4.7 Das Strukturmuster Kompositum

4.7.1 Name/Alternative Namen

Kompositum (engl. composite), Kompositionsmuster.

4.7.2 Problem

Man möchte Teil-Ganzes-Hierarchien erzeugen und dabei die Objekte in einer baumartigen Struktur gruppieren. Das folgende Bild gibt ein Beispiel einer solchen Teil-Ganzes-Hierarchie:

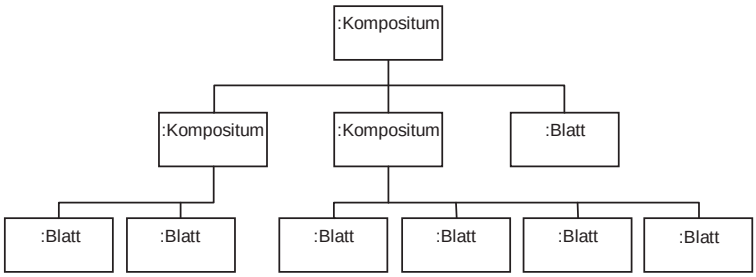


Bild 4-19 Struktur einer Teil-Ganzes-Hierarchie

Die Objekte in einer Baumstruktur werden auch als **Knoten** bezeichnet. Wie in Bild 4-19 zu sehen ist, können die Knoten der Baumstruktur dahingehend unterschieden werden, ob sie selber wieder zusammengesetzt sind (Kompositum-Objekte) oder ob es sich um einfache und nicht zusammengesetzte – also quasi atomare – Objekte (Blatt-Objekte) handelt. Ein Client-Programm soll für ausgesuchte Operationen mit einem Blatt-Objekt wie mit einem Kompositum-Objekt umgehen können, so dass für einen Client keine Unterscheidungen erforderlich werden. Ein Client soll nur die abstrakten Schnittstellen eines Knoten kennen, sei er zusammengesetzt oder nicht. Für ein Client-Programm soll es also verborgen sein, ob ein Knoten einfach oder zusammengesetzt ist.

Das **Kompositum-Muster** soll es erlauben, dass bei der Verarbeitung von Knoten in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden.

4.7.3 Lösung

Das Kompositum-Muster ist ein objektbasiertes Strukturmuster[47]. Durch den Einsatz des Kompositum-Musters wird es möglich, in einer Baumstruktur **zusammengesetzte**

[47] Kompositum ist zugleich auch der Name für eine Klasse, deren Objekte als Knoten in einem Baum auftreten können und deren Objekte selber auch weitere Knoten aggregieren können, um so die Baumstruktur aufzubauen,

Objekte (Gruppen von Objekten) gleich wie einzelne **einfache** oder **primitive** Objekte, sogenannte Blätter, zu behandeln. Dadurch wird der Aufwand im Client für die Verwaltung der resultierenden Baumstruktur verringert.

Die Grundlage des Kompositum-Musters ist die Definition einer abstrakten Klasse `Knoten`, deren Verhalten durch ihre Schnittstelle und ihre Verträge festgelegt wird. Eine Kompositum-Klasse ist von der abstrakten Klasse `Knoten` abgeleitet. Ein Kompositum-Objekt kann auf Grund des Klassendiagramms in Bild 4-20 gleichzeitig mehrere Objekte vom Typ `Knoten` aggregieren. An die Stelle einer Referenz auf ein Objekt der Klasse `Knoten` kann nach dem liskovschen Substitutionsprinzip eine Referenz auf ein Objekt einer von der Klasse `Knoten` abgeleiteten Klasse treten, wenn der Vertrag der Klasse `Knoten` beim Überschreiben eingehalten wird.

Eine Blatt-Klasse ist ebenfalls von der abstrakten Klasse `Knoten` abgeleitet, kann aber im Gegensatz zu einer Kompositum-Klasse keine Knoten aggregieren. Objekte von Blatt-Klassen – also Blätter – stellen somit Abschlusselemente einer Baumstruktur dar.

Damit ist es leicht, geschachtelte Objektstrukturen (Bäume) zu bilden.

Ein Baum besteht aus konkreten Objekten und ein Objekt ist entweder eine Instanz einer Kompositum- oder einer Blatt-Klasse. Das Kompositum-Muster ermöglicht es, eine Baumstruktur von Objekten aufzubauen.

Die Unterscheidung, ob es sich bei einer Klasse der Baumstruktur um eine Blatt-Klasse oder eine Kompositum-Klasse handelt, erfolgt lediglich anhand der Tatsache, ob ein Objekt dieser Klasse weitere Objekte der Baumstruktur aggregieren kann oder nicht.

Wird eine Nachricht an ein Kompositum versendet, so wird die Nachricht zum einen lokal für das zusammengesetzte Objekt ausgeführt und zum anderem an die Kinder weiterdelegiert (**Delegationsprinzip**). Ist der Empfänger hingegen ein Blatt, so wird die Operation direkt ausgeführt, da das Blatt keine Kinder hat.

4.7.3.1 Klassendiagramm

Das Klassendiagramm in Bild 4-20 zeigt, dass eine Klasse `Blatt` und eine Klasse `Kompositum` von einer abstrakten Klasse `Knoten` abgeleitet sind und dass ein Objekt der Klasse `Kompositum` mehrere Objekte vom Typ `Knoten` aggregieren kann:

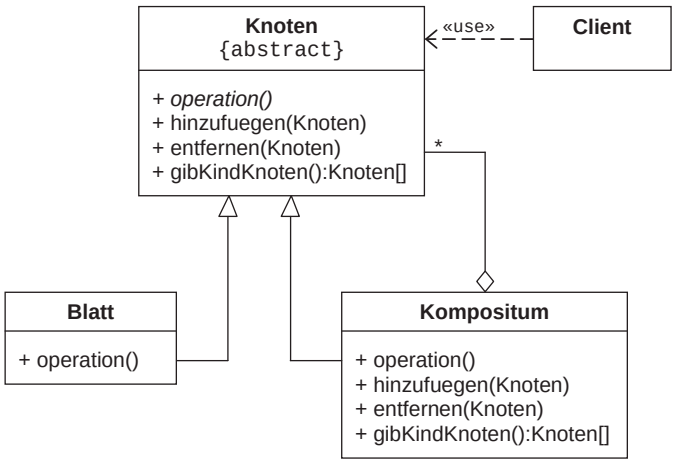


Bild 4-20 Klassendiagramm Kompositum

Wenn der Client nicht wissen soll, ob ein Objekt ein Blatt oder ein Kompositum ist, müssen beide Objekte dieselbe Schnittstelle haben. Die Implementierungen der Operation `operation()` können jedoch bei den verschiedenen Klassen voneinander abweichen.

Ein Client kann sowohl für ein Blatt als auch für ein Kompositum eine Kindoperation wie z. B. `gibKindKnoten()` aufrufen. Er soll ja nicht wissen, um was es sich bei einem Element handelt. Deshalb müssen die Operationen `hinzufuegen()`, `entfernen()` und `gibKindKnoten()` in der Wurzel der Klassenhierarchie bei der Komponente definiert werden. Es ist sinnvoll, in der Wurzelklasse `Knoten` ein Defaultverhalten für diese Kindoperationen zu implementieren, das für Blatt-Klassen geeignet ist und das von einer Kompositum-Klasse überschrieben wird.

Eine gängige Lösung ist, dass die abstrakte Klasse die Kindmethoden so implementiert, dass diese Methoden eine Exception werfen, wenn sie aufgerufen werden. Die Blatt-Klasse überschreibt dieses Verhalten nicht und wirft also eine Exception, wenn eine ihrer Kindmethoden aufgerufen wird. Die Kompositum-Klasse hingegen überschreibt die Kindmethoden in sinnvoller Weise und wirft dabei keine Exception. Dies bedeutet, dass die Nachbedingung einer Kindmethode in der Kompositum-Klasse verschärft wird, was den Vertrag einer Kindmethode nicht bricht.

Die Klassen `Kompositum` und `Blatt` sind im Bild 4-20 nur als Stellvertreter zu betrachten. In einer Baumstruktur kann es sowohl mehrere Kompositum-Klassen als auch mehrere Blatt-Klassen geben. Ein Beispiel für eine Baumstruktur mit mehreren Blatt-Klassen ist in Kapitel 4.7.5 zu sehen.

4.7.3.2 Teilnehmer

Knoten
Die abstrakte Klasse `Knoten` legt die Schnittstelle und das Verhalten der abgeleiteten Klassen `Kompositum` und `Blatt` fest. Es wird ein Defaultverhalten für die Kindoperationen implementiert.

Blatt
Die Klasse `Blatt` repräsentiert ein Abschlusselement in der Baumstruktur, das keine weiteren Knoten aggregiert und selbst immer nur Kind-Knoten sein kann.

Kompositum
Die Klasse `Kompositum` repräsentiert ein Knotenelement in der Baumstruktur, welches weitere Knoten aggregieren kann. Die Klasse `Kompositum` implementiert die kindbezogenen Operationen und überschreibt damit das Defaultverhalten, das in der Klasse `Knoten` implementiert ist.

4.7.3.3 Dynamisches Verhalten

Das dynamische Verhalten des Kompositum-Musters wird an einem Beispiel in Bild 4-21 verdeutlicht. Zuerst fügt der Client dem Objekt `k1` der Klasse `Kompositum` zwei Objekte `b1` und `b2` der Klasse `Blatt` sowie ein Objekt `k2` der Klasse `Kompositum` hinzu. Anschließend wird dem Objekt `k2` der Klasse `Kompositum` noch ein weiteres Blatt `b3` hinzugefügt. Die Baumstruktur sieht somit folgendermaßen aus:

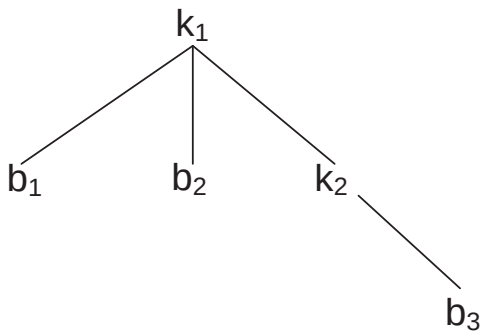


Bild 4-21 Baumstuktur des Beispiels

Nach dem Aufbau der Baumstruktur wird die Methode `operation()` des Objekts `k1` beispielhaft von einem Client aufgerufen. Der gesamte Ablauf dieses Aufrufs ist im folgenden Sequenzdiagramm dargestellt:

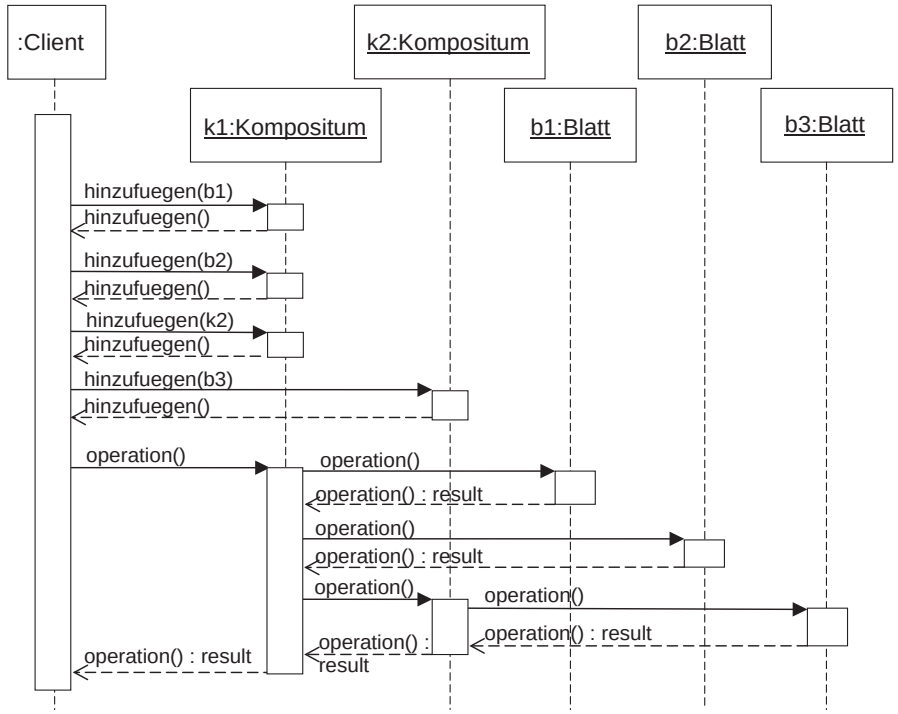


Bild 4-22 Sequenzdiagramm Kompositum-Muster

Wie in Bild 4-22 zu sehen ist, wird beim Aufruf der Methode `operation()` des Objekts `k1` die Anfrage rekursiv an alle untergeordneten Blätter und Kompositum-Instanzen weitergeleitet.

4.7.3.4 Programmbeispiel

Die Klasse `Knoten` definiert eine abstrakte Basisklasse, von der die Klassen `Kompositum` und `Blatt` abgeleitet werden. Die Klasse `Knoten` definiert die abstrakte Methode `operation()`. Es folgt der Quellcode der Klasse `Knoten`:

```
// Datei: Knoten.java
import java.util.ArrayList;
public abstract class Knoten
{
   private String name = "";
   static int ebene = 0;          // Zaehler fuer Ausgabe-Ebene
   ArrayList<Knoten> kindelemente = new ArrayList<Knoten>();

   public Knoten (String name)
   {
      this.name = name;
   }

   public abstract void operation();
```

```
   public void hinzufuegen (Knoten komp)
   {
      System.out.println ("Kind-Methode nicht implementiert!");
   }

   public void entfernen (Knoten komp)
   {
      System.out.println ("Kind-Methode nicht implementiert!");
   }

   public void gibKind()
   {
      System.out.println ("Kind-Methode nicht implementiert!");
   }

   public String gibName()
   {
      return this.name;
   }
}
```

Die Klasse `Kompositum` ist von der Klasse `Knoten` abgeleitet. Sie überschreibt die kindbezogenen Methoden und implementiert die ausgesuchte Operation `operation()`. Hier die Klasse `Kompositum`:

```
// Datei: Kompositum.java
import java.util.Iterator;
public class Kompositum extends Knoten
{
   public Kompositum (String name)
   {
      super (name);
   }

   public void hinzufuegen (Knoten komp)
   {
      this.kindelemente.add (komp);
   }

   public void entfernen (Knoten komp)
   {
      //alle Kindelemente durchlaufen
      for (Iterator<Knoten> iter = kindelemente.iterator();
            iter.hasNext();)
      {
         Knoten f = (Knoten) iter.next();
         if (f instanceof Kompositum)
         {
            ((Kompositum) f).entfernen (komp);
         }
      }
      kindelemente.remove (komp);
   }
```

```
   public void operation()
   {
      String formatString;
      // Berechnen der neuen Ausgabe-Ebene
      ebene++;
      // Berechnen des Formatstrings fuer die Ausgabe von
      // Leerzeichen entsprechend der erreichten Ebene
      formatString = "%" + (ebene * 2) + "s";
      // Ausgabe der Leerzeichen
      System.out.printf (formatString, "");
      // Ausgabe eines Pluszeichens gefolgt vom Namen der Komponente
      System.out.println ("+ " + super.gibName() + "");
      // Aufruf der Operation fuer alle Kindelemente
      for (Iterator<Knoten> iter = kindelemente.iterator();
            iter.hasNext();)
      {
         Knoten f = (Knoten) (iter.next());
         f.operation();
      }
      // Ausgabe-Ebene wieder zuruecksetzen
      --ebene;
   }
}
```

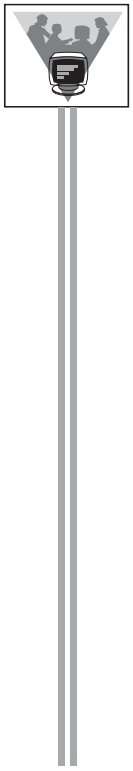
Die Klasse `Blatt` repräsentiert einfache und nicht zusammengesetzte Knoten einer Baumstruktur und hat im Gegensatz zur Klasse `Kompositum` keine untergeordneten Knoten:

```
// Datei: Blatt.java
public class Blatt extends Knoten
{
   public Blatt (String name)
   {
      super (name);
   }

   public void operation()
   {
      String formatString;
      // Berechnen des Formatstrings fuer die Ausgabe von
      // Leerzeichen entsprechend der erreichten Ebene
      formatString = "%" + (ebene * 2) + "s";
      // Ausgabe der Leerzeichen
      System.out.printf (formatString, "");
      // Ausgabe eines Minuszeichens gefolgt vom Namen des Knotens
      System.out.println (" - " + super.gibName());
   }
}
```

Die Klasse `TestKompositum` legt vier Kompositum-Objekte an. Um die Möglichkeit der Rekursion aufzuzeigen, wird `komp121` dem Kompositum `komp12` hinzugefügt. Anschließend werden sechs Instanzen der Klasse `Blatt` erstellt. Diese werden dann den Kompositum-Objekten hinzugefügt. Im nächsten Schritt wird der Inhalt der Baumstruktur ausgegeben. Zum Schluss werden ein Blatt- sowie ein Kompositum-Objekt

entfernt und der Baum erneut ausgegeben. Hier nun der Quellcode der Klasse `Test-Kompositum`:

```
// Datei: TestKompositum.java
public class TestKompositum
{
   public static void main (String[] args)
   {
      System.out.println ("Testprogramm zum Kompositum-Muster");
      System.out.println ("");

      // Aufbau der Objektstruktur
      Kompositum komp = new Kompositum ("Kompositum Ebene 1");
      Kompositum komp11 =
         new Kompositum ("Kompositum Ebene 11");
      Kompositum komp12 =
         new Kompositum ("Kompositum Ebene 12");
      Kompositum komp121 =
         new Kompositum ("Kompositum Ebene 121");

      komp.hinzufuegen (komp11);
      komp.hinzufuegen (komp12);

      komp12.hinzufuegen (komp121);

      Blatt blatt111 = new Blatt ("Blatt111");
      Blatt blatt112 = new Blatt ("Blatt112");

      Blatt blatt121 = new Blatt ("Blatt121");
      Blatt blatt122 = new Blatt ("Blatt122");
      Blatt blatt123 = new Blatt ("Blatt123");

      Blatt blatt1211 = new Blatt ("Blatt1211");

      komp11.hinzufuegen (blatt111);
      komp11.hinzufuegen (blatt112);

      komp12.hinzufuegen (blatt121);
      komp12.hinzufuegen (blatt122);
      komp12.hinzufuegen (blatt123);

      komp121.hinzufuegen (blatt1211);

      // Aufruf der Operation fuer das Kompositum
      System.out.println("Erste Ausgabe der Kompositum-Operation");
      System.out.println();
      komp.operation();

      // Modifikation der Objektstruktur
      komp12.entfernen (blatt122);
      komp12.entfernen (komp121);

      // erneuter Aufruf der Operation fuer das Kompositum
      System.out.println();
      System.out.println("Zweite Ausgabe der Kompositum-Operation");
      System.out.println();
      komp.operation();
   }
}
```



Hier das Protokoll des Programmlaufs:

```
Testprogramm zum Kompositum-Muster

Erste Ausgabe der Kompositum-Operation

  + Kompositum Ebene 1
    + Kompositum Ebene 11
      - Blatt111
      - Blatt112
    + Kompositum Ebene 12
      + Kompositum Ebene 121
        - Blatt1211
      - Blatt121
      - Blatt122
      - Blatt123

Zweite Ausgabe der Kompositum-Operation

  + Kompositum Ebene 1
    + Kompositum Ebene 11
      - Blatt111
      - Blatt112
    + Kompositum Ebene 12
      - Blatt121
      - Blatt123
```

4.7.4 Bewertung

4.7.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Da ein Objekt der Klasse `Blatt` dieselbe Schnittstelle hat wie ein Objekt der Klasse `Kompositum`, kann der Client Blatt-Objekte und zusammengesetzte Kompositum-Objekte einheitlich behandeln. Dies vereinfacht die Handhabung der Baumstruktur durch den Client.
- Das Strukturmuster Kompositum erlaubt es, verschachtelte Strukturen auf einfache Weise zu erzeugen bzw. um neue Blatt- bzw. Kompositum-Klassen zu erweitern. Blatt- und Kompositum-Klassen verwenden dieselbe Schnittstelle. Hierbei kann das zusammengesetzte Objekt mehrfach verschachtelt sein.

4.7.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Das Design und der Aufbau der Baumstruktur werden unübersichtlich, wenn man viele unterschiedliche Blatt- und Kompositionsklassen verwendet. Dieses Problem

entsteht aber durch die Art, wie das Strukturmuster Kompositum verwendet wird, und nicht durch das Strukturmuster selbst.

- Im Kompositum-Muster werden alle Knoten gleich behandelt. Bei Anwendungen, in denen der Aufbau einer Kompositum-Struktur gewissen Einschränkungen unterliegen soll, müssen diese Einschränkungen durch Typprüfungen zur Laufzeit gewährleistet werden. Als Beispiel hierzu wird folgendes Szenario betrachtet: Ordner können Ordner und Dateien enthalten. Source-Ordner sind ebenfalls Ordner, können aber nur bestimmte Dateien, nämlich Quellcode-Dateien enthalten. Eine solche Einschränkung kann über das Kompositum-Muster nicht umgesetzt werden.
- Das Entwurfsmuster Kompositum ist sehr mächtig. Sobald jedoch Änderungen an der Basisschnittstelle durchgeführt werden, wie z. B. das Hinzufügen einer neuen Methode, bedeutet dies auch, dass alle davon abgeleiteten Klassen potenziell ebenfalls geändert werden müssen.

4.7.5 Einsatzgebiete

Das Kompositum-Muster kann im Zusammenhang mit dem Entwurfsmuster **Befehl** angewandt werden. Mit Hilfe des Kompositum-Musters kann ein Befehl aus mehreren Befehlen zusammengesetzt werden. Dadurch können Befehlsskripte bzw. Makrobefehle erstellt werden und trotzdem im Rahmen des Befehlsmusters wie einfache Befehle behandelt werden.

Zusammengesetzte Nachrichten, die über mehrere Schichten einer Schichtenarchitektur (siehe Kapitel 5.1) weitergereicht werden, sind typischerweise ineinander verschachtelt und können mit Hilfe des Kompositum-Musters zusammengesetzt werden. Diese Anwendung des Kompositum-Musters ist in der Literatur als eigenständiges Muster unter dem Namen **Composite-Message** [Bus98] bekannt.

Beim Zusammensetzen von grafischen Oberflächen – wie es auch die Views im **MVC-Muster** sind –, ist das Kompositum-Muster weit verbreitet. Ein Fenster einer View wird dabei im Grafik-Paket Swing von Java aus Paneln aus verschachtelten Komponenten zusammengesetzt. Durch die Verschachtelung entsteht eine Baumstruktur aus Komponenten, die aus Komponenten zusammengesetzt sind (in Swing Panels[48]), und aus Blättern, die keine anderen Komponenten in sich tragen (z. B. Schaltflächen). Die Anwendung des Kompositum-Musters bei graphischen Oberflächen wird im folgenden Beispiel im Detail ausgeführt.

Anwendungsbeispiel: Grafische Oberflächen mit Swing

Es soll die in folgendem Bild dargestellte Oberfläche erzeugt werden:

[48] Panels stellen unsichtbare Objektbehälter dar, mit deren Hilfe es möglich ist, grafische Komponenten zu Gruppen zusammenzufassen.

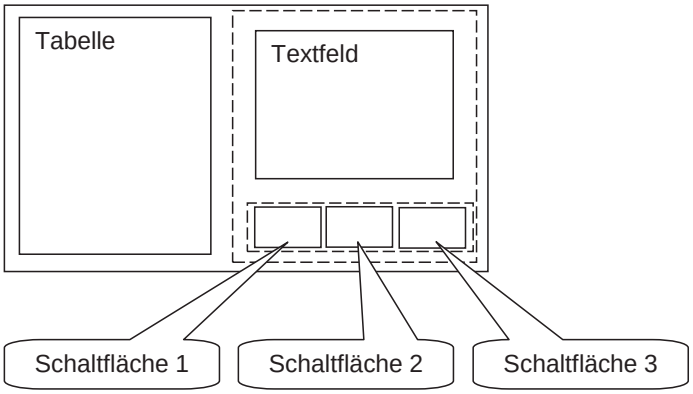


Bild 4-23 Zu realisierende grafische Oberfläche

Dabei repräsentieren die gestrichelten Linien unsichtbare Behälterobjekte (Panels), die andere grafische Komponenten gruppieren können. Der äußere Rahmen (Frame) stellt quasi das Wurzelelement der verschachtelten Struktur dar.

In Grafikpaket Swing von Java gibt es zur Realisierung dieser Beipieloberfläche die folgenden Klassen: `JTextField` für Textfelder, `JButton` für Schaltflächen, `JTable` für Tabellen und `JPanel` für die Gruppierung von Gestaltungselementen. Dabei erben alle diese Klassen von derselben Basisklasse, nämlich von der Klasse `JComponent`. Die Vererbung geht bei einigen Klassen über mehrere Stufen, was im Klassendiagramm in Bild 4-24 nicht gezeigt ist, damit die Analogie zum Klassendiagramm des Kompositum-Musters in Bild 4-20 deutlicher wird:

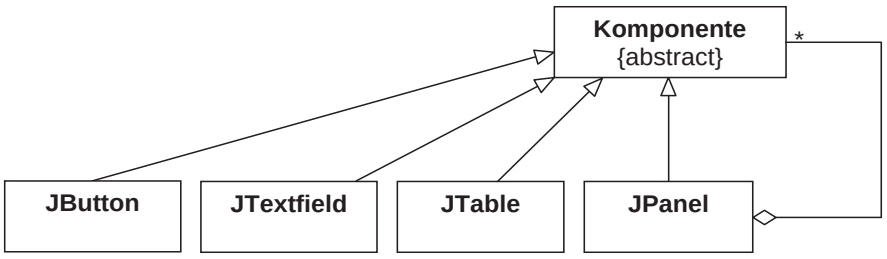


Bild 4-24 Klassendiagramm für eine Anwendung des Kompositum-Musters

Die Klasse `JComponent` selbst ist abgeleitet von der Klasse `Container` aus dem Paket der grafischen Benutzerschnittstelle AWT (**A**bstract **W**indow **T**oolkit). Damit haben alle in Bild 4-24 gezeigten Klassen die Eigenschaften der Klasse `Container` und können beliebig viele grafische Elemente aufnehmen. Diese Fähigkeiten werden aber im Bild nur von der Klasse `JPanel` genutzt, was durch die Aggregationsbeziehung im Bild 4-24 angedeutet wird. Die Klasse `JPanel` ist also im Sinne des Kompositum-Musters eine Kompositum-Klasse, während die Klassen `JTextField`, `JButton` und `JTable` Blatt-Klassen sind.

Daneben gibt es noch die Klasse `JFrame`, die ebenfalls von der Klasse `Container` abgeleitet ist, die aber nur als Wurzelelement für den äußeren Rahmen einer grafischen Oberfläche genutzt werden kann und daher nicht als Kompositum-Klasse angesehen werden kann.

Das folgende Codefragment zeigt, wie man in Swing die in Bild 4-23 skizzierte Oberfläche mit Objekten der genannten Klassen zusammensetzen kann. Die Layoutmanager werden bei dieser Betrachtung außer Acht gelassen. Hier das bereits erwähnte Codefragment:

```
// Blatt-Objekte erzeugen
JTextField textField = new JTextField ("Textfeld");
JButton button1 = new JButton ("Schaltflaeche 1");
JButton button2 = new JButton ("Schaltflaeche 2");
JButton button3 = new JButton ("Schaltflaeche 3");
JTable table = new JTable();

// Kompositum-Objekte erzeugen
JPanel panelRechts = new JPanel();
JPanel panelUnten = new JPanel();

// Struktur aufbauen
panelRechts.add (textField);
panelRechts.add (panelUnten);
panelUnten.add (button1);
panelUnten.add (button2);
panelUnten.add (button3);

// in aeusseren Rahmen einhaengen
JFrame frame = new JFrame ("Mein Fenster");
frame.add (table);
frame.add (panelRechts);
```

Was passiert nun, wenn man die obige Klassenhierarchie zugrunde legt und den äußeren Rahmen vergrößert? Das Frame-Objekt vergrößert sich selbst und leitet den Aufruf an alle Objekte, die in ihm enthalten sind, weiter. Durch die Weiterleitung geht der Aufruf unter anderem an das Objekt `panelRechts`. Dieses Objekt ist selber ein Kompositum-Objekt und leitet wiederum den Aufruf an alle enthaltenen Objekte weiter. Dies geht rekursiv weiter, bis alle Elemente vergrößert sind.

4.7.6 Ähnliche Entwurfsmuster

Von der Struktur des Klassendiagramms her ist das Entwurfsmuster Kompositum sehr ähnlich zu dem **Dekorierer-Muster**. Ein Klassendiagramm des Dekorierers ist quasi ein Spezialfall des Klassendiagramms des Kompositum-Musters, dadurch dass ein Dekorierer-Objekt nur ein einziges Objekt aggregieren kann. Es zeigen sich jedoch Unterschiede in der Verwendung dieser beiden Muster: Während mit Hilfe des Dekorierer-Musters Funktionalität dynamisch gebildet werden kann, dient das Kompositum-Muster dazu, Objekte zu einer hierarchischen Struktur zusammenzusetzen.

Das Muster **Whole-Part** [Bus98] ist sowohl vom Aufbau als auch von der Funktionalität her sehr ähnlich dem Kompositum-Muster. Das Muster Whole-Part fordert aber im

Unterschied zum Kompositum-Muster, dass auf die Teilobjekte (Parts) einer Struktur nicht mehr zugegriffen werden darf, sondern von außen nur das Gesamtobjekt (Whole) angesprochen werden darf. Auf Grund dieser Forderung müssen die beim Muster Whole-Part beteiligten Teilobjekte nicht unbedingt alle die gleiche Schnittstelle anbieten wie beim Kompositum-Muster.

Als Variante von Whole-Part wird in [Bus98] das Muster **Assembly-Parts** vorgestellt. Im Muster Assembly-Parts ist die Struktur des Gesamtobjekts starr festgelegt und auch in diesem Muster können die Teilobjekte unterschiedliche Schnittstellen haben. Als Beispiel wird in [Bus98] angeführt, dass ein Auto aus Karosserie, Fahrwerk, Motor etc. besteht und dass diese Struktur sich zwischen verschiedenen Autotypen nicht grundsätzlich ändert. Das Kompositum-Muster hingegen schränkt die Flexibilität beim Aufbau einer Datenstruktur prinzipiell nicht ein und lässt auch nachträgliche Änderungen an der Datenstruktur zu. Ein weiterer Unterschied ist, dass das Muster Assembly-Parts keine gemeinsame Schnittstelle für die Teilobjekte voraussetzt, wie sie im Kompositum-Muster durch die Klasse `Knoten` (siehe Bild 4-20) vorgegeben ist.