Entwurfsmuster

Entwurfsmuster sind ein Strukturierungsmittel, das hilft, nicht jedesmal das „Rad neu zu erfinden“. Erkennt man in einem Entwurf, den man macht, ein solches Entwurfsmuster, dann braucht man sich nicht noch einmal zu überlegen, wie „man das eigentlich macht“. Nach dem „aha, das ist wieder ein ...“ schaut man entweder in einer Lösung, die man schon einmal gemacht hat oder in der Literatur dazu nach. In der Regel wird man dann sogar größere Programmblöcke schon fertig haben.
Eines der Entwurfsmuster der OO Programmierung findet sich bereits in unserer Grundklasse für die grafische Darstellung, der Klasse Leinwand. Das dort verwendete Entwurfsmuster heißt Singleton und erfüllt den Zweck zu erzwingen, dass sich ein JAVA – Projekt nur genau eine Zeichenfläche beschaffen kann.

Singleton

Das Singleton ist ein Erzeugungsmuster. Bei Gamma finden wir eine Beschreibung des Zwecks:

> Zweck des Singleton – Musters
> Sichere ab, dass eine Klasse genau ein Exemplar besitzt
> und stelle einen globalen Zugriffspunkt darauf bereit.

An Beispielen nennt Gamma Druckerspooler, Dateisystem usw.
Die entscheidenden Teile unserer Klassendefinition sind:

```
public class Leinwand
{
  private static Leinwand leinwandSingleton;
  /**
   * Fabrikmethode, die eine Referenz auf das einzige Exemplar
   * dieser Klasse zurückliefert.
   */
  public static Leinwand gibLeinwand()
  {
    if (leinwandSingleton == null)
    {
      leinwandSingleton =
        new Leinwand("Möbelprojekt Grafik", 400, 400, Color.white);
    }
    leinwandSingleton.setzeSichtbarkeit(true);
    return leinwandSingleton;
  }
  ... ---- ...

  /**
   * Erzeuge eine Leinwand.
   */
  private Leinwand(String titel, int breite, int hoehe, Color grundfarbe)
  {
    fenster = new JFrame();
    ... --- ...
}
```

Klassenvariable gekennzeichnet durch *static*

Ein rein ***interner*** Konstruktor

Eine öffentlicheMethode, die diesen Konstruktor aufruft und ...

... durch die Abfrage am Anfang sicherstellt, ...

... dass er genau einmal aufgerufen werden kann.

Aufgabe:

- Untersuchen Sie das Kontextmenü der Klasse Leinwand:
  Wie würde ein Konstruktor angezeigt und was wird angezeigt ?

Das Kompositum-Muster im Raumplaner

Vergleicht man die Lösungen für eine Schrankwand und eine Sitzgruppe, ein weiteres Beispiel könnte eine Zusammenstellung von mehreren Konferenztischen sein, dann erkennt man, dass in der Methode **`gibAktuelleFigur()`** im Prinzip derselbe Code auftaucht. Arbeitet man dabei mit einem Iterator und abstrahiert in den Benennungen vom speziellen Beispiel der Schrankwand, Sitzgruppe oder Konferenztischgruppe, dann könnte dies eine Lösung für diese Methode in allen Fällen sein:

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

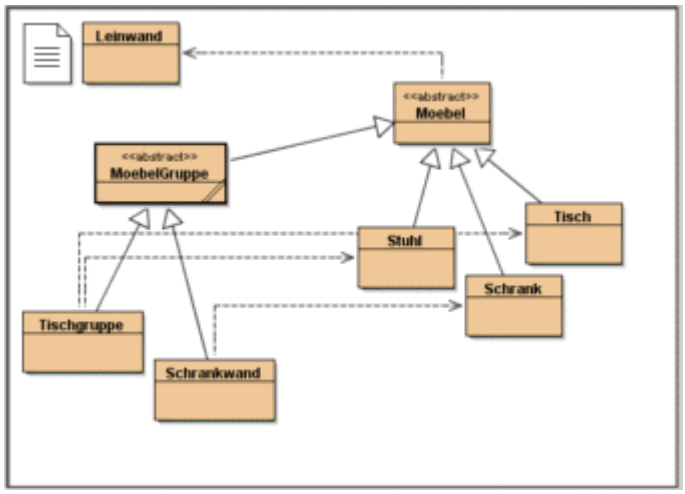
Der Sammlungstyp ist eine auf Moebel typisierte ArraList (z.B.). Der von ihr bereitgestellte Iterator gewährleistet den sequenziellen Zugriff auf alle zugehörigen Teilmöbel und die richtige Position und Orientierung wird durch die Transformation des GeneralPath-Objektes sichergestellt.

Eine Klasse MoebelGruppe

Schon an den gewählten Formulierungen für die Bezeichnung der zusammengefassten Objekte erkennt man, dass sie alle eine Gruppe repräsentieren und dass diese Gruppe damit ein Element unserer Modellierung sein sollte. Wir führen eine neue Klasse MoebelGruppe ein, von der wir die jeweiligen Klassen wie Schrankwand, Sitzgruppe oder Seminartischgruppe ableiten. Diese enthalten nur noch den Konstruktor, der dafür sorgt, dass jeweils die richtigen Teilobjekte mit der richtigen Position und Orientierung erzeugt und der Sammlung hinzu gefügt werden.

Die MoebelGruppe ist selbst auch ein Moebelobjekt

Die Möbelgruppe bringt bei unserer Lösung alle Eigenschaften von Moebel mit, ist also auch ein Moebel, d.h. erbt auch von dieser Klasse. Das Verschieben, Drehen und Ändern der Farbe funktioniert also genauso wie bei einem einzelnen Moebel-Objekt. Naheliegend ist nun zu fragen, ob in diesem Teil nicht auch Gemeinsamkeiten zu erkennen sind, die sich in die Klasse MoebelGruppe ausgliedern lassen.



Gruppen nicht nur bei Moebel-Klassen

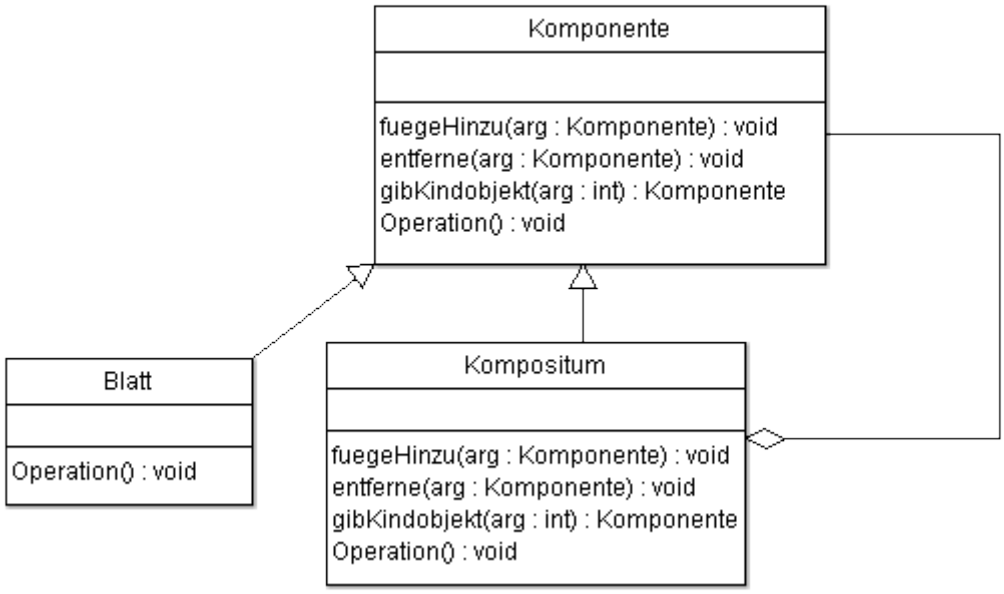
Wichtiger aber noch als diese Frage ist die Überlegung, dass eine Struktur, wie wir sie hier vorfinden, nicht nur bei Moebelklassen auftritt, sondern auch in anderen Anwendungsfällen. Das Bilden einer Gruppe von Objekten, die sich zusammen wieder wie eines der Objekte verhalten, ist also wiederkehrendes Entwurfsmuster, das Kompositum-muster.

Das Kompositum-Muster ist ein Strukturmuster

Gammas Beschreibung des Zwecks:

> Zweck des Kompositum – Musters
> Füge Objekte zu Baumstrukturen zusammen,
> um Teile – Ganzes – Hierarchien zu repräsentieren.
> Das Kompositum-Muster ermöglicht es Klienten (Nutzern), sowohl einzelne Objekte, als auch Kompositionen (s.o.) von Objekten einheitlich zu behandeln.

An Beispielen nennt Gamma gerade auch ein Grafiksystem. Kein Wunder also, dass wir hier eine passende Anwendung finden. Sein Klassendiagramm gibt eine mögliche Lösung vor, an der sich auch unsere Lösung orientiert. Hier ist jetzt das allgemeine Klassendiagramm für ein Kompositum dargestellt.



Eine wichtige Entwurfsentscheidung ist die Beantwortung der Frage, ob diese Methoden nur von der Gruppe, also dem Kompositum, bereitgestellt werden, wie wir das im o.a. Diagramm sehen. Alternativ können sie auch z.B. durch leere Implementationen der folgenden Art in der Klasse Moebel für alle Moebelklassen bereitgestellt werden:

Die drei allgemein auftretenden Methoden leer in Moebel implementieren

```
/*   fuegeHinzu
 * fügt ein Moebel der Gruppe hinzu.
 */
public void fuegeHinzu(Moebel moebel) {
}

/*   entferne
 * Entfernt das Moebel - Objekt.
 * Wenn es nicht existiert, passiert nichts.
 */
public void entferne(Moebel moebel) {
}
```

```
/*   gibKindObjekt
 * Gibt das Kindobjekt von der übergebenen Position zurück.
 * Ist die Position unzulässig, wird null zurückgegeben.
 */
public Moebel gibKindObjekt(int anPosition){
    return null;
}
```

Der Aufruf der Methode **`fuegeHinzu()`** führt also zu keiner weiteren Aktion. Die Methode **`gibKindObjekt()`** liefert dagegen null zurück, was aber ebenfalls sinnvoll ist, da es das Kennzeichen ist, dass kein Kindobjekt existiert.

Überschreiben

Die Klasse Gruppe muss diese Methoden allerdings auch inhaltlich implementieren. Das Bemerkenswerte ist nun, dass ein Gruppenobjekt „wissen“ muss, dass es nicht die Methode aus der Klasse Moebel verwenden darf, sondern dass es die zu ihr gehörende Methode verwenden muss. Man nennt diesen Vorgang Überschreiben (***override***). Die Methode **`fuegeHinzu()`** von Gruppe überschreibt die Methode **`fuegeHinzu()`** von Moebel, ersetzt sie also für Gruppenobjekte.
Der Text der überschreibenden Methode lautet:

```
/**
 * fuegt ein Element hinzu.
 * Dabei ist zu beachten, dass nicht nur das Element hinzugefuegt
 * wird, sondern auch seine Darstellung.
 */
public void fuegeHinzu(Moebel moebel)
{
    moebelListe.add(moebel);
    if (istSichtbar) moebel.zeige();
    aktualisiereUmriss();
    setzeGruppenTransformation();
}
```

Die hier vorgestellte Lösung arbeitet gezielt mit den Möglichkeiten der Verkettung von Affinen Transformationen, wie sie von der Klasse AffineTransform bereitgestellt werden. Die Erläuterung der beiden Methoden erfolgt weiter unten im Text.

Entsprechend **`fuegeHinzu(...)`** ist bei der Methode **`entferne(moebel)`** zu verfahren.

```
/**
 * entfernt ein Element.
 * Wenn es dies Objekt nicht gibt, passiert nichts.
 * Dabei ist zu beachten, dass nicht nur das Element entfernt wird,
 * sondern auch seine Darstellung.
 */
public void entferne(Moebel moebel)
{
    if (moebelListe.contains(moebel)){
        moebel.verberge();
        moebelListe.remove(moebel);
        aktualisiereUmriss();
        setzeGruppenTransformation();
    }
}
```

Hier sollte zunächst geprüft werden, ob das zu entfernende Objekt überhaupt in der

Gruppe enthalten ist. Auch bei der dritten Methode ist zu überprüfen, ob es ein Element an dieser Position auch wirklich gibt.

```
/**
 * gibt das Kindobjekt an der angegebenen Position zurueck.
 * Wenn es diese Position nicht gibt, wird null zurueckgegeben.
 */
public Moebel gibKindObjekt(int anPosition){
    if (anPosition>=0)
    if (anPosition<moebelListe.size())
    return moebelListe.get(anPosition);
    return null;
}
```

Aktualisiere Umriss

Wenn man nicht auf die sinnvoll modellierte Drehung der Moebel um ihr Zentrum verzichten will, muss man für eine Gruppe ihr Drehzentrum kennen. Dies lässt sich mit Hilfe der von allen Shapes bereitgestellten Methode zur Bestimmung ihres rechteckigen Umrisses erledigen.
Der Klasse Moebel wird dazu ein Attribut **`umriss`** hinzugefügt, das nun bei jedem Hinzufügen oder Entfernen eines Objektes aktualisiert werden muss. Die Methode dazu lautet ...

```
private void aktualisiereUmriss(){
    Shape figur=gibAktuelleFigur();
    umriss=figur.getBounds2D();
    breite=(int)umriss.getWidth();
    tiefe=(int)umriss.getHeight();
}
```

die dafür die Methode **`gibAktuelleFigur()`** aufruft

```
protected Shape gibAktuelleFigur()
{
    GeneralPath gruppe = new GeneralPath();
    for (Iterator<Moebel> it=moebelListe.iterator(); it.hasNext();)
    gruppe.append(it.next().gibAktuelleFigur(), false);
    return  gruppe;
}
```

Gruppentransformation

Außerdem muss man dafür sorgen, dass die Kindobjekte auch bewegt werden, wenn man die Gruppe bewegt. Eine mögliche Lösung wäre, diese Aufforderungen an die Kindobjekte weiter zu reichen. In unserem Fall geht das aber auch einfacher und zwar einfacher deswegen, weil wir uns dann nicht um die Zusammenhänge der verschiedenen, von verschiedenen Objekten ausgelösten geometrischen Operationen kümmern müssen: Wir verknüpfen einfach die Affinen Transformationen in der richtigen Weise.
Dazu müssen die Kindobjekte aber die Transformation der Gruppe kennen, müssen diese als Attribut speichern und von der Gruppe zugewiesen bekommen.

Dazu dient die Methode **`setzeGruppenTransformation()`**

```
/**
 * setzt die Gruppentransformation der Kindobjekte.
 */
protected void setzeGruppenTransformation(){
    if (moebelListe!=null){
```

```
            AffineTransform t = new AffineTransform();
            if (gruppenTransformation!=null)
                 t.concatenate(gruppenTransformation);
            t.translate(xPosition, yPosition);
            // relative Orientierung nach Definition:
            t.rotate(Math.toRadians(orientierung),
                   umriss.getX()+umriss.getWidth()/2,
                   umriss.getY()+umriss.getHeight()/2);
            // vorgegebene Rotation durch eine Gruppe:
            for (Iterator<Moebel> it=moebelListe.iterator(); it.hasNext();)
            it.next().setzeGruppenTransformation(t);
        }
    }
```

Sammlungsstruktur benutzen

Dass die Kompositumklasse Gruppe eine Sammlungsklasse nutzt, um die Kindobjekte zu verwalten, ist im oben angegebenen Text der Methoden schon zu erkennen. Es ist belanglos, welche dazu verwendet wird, allerdings sollte man sie in jedem Fall typisieren. Im Deklarationsteil der Klasse sollte es daher heißen:

```
protected ArrayList<Moebel> moebelListe;
protected Rectangle2D umriss;
```

Methoden aus Moebel überschreiben

Erstaunlich wenige Methoden aus Moebel müssen nun noch überschrieben werden, nämlich nur die Methoden zeichne() und loesche().

```
public void zeichne()
{
    setzeGruppenTransformation();
    for (Iterator<Moebel> it=moebelListe.iterator(); it.hasNext();)
    it.next().zeige();
}

public void loesche()
{
    for (Iterator<Moebel> it=moebelListe.iterator(); it.hasNext();)
    it.next().verberge();
}
```

Der Sinn ist, dass sie die entsprechenden Anforderungen an die Kindobjekte der Gruppe weiterreichen. Sie überschreiben also die entsprechenden Methoden aus der Klasse Moebel, die nun nur noch für die Blattobjekte der Hierarchie gelten.

Was steckt dahinter?

Warum ging das bei der Schrankwand mit dem GeneralPath so einfach zu realisieren, allerdings mit dem Mangel, dass die Kindobjekte nicht ihre Farbe behalten haben[1]?
Warum geht das so einfach mit den Affinen Transformationen?
Die Antwort ist sehr einfach: Beide erfüllen selbst das Kompositummuster!
Ein GeneralPath ist ein Shape, das aus Shapes bestehen kann und darunter können auch wieder -geschachtelt- weitere GeneralPath – Objekte sein.
Eine Affine Transformation kann verknüpft werden (concatenate) mit einer weiteren Affinen Transformation usw. und man erhält dann immer wieder eine Affine

1 Bei dieser Lösung kehrt sich der Mangel übrigens um: Nun ist zunächst einmal die Farbe der Gruppe belanglos, falls sie nicht gezielt gesetzt wird.

Transformation. Hier ist allerdings immer nur ein Kindobjekt zulässig.

Zwei Wege zum Ziel

In JAVA gibt es also zwei verschiedene Wege, die Möbelgrafikobjekte zu Gruppen zu verbinden:

- Man verbindet die Kindobjekte in der Gruppe zu einem `GeneralPath`.
  Das hat bei unserem Grafiksystem zur Folge, dass dieser dann notwendig alle Kindobjekte mit derselben Farbe zeichnet, da vor dem Zeichnen die Farbe festgelegt wird.
- Man gibt den Kindobjekten die Transformation der Gruppe mit.
  Nun können die Objekte verschiedene Farben haben, da sie getrennt von einander gezeichnet werden.

Den ersten beschriebenen Weg geht sinnvollerweise die Lösung für die Schrankwand, da in dem Fall unterschiedliche Farbgebungen für die Teilschränke sinnlos sind. Den zweiten Weg geht die hier beschriebene Kompositumlösung.
Dies liegt aber nicht am Kompositum-Muster selbst, sondern nur an unserer Lösung. Jede von beiden Möglichkeiten hat ihre Vor- und Nachteile. Für welche man sich entscheidet, sollte man nach den Anforderungen entscheiden, die das Problem stellt.

Aufgabe:

In der hier vorgestellten Lösung überschreiben Methoden andere Methoden der vererbenden Klasse. In diesen Sachzusammenhang gehört auch der begriff der Polymorphie.

- Stellen Sie schriftlich den Zusammenhang des Begriffes Polymorphie mit den Begriffen Vererbung und Überschreiben dar !
  Infos im BlueJ – Buch.
- Erläutern Sie ihn an einem Beispiel !

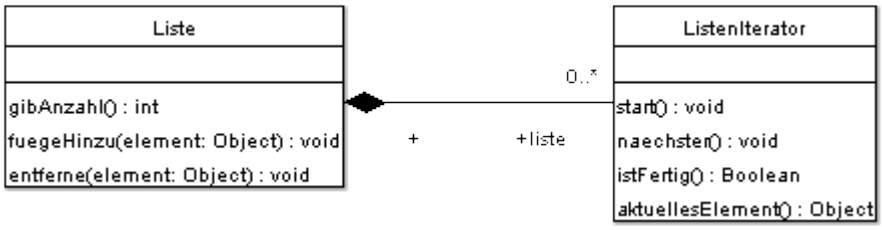
Das Iteratormuster

Wir haben in der vorigen Lösung mit einem Iterator gearbeitet. Ein Iterator ist selbst aber auch ein Entwurfsmuster!
Der Iterator ist ein „objektbasiertes Verhaltensmuster“. Bei Gamma finden wir als Beschreibung des Zwecks:

Zweck des Iterator – Musters
Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts,
ohne seine zugrundeliegende Repräsentation offen zu legen.

Beispiel

Eine ArrayList ist kein einfaches Objekt, sondern ein Objekt, das sich aus mehreren Objekten gleichen[1] Typs zusammensetzt. Nicht nur diese, alle Sammlungsklassen sollten eine Möglichkeit bieten, auf ihre Elemente zuzugreifen, ohne dass der Benutzer eine Vorstellung haben muss, wie das intern geschieht.
Dies führt zu einer Verallgemeinerung der Schnittstelle solcher Klassen und genau das liefert dieses Entwurfsmuster: Wir denken nur noch darüber nach, was ein Iterator tut, nicht mehr, wie er das macht.
Das setzt dann voraus, dass ein Exemplar einer solchen Sammlungsklasse ein Iterator – Objekt *hat*. Damit ist wiederum ein Beziehungstyp beschrieben. Da ein Iterator nicht unabhängig vom iterierten Objekt existieren kann, handelt es sich um eine Komposition.
Das Iteratormuster kann damit grafisch so beschrieben werden[2]:



Die Klasse ArrayList erbt über die in der JAVA Klassenbibliothek dargestellte Kette

```
java.lang.Object
      → java.util.AbstractCollection
            → java.util.AbstractList
                  → java.util.ArrayList
```

von der abstrakten Klasse (oder implementiert sie selbst) die Methode `iterator()`, die einen Iterator bereitstellt.
Iterator ist dabei ein interface, das von der zu der Klasse ArrayList gehörenden konkreten Iteratorklasse implementiert wird.

1 Dass es sich bei den Elementen in einer Liste um denselben Typ handeln muss, ist nicht in allen Programmiersprachen so. Auch bei JAVA gilt diese Forderung nur im strengen Sinne. Tatsächlich werden die Elemente – egal, was sie sind – als Exemplare der Klasse Object behandelt und sind durch den cast durchaus als Exemplare unterschiedlicher Klassen wiederherstellbar.
2 Vorlage: Gamma e.a.

Das Beobachtermuster

Will man weitere Elemente in ein Kompositum einbauen, dann wird man auf das Problem aufmerksam, dass das Kompositum die Attribute breite und tiefe hat, die – so ohne weiteres – für das Kompositum keine Bedeutung haben. Für viele Methoden, wie z.B. **`bewegeHorinzontal()`** ist der konkrete Wert dieser beiden Attribute bedeutungslos. Wichtig sind sie aber für die Methode **`dreheAuf(...)`**, die über die Methoden **`gibMitteX()`** und **`gibMitteY()`** auf die beiden Attributwerte zugreift. Der Sinn ist die Definition des Mittelpunktes der Figur als Drehzentrum. In der hier vorgestellten Lösung ist das teilweise mit dem Attribut umriss gelöst.
Im Unterricht haben Schülerinnen und Schüler ebenfalls an einer Lösung für das Hinzufügen von Möbelstücken gearbeitet, bei der automatisch die Werte so angepasst werden, dass anschließend eine Drehung um das Zentrum der Gruppe erfolgen kann.[1]
Zunächst haben die Schülerinnen und Schüler diskutiert, an welcher Stelle dieser Vorgang stattfinden soll. Möglich wäre z.B. eine Abfrage in **`zeige()`**, bei der alle Kindobjekte nach ihrer derzeitigen Position und Größe gefragt werden.
Schon hier wird deutlich, dass für die Bestimmung von **`breite`** und **`tiefe`** der Gruppe nicht die entsprechenden Attribute der Kindobjekte benötigt werden, sondern auch die Attribute **`xPosition`** und **`yPosition`**. Diese Attribute definieren bei unserer Konstruktion die relative Position der Kindobjekte in der Gruppe und haben damit auch Einfluss auf deren breite und tiefe.
Eine Schülergruppe ist nun sehr schnell darauf aufmerksam geworden, dass die Kindobjekte Methoden zur Veränderung der Attributwerte bereitstellen, nämlich **`bewegeHorinzontal()`** und **`bewegeVertikal()`**. Die Gruppe hat sich die Frage gestellt, wie denn nun die Gruppe davon erfährt, dass eine der Methoden für eines ihrer Kindobjekte aufgerufen worden ist. Man braucht also etwas, das die Objekte beobachtet, ob sich ein für die Gruppe relevante Eigenschaft eines Kindobjektes geändert hat. Diese Diskussion führt auf das Beobachtermuster.
Gamma schreibt zum Beobachtermuster:

Zweck

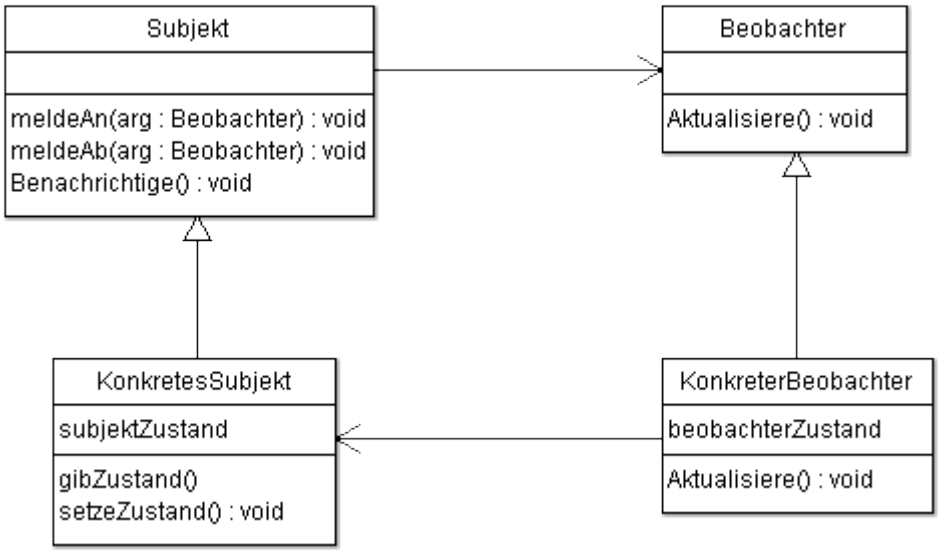
> Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Motivation

> Teilt man ein System in eine Menge von interagierenden Klassen auf, so ergibt sich häufig der Nebeneffekt, dass die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden muss. Man möchte diese Konsistenz üblicherweise nicht dadurch sicherstellen, dass man die Klassen eng miteinander koppelt, weil dies ihre Wiederverwendbarkeit einschränkt.

1 Selbstverständlich ist eine automatische Lösung nicht die einzige Lösung des Problems. s.o.

Die grafische Darstellung sieht folgendermaßen aus[1]:



Interessant ist ebenfalls die Diskussion gewesen, an welcher Stelle der Beobachter erzeugt werden soll und an welcher Stelle sich ein *Subjekt* beim Beobachter anmelden sollte. Die Gruppe ist zu dem Ergebnis gekommen, dass das Erzeugen zwar beim Erzeugen der Gruppe erfolgen muss, das Anmelden aber nicht beim Erzeugen des Objektes selbst, sondern erst dann, wenn es der Gruppe hinzugefügt wird. Ein einzelnes Möbel braucht nicht beobachtet zu werden[2].
Wird nun eine der relevanten Methoden aufgerufen, wird der Beobachter benachrichtigt und der informiert die Gruppe über die Veränderung.

1 Vorlage: Gamma e.a.
2 Diese Frage könnte sich allerdings dann stellen, wenn z.B. jedes Möbel automatisch auf zulässige Werte untersucht werden soll. Dann könnte eine Verschiebung z.B. an eine Klasse Raum weiter gemeldet werden, um Konsistenz der Werte zu überprüfen.