

TASK 3 – DEFENSIVE PROGRAMMING

Team 1, Gruppe 3

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

Im Projekt von Task 3 wird Defensive Programming unter anderem in der Form des Exception Handling verwendet. Wenn Fehler nicht ignoriert werden können, wird eine Exception an dieser Stelle geworfen, die dann mittels den zugehörigen catch-Blöcken aufgefangen und angemessen behandelt wird. Für den Programmierer wird auch eine entsprechende Ausgabe gemacht, damit klar ist, um welchen Fehler es sich handelt. Außerdem werden im GUI verschiedene Fehlermeldung und andere Informationsmeldungen ausgegeben, falls falsche Benutzereingaben gemacht werden. Im Falle von falschen Benutzereingaben, stürzt die Application nicht ab, sondern es wird der Prozess abgebrochen und der Zustand bleibt so, wie er es vor der falschen Eingabe war.

Im Folgenden werden einige Code-Beispiele dargestellt, die zeigen, wie Exceptions Handling im Projekt umgesetzt wurde.

Code-Beispiel 1:

```
@Override
public ArrayList<Product> getProductList(){

    ArrayList<Product> productList = new ArrayList<Product>();

    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:2d60347a07aa9b419");

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

        HttpResponse response = httpClient.execute(post);

        String jsonString = EntityUtils.toString(response.getEntity());

        JSONObject root = new JSONObject(jsonString);
        JSONObject jsonResponse = (JSONObject) root.get("RESPONSE");

        JSONArray products = (JSONArray) jsonResponse.get("ARTICLES");

        for(int i = 0 ; i < products.length(); i++){
            JSONObject article = (JSONObject) products.get(i);
```

[...]

```
        productList.add(newProduct);
    }
    else if(article.getString("TITLE").equals("ProductMinService-Gold")){
        PlanGold planGold = new PlanGold(article.getDouble("UNIT_PRICE"));
        ProductMinService newProduct = new ProductMinService(article.getInt("ARTICLE_NUMBER"), article.getString("TITLE"),
            article.getString("DESCRIPTION"),Double.parseDouble(article.getString("VAT_PERCENT")), planGold);
        productList.add(newProduct);
    }
    else { // if(article.getString("TITLE").equals("ProductMinService-Silver")){
        PlanSilver planSilver = new PlanSilver(Double.parseDouble(article.getString("UNIT_PRICE")));
        Product newProduct = new ProductMinService(article.getInt("ARTICLE_NUMBER"), article.getString("TITLE"),
            article.getString("DESCRIPTION"),Double.parseDouble(article.getString("VAT_PERCENT")), planSilver);
        productList.add(newProduct);
    }
}

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

return productList;
}
```

Im **Code-Beispiel 1** werden Ausschnitte der Methode getProductList() aus der Klasse FastBillConnection im Package Logic gezeigt. In dieser Klasse wird die Liste des aktuellen Product Angebots mittels FastBill API Request angefragt. Die gesamte Anfrage (API Request) wird in einem try-Block ausgeführt, denn es kann sein, dass Fehlersituationen auftreten. Diese Fehlersituationen werden in Form von catch-Blöcken, die die Exceptions auffangen, gehandhabt. Weiters gibt es eine finally-clause, die in jedem Fall am Ende nach dem try-Block ausgeführt wird und ein „Clean-Up“ macht.

Code-Beispiel 2:

```
@Override
public ArrayList<Customer> getCustomerList() {
    ArrayList<Customer> customerList = new ArrayList<Customer>();
    String filePath = "customerList.ser";
    File file = new File(filePath);
    if(file.exists() && file.canRead()) {
        try {
            FileInputStream fileInput = new FileInputStream(filePath);
            ObjectInputStream objectInput = new ObjectInputStream(fileInput);

            if(objectInput!=null) {
                customerList = (ArrayList<Customer>) objectInput.readObject();
                objectInput.close();
                fileInput.close();
            }

        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    return customerList;
}
```

Im **Code-Beispiel 2** wird die Methode `getCustomerList` der Klasse `MockupConnection` gezeigt. Diese verwendet Serialisierung und liest die Customer Liste aus einem File (`customerList.ser`) ein, wenn über den GUI durch Klick auf den zugehörigen Button eine Anfrage vom Benutzer gemacht wird. Es kann aber sein, dass beim Lesen des Files eine Fehlersituation auftritt, und dass dadurch eine Exception geworfen wird. Daher wird der gesamte Lesevorgang in einem try-Block ausgeführt. Wenn Exceptions auftreten, werden sie von den catch-Blöcken aufgefangen und entsprechendes Handling erfolgt.

Code-Beispiel 3:

```
@Override
public void createCustomer(Customer customer) {
    ArrayList<Customer> currentCustomerList = getCustomerList();
    String filePath = "customerList.ser";

    int maxExistingId=0;
    for(int i=0; i<currentCustomerList.size();i++){
        if(currentCustomerList.get(i).getId()>maxExistingId){
            maxExistingId=currentCustomerList.get(i).getId();
        }
    }
    maxExistingId++;
    customer.setId(maxExistingId);
    currentCustomerList.add(customer);

    try {
        FileOutputStream fileOutput = new FileOutputStream(filePath);
        ObjectOutputStream objectOutput = new ObjectOutputStream(fileOutput);
        objectOutput.writeObject(currentCustomerList);
        objectOutput.close();
        fileOutput.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Im **Code-Beispiel 3** wird die Methode `createCustomer` der Klasse `MockupConnection` gezeigt. Diese verwendet Serialisierung und schreibt den neu angelegten Customer in ein File (`customerList.ser`), wenn über den GUI durch Klick auf den zugehörigen „Save Customer“ Button eine Anfrage vom Benutzer gemacht und der Benutzer korrekte Eingaben gemacht hat. (Fehleingaben durch den Benutzer werden vor dem Speichervorgang schon abgefangen und behandelt.) Es kann aber sein, dass beim Schreiben des Files eine Fehlersituation auftritt, und dass dadurch eine Exception geworfen wird. Daher wird der gesamte Schreibvorgang in einem try-Block ausgeführt. Wenn

Exceptions auftreten, werden sie von den catch-Blöcken aufgefangen und entsprechendes Handling erfolgt.

In Bezug auf Defensive Programming ist in gewissen Situationen auch von Vorteil, wenn eine Meldung in Form eines Dialogs an den Benutzer stattfindet und angegeben wird, warum der eingegebene Wert nicht angenommen werden konnte. Daraufgehend wird die vom Benutzer gewünschte Aktion abgebrochen und der Vorgang wird quasi auf den Ausgangszustand zurückgesetzt, sodass nichts passiert. Der Benutzer kann dann wieder eine neue Anfrage machen, dass die Aktion durchgeführt wird. Denn eine Ausführung der Aktion würde eventuell zu schweren Fehlersituationen in der Application führen, sodass dann die Execution abgebrochen werden würde. Dies soll durch Defensives Programmieren vermieden werden. Die Application soll mit falschen Benutzereingaben umgehen können, und soll die Ausführung bei Fehlern weiterführen können.

Die folgenden Code-Beispiele veranschaulichen dies:

Code-Beispiel 4:

```
btnCreateInvoiceFor = new JButton("Create Invoice for Customer");
btnCreateInvoiceFor.setEnabled(false);
getContentPane().add(btnCreateInvoiceFor, "cell 0 10");

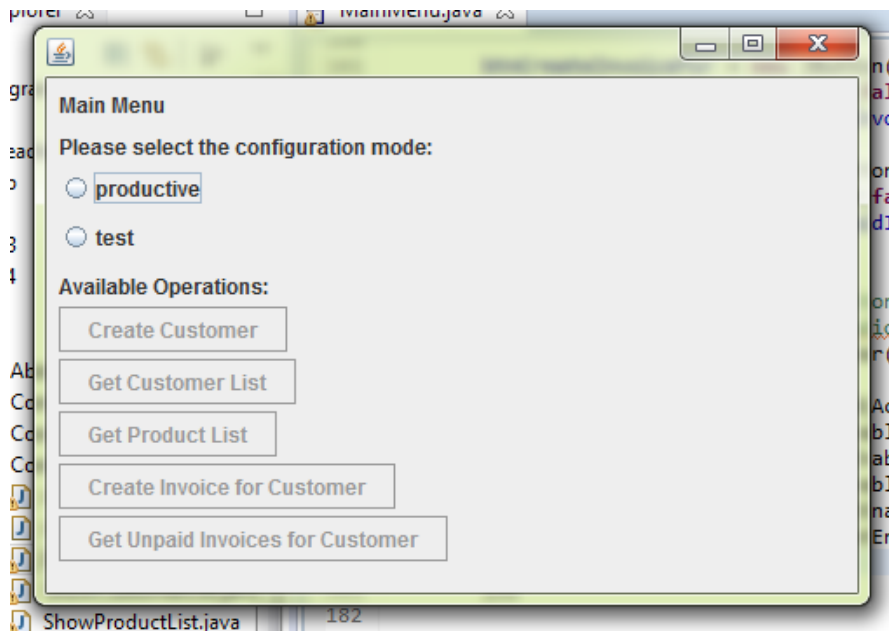
btnGetUnpaidInvoices = new JButton("Get Unpaid Invoices for Customer");
btnGetUnpaidInvoices.setEnabled(false);
getContentPane().add(btnGetUnpaidInvoices, "cell 0 11");

// 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);
    }
});

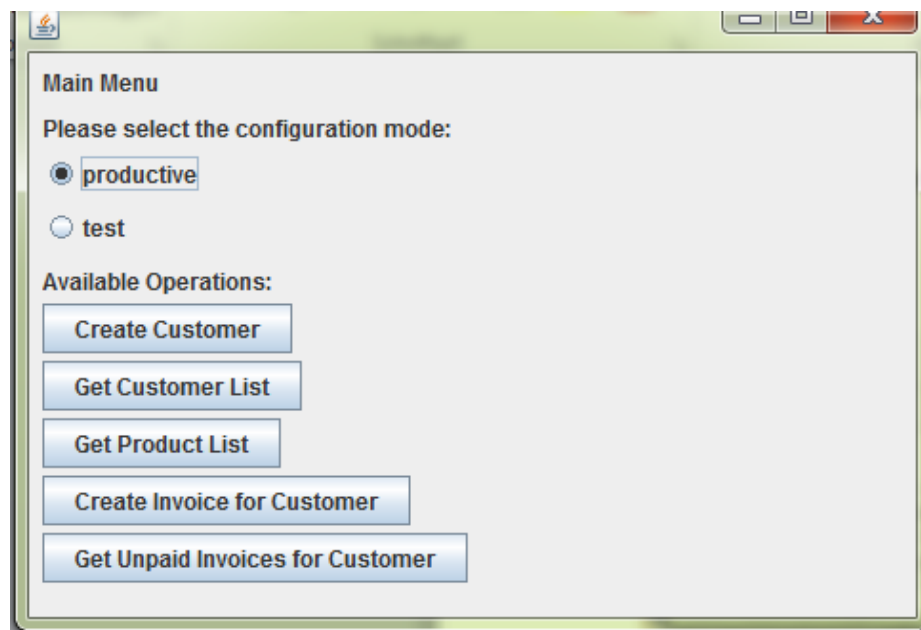
// Listener fuer test radiobutton
rdbtnTest.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        btnCreateCustomer.setEnabled(true);
        btnGetCustomerList.setEnabled(true);
        btnGetProductList.setEnabled(true);
        btnCreateInvoiceFor.setEnabled(true);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

Im **Code-Beispiel 4** wird gezeigt, dass zunächst der Konfigurationsmodus (configuration mode) vom Benutzer ausgewählt werden muss. Es sind zunächst alle Buttons, die sich untenstehend befinden, auf `setEnabled(false)` gesetzt, damit der Benutzer nicht die Auswahl des Modus übersieht, und eine Fehlersituation entsteht.

Hier die Ansicht im GUI, wenn der Configuration Mode noch nicht ausgewählt ist:

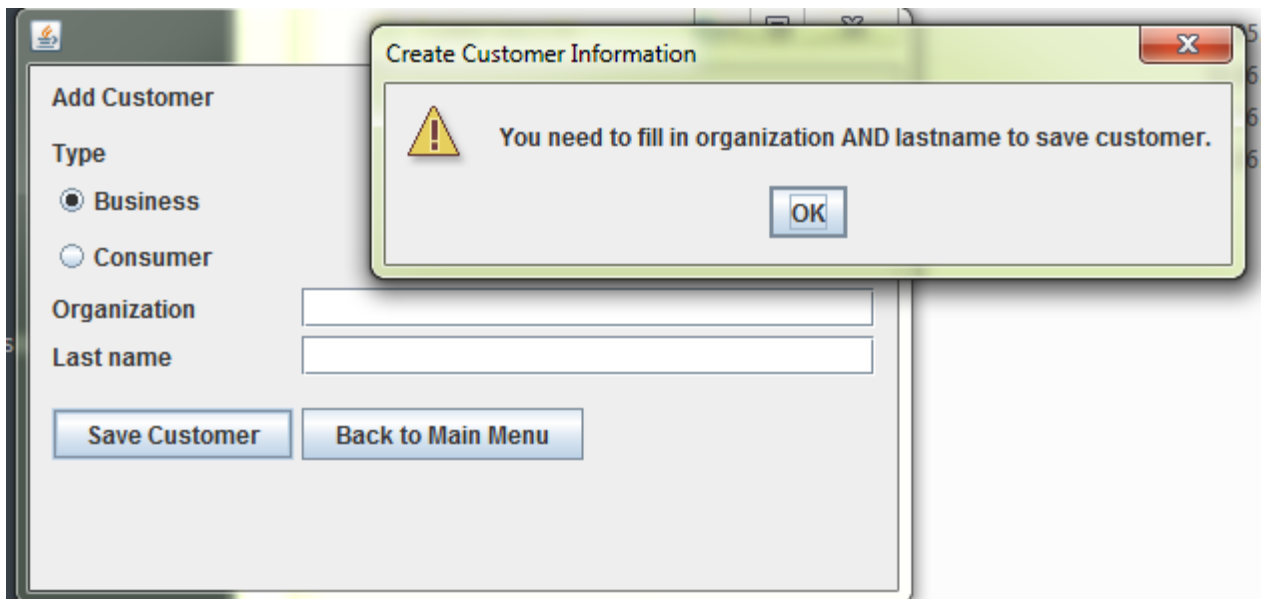


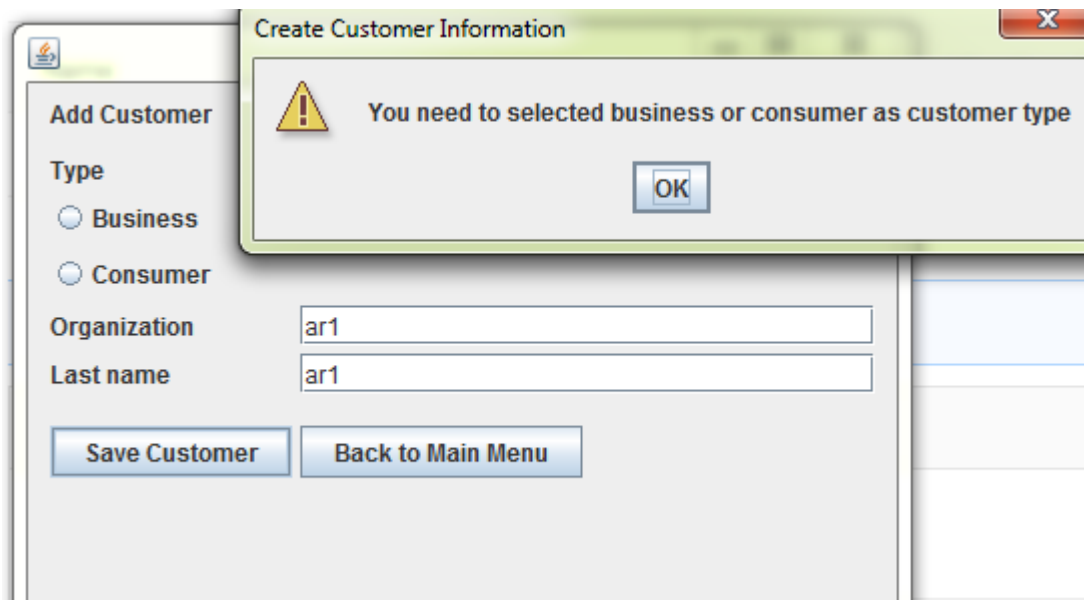
Sobald aber auf einen der radio buttons geklickt wird, werden die Buttons „aktiviert“, d.h. auf `setEnabled(true)` gesetzt, da der Configuration Mode ausgewählt wurde. Daher sind nun die Möglichkeiten der Operationsauswahl im GUI vorhanden und freigeschaltet:



Code-Beispiel 5:

```
btnSaveCustomer.addActionListener(new ActionListener() {  
  
    /**  
     * Bei Klicken auf den Button "Save Customer" werden die eingegebenen  
     * Informationen des neuen Customer, sofern die Benutzereingaben korrekt  
     * waren, gespeichert.  
     * Falls ein Fehler aufgetreten ist, wird eine entsprechende Meldung angezeigt.  
     * @param ActionEvent e  
     */  
    public void actionPerformed(ActionEvent e) {  
  
        if(!rdbtnBusiness.isSelected() && !rdbtnConsumer.isSelected()) && (textField.getText().equals("") || textField_1.  
            JOptionPane.showMessageDialog(frame,  
                "Empty fields are not allowed.",  
                "Create Customer Information",  
                JOptionPane.WARNING_MESSAGE);  
        } else if(!rdbtnBusiness.isSelected() && !rdbtnConsumer.isSelected()) && (!textField.getText().equals("") && !text  
            JOptionPane.showMessageDialog(frame,  
                "You need to selected business or consumer as customer type",  
                "Create Customer Information",  
                JOptionPane.WARNING_MESSAGE);  
        } else if(textField.getText().equals("") || textField_1.getText().equals("")){  
            JOptionPane.showMessageDialog(frame,  
                "You need to fill in organization AND lastname to save customer.",  
                "Create Customer Information",  
                JOptionPane.WARNING_MESSAGE);  
        } else {  
  
            String customerType = "";  
            if(rdbtnBusiness.isSelected()){  
                System.out.println("business");  
                customerType="business";  
            }  
        }  
    }  
});
```

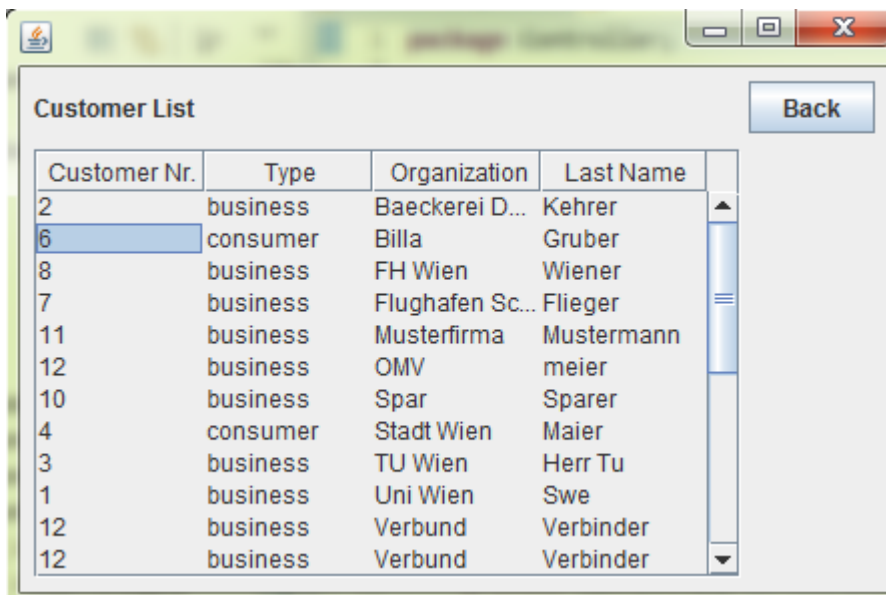




Im **Code-Beispiel 5** wird ein Ausschnitt der Methode `actionPerformed` des Button Listener für „Save Customer“ gezeigt. Es wird hier versucht, fehlerhafte Benutzereingaben schon abzufangen, bevor die Aktion des Speicherns überhaupt angefragt wird in der Application. Die Eingabebeschränkungen sind also dazu da, die Eingaben auf Korrektheit zu prüfen und mögliche Fehlersituationen abzuwenden. Wie man an den oben stehenden GUI-Screenshots sieht, wird dem Benutzer in der Application im Fall einer falschen Eingabe über ein Dialogfenster angezeigt, welcher Fehler aufgetreten ist. Der Zustand ist nach der fehlerhaften Eingabe genau derselbe wie davor. Denn die Anfrage des Benutzers wird abgebrochen und er kann erneut eine Eingabe versuchen.

Code-Beispiel 6:

```
public void setTheDataInTheTable(Object data[][], Object title[]){  
    model = new DefaultTableModel(data, title){  
        private static final long serialVersionUID = 1L;  
        public boolean isCellEditable(int rowIndex, int columnIndex) {  
            return false;  
        }  
    }  
}
```



Customer Nr.	Type	Organization	Last Name
2	business	Baeckerei D...	Kehrer
6	consumer	Billa	Gruber
8	business	FH Wien	Wiener
7	business	Flughafen Sc...	Flieger
11	business	Musterfirma	Mustermann
12	business	OMV	meier
10	business	Spar	Sparer
4	consumer	Stadt Wien	Maier
3	business	TU Wien	Herr Tu
1	business	Uni Wien	Swe
12	business	Verbund	Verbinder
12	business	Verbund	Verbinder

Im **Code-Beispiel 6** wird ein Ausschnitt der Methode `saveTheDataInTheTable` in der Klasse (Application Window) `ShowCustomerList` gezeigt. Hier werden mittels der Methode `isCellEditable` die Zellen des `JTable` auf nicht-editierbar gesetzt, damit der Benutzer, falls er die Tabelle anklickt, keine ungewünschten Datenänderungen vornehmen kann, mit denen die Application womöglich in Fehlersituation gebracht werden würde. Die Möglichkeit des Ändern der Zellen wird hier also bewusst nicht angeboten, denn es soll nur eine Anzeige erfolgen. Der GUI-Screenshot zeigt die Visualisierung der Code-Stelle.

Im Projekt wurden auch **Debugging-Hilfen** verwendet, damit das Verhalten der betrachteten Methoden während dem Programmieren genauer analysiert werden kann. Diese Debugging-Hilfen werden, nachdem die Methode fertiggestellt wurde, entweder als Kommentar erhalten oder ganz entfernt aus dem Quellcode. Denn sie dienen nur dazu, damit der Programmierer überprüfen kann, ob sich die Methoden korrekt verhalten und ob das Programm das gewünschte Verhalten zeigt. Fehler können so auch leichter entdeckt werden und die verschiedenen Fälle der Eingaben und Ähnliches können durchgespielt werden.

Code-Beispiel 7:

```
// TEST MAIN:
/*
public static void main (String args[]) {
    FastBillConnection fconn = new FastBillConnection();
    //Customer newCustomer=new Customer(0,"business","000MV","onu");
    // fconn.createCustomer(newCustomer);

    for(int i=0;i<clist.size();i++){
        System.out.println(clist.get(i).getId());
        System.out.println(clist.get(i).getCustomerType());
        System.out.println(clist.get(i).getOrganization());
        System.out.println(clist.get(i).getLastname());
    }

    ArrayList<Product> plist = fconn.getProductList();
    System.out.println("Size: " + plist.get(0).getName());
    for(int i=0;i<plist.size();i++){
        System.out.println(plist.get(i).getProductId());
        System.out.println(plist.get(i).getName());
        System.out.println(plist.get(i).getDescription());
        System.out.println(plist.get(i).getVatPercent());
        System.out.println(plist.get(i).getPlan().getMonthlyRate());
    }

}
*/
```

Die im **Code-Beispiel 7** gezeigte Test-Main-Methode wurde in der Klasse FastBillConnection im Package Logic benützt, um die API Request und ihre Ausgaben zu überprüfen bzw. um die daraus erstellte Customer Liste zur überprüfen, ob sie inhaltlich korrekt ist und ob alle Daten darin aufgenommen werden. Diese Methode wurde zum Debuggen verwendet, wird aber in der fertigen Version der Application nicht mehr vorhanden sein, denn sie dient nur zu Testzwecken.

Code-Beispiel 8:

```
try {
    post = new HttpPost("https://my.fastbill.com/api/1.0/api.php");
    String encoding = DatatypeConverter.printBase64Binary("morri.kocht@gmail.com:2d60347a");

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

    // System.out.println("Executing request " + post.getRequestLine());
    HttpResponse response = httpClient.execute(post);

    // System.out.println("-----");
    // System.out.println(response.getStatusLine());
    // System.out.println(EntityUtils.toString(response.getEntity()));
    String jsonString = EntityUtils.toString(response.getEntity());
    JSONObject root = new JSONObject(jsonString);
    JSONObject jsonResponse = (JSONObject) root.get("RESPONSE");
    JSONArray customers = (JSONArray) jsonResponse.get("CUSTOMERS");

    for(int i = 0 ; i < customers.length(); i++){
        JSONObject customer = (JSONObject) customers.get(i);
        Customer thisCustomer = new Customer(customer.getInt("CUSTOMER_NUMBER"), cust
            customer.getString("ORGANIZATION"),customer.getString("LAST_NAME"));
        customerList.add(thisCustomer);
    }
}
```

Im **Code-Beispiel 8** wurden in der Methode `getCustomerList` in der Klasse `FastBillConnection` im Package `Logic` ebenfalls zusätzliche Ausgaben mit `System.out.println` auf die Console als Debugging-Hilfen verwendet, um die API Requests korrekt zu erstellen und zu testen, ob die Abfragen in gewünschter Weise funktionieren, und die richtigen Daten liefern.