```java
public class Client implements Observer {

    public void update(Offer aOffer) {
        if(aOffer instanceof DeveloperJob){
            doSomething(aOffer);
        }
    }
}
```

Listing 0.0.1: Logger Implementierung - Threadsafe mit Double Checking

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

Einleitung

1.0.1 Verhaltensmuster

asdfasdf

2 Observer

2.1 Definition

„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. “

2.2 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-Abhängkeit, 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 4.1 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.

2.3 Beispiel

Um das Observer-Pattern besser zu illustrieren wird dieses mithilfe eines Beispiels implementiert. Das Beispielszenario behandelt eine fiktive Arbeitsvermittlung bei der sich Personen melden und benachrichtigt werden sobald ein Jobangebot die Arbeitsvermittlung erreicht. Wir wählen die Variante Subjekt als Interface und die Benachrichtigung druch das Push-Model implementiert wird. Hierfür erstellen wir zunächst das Interface Beobachter und Subject . Ein konkretes Subjekt wird gezwungen die Beobachter zu registrieren, zu entfernen und sie zu benachrichtigen (siehe Abbildung 1 und 2).

```
1 public interface Beobachter {
2     void update(Object aObject);
```

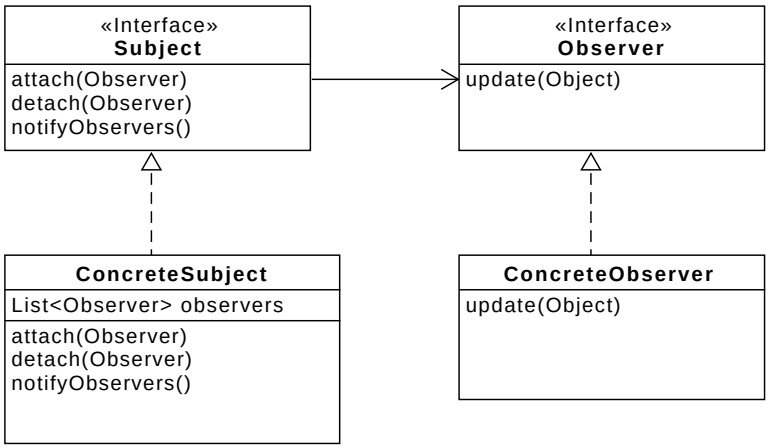

Abbildung 2.1: UML-Darstellung eines Observer-Pattern.

```
3 }
```

Listing 1: Beobachter

```
1 public interface Subjekt {
2
3     void registrieren(Beobachter aBeobachter);
4     void entferne(Beobachter aBeobachter);
5     void benachrichtige(Object aObject);
6
7 }
```

Listing 2: Subjekt

Im zweiten Schritt wird dann die Klasse Arbeitsvermittlung erstellt die das Interface Subjekt erben muss. Wird die Methode benachrichtigen aufgerufen, wird durch alle registrierten Beobachter iteriert und dessen Methode `update` aufgerufen. Die Informationen des jeweiligen Jobs werden hier durch Object dargestellt und beim Aufruf der Methode `update` übergeben. (siehe Abbildung 3)

```
1 public class Arbeitsvermittlung implements Subjekt {
2     private List<Beobachter> mBeobachter;
3
4     public void registrieren(Beobachter aBeobachter) {
5         mBeobachter.add(aBeobachter);
6     }
7     public void entferne(Beobachter aBeobachter) {
8         mBeobachter.remove(aBeobachter);
9     }
10    public void benachrichtige(Object aObject) {
11        for(Beobachter beobachter : mBeobachter){
12            beobachter.update(aObject);
```

```
13          }
14      }
15 }
```

Listing 3: Arbeitsvermittlung

Als letztes betrachten wir die Klasse `PersonA`. Diese übernimmt die Rolle des Beobachters indem sie von dem Interface `Beobachter` erbt und die Methode `update` implementiert.

```
1 public class PersonA implements Beobachter{
2     public void update(Object aObject) {
3         doSomething(aObject);
4     }
5 }
```

Listing 4: PersonA

2.4 Implementierung

Für das Observer-Pattern gibt es mehrere Implementierungsmöglichkeiten. In Abschnitt 1.1 ...

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 4.1 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.

Lagerung der Beobachter Hintegrund ist folgendes Szenario: Es gibt viele Subjekts und einige Beobachter. Die Subjekts müssen jetzt alle die Instanzen der jeweiligen Beobachter speichern. Um sich diesen Speicher zu ersparen, könnte man die Auflistung aller Beobachter in einer externen Datenstruktur bzw. Objekt lagern.

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.

Aufräumen der Instansen Es sollte Beachtet werden, das wenn ein Beobachter sich vom konkreten Objekt entfernt, auch die Instanz des konkreten SUbjekt aufräumt.

Vor dem Benachrichtigen das Subjekt zuerst updaten Bei der Implementierung des konkreten Subjekts sollte beachtet werden, vor der Benachrichtigung der Beobachter, zuerst das Subjekt zu aktualisieren. Gerade bei vererbten Methoden besteht Gefahr eines solchen Szenario.

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.

Subjekt als Interface oder Abstrakte Klasse Das Subjekt kann ein Interface oder als abstrakte klasse implementiert sein. Der Nachteil bei einer abstrakten Klasse ist, dass nicht auf verschiedene konkrete Subjekts eingegangen werden kann. Jedoch erhöht sich der PRogrammieraufwand durch die wiederholte Implementierung des Selben Codes.

2.5 Vor- und Nachteile

2.5.1 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.

2.5.2 Nachteile

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

3 Command

3.1 Definition

„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.“

3.2 Beschreibung

Das Ziel des Command-Pattern ist, die Benutzung des Receivers von dem Benutzer, dem Invoker, zu trennen. 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 also dann in dem konkreten Command-Ojbekt durchgeführt.

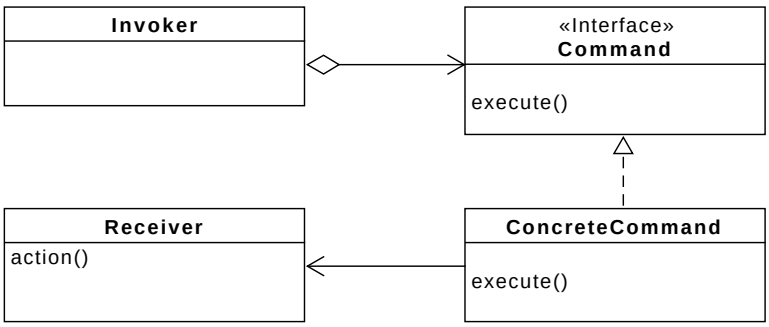

Abbildung 3.1: Command-Pattern als UML-Diagramm.

3.3 Beispiel

Als Beispieszenario wird ein Logistikunternehmen das verschiedene Pakete versendet gewählt. Bei der Übermittlung von unterschiedlichen Paketen zu unterschiedlichen Empfängern müssen verschiedene Schritte getätigt werden. Außerdem können sich durch neue Richtlinien

Arbeitsschritte ändern. Durch eine zusätzliche Schicht trennen wir das Vorhaben, ein Paket zu versenden von den tatächlichen Schritten, die es benötigt um ein Paket zu versenden. Als erstes konstruieren wir eine Schnittstelle `Command`. Dieser geben wir eine Methode `execute` mit dem Parameter Paket.

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

Listing 5: Command

Danach erstellen wir das Logistikunternehmen als Objekt um von dort alle Arten von Paketen zu versenden. Welches Command-Objekt zum Ausführen der jeweiligen Schritte verwenden wird von außerhalb bestimmt.

```
1 public class Logistikunternehmen {
2     Command mCommand = null;
3     public Logistikunternehmen(Command aCommand){
4         mCommand = aCommand;
5     }
6     public void versendePacket(Paket aPaket){
7         mCommand.execute(aPaket);
8     }
9 }
```

Listing 6: InlandskundeSecureCommand

Die erste Veriante um ein Paket zu versenden, ist das `InlandsKundenDefaultCommand`. Dieses kennt seinen Empfänger, nämlich den Inlandskunden und ruft auf diesem nur die Methode `empfangePaket` auf.

```
1 public class InlandsKundeDefaultCommand implements Command{
2     Inlandskunde mInlandskunde;
3     public void execute(Paket aPaket) {
4         mInlandskunde.empfangePaket(aPaket);
5     }
6 }
```

Listing 7: InlandsKundeDefaultCommand

Die zweite Variante ist die sichere Übermittlung eines Paketes mit einer zusätzlichen Verpackung. Wieder kennt das Command-Objekt den Empfänger und ruft dessen Methode auf. Allerdings wird Logik in dem Command-Objekt behandelt indem das Paket zusätzlich verpackt wird.

```
1 public class InlandskundeSecureCommand {
2     Inlandskunde mInlandskunde;
3     public void execute(Paket aPaket) {
4         mInlandskunde.empfangePaket(verpackePacket(aPaket));
5     }
```

```
6 }
```

Listing 8: InlandskundeSecureCommand

3.4 Implementierung

Für das Command-Pattern ist aus der Sicht des Autors folgende drei Erweiterungsmöglichkeiten des Command-Patterns entscheident. Alle drei stammen aus [GoF].

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 zurücknimmt um zum Ausgangszustand zurückzukommen. Man könnte hierfür auch das Command-Objekt derart erweitern, dass zusätzliche Informationen

Aufgaben des Command-Objekts Die Frage die man sich stellen sollte ist: Wie intelligent soll ein Command-Objekt sein. Einerseits kann man alle aufgaben an den Receiver deligieren. Das andere Extrem muss das Command-Objekt nichts von dem Receiver kennen und implementiert die komplette Logik.

Makro-Befehle Denkbar ist auch, mehrere Receiver an das Command-Objekt zu geben um so mehrere Aktionen durchführen zu können.

3.5 Vor- und Nachteile

3.5.1 Vorteile

- Austauschen von Befehlen eines Aufrufers ist sehr einfach und ohne Codeänderung am Aufrufers möglich.
- Durch die Entkopplung von Befehl und Aufrufers kann der Befehl bei anderen Aufrufern auch angewendet werden (Wiederverwendbarkeit).
- Einfache Implementierung der Rückgängig- oder Loggingfunktionen.

3.5.2 Nachteile

- Da jeder neue Befehl in eine neue Klasse abbildet, führt das zu einer hohen Anzahl der Klassen.

4 Visitor

4.1 Definition

„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.“

4.2 Beispiel

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.

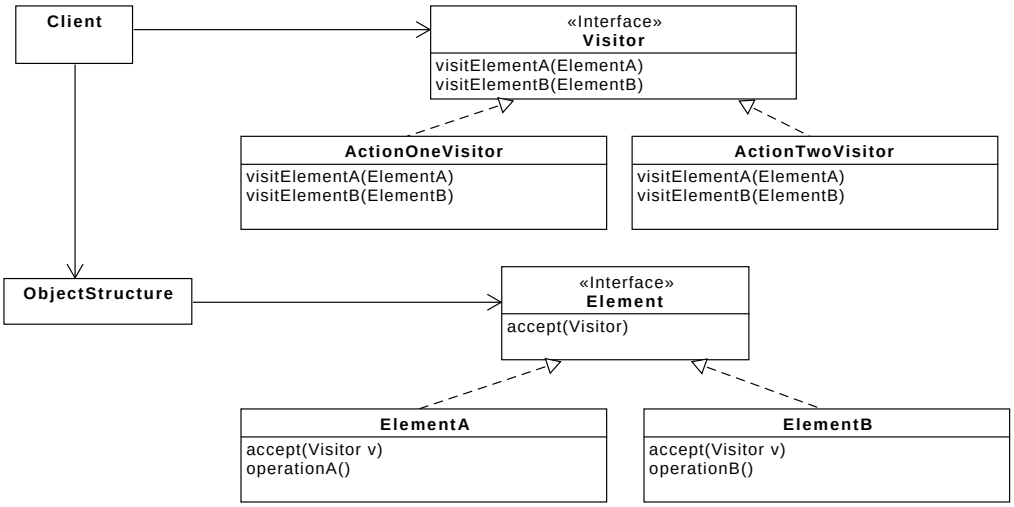

Abbildung 4.1: Eine UML-Darstellung von dem Visitor-Pattern.

4.3 Implementierung

Wer ist für die Traversierung der Objektstruktur verantwortlich? Ein Visitor-Objekt muss jedes einzelne Element der Objektstruktur aufsuchen. Hierfür gibt es drei Implementierungsmöglichkeiten.

Die Objektstruktur Die Objektstruktur übernimmt die Zuständigkeit über all seine Elemente zu traversieren. Dieses geschieht oft mithilfe eines Composite Patterns, bei dem ein Element seine Kindelemente mit deren Mehtode `accept` aufruft und einen Vistor 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 entscheidente Vorteil dieser VAriante ist, verschiedene Möglichkeiten der Traversierung durch die Objektstruktur anzubieten.

4.4 Beispiel

Um das Visitor-Pattern besser zu demonstrieren, wird dieses anhand des folgenden Beispiels erklärt. In einer Küche gibt es mehrere Sorten Obst und Gemüse. Die Variation dieser verschiedenen Sorten ist überschaubar und wird sich nicht mehr ändern. Allerdings ist unklar, welche Operationen auf diese Objekte angewendet werden kann und in Zukunft noch angewendet werden könnte. 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 9: Element

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

```
1 public interface Visitor {
2     void visit(Apfel aApfel);
3     void visit(Kiwi aKiwi);
4     void visit(Paprika aPaprika);
5 }
```

Listing 10: Visitor

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 WaschenVisitor implements Visitor {
2     public void visit(Apfel aApfel) {
3         wasche(aApfel);
4     }
5     public void visit(Kiwi aKiwi) {
6         wasche(aKiwi);
7     }
8     public void visit(Paprika aPaprika) {
9         wasche(aPaprika);
10    }
11    ...
12 }
```

Listing 11: WaschenVisitor

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 Apfel implements Element {
2     ...
3     public void accept(Visitor aVisitor) {
4         aVsisitor.visit(this);
5     }
6 }
```

Listing 12: Apfel

4.5 Vor- und Nachteile

4.5.1 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.

4.5.2 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.