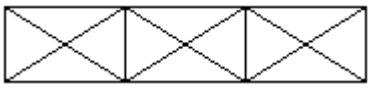
Ein Beziehungstyp: Schrank und Schrankwand

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);
}
```

Überlegungen

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.

Codeduplizierung ist immer ein Signal zum Nachdenken

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.

Beziehungen

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;
}
```

Beziehung realisieren

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.

Eine einfache Lösung

Damit kann unsere Klassendefinition so aussehen:

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

/**
 * Eine Schrankwand, die ...
 */
public class Schrankwand extends Moebel
{
  private Schrank schrank1;
  private Schrank schrank2;
  private Schrank schrank3;

  /**
   * Erzeuge eine neue Schrankwand mit einer Standardfarbe und Standardgroesse
   * an einer Standardposition.
   */
    public Schrankwand()
  {
    xPosition = 40;
    yPosition = 80;
    farbe = "blau";
    orientierung = 0;
    istSichtbar = false;
    breite = 180;
    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);
  }
```

erbt weiterhin von Moebel

Deklaration der Variablen

Konstruktor

Definition der Variablen

```
  /**
   * 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);
  }

}
```

Weiterhin ein GeneralPath, ...

... dem die Shapes hinzugefügt werden.

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.

Modellierung

Die Objektorientierung hat nicht nur mit den Konzepten Klasse und Objekt neue strukturelle Inhalte in die Informatik gebracht, sondern auch bei der Analyse und beim Entwurf neue Konzepte entwickelt.

Entwurfsmuster

Einerseits hat man sich mit Entwurfsmuster beschäftigt. Dazu gibt es ein Standardwerk von Gamma[1] u.a. , mit einigen dieser Entwurfsmuster werden wir uns noch beschäftigen.

UML

Wie schon vorher erwähnt, hat sich in der OO für die Kennzeichnung von Klassenbeziehungen inzwischen ein Quasistandard entwickelt (und wird noch weiter entwickelt), der mit ***UML*** = ***U***nified ***M***odeling ***L***anguage bezeichnet wird. Dazu gibt es ein Standardwerk von Bernd Oesterreich[2].

Beziehungstypen

U.a. beschäftigt sich die UML mit Beziehungstypen. Einige von ihnen untersuchen wir auch in diesem Kurs. Wir haben – orientiert an den Möglichkeiten, die BlueJ zur Zeit bietet – uns bisher nur mit den beiden Beziehungstypen erbt – Beziehung (auch als „ist ein -“ bezeichnet) und Nutzerbeziehung (auch als „hat ein -“ bezeichnet) beschäftigt. Das wollen wir nun etwas genauer untersuchen.

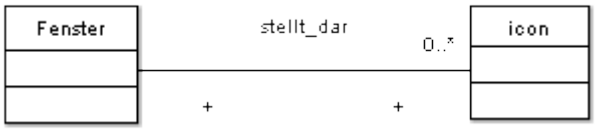
Assoziation und Aggregation

Assoziation

Die beiden Begriffe beschreiben Beziehungstypen. Es geht dabei um Beziehungen zwischen Objekten. Die Objekte werden in vielen Anwendungsfällen nicht derselben Klasse angehören, aber sie können derselben Klasse angehören.
Untersucht man z.B. die Oberflächen, die Windows oder KDE o.ä. uns anbieten, dann weisen die *icons* diesen Beziehungstyp auf: Die Fensteroberfläche stellt die icons dar. „*stellt dar*“ ist dabei eine sprachliche Beschreibung der hier vorliegenden Beziehung, genauso, wie wir sie oben bei „*ist ein*“ und „*hat ein*“ kennengelernt haben. Die Assoziation beschreibt davon die Nutzerbeziehungen.

Fensteroberfläche hat icons

Betrachten wir wieder die Fensteroberfläche, dann können wir feststellen, dass wir über die Oberfläche sagen können sie „stellt dar“ – wie oft eben nicht einen, sondern mehrere – icons.



Ein Fahrrad „hat einen“ Fahrer, eine Abteilung einer Firma „hat einen“ Mitarbeiter usw. Andererseits hat ein Fahrrad nicht nur einen Fahrer, es *besteht aus* sehr vielen Einzelteilen. Damit kommen wir auf einen Spezialfall einer Assoziation, die Aggregation.

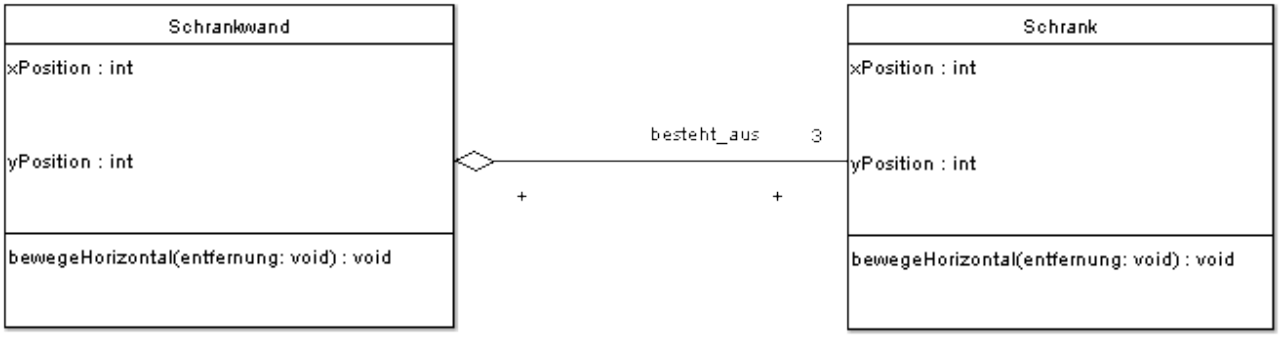
1 Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: Entwurfsmuster – Elemente wiederverwendbarer Software ; Addison – Wesley ; ISBN 3-89319-950-0
2 Bernd Oesterreich: Objektorientierte Softwareentwicklung – Analyse und Design mit der UML ; Oldenbourg ; ISBN 3-486-24787-5

Aggregation

Die Aggregation ist ein besonders interessanter Spezialfall, Bernd Oesterreich bezeichnet sie als Teil – Ganzes – Beziehung oder „hat“ - Beziehung. Oesterreich schreibt in seinem Buch:
Eine Aggregation ist die Zusammensetzung eines Objektes aus einer Menge von Einzelteilen. Ein Auto ist beispielsweise eine Aggregation von Rädern, Motor, Lenkrad usw. Auch diese Teile sind ggf. wieder Aggregationen: eine Bremse besteht aus ...
Statt von einzelnen Aggregationen ist manchmal von Teile-Ganzes-Hierarchien die Rede.

Die grafische Kennzeichnung der Beziehung im Klassendiagramm erfolgt mit einer einfachen Linie und einer offenen Raute am Ganzes – Ende. Dabei kann man an den Enden die möglichen Anzahlen der Beteiligten darstellen.

Schrankwand



(Diagramm erstellt mit ArgoUML)

Eine Schrankwand ist ein Beispielfall für diesen speziellen Beziehungstyp Aggregation. Die Schrankwand „*hat ein*“ – hier wieder in der Regel mehrere – Schrankelemente. Das klingt etwas holprig. Wir würden nämlich eher sagen: Eine Schrankwand „*besteht aus*“ z.B. drei Schrankelementen. Ist das Ende mit einer einzelnen Zahl gekennzeichnet, wie hier mit der 3, dann müssen es immer 3 Schränke sein, aus denen eine Schrankwand besteht. Sind es mindestens 1 und können theoretisch beliebig viele sein, dann gibt man an dem Ende 1..* an, bei höchstens drei würde man 0..3 angeben.

Komposition

Für die Verwendung des Begriffes Aggregation ist es wichtig zu klären, ob die benutzten Klassen auch unabhängig von der benutzenden Klasse existenzfähig sind. Falls diese Unabhängigkeit nicht gegeben ist, spricht man speziell von einer *Komposition*.
Ob die einzelnen Schrankelemente einer Schrankwand für sich „lebensfähig“ sind, kann man durchaus bezweifeln. Daher wäre die Beziehung zwischen beiden ggf. eine Komposition und man müsste statt der offenen Raute eine geschlossene verwenden.

Komposition und Kompositum

Eine unglückliche Begriffsähnlichkeit tritt bei den beiden Begriffen Komposition und dem Entwurfsmuster Kompositum auf. Es handelt sich nicht nur bei einer Komposition um eine besondere Form von Aggregation. Auch bei einem Kompositum geht es um eine spezielle Form von Aggregation:

Oesterreich[1]: *Mit dem Kompositum – Muster werden baumartige Aggregationen hergestellt (zusammengesetzt), die sowohl als einzelne Objekt als auch als Zusammmensetzung von Objekten in gleicher Weise benutzt werden können*.

Im Gegensatz zur Komposition können die Elemente eines Kompositum durchaus selbstständig lebensfähig sein. Das besondere Merkmal eines Kompositum ist, dass auch die ganze Aggregation (im Prinzip) dieselbe Schnittstelle aufweist, wie ihre Elemente. Das gilt auch rekursiv geschachtelt. Angewandt z.B. auf eine Schrankwand bedeutet das, dass wir sie als Ganzes verschieben, drehen oder umfärben können, selbst Komposita geschachtelt in andere Komposita werden dann mit verschoben usw.
Wegen der möglichen Selbstbezüglichkeit ist die grafische Darstellung des Kompositummusters etwas komplizierter, u.a. enthält sie auch „ist ein“ – Beziehungen.

Anwendung des Kompositummusters ?

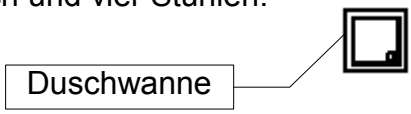
Weder unser erster Entwurf für die Schrankwand noch der zweite sind wie das Entwurfsmuster Kompositum realisiert. Die Behandlung von Entwurfsmustern allgemein und des Entwurfsmusters Kompositum speziell steht noch aus!

Anwendungsfälle von Aggregationen
Zunächst einmal wollen wir uns mit einfachen Anwendungsfällen von Aggregationen beschäftigen, ohne über die Entwurfsmuster nachzudenken. Dabei wollen wir auch nach Fällen von Kompositionen sehen.

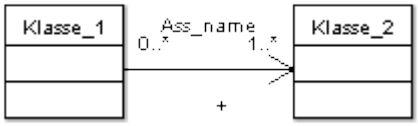
Aufgabe:

Untersuchen Sie die folgenden Beispiele daraufhin, ob sie Assoziationen, Aggregationen oder Kompositionen enthalten:

- Eine Sitzgruppe, bestehend aus einem quadratischen Tisch und vier Stühlen.
- Eine Duschwanne, bestehend aus drei Quadraten
- Ein Kochherd, der einen Benutzer benötigt.
- Ein Kochherd, der eine Küche benötigt.
- Auto, Fahrrad, Fahrzeug, PKW, LKW, Reifen, Türen, Fahrer



Zeichnen Sie die zugehörigen UML – Diagramme (auch in BlueJ – Variante) !



1Siehe dazu auch Gamma: „Entwurfsmuster“ ab S.239.