Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

Design Pattern

vorgelegt zum Abschluss des Seminars in der Vertiefungsrichtung Technische Informatik im Studiengang Informatik

Eduard Elsesser: Delegation
Reiner Fäth: Factory
Stefan Liebler & Lukas Hahmann: Decorator
Jonas Heisterkamp: Chain of Responsibility
Clemens Henker: Dependency Injection
Cihan Kinali: Singelton
Johannes Link: Proxy
Roman Müller: Visitor
Volker Schneider: Observer
Vitaliy Schreibmann: Adapter & Facade
Martin Spenkuch & Philipp Rimmele: State
Benno Willoweit: MVC
Mehmet Zahid Aydin: Strategy

Datum:

Betreuer: Prof. Grötsch

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Singleton

Singleton ist ein Entwurfsmuster (engl. *Design Pattern*), welches in der Softwaretechnik im Bereich der Entwicklung eingesetzt wird. Dieses *Design Pattern*, welches auch Einzelstück genannt wird, stellt sicher, dass es von einer Klasse nur eine Instanz geben darf und ermöglicht außenstehenden Klassen einen globalen Zugriff.

Im Folgenden werden Ihnen die Begrifflichkeit, der Anlass, die Absicht und der Einsatzbereich dieses Musters beschrieben.

1.1 Definition

Gang Of Four-Definition
Singleton:
Sichere ab, dass eine Klasse genau ein Exemplar besitzt und stelle einen globalen Zugriffspunkt darauf bereit. [Gam94]

1.2 Motivation

In der objektorientierten Programmierung kann es je nach Anwendung in einem Projekt sinnvoll sein, genau ein Exemplar einer Klasse zu erzeugen. Obwohl das Schreiben in eine Protokolldatei mehreren Clients gestattet ist, sollte es in dem Programm nur eine Logdatei geben, die mit Aktionen von Prozessen versehen werden sollte.

Wie können wir also sicherstellen, dass eine Klasse über genau ein Exemplar verfügt und dass einfach auf dieses Objekt zugegriffen werden kann? Eine globale Variable ermöglicht den Zugriff auf ein Exemplar, verhindert aber nicht das Erzeugen mehrerer Exemplare.

Demnach ist es besser, die Klasse selbst für die Verwaltung ihres einzigen Exemplars zuständig zu machen. Diese Klasse kann durch das Abfangen von Befehlen zur Erzeugung neuer Objekte gewährleisten, dass kein weiteres Exemplar erzeugt wird, und sie kann die Zugriffsmöglichkeit auf dieses anbieten. Das ist die Essenz des Singletonmusters.

1.3 Absicht

Mit Hilfe dieses Entwurfsmusters wird dem Softwareentwickler sichergestellt, dass eine Klasse genau ein Exemplar besitzt. Demzufolge wird ihm ein globaler Zugriffspunkt bereitgestellt.

1.4 Einsatzbereich

Die Anwendung des Entwurfsmusters Singleton eignet sich, wenn
1. sichergestellt werden soll, dass das betreffende Exemplar nur *einmal* instanziiert wird
2. ein *globaler Zugriffspunkt* auf das Exemplar existieren soll
3. das Exemplar durch die Bildung von Subklassen erweiterbar sein soll und andere Klassen in der Lage sein sollen, das erweiterbare Exemplar ohne Modifikationen ihres Quellcodes nutzen zu können.

1.5 Struktur

Alle anderen Objekte, die dieses Singleton Exemplar benötigen (Clients), nutzen ein und dieselbe Referenz auf das Exemplar der Singleton Klasse (Abb. 1-1).

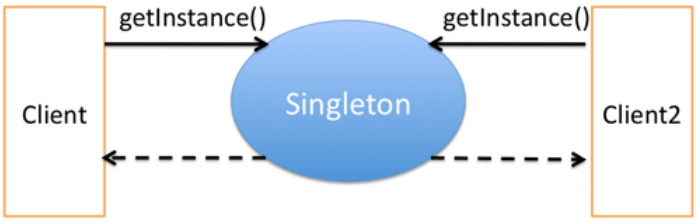


Abb. 1.1: Zugriff von Clients auf das Exemplar der Singleton-Klasse

Der allgemeine Aufbau des Entwurfsmusters Einzelstück besteht aus einer statischen Referenz auf seine eigene Klasse, einem Konstruktor und einer statischen Operation, die eine Referenz auf die Klasse Singleton zurückliefert.
Im Folgenden wird die allgemeine Struktur eines Singleton Entwurfsmusters in Form eines UML Klassendiagramms visualisiert (Abb. 1-2).

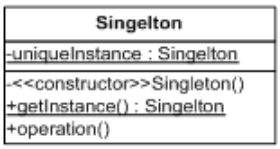


Abb. 1.2: Singleton-Klassendiagramm

Die Klasse Singleton definiert eine Operation, die es anderen Klassen ermöglicht, auf sein einziges Exemplar zuzugreifen. Die Aufgabe der statischen Operation *getInstance()* ist es, ein einziges Exemplar zurückzugeben. Somit ist die Klasse Singleton für die Erzeugung seines einzigen Exemplars zuständig. Ein Zugriff auf das Objekt dieser Klasse von Klienten erfolgt ausschließlich über die statische Operation des Einzelstücks.

1.6 Implementierung

Im Folgenden werden die zwei unterschiedlichen Implementierungsvariationen des Singleton *Design Patterns* vorgestellt.

1.6.1 Lazy Loading

Das Singleton Entwurfsmuster wandelt das einzige Exemplar zu einem normalen Exemplar seiner Klasse. Die Klasse wurde so codiert, dass nur ein einziges Objekt jemals erzeugt werden kann. Üblicherweise versteckt man die Anweisung, die das Exemplar erzeugt, hinter einer statischen Operation, die für die einmalige Initialisierung und somit für das einzige Vorhandensein eines Exemplars dieser Klasse zuständig ist. Diese Operation kann auf die Variable, die das einzige Exemplar enthält zugreifen, Außerdem stellt sie sicher, dass die Variable mit dem einzigen Exemplar initialisiert ist, bevor sie ihren Wert zurückgibt. Diese Ausübung des Grundgedankens stellt sicher, dass ein Einzelstück vor seiner ersten Verwendung erzeugt und initialisiert wird.

Andere Klassen greifen auf das Einzelstück ausschließlich durch die *getInstance* Funktion zu. Die Variable *uniqueInstance* wird mit null initialisiert und die statische Funktion *getInstance* gibt seinen Wert zurück, wobei ein Objekt dieser Klasse erzeugt wird, wenn er null enthält. *GetInstance* verwendet verzögerte die Initalisierung (engl. lazy initialization). Das *Lazy Loading* löst das Problem der pauschalen Erstellung, die zu Beginn einer Klasse erfolgt, durch eine verzögerter Instanziierung. Somit wird das Singleton erst dann erstellt, wenn es das erste Mal gebraucht wird, also beim ersten Aufruf von *getInstance()*.

Lazy Loading: Instanziierung beim ersten Bedarf:

```
1  // Klasse Singtон Variante 1
2  public class Singleton
3  {
4          // Static Fields
5
6          private static Singleton uniqueinstance = null;
7
8          // Static Methods
9
10         // Methode fuer die Singleton Referenz
11         public static Singleton getInstance()
12         {
13                 if ( Singleton.uniqueinstance == null )
14                 {
15                         Singleton.uniqueinstance = new Singleton ();
16                 }
17                 return Singleton.uniqueinstance;
18         }
19
20         // Constructors
21
22         // Initialisiert die Klasse Singleton
23         private Singleton()
24         {
25                 super ();
26         }
27
28         // Methods
29
30         // Operationsmethode
31         public void opearation()
32         {
33                 // Singleton - Operation
34         }
35 }
```

Abb. 1.3: Erste Variante der Klasse Singleton

Beachten Sie an dieser Stelle, dass der Konstruktor sowohl in dieser als auch in der nachfolgenden Variante *Eager Loading* privat ist, das heißt von außerhalb der Klasse unzugänglich ist. Versucht ein Klient, ein Singleton direkt zu erzeugen, so ergibt sich zur Übersetzungszeit ein Fehler. Dies stellt sicher, dass nur ein Exemplar erzeugt werden darf.

Synchronisierung

Jedoch schafft das *Lazy Loading* ein neues Problem im Bereich des Multithreadings. So kann es zur Erstellung zweier Singletons kommen, wenn ein Thread nach der null-Prüfung - direkt vor der Instanziierung - den Fokus abgibt und ein anderer Thread *getInstance()* durchläuft. Demnach erstellt der andere Thread Exemplar der Klasse Singleton. Der erste Thread erhält nun den Fokus wieder und weiß nicht, dass das Singleton bereits erzeugt wurde und erstellt es noch einmal. Somit wird die Einmaligkeit des Singletons verletzt. Daher bedarf es einer Synchronisation. Synchronisationen sind allerdings teuer und erzeugen einen Overhead bei jedem Aufruf von *getInstance()*. Bei einer performancekritischen Anwendung mit vielen Aufrufen von *getInstance()* sollte von dieser Variante Abstand genommen werden.

```
1        public static synchronized Singleton getInstance()
2        {
3                if ( Singleton.uniqueinstance == null )
4                {
5                        Singleton.uniqueinstance = new Singleton ();
6                }
7                return Singleton.uniqueinstance;
8        }
```

Abb. 1.4: Synchronisation der getInstance – Operation

1.6.2 Eager Loading

Eine recht einfache Implementierungsvariante ist das *Eager Loading*. Bei der vorgezogenen Instanziierung des Singletons findet die Objekterstellung bereits beim Laden der Klasse statt (Abb.1.5).

```
1  // Klasse Singtон Variante 2
2  public class Singleton
3  {
4          // Static Fields
5
6          private static final Singleton UNIQUEINSTANCE =
7                  new Singleton ();
8
9          // Constructors
10
11         // Initialisiert die Klasse Singleton
12         private Singleton()
13         {
14                 super ();
15         }
16
17         // Methods
18
19         // Operationsmethode
20         public void opearation()
21         {
22                 // Singleton - Operation
23         }
24 }
```

Abb. 1.5: Zweite Variante der Klasse Singleton

Ein großer Vorteil dieser Variante des Entwurfsmusters ist neben der Einfachheit der Implementierung die Threadsicherheit. Bevor die Anwendung überhaupt startet und Threads parallel auf das Singleton zugreifen können, wird das Objekt erstellt. Somit ist eine teure Synchronisierung von *getInstance()* nicht notwendig.
Jedoch fällt bei einem kritischen Blick auf, dass bei einem vorgezogenem Laden (*Eager Loading*) die Gefahr von verfrühter oder gar unnötiger Instanziierung besteht. Diese Problematik ist besonders bei Singletons, deren Erstellung mit einem umfangreichen und ressourcenintensiven Vorgang einhergehen, relevant. Ebenfalls spricht gegen das vorzeitige Laden, dass zur statischen Initialisierungszeit noch nicht alle nötigen Informationen zur Initialisierung des Singletons bereitstehen können. Das Singleton kann Werte benötigen, die erst im Zuge des Programmablaufs verfügbar sind.
Sinnvoll ist das *Eager Loading*, wenn man relativ kleine Singletons mit einfachem Erstellungsprozess hat, die mehrfach gebraucht werden.

1.7 Anwendungsbeispiel

In diesem Unterkapitel möchten wir Ihnen nun die Wichtigkeit des Singleton Entwurfsmusters anhand eines Szenarios zum Ausdruck zu bringen und den dazugehörigen Quellcodeteilausschnitt vorstellen.
Auf einem Computer können Protokolldateien, auch Logdateien genannt, bestimmter Aktionen von mehreren Usern an einem PC geschrieben werden, ohne dass diese es bemerken oder ihre Arbeit beeinträchtigt wird.
Abgesehen vom Betriebssystem schreiben meist Programme aus dem Hintergrund in Logdateien, um Aktionen von Prozessen oder Fehler oder Hinweise zu melden. Zudem werden diese Benachrichtigungen persistent oder zur Verfügung gestellt.
Um den Rahmen dieses Beispiels nicht zu sprengen und den Einsatz des Singleton Design Patterns nicht aus den Augen zu verlieren, möchten wir uns nicht mit der Implementierung und Anwendung der System–Protokolldatei beschäftigen. Stattdessen zeigen wir Ihnen das anhand eines komprimierten Beispiels.

Das folgende UML – Klassendiagramm zeigt den Aufbau der Klassen, die für die Implementierung unseres Beispiels notwendig sein wird.

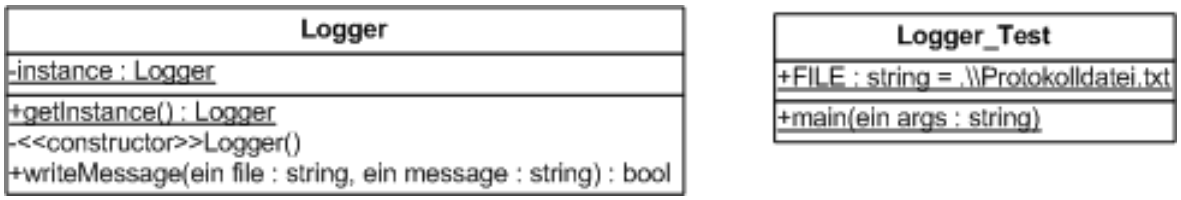


Abb. 1.6: UML-Klassendiagramme für das Protokolldateibeispiel

Die Klasse Logger (s. Abb. 1.7) enthält eine private statische Referenz vom Typ Logger, eine öffentlich statische Operation, die einen Wert vom Typ Logger zurückliefert und einen privaten Konstruktor. Außerdem beinhaltet die Klasse eine öffentliche Operation, die zwei Eingangsparameter erhält: *file* und *message*. Beide sind vom Typ String.
Die Klasse Logger_Test hingegen besitzt eine öffentliche unveränderbare statische Variable vom Typ String. Sie wird mit .\\Protokolldatei.txt initialisiert. Die öffentlich statische Operation *main* erhält standardgemäß einen Eingangsparameter *args* vom Typ Feld String.
Die folgende Abbildung zeigt die Implementierung dieser beiden Klassen in Verwendung der *Lazy Loading* Variation.

```
1  import java.io.BufferedWriter;
2  import java.io.FileWriter;
3
4  // Protokolldatei in Form eines Singleton Entwurfsmusters
5  public class Logger
6  {
7          // Statische Referenz fuer das Singleton Entwurfsmuster
8          private static Logger instance;
9
10         // Liefert das einzige Exemplar von der Klasse Logger zurueck
11         public static Logger getInstance()
12         {
13                 // Pruefung, ob das Objekt bereits initialisiert ist
14                 if ( Logger.instance == null )
15                 {
16                         // Initialisiert falls die Referenz null ist
17                         Logger.instance = new Logger ();
18                 }
19                 return Logger.instance; // Liefert die Referenz zurueck
20         }
21
22         // Constructors
23         private Logger()
24         {
25                 super (); // Konstruktor wurde auf private modifiziert
26         }
27
28         // Speichert eine Nachricht
29         public boolean WriteMessage( String file, String message )
30         {
31                 try
32                 {
33                         FileWriter fw = new FileWriter ( file, true );
34                         BufferedWriter bw = new BufferedWriter ( fw );
35                         bw.write ( message );
36                         bw.newLine ();
37                         bw.close ();
38                         fw.close ();
39                         return true;
40                 }
41                 catch ( Exception ex )
42                 {
43                         return false;
44                 }
45         }
46 }
```

Abb. 1.7: Implementierung der Klasse Logger

```
1  // Programmablauf fuer das Singleton Entwurfsmuster
2  public class Logger_Test
3  {
4          // Static Final Fields
5
6          public static final String FILE = ".\\Protokolldatei.txt";
7
8          // Static Methods
9
10         // Methode startet Testlauf
11         public static void main( String[] args )
12         {
13                 Logger.getInstance (). WriteMessage (
14                         Logger_Test.FILE,
15                         "Das Protokollprogramm wurde gestartet.");
16         }
17 }
```

Abb. 1.8: Implementierung der Klasse Logger_Test

Die Klasse Logger enthält die Operation *getInstance*, die dafür zuständig ist, dass es nur genau ein Exemplar von Logger gibt. Die Operation *WriteMessage* hingegen ist für das Anlegen der Datei Protokolldatei.txt und das Schreiben von Meldungen in diese zuständig. Dabei wird die Technik der Ströme verwendet. Es werden die Protokolle in Form einer Historie in die Datei abgelegt, um den Softwareentwickler allmögliche Meldungen (Fehler, Hinweise oder Aktionen) in zeitlicher Reihenfolge aufzulisten.
Die Klasse Logger_Test ist wie der Name schon bereits sagt die Ausführungsklasse. Sie enthält eine Hauptmethode, in der die *getInstance* Operation der Klasse Logger aufgerufen wird. Zudem wird die Operation *WriteMessage* Operation aufgerufen und zwei Parameter übergeben – Dateiname und die Meldung, die in die Datei geschrieben werden sollen. In diesem Anwendungsbeispiel erscheint nach der Ausführung des Programms die Meldung „Das Protokollprogramm wurde gestartet.“ in der Protkolldatei.txt – Datei.

1.8 Konsequenzen

Simple Anwendung: Eine Singletonklasse ist schnell und unkompliziert geschrieben.

Gegenüber *globalen Variablen* ergeben sich eine Reihe von Vorteilen:

Zugriffskontrolle: Das Singleton kapselt seine eigene Erstellung und kann damit genau kontrollieren, wann und wie Zugriff auf das Singleton erlaubt wird.

Sauberer Namensraum: Der Namensraum wird nicht mit unzähligen globalen Variablen überfrachtet, sondern gekapselt in einem Singleton bereitgestellt.

Spezialisierung: Ein Singleton kann abgeleitet werden, um ihm neue Funktionalität zuweisen zu können. Die Integration in bestehenden Code gestaltet sich einfach. Welche Unterklasse genutzt werden soll, kann dynamisch zur Laufzeit entschieden werden.

Lazy-Loading: Singletons können erst erzeugt werden, wenn sie auch wirklich gebraucht werden. **Prozedurales Programmieren:** Die zahlreiche Anwendung von Singletons führt zu einem ähnlich ungünstigen Zustand wie bei globalen Variablen. Dies entspricht der prozeduralen Programmierung und hat nichts mit Objektorientierung und Kapselung zu tun.

Globale Verfügbarkeit: Durch die globale Verfügbarkeit wird sichergestellt, dass das Entwurfsmuster Singleton *überall* in der Anwendung verfügbar ist. Enthält das Singleton nun Daten, so ist dies ein sehr fragwürdiges Modell. Daraus lassen sich drei grundlegende Fragen resultieren. Welche Daten können es sein, die in allen Schichten verfügbar sein sollen? Kann es nicht sogar gefährlich sein, bestimmte Daten oder Operationen überall frei verfügbar zu machen? Kann man sie nicht doch einer Schicht sauber zu ordnen und jede Schicht hinter wohldefinierten Schnittstellen und Datenaustausch kapseln? Singletons verleiten den Softwareentwickler zu einer unsauberen Programmierung. Daher sollte die Notwendigkeit von Singletons stets hinterfragt werden.

Problematisches Zerstören: Um in Sprachen wie C# mit *Garbage Collection* Objekte zu zerstören, darf ein Objekt nicht mehr referenziert werden und damit null zurückliefern. Dies ist bei Singletons schwierig sicherzustellen. Durch die globale Präsenz, passiert es sehr schnell, dass Codeteile noch eine Referenz auf das Singleton halten.

Senkung der Performance: Vor allem bei Mehrbenutzeranwendungen kann ein Einzelstück die Leistung senken, da er - besonders in der synchronisierten Form - ein Flaschenhals darstellt.

Fazit

In objektiver Betrachtung auf das Entwurfsmuster Singleton ist festzustellen, dass es sich um ein prozedurales Relikt im objektorientierten Gewand handelt. Die oft bedingungslose und globale Verfügbarkeit widerspricht vielem, was Objektorientierte Programmierung ausmacht (Kapselung, Schnittstellen, Schichten, Wiederverwendbarkeit etc.). Seine Verwendung sollte wohlüberlegt sein und sich auf Fälle auf einmaligen Strukturen beschränken, die keinen Zustand besitzen.

2 Delegation

Delegation erlaubt es einer Klasse Funktionen eines anderen Objekts zu benutzen.

Dieses Kapitel beschreibt das Design Pattern Delegation. Es werden die allgemeinen Informationen erläutert. Anschließend zeigt ein Beispiel das Entwurfsmuster. Zum Schluss wird dieses Design Pattern anhand von positiven und negativen Aspekten objektiv beurteilt.

Mit Vererbung kommt man bei einigen Konzepten in der objektorientierten Programmierung schnell an Deklarationsgrenzen und zu Fehlinterpretationen, hier bittet es sich an Delegation zu benutzen.

2.1 Definition

An implementation mechanism in which an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object. [Gam94]

Das Design Pattern Delegation ist ein Entwurfsmuster, das für die Wiederverwendung in Quellcode ein große Rolle spielt. Hierbei wird die Technik der Objektkomposition angewandt, welches Objekte bestehender Klassen in eine Struktur einbezieht. [Win98]

2.2 Motivation

Im Rahmen des Software Engineering werden in der Entwicklung unter anderem Implementierungen vorgenommen. Dabei werden unterschiedliche Techniken angewendet, um den Quellcode an die Anforderungen des Kunden anzupassen. Eine dieser Techniken ist die Vererbung. Mit Hilfe dieser können einige Quellcodeabschnitte wiederverwendet und somit die Trennung von Dienstanbieter und Dienstnutzer garantiert werden. Dieses genannte Konzept erfordert eine Bindung schon vor der Laufzeit.

Wie kann man die lose Bindung eines stellvertretenden Anbieters von Funktionalität und eines Nutzen derselben erreichen? Mit Hilfe der Delegation wird das Erzeugen eines Objekts eines Anbieters ermöglicht und somit der Zugriff auf dessen Operationen gewährt.

2.3 Absicht

Das Ziel dieses Entwurfsmusters ist die Vererbung zu vermeiden, einfache Quelltextstrukturen zu erzeugen und die Algorithmen als wiederverwendbar zu implementieren.

Verhinderung der Vererbung

Wird in einer Programmiersprache die Mehrfachvererbung nicht gestattet, bietet sich die Lösung an Klassen durch Referenzen anzusprechen.

Einfacher Quellcode

Die komplizierten Algorithmen können aufgrund der Übersichtlichkeit in Klassen aufgeteilt werden.

Wiederverwendbarkeit der Algorithmen

Bei Implementierung von komplexen Systemen kann durch dieses Design Pattern die Berechnung in Bereiche unterteilt werden. [Pag11]

2.4 Struktur

Dieser Teil befasst sich mit der Struktur des Entwurfsmusters Delegation. Die Grundidee ist es Funktionen nicht in Form von Vererbung, sondern durch Referenzen zur Verfügung zu stellen. In folgender Abbildung wird die allgemeine Beziehung gezeigt.

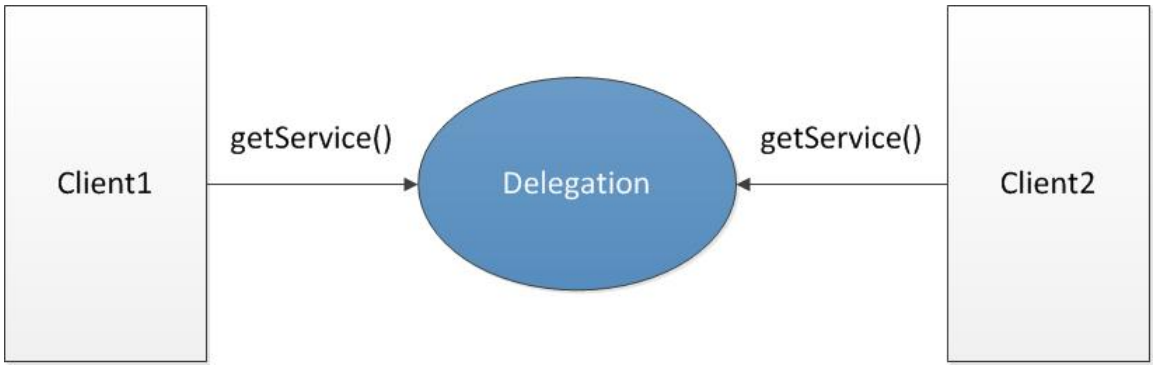


Abb. 2.1: Allgemeine Darstellung der Delegation

Die Graphik beschreibt die Struktur des Design Patterns in Hinblick auf ihre Abhängigkeiten. Mehrere Clients können die Funktionen einer Delegation benutzen, aber ein Client darf nur eine Delegation besitzen. Es sind auch mehrere Objekte der Klasse Delegation erlaubt.

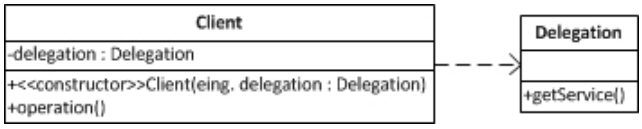


Abb. 2.2: UML Darstellung der Delegation

Wie man sieht wird die Referenz für die Delegation den Client als Parameter im Konstruktor übergeben. Nach der Initialisierung des Clients stehen ihm die Funktionalitäten der Delegation zum Aufrufen bereit.

2.5 Beispiel

Folgend wird das Entwurfsmuster anhand eines einfachen Beispiels erläutert.

2.5.1 Szenario

Das häufig vorkommende Problem ist die Vererbung, die die Rollen der Klassen einschränkt und die Wiederverwendung dramatisch sickt. In einem Unternehmen gibt es für die Person verschiedene Rollen wie Abteilungsleiter, Angestellter und Kunde.

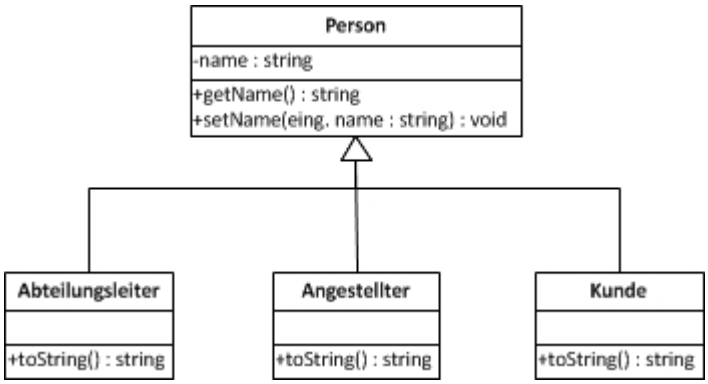


Abb. 2.3: Grenzen der Vererbung für Rollen der Klassen

Die obige Abbildung zeigt die Einschränkung der Vererbung. Nach objektorientierter Modellierung ist die Klassenaufteilung gut strukturiert, doch wenn man genauer überlegt, zeigen sich schon die Mängel, denn ein und dieselbe Person kann z.B. ein Abteilungsleiter und zugleich ein Angestellter sein. Um diese Fälle auch zu beachten, müsste es noch die Klassen AbteilungsleiterUndAngestellter, AbteilungsleiterUndKunde, AngestellterUndKunde und AbteilungsleiterAngestellterUndKunde geben. Durch die Verwendung von Delegation wird dieser enorme Implementierungsaufwand reduziert.

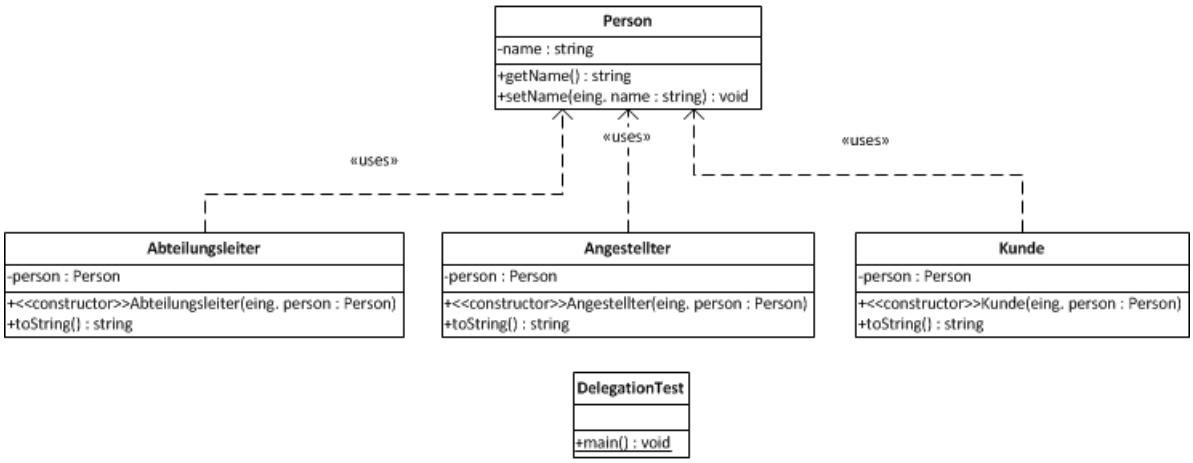


Abb. 2.4: Person für alle Rollen als Referenz

Nun kann eine Person die drei Rollen Abteilungsleiter, Angestellter und Kunde annehmen, ohne jedes Mal eine Fallunterscheidung durchzuführen. Die Rollenklassen können auf alle Funktionalitäten der Klasse Person zugreifen.

2.5.2 Implementierung

Anschließend folgt die Implementierung des Beispiels mit Java Code. Es werden die Klassen Person, Abteilungsleiter, Angestellter, Kunde und DelegationTest dargestellt. Danach kommt die Ausgabe auf der Konsole als Abbildung.

```
1  // Definition einer Person
2  public class Person
3  {
4          // Fields
5
6          private String name;
7
8          // Methods
9
10         // Liefert den Namen
11         public String getName()
12         {
13                 return this.name;
14         }
15
16         // Setzt den Namen
17         public void setName( String name )
18         {
19                 this.name = name;
20         }
21 }
```

Abb. 2.5: Implementierung der Klasse Person

```java
// Rolle fuer eine Person
public class Abteilungsleiter
{
        // Fields

        private Person person;

        // Constructors

        // Initialisiert die Klasse Abteilungsleiter
        public Abteilungsleiter( Person person )
        {
                this.person = person;
        }

        // Methods

        // Liefert die Infos der Klasse als Zeichenkette
        public String toString()
        {
                // toString() Modifikation fuer Abteilungsleiter
                return "Der Abteilungsleiter heisst " +
                        this.person.getName () + ".";
        }
}
```

Abb. 2.6: Implementierung der Klasse Abteilungsleiter

```java
// Rolle fuer eine Person
public class Angestellter
{
        // Fields

        private Person person;

        // Constructors

        // Initialisiert die Klasse Angestellter
        public Angestellter( Person person )
        {
                this.person = person;
        }

        // Methods

        // Liefert die Infos der Klasse als Zeichenkette
        public String toString()
        {
                // toString() Modifikation fuer Angestellter
                return "Der Angestellter heisst " +
                        this.person.getName () + ".";
        }
}
```

Abb. 2.7: Implementierung der Klasse Angestellter

```
1  // Rolle fuer eine Person
2  public class Kunde
3  {
4          // Fields
5
6          private Person person;
7
8          // Constructors
9
10         // Initialisiert die Klasse Kunde
11         public Kunde( Person person )
12         {
13                 this.person = person;
14         }
15
16         // Methods
17
18         // Liefert die Infos der Klasse als Zeichenkette
19         public String toString()
20         {
21                 // toString() Modifikation fuer Kunde
22                 return "Der Kunde heisst " +
23                         this.person.getName () + ".";
24         }
25 }
```

Abb. 2.8: Implementierung der Klasse Kunde

```
1  // Testprogramm fuer das Entwurfsmuster Delegation
2  public class DelegationTest
3  {
4          // Static Methods
5
6          // Startet den Testlauf
7          public static void main( String[] args )
8          {
9                  // Initialisierung der Person
10                 Person person = new Person ();
11                 person.setName ( "Thomas Mueller" );
12
13                 // Innitialisierung der Rollen
14                 Abteilungsleiter abteilungsleiter =
15                         new Abteilungsleiter ( person );
16                 Angestellter angestellter =
17                         new Angestellter ( person );
18                 Kunde kunde = new Kunde ( person );
19
20                 // Ausgabe der Daten auf der Konsole
21                 System.out.println ();
22                 System.out.println ( abteilungsleiter.toString () );
23                 System.out.println ( angestellter.toString () );
24                 System.out.println ( kunde.toString () );
25         }
26 }
```

Abb. 2.9: Implementierung der Klasse DelegationTest

```
Der Abteilungsleiter heisst Thomas Mueller.
Der Angestellter heisst Thomas Mueller.
Der Kunde heisst Thomas Mueller.
```

Abb. 2.10: Konsolenausgabe des Programmdurchlaufs von DelegationTest

Die obigen Abbildungen zeigen wie Delegation auf Codeebene aussieht und benutzt wird. Alle Rollen greifen auf die Funktionalität der Person zu, um den Namen der Person zu bekommen.

Falls der Name der Person geändert wird, wird die richtige Verwendung der Funktion setName() aus der Klasse Person sichergestellt.

2.6 Konsequenzen

Des Weiteren können folgende positiven und negativen Merkmale festgelegt werden. Darauf folgt eine objektive Betrachtung auf das Entwurfsmuster Delegation.

- Delegation vereinfacht die Zusammensetzung von Verhalten zur Laufzeit und erleichtert die Änderung der Zusammensetzungsstruktur. [Gam94]
- Wiederverwendbarkeit durch Objektkomposition anstelle der Vererbung.
- Die Codestruktur kann durch Referenzen einfach dargestellt werden.
- Dieses Entwurfsmuster ist dynamisch und somit schwieriger zu verstehen als statischer Code. [Gam94]
- Je mehr die Technik Delegation angewendet wird, desto komplexer werden die Beziehungen zwischen den Klassen.

Fazit

Bei der objektiven Feststellung ist das Entwurfsmuster Delegation ein fester Bestandteil der objektorientierten Programmierung. Es ist schon fast ein standardisiertes Design Pattern. Durch die einfache Kapselung der Klassen versimpelt sich die Klassenstruktur. Bei jeder Vererbung sollte man sich die Frage stellen, ob eine Delegation nicht vernünftiger wäre.

3 Factory Pattern

Unter den vielen verschiedenen Design Patterns gibt es zwei ähnlich lautende Patterns. Zum einen die Factory Method, zum anderen die Abstract Factory. Beide sind ähnlich aufgebaut und doch haben sie ein wenig andere Einsatzgebiete. Diese eben genannten Patterns werden hinsichtlich Absicht, Struktur, eines Beispieles und einer Bewertung in diesem Kapitel erläutert.

3.1 Factory Method

Starten wir zunächst einmal mit der Factory Method. Diese besitzt folgende Absichten.

3.1.1 Absichten

- Sich ändernde Eigenschaften zu kapseln
- Kompositionen der Vererbung vorziehen
- Gegen ein Interface statt gegen eine Implementierung zu programmieren
- Lose gekoppeltes Design zu erhalten
- Klassen für Erweiterungen öffnen und gegen Veränderung schließen

Aufgrund dieser Absichten wird sie meist bei Toolkits und Frameworks oder einfach nur bei der Erzeugung von Objekten verwendet.

3.1.2 Struktur

Die Factory Method verfolgt einen recht einfachen Aufbau (siehe Abb.3.1)

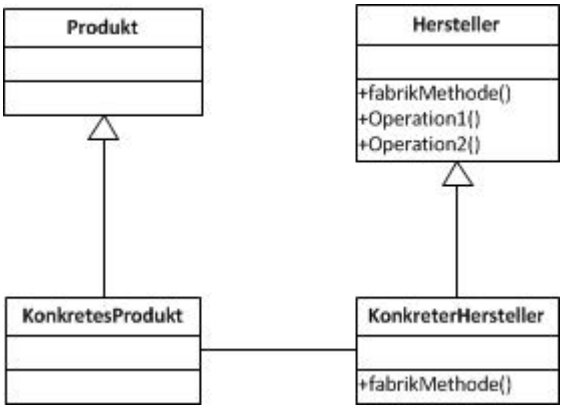


Abb. 3.1: Aufbau des Factory Method Pattern

Es besteht ein Interface oder eine abstrakte Klasse, in diesem Falle das Produkt. Es gibt die Gundzüge vor bzw. kann Methoden bereit stellen.
Diese Schnittstelle wird nun durch das konkrete Produkt implementiert.
Ähnlich wie zuvor ist auch das Schema aus der produzierenden Schicht aufgebaut, hier existiert eine abstrakte Klasse Hersteller, welche Standartoperationen bereit stellt. Des Weiteren beinhaltet diese auch die Vorlage für eine Fabrikmethode, welche das erzeugen von konkreten Produkten übernehmen soll.
Die gerade erwähnte Fabrikmethode wird nun von einer konkreten Fabrik wieder aufgegriffen. Die KonkreteFabrik ist von dem abstrakten Hersteller abgeleitet und implementiert nun die Fabrikmethode. Diese Klasse erzeugt nun die konkreten Produkte.
Ein Client würde gegen das Produkt und den Hersteller, also die Schnittstellen, programmiert werden und über eine konkrete Instanz einer Fabrik ein konkretes Produkt erhalten.

3.1.3 Beispiel

Als Beispiel wird eine bekannte Imbissbudenkette angenommen, die ihr Systemprogramm für ihre Imbissbuden erneuern will. Bisher gibt es als Produkte

- Currywurst
- Döner

Es wird gefordert dass

- die Produktpalette leicht erweitert werden kann
- einzelne Imbissbuden die Produkte, aufgrund regionaler Vorlieben, steuern können

Durch die eben genannten Vorgaben wird entschieden ein Factory Pattern zu verwenden. Für die generellen Gerichte, wie Döner und Currywurst, wird ein Interface Gericht implementiert.

```
1  public interface Gericht {
2          public void vorbereiten();
3          public void zubereiten();
4          public void servieren();
5          public void kassieren();
6  }
```

Abb. 3.2: Code für das Interface Gericht

Nun werden die Gerichte implementiert, z.B. der Döner aus Aschaffenburg

```
1  public class DoenerAB implements Gericht{
2          public DoenerAB()
3          {
4                  System.out.println("DoenerAB");
5          }
6          @Override
7          public void vorbereiten() {
8                  System.out.println("DoenerAB vorbereiten");
9          }
10         @Override
11         public void zubereiten() {
12                 System.out.println("DoenerAB zubereiten");
13         }
14         @Override
15         public void servieren() {
16                 System.out.println("DoenerAB servieren");
17         }
18         @Override
19         public void kassieren() {
20                 System.out.println("DoenerAB kassieren");
21         }
22 }
23 }
```

Abb. 3.3: Code für den Döner aus Aschaffenburg

oder die Currywurst aus Würzburg.

```
1  public class CurrywurstWue implements Gericht{
2          protected String Name;
3          public CurrywurstWue()
4          {
5                  Name="CurrywurstWue";
6          }
7          @Override
8          public void vorbereiten() {
9                  System.out.println("CurrywurstWue vorbereiten");
10         }
11         @Override
12         public void zubereiten() {
13                 System.out.println("CurrywurstWue zubereiten");
14         }
15         @Override
16         public void servieren() {
17                 System.out.println("CurrywurstWue servieren");
18         }
19         @Override
20         public void kassieren() {
21                 System.out.println("CurrywurstWue kassieren");
22         }
23 }
24 }
```

Abb. 3.4: Code für den Döner aus Aschaffenburg

Zusammen mit anderen Produkten sind dies konkrete Produkte, die erzeugt werden müssen.
Nun benötigen wir auch eine Fabrik, die uns diese Produkte erzeugt. Um dabei für regionale Erweiterungen offen zu sein, benötigen wir eine abstrakte Klasse, die allgemeine Aufgaben der Imbissbuden erledigt.

```
1  public abstract class Imbissbude {
2          public abstract Gericht waehleGericht(String typ);
3          public void bestellen(String typ)
4          {
5                  Gericht bestellung;
6                  bestellung = waehleGericht(typ);
7                  bestellung.vorbereiten();
8                  bestellung.zubereiten();
9                  bestellung.servieren();
10                 bestellung.kassieren();
11         }
```

Abb. 3.5: Code für die abstrakte Klasse Imbissbude

Für die regionalen Imbissbuden wird jeweils eine Fabrik implementiert, welche die Auswahl der regionalen Gerichte übernimmt. Hier am Beispiel der Imbissbude in Aschaffenburg.

```
1  public class ImbissbudeAB extends Imbissbude{
2          public Gericht waehleGericht(String typ)
3          {
4                  System.out.println("ueber Fabrikmethode implementiert in
5                                                          Aschaffenburg erzeugen");
6                  Gericht produziere = null;
7                  if(typ.equals("Currywurst"))
8                          produziere = new CurrywurstAB();
9                  else if (typ.equals("Doener"))
10                         produziere = new DoenerAB();
11                 return produziere;
12         }
13 }
```

Abb. 3.6: Code für die konkrete Fabrik ImbissbudeAB

Nun haben wir das Factory Method Pattern implementiert. Ein etwaiger Client programmiert nur gegen das Interface Gericht und die abstrakte Klasse Imbissbude. Die genauen Produkte erzeugt er über die jeweiligen regionalen Imbissbuden.

```
1  public class Client {
2          public static void main(String[] args) {
3                  Imbissbude bude = new ImbissbudeWue();
4  %               System.out.println("Wuerzburg");
5                  System.out.println("Bestelle Currywurst");
6                  bude.bestellen("Currywurst");
7                  System.out.println("Bestelle Doener");
8                  bude.bestellen("Doener");
9                  bude = new ImbissbudeAB();
10                 System.out.println("Aschaffenburg");
11                 System.out.println("Bestelle Currywurst");
12                 bude.bestellen("Currywurst");
13                 System.out.println("Bestelle Doener");
14                 bude.bestellen("Doener");
15         }
16 }
```

Abb. 3.7: Code für den Client

3.1.4 Bewertung

Vorteile

- Parallele Hierarchien strukturierbar
- Leichter Austausch einer Fabrik, die leicht andere Produkte erzeugen kann
- Entkoppelt Objekterzeugung von der Logik, daher einfache Erweiterung

Nachteile

- Neue Fabriken müssen explizit programmiert werden
- Der Aufwand um einfache Elemente zu erzeugen ist hoch
- Durch die vielen Subklassen steigt die Komplexität des Programmes

3.2 Abstract Factory

Fahren wir nun fort mit der Abstract Factory. Ihre Absichten lauten wie folgt:

3.2.1 Absichten

- System soll unabhängig von Produkterzeugung, Zusammensetzung und Darstellung sein
- System soll mit mehrere Produktfamilien arbeiten
- Konsistenz der Produkte soll sichergestellt werden
- Schnittstellen sollen sichtbar sein, nicht deren Implementierung

Aus diesem Grunde wird die Abstract Factory z.B. bei plattformunabhängigen Toolkits verwendet. Bei deren Verwendung interessiert nicht wie man seine Objekte bekommt sondern nur dass man sie bekommt. Des Weiteren sollen sie erweiterbar für neue Plattformen sein.

3.2.2 Struktur

Eine Abstract Factory ist wie folgt aufgebaut.

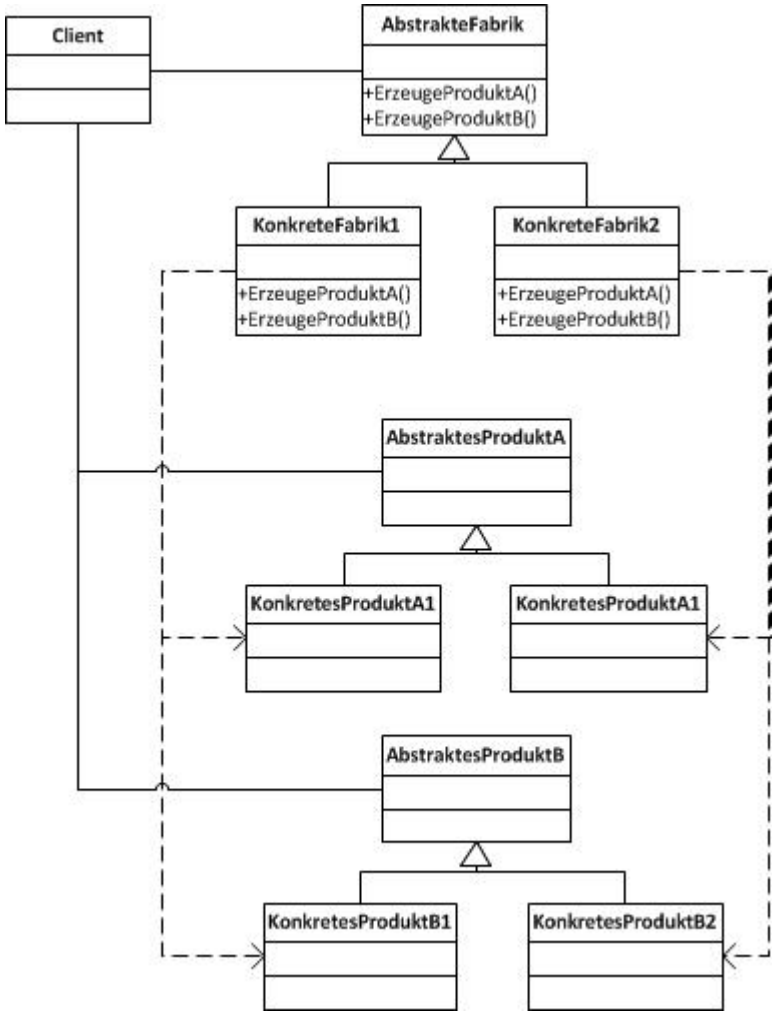


Abb. 3.8: Aufbau des Abstract Factory Pattern

Die abstrakte Klasse AbstrakteFabrik gibt die zu implementierenden Fabrik Methoden und allgemeine Grundaufgaben vor.
Die konkreten Fabriken implementieren die AbstrakteFabrik und erstellen jeweils die Produkte.
Die Produkte haben je Produktfamilie ein Interface und pro Spezialisierung ein konkretes Produkt.
Wie man hier anhand des Aufbaues sieht, wird zur Erzeugung einer Abstract Factory auf das Factory Method Pattern zurückgegriffen. Somit kann man sagen, Abstract Factory ist eine Art Erweiterung des Factory Method Patterns.

3.2.3 Beispiel

Im Beispiel beziehen wir uns wieder auf die Imbissbude, die auch schon im Factory Method Beispiel verwendet wurde. Nur diesmal sollten die einzelnen Zutaten über eine Produktfamilie erzeugt werden.
Sehen wir uns hierzu das Ganze anhand des Beispieles der Currywurst an.
Es besteht nach wie vor das Interface Gericht, welches die Vorgaben für die einzelnen Gerichte darstellt.
Nun benötigen wir für das Gericht Currywurst zunächst eine abstrakte Klasse, welche als Grundbaustein für die speziellen Arten von Currywurst dient.

```
1  public class Currywurst implements Gericht{
2          private Sosse sosse;
3          private Wurst wurst;
4          private Beilage beilage;
5          public Currywurst(CurrywurstZutatenFabrik zutatenfabrik)
6          {
7                  wurst = zutatenfabrik.erstelleWurst();
8                  sosse = zutatenfabrik.erstelleSosse();
9                  beilage = zutatenfabrik.erstelleBeilage();
10                 System.out.println("Currywurst mit " + wurst.getName()
11                 + ", "+ sosse.getName() + " und "+ beilage.getName());
12         }
13         @Override
14         public void vorbereiten() {
15                 System.out.println("Currywurst vorbereiten");
16         }
17         @Override
18         public void zubereiten() {
19                 System.out.println("Currywurst zubereiten");
20         }
21         @Override
22         public void servieren() {
23                 System.out.println("Currywurst servieren");
24         }
25         @Override
26         public void kassieren() {
27                 System.out.println("Currywurst kassieren");
28         }
29 }
```

Abb. 3.9: Code für die abstrakte Klasse Currywurst

Nun benötigen wir für die Zutaten eine Zutatenfabrik. Da es verschiedene regionale Zutatenvariationen gibt, erzeugen wir zunächst wieder eine abstrakte Klasse.

```
1  public interface CurrywurstZutatenFabrik{
2
3          public Beilage erstelleBeilage();
4          public Sosse erstelleSosse();
5          public Wurst erstelleWurst();
6  }
```

Abb. 3.10: Code für die abstrakte Klasse Zutatenfabrik

Für die einzelnen Zutatenvariationen gibt es ja nach Zutat wieder ein Interface, wie hier am Beispiel von Beilage.

```
1  public interface Beilage {
2          public String getName();
3  }
```

Abb. 3.11: Code für das Interface einer Zutatengruppe

Nun kreieren wir die konkreten Zutaten, wie hier im Falle eines Brötchens

```
1  public class Broetchen implements Beilage{
2          public String getName(){
3                  return "Broetchen";
4          }
5  }
```

Abb. 3.12: Code für ein konkretes Produkt

Nun, da wir die Zutaten implementiert haben, benötigen wir wieder eine konkrete Fabrik, die uns die Zutaten erzeugt.

```
1  public class CurrywurstABZutatenFabrik implements CurrywurstZutatenFabrik{
2          public Beilage erstelleBeilage(){
3                  return new Broetchen();
4          }
5          public Sosse erstelleSosse(){
6                  return new KetchupCurrySosse();
7          }
8          public Wurst erstelleWurst(){
9                  return new Rindswurst();
10         }
11 }
```

Abb. 3.13: Code für eine konkrete Zutatenfabrik

Die konkreten Ausprägungen der Imbissbude müssen jetzt nur noch eine spezielle Zutatenfabrik auswählen und können so verschiedene Produktvariationen erzeugen.

```
1  public class ImbissbudeAB extends Imbissbude{
2          public Gericht waehleGericht(String typ)
3          {
4                  Gericht produziere = null;
5                  CurrywurstZutatenFabrik zutatenfabrik = new CurrywurstABZutatenFabrik();
6                  if(typ.equals("Currywurst"))
7                          produziere = new Currywurst(zutatenfabrik);
8                  return produziere;
9          }
10 }
```

Abb. 3.14: Code für eine konkrete Zutatenfabrik

3.2.4 Bewertung

Vorteile

- konkrete Klassen werden isoliert, somit erscheinen sie nicht im Clientcode
- Austausch einer Produktfamilie ist durch Austausch einer Fabrik vereinfacht
- Stellt Konsistenz unter Produkten sicher

Nachteil

Erzeugung neuer Produktarten ist schwierig, da alle Fabriken geändert werden müssen.

4 Decorator Pattern

4.1 Einleitung

Im Folgenden wird das Decorator Pattern vorgestellt, das in die Kategorie der Strukturmuster eingeordnet wird. Es dient der dynamischen Erweiterung von Basisklassen um weitere Funktionalitäten und ermöglicht außerdem eine Art Mehrfachvererbung ohne die bestehenden Vererbungshierarchien zu verletzen.

4.2 Struktur

Nachdem die Ziele des Decorator Patterns bekannt sind, gehen wir in diesem Kapitel der Frage nach, wie das Pattern aufgebaut ist. In Abbildung 4.1 wird der Aufbau in Form eines UML-Diagrams veranschaulicht.

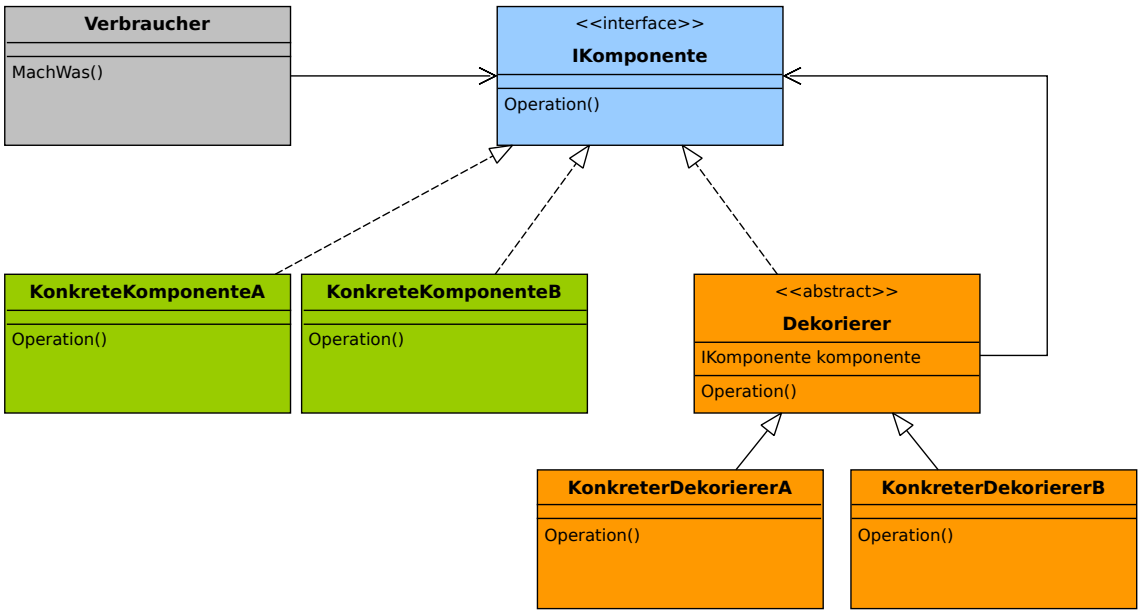


Abb. 4.1: Struktur des Decorator Patterns

Das gesamte Pattern basiert auf dem gemeinsamen Interface *IKomponente*. Der *Verbraucher* programmiert später immer gegen diese Schnittstelle. Wie in der Abbildung 4.1 zu sehen, implementiert jede Komponente und jeder Dekorierer diese Schnittstelle. Dieser Abstraktionsschritt erlaubt später, dass der *Verbraucher* mit einer konkreten Komponente oder einem Dekorierer arbeiten kann.

Die grün dargestellten Klassen *KonkreteKomponenteA* und *KonkreteKomponenteB* implementieren je eine Basisoperation. Objekte dieser Klassen sind eigenständig und könnten auch direkt vom *Verbraucher* verwendet werden.

Die orange dargestellten Klassen stellen die Dekorierer dar. Diese können die Basisorperation einer konkreten Klasse erweitern oder ersetzen. Dazu benötigen diese jedoch zwingend ein Objekt einer konkreten Komponente. Dieses Objekt wird in der Klasse *Dekorierer* als Attribut *komponente* vom Typ *IKomponente* abgespeichert. Das Interface *IKomponente* ermöglicht es, dass ein Dekorierer einen weiteren Dekorierer als Attribut *komponente* enthalten kann. Somit können die Dekorierer beliebig verschachtelt werden. Das innere Ende der Verschachtelung ist aber in jedem Fall ein Objekt einer konkreten Komponente. In einer Verschachtelung können die Dekorierer die konkrete Komponente in beliebiger Reihenfolge umhüllen und auch mehrfach verwendet werden. Je nach Anwendungsfall macht jedoch nicht jede Reihenfolge Sinn.

Die eigentliche Erweiterung einer Basisoperation geschieht in der Implementation der *Operation()*-Methode im konkreten Dekorierer, wie beispielsweise *KonkreterDekorierer*A. Im Normalfall wird hier die *Operation()*-Methode der zu erweiternden Komponente (gespeichert im Attribut *komponente*) aufgerufen. Vor und oder nach diesem Aufruf kann der Dekorierer zusätzliche Funktionalität implementieren. So können beispielsweise Methodenparameter oder Rückgabewerte angepasst oder ausgewertet werden. Es ist aber genauso möglich die gesamte Implementation der konkreten Komponente zu ersetzen.

4.3 Beispiele

Wir wissen nun bereits dass das Decorator-Pattern zum dynamischen Hinzufügen von Funktionalitäten dient und wie es diese Anforderung realisiert. Um nochmals zu verdeutlichen wann beziehungsweise wie das Pattern verwendet wird, zeigen wir hier eine Beispielimplementierung und danach bekannte Beispiele der realen Welt.

4.3.1 Beispielimplementierung

Dazu stellen wir uns vor, wir sind bei unserer Imbissbude um die Ecke und haben Hunger auf einen Döner. Bei der Imbissbude gibt es nicht einfach nur einen Döner, sondern man kann zwischen den Varianten Dürüm und Brottasche auswählen. Hat man sich beispielsweise für die Brottaschen-Variante entschieden, fragt der Verkäufer nach der Fleischsorte. Am heutigen Tag kann man neben dem normalen Rindfleisch auch Geflügelfleisch auswählen. Da wir uns nicht entscheiden können, mixen wir einfach beide Fleischsorten. Nachdem der Verkäufer den Döner mit Fleisch, sowie verschiedenen Salaten und einer Sauce dekoriert hat, fragt er, ob er auch Schafskäse hinzufügen soll. Dieser kostet zwar Aufpreis, aber da wir Schafskäse lieben, nehmen wir die Mehrkosten gerne in Kauf.

Der Mann in der Schlange hinter uns ist sichtlich von dem Döner angetan und bestellt sich auch gleich einen: „Ich bekomme auch einen Döner mit den selben Zutaten, allerdings als Dürüm und ohne den Schafskäse!“.

Wer sich jetzt fragt, was die Imbissbude mit dem Decorator-Pattern zu tun hat, der soll sich bitte noch einmal die Strukturabbildung 4.1 anschauen. Nun ersetzt man die *KonkreteKomponenteA* mit der Brottasche aus unserem Beispiel, sowie die *KonkreteKomponenteB*

mit dem Dürüm. Jeder Döner baut entweder auf eine Brottasche oder einen Dürüm auf. Eine dieser beiden ist immer enthalten, was sie zu Kandidaten für Komponenten macht (kommt überall genau einmal vor). Die Zutaten des Döners werden als Dekorierer realisiert. Damit können Sie für die Brottasche und den Dürüm verwendet werden. Also könnte *KonkreterDekoriererA* das Geflügelfleisch und *KonkreterDekoriererB* der Schafskäse sein.

Das Interface *IKomponente* implementieren wir in unserem Beispiel wie im Code-Listing 4.2 als *IGericht*. Mit dem Interface *IGericht* sind wir in der Lage den Preis für ein Gericht zu bestimmen und eine kurze Beschreibung auszugeben.

```
1  public interface IGericht {
2     float getPreis();
3     String getBeschreibung();
4  }
```

Abb. 4.2: Gemeinsames Interface IGericht für Brottasche, Geflügelfleisch, ...

```
1  public class Brottasche implements IGericht {
2     float getPreis() {
3        return 3.0f;
4     }
5
6     String getBeschreibung() {
7        return "Brottasche";
8     }
9  }
```

Abb. 4.3: Konkrete Klasse Brottasche, die durch Zutaten erweitert werden kann

Wie im Code-Ausschnitt 4.3 zu sehen, implementiert die Brottasche das gemeinsame Interface *IGericht*. Hier sieht man auch gleich den Grundpreis von 3 Euro.

Da niemand für eine leere Brottasche 3 Euro bezahlen würde, erzeugen wir uns nun die Voraussetzung für unsere Zutaten. Im Listing 4.4 wird die Basisklasse für jede Zutat, also jeden Dekorierer, implementiert. Elementar ist hier, dass jeder Dekorierer eine Referenz auf das zu dekorierende Objekt benötigt. Im Attribut *komponente* wird diese Referenz abgespeichert. Der Konstruktor dient lediglich zum Setzen der Referenz.

```
1  public abstract class GerichtDekorierer implements IGericht {
2
3     //Konstruktor erhaelt das zu dekorierende Gericht!
4     public GerichtDekorierer(IGericht komponente) {
5        this.komponente = komponente;
6     }
7
8     //Jeder Dekorierer greift ueber die Komponente
9     //auf das zu dekorierende Gericht zu
10    protected IGericht komponente;
11 }
```

Abb. 4.4: Basisklasse für jeden Dekorierer

Die Klasse *GefluegelfleischDekorierer* erbt von der Basisklasse *GerichtDekorierer*. Wie in der Implementierung im Code 4.5 zu sehen, wird lediglich der Preis der dekorierten

Komponente zurückgegeben. In unserem Beispiel könnte das der Preis der Brottasche sein. Die Beschreibung der Brottasche wird um das Geflügelfleisch erweitert.

```
1  public class GefluegelfleischDekorierer extends GerichtDekorierer {
2
3     //Konstruktor erhaelt das zu dekorierende Gericht!
4     public GefluegelfleischDekorierer(IGericht komponente) {
5        super(komponente);
6     }
7
8     public float getPreis() {
9        return komponente.getPreis();
10    }
11
12    public String getBeschreibung() {
13       return komponente.getBeschreibung() + ", Gefluegelfleisch";
14    }
15 }
```

Abb. 4.5: GefluegelfleischDekorierer erweitert ein Gericht um Gefluegelfleisch

Wie der *GefluegelfleischDekorierer* auch, erbt der *SchafskaeseDekorierer* (Abbildung 4.6) von der gemeinsamen Basisklasse *GerichtDekorierer*. Da der Schafskäse im normalen Döner nicht enthalten ist, wird hierfür ein Aufpreis verlangt. In diesem Beispiel wird jedoch keine Pauschale aufaddiert, sondern der Preis könnte bei jedem Döner variieren. Dazu wird der Aufpreis im Konstruktor übergeben.

```
1  public class SchafskaeseDekorierer extends GerichtDekorierer {
2
3     //Konstruktor erhaelt das zu dekorierende Gericht!
4     public SchafskaeseDekorierer(IGericht komponente, float aufpreis) {
5        super(komponente);
6        setAufpreis(aufpreis);
7     }
8
9     float aufpreis;
10
11    public float getAufpreis() {
12       return aufpreis;
13    }
14
15    public void setAufpreis(float aufpreis) {
16       this.aufpreis = aufpreis;
17    }
18
19    public float getPreis() {
20       return komponente.getPreis() + getAufpreis();
21    }
22
23    public String getBeschreibung() {
24       return komponente.getBeschreibung() + ", Schafskaese";
25    }
26 }
```

Abb. 4.6: SchafskaeseDekorierer erweitert ein Gericht um Schafskaese

Auf die restlichen Zutaten eines Döners verzichten wir an dieser Stelle, da alle Zutaten auf die selbe Art und Weise wie die beiden vorgestellten Dekorierer implementiert werden.

Der Code-Auszug 4.7 zeigt die Erzeugung und Verwendung meines Döners mit Brottasche, Geflügelfleisch und Schafskäse, sowie die Erzeugung des Döners für den Mann hinter

mir. Die Klasse *Dueruem* ist wie *Brottasche* eine konkrete Klasse mit einem Grundpreis von 3 Euro.

```
IGericht meinDoener;
IGericht andererDoener;

//hier wird mein Doener erzeugt:
meinDoener = new SchafskaeseDekorierer(0.5f,
                new GefluegelfleischDekorierer(
                    new Brottasche()));

System.out.printf(
   "Mein Doener:\n Beschreibung: %s\n Preis: %.2f Euro\n",
      meinDoener.getBeschreibung(), meinDoener.getPreis());

//hier wird der Doener des Mannes hinter mir erzeugt:
andererDoener = new GefluegelfleischDekorierer(
                    new Dueruem());
System.out.printf(
   "\nAnderer Doener:\n Beschreibung: %s\n Preis: %.2f Euro\n",
      andererDoener.getBeschreibung(), andererDoener.getPreis());
```

Abb. 4.7: Erzeugung und Verwendung eines Döner-Gerichtes

In der Abbildung 4.8 wird die Ausgabe nach dem Ausführen der Erzeugungs-Beispiels angezeigt. Hier sieht man den Preisunterschied, sowie die verschiedenen Beschreibungen der Döner-Instanzen.

```
Mein Doener:
 Beschreibung: Brottasche, Gefluegelfleisch, Schafskaese
 Preis: 3,50 Euro

Anderer Doener:
 Beschreibung: Dueruem, Gefluegelfleisch
 Preis: 3,00 Euro
```

Abb. 4.8: Ausgabe von Erzeugung und Verwendung eines Döner-Gerichtes

4.3.2 Konkrete Beispiele

Im Folgenden werden ein paar populäre Anwendungen des Decorator-Patterns vorgestellt.

Wer in Java schon einmal mit Dateien gearbeitet hat, ist mit dem Stream-Konzept vertraut. So wird beispielsweise wie in Abbildung 13.8 auf eine komprimierte Datei gepuffert zugegriffen. Der *FileStream* ist die konkrete Komponente und *BufferedStream* und *CompressingStream* stellen die Dekorierer dar. Tauscht man *FileStream* durch das Stream-Objekt eines Web-Requests aus, kann man problemlos auf die komprimierte Datei auf einem Webserver gehostet zugreifen. Das Stream-Konzept für die Ein- und Ausgabe ist nicht nur in Java zu finden, sondern beispielsweise auch in anderen Sprachen wie C#.

```
1 Stream s = new CompressingStream(
2                new BufferedStream(
3                    new FileStream(filename)));
```

Abb. 4.9: Stream-Konzept in Java verwendet das Decorator-Pattern

Nicht nur die Streams bauen auf das Decorator-Prinzip auf. Im Hintergrund von Collections in C# findet das Decorator-Prinzip ebenfalls Anwendung. Hier kann man beispielsweise eine ArrayList mit Inhalten erzeugen. Ruft man danach die *ArrayList.ReadOnly(myList)*-Methode auf und übergibt diese gerade erzeugte Liste, so erhält man eine Liste zurück, die nicht mehr veränderbar ist. Intern wird um die eigentliche Liste wieder ein Dekorierer gehüllt, der jeden schreibenden Zugriff ignoriert. Auf diese Weise könnte auch eine threadsichere Liste aus unserer vorherigen Liste erzeugt werden.

Neben diesen beiden Beispielen kann man zusätzlich noch viele Frameworks für objektorientierte grafische Benutzeroberflächen nennen. Hier gibt es Basiskomponenten, wie beispielsweise ein Textfeld. Dieses kann mit Scrollbalken und einem sichtbaren Rand dekoriert.

Die Beispiele aus der realen Welt zeigen, dass fast jeder Programmierer das Decorator-Pattern bereits angewendet hat, ohne überhaupt zu wissen, dass es sich hierbei um ein Pattern handelt.

4.4 Konsequenzen der Verwendung des Decorator Patterns

Im folgenden Kapitel sollen die Konsequenzen der Anwendung des Decorator Patterns disktuiert werden. Diese erstrecken sich vom Entwurf einer Software über ihren Aufbau bis hin zum Testen und dem Vergleich von Referenzen auf Decorator-Objekte.

4.4.1 Entwurf der Klassenstruktur

Dieses Kapitel beschreibt, was beim Entwurf der Klassen nach dem Decorator-Pattern beachtet werden muss und wann es überhaupt sinnvoll ist, dieses Pattern zu verwenden.

Als Strukturmuster [Eil10] hat das Decorator Pattern bereits Einfluss auf den Entwurf der neuen Sofwarekomponente. Wie bereits beschrieben, muss mindestens eine *Komponente* identifiziert werden und dazu verschiedene *Dekorierer*. Das mag gerade bei einer großen Menge von Dekorieren (z.B. Zutaten) viele, kleine, ähnliche Klassen ergeben. Aber denken wir einmal über eine alternative Implementierung nach. Sollen Dekorierer als Attribute abgebildet werden? Dann könnte man jeweils nur einen Wert (z.B. Preis) abbilden. Sollen die angebotenen Zutatenkombination als eigene Klassen abgebildet werden (z.B. Klasse „DönerMitGeflügelfleischUndSchafskäse“[Fre04, Seite 81])? Dann hätte man für alle Kombinationen noch viel mehr Klassen und würde die Flexibilität der Kombinationsmöglichkeiten verlieren. Kaum eine der Alternativen passt wirklich auf die Bedürfnisse der Imbissbude.

4.4.2 Abhängigkeiten der Decorator-Klassen / -Objekte

Im Folgenden werden die Abhängigkeiten (bzw. Unabhängigkeiten) der Objekte des Decorator-Patterns beschrieben und ihre Auswirkung auf Implementierung, Änderung und Wartbarkeit bzw. Testbarkeit erläutert.

Der größte Vorteil des Decorator-Pattern ist die Unabhängigkeit zwischen den einzelnen Decorator-Klassen beziehungsweise den Objekten. Eine Komponente muss nicht wissen ob und wie viele Dekorierer sie hat. Das Gleiche gilt für die Dekorierer selbst. Die einzige Verbindung wird durch das gemeinsame Interface festgelegt.

Sobald die Spezifikation der verschiedenen Klassen und das gemeinsame Interface vorliegt, können die einzelnen Klassen von verschiedenen Personen entwickelt werden. Jeder Entwickler kann eine oder mehrere Klassen implementieren, ohne dass die Entwickler sich gegenseitig in die Quere kommen oder an einer gemeinsamen Komponente arbeiten müssen (Probleme mit dem Zusammenführen unterschiedlicher Code-Dokumente kennt sicherlich jeder). Dadurch kann ein Modul, das auf dem Decorator-Pattern basiert, parallel, durch viele Entwickler, implementiert werden.

Einmal erstellt, können die Klassen einfach ausgetauscht oder geändert werden, ohne sich gegenseitig zu beeinflussen. Bei der Imbissbude erfordert die neue Zutat „Spezialsauce“ nur eine neue Klasse, die diese Zutat beschreibt. Die Klasse „Dürüm“ oder die Klassen der anderen Zutaten müssen dazu nicht geändert werden. Das Pattern verfolgt damit konsequent das Open-Closed Prinzip: Es ist offen für Erweiterung, aber geschlossen für Änderungen[Fre04, Seite 105].

Das Konzept der unabhängigen Klassen erhöht außerdem die Wartbarkeit. Es können gezielt einzelne Klassen getestet werden. So können Fehler schneller gefunden und behoben werden. Im Vergleich zu einer Singelton-Klasse kann man hier in einer völlig seperaten Testumgebung arbeiten.

4.4.3 Vergleich von Decorator-Objekten

Um für die Imbissbude Werbung zu machen, startet der Betreiber eine Werbeaktion: Es werden Rabattgutscheine verteilt, mit denen man einen Dürüm kostenlos bekommt. Da diese Aktion nur eine Woche gilt, soll der Dönerpreis vor der Abrechnung auf „0“ reduziert werden. Der Grundpreis bleibt dadurch unverändert. Jedes Objekt wird vor der Bestellung daraufhin überprüft, ob es sich um einen Dürüm handelt, um den Rabatt abzuziehen. Man kann aber nicht mit „<Objekt> instanceof Dueruem“ den Typ des Objektes abfragen, denn bei Verwendung des Decorator Patterns sind die Komponente („Dürüm“) und die dekorierte Komponente („Dürüm mit Schweinefleisch“) nicht vom selben Typ [Gam94, Seite 196].

Bei einem solchen Vergleich muss man selbst Kriterien definieren, die bei beiden Objekten gleich sein müssen, um die Objekte als „gleich“ zu bezeichnen. Es könnte der Typ des Gerichtes („Dürüm“ oder „Brottasche“), der als String gespeichert wird, als Kriterium genügen oder eine bestimmte Kombination von Zutaten („Dürüm mit Hähnchenfleisch und Salat“), welches über die Beschreibung verglichen wird. Welche Kriterien verwendet werden, muss für jeden Fall individuell entschieden werden.

Wenn für einen solchen Vergleich nachträglich alle Zutaten und Komponenten mit einem Vergleichskriterium augestattet werden müssen, erhöhen sich die Kosten für die Werbekampagne der Imbissbude um die Stunden des Programmierers. Am Besten werden beim

Entwurf nach dem Decorator-Pattern schon ein oder mehrere Vergleichskriterien definiert, dann lohnt sich auch die Werbekampagne.

4.4.4 Abstrakte Objekte sollen nicht zerlegt werden

Benutzer von Decorator-Objekten wissen nicht, ob es sich um eine dekorierte oder eine undekorierte Komponente handelt[Gam94, Seite 194]. Diese Tatsache macht zwar die Nutzung der Objekte einheitlich („Döner mit Geflügelfleisch“ wird wie „Döner mit Rindfleisch“ abgerechnet), eine Identifikation der einzelnen Dekorierer jedoch schwerer.

Bestellt der Kunde einen Dürüm mit Geflügelfleisch und Schafskäse und merkt anschließend, dass er doch kein Geflügelfleisch möchte, wird es kompliziert, diese Bestellung nachträglich zu editieren. Die Komponente „Dürüm“ wird mit „Geflügelfleisch“ und „Schafskäse“ dekoriert. Von nun an wird also nur noch mit dem Verweis auf „Schafskäse“ gearbeitet. Über dieses Objekt auf die Instanz von „Geflügelfleisch“ zu kommen, ist im Decorator-Pattern nicht vorgesehen. Eine solche Anforderung ist ein K.O. Kriterium für das Decorator Pattern, da dieses gerade versucht ein solches Verhalten zu verhindern.

Ein ähnliches Problem des Decorator-Patterns ist die Reihenfolge der Schachtelung der Objekte. Wenn nicht durch eine andere Komponente eine sinnvolle Reihenfolge der Dekorierer festgelegt wird, kann diese nach Belieben erfolgen. Dies mag bei manchen Anwendungsfällen gewünscht sein, aber es gibt auch Anwendungsfälle, bei denen die Reihenfolge wichtig ist und möglicherweise auch verändert werden muss. Wird beispielsweise nach dem Rabatt noch ein weiterer Dekorierer (z.B. Schafskäse) hinzugefügt, wird damit der Rabatt wieder zunichte gemacht (Der Dürüm kostet nun 0,50€ anstatt 0,00€). Allerdings ist das Sicherstellen der Reihenfolge nicht die Aufgabe des Dekorierers. Bei einer solchen Anforderung kann man auf andere Patterns, wie die Fabrik zurückgreifen.

4.5 Abgrenzung zu anderen Design Patterns

4.5.1 Proxy Pattern

Das Proxy Pattern gehört ebenso wie das Decorator Pattern in die Kategorie der Strukturmuster. Bei beiden wird mit abstrakten Objekten gearbeitet, die das eigentliche Objekt verdecken. Das Proxy Pattern wird allerdings nur dann verwendet, wenn das eigentlich Objekt in irgendeinerweise geschützt werden soll. Dies ist der Fall, bei Zugriffen auf sensible Ressourcen, oder bei Aufwändigen Operationen durch das eigentliche Objekt, die verzögert werden sollen. Das Decorator Patter dient hingegen der Erweiterung des Funktionsumfangs des eigentlichen Objektes. Außerdem gibt es beim Decorator Patter auch mehrere Decorator (beim Proxy nur eines).

4.5.2 Adapter Pattern

Auch das Adapter Pattern „verpackt“ ein Objekt. Allerdings ist auch hier die Aufgabe eine andere wie beim Decorator. Es soll kein Funktionsumfang erweitert werden, sondern eine Schnittstelle zwischen zwei Komponenten definiert werden, die den Austausch einer der

beiden Komponenten vereinfachen soll. Wir erinnern uns: Decorator fügt Funktionen zu dem verpackten Objekt hinzu.

4.5.3 Komposite Pattern

Ein weiteres Pattern der Stukturmuster ist das Komposite Pattern. Es soll Unterschiede zwischen einzelnen Klassen verbergen. So können beispielsweise in einem Baum Blätter und Knoten gleich behandelt werden, wenn sie nach Komposit-Diagramm entworfen sind. Die Eigenschaft, mehrere Klassen gleich zu behandeln hat es mit dem Decorator gemeinsam, allerdings bleibt die Aufgabe des Decorator Patterns, neben der Gleichbehandlung der Klassen, noch die Funktionserweiterung der darunterliegenden Klasse

4.5.4 Strategy Pattern

Der letzte Vergleich wird mit einem Pattern aus der Kategorie der Verhaltensmuster durchgeführt, dem Strategy Pattern. Beim Strategy Pattern werden unterschiedliche Algorithmen jeweils in eine Klasse gepackt, um sie so austauschbar zu machen. Der Unterschied wird hier schon deutlich, es wird ausgetauscht, nicht erweitert (wie beim Decorator Pattern).

4.6 Zusammenfassung

Das Decorator Pattern bietet optimale Möglichkeit Komponenten mit zusätzlichen Funktionen durch Dekorierer zu erweitern. Dabei bleibt es egal, ob mit der orginal Komponente gearbeitet oder mit einer dekorierten. Es eignet sich auch wunderbar um bestehende Frameworks zu erweitern. Populäre Beispiel sind die Java I/O Klassen, die fast ausschließlich auf diesem Pattern aufbauen.

Das Decorator Pattern eignet sich aber nicht, um nach der Erstellung eines Dekorierers auf dessen Vorgänger zuzugreifen oder gar die Verschachtelung der Dekorierer zu ändern.

5 Adapter

Definition Ein Adapter übersetzt ein Interface einer Klasse in ein von dem Client erwartetes Interface. Der Adapter erlaubt es Klassen mit einander zu arbeiten, die wegen Ihrer inkompatiblen Schnittstellen dazu nicht in der Lage wären. [Gam94]

5.1 Absicht

Es kommt häufig vor, dass eine Klasse nicht eingesetzt werden kann, weil diese ein bestimmtes Interface nicht unterstützt. Die Klasse so umzuschreiben, dass diese verwenden werden kann erfordert zu viel Zeit. Hier kommt der Adapter ins Spiel. Damit wird eine Verbindung zwischen der zu adaptierten Klasse und eigener Schnittstelle geschaffen.

Der Adapter wird auch dann eingesetzt, wenn die zu benutzende Klasse aus einer fremden Bibliothek stammt bzw. nicht verändert werde darf.

5.2 Struktur

Es wird zwischen zwei Ausprägungen eines Adapters unterschieden. Zum einen gibt es den Klassenadapter und zum anderen den Objektadapter. Der Klassenadapter ist in Java nicht implementierbar, denn dieser basiert auf einer Mehrfachvererbung. In den Abbildungen 5.1 und 5.2 sind die UML-Diagramme des Klassenadapters und der Objektadapters dargestellt.

In den Begrifflichkeit wird zwischen Adaptee, Adapter und Target unterschieden. Diese sind wie folgt Definiert:

Adaptee Die Klasse die angepasst werden muss, damit es mit dem Target kompatibel ist.

Adapter Der Adapter erledigt die eigentlich Arbeit und passt den Adaptee an den Target an.

Target Der Target ist das Ziel der Adaption.

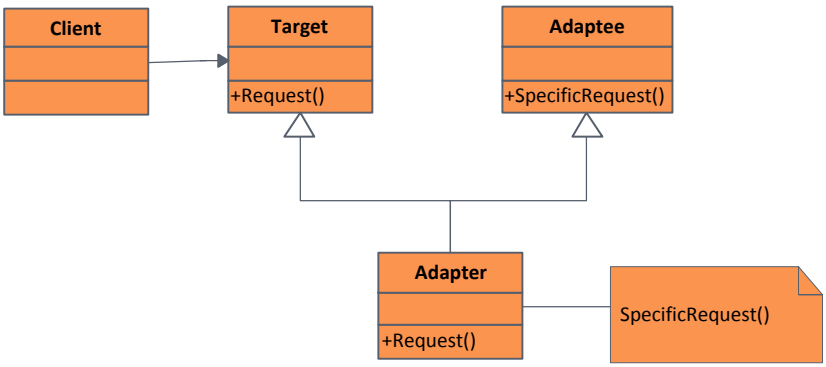


Abb. 5.1: Die UML-Darstellung des Klassenadapters

Wie zu sehen ist verwendet der Klassenadapter Mehrfachvererbung und kann die Methode des Adaptee *SpecificRequest()* überschreiben.

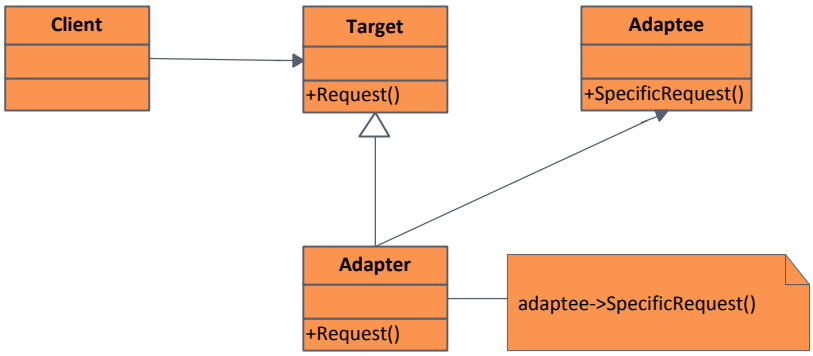


Abb. 5.2: Der Objektadapter

Der Objektadapter hält eine Referenz auf den Adaptee und ruft die nötige Methode *SpecificRequest() auf.*

5.3 Beispiel

5.3.1 Szenario

Es gibt ein typisches Szenario beim Adapter. Die GUI wurde geändert. Es kommt häufig vor dass die Oberfläche sich so sehr verändert hat, dass die alten Datensätze nicht direkt angezeigt werden können. Wenn die Daten aus der Datenbank kommen wird keiner die Methoden zum auslesen erneut schreiben. Die Daten müssen neu eingeordnet und neu strukturiert werden.

In unserem Fall liegen die wichtigsten Daten in einer Subklasse dem *AdapteeSub*. Diese müssen in den Target direkt geschrieben werden damit die *zeige()* Methode diese auch anzeigen kann.

5.3.2 Implementierung

Im Listing 5.3.2 ist gezeigt wie dieser anzuwende ist. Zuerst werden Listen erzeugt die die Daten halten sollen. Exemplarisch wird dann ein echter Target und ein anzupassender Adaptee erzeugt. Der Adapter wandelt den Adaptee um und bereitet diesen für die Ausgabe vor. Im letzten Schritt wird der Traget neben dem Adaptee zum Vergleich ausgegeben.

```
1   public static void main(String[] args)
2   {
3           //Erzeuge Vectoren
4           Vector<Adaptee> adapteeVector = new Vector<Adaptee>();
5           Vector<Target> vector = new Vector<Target>();
6
7           //Erzeuge Examples
8           Target exampleTarget = new Target("Target Jobs", "CEO");
9
10          AdapteeSub exampleAdapteeSub = new AdapteeSub("Adaptee", "Jobs");
11          Adaptee exampleAdaptee = new Adaptee(0, exampleAdapteeSub);
12
13          //Fuege Examples hinzu
14          vector.add(exampleTarget);
15          adapteeVector.add(exampleAdaptee);
16
17          //Adaptiere
18          for (Adaptee adaptee : adapteeVector) {
19                  Adapter adapter = new Adapter(adaptee);
20                  vector.add(adapter);
21          }
22
23          //Ausgabe
24          for (Target target : vector) {
25                  target.zeige();
26          }
27  }
```

Abb. 5.3: Anwendung eines Adapters im Client

Im nächsten Listing wird gezeigt wie der Adapter arbeitet. Weil die Daten etwas anders gespeichert werden, wird ein kleiner Konverter benötigt. Da die Klasse Adapter vom Target abgeleitet ist, können die Daten direkt mit super in den Target geschrieben werden.

```
1  //Konstruktor des Adapters
2  public Adapter(Adaptee adaptee) {
3                  super.setTargetName(adaptee.getSub().getFirstName() + " " +
   adaptee.getSub().getLastName() );
4                  super.setTargetID(9999);
5                  super.setTargetPosition(convert(adaptee.getRank()));
6          }
7  //Kleiner Konverter da Adaptee die Position im Unternehmen
8  //anders abspeichert
9          private String convert(int rank) {
10                 if(rank == 0)
11                         return "CEO";
12                 else
13                         return "Hausmeister";
14         }
```

Abb. 5.4: Der Kern des Adapters

5.4 Bewertung

Vorteile:

Klassenadapter Es sind keine weiteren Verweise notwendig, da der Adapter von Adaptee und vom Target erbt. So muss nur die eine wichtige Methode im Adapter überschrieben werden und schon sind wir fertig. So wird ein einfacher Zugriff auf beide Mitspieler gegeben.

Objektadapter Erlaubt einem Adapter mit vielen Adaptees zu arbeiten. Der Zugriff auf die Unterklassen des Adaptees ist frei und so ist es ein leichtes neue Funktionalitäten hinzuzufügen.

Nachteile:

Klassenadapter Da er direkt von der zu adaptierenden Klasse, erbt ist es nicht möglich die Unterklassen des Adapatee mit zu adaptieren.

Objektadapter Es ist etwas komplizierter mit den Subklassen des Adaptees umzugehen. Außerdem kann das Verhalten des Adaptee im Gegensatz zu Klassenadapter nicht direkt verändert werden. Dazu muss eine Subklasse des Adaptees geschrieben werden.

Abgrenzung Das Entwurfsmuster Brücke hat eine ähnliche Struktur wie das Adapter Pattern, hat aber eine andere Aufgabe. Bei der Brücke wird versucht das Interface von der Implementierung zu trennen.

6 Fassade

Definition Eine Fassade bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Eine Fassade definiert eine höher liegende Schnittstelle, die die Benutzung des Subsystems vereinfacht. [Gam94]

6.1 Absicht

Bei diesem Entwurfsmuster dreht sich alles um die Reduzierung der Komplexität und somit auch um die Erhöhung der Wartbarkeit des Systems.

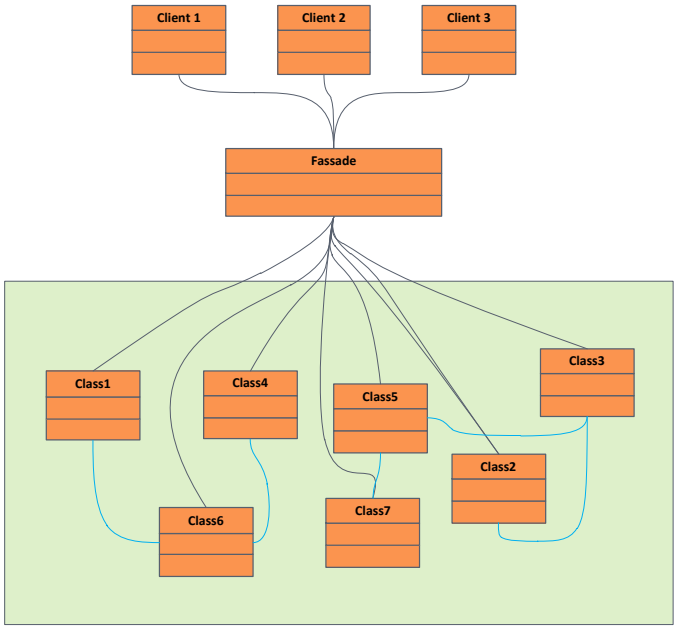
Durch den Einsatz dieses Musters werden die Änderungen im Subsystem nicht zum Client weitergereicht, sondern werden von der Fassade abgefangen. Oft reicht es aus die Fassade anzupassen und somit sind die Clients nicht anzufassen.

Sollte der Programmierer jedoch jemals Zugriff auf das Subsystem brauchen, so kann er die Fassade ignorieren und direkt auf die gewünschte Klasse zugreifen. Vorausgesetzt das Subsystem ist öffentlich. Bei diesem Muster geht es nicht um information hiding, sonder wirklich nur um eine einfache Handhabung des Subsystems.

6.2 Struktur

Die Struktur ist sehr einfach und ist in der Abbildung 6.1 dargestellt. Die Fassade steht im Mittelpunkt zwischen Client und Subsystem und fungiert als Vermittler.[Gam94](Seite 185)

Man könnte sich die Fassade als die freundlichen Empfangsdame in einer Firma vorstellen. Die Kunden sollten zuerst zu der Empfangsdame hin, die die Kunden dann weiter leitet. Ein Stammkunde könnten auch einfach vorbeigehen, falls er sich auskennt. So ist es auch in der Programmierung, wenn eine bestimmte Funktionalität des Subsystems benötigt wird, ist ein direkter Zugriff auf diese jederzeit möglich. Schöner wäre es aber wenn die Fassade benutzt wird.



s

Abb. 6.1: Aufbau des Entwurfsmusters Fassade

An ein Subsystem können mehrere Fassaden gekoppelt werden. Es stehen zwei Möglichkeiten der Realisierung zu Verfügung. Mit mindestens einer abstrakten Fassade können mehreren konkreten für die Bedürfnisse angepassten Fassaden erstellt werden. So können Fassaden entstehen die unterschiedliche Aufgaben erledigen aber alle dennoch an ein Subsystem gekoppelt sind. Der andere Weg ist der Einsatz von mehren Fassaden die jeweils ein Teil des Subsystems kontrollieren. Dadurch wird die Fassade nicht unnötig groß und lässt sich gut warten.

Fassade

- Es ist extrem wichtig, dass die Fassade keine Funktionalitäten besitzt. Diese sollten alle, egal wie klein, vom Subsystem erledigt werden.
- Die internen Beziehungen des Subsystems interessieren die Fassade nicht. Dies verringert die Komplexität der Fassade und erleichtert somit die Benutzung dieser.
- Die Fassade kennt das Subsystem so weit wie es notwendig ist und weiß welche Klasse welche Funktion erfüllt. Erst dadurch kann eine gezielte Weiterleitung der Aufgaben erreicht werden.

Subsystem

- Das Subsystem hat keine Ahnung von der Fassade und hält somit auch kein Referenz dieser. Das verhindert zum einen unbeabsichtigte ärgerliche Rekursionsaufrufe und die Klassen im Subsystem kennen nur für Sie wichtigen Klassen.

- Das Subsystem erfüllt nur die von der Fassade gegebene Aufgabe, mehr nicht.

In der Abbildung 6.2 ist dargestellt was passieren würde, wenn die Fassade nicht benutzt wird. Kurz gesagt Chaos! Die Übersicht ist schon längst verloren gegangen. Da spielt es keine Rolle wie gut das Diagramm ist und wie bunt es gestaltet wurde.

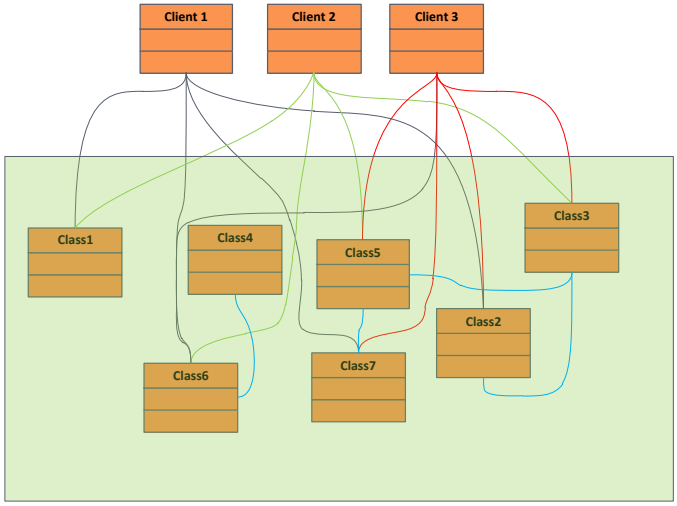


Abb. 6.2: Abbildung des Systems ohne die Fassade

6.3 Beispiele

6.3.1 Szenario

Das Szenario stammt aus dem Buch [Gam94] und wird hier kurz erläutert. Unser Subsystem ist ein Compiler. Dieser besteht aus Code-Generator, Parser, Scanner, ProgramNodeBuilder und den dazugehörigen Nodes. Von dem Code-Generator gibt es unterschiedlichen Ausprägungen, abhängig von der Zielmaschine. In der Abbildung 6.3 ist diese Beispiel als UML-Klassendiagramm dargestellt. Das Subsystem besteht aus vorhin aufgelisteten Klassen und der Compiler fungiert hier als Fassade

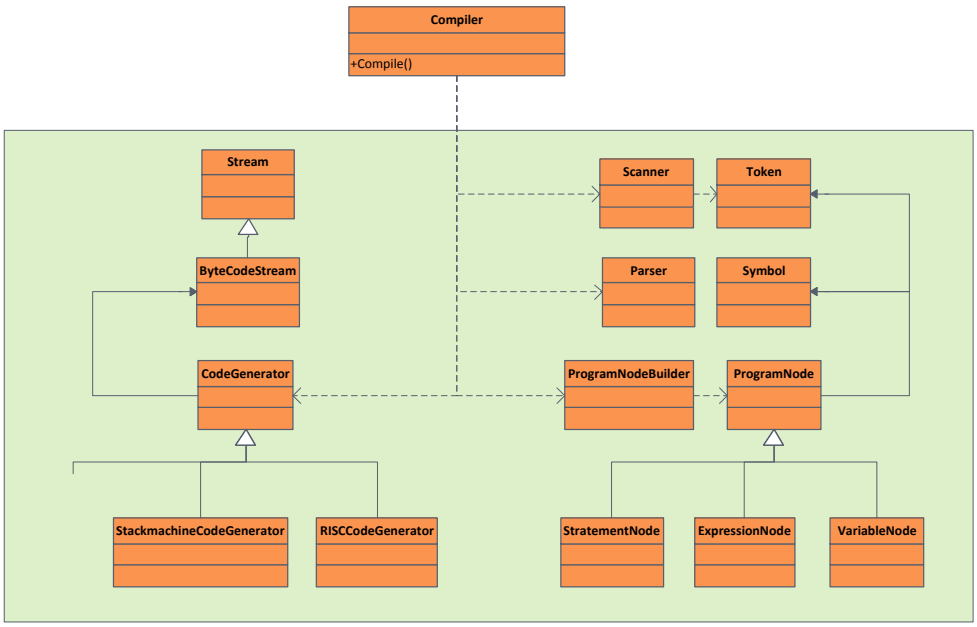


Abb. 6.3: Ein UML-Diagramm eines vereinfachten Compilers

6.3.2 Implementierung

Fassade wird häufig in Verbindung mit anderen Patterns benutzt. Als üblicher Partner zählt das Singelton Pattern.

Schauen wir uns zuerst die Fassade an. Abhängig von der nötigen Abstraktionstiefe kann diese viele oder wenige *public* Methoden besitzen. Hier wurden beide Möglichkeiten implementiert. Im Listing 6.3.2 wird die Fassade mit vielen Methoden gezeigt und im Listing **??** die kurze Variante. Jeder muss selbst entscheiden wie abstract er seine Fassade implementiert aber **je abstrakter desto besser**.

```
1  public static void main(String[] args) {
2
3                  IFassade fassade = Fassade.getInstance();
4
5                  fassade.setFile(args[0]);
6                  fassade.scanFile();
7                  fassade.parseFile();
8                  fassade.buildProgramNodes();
9                  fassade.generateCode(IFassade.STACKMACHINE);
10                 fassade.writeOut(args[1]);
11         }
```

Abb. 6.4: Unschöne anwendung einer Fassade. Hier wird die Funktionalität nicht verborgen

Durch die Benutzung von dem Singelton Pattern *Fassade.getInstance* wird sichergestellt, dass es nur eine Fassade gibt. Dadurch werden nicht unnötig viele Klassen instantiiert.

```
1         public static void main(String[] args) {
2
3                 IFassade fassade = Fassade.getIntance();
4                 fassade.compile(args, IFassade.RISCMACHINE);
5         }
```

Abb. 6.5: Aufruf der compile Methode der Klasse Fassade

Wie erreicht man diese hohe Abstraktion der Fassade? Die Methoden aus dem Listing 6.3.2 werden genomen undin eine neue Methode der Fassade gesteckst. Dies ist im Listing 6.3.2 demonstriert.

```
1 public void compile(String[] parameter, int type) {
2                 setFile(parameter[0]);
3                 scanFile();
4                 parseFile();
5                 buildProgramNodes();
6                 generateCode(type);
7                 writeOut(parameter[1]);
8         }
```

Abb. 6.6: Implementierung der Methoden compile

Nachdem wir wissen wie eine Fassade auszusehen hat, sollten wir die aufgerufenen Methoden genauer betrachten. Exemplarisch werden im folgenden einige Methoden gezeigt.

```
1  //Subklassen die von der Fassade benutzt werden
2          private CodeGenerator generator;
3          private ProgramNodeBuilder nodeBuilder;
4          private Scanner scanner;
5          private Parser parser;
6
7  //Einige Methoden aus der Fassaden Klassen
8          public void parseFile() {
9                  Parser parser = new Parser();
10                 parser.Parse(file);
11         }
12
13         public void generateCode(final int machine) {
14                 switch (machine) {
15                 case STACKMACHINE:
16                         generator = new StackMachineCodeGenerator();
17                         break;
18                 case CISCMACHINE:
19                         generator = new CISCCodeGenerator();
20                         break;
21                 case RISCMACHINE:
22                         generator = new RISCCodeGenerator();
23                         break;
24                 default:
25                         generator = new CISCCodeGenerator();
26                         break;
27                 }
28                 generator.generate();
29         }
```

Abb. 6.7: Einige Methodes in der Klasse Fassade

Im Beispiel 6.3.2 wurden in den Methoden jeweils die notwendigen Subklassen instantiiert. Dies dient hier nur zur Erhöhung der Verständlichkeit. Die Instanziierung sollte im Konstruktor der Fassade geschehen. Hier wird unnötigerweise eine Klasse neu gebaut.

Die Methode *generateCode()* kennt alle Zielmaschinen und lässt abhängig von der Auswahl des Benutzers den Code generieren. Eine Alternative wäre für jeden Code-Generator eine eigene Methode zu schreiben. Dies würde aber unnötigerweise die Anzahl der Methoden in der Fassade erhöhen. Damit wäre das Ziel der Fassade verfehlt. Je einfacher die Benutzung der Fassade und somit des Subsystems ist, desto besser ist es für den Client.

6.4 Bewertung

Vorteile

- Bei Änderung der Funktion muss nur die Fassade umgeschrieben werden. Der Client Code muss nicht angefasst werden und somit braucht der Client von der Veränderung nichts zu erfahren. Die Schnittstelle bleibt ja gleich.
- Die Benutzung der Fassade verursacht weniger Fehler, weil der Programmierer nicht die Funktionen des Subsystems benutzen muss. Er muss sich mit dem System auch nicht auseinander setzen, was Zeit spart.
- In der Fassade könnte Fehler behandelt werden bzw. es ist dort erforderlich. Der Client kennt das Subsystem nicht und kann daher keine Fehler behandeln. Er weiß nicht mal welche. (Dies kann auch als Nachteil angesehen werden)

- Die Anzahl der zu Klassen die von Client genutzt werden sinkt signifikant. Dies erleichtert die Wartung und Lesbarkeit vom Code.
- Der Client kann falls notwendig das Subsystem benutzen und die Fassade übergehen. Dies gilt nur falls das Subsystem öffentlich ist.

Nachteile

- Kann sehr schnell sehr komplex werden, da die Fassade viele Funktionen auf einmal implementieren muss.
- Möglicherweise muss für jeden Client eine eigene Fassade geschrieben werden.
- Das Vertrauen in die Fassade ist notwendig. Es muss sichergestellt werden dass die erwartete Funktion auch erfüllt wird.
- Bei zu vielen Fassaden kann eine kleine Änderung viele große Änderungen in den Fassaden zu Folge haben.

Abgrenzung Ein sehr ähnliches Pattern ist der sog. Vermittler. Dieser hat aber eine andere Aufgabe als die Fassade. Beim Vermittler ist für die Kommunikation zwischen Objekten verantwortlich. Im Gegensatz zum Vermittler kennt das Subsystem die Fassade nicht und es kommt keine neue Funktionalität hinzu.

7 Model-View-Controller

Ziel des Model-View-Controller (MVC) Musters ist es, die Datenstrukturen und Logik einer Anwendung von ihrer Repräsentation in der Benutzerschnittstelle zu trennen [Gam94, Pos. 404], [Kü11, S. 89].

7.1 Absicht

Bei Anwendungen die eine Benutzerschnittstelle (User Interface, UI) besitzen, dienen große Teile des Programmcodes der Verarbeitung von Benutzeraktionen sowie der Steuerung der Darstellung. Da die Eingabemethoden (z.B. Maus, Tastatur, Touchscreens, etc.) sowie die Elemente der Ausgabe (graphische oder textbasierte Elemente, Sprachausgabe usw.) vom jeweiligen Betriebssystem/UI-Framework zur Verfügung gestellt werden, hängen diese Codeteile vollständig von der Plattform ab.
Anders sieht es mit den aus Analyse und Design gewonnen Objekten, ihren Beziehungen untereinander und ihrem Verhalten aus. Diese domänenspezifischen Codeteile sollten unabhängig von Ein- und Ausgabemethoden wiederverwendbar sein. Hinzu kommt, dass sich die Anforderungen an die UI (z.B. der Einsatz auf Mobilgeräten) unabhängig von den Anforderungen an die abgebildeten Geschäftsprozesse (und umgekehrt) ändern.
Die Familie der MVC-Muster bietet Strukturen, mit denen sich die Trennung von Daten und Gechäftslogik auf der einen, sowie der Benutzerschnittstelle auf der anderen Seite verwirklichen lässt. Dies geschieht durch Kombination verschiedener grundlegender Entwurfsmuster, die in den vorangegangenen Kapiteln bereits vorgestellt wurden.

7.2 Struktur

7.2.1 Allgemeine Sicht

Das Model-View-Controller-Pattern teilt sich, wie der Name bereits vermuten lässt, in drei Teile, die softwarearchitektonisch klar voneinander getrennt sein müssen.

View Im View befinden sich alle Code-Elemente, die unmittelbar mit dem Erzeugen einer Benutzeroberfläche, also der Schnittstelle zwischen Benutzer und Programm, zu tun haben. Im Normalfall handelt es sich hierbei um eine GUI mit Buttons, Textfeldern usw. Allerdings wird hier nicht deren Funktionalität implementiert, sondern ausschließlich die Eingaben des Benutzers entgegengenommen, beziehungsweise diesem die Änderungen des Models angezeigt. Der View ist also die Präsentation des Models nach außen hin. Den Status des Models erfährt der View über ein Beobachterinterface, das er implementiert.

Controller Der Controller ist dafür verantwortlich, die Eingaben, die der User über den View tätigt, zu interpretieren und die entsprechenden Funktionen im Model auszuführen. Er kann außerdem als Beobachter des Models fungieren, falls dessen Änderungen direkte Auswirkungen auf die Programmsteuerung haben. Des Weiteren kann der Controller etwaige Einschränkungen im View, z. B. die Deaktivierung eines Buttons, durchführen.

Model Das Model enthält die eigentlichen darzustellenden Daten. Es hat dabei keine Kenntnis vom View oder dem Controller. Im Normalfall implementiert es ein Observable-Interface, damit Änderungen direkt an den View oder im Bedarfsfall auch an den Controller weitergegeben werden können.

In Abb. 7.1 ist der Zusammenhang zwischen den einzelnen Komponenten grafisch dargestellt.

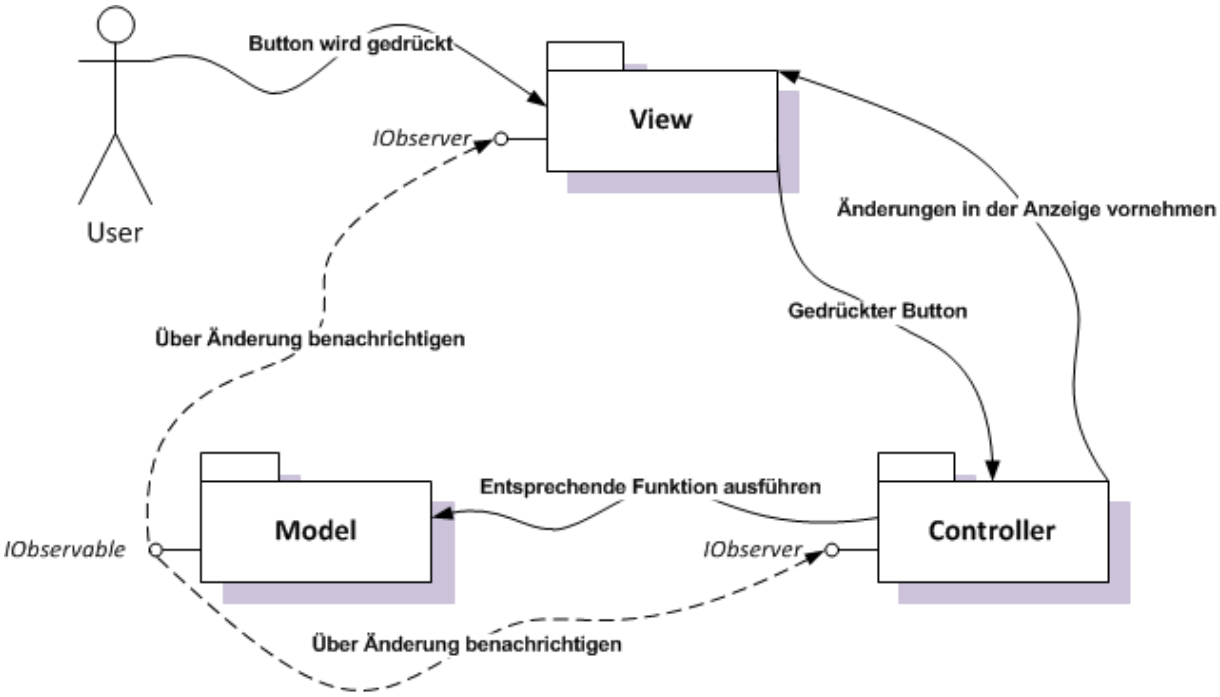


Abb. 7.1: Grafische Darstellung der MVC-Komponenten und des Datenflusses

7.2.2 Konkretes Beispiel

Nun sollen die bisher gewonnenen Erkenntnisse über das MVC-Pattern in einem konkreten Beispiel angewendet werden. Als Szenario soll ein einfaches Programm 'CD-Archiv' dienen, das auf die elementarsten Funktionen 'CD hinzufügen' und 'ausgewählte CD löschen' beschränkt sein soll.
Im Hauptfenster sollen lediglich eine Tabelle mit den gespeicherten CDs und zwei Buttons für das Hinzufügen und Löschen vorhanden sein. Wird der 'Add'-Button gedrückt, soll ein

neues Fenster geöffnet werden, in dem Interpret und Titel der neuen CD in zwei dafür vorgesehene Textfelder eingegeben werden können. Im Folgenden soll nun die Struktur dieses Programms, die natürlich nach MVC zu gestalten ist, näher betrachtet werden.
Zuerst soll die GUI erstellt werden, die natürlich in den View gehört. Auf deren Implementierung soll an dieser Stelle nicht genauer eingegangen werden, sie kann beliebig nach den gemachten Vorgaben erstellt werden. In der vorgestellten Lösung implementiert der View das Interface 'ActionListener' und somit die Methode 'actionPerformed', die die Aufgabe hat, die Quelle eines ausgelösten Events zu identifizieren. Wichtig ist, wie bereits erwähnt, dass hier ausschließlich die Eventquelle identifiziert und die verantwortliche Methode im Controller aufgerufen wird, aber keinerlei Steuerungsfunktionalität enthalten ist.
Als nächstes widmen wir uns dem Model, das im vorliegenden Fall einfach mit einem Vector von CDs und den selbsterklärenden Methoden 'addCD', 'removeCD' und 'getCDs' realisiert werden soll. 'CD' ist hierbei wiederum eine eigene Klasse mit den String-Attributen 'interpret' und 'title' und den dazugehörigen Getter-Methoden.
Jetzt muss das Herzstück des MVC-Patterns, der Controller, erstellt und die einzelnen Komponenten miteinander verknüpft werden. Am einfachsten gelingt dies, wenn wir uns am Ablauf eines vom User durchgeführten Button-Klicks orientieren:
Das ausgelöste Event läuft in der Methode 'actionPerformed' auf und dessen Quelle (der Button) wird identifiziert und an den Controller weitergegeben. Der View besitzt als Attribut also eine Referenz auf den Controller, die am besten im Konstruktor übergeben und gesetzt wird. Zum besseren Verständnis soll an dieser Stelle ein Ausschnitt einer möglichen Implementierung der 'actionPerformed'-Methode gezeigt werden.

```
1 //falls addButton geklickt wurde
2 if (e.getSource() == addButton)
3 {
4         //addButton-Methode im Controller ausfuehren
5         controller.addButton();
6 }
```

Abb. 7.2: Ausschnitt der Methode actionPerformed() im View

Im der Controller-Klasse müssen sich Referenzen auf den View und das Model befinden. In seinem Konstruktor wird die übergebene Modelreferenz gesetzt, ein neuer View erzeugt und -falls vorhanden- eine initiale Methode im View aufgerufen, die die GUI-Elemente erzeugt. Des Weiteren müssen im Controller natürlich noch die Methoden implementiert werden, die der View nach Identifizierung einer Eventquelle aufruft, wie die im Listing gezeigte Methode 'addButton'. In diesen Methoden muss nun der Controller entscheiden, welche weiteren Funktionen im Model oder auch im View auszuführen sind. In unserem Beispiel wird in der genannten Methode die Funktion 'createAddDialog' im View ausgelöst, um das neue Fenster anzuzeigen.
Wenn alle gewünschten Funktionalitäten vom Controller gesteuert werden, muss nur noch eine Main-Klasse mit der gleichnamigen Methode erstellt werden, in der ein neues Model und ein neuer Controller mit dessen Referenz erzeugt werden. Jetzt ist das kleine Programm fast fertig. Eine wichtige Funktionalität wurde jedoch bislang außer Acht gelassen.
Der View, der ja das Model nach außen hin repräsentiert, muss bei dessen Änderungen informiert werden. Dies lässt man am besten durch das bereits bekannte Pattern 'Observer'

passieren. Demnach muss die Model-Klasse ein Observable-Interface mit den entsprechenden Methoden implementieren und der View ein Observer-Interface. Im folgenden Listing wird die Observer-Methode 'cdRemoved' gezeigt, die vom Observable-Interface aufgerufen wird, sobald eine CD gelöscht wurde.

```
1 public void cdRemoved(int cdIndex)
2 {
3         //entsprechende Zeile aus der Tabelle loeschen
4         tableModel.removeRow(cdIndex);
5 }//end method cdRemoved()
```

Abb. 7.3: Ausschnitt der Methode actionPerformed() im View

Meldet man nun noch im Konstruktor des Controllers den View als Observer des Models an, ist unser CD-Archiv lauffähig.
Das Klassendiagramm unserer Beispiellösung findet sich in Abb. 7.4.

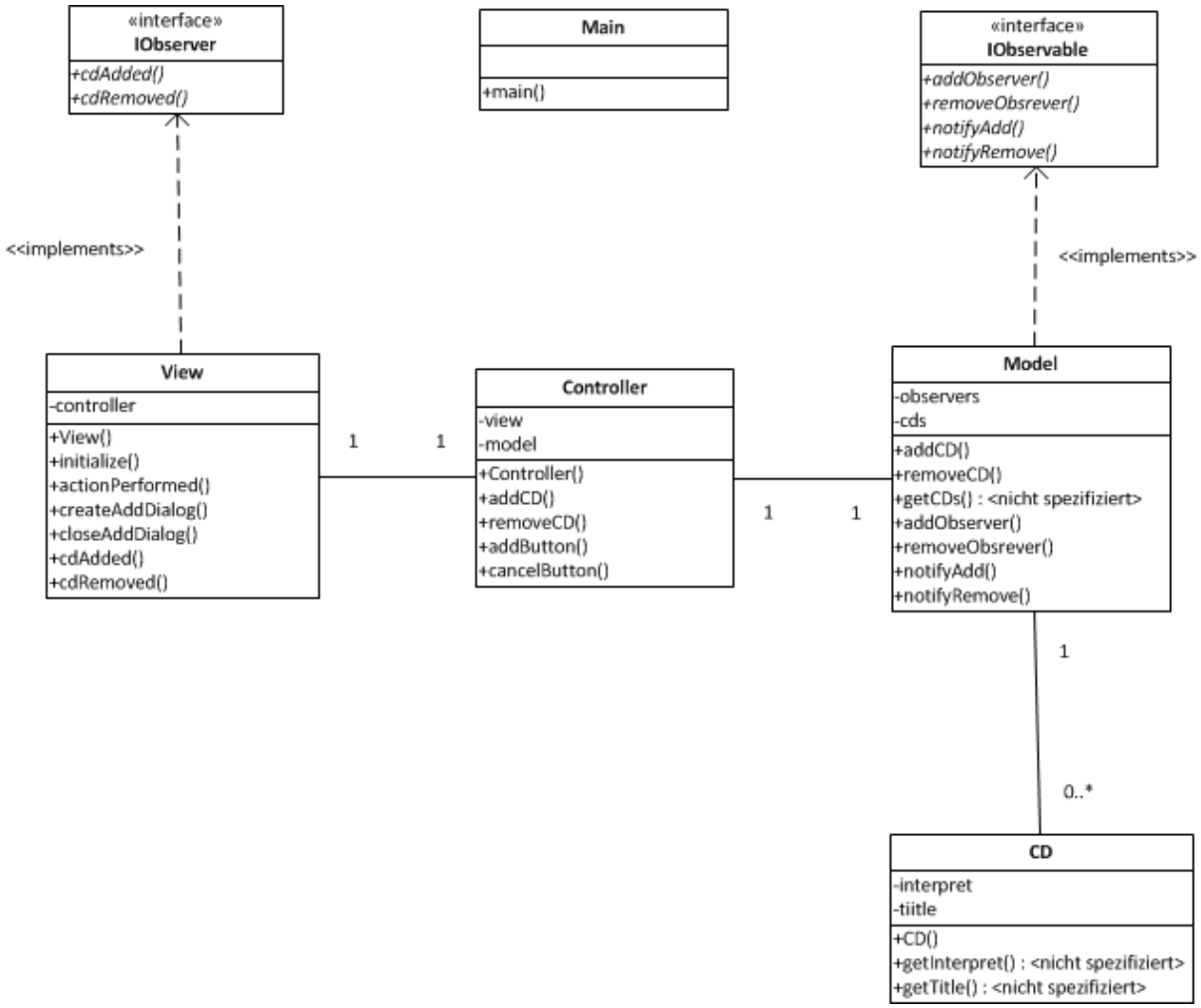


Abb. 7.4: Klassendiagramm eines Beispiel-CD-Archivs

7.3 Verwendete Entwurfsmuster

7.3.1 Composite

Das Composite-Muster beschreibt die Anordnung von Objekten in einer Baumstruktur, um Gruppen dieser Objekte (repräsentiert durch einen Knoten und seine Kindknoten) genauso verarbeiten zu können wie ein einzelnes Objekt (ein Blatt im Baum). Dieses Muster kommt im View zum Einsatz, der sich aus Gruppen von Views und diese wiederum aus einzelnen Views zusammensetzen [Gam94, Pos. 416, 3051]. So ist ein komplexer View, wie Beispielsweise ein Fenster, aus Subviews wie Menüleisten, Toolbars oder Tabellen zusammengesetzt. Diese setzen sich wiederum aus Menüs, Buttons, Spalten etc. zusammen. Dabei erben häufig alle Elemente von einem Grundlegenden View-Element (z.B. Java Swing: JComponent oder Cocoa Touch: UIControl). Der Vorteil dieser Hierarchie im MVC-Muster ist, dass ein Controller die Eingabeverarbeitung für bestimmte Teile eines Views, oder in einfachen Fällen den ganzen View übernehmen kann.

7.3.2 Observer

Das Observer-Muster modelliert eine 1:n Beziehung zwischen zwei Typen, über die Nachrichten ausgetauscht werden können. Es kommt als Teil des MVC-Musters zum Einsatz, um die Daten eines Models in verschiedenen Views darstellen zu können. Im oben gezeigten Beispiel ist es denkbar, dass der Inhalt der Tabelle (eine Liste von CDs) als Diagramm dargestellt wird, in dem der Anteil der jeweiligen Interpreten am CD-Archiv veranschaulicht wird. Je nach MVC-Variante wird das Observer-Muster entweder zwischen View und Model oder zwischen Model und Controller eingesetzt [Gam94, Pos. 416].

7.3.3 Weitere Muster

Neben den beiden grundlegenden Mustern Composite und Observer, wird MVC in vielen Quellen mit weiteren Mustern beschrieben. So wird der Controller aus Sicht des Views oft als Strategy-Implementierung der Eingabeverarbeitung [Kü11, S. 93] [Gam94, Pos. 428] [App11, S. 163], das Modifizieren von Views mit dem Decorator-Muster [Gam94, Pos. 428], und die Rolle des Controllers als Mittler zwischen View und Model als Mediator [App11, S. 164] beschrieben. Diese bei weitem nicht vollständige Aufzählung zeigt schon, dass MVC nicht als ein exakt beschreibbares Muster aufgefasst werden darf, sondern vielmehr als Konzept zur Organisation und Anbindung der Benutzerschnittstelle, das im Laufe der Zeit in vielen konkreten Varianten umgesetzt wurde.

7.4 Varianten und Abgrenzung

Das MVC Muster wurde schon Ende der 1970er Jahre mit der Entwicklung der ersten graphischen Benutzeroberfläche beschrieben [Ree79]. Seither hat sich zwar nicht das Ziel der Trennung von Model und View geändert, wohl aber wurden mit der Entwicklung der Betriebssysteme, sowie der plattformunabhängigen GUI-Toolkits neue Randbedingungen ge-

schaffen. Dies führt dazu, dass heute verschiedene Varianten des MVC-Musters, teilweise unter dem selben Namen verbreitet sind [Gre07, Introduction].

7.4.1 MVC und MVP

Auf die ursprüngliche Ausprägung des MVC-Musters beziehen sich sowohl [Gam94, Pos. 416] als auch [Kü11, S. 89]. Im Unterschied zu der oben vorgestellten Variante muss der Controller hier Benutzereingaben auf relativ niedrigem Niveau interpretieren und View und Model entsprechend aktualisieren. Beispielsweise ist das markieren von Text in einem Textfeld eine solche Aufgabe (Maustaste wird gedrückt, Maus bewegt sich, Maustaste wird losgelassen, welche Zeichen sind markiert?).
Die Verarbeitung solcher Ereignisse wird seit geraumer Zeit durch den View (z.B. das Textfeld) selbst vorgenommen, der Controller verarbeitet nur noch höherwertige Ereignisse (z.B. der Text hat sich geändert, ein Button wurde gedrückt, etc.), die von einer ganzen Gruppe von Views (Ein Fenster, ein Teil des Fensters, ein Dialog, etc.) stammen. Diese Variante des MVC-Musters ist auch unter dem Namen Model-View-Presenter (MVP) bekannt (in seiner spätesten Ausprägung), wobei der Controller hier den Namen Presenter trägt [Gre07, Dolphin Smalltalk MVP vs. Smalltalk-80 MVC], [Fow06a, MVP].

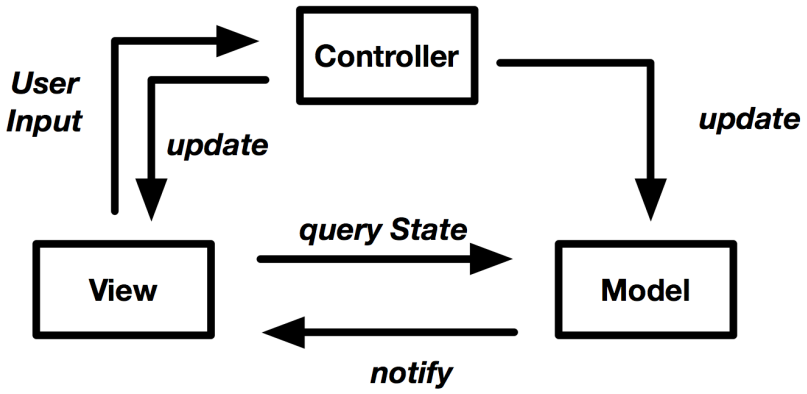


Abb. 7.5: Datenfluss MVC und MVP

7.4.2 Passive View

Eine Weiterentwicklung des MVP-Musters ist die Passive-View Variante. Hier registriert sich der View nicht mehr als Observer am Model, sondern wird vom Controller aktualisiert. Daraus ergeben sich mehrere Konsequenzen: Da die 1:n-Beziehung zwischern Model und View verloren geht, muss sie nun in der Beziehung zwischen Controller (Observer) und Model (Observable) realisiert werden, damit weiterhin mehrere Views die Daten eines Models darstellen können. Der eigentliche Gewinn dieser Variante liegt in der Passivität des Views: Die komplexe Aufgabe die Benutzerschnittstelle zu testen, kann durch ein Testmodul an Stelle des Views automatisiert werden [Fow06b] [App11, S. 163].

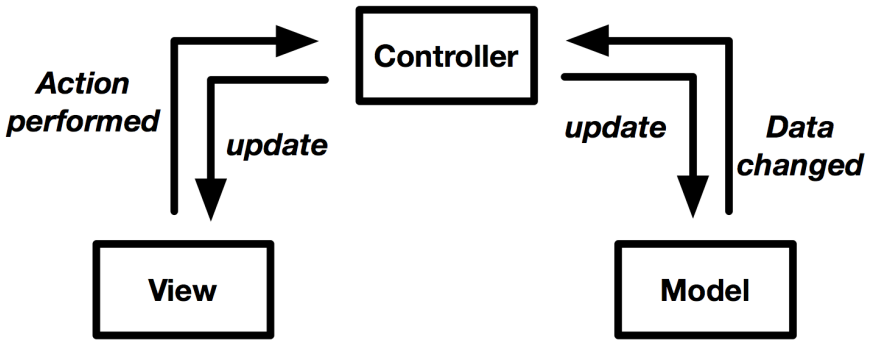


Abb. 7.6: Datenfluss Passive View

7.4.3 Databinding

Ein Teilaspekt des MVC-Patterns wird heute auf vielen Plattformen durch den sogenannten Databindung-Mechanismus abgedeckt. Dies ist eine Variante der Ursprünglichen Observer/ Observable-Beziehung zwischen Model und View. Dabei werden Attribute eines Model-Objekts (bspw. der Titel einer CD) mit den Eigenschaften von View-Elementen (bspw. dem Text eines Textfeldes) verknüpft. So können die Daten des Models mit ihrer Repräsentation im View synchron gehalten werden. Dabei kann durch sog. Binding-Objekte Einfluss auf Formatierung und Validierung der Daten genommen werden. Das Bindingobjekt übernimmt hier zusammen mit dem dahinter stehenden Databindingframework Aufgeben das Controllers im MVP Muster [Fow06a].

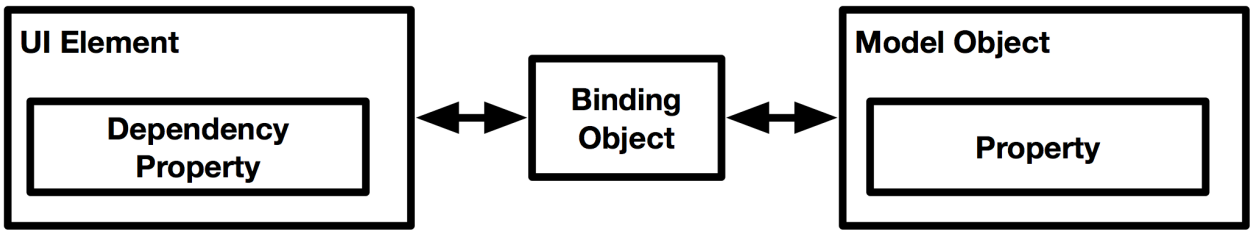


Abb. 7.7: Databinding am Beispiel WPF

7.4.4 Model-View-ViewModel

Eine vergleichsweise neue Entwicklung, die auch das MVC-Pattern betrifft, ist die Tatsache dass die Views auf vielen Plattformen in einer Markupsprache (meist ein XML-Dialekt) deklarativ beschrieben werden. Dies hat den Vorteil, dass die GUIs mit den passenden Tools von Grafikern und Designern gestaltet werden können, während die Logik der Anwendung weiterhin von Programmierern implementiert wird [Gos05].
Ein Beispiel für eine solche Plattform ist die Windows Presentation Foundation (WPF), für die Microsoft das Model-View-ViewModel Pattern beschreibt. Dieses Muster setzt auf die vorherigen Entwicklungen (MVC, MVP, Passive View, Databindung) auf. Dabei ist der

View vollständig deklarativ in XAML beschrieben, und wird an ein zugehöriges ViewModel mit Hilfe von Databindung und Commands gebunden. Das ViewModel enthält also zum einen eine für das Databinding geeignete Repräsentation der relevanten Model-Daten, und zum anderen Methoden („Commands“), die durch Aktionen das Benutzers im View ausgelöst werden [Gos05]. Im Übrigen ist das ViewModel dafür verantwortlich seine Daten mit denen des Models zu synchronisieren. Obwohl das MVVM-Muster nur von Microsoft so beschrieben wird, kann auch auf den Plattformen anderer Hersteller ähnlich vorgegangen werden [Pro11, Android Binding], [App11, Cocoa Bindings, IBActions/Outlets].

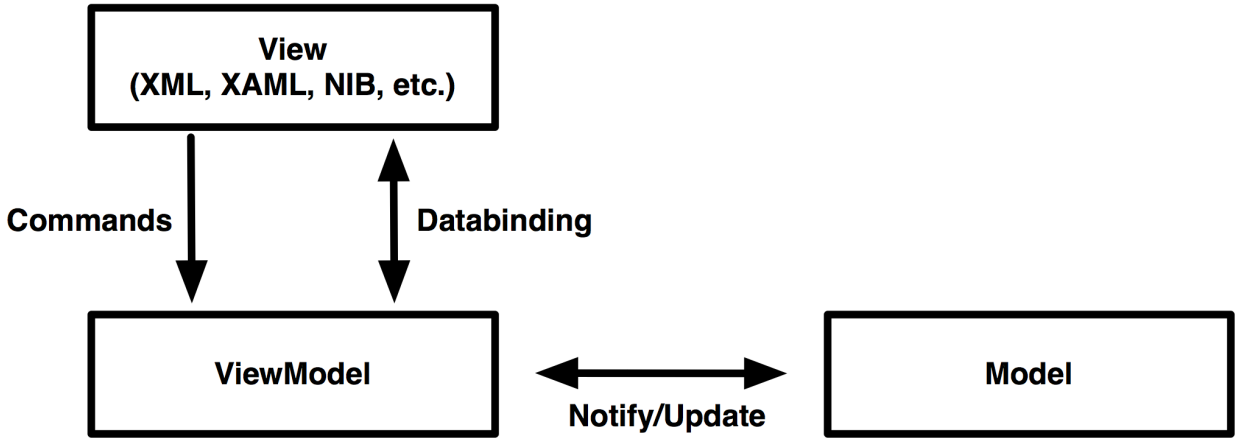


Abb. 7.8: Datenfluss MVVM

7.4.5 Gemeinsamkeiten und Unterschiede

Alle Varianten des MVC-Musters verfolgen das ursprünglich formulierte Ziel, die Daten und Geschäftslogik einer Anwendung von ihrer Benutzerschnittstelle zu trennen. Dies geschieht durch eine lose Kopplung von Model und View (Observer), oder gar eine Entkopplung (Passive View, MVVM). Die konkrete Aufgabenverteilung zwischen View und Controller hängt stark vom Angebot der Plattform ab (Widgets, Databinding, deklarative Views), und spiegelt sich in den unterschiedlichen MVC-Varianten wieder.

7.5 Bewertung

Sobald in der Anwendungsentwicklung eine Benutzerschnittstelle implementiert werden soll, muss für die Trennung von Daten und Logik auf der einen, und der UI auf der anderen Seite gesorgt werden. Die Familie der MVC-Muster bietet für diese Problematik entsprechende Lösungsansätze. Welche der MVC-Varianten anzuwenden ist, hängt im Wesentlichen von der Zielplattform ab, und ist vom Hersteller entsprechend dokumentiert.

8 Strategy

Das Strategiemuster (Strategie), auch bekannt als Policy bzw. Strategy bekannt, gehört zu den objektbasierten Verhaltensmustern.

Es beinhaltet die Definition einer austauschbaren Familie von Algorithmen.

8.1 Definition

Gang Of Four-Definition:

Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren [Gam94].

8.2 Absicht

Beim Einsatz des Strategiemusters im Softwareentwurf werden Kontext und Algorithmen über eine Schnittstelle gekoppelt. Dabei werden austauschbare Algorithmen in separaten Klassen oder Modulen gekapselt. Abhängig vom Einsatz (Kontext) können demzufolge verschiedene Algorithmen verwendet werden, ohne dass die Software nachträglich angepasst werden muss. Jede Algorithmenklasse kommuniziert dabei über dieselbe Schnittstelle.

Dies ermöglicht es Software besser wartbar, flexibler und damit auch zukunftsfähiger zu gestalten. Charakteristische Einsatzmöglichkeiten bestehen zum Beispiel in Sortier- und Suchsoftware. Dort können die verschiedenen Algorithmen abhängig von Datenmenge, Datentyp, Datenstruktur oder anwenderspezifischen Vorgaben Algorithmen gewählt werden [Swt11].

8.3 Struktur

Die Struktur des Strategiemusters ist in Abbildung dargestellt.

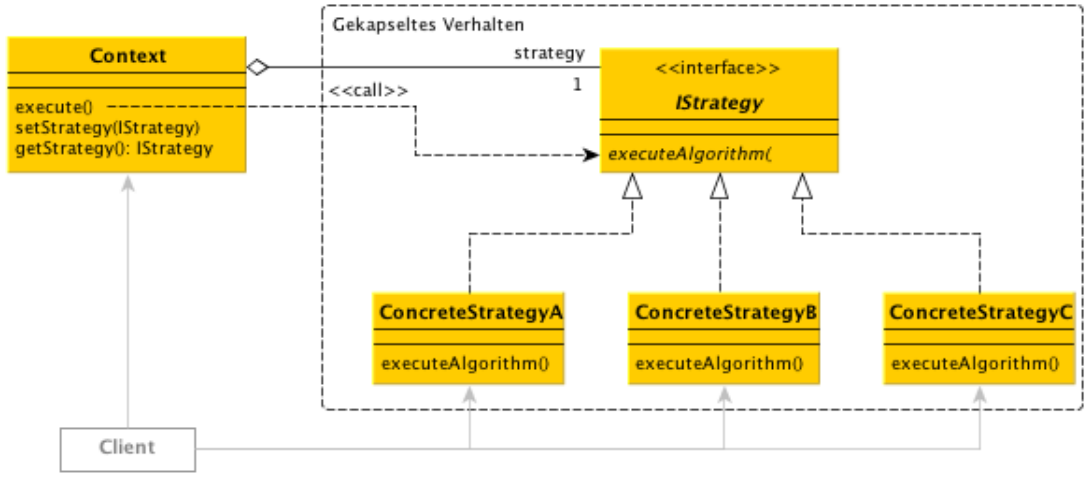


Abb. 8.1: Struktur des Strategiemusters [Hau11a]

In diesem vereinfachten Schema sind die drei wesentlichen Elemente des Strategiemusters enthalten. Zum einen der Kontext (Context), aus dem eine bestimmte Algorithmenfamilie angefordert wird. Diese Kommunikation geschieht über das Strategieobjekt (IStrategy). Hier wird individuell verwaltet, welche konkrete Klasse eines Algorithmus verwendet wird.

Context:

- Beinhaltet eine Referenz zu dem Strategieobjekt (Strategy)
- Kann auch eine Schnittstelle definieren, welche vom Strategieobjekt benötigt wird um auf Daten zuzugreifen.

Strategy:

- Deklariert eine allgemeine Schnittstelle zu allen unterstützten Algorithmen.
- Wird von Context verwendet, um auf konkrete Algorithmen zuzugreifen.

ConceteStrategy:

- Beinhaltet die Implementierung konkreter Algorithmen.
- Wird von Context verwendet, um auf konkrete Algorithmen zuzugreifen.
- Die Klassen der Algorithmenfamilie unterscheiden sich nur in ihrem Verhalten.
- Nach außen hin haben sie jedoch gleiche Schnittstellen und Eigenschaften [Swt11].

Der Klient (Client) stellt eine Simulation oder einen Testlauf mit der Methode main() dar.

8.4 Beispiel

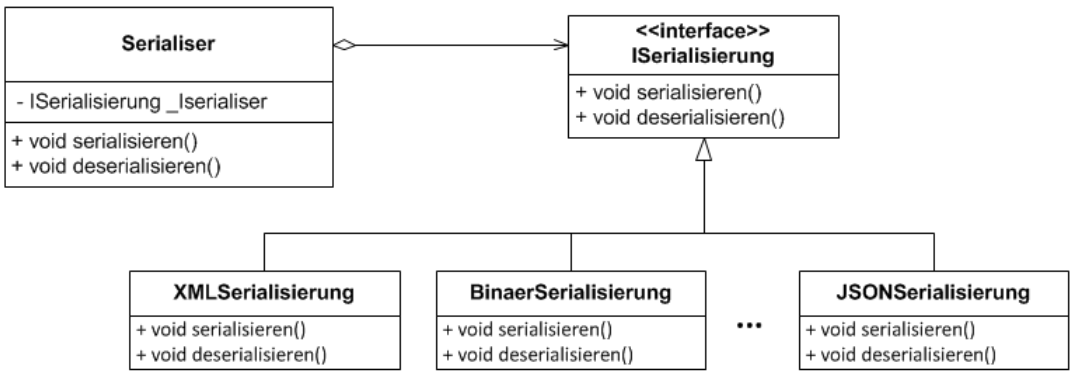


Abb. 8.2: UML-Klassendiagramm des Beispiels [Sen11]

Ein Objekt wird bei der Serialisierung in eine Datei umgewandelt. Der Zustand des Objekts wird gespeichert, damit dieses später wiederhergestellt werden kann. Der umgekehrte Vorgang wird als Deserialisierung bezeichnet. [Mic11]

Mit Hilfe dieses Strategiemusters wird erweiterbare Struktur zur Serialisierung der Objekte erzeugt. Ein Objekt kann zum Beispiel in XML, als binär (Java Object Serialization (JOS)), oder als ein neues Format zum Beispiel als JSON serialisiert oder deserialisiert werden.

8.5 Implementierung

Die Klasse **ISerialisierung** definiert eine Schnittstelle für alle unterstützte Verhalten. Die Strategieklasse kann als Interface oder als eine abstrakte Klasse implementiert werden. Unser Kontextobjekt verwendet intern diese Strategie.

```
1 // StrategieKlasse
2 public interface ISerialisierung {
3         public void serialisieren(String str);
4         public void deserialisieren(String str);
5 }
```

Abb. 8.3: Definition einer Strategie als Interface

Die KonkreteStrategien für unterschiedliche Serialisierungsarten werden einzeln definiert. Die KonkreteStrategien implementieren die Schnittstelle **ISerialisierung**.

```java
//KonkreteStrategie 1
public class XMLSerialisierung implements ISerialisierung {

        public void serialisieren(String str) {
                System.out.println("XML serialisieren");
        }

        public void deserialisieren(String str) {
                System.out.println("XML deserialisieren");
        }
}
//KonkreteStrategie 2
public class BinaerSerialisierung implements ISerialisierung {

        public void serialisieren(String str) {
                System.out.println("Binaer serialisieren");
        }

        public void deserialisieren(String str) {
                System.out.println("Binaer deserialisieren");
        }
}

//KonkreteStrategie 3
public class JSONSerialisierung implements ISerialisierung {

        public void serialisieren(String str) {
                System.out.println("JSON serialisieren");
        }

        public void deserialisieren(String str) {
                System.out.println("JSON deserialisieren");

        }
}
```

Abb. 8.4: Definition der KonkreteStrategien

Diese Klasse **Serializer** hat die Variable **_Iserialiser**, die auf die Schnittstelle referenziert. Auf diese Weise werden die konkreten Algorithmen (Serialisierungstypen) über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit gegen eine andere Implementierung ausgetauscht werden. Über die Methoden können dem Strategieobjekt die Daten als Parameter übergeben werden

```
1  //Kontext
2  public class Serialiser {
3
4          ISerialisierung _Iserialiser;
5
6          //Konstruktor
7          public Serialiser(ISerialisierung Iserialiser)
8          {
9                  _Iserialiser = Iserialiser;
10         }
11
12         //Die Verhalte
13         public void serialisieren(String str)
14         {
15                 _Iserialiser.serialisieren(str);
16         }
17
18         public void deserialisieren(String str)
19         {
20                 _Iserialiser.deserialisieren(str);
21         }
22 }
```

Abb. 8.5: Definition von Kontextklasse Serializer

Am Ende wird ein Klient definiert, der den Kontext **Serializer** verwendet, um die eine Art der Serialisierung durchführen lassen zu können, je nachdem, ob der Serialisierungstyp XML oder als andere X-Formate ist.

```
1  //Klient
2  public class Klient {
3          public static void main(String[] args) {
4                  //Default Verhalten
5                  Serialiser srz = new Serialiser(new XMLSerialisierung());
6
7                  srz.serialisieren("Hallo");
8                  srz.deserialisieren("Hallo");
9
10                 //Verhalten aendern
11                 srz = new Serialiser(new BinaerSerialisierung());
12                 srz.serialisieren("Hallo");
13                 srz.deserialisieren("Hallo");
14
15                 //Verhalten aendern
16                 srz = new Serialiser(new JSONSerialisierung());
17                 srz.serialisieren("Hallo");
18                 srz.deserialisieren("Hallo");
19         }
20 }
```

Abb. 8.6: Definition von Klient

8.6 Bewertung

Vorteile:

- Software, in der das Strategiemuster Anwendung findet, ist leicht erweiterbar und wartbar.

- Neuere Versionen von Algorithmen können dadurch einfach implementiert werden.

- Dank der einheitlichen Schnittstelle können konkrete Algorithmen in anderer Software wieder verwendet werden.

- Auswahl des Anwendungsverhaltens zur Laufzeit und dadurch erhöhen sich die Flexibilität und die Wiederverwendbarkeit.
  Ein Vorteil des Strategiemusters ist die Auswahl des Anwendungsverhaltens zur Laufzeit. Man muss nur die Verhaltensweisen als Strategie-Klasse definieren.

Nachteile:

- Klienten müssen die unterschiedlichen Strategien kennen.
  Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können.

- Erhöhte Anzahl von Objekten
  Die Verwendung von Strategieobjekten erhöht die Anzahl von Objekten in einer Anwendung.

9 Observer

9.1 Einleitung

Im folgenden Kapitel soll das Observer Design Pattern vorgestellt werden. Dazu wird zunächst die abstrakte Problemstellung beschrieben, für welche das Pattern gedacht ist. Nach der Erläuterung des Zwecks, wird auf die Struktur des Patterns eingegangen und die Rollen der einzelnen Akteure erklärt. Das Verständnis dieser Beschreibung soll dann durch ein Beispiel vertieft werden. Zuletzt sollen die sich ergebenden Konsequenzen bewertet werden und einige Beispiele aus der Softwarewelt folgen.

9.1.1 Motivation

Viele Situationen des täglichen Lebens finden sich auch in Softwaresystemen wieder. Um ein Beispiel aufzuzueigen, welches sowohl in der Realität, als auch in der Softwareentwicklung existiert, kann man sich die Arbeitsweise eines Zeitschriftenverlags vorstellen.

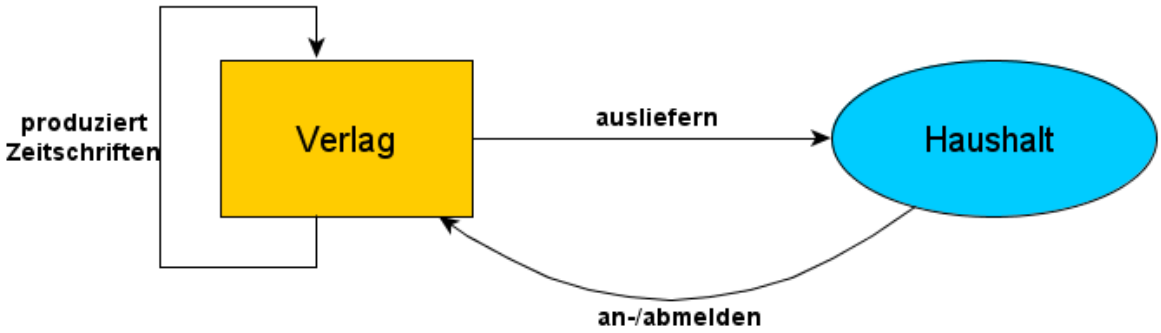


Abb. 9.1: Beziehung zwischen Verlag und Abonnent

Abbildung 9.1 zeigt einen *Verlag* und einen *Haushalt*. Der Haushalt kann sich beim Verlag an- und wieder abmelden. Der Verlag produziert Zeitschriften. Außerdem kennt er die Adresse des bei ihm angemeldeten Haushalts. Sobald eine neue Ausgabe vorliegt, wird der Verlag diese an den angemeldeten Haushalt ausliefern.

Üblicherweise wird nicht nur an einen Haushalt ausgeliefert, sondern es gibt Anmeldungen von zahlreichen unterschiedlichen Abonnenten: Haushalte, Unternehmen, Arztpraxen, Hotels, usw. Obwohl alle diese Einrichtungen sehr unterschiedlich sind, braucht der Verlag zur erfolgreichen Auslieferung jeweils lediglich eine Adresse.

Um aus diesem Beispiel eine Brücke zur Softwareentwicklung zu schlagen, soll die beschriebene Situation etwas formaler ausgedrückt werden:

Zwischen dem VERLAG *und den* ABONNENTEN *besteht eine 1-zu-n-Abhänigkeit. Die Produktion einer neuen Zeitschrift ändert den Zustand des Verlags. Sobald die Produktion einer neuen Ausgabe abgeschlossen ist, erhalten alle angemeldeten Abonnenten diese automatisch.*

Während der Entwicklung eines Softwaresystems befindet man sich häufig einer ähnlichen Situation. Dies trifft z.B. dann zu, wenn es gilt dem Benutzer Anwendungsdaten zu präsentieren. Die Verarbeitung von Daten erfolgt üblicherweise in einer als Business-Logic bezeichneten Schicht des Gesamtsystems. Hier werden Daten berechnet, mit anderen Daten verknüpft und durch Algorithmen verändert. Es ist allgemein darauf zu achten, dass diese Schicht und die spätere Präsentation (z.B. auf Bildschirm, Drucker o.ä.) nicht eng miteinander gekoppelt wird. Eine enge Kopplung würde die Wiederverwendbarkeit der Klassen einschränken.

Dennoch ist sicherzustellen, dass Daten- und Präsentationsschicht stets konsistent sind. Die Objekte der verschiedenen Schichten sollen also nur lose gekoppelt sein, sich aber so verhalten, als würden sie sich kennen. Um diesen Effekt zu errechen, gibt das Observer Design Pattern Empfehlungen, wie die Schnittstelle zwischen den Objekten aussehen kann. Das Pattern gibt also nicht vor wie die Klassen zu implementieren sind, sondern beschreibt wie sie miteinander in Beziehung treten können, um das gewünschte Verhalten zu realisieren. Der formale Zweck des Patterns lautet:

> *Definiere eine 1-zu-n-Abhänigkeit zwischen Objekten, so dass eine Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.* [Gam94]

9.2 Struktur

Abbildung 9.2 zeigt die Struktur des Observer Pattern in UML-Notation:

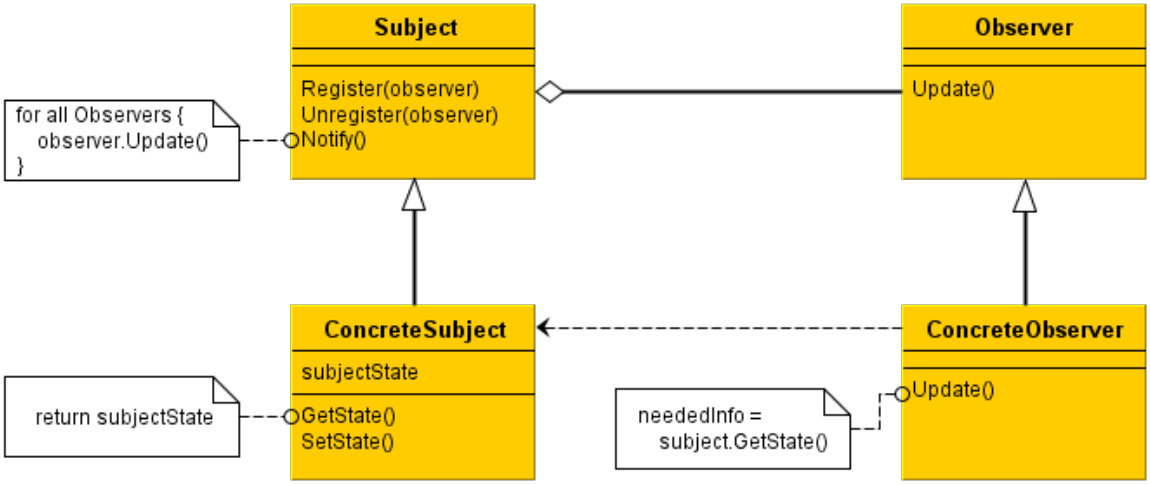


Abb. 9.2: Observer Klassendiagramm

9.2.1 Akteure

- Subject
  - ist meist eine abstrakte Klasse
  - implementiert grundlegende Funktionen
  - bietet eine Schnittstelle über die sich Observer an- und abmelden
  - verwaltet Referenzen auf angemeldete Observer
- Observer
  - ist meist ein Interface oder eine abstrakte Klasse
  - legt die Schnittstelle fest mit der Observer über Aktualisierungen benachrichtigt werden
  - das Subject kennt von seinen Observer nur diese minimale Schnittstelle
- ConcreteSubject
  - erbt von Subject
  - verfügt über einen internen Status
  - benachrichtigt angemenldete Observer bei einem Statuswechsel
- ConcreteObserver
  - erbt von Observer, bzw. implementiert dessen Methode(n)
  - enthält eine Referenz auf ein ConcreteSubject, um bei eine Aktualisierung benötigte Informationen anfragen zu können
  - kann sich selbstständig beim Subject an- und abmelden

9.3 Beispiel

Nach der Beschreibung des Observer Patterns soll nun das Verständnis mit Hilfe einer einfachen Beispielimplementierung vertieft werden. Hierfür stelle man sich folgende Situation vor:

Ein Lieferant für Lebensmittel aller Art passt in unregelmäßigen Abständen die Preislisten für die angebotenen Waren an. Die Aktualisierungen ergeben sich aus der aktuellen Marktsituation; die exakte Preisbildung findet aber beim Lieferanten intern statt und ist für Außenstehende nicht transparent. Der Ablauf ist hierbei auch nicht interessant, das Ergebnis ist es, worauf es ankommt. Außenstehende haben also ein Interesse daran, stets über die aktuellen Preise informiert zu sein. Um dies zu erreichen, bietet der Lieferant Interessenten die Möglichkeit sich anzumelden, um bei einer neuen Preisliste automatisch benachrichtigt zu werden. Die Intentionen der Interessenten selbst, sind grundverschieden. Zu den aktuellen Abonnenten gehören

- eine Dönerbude, die für das Tagesgeschäft einkauft
- eine Privatperson, die die Preisentwicklung im Lebensmittelbereich beobachtet
- ein Großhandel, der bei einem bestimmten Preis sofort bis zu seinen Kapazitätsgrenzen bestellt.

Außerdem soll es erlaubt sein, jederzeit weitere, bisher unbekannte Abonnenten anmelden zu können.

Durch das aus den vorhergehenden Abschnitten gesammelte Vorwissen, ergibt sich aus der Vorgabe schnell das in Abbildung 9.3 gezeigte Klassendiagramm.

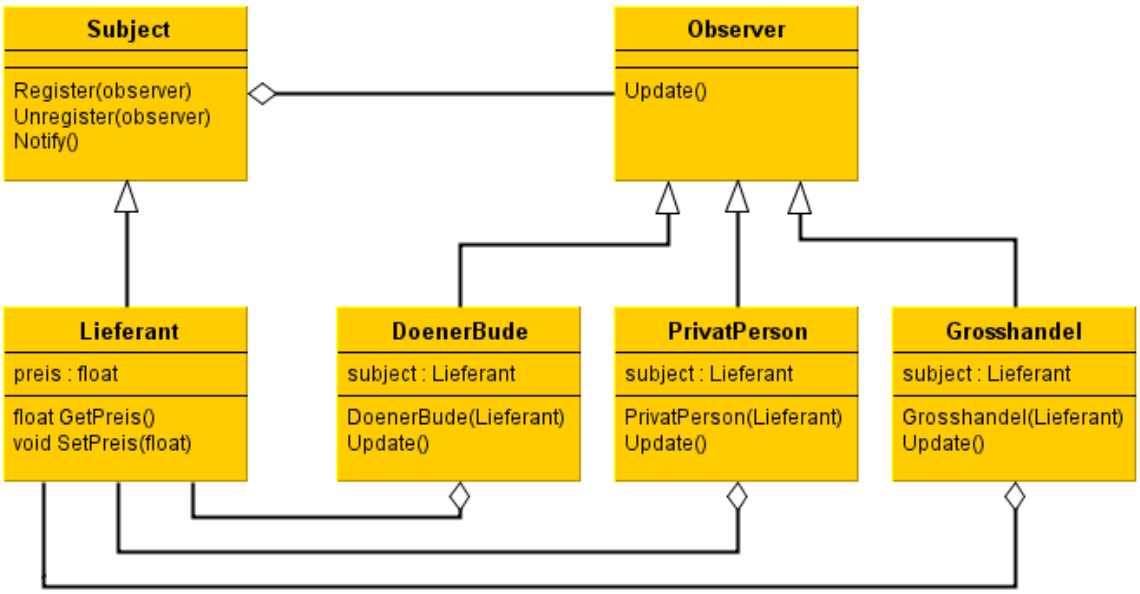


Abb. 9.3: Klassendiagramm der Beispielimplementierung

Um das Beispiel möglichst klein und verständlich zu halten, soll die Preisliste durch einen einzigen, aus einer *float*-Variable bestehenden, Preis repräsentiert werden. Eine Implementierung in der Java Programmiersprache folgt.

Den Anfang macht das Subject. Dieses wird in Listing 9.4 als abstrakte Klasse definiert.

```
1  import java.util.*;
2
3  public abstract class Subject {
4          protected List<Observer> obs =
5              new LinkedList<Observer>();
6
7          public void Register(Observer obs) {
8                  this.obs.add(obs);
9          }
10
11         public void Unregister(Observer obs) {
12                 this.obs.remove(obs);
13         }
14
15         public void Notify() {
16                 for (Observer o : obs) {
17                         o.Update();
18                 }
19         }
20 }
```

Abb. 9.4: Quelltext der Datei Subject.java

Die implementierten Methoden beschränken sich auf generischen Subjectcode. Dieser wäre, neben der Lieferantenklasse, auch für andere, von Subject erbende Klassen relevant.

Als Nächstes folgt in Listing 9.5 die Implementierung des konkreten Subject, die Lieferantenklasse.

```
1  public class Lieferant extends Subject {
2          private float preis;
3
4          public float getPreis() {
5                  return preis;
6          }
7
8          public void setPreis(float preis) {
9                  this.preis = preis;
10                 Notify();
11         }
12 }
```

Abb. 9.5: Quelltext der Datei Lieferant.java

Die abstrakte Subjectklasse zeigt bereits, wie mit den Observern zu verfahren ist. Daraus ergibt sich das in Listing 9.6 beschriebene minimale Interface.

```
1  public interface Observer {
2          public void Update();
3  }
```

Abb. 9.6: Quelltext der Datei Observer.java

Dieses Interface wird nun von allen konkreten Observerklassen implementiert. Dazu zählen, wie Anfangs bereits beschrieben, die Dönerbude (Listing 9.7), die Privatperson (Listing 9.8) und der Großhandel (Listing 9.9).

```
1  public class DoenerBude implements Observer {
2          private Lieferant subject;
3          private String lokation;
4
5          public DoenerBude(Lieferant lieferant, String lokation) {
6                  this.subject = lieferant;
7                  this.lokation = lokation;
8                  this.subject.Register(this);
9          }
10
11         public void Update() {
12                 final float preis = this.subject.getPreis();
13                 System.out.println("Bude aus " + lokation +
14                                 ": Preis ist " + preis + " Euro");
15         }
16 }
```

Abb. 9.7: Quelltext der Datei DoenerBude.java

```
1  public class PrivatPerson implements Observer {
2          private Lieferant subject;
3          private String name;
4
5          public PrivatPerson(Lieferant lieferant, String name) {
6                  this.subject = lieferant;
7                  this.name = name;
8                  this.subject.Register(this);
9          }
10
11         public void Update() {
12                 final float preis = this.subject.getPreis();
13                 System.out.println(name + ": Preis ist " + preis + " Euro");
14         }
15 }
```

Abb. 9.8: Quelltext der Datei PrivatPerson.java

```
1  public class Grosshandel implements Observer {
2          private Lieferant subject;
3
4          public Grosshandel(Lieferant lieferant) {
5                  this.subject = lieferant;
6                  this.subject.Register(this);
7          }
8
9          public void Update() {
10                 // Der Grosshandel erhaelt 50 Cent Nachlass.
11                 final float preis = this.subject.getPreis();
12                 System.out.println("Grosshandel: Preis ist "
13                  + (preis - .5f) + " Euro");
14         }
15 }
```

Abb. 9.9: Quelltext der Datei Grosshandel.java

Nach der Implementierung aller beschriebenen Klassen, muss noch ein Kontext erstellt werden, innerhalb dessen die Funktionalität demonstriert werden kann. Listing 9.10 zeigt hierfür eine *Main()*-Methode, in der alle notwendigen Objekte angelegt und initialisiert werden.

```
1  public class HelloObserver {
2          public static void main(String[] args) {
3                  Lieferant lieferant = new Lieferant();
4
5                  DoenerBude bude =
6                                  new DoenerBude(lieferant, "Wuerzburg");
7                  PrivatPerson bob =
8                                  new PrivatPerson(lieferant, "Bob");
9                  Grosshandel handel =
10                                 new Grosshandel(lieferant);
11
12                 lieferant.setPreis(5.0f);
13                 System.out.println("");
14                 lieferant.setPreis(8.0f);
15                 System.out.println("");
16                 lieferant.setPreis(2.0f);
17         }
18 }
```

Abb. 9.10: Quelltext der Datei HelloObserver.java

Am Ende der Methode, erfährt der Lieferant drei Preisänderungen. Nach jeder dieser Änderungen, werden die angemeldeten Observer benachrichtigt und können die für sie notwendigen Schritte unternehmen. Im Beispiel geben sie lediglich eine Textzeile aus. Listing 9.11 zeigt die Ausgabe auf der Konsole, nachdem das Programm ausgeführt wurde.

```
1  Bude aus Wuerzburg: Preis ist 5.0 Euro
2  Bob: Preis ist 5.0 Euro
3  Grosshandel: Preis ist 4.5 Euro
4
5  Bude aus Wuerzburg: Preis ist 8.0 Euro
6  Bob: Preis ist 8.0 Euro
7  Grosshandel: Preis ist 7.5 Euro
8
9  Bude aus Wuerzburg: Preis ist 2.0 Euro
10 Bob: Preis ist 2.0 Euro
11 Grosshandel: Preis ist 1.5 Euro
```

Abb. 9.11: Konsolenausgabe

Mit dieser Architektur haben wir die Vorgabe erreicht, neue Observer hinzufügen zu können. So wäre es ohne Weiteres möglich, eine Klasse für einen Online-Kühlschrank zu erstellen, der durch eine geschickte Verknüpfung verschiedener Vorgaben (z.B. Preis, Füllstand, Diätplan) selbstständig Bestellungen tätigen kann.

9.3.1 Alternativen

Die vorgestellte Implementierung ist nur eine von vielen Möglichkeiten das Observer Pattern zu realisieren. Da das Pattern keine strikten Vorgaben macht, bleibt Raum für Interpretation. So kann die Beziehung zwischen Subject und den Observern besser an die jeweilige

Problemstellung angepasst werden. Zwei der wichtigsten Möglichkeiten sollen vorgestellt werden.

Pull- versus Push-Modell

Die in der Beispielimplementierung erstellte Aktualisierungsschnittstelle ist minimal, sie besteht lediglich aus dem Aufruf der *Update()*-Methode. Die Methode erhält beim Aufruf weder Argumente, noch gibt der Methodenname Anhaltspunkte darüber, um welche Art von Aktualisierung es sich handelt. Diese Art der Implementierung der Aktualisierungsschnittstelle bezeichnet man als *Pull*-Modell. Dabei werden keine Annahmen darüber gemacht, wie die konkreten Observer aussehen und welche Aktualisierungsinformationen sie benötigen. Jeder konkrete Observer muss innerhalb seiner Update()-Implementierung das Subjectobjekt nach den Informationen fragen, die er benötigt.

Das andere Extrem, ist als *Push*-Modell bekannt. Dabei trifft der Designer der Observerschnittstelle bereits Annahmen darüber, welche Informationen die konkreten Observer brauchen werden. Diese Informationen können dann als Argumente an die Update()-Methode übergeben werden. Im Fall der Beispielimplementierung, könnte also die Update()-Methode den neuen Preis direkt mitliefern. In diesem Fall, müsste die Implementierung von *Notify()* allerdings in das konkrete Subject (Lieferant) verschoben werden.

Die Entscheidung für eines der genannten Modelle hängt hauptsächlich von der gegebenen Situation ab. Gemeinhin wird jedoch das Push-Modell als „korrekter“angesehen [Fre04, Seite 74].

Der Notify()-Aufruf

Wie wir im Beispiel gesehen haben, ruft die Lieferantenklasse bei jeder Änderung des Zustands via *setPreis()* intern *Notify()* auf. Das konkrete Subject kümmert sich also selbst darum, seine Observer über Statusänderungen zu benachrichtigen. Der sich daraus ergebende Vorteil ist, dass die Nutzer der Lieferantenklasse nicht daran denken müssen.

Eine andere Möglichkeit ist es, den *Notify()*-Aufruf dem Nutzer der Lieferantenklasse zu überlassen. Der Vorteil liegt darin, der Nutzer warten kann, bis mehrere Zwischenzustände gesetzt wurden, um die Observer erst danach zu informieren.

9.4 Bewertung

Für die Bewertung des Observer Patterns sollen einige wichtige Punkte herangezogen werden.

Lose Kopplung

Die Subject- und Observerklassen sind lose gekoppelt. Das Subject kennt seine Observer nicht direkt, sondern besitzt lediglich eine Liste von deren einheitlicher Schnittstelle. Die dahinterliegenden konkreten Observerklassen sind ihm nicht bekannt. Dadurch ist es möglich neue konkrete Observerklassen zu erstellen, ohne den Code der abstrakten oder konkreten Subjectklasse anpassen zu müssen. Die Kopplung ist daher minimal und abstrakt.

Multicast-Kommunikation

Das Observer Pattern ermöglicht sog. Multikast-Kommunikation. Dabei spezifiziert das Subject bei einer Aktualisierung den Nachrichtenempfänger nicht. Stattdessen wird die Aktualisierung automatisch und ohne Einschränkung an alle angemeldeten Observer versandt. Die Observer sind bei einer Benachrichtigung selbst dafür zuständig, diese zu bearbeiten oder zu ignorieren. Die Observer sind weiterhin selbst dazu in der Lage sich jederzeit beim betreffenden Subject an- und abzumelden.

In Kombination mit der losen Kopplung kann sich hieraus ein beachtenswerter Nachteil ergeben. Da das Subject seine konkreten Observer nicht kennt, kann es nicht wissen was eine Aktualisierung (*Notify()*) kostet. So kann es sein, dass sich der Zustand des Subjects häufig ändert, die Bearbeitung der Aktualisierung (*Update()*) in einem Observer aber relativ viel Zeit erfordert. In diesem Fall wartet das gesamte System. Es ist dann die Aufgabe des konkreten Observer dafür zu sorgen, dass das System nicht zu lange stillsteht, indem die Bearbeitung ausgelagert oder verschoben wird.

Beobachten mehrerer Subjects

Es ist möglich, einen Observer bei mehr als einem Subject anzumelden. Voraussetzung dafür ist, dass jedes konkrete Subject dieselbe Observerschnittstelle für die Aktualisierung verwendet. Dabei muss der Observer allerdings erfahren *welches* Subject die Benachrichtigung ausgelöst hat. Eine einfache Lösung hierfür ist, die Update()-Methode der Observerschnittstellt dahingehend zu erweitern, dass sich das Subject selbst als Argument übergibt, etwa *void Update(subject)*. Da der konkrete Observer Referenzen auf alle Subjects hält bei denen er angemeldet ist, kann er diese gegen das Argument der Update()-Methode prüfen.

Komplexe Aktualisierungsmechanismen

Wenn sich der Status des Subject sehr häufig ändert, müssen bei jeder Änderung alle Observer benachrichtigt werden. Dies kann zu unnötigen Aktualisierungen führen, v.a. wenn sich manche Observer nur für bestimmte Änderungen interessieren, andere aber ignorieren. Um diesen Aufwand zu reduzieren, ist es möglich ein sog. Aspekt-Konzept einzuführen. Dabei muss ein Observer bei der Anmeldung an ein Subject einen Hinweis darauf geben, für welche Art der Aktualisierungen er sich interessiert. Die Register()-Methode des Subject sähe dann etwa so aus: *void Register(Observer obs, Aspect interest)*. Die Observerschnittstelle könnte dann für jeden Aspekt eine eigene *UpdateAspectXYZ()*-Methode bereitstellen, oder den aktuellen Askpekt als Argument an die die Update()-Methode übergeben: *void Update(Aspect interest)*.

Ist eine noch komplexere Aktualisierungssemantik nötig, sollte man die Verwendung des **MVC** Architekturpatterns in Betracht ziehen.

9.5 Bekannte Verwendungen

Die vermutlich bekannteste Verwendung des Observer Pattern, findet sich im MVC (Model-View-Controller) Architekturpattern. Dabei wird das Subject als *Model* bezeichnet, die Ob-

serverschnittstelle heißt *View* und der *Controller* kümmert sich um die Aktualisierungssemantik. Auf MVC wird in einem anderen Kapitel detailliert eingegangen.

Daneben kommt das Pattern unter der Bezeichnung Signal-Slot-Konzept in zahlreichen Frameworks und Klassenbibliotheken zum Einsatz, z.B. Qt[Nok11] und boost.signals[boo11].

10 Chain of Responsibility

Chain of Responsibility erlaubt das Bearbeiten einer Anfrage von mehreren Empfängern, ohne dass sich Sender und Empfänger gegenseitig kennen.

In diesem Kapitel wird das Chain of Responsibility Design Patter beschrieben, dieses zählt zu der Gruppe der Behavioral Patterns. Das Verhaltens Pattern, welches in der deutschen Übersetzung auch Zuständigkeitskette genannt wird, wird folgend hinsichtlich seiner Definition und Struktur in der Theorie untersucht und ergänzend an einem Beispiel erläutert. Abschließend wird ein Fazit gezogen und die Vorteile sowie auch die Nachteile werden benannt.

10.1 Definition

Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an einer Kette entlang, bis ein Objekt sie erledigt. [Gam94]

Führe eine Kette in einem System ein, damit Nachrichten entweder dort behandelt werden, wo sie empfangen wurden oder an ein Objekt weitergeleitet werden, welches die Nachricht bearbeiten kann. [Ste01]

10.2 Motivation

Ein Problem bei größeren Programmen besteht darin, dass jedes Objekt die anderen Objekte kennen muss, mit denen es kommunizieren will. Es muss wissen, bei welchen Anfragen es sich an welches Objekt zu wenden hat. Hieraus entsteht eine enge Kopplung zwischen den Objekten oder es werden alternativ große Verwaltungsobjekte erzeugt, die einem Objekt Auskunft geben können, wer der Ansprechpartner ist.

10.3 Absicht

Die Absicht für das Chain of Responsibility Design Pattern besteht darin, dem Programmierer ein Werkzeug in die Hand zu geben, mit dessen Hilfe er die Kopplung zwischen Objekten in einem Programm verringern kann. Neben der verringerten Kopplung wird auch versucht, die Programmkommunikation flexibler zu gestalten.

10.4 Struktur

Das Chain of Responsibility Pattern verfolgt das Entkoppeln der einzelnen Elemente durch eine sich während Laufzeit findenden Kette. Der Sender kennt den Einstiegspunkt in die Kette explizit, aber den Bearbeiter nur implizit als Mitglied der Kette. Damit das erste Kettenglied die Nachricht entgegennehmen kann, muss es über eine kompatible Schnittstelle verfügen oder alternativ wie in der Sprache Smalltalk die Möglichkeit besitzen, dass eine nicht interpretierbare Anforderung an ein anderes Objekt weiter geleitet wird. Die Form der Kette sollte ein linearer Graph sein, durch diese Form hat jedes Kettenmitglied höchstens einen Nachfolger.

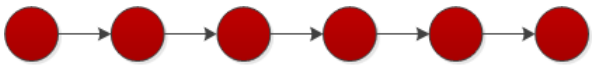


Abb. 10.1: Bild eines linearen Graphen

Zur Erstellung der Verkettung müssen die Regeln bekannt sein, die die Hierarchie wiederspiegeln. Neben Bäumen, die von dem Blatt zur Wurzel einen linearen Graphen bilden, eignen sich auch Ringstrukturen sehr gut, um einen linearen Graphen zu erstellen. Je komplexer jedoch die Ausgangsstruktur und deren hierarchische Einordnung in den neuen Graphen ist, umso komplexer wird auch das Erstellen der Regeln, um zur Laufzeit eine Kette zu generieren. Unter allen Umständen muss darauf geachtet werden, dass keine Zirkelbezüge in der Kette entstehen, da diese zu Endlosschleifen führen können.

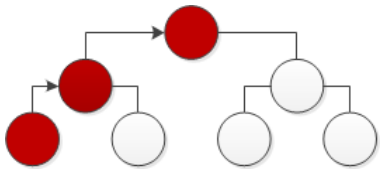


Abb. 10.2: Linearer Graph in einer Baumstruktur

Damit alle Objekte einer Kette auch problemlos mit einer ankommenden Nachricht umgehen können, wird ein abstrakter Handler erstellt, den alle Kettenmitglieder sich einverleiben, siehe Klassendiagramm nach GoF. Da jedes Objekt aus der abstrakten Klasse die Möglichkeit übernommen hat eine Referenz auf einen Nachfolger zu besitzen, der auch vom Typ des abstrakten Handlers ist, kann es nun als Kettenmitglied fungieren.

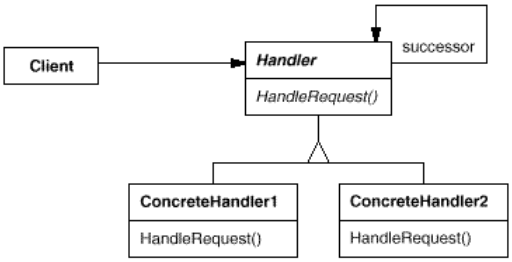


Abb. 10.3: Klassendiagramm nach GoF

Die Funktion, die mit der ankommenden Nachricht hantiert, muss immer die Nachricht weiterleiten, wenn sie für diese nicht zuständig ist. Sollte das Ende der Kette erreicht sein, geschieht nichts. Die Nachricht kann auch trotz Behandlung weitergesendet werden. Ob dies nun eine neue Nachricht ist oder immer noch dieselbe wird in der Literatur nicht geklärt, viele der angeführten Beispiele funktionieren nach diesem Prinzip. Dies geht soweit, dass in manchen Skripten deutscher Hochschulen Beispiele zu finden sind, in denen modifizierte Nachrichten auch noch dem ursprünglichen Sender zugeordnet werden.

10.5 Beispiel

Ein Beispiel zur Verdeutlichung der Struktur: Die Entscheidungskette einer Firma ist hierarchisch angeordnet. Es gibt die Vertreter, deren Abteilungsleiter und den Vorstand. Im Rahmen der Vertragsabschlüsse versuchen die Kunden regelmäßig über den Preis zu verhandeln. Die Firma hat aus diesem Grund eine Richtlinie herausgegeben, die besagt, dass Vertreter einen maximalen Nachlass von 5 Prozent gewähren dürfen, die Abteilungsleiter maximal 10 Prozent und alle Nachlässe über 10 Prozent müssen vom Vorstand entschieden werden. Mitarbeiter kann in der Firma nur werden, wer vorher die Schulung „Verhandlungstechniken und Vorschriften“ besucht hat. Dieses ist im Programm die abstrakte Klasse Mitarbeiter, die dafür sorgt, dass ein Handler für die Verhandlungsfähigkeit erzeugt wird. Jedem Mitarbeiter ist selbstverständlich sein Vorgesetzter bekannt, aber die weitere firmeninterne Struktur ist ihm unbekannt.

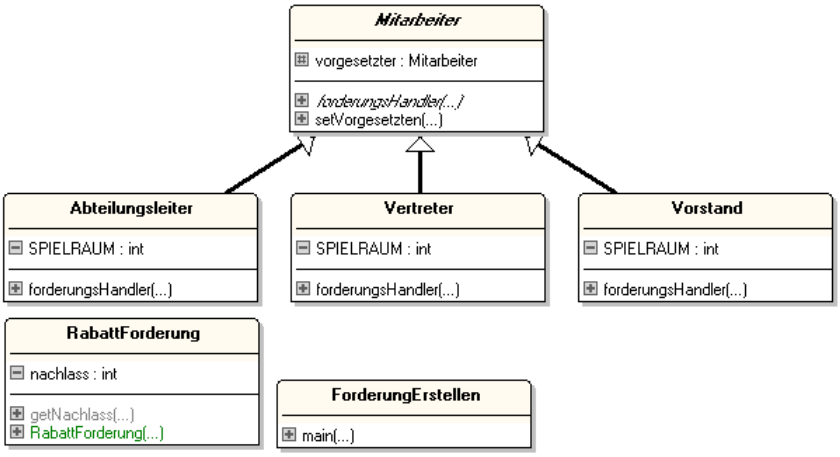


Abb. 10.4: Klassendiagramm des Beispiels

```
1  abstract class Mitarbeiter{
2          protected Mitarbeiter vorgesetzter;
3          public void setVorgesetzten(Mitarbeiter vorgesetzter){
4                  this.vorgesetzter=vorgesetzter;
5          }
6          abstract public void forderungsHandler(RabattForderung forderung);
7  }
```

Abb. 10.5: Implementierung der Klasse Mitarbeiter

```
1  class Vertreter extends Mitarbeiter{
2          private final int SPIELRAUM =5;
3          public void forderungsHandler(RabattForderung forderung){
4                  if(forderung.getNachlass() <= SPIELRAUM)
5                          System.out.println("Vertreter akzeptiert die Forderung");
6                  else
7                          if(vorgesetzter != null){
8                          System.out.println("Weiterleitung an Vorgesetzten");
9                                  vorgesetzter.forderungsHandler(forderung);
10                         }
11                         else
12                                 System.out.println("Kein Vorgesetzter vorhanden,
13                                 Anfrage wurde nicht behandelt");
14         }
15 }
```

Abb. 10.6: Implementierung der Klasse Vertreter

Wenn nun ein Kunde an einen Mitarbeiter mit dem Wunsch auf Preisnachlass herantritt, überprüft dieser Mitarbeiter, ob er es alleine entscheiden kann oder an seinen Vorgesetzten geben muss.

10.6 Verwendungsbereiche

Nach dem Buch Head First Design Patterns gehört dieses Pattern zu den eher selten benutzten Pattern. Die meisten Einsätze finden sich in der Oberflächengestaltung und bei der Eventkommunikation. Unter anderem wird das Chain of Responsibility Pattern von dem ET++ Framework eingesetzt oder auch von dem Servlet-Filter Framework. Auch im Exeption-Handler System von Java findet es sich wieder.

10.7 Kritik

Der größte Kritikpunkt an dem Pattern ist die fehlende Garantie auf Abarbeitung. Dies hat zur Folge, dass der Sender nur Anfragen stellen kann, mit deren Nichtbeantwortung er nicht in Bedrängnis kommt.

Des Weiteren ist bei größeren Programmen das Überwachen und Debuggen sehr schwer, da die Übersichtlichkeit verloren geht. Dies liegt an der Kette, die sich während der Laufzeit des Programms bildet. Durch die schlechte Überwachbarkeit ist die einfache Schnittstelle, die vorteilhaft ist, damit andere Programmierer Teile beisteuern können, ein Sicherheitsrisiko. Es kann der weitere Ablauf der Kette manipuliert werden, in dem man die Regel für den Nachfolger verändert, auch die Nachricht kann modifiziert werden.

Sollten Ketten sehr groß sein und viele Nachrichten gar nicht oder erst spät in der Kette behandelt werden, kann dies zu Performanceproblemen führen.

10.8 Fazit

Auf dieses Pattern sollte man immer mal wieder schauen, wenn man eine Nachrichtenkette in einem Programm hat, vor allem, wenn die Struktur dieser Kette schon vorgegeben ist und die einzelnen Objekte den Nachfolger der Kette kennen. Es ist sicher nicht geeignet, um mit vielen Programmierern und unübersichtlichen Kettenregeln in kurzer Zeit etwas zu realisieren, da die Fehleranfälligkeit zu hoch ist und ein Fehler sehr viel Zeit in der Behebung kostet.

Vorteile

- Flexibilität durch laufzeitabhängiges Bilden der Hierarchie
- Vereinfacht die Objekte, da sie die Struktur nicht kennen müssen
- Entkoppelt Sender und Empfänger, dass sie sich nicht kennen müssen

Nachteile

- Keine Garantie auf Bearbeitung der Anfrage
- Unübersichtlichkeit der Struktur beim Debuggen
- Komplexe Strukturen haben komplexe Regeln
- Bei zu großen Ketten kann es zu Performanceproblemen kommen

11 State

11.1 Kurzbeschreibung

Mit dem Design Pattern „State“ können Zustandsmaschinen elegant und erweiterbar realisiert werden und Gegebenheiten, bei denen Zustände auftreten, in Objektorientierten Code umgesetzt werden.

11.2 Gang Of Four-Definition

> „Ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.“[Hau11b]

11.3 Absicht / Ziel

In vielen aus der Realität abstrahierten Gegebenheiten treten Zustände auf. In der strukturierten Programmierung wurde in diesem Fall häufig eine Zustandsvariable definiert, welche den aktuellen Zustand repräsentiert. Dies führte zu umfangreichen und unübersichtlichen IF-Abfragen. Das State-Pattern versucht diese mit Hilfe der Objektorientierung zu vermeiden. Hierfür soll ein Objekt erzeugt werden, das sein Verhalten ändert, wenn sich sein interner Zustand ändert.

11.4 Struktur

11.4.1 Übersicht

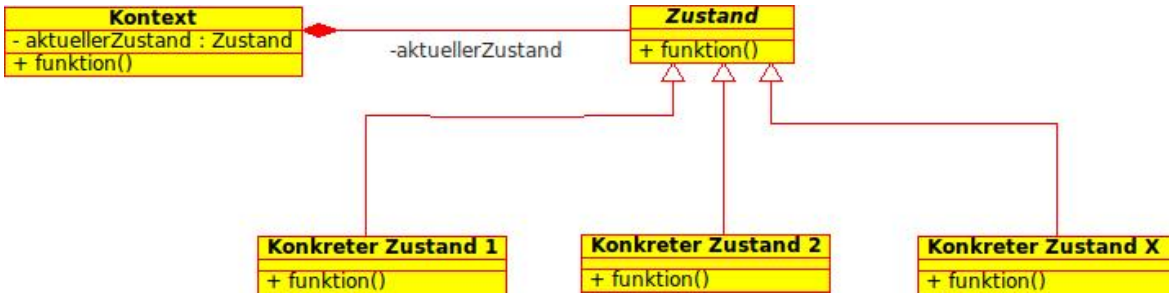


Abb. 11.1: Prinzipieller Aufbau des Patterns

Jeder Zustand wird in eine eigene Klasse ausgelagert, welche von der abstrakten Klasse „Zustand“ erbt oder ein Interface implementiert. Dieses Konstrukt repräsentiert nun das Zustandsobjekt und soll sich dem aktuellen Zustand entsprechend Verhalten. Das Zustandsobjekt muss einem Kontext zugeordnet werden, welcher den Zustand annehmen kann.[Wim10][Lov11]

11.4.2 Zustandsübergänge

Erklärung

Es gibt mehrere Möglichkeiten einen Zustandsübergang zu initiieren. Auf diese möchten wir im Folgenden genauer eingehen.[Hau11b]

Vom Kontext bestimmt

In der einfachsten Variante wird der Zustand vom Kontext bestimmt (siehe Abb. 11.1). Dies hat den Vorteil, das die Zustände keinen Verweis auf den Kontext benötigen, weil sie nicht für Zustandsübergänge verantwortlich sind. Des weiteren werden die Zustände so möglichst schlank. Allerdings erhöht sich die Komplexität des Kontextobjektes. Diese Variante ist relativ unflexibel und schwierig zu erweitern, daher sollte sie nur verwendet werden, wenn die Komplexität des Automaten überschaubar ist und Erweiterungen unwahrscheinlich sind.

Aus Rückgabewert

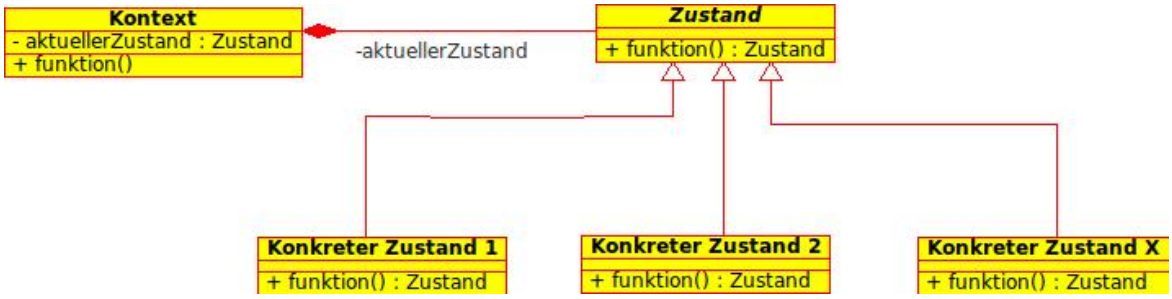


Abb. 11.2: Aufbau des Patterns mit Zustandswechsel aus Rückgabewert

In dieser Variante liefern die konkreten Zustände beim Aufruf der entsprechenden Funktionen den neuen Zustand als Rückgabewert zurück. Der eigentliche Zustandswechsel wird weiterhin im Kontext vorgenommen, wird aber im Vorgängerzustand erzeugt. Da bei dieser Variation der Rückggabewert der Funktionen der konkreten Zustände durch den neuen Zustand belegt ist, können diese keine anderen Ergebnisse zurückliefern.

Vom Zustand bestimmt

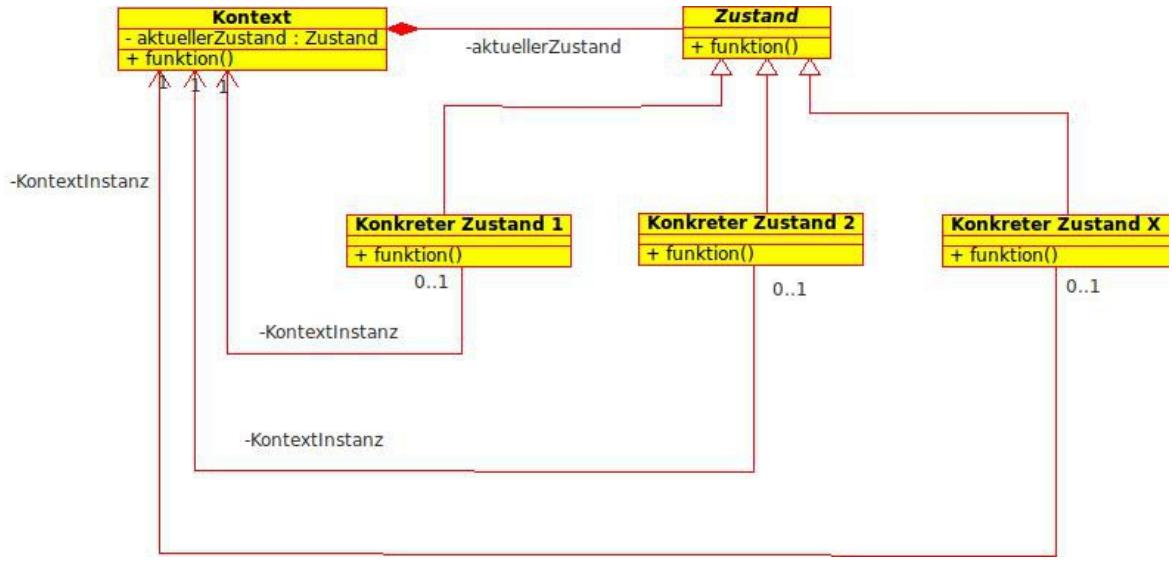


Abb. 11.3: Aufbau des Patterns mit Zustandswechsel vom Zustand bestimmt

Bei dieser Variante benötigen die Zustände einen Verweis auf das Kontextobjekt, weil in diesem der aktuelle Zustand gespeichert wird und die Zustände den aktuellen Zustand ändern können müssen. Der Zustandswechsel wird nun im jeweiligen konkreten Zustand vorgenommen. Diese Variante lässt sich leicht erweitern und bietet den Vorteil, das die Zustandsübergänge in den Zuständen gekapselt sind. Diese Variation bietet sich für die meisten Fälle an.

11.4.3 Erstellen der Zustände

Bei Bedarf erzeugen und danach löschen

Bei dieser Ausführung wird die Instanz des konkreten Zustands beim Übergang in den entsprechenden Zustand erzeugt. Wird der Zustand dann wieder verlassen, muss die Instanz gelöscht werden um einen Speicherüberlauf zu vermeiden. Diese Variante lässt sich relativ einfach Implementieren und sollte verwendet werden, wenn Zustandsübergänge nicht allzu häufig auftreten.

Alle Zustände im Kontext als Attribut

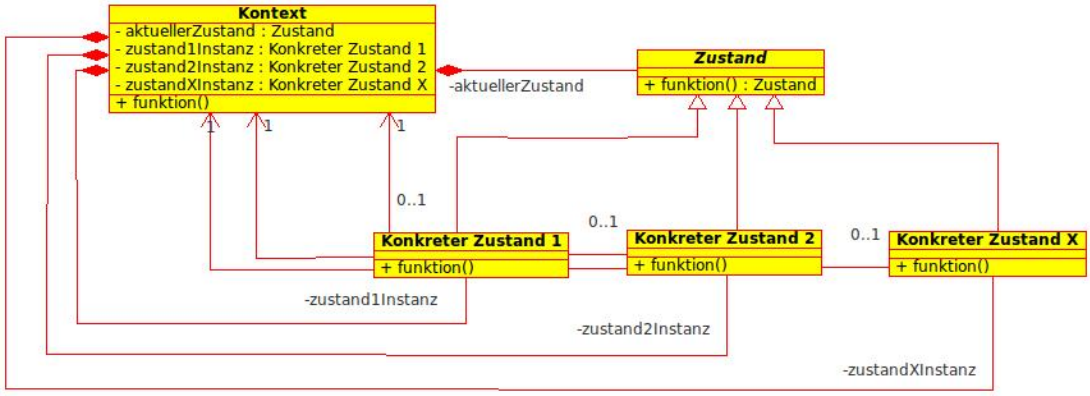


Abb. 11.4: Aufbau des Patterns mit Erhalt der Zustände

Bei dieser Ausführung werden beim Erzeugen des Kontextes auch alle konkreten Zustände erzeugt und als Attribut im Kontext gehalten. Diese Variation bietet sich an, wenn häufig Zustandsübergänge auftreten, weil somit nicht bei jedem Übergang ein neues Objekt erzeugt und das Alte gelöscht werden muss.

11.5 Beispiele

11.5.1 Übersicht

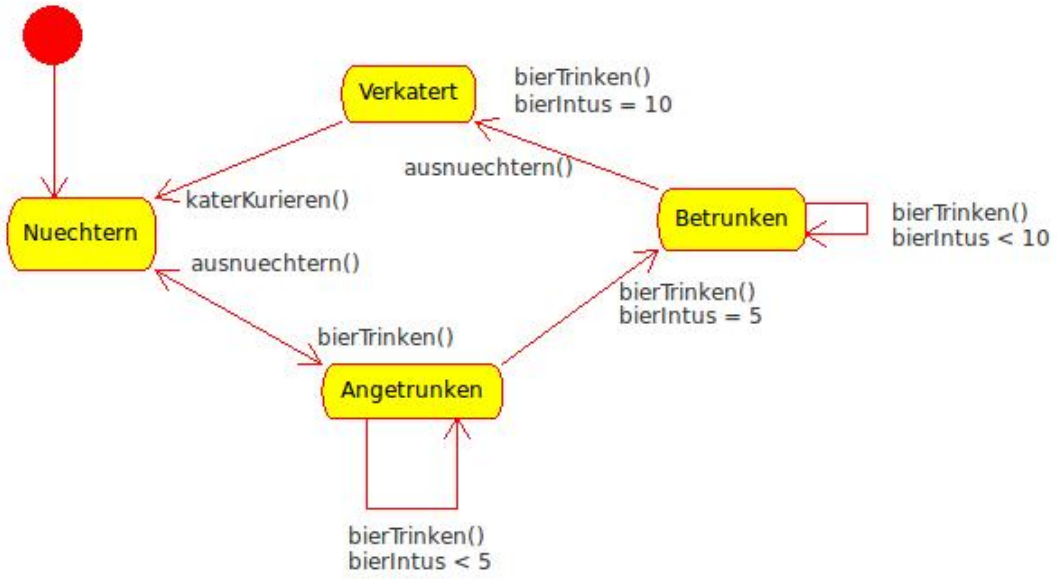


Abb. 11.5: Zustandsdiagramm des Studentenobjekts

Als Beispiel wollen wir einen Studenten mit einer seiner Freizeitbeschäftigungen Implementieren. Bei dieser Beschäftigung sind vier Zustände möglich. Der Initialzustand hierbei ist „Nuechtern“. Trinkt er nun ein Bier, wechselt der Zustand zu „Angetrunken“. Hört er mit dem Trinken auf und nüchtert sich aus, wechselt der Zustand wieder zu „Nuechtern“. Trinkt er jedoch weiter Bier, wechselt sein Zustand nach dem fünften Bier in den Zustand „Betrunken“. Von diesem Zustand aus kann jederzeit ein Zustandsübergang erwirkt werden, in dem er sich ausnüchtert. Allerdings wechselt der Zustand nun nicht mehr direkt auf „Nuechtern“, sondern auf „Verkatert“. Spätestens nach dem zehnten Bier, wird angenommen, das er kein weiteres Bier mehr verträgt und wechselt daher ebenfalls in den Zustand „Verkatert“. Von diesem Zustand aus, ist ein Zustandsübergang nur möglich, in dem er seinen Kater auskuriert. Nachdem er dies getan hat, wechselt sein Zustand wieder in „Nuechtern“.

11.5.2 Strukturierte Implementierung in Java

Zunächst wollen wir das Beispiel mittels der klassischen strukturierten Programmierung vorstellen. Um den Unterschied zur objektorientierten Programmierung zu veranschaulichen.

Abb. 11.6: Klassendiagramm des strukturierten Entwurfs

Wie das Klassendiagramm in Abbildung 11.6 zeigt, besteht die Implementierung aus lediglich einer relativ umfangreichen Klasse in der ein Aufzählungstyp erstellt und verwendet wird.

```
1  public class Student {
2          //Aufzaehlung der Zustaende als Enum
3          enum Zustaende
4          {
5                  nuechtern, angetrunken, betrunken, verkatert
6          }
7
8          //Erstellen einer Variable fuer den aktuellen Zustand
9          private Zustaende aktuellerZustand =Zustaende.nuechtern;
10
11         //Erstellen einer Zaehlvariablen fuer die Anzahl der getrunkenen Biere
12         private int bierIntus = 0;
13
14         //Konstruktor
15         public Student()
16         {
17         };
18
19         //Getter Methode fuer das Abfragen des aktuellen Zustandes
20         public String getAktuellerZustand()
21         {
22                 return aktuellerZustand.toString();
23         }
24
25         //Funktion bierTrinken() mit den entsprechenden Zustandsuebergaengen
26         public void bierTrinken()
27         {
28                 if(aktuellerZustand == Zustaende.nuechtern)
29                 {
30                         System.out.println("Prost!");
31                         bierIntus++;
32                         aktuellerZustand = Zustaende.angetrunken;
33                 }
34                 else if(aktuellerZustand == Zustaende.angetrunken)
35                 {
36                         System.out.println("Prost!");
37                         bierIntus++;
38                         if(bierIntus >=5)
39                         {
40                                 aktuellerZustand = Zustaende.betrunken;
41                         }
42                 }
43                 else if(aktuellerZustand == Zustaende.betrunken)
44                 {
45                         if(bierIntus>=10)
46                         {
47                                 System.out.println("So, jetzt ists aber genug!");
48                                 bierIntus = 0;
49                                 aktuellerZustand = Zustaende.verkatert;
50                         }
51                         else
52                         {
53                                 System.out.println("Prost!");
54                                 bierIntus++;
55                         }
56                 }
57                 else
58                 {
59                   System.out.println("keine Lust auf Bier!");
60                 }
61         }
```

```
62          //Funktion ausnuechtern() mit den entsprechenden Zustandsuebergaengen
63          public void ausnuechtern()
64          {
65                  if(aktuellerZustand == Zustaende.angetrunken)
66                  {
67                          System.out.println("Genug Bier fuer heute!");
68                          aktuellerZustand = Zustaende.nuechtern;
69                          bierIntus = 0;
70                  }
71                  else if(aktuellerZustand == Zustaende.betrunken)
72                  {
73                          System.out.println("Mir langts fuer heut!");
74                          aktuellerZustand = Zustaende.verkatert;
75                          bierIntus = 0;
76                  }
77                  else
78                  {
79                          System.out.println("Hab kein Alkohol intus!");
80                  }
81          }
82
83          //Funktion katerKurieren() mit den entsprechenden Zustandsuebergaengen
84          public void katerKurieren()
85          {
86                  if(aktuellerZustand == Zustaende.verkatert)
87                  {
88                          System.out.println("Lang schlafen und nen chilligen tag machen!");
89                          aktuellerZustand = Zustaende.nuechtern;
90                  }
91                  else
92                  {
93                          System.out.println("Hab kein Kater!");
94                  }
95          }
96  }
```

Abb. 11.7: Implementierung in strukturierter Programmierung

11.5.3 Objektorientierte Implementierung in Java

Nun wollen wir das selbe Beispiel mittels Objektorientierung und des State Design Patterns umsetzten.

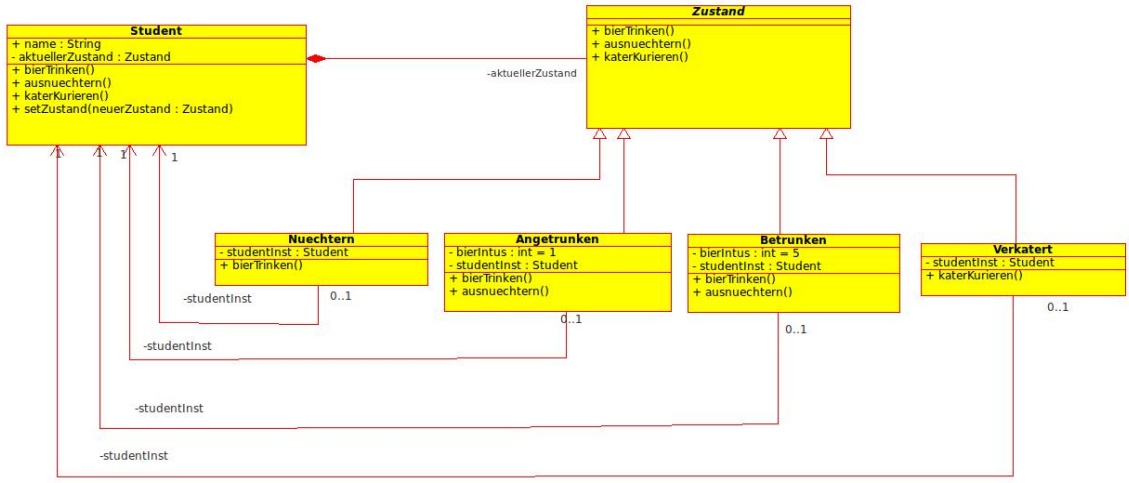


Abb. 11.8: Klassendiagramm des objektorientierten Entwurfes

Wie das Klassendiagramm in Abbildung 11.8 zeigt, werden die Zustände in die abstrakte Klasse „Zustand“ mit ihren Unterklassen ausgelagert. Die Klasse Zustand enthält die default Implementierungen der einzelnen Funktionen. Diese werden verwendet, wenn die erbenden Klassen diese Funktionen nicht selbst Implementieren. Grundsätzlich kann anstelle einer abstrakten Klasse hier auch ein Interface verwendet werden. Allerdings müssen dann alle Funktionen in den erbenden Klassen implementiert werden. Außerdem kann dem Klassendiagramm entnommen werden, dass wir die Variante gewählt haben, in der die Zustandsübergänge von den jeweiligen Zuständen vorgenommen werden. Daher benötigen die Zustände eine Referenz auf das Kontextobjekt, also in unserem Falle die Studenteninstanz.

```
1  public class Student {
2    private String name;
3    private Zustand aktuellerZustand;
4
5    //Konstruktor
6    public Student () {
7            aktuellerZustand = new Nuechtern(this);
8    };
9
10   //Setter Methode fuer name
11   public void setName ( String newVar ) {
12     name = newVar;
13   }
14
15   //Getter Methode fuer name
16   private String getName ( ) {
17     return name;
18   }
19
20   //Setter Methode fuer aktuellerZustand
21   public void setAktuellerZustand ( Zustand newVar ) {
22     aktuellerZustand = newVar;
23   }
24
25   //Getter Methode fuer aktuellerZustand
26   public Zustand getAktuellerZustand ( ) {
27     return aktuellerZustand;
28   }
29
30   //Funktion bierTrinken() ruft die Funktion des entsprechenden Zustandes auf
31   public void bierTrinken(  )
32   {
33           aktuellerZustand.bierTrinken();
34   }
35
36   //Funktion ausnuechtern() ruft die Funktion des entsprechenden Zustandes auf
37   public void ausnuechtern(  )
38   {
39           aktuellerZustand.ausnuechtern();
40   }
41
42   //Funktion katerKurieren() ruft die Funktion des entsprechenden Zustandes auf
43   public void katerKurieren(  )
44   {
45           aktuellerZustand.katerKurieren();
46   }
47 }
```

Abb. 11.9: Implementierung der Klasse Student in Objektorientierter Programmierung

```
1  abstract public class Zustand {
2
3    //Konstruktor
4    public Zustand () { };
5
6    //Default Implementierung der Funktion bierTrinken()
7    public void bierTrinken(  )
8    {
9            System.out.println("keine Lust auf Bier");
10   }
11
12   //Default Implementierung der Funktion ausnuechtern()
13   public void ausnuechtern(  )
14   {
15           System.out.println("Hab kein Alkohol intus");
16   }
17
18   //Default Implementierung der Funktion katerKurieren()
19   public void katerKurieren(  )
20   {
21           System.out.println("Hab keinen Kater");
22   }
23 }
```

Abb. 11.10: Implementierung der abstrakten Klasse Zustand

```
1  public class Nuechtern extends Zustand {
2
3    //Referenz auf das Kontextobjekt Student
4    private Student studentInst;
5
6    //Dem Konstruktor wird die Referenz auf das Kontextobjekt beim Instanzieren uebergeben
7    public Nuechtern (Student studentInst) {
8            this.studentInst = studentInst;
9    };
10
11   //Ueberschreiben der Default Funktion bierTrinken()
12   public void bierTrinken(  )
13   {
14           System.out.println("Prost");
15           //Zustandswechsel zu Angetrunken
16           this.studentInst.setAktuellerZustand(new Angetrunken(studentInst));
17   }
18 }
```

Abb. 11.11: Implementierung der Zustandsklasse Nuechtern

```
1  public class Angetrunken extends Zustand {
2
3    //Referenz auf das Kontextobjekt Student
4    private Student studentInst;
5
6    //Zaehlvariable in Zustand gekapselt
7    private int bierIntus = 1;
8
9    //Dem Konstruktor wird die Referenz auf das Kontextobjekt beim Instanzieren uebergeben
10   public Angetrunken (Student studentInst) {
11           this.studentInst = studentInst;
12   };
13
14   //Ueberschreiben der Default Funktion bierTrinken()
15   public void bierTrinken(  )
16   {
17         if(bierIntus < 5)
18         {
19                 System.out.println("Prost");
20                 bierIntus++;
21         }
22         else
23         {
24                 System.out.println("Jetzt werd ich langsam voll!");
25                 System.out.println("Prost!");
26                 //Zustandswechsel zu Betrunken
27                 studentInst.setAktuellerZustand(new Betrunken(studentInst));
28         }
29   }
30
31   //Ueberschreiben der Default Funktion ausnuechtern()
32   public void ausnuechtern(  )
33   {
34           System.out.println("Genug Bier fuer heut");
35           //Zustandswechsel zu Nuechtern
36           studentInst.setAktuellerZustand(new Nuechtern(studentInst));
37   }
38 }
```

Abb. 11.12: Implementierung der Zustandsklasse Angetrunken

```
1  public class Betrunken extends Zustand {
2
3    //Referenz auf das Kontextobjekt Student
4    private Student studentInst;
5
6    //Zaehlvariable in Zustand gekapselt
7    private int bierIntus = 5;
8
9    //Dem Konstruktor wird die Referenz auf das Kontextobjekt beim Instanzieren uebergeben
10   public Betrunken (Student studentInst) {
11           this.studentInst = studentInst;
12   };
13
14   //Ueberschreiben der Default Funktion bierTrinken()
15   public void bierTrinken(  )
16   {
17           if(bierIntus < 10)
18           {
19                   System.out.println("Prost");
20                   bierIntus++;
21           }
22           else
23           {
24                   System.out.println("So, jetzt genug");
25                   //Zustandswechsel zu Verkatert
26                   studentInst.setAktuellerZustand(new Verkatert(studentInst));
27           }
28   }
29
30   //Ueberschreiben der Default Funktion ausnuechtern()
31   public void ausnuechtern(  )
32   {
33           System.out.println("Mir langts");
34           //Zustandswechsel zu Verkatert
35           studentInst.setAktuellerZustand(new Verkatert(studentInst));
36   }
37 }
```

Abb. 11.13: Implementierung der Zustandsklasse Betrunken

```
1  public class Verkatert extends Zustand {
2
3    //Referenz auf das Kontextobjekt Student
4    private Student studentInst;
5
6    //Dem Konstruktor wird die Referenz auf das Kontextobjekt beim Instanzieren uebergeben
7    public Verkatert (Student studentInst) {
8            this.studentInst =studentInst;
9    };
10
11   //Ueberschreiben der Default Funktion katerKurieren()
12   public void katerKurieren(  )
13   {
14           System.out.println("Lang schlafen und nen chilligen Tag machen");
15           //Zustandswechsel zu Nuechtern
16           studentInst.setAktuellerZustand(new Nuechtern(studentInst));
17   }
18 }
```

Abb. 11.14: Implementierung der Zustandsklasse Verkatert

11.6 Anwendungsbeispiele

Dieses Design Pattern findet in der Praxis häufig Anwendung für die Implementierung von Zustandsmaschinen. Diese können bei der objektorientierten Abstrahierung realer Vorgänge auftreten. So zum Beispiel im Compilerbau oder bei der Implementierung von grafischen Oberflächen wie in Abbildung 11.15 zu sehen ist.[Hau11b][Erb09]

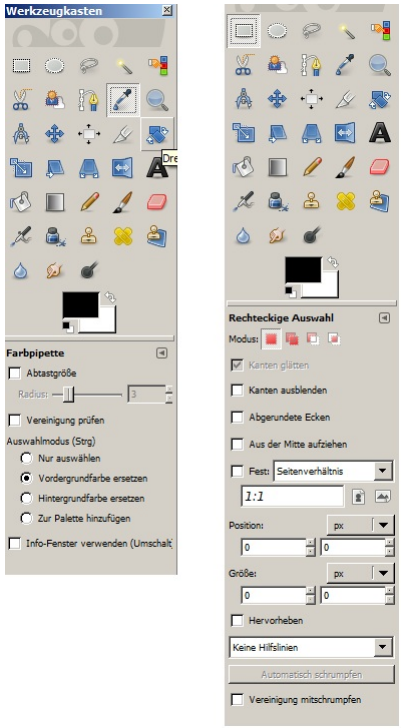


Abb. 11.15: Screenshots der Zustände beim Bildverarbeitungsprogramm GIMP

11.7 Bewertung

11.7.1 Hinweise

- Bei einfachen Anwendungen, die nicht erweitert werden sollen, kann es sinnvoll sein, den Ansatz der strukturierten Programmierung zu verwenden.
- Es empfiehlt sich zunächst immer ein Zustandsdiagramm zu erstellen.

11.7.2 Vorteile

- Kapselung: Funktionen können in die konkreten Zustände ausgelagert werden.
- Übersichtlichkeit der Klassen: Da die Funktionen in mehrere Klassen aufgesplittet werden, werden die Klassen kleiner und übersichtlicher.
- Erweiterbarkeit: Neue Zustände können einfach eingefügt werden.
- Sichere Zustandsübergänge: Die Schnittstellen für Zustandswechsel sind klar definiert und ein Zustandswechsel ist ein atomarer Befehl.

11.7.3 Nachteile

- Erhöhte Klassenanzahl: Durch das Auslagern von Zuständen muss für jeden Zustand eine eigene Klasse erzeugt werden, was die Übersichtlichkeit des Softwareentwurfes negativ beeinflusst.

- Implementierungsaufwand

12 Dependency Injection

12.1 Definition

Über das Thema Dependency Injection gibt es wenig einschlägige Standard-Literatur (eins der wenigen Bücher ist [Eil10]). Das Entwurfsmuster wurde 2004 zum ersten Mal (öffentlichkeitswirksam) von Martin Fowler in [Fow04] kommentiert. Folgende Definition ist leicht verständlich:

> „Dependency Injection (DI) ist ein Entwurfsmuster und dient in einem objektorientierten System dazu, die Abhängigkeiten zwischen Komponenten oder Objekten zu minimieren."
>
> Quelle: [wik11]

Martin Fowlers Definition geht hier etwas mehr ins Detail:

> „Dependency Injection (DI) ist eine Ausprägung von Inversion of Control und hilft, zwischen der Objekterzeugung und -initialisierung einerseits und regulärem Anwendungscode andererseits zu trennen."
>
> Nach: [Fow04]

Gefahren von Abhängigkeiten

Auf Code-Ebene gibt es eine Vielzahl von Abhängigkeiten. Ursachen sind Instanziierung von Objekten oder der Benutzung von Factories, aber auch das Aufrufen von statischen Methoden. Ebenso erzeugt das Implementieren von Schnittstellen oder Erweitern einer Klasse durch Vererbung Abhängigkeiten.

Abhängigkeiten stellen nicht grundsätzlich ein Problem dar, aber sie können die Architektur der Anwendung unnötig kompliziert machen und eine effiziente Testbarkeit erschweren. Als besonders gefährlich hat sich hier der Einsatz von Singletons erwiesen (vgl. [**?**]):

- Durch die Ermöglichung eines global-erreichbaren Service verhalten sich Singletons ähnlich problematisch wie globale Variablen: Es entstehen bei jeder Benutzung von dem Singleton Abhängigkeiten und der Programmfluss wird komplizierter – um diesen zu verstehen, muss der Singleton-Code analysiert werden.

- Das Single-Responsibility-Prinzip (siehe [**?**, S. 138ff.]) wird verletzt, da der reguläre Anwendungscode und der Code für die Limitierung der Objekterzeugung vermischt werden.

- Singletons fördern bzw. unterstützen die enge Kopplung im Code und erschweren so die Testbarkeit durch Unit-Tests.

- Die Testbarkeit wird außerdem beeinträchtigt, da Singletons ihren Zustand über die Programm-Lebensdauer persistieren – so können die einzelnen Testfälle und -klassen nicht unabhängig sein, was das Auffinden von Fehlern erschwert.

Inversion of Control

Das angesprochene Prinzip *Inversion of Control* steht für die Umkehrung des Kontrollflusses. Traditionell ist man als Entwickler bestrebt, den Kontrollfluss seiner Anwendung selbst zu leiten. Die Umkehrung dessen bedeutet, dass man „nur noch“ Bauteile mit passenden Schnittstellen entwickelt und das Zusammenfügen zu einem Ganzen und Ausführen einem unabhängigem Framework überlässt.

Im Sinne von DI verzichtet hierbei der Entwickler der Anwendung auf Objektinstanziierung und -initialisierung – beides ohnehin eher triviale Aufgaben – und überlässt dies dem DI-Framework. Der Vorteil darin ist, dass Abhängigkeiten zwischen Komponenten vermieden werden können und die Anwendung in von außen gezielt definierten Konfigurationen gestartet werden kann.

12.2 Motivation

12.2.1 Problematik der Factory

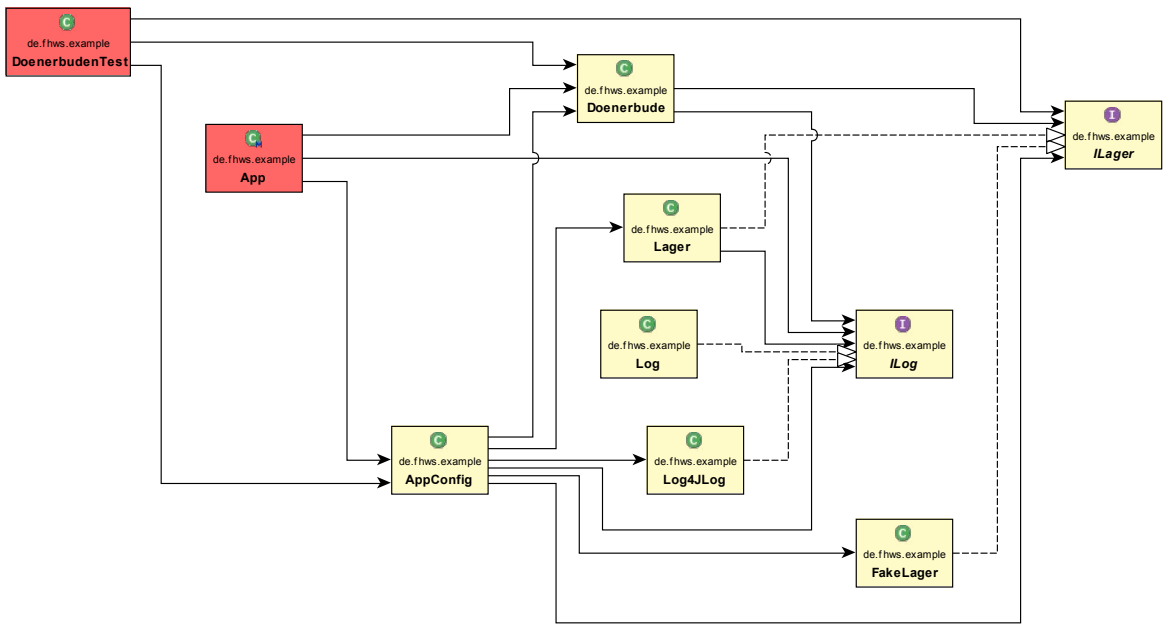


Abb. 12.1: Beispielanwendung ohne DI

Um die Wirkungsweise von DI zu illustrieren, folgt nun ein Beispiel. In Abbildung 12.1 ist eine traditionelle Anwendung zu sehen. Zur Vermeidung von Abhängigkeiten haben sich Interfaces bewährt. Nötig ist dann eine größere Factory, die sich um Instanziierung und Initialisierung der konkreten benötigten Klassen kümmert. So lassen sich direkte Zugriffe auf das `Lager` oder `Log4JLog` in anderen Klassen vermeiden.

Genau hier liegt das Problem: Wenn in der Anwendung Refactorings anstehen, ist die `AppConfig` auch häufig zu ändern – die Klasse sollte daher nie aus dem Auge gelassen werden. Die Klasse zieht Abhängigkeiten regelrecht an und sie wächst unaufhörlich. Um in der Praxis über solche Klassen die Kontrolle zu behalten, zerteilt man sie typischerweise in Aufgabenbereiche, wodurch der Überblick aber auch zusätzlich verloren gehen kann.

Die Testklasse `DoenerbudenTest` wurde hinzugefügt, um die Problematik der Factory zu verdeutlichen, da in dem Test eine spezielle `ILager`-Implementierung für den Test benötigt wird – das `FakeLager`. Die Entscheidung, zu welchem Zeitpunkt welche Implementierung von welchem Interface verwendet werden soll, trifft hier die Factory. Sie `AppConfig` weiß alles, sie kennt alle und kümmert sich um alles[1]. Dazu ist entweder viel duplizierter Code notwendig oder eine Factory mit einem ungewöhnlich hohen Anteil an Logik. Beides ist nicht ideal.

Die unabhängigen Klassen sind in Abbildung 12.1 rot eingefärbt, hier sind das die Klasse `App` mit der `main`-Methode und eine Testklasse, also die beiden „Startklassen“. Alle Abbildungen wurden mit dem Tool Class Dependency Analyzer (CDA)[2] erstellt.

12.2.2 DI als Factory-Ersatz

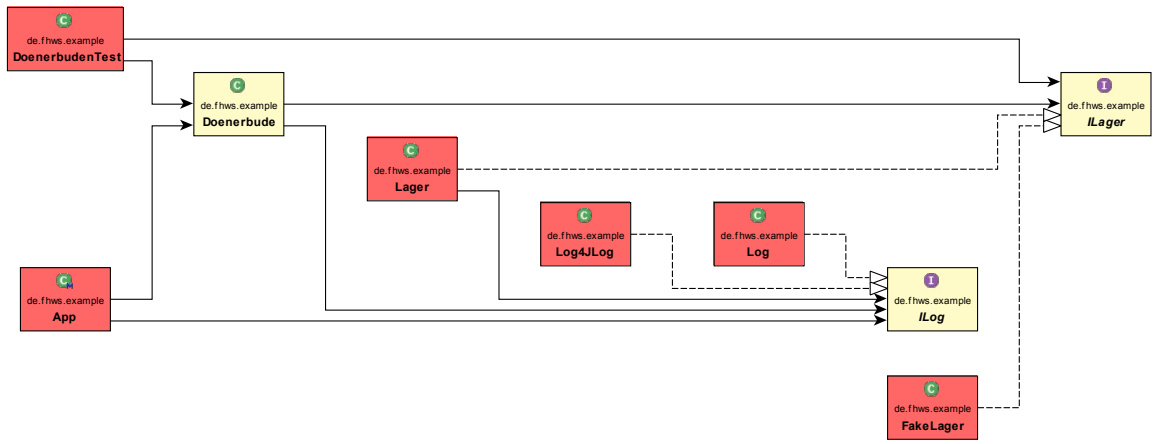


Abb. 12.2: Beispielanwendung mit DI

Die Abbildung 12.2 zeigt eine Anwendung, die durch verschiedene Refactorings aus Abbildung 12.1 hervorgegangen ist. Im Wesentlichen wurden diverse Annotations und ein DI-Framework hinzugefügt. Zwei Code-Zeilen sind zusätzlich in den Startklassen `App` und

[1] Vielleicht hätte man die Klasse lieber in `SuperNanny` umbenennen sollen?
[2] Siehe: http://www.dependency-analyzer.org/

`DoenerbudenTest` integriert worden. Die Konfiguration der Abhängigkeiten wurde in eine XML-Datei ausgelagert.

Auch hier wurden die unabhängigen Klassen rot eingefärbt. Im Grunde fehlt nur die Factory `AppConfig` mit ihren verursachten Abhängigkeiten. Dadurch werden alle `ILager`- und `ILogger`-Implementierungen unabhängig. Gemäß dem Prinzip *Inversion of Control* entscheidet eine unabhängige dritte Instanz anhand der XML-Konfiguration, welche Implementierung für welche Laufzeitkonfiguration infrage kommt.

12.3 Implementierung

Es gibt verschiedene DI-Implementierungen auf dem Markt. Allen DI-Frameworks ist gemein, dass sie sich bei der Annotation-Syntax (mehr oder weniger) auf die Java-Specification-Requests (JSR) #250[3] und #330[4] stützen. Die Diskussion der einzelnen Frameworks sprengt den Rahmen dieser Ausarbeitung, daher sei im Folgenden auf deren Webseiten verwiesen:

Oracle Enterprise JavaBeans http://www.oracle.com/technetwork/java/javaee/ejb/

Google Guice http://code.google.com/p/google-guice/

PicoContainer http://picocontainer.org/

Spring http://www.springsource.org/

Eclipse 4 http://www.eclipse.org/eclipse4/

12.4 Diskussion

Die Möglichkeiten von DI sind enorm. In Bezug auf die Verbesserung der Testbarkeit werden unnötige Abhängigkeiten verhindert und die Bereitstellung von Fake-Objekten ermöglicht. Ebenso lässt sich eine Anwendung in verschiedenen Versionen deployen oder von außen die Initialisierung bestimmter Klassen granular beeinflussen.

Trotzdem ist mit der Nutzung von DI auch einiger Mehraufwand verbunden. Es entstehen Abhängigkeiten zu einem DI-Framework. Vor allem die begrenzte Unterstützung in den integrierten Entwicklungsumgebungen macht es DI-Einsteigern schwer. Ebenso führt die geringe Aussagekraft von Fehler-Meldungen und das komplizierte Debuggen zu Hürden im Entwicklungsalltag.

Ob der Einsatz von DI sinnvoll ist, muss jeder Entwickler für sich und seine Anwendung selbst beantworten – wie auch vor dem Einsatz eines jeden anderen Design Pattern. Vor einem Einsatz in einem Produktivsystem sind der Aufwand und die Chancen von DI unbedingt zu diskutieren.

[3] Siehe: http://jcp.org/en/jsr/detail?id=250
[4] Siehe: http://jcp.org/en/jsr/detail?id=330

13 Visitor Pattern

13.1 Definition

Repräsentiert eine Operation, die auf Elemente einer Objektstruktur angewandt wird. Visitor erlaubt es neue Operationen zu definieren ohne die Klassen der Elemente auf denen die Operation operiert ändern zu müssen.

13.2 Motivation

Angenommen man hat einen Compiler der Code als Syntaxbäum behandelt. Der Compiler wird verschiedene Operationen auf die Elemente dieses Baumes ausführen müssen. Überprüfen ob alle Variablen definiert sind, Code generieren, Operationen zur Typüberprüfung, Codeoptimierung, Flussanalyse oder Überprüfung ob Variablen Werte zugewiesen wurden.

Die meisten Operationen werden Knoten, die Zuweisungen oder Knoten, die Variablen darstellen anders behandeln müssen. Man kann die Operationen in jeder einzelnen Klasse implementieren. Das würde das Programm aber schwer verständlich, schwer pflegbar und schwer änderbar machen.

Es wäre besser, wenn jede neue Operation separat hinzugefügt werden könnte und die Knotenklassen unabhängig von den Operationen wären, die auf ihnen operieren.

Dies erreicht man, indem man die verwandten Operationen jeder Klasse in einem separaten Objekt kapselt und dieses Objekt an die Knotenklassen weiterreicht. Dem Visitor. Wenn ein Element den Visitor akzeptiert, sendet es eine Anfrage an den Visitor. Es beinhaltet das Element als Argument. Der Visitor wird danach die Operation für das Element ausführen. Die Operation die eigentlich in dem Element selbst war.

Um es dem Visitor zu ermöglichen mehrere verschiedene Operationen auf Elemente ausführen zu können benötigt man eine abstrakte Elternklasse. Die abstrakte Elternklasse muss für jedes Element auf denen es operiert eine Operation bereitstellen.

Mit dem Visitor Pattern definiert man zwei Klassenhierarchien. Eine für die Elemente auf denen operiert wird. Und eine für die Visitors, die die Operationen implementieren. Eine neue Operation wird hinzugefügt, indem man eine neue Unterklasse zur Visitorhierarchie hinzufügt. Solange kein neuartiges Element zur Hierarchie hinzukommt, kann man so beliebig beliebig viele neue Operationen hinzufügen.

13.3 Anwendungsfälle

Wann ist das Pattern angebracht beziehungsweise nicht angebracht?

Man kann das Visitor Pattern benutzen wenn:

- Eine Objektstruktur viele Objekte unterschiedlicher Klassen beinhaltet. Und man Operationen auf die Objekte ausführen möchte, die von deren Klasse abhängig sind.
- Man mehrere verschiedene und nicht verwandte Operationen auf Objekte einer Struktur anwenden will und die Klassen selbst nicht mit mit den Implementierungen der Operationen Obeschmutzen möchte. Visitor erlaubt es verwandte Operationen in einer Klasse zu definieren. Wenn eine Objektstruktur über viele Applikationen verteilt ist benutze den Visitor nur in den Applikationen, in denen die Operationen des Visitors gebraucht werden.
- Wenn sich die Klassen der Objektstruktur selten ändern aber man oft neue Operationen für die Objekte definieren möchte. Das Verändern der Klassen der Objekte verlangt das ändern der Visitoren. Wenn sich die Klassen der Objektstruktur oft ändern ist es besser auf das Visitor Pattern zu verzichten und die Operationen innerhalb der Klassen selbst zu implementieren.

Wenn die Programmiersprache mehrfache Verteilung (multiple dispatch) unterstützt, so ist es gängige Meinung, dass man auf das Visitor Pattern verzichtet. Ein Ziel des Patterns ist es, in Programmiersprachen, die nur single dispatch unterstützen, multiple dispatch zu simulieren.

13.3.1 Single und multiple dispatch

In diesem Unterkapitel soll nur sehr kurz die Begriffe single dispatch und multiple dispatch dargestellt werden.

Von single dispatch spricht man, wenn ein Methodenaufruf nur von zwei Kriterien abhängt. Dem Namen der Methode die aufgerufen wird. Das Objekt welches die Methode enthält. Im Prinzip ein gewöhnlicher Methodenaufruf.

Von multiple dispatch spricht man, wenn mehr als zwei Kriterien für einen Methodenaufruf entscheidend sind. Der Name der Methode die aufgerufen wird. Das Objekt welches die methode enthält. Und zusätzlich die Parametertypen, die an die aufzurufende Methode übergeben werden. Angenommen man hat folgende Klassen und Beziehungen gegeben.

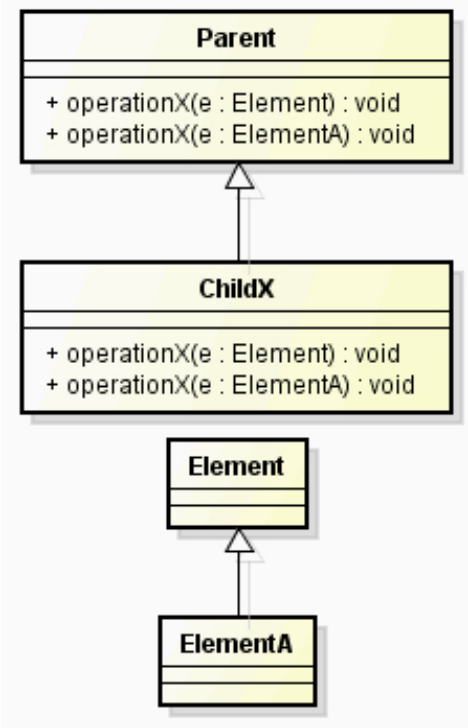


Abb. 13.1: Beispiel Diagramm für multiple dispatch.

Und möchte diesen Code ausführen.

```
1  public static void main(String[] args) {
2
3          Element[] elements = { new Element() , new ElementA() };
4          Parent p = new ChildX();
5
6          p.operationXY(elements[0]);
7          p.operationXY(elements[1]);
8  }
```

Abb. 13.2: Beispiel Code für multiple dispatch.

In Zeile 6 wird die Methode operationXY aufgerufen. Als Parameter wird ein Objekt vom Typ Element übergeben. An der Zeile ist nicht viel besonders. In Zeile 7 wird die Methode operationXY aufgerufen. Dieses mal wird als Parameter aber ein Objekt vom Typ ElementA übergeben. Es wird erwartet, dass die Methode operationXY(e : ElementA) vom Objekt ChildX aufgerufen wird. Dies ist aber nur bei Programmiersprachen der Fall, die multiple dispatch unterstützen (wie zum Beispiel Common Lisp, Perl 6, Groovy). Sprachen, die nur single dispatch unterstützen rufen in Zeile 7 die gleiche Methode auf, die auch in Zeile 6 aufgerufen wird. Und zwar operationXY(e : Element). Das liegt daran, dass zur Laufzeit nicht bestimmt wird, welcher Typ sich hinter elements[1] verbirgt. Es wird nur davon ausgegangen, dass dort ein Objekt vom Typ Element ist.

13.4 Struktur

In diesem Abschnitt soll kurz die Struktur und die Elemente des Visitor Patterns vorgestellt werden.

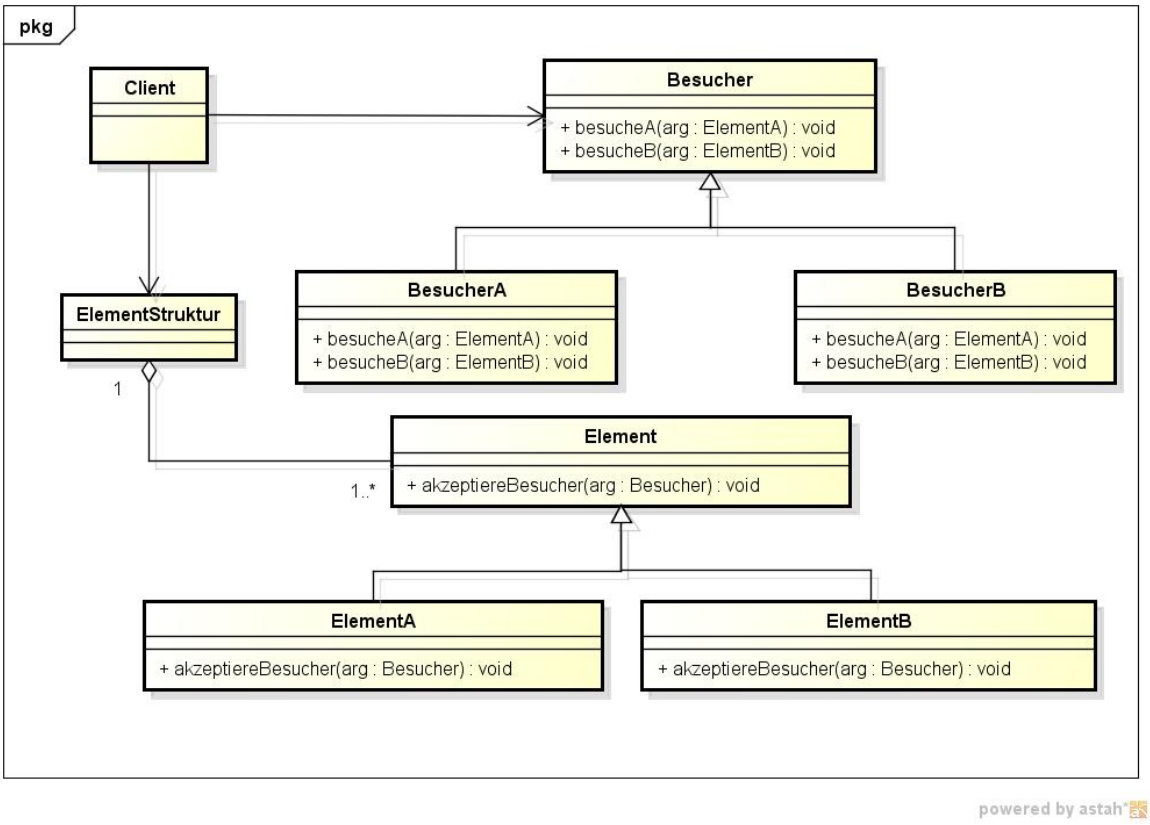


Abb. 13.3: Aufbau des Visitor Patterns.

Wie in der Abbildung zu sehen ist, hat das Visitor Pattern die abstrakte Klasse Besucher. Diese Klasse beinhaltet für jedes konkrete Element (ElementA, ElementB) eine abstrakte Methode, die als Parameter ein Element der Klasse übergeben bekommt, für welches es bestimmt ist. Jeder konkrete Besucher (BesucherA, BesucherB) erbt von der abstrakten Besucher Klasse und implementiert alle Methoden. Jede konkrete Besucher Klasse kapselt in seinen Methoden eine bestimmte Art von Operationen. Da es aber sein kann dass auf die konkreten Elemente zwar ähnliche Operationen ausgeführt werden, die aber nicht identisch sind, existiert für jedes konkrete Element eine Methode, in der das Element korrekt behandelt wird. Zum Beispiel gibt es bei Fluggesellschaften Prioritytickets und normale Flugtickets. Ein WarteschlangeVisitor zum Beispiel würde Methoden implementieren, die dafür sorgen, dass Kunden mit einem Priorityticket in der Warteschlange vor den Kunden mit normalen Flugtickets eingereiht werden. Dafür würde beispielsweise ein KassierVisitor von den Kunden mit Priorityticket mehr Geld verlangen. Das Beispiel ist vielleicht nicht besonders einfallsreich, zeigt aber hoffentlich was eine Absicht des Visitor Patterns ist. Alle Methoden die eine ähnliche Funktionalität haben, jedoch die konkreten Elemente unterschiedlich behandeln, werden in einer konkreten Besucher Klasse gesammelt. Element

ist die zweite abstrakte Klasse, die zum Visitor Pattern gehört. Sie beinhaltet auf jeden Fall die akzeptiereBesucher Methode, die alle konkreten Elemente implementieren müssen. Die akzeptiereBesucher Methode enthält nur eine Zeile.

```
1 arg.besucheElementA(this);
```

Abb. 13.4: Inhalt der akzeptiereBesucher Methode im ElementA.

In der Methode besucheElementA wird letztendlich die Operation auf das ElementA angewandt. Die konkreten Elemente befinden sich üblicherweise in einer ElementStruktur. Einem Baum zum Beispiel. Ob nun der Client, die ElementStruktur selbst oder die konkreten Visitor die Elemente der Struktur iterieren bleibt dem Entwickler überlassen. Da das Visitor Pattern fast immer mit einer Objekt Struktur gebraucht wird, ist es üblich dieses Pattern mit dem Iterator Pattern, welches die Elemente iteriert, zu benutzen.

13.5 Beispiel

In diesem Abschnitt soll an einem Beispiel das Visitor Pattern veranschaulicht werden. Dazu zeigt das nachfolgende Klassendiagramm die Struktur.

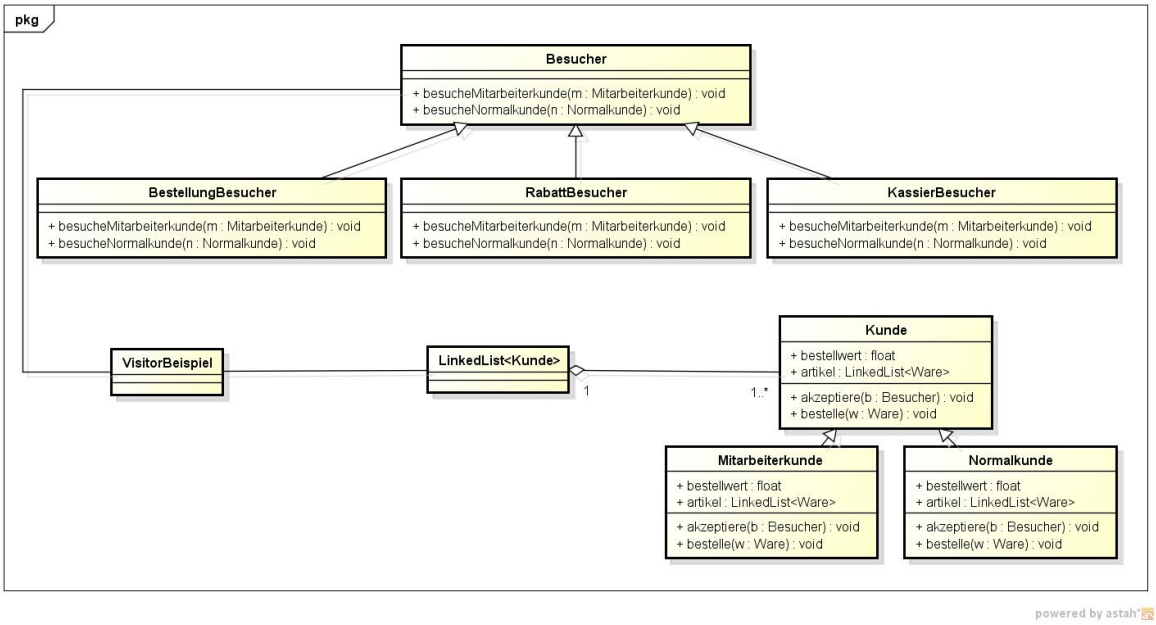


Abb. 13.5: Aufbau des Visitor Patterns.

Beginnen wir bei den Klassen, auf die sich die Operationen beziehen. Es gibt die abstrakte Klasse Kunde und die davon ableitenden Klassen Mitarbeiterkunde und Normalkunde. Diese Kunden können sich Waren bestellen, indem die Methode bestelle(w : Ware) aufgerufen wird. Diese Methode fügt der Liste der Klasse ein neues Listenelement hinzu. Nachdem der Kunde mit seinen Bestellungen fertig ist, kann er die Bestellung abschicken. Nun wollen

wir, dass die Liste des Kunden mit den Bestellungen verarbeitet werden soll. Zuerst soll der Gesamtbetrag der Bestellung ermittelt werden und in die Variable Bestellwert des Kunden geschrieben werden. Im zweiten Schritt soll dem Kunden unter Umständen ein Rabatt gegeben werden. Hier soll ein Teil der Variable Bestellwert abgezogen werden. Im letzten Schritt soll Bestellwert abkassiert werden, und dann auf 0 gesetzt werden. Das heiśst der Kunde hat seine Rechnung bezahlt.

Wir haben also drei unterschiedliche Operationen (Bestellung entgegennehmen, Rabatt geben, kassieren). Anstatt die Operationen in den Klassen der Kunden zu implementieren wird für jede Operation eine eigene Klasse angelegt. Die sogenannten Besucher. Bei diesem Pattern wird eine abstrakte Oberklasse angelegt. In dem Beispiel die Klasse Besucher. Da wir drei unterschiedliche Operationen haben, implementieren wir auch drei Unterklassen, die von Besucher erben. In dem Beispiel sind es die Klassen BestellungBesucher, RabattBesucher und KassierBesucher. Diese konkreten Besucher implementieren für jedes Element (Mitarbeiterkunde und Normalkunde) eine Methode, die die Operation auf das jeweilige Element ausführt. Die Klasse RabattBesucher implementiert die zwei Methoden besucheMitarbeiterkunde(m : Mitarbeiterkunde) und besucheNormalkunde(n : Normalkunde). In der ersten wird dem Mitarbeiterkunden zum Beispiel 10% Rabatt gegeben. In der zweiten wird dem Normalkunden zum Beispiel ab einem bestellwert von über 200 5% Rabatt gegeben. Ein Besucher steht also für eine Operationenart. Zum Beispiel Rabatt geben.

Nachfolgend kommt der Code der Methode main aus der Klasse VisitorBeispiel.

```
1  public static void main(String [] args)
2  {
3          LinkedList<Kunde> kundenListe = new LinkedList<Kunde>();
4
5          Mitarbeiterkunde mKunde = new Mitarbeiterkunde("Meier");
6          mKunde.bestelle(new Telefon());
7          mKunde.bestelle(new Kartoffeln());
8          kundenListe.add(mKunde);
9
10         Mitarbeiterkunde mKunde2 = new Mitarbeiterkunde("Max");
11         mKunde2.bestelle(new Computer());
12         mKunde2.bestelle(new Computer());
13         kundenListe.add(mKunde2);
14
15         Normalkunde nKunde = new Normalkunde("Norman");
16         nKunde.bestelle(new Computer());
17         nKunde.bestelle(new Telefon());
18         kundenListe.add(nKunde);
19
20         for(Kunde k : kundenListe){
21                 k.akzeptiere(new BestellungBesucher());
22                 k.akzeptiere(new RabattBesucher());
23                 k.akzeptiere(new KassierBesucher());
24         }
25 }
```

Abb. 13.6: Code der Methode main() aus der Klasse VisitorBeispiel.

Hier werden nacheinander drei Kunden mit ihren Bestellungen zur Liste hinzugefügt. In der for Schleife wird zum Schluss die Bestellung der Kunden verarbeitet. Wir sehen uns jetzt einen Schleifendurchlauf für das erste Element an. Das erste Element der Liste ist ein Objekt vom Typ Mitarbeiterkunde. In der for Schleife rufen wir drei mal hintereinander die Methode akzeptiere des Mitarbeiterkunden auf und übergeben hintereinander Objekte von

den Typen BestellungBesucher, RabattBesucher und Kassierbesucher. Was in der akzeptiere Methode passiert ist immer gleich. Und zwar der Aufruf.

```
1 b.besucheMitarbeiterkunde(this);
```

Abb. 13.7: Inhalt der akzeptiere() Methode der Klasse Mitarbeiterkunde.

Was den Unterschied bei den drei Aufrufen der akzeptiere Methode ist, sind die übergebenen Parameter. Beim ersten Aufruf aus der for Schleife wird ein BestellungBesucher übergeben. Das heiſst, dass in der akzeptiere Methode des MitarbeiterKunden die Methode besucheMitarbeiterkunde des BestellungBesucher aufgerufen wird. Beim zweiten Aufruf der akzeptiere Methode wird ein RabattBesucher übergeben. Das bedeutet, dass in der akzeptiere Methode die Methode besucheMitarbeiterkunde aufgerufen wird. Dieses mal aber ist es die Methode des RabattBesuchers. Auf diese Weise werden hintereinander drei unterschiedliche Operationen auf die Elemente von LinkedList<Kunde> kundenListe angewandt. Bei dem Normalkunden aus unserer Liste passiert fast das gleiche. Der einzige Unterschied ist die akzeptiere Methode des Normalkunden.

```
1 b.besucheNormalkunde(this);
```

Abb. 13.8: Inhalt der akzeptiere() Methode der Klasse Normalkunde.

13.6 Fazit

Das Visitor Pattern hilft dabei eine Struktur in ein Programm zu bringen. Methoden/Operationen, die miteinander verwandt sind, werden in einem konkreten Visitor gesammelt und Methoden/Operationen, die sich fremd sind, werden in verschiedenen Visitoren gesammelt. Dadurch werden Änderungen an Operationen leichter, weil sie alle in den dazugehörigen Klassen, den Visitoren, gesammelt sind. Man muss die Operationen nicht mehr in den Klassen der Elemente suchen und ersetzen sondern man muss nur eine speziele Klasse, den konkreten Visitor, ändern. Auſserdem werden die Klassen der Elemente auch übersichtlicher, weil sie nicht mehr die Methoden enthalten, die die Operationen ausführen.

14 Proxy Pattern

Vorabversion ohne Literaturverweise und Layoutkorrektur! Vorabversion ohne Literaturverweise und Layoutkorrektur! Vorabversion ohne Literaturverweise und Layoutkorrektur!

14.1 Absicht

Das Proxy-Muster, auch Stellvertreter-Muster, zählt zu den objektbasierten Strukturmustern. Es kontrolliert den Zugriff auf ein Objekt durch ein vorgelagertes Stellvertreterobjekt.

> ”Provide a surrogate or placeholder for another object to control access to it.” [Gam94]

14.2 Problem

Die Erzeugung und Initialisierung eines Objektes ist manchmal aufwendig, weil es entweder große Datenmengen (z.B. Bilddaten) enthält, sich in einem andern Adressraum befindet (Netzwerk, verteilte DB), oder eine sehr große Anzahl von Objekten, bei der Initialisierung, instanziiert (komplexe Anwendung).

14.3 Lösung

Die Objektinitialisierung wird bis zum eigentlichen Gebrauch durch den Verwender (Klienten) hinausgezögert. Der Verwender erhält, anstelle des realen Objekts, einen Proxy, der als Schnittstelle zu den Operationen des Originals fungiert. Der Proxy repräsentiert sich gegenüber dem Klienten als das wirkliche Objekt.

In Abbildung 14.1 wird der allgemeine Aufbau des Proxy Pattern, anhand eines Objektdiagramms dargestellt.

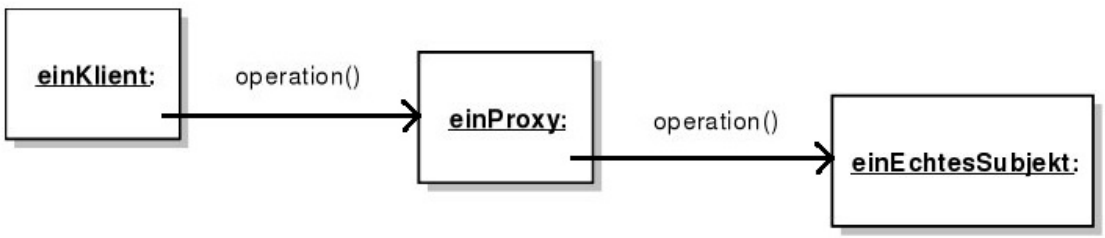


Abb. 14.1: Objektdiagramm eines Proxy-Musters

14.4 Struktur

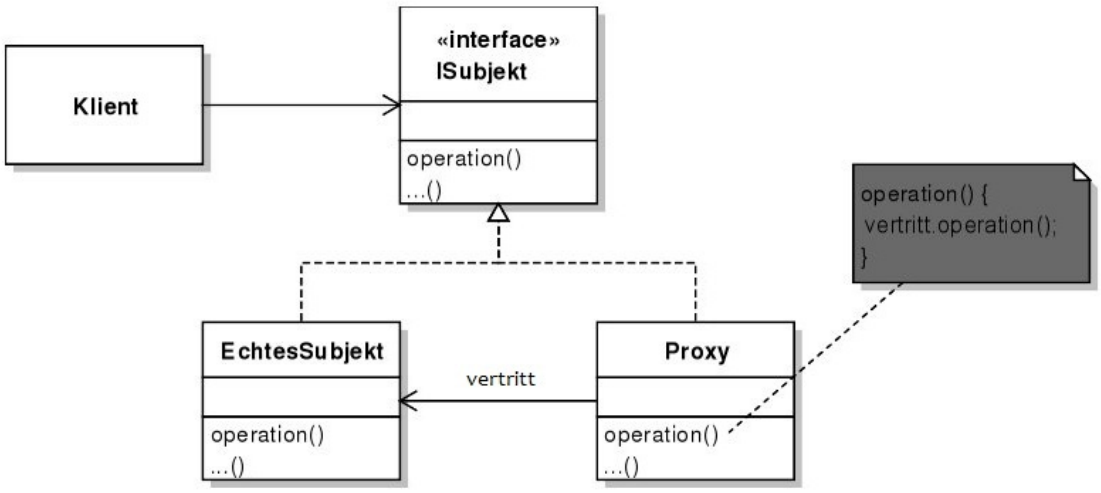


Abb. 14.2: Allgemeingültiges Klassendiagramm des Proxy Pattern

Nachfolgend wird der Zusammenhang des in Abbildung 14.2 aufgezeigten Klassendiagramms und dessen Klassen beschrieben.

Klient: Der `Klient` ist mit dem `Proxy` direkt verbunden, wobei der `Proxy` das Objekt `EchtesSubjekt` repräsentiert und dem `Klient`-Objekt vortäuscht mit dem Objekt `EchtesSubjekt` verbunden zu sein.

ISubjekt: Das Interface `ISubjekt` stellt die gemeinsame Schnittstelle dar, d.h. der `Proxy` kann auch überall dort verwendet werden, wo auch ein `EchtesSubjekt`-Objekt gebraucht wird. Durch das Interface `ISubjekt` wird sichergestellt, dass sowohl das `Proxy`- als auch das `EchtesSubjekt`-Objekt gleich aussehen. Die `Proxy`-Instanz enthält meistens nur die Operationen des `EchtesSubjekt`-Objekts, wobei dessen Attribute geschützt bleiben.

Proxy: Der `Proxy` vermittelt zwischen dem `Klient`- und dem `EchtesSubjekt`-Objekt. Weiterhin enthält das `Proxy`-Objekt eine Referenz/ Zeiger auf das `EchtesSubjekt`-Objekt. Für die Erzeugung (und evtl. Zerstörung) einer `EchtesSubjekt`-Instanz ist der `Proxy` verantwortlich.

EchtesSubjekt: Ein Objekt der Klasse `EchtesSubjekt` enthält die Daten und den Programmcode.

14.5 Ausprägungen und Implementierung

Das Proxy Pattern kann bei vielen Problemstellungen verwendet werden. Es gibt mehrere verschiedene Ausprägungen des Proxy Pattern, die in diesem Kapitel behandelt und in Java implementiert werden.

14.5.1 Virtual Proxy

Um das Erzeugen teurer Objekte zu kontrollieren wird ein virtual Proxy eingesetzt. Teure Objekte, sind Objekte die bei ihrem Erzeugung oder beim Laden, sehr viele Ressourcen beanspruchen. Eine Datei aus dem Netzwerk oder von der Festplatte kann so ein Objekt sein. Der Proxy ist dann für den Ladenvorgang des Objekts verantwortlich. Der Proxy entscheidet, ob das Objekt zu einem Zeitpunkt geladen wird. Beispielsweise muss bei einem Textdokument, dass Grafiken einbetten kann, nicht immer der gesamte Inhalt, zu jedem Zeitpunkt sichtbar sein. Eine eingebettet Grafik wird z.B. erst dann geladen, wenn sie sich im sichtbaren Bereich des Benutzers befindet. Jedoch kann es zu Fehlern kommen, wenn die Grafik erst beim direkten Betrachten geladen wird. Der virtuelle Proxy fungiert bis zu diesem Zeitpunkt als Stellvertreter für die Grafik im Textdokument und stellt z.B. Informationen über den benötigten Platz im Dokument zur Verfügung, um z.B. Layout-Fehler auszuschließen. Hierdurch entsteht ein Performance-Gewinn.

Implementierung in Java

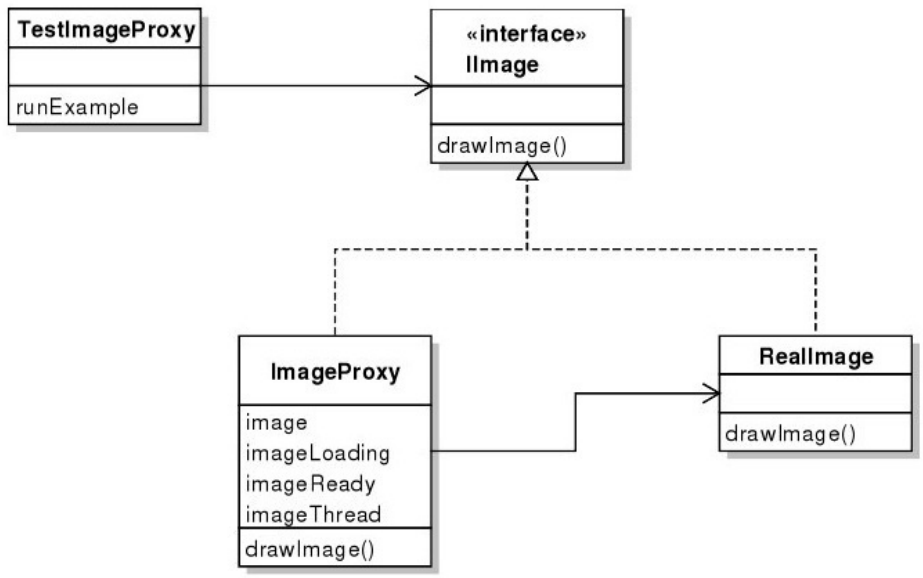


Abb. 14.3: Klassendiagramm eines Virtual Proxy

Das Klassendiagramm in Abbildung 14.3 zeigt den Aufbau eines einfachen virtual Proxy, der eine Bilddatei anzeigt.

Die Klassen `ImageProxy` (unser virtueller Proxy) und `RealImage` (unser echtes Subjekt) implementieren beide das Interface `IImage` und somit die Funktion `drawImage()`, die von der Klient-Klasse `TestImageProxy` auf der `ImageProxy`-Instanz aufgerufen wird und von dort aus, an das echte Subjekt `RealImage` weitergeleitet wird, um ein Bild auf den Bildschirm zu zeichnen.
Als erstes wird in Java das Interface `IImage` mit der Methode `drawImage()` erstellt.

```
1  package de.virtualProxy;
2
3  public interface IImage {
4          public void drawImage();
5  }
```

Nun wird die Klasse `RealImage`, die das Interface `IImage` implementiert, und die Methode `drawImage()` erstellt, die die Statusmeldungen ausgibt.

```
1  package de.virtualProxy;
2
3  public class RealImage implements IImage {
4
5          public RealImage(){ }
6
7          public void drawImage() {
8
9                  //Statusmeldung des echten Subjekts
10                 System.out.println("Hier ist das Objekt!\n");
11                 System.out.println("<<Das_Bild>>");
12         }
13 }
```

Als nächstes wird die Klasse `ImageProxy` erstellt die ebenfalls das Interface `IImage` implementiert. Sie instantiiert ein `RealImage`-Objekt und ruft dessen Methode `drawImage()` auf.

```
1  package de.virtualProxy;
2
3  //Implementiert das Interfaces IImage
4  public class ImageProxy implements IImage {
5
6          //erzeugt das RealImage Objekt (echtes Subjekt)
7          RealImage image = new RealImage();
8
9          Thread imageThread;
10
11         boolean imageLoading=false;
12         boolean imageReady=false;
13
14         public ImageProxy(){ }
15
16         public void drawImage() {
17
18                 while (!imageReady) {
```

```
                        //Statusmeldung des Proxy
                        System.out.println("Hier ist der Proxy!
                                        Bild wird geladen. Bitte warten...");

                        if (!imageLoading) {
                                imageLoading = true;
                                imageThread = new Thread(new Runnable() {
                                        public void run() {
                                                try {
                                                        //Verzoegerung um das
                                                        //Laden des Bildes zu emulieren
                                                        imageThread.sleep(0 , 2);
                                                        imageReady = true;

                                                        //Aufruf der Methode drawImage()
                                                        //auf dem RealImage Objekt
                                                        image.drawImage();
                                                } catch (Exception ex) {
                                                        ex.printStackTrace();
                                                }
                                        }
                                });
                                imageThread.start();
                        }
                }
        }
}
```

Zum Schluss noch die Test-Klasse.

```
package de.virtualProxy;

public class TestImageProxy {

        public TestImageProxy(){ }

        public void runExample() {
                ImageProxy iProxy = new ImageProxy();
                iProxy.drawImage();
        }

}
```

Was passiert hier? Die Test-Klasse ruft den Proxy auf, der Proxy ruft das echte Subjekt auf, das Bild wird geladen und über den Proxy an die Test-Klasse zurück gegeben.

14.5.2 Protection Proxy

Ein Protection Proxy wird eingesetzt um den Zugriff auf sein Orginalobjekt zu kontrollieren. Dem Orginalobjekt können somit unterschiedliche Zugriffsrechte hizugefügt werden. Die Aufgabe des Proxy ist es, die Zugriffsrechte des Aufrufers zu überprüfen und abzuhandeln. Der Proxy muss eine Referenz auf das Orginalobjekt halten. Mehrere Klienten nutzen meistens ein und denselben Proxy, man spricht hier auch von einer 1 zu n Beziehung.

InvocationHandler

In Java gibt es die Möglichkeit sich einen Proxy dynamisch erstellen zu lassen mittels dem Paket `java.lang.reflect`. Der Proxy wird zur Laufzeit erzeugt und implementiert ein

oder mehrere Interfaces, um Methodenaufrufe an eine festgelegte Klasse weiterzuleiten. Da der Proxy von Java automatisch erstellt wird gibt es keine Möglichkeit, der Proxy-Klasse, weitere Methoden hizuzufügen. Die Zugriffskontrolle kann also nicht in der Proxy Klasse implementiert werden. Die Lösung ist ein `InvocationHandler`, dessen Aufgabe darin besteht alle Methodenaufrufe auf dem Proxy zubeantworten. Der `InvocationHandler` implementiert also das Verhalten des Proxy, er liegt als Interface vor und bringt die Methode `invoke()` mit sich.
Funktionsweise eines InvocationHandlers

```
1  //Methodenaufruf auf dem Proxy
2  proxy.eineMethode(Argument);
3
4  //Proxy ruft dann Methode invoke() seines InvocationHandlers auf
5  //der Proxy, die Methode und die Argumente werden uebergeben
6  invoke(Object proxy, Method method, Object[] args)
7
8  //InvocationHandle entscheidet was getan werden soll mit der Methode, Zugriff ja/nein
9  //Aufruf der urspruenglichen Methode(eineMethode(Argument))
10 // auf dem Orginalobjekt
11 return method.invoke(Orginalobjekt, Argument)
```

Implementierung in Java

Die Abbildung 14.4 zeigt das Klassendiagramm eines Programms mit einem Protection Proxy. Das Programm verwaltet zwei Mitarbeiter in einer Datenbank. Mittels des Protection Proxy können diese ihren eigenen Namen ändern, aber nicht den des anderen, den Rang des anderen ändern, aber nicht ihren eigenen.

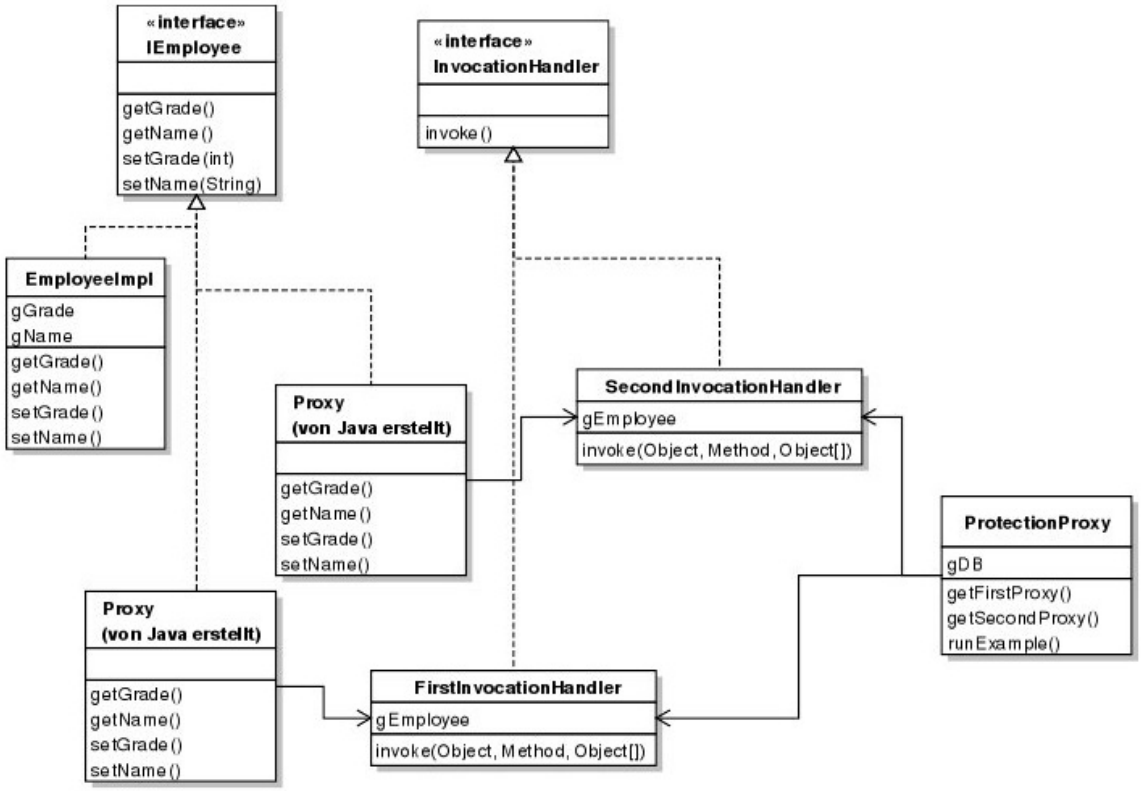


Abb. 14.4: Klassendiagramm eines Protection Proxy

```java
package de.protectionProxy;


import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;


//impelementiert den InvocationHandler
public class FirstInvocationHandler implements InvocationHandler {

	IEmployee gEmployee;

	//Referenz auf das echte Subjekt wird uebergeben
	//und eine Referenz darauf gehalten
	public FirstInvocationHandler(IEmployee gEmployee) {
		this.gEmployee = gEmployee;
	}

	//wird jedesmal aufgerufen wenn eine
	//Methode auf dem Proxy aufgerufen wird
	public Object invoke(Object aProxy, Method aMethod, Object[] aArgs)
			throws IllegalAccessException {

		try {
			if (aMethod.getName().startsWith("get")) {

				//Aufruf der Methode des echten
				//Subjekts mit urspruenglichen Argumenten
				//Zugriff gewaehrt
				return aMethod.invoke(gEmployee, aArgs);
			} else if (aMethod.getName().equals("setGrade")) {

				//Zugriff verweigert
				throw new IllegalAccessException();
			} else if (aMethod.getName().startsWith("set")) {
				return aMethod.invoke(gEmployee, aArgs);
			}
		} catch (InvocationTargetException aEx) {
			aEx.printStackTrace();
		}
		return null;
	}

}
```

```
1  package de.protectionProxy;
2
3  import java.lang.reflect.*;
4
5  public class ProtectionProxy {
6          //anlegen der DB
7          EmployeeDB gDB = new EmployeeDB();
8          public ProtectionProxy() {
9                  //Anlegen der Mitarbeiter
10                 IEmployee aEmp = new EmployeeImpl();
11                 aEmp.setName("Mueller");
12                 aEmp.setGrade(1);
13                 IEmployee bEmp = new EmployeeImpl();
14                 bEmp.setName("Maier");
15                 bEmp.setGrade(2);
16                 gDB.addEmployee(aEmp);
17                 gDB.addEmployee(bEmp);
18         }
19
20         public void runExample() {
21                 //ersten Mitarbeiter holen
22                 IEmployee aFirstEmp = gDB.getEmployee("Mueller");
23
24                 //der erste Proxy wird hier erzeugt
25                 IEmployee aFirstEmpProxy = getFirstProxy(aFirstEmp);
26
27                 //Aufruf der Methode auf dem Proxy
28                 aFirstEmpProxy.setName("Mueller-Maier");
```

```
1
2                  System.out.println("Neuer Name von Mueller: "+aFirstEmp.getName());
3                  try {
4
5                          //Aufruf der Methode auf dem Proxy
6                          aFirstEmpProxy.setGrade(10);
7
8                  } catch (Exception ex) {
9                          System.out.println(aFirstEmp.getName()+" Sie
10                                         koennen ihren Rang nicht selbst setzen. "+
11                                 "Ihr Rang ist: "+aFirstEmp.getGrade());
12                 }
13
14                 IEmployee aSecondEmp = gDB.getEmployee("Maier");
15
16                 //zweiter Proxy wird erzeugt
17                 IEmployee aSecondEmpProxy = getSecondProxy(aFirstEmp);
18                 aSecondEmpProxy.setGrade(5);
19                 System.out.println(aFirstEmp.getName()+", Ihr Rang wurde von: "
20                                 + aSecondEmp.getName()+" geaendert"
21                                 +". Ihr Rang ist jetzt: "+aFirstEmp.getGrade());
22                 try {
23
24                         //Aufruf der Methode auf dem Proxy
25                         aSecondEmpProxy.setName("Mueller-Maier_geaendert_von_Maier");
26                 } catch (Exception ex) {
27                         System.out.println("Nur Herr "+aFirstEmp.getName()+" kann
28                                         seinen Namen selbst aendern!");
29                 }
30         }
```

```
1         //ersten Proxy erstellen
2         IEmployee getFirstProxy(IEmployee aEmployee) {
3
4                                 //neuen Proxy erstellen
5                 return (IEmployee) Proxy.newProxyInstance(
6
7                                 //Class-loader um die Proxy-Klasse zu definieren
8                                 aEmployee.getClass().getClassLoader(),
9
10                                //Liste der Interfaces die der Proxy implementieren muss
11                                aEmployee.getClass().getInterfaces(),
12
13                                //ersten InvocationHandler erstellen
14                                new FirstInvocationHandler(aEmployee));
15        }
16        //zweiten Proxy erstellen
17        IEmployee getSecondProxy(IEmployee bEmployee) {
18
19                                //neuen Proxy erstellen
20                return (IEmployee) Proxy.newProxyInstance(
21                                bEmployee.getClass().getClassLoader(),
22                                bEmployee.getClass().getInterfaces(),
23
24                                //ersten InvocationHandler erstellen
25                                new SecondInvocationHandler(bEmployee));
26        }
27  }
```

Der vollständige Quellcode zum Protection Proxy kann in einer separaten Archive eingesehen werden.

14.5.3 Remote-Proxy

Ein Remote Proxy ist ein lokaler Stellvertreter für ein Objekt, das sich in einem anderen Adressraum befindet. Der Proxy hält stets eine Referenz auf sein zu vertretendes Objekt. Ein Beispiel dafür ist eine Client-Server-Architektur, wobei der Proxy im Client implementiert ist und das Originalobjekt im Server. Das Originalobjekt stellt den Service im Server dar, auf den der Proxy vom Client aus zugreift. Ist der Remote Proxy in Java implementiert befinden sich Client und Server meist in zwei unterschiedlichen, voneinander getrennten, virtuellen Maschinen, wobei es hier Unterschiede gibt.

- Client und Server befinden sich auf demselben Rechner im selben Prozess: d.h. ein Remote Proxy ist nicht nötig, da der der Client ohne Proxy auf das Originalobjekt zugreifen kann,

- Client und Server befinden sich auf demselben Rechner aber in unterschiedlichen Prozessen: d.h. der Proxy benötigt eine Referenz auf das Originalobjekt im anderen Adressraum.

- Client und Server befinden sich auf unterschiedlichen Rechnern in unterschiedlichen Prozessen: d.h. der Proxy benötigt eine Referenz auf das Originalobjekt im anderen Adressraum und eine Referenz auf den anderen Rechner

Problem: Wie greift man auf ein Objekt in einer anderen JVM zu?

Remote Method Invocation

In Java wird der Remote Proxy mittels der Remote Method Invocation(Java RMI) aus dem Paket `java.rmi.*` implementiert. Java RMI stellt Mechanismen zur verteilten Anwendungsprogrammierung bereit. Mittels RMI können Objekte in einer anderen JVM gefunden werden und deren Methoden aufgerufen werden. Die Proxys Stub(Client-Seite) und Skeleton(Server-Seite) werden von der JVM automatisch erstellt. Der Client findet den Server über die RMI-Registry, bei der der Server angemeldet werden muss und eine Referenz auf das Objekt für den Client hinterlegt.

Implementierung in Java

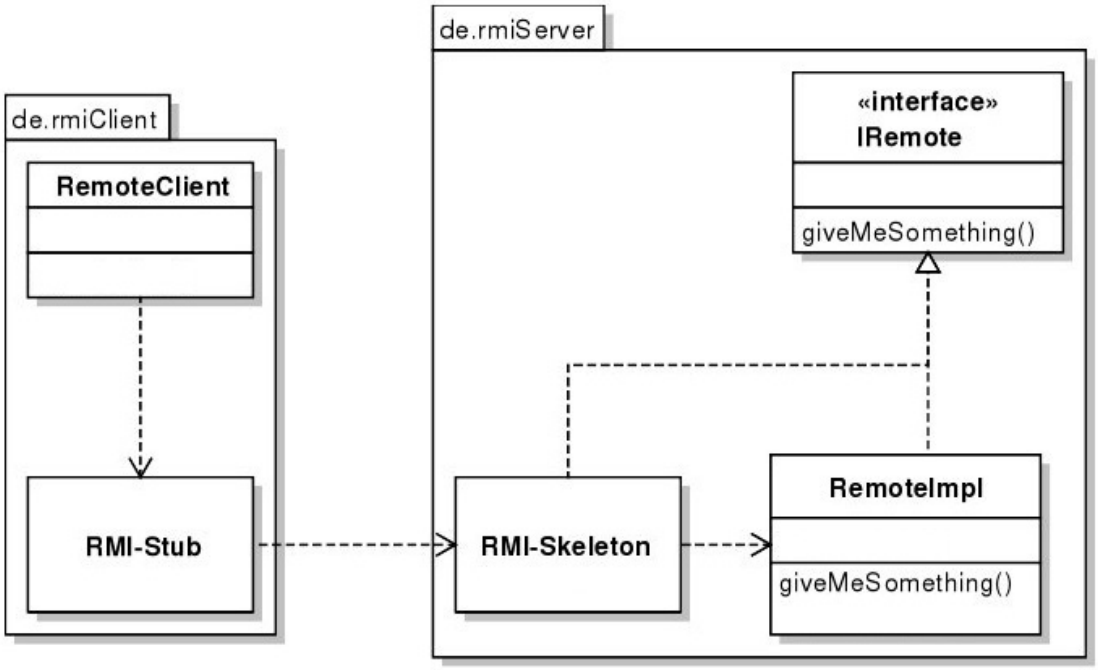


Abb. 14.5: Klassendiagramm eines Remote Proxy

Das Klassendiagramm in der Abbildung 14.5 zeigt eine einfache Implementierung eines Remote Proxy. Das Programm ruft vom Client aus über einen, von der JVM erstellten Remote Proxy eine Methode auf dem Server auf. Die Methode gibt eine Meldung aus.

```
1  package de.rmiServer;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  //muss von Remote abgeleitet sein
7  public interface IRemote extends Remote {
8
9          public String giveMeSomething() throws RemoteException;
10 }
```

```java
package de.rmiServer;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

//muss von UnicastRemoteObject abgeleitet sein um ein Remote Objekt zu erzeugen + eigenes Interfaces
public class RemoteImpl extends UnicastRemoteObject implements IRemote {

	public RemoteImpl() throws RemoteException {}
	Thread serverThread;
	public String giveMeSomething() {
		// TODO Auto-generated method stub
		return "Here is Remote with something!";
	}

	public static void main(String[] args) {

		try {
			//RMI Registrierung erstellen
			LocateRegistry.createRegistry(1099);

			//Remote Objekt erzeugen
			IRemote service = new RemoteImpl();

			//Registrieren Referenz auf Server und Referenz auf Objekt uebergeben
			Naming.rebind("//127.0.0.1/Server", service);
		} catch(Exception ex) {
			ex.printStackTrace();
		}
	}
}
```

```java
package de.rmiClient;

import java.rmi.Naming;
import java.rmi.RemoteException;
import de.rmiServer.IRemote;

public class RemoteClient {

	public RemoteClient(){ }

	public static void main(String[] args) {
		try {
			//Mit dem Server Verbinden
			IRemote service = (IRemote) Naming.lookup("//127.0.0.1/Server");

			//die Methode ausfuehren
			String s = service.giveMeSomething();

			System.out.println(s);
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
}
```

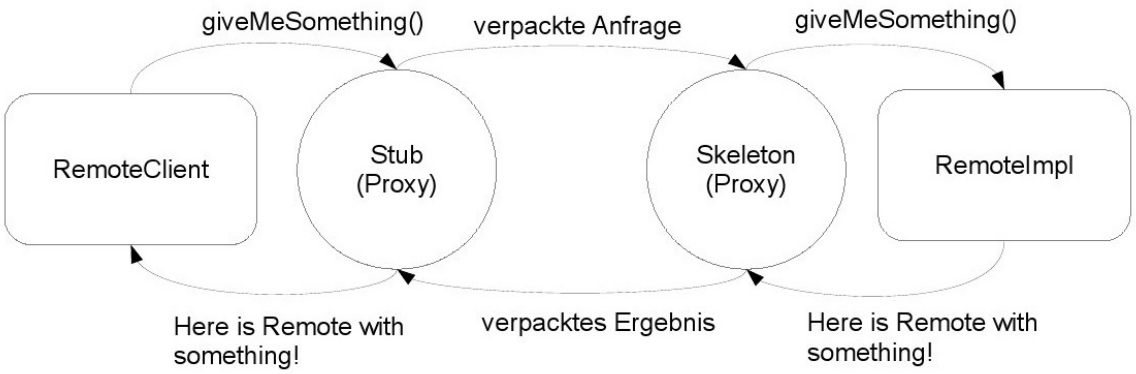


Abb. 14.6: Ablauf des Programms

In der Abbildung 14.6 wird der Ablauf des Programms dargestellt. Stub und Skeleton werden automatisch erzeugt.

14.5.4 Weitere Ausprägungen

Neben den oben angeführten Ausprägungen des Proxy Pattern gibt es noch weitere die an dieser Stelle kurz beschrieben werden.

Cache Proxy

Ein Cache Proxy wird zum Zwischenspeichern von Daten verwendet. Betrachtet man einen Datenbank, so reicht der Cache Proxy eine Anfrage eines Klienten an das Originalobjekt Datenbank weiter. Die von der Datenbank an den Proxy zurückgelieferten Ergebnisse, speichert er intern ab und liefert diese, bei einer gleichen Anfrage, ohne auf die Datenbank nochmals zuzugreifen, direkt an den Klienten zurück. Die Datenbank wir dadurch entlastet.

Firewall Proxy

Ein Firewall Proxy wird verwendet um den Zugriff auf einen Netzwerkschnittstelle zu kontrollieren.

Syncronizing Proxy

Bei mehreren gleichzeitigen, synchron stattfindenden, Zugriffen auf eine Komponente wird der Syncronizing Proxy verwendet. Betrachtet man wieder eine Datenbank, so kann es passieren, dass mehrere Klienten mittels des Syncronizing Proxy auf die Datenbank zugreifen und ein und den selben Datensatz anfragen oder ändern wollen. Der Proxy regelt ab hier dann die Zugriffe auf die Datenbank, um Inkonsistenz im Vorfeld zu vermeiden.

Counting Proxy

Der Counting Proxy wird eingesetzt wenn Komponenten nicht zufällig gelöscht werden sollen. Es können Referenzen auf gemeinsam genutzte Komponenten gezählt werden und gelöscht werden, wenn diese nicht mehr genutzt werden. Weiterhin können auch Statistiken über die Nutzung von Komponenten mitgeführt werden. Er eignet sich besonders für Analysezwecke, z.B. wie oft eine Komponente genutzt wird, um diese dann eventuell auszulagern.

14.6 Gegenüberstellung

Da sich viele Design-Pattern in ihrem Aufbau sehr ähnlich sind, werden nachstehend einige gegenübergestellt und verglichen. Die Abbildung 14.7 veranschaulicht die gemeinsame Grundstruktur verschiedener Entwurfsmuster.

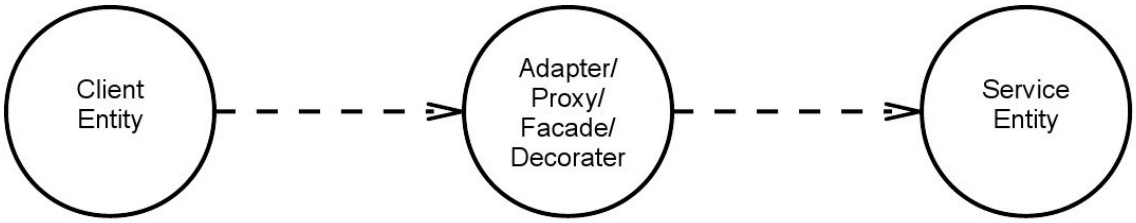


Abb. 14.7: gemeinsame Grundstruktur verschiedener Entwurfsmuster

14.6.1 Proxy vs. Decorater

- Das Proxy-Pattern kontrolliert den Zugriff auf sein Objekt.
- Das Decorater-Pattern fügt seinem Objekt Verhalten hinzu.
- Der Proxy vertritt sein Objekt gegenüber den Klienten.
- Der Decorater packt sein Objekt ein(Wrapper).

14.6.2 Proxy vs. Adapter

- Das Proxy-Pattern verändert das Verhalten seines Dienstes, behält aber dessen Schnittstelle.
- Das Adapter-Pattern verändert die Schnittstelle seines Dienstes aber behält dessen Verhalten.
- Ein Client kann sowohl die Proxy- als auch die Service Entity Instanz nutzen.
- Hingegen kann die Service Entity Instanz nicht vom Client der Adapter Instanz genutzt werden.

- Der Proxy kann in eine Service Instanz gecastet werden, da beide das gleiche Interface implementieren.
- Der Adapter kann nur in eine Instanz, die der Client erwartet gecastet werden.

14.6.3 Proxy vs. Fassade

- Proxys sind optional, Fassaden in der Regel nicht.
- Der Proxy kontrolliert das Verhalten seines Objekts.
- Die Fassade vereinfacht das Verhalten seines Objekts und kann auch Verhalten von ihm entfernen.

Literaturverzeichnis

[App11] Apple: *Cocoa Fundamentals*, 2011, http://developer.apple.com/library/ios/documentation/Cocoa/C
.

[boo11] *boost.signals*, 2011, http://www.boost.org/doc/html/signals.html.

[Eil10] Eilebrecht, K.; Starke, G.: *Patterns Kompakt - Entwurfsmuster für effektive Softwareentwicklung*, Spektrum, 2010.

[Erb09] Erb, B.: *Zustandsautomat in Java mithilfe des State Patterns)*, 2009, http://www.ioexception.de/2009/06/21/zustandsautomat-in-java-mithilfe-des-state-patterns/.

[Fow04] Fowler, M.: *Inversion of Control Containers and the Dependency Injection pattern*, 2004, http://martinfowler.com/articles/injection.html.

[Fow06a] Fowler, M.: *GUI Architectures*, 2006, http://www.martinfowler.com/eaaDev/uiArchs.html.

[Fow06b] Fowler, M.: *Passive View*, 2006, http://www.martinfowler.com/eaaDev/PassiveScreen.html.

[Fre04] Freeman, E.; Freeman, E.; Sierra, K.; Bates, B.: *Head First Design Patterns*, O'Reilly, 2004.

[Fre08] Freeman, E.; Freeman, E.: *Entwurfsmuster von Kopf bis Fuß*, O'Reilly, 2008.

[Gam94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[Gos05] Gossman, J.: *Introduction to MVVM*, 2005, http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[Gre07] Greer, D.: *Interactive Application Architecture Patterns*, 2007, http://www.aspiringcraftsman.com/2007/08/25/interactive-application-architecture/.

[Hau09] Hauer, P.: *Das Design Pattern Fassade*, 2009, http://www.philipphauer.de/study/se/design-pattern/facade.php.

[Hau11a] Hauer, P.: *Design Pattern*, 2011, http://www.philipphauer.de/study/se/design-pattern/strategy.php#gof.

[Hau11b] Hauer, P.: *State Design Pattern*, 2011, http://www.philipphauer.de/study/se/design-pattern/state.php.

[Kü11] Kübler; Rieck; Stanullo; Vollmer; Weiss: *Ferienkurs Design Patterns*, 2011, welearn.

[Lov11] Loviscach, P. D. J.: *Onlinetutorium State Pattern*, 2011, http://www.onlinetutorium.com/product_info.php?cPath=28_36&products_id=420.

[Mic11] Microsoft: *Serialisierung (C# und Visual Basic)*, 2011, http://msdn.microsoft.com/de-de/library/ms233843.aspx.

[Nok11] Nokia: *Qt*, 2011, http://qt.nokia.com.

[Pag11] Paggen, M.: *Delegation*, 2011, http://timpt.de/it/topic86.html.

[Pro11] Project, A. B.: *Android Binding*, 2011, http://code.google.com/p/android-binding/.

[Ree79] Reenskaug, T.: *Models-Views-Controllers*, 1979, http://heim.ifi.uio.no/ trygver/1979/mvc-2/1979-12-MVC.pdf.

[Sen11] Senyurt, B. S.: *Design Patterns:Strategy*, 2011, http://www.csharpnedir.com/dotnettv/watch/?id=159.

[Ste01] Stelting, S.; Leeuwen, O. M.-V.; Leeuwen, O. M.: *Applied Java Patterns*, Sun Microsystems, 2001.

[Swt11] SwtWiki: *Strategie*, 2011, http://www.imn.htwk-leipzig.de/~weicker/pmwiki/pmwiki.php/Main/Strategie.

[wik11] *Dependency Injection – Wikipedia*, 2011, http://de.wikipedia.org/wiki/Dependency_Injection.

[Wim10] Wimmer, G.: *State Pattern (Verhaltensmuster)*, 2010, http://www.gwimmer.info/infos/index.php?id=144.

[Win98] Winter, N.: *2 OOP für Fortgeschrittene*, 1998, http://www.fh-wedel.de/~si/seminare/ws97/Ausarbeitung/2.Winter/gamma12.htm.