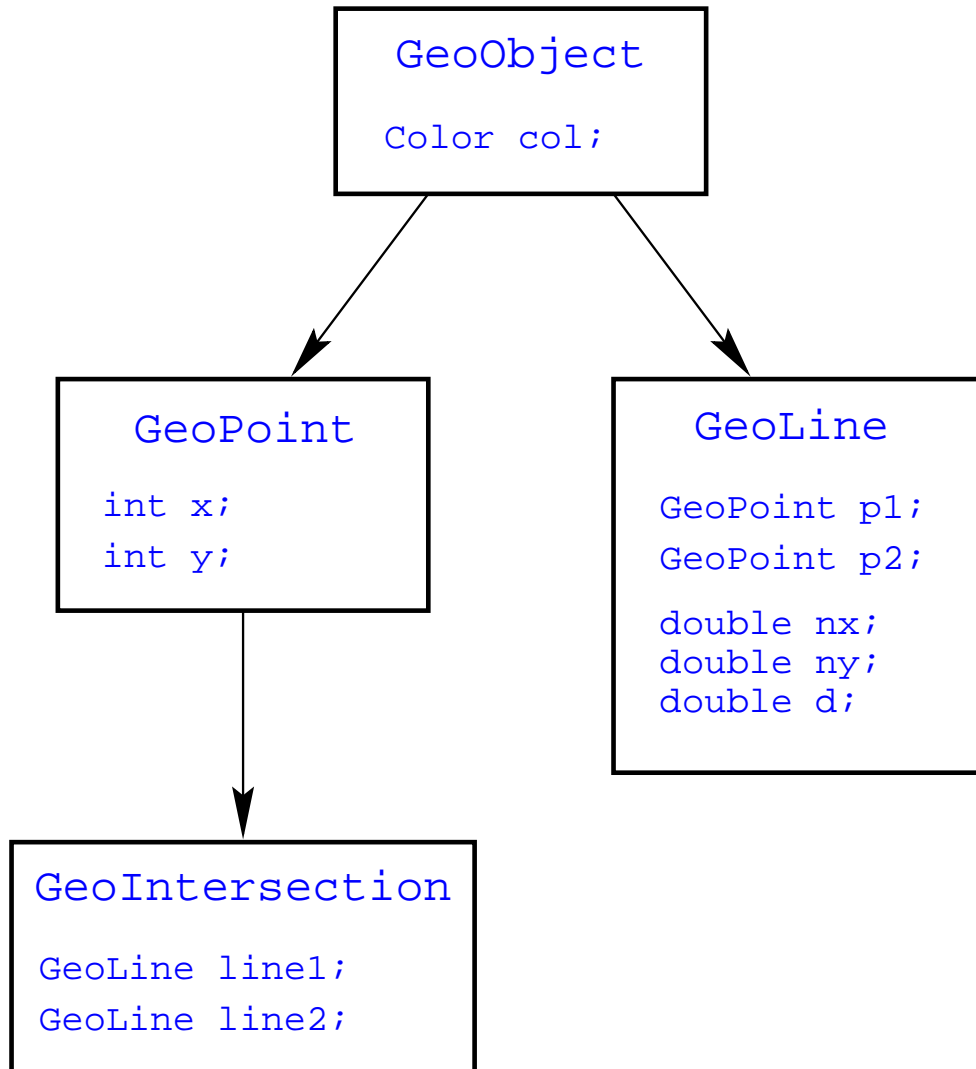
Ziel: Erstellen geometrischer Zeichnungen, an denen ein geometrischer Zusammenhang *interaktiv* demonstriert werden kann.



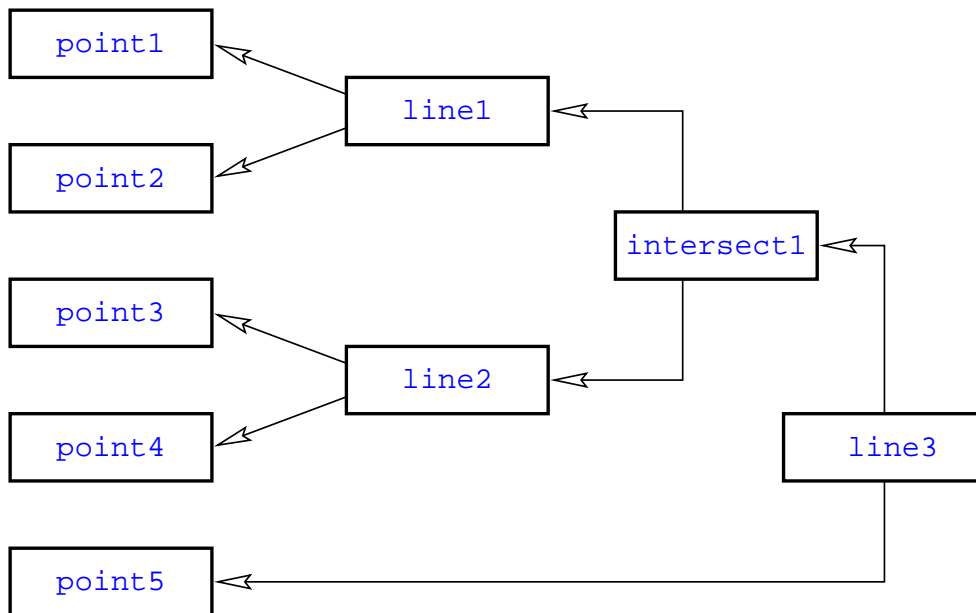
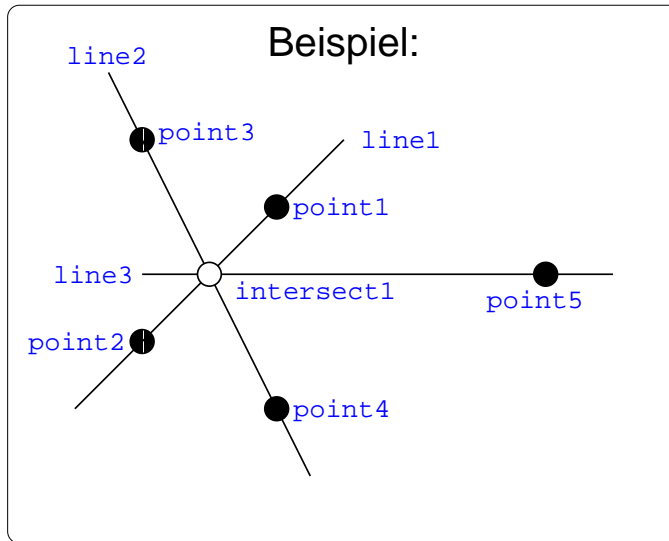
Klassen:

- [GeoObject](#): Geometrisches Objekt,
- [GeoPoint](#): "Freier" Punkt,
- [GeoLine](#): Verbindungsgerade zweier Punkte,
- [GeoIntersection](#): Schnittpunkt zweier Geraden.

Klassenhierarchie



Wer kennt Wen?



“Tiefe” Datenstrukturen

Die zugrundeliegende Datenstruktur im Beispiel Zeichenprogramm kann (im Gegensatz zum Diagramm Editor) sehr “tief” werden. (Objekte, die von Objekten abhängen, die von Objekten abhängen, die von Objekten abhängen, die von Objekten abhängen, . . .)

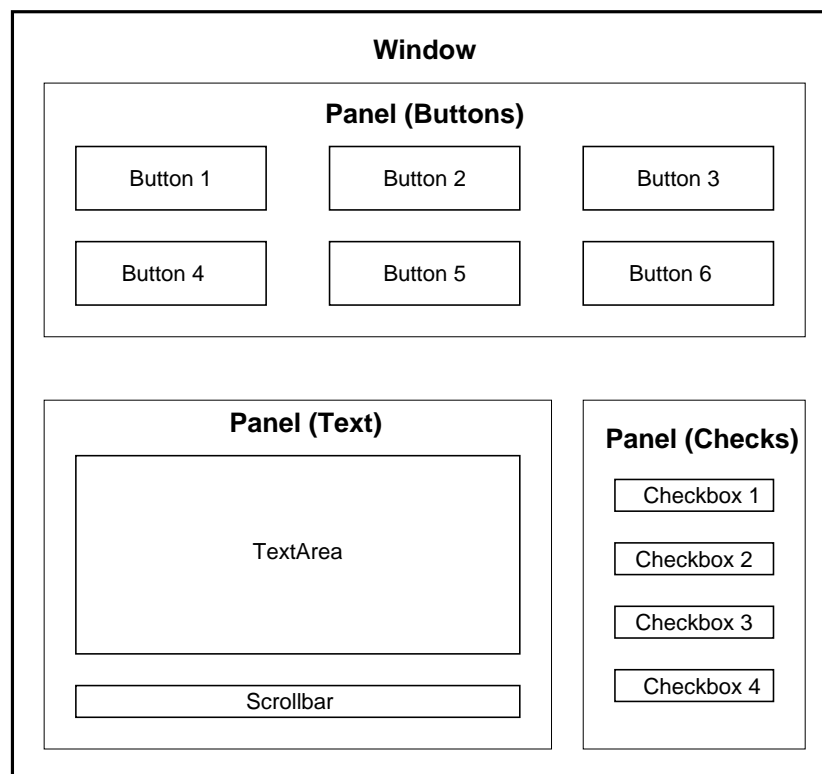
Um dieser Tatsache Rechnung zu tragen, bieten sich folgende Designmassnahmen an:

- Trennen von Positions-Berechnung und Zeichnen,
- *Rückwärtsverkett*en der Abhängigkeiten,
- Logik für “Vorfahren” und “Nachkommen”,
- nur die Elemente neu berechnen, die sich tatsächlich verändert haben (Nachkommen eines bewegten Elementes).

Beispiel 3: Graphische Benutzeroberfläche

Die *graphischen Benutzeroberflächen* in fensterorientierten Systemen tragen auf natürliche Weise eine hierarchische Struktur.

Fenster sind in (zumeist rechteckige) Sinneinheiten gegliedert, die *Knöpfe*, *Schieberegler*, *Textbereiche*, *Zeichnungen*, ... in strukturierter Weise enthalten.



Component — Container

Component:

Oberklasse aller auf dem Bildschirm darstellbarer Objekte ([Button](#), [TextField](#), [Scrollbar](#), [Canvas](#), [Container](#), ...)

Container:

Unterklasse von [Component](#). Bildschirmobjekt, welches selbst wieder eine Menge von [Component](#)-Objekten “enthalten” kann.

[Object](#) verhält sich zu [Vector](#)
wie [Component](#) zu [Container](#).

Baumartige Zeichen-Hierarchie

Die durch `Component` – `Container` Relationen gegebene Hierarchie hat automatisch die Struktur eines *Baumes*.

Dies wird bei der Implementierung des Fenstersystems von Java an verschiedenen Stellen ausgenutzt.

Beispiel: die `draw` Methode:

```
class Container extends Component{
    private Vector subComponents;
    ....
    public void draw(Graphics g) {
        for (int i=0,i<subComponents.size(),i++)
            ((Component) subComponents.elementAt(i)).draw(g);
        }
    }
    ....
}
```

Ein einziger Aufruf der Methode `draw` eines Containers bewirkt, dass *rekursiv* alle in ihm enthaltenen Komponenten (auch Container) gezeichnet werden.

