

Vererbung

Es ist sehr unbefriedigend, große Programmentextabschnitte 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 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 erben – abgeleiteten – Klasse enthalten nur noch genau die Methoden, in denen sie sich unterscheiden.

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 Methode, die nun in einer anderen Klasse steht, 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.

Viele andere Beispiele für die unter 2. beschriebene Variante finden wir in den von uns verwendeten 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);
        Rectangle2D umriss = tisch.getBounds2D();
        // nun noch transformieren:
        AffineTransform t = new AffineTransform();
        t.translate(xPosition, yPosition);
        t.rotate(Math.toRadians(orientierung), breite/2, tiefe/2);
        return t.createTransformedShape(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 Klassenbeziehungen¹ inzwischen ein Quasistandard entwickelt (und wird noch weiter entwickelt), der mit **UML** = **Unified Modeling Language** bezeichnet wird. UML kennt weitere Unterscheidungen von Beziehungstypen. Wir werden diese auch noch kennen lernen.

In einem der Standardwerke 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.

Kohäsion und Kopplung

Mit diesen beiden Begriffen werden Eigenschaften von Klassenentwürfen beschrieben, bei denen man die Qualität von Entwürfen beschreibt².

Bei unseren neuen Möbelklassen haben wir herausgefunden, dass sie sich allein in der Methode `gibAktuelleFigur()` unterscheiden. Das haben wir korrigiert, 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:

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 nun immer noch 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 t = new AffineTransform();
    t.translate(xPosition, yPosition);
    Rectangle2D umriss = shape.getBounds2D();
    t.rotate(Math.toRadians(orientierung),
            umriss.getX()+umriss.getWidth()/2,
            umriss.getY()+umriss.getHeight()/2);
    return t.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 `umriss` gelieferte Drehzentrum.

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 ist für sich 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.

Konzept hohe Kohäsion:

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 ...

Konzept lose Kopplung:

Der Begriff beschreibt den Grad der Abhängigkeit zwischen Klassen. Wir streben eine möglichst lose Kopplung an – also ein System ...

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