AR3: How and to what extent have you considered defensive programming? Discuss and show examples from your code.

Unsere Applikation ermöglicht es Benutzern Pläne gewisser Produkte auszuwählen und zu abonnieren. Je größer und komplexer das Programm desto wichtiger die defensiv eingesetzten Programmiertechniken. Defensives Programmieren ermöglicht es, Fehleingaben abzufangen und zweckentsprechend zu reagieren. Assertions, try-catch-Blöcke und andere relevante Error-Handling-Techniken stehen für defensive Programmierung. Auch in diesem Projekt haben wir wieder einige Techniken angewendet. Zur Veranschaulichung präsentieren wir nun einige Beispiele dafür:

```
@Override
public Customer getCustomerByLocalId(String localId) {
        for (Customer c : storage) {
                if (c.getLocalId().equals(localId)) {
                        return c;
                }
        }
        throw new IllegalArgumentException("The User with localId=" + localId + " does not exist");
}
```

1.) <u>Klasse: CustomerGenerator, Methode: getCustomerByLocalId</u>

In diesem Abschnitt verwenden wir Exceptions. Wird ein Customer gesucht, aber nicht gefunden so wird eine Exception mit „The User with localId=….. does not exist“ geworfen. Diese Technik wird auch bei anderen Generator-Klassen angewendet. Somit wird gewährleistet, dass das Programm konsistent arbeitet.

```
try {
        File fXmlFile = new File(xmlFileName);
        System.out.println("configXMLFileName: " + fXmlFile.getAbsolutePath());

        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();
```

```
catch (Exception e) {
        e.printStackTrace();
        throw new IllegalArgumentException(e.getMessage());
}
```

2.) Klasse: BackOfficeSystem, Methode: initialize

Die Klasse BackOfficeSystem beinhaltet einige Error-Handling-Techniken. Hier sehen wir nun einen try-catch-Block. Sobald ein Fehler beim Initialisieren auftritt, wird eine Exception geworfen. Am Ende der Methode wird diese Exception wieder gefangen und ausgegeben. Dabei liefert die abgefangene Exception Informationen über den jeweiligen Fehler.

```
org.junit.Assert.assertEquals("settings", doc.getDocumentElement().getNodeName());
```

3.) Klasse: BackOfficeSystem, Methode: intialize

Hier sieht man nun eine Assertion. Damit kann auf logische Fehler überprüft werden. Wir gehen grundsätzlich davon aus, dass doc.getDocumentElement().getNodeName() immer „settings“ beinhaltet. Ist dies jedoch nicht der Fall, so wird ein Fehler gemeldet. Die Verwendung vonAssertions ist die günstigste, schnellste und effizienteste Möglichkeit, Fehler zu entdecken.

```
try {
        Invoice[] invoicesArr = sys.getInvoice(selectedCustomer);
        ArrayList invoicesToStringList = new ArrayList<String>();
        for (Invoice i : invoicesArr)
                invoicesToStringList.add("InvoiceId: " + i.getInvoiceId());

        String[] invoicesStringArr = (String[]) invoicesToStringList
                        .toArray(new String[invoicesToStringList.size()]);

        listForOpenInvoices.setListData(invoicesStringArr);
} catch (NullPointerException e) {
        infoLabelOpenInvoices.setText("Es wurden keine Daten gefunden");
} catch (IllegalArgumentException e) {
```

4.) Klasse: Application, Methode: initialize

Auch diese Klasse beinhaltet wieder try-catch-Blöcke. Möglicherweise werden keine Daten gefunden oder es treten anderweitige Fehler auf. Ist dies der Fall, so wird eine Exception geworfen. Ist beispielsweise keine entsprechende Rechnung vorhanden und wird eine NullPointerException geworfen, so wird sie auch entsprechend aufgefangen. Und infolgedessen wird folgender Satz „Es wurden keine Daten gefunden“ an den User retourniert.

5.) Resümee

Bei diesem Projekt ging es in erster Linie darum, etwaige Pattern zu verstehen und anzuwenden. In vielen Fällen haben wir zwar defensive Programmierung angewendet, aber nicht immer und nicht überall haben wir davon Gebrauch gemacht. Natürlich gibt es noch Luft nach oben, aber im Grund genommen ist es nur eine Frage der Zeit und Erfahrung bis wir unser defensives Programmierverhalten perfektionieren.