Iterator & Composite Entwurfsmuster

Alexander Rausch

11. Januar 2009

Betreuer

Prof. Dr. Wolfgang Henrich

Prof. Dr. Burkhardt Renz

Seminar

Entwurfsmuster WS 08/09

Fachhochschule Gießen-Friedberg

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 11. Januar 2009

„Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“

Christopher Alexander: A Pattern Language, 1977

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Diese Ausarbeitung ist im Rahmen des Hauptseminars „Entwurfsmuster“ im Wintersemester 08/09 einstanden. Sie beschreibt und erläutert die Entwurfsmuster Iterator und Composite. Als Quellen wurden die Bücher „Entwurfsmuster von Kopf bis Fuß“ von Eric & Elisabeth Freeman [1] sowie der Klassiker über Entwurfsmuster, das Buch der Gang of Four „Entwurfmuster - Elemente wiederverwendbarer objektorientierter Software“ [2], benutzt.
Beim Entwurf von objektorientierter Software gibt es eine Vielzahl von Problemen und Hindernissen, die eine Architektur meistern muss, damit sie in einem wiederverwendbaren, erweiterbaren und wartbaren Code endet. Oft werden Funktionen an anderer Stelle, wie zum Beispiel in neuen Projekten, benötigt, die bereits in älteren Projekten umgesetzt wurden. Genauso häufig müssen Teile einer bestehenden Software neuen Anforderungen gerecht werden, so dass diese aktualisiert oder erweitert werden müssen.
Entwurfsmuster helfen dabei Software zu produzieren, die gut strukturiert, flexibel, erweiterbar und wiederverwendbar ist. Es sind Kochrezepte und Ideen für gute objektorientierte Programmierung. Sie wurden von vielen erfahrenen Entwicklern getestet, ausprobiert, weiterentwickelt und für stabil befunden. Die Muster liefern Lösungen für Probleme, die beim Entwerfen von objektorientierter Software immer wieder auftreten und lassen sich flexibel in eigene Architekturen einbauen. Sie helfen dabei, nicht jedes Mal „das Rad neu erfinden zu müssen“.
Diese Ausarbeitung versucht das Iterator und das Composite Muster Schritt für Schritt zu erklären. Am Ende des theoretischen Teils eines Kapitels wird der Einsatz des Musters in einem Beispiel konkret gezeigt.
Beide Beispiele nehmen Bezug auf eine fiktive Hotelmanagement-Software, die um Module für eine Belegungs- und Veranstaltungsübersicht, sowie die Personalverwaltung und eine Personalübersicht erweitert werden soll.

2 Das Iterator Entwurfsmuster

2.1 Motivation

Es gibt viele Datenstrukturen, die es ermöglichen, Sammlungen (sogenannte Collections) von Objekten zu verwalten. Als Beispiel werden hier das einfache Array und sämtliche Klassen des JAVA Collection Framework genannt wie ArrayList, Vector und HashTable. Werden diese Datenstrukturen genutzt, um Sammlungen von Objekten in seinen Klassen zu realisieren, kommt es häufig vor, dass ein Client mit diesen Objekten arbeiten möchte oder aus irgendwelchen Gründen eine Funktion benötigt wird, die diese Sammlungen durchlaufen muss. Klassen, die viele Objekte einer anderen Klassen enthalten, werden auch als Aggregate bezeichnet. Die einzelnen Objekte einer solchen Sammlung heißen Elemente.

Beispiel
Ein Raumverwaltungsmodul, das Räume enthält, ist ein Aggregat und die einzelnen Räume sind die Elemente.

Guter Stil ist es, wenn der Client auf die Aggregatklasse zugreifen kann, ohne dessen interne Struktur zu kennen. Dazu sollte er die Möglichkeit haben, auf verschiedene Arten die Collection durchlaufen zu können. Wenn der Client zum Durchlaufen verschiedener Collections, die im Programmkontext für die gleiche Sache benutzt werden, den gleichen Code nutzen möchte, ist es sinnvoll, wenn er anstelle der konkreten Typen der Aggregate nur ein Interface kennen muss, das ihm Methoden zum Traversieren zu Verfügung stellt. Um diesen Anforderungen gerecht zu werden, kann das Iterator-Muster benutzt werden.

2.2 Definition und Zweck

> „Das Iterator-Muster bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Objekt sequenziell zuzugreifen, ohne die zu Grunde liegende

Implementierung zu offenbaren.“[1]

Das Iterator Muster gehört zur Gruppe der objektbasierten Verhaltensmuster. Diese beschreiben die Kommunikation zwischen Objekten und komplexen Kontrollflüssen. Das Iterator Muster bietet Kontrolle über die Traversierung einer Datenstruktur mit zusammengesetzten Objekten. Es bietet einem Client (Anwender) die Möglichkeit, Aggregate zu durchlaufen und auf seine Elemente zuzugreifen, ohne das Innenleben bzw. die konkrete Implementierung der Collection zu kennen. Das bedeutet, dass der Client mit einfachen Mitteln eine Aggregatklasse durchlaufen kann, ohne sich um den Traversierungsalgorithmus oder die interne Datenstruktur der Collection zu kümmern.
Das Implementieren des Iterationsmechanismus in der Aggregatklasse würde den Code unnötig aufblähen. Die Aggregatklasse soll vorwiegend ihrer Verantwortlichkeit nachkommen, Sammlungen von Objekten zu verwalten und nicht als Iterationsverwalter zu fungieren. Deswegen schreibt das Interator Muster vor, diese Funktionalität in ein externes Iterator Objekt zu entkoppeln. Damit befolgt es das objektorientierte Entwurfsprinzip der Hohen Kohäsion.
Kohäsion ist ein Maß dafür, wie nah eine Klasse dem Ideal kommt, Methoden für nur einen bestimmten Zweck zu implementieren. Weiterhin sieht das Muster vor, das alle Iterator - und Aggregatklassen ein bestimmtes Interface erfüllen. Hier werden zum einen für alle Iteratorklassen Methoden festgelegt, die ein Client benötigt, um eine Collection zu durchlaufen, sowie Methoden für Aggregate, die einem Client das richtige Iterator-Objekt liefern. Das ermöglicht dem Client unter anderem eine polymorphe Iteration. Er kann mit gleichem Code verschiedene Collections durchlaufen. In seinem Code werden das Iterator Interface und das Interface für die Aggregatklassen genutzt. Somit kann zur Laufzeit entschieden werden, welches konkrete Aggregat durchlaufen werden soll.

2.3 Struktur

Im nachfolgenden Klassendiagramm wird die Struktur des Iterator Musters dargestellt. Die einzelen Klassen werden im Folgenden erläutert. Die Klassen- und Methodennamen wurden hier bewusst in Englisch gehalten, da sich diese Arbeit an der Sprache Java orientiert. In Java gibt es das Interface java.util.Iterator, welches die gleichen Bezeichner nutzt.

[1] Entwurfsmuster von Kopf bis Fuß, Seite 336

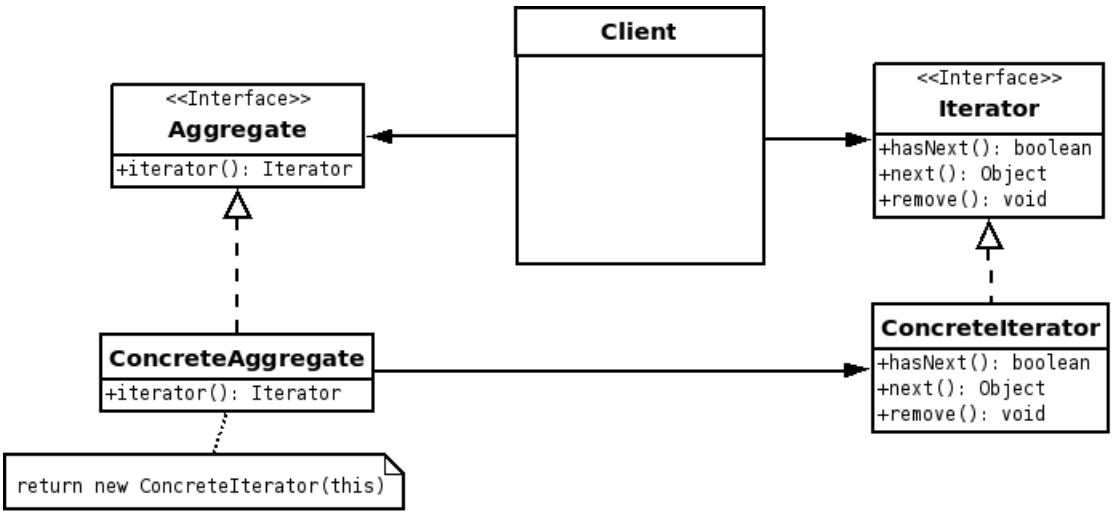

Abbildung 2.1: Struktur des Iterator Musters

2.4 Teilnehmer

Iterator

Das Iterator Interface legt Methoden fest, die ein Client benötigt, um eine Collection zu traversieren. Alle ConcreteIterator Klassen müssen dieses Interface implementieren.

ConcreteIterator

Die Klasse ConcreteIterator implementiert eine konkrete Traversierungsart zu einer Aggregatklasse. Diese Klasse enthält die Logik für das Iterieren der Collection und muss sich die Position merken, in der sich die Iteration gerade befindet.
Die Methode hasNext() liefert true, falls ein weiteres Element in der Collection vorhanden ist, das noch nicht durchlaufen wurde, andernfalls false. Mit dem Aufruf von next() erhält der Client das aktuelle Element und die Position wird um eins inkrementiert. Manchmal ist es sinnvoll ein Element zu löschen. Die Methode remove() entfernt das aktuelle Element aus der Collection.
Es können durchaus mehrere Iteratoren für ein Aggregat existieren. Zum Beispiel einen der die Collection vorwärts und einen der sie rückwärts durchläuft.

Aggregate

Das Aggregate Interface bietet dem Client eine Schnittstelle für die Aggregatklassen. Er ist hierdurch von der Implementiert entkoppelt und kommuniziert mit den Collections nur über dieses Interface. Die einzigste Vorgabe dieser Schnittstelle ist es, dass alle kon-

kreten Aggregate eine Methode iterator() implementieren müssen.

ConcreteAggregate
Die Klasse ConcreteAggregate enthält eine Collection von Objekten. Sie implementiert die Methode iterator() und liefert damit dem Client eine Instanz der passenden ConcreteIterator Klasse. Mit ihr kann der Client über die Collection iterieren.

Client
Der Client muss die beiden Schnittstellen Aggregate und Iterator kennen. Er befragt seine Aggregate nach einem Iterator Objekt und traversiert hiermit die Collections. Durch das Iterator Interface kann er zur Laufzeit die Traversierungsart wechseln und ist vor Änderungen im Aggregat geschützt. Sollte die Datenstruktur der Collection einmal geändert werden, braucht er seinen Code nicht mehr zu überarbeiten. Die Logik für die Iteration übernimmt das ConcreteIterator Objekt.

2.5 Externe und interne Iteratoren

Es gibt zwei Arten von Iteratoren. Diese unterscheiden sich einzig und allein in der Steuerung der Traversierung. Bei externen Iteratoren kontrolliert der Client die Iteration. Er muss die aktuellen Elemente erfragen und schaltet auf das nächste Element weiter.
Interne Iteratoren traversieren eine Collection automatisch und erhalten vom Client die gewünschte Operation übermittelt. Ein Client mit externen Iteratoren ist etwas flexibler. Er muss jedoch die Iteration selbst steuern.
Im Gegensatz dazu ist er mit internen Iteratoren auf bestimmte Operationen, wie zum Beispiel die Ausgabe einer Liste, beschränkt und hat keine Kontrolle. Er bekommt die Ausgabe als fertiges Ergebnis.

2.6 Sonderfall Nulliterator

Ein Nulliterator liefert in der Methode hasNext() immer ein false zurück. Manchmal lässt sich Code besser schreiben, wenn eine Aggregatklasse, die eine Pseudo Liste enthalten sollen, eine Iteration zurückliefert, die nichts durchläuft. Dies wird beispielsweiße im Composite Entwurfsmuster genutzt. Da die Blätterklassen keine Kinder enthalten können, wird hier ein Nulliterator zurückgegeben. In Kapitel 3.6.3 wird das an einem Beispiel gezeigt und auf den Nulliterator nochmals eingegangen.

2.7 Beispiel Belegungs- und Veranstaltungsübersicht

In einer Hotelmanagement-Software soll ein neues Modul namens „HotelOverview“ entstehen, das alle Raumbelegungen und Veranstaltungen anzeigt. Es existiert bereits ein Raumverwaltungsmodul sowie ein Modul für den Veranstaltungskalender. Die Datenstrukturen, in denen die Räume und Veranstaltungen verwaltet werden, sind verschieden. Das neue Modul soll möglichst einfach und dynamisch programmiert werden. Weiterhin sollte es möglichst sicher gegen zukünftige Codeänderungen am Raumverwaltungs- und Veranstaltungskalendermodul sein.
Anhand des Iterator Muster wird exemplarisch gezeigt, wie die theoretischen Grundlagen in einem Beispiel umgesetzt werden können. Ziel ist es, dass ein einheitliches Traversieren von Räumen und Veranstaltungen in den beiden bestehenden Modulen möglich wird.

2.7.1 Räume und Veranstaltungen

Die Räume (Room) und Veranstaltungen (Event) besitzen ein gemeinsames Interface namens ShowInformation. Dieses fordert eine printInfo() Methode, welche Informationen über das jeweilige Objekt geben soll. Ein Raum gibt beispielsweise Informationen über Name, Beschreibung, Raumnummer und Belegungsstatus aus. Veranstaltungen finden in einem Raum an einem bestimmten Datum statt. Wichtige Attribute werden per Konstruktor initialisiert, andere können über entsprechende set-Methoden gesetzt werden.

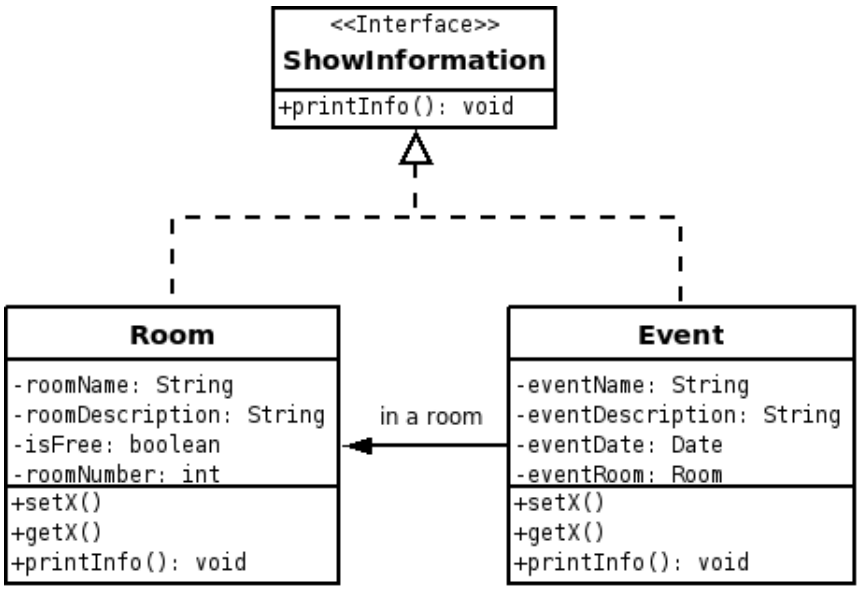

Abbildung 2.2: Klassendiagramm von Raum und Veranstaltung

2.7.2 Modul Raumverwaltung

Das Raumverwaltungsmodul (HotelRoomManagement) speichert Räume in einem klassischen Array ab. Da im Hotel nur begrenzt Platz vorhanden ist, können nur eine bestimmte Anzahl an Räumen hinzugefügt werden. Die maximale Anzahl wird per Konstruktor übergeben. Zum Hinzufügen und Löschen von Räumen bietet das Modul die Methoden addRoom() und removeRoom() an.
Zugriff auf alle Räume ermöglicht die Methode getRooms().

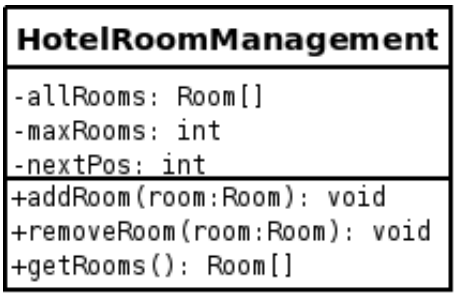

Abbildung 2.3: Klassendiagramm des Raumverwaltungsmoduls

2.7.3 Modul Veranstaltungskalender

Im Veranstaltungskalendermodul (HotelEventCalendar) werden Veranstaltungen als Schlüssel / Wert Paar in einer HashMap gespeichert. Als Schlüssel dient das Datum und als Wert eine ArrayList mit allen Veranstaltungen an diesem Datum. Außerdem besitzt das Modul noch Methoden zum Hinzufügen, Löschen und Abrufen von Veranstaltungen.

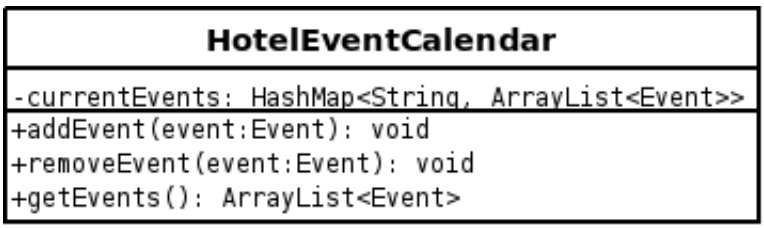

Abbildung 2.4: Klassendiagramm des Veranstaltungskalendermoduls

2.7.4 HotelOverview Modul ohne Iterator

Das neue HotelOverview Modul kann sich alle Informationen, die es benötigt, aus vorhandenem Code herrausholen. In zwei Schleifen kann es von jedem Raum- und Veran-

staltungsobjekt die Methode printInfo() aufrufen. Diese gibt alle Informationen über Raum oder Veranstaltung aus.
Nachfolgend ein Stück Code, der zeigt wie ein schneller Implementierungsversuch der Methode, die eine Übersicht über alle Räume und Veranstaltungen erstellt, aussehen könnte.

Listing 2.1: Implementierung des neuen Moduls ohne Iterator

```
1  public void showOverview() {
2    /* Array für Räume und ArrayList der Events erstellen */
3    Room[] allRooms = roomManagement.getRooms();
4    ArrayList<Event> allEvents = eventCalendar.getEvents();
5
6    System.out.println("------- AKTUELLE RAUMBELEGUNG -------");
7    /* Alle Räume anzeigen */
8    for(int i=0; i < allRooms.length; i++) {
9      if(allRooms[i] != null) {
10       allRooms[i].printInfo();
11     }
12   }
13
14   System.out.println("---------- VERANSTALTUNGEN ----------");
15   /* Alle Veranstaltungen anzeigen */
16   for(int i=0; i < allEvents.size(); i++) {
17     if(allRooms[i] != null) {
18       allEvents.get(i).printInfo();
19     }
20   }
21 }
```

Nachteile

Das Modul muss wissen, welche Datenstrukturen für die Räume und Veranstaltungen benutzt werden. Es benötigt zwei Schleifen um diese zu durchlaufen, da sich die Collections nicht einheitlich behandeln lassen. Der Code muss aktualisiert werden, wenn sich in der Raum- oder Veranstaltungsverwaltung die Datenstruktur der Collection ändert oder der Zugriff auf Collection geändert wird.
Wenn in Zukunft eine neue Übersicht hinzukommt, muss der Code um weitere Schleifen erweitert werden. Die komplette Logik der Iteration übernimmt der Client.

Die Funktionalität, welche sich im obigen Code verändert, ist die Iteration. Kann diese ausgelagert werden, ist das Modul in der Lage die Raum- und Veranstaltungsverwaltung einheitlich zu durchlaufen. Für genau diesen Zweck gibt es das Iterator Entwurfsmuster.

2.7.5 HotelOverview Modul mit Iterator

Nachfolgend die Struktur von Hotel-Overview unter Verwendung des Iterator Musters. Ein neues Interface namens HotelSoftwareComponent arbeitet als Schnittstelle zwischen den Modulen und Aggregateklassen. Raum- und Veranstaltungskalendermodul agieren als ConcreteAggregate Klassen und implementieren die iterator()-Methode. Mit iterator() liefern sie der HotelOverview Klasse die konkreten Iteratoren RoomManagementIterator und EventCalendarIterator. Diese beiden ConcreteIterator Klassen implementieren das Interface Iterator.
Sie besitzen die typischen Methoden wie hasNext() und next() und bieten HotelOverview eine komfortable Möglichkeit, die Elemente (Räume und Veranstaltungen) in HotelEventCalendar und HotelRoomManagement zu traversieren.

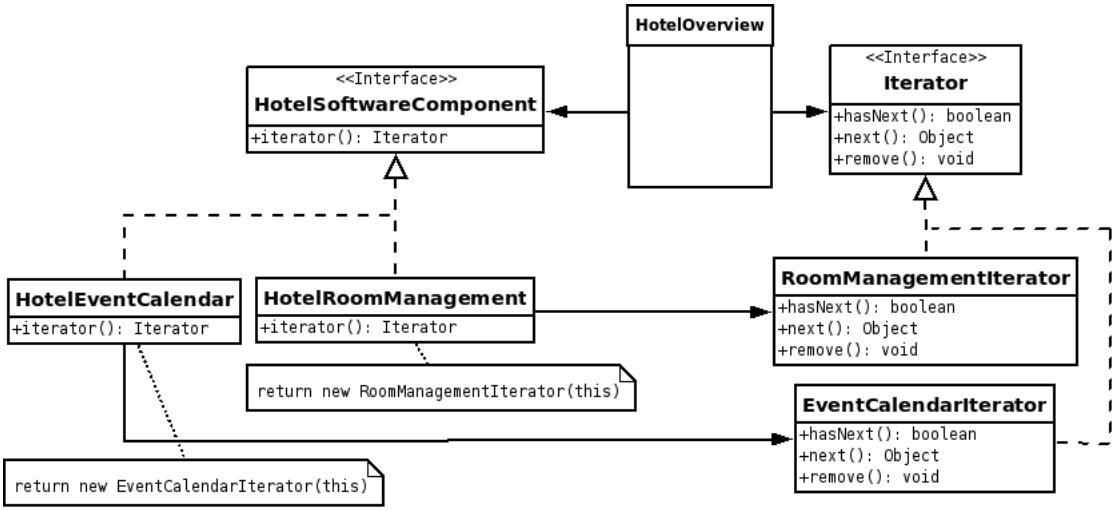

Abbildung 2.5: Klassendiagramm des neuen Moduls mit Iterator

Mithilfe des gemeinsamen Interfaces ShowInformation, welches die Methode printInfo() fordert, kann die Klasse HotelOverview die Übersicht sehr elegant ausgeben, da die Klassen Event und Room diese Schnittstelle implementieren.

Listing 2.2: Implementierung des neuen Moduls mit Iterator

```
1 /**
2  * Zeigt eine Informationsübersicht über Räume und Veranstaltungen an
3  * Mit Verwendung des Iterator Musters
4  */
5 public void showOverview() {
6   System.out.println("------- AKTUELLE RAUMBELEGUNG -------");
7   /* Alle Veranstaltungen anzeigen */
```

```
8     getComponentInfo(roomManagement);
9
10    System.out.println("---------- VERANSTALTUNGEN ----------");
11    /* Alle Veranstaltungen anzeigen */
12    getComponentInfo(eventCalendar);
13
14  }
15
16  /**
17   * Hilfsmethode welche die Informationen einer HotelSoftwareComponent ausgibt
18   * @param hc
19   */
20  private void getComponentInfo(HotelSoftwareComponent hc) {
21    /* Passenden Iterator des Aggregats holen */
22    Iterator componentIterator = hc.iterator();
23
24    /* HotelSoftwareComponent traversieren mit dem passenden Iterator */
25    while(componentIterator.hasNext()) {
26      /* Interface benutzen, um Räumen und Veranstaltungen gleich behandeln zu können */
27      ShowInformation showInformationClass = (ShowInformation) componentIterator.next();
            /* Liefert nächstes Element*/
28      /* Aufruf von printInfo() von einem Raum oder Veranstaltung */
29      showInformationClass.printInfo();
30    }
31  }
```

Vorteile

Im obigen Beispiel bleibt das Hotel-Overview Modul flexibel und wartbar. Es kann Module, die die Schnittstelle HotelSoftwareComponent implementieren, immer auf die gleiche Art und Weiße durchlaufen. Änderungen an Modulen bedeuten nicht mehr, dass der Code korrigiert werden muss. Es kann davon ausgehen, immer den richtigen Iterator geliefert zu bekommen um eine Traversierung durchzuführen.

2.8 Verwendungsbeispiele und verwandte Muster

Das Iterator Muster ist in den meisten objektorientierten Programmiersprachen zu finden. In den Bibliotheken für Behälterklassen erleichtern Iteratoren das Traversieren von Arrays, Vectoren und HashTables. In Java gibt es das Interface java.util.Iterator. Zu jeder Datenstruktur aus dem Java Collection Framework gibt es passende Iteratoren die dieses Interface implementieren.
In C# kann ein Iterator mit der Schnittstelle System.Collections.IEnumerator und der darin geforderten Methode GetEnumerator() realisiert werden. Klassen, die eine GetEnumerator()-Methode implementieren, können mit der foreach-Anweisung traversiert werden.
Im Composite Entwurfsmuster wird ein Iterator zum Durchlaufen der Kinder eines Kno-

tens benutzt. Zum Speichern der Schritte einer Iteration kann das Mementomuster benutzt werden. Mit diesem können die Zustände eines Objektes zwischenspeichern werden.

2.9 Vor- und Nachteile

Ein Iterator bietet einem Client die Möglichkeit, die Elemente eines Aggregates sequentiell traversieren zu können. Er muss keinerlei Kenntnisse über die Implementierung der Aggregatklasse besitzten. Außer den zwei Schnittstellen Aggregate und Iterator müssen ihm keine weiteren Klassen und Interfaces bekannt sein. Die Vorgabe, die Logik für die Iteration in einer externe Klasse zu implementieren und somit aus dem Aggregat herauszuhalten, fördert das Prinzip der hohen Kohäsion.
Nachteile, welche durch das Iterator Muster entstehen, sind lediglich erhöhte Laufzeit- und Speicherkosten, die durch zusätzliche Klassendefinitionen und der Polymorphie entstehen. Diese dürften aber nur in zeit- und speicherkritischen Systemen eine tragende Rolle spielen.

3 Das Composite Entwurfsmuster

3.1 Motivation

Grafiken, Homepages und Benutzeroberflächen sind meistens aus verschiedenen Komponenten zusammengesetzt. Diese Komponenten können weitere Komponenten beinhalten und lassen sich so zu immer komplexeren zusammensetzten.
Bilder können aus Linien, Kreisen und Rahmen bestehen. HTML Dokumente setzten sich aus Tags zusammen und bauen sich zu einem DOM[1]-Baum zusammen. Viele Tags können beliebig geschachtelt werden. Benutzeroberflächen in Java Swing lassen sich aus einer Vielzahl von Komponenten wie Panels, Buttons und Labels zusammenbauen. Ein Dateisystem enthält Dateien und Verzeichnisse. Dabei können Verzeichnisse viele weitere Unterverzeichnisse und Dateien enthalten.
Solche zusammengesetzten Komponenten bestehen grundsätzlich aus primitiven Klassen und Behälterklassen. Primitive Klassen können keine weiteren Objekte enthalten. Die Behälterklassen sind jedoch in der Lage, viele andere primitive Klassen und weitere Behälterklassen zu beinhalten. Ein Bild besteht aus vier Linien. Eine Linie kann aber keine Bilder enthalten.
Oft sollen die gleichen Operationen auf beide Klassentypen angewendet werden. Das könnte in dem vorherigen Beispiel eine Methode mit dem Namen draw() oder print() sein. Beispielsweise soll eine Grafik ihre Bestandteile wie ein Rechteck und ein Dreieck auf den Bildschirm zeichnen, wenn die Methode draw() aufgerufen wird.
Das Composite Muster zeigt, wie man Teil-Ganzes-Hierarchien von Objekten zusammenbauen kann. Der Client kann eine Operation rekursiv auf einen ganzen oder einen Teil des Baumes anwenden, ohne zwischen primitiven und Behälterklassen unterscheiden zu müssen.

[1] DOM=Das Document Object Model stellt die Struktur eines XML-Dokuments in einer Baumstruktur dar.

3.2 Definition und Zweck

„Das Composite-Muster ermöglicht es Ihnen, Objekte zu einer Baumstruktur zusammenzusetzen, und Teil-Ganzes-Hierachien auszudrücken. Das Composite-Muster erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.“[2]

Das Composite Muster gehört zur Gruppe der objektbasierten Strukturmuster. Diese fassen Objekte zu größeren Strukturen zusammen. Mit dem Composite Muster kann ein Client Objekte zu Bäumen zusammensetzen, welche eine Teil-Ganzes-Hierarchie repräsentieren. Er kann mit Teilen des Baumes oder mit dem Baum als Ganzes arbeiten. In einem Dateisytem, welches in einer Baumstruktur realisiert ist, können beispielsweise alle Dateien und Verzeichnisse ab dem Rootverzeichnis rekursiv aufgelistet werden. Genauso ist es möglich nur einen Teil, also ein bestimmtes Verzeichnis inklusive Dateien und Unterverzeichnisse, aufzulisten. Häufig wendet der Client die gleichen Operationen auf Klassen in solch einer Struktur an. Mit dem Composite Muster können alle zusammengesetzten und einzelnen Objekte im Baum gleich behandelt werden.
Hier heißt eine primitive Klasse Blatt (Leaf) und eine zusammengesetzte Kompositum (Composite). Sie bilden die Komponenten in einem Baum.
Die grundlegende Idee besteht darin, eine abstrakte Klasse namens Component zu erschaffen, die als gemeinsame Schnittstelle für Blätter und Komposita dient. Sie besitzt alle Methoden die in einem Blatt und einem Kompositum vorkommen. Neben den gemeinsamen Methoden befinden sich auch Methoden für die Kindverwaltung im Kompositum. Der Client benutzt zum Aufbauen der Baumstruktur nicht den Typ eines Blattes oder Kompositums, sondern benutzt den der abstrakten Komponentenklasse.

3.3 Struktur

Im nachfolgenden Klassendiagramm wird die Struktur des Composite Musters dargestellt. Die einzelen Klassen werden im Folgenden erläutert.

[2]Entwurfsmuster von Kopf bis Fuß, Seite 356

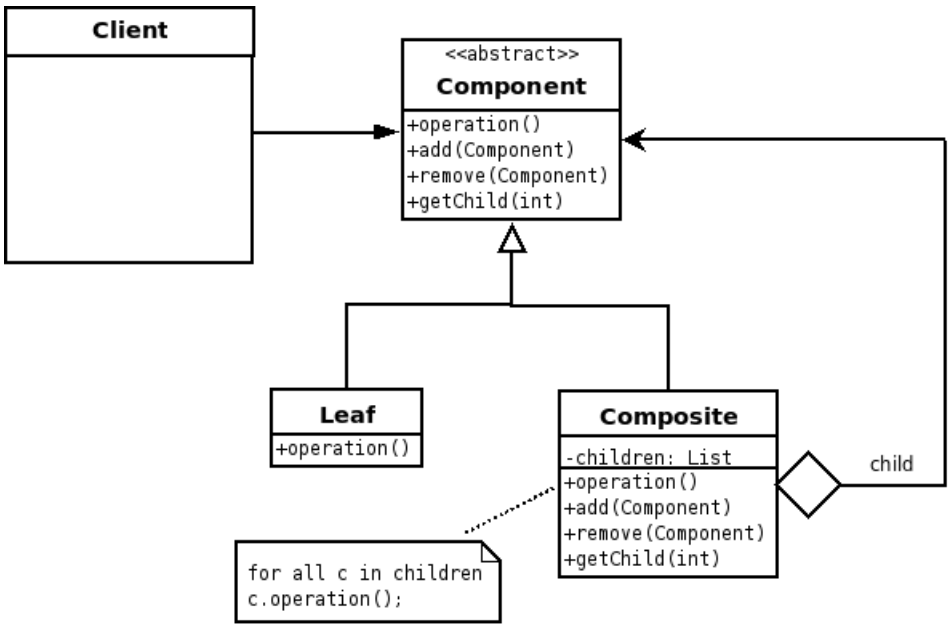

Abbildung 3.1: Struktur des Composite Musters

3.4 Teilnehmer

Component
Die abstrakte Klasse Component dient als Interface zwischen Client und Baumstruktur. Sie deklariert Methoden zur Kindverwaltung sowie alle Methoden die gemeinsam im Leaf und Composite genutzt werden. Für die gemeinsamen Methoden implementiert sie ein Defaultverhalten.

Composite
Die Klasse Composite repräsentiert eine Behälterklasse. Sie enthält eine Liste von Referenzen auf die Kinder (Leafs) und implementiert die Verwaltungsfunktionen add(Component), remove(Component) und getChild(int). In den gemeinsamen Methoden (hier die Methode operation()) führt sie ihren eigenen Code aus und ruft rekursiv von jedem Kind die gleiche Methode auf. Ist ein Kind eine weitere Composite Klasse, ruft diese wiederum die Methode bei allen Kindern auf usw. Für das Durchlaufen der Kinder wird oft das Iterator Muster benutzt (siehe Beispiel).

Leaf
Die Klasse Leaf repräsentiert ein primitives Blattobjekt. Sie kann keine Kinder enthal-

ten und implementiert das gewünschte Blattverhalten der gemeinsamen Methoden. (Im Klassendiagramm mit operation() angedeutet)

Client
Der Client baut mit der Schnittstelle der abstrakten Component Klasse die Baumstruktur auf. Er benutzt in seinem Code Component als Typ für Leaf und Composite Klassen.

Beispiel
Component c1 = new Composite();
Component c2 = new Leaf();
c1.add(c2);
Er kann die Methode operation() auf Leaf und Composite gleich anwenden:
c1.operation();
c2.operation();

3.5 Transparenz vs. Sicherheit

Da es in der Schnittstelle der Component Klasse Methoden gibt, die für eine Leaf Klassen nicht unbedingt sinnvoll erscheinen, kann sich die Frage stellen, ob die Methoden zur Kindverwaltung nicht lieber in der Composite Klasse deklariert werden sollten. Schließlich soll eine Klasse ein hohes Maß an Kohäsion besitzen und nur für wenige Dinge verantwortlich sein.
Im Beispiel des Composite Musters sollte jedoch ein Kompromiss eingegangen werden. Der Client hat es einfacher, wenn er in der Component Klasse alle Methoden sieht, die ihm zu Verfügung stehen. Er benötigt nur dieses eine Interface, um mit Leaf und Composite Klassen arbeiten zu können, und muss nicht zur Laufzeit entscheiden, welche Klasse er vorliegen hat. Die Struktur des Kompositums ist für ihn transparent. Er kann jedoch auch Methodenaufrufe bei Leaf Objekten starten, die sinnlos sind. Das Anhängen von Kindern an ein Leaf Objekt ist schließlich keine sinnvolle Operation. Sieht man eine Leaf Klasse jedoch als Component an, die niemals Kinder haben kann, so sind Aufrufe der Kindverwaltungsmethoden zwar nicht produktiv, können jedoch von der Component Klasse in einer Defaultimplementierung sinnvolle Handlungen bekommen.
Solch eine Defaultimplementierung kann zum Beispiel das Werfen einer Exception sein, die dem Client mitteilt, dass der Aufruf ungültig ist. Grundsätzlich ist es sinnvoll in den Defaultimplementierungen erst einmal alle Operationen fehlschlagen zu lassen. Alle Methodenaufrufe die nicht für Blätter bestimmt sind, schlagen nun fehl. Diese Methodik

gibt dem Aufrufer zwar weniger Sicherheit, aber steigert für ihn die Transparenz. Damit der Client Leaf und Composite Objekte gleich behandeln kann, ist die Transparenz meiner Meinung nach wichtiger. Steht die Sicherheit der Methodenaufrufe im Vordergrund oder wird diese als wichtiger erachtet, so müssen die Kindverwaltungsfunktionen in die Composite Klasse verschoben werden. Der Client muss zur Laufzeit beim Hinzufügen und Entfernen von Leaf Objekten bei Kompositas aufpassen, dass er den richtigen Typ verwendet. Er kann aber immer noch gemeinsame Operationen auf die gleiche Art und Weiße auf Leaf und Composite Objekte ausführen.

3.6 Beispiel Personalverwaltung

Die Hotelmanagement-Software soll um ein weiteres Modul namens „EmployeesManagement“ erweitert werden. Das Management und das Personal sollen übersichtlich und einfach verwaltet werden können. Gewünscht ist außerdem ein Modul mit dem Namen „EmployeesOverview“, welches das EmployeesManagement-Modul nutzt und eine Übersicht über alle Personen und Personalgruppen im Hotel ausgibt. Für diesen Anwendungsfall lässt sich wunderbar das Composite Muster einsetzen. Die Beziehung zwischen Personen und Personalgruppen lässt sich am besten in einer Baumstruktur darstellen. Eine gemeinsame Operation von der Person- und Personalgruppenklasse kann eine zur Ausgabe des Namens sein.

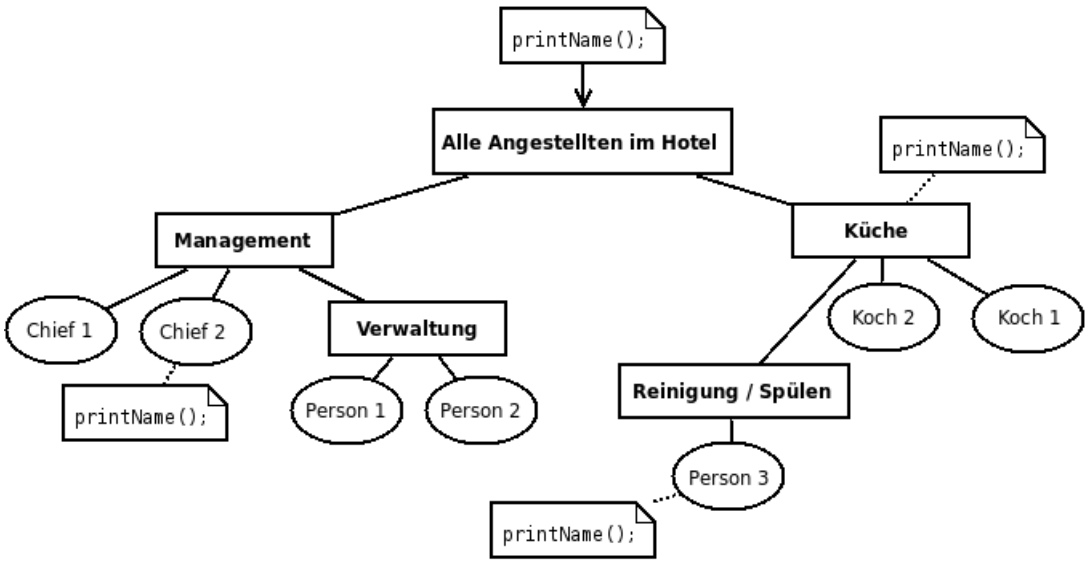

Abbildung 3.2: Entwurf der Architektur des Moduls EmployeesManagement

3.6.1 Component-Klasse EmployeesManagementComponent

Damit der Client, also das EmployeesManagement-Modul, eine Teil-Ganzes Hierarchie mit Personen und Personalgruppen aufbauen kann, benötigt er die Schnittstelle zu den Leaf und Composite Objekten. Hier sind alle gemeinsamen Methoden und Kindverwaltungsmethoden des Composite deklariert. Es gibt zwei gemeinsame Methoden. Die Methode printName() soll den Namen der Person bzw. der Personalgruppe ausgeben. Durch iterator() erhält der Client einen passenden Iterator für ein Composite-Objekt. Dieses wird später dazu benutzt, um den kompletten Baum traversieren zu können.

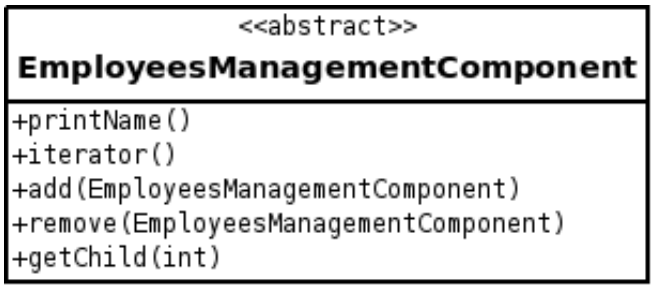

Abbildung 3.3: Klassendiagramm der abstrakten Klasse EmployeesManagementComponent

3.6.2 Composite-Klasse Personalgruppe

Personalgruppen (Group) können Personen enthalten und aus weiteren Untergruppen bestehen. Die Gruppe Küche kann eine weitere Untergruppe Reinigung / Spülung besitzen. Dies wird möglich, indem die Klasse Group ein Composite repräsentiert. Als Datenstruktur für Kinder wird eine ArrayList benutzt. In der Methode printName() wird zuerst der Gruppenname ausgegeben und anschließend die gleiche Methode bei allen Kindern rekursiv aufgerufen. Dies führen alle Untergruppen ebenfalls durch. Ein Aufruf von printName() in der Wurzelgruppe erzeugt somit eine Ausgabe von allen Personalgruppen und den darin enthaltenen Personen. Zum Durchlaufen der Kinder wird das Iterator Muster benutzt.
Dies ist ein Beispiel für einen internen Iterator. Der Client ruft printName() auf und stößt automatisch eine Iteration an, ohne jegliche Kontrolle über sie zu erhalten.

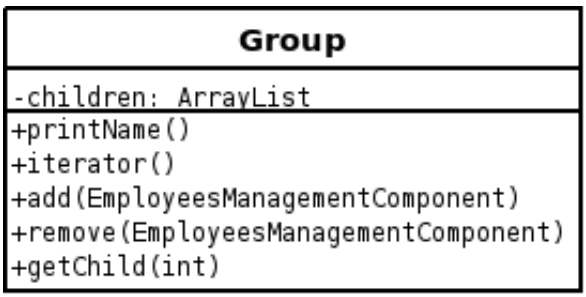

Abbildung 3.4: Klassendiagramm der Group-Klasse

Listing 3.1: Implementierung der Klasse Group

```
1  /* Erbt von der abstrakten Klasse EmployeesManagementComponent*/
2  public class Group extends EmployeesManagementComponent {
3    /** Gruppenname **/
4    private String groupName;
5
6    /** ArrayList mit den Kindern **/
7    private ArrayList<EmployeesManagementComponent> children;
8
9    /** Gruppenname und ArrayList initialisieren
10    * @param groupName - Name der Gruppe
11    **/
12   public Group(String groupName) {
13     this.groupName = groupName;
14     this.children = new ArrayList<EmployeesManagementComponent>();
15   }
16
17   /**
18    * Fügt eine neue Personalverwaltungs-Komponennte dem Baum hinzu.
19    * @param emc - Eine EmployeesManagementComponent
20    */
21   public void add(EmployeesManagementComponent emc) {
22     if(children instanceof ArrayList) {
23       children.add(emc);
24     }
25   }
26
27   /**
28    * Löscht eine bestimmte Personalverwaltungs-Komponennte aus dem Baum.
29    * @param emc - Eine EmployeesManagementComponent
30    */
31   public void remove(EmployeesManagementComponent emc) {
32     if(children instanceof ArrayList) {
33       children.remove(emc);
34     }
35   }
36
37   /**
```

```
38      * Liefert das Kind an einem bestimmten Index.
39      * @param i - Index des Kindes
40      * @return Kind welches sich am Index i befindet
41      */
42     public EmployeesManagementComponent getChild(int i) {
43       return children.get(i);
44     }
45
46     /**
47      * Liefert den Gruppennamen und ruft die Methode getName() von allen
48      * Kindern auf.
49      */
50     public void printName() {
51       /* Gibt Gruppennamen aus */
52       System.out.println("\t\n >> " + groupName + " <<");
53
54       /* Einen passenden Iterator zur ArrayList holen */
55       java.util.Iterator<EmployeesManagementComponent> childIterator = children.iterator
             ();
56
57       /* Liste mit Kindern durchlaufen und die Methode getName() aufrufen */
58       while(childIterator.hasNext()) {
59         EmployeesManagementComponent emc = (EmployeesManagementComponent) childIterator.
               next();
60
61         emc.printName();
62       }
63     }
64
65     /* Liefert einen passenden Iterator für die Kinder */
66     public Iterator<EmployeesManagementComponent> iterator() {
67       return children.iterator();
68     }
69 }
```

3.6.3 Leaf-Klasse Person

Eine Person (Person) stellt in der Hierarchie eine primitive Klasse dar. Diese Klasse kann keine Kinder enthalten. Sie implementiert in printName() das gewünschte Verhalten für ein Blatt (Leaf). Die Methode iterator() gibt einen Nulliterator zurück. Damit wird sie vom Client wie eine Composite-Klasse benutzt, mit dem Unterschied, dass sie niemals eine Iteration startet. Der Nulliterator liefert als Rückgabewert von hasNext() immer false. Dies erleichtert dem Client Code zu schreiben, der diesen Baum traversiert. Er kann Leaf und Composite Objekte gleich behandeln.

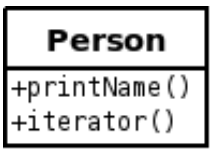

Abbildung 3.5: Klassendiagramm der Person-Klasse

Listing 3.2: Implementierung der Klasse Person

```
1  /* Erbt von der abstrakten Klasse EmployeesManagementComponent*/
2  public class Person extends EmployeesManagementComponent {
3
4    /** Name der Person **/
5    private String name;
6
7    /**
8     * Initialisiert die Person mit ihrem Namen
9     * @param name - Name der Person
10    */
11   public Person(String name) {
12     this.name = name;
13   }
14
15   /**
16    * Gibt den Namen der Person aus
17    */
18   public void printName() {
19     System.out.println(name);
20   }
21
22   @Override
23   /**
24    * Liefet einen Nulliterator zurück, da diese Klasse als Blatt keine weiteren
25    * Kinder haben kann.
26    */
27   public Iterator<EmployeesManagementComponent> iterator() {
28     return new NullIterator();
29   }
30 }
```

3.6.4 EmployeesManagement Modul

Die erstellte Architektur für die Personalverwaltung kann jetzt von einem Client, dem EmployeesManagement-Modul, genutzt werden. Es kann die Baumstruktur mit Hilfe der Kindverwaltungsmethoden Schritt für Schritt aufbauen. Wird versucht einen unsinnigen Methodenaufruf zu starten, wird eine UnsupportedOperationException geworfen.

Listing 3.3: Generieren der Baumstruktur im EmployeesManagement Modul

```java
/**
 * Generiert eine Baumstruktur der Personen im Hotel
 */
public void generateEmployeesTree() {
  EmployeesManagementComponent chief1 = new Person("Person 1");
  EmployeesManagementComponent chief2 = new Person("Person 2");

  EmployeesManagementComponent cook1 = new Person("Person 3");
  EmployeesManagementComponent cook2 = new Person("Person 4");

  EmployeesManagementComponent management = new Group("Hotelmanagement");
  EmployeesManagementComponent kitchen = new Group("Küche");
  EmployeesManagementComponent administration = new Group("Verwaltung");
  EmployeesManagementComponent kitchenCleaning = new Group("Reinigung / Spülen");
  EmployeesManagementComponent helper = new Group("Küchenhilfe");

  try {
    /* Hotelleitung und Verwaltung */
    management.add(chief1);
    management.add(chief2);

    /* eine Untergruppe mit den Personen aus der Verwaltung erstellen */
    administration.add(new Person("Person 5"));
    administration.add(new Person("Person 6"));
    management.add(administration);

    /* Küchenpersonal */
    kitchen.add(cook1);
    kitchen.add(cook2);

    kitchenCleaning.add(new Person("Person 7"));
    kitchenCleaning.add(new Person("Person 8"));

    kitchen.add(kitchenCleaning);

    helper.add(new Person("Person 9"));
    kitchenCleaning.add(helper);

    /* Unsinniger Aufruf welcher eine Exception wirft */
    //chief1.add(chief2);

    /* Wurzelgruppe (Ist Attribut der Klasse) */
    allPersons.add(management);
    allPersons.add(kitchen);
  }
  catch (UnsupportedOperationException e) {
    System.out.println("Unsinniger Methodenaufruf!");
  }
}
```

3.6.5 EmployeesOverview Modul

Der von EmployeesManagement erstellte Personen- und Personalgruppenbaum kann dazu benutzt werden, eine Übersicht über alle Personen und Personalgruppen anzuzeigen. Das Modul EmployeesOverview muss nur eine Referenz auf das Wurzelelement kennen, um die Methode printName() auf den ganzen Baum anzuwenden.

Listing 3.4: Übersicht über Personen und Personalgruppen anzeigen

```
1 /**
2 * Gibt den Baum aus, indem es die Methode printName() auf das oberste Composite
      anwendet
3 */
4 public void printTree() {
5   if(allPersons != null) {
6     /* printName() auf alle Knoten im Baum aufrufen */
7     allPersons.printName();
8   }
9 }
```

3.6.6 Baumstruktur der Personalverwaltung iterieren

Zum Abschluss des Beispiels soll ein weiteres Beispiel für einen externen Iterator gezeigt werden. Im EmployeesOverview-Modul wird noch eine Funktionalität hinzugefügt, die jeweils alle Personen und Personalgruppen zählt. Dazu muss das Modul den kompletten Baum iterieren und überprüfen, ob das aktuelle Baumelement ein Composite oder Leaf Objekt ist.
Ein Iterator für eine Baumstruktur ist nicht ganz trivial. Er muss sich angefangen bei der Wurzel von jedem Composite und Leaf Objekt den passenden Iterator für die Kinder geben lassen. Die Leaf Objekte liefern hierbei einen Nulliterator zurück. Der komplette Baum wird so rekursiv durchlaufen und die Iteratoren von allen Baumknoten werden auf einem Stack zwischengespeichert. Ist eine Iteration von einem Teil des Baumes beim letzten Element angekommen, wird der Iterator vom Stack genommen und der nächste Iterator geladen. Somit werden Schritt für Schritt alle Composite Kinder durchlaufen, bis keine weiteren Iteratoren mehr im Stack vorhanden sind.
Der Code für das untere Listing stammt aus dem Buch Entwurfsmuster von Kopf bis Fuß[3]. Er wurde auf das Beispiel für die Personalverwaltung angepasst.
Dies ist ein typisches Beispiel für einen externen Iterator. Der Client erhält das Werkzeug um die Baumstruktur durchlaufen zu können und kann damit eine beliebige Logik realisieren.

[3]Seite 369

Listing 3.5: Beispiel für einen Composite Iterator

```
1  public class CompositeIterator implements Iterator<EmployeesManagementComponent> {
2    /** Der Stack der die einzelen Iteratoren der Kompositas enthält **/
3    private Stack<Iterator<EmployeesManagementComponent>> stack;
4    /**
5     * Stack initialisieren und Iterator des obersten Kompositums auf den Stack pushen.
6     * @param compositeIterator
7     */
8    public CompositeIterator(Iterator<EmployeesManagementComponent> compositeIterator) {
9      stack = new Stack<Iterator<EmployeesManagementComponent>>();
10     stack.push(compositeIterator);
11   }
12   public EmployeesManagementComponent next() {
13     if(hasNext()) {
14       /* Den obersten Iterator vom Stack holen */
15       Iterator<EmployeesManagementComponent> iterator = stack.peek();
16
17       /* Das nächste Element dieses Iterators holen */
18       EmployeesManagementComponent emc = (EmployeesManagementComponent) iterator.next()
             ;
19
20       /* Falls das Element eine Gruppe ist wird der Iterator dieses Kompositums auf den
             Stack gepusht */
21       if(emc instanceof Group) {
22         stack.push(emc.iterator());
23       }
24       return emc;
25     }
26     return null;
27   }
28   public boolean hasNext() {
29     /* Falls der Stack leer ist gibt es keine weiteren Elemente */
30     if(stack.empty()) {
31       return false;
32     }
33     else {
34       /* Obersten Iterator vom Stack holen */
35       Iterator<EmployeesManagementComponent> iterator = stack.peek();
36       /* Falls dieser Iterator keine weiteren Elemente enthält pope ihn vom Stack*/
37       if(!iterator.hasNext()) {
38         /* Aktueller Iterator ist fertig - diesen vom Stack nehmen */
39         stack.pop();
40         /* Erneut hasNext aufrufen um zu sehen ob es weitere Elemente in anderen
               Iteratoren gibt */
41         return hasNext();
42       }
43       /* Es sind noch Elemente vorhanden */
44       else {
45         return true;
46       }
47     }
48   }
49 }
```

Das EmployeesOverview-Modul kann nun den Personalbaum iterieren und die Anzahl von Personen und Personalgruppen ermitteln. Um festzustellen, ob es sich bei einem Baumknoten um ein Composite Objekt handelt, wurde eine Methode isComposite() als gemeinsame Operation in der EmployeesManagementComponent Schnittstelle hinzugefügt.

Listing 3.6: EmployeesOverview verwendet den Composite Iterator

```
1  public void printPersonCount() {
2    CompositeIterator compositeIterator = new CompositeIterator(allPersons.iterator());
3    int countGroups=0, countPersons=0;
4
5    /* Baum traversieren */
6    while(compositeIterator.hasNext()) {
7      EmployeesManagementComponent emc = compositeIterator.next();
8      //Prüfen ob es sich um ein Composite handelt
9      try {
10       if(emc.isComposite()) {
11         System.out.println("Gruppe");
12         countGroups++;
13       }
14       else {
15         System.out.println("Person");
16         countPersons++;
17       }
18     }
19     catch(UnsupportedOperationException uoe) {
20       continue;
21     }
22   }
23   System.out.println("Anzahl Gruppen: " + countGroups);
24   System.out.println("Anzahl Personen: " + countPersons);
25 }
```

3.7 Verwendungsbeispiele und verwandte Muster

Das Composite Entwurfsmuster taucht beispielsweise im Java AWT / Swing Framework auf. Alle GUI Elemente wie Panels, Buttons und Labels besitzen dort eine gemeinsame Schnittstelle, die abstrakte Komponentenklasse Container. Mit dem Composite Muster ist es möglich, praktisch unendlich viele verschiedene Variationen von grafischen Benutzeroberflächen zusammenzubauen.

Das Decorator Entwurfsmuster sieht von der Struktur her dem Composite Muster sehr ähnlich. Es enthält jedoch nur eine Komponenten Klasse und die Blätterklassen stellen die verschiedenen Funktionalitäten dar.

Das Iterator Muster kann Composite Objekten dabei helfen, ihre Kinder zu traversie-

ren. Außerdem wird ein Iterator dafür eingesetzt, um eine Composite Baumstruktur zu durchlaufen.
Im Command Muster kann das Composite Muster zum Erstellen von aneinanderhängenden Befehlen den sogenannten Makros benutzt werden.

3.8 Vor- und Nachteile

Zu den Vorteilen des Composite Musters zählen unter anderem die Transparenz und Einheitlichkeit, die für den Client geschaffen wird. Leaf und Composite Objekte können gleich behandelt werden. Operationen können auf alle Komponenten in der Baumstruktur angewendet werden. Ob es sich dabei um ein Leaf oder Composite Objekt handelt, bleibt für den Client transparent. Auch das Hinzufügen von neuen Elementen gestaltet sich sehr einfach. Mit den zur Verfügung gestellten Kindverwaltungsmethoden lässt sich der Baum während der Laufzeit manipulieren.
Allerdings kann ein Entwurf, der das Composite Muster verwendet, zu allgemein werden. Können zusammengesetzte Komponenten aus zu vielen Blattkomponenten bestehen, lassen sich die verschiedenen Variationen eines Kompositums nicht einschränken. Die Software kann dynamische Ausmaße annehmen, die eventuell nicht erwünscht sind.

4 Fazit

Beide in dieser Ausarbeitung vorgestellten Entwurfsmuster helfen dabei, bestimmte Probleme sauber zu lösen. Iteratoren ermöglichen es, auf die Elemente einer Collections zuzugreifen, ohne das die Implementierung der Aggregatklasse bekannt sein muss.
Das Composite Muster gibt die Möglichkeit, Teil-Ganzes Hierarchien zu erzeugen, bei denen primitive und zusammengesetzte Komponenten gleich behandelt und Operationen auf diese durchgeführt werde können.
In beiden Fällen wird dem Client die Arbeit erleichtert. Es entstehen Softwaremodule, deren Code wiederverwendbar und dynamisch ist. Entwickler sollten Entwurfsmuster nicht übermäßig verwenden und Software nicht nur noch mit Entwurfsmustern entwerfen. Sie sollten dann eingesetzt werden, wenn es Anforderungen an eine Software gibt, die bestimmte Probleme erahnen lassen oder zukünftige Erweiterungen enthalten können. Entwurfsmuster sind nicht dafür gedacht, überall und für jedes noch so kleine Problem eingesetzt zu werden.
Die Dynamik und Wiederverwendbarkeit, die dort geschaffen wird, kann die Klassenstrukturen unnötig aufblähen. Werden sie aber dort eingesetzt, wo es Sinn macht, werden durch sie jede Menge Arbeit, Kosten und Zeit gespart.

5 Anlagen

In den Anlagen befindet sich eine CD-Rom mit folgenden Dateien:

- Ausarbeitung im PDF Format
- Seminarvortrag zu dieser Arbeit im PDF Format
- Vollständiger Quellcode der Beispiele in Java

Literaturverzeichnis

[1] **Eric Freeman, Elisabeth Freemann u.a.** Entwurfsmuster von Kopf bis Fuß O'Reilly 2006, ISBN 3-89721-421-0

[2] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides** Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software Addison Wesley 1996, ISBN 3-89319-950-0