Design Patterns

Lukas Wais

23. Juni 2019

1 Dekorator

Dieses Muster soll es erlauben, zur Laufzeit eine zusätzliche Funktionalität zu einem vorhandenen Objekt in dynamischer Weise hinzuzufügen.

1.1 Definition

Lösungsansatz. In einem Dekorierer soll das zu verzierende, sprich das zu erweiternde Objekt zu aggregieren (verbinden) und gleichzeitig einem Kunden, dem Client, dieselbe Schnittstelle wie die zu verzierende Komponente angeboten werden.

Soll der Dekorierer eine bestimmte Methode der zu dekorierenden Klasse um eine zusätzliche Funktionalität erweitern so überschreibt er diese Methode. Um die bestehende Funktion zu nutzen, ruft der Dekorierer in der überschreibenden Methode die überschriebene Methode des aggregierten Objekts auf.

Wichtig. Ein Dekorierer muss alle geerbten Methoden überschreiben.

1.2 UML - Diagramme

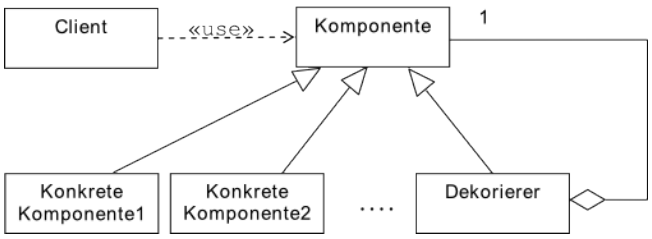

Abbildung 1: UML-Diagramm Dekorierer

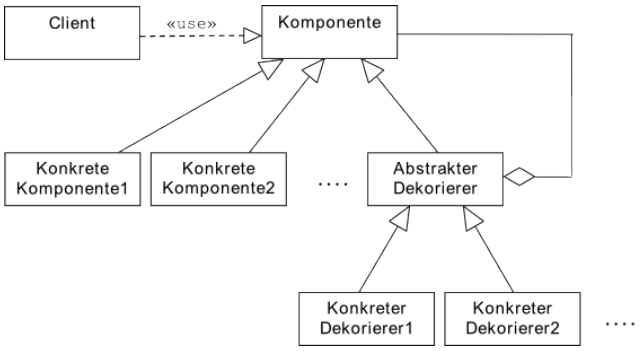

Abbildung 2: UML-Diagramm mehrere Dekorierer

1.3 Konkretes Beispiel

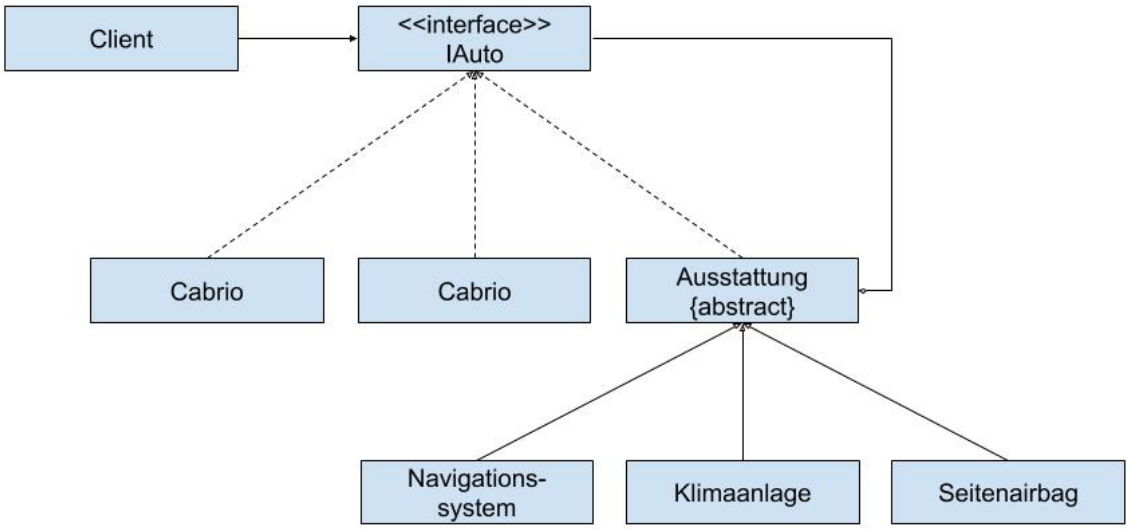

Abbildung 3: UML-Diagramm Dekorierer

```
1 public interface IAuto {
2
3   public int gibKosten();
4   public void zeigeDetails();
5 }
```

Listing 1: IAuto.java

```java
class Cabrio implements IAuto {
  public void zeigeDetails() {
    System.out.print("Cabrio");
  }

  public int gibKosten() {
    return 50000;
  }
}
```

Listing 2: Cabrio.java

```java
public abstract class Ausstattung implements IAuto {
  protected IAuto auto;

  public Ausstattung(IAuto pIAuto) {
    auto = pIAuto;
  }
}
```

Listing 3: Ausstattung.java

```java
class Klimaanlage extends Ausstattung {
  public Klimaanlage(IAuto pIAuto) {
    super(pIAuto);
  }

  // "dekoriert" die Details
  public void zeigeDetails() {
    auto.zeigeDetails();
    System.out.print(", Klimaanlage");
  }

  // "dekoriert" die Kosten
  public int gibKosten() {
    return auto.gibKosten() + 1500;
  }
}
```

Listing 4: Klimaanlage.java

```
class Navigationssystem extends Ausstattung {
  public Navigationssystem(IAuto pIAuto) {
    super(pIAuto);
  }

  // "dekoriert" die Details
  public void zeigeDetails() {
    auto.zeigeDetails();
    System.out.print(", Navigationssystem");
  }

  // "dekoriert" die Kosten
  public int gibKosten() {
    return auto.gibKosten() + 2500;
  }
}
```

Listing 5: Navigationssystem.java

```
class Seitenairbags extends Ausstattung {
  public Seitenairbags(IAuto pIAuto) {
    super(pIAuto);
  }

  // "dekoriert" die Details
  public void zeigeDetails() {
    auto.zeigeDetails();
    System.out.print(", Seitenairbags");
  }

  // "dekoriert" die Kosten
  public int gibKosten() {
    return auto.gibKosten() + 1000;
  }
}
```

Listing 6: Seitenairbag.java

```java
class Client {

  // Auto mit Klimaanlage
  public static void main(String[] args) {
    IAuto auto = new Klimaanlage(new Limousine());
    auto.zeigeDetails();
    System.out.println("\nfuer " + auto.gibKosten() + " Euro\n");

    // Dynamische Erweiterung der Limousine mit Ausstattungen
    auto = new Navigationssystem(new Seitenairbags(auto));
    auto.zeigeDetails();
    System.out.println("\nfuer " + auto.gibKosten() + " Euro\n");

    // Cabrio Variante
    auto = new Navigationssystem(new Seitenairbags(new Cabrio()));
    auto.zeigeDetails();

    System.out.println("\nfuer " + auto.gibKosten() + " Euro\n");
  }
}
```

Listing 7: Client.java

1.4 Vor- und Nachteile

1.4.1 Vorteile

- Flexibler als Vererbung.
- Weniger Klassen nötig.
- Funktionalität beliebig (tief) zusammensetzbar.
- Funktionalität zur Laufzeit änderbar (hinzufügbar und entfernbar).
- Dekorierte Klasse bleibt erweiterbar.

1.4.2 Nachteile

- Etwas ineffizienter (durch Forwarding).
- Dekorator $\neq$ dekoriertes Objekt (keine Objektidentität).
- Auch Zugriffe zu Feldern müssen über Methoden gehen.

2 Besucher

Das Besuchermuster soll es erlauben, die Daten aller Objekte einer Objektstruktur zentral auszuwerten.

2.1 Definition

Lösungsansatz. Ein wichtiges Kennzeichen des Besuchermusters ist es, dass die Objekte der Objektstruktur Instanzen von unterschiedlichen Klassen sein können.
Für jede zu realisierende Operation wird eine **Besucher-Klasse** bereitgestellt, die einen Satz von überladenen **visit()-Methoden** enthält und zwar jeweils eine separate visit()-Methode für jede in der Objektstruktur vorhandene Element-Klasse.
Die **accept()-Methode** einer Element - Klasse hat als Übergabeparameter ein Besucher-Objekt . Das Objekt, dessen accept()-Methode aufgerufen wird, ruft im Rumpf der accept()-Methode die für eine Element - Klasse spezifische visit()-Methode des übergebenen Besuchers auf und **übergibt dabei eine Referenz auf sich selbst.**

```
1  @Override
2  public <T> T accept(ExprVisitor<T> visitor) {
3    return visitor.visit(this);
4  }
```

Listing 8: Beispiel Accept-Methode

2.2 Klassendiagramm

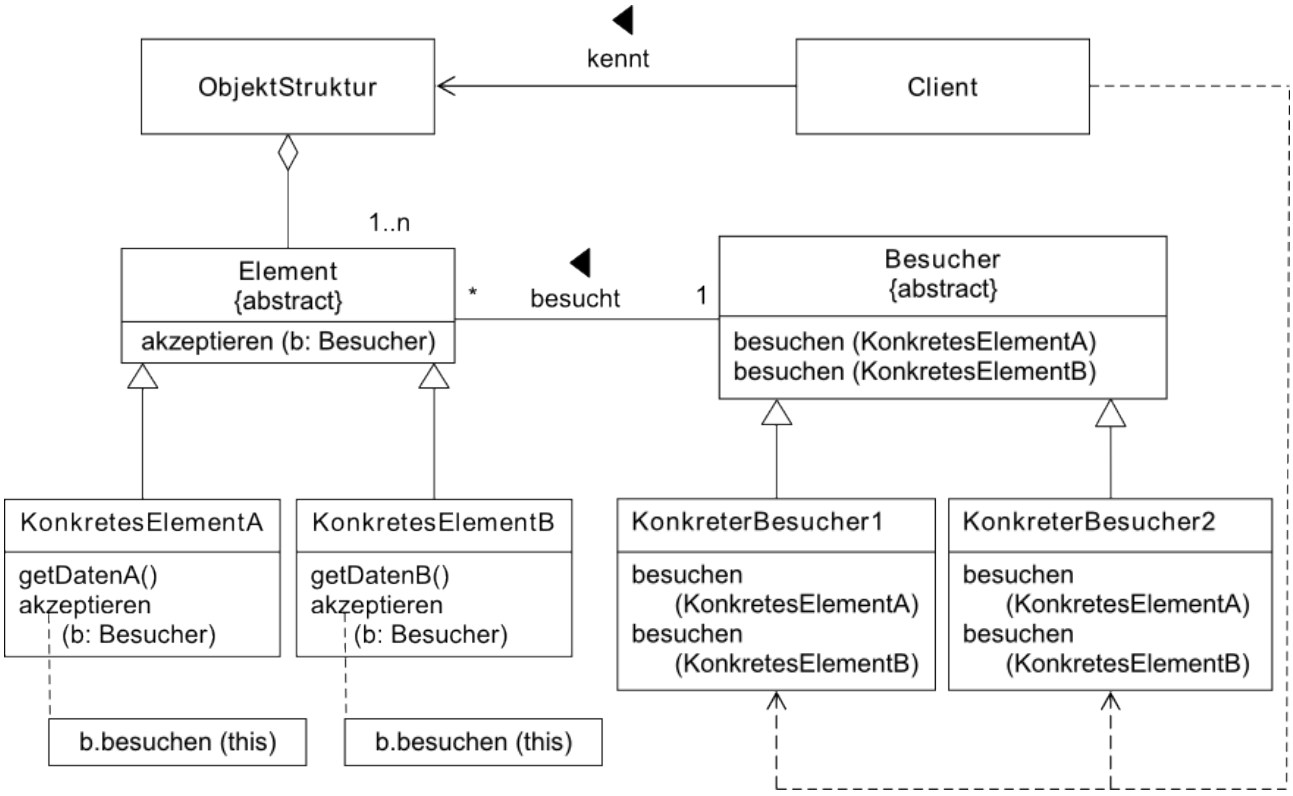

Abbildung 4: Klassendiagramm Visitor

2.3 Konkretes Beispiel

Die abstrakte Klasse MitarbeiterBesucher repräsentiert die abstrakte Besucher-Klasse und definiert für jede konkrete Element-Klasse, welche besucht werden soll. **Visit-Methoden**

```
1 package Visitor;
2
3 // Datei: MitarbeiterBesucher.java
4 abstract class MitarbeiterBesucher {
5   public abstract void besuchen(Teamleiter t);
6
7   public abstract void besuchen(Sachbearbeiter s);
8 }
```

Listing 9: Abstrakte Klasse MitarbeiterBesucher.java

Die Klasse Gehaltsdrucker stellt einen konkreten Besucher dar und wird aus diesem Grund von der Klasse MitarbeiterBesucher abgeleitet. In den Implementierungen der jeweiligen visit()-Methoden werden **Informationen über das jeweils gerade besuchte Objekt ausgegeben.**

```
1  package Visitor;
2
3  // Datei: Gehaltsdrucker.java
4  class Gehaltsdrucker extends MitarbeiterBesucher {
5    public Gehaltsdrucker() {
6      System.out.print("**************************************");
7      System.out.println("**************************");
8      System.out.println("                              Gehaltsliste");
9      System.out.print("Position" + "\t" + "Vorname" + "\t" + "\t" + "Name" + "\t"
       + "\t");
10     System.out.println("Gehalt" + "\t" + "  Praemie");
11     System.out.print("**************************************");
12     System.out.println("**************************");
13   }
14
15   public void besuchen(Teamleiter t) {
16     String vorname;
17     String name;
18     if (t.getVorname().length() < 8)
19       vorname = t.getVorname().concat("\t");
20     else
21       vorname = t.getVorname();
22
23     if (t.getName().length() < 8)
24       name = t.getName().concat("\t");
25     else
26       name = t.getName();
27
28     System.out.print("Leiter " + t.getTeambezeichnung() + "\t" + vorname + "\t"
       + name + "\t");
29     System.out.printf("%7.2f", t.getGrundgehalt());
30     System.out.printf("%3.2f", t.getPraemie());
31     System.out.println();
32   }
33
34   public void besuchen(Sachbearbeiter s) {
35     String vorname;
36     String name;
37     if (s.getVorname().length() < 8)
38       vorname = s.getVorname().concat("\t");
39     else
40       vorname = s.getVorname();
41
42     if (s.getName().length() < 8)
43       name = s.getName().concat("\t");
44     else
45       name = s.getName();
```

```
46
47      System.out.print("Sachbearbeiter" + "\t" + vorname + "\t" + name + "\t");
48      System.out.printf("%7.2f", s.getGehalt());
49      System.out.println("  ---");
50    }
51  }
```

Listing 10: Gehaltsdrucker.java abgeleitet von MitarbeiterBesucher

Durch die Klasse Gesellschaft wird die Objektstruktur realisiert. Sie beinhaltet **Beispielinstanzen der konkreten Objekte**.

```
1  package Visitor;
2
3
4  // Datei: Gesellschaft.java
5  import java.util.ArrayList;
6  import java.util.List;
7
8  // Diese Klasse repraesentiert eine Firma und enthaelt ihre
9  // Mitarbeiter
10 class Gesellschaft {
11   private List<Mitarbeiter> personal;
12
13   public Gesellschaft() {
14     this.personal = new ArrayList<Mitarbeiter>();
15     initialisiereBeispieldaten();
16   }
17
18   private void initialisiereBeispieldaten() {
19     // Sachbearbeiter Team 1
20     ArrayList<Mitarbeiter> team1 = new ArrayList<Mitarbeiter>();
21
22     team1.add(new Sachbearbeiter("Markus", "Mueller ", 48200.0f));
23     team1.add(new Sachbearbeiter("Silvia", "Neustedt", 45500.0f));
24
25     // Sachbearbeiter Team 2
26     ArrayList<Mitarbeiter> team2 = new ArrayList<Mitarbeiter>();
27
28     team2.add(new Sachbearbeiter("Alexandra", "Weiss", 37120.0f));
29     team2.add(new Sachbearbeiter("Michael", "Kienzle", 35500.0f));
30
31     // Teamleiter
32     Teamleiter chef1, chef2;
33
34     chef1 = new Teamleiter("Frank", "Hirschle", 40000.0f, 400.0f, "Team 1");
35     chef2 = new Teamleiter("Corinna", "Steib", 35000.0f, 350.0f, "Team 2");
36
37     // alle Personen in die Personalliste
38     this.personal.add(chef1);
39     this.personal.add(chef2);
40     this.personal.addAll(team1);
41     this.personal.addAll(team2);
```

```
42    }
43
44    public List<Mitarbeiter> getPersonal() {
45      return personal;
46    }
47  }
```

Listing 11: Gesellschaft.java

Die abstrakte Klasse Mitarbeiter ist die **Basisklasse für die konkreten Elemente** dieses Beispiels. Sie entspricht somit der abstrakten Klasse Element aus der allgemeinen Beschreibung des Besucher-Musters. Die abstrakte Klasse Mitarbeiter **gibt für konkrete Element-Klassen die Deklaration der abstrakten Methode accept() vor.**

```
1  package Visitor;
2
3  // Datei: Mitarbeiter.java
4  abstract class Mitarbeiter {
5    protected int personalnummer;
6    private static int anzahlMitarbeiter = 0;
7    protected String name;
8    protected String vorname;
9
10   Mitarbeiter(String vorname, String name) {
11     this.personalnummer = anzahlMitarbeiter++;
12     this.vorname = vorname;
13     this.name = name;
14   }
15
16   public int getPersonalnummer() {
17     return personalnummer;
18   }
19
20   public String getName() {
21     return name;
22   }
23
24   public void setName(String name) {
25     this.name = name;
26   }
27
28   public String getVorname() {
29     return vorname;
30   }
31
32   public void setVorname(String vorname) {
33     this.vorname = vorname;
34   }
35
36   public String toString() {
37     return ("PersonalNr." + this.personalnummer + "Name:" + this.vorname + " " +
       this.name);
38   }
```

```
39
40   public abstract void akzeptieren(MitarbeiterBesucher v);
41 }
```

Listing 12: Abstrakte Klasse Mitarbeiter.java abgeleitet von MitarbeiterBesucher

Die Klasse Sachbearbeiter ist eine **konkrete Element-Klasse**. Sie ist von der abstrakten Klasse Mitarbeiter abgeleitet und implementiert die accept()-Methode.

```
1  package Visitor;
2
3  // Datei: Sachbearbeiter.java
4  class Sachbearbeiter extends Mitarbeiter {
5    private float gehalt;
6
7    public Sachbearbeiter(String vorname, String name, float gehalt) {
8      super(vorname, name);
9      this.gehalt = gehalt;
10   }
11
12   public float getGehalt() {
13     return gehalt;
14   }
15
16   public void akzeptieren(MitarbeiterBesucher v) {
17     // sich selbst besuchen lassen
18     v.besuchen(this);
19   }
20 }
```

Listing 13: Klasse Sachbearbeiter.java abgeleitet von Mitarbeiter

Die Klasse Teamleiter stellt ebenfalls eine **konkrete Element-Klasse** dar. Sie ist von der abstrakten Klasse Mitarbeiter abgeleitet und implementiert die von der abstrakten Klasse Mitarbeiter vorgegebene accept()-Methode.

```
1  package Visitor;
2
3  // Datei: Teamleiter.java
4  class Teamleiter extends Mitarbeiter {
5    private String teambezeichnung;
6    private float grundgehalt;
7    private float praemie;
8
9    public Teamleiter(String vorname, String name, float grundgehalt, float
       praemie, String teambezeichnung) {
10     super(vorname, name);
11     this.grundgehalt = grundgehalt;
12     this.praemie = praemie;
13     this.teambezeichnung = teambezeichnung;
14   }
15
16   public String getTeambezeichnung() {
17     return teambezeichnung;
18   }
19
20   public void setTeambezeichnung(String teambezeichnung) {
21     this.teambezeichnung = teambezeichnung;
22   }
23
24   public float getGrundgehalt() {
25     return this.grundgehalt;
26   }
27
28   public void setGrundgehalt(float grundgehalt) {
29     this.grundgehalt = grundgehalt;
30   }
31
32   public float getPraemie() {
33     return this.praemie;
34   }
35
36   public void setPraemie(float praemie) {
37     this.praemie = praemie;
38   }
39
40   public void akzeptieren(MitarbeiterBesucher v) {
41     // sich selbst besuchen lassen
42     v.besuchen(this);
43   }
44 }
```

Listing 14: Klasse Teamleiter.java abgeleitet von Mitarbeiter

Innerhalb der main()-Methode der Klasse PersonalVerwaltung wird eine neue Objektstruktur als Objekt der Klasse Gesellschaft angelegt. Aus dieser kann die aktuelle Belegschaft in Form einer Mitarbeiterliste ermittelt werden. Die Belegschaft wird durchlaufen und die jeweiligen Mitarbeiter werden durch den Gehaltsdrucker **besucht**.

```
1  package Visitor;
2
3
4  // Datei: PersonalVerwaltung.java
5  import java.util.List;
6
7  public class PersonalVerwaltung {
8    public static void main(String[] args) {
9      // Initialisierungen vornehmen
10     Gesellschaft firma = new Gesellschaft();
11     List<Mitarbeiter> belegschaft = firma.getPersonal();
12
13     // Besucher-Objekt fuer die Liste erzeugen
14     Gehaltsdrucker besucher = new Gehaltsdrucker();
15
16     // Ueber die Liste iterieren und Besuche durchfuehren
17     for (Mitarbeiter arbeiter : belegschaft) {
18       arbeiter.akzeptieren(besucher);
19     }
20   }
21 }
```

Listing 15: Klasse PersonalVerwaltung.java mit der main()-Methode

2.4 Vor- und Nachteile

2.4.1 Vorteile

- Einfaches Hinzufügen von neuer Funktionalität.
- Zentralisierung des Codes einer Operation.
- Möglichkeit Klassenhierarchien-übergreifender Besuche (nicht wie beim Iterator-Muster).
- Sammeln von Informationen.
- Verbesserung der Wartbarkeit.
- Möglichkeit Frameworks zu erweitern.

2.4.2 Nachteile

- Hoher Aufwand beim Hinzufügen von Element-Klassen.
- Hoher Aufwand bei der nachträglichen Anwendung des Musters.
- Overhead (durch das simulierte Double Dispatch in der accept()-Methode entsteht zusätzlicher Aufwand, der die Performance verschlechtert).
- Aufweichung der Kapselung privater Daten.

3 Adapter

Eine Klasse soll wiederverwendet werden. Die zu wiederzuverwendende Klasse bietet zwar die richtigen Daten an, hat aber eine unpassende Schnittstelle für den Zugriff eines Clients auf diese Daten.

3.1 Definition

Lösungsansatz. Das Adapter-Muster hat zum Ziel, eine vorhandene „falsche“ Schnittstelle einer bereits vorhandenen Klasse an die vom Client gewünschte Form anzupassen.

3.2 Klassendiagramm

3.2.1 Klassen-Adapter (Adapter mit Vererbung)

Die Klasse Adapter verwendet für den Aufruf der „alten“ Operationen einfach eine Selbstdelegation an den ererbeten Anteil. Die zu adaptierende Klasse stellt die anzupassende Komponente dar.

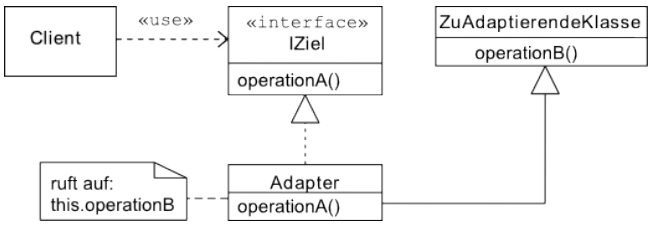

Abbildung 5: Klassendiagramm Klassen-Adapter

3.2.2 Objekt-Adapter (Adapter mit Delegation)

Die Adapter-Klasse implementiert die Aufrufschnittstelle IZiel und delegiert den Aufruf an das aggregierte Objekt der zu adaptierenden Klasse weiter.

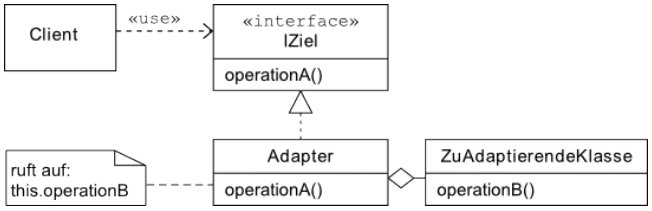

Abbildung 6: Klassendiagramm Objekt-Adapter

3.3 Konkretes Beispiel

Als Beispiel soll eine kleine Anwendung dienen, die Personendaten aus einer CSV-Datei ausliest. Das Einlesen von CSV-Dateien aus einer Datei in einen Zwischenpuffer sei in diesem Beispiel schon von einer Klasse implementiert worden, die vor längerer Zeit geschrieben wurde und deshalb **nicht** auf die Architektur der Anwendung, d.h. auf die Aufrufschnittstelle des Clients, abgestimmt ist. Um diese Klasse zur Anwendung **kompatibel zu machen**, wird eine Adapter-Klasse eingesetzt.

Hierbei stellt die Klasse *CSVLeser die zu adaptierende Klasse dar*, die Klasse *CSVLeserAdapter den Adapter* und die Klasse *TestAdapter den Client*. Die Klasse CSVLeserAdapter muss das Interface IPersonenLeser implementieren.

```
1  package Adapter;
2
3  // Datei: Person.java
4  public class Person {
5    private String nachname;
6    private String vorname;
7
8    public Person(String nachname, String vorname) {
9      this.nachname = nachname;
10     this.vorname = vorname;
11   }
12
13   public void print() {
14     System.out.println(vorname + " " + nachname);
15   }
16 }
```

Listing 16: Person.java Hilfsklasse spielt keine Rolle im Adapter Muster

```java
package Adapter;

// Datei: CSVLeser.java
import java.io.*;
import java.util.Vector;

public class CSVLeser {
  // hat als Rueckgabewert einen Vektor vom Typ eines String-Arrays
  public Vector<String[]> lesePersonenDatei(String file) {
    Vector<String[]> personen = new Vector<String[]>();
    try {
      BufferedReader input = new BufferedReader(new FileReader(file));
      String strLine;

      while ((strLine = input.readLine()) != null) {
        String[] splitted = strLine.split(",");
        if (splitted.length >= 2)
          personen.add(new String[] { splitted[0], splitted[1] });
      }
      input.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    return personen;
  }
}
```

Listing 17: CSVLeser.java die zu adaptierende Klasse

```java
package Adapter;

// Datei: IPersonenLeser.java
import java.util.Vector;

public interface IPersonenLeser {
  // hat als Rueckgabewert einen Vektor von Personen
  public Vector<Person> lesePersonen();
}
```

Listing 18: IPersonenLeser.java Schnittstelle für alle Klassen die Personendaten einlesen können

```java
package Adapter;

// Datei: CSVLeser.java
import java.io.*;
import java.util.Vector;

public class CSVLeser {
  // hat als Rueckgabewert einen Vektor vom Typ eines String-Arrays
  public Vector<String[]> lesePersonenDatei(String file) {
    Vector<String[]> personen = new Vector<String[]>();
    try {
      BufferedReader input = new BufferedReader(new FileReader(file));
      String strLine;

      while ((strLine = input.readLine()) != null) {
        String[] splitted = strLine.split(",");
        if (splitted.length >= 2)
          personen.add(new String[] { splitted[0], splitted[1] });
      }
      input.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    return personen;
  }
}
```

Listing 19: CSVLeserAdapter.java Adapter für die Klasse CSVLeser

```java
package Adapter;


// Datei: TestAdapter.java
import java.util.Vector;

public class TestAdapter {
  public static void main(String[] args) {
    IPersonenLeser leser = new CSVLeserAdapter("Personen.csv");
    Vector<Person> personen = leser.lesePersonen();

    for (Person person : personen)
      person.print();
  }
}
```

Listing 20: TestAdapter.java mit main()-Methode dient als Client

3.4 Vor- und Nachteile

3.4.1 Vorteile

- Ermöglicht Kommunikation zwischen zwei unabhängigen Softwarekomponenten.
- Adapter können um beliebig viele Funktionen erweitert werden (z.B. Filter).
- Sie sind individuell an die Lösung angepasst und können daher optimiert werden.
- Klassen können leicht ausgetauscht werden.
- Ein Adapter kann auch auf ein Objekt einer Unterklasse der zu adaptierenden Klasse angewandt werden.

3.4.2 Nachteile

- Durch das Adapter-Muster wird beim Aufruf einer Operation ein zusätzlicher Zwischenschritt eingeführt, dies kann bei komplexen Adapter zu zeitlicher Verzögerung führen.
- Durch die individuelle Anpassung der Adapter auf die jeweilige Lösung weisen sie eine schlechte Wiederverwendbarkeit auf.

4 Composite

Man möchte Teil-Ganzes-Hierachien erzeugen und dabei die Objekte in einer **baumartigen** Struktur gruppieren.

k . . . Knoten
b . . . Blatt

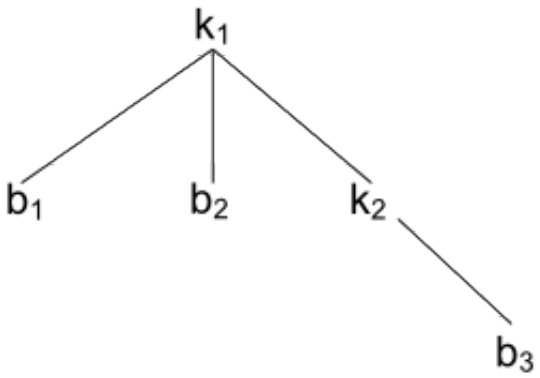

Abbildung 7: Struktur einer Teil-Ganzes-Hierarchie

Lösungsansatz. Das Kompositum-Muster soll es erlauben, dass bei der Verarbeitung von Knoten in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden. Durch den Einsatz des Kompositum-Musters wird es möglich, in einer Baumstruktur **zusammengesetzte 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.

4.1 Klassendiagramm

4.1.1 Teilnehmer

Konten
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 Knotenelement 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 weiter

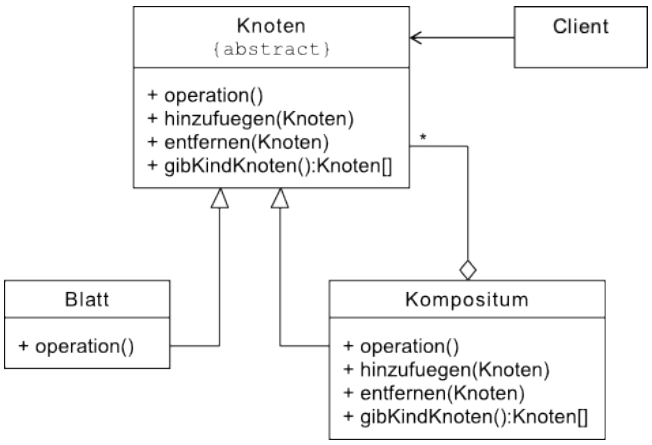

Abbildung 8: Struktur einer Teil-Ganzes-Hierarchie

Knoten aggregieren kann. Die Klasse *Kompositum* implementiert die kindbezogenen Operationen und **überschreibt** damit das Defaultverhalten, das in der Klasse *Knoten* implementiert ist.

4.2 Konkretes Beispiel

```
1  package Composite;
2
3  // Datei: Knoten.java
4  import java.util.ArrayList;
5
6  public abstract class Knoten {
7    private String name = "";
8    static int ebene = 0; // Zaehler fuer Ausgabe-Ebene
9    ArrayList<Knoten> kindelemente = new ArrayList<Knoten>();
10
11   public Knoten(String name) {
12     this.name = name;
13   }
14
15   public abstract void operation();
16
17   public void hinzufuegen(Knoten komp) {
18     System.out.println("Kind-Methode nicht implementiert!");
19   }
20
21   public void entfernen(Knoten komp) {
22     System.out.println("Kind-Methode nicht implementiert!");
23   }
24
25   public void gibKind() {
26     System.out.println("Kind-Methode nicht implementiert!");
27   }
28
29   public String gibName() {
30     return this.name;
31   }
32 }
```

Listing 21: Knoten.java

Definiert die abstrakte Basisklasse Kompositum und Blatt werden davon abgeleitet.

```java
package Composite;

// 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;
  }
}
```

Listing 22: Kompositum.java extends Knoten

Überschreibt die kindbezogene Methoden und implementiert die ausgesuchte Operation operation().

```java
package Composite;

// 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());
  }
}
```

Listing 23: Blatt.java extends Knoten

Repräsentiert eine einfche und nicht zusammengesetzte Knoten einer Baumstruktur und hat im Gegensatz zur Klasse Kompositum **keine** untergeordneten Knoten.

4.3 Vor- und Nachteile

4.3.1 Vorteile

- 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 Kompositum-Muster erlaubt es, verschachtelte Strukturen auf einfache Weise zu erzeugen bzw. um neue Blatt- und Kompositum-Klassen zu erweitern.

4.3.2 Nachteile

- Das Design und der Aufbau der Baumstruktur werden unübersichtlich, wenn man viele unterschiedliche Blatt- und Kompositum-Klassen verwendet.
- Beim Kompositum-Muster werden alle Knoten gleich behandelt → nicht gut, wenn bei dem Aufbau einer Kompositum-Struktur gewissen Einschränkungen unterliegen soll. Diese Einschränkungen müssen durch Typüberprüfungen zur Laufzeit gewährleistet werden.
- Sobald Änderungen and der Basisschnittstelle vorgenommen werden, müssen potentielle alle davon abgeleiteten Klassen ebenfalls geändert werden.