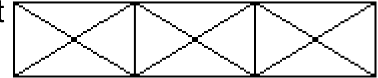
Ein Beziehungstyp: Schrank und Schrankwand

Wir wollen in unser Raumplanersystem eine Schrankwand mit aufnehmen, die aus mehreren gleichen Schrankelementen zusammenzustellen ist.
Dazu sollte man zunächst einmal ein einzelnes Schrankelement zeichnen können, so dass wir uns auch eine Klasse Schrank konstruieren.



Wir verwenden wieder die abstrakte Klasse Moebel, so dass vom Text eigentlich nur die übliche Methode **`gibAktuelleFigur()`** interessant ist:

```
/**
 * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
 */
protected Shape gibAktuelleFigur()
{
    GeneralPath schrank = new GeneralPath();
    Shape rahmen = new Rectangle2D.Double(0, 0, breite, tiefe);
    Shape linie1 = new Line2D.Double(0, 0, breite, tiefe);
    Shape linie2 = new Line2D.Double(breite, 0, 0, tiefe);
    schrank.append(rahmen, false);
    schrank.append(linie1, false);
    schrank.append(linie2, false);

    return  transformiere(schrank);
}
```

In diesem Fall habe ich die Figur aus einem Rechteck und zwei Linien (das innere Kreuz) zusammengesetzt. Man muss das nicht so machen, es geht natürlich auch mit der Methode lineTo von GeneralPath, den wir wegen des Zusammensetzens sowieso benötigen. Nun lässt sich auf einfache Weise eine Schrankwand aus drei solchen Schränken zusammensetzen:
`gibAktuelleFigur()` lautet dann:

```
/**
 * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
 */
protected Shape gibAktuelleFigur()
{
    GeneralPath schrank = new GeneralPath();
    Shape rahmen = new Rectangle2D.Double(0, 0, breite/3, tiefe);
    Shape linie1 = new Line2D.Double(0, 0, breite/3, tiefe);
    Shape linie2 = new Line2D.Double(breite/3, 0, 0, tiefe);
    schrank.append(rahmen, false);
    schrank.append(linie1, false);
    schrank.append(linie2, false);

    rahmen = new Rectangle2D.Double(breite/3, 0, breite/3, tiefe);
    linie1 = new Line2D.Double(breite/3, 0, 2*breite/3, tiefe);
    linie2 = new Line2D.Double(2*breite/3, 0, breite/3, tiefe);
    schrank.append(rahmen, false);
    schrank.append(linie1, false);
    schrank.append(linie2, false);

    rahmen = new Rectangle2D.Double(2*breite/3, 0, breite/3, tiefe);
    linie1 = new Line2D.Double(2*breite/3, 0, breite, tiefe);
    linie2 = new Line2D.Double(breite, 0, 2*breite/3, tiefe);
    schrank.append(rahmen, false);
    schrank.append(linie1, false);
```

```
        schrank.append(linie2, false);

        return  transformiere(schrank);
    }
```

Zu dieser Lösung lassen sich viele Überlegungen machen. Eine betrifft die Größenangabe des Attributes breite: sie ist hier auf den dreifachen Wert der Breite eines Schrankes gesetzt, so dass auch die Methode `dreheAuf(...)` das – bezogen auf diese Breite – Richtige macht: Drehzentrum ist der Schnittpunkt des mittleren Kreuzes.
Es gibt aber wichtige weitere Überlegungen, die insbesondere die Modellierung betreffen. Erkennbar ist es wieder daran, dass wir durch Codeduplizierung die Klasse Schrankwand erzeugt haben und nicht aus eigenem Code.

Statt Codeduplizierung durchzuführen gilt es, sich vorher über die Beziehungen zwischen den Klassen Gedanken zu machen. Im Klassendiagramm der derzeitigen Lösung gibt es aber keine Beziehung zwischen den beiden, sondern nur eine erbt – Beziehung zu Moebel.

Die beiden Klassen haben ganz offensichtlich eine Beziehung, man kann nämlich die Klasse Schrank nutzen, um eine Schrankwand zu erzeugen. Grundsätzlich tritt also eine Nutzerbeziehung[1] auf. Der Gedanke ist: Wozu muss die Schrankwand wissen, wie ein einzelner Schrank aussieht. Das kann Schrank übernehmen.
Die Lösung bereitet aber mehr Schwierigkeiten, als man zunächst vermutet. Eine erste Veränderung ergibt sich daraus, dass Schrankwand das Erzeugen der Objekte nun an die Klasse Schrank weiterreichen muss und daher deren Konstruktor so verändert werden muss, dass er nicht mit Standardwerten arbeitet, sondern mit den von der Klasse Schrankwand bestimmten. Ihr Konstruktor bekommt daher die Form:

```
  /**
   * Erzeuge einen neuen Schrank.
   */
    public Schrank(int x, int y, String f, int o, int b, int t)
  {
    xPosition = x;
    yPosition = y;
    farbe = f;
    orientierung = o;
    istSichtbar = false;
    breite = b;
    tiefe  = t;
  }
```

Nun kann Schrankwand mit

```
      schrank1 = new Schrank(0, 0, farbe, orientierung, breite/3, tiefe);
```

ein Objekt vom Typ Schrank erzeugen.

1 Genauer formuliert, stellt Schrankwand eine spezielle Form einer Aggregation dar: Eine Schrankwand besteht nämlich aus Schrankobjekten. Wir werden uns damit noch beschäftigen.

Wo macht man das? - Wohin gehört der Aufruf in der Klassendefinition?

Die Antwort sollte sich an der Frage orientieren: Wann wird ein Objekt der Klasse Schrankwand erzeugt?
Das Erzeugen – und damit das Definieren – der Schrankobjekte sollte also im Konstruktor der Klasse Schrankwand erfolgen. Die Schrankobjekte sind Objektvariablen von Schrankwand und müssen daher im Kopf der Klassendefinition deklariert werden.
In der Methode `gibAktuelleFigur()` arbeiten wir mit einem GeneralPath schrankwand, dem wir nacheinander die drei Schränke mit append hinzufügen.
Leider arbeitet `gibAktuelleFigur()` aber noch nicht so wie wir das brauchen, wir bekommen eine Fehlermeldung beim Aufruf von append, die bei genauerem Nachdenken völlig klar ist. Die Methode append benötigt Shape – Objekte. Unsere Schrankobjekte implementieren aber – wie man dem Klassendiagramm entnehmen kann, da es Shape nicht enthält – das Interface Shape überhaupt nicht. Das wollen wir an dieser Stelle auch gar nicht.
Ein Moebelobjekt ist kein Shape, es hat zwar eine grafische Darstellung, mit dieser ist es aber nicht identisch, steht auch nicht für sie. Von dem Möbelobjekt benötigen wir für die Methode `gibAktuelleFigur()` aber nur das Shape, für das es steht. Dies können wir aber bekommen, indem wir eben gerade diese Methode `gibAktuelleFigur()` für die Schrankobjekte benutzen. Sie gibt uns den „Shape – Aspekt“ der Schränke.

Damit kann unsere Klassendefinition so aussehen:

```
import java.awt.Shape;
import java.awt.geom.GeneralPath;

/**
 * Eine Schrankwand, die ...
 */
public class Schrankwand extends Moebel          [erbt weiterhin von Moebel]
{
  private Schrank schrank1;
  private Schrank schrank2;                      [Deklaration der Variablen]
  private Schrank schrank3;

  /**
   * Erzeuge eine neue Schrankwand mit einer Standardfarbe und Standardgroesse
   * an einer Standardposition.
   */
    public Schrankwand()
  {                                              [Konstruktor]
    xPosition = 40;
    yPosition = 80;
    farbe = "blau";
    orientierung = 0;
    istSichtbar = false;
    breite = 180;                                [Definition der Variablen]
    tiefe  = 37;
    schrank1 = new Schrank(0, 0, farbe, orientierung, breite/3, tiefe);
    schrank2 = new Schrank(breite/3, 0, farbe, orientierung, breite/3, tiefe);
    schrank3 = new Schrank(2*breite/3, 0, farbe, orientierung, breite/3, tiefe);
  }
```

```
  /**
   * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
   */
  protected Shape gibAktuelleFigur()
  {
      GeneralPath schrankwand = new GeneralPath();
      schrankwand.append(schrank1.gibAktuelleFigur(), false);
      schrankwand.append(schrank2.gibAktuelleFigur(), false);
      schrankwand.append(schrank3.gibAktuelleFigur(), false);

      return  transformiere(schrankwand);
  }

}
```


Warum funktioniert das?

Bisher ist alles noch sehr einfach zu verstehen. Schwieriger wird es zu erklären, weshalb nun auch die Methoden richtig arbeiten. Untersuchen wir einmal den Aufruf der Methode `aendereFarbe("rot")` für ein Objekt schrankwand1.
Es wird die von Moebel geerbte Methode verwendet, die selbst nur die Variable `farbe` von schrankwand1 neu setzt und dann die Methode `zeichne()` aufruft. Dies ist wieder eine Methode von Moebel, die nun – um sich das zu zeichnende Shape zu beschaffen – die Methode `gibAktuelleFigur()` von schrankwand1 aufruft. Diese Methode ruft jeweils Shapes mit der Methode `gibAktuelleFigur()` von den drei Schrankobjekten ab und fügt sie zu einem Shape zusammen.
Es wird gezeichnet mit Hilfe der Methoden von Leinwand und hier geschieht nun der entscheidende Aspekt: `leinwand.zeichne(this, farbe, figur);` wird mit dem Parameter `farbe` aufgerufen und das ist die Farbe von `schrankwand1`! Egal, welche Farbattribute die einzelnen Schrankobjekte haben, es wird zum Zeichnen immer der aktuelle Attributwert von `farbe` des Objektes `schrankwand1` genommen, da dieses die zeichne – Methode von leinwand aufruft!
Das kann gewollt sein, ist aber durchaus unbefriedigend, wenn man mit dem Inspektor sich die Attributwerte der drei Schränke ansieht: Alle haben noch den Wert „blau“!

Aufgabe:

- Untersuchen Sie: Wie sieht es bei `bewegeVertikal( 50 )` aus?
- Was passiert bei den anderen Methoden?

Verblüffenderweise geht auch bei den Verschiebemethoden alles gut. Dies liegt daran, dass die Schränke weiterhin nur ihre relative Position in `schrankwand` „kennen“ und diese beim Aufruf von `gibAktuelleFigur()` zurückliefern. Diese relative Position ändert sich aber durch ein Verschieben der gesamten Schrankwand nicht. Dass diese schließlich insgesamt richtig steht, regelt ihre eigene Transformation, wenn die einzelnen Teile mit relativ richtiger Position – durch deren Transformation – mit schrankwand.append(...) eingebaut wurden. Hier macht sich bemerkbar, dass sich lineare Transformationen problemlos verketten lassen.
Auch `dreheAuf()` arbeitet aus dem selben Grund richtig.