Fakultät
Informatik

Design Patterns und Anti-Patterns

über das Thema

Verhaltensmuster: Visitor-, Command- und Observer-Pattern

Autor: Johannes Pfann
johannes.pfann@fau.de

Abgabedatum: 05.06.2016

Inhaltsverzeichnis

1 Einleitung

1.1 Verhaltensmuster

Das Observer-,Command- und Visitor-Pattern sind sog. Verhaltensmuster. Mit Verhaltensmuster bezeichnet man eine Kategorie der Desing-Pattern, die aus dem Buch der Gang of Four (vgl. [GOF95]) dargestellt werden. Das Ziel der Desing-Pattern in dieser Kategorie ist das definieren von Strukturen, um komplexes Verhalten von Software besser zu modellieren. Um dieses Ziel zu erreichen, wird das Verhalten aus den Objekten und Klassen gekapselt und gezielt die Interaktion zwischen diesen beschrieben.

2 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. “(vgl. [GOF95])

2.1 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 das jeweilige 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.

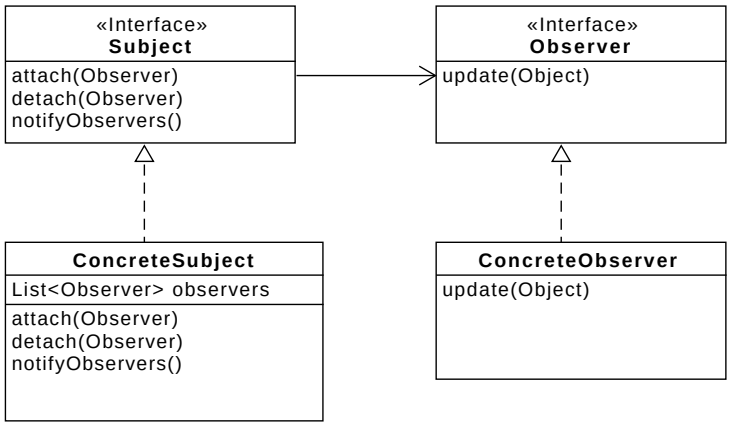

Abbildung 1: UML-Darstellung eines Observer-Pattern.

2.2 Beispiel

Das Beispielszenario behandelt eine Arbeitsvermittlung deren Softwaresystem mit dem Observer-Pattern umgesetzt wird. In dem Softwaresystem gibt es eine zentrale Komponente, das JobCenter, welches 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 nimmt das Jobcenter die Rolle Subjects ein und die Homepage und die Monitore die Rolle Observer. Nachfolgend in Listing 2.1 und 2.2 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 2.1: Observer Interface

Im zweiten Schritt wird dann die Klasse Jobcenter (siehe Listing 2.3) implementiert, die von dem Interface Subject erbt und somit die Methoden `attach(Observer aObserver`,

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

Listing 2.2: Subject Interface

`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 die Tatsache das das Jobangebot an dieser Stelle nur das Interface Observer kennt und das durch die Iteration ausnahmslos alle Observer benachrichtigt werden.

```
1 public class JobCenter implements Subject {
2
3     List<Observer> observers = new LinkedList<Observer>();
4
5     public void attach(Observer aObserver) {
6         mObserver.add(aObserver);
7     }
8     public void detach(Observer aObserver) {
9         mObserver.remove(aObserver);
10    }
11    public void notifyObservers(Offer aOffer) {
12        for(Observer observer : observers){
13            observer.update(aOffer);
14        }
15    }
16 }
```

Listing 2.3: Jobcenter

Als letztes betrachten wir die Klasse `Homepage` (siehe Listing 2.4). Diese übernimmt die Rolle des Observers indem sie von dem Interface `Observer` erbt und die Methode `update(Offer aOffer)` implementiert.

```
1 public class Homepage implements Observer {
2
3     public void update(Offer aOffer) {
4          updateHomepage(aOffer);
5     }
6 }
```

Listing 2.4: 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, damit diese sich selbständig aktualisieren können.

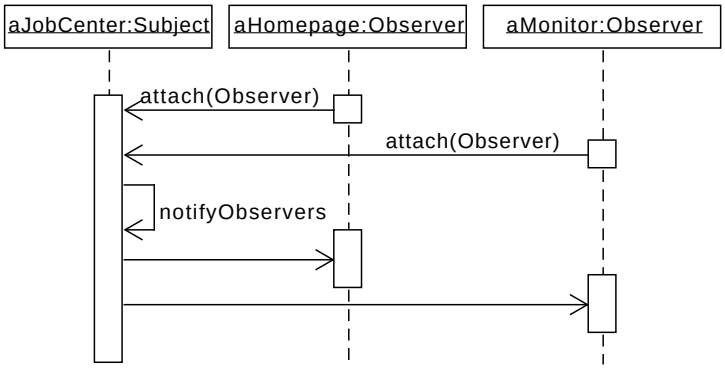

Abbildung 2: Sequenzdiagramm mit dem Ablauf von JobCenter und desssen Clients

2.3 Implementierungsmöglichkeiten

Push- oder Pull-Model Bei den beiden Modellen werden die unterschiedlichen Arten der Übertragung von Daten betrachtet. Bei dem Push-Model ist die Idee, alle Daten über die Update-Methode der Observer zu übermitteln. In unserem Fall muss das UML-Diagramm in Abbildung 1 angepasst werden. Die Methode `update()` benötigt zusätzliche Parameter. Das `ConcreteSubject` übergibt dann die Parameter beim Aufruf der Update-Methode. Daraus ergeben sich zwei Nachteile. Der Erste, das Subject muss die Observer besser kennen. Die Update-Methode könnte so auf eine bestimmte Aufgabe zugeschnitten werden und somit für andere Anwendungsfälle ungeeignet sein. Der zweite Nachteile ist, das nicht alle Observer diese Informationen benötigen und somit unnötig viele Parameter übermittelt bekommen. Die Pull Methode verfolgt einen anderen Ansatz. Hier werden keine Daten als Parameter übergeben sondern die Observer greifen direkt nach einer Benachrichtigung auf die Daten der Instanz des `ConcreteSubject` zu. Der Nachteil dieser Variante ist, das die einzelnen Beobachter das Subjekt kennen müssen und gleichzeitig das Subject öffentliche Methoden für diesen Zugriff bereitstellen muss.

Beobachter beobachtet mehr als ein Subjekt Wenn ein Observer mehrere Subjects beobachtet, kann es passieren, das er im Falle einer Benachrichtigung nicht feststellen kann, von wem die Update-Methode aufgerufen wurde. Abhilfe schafft die Erweiterung der Update-Methode mit einem zusätzlichen Parameter `Subject`. Das Subject kann

sich bei der Benachrichtigung selbst übermitteln, damit der Observer weiß von wem er benachrichtig wurde.

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. Fügt man die Notify-Methode nach der Änderung eines Werte in dem Subject ein, wird ein Automatismus geschaffen, sodass die Benachrichtigung nicht vergessen werden kann. Der Nachteil dieser Implementierung ist, das dadurch die Anzahl der Updates höher ausfallen könnte. Die andere Variante überlässt den Client die Verantwortung, nachdem er seine Änderungen übermittelt hat, die Notify-Methode aufzurufen. Hier können zwar die Benachrichtigungen gezielter ausgeführt werden, allerdings könnte die Benachrichtigung auch vergessen werden.

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 `attach(Observer)` 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 Subject und den Observer zu komplex wird, empfielt sich die Verwaltungsaufgaben zwischen Subject und dessen Observer auszulagern. In der Literatur wird von einem ChangeManager gesprochen. Dieser regelt folgende Gebiete: Das Mapping zwischen den Subjects und den Observer. Definiert spezielle Aktualisierungsstrategien und übernimmt die Benachrichtigung eines konkreten Subjects für dessen Observer.

2.4 Fazit

2.4.1 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.
- Es muss im Voraus nicht bekannt sein, wie viele abhängige Objekte sich zur Laufzeit registrieren, und welche das sind.

2.4.2 Nachteile

- In manchen Fällen kann das Observer-Pattern allerdings zu Problemen führen. Beispielswiese könnte es zu Situationen kommen, dass mehrere Observer gleichzeitig Subjects von weiteren Observer sind usw. Die Komplexität steigt schenll bei solchen Sytemen und gerade bei der Fehlersuche kann dadurch erschwert werden.
- Außerdem besteht die Gefahr von zyklischen Abhängigkeiten. Ein Subject kann durch komplexe Verkettungen gleichzeitig auch ein Observer dieser Abhängigkeitskette sein, sodass eine Benachrichtigung gleichzeitig zu einer eigenen Aktualisierung führt, die wiederum eine Benachrichtigung auslöst.

3 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.“(vgl. [GOF95])

3.1 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 Szenario ohne dem Command-Pattern, 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 verwenden. Der Invoker muss also nur ein passendes Command-Objekt erhalten und dessen Methode `execute()` aufrufen. Die konkrete Durchführung dieser Methode wird dann in dem konkreten Command-Ojbekt behandelt. Abbildung 3 zeigt ein UML-Diagramm für das Command-Pattern.

3.2 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 der Absicht etwas zu versenden. Möchten wir dieses Szenario implementieren, hätten wir drei Teile: Den Lieferdienst,

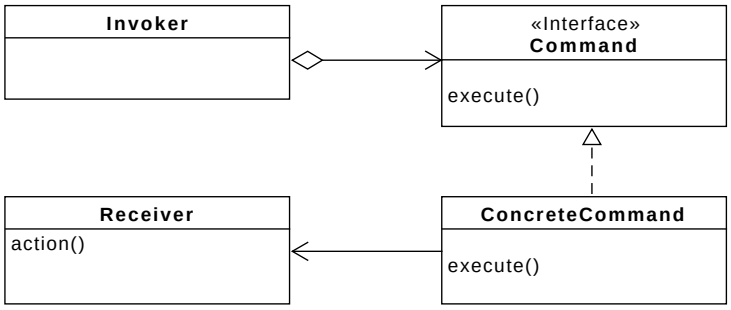

Abbildung 3: Command-Pattern als UML-Diagramm.

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 3.1). 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 3.1: Command Interface

In Listing 3.2 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.

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

Listing 3.2: DeliveryService

Danach werden verschiedene konkrete Commands implementiert, die einerseits den Empfänger kennen und andererseits wissen, wie sie diesen beliefern müssen. Listing 3.3 zeigt exemplarisch eine Implementierung eines Commands.

In Listing 3.4 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 wird. Dem Zweiten, das `DeliverLetterCommand` wird

```
1  public class DeliverBigPackageCommand implements Command {
2      ...
3      public DeliverBigPackageCommand(Company aCompany,
4                      Package aPackage){
5          mCompany = aCompany;
6          mPackage = aPackage
7      }
8      @Override
9      public void execute() {
10         mCompany.checkPackage(mPackage);
11         mCompany.deliverPackage(mPackage);
12     }
13 }
```

Listing 3.3: DeliverBigPackageCommand

ebenfalls ein Empfänger und ein zusätzliches Objekt übergeben. Beide Commands implementieren unterschiedliche Schritte zur Ausliefern eines Objektes, können jedoch beide auf diese Art von dem DeliveryService versendet werden. (Listing 3.4 Zeile 10-13).

```
1  public class main {
2      public final static void main(String[] args){
3          DeliverService deliverService = new DeliverService();
4          DeliverBigPackageCommand commandDeliverToDATEV =
5          new DeliverBigPackageCommand(new Company(),
6                                       new Package());
7          DeliverLetterCommand commandDeliverToJohannes =
8          new DeliverLetterCommand(new PrivatePerson(),
9                                   new Letter());
10
11         deliverService.sendObject(
12         commandDeliverToDATEV);
13         deliverService.sendObject(
14         commandDeliverToJohanes);
15     }
16 }
```

Listing 3.4: Main

3.3 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.

3.4 Fazit

3.4.1 Vorteile

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

3.4.2 Nachteile

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

4 Visitor

„Das Visitor Pattern ermöglicht es, neue Operationen auf den Elementen einer Struktur zu definieren, ohne die Elemente selbst anzupassen.“(vgl. [EIST06])

4.1 Beschreibung

Das Ziel des Visitor-Pattern ist, die Operationen auf Elemente in einer Objektstruktur außerhalb dieser Objektstruktur zu lagern um diese beliebig auszuwechseln. Die Voraussetzung für jedes Element dieser Objektstruktur ist ein Interface `Element`, welches eine Methode `accept(Visitor)` erfordert. Dabei repräsentiert das übergebene Objekt `Visitor` die Operationen, die auf dieses Element angewendet werden sollen. Der Visitor selbst ist auch ein Interface. Das Interface muss für jeden Typ Element eine eigene Methode implementieren, die für die Bearbeitung der jeweiligen Elemente verantwortlich sind. Das bedeutet für unser UML-Diagramm, mit den zwei verschiedenen Elementen `ElementA` und `ElementB`, das wird zwei Methoden `visitElementA(ElementA)` und `visitElementB(ElementB)` definieren müssen. Der Ablauf des Visitor-Pattern ist folgendermaßen: Zunächst wird einem Element ein konkretes Visitor-Objekt übergeben. Dieses Element ruft dann die für sich implementierte Visit-Methode auf und übergibt sich selbst dem Visitor. Der Visitor kann dann auf das Element zugreifen und es bearbeiten.

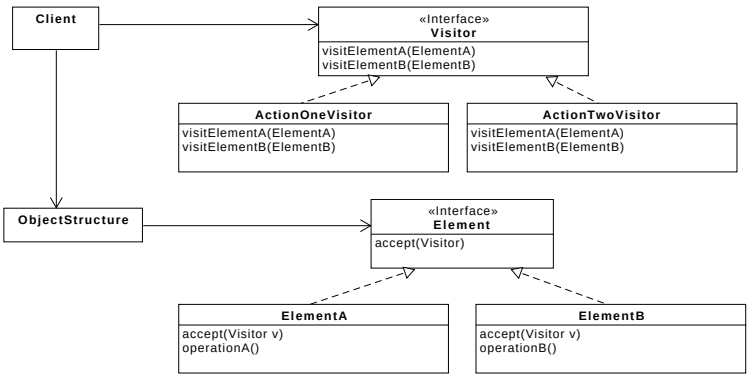

Abbildung 4: Eine UML-Darstellung von dem Visitor-Pattern.

4.2 Beispiel

Das folgende Beispiel soll das Visitor-Pattern und die Möglichkeiten, welche sich aus dem Pattern ergeben, besser erklären. 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. 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` und lassen jedes

Element unserer Objektstuktur dieses Interface implementieren. Damit verschaffen wir jedem Element die Möglichkeit ein Visitor-Objekt zu erhalten und auf dessen Methoden zuzugreifen.

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

Listing 4.1: Element Interface

Als nächstes erstellen wir das Visitor-Objekt (Listing 4.2). Dieses muss für jeden Elementtyp der Objektstruktur eine eigene Methode bereitstellen. Für das Beispiel wurde exemplarisch ein Potato-Element und ein Broccoli-Element erstellt. Für diese beiden Elemente muss das Visitor-Objekt die Methoden definieren, nämlich eine `visitPotato(Potato)` und `visitBroccoli(Broccoli` Methode.

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

Listing 4.2: Visitor Interface

In Listing 4.3 wird ein konkretes Element vorgestellt. Dieses muss durch das Interface Element die Accept-Methode implementieren und bekommt einen beliebigen Visitor. In dieser Accept-Methode ruft das Element Potato dann die für ihn gedachte Visitor-Methode, nämlich die `visitPotato(Potato)` auf und übergibt sich selbst als Potato-Element.

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

Listing 4.3: CleanVisitor

Daraufhin kann ein konkreter Visitor, hier der CleanVisitor (siehe Listing 4.4) das übergebene Potato modifizieren. Analog werden jetzt die Visit-Methoden für die anderen Elemente implementiert. Voraussetzung ist allerdings an dieser Stelle, das die jeweiligen Elemente öffentliche Attribute oder Methode bereitstellen, damit diese auch entsprechend bearbeitet werden können.

Abschließend veranschaulicht das Sequenzdiagramm in Abbildung 5 noch einmal den gesamten Ablauf. Ein beliebiger Client erstellt zunächst die Elemente der Objektstruktur und den Visitor. Dann ruft er bei beiden Elementen mit die Accept-Methode auf und

```java
public class CleanVisitor implements Visitor {

    @Override
    public void visitBroccoli(Broccoli aBroccoli) {
        System.out.println("Clean broccoli");
        modifyBroccoli(aBroccoli);
    }

    @Override
    public void visitPotato(Potato aPotato) {
        System.out.println("Clean potatoes");
        modifyPotato(aPotato);
    }
}
```

Listing 4.4: CleanVisitor

übergibt den Elementen den erstellten konkreten Visitor. Diese rufen ihrerseits die Accept-Methode auf und übergeben sich selbst. Der Visitor ist nun in der Lage die übergebenen Elemente in den dafür vorgesehenen Visit-Methoden zu bearbeiten.

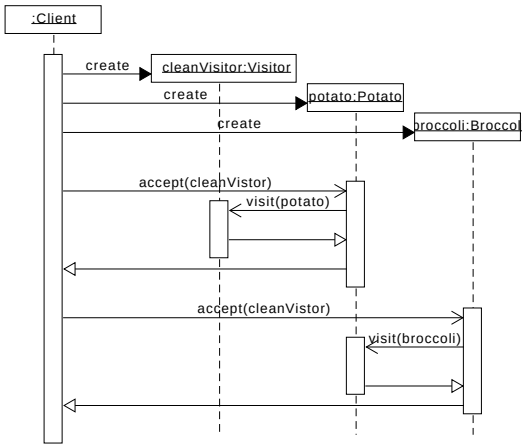

Abbildung 5: Der Ablauf des Visitor-Pattern als Sequenzdiagramm

4.3 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 (siehe [GOF95] Abschnitt Visitor)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 Traversierungscode in den verschiedenen Vistors zu implementieren. 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.

4.4 Fazit

4.4.1 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

4.4.2 Nachteile

- Problematisch ist, dass Elemente öffentliche Attribute und Methoden bereitstellen müssen zur Manipulation durch den Visitors.
- Das Hinzufügen von neuen Elementen ist mit der Änderung von allen Vistors verbunden.

5 Quellen

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