Kompositum nach Anforderungen

Die Kompositum – Klasse ist zunächst (praktisch) leer.
Da sie von Moebel erben soll, merkt man beim ersten Versuch, sie zu übersetzen, dass die Methode gibAktuelleFigur() implementiert werden muss. Wir machen das z.B. im Körper allein mit der Rückgabe von null.
Zu den Anforderungen, die wir in der Testklasse definieren, wird sicher zunächst gehören, dass man ein konkretes Moebel z.B. einen Tisch tisch1 und eine MoebelGruppe gruppe1 erzeugen kann und das man dieser das konkrete Moebel hinzufügen kann mit dem Aufruf einer Methode fuegeHinzu(tisch).
Dies wird verweigert, da MoebelGruppe diese nicht implementiert. Wir schreiben zunächst den Kopf und einen leeren Rumpf:

```
public void fuegeHinzu(Moebel moebel){}
```

Nun kann man den Test starten und er wird erfolgreich verlaufen, da wir an ihn ja keine inhaltlichen Anforderungen gestellt haben. Welches sind diese inhaltlichen Anforderungen?

- Wenn wir ein konkretes Moebel hinzugefügt haben, dann soll die MoebelGruppe dieses enthalten!

MoebelGruppe muss daher diese Methode bereitstellen. Wir gehen vor wie vorher beschrieben.

```
assertTrue(moebelGruppe.enthaelt(tisch1));
```

als Aufruf; wir implementieren die Methode. Der Rumpf sollte zunächst nur

```
return false;
```

enthalten. Folgerichtig führt der Test zu einem Fehler. Nun geht es an die inhaltliche Implementierung! Dazu brauchen wir zunächst einmal ein Attribut von MoebelGruppe, das die hinzugefügten Moebel speichert. Wir wählen dazu eine Implementierung der Schnittstelle **`List`**, die **`ArrayList`**. im Kopf der Klassendefinition also:

```
List moebelListe;
```

im Konstruktor:

```
moebelListe = new ArrayList();
```

... in der Methode fuegeHinzu(Moebel moebel) dann

```
moebelListe.add(moebel);
```

Nun können wir sinnvoll testen, da die ArrayList die Methode contains(Object elem) implementiert. Unser Test führt zu einem Erfolg.
Entsprechend gehen wir nun vor, um die Methode entferne zu implementieren.

```
        moebelGruppe.entferne(tisch1);
        assertFalse(moebelGruppe.enthaelt(tisch1));
```

Die Methode zeigt ein schönes Beispiel von wiederverwendeten Methoden:

```
    public Moebel entferne(Moebel moebel){
        if (enthaelt(moebel)){
            Moebel temp = moebel;
            moebelListe.remove(moebel);
            return temp;
        } else{
            return null;
        }
    }
```

Der Test erfolgt nun erfolgreich. Allerdings zeichnet unsere Möbelgruppe natürlich noch nichts, da unsere Methode gibAktuelleFigur() noch nichts sinnvolles macht.

Die Methode gibAktuelleFigur()

Wir erinnern uns:

- Sie berechnet das zu zeichnende Shape anhand der gegebenen Daten.
- Diese Methode muss wegen der Vererbung implementiert werden.

Eine naheliegende Lösung für unser Kompositumproblem ist es, an dieser Stelle einen GeneralPath zu benutzen. Ihm werden die eingefügten Möbelstücke hinzugefügt. Wir erinnern uns, dass die Methode append aber kein Moebel übergeben bekommen kann, sondern ein Shape. Die folgende Lösung arbeitet mit einem Iterator. Der Iterator stellt ein weiteres Entwurfsmuster dar, mit dem wir uns noch beschäftigen werden, das wir hier aber zunächst einfach verwenden. Die Methode lautet dann:

```
protected Shape gibAktuelleFigur(){
    GeneralPath figur = new GeneralPath();
    if (moebelListe.size()>0){
        for (Iterator it=moebelListe.iterator(); it.hasNext();){
            figur.append(((Moebel) it.next()).gibAktuelleFigur(),
                         false);
        }
    }
    return transformiere(figur);
}
```

Fertig?

Im Prinzip haben wir die Lösung, allerdings misslingt noch die Drehung, da unser Programm noch nicht die Mitte kennt, da seine breite und tiefe nicht stimmen. Wie lässt sich das lösen?
Eine Möglichkeit ist, das Programm in der Methode fuegeHinzu() – und dann natürlich auch in entferne() – so zu modifizieren, dass diese die Attribute breite und tiefe bestimmen bzw. anpassen. Allerdings haben wir dann das Problem, dass wir diese Werte aus den Werten der xPosition usw. der hinzugefügten Objekte bestimmen müssen.
Eine andere Lösung ist ein grundsätzliches Redesign: Wir entfernen die Methoden gibMitteX() und gibMitteY(), da ein Shape das von ihm eingenommene Rechteck kennt. Der GeneralPath ist ein Shape und jedes Shape muss die Methode getBounds() implementieren, die als Wert ein Rectangle – Objekt liefert, von dem wir nun mit den Methoden getX(), getWidth() usw. die entsprechenden Werte abfragen können.
Die Methode transformiere von Moebel wird modifiziert:

```
  protected Shape transformiere(Shape shape)
  {
      AffineTransform t = new AffineTransform();
      Rectangle r = shape.getBounds();
      t.rotate(Math.toRadians(orientierung),
               xPosition+r.getX()+r.getWidth()/2,
               yPosition+r.getY()+r.getHeight()/2);
      t.translate(xPosition, yPosition);
      return  t.createTransformedShape(shape);
  }
```

Die zweite Variante: Die Moebel selbst werden zu einer Gruppe verbunden

Das Problem der Drehung

Das eigentliche Problem beim Realisieren des Kompositum nach der zweiten Variante ist das Bestimmen des Drehzentrums. Dazu bieten sich mehrere Möglichkeiten an:

1. Wir geben das Drehzentrum „per Hand“ ein. Jede Möbelgruppe müsste also eine Methode definiereDrehZentrum(...) bereitstellen. Das kann durchaus eine sinnvolle Entscheidung sein, da das Drehzentrum nicht notwendig im zu berechnenden Zentrum des Shapes liegen muss.
2. Wir fordern, dass die Objekte beim Einfügen in die Gruppe schon ihre richtige Position innerhalb der Gruppe haben. Nachträgliche Änderungen sind dann nicht mehr möglich.
3. Nachträgliche Änderungen der Position innerhalb der Gruppe können nur über spezielle von der Gruppe bereitgestellte Methoden durchgeführt werden, also ohne die Objekte direkt zu verändern.
4. Die vierte Variante führt uns auf ein weiteres OO Entwurfsmuster, das Beobachtermuster. Bei jedem eingefügten Objekt wird sein Elternobjekt als Beobachter angemeldet. Findet an dem Objekt selbst eine Veränderung statt, muss es das Elternobjekt benachrichtigen, das dann seine Werte entsprechend anpassen kann.
5. Wer sagt eigentlich, dass man um den „richtigen“ Punkt drehen muss? Eine Drehung um die Ecke (xPosition;yPosition) und anschließende Verschiebung kann auch den Zweck erfüllen!

Bewertung

Betrachtet man den ggf. notwendigen Aufwand, fragt man sich, ob er sich lohnt. Diese Frage sollte aber in erster Linie an den Anforderungen geprüft werden: Ist es in unserer Architektenzeichnung überhaupt notwendig, Farben einzusetzen und wenn ja, sind diese bei Gruppen verschieden ?
Wird keine dieser Fragen mit ja beantwortet, dann sollte man sich für unsere erste Variante entscheiden.
Was steckt hinter ihrem Erfolg der ersten Variante ?
Betrachten wir die Methode **`append(Shape s, boolean connect)`** von GeneralPath, dann erkennt man sehr schnell, dass der GeneralPath selbst ein Kompositum darstellt:
„Füge Objekte zu Baumstrukturen zusammen, um Teile – Ganzes – Hierarchien zu repräsentieren. Das Kompositummuster ermöglicht es Klienten (Nutzer), sowohl einzelne Objekte, als auch Kompositionen (s.o.) von Objekten einheitlich zu behandeln.“ heißt es bei Gamma.
Genau das aber machen wir beim GeneralPath !

Wenn die zweite Variante von Kompositum realisiert ist, also wirklich mit den Möbeln:

An dieser Stelle erfolgt ein neues Redesign: Der Aufruf von transformiere steht immer noch an der falschen Stelle (Verstoss gegen Kohäsion). Er gehört in die Methode zeichne()! Da hier auch die Transformationen weiter gereicht werden, kann er nun an der richtigen Stelle stehen.