Vererbung

Es ist sehr unbefriedigend, große Programmtextabschnitte durch Kopieren in andere Klassendefinitionen zu übertragen, ohne dass in weiten Teilen auch nur eine Zeile geändert wird. Absolut naheliegend ist, diese Abschnitte in eine eigene „Datei“ auszugliedern, um dann jedesmal darauf zuzugreifen. In der OO ist das geeignete Mittel dafür aber keine Datei, sondern eine neue Klasse (obwohl JAVA für den Klassentext natürlich – wie bei allen vorher uns uns betrachteten Klassen – eine eigene Datei vom Typ *.java verwendet.)
Hierin steckt aber nicht nur ein Schreibkonstrukt, sondern eine Abstraktionsstufe. Wir suchen die gemeinsamen Attribute und Methoden aller Möbelklassen, abstrahieren dabei von einer konkreten Möbelklasse und definieren das Abstraktionsergebnis durch eine eigene Klasse Moebel. Diese stellt nun die allen anderen Klassen gemeinsamen Eigenschaften bereit und die von ihr erbenden – abgeleiteten – Klasse enthalten nur noch genau die Methoden, in denen sie sich unterscheiden.
Bei unseren neuen Möbelklassen haben wir herausgefunden, worin sie sich unterscheiden: Es ist eigentlich nur die Methode **`gibAktuelleFigur()`**. Allerdings enthält sie in der vorliegenden Form noch einen Abschnitt, der in allen Klassen gleich ist, nämlich der Teil, in dem die Transformation des konkreten Falles eines Shape durchgeführt wird.

Hier zeigt sich, dass die ursprüngliche Modellierung ungünstig war, da die Methode eigentlich zwei Aufgaben erfüllte:
1. Die Definition der konkreten Figur als GeneralPath, Arc2D.Double, Ellipse2D.Double, Rectangle2D.Double, Line2D.Double, CubicCurve2D.Double, QuadCurve2D.Double, Polygon (o.ä.)
2. Die Transformation dieser konkreten Figur als Shape durch eine Verkettung von linearen Transformationen, um die gewünschte Lage und Orientierung zu erzielen.

Sinnvollerweise gliedert man den Code, der gemeinsam ist, in eine eigene Methode aus, die wir hier **`transformiere(Shape shape)`** nennen können.
Die Methode wäre dann:

```
protected Shape transformiere(Shape shape)
{
    AffineTransform t1 = new AffineTransform();
    AffineTransform t2 = new AffineTransform();
    t1.translate(xPosition, yPosition);
    t2.rotate(Math.toRadians(orientierung),gibMitteX(),gibMitteY());
    t2.concatenate(t1);
    return  t2.createTransformedShape(shape);
}
```

Sie besteht aus zwei Transformationen. Die erste ist eine Translation, also eine Verschiebung auf die Zielkoordinaten **`(xPosition, yPosition)`**. Die zweite ist eine Rotation um das von **`gibMitteX()`** und **`gibMitteY()`** gelieferte Drehzentrum. Beide werden durch die Methode **`concatenate`** der zweiten Transformation mit einander verkettet.

protected

Gegenüber der Methode **`gibAktuelleFigur()`** der ursprünglichen Klassendefinition hat sich außerdem die Angabe zur Sichtbarkeit geändert. Weshalb das notwendig ist, kann man leicht feststellen, wenn man es bei **`private`** belässt: Bei dieser Definition kann die nun in einer anderen Klasse stehende Methode nicht aufgerufen werden. **`protected`** macht eine Methode innerhalb des gesamten Paketes sichtbar, nicht aber nach außen.

extends

Es ergibt sich das Problem, dass unser Projekt nun zwar eine neue Klasse hat, BlueJ und JAVA aber nichts davon wissen, dass die Möbelklassen ihre Attribute und Methoden von der Klasse Moebel erben sollen. Das müssen wir in den Klassentext einarbeiten. Der Kopf wird ergänzt um **`extends Moebel`** . Nun weiß JAVA beim Übersetzen, dass es dort nach den fehlenden Attributen und Methoden nachsehen kann.

abstract

Wenn wir die gemeinsamen Methoden in die neue Klasse Moebel ausgegliedert haben und die verkürzte Methode **`gibAktuelleFigur()`** in der konkreten Möbelklasse – z.B. Stuhl – verblieben ist, dann kennt die Klasse Moebel die Methode **`gibAktuelleFigur()`** nicht. Es gibt aber Methoden in Moebel, die darauf zugreifen! Das muss zu einem Fehler führen. Wir können das auf zwei Arten lösen:

1. Wir erstellen eine eigene Methode **`gibAktuelleFigur()`** in Moebel, die nichts tut (z.B. leer).
   In diesem Fall wird die Methode von Moebel durch die erbende Klasse überschrieben.
2. Wir erstellen einen reinen Methodenkopf und fügen ihm das Wort abstract hinzu.
   In diesem Fall wird unsere Klasse aber notwendig eine abstrakte Klasse, also eine Klasse, von der keine Objekte erzeugt werden können.

Andere Beispiele für die unter 2. beschriebene Variante finden wir viele in den o.a. Grafikklassen, z.B. Ellipse2D, von der selbst keine Objekte erzeugt werden, sondern nur von Ellipse2D.Double und Ellipse2D.Float.

Die verbleibende Klassendefinition von Tisch ist dann nur noch sehr kurz:

```
import java.awt.Shape;
import java.awt.geom.Ellipse2D;
/**
 * Ein Tisch, der manipuliert werden kann und sich ...
 */
public class Tisch extends Moebel
{
  /**
   * Konstruktor wie vorher:
   */
    public Tisch()
  {
    xPosition = 120;
    yPosition = 150;
    orientierung = 0;
    farbe = "rot";
    istSichtbar = false;
    breite = 120;
    tiefe  = 100;
  }

  /**
   * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
   */
  protected Shape gibAktuelleFigur()
  {
      Shape tisch = new Ellipse2D.Double(0 , 0, breite, tiefe);
      return  transformiere(tisch);
  }
}
```

Klassendiagramme und Modellierung

Beim Einfügen der neuen Klasse können wir gleich mit angeben, dass die Klasse Moebel eine abstrakte Klasse sein soll. Diese Möglichkeit wird uns als eine der Auswahlmöglichkeiten angeboten, wenn wir in BlueJ auf den Button Neue Klasse klicken.
Im Klassendiagramm wird die Klasse durch die über dem Namen eingefügte Zeile <<abstract>> gekennzeichnet. Das ist aber nicht die einzige Form der Veränderung nach dem Einfügen der neuen Klasse Moebel. Zusätzlich werden Pfeile von den konkreten Möbelklassen in das Diagramm eingetragen. Sie haben eine andere Gestaltung: Sie sind mit durchgezogener Linie gezeichnet und die Pfeilspitze besteht aus einem geschlossenen Dreieck. Damit kennzeichnet BlueJ den anderen Beziehungstyp:

- gestrichelte Linien und eine offene Pfeilspitze kennzeichnen eine Nutzer- Beziehung
- durchgezogene Linien und eine geschlossene Pfeilspitze kennzeichnen erbt – Beziehung

BlueJ arbeitet hier mit einem sehr kleinen Vorrat an Kennzeichnungen. In der OO hat sich für die Kennzeichnung von Klassenbeziehungen[1] inzwischen ein Quasistandard entwickelt (und wird noch weiter entwickelt), der mit ***UML*** **=** ***U***nified ***M***odeling ***L***anguage bezeichnet wird. UML kennt weitere Unterscheidungen von Beziehungstypen. Wir werden diese auch noch kennen lernen.

In einem der Standarwerke der UML von Bernd Oesterreicher „Objektorientierte Softwareentwicklung – Analyse und Design mit der ...“ heißt es:

Erdrückende Vielfalt?

Die Grundkonzepte objektorientierter Softwareentwicklungsmethodik sind ausgereift und bewähren sich in der Praxis. Andererseits bietet gerade die UML einen beachtlichen Detailreichtum und ist daher durchaus mit Bedacht anzuwenden.

Die Vielfalt der Beschreibungsmöglichkeiten kann sehr erdrückend wirken, ein tieferes Verständnis für die UML-Konstrukte in allen Facetten erfordert einigen Aufwand. In einer ersten Annäherung kann man sich deshalb auf die grundlegenden Elemente beschränken. Es bleiben damit gegebenenfalls zwar semantische Lücken in der Modellierung, dennoch kann die Arbeit auf diesem Niveau in der Praxis ausreichen. Ganz abgesehen davon wird vielerorts) überhaupt keine Systematik angewendet.

Übrigens erkennt BlueJ nicht selbständig, wenn eine Nutzerbeziehung entfernt wird. Man sollte daher per Hand die Pfeile von den konkreten Möbelklassen zur Klasse Leinwand entfernen. Hinzu kommende Beziehungen werden dagegen richtig erkannt, so dass der Pfeil von der abstrakten Moebelklasse zur Klasse Leinwand erscheinen müsste.

Aufgabe:

- Fügen Sie in das Projekt die neue Klasse Moebel ein und passen Sie die von Ihnen erstellten Klassen an.
- Versuchen Sie den Unterschied zwischen den beiden Beziehungstypen im Programmtext zu finden und ihn sprachlich zu beschreiben.

1 UML definiert nicht nur die Darstellung von Klassenbeziehungen, sondern auch andere Modellierungsaspekte wie z.B. Sequenzen; hier geht es zunächst jedoch nur um Klassenbeziehungen.

Kopplung und Kohäsion

Mit diesen beiden Begriffen werden Eigenschaften von Klassenentwürfen beschrieben, bei denen man die Qualität von Entwürfen beschreibt[2]. Mit einem dieser Fälle haben wir uns auf der ersten Seite dieses Abschnittes beschäftigt. Dort hieß es:

„... die Methode **`gibAktuelleFigur()`**. *Allerdings enthält sie in der vorliegenden Form noch einen Abschnitt, der in allen Klassen gleich ist, nämlich den Teil, in dem die Transformation des konkreten Falles eines Shape durchgeführt wird.*
Hier zeigt sich, dass die ursprüngliche Modellierung ungünstig war, da die Methode eigentlich zwei Aufgaben erfüllte:
3. Die Definition der konkreten Figur ...
4. Die Transformation dieser konkreten Figur ...“

Barnes/Kölling schreiben: „Der Begriff Kohäsion bezieht sich auf die Anzahl und Vielfalt der Aufgaben, für die eine einzelne Einheit in einer Anwendung zuständig ist... Idealerweise sollte eine Programmeinheit für genau eine in sich geschlossene Aufgabe zuständig sein...“

Nun, genau das haben wir bei unserem ersten Entwurf verletzt. Hier zeigt sich wieder, dass hinter dem beim Codeduplizieren entstandenen Gefühl, „das müsste raus“ mehr steckt (was übrigens nicht immer der Fall sein wird). Die Transformation für sich ist eine geschlossene Aufgabe, die für ein Shape – und das verwenden wir zur Darstellung unseres konkreten Möbels – zur Verfügung gestellt werden muss.

Der vorliegende erste Entwurf der Methode **`gibAktuelleFigur()`** verstieß gegen das Prinzip der Kohäsion.

Außerdem hatten wir im ersten Entwurf von **`transformiere(Shape shape)`** mit der Definition von zwei Affinen Transformationen zu viel gemacht. Siehe dazu der reduzierte Text der Methode auf Seite 8 unten. Die ursprüngliche Variante war also ungünstig. Das stellt aber keinen Verstoß gegen Entwurfsprinzipien dar.

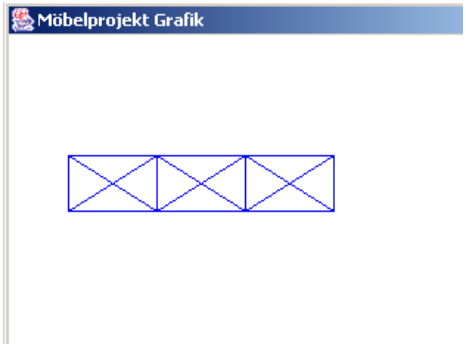
Entwurfkonzepte

Konzept zu Kopplung: (Barnes/Kölling; S.225) Der Begriff beschreibt den Grad der Abhängigkeit zwischen Klassen. Wir streben eine möglichst lose Kopplung an – also ein System ...

Konzept zu Kohäsion: (Barnes/Kölling; S.226) Der Begriff beschreibt, wie gut eine Programmeinheit eine logische Aufgabe oder Einheit abbildet. In einem System mit hoher Kohäsion ist jede Programmeinheit ... verantwortlich für genau eine wohldefinierte Aufgabe ...

2 Siehe dazu der Abschnitt 7.3 von Barnes / Kölling: Objektorientierte Programmierung in JAVA

Ein Beziehungstyp: Schrank und Schrankwand



Wenn wir in unser lieferbaren Möbel eine Schrankwand mit aufnehmen, dann sollten wir einmal Schrankelemente zeichnen können und daraus dann die Schrankwand zusammenstellen. Die Schrankwand könnte dann wie rechts zu sehen dargestellt werden.
Zunächst einmal wird man ein einzelnes Schrankelement zeichnen wollen. Daher wird man 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:

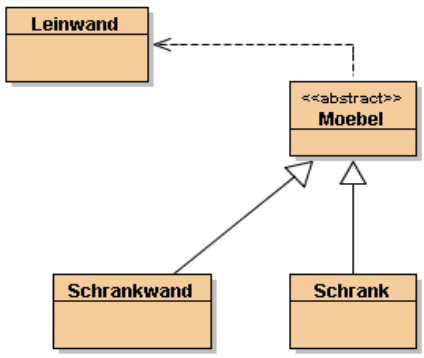
```
/**
 * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
 */
protected Shape gibAktuelleFigur()
{
    GeneralPath schrank = new GeneralPath();
    Shape rahmen = new Rectangle2D.Double(0, 0, breite/3, tiefe);
    Shape linie1 = new Line2D.Double(0, 0, breite/3, tiefe);
    Shape linie2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

3 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
{
  Schrank schrank1;
  Schrank schrank2;
  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 fügt die Shapes, die sie jeweils mit de Methode **`gibAktuelleFigur()`** von den drei Schrankobjekten abruft 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.

Ein intensiverer Blick auf diese Transformation wäre nun sinnvoll.