VU Software Engineering 2
VU-Leiter: Prof. Uwe Zdun
Gruppenarbeit Task 3 – Design Patterns in Practice 3

TASK 3 – DESIGN PATTERNS

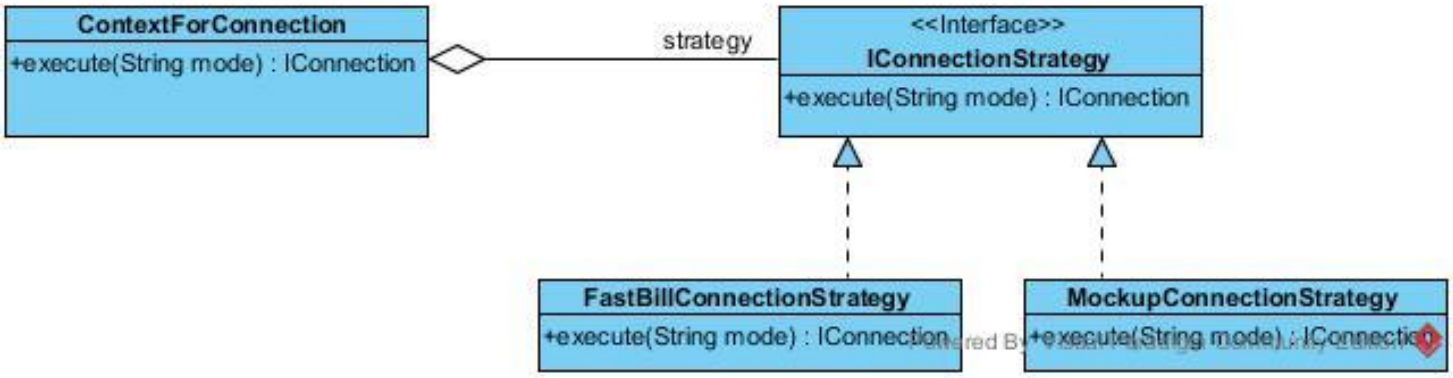
Team 1, Gruppe 3

Cordula Eggerth (0750881), a0750881@unet.univie.ac.at
Sandra Hofmarcher (1404086), a01404086@unet.univie.ac.at
Jasmin Klementschitz (1328827), a01328827@unet.univie.ac.at
Martin Regenfelder (1104500), a01104500@unet.univie.ac.at

Strategy Pattern („ConnectionStrategyPattern“):
Implementiert von: Cordula Eggerth

Im Projekt wurde das Strategy Pattern (mit Hilfes des Interface IConnectionStrategy) verwendet. Das Pattern ermöglicht, dass man verschiedene Algorithmen auswechselbar machen kann, und diese über ein einheitliches Interface anspricht. Das Strategy-Interface der allgemeinen Definition des Patterns (siehe Folien Kapitel „Design Patterns 1“ von Prof. Zdun) wurde im Projekt Interface IConnectionStrategy benannt. Es stellt die Schnittstelle für die Algorithmen dar, die je nach passender Situation ausgewechselt werden können, um eine Connection zu erzeugen und den Client, der hier durch die Klasse ContextForConnection dargestellt wird, unabhängig vom jeweiligen Algorithmus werden lässt.
Die untenstehende Abbildung zeigt das UML-Diagramm der Anwendung des Strategy Patterns im Projekt. Der „Client“ ist die ContextForOpereation Klasse. Sie jeweils einen der Algorithmen, wenn die Methode execute aufgerufen wird. Die Klassen FastBillConnectionStrategy und MockupConnectionStrategy implementieren das Interface IConnectionStrategy bzw. dessen Methode „execute(String mode): IConnection“. Sie können darüber im Client verwendet werden, je nachdem, welche Connection gefragt ist.

UML-Klassendiagram des ConnectionStrategyPattern:



Zum Interface IConnectionStrategy (siehe untenstehendes Code-Stück):

```
1  package ConnectionStrategyPattern;
2
3
4  import Logic.IConnection;
5
6  /**
7   * Das Interface IConnectionStrategy gibt die Methode execute, die jeweils von den Klassen
8   * FastBillConnectionStrategy und MockupConnectionStrategy implementiert wird in ihrer jeweiligen
9   * Ausfuehrung zur Erstellung einer Connection (je nach Konfigurationsmodus), an.
10  * @author Cordula Eggerth
11  *
12  */
13 public interface IConnectionStrategy {
14
15     /**
16      * execute legt je nach mode (Konfigurationsmodus, d.h. entweder productive oder test) eine Connection
17      * an und gibt diese an die aufrufende Klassen zurueck.
18      * @param mode
19      * @return IConnection
20      */
21     public IConnection execute(String mode);
22
23 }
24
```

Im Interface IConnectionStrategy wird die Methode execute(String mode), die ein Objekt des Typs IConnection zurückliefert, deklariert. Diese Methode wird dann je nach Typ der gefragten Connection überschrieben.

Zur Klasse MockupConnectionStrategy (siehe untenstehendes Code-Stück):

```java
package ConnectionStrategyPattern;

import AbstractFactoryPattern.IConnectionFactory;
import AbstractFactoryPattern.MockupConnectionFactory;
import Logic.IConnection;

/**
 * MockupConnectionStrategy implementiert das Interface IConnectionStrategy. Es legt eine neue
 * MockupConnection an und gibt diese an die aufrufende Klasse zurueck. Die Klasse ist Teil des
 * ConnectionStrategyPattern.
 * @author Cordula Eggerth
 *
 */
public class MockupConnectionStrategy implements IConnectionStrategy {

    /**
     * execute legt eine neue MockupConnection an und gibt diese zurueck.
     * @param mode
     * @return IConnection
     */
    @Override
    public IConnection execute(String mode) {

        IConnectionFactory factory = null;
        factory = new MockupConnectionFactory();
        IConnection connection = factory.createConnection();
        return connection;

    }
}
```

In MockupConnectionStrategy wird die Methode execute aus dem Interface IConnectionStrategy überschrieben. Hier wird eine neue MockupConnectionFactory angelegt, um eine IConnection vom Typ MockupConnection zu erzeugen.

Zur Klasse FastBillConnectionStrategy (siehe untenstehendes Code-Stück):

```java
package ConnectionStrategyPattern;

import AbstractFactoryPattern.FastBillConnectionFactory;
import AbstractFactoryPattern.IConnectionFactory;
import Logic.IConnection;

/**
 * FastBillConnectionStrategy implementiert das Interface IConnectionStrategy. Es legt eine neue
 * FastBillConnection an und gibt diese an die aufrufende Klasse zurueck. Die Klasse ist Teil des
 * ConnectionStrategyPattern.
 * @author Cordula Eggerth
 *
 */
public class FastBillConnectionStrategy implements IConnectionStrategy {

    /**
     * execute legt eine neue FastBillConnection an und gibt diese zurueck.
     * @param mode
     * @return IConnection
     */
    @Override
    public IConnection execute(String mode) {

        IConnectionFactory factory = null;
        factory = new FastBillConnectionFactory();
        IConnection connection = factory.createConnection();
        return connection;

    }

}
```

In FastBillConnectionStrategy wird die Methode execute aus dem Interface IConnectionStrategy überschrieben. Hier wird eine neue FastBillConnectionFactory angelegt, um eine IConnection vom Typ FastBillConnection zu erzeugen.

Zur Klasse ContextForConnection (siehe untenstehendes Code-Stück):

```java
/**
 * ContextForConnection ist Teil des ConnectionStrategyPattern. Die
 * Ausfuehrung des Strategy Patterns und der Methode execute.
 * @author Cordula Eggerth
 *
 */
public class ContextForConnection {

    /**
     * Instanzvariable
     */
    private IConnectionStrategy strategy;

    /**
     * Konstruktor
     * @param strategy
     */
    public ContextForConnection(IConnectionStrategy strategy){
        this.strategy = strategy;
    }

    /**
     * Execute erstellt ein Objekt vom Typ IConnection je nachdem o
     * FastBillConnection) oder test (d.h. Verwendung der MockupCon
     * @param mode
     * @return
     */
    public IConnection execute(String mode){
        return strategy.execute(mode);
    }

}
```

Die Klasse ContextForConnection hat ein Objekt vom Typ IConnectionStrategy als Instanzvariable und ruft auf dieser die Methode execute auf, die einen String mit Konfigurationsmodus übergeben bekommt, und dann ein Objekt vom Typ IConnection zurückliefert. Diese Klasse entspricht dem allgemeinen „Client“, der das Strategy Interface verwendet.

Aufruf des Strategy Pattern in AddCustomer.java in der Methode actionPerformed des Button Listener des „Save Customer“ Buttons (siehe untenstehendes Code-Stück):

```
        Customer newCustomer = new Customer(0,customerType,organization,lastname);
        IConnection connection = null;

        ContextForConnection context = null;

        if(mode.equals("productive")) {
            context = new ContextForConnection(new FastBillConnectionStrategy());
        }
        else { // if mode test is selected
            context = new ContextForConnection(new MockupConnectionStrategy());
        }

        connection = context.execute(mode);
        connection.createCustomer(newCustomer);

        JOptionPane.showMessageDialog(frame, "New Customer was successfully created.");

    }
```

Der Aufruf des Strategy Pattern erfolgt z.B. hier in der Klasse AddCustomer.java, in der ein Objekt vom Typ ContextForConnection angelegt wird, dem im Konstruktor die jeweilige ConnectionStrategy übergeben wird. Darauf wird dann die Methode execute in der jeweiligen Ausführung aufgerufen, und dann wird der Customer über die jeweilige Connection (FastBill oder Mockup) angelegt.

Vorteile/Nachteile des Strategy Patterns:

Im Strategy Pattern können Familien von verwandten Algorithmen definiert werden, die dann je nach Kontext in der jeweils passenden Ausführung benützt werden. Dies findet z.B. im vorliegenden Projekt über das Interface IConnectionStrategy statt, das von FastBillConnectionStrategy und MockupConnectionStrategy implementiert wird.

Die Verwendung des Interface IConnectionStrategy ist hier eine gute Alternative zur Vererbung, weil trotzdem verschiedene Arten von Algorithmen und Verhalten der Methoden durchgeführt werden können.

Durch die Verwendung des Strategy Pattern können auch verschiedene Algorithmen derselben Verhaltensweise angeschlossen werden.

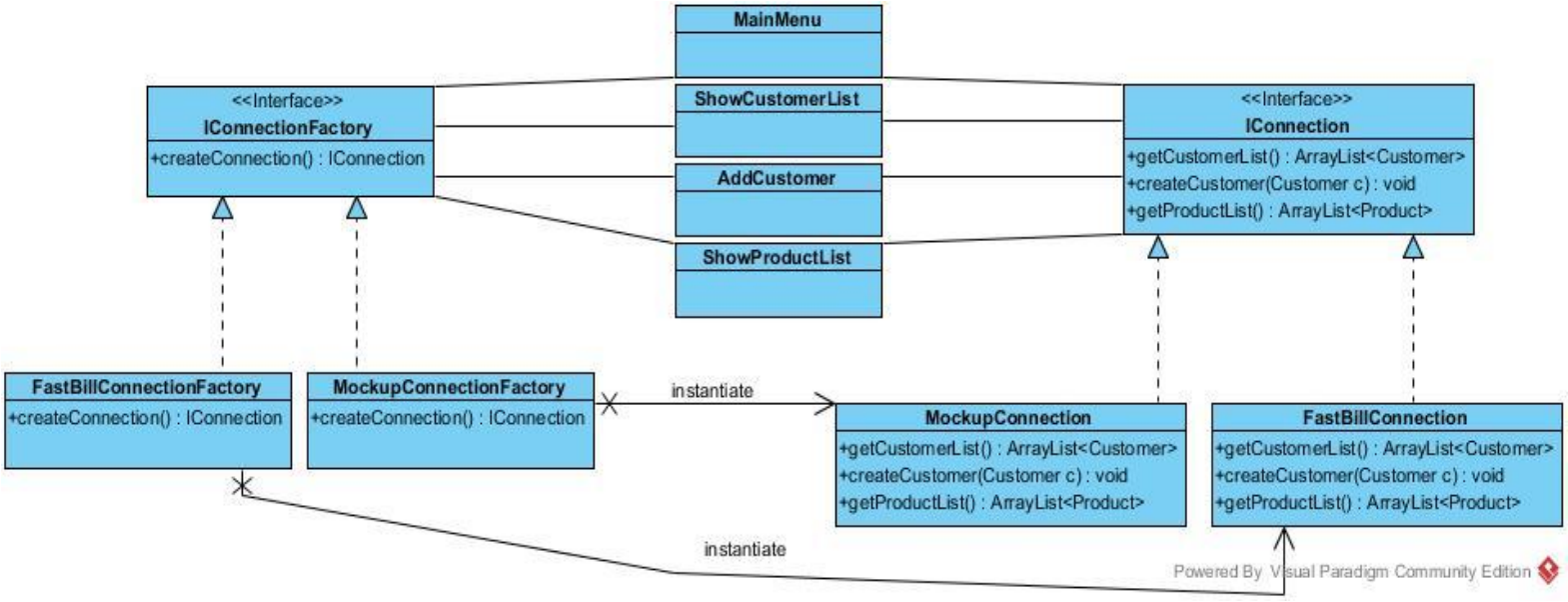
Ein Nachteil des Strategy Patterns kann sein, dass der Client (u.a. die MainMenu-Klasse) die verschiedenen Arten der Connection kennen muss, damit diese verwendet werden können. Allerdings war dies kein allzu großes Problem

im Projekt. Außerdem steigt auch die Anzahl der Objekte durch die Verwendung des Strategy Patterns, und dadurch die Komplexität.

Abstract Factory Pattern:
Imlementiert von: Cordula Eggerth

Das Abstract Factory Pattern dient im Projekt zur Erstellung der jeweiligen Connection, entweder vom Typ FastBillConnection oder MockupConnection, um dem geforderten Konfigurationsmodus mit API Requests bzw. mit Mockup (das Serialisierung verwendet) zu entsprechen. Es werden also zwei Ausführungen der ConnectionStrategy angeboten. Diese können jeweils die passende Connection erzeugen.
Vorteilhaft ist hier, dass das Pattern dabei hilft, dass man die Klassen, die für die Connection bildbar sind, durch die Abstract Factory steuern kann. Es kann nur eine dieser zwei Ausprägungen der Connection erzeugt werden über die jeweilige passende konkrete Factory. Daher können Produktfamilien leichter getauscht werden, indem man die entsprechende konkrete Factory einsetzt.
Schwierig ist es nur, dass man neue Produktarten nicht so leicht dazuhängen kann – es muss extra ein neuer Bereich über die Erweiterung der bestehenden Interfaces angelegt werden und für die neue Produktart komplett überdacht werden.

UML-Klassendiagram des Abstract Factory Pattern:



Zum Interface IConnectionFactory (siehe untenstehendes Code-Stück):

```java
package AbstractFactoryPattern;

import Logic.IConnection;

/**
 * IConnectionFactory ist das Interface, das als Grundlage fuer die Erzeugung einer Factory dient.
 * Es ist Teil des Abstract Factory Patterns.
 * @author Cordula Eggerth
 */
public interface IConnectionFactory {

    /**
     * createConnection erzeugt entweder eine FastBillConnection (mit Verwendung der FastBillAPI) oder
     * eine MockupConnection (mit Verwendung der Daten aus dem File bzw. Serialisierung)
     * @return IConnection
     */
    public IConnection createConnection();

}
```

Das Interface IConnectionFactory dient dazu, dass man entweder eine FastBillConnectionFactory oder eine MockupConnectionFactory erzeugen kann, und damit jeweils nachgelagert die passende Connection anlegen kann. Das Interface wird von den genannten Klassen, die die Methode createConnection() ausgestalten, implementiert.

Zur Klasse FastBillConnectionFactory (siehe untenstehendes Code-Stück):

```java
package AbstractFactoryPattern;

import Logic.FastBillConnection;

/**
 * FastBillConnectionFactory implementiert das Interface IConnectionFactory und dient zur Erzeugung einer
 * fastbillconnection. Es ist Teil des Abstract Factory Pattern.
 * @author Cordula Eggerth
 */
public class FastBillConnectionFactory implements IConnectionFactory {

    /**
     * createConnection implementiert die Methode aus dem Interface IConnectionFactory zur Erzeugung
     * einer connection - hier als FastBillConnection.
     */
    @Override
    public IConnection createConnection() {
        return new FastBillConnection();
    }
}
```

Die Klasse FastBillConnectionFactory implementiert das Interface IConnectionFactory und implementiert die Methode createConnection(), die eine neue Connection vom Typ FastBillConnection zurückliefert.

Zur Klasse MockupConnectionFactory (siehe untenstehendes Code-Stück):

```java
package AbstractFactoryPattern;

import Logic.IConnection;

/**
 * MockupConnectionFactory implementiert das Interface IConnectionFactory und dient zur Erzeugung einer
 * mockupconnection. Es ist Teil des Abstract Factory Pattern.
 * @author Cordula Eggerth
 */
public class MockupConnectionFactory implements IConnectionFactory {

    /**
     * createConnection implementiert die Methode aus dem Interface IConnectionFactory zur Erzeugung
     * einer connection - hier als MockupConnection.
     */
    @Override
    public IConnection createConnection() {
        return new MockupConnection();
    }

}
```

Die Klasse MockupConnectionFactory implementiert das Interface IConnectionFactory und implementiert die Methode createConnection(), die eine neue Connection vom Typ MockupConnection zurückliefert.

Zum Interface IConnection (siehe untenstehendes Code-Stück):

```java
package Logic;

import java.util.ArrayList;

/**
 * Das Interface IConnection gibt die Methode vor, die in den Connections angeboten werden.
 * Die Klasse wird im Abstract Factory Pattern verwendet.
 * @author Cordula Eggerth
 *
 */
public interface IConnection {

    /**
     * getCustomerList liefert eine Liste der bestehenden Customers.
     * @return ArrayList Customer
     */
    public ArrayList<Customer> getCustomerList();

    /**
     * createCustomer legt einen neuen Kunde, der als Parameter uebergeben wurde, an.
     * @param customer
     */
    public void createCustomer(Customer customer);

    /**
     * getProductList liefert eine Liste der angebotenen Products.
     * @return ArrayList Customer
     */
    public ArrayList<Product> getProductList();

}
```

Das Interface IConnection gibt die Methoden vor, die von der jeweiligen implementierenden Klasse angeboten werden müssen, wie z.B. getCustomerList(), createCustomer(Customer customer) oder getProductList(). Die Ausgestaltung der jeweilige Methode unterscheidet sich allerdings darin, welche Art von Connection vorliegt.

Zur Klasse FastBillConnection (siehe untenstehendes Code-Stück):

```
 */
public class FastBillConnection implements IConnection {

    /**
     * getCustomerList holt die Customer Liste von FastBill.
     * Dafuer wird ein API Request gesendet und ein JSON Format als Response erhalten.
     * Das JSON Format wird eingelesen und eine customerList (ArrayList von Typ Custom
     * wird daraus erstellt.
     * @return customerList
     */
    @Override
    public ArrayList<Customer> getCustomerList(){

        ArrayList<Customer> customerList = new ArrayList<Customer>();

        HttpClient httpClient = HttpClientBuilder.create().build();
        HttpPost post = null;
        try {
            post = new HttpPost("https://my.fastbill.com/api/1.0/api.php");
            String encoding = DatatypeConverter.printBase64Binary("morri.kocht@gmail.c

            post.setHeader("Authorization", "Basic " + encoding);
            String json = "{\"SERVICE\":\"customer.get\",\"FILTER\":{}}";
            HttpEntity entity = new ByteArrayEntity(json.getBytes("UTF-8"));
```

Die Klasse FastBillConnection implementiert die Methoden des Interface IConnection und macht API Requests an die FastBillAPI – sie bietet also den Konfigurationsmodus „productive“ an.

Zur Klasse MockupConnection (siehe untenstehendes Code-Stück):

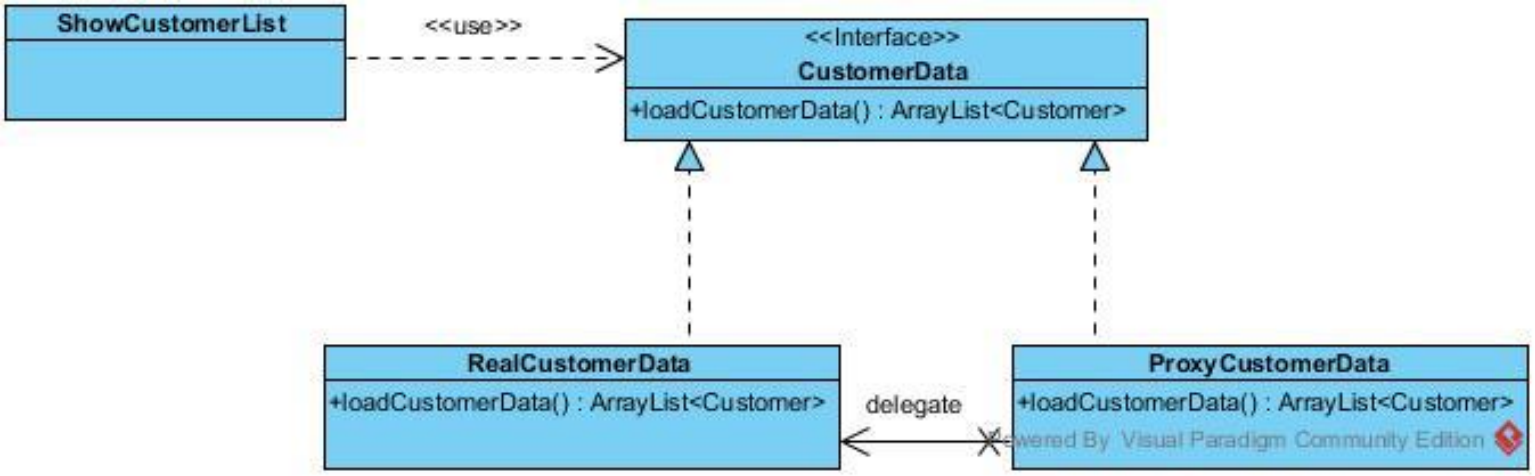
```
1  package Logic;
2
3  import java.io.File;
15
16 /**
17  * Die Klasse MockupConnection implementiert das Interface IConnection, das die Methoden
18  * den Connections angeboten werden, vorgibt.
19  * Die Klasse wird im Abstract Factory Pattern verwendet.
20  * @author Cordula Eggerth
21  *
22  */
23 public class MockupConnection implements Serializable, IConnection {
24
25     /**
26      * Instanzvariablen
27      */
28     private static final long serialVersionUID = 1L;
29     private FastBillConnection fastBillConnection;
30
31     public MockupConnection(){
32         this.fastBillConnection = new FastBillConnection();
33     }
34
35     /**
36      * writeCurrentCustomerListIntoFile schreibt die aktuelle Customer List
37      * (von FastBill) in ein File.
38      */
39     public void writeCurrentCustomerListIntoFile(){
40         String filePath="customerList.ser";
41
42         ArrayList<Customer> filedCustomerList = fastBillConnection.getCustomerList();
43
44         try {
45             FileOutputStream fileOutput = new FileOutputStream(filePath);
```

Die Klasse MockupConnection implementiert die Methoden des Interface IConnection und macht Anfragen, für deren Erfüllung Informationen in ein File geschrieben werden oder aus einem File gelesen werden – sie bietet also den Konfigurationsmodus „test“ an und verwendet Serialisierung der jeweiligen Objekte.

Proxy Pattern:
Implementiert von: Cordula Eggerth

Das Proxy Pattern wird im Projekt angewendet, indem bei erstmaliger Verwendung der Application das File, das eigentlich leer wäre, also ein Proxy File ist, über eine Anfrage des Ladevorgang von der echten Liste von FastBill, mit aktuellen Daten geladen wird. Somit muss der Benutzer der Application beim „test“ configuration mode nicht mit einer leeren Liste starten, sondern das Proxy Pattern ermöglicht, dass eine Customer Liste mit aktuellen Daten geladen wird und als Datenbasis dienen kann. ProxyCustomerData fragt den echten Ladevorgang der Daten bei RealCustomerData an, wo die Daten geladen werden und dann das File, wie gewünscht on-demand, mit diesen Daten gesetzt wird.

UML-Klassendiagramm der Anwendung des Proxy Patterns:



Zum Interface CustomerData (siehe untenstehendes Code-Stück):

Im Interface CustomerData wird die Methode loadCustomerData vorgegeben, die von den implementierenden Klassen ausgestaltet wird.

```java
package ProxyPattern;

import java.util.ArrayList;

/**
 * Das Interface CustomerData gibt die Methode loadCustomerData vor, die von RealCustomerData und
 * ProxyCustomerData implementiert werden.
 * Es ist Teil des ProxyPattern.
 * @author Cordula Eggerth
 *
 */
public interface CustomerData {

    /**
     * loadCustomerData dient dazu, die Customer Daten zu laden.
     * @return ArrayList Customer
     */
    public ArrayList<Customer> loadCustomerData();

}
```

Zur Klasse RealCustomerData (siehe untenstehendes Code-Stück):

```java
package ProxyPattern;

import java.util.ArrayList;

/**
 * RealCustomerData ladet (bzw. schreibt) neue Kundendaten in ein File, falls die Kundenliste aus dem File leer ist, damit in der C
 * Daten angezeigt werden koennen. Dieser Ladevorgang, der von RealCustomerData durchgefuehrt wird, wird von ProxyCustomerData
 * angefragt.
 * Die Klasse RealCustomerData implementiert das Interface CustomerData.
 * Die Klasse ist Teil des ProxyPattern.
 * @author Cordula Eggerth
 */
public class RealCustomerData implements CustomerData {

    /**
     * loadCustomerData implementiert die Methode des Interface CustomerData,
     * um Customer Daten zu laden.
     * @return customerList
     */
    @Override
    public ArrayList<Customer> loadCustomerData() {

        MockupConnection connection = new MockupConnection();

        connection.writeCurrentCustomerListIntoFile();  // falls noch keine Liste im File angelegt, dann schreibt aktuelle Liste

        ArrayList<Customer> customerList=connection.getCustomerList();
        return customerList;
    }

}
```

In der Klasse RealCustomerData wird die Methode loadCustomerData aus dem Interface CustomerData implementiert. Es wird eine aktuelle Liste der Customer Daten in das File, aus dem die Daten im configuration mode „test“ gelesen werden, geschrieben, damit der Benutzer direkt mit diesen Daten arbeiten kann.

Zur Klasse ProxyCustomerData (siehe untenstehendes Code-Stück):

ProxyCustomerData.java

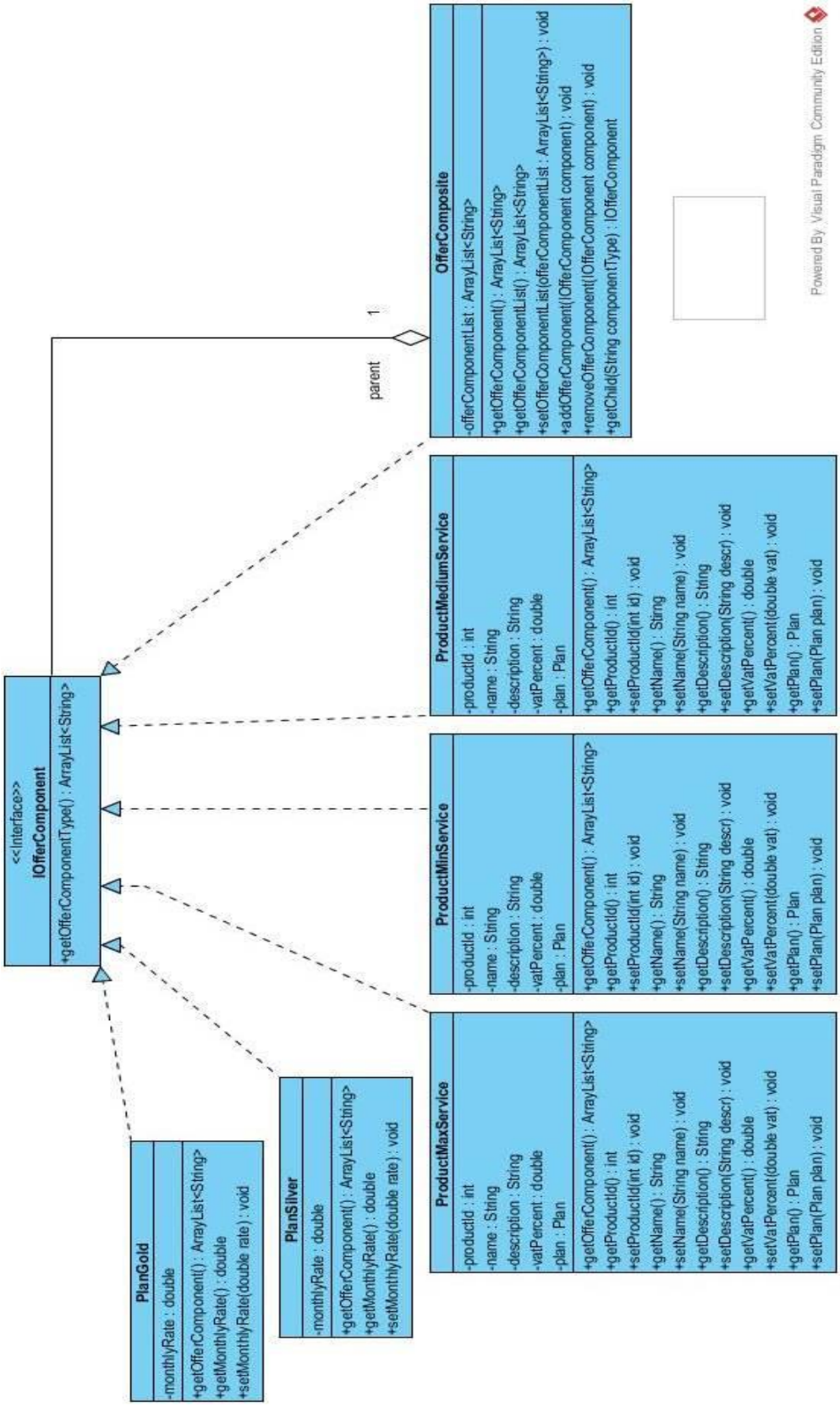
```
1  package ProxyPattern;
2
3  import java.util.ArrayList;
7
8  /**
9   * ProxyCustomerData implementiert das Interface CustomerData und ladet, fal
10  * eine neue Kundenliste mit den aktuellen FastBill Daten in das File. Der L
11  * Die Klasse ist Teil des ProxyPattern.
12  * @author Cordula Eggerth
13  */
14 public class ProxyCustomerData implements CustomerData {
15
16     /**
17      * loadCustomerData implementiert die Methode des Interface CustomerData
18      * um Customer Daten zu laden.
19      * Falls die customerList Daten enthaelt, werden diese verwendet.
20      * Falls die customerList keine Daten enthaelt, werden die Daten ueber R
21      * wo der Ladevorgang durchgefuehrt wird.
22      * @return customerList
23      */
24     @Override
25     public ArrayList<Customer> loadCustomerData() {
26
27         MockupConnection connection = new MockupConnection();
28         ArrayList<Customer> customerList = new ArrayList<Customer>();
29
30         if(connection.getCustomerList().size()==0){
31             RealCustomerData updatedCustomerData = new RealCustomerData();
32             customerList = updatedCustomerData.loadCustomerData();
33             return customerList;
34         } else {
35             return connection.getCustomerList();
36         }
```

In der Klasse ProxyCustomerData wird das Interface CustomerData mit der dazugehörigen Methode loadCustomerData implementiert. Die Customer Daten werden zunächst aus dem File gelesen. Falls die serialisierte Liste leer ist, wird RealCustomerData angefragt, dass über seinen Ladevorgang eine Customer Daten Liste erzeugt wird. Denn der Proxy hätte sonst keine Daten und der Benutzer müsste mit einer leeren Liste arbeiten. Daher ist es hier praktisch mittels Proxy Pattern die Daten über den von RealCustomerData ausgeführten Ladevorgang in die customerList zu laden, und dann die weiteren Serialisierungsoperationen basierend auf dieser Liste weiterzuführen.

Composite Pattern:
Implementiert von: Cordula Eggerth

Das Composite Pattern hilft im vorliegenden Projekt dabei, das Angebot an Produkten/Plänen zu aggregieren, und sich einen Überblick über das Gesamtangebot der Firma, also das OfferComposite, zu verschaffen. Es können dafür die verschiedenen OfferComponents (z.B. Products, Pläne in ihrer jeweiligen Ausführung) zu OfferComposites aggregiert werden, je nachdem, welchen Umfang der Benutzer haben möchte.
Praktiksch beim Einsatz das Composite Pattern hier ist, dass man die OfferComponents, die durchaus aus unterschiedlichen Klassen stammen über eine gemeinsame Zugangsweise alle als OfferComponent einheitlich ansprechen kann bzw. innerhalb eines OfferComposite-Objekts mit einheitlich ihnen arbeiten kann, da sie alle das Interface IOfferComponent implementieren. Es ist hier auch von Vorteil, dass man, wenn dies gewünscht ist, leicht weitere Klassen als OfferComponents aufnehmen kann, da diese nur das Interface IOfferComponent implementieren müssen.

UML-Klassendiagramm der Anwendung des Proxy Patterns:



Zum Interface IOfferComponent (siehe untenstehendes Code-Stück):

IOfferComponent.java

```java
package CompositePattern;

import java.util.ArrayList;

/**
 * Das Interface IOfferComponent ist Teil des CompositePatterns.
 * Es zeigt, welchen Kategorien von Komponenten sich das derzeitige Sortiment zusammensetzt.
 * @author Cordula Eggerth
 *
 */
public interface IOfferComponent {

    /**
     * Die Methode getOfferComponentType liefert die Kategorie der Angebotskomponente zurueck.
     * @return
     */
    public ArrayList<String> getOfferComponentType();
}
```

Das Interface IOfferComponent bietet die Methode getOfferComponentType an.

Zu den Klassen PlanGold und PlanSilver (die alle abgeleitet sind von Klasse Plan):

```java
package CompositePattern;

import java.io.Serializable;

/**
 * Die Klasse Plan definiert den monatlichen Preis, den man fuer eine
 * Produkt Subscription bezahlen muss.
 * @author Cordula Eggerth
 *
 */
public class Plan implements Serializable {

    /**
     * Instanzvariablen
     */
    private static final long serialVersionUID = 1L;
    private double monthlyRate;   // monatlich zu bezahlender Preis fuer das Produk

    /**
     * Konstruktor
     * @param monthlyRate
     */
    /**
     */
    public Plan(double monthlyRate) {
        this.monthlyRate = monthlyRate;
    }
```

Plan ist die Oberklasse, von der die Plans PlanGold und PlanSilver abgeleitet werden. Die abgeleiteten Plans implementieren das Interface IOfferComponent und können so zu einem OfferComposite hinzugefügt werden.

Zu den Klassen ProductMaxService, ProductMediumService, ProductMinService (die alle abgeleitet sind von Klasse Product):

```
1  package CompositePattern;
2
3  import java.io.Serializable;
4
5  /**
6   * Die Klasse Product ist die model Ober-Klasse fuer Produkte.
7   * Davon werden 3 Produkte abgeleitet.
8   * @author Cordula Eggerth
9   *
0   */
1  public class Product implements Serializable {
2
3      /**
4       * Instanzvariablen
5       */
6      private static final long serialVersionUID = 1L;
7      private int productId;
8      private String name;
9      private String description;
0      private double vatPercent; // umsatzsteuer in prozent
1      private Plan plan;      // 1 product hat jeweils 2 plans, aus denen 1 ausgewaehlt wird bei erstellu
2
3      /**
4       * Konstruktor
5       * @param productId
6       * @param name
7       * @param description
8       * @param vatPercent
9       * @param plan
0       */
1      public Product(int productId, String name, String description, double vatPercent, Plan plan) {
2          this.productId = productId;
3          this.name = name;
4          this.description = description;
```

Product ist die Oberklasse, von der die Produkte ProductMaxService, ProductMediumService und ProductMinService abgeleitet werden. Die abgeleiteten Produkte implementieren das Interface IOfferComponent und können so zu einem OfferComposite hinzugefügt werden.

Zur Klasse OfferComposite (siehe untenstehendes Code-Stück):

```
*/
public class OfferComposite implements IOfferComponent {

    /**
     * Instanzvariable
     */
    private ArrayList<IOfferComponent> offerComponentList;

    /**
     * Konstruktor
     */
    public OfferComposite () {
        this.offerComponentList = new ArrayList<IOfferComponent>();
    }

    /**
     * GET -UND SET-METHODEN
     */

    /**
     * get offerComponentList
     * @return offerComponentList
     */
    public ArrayList<IOfferComponent> getOfferComponentList() {
        return offerComponentList;
    }

    /**
     * set offerComponentList
     * @param offerComponentList
     */
    public void setOfferComponentList(ArrayList<IOfferComponent> offerComponentList) {
        this.offerComponentList = offerComponentList;
    }
```

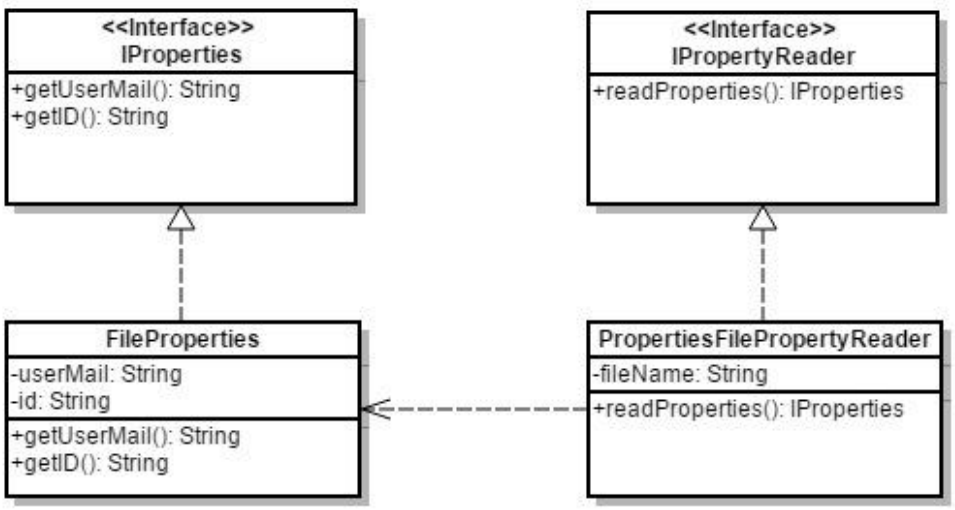
```java
/**
 * addOfferComponent fuegt eine offerComponent zur offerCompositeList hinzu
 * @param offerComponent
 */
public void addOfferComponent(IOfferComponent offerComponent){
    offerComponentList.add(offerComponent);
}

/**
 * removeOfferComponent loescht eine offerComponent aus der offerCompositeList
 * @param offerComponent
 */
public void removeOfferComponent(IOfferComponent offerComponent){
    offerComponentList.remove(offerComponent);
}


/**
 * getChild liefert anhand eines uebergebenen componentType die jeweilige IOfferComponent in
 * der offerComponentList.
 * @param componentType
 */
public IOfferComponent getChild(String componentType) {
    for(int i=0;i<this.offerComponentList.size();i++) {
        if(this.offerComponentList.get(i).getOfferComponentType().get(0).equals(componentType)){
            return this.offerComponentList.get(i);
        }
    }
    return null;
}
```

Die Klasse OfferComposite implementiert das Interface IOfferComponent und ist ein Aggregat, das aus mehreren Plänen oder Produkten bestehen kann. Es kann Teile des Produkt- und Planangebots der Firma enthalten oder auch das ganze Angebot. Die Klasse bietet auch die Methoden addOfferComponent, removeOfferComponent und getChild, die Teile des allgemeinen Composite Pattern sind zusätzlich zu den eigenen Methoden.

Factory Method Pattern:
Implementiert von: Sandra Hofmarcher

Das Factory Method Pattern wird dafür verwendet, flexibel bei der Auswahl vom Auslesen der Zugangsdaten für den FastBill-Account zu bleiben. Durch die Einbindung des Pattern, kann jederzeit eine weitere Möglichkeit, die Benutzerdaten auszulesen, implementiert werden.

UML-Klassendiagramm der Anwendung des Factory Method Patterns:



Zum Interface IPropertyReader (siehe untenstehendes Code-Stück):

Im Interface IPropertyReader wird die Methode readProperties(): vorgegeben, die von der implementierenden Klasse ausgestaltet wird.

IPropertyReader.java

```java
package FactoryMethodPattern;

/**Das Interface dient zur Abstraktion der Klasse und gewährt die nötige Flexibilität zur Laufzeit.
 *
 * @author Sandra Gabriela Hofmarcher
 *
 */
public interface IPropertyReader {

    /**Methode um die benoetigten Zugangsdaten aus einer Datei zu lesen
     *
     * @return IProperties
     */
    public IProperties readProperties();

}
```

Zur Klasse PropertiesFilePropertyReader (siehe untenstehendes Code-Stück):

PropertiesFilePropertyReader.java

```java
package FactoryMethodPattern;

import java.io.FileInputStream;

/**Die Klasse liest die benoetigten Benutzerdaten aus der Datei aus und uebergibt sie der Schnittstelle IProperties
 *
 * @author Sandra Gabriela Hofmarcher
 *
 */

public class PropertiesFilePropertyReader implements IPropertyReader {

    /**
     * Name bzw Path der zu lesenden Datei, in welcher die benötigten Benutzerdaten vermerkt sind
     */
    private String fileName= "implementation/config.properties";


    @Override
    public IProperties readProperties() {

        IProperties iprop = null;
        InputStream input = null;


        try{
            Properties prop = new Properties();
            input = new FileInputStream(fileName);

            prop.load(input);

            iprop = new FileProperties(prop.getProperty("email"), prop.getProperty("key"));

        } catch(IOException e){
            e.printStackTrace();
        }finally{
            if(input != null){
                try{
                    input.close();
                }catch (IOException e){
                    e.printStackTrace();
                }
            }
        }
        return iprop;
    }

}
```

Die Klasse PropertiesFilePropertyReader implementiert das Interface IPropertyReader und definiert die dadurch erhaltene Methode readProperties():.

Innerhalb des try/catch-Blocks wird die Datei mittels des Dateipfades eingelesen. Die daraus gelesenen Benutzerdaten werden dem Interface IProperties übergeben.

Zum Interface IProperties (siehe untenstehendes Code-Stück):

*IProperties.java

```
1  package FactoryMethodPattern;
2
3  /**Das Interface dient zur Abstraktion der Klasse, um die Fexibilität zur Laufzeit zu gewähren.
4   *
5   * @author Sandra Gabriela Hofmarcher
6   *
7   */
8  public interface IProperties {
9
10     /**Methode um die Benutzer- E-Mail zu erhalten
11      * @param email
12      *
13      * @return userEmail
14      */
15     String getUserMail();
16
17     /**Methode um den API-Key zu erhalten
18      *
19      * @return id bzw. API-Key
20      */
21     String getID();
22
23 }
```

Um die Flexibilität des Zugriffes auf die Benutzerzugangsdaten für die API-Schnittstelle zu gewähren, dient das Interface IProperties als Abstraktion.

Zur Klasse FileProperties (siehe untenstehendes Code-Stück):

*FileProperties.java

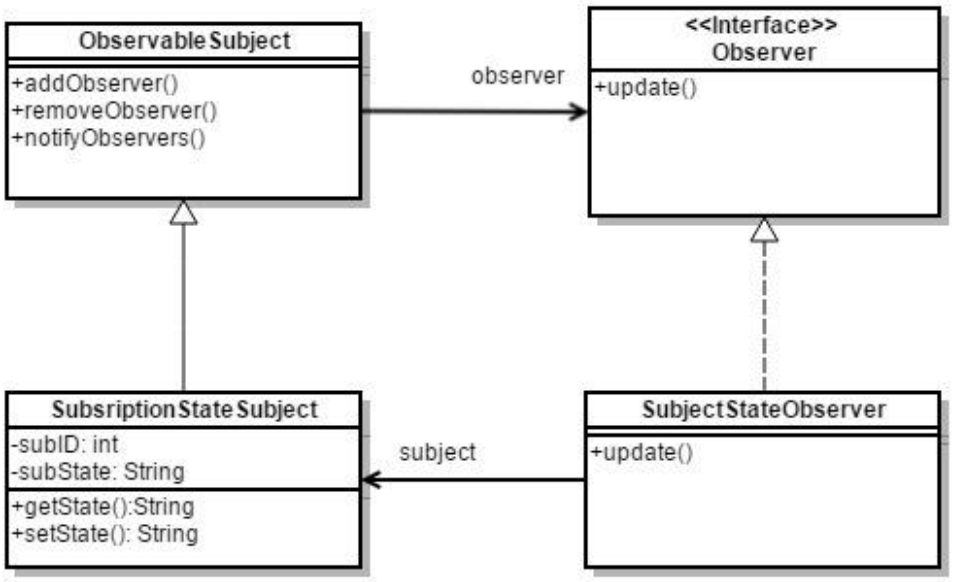
```java
package FactoryMethodPattern;

/**Diese Klasse speichert die ausgelesenen Benutzerdaten für die API-Schnittstelle.
 * Bewusst wurden keine set-Methoden erzeugt, um die Benutzerdaten vor Korruption zu bewahren.
 * Das Objekt ist immutable.
 *
 * @author Sandra Gabriela Hofmarcher
 *
 */
public class FileProperties implements IProperties {

    /**Benutzername bzw. E-Mailadresse für FastBill
     *
     */
    private String userMail;

    /**ID bzw. API-Key für FastBill
     *
     */
    private String id;

    /**Der Konstruktor erhält die Variablen userMail und id über das Interface IProperties
     *
     * @param userMail
     * @param id
     */
    FileProperties(String userMail, String id){
        this.userMail = userMail;
        this.id = id;
    }

    @Override
    public String getUserMail() {
        return this.userMail;
    }

    @Override
    public String getID() {
        return this.id;
    }

}
```

Die Klasse FileProperties erhält die Benutzer-Zugangsdaten für die API-Schnittstelle über das Interface IProperties. Das Objekt ist imutable und besitzt deswegen auch keine set-Methoden.

Observer Pattern:
Implementiert von Sandra Hofmarcher

Das Observer Pattern wird dazu verwendet, um den Status einer Subscription zu überwachen. Das SubscriptionStateSubject wird mit der Funktion subscripe(Plan plan) in der Klasse Customer automatisch angelegt.

UML-Klassendiagramm der Anwendung des Factory Method Patterns:



Zur Klasse SubscriptionStateSubject (siehe untenstehendes Code-Stück):

SubscriptionStateSubject.java

```java
package ObserverPattern;

import java.util.Observable;
/**
 * Die Klasse SubscriptionStateSubject speichert sowohl die ID, als auch den Status der jeweiligen Subscription.
 * Sie erweitert die Klasse Observable und meldet an SubjectStateObserver, wenn sich der Status aendert.
 * @author Sandra Gabriela Hofmarcher
 *
 */
public class SubscriptionStateSubject extends Observable  {

    /**
     * subID ist die ID der Subscription
     */
    int subID;

    /**
     * Status der Subscription
     */
    String subState= null;

    /**
     * Konstruktor
     * @param subID
     */
    public SubscriptionStateSubject(int subID){
        this.subID = subID;
    }


    /**
     * Methode um die SubscriptionID zu erhalten
     * @return subState
     */
    public String getState(){
        return this.subState;
    }

    /**
     * Methode um den Status zu aendern. Gleichzeitig setzt diese Funktion ein Aenderungs-Flag und benachrichtigt die Observer
     * @param subState
     */
    public void setState(String subState){
        this.subState = subState;
        setChanged();
        notifyObservers();
    }

}
```

Die Klasse SubscriptionStateSubject enthält die ID der dazugehörigen Subscription, dessen Status sie verwahrt. Die Klasse erweitert die Klasse Observable. Wenn sich der Status innerhalb von SubscriptionStateSubject ändert, wird ein Änderungs-Flag gesetzt und die Methode notifyObservers() aufgerufen, um die Observer zu aktualisieren.

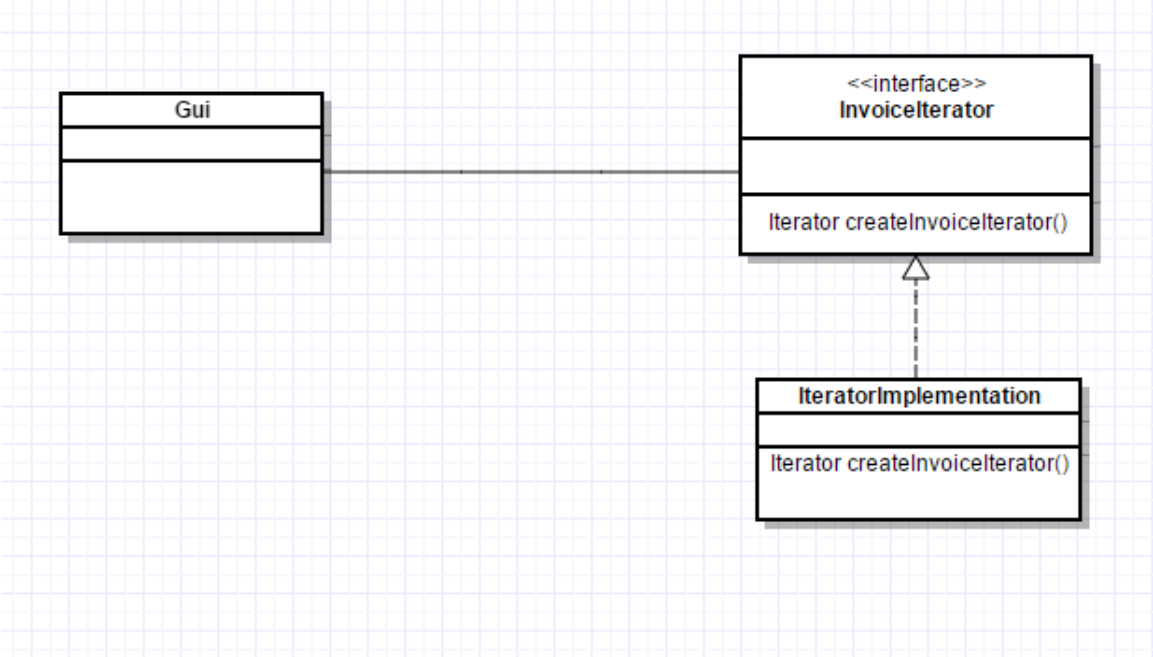
Zur Klasse SubjectStateObserver (siehe untenstehendes Code-Stück):

```
1  package ObserverPattern;
2
3  import java.util.Observable;
5
6  /**
7   * Die Klasse dient als Beispiel für einen Observer, der informiert wird, wenn Aenderungen im SubscriptionStateSubject auftreten.
8   * @author Sandra Gabriela Hofmarcher
9   *
0   */
1
2  public class SubjectStateObserver implements Observer {
3
4      @Override
5      public void update(Observable o, Object arg) {
6      }
7
8  }
```

Die Klasse SubjectStateObserver beobachtet die Klasse SubscriptionStateSubject bzw. dessen darin gespeichert Status. Wenn sie mittels notifyObservers() darüber informiert wurde, dass eine Änderung vorgenommen wurde. Aktualisiert sie mittels update den Status.

Iteratorpattern („InvoiceIterator und SubscriptionIterator“):
Implementiert von: Martin Regenfelder

Im Projekt wurde für Collections des Typs Invoice und Subscription jeweils ein Iterator angelegt. Hierbei wird über ein entsprechendes Iterator-Interface die darunter liegende Collection jeweils nur als Iterator zur verfügung gestellt. Dies bewirkt, dass man sich keine Gedanken darüber machen muss, welche Datenstruktur denn darunter Liegt, da in jedem Fall immer der Iterator verwendet wird.



Umsetzung im Code:
InvoiceIterator

```
package iteratorPattern;

import java.util.Iterator;
/**
 * Interface für Collections of Invoices
 * @author Martin
 *
 */
public interface InvoiceIterator {

    @SuppressWarnings("rawtypes")
    public Iterator createInvoiceIterator();

}
```

SubscriptionIterator

```
package iteratorPattern;

import java.util.Iterator;
/**
 * Interface für Collections of Subscriptions
 * @author Martin
 *
 */
public interface SubscriptionIterator {

    @SuppressWarnings("rawtypes")
    public Iterator createSubscriptionIterator();

}
```

Implementation:

```
/**
 * Liefert ArrayList<Invoice> als Iterator
 */
@Override
public Iterator<Invoice> createInvoiceIterator() {
    return getAllInvoices().iterator();
}

/**
 * Liefert ArrayList<Subscription> als Iterator
 */
@Override
public Iterator<Subscription> createSubscriptionIterator() {
    return getAllSubscriptions().iterator();
}
```

In beiden Fällen wird ein Iterator für eine ArrayList returned.

Anwendungsbeispiel

```
MockupConnection mcon = new MockupConnection();
InvoiceIterator invoiceIterator = mcon;
SubscriptionIterator subscriptionIterator = mcon;
Iterator<Invoice> invoice = invoiceIterator.createInvoiceIterator();
Iterator<Subscription> subscription = subscriptionIterator.createSubscriptionIterator();
while(invoice.hasNext()){
    Invoice inv = (Invoice) invoice.next();
    System.out.println(inv.getInvoiceId());
    System.out.println(inv.getSubscriptionID());
}
while(subscription.hasNext()){
    Subscription sub = (Subscription) subscription.next();
    System.out.println(sub.getSubscriptionStatus());
    System.out.println(sub.getCustomerID());
}
```

Eine Mockupconnection (die beide Interfaces implementiert) wird erzeugt. Beide InvoiceIterator und SubscriptionIterator werden initialisiert.
Ein Iterator für Invoices und ein Iterator für Subscriptions wird erzeugt. Mit diesen kann dann durch jedes Objekt der Collecitons durchiteriert werden, und auf ihre jeweiligen Attribute zugegriffen werden.

Quellen:

Uwe Zdun. 2017. Foliensatz Design Patterns 1-4.