1 Observer

„Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, das alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden. “

1.1 Beschreibung

Das Observer-Pattern definiert durch die Interfaces `Subject` und `Observer` zwei Rollen denen unterschiedliche Aufgaben zugesprochen werden. Die Rolle Subject benachrichtigt die Rolle Observer, wobei diese sich um die Verarbeitung der Benachrichtigung kümmern muss. In der Regel besteht zwischen den zwei Rollen eine 1-zu-n-Abhängigkeit, sodass mehrere Observer von einem Subject benachrichtigt werden. Damit die jeweiligen Subject die einzelnen Observer kennen und benachrichtigen können, fordert die Rolle Subject die Implementierung zweier Methoden `attach(Obsever`, zum Anmelden und eine `detach(Observer)` zum Abmelden eines Observers. Die Rolle Observer hingegen definiert eine Methode `update(Object)` die zur Aktualisierung eines Observers von einem Subject aufgerufen wird, um diesen zu benachrichtigen. In Abbildung 1 sieht man ein UML-Diagramm mit den zwei Interfaces Subject und Observer. Die beiden konkreten Klassen `ConreteSubject` und `ConreteObserver` implementieren diese zwei Interfaces. Das ConcreteSubject muss neben der Implementierung der drei Methoden attach, detach und notifyObservers auch eine Collection zum Speichern der angemeldeten Observer beinhalten. Ein konkreter Observer muss hingegen nur die Methode update implementieren um auf eine Benachrichtigung entsprechend zu reagieren.

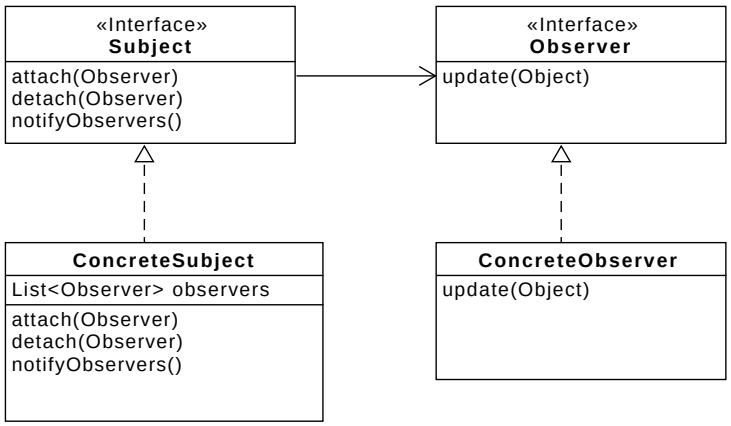

Abbildung 1: UML-Darstellung eines Observer-Pattern.

1.2 Beispiel

Das Beispielszenario behandelt eine fiktive Arbeitsvermittlung deren Softwaresystem mit dem Observer-Pattern umgesetzt wird. In dem Softwaresystem gibt es eine zentrale Komponente JobCenter, welche in regelmäßigen Abständen neue Stellenauschreibungen von externen Arbeitgebern mitgeteilt bekommt. Diese werden auf der Homepage, sowie auf zahllosen Monitoren in der Arbeitsvermittlung aufgelistet. Bei einer neuen Stellenausschreibung soll die Komponente Jobcenter selbständig alle abhängigen Komponenten, wie Homepage und Monitore benachrichtigen, sodass diese sich aktualisieren können. Für dieses Szenario identifizieren wir das Jobcenter als die Rolle des Subjects und die Homepage und Monitore als Observer. Nachfolgend in Listing 1.1 und 1.2 sind diese zwei Rollen als Interfaces umgesetzt. Das Subject definiert neben der Methode `notifyObsevers()`, zum benachrichtigen aller angemeldeten Observer eine Methode `attach(Observer)` und `detach(Observer)` zum An- und Abmelden eines Observers. Das Interface Observer muss hingegen nur zum aktualisieren einer Stellenausschreibung, eine Methode `update(Offer)` bereitstellen. Der Parameter `Offer` symbolisiert eine solche Stellenausschreibung, die über die genannte Update-Methode übergeben wird.

```
1 public interface Observer{
2   void update(Offer aOffer);
3 }
```

Listing 1.1: Observer Interface

```
1 public interface Subject{
2   void attach(Observer aObserver);
3   void detach(Observer aObserver);
4   void notifyObservers();
5 }
```

Listing 1.2: Subject Interface

Im zweiten Schritt wird dann die Klasse Jobcenter (siehe Listing 1.3) implementiert, die von dem Interface Subject erbt und somit die Methoden `attach(Observer aObserver`, `detach(Observer aObserver)` und `notifyObservers()` umsetzt. Zusätzlich initialisiert das Subject eine Collection um die übergebenen Observer über die Attach-Methode zu speichern. Mit der Detach-Methode wird ein Obsever der Collection entfernt. Beim Aufruf der NotifyObserver-Methode wird dieser eine Stellenausschreibung übergeben, die allen gespeicherten Observern über die Update-Methode übergeben wird. Nennenswert ist einmal die Tatsache das das Jobangebot an dieser Stelle nur das Interface Observer kennt und das durch die Iteration ausnahmslos alle Observer benachrichtigt werden.

Als letztes betrachten wir die Klasse `Homepage` (siehe Listing 1.4). Diese übernimmt

```java
public class JobCenter implements Subject {

    List<Observer> observers = new LinkedList<Observer>();

    public void attach(Observer aObserver) {
        mObserver.add(aObserver);
    }
    public void detach(Observer aObserver) {
        mObserver.remove(aObserver);
    }
    public void notifyObservers(Offer aOffer) {
        for(Observer observer : observers){
            observer.update(aOffer);
        }
    }
}
```

Listing 1.3: Jobcenter

die Rolle des Observers indem sie von dem Interface `Observer` erbt und die Methode `update(Offer aOffer)` implementiert.

```java
public class Homepage implements Observer {

    public void update(Offer aOffer) {
         updateHomepage(aOffer);
    }
}
```

Listing 1.4: Observer - Homepage

In Abbildung 2 wird der Ablauf eines solchen Szenarios anhand eines Sequenzdiagramm dargestellt. Zunächst melden sich die Observer `Homepage` und `Monitor` am Jobcenter an. Danach wird das JobCenter aktualisiert und alle angemeldeten Observer benachrichtigt.

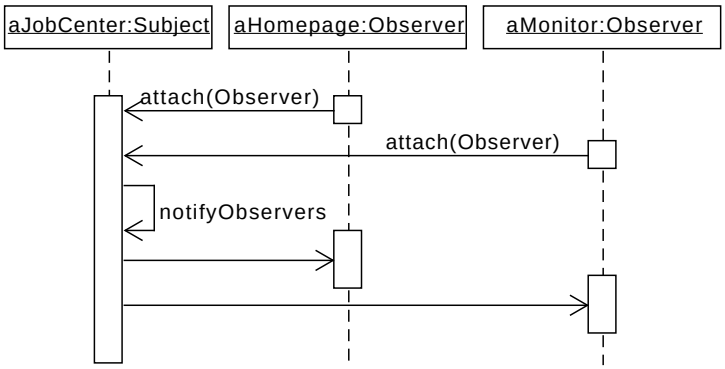

Abbildung 2: Sequenzdiagramm mit dem Ablauf von JobCenter und desssen Clients

1.3 Implementierungsmöglichkeiten

Push- oder Pull-Model Bei den beiden Modellen werden die Unterschiedlichen Arten der Übertragung der Daten betrachtet. Bei dem Push-Model wird bei der Benachrichtigung ein oder mehrere Parameter übertragen. In unserem Fall muss das UML-Diagramm in Abbildung 1 angepasst werden. Die Methode `aktualisieren()` benötigt einen zusätzlichen Parameter. Dieser wird dann bei dem Objekt `KonkretesSubjekt` beim Aufruf der Beobachter übergeben. Der Nachteil dieser Variante ist, das im Extremfall mehrere Parameter übergeben werden, jedoch nicht von jeden Observer benötigt werden. Oder ein neuer Beobachter benötrigt andere Daten wodurch man das Interface `Beobachter` und das Objekt `KonkretesSubjekt` anpassen muss. Die Pull Methode verfolgt einen anderen Ansatz. Hier werden keine Daten als Parameter übergeben sondern die Beobachter greifen direkt nach einer Benachrichtigung auf die Instanz `KonkretesSubjekt` zu um die Daten zu erhalten. Der Nachteil dieser Variante ist, das die einzelnen Beobachter das Subjekt kennen müssen.

Beobachter beobachtet mehr als ein Subjekt Für diesen Fall wird der Beobachter benachrichtigt und kann nicht wissen von welchem Objekt er aufgerufen wird. Abhilfe kann geschaffen werden, indem das konkrete Subjekt seine Instance beim Aufruf übermittelt, sodass der Beobachter eine Fallunterscheidung durchführen kann.

Ausführung des Updates Das Anstoßen der Benachrichtigungen kann entweder vom Client als auch vom konkreten Subjekt durchgeführt werden. Der Vorteil dies dem Konkretem Subjekt zu überlassen ist die Vermeidung von Fehlern, indem der Client keine Benachrichtigung triggert. Falls die Verantwortung dem Client überlassen wird, kann dieser jedoch besser das Update nach Notwendigkeit steuern.

Erweiterung der Benachrichtigung für ein bestimmtes Interesse In bestimmten Fällen kann es sein, das ein Beobachter nur auf bestimmte Events benachrichtigt werden möchte. In diesem Fall bietet sich an, die Methode `registriere()` um einen weiteren Parameter zu erweitern, damit der Beobachter dadurch das Interesse auf ein bestimmtes Event signalisieren kann.

Auslagerung der Verwaltung der Beobachter Wenn die Beziehung zwischen den konkreten Subjekten und den Beobachtern zu komplex wird, empfiel sich die Verwaltung dieser auszulagern. In der Literatur wird von einem ChangeManager gesprochen. Dieser regelt folgende Gebiete: Das Mapping zwischen den Subjekten und den Beobachtern.

Definiert spezielle Updatestrategien. Übernimmt die Benachrichtigung eines konkreten Subjekts für dessen Beobachter.

1.4 Fazit

1.4.1 Vorteile

- Wiederverwendbarkeit: Die Aufteilung von Beobachter und Subjekt sind Rollenbasiert. Ein Objekt kann sowohl ein Subjekt als auch ein Beobachter sein.
- Abstrakte Kopplung von Beobachter und Subjekt: Das Subjekt kennt keinen konkreten Beobachter. (GOF)
- Es muss im Voraus nicht bekannt sein, wie viele abhängige Objekte sich zur Laufzeit registrieren, und welche das sind.

1.4.2 Nachteile

- Gefahr von Zyklen
- Aktualisierungskaskaden bei großen Systemen und somit schwere Fehlersuche Falls sich nicht vom Subjekt abgemeldet, jedoch öfters angemeldet wird. Unerwartete Nebenerffekte

2 Command

„Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen."

2.1 Beschreibung

Das Ziel des Command-Pattern ist, den Requests und Receiver von dem Invoker auszulagern um diese flexibel austauschbar zu gestalten. Hierzu wird eine zusätzliche Schicht eingeführt, nämlich das Command-Objekt. In einem einfachen Szenario, würde der Invoker den Receiver als Objekt besitzen um darauf dessen Methoden aufzurufen. Dadurch ergibt sich allerdings, das der Receiver an den Invoker gebunden ist. Um das dieses Problem zu lösen, führt man auf der Seite des Invokers eine Schnittstelle, `Command` ein, die eine Methode `execute` besitzt. Auf der andere Seite erstellt man ein konkretes Objekt, das von dieser Schnittstelle erbt und gleichzeitig den Receiver kennt, um dort diesen zu

benutzen. Der Invoker muss also nur ein passendes Command-Objekt erhalten und dessen `execute` Methode aufrufen. Die konkrete Durchführung dieser Methode wird dann in dem konkreten Command-Ojbekt behandelt.

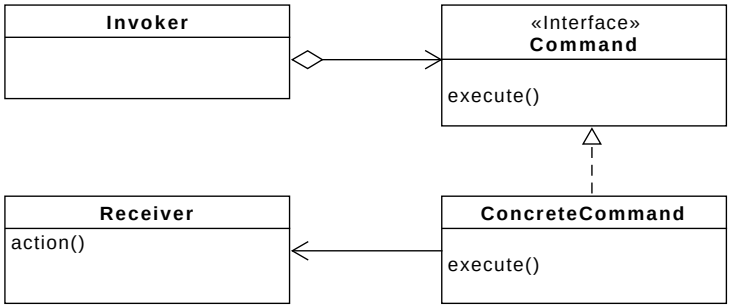

Abbildung 3: Command-Pattern als UML-Diagramm.

2.2 Beispiel

Das Command-Pattern wird anhand eines Lieferdienstes näher beschrieben. Ein Lieferdienst hat grundsätzliche die Aufgabe Sachgegenstände zu versenden. Welche Gegenstände das sind, bzw. an wen diese gesendet werden, ist losgelöst von dem Dienst etwas zu versenden. Möchten wir dieses Szenario implementieren, hätten wir drei Teile: Den Lieferdienst, eine Ausführung und einen Empfänger. Das Command-Pattern gliedert genau diese drei Akteure und stellt als erstes die Ausführung als Interface dar, nämlich dem `Command` (siehe Abbildung 2.1). Diese enthält eine Methode `execute()` die dann ein Invoker, in unserem Fall der `DeliveryService` aufrufen kann.

```
1 public interface Command {
2     void execute();
3 }
```

Listing 2.1: Command Interface

In Listing 2.2 wird der Aufruf des DeliveryService demonstriert. Dieser kennt nur das Interface Command, welches die Execute-Methode besitzt. Welche konkrete Implementierung eines Commands in der Methode `sendObject(Command aCommand)` übergeben und dann ausgeführt wird, ist für den DeliveryService nicht ersichtlich.

Danach werden verschiedene konkrete Commands implementiert, die einerseits den Empfänger kennen und andererseits wissen, wie sie diesen beliefern müssen. Listing 2.3 zeigt exemplarisch eine Implementierung eines Commands.

```
1 public class DeliveryService {
2     public void sendObject(Command aCommand){
3         aCommand.execute();
4     }
5 }
```

Listing 2.2: DeliveryService

```
1 public class DeliverBigPackageCommand implements Command {
2     ...
3     public DeliverBigPackageCommand(Company aCompany,Package aPackage){
4         mCompany = aCompany;
5         mPackage = aPackage
6     }
7     @Override
8     public void execute() {
9         mCompany.checkPackage(mPackage);
10        mCompany.deliverPackage(mPackage);
11    }
12 }
```

Listing 2.3: DeliverBigPackageCommand

In Listing 2.4 wird das Zusammenspiel der Akteure dargestellt. Zunächst wird ein DeliveryService initialisiert, danach zwei konkrete Commands. Das Erste ist das `DeliverBigPackageCommand` das einerseits dem Empfänger und andererseits, für unser Beispiel speziell zugeschnitten, ein Objekt Package übergeben werden. Dem Zweite, das `DeliverLetterCommand` wird ebenfalls ein Empfänger und ein zusätzliches Objekt bereitgestellt. Beide Commands implementieren unterschiedliche Schritte zur Ausliefern eines Objektes, können jedoch beide auf diese Art von dem DeliveryService versendet werden. (Listing 2.4 Zeile 10-13).

```
1 public class main {
2
3     public final static void main(String[] args){
4         DeliverService deliverService = new DeliverService();
5         DeliverBigPackageCommand commandDeliverToDATEV =
6         new DeliverBigPackageCommand(new Company(),new Package());
7         DeliverLetterCommand commandDeliverToJohannes =
8         new DeliverLetterCommand(new PrivatePerson(), new Letter());
9
10        deliverService.sendObject(
11        commandDeliverToDATEV);
12        deliverService.sendObject(
13        commandDeliverToJohanes);
14    }
15 }
```

Listing 2.4: Main

2.3 Implementierungsmöglichkeiten

Die Literatur beschreibt zwei Möglichkeiten die durch das Command-Pattern realisiert werden können und eine, die auf unterschiedliche Art umgesetzt werden kann. Zunächst wird die Undo-Funtion und die Makro-Befehle vorgestellt. Dann wird über die Aufgaben eines Command-Objekts gesprochen.

Erweiterung durch eine Undo-Funktion Da jetzt jeder Befehl bzw. Aktion in einem Objekt gekapselt ist, kann man sehr einfach diese in einer Liste oder ähnlichem Lagern. Mit dieser Erkenntnis könnte man auf diese Art eine Undo-Funktion realisieren, die alle getätigten Befehle rückgängig macht. Für die Realisierung dieser Funktion muss man zunächst das Command-Inteface um eine weitere Funktion, eine Undo-Funktion, erweitern. Der Invoker müsste einen Mechanismus bereitstellen alle ausgeführten Requests zu speichern und das konkrete Command-Objekt muss sich folgende Informationen speichern:

- Alle Parameter die dem Command-Objekt zur Ausführung übergeben wurde.
- Das Receiver-Objekt
- Den Ausgangszustand des Receivers

Makro-Befehle Denkbar ist auch, mehrere Receiver an das Command-Objekt zu geben um so mehrere Aktionen durchführen zu können. Eine andere Möglichkeit wäre, in ein Command andere Commands auszuführen. Beides führt zu einer leichten Implementierung eines Commands das mehrere unabhängige Schritte ausführt.

Aufgaben des Command-Objekts Für die Implementierung des Command-Objekts ist noch zu klären: Wie intelligent soll ein Command-Objekt sein. Einerseits kann man alle Aufgaben an den Receiver deligieren. Andererseits könnte das Command-Objekt alle Aufgaben selbst übernehmen.

2.4 Fazit

2.4.1 Vorteile

- Entkopplung von Befehl und Ausführung.
- Hohe Flexibilität durch Austauschbarkeit

2.4.2 Nachteile

- Da jede neue Ausführung eines Commands in eine neue Klasse implementiert werden muss, führt das zu einer hohen Anzahl der Klassen.

3 Visitor

„Darstellung einer auf die Elemente einer Objektstruktur anzuwendenden Operation. Das Design Pattern Visitor ermöglicht die Definition einer neuen Operation, ohne die Klasse der von ihr bearbeiteten Elemente zu verändern."

3.1 Beschreibung

Das Visitor Pattern wird eingesetzt um die Operationen aus den Objektstruktur herauszunehmen, um sie erweiterbar zu machen. Die Voraussetzung dafür ist, dass jedes Element das Interface `Element` implementieren um eine Schnittstelle bereitzustellen, die einen Visitor aufnehmen kann. In diesem Fall heißt diese Schnittstelle `accept(Visitor v`. Das Interface `Visitor` stellt ihrerseits weitere Schnittstellen zum Besuchen der jeweiligen konkreten Elemente in der Objektstruktur bereit. In diesem Fall wären es zwei Methoden zum besuchen von den konrketen Elementen `ConcreteElementA` und `ConcreteElementB`. Erhält ein Element einen Visitor, kann er die, für sich bestimmte Methode des Visitors aufrufen. Welche Aktionen diese Methode dann durchführt, kann durch die verschiedenen konkreten Visitors bestimmt werden.

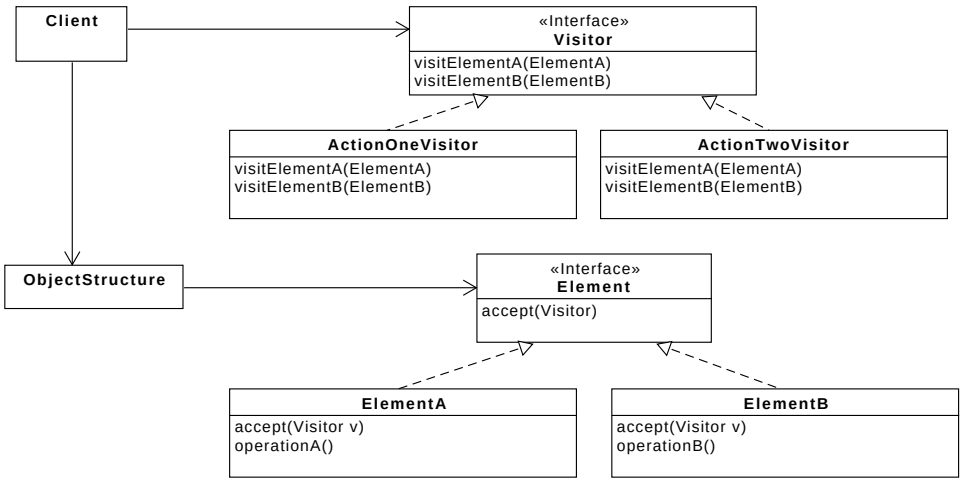

Abbildung 4: Eine UML-Darstellung von dem Visitor-Pattern.

3.2 Beispiel

Um das Visitor-Pattern besser zu demonstrieren, wird dieses anhand des folgenden Beispiels erklärt. In einer Küche gibt es mehrere Sorten Gemüse. Die Variation dieser verschiedenen Sorten ist überschaubar und wird sich nicht mehr ändern. Allerdings ist unklar, welche Operationen auf diese Objekte angewendet werden und in Zukunft noch angewendet werden könnten. Um dieses zu berücksichtigen erstellen wir eine Objektstruktur und verlagern die Operationen auf dieses nicht in den Objekten selbst, sondern außerhalb. Hierfür erstellen wir zunächst ein Interface `Element` mit der Schnittstelle `accept(Visitor aVisitor`, der Objekte mit dem Interface `Vistor` entgegennehmen kann. Methode.

```
1 public interface Element {
2     void accept(Visitor aVisitor);
3 }
```

Listing 3.1: Element Interface

Dementsprechend benötigen wir ein Interface `Visitor` das die Methoden zum Besuchen der einzelnen VArianten der Elemente bereitstellt.

```
1 public interface Visitor {
2     void visitBroccoli(Broccoli aBroccoli);
3     void visitPotato(Potato aPotato);
4     ...
5 }
```

Listing 3.2: Visitor Interface

Zunächst betrachten wir einen konkreten Visitor. Die erste Operation für alle Sorten bezieht sich auf das Waschen. Hierfür wird ein WaschenVisitor implementiert, der jeweils alle Methoden bereitstellt zum besuchen des jeweiligen Elements.

```
1  public class CleanVisitor implements Visitor {
2
3      @Override
4      public void visitBroccoli(Broccoli aBroccoli) {
5          System.out.println("Clean broccoli");
6          modifyBroccoli(aBroccoli);
7      }
8
9      @Override
10     public void visitPotato(Potato aPotato) {
11         System.out.println("Clean potatoes");
12         modifyPotato(aPotato);
13     }
14 }
```

Listing 3.3: CleanVisitor

Nachfolgend betrachten wir nur das Element `Apfel`. In der visit-Methode wird ein Apfel übergeben, der dann auf diesem Element entsprechenden Operationen ausführt. Diese Methode wird allerdings in der accept-Methode der Klasse `Apfel` aufgerufen. Die Methode `accept` ruft die Methode `visit` des aktuellen Visitors auf (in diesem Fall der WaschenVisitor) und übergibt sich selbst dieser

```
1 public class Potato implements Element {
2     @Override
3     public void accept(Visitor aVisitor) {
4         aVisitor.visitPotato(this);
5     }
6 }
```

Listing 3.4: CleanVisitor

3.3 Implementierungsmöglichkeit

Zur Bearbeitung der Objektstruktur muss jedem Elementen ein Visitor-Objekt über die Accept-Methode übergeben werden. Hierfür benötigt es einen Mechanismus auf diese Elemente zuzugreifen. Die Literatur stellt drei elegante Varianten zur Traversierung der Objektstruktur vor. Nachfolgend wird erklärt, wie diese aussehen.

Die Objektstruktur Die Objektstruktur übernimmt die Zuständigkeit, über seine Elemente zu traversieren selbst. Hierfür wird das Composite-Pattern eingesetzt. Dem Root-Element kann man ein Visitor-Objekt über die Accept-Methode übergeben, der diesen weiter verarbeitet und danach seinen Kindelementen übergibt.

Der Iterator Eine andere Möglichkeit ist ein interner oder externer Iterator. Dieser könnte bei dem Aufruf des nächsten Elements die Methode `accept` anstoßen.

Der Visitor Die letzte der drei Varianten ist, die Traversierung den Visitor-Objekten zu überlassen. Der Nachteil dieser Variante ist , mehrfach den Traversierungscodein den verschiedenen Vistors zu haben. Hierfür könnte man allerdings eine abstrakte Klasse einführen, der den redundanten Code eliminiert. Der entscheidende Vorteil dieser Variante ist, verschiedene Möglichkeiten der Traversierung durch die Objektstruktur anzubieten.

3.4 Fazit

3.4.1 Vorteile

- es können weitere Operationen hinzugefügt werden, ohne die Objektstruktur anzupassen.
- Funktionalität kann so gezielt auf bestimmte Arten von Objekten eingesetzt werden.
- Verwandte Operationen werden im Visitor zentral verwaltet
- Visitor können mit Objekten aus voneinander unabhängigen Klassenhierachien arbeiten. Mit dem Iterator-Pattern könnten zum Beispiel nur Funktionen aufgerufen werden, die in der Schnittstelle implementiert werden. Beim Iterator-Pattern können auf die verschiedenen Methoden der konkreten Elemente zugegriffen werden.

3.4.2 Nachteile

- Der Nachteil ist das durchbrechen der Kapselung. Den Vistors müssen unter Umständen bestimmte Operationen der Elemente zur Verfügung gestellt werden, die den internen Zustand des Objekts verändern.
- Das Hinzufügen von neuen Elementen ist mit der Änderung von allen Vistors verbunden.

Literatur

[GOF95] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, München 2004, ISBN 3-8273-2199-9.

[EIST06] Karl Eilebrecht, Gernot Starke: Patterns Kompakt: Entwurfsmuster für effektive Softwareentwicklung Spektrum Akademischer Verlag, 4 Auflage 2006, ISBN 3-8274-1443-1

[IN15] Michael Inden: Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung, dpunkt.verlag,Heidelberg 2015, ISBN 978-86490-203-1