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

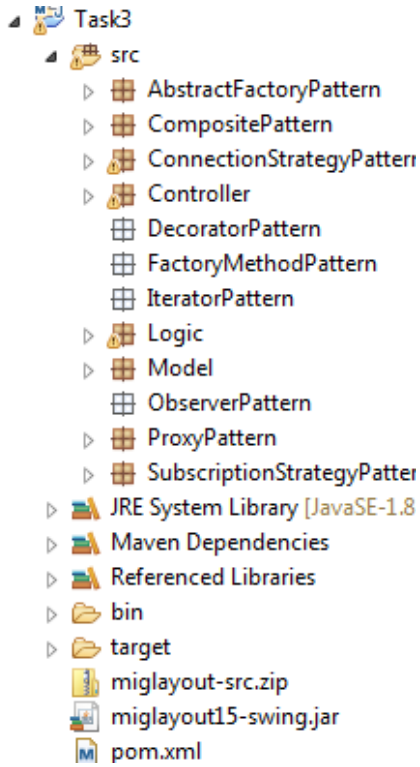
TASK 3 – CODING PRACTICES

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

Hinsichtlich der Anwendung von Coding Practices wurde auf die Namensgebung bei Methoden, Interfaces, Klassen und Packages geachtet. Außerdem wurden die Coding Practices in Bezug auf die Kommentare, Formatierung und die logische Code-Organisation berücksichtigt und im Projekt angewendet. Dank den Coding Practices konnte die Lesbarkeit, Wartbarkeit und Erweiterbarkeit des Codes erhöht werden.
Anschließend wird anhand von Beispielen aus dem Projekt für Task3 erläutert, wie und wo die Coding Practices umgesetzt wurden.

Bereits in der Strukturierung des Java-Projekts wurden die Coding Practices berücksichtigt. Die Packages und Klassen behandeln jeweils in sich zusammenpassende Inhalte und haben klar abgegrenzte Verantwortungsbereiche:



Bei der Namensgebung wurde darauf geachtet, dass die Namen der Klassen, Variablen, Methoden und Packages klar ersichtlich werden lassen, was genau damit gemeint ist, und welche Teil des Projekts sie sind bzw. welche Aufgaben sie erfüllen. Im Sinne der Bedeutung sollte die Namenswahl anderen nicht mit dem Projekt vertrauten Programmierern klar sein und es sollte leicht erkennbar sein, welche Aufgaben damit erfüllt werden. Die gewählten Bezeichnungen sollten zusätzlich leicht auszusprechen, nicht extrem lang oder extrem kurz bzw. uneutlich sein und sie sollten leicht über die Suchfunktion auffindbar sein, denn zu kurz Namen z.B. aus nur wenigen Buchstaben oder zu allgemeine Namen sind schlecht merkbar und können auch schlecht gesucht werden.
Die Namen von Objekten und Klassen sollten Nomen oder Phrasen aus Nomen sein: z.B.: FastBillConnection. Die Namen von Methoden sollten Verben oder Verbphrasen sein: z.B.: createCustomer(Customer customer).
Im Sinne der Coding Practes wurde auch die Prinzipien für das Kommentieren des Code berücksichtigt. Die Kommentare (javadoc und normale java comments) sollten der Dokumentation und als Orientierungshilfe für Programmierer und all jene, die mit dem Code arbeiten, dienen. Die Kommentare sollten übersichtlich darstellen, was gewisse Code-Teile machen, damit auch neu dazukommenden Kollegen ermöglicht wird, schnell zu verstehen, was die jeweiligen Teile des Code machen. Kommentare sollten im Großen und Ganzen nicht allzu kurz aber auch nicht allzu lang sein. Wichtig ist, dass gut vermittelt wird, was die bedeutendsten Punkte sind, die der Leser mitnehmen und verstehen sollte in Bezug auf das Code-Stück, auf das sich der Kommentar bezieht.
Hinsichtlich der logischen Anordnung und der Code-Formatierung gilt es zu beachten, dass ein passender Einrückungsstil gewählt wird. Dieser muss nicht ein bestimmter Stil sein, sondern sollte durchgehend durch das Projekte durchgehalten werden, und sollte eine gute Strukturierung des Codes ermöglichen, um die Verständlichkeit zu erhöhen.

Im folgenden Dokumentationsabschnitt werden einige Beispiele vorgestellt, wie Coding Practices im Projekt für Task3 umgesetzt wurden.

Code-Beispiel 1:

Customer.java

```java
/**
 * Model Klasse des Customer.
 * Sie enthaelt die Instanzvariablen des Customer (abgestimmt auf die notwendigen Instanzva
 * die durch die FastBill API vorgegeben werden) und die Get- und Set-Methoden fuer Custome
 * @author Cordula Eggerth
 *
 */
public class Customer implements Serializable {

    /**
     * Instanzvariablen
     */
    private static final long serialVersionUID = 1L;
    private int id;
    private String customerType;
    private String organization;
    private String lastname;

    /**
     * Konstruktor
     * @param id
     * @param customerType
     * @param organization
     * @param lastname
     */
    public Customer(int id, String customerType, String organization, String lastname) {
        this.id = id;
        this.customerType = customerType;
        this.organization = organization;
        this.lastname = lastname;
    }

```

Im ***Code-Beispiel 1*** wird Javadoc zur Erklärung der Aufgabenbereiche der Klasse verwendet. Es werden darin mittels @param und @return bzw. auch @author die Informationen über Parameter, Returnwerte und Autoren gegeben.
Die Klasse Customer hat einen leicht verständlichen, nicht zu langen, nicht zu kurzen und gut suchbaren Name. Auch die Instanzvariablen entsprechen den Coding Practices für die Namensgebung, da sie gut lesbar, verständlich, bezüglich Länge angemessen und von der Bedeutung her leicht erkennbar sind.
Der Konstruktor und seine Parameter sind ebenfalls klar aufgebaut und haben leicht erkennbare und gut verständliche Parameternamen.

Code-Beispiel 2:

```java
/**
 * GET- UND SET-METHODEN
 */

/**
 * get Id des Customer
 * @return id
 */
public int getId() {
    return id;
}

/**
 * set Id des Customer
 * @param id
 */
public void setId(int id) {
    this.id = id;
}

/**
 * get Customer Type
 * @return customerType
 */
public String getCustomerType() {
    return customerType;
}
```

Im ***Code-Beispiel 2*** wird mittels javadoc comments erklärt, was die einzelnen Methoden als Aufgabenbereiche haben. Die return-Werte und Parameter werden mittels @return bzw. @param klar und deutlich ausgewiesen. Die Methoden entsprechen der Konvention für Getter und Setter, wie z.B. getId() oder setId(int id), indem sie eine Verbalphrase enthalten und leicht lesbar und verständlich sind.

Code-Beispiel 3:

FastBillConnectionFactory.java

```
1  package AbstractFactoryPattern;
2
3  import Logic.FastBillConnection;
5
6  /**
7   * FastBillConnectionFactory implementiert das Interface IConnectionFact
8   * fastbillconnection. Es ist Teil des Abstract Factory Pattern.
9   * @author Cordula Eggerth
10  */
11 public class FastBillConnectionFactory implements IConnectionFactory {
12
13     /**
14      * createConnection implementiert die Methode aus dem Interface ICor
15      * einer connection - hier als FastBillConnection.
16      */
17     @Override
18     public IConnection createConnection() {
19         return new FastBillConnection();
20     }
21 }
22
```

Im ***Code-Beispiel 3*** wird ebenfalls mittels javadoc comments erklärt, was die einzelnen Methoden als Aufgabenbereiche haben. Die erwähnten Packages wie hier AbstractFactoryPattern sind klar und deutlich. Denn dies zeigt, dass es sich tatsächlich bei der vorliegenden Klasse um eine Klasse, die dem AbstractFactoryPattern angehört, handelt. Der Name der Klasse FastBilConnectionFactory ist klar verständlich und die Methode createConnection kann leicht verstanden und gesucht werden. Sie erstellt eine FastBillConnection.

Code-Beispiel 4:

```
@Override
public void createCustomer(Customer 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.com:2d60347a07aa9b4196dab08f51

        post.setHeader("Authorization", "Basic " + encoding);
        String json = "{\"SERVICE\":\"customer.create\",\"DATA\":{\"CUSTOMER_TYPE\":\""+customer.getCustomerTyp
        HttpEntity entity = new ByteArrayEntity(json.getBytes("UTF-8"));
        post.setEntity(entity);

        HttpResponse response = httpClient.execute(post);

            // System.out.println(EntityUtils.toString(response.getEntity()));
            System.out.println(EntityUtils.toString(response.getEntity()));

    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        post.abort();
    }
}
```

Im ***Code-Beispiel 4*** wird gezeigt, dass die verwendeten Variablen, wie z.B. httpClient oder response passende Namen haben, die leicht darauf schließen lassen, welchen Aufgabenbereich die jeweilige Variable hat. Der Code wird außerdem gut strukturiert und mit einer passenden Einrückung formatiert. Der Code innerhalb der Methode ist um einen Tab weiter nach rechts gerückt. Dann beginnt der try-Block, innerhalb von dem der Code wiederum um einen Tab nach rechts gerückt ist. Nach dem try-Block beginnen die catch-Blöcke, in denen der Code wieder um einen Tab nach rechts gerückt wird. Diese Einrückungsmethodik und Code-Strukturierung dient der besseren Nachvollziehbarkeit und Übersichtlichkeit.

Code-Beispiel 5:

```
btnGetCustomerList = new JButton("Get Customer List");
btnGetCustomerList.setEnabled(false);
getContentPane().add(btnGetCustomerList, "cell 0 8");

        btnGetProductList = new JButton("Get Product List");
        btnGetProductList.setEnabled(false);
        getContentPane().add(btnGetProductList, "cell 0 9");
        btnGetProductList.addActionListener(new ActionListener() {

            /**
             * Bei Klick auf den Button "Get Product List" wird
             * auf das Fenster ShowProductList.java weitergeschaltet.
             * @param ActionEvent e
             */
    public void actionPerformed(ActionEvent e) {

        String mode = "";
        if(rdbtnProductive.isSelected()){
            System.out.println("productive");
            mode="productive";
        }
        else if(rdbtnTest.isSelected()){
            System.out.println("test");
        mode="test";
        }
```

Im ***Code-Beispiel 5*** wird gezeigt, dass die Variablen und Methoden klare und deutliche Namen erhalten haben. Beispielswise ist über den Sting mode leicht zu erkennen, das es sich um den Modus der Konfiguration handelt. Auch der Button für „GetCustomerList“ ist entsprechend der angebotenen Funktionalität benannt worden und die Liste der Produkte wurde poductList, was naheliegend ist, genannt. Die Einrückung wurde ebenfalls besonders in der actionPerformed Methode beim Schreiben der if-else-statements berücksichtigt.

Code-Beispiel 6:

```
btnGetUnpaidInvoices = new JButton("Get Unpaid Invoices for Customer");
btnGetUnpaidInvoices.setEnabled(false);
getContentPane().add(btnGetUnpaidInvoices, "cell 0 11");

// ActionListener fuer RadioButtons
 // Listener fuer productive radiobutton
rdbtnProductive.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        btnCreateCustomer.setEnabled(true);
        btnGetCustomerList.setEnabled(true);
        btnGetProductList.setEnabled(true);
        btnCreateInvoiceFor.setEnabled(true);
        btnGetUnpaidInvoices.setEnabled(true);
    }
});

// Listener fuer test radiobutton
rdbtnTest.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        btnCreateCustomer.setEnabled(true);
        btnGetCustomerList.setEnabled(true);
        btnGetProductList.setEnabled(true);
        btnCreateInvoiceFor.setEnabled(true);
        btnGetUnpaidInvoices.setEnabled(true);
```

Im ***Code-Beispiel 6*** wird insbesondere klar, dass auf eine leicht verständliche Namensgebung der Buttons Wert gelegt wurde, denn dadurch sind die Buttons, die in der Design-Ansicht erzeugt wurdne, in der Code-Ansicht leichter auffindbar und die Situation kann schneller von neu einzuarbeitenden Programmierern erfasst werden.