# 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 umgesetzt 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 Zeichenflaeche. Das dort verwendete Entwurfsmuster heißt Singleton und erfüllt den Zweck zu erzwingen, dass sich ein Projekt nur genau eine Zeichenfläche beschaffen kann.

## *Singleton*

Das Singleton ist ein Erzeugungsmuster. Bei Gamma[1] 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.

## *Realisierung bei Java*

Bei Java reicht es, den Konstruktor [wenn es mehrere gibt alle] der Klasse auf **`private`** zu setzen und eine Klassenmethode zu definieren, welche die einzige Instanz der Klasse zurückgibt.
Diese Methode muss zunächst prüfen, ob es bereits eine Instanz der Klasse gibt. Beim ersten Aufruf muss sie daher eine Instanz erzeugen. Das kann die Methode, da sie Teil der Klassendefinition ist und daher den privaten Konstruktor aufrufen kann.
Bei allen nachfolgenden Aufrufen wird einfach die existierende Instanz zurückgegeben. Das bedeutet aber auch, dass die Klasse in einer Klassenvariable [statisches Attribut, Kennzeichnung **`static`**] diese Instanz halten muss.

## *Probleme bei Python*

Wegen der fehlenden Möglichkeit, die Sichtbarkeit zu definieren, ist die Umsetzung bei Python etwas schwieriger als bei Java. Daher findet man für Python eine andere Möglichkeit das Ziel zu erreichen, nämlich mit einer Klasse Borg. Dabei wird zwar nicht verhindert, dass es mehrere Objekte des Typs gibt, aber erzwungen, dass alle Instanzen sich dieselben Attribute teilen.
Grundsätzlich kann man aber auch genauso vorgehen wie bei Java und sogar dadurch das direkte Aufrufen des Konstruktors verhindern, dass man in dem Fall einen Fehler auslöst.

---

1 Ralph Johnson / Erich Gamma / Richard Helm / John Vlissides : **Entwurfsmuster** Elemente wiederverwendbarer objektorientierter Software
ISBN: 978-3-8273-2199-2 [addison-wesley]
Alternative: Entwurfsmuster von Kopf bis Fuß; O'Reilly; ISBN 978-3-89721-421-7

### *In der Klasse Zeichenflaeche*

Der relevante Abschnitt der Klassendefinition:

```
class Zeichenflaeche(wx.Panel):

    __zeichenflaeche = None

    @staticmethod
    def GibZeichenflaeche(parent=None):
        if Zeichenflaeche.__zeichenflaeche == None:
            if parent==None:
                return None
            else:
                Zeichenflaeche.__zeichenflaeche = Zeichenflaeche(parent)
        return Zeichenflaeche.__zeichenflaeche

    def __init__(self, parent):
        if Zeichenflaeche.__zeichenflaeche != None:
            raise Exception('Konstruktor nicht direkt aufrufen')
        wx.Panel.__init__(self, parent, -1)
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        if USE_BUFFER:
            self.Bind(wx.EVT_SIZE, self.OnSize)
        self.objekte = []
```

Eine Klassenmethode (statische M.), die diesen Konstruktor aufruft und ...

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

dass er genau einmal aufgerufen werden kann.

Fehler auslösen, wenn der Konstruktor nicht von der Methode aufgerufen wird.

Die Abfrage nach dem **`parent`** sorgt dafür, dass beim erstenmal, wenn die Methode vom Grafikfensterobjekt aufgerufen wird, der zugeordnete Frame definiert wird. Die anderen Aufrufe erfolgen ohne Parameter. Aber selbst wennerneut ein Frame übergeben werden sollte, kann der Frame nicht ausgetauscht werden, da dann ein Fehler ausgelöst wird.

# Das Kompositum-Muster im Raumplaner

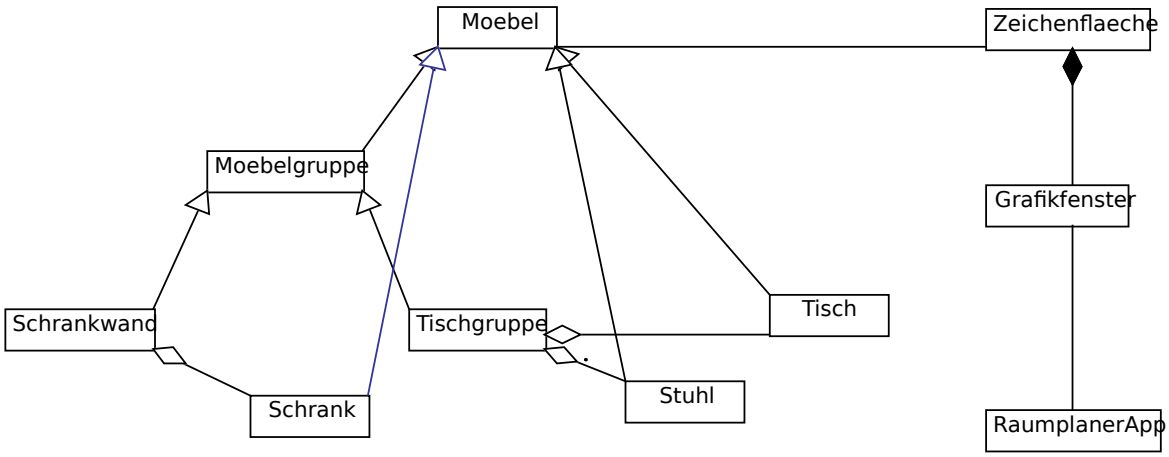
Vergleicht man die Lösungen für eine Schrankwand, eine Tischgruppe oder weitere Beispiele, dann erkennt man, dass in der Methode **`GibFigur()`** im Prinzip immer derselbe Code auftaucht. Arbeitet man, wie es sein sollte, mit einer Sammlungsklasse, die einen Iterator bereitstellt, dann kann man in den Benennungen vom speziellen Beispiel der Schrankwand, Sitzgruppe oder Konferenztischgruppe abstrahieren. Es gibt also eine Lösung für diese Methode, die auf alle diese Anwendungsfälle passt.

### *Polymorpher Zugriff*

Die Sammlungsklasse kann eine auf beliebige **`Moebel`**[1] anwendbare Liste sein. Der von ihr bereitgestellte Iterator gewährleistet den sequenziellen Zugriff auf alle zugehörigen Teilmöbel, unabhängig von der Zugehörigkeit zu einer bestimmten Möbelklasse und die richtige Position und Orientierung wird durch die Transformation des **`GraphicsPath`**-Objektes sichergestellt. Um den polymorphen Zugriff zu gewährleisten, muss jedes der beteiligten Objekte die Methoden (Java: mit derselben Signatur) bereitstellen. Der Aufruf kann dadurch unabhängig von der Realisierung der Methode für das spezielle Objekt immer in derselben Weise erfolgen.

### *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 hinzugefügt werden.




Ein Klassendiagramm von ArgoUML[2] ohne Methoden und Attribute

1 Das kennzeichnet auch das Vorgehen bei Java: Dort verwendet man als Typ der Sammlungsklasse beispielsweise eine ArrayList<Moebel>. Wir finden hier einen leicht verständlichen Anwendungsfall von Polymorphie. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen zwar unterschiedlich implementiert ist, wegen der gleichen Signatur unabhängig vom konkreten Typ aber gleich aufgerufen werden kann.

2 Klassendiagramm von Dia im Anhang.

### *Die Moebelgruppe ist selbst auch ein Moebelobjekt*

Die Möbelgruppe bringt bei dieser Lösung alle Eigenschaften von **`Moebel`** mit, ist auch ein **`Moebel`**, sie erbt also auch von dieser Klasse. Das Verschieben, Drehen und Ändern der Farbe funktioniert also genauso wie bei einem einzelnen **`Moebel`**-Objekt, solange eine dieser Methoden nicht in der Klassendefinition der Gruppe überschrieben wird. Naheliegend ist nun zu fragen, ob im individuellen 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 Möbelklassen auftritt, sondern auch in anderen Anwendungsfällen. Das Bilden einer Gruppe von Objekten, die sich zusammen wieder wie eines der Objekte verhalten, ist ein wiederkehrendes Entwurfsmuster, das Kompositum-Muster.

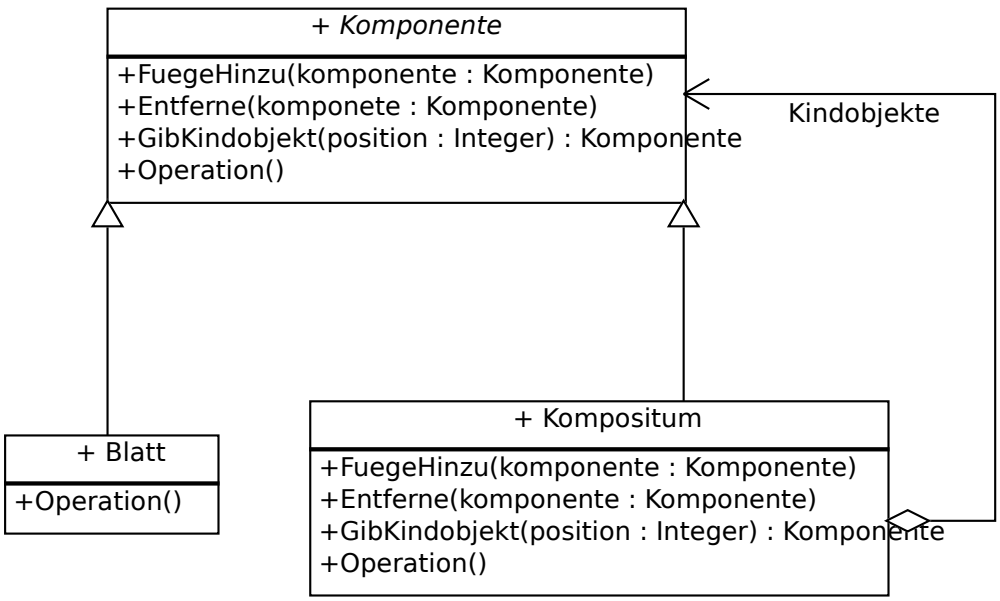
### *Das Kompositum-Muster ist ein Strukturmuster*

Beschreibung des Zwecks[1]:

> Zweck des Kompositum – Musters
> Füge Objekte zu Baumstrukturen zusammen,
> um Teil–Ganzes–Hierarchien zu repräsentieren.
> Das Kompositum-Muster ermöglicht es Klienten (Nutzern), sowohl einzelne Objekte, als auch Komposita 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.

**Ein Klassendiagramm für ein Kompositum[2]:**



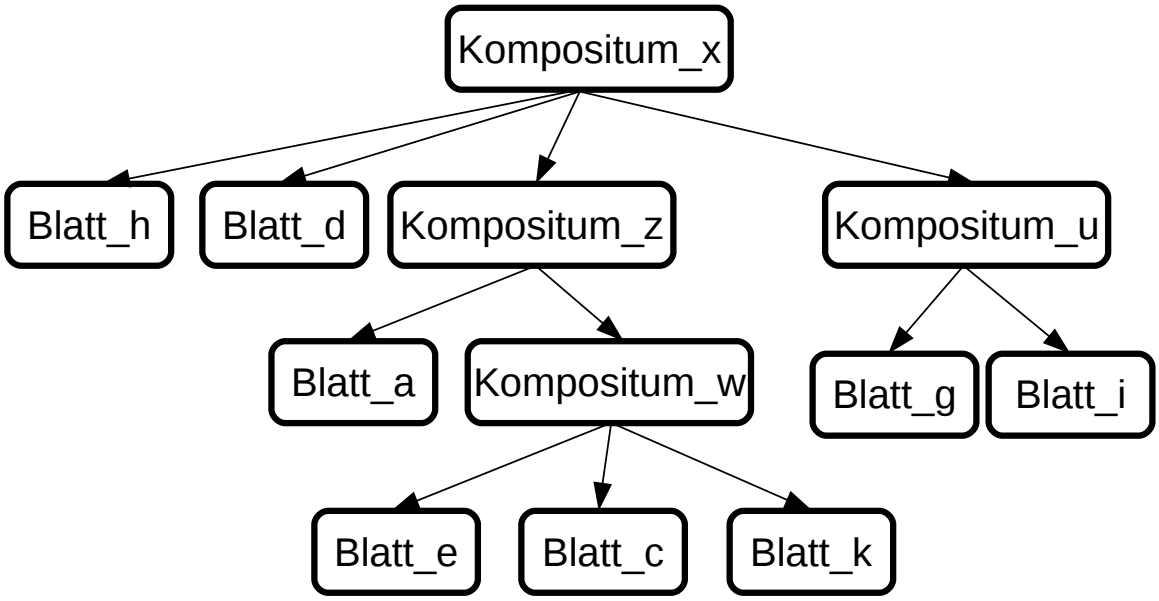

1 Nach Gamma e.a.: Entwurfsmuster
2 Ebenfalls nach Gamma e.a.

***Aufgabe:*** Analysieren Sie die im Modell angegebenen Beziehungen.

### *Teil-Ganzes-Hierarchie?*

Mit *Teil-Ganzes-Beziehungen* bezeichnet man die beiden Beziehungstypen Aggregation und Komposition. Der zusätzliche Begriff Hierarchie beschreibt, dass diese Beziehung auch rekursiv geschachtelt auftreten kann. Ein Kompositum kann neben Blattklassen also auch wieder als Komponente ein Kompositum enthalten, das wiederum aus …

**Baumdiagramm [Hierarchie] für ein Kompositum**




### *Eine eigene abstrakte Komponentenklasse[1]?*

Es ist keinesfalls zwingend, eine separate, gegebenenfalls abstrakte Klasse für die Komponente zu realisieren, also beispielsweise eine Klasse **`MoeblierungsKomponente`**. Sinnvoll ist auch eine Lösung, bei der eine Klasse **`MoebelGruppe`** das Kompositum-Muster implementiert oder die Klasse **`Moebel`** selbst.
Beachten Sie auch den Hinweis am Schluss zu anderen Varianten der Lösung.

### *Die Implementation der Methoden*

Eine vorgegebene Komponentenklasse wäre sehr einfach. Sie müsste eine Liste nutzen, um die Kindobjekte zu verwalten. Die hat hier den Namen **`__komponenten`**. Fügt man dem Projekt nicht solch eine Klasse hinzu, muss man die Methoden in den entsprechenden Klassen implementieren, also beispielsweise in **`MoebelGruppe`**, diesen Weg geht die hier anschließend dargestellte Lösung.

```
class Komponente():
    def __init__(self,):
        self.__komponenten = []
```

1 Bei Gamma entsprechend Java eine Schnittstelle [Interface]

```
    def FuegeHinzu(self, komponente):
        self.__komponenten.append(komponente)

    def Entferne(self, moebel):
        if komponente in self.__komponenten:
            self.__komponenten.remove(komponente)

    def GibKindObjekt(self, anPosition):
        if (anPosition<=len(self.__komponenten)) and (anPosition>0):
            return self.__komponenten[anPosition]
        else:
            return None
```

Bei der zweiten Methode muss geprüft werden, ob es das Element überhaupt in der Liste gibt. Bei der dritten Methode ist zu überprüfen, ob es ein Element an dieser Position auch wirklich gibt, daher die Abfrage, die hier im Misserfolgsfall zu der Rückgabe None führt. Es ist immer eine schwierige Entscheidung, wie man auf einen fehlerhaften Zugriffsversuch reagiert. So, wie es hier gelöst ist, muss der Anwender überprüfen, ob **`None`** zurückgegeben wurde. Es kann auch sinnvoll sein, den Zugriff nicht in dieser Klasse abzuprüfen und es dem Benutzer zu überlassen, den **`IndexError`** abzufangen.
Zumindest abstrakt angelegt werden müssten hier die Methoden, die das gemeinsame Verhalten von Blatt- und Kompositum-Objekten beschreiben.

### *Realisierung in Moebelgruppe*

Die Klasse **`Moebelgruppe`** realisiert diese Methoden sinnvoll wie oben angegeben und stellt das Attribut **`__komponenten`** bereit, das hier mit **`__moebelListe`** bezeichnet ist.

```
class MoebelGruppe(Moebel):
    """Klasse MoebelGruppe
    Realisiert Möbelgruppen nach dem Kompositummuster"""

    def __init__(self,
                 xPos=100,
                 yPos=200,
                 winkel=0,
                 farbe='red'):
        """Konstruktor mit vordefinierten Parametern
        für die Position, Winkel und Farbe;
        breite und tiefe sind zu Anfang 0,
        die Gruppe ist immer sichtbar"""
        self.__moebelListe=[]
        Moebel.__init__(self, xPos, yPos, 0, 0, winkel, farbe, True)

    def FuegeHinzu(self, moebel):
        if self.GibSichtbar(): moebel.Zeige()
        else: moebel.Verberge()
        self.__moebelListe.append(moebel)
        self.AktualisiereAbmessungen()

    def Entferne(self, moebel):
        if moebel in self.__moebelListe:
            moebel.Verberge()
            self.__moebelListe.remove(moebel)
            moebel.SetzeGruppe(None)
            self.AktualisiereAbmessungen()
```

```
    def GibKindobjekt(self, position):
        """gibt für eine MoebelGruppe das Kindobjekt zur Position zurück"""
        if position<len(self.__moebelListe):
            if position>=0:
                return self.__moebelListe[position]
        return None

    def GibFigur(self):
        """definiert die zu zeichnende Figur"""
        path = self.GibZeichenPfad()
        for moebel in self.__moebelListe:
            path.AddPath(moebel.GibFigur())
        return self.Transformiere(path)
```

Die Methode **`GibFigur()`** holt sich von allen Kindobjekten die Figur und fügt sie zu einer eigenen zusammen.
Zusätzlich wird noch eine Methode **`AktualisiereAbmessungen()`** zum Aktualisieren der Attribute **`breite`** und **`tiefe`** in **`Moebel`** eingefügt.

### *Wozu aktualisiert man Breite und Tiefe?*

Wenn man nicht auf die sinnvoll modellierte Drehung der Möbel um ihr Zentrum verzichten will, muss eine Gruppe ihr Drehzentrum kennen und dafür benötigt die Gruppe ihre Abmessungen. Dies lässt sich mit Hilfe der von allen Path-Objekten bereitgestellten Methode **`GetBox()`** zur Bestimmung ihres rechteckigen Umrisses erledigen. Sie liefert ein 4-Tupel mit den aktuellen Werten für die Position und **`breite`** und **`tiefe`**. Mit diesen Werten gelingt die Transformation dann richtig.

### *Weitere Ergänzungen*

Die Methoden zum Bewegen von **`Moebel`** können weiter verwendet werden. Anders ist es bei den folgenden Methoden:

```
    def AendereFarbe(self, neueFarbe):
        """Veraendernde Methode fuer die Farbe"""
        self.Verberge()
        for moebel in self.__moebelListe:
            moebel.AendereFarbe(neueFarbe)
        self.Zeige()

    def Verberge(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert False"""
        super(Moebelgruppe, self).Verberge(True)
        for moebel in self.__moebelListe:
            moebel.Verberge()

    def Zeige(self):
        """Veraendernde Methode fuer die Sichtbarkeit mit Wert True"""
        super(Moebelgruppe, self).Zeige(True)
        for moebel in self.__moebelListe:
            moebel.Zeige()
```

### *Was steckt hinter dieser einfachen Lösung*

- Warum geht das bei der Moebelgruppe mit dem GraphicsPath so einfach zu realisieren?

  [Allerdings mit dem Mangel, dass die Kindobjekte nicht ihre Farbe behalten haben.]
- Warum geht das so einfach mit den [Affinen] Transformationen?

Die Antwort ist: Beide erfüllen selbst das Kompositummuster!
Ein **`GraphicsPath`** kann aus **`GraphicsPath`**-Objekten bestehen kann und darunter können auch wieder geschachtelt weitere **`GraphicsPath`**-Objekte sein.
Eine Affine Transformation kann verknüpft werden (concatenate, Methodenname aber **`Concat`**) mit einer weiteren [Affinen] Transformation usw. und man erhält dann immer wieder eine [Affine] Transformation. Hier ist allerdings immer nur ein einzelnes Kindobjekt zulässig.

### *Zwei Wege zum Ziel*

Es gibt allerdings zwei verschiedene Wege, die Möbelgrafikobjekte zu Gruppen zu verbinden:

1. Man verbindet die Kindobjekte in der Gruppe zu einem **`GraphicsPath`**.
   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.
2. Man lässt die Kindobjekten auf die Transformation der Gruppe zugreifen.
   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 und bei der hier vorgestellten Lösung wird ebenfalls so vorgegangen.
Der zweite Weg ist aber vielleicht bei einer Kompositumlösung sinnvoller. Dies liegt aber nicht am Kompositum-Muster selbst. Jede von beiden Möglichkeiten hat ihre Vor- und Nachteile. Für welche man sich entscheidet, sollte man nach den Anforderungen entscheiden, die das Problem stellt.

### *Änderungen in Moebel für die zweite Variante*

Die Klasse Moebel führt ein Attribut für eine mögliche Gruppe zu der das Objekt gehört ...

```
        self.__gruppe=None
```

… und eine Methode zum Setzen:

```
    def SetzeGruppe(self, gruppe):
        '''setzt die Zugehörigkeit zu einer Gruppe (Kompositum)'''
        self.__gruppe = gruppe
```

Bevor das Ergebnis der Transformation in ***`Transformiere()`*** zurückgegeben wird, wird der Pfad mit der Transformation der Gruppe transformiert.

```
    def Transformiere(self, path):
        """Transformiert den übergebenen Pfad"""
        gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
        gc.PushState()
```

```
# Hilfvariable verwenden
x,y = self.GibXPosition(), self.GibYPosition()
w = self.GibOrientierung()
b,h = self.GibBreite(), self.GibTiefe()
gc.Translate(x+b/2, y+h/2)
gc.Rotate(radians(w))
gc.Translate(-b/2, -h/2)
transformation = gc.GetTransform()
gc.PopState()
path.Transform(transformation)
if self.__gruppe != None:
    return self.__gruppe.Transformiere(path)
return path
```

## *Überlegungen zum Sinn des Kompositum-Musters*

Sowohl im Buch von Gamma e.a. als auch im Buch "Entwurfsmuster von Kopf bis Fuß" wird die Problematik des Kompositum-Musters intensiv behandelt.

### *Das Muster verstößt gegen ein allgemeines Prinzip*

Mit der Forderung, dass alle Klassen dieselbe Schnittstelle aufweisen sollen, verstößt es in der angegebenen Form gegen das Prinzip, dass eine Klasse nur die Methoden bereitstellen soll, die sie auch benötigt. Und die Kindobjekte, die Blattklassen der Hierarchie sind, benötigen keine Methoden für die Verwaltung von Kindobjekten, da sie keine besitzen. Interessanterweise zeigt das Diagramm die Methoden **`FuegeHinzu()`**, **`Entferne()`** und **`GibKindobjekt()`** auch nicht bei den Blattklassen, lässt sie aber von Komponente erben. Sie müssen dort also in einer Default-Implementation vorliegen, die nichts macht.

### *Ein unauflösbarer Widerspruch*

Dieser Widerspruch lässt nicht auflösen. So kann es sinnvoll sein, nur in der Gruppenklasse diese Methoden vorzusehen. Dann muss man allerdings darauf vertrauen, dass der Nutzer für sie keine dieser Methoden für ein Blatt aufruft.
Man tut also gut daran, sich genau zu überlegen, ob man das strenge Muster verlässt. In jedem Fall gewährleistet man aber zumindest, dass sich Gruppen auch wie einzelne Objekte verhalten können. Dafür ist im vorliegenden Fall schon viel zu tun.

### *Empfehlung*

Meine Empfehlung für die Lösung ist, Moebelgruppe von Moebel erben zu lassen und allein dort die speziellen Methoden **`FuegeHinzu()`**, **`Entferne()`** und **`GibKindobjekt()`** zu realisieren. Dann bietet die Klasse **`Moebel`** zwar nicht die Methoden an, sie sind aber auch für Blattklassen nicht sinnvoll.

### *Alternativen*

Man könnte sich dafür in diesem Anwendungsfall allerdings zwei mögliche Lösungen vorstellen:

- Wenn an einem Blatt die Methode **`FuegeHinzu()`** aufgerufen wird, bildet man aus dem bisherigen und dem neuen Blatt ein Kompositum mit den beiden.
  Es muss dann geklärt werden, ob bei **`Entferne()`** entsprechend anders herum gearbeitet werden soll.
- Man arbeitet grundsätzlich nur mit Komposita. Ein Blatt ist dann ein Kompositum mit genau einem Element in der Liste.

### *Aggregation oder Komposition?*

Die Alternativen führen auf die Überlegung, ob denn leere Gruppen überhaupt sinnvoll sind und ob es im Gegensatz zu der Darstellung im Klassendiagramm, die eine Aggregation zeigt, nicht eine Komposition sein sollte.

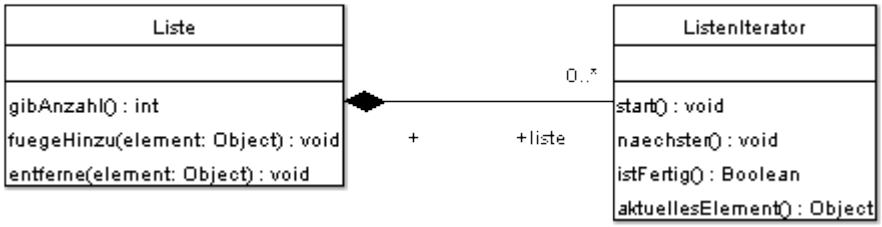
# Das Iteratormuster

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 Liste ist [wie Tupel, ...] kein einfaches Objekt, sondern ein Objekt, das sich aus mehreren Objekten 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. Erweitert man Python also um einen weiteren Sammlungstyp, sollte man dazu auch einen Iterator realisieren.
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[1]:




**for und range**

Python stellt für alle sequenziellen Objekte die for-Schleife zur Verfügung. Die Problematik von range bei den 2er-Versionen (es wird eine Liste konkret angelegt) wird bei den 3er-Versionen von Python sinnvoller durch das Erstellen eines Iterators ersetzt.

1 Vorlage: Gamma e.a.

## Anhang

Klassendiagramm von Dia zum Projekt mit Moebelgruppe
Die Linienführung ist etwas schwieriger zu lesen.