# 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:

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.

```
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();
    ... --- ...
}
```

## *Aufgabe:*

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

**Kompositummuster**

Das Kompositummuster ist ein Strukturmuster. Es passt gut zu unseren Beispielen mit der Schrankwand und der Sitzgruppe. Gammas Beschreibung des Zwecks:

> Zweck des Kompositum – Musters
> 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.

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.

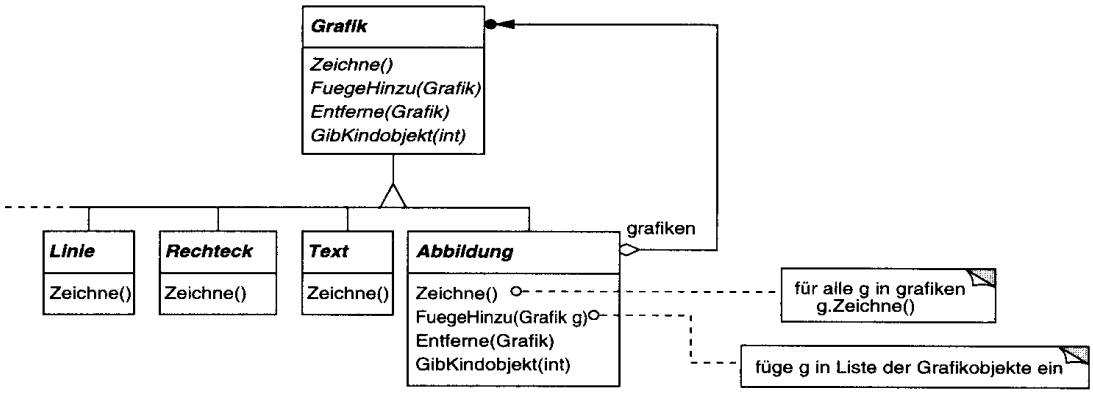



*Abbildung 4.32*

Wir werden aber einige Änderungen berücksichtigen, auf die ich anschließend noch eingehen möchte. Zunächst aber beschäftigen wir uns mit den eigentlichen Kompositum-methoden. Wir bauen sie in die abstrakte Klasse Moebel ein, damit auch die Blattobjekte diese bereitstellen, obwohl sie für diese leer sind:

```
/*
 * Haengt ein Moebel am Ende der Liste an.
 */
public void fuegeHinzu(Moebel moebel){};

/*
 * Entfernt das Moebel an der Position i der Liste.
 * Wenn i unzulaessig ist, passiert nichts.
 */
public void entferne(int i){};

/*
 * Entfernt ein Moebel aus der Liste.
 * Wenn es nicht existiert, passiert nichts.
 */
public void entferne(Moebel moebel){};

/*
 * 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 als Kennzeichen, dass kein Kindobjekt existiert.

**Überschreiben**

Die Klasse Gruppe allerdings muss diese Methoden 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. Der Text der überschreibenden Methode lautet:

```
  public void fuegeHinzu(Moebel moebel){
      liste.add(moebel);
      moebel.setzeElternTransformation(gibTransform());
  }
```

**Der Datentyp Liste**

Dabei wird deutlich, dass wir natürlich eine besondere Datenstruktur zum Speichern der Moebelobjekte brauchen, die zu der Gruppe gehören. Wie dem Namen zu entnehmen ist, handelt es sich um eine Liste. Sehen wir in der JAVA Klassenbibliothek nach, dann finden wir, dass `List` ein interface ist. Die konkrete Implementation von Liste, die wir hier verwenden, ist dem Kopf der Klasse zu entnehmen:

```
  private ArrayList liste;
```

und im Konstruktor heißt es:

```
    liste = new ArrayList();
```

`add` ist also eine Methode der Klasse ArrayList. Sie fügt am Ende der Liste ein Element an. Weitere Methoden sind:

```
  /*
   * Prüft, ob ein Index zulässig ist.
   */
  private boolean zulaessig(int index){
      if ((index>=0) && (index < liste.size())){return true;} else {return
false;}
  }

  /*
   * Entfernt das Moebel an der Position i der Liste.
   * Wenn i unzulaessig ist, passiert nichts.
   */
  public void entferne(int i){
      if (zulaessig(i)){
          liste.remove(i);
      }
  }
```

Auf die Elemente einer ArrayList kann man also über einen Index zugreifen, der von 0 bis zur akuellen Länge (minus 1!) geht. Die Hilfsmethode `zulaessig(i)` verhindert Zugriffsversuche auf nicht existierende Objekte. Dies nutzen wir auch bei ...

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

```
        }
        else{
            return null;
        }
```

... während bei der Methode ...

```
    /*
     * Entfernt ein Moebel aus der Liste.
     * Wenn es nicht existiert, passiert nichts.
     */
    public void entferne(Moebel moebel){
        if (liste.contains(moebel)){
            liste.remove(liste.indexOf(moebel));
            moebel.setzeElternTransformation(new AffineTransform());
        }
    }
```

... ein unzulässiger Zugriff durch einen Aufruf der Methode `contains(...)` verhindert wird.

**Nun zu den anderen Methoden:**

Ändern wir in unserem Projekt nur die `zeichne()` - Methode in der Abbildungsklasse, also unserer Möbelgruppe, dann bekommen wir nicht die gewünschte Lösung. Ein möglicher Ansatz wäre, nun alle entsprechenden Methoden – wie beispielsweise die Methode `bewegeHorinzontal()` – so zu modifizieren, dass sie die entsprechenden Methoden für die Kindobjekte aufrufen und dabei die Parameter an diese Kindobjekte weiterreichen. Das geht zwar, schlägt aber bei `dreheAuf(...)` fehl, da diese Methode sich das Drehzentrum selbst holt und das Kindobjekt keineswegs um das Zentrum der Gruppe dreht.
Betrachtet man nun genauer, wie JAVA zeichnet, dann fallen wir mit einem solchen Ansatz hinter das zurück, was wir bei unserem Ansatz zur Schrankwand durch das Verketten der Abbildungen schon viel besser gelöst haben.

Wir werden nun auch beim Kompositum diesen Ansatz gehen, allerdings anders herum. Dazu soll das Objekt, welches bei dieser Lösung das Zeichnen konkret ausführt – also das Kindobjekt – die verknüpfte Abbildung erzeugen und muss daher neben seiner eigenen Transformation, durch welche die Lage in der Gruppe beschrieben wird und die Transformation seines Elternobjektes kennen. Die Elternobjekte müssen dazu jeweils ihre Transformation an die Kindobjekte weiterreichen.
Dies hat aber ein kleines redesign von Moebel zur Folge, da für die von ihr erbende Klasse Gruppe nun eine – zumindest protected – Methode bereitstehen muss, welche die Transformation bereitstellt.
Weitere Änderungen durch Überschreiben der entsprechenden Methoden von Moebel sind dort notwendig, wo auch Attribute der Kindobjekte gesetzt werden müssen.
Beispielsweise muss die Methode zeige()...

```
    /*
     * zeige(): Aendert auch die Sichtbarkeit der Kindobjekte
     */
    public void zeige(){
        for (Iterator it = liste.iterator(); it.hasNext();){
            ((Moebel) it.next()).zeige();
        }
    }
```

... auch die Sichtbarkeit der Kindobjekte ändern. Ähnliches gilt auch für die Methode `aendereFarbe(...)`.
Das war's im Prinzip.

### *Zwei Wege zum Ziel*

In JAVA gibt es 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 unsere Lösung für die Schrankwand. Den zweiten geht unsere oben 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.

#### Eine Idee

Interessant wäre es, wenn Blattobjekte des Kompositum – Baumes, also Objekte, die selbst keine Gruppen darstellen, die Methode **`fuegeHinzu(...)`** so implementieren, dass sie automatisch eine neue Gruppe anlegen, die nun das ursprüngliche Objekt und das hinzugefügte enthält und diese Gruppe an der vorigen Stelle des Objektes in den Kompositum – Baum einfügen.

## *Aufgabe:*

- 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 !

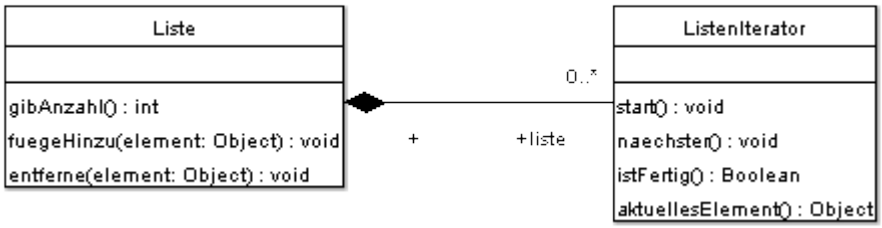
### *Iterator*

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**

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:




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.

### *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.
Im Unterricht haben die Schülerinnen und Schüler 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.[2]
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.

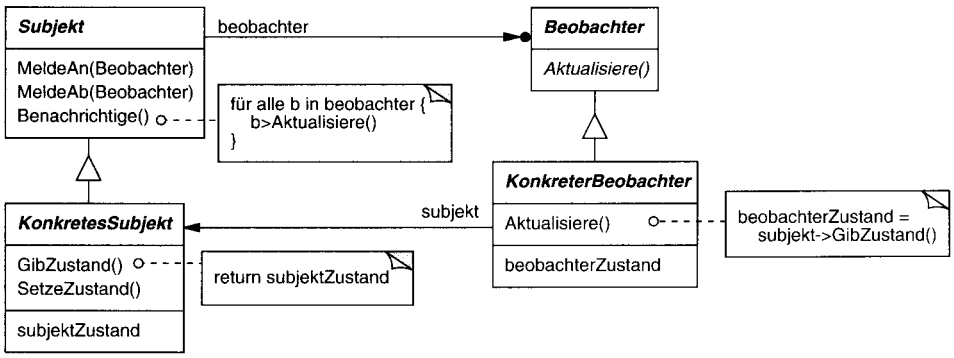
**Motivation**

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.

2 Selbstverständlich ist eine automatische Lösung nicht die einzige Lösung des Problems. Es könnte auch sein, dass die Klasse Gruppe eine Methode definiereZentrum(xZentrum, yZentrum) bereitstellt, die einen schreibenden Zugriff z.B. auf die Attribute breite und tiefe (oder neu definierte) ermöglicht. Macht man das mit breite und tiefe, dann brauchen gibMitteX und gibMitteY nicht verändert zu werden. Im anderen Fall muss man sie überschreiben.

Die grafische Darstellung sieht folgendermaßen aus:

Struktur des Beobachtermusters




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[3].
Wird nun eine der relevanten Methoden aufgerufen, wird der Beobachter benachrichtigt und der informiert die Gruppe über die Veränderung.

**Anmerkung zu einer Lösung**

Die tatsächliche Lösung ist keineswegs trivial, insbesondere wird sie dann aufwändig, wenn man noch berücksichtigt, dass die Kindobjekte gedreht sein können, was insbesondere bei Winkeln schwierig wird, die von 0°, 90°, 180° und 270° verschieden sind.

3 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.